Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism

Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas
University of Illinois at Urbana-Champaign
{skarlat2,kokolis2,tyxu,torrella}@illinois.edu

Abstract

The unprecedented growth in the memory needs of emerging memory-intensive workloads has made virtual memory translation a major performance bottleneck. To address this problem, this paper introduces *Elastic Cuckoo Page Tables*, a novel page table design that transforms the sequential pointer-chasing operation used by conventional multi-level radix page tables into fully-parallel look-ups. The resulting design harvests, for the first time, the benefits of memory-level parallelism for address translation. Elastic cuckoo page tables use Elastic Cuckoo Hashing, a novel extension of cuckoo hashing that supports efficient page table resizing. Elastic cuckoo page tables efficiently resolve hash collisions, provide process-private page tables, support multiple page sizes and page sharing among processes, and dynamically adapt page table sizes to meet application requirements.

We evaluate elastic cuckoo page tables with full-system simulations of an 8-core processor using a set of graph analytics, bioinformatics, HPC, and system workloads. Elastic cuckoo page tables reduce the address translation overhead by an average of 41% over conventional radix page tables. The result is a 3–18% speed-up in application execution.

CCS Concepts. **• Software and its engineering → Operating systems**; **Virtual memory**; **• Computer systems organization → Architectures**.

Keywords. Virtual Memory, Page Tables, Cuckoo Hashing

ACM Reference Format:
Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378493

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378493

1 Introduction

Virtual memory is a cornerstone abstraction of modern computing systems, as it provides memory virtualization and process isolation. A central component of virtual memory is the page table, which stores the virtual-to-physical memory translations. The design of the page table has a significant impact on the performance of memory-intensive workloads, where working sets far exceed the TLB reach. In these workloads, frequent TLB misses require the fetching of the virtual-to-physical memory translations from the page table, placing page table look-ups on the critical path of execution.

The *de facto* current design of the page table, known as *radix page table*, organizes the translations into a multi-level tree [4, 19, 38]. The x86-64 architecture uses a four-level tree, while a fifth level will appear in next-generation architectures such as Intel's Sunny Cove [36, 37], and is already supported by Linux [20]. A radix page table can incur a high performance overhead because looking-up a translation involves a page table walk that *sequentially* accesses potentially all the levels of the tree from the memory system.

Despite substantial efforts to increase address translation efficiency with large and multi-level TLBs, huge page support, and Memory Management Unit (MMU) caches for page table walks, address translation has become a major performance bottleneck. It can account for 20–50% of the overall execution time of emerging applications [8–10, 12–15, 21, 42, 56]. Further, page table walks may account for 20–40% of the main memory accesses [13]. Such overhead is likely to be exacerbated in the future, given that: (1) TLB scaling is limited by access time, space, and power budgets, (2) modern computing platforms can now be supplied with terabytes and even petabytes of main memory [30, 58], and (3) various memory-intensive workloads are rapidly emerging.

We argue that the radix page table was devised at a time of scarce memory resources that is now over. Further, its sequential pointer-chasing operation misses an opportunity: it does not exploit the ample memory-level parallelism that current computing systems can support.

For these reasons, in this paper, we explore a fundamentally different solution to minimize the address translation overhead. Specifically, we explore a new page table structure that eliminates the pointer-chasing operation and uses parallelism for translation look-ups.

A natural approach would be to replace the radix page table with a *hashed page table*, which stores virtual-to-physical translations in a hash table [22, 33, 39, 40, 68]. Unfortunately, hashed page tables have been plagued by a number of problems. One major problem is the need to handle hash collisions. Existing solutions to deal with hash collisions in page tables, namely collision chaining [35] and open addressing [73], require sequential memory references to walk over the colliding entries with special OS support. Alternatively, to avoid collisions, the hash page table needs to be dynamically resized. However, such operation is very costly and, hence, proposed hashed page table solutions avoid it by using a large global page table shared by all processes. Unfortunately, a global hashed page table cannot support page sharing between processes or multiple page sizes without adding an extra translation level.

To solve the problems of hashed page tables, this paper presents a novel design called *Elastic Cuckoo Page Tables*. These page tables organize the address translations using *Elastic Cuckoo Hashing*, a novel extension of cuckoo hashing [50] designed to support gradual, dynamic resizing. Elastic cuckoo page tables efficiently resolve hash collisions, provide process-private page tables, support multiple page sizes and page sharing among processes, and efficiently adapt page table sizes dynamically to meet process demands. As a result, elastic cuckoo page tables transform the sequential pointer-chasing operation of traditional radix page tables into fully parallel look-ups, allowing address translation to exploit memory-level parallelism for the first time.

We evaluate elastic cuckoo page tables with full-system simulations of an 8-core processor running a set of graph analytics, bioinformatics, HPC, and system workloads. Elastic cuckoo page tables reduce the address translation overhead by an average of 41% over radix page tables. The result is a 3–18% speed-up in application execution.

2 Background

2.1 Radix Page Tables

All current architectures implement *radix page tables*, where the page table is organized in a multi-level radix tree. For example, Figure 1 shows the structure of the x86-64 page table. Given a 48-bit Virtual Address (VA), since the standard page size is 4KB, the lowest 12 bits are the page offset. The remaining 36 bits are divided into four 9-bit fields. Each field is used as an index into one of the four levels of the page table. Such levels are known as PGD (Page Global Directory), PUD (Page Upper Directory), PMD (Page Middle Directory), and PTE (Page Table Entry), respectively [19]. The translation starts with the CR3 register, which contains the base of the PGD table. By adding the CR3 and bits 47–39, one obtains a PGD entry whose content is the base of the correct PUD table. Then, by adding such content and bits 38–30, one obtains a PUD entry whose content is the base of the correct PMD table. The process continues until a PTE entry is read. It contains the physical page number and additional flags that the hardware inserts in the TLB. The physical page number concatenated with bits 11–0 is the Physical Address (PA).

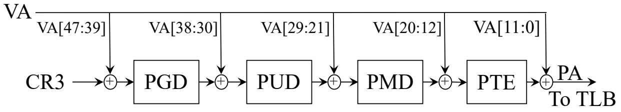


Figure 1. Virtual to physical address translation in x86-64.

The process described is called a *page table walk*. It is performed in hardware on a TLB miss. A page table walk requires four *sequential* cache hierarchy accesses.

To increase the reach of the TLB, the x86-64 architecture supports two large page sizes, 2MB and 1GB. When a large page is used, the page table walk is shortened. Specifically, a 2MB page translation is obtained from the PMD table, while a 1GB page translation is obtained from the PUD table.

To alleviate the overhead of page table walks, the MMU of an x86-64 processor has small caches called Page Walk Caches (PWCs). The PWCs store recently-accessed PGD, PUD, and PMD table entries (but not PTE entries) [1, 8, 11, 12, 38]. On a TLB miss, before the hardware issues any request to the cache hierarchy, it checks the PWCs. It records the lowest level table at which it hits. Then, it generates an access to the cache hierarchy for the next lower level table.

2.1.1 Struggles with Emerging Workloads. Emerging workloads, e.g., in graph processing and bioinformatics, often have multi-gigabyte memory footprints and exhibit low-locality memory access patterns. Such behavior puts pressure on the address translation mechanism. Recent studies have reported that address translation has become a major performance bottleneck [8–10, 12–15, 21, 42, 56]. It can consume 20–50% of the overall application execution time. Further, page table walks may account for 20–40% of the main memory accesses [13].

To address this problem, one could increase the size of the PWCs to capture more translations, or increase the number of levels in the translation tree to increase the memory addressing capabilities. Sadly, neither approach is scalable. Like for other structures close to the core such as the TLB, the PWCs' access time needs to be short. Hence, the PWCs have to be small. They can hardly catch up with the rapid growth of memory capacity.

Increasing the number of levels in the translation tree makes the translation slower, as it may involve more cache hierarchy accesses. Historically, Intel has gradually increased the depth of the tree, going from two in the Intel 80386 to four in current processors [4, 38]. A five-level tree is planned for the upcoming Intel Sunny Cove [36, 37], and has been implemented in Linux [20]. This approach is not scalable.

2.2 Hashed Page Tables

The alternative to radix page tables is *hashed page tables*. Here, address translation involves hashing the virtual page number and using the hash key to index the page table. Assuming that there is no hash collision, only one memory system access is needed for address translation.

Hashed page tables [22, 33, 39, 40, 68] have been implemented in the IBM PowerPC, HP PA-RISC, and Intel Itanium architectures. The support for a hashed page table in the Itanium architecture was referred to as the long-format Virtual Hash Page Table (VHPT) [23, 28, 35]. In that design, the OS handles hash collisions. Upon a hash collision, the VHPT walker raises an exception that invokes the OS. The OS handler resolves the collision by searching collision chains and other auxiliary OS-defined data structures [8].

2.2.1 Challenges in Hashed Page Tables.

Barr et al. [8] summarize three limitations of hashed page tables. The first one is the loss of spatial locality in the accesses to the page table. This is caused by hashing, which scatters the page table entries of contiguous virtual pages. The second limitation is the need to associate a hash tag (e.g., the virtual page number) with each page table entry, which causes page table entries to consume more memory space. The third one is the need to handle hash collisions, which leads to more memory accesses, as the system walks collision chains [8].

Yaniv and Tsafrir [73] recently show that the first two limitations are addressable by careful design of page table entries. Specifically, they use Page Table Entry Clustering, where multiple contiguous page table entries are placed together in a single hash table entry that has a size equal to a cache line. Further, they propose Page Table Entry Compaction, where unused upper bits of multiple contiguous page table entries are re-purposed to store the hash tag.

Unfortunately, hash collisions are a significant concern and remain unsolved. Existing strategies such as collision chaining [35] and open addressing [73] require expensive memory references needed to walk over the colliding entries.

To assess the importance of collisions, we take the applications of Section 7 and model a global hash table. We evaluate the following scenario: (1) the table has as many entries as the sum of all the translations required by all the applications, and (2) the hash function is the computationally-expensive BLAKE cryptographic function [5] which minimizes the probability of collisions.

Figure 2 shows the probability of random numbers mapping to the same hash table entry. The data is shown as a cumulative distribution function (CDF). The figure also shows the CDF for a global hash table that is over-provisioned by 50%. For the baseline table, we see that only 35% of the entries in the hash table have no collision (i.e, the number of colliding entries is 1). On the other hand, there are entries with a high number of collisions, which require time-consuming collision resolution operations. Even for the over-provisioned table, only half of the entries in the table have no collision.

2.2.2 Drawbacks of a Single Global Hash Table.

One straightforward design for a hashed page table system is to have a single global hash table that includes page table entries from all the active processes in the machine. This design is attractive because 1) the hash table is allocated only once, and 2) the table can be sized to minimize the need for dynamic table resizing, which is very time consuming.

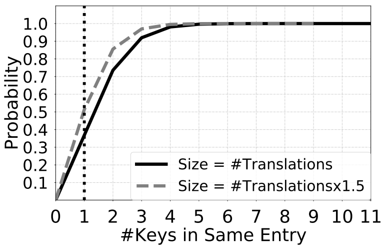


Figure 2. Cumulative distribution function of the number of keys mapping to the same hash table entry.

Sadly, such design has a number of practical drawbacks that make it undesirable [22, 73]. First, neither multiple page sizes (e.g., huge pages) nor page sharing between processes can be supported without additional complexity. For example, to support these two features, the IBM PowerPC architecture uses a two-level translation procedure for each memory reference [34]. Second, when a process is killed, the system needs to perform a linear scan of the entire hash table to find and delete the associated page table entries. Note that deleting an entry may also be 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.

2.2.3 Resizing Hashed Page Tables.

To reduce collisions, hash table implementations set an *occupancy threshold* that, when reached, triggers the resizing of the table. However, resizing is an expensive procedure if done all at once. It requires allocating a new larger hash table and then, for each entry in the old hash table, rehash the tag with the new hash function and move the (tag, value) pair to the new hash table. In the context of page tables, the workload executing in the machine needs to pause and wait for the resizing procedure to complete. Further, since the page table entries are moved to new memory locations, their old copies cached in the cache hierarchy of the processor become useless. The correct copies are now in new addresses.

An alternative approach is to *gradually* move the entries, and maintain both the old and the new hash tables in memory for a period of time. Insertions place entries in the new hash table only, so the old hash table will eventually become empty and will then be deallocated. Further, after each insertion, the system also moves one or more entries from the old table to the new one. Unfortunately, a look-up needs to access both the old and the new hash tables, since the desired entry could be present in either of them.

In the context of page tables, gradual rehashing has two limitations. First, keeping both tables approximately doubles the memory overhead. Second, a look-up has to fetch entries from both tables, doubling the accesses to the cache hierarchy. Unfortunately, half of the fetched entries are useless and, therefore, the caches may get polluted.

2.3 Cuckoo Hashing

Cuckoo hashing is a collision resolution algorithm that allows an element to have multiple possible hashing locations [50]. The element is stored in at most one of these locations at a time, but it can move between its hashing locations. Assume a cuckoo hash table with two hash tables or ways T_1 and T_2, indexed with hash functions H_1 and H_2, respectively. Because there are two tables and hash functions, the structure is called a *2-ary* cuckoo hash table. Insertion in cuckoo hashing places an element x in one of its two possible entries, namely $T_1[H_1(x)]$ or $T_2[H_2(x)]$. If the selected entry is occupied, the algorithm kicks out the current occupant y, and re-inserts y in y's other hashing location. If that entry is occupied, the same procedure is followed for its occupant. The insertion and eviction operations continue until no occupant is evicted or until a maximum number of displacements is reached (e.g., 32). The latter case is an insertion failure.

A look-up in cuckoo hashing checks all the possible hashing locations of an element, and succeeds if it is found in either of them. In the example above, $T_1[H_1(x)]$ and $T_2[H_2(x)]$ are checked. The locations are checked *in parallel*. Hence, a look-up takes a constant time. A deletion operation proceeds like a look-up; then, if the element is found, it is removed.

Figure 3 shows an example of inserting element x into a 2-ary cuckoo hash table. Initially, in Step ①, the table has three elements: a, b, and c. The insertion of x at $T_1[H_1(x)]$ kicks out the previous occupant b. In Step ②, the algorithm inserts b in $T_2[H_2(b)]$, and kicks out c. Finally, in Step ③, a vacant entry is found for c. This example can be generalized to *d-ary* cuckoo hashing [24], which uses d independent hash functions to index d hash tables.

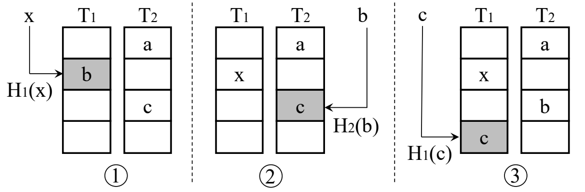


Figure 3. An example of insertion in cuckoo hashing.

Like any hash table, the performance of cuckoo hash tables deteriorates with high occupancy. To gain insight, Figure 4 characterizes a d-ary cuckoo hash table (where $d \in \{2, 3, 4, 8\}$) as a function of the occupancy. We take random numbers and, like before, hash them with the BLAKE cryptographic hash function [5]. The actual size of the cuckoo hash table does not matter, but only the hash table occupancy.

Figure 4(a) shows the average number of insertion attempts required to successfully insert an element as a function of the table occupancy. We see that, for low occupancy, we either insert the key on the first attempt or require a single displacement. As occupancy increases, insertion performance deteriorates — e.g., after 50% occupancy in the 2-ary cuckoo, and after ~70% in the 3, 4, and 8-ary cuckoo hash tables. Figure 4(b) shows the probability of insertion failures after 32 attempts. We see that the 2-ary cuckoo hash table has a non-zero probability of insertion failures after 50% occupancy, while the 3, 4, and 8-ary hash tables exhibit insertion failures only after 80% occupancy.

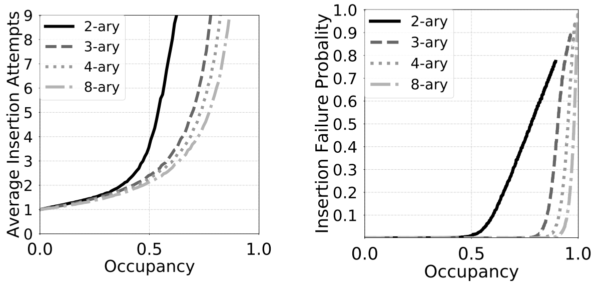


Figure 4. Characterizing d-ary cuckoo hashing.

3 Rethinking Page Tables

As indicated in Section 2.1.1, radix page tables are not scalable. Further, as shown in Section 2.2.2, having a single global hashed page table is not a good solution either. We want to provide *process-private hashed page tables*, so that we can easily support page sharing among processes and multiple page sizes. However, a default-sized hashed page table cannot be very large, lest we waste too much memory in some processes. Hence, we are forced to have modest-sized hashed page tables, which will suffer collisions.

One promising approach to deal with collisions is to use cuckoo hashing (Section 2.3). Unfortunately, any default-sized cuckoo hash table will eventually suffer insertion failures due to insufficient capacity. Hence, it will be inevitable to resize the cuckoo hash table.

Sadly, resizing cuckoo hash tables is especially expensive. Indeed, recall from Section 2.2.3 that, during gradual resizing, a look-up requires twice the number of accesses — since both the old and new hash tables need to be accessed. This requirement especially hurts cuckoo hashing because, during normal operation, a look-up into a d-ary cuckoo hash table already needs d accesses; hence, during resizing, a look-up needs to perform $2 \times d$ accesses.

To address this problem, in this paper, we extend cuckoo hashing with a new algorithm for gradual resizing. Using this algorithm, during the resizing of a d-ary cuckoo hash table, a look-up *only requires d accesses*. In addition, the algorithm never fetches into the cache page table entries from the old hash table whose contents have already moved to the new hash table. Hence, it minimizes cache pollution from such entries. We name the algorithm *Elastic Cuckoo Hashing*. With this idea, we later build per-process *Elastic Cuckoo Page Tables* as our proposed replacement for radix page tables.

4 Elastic Cuckoo Hashing

4.1 Intuitive Operation

Elastic cuckoo hashing is a novel algorithm for cost-effective gradual resizing of d-ary cuckoo hash tables. It addresses the major limitations of existing gradual resizing schemes. To understand how it works, consider first how gradual resizing works with a baseline d-ary cuckoo hash table.

Cuckoo Hashing. Recall that the d-ary cuckoo hash table has d ways, each with its own hash function. We represent each way and hash function as T_i and H_i, respectively, where $i \in 1..d$. We represent the combined ways and functions as T_D and H_D, respectively. When the occupancy of T_D reaches a *Rehashing Threshold*, a larger d-ary cuckoo hash table is allocated. In this new d-ary table, we represent each way and hash function as T'_i and H'_i, respectively, where $i \in 1..d$, and the combined ways and functions as T'_D and H'_D, respectively.

As execution continues, a collision-free insert operation only accesses a randomly-selected single way of the new d-ary hash table — if there are collisions, multiple ways of the new d-ary hash table are accessed. In addition, after every insert, the system performs one *rehash* operation. A rehash consists of removing one element from the old d-ary hash table and inserting it into the new d-ary hash table. Unfortunately, a look-up operation requires probing all d ways of both the old and the new d-ary hash tables, since an element may reside in any of the d ways of the two hash tables. When all the elements have been removed from the old d-ary table, the latter is deallocated.

Elastic Cuckoo Hashing. A d-ary *elastic* cuckoo hash table works differently. Each T_i way in the old d-ary hash table has a *Rehashing Pointer* P_i, where $i \in 1..d$. The set of P_i pointers is referred to P_D. At every T_i, P_i is initially zero. When the system wants to rehash an element from T_i, it removes the element pointed to by P_i, inserts the element into the new d-ary table, and increments P_i. At any time, P_i divides its T_i into two regions: the entries at lower indices than P_i (*Migrated Region*) and those at equal or higher indices than P_i (*Live Region*). Figure 5 shows the two regions for a 2-ary elastic cuckoo hash table. As gradual rehashing proceeds, the migrated regions in T_D keep growing. Eventually, when the migrated regions cover all the entries in T_D, the old d-ary table is deallocated.

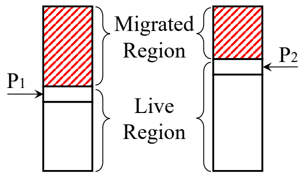


Figure 5. 2-ary elastic cuckoo hash table during resizing.

The insertion of an element in a d-ary elastic cuckoo hash table proceeds as follows. The system randomly picks one way from the old d-ary table, say T_i. The element is hashed with H_i. If the hash value falls in the live region of T_i, the element is inserted in T_i; otherwise, the element is hashed with the hash function H'_i of the *same way* T'_i of the *new* d-ary table, and the element is inserted in T'_i.

Thanks to this algorithm, a look-up operation for an element only requires d probes. Indeed, the element is hashed using all H_D hash functions in the old d-ary table. For each way i, if the hash value falls in the live region of T_i, T_i is probed; otherwise, the element is hashed with H'_i, and T'_i in the new d-ary table is probed.

Elastic cuckoo hashing improves gradual resizing over cuckoo hashing in two ways. First, it performs a look-up with only d probes rather than $2 \times d$ probes. Second, it minimizes cache pollution by never fetching entries from the migrated regions of the old d-ary table; such entries are useless, as they have already been moved to the new d-ary table.

4.2 Detailed Algorithms

We now describe the elastic cuckoo algorithms in detail.

Rehash. A rehash takes the element pointed to by the rehashing pointer P_i of way T_i of the old d-ary hash table, and uses the hash function H'_i to insert it in the *same* way T'_i of the new d-ary table. P_i is then incremented.

Figure 6 shows an example for a 2-ary elastic cuckoo hash table. On the left, we see a hash table before rehashing, with P_1 and P_2 pointing to the topmost entries. On the right, we see the old and new hash tables after the first entry of T_1 has been rehashed. The system has moved element d from T_1 to T'_1 at position $T'_1[H'_1(d)]$.

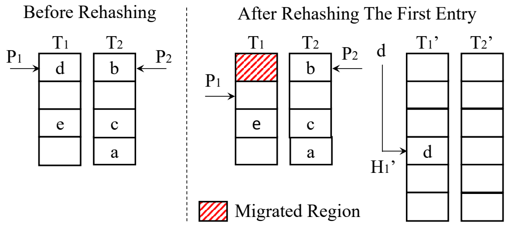


Figure 6. Example of the rehash operation.

Look-up. The look-up of an element x involves computing $H_i(x)$ for all the ways of the old d-ary hash table and, for each way, comparing the result to P_i. For each way i, if $H_i(x)$ belongs to the live region (i.e., $H_i(x) \geq P_i$), way T_i in the old hash table is probed with $H_i(x)$; otherwise, way T'_i in the new hash table is probed with $H'_i(x)$. The algorithm is:

```
function LOOK-UP(x):
    for each way $i$ in the old $d$-ary hash table do:
        if $H_i(x) < P_i$ then:
            if $T'_i[H'_i(x)] == x$ then return true;
        else:
            if $T_i[H_i(x)] == x$ then return true;
    return false
```

As an example, consider 2-ary elastic cuckoo hash tables. A look-up of element x involves two probes. Which structure is probed depends on the values of P_1, P_2, $H_1(x)$, and $H_2(x)$. Figure 7 shows the four possible cases. The figure assumes that P_1 and P_2 currently point to the third and second entry of T_1 and T_2. If $H_1(x) \geq P_1 \land H_2(x) \geq P_2$ (Case ①), entries $T_1[H_1(x)]$ and $T_2[H_2(x)]$ are probed. If $H_1(x) < P_1 \land H_2(x) < P_2$ (Case ②), entries $T'_1[H'_1(x)]$ and $T'_2[H'_2(x)]$ are probed. If $H_1(x) < P_1 \land H_2(x) \geq P_2$ (Case ③), entries $T'_1[H'_1(x)]$ and $T_2[H_2(x)]$ are probed. Finally, if $H_1(x) \geq P_1 \land H_2(x) < P_2$ (Case ④), $T_1[H_1(x)]$ and $T'_2[H'_2(x)]$ are probed.

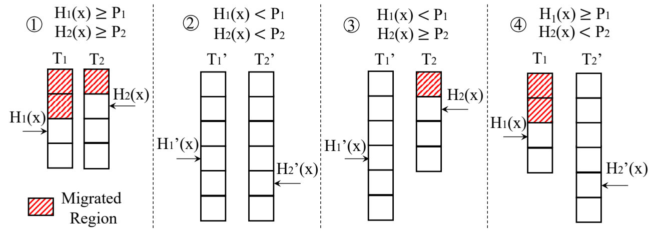


Figure 7. Different cases of the look-up operation.

Observation 1. Given the parallelism in a look-up operation, a look-up during resizing takes the time needed to perform a hash of an element, a comparison to the value in the rehashing pointer, and either a probe in one way of the old hash table or a second hash and a probe in one way of the new hash table.

Delete. A delete follows the look-up procedure and, on finding the element, clears the entry. Therefore, it takes the same time as a look-up plus a write to the target way.

Insert. The insert of an element x involves randomly picking one way i in the old d-ary hash table and checking if $H_i(x) < P_i$ is true. If it is not, the element is inserted at $T_i[H_i(x)]$; otherwise, the element is inserted in the same way of the new d-ary hash table at $T'_i[H'_i(x)]$. In either case, if the insertion causes the eviction of another element y, the system randomly picks a different way than the one just updated, and repeats the same procedure for y. This process may repeat multiple times, every time picking a different way than *the immediately previous* one. It terminates when either an insertion does not cause any eviction or a maximum number of iterations is reached.

The following algorithm describes the insert operation. In the algorithm, RAND_PICK returns a random way from a set of ways. We use $x \leftrightarrow y$ to denote the swapping of the values x and y, and use $\perp$ to denote an empty value.

```
function INSERT(x):
    $i \leftarrow$ RAND_PICK($\{1, \cdots, d\}$);
    for loop = 1 to MAX_ATTEMPTS do:
        if $H_i(x) < P_i$ then:
            $x \leftrightarrow T'_i[H'_i(x)]$;
            if $x == \perp$ then return true;
        else:
            $x \leftrightarrow T_i[H_i(x)]$;
            if $x == \perp$ then return true;
        $i \leftarrow$ RAND_PICK($\{1, \cdots, d\} - \{i\}$);
    return false
```

Hash Table Resize. Two parameters related to elastic cuckoo hash table resizing are the *Rehashing Threshold* (r_t) and the *Multiplicative Factor* (k). When the fraction of a hash table that is occupied reaches r_t, the system triggers a hash table resize, and a new hash table that is k times bigger than the old one is allocated.

We select a r_t that results in few insertion collisions and a negligible number of insertion failures. As shown in Figure 4, for 3-ary tables, a good r_t is 0.6 or less. We select a k that is neither so large that it wastes substantial memory nor so small that it causes continuous resizes. Indeed, if k is too small, as entries are moved into the new hash table during resizing, the occupancy of the new hash table may reach the point where it triggers a new resize operation.

Appendix A shows how to set k. It shows that $k > (r_t + 1)/r_t$. For example, for $r_t = 0.4$, $k > 3.5$; hence $k = 4$ is good. For $r_t = 0.6$, $k > 2.6$; hence $k = 3$ is good. However, a k equal to a power of two is best for hardware simplicity.

To minimize collisions in the old hash table during resizing, it is important to limit the occupancy of the live region during resizing. Our algorithm tracks the fraction of used entries in the live region. As long as such fraction does not exceed the one for the whole table that triggered the resizing, each insert is followed by only a single rehash. Otherwise, each insert is followed by the rehashing of multiple elements until the live region falls back to the desired fraction of used entries.

We select the Rehashing Threshold to be low enough that the frequency of insertion failures is negligible. However, insertion failures can still occur. If an insertion failure occurs outside a resize, we initiate resizing. If an insertion failure occurs during a resize, multiple elements are rehashed into other ways and then the insertion is tried again. In practice, as we show in Section 8.1, by selecting a reasonable Rehashing Threshold, we completely avoid insertion failures.

Elastic cuckoo hash tables naturally support downsizing when the occupancy of the tables falls below a given threshold. We use a *Downsizing Threshold* (d_t) and reduce the table by a *Downsizing Factor* (g). For gradual downsizing, we use an algorithm similar to gradual resizing.

5 Elastic Cuckoo Page Table Design

Elastic cuckoo page tables are process-private hashed page tables that scale on demand according to the memory requirements of the process. They resolve hash collisions and support multiple page sizes and page sharing among processes. In this section, we describe their organization, the cuckoo walk tables, and the cuckoo walk caches.

5.1 Elastic Cuckoo Page Table Organization

An elastic cuckoo page table is organized as a d-ary elastic cuckoo hash table that is indexed by hashing a Virtual Page Number (VPN) tag. A process in a core has as many elastic cuckoo page tables as page sizes. Figure 8 shows the elastic cuckoo page tables of a process for the page sizes supported by x86: 1GB, 2MB, and 4KB. In the figure, each page table uses a 2-ary elastic cuckoo hash table. The tables are named using x86 terminology: PUD, PMD, and PTE. Each table entry contains multiple, consecutive page translation entries. In the example, we assume 8 of them per table entry. Hence, using an x86-inspired implementation, the tables are indexed by the following VA bits: 47-33 for PUD, 47-24 for PMD, and 47-15 for PTE. All the hash functions are different.

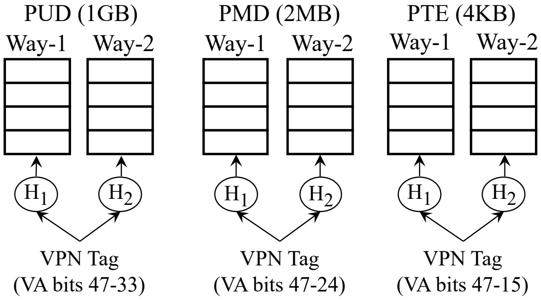


Figure 8. Elastic cuckoo page tables for a process.

By using multiple independent page tables, this design can support any page sizes. This is contrast to radix page tables which, due to the way they perform the translation, can only support a few rigid page sizes.

In addition, this design can attain high performance by exploiting two levels of parallelism. The first one is between tables of different page sizes. Each elastic cuckoo page table is independent from the others and can be looked-up in parallel. This is in contrast to radix page tables, where each tree level has to be walked sequentially. The second level of parallelism is between the different d ways of a d-ary table. A translation may reside in any of the d ways. Hence, all ways can be accessed in parallel. Overall, virtual page translation using elastic cuckoo page tables can exploit the memory-level parallelism provided by modern processors.

5.1.1 Page Table Entry. The entries in an elastic cuckoo page table use the ideas of page table entry clustering and compaction [68, 73]. The goal is to improve spatial locality and to reduce the VPN tag overhead. Specifically, a single hash table entry contains a VPN tag and multiple consecutive physical page translation entries packed together. The number of such entries packed together is called the clustering factor, and is selected to make the tag and entries fit in one cache line.

In machines with 64-byte cache lines, we can cluster eight physical page translation entries and a tag in a cache line. This is feasible with the compaction scheme proposed in [73], which re-purposes some unused bits from the multiple contiguous translations to encode the tag [38, 73].

As an example, consider placing in a cache line eight PTE entries of 4KB pages, which require the longest VPN tag — i.e, the 33 bits corresponding to bits 47–15 of the VA. In an x86 system, a PTE typically uses 64 bits. To obtain these 33 bits for the tag, we need to take 5 bits from each PTE and re-purpose them as tag. From each PTE, our implementation takes 4 bits that the Linux kernel currently sets aside for the software to support experimental uses [44], and 1 bit that is currently used to record the page size — i.e., whether the page size is 4KB or more. The latter information is unnecessary in elastic cuckoo page tables. With these choices, we place eight PTE entries and a tag in a cache line. We can easily do the same for the PMD and PUD tables, since we only need to take 3 and 2 bits, respectively, from each physical page translation entry in these hash tables.

5.1.2 Cuckoo Walk. We use the term *Cuckoo Walk* to refer to the procedure of finding the correct translation in elastic cuckoo page tables. A cuckoo walk fundamentally differs from the sequential radix page table walk: it is a *parallel* walk that may look-up multiple hash tables in parallel. To perform a cuckoo walk, the hardware page table walker takes a VPN tag, hashes it using the hash functions of the different hash tables, and then uses the resulting keys to index multiple hash tables in parallel.

As an example, assume that we have a 2-ary PTE elastic cuckoo page table. Figure 9 illustrates the translation process starting from a VPN tag. Since the clustering factor is 8, the VPN tag is bits 47–15 of the VA. These bits are hashed using the two hash functions H_1 and H_2. The resulting values are then added to the physical addresses of the bases of the two hash tables. Such bases are stored in control registers. To follow x86 terminology, we call these registers `CR3-PageSize`$_j$`-way`$_i$. The resulting physical addresses are accessed and a hit is declared if any of the two tags match the VPN tag. On a hit, the PTE Offset (bits 14-12) is used to index the hash table entry and obtain the desired PTE entry.

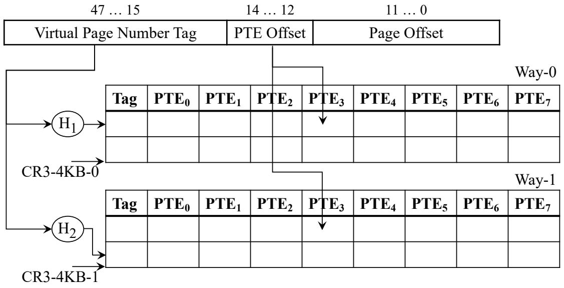


Figure 9. Accessing a 2-ary PTE elastic cuckoo page table.

Assume that the system supports S different page sizes. In a translation, if the size of the page requested is unknown, the hardware has to potentially look-up the d ways of each of the S elastic cuckoo page tables. Hence, a cuckoo walk may need to perform up to $S \times d$ parallel look-ups. In the next section, we show how to substantially reduce this number.

Similarly to radix page table entries, the entries of elastic cuckoo page tables are cached on demand in the cache hierarchy. Such support accelerates the translation process, as it reduces the number of requests sent to main memory.

5.2 Cuckoo Walk Tables

We do not want cuckoo page walks to have to perform $S \times d$ parallel look-ups to obtain a page translation entry. To reduce the number of look-ups required, we introduce the *Cuckoo Walk Tables* (CWTs). These software tables contain information about which way of which elastic cuckoo page table should be accessed to obtain the desired page translation entry. The CWTs are updated by the OS when the OS performs certain types of updates to the elastic cuckoo page tables (Section 5.2.1). The CWTs are automatically read by the hardware, and effectively prune the number of parallel look-ups required to obtain a translation.

Using the CWTs results in the four types of cuckoo walks shown in Figure 10. The figure assumes three page sizes (1GB, 2MB, and 4KB) and 2-ary elastic cuckoo page tables.

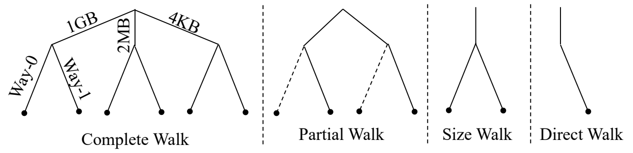


Figure 10. Different types of cuckoo walks.

Complete Walk: In this case, the CWTs provide no information and, hence, the hardware accesses all the ways of the elastic cuckoo page tables of all the page sizes.

Partial Walk: The CWTs indicate that the page is not of a given size. Hence, the hardware accesses potentially all the ways of the rest of the elastic cuckoo page tables. In some cases, the search of some of the ways may be avoided. This is represented in the figure with dashed lines.

Size Walk: The CWTs indicate that the page is of a given size. As a result, the hardware accesses all the ways of a single elastic cuckoo page table.

Direct Walk: The CWTs indicate the size of the page and which way stores the translation. In this case, only one way of a single elastic cuckoo page table is accessed.

Ideally, we have one CWT associated with each of the elastic cuckoo page tables. In our case, this means having a PUD-CWT, a PMD-CWT, and a PTE-CWT, which keep progressively finer-grain information. These tables are accessed in sequence, and each may provide more precise information than the previous one.

However, these software tables reside in memory. To make them accessible with low latency, they are cached in special caches in the MMU called the *Cuckoo Walk Caches.* These caches replace the page walk caches of radix page tables. They are described in Section 5.3. In our design, we find that caching the PTE-CWT would provide too little locality to be profitable — an observation consistent with the fact that the current page walk caches of radix page tables do not cache PTE entries. Hence, we only have PUD-CWT and PMD-CWT tables, and cache them in the cuckoo walk caches.

The PUD-CWT and PMD-CWT are updated by OS threads using locks. They are designed as d-ary elastic cuckoo hash tables like the page tables. We discuss the format of their entries next.

5.2.1 Cuckoo Walk Table Entries.

An entry in a CWT contains a VPN tag and several consecutive *Section Headers*, so that the whole CWT entry consumes a whole cache line. A section header provides information about a given Virtual Memory *Section.* A section is the range of virtual memory address space translated by one entry in the corresponding elastic cuckoo page table. A section header specifies the sizes of the pages in that section and which way in the elastic cuckoo page table holds the translations for that section.

To make this concept concrete, we show the exact format of the entries in the PMD-CWT. Figure 11 shows that an entry is composed of a VPN tag and 64 section headers. Each section header provides information about the virtual memory section mapped by one entry in the PMD elastic cuckoo page table. For example, Figure 11 shows a shaded section header which provides information about the virtual memory section mapped by the shaded entry in the PMD elastic cuckoo page table (shown in the bottom of the figure). Such entry, as outlined in Section 5.1.1, includes a VPN tag and 8 PMD translations to fill a whole cache line.

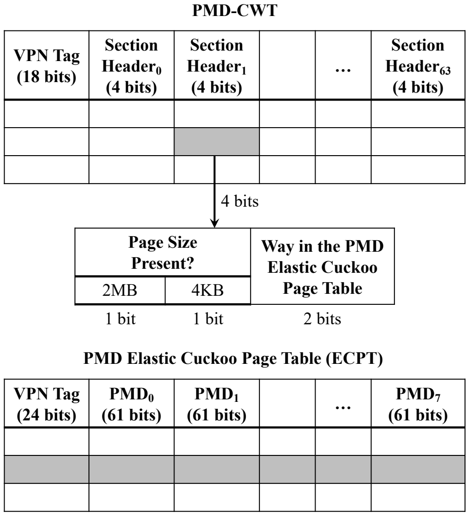

PMD-CWT
VPN Tag (18 bits)
Section Header$_0$ (4 bits)
Section Header$_1$ (4 bits)
...
Section Header$_{63}$ (4 bits)
4 bits
Page Size Present?
2MB
4KB
Way in the PMD Elastic Cuckoo Page Table
1 bit
1 bit
2 bits
PMD Elastic Cuckoo Page Table (ECPT)
VPN Tag (24 bits)
PMD$_0$ (61 bits)
PMD$_1$ (61 bits)
...
PMD$_7$ (61 bits)

Figure 11. PMD-CWT entry layout.

Since a PMD page is 2MB, and a virtual memory section (i.e., a row or entry in the PMD elastic cuckoo page table of Figure 11) may map up to 8 PMD pages, a virtual memory section comprises 16MB. Note that part or all of this virtual memory section may be populated by 4KB pages (whose translations are in the PTE elastic cuckoo page table) rather than by 2MB pages.

Then, a PMD-CWT row or entry provides information for 64 of these sections, which corresponds to a total of 1 GB. Hence, the VPN tag of a PMD-CWT entry contains bits 47-30 of the virtual address. Given a 64-byte line, this design allows at least 4 bits for each section header of a PMD-CWT entry — but we cannot have more section headers in a line.

These 4 bits encode the information shown in Figure 11. The first bit (*2MB Bit*) indicates whether this memory section maps one or more 2MB pages. The second bit (*4KB Bit*) says whether the section maps one or more 4KB pages. The last two bits (*Way Bits*) are only meaningful if the 2MB Bit is set. They indicate the way of the PMD elastic cuckoo page table that holds the mapped translations of the 2MB pages in this virtual memory section. Note this encoding assumes that the elastic cuckoo page tables have at most four ways.

Table 1 shows the actions taken by the hardware page walker when it reads the target section header from the PMD-CWT. If the 2MB and 4KB Bits are clear, no 2MB or 4KB page is mapped in this section. Therefore, the walker does not access the PMD or PTE elastic cuckoo page tables. If the 2MB Bit is set and the 4KB Bit is clear, the walker performs a *Direct Walk* in the PMD elastic cuckoo page table using the way indicated by the Way Bits. Only *one access* is needed because all the translation information for this section is present in a single entry of the PMD table.

Page Size Present?		Way in the PMD ECPT	Action
2MB	4KB		
0	0	X	No access to PMD ECPT or PTE ECPT
1	0	i	Direct Walk in PMD ECPT Way i
0	1	X	Size Walk in PTE ECPT
1	1	i	Partial Walk: Direct Walk in PMD ECPT Way i and Size Walk in PTE ECPT

Table 1. Actions taken by the page walker for the different values of a PMD-CWT section header. In the table, ECPT means elastic cuckoo page table.

If the 2MB Bit is clear and the 4KB Bit is set, all the translations for this section are in the PTE elastic cuckoo page table. Sadly, there is no information available in the PMD-CWT section header about which way(s) of the PTE elastic cuckoo page table should be accessed. Hence, the walker performs a *Size Walk* in the PTE elastic cuckoo page table. Finally, if both the 2MB and 4KB Bits are set, the target page could have either size. Hence, the walker performs a *Partial Walk*. The walk is composed of a *Direct Walk* in the PMD elastic cuckoo page table (using the way indicated by the Way Bits), and a *Size Walk* in the PTE elastic cuckoo page table.

The PUD-CWT is organized in a similar manner. Each section header now covers a virtual memory section of 8GB. A section header has 5 bits: a *1GB Bit*, a *2MB Bit*, and a *4KB Bit* to indicate whether the section maps one or more 1GB pages, one or more 2MB pages, and/or one or more 4KB pages, respectively; and two *Way Bits* to indicate the way of the PUD elastic cuckoo page table that holds the mapped translations of the 1GB pages in this virtual memory section. The Way Bits are only meaningful if the 1GB Bit is set. The actions taken based on the value of these bits are similar to those for the PMD-CWT. To simplify, we do not detail them. Overall, our design encodes substantial information about virtual memory sections with only a few bits in the CWTs.

With this encoding of the PUD-CWT and PMD-CWT, the OS updates these tables infrequently. Most updates to the elastic cuckoo page tables do not require an update to the CWTs. For example, take a section header of the PMD-CWT. Its 2MB Bit and 4KB Bit are only updated the *first time* that a 2MB page or a 4KB page, respectively, is allocated in the section. Recall that a section's size is equal to 4096 4KB pages. Also, the Way Bits of the section header in the PMD-CWT are not updated when a 4KB page is allocated or rehashed.

Finally, the conceptual design is that, on a page walk, the hardware accesses the PUD-CWT first, then the PMD-CWT and, based on the information obtained, issues a reduced number of page table accesses. The actual design, however, is that the CWT information is cached in and read from small caches in the MMU that are filled off the critical path. We describe these caches next.

5.3 Cuckoo Walk Caches

The PUD-CWT and PMD-CWT reside in memory and their entries are cacheable in the cache hierarchy, like elastic cuckoo page table entries. However, to enable very fast access on a page walk, our design caches some of their entries on demand in *Cuckoo Walk Caches* (CWCs) in the MMU. We call these caches PUD-CWC and PMD-CWC, and replace the page walk caches of radix page tables. The hardware page walker checks the CWCs before accessing the elastic cuckoo page tables and, based on the information in the CWCs, it is able to issue fewer parallel accesses to the page tables.

The PUD-CWC and PMD-CWC differ in a crucial way from the page walk caches of radix page tables: their contents (like those of the CWTs) are *decoupled* from the contents of the page tables. The CWCs store page size and way information. This is unlike in the conventional page walk caches, which store page table entries. As a result, the CWCs and CWTs can be accessed independently of the elastic cuckoo page tables. This fact has two implications.

The first one is that, on a CWC *miss*, the page walker can proceed to access the target page table entry *right away*— albeit by issuing more memory accesses in parallel than otherwise. After the page walk has completed, the TLB has been filled, and execution has restarted, the appropriate CWT entries are fetched and cached in the CWCs, off the critical path. Instead, in the conventional page walk caches, the entries in the page walk caches have to be sequentially generated on the critical path before the target page table entry can be accessed and the TLB can be filled.

The second implication is that a CWC entry is very small. It only includes a few page size and way bits. This is unlike an entry in conventional page walk caches, which needs to include the physical address of the next level of page translation, in addition to the page size and other information. For example, a PMD-CWC section header covers a 16MB region (4096 4KB pages) with only 4 bits, while an entry in the traditional PMD page walk cache only covers a 2MB

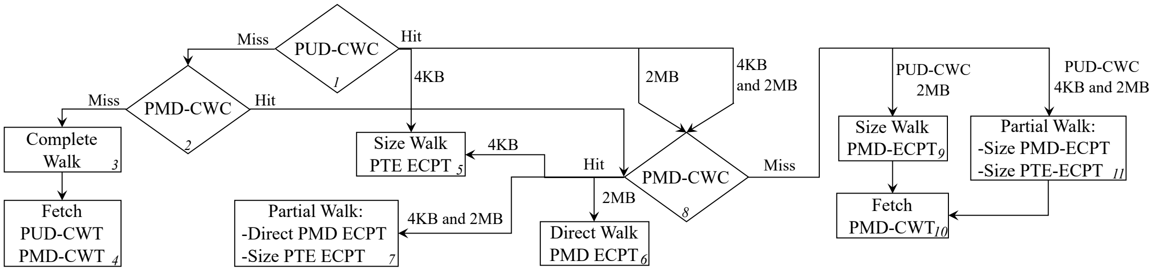


Figure 12. Steps of a page translation. ECPT stands for Elastic Cuckoo Page Tables.

region (512 4KB pages) using 64 bits. The result is that CWCs have a small size and a very high hit rate.

6 Implementation

6.1 Steps of a Page Translation

Figure 12 presents the steps of a page translation with Elastic Cuckoo Page Tables (ECPTs). For simplicity, we assume that there is no 1GB page. On a TLB miss, the page table walker hardware first checks the PUD-CWC ①. If the PUD-CWC hits, the corresponding section header is examined. It can indicate that the section contains only 4KB pages, only 2MB pages, or both 2MB and 4KB pages. In the case of only 4KB pages, the page walker performs a Size Walk in the PTE ECPT ⑤, which will either bring the translation to the TLB or trigger a page fault. In the case of a section with only 2MB pages or with both 2MB and 4KB pages, the PMD-CWC is accessed to obtain additional information ⑧.

If the PMD-CWC hits, the smaller section accessed may again contain only 4KB pages, only 2MB pages, or both 2MB and 4KB pages. In the first case, the page walker performs a Size Walk in the PTE ECPT ⑤; in the second case, it performs a Direct Walk in the PMD ECPT ⑥; and in the third case, it performs a Partial Walk ⑦, which includes a Direct Walk in the PMD ECPT and a Size Walk in the PTE ECPT.

If, instead, the PMD-CWC misses, the walker uses the partial information provided by the PUD-CWC. Specifically, if the PUD section contained only 2MB pages, the walker performs a Size Walk in the PMD ECPT ⑨ (since there is no information on the PMD ECPT ways); if it contained both 2MB and 4KB pages, it performs a Partial Walk ⑪, which includes Size Walks in both the PMD ECPT and the PTE ECPT . In both cases, after the TLB is filled and execution resumes, the hardware fetches the missing PMD-CWT entry into the PMD-CWC ⑩.

A final case is when the access to the PUD-CWC misses. The walker still accesses the PMD-CWC to see if it can obtain some information ②. If the PMD-CWC hits, the section accessed may contain only 4KB pages, only 2MB pages, or both 2MB and 4KB pages. The walker proceeds with actions ⑤, ⑥, and ⑦, respectively. If the PMD-CWC misses, the walker has no information, and it performs a Complete Walk to bring the translation into the TLB ③. After that, the hardware fetches the missing PUD-CWT and PMD-CWT entries into the PUD-CWC and PMD-CWC, respectively ④.

Since we are not considering 1GB pages in this discussion, on a PUD-CWC miss and PMD-CWC hit, there is no need to fetch the PUD-CWT entry. If 1GB pages are considered, the hardware needs to fetch the PUD-CWT entry in this case.

6.2 Concurrency Issues

Elastic cuckoo page tables naturally support multi-process and multi-threaded applications and, like radix page tables, abide by the Linux concurrency model regarding page table management. Specifically, multiple MMU page walkers can perform page table look-ups while one OS thread performs page table updates.

When a page table entry is found to have the Present bit clear, a page fault occurs. During the page fault handling, other threads can still perform look-up operations. Our scheme handles these cases like in current radix page tables in Linux: when the OS updates the elastic cuckoo page tables or the CWTs, it locks them. Readers may temporarily get stale information from their CWCs, but they will eventually obtain the correct state.

The OS uses synchronization and atomic instructions to insert entries in the elastic cuckoo page tables, to move elements across ways in a page table, to move entries across page tables in a resizing operation, to update the Rehashing Pointers in a resizing operation, and to update the CWTs.

If the CWCs were coherent, updates to CWT entries would invalidate entries in the CWCs. In our evaluation, we do not make such assumption. In this case, when the page walker uses information found in CWCs to access a translation and the accesses fail, the walker performs the walk again. This time, however, it accesses the *remaining* ways of the target elastic cuckoo page table or, if a resize is in progress, the remaining ways of both tables. This action will find entries that have been moved to another way. After the translation is obtained, the stale entries in the CWCs are refreshed. A similar process is followed for the Rehashing Pointers.

7 Evaluation Methodology

Modeled Architectures. We use full-system cycle-level simulations to model a server architecture with 8 cores and 64 GB of main memory. We model a baseline system with 4-level radix page tables like the x86-64 architecture, and our proposed system with elastic cuckoo page tables. We model these systems (i) with only 4KB pages, and (ii) with multiple page sizes by enabling Transparent Huge Pages (THP) in the Linux kernel [70]. We call these systems *Baseline 4KB*, *Cuckoo 4KB*, *Baseline THP*, and *Cuckoo THP*.

The architecture parameters of the systems are shown in Table 2. Each core is out-of-order and has private L1 and L2 caches, and a portion of a shared L3 cache. A core has private L1 and L2 TLBs and page walk caches for translations. The Cuckoo architectures use 3-ary elastic cuckoo hashing for the Elastic Cuckoo Page Tables (ECPTs) and 2-ary elastic cuckoo hashing for the CWTs. The Baseline architectures use the 4-level radix page tables of x86-64. We size the CWCs in the Cuckoo architecture to consume less area than the Page Walk Caches (PWC) in the Baseline architectures, which are modeled after x86-64. Specifically, the combined 2 CWCs have 18 entries of 32B each, for a total of 576B; the combined 3 PWCs have 96 entries of 8B each, for a total of 768B.

Processor Parameters	
Multicore chip	8 OoO cores, 256-entry ROB, 2GHz
L1 cache	32KB, 8-way, 2 cycles round trip (RT), 64B line
L2 cache	512KB, 8-way, 16 cycles RT
L3 cache	2MB per core, 16-way, 56 cycles RT
Per-Core MMU Parameters	
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
PWC	3 levels, 32 entries/level, 4 cycles RT
Elastic Cuckoo Page Table (ECPT) Parameters	
Initial PTE ECPT size	16384 entries × 3 ways
Initial PMD ECPT size	16384 entries × 3 ways
Initial PUD ECPT size	8192 entries × 3 ways
Initial PMD-CWT size	4096 entries × 2 ways
Initial PUD-CWT size	2048 entries × 2 ways
Rehashing Threshold	$r_t = 0.6$
Multiplicative Factor	$k = 4$
PMD-CWC; PUD-CWC	16 entries, 4 cycles RT; 2 entries, 4 cycles RT
Hash functions: CRC	Latency: 2 cycles; Area: $1.9 * 10^{-3} mm$ Dyn. energy: $0.98 pJ$; Leak. power: $0.1 mW$
Main-Memory Parameters	
Capacity; #Channels; #Banks	64GB; 4; 8
t_{RP}-t_{CAS}-t_{RCD}-t_{RAS}	11-11-11-28
Frequency; Data rate	1GHz; DDR

Table 2. Architectural parameters used in the evaluation.

Modeling Infrastructure. We integrate the Simics [46] full-system simulator with the SST framework [6, 59] and the DRAMSim2 [61] memory simulator. We use Intel SAE [17] on Simics for OS instrumentation. We use CACTI [7] for energy and access time evaluation of memory structures, and the Synopsys Design Compiler [65] for evaluating the RTL implementation of the hash functions. Simics provides the actual memory and page table contents for each memory address. We model and evaluate the hardware components of elastic cuckoo page tables in detail using SST.

Workloads. We evaluate a variety of workloads that experience different levels of TLB pressure. They belong to the graph analytics, bioinformatics, HPC, and system domains. Specifically, we use eight graph applications from the GraphBIG benchmark suite [48]. Two of them are social graph analysis algorithms, namely Betweenness Centrality (BC) and Degree Centrality (DC); two are graph traversal algorithms, namely Breadth-Fist Search (BFS) and Depth-First Search (DFS); and four are graph analytics benchmarks for topological analysis, graph search/flow and website relevance, namely Single Source Shortest Path (SSSP), Connected Components (CC), Triangle Count (TC), and PageRank (PR). From the bioinformatics domain, we use MUMmer from the BioBench suite [3], which performs genome-level alignment. From the HPC domain, we use GUPS from HPC Challenge [45], which is a random access benchmark that measures the rate of integer random memory updates. Finally, from the system domain, we select the Memory benchmark from the SysBench suite [66], which stresses the memory subsystem. We call it SysBench.

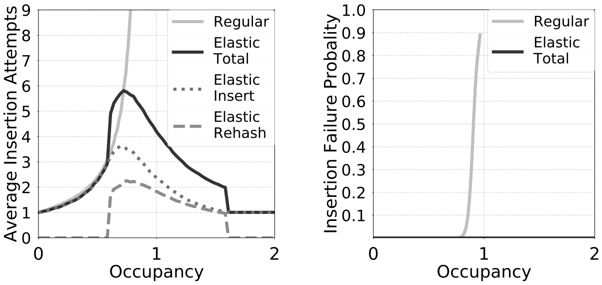


Figure 13. Elastic cuckoo hashing characterization.

The memory footprints of these workloads are: 17.3GB for BC, 9.3GB for DC, 9.3GB for BFS, 9GB for DFS, 9.3GB for SSSP, 9.3GB for CC, 11.9GB for TC, 9.3GB for PR, 6.9GB for MUMmer, 32GB for GUPS, and 32GB for SysBench.

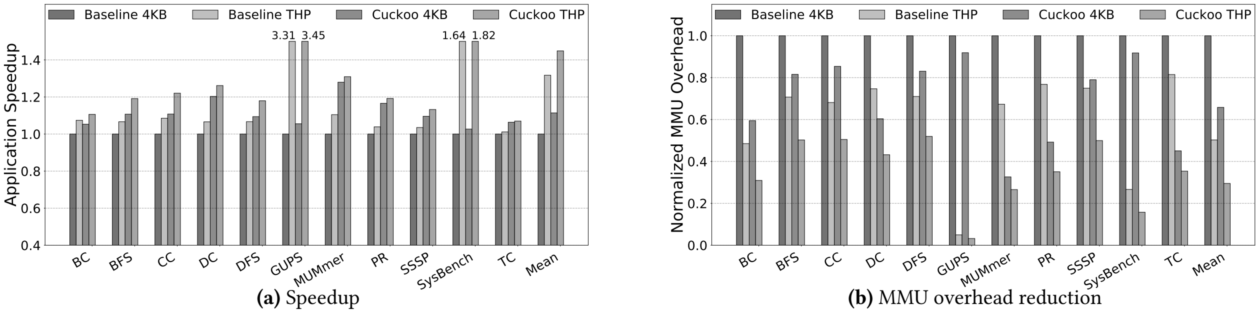
For each individual workload, we perform full-system simulations for all the different configurations evaluated. When the region of interest of the workload is reached, the detailed simulations start. We warm-up the architectural state for 50 million instructions per core, and then measure 500 million instructions per core.

8 Evaluation

8.1 Elastic Cuckoo Hashing Characterization

Figure 13 characterizes the behavior of elastic cuckoo hashing. It shows the average number of insertion attempts to successfully insert an element (left), and the probability of insertion failure after 32 attempts (right), both as a function of the table occupancy. We use 3-ary elastic cuckoo hashing and the BLAKE hash function [5]. Unlike in Figure 4, in this figure, the occupancy goes beyond one. The reason is that, when occupancy reaches the rehashing threshold $r_t = 0.6$, we allocate a new hash table with a multiplicative factor $k = 4$ and perform gradual resizing. Recall that, in gradual resizing, every insert is followed by a rehash that moves one element from the current to the new hash table.

The average number of insertion attempts with elastic cuckoo hashing is the *Elastic Total* curve. This curve is the addition of the *Elastic Insert* and the *Elastic Rehash* curves.



(a) Speedup

(b) MMU overhead reduction

Figure 14. Performance impact of elastic cuckoo page tables: (a) application speedup and (b) MMU overhead reduction.

The latter are the rehashes of elements from the old to the new table during resizing. In more detail, assume that one element is inserted during resizing. This insertion plus any additional re-insertions due to collisions are counted in *Elastic Insert*. The associated rehash of one element from the old to the new hash table plus any additional re-insertions due to collisions are counted in *Elastic Rehash*.

We see that, at an occupancy equal to r_t, *Elastic Rehash* jumps off. As resizing proceeds and elements start spreading over the larger new table, the insertion attempts due to rehashes decrease. When the resize terminates at an occupancy of ≈1.6, the *Elastic Rehash* curve falls to zero. The *Elastic Insert* curve gradually decreases as resizing proceeds. As a result, *Elastic Total* has a peak at the beginning of the resize but then goes down to 1 after the resize completes. Another peak would occur later, when the new table is resized.

The figure also shows the attempts with *Regular* cuckoo hashing without resizing. This curve is taken from Figure 4. Overall, elastic cuckoo hashing with resizing never reaches a large number of insertion attempts. Further, as shown in the rightmost figure, no insertion failure occurs.

8.2 Elastic Cuckoo Page Table Performance

Figure 14 evaluates the performance impact of elastic cuckoo page tables. Figure 14a shows the speedup of the applications running on *Baseline THP*, *Cuckoo 4KB*, and *Cuckoo THP* over running on *Baseline 4KB*.

The figure shows that, with 4KB pages only, using elastic cuckoo page tables (*Cuckoo 4KB*) results in an application speedup of 3–28% over using conventional radix page tables (*Baseline 4KB*). The mean speedup is 11%. When Transparent Huge Pages (THP) are enabled, the speedups with either radix page tables or elastic cuckoo page tables improve substantially, due to reduced TLB misses. The speedup of *Cuckoo THP* over *Baseline THP* is 3–18%. The mean is 10%. These are substantial application speedups.

In fact, *Cuckoo 4KB* outperforms not only *Baseline 4KB* but also *Baseline THP* in several applications. Some applications such as SSSP and TC do not benefit much from THP. However, *Cuckoo 4KB* also outperforms *Baseline THP* in applications such as MUMmer that leverage 2MB pages extensively.

The performance gains attained by elastic cuckoo page tables come from several reasons. First and foremost, a page walk in elastic cuckoo page tables directly fetches the final translation, rather than having to also fetch intermediate levels of translation sequentially as in radix page tables. This ability speeds-up the translation. Second, performance is improved by the high hit rates of CWCs, which are due to the facts that CWCs do not have to store entries with intermediate levels of translation and that each CWC entry is small — again unlike radix page tables. Finally, we observe that when the page walker performs Size and Partial Walks, it brings translations into the L2 and L3 caches that, while not loaded into the TLB, will be used in the future. In effect, the walker is prefetching translations into the caches.

Figure 14b shows the time spent by all the memory system requests in the MMU, accessing the TLBs and performing the page walk. The time is shown normalized to the time under *Baseline 4KB*. This figure largely follows the trends in Figure 14a. On average, *Cuckoo 4KB* reduces the MMU overhead of *Baseline 4KB* by 34%, while *Cuckoo THP*'s overhead is 41% lower than that of *Baseline THP*. The figure also shows that applications like GUPS and SysBench, which perform fully randomized memory accesses, benefit a lot from THPs.

We have also evaluated applications that have a lower page walk overhead, especially with THP, such as MCF and Cactus from SPEC2006 [32], Streamcluster from PARSEC [16], and XSBench [71]. Their memory footprints are 1.7GB, 4.2GB, 9.1GB, and 64GB, respectively. In these applications, while elastic cuckoo page tables reduce the MMU overhead compared to radix page tables, address translation is not a bottleneck. Hence, application performance remains the same.

8.3 Elastic Cuckoo Page Table Characterization

8.3.1 MMU and Cache Subsystem. Figure 15 characterizes the MMU and cache subsystem for our four configurations. From top to bottom, it shows the number of MMU requests Per Kilo Instruction (PKI), the L2 cache Misses Per Kilo Instruction (MPKI), and the L3 MPKI. In each chart, the bars are normalized to *Baseline 4KB*.

MMU requests are the requests that the MMU issues to the cache hierarchy on a TLB miss. For the baseline systems, they are memory requests to obtain page translations for

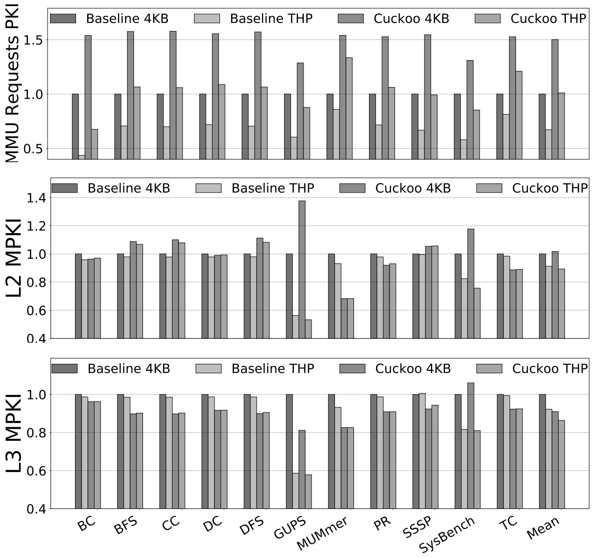


Figure 15. Characterizing the MMU and cache subsystem.

the four radix tree levels; for the cuckoo systems, they are requests for page translations and for CWT entries. From the figure, we see that *Cuckoo 4KB* and *Cuckoo THP* issue many more requests than *Baseline 4KB* and *Baseline THP*, respectively. There are two reasons for this. The first and most important one is that many of the walks of the cuckoo page walker are not Direct Walks. Therefore, they request more page table entries than required — in fact no access to the PTE elastic cuckoo page table can use a Direct Walk because we have no PTE-CWT table. The second reason is the accesses to the CWTs themselves.

Fortunately, most of these additional MMU accesses are intercepted by the L2 and L3 caches and do not reach main memory. Indeed, as we show in the central and bottom chart of Figure 15, the L2 MPKI of the Cuckoo systems is similar to that of the Baseline systems, and the L3 MPKI of the Cuckoo systems is lower than that of the Baseline systems. The reason is two fold. First, the CWT entries are designed to cover large sections of memory. Hence, the needed entries fit in a few cache lines, leading to high cache hit rates. The second reason is that the additional elastic cuckoo page table entries that are brought in by accessing multiple ways of a table are typically later re-used by another access. Accessing them now prefetches them for a future access. In summary, despite elastic cuckoo page tables issuing more MMU requests, most of the requests hit in the caches and the overall traffic to main memory is lower than with radix page tables.

To illustrate this point, Figure 16 considers all the MMU accesses in the MUMmer application and groups them in bins based on the time they take to complete. The figure shows data for *Baseline THP* and *Cuckoo THP*. On top of the bars, we indicate the likely layer of the memory hierarchy accessed for certain ranges of latencies: cache hit, 1st DRAM access, 2nd DRAM access, and 3rd DRAM access. *Cuckoo THP* never performs more than one DRAM access. From the figure, we see that *Cuckoo THP* MMU accesses have much lower latencies; most are intercepted by the caches or at worst take around 200 cycles. *Baseline THP* MMU accesses often perform two or three DRAM accesses, and have a long latency tail that reaches over 500 cycles.

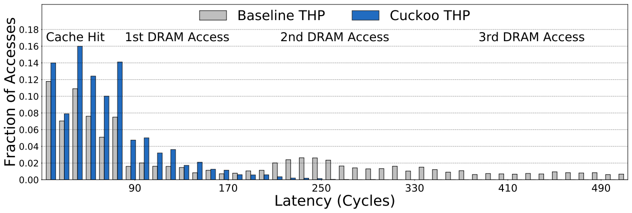


Figure 16. Histogram of MMU accesses in MUMmer.

8.3.2 Types of Walks. We now consider the elastic cuckoo page translation process of Figure 12, and measure the relative frequency of each type of cuckoo walk shown in Figure 10. We use *Cuckoo THP* with 2MB huge pages. As a reference, the average hit rates of PUD-CWC and PMD-CWC across all our applications are 99.9% and 87.7%, respectively.

Figure 17 shows the distribution of the four types of cuckoo walks for each application. Starting from the bottom of the bars, we see that the fraction of *Complete Walks* is negligible. A complete walk only happens when both PUD-CWC and PMD-CWC miss, which is very rare. *Partial Walks* occur when the memory section accessed has both huge and regular pages. Situations where both page sizes are interleaved in virtual memory within one section are infrequent. They occur to some extent in BC, and rarely in other applications.

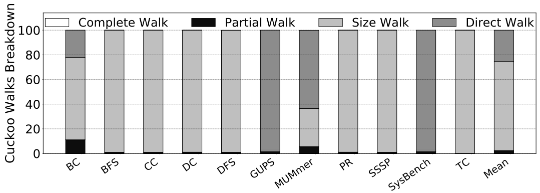


Figure 17. Distribution of the types of cuckoo walks.

Fortunately, most walks are of the cheap kinds, namely *Size Walks* and *Direct Walks*. *Size Walks* occur when the memory segment only contains regular pages. Hence, we observe this behavior in applications that do not take advantage of huge pages, such as BFS, CC, DC, DFS, PR, SSSP, TC and, to a lesser extent, BC. These walks are the most common ones. They also occur when the region only has huge pages but the PMD-CWC misses — an infrequent case.

Finally, *Direct Walks* occur when the memory segment only contains huge pages. We observe them in applications that use huge pages extensively, like GUPS, SysBench, and MUMmer. Overall, cheap walks are the most common.

8.3.3 Memory Consumption. Figure 18 shows the memory consumption of the page tables in the different applications for various configurations. For the Cuckoo systems, the bars also include the memory consumption of CWTs. The first bar (*Required*) is the number of PTEs used by the application times 8 bytes. The figure then shows *Baseline 4KB*, *Cuckoo 4KB*, and *Cuckoo 4KB* with table downsizing. Note that we have not used downsizing anywhere else in this paper. On average, *Required* uses 26MB. *Baseline 4KB* consumes on average 27MB, as it also needs to keep the three upper levels of translation. *Cuckoo 4KB* consumes on average 36MB, due to allocating hash tables with a power-of-two number of entries. The CWTs add very little space. Overall, the absolute increase in memory consumption needed to support *Cuckoo 4KB* is tiny, compared to the total memory capacity of modern systems. Finally, *Cuckoo 4KB* with table downsizing consumes on average 29MB.

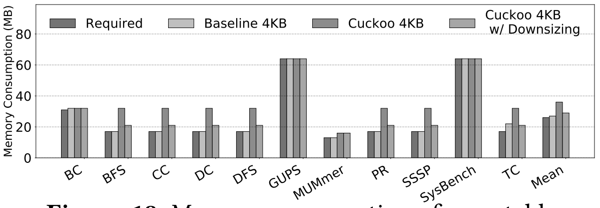


Figure 18. Memory consumption of page tables.

8.3.4 Comparison to 2-ary Cuckoo Page Tables. We repeat our evaluation of elastic cuckoo page tables using 2-ary elastic cuckoo hashing for the page tables rather than our default 3-ary design. Our results show that using 3-ary structures speeds-up applications over using 2-ary structures: the applications run on average 6.7% faster in *Cuckoo 4KB* and 3.3% faster in *Cuckoo THP*. The reason is that 2-ary structures are liable to hash collisions at lower occupancies and, therefore, need to be resized sooner. We find that, over the course of an application, 2-ary tables require more rehashing operations than 3-ary tables: on average across our applications, they require 63.3% more rehashes in *Cuckoo 4KB* and 84% in *Cuckoo THP*. The additional accesses issued by the MMUs with 3-ary structures have relatively little performance impact because they typically hit in the caches.

9 Other Related Work

To reduce TLB misses, recent studies have proposed to optimize TLB organizations by clustering, coalescing, contiguity [14, 18, 21, 42, 54–56, 64, 72], prefetching [15, 41, 63], speculative TLBs [9], and large part-of-memory TLBs [47, 62]. To increase TLB reach, support for huge pages has been extensively studied [21, 26, 27, 29, 43, 49, 51–53, 57, 60, 67, 69], with OS-level improvements [26, 43, 51, 52]. Other works propose direct segments [10, 25] and devirtualized memory [31], and suggest that applications manage virtual memory [2].

Many of these advances focus on creating translation contiguity: large contiguous virtual space that maps to large contiguous physical space. In this way, fewer translations are required, TLB misses are reduced, and costly multi-step page walks are minimized. Unfortunately, enforcing contiguity hurts the mapping flexibility that the kernel and other software enjoy. Further, enforcing contiguity is often impossible — or counterproductive performance-wise. Instead, in our work, we focus on dramatically reducing the cost of page walks by creating a single-step translation process, while maintaining the existing abstraction for the kernel and other software. As a result, we do not require contiguity and retain all the mapping flexibility of current systems.

10 Conclusion

This paper presented Elastic Cuckoo Page Tables, a novel page table design that transforms the sequential address translation walk of radix page tables into fully parallel lookups, harvesting for the first time the benefits of memory-level parallelism for address translation. Our evaluation showed that elastic cuckoo page tables reduce the address translation overhead by an average of 41% over conventional radix page tables, and speed-up application execution by 3–18%. Our current work involves exploring elastic cuckoo page tables for virtualized environments.

Acknowledgments

This work was supported by NSF under grants CCF 16-49432 and CNS 17-63658.

Appendix A: Multiplicative Factor

To find a good value for the Multiplicative Factor k in a resize operation, we compute the number of entries that the new table receives during the resize operation. To simplify the analysis, we neglect insertion collisions, which move elements from one way to another way of the same table. Hence, we only need to consider 1 way of the old table (which has T entries) and 1 way of the new table (which has $k \times T$ entries). A similar analysis can be performed considering collisions, following the procedure outlined in [24, 50].

Recall that, during resizing, an insert operation can insert the entry in the old hash table (*Case Old*) or in the new hash table (*Case New*). In either case, after the insert, one entry is moved from the old to the new table, and the Rehashing Pointer is advanced. Therefore, in Case Old, the new table receives one entry; in Case New, it receives two.

If all the inserts during resizing were of Case Old, the new table would receive at most T elements during resizing, since the Rehashing Pointer would by then reach the end of the old table. If all the inserts during resizing were of Case New, there could only be $r_t \times T$ insertions, since the old table had $r_t \times T$ elements and, by that point, all elements would have moved to the new table. Hence, the new table would receive $2r_t \times T$ elements during resizing.

The worst case occurs in a resize that contains some Case Old and some Case New inserts. The scenario is as follows. Assume that all the $r_t \times T$ elements in the old table are at the top of the table. During resizing, there are first $r_t \times T$ Case New inserts, and then $(1 - r_t) \times T$ Case Old inserts. Hence, the new table receives $(r_t + 1) \times T$ elements. This is the worst case. Hence, to avoid reaching the point where a new resize operation is triggered inside a resize operation, it should hold that $(r_t + 1) \times T < r_t \times k \times T$, or $k > (r_t + 1)/r_t$.

References

[1] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting Hardware-assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*.

[2] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*.

[3] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. 2005. BioBench: A Benchmark Suite of Bioinformatics Applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*.

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

[5] Jean-Philippe Aumasson, Willi Meier, Raphael Phan, and Luca Henzen. 2014. *The Hash Function BLAKE.* Springer.

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

[7] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (June 2017).

[8] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*.

[9] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*.

[10] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*.

[11] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. 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)*.

[12] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*.

[13] Abhishek Bhattacharjee. 2017. Translation-Triggered Prefetching. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.

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

[15] Abhishek Bhattacharjee and Margaret Martonosi. 2010. 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)*.

[16] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[17] Nadav Chachmon, Daniel Richins, Robert Cohn, Magnus Christensson, Wenzhi Cui, and Vijay Janapa Reddi. 2016. Simulation and Analysis Engine for Scale-Out Workloads. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*.

[18] J. Bradley Chen, Anita Borg, and Norman P. Jouppi. 1992. A Simulation Based Study of TLB Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*.

[19] Jonathan Corbet. 2005. Four-level page tables. https://lwn.net/Articles/117749/.

[20] Jonathan Corbet. 2017. Five-level page tables. https://lwn.net/Articles/717293/.

[21] Guilherme Cox and Abhishek Bhattacharjee. 2017. 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)*.

[22] Cort Dougan, Paul Mackerras, and Victor Yodaiken. 1999. Optimizing the Idle Task and Other MMU Tricks. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*.

[23] Stephane Eranian and David Mosberger. 2000. *The Linux/ia64 Project: Kernel Design and Status Update.* Technical Report HPL-2000-85. HP Laboratories Palo Alto.

[24] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. 2005. Space Efficient Hash Tables with Worst Case Constant Access Time. *Theory of Computing Systems* 38, 2 (Feb. 2005), 229–248.

[25] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*.

[26] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*.

[27] Mel Gorman and Patrick Healy. 2010. Performance Characteristics of Explicit Superpage Support. In *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*.

[28] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. 2005. Itanium — A System Implementor's Tale. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC'05)*.

[29] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. 2015. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'15)*.

[30] Frank T. Hady, Annie P. Foong, Bryan Veal, and Dan Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* 105, 9 (2017).

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

[32] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (Sept. 2006), 1–17.

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

[34] IBM. 2005. 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.

[35] Intel. 2010. 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.

[36] Intel. 2015. 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.

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

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

[39] Bruce L. Jacob and Trevor N. Mudge. 1998. 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)*.

[40] Joefon Jann, Paul Mackerras, John Ludden, Michael Gschwind, Wade Ouren, Stuart Jacobs, Brian F. Veale, and David Edelsohn. 2018. IBM POWER9 system software. *IBM Journal of Research and Development* 62, 4/5 (June 2018).

[41] Gokul B. Kandiraju and Anand Sivasubramaniam. 2002. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*.

[42] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*.

[43] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.

[44] Linux Kernel. 2019. Page Table Types. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/include/asm/pgtable_types.h?h=v4.19.1.

[45] Piotr R. Luszczek, David H. Bailey, Jack J. Dongarra, Jeremy Kepner, Robert F. Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*.

[46] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A Full System Simulation Platform. *IEEE Computer* (2002).

[47] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. 2017. CSALT: Context Switch Aware Large TLB. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*.

[48] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*.

[49] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*.

[50] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* 51, 2 (May 2004), 122–144.

[51] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. 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)*.

[52] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*.

[53] Misel-Myrto Papadopoulou, Xin Tong, André Seznec, and Andreas Moshovos. 2015. Prediction-Based Superpage-Friendly TLB Designs. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*.

[54] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. 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)*.

[55] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. 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)*.

[56] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*.

[57] Binh Pham, Ján Veselŷ, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. 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)*.

[58] Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. 2011. *Phase Change Memory: From Devices to Systems* (1st ed.). Morgan & Claypool Publishers.

[59] Arun F. Rodrigues, Jeanine Cook, Elliott Cooper-Balis, K. Scott Hemmert, Chad Kersey, Rolf Riesen, Paul Rosenfeld, Ron Oldfield, and Marlow Weston. 2006. The Structural Simulation Toolkit. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'10)*.

[60] Theodore H. Romer, Wayne H. Ohlrich, Anna R. Karlin, and Brian N. Bershad. 1995. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*.

[61] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* (2011).

[62] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. 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)*.

[63] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. 2000. Recency-Based TLB Preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*.

[64] Shekhar Srikantaiah and Mahmut Kandemir. 2010. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*.

[65] Synopsys. 2019. Design Compiler. https://www.synopsys.com.

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

[67] Madhusudhan Talluri and Mark D. Hill. 1994. 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)*.

[68] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. 1995. A New Page Table for 64-bit Address Spaces. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*.

[69] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in Supporting Two Page Sizes. In *19th International Symposium on Computer Architecture (ISCA'92)*.

[70] The Linux Kernel Archives. 2019. Transparent Hugepage Support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[71] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*.

[72] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation Ranger: Operating System Support for Contiguity-Aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*.

[73] Idan Yaniv and Dan Tsafrir. 2016. 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)*.