electronics

Article

CCoW: Optimizing Copy-on-Write Considering the Spatial Locality in Workloads

Minjong Ha and Sang-Hoon Kim *

Department of Artificial Intelligence, Ajou University, 206 Worldcup-ro, Suwon 16499, Korea; hamj1128@ajou.ac.kr
* Correspondence: sanghoonkim@ajou.ac.kr; Tel.: +82-31-219-3423

Abstract: Copy-on-Write (CoW) is one of the most essential memory management techniques enabling efficient page sharing between processes. Specifically, combined CoW with the fork system call, applications, even with a huge memory footprint, can take a snapshot of the current in-memory data at low overhead. However, since the CoW takes place per page in the page fault handler, each time the page fault occurs, the operating system should get involved. This leads to significant performance degradation for write-intensive workloads. This paper proposes coverage-based copy-on-write (CCoW), an optimized CoW scheme considering the locality in memory accesses to mitigate the problem of CoW. CCoW measures the spatial locality in process address spaces with the concept of coverage. While processing CoW, CCoW copies multiple pages in advance for high-locality memory regions, thereby minimizing the involvement of OS for write-intensive workloads. We explain the challenges for measuring the locality and provide the optimization to implement the concept. Evaluation with a prototype demonstrates that this approach can improve the overall performance of applications by up to 10% with a small amount of memory overhead.

Keywords: copy-on-write; virtual memory system; fork; Redis; page fault

check for updates

Citation: Ha, M.; Kim, S.-H. CCoW: Optimizing Copy-on-Write Considering the Spatial Locality in Workloads. *Electronics* **2022**, *11*, 461. https://doi.org/10.3390/electronics11030461

Academic Editor: Gianpaolo Vitale

Received: 23 December 2021
Accepted: 27 January 2022
Published: 3 February 2022

1. Introduction

The primary role of an operating system (OS) is to manage the precious system resources, and copy-on-write (CoW) is one of the most fundamental memory management techniques adopted by most of contemporary operating systems. When two or more processes need to have the same data, the CoW scheme allows processes to share the same pages rather than immediately duplicating the pages. The shared pages are duplicated on-demand, only when one of the processes writes to the shared pages. Virtual memory systems can effectively provide data sharing between processes, and various virtual memory features, such as duplicating the address space during process forks, deduplicating same pages, zero page sharing, are implemented based on the copy-on-write [1–3].

Owing to these features, applications can make a copy of themselves with low space overhead, and use the clone appropriately. For example, Redis, one of the popular in-memory key-value store systems [4], uses copy-on-write in conjunction with the fork system call for persisting in-memory data to the storage. While serving inbound requests, the Redis main process periodically spawns a child process with the fork. The child process begins with a memory snapshot identical to the main process to save the in-memory data in storage. As the snapshot is isolated from the address space of the main process but managed by the copy-on-write, the main and child processes do not require any complicated mechanism to maintain the consistency between the current data and the snapshot.

Thus, we can consider that copy-on-write is an essential in the virtual memory system. However, current copy-on-write is problematic in memory-intensive applications with write-intensive workloads. Specifically, the data duplication usually occurs in the page fault handler in the OS. Since the data duplication is processed *per page*, with several writes, the process can incur a considerable number of page faults. The OS is involved in each page fault, resulting in frequent user-kernel mode switches. Considering the huge memory footprint of memory-intensive applications, the number of mode switches is large. In addition, the page table is modified during the duplication, which leads to translation look-aside buffer (TLB) shootdown of all cores in the system. All of these incur non-negligible overheads and deteriorate the performance of the applications.

This paper proposes *coverage-based copy-on-write (CCoW)*, a novel copy-on-write optimization scheme. When a page is accessed for write, its nearby pages are also likely to be accessed soon for write due to the spatial locality in memory accesses. CCoW exploits the spatial locality to reduce the number of page faults for copy-on-write. Specifically, CCoW processes the copy-on-write in a large granularity (called a *region*). By copying multiple pages in the page fault handler, CCoW can reduce a considerable number of page faults for copy-on-writes and accompanying overheads. However, the degree of the spatial locality varies widely depending on the location in the process address space and duplicating the low-locality parts of memory incurs only the overheads in terms of time and space. To overcome this shortcoming, we propose a precise low-overhead mechanism to assess the spatial locality in the process address space. CCoW counts the number of copy-on-writes and writes in each region. By carrying the locality information over forks, we can estimate the degree of spatial locality, and CCoW effectively performs the precopy only for high-locality regions.

We implemented the proposed CCoW scheme in the Linux kernel. As integrated in the virtual memory system of operating system, applications can benefit from CCoW without a modification. We analyzed the performance characteristics with a microbenchmark, and evaluation using the benchmark with realistic workloads shows that CCoW can improve the application performance by up to 10% with a reasonable amount of memory overhead.

The rest of this paper is organized as follows. In Section 2, we overview the background and related work of the paper, including the virtual memory and fork. We explain the details of the CCoW design and its implementation in Section 3. Section 4 presents the evaluation results of the CCoW. Finally, we conclude this paper in Section 5.

2. Background and Related Work

2.1. Paging and Virtual Memory

Almost all modern computers and operating systems adopt *paging* and *virtual memory* as their primary memory management scheme [5]. The main memory is divided into same-sized pages, and OSs allocate or deallocate memory from user processes in the page unit. The OSs also maintain the mappings of the address spaces of processes to the physical location on the system. Each logical page in the process address space is mapped to its physical location, and this mapping is stored in the form of a *page table*. To handle memory reference for a process, a memory management unit (MMU), a hardware component in the processor, translates the requested address to its physical address by referring to the page table. The page table comprises page table entries (PTEs). Each PTE contains the mapping information and may have additional fields for describing the status of the corresponding page and mapping.

The pages size, although architecture-specific, is usually 4 KB in most architectures. This implies that each 4 KB in the process address space should have one PTE. Considering the huge size of process address space, the size of page table, even for a single process, can be enormous. For example, the page table for a process in 64-bit architecture with 4 KB page and 8-byte PTE would be 32 PB ($2^{64}/2^{12} \times 8 = 2^{55}$ bytes) in size. However, the address space is usually sparsely populated, and most of the address space is not required. This enlightens the hierarchical organization of the page table. The entire page table is divided into page table pieces that fit on a page. The page table pieces are not allocated to non-allocated address regions. The populated page table pieces can be summarized as higher-level page table pieces. This indirection is repeated until only pieces on one page exist, thereby allowing a compact form of page tables.

The small page size can be problematic as the systems become capable of handling a huge amount of physical memory. With the hierarchical page table organization, each virtual address translation requires multiple memory accesses, one for each page table level, which is unacceptable. To mitigate the high overhead of virtual to physical address translation, many modern architectures incorporate a cache for address translation. The MMU keeps a number of recent translation results in a hardware logic called a translation look-aside buffer, also known as TLB. Usually the TLBs of modern architectures can hold around 500 to 2000 entries [6,7]. The entries are indexed by hardware so that the processor core can look up the translation very quickly. By leveraging the locality of memory references, many address translations can be performed without walking through the page table (referred to as TLB hit). As the memory footprint for memory-intensive applications grows rapidly, the number of virtual to physical page mappings for a process also increases. However, due to the hardware limitations, the number of TLB entries cannot keep up with the rapid growth of application memory footprints. Thus, the TLB miss rates increase, causing bottlenecks in the performance of memory-intensive applications [8–11]. To overcome this limitation, some architectures support additional page sizes larger than the size of 4 KB base pages. For example, modern Intel architectures support 2 MB and 1 GB page sizes [7]. With such a huge page size, one address translation can cover a wider address range, effectively increasing the coverage the TLB can provide with the same number of entries. For an instance, a system with 1024 TLB entries and 4 KB base page size can provide TLB coverage of 4 MB, whereas the same number of entries with 1 GB huge pages provides 1 TB coverage.

Linux utilizes the huge page in the form of transparent huge pages (THPs). As the name suggests, Linux implicitly provides user processes with huge pages whenever possible. If THP is not enabled, Linux allocates memory to processes in the 4 KB base page unit. If THP is enabled, Linux attempts to allocate a huge page (2 MB in size) instead of the base page, allowing a coarse-grained page mapping. This large granularity allows for efficient page sharing between parent and children processes through the fork. In case huge page allocation is not feasible at the moment, Linux falls back to the base page allocation. Linux periodically scans process address spaces to find base pages and consolidate them into huge pages.

There has been studies attempting to promote huge pages for performance while mask their shortcomings further. Ingens [12,13] proposes to prepare huge pages asynchronously off the critical path. Hawkeye [14] presents fine-grained huge page promotion scheme based on memory access patterns to maximize performance with a minimal number of huge page promotion. Zhu et al. [15] generalize the processes of using huge pages, and optimize the lifecycle of huge pages. Part et al. [16] allow holes in huge page, providing the flexibility in memory management with huge pages.

The huge page, however, is a double-blade sword. Due to the increased management unit size, page allocation suffers from internal fragmentation. If an allocated address range is smaller than the huge page size, the rest of the page cannot be utilized and gets wasted. This so-called *memory bloat* can significantly decrease memory utilization on the systems with huge pages [12–17]. The increased page size can negatively affect program performance as well. Modern OSs adopt the copy-on-write scheme extensively for efficient memory sharing between processes. The CoW is, however, processed only at the base page granularity. Thus, to handle CoW on a huge page, the huge page is split into base pages, and only the faulty page is copied. Breaking huge pages takes a considerable amount of time, resulting in intermittent long page fault handling. In this sense, some applications, even memory-intensive ones, do not recommend using huge pages for stable performance and memory utilization [4,18].

In general, there are ranges of address space in the process address space where all the pages in the range have the same permission and characteristics. For management, modern OSs usually adopt the concept of 'virtual memory area (VMA)' to represent such ranges of address space. We can classify the pages in the process address space according to their origin. Some pages can be loaded from a backing file on the secondary storage, referred to as 'file-backed pages'. Whereas, some pages are dynamically populated without any backing data. The pages for stack and heap are in this case, the so-called 'anonymous pages'.

2.2. Fork and Copy-on-Write

Fork is one of the POSIX standard system calls to create a new process. When a process invokes the fork system call, a new process is created as the child of the calling process. Under the hood, the OS creates the child process by *duplicating the entire address space* of the calling process. This implies that the child process should start with exactly the same data as the parent process. To handle the address space duplication efficiently, most of modern OSs use the *copy-on-write (CoW)* technique. To duplicate the address space of the parent, the OS does not actually copy each page. Instead, the page table of the child process is constructed by copying the page table of the parent process. This effectively makes a shared mapping to the address space of the parent. While making the shared mapping, the write permission for each page is dropped by clearing the permission bit in the corresponding PTE. After copying the mapping, both parent and child can read the shared pages as their own pages. When one of the processes makes a write access to a page, the MMU, due to the lack of write permission, triggers a page fault. In the page fault handler, the OS allocates a new page, copies the original page, and updates the corresponding page mapping of the fault-causing process with write permission. At this point, the parent and child can have different data on the same virtual address.

This copy-on-write mechanism is extensively used as the fundamental key mechanism for realizing many virtual memory features. Specifically, reads of non-initialized heap regions are usually handled with shared mapping to a zero page, which is a special page containing all zeros. Kernel same-page merging (KSM) is the technique of deduplicating same pages in the system. The OS scans the pages in the system to identify pages with identical data. When such pages are found, the OS reclaims all but one page and updates the corresponding page tables to share the remaining page. In the processing, the write permission is dropped so that subsequent write access to the page is identified and copied.

With the high efficiency of copy-on-write, process creation becomes efficient, and some data-intensive applications leverage this advantage to create a data copy. The Redis, one of the popular in-memory key-value store services [4], is one such case [19]. The Redis is designed to primarily keep the data in memory to provide high throughput and low latency. However, some applications demand the persistence of stored data, and Redis complements the in-memory design with fork. The Redis applies inbound requests to the in-memory index and data structures only, and periodically invokes the fork system call. This effectively creates a child process with duplicated memory contents of the original Redis process, and the calling process (i.e., the original process) continues processing inbound requests. The child process diverts its execution; using the current memory contents as a snapshot, it serializes in-memory data structures into files, thereby ensuring the persistence of the in-memory snapshot. After flushing the snapshot, the child process terminates. The original process can make another snapshots in the same way, and upon a system crash, Redis can be recovered by reading the last snapshot.

Although the fork is an invaluable system call, its overhead has been criticized. Baumann et al. [20] analyzed the fork and found that fork causes the performance degradation in modern applications. For example, as the modern applications become more complex, the OS should consider approximately 25 special cases to start processing the fork system call to conform to the POSIX specification. They summarized the problems of the fork system call and suggest the features that the fork system call should have for the modern computer. They also provide alternatives ways of replacing the fork. Zhao et al. [19] pointed out that the fork implementation in current systems is inefficient since applications with a large memory footprint require a long time to set up the page table. As a solution, they generalized the copy-on-write technique so that the page table is copied on writes as well as regular pages.

3. CCoW Design

In this section, we first introduce our motivation behind improving the copy-on-write, and explain the basic concept of the coverage-based copy-on-write (CCoW). Then we explain the way CCoW captures the locality under different scenarios and the optimization to capture the locality at a low overhead.

3.1. Motivation

As discussed earlier, the copy-on-write mechanism plays a key role in implementing virtual memory features in modern OSs. However, its advantages in terms of space have been diminishing in the modern computing environments and write intensive workloads, which are common in data centers [21,22]. Emerging memory technologies such as storage-class memory (SCM) and persistent memory enable increased data density for memory modules while lowering the cost per unit data. Nowadays building a node with a huge amount of memory in the terabyte scale has become cheaper than ever. In addition, cloud service providers have reported that the nodes in data centers are suffering from low memory utilization, leaving 40–50% of memory unused [23–26]. In this situation, it becomes feasible to trade memory space for performance in performance-critical systems [27].

The advantages in terms of performance have been diminishing as well. The performance benefit of copy-on-write can be characterized by the frequency and performance of page fault handling. While spawning a child process, the write permission to all pages is dropped. From the perspective of correctness, this is inevitable; however, it leads to frequent page faults after the fork, in serving each write request. This storm of write page faults not only happen to child processes but also to the parent process. To make worse, the page fault handling time is not improved recently but tends to be prolonged due to security reasons. In the past, the entire kernel address space was persistently mapped to a part of the user process address space. However, this address space layout allows malicious user processes to indirectly read the critical data in the kernel address space by exploiting the speculative execution in the processors [28,29]. To mitigate such critical security vulnerability, modern OSs employ kernel page table isolation (KPTI). In general, only a limited portion of the kernel address space is mapped to the process address space, and the rest of kernel address space is dynamically mapped and unmapped during the interrupt and system call handling. This must be accompanied by TLB flushing, which can significantly degrade the system performance.

In this work, we aim to reduce the overhead of copy-on-write by leveraging the spatial locality of memory references. Currently, the copy-on-write takes place per-page, and each time the page fault occurs, the OS should get involved. Our key idea is to reduce the frequency of OS involvement by leveraging the spatial locality of memory accesses. If a page is accessed for write, nearby pages are also likely to be accessed for write in the near future. Thus, if we perform the copy-on-write not only for the faulty page but also nearby pages together (i.e., precopy nearby pages), we can amortize the overhead for the copy-on-write during the page fault handling.

We, however, should be careful, not blindly always copy all nearby pages. If the precopied pages are actually written later, the overhead incurred for the precopy is paid back. However, if the precopied pages are not written afterwards, the precopy only incurs extra overhead in terms of time and space. Thus, it is crucial to identify the proper target pages to precopy.

Similar approaches have been employed to minimize the page fault handling overhead. Linux employs the so-called ‘fault around’ feature. While handing a page fault, Linux initiates the page fault handling for the pages that are around the faulty page [2]. This feature, however, is only applied to the read page faults for file-backed memory regions. Given that the proposed idea focuses on write page faults for anonymous pages, we can argue that our approach is different from the fault around feature.

Many state-of-the-art designs [12–17] have been proposed to optimize the use of huge pages in the OS. These systems, in common, present a scheme to identify the best candidate pages to be converted to huge pages and to efficiently promote to (i.e., convert base pages to a huge page) or demote from (i.e., converts a huge page into base pages) huge pages. However, regardless of the proposed schemes, copy-on-write is performed in the base page granularity only, after breaking the huge page into base pages if necessary. Thus, their copy-on-write performance characteristics are the same to the default Linux system with the transparent huge page (THP) mechanism. In contrast, our proposed scheme is unique in that it performs copy-on-write at a different granularity according to the locality degree in memory accesses.

3.2. Identifying the Spatial Locality

To realize the proposed scheme, we should take two challenging issues into consideration. Firstly, target pages should be identified precisely and timely, so that the benefit of the precopy is maximized while the overhead for the precopy is minimized. Once a page is copied by a write, the page will not trigger any further page fault. This effectively means that the system lost the opportunity to optimize the write access. Thus, the system should be able to *foresee* future pages usage to determine which pages should be precopied and which are not. Secondly, identifying the target pages should have low-overhead since OSs cannot afford time-consuming processing in the performance-critical memory management subsystem. As discussed in Section 2, many virtual memory features in modern OSs are based on the copy-on-write mechanism. Thus, the overhead can easily outweigh the benefit of the optimized copy-on-write if the overall implementation is not sufficiently efficient.

To predict the future of a page, we first collect the history of forks for user processes. Specifically, the OS monitors the number of forks that each process invokes. A low count for a process implies that there is little opportunity to exploit the process, and the OS does not need to fully track the write page faults for this process. In contrast, when a process invokes the fork system call more than a threshold, the system can expect the optimization opportunity. This happens with Redis, which periodically invokes forks to take in-memory snapshots or with the shell script that forks multiple command-line commands. In response, the system starts to track the page faults for the process.

Next, we propose a method to predict the optimization opportunity from the history, assuming that the overall behavior of applications does not change significantly. To this end, we divide the process address space into fixed-sized *regions*. Each region maintains a bitmap, where each bit corresponds to a page in the region. A process is spawned with all bitmaps cleared, as for newly populated virtual memory areas (VMAs). When a part or entire VMA is unmapped, the bitmaps in the corresponding address range are also released. The bitmap is only allocated for the parts of VMA that are populated, and one 4 KB page information is summarized into one bit. Thus, the space overhead for the bitmaps is approximately 0.003% of the populated address space.

Initially, writes are processed through the copy-on-write as is. A write access is trapped to the page fault handler, whereby the corresponding bitmap entry is set. Over time, the bitmap effectively captures the accesses to the region, and we can quantify the degree with the *coverage*. The coverage of a region is calculated as the percentage of copy-on-written pages out of all pages in the region, as follows:

$$Coverage\ (\%) = (nr\_CoW\_pages / nr\_pages\_per\_region) \times 100 \tag{1}$$

Suppose a system uses 4 KB pages and the address space is divided into 2 MB regions. Then each region has 512 4 KB pages. If 300 pages are copied on writes, the coverage of the region is $300/512 \times 100 = 58.6\%$.

The higher the coverage of a region, the more the region can benefit from the optimization opportunity of the precopy. This coverage information is carried over the fork, and used as a metric to determine the benefit of precopying nearby pages. Specifically, if a memory region has a high coverage, the pages in the region are likely to be copy-on-written eventually. Thus, it would be beneficial to precopy other pages in the region while processing a write page fault for a page. Whereas, optimization opportunity in precopying pages is little if the coverage is low. Thus, only the faulty page is copy-on-written by the page fault handler. Henceforth, we will refer to this scheme as *coverage-based copy-on-write* or *CCoW* to the rest of the paper.

Figure 1 illustrates the concept of CCoW. There are two regions, regions 0 and 1, each comprising six pages. The pages with green shade are populated with the copy-on-write. When the process invokes the fork system call, write permission for all pages, including the green ones, are dropped during the fork. Let us assume that the threshold for determining the high-locality region is 60%. In region 0, four pages (page 1, 2, 3, and 4) had been copied on write before the fork, providing a coverage of $4/6 = 66\%$. Thus, this region is considered to have high spatial locality, and all pages are copied to handle the page fault for page 3, as shaded in red. Whereas, the lower region provides a coverage of 33% as two out of six pages had been copy-on-written before the fork. Therefore, this region has a lower coverage than the threshold, implying the low spatial locality in the region. Thus, when the page 9 is accessed for write, only the faulty page is copied on write in the page fault handler, just like the normal copy-on-write procedure.

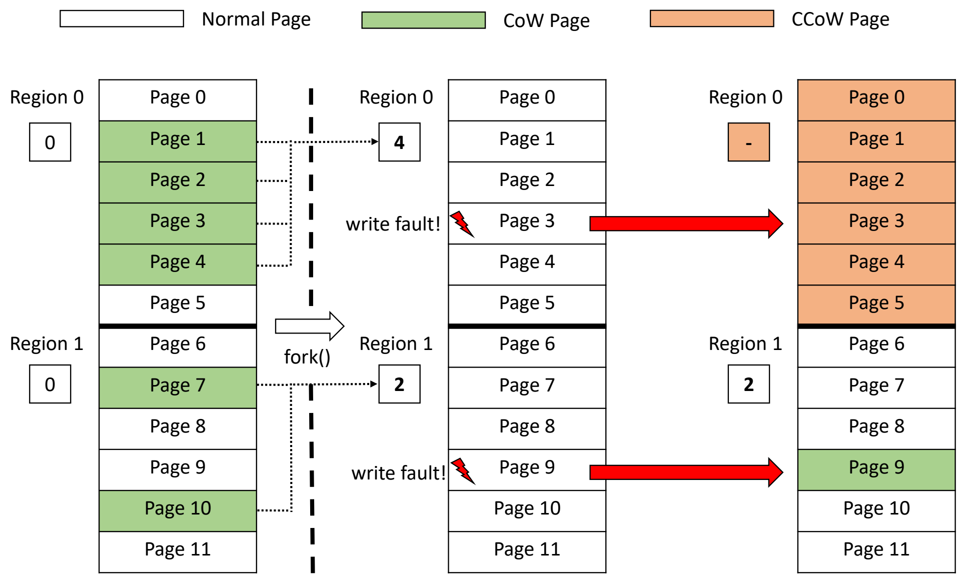


Figure 1. Processing fork and page faults with CCoW in normal regions.

3.3. Tracking Access to Precopied Pages

CCoW is supposed to amortize the overhead for frequent page fault handling overhead. However, copying pages in advance leads to another issue: tracking page accesses after the precopy. In the original copy-on-write scheme, the first write to each page is captured by the page fault handler. Only the faulty page is copied, and the system can precisely track each page access through the page fault handler. On the other hand, when the system precopies an entire region, all pages in the region are mapped to the process with the write permission. Thus, subsequent writes to those precopied pages can take place without triggering the page fault handling mechanism, so the system cannot track the accesses to the precopied pages. This can be problematic when the process performs forks repeatedly. Suppose a region has high spatial locality and an *epoch* is defined as the period between two forks. The high locality can be captured by counting the page faults happened in the region. Then suppose that the process creates a new process with a fork. When a new epoch is started with the fork, the first write access to one of the pages in the region will initiate CCoW, copying all pages in the region. Now the process has all pages with write permission, and no further page faults are generated from the region until the end of the epoch. When a new epoch is started again, the region is considered to have low coverage in the epoch, given that the region only has one copy-on-write page. Thus, each write to the region is processed through the original copy-on-write mechanism per page, thereby compromising the performance optimization opportunity even if the region has high locality. Note that the region is considered to have high locality in the next epoch again, and the process is repeated to alternate the situation.

To resolve this issue, we need a mechanism to track page accesses after precopy. This study proposes leveraging *the dirty bit* in the page table entry (PTE). In general, modern architectures maintain various information in the PTE for each page, and the dirty bit is one of the fields supported by most architectures. When MMU processes a write memory access, it automatically sets the dirty bit of the corresponding page. This conversely implies that when the dirty bit for a page is set, the page has been updated with write accesses. Based on this idea, we modified the mechanism for calculating the coverage. Initially all regions are considered to be the normal regions. During the fork, the coverage for normal regions is calculated with the number of copy-on-writes using Equation (1). While handling the page fault for a high-locality region, all pages in the region are precopied with their dirty bit cleared. Furthermore, the region is marked as a precopied region. During the fork, the coverage for the precopied region is computed with the number of dirty pages in the region as follows:

$$Coverage\ (\%) = (nr\_dirty\_pages / nr\_pages\_per\_region) \times 100 \tag{2}$$

The computed coverage for each region is carried through the fork and used as the metric for the spatial locality of the region. When a write page fault occurs in a region, the system checks the coverage of the region and may precopy pages for the high-locality regions.

Figure 2 illustrates a situation where coverage is calculated after precopies. Suppose both regions are precopied (shaded in red), and pages 0, 2, 3, 4, and 11 all have the dirty bit set. To calculate the coverage while processing the fork, the system uses the dirty bit instead of the copy-on-write count since the regions are precopied in the current epoch. Thus, region 0 has a coverage of 66%. Whereas the coverage of region 1 is 15%. To handle the page fault for page 3, region 0 is precopied again, whereas, the page fault for page 9 is handled with the original mechanism. This way, we can keep high-locality regions from slipping out of focus.

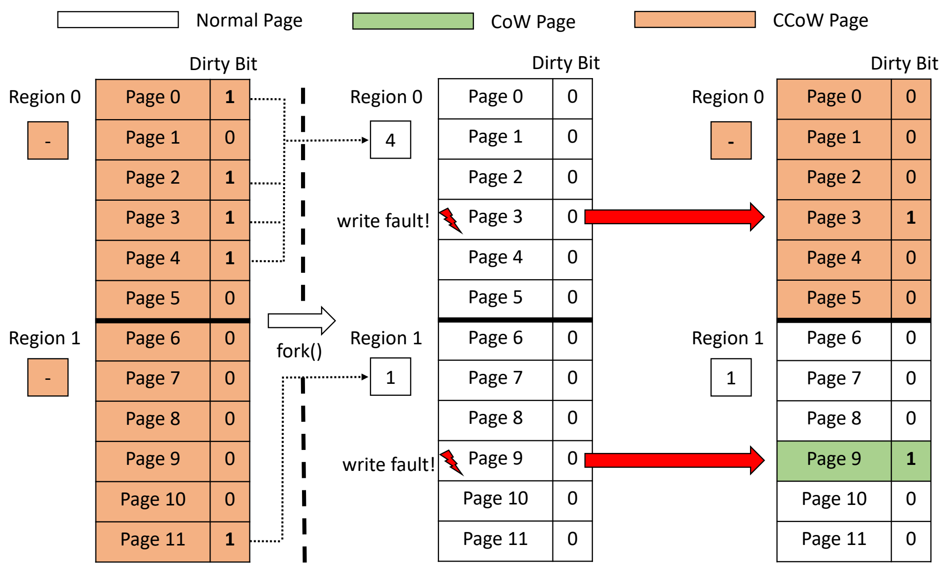


Figure 2. Processing fork and page faults in precopied regions.

3.4. Capturing the Locality

As the page fault handler lies on the performance-critical path in the OS, we should minimize the overhead for the implementation of the proposed scheme. Basically, CCoW requires a mechanism to calculate the coverage, and the simplest way of implementing this feature is tracking the pages triggering page faults with a bitmap as we described in Section 3.2. While handling a page fault, the system sets the bit corresponding to the faulty page. During the fork, the system scans the bitmap to count the number of faulty pages for each region and calculates the coverage. The bitmap is reset after the calculation. This approach is simple but imposes high space and time constraints. The system should maintain one bit bitmap entries for each 4 KB page and inspect the entire bitmap during the fork. Considering the huge memory footprint of memory-intensive applications, these operations will can incur a high overhead during the fork, offsetting the performance benefit of CCoW.

We optimize this implementation by exploiting the characteristics of the page fault. If a page fault happens from a page, the page does not incur additional page faults until the process creates a new process. Thus, each page can trigger one page fault at most, and for a region with n pages, the page faults can only happen up to n times. This implies that counting the number of page faults per region is sufficient to compute the coverage, rather than maintaining the bitmap for individual pages. Thus, we replace the bitmap with the *fault counters*. Each region is associated with a pair of counters as follows: one for counting the page faults in the last epoch, and another for counting the page faults in the current epoch. The former is used to determine the spatial locality of regions, whereas the latter is used to monitor the spatial locality of the current epoch. During the fork, the current fault counter is copied onto the previous fault counter. If a region is precopied in this epoch, the number of dirty pages in the region is written to the previous fault counter instead. The counters are populated while creating a new virtual memory area (VMA), and reclaimed when their corresponding VMA is shrunken or unmapped. This optimization reduces the space overhead of CCoW from one bit per page to a few bytes per region.

4. Evaluation

This section reports the evaluation results of the proposed CCoW scheme. We implement CCoW in the Linux Kernel v5.7.7, and it took approximately 400 lines of code. The evaluation was performed on a server equipped with one Intel Xeon Gold 5215 CPU and 128 GB of memory. To analyze various performance characteristics, we used an in-house microbenchmark. To evaluate on realistic workloads, we used the Yahoo cloud service benchmark (YCSB) [30,31] for Redis [4]. Those programs are configured with the default parameters unless otherwise specified. Because CCoW is implemented at the kernel level, no modification was required for user applications.

4.1. Characterizing CCoW Performance

Since CCoW is controlled by two parameters, namely, the region size and the coverage threshold, these parameters determine the performance and execution behavior of CCoW. In this sense, first, we evaluated the influence of the region size on the performance and overhead of CCoW. We built a microbenchmark program to evaluate the efficiency of copy-on-write. The program is modeled after the execution behavior of Redis. It first populates the 16 GB of memory space divided into 1 KB blocks, and then, a block is selected and updated repeatedly with the predefined data. The benchmark iterates the operations until it writes 160 GB of data is written. The target blocks are selected according to the Zipfian distribution with the parameter $\alpha = 1.0$ to provide a reasonable amount of locality in the accesses. These operations simulate the update operations of Redis with YCSB workloads.

To imitate the snapshot feature of Redis, the benchmark periodically forks a child process. After creating the child process, the performance of the main benchmark process drops sharply due to the increased page fault handling overhead. The performance is recovered and stabilized over time as fewer pages remain for the copy-on-write. We measure the time from performance decline to recover back to the 99% of the normal performance using the original CoW configuration, and used this time as the interval for the forks. The child process was kept idle for two fork intervals before exiting.

We measure the average throughput of the benchmark and the memory footprint of the process while varying the size of region from 32 KB to 2 MB. The throughput indicates the performance gain with CCoW, so the higher is the better. The memory footprint is measured by summing the residential set size (RSS) of the processes, and it indicates the memory overhead of the CCoW scheme. The results are summarized in Figure 3. The original configuration without the precopy is denoted as 'CoW' and the performance values are normalized to that of the CoW configuration.

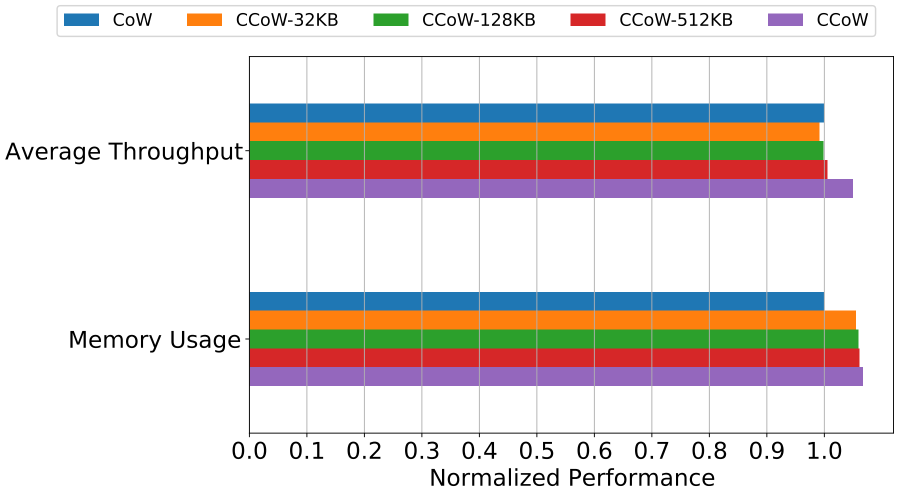


Figure 3. The performance and overhead of CCoW for various region sizes.

Overall, the system performance did not improve significantly in a small region, and worsened with 32 KB regions. Whereas in larger region configurations appreciable improvement was observed. However, the performance change was marginal up to 512 KB region size. This is due to the limited exploitation opportunities in small regions. The monitoring overhead was consistent regardless of the region size. When the region was 2 MB, the benefit outweighed the overhead, and we can observe approximately 5.0% of performance improvement. However, the performance did not improved further with larger region sizes. The memory footprint exhibited a different trend than that of the performance. Even with a small region size, it incurred a considerable amount of memory bloat, which increases as the region size increases. However, it did not increase much even with 2 MB regions. From this evaluation, we can conclude that 2 MB regions provides the maximum performance benefit with a reasonable amount of memory overhead. Thus, we used this region size to the rest of the study.

Next, to find the best coverage threshold for the precopy, we measure the throughput and the memory footprint while changing the CCoW threshold value from 70% to 95%. Figure 4 summarizes the measurement results. All metrics are normalized to that of the original 'CoW' configuration. The number next to 'CCoW-' is the threshold value for the configuration. The 'CCoW-All' configuration is an extreme configuration where the threshold is set to zero so that each page fault copies a 2 MB region. This configuration will effectively behaved similarly to the system with 2 MB huge page.

When the threshold value is high, the system precopies only if it is highly confident. Thus, there is decreased exploitation opportunity, thereby displaying slight performance improvement. In contrast, when the threshold value is too small, there is a high chance for the system to mispredict low-locality regions as high-locality regions. Precopying low-locality regions only incurs overhead without any benefits, offsetting the performance benefit. Thus, the performance peaks at a threshold of 80% and declines for lower threshold values. The space overhead is in inversely proportional to the threshold value. The lower the threshold of the system configuration, the more pages are copied, thereby increasing the memory footprint. For the 'CCoW-All' configuration, we observed a very high memory bloat. In this configuration, each page fault incurs a copy of 2 MB region, eventually making the parent process copy the original data approximately in its entirety. As the child process can run for two fork periods, multiple child instances exist simultaneously, thereby making the accumulated memory footprint very large. Based on this evaluation, we used a coverage threshold 80% for the rest of the study.

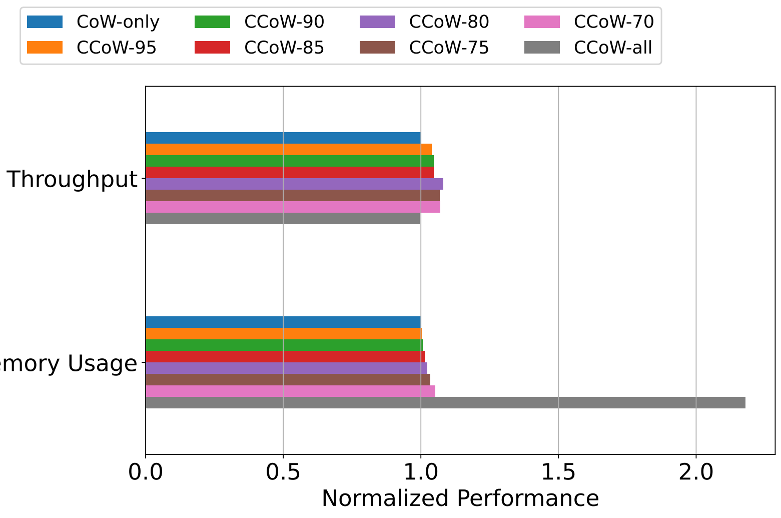
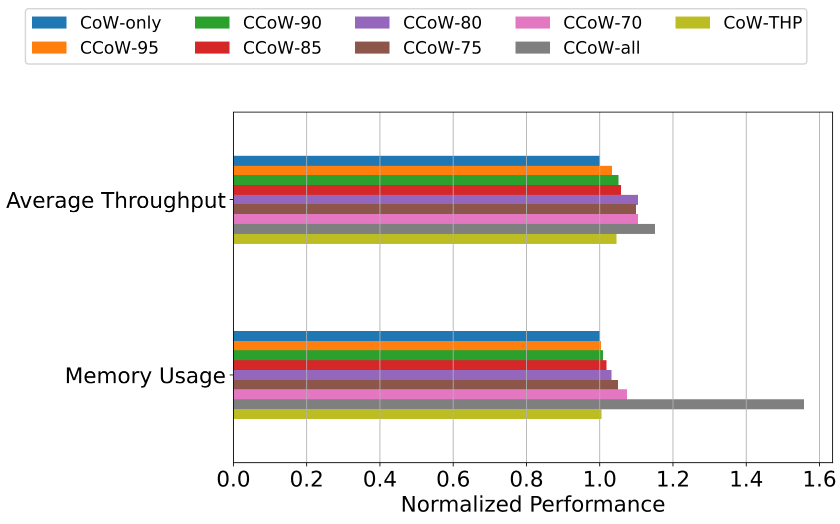


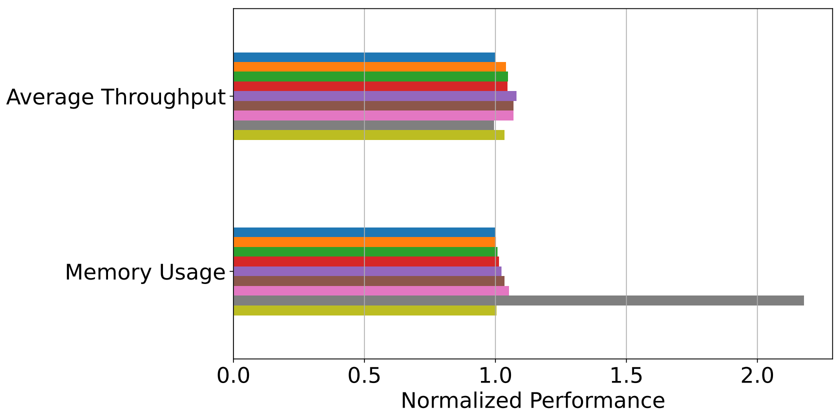
Figure 4. The performance and overhead of CCoW for various thresholds.

The best region size and the threshold vary according to the workload characteristics. To evaluate the influence of workload, we measure the performance of CCoW on the workloads with various localities. Specifically, we changed the parameter α of the Zipf distribution, which determines the degree of locality. The accesses are distributed uniformly when α is 0, and the higher the value of α, the higher the level of locality the workload exhibits. When α is 1.0, approximately 80% of the operations involve 20% of the data. This degree of locality is commonly found in several real workloads, as the Pareto principle states. We measure with three different α values, 1.0, 0.9, and 1.1, where 1.0 is the baseline, and 0.9 and 1.1 represent the low- and high-locality workload, respectively. The original CoW performance varies according to the workloads, so the fork period for a workload was set according to the time measured with the original CoW setup. For example, if the original CoW configuration requires 10 seconds to recover the normal performance after a fork, the other CCoW configurations also fork child processes every 10 s.

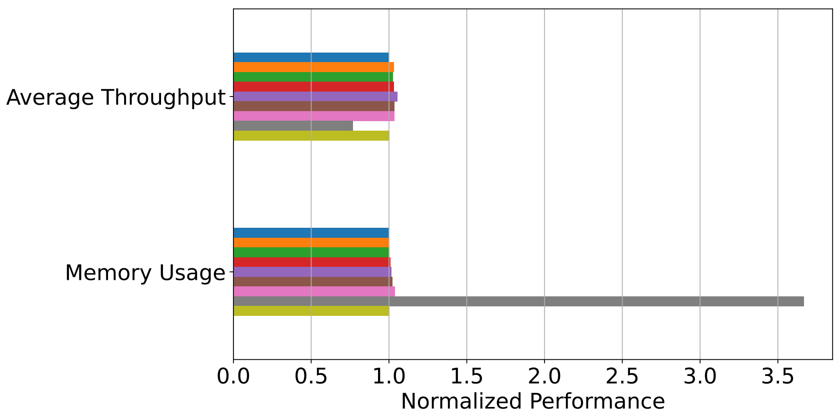
Figure 5 summarizes the average throughput and memory usage of CCoW with different locality workloads. For the low-locality workload, the configurations with small CCoW thresholds exhibit better performance than those with large thresholds. 'CCoW-all' even outperforms the original CoW by 15% in the low-locality workload. This is due to the effectiveness of the precopy. In the low-locality workload, a large part of memory should be replicated as accesses are spread over the entire process address space. In effect, copying entire regions results in the precopying of the necessary memory in advance with low overhead. Thus, the smaller the threshold is, the higher the performance of the program with the low-locality workload. However, this trend has opposite effect with high-locality workloads. With high-locality workloads, many accesses are focused on a few pages. This implies that only a small part of memory needs to be replicated throughout the copy-on-write. Copying the entire region on a page fault tends to copy the pages that are not accessed at all. This only incurs a temporal overhead, impairing the performance with higher-locality workloads. As a result, CCoW-all exhibits the worst performance with the high-locality workload. Other configurations show similar patterns of baseline workloads; the performance peaks at the threshold value of 80% and declines with smaller thresholds.



(**a**)



(**b**)



(**c**)

Figure 5. The performance and overhead of CCoW for various workload localities. (**a**) Low-locality workload ($\alpha = 0.9$). (**b**) Baseline ($\alpha = 1.0$). (**c**) High-locality workload ($\alpha = 1.1$).

The memory usage of the benchmark shows a consistent trend regardless of the degree of locality of the workloads. 'CCoW-all' always represents the highest memory usage because it always copies all pages in the memory after a fork. Besides that, the memory footprints are inversely proportional to the threshold value; the smaller the threshold value is, the more memory the benchmark utilizes. The memory amplification is only increased by up to 10% compared to the original CoW configuration, which is considered to be in a reasonable range.

In addition to analyzing the performance of CCoW, we compare the performance of CCoW to that of the transparent huge page (THP) scheme of the Linux. THP is somewhat similar to CCoW in that it aims at mitigating the overhead originated from small pages. 'CoW-THP' in Figure 5 represents the performance of the THP-enabled configuration. Note that the THP-enabled system handles CoW by splitting huge pages into base pages before copying the faulty page, and so does for other schemes optimizing THP [12–15,17].

We can observe that THP exhibits better performance than the default 'CoW-only' configuration. We attribute the performance gain to the increased efficiency in address translation with huge pages. Specifically, according to the THP scheme, the hot part of the process address space is likely to be broken into base pages, thereby providing the same performance to 'CoW-only' configuration. However, the cold part of the process address space is not split, and maintained with huge pages. Thus, this can boost the application performance to some extend. However, THP does not provide as much performance improvement as CCoW does.

Figure 6 shows the cumulative distribution of the throughput during the evaluation. The x-axis represents the throughput in operations per second, and the y-axis represents the cumulative ratio of the performance to the throughput value. Except for CCoW-all, we can find three frequently observed throughput range regardless of the configurations. The first group in the cumulative ratio of 0 to 0.1 indicates the period during which the benchmark performance declines right after the fork. Then the performance recovers over time, as in the second group with a cumulative ratio of 0.1 to 0.7. The remaining cumulative ratios in the range of of 0.7 to 1.0 are from accesses that do not incur page faults.

Overall, CCoW configurations tend to have more severe performance drops than the original CoW. Specifically, with the high-locality workload of the original CoW scheme, the throughput drops to approximately 1900 K operations per second right after the fork. It then slowly ramps up to the 2500 K operations per second range. With CCoW, the performance dropped more, to the 1700 K operations per second range. However, the performance recovered faster, demonstrating better performance than the original CoW most of the time (i.e., mostly on the right-hand side of the cumulative graph). We can observe the similar trend from other workloads as well, and CCoW-all configuration demonstrates extreme behavior; right after the fork the performance drops significantly, and stays low while the most of the address space is copied with spread accesses. After that point, however, only few page faults occur, so most accesses are processed without page faults. Thus, the throughput has a bimodal distribution in CCoW all.

From this evaluation, we confirmed that CCoW provides optimal performance by optimizing the common case. However, the performance drop should be addressed to obtain better performance characteristics. To this end, we are currently working on to throttling the amount of precopied data right after the fork.

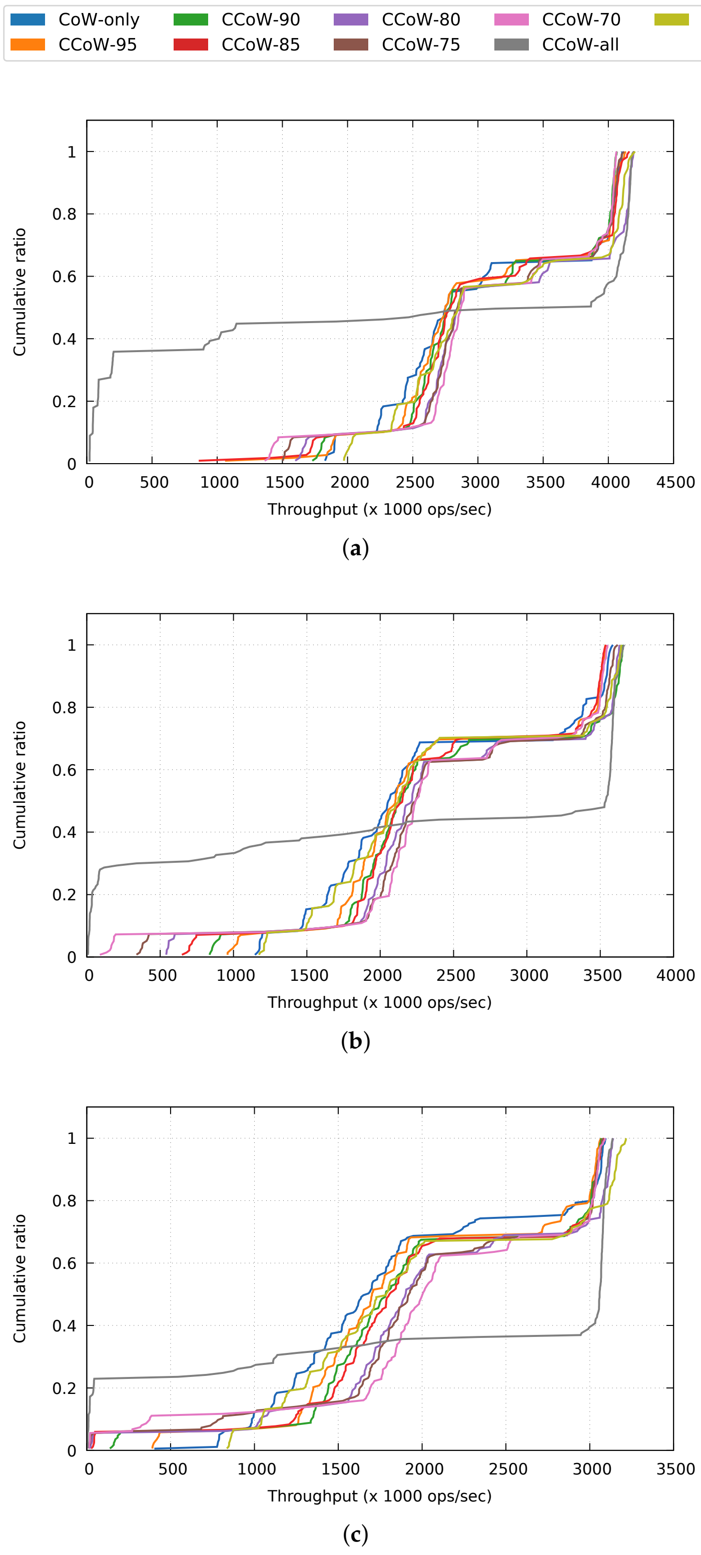


Figure 6. Cumulative distribution of the throughput during the evaluation. (**a**) High-locality workload ($\alpha = 1.1$). (**b**) Baseline ($\alpha = 1.0$). (**c**) Low-locality workload ($\alpha = 0.9$).

4.2. CCoW Performance on Realistic Workload

To evaluate the proposing CCoW on a realistic workload, we used the Redis and YCSB. The Redis is an in-memory key-value database widely used for accelerating Internet-scale applications. We used the YCSB Benchmark to populate key-value pairs in a Redis instance and to perform operations on them. Specifically, the Redis instance is initialized with 10 GB of key-value pairs with the default YCSB configuration. All keys and values are in 23 and 100 bytes in size, respectively, and each key contains 10 fields of values. After populating the Redis instance, we configured it to make snapshots, and then fed update operations with YCSB. To incorporate the temporal locality in the key-value accesses, we set up the YCSB workload to select target keys according to the Zip distribution using the parameter value of 1.0. While making 100 GB of updates, we collected the throughput for every second of the YCSB benchmark report. Figure 7 summarizes the average throughput and memory usage of the Redis instance when the system is configured to use the original CoW or CCoW. Note that we used 2 MB for the region size, and all result values were normalized to that of CoW.

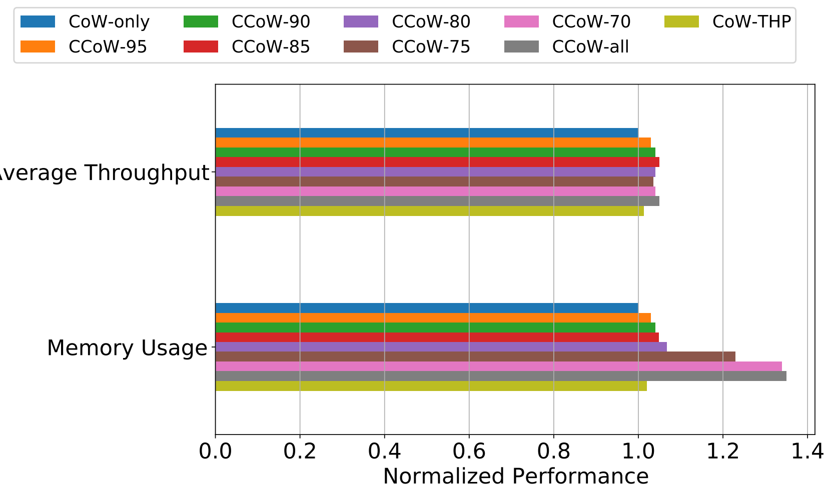


Figure 7. Overall performance of the Redis with CCoW.

Overall, all CCoW configurations outperformed the original CoW, regardless of the coverage threshold. Likewise as we analyzed above, the performance was determined by the trade-off between the performance gain from the mitigated copy-on-write and the overhead of copying additional pages. When the threshold value is high, only few regions are precopied, making both the optimization opportunity and the memory overhead small. When the threshold value decreases below 85%, the memory footprint increases and incurs more overhead. As a result, the average throughput of CCoW varies according to the coverage threshold, but demonstrates up to 5% of performance improvement compared to the original CoW.

With the Redis and YCSB workload, we observed only a marginal performance improvment with THP. This is due to that, in the workload, write accesses are scattered all over the process address space, and huge pages are effectively split into base pages while handling CoW. As the Redis process can have only a few huge pages, its performance is similar to that of the base configuration. This result demonstrates that the THP-based approach is less effective in write-intensive workloads, and CCoW outperforms THP.

To evaluate the accuracy of the mechanism in identifying high-locality regions, we classified the reason for the copy generating mechanism for each copied page. Specifically, we collected the ratio of precopied pages out of all copied pages. When the precopy ratio is x%, increasing the total memory footprint by y%, we can calculate the ratio of unnecessary precopy by dividing y with x. For example, on the CCoW-80 configuration, 26.9% of copied pages are precopied, increasing the memory footprint by 6.7%. This implies that 24.9% of the precopy pages are not referenced. Table 1 summarizes the calculation. The unnecessary precopy ratio ranges from 23.4% to 35.6%, and from the evaluation result it can be concluded that the proposed scheme accurately captures high-locality regions.

Table 1. CCoW rates in the cases.

Case	Precopy Ratio	Increased Memory Footprint	Unnecessary Precopy Ratio
CoW	0%	0%	-
CCoW-95	11.1%	2.6%	23.4%
CCoW-90	12.3%	2.9%	23.5%
CCoW-85	22.4%	5.2%	23.2%
CCoW-80	26.9%	6.7%	24.9%
CCoW-75	77.2%	23.3%	30.2%
CCoW-70	94.8%	33.8%	35.6%
CCoW-all	100%	36%	36%

5. Conclusions

In this study, we proposed CCoW, an optimized copy-on-write scheme for the workloads with high spatial locality. CCoW divides the process address space into regions and estimates their locality with the coverage. A write to a high-locality region leads the page fault handler to precopy nearby pages. To properly track the coverage after the precopy, CCoW leverages the dirty bit in the page table. Evaluation with benchmarks confirmed that the proposed scheme can identify high-locality regions with small overhead, enabling performance gain from applications without a modification.

As we mentioned, the performance drops significantly right after fork due to the huge amount of data to precopy. We are currently working on to manage the performance drop by throttling the rate of precopy and performing the precopy asynchronously. We are also planning to incorporate an adaptive mechanism that tunes the configuration parameters according to the characteristics of the current workload.

Author Contributions: Conceptualization, M.H. and S.-H.K.; methodology, M.H.; software, M.H.; validation, M.H. and S.-H.K.; formal analysis, M.H. and S.-H.K.; investigation, M.H. and S.-H.K.; resources, S.-H.K.; data curation, M.H.; writing—original draft preparation, M.H.; writing—review and editing, M.H. and S.-H.K.; visualization, M.H.; supervision, S.-H.K.; project administration, S.-H.K.; funding acquisition, S.-H.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (20ZS1310) and the BK21 FOUR program of the National Research Foundation of Korea funded by the Ministry of Education (NRF5199991014091).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gorman, M. *Understanding the Linux Virtual Memory Manager*; Prentice Hall: Upper Saddle River, NJ, USA, 2007.
2. Bovet, D.P.; Cesati, M. *Understanding the Linux Kernel*; O'Reilly: Newton, MA, USA, 2001.
3. Love, R. *Linux Kernel Development*, 3rd ed.; Addison Wesley: Boston, MA, USA, 2010.
4. Labs, R. Redis. Available online: https://github.com/redis/redis (accessed on 7 June 2021).
5. Silberschatz, A.; Galvin, P.B.; Gagne, G. *Operating System Concepts*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2018.
6. Harris, S.L.; Harris, D. *Digital Design and Computer Architecture*; Morgan Kaufmann: Burlington, MA, USA, 2022.
7. Abi-Chahla, F. Intel Core i7 (Nehalem): Architecture By AMD? Available online: https://www.tomshardware.com/reviews/Intel-i7-nehalem-cpu,2041.html (accessed on 18 October 2021).
8. Pham, B.; Bhattacharjee, A.; Eckert, Y.; Loh, G.H. Increasing TLB reach by exploiting clustering in page translations. In Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14), Orlando, FL, USA, 15–19 February 2014; pp. 558–567.

9. Baruah, T.; Sun, Y.; Mojumder, S.A.; Abellán, J.L.; Ukidave, Y.; Joshi, A.; Rubin, N.; Kim, J.; Kaeli, D. Valkyrie: Leveraging Inter-TLB Locality to Enhance GPU Performance. In Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT'20), Virtual, 5–7 October 2020.
10. Vavouliotis, G.; Alvarez, L.; Karakostas, V.; Nikas, K.; Koziris, N.; Jiménez, D.A.; Casas, M. Exploiting Page Table Locality for Agile TLB Prefetching. In Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21), Valencia, Spain, 14–18 June 2021.
11. Schildermans, S.; Aerts, K.; Shan, J.; Ding, X. Ptlbmalloc2: Reducing TLB Shootdowns with High Memory Efficiency. In Proceedings of the 2020 IEEE International Conference on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom), Exeter, UK, 17–19 December 2020.
12. Kwon, Y.; Yu, H.; Peter, S.; Rossbach, C.J.; Witchel, E. Coordinated and Efficient Huge Page Management with Ingens. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16), Savannah, GA, USA, 2–4 November 2016; pp. 705–721.
13. Kwon, Y.; Yu, H.; Peter, S.; Rossbach, C.J.; Witchel, E. Ingens: Huge Page Support for the OS and Hypervisor. *SIGOPS Oper. Syst. Rev.* **2017**, *51*, 83–93. [CrossRef]
14. Panwar, A.; Bansal, S.; Gopinath, K. HawkEye: Efficient Fine-Grained OS Support for Huge Pages. In Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19), Providence, RI, USA, 13–17 April 2019; pp. 347–360.
15. Zhu, W.; Cox, A.L.; Rixner, S. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20), Boston, MA, USA, 15–17 July 2020; pp. 829–842.
16. Park, C.H.; Cha, S.; Kim, B.; Kwon, Y.; Black-Schaffer, D.; Huh, J. Perforated page: Supporting fragmented memory allocation for large pages. In Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20), Valencia, Spain, 30 May–3 June 2020; pp. 913–925.
17. Yan, Z.; Lustig, D.; Nellans, D.; Bhattacharjee, A. Nimble Page Management for Tiered Memory Systems. In Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19), Providence, RI, USA, 13–17 April 2019; pp. 331–345.
18. MongoDB. Available online: https://github.com/mongodb/mongo (accessed on 9 September 2021).
19. Zhao, K.; Gong, S.; Fonseca, P. On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In Proceedings of the 16th European Conference on Computer Systems (EuroSys'21), Online, 26–28 April 2021; pp. 540–555.
20. Baumann, A.; Appavoo, J.; Krieger, O.; Roscoe, T. A Fork() in the Road. In Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19), Bertinoro, Italy, 13–15 May 2019; pp. 14–22.
21. Ge, Y.; Wang, C.; Shen, X.; Young, H. A Database Scale-out Solution for Emerging Write-Intensive Commercial Workloads. *SIGOPS Oper. Syst. Rev.* **2008**, *42*, 102–103. [CrossRef]
22. Li, C.; Feng, D.; Hua, Y.; Qin, L. Efficient live virtual machine migration for memory write-intensive workloads. *Future Gener. Comput. Syst.* **2019**, *95*, 126–139. [CrossRef]
23. Reiss, C.; Tumanov, A.; Ganger, G.R.; Katz, R.H.; Kozuch, M.A. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC'12), San Jose, CA, USA, 14–17 October 2012.
24. Zhang, Q.; Zhani, M.F.; Zhang, S.; Zhu, Q.; Boutaba, R.; Hellerstein, J.L. Dynamic Energy-Aware Capacity Provisioning for Cloud Computing Environments. In Proceedings of the 9th International Conference on Autonomic Computing (ICAC'12), San Jose, CA, USA, 16–20 September 2012; pp. 145–154.
25. Sharma, B.; Chudnovsky, V.; Hellerstein, J.L.; Rifaat, R.; Das, C.R. Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters. In Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11), Cascais, Portugal, 26–28 October 2011.
26. Zhang, Q.; Hellerstein, J.; Boutaba, R. Characterizing Task Usage Shapes in Google Compute Clusters. In Proceedings of the 5th International Workshop on Large Scale Distributed Systems and Middleware (LADIS'10), Seattle, WA, USA, 2–3 September 2011.
27. Sebastian, A.; Le Gallo, M.; Khaddam-Aljameh, R.; Eleftheriou, E. Memory devices and applications for in-memory computing. *Nat. Nanotechnol.* **2020**, *15*, 529–544. [CrossRef] [PubMed]
28. Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Fogh, A.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; et al. Meltdown: Reading Kernel Memory from User Space. In Proceedings of the 27th USENIX Security Symposium (USENIX Security'18), Baltimore, MD, USA, 15–17 August 2018.
29. Kocher, P.; Horn, J.; Fogh, A.; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; et al. Spectre Attacks: Exploiting Speculative Execution. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP'19), San Francisco, CA, USA, 18–19 May 2019.
30. Cooper, B.F. YCSB: Yahoo! Cloud Serving Benchmark. Available online: https://github.com/brianfrankcooper/YCSB (accessed on 21 June 2021).
31. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking Cloud Serving Systems with YCSB. In Proceedings of the Symposium on Cloud Computing (SoCC'10), Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154.