Pay Migration Tax to Homeland: Anchor-based Scalable Reference Counting for Multicores

Seokyong Jung, Jongbin Kim, Minsoo Ryu, Sooyong Kang, and Hyungsoo Jung,
Hanyang University

https://www.usenix.org/conference/fast19/presentation/jung

This paper is included in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

Pay Migration Tax to Homeland: Anchor-based Scalable Reference Counting for Multicores

Seokyong Jung, Jongbin Kim, Minsoo Ryu, Sooyong Kang, Hyungsoo Jung*
Hanyang University
{*syjung, jongbinkim, msryu, sykang, hyungsoo.jung*}*@hanyang.ac.kr*

Abstract

The operating system community has been combating scalability bottlenecks for the past decade with victories for all the then-new multicore hardware. File systems, however, are in the midst of turmoil yet. One of the culprits behind performance degradation is reference counting widely used for managing data and metadata, and scalability is badly impacted under load with little or no logical contention, where the capability is desperately needed. To address this, we propose PAYGO, a reference counting technique that combines per-core hash of local reference counters with an anchor counter. PAYGO imposes the restriction that decrement must be performed on the original local counter where the act of increment has occurred so that reclaiming zero-valued local counters can be done immediately. To this end, we enforce migrated processes running on different cores to update the anchor counter associated with the original local counter. We implemented PAYGO in the Linux page cache, and so our implementation is transparent to the file system. Experimental evaluation with underlying file systems (i.e., `ext4`, `F2FS`, `btrfs`, and `XFS`) demonstrated that PAYGO scales file systems better than other state-of-the-art techniques.

1 Introduction

Reference counting is a general technique, originally introduced by Collins [8] almost six decades ago, to determine the liveness of an object for automatic storage reclamation. Since the early version of UNIX kernel used reference counting to manage data (e.g., page cache) and metadata (e.g., inode), reference counting has gained widespread acceptance in the systems community thereafter, e.g., file systems, HBase [21], RocksDB [22] and MariaDB [23].

However, a recent study by Min et al. [17] found that reference counters, among many other factors, in modern file systems are not scalable, thus leading file systems to suffer performance degradation on multicore hardware, even with applications with little or no logical contention. For example, the traditional way of referencing (let us call it 'traditional reference counter'), which is currently being used in the Linux operating system for page cache, uses a single shared atomic counter. By using atomic operations, an object can be safely referenced even when multiple threads update at the same time. The traditional reference counters, however, degrade the performance of applications on multicores due to excessive atomic operations on a shared counter.

In order to be a good reference counter for concurrent applications, there are important properties to consider; 1) updates on reference counters must be scalable, 2) reading an accurate (zero or positive) counter value should be cheap, 3) reference counters should be space-efficient and 4) all these should be guaranteed without incurring extra delay to manage reference counters. We denote overheads required for achieving the four properties as *counting overhead*, *query overhead*, *space overhead*, and *time overhead*, respectively.

Counting overhead. The counting overhead, which is the most important property for scalable counting, indicates the cost of updating (REF/UNREF) a reference counter itself when there is a heavy load on referencing an object. Since the counting overhead is a crucial hurdle for achieving scalability, all reference counting techniques strive hard to eliminate it first. The traditional reference counter which uses a single shared counter has the highest counting overhead due to the hardware-based synchronization bottleneck [13].

Query overhead. The query overhead measures the cost of query operation which checks if the reference counter of an object is zero and so we can safely reclaim the object from memory. The traditional technique can detect zero by reading a single atomic counter.

Space overhead. The space overhead indicates how much space they use for reference counting. In terms of space overhead, the traditional reference counter is a (de facto) optimal technique since it does not require any other data structure than one atomic counter per object.

Time overhead. The time overhead represents any other delay than the counting overhead introduced by a reference

*Contact author and principal investigator

counting technique to manage all data structures it maintains. The traditional reference counter has minimal time overhead since it maintains only per-object atomic counters. However, some reference counting techniques that exploit distributed local reference caches have the synchronization overhead between local counters and a global counter. This synchronization plays two roles: 1) the global counter becomes ready (i.e., up-to-date) for zero detection and 2) the local counter, if it resides in hash, can be reclaimed. We generally denote this type of overhead as the *time overhead*.

Our analysis of prior proposals (§2.1) suggests that it is challenging to achieve all four properties, possibly due to tradeoffs between different properties. In this work, we propose *pay-as-you-go* (PAYGO[1]) reference counting that ensures scalable counting and space efficiency with negligible time overhead. Although based on a well-known per-core hash technique, PAYGO introduces a novel concept of an *anchor counter* that enables the immediate reclamation of local zero-valued counter entries, which is pivotal to reducing the forceful eviction of the conflicting hash entries when the number of objects accessed in a core becomes large. The instant reclamation is indeed a critical feature for escaping performance degradation that may otherwise occur due in large part to the heavy cost of operations for resolving collisions, including forceful evictions.

We implemented PAYGO and applied it to page cache in Linux, leaving existing file systems almost intact. To see the applicability of PAYGO to user applications, we also implemented new PAYGO system calls that can be used for reference counting user-level objects. Evaluation results with various underlying file systems (i.e., `ext4`, `F2FS`, `btrfs` and `XFS`) demonstrated that PAYGO shows substantial improvements against state-of-the-art reference counting techniques.

2 Related Work and Motivation

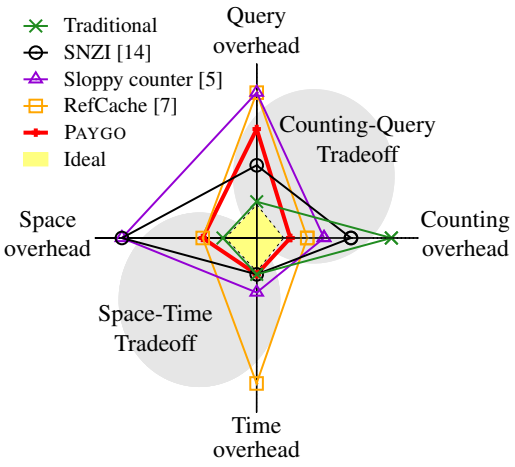
2.1 Related Work

There have been many proposals attempting to address some of the properties introduced in §1, and the techniques available so far utilize at least one of the following features:

Contention distribution. One of the major factors impeding the scalability of a reference counter is cache line contention: updating the same reference counter *atomically* by many threads results in high contention. SNZI [14] mitigates the contention by dispersing the reference counters at compile time. It manages distributed counters using a binary tree with a fixed-sized depth. While it shows better scalability than the traditional reference counter, it is still slower than other techniques due to the possible contention on a particular counter that changes frequently. However, it can perform zero detection in constant time by checking the indicator of the root node in the binary tree, although determining the global count value is impossible. Other techniques [1, 18] alleviate the contention problem by distributing reference counters according to the degree of contention at runtime, but they empirically judge the degree of contention and distribute reference counters so they cannot relieve the contention for a reference counter under general workloads where we can hardly predict the degree of contention. Carrefour [12] also distributes contention dynamically, but hardware profiling is required to verify memory traffic. Proposals in this category still rely on atomic instructions for updating reference counters, so they seldom achieve linear scalability.

Cache affinity. Another factor that hinders scalability of reference counters is cache misses. To reduce the cache misses, a local reference counter is used in a way that an object has per-core local counters and updates are made to the local counters *nonatomically*. The downside of this approach is the overhead needed for summing all local counter values to obtain the global count. To alleviate this side effect, there is a way to obtain the global count in advance and store it in the central counter [9, 5, 10], which incurs extra time overhead. Sloppy counter [9, 5] updates only the local counter if the updated value is less than a certain threshold. If the value exceeds the threshold, the local counter value is transferred to the central counter. The central counter is therefore an approximation of the global count. Before transferring the local counter value, it acquires the global lock for the central counter, which incurs extra time overhead. `percpu_ref` [10], a variant of the sloppy counter, implemented in Linux for managing memory objects in several device drivers, also primarily changes the local counter. The techniques exploiting cache affinity have the *counting-query tradeoff*: nonatomic updates on local reference counters earn good scalability in exchange for longer query time to read a global count by collecting the sum of local counters. They also have bad space efficiency due to the per-object, per-core local counters.

Per-core hash. To improve the space overhead of cache affinity-based techniques, recent years have seen attempts to use per-core hash of reference caches that would fulfill the main duty of reference counting with much less space overhead [7, 4, 3]. They can substantially reduce the space overhead by using per-core hash which keeps the local reference counters for only those objects in use. Techniques based on per-core hash inevitably face the problem of reclaiming a hash table entry whose local counter is zero (i.e., the corresponding object is not in use). Existing techniques address this using quiescent period-based synchronization which is widely used in Linux to reclaim objects, such as *read-copy-update* (RCU) [16]. The reference counting algorithm exploiting per-core hash with quiescent period-based synchronization cannot avoid the *space-time tradeoff*: they achieve better space efficiency in exchange for time overheads not only in synchronization between local and global counters but also in hash entry reclamation. RefCache [7], which is one of quiescent period-based techniques, manages its local counters in per-core hash, and the counter values are flushed



	Traditional*	SNZI	Sloppy counter	RefCache	PAYGO
Counting overhead	atomic ops.	atomic ops.‡	global lock	–	–
Space overhead†	$O(N)$	$O(M \cdot N)$	$O(M \cdot N)$	$O(M \cdot C + N)$	$O(M \cdot C + N)$
Query overhead§	$O(1)$	$O(1)$	$O(M)$	$O(1) + 2 \cdot epoch$	$O(M)$§§
Time overhead	–	–	every threshold	every epoch and collision	–

* A single atomic reference counter
† N: # of objects, M: # of local counters per object, C: # of hash entries
‡ SNZI recursively updates the counter of the parent node whenever the counter of the child node changes from 0 to 1 and vice versa.
§ Time to determine if the reference counter of a *single* object is zero or not
§§ PAYGO has practically less query overhead than Sloppy counter (§3.4).

Figure 1: A comparison of reference counting techniques under workloads accessing a shared counter.

into a central counter every epoch. OpLog [4] generalized RefCache's idea by using operation logs for the shared data structure with a local timestamp. In descending order of the timestamp, per-core logs are applied to the data structure.

2.2 Motivation

The design of PAYGO is motivated by two observations:

Observation 1. Our analysis of existing algorithms in §2.1 is summarized in Figure 1. Noticeable is the observation that attaining the counting scalability (i.e., low counting overhead) demands a sacrifice of two other properties due to the counting-query and space-time tradeoffs. By escaping those tradeoffs we can attain more good properties; for example, escaping the space-time tradeoff enables us to make both space and time overheads low while achieving scalability.

Observation 2. Another and more important observation is that the excessive time overhead may eventually incur severe performance degradation. As described in §2.1, techniques based on per-core hash sacrifice time overhead to reduce space overhead. The time overhead under consideration in such techniques is the overhead of reclaiming hash entries, when the number of objects accessed in a core becomes large so that frequent hash collisions occur and therefore forceful evictions for the conflicting hash entries need to be exercised to make room for newly accessed objects. The eviction of a hash entry needs to flush the local counter value to the global counter and therefore causes additional synchronization overhead between local and central counters.

For example, RefCache [7], designed for a new virtual memory system, is perfectly scalable when n threads are repeatedly performing `mmap`/`munmap` on a single shared physical page (see Figure 8 in [7]), since the forceful eviction of hash entries due to collisions seldom occurs and so may not be a serious design consideration in virtual memory systems. However, if we use RefCache in *page caches* under file system benchmarks that may access far more objects, frequent evictions that internally acquire/release object locks to protect the flush of local counter values to the central counters, may result in serious time overhead, leading to performance degradation. To confirm it, we conducted a preliminary experiment that measures the throughput of RefCache for page caches. The experimental environment is shown in §6.1, and we ran the FXMARK microbenchmark so that 96 threads read 64 bytes (or 4 KiB) on a shared file, with a sequential access pattern and varying the file size. Figure 2 shows the result that confirms our conjecture. The throughput decreases as the file size (i.e., the number of objects) gets bigger, due to increased hash collisions triggering more forceful evictions. We found that the hash collisions start slightly occurring from the point when the file size is 1 MiB (i.e., 256 objects) and become excessive as the file size increases. Hence, reclaiming garbage hash entries in a timely manner is critical for avoiding the performance degradation of per-core hash-based reference counting techniques.

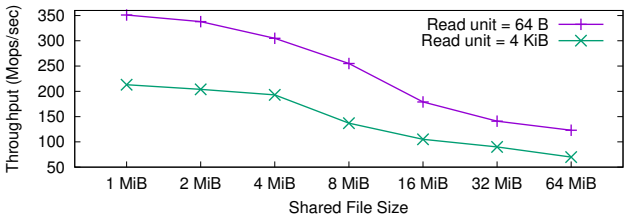


Figure 2: Performance of RefCache: hash table size = 4,096 entries (default size), `ext4` file system.

The above observations guide us to conclude that escaping the space-time tradeoff is crucial for scalable reference counting techniques. By escaping the space-time tradeoff, we can achieve true scalable counting keeping both space and time overheads low, which is our design goal of PAYGO whose comparative properties are depicted in Figure 1.

3 PAYGO

Of counting-query and space-time tradeoffs, we aim at escaping the *space-time tradeoff* while embracing the other. PAYGO achieves this by using a per-core hash-based reference cache with a new technique called *anchoring*. PAYGO is designed on the following assumptions; (i) objects are referenced and unreferenced by the same process and (ii) the lifetime of references is reasonably short not to put the static size of per core *hash* in jeopardy (see §3.5).

3.1 Design Overview

Design rationale. To escape the space-time tradeoff, per-core hash-based techniques must ensure the safety condition such that a local reference cache entry can be reclaimed immediately upon releasing all references to an object, all done without sacrificing other properties. In this respect, RefCache earned the counting scalability in exchange for the increased time overhead required for reclaiming obsolete cache entries. For addressing this issue, our main design rationale behind PAYGO lies in a simple goal; we make a local reference cache zero (i.e., ready to be reclaimed) right after all references are released. To this end, we enforce the restriction that a process, once referencing an object, must be *anchored* to the original reference cache to inform of any unreferencing to the object irrespective of which core the process runs on. For this purpose, PAYGO's per-core hash entry consists of a local counter and an *anchor* counter fields, and the sum of two represents a local count for an object initially accessed in that core.

Access rules. First, we establish ground rules in accessing a pair of local and anchor counters to preserve the correctness, that is '*never miscount*'. Access rules for `(UN)REF` are described as follows. For the `REF` operation, a process always accesses a local reference cache and updates the local counter field nonatomically. At this time, the process is logically anchored to this core (homeland) and anchor core ID is recorded in a `task_struct`. For the `UNREF` operation, acting on local or anchor counters depends on whether migrated or not in between `REF` and `UNREF`; if the process remains at the same core (homeland), `UNREF` is done on the same local counter nonatomically. Otherwise, the migrated process *atomically* updates the anchor counter of the original reference cache in the homeland core. The use of an atomic operation on an anchor counter is indeed for correct counting even with multiple processes in concurrent environments.

We summarize the access rules in Table 1, and we ensure that a local reference cache becomes zero upon the completion of `REF`/`UNREF` operations. This allows PAYGO to reclaim zero-valued local reference caches immediately from hash, thus retaining the hash space efficiency without time overhead. The common rule governing both `REF` and `UNREF` is, we disable *preemption* while performing two operations in order to prevent *malicious data races* on a local counter. Of course, there are other ways of doing this, such as kernel spin locks (i.e., `spinlock_t`), but disabling/enabling preemption is by far the fastest method we found it suitable for our purpose and has been used in prior work [7, 10]. An in-depth performance analysis will be presented in §6.2.5.

Table 1: Access rules for `REF` and `UNREF` from homeland and foreign land. (✓: *nonatomic*, ✓̸: *atomic*, ×: no-op)

Type	PAYGO Entry			
	local counter		*anchor counter*	
	`REF`	`UNREF`	`REF`	`UNREF`
Homeland	✓	✓	×	×
Foreign land	×	×	×	✓̸

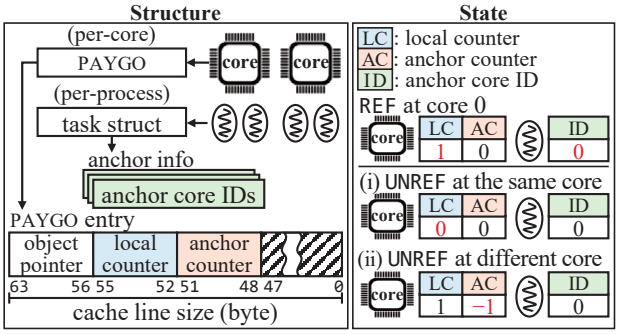


Figure 3: An overall structure of PAYGO and state of the structure when referencing and unreferencing an object.

Overall structure of PAYGO. Next, we describe the overall structure of PAYGO. Figure 3 shows the structure of PAYGO and the state of the data when an object is referenced and unreferenced. For each core, there is per core hash of reference caches, each entry of which consists of an object pointer and two counters, a *local* and an *anchor* counters. The space overhead for this hash table is much smaller than the Linux sloppy counter and larger than the traditional one, but it is similar to RefCache. Given hash of reference caches, the `UNREF` operation atomically decreases the anchor counter of the anchored core only when process migration occurs, by the access rules in Table 1. To do this, each process stores anchor information that bookkeeps the core IDs in which an object is referenced. The anchor information internally maintains multiple anchor core IDs to deal with a case where a process references an object multiple times on different cores without unreferencing it. Matching anchor core ID is removed after `UNREF` is done on the corresponding core. Unlike hash, PAYGO requires extra memory space for storing anchor information in a task structure, and this is surely regarded as additional memory overhead.

On the right side of Figure 3 shows the state of the data when a process references and unreferences an object. When a process references an object at core 0, it raises the local counter of core 0 and keeps core 0 in the process's anchor

information. When the process unreferences the object, it first searches the current core ID in its anchor information. If found, the process decreases the local counter of the current core; otherwise, it decreases the anchor counter of any core in the anchor information *atomically*.

3.2 PAYGO Operations

PAYGO has three operations: REF/UNREF operations to increase/decrease a reference counter and READ-ALL operation to read the global value of the reference counter which is equivalent to the query operation.

REF **operation.** When a REF operation of an object is invoked, it finds the PAYGO entry for the object in the hash of the current core. If the PAYGO entry is found, its local counter is increased. If the PAYGO entry is not found, a new PAYGO entry is created in the hash and the local counter is increased, and then the current core ID is stored in the process' anchor information. The REF operation is executed while preemption is disabled to prevent multiple processes from updating the same hash entry concurrently.

UNREF **operation.** When an UNREF operation of an object is invoked, it first checks the anchor information of the process. If the core ID stored in the anchor information is the same as the current core, the process finds the PAYGO entry for the object in the hash of the current core and decreases the local counter. If the process has migrated to another core, it finds the PAYGO entry for the object in the hash of the anchored core and atomically decreases the anchor counter. The UNREF operation is also performed with preemption being disabled for the same reason as the REF operation.

READ-ALL **operation.** When a READ-ALL operation is invoked, it finds all the PAYGO entries for the object in all per-core hash tables and computes the sum of the local and the anchor counters of all valid PAYGO entries. The READ-ALL operation is performed while the preemption is disabled in order to prevent any scheduling delays that may slow down the process. Nevertheless, this does not guarantee to read the correct sum since the REF and the UNREF operations may modify the counters during the READ-ALL operation. Object reclamation therefore needs a delicate design (§3.3).

3.3 Object Reclamation

Objects are a target of reference counting, and operating systems often reclaim objects that are not referenced by any process in order to keep memory pressure under control. Once an object is chosen to be reclaimed, the reclaiming process should prevent any additional reference to the object and check again the zero value of the reference counter. In traditional reference counting, this can be done atomically by comparing the shared atomic counter with zero and swapping it to a negative value. The synchronization used in the traditional method is based on *atomic read-modify-write* operation (e.g., CMPXCHG).

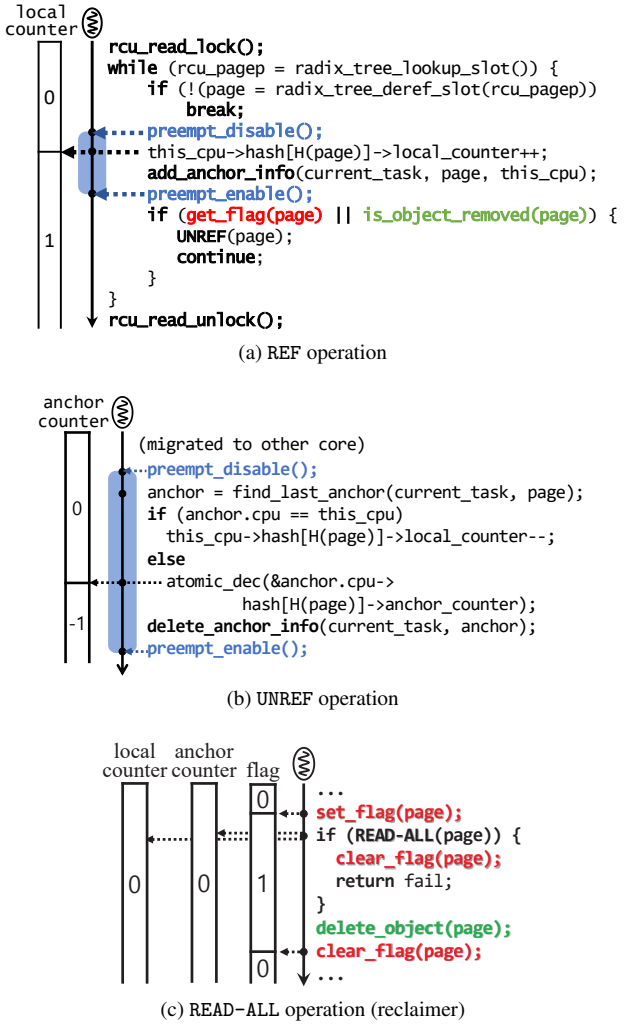


Figure 4: Code snippets of how PAYGO's REF, UNREF and READ-ALL are implemented and used in the Linux page cache, where $H()$ is a hash function.

In PAYGO, it needs more steps to correctly handle the case. Since the READ-ALL operation cannot acquire the sum in one snapshot, it uses a flag to indicate its commencement, which helps prevent the additional reference to the object. The synchronization method we use here is based on the *read-after-write* (RAW) pattern [2]. Important to notice is the invariant that at least one of a reclaiming process and referencing processes, if they run concurrently, must detect both events and then retreat itself for safety, thus never allowing malicious data race. We enforce these checking conditions to be verified at the end of REF and READ-ALL routines.

Figure 4 shows the code snippets of how REF, UNREF and READ-ALL operations are implemented in the Linux page cache with a special flag indicating that the current page is accessed for reclamation. Accessing the special flag may

cause contention only if the same page is repeatedly reclaimed (or flushed in the Linux page cache) while many processes read it, which we seldom, if ever, witnessed in Linux. In Figure 4a, the code executed while preemption is disabled denotes the REF operation. In Figure 4b, the whole code is the UNREF operation. READ-ALL operation which iterates all core's hash, finds the PAYGO entry and collects the sum of all entries again with the preemption being disabled, is only shown as a function call in Figure 4c. Notice that there is additional code around the REF and READ-ALL operations for the correct implementation of reclaiming page caches.

As shown in Figure 4a, the entire routine for referencing a page is protected by rcu_read_lock() and rcu_read_unlock(). The REF operation starts by obtaining an rcu reference to the page. Once it obtains the rcu reference, it retrieves the page object and then performs the REF operation. After that, a flag is checked to see if a reclaiming process is being tried. If the flag is clear, then the page is checked whether or not it is removed. This makes sure that the page is not already reclaimed before the flag is checked. Only if both conditions are passed, the page is safely referenced. Otherwise the flag is set, then the process retries until the reclaiming process clears the flag. If the page was already removed, the reference process fails. For the reclaiming process, the READ-ALL operation is performed after setting a flag. If the page is not referenced by any thread, the page object is safely removed and the flag is cleared. If the page is already referenced by other thread, it is not removed and the flag is cleared, thus failing to reclaim the page object.

3.4 Anchoring in Action

Reference counting techniques exploiting per-core hash, such as RefCache [7], allow processes to update nonatomic local counters of the running core. This means that when a process at core 0 increases the local counter of core 0, migrates to core 1, and decreases the local counter of core 1, then we have two local counters with values of 1 and -1, respectively. The spread-out local counters are problematic if per-core hash is used to reduce space overhead. As an example, RefCache uses background threads to flush the local counters every epoch, which inevitably delays the reclamation of zero-valued reference cache entries.

The anchoring technique in PAYGO enforces the REF and UNREF operations to act on the same PAYGO entry, thus guaranteeing the sum of its local and anchor counters to eventually become zero. Any zero-valued PAYGO entry can be recycled immediately when another REF operation accesses the same hash bucket. Figure 5 shows an example of an object accessed by multiple threads in a system with four cores. At core 0, a red thread references and unreferences the object by increasing and decreasing the local counter of core 0. At core 1, a blue thread followed by a green thread reference the object. Then, a yellow thread also references the object at core

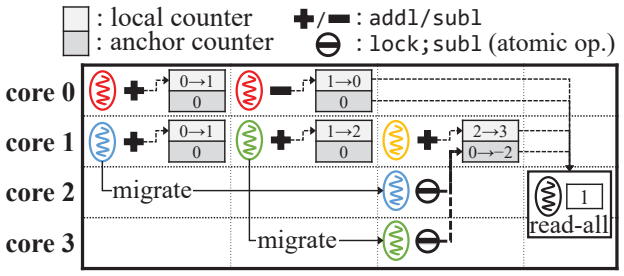


Figure 5: Usages of an anchor counter.

1. Since the yellow thread is using core 1, the blue thread and the green thread have to migrate to other cores (namely, core 2 and core 3, respectively), and unreference the object using the same anchor counter of core 1. As shown in this example, an anchor counter has the risk of being modified by multiple threads in parallel, so we use an atomic operation.

The anchoring technique gives us another opportunity of reducing the query overhead. Since the sum of local and anchor counters in a core can never be negative, during the zero detection (i.e., query), upon seeing a positive value of the sum in a core, we can immediately stop zero detection safely concluding that the object is currently being used by at least one process.

Discussion. Since decreasing an anchor counter uses an atomic operation, there is a performance concern when systems have processes that are all accessing the same anchor counter, thus hitting the hardware-based synchronization bottleneck. This is the worst case that can occur when processes are migrated frequently between REF and UNREF. But, the general design rationale for the OS scheduler usually inhibits such frequent process migration unless there are compelling reasons, such as severe load imbalance.

Nonetheless, the chance of migrating a process can increase if the interval between REF and UNREF becomes distant. Even if it occurs, atomic operations on anchor counters would not have bad impact on performance, since the price for process migration is much larger than the pure cost of reference counting itself. Hence, the performance degradation caused by atomic operations can be neglected (see anchoring overhead in §6.2.3). To alleviate any possible bottleneck on the same anchor counter, the OS scheduler can give a temporary CPU affinity to processes that are in between REF and UNREF, to prevent process migration.

One may raise concern about the overhead of searching the matching core ID in anchor information when a process references an object multiple times or numerous objects without unreferencing. Since PAYGO stores the same anchor ID in anchor information even if the same object is referenced again, this issue will surely impact performance due to the search cost, but we have not discovered such cases yet inside file systems or data management systems. If the case is found, then augmenting an additional search structure must be necessary.

3.5 Table Overflow

The table overflow problem of hash tables is a fundamental issue that per-core hash-based counting techniques should address. In the context of reference counting, the table overflow occurs when there are a large number of live objects. For instance, if a process opens many files, then the corresponding `dentry` objects will be alive in per-core hash until closed. Conventional methods, such as table doubling, are hard to use or to be efficiently designed due mainly to high concurrency. We deal with the overflow similar to the way Linux swap space is managed. First, an object that uses PAYGO has a list, called an *overflow counter list*, protected by an object lock. When a live object needs to be evicted from per-core hash, we acquire the object lock, evict the entry from hash, add the evicted counter information to the overflow counter list and then release the lock. Later, the owner process of the evicted entry can reload the evicted counter information from the overflow list while holding the object lock. Further improvements can be made to the shared overflow list, but we hold off until it really matters since '*premature optimization is the root of all evil'* [15]. What really matters here is the lifetime of the referenced object, and the concerned place (i.e., page cache) suffering bottlenecks has short-lived objects that begin and end its lifetime inside `read`/`write` system calls. PAYGO scales file operations well under such conditions.

4 PAYGO Implementation

We implemented PAYGO in Linux kernel version 4.12.5 and applied it to the page cache that can affect many concrete file systems suffering scalability issues. For experiments, we take the code base implementing RefCache and SNZI from sv6 [6] and adapt it to Linux page cache. Noticeable is the observation that other latent contention often arises after PAYGO eliminated contention on reference counters.

The Linux page cache is implemented using a radix tree, and its operations are made lockless for the performance benefits [20]. However, `read` system calls using a page cache still have scalability issues, such as the usage of atomic reference counter to synchronize between reading a page from a page cache (`REF`/`UNREF`) and flushing the page from memory to storage (`READ-ALL`). Therefore, threads trying to read the same page contend on the same reference counter and have poor performance under such loads [17].

In the original implementation of a reference counter in a page (`_refcount`) has two purposes. First, it is used as a status variable. If the value is zero, it means that the page is unused. If the value is two, the page is active and is stored in a page cache, but it is not referenced by any threads. The `_refcount` of a value above two is used as a reference counter. For example, the `_refcount` value of three indicates that there is one process referencing the page. Here, we left the `_refcount` to be used as a status variable and use PAYGO to replace the referencing part of `_refcount`.

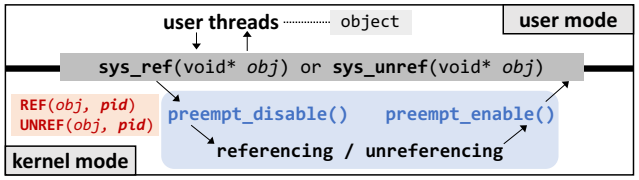


Figure 6: User-level PAYGO in Linux.

5 User-Level PAYGO

PAYGO, although being motivated by pressing concerns in file systems and intended to address it, can be easily extended to a user-level reference counting method for applications above the kernel. The development of scalable user-level reference counting is more demanding indeed, since there are many latent use cases where contention may arise once the present performance matters are all cleared away. For example, managed language runtimes (such as the JAVA runtime) use referencing counting for collecting garbage objects, and the same is true in the database land; popular database systems, such as HBase [21], RocksDB [22] and MariaDB [23], also use reference counting for managing memory objects. To the best of our knowledge, they use either hardware based atomic operations or lock based synchronization primitives to safely orchestrate concurrent accesses to the shared counter variables, but both methods are all vulnerable to performance bottlenecks in highly concurrent environments.

To make applications benefit from the better scalability of PAYGO, we implement three system calls, `sys_ref()`, `sys_unref()` and `sys_readall()`, which enable applications to exploit core kernel-level PAYGO operations without difficulty for user-level reference counting (Figure 6). Despite there being the inherent overhead required in switching between user mode and kernel mode, reference counting on user-level objects through PAYGO system calls enables applications to achieve far better scalability than their legacy reference counting techniques. Furthermore, PAYGO will exhibit much less overhead for managing garbage entries in per-core hash, and this is a required feature especially when applications hold a large number of live references.

Enabling applications to directly exploit the reference counting technique in the kernel via system calls poses two nontrivial issues. First, the system call overhead should be sufficiently minimized to benefit from the original performance of the kernel-level reference counting technique. To this end, we make PAYGO system calls lightweight such that they just wrap core kernel-level PAYGO routines with `preempt_disable()`/`preempt_enable()` executed beforehand/afterward. The wrapped routines here basically refer to the code segments bounded by `preempt_disable()` and

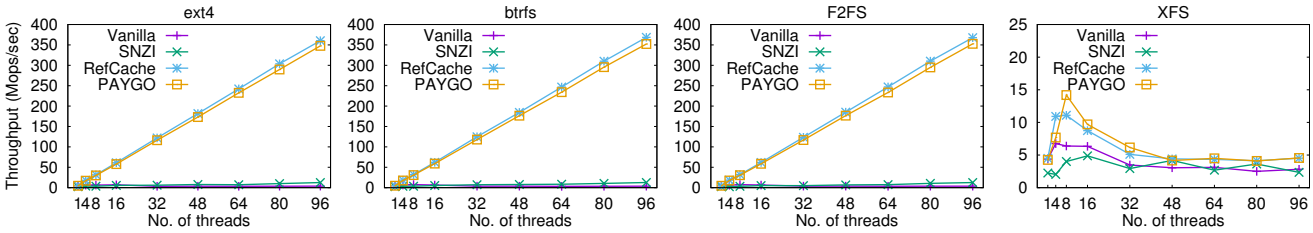


Figure 7: Scalability comparison under strongly contending workloads: the Linux page cache.

`preempt_enable()` in Figure 4. One subtle matter is, we have to transform the virtual address of a user object into a unique one inside per-core hash, by combining it with the `pid` of a user process. Second, since applications are not as reliable as kernel, the abnormal termination of applications may leave the kernel data structures for reference counting incorrect. When an application terminates after referencing an object but before unreferencing it, the corresponding counter in the kernel can never become zero. To resolve the problem, when terminating a process, the `task_struct` needs to be checked to detect any left-over counter values in the corresponding PAYGO entries in per-core hash tables. Any such left-over counters, if found, must be decreased.

6 Evaluation

In this section, we measure the overall performance and scalability of PAYGO, especially in page cache, and compare with other reference counting techniques including RefCache, SNZI and traditional reference counter under various file systems. For analyzing the performance of user-level PAYGO, we compare PAYGO with existing user-level reference counting techniques.

6.1 Experimental Setup

We perform all of the experiments in Linux kernel version 4.12.5 on our 96-core system equipped with four 24-core Intel Xeon E7-8890 v4 CPUs and 1 TiB DDR4 DRAM. We run FXMARK microbenchmark [17] with a RAM disk and filebench [11, 25] with a Samsung SM1725 NVMe SSD. To show the general applicability of PAYGO, we conduct experiments under four different file systems (i.e., `ext4`, `btrfs`, `F2FS` and `XFS`). In `ext4`, we used the default *journaling* mode and did not see any lock contention in the journaling subsystem observed in the prior study [17]. Page structures cached in memory are freed before every experiment, and the Linux security module is turned off to avoid the unrelated performance degradation.

6.2 Scalability

This section explores the multicore scalability of concerned file systems under file system benchmarks, with the degree of contention being varied from strongly contending to weakly contending. Our evaluation methodology follows similar approaches used in [17], and the important metric is the number of `REF/UNREF` (i.e., file reads) with the degree of contention on reference counters being controlled by the size of files accessed by benchmark threads. Experiments under this controlled environment may reveal the weakness and strength of tested schemes that may overlook at the time it was proposed.

6.2.1 Strongly Contending Workloads

To evaluate the performance of file operations under strongly contending scenarios, we ran the shared block read workload (i.e., DRBH) in FXMARK, a microbenchmark that is intended to stress file systems. For the evaluation, a varying number of threads repeatedly read the same 4 KiB data block, thus stressing the reference counting part enormously. This workload is known to reveal the contention resilience of any reference counting approach, since the stock Linux suffers the most. Figure 7 shows the results. With this workload, all file systems under consideration in stock Linux (i.e., vanilla) undergo severe scalability bottlenecks arising from contention on the same reference counter. SNZI shows slight improvements over the vanilla scheme that uses the traditional reference counter. PAYGO and RefCache perfectly scale the throughput of `ext4`, `btrfs` and `F2FS`. The main reason for slightly lower performance of PAYGO than RefCache is because the number of instructions executed by PAYGO is slightly greater than RefCache. By profiling on clock cycles, we obtain the cycle difference that matches the performance difference we observe here.

Interesting is the performance degradation that has been consistently observed in `XFS` primarily due to contention on the semaphore inside an `inode` structure, which completely renders all reference counting methods useless. Although a further investigation is needed, it is worthwhile putting effort to redesign this coarse-grained locking so that `XFS` can reap performance benefits from better counting techniques.

6.2.2 Weakly Contending Workloads

To evaluate the performance of file systems under weakly contending scenarios, we used filebench, a benchmark that

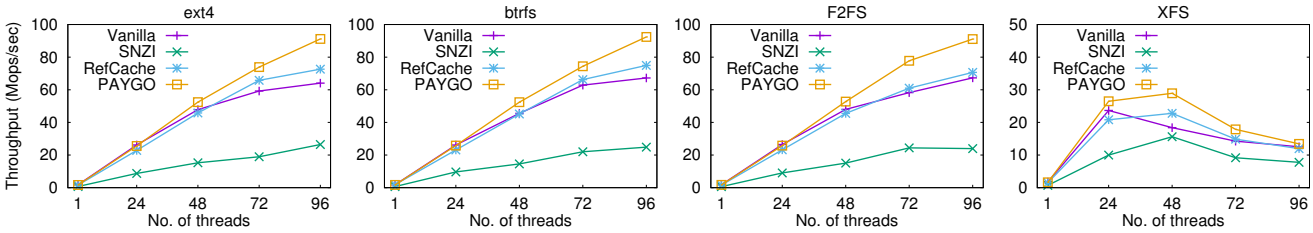


Figure 8: Scalability comparison under weakly contending workloads: the Linux page cache.

can flexibly add and test workloads to file systems and storage. Before we run the filebench, we modified the filebench code to experiment with more flexibility on multicores. Originally, filebench is implemented with a lock for each file and only one thread can access the file at a time. We eliminated the file lock so that multiple threads can access the file concurrently. For the evaluation, we ran the randomread workload with participating threads performing 64 bytes random reads from one of ten 128 MiB files. Since weakly contending workloads disperse contention on reference counters, it may expose any latent overhead (or downside) of given counting techniques that has been overlooked in exchange for resolving high contention arising under strongly contending scenarios.

The throughput results are shown in Figure 8. Strikingly the vanilla scheme deployed in the stock Linux page cache performs well after it reduces hotspot contention; it outclasses SNZI all the time and sometimes outperforms RefCache with a slight margin. As the thread count increases, the throughput gap between PAYGO and RefCache widens due to a large number of garbage entries that increase hash collisions in RefCache's per-core hash, which were not observed under strongly contending workloads.

Again, none of the tested counting techniques scale the performance with `XFS` due to bottlenecks inside `XFS`, and this will be discussed in detail in the following section.

6.2.3 In-Depth Analysis

In order to reveal detailed information about various system activities, we perform an in-depth analysis with moderately contending workloads being profiled over different metrics.

Stressing page cache. We first ran the randomread workload of the filebench microbenchmark, by varying the number of files whose size is set to 32 MiB. We chose the moderately contending workload as a good proxy for stressing reference counting schemes with a reasonable balance of contention and the count of objects referenced. Figure 9 shows the throughput and the CPU breakdown of the benchmark.

First, the in-depth profiling gives clear explanations for two strange observations in XFS and SNZI. The first observation is the poor scalability of XFS. The main culprit for this problem is due to severe lock contention inside the file system; `xfs_ilock()` and `xfs_iunlock()` on the `inode` of a file. Lock contention mainly depends on the number of files, not the file size. High contention on the `inode` lock indeed leads to severe performance degradation regardless of reference counting schemes. This perhaps needs attention from our community. The second observation is the poor scalability of SNZI, and SNZI also has a similar culprit for the issue; it scales poorly regardless of file systems at this time. Since the only publicly available code base for SNZI can be taken from sv6 [6], we show the results as is.

The vanilla scheme can scale the performance of `ext4`, `btrfs` and `F2FS` quite well as the thread count increases. Although the overhead of atomic instructions grows in proportion to the thread count, the dispersed contention cancels out the negative impact of atomic operations we have seen in Figure 7. With 72 threads, the vanilla scheme performs almost on a par with RefCache. An in-depth analysis of performance over different contention levels will be discussed in the next experiment.

RefCache shows worse core scalability than PAYGO, and this is mainly because of the increased overhead of handling hash collisions (i.e., atomic lock operations) in RefCache. Also, noticeable is the slight performance degradation of the vanilla, RefCache and PAYGO as the file count increases. This is due to the increased memory access overhead for reading files larger than cache memory, which is not observed in experiments with the same number of smaller files, although results are omitted here due to the space limitation.

Performance spectrum over degree of contention. We further investigate the performance spectrum over different contention levels to fully grasp the nature of the space-time tradeoff. For this evaluation, we modified the FXMARK benchmark in a way that 96 threads perform 64 bytes sequential reads per 4 KiB page on a single file whose size is varied from 1 MiB to 64 MiB, with `ext4` mounted. Figure 10 shows the performance spectrum of three concerned schemes. The most noticeable result is the sharp throughput decrease in RefCache as the file size grows, which clearly shows the negative effect of a large time overhead to scalability and so the necessity of the instant reclamation of hash entries. PAYGO effectively addresses the problem and undergoes no performance overhead for that issue. The gradual degradation of the throughput in PAYGO is due to the file data overflow in cache memory, resulting in the increased memory access overhead, which also occurs in other schemes.

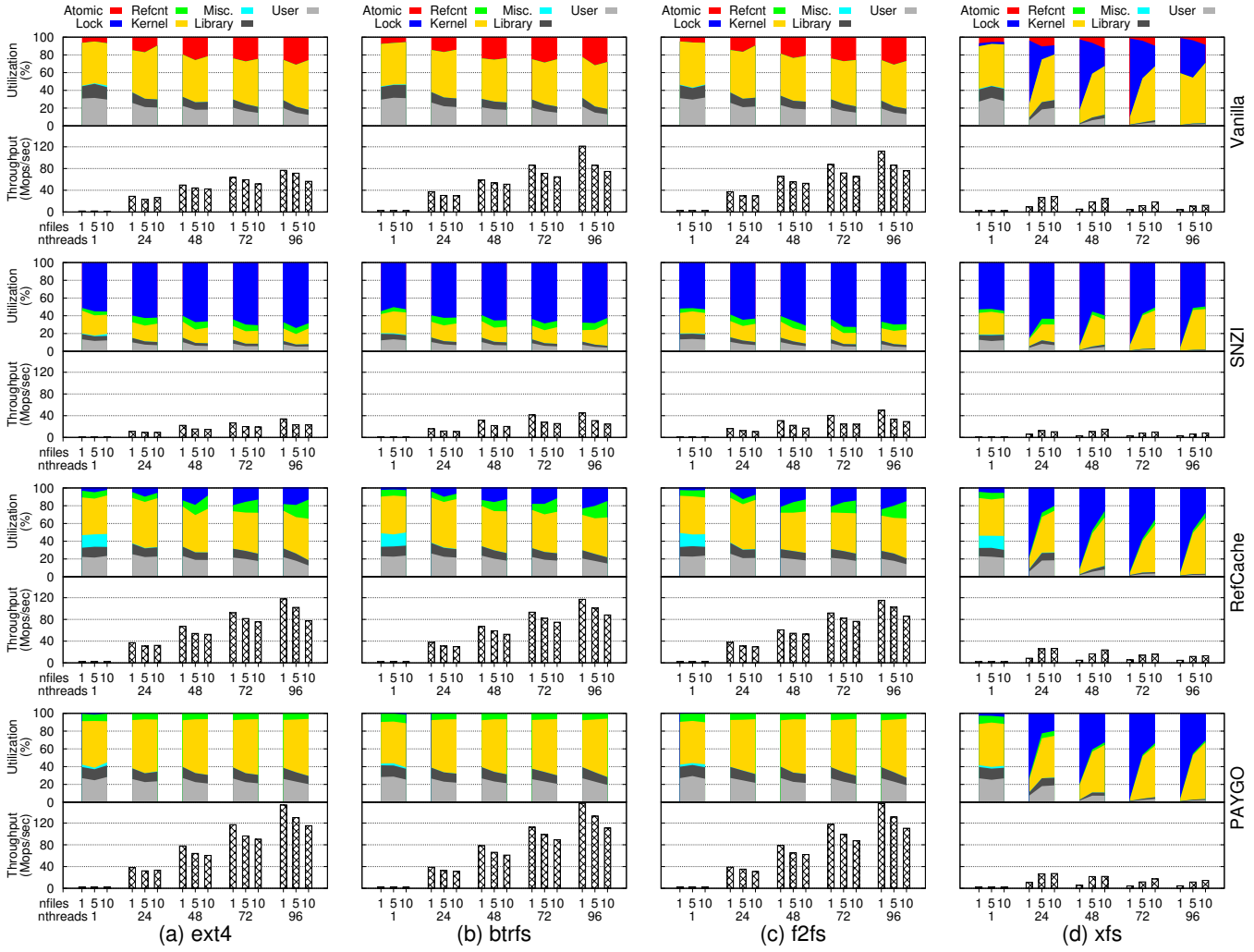


Figure 9: The performance and the CPU breakdown of file systems (i.e., column labels (a)-(d)) with different reference counting schemes (i.e., row labels on the right side) under moderately contending workloads: the Linux page cache.

As we analyzed earlier, the vanilla scheme is ill-suited for the strongly contending condition (i.e., 1 MiB). But its performance rebounds quickly as soon as the degree of contention is alleviated, and it outperforms RefCache once it passes a break even point (i.e., 16 MiB file in our case). In-depth looking through profiling reveals that the acquire/release of an object lock in RefCache to handle hash collisions incur more overhead than the atomic operations in the vanilla scheme when hash collisions occur frequently due to a large number of objects accessed. After the break even point, the throughput of the vanilla scheme starts to decrease because the increased memory access overhead due to the file data overflow in cache memory becomes larger than the merit of dispersed contention.

Anchoring overhead. Since the anchor counter can be contended by only migrated threads, the frequency of thread migration determines the anchoring overhead. As described in §3.4, the design rationale for the OS scheduler usually inhibits frequent process migration. To confirm it, we ran the openfiles workload of the filebench on all cores that could cause thread migration between `REF` and `UNREF` operations, and counted the number of migration. For this experiment, we created 2,000 threads running the openfiles workload on 36 physical cores (disabling 60 cores), which hopefully causes frequent thread migration due to the load imbalance. However, during the 60 seconds experiment, less than 10,000 times of migration occurred.

Moreover, regardless of how the Linux scheduler is implemented, the more frequent the thread migration occurs, the less effective the CPU time is due to the long latency of the context switch. The latency of the context switch can be as short as 1 microsecond [24, 19] which is still relatively larger than the overhead of the atomic operation [13]. Recently, there has been an effort to reduce the latency of the context switch to several tens of nanoseconds by emulating a thread at the user level [24], but no such study has been

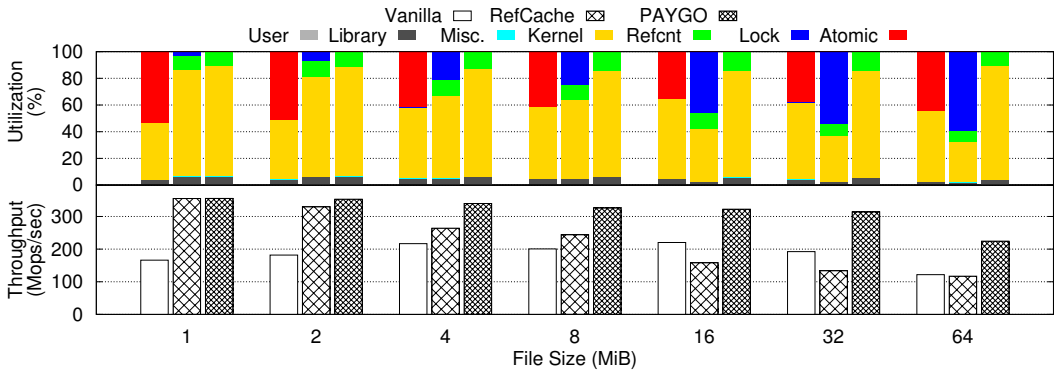


Figure 10: Performance spectrum of the vanilla, RefCache and PAYGO over varying contention levels on `ext4`.

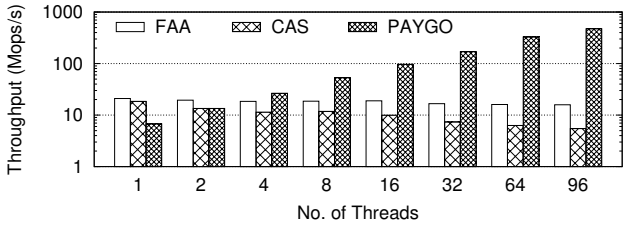


Figure 11: Throughput under the strongly contending workload (user-level reference counting).

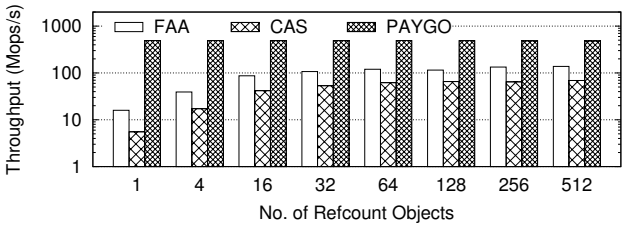


Figure 12: Performance spectrum of *FAA*, *CAS* and user-level PAYGO with a varying number of objects.

found in the kernel level. Therefore, there is practically no reduction in system throughput due to frequent changes of the anchor counter in PAYGO.

6.2.4 Scalability of User-level PAYGO

Next, we evaluate the performance of user-level PAYGO system calls to see its applicability. For the evaluation, we use a microbenchmark that has a varying number of threads (un)referencing user-level objects repeatedly. For comparison, we implement two methods based on our observation. The first one is to use atomic `fetch_add` and `fetch_sub` for reference counting. We call it *FAA*, and this is a typical implementation widely adopted in many systems. Note that this technique does not show performance collapses, but it cannot scale performance mainly due to hardware-based synchronization bottlenecks. The second one is to implement what is being used in the Linux page cache, which is based on the atomic compare-and-swap instruction. We call this *CAS*.

Figure 11 shows the throughput (i.e., the number of fetch_add/fetch_sub and `REF/UNREF` operations per second) of three schemes as we increase the number of threads, all of which access a single shared user-level object. As we manifested, the mode switch overhead of user-level PAYGO is quite noticeable and expected, considering the performance of *FAA* and *CAS* until 2 threads in our system. The performance number *FAA* and *CAS* achieve with 1 thread, however, is the peak number obtainable for reference counting a single shared object. After 2 threads, both *FAA* and *CAS* are either saturated or slowly degrading. Meanwhile, our user-level PAYGO scales the performance with no contention overhead. Figure 12 shows the performance spectrum of three methods as we increase the number of referenced objects with 96 threads. As shown in the figure, *FAA* and *CAS* suffer from synchronization bottlenecks initially when all of 96 threads access a small number of objects, but they slowly gain throughput up to a certain point as contention is dispersed. We believe that the saturation point observed here (i.e., 137 Mops/s) reaches the maximum capacity that our 96-core server can support. On the other hand, our user-level PAYGO could sustain the maximum throughput regardless of the count of objects.

Impact on application performance. As demonstrated above, user-level PAYGO may have a profound impact on application performance especially on multicore hardware. We have been conducting an in-depth code-level analysis of latent bottlenecks caused by reference counting in MongoDB, MariaDB, Boost.Asio, etc. What we have learnt from our preliminary study on such systems is that many applications using user-level reference counting mostly suffer performance bottlenecks that start occurring much earlier before the reference counting is responsible for severe performance degradation. For example, database systems we analyzed have recently undergone major changes to enhance its multicore scalability. As the systems community is battling pressing concerns, the contention around reference counting will soon appear as a primary bottleneck in achieving scalable performance.

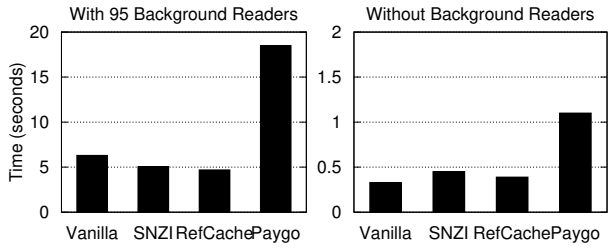


Figure 13: Query overhead comparison.

6.2.5 Comparing `preempt_disable()` and spin lock

As we briefly mentioned in §3.1, the use of `preempt_disable()`, instead of kernel spin locks or something similar, to prevent data races in `REF`/`UNREF` needs concrete justification. Hence, we compared the overheads of both methods by measuring the average clock cycles per each function pair (i.e., `preempt_disable()`/`preempt_enable()`, `spin_lock()`/`spin_unlock()`) by iterating them up to a billion times. The results show that the clock cycles for `preempt_disable()` and `spin_lock` converge to 14 cycles and 50 cycles, respectively. Throughout the experiments, the overhead of `preempt_disable()` remains a constant fraction ($\sim$30%) of that of `spin_lock` regardless of the number of iterations performed. The main reason for the higher overhead of `spin_lock` is because it internally invokes `preempt_disable()` and executes additional code segments including an atomic instruction for cross-core communication supporting mutual exclusion on a global object. This is undoubtedly overkill for our case where we also use an atomic operation to safely decrease an anchor count from remote cores. In conclusion, `preempt_disable()` is a fast and safe method, as it has shown its usefulness in prior work, for preventing data races on local counters in our `REF`/`UNREF` implementations.

6.3 Query Overhead

In this section, we conduct the performance evaluation of the `READ-ALL` operation. To evaluate the query overhead of PAYGO, we measure the time to flush a 4 GiB file in the Linux page cache with and without background readers on `ext4`. The experiment first reads the entire file so that file blocks are all loaded in page caches. Then, it measures the time taken to drop the file from page caches. Since page caches are all clean (i.e., unmodified), dropping page caches is comprised of pure CPU activities. Figure 13 shows the completion time of different reference counting techniques. With background readers, RefCache surprisingly outpaced all other competitors, since RefCache may read a batch of global counters for multiple pages safely if their hash entries were flushed two epochs ago and no referencing occurred in between. Although PAYGO has less query overhead than RefCache for a single reference counter, the benefit of syncing the entire hash of dirty reference caches to global counters predominates the time overhead of two epochs with a large number of objects. Meanwhile PAYGO exhibits the overhead of reading a large number of local counters for each page and the vanilla scheme suffers contention due to atomic operations. Without background readers, the vanilla scheme is better than RefCache, but PAYGO still shows the same overhead of reading local counters. Nevertheless, the query overhead of PAYGO does not commensurate with the number of cores owing to its early detection of positive reference counter values (§3.4).

7 Limitations and Future Work

The limitations of PAYGO can be summarized as follows. First and foremost, PAYGO is not completely free from the counting-query tradeoff. We do not have a clue on whether it is possible or not. A proposal achieving low overhead in all directions must be a major breakthrough in systems research. Second, the way we handle the table overflow is rather naive, and one may find practical use cases that can stress PAYGO in that the overflow counter list is spotted as a bottleneck point. Our ongoing work is to apply user-level PAYGO to language runtime systems that surely benefit from user-level PAYGO.

8 Conclusion

Reference counting in modern file systems is not scalable on multicores, even under workloads with little or no logical contention. Through in-depth survey of present reference counting techniques designed for scaling file I/O operations, we found that there are space-time tradeoff and query-counting tradeoff in designing scalable reference counting techniques. In this paper, we have presented a novel reference counting scheme, PAYGO, that escapes the space-time tradeoff by using an anchor counter. PAYGO provides scalable counting and space efficiency with negligible time delay for the reclamation of hash entries. We have implemented PAYGO in the page cache in Linux. Our evaluation with different file system benchmarks demonstrated that PAYGO is practically useful in addressing severe contention arising in other reference counting techniques.

Acknowledgements. We would like to thank our shepherd, Vijay Chidambaram, and the anonymous reviewers for helping us improve this paper. This work was supported by the National Research Foundation of Korea grant (2017R1A2B4006134) and the Ministry of Science and ICT (MSIT), Korea (R0114-16-0046, Software Black Box for Highly Dependable Computing), and (2016-0-00023, National Program for Excellence in SW) supervised by the Institute for Information and communications Technology Promotion (IITP), Korea.

References

[1] ACAR, U. A., BEN-DAVID, N., AND RAINEY, M. Contention in structured concurrency: Provably efficient dynamic non-zero indicators for nested parallelism. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2017), ACM, pp. 75–88.

[2] ATTIYA, H., GUERRAOUI, R., HENDLER, D., KUZNETSOV, P., MICHAEL, M. M., AND VECHEV, M. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 487–498.

[3] BHAT, S. S., EQBAL, R., CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scaling a file system to many cores using an operation log. In *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles* (2017), ACM.

[4] BOYD-WICKIZER, S. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels*. PhD thesis, Massachusetts Institute of Technology, 2014.

[5] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., ZELDOVICH, N., ET AL. An analysis of linux scalability to many cores. In *OSDI* (2010), vol. 10, pp. 86–93.

[6] CLEMENTS, A., ZELDOVICH, N., ET AL. sv6: Posix-like scalable multicore research os kernel. `https://github.com/aclements/sv6`, 2014.

[7] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 211–224.

[8] COLLINS, G. E. A method for overlapping and erasure of lists. *Commun. ACM 3*, 12 (Dec. 1960), 655–657.

[9] CORBET, J. The search for fast, scalable counters. `https://lwn.net/Articles/170003/`, 2006.

[10] CORBET, J. Per-cpu reference counts. `https://lwn.net/Articles/557478/`, 2013.

[11] CORBET, J. Filebench. `https://github.com/filebench/filebench/wiki`, 2017.

[12] DASHTI, M., FEDOROVA, A., FUNSTON, J., GAUD, F., LACHAIZE, R., LEPERS, B., QUEMA, V., AND ROTH, M. Traffic management: a holistic approach to memory placement on numa systems. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 381–394.

[13] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 33–48.

[14] ELLEN, F., LEV, Y., LUCHANGCO, V., AND MOIR, M. Snzi: Scalable nonzero indicators. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (2007), ACM, pp. 13–22.

[15] KNUTH, D. E. Structured programming with go to statements. *ACM Comput. Surv. 6*, 4 (Dec. 1974), 261–301.

[16] MCKENNEY, P. E., BOYD-WICKIZER, S., AND WALPOLE, J. Rcu usage in the linux kernel: One decade later. *Technical report* (2013).

[17] MIN, C., KASHYAP, S., MAASS, S., AND KIM, T. Understanding manycore scalability of file systems. In *USENIX Annual Technical Conference* (2016), pp. 71–85.

[18] NARULA, N., CUTLER, C., KOHLER, E., AND MORRIS, R. Phase reconciliation for contended in-memory transactions. In *OSDI* (2014), vol. 14, pp. 511–524.

[19] PETER, S., LI, J., ZHANG, I., PORTS, D. R., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS) 33*, 4 (2016), 11.

[20] PIGGIN, N. A lockless page cache in linux. In *Proceedings of the Linux Symposium* (2006), vol. 2.

[21] Apache HBase. `https://github.com/apache/hbase/blob/rel/2.1.0/hbase-server/src/main/java/org/apache/hadoop/hbase/io/hfile/bucket/BucketCache.java#L484`.

[22] Facebook RocksDB. `https://github.com/facebook/rocksdb/blob/v5.14.3/utilities/persistent_cache/hash_table_evictable.h#L62`.

[23] MariaDB Server. `https://github.com/MariaDB/server/blob/10.3/storage/innobase/buf/buf0buf.cc#L4350`.

[24] SEO, S., AMER, A., BALAJI, P., BORDAGE, C., BOSILCA, G., BROOKS, A., CASTELLO, A., GENET, D., HERAULT, T., JINDAL, P., ET AL. Argobots: A lightweight threading/tasking framework. Tech. rep., Argonne National Laboratory (ANL), 2016.

[25] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking. *USENIX; login 41* (2016).

Notes

[1] PAYGO: *pay* migration tax (i.e., the anchoring overhead) *as you go* to other core.