Every Walk's a Hit: Making Page Walks Single-Access Cache Hits

Chang Hyun Park
Uppsala University
Uppsala, Sweden
chang.hyun.park@it.uu.se

Ilias Vougioukas
Arm Research
Cambridge, United Kingdom
Ilias.Vougioukas@arm.com

Andreas Sandberg
Arm Research
Cambridge, United Kingdom
andreas.sandberg@arm.com

David Black-Schaffer
Uppsala University
Uppsala, Sweden
david.black-schaffer@it.uu.se

ABSTRACT

As memory capacity has outstripped TLB coverage, large data applications suffer from frequent page table walks. We investigate two complementary techniques for addressing this cost: reducing the number of accesses required and reducing the latency of each access. The first approach is accomplished by opportunistically "flattening" the page table: merging two levels of traditional 4 KB page table nodes into a single 2 MB node, thereby reducing the table's depth and the number of indirections required to traverse it. The second is accomplished by biasing the cache replacement algorithm to keep page table entries during periods of high TLB miss rates, as these periods also see high data miss rates and are therefore more likely to benefit from having the smaller page table in the cache than to suffer from increased data cache misses.

We evaluate these approaches for both native and virtualized systems and across a range of realistic memory fragmentation scenarios, describe the limited changes needed in our kernel implementation and hardware design, identify and address challenges related to self-referencing page tables and kernel memory allocation, and compare results across server and mobile systems using both academic and industrial simulators for robustness.

We find that flattening does reduce the number of accesses required on a page walk (to 1.0), but its performance impact (+2.3%) is small due to Page Walker Caches (already 1.5 accesses). Prioritizing caching has a larger effect (+6.8%), and the combination improves performance by +9.2%. Flattening is more effective on virtualized systems (4.4 to 2.8 accesses, +7.1% performance), due to 2D page walks. By combining the two techniques we demonstrate a state-of-the-art +14.0% performance gain and -8.7% dynamic cache energy and -4.7% dynamic DRAM energy for virtualized execution with very simple hardware and software changes.

CCS CONCEPTS

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

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507718

KEYWORDS

Flattened page table, page table cache prioritization

ACM Reference Format:
Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. Every Walk's a Hit: Making Page Walks Single-Access Cache Hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507718

1 INTRODUCTION

The problem: Traditional page walks do not scale well with large data sets. While memory capacity has grown by 100× over the past decade, TLB sizes have merely tripled to around 1500 L2 TLB entries, delivering a reach of only 3 GB with 2 MB large pages. As a result, applications that use the large amount of available physical memory often suffer from significant numbers of TLB misses and resulting page walk delays. Virtualized environments see an even larger penalty for the 2D page walk to translate each level of the guest page walk on the hypervisor side [17]. This problem will be exacerbated with 5-level page tables for larger memories [31].

Background: Traditional page table trees have significant overheads on today's systems. Traditional page tables were designed with the assumption that memory is managed in contiguous blocks of exactly one page. The practical implication of this is that nodes in the page table are the same size as a page, leading to a multi-level deep tree of 4 KB page table nodes, each mapped to its own 4 KB memory page. Having each page table node be one memory page vastly simplifies allocation as the operating system does not have to maintain a separate reserve of larger blocks to guarantee it can allocate larger page table nodes. Keeping the nodes small also avoids wasting memory due to fragmentation. However, the small size of the page table node allocations combined with today's large memory capacities results in a deep page table tree, which incurs multiple serial memory indirections on each page table walk.

Solution: Flatten the page table to reduce indirections and prioritize caching the page table to reduce latency. While there have been many proposals to increase the effective TLB coverage to avoid page walks, we instead seek to reduce the cost of page walks. First, we reduce the number of architectural memory accesses in a page table walk (typically 4 without virtualization or 24 with virtualization enabled). This is done by *flattening the page table* to reduce the number of indirections required for a page walk by using

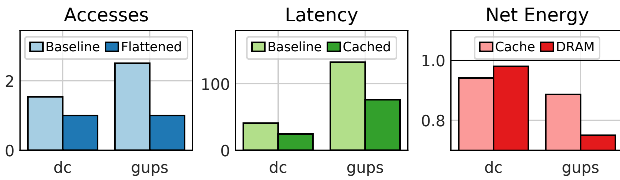

Figure 1: Left: reduction in memory requests per page walk from *flattening the page table.* Center: reduction in page walk latency from *prioritizing caching page table entries.* Right: reduction in dynamic energy of the cache hierarchy and DRAM from flattening and prioritizing. For benchmarks with high/low (gups/dc) TLB miss rates

large pages to store more entries in each page table node. Second, we reduce the latency of the accesses by increasing the effectiveness of on-chip caches for the page table. This is achieved by *preferentially caching page table entries* during periods of high TLB misses and low data reuse. Together, we reduce the number of memory accesses per translation to 1 (non-virtualized) or 3 (virtualized), and reduce average walk latencies from 50.9 cycles down to 29.1 cycles.

Flattening the page table is based on two observations. First, the radix nature of the page table tree structure allows two levels of standard (4 KB) page table nodes to be combined into one level of large (2 MB) nodes. This reduces the tree's depth and the number of indirections required to walk the tree, while leveraging existing 2 MB page support. And, second, the page table itself is so much smaller and more static than the actual data, that the bloating [34, 42] and variable latency [34, 41] problems that plague 2 MB pages for data are not significant. To take advantage of these observations, we flatten the page table using large pages for nodes. To ensure that the OS is able to reliably allocate page table nodes, we make flattening optional on a per page table node basis. Through a prototype, we find that the OS changes required to implement this new arrangement are small.

Concretely, for an 8 GB application we can reduce the number of pages in the ≈16 MB page table from 4106 4 KB pages with a standard 4-level table to only nine 2 MB pages in a flattened 2-level table, thereby reducing the number of indirections required on TLB misses from 4 to 2. While this incurs a ≈2 MB overhead (9×2 MB vs. 4106×4 KB), the cost is negligible for large applications because the total size of the page table is so small (18 MB vs. 8 GB data).

Prioritizing caching page table entries is also based on two observations. First, the page table itself is close to the size of the LLC, and, second, high TLB miss rates are correlated with high data miss rates. This implies that we can expect to keep most of the page table in the cache and that doing so will not significantly hurt data access latency. We find that preferentially keeping page table entries in the cache during phases of high TLB miss rates is a simple and effective way to reduce the latency of page walk accesses, and, when combined with flattening, we can achieve single-access cache hits for most page walks.

Context: Page Walker Caches are excellent. Page walk caches (PWCs) already reduce the theoretical 4 memory system accesses per page walk to < 1.5 on average (max 2.5 on our random access benchmark), and from 24 to 4.4 for virtualized systems. Flattening reduces this to 1 (2.8 virtualized), while cache prioritization reduces the latency of each access.

Contributions. The impact of our approach for applications with relatively few TLB misses (dc) and many (gups) is shown in Figure 1. While flattening significantly reduces the number of memory requests per page walk (left), by itself it has limited performance benefit. However, for virtualized systems, and with realistic memory fragmentation, its impact increases significantly (Section 4), as they have more complex page walks and more pages in the their page tables. Prioritization significantly reduces latency (middle) by avoiding most DRAM accesses for page walks and does so without significantly hurting the overall cache performance (Section 5). When combined, the approaches beat the state of the art for performance and deliver significant reductions in dynamic cache and DRAM energy (right). Our contributions:

- We identify, evaluate, and combine complementary approaches for reducing the impact of page walks: flattening to reduce the number of accesses and cache prioritization to reduce their latency.
- We identify and quantify the importance of handling large allocation failures in the kernel on real systems.
- We demonstrate the flexibility to dynamically choose where in the page table to flatten to efficiently support large data pages and evaluate its impact across three fragmentation scenarios.
- We quantify the benefits for virtualized systems and explore the trade-offs in flattening the page table for the host, guest, or both in virtualized systems.
- We show that flattened tables are not naturally compatible with recursive page tables and provide an efficient dereferencing solution.
- We demonstrate the limited OS changes required by reporting on the code changes for a Linux implementation of a flattened page table.
- We present simulations from server and mobile system on academic and industrial simulators for robustness.

2 RELATED WORK

There has been a tremendous amount of work aimed at improving translation range and efficiency (and thereby reducing the number of page walks) [8, 16, 21, 23, 25, 27, 33, 33, 37, 38, 42–46, 53]. Other works have focused on reducing the TLB miss penalty by improving the page table walk caches [14, 17, 18], using speculation to hide latency [5, 8, 15, 47], optimizing hash page tables [52], and replicating page tables across NUMA nodes [3]. For virtualized systems, Gandhi *et al.* proposed merging the 2D page table into a single dimension where possible [26]. Ahn *et al.* proposed flat host page tables for precisely the virtual machine memory size [5], which will perform similarly to our host page table flattening without the benefits of our guest-flattening. Ausavarungnirun *et al.* proposed bypassing the shared cache for the lower levels of the tree to avoid pollution in throughput-oriented GPU systems [13]. Mazumdar *et al.* proposed predicting dead TLB entries and dead page table entries in the LLC [37]. This could be used to extend our cache prioritization, although we have not investigated this. Below we

discuss four works that seek to directly reduce the cost of page walks.

Ryoo *et al.* proposed POM_TLB [48] to use a part of the DRAM as a large set-associative TLB. This requires a single memory lookup, and the entries can be cached. On a miss in the in-DRAM TLB, a conventional page table walk is required. Because this space is allocated at system boot, the required large contiguous partition can be guaranteed. This approach has the benefit of not requiring any OS changes, but comes at the cost of the complexity of scanning the structure at address space teardown, and possible interference or security implications as the on-DRAM TLB is shared by all cores, processes, and VMs. Marathe *et al.* extended POM_TLB with a cache prioritization extension (CSALT [35]), to keep page table entries in the cache during frequent context switches.

Margaritov *et al.* proposed ASAP [36], which prefetches the lower levels of the page table during the time spent accessing the higher levels. However, as modern PWCs effectively eliminate the accesses to the higher levels, our simulations show very little opportunity for such prefetching. ASAP requires that appropriate pages are stored sequentially in memory to enable prefetching without pointer chasing. This requires that the kernel can allocate contiguous segments of memory for page table entries for prefetching to function, which is difficult to guarantee. If such regions are not available, ASAP is unable to prefetch.

Elastic Cuckoo Hashing [49] restructures the page table into a hash table to generate lookup addresses without pointer chasing. ECH uses multiple hashed regions for parallel lookups, which requires the OS to allocate large contiguous blocks upon creating or resizing the page table. Since the OS cannot guarantee large contiguous memory allocations, this would make it difficult to implement in practice.

Compendia [6] is a parallel work that addressed flattening the page table. They claim roughly twice the performance benefits that we observed. However, their methodology uses saved page walk cycle estimates to compute performance change rather than simulation, and does not include overlapping page walks, a data cache hierarchy, or realistic memory fragmentation [42, 54], which may account for the differences. Further, they do not claim a kernel prototype, address self-referencing page tables, or compare to previous proposals.

Flattening alone achieves comparable main memory access reductions to what ECH could achieve (with its way-caching), but without the complexity of dynamically resizing the hash table, is better than what POM_TLB could achieve, as its cache does not cover the full page table, and is simpler than ASAP, as the layout modification leverages existing large page support. Importantly, flattening provides a graceful fallback to 4 KB pages when larger contiguous allocations fail, which our prototype kernel shows happens on heavily-loaded systems (Section 6.2). As a result, proposals that require large contiguous allocations to function (such as ECH) face severe implementation challenges. Our final proposal to combine flattening and cache prioritization goes beyond CSALT's prioritization as it does not require a separate POM_TLB cache to achieve similar results, and improves on Compendia in both energy and performance by coordinating with the memory hierarchy.

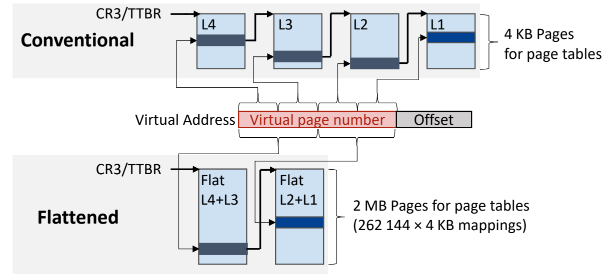

Figure 2: Conventional 4-level page table (top) and our proposed 2-level flattened page table (bottom). We utilize 2 MB pages for the flattened L4+L3 and flattened L2+L1 page table levels, however flattening can be applied using many combinations of natively supported page sizes.

3 FLATTENING THE PAGE TABLE

Page tables are organized as trees that entail a series of memory indirections for each page walk. These pointer-chasing accesses lead to long latencies to satisfy TLB misses. We can reduce the number of levels in the tree (flatten it) to reduce the number of indirections by using larger nodes in the tree. When combined with modern page walker caches (PWCs), this achieves effective page walks of a single memory access. In Section 5 we will add preferential caching to make this single access a cache hit.

3.1 Conventional Page Tables

We consider conventional 64-bit x86 and Arm page tables, which use a page size of 4 KB and entry size of 8 B, leading to a 512-ary radix tree. However flattening can be applied to other configurations. Figure 2 illustrates address translation pointer chasing through the page table tree for conventional (4 KB nodes, top) and one flattened (2 MB nodes, bottom) page table configuration. The conventional tree has 512 entries per level, with each 9-bit segment of the address used to index within each level[1], and the address of the first page table node in the CR3 (x86) or TTBR (Arm) register.

While an L1 (leaf) page table entry represents a 4 KB virtual memory region, each entry in higher-level nodes can either point to a lower-level node or directly represent a translation of a larger region: e.g., L2 entries can either point to an L1 node of 512 4 KB translations or directly translate a contiguous 512×4 KB = 2 MB region. Thus the page table elegantly supports large pages by tracking whether each entry is a pointer to a lower-level node or direct translation. Some TLB designs leverage this fact to store partial walks and final translations in the same HW structure (e.g. L2 TLB) [9, 12].

3.2 Flattening the Page Table

Flattening uses the radix nature of the page table to naturally merge levels of the page table into single, larger levels, resulting in a shallower tree. Merging two levels results in page table levels that naturally use the next larger page size. For example, consider merging the L2 and L1 levels in the page table. As each 4 KB L2 node points to 512 4 KB L1 nodes, and each L1 node provides 512 translations, if we flatten the L2 and L1 levels, we would replace each L2

[1]We label the page table L4, L3, L2 and L1 from root to leaf.

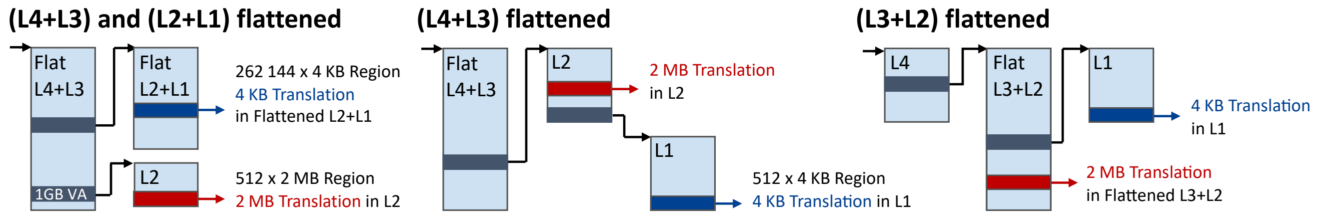

Figure 3: Flattening the L4+L3 and L2+L1 levels (left): 4 KB pages (flattened L2+L1) require only two accesses, while 2 MB pages are mapped to 1 GB VA ranges that do not flatten their L2+L1 (unflattened L2). Other approaches may flatten the first two levels (middle, L4+L3) or the middle two levels (right, L3+L2). Our prototype OS implementation targets flattening L3+L2 (right) while our evaluations look at flattening both L4+L3 and L2+L1 (left).

node and its 512 L1 child nodes with a single 2 MB flattened node, which covers all 512×512 (262 144) translations (Figure 2, bottom). If all translations in the region are mapped, this approach saves the 4 KB of space needed to hold the original L2 node. However, if some L1-sized regions of the translations are not mapped, this approach will waste 4 KB of space for each such region. In the traditional page table, L1-sized regions that are not used would not have allocated 4 KB page table nodes, thereby saving memory. This bloating in the page table itself is a side-effect of flattening the page table, but as the page table itself is a minuscule fraction of the size of the actual data (roughly 1/512th the size with regular pages and a contiguous VA space), the overhead is negligible.

The choice of which page table levels to flatten is flexible. In this work, we use the design shown in the bottom of Figure 2 where we flatten the first two (L4+L3) and the latter two (L2+L1) levels into 2 MB page table nodes. However, one could flatten L3+L2 instead[2], which would be beneficial in the presence of 2 MB data pages (shown in Figure 3, right), or possibly even flatten the top three levels (L4+L3+L2) using a 1 GB page table node that points to 2 MB data pages directly. Alternatively, one could flatten the top two levels (L4+L3) using 2 MB page table nodes and allocate all memory in 1 GB data pages. These approaches all have different trade-offs in terms of bloating within the page table itself and the ease of creating sufficiently large allocations. The choice of which levels to flatten can be made on a per-process and per-table basis at runtime by the OS, with or without application hints.

Because they are based on the radix nature of the page table, flattened page tables provide a key characteristic for practical implementation: graceful fallback to conventional page tables when needed. If the OS is unable to allocate a 2 MB page for a flattened page table node, it can instead allocate the standard two levels of 4 KB page table nodes at any place in the page table with no additional overhead. This allows the OS to choose whether to take the time at allocation for compaction, merge the two levels at a later point when a 2 MB page is available, or simply use the standard two-level approach.

Only small changes are needed to support flattened page tables. The system architecture needs to be able to inform the hardware page walker of which nodes are flattened with one bit in one of the control registers for the root and one bit in each page table entry (two bits to support 1 GB flattened nodes). The hardware page table walker can then read those bits to determine which parts of the virtual address should be used for index bits at each level. We describe the changes in more detail in Section 6.1.

In order to use the flattened page tables, the encoding of the VA bits that are used for the indexing of the page tables needs to be adjusted. Traditionally for a 4-level tree structure every page table has $2^9 = 512$ entries. This means that the VA is decoded using 9 bits to index into the page tables and 12 bits which are used for the offset once the translation is complete, and the page has been retrieved.

When flattening tables, each combined table structure consists of $2^9 \times 2^9 = 262\,144$ entries fitting into a 2 MB page. In this case, each flattened table requires 18 bits for indexing. As shown in Figure 2 (bottom) this still requires the same total number of VA bits in order to traverse the page table structure and recover the physical address.

3.3 Page Walk Caches

Page walks are cached in three ways: translation caches (TLBs), page table entries in the regular data cache hierarchy, and through Page Walker Caches (PWCs), such as Intel's Paging Structure Cache [29]. The PWC allows walks to skip lookups for some levels of page table by matching the index bits of each level of the page table node with those cached by previous page walks. Intel's PWC is organized in three depths of translation caching: L4, L3 and L2. An L4 PWC holds previous walk paths that share the top 9 bit virtual address, allowing the walker to skip accessing the L4 page table entry, and go directly to the L3 page table entry. As each L4 entry covers 512 GB of virtual address space, this means that accesses that stay within a 512 GB virtual address range will hit in the PWC and be able to skip the L4 lookup. With an L2 PWC, a walk that matches all upper 27 bits of the virtual address will be able to skip the first three levels of the page table, and directly access the level 1 page table node. Such L2 PWC hits enable single-access translations (only a L1 entry access is required) for TLB misses within 2 MB regions of virtual address space.

Thus, the PWC enables page walks to skip one, two or three levels of the page walk depending on the locality of the virtual address. The impact of the PWC is shown in Figure 10, baseline. The PWC enables skipping an average of 2.2 to 2.9 of the 4 accesses

[2]We have also simulated this flattening on both our simulators but omit them from the paper due to space constraints. Representative results can be seen in the mobile case study in Figure 14.

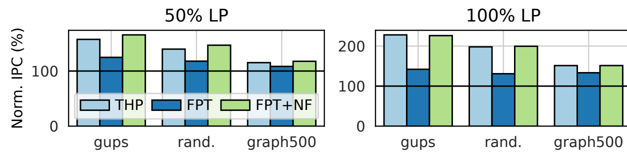

Figure 4: Impact of avoiding redundant page table entries for large pages (FPT+NF vs. FPT) for 50% and 100% large pages. Normalized to 0% large pages.

a naive page walker would need for all benchmarks except gups and random access. Those two benchmarks are an exception, as they exhibit a highly random access pattern across a large enough virtual address range that the PWC is only able to skip one level (one memory access) per page walk on average.

As a two-level flattened page table consists of a root level with up to 2^{18} page table entries, the L3 PWC can match the top 18 bits of the virtual address. A PWC hit will then skip the first level of the flattened page table (the L4+L3 entry) and directly access the flattened L2+L1 page table entry, resulting in a single memory access for each TLB miss. Our evaluations (Section 7) confirm that the flattened page table augmented with a PWC sees an average of one memory access per page walk (Figure 10). It is worth noting that merging the first two levels of the page table (L4+L3) may make the PWC more effective as each PWC hit translates twice as many index bits and fewer PWCs are required since there are fewer levels, enabling each one to cache more entries.

When running many applications with small memory footprints, the benefits from flattening may be smaller as the PWCs already skip many of the page table accesses. In such cases, flattening may not be as beneficial and may result in bloating (Section 3.2). However, cache prioritization (Section 5) will continue to be beneficial whenever TLB pressure is high, either due to many small applications co-running with frequent context switches or simultaneous multithreading, or fewer applications with very large page tables.

3.4 Supporting Large Data Pages

If the lower two levels (L2+L1) are flattened, then there is no way to use an L2 entry to directly provide a 2 MB translation. This requires 512 replicated translations in the L2+L1 node all pointing to the same 2 MB page. While this may be helpful for sparse accesses (since it enables only two accesses for the page walk) it puts pressure on the cache if many of those replicated entries are accessed.

Figure 4 shows the resulting loss in performance with flattening (FPT, middle dark blue) compared to a traditional page table (THP, left light blue) for two fragmentation scenarios: 50% large pages (realistic [42]) and 100% (performance limit, but unrealistic). To mitigate this, we take advantage of the flexibility to flatten different parts of the page table differently. Specifically, we allocate/promote 2 MB data pages in 1 GB virtual address regions and mark those regions to not have their L2 and L1 tables flattened (Figure 3 left, bottom mapping). This allows allocations in the 1 GB regions to behave as L4+L3 flattened tables (Figure 3 center) while other regions also have their L2+L1 levels flattened. Page walks to 4 KB mappings in this region will require up to 3 memory accesses, but 2 MB mappings will only require up to 2 accesses and no replicated entries. With this optimization (FPT+NF in Figure 4), flattening surpasses the baseline by providing the benefits of L4+L3 flattening with efficient large page access.

The OS can decide how to flatten the page table, for example, based on mapping statistics gathered for page promotion or hints from the application. In our L4+L3 and L2+L1 flattened simulations, we heuristically mark a 1 GB region to not have L2+L1 flattened if there are 32 or more 2 MB pages in it, but this threshold can be dynamically adjusted by the kernel. Applications with many 2 MB pages may benefit instead from flattening the L3+L2 levels (Figure 3, right), enabling the PWC to skip most L4 accesses, providing single-access page walks to 2 MB pages and two accesses for 4 KB pages.

3.5 Accessing Recursively Mapped Page Tables

To manipulate page tables Linux uses a mapping of all physical memory to a contiguous virtual address range in kernel space and a software page table walk. However, Windows uses recursive page tables where the page tables map themselves into their own address space. This is done by having a special recursion entry in the top-level (L4) node that points back to the node itself, and using recursions through this node to prevent the page walker from reaching the ordinary leaf node. As a result, the walk returns the address of a the page table node where it stops, and not the normal data translation/frame.

The degree of recursion can be controlled by the number of index fields in the VA that are filled with the recursion entry index. Recursive access to the page table for a 4-level table with the middle two levels flattened (L4, L3+L2, L1), is shown in Figure 5 with a normal translation of a data page to the left.

If the top 9 bits of the VA are the recursion index (Figure 5, middle), the 3-step page walk will recurse once and step through the top (L4) node twice: L4 → L4 → L3+L2. The final VA bits will then return the address of the L1 page table node indexed from the entry in the L3+L2 node. In a similar manner, if the top 18 bits of the VA have the recursion index concatenated twice, then the page walker will recurse twice (Figure 5, right), and step through the top (L4) node three times: L4 → L4 → L4, returning the address of the L3+L2 page table node indexed by the L4 node.

Since the address encoding is the same for data translations and pointers to page table nodes, the same page walk can return either data translations (with no recursions) or the addresses of the page table nodes themselves (by adding recursions, which causes the page walk to end earlier on a page table node). The more recursions, the higher the node level returned, and the specific node is selected by the remaining VA bits.

Traditional page walks terminate and return a large page translation when they encounter an entry marked as a large page (e.g., a 1 GB page is returned if marked at L3 or a 2 MB if at L2). To make recursive page table walks work with flattened page table nodes, the walker is modified to recognize pointers to flattened page tables as large page mappings (in addition to normal 2 MB mappings) while looking up L2 entries (Figure 5, right).

Another problem arises when the root page table node is a flattened node. The page walker cannot simply use 18 bits of the VA to index recursively into flattened nodes. For example, accessing

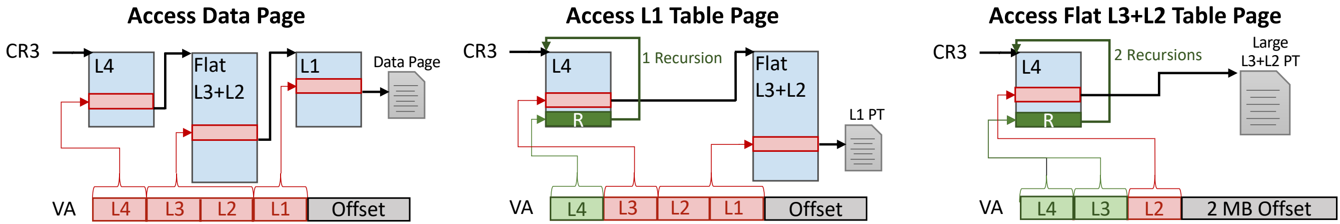

Figure 5: Recursive mappings used to access page tables in a L4, L3+L2, L1 page table organization.

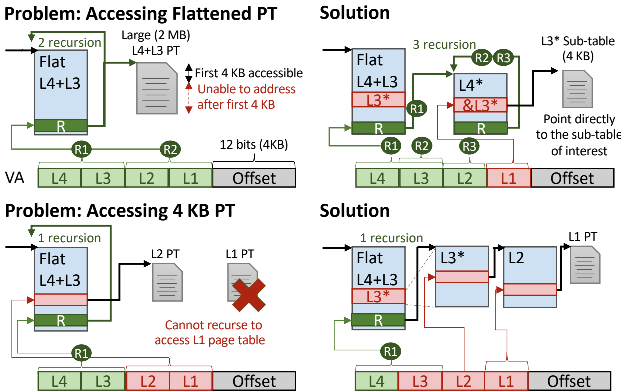

Figure 6: Recursion problems with flattened L4+L3 nodes and how a glue sub-table (L4*) acts as a traditional unflattened L4 to enable recursion.

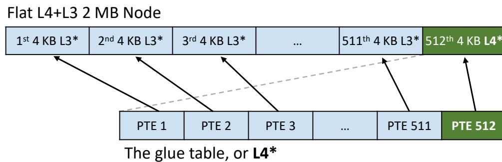

Figure 7: A 4 KB glue sub-table, L4*, points back to all the 4 KB, L3*, sub-tables of the flattened L4+L3 2 MB table. Note that the pointers treat the L3* sub-tables as traditional 4 KB tables.

the L4+L3 node in a flattened L4+L3, L2, L1 page table (Figure 6) requires two recursions of 18 bits each for the L4+L3 2 MB node (top left), leaving insufficient bits for the final indexing of the node. Similarly, it is not possible to reach the L1 node in this situation (bottom left), as even a single recursion will overshoot in address bits.

To address this, we propose using a 4 KB region within the flattened L4+L3 page table as a *glue sub-table*. This glue table serves two purpose: First, it makes it possible to use a page table with a flattened L4+L3 on systems or devices that do not natively support flattened page tables by accessing the page table through the glue table. Second, it enables recursion in systems with flattened L4+L3 tables without any corner cases in the architecture.

The insight behind this is that a flattened L4+L3 table can be thought of as a series of concatenated L3 sub-tables (Figure 7, L3* in Figure 6). To support recursion on flattened L4+L3 tables, we embed a traditional L4 table as one of the L3* sub-tables in the flattened L4+L3 table. We denote this specific L3* sub-table as L4*. This sub-table contains pointers to all of the L3* sub-tables (including itself) within the flattened L4+L3 table. This is illustrated in Figure 7 where the embedded sub-table is highlighted in green at the top, and the contents of the embedded sub-table is shown at the bottom. This table corresponds to the self-referencing entry in a traditional L4 table and occupies a single sub-page (the 512th sub-table in the figure, real systems usually select a random index at boot time) of the 2 MB L4+L3 table. As we are using a sub-table within the 2 MB L4+L3 table, we do not require an additional 4 KB allocation for the L4*. On recursive page table walks the upper 9 bits of the 18-bit index accesses the L4* sub-table of the flattened L4+L3. This triggers a recursion and the lower 9 bits select an entry in the L4*, which holds an address to a L3* sub-table of the flat L4+L3 table, providing recursion back to a sub-table of the flattened L4+L3 table. If multiple recursions are required, the lower 9 bits can point to the PTE entry that points to the L4* and the next recursive walks will access the L4* 4 KB table.

Figure 6 shows an example of recursion on a flattened L4+L3 table with a glue table. The first case (top right) illustrates how an arbitrary entry in the L4+L3 table can be accessed. To access an entry in the L4+L3 table, the generated virtual address needs to trigger three recursions. The first step of the walk performs two recursions (R1, R2) since it uses 18 bits of address to access an entry within the glue table. The top 9-bits of the index (the L4 bits) trigger the first recursion (R1) by selecting the glue table (or L4*). The lower bits (the L3 bits) select the recursion entry within the glue table which corresponds to the second recursion (R2). We conceptually describe R1 and R2 as two recursions, however, they are actually single memory accesses into a PTE in the L4+L3 node. The next recursion (R3) uses the L2 bits to index into the L4* table to accesses the recursion entry again. The final step of the walk uses the L1 bits to access an entry within the glue table which contains a pointer to an L3* sub-table (denoted as &L3*). The second example (bottom right) shows how an L1 table can be accessed using a single recursion. In this case, the L4 bits select the glue-table (R1) and the L3 bits select an entry within the glue table. The page table walker then uses the L2 bits to index into the L3* table, the L1 bits to index into the L2 table, and the offset bits to access the L1 table.

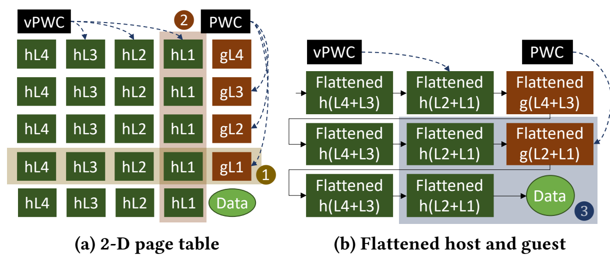

Figure 8: Flattening in virtualized 2-D page table walks. The PWC and vPWC skip stages of the virtualized walks.

3.6 Implications for Five-Level Page Tables

Unless programs spread their data across the full address space, 5-level page tables [31] will behave similarly to 4-level page tables with the PWC caching translations for essentially all 5th-level translations. However, their larger address space opens up more possibilities for flattening. In particular, merging the L5+L4 levels and the L3+L2 levels and directly translating 2 MB pages or using an L1 level when 4 KB pages are needed would be quite attractive. It might even be desirable to use a 1 GB page by merging three levels, if the kernel can reliably allocate such regions.

4 FLATTENING AND VIRTUALIZATION

With virtualization, guests use a set of page tables to map from the guest virtual address (gVA) to the guest physical address (gPA), and the hypervisor uses a second set of page tables to map from the guest physical address to the host physical address (hPA). A guest translation therefore needs a two-dimensional page walk (Figure 8a) in which each of the four guest page table level access first requires its own four-access host page table walk, and a final four-access host page table walk is required to translate the final guest physical address, incurring a total of 24 memory accesses. As there are two page tables involved, there are three possible combinations of page table flattening. Either the host or guest page tables can be flattened (14 accesses), or both (8 accesses), can be flattened. (See Figure 8b.)

4.1 Effective Memory Accesses per Page Walk

Although virtualization naively incurs 24 memory accesses per walk, in practice many of these accesses are eliminated by a combination of three techniques (Figure 8a): PWCs, large pages, and host translations in the TLB. Figure 8a shows how the PWC skips levels of the guest (5th column, skip downwards) walk and a vPWC skips them for the host walk (applicable on all five rows). Our evaluations show that this reduces the number of accesses to as low as 3 to 4.8, with an average of 4.4. Even the two most random access applications, GUPS and random access, require only 9.6 and 9.4 memory accesses.

Hypervisors also try to map the guest physical pages to host physical pages in large pages to make host translations more efficient [1, 10]. Using large pages comes with the benefit of removing the last level of the conventional four-level page table. In a 2-D page table, a 2 MB guest mapping can remove a row of memory accesses (❶ in Figure 8a). Large page mappings for the gPA to hPA mapping remove columns (❷).

With host and guest page tables flattened, the number of accesses per page walk is reduced to 8, and, with the help of the PWC, this comes down to 2.8 in practice, as shown by the box (❸ in Figure 8b). The guest PWC allows the page walk to skip the flattened g(L4+L3), and directly access the flattened g(L2+L1). To access the g(L2+L1), the host page table is traversed, and here the vPWC skips the flattened h(L4+L3), and directly accesses the flattened h(L2+L1). After the gPA of the data is resolved, the host page table is traversed once more for the actual data gPA to hPA translation, resulting in another walk, where the vPWC skips the h(L4+L3), enabling a single access final translation. Finally, after 2.8 accesses, the actual data can be accessed. Thus, flattening both guest and host page tables allows reducing the number of memory accesses from on average 4.4 down to 2.8.

5 CACHE PRIORITIZATION

The other half of our approach is to reduce the latency (and energy) of each page table access by increasing the chances that it will be a cache hit. We do so by biasing the cache replacement policy to keep page table entries when an application is experiencing high TLB miss rates. Unlike CSALT [35], which biases the cache to store TLB entries to support their DRAM-TLB-cache design, we bias the cache to store page table entries for the existing page table walker.

Biasing the replacement policy to favor page table entries means evicting more data, but we find that applications with high TLB miss rates also exhibit high data miss rates (L2 and L3 data miss ratios of 95% and 80%). This, combined with the page table access being on the critical path to the data access, suggests that allocating more cache space to the (much smaller) page table over the data itself is likely to be more beneficial than caching the data[3].

As data sets grow in the future, the likelihood of hitting in the cache will always remain higher for the page table than for the data for applications with limited locality simply due to the disparity in size between the two. Concretely, an application with 8 GB of densely allocated memory requires 2^{21} 4 KB pages represented by 8 B each, for a total of 16 MB of space. In the conventional four-level page table, each intermediate level is 1/512 the size of the previous one, resulting in only 32 KB for the L2 level, which is likely to be skipped by the PWC. From this analysis we see that an 8 GB memory workload could fit all leaf page table entries into a 16 MB cache, and is thus practical to cache in today's systems. Even with a random access pattern, caching even a portion of the page table entries would deliver a significant benefit. Future, larger workloads might even benefit from prioritizing among different levels of the page table itself, although we have not explored this.

In addition to our simulations, we explored the potential of preferentially caching page tables on current system by creating a thread that runs on another core and periodically touches the page table of a target application to keep it in the shared LLC. We ran this thread together with graph500 (scale 24) on an Intel i7-9700 with a shared 12 MB LLC, and found a 5% performance increase from keeping the entire page table in the LLC (miss ratio of 0% for

[3] MASK [13] identified that the *opposite* approach of prioritizing data over page table entries is better for latency-insensitive GPUs.

the page table thread). While this experiment both uses extra LLC bandwidth and does not bring the page table into the target's private L2, unlike our proposed and simulated design, it does demonstrate that preferentially caching page tables is a promising approach. This leads to our conclusion that if the program accesses the memory in a way that does not make good use of the TLB and the caches for data, then we would be better off prioritizing page table entries in the caches.

6 IMPLEMENTATION

6.1 Hardware Changes

Flattened page tables require augmenting the hardware page table walkers to be aware of the size of the page used at each level of the page table. This requires two additional bits (for 4 KB, 2 MB, and 1 GB pages, or one for just 4 KB and 2 MB) in the CR3/TTBR register (for the root node) and at each entry in the page table, possibly in the currently unused bits. These bits indicate the size of the page at the next level of the page table, and are needed to determine how many bits of the virtual address to use as index bits in the walk. As page table nodes are aligned, using flattened page tables frees up 9 or 18 bits (2 MB or 1 GB nodes) in the page table entries that point to flattened page table nodes, leading to more available bits. Similar small changes are required to the PWC, but there is the potential to use storage more effectively if there are fewer levels in the page table. Overall these changes are minor and incur essentially no hardware overhead.

Cache prioritization requires detecting phases of high cache and TLB miss and enforcing the prioritization. Detection is easily accomplished using existing hardware counters. Prioritization can be accomplished with a range of techniques. For example, existing cache partitioning technique (such as way-partitioning) can allow the OS to pre-load page table entries into part of the cache that will not contend with the application's own data. Alternatively, a per-cacheline tag bit can indicate if the entry is a page table and then bias the replacement policy away from such lines. The cost of such hardware is less than 0.2% of the cache size, and is already present in server-class processors that support per-context cache partitioning [11, 32]. Indeed, Arm's MPAM already stores a partition ID that can be used to differentiate processes or even I/D cache lines [11]. We use this approach for prioritization in the L2 and LLC during phases of high TLB miss rates: when choosing a victim for replacement, 99% of the time we choose to evict data over page table entries. If there are no data entries in the set, or in the other 1% of the evictions, we evict the LRU entry. We empirically found that this ratio works well.

To limit impact on co-runners in shared caches, prioritization can occur within a context's (core/process) allocation by using identifiers in the tags [11, 32]. For our multicore simulations we used this approach to prevent one process' data from evicting another's page table.

As flattened page tables require only trivial hardware changes, the energy benefits will be proportional to the reduction in memory system accesses and execution times.

6.2 Software Changes

We have a working operating system prototype based on Linux 5.8.13 running on an industrial Armv8 functional simulator and on existing HW by adding an additional shim level before the large page tables. The change to the kernel is small: +614/-109 lines. Page tables are automatically flattened if a sufficiently large allocation can be provided by the kernel. Our implementation flattens L3+L2 (Figure 3, right). L5+L4 flattening could be added with only minimal changes. We have not implemented dynamical flattening of page table levels after allocation. However, this would be straight-forward to implement by allocating a large page and copying the page table entries of the lower nodes (the L2 child nodes of the L3 node that is being flattened into an L3+L2) into the new flattened node. The upper node (L4) entry can then be updated to point to the flattened node.

Most Linux's page table management is shared between architectures. The kernel internally assumes that all architectures implement a 5-level radix tree where the highest level, L5 (PGD using Linux's terminology), is always implemented. This avoids special cases since unimplemented levels can be *folded* into the parent level using a virtual entry that points back to an entry in the parent table. For example, in a system with a three level page table, the kernel would implement L5 (PGD), L2 (PMD), and L1 (PTE), with L3 (PUD) and L4 (P4D) folded into the L5 table. Internally, the kernel treats L3 and L4 as having a single virtual entry (effectively the corresponding L5 entry) and no storage.

Using this existing support, we can readily implement support for flattened L2+L3 tables. We simply fold the L3 table, request 2 MB instead of 4 KB when allocating an L2 table, and change the macros that define the bit ranges used to index into the tables. However, this approach does not work in practice since it makes 2 MB pages a hard requirement, which could lead to a situation where fragmented systems fail to allocate page table node even when there is free memory.

To support a graceful fallback to 4 KB table nodes, we needed to be able to selectively fold the L2 and L3 tables per sub-tree of the page table. The kernel currently only supports folding an entire level across a process' entire address space. For our our prototype, we changed the signature of a handful of functions to add a mechanism to determine if a level needs to be folded based on the state of the parent entry. Overall, this change is small (roughly 100 lines of shared code a few tens of lines per architecture) and mostly mechanical.

With selective sub-tree folding in place, we can allocate large L3 tables. If we succeed in allocating a 2 MB page for a flattened L3+L2 table, we treat the table as an L2 table and fold the L3 table. The L4 entry is configured to point to the new L2 node and a bit in the entry is set to indicate that the next node has been flattened. If the flattened L3+L2 table allocation fails, we allocate normal 4 KB L3 and L2 tables.

We stress-tested our prototype kernel on a server system with 128 hardware threads by building a Linux kernel with 100 concurrent processes. Since the hardware does not support flattened tables, we allocate and manage the flattened table in the kernel, but inject a shim table between the L4 table and the flattened L3+L2 table to be compatible with the existing architecture. We found that 0.5% (20

Table 1: Server simulation configurations

Processor	2 GHz Out-of-order x86 Processor
L1 I/D Cache	4-cycle, 32 KB, 8-way, 64 B block
L2 Cache	12-cycle, 256 KB, 8-way, 64 B block
L3 Cache	42-cycle, 16 MB, 8-way, 64 B block
Memory	DDR4-2400, 4 channels
L1 TLB (Parallel lookup)	4 KB: 1-cycle, 64-entry , 4-way 2 MB: 1-cycle, 32-entry, 4-way
L2 TLB	4 KB/2 MB: 9-cycle, 1 536-entry, 12-way
PWC (Parallel lookup)	L4: 1-cycle, 4-entry FA L3: 1-cycle, 4-entry FA L2: 1-cycle, 24-entry FA
Nested TLB [17]	1-cycle 16-entry FA

out of 3464 compiler invocations) failed at least one of the two 2 MB allocations needed for the flattened page table on a system with 6% memory oversubscription (500 MB swap with 8 GB RAM). This increased to 12% failures on a system with 50% oversubscription. When a 2 MB allocation failed, the prototype used our fall-back path with traditional 4 KB tables.

7 EVALUATION

We implemented our proposal in the gem5 [20] simulator (Table 1). We model the TLB [30] and the Paging Structure Caches (our PWC) [29, 51] of the Intel Skylake microarchitecture. Unfortunately, the page walker cache provided in gem5 is modeled as a cache with 64 B-lines, which does not accurately represent either page table caches nor translation caches [14]. We implemented our own PWC, which sends misses to the L1 data cache [32], as in Intel processors. Our flattened page tables use the L2 PSC and we model a vPSC for the host page table under virtualization.

This work requires comparing two different page table organizations. To keep all other aspects of the simulation state the same, such as data layout in memory and kernel interrupts, we use system-call emulation (SE) mode to vary the page table organization, while keeping all other states identical. SE is sufficient as we do not study the effect of changes in memory mapping or the page table during the runtime of the program, and focus our evaluation on the program execution itself. The gem5 SE mode normally does not model a page table walker nor a page table in the form of a radix tree. We included page table walks by constructing the page table based on the simulator VA to PA mappings, and having the page table walker appropriately access each level through the caches.

We evaluate three large page fragmentation scenarios [34, 41]: 0% (all 4 KB pages; worst case for page walks), 50% (typical real-world [42, 54]), and 100% (all large pages; best performance but unrealistic). For the 50% scenario, we allocate large pages for the lower-half of the address space to simulate an OS that runs out of free large pages. Mosalloc [4] proposed another layout that uses random windows of contiguous memory. The authors found that this can sometimes result in behaviors similar to either 4 KB allocations (our 0% scenario) or 2 MB allocations (our 100% scenario). Thus, we used the lower-half of the address space in large pages scheme.

We evaluate our proposals on benchmarks that stress the TLB: GraphBIG [40] (LDBC-1000k dataset, 6.6 GB of memory usage) and graph500 (scale 24, 5.4 GB); benchmarks with significant TLB misses from biobench [7] and SPECCPU 2006 [28]; GUPS (N = 30, 8 GB); large linear classification (liblinear) [24] (inputs: url_combined and HIGGS); a hashjoin microbenchmark [3], and the XSBench [50]. As large in-memory databases, such as memcached, and very large graphs exhibit random access patterns, we also include a microbenchmark that represents such random access behavior.

7.1 Non-virtualized Execution

Figure 9 shows the performance of our approach and those of ASAP [36], Elastic Cuckoo Hashing [49], and CSALT [35]. The performance numbers are presented relative to a baseline system with a 4-level page table and Intel-style PWCs. We plot results for three fragmentation scenarios stacked to show the change as the percent of large pages increases: bottom (0% large pages), middle (50% large pages, realistic [42]), and top (100% large pages, best performance, but unrealistic). Performance is normalized to the baseline configuration with 0% large pages (horizontal black line/dark gray bar), and the effect of 50% and 100% large pages can be seen on the baseline system in the gray bar in the middle.

We see two trends in the results: First, for the 0% (bottom) and 50% (middle) large page cases, flattening the page table (FPT), prioritizing page table entries in the cache (PTP), and the combination (FPT+PTP) contribute to increasing performance improvements beyond the state-of-the-art (blue bars increasing to the right). Second, as the fragmentation decreases (stacked from bottom to top), the impact of flattening and prioritization also decreases. This is because the TLB misses are much less frequent (less opportunity to reduce latency) and the page table size itself is drastically smaller (less need to preferentially cache it). Interestingly, the combination (FTP+PTP) is almost as effective as moving from 0% to 50% large pages (9.2% vs. 11.0%). For the detailed analysis below we look at the 0% large page scenario as flattening and prioritization have the largest potential[4]. Figure 10 shows that the flattened page table together with the PWC results in single-access page walks for all workloads. Overall, flattening the page table for 4 KB pages shows a geometric mean performance improvement of 2.3% (solid light blue bar) vs. 1.7% for ASAP (solid dark green bar) and a net performance loss for ECH (solid medium green bar).

Prioritizing page table entries in the L2 and L3 caches on the baseline improves performance by 6.8%. This does increase data misses slightly (L2: +4.7 percentage points) in exchange for far fewer page walk misses (L2: -36.2 percentage points). Figure 10 (bottom) shows that this prioritization significantly reduces page walk latency from an average 50.9 cycles per walk (baseline) down to 33.0 (prioritizing). Flattening and prioritizing together results in 29.1 cycles per walk.

ECH shows lower performance than the baseline as it requires three (4 KB pages) or four (mixed 4 KB/2 MB pages) concurrent memory accesses vs. a single memory access with flattening. CSALT provides little benefit for the 0% large page scenario. We believe this is due to CSALT having been designed and evaluated only with large pages, which makes it poorly optimized for the much larger page tables from our 0% and 50% scenarios, and their assumption of

[4]We expect similar effects for 2 MB pages in the future if TLB reach does not increase as fast as data sizes.

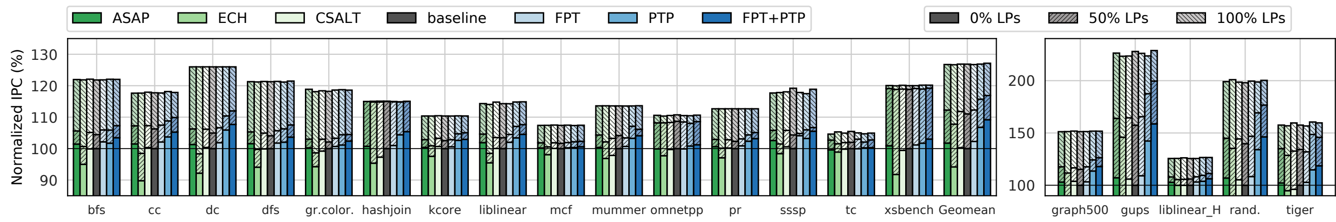

Figure 9: Performance of flattening (FPT), cache prioritization (PTP), both combined (FTP+PTP), and the related work (ASAP, ECH, CSALT). Three fragmentation scenarios are shown stacked: 0% large pages (bottom, all 4 KB pages), 50% large pages (middle, realistic), and 100% large pages (top, unrealistic for actual systems). Normalized to the PWC-equipped baseline (solid line and dark middle bar) with 0% large pages.

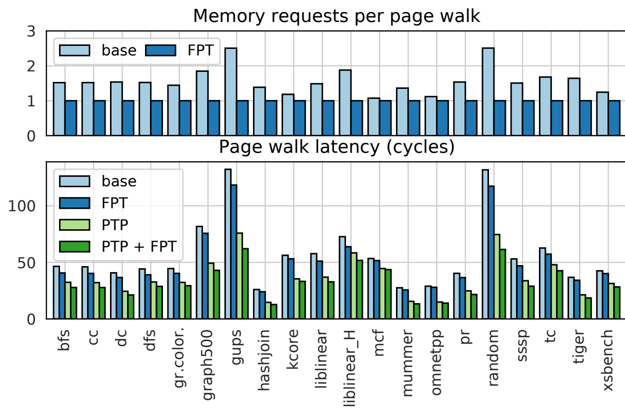

Figure 10: Memory accesses and latency per page walk across a traditional 4-level page table (baseline), flattened (FPT), and cache prioritization (PTP), all with PWCs.

very frequent (every 10 ms) context switches, which would make a PWC less effective.

Changing PWC size resulted in a performance impact of -1.5% to +2.4%, when sweeping the L3 PWC entries from 1 to 16 (baseline 4) for the most sensitive benchmark, GUPS. In comparison, flattening gave a benefit of 8.9% as it benefits from a single memory access, instead of the 2 memory access from a L3 PWC hit. Achieving a similar benefit to flattening (single memory access) would require increasing the L2 PWC size to approximately 4096 entries. The larger 16-entry L3 PWC provided +2.9% on top of our cache prioritization.

With larger data sets the ratio of page table size to LLC size increases, potentially making preferential caching less effective. To evaluate this, we increased the page table to LLC ratio by 2x, 4x, 8x, and 16x over the previously presented results. We shrunk the LLC size proportionally to evaluate higher page table to LLC ratios. This experiment does not take into account the increased capacity pressure of the larger workload on the L1 and L2 caches, however, as the workloads are already far surpassing the L1 and L2 cache sizes, we believe the effects would be similar. Even with these increases, we saw similar performance benefits of preferential caching: geometric mean improvements of 6.8% (baseline), 5.9% (2x), 5.6% (4x), 6.5% (8x), and 7.0% (16x). In the 16x page table to LLC ratio, we are caching 6.3% of the page table while still maintaining the benefit of cache prioritization. This scenario represents a 128 GB workload running on a 16 MB LLC. Based on these results we expect cache prioritization to provide benefit up to and possibly above the 128 GB point as we did not see any trend of the performance dropping off.

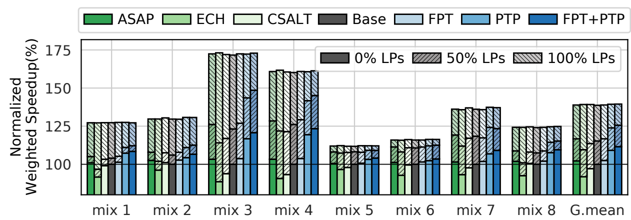

Figure 11: Multicore performance. Mean for all 20 mixes.

However, for systems with vastly more memory (e.g., 4 TB or 0.2% of the page table cached in a 16 MB LLC), page table entries were found to suffer capacity misses in the LLC [19]. We cannot predict the benefit of cache prioritization for such systems and it is possible that cache prioritization may not work in such capacity constrained scenarios. If large pages are widely available in these systems, mapping the 4 TB region with 2 MB large pages will result in a 16 MB page table, allowing us to efficiently keep these page table entries in the cache through prioritization.

To summarize, for 0% large pages, flattening the page table improved performance by 2.3% over the baseline. Prioritizing caching of the page table resulted in a 6.8% improvement, while the combination delivered 9.2%, which is significantly greater than the state of the art (bottom bars in Figure 9: ASAP 1.7%, ECH -5.9%, CSALT 0.3%). However, as the proportion of large pages increases, the relative improvement decreases. For the 50% large page scenario, the combination of flattening and prioritization delivered a 5.8 percentage point improvement vs. ASAP 1.2, ECH -3.2, CSALT 0.7 (middle bars in Figure 9).

Multicore. To evaluate the effect on shared caches and the memory hierarchy, we evaluated flattening and prioritizing with multiprogrammed workloads (32 MB shared L3, 4 cores with private L1s and L2s). We evaluated a total of 20 workloads: 11 homogeneous and 9 heterogeneous. Figure 11 shows the normalized weighted

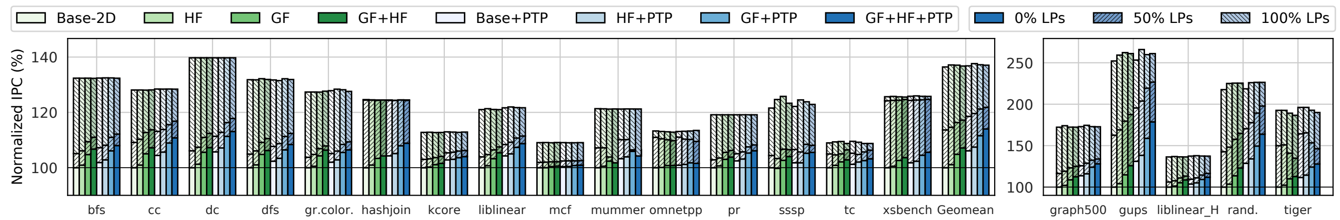

Figure 12: IPC comparison in virtualized environments of various combinations of flattening the page table for the host (HF) and guest (GF) and both (HF+GF), with (blue) and without (green) cache prioritization (PTP).

Table 2: Benchmark mixes for the multicore evaluation.

Mix	Benchmarks	Mix	Benchmarks
1	dc×4	2	liblinear_H×4
3	rand×2, dc×2	4	rand×2, hashjoin×2
5	hashjoin×2, mummer×2	6	liblinear×2, xsbench×2
7	tiger×2, dfs, bfs	8	rand, liblinear, dc, cc

speedup for 8 mixes (Table 2) and the geometric mean of all 20. The first two entries show that homogeneous mixes behave similarly to the individual benchmarks in Figure 9. This was consistent across all 11 homogeneous mixes. The heterogeneous workloads show a similar performance improvement trend to the individual benchmarks: Flattening and prioritizing each introduce performance improvements, and work together resulting in an average of 2.2%, 9.2% and 11.5% improvement, respectively, for the 0% LP scenario. The improvements are 1.4, 8.6 and 10.3 percentage points, respectively, for the 50%, and 0.2, 0.7 and 0.8 percentage points, respectively for the 100% scenario.

7.2 Virtualized Executions

The performance benefits for virtualized systems are shown in Figure 12. The green bars show the effects of flattening the host page tables (HF), the guest page tables (GF), or both (GF+HF). The blue bars include cache prioritization. The baseline (Base-2D, leftmost) is a virtualized system with the 2D 4-level page table, which naively incurs up to 24 accesses per page walk, but, because we include two sets of PWCs for the guest and the host and a nested TLB [17] to hold host translations, the average number of accesses is only 4.4.

Flattening in virtualized environments delivered a larger performance improvement than native environments. Flattening the host page table alone resulted in a 1.1% performance improvement, while flattening the guest page table alone delivered 4.9% improvement.

To understand the difference, consider Graph500, which which requires 11 MB of guest and 22 KB of corresponding host page tables for its 5.4 GB gVA. The Nested TLB [17] and the vPWC (host PWC) work together to efficiently cache the small host translations (22 KB) for the guest page tables. As a result, the host translations for the guest page table accesses are effectively single access, even for the baseline 2D walks. This means there is less benefit for flattening the host page table for the guest page table accesses. Indeed, most of the benefit for host flattening comes from the final host data address translation. Guest flattening, however, allows flattening

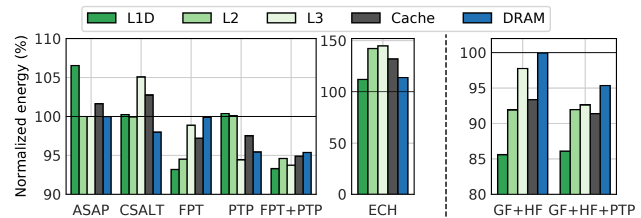

Figure 13: Dynamic energy consumption of the cache hierarchy and DRAM for data and page walks for native (left, center) and virtualized (right) executions. Normalized to respective baselines.

the 11 MB page table, resulting in fewer guest accesses (which also leads to fewer host table walks).

Finally, flattening both page tables is the most effective. Flattening both host and guest page tables delivered the largest performance improvement of 7.1%. Adding page table prioritization in the cache, increased this significantly to 7.5%, 11.6%, and 14.0% performance improvements for flattening host, guest, and both, respectively. We found that page table prioritization in the cache consistently provided a 6.1 to 6.9 percentage point improvement, which is similar to the native results. The effects of large pages are similar to the non-virtualized executions.

7.3 Dynamic Energy

We present dynamic energy normalized to the baseline in Figure 13 for the 0% LP scenario. The cache energy is comprised of the L1D, L2 and L3 modeled with the CACTI [39] at 22nm. We include both data and page table walks. For DRAM, we report relative off-chip accesses.

ASAP which issues prefetches into the cache hierarchy for the lower two levels needs to re-access the lower two levels resulting in higher L1D accesses. CSALT does reduce off-chip DRAM accesses by 2.0% but increases the L3 access 5% resulting in 2.7% higher cache energy consumption. ECH issues three accesses per walk for 4 KB pages resulting in higher cache (32%) and memory (14%) energy consumption. This is a different behavior from the baseline, ASAP, CSALT and our work, all of which benefit from fewer page walk accesses due to the PWC.

Flattening reduces the number of memory accesses to the cache hierarchy (-2.8%). Cache prioritization increases L2 hits and reduces accesses to the L3 and the DRAM, reducing cache hierarchy energy

Table 3: Mobile-core simulation configuration.

Processor	3 GHz Out-of-order Armv8 Processor
Caches	L1 I/D: 32 KB 4w, L2: 512 KB 8w, L3: 2 MB 16w
Memory	90 ns, 48 GB/s
L1 TLB	I: 32-entry FA, D: 48-entry FA
L2 TLB/PWC	4 KB, full translations: 1 536-entry 6w 1 GB, 2 MB, partial & full translations: 256-entry 4w

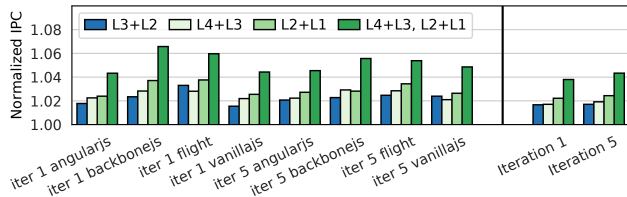

Figure 14: Performance of Speedometer 2.0 in a virtualized system normalized to baseline 2-D page table.

(-2.5%) and DRAM accesses (-4.6%). Finally the combination of both results in a dynamic energy reduction of 5.1% and 4.7%, for cache and DRAM, respectively. We see a similar trend in virtualization with flattening both guest and host table reducing cache energy by 6.7% and adding prioritization resulting in 8.7% cache and 4.7% DRAM energy saving.

7.4 Case Study: Flattening for Mobile Systems

We evaluated flattening alone on a production industrial simulator used for next-generation mobile core exploration, configured as a high-end mobile device (Table 3). We used the Speedometer 2.0 [2] benchmark, which tests common browser operations such as DOM APIs, JavaScript, CSS resolution, and layout. It is a good representation of real-world mobile system performance, including JITing across iterations (e.g., iteration 1 executes 9.5% more instructions than iteration 5). The system is based on a standard AOSP 10.0 distribution which does not use transparent huge pages. We use virtualization, as future mobile systems are expected to use pKVM for increased security [22].

Figure 14 shows the performance gains for a range of flattening options. The improvement is largest for flattening both L4+L3 and L2+L1 (dark blue bars, 3.8% and 4.3% for iterations 1 and 5 respectively), which is consistent with our earlier server results. Overall, flattening closer to the leaf nodes delivers the largest benefit, as they make up the majority of the nodes and are least likely to be cached, particularly under virtualization.

7.5 Flattening Other Levels

We have also simulated flattening the L3+L2 layer, which is by design beneficial for 2 MB data mappings as discussed in Section 3.4. For L3+L2 flattening, our results show a 0.2, 0.3, 0.1 percentage point benefit over the baseline for 0%, 50% and 100% large page scenarios, respectively. The improvements for virtualization (flattening both host and guest) is 0.7, 1.0, and 1.2 percentage points for the 0%, 50% and 100% large page scenarios. Finally, for the 100% large page scenario, we found that L2+L3 flattening outperformed L4+L3 and L2+L1 flattening by 0.3 and 0.8 percentage points, for native and virtualized executions. These results are consistent with what we saw for the mobile system (Figure 14).

8 CONCLUSION

In this work we explored two complementary techniques for reducing the impact of page walks: reducing the *number of accesses* by flattening the page table and *reducing the latency* of the accesses by preferentially caching page table entries. We evaluated server and mobile systems across a wide range of benchmarks with both academic and industrial simulators.

We show that modern PWCs result in little impact from flattening for non-fragmented and non-virtualized systems, but that with the increased page table sizes of realistically fragmented systems and complexity of virtualized page walks, flattening provides significant benefit. Further, we see that preferentially caching page table entries during periods of high TLB miss rates provides significant benefit in all scenarios, as high TLB miss rates are strongly correlated with high data cache miss rates, and the page table is sufficiently smaller than the data that it is far more likely to see reuse through the cache hierarchy. We further identify the challenges of self-referencing page tables and provide a practical solution.

Combined, flattening and prioritization allow us to serve the vast majority of page walks with a single cache hit, delivering significant performance (+14.0%, +7.2% with realistic large page fragmentation) and dynamic energy (-8.7% cache and -4.7% DRAM) benefits beating the state-of-the-art. Implementation requires only very small changes to the operating system (as we leverage existing large page support and provide a graceful fallback path) and hardware (as we use existing performance counters and cache partitioning techniques, or need one bit per tag). If main memory growth continues to outpace TLB growth, we expect that these techniques will become increasingly important.

ACKNOWLEDGMENTS

The authors would like to thank the shepherd, Jayneel Gandhi, and the anonymous reviewers of this paper. We would also like to thank Probir Sarkar, and Abhilash V. Varier, and Richard Grisenthwaite from Arm for valuable insights and feedback.

This work was supported by the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows Program (grant No 2015.0153), the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant No 715283), and the NRF of Korea through the Postdoctoral Fellowship Program (NRF-2020R1A6A3A03037317). The computations and data handling were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at NSC (2021/22-435) and UPPMAX (2021/23-626) partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

REFERENCES

[1] 2015. Huge Page Support - Xen. https://wiki.xenproject.org/wiki/Huge_Page_Support.
[2] 2018. https://browserbench.org/Speedometer2.0/.
[3] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proc. Internationla Conference on Architectural*

Support for Programming Languages and Operating Systems (ASPLOS). ACM, 283–300. https://doi.org/10.1145/3373376.3378468

[4] Mohammad Agbarya, Idan Yaniv, Jayneel Gandhi, and Dan Tsafrir. 2020. Predicting Execution Times With Partial Simulations in Virtual Memory Research: Why and How. In *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Press, 456–470. https://doi.org/10.1109/MICRO50266.2020.00046

[5] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting Hardware-assisted Page Walks for Virtualized Systems. In *Proc. International Symposium on Computer Architecture (ISCA)*. 476–487. https://doi.org/10.1109/ISCA.2012.6237041

[6] Sam Ainsworth and Timothy M. Jones. 2021. Compendia: Reducing Virtual-Memory Costs via Selective Densification. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM 2021)*. Association for Computing Machinery, New York, NY, USA, 52–65. https://doi.org/10.1145/3459898.3463902

[7] 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 *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2–9. https://doi.org/10.1109/ISPASS.2005.1430554

[8] 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 *Proc. International Symposium on Computer Architecture (ISCA)*. IEEE. https://doi.org/10.1109/ISCA45697.2020.00050

[9] AMD 2020. *Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors.* AMD.

[10] Andrea Arcangeli. 2010. Transparent Hugepage Support. In *KVM Forum 2010*. https://www.linux-kvm.org/images/9/9e/2010-forum-thp.pdf

[11] Arm 2020. *Arm Architecture Reference Manual Supplement: Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A.* Arm.

[12] Arm 2020. *Arm Cortex-A76 Core Technical Reference Manual.* Arm.

[13] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 503–518. https://doi.org/10.1145/3173162.3173169

[14] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In *Proc. International Symposium on Computer Architecture (ISCA)*. 48–59. https://doi.org/10.1145/1815961.1815970

[15] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proc. International Symposium on Computer Architecture (ISCA)*. ACM, 307–318. https://doi.org/10.1145/2000064.2000101

[16] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proc. International Symposium on Computer Architecture (ISCA)*. 237–248. https://doi.org/10.1145/2485922.2485943

[17] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 26–35. https://doi.org/10.1145/1346281.1346286

[18] Abhishek Bhattacharjee. 2013. Large-Reach Memory Management Unit Caches. In *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 383–394. https://doi.org/10.1145/2540708.2540741

[19] Abhishek Bhattacharjee. 2017. Translation-Triggered Prefetching. In *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 63–76. https://doi.org/10.1145/3037697.3037705

[20] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[21] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 435–448. https://doi.org/10.1145/3037697.3037704

[22] Will Deacon. 2020. Virtualising for the Masses: Exposing KVM on Android. https://mirrors.edge.kernel.org/pub/linux/kernel/people/will/slides/kvmforum-2020-edited.pdf. In *KVM Forum*.

[23] Yu Du, Miao Zhou, Bruce R. Childers, Daniel Mossé, and Rami Melhem. 2015. Supporting Superpages in Non-Contiguous Physical Memory. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 223–234. https://doi.org/10.1109/HPCA.2015.7056035

[24] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A Library for Large Linear Classification. *J. Mach. Learn. Res.* 9 (June 2008), 1871–1874. http://jmlr.org/papers/v9/fan08a.html

[25] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Press, 178–189. https://doi.org/10.1109/MICRO.2014.37

[26] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proc. International Symposium on Computer Architecture (ISCA)*. IEEE Press, 707–718. https://doi.org/10.1109/ISCA.2016.67

[27] Faruk Guvenilir and Yale N. Patt. 2020. Tailored Page Sizes. In *Proc. International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA45697.2020.00078

[28] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17. https://doi.org/10.1145/1186736.1186737

[29] Intel 2008. *TLBs, Paging-Structure Caches, and Their Invalidation.* Intel.

[30] Intel 2016. *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Intel.

[31] Intel 2017. *5-Level Paging and 5-Level EPT.* Intel.

[32] Intel 2019. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3.* Intel.

[33] 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 *Proc. International Symposium on Computer Architecture (ISCA)*. 66–78. https://doi.org/10.1145/2749469.2749471

[34] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 705–721. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon

[35] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. 2017. CSALT: Context Switch Aware Large TLB. In *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 449–462. https://doi.org/10.1145/3123939.3124549

[36] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 1023–1036. https://doi.org/10.1145/3352460.3358294

[37] Chandrashis Mazumdar, Prachatos Mitra, and Arkaprava Basu. 2021. Dead Page and Dead Block Predictors: Cleaning TLBs and Caches Together. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Press, 507–519. https://doi.org/10.1109/HPCA51647.2021.00050

[38] Sparsh Mittal. 2017. A Survey of Techniques for Architecting TLBs. *Concurrency and Computation: Practice and Experience* 29, 10 (2017), e4061. https://doi.org/10.1002/cpe.4061

[39] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Press, 3–14. https://doi.org/10.1109/MICRO.2007.33

[40] 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 *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/2807591.2807626

[41] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 679–692. https://doi.org/10.1145/3173162.3173203

[42] 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 *Proc. International Symposium on Computer Architecture (ISCA)*. IEEE Press. https://doi.org/10.1109/ISCA45697.2020.00079

[43] Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. 2016. Efficient Synonym Filtering and Scalable Delayed Translation for Hybrid Virtual Caching. In *Proc. International Symposium on Computer Architecture (ISCA)*. 90–102. https://doi.org/10.1109/ISCA.2016.18

[44] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. In *Proc. International Symposium on Computer Architecture (ISCA)*. 444–456. https://doi.org/10.1145/3079856.3080217

[45] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[46] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 258–269. https://doi.org/10.1109/MICRO.2012.32

[47] Bin 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 *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12. https://doi.org/10.1145/2830772.2830773

[48] 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 *Proc. International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, USA. https://doi.org/10.1145/3079856.3080210

[49] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3373376.3378493

[50] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XS-Bench - 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*. Kyoto. https://www.mcs.anl.gov/papers/P5064-0114.pdf

[51] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. 2017. *Reverse Engineering Hardware Page Table Caches Using Side-Channel Attacks on the MMU*. Technical Report. Vrije Universiteit Amsterdam.

[52] Idan Yaniv and Dan Tsafrir. 2016. Hash, Don't Cache (the Page Table). In *Proc. ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS)*. https://doi.org/10.1145/2964791.2901456

[53] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. 2010. Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation. In *Proc. International Conference on Supercomputing (ICS)*. ACM, 159–168. https://doi.org/10.1145/1810085.1810109

[54] Weixi Zhu, Alan L. Cox, and Scott Rixner. 2020. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *Proc. USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 829–842. https://www.usenix.org/conference/atc20/presentation/zhu-weixi