Lodic: Logical Distributed Counting for Scalable File Access

Jeoungahn Park, *KAIST;* Taeho Hwang, *Hanyang University;* Jongmoo Choi, *Dankook University;* Changwoo Min, *Virginia Tech;* Youjip Won, *KAIST*

https://www.usenix.org/conference/atc21/presentation/park

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

LODIC: Logical Distributed Counting for Scalable File Access

Jeoungahn Park[*] Taeho Hwang[†] Jongmoo Choi[‡]
Changwoo Min [§] Youjip Won [*]

[*]*KAIST, Korea* [†]*Hanyang University, Korea* [‡]*Dankook University, Korea* [§]*Virginia Tech, USA*

Abstract

We develop a memory-efficient manycore-scalable distributed reference counter for scalable file access, *Logical Distributed Counting* (LODIC). In Logical Distributed Counting, we propose to allocate the local counter on a *per-process* basis. Our process-centric counter design saves the kernel from the excessive memory pressure and the counter query latency issues in the existing *per-core* based distributed counting schemes. The logical distributed counting is designed to dynamically incorporate the three characteristics for reference counting: i) the population of the object, ii) the reference brevity, and iii) the degree of sharing. The key ingredients of the logical distributed counting are *Memory mapping*, *Counter Embedding*, and *Process-space based reverse mapping*. Via mapping a file region to the process address space, LODIC can allocate the local counter at the process address space. With Counter Embedding, the logical distributed counting defines the local counters without the significant changes in the existing kernel code and without introducing significant memory overhead for the local counters. Exploiting the virtual memory segment allocation algorithm of the existing Linux kernel, the process-space based reverse mapping locates the local counter of the physical page without the substantial overhead. Logical Distributed Counting increases the throughput by 65× against stock Linux in reading the shared file block. LODIC exhibits as good performance as the ideal scalable reference counter when deployed in the RocksDB (key value storage) and NGINX (web server) applications.

1 Introduction

Reference counting is a vital part of the modern Operating System (OS). Various kernel objects, e.g. page frame, inode, and file descriptor table maintain the reference counter to prohibit them from being reclaimed prematurely while allowing them to be accessed concurrently by a number of processes without the exclusive lock. The contention on updating the reference counter makes the associated cacheline

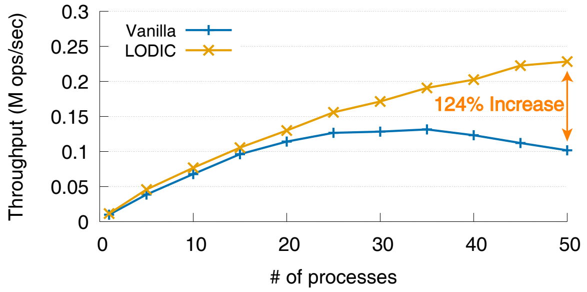


Figure 1: Web server performance accessing the shared web page: `wrk` benchmark [69] of NGINX web server engine, 120 CPU-cores (Intel Xeon E7-8870 v2 processors, 8 sockets and 15 cores per socket) and 780GB DDR3 DRAM

to be fetched across the CPU cores, and renders the scalability failure due to the cacheline bounce. The importance of the scalable reference counting gets emphasized further as the computer system is loaded with a larger number of CPU cores [17, 23, 31, 40]. Recent demands for the larger main memory in Deep Learning [22], Graph Analysis [48], Virtual Machine consolidation [35, 68], and Big Data Analytics [41] applications make the Operating System to host a larger number of kernel objects and compound the importance of the scalable and the memory-efficient reference counting. Machines with hundreds of the cache coherent cores and the multi-terabytes of the main memory are currently available in the market today: SGI's Ultra Violet 3000 [10], Dell's PowerEdge R920 [7], and HPE's Superdome X servers [37]. In the foreseeable future, we expect a system with thousands of cache coherent low-processing power cores and with even petabytes of main memory [38].

Reading a file block is one of the most essential operations in the computer system, e.g., accessing the web page, searching the database, scanning the key-value file, and etc. Immature implementation of the reference counter for the page frame in the modern Operating Systems makes the shared file block access easily vulnerable to cacheline bounce associated with the reference counter update, leading to scalability and performance collapse [51].

We examine the performance of web server where a popular web page is shared by the number of clients. We increase the number of clients that request for the same web page. Web server daemon deploys multiple processes. Each process services the web page requests from the clients. Fig. 1 illustrates the results. In vanilla Linux that adopts the global reference counting for the page cache entry, the web server throughput saturates at 30 cores. With our logical distributed counting, the web server throughput increases as much as by 2.23× when there are fifty processes. This simple experiment shows that from the application's point of view, reference counting is a vital component in making the shared file block access scalable.

A fair number of works have been proposed to mitigate the contention on the global reference counter [5,14,17,23,26,31,40]. They mitigate the contention on the global counter via allocating the per-core local counters and subsequently via distributing the accesses to the global counter to a number of local counters. They represent a global state of the references using a set of local counters. These works trade the memory pressure and the counter query latency for the scalability for the counter update. To mitigate the memory pressure for the local counters, the recent works propose to allocate the counter cache for each core. The per-core counter cache hosts the recently accessed counters [16,18,23,40].

The existing distributed counter designs are grounded upon the view that the cacheline bounce is caused by the contention among the processors. This processor-centric view on the cacheline contention leads the kernel to define the local counters for each processor core. They blindly define the same number of local counters for all objects regardless of the access popularity. The per-core based distributed counting schemes fail to take into account the actual degree of sharing and impose overly pessimistic estimation on the number of local counters that are required to mitigate the counter contention.

In this work, we view that the cacheline contention in updating the reference counter is driven by the contention among the *processes*, not by the contention among the *processors*. Based upon this new view, we propose to allocate the local counters for a given object in *per-process* basis. We allocate the local counters for a given object to each process that accesses it. We call this distributed counting scheme, *Logical Distributed Counting*, LODIC for short. We call it a *logical* counter since the local counters are associated with the logical entity, the process, not with the physical entity, the processor. In LODIC, the counter query latency is governed by the actual number of processes that share a given file block.

LODIC is designed to address the scalability issue in reading the shared file block [40,51]. There are three characteristics that need to be incorporated in designing the reference counting scheme: the degree of sharing, the object population, and the reference brevity. Each kernel object, e.g., page frame, dentry entry, inode, and file descriptor table, has widely different reference characteristics along these three axis. The reference counter design should be tailored with respect to the characteristics of the object.

The Logical Distributed Counting consists of three key ingredients: (i) file mapping, (ii) counter embedding, and (iii) process-space based reverse mapping. The first ingredient is to map a file region that needs to be shared to the process address space. This plain and simple mechanism provides a foundation for the logical distributed counting. Via attaching the file-backed page frame to the process address space, LODIC enables the kernel to define the reference counter for the page frame in the process address space, i.e., in per-process basis. The second ingredient is Counter Embedding. With Counter Embedding, LODIC represents the logical counter using the unused bits in page table entry (PTE). Counter Embedding eliminates the need to define a new kernel data structure to represent the local counter and makes the logical counter memory-efficient. The third ingredient is process-space based reverse mapping. Locating the page table of a given page frame is the most expensive task in accessing the local counter in logical distributed counting. For reverse mapping, LODIC scans the virtual memory segments of the process, the *process-space*, unlike the existing reverse mapping feature, `rmap()` [24] that scans the virtual memory segments associated with the file, the *file-space*. The virtual segment allocation algorithm of Linux tends to place the file mapped segments at the high-end of the process virtual address space. Exploiting this nature, the process-space based reverse mapping of LODIC can locate the page table for a given page frame in nearly constant amount of time regardless of the degree of sharing and successfully scales with the degree of sharing. The contribution of LODIC can be summarized as follows.

- Logical Distributed Counting allocates the local counters with respect to the *actual degree of sharing*. The number of local counters for a file block corresponds to the number of processes that map the block which can be substantially smaller than the total number of CPU cores in the large scale manycore system.
- Logical Distributed Counting scheme develops a scalable reverse mapping technique, *process-space based reverse mapping*. It effectively exploits the virtual memory allocation algorithm of existing Linux and can locate the logical counter within a constant amount of time in common cases. The process-space based reverse mapping makes the reverse mapping overhead sufficiently small, whose performance impact associated with accessing the local counter becomes hardly visible from outside.
- Logical Distributed Counting is nearly memory free. The logical counter in LODIC exploits the unused bit space in the page table entry to represent the reference counter. LODIC successfully addresses the memory pressure issue in the logical counting.
- With all these benefits, LODIC increases the performance

of shared block read by 64× compared to the stock Linux. When the file block is shared and accessed by 120 cores.

2 Background and Motivation

2.1 Reference Counting in File Block Access

Page Cache and Reference Counting. OS kernel manages the page cache (or buffer cache) to speed up the file access by caching the frequently accessed disk contents to memory. Linux kernel (and many other OSes) manages per-file (`inode`) page cache. A page cache maintains the mapping information from a file offset to a physical page frame that holds the associated disk content. In servicing a `read/write` system call, OS kernel first looks up the page cache to avoid costly storage access. OS kernel tends to keep as much file contents as possible in a page cache. However, under the memory pressure, OS kernel evicts infrequently accessed pages to secure more free memory. One invariant is that OS kernel should not reclaim a page which is still being accessed. Reference counting is commonly used to track the page access count in many OS kernels including Linux. For instance, OS kernel (atomically) increases per-page reference counter before accessing a page in the page cache and decreases it after the access completes. A positive page reference counter means that the page is being accessed and that the page should not be reclaimed.

Page Cache Reference Counting in Linux Linux kernel maintains per-`inode` page cache (`address_space`) using a radix tree (`XArray`). The page cache maps a file offset to `struct page`, which is metadata on a physical page frame. Linux kernel maintains two atomic reference counter per-page. They are defined in the `struct page`; `_mapcount` representing how many times a page is `mmap`-ed and `_refcount` representing how many tasks are accessing a page [50]. When a file is shared among the multiple processes concurrently, the concurrent update on the `_refcount` becomes the performance bottleneck [51]. When a page frame is chosen for the reclamation, the kernel unlinks it not only from the page cache but also from any page table entries that map a given page cache entry. To quickly locate the associated page table entries, the kernel maintains a reverse mapping from a page frame to a page table entry [24, 25, 43, 67]. Linux kernel maintains an interval tree, which is a set of virtual memory segments mapping a given port of a file to virtual address space (stored in `mapping` field under `struct address_space`).

Challenges in Page Cache Reference Counting. There are two challenges in page cache reference counting. First, the number of reference counters can be prohibitively large. For example, 1 million `_refcounts` are needed to cache 40 GB in Linux. Next, the access characteristics (*e.g.*, access frequency) of a page cache reference counter is determined by the applications that accesses the files, which is out of kernel's control. Therefore, any reference counting scheme for page cache should be memory-efficient to deal with the large memory capacity and should be adaptable to application's file access behavior.

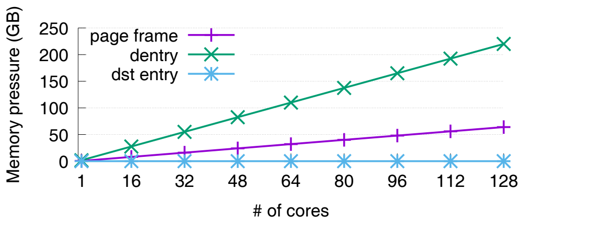


Figure 2: Memory consumption of sloppy counters for page frame (`struct page`, `dentry` and `dst_entry`) when DRAM size is 256GB, the number of `struct page` is 67M, the number of `dentry` is 220M, and the number of `dst_entry` is 32K.

2.2 Scalable Reference Counting

Per-Core Distributed Reference Counting. To avoid the performance collapse caused by cacheline bouncing in atomic counter (used in stock Linux kernel), per-core distributed reference counting schemes – *e.g.*, Scalable Non-Zero Indicator (SNZI) [31], Sloppy Counter [17], and Approximate Counters [26, 27] – have been proposed. This approach manages the per-core local counters in addition to the central object counter. In the common case, each thread accesses its per-core local counter, not incurring cacheline bouncing.

While this approach solves the performance problem of the atomic counter, it has two critical problems that we opted out using it for page cache. First of all, its memory consumption linearly increases as the number of objects and the number of CPU's increase (see Table 1). When a large number of objects (*e.g.*, millions page frames [1,3]) exist in a multi-core machine having tens or hundreds of CPU's, its memory overhead is detrimentally large. For example, Figure 2 shows that sloppy counter requires 60GB of additional memory just to store the reference counters for the page frames (`struct page`) in a 128-core machine with 256GB DRAM (24%). Another drawback is the query performance – getting the true counter value – is often proportional to the number of CPU's. In sloppy counter, the entire local counters should be traversed to get the true value (see Table 1). In managing the page cache, the slow query can slow down the reclamation of page frames and can cause unnecessary swapping in the worst case.

Hash Table Based Reference Counting. To mitigate the excessive memory usage of the per-core schemes, per-core hash table based distributed counting schemes have been proposed. They store local counters at the per-core hash table to bound the memory usage regardless of the number of objects.

However, they do not fundamentally solve the limitation of per-core distributed reference counter. Still their memory consumption is proportional to the number of CPU's as shown in Table 1. Also, when the hash collision happens, they perform its slow path incurring the central contention:

	Atomic Counter	**SNZI** [31]	**Sloppy Counter** [17]	**RefCache** [23]	**PayGo** [40]	**LODIC**
Counting Overhead	Contending atomic ops	Non-contending atomic ops	Global lock	Non-atomic ops	Mostly non-atomic ops	Mostly non-contending atomic ops
Space Overhead	$O(N)$	$O(N{\cdot}C)$	$O(N{\cdot}C)$	$O(C{\cdot}H+N)$	$O(C{\cdot}H+N)$	$O(N)$
Query Overhead	$O(1)$	$O(1)$	$O(C)$	$O(1)+2{\cdot}epoch$	$O(C)$	$O(S)$
Time Overhead	None	None	Every threshold	Every epoch and collision	Every hash collision	None

N: # of objects C: # of CPUs H: size of hash table S: degree of sharing

Table 1: Comparison of reference counting techniques.

for example, RefCache [23] evicts the original entry to the central reference and PayGo [40] evicts to the central overflow counter list. Therefore, unless the hash table size is large enough (*e.g.*, 2× of active objects), they still suffer from the hash table collision and the slow path execution. Also, they do not improve the query performance because they have to traverse the entire per-core hash tables. In addition to the traversal overhead, RefCache has to wait for two epoch periods until the counter converges, which incurs more delay in getting the true counter value.

3 Design Principles of LODIC

In this section, we present three design principles of LODIC, a scalable reference counting scheme designed for page cache and file access. LODIC should be able to deal with a large number for objects in multi-core machines leveraging the skew of file access and the property of page cache access.

P1. Millions of Counters Are Not Uncommon. Recent advances in multi-core processors and memory technologies make a server equipped with hundreds of cores and hundreds GB to a few TB main memory prevalent. For example, a recent 2-socket Intel Xeon server has up to 112 logical cores and 9 TB memory (3 TB DRAM and 6 TB NVM). We expect more cores with larger memory will be prevalent to achieve higher performance in data-centric applications.

Reference counting should be scalable to the memory capacity and the number of CPU cores. It should scale at least millions of page frames (40 GB memory == 10M reference counters) with a hundred CPU's. To be future-proof, it should scale a few hundred millions (*i.e.*, TB scale) with a few hundreds CPU's. *Ideally, the memory consumption should solely depend on the number of reference counters and the query performance should not depend on the number of cores, which have been increasing.* While the counter caching approach using per-core hash table [23, 40] partly mitigates memory pressure, the current approaches relying on per-core structures [17, 23, 26, 27, 31] are not appropriate in both memory consumption and query performance.

P2. Not All File Pages Are Popular. It is well-known that real-world data access are skewed. Examples include the web page accesses [19], the popularity of the vocabularies in the dictionary [28], and the access distribution in the key-value store [20] to list a few. That implies that the accesses to a small subset of file pages account for dominant fraction of all file accesses while most file pages are not concurrently

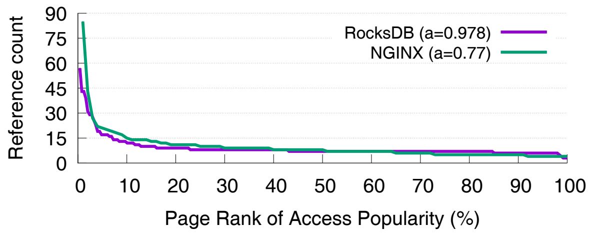


Figure 3: Maximum reference counter distribution of page frames for file access under real-world skewed key-value store and web server accesses on a 120-core server.

accessed at all. The existing distributed reference counting schemes relying on per-core structures [17, 23, 26, 27, 31] fail to exploit such access skew. These works assume that all objects are concurrently accessed from all CPUs. Subsequently, they blindly pre-allocate same number of per-core local counters or same amount of per-core counter cache (hash table) for all objects. The pessimistic assumption on the degree of concurrent access renders overly excessive memory pressure and the query overhead. *Ideally, reference counting scheme should allocate the local counters with respect to the actual degree of sharing, the actual number of processes that share a page.*

We examine the impact of access skew in reference counter design; we run RocksDB key-value store [34] and NGINX web server [60]. For key-value store access, we use the access skew of the English dictionary (Zipf distribution with $\alpha = 0.98$) [28]. For web page access, we use the measurement results for web page access (Zipf distribution with $\alpha = 0.77$) [19]. In this experiment, we record the largest reference counter value in each page frame, which we observed while running the workloads. 100 concurrent clients stress RocksDB having 17,000 key-value pairs with 100-byte values. NGINX runs with 100 concurrent workers. Both were run on a 128-core server. Figure 3 shows the distribution of the maximum reference counter values. X-axis denotes the popularity rank of the pages. Unsurprisingly, the distribution is highly skewed. The sum of the maximum reference counter values for the individual file pages corresponds to the upper bound on the total number of local counters, which are required to represent the concurrent accesses. The sloppy counter [17] allocates 120 local counters for each page. According to this experiment, sloppy counter creates 13.6× more local counters for RocksDB and 12× more local counters for NGINX than are actually needed, respectively.

P3. Reference Duration Is Extremely Short in File Access. One important factor in designing the distributed reference scheme is how to handle the reference split. A process can be scheduled to the different core while the reference is active. Then, the original local counter becomes to reside at the different core from the core where the process is running as a result of the process migration. We call this situation as reference split.

There are two types of approaches to address the reference split problem. The first type of approach is to eliminate the possibility of the reference split. One can temporarily disable the interrupt [47] or the preemption [45] to prevent the process migration. Another type of approach is associated with resolving the reference split when it happens. When the reference split happens, the process can decrease (*i.e.*, unreference) either the local counter at the current core (*e.g.*, RefCache [23]) or the original local counter at the remote core (*e.g.*, PayGo [40]).

Each type of approaches has its own disadvantage; the limited usage, the excessive query latency and the memory overhead. The first is the limited usage. Disabling the interrupt or the preemption cannot be used if the code does not allow to disable the interrupt or the preemption. The second is excessive query latency. RefCache should wait for two epochs to get the real reference counter value. Refcache trades the query latency with the cost of decrementing the counter at the local core. The third is the memory overhead. PayGo requires not only the local counter but also the anchor counter at each core. Also, PayGo needs to maintain the anchor ID at each task struct to handle the reference split.

The above mentioned techniques for the reference split may not deserve its the overhead if the reference split is unlikely to happen. More importantly, all these elaborate schemes deserve its overhead only when the reference split happens frequently. *Therefore, an ideal approach should pay the cost of the reference split only when it really happens.*

To understand how often the reference split can happen in accessing the page cache, we measured the time interval between the reference and the unreference operations for five popular kernel objects including `struct page`. As Figure 4 shows, the reference duration of `page` is very short (0.14 μsec). For page access, the reference split happens very rarely (0.0005%, 5 out of one million page references). To measure the reference duration of `page`, we repeatedly read 4KB file block for 30 seconds. *Based on our measurement, we carefully conclude that any feature for handling the reference split may hardly deserve its overhead for the reference counting for page cache.*

4 Design of Lodic

4.1 Design Overview

Lodic is designed to scale with a large amount of physical memory and with a large number of CPU's by leveraging the file access skew and the short access duration. We designed

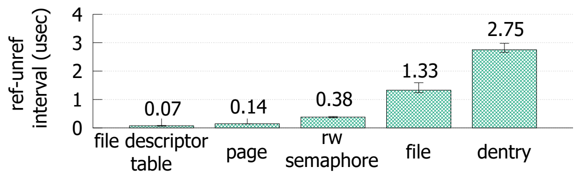


Figure 4: Reference duration for different kernel objects in Linux OS: file descriptor table (`struct files_struct`), physical page frame (`struct page`), read-write semaphore (`rw_semaphore`), file object (`struct file`) and directory entry (`struct dentry`).

Lodic in the following three key insights:

First, per-core local counter approaches are by design memory inefficient especially for the file-backed page frames. In file block accesses, the accesses towards a small subset of popular pages account for the dominant fraction of file block accesses as shown in Figure 3. To well exploit the characteristics of the shared file page access, we carefully argue that allocating the local counter in *per-process* basis than in *per-core* basis is appropriate for accessing the page frames. If the local counters are allocated only for the processes which actually share the given file, the memory pressure for the local counters does not increase linearly with CPU-core count but it is only dependent on the actual degree of page sharing. The memory saving of the per-process local counter against the per-core local counter can be particularity substantial in the large scale machine with hundreds of cores since the number of processes that share the given file can be much smaller than the number of CPU cores in the system. Moreover, Lodic reduces the query overhead substantially because it checks the local counters of only those processes that are actually sharing the file. In theory, the number of processes sharing a file can be greater than the number of CPU's. However, we believe such case is rare in practice because, for example, the administrator guides in many servers suggest not to create more (worker) processes beyond the number of CPU's to avoid unnecessary contention on the shared resources [58].

Next, we propose a *selective distributed counting scheme* that exploits the skew in the file accesses. Lodic can selectively apply the distributed counting for the fraction of a file. Not all file blocks in a file are frequently accessed. There may exist a hot region in a file whose reference counting becomes a bottleneck. Thus, we pay the extra overhead of distributed counting when only needed. For instance, the first few levels in B+-tree and log-structured merge (LSM) tree [59] and frequently accessed web pages are the typical examples of the hot file regions. Note that dynamically detecting the hot file blocks are well studied in the prior works [21, 52] and it is out of scope of this paper.

Lastly, we optimize Lodic's *design for the common case that reference split not happening* unlike the prior works [23, 40] preparing handling of the reference split all the time. As shown in §3, the reference split happens extremely rarely.

Lodic exploits two properties of the modern computer system design: (i) there exists a few unused bits left in PTE and

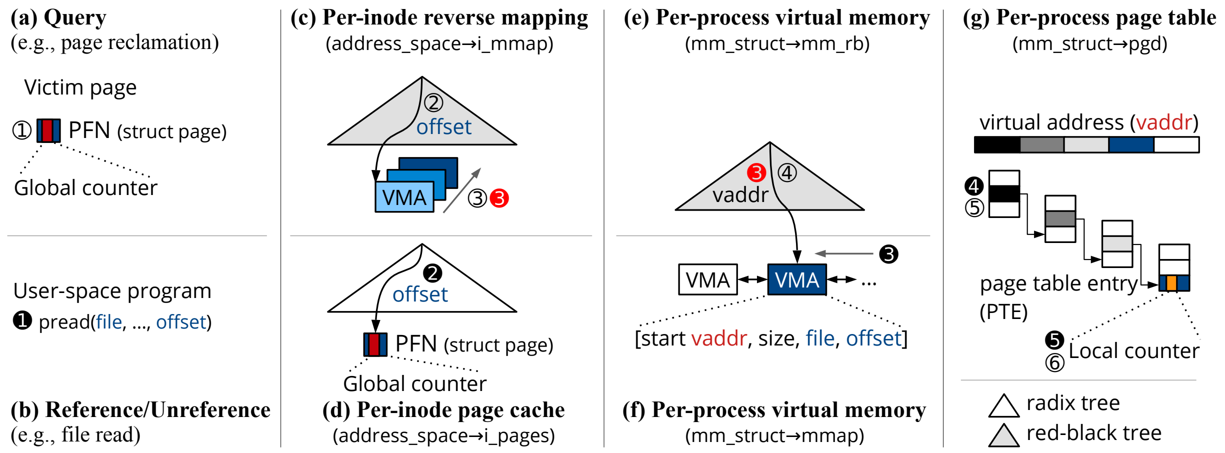


Figure 5: Illustrative examples of LODIC for reference/unreference operations (ⓝ) and a query operation (ⓝ). ❸ is an unoptimized step, which replaced by LODIC's optimized step ❸ (see the details in §4.4.2).

(ii) Operating System uses the search tree structures to organize the segments in a virtual address space. While the LODIC is currently built on top of Linux, it can be applied to the other Operating Systems. For example, similar to Linux that uses the red-black tree, FreeBSD and Windows use a splay tree and AVL tree, respectively, to manage process address space.

In the next sections, we present three main components of LODIC: (1) per-process selective distributed counting (§4.2), (2) efficient access of a local counter (§4.4), and (3) local counter embedding to page table entry (PTE) (§4.3). We then summarize how these are used for each LODIC operations (§4.5). Figure 5 illustrates the overview of LODIC design for reference/unreference and query operations.

4.2 Selective Distributed Reference Counting

LODIC realizes the per-process distributed counting by `mmap`-ing a file region to the process virtual address space. By doing so, LODIC allocates the local counters for the virtual pages in the mapped file region. When a program accesses the file block of the mapped file, instead of updating the global reference counter (`_refcount`) in the page frame (`struct page`), LODIC updates the local counters defined for the associated virtual page. The local counter is defined at the process virtual address space. The number of local counters for a given physical page corresponds to the number of virtual pages that map the given physical page frame. This approach deploys the distributed counters for the mapped file regions and for the processes that map a given file.

When a page frame is accessed (❶❷ in Figure 5), LODIC kernel first checks if the given physical page is `mmap`-ed page or not. If the page is a `mmap`-ed one, the kernel performs reverse mapping to find a virtual address of the page (❸ in Figure 5) then updates the local counter embedded in the associated page table entry (❹❺ in Figure 5). If the page is not `mmap`-ed (*i.e.*, `_mapcount == 0`) or the contention is low, the kernel falls back to updating the global reference counter in the page frame. Currently, LODIC determines that the contention on the page is low when the page is shared by less than four processes (*i.e.*, `_mapcount < 4`).

4.3 Local Counter Embedding to PTE

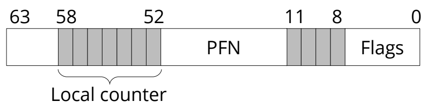


Figure 6: LODIC embeds a local counter to unused bits (colored in gray). In x86-64 architecture, a counter is embedded in bit [58:52].

In allocating the per-process local counter, we propose a *Counter Embedding* technique. With Counter Embedding, LODIC defines the local counter without allocating the separate memory. In Counter Embedding, we represent the per-process local counter using the unused (ignored) bits in the page table entry (PTE). Modern processor architectures leave a few bit in the PTE for software use. For example, x86-64 [12], ARM64 [15], and MIPS64 [53] have 11, 4, and 8 unused bits for software use, respectively. In Figure 6, LODIC in 64 bit x86 architecture is using 7-bit embedded local counter at bits [58:52]. Other CPU technologies also utilize the unused PTE bits (ARM ASID [9], Intel MKTME [39], and AMD SEV [13]). CPU's with these technologies may leave a fewer bits available for the embedded counter. Regardless of the number of unused bits left in PTE, the design of LODIC remains unchanged. As the local counter embedded at PTE consists of a fewer bits, LODIC will more frequently access the global counter since the embedded counter tends to overflow more frequently. However, in our experimental experience, LODIC rarely accesses the global counter because the local counter overflows infrequently.

The counter embedding has two advantages: First, it does not require any additional memory for the local counter. With counter embedding, LODIC's space overhead is $O(N)$ as presented in Table 1. In addition, it minimizes the kernel code changes in implementing the local counter. It does not bring any new data structures nor does it modify any for local counter implementation.

LODIC updates the embedded local counter using an atomic `compare-and-swap` (`CAS`) instruction. We decided to use the atomic instruction on purpose. We can simply guarantee the correctness of the local counter update even when the reference split happens without relying on any additional mechanisms such as epoch based counter management in RefCache [23] and anchor counter in PayGo [40]. More importantly, using the atomic instruction does not slow down the common case operation because *non-contending* atomic operations are as cheap as non-atomic operations in modern processor architectures [63]. It is often used in designing the highly optimized lock-free data structures [54]. With 7-bit embedded counter, the embedded local counter overflows when the number of concurrent threads accessing a page is greater than 2^7, which is extremely rare. To handle the overflow, LODIC falls back to using the global counter to represent the total number of references [61, 65].

4.4 Reverse Mapping from a Physical Page Frame to PTE

The performance and scalability of LODIC critically relies upon the efficiency of the reverse mapping (steps between ❷ and ❹ in Figure 5). If the reverse mapping is slow or becomes a scalability bottleneck in concurrent accesses, it degrades performance significantly. Our evaluation results in §5.2.3 show that the legacy reverse mapping which is used in `rmap()` [24] can degrade the performance by 3 times. We first describe how the existing kernel handles the reverse mapping from a page cache to process virtual address space (§4.4.1) then propose our approach to accelerate the reverse mapping for LODIC (§4.4.2).

4.4.1 Process Space vs. File Space

Process Space: Virtual Address Space. A virtual address space of a process is composed of a set of virtual memory segments. We call this abstraction as *process space*. A virtual memory segment is a virtually contiguous memory region of a process having the same permission and file backend. It is represented by a start virtual address, the size, and backed file and offset (see Figure 5f). A virtual memory segment corresponds to `struct vm_area_struct` in Linux and `struct vm_space` in FreeBSD, respectively. In this paper, we use the segment and the virtual memory area (or VMA), interchangeably. OS kernel uses the process-space structure to check if a given virtual memory access is legal. For example, upon a page fault, OS first checks if the faulting address is legitimate in the process space then handles the fault (*e.g.*, allocating physical page or copy-on-write, etc).

Existing OSes adopt difference data structures for process address space: *e.g.*, red-black tree (Linux) [2], splay tree (FreeBSD) [36], and AVL tree (Windows) [62]. The leaf nodes correspond to the segments. The leaf nodes form a linked list. These data structures have pros and cons. Red-black tree and AVL tree guarantee the worst-case time complexity to $O(\log n)$ for search and update (insert and delete). Splay tree provides amortized time complexity to $O(\log n)$, *i.e.*, the time complexity for k operations is $O(k \log n)$, but the worst-case time complexity of a single operation is worse than the red-black tree or AVL tree. All three OSes rely only on a single lock to protect the process-space structure from race conditions. A number of techniques have been proposed to make the process-space data structures friendly to the concurrent updates [23, 32, 66] to improve the scalability of virtual memory subsystem.

File Space: Page Cache and Reverse Map For each file, the kernel maintains the information associated with a set of the physical pages that cache its file blocks and a set of virtual memory segments that map its file regions if the file (or region of it) is memory mapped. We call this abstraction a *file space*. It corresponds to `struct address_space`[1] in Linux. File space encapsulates two mapping information in it; (i) a mapping from a tuple of (inode, file offset) to a physical page which caches the file block (Figure 5d) and (ii) a mapping from the physical page to the associated page table if the file block is memory mapped. The latter mapping is called *reverse mapping* since it is used to map the physical page to the associated virtual address or the associated page table equivalently. Linux uses a radix tree to organize the set of physical pages in a file space. When the kernel needs to locate the physical page for the given file block (*e.g.*, `read()` and `write()` system calls Figure 5b), the kernel looks up the radix tree of the file space with the given file offset (*i.e.*, `struct address_space` in Figure 5d). In addition, a file can be memory mapped to one or more processes. If the multiple processes map a given file region, the physical page that holds the memory mapped file block can be associated with the multiple process address spaces. To maintain the mappings from the physical page to multiple process address spaces, the kernel maintains the reverse mapping in per-file basis (Figure 5d) in the file space. One example of using the reverse mapping is for page reclamation (Figure 5b). Before the eviction, the kernel should make sure that the page is not being referenced by any threads. The associated page table entries should be invalidated after the page is reclaimed. To quickly invalidate the page table entries of the physical page, the kernel scans the reverse mapping in the file space.

When we map a file region to the process address space, we associate the page frames in a given file region not only to the file space but also to the process space.

[1]The name `struct address_space` is, we believe, deeply mis-named.

4.4.2 Accelerating the Reverse Mapping for LODIC

The reverse mapping is the linkage between two spaces: process space and file space. As illustrated in Figure 5c, a reverse mapping (in Linux) maintains a mapping from a file offset to a list of virtual memory segments, which maps the file region. The main usage of the reverse mapping is page reclamation, which happens in every ten of seconds. When a file region is mapped by multiple processes, the kernel needs to examine all segments that map a given file region each of which is associated with different process (❸ in Figure 5c) and then to look up the process space to locate the proper virtual memory segment (❸ in Figure 5e). Unlike the page reclamation, where the reverse mapping is not in the critical path, LODIC needs to perform the reverse mapping in the critical path at every page cache access to locate the local counter. The existing reverse mapping relying on the file-space data structure does not meet the required level of performance and scalability for LODIC.

LODIC*'s approach of mapping a file region to the process address enables a new optimization for fast look up of the reverse mapping, which is impossible without mapping the file region.* Since the page cache access (*i.e.*, processing `read()` and `write()` system calls) is executed in the context of a calling user-space process context, LODIC can directly exploit the process-space data structure (*i.e.*, `current`$\rightarrow$`mm` in Linux) for efficient look up of the reverse mapping (❸ in Figure 5f).

Leveraging the process-space data structure for the reverse mapping has two important advantages: First, the number of segment to examine is independent of the degree of sharing. Without mapping the file region and leveraging the process-space data structure, LODIC has to traverse the list of the virtual memory segments in the file-space to find out the corresponding virtual address argument. Hence the reverse mapping look up cost linearly increases as the file region is shared by more processes. Next, we can further optimize the reverse mapping look up operation by leveraging the OS's virtual memory layout, which we explain next.

Application	FxMark	RocksDB	NGINX
Number of segments	20	85	62
Position in the segment list	5	4	5

Table 2: Total number of segments in the applications and the position of a memory-mapped file segment in the segment list.

A process virtual address can have tens or hundreds of virtual memory segments. Table 2 illustrates the number of segments in a few popular applications. In reverse mapping, i.e. in locating the page table entry for a given physical page, LODIC exploits the way in which the Linux kernel maps the region of the file onto the process virtual address space (`arch_get_unmapped_area_topdown()`). Exploiting the very nature of kernel's memory mapping, LODIC can quickly locate the target virtual memory segment regardless of the total number of segments in the process virtual address space. Linux maps the shared libraries and the memory-mapped files to the virtual address space as follows; when the process starts, the kernel defines the base address in the process virtual address space where the memory mapping starts, `mmap_base`. Starting from the base address, the kernel scans the process virtual address space towards the low end of the virtual address space and looks for the free virtual address region that can accommodate the given size of the file segment (or the shared library). When it finds the free virtual address region that can accommodate the given file segment, it reserves the region of the virtual address space, creates the virtual memory segment object (`struct vma`), and maps the created virtual memory segment to the allocated virtual address region. Due to this mapping mechanism, the virtual memory segment of the memory mapped file is likely to be located just below the stack and the `vDSO` [8]. Table 2 shows the position of the segment for the memory mapped file in the virtual segment list of FxMark, RocksDB and NGINX, respectively. FxMark has twenty virtual memory segments in its virtual address space. From `mmap_base`, the segment for the memory mapped file corresponds to the fifth one in the list of the virtual memory segments.

To leverage such layout property of memory mapped file regions, LODIC *scans the segment list of process-space starting from* `mmap_base` *in reverse direction.* By scanning the virtual memory segments in reverse direction (*i.e.*, from high to low virtual addresses), LODIC can quickly find the segment associated with a given page frame (❸ in Figure 5). Specifically, LODIC iterates the list of virtual memory segments (`mm_struct`$\rightarrow$`mmap`) in a reverse direction. It searches a segment that harbors a given page frame. For each segment, the kernel checks if the segment is associated with the same file as the given physical page. If they match, LODIC compares the file offset of the physical page and the file offset of the segment, and check if the page frame belongs to the segment. If we find the associated segment, then we obtain the virtual page number of the given physical page based upon the start virtual address of the segment and the offset of the given page frame within the segment. Once the segment is located and LODIC gets the virtual address of the mapped page, LODIC then walks the page table to locate the embedded local counter (❹❺ in in Figure 5). In summary, LODIC can locate the page table entry within nearly constant amount of time independent of the degree of sharing and independent of the number of segments in the virtual address space.

4.5 LODIC Operations

We summarize how LODIC performs each operation with the examples in Figure 5.

Reference Operation. When a page frame in a page cache is accessed (❶❷ in Figure 5), the kernel first performs a reference operation, increasing its reference counter. If the page is memory mapped and it is shared by more than four processes (*i.e.*, `_mapcount >=4`), LODIC first looks up the corresponding segment by scanning process's list of virtual memory segments in a reverse order using the file and offset as a key (❸). After finding the segment, LODIC calculates the virtual address of the page using the start virtual address of the segment and offset of the mapping. With the virtual address, LODIC walks the page table to spot the page's local counter (❹❺) and increases the local counter. LODIC relies on atomic `CAS` to update the page table entry to increase local counter value. If the page is not shared or it is shared by less than four processes, LODIC falls back to increasing page's global counter (`_refcounter` in `struct page`).

Unreference Operation. After finishing the access of the page, the kernel performs an unreference operation, decreasing the page's reference counter. The procedure is similar to the reference operation but LODIC handles the situation when the counter changes between the reference and unreference operations. LODIC first tries to decrements the page's local counter if the page is memory mapped. The local counter can be zero if the number of sharing processes is increased beyond four after the reference operation. To handle such cases, LODIC decrements the global counter when the local counter is zero. Like the reference operation, LODIC relies on atomic `CAS` to decrement the counter. Since LODIC relies on atomic operation in updating the counter, LODIC does not need any other special mechanism to handle the reference split.

Query Operation. When the kernel reclaims the clean pages in the page cache to secure more free memory, it first performs a query on the page to check if the page is being accessed or not (① in Figure 5). LODIC first looks up the reverse mapping of a file with the file-backed page's offset and gets the list of virtual memory segments associated with the page (②). For each virtual memory segment (③), LODIC calculates page's virtual address (④). With the page's virtual address, LODIC locates the page's local counter embedded at the PTE (⑤⑥). If any of the page's global counter and one of the local counters are not zero, LODIC returns the results as non-zero. Otherwise, it return zero so that the kernel can safely reclaim the page. For correctness of a query operation, the local counters and the global counter should remain unchanged while the query is in-flight. To ensure this, we use the same mechanism proposed in [40]; we define a flag for each physical page to denote if there is a query in-flight. Before the query starts, the kernel sets this flag. If this flag is set, the counter update operation blocks. We use `atomic_write` to set and to reset the flag [50].

Latency Analysis of Query Operation. LODIC can return the query operation as soon as it encounters the non-zero local or the global counters since all local counters are guaranteed to be non-negative. This is not possible in some distributed reference counting schemes, such as sloppy counter [17] and RefCache [23]. In these schemes, the local counter can become negative and the positive local counter does not guarantee that the counter's true value is positive.

To quantify the performance benefit, we analyze the number of local counter iterations for a query operation. Suppose that there are N counters. We denote the probability that the i-th counter is zero as $P(c_i = 0)$, where c_i is i-th counter value. When X is the number of local counters that the kernel has to examine until encountering a non-zero logical counter, $P(X = k) = p^{k-1}(1-p)$, where p is $P(c_i = 0)$. Therefore, the expected number of local counter traversal is as follows:

$$E(X_N) = \sum_{i=1}^{N} i \cdot p^{i-1}(1-p) = \frac{1-p^N}{1-p} \qquad (1)$$

$E(X_N)$ grows slowly with N. When $N = 120$ and $p = 0.5$, we obtain $E(X_N) \approx 2$; the kernel examines less than two counters on the average until it encounters a non-zero counter.

5 Evaluation

We implemented LODIC on Linux v4.11.6. We examine the query latency, the memory pressure, and the application performance in five different reference counting schemes: Global Counter (Vanilla Linux), No Counter (NCount), RefCache [23], PayGo [40] and LODIC. We use the microbenchmark FxMark [51] and two data intensive applications – RocksDB key-value store [34] and NGINIX web server [60]. Scalable file block read plays an important role in these applications. We choose RocksDB and NGINX in our evaluation because the multi-core servers are widely used in cloud environments, and a key-value store and a web server are the two most popular workloads in the cloud environment. No Counter (NCount) is a null counter that actually does not perform reference counting. This is to simulate the ideal scalable counter. The server has 120 CPU-cores (Intel Xeon E7-8870 v2 processors, 8 sockets and 15 cores per socket) and 780GB DDR3 DRAM. In the rest of this section, we first present how LODIC affects the performance of real-world applications (§5.1) then analyze how our design decisions affect the performance (§5.2).

5.1 Real World Applications

RocksDB. In key-value store, a number of processes can read the same database file, simultaneously. Especially in LSM-tree based key-value store, the run at level 0 can be read by a large number of users simultaneously and the efficiency of the distributed counting scheme can play a vital role in its performance behavior. We use modified `db_bench` [33] so that the multiple processes access the shared database. We perform sequential scan and search on the random key with a key size of 16 byte and a value size of 100 bytes on a working set of 1,000 records. With LODIC, the performance of RocksDB is as good as NCount, RefCache and PayGo (Figure 7). LODIC brings 23% performance improvement against the global counter under 100 processes.

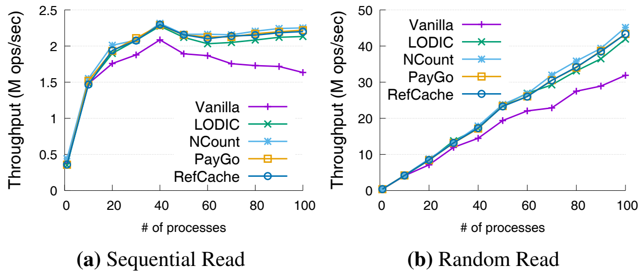


Figure 7: Performance of RocksDB key-value store.

NGINX. We examine the web server performance under the different distributed counting schemes. We vary the number of clients accessing the same web page using `wrk` benchmark [69]. We vary the number of clients from one to fifty. All clients access `index.html` (612 byte). Figure 8 illustrates the results. The LODIC, NCount, RefCache and PayGo yield the similar performance, showing 2.5× performance improvement against vanilla Linux. Our evaluation results confirm that LODIC performs nearly as good as the ideal contentionless counting scheme (NCount) does.

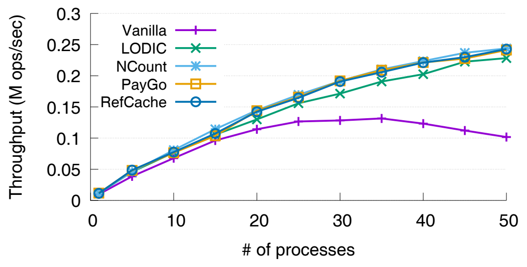


Figure 8: Performance of NGINX web server.

5.2 Analysis on LODIC Design

5.2.1 Memory Pressure

We compute the memory pressure for the RefCache, PayGo and LODIC (Figure 9). In RefCache and PayGo, the memory for distributed counting consists of the memory for the per-core hash tables and the memory for the per-page metadata (Table 1). Refcache and PayGo allocate 64 KB and 256 KB for per-core hash table (4096 entries), respectively. RefCache and PayGo need 16 bytes for each page frame to store the additional information, *e.g.*, lock, pointer, and epoch number. For our server with 780 GB memory (195 M page frames), the additional memory required for all physical page frames corresponds 3.1 GB (195 M×16 byte). Total amount of memory required for the distributed counting in RefCache and in PayGo correspond to 3.12 GB and 3.3 GB, respectively.

In LODIC, the total amount of memory for all local counters corresponds to only the page table pages that are required to map the file (or part of the file). For 4KB, 100MB and 1GB files, the total amount of memory required for LODIC corresponds to 480KB, 24MB, and 240MB, respectively, when the file is shared by 120 processes. Unlike PayGO and RefCache, LODIC does not have the overhead of handling the hash collision and the cache miss that arise in the counter cache based distributed counting scheme. According to our experiment, as the hash collision increases, the performance of these schemes can drop by 40% from 186 Mops to 70 Mops for 4 KB read.

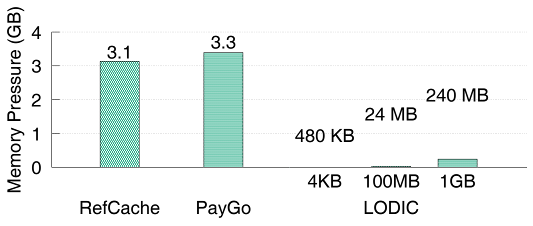


Figure 9: Memory consumption for RefCache, PayGo, and LODIC on a 120-core machine (the degree of sharing in LODIC is 120-core).

5.2.2 Query Overhead

We measure the latency of counter query under two scenarios: (1) the latency to reclaim all page cache entries for a given file (Figure 10) and (2) the latency to check if a given page frame can be reclaimed (Figure 11).

We examine the latency to reclaim all page cache entries of the 1 GB file under vanilla Linux (Vanilla), RefCache, PayGo, and LODIC. We use `fadvise` to trigger the page reclamation. In LODIC, we vary the fraction of the mapped pages (*i.e.*, hot page ratio) in the file; 10%, 20% and 100%. Figure 10 illustrates the results. 'Vanilla' renders the best case. Here, the kernel examines only the single global counter for each page frame to see if it can be reclaimed. The kernel repeats this process for all page frames for the given file. In per-core distributed counting schemes, the latency of reclaiming all page frames is problematic. It examines all per-core local counters for all pages in the file. The latency of `fadvise` corresponds to over 500 msec regardless of the number of processes that share the file. For the page reclamation, the LODIC uses the red-black tree (❸ in Figure 5c) stored in `i_mmap` field of `struct address_space` like the Vanilla. In LODIC, the number of the local counters for a page frame is proportional to the number of the processes that map the file. For unmapped page, there is no local counter. The query latency of LODIC is longer than the vanilla Linux. However, LODIC successfully reduces the query latency properly incorporating the actual degree of sharing. When 10% of the entire file is hot region (100 MB is mapped), LODIC renders less than 1/4 of the `fadvise` latency of the per-core distributed counter when the file is shared by 45 processes. However, in the worst case (*i.e.*, the entire file is hot), which we believe unrealistic, the query latency of LODIC exceeds the query latency of the per-core distributed counting scheme when the number of processes is greater than 40. This is due to the overhead of reverse mapping in LODIC.

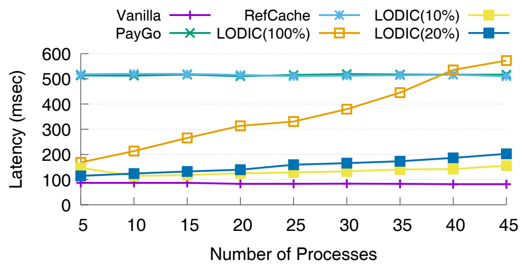


Figure 10: Query latency: reclaiming all page frames (`fadvise`).

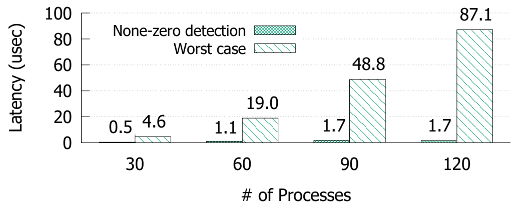


Figure 11: Query latency: reclaiming a page frame.

For counter query, we also measure the latency to determine if a given page is reclaimable. In LODIC, the kernel returns as soon as it encounters the non-zero local counter. As Figure 11 shows, LODIC examines only one or two local counters in most cases until it encounters non-zero local counter. The result well matches Equation 1. In the worst case, LODIC needs to scan all logical counters till it finds the non-zero local counter. In this case, the latency linearly increases with the degree of sharing.

5.2.3 Reverse Mapping Overhead

Performance and Scalability. We first examine the efficiency of the file-space based reverse mapping (❸ in Figure 5) and the process-space based reverse mapping (❸ in Figure 5). Figure 12 shows the time to perform the reverse mapping under two different reverse mapping schemes. The reverse mapping latency reduces to 1/20 when we use the process-space based reverse mapping instead of the file-space based reverse mapping.

Next, we examine the performance impact of the reverse mapping. We compare the latency to read 4 KB block from the different regions of a file: (1) from plain file in Vanilla Linux, (2) from the unmapped region of a file in LODIC-enabled Linux, and (3) from the memory-mapped region of the file in LODIC-enabled Linux. Figure 13 illustrates the result. In LODIC-enabled Linux, reading the memory-mapped file block renders 20% longer latency than reading the unmapped file block. In LODIC, the filesystem checks the map count of the file block before it performs reverse mapping. It performs the reverse mapping only when the map count is non-zero.

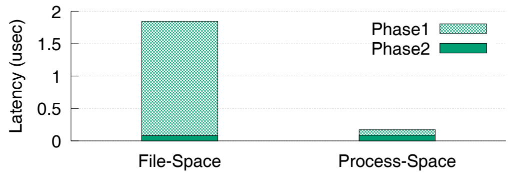


Figure 12: Latency break-down: file-space based reverse mapping vs. process-space based reverse mapping

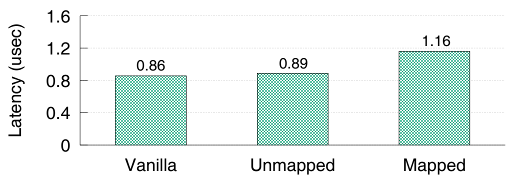


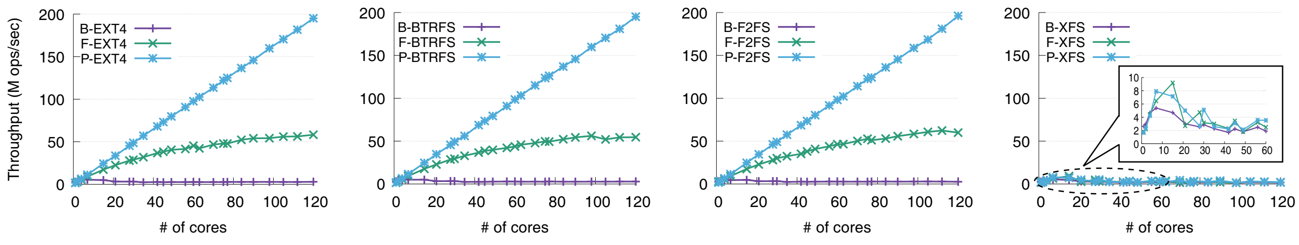
Figure 13: Read latency with various reverse mapping schemes.

Lastly, we examine the performance of reading a shared file block under different reference counting schemes: (1) process-space based reverse mapping (LODIC), (2) file-space based reverse mapping (LODIC) and (3) vanilla Linux. We use DRBH workload of `FxMark` [51]. Figure 14 presents the results. Using the process-space based reverse mapping, LODIC achieves 65× performance against vanilla Linux under 120 CPU cores. The benchmark performance successfully scales linearly with respect to the increase in the number of processes (*i.e.*, the number of active cores). On the other hand, the file-space based reverse mapping fails to scale due to its inefficient access path shown at ❸ in Figure 5.

Different from the rests, XFS barely yields any performance gain under LODIC. We find that reference counter in per-inode rw-semaphore in XFS is the root cause of the scalability bottleneck here (Figure 14d). The performance bottleneck caused by this problematic inode reference counter has been out-shadowed by the severe scalability overhead of the reference counter for physical page. As LODIC effectively resolves the scalability issue in the reference counter of the physical page, the inefficiency of inode reference counter in XFS comes to the surface and prohibits XFS from scaling well. LODIC, however, is not entirely ineffective; LODIC brings as much as 2× performance in XFS (Figure 14d).

Interference with `mmap`. The `mmap` and `munmap` operation establish an exclusive lock on the process-space and the file-space structure to insert and delete the virtual memory segment. We examine how the `mmap` and `munmap` operations interfere with the performance of LODIC. There are 60 processes that read the shared file block. We create a background process that calls `mmap`, sleeps for one second and does `munmap`. We measure the performance of the shared file read with and without the background processes, respectively, and compute the performance ratio between the two.

Figure 15 shows the performance degradation varying the number of background processes. The performance of process-space based reverse mapping is barely affected by the background `mmap/munmap`. On the other hand, the performance of the file-space based reverse-mapping collapses to 1/100 when there are five or more background processes.



(a) EXT4 **(b)** BTRFS **(c)** F2FS **(d)** XFS

Figure 14: FxMark (DRBH): Baseline ('B'), File-based reverse mapping ('F'), Process-based reverse mapping ('P').

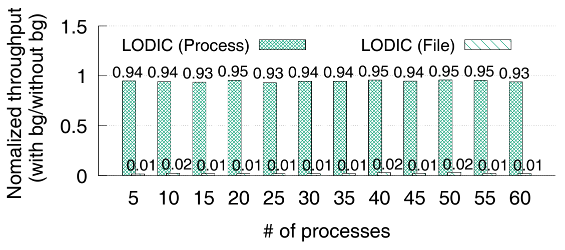


Figure 15: `mmap()` interference on the reverse mapping.

5.2.4 Counter Contention on the Local Counter

We examine the theoretical worst-case performance of LODIC; when the threads compete for updating the local counter. We create multiple threads in a process and have them access the same file block. We vary the number of threads per process to vary the contention on the local counters. We maintain the total number of threads in the system to be 120. Table 3 illustrates the performance. As the number of threads per process increases, the performance decreases. This is because PTE contention occurs when many threads of a process access the same file page at the same time. However, compared to the case using the global counter, LODIC is meaningful in that the contention is localized within a process because a PTE is shared by threads only within a process. We believe that in practice this situation can be avoided via properly limiting the number of threads per process. For example, it is common for server programs to provide tuning knobs to change the number of threads per process [56–58].

# of Threads	1	2	4	8	16
Throughput (M ops/sec)	164.16	58.87	32.54	19.61	8.79

Table 3: Effect of the contention on the local counters

6 Related Work

There were a number of sophisticated designs for the scalable reader-write lock. For the scalability, a few works exploit the memory barrier and the atomic instruction in the reader [44, 64]. Big-reader lock (`brlock`) uses an array of reader flags to avoid using the memory barrier [4]. Passive reader-write lock (prwlock) uses version-based consensus protocol instead of the expensive mutex protected flags to avoid using the memory barrier [46]. A new reader-writer lock called `percpu-rwlock` in Linux [6] is known to work when the writers are rare. Bravo [30] uses the global hash table to address the memory pressure issue of the per-core reference counters [6, 17, 26, 27, 31] for rw-semaphore. SHFLLOCK [42] leverages a waiting thread to achieve high scalability in NUMA machines without introducing the additional memory consumption for NUMA-local locks. Hydralist separates the search layer and the data node for the manycore scalability of the index structure [49].

A few works adopt the actual degree of sharing to ease the contention [11,29,55]. Nerula et al. [55] detect the access conflict at run-time and distribute the conflict records to per-core basis. It cannot be applied to the generic workload. Dashti et al. [29] use the hardware profiling to detect the degree of sharing. It fails to scale when the sharing degree is high.

7 Conclusion

The per-core based distributed counting scheme has reached its limit due to the increase in the number of cores and the memory size in the recent computing system. In this work, we view that the counter contention is driven by the contention among the processes. This process-centric view enables us to devise a new counting scheme that can naturally incorporate the degree of sharing, the population and the reference brevity characteristics of the physical page frame. Via defining the local counter in per-process basis, LODIC is successful in striking the balance among the three factors of the reference counter: memory pressure, the counter query latency, and counter update performance. We show that software overhead of the proposed logical distributed counting is insignificant and hardly visible from outside.

Acknowledgements We would like to thank our shepherd James Bottomley and the anonymous reviewers for their valuable feedback. We also would like to thank Yeonjin Noh and Hyeongjun Kim for their help on the earlier draft of this work. This work was supported by IITP, Korea (grant No. 2018-0-00549 and No. 2014-3-00035) and NRF, Korea (grant No. NRF-2020R1A2C3008525).

References

[1] 10M Concurrent Websockets. http://goroutines.com/10m.

[2] Linux rbtree. https://www.kernel.org/doc/Documentation/rbtree.txt.

[3] Number of dentry objects. https://serverfault.com/questions/561350/unusually-high-dentry-cache-usage.

[4] Big reader locks. https://lwn.net/Articles/378911/, 2010.

[5] The search for fast, scalable counters. http://lwn.net/Articles/170003/, 2010.

[6] percpu_rwlock: Implement the core design of per-cpu reader-writer locks. https://lore.kernel.org/patchwork/patch/360375/, 2013.

[7] Dell poweredge specification. http://media.zones.com/images/pdf/Dell_PowerEdge_R920.pdf, 2014.

[8] vdso. https://https://lwn.net/Articles/615809/, 2014.

[9] Address space identifier. https://lwn.net/Articles/699820/, 2016.

[10] Sgi ultraviolet 3000. https://www.sgi.com/products/servers/uv/uv_3000_30.html, 2016.

[11] Umut A Acar, Naama Ben-David, and Mike Rainey. Contention in structured concurrency: Provably efficient dynamic non-zero indicators for nested parallelism. *ACM SIGPLAN Notices*, 52(8):75–88, 2017.

[12] AMD. Amd64 architecture programmer's manual volume 2: System programming.

[13] AMD. Secure encrypted virtualization. https://developer.amd.com/sev/.

[14] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W Wisniewski, Jimi Xenidis, Michael Stumm, et al. Experience distributing objects in an smmp os. *Transactions on Computer Systems (TOCS)*, 25(3):6, 2007.

[15] ARM. Arm® architecture reference manual armv8, for armv8-a architecture profile.

[16] Srivatsa S Bhat, Rasha Eqbal, Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proc. of ACM SOSP*, 2017.

[17] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *Proc. of USENIX OSDI*, 2010.

[18] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Oplog: a library for scaling update-heavy data structures. *Technical Report MIT-CSAIL-TR-2014-019*, 2014.

[19] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proc. of IEEE INFOCOM*, 1999.

[20] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *Proc. of USENIX FAST*, 2020.

[21] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, 29(3):267–290, 1999.

[22] Wonje Choi, Karthi Duraisamy, Ryan Gary Kim, Janardhan Rao Doppa, Partha Pratim Pande, Radu Marculescu, and Diana Marculescu. Hybrid network-on-chip architectures for accelerating deep learning kernels on heterogeneous manycore platforms. In *Proc. of IEEE CASES*, 2016.

[23] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proc. of ACM EUROSYS*, 2013.

[24] Corbet. The object-based reverse-mapping vm. https://lwn.net/Articles/23732/, Feb 2003.

[25] Corbet. Kswapd and high-order allocations. https://lwn.net/Articles/101230/, Sep 2004.

[26] J Corbet. Per-cpu reference counts. https://lwn.net/Articles/557478/, 2013.

[27] J Corbet. The search for fast, scalable counters. https://lwn.net/Articles/170003/, 2016.

[28] Alvaro Corral, Gemma Boleda, and Ramon Ferrer-i Cancho. Zipf's law for word frequencies: Word forms versus lemmas in long texts. *PLOS ONE*, 10(7), 2015.

[29] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGARCH Computer Architecture News*, 41(1):381–394, 2013.

[30] Dave Dice and Alex Kogan. Bravo: Biased locking for reader-writer locks. In *Proc. of USENIX ATC*, 2019.

[31] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: Scalable nonzero indicators. In *Proc. of ACM PODC*, 2007.

[32] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the BSDCan Conference*, 2006.

[33] Facebook. db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[34] Facebook. RocksDB. http://rocksdb.org.

[35] Md Hasanul Ferdaus, Manzur Murshed, Rodrigo N Calheiros, and Rajkumar Buyya. Virtual machine consolidation in cloud data centers using aco metaheuristic. In *Proc. of EuroPar*, 2014.

[36] FreeBSD. https://www.freebsd.org.

[37] HPE. Hpe integrity superdome x. https://www.hpe.com/us/en/servers/superdome.html, 2016.

[38] HPE. Hpe the machine. http://www.labs.hpe.com/research/themachine/, 2016.

[39] Intel. Multi-key total memory encryption. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf.

[40] Seokyong Jung, Jongbin Kim, Minsoo Ryu, Sooyong Kang, and Hyungsoo Jung. Pay migration tax to homeland: Anchor-based scalable reference counting for multicores. In *Proc. of USENIX FAST*, 2019.

[41] Karim Kanoun, Martino Ruggiero, David Atienza, and Mihaela Van Der Schaar. Low power and scalable manycore architecture for big-data stream computing. In *Proc. of IEEE ISVLSI*, 2014.

[42] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 586–599. ACM, 2019.

[43] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proc. of Linux Symposium*, 2007.

[44] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Proc. of ACM SPAA*, 2009.

[45] linux. Proper locking under a preemptible kernel: Keeping kernel code preempt-safe. https://www.kernel.org/doc/Documentation/preempt-locking.txt.

[46] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proc. of USENIX ATC*, 2014.

[47] lwn. Linux generic irq handling. https://static.lwn.net/kerneldoc/core-api/genericirq.html.

[48] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proc. of ACM EUROSYS*, 2017.

[49] Ajit Mathew and Changwoo Min. Hydralist: A scalable in-memory index using asynchronous updates and partial replication. *Proc. VLDB Endow.*, 13(9):1332–1345, 2020.

[50] Paul E McKenney. Overview of linux-kernel reference counting, 2007.

[51] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proc. of USENIX ATC*, 2016.

[52] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In William J. Bolosky and Jason Flinn, editors, *Proc. of USENIX FAST*, page 12. USENIX Association, 2012.

[53] MIPS. Mips® architecture for programmers volume iii: Mips64® / micromips64™ privileged resource architecture.

[54] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 103–112. ACM, 2013.

[55] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *Proc. of USENIX OSDI*, 2014.

[56] NGINX. Apache performance tuning: Mpm directives. https://www.liquidweb.com/kb/apache-performance-tuning-mpm-directives/.

[57] NGINX. Thread pools in nginx boost performance 9x! https://www.nginx.com/blog/thread-pools-boost-performance-9x/.

[58] NGINX. Tuning nginx for performance. https://www.nginx.com/blog/tuning-nginx/.

[59] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514, 2017.

[60] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[61] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *ACM SIGPLAN Notices*, 41(10):263–272, 2006.

[62] M.E. Russinovich, D.A. Solomon, and A. Ionescu. *Windows Internals*. Pearson Education, 2012.

[63] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architectures and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, pages 445–456. IEEE Computer Society, 2015.

[64] Michael L Scott and John M Mellor-Crummey. Synchronization without contention. In *Proc. of ASPLOS*, 1991.

[65] Rifat Shahriyar, Stephen M Blackburn, and Daniel Frampton. Down for the count? getting reference counting back in the ring. In *Proceedings of the 2012 international symposium on Memory Management*, pages 73–84, 2012.

[66] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. *ACM SIGPLAN Notices*, 46(11):79–88, 2011.

[67] Rik Van Riel. Page replacement in linux 2.4 memory management. In *Proc. of USENIX ATC*, 2001.

[68] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *Proc. of USENIX ATC*, 2009.

[69] Modern http benchmarking tool, 2013. https://github.com/wg/wrk.