Tailored Page Sizes

Faruk Guvenilir, Yale N. Patt
The University of Texas at Austin

***Abstract*—Main memory capacity continues to soar, resulting in TLB misses becoming an increasingly significant performance bottleneck. Current coarse grained page sizes, the solution from Intel, ARM, and others, have not helped enough. We propose Tailored Page Sizes (TPS), a mechanism that allows pages of size 2^n, for all n greater than a default minimum. For x86, the default minimum page size is 2^{12} (4KB). TPS means one page table entry (PTE) for each large contiguous virtual memory space mapped to an equivalent-sized large contiguous physical frame. To make this work in a clean, seamless way, we suggest small changes to the ISA, the microarchitecture, and the O/S allocation operation. The result: TPS can eliminate approximately 98% of page walk memory accesses and 97% of all L1 TLB misses across a variety of SPEC17 and big data memory intensive benchmarks.**

I. Introduction

Page based virtual memory has been a fundamental memory-management component of modern computer systems for decades [20], [22], [36]. Virtual memory provides each application with a very large, private virtual address space, resulting in memory protection, improved security due to memory isolation, and the ability to utilize more memory than physically available through paging to secondary storage. In addition, applications do not have to explicitly manage a single shared address space; the virtual-to-physical address mapping is controlled by the operating system and hardware. Current systems divide the virtual address space into conventional, coarse-grained, fixed size, virtual pages which are mapped to physical frames via a hierarchy of page tables. For example, x86-64 supports page sizes of 4KB, 2MB, and 1GB. The smallest page size is often referred to as the base page size; larger page sizes are called superpages or huge pages. Translation Lookaside Buffers (TLBs) cache virtual-to-physical translations to reduce the cost of page table walks.

The current trend of increasing computer system physical memory capacity continues. Client devices with tens of gigabytes of physical memory and servers with terabytes of physical memory are becoming commonplace. Applications that leverage these large physical memory capacities suffer costly virtual-to-physical translation penalties due to realistic constraints on TLB sizes.

At the 4KB page size, a typical L1 TLB capacity of 64 entries will only cover 256KB of physical memory. This problem is referred to as limited TLB reach. With larger page sizes, current processors typically contain multiple L1 TLBs, one for each supported page size as shown in Figure 1 [30]. Even at the 1GB page size, a typical L1 TLB capacity of 4 entries for this page size will span only 4GB of physical memory.

We thank Intel Corporation and Microsoft Corporation for their generous financial support of the HPS Research Group.

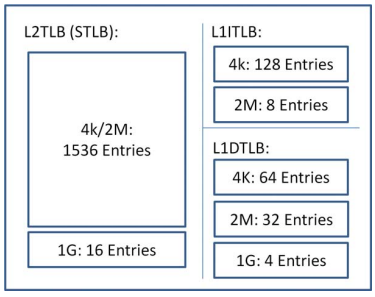

Fig. 1. TLBs in Recent Intel Skylake Processor

Prior work [8], [13], [23], [31], [32], [34] has demonstrated that some applications can spend up to 50% of their execution time servicing page table walks. Large L2 TLB capacities with thousands of entries reduce some of the impact from infrequent, but still very costly, page walks. Figure 2 shows the percentage of total application execution time the processor spends on page walks, as collected from performance counter data on physical hardware with Transparent Huge Pages active. Three cases are shown: 1) native execution with no interference, 2) native execution with a simultaneous multi-threading (SMT) hardware thread competing for TLB resources, and 3) virtualized execution with two dimensional page walks. While native execution page walk overhead is generally modest due to the large L2 TLB capacity, the results show SMT interference and virtualized execution can cause significant increases in page walk overhead. Page walk overhead will further increase with upcoming five level page tables [29].

High numbers of L1 TLB misses can additionally impose a performance penalty. Figure 3 demonstrates the performance improvement of a perfect L1 TLB over a perfect L2 TLB baseline. This study was performed with cycle-based simulation modeling out-of-order effects (more details in Section IV-A). The out-of-order window can often hide many L1 TLB misses by overlapping this latency with other useful work. But, when memory accesses are on the critical path of execution (e.g., linked data structure traversal), even frequent L1 TLB misses can cause an appreciable performance penalty as shown. Both limited L1 and L2 TLB capacity still play major roles in translation overhead.

The coarse granularity of conventional page sizes in the most common modern processor families (x86-64 and ARM) are inadequate. For example, consider a single 256MB data structure. Provided the operating system is able to identify free contiguous memory such that allocating any available page size for this new data structure is possible, the tradeoff between

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00078

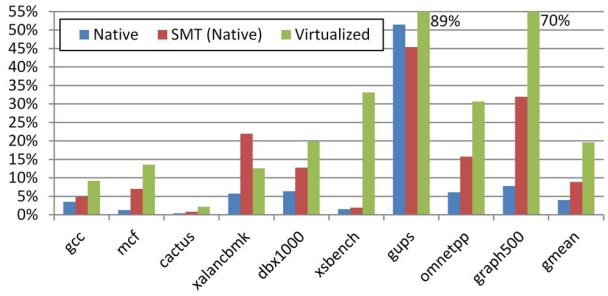

Fig. 2. Page Walk Overhead: Percent of Execution Time Spent Page Walking

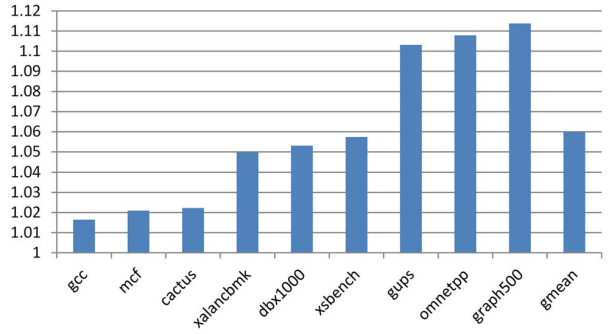

Fig. 3. Speedup of Perfect L1 TLB over Perfect L2 TLB Baseline

page sizes presents serious problems with any choice. Choosing the 2MB page size requires 128 TLB entries, already well over the L1 TLB capacity for this single data structure. Choosing the 1GB page size results in 768MB wasted physical memory, internal fragmentation of 75%. The current trend to increase the gaps between consecutive conventional page sizes only worsens the tradeoff. For example, ARM supports a translation granule with consecutive page sizes of 64KB and 512MB [5].

These problems, along with the continued growth of physical memory capacity, indicate change in the virtual memory mechanism is necessary to deal with the growing translation overhead. We propose Tailored Page Sizes (TPS), a simple and clean extension to current processor architectures that reduces translation overhead, significantly lowers page walk memory references, and substantially reduces L1 TLB misses in TLB intensive workloads that will continue to scale with increasing physical memory capacity. TPS introduces support for pages sized at any power-of-two larger or equal to the base page size (e.g., 4KB in x86-64). TPS includes changes to operating system software, the ISA, and the microarchitecture. When fragmentation is high, resulting in insufficient contiguity to utilize conventional pages well, TPS allows the OS to leverage what contiguity it can for performance with intermediate page sizes.

TPS leverages the natural levels of address space contiguity that occur due to the standard operation of the OS buddy allocators and memory compaction daemon. In addition, applications already perform mapping system calls at runtime to large contiguous regions of their virtual address space, but current OS implementations split these mapping requests into however many smaller pages are required to compose the larger request. Utilizing these common properties, TPS is able to create the necessary virtual-to-physical mappings with only a small number of appropriately sized fine-grained pages for many memory intensive applications. Each tailored page requires a single PTE cached in the TLB.

TPS does not sacrifice backwards compatibility. The current paging model is still supported, with the additional flexibility of choosing from many newly available page sizes. TPS provides a path forward for OS implementations to gradually adopt the introduced page sizes, while retaining the option of solely choosing the original, conventional page sizes for whatever reasons may arise.

In summary, our key contributions are as follows:

- We propose the TPS ISA extension to allow support for pages of any power-of-two size larger or equal to the base page size.
- We define the microarchitectural changes necessary to facilitate virtual-to-physical address translation of the newly-supported page sizes. This includes changes to the hardware page walking mechanism, and a straightforward L1 TLB enhancement to support fast translations.
- We describe and justify via simulation the straightforward OS changes that must be implemented to leverage the newly supported page sizes.
- We show that, across a variety of benchmarks, TPS is able to raise the L1 TLB hit rate to more than 99%, almost eliminating execution time spent accessing L2 TLBs and page walking.

II. Background

This section provides background on the current common virtual memory translation implementations. The following topics will directly apply to most architectures utilizing a hierarchical page table tree that supports conventional super-pages (e.g., ARM, x86-64). We will primarily cover the x86-64 architecture for ease of explanation.

A. Virtual Memory Hardware

Virtual memory provides each process with the illusion of a very large, private address space. Hardware performs address translation at runtime, while the operating system (OS) is responsible for partitioning the virtual address space into virtual pages, and mapping them to physical frames. The architecture defines the contract between the hardware and software.

Page Table: The page table contains the mappings from virtual page to physical frame for the process address space. The page table itself is stored in memory (and mapped as part of the virtual address space), and consists of page table entries (PTEs). Each PTE contains a single mapping from virtual page to physical frame, as well as bookkeeping bits for protection and other purposes. Intel x86-64 currently implements the page table as a four level hierarchical radix tree. Every memory reference executed by a process uses virtual addresses. The processor core contains a hardware page table walker to start

with the original virtual address, and traverse the page table tree to eventually produce the PTE containing the page frame number for the requested virtual address.

Page Sizes: The base page size is the smallest unit of virtual-to-physical translation. In x86-64, the base page size is 4KB. Huge pages (also called superpages) simply refer to page sizes larger than the base page size. In x86-64, huge page sizes of 2MB and 1GB are currently supported. Two currently available software mechanisms in Linux are able to leverage huge pages: Transparent Huge Pages (THP) [16] and HugeTLBFS [17].

TLB: Hardware must translate from virtual address to physical address on *every* memory access. Since page table walks are slow (they require potentially multiple memory accesses), processors use TLBs to cache recently used PTEs. On a TLB hit, the translation can be completed in one or a few cycles. On a TLB miss, a page walk is performed, often requiring tens or hundreds of cycles.

It is worth noting that supporting multiple page sizes in a single set-associative TLB is non-trivial. The bits used to index the TLB are typically the least significant bits of the virtual page number. However, the size of the virtual page number depends on the page size, which is unknown while performing the TLB lookup.

MMU Cache: Because a page walk would normally require four memory accesses (one for each level in the page table hierarchy), processors typically contain memory management unit (MMU) caches which contains recently used PTEs from upper levels of the page table hierarchy. A hit in an MMU cache will reduce the number of memory accesses to fewer than four, the number of accesses depending on which level of page table hits in the cache.

B. OS Software for Virtual Memory

The operating system has several components responsible for virtual memory management. The OS maintains all processes' page tables and creates virtual-to-physical mappings upon a process request. We briefly describe the relevant components in the following paragraphs.

Buddy Memory Allocation: The buddy allocator tracks all free physical memory, keeping free lists of power-of-two sized blocks. Each free list is associated with a specific power-of-two size. When an allocation request for a new mapping occurs, the free list that matches the requested size is queried for a free block to use as the physical frame(s) for the necessary mapping(s). If there are no available blocks of the requested size, the free list containing blocks of the smallest size larger than the request is used. The larger-than-required free block is iteratively split in half to produce an appropriately sized free block. The two halves of any split block form a unique pair of buddy blocks (i.e., every block has a unique buddy block). When the split process finishes, the remaining left over free blocks are added to the appropriate free lists based on block size. When a process later deallocates and frees a block of physical memory, the allocator checks if its buddy block is also free. If so, the blocks merge, and the buddy block merge process repeats until the checked buddy block is not free. The block resulting from the merge operation is added to the appropriate free list. Thus, the buddy allocator splits and merges blocks of physical memory as appropriate during allocations and deallocations.

Demand Paging and Lazy Allocation: With demand paging, the operating system only performs page table setup upon receiving virtual-to-physical mapping requests. The OS marks PTEs initially as invalid since they do not yet actually point to physical frames. When a process first references a location on a newly mapped page, a page fault occurs, which notifies the operating system that a demand for the page exists. At this point, the page frame is appropriately selected and initialized. Although the operation of the buddy allocator and the prevalence of infrequent larger mapping requests easily lead to utilizing many contiguous page frames for contiguous virtual pages, this may not always be possible in cases of high allocation contention and interleaving of scattered demand requests.

Memory Compaction: The memory compaction daemon [18] is primarily responsible for reducing external fragmentation. With memory compaction, scattered blocks of used physical memory are migrated to adjacent locations to create larger, contiguous blocks of free memory. Memory compaction can also be explicitly invoked if sufficient contiguous memory cannot be found when the OS receives an allocation request.

III. Our Solution

Tailored Page Sizes is a simple and clean extension to current processor architectures. TPS leverages key existing features of the virtual memory mechanisms to reduce translation overheads. In the following subsections, we discuss the architectural implementation details for TPS, the OS features needed to support TPS, and additional cross-layer considerations.

A. Architectural Considerations

1) Page Table and PTE changes: To support TPS, the PTE structure and page walk process need to be updated. Figure 4 shows an overview of the typical x86-64 hierarchical page table. The current implementation supports three page sizes: 4KB, 2MB, and 1GB. For the 4KB page size, a page walk requires four physical memory accesses. If a larger page size is used, fewer accesses are required. For example, with a 2MB page, a bookkeeping bit in the 2nd level PTE identifies that this level is the final step of the page walk process. Similarly, for a 1GB page, a bit in the 3rd level PTE identifies that this level is the final step of the page walk process.

To support pages at any power-of-two (larger than the base page) size, the PTE must include an additional field. There are nine page size options from 4KB up to (not including) 2MB. Four reserved bits of the PTE could be used to indicate what size page a given PTE is pointing to. But, reserved bits in the PTE are limited, so we also propose an alternative solution that only requires one reserved bit (called T in Figure 5). Note that larger pages have fewer page frame number (PFN) bits. For example, assume 40 bits of physical address. A 4KB page has 12 bits of page offset and 28 PFN bits, while an 8KB page

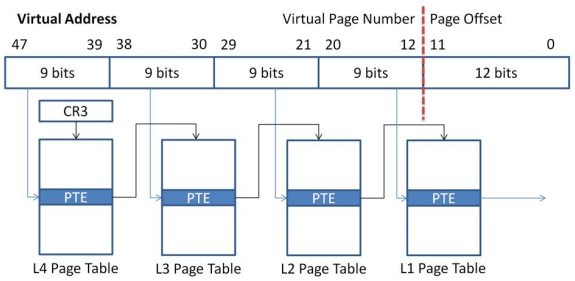

Fig. 4. Four Level Hierarchical Page Table

has 13 bits of page offset and 27 PFN bits. The one reserved bit (T) specifies whether the PTE corresponds to a standard conventional page (e.g., 4KB), or a tailored intermediate page (e.g., >4KB). If the page is tailored, the PTE must now have at least 1 bit of PFN that is thus unused (e.g., bit s_0 in Figure 5). This PTE bit (that would otherwise be part of a PFN) specifies whether the page size is the smallest tailored size (e.g., 8KB), or larger (e.g., >8KB). If the page is larger, then the PTE must have yet another unused PFN bit (e.g., bit s_1), and this process can repeat (similar to RISC-V PMP NAPOT encodings [48]). This can be easily implemented in hardware with a priority encoder to identify the page size.

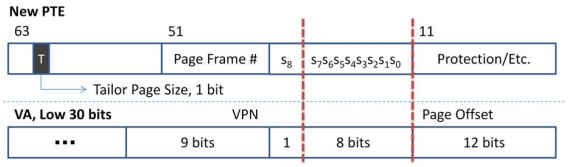

Fig. 5. Identifying Page Size with Only One PTE Bit

When performing the page walk, hardware has a challenge: the page size is not known until the PTE is read. Fine-grained tailored page sizes may require one additional memory access for the page walk process. Figure 6 shows the details.

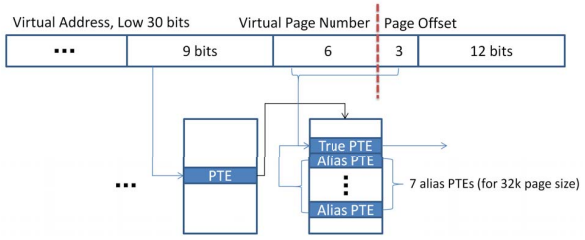

Fig. 6. Extra Page Walk Access with Alias PTE

This example assumes a 32k page size, which implies the address has 15 bits of page offset. We call the nine bit subsection of a virtual address used to identify a specific PTE within the page table a *page table index*. Because this page table index is used to look up a 512 entry page table, a tailored page size may have multiple different page table index values that point to different PTEs, but actually represent the same page. Thus, when a tailored page is created, all PTEs that could be pointed to by an address on that page are updated to indicate what the page size is. During the final access of the page walk process, the page size field in the PTE is examined. This field indicates how many bits of the nine bit subsection are not part of the virtual page number, but actually part of the page offset. Here, the additional memory access is performed with the bits of virtual address that are actually part of the page offset set to zero. This results in one PTE being the "true" PTE for the tailored page, with the rest (called "alias PTEs") simply indicating one more access is necessary in the page walk. Note that the goal of TPS is to nearly eliminate TLB misses in most cases, so the one additional memory access required by only some page walks actually occurs rarely and is outweighed by the reduction in number of page walks (see Section IV-B). In addition, the time spent to set up all the alias PTEs is relatively inconsequential. With only conventional page support, these PTEs would need to be set up anyway as true PTEs for the numerous additional pages that would be created.

Maintaining alias PTEs as complete copies of the true PTE is a valid approach. This approach is functionally correct with TPS and does not require an extra lookup in the page walk process. However, the tradeoff with this approach is that any PTE update would require updating all alias/true PTEs corresponding to the page. If we generally expect PTE updates to be significantly less frequent than extra lookups induced by the alias PTEs, this approach would be better. Regardless, either approach is possible in a TPS-based system.

2) TLB Design: We update the design of the TLB to support TPS. A recent Intel Skylake Processor [30] includes two levels of TLB for data accesses. The L1 TLB is split into three parts for the three supported page sizes. It contains 64 entries for the 4KB page size, 32 entries for the 2MB page size, and 4 entries for the 1GB page size. The L2 TLB contains 16 entries for the 1GB page size, and 1536 entries for the 4KB/2MB page sizes. To support TPS, we modify the L1 TLB to contain a 32 entry fully-associative (as in other commercial designs [2], [47]) TPS TLB. The TPS TLB takes the place of the existing 32 entry and 4 entry larger page size L1 TLBs. The TPS TLB supports any page size. We still retain the 64 entry 4KB L1 TLB. Existing, productized, AMD designs (e.g., Zen [3]) contain a 64 entry fully-associative any page size L1DTLB (for conventional pages). The 32-entry any page size TPS TLB should reasonably be able to meet similar timing constraints in similar processors. Alternative skewed-associative [44], [53] TLB designs are possible.

In the newly added any-page size L1 TLB, we add a *page mask* field to each TLB entry, as shown in Figure 7. The page mask field is populated when the TLB is filled. The TLB is searched by the Virtual Page Number (VPN) of the memory access. Normally, the VPN is compared to the VPN tag stored within the TLB to identify a hit. In the case of our any-page size TLB, the incoming VPN is first masked with the entry's mask field, then compared to the VPN tag to identify a hit. This adds a single gate delay on every associative TLB lookup, which is unlikely to impact the observed latency of the L1

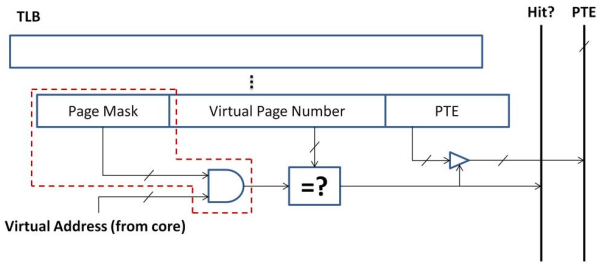

Fig. 7. TLB Hardware. Changes to existing hardware are contained within the dashed box.

TLB lookup.

B. Operating System Considerations

1) Paging and Buddy Allocation: While the standard approach in demand paging does allow intermediate contiguity of mapped physical frames, changes are necessary to extract the full potential of Tailored Page Sizes. Utilizing an eager paging strategy as in [34] is possible. Rather than lazily allocating conventional pages on demand when first accessed, we identify the appropriate tailored page size and eagerly allocate the few tailored pages (which use appropriately sized frames of physical memory provided by the buddy allocator) at allocation request time (e.g., when the process performs an mmap system call).

However, there are some drawbacks to changing the OS's paging strategy. Application start-up time and allocation latency may be adversely affected if the application must wait for entire large pages to be initialized (which is also a problem using standard large page sizes). Additionally, the larger the page size, the more costly the swapping. But, swapping is becoming less common: servers running big memory workloads often run with swapping disabled, preferring to keep entire working sets in memory to minimize latency [8], [37]. Apple iOS also does not swap to secondary storage [4]. The continuing increase in physical memory capacity significantly reduces the frequency of swapping [8].

To further improve the robustness of TPS in the cases where these concerns present significant problems, we utilize an alternative to both demand and eager paging: *demand paging with frame reservation*. Our approach is similar to the reservation based paging strategy used in FreeBSD [14], [41] and previously proposed in [42], [56]. Reserved frames are neither free nor in use; they can transition to either state depending on system demands.

When a large size allocation request occurs, the operating system still identifies the desired optimal tailored page size N, but does not initially allocate the entire region as in standard demand paging. Instead, the buddy allocator is queried for a free memory block of size N, which is then removed from the allocator free list and placed into a *paging reservation table* that also saves the requested range of virtual addresses. This free memory block of size N is reserved for virtual addresses within the range.

When the first demand request (memory access) occurs to a location within that range, only the conventional page containing the demand request is allocated (as in standard demand paging). The appropriate frame is chosen from the block of size N previously saved into the paging reservation table, rather than the buddy allocator free list. For a subsequent demand request to a yet-to-be-mapped part of the virtual address range, the already mapped page is grown (also called "page promotion", or "upgrading the page size") to include the location of the new request. The physical memory locations are identified from the paging reservation table.

Upgrading the page size simply requires updating the appropriate PTEs for the newly-mapped larger page. While the newly mapped memory must be appropriately initialized, no changes to or migration of the previously mapped frame is necessary. TPS's support for page sizes at every power-of-two allows frame reservations to be incrementally filled with growing page sizes by the OS as demand requests within the reserved region arrive. Unlike the FreeBSD's approach, TPS can adjust page promotion aggressiveness based on a utilization threshold. To prevent memory footprint bloat, TPS can be configured to only upgrade to larger page sizes when 100% of the larger page's constituent pages are utilized. Conversely, for better TLB performance, TPS could also be configured to upgrade to larger page sizes when lower percentages (e.g., 50%) of the constituent pages are utilized. The promotion threshold can be adjusted between the two extremes to balance the tradeoff based on the machine's memory load.

These straightforward changes to the paging algorithm and allocator allow the OS to map an application's utilized virtual address space with a minimal number of appropriately sized pages.

2) Fragmentation: A primary general drawback of supporting more than one page size is fragmentation. When fragmentation does become a problem, TPS always has the simple fall back of only allocating from the originally supported conventional page sizes. In addition, the OS can request memory compaction when fragmentation is high to present more opportunities for fine-grained tailored allocations and merges, as with standard conventional allocations. External fragmentation occurs when free memory blocks and allocated memory blocks are interspersed. This can prevent a large contiguous allocation even though total free memory exceeds the allocation size. Internal fragmentation occurs when part of an allocated memory block is unused.

External Fragmentation: To minimize external fragmentation, TPS conservatively only upgrades page reservations when utilization is near 100%, as described in Section III-B1. As external fragmentation increases, the OS will be unable to create desired page sizes and reservations due to the lack of memory contiguity, and be forced to create smaller pages. Under heavy external fragmentation, conventional large pages cannot be created; however, it is possible that whatever minimal memory contiguity is available can be leveraged by TPS to create intermediate tailored page sizes.

Internal Fragmentation: With larger pages, the potential for more waste due to internal fragmentation is increased. The most conservative policy is to completely disallow any extra loss due to internal fragmentation (as compared to exclusive use of the smallest page size) by creating reservations out of the fewest number of pages that exactly spans the reservation (e.g., an aligned 28kB request results in 16kB+8kB+4kB).

On the aggressive side, TPS can choose the smallest size page still larger than the requested memory allocation. Because only power-of-two page sizes are supported, this could lead to approximately 50% waste in some cases. For example, an allocation request of 2052 KB would result in a 4 MB page reservation being created. When fragmentation is high, TPS can throttle towards more conservative page size reservations. This choice presents a tradeoff between magnitude of internal fragmentation, and the number of TLB entries required to translate a single logical region.

Existing OS proposals like Ingens and Translation Ranger [38], [60] already address the issues of maximizing memory contiguity and managing larger page allocations to improve virtual memory performance and reduce fragmentation. These existing approaches can be used on a TPS based system to synergistically maximize translation benefits. With the inclusion of these techniques, TPS would have greater ability to create the largest-sized appropriate pages to minimize the number of TLB entries required by the application.

3) Memory Compaction and Page Merging: TPS does not require any changes to the standard memory compaction daemon operation to improve performance. Long-running large footprint benchmarks already benefit from upfront compaction at application launch to create large conventional pages; TPS maximizes the benefit from this sort of compaction.

A possible future optimization might be to allow the memory compaction daemon to be aware of physical frames that could be potentially merged into a single larger frame that would require only one PTE for translation. A page merge is possible when two appropriately aligned and adjacent frames map contiguous virtual address space with identical permissions. Multiple allocations to contiguous virtual addresses may have been unable to use contiguous frames due to other intervening allocations. Whenever memory compaction is performed, the daemon could migrate frames taking into account potential merges. When the PTEs pointing to migrated pages are updated, the OS performs a page merge by updating the relevant PTEs (and invaliding appropriate TLB entries) if the frames were successfully setup for a merge.

C. Other Considerations

1) PTE Accessed and Dirty Bits: The processor is required to update the Accessed and Dirty (A/D) bits for a given PTE when loads read from (stores write to) a page with the bit cleared. The TLB also caches these bits to identify whether the additional store to update the PTE will actually be required. One concern with larger pages in general is that keeping track of accessed and dirty data at a larger granularity may incur additional overheads during swapping and writing back dirty pages to secondary storage. Current large page sizes already have to deal with this problem; there is no additional overhead introduced by supporting more intermediate page sizes. However, as with large pages, when fragmentation is high, swapping frequent, or I/O pressure high due to cleaning dirty pages, the OS has the option of splitting larger pages into smaller pages to reduce the associated costs.

As an alternative, recall that intermediate tailored page sizes will have multiple alias PTEs which are simply used to point to the true PTE. The remaining bits in the alias PTEs are thus unused and could be collected into a bit vector representing the referenced/modified state of a tailored page's constituent conventional pages. The vector can be cached with the TLBs and does not actually need to be loaded on a page walk because the A/D bits exhibit sticky behavior. The first read/write will update the in-memory A/D bit to guarantee it is set, and update the cached TLB bit to prevent extraneous updates. Note that this bit vector need not be strictly tied to the TLB lookup; the bit vector's lookup and update operations can proceed after the standard TLB lookup in parallel with the subsequent memory access pipeline stages. A tailored page can have up to 256 constituent conventional pages. Since tracking up to 512 bits may be too costly both in terms of TLB area and additional memory accesses required, we can impose an upper bound on the bit vector length. For example, a 16 bit limit would significantly reduce costs while still allowing for fine-grained tracking. Each bit's tracking granularity would be a function of the page size. A bit in the PTE can specify whether to enable or disable this fine-grained metadata tracking. The bit vector updates use the same mechanism already used by the existing modify bit update operation and do not block forward progress.

2) TLB Shootdowns: In x86-64, the INVLPG instruction is used to invalidate any out-of-date PTEs that may be stored in the processor TLBs. No changes are needed to the operation of this instruction. As in standard operation, appropriate shootdowns remain necessary during memory compaction. When a page results from merging adjacent pages, the PTE that may have been present in a TLB for the smaller page size will still be correct for its portion of the larger page. For optimal TLB entry replacement, the ideal policy is to only update LRU information for the largest page size which returns a hit. This is unnecessary, however, because as pages grow, the likelihood of extraneous smaller page TLB entries being aged out increases. Thus, no new shootdowns are needed when performing page merges.

3) Copy on Write: Copy on Write is an OS technique that enables multiple virtual pages that contain identical data to point to the same physical frame by maintaining the PTE in read-only state. When a page is written to, a page fault occurs, and the OS copies the frame and updates the mapping. With larger pages, opportunities to use copy-on-write will be reduced, simply because there is lower likelihood such large regions of memory are identical. If there is substantial desire to share a particular small page, the OS can simply prioritize using a smaller page for such a page. If a larger page is read shared,

TABLE I
SIMULATED PROCESSOR CONFIGURATION

Core	4-Wide Issue, 256 Entry ROB, 3.2 GHz Clock Rate
L1 Caches	32 KB I$, 32 KB D$, 64 Byte Cache Lines, 4 Cycle Latency, 8-way Set Associative
Last Level Cache	2MB, 16-way Set Associative, 64 Byte Cache Lines, 10-cycle Latency
TLBs	128 4k + 8 2M L1ITLB; 64 4k + 32 2M + 4 1G L1DTLB; 1536 4k/2M + 16 1G STLB

then upon a write, the OS has multiple options. The OS could only copy the part of the larger page that was written to as a smaller page, and create multiple larger page mappings still sharing the parts of the page that were not written to. This saves copy time and reduces memory utilization. Alternatively, the OS could copy the entire large range, which is more expensive in terms of copy time and memory utilization, but reduces TLB pressure.

IV. RESULTS

A. Methodology

To evaluate the performance impact of our proposal, we perform a two-step evaluation in simulation. Implementing the proposed OS changes in an existing OS requires a physical processor with the proposed ISA support. Also, it is preferable to run workloads from start to finish in order to fully understand the TLB behavior and translation overheads of memory-intensive applications. However, doing this on a cycle-based or full system simulator is infeasible; months of simulation time would be required.

To study the impact of TPS, we primarily evaluate how TLB hit rates are affected. We constructed a PIN-based OS-allocator and virtual memory simulator that traces memory management system calls and all memory accesses. The simulator models the relevant parts of the microarchitecture and operating system. We modeled a realistic TLB hierarchy with MMU caches, and identified the number of accesses, hits, and misses to each level of the hierarchy. Additionally, we model the required OS changes including those to the allocator and application page tables.

To demonstrate the importance of hitting in the L1 TLB, we use ZSim [51], a cycle-based simulator of an x86 superscalar out-of-order processor. The simulator has been strongly correlated to existing Intel processors and faithfully models core microarchitectural details and the memory hierarchy. We added TLBs to the simulator to model the impact of TLB misses. Evaluation is performed on a single-core system. Table I lists the baseline processor configuration.

The benchmarks we evaluate are the SPEC17 suite, Graph 500, GUPS, XSBench, and DBx1000. Since the SPEC17 suite is not fully representative of modern TLB intensive workloads, we profiled all the benchmarks as shown in Figure 8 to determine each benchmark's TLB pressure, as measured in TLB misses per thousand instructions (MPKI). For evaluation, we chose the TLB intensive SPEC17 benchmarks, as measured by an MPKI of greater than five.

Performance counter and real system evaluation was carried out on a machine with a recent Intel Kaby Lake processor and 64 GB of DDR4 memory.

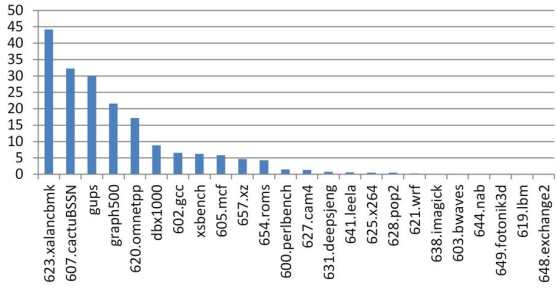

Fig. 8. L1 DTLB MPKI

B. Data

Prior OS work [38] observed that Linux Transparent Huge Pages can result in memory footprint increases of up to 70% for some benchmarks. In our PIN-based simulator, we simulated running with only 2MB pages to see the maximum memory size impact of using only 2MB pages compared to using only 4KB pages on the benchmarks. Results in Figure 9 show only modest increase in memory utilization for the benchmarks, but it is important to note that mixing in the 1GB page size increases the potential memory loss due to internal fragmentation. For all remaining experiments, we ran TPS under the frame reservation strategy, requiring 100% utilization of smaller page sizes before merging into the next larger tailored page size. This approach guarantees identical memory usage to using only the base page size of 4KB, but loses some opportunity to reduce TLB misses.

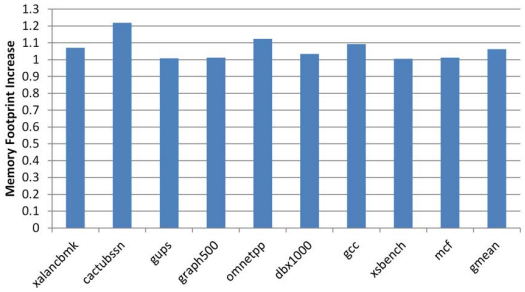

Fig. 9. Increase in Memory Utilization with Exclusive 2MB Pages

Figure 10 shows the PIN-based simulation results. No migration or compaction is performed during this study, but initial memory state is assumed to be lightly-loaded. We report the percentage of L1 DTLB misses eliminated for each benchmark with TPS enabled compared to the baseline of reservation-based Transparent Huge Pages (THP). We also implemented and evaluated the impact of CoLT [46] and RMM

[34] on the L1 TLB hit rate. As shown, TPS eliminates 98.0% of L1 TLB misses on average, while CoLT only eliminates approximately 36.6% of L1 DTLB misses on our benchmarks. RMM eliminates no L1 DTLB misses. This is because RMM introduces a Range TLB at the L2 level of the hierarchy. CoLT has minimal impact for the benchmark GUPS because of its random access behavior. Increasing the reach of each TLB entry by a small factor does little to mitigate a random memory access pattern to gigabytes of physical memory. In contrast, tailoring a few very large pages to the size of this memory region significantly reduces TLB misses.

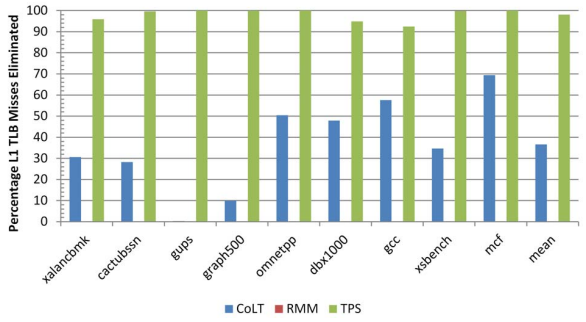

Fig. 10. L1 DTLB Misses Eliminated (Baseline: Reservation-based THP).

Figure 11 shows the reduction in the number of page walk memory references. TPS and RMM have near identical best performance in terms of maximal reduction in page walk memory references. RMM generally slightly outperforms TPS primarily because RMM has no size or alignment restrictions on large regions. While eager paging is best for page walk reduction, this policy can have unacceptable impact on allocation latency in some scenarios. TPS slightly outperforms RMM on the gcc benchmark because of the limited number of available Range TLB entries.

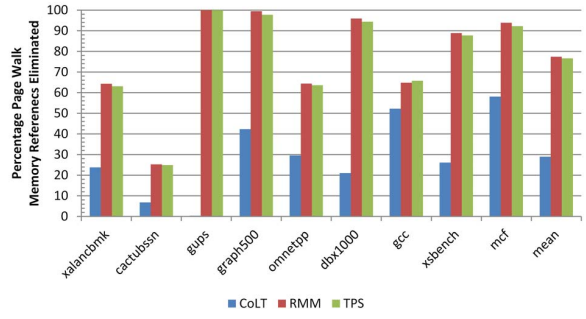

Fig. 11. Page Walk Memory References Eliminated (Baseline: Reservation-based THP).

To approximate performance impact, we break down total execution time, T, into 3 parts:

$$T = T_{IDEAL} + T_{L1DTLBM} + T_{PW}$$

T_{IDEAL} is the ideal execution time of the benchmark assuming no page walks or TLB misses. $T_{L1DTLBM}$ is the execution time lost due to L1TLB misses that hit in the L2TLB (did not cause page walks). T_{PW} is the execution time lost due to page walks.

For the THP baseline case, we measured $T_{L1DTLBM}$ via ZSim simulation.

The performance counter measurements cannot be directly used to calculate T_{PW}. The performance counters measure cycles where a page walker is active; this means that it is possible for an application to be making forward progress due to the out-of-order window even while the page walker is active. Thus, eliminating page walker cycles will not always translate to a one-to-one reduction in execution time. To estimate how much a reduction in page walker cycles will actually be realized as savings in execution time, we collected real machine performance counter information at two configurations, measuring two values at each configuration:

1) Transparent Huge Pages disabled (only 4k pages).
   a) Total Execution Cycles: TC_{THP_d}
   b) Page Walker Cycles: PWC_{THP_d}
2) Transparent Huge Pages enabled.
   a) Total Execution Cycles: TC_{THP_e}
   b) Page Walker Cycles: PWC_{THP_e}

From these two points, we calculated how a reduction in PWC translated to a reduction in TC going from THP disabled to THP enabled. Assuming the trend holds, we can then estimate how much further reductions in PWC will translate to total execution time savings. Figure 12 shows the percentage of PWC savings that directly translates to total execution time savings for each benchmark.

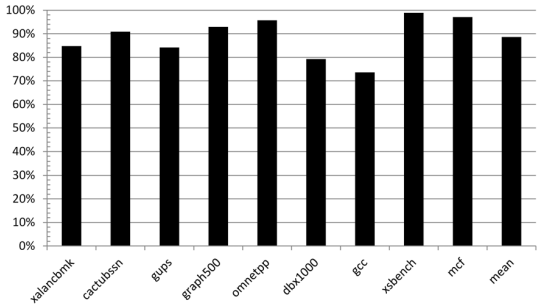

Fig. 12. Savable Page Walker Cycles

We use this performance counter information to calculate T and T_{PW}. Using the equation above, we can solve for T_{IDEAL}.

To determine T with TPS active, we scale T_{PW} by the ratio of page walk memory references eliminated (as determined via simulation), and we scale $T_{L1DTLBM}$ by the ratio of L1 DTLB misses eliminated. We perform the same calculations for CoLT and RMM. Figure 13 shows the speedup results for native execution, running on a core alone.

TPS is able to achieve an average performance improvement of 15.7%, as compared to 9.4% for RMM and 2.7% for CoLT, which is 99.2% of the maximal ideal savings (i.e., eliminating 100% of TLB misses) for TPS.

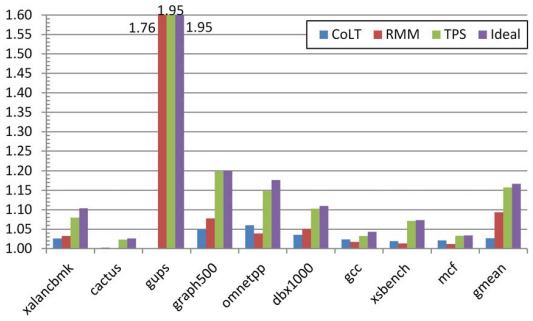

Fig. 13. Speedup - Native (no SMT)

Figure 14 shows the speedup results for native execution, running with a hardware SMT thread competing for core and TLB resources.

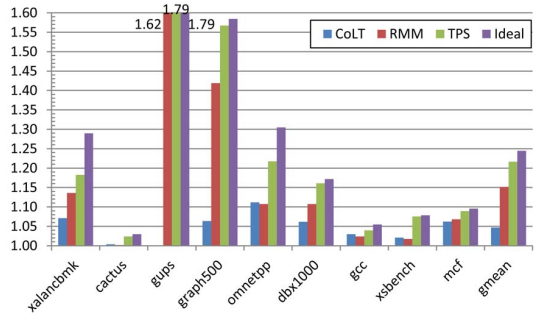

Fig. 14. Speedup - Native (SMT)

TPS is able to achieve an average performance improvement of 21.6%, as compared to 15.2% for RMM and 4.7% for CoLT, which is 97.7% of the maximal ideal savings for TPS. Some relative values (e.g., for GUPS) are lower for SMT compared to running alone. While the absolute number of cycles spent page walking increased when running under SMT, some applications experienced a greater magnitude of total slowdown from the core resource sharing. This resulted in the page walk overhead representing a smaller percentage of the total execution cycles.

We performed a study to evaluate the impact of external fragmentation on TPS. We examined the state of physical memory on our heavily loaded test server running Linux kernel version 3.10, with both Transparent Huge Pages and Memory Compaction enabled and set to `always`. Free memory utilization on the test server was raised to allow just enough for our benchmarks to run (less free memory would cause an out of memory error, which is experimentally uninteresting). Using `/proc/buddyinfo` and `/proc/pid/pagemap`, we identified how much free memory contiguity was available. Figure 15 shows what percentage of free memory could be used by singular page sizes ranging from 4 KB to 16 MB. Each bar represents what percentage of free memory could be used if only that bar's single page size was used for all allocations. Because 4KB is the smallest possible page size, coverage at this size must be 100%. The key takeaway is that even on a heavily loaded and fragmented system, significant intermediate levels of contiguity exist that can be leveraged by TPS, while only a very small portion of memory contiguity is exclusive to the existing conventional page sizes.

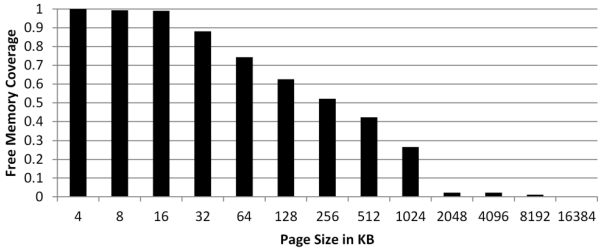

Fig. 15. Free Memory Coverage by Various Page Sizes.

Using the dumped free memory state as input to the PIN virtual memory simulator, we evaluated the potential of TPS under high fragmentation. Figure 16 shows that even under these conditions, TPS can attain significant reduction in TLB misses. No memory compaction takes place throughout the simulation. Note that GUPS sees minimal benefit from TPS under high fragmentation conditions. Due to the random access memory behavior of this benchmark, even intermediate page sizes provide limited benefit when the memory accesses have almost no spatial locality. For other similarly large memory footprint benchmarks like XSBench and Graph500, significant reduction in TLB misses occurs because these benchmarks exhibit some locality in memory references. For long running, large memory benchmarks like GUPS, performing memory compaction at initial allocation time or incremental guided memory compaction over time would help TPS incrementally grow page sizes and reduce TLB misses. As shown in Figure 15, even when memory compaction is running, significant memory contiguity at non-conventional page sizes exists that TPS would be able to take advantage of.

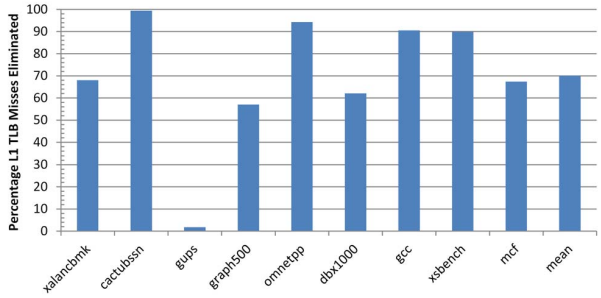

Fig. 16. L1 DTLB Misses Eliminated

TPS increases the OS allocator complexity which can affect application runtime. Figure 17 shows the system time percentage (relative to total execution time) of the workloads. OS allocator work in these memory intensive workloads is very low relative to total execution. The average system time percentage is 0.16%. Even an unrealistic 10x increase in

system work due to TPS changes would not cause significant application slowdown.

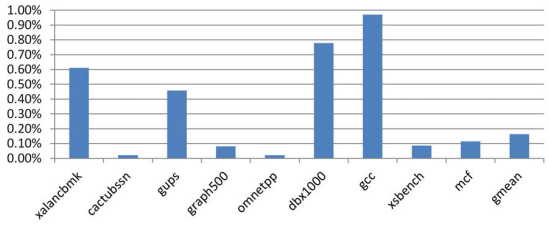

Fig. 17. Percentage of Total Execution Time Spent in System

We investigated TPS's actual runtime utilization of each page size across the benchmarks. Figure 18 shows the results. Each point on the line represents how many pages were in use by that application at the particular page size shown on the x-axis. As shown, each workload still utilizes nearly all available page sizes. All benchmarks tend to have higher counts of the relatively smaller page sizes because of the conservative page promotion policy. The relatively small total number of unique pages is what ultimately enables TPS to eliminate nearly all TLB misses, as shown in previous paragraphs.

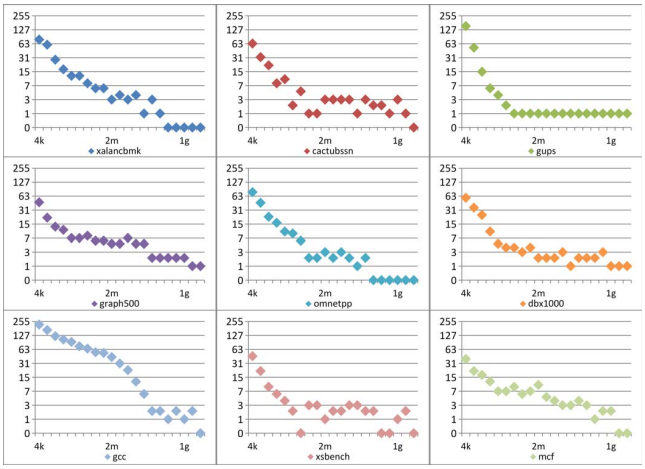

Fig. 18. TPS: Per Benchmark Page Size Counts

V. Related Work

The general area of virtual memory, address translation, and TLB design remains an active area of research. Prior work has shown that excessive page walks may significantly degrade performance in applications suffering from limited TLB reach [8], [13], [23], [31], [32], [34]. We have further shown that L1 TLB misses that still hit in the L2 TLB may cause non-trivial performance degradation.

Redundant Memory Mappings (RMM) [34], [35] is the most closely related work to TPS. RMM leverages arbitrary ranges of contiguous virtual pages that map to contiguous physical frames to provide an alternative range translation mechanism. In the operating system, RMM requires a Range Table to be maintained concurrently with the standard Page Table. Both the Range Tables and the Page Tables map the process virtual address space to physical frames. In hardware, a cache of Range Table Entries (or Range TLB; similar to how a TLB is a cache of Page Table Entries) provides an alternative mechanism for the hardware to translate a virtual address to a physical address. Each Range Table Entry (RTE) is similar to a segment descriptor, containing base, limit, offset, and protection information. When a translation misses in the L1 TLB, the L2 TLB and Range TLB are looked up in parallel because both are able to provide the necessary translation. If the Range TLB provides the translation, the PTE for the page can subsequently be constructed and installed in the L1 TLB. Because range translations do not have alignment or size restrictions, it is likely that RMM is more amenable to system external fragmentation. RMM introduces the software complexity of maintaining two trees in parallel to represent the same virtual-to-physical address space mappings (the Range Table Tree and the Page Table Tree). Thus, RMM additionally needs to deal with the existing page table fine-grained locking scheme to manage both trees concurrently [19].

Huge pages or superpages [16], [17], [42] are already utilized in current processors. The prevalent (e.g., Intel and ARM) approaches offer limited choices between conventional page sizes. The limitations of only supporting conventional, coarse-grained huge pages as opposed to offering tailored page sizes are discussed and evaluated throughout the paper.

Intel Itanium [15], [26] and SPARC [55] offer more page sizes, but these mechanisms require software-controlled TLB structures to pin TLB entries to improve performance. TPS is largely orthogonal to these approaches since it eliminates most page walks altogether. Software-controlled TLB approaches could still be used with TPS to accelerate page walks when they may be required. Itanium splits the address space into 8 regions, each with a configurable page size. This approach limits benefit despite the many page sizes offered.

Romer et al.'s work [49] in superpages considered adding more variability to available page sizes, but this approach only evaluates a page relocation based approach to merge and promote smaller pages into larger pages. Our frame reservation and eager allocation based approach reduces the need to perform extraneous memory copies to create large pages. In addition, this work only considers a software-managed TLB and does not describe the hardware changes necessary to support additional page sizes for hierarchical radix tree page tables.

Ingens [38] is a purely operating system proposal that significantly improves on Transparent Huge Pages [16] by offering cleaner tradeoffs between memory consumption, performance, and latency. HawkEye [43] is another OS technique that further improves upon Ingens. HawkEye balances fairness in huge page allocation across multiple processes, performs asynchronous page pre-zeroing, de-duplicates zero-filled pages, and performs fine-grained page access tracking and measurement of address translation overheads through hardware performance counters. Because TPS and the increased base page size provide improvements to the underlying hardware, our mechanisms

and techniques like Ingens/HawkEye could work cooperatively to improve these tradeoffs. TPS can additionally supply fine-grained metadata information about larger pages to the OS. By offering more choice in page sizes, our work opens the door to further interesting OS research like Ingens/HawkEye along this path.

Early commercial processors have used segmentation for address translation. Several processors provided support for segmentation without paging [25], [27], [39]. Other processors supported both segmentation and paging [31]. Unlike past segmentation approaches, TPS adheres to the page-based virtual memory paradigm, enabling its benefits. TPS still allows for segmentation on top of paging.

Direct segment [8] is a segmentation-like approach to address translation. It is as an alternative to page based virtual memory for big memory applications that can utilize a single, large translation entity. A hardware segment maps one contiguous range of virtual address space to contiguous physical memory. The remaining virtual address space is mapped to physical memory with the existing page-based virtual memory approach. A particular virtual addresses is translated to its physical address via either the hardware direct segment or the page table hardware and TLBs. Like standard segmentation, direct segment utilizes base, limit, and offset registers and does not require page walks within the segment. Unlike TPS, this mechanism requires that the application explicitly creates a direct segment during its startup. The OS must at that time be able to reserve a single large contiguous range of physical memory for the segment. Direct segment requires application changes and is only suited for large memory workloads that can leverage a single, large segment. DVMT [1] is a technique that decouples address translation from access permissions. However, DVMT also requires explicit application changes.

Sub-blocked TLBs [56], CoLT [46], and Clustered TLBs [45] combine near virtual-to-physical page translations into single TLB entries. These approaches rely on the default operating system memory allocators assigning small regions of contiguous or clustered physical frames to contiguous virtual pages. However, these approaches are limited to a small number (e.g., 16) of page translations per TLB entry. This hinders the generality of their applicability to data sets of any size, thus limiting their potential benefits.

Various techniques to accelerate page walks seek to reduce TLB miss cost, rather than reducing or eliminating TLB misses. MMU caches reduce page walk latency by caching higher levels of the page table, thereby skipping one or more memory accesses during the page walk process [6], [11], [28]. Currently available processors cache PTEs in the data cache hierarchy to reduce page walk latency on MMU cache misses [30]. The POM-TLB [50] caches TLB entries in memory to reduce the cost of page walks. TPS is orthogonal to these approaches since it eliminates most page walks altogether. These mechanisms can still be used with TPS to accelerate page walks when they may be required.

Address translation overhead can be lowered by reducing the TLB miss rate. Synergistic TLBs [54] and shared last-level TLBs [10], [12], [40] seek to reduce the number of page walks and improve TLB reach. Prior work has proposed hardware PTE prefetchers [13], [33], [52]. These approaches prefetch PTEs into the TLB before they are needed for translation. However, memory access pattern predictability limits TLB prefetcher effectiveness; for example, in applications with random access behavior, TLB prefetching will be unlikely to help. Other prior work has proposed speculative translation based on huge pages [7]. Like with TLB prefetching, this mechanism favors sequential patterns and relies on address contiguity. [44] proposed a prediction technique that allows a single set associative TLB to be shared by all page sizes. Other prior work has proposed gap-tolerant mechanisms that allow conventional superpage creation even when retired physical pages cause non-contiguity in available physical memory [21]. However, each TLB entry still only maps a single conventional page. This limits TLB reach for memory intensive applications. Unlike these approaches, TPS creates translations for page sizes tailored to the application's data set, caching them in the TLB. TPS can work together with these approaches to improve translation latency.

Prior work in fine-grained memory protection [24], [57], [58] identify similarity and contiguity across many conventional base pages, similar to how TPS identifies contiguity in order to tailor a page of appropriate size. However, these approaches only exploit the contiguity of fine-grained protection rights across these larger ranges, while TPS leverages and further enhances the address space contiguity during memory allocation and compaction to facilitate faster address translation by greatly improving TLB hit rates.

Prior work in virtual caches reduces translation overhead by translating only after a cache miss [9], [59]. However, for poor-locality workloads suffering from many TLB misses, virtual caches just shift the still-necessary translation to a higher level of the cache hierarchy while increasing system complexity in order to deal with the synonym problem. The translation penalty will still be incurred when physical addresses are actually needed, which TPS seeks to nearly eliminate.

VI. Conclusion

We have shown that current conventional page sizes and TLB limitations are insufficient to deliver scalable, high-performance virtual memory translation. We designed Tailored Page Sizes to allow support for pages of any power-of-two size larger or equal to the base page size. TPS requires small changes to hardware and small improvements to operating system software to, at no additional memory cost, significantly reduce L1 TLB misses and page walk memory references, and improve TLB reach.

Acknowledgment

We thank the members of the HPS Research Group and the anonymous reviewers for their valuable suggestions and feedback.

REFERENCES

[1] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-it-yourself virtual memory translation," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 457–468.

[2] *AMD64 Architecture Programmer's Manual*, AMD Corporation, March 2017.

[3] *Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors*, AMD Corporation, February 2020.

[4] *Apple Developer Guide: About the Virtual Memory System*, Apple Inc., May 2013. [Online]. Available: https://developer.apple.com/library/content/documentation/Performance/Conceptual/ManagingMemory/Articles/AboutMemory.html

[5] *ARM Cortex-A Series Programmer's Guide for ARMv8-A*, ARM Holdings, 2015.

[6] 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.

[7] ——, "Spectlb: A mechanism for speculative address translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, 2011, pp. 307–318.

[8] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 237–248.

[9] A. Basu, M. D. Hill, and M. M. Swift, "Reducing memory reference energy with opportunistic virtual caching," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12, 2012, pp. 297–308.

[10] S. Bharadwaj, G. Cox, T. Krishna, and A. Bhattacharjee, "Scalable distributed last-level tlbs using low-latency interconnects," in *Microarchitecture (MICRO), 2018 51st Annual IEEE/ACM International Symposium on*, 2018.

[11] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, 2013, pp. 383–394.

[12] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11, 2011, pp. 62–63.

[13] A. Bhattacharjee and M. Martonosi, "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '09, 2009, pp. 29–40.

[14] Z. Bodek, "Transparent superpages for freebsd on arm," 2014. [Online]. Available: https://www.bsdcan.org/2014/schedule/attachments/281_2014_arm_superpages-paper.pdf

[15] M. Chapman, I. Wienand, and G. Heiser, "Itanium page tables and tlb," 2003.

[16] J. Corbet, "Transparent hugepages," https://lwn.net/Articles/359158/, October 2009. [Online]. Available: https://lwn.net/Articles/359158/

[17] ——, "Huge page part 1 (introduction)," https://lwn.net/Articles/374424/, February 2010. [Online]. Available: https://lwn.net/Articles/359158/

[18] ——, "Memory compaction," https://lwn.net/Articles/368869/, January 2010. [Online]. Available: https://lwn.net/Articles/368869/

[19] ——, "Split pmd locks," https://lwn.net/Articles/568076/, September 2013. [Online]. Available: https://lwn.net/Articles/568076/

[20] P. J. Denning, "Virtual memory," *ACM Computing Surveys (CSUR)*, vol. 2, no. 3, pp. 153–189, 1970.

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

[22] J. Fotheringham, "Dynamic storage allocation in the atlas computer, including an automatic use of a backing store," *Communications of the ACM*, vol. 4, no. 10, pp. 435–436, 1961.

[23] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, 2014, pp. 178–189.

[24] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin, "A case for unlimited watchpoints," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, 2012, pp. 159–172.

[25] "Intel 8086," http://en.wikipedia.org/wiki/Intel_8086.

[26] "Intel® itanium® architecture developer's manual, vol. 2," http://www.intel.com/content/www/us/en/processors/itanium/itanium-architecture-software-developer-rev-2-3-vol-2-manual.html.

[27] *Introduction to the iAPX 432 Architecture*, Intel Corporation, 1981.

[28] *TLBs, Paging-Structure Caches and their Invalidation*, Intel Corporation, 2008.

[29] *5-Level Paging and 5-Level EPT*, Intel Corporation, May 2017.

[30] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, April 2018.

[31] B. Jacob and T. Mudge, "Virtual memory in contemporary microprocessors," *IEEE Micro*, vol. 18, no. 4, pp. 60–75, Jul. 1998.

[32] ——, "Performance analysis of the memory management unit under scale-out workloads," in *Proceedings of the 2014 IEEE International Symposium on Workload Characterization*, 2014, pp. 1–12.

[33] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: An application-driven study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02, 2002, pp. 195–206.

[34] 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 *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 66–78.

[35] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Energy-efficient address translation," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 631–643.

[36] T. Kilburn, D. B. Edwards, M. J. Lanigan, and F. H. Sumner, "One-level storage system," *IRE Transactions on Electronic Computers*, no. 2, pp. 223–235, 1962.

[37] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server engineering insights for large-scale online services," *IEEE micro*, vol. 30, no. 4, pp. 8–19, 2010.

[38] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens." in *OSDI*, 2016, pp. 705–721.

[39] W. Lonergan and P. King, "Design of the b 5000 system," *Datamation*, vol. 7, no. 5, May 1961.

[40] D. Lustig, A. Bhattacharjee, and M. Martonosi, "Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 1, p. 2, 2013.

[41] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[42] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002, pp. 89–104.

[43] 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*. ACM, 2019, pp. 347–360.

[44] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly tlb designs," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 210–222.

[45] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing tlb reach by exploiting clustering in page translations," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 558–567.

[46] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*. IEEE, 2012, pp. 258–269.

[47] S. Phillips, "M7: Next generation sparc," in *Hot Chips 26 Symposium (HCS), 2014 IEEE*. IEEE, 2014, pp. 1–27.

[48] *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*, RISC-V Foundation, June 2019.

[49] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing tlb and memory overhead using online superpage promotion," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 176–187.

[50] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 469–480.

[51] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 475–486. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485963

[52] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based tlb preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00, 2000, pp. 117–127.

[53] A. Seznec, "Concurrent support of multiple page sizes on a skewed associative tlb," *IEEE Transactions on Computers*, vol. 53, no. 7, pp. 924–927, 2004.

[54] S. Srikantaiah and M. Kandemir, "Synergistic tlbs for high performance address translation in chip multiprocessors," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2010, pp. 313–324.

[55] *UltraSPARC T2 Supplement to the UltraSPARC Architecture*, Sun Microsystems, 2007.

[56] M. Talluri and M. D. Hill, "Surpassing the tlb performance of superpages with less operating system support," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI, 1994, pp. 171–182.

[57] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood, "A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41, 2008, pp. 94–105.

[58] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X, 2002, pp. 304–316.

[59] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton, "An in-cache address translation mechanism," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86, 1986, pp. 358–365.

[60] 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*, 2019, pp. 698–710.