The Impact of Page Size and Microarchitecture on Instruction Address Translation Overhead

YUFENG ZHOU and ALAN L. COX, Rice University
SANDHYA DWARKADAS and XIAOWAN DONG, University of Rochester

As the volume of data processed by applications has increased, considerable attention has been paid to data address translation overheads, leading to the widespread use of larger page sizes ("superpages") and multi-level translation lookaside buffers (TLBs). However, far less attention has been paid to instruction address translation and its relation to TLB and pipeline structure. In prior work, we quantified the impact of using code superpages on a variety of widely used applications, ranging from compilers to web user-interface frameworks, and the impact of sharing page table pages for executables and shared libraries. Within this article, we augment those results by first uncovering the effects that microarchitectural differences between Intel Skylake and AMD Zen+, particularly their different TLB organizations, have on instruction address translation overhead. This analysis provides some key insights into the microarchitectural design decisions that impact the cost of instruction address translation. First, a lower-level (level 2) TLB that has both instruction and data mappings competing for space within the same structure allows better overall performance and utilization when using code superpages. Code superpages not only reduce instruction address translation overhead but also indirectly reduce data address translation overhead. In fact, for a few applications, the use of just a few code superpages has a larger impact on overall performance than the use of a much larger number of data superpages. Second, a level 1 (L1) TLB with separate structures for different page sizes may require careful tuning of the superpage promotion policy for code, and a correspondingly suboptimal utilization of the level 2 TLB. In particular, increasing the number of superpages when the size of the L1 superpage structure is small may result in more L1 TLB misses for some applications. Moreover, on some microarchitectures, the cost of these misses can be highly variable, because replacement is delayed until all of the in-flight instructions mapped by the victim entry are retired. Hence, more superpage promotions can result in a performance regression. Finally, our findings also make a case for first-class OS support for superpages on ordinary files containing executables and shared libraries, as well as a more aggressive superpage policy for code.

CCS Concepts: • **Software and its engineering** → **Operating systems**; **Memory management**; **Virtual memory**; • **Computer systems organization** → *Serial architectures;*

Additional Key Words and Phrases: Superpage, Translation Lookaside Buffer, microarchitecture, instruction access, page table, memory hierarchy, address translation

This article is an extension of a conference paper. The article extends our ISPASS 2019 [55] paper by contrasting the behavior of two different microarchitectures to provide insight into the impact of microarchitecture design decisions coupled with page size on the behavior of instruction address translation.

This work was funded in part by National Science Foundation (NSF) Awards CNS-1319353, CNS-1618497, CNS-1618588, CNS-1900803, and CNS-2008857.

Authors' addresses: Y. Zhou and A. L. Cox, Department of Computer Science, Rice University; emails: {yufengz, alc}@rice.edu; S. Dwarkadas and X. Dong, Department of Computer Science, University of Rochester; emails: {sandhya, xdong}@cs.rochester.edu.

Authors. current addresses: S. Dwarkadas, University of Virginia; X. Dong has graduated. Her current email address is xiaowand@google.com.

1544-3566/2023/07-ART38 $15.00
https://doi.org/10.1145/3600089

ACM Reference format:
Yufeng Zhou, Alan L. Cox, Sandhya Dwarkadas, and Xiaowan Dong. 2023. The Impact of Page Size and Microarchitecture on Instruction Address Translation Overhead. *ACM Trans. Arch. Code Optim.* 20, 3, Article 38 (July 2023), 25 pages.
https://doi.org/10.1145/3600089

1 INTRODUCTION

With the growing importance of big data workloads, many researchers have sought to reduce the overhead of virtual-to-physical address translation during data accesses. Their work has proposed both architectural changes to the way that address translation is performed in hardware [29, 33, 38, 50–52] as well as improvements to the automatic support for larger page sizes ("superpages") in **Operating Systems (OSs)** [26, 40, 56]. However, much less attention has been paid to the performance impact of address translation on instruction accesses, at either the architectural or OS level [32, 55].

Address translation overhead for instruction accesses is exacerbated by two application trends: (1) their increasing size and complexity and (2) their reliance on shared libraries. For example, the Clang compiler increased in size from 31MB of x86-64 machine code in version 3.0 (2012) to 56MB in version 6.0.0 (2018),and a recent version of the Node.js runtime environment uses 20shared libraries.In previous work [55], we showed that even on modern processors, with their ever larger **Instruction Translation Lookaside Buffers (ITLBs)**, a variety of widely used applications, ranging from compilers to web user-interface frameworks, suffer from high instruction address translation overheads, which directly lead to performance degradations. We also evaluated the impact on performance of the use of automatic OS-level superpage promotion and page table sharing for code pages, along with transparent padding of a program's main executable out to a full superpage.

In this article, we extend our previous results [55] to provide an in-depth analysis of the impact of the address translation architecture (hardware and software) on the overhead of instruction address translation using two different hardware architectures and a combination of a few different OS techniques at the software level.

At the hardware level, we quantify the effects of different **Translation Lookaside Buffer (TLB)** organizations on address translation overhead using Intel and AMD processors. Recent Intel and AMD processors differ in their TLB organizations along two dimensions: (1) whether there are separate structures for data versus instruction mappings, and (2) whether there are separate structures for different page size mappings. Intel processors have a split design in the **Level 1 (L1)** ITLB on the second of these dimensions (page size), whereas AMD processors have a split design in the **Level 2 (L2)** TLB on the first of these dimensions (data versus instructions). We use the ability to collect information on the processor stall cycles due to instruction address translation that is available on modern Intel processors to analyze the overhead. To the best of our knowledge, this is the first journal article in which this end-to-end counter that captures all forms of delay due to instruction address translation is employed for analysis.

At the software level, we evaluate the benefits of a new in-kernel mechanism that transparently pads *all* executable regions, including shared libraries, out to a superpage boundary. Generally, the size of an executable region is not an integer multiple of the superpage size. Consequently, without padding, executable regions generally have *residual* code at the end that is mapped using 4KB pages. In the worst case, on an Intel Skylake processor, despite a large L2 TLB shared between data and instruction mappings, mappings for this residual code can occupy almost one-third of the entries in the L2 **Shared Translation Lookaside Buffer (STLB)**, displacing many mappings

to data. Our *padding* mechanism addresses this problem while minimizing the amount of added NOP-filled physical memory that must be allocated.

Our findings are as follows:

- Instruction address translation overhead can be large, up to 13.44% of execution cycles, on modern processors regardless of their microarchitecture.
- Page walk cycles for instructions can be higher than for data.
- The performance benefits of using superpages on code can be larger than on data, making a case for first-class OS support for superpages on ordinary executable files and shared libraries, as well as a potentially more aggressive superpage policy for code.
- We show that an aggressive code superpage promotion policy can improve the performance of applications by up to 8.33% compared to no code superpages. However, when the L1 TLB uses separate structures for different page sizes, tuning the aggressiveness of the policy can be more difficult.
- We quantify the impact of the aggressive use of code superpages on time spent within the OS on behalf of the application. In one case, OS-space execution cycles are reduced by 22%.
- We demonstrate that padding applied to residual code, including within shared libraries, can further improve performance by up to 8% over the most aggressive code superpage policy without padding, despite the potential for a larger memory footprint.
- Prior work [28, 39, 44] has used page table walks as a proxy for the cost of address translation. Using the counter for instruction address translation stall cycles on modern Intel processors, we show that pipeline stalls add significantly to the instruction address translation overhead beyond the cost of instruction page walk cycles when the number of superpage entries is limited.
- Using code superpages can result in underutilization of the ITLB when using separate L2 TLBs for instructions and data. With an L2 TLB shared between instructions and data, using code superpages can reduce data page walk cycles more than using data superpages.

2 BACKGROUND

With increasing data and code footprints, using superpages can reduce address translation overhead by increasing TLB coverage and reducing the number of memory accesses that must be performed on a TLB miss to walk the page table. Consequently, modern processors support superpages of multiple sizes (e.g., 2MB and 1GB on x86 processors), and their TLBs provide a growing number of entries for storing these superpage mappings [3, 8]. Their microarchitectures differ, however, in a number of dimensions. In particular, Intel and AMD have made different choices in whether their multi-level TLBs have separate lower-level structures for instruction versus data mappings, as well as in whether separate structures are used for different-sized mappings at L1. In recent generations, Intel has supported separate structures for 4KB versus 2MB mappings in the L1 ITLB. In the case of the Skylake microarchitecture (Table 1), only 8 entries for 2MB superpages are supported (per hyperthread) in the L1 ITLB. In contrast, AMD Zen+ (see Table 1) provides a unified structure of 64 entries for 4KB and 2MB mappings in the L1 ITLB. At L2, Intel has a single TLB structure that is shared across instructions and data, whereas AMD's L2 TLB is split into two structures for instructions versus data. We evaluate the impact of these microarchitectural choices by analyzing performance on both.[1]

[1]Both AMD and Intel have continued to make the same choices in their more recent microarchitectures, such as Zen 4 and Golden Cove. However, the number of entries in Golden Cove's L1 ITLB has doubled for both 4KB and 2MB mappings, and the number of entries in the L2 STLB has increased by a third. Although all of the STLB entries can cache 4KB mappings, just half of the entries can cache 2MB mappings; the other half can cache 1GB mappings. Zen 4 has larger data TLBs, but its ITLBs are unchanged.

OSs also differ in their support for superpages. Linux does not transparently support superpages on code that is demand-paged from a regular disk-based file system. To map code from files with superpages, Linux requires either (1) that the user copy the executable and shared libraries to a special huge page file system [43] or (2) that the user process first copies its code from the original virtual memory region backed by 4KB pages to superpage-backed anonymous memory (created by an `madvise`(MADV_HUGEPAGE) call), and then remaps the superpage-backed memory to the original virtual memory region using the `mremap` system call. The second approach incurs significant overhead because "copy-and-remap" has to be repeated every time an application is started. Only **Just-in-Time (JIT)** compiled code that is written to anonymous virtual memory can enjoy the benefits of superpages automatically.

In contrast, FreeBSD supports automatic superpage promotion for code from any file system. This avoids the overhead of the copy-and-remap approach. FreeBSD uses a reservation-based allocator to support superpages transparently [47]. When an application faults in a (virtually) superpage-aligned region for the first time, the page fault handler reserves contiguous physical memory (reservations) but does not map the entire reservation immediately. The page fault handler then allocates base pages from the reservation on subsequent page faults in the region, bringing in a 64KB-aligned cluster of pages to improve I/O performance under the assumption of spatial locality. When the reservation becomes fully populated, the system performs a *promotion*, replacing the entire leaf-level page table page with a single superpage mapping. However, reservations that have not been fully populated can be broken if ever there is a shortage of free physical memory. For other processes that access the same code in the future, as long as all constituent pages in a reservation remain resident, the kernel bypasses incremental promotion and immediately creates superpage mappings upon (soft) page faults into the region. Due to the advantages of FreeBSD's support for automatic promotion, we use FreeBSD as the baseline OS for our evaluation.

3 METHODOLOGY

3.1 Experimental Setup

Our Intel-based system uses a Xeon E3-1240 v5 (Skylake) quad-core processor with 8MB of **Last-Level Cache (LLC)**, and our AMD-based system uses a Ryzen 7 2700X (Zen+) octa-core processor with 16MB of LLC. Although the total LLC size on the AMD processor is larger, its implementation is evenly split between two core complexes, and the four cores within a core complex can only spill to the local half of the LLC [18]. Unless otherwise noted, we pin workloads to just one core complex, so we are effectively using the same number of cores and the same amount of LLC on both the AMD and Intel processors. Both processors support two hyperthreads (SMT threads) per core, and all hyperthreads are enabled on both systems. Table 1 describes both processors' TLB organizations. Both systems have 32GB of RAM. The OS is FreeBSD 11.2-RELEASE. Both OS-controlled frequency downscaling and hardware-controlled frequency upscaling are disabled. Thus, the frequency is fixed on the Intel system to 3.5GHz and on the AMD system to 4.0GHz.

We collect information from hardware performance-monitoring counters [15, 16, 21] (Table 2) using the `pmcstat` [25] utility. Worth noting is that ICACHE_64B.IFTAG_STALL is a counter introduced in Skylake that measures the end-to-end cost of ITLB misses [4, 20]. We measure user and kernel space separately. Unless otherwise stated, the numbers presented are for user space, because by default kernel code is already mapped with superpages to the fullest extent possible. We report the minimum of three runs for all benchmarks except Javac and Derby. For Javac and Derby, we use 10 runs due to their larger variance. Unless otherwise noted, we use all of the available hyperthreads for each application evaluated. When we use only a subset of the available hyperthreads, we assign dedicated ones to applications with the `cpuset` [19] utility and monitor only the activity of those hyperthreads. This means that for the server-oriented applications, we collect counters

Table 1. Skylake and Zen+ TLB Structures

Intel Skylake		
L1 ITLB		
4KB	128 entries	8-way set associative
2MB	8 entries per thread	fully associative
L1 DTLB		
4KB	64 entries	4-way set associative
2MB	32 entries	4-way set associative
1GB	4 entries	fully associative
L2 STLB		
4KB + 2MB	1,536 entries	12-way set associative
1GB	16 entries	4-way set associative

AMD Zen+		
L0 ITLB		
All sizes	8 entries	fully associative
L1 ITLB		
All sizes	64 entries	fully associative
L2 ITLB		
4KB + 2MB	512 entries	8-way set associative
L1 DTLB		
All sizes	64 entries	fully associative
L2 DTLB		
4KB + 2MB	1,536 entries	12-way set associative

The Skylake and Zen+ microarchitectures support 4KB, 2MB, and 1GB page mappings in the TLB. Skylake has separate L1 TLB structures for 4KB versus 2MB page mappings. However, in the L2 STLB, 4KB and 2MB page mappings reside in the same structure. In contrast, all of Zen+'s TLB structures support page mappings of different sizes within the same structure. DTLB, data TLB.

Table 2. Hardware Performance Counters and Their Interpretations

Descriptions	Intel Counters & Equations	AMD Counters & Equations
No. instructions retired	INST_RETIRED.ANY_P	EX_RET_INST
Execution cycles	CPU_CLK_UNHALTED.THREAD_P	LS_NOT_HALTED_CYCLE
Inst addr translation cycles (ITLB stall)	ICACHE_64B.IFTAG_STALL	N/A
Inst addr translation overhead	ICACHE_64B.IFTAG_STALL / CPU_CLK_UNHALTED.THREAD_P	N/A
Inst page table walk cycles	ITLB_MISSES.WALK_PENDING	N/A
No. of inst page table walks	ITLB_MISSES.WALK_COMPLETED	N/A
Data page table walk cycles	DTLB_LOAD_MISSES.WALK_PENDING + DTLB_STORE_MISSES.WALK_PENDING	N/A
No. data page table walks	DTLB_LOAD_MISSES.WALK_COMPLETED + DTLB_STORE_MISSES.WALK_COMPLETED	N/A
LLC stall cycles	CYCLE_ACTIVITY.STALLS_L3_MISS	N/A

only on hyperthreads running the server processes. For all benchmarks, we perform warmup runs before taking any measurements.

3.2 Workloads

Our workloads are based on seven widely used applications that all have large code sizes, covering a range of application types, including two compilers (Clang and Javac), three database applications (PostgreSQL, MySQL, and Derby), two language runtimes with JIT compilation capabilities (Java and Javascript runtimes), a web application framework (Node.js), and a machine emulator performing JIT-compilation-like binary translation (QEMU). The applications differ in whether they execute statically compiled code or JIT-compiled code, and whether they are statically linked or dynamically linked. We also chose multiple database management systems to represent different points in the database design spectrum. PostgreSQL has a multi-process concurrency model, whereas MySQL is multi-threaded. Both are written in C. Derby is a multi-threaded database application written in Java.

Table 3 lists the main executable size, the number of linked shared libraries, whether the workload generates JIT-compiled code, and the concurrency model of each workload. Main executables and shared libraries are demand-paged from ordinary files. JIT-compiled code is generated at runtime and written into anonymous memory. The main executables range in size from 5.953MB to as large as 55.895MB, all of which are large enough for at least two code superpages. All applications

Table 3. Workload Characteristics

	Main Executable Size (MB)	No. Shared Libraries Linked	No. Shared Libraries <1MB	No. Shared Libraries <2MB	Large Shared Library Size (MB)	JIT-Compiled Code in Anon Mem	Multi-Thread	Multi-Process
Clang	55.895	0	0	0	N/A	N	N	Y
PostgreSQL	5.953	15	10	14	1.53–2.88	N	N	Y
Javac	20.000	13	11	12	1.59 and 11.22	Y	Y	N
Derby	12.000	15	13	14	1.59 and 11.22	Y	Y	N
Node.js	23.836	20	16	18	1.59–2.88	Y	N	N
MySQL	40.094	15	13	14	1.59 and 2.27	N	Y	N
QEMU	12.359	48	41	47	1.59 and 2.27	Y	Y	Y

OpenJDK 8's main executable consists of a single 4KB page of code that hands off control to a large 11.220MB shared library, `libjvm.so`, which actually implements the Java virtual machine. For Javac and Derby, the main executable refers to JIT-compiled code rather than the main executable or `libjvm.so`.

except for Clang link a considerable number of shared libraries. The applications that do rely on shared libraries each link at least 13, with QEMU using as many as 48 libraries. We observe that most shared libraries are too small (<2MB) for superpages. There are, however, a few exceptions, such as `libcrypto.so.8` (2.27MB) linked by `PostgreSQL`, `Node.js`, `MySQL`, and `QEMU`. In fact, most libraries are smaller than 1MB. In terms of concurrency models, we note that some applications, such as PostgreSQL, are inherently multi-process, where the server consists of six main processes plus one worker process per client. In other cases, we run multiple instances of the application. Moreover, we ensure that all of the applications are run with the same starting conditions of a freshly booted machine.

Clang. Clang is a C/C++ compiler with a built-in assembler based on the LLVM infrastructure. We run Clang version 6.0 that comes with the FreeBSD 11.2 system. We run one instance of Clang per hyperthread, each compiling the source code for the Dhrystone [1] benchmark, to simulate the parallel compilation of a large application consisting of many source files. The compilation options are `-Wno-everything -O2 -c`.

PostgreSQL. PostgreSQL is an object-relational database system [13]. We run version 9.6.8 of PostgreSQL and use pgbench [12] to perform transactions that are loosely based on TPC-B. We run pgbench on a separate machine that is connected to the PostgreSQL machine under test with a dedicated 10Gbps Ethernet link. A fixed number of back-to-back transactions are performed on a 5GB database, which easily fits in memory while still exceeding the TLB's maximum coverage using 2MB superpages. We set the `-C` option of pgbench to use one persistent connection per client. We run six worker processes per physical core to keep the processor fully busy on the server side, on both the Skylake and the more powerful Zen+ cores.

Javac and Derby. We use OpenJDK 8 [14], which is a Java runtime supporting JIT compilation, to run Compiler.compiler and Derby from the SPECjvm2008 benchmark suite [10, 11]. Compiler.compiler compiles a set of `.java` files using the `javac` compiler. We refer to the benchmark as "Javac" for simplicity. Derby is an open source in-memory database written in Java. We use the `specjvm.hardware.threads.override` option to allow the benchmarks to scale appropriately with the number of cores allocated. For each benchmark, we run a fixed number of operations.

Node.js. Node.js is a JavaScript runtime built on the V8 JavaScript engine [6]. We run version 8.11.1 of Node.js and use the React server-side rendering benchmark [17]. By default, this benchmark tries to run for a fixed amount of time, varying the number of iterations. We instead fix the number of iterations to achieve roughly the same duration as the benchmark would by default. This ensures that the same fixed amount of work across runs. We run one Node.js process per hyperthread, which is typical of a cluster configuration of Node.js that exploits multi-core systems.

MySQL. MySQL is a relational database system [23]. We run version 8.0.2 of MySQL and use the read-write OLTP test of sysbench [24] to test the MySQL server. We run sysbench on a separate machine that is connected to the MySQL machine under test with a dedicated 10Gbps Ethernet link. We run seven worker threads per physical core within the server process to keep the processors fully busy without incurring a significant increase in queueing delay. We perform 1,600,000 back-to-back transactions in total on a 5GB database, which easily fits in memory while still exceeding the TLB's maximum coverage using 2MB superpages.

QEMU. QEMU is an emulator that supports running programs compiled for a different architecture [7]. We use QEMU version 4.2.1 to run a full FreeBSD 13.0 system compiled for AArch64. Within an emulated system, we run one instance of the bundled Clang compiler per hyperthread, each compiling the source code of SQLite version 3.34.0 [9] a fixed number of times. To avoid the multiprocessor scaling limitations of QEMU's virtual CPU emulation [30], we run one instance of QEMU per physical core, limiting each instance/emulated system to run only two virtual CPUs on the two hyperthreads that share a physical core. Since code targeted at AArch64 does not run natively on our x86 processor, QEMU performs dynamic binary translation, which generates a large amount of code that is stored in anonymous virtual memory. For the remainder of the article, we will refer to this code as JIT-compiled code.

4 QUANTIFYING INSTRUCTION ADDRESS TRANSLATION OVERHEAD

The ICACHE_64B.IFTAG_STALL counter measures all ITLB-related stalls, including cycles waiting for STLB hits, STLB misses, and instruction page (table) walks. Figure 1 presents the end-to-end instruction address translation overhead using this counter on our Intel-based system, as well as cycles spent on instruction page walks alone (as reported by the ITLB_MISSES.WALK_PENDING counter). Both the ICACHE_64B.IFTAG_STALL and ITLB_MISSES.WALK_PENDING counters are presented as a percentage of the overall user-space unhalted execution cycles on a modified kernel where code superpages are disabled. Unhalted cycles encompass not only cycles in which instructions are moving through the pipeline but also cycles in which the pipeline is stalled for reasons including but not limited to TLB and cache misses. We use the term *unhalted cycles* interchangeably with *execution cycles* in this article. The figure presents data from two different kernels: one stock FreeBSD and the other a modified kernel that does not create superpage mappings for code (resulting in only 4KB code mappings). Data superpages are allowed to exist as usual in both kernels.

To put the results of Figure 1 in perspective, Intel's VTune profiler reports instruction address translation as a performance problem when instruction address translation stall cycles exceed 5% of the execution cycles [15]. Six of the seven workloads here exceed this 5% threshold. Moreover, when code superpages are not used, three of the seven exceed 10%. At nearly 14%, PostgreSQL suffers the largest overhead.

FreeBSD's automatic code superpage promotion does benefit all applications except PostgreSQL. In the Java applications (Javac and Derby) and QEMU, the JIT-compiled code regions grow sequentially, and so the superpage-sized JIT-compiled code regions are naturally promoted into superpages. As a result, we see performance improvements in all three workloads. In contrast, most file-backed superpage-sized code regions are not promoted. Consequently, some workloads with only file-backed code, such as Clang, receive little performance improvement. This is because executable files contain code that may not be needed by a particular workload, and therefore that code is not demand-paged into memory. Under FreeBSD's conservative superpage promotion policy, such non-resident code disqualifies a superpage-sized region in the main executable or shared libraries from automatic promotion into a superpage. In fact, MySQL is the only exception where five code superpages are created in the main executable by stock FreeBSD. The rest of the

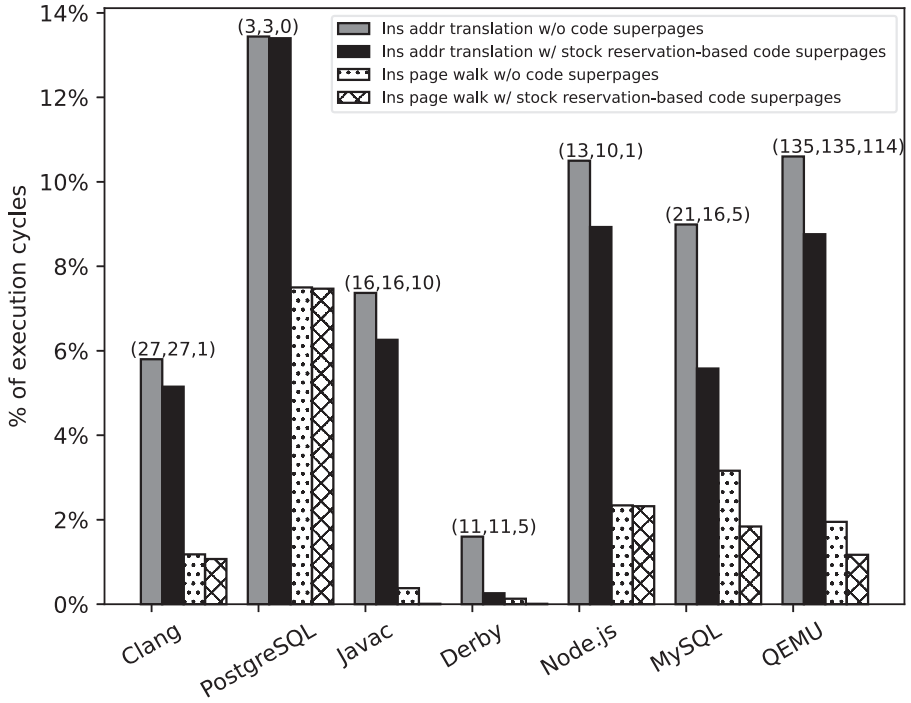

Fig. 1. Percentage of w/o code superpages execution cycles performing instruction address translation or instruction page table walks in user space on the Intel processor. Each group of bars is annotated with a tuple (x,y,z), where x is the total number of superpage-sized code regions (including the main executable, libraries, and JIT-compiled code) in the application, y is the number of *touched* superpage-sized code regions, and z is the number of *promoted* code superpages under stock FreeBSD.

applications have either no code superpages (PostgreSQL) or one code superpage (in the case of Node.js and Clang) created for the main executable under stock FreeBSD. We explore more aggressive superpage promotion policies for file-backed code in Section 5.

FreeBSD's automatic code superpage promotion cannot help with sub-superpage-sized code regions. For instance, although the total amount of JIT-compiled code in the Node.js workload is 4036KB, none of the JIT-compiled code regions are superpage-sized, and therefore none are superpage-mapped. This is because Node.js's JIT compiler creates 512KB code regions that are randomly placed throughout the address space. However, simply increasing the granularity of these code regions from 512KB to 2MB would still not allow for superpages, because Node.js co-locates a small amount of read-write metadata with read-only JIT-compiled code in each region. In the case of file-backed code, sub-superpage-sized code regions generally come from shared libraries and the end of the main executable. Such *residual* code regions are likewise mapped using only 4KB pages and range between a few kilobytes and nearly 2MB, as can be seen by taking the sizes in Table 3 modulo 2MB. Nonetheless, their potential impact on TLB contention is concerning. As Table 3 shows, main executable size is generally not an integer multiple of the superpage size. In the worst case, such as PostgreSQL, the residual code region of the main executable is nearly a whole superpage in size. On a Skylake processor, despite a large L2 TLB shared between data and instructions, mappings for this residual code region can occupy almost one-third of the entries in the STLB, displacing many mappings to data. Table 3 also shows that all applications, except for Clang, use 13 or more shared libraries, and very few of these libraries can be mapped using

superpages because they are too small. Nonetheless, the mappings for even a moderately sized shared library, such as the standard C library, could occupy more than a quarter of the entries in the STLB. Moreover, when taken together, the mappings for these small and moderately sized shared libraries could exceed the capacity of the STLB. We explore automatic *padding* as a solution to residual code in Section 6.

Finally, Figure 1 also clearly shows that instruction address translation cycles are often significantly larger than page walk cycles for instruction access. PostgreSQL is the only workload for which page walk cycles constitute more than half of the address translation overhead. At the other extreme, when code superpages are disabled, Clang's instruction address translation cycles are 4,9X the instruction page walk cycles. In general, in all applications except PostgreSQL, page walk cycles are less than one-third of the instruction address translation stall cycles. This observation suggests that, contrary to conventional wisdom [28, 39, 44], page walk cycles should not be used as an approximation to the instruction address translation overhead. We explore this topic in more detail in Section 7.

5 AGGRESSIVE SUPERPAGE PROMOTION FOR FILE-BACKED INSTRUCTIONS

FreeBSD's superpage promotion policy requires that a superpage reservation be fully populated before promotion is performed (see Section 2). More aggressive promotion policies for file-backed code regions trade additional physical memory consumption and I/O operations for reduced address translation overhead. We introduce an occupancy threshold to superpage reservations that are file-backed, which when exceeded results in the kernel *automatically* paging in the missing pages and performing a promotion. Specifically, the kernel keeps track of the number of resident pages in each file-backed reservation and checks the occupancy when handling a *hard* page fault (i.e., when I/O must be performed to fill the physical page). If the hard page fault handling results in the number of resident pages crossing the threshold, the remaining non-resident pages in the reservation are also loaded, and a superpage promotion is performed. We explore the impact of this more aggressive code superpage promotion policy by varying the threshold.

By default, on a hard page fault, FreeBSD loads a 64KB-aligned cluster of data and/or instructions into memory from the file that holds the executable or shared library and maps the underlying pages. Any 2MB superpage-sized region of code contains 32 such clusters, and stock FreeBSD conservatively requires all 32 clusters to be resident before promotion to a superpage. Figure 2 provides a cumulative count of the number of superpage-sized and aligned code regions of file-backed main executable and library code that contain less than or equal to zero to thirty-two 64KB clusters that are accessed by each workload. We use the same line for Javac and Derby since their cumulative counts are the same.

As the figure shows, a small relaxation of stock FreeBSD's residency requirement would result in a significant increase in code superpage mappings without a significant increase in memory consumption or I/O operations. In fact, PostgreSQL requires only a threshold of 28, where 7 more clusters, totaling 448KB of code, would be brought into memory for both of its reservations on the main executable to be promoted. (The third and final promotion for PostgreSQL at a threshold of 1 would be on code from a different file, libcrypto.) Moreover, once we lower the threshold to 16, we would have all reservations that are backed by the main executable file promoted for three additional applications. They are Node.js, Javac, and Derby. In contrast, Clang would still have 6 out of 27 reservations not promoted. Note that in practice, other applications that use a shared library and that access a different part of the library can have a synergistic effect on the promotion of the library's superpage-sized regions. For instance, even though MySQL only touches 29 clusters of libcrypto, promotion of libcrypto occurs at a threshold of 31, because other applications have already touched two of the clusters that are never touched by MySQL.

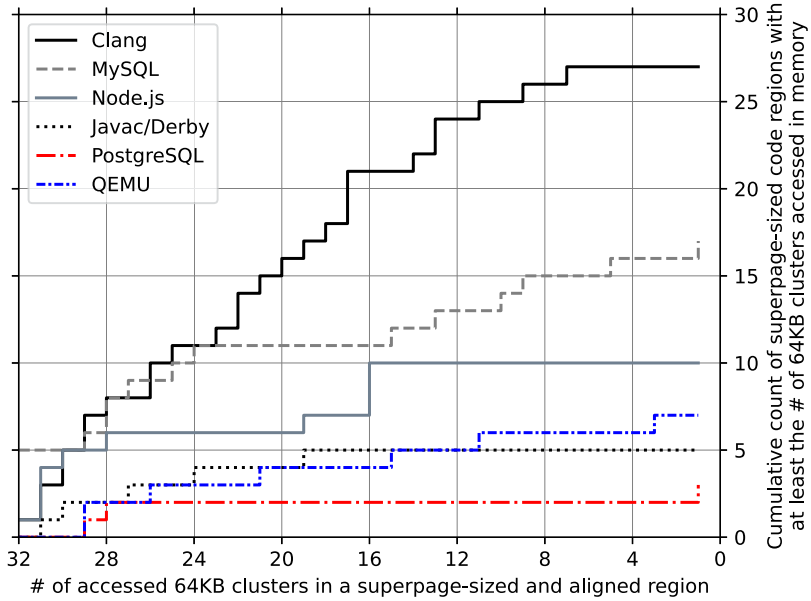

Fig. 2. Cumulative count of the number of superpage-sized and aligned regions (*Y* axis) of file-backed main and library code with at least the number of 64KB clusters (*X* axis) accessed.

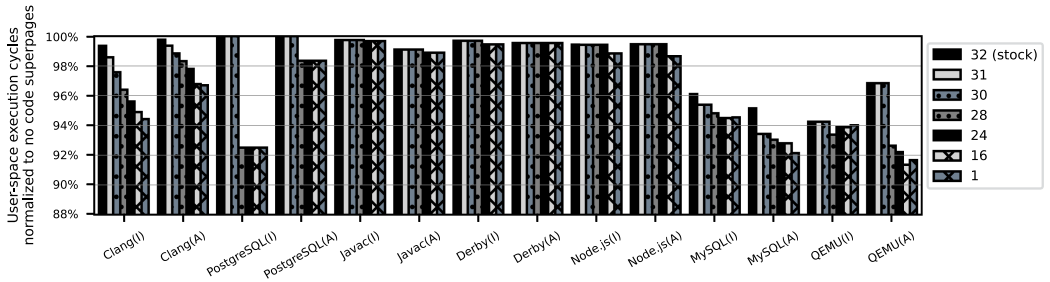

Fig. 3. Normalized user-space execution cycles under different promotion thresholds. Legend shows the number of 64KB clusters that are required to be physically resident before the corresponding superpage-sized region is promoted. *(I)* indicates that the workload was run on the Intel system, and *(A)* indicates the AMD system.

In Figure 3, we show the normalized user-space execution cycles at seven different thresholds for both the Intel and AMD systems. A threshold of 32 represents the stock FreeBSD superpage policy, which is the most conservative. In contrast, a threshold of 1 represents the most aggressive superpage promotion policy where any superpage-sized region that is touched is immediately loaded in full and promoted into a superpage. In general, we find that a more aggressive promotion policy, and the resulting increase in the number of code superpages, leads to fewer unhalted cycles. Although the most aggressive policy is not always optimal, specifically for QEMU, it never leads to more unhalted cycles than stock FreeBSD or not using code superpages.

Among the applications, only MySQL and QEMU achieve substantial reductions in execution cycles with the stock superpage policy. Nonetheless, both applications still see further gains with a more aggressive superpage policy for code. The size of MySQL's file-backed code, when converted

to superpages, exceeds the L1 ITLB capacity for superpage mappings on the Intel processor but not on the AMD processor. Consequently, the AMD processor sees an even greater reduction in user-space execution cycles when an aggressive code superpage policy is applied. In contrast, QEMU sees performance benefits on the stock superpage policy because a substantial fraction of the JIT-compiled code regions, where the majority of execution time is spent, are already superpage-mapped. Nonetheless, QEMU(A) sees substantial performance gains at threshold 28 due to the two additional superpages created (relative to the superpages that existed at threshold 30; see Figure 2). Based on page access traces, these two superpages eliminate the use of 575 different 4KB mappings, thereby substantially reducing pressure on the 512-entry L2 ITLB. Furthermore, QEMU(A) sees additional performance gains at threshold 24 and again at threshold 16. In contrast to QEMU(A), the optimal threshold for QEMU(I) is 28. Further reductions in the threshold, although still better than the stock policy, lead to a small 0.63% performance regression compared to threshold 28. In this case, converting less frequently used code regions into superpages increases competition for the limited superpage capacity of the L1 ITLB, displacing frequently used mappings for JIT-compiled code superpages. For QEMU(A), it is unclear why a similar regression from the optimal threshold of 16 happens at the most aggressive threshold 1. On the AMD processor, there are not separate TLB structures for different page sizes. An entry in either the L1 or the L2 ITLB can hold either a 2MB or 4KB mapping.

6 AUTOMATICALLY PADDING RESIDUAL CODE

Mappings for residual code regions often exceed the size of the last-level TLB (see Section 4). We deal with residual code regions using an automatic in-kernel *padding* mechanism that leverages the standard layout of ELF executable files [2] to convert the residual code region into a superpage.

At runtime, an ELF executable file's program header table is read by the OS to load the file's contents into the address space of a process. This table defines a set of memory segments, where each segment consists of adjacent sections within the file, such as code and read-only data, that are to be mapped together within the same region of the address space with the same access permissions. However, the segments are not required to start or end on a page boundary within the file or the address space. Consequently, a physical page caching a portion of the file may contain code and/or data from two different segments, and this page will be mapped twice within the address space. For example, if the physical page contains both code and read/write data, it may be mapped with execute-only permissions as part of one segment and copy-on-write permissions as part of another. Moreover, given the page granularity of virtual-to-physical mappings, the data within that page appears alongside the code with execute-only permissions in one segment and the code within that page appears alongside the read/write data but without execute permissions in the other.

This possible sharing of a physical page between two segments necessarily affects the linker's placement of segments within the address space. Consider a segment that ends at offset *off* within a page followed by a segment that begins at offset *off* within the same page. The virtual address that is the end of the first segment must be separated from the virtual address that is the start of the second by a distance that is a multiple of the page size. Otherwise, the same physical page could not be mapped at the end of the first segment and the beginning of the second. In the ELF specification, this restriction on the placement of segments is expressed as follows. The virtual address at which a segment is mapped modulo *maxpagesize* must equal the file position from which the segment is loaded modulo maxpagesize. In effect, depending on the value of maxpagesize, this requirement can create gaps between segments in the address space that our padding mechanism will exploit.

Table 4. PostgreSQL Main Executable Layout in Virtual Address Space

	Default			With In-Kernel Padding		
Region	Start	End	Prot.	Start	End	Prot.
Code	0x400000	0x9f4000	r-x	0x400000	0xa00000	r-x
Data	0xbf4000	0xc00000	rw-	0xbf4000	0xc00000	rw-

The default linker on our system, the GNU binutils ld linker (version 2.17.50), has `maxpagesize` set to 2MB by default.[2] Consequently, as illustrated by the example in Table 4, the end of the executable segment and the beginning of the data segment are separated by 2MB of unused virtual address space.

We modified the kernel to exploit this unused space. Specifically, we modified the kernel to transparently and automatically extend the executable segment up to the next superpage boundary, back it with a physical reservation, and then fill in no-ops as needed. To keep the physical memory allocated for holding no-ops to a minimum, the file content that comes after the code segment (i.e., the data segment) is used to pad out the residual code. Only if that file content is insufficient to reach the end of the superpage will no-ops be added to fill the rest. In other words, some or possibly all of the data section is paged in and mapped twice: once within the superpage that contains the residual code region (which is read/execute-only) and once within the data segment, where it is mapped copy-on-write. This is feasible because (1) the data segment is right up against the code segment in the file, so they will automatically share the physical reservation that holds the residual code region, and (2) the default 2MB `maxpagesize` ensures correct alignment for the data segment regardless of page size used. This double mapping of the data segment means that in cases where the data segment is large enough to fully pad the residual code region, the physical memory footprint does not increase. In fact, as we explain later, in such cases the physical memory footprint is strictly reduced due to savings in kernel data structures.

Padding has the largest impact on PostgreSQL and moderately sized shared libraries, like the standard C library (libc). Padding on PostgreSQL's main executable reduces unhalted execution cycles by 7.75%. Padding on libc has the most impact on MySQL due to its heavy use of the library, reducing unhalted cycles by 1.2%. For the rest of the applications that link libc, padding reduces unhalted cycles by up to 0.45%.

Note that when padding is done on shared libraries, the start of every library's code section has to be aligned at a superpage boundary. In contrast, by default, libraries are packed back to back in the address space under stock FreeBSD. The forced alignment can spread executable mappings out and increase contention in certain sets of the L2 TLB. This side effect has a small but measurable negative impact on Node.js. Unhalted cycles increase by roughly 0.4% when superpage alignment is forced onto Node.js's libraries.

7 ITLB, DATA TLB, STLB, AND PIPELINE INTERACTIONS

In this section, we look at how our workloads interact with the different TLB organizations of Intel Skylake and AMD Zen+. In particular, at L1, their organizations differ in that Skylake has separate structures for caching the different page sizes, whereas Zen+ has a single structure that caches all of the page sizes. Additionally, at L2, their organizations differ in that Zen+ has separate structures for caching instruction versus data mappings, whereas Skylake has a single structure caching both.

[2]In contrast, LLVM's LLD linker and the Gold linker set `maxpagesize` to 4KB by default but allow it to be changed from the command line.

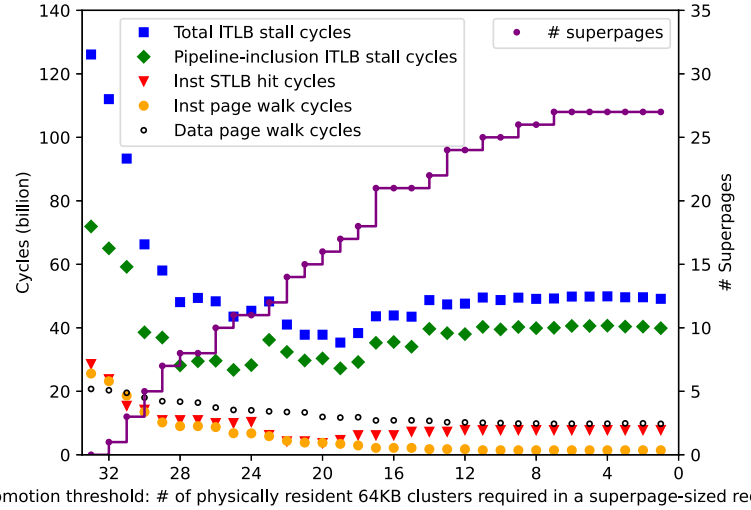

Fig. 4. Clang TLB performance on Skylake with incrementally more aggressive code superpage promotion. Threshold 33 is where code superpages are disabled. Threshold 32 is essentially stock FreeBSD. Threshold 1 is what is commonly referred to as the "first-touch" policy. For reference, the execution cycles at threshold 33 are 2.165 billion.

We begin by presenting various cycle counts side by side for Clang and MySQL running on Skylake. These two applications are particularly interesting because they can use more than eight superpages for their respective main executable at more aggressive promotion thresholds, exceeding the capacity for 2MB entries in Skylake's L1 ITLB (see Table 1). Data page walk cycles are also included to provide additional context.

7.1 Instruction Address Translation Overhead Breakdown

Figures 4 and 5 show the changes in TLB behavior as the superpage promotion threshold is changed for Clang and MySQL, respectively. The figures also include a line indicating the number of superpages at each threshold.[3] Note that there is a point to the left of threshold 32 (stock FreeBSD) to represent the case when no code superpages are allowed. We decompose the instruction address translation overhead into three components: page walk cycles, STLB hit cycles, and pipeline-inclusion stalls; the latter corresponds to the cost of replacing an entry in the L1 ITLB. Page walk cycles are measured by the ITLB_MISSES.WALK_PENDING counter,[4] STLB hit cycles are the per-hit cost (seven cycles [21]) times the STLB hit count, as measured by the ITLB_MISSES.STLB_HIT counter. After subtracting these two components, the rest of the instruction address translation cycles is attributed to pipeline-inclusion stalls.

These two figures show that both instruction and data page walk cycles keep decreasing as we increase the number of code superpages. Data page walk cycles decrease because of the reduced competition for entries in the STLB as more 4KB code mappings are replaced by 2MB code mappings. However, the reductions in both instruction and data page walk cycles gradually drop. This

[3]In contrast to the lines of Figure 2, these lines reflect the actual promotions within the main executable and shared libraries. For example, even though MySQL only touches 29 clusters of libcrypto, promotion of libcrypto occurs at threshold 31 in Figure 5, because other applications have loaded two clusters that are not used by MySQL.
[4]Although each Intel core can do two page walks in parallel [21], we observe that only one walker is ever active to serve ITLB misses, even though we are running concurrent workloads that exercise both hyperthreads in a core.

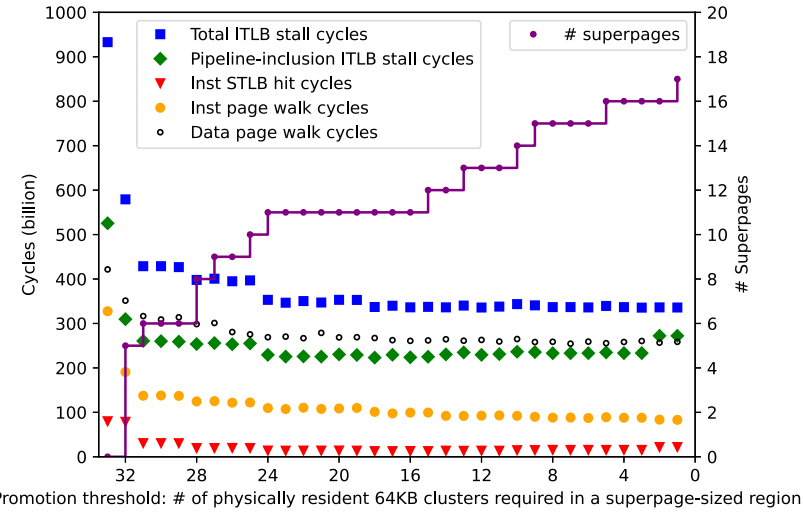

Fig. 5. MySQL TLB performance on Skylake with incrementally more aggressive code superpage promotion. Threshold 33 is where code superpages are disabled. Threshold 32 is essentially stock FreeBSD. Threshold 1 is what is commonly referred to as the "first-touch" policy. For reference, the execution cycles at threshold 33 are 10.076 billion.

is unsurprising because as the promotion threshold is decreased, fewer 4KB code mappings that are actually *used* are being incrementally replaced by 2MB code mappings. Section 7.3 discusses these observations in more detail.

More importantly, these two figures show that pipeline-inclusion ITLB stall cycles consistently account for more than 50%, and sometimes more than 80%, of the instruction address translation overhead. Thus, page walk cycles and STLB hit cycles do not even come close to accounting for the instruction address translation overhead, so neither is a good approximation to the address translation overhead. To understand why, Yasin [20] states that the L1 ITLB sometimes stalls on a miss because replacement is delayed until all of the in-flight instructions mapped by the victim entry are retired from the pipeline. Section 7.2 discusses in more detail the impact that L1 ITLB structure and capacity have on performance.

We note that a more aggressive code superpage policy costs little to no extra memory for two reasons:

- When aggressively promoting reservations, the nearly fully populated reservations require only a small number of extra code pages to be brought into memory. In fact, MySQL needs only 5.6% more 4KB pages (or 1,152KB in absolute terms) over FreeBSD's default policy to achieve 85% of the best possible gains in performance.
- The use of superpages saves page table memory since an entire 4KB leaf-level page table is replaced by a single higher-level PTE. There is also savings on reverse mapping data (i.e., physical-to-virtual mapping entries) of roughly 3X the savings on ordinary page table memory. These savings scale up with the number of processes sharing the code.

7.2 L1 ITLB

Prior work [22, 48] has suggested that the number of code superpages should be carefully managed to not exceed the capacity of the L1 ITLB. Notably, their older Intel processors did not cache 2MB instruction mappings in the STLB (only 4KB instruction mappings were cached in the STLB), so

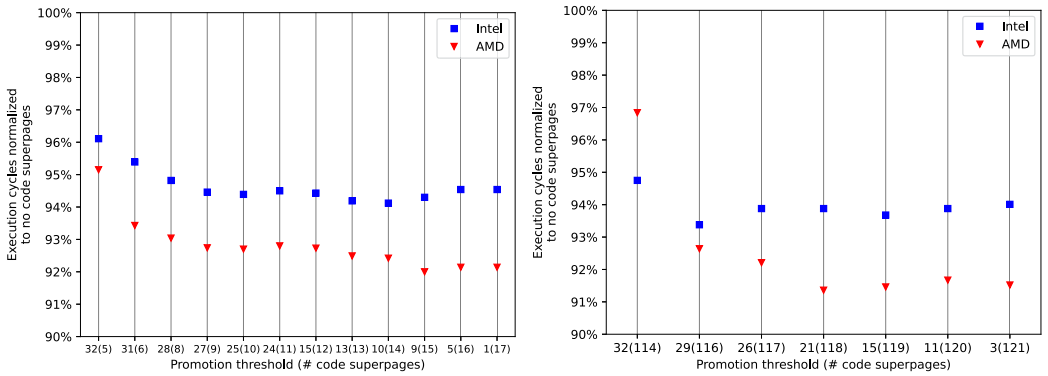

Fig. 6. Normalized user-space execution cycles under different promotion thresholds, with the number of code superpages at a given threshold in parentheses; normalized to no code superpages.

every ITLB miss on a 2MB instruction mapping incurred a page walk. However, as explained in the following, we achieve the best overall performance for Clang and MySQL on Intel Skylake when the number of code superpages exceeds the L1 ITLB size.

Figures 4 and 5 show that the instruction address translation cycles for both Clang and MySQL actually hit their optimal values at a moderate number of aggressively promoted superpages. Then, as the number of superpages increases beyond that point, pipeline-inclusion ITLB stall cycles unfortunately increase. This is because the ITLB on Intel Skylake has only eight 2MB entries per thread (see Table 1), and prior work has shown that overflowing this eight-entry capacity can significantly slow execution, for example, increasing the overall unhalted cycles by 5X in a microbenchmark that suffers frequent pipeline-inclusion ITLB stalls [55]. In effect, as the number of code superpages keeps increasing, the diminishing returns from more instruction STLB hits are overshadowed by the increasing cost of more ITLB misses. Despite this regression in the ITLB performance, overall execution cycles do not necessarily deteriorate (and certainly do not increase by 5X) in real-world workloads. In particular, both Clang and MySQL's overall performance (as shown in Figure 3) keeps improving even as ITLB performance starts to worsen. Because the STLB is shared between instruction and data mappings, promoting additional 4KB code mappings to a 2MB code mapping frees up entries in the STLB for additional data mappings, yielding an overall performance gain. Nonetheless, these results suggest that future microarchitectures should pay attention to ensuring that the L1 ITLB capacity meets the needs of in-flight instructions, which can number in the hundreds.

In contrast to the Skylake L1 ITLB, which has separate structures for caching 4KB versus 2MB mappings, the AMD Zen+ L1 ITLB consists of a single structure in which all 64 entries can cache either 4KB or 2MB mappings, giving Zen+ a much larger L1 ITLB capacity for superpage mappings [18]. Consequently, for large applications, such as Clang and MySQL, instruction mappings are more likely to stay in the L1 ITLB on Zen+ than on Skylake, even if we adopt the most aggressive code superpage policy. Figure 6(a) shows that Zen+ sees a larger impact than Skylake when using code superpages, with greater improvements with increasing superpage counts.

However, with these larger applications, an aggressive policy could end up creating too many sparsely used code superpages, increasing pressure on the L1 ITLB. QEMU is an application where the number of JIT-compiled code superpages in anonymous memory (114) under the stock superpage policy already overwhelms the L1 ITLB on both Skylake and Zen+. Nonetheless, as shown in Figure 6(b), on Zen+ we see continued improvement from creating more file-backed superpages

Table 5. Instruction Page Walks (Intel, ITLB_MISSES.WALK_COMPLETED) or L2 ITLB Misses (AMD, BP_L1TLBMISS_L2MISS) When Applying the Most Aggressive Code Superpage Policy as a Percentage of the Baseline Count, and Walks/Misses Per Thousand Instructions (PKI) Retired

	Intel		AMD	
Benchmarks	**Walks PKI**	**Walks % Baseline**	**Misses PKI**	**Misses % Baseline**
Clang	0.013	6.24%	0.002	0.34%
PostgreSQL	0.256	8.87%	0.169	5.31%
Javac	0.001	2.50%	0.002	0.44%
Derby	0.001	7.20%	0.005	0.77%
Node.js	0.795	99.38%	0.029	12.94%
MySQL	0.240	20.73%	0.124	4.71%
QEMU	0.026	7.89%	0.193	17.50%

after threshold 29. This is because as we replace 4KB file-backed code pages with superpages, we are steadily reducing the competition between 4KB and 2MB mappings within Zen+'s *L2* ITLB. In contrast, on Skylake, the increased pipeline-inclusion stalls from L1 ITLB misses result in performance regression after threshold 29.

7.3 L2 TLB: AMD Zen+ versus Intel Skylake

We now turn our attention to the effects of L2 TLB organization. AMD Zen+ has separate L2 TLB structures for instruction and data mappings, whereas Intel Skylake has a single L2 structure, the STLB, that is used for both. With the separate L2 instruction and data TLBs on Zen+, we find that there is either under- or overutilization of the L2 ITLB for most of our applications, making a case for Skylake's approach of a shared L2 TLB.

Table 5 presents the instruction L2 TLB misses per thousand instructions executed for Zen+ and the instruction page walks per thousand instructions executed for Skylake, when using the most aggressive code superpage policy for both. The table also shows each of these counts as a percentage of its respective baseline count, when there are no code superpages.

For some applications on Zen+, Table 5 shows that instruction L2 TLB misses all but stop occurring, because instruction mappings do not compete with data mappings in the L2 TLB and the translations for those applications' working sets of code fit in the L2 ITLB. In those applications, an examination of Figure 1 shows that the number of superpage mappings for code is small relative to the size of the L2 ITLB, leading to the underutilization of the L2 ITLB when using code superpages on Zen+. For example, for the Clang and Java workloads (i.e., Javac and Derby), less than 1% of the baseline L2 ITLB misses remain after applying the most aggressive code superpage policy. For the PostgreSQL and MySQL databases, about 5% of the baseline L2 ITLB misses remain because of competition for space in the L2 ITLB with the OS when it performs network and disk I/O. In contrast, the STLB on Skylake allows entries not utilized for instruction mappings due to a small number of code superpages to be utilized for data mappings instead.

However, for two applications, Zen+'s smaller L2 ITLB (compared to Skylake's STLB) results in overutilization (and correspondingly higher miss rates). Although using code superpages with Node.js results in a significant drop in instruction L2 TLB misses (to 12.94% of the baseline) on Zen+, mappings for the large number of JIT-compiled 4KB pages (1,009) still compete for space in the 512-entry L2 ITLB. As the workload with the largest amount of code, including 135 code superpages, QEMU experiences the smallest decrease in instruction L2 TLB misses (to 17.50% of the baseline). In short, QEMU is a case where the larger STLB on Skylake offers more capacity for instruction mappings than the smaller dedicated L2 ITLB on Zen+.

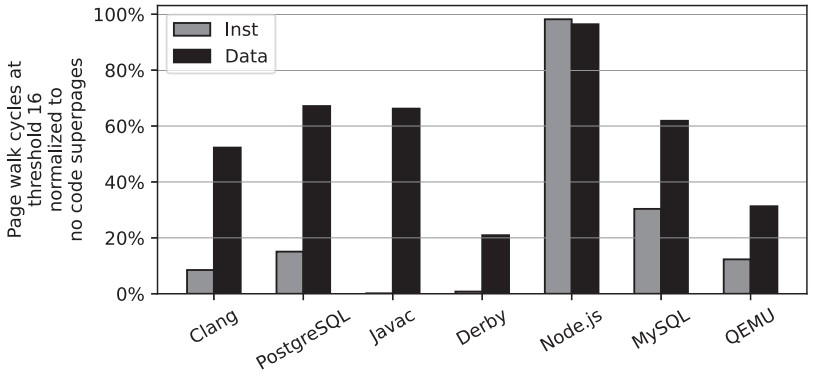

Fig. 7. Normalized page table walk cycles on Skylake when the code superpage promotion policy requires sixteen 64KB clusters to be physically resident.

In contrast to Zen+'s separate L2 structures, Skylake's STLB results in all applications still incurring at least 2.5% of their baseline instruction L2 TLB page walks even when we apply the most aggressive code superpage policy. We attribute this difference to the competition between instruction and data mappings in the STLB. Notably, Node.js exhibits almost no reduction in instruction L2 TLB page walks because none of the JIT-compiled code is superpage-mapped (see Section 4). However, in the case of QEMU, where the Zen+ L2 ITLB is overutilized, the larger capacity of Skylake's 1,536-entry STLB is exploited, resulting in a mere 0.026 page walks per thousand instructions by using entries not used for data mappings for instruction mappings instead. Overall, we believe that an L2 TLB shared by instruction and data mappings will better handle the varied demands of modern applications.

7.4 Shared L2 TLB: Code Superpages versus Data Superpages

On processors like Skylake where the STLB is shared between instruction and data address translations, improving instruction address translation can sometimes reduce data address translation overhead as well. We observe that there is significant improvement in data page walk cycles as a result of using more code superpages. In both Figures 4 and 5, data page walk cycles drop by more than 30% as we go from no code superpages to full code superpages. Although not included in the figures, we did measure the number of data STLB misses, which go down with the increasing number of code superpages, leading to a reduction in data page walk cycles. Expanding this observation to more workloads, Figure 7 shows the instruction and data page walk cycles normalized to stock FreeBSD when the code superpage promotion policy requires sixteen 64KB clusters to be physically resident. In six out of seven of our workloads, using code superpages reduces instruction page walk cycles by more than 70%, and data page walk cycles by more than 30%. Notably, the data page walk cycles of Derby decrease by almost 80%.

More surprisingly, we find that for some widely used applications, the performance benefit of using superpages on code is larger than that of using them on data when the L2 TLB is shared between instruction and data address translations. Zhu et al. [56] showed that more aggressive data superpage policies could improve performance over FreeBSD's stock policy. Here we run the workloads under the most aggressive possible data superpage policy on Intel Skylake, and we find that even under that policy, code superpages still benefit performance more than data superpages in four out of seven of our workloads. Figure 8 shows the normalized user-space execution cycles where we compare the impact of code versus data superpages by selectively applying the most aggressive superpage promotion policy (where any superpage-sized region that is touched is

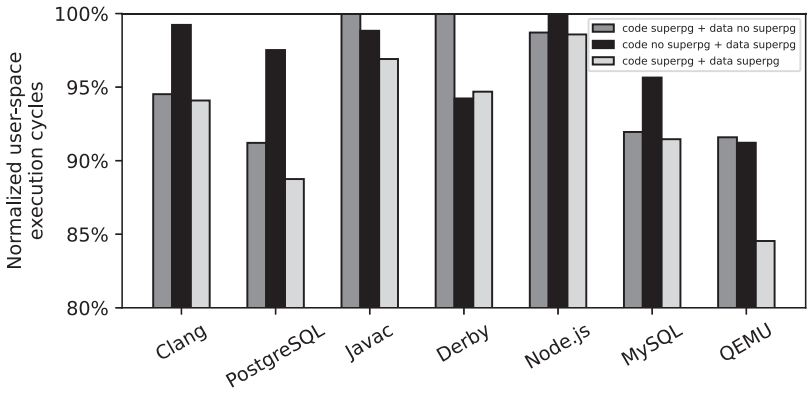

Fig. 8. User-space execution time on Skylake normalized to code no superpg + data no superpg. Both code superpg and data superpg use the most aggressive promotion policy.

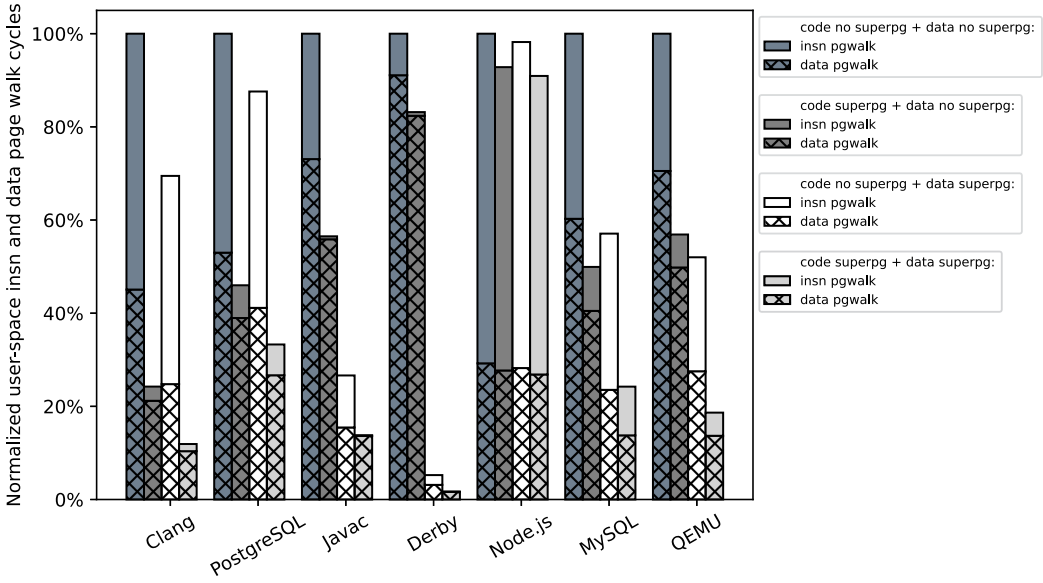

Fig. 9. Combined user-space instruction and data page walk cycles on Skylake normalized to code no superpg + data no superpg. Both code superpg and data superpg use the most aggressive promotion policy. Each bar consists of two components, one representing instruction page walk cycles and the other data page walk cycles. The components are in proportion to their corresponding cycle counts.

immediately faulted in and promoted into a superpage) on either code or data. In the case of Clang, PostgreSQL, Node.js, and MySQL, execution cycles decrease more with code superpages alone than with data superpages alone. This is despite the fact that for PostgreSQL and MySQL the amount of data far exceeds code by a factor of 100X. In fact, when the use of superpages is enabled for code and/or data, there are 2,805 superpages of data versus 18 superpages of code in PostgreSQL, and 2,675 superpages of data versus 36 superpages of code in MySQL. In addition, Figure 9 shows that using code superpages reduces data page table walk cycles more than using data superpages does for Clang, PostgreSQL, and Node.js. In short, when a workload has a significant amount of either code or data, the use of code superpages is absolutely crucial,

due to interference in the STLB, even when the amount of data is orders of magnitude larger than code.

8 THE IMPACT OF CORE COUNTS

Increasing the level of parallelism exacerbates instruction address translation overhead, because contention over space in the TLB and data cache becomes more intense. For Clang in one test, the instruction address translation overhead increases from 4.6% when using a single hyperthread on one core to 5.5% when using both hyperthreads due to competition in the TLB, and to 5.8% of the execution cycles when increasing the number of cores from one to four (using both hyperthreads on each) due to competition in the LLC. For PostgreSQL under persistent connection mode (PostgreSQL-p), the instruction address translation overhead increases from 14.6% to 15.4% when the number of cores goes from one to four. As the parallelism demanded by modern applications and supported by modern hardware continues to grow, attention to the efficiency of instruction address translation is paramount.

Code superpages have a larger impact on performance as a workload scales to use more cores. In general, at higher core counts, there is increased competition in the LLC, which is shared across cores. This increased competition means that the memory accesses performed during a page walk are more likely to miss in the LLC, increasing the page walk cycles. Depending on the workload, when going from one to four cores, per-core LLC stall cycles increase between 2.4X and 4.31X, and per-core instruction page walk cycles increase between 1.06X and 4.16X. However, when code superpages are used, competition for the LLC is slightly reduced because the page table occupies less memory. Consequently, across all of our workloads, the reduction in per-core unhalted execution cycles from the use of code superpages is consistently larger at four cores than at one, by between 0.33 and 4.21 percentage points.

9 OS SUPPORT FOR FILE-BACKED CODE SUPERPAGES

In contrast to Linux's *copy-and-remap* approach (described in Section 2), FreeBSD has direct support for file-backed code superpages. Copy-and-remap can have performance consequences due to unnecessary copying of untouched superpages (e.g., five such pages exist in MySQL in Figure 1). Moreover, in some cases, the overhead of copy-and-remap can be larger than the benefits of using code superpages. Unlike in stock FreeBSD, where (physical) superpages containing file-backed code persist and can be shared among different processes, code superpages created by copy-and-remap are based on anonymous memory that is generally private to the process and does not persist. With the exception of child processes that inherit superpage mappings from parent processes, copy-and-remap has to be re-performed by each and every new process that wants to use code superpages. To quantify this overhead, we developed a small dynamically linked library that performs Linux's copy-and-remap approach in FreeBSD. By using the `LD_PRELOAD` environment variable on the library, we make sure that copy-and-remap is performed before the `main` function starts executing so that the application gets the full benefit of code superpages. This library performs copy-and-remap only for superpage-sized code regions, as is the case on Linux. In a full kernel build workload, copy-and-remap leads to a 17.89% increase in overall execution cycles compared to the baseline configuration of no code superpages, and a 24.57% increase over FreeBSD's superpage mechanism under an aggressive policy. The overhead is so large because successive invocations of the compiler do not have a parent-child process relationship, leading to an excessive number of copy-and-remap operations. In contrast, on applications that perform frequent process fork operations, such as PostgreSQL, repeated copy-and-remap operations are not necessary because child processes automatically inherit the parent's superpage mappings.

Table 6. Percentage of Total Execution Cycles Spent in Kernel Space with User-Space Code Superpages Disabled (Baseline), and Kernel-Space Execution Cycles When Applying the Most Aggressive Code Superpage Policy as a Percentage of Baseline Total Execution Cycles

Benchmarks	Intel Baseline	Intel Most Aggressive	AMD Baseline	AMD Most Aggressive
Clang	9.34%	7.55%	10.61%	8.27%
PostgreSQL	31.48%	31.48%	30.72%	31.02%
Javac	1.52%	0.74%	0.97%	0.96%
Derby	2.48%	1.53%	0.77%	0.94%
Node.js	0.67%	0.53%	0.88%	0.87%
MySQL	19.09%	19.58%	38.81%	39.87%
QEMU	11.49%	10.69%	15.09%	14.05%

As mentioned in Section 3.1, the FreeBSD kernel uses code superpages as much as possible for its own code by default. Although aggressive superpage promotion policies on user-space applications do not directly affect kernel-space code superpages or kernel address translation overhead, aggressive policies can improve kernel-space execution time indirectly. For reference, Table 6 lists for each workload the percentage of total execution cycles spent in kernel space.

For most of the workloads, the percentage of total execution cycles spent in kernel space is about the same on the Intel Skylake and AMD Zen+ processors. In almost all cases, time spent in the kernel is mostly due to I/O and other activity unrelated to superpage management. However, smaller, shallower process page tables as a result of the use of code superpages make kernel-level operations such as fork, execve, and exit cheaper. Although the initial incremental promotion of a code superpage incurs a one-time overhead, the resulting smaller, shallower process page tables benefit all executions from then on, because future processes will immediately map the code as a superpage.

The degree to which superpage policies improve kernel-space execution is application dependent. For instance, in the case of Clang, more aggressive superpage promotion substantially reduces the number of instructions retired and execution cycles in kernel space. There is less work to do during process creation and termination as we replace hundreds of 4KB page PTEs with a single upper-level superpage PTE. In particular, the performance improvement in kernel-space execution is higher than in user-space execution for Clang. Therefore, the improvement in overall execution time would be higher than the improvement in user-space execution time presented in Figure 3. Since Javac, Derby, and Nodejs only spend around 1% of their overall execution times in kernel space (shown in Table 6), the impact of kernel-space execution on their overall execution times is trivial, although their kernel-space execution cycles and retired instruction counts also somewhat decrease. More aggressive superpage promotion does not substantially improve the performance of kernel-space execution for PostgreSQL, MySQL, or QEMU, because the number of kernel-space instructions retired does not decrease for these workloads. As a result, the performance benefits of aggressive superpage promotion is smaller for the overall execution time than for the user-space execution time.

10 RELATED WORK

Superpages. To the best of our knowledge, existing research on superpages often overlooks the code side. Most prior work is based on Linux, which does not provide automatic and transparent support for superpage mappings on executable files. On the data side, Ingens [40] promotes/demotes superpages based both on the number of physically resident pages and on their access frequency. SmartMD [37] examines the impact of using superpages on memory deduplication in virtual environments. Superpages can reduce the effectiveness of memory deduplication techniques if not carefully used. SmartMD's superpage promotion/demotion heuristic is based

on a combination of access frequency and duplication of each page within a superpage region. Carrefour-LP improves the performance of superpages in NUMA systems by dynamically splitting superpages as needed to balance the load across memory controllers [34]. Illuminator [49] and Gorman and Healy [35] improve the ability to allocate superpages by grouping pages that are immovable to avoid the possibility of fragmentation. In subsequent work, Gorman and Healy [36] provide APIs for applications to request superpage allocation explicitly as in `libhugetlbfs` [42] and evaluate the performance impact of using superpages. Ausavarungnirun et al. [27] proposed a new GPU memory manager that allocates contiguous virtual pages to contiguous physical pages in GPU memory to allow the use of superpages.

Compile-Time Optimizations. Ottoni and Maher [48] explored the performance benefits of using superpages on code using `libhugetlbfs` [42]. They show that mapping the hot functions into superpages further improves performance of the server applications tested. Their evaluation uses the Ivy Bridge processor, which does not support 2MB mappings in the L2 STLB, requiring their judicious use of the eight 2MB mapping L1 ITLB entries for hot functions. Mashhadi [41] also explored the performance improvement of mapping hot functions to superpages. On their Haswell processor where the L2 STLB supports 2MB superpages, superpages improve performance by 1% to 2% across the applications they tested. Our work shows that despite the L2 STLB support for 2MB superpages, the problem of performance regression in the L1 ITLB when code superpages are overused is not entirely eliminated on Intel processors. The limited capacity for 2MB entries in the ITLB can still lead to sizeable cumulative end-to-end costs for missing in the ITLB. Compile-time optimizations that improve the locality of code should in principle reduce the likelihood of such interaction, and thus reduce the average cost of ITLB replacement. Future work can explore the effects of compile-time optimizations in this regard.

Hardware Techniques to Improve TLB Performance. To improve address translation efficiency, direct segments and redundant memory mapping support large segments of various lengths, each of which can be translated by a single translation entry [29, 38]. Existing research also improves address translation performance by leveraging memory contiguity and coalescing page translations [33, 50–52]. Vavouliotis et al. [54] propose a composite ITLB prefetcher to reduce the high instruction STLB miss rates of server workloads.

Sharing Page Tables. Previous works have focused on sharing page tables for applications handling a large amount of data [45, 46]. Dong et al. [31, 32] share page tables among Android application processes forked from a template process called `Zygote`. In our previous work [55], we implemented a general approach to page table sharing that does not rely on a special fork model. In principle, our design can be implemented on any standard Unix system. Moreover, page table sharing is proposed to be used for address translation deduplication in next-generation computing systems with ample memory [53].

Hardware Bugs. Multiple generations of Intel processors [5] may unexpectedly incur a machine check error when an instruction address translation hits multiple entries in the ITLB, each corresponding to a different page size, and the entries have different attributes (e.g., the physical address). This can occur, for example, if an executable superpage mapping with different attributes is created without first invalidating all of the 4KB mappings from the TLB that the superpage mapping replaces. However, because of the differing attributes on the mappings, failing to first invalidate the 4KB mappings from the TLB is itself an OS software bug. Consequently, non-virtualized systems running a correctly implemented native OS are not impacted by this hardware bug. Neither are virtualized systems where the guest OS is trusted to perform the TLB invalidations. However, a malicious guest OS running in a virtualized system with an affected processor may launch a denial of service attack by intentionally not doing the TLB invalidations. This attack can be prevented by not creating any executable superpage mappings in the hypervisor-managed nested page table.

11 CONCLUSION

In this article, we examined the instruction address translation overhead for seven widely used applications, ranging from compilers to web user-interface frameworks. The applications have large code sizes, ranging from 22MB to 270MB. We found that the instruction address translation overhead for these applications is non-trivial, ranging from 1.6% to 13.44% of overall execution cycles. Moreover, we found that this overhead increases as the level of parallelism goes up.

Our findings make a case for first-class OS support for superpages on ordinary files containing executables and shared libraries, as well as a more aggressive superpage promotion policy for code. FreeBSD's stock superpage promotion policy when applied to the code in our application suite reduces execution cycles by up to 5.57%. Two techniques can further reduce execution cycles by reducing instruction address translation overhead. First, a more aggressive promotion policy for code can further reduces execution cycles by up to 8.33% over stock FreeBSD, with the gains achieved without the need to resort to Linux's first-touch promotion policy for data superpages. Such a policy would alleviate TLB pressure without significantly increasing the memory footprint. Second, padding the sub-superpage-sized residual regions of the main executable and shared libraries out to a superpage boundary further reduces execution cycles by up to 8% over the more aggressive promotion policy alone.

We made several observations about the interplay between the use of code superpages and ITLB design. First, L1 ITLB design can have an impact on what code superpage policy should be adopted. When the L1 ITLB does not have separate structures for base page entries and superpage entries, as is the case with AMD Zen+, the policy can simply be to aggressively promote code superpages at a certain occupancy threshold. The policy's only concern needs to be the cost of the additional memory and I/O operations. However, an L1 ITLB that has separate structures for 4KB versus 2MB mappings, as is the case with Intel Skylake, can present a challenge to such a superpage policy. Increasing the number of superpages when the size of the L1 superpage structure is small may result in more TLB misses for some applications. Moreover, on some microarchitectures, the cost of these misses can be highly variable. Specifically, replacement can be delayed until all of the in-flight instructions mapped by the victim entry are retired. Second, although separate instruction and data TLBs at the lower level might avoid direct impact on data translation overheads, they can incur much higher instruction translation overheads. At the same time, when code superpages reduce ITLB pressure, such a design does not allow use of the excess capacity for data translation.

Given these observations, our recommendation is for future systems to adopt a TLB design where all entries in the L1 ITLB can hold either base page mappings or superpage mappings, and where the L2 TLB is shared between code and data. Such a TLB design, coupled with the use of an aggressive code superpage policy, leads to better performance for a wide variety of applications. An L1 ITLB having unified capacity for base and superpage mappings is able to provide much greater coverage for code mappings without the worry about a separate structure for superpage mappings having only limited capacity, while also reducing the likelihood of potential pipeline-inclusion stalls. An L2 TLB unified between code and data is able to serve both code and data translations without wasting any of its capacity. The aggressive use of code superpages (1) increases coverage of the TLB, (2) reduces the cost of page walks and the pressure on the LLC, and also (3) increases reuse of code TLB entries, reducing the likelihood of code entries being evicted from the L2 TLB in the face of competition from data entries.

REFERENCES

[1] Reinhold P. Weicker. n.d. Dhrystone, The Classic Benchmark. Retrieved June 2, 2023 from https://github.com/sifive/benchmark-dhrystone.

[2] TIS Committee. n.d. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. Retrieved June 2, 2023 from https://refspecs.linuxfoundation.org/elf/elf.pdf.
[3] WikiChip. n.d. Haswell - Microarchitectures - Intel. Retrieved June 2, 2023 from https://en.wikichip.org/wiki/intel/microarchitectures/haswell_(client).
[4] Intel Open Source: TMA Metrics. Retrieved June 2, 2023 from https://download.01.org/perfmon/TMA_Metrics.xlsx.
[5] Kernel. n.d. iTLB Multihit. Retrieved June 2, 2023 from https://www.kernel.org/doc/html/next/admin-guide/hw-vuln/multihit.html.
[6] Node.js. n.d. Node.js Home Page. Retrieved June 2, 2023 from https://nodejs.org/en/.
[7] QEMU. n.d. QEMU Home Page. Retrieved June 2, 2023 from https://www.qemu.org/.
[8] WikiChip. n.d. Skylake (client) - Microarchitectures - Intel. Retrieved June 2, 2023 from https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client).
[9] SQLite. n.d. SQLite Home Page. Retrieved June 2, 2023 from https://www.sqlite.org/index.html.
[10] SPEC. 1995–2008. SPECjvm 2008. Retrieved June 2, 2023 from https://www.spec.org/jvm2008/.
[11] SPEC. 1995–2008. SPECjvm2008 Benchmarks. Retrieved June 2, 2023 from https://www.spec.org/jvm2008/docs/benchmarks/index.html.
[12] PostgreSQL. 1996–2018. pgbench—Run a Benchmark Test on PostgreSQL. Retrieved June 2, 2023 from https://www.postgresql.org/docs/9.6/static/pgbench.html.
[13] PostgreSQL. 1996–2018. PostgreSQL: The World's Most Advanced Open Source Relational Database. Retrieved June 2, 2023 from https://www.postgresql.org/.
[14] OpenJDK. 2014. JDK 8. Retrieved June 2, 2023 from http://openjdk.java.net/projects/jdk8/.
[15] Intel. 2014. Using Intel VTune Amplifier XE to Tune Software on the 6th Generation Intel Core Processor Family. Retrieved June 2, 2023 from https://software.intel.com/sites/default/files/managed/dc/3a/Using-intel-vtune-amplifier-xe-on-6th-generation-intel-core-processors-1-0.pdf._Intel_VTune_Amplifier_XE_on_6th_Generation_Intel_Core_Processors_1.0.pdf.
[16] Intel. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Vol. 3. Intel.
[17] NPM. 2016. React Server-Side Rendering Benchmark. Retrieved June 2, 2023 from https://www.npmjs.com/package/react-ssr-benchmarks.
[18] AMD. 2017. Software Optimization Guide for AMD Family 17h Processors. Retrieved June 2, 2023 from https://www.amd.com/en/support/tech-docs/software-optimization-guide-for-amd-family-17h-processors.
[19] FreeBSD. 2018. CPUSET(1) FreeBSD General Commands Manual. Retrieved June 2, 2023 from https://www.freebsd.org/cgi/man.cgi?query=cpuset&apropos=0&sektion=0&manpath=FreeBSD+11.2-RELEASE+and+Ports&arch=amd64&format=html.
[20] Ahmad Yasin. 2018. How TMA* addresses challenges in modern servers and enhancements coming in IceLake. In *Proceedings of the Scalable Tools Workshop*. https://dyninst.github.io/scalable_tools_workshop/petascale2018/assets/slides/TMA%20addressing%20challenges%20in%20Icelake%20-%20Ahmad%20Yasin.pdf.
[21] Intel. 2018. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Vol. 3. Intel.
[22] Stephane Eranian. 2018. Linux perf_events Updates. In *Proceedings of the Scalable Tools Workshop*.
[23] MySQL. 2018. MySQL Home Page. Retrieved June 2, 2023 from https://www.mysql.com/.
[24] MySQL. 2018. MySQL Benchmark Tool. Retrieved June 2, 2023 from https://dev.mysql.com/downloads/benchmarks.html.
[25] FreeBSD. 2018. PMCSTAT(8) FreeBSD System Manager's Manual. Retrieved June 2, 2023 from https://www.freebsd.org/cgi/man.cgi?query=pmcstat&apropos=0&sektion=0&manpath=FreeBSD+11.2-RELEASE+and+Ports&arch=amd64&format=html.
[26] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. 631–644.
[27] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and OnurMutlu. 2017. Mosaic: A GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO'17)*. ACM, New York, NY, 136–150. https://doi.org/10.1145/3123939.3123975
[28] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 48–59. https://doi.org/10.1145/1815961.1815970
[29] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 237–248. https://doi.org/10.1145/2485922.2485943

[30] Emilio G. Cota and Luca P. Carloni. 2019. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19)*. ACM, New York, NY, 74–87. https://doi.org/10.1145/3313808.3313811

[31] X. Dong, S. Dwarkadas, and A. L. Cox. 2015. Characterization of shared library access patterns of Android applications. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*. Poster paper. 112–113. https://doi.org/10.1109/IISWC.2015.19

[32] Xiaowan Dong, Sandhya Dwarkadas, and Alan L. Cox. 2016. Shared address translation revisited. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. ACM, New York, NY, Article 18, 15 pages. https://doi.org/10.1145/2901318.2901327

[33] Y. Du, M. Zhou, B. R. Childers, and D. Mosséndr R. Melhem. 2015. Supporting superpages in non-contiguous physical memory. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. 223–234. https://doi.org/10.1109/HPCA.2015.7056035

[34] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quema. 2014. Large pages may be harmful on NUMA systems. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*. 231–242. https://www.usenix.org/conference/atc14/technical-sessions/presentation/gaud.

[35] Mel Gorman and Patrick Healy. 2008. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th International Symposium on Memory Management (ISMM'08)*. ACM, New York, NY, 41–50. https://doi.org/10.1145/1375634.1375641

[36] Mel Gorman and Patrick Healy. 2012. Performance characteristics of explicit superpage support. In *Proceedings of the 6th Annual Workshop on the Interaction Between Operating Systems and Computer Architecture (WIOSCA'10)*.

[37] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John C. S. Lui. 2017. SmartMD: A high performance deduplication engine with mixed pages. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 733–744. https://www.usenix.org/conference/atc17/technical-sessions/presentation/guo-fan.

[38] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 66–78. https://doi.org/10.1145/2749469.2749471

[39] Vasileios Karakostas, Osman S. Unsal, Mario Nemirovsky, Adrián Cristal, and Michael M. Swift. 2014. Performance analysis of the memory management unit under scale-out workloads. In *Proceedings of the 2014 IEEE International Symposium on Workload Characterization (IISWC'14)*. 1–12.

[40] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 705–721. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon.

[41] Rahman Lavaee Mashhadi. 2018. *Profile-Guided Memory Layout: Theory and Practice*. Ph.D. Dissertation. University of Rochester, Rochester, NY.

[42] Mel Gorman. 2010. Huge Pages Part 1 (Introduction). Retrieved June 2, 2023 from https://lwn.net/Articles/374424/.

[43] H. J. Lu, Kshitij Doshi, Rohit Seth, and Jantz Tran. 2006. Using hugetlbfs for mapping application text regions. In *Proceedings of the Linux Symposium*.

[44] Artemiy Margaritov, Dmitrii Ustiugov, Amna Shahab, and Boris Grot. 2021. PTEMagnet: Fine-grained physical memory reservation for faster page walks in public clouds. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM, New York, NY, 211–223. https://doi.org/10.1145/3445814.3446704

[45] Jim Mauro and Richard McDougall. 2006. *Solaris Internals* (2nd ed.). Prentice Hall PTR, Upper Saddle River, NJ.

[46] Dave McCracken. 2003. Sharing page tables in the Linux kernel. In *Proceedings of the Linux Symposium*. 315.

[47] Juan Navarro, Sitarar Iyer, Peter Druschel, and Alan Cox. 2002. Practical, transparent operating system support for superpages. *SIGOPS Operating Systems Review* 36, SI (Dec. 2002), 89–104. https://doi.org/10.1145/844128.844138

[48] G. Ottoni and B. Maher. 2017. Optimizing function placement for large-scale data-center applications. In *Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'17)*. 233–244. https://doi.org/10.1109/CGO.2017.7863743

[49] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making huge pages actually useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, New York, NY, 679–692. https://doi.org/10.1145/3173162.3173203

[50] C. H. Park, T. Heo, J. Jeong, and J. Huh. 2017. Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations. In *Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 444–456. https://doi.org/10.1145/3079856.3080217

[51] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[52] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced large-reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE, Los Alamitos, CA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[53] Michael M. Swift. 2017. DRAFT : Towards O(1) memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'17)*. 1–8.

[54] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. 2021. Morrigan: A Composite instruction TLB prefetcher. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*. ACM, New York, NY, 1138–1153. https://doi.org/10.1145/3466752.3480049

[55] Y. Zhou, X. Dong, A. L. Cox, and S. Dwarkadas. 2019. On the impact of instruction address translation overhead. In *Proceedings of the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'19)*. 106–116.

[56] Weixi Zhu, Alan L. Cox, and Scott Rixner. 2020. A comprehensive analysis of superpage management mechanisms and policies. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. 829–842. https://www.usenix.org/conference/atc20/presentation/zhu-weixi.

Received 17 January 2023; revised 12 April 2023; accepted 8 May 2023