Perforated Page: Supporting Fragmented Memory Allocation for Large Pages

Chang Hyun Park*, Sanghoon Cha†, Bokyeong Kim†, Youngjin Kwon†, David Black-Schaffer*, Jaehyuk Huh†
*Department of Information Technology, Uppsala University
{chang.hyun.park,david.black-schaffer}@it.uu.se
†School of Computing, KAIST
{shcha,bkkim,yjkwon,jhhuh}@casys.kaist.ac.kr

***Abstract*—The availability of large pages has dramatically improved the efficiency of address translation for applications that use large contiguous regions of memory. However, large pages can be difficult to allocate due to fragmented memory, non-movable pages, or the need to split a large page into regular pages when part of the large page is forced to have a different permission status from the rest of the page. Furthermore, they can also be expensive due to memory bloating caused by sparse accesses to application data. In this work, we enable the allocation of large 2MB pages even in the presence of fragmented physical memory via *perforated pages*. Perforated pages permit the OS to punch 4KB page-sized *holes* in the physical address range allocated to a large page and re-map them to other addresses as needed. This not only enables the system to benefit from large pages in the presence of fragmentation, but also allows for different permissions to exist within a large page, enhancing sharing flexibility. In addition, it allows unused parts of a large page to be used elsewhere, mitigating memory bloating. To minimize changes to the system, perforated pages reuse the 4KB-level page table entries to store the hole locations and translates holes into regular 4KB pages. For performance, the proposed technique caches the translations for hole pages in the TLBs and track holes via cached bitmaps in the L2 TLB.**

By enabling large pages in the presence of physical memory fragmentation, perforated pages increase the applicability and resulting benefits of large pages with only minor changes to the hardware and OS. In this work, we evaluate the effectiveness of perforated pages with timing simulations under diverse and realistic fragmentation scenarios. Our results show that even with fragmented memory, perforated pages accomplish 93.2% to 99.9% of the performance achievable by ideal memory allocation, and 2.0% to 11.5% better performance over the conventional system running with fragmented memory.

***Index Terms*—virtual memory, address translation, translation lookaside buffer, memory management, virtualization**

I. Introduction

With the dramatic increase of memory footprints in data-intensive applications, the efficiency of address translation has become a critical performance bottleneck. This has led to numerous proposals to increase the efficiency of the translation look-aside buffer (TLB). One of the most widely-adopted is to increase the page size (e.g., to 2MB or 1GB), thereby multiplying the translation coverage of each TLB entry (by 512 times for 2MB pages), but at the cost of additional operating system or application support.

However, while large pages reduce translation overheads by improving TLB reach, they lead to physical memory bloating if the application does not use the full contiguous region [29], and make content-based page sharing less effective [7], [29]. Furthermore, regions cannot be promoted to larger pages when there are immovable 4KB pages [35] or 4KB pages with different permissions in the physical region. Fundamentally, today's large pages impose a trade-off between translation efficiency and memory management flexibility. In this paper we mitigate this trade-off by providing the flexibility needed to overcome each of the aforementioned limitations.

To mitigate this trade-off, we propose the ability to *punch* 4KB *hole pages* out of large 2MB pages via *perforated pages*. This enables the use of large pages even when parts of the pages are not allocated (avoids bloating), not allocated contiguously in physical memory (handles fragmentation), or have different permissions (increases sharing). For example, Figure 1 shows a perforated page that is allocated to a physical memory region with an immovable page. To enable a large page in this situation (VA space 1), a hole is punched in the perforated page that overlaps the immovable physical page and the hole in the virtual page is mapped to a separate physical 4KB region. The example also shows the second process (VA space 2) sharing the perforated page, but with a sub-page that has a different permission status. Such a partial change of permission within a large page can occur when a new read/write permitted hole page created as a result of copy on write, while the perforated page remains write protected. This hole page is similarly mapped to a separate 4KB physical region to enable the remainder of the perforated page to be shared. Both of these situations would force today's architectures to split the large page into separate 4KB pages.

Our implementation provides translations for the non-hole portions of perforated pages with comparable performance and overhead to traditional large pages, and efficiently tracks and stores an arbitrary number of holes pages for each perforated page. By allowing the overlapped mapping of large and regular pages, perforated pages provides both the improved TLB performance of large pages and the flexible memory management of regular pages.

We propose two changes to existing address translation mechanism to support perforated pages. First, the page table entry (PTE) for a perforated page should contain both the address of 2MB physical memory chunk and the address of the next-level PTE page to track the hole pages. If a virtual

address does not belong to a hole page, the physical large page address in the PTE will be used to calculate the final translated address. For addresses that do map to hole pages, the next-level PTE will be accessed during page table walks to find the translations. Since the current PTE can only contain one of the two addresses, we implement shadow PTEs for perforated pages to store the additional address of the PTE page for the hole pages. This change enables the tracking of arbitrary hole pages for each perforated page.

The second change enables efficient identification of hole pages within a perforated page during translation by providing two levels of *hole bitmaps*, which can be cached in regular TLB entries (Figure 3). The first-level hole bitmap is stored in the unused bits of the perforated page PTE in the TLB. These bits are unused as perforated pages track 2MB regions, resulting in 9 unused address translation bits vs. a standard 4KB page. This first-level hole bitmap *filters* accesses to the second-level hole bitmap by tracking which coarse-grained regions of the perforated page do not have holes. The second-level hole bitmap indicates which of the 4KB sub-pages of the 2MB perforated page are holes and require access to the next-level PTE for translation. These bitmaps are stored in a reserved part of the physical memory and cached in the TLB along with the other PTEs on-demand, and accessed only if the first-level hole bitmap does not filter access to them.

Virtualized systems have even higher levels of fragmentation due to fragmentation in both the hypervisor and guest operating system [38]. To address this, we extend perforated pages to support 2-dimensional page walks and add interaction between hypervisor and guest hole bitmaps.

The contributions of this paper are:

- Overlapping large 2MB and regular 4KB pages to enable the use of large pages in the presence of fragmentation.
- An efficient implementation of perforated pages with shadow PTEs and on-demand, TLB-cached, hierarchical bitmaps for rapid translation and flexibility.
- Minimal HW and SW changes by taking advantage of existing PTE structures and walkers.
- Addressing both guest and host fragmentation in virtualized systems.

We evaluate perforated pages with an out-of-order execution simulator under diverse and realistic fragmentation scenarios.

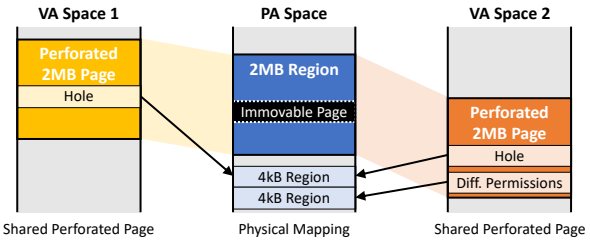

Fig. 1. Perforated pages support allocation of large pages even when the physical backing 2MB region is fragmented by immovable pages (center) and allows shared large pages to contain sub-pages with different permissions (right). Not shown: portions of perforated pages can be left unallocated to avoid physical memory bloat as well.

Our results demonstrate that perforated pages enable overlapped mappings with only minor performance degradations when the fragmentation ratio is low. Even when fragmentation is severe and randomly placed, perforated pages still provide better performance than the regular page sizes.

To evaluate perforated pages we start by reviewing large page support in current systems, its challenges, and previous approaches (Section II), and the impact of fragmentation on real systems (Section III). We then describe the architectural (Section IV) and virtualization support (Section V) and software changes required (Section VI), before evaluating the proposal via simulation (Section VII).

II. Background

For compatibility with the page table itself, the selection of available page sizes is tied to the page table organization. In x86, with a four-level page table tree, 4KB pages correspond to the coverage of a level-1 page table entry (PTE), while level-2 and level-3 PTEs correspond to 2MB and 1GB pages, respectively. This organization makes it easy to store large pages in the page table by ending the page walk at the appropriate level for the given page size[1].

As data sizes have increased, so too has the hardware TLB capacity, resulting in today's multi-level TLB structures to deliver low average latency with large effective capacity. In the latest x86 architectures, the level-1 TLB stores about 64 to 100 entries of PTEs for all page sizes, while the level-2 TLB stores between 1536 and 2048 entries of both 2MB and 4KB pages in a shared array.

A. OS Large Page Support

Operating system support is required for applications to make effective use of large pages. The most basic interface is to allow applications to explicitly request huge pages when allocating memory. In Linux, calling `mmap` with `MAP_HUGETLB` flag for allocations that are aligned to the large (huge) page size will result in the desired large page allocation. However, this low-level interface requires changes to the application memory allocation code.

To provide a more friendly interface, Linux provides *transparent* huge page (THP) support. THP provides large pages to user processes automatically: whenever there is an opportunity to allocate a large page for a user process, e.g., the requested allocation is large enough and aligned, and there is an available contiguous mapping, the kernel will allocate large pages *transparently* to the process. Whenever the user process decides to modify (i.e. change permissions, free pages, etc.) any part of the transparent large page, the kernel will split the large page into regular page entries, and act on the page as requested. The kernel may also choose to transparently promote groups of regular pages to large pages to improve TLB performance. However, this requires either sufficient contiguous physical memory or explicit (and expensive) compaction/relocation of existing allocations to create such regions.

[1]In this work we consider "large" pages to be 2MB, or "huge" pages, however our approach could apply to a variety of page sizes.

B. Related Work

There have been numerous prior works addressing translation performance for large workloads. To improve TLB coverage, Pham et al. proposed coalescing multiple adjacent translations [40] and clustering, in both virtual and physical space, of small groups of nearby pages to translate any permutation of memory mappings within the groups [39]. Park et al. proposed supporting contiguous translations with the help of the OS [36] to adapt the degree of contiguity at runtime. Basu et al. revived segmentation to efficiently translate very large contiguous regions of memory [14]. Karakostas et al. added support for multiple segments for flexibility [27] and Park et al. introduced a many-segment translation system that delays translations for cache misses [37]. These approaches require large contiguous physical memory blocks, which makes fragmentation more of a challenge.

Yan et al. proposed OS changes to efficiently generate large contiguous regions, but they were hampered by unmovable pages [46]. The recent studies on improving the Linux transparent huge page system by Kwon et al. [29] and Panwar et al. [35] have targeted latency, performance variability, and memory consumption by minimizing wasted efforts in trying to compact page blocks with non-movable pages.

Multiple works have reduced TLB misses and latency by improving the page table walker [12], [13], [16] and employing prefetching [17], [31], [32]. Bhargava et al. and Ahn et al. investigated the translation challenges for virtualized systems [9], [10], [15]. Pham et al. applied speculation to glue together regions of large pages that have been splintered by the hypervisor [38]. Alverti et al. used contiguous allocations and a page table walk predicting mechanism along with speculation to minimize page walk latencies [11]. The above methods could be applied to our approach to reduce latencies as well.

Swanson et al. proposed providing support for large pages with non-contiguous physical backing. They did so by adding an intermediate address layer to enable large pages in the core-cache side and non-contiguous small physical pages via a memory controller TLB [44]. Their approach does not benefit workloads that do not fit into the cache [48]. Du et al. proposed supporting large pages with non-contiguous physical memory [19] (Section VII).

Bitmaps have been widely used for fine-grained memory management. Prior work on DRAM caches [26] and heterogeneous memory [41] systems use bitmaps for fine-grained movement/storage. Bitmaps have been used in the TLB to mark accessed-dirty subblocks [42] or the validity of a coalesced [40] or clustered mapping [38]. Seshadri et al. proposed bitmaps to track cacheline overlaying pages for fine-grained use of virtual memory features. We use bitmaps to mark parts of the perforated page that are hole pages.

III. Motivation

A. Large Page Management Challenges

Memory bloating: When applications do not use all of the physical memory allocated to their large pages, they waste physical memory, leading to memory bloating. This problem occurs because the operating system's transparent huge page support (THP) is not aware of the access patterns of the applications and tries to allocate large pages when possible (e.g., from virtual address regions that are multiples of 2MB, 2MB-aligned, and have identical permissions). If the application only accesses the data sparsely, this results in the whole 2MB of data being allocated in physical memory while only a small amount of it is actually used, thereby increasing real memory consumption compared to using only 4KB pages.

Memory bloating is one of the reasons that large page support is recommended to be disabled in common applications such as Redis, MongoDB, Splunk, and VoltDB [3], [4], [6], [8], [29]. Perforated pages can address this by punching holes in the large pages for the regions of regular pages that are not used by the application. The OS can then use the regular physical pages that correspond to those holes for other pages. By allowing holes, perforated pages provide efficient memory utilization for sparsely used regions and performance benefits close to those of large pages.

Trade-off between saving memory and performance: VMware's transparent page sharing [43] and the Linux kernel's same page merging (KSM) [1] are common services in virtualized environments to save memory by deduplication. The services detect pages with the same content and create shared mappings to the identical pages. These memory-saving services have a clear trade-off with large pages: large pages increase translation efficiency, but reduce the opportunities for sharing by 4-10x because the chance of having identical content decreases with the size of the region [38]. Previous work has addressed this trade-off by breaking cold large pages into regular 4KB pages to increase the chance of finding identical pages [21], [29]. To evaluate these effects, we conducted an experiment using KSM on two Linux virtual machines (VM) executing the *mcf* benchmark. The two VMs consumed 5.16GB in total, but KSM reduced this by 11%. We observed that page sharing broke 40% of the VMs' large pages into regular pages and shared almost all of them. Due to this 40% reduction in large pages, the VM suffered a 34% increase in TLB misses that translated into 4.9% performance loss. While page sharing trades performance for memory saving, our perforated pages do not, as they allow holes in the large pages to accommodate non-identical sub-pages in large pages.

Compaction overheads and immovable pages: Promoting pages to a large page requires a contiguous physical memory region of the large page size. However, the longer the system has been running, the more the memory becomes fragmented. The Linux kernel actively compacts physical memory to create free contiguous memory when it runs out of free contiguous memory. This memory compaction is expensive: it requires scanning pages, copying physical pages to new locations, updating PTEs, and TLB shootdowns. Linux performs these compactions in the page fault handler, which increases fault latency [29], [34], or in a background thread, which consumes memory bandwidth and a CPU core. Unfortunately not all pages can be compacted. Pages used for device drivers,

TABLE I
REDIS PERFORMANCE AS A FUNCTION OF PAGE SIZE.

Factor	Large pages	Regular pages
Memory allocation (GB)	92.9	77.7
TLB misses (MPKI)	1.00	1.83
Normalized Performance (requests/s)	1.29	1.00

file caches, and other operating services can be marked as *immovable*, preventing compaction [35].

Perforated pages enable cheaper page promotion by allowing the OS to either move existing pages to generate a contiguous region (if the pages are movable) or punch holes in the perforated page (if not). Further, more regions can be promoted by punching holes for immovable sub-pages. This flexibility reduces the OS compaction burden as creating holes in the large page may be far cheaper than the overhead of compaction.

B. Case Study: Memory Bloating

To investigate the trade-off between application memory bloat and address translation performance, we compared the effects of large pages and regular pages on the Redis in-memory database. We configured Redis with 4 million keys and 16KB objects, as we observed significant memory bloating for this configuration when running with transparent huge pages enabled. Table I shows the allocated memory size, TLB misses per kilo instructions (MPKI), and normalized performance using large (2MB) and regular (4KB) pages.

The table shows that with large pages, memory allocation bloats by 20%. To evaluate the effect on performance, one million requests to random keys are issued to the server to measure TLB misses and execution times. With large pages, the TLB MPKI is 82% lower than with regular pages, showing the benefit of the increased translation coverage. This results in a performance improvement of 29% (requests per second).

To understand the potential of perforated pages, we use a trace-based simulation to compare TLB misses for regular and large pages vs. perforated pages. (Timing performance is evaluated in Section 6 with the gem5 out-of-order execution model.) As with the earlier hardware-based experiments, we allocated 4 million keys for 16KB entry size in Redis, and then obtained the memory address trace (via Pin [30]) from 1 million random key accesses. We fed the access trace into a TLB simulator (1.5k-entry, 12-way) with perforated page support.

The simulations reports TLB MPKIs of 1.75, 1.05 and 1.45 for the regular 4KB, 2MB, and perforated pages, respectively. As the Redis experiment has random access patterns with large memory working sets, even large pages cannot drastically reduce TLB misses. However, even for such harsh patterns, the proposed technique can reduce the TLB misses significantly, achieving roughly half of TLB miss reductions from pure large pages, but without the negative bloating effect inherent in large pages.

C. Case study: Real-world Fragmentation

To understand real-world fragmentation challenges, we examined the memory state of a machine running Linux 5.3.8 with 12GB of physical memory. Our benchmark is compiling the Linux kernel, which allocates memory both in user space (compiler and build tools) and kernel (kernel metadata and file caches, known as the *page cache*). For performance, the OS keeps the page cache until it runs out of free memory. This is particularly relevant because the *kernel page cache allocations create non-movable pages*, thereby making memory compaction and large page allocation difficult.

To evaluate the severity of this effect, kernel compilation ran until the free memory had been depleted and then started a benchmark that attempted to allocate 2GB of memory with large pages. When the benchmark first requested its allocation, it received only 2.5% large pages (e.g., 25 2MB pages) due to the system memory fragmentation. We then let the benchmark idle for for over 100 seconds to give the kernel a chance to compact memory in the background. At this point, the fragmentation was reduced, and 2MB large pages were used for 22% of the application memory.

To investigate the potential of perforated pages, we simulated a policy for creating perforated pages based on our performance evaluation in Section VII-B. When $<$25% of a 2MB region is non-movable and at most 4 of 8 filter bits are set (not too fragmented), we allocate the 2MB region as a perforated page. Note that in conventional systems these 2MB regions cannot be allocated in large pages and must be allocated using regular pages. To evaluate this scenario, we measured how many perforated pages the OS could deliver. By applying this perforated page creation policy, we found that *an additional 18% of the 2GB could be allocated using 2MB perforated pages*.

We then used the system for other tasks, such as post processing results for the study, which added memory pressure leading to more page caches being freed, and then re-ran the benchmark. Unlike the first run, this time we found that the OS could deliver 49% 2MB large pages and the potential to allocate 50% perforated pages. The improvement in the ability to allocate large and perforated pages is due to the reduction of immovable pages. These measurements demonstrate that fragmentation can be both common and severe under typical workloads.

To assess the performance implication of perforated pages in this scenario, we simulated the performance improvements for a random access benchmark (Section VII-B). This indicates a 20.3% performance improvement by moving from a system with only 22% large pages to a system with 22% large pages and 18% perforated pages. For the example of the second execution, when half of the entire memory is allocated as large pages, and the other half as perforated pages, the performance improvement is 69.4% over the conventional system, which only has half the memory in large pages and the remaining half in regular pages.

IV. Architecture

A. Overview

In this proposal, large pages can be represented either as standard 2MB pages (contiguous physical region) or as more flexible 2MB perforated pages (non-contiguous physical regions). Note that we follow the x86 architecture and use 2MB large pages and 4KB regular pages, but the general idea can be applied to different combinations of page sizes.

Figure 1 (VA space 1) shows a perforated page in a virtual address space with a hole mapped to a physical page that is not contiguous with the rest of the perforated page (center). Furthermore, the perforated page is shared with the second virtual address space (VA space 2), but that address space has a different permission for one page in the shared allocation. For the 4KB region with a different permission, the hole in its perforated page (VA space 2) points to another 4KB physical page with the appropriate permission. This illustrates a copy-on-write (CoW) situation where the first and second address spaces shared the same perforated page, but the second address space wrote to one page, causing a CoW and a new mapping with a different permission. With perforated pages, the OS does not need to copy the whole large page nor split it into smaller pages.

Perforated pages require changes 1) to identify hole pages in a perforated page, and 2) to provide the translation for hole pages. First, to identify hole pages, we use *hole bitmaps* to track which sub-pages are holes. For efficient accesses to hole bitmaps, we re-purpose L2 TLB entries to cache hole bitmaps on-demand, and use a coarse-grained bitmap *filter* to improve storage efficiency and latency. Second, to translate hole pages, we build upon the standard multi-level page table structure but add *shadow L2 page table entries* for perforated pages. The main L2 PTE of a perforated page contains the physical address of the 2MB region, while the shadow L2 PTE points to the next L1 page table node containing the translation for any 4KB hole pages. In addition, to avoid changes to the latency-sensitive L1 TLB, we handle perforated pages in the L2 TLB, where accesses are translated into 4KB regular pages (either hole or not) and then stored in the L1 TLB for future use.

B. Hole Page Tracking

Perforated pages use *hole bitmaps* to efficiently track where holes exist in the perforated page and *shadow L2 PTEs* to access the translations for the 4KB hole pages.

Shadow L2 PTEs for accessing hole translations: For perforated pages, the L2 PTE must hold two addresses: the physical address of the 2MB region for the non-hole portions and the physical address of the L2 page table node for any hole translations. As current PTEs can contain only one address, we employ a *shadow L2 page table* to store the second.

Figure 2 shows a page table with a shadow L2 node. The shadowed L2 page table nodes are placed adjacent to the corresponding L2 page table nodes. This simplifies the address calculation for the shadowed node since adding a 4KB offset to the original node is the address of its shadowed node, thereby avoiding additional indirections and is easy to allocate thanks to the kernel's buddy allocator.

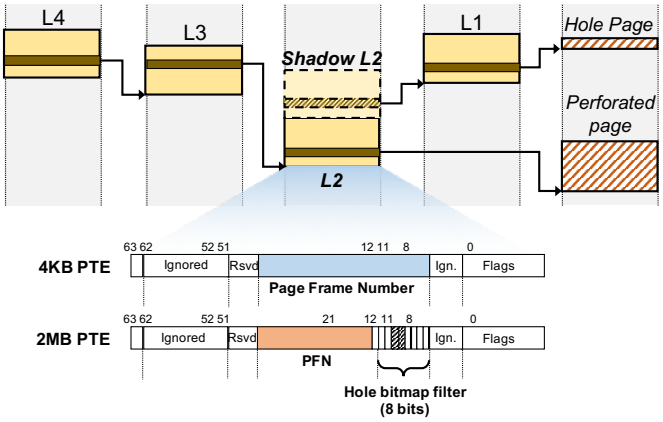

Fig. 2. Page table with perforated page support. The shadow L2 page table is allocated adjacent to the real L2 page table using the buddy allocator. The extra bits of 2MB PTE are used to hold the bitmap filter.

For a perforated page, the original L2 PTE contains the address of the 2MB data page, and the shadow L2 PTE contains the address of the next-level L1 page table node for accessing hole translations. Note that the shadow L2 PTEs are not stored in the TLB. Instead, they are accessed during the page walk to translate 4KB hole pages and the found 4KB hole PTE is installed in the TLB. However, to accelerate page table walks, the content of the shadow L2 can be stored in the Page Walk Cache and regular cache hierarchy. The level-2 TLB holds regular 4KB and 2MB pages, as well as perforated 2MB pages and hole 4KB pages.

Hole page bitmap for identifying holes: We use a 512-bit bitmap (one bit per 4KB sub-page) to track holes in the perforated page and indicate if an access to the shadow PTE is required. A naive implementation of the bitmap storage is to extend each L2 PTE by 512 bits to store the status bits along with the PTE entry. However, such an approach would require an eight-fold increase in TLB capacity and L2 page table node size.

Instead, we decouple the hole bitmap from the PTE, and use a separate block of contiguously allocated physical memory to store the hole bitmaps for the whole system physical address space. As every 4KB page of the system has a bit in the hole bitmap, the location of the relevant bitmap can be easily calculated from base location of the bitmap storage and the physical location of the page. This requires 0.003% of the physical memory, or 4MB for a 128GB system.

Figure 4 shows that translations for perforated pages must now first check the bitmap to see if the 4KB sub-page corresponds to a hole. If it does, then the shadow L2 PTE is accessed and a L1 page walk returns the 4KB hole page translation. If it is not a hole, the translation can be directly generated using the 2MB large page physical address of the perforated page. To make this more efficient, we maintain a second-level, coarse-grained hole bitmap that serves as a *filter* to reduce the need to access the hole bitmap if the coarse-grained hole bitmap region has no holes. (See Figure 2,

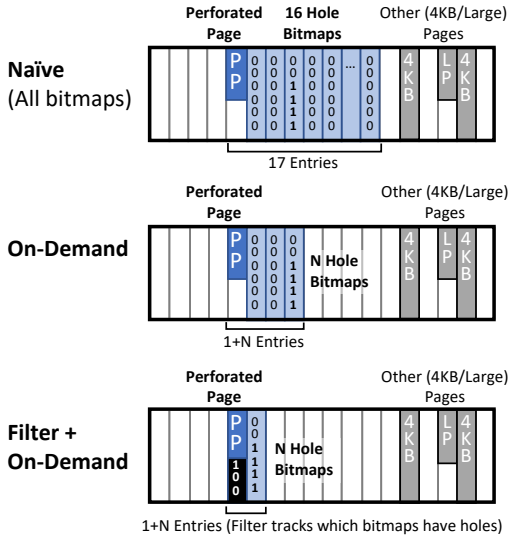

Fig. 3. Generating 4KB pages from large pages. (top) A **Naïve** approach requires 16 L2 TLB entries for each perforated page (PP) to store the hole bitmaps. On detection of a hole from the bitmap, a page walk is initiated to load a 4KB page translation for the hole. (middle) An **on-demand** approach only loads hole bitmaps as needed. Unused bitmap space can then be used by other L2 TLB entries. (bottom) **Filtering** uses bits in the perforated page TLB entry (black) to identify hole bitmaps that do not contain holes, and thereby avoid the need to store them in the L2 TLB.

bottom.) As the PTE for the 2MB perforated page requires 9 fewer address translation bits than a 4KB page, we have 9 unused bits available in its PTE, of which we repurpose 8 for our hole bitmap filter. Each of the 8 bits in the filter indicates whether there are holes in the corresponding coarse-grained region of the full hole bitmap (e.g., 256KB region of the 2MB page). The filter is placed in the L2 PTE and cached in the perforated page TLB entry. With this approach we can avoid accessing the bitmap for any coarse-grained region in the perforated page that has no holes.

C. L2 TLB Extension

Our design leaves the latency-sensitive L1 TLB untouched. To accomplish this, we *generate* 4KB pages for sub-pages of the perforated page that are not holes directly from the L2 TLB and install them in the L1 TLB as any other page. The L2 TLB, however, needs to be modified for accessing perforated pages and hole bitmaps.

To make accesses to hole bitmaps faster, we cache them in L2 TLB entries. This is possible as modern machines support physical address widths of well over 40 bits (46 and 48 on our Intel E5-2630 and AMD FX-8105 systems), leading to at least 28 bits of physical address, plus several bits for permissions. We assume that 32 bits of hole bitmap can be stored in a TLB entry, thereby requiring 16 L2 TLB entries to cache a full perforated page bitmap.

A naïve approach is to *load the full bitmap* into the L2 TLB when a perforated page is brought in. (Figure 3, top.) As all bitmap entries for the perforated page will be in the same 64B cache line, a single memory access is sufficient. This will require 17 L2 TLB entries for each perforated page, not

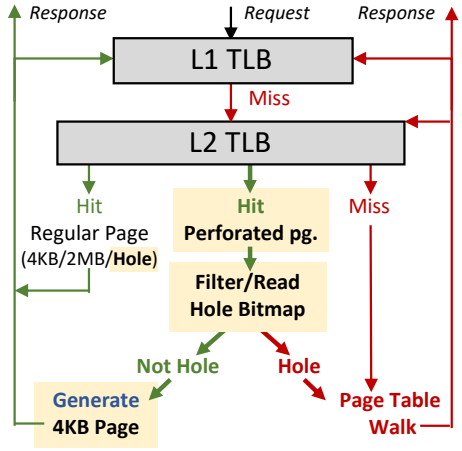

Fig. 4. Translation Flow. Translation requests that miss in the L1 TLB and hit on a perforated page in the L2 TLB are serviced by generating a 4KB page that is then installed in the L1 TLB.

Filter?	Bitmap Present?	Is Hole?	Actions	Added Cost
Not Hole	skip	n/a	Generate 4kB Page	-
Possibly Hole	Present	No	Generate 4kB Page	Bitmap (L2 TLB)
		Yes	Page Table Walk	Bitmap (L2 TLB) Page Table Walk
	Not Present	No	Load Bitmap Generate 4kB Page	Bitmap (Mem)
		Yes	Load Bitmap Page Table Walk	Bitmap (Mem) Page Table Walk

Fig. 5. Handling holes. If the filter determines it is not a hole then there is no additional cost. If the access is not filtered, and the bitmap is not present in the L2 TLB, it is fetched from the memory. If the access is to a hole, the hole page table entry is fetched via page table walk (red); otherwise the perforated page generates a 4KB TLB entry to be used in the L1 TLB.

including any eventual 4KB hole translation entries. However, this approach may bring in the bitmaps of unused regions of the perforated page, thereby wasting L2 TLB capacity that could be used for other translations. A more efficient approach is to *load the bitmaps on-demand* (Figure 3, middle). This provides more TLB capacity for other translations as long as the full perforated page is not accessed. Our proposed design (Figure 3, bottom) combines a coarse-grained filter bitmap in the perforated page TLB entry with on-demand loading of the bitmap data. The design avoids wasting L2 TLB capacity for bitmaps with no holes (by filtering) and stores bitmaps in the L2 TLB only for the regions that are actually accessed (on-demand loading).

D. Address Translation Flow

Perforated large page support requires changes to the L2 TLB lookup and page table walk logic. Figure 4 shows our proposed address translation flow. On an L2 TLB access, the corresponding 2MB or 4KB entry is located in the standard manner. If the translation hits on a regular 4KB or 2MB entry, or an already-translated 4KB hole entry, the translation is sent back to the core and inserted into the L1 TLB (left path). If the lookup hits on a perforated page, the TLB first checks the filter bitmap in the TLB entry. If the filter indicates the translation is in a coarse-grained region of the perforated page that does

not have any holes, a 4KB translation is generated from the 2MB perforated page and inserted into the L1 TLB to handle future requests, with no additional overhead (Figure 5, row 1). However, if the filter indicates the region *might* have holes, the hole bitmap needs to be consulted, either via a second L2 TLB access (if it is already cached in the L2 TLB, Figure 5 rows 2 and 3) or a memory access (if not, Figure 5 rows 4 and 5). If the final access to the bitmap shows that the translation is on a hole, a page table walk is initiated via the shadow PTE entry and the 4KB translation for the hole is installed in both the L2 and L1 TLBs.

E. Changes and Overhead Analysis

Storage: The overall page table structure is not changed, but perforated pages require a shadow L2 PTE (4KB of additional page table data per perforated page) plus storage for the bitmaps (0.003% of physical memory).

HW: The HW change is mostly limited to the L2 TLB controller (storing, searching, and fetching of hole bitmaps) and the page table walker logic (checking perforated pages and generating 4KB regular pages for hits).

Potential performance overheads: 1) L2 TLB translation latency can be increased due to the secondary L2 TLB access for the bitmap and the memory access if the bitmap is not present (Figure 5, rightmost column). 2) TLB capacity is consumed by bitmap entries, requiring up to 16 additional TLB entries per perforated page if the filter is ineffective and the whole range is accessed. Note that the increased coverage of a perforated page (512×) far exceeds this maximum increased TLB usage (17×)[2].

TLB shootdown: A perforated page shootdown requires invalidating translations for the perforated page itself, all of its bitmap entries in the L2 TLB, all of its hole entries in the L1+L2 TLBs, and all of its generated non-hole entries in the L1 TLB. This could be up to 512+17 total entries, so instead of searching for them individually, we take the approach of the Linux kernel [28] and flush the TLB[3].

Shootdowns for non-hole and hole pages that are part of perforated pages are the result of either *punching new holes* or *patching existing holes*. This occurs due to copy-on-writes(CoW) or compaction, which already require shootdowns. Punching and patching must also update the perforated page bitmap, which requires invalidating the bitmap entry in the L2 TLB and updating the bitmap in system memory. Bitmap changes will be batched with the required existing shootdowns, making them negligible[4]. On subsequent translations the new bitmap will be fetched, although it could be updated in-place in the TLB with the right ISA support.

[2]In the worst case a perforated page requires 1 entry for the page, 16 entries for the bitmaps, and 512 4kB hole page entries, if every sub-page is a hole. In the baseline system, mapping the same data would require 512 4KB entries. So as long as the perforated page has more than 17 entries that are not holes, it is a more efficient use of L2 TLB capacity.

[3]Modern systems allow per-process TLB flushing to reduce this cost for large L2 TLBs

[4]TLB shootdowns are known to be expensive, but it is the synchronization of all participating HW cores that results in the μs-scale latency. The invalidation call to the TLB itself is about 158ns [28].

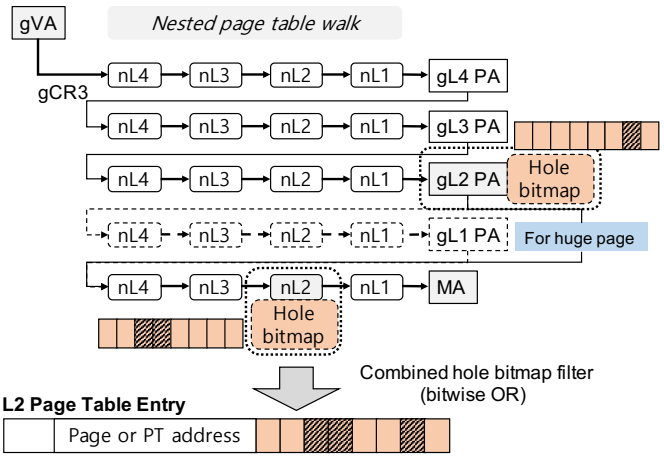

Fig. 6. 2D translation in virtualized systems. The hole bitmap filter of the guest and host page tables are combined (bitwise-or) by the page table walker.

In addition to updating the bitmap, patching holes requires a regular shootdown of the hole page in the L1 and L2 TLBs. On subsequent translations the perforated page will simply generate a translation for the new non-hole. Similarly, punching a hole requires both updating the bitmap and a regular shootdown of the non-hole page, which may be in the L1 TLB. On subsequent translations, a fault will be generated when trying to access the shadow PTE for the new hole, and the hole 4KB entry will be brought in as normal. Note that both regular shootdowns for CoW (punching) and compaction (patching) are already issued in current systems.

V. Virtualization

System virtualization has been widely adopted to improve system utilization by consolidating multiple virtual machines on a physical system. Virtualization first requires that a guest virtual address (gVA) for a process within a virtual machine (VM) to be translated to a guest physical address (gPA), which is in the virtual physical address space of the VM. The guest physical address is then translated to a host machine address (MA) by the second address translation. HW support for these 2D nested page walks is essential for performance in virtualized environments.

In virtualized systems, the TLBs store either the final translation from gVA to MA, or intermediate translations (gVA to gPA and gPA to MA). If large pages cannot be mapped in either the gVA-gPA or gPA-MA mappings, the translation must be stored in intermediate form. For example, a guest large page backed by regular host machine pages cannot use a large page mapping from gVA to MA. Instead it has one large mapping from gVA to gPA, and 512 regular mappings from gPA to MA [38].

To support perforated pages in virtualized systems we must set the hole page status bit properly considering both of the guest and host page tables. Figure 6 details the translation path. If a mapping uses a perforated page in both of the guest and host page tables, its translation can be done by a perforated page entry in the TLB. With perforated pages, the guest can use holes to avoid unmovable guest physical

addresses, however, the host may have different sets of holes, resulting in two independent sets of holes in the two layers of mappings. In an effort to install the perforated page entry for gVA to MA mappings, the hole bitmap cached in the L2 TLB must be aware of the holes made by both guest and host mappings. Therefore, when the page table walker traverses both page tables and bitmaps of both mappings, the hole bitmaps of both mappings are combined by a bitwise-OR operation, and then inserted into the TLB.

The location of the host hole bitmap is directly accessible in the host machine address. However, the location of the guest hole bitmap is set in the guest physical address space. Accessing the guest hole bitmap requires an additional gPA to MA translation. As this translation is frequently accessed and is rarely moved, the host can cache the MA address of the guest host bitmap in a guest control register.

VI. Operating System Interaction

Choosing perforated pages: The operating system must choose when to allocate perforated pages vs. multiple regular 4KB pages. The trade-off depends on the application's access patterns and the fragmentation status of the memory: the more an application accesses portions of perforated pages that do not have holes, the better it will perform, and the more pages are fragmented in compact chunks, the more effective the bitmap filter will be. These tradeoffs are explored in Section VII-B.

Page allocation: When the OS allocates L2 page table nodes it must now allocate a second contiguous page for the shadow L2 page table node. As page table allocations are also handled by the buddy allocator, the allocation path can be easily modified to allocate two contiguous pages. The OS also then allocates the L1 page table nodes in the same manner as existing page table allocations.

When a perforated page is allocated and its mapping inserted into the L2 PTE, the OS must mark the holes in the bitmap. However, it can choose whether the hole pages are allocated at this point or lazily on subsequent faults. This gives the OS the flexibility to lazily allocate physical space for holes. Because the holes are marked in the bitmap when the perforated page is first allocated, no shootdowns are required for allocating the hole pages as no bitmap entries need to be updated.

The lazy allocation of hole pages can potentially support a reservation-based large page allocation [29], [33] with perforated pages and unallocated holes. The perforated page could reserve a whole large page, but actually map (and patch the holes) when the regular pages are actually touched, preventing unused page mappings from causing memory bloating. In this case, the bitmaps need to be updated for every patching operation, requiring TLB shootdowns, thereby trading off shootdowns for better memory utilization.

Modifying the page mapping: If an application frees a regular page that is part of a large page, or the OS remaps a regular page of the large page to another region (e.g. copy-on-write), the OS can either splinter the large page into regular pages or punch holes (mark hole bitmap) in the large page to transition it into a perforated page. Holes can be punched in perforated pages as well. When adding holes to large pages or perforated pages, the OS must trigger a TLB shootdown to the affected TLB entries and/or the bitmap entries. (Section IV-E)

Perforated pages can also be patched, to remove holes, and even reconstruct a non-perforated large page. This can be a result of memory compaction. When unusable memory that was not previously compactable is freed, the OS can compact it into the perforated page to remove the holes. As with punching, patching also needs an appropriate TLB shootdown.

TABLE II
Simulation configurations

Component	Configuration
Processor	2GHz Out-of-order x86 Processor
L1 I/D Cache	4-cycle, 32KB, 8-way, 64B block
L2 Cache	20-cycle, 2MB, 8-way, 64B block
Memory	DDR4-2400, 4 Channel
L1 TLB	1-cycle, 64-entry, 4-way, 4KB 1-cycle, 32-entry, 4-way, 2MB
L2 TLB	9-cycle, 1,536-entry, 12-way Shared by 4KB and 2MB entries
Page Struct. Cache	4-entry L3 cache entries 24-entry L2 cache entries

VII. Evaluation

A. Simulation Methodology

To evaluate perforated pages, we implemented the HW and OS support in the Gem5 simulator under system-call emulation (SE) mode [18]. (As we simulated the computational region of each application, the memory mappings were setup prior to the simulated region and we observed no significant changes in memory mapping from the applications during simulation.)

The core, cache and TLB parameters are listed in Table II. As SE mode for x86 does not model TLB miss latencies, we added the multi-level TLB, page walker cache, and all delays associated with TLB misses. The two-level TLBs are organized similarly to the Intel Skylake microarchitecture [2], [24]. The L1 TLB contains 32 entries for 2MB huge pages, and 64 entries for 4KB pages. The L2 TLB is an 12-way TLB with 1.5K entries, and each can hold 2MB, 4KB, perforated page, or bitmap entries. As observed by others, we assume that the 4KB and 2MB page entries can be looked up concurrently in the L2 TLB [2]. We implemented a page walk translation cache based on the Intel Page Structure Cache (PSC) [23], [25], [45]. Finally, as the TLB is per core and works independently on each core, we evaluate a single-core system.

We first study the performance impact of perforated pages on a microbenchmark that fits in the L2 TLB in the baseline with the best allocation scenario, but is capacity-pressured with less optimal memory allocations, to draw conclusions for a conservative system configuration. We then use mcf, omnetpp, libquantum, and zeusmp from SPEC CPU 2006 [22], mummer and tigr from the biobench suite [20], and SPEC CPU 2017 [5]

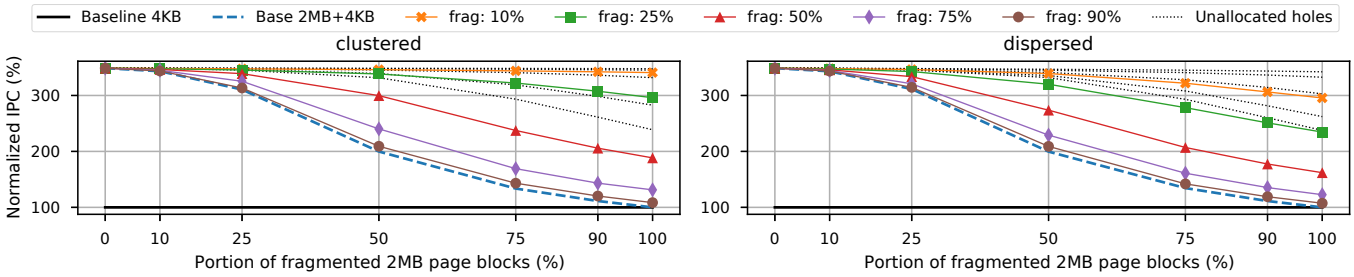

Fig. 7. Sensitivity to the *portion* (percentage of 2MB regions that are fragmented = percentage of perforated pages) and the *fragmentation* (percentage of holes or unallocated pages within each 2MB region, or perforated page). IPC values are normalized to the performance of the baseline 4KB TLB.

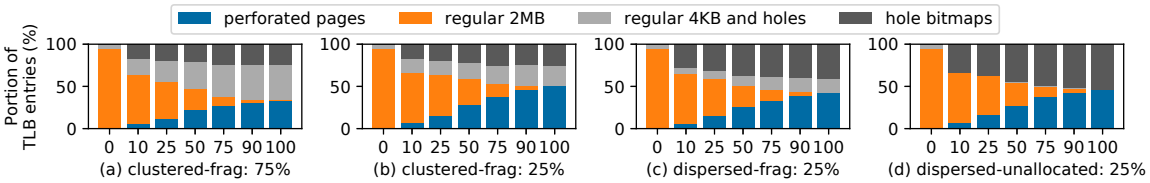

Fig. 8. TLB content breakdowns for various distribution and fragmentations. The x-axis shows the *portion* (percentage of 2MB regions that are fragmented = percentage of perforated pages).

(omnetpp17 and xz) to show the impact of perforated page on larger applications[5].

To explore a range of scenarios, we emulated a fragmented system using two distributions of hole pages: *dispersed* and *clustered*. Dispersed spreads the hole sub-pages throughout the fragmented 2MB page blocks, while clustered places them together. We expect the clustered distribution to perform better due to our bitmap filter. Secondly, to control the severity of fragmentation, we explore the *portion* of 2MB regions that are fragmented and the degree of *fragmentation* within each 2MB region.

B. Sensitivity to Fragmentation

We first explore performance with regards to system and application fragmentation. We assume a worst case memory access scenario for the TLB by running a random memory access benchmark with a 2GB dataset that fits in the TLB when using 2MB pages. As fragmentation increases, conventional large pages must be split into regular pages, reducing the effective reach of the TLB, and hurting performance.

Figure 7 shows the performance impact of fragmentation across the *portion* of fragmented 2MB regions (% with holes, x-axis) and the *fragmentation* inside each region (% of holes, curves), for both *clustered* (left) and *dispersed* (right) distributions, normalized to 4KB pages. The dashed curve shows the performance for conventional large page support (Base 2MB + 4KB).

Note that the real-world fragmentation scenario we observed (Section III-C) had 50% fragmented 2MB blocks with up to 25 holes clustered in each block. This corresponds to the curve *frag-25%* on the left (*clustered*) figure at the x-axis of 50% (*portion*). From that point we can see the potential performance benefit (59.4%) of perforated pages (*frag-25%*) over traditional large pages (*Base 2MB+4KB*).

[5]Other benchmarks from these suites were evaluated but as they were not TLB-sensitive, they showed little difference.

***Portion* of fragmentation:** As the portion of fragmented 2MB blocks increases (towards the right) performance decreases. This is expected for *Base 2MB + 4KB* as the conventional TLB is forced it to split large pages due to fragmentation, reducing TLB coverage and increasing the number of page walks.

Perforated pages show significantly less performance loss as fragmentation increases, maintaining nearly all of the benefit of large pages even when 50% of the total large pages have 10% holes. As expected, as the *portion* of pages that have holes increases (x-axis, moving right) the performance decreases as not all of the hole pages can fit in the TLB. In the extreme case of all pages containing holes and 90% fragmentation in each perforated page, the performance is roughly equivalent to the *baseline 2MB + 4KB* as essentially all pages are holes, and therefore require a 4KB page translation in the TLB. While perforated pages should benefit from the 10% that are not holes, the second L2 TLB access for the 90% hole pages largely negates this benefit. Figure 8 shows the breakdown of the L2 TLB entries for these scenarios. As the *portion* of fragmented 2MB blocks increases (left-to-right), the portion of *2MB* entries in the TLB decreases and the number of *perforated* page entries increases. As the *fragmentation* of the perforated pages increases (8b to 8a) the portion of 4KB hole pages in the TLB increases, reducing its effective coverage.

***Fragmentation* per block:** With increased *fragmentation*, the perforated pages access the *hole region/page bitmaps* more frequently and bring more bitmaps and 4KB hole page translations into the TLB. These occupy TLB entries that could be used for large page entries, thereby reducing its effective coverage. Figures 8 (a) and (b) show the effect of such *fragmentation* on the TLB. As the fragmentation decreases from 75% (a) to 25% (b), the ratio of *perforated* and *2MB* TLB entries increases while *4KB* (needed for hole pages) decreases. The analysis suggests that performance is better for perforated pages with up to 50% hole pages (red triangles in Figure 7),

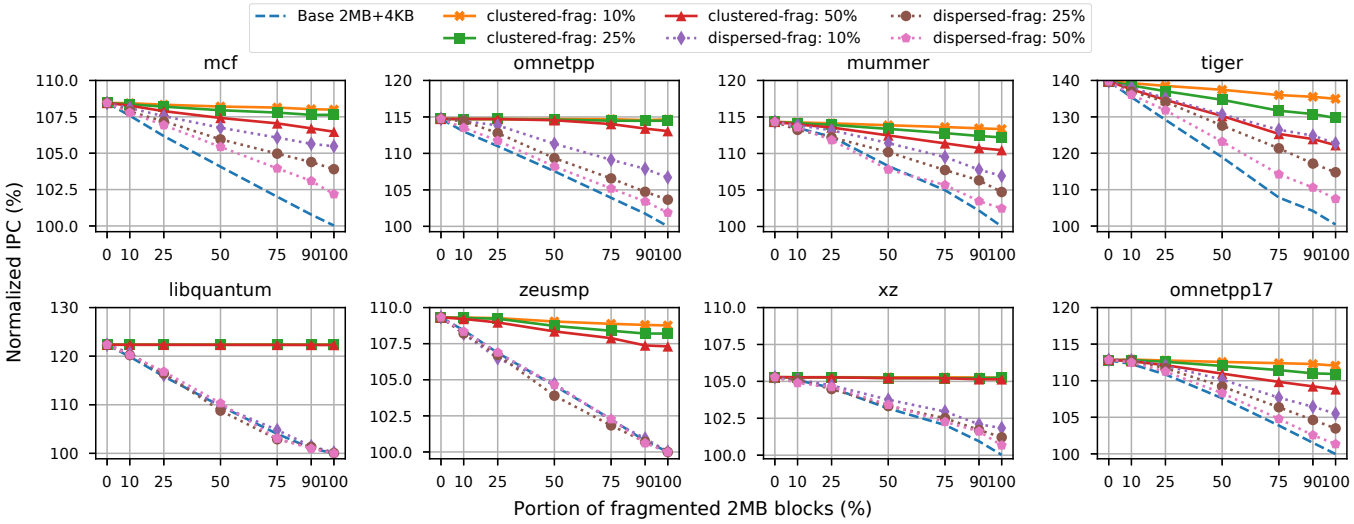

Fig. 9. Performance of native workloads, normalized to the performance of base 4KB TLB

but beyond that performance is similar to the baseline 2MB + 4KB design.

Hole *type*: Hole pages can be either *unallocated* or *allocated*. As unallocated hole pages are not accessed, the corresponding hole pages are not inserted into the TLB. Therefore, the more unallocated holes, the greater the effective TLB coverage and performance.

Figure 7 shows the extreme case of having all holes unallocated as a black dotted curves. For the clustered (left) unallocated hole curves, each black dotted curve represents different degrees of fragmentation (10% top, 90% bottom). In this case, the hole bitmap filter is effective for smaller degrees of fragmentation and we see better performance for lower degrees of fragmentation. However, for the dispersed case (right) the filter is ineffective, resulting in slightly larger performance drops due to more hole bitmap lookups. Figure 8 shows the breakdown of how TLB entries are used for allocated (c) and unallocated (d) holes. With unallocated holes, as the corresponding 4KB pages are not loaded into the TLB, and perforated pages are able to use the entries to store more *perforated*, *2MB*, and *bitmap* entries.

This data shows that perforated pages can provide significantly better performance in an environment with unallocated pages, particularly if they are clustered. We see that even when the unallocated hole pages are *dispersed* (worse performance), at the worst case (portion: 100%, frag: 90%), perforated page performance drops by 31.2% from the best performing configuration (2MB with portion of 0% perforated page). This is 2.4× faster than the Baseline 2MB+4KB TLB.

Hole *distribution*: Perforated pages perform better with *clustered* holes as the hole region bitmap can filter many lookups to the page region bitmap, thereby saving the additional L2 TLB access required to check the bitmap as well as the L2 TLB capacity to store the bitmap. With *dispersed* holes, the filtering is ineffective as the holes are distributed across all regions. This effect is shown in Figure 8 (b) and (c). The *clustered* distribution allows for effective filtering (avoids

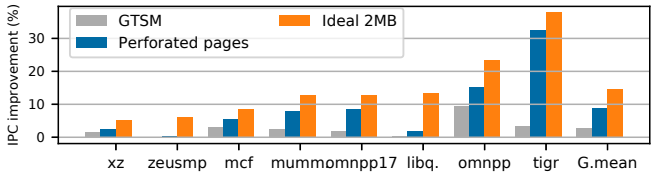

Fig. 10. Performance (IPC) improvement of GTSM, perforated pages, and ideal 2MB normalized to 4KB TLB baseline.

68% of bitmap lookups), which results in fewer bitmaps being inserted into the TLB (58% less).

C. Application Performance

Figure 9 shows application performance for the different distributions and varying amounts of *fragmentation* and *portions* of fragmented 2MB blocks. We do not evaluate unallocated hole pages, as these workloads do not free unused pages back to the OS and we assume that all memory has been allocated.

Across all applications, perforated page performance decreases with the *portion* of fragmented blocks and the *fragmentation* within the blocks. The effectiveness of the filtering is clearly seen in the significantly better performance of the *clustered* (solid line) distributions compared to the *dispersed* (dashed line) ones. In the case of *mcf* at 50% fragmentation and 50% fragmented blocks, the hole region bitmap filter eliminated up to 85% of bitmap accesses and resulted in 26% fewer bitmap entries in the TLB, enabling greater effective TLB coverage and better translation performance. For *libquantum* the absolute TLB MPKI were rather low (0.43, 0.92, for clustered and dispersed), but it is heavily sensitive to translation latency. As a result, it sees significantly worse performance for *dispersed* fragmentation due to the second L2 TLB access to the bitmap required for each non-filtered access.

From the real-world fragmentation study (Section III-C) we found that 50% of the allocated memory was unfragmented

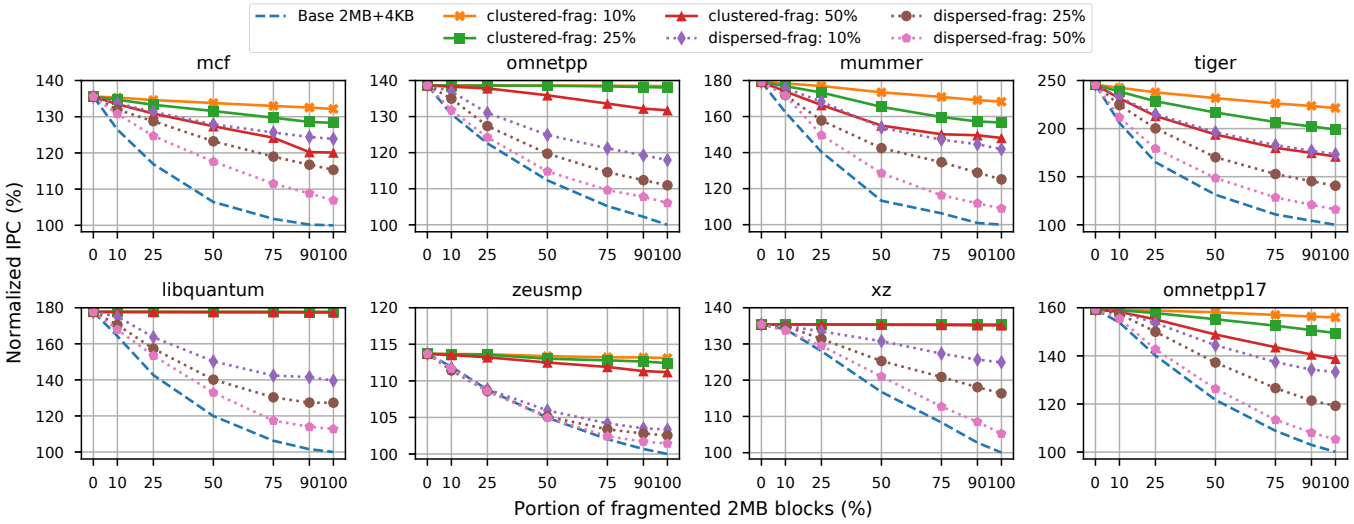

Fig. 11. Performance of virtualized executions, normalized to the performance of base 4KB TLB on a virtualized system

(large pages) and the remaining 50% fragmented with a degree of 25% (i.e. 25% of sub-pages in each large page would needed to be holes) but clustered. For the corresponding configurations in Figure 9, we see that perforated pages are able to achieve between 93.2% and 99.9% of the ideal large page performance (no fragmentation), and provide between 2.0% to 11.5% IPC improvement over conventional TLBs in the fragmented system.

D. Comparison to GTSM

Gap-Tolerant Sequential Mapping (GTSM) [19] enables large page allocation in the presence of memory-fault (hole) fragmentation. Memory faults of even 1% are shown to prevent the use of large pages [19]. GTSM divides the physical memory into 4MB chunks and provides flexible mappings using sub-blocks of 64KB to generate a single 2MB large page within each 4MB chunk[6]. The remaining sub-blocks can then be used as regular 4KB pages. While this allows for mapping around faults, it limits the allocation of large pages to one-half of the available memory. Our work provides a more flexible mapping approach by punching holes for the faulty physical pages within large pages, remapping the holes to functional regular pages, thereby enabling the use of these, otherwise faulty, large pages by utilizing perforated pages.

We compare GTSM to our work for the specific fragmentation scenario GTSM is targeting: 5.5% randomly placed faults, resulting in zero un-fragmented 2MB blocks. Beyond this error rate GTSM cannot provide even 50% of the memory as large pages. We increase the size of the L2 and L3 page walker caches to 32 entries for GTSM, as GTSM is sensitive to the size of the page walker cache, and these are the original parameters of GTSM [19]. Figure 10 shows the performance improvement of our perforated pages, GTSM, and an ideal 2MB (no-fragmentation) scenario, normalized to the IPC of running with only 4KB pages. The analysis shows that perforated pages outperform GTSM by 6.2% on average, even for this scenario with very limited fragmentation.

[6]GTSM supports three different large page sizes, 1MB, 2MB, and 4MB, but the fundamental design is the same.

E. Virtualized System

We implemented the two-dimensional page walk [47] for virtualization in our simulation infrastructure by mapping the guest memory and (guest) page tables using another set of (host) page tables. Any access to the guest physical space (native physical memory access) is augmented with an address translation from guest physical to the host machine address, as shown in Figure 6.

As described in Section V, reading in page bitmaps under virtualization requires reading both guest and host bitmaps, in addition to the standard 2D page walk. We found that even with the state-of-art page caching for two dimensional page walk, the page walk resulted in an average of six memory accesses (still many fewer than the default 24). Figure 11 shows using perforated pages is beneficial compared to using regular pages, which experiences more page walks due to less TLB coverage.

In virtualized systems, the performance improvements from large pages are greater than in native systems due to the increased TLB miss costs. As the cost of the page table walk becomes greater, we observe more benefit from perforated pages: while *libquantum* did not perform well with the *dispersed* distribution, in a virtualized context it now outperforms the non-perforated page baseline. Based on the same configurations that we observed in the fragmentation study, we see that perforated pages achieve between 78.9% and 99.9% of the ideal large page performance, and provide between 7.2% to 48.0% IPC improvement over conventional TLBs in the fragmented system.

In our evaluation we have explored the effects of a fragmented memory environment and shown that perforated pages are far more effective at retaining the benefits of larger effective TLB reach across a variety of fragmentation scenarios than the standard 4KB/2MB page system. These benefits increase

further in a virtualized environment where TLB misses are even more expensive, and decrease when the holes are not clustered in the perforated page, as filtering becomes less effective, requiring both more L2 TLB accesses for the bitmap and more storage for the hole pages.

VIII. Conclusion

While large pages improve translation efficiency by increasing effective TLB reach, they have strict requirements for contiguous physical memory regions that both limit the contexts in which they can be used and results in physical memory bloat for applications with sparse memory usage. In this work we addressed this inflexibility by introducing perforated pages, which allow the OS to *punch* out 4KB *holes* in 2MB large pages. This provides the benefit of large pages even when the physical backing memory region is fragmented (immovable pages) or if shared pages have variations (different permissions), while also removing the requirement that large pages allocate the full physical memory (reducing bloat).

We implement perforated pages efficiently by building upon the existing hierarchical page table with shadow PTE entries to translate holes, and by providing two-levels of hole bitmaps that are filtered and cached in the L2 TLB. Our results show that perforated pages can provide most of the benefits of the extended reach of large pages, from 2.0% to 11.5% improvement in native environments and 7.2% to 48.0% improvement in virtualized environments, even in the presence of realistic memory fragmentation which prevents the use of large pages.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF-2019R1A2B5B01069816), the Institute for Information & communications Technology Promotion (IITP-2017-0-00466), the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows Program, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (715283), The NRF and IITP projects are funded by the Ministry of Science and ICT, Korea.

References

[1] "How to use the Kernel Samepage Merging feature," https://www.kernel.org/doc/Documentation/vm/ksm.txt.

[2] "Intel Skylake," https://www.7-cpu.com/cpu/Skylake.html.

[3] "MongoDB recommends disabling huge pages," https://docs.mongodb.com/manual/tutorial/transparent-huge-pages.

[4] "Redis Administration – Redis," https://redis.io/topics/admin.

[5] "SPEC CPU 2017," https://www.spec.org/cpu2017/.

[6] "Splunk recommends disabling huge pages," https://docs.splunk.com/Documentation/Splunk/7.3.1/ReleaseNotes/SplunkandTHP.

[7] "Transparent Page Sharing (TPS) in hardware MMU systems (1021095)," https://kb.vmware.com/s/article/1021095.

[8] "VoltDB recommends disabling huge pages," https://docs.voltdb.com/AdminGuide/adminmemmgt.php.

[9] J. Ahn, S. Jin, and J. Huh, "Fast Two-Level Address Translation for Virtualized Systems," *IEEE Transactions on Computers*, vol. 64, no. 12, pp. 3461–3474, 2015.

[10] J. Ahn, S. Jin, and J. Huh, "Revisiting Hardware-assisted Page Walks for Virtualized Systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12, 2012, pp. 476–487.

[11] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris, "Enhancing and Exploiting Contiguity for Fast Memory Virtualization," in *Proceedings of the 47th International Symposium on Computer Architecture*, ser. ISCA '20. IEEE, 2020.

[12] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A mechanism for speculative address translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. IEEE, 2011, pp. 307–317.

[13] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, 2010, pp. 48–59.

[14] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *Proceedings of the 2013 40th Annual IEEE/ACM International Symposium on Computer Architecture*, 2013, pp. 237–248.

[15] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *Proceedings of the 2008 13th Annual IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 26–35.

[16] A. Bhattacharjee, "Large-reach Memory Management Unit Caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. ACM, 2013, pp. 383–394.

[17] A. Bhattacharjee, "Translation-Triggered Prefetching," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. ACM, 2017, pp. 63–76.

[18] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[19] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, ser. HPCA '15. IEEE, 2015, pp. 223–234.

[20] M. Franklin, D. Yeung, n. Xue Wu, A. Jaleel, K. Albayraktaroglu, B. Jacob, and n. Chau-Wen Tseng, "BioBench: A Benchmark Suite of Bioinformatics Applications," in *Proceedings of the 2005 Annual IEEE/ACM International Symposium on Performance Analysis of Systems and Software*, vol. 00, 2005, pp. 2–9.

[21] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee, "Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '15, 2015, pp. 39–51.

[22] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

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

[24] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Intel, 2016.

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

[26] D. Jevdjic, S. Volos, and B. Falsafi, "Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. ACM, 2013, p. 404–415.

[27] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant Memory Mappings for Fast Access to Large Memories," in *Proceedings of the 2015 42Nd Annual IEEE/ACM International Symposium on Computer Architecture*, 2015, pp. 66–78.

[28] M. K. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy Translation Coherence," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. ACM, 2018, p. 651–664.

[29] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and Efficient Huge Page Management with Ingens," in *Proceedings of the 2016 12th USENIX Conference on Operating Systems Design and Implementation*, 2016, pp. 705–721.

[30] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings*

of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005, pp. 190–200.

[31] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched Address Translation," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. ACM, 2019, pp. 1023–1036.

[32] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched Address Translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. ACM, 2019, p. 1023–1036.

[33] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 89–104, Dec. 2002.

[34] A. Panwar, S. Bansal, and K. Gopinath, "HawkEye: Efficient Fine-grained OS Support for Huge Pages," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, 2019, pp. 347–360.

[35] A. Panwar, A. Prasad, and K. Gopinath, "Making Huge Pages Actually Useful," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, 2018, pp. 679–692.

[36] C. H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations," in *Proceedings of the 2017 44th Annual IEEE/ACM International Symposium on Computer Architecture*, 2017, pp. 444–456.

[37] C. H. Park, T. Heo, and J. Huh, "Efficient Synonym Filtering and Scalable Delayed Translation for Hybrid Virtual Caching," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, 2016, pp. 90–102.

[38] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '15. IEEE, 2015, pp. 1–12.

[39] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *Proceedings of the 2014 20th Annual IEEE International Symposium on High Performance Computer Architecture*, 2014, pp. 558–567.

[40] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 258–269.

[41] J. H. Ryoo, M. R. Meswani, A. Prodromou, and L. K. John, "SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization," in *Proceedings of 23rd International Symposium on High Performance Computer Architecture*, ser. HPCA '17. IEEE, 2017, pp. 349–360.

[42] J. H. Ryoo, S. Song, and L. K. John, "Puzzle Memory: Multifractional Partitioned Heterogeneous Memory Scheme," in *Proceedings of the 36th International Conference on Computer Design*, ser. ICCD '18. IEEE, 2018, pp. 310–317.

[43] V. Seget, "VMware Transparent Page Sharing (TPS) Explained," https://www.vladan.fr/vmware-transparent-page-sharing-tps-explained, 2017.

[44] M. Swanson, L. Stoller, and J. Carter, "Increasing TLB reach using superpages backed by shadow memory," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ser. ISCA '98. IEEE, 1998, pp. 204–213.

[45] S. van Schaik, K. Razavi, B. Gras, H. Bos, and C. Giuffrida, "Reverse engineering hardware page table caches using side-channel attacks on the mmu," Vrije Universiteit Amsterdam, Tech. Rep., 2017.

[46] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation Ranger: Operating System Support for Contiguity-aware TLBs," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. ACM, 2019, pp. 698–710.

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

[48] L. Zhang, E. Speight, R. Rajamony, and J. Lin, "Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. ACM, 2010, p. 159–168.