

On the Impact of Instruction Address Translation Overhead

Yufeng Zhou¹, Xiaowan Dong², Alan L. Cox¹, and Sandhya Dwarkadas²

¹Department of Computer Science, Rice University
{yufengz, alc}@rice.edu

²Department of Computer Science, University of Rochester
{xdong, sandhya}@cs.rochester.edu

Abstract—Even on modern processors with their ever larger instruction translation lookaside buffers (TLBs), we find that a variety of widely used applications, ranging from compilers to web user-interface frameworks, suffer from high instruction address translation overheads. In this paper, we explore the efficacy of different operating system-level approaches to automatically reducing this instruction address translation overhead. Specifically, we evaluate the use of automatic superpage promotion and page table sharing as well as a transparent padding mechanism that enables small code regions to be mapped using superpages. Overall, we find that the combined effects of these different approaches can reduce an application’s total execution cycles by up to 18%. Surprisingly, we find that improving address translation performance in the first-level instruction TLB can significantly reduce the address translation overhead for data accesses. The overall reduction in execution cycles is more than double the instruction address translation overhead on stock FreeBSD, demonstrating that data address translation and access synergistically benefit from less contention in the caches and TLBs that might be shared across instruction and data.

Index Terms—Superpage, Translation Lookaside Buffer, Operating System, Microarchitecture, Memory Management, Instruction Access, Page Table, Memory Hierarchy, Address Translation, Virtual Memory

I. INTRODUCTION

In recent years, 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 [1]–[6] as well as improvements to the automatic support for superpages in operating systems [7]. However, much less attention has been paid to the performance impact of address translation on instruction accesses, at either the architectural or operating system level [8]. Instead, most of the work aimed at reducing address translation overhead for instruction accesses has focused on compile-time techniques for code layout optimization [9], [10]. In this paper, we explore the efficacy of different operating system-level approaches to automatically reducing instruction address translation overhead.

Even on modern processors with their ever larger instruction TLBs, we find that a variety of widely used applications, ranging from compilers to web user-interface frameworks, suffer from high instruction address translation overheads. For example, on an Intel Xeon E3-1240 v5 (Skylake) processor [11], we

find that the PostgreSQL (version 9.6.8) database executing a select-only workload spends up to 14.9% of its execution cycles stalled on instruction address translation. To put this in perspective, Intel’s VTune profiler reports instruction address translation as a performance problem when the stall cycles exceed 5% of the execution cycles [12]. Five out of the six applications that we examine in detail in this paper have stall cycles exceeding this 5% threshold. Further, these applications are representative of multiple languages (C, Java, JavaScript) and run-time environments.

We believe that the address translation overhead for instruction accesses is being exacerbated by two trends. The first one is the ever increasing size and complexity of the applications. For example, the Clang compiler has 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 run-time environment uses 20 shared libraries. Applications that make extensive use of shared libraries have been observed to have higher instruction TLB miss rates [8], [13]. Second, when the number of logical or physical cores running an application increases, often its instruction address translation overhead increases because of the greater competition for shared hardware resources, such as TLBs and caches. These effects are most pronounced in applications that have a multi-process architecture, such as PostgreSQL, or in multi-process workloads, such as parallel compilation with Clang. Essentially, because each process has its own private page table, there is more competition for space in shared caches. For example, during initialization, PostgreSQL (version 9.6.8) starts 6 processes. Then, during operation, another process is created to service each client. In one workload, as the number of clients increased from 8 to 12, the instruction address translation overhead grew from 7.7% to 8.3% of the execution cycles.

The first approach that we evaluate is automatic superpage promotion. We performed this analysis on FreeBSD because of its existing support for automatic superpage promotion on code. Notably, any sufficiently large and aligned region of code, including code obtained from executables and shared libraries stored in a regular file system, can be promoted [14]. However, a promotion always requires that every 4KB page within an eligible region be resident in

memory, and FreeBSD’s policy is never to perform I/O just to enable promotion. Consequently, we find that only a few eligible regions of code are actually promoted. Therefore, we explore the effects of relaxing this policy. Essentially, we modified the page fault handler to retrieve the remaining non-resident pages within an eligible region from the file system once the number of already resident pages crosses a configurable threshold. For example, for PostgreSQL (version 9.6.8), setting this threshold to 448 out of 512 pages enabled superpage promotions that resulted in an 8.5% reduction in execution cycles. In this case, the additional physical memory allocated due to this relaxed policy is negligible compared to the applications’ memory consumption for data and is partly compensated for by the savings in page table memory.

Code size is rarely an integer multiple of the superpage size. Consequently, virtually all large executables have a *residual* code region that is mapped using 4KB pages. In the worst case, on a Skylake processor, mappings for this region could occupy almost 1/3 of the entries in the second-level TLB (STLB), displacing many mappings to data. We evaluate two techniques to mitigate such STLB pressure. First, we evaluate the benefits of *padding*, an in-kernel mechanism that transparently pads the residual part of the code to a superpage boundary. Second, many shared libraries used by applications are only tens to hundreds of KBs, making them too small to use superpage mappings. In current operating systems, although the physical memory storing the code for a shared library is shared among processes, each process has a private copy of the page table entries (PTEs) mapping that shared library. Thus, the memory occupied by these PTEs grows linearly with the number of processes. More significantly, these PTEs compete for space all over the memory hierarchy, because a page table walk loads the corresponding PTEs into caches on a TLB miss. Sharing PTEs can be an effective approach to alleviating this duplication [8]. To share PTEs, we leverage the multi-level structure of the x86-64 page table. Essentially, we added an option to the kernel’s virtual memory system that utilizes a single copy of the page table pages (PTPs) storing the PTEs for mapping executable files into virtual memory. We find that sharing PTPs across different processes can reduce execution cycles by as much as 6.9%. Moreover, the combined effects of using superpages to map the main executable and sharing PTPs for the small shared libraries can reduce execution cycles up to 18.2%.

In summary, our contributions are as follows:

- We show that the overhead of instruction address translation for a variety of widely used applications is non-trivial and deserves attention. We find that this overhead is likely to increase as the level of parallelism goes up.
- We show that improving address translation performance in the first-level instruction TLB can reduce the address translation overhead for data accesses, because many modern microarchitectures, such as Skylake and ThunderX2 for AArch64, share the second-level TLB between instruction and data translations.
- We analyze the performance impact of in-kernel au-

tomatic superpage promotion for code across different levels of aggressiveness. We also apply padding to the residual code region to further improve performance.

- We explore a page table sharing scheme for shared libraries that are generally too small for superpages.
- Contrary to prior work [15], which focuses on the cost of TLB misses that result in page table walks, we show that state-of-the-art instruction TLB (ITLB) designs must pay attention both to their interaction with in-flight instructions and with multi-level TLB designs where the lower levels may be contended for by instruction and data translations.

II. BACKGROUND

Today, large computing systems support terabytes of physical memory to accommodate memory-hungry applications such as in-memory databases. However, a TLB’s size cannot scale at the same rate as physical memory. Consequently, virtual-to-physical address translation overhead can be significant for big memory workloads [1], [2].

Using larger page sizes (superpages) can reduce this address translation overhead by increasing TLB coverage. Consequently, modern processors support superpages of potentially multiple sizes (e.g., 2MB and 1GB on x86 processors), and their TLBs provide a growing number of entries for storing these superpage mappings. For example, the number of 2MB superpage mappings that can be stored in Intel’s second-level TLB (STLB) has grown from 1,024 in Haswell [16] to 1,536 in Skylake [11]. However, due to the L1 TLB being on the instruction critical path, only a small number of superpage mappings are supported at the L1 level. In the case of Skylake (see Table I), only 8 entries for 2MB superpages are supported at the instruction (I) TLB (under the assumption that code size is usually not very large, 1GB entries are not supported). Unless otherwise noted, we use “superpage” and “2MB superpage” interchangeably.

We focus here on surveying the state of the art in terms of support for code superpages within the operating system, in particular, in Linux and FreeBSD, and of support for sharing instruction page tables across processes.

A. Superpage Support for Instructions

Linux does not transparently support superpages on code from a regular 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 [17] or (2) that the user process first copies its code from the original virtual memory region backed by 4KB pages to superpage-backed 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. 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. It uses a reservation-

based allocator to support superpages transparently [14]. 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 in order 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 PTP with a single superpage mapping. Reservations that have not been fully populated can be broken if ever there is a shortage of free physical memory. In this paper, we use FreeBSD to analyze the performance impact of automatic superpage promotion on code, as its overhead is smaller than Linux’s copying and remapping approach.

B. Support for Sharing Instruction Page Tables

There are plenty of opportunities for sharing PTPs in desktop and server applications. These applications can share page tables for the main executable and shared libraries across different processes in the following scenarios:

- **Multi-process applications** A multi-process architecture is widely used by desktop and server applications for robustness and security. For example, Chromium uses separate processes for web browser tabs to protect the application from bugs in the rendering engine [18]. Firefox has moved from a multi-threaded architecture to a multi-process architecture for the same reason [19]. PostgreSQL has been using a multi-process architecture for decades.
- **Multiple instances of one application** For example, one may be simultaneously running multiple Clang processes for a parallel build job, or invoking multiple V8 JavaScript engine instances in a multi-process browser.
- **Different applications invoking the same shared libraries** Common shared libraries are linked by different applications. For example, almost every application relies on the standard C library `libc.so`.

Dong et al. [8] share subsets of page tables for shared library code among Android applications, with the goal of sharing both page table pages and TLB entries. Every Android application forks from a template process called *Zygote*, which preloads all shared libraries common to typical Android applications at system initialization time. Consequently, the virtual-to-physical mappings of the preloaded shared libraries that are inherited from the *Zygote* are identical across all Android applications, allowing TLB entries to be shared with the aid of ARM’s protection domain support [8]. When creating a page table for an Android application at a fork, instead of creating duplicate copies of the PTPs, the OS populates the upper-level PTEs with the physical addresses of existing leaf-level page table pages (PTPs) mapping the shared libraries. Sharing PTPs reduces overhead both at startup time due to reduction in the number of PTPs initialized, and over the course of execution due to fewer soft page faults (physical frames being present in memory but with missing mappings) and more efficient

cache utilization. We design, implement, and evaluate a more generalized approach for page table sharing that does not rely on a process-to-process relationship.

III. EVALUATION FRAMEWORK

For our experiments we used an Intel Xeon E3-1240 v5 quad-core processor with a 3.5GHz base frequency, 8MB of last-level cache (LLC), and hyperthreading enabled. We installed 32GB of RAM in the system. TLB-related information is listed in Table I. The baseline system is FreeBSD 11.2-RELEASE, which disables CPU scaling and sets the frequency to the maximum of 3.5GHz by default. We collect information from hardware performance-monitoring counters [12], [20], [21] (Table II¹) using the `pmcstat` [22] utility. We report the median of three runs for all benchmarks except Javac and Derby. For Javac and Derby, we use ten runs due to larger-than-usual variance. We find that the medians stabilize after those numbers of runs. We assign dedicated cores to applications with the `cpuset` [23] utility, and monitor only the activity of those cores. This means that for the server-oriented applications, we collect counters only on cores running the server processes. Unless otherwise noted, we assign two cores (four hyperthreads) to each application evaluated. For all benchmarks, we perform warmup runs to ensure that the necessary code and data pages are faulted in.

In all cases, kernel code is mapped entirely using superpages, so that our optimizations do not directly impact kernel-level address translation overhead. Since our optimizations only directly impact user space address translation overhead, unless otherwise stated, the numbers presented in the evaluation are for user space. For reference, Table III lists for each benchmark the percentage of total execution cycles spent in kernel space. In almost all cases, time spent in the kernel is mostly due to IO and other activity unrelated to memory management and hence remains unchanged by our optimizations. However, smaller process page tables as a result of our optimizations make the execution of kernel-level operations such as `fork`, `exec`, and `exit` cheaper. Superpage promotions incur a one-time additional overhead. We report the impact of these kernel operations on kernel space execution time where significant.

We evaluated six widely used applications, including a compiler, two database applications, and two language runtimes with JIT compilation capabilities. We believe that these applications reflect a wide range of circumstances (Table IV).

Clang: Clang is a compiler front end that uses the LLVM compiler infrastructure as its back end. 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 of a benchmark named Dhrystone [24].

¹(1) `ICACHE_64B.IFTAG_STALL` is a counter recently introduced in Skylake and it measures the end-to-end cost of ITLB misses. (2) Intel processors since Broadwell can do two page walks in parallel [21]. As a result, the “WALK_PENDING” counters can increment by up to 2 per CPU cycle as they add 1 per cycle for each ongoing page walk. In contrast, the “WALK_ACTIVE” counter increments by up to 1 per CPU cycle as it records the number of CPU cycles where there is at least one ongoing walk.

TABLE I
TLB CONFIGURATION (SKYLAKE)

ITLB		
4KB	128 entries	8-way set associative
2MB	8 entries per thread	fully associative
DTLB		
4KB	64 entries	4-way set associative
2MB	32 entries	4-way set associative
1GB	4 entries	fully associative
Shared TLB		
4KB + 2MB	1536 entries	12-way set associative
1GB	16 entries	4-way set associative

TABLE II
HARDWARE PERFORMANCE COUNTERS

Descriptions	Counters & Equations
# of instructions retired	INST_RETIRED.ANY_P
Execution cycles	CPU_CLK_UNHALTED.THREAD_P
Inst addr translation cycles (ITLB stall)	ICACHE_64B.IFTAG_STALL
Inst addr translation overhead	ICACHE_64B.IFTAG_STALL / CPU_CLK_UNHALTED.THREAD_P
% of inst addr translation cycles spent on instruction page table walk	ITLB_MISSES.WALK_ACTIVE / ICACHE_64B.IFTAG_STALL
Inst page table walk cycles	ITLB_MISSES.WALK_PENDING
# of inst page table walks	ITLB_MISSES.WALK_COMPLETED
Avg. cycles per inst page table walk	ITLB_MISSES.WALK_PENDING / ITLB_MISSES.WALK_COMPLETED
Data page table walk cycles	DTLB_LOAD_MISSES.WALK_PENDING + DTLB_STORE_MISSES.WALK_PENDING
# of data page table walks	DTLB_LOAD_MISSES.WALK_COMPLETED + DTLB_STORE_MISSES.WALK_COMPLETED
LLC stall cycles	CYCLE_ACTIVITY.STALLS_L3_MISS

PostgreSQL: PostgreSQL is an object-relational database system [25]. We run version 9.6.8 of PostgreSQL, and use pgbench [26] to perform select-only transactions. The server processes and the client process run on two separate but identical FreeBSD systems as described above with a dedicated Gigabit Ethernet link. Unless stated otherwise, we configure pgbench to simulate 10 clients, resulting in 10 worker processes (5 per physical core) on the server side. In this way, the cores are kept busy by overlapping computation at the database with network and I/O activities. A fixed number of back-to-back transactions are performed on a 5GB database, and we use the `-C` option of pgbench to toggle between reconnecting after each transaction (reconnect mode) and using one persistent connection per client (persistent connection mode). We use the reconnect mode by default unless stated otherwise.

Javac and Derby: We use OpenJDK 8 [27], which is a Java runtime supporting JIT compilation, to run `Compiler.compiler` and `Derby` from the SPECjvm2008 benchmark suite [28],

TABLE III
PERCENTAGE OF TOTAL EXECUTION CYCLES SPENT IN KERNEL SPACE

Benchmarks	Percentage
Clang	7%
PostgreSQL (reconnect)	65%
PostgreSQL (persistent connection)	17%
Javac	1%
Derby	0.7%
Node.js	0.4%
MySQL	20%

[29]. `Compiler.compiler` compiles a set of `.java` files using the `javac` compiler. We refer to the benchmark as “Javac” for simplicity. `Derby` uses an open-source database written in Java and stresses the use of (bigger-than-64-bit) decimal arithmetic computations and the locking behavior. We use the `specjvm.hardware.threads.override` option to allow the benchmarks to scale appropriately with the number of cores allocated. We also use the `XX:+AlwaysPreTouch` option to pretouch the java heap during JVM initialization and fix the number of operations. We pay attention to both the large 11.220MB JVM proper (which comes in the form of a shared library `libjvm.so`) and JIT-compiled code. When we disable automatic code superpage promotion or employ more aggressive code superpage promotion, they are applied to both the JVM and the JIT-compiled code.

Node.js: Node.js is a JavaScript runtime built on Chrome’s V8 JavaScript engine [30]. We run version 8.11.1 of Node.js, and use the React server-side rendering benchmark [31]. We modified the benchmark to run a fixed number of iterations for roughly the same duration as it would originally. We run one Node.js process per hyperthread, which is typical of a master-worker configuration of Node.js on a multi-core system.

MySQL: MySQL is a relational database management system [32]. We run version 8.0.2 of MySQL, and use the read-only OLTP test of sysbench [33] to test the MySQL server. We run the server process and the client process on two separate but identical FreeBSD systems as described above with a dedicated Gigabit Ethernet link. We allow 14 worker threads per physical core for the server process (in order to saturate the cores without incurring a significant increase in queueing delay). We perform 1,600,000 back-to-back transactions in total on a 1.2G database.

IV. INSTRUCTION ADDRESS SPACE AND TRANSLATION OVERHEAD ANALYSIS

Table IV lists the main executable sizes, linked shared libraries, and concurrency models of the applications we examined. (For Javac and Derby, the main executable refers to JIT-compiled code.) The main executables range in size from 5.953MB to as large as 55.895MB, all of which are large enough for superpages. All applications except for Clang link shared libraries. The applications that do rely on shared libraries each link at least 13, with Node.js using as many as 20 libraries. Most shared libraries are too small for superpages, except for a few larger shared libraries, such as `libcrypto.so.8` (2.27MB) linked by PostgreSQL and `Node.js`. We note that PostgreSQL is a multi-process application. The PostgreSQL server consists of 6 processes plus one worker process per client. For a parallel build, multiple instances of Clang will run concurrently.

As outlined in Section II-A, FreeBSD loads 64K-aligned regions of data/instructions into memory (64KB clusters) on a page fault. Any 2MB superpage-sized region of code contains 32 such clusters, resulting in a likelihood of code superpages that are mostly populated. Figure 1 provides a cumulative histogram of the number of 64KB clusters resident in memory

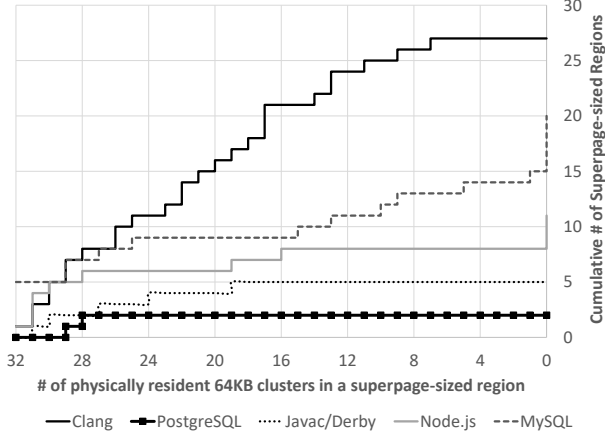


Fig. 1. Cumulative histogram of the # of superpage-sized regions with respect to the # of physically resident 64KB clusters within a region

in each code superpage-sized region for our applications. We use one plot for both Javac and Derby since they have the same results. Under FreeBSD’s conservative superpage promotion policy, Clang and Node.js each have one code superpage promotion. Javac, Derby, and PostgreSQL have none. MySQL has five superpage promotions.

Figure 2 illustrates the instruction address translation overhead (ITLB stall, including cycles servicing ITLB hits, STLB hits, and STLB misses) as a percentage of the overall execution cycles on 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). PostgreSQL does not have any superpage-sized regions that qualify for automatic promotion; “Postgres-p” and “Postgres-r” refer to the persistent connection mode and the reconnect mode of pgbench respectively. As the figure shows, even with FreeBSD’s automatic superpage promotion, up to 14.9% of the application’s execution time can be spent on instruction address translation.

Moreover, when we increase the level of parallelism, the instruction address translation overhead becomes even worse, since the contention over TLB and cache space becomes more intense across logical and physical cores. 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, and to 5.8% of the execution cycles when increasing the number of cores (using both hyperthreads on each) from 1 to 4. For PostgreSQL under persistent connection mode, the instruction address translation overhead increases from 14.6% to 15.4% when the number of cores goes from 1 to 4. 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.

TABLE IV
APPLICATIONS’ MAIN EXECUTABLE AND SHARED LIBRARIES LINKED

	Main executable size (MB)	# of shared libraries linked	# of shared libraries <1MB	Large shared library size (MB)	Multi-thread	Multi-process
Clang	55.895	0	0	N/A	N	N
PostgreSQL	5.953	15	10	1.53 to 2.88	N	Y
Javac	20.000	13	11	1.59 and 11.22	Y	N
Derby	12.000	15	13	1.59 and 11.22	Y	N
Node.js	23.836	20	16	1.59 to 2.88	N	N
MySQL	40.094	15	13	1.59 and 2.27	Y	N

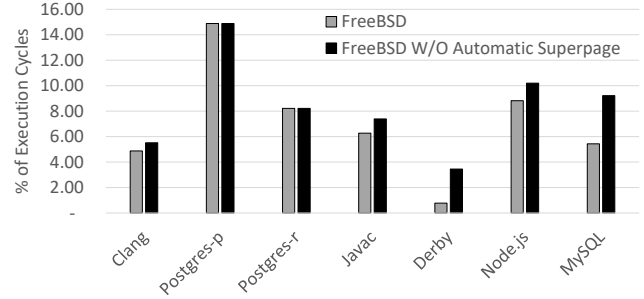


Fig. 2. Percentage of execution cycles servicing instruction address translation

V. AUTOMATIC SUPERPAGE PROMOTION FOR INSTRUCTIONS

Motivated by Figure 1, we explore more aggressive code superpage promotion policies. Aggressive promotion policies trade additional physical memory consumption with reduced address translation overhead (both page table size and TLB occupancy). We accomplish this by using a threshold of occupancy of superpage-sized and aligned reservations, which when exceeded results in the kernel *automatically* filling in the missing clusters and performing a promotion. We evaluate the impact on performance when varying this threshold. The kernel keeps track of the number of resident 64KB clusters in each reservation and checks the occupancy when handling a hard page fault (when both the physical page and the mapping are missing). If the miss handling results in the number of resident clusters being above the threshold, the remaining missing clusters in the reservation are also loaded, and a superpage promotion is performed.

In effect, FreeBSD’s default superpage promotion policy has a threshold of 32, promoting a reservation if and only if all 32 clusters within a reservation are made resident over the course of execution. It is clear from Figure 1 that once we lower the threshold to 16, we would have all reservations promoted for PostgreSQL, Node.js, Javac, and Derby. Clang would still have 6 out of 27 reservations not promoted at a threshold of 16, suggesting that its access to code is relatively spread out. Moreover, PostgreSQL requires only 7 more clusters (= 448KB) to be brought into memory for both of its 2 reservations to be promoted.

For the JVM applications, in addition to more aggressively mapping the JVM (`libjvm.so`), we also modified the JVM to expand its JIT-compiled code heap in 2MB increments and

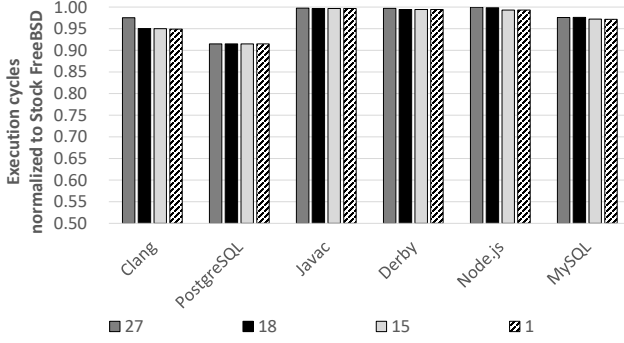


Fig. 3. Normalized execution cycles under 4 different promotion thresholds (Legend shows the # of 64KB clusters that are required to be physically resident before the corresponding superpage-sized region can be promoted)

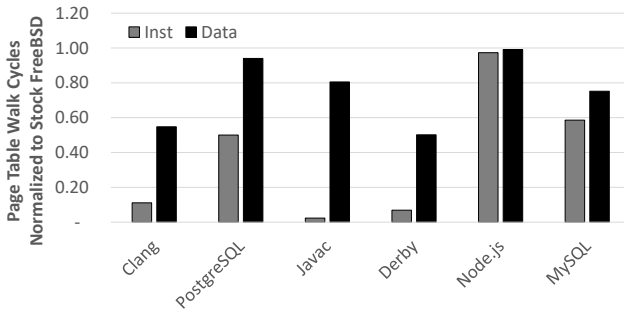


Fig. 4. Normalized page table walk cycles when code superpage promotion heuristic requires 18 64KB clusters to be physically resident

to pre-touch its code heap upon expansion. This way, the JIT-compiled code is entirely mapped with superpages. In the rest of this section, we focus on only the JVM when analyzing the impact of different superpage promotion thresholds, while the entire JIT-compiled code is always mapped by superpages.

In Figure 3, we show the normalized execution cycles at four different thresholds. A threshold of 1 represents the most aggressive superpage promotion policy where any superpage-sized region that is touched is immediately faulted in and promoted into a superpage. Across the board, the best performance is achieved at the most aggressive threshold of 1. However, we find that a less aggressive threshold of 18 can achieve performance very close to the optimal performance. In fact, for PostgreSQL and MySQL, where we see densely accessed regions, a relatively conservative threshold of 27 gives us 100% and 85% of the maximum possible performance gains respectively. In contrast, for Clang the performance gains at threshold 18 is double the gains at threshold 27 because Clang’s access to superpage regions is relatively spread out.

In addition, we find that improving instruction address translation performance is beneficial for data address translation, as the STLB is shared between code and data. Figure 4 shows the normalized cycles spent on instruction and data page table walk at threshold 18. Instruction page table walk cycles goes down as low as only 2% of stock FreeBSD; data page table walk cycles also decrease to as low as 50% of stock FreeBSD.

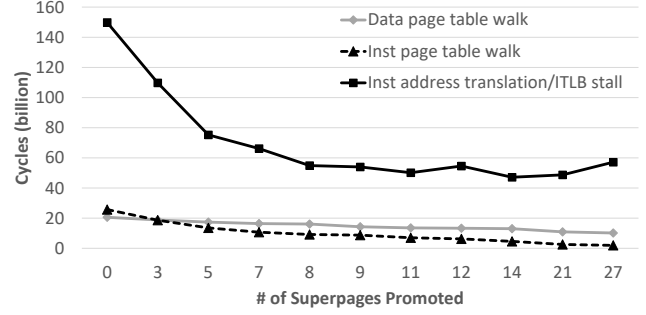


Fig. 5. Clang TLB performance with incrementally more aggressive code superpage promotion

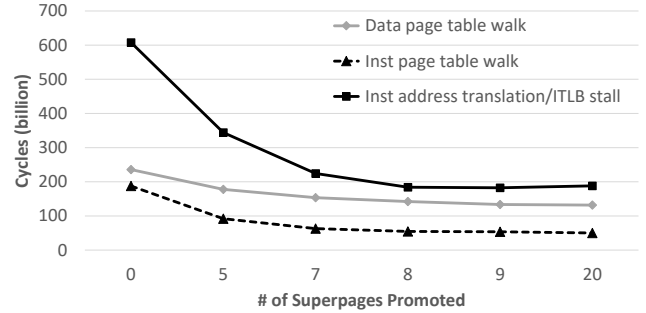


Fig. 6. MySQL TLB performance with incrementally more aggressive code superpage promotion

We present more detailed trend lines for Clang and MySQL. These two applications are particularly interesting because they can use more than 8 superpages for their respective main executable, exceeding the capacity for 2MB entries in the L1 ITLB (Table 1).

Figures 5 and 6 show the way TLB performance changes as we get more aggressive in creating superpages. Each data point corresponds to an upward step in Figure 1. It is clear that both instruction and data page table walk cycles keep decreasing as we increase the number of superpages. The reduction rates gradually drop, reflecting a diminishing margin of return. We note that being more aggressive in code superpage promotion 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 upper-level PTE. This saving scales up with the number of processes.

More interestingly, there is an anomaly in the cycles spent on instruction address translation. The instruction address translation cycles in both applications actually hit optimal value at a small number of aggressively promoted super-

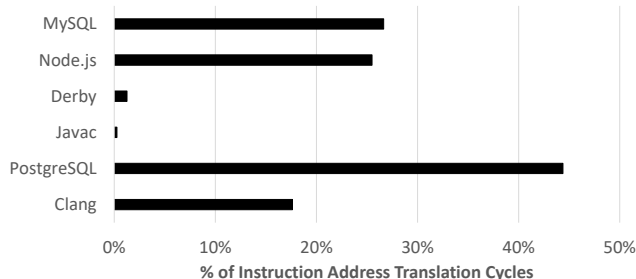


Fig. 7. Percentage of instruction address translation cycles spent on instruction page table walk (stock FreeBSD)

pages. As the number of superpages increases beyond that optimal point, ITLB performance regresses. This is because the ITLB has only 8 2MB entries per thread (Table I). When the number of ITLB entries increases, contention for 2MB slots in the ITLB goes up. Eventually the diminishing returns from more instruction STLB hits is overshadowed by the increasing cost of servicing ITLB misses. Note that despite this performance regression on the ITLB side, overall execution cycles don’t necessarily deteriorate. In particular, both Clang’s and MySQL’s overall performance (as shown in Figure 3) keeps improving even as ITLB performance starts to worsen, thanks to the fact that the second-level TLB is shared between instruction and data. A reduction in the number of ITLB entries frees up slots in the STLB for the data side. Performance gains on the DTLB side then make up for the loss in the ITLB. In conclusion, increasing the number of code superpages will most likely net us an overall performance gain due to reduced contention in the STLB.

Finally, we note that the heterogeneous structure of the ITLB that is the root cause of this anomaly in instruction address translation cycles is an artifact of Intel’s microarchitecture. In particular, AMD’s ITLB does not distinguish between 4KB and 2MB entries [34].

A. Cost within the TLB

A closer look at Figure 5 reveals that instruction address translation cycles can be huge relative to page table walk for instruction access. When we have 0 resident superpages for Clang, instruction address translation cycles is 5.8 times instruction page walk cycles. When we adopt the most aggressive superpage promotion scheme, that ratio increases to 29.3. The large ratios suggest that page table walk is not the predominant cost in some cases. Figure 7 more comprehensively illustrates this point. We find that in most cases, page table walk is less than 30% of instruction address translation cycles.

More interestingly, we find that the end-to-end cost of an ITLB miss can be substantially higher than the 7-cycle cost of hitting in the STLB, as claimed by Intel’s documentation [12]. In fact, when we multiply the number of ITLB misses by 7, the result does not even come close to accounting for the gap between ITLB stalls and instruction page table walk cycles. As it turns out, an in-flight instruction can hold onto

its corresponding ITLB entry and prevent an STLB hit that needs to replace the held-onto ITLB entry from progressing. As a result, the cost of an ITLB miss increases.

We designed a microbenchmark as an extreme to show how much the ITLB replacement cost can potentially increase in a filled pipeline. Essentially, the benchmark program tries to repeatedly stream through a set of 2MB executable mappings. The main program is made of trivial functions that do nothing but increment a local variable and call the next trivial function (with the last one calling the first one and forming a recursion). The functions are mapped into different but back-to-back superpages. As a result, the number of 2MB code mappings is the same as the number of distinct trivial functions. We made sure that within their respective superpages, the functions start at incrementally higher offsets such that collectively the functions would fully fit in the cache.

When the number of distinct functions increases from 8 to 9, we exceed the L1 ITLB’s capacity for 2MB entries (Table I). As a result, instruction address translation overhead increases from virtually 0% of execution time to more than 70%, slowing down overall execution to take 5X more cycles to complete. In the case of 9 2MB mappings (or functions), virtually 100% of the ITLB misses still hit in the STLB. Yet, we calculated the average cost of an ITLB miss to be 28.7 cycles with 9 superpages. Assuming the best-case 7-cycle cost of hitting in the STLB, there would be an average stall of nearly 22 cycles in the ITLB for a replacement to occur.

VI. RESIDUAL MAPPINGS

In this section, we discuss two techniques that deal with situations when a code region is not an integer multiple of the superpage size. *Padding* deals with the residual code region when code size is not an integer multiple of the superpage size (as is dominantly the case). *Page table sharing* deals with shared libraries, which are usually too small to justify the use of superpages.

A. Padding

Code size is rarely an integer multiple of the superpage size. Consequently, executables typically have a residual region at the end that is too small to be mapped using superpages. For example, the residual region of PostgreSQL’s code is 500 pages, sufficiently large to be problematic. Specifically, the number of 4KB page mappings in PostgreSQL’s residual region by itself exceeds the 128 4KB entries in the ITLB, so accesses to more than a quarter of these pages will result in 4KB page mappings spilling into the STLB, perhaps displacing 2MB page mappings for data.

We propose instead to use “padding” in order to allow the residual code region to be converted into a superpage. There are two main ways in which “padding” can be achieved.

- The first one is to modify the linker script so that the linker bloats the executable segment up to the next superpage boundary, and then updates the program header to reflect the padding. In this method, the linker is actually filling extra content (e.g. sequences of no-ops)

into the executable file. When the image activator loads the program, it will be instructed by the updated program header to end the executable mapping at a superpage boundary. Since the “padding” is backed by actual file content, this method requires no kernel modification.

- Alternatively, the kernel can automatically extend the executable mapping up to the next superpage boundary, back it with a physical reservation, and then fill in the gap following residual code with no-ops or zeros. This method requires no linker modification.

We implement the approach where the kernel transparently and automatically handles the padding. In order to reduce the memory consumed by padding, our implementation uses the pages that come after the executable segment (the data segment) in the executable file to pad out the residual code. These pages are mapped twice: once within the residual code superpage (which is read-only by default) and once within the data segment, where they are mapped copy-on-write by default. Extending the residual code mapping to the end of a superpage is made possible because the data segment is separated from the code by 2MB of unused virtual address space (Table V).

B. Shared Page Table Pages

Most libraries are not large enough for executable superpages to apply. Moreover, common libraries are referenced across different processes, as described in Section II-B. This combination of circumstances makes sharing page tables a perfect fit for libraries for two reasons.

- From the perspective of the MMU, shared PTPs work just like ordinary PTPs, meaning that they can map code at base page granularity just as well.
- Shared PTPs can be shared across processes running different main executables. In the best case, only one copy of the PTPs mapping a library’s executable code ever needs to exist in memory. The more processes referencing the library and using the shared PTPs, the more the savings in physical memory.

Every active file has an associated *vm_object* [35] that tracks its resident pages. We attach the shared PTP to the *vm_object*. When mapping a file, its *vm_object* is looked up and if a shared PTP already exists in the corresponding *vm_object*, it is inserted into a process’s higher level entry in the multi-level page table. An unused and non-reserved bit in the higher level entry can be used to mark the presence of a lower-level shared PTP in the page table hierarchy. There are three main code paths through which PTP sharing happens: page fault handling, *exec-time* pre-faulting, and *fork()*.

Although the design is flexible enough to support sharing at all levels of the page table hierarchy, we share only leaf-level PTPs in the current implementation due to orders of magnitude diminishing returns in memory savings and in the ability to share at higher levels. To make sure that we get maximal usage of shared PTPs across processes with different address space layouts, we modified the dynamic linker and the “exec” family

TABLE V
POSTGRESQL MAIN EXECUTABLE LAYOUT IN VIRTUAL ADDRESS SPACE

Region	Default			With in-kernel padding		
	Start	End	Prot.	Start	End	Prot.
Code	0x400000	0x9f4000	r-x	0x400000	0xa00000	r-x
Data	0xbf4000	0xc00000	rw-	0xbf4000	0xc00000	rw-

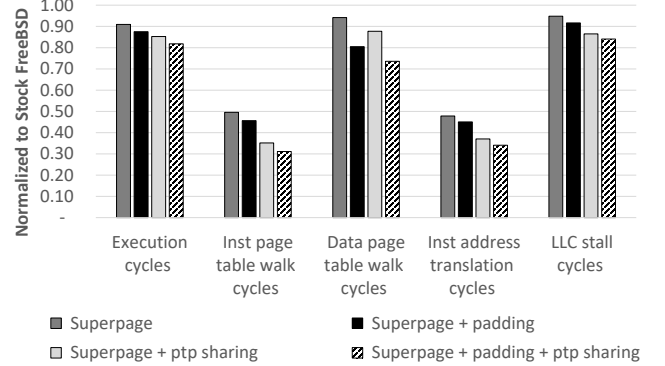


Fig. 8. PostgreSQL user-space results

of system calls to make sure that mappings for text sections start at 2MB-aligned virtual addresses.² In addition, the shared PTPs are managed using copy-on-write (COW). A new copy will be created when the shared PTP is written due to, for example, COW on the underlying shared physical memory, or modification to permissions or mapping in the shared PTP.

This scheme is more general than the one described in the work of Dong et al. [8]: (1) we do not rely on the presence of a template process that has a set of commonly used libraries pre-loaded, and (2) we allow sharing on a per-library basis as opposed to constraining page table sharing to a set of pre-loaded libraries.

C. Evaluation

We use PostgreSQL for our evaluation, since it is the only application that heavily accesses its residual code. Node.js touches only one page in its residual part, while the rest of the applications do not execute their respective residual code at all. Moreover, PostgreSQL is a naturally multi-process application that dynamically links a reasonable number of shared libraries. We apply shared PTPs on libraries. We pad out the main executable, and map it with superpages entirely.

Table V presents snapshots of the layout of the PostgreSQL main executable in its virtual address space before and after padding is applied. After padding, the executable segment’s end address of 0x9f4000 is extended up to 0xa00000. As a result, the executable segment can then be mapped by 3 superpages, avoid hundreds of 4KB mappings.

Figure 8 shows the user-space impact of page table sharing and padding. For the rest of this section, the numbers presented are normalized to stock FreeBSD. The four bars with different

²The requirement of 2MB alignment reduces available bits for address layout randomization (ASLR).

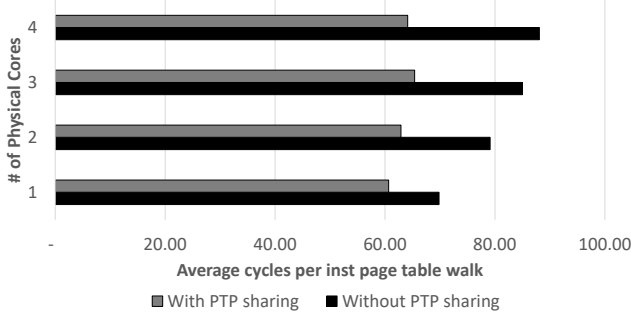


Fig. 9. PostgreSQL average cycles per instruction page table walk with and without sharing page table (with padding and superpages enabled in both cases)

shadings under each counter, from left to right, represent four slightly different configurations. (1) No padding. No page table sharing. Aggressive promotion leads to 2 superpages in the main executable. (2) Padding is applied. No page table sharing. Aggressive promotion leads to 3 superpages in the main executable. (3) No padding. Page table sharing is applied on libraries. Aggressive promotion leads to 2 superpages in the main executable. (4) Padding is applied. Page table sharing is applied on libraries. Aggressive promotion leads to 3 superpages in the main executable.

It is clear that the application sees improvement in TLB performance and overall execution cycles as we pad out the residual code and map it with superpages. Note that in this particular case, improvements brought by padding come at no extra memory consumption since the residual part is only 12 4KB pages short of being a superpage, and so the padding is backed by the data segment. In fact, since an entire PTP is now replaced by a single upper-level PTE, we save 4KB worth of page table memory for each process referencing the residual code. The more worker processes we run in PostgreSQL, the greater the savings.

Page table sharing also improves both TLB performance and overall execution cycles by (1) de-duplicating page table data in the LLC, thereby reducing LLC contention and incurring significantly fewer LLC stall cycles; and (2) incurring significantly fewer data and instruction page table walk cycles, due to more LLC hits when there is a walk. Figure 9 demonstrates the reduced average instruction page table walk cost after we share page tables. The more cores and worker processes, the greater the improvement in cycles per walk.

Figure 10 shows the impact of padding and page table sharing in kernel space. Padding visibly reduces the number of instructions retired and execution cycles, reflecting less work for process creation and teardown as we replace the processing of hundreds of PTEs with just a single upper-level PTE. Page table sharing has a similar effect in reducing OS work. We insert entire shared PTPs directly at process creation (instead of inserting each 4KB mapping individually), and we skip over shared PTPs during process teardown. The effect is evidenced by 10.5% fewer OS-space retired instructions, resulting in a

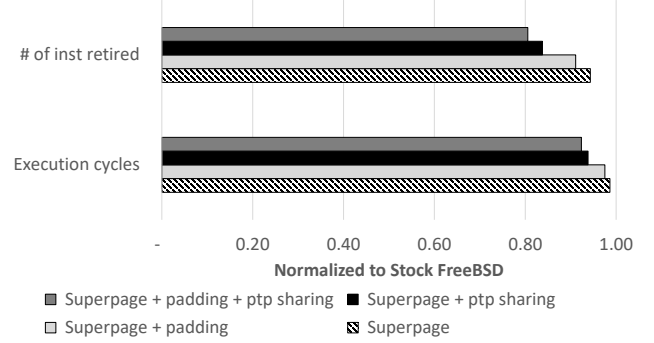


Fig. 10. PostgreSQL kernel-space results

4.8% reduction in OS-space execution cycles.

The rightmost bars under each counter in Figure 8 show the result of simultaneously applying padding, page table sharing, and aggressive superpage promotion. User-space execution cycles reduce by over **18%**. As comparison, instruction address translation cycles accounts for 8.8% of execution cycles on stock FreeBSD. The 18% reduction in execution cycles is more than twice that. This observation directly demonstrates the fact that a 1% improvement in instruction address translation cycles could be significantly *amplified* in overall runtime reduction due to the data side synergistically benefiting from less contention in the STLb and caches.

VII. 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 [7] promotes/demotes superpages based both on the number of physically resident pages and on their access frequency. SmartMD [36] 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 [37]. Illuminator [38] and Gorman et al. [39] improve the ability to allocate superpages by grouping pages that are immovable in order to avoid the possibility of fragmentation. In subsequent work, Gorman et al. [40] provide APIs for applications to request superpage allocation explicitly as in `libhugetlbfs` [41] and evaluate the performance impact of using superpages. Ausavarungnirun et al. proposed a new GPU memory manager that allocates contiguous virtual pages to contiguous physical pages in GPU memory to allow the use of superpages [42].

Compile-Time Optimizations Ottoni et al. explored the performance benefits of using superpages on code using

libhugetlbfs [9], [41]. 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 8 2MB mapping L1 ITLB entries for hot functions. Lavaee et al. [10] 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 [1], [2]. Existing research also improves address translation performance by leveraging memory contiguity and coalescing page translations [3]–[6].

Sharing Page Tables Previous works have focused on sharing page tables for applications handling a large amount of data [43], [44]. Dong et al. [8], [45] shares page tables among Android application processes forked from a template process called *zygote*. In contrast, we implement 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 [46].

VIII. CONCLUSIONS

In this paper, we examine the instruction address translation overhead of six widely used applications, and up to seven different workloads. It is clear that the overhead of instruction address translation for a variety of widely used applications is non-trivial. This overhead increases as the level of parallelism goes up in modern applications and hardware platforms.

Three techniques help reduce instruction address translation overhead and therefore reduce overall execution cycles. First, relaxing the superpage promotion policy for code can reduce execution cycles by as much as 8.5% compared to stock FreeBSD. Second, padding the residual part of code out to a superpage boundary further reduces execution cycles by up to 4% under a sufficiently relaxed promotion policy. Third, for small code regions such as small shared libraries, page table sharing complements superpages by reducing contention over the entire memory hierarchy and by reducing the cost of page

walks. The three techniques combined reduce execution cycles by over 18% compared to stock FreeBSD.

Somewhat unexpectedly, improving address translation performance in the first-level instruction TLB can reduce the address translation overhead for data accesses. This concomitant improvement in data address translation is due to the fact that many modern microarchitectures share the second-level TLB between instruction and data translations.

Finally, we make two observations about the design of the instruction TLB. First, future TLB designs should pay attention to interactions within the pipeline to avoid substantially increasing the cost of first-level instruction TLB replacement. Second, a level-one instruction TLB that has separate capacity for base page entries and superpage entries can see regression in its performance when the number of superpages exceeds the capacity for superpage entries. Fortunately, the regression can be made up for by reduced pressure on the second-level shared TLB and all across the memory hierarchy.

ACKNOWLEDGEMENTS

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

REFERENCES

- [1] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, “Efficient virtual memory for big memory servers,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13. New York, NY, USA: ACM, 2013, pp. 237–248. [Online]. Available: <http://doi.acm.org/10.1145/2485922.2485943>
- [2] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, “Redundant memory mappings for fast access to large memories,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15. New York, NY, USA: ACM, 2015, pp. 66–78. [Online]. Available: <http://doi.acm.org/10.1145/2749469.2749471>
- [3] B. Pham, A. Bhattacharjee, Y. Eckert, and G. Loh, “Increasing tlb reach by exploiting clustering in page translations,” in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 558–567.
- [4] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, “Colt: Coalesced large-reach tlbs,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 258–269. [Online]. Available: <https://doi.org/10.1109/MICRO.2012.32>
- [5] C. H. Park, T. Heo, J. Jeong, and J. Huh, “Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations,” in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 444–456.
- [6] Y. Du, M. Zhou, B. R. Childers, and D. M. R. Melhem, “Supporting superpages in non-contiguous physical memory,” in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 223–234.
- [7] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, “Coordinated and efficient huge page management with ingens,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 705–721. [Online]. Available: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon>
- [8] X. Dong, S. Dwarkadas, and A. L. Cox, “Shared address translation revisited,” in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys ’16. New York, NY, USA: ACM, 2016, pp. 18:1–18:15. [Online]. Available: <http://doi.acm.org/10.1145/2901318.2901327>
- [9] G. Ottoni and B. Maher, “Optimizing function placement for large-scale data-center applications,” in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2017, pp. 233–244.

- [10] R. Lavaee Mashhadi, "Profile-guided memory layout: Theory and practice," Ph.D. dissertation, 2018.
- [11] "Skylake (client) - Microarchitectures - Intel," [https://en.wikichip.org/wiki/intel/microarchitectures/skylake_\(client\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)).
- [12] "Using Intel VTune Amplifier XE to tune software on the 6th generation Intel Core processor family," 2014, https://software.intel.com/sites/default/files/managed/dc/3a/Using_Intel_VTune_Amplifier_XE_on_6th_Generation_Intel_Core_Processors_1.0.pdf.
- [13] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, ser. IISWC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 81–90. [Online]. Available: <http://dx.doi.org/10.1109/IISWC.2011.6114205>
- [14] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 89–104, Dec. 2002. [Online]. Available: <http://doi.acm.org/10.1145/844128.844138>
- [15] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. M. Swift, "Performance analysis of the memory management unit under scale-out workloads," *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–12, 2014.
- [16] "Haswell - Microarchitectures - Intel," [https://en.wikichip.org/wiki/intel/microarchitectures/haswell_\(client\)](https://en.wikichip.org/wiki/intel/microarchitectures/haswell_(client)).
- [17] H. Lu, K. Doshi, R. Seth, and J. Tran, "Using hugetlbfs for mapping application text regions," in *Linux Symposium*, 2006.
- [18] "Multi-process architecture," <https://www.chromium.org/developers/design-documents/multi-process-architecture>.
- [19] "Multiprocess firefox," https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Multiprocess_Firefox.
- [20] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel, September 2016, vol. 3.
- [21] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, April 2018, vol. 3.
- [22] "PMCSTAT(8) FreeBSD System Manager's Manual," 2018, <https://www.freebsd.org/cgi/man.cgi?query=pmcstat&apropos=0&sektion=0&manpath=FreeBSD+11.2-RELEASE+and+Ports&arch=amd64&format=html>.
- [23] "CPUSET(1) FreeBSD General Commands Manual," 2018, <https://www.freebsd.org/cgi/man.cgi?query=cpuset&apropos=0&sektion=0&manpath=FreeBSD+11.2-RELEASE+and+Ports&arch=amd64&format=html>.
- [24] "Dhrystone, The Classic Benchmark," <http://www.ct.se/dhrystone/>.
- [25] "PostgreSQL: The World's Most Advanced Open Source Relational Database," 1996 - 2018, <https://www.postgresql.org/>.
- [26] "pgbench - run a benchmark test on PostgreSQL," 1996 - 2018, <https://www.postgresql.org/docs/9.6/static/pgbench.html>.
- [27] "Openjdk 8," 2014, <http://openjdk.java.net/projects/jdk8/>.
- [28] "SPECjvm2008," 1995 - 2008, <https://www.spec.org/jvm2008/>.
- [29] "SPECjvm2008 Benchmarks," 1995 - 2008, <https://www.spec.org/jvm2008/docs/benchmarks/index.html>.
- [30] "Node.js," <https://nodejs.org/en/>.
- [31] "React server-side rendering benchmark," 2016, <https://www.npmjs.com/package/react-ssr-benchmarks>.
- [32] "MySQL," 2018, <https://www.mysql.com/>.
- [33] "MySQL Benchmark Tool," 2018, <https://dev.mysql.com/downloads/benchmarks.html>.
- [34] "Software optimization guide for AMD family 17h processors," 2017, https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf.
- [35] M. K. McKusick, G. Neville-Neil, and R. N. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed. Addison-Wesley Professional, 2014.
- [36] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. S. Lui, "Smartmd: A high performance deduplication engine with mixed pages," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 733–744. [Online]. Available: <https://www.usenix.org/conference/atc17/technical-sessions/presentation/guo-fan>
- [37] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quema, "Large pages may be harmful on NUMA systems," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 231–242. [Online]. Available: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/gaud>
- [38] A. Panwar, A. Prasad, and K. Gopinath, "Making huge pages actually useful," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 679–692. [Online]. Available: <http://doi.acm.org/10.1145/3173162.3173203>
- [39] M. Gorman and P. Healy, "Supporting superpage allocation without additional hardware support," in *Proceedings of the 7th International Symposium on Memory Management*, ser. ISMM '08. New York, NY, USA: ACM, 2008, pp. 41–50. [Online]. Available: <http://doi.acm.org/10.1145/1375634.1375641>
- [40] —, "Performance characteristics of explicit superpage support," in *WIOSCA 2010 - Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, 01 2012.
- [41] "Huge Pages/libhugetlbfs: 2010," 2010, <https://lwn.net/Articles/374424/>.
- [42] R. Ausavarungrun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "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*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 136–150. [Online]. Available: <http://doi.acm.org/10.1145/3123939.3123975>
- [43] J. Mauro and R. McDougall, *Solaris Internals (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.
- [44] D. McCracken, "Sharing page tables in the linux kernel," in *Linux Symposium*, 2003, p. 315.
- [45] X. Dong, S. Dwarkadas, and A. L. Cox, "Characterization of shared library access patterns of android applications," in *2015 IEEE International Symposium on Workload Characterization, poster paper*, Oct 2015, pp. 112–113.
- [46] M. M. Swift, "Draft : Towards o(1) memory," 2017.