Prudent Memory Reclamation in Procrastination-Based Synchronization

Aravinda Prasad
Indian Institute of Science, Bangalore &
IBM Linux Technology Center
aravinda@csa.iisc.ernet.in

K Gopinath
Indian Institute of Science, Bangalore
gopi@csa.iisc.ernet.in

Abstract

Procrastination is the fundamental technique used in synchronization mechanisms such as Read-Copy-Update (RCU) where writers, in order to synchronize with readers, defer the freeing of an object until there are no readers referring to the object. The synchronization mechanism determines when the deferred object is safe to reclaim and when it is actually reclaimed. Hence, such memory reclamations are completely oblivious of the memory allocator state. This induces poor memory allocator performance, for instance, when the reclamations are ill-timed. Furthermore, deferred objects provide hints about the future that inform memory regions that are about to be freed. Although useful, hints are not exploited as deferred objects are not *"visible"* to memory allocators.

We introduce Prudence, a dynamic memory allocator, that is tightly integrated with the synchronization mechanism to ensure visibility of deferred objects to the memory allocator. Such an integration enables Prudence to (i) identify the safe time to reclaim deferred objects' memory, (ii) have an inclusive view of the allocated, free and about-to-be-freed objects, and (iii) exploit optimizations based on the hints about the future during important state transitions. Our evaluation in the Linux kernel shows that Prudence integrated with RCU performs 3.9× to 28× better in micro-benchmarks compared to SLUB, a recent memory allocator in the Linux kernel. It also improves the overall performance perceptibly (4%-18%) for a mix of widely used synthetic and application benchmarks. Further, it performs better (up to 98%) in terms of object hits in caches, object cache churns, slab churns, peak memory usage and total fragmentation, when compared with the SLUB allocator.

ASPLOS '16, April 02–06, 2016, Atlanta, GA, USA

DOI: http://dx.doi.org/10.1145/2872362.2872405

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.4.2 [*Operating Systems*]: Storage Management—Allocation/deallocation strategies

General Terms Design, Synchronization, Performance

Keywords Read-Copy-Update (RCU), Memory Reclamation, Dynamic Memory Allocator

1. Introduction

The evolution of multicore systems has increased the need for highly scalable synchronization mechanisms. Traditional lock-based synchronization techniques such as reader-writer locks do not scale for large multicore systems [4, 44]. This has led to the exploration of non-traditional synchronization mechanisms such as Read-Copy-Update (RCU) [31, 34, 37], where synchronization is achieved via procrastination [32]. Although synchronization via procrastination was first proposed in the 1980s [27], there is an increased interest in such techniques in the recent past [1, 9, 23, 29, 30, 43] as they achieve near-linear scalability even at high core counts.

Synchronization via procrastination has been applied to various data structures such as lists, trees [1, 9, 23], hash tables [32, 39, 41, 42], and used in number of operating systems including DYNIX/PTX [37], Linux [2], Tornado [16], K42 [5]. RCU is widely used in the Linux kernel [33] with several subsystems using the RCU synchronization primitive to achieve better scalability [36, 38].

In order to update an object, writers in synchronization via procrastination, create a new version of the object and defer the freeing of the old object versions until there are no readers referring to the old version. As a consequence, multiple versions of the object exist at the same time resulting in readers accessing stale data. Such staleness, however, is acceptable in many situations [32]. This is a radical departure from traditional synchronization mechanisms where writers update objects in place.

From the memory allocator's stance, synchronization via procrastination generates high stress on the memory alloca-

tor. This is because of frequent allocation and freeing of objects as each update operation allocates a new object and defers the freeing of the old version of the object. Furthermore, the synchronization mechanism determines when the old version of the object is safe to reclaim and when it is actually reclaimed. Normally such memory reclamations are batched and either offloaded to worker threads or processed in the background to avoid interference with the application [21, 35, 40, 43]. This induces unconventional allocation and freeing patterns on the underlying memory allocators; object allocation is spread over an interval of time, whereas freeing occurs in bursts. Existing memory allocators are not designed to handle such allocation patterns.

Additionally, procrastination-based synchronization mechanisms throttle the amount of memory reclamations performed at a time depending on the backlog of the deferred objects. However, such throttling is completely oblivious of the memory allocator state. Ill-timed throttling by the synchronization mechanism can adversely affect the performance of memory allocators and further cause out-of-memory situations when the update rates are high.

Apart from its impact on the memory allocator's performance, the deferred freeing of objects has an inherent property that interests memory allocators from the perspective of providing hints about the future. Intuitively such insight into the memory regions that are about to be freed are valuable to memory allocators. However, existing memory allocators cannot take advantage of hints about the future as deferred objects are not *visible* to memory allocators.

It is important to note that in conventional synchronization mechanisms (such as reader-writer locks), readers and writers have mutual exclusion. Thus, writers neither copy and update an object nor wait for the readers to free the old version of the object. Instead, objects are updated in place. Hence, such synchronization mechanisms neither provide hints about the future nor exhibit such pathological effect on memory allocators. In addition, they are not impacted by the performance of memory allocators as writers do not allocate a new object and free the old version of the object.

We introduce Prudence, a dynamic memory allocator based on slab allocator [8] principles, but that tightly integrates with procrastination-based synchronization mechanism to gain visibility into the deferred objects. In memory allocator parlance, we are integrating a specific mutator (synchronization via procrastination) with the underlying memory allocator to gain visibility to the lifetime information of the objects. This enables the memory allocator to efficiently handle the objects that are deferred for freeing. Such an integration enables Prudence to wait until the completion of readers referring to the old version of the object before reclaiming the associated memory. Furthermore, integration with the synchronization mechanism enables Prudence to have an inclusive view of the allocated, free and about-to-be-freed objects equipping Prudence to properly time the reclamation of deferred objects. Additionally, Prudence exploits optimizations based on hints about the future during important state transitions.

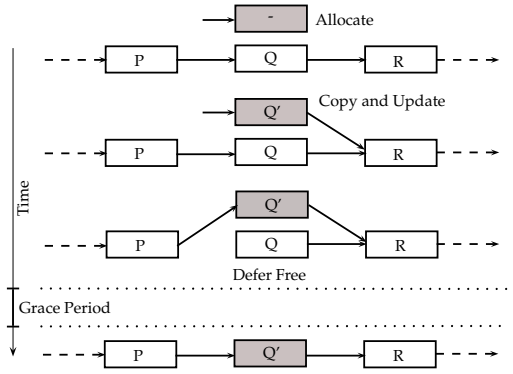

Figure 1. Procrastination-based synchronization technique applied to linked list update operation

Prudence provides a simple *turnkey* replacement for existing subsystems to defer the freeing of an object. Hence, any increased complexity of such an enhancement is limited to memory allocators and does not extend to subsystems using synchronization via procrastination.

We evaluate our idea with the RCU synchronization mechanism in the Linux kernel along with a slab-based memory allocator. Our evaluation of Prudence integrated with RCU shows 3.9× to 28× performance improvement for micro-benchmarks compared to the SLUB allocator. Prudence improves the overall performance of widely used synthetic (Postmark, Netperf) and application (Apache, PostgreSQL) benchmarks. Further, it performs better (up to 98%) in terms of object hits in caches, object cache churns, slab churns, peak memory usage and total fragmentation when compared with the SLUB allocator.

To the best of our knowledge, Prudence is the first memory allocator designed to handle reclamation of deferred objects in the memory allocator by tightly integrating with the synchronization mechanism. In addition, it exploits optimizations based on hints about the future.

We briefly introduce procrastination-based synchronization mechanism and the slab memory allocator (§2) followed by the impact on memory allocators due to processing deferred objects in a synchronization setting (§3). The design of Prudence memory allocator is then covered in §4 with evaluation of Prudence in the Linux kernel in §5.

2. Background

2.1 Synchronization via Procrastination

In this section, we explain synchronization via procrastination using the example of a linked list update operation. Readers in synchronization via procrastination traverse the list concurrently with writers without explicitly synchronizing with the writers. Hence the wait-free readers scale lin-

```
writer(...) {
  ...
  old_object = list_update(...);
  rcu_register_callback(cb_func, old_object);
  ...
}

/* Invoked after a grace period by RCU */
cb_func(void *old_object) {
  free(old_object);
}
```

Listing 1. The RCU defer free API

early with near-zero overhead. However, writers are responsible for guaranteeing consistent view of data structures to readers.

A writer updating object Q (Figure 1), instead of updating it in place, allocates a new object, copies the contents of Q to the new object and performs the update on the new object. The writer inserts the updated object Q′ and removes the old object Q from the list. Once object Q is removed from the list, no new readers can gain reference to it. But there could be pre-existing readers referring to it. Hence, the writer defers the freeing of object Q until the completion of the pre-existing readers. As readers traversing the list are wait-free and hence make progress, the number of readers referring to object Q eventually becomes zero.

The synchronization mechanism keeps track of the deferred objects in the system and waits for the completion of pre-existing readers before reclaiming the deferred objects' memory. Different techniques are available to identify the completion of pre-existing readers [22]. One such technique employed by RCU in the Linux kernel restricts readers from (i) holding reference to an object outside the read-side critical section and (ii) relinquishing the CPU inside a read-side critical section. Hence, a context switch on a CPU implies the completion of all prior read-side critical sections on that CPU i.e., there are no readers holding reference to deferred objects on that CPU. Therefore, a context switch on all the CPUs after an update operation implies that there are no readers in the entire system holding reference to the object deferred during the corresponding update operation [31, 34]. Hence, a deferred object has to wait for a context switch on all the CPUs to ensure the completion of pre-existing readers. Such a wait time after which a deferred object is eligible for freeing is called a grace period. RCU invokes memory reclamation routine for the deferred objects which have waited for a grace period.

RCU can wait for the completion of pre-existing readers for several deferred objects without explicitly tracking each and every deferred object. This is because a context switch on all the CPUs after a set of deferred frees ensures the completion of the pre-existing readers for these deferred objects.

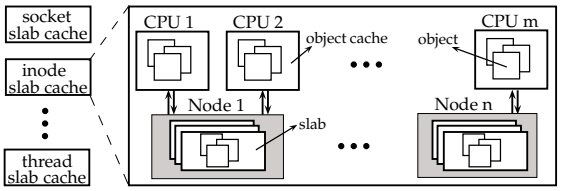

Figure 2. Structure of a slab allocator

2.2 Defer freeing an object

The code snippet in Listing 1 shows the API provided by RCU for writers to asynchronously defer the freeing of an object. A writer after updating an object registers a callback function with RCU to defer free the old version of the object. RCU invokes the callback function `cb_func`, which performs the actual freeing, once the deferred object is safe for freeing, i.e., after the completion of the grace period.

2.3 Slab Allocator

A slab allocator [8] is a dynamic memory allocator that efficiently manages allocation requests for objects of the same type and size. The slab allocator consists of caches (referred as slab caches) of commonly used objects. For example, slab allocators in OS kernels have different slab caches for commonly used objects such as inodes, threads.

Each individual slab cache constitutes a per-CPU object cache and a per-NUMA node list of slabs as depicted in Figure 2. The per-CPU object cache (hereafter referred as object cache) is a cache of free objects. The per-NUMA node list (hereafter referred as node list) contains the list of slabs belonging to the same NUMA domain. A slab is one or more contiguous pages of memory, split-up into equal sized objects.

Object caches are filled by moving the required number of objects from slabs in the node list. A slab is marked fully allocated, partially allocated or free depending on the number of objects available in the slab. The slab is considered fully allocated or full when all of its objects are moved to object caches.

The slab allocator attempts to serve an allocation request by looking for a free object in the current CPU's object cache. An allocation request successfully served from the object cache involves minimal work and is efficient; such an allocation is referred as cache hit. An allocation request that cannot be served from the object cache requires refilling of the object cache from the current node's list of slabs. If all the slabs in the node list are full, the slab cache is grown by allocating one or more slabs from the page allocator.

Similarly, upon free, the slab allocator puts the object into the current CPU's object cache. In case the object cache overflows, the objects are flushed into the respective slabs in the node list. If the number of free slabs crosses the threshold, the slab cache is shrunk by returning the pages associated with the free slabs to the page allocator.

3. Motivation

In this section we discuss the important aspects of synchronization via procrastination that impact the performance of dynamic memory allocators.

3.1 Bursty freeing of memory

Real-world applications perform thousands of update operations during a single grace period interval [40]. Each update operation defers the freeing of one or more objects. For example, tree re-balancing results in multiple deferred objects [9]. Hence, there can be thousands of deferred objects awaiting the completion of the grace period. The synchronization mechanism initiates the memory reclamation of eligible deferred objects after the completion of the grace period. As a result, freeing of objects occurs in bursts.

Bursty freeing of objects causes object cache overflow resulting in object cache flushing. Moreover, object cache flushing can happen in parallel on all CPUs as deferred objects are processed in parallel on all CPUs [40]. Parallel flushing of object caches on all CPUs causes contention on the node list lock. In addition, it increases the number of free slabs in the node list resulting in slab cache shrinking when the number of free slabs exceeds the threshold.

The worst case occurs when the synchronization mechanism schedules memory reclamation of deferred objects soon after refilling the object cache. Freeing deferred objects during reclamation causes object cache overflow. This results in object cache flush, which wastes resources and time spent by memory allocator in refilling the object cache.

Prudence acts with caution by considering the number of deferred objects waiting for reclamation when performing object cache refill/flush.

3.2 Extended object lifetimes

Lifetime of an object (including that of a deferred object) is the time interval between the allocation of an object and freeing of that object back to the memory allocator. Although deferred objects are safe for freeing soon after the completion of the grace period, they are in fact freed only when processed by the synchronization mechanism.

The synchronization mechanism incurs delays in processing the deferred objects and hence the memory associated with the deferred objects may not be reclaimed immediately after the completion of the grace period. For example, as RCU has to explicitly process individual deferred objects by invoking the registered callback functions, the processing of deferred objects towards the end of the queue is delayed. Additionally, processing of deferred objects is further delayed due to external factors such as triggering of interrupts and preemption.

Furthermore, to avoid latency spikes and jitters due to processing of deferred objects, the number of deferred objects processed at a given time is throttled by the synchronization mechanism [21, 35, 40]. Throttling limits the number of deferred objects processed at a time, further delaying the memory reclamation of safe deferred objects.

All these factors result in extended object lifetimes. Existing memory allocators are not aware of safe deferred objects. Hence, they cannot reuse/reallocate these deferred objects until the objects are processed by the synchronization mechanism.

3.3 High object cache and slab churn rate

Bursty freeing of memory and extended object lifetimes, together amplify the performance impact on dynamic memory allocators. Extended object lifetimes result in object cache refill and slab cache grow operations, because memory allocator cannot reuse safe deferred objects until processed by the synchronization mechanism. Bursty freeing of memory, however, results in object cache flush and slab cache shrink. Together they result in high object cache and slab churn rates.

Additionally, throttling the number of deferred objects processed at a given time is completely oblivious of the memory allocator state. Ill-timed throttling induces poor memory allocator performance. For instance, the synchronization mechanism may (i) limit the number of deferred objects processed when the object cache is empty causing object cache refill (ii) process more deferred objects when the object cache is full leading to object cache flushing. Hence, throttling may cause unnecessary object cache refill/flush operations.

Object cache refill/flush and slab cache grow/shrink operations are expensive and impact the performance of memory allocators. Our evaluation in the Linux kernel reveals that the object allocation cost, compared to cache hit, is $4\times$ expensive if it involves object cache refill and $14\times$ expensive if it involves slab cache grow operation. Hence, unnecessary object cache and slab churns are better avoided.

Increasing the size of object cache does not help in avoiding high object churns, instead it causes even higher churn rate. An allocation request that cannot be served from the object cache due to non-availability of objects results in more objects getting refilled in the object cache. If such object cache refill is followed by bursty freeing of objects, the object cache can overflow resulting with more number of objects flushed from object cache. Normally half of the object cache is flushed during the overflow. Moreover, increasing the size of object cache does not eliminate extended object lifetimes. We believe such issues are better handled by having deferred objects visible to the memory allocator.

3.4 Denial-of-service attacks

Extended object lifetimes can be exploited to create denial-of-service attacks by generating a load on the system that triggers high rate of deferred frees. Due to extended object lifetimes, the number of deferred objects waiting for reclamation increases steadily and ultimately exhausts the entire memory in the system. Such DoS attacks have been reported

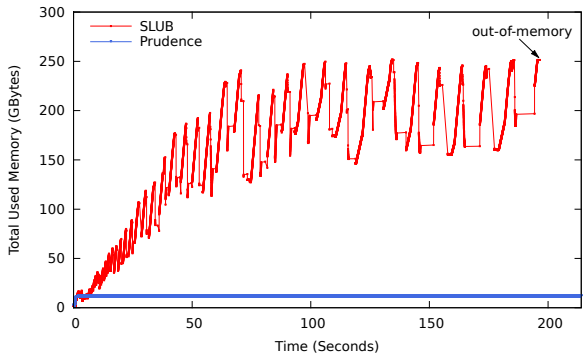

Figure 3. Impact of RCU on the SLUB allocator in the Linux kernel can be seen. Variations in the total used memory incurred by Prudence (up to a few hundred MB) is not noticeable in the graph.

for RCU implementation in the Linux kernel where a malicious user performs file open-close operations in a tight loop to generate high rate of deferred objects [35].

Prudence eliminates extended object lifetimes by having deferred objects visible to the memory allocator. Prudence can reallocate deferred objects immediately after they are eligible for freeing without waiting for processing by an entity external to the memory allocator.

3.5 Evaluating the impact of RCU in the Linux kernel

We evaluate the impact of RCU on the SLUB allocator [10] in the Linux kernel. The test machine is an Intel Xeon E5-4640 processor with 4 CPU sockets, 8 cores per socket for a total of 32 CPUs; with two-way hyper-threading enabled the total number of logical CPUs available is 64. The test machine has 252 GB of physical memory running Linux kernel 3.17.0.

We stress the RCU subsystem by running a workload that continuously performs linked list update operation on all the CPUs. Each CPU updates a different list and hence does not contend for list lock. Every update operation results in the allocation of a new object followed by defer freeing of the old version of the object. We use 512 bytes as object size and sample the total used memory in the system every 10 ms. The result is plotted in Figure 3.

The total used memory in the system increases during the update operation as several object allocation requests result in slab cache growth leading to allocation of pages from the page allocator. This is followed by a steep drop in the total used memory due to bursty free of objects triggered after the completion of grace periods. Free burst increases the number of free slabs resulting in slab cache shrinking leading to returning of pages to the page allocator.

Heavy slab churns are clearly evident from Figure 3 with spikes of slab cache grow and shrink operations. However, Figure 3 does not reveal heavy object churns. This is because flushing/refilling of object cache does not affect the total used memory in the system unless they result in slab cache grow/shrink.

Even though the rate at which the objects, both allocated and deferred free are the same, the total used memory in the system gradually increases and ultimately the system hits out-of-memory (OOM) at the 196th second. Increase in the total used memory is due to extended object lifetimes as RCU is unable to keep up the rate of processing the deferred objects resulting in many objects waiting for reclamation. Existing RCU implementation in the Linux kernel expedites the processing of deferred objects when the system is under memory pressure. This can be observed at around 70 seconds when RCU attempts to process more deferred objects as the memory pressure increases. Despite this, RCU fails to keep up with the rate at which objects are deferred for freeing resulting in OOM.

Ideally, the total used memory should remain the same as the rate at which objects, both allocated and deferred free are the same. Prudence achieves near-ideal performance which is explained in §5.5.

3.6 Hints about the future

We look at the deferred frees from the perspective of providing hints about the memory regions that are about to be freed. Such hints enable dynamic memory allocators to know the future free operations in advance. Moreover, the hints are precise and bound to happen, unlike predictions. Intuitively any hint about the future is valuable. We believe that the hints about the future can be exploited to enhance the performance of dynamic memory allocators.

Prudence demonstrates that optimizations based on hints are viable by exploiting the hints during important state transitions.

4. The Prudence Dynamic Memory Allocator

Prudence is a slab-based [8] dynamic memory allocator integrated with procrastination-based synchronization mechanism. The basic design principle of Prudence is to have deferred objects visible and processed in the memory allocator.

In order to equip memory allocators with the capability to observe and reclaim deferred objects, the memory allocator should (i) provide an interface to other subsystems to defer the freeing of an object, (ii) be capable of identifying the safe time to reclaim the deferred objects' memory.

To incorporate requirement (i) Prudence exports a new API to defer the freeing of an object. The code snippet in Listing 2 shows how to asynchronously defer free an object using Prudence. The new API is a simple *turnkey* replacement for the existing API provided by synchronization mechanisms. With the new API, subsystems can request Prudence to reclaim the deferred objects instead of registering a callback function with the synchronization mechanism. In addition, Prudence allows subsystems to issue regular allocation and free operations along with the deferred free.

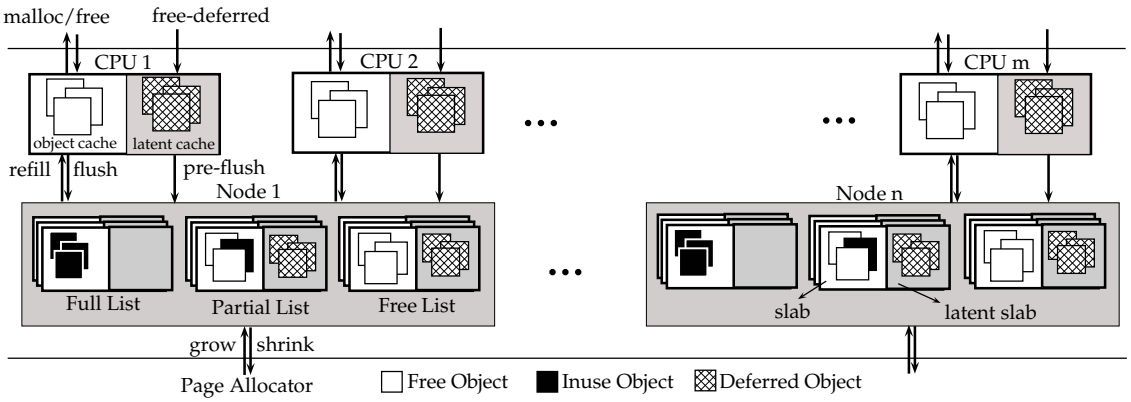

Figure 4. An individual slab cache in Prudence memory allocator

```
writer(...) {
  ...
  old_object = list_update(...);
  free_deferred(old_object);
  ...
}
```

Listing 2. Defer freeing an object in Prudence

For requirement (ii) we tightly integrate the synchronization mechanism with the memory allocator. Procrastination-based synchronization mechanisms employ different memory reclamation schemes to identify the completion of pre-existing readers referring to a deferred object. Generally, to identify the grace period, these schemes maintain the state of the object (active or deferred free) along with the state of the readers. We modify the synchronization mechanism to provide information on the grace period state to the memory allocator and further integrate such information with the deferred objects. Prudence identifies the safe time to reclaim a deferred object depending on whether the grace period is ongoing or completed. However, the synchronization mechanism is still responsible for computing the grace period.

4.1 Latent Cache and Latent Slab

Prudence introduces latent caches and latent slabs to efficiently handle deferred objects and to effectively exploit hints about the future. Both latent caches and latent slabs hold deferred objects that are not yet eligible for reuse/reclamation. A latent cache is defined for every object cache and holds deferred objects at object cache level. Similarly, a latent slab is defined for every slab and holds deferred objects belonging to that slab as shown in Figure 4. Objects in latent caches and latent slabs remain hidden until the completion of the grace period. After the completion of the grace period, objects in the latent cache are merged with the object cache and objects in the latent slab are merged with the slab.

Prudence enforces a limit on the size of a latent cache. The limit is set to the size of the object cache. Once the limit is reached, subsequent objects deferred for freeing are placed into the latent slab. The rationale for setting such a limit is a proactive measure to avoid object cache overflow when the safe deferred objects in latent cache are merged with the object cache. The limit is not applicable to a latent slab as it holds only the deferred objects that belong to the slab. Hence the size of the latent slab cannot exceed the size of the slab.

Latent cache/latent slab completely eliminates the extended object lifetimes because there is no delay incurred in reclaiming the deferred objects. The deferred objects are tracked and processed in the memory allocator and are considered for reallocation immediately after the completion of the grace period. They need not wait for processing from an entity external to the memory allocator.

4.2 Exploiting hints about the future

Information on the objects that are about to be freed is available to Prudence from latent caches/slabs. Prudence uses this information during cache refill and flush operations to avoid performance impact. In addition, Prudence employs optimizations that take advantage of hints about the future to improve memory allocator attributes. This section describes the optimizations employed by Prudence along with the pseudo-code (presented in Algorithm 1) for important functions.

Object cache refill: In cases when an allocation request cannot be served from an object cache, Prudence checks for deferred objects in the latent cache that have waited for a grace period. Eligible deferred objects, if any, are merged into the object cache and the allocation request is served from the object cache (lines 8-11). Object cache refill is required only when there are no eligible deferred objects in latent cache (line 12).

Prudence considers the number of deferred objects in the latent cache while performing an object cache refill. The object cache is partially refilled if there are deferred objects waiting for the completion of the grace period in the latent cache. For example, if the object cache size is o and there are

Algorithm 1 Prudence

```
1:  function MALLOC(size)
2:      slab_cache ← GET_SLAB_CACHE(size)
3:      obj_cache ← slab_cache[current.cpu].obj_cache
4:      object ← GET_OBJECT(obj_cache)
5:      if object then                                  ▷ Cache Hit
6:          return object
7:      latent_cache ← slab_cache[current.cpu].latent_cache
8:      MERGE_CACHES(obj_cache, latent_cache)
9:      object ← GET_OBJECT(obj_cache)                  ▷ Retry
10:     if object then
11:         return object
12:     return REFILL_OBJECT_CACHE(obj_cache)

13: function REFILL_OBJECT_CACHE(obj_cache)
14:     objs_to_refill ← obj_cache.size − latent_cache.count
15:     LOCK(current.node)
16:     while objs_to_refill > 0 do
17:         if NOT_LIST_EMPTY(node.partial_list) then
18:             Pick a slab S from node.partial_list such that
19:                 the total fragmentation is minimized
20:         else if NOT_LIST_EMPTY(node.free_list) then
21:             Pick the first slab S from node.free_list
22:         else break
23:         REFILL(obj_cache, S, objs_to_refill)
24:         objs_to_refill ← objs_to_refill − S.free_objs
25:     UNLOCK(current.node)
26:     object ← GET_OBJECT(obj_cache)                  ▷ Retry again
27:     if object then     ▷ Successful partial/complete refill
28:         return object
29:     if GROW(slab_cache) then                        ▷ Add more slabs
30:         return GET_OBJECT(obj_cache)
31:     if deferred objects waiting for grace period then
32:         Wait for a grace period and retry allocation
33:     else return 0                                   ▷ Trigger out of memory

34: function FREE_DEFERRED(object)
35:     object.gp_state ← GET_GRACE_PERIOD_STATE()
36:     slab_cache ← GET_SLAB_CACHE(object.size)
37:     obj_cache ← slab_cache[current.cpu].obj_cache
38:     latent_cache ← slab_cache[current.cpu].latent_cache
39:     if latent_cache.count < threshold then          ▷ Fast Path
40:         PUT_OBJECT(object, latent_cache)
41:         if ((obj_cache.count + latent_cache.count)
42:             > obj_cache.size) then
43:             SCHEDULE_IDLE_PREFLUSH(latent_cache)
44:         return
45:     FLUSH(obj_cache)
46:     MERGE_CACHES(obj_cache, latent_cache)
47:     if latent_cache.count < threshold then          ▷ Retry
48:         PUT_OBJECT(object, latent_cache)
49:     else                          ▷ Put object into latent slab
50:         PUT_OBJECT(object, object.latent_slab)
51:         PRE_MOVE_SLAB(object.slab)

52: function PRE_MOVE_SLAB(slab)
53:     LOCK(current.node)
54:     if IS_ON_NODE_FULL_LIST(slab) then
55:         MOVE_TO_NODE_PARTIAL_LIST(slab)
56:     else if slab.allocated = slab.deferred then
57:         MOVE_TO_NODE_FREE_LIST(slab)
58:     UNLOCK(current.node)
59:     if free_slabs > free_limit then SHRINK(slab_cache)

60: function MERGE_CACHES(obj_cache, latent_cache)
61:     for obj in latent_cache do
62:         if GRACE_PERIOD_COMPLET(obj.gp_state) then
63:             MOVE_TO_OBJ_CACHE(obj) ▷ Safe for reuse
64:         if obj_cache.count = obj_cache.size then
65:             break
```

d deferred objects in the latent cache, then $o - d$ objects are refilled (line 14). This avoids overflowing the object cache later when d deferred objects are merged into the object cache.

Latent cache pre-flush: Prudence initiates latent cache pre-flush if it foresees an object cache flush after a grace period. Such cases arise when the number of objects in object cache and latent cache together exceed the size of the object cache (lines 41-43). Pre-flushing moves deferred objects in the latent cache to the respective latent slabs.

Prudence schedules pre-flushing during CPU idle time (inspired by [17]). Utilizing CPU idle time eliminates the interference of pre-flushing with object allocation, free and deferred free calls. However, pre-flushing is not performed if idle cycles are unavailable. In such cases, flushing is performed at the time of merging the latent cache with their object cache (line 45).

The number of deferred objects moved from the latent cache to the latent slabs during pre-flush operation is decided based on (i) the number of objects in object cache and latent cache and (ii) object allocation, free and deferred free rate during recent few grace period intervals. The deferred objects in the latent cache are aggressively pre-flushed when the latent cache is almost full and the allocation rate is lower than the free/deferred free rate. Prudence takes a less aggressive approach by intermittently sleeping or yielding the CPU when the allocation rate is higher than the free/deferred free rate, because higher object allocation rate empties the object cache faster. Pre-flushing is terminated when the number of

objects in object cache and latent cache together falls below the size of the object cache.

In cases where a grace period is completed during the process of pre-flushing, the eligible/safe deferred objects in the latent cache are merged into the object cache and the pre-flushing is continued. This is beneficial as it avoids the merging of deferred objects from latent cache to object cache during an allocation request.

The pre-flushing of different latent caches is likely to spread over an interval of time. Hence, pre-flushing reduces the lock contention on the node list. In current memory allocators, due to bursty freeing of objects, flushing happens in parallel on all CPUs soon after the completion of grace periods causing high contention on the node list lock.

Object cache flush: As already mentioned, Prudence attempts to avoid flushing the object cache by partially refilling the object cache during object cache refill. Further, it pre-flushes the latent cache if it foresees an object cache flush after a grace period. However, object cache flushing cannot be eliminated. For instance, flushing is required when a series of free operations overflows the object cache.

In Prudence, the number of objects flushed during an object cache flush depends on the number of deferred objects in the latent cache (similar to object cache refill technique). More objects are flushed if there are more deferred objects in the latent cache.

Slab pre-movement: Similar to other slab-based allocators [10, 18], Prudence groups the slabs that are full, partially full and free as shown in Figure 4. Prudence performs pre-movement of a slab between full, partial and free lists if it foresees such a movement in the future (lines 54-57). For example, pre-movement of a full slab from the full list to the partial list is performed when an object associated with the full slab is deferred for freeing. Pre-movement avoids contention on the node list lock as movement of slabs is spread over an interval of time. Whereas, in the case of existing memory allocators, slab movement is triggered in parallel on all CPUs after the completion of a grace period due to the bursty freeing of objects causing high contention on the node list lock.

Reduces total fragmentation: Slab-based allocators consume more memory than requested due to unused objects in the object caches and slabs. Total fragmentation f_t measures the excess memory consumed by the allocator and is defined as the ratio of the allocated and the requested memory [6, 8]:

$$f_t = \frac{allocated}{requested} = \frac{slabs\_allocated \times slab\_size}{objects\_requested \times object\_size}$$

Prudence reduces total fragmentation by considering deferred objects in the latent slab at the time of selecting a slab for object cache refill (lines 17-21). We explain the slab selection optimization with the help of an example.

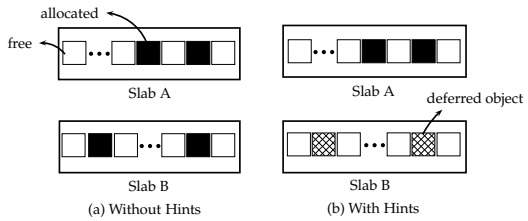

Figure 5. Prudence considers hints on deferred objects during slab selection for object cache refill to reduce total fragmentation [6, 8] and hence selects slab A.

Consider two slabs A and B belonging to a slab cache with two allocated objects in both the slabs as shown in Figure 5(a). It is possible to have slabs in such a state after a couple of allocation and free cycles. We consider this scenario to demonstrate how Prudence chooses a better slab to reduce the total fragmentation when information on deferred objects is available.

For existing memory allocators, with information on the allocated/free objects in a slab, there is no advantage of selecting slab A over slab B or vice versa to refill the object cache. Hence, let us assume slab B is selected for object cache refill.

Now consider this scenario when information on deferred objects is available to the memory allocator as shown in Figure 5(b). Let the slabs A and B be in the same state as before. However, let us assume that the two allocated objects in slab B are deferred for freeing. The memory regions corresponding to these deferred objects will be free after the grace period. With that hint, there is an advantage of selecting slab A over slab B to refill the object cache, because, it is known that the entire slab B will be free after the completion of the grace period. Hence, the pages associated with slab B can be returned to the page allocator if slab A is chosen. This reduces total fragmentation as the number of requested objects remains the same in both the cases, however, the number of allocated slabs will be reduced by one in Prudence.

Furthermore, Prudence does not select a slab for object cache refill when most of the allocated objects in that slab are deferred for freeing (unless it needs to grow the slab cache). This is based on the hope that a few remaining allocated objects will be freed/deferred for freeing in the future. If that happens, the pages associated with the slab can be returned to the page allocator.

Handling memory pressure: Dynamic memory allocators invoke out-of-memory (OOM) handlers to reclaim memory when the system is running low on memory. OOM handlers are disruptive and kill processes to reclaim the memory.

Prudence delays the triggering of OOM handlers if there are several deferred objects waiting for the completion of the grace period (line 31). This can avoid killing of processes to

reclaim the memory as the deferred objects can be reallocated after the grace period.

4.3 Reuse of existing optimizations

Slab allocators have been used for decades in many operating systems, libraries and utilities and are highly optimized for performance based on empirical studies and mathematical models. Prudence reuses such optimizations employed in the existing memory allocators. For example, our implementation of Prudence in the Linux kernel reuses the existing heuristics employed by SLUB allocator to decide the size of the object cache, the size of a slab, the threshold after which the slab shrinking should be considered. Other factors influencing the performance such as the slab coloring scheme to improve the hardware cache utilization are also reused.

5. Evaluation

We implement Prudence in the Linux kernel by tightly integrating with the RCU subsystem. Two new APIs `kfree_deferred()` and `kmem_cache_free_deferred()`[1] are exported by Prudence to enable other kernel subsystems to defer free an object. Kernel subsystems that use RCU are modified to use the new APIs, while the rest of the kernel subsystems continue to use SLUB as the memory allocator.

Prudence, analogous to RCU, does not need to explicitly track each and every deferred object in latent cache or latent slab. All these deferred objects are eligible for reuse/reallocation after the completion of the grace period.

5.1 Experiment setup

The test machine is as detailed in §3.5. A mix of micro, synthetic (Postmark, Netperf) and application benchmarks (Apache, PostgreSQL) have been selected to measure the performance of Prudence.

We compare our results with the SLUB allocator in the Linux kernel. The SLUB allocator is the recent allocator in the Linux kernel based on the slab allocation principle [8] and is the default allocator in the Linux kernel. The SLUB implementation in the Linux kernel has better object management techniques, less metadata footprint and lower total fragmentation compared to the earlier SLAB allocator in the Linux kernel [28]. SLUB also performs better on large multicore systems compared to the SLAB allocator [28].

5.2 Micro Benchmark

We execute `kmalloc()`/`kfree_deferred()` operations in a tight loop, for different object sizes, on all CPUs and measure the total number of allocation/deferred free operations executed per second. A total of 5 million pairs of allocation and deferred free operations are executed per CPU per object size. The benchmark is run thrice and we report the average value across three runs along with the standard deviation.

[1] `kmem_cache_free_deferred()` requires slab cache identifier as an additional argument [8, 18]

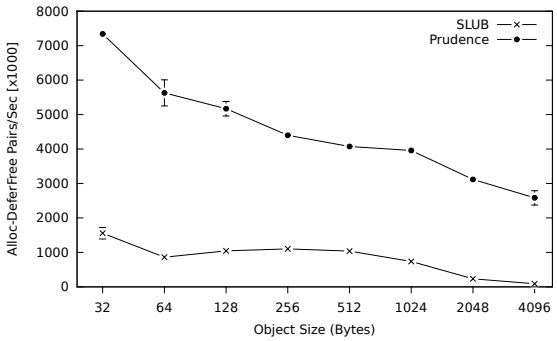

Figure 6. `kmalloc()`/`kfree_deferred()` pairs of operations executed per second for different allocation size requests

Prudence performs 3.9× to 28.6× better than SLUB on the number of `kmalloc()`/`kfree_deferred()` pairs executed per second as shown in Figure 6. The improvement is due to eliminating extended object lifetimes and avoiding unnecessary object cache and slab churns. The improvement is higher for larger objects, with the highest improvement of 28.6× achieved for 4096 bytes object size. Larger objects are normally optimized for memory efficiency, hence have fewer objects in object cache and smaller slabs. Fewer objects in object cache result in more frequent object cache refill/flush and smaller slabs result in frequent slab churns. Hence, avoiding unnecessary object cache refill/flush and slab churns results in better performance for larger objects.

5.3 Synthetic and application benchmark details

Postmark-1.51: Postmark [26] simulates the behavior of a mail server by executing file read, append, create and delete operations stressing the underlying filesystem. We execute 64 instances of Postmark on an ext4 filesystem. Each instance operates on a pool of 500 files and 100 directories.

Netperf-2.6.0: The Netperf [25] benchmark measures the performance of network. We execute TCP Connect/Request /Response (TCP_CRR) test on localhost.

Apache-2.4.6: We use ApacheBench (`ab`) [14] part of Apache HTTP server [15] to benchmark the Apache server. The benchmark measures the number of requests the Apache server handles per second. We execute `ab` on localhost with 128 parallel instances.

PostgreSQL-9.2.7: We benchmark PostgreSQL [20] using `pgbench` [19]. The `pgbench` tool, part of PostgreSQL, performs a sequence of SQL operations in multiple concurrent database sessions generating stress similar to TPC-B. We run `pgbench` with 64 threads and 128 clients on localhost.

All the benchmarks execute a fixed number of transactions/jobs on a fresh boot. The number of transactions/jobs is chosen such that each run executes for 5-10 mins. We repeat the run three times and present the average of three runs

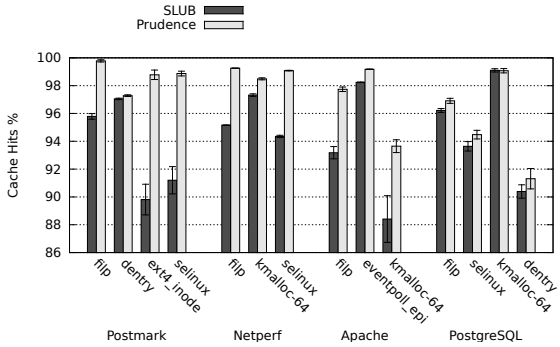

Figure 7. Improvement in the percentage of allocation requests served from the object cache

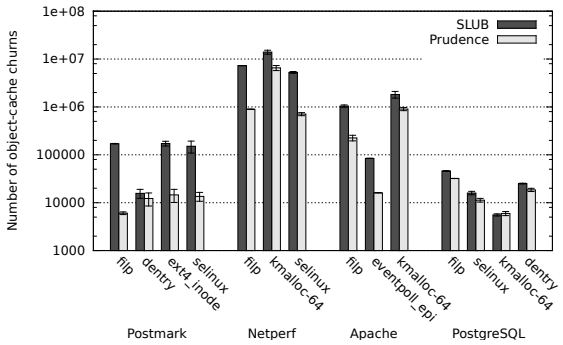

Figure 8. Number of object cache churns incurred by different slab caches

along with the standard deviation. Fixed number of transactions/jobs executed for every benchmark enables a fair comparison of absolute numbers of the memory allocator attributes.

We use the same names for the slab cache as used in the Linux kernel here. The `filp` slab cache holds objects for file descriptor table, `eventpoll_epi` holds objects related to Linux specific poll() system call [12], `dentry` holds directory entry objects, `ext4_inode` holds inodes of ext4 filesystem and `selinux` holds Security Enhanced Linux (SELinux) related objects. We use `kmalloc-64` to represent kmalloc slab cache of size 64 bytes.

We report the performance of the slab caches which have performed more than a million allocation and deferred free operations per run. Each benchmark stresses different slab caches depending on the type of operation/transaction performed. For example, the `ext4_inode` slab cache is stressed a lot while running the Postmark benchmark, however, it has fewer than a few hundred allocations and deferred free requests when running Netperf.

5.4 Results

In this section we report the performance of Prudence.

Object cache hits: Prudence performs better in terms of cache hits for all the slab caches as shown in Figure 7. Improvement in cache hits is due to better management of deferred objects through latent cache. The latent cache ensures immediate availability of deferred objects in the object cache for subsequent allocation requests after the completion of a grace period. This results in serving more requests from the object cache, improving cache hits. In the SLUB allocator, deferred objects are not immediately available for allocation after the completion of a grace period and hence it frequently runs out of objects in object cache requiring to refill the object cache.

Furthermore, when object cache overflows, Prudence flushes only the required number of objects by considering the number of deferred objects in the latent cache. This avoids unnecessary flushing of more objects from the object cache increasing cache hits.

Object cache churns: Prudence reduces the number of object cache churns (pairs of object cache refill/flush operations) by 25.97% to 96.47% compared to the SLUB allocator (except for PostgreSQL `kmalloc-64` which increased by 6%) as shown in Figure 8. Improvement in churn rate is due to avoiding unnecessary refill and flush operations caused by bursty freeing of objects together with extended object lifetimes. If Prudence foresees an object cache flush due to bursty freeing, then subsequent deferred objects are moved to latent slabs. Extended object lifetimes is completely eliminated using latent caches reducing unnecessary object cache refills. Furthermore, improvement in churn rate is due to avoiding unnecessary object cache flushes by performing partial refill of object cache. In addition, more objects are flushed during object cache overflow by considering deferred objects in the latent cache.

Increase in object cache churn rate in the case of PostgreSQL `kmalloc-64` is due to PostgreSQL triggering several free operations outside the context of deferred frees on the `kmalloc-64` slab cache. These free operations interfere with the decisions taken by Prudence resulting in more object cache churns.

Slab churns: Figure 9 shows the number of slab churns (pairs of slab grow/shrink operations) performed by Prudence and SLUB during benchmark execution. Prudence reduces the number of slab churns by 21% to 98.3% except for Postmark `dentry`. Although the slab churn rate for `dentry` is reduced by 3.1%, the reduction is not as high as compared to other slab caches. The number of slab churns for Netperf `filp` is reduced drastically from 364K in SLUB to 6K in Prudence.

Improvement in slab churn rate is due to avoiding unnecessary slab grow/shrink operations caused by bursty freeing of objects together with extended object lifetimes. Further, reduction in the number of object cache churns also reduces the number of slab churns.

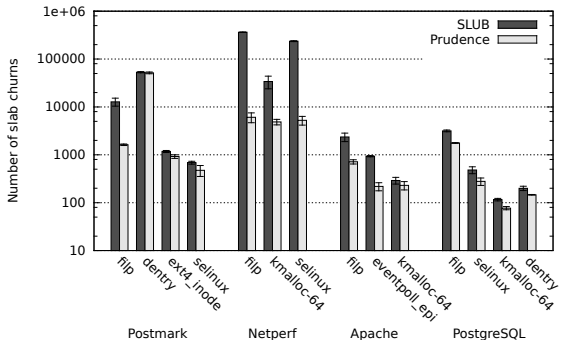

Figure 9. Number of slab churns incurred by different slab caches

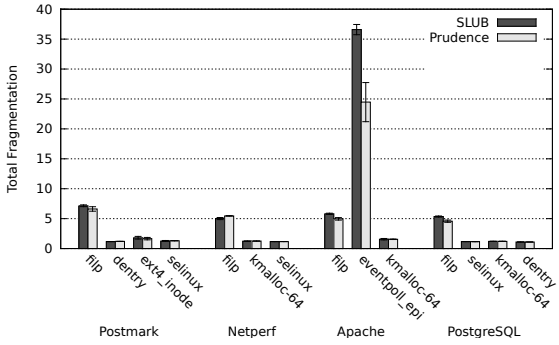

Figure 11. Total Fragmentation [6, 8] measured after the completion of each run for different slab caches

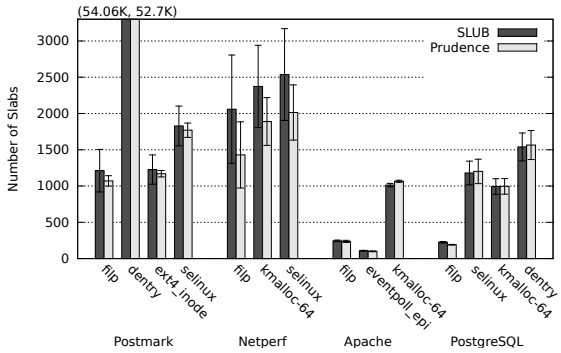

Figure 10. Peak slab usage for different slab caches

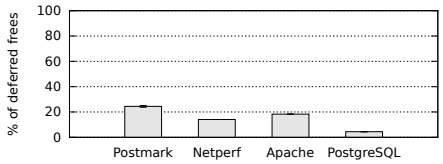

Figure 12. Percentage of deferred frees out of total memory free operations

Peak slab usage: Figure 10 shows the peak number of slabs allocated by Prudence and SLUB. Peak slab usage also indicates the maximum memory consumed by respective slab caches during the execution of the benchmark. Prudence reduces the peak slab usage by 2.5% to 30.6% for most of the slab caches and performs equally well ($\pm 2\%$ change) for others compared to SLUB (except for Apache `kmalloc-64` which increased by 5%). Reduction in the peak slab consumption is due to eliminating extended object lifetimes. In Prudence, deferred objects are immediately available for reuse/reallocation after the completion of grace period which avoids unnecessary slab grow operations.

The intensity of slab churns is clearly evident from the peak slab usage. For example, in the case of Netperf `filp`, the peak number of slabs allocated is 2060, however, a total of 364K slab churns are incurred in SLUB. In Prudence, the peak slab usage for Netperf `filp` is reduced to 1429 and the number of slab churns incurred is only 6K.

Total fragmentation: We measure the total fragmentation after the completion of each run (as followed in [8]). Prudence reduces fragmentation for many slab caches (7% to 33% reduction) and performs equally well ($\pm 2\%$ change) for the other slab caches compared to the SLUB allocator as shown in Figure 11. However, the total fragmentation is increased by 8.7% for the Netperf `filp` slab cache.

The SLUB allocator selects the first slab in the node partial list to refill the object cache. Prudence, as mentioned already, selects a slab considering the number of deferred objects in the latent slab. The number of slabs in the node partial list can be high and traversing the entire list to select a slab increases refill latency. Hence, Prudence traverses the first 10 slabs in the node partial list reducing the opportunity to improve total fragmentation. This is a trade-off between improving total fragmentation and reducing latency.

Throughput: Overall performance improvement depends on how much an application exercises the memory allocator [6–8, 13]. Prudence is designed to optimize deferred frees and hence the number of frees occurring in the context of RCU deferred frees indicates the opportunity for optimization. Figure 12 shows the percentage of deferred frees out of the total free operations triggered by each benchmark. The percentage of deferred frees varies from 4.4% for PostgreSQL to 24.4% for Postmark.

Figure 13 shows the average throughput improvement by Prudence compared to the SLUB allocator. Postmark achieves the highest performance improvement of 18%. Objects in different slab caches such as `filp`, `ext4_inode` are deferred for freeing during file deletions and are allocated during file creation. The improvement in throughput is due to better handling of deferred objects during file creation and deletion. Netperf and Apache respectively trigger 14% and

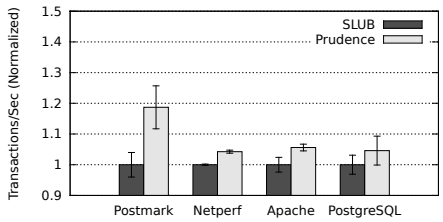

Figure 13. Improvement in overall performance of Prudence compared to SLUB

18% of deferred frees over total free operations and achieve 4.2% and 5.6% improvement in performance. In both the benchmarks, objects are deferred for freeing during connection tear town. Additionally, in case of Apache, objects are deferred for freeing during the removal of the target file descriptor from epoll [12] instance. PostgreSQL achieves 4.6% improvement in performance. However, the variation in the throughput is high for both SLUB and Prudence.

In all the benchmarks the SELinux subsystem allocates and defer frees `selinux` objects during creation and deletion of various resources such as files and sockets.

5.5 Endurance

We discuss the impact of RCU on Prudence for the experiment mentioned in §3.5. It is evident from Figure 3 that Prudence does not suffer from high slab churns or extended object lifetimes.

The total memory used in Prudence increases initially because the initial set of objects deferred for freeing are available for reallocation only after waiting for a grace period. However, Prudence hits equilibrium once the rate at which deferred objects are eligible for reallocation reaches the rate at which objects are allocated. Hence, the total used memory after the initial increase remains almost the same. Prudence still incurs slab churns, however they are fewer in number. These slab churns are not noticeable in Figure 3 because of small variations in the total used memory. Furthermore, Prudence is not susceptible to denial-of-service attacks as it eliminates extended object lifetimes.

6. Related Work

Previous work attempts to optimize memory reclamation in the synchronization mechanism only. The RCU implementation in the Linux kernel employs such optimizations which include offloading the reclamation of deferred frees to kernel threads [11] and throttling the number of objects reclaimed based on the backlog [40]. TRASH [39] also employs throttling to reclaim deferred objects based on backlog. Masstree [30] and CITRUS [1] mention that a technique similar to RCU is employed for memory reclamation. However, further details on memory reclamation are not provided. Silo [43] performs memory reclamation either in the main thread or in a background task. However, none of these mechanisms exploit the hints provided by deferred objects on the future free operations.

A completely orthogonal work predicts the object lifetimes to enable optimization in the memory allocator [3, 24]. However, predictions are not precise and can go wrong. With Prudence, hints provide precise information on the memory regions that are about to be freed and such frees are guaranteed to happen. Prudence can be further optimized based on predictions of object lifetimes for objects that are freed outside of the deferred free context.

7. Conclusion

We introduce Prudence, a dynamic memory allocator designed to handle memory reclamation of deferred objects by tightly integrating it with procrastination-based synchronization mechanism. Prudence avoids poor memory allocator performance induced by synchronization mechanisms by properly timing the memory reclamations. It also employs novel optimizations exploiting hints about the future and further demonstrates that such optimizations improve the performance of memory allocators.

Acknowledgments

We would like to thank our shepherd Emmett Witchel, the anonymous reviewers, Paul McKenney, Sorav Bansal, Murali Krishna Ramanathan and Vinayaka Pandit for their constructive comments which helped to improve this paper. Many thanks to Akshatha Deshpande, Sunita Rao, Ananth Narayan, Arpith K Sharma, Ashish Panwar and Madhuri Prasad.

Disclaimer: The views in the article are solely of the authors and not of their employers.

References

[1] Maya Arbel and Hagit Attiya. Concurrent updates with rcu: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM.

[2] Andrea Arcangeli, Mingming Cao, Paul E McKenney, and Dipankar Sarma. Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 297–309, 2003.

[3] David A Barrett and Benjamin G Zorn. Using lifetime predictors to improve memory allocation performance. In *ACM SIGPLAN Notices*, volume 28, pages 187–196. ACM, 1993.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on*

Operating Systems Principles, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[5] Andrew Baumann, Jeremy Kerr, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *6th Linux. Conf. Au*, 2005.

[6] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[7] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Oopsla 2002: reconsidering custom memory allocation. *ACM SIGPLAN Notices*, 48(4S):46–57, 2013.

[8] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[9] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM.

[10] Jonathan Corbet. The SLUB allocator. `http://lwn.net/Articles/229984/`, 2007.

[11] Jonathan Corbet. Relocating rcu callbacks. `http://lwn.net/Articles/522262/`, 2012.

[12] Jonathan Corbet. Epoll evolving. `https://lwn.net/Articles/633422/`, 2015.

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

[14] The Apache Software Foundation. Apache HTTP server benchmarking tool. `https://httpd.apache.org/docs/2.2/programs/ab.html`, 2015.

[15] The Apache Software Foundation. Apache HTTP Server Project. `http://httpd.apache.org/`, 2015.

[16] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI*, volume 99, pages 87–100, 1999.

[17] Richard Golding, Peter Bosch, John Wilkes, USENIX Association, et al. Idleness is not sloth. In *USENIX*, pages 201–212, 1995.

[18] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall, 2004.

[19] The PostgreSQL Global Development Group. pgbench. `http://www.postgresql.org/docs/devel/static/pgbench.html`, 2015.

[20] The PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org/`, 2015.

[21] Dinakar Guniguntala, Paul E McKenney, Josh Triplett, and Jonathan Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, 2008.

[22] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[23] Philip W Howard and Jonathan Walpole. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience*, 2013.

[24] Hajime Inoue, Darko Stefanović, and Stephanie Forrest. On the prediction of java object lifetimes. *Computers, IEEE Transactions on*, 55(7):880–892, 2006.

[25] Rick Jones. NetPerf. `http://www.netperf.org/`, 2012.

[26] Jeffrey Katcher. Postmark: A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, 1997.

[27] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, September 1980.

[28] Christoph Lameter. SLUB: The unqueued slab allocator. `http://lwn.net/Articles/229096/`, 2007.

[29] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 219–230, Philadelphia, PA, June 2014. USENIX Association.

[30] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European conference on Computer Systems*, pages 183–196. ACM, 2012.

[31] Paul E McKenney. *Exploiting Deferred Destructions: An Analysis of Read-Copy-Update Techniques in Operating System kernels*. PhD thesis, Oregon Health & Science University, 2004.

[32] Paul E McKenney. Structured deferral: synchronization via procrastination. *Communications of the ACM*, 56(7):40–49, 2013.

[33] Paul E McKenney. RCU Linux usage. `www.rdrop.com/users/paulmck/RCU/linuxusage.html`, 2014.

[34] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[35] Paul E McKenney, Dipankar Sarma, Ingo Molnar, and Suparna Bhattacharya. Extending RCU for realtime and embedded workloads. In *Ottawa Linux Symposium, pages v2*, pages 123–138, 2006.

[36] Paul E McKenney, Dipankar Sarma, and Maneesh Soni. Scaling dcache with RCU. *Linux Journal*, 2004(117):3, 2004.

[37] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[38] James Morris. Have You Driven an SELinux Lately? In *Linux Symposium Proceedings*, 2008.

[39] Robert Olsson and Stefan Nilsson. Trash a dynamic lc-trie and hash data structure. In *High Performance Switching and Routing, 2007. HPSR'07. Workshop on*, pages 1–6. IEEE, 2007.

[40] Dipankar Sarma and Paul E McKenney. Making RCU safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)*, pages 182–191, 2004.

[41] Josh Triplett, Paul E McKenney, and Jonathan Walpole. Scalable concurrent hash tables via relativistic programming. *ACM SIGOPS Operating Systems Review*, 44(3):102–109, 2010.

[42] Josh Triplett, Paul E McKenney, and Jonathan Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *USENIX Annual Technical Conference*, page 11, 2011.

[43] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[44] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43:76–85, April 2009.