RICE UNIVERSITY

Exploring Superpage Promotion Policies for Efficient Address Translation

by

Weixi Zhu

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

[illegible]

Scott Rixner, Chair
Professor of Computer Science and
Electrical and Computer Engineering

Alan L. Cox

Alan L. Cox
Professor of Computer Science and
Electrical and Computer Engineering

[illegible]

Ang Chen
Assistant Professor of Computer Science

Houston, Texas

November, 2018

ABSTRACT

Exploring Superpage Promotion Policies for Efficient Address Translation

by

Weixi Zhu

The growth of modern applications' memory footprints are rapidly outpacing TLB coverage. Though hardware manufactures provide TLB entries for superpages (mapping much larger areas), existing OS superpage support is either too aggressive and punished for the accompanying overhead or too conservative to exploit more address translation efficiency. This thesis explores the design space of OS superpage promotion policies. A data collection infrastructure is built based on QEMU and works collaboratively with kernel instrumentation to collect memory traces. By exploiting the decoupling between superpage allocation and promotion in FreeBSD, changes of page tables are simulated for different promotion policies without changing their memory allocation. A TLB simulator built by reverse-engineering is validated by empirical results to precisely predict and compare the TLB performance of Intel Skylake processors. Experiments showed that promoting superpages with 1/4 of its constituent 4KB pages utilized can reduce an average of 82% page walk latency compared to FreeBSD's original policy, while introducing negligible overhead because many of such superpages are utilimately mostly utilized. Additionally, the dynamic memory allocator does not manage the memory in a favorable way for aggressive promotion policies, causing them to bring too much overhead.

Contents

Illustrations

Tables

Chapter 1

Introduction

The memory usage of applications has been growing faster than the reach of the address translations that can be cached in the translation-lookaside buffers (TLBs) of modern processors. Therefore, it has become increasingly important to effectively utilize superpages (called huge pages in Linux) in order to increase TLB reach and minimize address translation overheads.

Modern operating systems typically either take a very aggressive or very conservative approach to allocating superpages. For example, the transparent huge page (superpage) support in Linux aggressively allocates huge pages during the first page fault to a 4KB page, if the 2MB huge page containing this 4KB page is avialable. However, superpage support in FreeBSD gradually populates 4KB pages in each 2MB region and defers the superpage promotion until all 512 constituent 4KB pages are populated. While superpages can decrease address translation overheads, they can also increase I/O and other overheads. Therefore, there are pros and cons to both of these approaches.

1.1 Trade-offs between Conservative and Aggressive Superpage Policies

The address translation benefits of superpages are somewhat obvious. A single translation covers a larger region of memory, so the overall reach of the TLB can be increased without needing to increase the number of entries. However, superpages can also incur other overheads. First, if the entire superpage is not accessed after being allocated, then memory would need to be initialized unnecessarily (either by transferring data from disk or zeroing pages). Second, if only part of the superpage is written and becomes dirty, if the page is evicted, the entire superpage would need to be written back to disk, instead of just the smaller, constituent dirty pages. Third, depending on how superpages are managed, they can lead to memory bloat and fragmentation. This occurs either when the entire superpage is not accessed and when superpages are not broken/demoted under memory pressure. Moreover, memory shortage cannot be alleviated by swapping out huge pages, as they are always pinned in memory by Linux. Linux migrates pages in order to mitigate these effects, but the migration itself is costly. Due to these drawbacks, huge page support is frequently disabled on Linux [1, 2, 3].

FreeBSD takes a conservative approach to promoting superpages based on a reservation-based memory allocator. When a superpage-sized region of memory (2MB in X86) is allocated, FreeBSD creates a *reservation* for an aligned region of physical memory. A reservation is a bookkeeping entry which tracks the utilization of the 512 constituent 4KB pages of the 2MB region. Translations are maintained only for 4KB pages until such time as the page is *promoted* to a superpage, whereby the 512 translations are replaced by a single 2MB translation.

On the initial page fault to a large enough memory region, FreeBSD will create a reservation and allocate a single 4KB page within that reservation. Once created, additional 4KB pages can be populated within the reservation as they are accessed. Furthermore, the reservation can be broken at any time to reclaim reserved pages that have not yet been populated. This strategy eliminates page migrations and can reduce memory fragmentation. The reservation is promoted to a superpage only when all 512 constituent 4KB pages have been populated. This reservation-based strategy mitigates, or even completely eliminates, the I/O, page-zeroing, and migration overheads of superpages. However, some superpage benefits are sacrificed because not all reservations can be promoted as superpages.

There is a tension between an aggressive approach, such as that taken by Linux, and a conservative approach, such as that taken by FreeBSD, to superpage management. The more aggressively superpages are created/promoted, the larger the address translation benefits are, as the TLB can cache translations for larger regions of memory. The more conservatively superpages are created/promoted, the fewer additional I/O, page-zeroing, memory bloating, fragmentation and migration overheads are incurred.

1.2 Design Space Exploration

This thesis performs a design space exploration of superpage promotion policies to better understand these trade-offs. Starting with the FreeBSD reservation-based system enables a wide range of policy implementations. Reservations could be promoted immediately, yielding an aggressive policy, or only when fully populated (as done in FreeBSD) yielding a conservative policy, or any time in between. This thesis considers policies along several dimensions, including the number of populated pages, the number of dirty pages, and the lifetime of the reservation.

A trace-driven simulation-based approach is used to compare the trade-offs among different policies. The simulator was validated against Intel's Skylake processors, released in 2015. The validation process itself led to some interesting observations about how the Skylake TLB operates. Surprisingly, the DTLB (L1 data TLB) performance counter does not necessarily count what one might naively expect. Memory references do not block waiting for DTLB misses to resolve, so subsequent accesses to the same page will incur additional DTLB misses if they occur before the initial miss is resolved. In other words, these duplicated DTLB misses could be overlapped but still counted by the performance counter. Furthermore, speculative memory references also incur DTLB misses, leading to far more DTLB misses than one might expect. In contrast, the STLB (L2 TLB) behaves much more predictably. As STLB misses will trigger expensive page walks that are hardly to be overlapped. Speculative executions and memory prefetching are also unlikely to trigger costly page walks, making the STLB behavior more predictable.

Surprisingly, promoting superpages upon reaching a small (aggressive) utilization threshold can achieve competitive performance gain compared with the most aggressive policy, while introducing negligible I/O overheads. The underlying reasons are that utilization has a very strong correlation with the number of memory accesses to a reservation and a reservation is usually either sparsely populated or densely populated. So that a utilization threshold can help target the superpages that cause the most TLB misses and predict future utilizations to avoid promoting underutilized superpages. The exploration also finds it not a good choice to promote reservations by the lifetime of their residency in the memory, as lifetime is shown not to be a good predictor of superpage utilization.

Further, the DTLB performance is less affected by the aggressiveness of the superpage policy, compared to the STLB. Because the DTLB caches fewer entries than the STLB, using superpages for Intel Skylake DTLB can only improve its TLB reach from 128KB to 64MB, which is far smaller than common applications' memory footprints. In other words, it is easy to reach the limit of exploiting the spatial locality for the smaller DTLB.

1.3 Contributions

This work simulates and evaluates different superpage promotion policies within the FreeBSD reservation-based physical memory allocation mechanism. Specifically, we show that we can gain competitive performance compared to the most aggressive policy while introducing negligible overheads.

Our contributions are threefold:

1. Building a data collection infrastructure by modifying QEMU to track memory traces from both kernel space and user space and kernel events collaboratively.

2. Using reverse-engineering to understand and faithfully simulate the TLB performance in Intel Skylake processors. The consistence of our TLB simulation has been validated by empirical results.

3. Exploring the design space of variant superpage promotion policies between aggressive and conservative allocations to discuss the trade-offs.

1.4 Organization

This thesis is organized in the following way. Chapter 2 introduces the background information about x86 address translation, TLBs and FreeBSD's reservation-based memory allocation as well as their related work. Chapter 3 introduces a data collection infrastructure and the employed benchmarks. Chapter 4 discusses the TLB characteristics in Intel Skylake processors and the empirical comparisons between simulation and real-world performance. Chapter 5 introduces the design space of the explored superpage promotion policies. Chapter 6 shows and discusses the empirical results of the trade-offs from the explored policies. Chapter 7 discusses potential future work and concludes this work.

Chapter 2

Background and Related Work

Address translation overhead can seriously slow down modern applications [4, 5, 6], especially for those that have large memory footprints [7, 8]. So the OS and hardware collaborate to support superpages to alleviate such overhead. This chapter discusses both background knowledge and related work that alleviates address translation overhead. Section 2.1 explains how page-based address translation works in the x86-64 architecture. Section 2.2 then describes the organization of the TLB in Intel Skylake processors and section 2.4 provides background information of FreeBSD's reservation-based superpage promotion policy. Discussion of related work is divided into two aspects including both hardware and software. Section 2.3 discusses innovations on hardware designs for accelerating address translation. Section 2.5 discusses how superpages or huge pages are supported on current operating systems. Section 2.3 first discusses how hardware TLBs support multiple page sizes in response to the nature of paging hierarchy, then introduces OS-independent page coalescing and clustering. After that, a segment-based address translation scheme which conceptually employs arbitrarily large and unaligned pages is reviewed as well as previous work on reducing the page walk latency by caching the paging hierarchy or moving the page walk to the non-critical speculative execution path. Section 2.5 reviews how superpages or huge pages are supported in commonly used operating systems including FreeBSD, Linux, Windows and OS X, as well as online promoting techniques for superpage allocations. Section 2.6 summarizes how hardware and software collaborate for faster address translation.

2.1 Address Translation in X86-64

X86 processors store the virtual-to-physical address mappings in hierarchical page tables with a radix tree data structure. The page walker, the hardware in charge of address translation, traverses the page table starting from its root in a top-down manner to find the corresponding page table entry. The top-down traversal is called a page walk. Current x86-64 paging can translate a 48-bit virtual address to a 52-bit physical address, supporting up to 256TB virtual address space [9]. The smallest and most common page size in x86 is 4KB. So, each existing node on the radix tree is stored in a 4KB-aligned page, containing 512 8-Byte entries; nine bits are required to index the 512 entries.

An example of a page walk for virtual address `0x5610E65BC7FE` is demonstrated in fig. 2.1. The virtual address contains four 9-bit indexes (`0x0AC`, `0x043`, `0x132`, and `0x1BC`) for the four levels of the page table, named PML4 (Page Map Level 4), PDPT (Page Directory Pointer Table), PD (Page Directory) and PT (Page Table). A hardware page walker starts the page walk by reading the CR3 register, which points to a 4KB physical page at the PML4 level. A PML4 page contains 512 entries. The page walker then uses the upper 9-bit index in the virtual address (`0x0AC`) to find the appropriate entry in the PML4 page. The PML4 entry points to another 4KB page on the next level (PDPT). This process is repeated 4 times until a valid PT entry is found, which points to a 4KB data page. The physical address is computed by adding the 12-bit page offset (`0x7FE`) into the 4KB data page.

Invalid entries at any level in the page table will trigger a page fault, while valid entries can either point to the next level or a physical data page. Physical pages

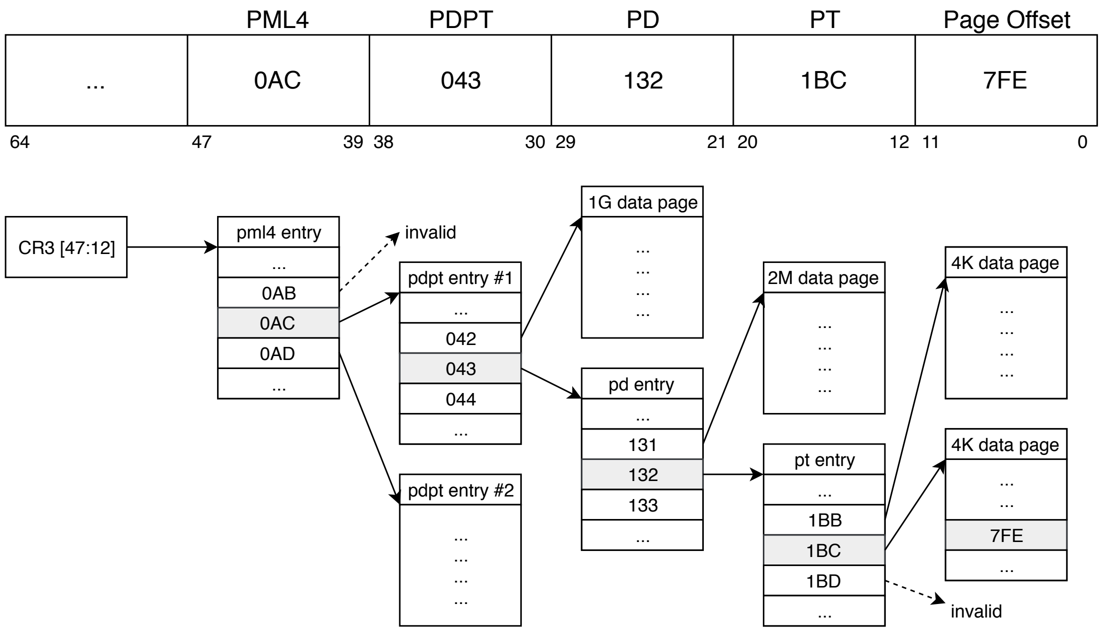
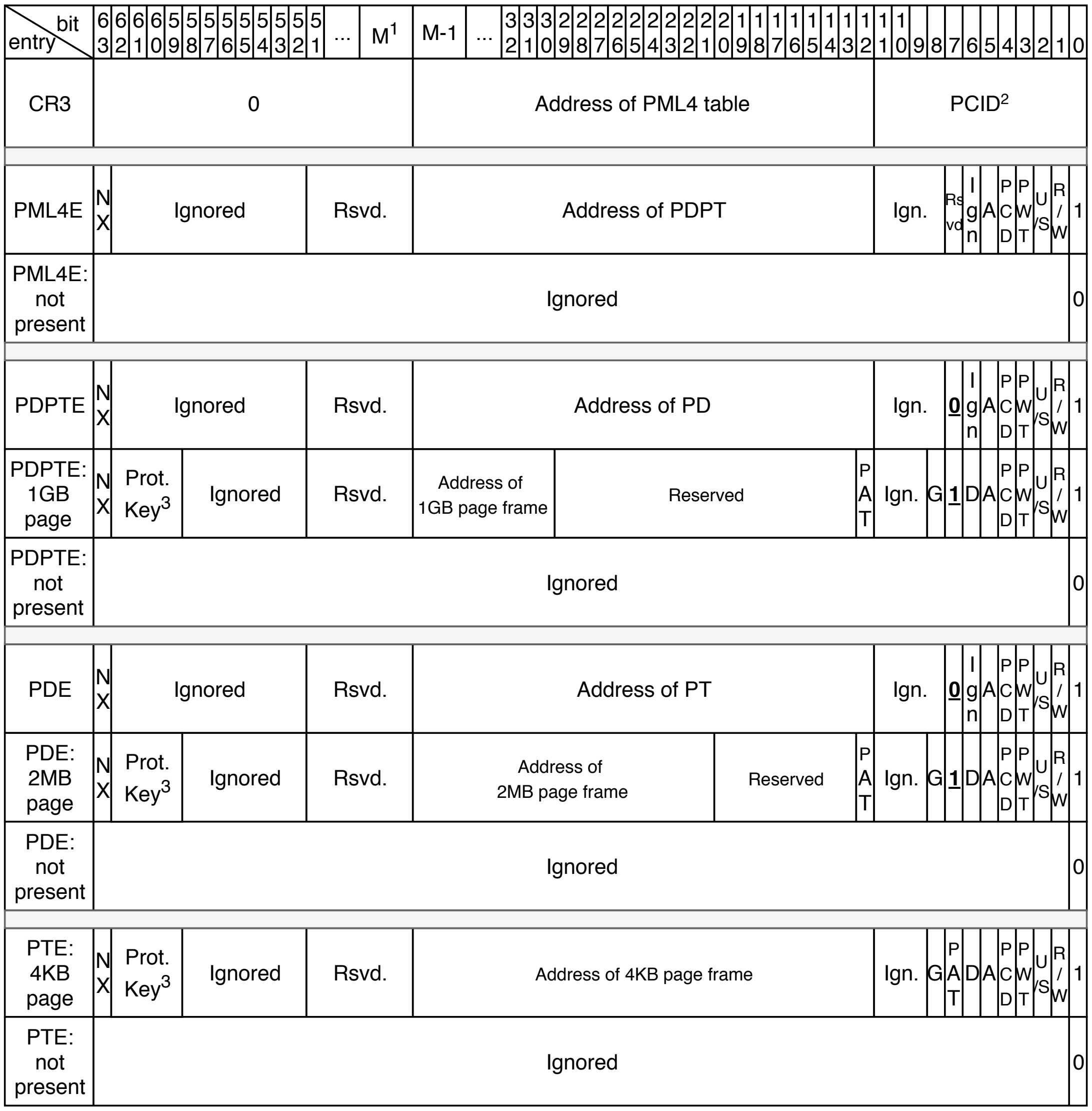


Figure 2.1 : Address translation to a 4KB Page in x86-64 paging (48-bit) for virtual address `0x5610E65BC7FE`

pointed by entries on PT, PD and PDPT levels correspond to 4KB, 2MB or 1GB data pages. Formats of these entries are shown in fig. 2.2. Both 2MB and 1GB pages are called superpages. Unless otherwise noted, however, this thesis will use superpages to mean 2MB pages as currently 1GB pages are infrequently used by applications.

A page walk is expensive because it typically takes 4 memory accesses (3 with a superpage). Even in modern processors with fast and well-designed data caches, the average memory access latency can still be hundreds of CPU cycles. This can seriously slow down programs because the processor is halted to wait for each page walk to be finished. The next section discusses how the TLB works to accelerate address translations.



bit / entry	63	62–59	58–52	51 … M[1]	M-1 … 30	29 … 21	20 … 13	12	11–9	8	7	6	5	4	3	2	1	0
CR3	0				Address of PML4 table				PCID[2]									
PML4E	NX	Ignored		Rsvd.	Address of PDPT				Ign.		Rsvd	Ign	A	PCD	PWT	U/S	R/W	1
PML4E: not present	Ignored																	0
PDPTE	NX	Ignored		Rsvd.	Address of PD				Ign.		**0**	Ign	A	PCD	PWT	U/S	R/W	1
PDPTE: 1GB page	NX	Prot. Key[3]	Ignored	Rsvd.	Address of 1GB page frame	Reserved		PAT	Ign.	G	**1**	D	A	PCD	PWT	U/S	R/W	1
PDPTE: not present	Ignored																	0
PDE	NX	Ignored		Rsvd.	Address of PT				Ign.		**0**	Ign	A	PCD	PWT	U/S	R/W	1
PDE: 2MB page	NX	Prot. Key[3]	Ignored	Rsvd.	Address of 2MB page frame		Reserved	PAT	Ign.	G	**1**	D	A	PCD	PWT	U/S	R/W	1
PDE: not present	Ignored																	0
PTE: 4KB page	NX	Prot. Key[3]	Ignored	Rsvd.	Address of 4KB page frame				Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1
PTE: not present	Ignored																	0

Figure 2.2 : Formats of CR3 and Paging-Structure Entries for 48-bit X86-64 Paging [9]

1. Maximum Physical Address
2. Enabled when CR4.PCIDE=1
3. Enabled when CR4.PKE=1

2.2 Hardware Translation Look-aside Buffers (TLBs)

Translation look-aside buffers (TLBs) cache entire translations, so a hit in the TLB eliminates the need for a page walk. X86 Processors like recent three Intel micro-architectures (Sandy Bridge, Haswell and Skylake) or ARM processors like Cortex-A, all employ a two-level TLB structure. The existence of TLBs changes the procedure for address translation. Before performing a page walk, the processor will first look up the virtual address in the TLB. The TLB works like a data cache but stores virtual address and PTE (leaves of the page table radix tree) pairs. If the TLB has the PTE covering the queried virtual address, called a TLB hit, the PTE is directly returned. Otherwise, named a TLB miss, a hardware page walker will perform a page walk to fetch the corresponding PTE from data caches or the main memory. The physical address will then be computed by concatenating the physical address of the page frame and the page offset. The more TLB hits, the less overhead is induced by address translation.

The page table entries contain many controlling bits besides the physical address of the page frame or next lower-level page table. This work specifically pays attention to the read/write bit (#1 R/W), the access bit (#5 A) and the dirty bit (#6 D) in Figure 2.2. A physical page is read-only if the read/write bit is unset and writable if set. The access bit is always unset when the page is initialized or paged in from disk during page fault handling. Upon first touch (which must miss in the TLBs), a page walker will perform a page walk, set the access bit of this entry and insert it into the TLBs. Dirty bits are maintained by the TLBs. Any modifications on a clean page (dirty bit unset) will trigger another page walk, even if there is already an entry in the TLB. The page walker will set the dirty bit in the page table, then either insert or update the cached entry in the TLBs.

Similar to data caches, there are usually multiple levels of TLBs in processors and each core has its own dedicated TLBs. A TLB may be fully-associative or set-associative. In a k-way set-associative TLB for example, the input virtual page frame will be split into *tag* bits and *hash* bits (usually the least significant part to benefit from spatial locality). The *hash* bits are used to compute the block index. Inside the selected block, k tags will be compared simultaneously with the input *tag* bits. If none of them match, the page walker will do a page walk to insert the (*virtual address*, *page table entry*) pair. The TLB entry with the least priority will be evicted to make space, if all TLB entries are occupied. This procedure is called a replacement in the TLB. The most common TLB replacement policy is the least recent used (LRU) algorithm [10]. By evicting the least recently used tags, the LRU policy holds the most recently used entries in the cache to take advantage of temporal locality.

To serve as a fast cache, the TLB is usually implemented by content-addressable memory (CAM). CAM allows searching the entire memory in a single operation at a very high speed and returns the content if hit. In a fully associative TLB, a search compares all of its entries in parallel. This requires circuits and logic gates between all storage bits and the input bits, bringing thermal issues. A TLB must also be located in the very narrow space between the CPU and its caches in order to translate the virtual address before it fetches data or instructions from the cache. Both spatial and thermal issues make hardware vendors judicious and cautious about the TLB size. The size of TLBs has grown a lot over the decades, e.g. 1,536 second-level TLB entries in modern CPUs like Intel Skylake. However, the memory capacity has grown much faster as common workloads consume orders of magnitudes more memory.

Memory virtualization provides address space isolation among processes. A process should never be allowed to use the virtual address mappings in other processes' address space. Therefore, a TLB must completely flush all of its entries when a context switch happens or the CR3 register is loaded. Intuitively, this increases the TLB misses in the future. The kernel page tables are shared among all processes. And a global bit (#8 G in Figure 2.2) is used in TLBs to avoid kernel mappings being flushed when switching between kernel mode and user mode. Process-context Identifiers (PCIDs in Figure 2.2) were later introduced to allow the isolated virtual mappings to coexist in the TLBs. They work together with virtual addresses to serve as the look-up keys. With PCIDs, TLBs do not need to be completely flushed upon context switch or CR3 loading.

2.2.1 TLBs for Multiple Page Sizes

Modern x86 processors have TLBs supporting multiple page sizes in response to different available page sizes in x86-64 paging. There are 2 types of hardware designs. One uses separate TLBs where each one caches mappings for a dedicated page size. The other one employs a unified TLB design where each entry can cache mappings for 2 or more page sizes. Skewed TLBs or TLBs predicting page sizes are explored by academia but have not been employed by hardware vendors because of either complex hardware design or energy issues [11, 12, 13, 14]. This thesis assumes that for each level of the TLBs, the separated TLBs or the unified TLB is queried serially in a descending order of available page sizes.

2.2.2 TLBs in Intel Skylake Processors

This work focuses on TLB performance of Intel Skylake x86 processors released in 2015. Until 2018, most Intel processors used the Skylake microarchitecture whose TLBs work in the same way. To isolate the TLB performance of memory accesses,

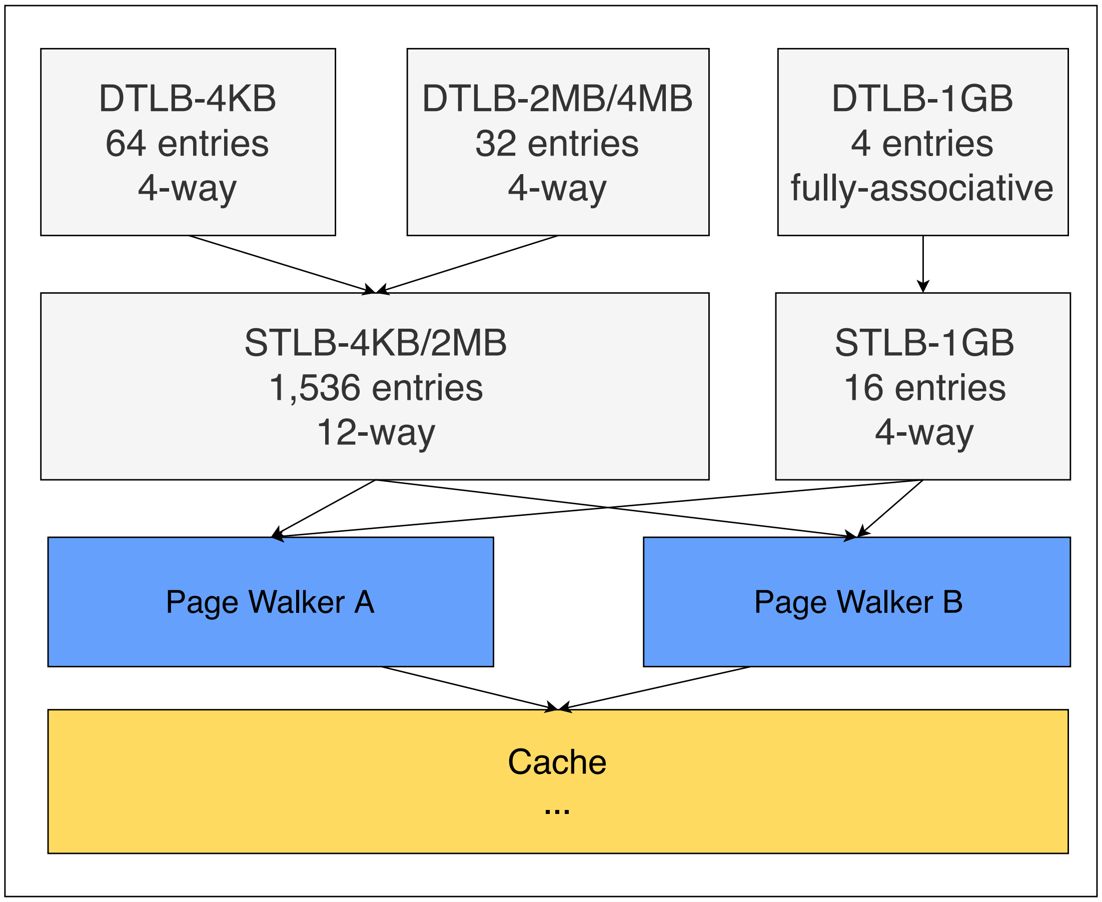


Figure 2.3 : Diagram of TLBs on one core of Intel Skylake CPU, excluding instruction TLBs.

Level 1	
DTLB-4KB	64 entries, 4-way set associative
DTLB-2MB/4MB	32 entries, 4-way set associative
Level 2	
STLB-4KB/2MB	1536 entries, 12-way set associative

Table 2.1 : TLB components on one core of an Intel Skylake processor. TLB components for instructions or 1GB pages are ignored.

three of the Skylake TLB components in table 2.1 are considered, including two L1 DTLBs (data TLBs) split for 4KB and 2MB mappings and one second-level STLB (shared TLB). Note that there are both DTLB and STLB componenets for 1GB pages, but ignored by this work because 1GB pages are not used frequently in the user space. The STLB is unified for both 4KB and 2MB mappings. Figure 2.3 depicts an abstract diagram of Intel Skylake data TLBs (instruction TLBs are ignored) in one core.

By using superpages, i.e. 2 MB or 1 GB pages, the TLB entry is allowed to cover a larger memory region. For instance, the STLB in Skylake processors can extend its TLB reach from 6MB to 3GB.

2.3 Hardware Designs for Accelerating Address Translation

As the TLB locates on the critical path of execution, its size grows slowly overtime [9]. Limited TLB size with page-based address translation restricts TLB reach. TLB reach, as the name implies, is defined as the memory coverage by summarizing all its entries. Intuitively, larger TLB reach can lead to less TLB misses. Motivated by enlarging the TLB reach, people have explored TLBs supporting multiple page sizes, TLB coalescing and TLBs supporting arbitrary large mappings [11, 15, 16, 17, 12, 18, 19, 20, 7, 21, 22]. Besides enlaring the TLB reach, people have also explored caching and speculating techniques to reduce page walk latency [9, 8, 23]. Though these related TLB innovations all contribute to efficient address translation, this work mainly focus on TLBs supporting multiple page sizes, which are the most commonly used hardware in commercial processors.

2.3.1 TLBs Supporting Multiple Page Sizes

To serve multiple page sizes in paging, modern TLBs are designed to support various page sizes. TLBs in commercial processors usually contain several peer-level components dedicated to different page sizes, or use a unified design supporting various page sizes for each entry. Because a TLB has no idea which page size is used for a given virtual address, there would be multiple possible page frame numbers. These page frame numbers must then be compared within all TLB ways.

A naive solution is called hash-rehash [14]. The TLB is probed for all possible page sizes, with a different hash function for each time. Concurrent look-ups can be available as long as the TLB performs probing for different page sizes simultaneously, however complicate the port utilization. TLBs supporting 2 page sizes, like the STLBs in Intel Skylake CPUs, may use this approach [14].

Skewed TLBs reduce comparisons by restricting the {page size, virtual address} space on each TLB way [11, 15, 16, 17]. For any given virtual address *va*, it can be cached on each way E with a unique page size S. In other words, the page size S is inferred and determined given *va* and E. Therefore, the total look-ups on a skewed TLB is reduced by a factor of the number of possible page sizes, compared with a set-associative TLB. Skewed TLBs are able to concurrently support multiple page sizes. However, skewed TLBs suffer from the complex implementation. As indicated above, there must be multiplexers to select hash functions for each way. This adds difficulties to the latency management, and costs more energy [14]. The replacement policy would also be relatively complicated [17]. Meanwhile, the effective ways possible to cache a certain mapping is reduced by a factor of number of available page sizes, compared with set-associative TLBs. That is to say, it sacrifices the chip space to achieve concurrent support. Because of these reasons, no commercial processors are known to employ this design.

Papadopoulou et al. proposed a superpage prediction mechanism for TLBs to reduce multiple look-ups [12]. This mechanism predicts which page size to be probed first in the TLB lookup order. Correct prediction which hits the TLBs will stop the remaining TLB probes, saving both TLB hit latency and look-up energy. They mainly focused on a binary predictor to distinguish between base page and moderate-sized superpages, regardless of the page sizes to cater to the TLBs supporting 2 major page sizes in commercial processors [9]. The prediction mechanism takes advantage of either PC counters in x86 architecture [9] or source registers which dominate the virtual address computation in SPARC architecture [12]. Both of them can be available ahead before the TLB look-up, so that the binary predictor can use such values to index in a pattern history table to determine which page size to be probed before the TLB lookups or even in parallel with virtual address computation. Their binary predictor mimics the design of branch predictors [24]. And the worst misprediction rate is reported to be 1.2% in Canneal benchmark [12].

This thesis explores how superpages affect the TLB performance with naive hash-rehash solution, as these TLBs are immediately available on existing commercial processors. Other innovations on TLBs are not currently available, therefore out of the exploration.

2.3.2 TLB Coalescing

TLB coalescing has been explored to increase the effective TLB reach without any OS support [18, 19]. A cache line is usually 64 Bytes which can contain 8 64-bit mappings. Before a page walker inserts the page table entries into the TLB, it examines whether these mappings are contiguous and share identical controlling bits. Once this combination logic satisfies, these entries may be compressed and inserted as one single TLB entry. Pham et al. observed rich compressing opportunities reported in [18]. By exploiting the abundant address mappings, TLB coalescing can boost the TLB reach by an order of 10 (8-16) and also provide a subtle form of TLB prefetching [14]. Commercial processors like AMD Ryzen CPUs have employed TLB coalescing [20].

2.3.3 Arbitrary Large Mappings (Segments)

Basu et al. proposed direct segments for large memory server workloads [7]. A direct segment refers to a virtual mapping between contiguous virtual and physical memory regions, not necessarily aligned to any page boundary. They specifically focused on memory intensive big-memory workloads including key-value stores, databases, graphic computing and HPC (high performance computing) workloads. As studied in [7], these workloads suffer heavily from TLB miss penalties while rarely utilize swapping, copy-on-write or fine-grained memory protections. This motivates a direct segment hardware containing 3 registers: start/end address of the virtual region and a shift value from the virtual to the physical region. TLB misses can be mostly eliminated for big-memory server workloads by looking at the 3 direct segment registers before probing the TLB. The mapped memory region is called a primary region sharing uniform memory access permissions. However, this approach is not a general solution for address translation, and requiring software support to assign the primary region. Except for big-memory workloads, Grandhi et al. applied direct segments on VMMs to reduce the VMM address translation overhead to near or even better than native [21].

Karakostas and Gandhi et al. extended direct segments to a more general solution, called redundant memory mappings [22]. Instead of using a single big direct segment, they break the single large segment into several smaller segments (they call them ranges) to support workloads other than large-memory server workloads. Meanwhile, they propose a hardware range TLB and a software range table together as a supplementary address translation scheme besides standard paging. To be transparent to workloads, they explored modifications on the operating system to exploit ranges automatically, or use eager paging to pre-populate the holes in order to create ranges.

2.3.4 Caching and Speculating by Exploiting the Paging Hierarchy

Both Intel and AMD use MMU caches to accelerate page walks [9, 8]. The AMD processors use a unified page table cache (tagged with physical addresses) while the Intel processors use a split translation cache (tagged by virtual addresses) [9, 8]. Barr et al. later explored the full design space of MMU caches including 5 types differentiating on whether physically or virtually tagged, unified or split and caching the translation path or not [23]. They reported translation cache, which is defined to cache the virtual indices from different levels of page table entries, to be a better design. Specifically, unified translation cache or translation-path cache are the preferred designs. Despite of the various design choices, the MMU cache aims at reducing the page walk latency. Hits in MMU cache can help skip some steps in a page walk (typically contains 4 memory accesses) and therefore accelerate page walks.

In 2011, Barr et al. explored SpecTLB to speculate address translations [4]. This device can interpolate 4KB address translations for TLB misses and confirm the interpolations in parallel with speculative executions. Correct interpolations can remove the TLB miss handling to the non-critical execution path and hide the expensive page walk penalties. However, their interpolation depends on a reservation-based physical memory allocator, as discussed in section 2.4. A 2MB reservation in FreeBSD may not be promoted but contain many 4KB base pages. They exploit this feature to interpolate 4KB mappings, as long as the reservation exists and the speculated pages are already faulted. Their simple prediction heuristic, which examines if the 4KB mapping belongs to a group of contiguous mappings by checking whether (virtual address)[20:12] equals (physical address)[20:12], resulted in very low misprediction rate [4].

This work also discusses how the page walk latency is affected by superpages. As mentioned in section 2.1, page walks fetching a superpage entry will take one less memory access than normal 4KB page walk. Microbenchmarks estimate the average page walk latency to be 21 and 35 cycles for 2MB and 4KB pages, both of which are applied to compare the CPU stall cycles for page walks among different superpage promotion policies.

2.4 FreeBSD reservation-based superpage promotion mechanism

FreeBSD avoids allocating underutilized superpages by decoupling superpage allocation from promotion with a reservation system [25]. A reservation is a bookkeeping entry created to track the utilization of 512 4KB pages in a 2MB-aligned memory region. The 2MB memory is reserved for some specific virtual memory object which is able to consume all of it. For example, reservations will not be created for a memory object mapping a 1MB disk file. Figure 2.4 shows the lifetime of a reservation. During a page fault, FreeBSD tries to allocate a 4KB page inside an existing reservation, or creates a reservation if not found. 4KB pages can be gradually populated in a reservation, so reservations can be categorized as either partially or fully pop-

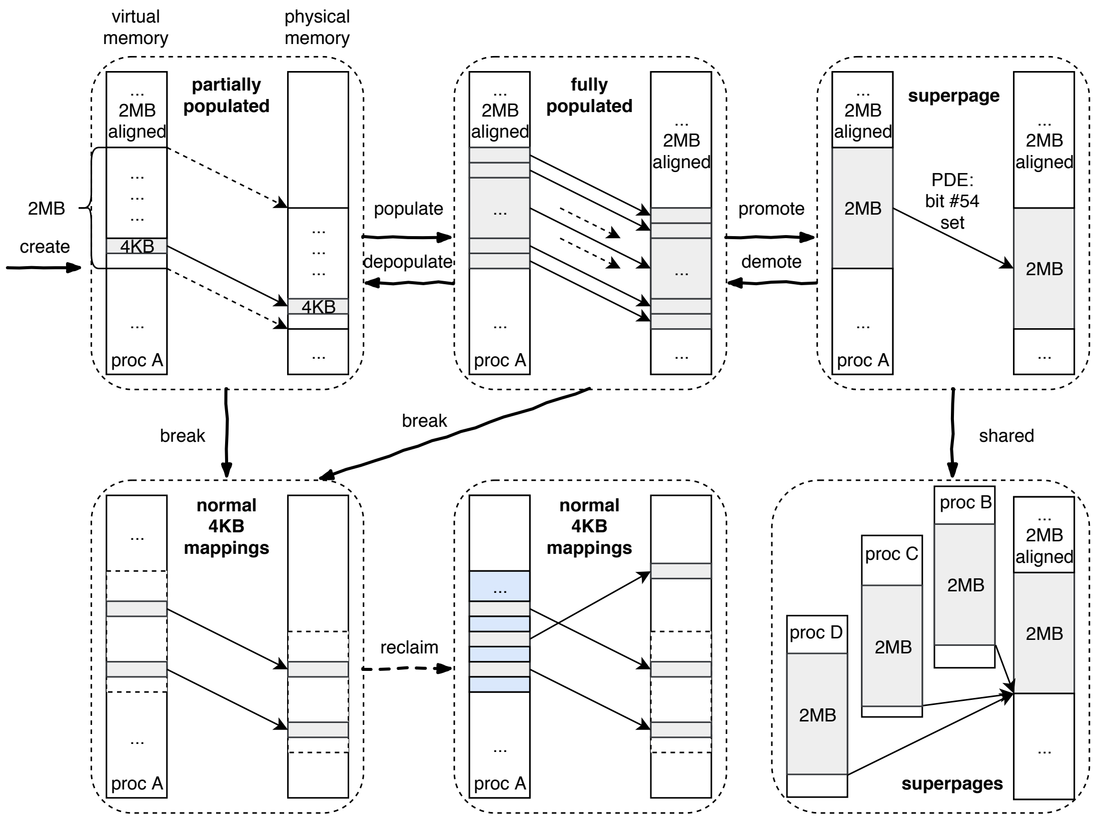


Figure 2.4 : FreeBSD's superpage reservation timeline. Bit #54 in PDE is set by the OS denoting a promoted PDE, which is ignored by the TLB.

ulated. Free pages in partially populated reservations can be reclaimed by breaking the reservation.

FreeBSD only promotes fully populated reservations to superpages. This prevents allocating underutilized superpages. However, to promote a fully populated reservation, FreeBSD requires the 12 control bits of all 512 constituent translation entries to be identical. These 12 control bits are the page metadata, including memory protection, caching and ring privilege, OS-defined management, page reference and modification. The legitimacy of superpage promotion is guaranteed by the equality of 10 bits except for the reference bit and the dirty bit. Requiring the equality of reference and dirty bits helps eliminate extra I/O overhead. The former one avoids any extra page zeroing or disk reads upon promotion, while the later one prevents any extra disk writes when the pages are evicted.

However, one can still dirty a superpage by modifying only one internal 4KB page. To prevent such a situation, FreeBSD additionally write-protects clean but writable superpages. Any modification to a clean superpage will demote it back to 4KB pages. The precondition of the dirty bits equality then requires all 4KB pages to get modified for a second promotion. Therefore, using superpages in FreeBSD incurs no more I/O overhead than using 4KB pages only.

A dirty superpage may also be strategically demoted by jemalloc(). When a program wants to free memory via jemalloc(), jemalloc() does not free pages directly but use MADV_DONTNEED via madvise() system calls instead. If a superpage is involved, FreeBSD will defer the page reclamation by demoting the superpage and marking the first 4KB page unmodified. Once used again and the first 4KB page is modified, the superpage can be promoted back quickly. In this work, this feature is disabled by setting ”lg_dirty_mult:-1”.

2.5 OS Supports for Superpages

Superpages or huge pages are natural results from the hierarchical page tables. The usage of superpages can increase the TLB reach thus motivating the corresponding OS supports [26, 27, 28, 29, 25, 30]. However the OS support for superpages is not trivial because using superpages will introduce problems like memory fragmentation, long page fault latency or memory bloating [31], which can counter-intuitively degrade performance. This section discusses the current OS superpage supports on common operating systems including Linux, FreeBSD, Windows and OS X.

2.5.1 FreeBSD Reservation-based Superpage Support

In 2002, Navarro et al. firstly designed and implemented transparent superpages in FreeBSD [25], using a reservation-based memory allocation mechanism [26]. It reserves an aligned and contiguous superpage (normally 2MB) block in physical memory, populates internal 4KB pages gradually and tries to promote a fully populated reservation to a superpage eventually. This conservative approach successfully avoids any extra I/O cost. It usually gains performance benefit exceeding 30% and introduces negligible overhead in an adversarial benchmark [25]. Their implementation was later merged into FreeBSD's mainline. More details have been discussed in Section 2.4.

This work explored opportunities to use superpages more aggressively in FreeBSD. As mentioned in section 2.4, promotions of superpages are deferred until fully population of reservations, leaving space for early promotions. At the same time, most frequently used reservations are observed to get fully-populated eventually, motivating the exploration of capturing early promotion opportunities. Since FreeBSD promotes superpages judiciously to prevent any extra I/O cost, this work explored trade-offs between TLB performance benefit and potential I/O cost penalty under different superpage promotion policies.

2.5.2 Linux Huge Page Support

Mainline Linux provides both persistent huge pages and transparent huge pages for anonymous memory [32, 29]. Their details are elaborated as follows.

Linux Persistent Huge Pages

Linux supports "persistent" huge pages [32]. The word "persistent" refers to the feature that "Huge pages cannot be swapped out under memory pressure" [32]. The persistent huge pages break the demand paging mechanism. In other words, administrator must pre-allocate a huge page pool, with a designated number of huge pages. Users with certain privileges can allocate memory with these persistent huge pages via mmap system calls, or shared memory system calls. To use mmap system calls, either a filesystem named hugetlbfs must be mounted or the MAP_HUGETLB flag should be used.

Linux Transparent Huge Pages

Linux also supports transparent huge pages [29]. Unlike the persistent huge pages, transparent huge pages do not require pre-allocations from users. Linux provides three modes of the transparency for anonymous memory, including "always", "madvise" and "never". The "always" and "never" modes are self-explanatory, while the "madvise" mode only use huge pages for the memory region mapped by mmap system calls with MADV_HUGEPAGE advise. Linux tries to allocate huge pages in page faults if in "always" mode or in MADV_HUGEPAGE regions. However, the allocation might not succeed immediately, because the required large aligned and contiguous free memory may not be available due to memory fragmentation. Linux has 2 mechanism to deal with the huge page allocation failure. In “always” mode, Linux will migrate pages and compact the memory in order to reclaim huge pages. Note that the page migration and memory compaction are happening inside the very page fault, which will effectively stall the application because of the long page fault latency. In “defer” mode, such failure will only kick the kswapd in the background to reclaim pages as well as kcompactd to compact the memory. Another background daemon called khugepaged will install huge pages later. The “defer” mode avoids the long page fault latency issues by gracefully falling back to use 4KB pages for the allocation failure and postpone the allocation of huge pages. However, transparent huge page support is not aware of whether a huge page will be used frequently. Thus if a 2MB huge page is only touched with 1 byte, the extra effort of allocating a huge page can degrade the performance. Real-world benchmarks usually sugguest disabling Linux transparent huge pages [3, 1, 2, 33]

Ingens explored the tradeoffs between performance and memory savings as well as fair promotions for transparent huge pages in Linux [30]. They borrowed the idea of population bitmaps in FreeBSD’s reservation mechanism to track the spatial utilization of huge pages [25]. Suggested with a 90 percent spatial utilization, Ingens will only promote a huge page after over 90% 512 4KB pages are faulted. Resulting in a small performance degradation compared with Linux’s vanilla transparent huge page support, they achieved good trade-offs between performance and memory bloating [30]. Ingens also exploits the idle page tracking feature [34] in Linux to gather the temporal utilization of huge pages [30]. It then models the huge page allocation priority with the “idle huge pages” for each process. Fair performance was reported on Canneal benchmarks [30].

Some policies explored in this work borrowed the design philosophy of both vanilla Linux transparent huge pages and Ingens. Policy *greedy* promotes a FreeBSD reservation to a superpage upon the first touch, which corresponds to the "always" mode. At the same time, policy *pop-461* and *dirty-461* borrow the 90% spatial utilization in Ingens, as 461 is 90 percent of 512.

2.5.3 Persistent Superpages in Windows and OS X

Microsoft supports large pages in 64-bit Windows [35]. Similar with Linux persistent huge pages, large pages in Windows are not pageable, i.e. pinned in physical memory. In addition, they are always read/write. To use large-page support in Windows, administrator should enable the "SeLockMemoryPrivilege" privilege. The users then need to include the "MEM_LARGE_PAGES" flag in virtual memory allocators. Windows do not deal with memory fragmentation, but recommend users allocate all large-pages on start time [35].

OS X supports 2MB superpages on X86 architecture [36]. Similar with both Linux persistent huge pages and Windows large-pages, the superpages on OS X are always wired. Programs can allocate superpages for anonymous memory in OS X by using the "VM_FLAGS_SUPERPAGE_SIZE_2M" flag in mmap system calls.

2.5.4 Profiling-based Online Superpage Promoting

Online promotion is a profiling-based superpage promoting strategy [27, 28, 37]. The TLB misses are profiled or sampled to help the OS or agent decide whether or not to allocate superpages. However, this kind of approach is not hardware independent. It requires both hardware counters and performance parameters, which may vary on different hardware, to tune the decision model.

2.6 Summary

One common solution to try to mitigate the overheads of address translation is to use superpages.

There are several benefits of using superpages. First and intuitively, there will be fewer TLB misses. The TLB reach, defined by summing the page sizes of all TLB entries, can be enlarged by a factor of 512 if all TLB entries use superpages instead of 4KB pages. With larger TLB reach, it is less likely for a memory reference to miss in the TLBs. Second, there will be fewer memory accesses to perform a page walk, e.g. 1 less memory access for a superpage. Third, there will be less work than creating lower level page table pages. Using a superpage can save both the effort and memory space of creating 512 PT entries. Last, there will be fewer page faults. Up to 511 page faults can be saved in the ideal case where all 4KB pages are accessed. The last benefit does not occur in FreeBSD's superpage support but is possible in Linux [29].

However, the benefits do not come for free. Hardware TLBs effectively support superpages, but there are pitfalls for the OS virtual memory management system. First, using superpages loses fine-grained memory controls. Because the TLBs maintain dirty bits, a modification on any single byte of a 2MB page will set the dirty bit. If the dirty 2MB page ultimately has only one internal 4KB page modified, extra I/O overhead can be incurred, which could easily surpass all address translation benefits. Second, allocating superpages requires 2MB aligned and contiguous memory blocks. As workloads keep running, the memory gets more fragmented and such memory blocks would be more and more scarce. The operating system has to pay extra effort to create available resources for future superpage allocations, like page migration in

Linux [29]. However, the population process in FreeBSD explained in Section 2.4 can alleviate the memory fragmentation issue without actively migrating pages. Superpage policies that do not rely on reservation-based physical memory allocators would also induce page fault latency and memory bloating issues.

While hardware innovations focus on increasing the TLB reach or reducing the page walk penalty, software innovations explore the OS support for superpages. Novel hardware designs, like direct segments, also require the OS modifications to adapt to their new address translation mechanism. Their ultimate goal is to improve address translation performance. This work, however, focuses on how to make the TLB performance better on existing hardware by exploring the design space of superpage promotion policies. By using a data collection infrastructure and a faithful TLB simulator, it is able to quickly examine a large variety of promotion policies without implementing them.

Chapter 3

Methodology

In order to explore a large number of superpage promotion policies, memory traces and address translation kernel events were collected from a real system and used to drive a trace-based address translation simulator. The simulation infrastructure consists of three components: kernel instrumentation, memory access tracing, and TLB simulation. Since FreeBSD's reservation system decouples superpage promotion from allocation, new promotion policies can easily be simulated upon this flexible substrate without perturbing the rest of the system. For the duration of a reservation, it is guaranteed that no other allocation can interfere with that reservation. So, pages can safely be promoted or demoted at any time during the lifetime of the reservation. Furthermore, changing the promotion behavior does not induce any additional memory bloat or fragmentation.

This chapter discusses the methodology applied to simulate different superpage promotion policies in FreeBSD. Section 3.1 explains why instrumenting the FreeBSD kernel is necessary. Section 3.2 discusses how a modified QEMU traces the memory references as well as kernel instrumentation. Section 3.3 discusses the reservation status simulation and TLB performance simulation for different superpage promotion policies.

3.1 FreeBSD Kernel Instrumentation

The kernel events of managing reservations (creating, breaking or freeing) are instrumented to acknowledge the simulator of existing reservations. Since new policies can promote superpages at different times, the original superpage demotion events are no longer sufficient to guarantee the safety of using superpages. So changes of memory protection or mappings are instrumented to detect superpages that must be broken. Last, TLB invalidations are instrumented to help with the TLB simulation. The promotion policies are simulated synchronously in order to provide up-to-date page tables for the TLB simulation, which processes memory access streams.

Instrumented kernel events are encoded with a series of memory accesses, which will visit certain static variables predefined in the kernel space. Their virtual addresses can be probed with SYSCTL system calls. Therefore, the TLB simulator can parse the memory accesses to know which kernel events are happening at the exact time, so that it can jump to maintain the page table changes or determine superpage promotions. Table 3.1 describes all of the kernel events instrumented in FreeBSD.

3.2 Memory Access Tracing by QEMU

To simulate different promotion policies, full-system memory accesses are traced on a data collection infrastructure built upon QEMU (Quick Emulator) [38]. Load and store instructions are intercepted so that they can be traced, along with their associated page table entries. Furthermore, changes to the CR3 register are traced to allow the TLB simulator to emulate PCIDs. The traces are compressed using LZ4 data compression to manage their size [39]. Common binary instrumentation tools, like Intel Pin [40], however do not support kernel space memory reference tracing and are

Kernel Events	Description
RV_CREATE	create a reservation
RV_BREAK	break a reservation
RV_FREE	free a reservation
VM_PROMOTE	promote a reservation
VM_DEMOTE	demote a reservation
VM_MAPPING	share a superpage mapping
VM_UNMAP	munmap() system call
VM_PROTECT	mprotect() system call
VM_PAGEOUT	page laundering
INVLPG	TLB invlpg
INVLTLB	TLB flush
INVLTLB_G	TLB global flush
INVPCID	TLB PCID flush

Table 3.1 : Instrumented Kernel Events in FreeBSD

not maintained for FreeBSD platforms.

3.2.1 TCG

QEMU uses TCG (Tiny Code Generator) as a binary translator to emulate the guest machine. It translates the instructions on target architecture (the emulated processor) to an intermediate representation called "tcg ops". The tcg ops are later translated to instructions on the host architecture, which runs QEMU. The translation is performed in a batch mode. Each batch is called a translation block (TB), containing several

Benchmark	429.mcf	gcc	605.mcf	lbm	omnetpp
Touched reservations	1142	4108	2443	1796	208
Memory accesses (10^{11})	4.87	3	5	2	2
Benchmark	cam4	deepsjeng	roms	xz	canneal
Touched reservations	698	3561	7245	7873	446
Memory accesses (10^{11})	2	2	2	2	3
Benchmark	derby	graphchi	gups	postgresql	redis
Touched reservations	528	481	2270	1149	3533
Memory accesses (10^{11})	1.18	2	0.84	1.74	2.99
Benchmark	svm.mnist	tradesoap			
Touched reservations	359	591			
Memory accesses (10^{11})	3	1.6			

Table 3.2 : Details of 17 benchmarks used for evaluation

instructions on the guest architecture. They are translated to instructions on the host architecture and then executed. Translated TBs will be cached by QEMU so that duplicate translations can be avoided.

Figure 3.1 demonstrates a flowchart of TCG. TCG always tries to go through the fast path where a valid TB already exists in the buffer. If not found, it then performs the 2-step translation as discussed above and inserts the generated TB into the buffer.

3.2.2 QEMU Memory Tracing

QEMU itself supports memory tracing. The vanilla memory tracing is only available when it runs in full-system emulation mode, using TCG. The trace event for memory

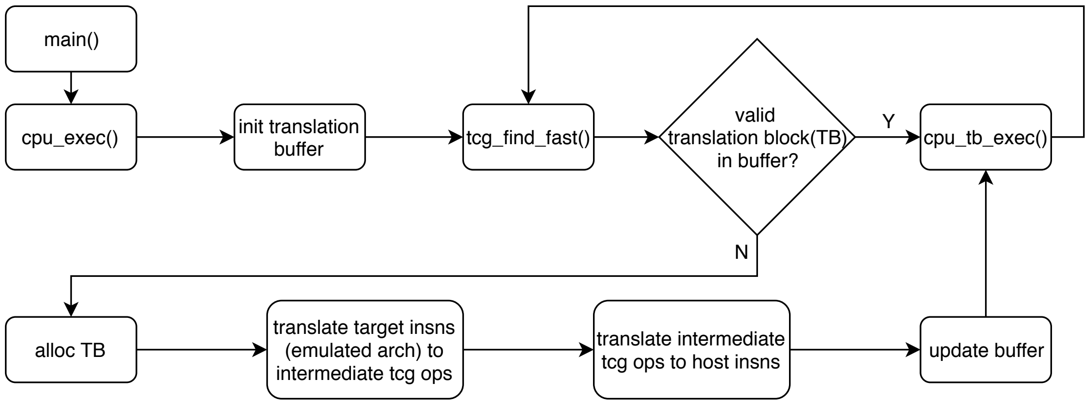

main()
cpu_exec()
init translation buffer
tcg_find_fast()
valid translation block(TB) in buffer?
Y
cpu_tb_exec()
N
alloc TB
translate target insns (emulated arch) to intermediate tcg ops
translate intermediate tcg ops to host insns
update buffer

Figure 3.1 : A brief flowchart of how QEMU emulates guest with TCG

tracing in QEMU is called "trace_guest_mem_before_exec()". The memory trace entries appear in 2 places in figure 3.1, one in the inline software MMU helper functions used by some CPU emulation code, another in the TCG front-end when it translates target instructions to intermediate tcg ops.

There are 2 drawbacks for QEMU's vanilla memory tracing. First, it drops many memory traces for not stalling the guest emulation. The QEMU tracing mechanism uses an asynchronous thread to record and flush the traces. The dedicated thread does not block the main emulation thread so that many memory accesses are ignored. Second, it does not provide the guest physical address for each memory reference. QEMU uses a software TLB to cache the direct address translation from guest virtual address to machine virtual address, skipping the guest physical address on the intermediate level. Therefore, the guest physical address may not be fetched directly.

QEMU is thus modified to eliminate the above 2 drawbacks. Memory traces are collected in a serial manner such that no records are discarded. However, this slows down the emulation speed. Additionally, each time a memory access is intercepted, a page walk is finished right before its execution to fetch the PTE from the guest page table given the virtual address.

3.2.3 QEMU X86 Page Walk Helper

QEMU's own page walk helper function is modified to return a PTE of the page table rather than computing the physical address for the given virtual address. Additionally, for invalid PTEs, "-1" is returned instead denoting a page fault. So the simulator can not only compute the translated physical address but also know the privilege bits of this PTE. A PDE may also be returned when using a superpage. To distinguish the returned entry among PTE, "PDE: 2MB page" and "PDPTE: 1GB page" in Figure 2.2, the page walker function encodes the page size with bit #57 and bit #58, neither of which involves in x86 address translation nor interferes with FreeBSD's OS defined bits.

The page walk function must be called right before each memory access's execution. For inline software MMU load and store functions called during the execution, page walks can be inserted directly inside the inline functions. For memory trace entries in the vanilla TCG front-end, the load and store are translated but have not yet been executed. The page walk code is injected into the TCG helper functions. After being translated, the helper functions are packed together in the translation block. When this TB is executed, QEMU jumps to resolve the page walk in the helper function. So the page walk is correctly deferred to the time right before each memory instruction is executed.

3.2.4 On-the-Fly Compressing

The modified QEMU uses LZ4 compression interface to compress the trace buffer on the fly [39]. Data compression makes it feasible to collect memory accesses (pairs of virtual address and PTE) at an order of 10^{11}. To cover the interesting accessing patterns of each benchmark, they are profiled with performance counters to determine the trace length. Table 3.2 shows the number of memory accesses traced for each benchmark.

3.2.5 Timer

QEMU sends hardware exceptions to the emulated processor driven by host timer. However, the guest is slowed down by a factor of hundreds running in memory tracing mode. So the relatively faster host timer will result in much more exceptions in the guest OS, and the memory accesses will be mostly kernel accesses handling hardware exceptions.

The modified QEMU exploits its record-and-replay feature to mimic the guest timer by timing the number of executed TBs with a time coefficient, which is $2^7 ns$ by default. The coefficient was well tuned to $2ns$ by 429.mcf and 605.mcf benchmarks. However, since the translation block size is not uniform, time emulation by executed TBs cannot provide a perfect solution. On benchmarks with interactions between host and guest via the network, it is harder to preserve the simulation fidelity by tuning the coefficient.

3.3 TLB Simulation

The TLB performance of different policies are simulated to predict their performance on three components of Intel Skylake TLBs in table 2.1. With kernel events instrumented, the altered page tables are emulated while the memory allocation remains unchanged, so that each benchmark only requires one memory trace.

Two assumptions allow fair comparisons of different promotion policies on the same memory trace. First, ignoring changes of the memory accesses from preemptive populations or additional promotions in the new policies should hardly affect their TLB performance. It is simply because of the very large number of memory references traced (10^{11}) compared to the tiny number of reservations (2278 on average) for benchmarks in table 3.2. Second, the breaking time of reservations is almost unaffected by the new promotion policies. In the real world, partially populated but additionally promoted reservations by new promotion policies may be broken at a later time than simulation, because FreeBSD only assumes dirty superpages to be fully utilized. Reservations in the simulation can only be broken at the same or earlier time, so it does not exaggerate the superpage benefits. And such superpages (partially populated when being broken) only account for 8.8% of the total reservations.

Chapter 4 further elaborates how the TLB simulation is validated to be faithful to Intel Skylake CPUs' TLB performance.

3.4 Benchmarks

Table 3.3 summarizes the 17 benchmarks selected for evaluation, covering popular benchmark suites or real-world applications. And table 3.2 shows the number of touched reservations and traced memory accesses. Each benchmark has been profiled to configure a long enough tracing length to cover their interesting accessing patterns.

Benchmarks from existing suites include 429.mcf from SPEC CPU2006 [41]; gcc, 605.mcf, lbm, omnetpp, cam4, deepsjeng, roms and xz from the speed variant of SPEC CPU2017 [42]; canneal from PARSEC [43]; derby from SPEC JVM2008 [44] and tradesoap from DACAPO [53]. The gups benchmark is configured to randomly access 4GB memory [48]. Popular real-world applications are also covered. The graphchi benchmark uses GraphChi library [46] to calculate page ranks of a US patent citation network [47]. The svm.mnist benchmark uses LibSVM library [51] to train a linear SVM (support vector machine) model on MNIST dataset [52]. Two database platforms, postgresql [49] and redis [50] are benchmarked by their own tools.

Benchmark Suite	Benchmark	Description
SPEC-CPU2006 [41]	429.mcf	Route planning
SPEC-CPU2017 [42]	602.gcc_s	GNU C compiler
	605.mcf_s	Route planning
	619.lbm_s	Fluid dynamics (floating point)
	620.omnetpp_s	Network event simulation
	627.cam4_s	Atmosphere modeling (floating point)
	631.deepsjeng_s	AI: alpha-beta tree search (Chess)
	654.roms_s	Ocean modeling (floating point)
	657.xz_s	General data compression
PARSEC [43]	Canneal	Simulated cache-aware annealing
SPEC-JVM2008 [44]	Derby	Java BigDecimal library stressed on telco benchmark [45].
\	Graphchi [46]	Pagerank computing on US Patent citation network [47]
\	Gups [48]	Random-access (serialized)
\	Postgresql [49]	Object-relational database
\	Redis [50]	In-memory database
\	Svm.mnist	Libsvm [51] training MNIST dataset [52]
DACAPO [53]	Tradesoap	JVM running daytrader benchmark [54]

Table 3.3 : Summation of 17 benchmarks selected for evaluation.

Chapter 4

TLB Simulation

The TLB simulator is driven by memory accesses to predict the real-world TLB performance of Intel Skylake processors, which is the microarchitecture released by Intel in 2015. Processors of subsequent microarchitectures, including Kaby Lake and Coffee Lake, are optimized variants that share an identical TLB structure. The three Skylake TLB components in table 2.1 are simulated. This chapter discusses how the details of Skylake TLBs' characteristics are reverse-engineered. Further, empirical results show that the TLB simulation algorithm is faithful enough to predict the real-world performance of Skylake TLBs. Section 4.1 introduces the microbenchmark settings and implications from reverse engineering. Section 4.2 illustrates the close similarity between Skylake TLB simulation and its real-world performance. Section 4.3 summarizes this chapter.

4.1 Reverse-engineering Microbenchmarks

Hardware vendors only release the size or set-associativity of TLBs, which is not enough to precisely model TLB performance. A series of pointer chasing microbenchmarks are designed to infer the characteristics of the replacement policy, indexing functions and inclusiveness for Intel Skylake TLBs. Each pointer chasing micro-benchmark initializes several pointers with configurable memory addresses and constructs them as a cyclic singly-linked list. Hardware performance counters are used to count TLB misses and reveal the TLB behavior by looping over the cyclic singly-linked list for a specific number of times on Linux, where it can allocate and use 2MB pages directly. Three TLB characteristics including replacement policies, indexing functions and inclusiveness are inferred under different configurations of the pointer chasing micro-benchmark. To simplify the following explanations, n denotes the number of TLB entries and k denotes the set-associativity.

4.1.1 Replacement Policies

To infer replacement policies, the length of the cyclic list is configured to be k and $k + 1$ with pointers' memory addresses varying only on the most significant bits so they will be mapped into the same TLB set. Then the TLB miss rate of looping over the cyclic list is proflied. Profiling shows negligible TLB misses in the former case (length=k) but almost 100% references to the cyclic list miss the TLB on the later case (length=$k + 1$). Such behavior holds on all three TLB components in table 2.1. Therefore, they all use an exact LRU replacement policy [10].

4.1.2 Indexing Functions

Indexing functions are detected by exploiting the LRU policy. The number of indexing bits are determined by the entry size n and set-associativity k, which must be an integer $log_2(n/k)$. For a k-way set associative TLB, construct a cyclic list with $k + 1$ pointers varying on the highest $\lceil log_2(k + 1)\rceil - 1$ bits and another detection bit. By shifting the detection bit from the highest bit (Bit $\#\lceil log_2(k + 1)\rceil$) to the lowest, references to the cyclic list must either miss the TLB every time or never, depending on whether the detection bit is involved in set indexing. Because if the k+1 pointers are mapped to the same set of the set-associative cache, exact LRU replacement

	4KB page	2MB page
DTLB	va[15:12]	va[24:22]
STLB	va[18:12] xor va[25:19]	va[27:21]

Table 4.1 : Indexing functions for 4KB and 2MB pages in Skylake TLBs

policy will make 100% references to the cyclic list miss the TLB. Otherwise, the k+1 pointers will be partitioned into at least 2 sets, where (k+1)/2 is strictly less or equal than k, resulting in 0% TLB miss rate no matter which replacement policy is used. The later case denotes a positive detection which implies that the detected bit is involved in the indexing function.

The detection method found 4 indexing bits for DTLB-4K, 3 indexing bits for DTLB-2M/4M and 7 indexing bits for 2MB pages in STLB-4K/2M. The three numbers exactly fit the formula $log_2(n/k)$, so their indexing functions are unique and determined. However, 14 bits are detected in 4KB page indexing of the STLB-4K/2M, while the formula gives 7 bits. Additional micro-benchmarks are configured to infer which 7 pairs of 2 bits are used (using 2 detection bits). The logic gates between each pairs are found to be exclusive-or gates. Table 4.1 summarizes the 4 indexing functions.

However, the STLB still can have three possible designs, depending on how 4KB and 2MB page are probed. Figure 4.1 illustrates a parallel STLB probing design that looks up a virtual address as both 4KB and 2MB pages simultaneously. Another two possible designs probe the page sizes serially; figure 4.2 probes 4KB pages first and figure 4.3 probes 2MB pages first. It is hard to determine which of the three is true for Skylake STLBs, because the hardware performance counters for DTLB-miss/STLB-

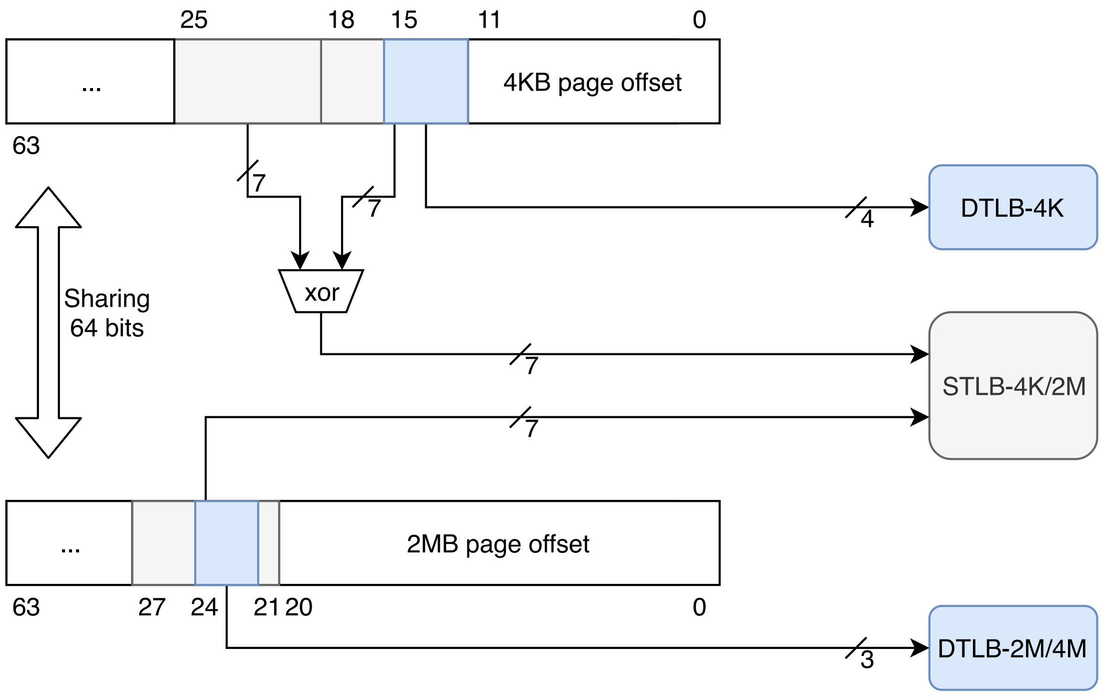


Figure 4.1 : Set Selection Diagram in Intel Skylake TLBs. The 2 page table entries are sharing the same 64 bits. This variant assumes that the STLB look-up is parallel to hit the STLB as both 4KB or 2MB page sizes.

hit do not distinguish STLB hits between 2MB or 4KB pages.

4.1.3 Inclusiveness

The Skylake STLB is inferred to be non-inclusive. An inclusive L2 cache must contain all entries in L1 caches, so evicting an entry from an inclusive STLB will consequently invalidate the same entry in the DTLB, called back invalidation. To detect this, a cyclic list of 16 pointers is constructed. The 16 pointers share identical STLB index, while each group of four of them are indexed into a different DTLB set. Negligible DTLB misses are observed when looping over the cyclic list, but each STLB set has

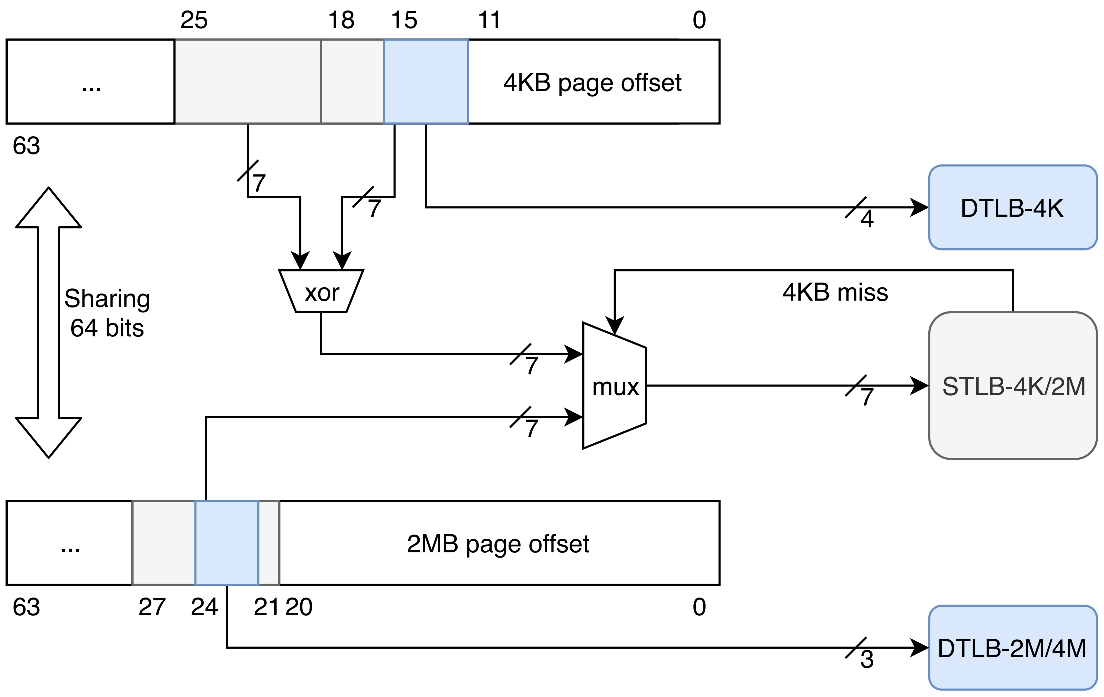


Figure 4.2 : Set Selection Diagram in Intel Skylake TLBs. The 2 page table entries are sharing the same 64 bits. This variant assumes that the STLB look-up is serial to hit the STLB as a 4KB page first.

only 12 entries and inclusiveness is expected to generate all TLB misses. Therefore, back invalidation does not happen and the Skylake STLB is non-inclusive of the DTLB.

The Skylake STLB is also found to be non-exclusive. An exclusive L2 cache would not contain any entries in L1 caches, so an exclusive STLB would only be populated by evictions from the DTLB. However, looping over a cyclic list composed of 13 pointers indexed into the same DTLB and STLB set makes all references to the cyclic list missed. An exclusive STLB is expected to show negligible STLB misses even if the length of the cyclic list is increased to 16 (12+4). Thus the Skylake STLB is also not

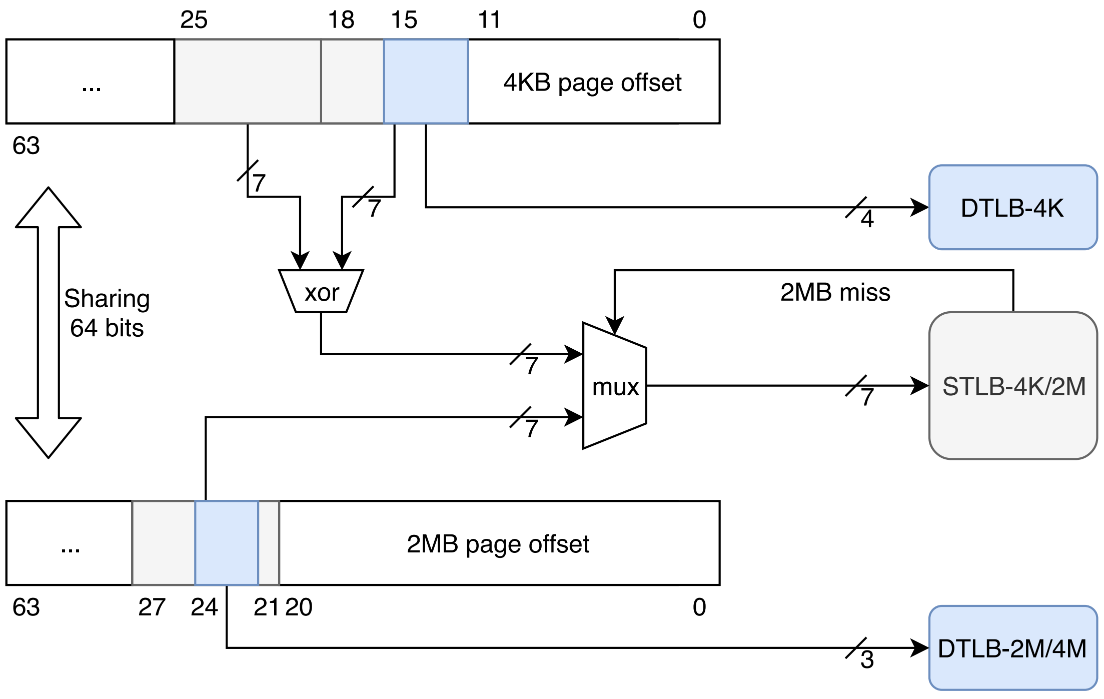


Figure 4.3 : Set Selection Diagram in Intel Skylake TLBs. The 2 page table entries are sharing the same 64 bits. This variant assumes that the STLB look-up is serial to hit the STLB as a 2MB page first.

exclusive of the DTLB.

The behavior of insertion and eviction in Skylake TLBs is thus determined. Evicted entries from the DTLB do not get inserted back to the STLB (**non-exclusive**), so TLB entries found by the page walker must be inserted into both DTLB and STLB. At the same time, evicting entries from the STLB brings no side effects, because they do not back-invalidate entries in the DTLB (**non-inclusive**).

4.1.4 Skylake TLB Simulation Algorithm

The following assumptions are additionally made to build the simulation algorithm for Skylake TLBs:

1. The CPU always tries to hit with a 2MB mapping first.
2. Different page sizes are always queried in a serial fashion.
3. If hit with a 2MB mapping, the look-up is terminated so that the 4KB LRU queue is not affected.
4. All memory references are queried in order.
5. The TLB replacement/update must finish before each memory access.
6. Page walker always inserts the entry into both DTLB and STLB.

However, some assumptions are not necessarily true for Skylake TLBs. They can either hit 4KB pages first or both page sizes in parallel, so the LRU queue for replacement could be changed. But this only provides minor side effects for TLB simulation, because it is very rare for the TLB to have both 4KB pages (not valid again but correct to use) and valid 2MB pages together. Real-world processors execute memory loads and stores out of order to overlap some latency, so the DTLB replacements or updates may not get finished before the next TLB query. This makes the performance counter counts duplicate DTLB misses (not for STLB misses) when querying the identical page consequently. Assuming that memory accesses executions are in-order and the TLB replacement/update is synchronized do not affect simulating the STLB performance, which effectively dominates the TLB performance. The last assumption has been validated to be true.

Algorithms 1, 2, 3 and 4 describe the simulation algorithm for Intel Skylake TLBs.

Algorithm 1: HIT-TLB

```
Data: A TLB $TLB$, Current PCID $Pcid$, A Virtual Tag $Tag$, An Index
      $Index$, A Page Size Value $Size$
Result: A Boolean Value Denoting Hit or Not
if $TLB - Set[Index]$ contains valid entry $(Pcid, Tag, Size)$ then
    Increase ages of all entries in $TLB - Set[Index]$ by 1;
    Set age of $(Pcid, Tag, Size)$ as 0;
    Return $True$;
Return $False$;
```

Algorithm 2: Hit-DTLB

```
Data: Current PCID $Pcid$, A Virtual Address $VA$ to Look up
Result: A Boolean Value Denoting Hit or Not
$VPN_{2MB} \longleftarrow VA/2^{21}$;
$VPN_{4KB} \longleftarrow VA/2^{12}$;
$Index_{DTLB-2MB} \longleftarrow VPN_{2MB}/2 \mod 2^3$;
if $HIT - TLB(DTLB_{2MB}, Pcid, VPN_{2MB}, Index_{DTLB-2MB}, 2MB)$ then
    Return $True$;
$Index_{DTLB-4KB} \longleftarrow VPN_{4KB} \mod 2^4$;
if $HIT - TLB(DTLB_{4KB}, Pcid, VPN_{4KB}, Index_{DTLB-4KB}, 4KB)$ then
    Return $True$;
/* DTLB Miss                                                    */
Return $False$;
```

Algorithm 3: Hit-STLB

```
**Data:** Current PCID $Pcid$, A Virtual Address $VA$ to Look up
**Result:** A Boolean Value Denoting Hit or Not
$VPN_{2MB} \longleftarrow VA/2^{21}$;
$VPN_{4KB} \longleftarrow VA/2^{12}$;
$Index_{STLB-2MB} \longleftarrow VPN_{2MB} \mod 2^7$;
**if** $HIT-TLB(STLB, Pcid, VPN_{2MB}, Index_{STLB-2MB}, 2MB)$ **then**
    Replace the hit entry with the oldest in
      $DTLB_{2MB} - Set[Index_{DTLB-2MB}]$;
    Return $True$;
$R \longleftarrow VPN_{4KB} \mod 2^7$; $L \longleftarrow VPN_{4KB}/2^7 \mod 2^7$;
$Index_{STLB-4KB} \longleftarrow (LxorR)$;
**if** $HIT-TLB(STLB, Pcid, VPN_{4KB}, Index_{STLB-4KB}, 4KB)$ **then**
    Replace the hit entry with the oldest in
      $DTLB_{4KB} - Set[Index_{DTLB-4KB}]$;
    Return $True$;
/* STLB Miss                                                  */
Return $False$;
```

Algorithm 4: Main

```
Data: Current PCID $Pcid$, A Virtual Address $VA$ to Look up, A Page Size
      Value $Size$ for $VA$ Determined by Page Table
Result: A Boolean Value Denoting Hit or Not
Set all TLB entries invalid with infinite age;
$VPN_{2MB} \longleftarrow VA/2^{21}$;
$VPN_{4KB} \longleftarrow VA/2^{12}$;
if Hit-DTLB returns $False$ then
    /* DTLB misses */
    if Hit-STLB returns $False$ then
        /* STLB misses, Page walk: Found page size to be $Size$ */
        $Entry \longleftarrow (Pcid, VPN_{Size}, Size)$ Set age of $Entry$ as 0;
        Replace $Entry$ with the oldest in $DTLB_{Size} - Set[Index_{DTLB-Size}]$;
        Increase ages of all other entries in $DTLB_{Size} - Set[Index_{DTLB-Size}]$;
        Replace $Entry$ with the oldest in $STLB - Set[Index_{STLB-Size}]$;
        Increase ages of all other entries in $STLB - Set[Index_{STLB-Size}]$;
        Return $False$;
Return $True$;
```

4.2 Empirical Validation

Empirical comparison shows that the above TLB algorithms are faithful to simulate performance of Intel Skylake TLBs. To compare, the real-world TLB performance is profiled by performance counters for the following three events [9]:

1. Retired memory accesses
2. DTLB misses hit in STLB
3. Retired STLB misses

The above three events are accumulated on each 100 milliseconds. Therefore, both real-time DTLB and STLB miss rate curves can be plotted.

4.2.1 Empirical Results

Figures 4.4, 4.5, 4.6 and 4.7 compare the TLB performance between simulation and real world on two deterministic benchmarks, 429.mcf from SPEC-CPU2006 [41] and 605.mcf from SPEC-CPU2017 [42]. The later benchmark differs by using a larger data set and an updated computing algorithm than 409.mcf. Both benchmarks are compared on an Intel Xeon E3-1245 V6 processor, an optimized variant of Skylake but sharing the same TLB design. Specifically, to highlight that both 4KB and 2MB pages are simulated faithfully in the shared STLB, the superpage promotion is switched between enabled and disabled. Unless the comparison shows strong performance prediction in each case, the TLB simulation is faithful.

Three things are learned from the performance comparisons in the four figures (4.4,4.5, 4.6 and 4.7):

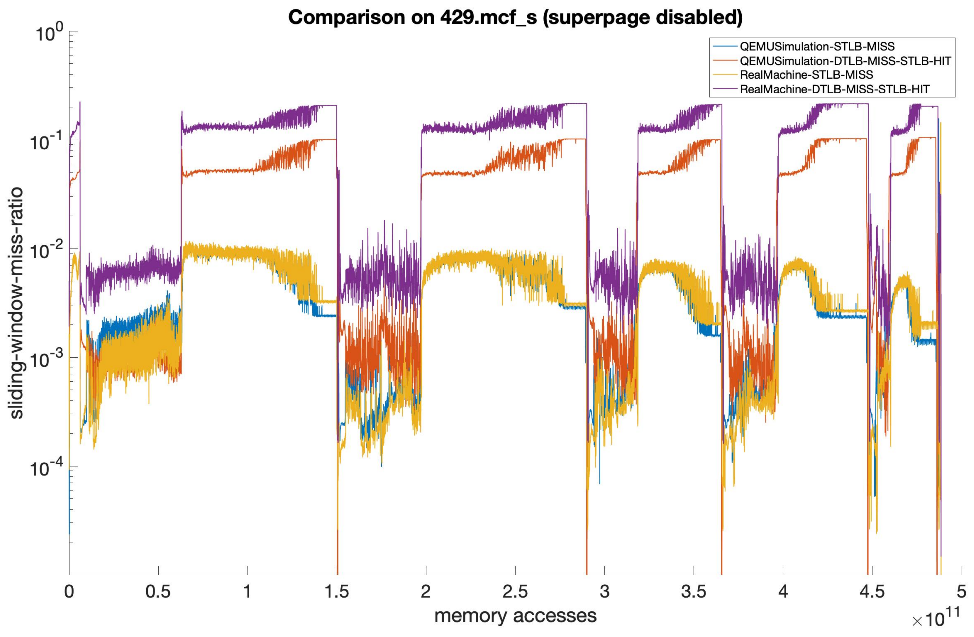


Figure 4.4 : TLB miss ratio curves between simulation and real-machine. The benchmark is a full run of 429.mcf in SPEC-CPU2006 [41]. The left shift value for QEMU record-and-replay feature is tuned to be 1. The y-axis represents a log-scale sliding-window miss rate, and the x-axis represents cumulative memory accesses. The real-machine is profiled with **superpage promotion disabled** in FreeBSD. The simulation uses traces collected with superpage promotion enabled but assumes all user-space page sizes as 4KB.

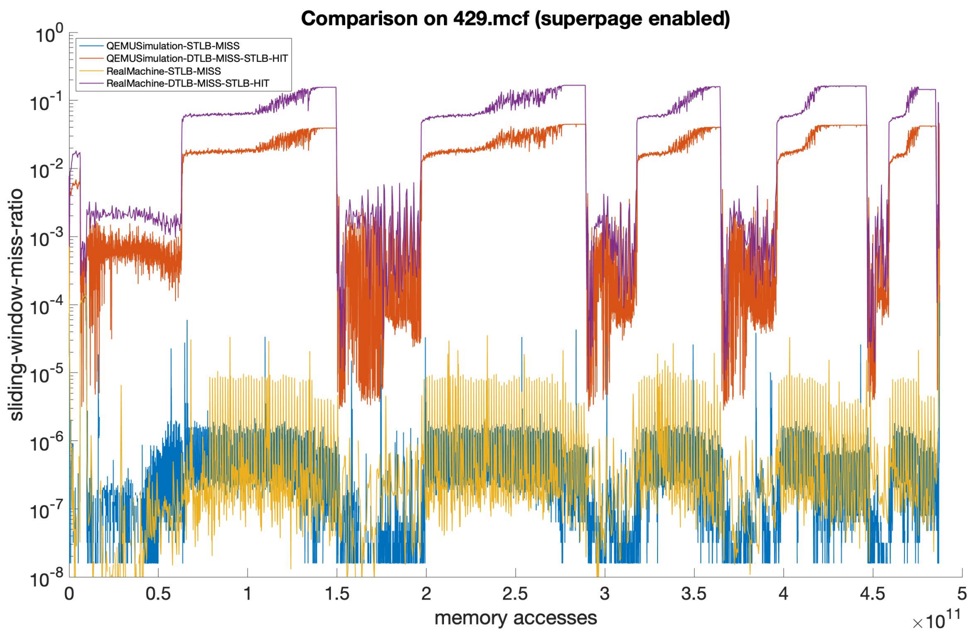


Figure 4.5 : TLB miss ratio curves between simulation and real-machine. The benchmark is a full run of 429.mcf in SPEC-CPU2006 [41]. The left shift value for QEMU record-and-replay feature is tuned to be 1. The y-axis represents a log-scale sliding-window miss rate, and the x-axis represents cumulative memory accesses. The real-machine is profiled with **superpage promotion enabled** in FreeBSD. The simulation uses traces collected with superpage promotion enabled and do not alter the page tables.

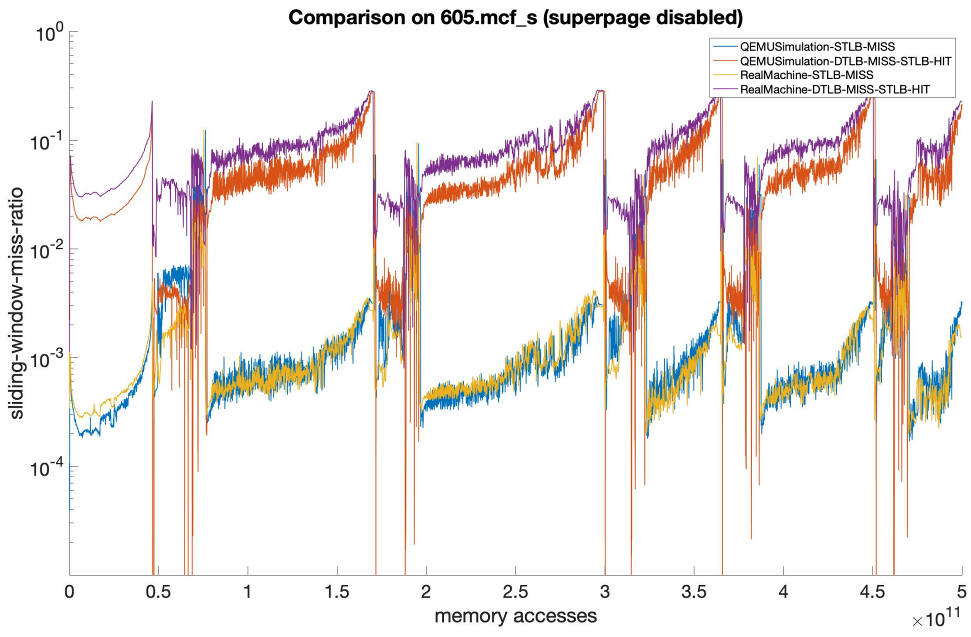


Figure 4.6 : TLB miss ratio curves between simulation and real-machine. The benchmark is a partial run of 605.mcf from SPEC-CPU2017 [42]. The left shift value for QEMU record-and-replay feature is tuned to be 1. The y-axis represents a log-scale sliding-window miss rate, and the x-axis represents cumulative memory accesses. The real-machine is profiled with **superpage promotion disabled** in FreeBSD. The simulation uses traces collected with superpage promotion enabled but assumes all user-space page sizes as 4KB.

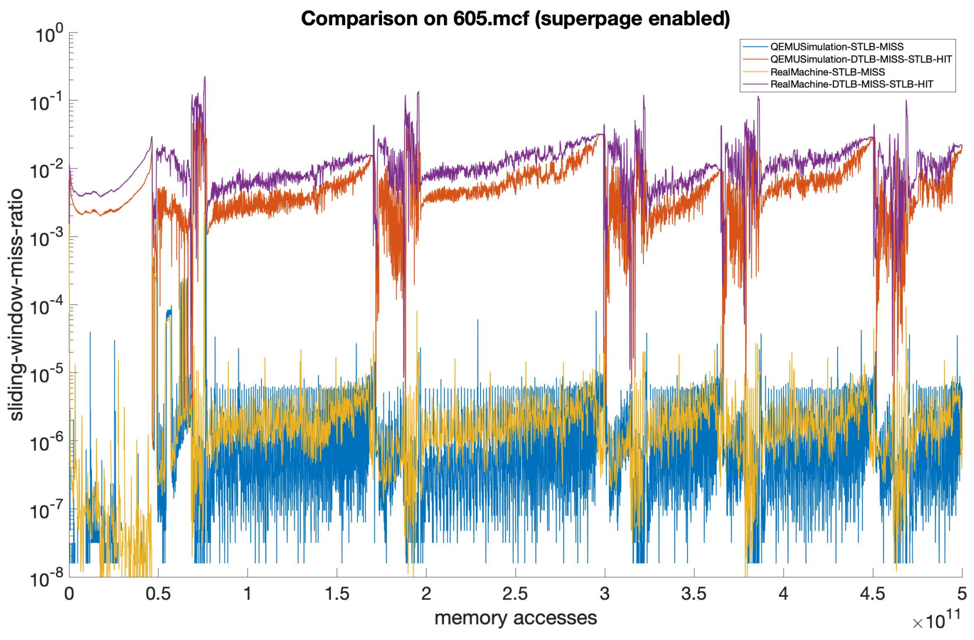


Figure 4.7 : TLB miss ratio curves between simulation and real-machine. The benchmark is a partial run of 605.mcf from SPEC-CPU2017 [42]. The left shift value for QEMU record-and-replay feature is tuned to be 1. The y-axis represents a log-scale sliding-window miss rate, and the x-axis represents cumulative memory accesses. The real-machine is profiled with **superpage promotion enabled** in FreeBSD. The simulation uses traces collected with superpage promotion enabled and do not alter the page tables.

- STLB performance is closely tracked by the simulation when superpages are disabled
- Simulation mismatches the STLB performance when superpages are enabled
- DTLB performance is tracked but constantly underestimated

The next section further elaborates why the STLB data misses are actually well predicted and why the DTLB is underestimated but can help evaluate the promotion policies.

4.2.2 Noise from STLB Code Misses

This section elaborates how the misses from ignored instructions are interfering with the STLB simulation when superpages are enabled exemplified on 605.mcf benchmark. Remember that the Skylake STLB is shared between instructions and data. When superpages are enabled on 605.mcf, the STLB data misses drop significantly and no longer dominate the STLB performance.

Figure 4.8 shows the code misses in STLB causing page walks profiled by hardware performance counter "ITLB_MISSES.WALK_COMPLETED". Despite 2 peaks exceeding 10^{-3}, the STLB miss rate of code misses, the number of code misses dividing the number of memory accesses, is mostly negligible.

With code misses profiled, it is able to decouple the code misses and data misses for STLB. Figure 4.9 subtracts the code misses from the total number of STLB misses to approximate the STLB behavior when only memory accesses query the STLB. To elaborate the comparison between the TLB simulation and the subtracted real-world profiling, both zoom-in and smoothing are applied on this figure (4.9).

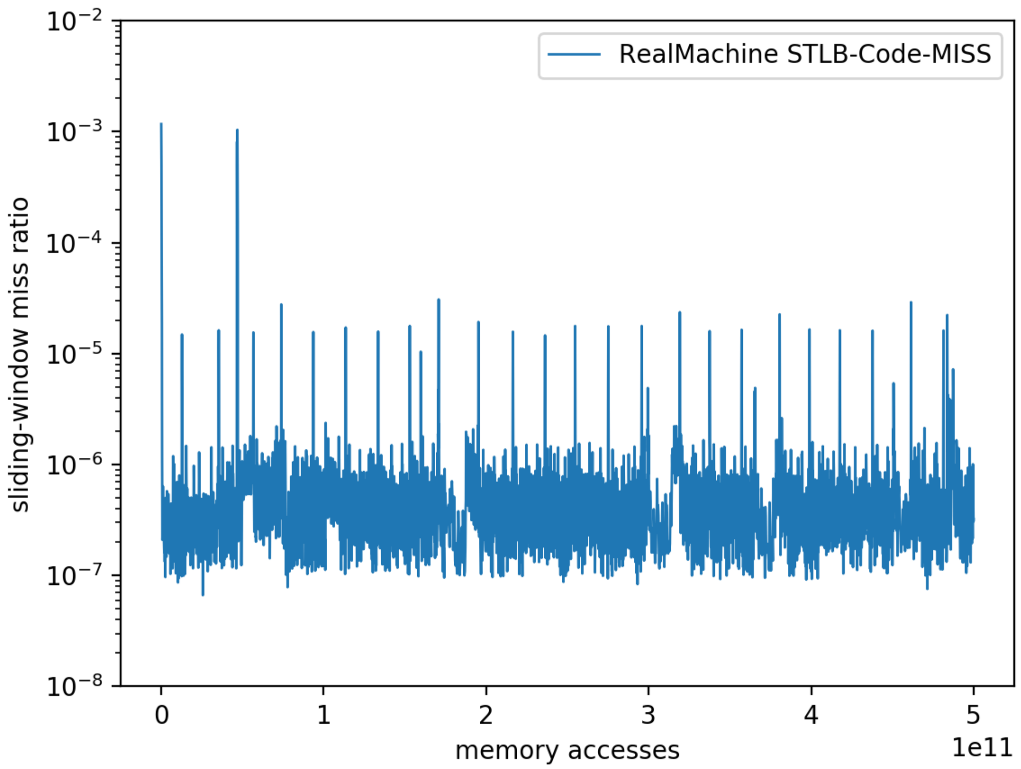


Figure 4.8 : STLB code misses dividing number of retired memory accesses per 100 milliseconds.

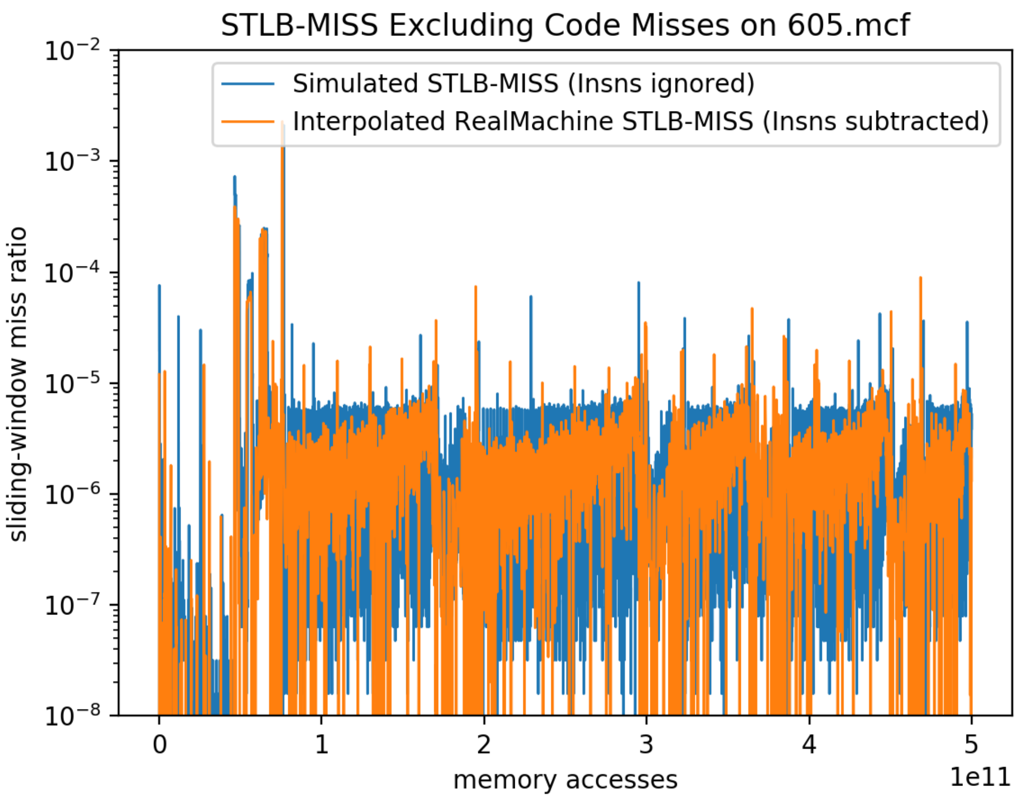


Figure 4.9 : Comparison of STLB miss ratio curves **(excluding code miss effects)** between simulation and real-machine. The benchmark is a partial run of 605.mcf from SPEC-CPU2017 [42]. The y-axis represents a log-scale sliding-window miss rate, and the x-axis represents cumulative memory accesses. The real-machine curve is profiled with **superpage promotion enabled** in FreeBSD. It approximates STLB data misses by subtracting STLB code misses from data misses, assuming that a code miss will correspond to another data miss to fetch back the evicted data entry. Both data STLB misses and code STLB misses are linearly interpolated in order to perform the subtraction. The simulation uses traces collected with superpage promotion enabled and do not alter any page size.

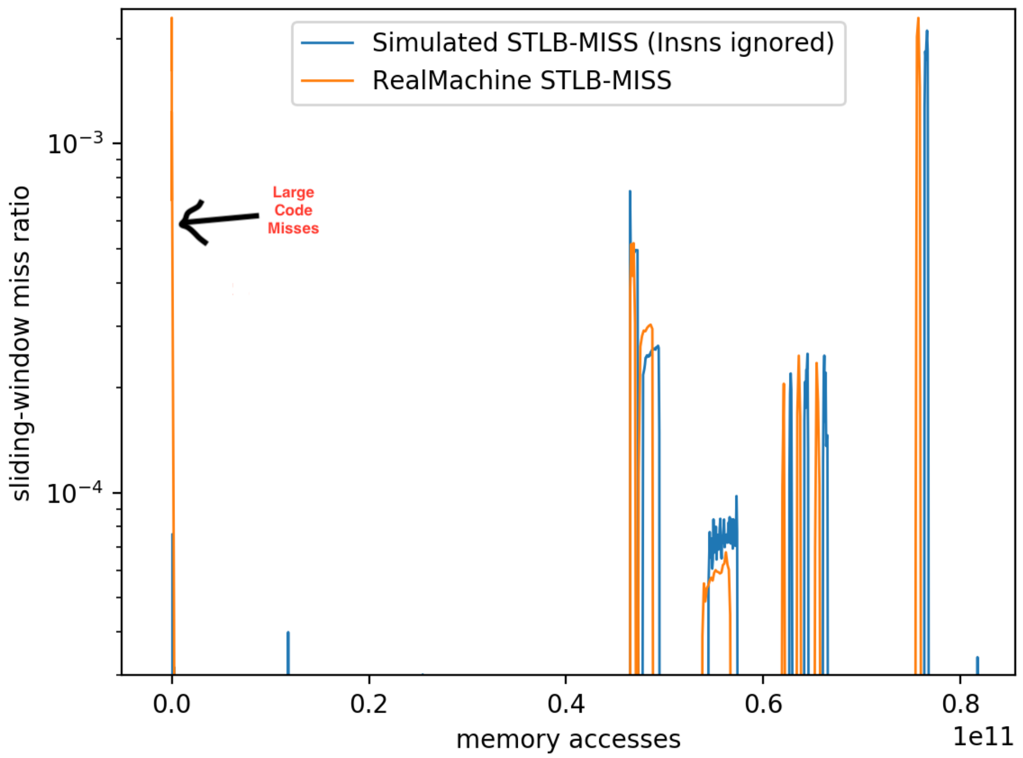


Figure 4.10 : Zoom in on the peaks above 10^{-4} in Figure 4.7. STLB code misses are not subtracted from the real-machine curve.

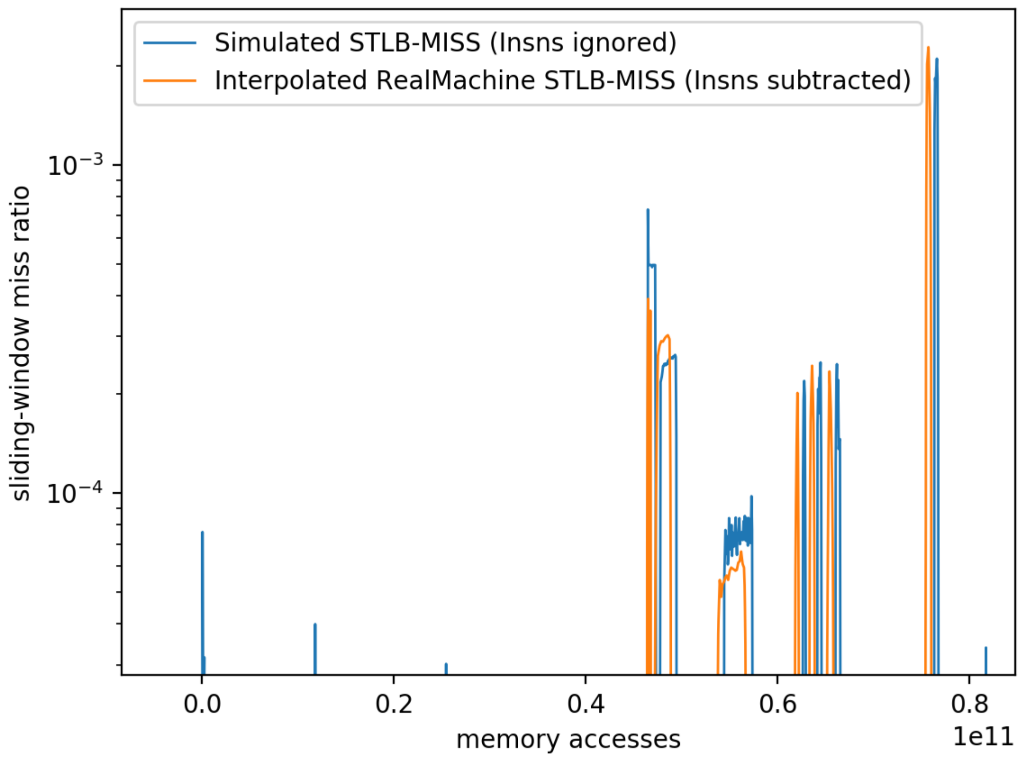


Figure 4.11 : Zoom in on the peaks above 10^{-4} for STLB miss curves in Figure 4.9. STLB code misses are subtracted.

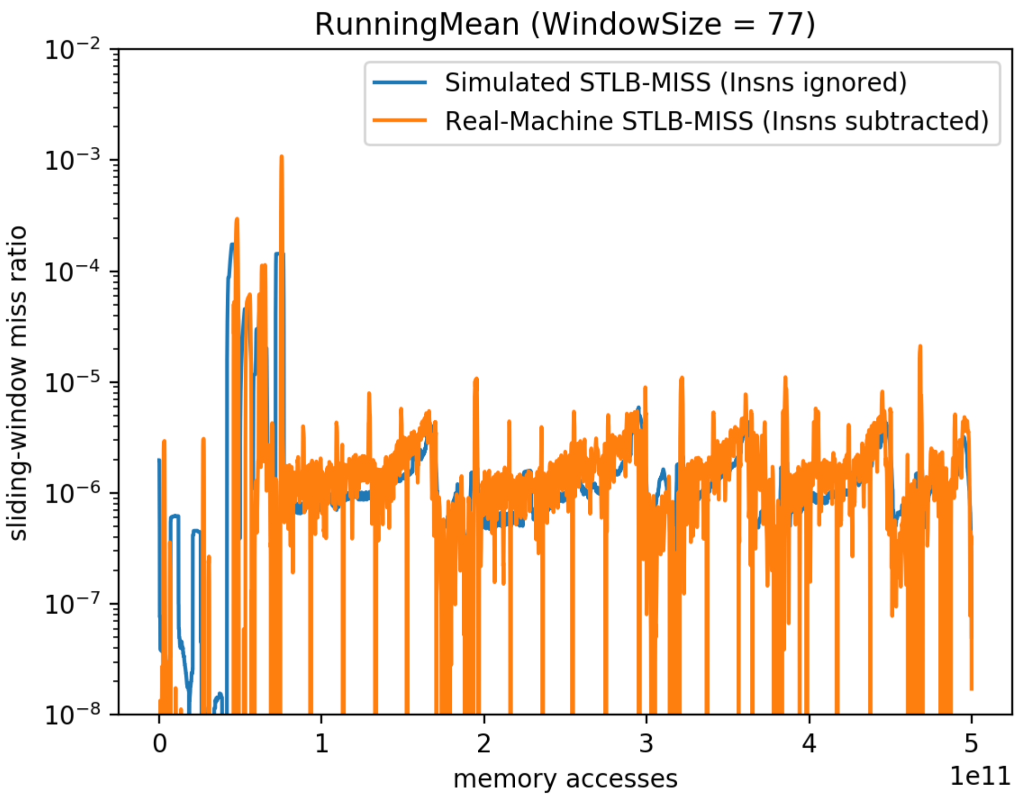


Figure 4.12 : Smoothed by applying a mean convolution filter, window size = 77

Figures 4.10 and 4.11 zoom in the area of figure 4.7 where the STLB miss rate is high. The strong correlation between STLB (real) and STLB (simulation) curves shows how STLB performance is closely tracked when data misses dominate. Additionally, figure 4.12 subtracts the code misses from STLB (real) and compares it again with STLB (simulation) by smoothing both curves into a longer time scale. Again, strong correlation between simulation and the real world is shown on areas where data misses do not dominate the STLB performance. Combine with the closely tracked STLB performance when superpage is disabled (figure 4.6 and figure 4.4) and the fact that applications' data dominate the memory footprints, the simulation is able to closely predict the Skylake STLB performance.

4.2.3 Underestimation of DTLB Misses

Intel tries to overlap the DTLB miss penalty with out-of-order execution [9]. So the DTLB miss performance counter is found to count duplicate misses when a memory access misses in the DTLB but hits the STLB and a subsequent accesses is to the same page before the STLB hit is resolved. In other words, DTLB updates do not stall succeeding memory accesses. Therefore, all figures 4.4, 4.5 4.6 and 4.7 show that the TLB simulator can track the trend of DTLB miss rates but underestimates their magnitudes. While the DTLB miss performance counter can also be affected by speculative executions and prefetching, ignoring these effect does not substantially hurt the simulation.

The aforementioned DTLB underestimation is independent of the page size, so that DTLB performance of different promotion policies should be consistently underestimated on the same benchmark. Together with the closely tracked STLB performance, the simulator can fairly predict and compare the TLB performance among different superpage promotion policies.

4.3 Summary

Empirical comparisons show a strong correlation of the TLB performance between the TLB simulation and real world profiling. Though not perfectly simulated, the STLB, which dominates the TLB performance, is tracked closely to indicate what TLB performance will be given superpage promotion policies. Furthermore, their DTLB performance can also be fairly compared. The faithfulness of the TLB simulation provides the foundation to fairly evaluate different superpage promotion policies.

Chapter 5

Design Space of Superpage Promotion Policies

This chapter explores 16 new promotion policies in addition to FreeBSD's policies. As mentioned before, they share FreeBSD's reservation system, but have different criteria for promoting superpages. As a point of comparison, two existing policies, *FreeBSD* and *4K-user* are also shown. *FreeBSD* denotes the vanilla promotion policy in FreeBSD and *4K-user* denotes a policy that disables all superpage promotions in the user space.

5.1 Capture 2 Types of Promotion Opportunities

FreeBSD's promotion policy misses two types of promotion opportunities. First, it takes a long time for a reservation to satisfy the strict promotion conditions. During that time, the constituent 4KB pages can consume up to one third of the TLB entries in a Skylake STLB. Second, some reservations come close to satisfying the promotion conditions, but never do. For example, a reservation in which only 511 pages become populated would never be promoted.

The new promotion policies form a design space of more aggressive promotion policies that can promote more reservations and can promote reservations earlier. Specifically, 2 policies are first considered as the extreme cases. *Greedy* promotes every reservation to a superpage upon its first touch, catching all promotion opportunities. *Foresight* also promotes upon first touch, but only for those reservations

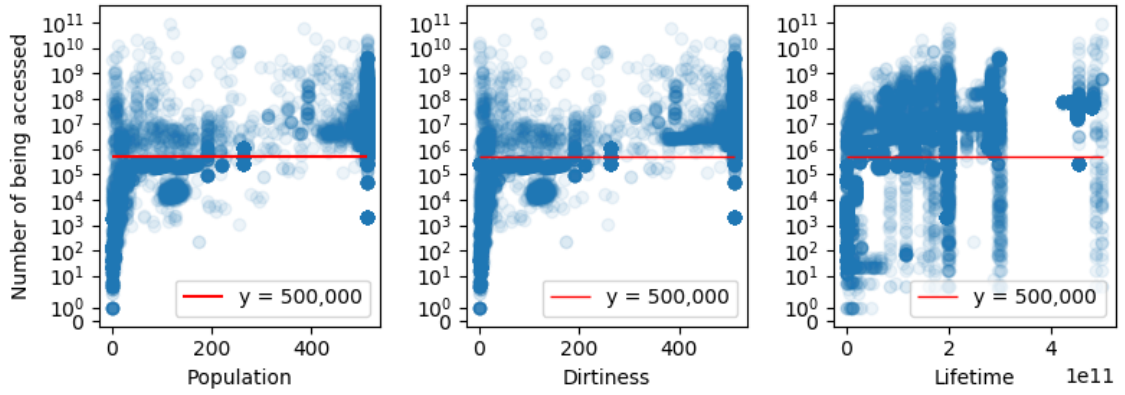


Figure 5.1 : Distribution of the number of accesses to all reservations touched by benchmarks in table 3.2 vs. their population, dirtiness and lifetime. Each circle denotes a reservation, the darker the more overlapped. Reservations accessed over 500,000 times are classified as frequently accessed ones.

that FreeBSD would ultimately promote. The *foresight* policy captures all of the possible address translation benefits for pages that FreeBSD would promote without incurring any additional overhead. However, *foresight* would require an oracle, so is not an implementable policy (whereas all other policies could be implemented).

5.2 Promote Frequently-used Reservations

The remainder of the policies focus on promoting frequently-accessed reservations. However, monitoring all accesses to each reservation is not possible. Therefore, three types of available thresholds are used as proxies, including population, dirtiness (number of dirty 4KB pages) and lifetime (number of total memory accesses to any page since creation). Figure 5.1 shows the distribution of the number of memory accesses to all reservations ever touched by the benchmarks in table 3.2 against the three

Policy	**Description**
4K-user	disable superpage promotions
FreeBSD	vanilla policy in FreeBSD
foresight	partially-ideal
greedy	most aggressive
pop-x	x = 64, 128, 256, 461, 509
dirty-x	x = 64, 128, 256, 461, 509
life-x	x = 10^6, 10^7, 10^8, 10^9

Table 5.1 : Explored design space of superpage promotion policies

types of thresholds. By classifying the reservations accessed over $5 \cdot 10^4$ as frequently accessed ones, a binary classifier which identifies reservations with no less than 64 populated pages as frequently accessed can achieve 82.8% precision. Similarly, the dirty distribution is similar to the population distribution. However, the lifetime distribution is not as amenable to such a simple classifier. Chapter 6 further elaborates that desirable lifetime thresholds vary from benchmark to benchmark.

Table 5.1 summarizes the policies explored including FreeBSD's original policies (*4K-user* and *FreeBSD*) and the 16 new policies. The population and dirtiness policies use 5 thresholds including 64, 128, 256, 461 and 509. Threshold 461 corresponds to the 90% utilization as suggested in Ingens [30]. Threshold 509 denotes a non-overhead promotion, because superpage promotion releases 3 pages that are used to store the meta data of over 500 4KB pages. Note that *greedy* is equivalent to *pop-1*. The lifetime policies use four thresholds from 10^6 to 10^9.

Chapter 6

Evaluation

This chapter discusses the empirial evaluation results of the 18 policies in table 5.1 on 17 benchmarks in table 3.2. Section 6.1 introduces different metrics to measure their performance. Section 6.2 shows the empirical results under these measurements.

6.1 Metrics

Table 6.1 summarizes the used metrics, including DTLB misses, STLB misses and average page walk latency for evaluating TLB performance. The page walk latency uses 35 cycles for 4KB-page STLB misses and 21 cycles for 2MB-page STLB misses, determined by microbenchmarks.

The page fault savings and I/O overheads are all quantified by the number of corresponding 4KB pages. If a reservation is promoted earlier than it would be in policy *FreeBSD*, then the pre-population from the early promotion is likely to save some future page faults. Additionally, such promoted reservations may not get fully utilized. So, extra effort is wasted for pre-populating them, called extra zeroed pages. The overhead is further separate for reservations backing anonymous memory and disk files, distinguishing page zeroing and disk-in overhead. Similarly, as extra write-protection no longer exists in the new policies, modified superpages will falsely contain clean 4KB pages. These pages are quantified as false dirty pages to denote disk-out overhead. For anonymous memory, it only shows the potential burden to swap the

Metric	Unit
DTLB miss	Number of misses
STLB miss	Number of misses
Page walk latency	Number of CPU stall cycles
Page fault savings	Number of prefaulted 4KB pages
Extra zeroed pages	Number of extra zeroed 4KB pages
False dirty pages	Number of clean 4KB pages in modified superpages

Table 6.1 : Metrics for evaluating different superpage management policies

superpages.

6.2 TLB Performance

The TLB performance including DTLB misses, STLB misses and estimated page walk latency is compared by normalizing to the performance of policy *FreeBSD* in figures 6.1, 6.2 and 6.3. Specifically, policy *4K-user* is excluded from the TLB performance comparison because it is too bad to be meaningful. Note that there are no page fault savings nor I/O overhead for policies *4K-user* and *FreeBSD* either.

6.2.1 DTLB Misses

Figure 6.1 compares the DTLB misses. While *greedy* usually eliminates the most DTLB misses, *foresight* is not as aggressive as it is expected. So, there is a considerable room to benefit the DTLB by promoting partially populated reservations. However, DTLB thrashing happens for *greedy* on benchmarks like canneal. It is more likely to result from the smaller size of DTLB-2M, which is half the size of the

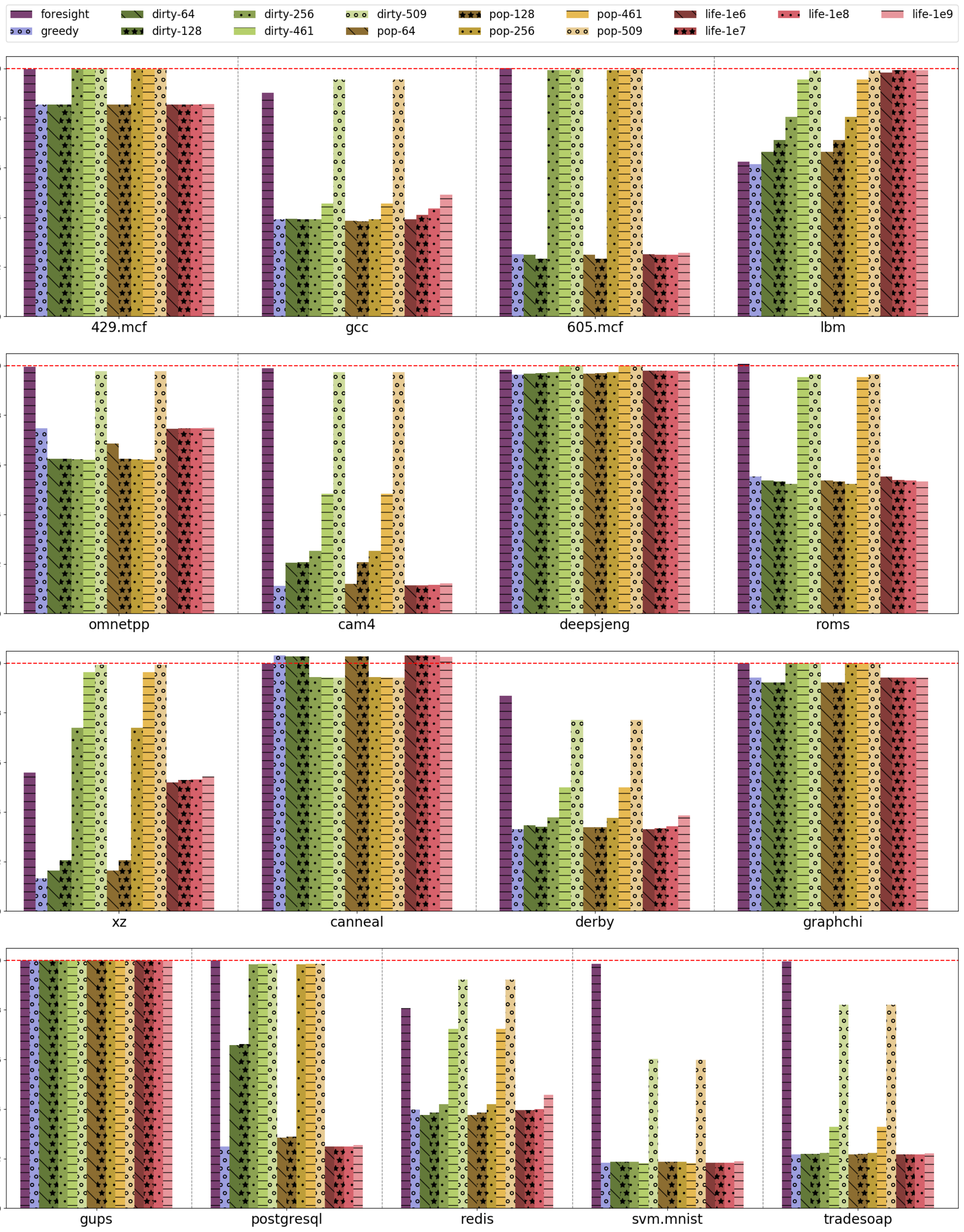


Figure 6.1 : DTLB misses dividing FreeBSD policy among policies shown in table 5.1, excluding policy *4K-user*.

DTLB-4K (64 entries). When *greedy* promotes all reservations that occupy the most memory, the DTLB gets overwhelmed by queries for 2MB pages.

Policy *dirty-64* and *pop-64* can get competitive DTLB performance to *greedy*. When switching to another threshold 128, the dirty and population predictors are as good. They can be better than *greedy* on benchmark omnetpp, possibly because of less DTLB thrashing. However, on postgresql benchmark, the dirty predictor brings more DTLB misses than the population predictor. It is because postgresql would fault anonymous pages in advance to avoid future page faults, who might not get a chance to be modified. So that reservations are promoted earlier under the same threshold of population.

Performance of the *life-x* policies is unstable. They can be better than *greedy* on benchmarks like postgresql but much worse on benchmarks like xz. The unstable DTLB performance shows the difficulty of choosing a good threshold for lifetime based promotion policies.

6.2.2 STLB performance

Figure 6.2 compares the STLB performance and figure 6.3 measures the CPU stall cycles for their page walks. Unsurprisingly, both *foresight* and *greedy* can save more percentage of STLB misses than DTLB misses because mainly using 2MB pages do not effectively reduce the available TLB entries, except for benchmarks deepsjeng and gups, whose memory access patterns do not let the TLB benefit much from their locality. Again, policy *greedy* advances *foresight* a lot, showing the necessity of promoting partially populated reservations. Thresholds like 64 or 128 all work well on policy *pop-x* and *dirty-x*, bringing competitive performance with the most aggressive one, *greedy*. Policy *life*-10^6 shows close TLB performance with *greedy* except for

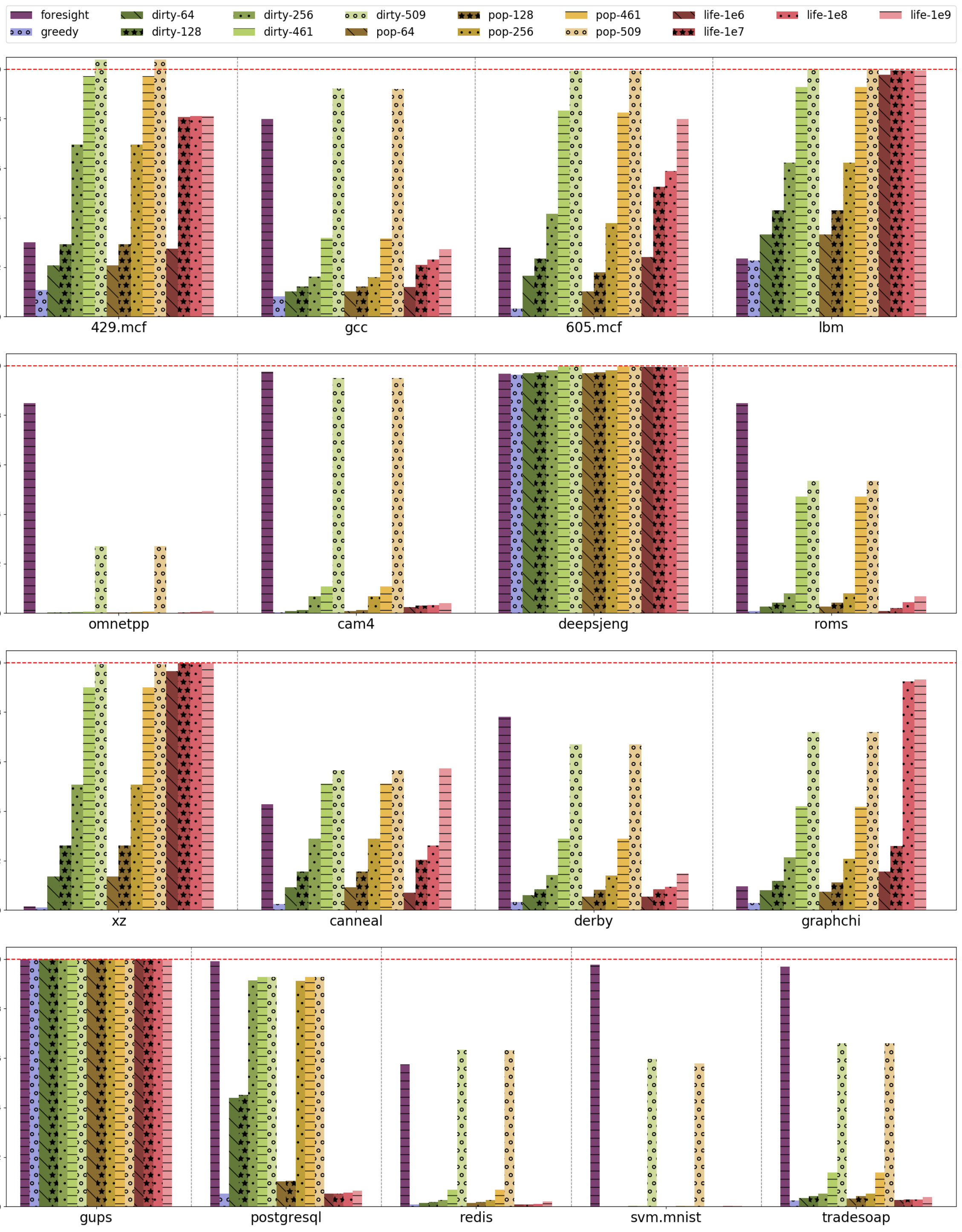


Figure 6.2 : STLB misses dividing FreeBSD policy among policies shown in table 5.1, excluding policy *4K-user*.

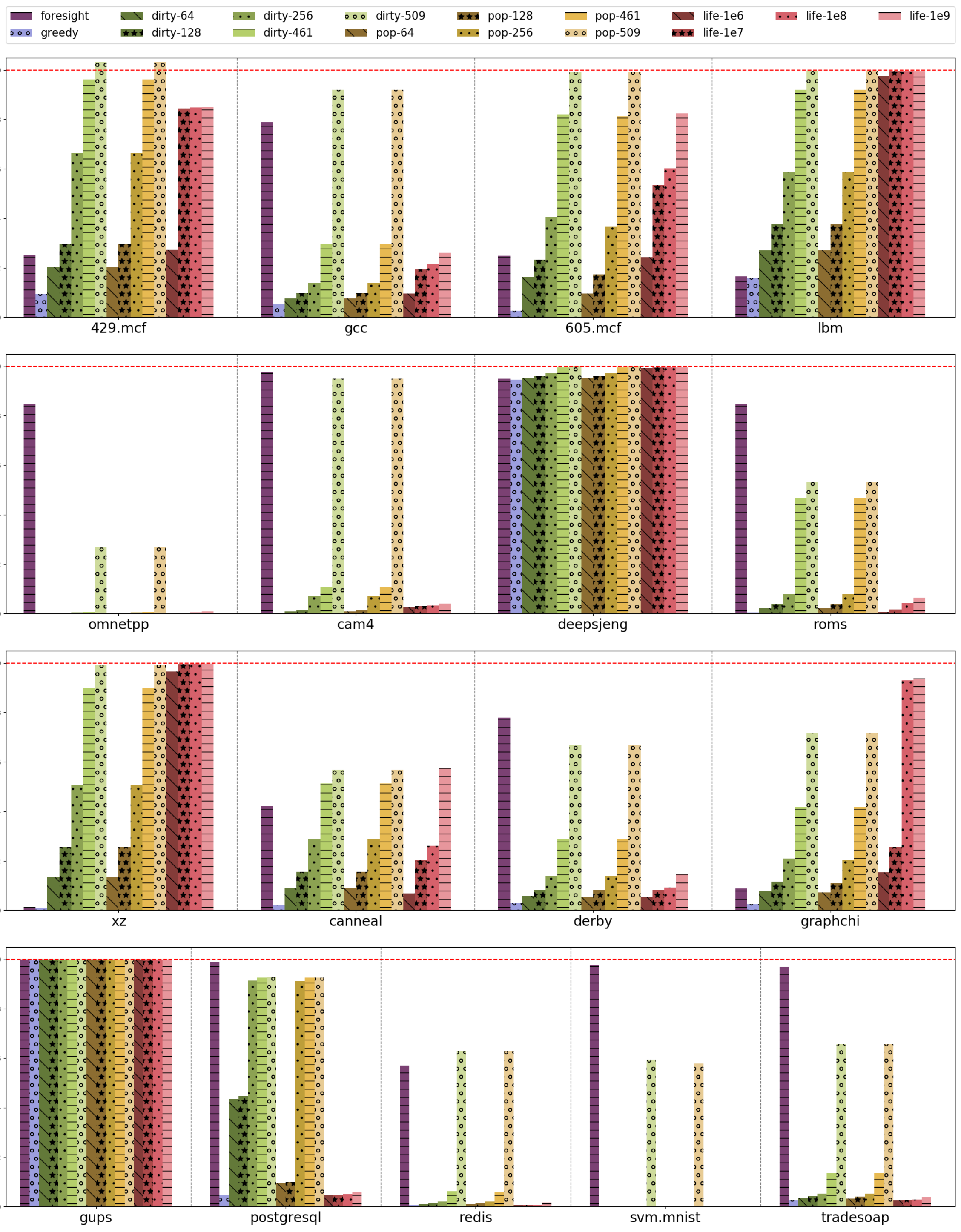


Figure 6.3 : Estimated page walk latency dividing FreeBSD policy among policies shown in table 5.1, excluding policy *4K-user*.

benchmarks lbm and xz, suggesting that a good lifetime threshold is benchmark-specific.

6.2.3 Prefetching

Figure 6.4 and figure 6.5 evaluate the level of prefetching the 4KB pages inside existing reservations by measuring how many page faults can be avoided as a positive side effect of early promotion. Unsurprisingly, the aggressive policies can always save more page faults, denoting a more aggressive level of prefetching memory.

6.2.4 Overhead from Anonymous Reservations

Figure 6.6 shows the page zeroing overhead of promoting superpages for anonymous memory. Though policies *greedy* and *life*-10^6 usually zero more than 50 4KB pages for each reservation, policies *pop-64* and *dirty-64* shows negligible page zeroing overhead on most benchmarks. Even on benchmarks that the two policies have considerable overheads, they do better than *greedy*. On cam4 and redis, *greedy* zeros extra 175 and 246 pages per reservation but *dirty-64* or *pop-64* zeros extra 95 and 189 pages per reservation.

Figure 6.7 shows the false dirty pages created in reservations backing anonymous memory. Since anonymous memory is usually modified, as it is a waste to allocate anonymous memory without future modification, the figure is very similar with figure 6.6. Though users usually try to avoid memory swapping, real OS implementation must consider this as a potential burden for swapping aggressively promoted superpages. Both *pop-64* and dirty-64 still give good results.

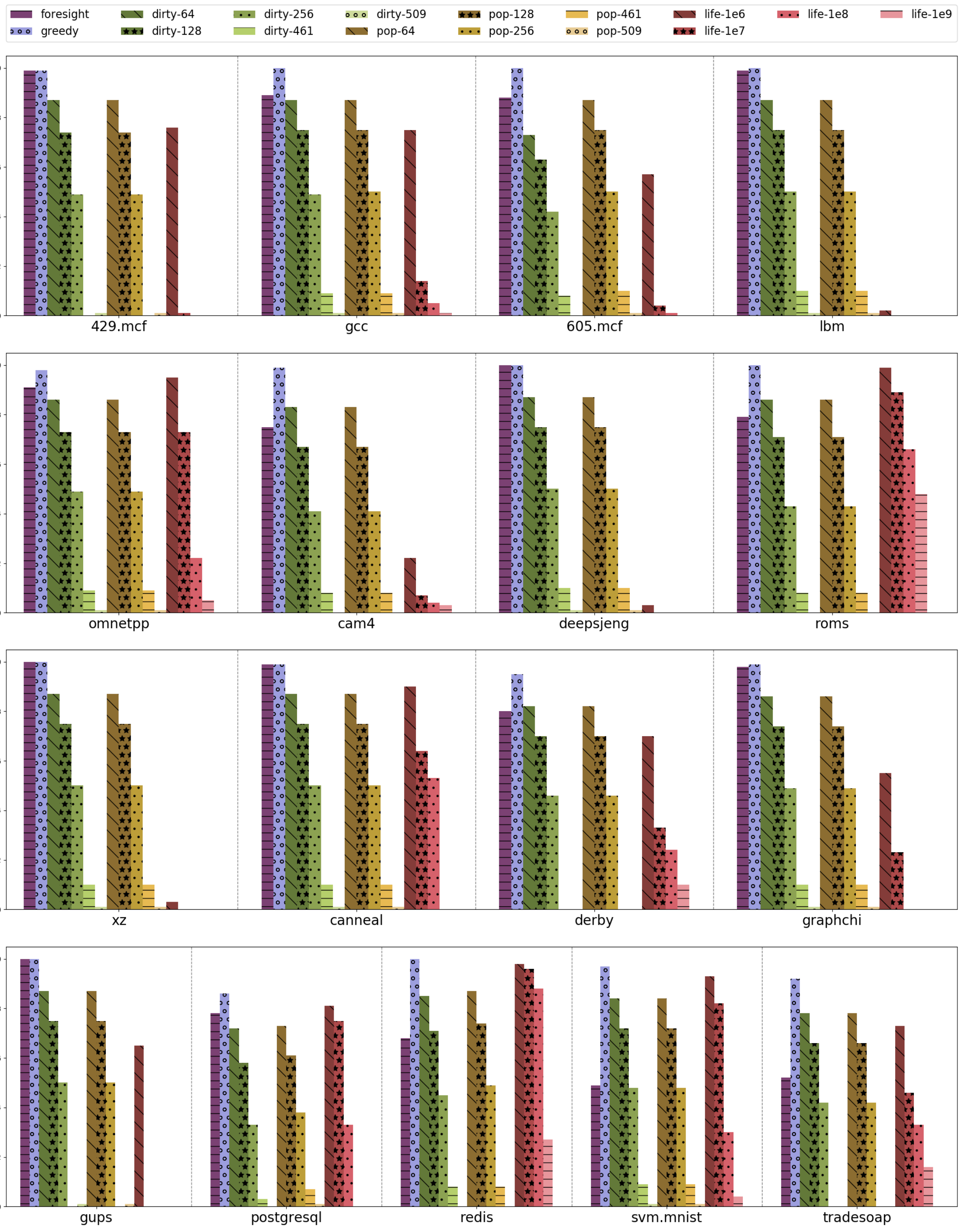


Figure 6.4 : Saved page faults proportional to total faults (0 for *4K-user* or *FreeBSD*).

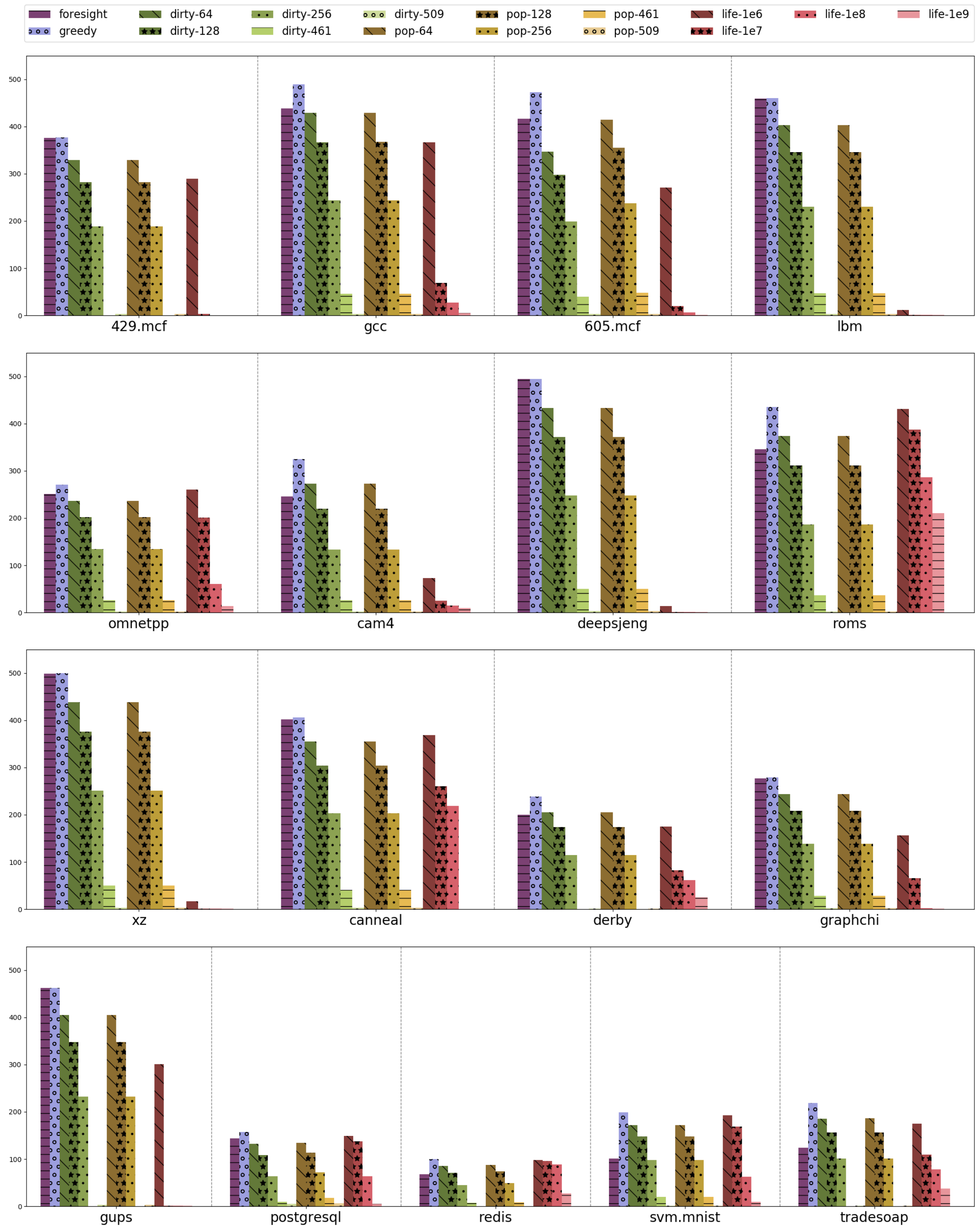


Figure 6.5 : Saved page faults per touched reservation (0 for *4K-user* or *FreeBSD*).

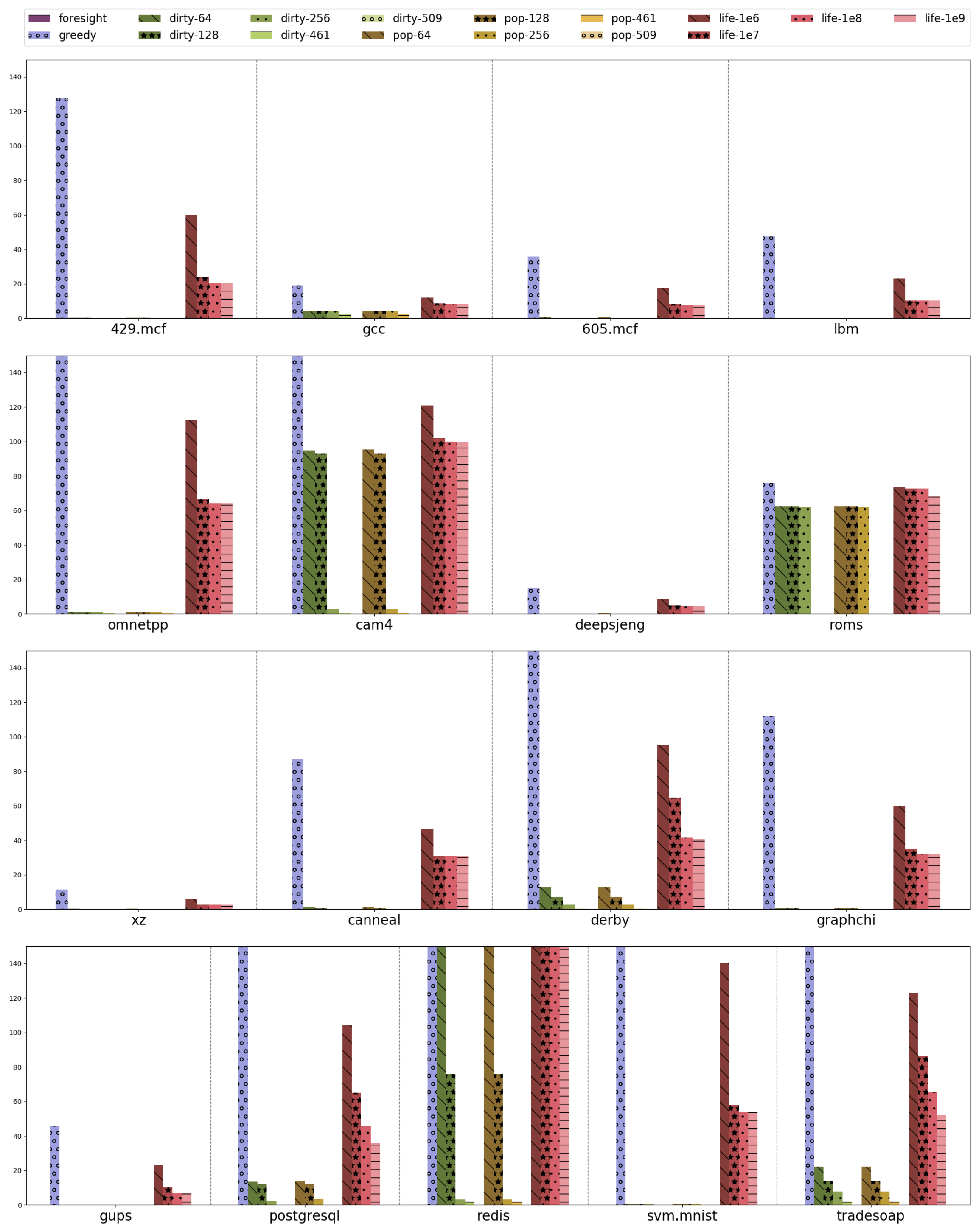


Figure 6.6 : Extra zeroed pages per reservation backing anonymous memory

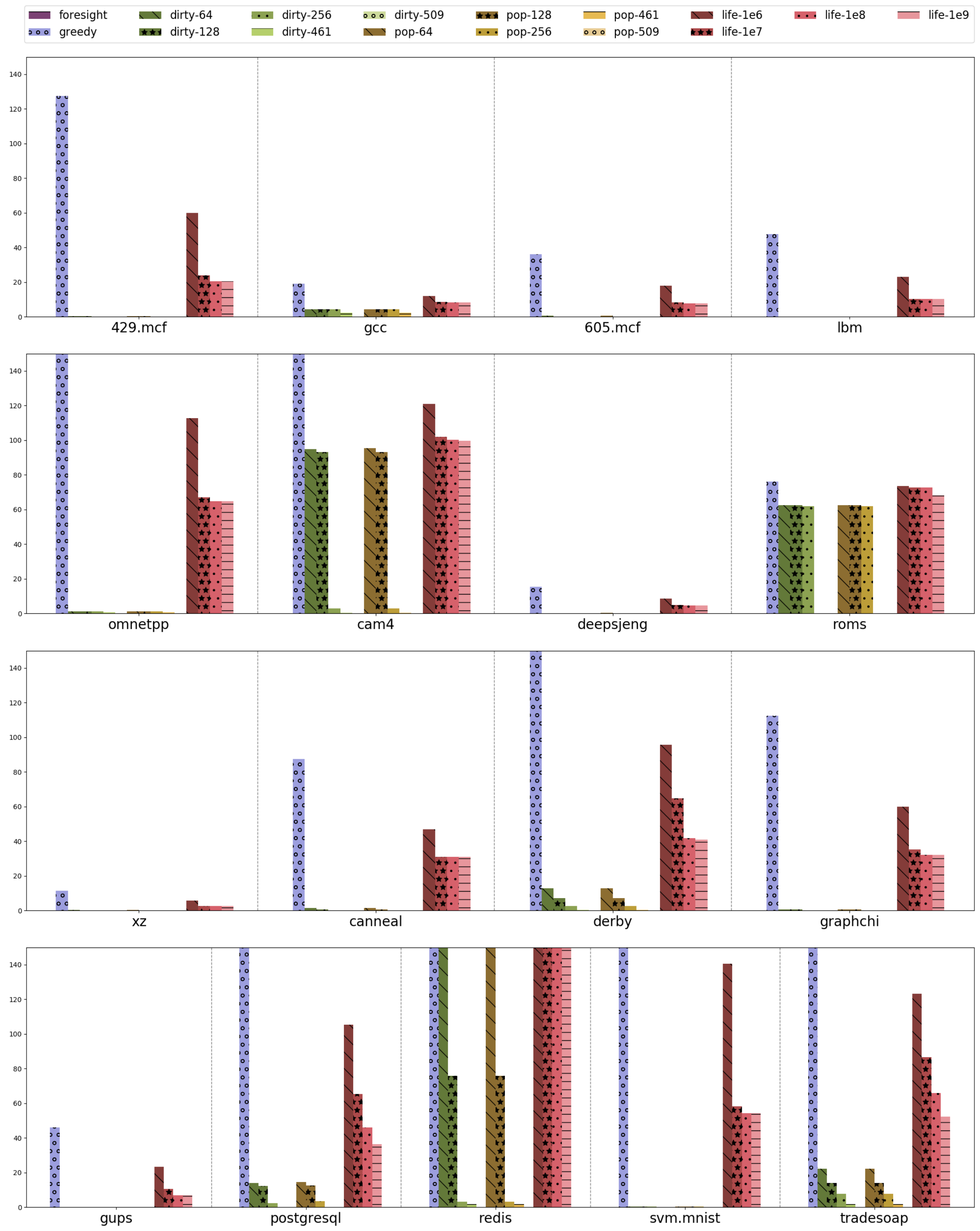


Figure 6.7 : False dirty pages per reservation backing anonymous memory, burdening the I/O when being swapped out.

Policy	*greedy*	*dirty-x*		*pop-x*				*life-x*			
		64	*128*	*64*	*128*	*256*	*461*	10^6	10^7	10^8	10^9
	Average number of extra disk-read pages										
gcc	443.5	0	0	71	71	0	0	443.5	317.75	317.75	317.75
omnetpp	437	0	0	267	120.67	0	0	267	267	267	267
cam4	476.25	0	0	109	0	0	0	476.25	476.25	476.25	476.25
roms	488.5	0	0	0	0	0	0	488.5	488.5	488.5	488.5
derby	264.8	67.6	67.6	162.6	80.8	13.2	2.2	264.8	197.2	95	95
graphchi	433	0	0	181.5	181.5	0	0	433	433	433	433
tradesoap	264	67.6	67.6	161.8	80.8 13.2	2.2	264	196.4	94.2	94.2	
	Average number of false dirty pages										
gcc	509.75	0	0	128	128	0	0	509.75	384	384	384
omnetpp	507.67	0	0	337.67	167	0	0	337.67	337.67	337.67	337.67
cam4	507	0	0	128	0	0	0	507	507	507	507
roms	511	0	0	0	0	0	0	511	511	511	511
derby	477.2	67.6	67.6	374.8	272.4	204.8	102.4	477.2	409.6	307.2	307.2
graphchi	507	0	0	251	251	0	0	507	507	507	507
tradesoap	477.2	67.6	67.6	374.8	272.4	204.8	102.4	477.2	409.6	307.2	307.2

Table 6.2 : Disk I/O overheads for reservations mapping disk files. Policies with all zero values are excluded.

Benchmark	Reservations backing disk files
gcc	4
omnetpp	3
cam4	4
roms	2
derby	5
graphchi	2
tradesoap	5

Table 6.3 : Reservations backing disk files for each benchmark. Benchmarks without such reservations are not shown.

6.2.5 Disk I/O Overhead

Table 6.2 summarizes the disk I/O overhead for benchmarks who have reservations mapping disk files. Unfortunately, applications tend not to process disk I/O via direct allocation (mmap), but rather use file operations (such as read and write). Table 6.3 summarizes the number of such reservations for each benchmark. Among the 17 benchmarks, the maximum number of reservations mapping disk files is 5. While inconclusive, this sparse data suggests that using a threshold of dirty pages could be a desirable choice.

DTLB design	*FreeBSD*	*pop-64*
64(4KB)+32(2MB)	1.00	0.25
64(4KB/2MB)	0.99	0.15
96(4KB/2MB)	0.71	0.10

Table 6.4 : Normalized DTLB misses of 3 DTLB designs on 605.mcf

6.3 Discussion of Policies

6.3.1 Combination of policies in real OS implementation

The operating system should consider using an aggressive population or dirtiness threshold, such as 64, to promote superpages for anonymous memory. The overall comparison in table 6.5 shows that *pop-64* and *dirty-64* achieve performance benefits close to *greedy* while introducing far less page-zeroing overheads. The *lifetime* policies are not as good at limiting page-zeroing overheads, but the reductions in address translation overheads may still justify the additional overhead.

It is advisable to separate the promotion threshold for reservations mapping disk files, because disk I/O is much more expensive. Besides, the write-protection strategy should be utilized as files are sometimes mapped in memory without further modifications. For example, the OS can promote a clean file reservation with a population threshold of 256. When it is demoted upon modification, the second promotion can require a dirtiness threshold of 256. For derby, such a strategy would only incur an extra disk-in overhead of 13.2 4KB pages per reservation, without any disk-out overhead.

					dirty-x				
Policy	*4K-user*	*FreeBSD*	*greedy*	*foresight*	*64*	*128*	*256*	*461*	*509*
DTLB misses	28.43	1.00	0.53	0.92	0.55	0.56	0.67	0.77	0.94
STLB misses	1347.85	1.00	0.15	0.65	0.22	0.25	0.36	0.52	0.79
Page walk latency	1381.76	1.00	0.14	0.64	0.21	0.24	0.36	0.52	0.79
Page fault reduction	0.00	0.00	347.89	315.42	298.57	254.07	166.65	23.94	2.73
Page zero overhead	0.00	0.00	128.48	0.00	23.81	16.06	5.16	0.43	0.01

	life-x				*pop-x*				
Policy	10^6	10^7	10^8	10^9	*64*	*128*	*256*	*461*	*509*
DTLB misses	0.57	0.57	0.58	0.59	0.53	0.53	0.67	0.77	0.94
STLB misses	0.29	0.37	0.42	0.46	0.19	0.23	0.36	0.52	0.79
Page walk latency	0.29	0.37	0.42	0.46	0.18	0.22	0.35	0.52	0.79
Page fault reduction	202.07	90.68	54.18	19.64	302.53	257.72	169.49	24.84	2.84
Page zero overhead	74.42	52.02	47.11	45.34	23.89	16.09	5.23	0.43	0.01

Table 6.5 : Overall performance of 18 policies. DTLB misses, STLB misses and page walk latency are the mean number of those normalized to *FreeBSD* policy over all benchmarks. Page fault reduction and page zero overhead are quantified by the average number of 4KB pages per reservation (≤ 512).

6.3.2 Malloc Designs Unaware of Superpages

Additionally, the fact that *greedy* incurs so much page zeroing overhead indicates that the design of system memory allocator is not aware of superpages. There are two issues with malloc. First, while recent implementations consider fragmentation issues [55, 56, 57], they are not aware of superpage alignment issues. Second, malloc frequently overallocates anonymous memory (via mmap), resulting in many reservations that are created but hardly utilized. Therefore, the former issue fragments the memory at the reservation level and the later issue introduces extra overhead of using superpages. Future malloc implementations should consider the existence of the reservation system, thus serving superpage promotions better.

6.4 Discussion on TLB designs

Among the 17 benchmarks, the TLB simulation shows that policy *FreeBSD* has an overall STLB miss rate of $3.6 \cdot 10^{-4}$, but a higher overall DTLB miss rate of $8.5 \cdot 10^{-3}$. And the higher DTLB miss rate is even underestimated as explained in chapter 4. However, table 6.5 shows that the DTLB performance is less affected than the STLB performance by changing superpage promotion policies. For example, policy *pop-64* can reduce STLB misses by 81% but DTLB misses by only 47%.

Table 6.4 explores how TLB performance is affected by different combinations of TLB designs and superpage promotion policies. The Skylake TLB uses a split DTLB with 64 entries for 4KB pages and another 32 entries for 2MB pages. Additionally, a unified DTLB with 64 entries and another unified DTLB with 96 entries are simulated. The DTLB designs are combined with policies *FreeBSD* and *pop-64* on 605.mcf benchmark. This shows that, as would be expected, more aggressive superpage promotion policies benefit more from unified TLB designs, as there are more entries available to cache superpage translations.

Chapter 7

Conclusion and Future Work

7.1 Conclusions

Modern applications process more and more data. Meanwhile, DRAMs are becoming larger and cheaper, allowing applications to efficiently use more memory. Consequently, the growing memory usage requires faster and more efficient address translation, which is heavily dependent on how many translation entries can be cached by the TLB hardware. However, the TLB look-up is on the critical path of instruction executions, so the number of TLB entries cannot grow as fast as DRAM capacity. This work focuses on superpages which can provide efficient address translation by largely increasing the TLB coverage without requiring more chip space. Because existing superpage support is falling behind applications' demands. The transparent huge page (superpage) support in Linux is too aggressive to introduce more I/O overhead, making it impractical; the reservation-based superpage support in FreeBSD is too conservative to capture all benefits of TLB performance.

This work focuses on exploring the design space of superpage promotion policies to provide a faster address translation for modern applications running on existing hardware. A data collection infrastructure is built to precisely trace the memory accesses. With the memory trace, a simulator is able to simulate different superpage promotion policies in the design space. Moreover, the TLB characteristics are reverse-engineered so the simulator can accurately evaluate the TLB performance of commercial processors under different promotion policies. The simulation based approach allows fast evaluation of a wide variety of different promotion policies without implementing them in real OSes. Among the design space, promotion policies considering the population (number of faulted 4KB pages), dirtiness (number of modified 4KB pages) and lifetime (number of executed memory accesses since creation) of reservations are evaluated. As a result, population-based promotion policies can efficiently promote superpages covering the dominant 4KB pages causing TLB misses, while these superpages incur negligible overheads.

This work provides a data collection infrastructure which can collect a large number of memory accesses to be representative to reflect modern applications' performance, because it is well implemented for collecting memory accesses at an order of 10^{12}. Besides, this data collection infrastructure built on QEMU can accurately collect memory references from both kernel space and user space and compress them on the fly. In the meantime, corresponding page table entries are fetched accompanied with the virtual addresses by page walk code in QEMU. Unlike common memory tracing tools which can only collect accesses for applications running in seconds, it allows tracing modern applications for 10 minutes or more (their running progress in real world) such that the memory trace can cover a large enough interesting memory access patterns for simulating the TLB behavior. Such feature is important for simulating TLB performance, because modern applications use and access far more memory. And this work used profiling to observe that a window of at least 10^{11} memory accesses is required to represent the TLB behavior for modern applications, which length is not well supported by previous tools but easily by the data collection infrastructure from this work.

Previous memory tracing tools are usually maintained for specific OS platforms and hardware architecture, like Linux and x86. They also require complicated kernel instrumentations to know the page table information, because these tools cannot collect memory accesses in the kernel space. And it is hard to guarantee that the page table entry is simultaneously fetched for each memory access. In the data collection infrastructure provided by this work, the above restrictions or limitations do not exist. The memory tracing happens in QEMU's emulation time, where it does not care about which operating system is running. At the same time, it can be easily extended to trace all kinds of architecture that QEMU supports, with collecting the page table information simultaneously.

It is also essential to use accurate TLB simulation algorithm to evaluate TLB performance for commercial processors with the collected trace. This work finds that the TLB in commercial processors, like the Intel Skylake processors, behaves very differently from what people would expect. By reverse engineering the TLB behavior with dependency chain benchmarks, the indexing functions, inclusiveness (how entries are inserted and evicted) and replacement policies of Intel Skylake TLBs are figured out. Moreover, this work is the first to compare the simulated TLB performance with the real-world case with the plotted TLB miss rate curves. Previous work using simulation methods only provides the total number of TLB misses without validating that the TLB performance is consistently faithfully simulated. This work guarantees that the simulation-based evaluation can well predict TLB performance under different superpage promotion policies.

The design space is quickly explored replying on the support from the data collection infrastructure and the faithful TLB simulation, where algorithms different promotion policies are implemented easily in the simulator. Three dimensions of each reservation are mainly considered for deciding their promotion, including population (number of faulted 4KB pages), dirtiness (number of modified 4KB pages) and lifetime (number of executed memory accesses since creation). Though population and dirtiness can be directly monitored, the lifetime is measured by an approximation from the number of memory accesses because time is not accurate in QEMU emulated guests.

Among 17 benchmarks, it is found easy to determine a good population or dirtiness threshold for most benchmarks, where the threshold is good because the promotion policy can gain most TLB performance benefits (performance close to the most aggressive promotion policy) while incurring negligible I/O overheads. Such good thresholds could be either 64 or 128, meaning that 64 or 128 4KB pages are faulted or modified in a reservation containing 512 4KB pages.

The underlying reason that these good threshold could exist is because real-world applications share some common memory accessing behaviors. This work found a ubiquitous memory access pattern in various real-world applications that frequently accessed reservations happen to be the mostly populated/modified ones. In other words, real-world applications do not access memory randomly, providing an important insight for the design of superpage promotion policy in general-purpose operating systems. Therefore, it is possible to precisely predict (over 80%) that a reservation will be fully or mostly populated in the future upon reaching a population/dirtiness threshold of 64, promoting which will result in negligible I/O overheads and save many future 4KB page faults. Larger population/dirtiness threshold like 128 will make the prediction even more precise and further reduce the I/O overheads from superpage promotion. At the same time, promoting such reservations can gain dominant address translation benefits, because these heavily used reservations can easily overwhelm the TLBs with many 4KB pages (3 such reservations can almost fulfill one Skylake STLB). Using superpages for them not only avoids the flooding issue but also enjoys the fact that they dominate most memory accesses. Therefore, utilization-aware superpage promotion policies, like *pop-64* or *pop-128*, and *dirty-64* or *dirty-128* can bring near-ideal trade-offs.

Additionally, the lifetime based promotion policies do not constantly perform well. The lifetime threshold denotes how long the reservation stay in the memory since its creation, so lifetime-based promotion is considering promoting long-lived reservations to superpages. However, different benchmarks have different ways of using reservations. Reservations may be quickly used just after its creation, or much later after some processing. Thus it is hard to determine a uniformly good lifetime threshold for different benchmarks. Furthermore, the lifetime-based promotion cannot effectively avoid I/O overheads, indicating that long-lived reservations are not necessarily the ones mostly populated or modified. This is not surprising because applications do not usually free memory immediately, but let the OS free it when all computation is finished.

The above exploration of superpage promotion policies shows that there is still room to improve the TLB performance. The reservation-based superpage policy in FreeBSD should be encouraged to be more aggressive and gain better TLB performance. Though Linux does not provide a reservation mechanism, the developers should be able to realize why its transparent huge page policy is treated as impractical thus usually suggested to be disabled.

This work also found that the hardware TLB should make changes for supporting superpages when OSes choose wiser superpage promotion policies. As superpages are becoming more practical in Linux and more aggressively promoted in FreeBSD, there will be more 2MB TLB entries. Hardware vendors should consider not limiting the number of entries for 2MB pages by either increasing the number of 2MB entries or making the TLB entries shared for both 4KB and 2MB page sizes.

7.2 Future Work

The dynamic system allocator malloc was not well designed for supporting reservations superpages. Because many reservations are found to be hardly populated, causing aggressive superpage promotion policies to incur I/O huge overheads. However, the performance of superpage management system in FreeBSD is very dependent on how well the 4KB pages are allocated in reservations. There are two considerations for designing malloc to support better superpage management. First, though previous malloc designs consider avoiding memory fragmentation, it is unaware of 2MB boundaries. Memory tend to fragment when applications keep running, which makes it even worse for 2MB bounded reservations. Considering reducing fragmentations in 2MB boundaries should allow more reservations to be available to be promoted. Second, malloc makes unnecessary mmap calls. Reservations are created in response to these mmap calls, where only a small amount of the mapped memory is really needed by applications. The side effect is that these unnecessarily created reservations never get a chance to be promoted under a wise policy, but heavily hurt the overall performance under aggressive policies, like *greedy* in this work. Making the malloc design aware of superpages may make OSes without reservation systems have a better superpage allocation policy.

The implementation of superpage policies in real-world OSes should also consider other aspects. First, the overhead of both initialization or disk writing can be overlapped. For example, Linux is already able to asynchronously allocate transparent huge pages to overlap the higher page fault latency for 2MB pages [29]. Second, the early promoted superpages should be carefully swapped out or flushed to disks, as the OS can no longer track the utilization of these superpages. Memory bloating issues may also happen if reservations are promoted too aggressively. Therefore, the OS should be able to dynamically adjust the promotion thresholds to carefully determine the degree of being aggressive depending on the memory pressure, preventing the worst case from being worse. The superpage management may also consider using a combination of promotion thresholds or policies on different applications and on different types of reservations between anonymous memory and those mapping disk files.

On ARM architecture where 16KB superpages are supported, two-stage promotion may be considered because both 16KB and 2MB can be frequently used by modern applications. As suggested by the desirable performance of *pop-64*, a conservative promotion on 16KB superpages and a threshold-based more aggressive promotion for 2MB superpages could be reasonable, since applications tend to have either more than 64 pages or few pages.

The data collection infrastructure may be extended to support accurate simulation of more benchmarks. It would be interesting to simulate parallel computing benchmarks with multiple CPU cores under unbiased scheduling, and also interesting to simulate time-accurate, unbiased server-client interactions from network benchmarks.

Finally, new hardware innovations may help eliminate the disk writing or swapping overheads of using superpages by marking the dirty bits from cache write-backs instead of TLB page walks. The TLB marks dirty bits on the granularity of the page size, while a cache-line is commonly 64 Byte independent of page sizes. Marking dirty bits with cache-line may eliminate the overhead issue of using superpages once for all, but requires cycle-accurate simulations to validate the feasibility.

Bibliography

[1] "Redis suggests disabling Linux kernel feature transparent huge pages. ." `https://redis.io/topics/admin`. [Online; accessed October-2018].

[2] "MongoDB suggests disabling Linux kernel feature transparent huge pages. ." `https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/`. [Online; accessed October-2018].

[3] "IBM suggests disabling Linux kernel feature transparent huge pages. ." `http://www-01.ibm.com/support/docview.wss?uid=swg21664088`. [Online; accessed October-2018].

[4] T. W. Barr, A. L. Cox, and S. Rixner, "Spectlb: a mechanism for speculative address translation," in *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA*, pp. 307–318, 2011.

[5] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *17th International Conference on High-Performance Computer Architecture (HPCA-17 2011), February 12-16 2011, San Antonio, Texas, USA*, pp. 62–63, 2011.

[6] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative TLB for chip multiprocessors," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, pp. 359–370, 2010.

[7] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pp. 237–248, 2013.

[8] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pp. 26–35, 2008.

[9] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual. ." `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`. [Online; accessed October-2018].

[10] T. Johnson, D. Shasha, *et al.*, "2q: a low overhead high performance bu er management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, pp. 439–450, 1994.

[11] A. Seznec, "A case for two-way skewed-associative caches," in *ACM SIGARCH computer architecture news*, pp. 169–178, ACM, 1993.

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

[13] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures

with multiple page sizes," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 435–448, 2017.

[14] A. Bhattacharjee and D. Lustig, "Architectural and operating system support for virtual memory," *Synthesis Lectures on Computer Architecture*, vol. 12, no. 5, pp. 1–175, 2017.

[15] F. Bodin and A. Seznec, *Skewed associativity enhances performance predictability.* ACM, 1995.

[16] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pp. 187–198, IEEE, 2010.

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

[18] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*, pp. 258–269, 2012.

[19] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pp. 558–567, 2014.

[20] M. Clark, "A new$\times$ 86 core architecture for the next generation of computing," in *Hot Chips 28 Symposium (HCS), 2016 IEEE*, pp. 1–19, IEEE, 2016.

[21] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pp. 178–189, 2014.

[22] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pp. 66–78, 2015.

[23] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," in *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, pp. 48–59, 2010.

[24] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th annual international symposium on Microarchitecture*, pp. 51–61, ACM, 1991.

[25] J. Navarro, S. Iyer, P. Druschel, and A. L. Cox, "Practical, transparent operating system support for superpages," in *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, 2002.

[26] M. Talluri and M. D. Hill, "Surpassing the TLB performance of superpages with less operating system support," in *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994.*, pp. 171–182, 1994.

[27] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing TLB and memory overhead using online superpage promotion," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*, pp. 176–187, 1995.

[28] Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee, "Reevaluating online superpage promotion with hardware support," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01), Nuevo Leone, Mexico, January 20-24, 2001*, pp. 63–72, 2001.

[29] "Transparent huge pages. ." `https://www.kernel.org/doc/Documentation/vm/transhuge.txt`. [Online; accessed October-2018].

[30] 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 2016, Savannah, GA, USA, November 2-4, 2016.*, pp. 705–721, 2016.

[31] N. Ganapathy and C. Schimmel, "General purpose operating system support for multiple page sizes," in *1998 USENIX Annual Technical Conference, New Orleans, Louisiana, USA, June 15-19, 1998*, 1998.

[32] "Persistent huge pages. ." `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`. [Online; accessed October-2018].

[33] A. Arcangeli, "Transparent Hugepage Support. ." `https://www.linux-kvm.org/images/9/9e/2010-forum-thp.pdf`. [Online; accessed October-2018].

[34] "Idle Page Tracking. ." `https://www.kernel.org/doc/html/latest/`

`admin-guide/mm/idle_page_tracking.html`. [Online; accessed October-2018].

[35] "Windows Large Page Support. ." `https://docs.microsoft.com/en-us/windows/desktop/memory/large-page-support`. [Online; accessed October-2018].

[36] "OS X Superpage Support. ." `https://www.unix.com/man-page/osx/2/mmap/`. [Online; accessed October-2018].

[37] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski, "Multiple page size modeling and optimization," in *14th International Conference on Parallel Architecture and Compilation Techniques (PACT 2005), 17-21 September 2005, St. Louis, MO, USA*, pp. 339–349, 2005.

[38] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, p. 46, 2005.

[39] Y. Collet *et al.*, "Lz4: Extremely fast compression algorithm," *code. google. com*, 2013.

[40] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, pp. 190–200, ACM, 2005.

[41] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[42] J. Bucek, K.-D. Lange, *et al.*, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 41–42, ACM, 2018.

[43] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 72–81, ACM, 2008.

[44] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko, "Specjvm2008 performance characterization," in *SPEC Benchmark Workshop*, pp. 17–35, Springer, 2009.

[45] M. F. Cowlishaw, "The 'telco'benchmark," *URL: http://www2. hursley. ibm. com/decimal, 3pp, IBM Hursley Laboratory*, 2002.

[46] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pp. 31–46, 2012.

[47] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pp. 177–187, ACM, 2005.

[48] I. Earl Joseph, "Gups (giga-updates per second) benchmark," *URL http://www. dgate. org/˜ brg/files/dis/gups*, 2000.

[49] B. Momjian, *PostgreSQL: introduction and concepts*, vol. 192. Addison-Wesley New York, 2001.

[50] J. L. Carlson, *Redis in action.* Manning Publications Co., 2013.

[51] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[52] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[53] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: java benchmarking development and analysis," in *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pp. 169–190, 2006.

[54] "Apache DayTrader Benchmark Sample." `http://geronimo.apache.org/GMOxDOC20/daytrader.html/`. [Online; accessed October-2018].

[55] C. Lever and D. Boreham, "Malloc () performancein amultithreadedlinuxen vironment," 2000.

[56] A. Bohra and E. Gabber, "Are mallocs free of fragmentation?," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 105–117, 2001.

[57] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the bsdcan conference, ottawa, canada*, 2006.