Compendia: Reducing Virtual-Memory Costs via Selective Densification

Compendia: Reducing Virtual-Memory Costs via Selective Densification

Sam Ainsworth
University of Edinburgh
Edinburgh, UK
sam.ainsworth@ed.ac.uk

Timothy M. Jones
University of Cambridge
Cambridge, UK
timothy.jones@cl.cam.ac.uk

Abstract

Virtual-to-physical memory translation is becoming an increasingly dominant cost in workload execution; as data sizes scale, up to four memory accesses are required per translation, and 24 in virtualised systems. However, the radix trees in use today to hold these translations have many favourable properties, including cacheability, ability to fit in conventional 4 KiB page frames, and a sparse representation. They are therefore unlikely to be replaced in the near future.

In this paper we argue that these structures are actually *too* sparse for modern workloads, so many of the overheads are unnecessary. Instead, where appropriate, we expand groups of 4 KiB layers, each able to translate 9 bits of address space, into a single 2 MiB layer, able to translate 18 bits in a single memory access. These fit in the standard huge-page allocations used by most conventional operating systems and architectures. With minor extensions to the page-table-walker structures to support these, and aid in their cacheability, we can reduce memory accesses per walk by 27%, or 56% for virtualised systems, without significant memory overhead.

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

Keywords: Virtual Memory, Virtualisation

1 Introduction

As working sets increase in size, page-table translation is becoming more expensive [7, 8, 12, 24], accounting for up to 50% of the execution time of emerging workloads [38].

Performing virtual-to-physical translation by using four-level radix tables, the standard solution in use today, has many compelling benefits. Each layer fits within a standard 4 KiB page frame, which can be allocated like any other user-space data, avoiding external fragmentation. This allows the structure to be stored sparsely: rather than mapping all 48 bits of the address space, requiring 512 GiB of space just for the table, any tree paths without data can be left unallocated. While other structures have been proposed, such as hash-table-based storage [21, 38, 40, 45], the predictability and cacheability [45] of radix trees for small working sets, and their ease of allocation into conventional page frames, has kept them as the structure of choice for modern systems.

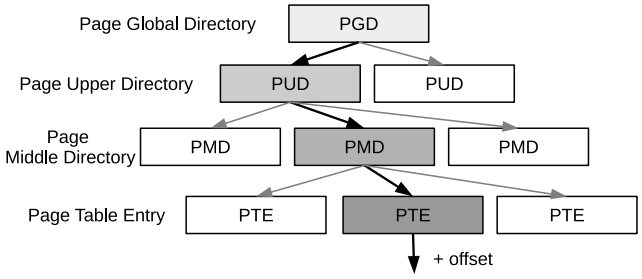


Figure 1. Virtual pages are translated to physical frames via a four-layer radix tree, each layer indexed by 9 bits.

Still, this sparsity results in costly worst cases, of four memory accesses per translation in standard setups, and 24 in virtualised systems [23]. Though many accesses can be hidden with caching, as workload sizes increase, the cost of long chains of high-latency memory accesses becomes untenable. With 5-layer translations becoming necessary [15], the worst case becomes 35 accesses.

We argue that the radix-tree structure is often stored too sparsely, causing unnecessary memory accesses. Conveniently, most modern architectures and operating systems also support huge pages [32] of 2 MiB in size for user data. We propose utilizing these to merge adjacent layers, translating 18 bits in a single memory access, rather than 9 bits each in two, in regions where the structure is densely occupied. We handle the implications of complexities in terms of common caching systems, allowing the overlapping of densified structures where favorable, and in combination with transparent huge-page support. Compendia naturally extends to virtualised environments, where the need for high-performance and easily compatible solutions is even more pressing [4, 29].

Compendia gives 5.5% speedup (41% maximum) across a large set of translation-intensive workloads in conventional setups and 18% (70% maximum) on virtualised systems. This is achieved by bringing down the average number of memory accesses per TLB miss from 1.39 to 1.01; very close to optimal. It combines well with transparent huge pages, where it can give a further 60% speedup on large datasets.

Figure 2. An x86-64 Linux page-table entry [1].

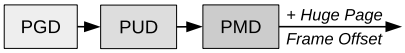


Figure 3. For a 2 MiB Huge Page, the radix tree loses a level, as the frame covers 9 bits more address space.

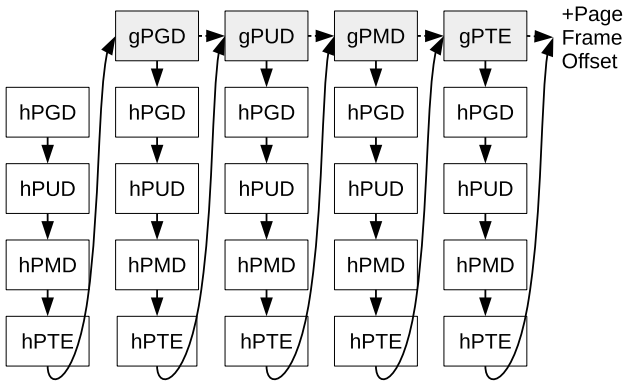


Figure 4. Two-dimensional page walks [23] on virtualised systems require up to 24 memory accesses for virtual-to-physical translation, as each guest physical address (gPA), both in the operating system's page tables and its actual memory access, is translated to a host physical address (hPA).

2 Background

2.1 Page-Table Walks

Systems with 64-bit word sizes typically use 48-bit virtual addresses, which are translated to physical frames using a four-layer radix-tree translation [14]. Each page is 4 KiB in size, which requires 12 bits from the address to access each byte within the page or frame; the other 36 bits are translated 9 bits at a time in the radix tree (figure 1). We use the first 9 bits to look up one of 512 64-bit word-sized entries in the Page Global Directory (in Linux terminology [14]). This gives us an element of the form shown in figure 2, including control bits and the physical frame the relevant 4 KiB PUD entry is stored in. We then use the next 9 bits to perform a similar lookup in the relevant PUD frame to find the frame of the relevant PMD, and then the PTE, giving the frame of the translated page, to which we add the 12-bit offset.

This four-layer hierarchy covers 256 TiB of addressable space. Although this is considerably larger than the physical memory of most machines, five-layer translations, to cover 128 PiB of space [15], are becoming increasingly necessary for high-end systems, especially when coupled with the use of memory-mapped I/O.

This radix-tree structure provides two useful properties. Splitting the translation between many different frames means we can leave parts of the hierarchy sparsely allocated, by storing zero entries to unpopulated parts of the page table. This avoids the page table itself taking up 512 GiB of memory for applications that typically do not use all of the space. It also allows the radix tree to be implemented via the same 4 KiB frames that virtual memory itself is allocated in.

Still, this gives a worst case of five memory accesses in total for a single load from a virtual address. This is tolerable for many workloads since these accesses are often very cacheable. Each TLB entry allows access to 4 KiB of memory by storing the relevant virtual-to-physical mapping. Page-table structures themselves are also very cacheable, and use dedicated caches for this purpose [7, 42], as each higher level covers exponentially more data: a single PMD entry gives access to 512 PTE entries, so if the PMD entry is cached then a single translation memory access reaches 2 MiB of data, and a single cached PUD entry reaches 1 GiB in two accesses. However, performance is tied to workload size, and large workloads with complex memory patterns spend the majority of their run-time handling misses [24].

2.2 Huge Pages

A widely deployed existing solution is Huge Pages. Typically, this is where 4 KiB, 2 MiB, and possibly 1 GiB frames are all supported as units of contiguous allocation. Each increase in size both removes a layer of the radix table (figure 3), from the least cacheable end, and allows each TLB entry to cover a wider portion of the address space, thus reducing the number of page-table walks as well as their cost. The Linux kernel provides transparent huge-page support, where allocation into a 2 MiB frame is implemented automatically when applicable, and can be treated identically to 4 KiB pages from the programmer's perspective.[1] Still, huge-page support is no panacea; translation overheads remain high for many workloads [24]. Heterogeneity in page size often degrades performance via physical-memory fragmentation [24, 26, 32, 33], and has limited support in many operating systems [22].

2.3 Virtualisation

The standard setup for virtualised systems, where multiple guest operating systems can run above a host OS, is a two-dimensional nested page table [23]: guest virtual addresses (gVAs) are translated to virtualised guest physical addresses (gPAs), which are translated to the system's true host physical addresses (hPAs). This gives the virtualised operating system the ability to map and unmap its guest physical pages, which are then in turn translated to the host's view of memory. As each translation in each dimension involves a four-level page-table walk, this multiplies to 24 memory accesses per virtual-to-physical translation (figure 4). With five-layer

[1] This size is chosen because 2 MiB of space can be indexed in 21 bits: 12 bits of standard 4 KiB frames, and 9 bits from the now-eliminated PTE.

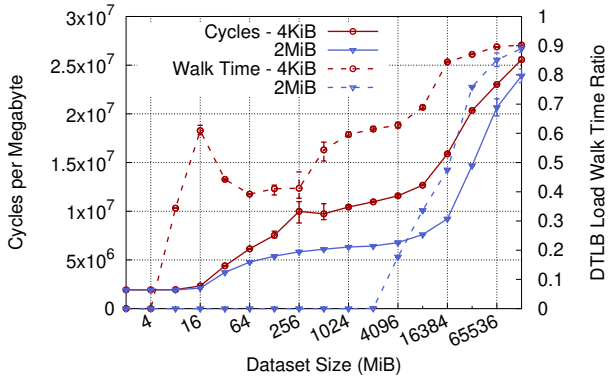


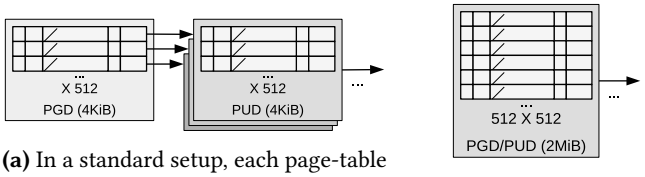
Figure 5. CPU cycles of execution time per megabyte of data input (left), and fraction of cycles with active page-table walks (right), on RandomAccess, with increasing dataset sizes for 4 KiB and 2 MiB pages. If execution time increased proportionately with input size, *Cycles per Megabyte* would be flat. When spikes in *DTLB Load Walk Time Ratio* correlate with *Cycles per Megabyte*, an extra layer of page-table-walker cache misses (PGD→PUD→PMD) occurs.

57-bit translations [15], this increases to 35 accesses. The alternative of shadow paging [43], where a one-dimensional mapping translates straight from gVAs to hPAs, is not implemented in today's systems; it requires heavy software cleanup to invalidate incorrect mappings on updates to the table, for every process in the guest OS [18].

For small workloads, many accesses are hidden through caching [7], via TLBs and page-table caches for both guest and the host tables. However, the linear performance loss with increasing data size in single-dimension translations becomes quadratic, rapidly slowing large workloads [29].

3 Motivation

As datasets grow, page-table walks become a significant cause of performance loss. In figure 5, with 4 KiB pages, a significant proportion of clock cycles are spent performing page-table walks, even with small data sets, on a modern Intel Skylake Xeon W-2195 server running the RandomAccess benchmark [27]. Up to 256 MiB, this is only the effect of a single memory-access miss (the PTE): accesses in the first three stages of the page table are in similar regions and so are cacheable [7, 42], the misses can be overlapped through having multiple page-table walkers, and so performance is only mildly impacted. The spike at 16MiB, reproducible both on multiple systems and with the number of iterations increased to extend run-time, therefore does not impact performance significantly. However, from 256 MiB onwards we see the effects of the third level (PMD) becoming uncacheable, and thus slowdown. Finally, the second level (PUD) starts to miss from around 8 GiB onwards, causing further detriment.



(a) In a standard setup, each page-table level is stored across 4 KiB frames, holds 512 elements, and thus translates 9 bits of the virtual address. One memory access is required to look up each level.

(b) Merging to a 2 MiB Compendia frame translates 18 bits in one access.

Figure 6. If we are willing to sacrifice some sparsity of our data structure, then levels of the radix tree can be flattened, reducing the total number of memory accesses.

Transparent huge-page support, with 2 MiB pages, eliminates this wasted time for small data sizes, as there are no PTE memory accesses, but by 4,096 MiB of input we start to see significant time spent on page-table walks. Even though the resulting code is still faster than with 4 KiB pages, we still see observable performance drops as a result. The gap narrows towards the right of the graph, as even transparent huge-page translation starts to require multiple memory accesses. A significant amount of performance is left on the table, with or without transparent huge pages, and the cost increases dramatically with increased working sets.

Four-stage radix-tree page tables were designed for an era when memory was in short supply; they allow for page-table mappings to be stored very sparsely. We argue that this sparsity is now unneccessary in many circumstances, and results in needless memory accesses. At the same time, the cacheability of such structures make radix trees difficult to beat. To avoid these unneccessary memory accesses, caused purely by the data structure being overly sparse, we can merge multiple levels together into a single level, or node, of the radix tree (figure 6), whenever a significant proportion of entries in a given region are mapped (i.e., when lack of sparsity causes few memory savings). This means we only need to perform a single memory access, instead of two, for those merged levels, reducing the number of memory accesses and thus the cost of each page-table walk.

4 Compendia

Compendia reduces sparsity in regions of the page table where this sparsity is unwarranted, reducing the number of memory accesses needed for a virtual-to-physical address translation, in particular, those that need to access DRAM. Adjacent layers of the table are merged once their occupancy reaches a threshold. This merging can occur either at the top level between the PGD and PUD (figure 6), between the middle PUD and PMD layers, and/or between the PMD and PTE layers closest to the translated frame.

Compendia frames fit within existing physical frames supported by common operating systems; that is, 4 KiB and

2 MiB regions. This simplifies implementation and limits fragmentation [34], by allowing the operating system to use its existing layout systems to allocate frames for the page table. Conveniently, 2 MiB is the correct amount of space needed to merge two layers, transforming a set of 513 512-entry 4 KiB frames into one 262,144-entry 2 MiB frame.[2]

As Compendia does not change the format of user pages, the only change that needs to be made to the operating system kernel is within the page-table walker logic, and the page-table structure itself, to support, make allocation and merging decisions, and interpret the new format. Though hardware page-table walkers need modification to support indexing into the new flattened structure, this is a trivial extension of the state machine logic, and the TLB is unaltered.

4.1 Selective Densification

Merging levels of the page table will not always be desirable; for many applications, the page table is sparsely populated, as much of the address space is not allocated. At one extreme, when all virtual addresses within a 1 GiB region (512 × 512×4 KiB) of the page table, coverable by two 4 KiB-based layers, are mapped, then all possible 513 4 KiB frames within those two layers will be backed by physical frames. In this case, merging these into a single 2 MiB layer will cause no further space utilisation. By contrast, if only a single 4 KiB page of virtual address space is mapped within a 1 GiB region, then the two mapped 4 KiB frames in the two layers will expand 256-fold.

Our implementation densifies a layer with its child layer when an eighth of its 512 children are non-zero.[3] This strategy avoids measuring the occupancy of any other frames in the system, and strikes a balance between occupancy and density. It also avoids measurable overhead to check this condition, as no merging occurs under alterations to only the PTE layer of the page table. The theoretical worst overhead from this is when we have 52 non-zero entries in our frame, each containing only a single 4 KiB mapping within them. This would cause a 5× overhead once the mapped data is taken into account. Still, this pattern of sparsity is unlikely, and so overheads are typically low.

We use the same threshold throughout the hierarchy: that is, for merging PGD and PUDs, PUDs and PMDs, or PMDs and PTEs. This is despite the tradeoffs of such merges being different. A single merged PGD/PUD layer will be used for the entire hierarchy, universally reducing the maximum number of memory accesses for a translation by one. However, most PGD entries are likely to be cached, as only a small fraction of the 48-bit virtual address space will be in active use for most workloads, and so PMD/PTE merges are more likely to reduce memory accesses to DRAM. Other thresholding options are available, and can be built into a given software operating system without impacting any hardware; for example, many workloads could be densified when a large contiguous block is requested all at once.

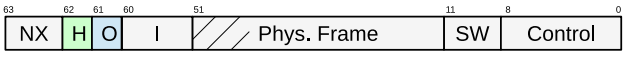


Figure 7. We add two bits within the ignored region [1] of our radix-tree layers: the first, H, indicates that we are pointing to a merged 2 MiB frame. The second, O, is used to denote overlap, to improve caching (section 4.3).

In our implementation, we only merge two 4 KiB layers into one 2 MiB layer, though we do this at multiple points in the hierarchy. We could go further, densifying three layers into 1 GiB. With current RAM sizes, this would often come at significant overhead and fragmentation, though future systems may benefit. Though this happens in none of our workloads, theoretically, parts of the page table may become significantly *less* occupied over time. In this case, the 2 MiB Compendia frame can be unmerged into 513 4 KiB entries, with each 4 KiB page-table frame that is entirely zero left unmapped. For hysteresis, this should occur at a lower threshold than the merging ($\frac{1}{16}$ in our case), and many other heuristics can be added to kernel software to best make use of densification and undensification.

4.2 Huge and Overlap Bits

To identify that we are about to access a Compendia frame, and thus we should use the following 18 bits of the virtual address to index into the next frame rather than the following 9,[4] we add the H, or *huge* bit into our base register, PGD, and PUD entries, to indicate a merging of PGD/PUD, PUD/PMD, and PMD/PTE, respectively (figure 7). We also add a new O, or *overlap* bit, to indicate that we should use the *previous* 9 bits along with the next 9 bits to index into the next level, so that our Compendia frames can overlap (section 4.4) with each other while responding well to caching (section 4.3).

These bits are stored in the currently ignored region of table entries, as shown in figure 7. Alternatively, they could be stored in the least significant bits of the page-table frame index: when bit 7, or `_PAGE_BIT_PSE` [1] is set, we know that the following frame is large, as currently used for huge-page support of user addresses. Since Compendia frames, like huge pages, are aligned to 2 MiB boundaries, the least significant bits of the frame index within the page-table entry (or PGD, PUD or PMD) can be used to distinguish the states.

[2]The 513th entry is used to index the other 512, and so disappears once merged into a single flat 2 MiB index.

[3]We chose $\frac{1}{8}$ before experimentation to avoid cherrypicking, and to control the theoretical worst-case overhead. Varying this threshold yields similar performance numbers across a wide range of values since the page table is locally very dense where merges occur.

[4]For a 48-bit address, 9 bits are used to index into each of the PGD, PUD, PMD and PTE, with the final 12 bits indexing into the 4KiB user frame.

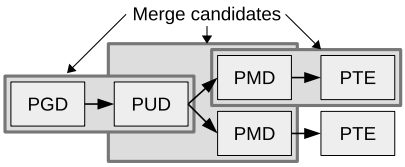


Figure 8. When multiple levels feature the requisite level of density, then we are faced with a choice in how to densify the merged radix-tree levels. Typically, it is best to merge closer to the PTE, as the number of entries increases exponentially down the hierarchy, and thus entries become less cacheable.

4.3 Interaction with Caching

Theoretically, in a densely occupied region of memory, Compendia can bring the total number of memory accesses required for address translation down from four to two, by merging the PGD with the PUD and the PMD with the PTE. However, this ignores the impact of caching. Typically, the page-table-walker caches [42] will attempt to cache the first three layers (the PTE stores the final mapping, which is implicitly cached by the TLB), and for small working-set sizes, all used elements from the first two levels will be cached entirely. Often the benefit of Compendia will be to avoid a separate memory access of both the PMD and PTE, by merging into a combined PMD/PTE, and only for larger workloads, especially when using transparent huge pages to eliminate the PTE layer, will the PGD or PUD start to be impacted.

Compendia frames are designed to minimize the total number of memory accesses, with the expectation that caching will hide much of the latency to earlier levels of the radix tree, and so that merging of levels never results in worse behaviour than a baseline system even with caching. This includes when levels can overlap, as discussed next.

4.4 Overlapping of Compendia Frames

We can densify at multiple different levels at once. In the example in figure 8, we are presented with two options: either densify the PGD/PUD and PMD/PTE, resulting in a maximum of two memory accesses, or densify the middle layer, resulting in three. Any other combinations would cause overlap in the bits translated at each level, adding redundancy into the data structure, without reducing memory access count down a linear path. Still, with complex patterns of density down different branches, overlapping may be favourable, and thus we must support it in a way that caches effectively.

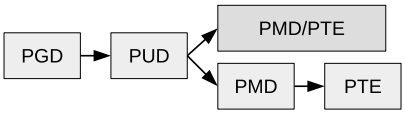
4.4.1 Linear Overlap. The choice of merging PGD/PUDs and PMD/PTEs, over separate PGDs and PTEs with a merged PUD/PMD pair (figure 8), has two benefits in its favour. First is the reduction in the total number of layers. The second is in terms of cacheability. As the total number of entries, and thus frames, increases exponentially (by a factor of 512 each round) along the four-layer hierarchy, PMD layers are less cacheable than PUD layers, as we are likely to access more PMDs than PUDs in a given working set. A merging of the PUD and PMD layers effectively eliminates the PUD memory access, and a merging of the PMD and PTEs eliminates the PMD access. But if the PUD element is already likely to be cached, this is of less benefit. We should prioritise merging layers closer to the final translation when there is conflict.

4.4.2 Branching Overlap. In reality our data structure is a complex, branching tree, with 512 children at every layer, rather than the linear section shown in figure 8. This makes our decision more complex, as shown in figure 9. In these examples the PUD is occupied enough to be a candidate for PUD/PMD merging, as is the PMD at the top of the diagrams. However, the bottom PMD/PTE pair does not reach this threshold. If we avoid overlap, then we are presented with two options. The first, figure 9(a), keeps the densification in the leaf nodes of the tree where it is most likely to be useful, but gives no benefit to the bottom PMD/PTE pair. The second, figure 9(b), benefits all, but is likely to underperform in the presence of a cache, relative to figure 9(a).

To get the best of both worlds, we can overlap (figure 9(c)) by storing redundant copies of data in each level. This means that, in our PUD/PMD in figure 9(c), there will be 512 entries all pointing to the base of the same PMD/PTE table—the "PMD" region displayed in figure 10. We index into the PMD/PTE by using both the previous 9 bits we used for the PMD part of the PUD/PMD, as well as the next 9 bits for the PTE part that would otherwise be standard, and so we denote this using the 0 or *overlap* bit in the header metadata (section 4.2). The next time we access a different element within this PMD/PTE, the relevant base will be stored in the cache. This means that, though we gain no benefit from the merged PMD/PTE table in our first access, duplicating the merging at the PUD/PMD layer, future cached accesses can directly load from the PMD/PTE layer with a single access, unlike with split PMD and PTE layers, which require two.

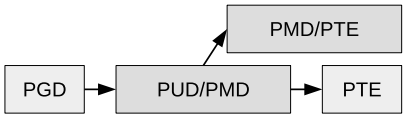
4.4.3 Multi-Layer Branching Overlap. We may have a scenario as shown in figure 11(a), where we have three layers of intersecting densification. The setup described so far causes a redundant memory access: we could reach a PTE value in the top merged PMD/PTE in two memory accesses (one in the PGD/PUD and one in the PMD/PTE), but as the "PUD" layer contains 512 copies of the pointer to the PUD/PMD frame, we indirect through three.

To solve this, when the next-but-one layer is densified, we replace individual elements in the 512-element range that correspond to a dense later level with a direct pointer, as shown in figure 11(b), while leaving the 'overlap' value to the PUD/PMD in place for any layers without merged PMD/PTEs. If the PUD layer either contains few non-zero entries, or mostly points to merged PMD/PTE entries (the two are equivalent here, as neither benefit from PUD/PMD densification), then it will be eliminated entirely. The PGD/PUD



(a) Density is pushed to the end of the tree where it is most likely to be useful, but the longest path is through four frames.

(b) The longest path is three frames, but with sparsity at the end, increasing memory accesses under caching.



(c) To get the best of both worlds, we overlap Compendia frames, where information is redundant between levels.

Figure 9. With different density patterns down each path, there is a conflict over the best pattern for each branch. In this example, we assume that the top PMD/PTE level in each diagram is sufficiently occupied to be dense, but the bottom one is not.

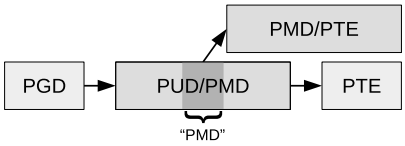


Figure 10. Instead of storing 512 different offsets into a Compendia frame within the previous level under overlap, we store 512 copies of the base, each carrying the 'O' bit to indicate the need to index with the previous 9 bits as well as the next. This adds the base of the entire PMD/PTE frame to the cache for next time, avoiding PUD/PMD lookup.

will then point to the relevant 4 KiB PMD entries instead, for layers that have not merged into PMD/PTE layers.

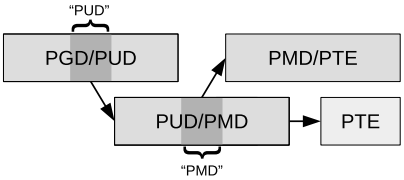
An access into the top PMD/PTE may still go through the PUD/PMD layer, even though we can access it directly from the PGD/PUD. The PUD/PMD layer may be cached, avoiding the lookup in the PGD/PUD. Still, this does not involve redundant work: we simply look up the relevant element for the PMD/PTE using the offset value in the "PMD" layer of the PUD/PMD instead of the PGD/PUD.

4.5 Huge-Page Support

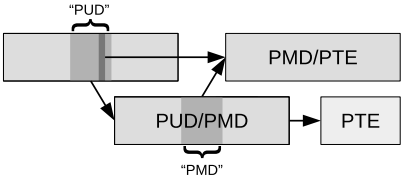
Transparent huge pages (THPs), where user pages can map to 2 MiB frames, are conceptually similar to Compendia: both involve combining 4 KiB and 2 MiB pages to limit virtual-translation overheads. Each has value separately or together.

For an access along a fully densified path, a Compendia translation takes two memory accesses; in existing page tables, three memory accesses are required to translate a 2 MiB huge page. Still, this ignores the effect of caching, both in page-table-walker caches and the TLB.

For smaller working sets, the only cache misses in a walk will be at the PMD and PTE levels. Both Compendia and THPs typically turn these into a single miss instead of two: Compendia through layer merging, and THPs through eliminating the PTE entry, by 2 MiB-aligning the user frames. A single TLB entry in a THP setup will cover an entire 2 MiB frame; with a Compendia-only setup, each 4 KiB user frame will be translated and stored in the TLB separately.



(a) Naïve overlap of Compendia frames could cause superfluous loads: in this example, we travel through three dense frames.

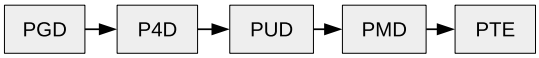


(b) We can fix this by replacing individual entries within an expanded 4 KiB PUD/PMD range in the PGD/PUD table directly with the PMD/PTE frame, if it exists, leaving only sparse entries, with 4 KiB PTE and PMD frames, to be directed through the PUD/PMD frame.

Figure 11. We can overlap dense frames to support complex branching sparsity and density throughout our page table, without sacrificing number of memory accesses.

Compendia offers a simpler upgrade path than THP for operating systems with heavy reliance on 4 KiB pages [22], as it means that applications do not need to be exposed to 2 MiB page frames. Since 2 MiB frames are limited to the page table itself in Compendia, less fragmentation [34] in frame allocation occurs: each 4 KiB user page can be (de)allocated separately, and a single 2 MiB Compendia frame can cover 1 GiB of data, rather than requiring 512 separate 2 MiB THPs.

Still, the two techniques coexist well. As with conventional page tables, we can store huge-page frame mappings in Compendia's PMD layer, eliminating the PTE layer for that frame, making a PMD/PTE merge useless, and thus making a PUD/PMD merge desirable. With Compendia and transparent huge pages combined, we can translate an address with a single memory access provided the PGD entry alone is cached, removing the need to have a cache large enough



(a) Five-level page tables introduce a new layer, P4D, to increase virtual-address size from 48 to 57 bits.

(b) If most entries are for 4 KiB pages, then density is best pushed closer to the PTE, and so the PGD is left sparse.

(c) If most PMD entries down a path are to huge pages, then all levels can be made dense even with an odd number of layers.

Figure 12. Compendia also extends to future five-layer page tables, despite their odd number of levels.

to cache all PUD entries. By contrast, THP and Compendia alone only allow this ideal case when all PGD and PUD entries are cached (but neither needs to cache PMD entries).

We face the same problem under coexistence of 4 KiB and 2 MiB huge pages in a Compendia system as we do with overlapping PUD/PMD and PMD/PTE pairs in figures 8 to 11: what is best for the 4 KiB user pages (often a PMD/PTE merge) is not best for a 2 MiB page, which has no PTE. We use the same solutions, to allow efficient memory accesses for both in a shared-memory hierarchy: overlap bits are used when a huge page is placed within a PMD/PTE-merged region of the page table, and the huge page is pointed to directly within a PUD/PMD merge. Huge pages are treated as 'zero' entries for the purpose of PMD/PTE merges, to avoid triggering unneccessary merging.

4.6 Five-Layer Page Tables

When extended to five levels [15] (figure 12(a)), it is impossible for translations of 4 KiB pages to travel entirely via 2 MiB Compendia frames.[5] This means that even down a single linear path we must make a choice over which of the levels to densify. The top-level PGD is highly cacheable, as each entry covers a large portion of the address space, so density is best pushed to the lower levels (figure 12(b)), where it is more likely to reduce the number of accesses to main memory.

However, this ignores that this five-level setup, designed for systems with very large addressing requirements and thus a large amount of memory, will likely also be using huge pages for data. In this case, the PTE level is unnecessary, and so most memory accesses will likely go through a four-level PGD-P4D-PUD-PMD translation. At this point, a merging of PGD/P4D and PUD/PMD is the most sensible setup; Compendia can adapt to these dynamically, with coexistence of 4 KiB and 2 MiB pages, via overlapping (section 4.4).

[5]Without overlapping, in which at least one of the layers will not reduce the number of memory accesses relative to a 4 KiB level.

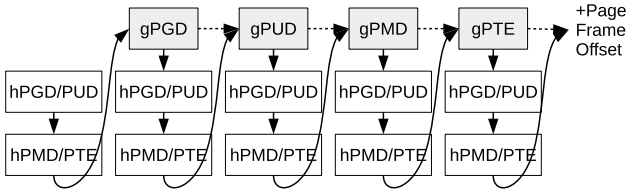
4.7 Virtualisation Support

In a virtualised setup, the four-memory-access worst case for a single layer of translation is replaced with a 24-access worst case for a dual-layer setup (figure 4). Compendia can reduce this at both layers. Regardless of the density distribution within the virtualised system's mapping from guest virtual addresses (gVA) to guest physical addresses (gPA), and even if the guest does not support huge pages in any form, including within the page table, we can still densify the translation layer from guest physical addresses to host physical address (hPA). Indeed, there is little need for sparsity at all in this intermediate translation layer [4]. Since the utilised guest physical addresses can be allocated like true physical addresses, starting from 0 and moving upwards, past a small threshold for low-memory virtualised systems, we can store our hierarchy entirely as two-level Compendia frames. A linear mapping would suffice [4], since no sparsity is necessary. However, Compendia in a radix-tree format allows intercompatibility using the same setup as a non-virtualised system, and avoids fragmentation [34] by allowing all system-level allocations to fit into standard 4 KiB and 2 MiB frames.

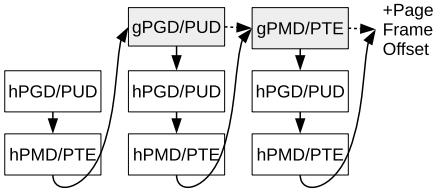
Densifying at the host level alone, in its gPA to hPA mappings, allows us to reduce the worst case from 24 to 14 accesses (figure 13(a)). However, as with a conventional single-layer mapping, depending on density within the guest's gVA to gPA mapping, we can also densify further within that layer. This can bring the worst case down further, to 8 total accesses (figure 13(b)). Again, as with the single-layer version, many of these accesses may be cached, and the gPMD/PTE and hPMD/PTE may be the most useful merges in practice; typically higher levels will be cached both with and without Compendia. Still, we will see that with Compendia the average number of memory accesses reduces significantly, even where the non-virtualised system faces little improvement, due to increased cache pressure without Compendia.

4.8 Summary

This section has introduced the complexities and implementation details of Compendia, where we selectively densify groups of two layers of 4 KiB page table entries into 2 MiB huge pages, able to cover 18 bits of address space in a single access instead of 9. Ignoring caches, for densely populated regions we can exchange four memory accesses on page-table walks for two, or 24 for eight in virtualised cases. With caches, we can allocate densified Compendia frames in such a way as to optimise cache coverage, by redesigning metadata in areas where the new dense levels overlap to store the most useful value in the cache. In the next section, we will look at how this affects the performance of real workloads, in the presence of a typical caching environment.



(a) Even if a workload's page table is sparsely populated, or does not support 2 MiB pages at all (within the page table or data), Compendia still gives benefit, reducing 24 accesses (figure 4) to 14.



(b) If the guest page table also features some regions of density in its mapping, this can be reduced further to eight.

Figure 13. There is no need for guest physical addresses (gPAs) to have a sparse mapping to hypervisor physical addresses: we can allocate gPAs starting at 0 and moving upwards. This means we can significantly reduce the number of memory accesses in virtualised setups.

5 Experimental Setup

To simulate our system, we use BadgerTrap [19], a Linux-kernel modification to instrument TLB misses. We added support for our dense Compendia format, along with relevant structures in the page tables. Our simulator then counts the number of memory accesses required to service each TLB miss (64-element L1, 1024-element L2 TLBs), both with and without Compendia, for both standard and virtualised setups. These memory accesses go through a cache simulation, where we assume a 64-entry 8-way-associative unified cache for PGD, PUD and PMD layers, sized to be realistic compared with real-world systems [42]. Cache-hit costs are assumed to be negligible relative to main-memory accesses. For virtualisation, the cache and TLB store direct gVA-hPA mappings, whereas the page table uses separate two-dimensional gVA-gPA and gPA-hPA mappings, as is typical [23], and a separate 64-entry 8-way-associative cache for gPA-hPA mappings.

We ported BadgerTrap [19] to Linux Kernel 4.4.0, running on Ubuntu 16.04 on an x86-64 machine, and ran workloads on top of this. We use this to generate the number of memory accesses per TLB miss, both before and after Compendia support, and both with and without virtualised nested page tables. To estimate speedup, we first collect cycles spent on page misses, by using Perf to access hardware performance counters, averaged over three successive runs. On an out-of-order superscalar this is not a direct measure of the cost of memory translation, since multiple translations can occur simultaneously with computation, so we combine this with a per-workload "savable page-walker cycle" factor, calculated using the method from Guvenilir and Patt [20]. We then take the time spent on page walks, the "savable page-walker cycle" factor, and the reduction in page-table memory accesses from our BadgerTrap simulator, and interpolate to derive speedup.

We also measured the overhead of a Compendia merge in isolation. This takes approximately 20 microseconds. In figure 16(a), all page-table sizes are below 30 MiB and so merging to 2 MiB frames means at most 15 merges. The total cost of this is therefore less than 3 milliseconds for workloads that take hundreds to thousands of seconds to complete. Since this is smaller than the noise of successive runs, we do not consider it further in the evaluation.

We look at the translation-bound workloads evaluated in two recent papers [20, 38]: RandomAccess taken from HPC Challenge [27] with 8 GiB of input unless otherwise specified, Graph500 [30] (-s 24 -e 20), SPEC CPU2017 [13] using ref inputs, GraphBIG [31] using the standard 8 GiB synthetic graph, XSBench [41] (large), and Dbx1000 [46] (TPCC -n32 and YCSB -s20000000). We first look at 4KiB pages within applications, to allow simulation on workloads with moderate dataset sizes while still requiring multiple accesses per miss. We then combine with transparent huge pages at large dataset sizes, to show Compendia more generally.

6 Evaluation

In a standard setup, Compendia support brings down the average number of memory accesses per TLB miss from 1.39 to 1.01. In virtualised systems, an average of 3.77 is reduced to just 2.04, very close to the ideal of 2. This results in speedups of 5.5% and 18%, respectively.

6.1 Analysis

Figure 14 shows average number of memory accesses per TLB miss before and after Compendia support for a non-virtualised system, along with speedup. An average of 1.39 memory accesses per walk is reduced to just 1.01, and is rarely above the ideal of a single memory access per walk. This results in a geomean speedup of 5.5% (maximum 41%).

RandAcc gains the biggest speedup, and largest reduction in average number of memory accesses per walk. This is because it follows a very random memory-access pattern, and thus its TLB misses are both frequent, and not temporally local in either of the PMD or PTE levels of the four-stage hierarchy. While the PGD and PUD levels are usually cached, since the dataset fits within few entries at this stage, Compendia support effectively eliminates PMD misses. Graph500 closely follows; it too spends a large proportion (70%) of its execution time on page-table walking, and so the significant reduction in memory accesses gives significant benefit. By contrast, though the GraphBIG workloads (BFS-PR) are also graph workloads and also see a large reduction in memory

accesses per miss, their speedup is smaller but still significant; the dynamic graph structure they use is slower than the static CSR structures used in Graph500, but more local, and so TLB hits are more frequent. This is also true for Dbx1000, where a similar memory-access improvement to Graph500 results in only a moderate performance improvement.

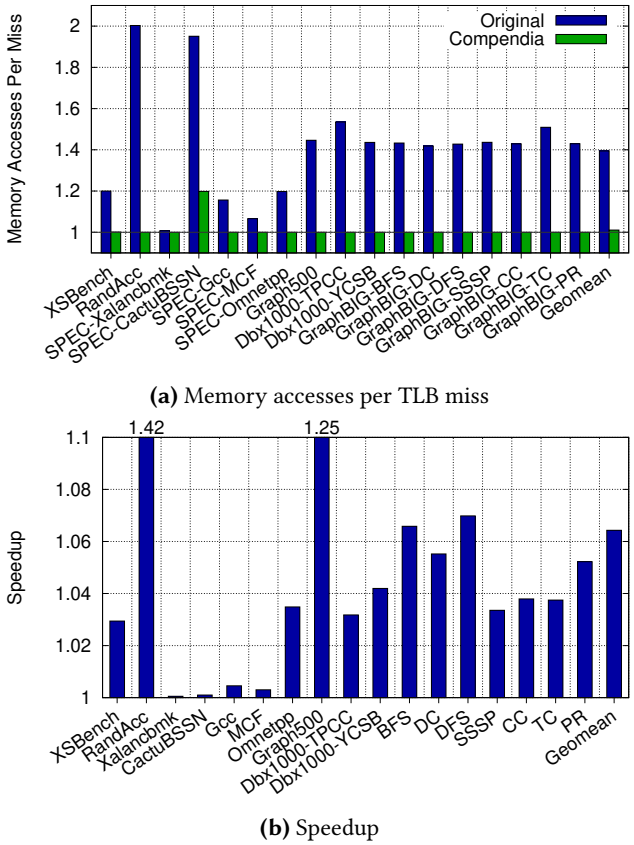
Xalancbmk sees no significant speedup. This is because its memory accesses are local within the page-table cache, and the dataset is small, and so even though the workload spends 7% of its execution time on page-table walks, the PMD entries are cacheable, and so it reaches an ideal number of memory accesses per walk regardless of Compendia support. Many of the other SPEC CPU2017 workloads show the same to a lesser degree; their working-set sizes are often small (figure 16(b)) and/or local, and so we should expect comparatively less benefit than for workloads with larger data sizes that use multiple PMD and PUD entries.

All but one workload has its number of memory accesses per TLB miss reduced to ideal levels; the exception is CactuBSSN, which requires either a larger 128-sized page-table cache, or combining Compendia with transparent huge pages, to reduce to one access per miss. Still, for this particular workload, the performance impact is negligible, as though it often requires multiple loads per TLB miss, TLB hits are common.

The results shown in the figures assume the default $\frac{1}{8}$ threshold (section 4.1), though performance, and memory consumption, is stable with varying thresholds. Extreme values ($\frac{7}{8}$) avoid any densification for workloads with small working sets consisting of many small objects, like Xalancbmk, and thus any speedup or memory overhead. Workloads that allocate dense blocks of memory, like RandAcc, still achieve the majority of their total performance improvement, reduced slightly by levels closer to the root being left undensified even with dense occupancy lower down. Low values of $\frac{1}{16}$ give similar performance and memory overhead to the default; however, densification of all levels regardless of occupancy, with no 4 KiB levels, causes table size to exceed user-allocated memory in some cases in our test set.

6.2 Performance Versus Input Size

In figure 15, we look in more detail at how dataset sizes affect memory access per miss and thus performance improvement, and combination with transparent huge pages. On RandAcc, until 128 MiB, Compendia support does not affect the workload at all; the PGD, PUD and PMD layers are cached, and thus no memory accesses are eliminated. From 128 MiB onwards, however, as previously observed in figure 5, we start to see slowdown from misses in the PMD layer that we can eliminate by merging the PMD/PTE layers, returning to a single memory access per page-table miss. In effect, Compendia allows a 512-fold increase in dataset size before memory accesses start increasing past this ideal, by improving the utility of cached elements by the same amount. From 32 GiB onwards, while Compendia still improves the



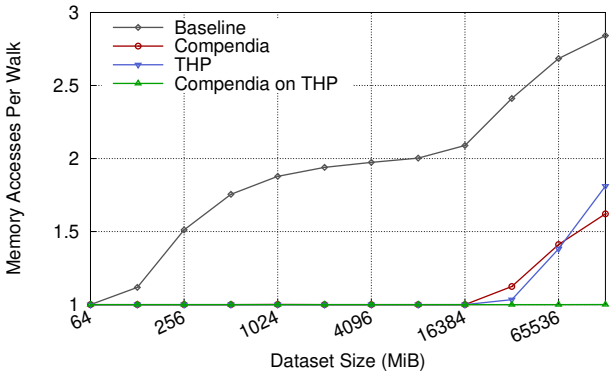
(a) Memory accesses per TLB miss

(b) Speedup

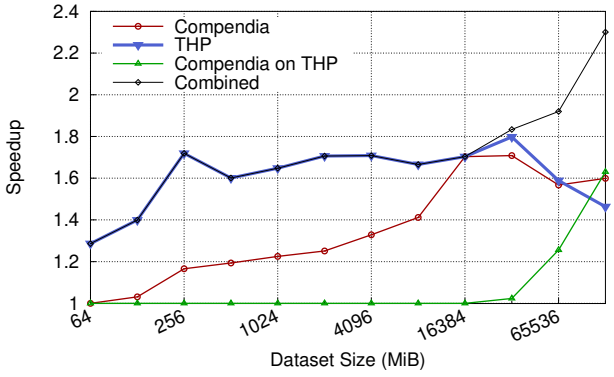
Figure 14. Average memory accesses per TLB miss, and speedup, before and after Compendia support.

situation, both it and transparent huge pages cannot achieve the ideal of 1 access per translation without being combined.

In terms of accesses per miss, Compendia performs similarly to transparent huge pages (THP), even though theoretically for very large data sizes the latter requires three accesses instead of two, as with the data sizes observed. Both, in effect, eliminate the PTE access, either by merging it with the PMD or removing it entirely respectively. Compendia achieves this while reducing the number of huge pages compared with THP, limited by fragmentation in practice [34], by a factor of 512, by allocating all application data in 4 KiB frames. Still, for small data sizes up to 8 GiB, transparent huge pages perform better despite the similar number of accesses per walk, as though they perform similarly in the page-table cache, the larger, more cacheable 2 MiB TLB entries for the 2 MiB user pages in transparent huge pages cause fewer walks to occur when some locality is observed. However, once this locality disappears with larger data, these benefits vanish, and their performance is comparable. Past 32 GiB, the best performance can only be obtained by combining the two approaches, so that only a single PUD/PMD merged access is required per walk. Larger caches increase the point where both are needed by a linear factor, whereas

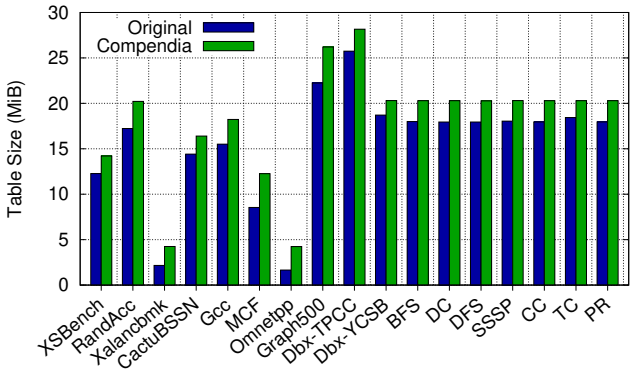


(a) Memory accesses per TLB Miss

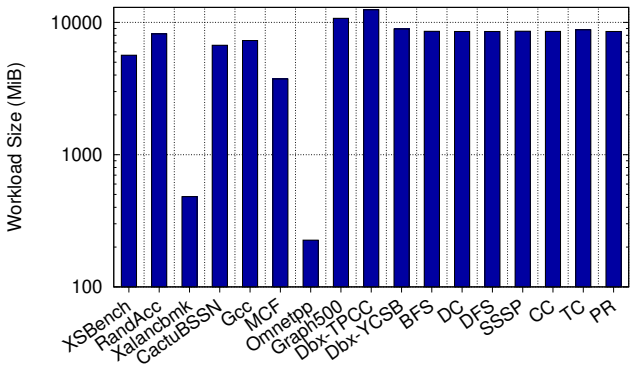


(b) Speedup

Figure 15. Average memory accesses per TLB miss, on RandomAccess at various sizes, before and after Compendia support, along with associated speedup, and the speedup with transparent huge pages. The "*Compendia on THP*" bar has THP as its baseline, and so there is no additional speedup to be gained until 32 GiB onwards.



(a) Page-table size, before and after Compendia support



(b) Size of total allocated user data

Figure 16. Memory consumption of page tables, contrasted with total memory consumption per application.

the switch to Compendia and THPs each provide an effective factor of 512× improvement in capacity respectively.

6.3 Memory Consumption

Figure 16(a) shows memory consumption of the page tables before and after Compendia support; figure 16(b) shows the total memory consumption of the application. On average, memory consumption of the entire application is only increased by 0.06% as a result. Still, some workloads (Xalancbmk and Omnetpp) do see a significant increase in the size of the tables themselves. This is because, at small table sizes, the densification of even a single group of 4 KiB pages into a 2 MiB page is enough to make a significant impact on the table size if few of those 4 KiB pages in a region are occupied. This means that the heap in these examples only uses a small fraction of the 1 GiB region covered by a single PMD/PTE merge, but over 12.5% of the PMD's entries are allocated and thus we densify. Still, even in this case this is a valuable tradeoff, since we still see speedups for Omnetpp, and so setting the threshold higher is likely to be undesirable. As a single 2 MiB Compendia frame is sufficient to cover up to 1 GiB of user data, the doubling in page-table size for Omnetpp and Xalancbmk has a negligible impact on overall resident set size. Omnetpp is close to the minimum merge threshold, and so for smaller workloads that do not reach the threshold will see no change and thus no overhead.

6.4 Versus Elastic Cuckoo Hash Page Tables

The most relevant recent work on the topic is Elastic Cuckoo Hash Page Tables [38], which trades off four sequential radix page-table walks for three parallel hash-table accesses. This means that the worst-case latency for a translation is a single memory access, but, unlike a radix tree, there are no highly cacheable upper layers, and so three accesses are always performed on a page walk to translate a 4 KiB page.

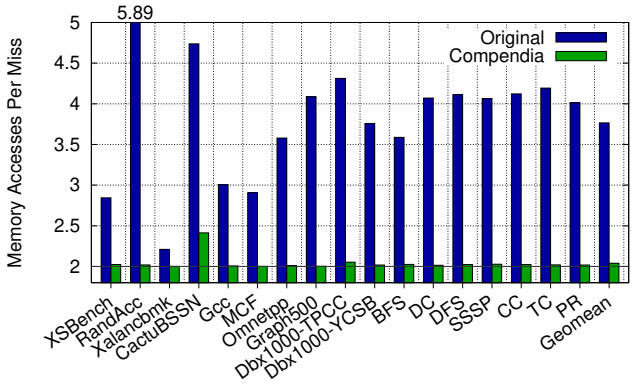
In figure 14, once the same cache used for a standard radix table is applied, Compendia almost always also has only a single DRAM-access latency. Since neither technique alters the TLB, and thus suffers page-table walks at the same point, Compendia achieves the same level of performance. In systems that are bandwidth-bound, particularly those with

multiple cores or multiple page-table walkers, then Compendia will improve performance by reducing bandwidth demands by a factor of three.

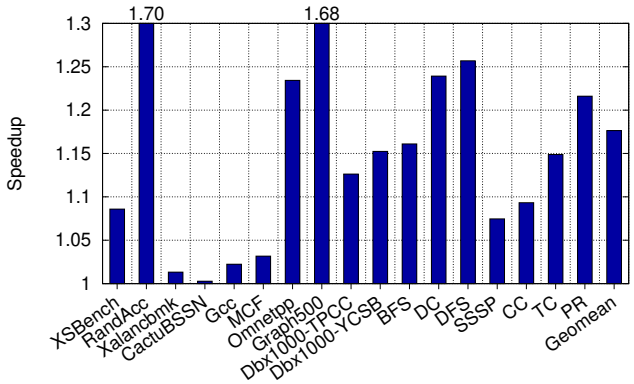
Still, in the worst case, even with full density in the page-table structure, Compendia has a worst case latency of two, whereas elastic cuckoo techniques can always access their translation in one round. This only applies for large working sets with random accesses, however, and can be mitigated by combining Compendia with transparent huge pages (figure 15). Compendia also uses the same infrastructure as existing radix trees, and fits in the same frames currently used by modern operating systems, avoiding problems of compatibility and memory fragmentation.

6.5 Virtualisation

In virtualised systems, the overheads of multiple walks in the nested layers become quadratic rather than linear. Figure 17 shows the effect on both memory accesses, and associated speedup, for our set of benchmarks. Even the workloads that were negligibly affected in the non-virtualised case show speedup. Most workloads, save for CactuBSSN and Dbx1000, are negligibly above the ideal two accesses per translation, and the remainder could be improved further by using transparent huge-page support and instead merging the PUD and PMD layers in the two-dimensional hierarchy. The geomean speedup is 18%. Conversely, virtualisation overhead without Compendia is 29%. With Compendia this drops to just 9% (or 15% compared to a Compendia baseline). The extra memory overhead on top of figure 16 is negligible, as the gPA-hPA translation is naturally highly dense [4].



(a) Memory accesses per TLB miss



(b) Speedup

Figure 17. Compendia evaluated on a virtualised two-dimensional nested page-table setup.

6.6 Summary

Compendia support brings about a speedup of 5.5%, and 18% for virtualised systems, while using only 0.06% more memory and while fitting in the same frame allocations and cache infrastructure [42] as existing radix-tree implementations. Even for workloads with large datasets, this typically brings memory accesses per translation down to ideal levels, once caching is taken into account, and Compendia is effective with or without transparent huge-page support.

7 Related Work

Bhattacharjee, Lustig and Martonosi [10] present a comprehensive introduction to the topic of address translation. Below, we categorize the most relevant topics.

7.1 Alternative Virtual-Memory Translation

Compendia extends conventionally used radix-table translation mechanisms. Other mechanisms that deal with the sparsity of virtual-to-physical tables in other ways have been proposed. Huck and Hays introduce the concept of hashed tables [21]. Clustered Page Tables [40] extend this to merge close entries, to provide locality. Both techniques store one table per system, rather than per address space, to avoid storing and resizing many variably sized structures in contiguous physical memory. Elastic Cuckoo Page Tables [38] are a more recent technique, storing one table per address space, by using cuckoo hash tables accessed in parallel, thus trading off bandwidth for latency. Hashed tables can perform fewer memory accesses than radix-tree implementations [45], but suffer from challenges when handling huge pages.

Paging is not the only mechanism by which physical memory can be allocated. Continuous segments can instead be allocated to applications; a recent example of this are Direct Segments [8], where a single large mapping in physical memory is allowed per application. This can be very efficient, by avoiding translation for that region entirely, but it requires application changes, causes fragmentation in memory through requiring contiguity and is unsuitable for applications without a single distinct large region. CARAT [39] replaces paging with software management of physical memory, through a trusted compiler interface. This avoids the need of any paging hardware, at the expense of slowdown.

7.2 Page-Size Changes

The overheads of address translation can also be reduced by supporting larger pages. This can cause applications to miss less frequently than Compendia alone, which is orthogonal to page sizing and can be used concurrently with such techniques. Still, supporting multiple page sizes in user-space can cause compatibility issues and fragmentation [34, 38].

A degree of 2 MiB huge-page support is now common in most operating systems; Navarro et al. [32] discuss the implementation of multiple different page sizes in an operating system. HawkEye [33] and Ingens [26] extend standard transparent huge-page support by handling many of the challenges that supporting multiple page sizes transparently at the application level can come with. Tailored Page Sizes [20] extend the concept, so that any power-of-two frame allocation can be supported and translated. Redundant Memory Mappings [25] achieve a similar effect using a range table.

7.3 Caching

Address translation is cached at two levels. TLBs store the direct virtual-to-physical mapping, avoiding any memory accesses. One way to reduce overheads of the page-table walk is therefore to increase the size of the TLB. POM-TLB [36] and CSALT [28] store a cache of translations in main memory, to access many recently used translations in a single memory access. Other recent work on TLBs readapts them to better support multiple page sizes [16], codesigns them with allocators to encode partial coalescing [35], and actively defragments page allocations [44] to assist range-storing TLBs.

The second level is within dedicated page-table-walker caches, which store partial translations of the first three levels of the four-level hierarchy. This caching of sublevels [7] is a large part of why the worst-case performance of radix trees is rarely seen with small datasets. Van Schaik et al. [42] reverse engineer the caches found in today's systems. Caching can be improved through alternative designs: Bhattacharjee [11] reallocates physical frames for adjacency, to improve coalescing caches, and shares MMU caches between cores.

7.4 Virtualisation and Translation

Nested page tables [9], where a two-dimensional translation, first from guest virtual to guest physical address, then from guest physical to host physical, is performed, are the standard technique used in virtualised setups today. This has a bad worst case of 25 memory accesses, but is often very cacheable, and the structures are easy to update. Shadow pages [43] instead directly transform between guest virtual and host physical addresses, at the expense of increasing the work of the hypervisor, and are thus slower in practice [2]. Agile Paging [18] combines shadow and nested page tables.

Techniques exist to reduce the occurence of the worst case in nested page-table design: BabelFish [37] shares translations between multiple containers. Thermostat [3] handles the complexity of 2 MiB and 4 KiB page support for memory systems with two-tier properties, for virtualised systems. Prefetched Address Translation [29] (ASAP) utilises layout configurations in the operating system to approximate direct translation, and thus speculatively break linked-list chains in two-dimensional radix tables. Alverti et al. [6] rearrange virtual mappings to better provide locality, for prediction.

DVMT [5] allows the application to manage its own translation. Efficient Memory Virtualization [17] applies Direct Segments [8] to virtualised systems. Flat tables [4] observes that the intermediate layer can be flattened, reducing to a worst case of eight. In many ways, the translation of gPAs to hPAs in Compendia uses the same property, but instead structures the data in standard 2 MiB frames rather than table-specific allocation regions, and allows use of the same structures as in non-virtualised systems. Compendia also allows the use of sparsity in mappings where it is desirable.

8 Conclusion

Compendia is an extension to the standard four-level radix-tree design to utilize regions of density within the virtual-to-physical mapping to reduce translation costs, by merging layers into standard 2 MiB huge pages.

The small number of changes relative to a standard radix tree are trivial to add into existing designs, integrate well with other extensions, such as transparent huge pages, allow the use of conventional radix trees when translations are sparse, and have a negligible impact on memory utilization. In turn, they allow moderate speedups (5.5% geomean, 42% maximum) in conventional setups and larger speedups (18% geomean, 70% maximum) for virtualised systems. With the same caching techniques used for current radix-tree designs, translations can typically be performed with just a single memory access. Compendia is a fast and deployable solution, without pessimistic cases relative to current techniques, that will mitigate bottlenecks as application datasets scale.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant reference EP/P020011/1. Additional data related to this publication is available at https://doi.org/10.17863/CAM.68700.

References

[1] Linux source code: pgtable_types.h. https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/pgtable_types.h, 2020.

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS*, 2006. doi:10.1145/1168917.1168860.

[3] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ASPLOS*, 2017. doi:10.1145/3037697.3037706.

[4] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting hardware-assisted page walks for virtualized systems. In *ISCA*, 2012. doi:10.1145/2366231.2337214.

[5] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-it-yourself virtual memory translation. In *ISCA*, 2017. doi:10.1145/3140659.3080209.

[6] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris. Enhancing and exploiting contiguity for fast memory virtualization. In *ISCA*, 2020. doi:10.1109/ISCA45697.2020.00050.

[7] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *ISCA*, 2010. doi:10.1145/1816038.1815970.

[8] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *ISCA*, 2013. doi:10.1145/2508148.2485943.

[9] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS*, 2008. doi:10.1145/1346281.1346286.

[10] A. Bhattacharjee, D. Lustig, and M. Martonosi. *Architectural and Operating System Support for Virtual Memory*. 2017. doi:10.2200/S00795ED1V01Y201708CAC042.

[11] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *MICRO*, 2013. doi:10.1145/2540708.2540741.

[12] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *PACT*, 2009. doi:10.1109/PACT.2009.26.

[13] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *ICPE*, 2018. doi:10.1145/3185768.3185771.

[14] Jonathan Corbet. Four-level page tables. https://lwn.net/Articles/106177/, 2004.

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

[16] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. In *ASPLOS*, 2017. doi:10.1145/3037697.3037704.

[17] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *MICRO*, 2014. doi:10.1109/MICRO.2014.37.

[18] J. Gandhi, M. D. Hill, and M. M. Swift. Agile paging: Exceeding the best of nested and shadow paging. In *ISCA*, 2016. doi:10.1145/3007787.3001212.

[19] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. BadgerTrap: A tool to instrument x86-64 TLB misses. *SIGARCH Comput. Archit. News*, 42(2):20âĂŞ23, September 2014. doi:10.1145/2669594.2669599.

[20] Faruk Guvenilir and Yale N. Patt. Tailored page sizes. In *ISCA*, 2020. doi:10.1109/ISCA45697.2020.00078.

[21] Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *ISCA*, 1993. doi:10.1145/173682.165128.

[22] 7-Zip Forums Igor Pavlov. Windows 10 works incorrectly with large memory pages (2 MB). https://sourceforge.net/p/sevenzip/discussion/45797/thread/e730c709/, 2018.

[23] Advanced Micro Devices Inc. AMD-V nested paging. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf, 2008.

[24] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift. Performance analysis of the memory management unit under scale-out workloads. In *IISWC*, 2014. doi:10.1109/IISWC.2014.6983034.

[25] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, AdriÃąn Cristal, Mark D. Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *ISCA*, 2015. doi:10.1145/2872887.2749471.

[26] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Ingens: Huge page support for the OS and hypervisor. In *OSDI*, 2016. doi:10.1145/3139645.3139659.

[27] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *SC*, 2006. doi:10.1145/1188455.1188677.

[28] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. CSALT: Context switch aware large TLB. In *MICRO*, 2017. doi:10.1145/3123939.3124549.

[29] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *MICRO*, 2019. doi:10.1145/3352460.3358294.

[30] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the Graph 500. In *Cray User's Group (CUG)*, 2010.

[31] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. GraphBIG: Understanding graph computing in the context of industrial solutions. In *SC*, 2015. doi:10.1145/2807591.2807626.

[32] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *OSDI*, 2002. doi:10.1145/844128.844138.

[33] Ashish Panwar, Sorav Bansal, and K. Gopinath. HawkEye: Efficient fine-grained OS support for huge pages. In *ASPLOS*, 2019. doi:10.1145/3297858.3304064.

[34] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. In *ASPLOS*, 2018. doi:10.1145/3173162.3173203.

[35] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations. In *ISCA*, 2017. doi:10.1145/3079856.3080217.

[36] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB. In *ISCA*, 2017. doi:10.1145/3079856.3080210.

[37] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. BabelFish: Fusing address translations for containers. In *ISCA*, 2020. doi:10.1109/ISCA45697.2020.00049.

[38] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic Cuckoo Page Tables: Rethinking virtual memory translation for parallelism. In *ASPLOS*, 2020. doi:10.1145/3373376.3378493.

[39] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. CARAT: A case for virtual memory through compiler- and runtime-based address translation. In *PLDI*, 2020. doi:10.1145/3385412.3385987.

[40] M. Talluri, M. D. Hill, and Y. A. Khalidi. A new page table for 64-bit address spaces. In *SOSP*, 1995. doi:10.1145/224057.224071.

[41] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench – the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR*, 2014. doi:10.11484/jaea-conf-2014-003.

[42] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. Revanc: A framework for reverse engineering hardware page table caches. In *EuroSec*, 2017. doi:10.1145/3065913.3065918.

[43] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI), December 2003. doi:

10.1145/844128.844146.

[44] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation Ranger: Operating system support for contiguity-aware TLBs. In *ISCA*, 2019. doi:10.1145/3307650.3322223.

[45] Idan Yaniv and Dan Tsafrir. Hash, don't cache (the page table). In *SIGMETRICS*, 2016. doi:10.1145/2964791.2901456.

[46] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. In *VLDB*, 2014. doi:10.14778/2735508.2735511.