Deduplicating TLB entries for shared pages

Jeongseob Ahn^{a)}

*Department of Software and Computer Engineering, Ajou University,
Suwon, South Korea*

a) jsahn@ajou.ac.kr

Abstract: Although the page sharing among different address spaces can effectively reduce the memory footprint, the corresponding address mappings still require their own TLB entries. Consequently, redundant address mappings for the shared pages reside in TLBs. Our goal in this study is to increase the TLB density by deduplicating redundant copies of address mappings into a single mapping. In virtualized servers, the case of hosting the same guest operating systems, the same third-party libraries, and the same application is commonly found. Such environments generate the multiple identical address mappings, virtual to physical address, for shared pages among different virtual machines. We exploit this unique characteristic of the virtual memory layout being analogous in the same execution environment, and explore TLB deduplication, called DeTLB, which merges redundant virtual to physical address mappings of different address spaces into a single TLB entry. To demonstrate the performance impact, we develop a QEMU based trace simulator and evaluate the number of TLB misses by running an Apache Spark ALS and a microbenchmark with the Linux KSM module. The results indicate that DeTLB can effectively reduce the number of TLB misses for both data and instructions.

Keywords: virtual memory, TLB, deduplication, page sharing

Classification: Integrated circuits

References

- [1] AMD64 architecture programmers manual volume 2: System programming. Whitepaper.
- [2] Configuring ART. <https://source.android.com/devices/tech/dalvik/configure> [Online; accessed Feb-14-2018].
- [3] Intel 64 and IA-32 Architectures Software Developers Manual. Reference number: 325462-057US. <https://software.intel.com/en-us/articles/intel-sdm> [Online; accessed Feb-14-2018].
- [4] Linux Pagemap. <https://www.kernel.org/doc/Documentation/vm/pagemap.txt> [Online; accessed Feb-14-2018].
- [5] Linux Test Project. <https://github.com/linux-test-project/ltp> [Online; accessed Feb-14-2018].
- [6] Scaling MySQL With Virident Flash Drives and Multiple Instances of Percona Server. <https://www.percona.com/files/white-papers/scaling-multiple-mysql-percona-server-irident.pdf> [Online; accessed Feb-14-2018].
- [7] A. Arcangeli, *et al.*: “Increasing memory density by using ksm,” OLS (2009).

- [8] S. Barker, *et al.*: “An empirical study of memory sharing in virtual machines,” Proc. of the 2012 USENIX Conference on Annual Technical Conference (ATC) (2012).
- [9] S. Boyd-Wickizer, *et al.*: “An analysis of Linux scalability to many cores,” Proc. of the 9th Symposium on Operating Systems Design and implementation (OSDI) (2010).
- [10] E. Bugnion, *et al.*: “Disco: Running commodity operating systems on scalable multiprocessors,” Proc. of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP) (1997).
- [11] X. Dong, *et al.*: “Shared address translation revisited,” Proc. of the Eleventh European Conference on Computer Systems (EuroSys) (2016) (DOI: [10.1145/2901318.2901327](https://doi.org/10.1145/2901318.2901327)).
- [12] M. Ferdman, *et al.*: “Clearing the clouds: A study of emerging scale-out workloads on modern hardware,” Proc. of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2012) (DOI: [10.1145/2150976.2150982](https://doi.org/10.1145/2150976.2150982)).
- [13] D. Gupta, *et al.*: “Difference engine: Harnessing memory redundancy in virtual machines,” Proc. of the 8th Symposium on Operating Systems Design and implementation (OSDI) (2008).
- [14] Y. Jang, *et al.*: “Breaking kernel address space layout randomization with intel TSX,” Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS) (2016) (DOI: [10.1145/2976749.2978321](https://doi.org/10.1145/2976749.2978321)).
- [15] Y. A. Khalidi and M. Talluri: “Improving the address translation performance of widely shared pages,” Technical Report (Mountain View, CA, USA, 1995).
- [16] G. Milós, *et al.*: “Satori: Enlightened page sharing,” Proc. of the 2009 USENIX Conference on Annual Technical Conference (ATC) (2009).
- [17] G. Pekhimenko, *et al.*: “Base-delta-immediate compression: Practical data compression for on-chip caches,” Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT) (2012).
- [18] D. Skarlatos, *et al.*: “Pageforge: A near-memory content-aware page-merging architecture,” Proc. of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (2017) (DOI: [10.1145/3123939.3124540](https://doi.org/10.1145/3123939.3124540)).
- [19] Y. Tian, *et al.*: “Last-level cache deduplication,” Proc. of the 28th ACM International Conference on Supercomputing (ICS) (2014) (DOI: [10.1145/2597652.2597655](https://doi.org/10.1145/2597652.2597655)).
- [20] C. A. Waldspurger: “Memory resource management in VMware ESX server,” Proc. of the 5th Symposium on Operating Systems Design and implementation (OSDI) (2002) (DOI: [10.1145/844128.844146](https://doi.org/10.1145/844128.844146)).

1 Introduction

In cloud computing, the server consolidation technique hosting multiple virtual machines (VMs) on a physical server has been widely used to improve system utilization and energy efficiency. In virtualized systems, the memory is spatially partitioned and each memory region is associated with a single VM at a time. As the requirement of main memory size in VMs continues to increase, the large memory footprint of VMs is a major obstacle to increase server consolidation levels.

To reduce the memory consumption, memory sharing technique among virtual machines has been rehabilitated from clouds to mobile systems. In consolidated

environments, we may see that a single server hosts the same guest operating systems of virtual machines, the same third-party libraries, and even the same applications [20]. To exploit the qualities, the traditional page sharing technique has been extended to be able to deduplicate the same content pages between virtual machines, called content-based page sharing [7, 8, 10, 20]. The content-based page sharing becomes a popular technique to reduce the memory footprint in virtualized systems. This technique is not limited to virtualized environments. In native computing, redundant copies of pages such as code regions are deduplicated. Since running multiple instances of the same application has been preferred in large scale datacenters to scale performance of some applications [6, 9], the page deduplication can be effective to save the memory footprint in modern computer systems. In mobile and embedded systems, there are many efforts to reduce the memory footprint because these do not have sufficient memory spaces. For example, in the Android environment, all the processes are inherited from the startup process called *Zygote*, which preloads commonly used shared libraries to reduce the memory footprint [2]. Another major source of memory sharing is the common dataset. In particular, many machine learning and data analytic applications share massive data repositories to produce different models and test various algorithms. Since data is getting bigger and bigger, memory sharing would be more effective.

Although the memory footprint can be reduced by deduplicating redundant copies of the same content pages, the corresponding address mappings still require their own TLB entries, which are redundant, because TLBs have not been designed to be aware of shared pages. The key observation is that address mappings between virtual page numbers and physical frame numbers for the shared regions are highly likely to be identical between the same guest OSes and the same applications due to the same virtual memory layout. To eliminate those redundant copies of TLB entries, we introduce a HW technique, called DeTLB, deduplicating redundant TLB entries for the shared regions. Our design of TLB deduplication exploits the identical virtual memory layout across different address spaces. Deduplicating redundant address mappings in TLBs is able to increase the TLB density because the saved TLB entries can be utilized for others. As a result, the performance of TLB sensitive applications increases.

However, the virtual memory layout might be changed at every single run in traditional OS environments such as Linux and Windows employing the address space layout randomization (ASLR) technique. ASLR has been widely used to defend against return-oriented programming attacks. It means that the address mappings of the shared pages may not become identical because the virtual page numbers of the address mappings can be difference between address spaces. We may lose the opportunity to deduplicate the mappings. Thus, ASLR is a major obstacle to prevent TLB deduplication. We discuss the effectiveness of DeTLB when running with the ASLR enabled operating systems and discuss a couple of possible solutions to overcome the limitation.

In this study, we first demonstrate cases of sharing the same content pages, which have redundant virtual to physical page mappings, by running multiple virtual machines on a KVM host server with the Kernel Same-page Merging (KSM) module enabled. To see the potential impact of DeTLB, we also measure

how many TLB misses can be reduced on our QEMU based simulation environment by running an Apache Spark application and a microbenchmark. Our preliminary results show that data and instruction TLB misses can be reduced by up to 48% and 49%, respectively for microbenchmark and instruction TLB misses for the Spark application are reduced by 26%. Measuring performance behavior with a timing model is our on-going study.

2 Background and motivation

2.1 Memory deduplication

Memory sharing is a well-known technique to effectively reduce the memory footprint in modern computer systems. For example, as most applications running on a system share the standard library (e.g., `libc`) code, each application does not need to maintain their own copy of the shared library code. Instead of having redundant copies of the same code regions, merging the redundant pages allows to increase the memory density. This concept is further extended to the contents of pages [10]. Data pages including a common data set can be shared in data intensive applications to decrease the memory footprint. This advantage becomes more powerful in virtualized clouds and large-scale datacenters.

In virtualized systems, we can easily see the case where a host server is running multiple virtual machines of the same guest operating system, the same third-party libraries, and even the same applications [8, 20]. This enables to increase the consolidation ratio on a single server by deduplicating redundant copies into a single one.

At scale, running multiple instances of the same application has been a preferred way because multi-process versions are known to present better performance and fault isolation, compared to a beefy multi-threaded version, in a couple of cases. For instance, memcached, which is a popular object caching system, does not scale well with a multi-threaded version owing to the lock-based key-value hash table [9]. As a result, deploying multiple instances of memcached has been recommended. According to Tkachenko *et al.*, MySQL shows better performance with multiple instances than a single instance [6]. In such environments, deduplicating the same content pages across different processes can reduce the memory footprint. In Linux, the KSM (Kernel Same-page Merging) technique has been implemented for merging the same content pages among different processes (or VMs) [7] and VMware has also developed their in-house technique called content-based page sharing [20].

In this study, we analyzed the behavior of KSM when running two virtual machines on a single server where each virtual machine runs the memcached application. We wrote a program to obtain the address information by using the Linux `pmap` interface [4] in both the host and guest operating systems. Fig. 1(a) shows the guest virtual (vpn) to host physical address (pfn) mapping information for each memcached instance. For the shared libraries including `libc`, `pthread`, and event libraries, the physical frame number (pfn) is identical between two memcached processes. Only the stack and heap areas are mapped into different physical pages because those would not be shared in general. Fig. 1(b) shows how

Fig. 1. Virtual to physical page mapping information and memory footprint for two memcached instances

many pages can be shared by enabling KSM. As expected, the memory footprint is effectively reduced. Around 40% of pages is merged by page sharing (white portion). Note that we did not populate any data to see the effect of code sharing and did not account for the region of operating systems in this experiment due to our user-level tool. Thus, the savings which we measured only came from sharing of the code segments of the memcached application. We believe that the portion of savings would increase when accounting for the operating systems regions.

As contemporary TLBs have been designed to cache address mappings without considering the content-based page sharing cases, the limited space of TLBs has not been effectively utilized. The primary goal of this study is to improve TLB performance by deduplicating redundant TLB entries.

2.2 Related work

There have been several studies optimizing memory deduplication techniques to further reduce memory footprint [8, 13, 16]. Since those prior work focused on the aspect of memory deduplication, redundant TLB entries indicating the same address mappings still exist. Recently, Dong *et al.* proposed deduplicating page tables for the shared region with modifications of manipulating page tables in the Android environment [11]. DeTLB is inspired by this study. Unlike this software technique, we explore a HW technique which does not require modifications of operating systems so that it is transparent to software. We explore a TLB deduplication technique to merge redundant TLB entries across different address spaces. For the shared pages between multiple instances of the same application or between different virtual machines, we can effectively reduce the TLB footprint, resulting in better utilization of TLBs. Khalidi and Talluri explored the case of address translation for shared pages, but the case of content-based page sharing and cloud environment was not considered [15].

Meanwhile, Skarlatos *et al.* introduced a HW technique to accelerate the deduplication of memory pages by exploiting characteristics of the Error Correction Codes (ECC) engine in DRAM [18]. On the other hand, there have been an effort deduplicating cache blocks which have the same data in HW [19]. This results in

Traditional TLB			DeTLB			
ASID	VPN	PFN	ASID or bitmap		VPN	PFN
5582	7FFFF75B3	3F7C6	S	1 1	7FFFF75B3	3F7C6
5692	7FFFF75B3	3F7C6	S	1 1	7FFFF7978	3FF82
5582	7FFFF7978	3FF82	S	1 1	7FFFF7B96	12456
5692	7FFFF7978	3FF82	S	1 1	7FFFF7DDA	3FBC3
5582	7FFFF7B96	12456	P	5582	61C	33729
5692	7FFFF7B96	12456	P	5692	61C	3374B
5582	7FFFF7DDA	3FBC3	P	5582	7FFFFFFFB	33729
5692	7FFFF7DDA	3FBC3	P	5692	7FFFFFFFB	3374B
5582	61C	2A343				
5692	61C	12E4A				
5582	7FFFFFFFB	33729				
5692	7FFFFFFFB	3374B				

Fig. 2. Overview: deduplicating redundant mappings across two different ASIDs

increasing the capacity of caches. Although the concept is similar to our DeTLB, deduplicating the same TLB contents has not been considered.

3 TLB deduplication

This section explores a case for TLB deduplication, called DeTLB, merging the redundant virtual to physical page mappings into a single TLB entry across different address spaces. We first briefly give an overview of DeTLB and then introduce the architecture of DeTLB. Second, we describe how the TLB operations (lookup and fill) need to be extended. Last, we discuss the effectiveness of DeTLB under the ASLR enabled systems which change the virtual memory layout at every single run and possible extension points.

3.1 Overview

DeTLB deduplicates redundant copies of address mappings in TLBs, which come from the page sharing, to increase space utilization of TLBs. Fig. 2 shows differences between the traditional TLB (left) and DeTLB (right). This running example is derived from the case used in Fig. 1. The virtual page, 0x7FFFF75B3, for two virtual machines (5582 and 5692) must be translated into the same physical page, 0x3F7C6, because the page is being shared. So, DeTLB does not have separate address mappings for each virtual machine. Instead, it maintains a sharing bit vector and the corresponding bits belonging to virtual machine 1 and 2 are set to 1. In this example, total four pages are freed by DeTLB and the saved entries will be used for others. As a result, we are able to achieve the TLB misses reduction.

For the private pages (lower), the entries have the ASID value as normal TLBs instead of the sharing bit vector because those mappings do not need to record multiple ASIDs. In DeTLB, the ASID field of traditional TLBs are used to record either the list of sharers for shared pages or an ASID for private pages.

3.2 Architecture

In this section, we present the design and architecture of DeTLB. To merge redundant copies of address mappings belonging to different address spaces for each, modest changes to the traditional memory management unit (MMU) are

Fig. 3. An entry of traditional TLB (upper) and DeTLB (lower)

required. Specifically, the traditional TLB entry format needs to be changed to represent sharers for shared mappings.

Fig. 3 illustrates the format of the traditional TLB entry and the proposed DeTLB entry. Most contemporary TLBs already keep an address space ID (ASID) for each entry to avoid TLB flushes on a context switch. We extend the ASID field to record sharers. We describe our architecture with x86 processors, but it is not limited to specific architectures. Traditional x86 architectures generate an ASID through the built-in logic, Gen ASID, by taking the CR3 register [3]. Basically, the logic is a hash function. Each address space has a CR3 register indicating the base address of each page table. Since DeTLB is required to embed multiple ASIDs for shared mappings, we replace the field of ASID in the traditional TLB entry with a sharing bit vector to keep track of the list of sharers. To do that, we slightly change the Gen ASID logic to generate an index of the sharing bit vector. The dotted line indicates how a DeTLB entry is comprised. In the following, we discuss implementation issues for supporting DeTLB.

Available space: To record sharers in TLB entries, a TLB entry must have enough free space. In the current Intel x86 architecture, 12 bits of a TLB entry are used to keep an ASID [3] and those bits can be used for storing the list of sharers in DeTLB. Otherwise, if the number of sharers exceeds 12, we need to decide whether increasing the size of a TLB entry or limiting the number of sharers. Even though the number of sharers is limited due to the lack of space, it does not cause any correctness issues. This is a design point of our DeTLB. We will explore the sweet spot considering the gain and the HW cost through evaluations. Beyond that, an alternative solution is taking advantage of the reserved bits in the TLB entry because those bits are not used [1].

Invalidation: Since both the traditional ASID and the index of the sharing bit vector from Gen ASID may not be unique values due to the characteristics of the hash. Two different address spaces may generate the same ASID or the same index of the sharing bit vector. In this case, we need to be able to invalidate the ASID in the sharing bit vector. To resolve the conflict, x86 already supports the INVPCID instruction to invalidate mappings corresponding to the ASID. DeTLB can take advantage of this facility to invalidate the index of a sharing bit vector. Although the cost of the Gen ASID logic would be slightly increased in DeTLB, we can replace the ASID bits with the sharing bit vector without majorly restructuring the TLB entry.

3.3 DeTLB operations

For supporting DeTLB, we need to change handling TLB misses in two different ways as presented in Fig. 4. When looking up DeTLB ❶, we take a pair of ASID

Fig. 4. Handling TLB misses and deduplicating redundant mappings in DeTLB

and VPN as the traditional TLB. If we fail to find any matched VPN ②, then it takes the normal path traversing page tables to fill the virtual to physical page mapping ③. Conversely, if we find the matched VPN, but the corresponding bit in the sharing vector is not set ④, it also requires page table walks. The only difference is that before filling the PFN in a new TLB entry, DeTLB takes an additional step to compare the PFN from the page tables to the existing one ⑤. If it is the same thing, instead of allocating a new entry in the TLB, we do deduplicate the entry by simply setting the corresponding bit in the sharing vector. Then, next time, if we find the matched VPN and the corresponding bit is set ⑥, we do not need to traverse the page tables costly. For every TLB access for shared pages, we examine the corresponding bit in the sharing vector for given address space ID while performing the tag matching process. Although DeTLB increases the latency of matching entries and handling TLB misses, it would not be significant because we take the additional step only for shared pages which is indicated in the page table entry.

If one of the sharers updates the shared page, we need to clear the corresponding ASID index for the shared page mapping because the writer is no longer a sharer. Since the read-only attribute of the shared page table entry has been set, we can take the page fault exception that occurs in writing read-only pages as usual.

3.4 Challenges

A typical case where DeTLB is effective is that the same guest OSes of virtual machines, the same applications, and the same library are running in virtualized clouds. Also, DeTLB would be running well on the Android process model. In the Android Runtime (ART) platform, all the processes are inherited from the startup process and the startup process preloads commonly used shared libraries [2]. So, we could deduplicate those pages shared by different address spaces. The source of deduplication is the same virtual memory layout between them. In traditional computing, however, operating systems may enable the ASLR (Address Space Layout Randomization) technique which has been introduced to cope with the return-oriented programming (ROP) attack. For every single run, the virtual memory layout of the same applications is changed. In such a case, it might be difficult to find the mergeable address mappings on ASLR enabled systems.

While exploring the impact of ASLR on DeTLB, we found out that code segments would not be affected unless your applications are compiled with the Position Independent Executable (PIE) option (e.g., `-fPIE` in gcc). We conducted a simple experiment to check whether the PIE option is enabled or not for selected applications under the latest Linux distribution, Ubuntu 16.04. Table I shows that a couple of popular applications were not compiled with the option. All the tested

Table I. PIE compile option in selected applications

Application	PIE Enabled
Apache	✓
Sshd	✓
Bash	✗
Docker	✗
Java	✗
Mysql	✗
Python	✗

applications came from the official Debian package tool (a.k.a APT). For Java, we ran an application of Apache Spark. Although DeTLB can work well on applications compiled without the option, DeTLB may eventually be ineffective when the option is set.

Possible extensions: We investigated how the ASLR technique is implemented to discuss possible extensions to our naive DeTLB. The key technique of ASLR is to add a random number as an offset to the code. Briefly, the x86 Linux implementation relies on the `get_random_int()` kernel function to generate the offset. One possible solution is that if we could let processors know the randomly generated offset in a secure location, then DeTLB would still be able to merge the address mappings by maintaining both the base address and the random offset. Another possible solution is to support ASLR in hardware so that the MMU can be aware of how the memory layout is randomized. As traditional software-based ASLR has been broken [14], it would be an attractive option when considering the concern of the ASLR attacks.

Another facet is deduplicating (or compressing) the TLB entries for the different virtual page number, but the same physical frame number. A straightforward way to keep multiple virtual page numbers per an entry is to adopt the base and delta representation scheme [17] because the number of bits randomized by ASLR is limited. It may increase the cost and complexity of the hardware, but it would better exploit mergeable cases more than pure DeTLB. These explorations are part of our on-going work.

4 Evaluation

Experimental setup: To evaluate our proposed scheme, we modify the QEMU emulator to extract memory access traces and simulate the traditional TLB and DeTLB structures with a single ARM core. Since simulating virtual machines on QEMU is not feasible as of writing this paper, we mimic page sharing cases by running two instances of the Apache Spark application, collaborative filtering (ALS) [12], on a QEMU virtual machine. Although it cannot simulate the sharing case for operating systems and data segments, it is still possible to share code segments for applications. In our future work, we will investigate the methodology to build the cloud-like virtual machine environment and evaluate performance with timing simulations. Table II presents our experimental environment and used applications.

Table II. Experimental environment used for evaluation

Parameter	Value
Processor Model	QEMU-based ARM 64 bits
Instruction TLB	4-way, 128/256/512 entries
Data TLB	4-way, 256/512/1024 entries
Operating System	Ubuntu 16.04
Spark ALS	Movie recommendation application
Microbench	A synthetic scenario triggering KSM

Fig. 5. Relative I-TLB misses for Spark ALS

TLB performance: Since it is difficult to evaluate the case where data segments are deduplicated for existing applications in our limited evaluation environment, we first evaluate how many the number of instruction TLB (I-TLB) misses can be reduced by DeTLB, compared to traditional TLBs while executing one billion instructions of Apache Spark ALS. Fig. 5 shows the relative TLB miss rate by varying the number of TLB entries from 128 to 512 entries with 4-way. DeTLB, by introducing a minor addition to the traditional TLB structures, shows that the TLB misses can be reduced by 26%. By increasing the number of TLB entries, we can observe that the miss rate is gradually improved. As I-TLB is on the critical path of the instruction execution, we expect that DeTLB will improve the overall execution time by freeing the TLB entries which are used to keep redundant copies.

Second, we build a controlled environment to see the performance of DeTLB for the same data pages. To trigger the KSM module for merging the same data pages, we wrote a microbenchmark explicitly sharing an arbitrary data region by leveraging the `madvise` system call with the `MADV_MERGEABLE` option. It is an interface for applications to allow anonymous pages to be shared across processes by KSM. To implement the microbenchmark, we referred to the testcases of the Linux Test Project [5] to mimic the sharing usecases of native applications.

In this evaluation, we generate a 256 MB shared data region for two processes and let the KSM module know that those memory pages are candidates to be merged. We can consider the usecases sharing a dataset in different instances of the same application or running different algorithms on the same dataset. For example, users would run the Pagerank and Bellman-Ford algorithms on the same graph dataset. Due to the simulation time and limited memory of our environment, we scale down the size of data in this experiment. Fig. 6 shows the relative TLB misses for both data and instructions by varying the number of TLB entries. With 1024

Fig. 6. Relative TLB misses for microbenchmark

entries, the number of TLB misses for D-TLB and I-TLB is significantly reduced by 48% and 49%, respectively, compared to traditional TLB. We observe the trend that the number of misses is reduced when decreasing the number of entries because it is difficult for DeTLB to find the identical mappings in the limited size of TLBs. Through the evaluation, we are able to see that our proposed DeTLB is effective and has a potential to improve the overall performance.

5 Summary and future work

In this paper, we proposed deduplicating TLB entries, called DeTLB, to effectively reduce the TLB footprint by merging redundant copies of address mappings across processes or virtual machines. Without major changes to the traditional TLB, DeTLB is able to save the space of TLBs, which could be alternatively utilized by other meaningful things. We believe that DeTLB can increase the consolidation ratio for future big memory applications. In future work, we will investigate the design of DeTLB under the constrained execution environments discussed in Section 3.D and analyze the performance benefit with timing simulations.

Acknowledgments

This research was supported by the National Research Foundation of Korea grant (NRF-2017R1C1B5075437) funded by the Ministry of Science, ICT and future Planning (MSIP) and by the new faculty research fund of Ajou University.