On-Demand Triggered Memory Management Unit in Dynamic Binary Translator

Benyi Xie[1,2], Xinyu Li[1,2], Yue Yan[1,2], Chenghao Yan[1,2], Tianyi Liu[3], Tingting Zhang[1,4], Chao Yang[5], and Fuxin Zhang[1,2(✉)]

[1] SKLP, Institute of Computing Technology, CAS, Beijing, China
{xiebenyi21b,lixinyu20s,yanyue21s,yanchenghao21s}@ict.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
fxzhang@ict.ac.cn
[3] The University of Texas at San Antonio, San Antonio, USA
tianyi.liu@utsa.edu
[4] Loongson Technology Co. Ltd., Beijing, China
zhangtingting@loongson.cn
[5] State Grid Liaoning Electric Power Supply Co. Ltd., Shenyang, China
yangchaoneu@sina.com

Abstract. User-level Dynamic Binary Translators (DBTs) linearly map the guest virtual memory to host virtual memory to achieve optimal performance. When the host page size exceeds the guest page size, multiple small guest pages are mapped to a single large host page, resulting in inappropriate permissions mapping. DBTs face security and correctness risks accessing the inappropriately mapped host page. Our survey reveals that most of the state-of-the-art user-level DBTs suffer from these risks. While system-level DBT can avoid these risks through a software Memory Management Unit (MMU). However, the software MMU fully emulates guest memory management, leading to slower performance than the linear mapping approach of user-level DBTs.

To address the balance of performance and risks, we propose a DBT memory management method named On-Demand Triggered MMU (ODT-MMU), that combines the strengths of both user-level and system-level DBTs. ODT-MMU utilizes linear mapping for non-risky page accesses and triggers a software MMU when accessing risky pages. We implement ODT-MMU in two ways to accommodate various application scenarios: a platform-independent implementation named ODT-InterpMMU, and a hardware-accelerated implementation named ODT-ManipTLB. ODT-ManipTLB is designed for host Instruction Set Architectures (ISAs) that support programmable TLB. Experimental results demonstrate that both implementations can effectively mitigate risks associated with page size. Furthermore, ODT-ManipTLB achieves over 2000x performance improvement compared with the ODT-InterpMMU, while maintaining comparable performance to the DBT without ODT-MMU. Additionally, our work is applied to two industrial DBTs, XQM and LATX.

C. Li et al. (Eds.): APPT 2023, LNCS 14103, pp. 297–309, 2024.
https://doi.org/10.1007/978-981-99-7872-4_17

Keywords: Binary translator · Memory management · Page size · TLB

1 Introduction

DBT enables the emulation of guest binaries on a host machine. Based on the emulation level of the guest, DBTs can be categorized into two types: user-level DBTs, which facilitate the migration of user applications, and system-level DBTs, which facilitate the migration of an OS. It is crucial for both types of DBTs to effectively and efficiently emulate memory management as guest binaries expect. System-level DBTs typically employ a software MMU to emulate the guest physical memory. Due to no need for emulating physical memory, user-level DBTs linearly map guest virtual memory to host virtual memory.

The linear mapping method, which reuses the host virtual memory, provides high performance for user-level DBTs. However, it introduces potential risks when discrepancies exist between the guest and host memory management. The difference in page size is the primary discrepancy between modern OSes, especially when the host page size exceeds the guest page size. Figure 1a illustrates a scenario that highlights security risks. It depicts four 4-KB private guest pages with different protection flags being linearly mapped to a 16-KB host page. The linear mapping renders four guest pages readable, writable, and executable, thereby introducing security risks such as overflow attacks. Figure 1b illustrates a scenario that highlights correctness risks arising when shared pages are used among multiple processes. DBT allocates a single 16-KB physical page to accommodate the shared 4-KB page. Consequently, the neighboring private pages are forced to be linearly mapped to the same 16-KB host physical page. This mapping causes the private pages can be overwritten by shared processes, resulting in correctness risks. Our survey reveals that most state-of-the-art user-level DBTs, including ExaGear [10,11], JIT Rosetta2[1], and user-level QEMU [5,18], suffers from the aforementioned risks, as shown in Table 1.

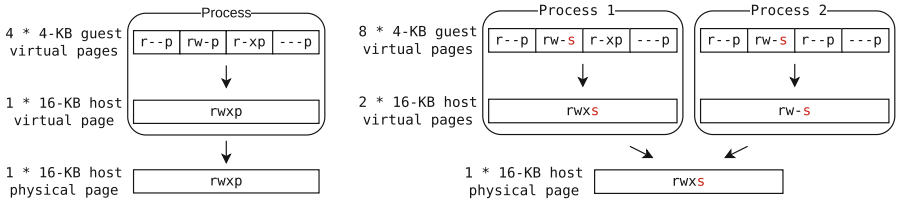
In contrast, system-level DBTs, such as system-level QEMU, do not encounter these risks due to the utilization of a software MMU. The software MMU fully emulates guest memory management, encompassing virtual-to-physical address translation and access permission checks. Despite its ability to mitigate the aforementioned security and correctness risks, the software MMU exhibits lower performance compared with the linear mapping approach.

On one hand, user-level DBTs utilize linear mapping, which provides high performance but entails potential risks. On the other hand, system-level DBTs employ a software MMU, which eliminates risks but exhibits lower performance. The distinctive characteristics of these two types of DBTs' memory management motivate us to propose an approach that combines their respective advantages.

[1] Rosetta has two versions: an Ahead-Of-Time (AOT) DBT for running X86_64 macOS applications on M-series silicon (AArch64) macOS [2], and a Just-In-Time (JIT) DBT for running X86_64 Linux applications on AArch64 Linux virtual machine [3]. Here we use the JIT version.

Table 1. Page size risks status of state-of-the-art DBTs. All of these DBTs target x86 or x86_64 Linux applications as guests. (QEMU refers to the user-level one.)

DBT	Proprietary	Host	Page size	Risks
ExaGear	Huawei	AArch64 Linux	64 KB	Existing
JIT Rosetta2	Apple	AArch64 Linux	16 KB	Existing
QEMU	–	Many ISAs Linux	8 KB/16 kB/64 KB/...	Existing
LATX (ODT-MMU)	Loongson	LoongArch Linux	16 KB	Mitigated
XQM (ODT-MMU)	Loongson	MIPS Linux	16 KB	Mitigated
QEMU (ODT-MMU)	–	Many ISAs Linux	8 KB/16 kB/64 KB/...	Mitigated



(a) Security risk caused by linearly mapping private pages. Guest permissions are inappropriately mapped to the host.

(b) Correctness risk caused by linearly mapping shared pages among processes. After the linear mapping, 12-KB data (3 * 4-KB pages) are lost from the initial 28-KB data (7 * 4-KB pages).

Fig. 1. DBT risks caused by linearly mapping small-size pages to large-size pages, for example, mapping 4-KB pages to 16-KB pages. Abbreviations: r readable, w writable, x executable, p private, s shared.

The new memory management we proposed, called on-demand triggered MMU (ODT-MMU), combines the linear mapping method with the triggering of the software MMU when risks arise. The detailed contributions of ODT-MMU are summarized as follows:

- ODT-MMU enables the utilization of linear mapping for non-risky page accesses and triggers software MMU when accessing risky pages. This approach effectively mitigates the risks related to page size and maintains the high performance of non-risky page accesses.
- To cater to various application scenarios, we implement ODT-MMU in two ways: ODT-InterpMMU, a platform-independent implementation that interprets the risky page accesses, and ODT-ManipTLB, which leverages the programmable TLB to enhance the risky page access performance.
- To the best of our knowledge, this work presents the first public analysis of the risky page accesses and the first applied solution in industrial DBTs: LATX [24] and XQM. This demonstrates the practicality of the proposed approach and showcases the effectiveness and efficiency of the ODT-MMU.

The rest of this paper is organized as follows: Sect. 2 provides a brief background and related work of DBTs' Memory Management and OS page

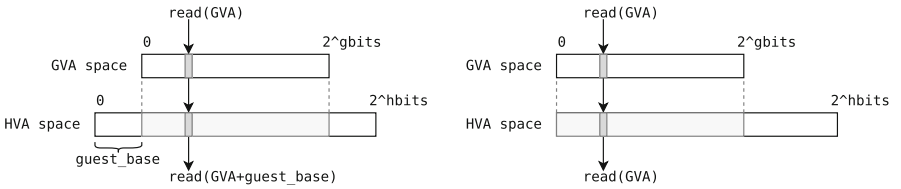
size. Section 3 introduces the design of ODT-MMU, including the related data structures, on-demand mechanism, and two implementations: software-based ODT-InterpMMU, and hardware-based ODT-ManipTLB. Section 4 evaluates our experimental results. The last section concludes this paper.

2 Background and Related Work

This section offers an overview of memory management in DBTs, including software MMU and linear mapping, and the diverse page sizes supported by hardware and OSes. Furthermore, this section presents related work in these areas.

2.1 Memory Management in DBTs

System-level DBTs that aim to achieve full OS translation must emulate the translation of guest virtual memory to guest host memory and the permission-checking mechanism. Typically, system-level DBTs employ a software MMU to emulate the guest memory management. The software MMU consists of a software TLB and a collection of page table look-up algorithms. A guest virtual memory access is translated into tens of host instructions if the software TLB hits, otherwise, hundreds of host instructions are needed to perform page table walk, software TLB refill, and eventually memory access. Consequently, memory emulation becomes a critical bottleneck in system-level DBTs, leading to extensive research efforts focused on improving memory emulation in system-level DBTs. Work [22] analyzes the memory emulation overhead in system-level QEMU and improves the software MMU performance inspired by optimizations applied to hardware TLB. ESPT [6] and HSPT [23] embed the guest page table into the host page table to leverage host hardware MMU. Captive [20] runs DBT in virtualization mode to facilitate the host hardware memory virtualization. Dual-TLB [28] and BTMMU [9] employ the host programmable TLB to accelerate memory access. All these software MMU improvements can be utilized to optimize our ODT-MMU. For demonstration, we implement the ODT-ManipTLB by utilizing the similar mechanism used by Dual-TLB and BTMMU.



(a) Linear mapping with guest base. One guest read is translated into one host add and one host read.

(b) Linear mapping without guest base. One guest read is translated into one host read.

Fig. 2. The linear mapping from Guest Virtual Address (GVA) to Host Virtual Address (HVA) in user-level DBT.

Unlike system-level DBTs, user-level DBTs focus on running user applications on a host machine. Hence, user-level DBTs are not responsible for emulating guest physical memory. Therefore user-level DBTs typically do not use software MMU to emulate guest memory management, instead, user-level DBTs reuse the host memory management through linearly mapping Guest Virtual Address (GVA) to Host Virtual Address (HVA), as illustrated in Fig. 2. Through linear mapping, one guest memory access is translated into two host instructions: one instruction adds an offset, called *guest base*, to GVA, and another instruction performs the memory access. High-performance user-level DBTs, such as ExaGear [10], Rosetta2 [3], and LATX [24] default to set guest base to zero, thus achieving one-to-one translation for memory access. User-level QEMU [5,18] defaults to a non-zero guest base but provides an option to set the guest base to zero. Furthermore, Bintrans [17] discusses the DBT security risks introduced by the page size and provides a basic solution by temporarily changing the memory permissions. Compared with our ODT-MMU, which utilizes software MMU and needs only one OS signal, Bintrans needs three OS signals.

2.2 Page Sizes

Contemporary ISAs universally support diverse page sizes within a page table. X86_64 [1] and AArch64 [4] utilize page table walking hardware to offer several fixed page size combinations, such as x86_64's 4 KB-2 MB-1 GB combination, and AArch64's 16 KB-2 MB-32 MB-1 GB combination. MIPS [14] and LoongArch [12] achieve arbitrary page size (which must be a power of two) combinations through the software-programmable TLB. For our ODT-MMU implementations, we use a 4 KB-16 KB combination. In addition to existing page size support in industrial products, extensive research is dedicated to multiple page size support in hardware. Subblock TLB [15,21] is proposed to achieve medium-sized pages (64 KB) with high performance, surpassing the traditional superpage TLB. Skewed TLB [16,19] is introduced to support concurrently multiple page sizes within a single process using a set-associative TLB.

Table 2. The default page size for various OSes (distros) on different ISAs.

OS (Distro)	ISA	Page Size (KB)
Linux	x86/x86_64	4
Linux	UltraSPARC	8
Linux (Loongnix)	MIPS/LoongArch	16
macOS/Linux (Asahi)	AArch64	16
Linux (CentOS)	AArch64	64

Due to the hardware's support for multiple page sizes, various OSes often employ distinct default page sizes. Table 2 presents the default page sizes employed by various OSes (distros). Particularly, in the area of personal computers, macOS, Asahi Linux, and Loongnix employ 16-KB page size by default,

which diverges from the traditional ISAs, like x86/x86_64 using 4-KB page size by default. In the area of servers, AArch64 CentOS employs a default page size of 64 KB, which also differs from the default page size of x86/x86_64. Moreover, there exist endeavors implementing multiple page sizes in OSes. In work [7], a multiple-page-size mechanism is implemented in IRIX OS utilizing TLB in R10000. In work [26], a similar multiple-page-size mechanism is implemented in x86 Linux OS. In work [27], a variable-page-size mechanism is implemented in MIPS Linux through variable-page-size TLB (VTLB). However, these studies primarily focus on improving performance by introducing multiple page sizes in OSes but disregard the security and correctness risks of executing small-page applications on a large-page OS, resulting in compatibility problems [13]. Furthermore, these studies typically involve modification of OS, which is not friendly to the compatibility problems, as it may introduce new compatibility problems.

3 On-Demand Triggered MMU

This section presents the design of ODT-MMU. We first introduce the related data structures and the on-demand mechanism. Then using the data structures and the on-demand mechanism as a foundation, we introduce ODT-InterpMMU, a software-based implementation that addresses the page size risks by interpreting the risky page accesses. Lastly, we introduce ODT-ManipTLB, a hardware-accelerated implementation that achieves high performance by utilizing the host's programmable TLB. Since our analysis focuses on user-level DBTs, without specifically referring to system-level DBTs, all the DBTs mentioned in the rest of this paper pertain to user-level DBTs.

3.1 Data Structures and On-Demand Mechanism

We leverage the software MMU inspired by system-level DBT to address the limitations of linear mapping. Therefore, a page table is added to user-level DBTs, called shadow page table, as shown in Fig. 3. The shadow page table only records the mappings from risky guest virtual pages to corresponding host virtual pages. The host virtual pages that are mapped in this manner are referred to as shadow pages. Additionally, a dedicated memory region, outside the linear mapped region, is allocated for these pages.

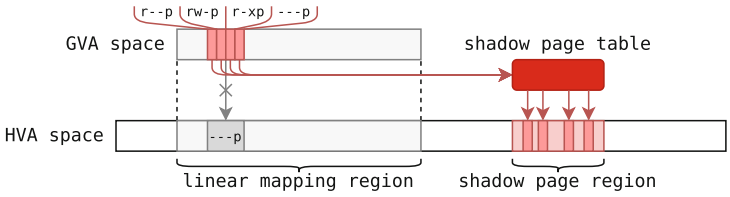


Fig. 3. Shadow page table maps risky pages to shadow page region. The linear mapping of risky pages is disabled.

Figure 4 illustrates the process of the on-demand mechanism. (1) During emulation of guest system calls related to virtual page management and permissions management, including `mmap`, `munmap`, `mprotect`, and `mremap`, (2) taking `mmap` as a specific example, if the guest tries to allocate a risky page (as shown in Fig. 1), a shadow page is allocated. (3) The linearly mapped host page is disabled by revoking its read, write, and execute permissions. (4) Consequently, if the guest attempts to access risky pages, it will trigger an OS signal due to the violation of page permissions. (5) The ODT-MMU can be invoked within the signal-handling function. Since the on-demand mechanism only modifies the permissions of risky pages, the performance of non-risky pages remains unaffected.

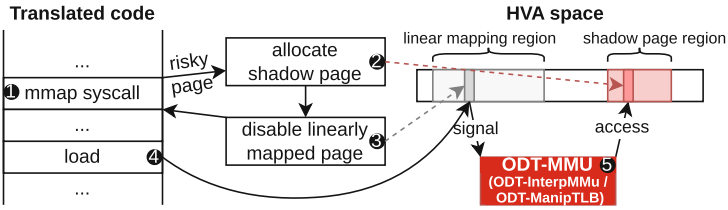


Fig. 4. The process of the on-demand mechanism.

3.2 ODT-InterpMMU: Interpreting the Risky Page Accesses

ODT-InterpMMU handles the risky memory access by interpreting it. During interpretation, the corresponding shadow page is retrieved from the shadow page table. The risky memory access is redirected to the corresponding shadow page. The outline of the ODT-InterpMMU code is depicted in Fig. 5. Within the signal handling function, the OS typically provides the Program Counter (PC), the accessed memory address (linearly mapped address), and the General Purpose Registers (GPRs). The interpretation is conducted based on the opcode of the instruction pointed at by the PC. For example, a load-byte instruction is interpreted as moving one byte from the shadow address to the destination GPR, and a store-byte instruction is interpreted as moving one byte from the destination GPR to the shadow address.

3.3 ODT-ManipTLB: Manipulating the Hardware TLB

ODT-ManipTLB leverages the host TLB for improved performance. Host ISAs like MIPS [14] and LoongArch [12] offer a VTLB that enables variable page size settings and programmability through software. The VTLB can be utilized to cache recently accessed shadow page table entries. As long as the VTLB hits, there is no overhead in accessing risky pages. Overhead only occurs when VTLB misses, and the ODT-ManipTLB is invoked as shown in Fig. 4. ODT-ManipTLB is responsible for obtaining the physical address of the shadow page and refilling the VTLB, as depicted in Fig. 6.

```
// Algorithm: InterpMMU(mc, lma, gprs)
// Input:
//   - mc: uint32_t (machine code)
//   - lma: uint64_t (linearly mapped address)
//   - gprs: GPRs array (General Purpose Registers)
// Output: None
uint64_t shadow_addr = shadow_page_table(lma - guest_base);
switch ( opcode(mc) )
case OPCODE_LOAD_BYTE:
    *gprs[dest(mc)] = *(uint8_t *)shadow_addr; break;
case OPCODE_STORE_BYTE:
    *(uint8_t *)shadow_addr = *gprs[dest(mc)]; break;
case ...
}
```

Fig. 5. The code of ODT-InterpMMU. The interpretation involves a switch-case statement based on the opcode of the instruction, which triggers the OS signal.

```
// Algorithm: ManipTLB(lma)
// Input: lma: uint64_t (linearly mapped address)
// Output: None
uint64_t shadow_addr = shadow_page_table(lma - guest_base);
// Following two funcs are implemented by kernel module
uint64_t physical_addr = get_physical_addr(shadow_addr);
refill_vtlb(lma, physical_addr);
```

Fig. 6. The code of ODT-ManipTLB. A dedicated kernel module is designed to get the physical address and refill the VTLB.

4 Evaluation

This section presents an evaluation of the experimental results of ODT-MMU. The experiments are conducted on Loongson's 3A4000 [8], which operates on Linux with a page size of 16 KB. The ODT-MMU is implemented in QEMU, targeting x86 Linux applications with a page size of 4 KB. The experiments include the following tests:

- Effectiveness tests include a collection of constructed unit tests and a real-world application - Wine [25], to evaluate the effectiveness of resolving the risks depicted in Fig. 1.
- Regression tests incorporate the industrial standard benchmark - SPEC CPU 2000, to ensure that non-risky memory accesses are not affected.
- Performance tests include a set of constructed read/write unit tests, to evaluate the performance of ODT-InterpMMU and ODT-ManipTLB.

4.1 Effectiveness Tests

Effectiveness tests aim to evaluate the effective mitigation of security and correctness risks. Unit tests are designed based on Fig. 1. To evaluate the security risks, multiple 4-KB pages with various permissions are allocated and read/written to determine whether the DBT on the 16-KB host raise segmentation fault when

the reads or writes are not permitted. To evaluate the correctness risks, multiple processes are created. Among these processes, multiple shared and private 4-KB pages are allocated with various permissions. These pages are then read and written to verify whether the DBT on the 16-KB host correctly writes private data and blocks the non-permitted reads or writes by a segmentation fault. Experimental results show the private data are correctly written to private pages and no overwrite occurs in ODT-MMU QEMU. Additionally, all non-permitted reads and writes are blocked, and the permission-related results are presented in Table 3. The original QEMU is incapable of addressing the security and correctness risks, whereas ODT-MMU QEMU mitigates these risks as expected.

Table 3. Effectiveness tests for original QEMU and ODT-MMU QEMU. All test cases are derived from Fig. 1. Abbreviations: Y permitted, N not permitted.

Issue Type	Permissions	Expected Results		Original QEMU		ODT-MMU QEMU	
		Read	Write	Read	Write	Read	Write
Security	r–p	Y	N	Y	Y	Y	N
Security	rw-p	Y	Y	Y	Y	Y	Y
Security	r-xp	Y	N	Y	Y	Y	N
Security	—p	N	N	Y	Y	N	N
Correctness	r–p	Y	N	Crash	Crash	Y	N
Correctness	rw-s	Y	Y	Crash	Crash	Y	Y
Correctness	r-xp	Y	N	Crash	Crash	Y	N
Correctness	—p	N	N	Crash	Crash	N	N

Furthermore, we conduct tests on a well-known multi-process application - Wine. A typical Wine program is associated with two processes: a `wineserver` responsible for emulating the Windows kernel, and an emulated Windows application. Shared pages are utilized to share data between these processes. Our experiments demonstrate that QEMU crashes when running Windows applications such as Notepad and Tencent WeChat, whereas ODT-MMU QEMU executes these applications smoothly.

4.2 Regression Tests

To evaluate whether ODT-MMU affects the performance of non-risky memory accesses, we begin by analyzing the memory accessing behavior of the SPEC CPU 2000 Integer test suite. Since all tests within CPU 2000 Integer are single-process, the presence of risky pages is solely attributed to security risks. Figure 7 presents the statistics regarding the risky memory pages, including the number of risky pages, as well as the ratio of memory accesses that read/write the risky pages to the total number of memory accesses. Several tests, such as `164.gzip` and `300.twolf`, exhibit more than 30% memory accesses being risky. The total count of risky pages does not exceed 20 for any of the tests. The findings of Fig. 7 indicate a high concentration of risky accesses on a few pages, making

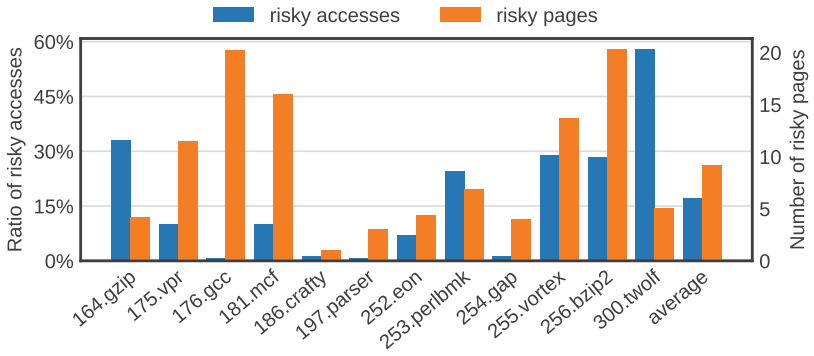


Fig. 7. The statistics of risky memory pages in SPEC CPU 2000 Integer. Left axis shows the ratio of the number of risky memory accesses to the number of all memory accesses. Right axis shows the number of risky pages.

them suitable for acceleration by VTLB, as VTLB typically incorporates 64 entries.

Subsequently, we evaluate the performance of the SPEC CPU 2000 Integer tests. The execution time of ODT-MMU QEMU (ODT-ManipTLB enabled) is normalized to that of the original QEMU, and the experimental results are illustrated in Fig. 8. The overall normalized performance hovers around 100%, suggesting that ODT-MMU has no impact on non-risky memory accesses. Conversely, when ODT-ManipTLB is enabled, several tests demonstrate a slight improvement in performance. This can be attributed to Linux's inefficient utilization of VTLB, and the enabling of VTLB in QEMU is tantamount to increasing the overall number of TLB entries, thereby slightly reducing the TLB miss rate and improving the TLB lookup performance.

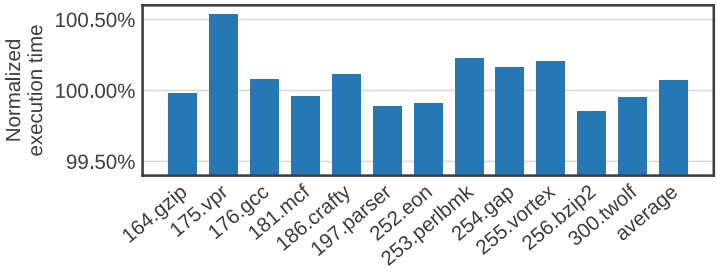


Fig. 8. The normalized execution time of ODT-MMU QEMU in SPEC CPU 2000 Integer. Normalization is achieved by dividing the execution time of ODT-MMU QEMU (ODT-ManipTLB enabled) by the execution time of the original QEMU.

4.3 Performance Tests

To evaluate the performance of risky memory accesses in ODT-MMU and original QEMU, we construct a series of read/write unit tests. Figure 9 demonstrates that ODT-InterpMMU is significantly slower than the original QEMU, which spends over 2000 ns to emulate one risky guest read/write operation. The low performance is mainly caused by the interpretation of software MMU and the trigger of OS signals for each risky read/write operation. Consistent with the findings of regression tests depicted in Fig. 8, ODT-ManipTLB exhibits a modest performance improvement compared with the original QEMU. This is because the utilization of VTLB equates to an increase in overall TLB entries, which results in fewer TLB misses and overall performance improvement.

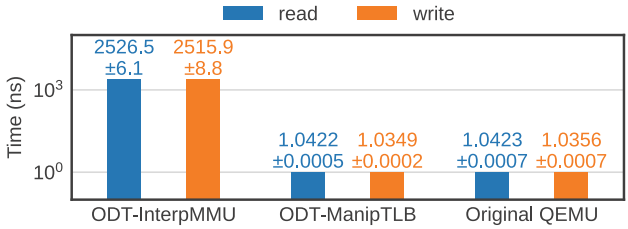


Fig. 9. The execution time (5 digits are reserved) per guest read/write for ODT-InterpMMU, ODT-ManipTLB, and original QEMU. ODT-ManipTLB QEMU shows slightly higher performance compared with the original QEMU.

5 Conclusion

This paper focuses on analyzing memory management in user-level DBT. Our analysis has identified security and correctness risks in linearly mapping memory management utilized by user-level DBT. These risks arise when executing small-page guest applications on a large-page host OS, as the guest page permissions cannot be appropriately mapped to the host page, resulting in risky access to these pages. The importance and urgency of these risks are increasing with the current transition trend from traditional small-page OSes, such as 4-KB x86 Linux, to large-page OSes, such as 16-KB LoongArch and 16-KB AArch64 Linux. To tackle these risks, we introduce ODT-MMU, a novel DBT mechanism capable of triggering software MMU on demand. ODT-MMU includes a platform-independent implementation called ODT-InterpMMU and a hardware-accelerated implementation called ODT-ManipTLB. Both implementations effectively mitigate security and correctness risks without affecting non-risky memory accesses. Compared with ODT-InterpMMU, ODT-ManipTLB achieves a significant performance improvement of over 2000x by utilizing Loongson's programmable VTLB. Compared with the original DBT, ODT-ManipTLB

does not incur noticeable performance loss. In addition to our implementations on Loongson's platform, ODT-MMU can be utilized to mitigate the security and correctness risks in other ISAs as well.

Acknowledgment. This project is funded by the 2022 National Key Research and Development Program "Security Protection Technology for Distribution Network Key Information Infrastructure" Project 3 Distribution Network Computing Equipment Security Enhancement Technology Research and Localization Development (Project No. 2022YFB3105103).

References

1. AMD: AMD64 Architecture Programmer's Manual Volume 2: System Programming (2020)
2. Apple: About the Rosetta translation environment (2021). https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment. Accessed 10 June 2023
3. Apple: Running intel binaries in Linux VMS with Rosetta (2022). https://developer.apple.com/documentation/virtualization/running_intel_binaries_in_linux_vms_with_rosetta. Accessed 10 June 2023
4. Arm: Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile (2021)
5. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track (2005)
6. Chang, C.R., Wu, J.J., Hsu, W.C., Liu, P., Yew, P.: Efficient memory virtualization for Cross-ISA system mode emulation. In: International Conference on Virtual Execution Environments (2014)
7. Ganapathy, N., Schimmel, C.: General purpose operating system support for multiple page sizes. In: USENIX Annual Technical Conference (1998)
8. Hu, W., Wang, J., Gao, X., Chen, Y., Liu, Q., Li, G.: Godson-3: a scalable multicore RISC processor with x86 emulation. IEEE Micro **29**, 17–29 (2009)
9. Huang, K., Zhang, F., Li, C., Niu, G., Wu, J., Liu, T.: BTMMU: an efficient and versatile cross-ISA memory virtualization. In: Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (2021)
10. Huawei: Huawei kunpeng exagear (2022). https://mirrors.huaweicloud.com/kunpeng/archive/ExaGear/. Accessed 10 June 2023
11. Huawei: Technical constraints-introduction-user guide-binary translator (ExaGear)-Kunpeng DevKit-Kunpeng documentation: technical constraints (2023). https://www.hikunpeng.com/document/detail/en/kunpengdevps/ug-exagear/usermanual/kunpengexagear_06_0005.html. Accessed 10 June 2023
12. Loongson Technology Corporation Limited: LoongArch Reference Manual - Volume 1: Basic Architecture (2023)
13. Marcan: Asahi Linux progress report: September 2021 (2021). Accessed 10 June 2023
14. MIPS Technologies Inc.: MIPS Architecture for Programmers Volume III: The MIPS64 and microMIPS64 Privileged Resource Architecture (2014)
15. Navarro, J.E., Iyer, S., Druschel, P., Cox, A.L.: Practical, transparent operating system support for superpages. In: USENIX Symposium on Operating Systems Design and Implementation (2002)

16. Papadopoulou, M.M., Tong, X., Seznec, A., Moshovos, A.: Prediction-based superpage-friendly TLB designs. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 210–222 (2015)
17. Probst, M.: Dynamic binary translation (2003)
18. QEMU: QEMU, a generic and open source machine & userspace emulator and virtualizer (2003). https://github.com/qemu/qemu. Accessed 10 June 2023
19. Seznec, A.: Concurrent support of multiple page sizes on a skewed associative TLB. IEEE Trans. Comput. **53**, 924–927 (2004)
20. Spink, T., Wagstaff, H., Franke, B.: Hardware-accelerated cross-architecture full-system virtualization. ACM Trans. Archit. Code Optim. (TACO) **13**, 1–25 (2016)
21. Talluri, M., Hill, M.D.: Surpassing the TLB performance of superpages with less operating system support. In: ASPLOS VI (1994)
22. Tong, X., Koju, T., Kawahito, M., Moshovos, A.: Optimizing memory translation emulation in full system emulators. ACM Trans. Archit. Code Optim. (TACO) **11**, 1–24 (2015)
23. Wang, Z., et al.: HSPT: practical implementation and efficient management of embedded shadow page tables for cross-ISA system virtual machines. In: Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (2015)
24. Weiwu, H., et al.: Loongson instruction set architecture technology. J. Comput. Res. Dev. **60**, 2–16 (2023)
25. WineHQ: Wine, a windows compatibility layer for POSIX-compliant operating systems (1993). https://www.winehq.org/. Accessed 10 June 2023
26. Winwood, S., Shuf, Y., Franke, H.: Multiple page size support in the Linux kernel (2002)
27. Zhang, X., Jiang, Y., Cong, M.: Performance improvement for multicore processors using variable page technologies. In: 2011 IEEE Sixth International Conference on Networking, Architecture, and Storage, pp. 230–235 (2011)
28. Zhenhua, W.: A dual-TLB method to accelerate the memory access of binary translation. Master's thesis, University of Chinese Academy of Sciences, Beijing, China (2015)