This is the accepted (long) version of the paper published in IEEE CLOUD 2021.
The published (short) version is available at https://doi.org/10.1109/CLOUD53861.2021.00063.

Exploiting Sub-page Write Protection for VM Live Migration

Yosuke Ozawa
The University of Tokyo
Tokyo, Japan
ozawa@os.ecc.u-tokyo.ac.jp

Takahiro Shinagawa
The University of Tokyo
Tokyo, Japan
shina@ecc.u-tokyo.ac.jp

***Abstract*—Virtual machine (VM) live migration is an essential feature for cloud vendors to manage their cloud infrastructures. Since VM live migration transfers a large amount of written memory to synchronize the memory between VMs, reducing the amount of memory transfer has a significant impact on the efficiency and success of the migration. Various software-based approaches have been proposed to reduce memory transfer, but they consume significant CPU and memory resources, degrading performance under heavy load. We propose to exploit a hardware-based sub-page write protection to reduce unnecessary memory transfers by performing fine-grained write detection in live VM migration. However, since write protection is different from write detection, applying it naively may incur a large overhead that outweighs its benefits. In this study, we identified the root cause of the large overhead in a naive implementation and introduced an optimization to avoid it. Performance experiments with our emulator-based pseudo-migration demonstrated that for many workloads, the sub-page write protection could reduce migration time as much as differential compression built in QEMU with less CPU and memory consumption, and for some workloads, only the sub-page write protection could complete the migration. We also show that the maximum CPU overhead can be reduced by 25.5 percentage points by our optimization.**

***Index Terms*—Virtualization; VM live migration; Sub-page write protection; Intel SPP**

I. Introduction

Virtual machine (VM) live migration is a technique for migrating a VM running on one host to another without stopping the VM. VM live migration is useful for load balancing, hardware maintenance, power management, and fault tolerance [1], and therefore widely used in cloud data centers. In VM live migration, the pre-copy method [2] is commonly used to synchronize the memory between the source and destination VMs. In this method, memory contents of the source VM are transferred to the destination VM while the source VM is running, and then newly written memory contents during the transfer are transferred again. This process is iterated until the amount of memory transfer in one iteration becomes sufficiently small. Therefore, reducing the amount of memory transfer is crucial for reducing downtime, migration time, network load, and energy consumption [3]. Moreover, the reduction rate in the amount of memory transfer per iteration determines the success or failure of the migration.

In order to detect the newly written memory area during the memory transfer in each iteration, nested paging is usually used. In nested paging, when the geust OS in the VM writes to a guest physical page, the CPU sets the dirty bit of the corresponding entry in the nested page table. Therefore, the virtual machine monitor (VMM) can detect the newly written guest physical pages by scanning the dirty bits in the nested page table. The write detection in nested paging is performed in hardware, and thus has very little software runtime overhead. However, since write detection is performed on a page-by-page basis, memory areas that have not actually been written may be transferred.

Many software-based approaches to write detection with finer granularity than a page have been proposed [4]–[9]. One approach is to calculate the difference from the pages that are already transferred in the previous iterations [4], [5]. Another approach is to compute the hash value of the memory in a unit smaller than the page [6], [7]. Page compression [8], [9] also has an effect similar to fine-grained write detection. However, these approaches require a certain amount of computation based on the page contents, which can have a significant impact on the VM performance due to the CPU and memory overhead, especially when the host is heavily loaded. Several methods to reduce the number of pages to be transferred have also been proposed [10]–[18], but they do not consider the effect of write detection at finer granularity than the page.

In this paper, we propose to exploit a hardware-based sub-page write protection feature for fine-grained write detection in VM live migration. A sub page is one of the multiple

Yosuke Ozawa and Takahiro Shinagawa. Exploiting Sub-page Write Protection for VM Live Migration. In Proceedings of the 2021 IEEE 14th International Conference on Cloud Computing (IEEE CLOUD 2021), Sep 2021.
https://doi.org/10.1109/CLOUD53861.2021.00063

fixed-length partitions of a single page; for example, a 4-KiB page could be divided into 32 pieces of 128-byte sub pages. The sub-page write protection is a feature added to nested paging that allows write protection to be set for each sub page. The advantages of the sub-page write detection based on the sub-page write protection are less CPU overhead due to no computation on memory contents, and less memory overhead since only a bit indicating that write protection is enabled needs to be held in memory for each specified sub page. However, the sub-page write protection is not a dirty bit function that records writes in the page table in the background, but a function that generates a page fault event whenever a write is made to a write-protected sub page. Therefore, it could incur a large page fault overhead when the frequency of sub-page write is very high.

In order to clarify these gains and losses, we observed the behavior of sub-page writes during VM live migration for various workloads, and analyzed the reduction in memory transfers and the overhead of increased page faults. The target CPU architecture for the measurement was Intel 64 with the sub-page write permissions feature (SPP) [19], which is included in the latest products that Intel has started shipping. At the time of writing, we were unable to obtain an actual CPU equipped with SPP. So, we used Bochs [20], an IA-32 emulator that can emulate SPP, with some bug fixes by us, as our experimental environment. We also implemented write detection using SPP in QEMU/KVM. Unfortunately, the emulator is significantly slower than the actual CPU, and therefore using the emulator alone will result in workloads during migration that are far from reality. Therefore, we devised a *pseudo-migration* that combines the emulator with an actual physical machine to reproduce the workload in an execution environment close to reality.

From our observational experiments, we confirmed that only a portion of a 4-KiB page is written on average for many workloads, and therefore, sub-page write protection is likely to significantly reduce the amount of memory transfer. We also identified that most of the additional page faults caused by using the sub-page write protection were due to a single repetitive instruction in the guest kernel. Therefore, in order to reduce such repetitive page faults, we designed an optimization that predicts the sub pages that will be accessed in the near future and removes write protection in advance when such an instruction is detected.

In order to quantify the effectiveness of our approach, we conducted performance experiments and demonstrated that the sub-page write protection was able to reduce the amount of memory transfer and achieve the same level of migration time as the differential compression method implemented in QEMU, called Xor Binary Zero Run Length Encoding (XBZRLE) [5], with lower CPU overhead and less memory consumption. Moreover, in the SPEC CPU 2017 benchmark experiments, we found that only the sub-page write protection could complete the VM live migration in a workload. In this workload, we also found that the CPU overhead during migration can be reduced by 25.5 percentage points (from 80.0% to 54.5%) with our optimization to reduce page faults.

The contributions of this paper are as follows.

- We have proposed a novel method to reduce the amount of memory transfer in VM live migration by applying a CPU's new feature, the sub-page write protection, to fine-grained write detection.
- We have devised a pseudo-migration that combines an emulator with a real machine to reproduce a realistic workload during migration without an actual CPU.
- We have identified that most of the additional page faults of using the sub-page write protection are caused by repetitive instructions, and have shown an optimization technique to avoid them.
- We have quantified the effectiveness of exploiting the sub-page write protection and shown that it could achieve the same reduction as the standard software-based method with less resource consumption.

The remainder of this paper is organized as follows. Section II explains the background of VM live migration and fine-grained write detection. Section III presents the results of our observation experiments. Section IV describes our design of sub-page write detection using sub-page write protection. Section V shows the implementation of sub-page write detection using Intel SPP. Section VI demonstrates the effectiveness of sub-page write protection in VM live migration with emulator-based experiments. Section VII introduces some related works and Section VIII summarizes this paper.

II. Background

In this section, we provide background knowledge on VM live migration. We will first explain how VM live migration works in Section II-A and then discuss the granularity of write detection and memory transfer in Section II-B.

A. VM live migration

There are three major methods for VM live migration: the pre-copy method [2], the post-copy method [21], and the hybrid method that combines both of them [22]. In this paper, we focus on the pre-copy method, which is the most commonly used one.

The pre-copy method is divided into two phases. First, in the warm-up phase, the memory copy from the source VM to the destination VM is performed iteratively so that the memory contents are almost synchronized. Then, in the stop-and-copy phase, the source VM is stopped, the last synchronization is performed, and finally the execution is resumed on the destination VM. By reducing the time taken for the stop-and-copy phase to a few hundred milliseconds, users can feel as if the VM is running continuously. Since memory transfer is the dominant cost in both phases, reducing the amount of memory transfer has a significant impact on the performance of VM live migration. Note that in cloud data centers, where live VM migration is performed, storage is shared between VMs and there is no need to transfer data.

The specific operations in each iteration in the warm-up phase are as follows. In the first iteration, the entire memory of the source VM is transferred to the destination VM. In the second and subsequent iterations, the memory area that has been written during the previous iteration is detected by the VMM, and only the contents of that memory area are transferred. By repeating this process, the amount of memory transfer gradually decreases, and when the amount of memory to be transferred falls below a pre-specified threshold, the phase shifts to the stop-and-copy.

We refer to the time from the start of the first iteration until the VM resumes at the destination as the *total migration time*. We also refer to the total amount of memory transfer during the warm-up and stop-and-copy phases as the *total transferred memory*.

B. The granularity of write detection

As mentioned above, the pre-copy method needs to detect the newly written memory area during memory transfer in each iteration and re-transfer the memory in that area in the next iteration. Since the location and frequency of writes vary depending on the application and workload, the VMM needs to properly and efficiently capture the newly written area to minimize the amount of memory transfer. Reducing the amount of memory transfer in each iteration leads to a reduction in the total transferred memory, which directly leads to the reduction in migration time and network load. In addition, if the amount of transferred memory does not decrease with each iteration, the amount of memory to be transferred in a iteration cannot fall below the threshold and the migration cannot be completed.

In the VM live migration mechanism currently in practical use, the granularity of the write detection is basically per page. This is because the write detection is usually implemented using dirty bits, which are an ancillary feature of the CPU's nested paging. However, the application does not necessarily write to the entire page. For example, even if only a few hundreds bytes of a 4-KiB page is written, the entire 4-KiB page must be transferred because the dirty bits alone cannot detect that actual written bytes. If there are many partial writes to a page, the amount of memory transfer in each iteration may be several hundred to several thousand times larger than the amount of memory actually written.

III. Observation

In order to confirm the effectiveness of sub-page write detection on memory transfer reduction in VM live migration, we measured the average number of sub pages that were written within a 4-KiB page. In this experiment, we calculated the number of instructions executed during the second iteration of VM live migration on a physical machine and measured the number of written sub pages by reproducing the execution on Bochs. The workloads we used is shown in Table I.

Figure 1 shows the result of observation. The error bars show the standard deviations. We found that only one sub page was written in approximately 1,300 4-KiB pages, and only two sub pages were written in approximately 700 4-KiB pages on average. Although the standard deviation is a bit rather large because the number varies relatively widely among workloads, the overall trend was that only a part of sub pages of the 4-KiB pages were written. Therefore, by detecting and transferring only such sub pages using sub-page write detection, the amount of memory to be transferred is expected to be significantly reduced.

On the other hand, there were approximately 16,000 cases

TABLE I
EXPERIMENTAL WORKLOAD

Name	Workload
idle	no specific task
kernel compile	Linux 5.5.7 compilation
redis+YCSB	Redis 5.0.3 [23] with six workloads selected from YCSB 0.17.0 [24], [25] (`workloada` to `workloadf`) [parameters] field size: 100 bytes number of fields: 10 record size: 1 KiB number of records: 256 K used memory: approximately 500 MiB
SPECCPU	13 workloads of SPEC CPU 2017 v1.0.1 (`500.perlbench_r`, `502.gcc_r`, `505.mcf_r`, `520.omnetpp_r`, `523.xalancbmk_r`, `531.deepsjeng_r`, `541.leela_r`, `557.xz_r`, `508.namd_r`, `511.povray_r`, `519.lbm_r`, `538.imagick_r`, `544.nab_r`)

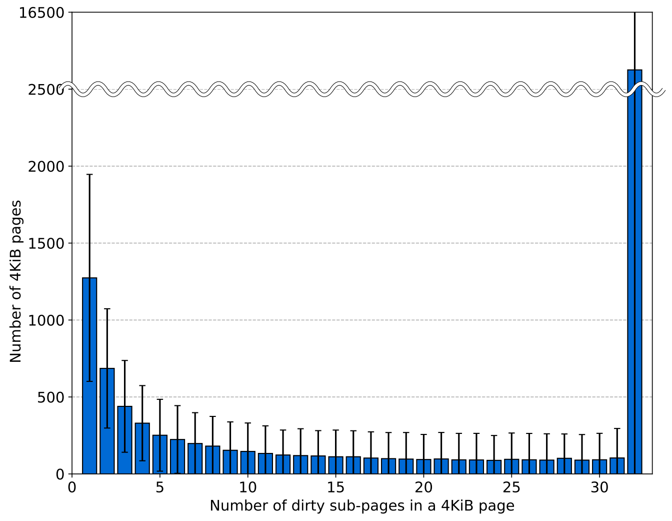


Fig. 1. Average number of sub-pages written in a 4-KiB page

Listing 1. The Linux kernel code that writes to a whole page

```
SYM_FUNC_START(clear_page_rep)
movl $4096/8,%ecx
xorl %eax,%eax
rep stosq
ret
SYM_FUNC_END(clear_page_rep)
```

where all sub pages in a 4-KiB page were written. We identified the locations where such writes were occurring and investigated what instructions were being executed. As a result, we found that a single location accounted for the highest percentage of instructions in all workloads except idle, which accounted for more than 90% in many cases. Listing 1 shows the disassembled version of the code for that location. This is a function in the Linux kernel called `clear_page_rep`, which clears an entire page by filling it with zeros. Specifically, this function assigns $4096/8 = 512$, the number of iterations, to the `%rcx` register with ”`movl $4096/8,%ecx`”, assigns zero to the 64-bit `%rax` register with ”`xorl %eax, %eax`”, and writes the value of `%rax` (zero) 512 times to the contiguous memory area with ”`rep stosq`”. The page address is implicitly specified in `%rdi`. Therefore, by detecting such codes and avoiding page faults caused by them, it is expected that the number of page faults can be reduced.

IV. Sub-page Write Detection

In this section, we describe our method of sub-page write detection for VM live migration base on the sub-page write protection. We first explain the hardware-based sub-page write detection mechanism in Section IV-A, and then describe the sub-page write detection system for VM live migration in Section IV-B.

A. The sub-page write protection

As described above, the sub-page write protection allows write protection to be set for each sub page. The write protection is usually represented by bits held in a data structure associated with the page table in nested paging. When the guest OS writes to a sub page with the write protection enabled, the CPU generates a page fault. We call this page fault a *sub-page fault*.

In normal paging, the page table has a dirty bit for each page in addition to write protection. However, since the sub-page write protection was introduced mainly for the purpose of write protection for hardware devices in virtualization, it does not have the function of dirty bits. Therefore, in order to perform write detection, we need to emulate dirty bits with write protection in cooperation with the VMM.

A sub-page fault requires a certain amount of CPU cycles for context switching between the VM and VMM. In addition, a sub-page fault handler needs to perform a number of operations, such as identifying the type of the sub-page fault, obtaining the address where the sub-page fault occurred,

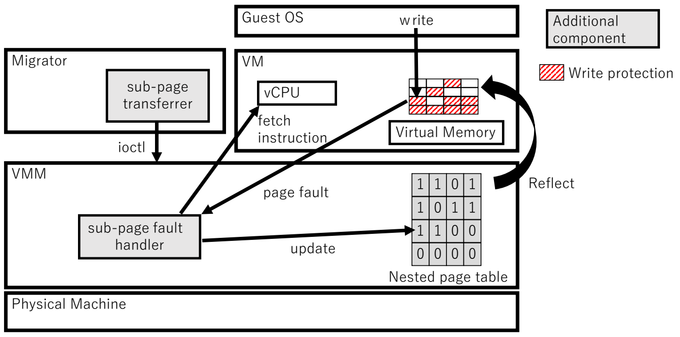


Fig. 2. The overview of our sub-page write detection system

and updating the write protection settings. Therefore, a large number of sub-page faults incur a large amount of overhead.

B. The sub-page write detection system

Figure 2 shows the overview of our sub-page write detection system using the sub-page write protection mechanism. The VMM manages the nested page table with the write protection settings for sub pages. At the beginning of each iteration of the VM live migration, the VMM enables write protection for all sub pages in the memory-allocated area of the guest physical address space of the source VM. When the guest OS in the source VM writes to a sub page in the guest physical address space for which write protection is enabled, the CPU generates a sub-page fault and passes the control to the VMM. In the sub-page fault handler, the VMM records the address of the sub page that is being written, then disables write protection for that sub page, and finally resumes the VM execution. By repeating this operation, the VMM can record the sub pages written in the VM in each iteration.

In parallel with recording the written sub pages in the sub-page fault handler, the VMM enumerates all the sub pages written in the previous iteration, except in the first iteration where it assumes that all sub pages have been written. Then, the sub-page transferrer in the migrator, which is our additional component of the VMM, acquires the contents of written sub pages via ioctl and transfers them to the destination. When the amount of transferring memory falls below a certain threshold, the system enters the stop-and-copy phase, where the final memory transfer takes place.

In order to reduce the runtime overhead caused by sub-page faults, we adopted two optimizations. First, in VM live migration, even if a sub page is written multiple times in each iteration, only the last written contents will be transferred. Therefore, instead of enabling write protection immediately after recording the address of the written sub page in the sub-page fault handler, we keep write protection disabled during that iteration and delayed enabling write protection until the start of the next iteration. In this way, we can reduce the overhead of multiple sub-page faults occurring on the same sub page in each iteration.

The second optimization is to predict the memory area where the next sub-page fault will occur by examining the instruction that caused the sub-page fault. For example, as shown in Section III, if an instruction repeatedly writes to consecutive memory areas, it is expected that the memory areas following the sub page that caused the sub-page fault will also be written. Therefore, instead of disabling write protection each time an individual sub-page fault occurs, we can disable write protection for consecutive sub pages at once in advance when the first sub-page fault occurs, thereby reducing the number of sub-page faults.

Currently, we only assume the optimization for memory clearing instructions, since the performance is satisfactory as we will show later, but we will be able to improve the performance by introducing more advanced predictions. Note that even if the prediction is wrong and write protection is disabled more than necessary, its only effect is that sub pages that have not been written will be considered written. Therefore, it will not affect the correctness of the migration, although it may slightly increase the amount of memory transfer.

V. IMPLEMENTATION

In this study, we used Intel 64 with SPP as the CPU architecture that provides sub-page write protection. For the CPU emulator, we used Bochs. For the VMM, we used a KVM based on a patch that adds SPP support [26] and implemented the sub-page write detection described in Section IV-B. We have also implemented some of the functionality required for sub-page migration in QEMU.

In this section, we first give a brief introduction to Intel SPP (Section V-A), and then explain the modifications we have made to Bochs (Section V-B), KVM (Section V-C), and QEMU (Section V-D), respectively.

A. *Intel SPP*

Intel SPP is an extension of the extended page table (EPT), which is the nested paging mechanism of Intel CPUs. It divides a regular 4-KiB page into thirty-two 128-byte sub pages, and write permission can be set for each sub page. To use SPP, we first need to enable the SPP function in the VM-execution control of the virtual machine control structure (VMCS), and then set the SPP bit to 1 in the leaf page table entry of the EPT. SPP consists of a structure similar to a page table; we need to set a pointer to the top page table, called the root SPP table (SSPL4), in VMCS.

If the SPP bit of the EPT entry is set to 1, the CPU performs a page walk of the SPP page table, starting from SSPL4, to identify the leaf table called the SPP vector table (SSPL1). SSPL1 consists of sixty-four 64-bit entries (SPP vectors), each of which corresponds to one 4-KiB guest physical page. Each 64-bit SPP vector consists of thirty-two pairs of 2 bits, and each 2 bits corresponds to one sub page. The upper bit of the two bits are the write permission bit. If the write permission bit is set to 0, when the corresponding sub page is written, an EPT violation (corresponding to a page fault) will occur. At this time, the software can determine whether the EPT fault occurred in the EPT or in the SPP by examining the SPP bit of the corresponding EPT entry.

B. *Modifications to Bochs*

Intel SPP is supposed to be implemented in Ice Lake, the 10th generation Intel Core processor. However, the Ice Lake generation CPUs currently available are only for mobile PC platforms, and we were unable to obtain a CPU with SPP implemented. Therefore, we used the Bochs emulator that implements SPP emulation. We used the Bochs version r13850. Unfortunately, the SPP emulation of this version of Bochs had the following bugs.

Bochs internally implements TLBs; it caches the entries in the EPT page table. According to the Intel Software Developers Manual [19], SPP vectors are also cached in the TLB (see Section 28.3.1 “Information That May Be Cached”). Despite this, Bochs did not cache SPP vectors. As a result, if one of the thirty-two sub pages of a guest physical page is given write permission, the entry of that guest physical page is cached in the TLB with write permission. Therefore, the entire page is considered to be allowed to be written, and the SPP write protection does not work. To fix this, we modified the TLB implementation of Bochs to also cache the SPP information correctly so that the write protection works correctly.

C. *Modifications to KVM*

We used KVM on Linux 5.5.7 with the SPP support patch applied as the base of our VMM that implements sub-page write detection. This patch allows the KVM to enable SPP, build SPP page tables, and handle EPT violation caused by SPP-related events when they occur. Based on this KVM, we implemented a function to fetch the instruction that caused the EPT violation in order to realize the optimization based on instructions, as described in Section IV-B.

D. *Modifications to QEMU*

We have implemented the sub-page write detection based on QEMU 4.2.50. In order to retrieve the contents of sub pages written by VMs from QEMU, we implemented a mechanism to retrieve and set the value of the write permission bit of a specified sub page and the contents of the sub page via ioctl in cooperation with KVM. In addition, since QEMU sends the physical addresses of the pages to be transferred together with the metadata when transferring memory, we added a new 1-byte value to the metadata to specify the physical addresses in sub-page units.

VI. EVALUATION

In order to quantify the effectiveness of the sub-page write detection based on the sub-page write protection in VM live migration, we compared the performance of the following three methods: (1) the conventional 4-KiB page-unit write detection method (“4-KiB page write detection”), (2) the method combining 4-KiB page write detection and XBZRLE

("4-KiB page write detection + XBZRLE"), and (3) the sub-page write detection method using Intel SPP ("128-byte sub-page write detection"), which is our proposal.

The experimental environment is as follows. We used a machine with an Intel Core i7-4790K CPU and 16-GiB memory. We ran a guest OS on QEMU/KVM on Bochs, that is, two VMs are nested. The VM that ran on Bochs (L1 VM) had one vCPU and 2 GiB of memory, and QEMU/KVM ran on the VM with Debian 10 (buster) and Linux 5.5.7 kernel with the SPP patch. The VM that ran on QEMU/KVM (L2 VM) consists of one vCPU and 1 GiB of memory, running Debian 10 (buster) and Linux 5.5.7. The network bandwidth for VM live migration was assumed to be 1 Gbps. The memory size used by XBZRLE for temporary storage of transferred pages was set to 512 MiB. In the following, the VM refers to the L2 VM unless otherwise specified.

In the remainder of this section, we first explain the pseudo migration method to mitigate the slow CPU emulation of Bochs in Section VI-A. We then show the results of three categories of experiments; First, Section VI-B shows the results of measuring the effect of memory transfer reduction on total migration time. Second, Section VI-C shows the results of comparing the CPU and memory overheads during memory transfers. Last, Section VI-D shows the results of measuring the runtime overhead in the sub-page write detection. The workloads used are those shown in Table I.

A. Pseudo Migration

We used Bochs for emulation. However, since Bochs is a full software-based CPU emulator, its performance is very slow compared to a real physical machine. Therefore, in order to estimate the performance on a real machine as accurately as possible while using the emulator, we devised a pseudo migration as shown in the following procedure.

1) Measure the performance of the physical machine: First of all, to estimate the number of instructions that can be executed per unit time on the physical machine, we measured instructions per second (IPS) on the physical machine using the `perf` command beforehand. Since IPS depends on the workload, we measured the IPS for each workload.

2) Measure the amount of memory transfer on the VM: Next, to estimate the time taken for each iteration, we measured the amount of memory transfers in each iteration on the VM. For the first iteration, we took the amount of all guest physical memory as the amount of memory transfer. In subsequent iterations, we calculated the number of bytes of pages (sub pages) written on the VM in the previous iteration. When XBZRLE was used, we took the size of the data after differential compression as the amount of memory transfer.

3) Execute instrucions on the VM: Then, we calculated the number of instructions that can be executed in each iteration on the VM as follows. First, to find the time required for memory transfer, we divided the amount of memory transfer obtained above by the network bandwidth. Second, to find the number of instructions per workload that can be executed on the VM, we multiplied the pre-measured IPS by the memory transfer time obtained above. Finally, we execute the number of instructions calculated above on the VM of Bochs.

4) Iterate the above steps: We iterated the above steps until the amount of memory transfer reached the size that could be transferred within 300 milliseconds. However, if the number of iterations exceeded 20, we determined that the migration could not be completed.

B. Total Migration Time

Since the effect of memory transfer reduction by using sub pages on VM live migration is closely related to the workload and the behavior of the application in each iteration, we measured its effect in terms of total migration time. Figure 3 shows the total migration time of the pseudo migration for each workload. The error bars indicate the standard errors. A bar higher than the wavy line indicates that the VM live migration did not complete. Since the total migration time was almost proportional to the total amount of memory transfer, the results show the effectiveness of memory transfer reduction in actual VM live migration.

The experimental results show that for most workloads, the "128-byte sub-page write detection" achieved almost the same reduction in migration time as the "4-KiB page write detection + XBZRLE". In addition, for the four SPECCPU workloads (`520.omnetdp_r`, `531.deepsjeng_r`, `557.xz_r`, and `519.lbm_r`), the "4-KiB page write detection" could not complete the migration, while the "4-KiB page write detection + XBZRLE" could complete two of them (`520.omnetdp_r`, `577.xz_r`), and the "128-byte sub-page write detection" could complete one more migration (`531.deepsjeng_r`).

As a typical example, we show in Figure 4 the transition in the amount of untransferred memory in each iteration of the workload `502.gcc_r`. In `502.gcc_r`, the first iteration took about 4.5 seconds and the next iteration took about 4.4 seconds. However, in the subsequent iterations, the "4-KiB page write detection" could not sufficiently reduce the amount of memory transfer, whereas "4-KiB page write detection + XBZRLE" and "128-byte sub-page write detection" were able to reduce the differences sufficiently to complete the migration in about 5 iterations.

We also found that "128-byte sub-page write detection" was able to reduce the migration time more than "4-KiB page write detection + XBZRLE" for some workloads. For example, for the workload `557.xz_r`, as shown in Figure 5, the difference in "128-byte sub-page write detection" was initially larger than "4-KiB page write detection + XBZRLE", but later reversed, resulting in a shorter migration time. This was probably because the contents written to memory were not easily compressed by XBZRLE.

Figure 6 shows the result in `531.deepsjeng_r`. In this workload, only "128-byte sub-page write detection" was able to complete the migration. As can be seen from the graph, the "4-KiB page write detection" was not able to reduce the amount of memory transfer even as the iterations progressed. Even with "4-KiB page write detection + XBZRLE", the amount of memory transfer did not decrease below around 192

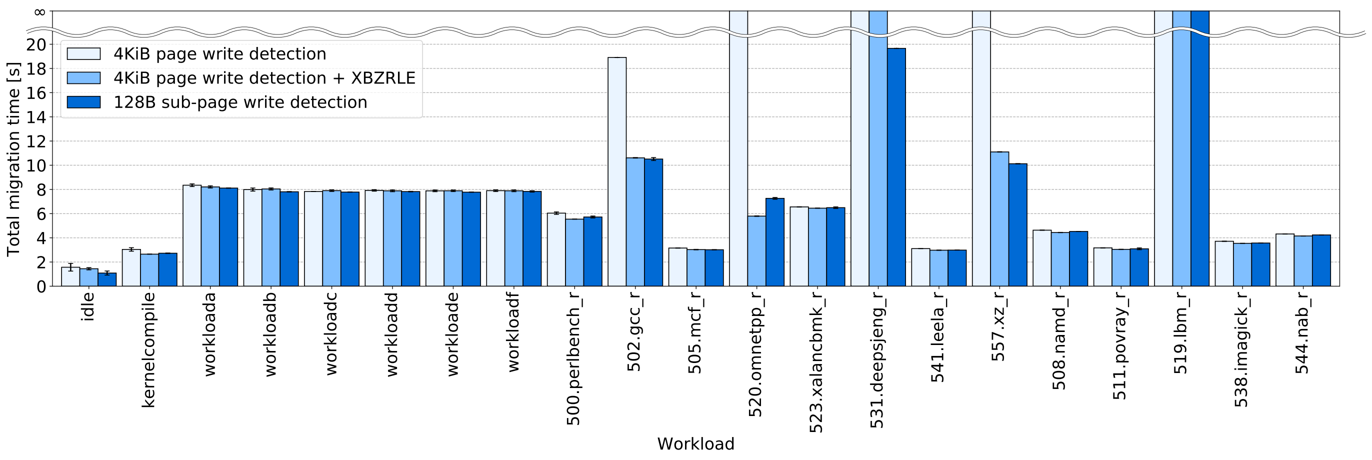


Fig. 3. Total migration time

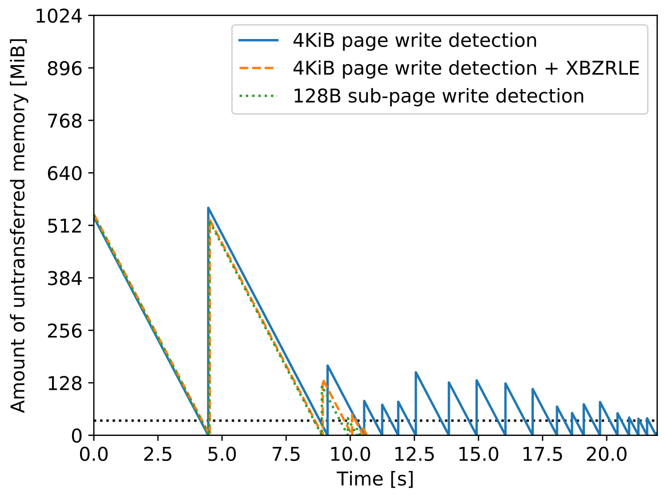


Fig. 4. Transition of the untransferred memory in `502.gcc_r`

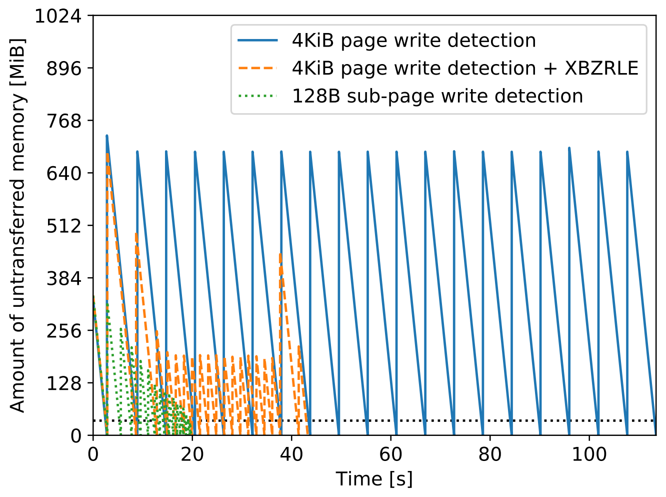


Fig. 6. Transition of the untransferred memory in `531.deepsjeng_r`

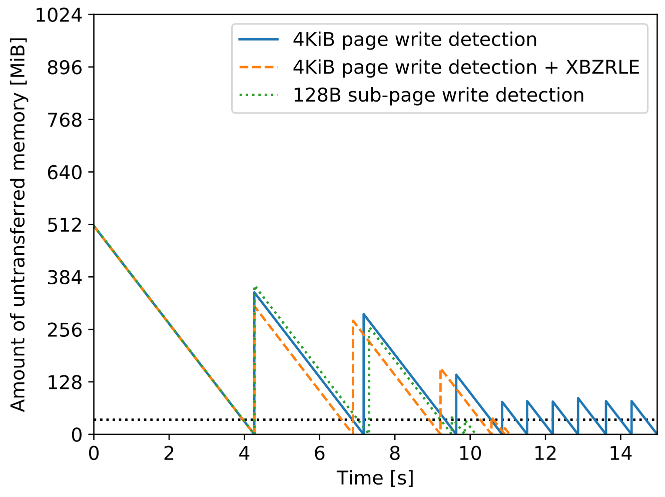


Fig. 5. Transition of the untransferred memory in `557.xz_r`

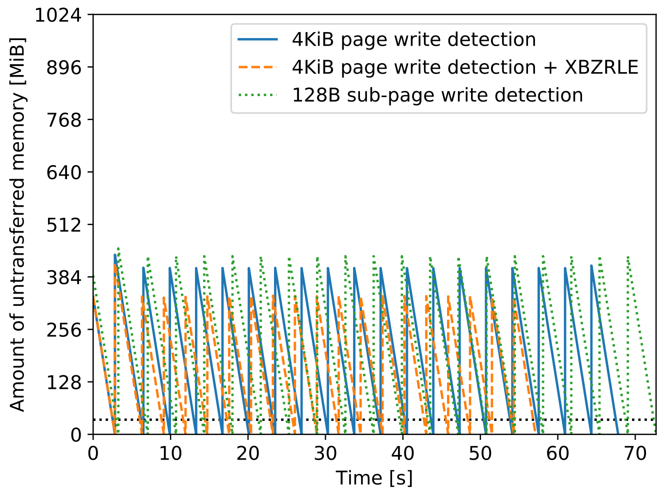


Fig. 7. Transition of the untransferred memory in `519.lbm_r`

MiB, indicating that the migration could not be completed in 20 iterations. On the other hand, the "128-byte sub-page write detection" was able to complete the migration in about 17 iterations. This was because `531.deepsjeng_r` wrote to memory very frequently, and the written values were difficult to compress differentially. In contrast, "128-byte sub-page write detection" steadily reduced the amount of memory transfer with each iteration, regardless of the value being written, and eventually completed the migration.

Unfortunately, there was a workload where the amount of transferred memory increased when using the "128-byte sub-page write detection". Figure 7 shows the result in `519.lbm_r`. From the graph, we can see that "128-byte sub-page write detection" slightly increased the amount of memory transfer, while "4-KiB page write detection + XBZRLE" slightly decreased the amount of memory transfer compared to "4-KiB page write detection". This was because this workload wrote to the entire 4 KiB page evenly, so there was no reduction in the amount of memory transfer by using sub pages, while metadata was added to each sub page during the transfer, increasing the amount of transferred memory. Having said that, none of the write detection methods were able to complete the migration of this workload.

In total, the "128-byte sub-page write detection" achieved the same or better migration time reduction than the "4-KiB page write detection + XBZRLE", although its effect varies depending on the workload.

C. *Overhead in memory transfer*

To estimate the overhead in memory transfer at each iteration, we measured the time to format the data to transfer memory, and the amount of memory consumed in the VMM.

1) CPU overhead: To estimate the CPU overhead in memory transfer, we measured the number of instructions required to format a single 4-KiB page in QEMU. In the "4-KiB page write detection", the number of instructions was measured by executing ioctl to get the information of written pages, determining whether the page is all zero, and formatting the entire page to the specified format. In the "4-KiB page write detection + XBZRLE", the number of instructions to store pages in temporary storage and to compress pages were also included. In the "128-byte sub-page write detection", it was similar to that of the "4-KiB page write detection", except that the VMM obtained sub pages instead of pages.

Figure 8 shows the results. The error bars indicate the standard errors. Compared to the "4-KiB page write detection", the number of instructions required for "4-KiB page write detection + XBZRLE" increased by a factor of 1.33 to 175.38. This was because XBZRLE requires a large number of instructions for compression including the byte-by-byte value comparison. The number of instructions for the "128-byte sub-page write detection" also increased by a factor of 1.22 to 15.84 due to the increase in per-page processing for handling sub pages, but the number of instructions is kept lower than in the "4-KiB page write detection + XBZRLE".

2) Memory overhead: To estimate the memory overhead in memory transfer, we measured the memory usage of the entire host machine (Bochs) at the start of the second iteration of the VM live migration using the `free` command. Figure 9 shows the results. The error bars indicate the standard errors.

The results show that the "4-KiB page write detection + XBZRLE" increased the memory usage of the host machine by 1.21 to 1.85 times compared to the "4-KiB page write detection". This is because we set the maximum amount of memory for temporary page storage used in XBZRLE's differential compression to 512 MiB, so the VMM consumed memory to store as much transferred memory as possible. On the other hand, the "128-byte sub-page write detection" hardly increased the memory usage.

D. *Runtime overhead*

To estimate the runtime overhead of the "128-byte sub-page write detection" due to sub-page faults and the effect of zero-clear instruction detection optimization, we calculated the number of instructions executed in the VMM by multiplying the number of SPP-related events during the pseudo migration by the number of instructions required for one event handling. Figure 10 shows the relative overhead compared to the case without SPP. The error bars indicate the standard error. In the idle workload, the overhead was as large as 200.74%, but this overhead was not a problem because the VM was doing nothing. For the other workloads, we found that the overhead of the workloads for which "4-KiB page write detection" was unable to complete the migration was relatively large. This means that these workloads have a very high write frequency. Interestingly, the sub-page write protection has the effect of slowing down only the write instructions, thereby reducing the write pace in this type of workload. As a result, the sub-page write protection improves the migration success rate not only through fine-grained write detection, but also through the ability to adjust the write speed.

As for the zero-clear instruction detection optimization, we found that the maximum overhead of simply detecting writes in units of 128-byte sub pages was 80.0%, while it could be reduced to 54.5% by the optimization. From this result, we can say that the zero-clear instruction detection optimization was effective. This overhead can be further reduced by improving the prediction capability, but it needs to be balanced with the effect of suppressing the write speed, which is our future work.

VII. Related Work

One approach for detecting writes at a finer granularity than the page is to calculate the difference from the pages that have already been transferred. Svärd et al. [4] proposed a method to temporarily store a part of the transferred pages as caches and reduce the amount of memory transfer with differential compression. In this method, XOR Binary RLE (XBRLE) is used as the differential compression algorithm, where run-length compression is performed after XORing between pages. Shribman et al. [5] proposed an improved version of XBRLE, Xor Binary Zero Run Length Encoding

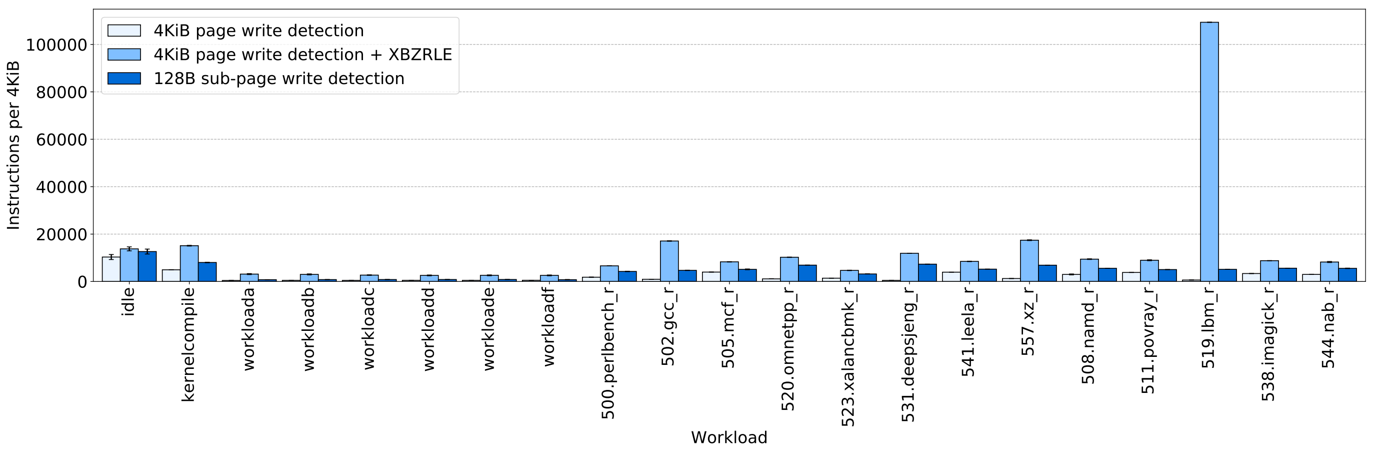


Fig. 8. The number of instructions to prepare a 4-KiB page in memory transfer

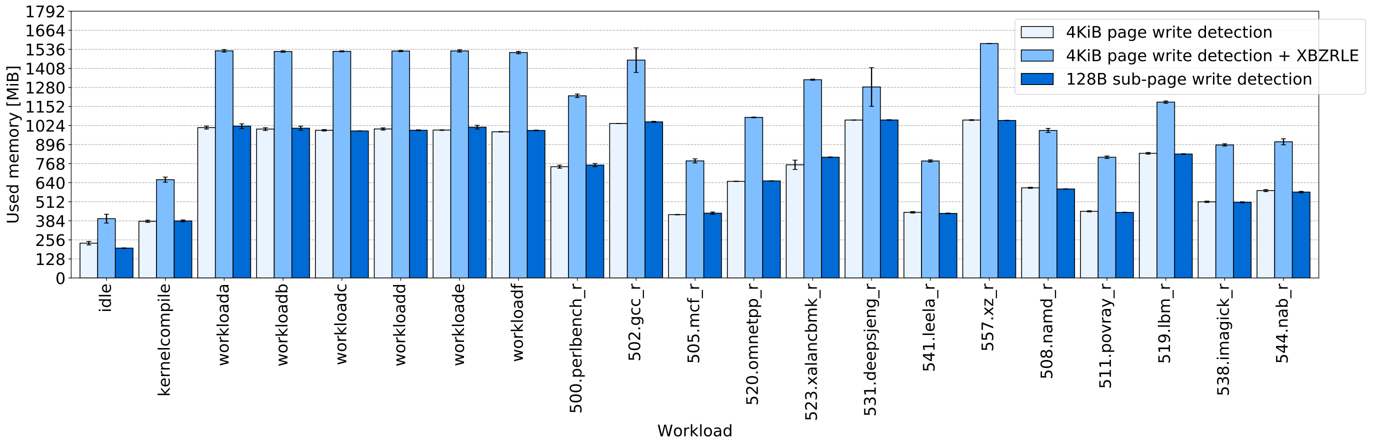
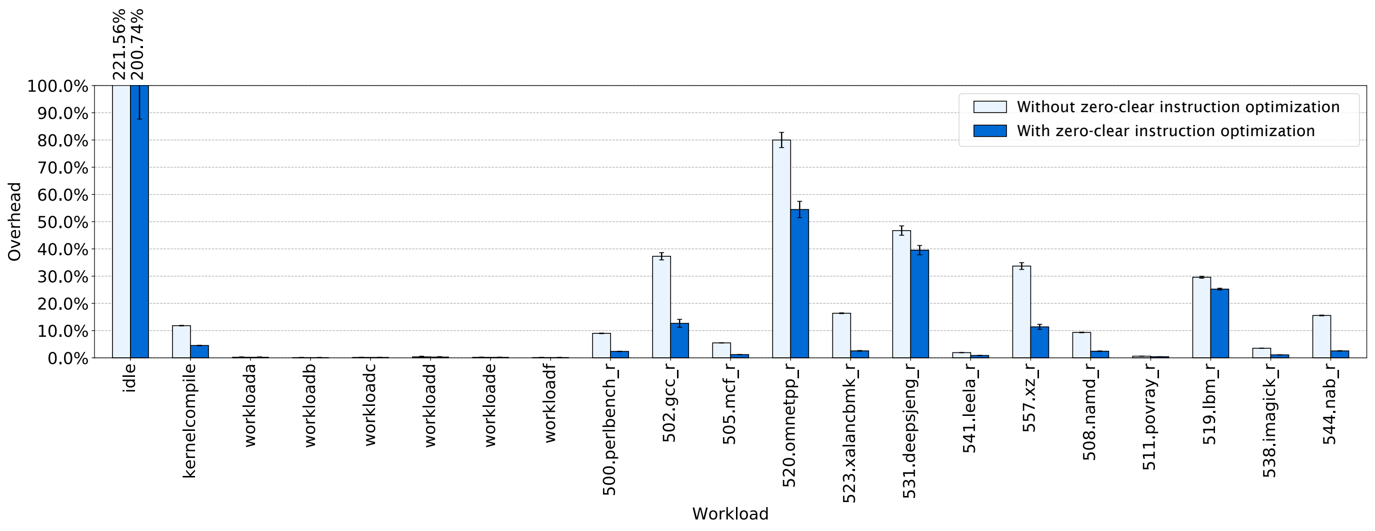


Fig. 9. The amount of memory consumed in the VMM in memory transfer



.

Fig. 10. The runtime overhead due to the SPP handling

(XBZRLE), which is used in QEMU. This algorithm improves compression efficiency by performing run-length compression only on the area that becomes zero after XORing between pages. However, since these differential compression methods need to save a part of the original pages, memory consumption increases as the number of written pages increases. In addition, since the compression ratio depends on the values in memory, there are cases where compression is not very effective even if the difference is small.

Another approach is to compute the hash value of the memory contents to detect the written area at a fine granularity. Wood et al. [6] used a method of dividing a page into four sub pages and calculating their hash values to reduce the amount of memory transfer for VM live migration in WANs. Deshpande et al. [7] proposed a method to compute the hash values of sub pages for deduplication among VMs when live-migrating multiple VMs simultaneously in a cluster environment. Li et al. [10] focus on write-not-dirty (write-not-modify), and propose a method to avoid the transfer of such fake-dirty pages that are dirty but whose values are not modified by hashing. Zhang et al. [27] introduced hash based fingerprints technology to identify identical and similar memory pages and utilizes RLE encoding algorithm to perform deduplication. The hash-based methods are expected to reduce memory transfer, but they consume more memory when the sub page size is reduced, and require byte-by-byte comparison to deal with hash collisions.

These software-based methods may be able to hide computation and memory overhead through parallel processing if the host machine has sufficient resources. However, in real-world use cases, the host machine does not always have sufficient resources. For example, Birke et al. [28] performed an analysis on a private cloud and found that more than 50% of the physical machines allocated almost all of their real memory to VMs. Therefore, if the host machine consumes more memory for VM live migration when the VMs are almost occupying the memory, it may lead to VM performance degradation.

Memory compression is an effective approach to reducing the amount of memory transfer memory in VM live migration. Jin et al. [8] proposed a method that selects an effective compression algorithm based on the content of the page to be transferred. Li et al. [9] proposed a method to dynamically change the compression ratio according to the network bandwidth. Memory compression using Zlib has also been implemented in QEMU/KVM. However, memory compression generally consumes more CPU time as the compression ratio is increased, and the compression ratio varies greatly depending on the contents and the compression algorithm. Memory compression can be used in conjunction with sub-page write detection. However, the compression ratio will decrease if the data size is too small. Therefore, it is necessary to devise a method such as compressing multiple sub pages at once.

Predicting memory write access patterns and skipping transfers of frequently written pages is another approach. Clark et al. [2] and Lin et al. [11] proposed an algorithm that excludes pages that are dirtied in both of two consecutive iterations from being transferred in subsequent iterations. Hu et al. [12] and Shi et al. [13] proposed a method to predict memory accesses in multiple iterations using a more complex algorithm. Alamdari [29] adopted the concept of reuse distance to reduce the transferring number of dirtied pages. These methods may also be applied to sub-page write detection.

Eliminating unnecessary regions of VM memory in advance is also effective in reducing the total amount of transferred memory. Hines et al. [14] proposed a method to eliminate unused memory dynamically by using memory ballooning. Ma et al. [15] proposed a method to obtain the actual allocated VM pages based on the guest OS information and transfer only those pages. Koto et al. [30] avoids transferring soft pages, such as free pages or file cache pages. These methods may be used in conjunction with sub-page write detection.

Google Compute Engine (GCE) live migration [31] combines the pre-copy and post-copy methods. It uses the pre-copy method for most cases, but switches to the post-copy when the VMs has a large dirty set and high write rate. This approach is also orthogonal to ours.

Another interesting approach is to indirectly reduce the amount of memory transfer by limiting the VM workload. Clark et al. [2] proposed a method to reduce the number of dirty pages by stopping the process that generates the most page faults on the guest OS. Jin et al. [16] proposed a method to reduce the execution time of vCPU, and Yiqiu et al. [17] proposed a method to add sleep tasks to the task queue of the guest OS. QEMU/KVM also implements a method to limit vCPU activity [18]. The sub-page write protection has the property that the number of sub-page faults increases as the frequency of memory writes increases. Therefore, it can be applied to automatic adjustment of memory write frequency.

Checkpoint and replay [32], [33] greatly reduces the amount of memory transfer with an approach that does not fully restore the VM state. In this approach, the VM state is transferred by checkpointing in advance, and then only the minimum amount of logs are sent to replay the VM behavior at the destination. However, the memory contents of the VM may not be completely reproduced, which may cause compatibility issues.

Distributed Shared Memory (DSM) [34] is a classical method to synchronize memory across multiple machines. Zhang et al. [35] proposed a DSM using write detection with EPT write permissions to realize VMs running distributed across multiple physical machines. Zekauskas et al. [36] proposed a fine-grained write detection by modifying the compiler and achieved up to 44% reduction in data transfer. Schoina et al. [37] implemented fine-grained write permission by modifying the memory interface and inserting code before memory write instructions. Some of these DSM techniques might be applied to VM live migration.

VIII. SUMMARY

In this study, we estimated the effectiveness of sub-page write detection using the sub-page write protection mechanism in VM live migration. The sub-page write protection is a function that can set write protection for each sub page, and it can realize write detection in units of sub pages with less memory consumption, while it incurs a page fault cost for each write access to the sub page with write protection.

We targeted the CPU architecture with Intel SPP that is the first one to support sub-page write protection, implemented sub-page write detection and pseudo-migration on QEMU/KVM, ran various workloads on Bochs that can emulate Intel SPP, and estimated the effect of reducing the total migration time and the overhead on CPU and memory.

The experimental results show that sub-page write detection with sub-page write protection achieves the same level of memory transfer reduction and total migration time reduction as the method combining 4-KiB page write detection and XBZRLE in QEMU, while only slightly increasing memory consumption. In addition, for the SPECCPU `531.deepsjeng_r`, only sub-page write protection was able to complete the migration. On the other hand, we found that there were some workloads for which even sub-page write detection could not complete the migration.

For future work, we need to evaluate the performance on a real machine equipped with a CPU that supports sub-page write protection. In order to reduce the overhead of page faults caused by SPP, we need to consider further optimizations such as avoiding page faults by predicting writes per sub page. Additionally, we need to consider evaluating and reducing the cost of maintaining data consistency for sub-page write protection across multiple CPU cores, which is also a problem for regular pages, but is expected to increase even more for sub pages. Finally, sub-page write protection may be used to adjust the write speed to handle workloads that could not be completed by conventional methods.

References

[1] T. Le, "A survey of live Virtual Machine migration techniques," *Computer Science Review*, vol. 38, pp. 1–17, Nov. 2020.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation*, ser. NSDI '05, 2005, pp. 273–286.

[3] S. Akiyama, T. Hirofuchi, and S. Honiden, "Evaluating impact of live migration on data center energy saving," in *Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science*, ser. CloudCom 2014, Dec. 2014, pp. 759–762.

[4] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth, "Evaluation of delta compression techniques for efficient live migration of large virtual machines," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '11, Mar. 2011, pp. 111–120.

[5] A. Shribman and B. Hudzia, "Pre-copy and post-copy vm live migration for memory intensive applications," in *Proceedings of the 18th International Conference on Parallel Processing Workshops*, ser. Euro-Par 2012, Aug. 2012, pp. 539–547.

[6] U. Deshpande, X. Wang, and K. Gopalan, "Live gang migration of virtual machines," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11, Jun. 2011, pp. 135–146.

[7] T. Wood, K. K. Ramakrishnan, P. Shenoy, J. Van Der Merwe, J. Hwang, G. Liu, and L. Chaufournier, "Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines," *IEEE/ACM Transactions on Networking*, vol. 23, no. 5, p. 1568–1583, Oct. 2015.

[8] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive memory compression," in *Proceedings of 2009 IEEE International Conference on Cluster Computing and Workshops*, ser. CLUSTER 2009, 31 Aug.–4 Sep. 2009, pp. 1–10.

[9] C. Li, D. Feng, Y. Hua, W. Xia, L. Qin, Y. Huang, and Y. Zhou, "Bac: Bandwidth-aware compression for efficient live migration of virtual machines," in *Proceedings of the 36th IEEE International Conference on Computer Communications*, ser. INFOCOM 2017, May 2017, pp. 1–9.

[10] C. Li, D. Feng, Y. Hua, and L. Qin, "Efficient live virtual machine migration for memory write-intensive workloads," *Future Generation Computer Systems*, vol. 95, pp. 126–139, Jun. 2019.

[11] C. Lin, Y. Huang, and Z. Jian, "A two-phase iterative pre-copy strategy for live migration of virtual machines," in *Proceedings of the 8th International Conference on Computing Technology and Information Management*, ser. ICCM 2012, Apr. 2012, pp. 29–34.

[12] B. Hu, Z. Lei, Y. Lei, D. Xu, and J. Li, "A time-series based precopy approach for live migration of virtual machines," in *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ser. ICPADS 2011, Dec. 2011, pp. 947–952.

[13] B. Shi and H. Shen, "Memory/disk operation aware lightweight vm live migration across data-centers with low performance impact," in *In Proceedings of the 38th IEEE Conference on Computer Communications*, ser. INFOCOM 2019, 29 Apr.–2 May 2019, pp. 334–342.

[14] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '09, Mar. 2009, p. 51–60.

[15] Y. Ma, H. Wang, J. Dong, Y. Li, and S. Cheng, "Me2: Efficient live migration of virtual machine with memory exploration and encoding," in *Proceedings of the 2012 IEEE International Conference on Cluster Computing*, ser. CLUSTER 2012, Sep. 2012, pp. 610–613.

[16] H. Jin, W. Gao, S. Wu, X. Shi, X. Wu, and F. Zhou, "Optimizing the live migration of virtual machine by CPU scheduling," *Journal of Network and Computer Applications*, vol. 34, no. 4, pp. 1088–1096, Jul. 2011.

[17] F. Yiqiu, Z. Chen, and G. Junwei, "Improvement on live migration of virtual machine by limiting the activity of CPU," in *Proceedings of the 2017 International Conference on Computer Systems, Electronics and Control*, ser. ICCSEC 2017, Dec. 2017, pp. 1420–1424.

[18] QEMU/KVM. Autoconverge Live Migration. [Online]. Available: https://wiki.qemu.org/Features/AutoconvergeLiveMigration

[19] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel, Nov. 2020. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[20] Bochs. [Online]. Available: http://bochs.sourceforge.net/

[21] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, p. 14–26, Jul. 2009.

[22] S. Sahni and V. Varma, "A hybrid approach to live migration of virtual machines," in *Proceedings of the 2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, Oct. 2012, pp. 1–5.

[23] Redis. [Online]. Available: https://redis.io/

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, Jun. 2010, pp. 143–154.

[25] YCSB. [Online]. Available: https://github.com/brianfrankcooper/YCSB

[26] QEMU/KVM. SPP-Patch. [Online]. Available: https://lore.kernel.org/kvm/20200516125507.5277-1-weijiang.yang@intel.com/

[27] X. Zhang, Z. Huo, J. Ma, and D. Meng, "Exploiting data deduplication to accelerate live virtual machine migration," in *2010 IEEE International Conference on Cluster Computing*, 2010, pp. 88–96.

[28] R. Birke, A. Podzimek, L. Y. Chen, and E. Smirni, "Virtualization in the private cloud: State of the practice," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 608–621, Sep. 2016.

[29] J. F. Alamdari and K. Zamanifar, "A reuse distance based precopy approach to improve live migration of virtual machines," in *2012 2nd IEEE International Conference on Parallel, Distributed and Grid Computing*, 2012, pp. 551–556.

[30] A. Koto, H. Yamada, K. Ohmura, and K. Kono, "Towards unobtrusive vm live migration for cloud computing platforms," in *Proceedings of the Asia-Pacific Workshop on Systems*, ser. APSYS '12. New York,

NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2349896.2349903

[31] A. Ruprecht, D. Jones, D. Shiraev, G. Harmon, M. Spivak, M. Krebs, M. Baker-Harvey, and T. Sanderson, "Vm live migration at scale," *SIGPLAN Not.*, vol. 53, no. 3, p. 45–56, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3296975.3186415

[32] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtual machine based on full system trace and replay," in *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 101–110. [Online]. Available: https://doi.org/10.1145/1551609.1551630

[33] T. Knauth and C. Fetzer, "Vecycle: Recycling vm checkpoints for faster migrations," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 210–221. [Online]. Available: https://doi.org/10.1145/2814576.2814731

[34] K. Li, "Ivy: A shared virtual memory system for parallel computing," in *Proceedings of the 1988 International Conference on Parallel Processing*, ser. ICPP 1988, 1988, pp. 94–101.

[35] J. Zhang, Z. Ding, Y. Chen, X. Jia, B. Yu, Z. Qi, and H. Guan, "GiantVM: A type-ii hypervisor implementing many-to-one virtualization," in *In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20, Mar. 2020, pp. 30–44.

[36] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad, "Software write detection for a distributed shared memory," in *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '94, Nov. 1994.

[37] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain access control for distributed shared memory," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI, Nov. 1994, pp. 297–306.