Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables

Jovan Stojkovic
University of Illinois
Urbana-Champaign, USA
jovans2@illinois.edu

Dimitrios Skarlatos
Carnegie Mellon University
Pittsburgh, USA
dskarlat@cs.cmu.edu

Apostolos Kokolis
University of Illinois
Urbana-Champaign, USA
kokolis2@illinois.edu

Tianyin Xu
University of Illinois
Urbana-Champaign, USA
tyxu@illinois.edu

Josep Torrellas
University of Illinois
Urbana-Champaign, USA
torrella@illinois.edu

ABSTRACT

A major reason why nested or virtualized address translations are slow is because current systems organize page tables in a multi-level tree that is accessed in a sequential manner. A nested translation may potentially require up to twenty-four sequential memory accesses. To address this problem, this paper presents the first page table design that supports *parallel* nested address translation. The design is based on using hashed page tables (HPTs) for both guest and host. However, directly extending a native HPT design to a nested environment leads to minor gains. Instead, our design solves a new set of challenges that appear in nested environments. Our scheme eliminates all but three of the potentially twenty-four sequential steps of a nested translation—while judiciously limiting the number of parallel memory accesses issued to avoid over-consuming cache bandwidth. As a result, compared to conventional nested radix tables, our design speeds-up the execution of a set of applications by an average of 1.19x (for 4KB pages) and 1.24x (when huge pages are used). In addition, we also show a migration path from current nested radix page tables to our design.

CCS CONCEPTS

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

KEYWORDS

Virtual Memory, Page Tables, Virtualization

ACM Reference Format:
Jovan Stojkovic, Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, Josep Torrellas. 2022. Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables. In *Proceedings of 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507720

1 INTRODUCTION

Cloud computing relies on virtualization hardware to provide strong isolation and enable server consolidation. Virtual Machines (VMs) are multiplexed over hardware resources and offer a safe sandbox for user services. Today, all major cloud providers use VMs, e.g., Amazon's EC2 [7], Microsoft's Azure [62], and Google's Compute Engine [32]. Moreover, to reduce the overheads of VMs, lightweight virtualization frameworks have emerged, such as AWS's Firecracker [2] and Google's gVisor [33]. With these frameworks, some of the main performance overheads of traditional VMs, such as long boot-up times, have been successfully curbed.

However, in spite of nearly twenty years since the inception of virtualization hardware [1, 8, 42] and extensive research [3, 6, 13, 15, 16, 25, 29, 30, 38, 39, 52–54, 66, 72, 76, 80, 89], address translation still introduces substantial performance overhead in virtualized systems. A major reason why address translation has high overhead is because page tables are currently organized in a multi-level tree that is accessed in a sequential manner. This organization is called radix page tables.

In a native (i.e., not virtualized) environment, a virtual to physical address translation requires traversing a tree with four levels of page tables. Such traversal potentially requires issuing up to four sequential memory accesses. In a nested (i.e., virtualized) environment, accessing the table at each level of this tree (now called *guest* page table level) itself requires performing a translation that traverses four levels of pages tables (now called *host* page table levels). Traversing these four levels of host page tables again potentially requires issuing up to four sequential memory accesses. When every step is counted, a nested address translation can require up to twenty-four sequential memory accesses. If this is not bad enough, this problem is likely to get worse soon: new processors such as Intel's Sunny Cove [44, 45] add a fifth level to the tree. As a result, a nested translation may require up to thirty-five sequential memory accesses.

To reduce the overhead of both native and nested address translation, current systems rely on supporting huge pages [56, 66, 67, 84] and hardware caching of address translations. Huge pages reduce the metadata required to support translations. Hardware caching reduces the need to perform expensive memory accesses during translation. Adopted solutions for hardware caching include larger multi-level TLBs and, to reduce the cost of TLB misses, Page Walk caches in the Memory Management Unit (MMU) of the processor.

For the specific case of nested translation, the MMU of current systems further includes Nested TLBs [9, 16]. They cache the traversal of the four host pages table levels needed to access each guest page table level.

Unfortunately, these techniques are now becoming insufficient. Even with them, nested address translation can account for more than 50% of the execution time of applications [6, 18, 38, 52]. With TLB access times already overtaking those of the L2 cache [87], and the upcoming commoditization of terabytes of main memory capacity [47, 48, 73, 86], a redesign of the address translation mechanisms seems inevitable.

To address this challenge, we propose to speed-up the process of nested address translation by exploiting *parallelism.* For that, we rely on hashed page tables (HPTs) [14, 49, 83, 89]. HPTs fundamentally eliminate the sequential steps of radix page tables. In theory, they perform address translation in one step, by hashing a virtual page number into a hash key, accessing the corresponding table entry, and reading the physical page number from there. HPTs have been implemented in the past [26, 27, 35, 40, 41, 43, 50] and recent work [79, 89] has solved some of their traditional shortcomings [14, 35, 89].

In particular, in this paper, we consider our previously-proposed native HPT design called Elastic Cuckoo Page Tables (ECPTs) [79]. If we directly extend the ECPT design in a nested manner for both guest and host, we find that the resulting system delivers minor performance gains over nested radix page tables. The main reason is that a nested ECPT translation sometimes results in many parallel memory accesses, which consume too much bandwidth. Consequently, we analyze the translation mechanisms and redesign them so that they issue fewer parallel memory accesses.

Specifically, we focus on three aspects of a nested ECPT design. The first one concerns misses on a key hardware cache that ECPTs use to cache guest metadata. On a miss, to find the correct location requested by the access, the hardware needs to first translate the missing guest address, causing additional memory accesses. To minimize this problem, we introduce a new hardware cache in the MMU called the Shortcut Translation Cache (STC). The STC keeps the mapping between missing guest addresses and their translations. Intuitively, the STC caches translations of ECPT metadata in a manner logically similar to how the Nested TLB [16] caches translations of radix page tables. The impact of the STC is a reduction in the number of memory accesses caused by address translation.

The second aspect of our design has to do with some metadata that the native ECPT design kept uncached. In a nested ECPT design, such metadata does benefit from hardware caching. Hence, our design caches the metadata in the MMU—some of it adaptively, depending on the locality of the application. The result is a further reduction in the number of memory accesses caused by address translation.

The final aspect of our design leverages the fact that the system may know the page size used by the page tables. If it does, we can further reduce the number of memory accesses involved in address translation.

We call the resulting design *Nested ECPTs*. To our knowledge, Nested ECPTs is the first practical design for *parallel* nested address translation. It eliminates all but three of the potentially twenty-four sequential steps of a nested radix translation. Moreover, we also show how industry can migrate from the current nested radix page tables to Nested ECPTs.

We evaluate Nested ECPTs with full-system simulations. We show that they successfully exploit parallelism during nested address translation and deliver substantially higher performance than nested radix page tables. Compared to nested radix tables, Nested ECPTs speed-up the execution of a set of applications by an average of 1.19x (for 4KB pages) and 1.24x (when huge pages are used).

Overall, our contributions are:

- Nested ECPTs, the first page table design for *parallel* nested address translation.
- A migration path from current systems to Nested ECPTs.
- An evaluation of Nested ECPTs.

2 BACKGROUND

2.1 Radix Page Tables

Radix page tables, which are the design used by most current architectures, organize the translation in a multi-level tree. To translate a memory address, the hardware walks over each level of the tree sequentially. Currently, the x86-64 architecture implements a 4-level tree. In addition, x86-64 supports large pages of 2MB and 1GB. For such pages, the translation is shortened by one or two levels.

To reduce translation overhead, MMUs have small per-core caches called Page Walk Caches (PWCs) [3, 13, 16, 17]. PWCs store intermediate page table entries. On a TLB miss, the hardware performs a page walk. The walk involves checking the PWC to obtain the desired translation and, on a PWC miss, accessing the memory hierarchy to get the translation.

Native Address Translation. Figure 1 shows a native x86-64 page walk to translate a virtual address (VA) to its physical address (PA). The hardware walks through four levels of tables (L_4 to L_1) called PGD, PUD, PMD, and PTE. The hardware first reads the CR3 register, which stores the base of the L_4 table. By adding CR3 and bits 47–39, the hardware obtains the L_4 entry whose content is the base of the correct L_3 table. Then, by adding such content and bits 38–30, one obtains the L_3 entry whose content is the base of the correct L_2 table. The process continues until the correct L_1 entry is read. The L_1 entry stores the physical page number, which together with the page offset (bits 11–0) is the PA. Thus, in the worst case, a page walk requires 4 sequential memory access.

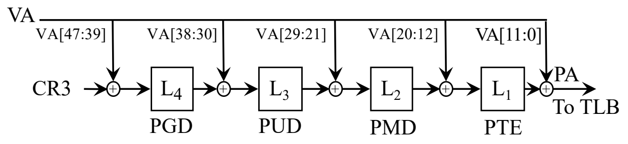

Figure 1: Native page walk in the x86-64 architecture.

For 2MB or 1GB page sizes, the page walk ends at L_2 or at L_3, respectively, resulting in up to three or two memory references. The recently-accessed L_4, L_3, and L_2 table entries are cached in the PWC for future access, but L_1 entries are not [13, 46].

Virtualized Address Translation. Address translation in a virtualized environment is more complex because the physical memory is managed by the hypervisor and is not exposed to guest OSes. A PA viewed by a guest OS is in fact a *guest physical address* (gPA), which needs to be further translated into a *host physical address* (hPA). Hence, the nested address translation of a guest virtual address (gVA) involves two phases: from gVA to gPA, and from gPA to hPA.

Modern hardware-assisted virtualization implements nested paging (e.g., Intel's EPT [46] and AMD's nested paging [9]). Nested paging uses two layers of page tables: each guest OS maintains a *guest page table* that maps gVAs to gPAs, and the hypervisor manages a *host page table* per guest that maps gPAs to hPAs.

Figure 2 shows a nested page walk in x86-64. Square boxes are levels of the host page table (hL_i) and circular boxes are levels of the guest page table (gL_i). The translation stars with a gVA and produces an hPA.

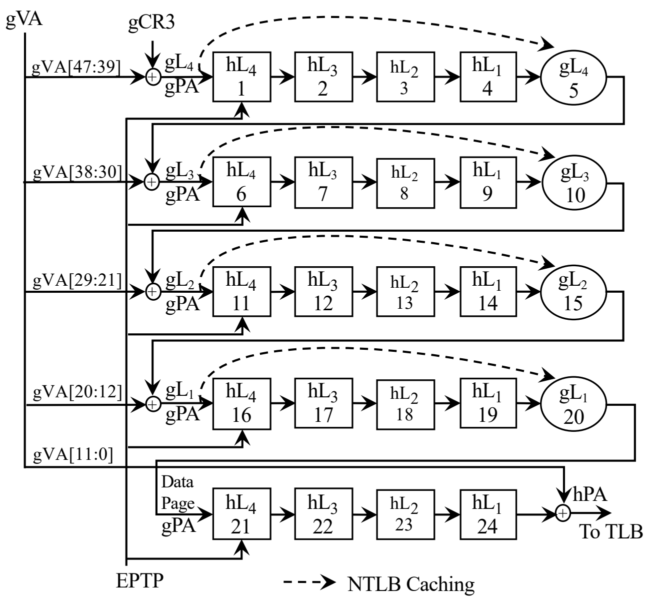

Figure 2: Nested page walk in x86-64. The numbers in the squares or circles are the steps in the translation process.

In order to access each level of the guest page table (gL_i), where $i = \{4,3,2,1\}$ in Steps 5, 10, 15, and 20, the hardware first needs to translate the gPA of the gL_i table to an hPA. The translation of such gPA requires a page walk that iterates over the hL_4 to hL_1 levels of the host page table (Steps 1–4, 6–9, 11–14, and 16–19). At the end, the gL_1 entry at Step 20 produces the gPA of the target page. Then, a final walk is needed to translate this gPA to the hPA of the page (Steps 21–24). Finally, the hardware adds the address obtained from Step 24 to the page offset to obtain the final hPA. The process potentially requires up to 24 *sequential* memory references (Steps 1–24).

Support for huge pages changes the page walk as follows. If the host supports 2MB or 1GB pages, the hL_1 levels of the translation or both the hL_1 and hL_2 levels, respectively, are eliminated. If, in addition, the guest supports 2MB or 1GB pages, the gL_1 level or both the gL_1 and gL_2 levels, respectively, are eliminated.

Recently-accessed table entries of gL_i for $i = \{4,3,2\}$, and of hL_i for $i = \{4,3,2,1\}$ are cached in PWCs [9, 16]. Further, there are Nested TLBs (NTLBs) that cache the translation of the gPA of Level i guest page table (gL_i) to its hPA, as shown with dotted lines in Figure 2. These cached translations can skip four host page table accesses.

2.2 Hashed Page Tables

Hashed page tables (HPTs) are an alternative design to radix page tables. They have been implemented in the IBM PowerPC, HP PA-RISC, and Intel Itanium (IA-64) [26, 40, 43, 49, 50]. In HPTs, address translation is performed by hashing the virtual page number, indexing a hash table, and reading the physical page number from the entry. Unrealistically assuming no hash collisions, no multiple page sizes, and no page sharing across processes, only one memory reference is needed for address translation in the native setup.

In a virtualized environment that uses HPTs, only three memory references are needed for a nested address translation [89]—again unrealistically assuming no hash collisions, no multiple page sizes, and no page sharing.

The idea is as follows. We use the gVA to access the guest HPT and read the gPA of the target data page; this is the outcome of Step 20 in Figure 2. However, to find the guest HPT entry in host memory, we first need to access the host HPT. This is the equivalent of Steps 1-19 in Figure 2. Finally, once we get the gPA of the target data page, we need to access again the host HPT to obtain the hPA of the target page. This is the equivalent of Steps 21-24 in Figure 2. Overall, at most three memory accesses are needed: host HPT, guest HPT, and host HPT.

Figure 3 shows such a nested page walk. On the right, we show the host memory, which is real. On the left, we show in dashes the memory as seen by the guest, which is virtualized.

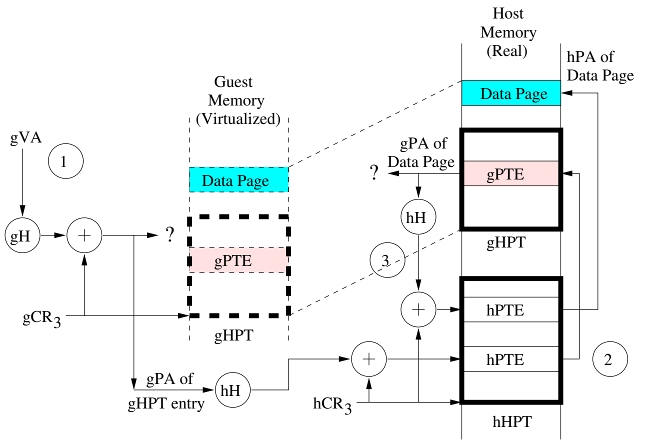

Figure 3: Nested page walk with hashed page tables (HPTs) that unrealistically assumes no hash collisions, no multiple page sizes, and no page sharing. For simplicity, this figure shows a contiguous gHPT in host memory.

In Step ①, the hardware attempts to use the gVA to access the guest HPT (gHPT) entry (shown in the figure as gPTE). It does so by hashing the gVA using the guest hash function (gH) and adding the resulting hash key to the base of the gHPT (stored in the gCR3 register). However, the resulting gPA is virtualized and cannot be directly accessed (hence the question mark in the figure). It needs to be translated to an hPA by accessing the host HPT (hHPT). Therefore, it is hashed using the host hash function (hH) and the resulting hash key is added to the base address of the hHPT (stored in the hCR3 register). The resulting entry in the hHPT (an hPTE) tells where the gPTE is.

In Step ②, the hardware uses the contents of this hPTE as a pointer to access the desired gPTE. This gPTE contains the gPA of the target data page.

In Step ③, the hardware translates this gPA to the hPA of the data page. This is again done by hashing gPA using hH, adding the resulting hash key to hCR3, and accessing the resulting entry in the hHPT. The contents of this hPTE is a pointer to the target data page in host memory.

HPTs have shortcomings. The first one is that hash collisions are expensive: resolving them requires either memory accesses to walk a collision chain [14], open-addressed hash table slots [89], or invoking the OS [27, 35, 43]. The second shortcoming is that, to avoid dynamic resizing of hash tables, the traditional design uses a single HPT shared by all the processes. Such a design cannot support multiple page sizes or page sharing between processes without introducing additional levels of translation. For example, the PowerPC architecture implements a two-level translation procedure for each memory reference to support huge pages and page sharing [41]. The result is additional memory references.

2.3 Elastic Cuckoo Page Tables (ECPTs)

A design that addresses the shortcomings of HPTs is Elastic Cuckoo Page Tables (ECPTs) [79]. ECPTs resolve hash collisions by using cuckoo hashing [28, 65]. A target entry can be in one of d locations in a d-way (or d-ary) cuckoo hash table. Insertions always find space by evicting existing entries and rehashing them in the other ways until, in practically all cases, all entries find a slot [28, 65].

ECPTs use process-private HPTs and, hence, support both multiple page sizes and page sharing without introducing any additional level of translation. ECPTs scale the hash tables on demand according to the memory requirements of the process. Hash table resizing is performed while the process is running.

With this design, a translation is found by looking-up all d ways of an ECPT in parallel. If the system has multiple page sizes, each size has an ECPT. Accesses to the different ECPTs and to their different ways are in parallel. An ECPT entry contains a virtual page number (VPN) tag and multiple consecutive page translation entries packed together to exploit spatial locality. For example, in systems with 64B cache lines, eight consecutive translation entries are stored in a cache line and utilize a single tag.

Issuing many ECPT accesses in parallel can consume substantial bandwidth. To minimize this problem, ECPTs are augmented with Cuckoo Walk Tables (CWTs). CWTs record which ECPT and which way in that ECPT store a given translation. CWTs are cached in a Cuckoo Walk Cache (CWC) in the MMU. Before the hardware attempts to access the ECPTs, it first checks the CWC and, on a hit, prunes the number of parallel accesses issued to the ECPTs.

3 PARALLEL NESTED TRANSLATION: PLAIN DESIGN

Our goal is to support high-performance parallel nested address translation. For that, we explore using HPTs for both guest and host, which fundamentally eliminate the sequential steps of radix page tables. We base our design on ECPTs, since past work has shown that they are a competitive design in a native environment.

In this section, we build a design that directly incorporates the ECPT structures from [79] into host and guest HPTs. We call the design *Plain Nested ECPTs*. As the evaluation will show, this design leads to minor performance gains over nested radix page tables. In Section 4, we discuss the reasons for the small gains and modify the design to address the challenges of a nested environment. We call the modified design *Advanced Nested ECPTs*. Our evaluation section shows the performance of both designs.

Our Plain Nested ECPT design can be understood by examining Figure 3. The hHPT and gHPT are implemented as host ECPTs (hECPTs) and guest ECPTs (gECPTs). There are as many hECPTs (and gECPTs) as page sizes are supported by the host (and guest). To be compatible with the x86 architecture, we assume three page sizes: 1GB, 2MB, and 4KB. Correspondingly, we name the three hECPTs as PUD-, PMD-, and PTE-hECPT, respectively, and the three gECPTs as PUD-, PMD-, and PTE-gECPT, respectively. Note that we are not limited to these specific page sizes, as ECPTs can support any page sizes. Each hECPT and gECPT is organized in d ways.

We describe the Plain Nested ECPT design in two steps. First, we do not limit the number of parallel memory accesses issued. Then, we augment the design with special caches [79] to minimize the number of parallel memory accesses. We use the term Nested ECPT Walk to refer to a nested page walk.

3.1 Design without Limiting Memory Accesses

Figure 4 shows how the design supports the three steps of a nested translation from Figure 3. We assume that each ECPT has 3 ways. We now discuss each step.

Step ①: From gVA to hPTE. This step takes the Virtual Page Number (VPN) of the gVA and obtains potentially multiple hECPT entries that may contain a pointer to the gECPT entry with the gPA. The entries in hECPTs are referred to as hPTEs.

The process is shown in Figure 4. Given that the gVA can reside in guest pages of $n = 3$ different sizes, potentially all the three gECPTs are looked-up (PUD-, PMD-, and PTE-gECPT). Each of these gECPTs has $d = 3$ ways which, potentially, also need to be looked-up. Each way of each gECPT is pointed to by a $gCR3_{i,j}$ register, where i is the page size and j is the way ID. Note that each gECPT and way can potentially use a different guest hash function (gH) but, for simplicity, Figure 4 shows a single gH for all the ways and all the gECPTs. Overall, in summary, the VPN of the gVA is first hashed with the corresponding gH for each gECPT and way, and then added to the corresponding $gCR3_{i,j}$. The result is $n \times d$ addresses which are gPAs of gECPT entries.

Each of these addresses needs to be translated to host physical. Since the desired translation can reside in host pages of different sizes, the hardware needs to check potentially all n hECPTs (i.e.,

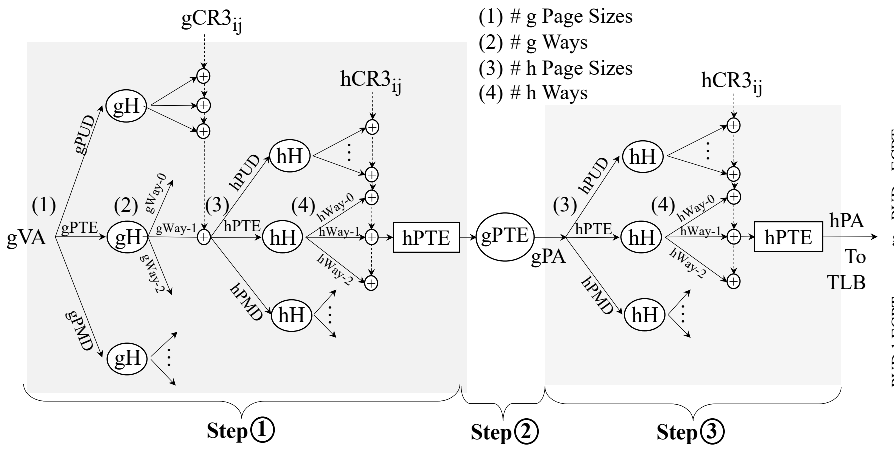

Figure 4: Nested ECPT walk with worst-case number of memory accesses.

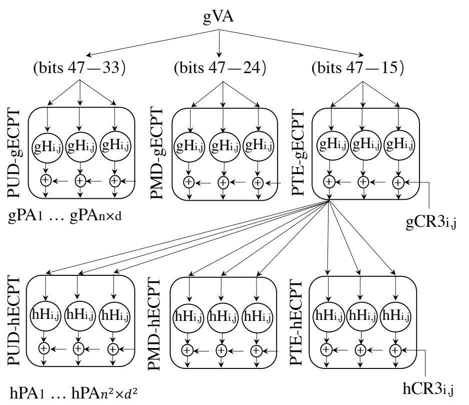

Figure 5: Worst-case Step ① in Figure 4.

PUD-, PMD-, and PTE-hECPT) and, in each one, potentially all d ways. Hence, for each hECPT and way, the address is first hashed with the corresponding hH, and then added to the corresponding hCR3$_{i,j}$. Overall, the total number of parallel accesses to hECPTs in Step ① is potentially $n^2 \times d^2$.

In practice, the number of accesses is much smaller thanks to the caching structures that will be described later. Section 9 evaluates several applications with $n = 3$ and $d = 3$ and shows that the average number of parallel hECPT accesses in Step ① is 2.8.

Figure 5 shows again the worst-case Step ① with $n = 3$ and $d = 3$. The three ways of the three gECPTs are looked up using the correct gVA bits, assuming that each table entry contains eight consecutive translation entries with a single tag. In each case, the correct gH$_{i,j}$ and gCR3$_{i,j}$ are used. For each of the resulting $n \times d$ gPAs, the process is repeated in the hECPTs, creating a worst-case $n^2 \times d^2$ hPAs.

At this point, each of these hECPT accesses will read an hPTE and check the tag for a match. Note that the hPTEs are tagged with host VPN, not with guest VPN. For each gPA$_i$ obtained in the first step of Figure 5, at most only one of the $n \times d$ hECPT accesses will declare a tag match. But, since the hardware is looking for a guest VPN, the hardware will not know at the end of Step ① which of these potential $n \times d$ tag-hitting hPTEs is the one that contains a pointer to the desired gECPT entry. For that, it will need to execute Step ②.

Step ②: From hPTE to gPA of the data page. This step uses the pointers in the (at most) $n \times d$ matching hPTEs to access gECPT entries. Then, the hardware checks the tags of these gECPT entries for a match with the original gVA VPN. At most one gECPT entry succeeds. The contents of this gPTE has the gPA of the data page.

While, in the worst case, $n \times d$ parallel accesses are needed in Step ②, in practice, our experiments with $n = 3$ and $d = 3$ show an average of 2.8 parallel accesses in Step ②.

Step ③: From gPA to hPA of the data page. This final step finds the hPA as shown in Figure 4. In the worst case, it has to access each of the n hECPTs and, in an hECPT, each of its d ways. For each hECPT and way, the gPA is hashed with the corresponding hH, and then added to the corresponding hCR3$_{i,j}$. Of the resulting accesses to the hECPTs, at most one finds an hPTE with matching gPA VPN. That hPTE contains the hPA of the data page (Figure 3).

In the worst case, this step involves $n \times d$ parallel accesses to the hECPTs. In practice, our experiments with $n = 3$ and $d = 3$ show an average of 1.6 parallel accesses in Step ③.

Overall, a Nested ECPT walk requires, in the very worst case, $n^2 \times d^2$ parallel memory accesses, then $n \times d$ parallel memory accesses, and then $n \times d$ parallel accesses. In practice, using the caching structures described next, the average number of accesses observed are 2.8, then 2.8, and then 1.6.

3.2 Augmenting the Design with Caches

The ECPT design for native translations [79] includes Cuckoo Walk Tables (CWTs), which are software structures that help reduce the number of parallel look-ups in a page table walk. There is one CWT for each page size (i.e., PUD-CWT, PMD-CWT, and PTE-CWT), which contains information about which way of the ECPT (if any) has a given translation. Further, entries from these CWTs are cached in a special hardware Cuckoo Walk Cache (CWC) in the MMU. After a TLB miss, the hardware first accesses the CWC and, on a hit, learns the subset of ECPTs (i.e., PUD, PMD, or PTE) and of ways in these ECPTs that it needs to access. In the best case, the hardware only needs to access one way of one ECPT.

In this paper, we propose to have both guest and host Cuckoo Walk Tables (gCWTs and hCWTs), and one guest and one host Cuckoo Walk Cache (gCWC and hCWC). The guest OS manages gCWTs, while the hypervisor manages hCWTs. The hardware automatically accesses the gCWC and hCWC during translation.

Figure 6 shows the Nested ECPT walk from Figure 4 optimized with CWCs. It shows the best case, where each look-up of gCWC and hCWC determines that a single memory access is needed. For this reason, the figure shows a single arrow coming out of every CWC, and the use of only one CR3 (generically represented by CR3$_{i0,j0}$).

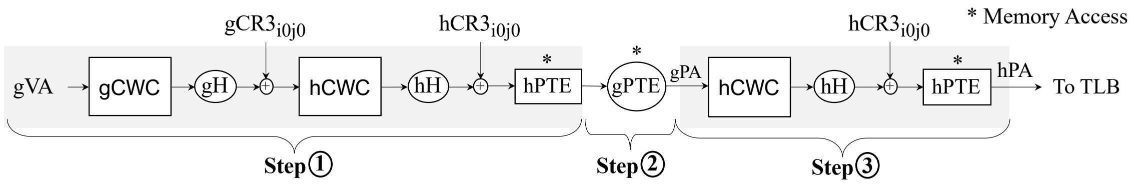

Figure 6: Nested ECPT walk with Cuckoo Walk Caches (CWCs). The figure shows the best-case memory accesses.

In this case, the walk requires only three memory accesses, which are shown with an asterisk where they occur. After the walk completes, the hPA is loaded into the TLB.

4 PARALLEL NESTED TRANSLATION: ADVANCED DESIGN

Unfortunately, as we will see in the evaluation, the Plain Nested ECPT design delivers only about 5% average performance gains over nested radix page tables. The reason is that an ECPT design that worked well in a native environment sometimes results in many parallel memory accesses in a nested ECPT walk, and consumes substantial bandwidth. Consequently, to improve the nested ECPT design, we analyze the translation mechanisms and redesign some of them so that fewer parallel memory accesses are issued.

Specifically, we focus on three aspects of the Nested ECPT design. The first one concerns misses in the gCWC. After a miss, as the hardware attempts to load the corresponding entry into the gCWC in the background, the hardware needs to find the correct memory location to load from. For that, it needs to first translate the missing guest address into a host address, causing additional memory accesses.

The second issue has to do with PTE hCWT data. The native ECPT design did not cache PTE CWT data in the CWC due to insufficient locality. In a nested ECPT design, the PTE hCWT data does benefit from being cached—in some cases adaptively.

The final issue is that a nested environment can leverage system knowledge of the page size used by the page tables.

In this section, we modify the Plain Nested ECPT design to address these issues. We call the new design *Advanced Nested ECPTs*. We consider each of the three issues in turn.

4.1 Translating Guest Cuckoo Walk Tables

During a walk, when a CWC misses, the hardware has no choice but to continue the translation by issuing the $n \times d$ accesses in that step. Then, in the background, the hardware accesses the corresponding CWT and loads the missing entry into the CWC.

If the miss occurred in the hCWC and, therefore, an hCWT entry needs to be accessed, the hardware directly accesses the hCWT. However, if the miss occurred in the gCWC, as the hardware tries to access the corresponding gCWT entry, it finds that it only knows the guest PA of it. Therefore, it first needs to go through a translation step to locate the host PA of the gCWT entry. After that, the hardware can proceed with the access to the gCWT as in the case of the hCWT.

Locating the host PA of a gCWT entry involves a process similar to the translation of a gPA to an hPA shown in Step ③ of Figure 6. Unfortunately, this translation process introduces operations and memory traffic in the background that hurt system performance and can also potentially slow down subsequent accesses to the gCWC.

To eliminate these problems, we consider two possible approaches. One is for the hypervisor to map the gCWTs to a known, contiguous region in the host physical memory. In this way, the hardware can directly access the gCWTs without the need for any additional translation. This design is simple and feasible because gCWTs are very small in size. However, it would require that the guest OS communicate the allocation of the gCWTs to the hypervisor, and thus makes the virtualization process less transparent.

The second approach is to *cache* the gPA to hPA translations of gCWT entries in a very small MMU cache. We call this new cache the *Shortcut Translation Cache* (STC). This is a more virtualization-friendly approach. Our Advanced design uses this approach. With the STC, we remove the operations and memory traffic involved in translating gCWT accesses, improving system performance. We will see in Section 9.4 that a 10-entry STC achieves a hit rate close to 100%.

Intuitively, the STC caches the translations of gCWT data in a manner logically similar to how the Nested TLB [16] caches the translations of radix page tables. Both structures cache translations of guest PAs to host PAs, although the addresses correspond to completely different data structures.

4.2 Caching PTE hCWT Entries

In a native ECPT design, when accessing PTE-ECPTs, caching PTE CWT entries in the CWC could reduce the number of memory accesses. Unfortunately, since the CWC is small, applications with highly-random access patterns can lead to CWC thrashing. The result is increased memory accesses due to subsequent fetches of the needed PTE CWT entries. As a result, the native ECPT design [79] opted not to use a PTE CWT. Similarly, the Plain Nested ECPT design uses no PTE gCWT or PTE hCWT.

In the Advanced design, we again do not use a PTE gCWT for the guest due to poor locality, and also due to a new optimization that we introduce in Section 4.3. However, we find that using this structure for the host (PTE hCWT) and caching it in the hCWC can be beneficial. There are two opportunities to use the hCWC, shown in Steps ① and ③ in Figure 6. We consider each case in turn.

Caching PTE hCWT Entries in Step ①. This step probes the hECPTs to identify the location of gECPT entries. Due to the small size of the gECPTs (especially compared to the address space of the application and its data), and due to the large coverage per entry, this step enjoys very high locality. As a result, in our Advanced design, in Step ①, we use a PTE hCWT and cache it in the hCWC.

Adaptive Caching of PTE hCWT Entries in Step ③. This step probes the hECPTs to identify the location of the requested data pages. The locality of such accesses is very application dependent. Some applications exhibit page locality, while others do not. Therefore, in our Advanced design, we use *Adaptive Caching* of PTE hCWT entries to a different hCWC in Step ③. Specifically, we start by enabling the caching of PTE hCWT entries and, as the application runs, monitor the hit rate of such entries in the hCWC. If their hit rate is low, then PTE hCWT caching is disabled. Then, the hardware monitors the hit rate of the PMD hCWT entries (the next level of cuckoo walk tables) in the hCWC. If the hit rate of PMD hCWT entries in the hCWC is very high, the caching of PTE hCWT entries is re-enabled. By monitoring both hit rates, individual applications typically converge soon to one of the two states.

4.3 Leveraging Page Sizes Used by Page Tables

In a nested environment, if we know that page tables use pages of only a single size, we can optimize page walks. For example, assume that we know that page tables use only 4KB pages. We can then trim the number of parallel memory accesses performed during the second part of Step ① in Figure 4. At that time, the hardware looks-up the hECPTs, looking for an hPTE entry that points to the gECPTs (Figure 3). We know that the gECPTs are allocated in 4KB pages. Hence, only the PTE-hECPT needs to be looked-up, and the PUD-hECPT and PMD-hECPT can be skipped.

While this optimization may not apply in the future, it is applicable to today's state-of-the-art systems. Specifically, in hypervisors such as KVM, host page tables are limited to using only 4KB page allocations [55]. Similarly, OS kernels only use 4KB pages for native page tables and, in virtualized environments, for guest page tables. One reason for this choice is the fact that the page tables for most processes are relatively small and, hence, using large pages would lead to significant memory waste. In addition, 4KB pages provide flexibility when main memory is fragmented, and can be allocated and initialized quickly. Finally and perhaps as importantly, legacy reasons have resulted in popular processors using only 4KB pages for page tables.

In our Advanced design, we assume that page tables use only 4KB pages and apply this optimization. As we will see in Section 9.1, this improvement has a minor impact.

4.4 Avoiding Stale hECPT Entries in the Plain and Advanced Designs

We point out a design decision that, because of its basic importance, we apply to both the Plain and the Advanced Nested ECPT designs. To understand it, consider nested radix page tables (Figure 2). There, caching the address translation of a level of the guest page table (gL_i) in the NTLB is beneficial. The closest parallel to NTLBs in Nested ECPTs would be to cache in the MMU the translation of hPTEs-to-gPTEs in Step ② of Figure 6. If this worked, one would eliminate one of the three sequential memory access steps of Nested ECPTs.

However, we find that this approach is not desirable. The reason is that, in Nested ECPT systems, the hPA of a gPTE changes often, for two reasons. First, due to cuckoo rehashing, inserting an entry in a gECPT may shuffle other gPTEs between the *d* ways of the gECPT. Second, due to the dynamic resizing of a gECPT, entries from the old gECPT are migrated to the new one. In either case, when the hPA of a gPTE changes, the hPTE that maintained the original pointer to the gPTE becomes stale. To avoid flushing such translations, neither the Plain nor the Advanced Nested ECPT design caches the mapping of hPTEs-to-gPTEs in Step ②.

5 OVERALL DESIGN

From now on, we refer to the Advanced design as simply the Nested ECPT design. Figure 7 shows the complete layout of the guest and host memories, and the modules in the MMU in Nested ECPTs. For simplicity, the three gECPTs, three hECPTs, three gCWTs, and three hCWTs are each combined into a single box. Further, in host memory, we only show the single relevant entry of the gECPTs (called gPTE in the figure) and of the gCWTs (called gCWT entry in the figure). This is because such tables may not be contiguous in host memory.

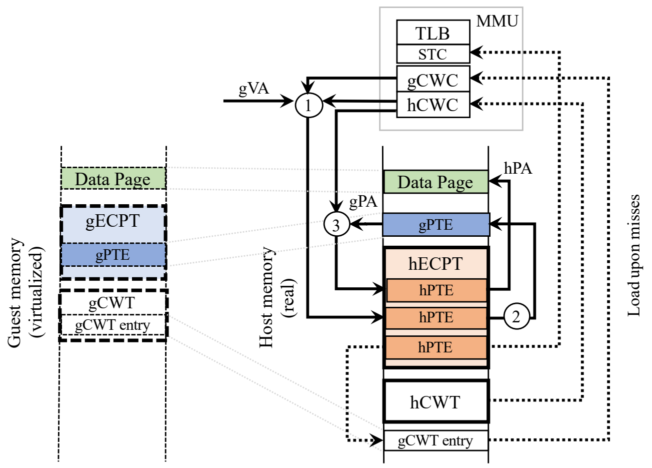

Figure 7: Layout of the guest and host memory, and the modules in the MMU with Nested ECPTs.

The solid arrows show the steps of a nested translation after a TLB miss. The steps are numbered as in Figure 6. In Step ①, the hardware takes the gVA, looks-up the gCWC and hCWC and, in the best case, issues a single memory access that reads an hPTE. In Step ②, the hardware uses the contents of the hPTE to issue a memory access that reads the gPTE with the target gPA. Finally, in Step ③, the hardware takes the gPA, looks-up the hCWC and, in the best case, issues a single memory access to read the hPTE that contains the target hPA. The pair {gVA, hPA} is loaded into the TLB.

The dashed arrows show the operations when CWCs miss. On an hCWC miss, an entry from the corresponding hCWT is loaded into the hCWC. On a gCWC miss, as described in Section 4.1, the hardware first finds the hPTE that contains the host physical address of the corresponding gCWT entry. Assume that this hPTE is the bottom-most hPTE in the figure. Then, the hardware loads this hPTE into the STC for fast translation in the future. Finally, the hardware uses this hPTE to access the target gCWT entry, which it loads into the gCWC.

6 MIGRATION PATH TO NESTED ECPTS

Nested ECPTs are a radical redesign of the page tables. To move current nested radix page table systems to Nested ECPTs, we propose two possible migration paths. One is for the machine to fully support both nested radix page tables and Nested ECPTs. At machine boot-up, a control register selects one of the two page table designs. This approach provides flexibility. However, it is clumsy and requires the hardware logic and MMU caching structures of the two approaches.

A more reasonable migration path is to use radix page tables in the guest OS and ECPTs in the host. This hybrid design supports legacy OS kernels. Thanks to the VM abstraction, the guest OS does not need changes, while the hypervisor is modified to support ECPTs for high performance.

Figure 8 shows a nested page walk in this *Hybrid Design*. We start from the design in Figure 2 and replace the four hL_i steps in each level of guest radix page table with Step ③ in Figure 4. This step translates the gPA of an entry in Level *i* of the guest radix page table to its hPA. As discussed in Section 3.2, this step tries to use the hCWC to obtain the target hPTE in a single memory access, as shown in Step ③ of Figure 6. Finally, once the gPA of the target data page is found in Step 8, a final Step ③ from Figure 4 is used to locate its hPA. A nested page walk now involves 9 sequential steps. As in the nested radix design, NTLBs can be used to eliminate these look-ups. NTLBs' operation is shown with dashed lines.

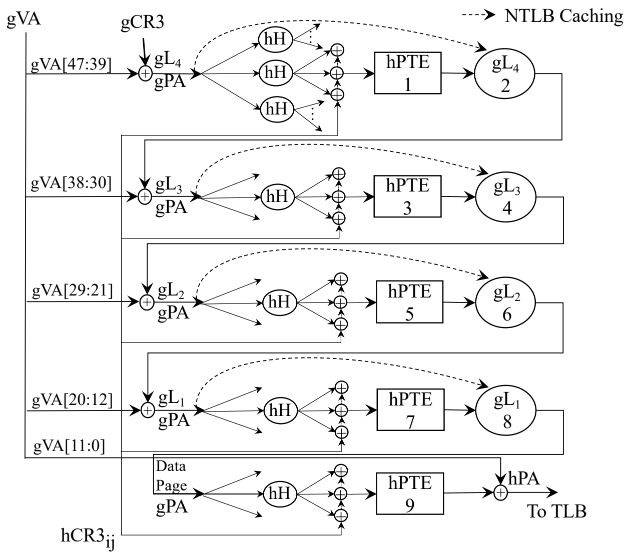

Figure 8: Nested Hybrid page walk: the guest OS uses radix page tables and the host uses ECPTs.

The hCWC used in each row of the walk is slightly different because the data locality of PTE-CWTs decreases as we move from top to bottom. Specifically, based on our experiments, the hCWC in the first and second rows from the top cache PUD-, PMD-, and PTE-CWTs. The hCWC in the third row caches PUD- and PMD-CWTs and, adaptively, PTE-CWTs. The hCWC in the fourth and fifth row only cache PUD- and PMD-CWTs.

This hybrid design replaces some of the sequential steps of radix page tables with the parallel steps of ECPTs. It is a design point that sits in between the two approaches. While this design is easy to use, it still largely requires the hardware resources of the two prior approaches: hCWCs, radix page walk caches, and NTLB caches. They all co-exist in the MMU.

7 OS/HYPERVISOR SUPPORT

This paper focuses on the architecture design of Nested ECPTs. Nested translations spend practically all of their time in hardware-assisted translation in user space, not in the OS or hypervisor software. The OS is invoked only in page faults, which are rare in the steady state of our applications.

For this reason, we can model and evaluate Nested ECPTs with a full-system simulator running KVM-based full-featured VMs on simulated hardware—without implementing ECPTs in Linux or KVM. Specifically, we use the Simics [59] full-system simulator and instrument the OS with the Intel SAE [22]. The simulator intercepts on-the-fly all the virtual memory operations of the KVM and guest OS (plus all the instructions executed). These operations are then processed by our back-end cycle-level processor/memory simulator. We leverage Simics to provide the actual memory and page table contents to the timing backend for each memory address of both the host and the guest. With modest modifications to Linux and KVM, this methodology allows us to implement and evaluate any page table organization in the simulator, while using the actual page table entries, support THP, and reflect any updates by guest and host. The high-level OS- and hypervisor-level memory management operations remain the same, as they are unaware of the underlying page table structures.

Our future work is to implement ECPTs in Linux and KVM. In this case, the ECPT designs, including both native and Nested ECPTs, will maintain the same interfaces as the radix page tables in the virtual memory system. Their support in the OS and hypervisor is relatively easy. Conceptually, only the page table implementations need to be replaced to support ECPTs.

In the Linux kernel, the majority of the changes required to support ECPTs are in the memory management code and under the page table handling [57]. Most of the page table functionality is handled in the kernel through a set of macros and functions that assist in allocating page tables and locating page table entries. Supporting ECPTs requires modifications to these functions. However, page table usage (e.g., checking the dirty bit) and modification operations (e.g., setting the present bit) remain the same. This is because these functions operate on a per page-table entry basis, which remains practically the same with ECPTs. Furthermore, since this functionality is reused by KVM and the guest OS, the support of ECPTs is naturally reused in Nested ECPTs. Overall, the changes added for ECPTs and Nested ECPTs are likely to be hidden behind a relatively small interface.

8 EVALUATION METHODOLOGY

Modeled Architectures. We use the Simics [59] full-system simulator integrated with the SST framework [10, 74] and the DRAMSim2 [75] memory simulator to model a server architecture with 8 cores and 80GB of main memory. We model the ten page table architecture configurations shown in Table 1. For both native and nested page tables, we model a system with radix page tables and a system with our Advanced ECPT design. We also model the Nested Hybrid Design of Section 6. In all cases, we model environments that only use 4KB pages and environments that also use 2MB pages, by enabling Transparent Huge Pages (THP) in Linux [84]. For the nested environments, we deploy a KVM VM that runs on the 8 cores of the host and can utilize 80GB of memory. The nested THP enables THP for both the host and the guest.

Table 1: Modeled page table architecture configurations.

Page Table Architecture		Description
Native	Nested	
Radix	Nested Radix	Radix page tables with only 4KB pages
Radix THP	Nested Radix THP	Radix page tables with 4KB+huge pages
ECPTs	Nested ECPTs	Advanced ECPTs with only 4KB pages
ECPTs THP	Nested ECPTs THP	Advanced ECPTs with 4KB + huge pages
—	Nested Hybrid	Hybrid Design with only 4KB pages
—	Nested Hybrid THP	Hybrid Design with 4KB + huge pages

The architecture parameters are shown in Table 2. Each core has private L1 and L2 caches. The L3 cache is shared and physically distributed. Cache misses are handled through Miss Status Handling Registers (MSHRs). Each core has private L1 and L2 TLBs, and 4 page table walkers. The radix page tables have a per-core page walk cache (PWC) and, when nested, they additionally have a per-core nested PWC (NPWC) and a per-core Nested TLB (NTLB). Table 2 shows the sizes of the guest and host structures in all the nested designs; the sizes in the native designs are the same as the guest ones in the nested designs. There is no PTE-gCWT in the nested design and no PTE-CWT in the native one because of the low locality of the data. Note that, in Nested ECPTs, we use separate hCWCs for Step ① and Step ③.

To be conservative, we sized the structures in the ECPT designs to make their total size strictly smaller than those in the radix designs. The MMU caches in Radix, ECPTs, Nested Radix, Nested ECPTs, and Nested Hybrid use 768, 672, 1680, 1488, and 1408 bytes, respectively. Table 3 reports the estimated area and power of these structures. For the measurements, we use Cacti [12] with 22nm technology. From the table, we see that these structures consume little area and power in all the designs.

Applications. We evaluate a variety of applications with different levels of TLB pressure. Table 4 shows the domain, the suite, the name, and the memory footprint for each application. For each application, we perform full-system simulations of all the different configurations evaluated. We instrument the applications to identify the region of interest. In that region, we warm-up the architectural state for 50M instructions, and then measure 500M instructions. Our simulation methodology is deterministic, producing the same result for every run that we start from a given checkpoint. For this reason, our plots in the next section do not show any error bars.

9 EVALUATION

9.1 Performance of Nested ECPTs

Figure 9 shows the speedup of the different architecture configurations of Table 1 over the *Nested Radix* configuration. The figure shows the results for each application and the geometric mean of all the applications. The native configurations are only shown in the mean bars. Recall that the Nested ECPTs and Nested ECPTs THP configurations are the *Advanced* design. To understand the performance contributions of our new techniques of Section 4, their bars are broken down into the effects of (i) STC, (ii) Step-① PTE-hCWT Caching, (iii) Step-③ PTE-hCWT Adaptive Caching, and (iv) Page Table Allocation in 4KB Pages. The rest of the bar is the speedup of the *Plain* Nested ECPT design of Section 3.

We focus first on the two nested configurations with only 4KB pages: Nested Radix (first bars) and Nested ECPTs (fifth bars). Nested ECPTs speeds-up the applications over Nested Radix by 1.04x–1.33x, with an average of 1.19x. The applications with the most speedup, like DC, MUMmer and SysBench, are typically those

Table 2: Architectural parameters used in the evaluation.

Processor Parameters	
Multicore chip	8 4-issue OoO cores, 128-entry ROB, 2GHz
L1 cache	32KB, 8-way, 2 cyc. round trip (RT), 64B line
L2 cache	512KB, 8-way, 16 cycles RT, 20 MSHRs
L3 cache	Slice: 2MB, 16-way, 56 cycles RT, 20 MSHRs
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
Radix Page Table Parameters	
Nested TLB (NTLB)	24 entries, fully associative (FA), 4 cycles RT
Page Walk Cache (PWC)	3 levels, 32 entries/level, FA, 4 cycles RT
Nested PWC (NPWC)	5 levels, 16 entries/level, FA, 4 cycles RT
Elastic Cuckoo Page Table (ECPT) Parameters	
Initial PTE gECPT/hECPT size	16384 entries × 3 ways
Initial PMD gECPT/hECPT size	16384 entries × 3 ways
Initial PUD gECPT/hECPT size	8192 entries × 3 ways
Initial PTE hCWT size	4096 entries × 2 ways
Initial PMD gCWT/hCWT size	4096 entries × 2 ways
Initial PUD gCWT/hCWT size	2048 entries × 2 ways
gCWC	16PMD + 2PUD entries, FA, 4 cycles RT
hCWC (in Step 1)	4PTE entries, FA, 4 cycles RT
hCWC (in Step 3)	16PTE + 4PMD + 2PUD, FA, 4 cycles RT
Shortcut Trans. Cache (STC)	10 entries, FA, 4 cycles RT
Hash functions: CRC	Latency: 2 cycles
Hybrid Design Parameters	
Initial PTE hECPT size	16384 entries × 3 ways
Initial PMD hECPT size	16384 entries × 3 ways
Initial PUD hECPT size	8192 entries × 3 ways
Initial PTE hCWT size	4096 entries × 2 ways
Initial PMD hCWT size	4096 entries × 2 ways
Initial PUD hCWT size	2048 entries × 2 ways
hCWC	16PTE(Rows 1-3)+16PMD+2PUD, FA, 4 RT
Page Walk Cache (PWC)	16 entries, FA, 4 cycles RT
Nested TLB (NTLB)	24 entries, FA, 4 cycles RT
Main-Memory Parameters	
Capacity; #Channels; #Banks	80GB; 4; 8
t_{RP}-t_{CAS}-t_{RCD}-t_{RAS}	11-11-11-28
Frequency; Data rate	1GHz; DDR
Host and VM Parameters	
Host OS; Guest OS	Ubuntu Server 16.04; Ubuntu Cloud 16.04
Hypervisor	QEMU-KVM

Table 3: Area and power of the hardware caches in the MMU.

Configuration	Size (B)	Area (mm^2)	Power (mW)
Nested Radix	1680	0.01	2.9
Nested ECPTs	1488	0.03	5.2
Nested Hybrid	1408	0.02	2.8

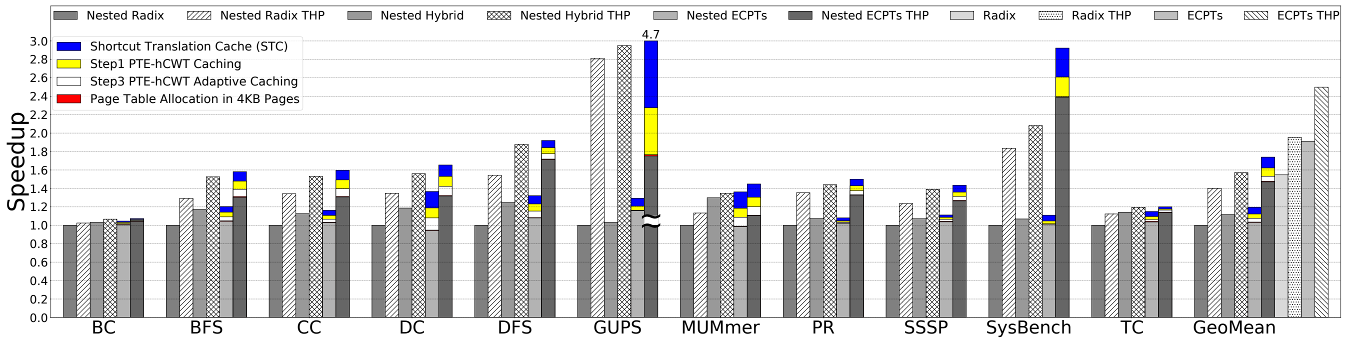

Figure 9: Speedup of the different architecture configurations over the *Nested Radix* configuration.

Table 4: Applications evaluated.

Application			Memory Footpr.
Domain	Suite	Name	
Graph analytics	GraphBIG [64]	Betweenness Centrality (BC)	17.3 GB
		Breadth-First Search (BFS)	9.3 GB
		Connected Components (CC)	9.3 GB
		Degree Centrality (DC)	9.3 GB
		Depth-First Search (DFS)	9.0 GB
		PageRank (PR)	9.3 GB
		Shortest Path (SSSP)	9.3 GB
		Triangle Count (TC)	11.9 GB
HPC	Challenge [58]	GUPS	64.0 GB
Bioinformatics	BioBench [5]	MUMmer	6.9 GB
Systems	SysBench [82]	SysBench	64.0 GB

where Nested Radix services relatively more page walk accesses from main memory compared to Nested ECPTs.

Consider now the two nested configurations with huge pages: Nested ECPTs THP (second bars) and Nested Radix THP (sixth bars). Huge pages improve performance significantly. This is especially the case for GUPS, which can exploit huge pages for the whole dataset, and SysBench. With huge pages, Nested ECPTs THP speeds-up the applications over Nested Radix THP even more: 1.05x–1.59x, with an average of 1.24x.

Overall, replacing radix page tables with ECPTs in a nested environment improves the performance across the board for all programs, often substantially.

Consider now the new techniques. Without any new technique, the average speedup of Nested ECPTs over Nested Radix is only 3% and 5% without and with THP, respectively. Hence, the new techniques are responsible for most of the speedups of Nested ECPTs. From Figure 9, without THP, the average speedup contributions of STC, Step-① Caching, Step-③ Adaptive Caching, and 4KB Page Allocation are 6.8%, 4.6%, 4.2%, and 0.4%, respectively. With THP, the speedup contributions are 7.9%, 6.5%, 4.1%, and 0.5%. Clearly, the first three techniques have a substantial impact. The high effectiveness of STC is due, in part, to the fact that it reduces the number of MMU-initiated L2 misses by 17%.

The fourth technique does not help the steady-state of the applications much. However, it speeds-up page walks during the warm-up, when walks are more expensive: the lack of cached information causes a walk to access ECPTs for *all page sizes*. Our technique minimizes this effect. While not shown in the figure, if we include the warm-up period, 4KB Page Table Allocation speeds-up the 95th percentile tail latency of the page walks for the whole application by an average of 9.4% (no THP) and 8.9% (with THP). So, it is also important.

Consider now the Nested Hybrid design. For all the applications, Nested Hybrid performs better than Nested Radix but worse than Nested ECPTs. On average, Nested ECPTs is 7% and 11% faster than Nested Hybrid for 4KB pages and THP, respectively. Still, on average, Nested Hybrid outperforms Nested Radix by 12% and 13% for 4KB pages and THP, respectively. The results highlight the significant performance improvement attained by only migrating the host to ECPTs.

For reference, the figure also shows the speedups of the native configurations, relative to Nested Radix. As expected, the native configurations are generally faster than the nested ones, since they do less work. However, there are a few exceptions where the nested designs with huge pages deliver higher speedups than the native ones without huge pages. This effect occurs in applications where huge pages are highly useful, such as in GUPS, SysBench, and DC. We have verified some of this behavior with real-systems measurements. Due to these applications, we see that the average speedup of Nested ECPTs THP over Radix is 1.11x.

To gain further insight, Figure 10 shows the number of MMU busy cycles in Nested Radix and Nested ECPTs, normalized to Nested Radix. These are cycles when the MMU is busy servicing L2 TLB misses, including when the MMU is waiting for its outstanding memory requests. The figure shows that Nested ECPTs designs uniformly spend substantially fewer cycles in translation than Nested Radix designs. On average, Nested ECPTs use 25% and 31% fewer MMU busy cycles than Nested Radix for 4KB-only and THP. The per-application results in Figure 10 are only weakly correlated with Figure 9, since Figure 10 shows normalized cycle counts.

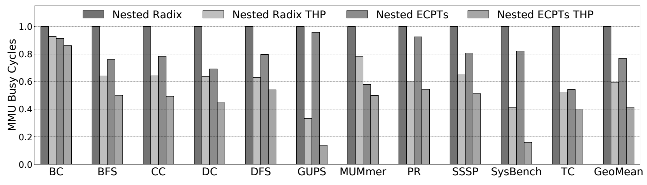

Figure 10: MMU busy cycles in nested configurations.

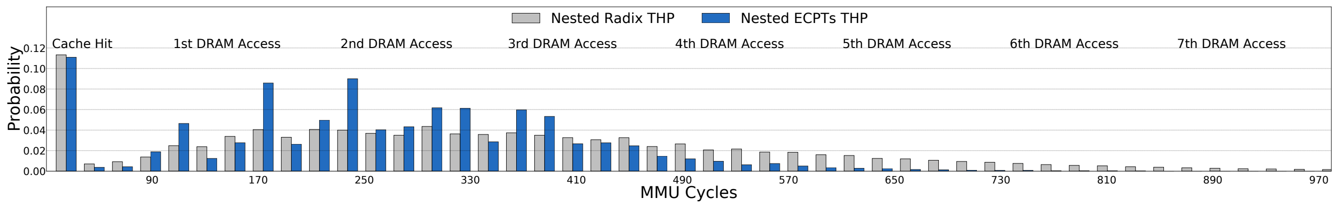

Figure 11: Histogram of the latency of the nested page walks in the MUMmer application.

9.2 Impact of Adaptive PTE hCWT Caching

Recall that, to support adaptive caching of PTE hCWT entries in Step ③ of Figure 6, we measure the hit rates of both PMD hCWT and PTE hCWT entries in the hCWC. Figure 12 shows these values for our applications. The left chart shows that, in all the applications except GUPS and SysBench, PTE hCWT entries enjoy a very high hit rate in the hCWC. Therefore, the applications can benefit from enabling PTE hCWT caching. The right chart shows that, in GUPS and SysBench, PMD hCWT entries have a lower hit rate in the hCWC than in other applications. Based on this analysis, we define two thresholds (the dashed lines): if the hit rate of PTE hCWT entries is below 0.5, we disable PTE hCWT caching; when PTE hCWT caching is disabled and the hit rate of PMD hCWT entries is above 0.85, we enable PTE hCWT caching.

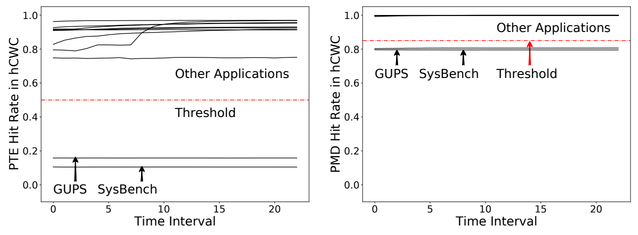

Figure 12: Hit rate of the PTE hCWT entries (left) and PMD hCWT entries (right) in the hCWC. An interval is 5M cycles.

9.3 MMU and Cache Characterization

The performance gains provided by Nested ECPTs are due to two effects. The first one is its ability to issue memory accesses in parallel when performing the nested page translation—even if it issues more accesses than Nested Radix. The second effect is that Nested ECPTs only fetch into the caches actual translations, building-up useful state in the memory hierarchy. This is in contrast to Nested Radix, which fetches many intermediate translation entries during a walk. These entries cause pollution in the memory hierarchy.

To understand these effects, Fig. 13 characterizes the behavior of the MMU, L2 cache, and L3 cache for the nested environments. Starting from the top, Figure 13(a), shows the number of requests that the MMU issues to the cache hierarchy per Kilo instruction (RPKI). In Nested Radix, these requests are those issued to obtain a translation, while in Nested ECPTs, they are those that request translations and those that request hCWT/gCWT entries. We can see that the ECPT configurations issue more requests—on average 13% and 15% more for 4KB pages and THP, respectively. However, many of these accesses are issued in parallel.

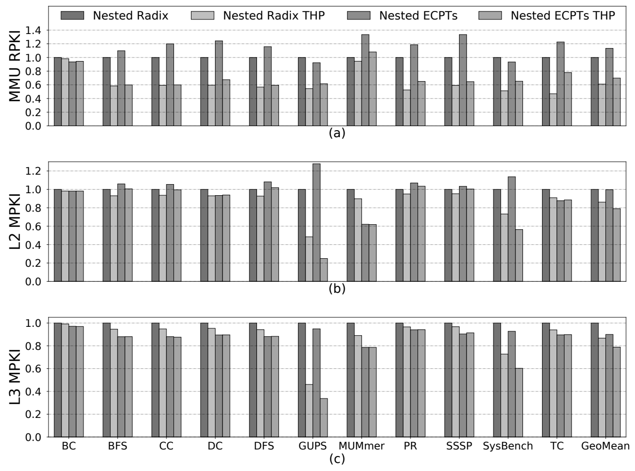

Figure 13: Characterizing the MMU and cache subsystem.

Figures 13(b) and 13(c) show the normalized misses PKI (MPKI) in the L2 and L3 caches, respectively, for the same configurations. These misses are initiated by both the MMU and the processor. Across configurations, the differences are mostly caused by the requests issued by the MMU. We see that the requests issued by the MMU in Nested ECPT designs enjoy higher cache locality than those in Nested Radix: in the L2, both designs have similar average MPKI; in the L3, Nested ECPTs has a 10% and 11% lower MPKI than Nested Radix for 4KB-only pages and THP, respectively. Therefore, while Nested ECPTs issue more MMU requests, they end up issuing fewer main memory accesses. As indicated above, a major reason is that they only fetch actual translations, while Nested Radix fetch intermediate translations, polluting caches.

The L2 and L3 miss patterns in Nested ECPTs are not any more bursty or demanding than in Nested Radix. On average, with Nested ECPTs, the L2 and L3 use 4.4 and 3.8 MSHRs, respectively, at a time. The same numbers for Nested ECPTs THP are 4.2 and 3.6. The maximum number of MSHRs in use in L2 or L3 is 12.

To shed additional light, Figure 11 shows a histogram of the latency of the nested page walks in the MUMmer application for Nested Radix THP and Nested ECPTs THP. For each page walk, we measure the latency in cycles from when the L2 TLB miss occurs until the page walk completes. Then, we group the page walks in bins according to their latency. The figure is annotated with the ranges of latencies for caches, 1st DRAM access, 2nd, 3rd, and so on. We see from the figure that Nested Radix THP exhibits a long tail of page walks of several hundreds of cycles. This is because of the sequential pointer chasing process that Nested Radix imposes.

In contrast, Nested ECPTs THP page walks are typically over with a latency equivalent to about four DRAM accesses.

9.4 Characterizing Nested ECPT Walks

When the hardware accesses the hCWC or gCWC in a nested ECPT walk (Figure 6), it may issue from 1 to $n \times d$ accesses. The paper that introduced ECPTs for native environments [79] used a naming convention to refer to the different possible outcomes: *Direct Walk* if it issues 1 memory access, *Size Walk* if it accesses all the d ways of one ECPT, *Partial Walk* if it accesses at worst all the ways of two ECPTs, and *Complete Walk* if it accesses all the d ways of all the n ECPTs. Outcomes with few accesses are preferred.

Figure 14 shows the relative frequency of these types of outcomes for Nested ECPTs THP using the complete mappings of the applications. For each application, the left bar is the information provided by the hCWT, while the right bar is the information provided by the gCWT. The gCWT aims to trim the first part of Step ① of the walk as shown in Figure 4. The hCWT prunes the second part of Step ① as well as Step ③.

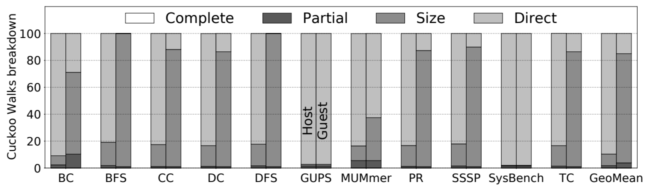

Figure 14: Breakdown of the types of host (left bar) and guest walks (right bar) for each application in Nested ECPTs THP.

Figure 14 shows that, in the host, the majority of the walks are the very cheap direct walks (90% on average). Direct walks are common because the hypervisor frequently uses huge pages. In the guest, the majority of the walks are size walks (82% on average). The exceptions are GUPS, SysBench and MUMmer, where huge pages are very effective and, therefore, direct walks dominate. Complete walks are negligible for both guest and host.

Overall, most of the information obtained from CWTs results in outcomes with few accesses. It can be shown that, on average for our applications, a Nested ECPT walk (Figure 4) with THP issues 2.8 parallel memory accesses in Step ①, 2.8 in Step ②, and 1.6 in Step ③. Without THP, only Step ③ changes: it has slightly more accesses on average, namely 1.7.

Finally, we measure the average hit rates of MMU caches for Nested ECPT THP. The hit rate of our 10-entry STC is 99%. If we reduced its size to 8 and 4 entries, its hit rate would be ≈90% and ≈50%, respectively, which are too low. In the gCWC, the hit rates are 99% for its PUD entries and 86% for its PMD entries. In the hCWC, the hit rates are 99% for its PUD entries, 80% for its PMD entries and, for its PTE entries, 99% in Step ① and 67% in Step ③. Overall, gCWC and hCWC effectively reduce the number of accesses issued by nested ECPTs walks.

9.5 Memory Consumption

On average across all the applications, the memory required to hold all the page table entries is 60MB. This number is computed by multiplying the number of page table entries by 8 bytes, and therefore is independent of the page table organization chosen. However, when we measure the memory used by all the virtual memory structures, the number is higher, due to various structure overheads: 84MB for Nested Radix (of which 56MB are for host and 28MB for guest structures) and 97MB for Nested ECPTs (of which 61MB are for host and 36MB for guest structures). Overall, Nested ECPTs only use slightly more memory than Nested Radix.

9.6 Comparison to Other Advanced Designs

Agile Paging [30] combines nested and shadow paging [85] by leveraging the idea that the page table entries of upper levels in the radix tree are unlikely to be changed. However, Agile Paging still requires 4 sequential memory accesses in the best case scenario, as well as some hypervisor intervention cost. We simulate an ideal Agile implementation with at most 4 sequential memory requests, all the caching structures of radix, and no hypervisor costs. Nested ECPTs outperform this ideal Agile Paging design by 16% on average.

POM-TLB [76] is a large TLB that is part of memory. Although the design eliminates a significant portion of page table walks, an L2 TLB miss may be propagated to the POM-TLB in DRAM and, on a miss, still require a page walk. We simulate the POM-TLB design with a perfect page size predictor. On average, nested ECPTs outperform POM-TLB by 14%.

Flat nested page tables [3] combine a guest radix page table with a host flat page table. The design reduces the maximum number of sequential memory references from 24 to 9. We simulate this design and find that Nested ECPTs outperforms flat nested page tables by 12% (no THP) and 15% (THP) on average. The limitation of flat nested page tables is the potentially up to 9 sequential memory accesses.

10 OTHER RELATED WORK

A number of studies have measured the overhead of nested page table walks for virtualized memory translation [4, 6, 21, 29, 38, 72, 76]. To reduce TLB misses, prior work has proposed new TLB designs with support for clustering, coalescing, entry-sharing, speculation, multiple page sizes, and prefetching [11, 14, 19, 20, 24, 34, 37, 51, 61, 63, 68–71, 77, 78, 81, 88]. Furthermore, a few virtualization-specific TLB designs have been proposed, including large part-of-memory TLBs [76], context-aware TLBs [60], and TLB designs with virtualization support [23].

To reduce nested page walk overhead, MMU caches [9, 16], devirtualized memory [39], application-managed memory translation [4], and optimized huge page support [56, 66, 67] have been proposed.

Moreover, other designs have been proposed to reduce the memory references required for nested page walks by exploiting virtual and physical address space contiguity [6, 15, 29, 30, 52, 53]. These approaches create translations that map very large contiguous regions of virtual memory to contiguous physical memory. As a result, the number of required translation entries reduces, potentially lowering overhead. While promising, these approaches face the challenge that creating very large contiguous physical address spaces in actively-used cloud platforms is hard. Indeed, even finding the more modest 2MB-sized pages supported by Linux Transparent Huge Pages (THP) is often hard [31, 36, 56]. Going beyond them is harder. One important characteristic of Nested ECPTs is that it does not rely on the need for physical memory contiguity.

11 CONCLUSION

This paper presented the first page table design for parallel nested address translation. The design, called Nested Elastic Cuckoo Page Tables (Nested ECPTs), eliminates all but three of the potentially twenty-four sequential steps of a nested radix page table translation—while judiciously limiting the number of parallel memory accesses issued to avoid over-consuming cache hierarchy bandwidth. As a result, compared to conventional nested radix tables, Nested ECPTs speed-up the average execution time of a set of applications by 1.19x (for 4KB pages) and by 1.24x (when huge pages are used). In addition, we described a possible migration path from current systems to Nested ECPTs.

ACKNOWLEDGMENTS

This work was supported by NSF under grant CNS 1956007 and CNS 2107307. We thank Dan Tsafrir for his valuable feedback.

REFERENCES

[1] Keith Adams and Ole Agesen. 2006. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications . In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*.

[3] 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)*.

[4] 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)*.

[5] 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)*.

[6] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2020. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA'20)*.

[7] Amazon Web Services. 2021. Elastic Compute Cloud (EC2). https://aws.amazon.com/ec2.

[8] AMD. 2005. AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual.

[9] AMD. 2008. AMD-V^{TM} Nested Paging. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-11-final-TM.pdf.

[10] 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.

[11] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. 1994. Software Prefetching and Caching for Translation Lookaside Buffers. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI'94)* (Monterey, California).

[12] 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).

[13] 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)*.

[14] 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)*.

[15] 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)*.

[16] 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)*.

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

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

[19] 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)*.

[20] 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)*.

[21] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and H. Reza Taheri. 2013. Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications. *VMWare Technical Journal* (2013).

[22] 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)*.

[23] Xiaotao Chang, Hubertus Franke, Yi Ge, Tao Liu, Kun Wang, Jimi Xenidis, Fei Chen, and Yu Zhang. 2013. Improving Virtualization in the Presence of Software Managed Translation Lookaside Buffers. In *Proceedings of the 2013 International Conference on Computer Architecture (ISCA'13)*.

[24] 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)*.

[25] Christoffer Dall and Jason Nieh. 2014. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

[26] 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)*.

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

[28] 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.

[29] 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)*.

[30] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*.

[31] 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)*.

[32] Google. 2021. Cloud Compute Engine. https://cloud.google.com/compute.

[33] Google. 2021. gVisor: Container Runtime Sandbox. https://gvisor.dev/docs/.

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

[35] 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)*.

[36] 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 International Conference on Virtual Execution Environments (VEE'15)*.

[37] F. Guvenilir and Y. N. Patt. 2020. Tailored Page Sizes. In *Proceedings of the 2020 47th Annual International Symposium on Computer Architecture (ISCA'20)*.

[38] Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo F. Oliveira, Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu. 2020. The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA'20)*.

[39] 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)*.

[40] 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)*.

[41] 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.

[42] Intel. 2005. Intel Virtualization Technology Specification for the IA-32 Intel Architecture.

[43] 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.

[44] 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.

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

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

[47] Intel. 2021. Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[48] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv:1903.05714* (2019).

[49] 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)*.

[50] 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).

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

[52] 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)*.

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

[54] Samuel T. King, George W. Dunlap, and Peter M. Chen. 2003. Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC'03)*.

[55] KVM. 2021. Page Table Allocation in KVM. https://git.kernel.org/pub/scm/virt/kvm/kvm.git/tree/arch/x86/mm/init_64.c#n224.

[56] 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)*.

[57] Linux Kernel. 2021. Page Table Header File. https://github.com/torvalds/linux/blob/master/include/linux/pgtable.h.

[58] 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)*.

[59] 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).

[60] 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 IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*.

[61] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*.

[62] Microsoft Azure. 2021. Cloud Computing Services. https://azure.microsoft.com.

[63] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jimenez. 2020. CHiRP: Control-Flow History Reuse Prediction. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*.

[64] 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)*.

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

[66] 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)*.

[67] 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)*.

[68] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. 2020. Perforated Page: Supporting Fragmented Memory Allocation for Large Pages. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*.

[69] 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)*.

[70] 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)*.

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

[72] Binh Pham, Ján Veselý, 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)*.

[73] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*.

[74] 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)*.

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

[76] 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)*.

[77] 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)*.

[78] D. Skarlatos, U. Darbaz, B. Gopireddy, N. S. Kim, and J. Torrellas. 2020. BabelFish: Fusing Address Translations for Containers. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA'20)*.

[79] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[80] J. E. Smith and Ravi Nair. 2005. The Architecture of Virtual Machines. *IEEE Computer* 38, 5 (2005), 32–38. https://doi.org/10.1109/MC.2005.173

[81] 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)*.

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

[83] 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)*.

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

[85] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*.

[86] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 496–508. https://doi.org/10.1109/MICRO50266.2020.00049

[87] www.7-cpu.com. 2021. Intel Skylake Timing. https://www.7-cpu.com/cpu/Skylake.html.

[88] 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)*.

[89] 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)*.