A Lightweight Log-based Deferred Update for Linux Kernel Scalability

Your N. Here Second Name Name
Your Institution *Name Institution*

Abstract

1 Introduction

최근 코어수가 증가하고 있다. 이처럼 100개 이상 코어수가 증가한 매니코어 환경에서 리눅스 커널에 대한 확장성 문제가 있다. 확장성 문제 중 하나가 락 경합 때문에 발생하는 serialization 문제이다. 여러 락 serialization 문제 중 하나가 update operation이기 때문에 발생하는 문제가 있다. 그 이유는 pdate operation는 동시에 수행되지 못하기 때문이다. 리눅스 커널의 reverse page mapping이 update rate가 높은 구조로 되어 있다. 따라서 리눅스 커널의 rmap update lock에 때문에 serialize 되어 scalability 문제가 있다.

이처럼 update serialization 문제를 해결하기 위해 여러 concurrent updates 방법[][][] 들이 연구되고 있다. 이러한 concurrent updates 방법들은 워크로드 특성인 update ratio에 따라 많은 성능 차이를 보인다. 이 중 high update ratio를 가진 커널의 rmap과 같이 update-heavy한 data structure일 경우, 이를 해결하기 위한 방법 중 하나는 log-based 알고리즘[][][]을 사용하는 것이다. log-based 알고리즘은 이며, 마치 CoW와 유사한 방법을 사용한다. Silas Boyd-Wickizer ,. update heavy한 구조를 timestamp 기반의 per-core log를 활용한 방법을 제안하였다. timestamp 기반의 per-core log를 활용한 concurrent updates 방법은 update 부분만 고려했을 때 굉장히 좋은 scalability를 가지고 있다. 하지만 timestamp 관리 비용에 대한 문제가 있다. 다시 말하면, core가 늘어 날 수로 timestamp 관리하는 비용기 커진다. 예를 들어 코어 수가 1000개로 늘어 날 경우 log를 merge하는 reader가 각 코어에 쌓인 log를 시간 순서로 정렬하고 관리하는 비용이 커진다. 또한 per-core로 log를 저장하는 공간에 대해서도 고려해야한다. OpLog는 per-core hash table로 root 포인터를 가지고 있는 object의 log를 bucket단위로 저장하는데, 만약 root의 object 많은 경우, 잦은 hash 충돌로 인한 문제가 발생한다.

본 논문은 high update rate를 가진 data structure를 위한 새로운 방법(LDU)을 제안하였다. LDU는 timestamp를 이용함에 따라 생기는 복잡성을 해결하기 위해, 조금 더 simple하고 lightweight한 방법으로 접근하여 update-heavy한 data structure를 위한 concurrent updates 방법을 제안하였다. LDU는 OpLog와 같이 분산 시스템에서 사용하는 log기반의 concurrent updates 방식과 최소한의 shared-memory system의 atomic 연산을 이용하여 이 문제를 해결 하였다. LDU도 역시 log-based 알고리즘이 가지고 있는 4가지 장점을 가진다. lock-less concurrent update: lock 필요없으므로, lock에 대한 오버헤드가 없다. cache : corse graind lock(FC), single core에서 log의 operation을 수행하므로 캐시 히트율이 높다. Simpler and easier: 다른 알고리즘(트리, 리스트, 스택, 큐)에 쉽게 적용이 가능. optimization : update-side absorbing, reuse garbage object practical system인 intel xeon 120코어를 대상으로 리눅스 커널 fork scalability 개선

LDU를 리눅스의 update heavy한 data structure 때문에 fork scalability 문제를 발생시키는 2가지 anon, file reverse page mapping에 적용하였다. 실험 결과 및 발견된 결과 * 120core에서 2x 성능 향상

새로운 simple하고 lightweight한 log-based concurrnet updates 방법 제안: practical system인 intel xeon 120코어를 대상으로 리눅스 커널 fork scalability 개선 :

This paper is organized as follows. Section 2 summarizes related works and compare our contributions to previous works. Section 3 describes the design of the LDU algorithm and Section 4 explains how to apply to Linux kernel. section 5 explains our implementations in Linux and Section 6 shows the results of the experimental evaluation. Finally, section 7 concludes the paper.

2 Background and Problem

3 LDU Design

LDU는 리눅스 커널의 high update rates를 가진 data strcuture의 Scalability 위한 새로운 로그 기반의 위한 Concurrent Updates 방법이다. LDU는 timestamp를 이

용하여 발생하는 log를 관리에 대한 어려움을 해결하였다. time-stamp를 사용하지 않기 위해 LDU는 최소한의 shared memory system atomic 연산을 사용하도록 하여 설계하였다. 이러한 LDU의 기본적인 철학은 distributed system에서 주로 사용하는 time-stamp log기반 방식의 concurrent updates 방법과 shared memory system에서만 사용할 수 있는 CAS와 같은 atomic operation을 절묘하게 결합하여 설계하였다. 즉 저장된 log를 consistency를 유지하기 위해 최소한의 atomic operation을 사용하도록 설계하였다. LDU는 cache invalidate 줄이기(mitigating)을 최소화 하기 위해 3가지 방법(light weight queue, Update-side Abosrbing, reusing garbage object)을 병행 하였다. This section explains these algorithmic design aspects of LDU.

3.1 Log-based Concurrent updates

Update heavy한 구조 때문에 발생하는 scalability 문제에 대한 해결 책 중 하나는 Log기반의 알고리즘을 사용하는 것이다. Log-based 알고리즘은 lock을 피하기 위해 update가 발생하면, data structure의 update operation(insert or remove)을 argument와 함께 저장하고, 주기적 또는 before a read operation에 applies the updates in all the logs to the data structure, so reader can read up to date data structure. Log-based 방법은 마치 CoW(Copy on Write)와 같이, read 전에 저장된 log가 수행됨으로 read가 간혈적으로 수행되는 data structure에 적합한 방법이다.

Log-based 방법은 총 4가지의 장점을 가진다. 첫째로, Update가 수행하는 시점 로그를 저장하는 순간에는 Lock이 필요가 없다. 둘째로, 저장된 Log를 corse-grain Lock과 함께 하나의 코어에서 수행하므로 cache에 효율성이 높아진다. 또한 OpLog와 같이 log를 per-core로 저장하면, cache invalidate에 대한 오버헤드를 최소화 할 수 있다. 셋째로, 기존 여러 데이터 structure에 쉽게 적용할 수 있는 장점이 있다. 마지막으로 저장된 Log를 실제 수행하지 않고, 여러가지 optimization 방법을 사용하여 적은 operation으로 Log를 줄일 수 있다.

3.2 Approach

LDU도 log-based approach를 가진다 그러므로 log-based 방법의 장점을 모두 가진다. LDU는 먼저 lock 없이 Concurrent Updates를 위해 수행 시점에서는 Atomic하게 Operation Log를 Global queue에 저장한다. Lock이 필요 없어 scalability가 뿐만 아니라 Lock 자체의 오버헤를 줄일 수 있다. Log를 저장하는 저장된 Log는 주기적으로 또는 read 전에 호출되게 되는데, LDU는 FC와 같이 operation Log를 corse grain lock과 함께 single core에서 수행하므로 cache의 miss를 줄여 성능을 향상시킨다. 또한 LDU는 Log-based 방법과 같이 기존 data structure를 많이 수정하지 않아 다른 data structure에 쉽게 적용할 수 있다. LDU가 global queue를 사용함에 따라, CAS operations on the head of the queue limit scalability(the bottleneck on the log is similar to that the Michael and Scott queue) 이처럼 LDU는 global queue의 사용에 대한 cache invalidate 줄이고, 불필요한 연산을 줄이기 위해, CAS를 최소화한 queue를 사용, update-side absorbing, reusing garbage object 등과 같이 3가지 optimization을 수행하였다.

LDU는 global queue의 head 포인터에 대한 CAS 연산을 최대한 줄이기 위해 log를 저장하고자 하는 큐를 multiple producers and single consumer에 기반하는 non-blocking queue를 이용하였다. 이 큐는 다른 non-blocking list와 다르게 insert operation을 head의 first 노드에 삽입함에 따라, CAS가 발생되는 횟수가 상대적으로 적다. 게다가 remove를 위한 복잡한 알고리즘이 필요없다. 그 이유는 저장된 log를 수행하는 single consumer는 head pointer를 NULL로 xchg 명령을 사용하여 atomic하게 제거한다. 따라서 remove를 위한 복잡한 알고리즘이 필요없다. 이러한 큐는 이미 Linux 커널에 Lock-less list라는 이름으로 구현되어 있으며, 이미 커널에 많은 부분에 사용되고 있다. LDU도 Linux 커널에 있는 Lock-less list를 활용 하였다.

object에 대한 순서가 만약 Insert와 remove가 변경된다면, 문제가 발생한다. If a process logs an insert operation on one core, migrates to another core, and logs a remove operation, the remove should eventually execute after the insert[OpLog].

LDU는 최적화를 위해 update-side absorbing이라는 방법을 사용한다. Update operation는 만약 같은 object에 대해서 insert와 remove가 발생하였으면, 같은 object에 대해서는 insert oepration과 remove operation은 수행을 안해도 되는 operation이다. 이러한 방법은 log 기반으로 deferred 로 수행하기 문에 가능한것인데, log로 저장을 했기 때문에 삭제가 가능한 operation을 삭제하여 성능을 높이는 방법이다. OpLog는 이러한 Log 삭제를 같은 코어서 동작하면 적은 operation으로 바로 바로 삭제를 하나 다른 코어로 이동된 Log 같은 경우 삭제를 위해 검색을 해야하는 추가적인 작업을 수행하는 문제가 있다. LDU는 Log를 줄이는 작업에는 shared memory system의 atomic operation을 사용하였다. LDU는 모든 object에 insert와 remove의 mark 필드를 추가해서 global head 포인터에 비싼 CAS연산을 사용을 줄이는 방법을 사용한다. 이것은 삭제가능한 operationg일 경우 CAS를 사용하여 로그로 저장하지 않고 update를 수행할 때 SWAP operation을 활용해 바로 바로 지워주는 알고리즘을 사용하였다. 예를 들어 만약A라는 object를 대상으로 insert-remove operation이 수행될 경우 처음 insert operation은 insert mark 필드에 표시하고 queue에 저장한다. 다음 remove operation 부터는 log를 queue에 저장하지 않고 insert에 표시한 mark 필드에 표시한 값만 지워주는 방식으로 사용한다.

이것은 상대적으로 가벼운 연산만 사용해서 log를 지워주는 효과를 가져주므로 성능 향상을 준다.

LDU의 또 다른 최적화 기법은 update-side absorbing 때문에 취소된 garbage object를 재활용하는 것이다. 이것은 queue에는 존재하만 update-side aborsrbing 때문에 취소 되어 garbage가 되어 queue에 보관되어 있는 log일 경우, 다음 update operation에 대해서는 로그를 새로 많들어 넣지 않고 기존 로그를 재활용하는 방법이다. 예를 들어 같은 오브젝트에 대해서 insert-remove-insert 순서로 update가 수행될 경우에는 세번째 insert 명령어는 global operation list에 들어가 있지만 remove가 발생하여 mark filed가 zero인 경우(garbage object)이다. 이러한 경우 다음 insert operation에 대해서 비싼 연산인 list에 연산을 다시 넣지 않고, 해당 object의 mark 필드만 변경하여 log를 재활용하는 방법을 사용하였다.

To keep the log from growing without end, log-based method periodically apply the time-ordered operations at the head of the log to remove these opearation from the log. LDU는 주기적으로 Log를 적용함으로 Log가 쌓여서 발생하는 메모리 낭비를 줄인다.

LDU는 2가지 limitation을 가지고 있다. 첫번째로는 리눅스 커널과 같이 search가 two update operations(one to insert and one to remove an element) 함수 외부해서 수행하는 data structure에만 적용이 가능하다. 즉 다시말하면 리눅스의 data structure의 경우에는 search와 update가 분리되어 있어 같은 update operation이 연달아 호출되지 않는다. 즉 이러한 구조는 search가 외부에 있기 때문에 insert operation 다음에는 반드시 remove opration이 발생하고 remove operation 다음에는 반드시 insert operation이 발생하는 특징이 있다. 이와 반대로 research 분야에서 많이 연구되는 CSDS(Concurrent search data structures)[]의 방법들은 search가 update operation 내부에 존재하는 구조로 되어 있어서 같은 노드에 대해서 insert-insert 또는 remove-remove 순서로 동작할 수 있다. 이러한 경우에는 LDU를 적용하지 못한다. 하지만 CSDS 구조가 아닌 data structure를 사용하여

3.3 LDU example

3.3.1 GLDU example

Figure 2 gives an example of deferred update with six update operations and one read operation. In this figure, execution flows from top to bottom. The data structure for *physical update* is a tree, and initial values in the tree are node `A` and `B`. In contrast, the data structure for *logical update* is lock-less list. In the top figure, `Core0`, `Core1` and `Core2` perform the logical insert operation to nodes `C`, `D` and `E`, respectively. The logical inserts set the insert mark, and they then insert their nodes into lock-less list. In this case, none of the lock is needed because LDU uses the lock-less list;all threads can execute the update con-

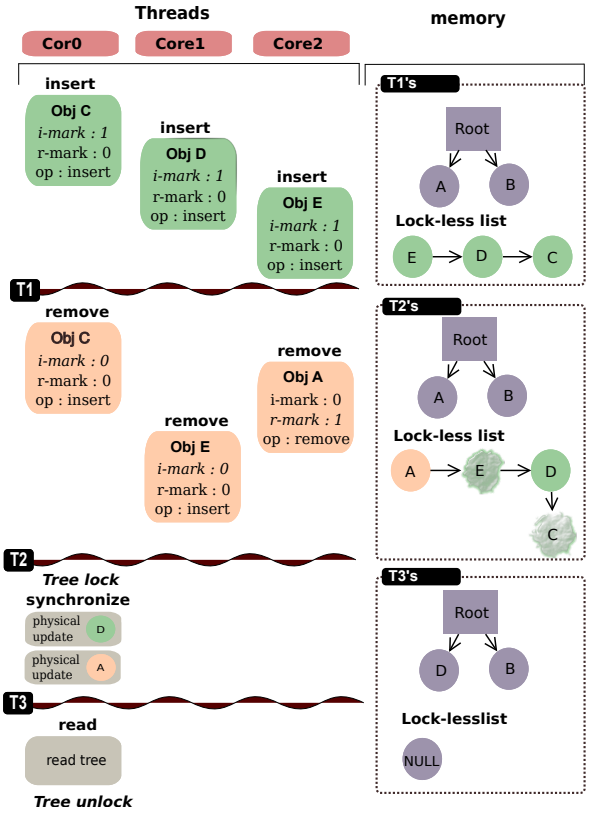

Figure 1: LDU example showing six update operations and one read operation. The execution flows from top to bottom. Memory represents original data structure and logging queue at T1, T2 and T3, respectively.

currently. At `T1`, the tree contains node `A` and `B` and the lock-less list contains node `E`, `D` and `C`. When removing the node `C`, the node `C`, whose mark field was marked by insert, atomically cleans up the insert marked field. At `T2`, the lock-less list contains nodes `A`, `E`, `D`, and `C`, and the marking field is zero for nodes `E` and `C`. Before running the `synchronize` function, they need to lock the original tree's lock using the exclusive lock in order to protect the tree's operation. The `synchronize` migrates from lock-less list node to tree node, each of which is the marked node, so nodes `A` and `D` are migrated. Finally, the tree contains nodes `D` and `B`, so the reader can read eventually consistent data.

3.3.2 PLDU example

Figure 2 gives an example of deferred update with six update operations and one read operation. In this figure, execution flows from top to bottom. The data structure for *physical update* is a tree, and initial values in the tree

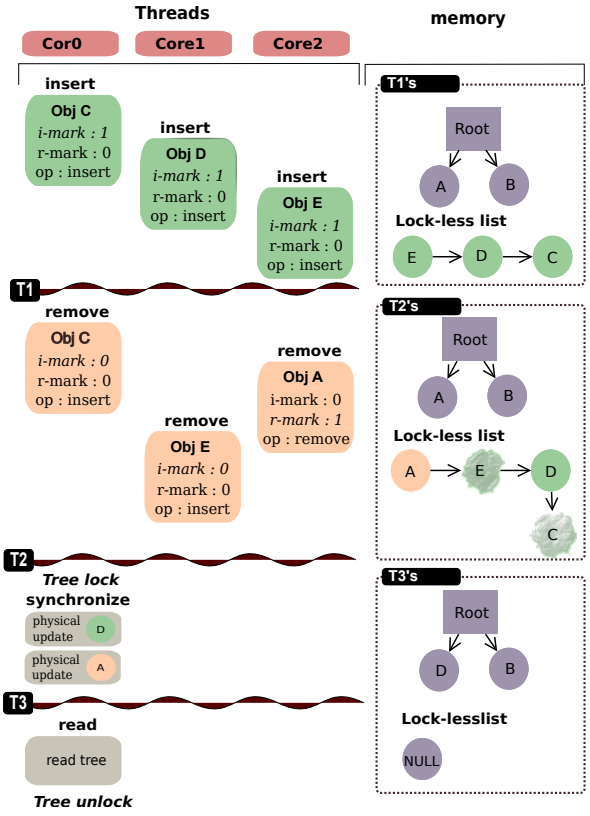

Figure 2: LDU example showing six update operations and one read operation. The execution flows from top to bottom. Memory represents original data structure and logging queue at T1, T2 and T3, respectively.

are node A and B. In contrast, the data structure for *logical update* is lock-less list. In the top figure, Core0, Core1 and Core2 perform the logical insert operation to nodes C, D and E, respectively. The logical inserts set the insert mark, and they then insert their nodes into lock-less list. In this case, none of the lock is needed because LDU uses the lock-less list;all threads can execute the update concurrently. At T1, the tree contains node A and B and the lock-less list contains node E, D and C. When removing the node C, the node C, whose mark field was marked by insert, atomically cleans up the insert marked field. At T2, the lock-less list contains nodes A, E, D, and C, and the marking field is zero for nodes E and C. Before running the `synchronize` function, they need to lock the original tree's lock using the exclusive lock in order to protect the tree's operation. The `synchronize` migrates from lock-less list node to tree node, each of which is the marked node, so nodes A and D are migrated. Finally, the tree contains nodes D and B, so the reader can read eventually consistent data.

3.4 GLDU vs. PLDU

3.5 The LDU Algorithm

3.5.1 GLDU logical update

The pseudo code for LDU's *logical update* is given in The `logical_insert`, the concurrent update function, checks whether this object already has been removed by `logical_remove`. If this object has been removed, `logical_insert` initializes the marking field and then they return, which is fastpath. The marking field needs synchronization because this field in the *logical update* is shared with the *physical update*, so the CAS operation is needed. When the marking field has been initialized, they set the marking field, then they check whether or not this node already has been inserted in lock-less list. If the node does not exist in lock-less list, then they insert the node into lock-less list.

3.5.2 GLDU Physical update

The pseudo code for LDU's *physical update* is given in Figure **??**. First, they check whether lock-less list is an empty list or not, then they iterate the lock-less list. If the marking field has been set, they execute migration from lock-less to original data structure. Because the marking field in *physical update* is shared with *logical update*, the CAS operation is needed. They initialize the used field, which needs to protect the object from freed through destructor. The programmer must acquire locks on the `synchronize_ldu` function, which migrates log to original data structure. Finally, the `physical_update` executes original functions by using the operation log.

3.5.3 The PLDU Algorithm

3.5.4 PLDU logical update

* By default OpLog executes logged updates in temporal order. For example, consider a linked list. If a process logs an insert operation on one core, migrates to another core, and logs a remove operation, the remove should eventually execute after the insert. * OpLog relies on timestamps from a system-wide synchronized clock to tell it how to order entries in different cores' log. *This ordering ensures linearizaility, making OpLog compatible with existing data structure semantics.

```
bool gldu_logical_insert(struct vm_area_struct *vma,
  struct address_space *mapping)
{
  struct ldu_node *add = &vma->ldu.node[0];
  struct ldu_node *del = &vma->ldu.node[1];

  if(!xchg(&del->mark, 0)){
    add->mark = 1;
    if(!test_and_set_bit(LDU_ADD, &vma->ldu.used)){
      add->op_num = LDU_OP_ADD;
      add->key = vma;
      add->root = &mapping->i_mmap;
      ldu_logical_update(mapping, add);
    }
  }

  return true;
}

bool gldu_logical_remove(struct vm_area_struct *vma,
    struct address_space *mapping)
{
  struct ldu_node *add_dnode = &vma->dnode.node[0];
  struct ldu_node *del_dnode = &vma->dnode.node[1];

  if(atomic_cmpxchg(&add_dnode->mark, 1, 0) != 1) {
    atomic_set(&del_dnode->mark, 1);
    if(!test_and_set_bit(LDU_DEL, &vma->ldu.used)){
      del_dnode->op_num = LDU_OP_DEL;
      del_dnode->key = vma;
      del_dnode->root = &mapping->i_mmap;
      i_mmap_ldu_logical_update(mapping, del);
    }
  }

  return true;
}
```

Figure 3: LDU logical update algorithm. `logical_insert` represents non-blocking insert function. It may be called by original insert position without locks. The fastpath is that when their object was removed by `logical_remove`, `logical_insert` just changes node's marking field.

4 Concurrent updates for Linux kernel

4.1 Case study:reverse mapping

4.2 anon vma

* LDU.
  * PLDU.

4.3 file mapping

* LDU.
  * PLDU.

```
void synchronize_gldu(struct address_space *mapping)
{
  struct llist_node *entry;
  struct ldu_node *dnode, *next;
  struct ldu_head *lduh = &mapping->lduh;

  entry = llist_del_all(&lduh->ll_head);
  llist_for_each_entry_safe(dnode,
      next, entry, ll_node) {
    struct vm_area_struct *vma =
        ACCESS_ONCE(dnode->key);
    if (atomic_cmpxchg(&dnode->mark,
        1, 0) == 1) {
      ldu_physical_update(dnode->op_num,
          vma,
          ACCESS_ONCE(dnode->root));
    }
    clear_bit(dnode->op_num, &vma->dnode.used);
    if (atomic_cmpxchg(&dnode->mark, 1, 0) == 1) {
      ldu_physical_update(dnode->op_num, vma,
          ACCESS_ONCE(dnode->root));
    }
  }
}
```

Figure 4: LDU physical update algorithm. `synchronize_ldu` may be called by reader and converts update log to original data structure traversing the lock-less list.

5 Implementation

6 Evaluation

This section answers the following questions experimentally:

- Does LDU's design matter for applications?
- Why does LDU's scheme scale well?

6.1 Experimental setup

To evaluate the performance of LDU, we use well-known three benchmarks:AIM7 Linux scalability benchmark, Exim email server in MOSBENCH and lmbench. We selected these three benchmarks because they are fork-intensive workloads and exhibit high reverse mapping lock contentions. Moreover, AIM7 benchmark has widely been used in practical area not only for testing the Linux but also for improving the scalability. To evaluate LDU for real world applications, we use Exim which is the most popular email server. A micro benchmark, Lmbench, has been selected to focus on Linux fork operation-intensive fine grained evaluations. Finally, we wanted to focus on Linux fork performance and scalability;therefore, we selected lmbench, a micro benchmark.

```
bool pldu_logical_update(struct address_space *mapping,
    struct ldu_node *dnode)
{
  struct pldu_deferred_i_mmap *p;
  struct i_mmap_slot *slot;
  struct llist_node *first;
  struct llist_node *entry;
  struct ldu_node *ldu;

  slot = &get_cpu_var(i_mmap_slot);
  p = &slot->mapping[hash_ptr(mapping, HASH_ORDER)];
  first = READ_ONCE(p->list.first);
  if (first) {
    ldu = llist_entry(first, struct ldu, ll_node);
    if (ldu->root != dnode->root) {
      //pr_info("conflict hash table\n");
      locked_mapping = READ_ONCE(ldu->key2);
      entry = llist_del_all(&p->list);
      llist_add(&dnode->ll_node, &p->list);
      put_cpu_var(i_mmap_slot);
      down_write(&locked_mapping->i_mmap_rwsem);
      synchronize_ldu_i_mmap_internal(entry);
      up_write(&locked_mapping->i_mmap_rwsem);
      goto out;
    }
  }

  llist_add(&dnode->ll_node, &p->list);
  put_cpu_var(i_mmap_slot);

out:
  return true;
}

bool pldu_logical_insert(
    struct vm_area_struct *vma,
    struct address_space *mapping)
{
  struct ldu_node *add = &vma->dnode.node[0];
  struct ldu_node *del = &vma->dnode.node[1];

  if (atomic_cmpxchg(&del->mark, 1, 0) != 1) {
    BUG_ON(atomic_read(&add->mark));
    atomic_set(&add_dnode->mark, 1);
    if (!test_and_set_bit(OP_ADD, &vma->used)) {
      add->op_num = OP_ADD;
      add->key = vma;
      add->key2 = mapping;
      add->root = &mapping->i_mmap;
      ldu_logical_update(mapping, add_dnode);
    }
  }

  return true;
}
```

Figure 5: LDU logical update algorithm. `logical_insert` represents non-blocking insert function. It may be called by original insert position without locks. The fastpath is that when their object was removed by `logical_remove`, `logical_insert` just changes node's marking field.

```
void synchronize_ldu(struct llist_node *entry)
{
  struct ldu_node *dnode;
  struct address_space *mapping;
  struct vm_area_struct *vma;

  llist_for_each_entry(dnode,
      entry, ll_node) {
    vma = READ_ONCE(dnode->key);
    if (atomic_cmpxchg(
        &dnode->mark, 1, 0) == 1) {
      ldu_physical_update(
          dnode->op_num, vma,
          READ_ONCE(dnode->root));

    }
    clear_bit(dnode->op_num,
        &vma->dnode.used);
    if (atomic_cmpxchg(
        &dnode->mark, 1, 0) == 1) {
      ldu_physical_update(
          dnode->op_num, vma,
          READ_ONCE(dnode->root));
    }
  }
}
```

Figure 6: LDU physical update algorithm. `synchronize_ldu` may be called by reader and converts update log to original data structure traversing the lock-less list.

In order to evaluate Linux scalability, we used four different experiment settings. First, we used the stock Linux as the baseline reference. Second, we used ordered Harris lock-free list while we apply unordered Harris lock-free list for the third setting (see section 5). Finally, we used combination of unordered Harris lock-free list for anonymous mapping and our LDU for file mapping. Since we cannot obtain detailed implementation of Oplog, we could not include comparison between LDU and Oplog in this paper.

We compare our LDU implementation to a concurrent non-blocking Harris linked list [**?**];therefore, we implement the Harris The code refers from sysnchrobench [**?**] and ASCYLIB [**?**], and we convert their linked list to Linux kernel style. Because both synchrobench and ASCYLIB leak memory, we implement additional garbage collector for the Linux kernel using Linux's work queues and lock-less list.

In order to further improve performance, we move their ordered list to unordered list. A feature of the Harris linked list is all the nodes are ordered by their key. Zhang [**?**] implements a lock-free unordered list algorithm, whose list is each insert and remove operation appends an intermediate node at the head of the list;these approach is practically hard to implement. Indeed, Linux

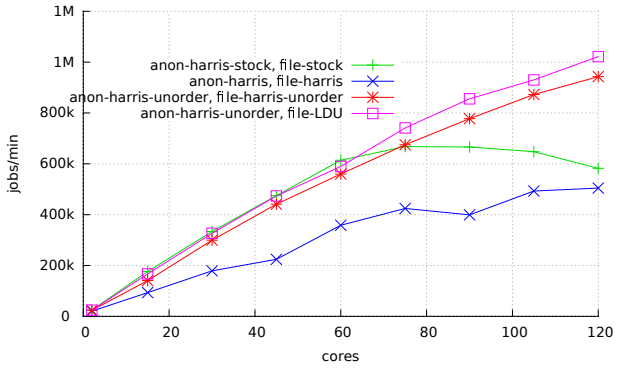

Figure 7: Scalability of AIM7 multiuser for different method. The combination LDU with unordered harris list scale well;in contrast, up to 60 core, the stock Linux scale linearly, then it flattens out.

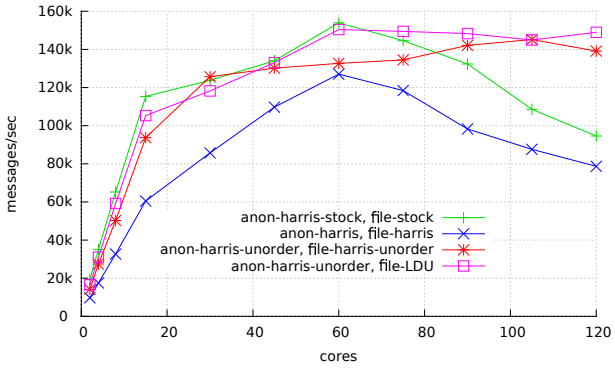

Figure 8: Scalability of Exim. The stock Linux collapses after 60 core;in contrast, both unordered harris list and our LDU flatten out.

does not require contains operation because the Linux data structures such as list, tree and hash table not depended on search key;they depend on their unique object. This feature can eliminates the ordered list in Harris linked list. Therefore, we perform each insert operation appends an intermediate node at the first node of the list;on the other hand, each remove operation searches from head to their node.

We ran the three benchmarks on Linux 3.19.rc4 with stock Linux with the automatic NUMA balancing feature disabled because the Harris linked list has the iteration issue [**?**]. All experiments were performed on a 120 core machine with 8-socket, 15-core Intel E7-8870 chips equipped with 792 GB DDR3 DRAM.

6.2 AIM7

AIM7 forks many processes, each of which concurrently runs. We used AIM7-multiuser, which is one of workload in AIM7. The multiuser workload is composed of various workloads such as disk-file operations, process creation, virtual memory operations, pipe I/O, and arithmetic operation. To minimize IO bottlenecks, the workload was executed with tmpfs filesystems, each of which is 10 GB. To increase the number of users during our experiment and show the results at the peak user numbers, we used the crossover.

The results for AIM7-multiuser are shown in Figure 7, and the results show the throughput of AIM7-multiuser with four different settings. Up to 60 core, the stock Linux scales linearly while serialized updates in Linux kernel become bottlenecks. However, up to 120core, unordered harris list and our LDU scale well because these workloads can run concurrently updates and can reduce the locking overheads due to reader-writer

AIM7	user	sys	idle
Stock(anon, file)	2487 s	1993 s	4647 s(51%)
H(anon, file)	1123 s	3631 s	2186 s(31%)
H-unorder(anon, flie)	3630 s	2511 s	1466 s(19%)
H-unorder(anon), L(file)	3630 s	1903 s	1662 s(23%)
EXIM	**user**	**sys**	**idle**
Stock(anon, file)	41 s	499 s	1260 s(70%)
H(anon, file)	47 s	628 s	1124 s(62%)
H-unorder(anon, file)	112 s	1128 s	559 s(31%)
H-unorder(anon), L(file)	87 s	1055 s	657 s(37%)
lmbench	**user**	**sys**	**idle**
Stock(anon, file)	11 s	208 s	2158 s(91%)
H(anon, file)	11 s	312 s	367 s(53%)
H-unorder(anon, file)	11 s	292 s	315 s(51%)
H-unorder(anon), L(file)	12 s	347 s	349 s(49%)

Table 1: Comparison of user, system and idle time at 120 cores.

semaphores(`anon_vma`, `file`). The combination of LDU with unordered harris list has best performance and scalability outperforming stock Linux by 1.7x and unordered harris list by 1.1x. While the unordered harris list has 19% idle time(see Table 1), stock Linux has 51% idle time waiting to acquire both `anon_vma's rwsem` and `file's i_mmap_rwsem`. We can notice that although LDU has 23% idle time, the throughput is higher than unordered harris list. In this benchmark, the ordered harris list has the lowest performance and scalability because their `CAS` fails frequently.

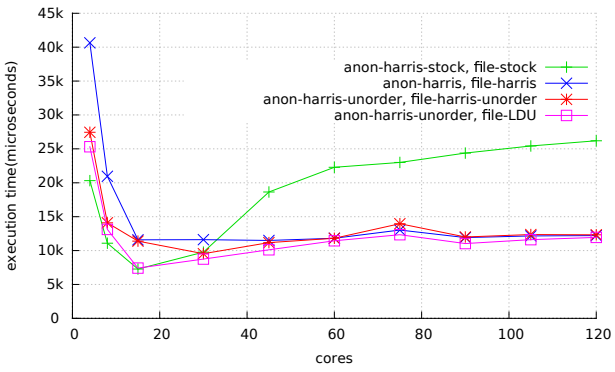

Figure 9: Execution time of lmbench's fork micro benchmark. The fork micro benchmark drops down for all methods up to 15 core but either flattens out or goes up slightly after that. At 15 core, the stock Linux goes up;the others flattens out

6.3 Exim

To measure the performance of Exim, shown in Figure 8, we used default value of MOSBENCH to use tmpfs for spool files, log files, and user mail files. Clients run on the same machine and each client sends to a different user to prevent contention on user mail file. The Exim was bottlenecked by per-directory locks protecting file creation in the spool directories and by forks performed on different cores [**?**]. Therefore, although we eliminate the fork problem, the Exim may suffer from contention on spool directories.

Results shown in Figure 8 show that Exim scales well for all methods up to 60 core but not for higher core counts. The stock Linux shows performance degradation for more than 60 core. Both unordered harris list and our LDU do not suffer from performance loss because they do not acquire the `anon_vma` semaphore and `i_mmap` semaphore in fork. LDU performs better due to the fact that it uses both update-side absorbing and lock-less list, outperforming stock Linux by 1.6x and unordered harris list by 1.1x. Even though we applied scalable solution, Exim shows limitation on scalability improvement since the main bottleneck is per-directory lock contention on spool directories. The unordered harris list has 31% idle time, whereas LDU has 37% idle time due to the their efficient concurrent updates.

6.4 lmbench

lmbench has various workloads including process creation workload(fork, exec, sh -c, exit). This workload is used to measure the basic process primitives such as creating a new process, running a different program, and context switching. We configured process create workload to enable the parallelism option which specifies the number of benchmark processes to run in parallel [**?**]; we used 100 processes.

The results for lmbench are shown in Figure 9, and the results show the execution times of the fork microbenchmark in lmbench with four different methods. Three methods outperform stock Linux by 2.2x at 120 cores;however, before 30 core, two harris list have lower performance due to their execution overheads. While stock Linux has 90% idle time, other methods have approximately 50% idle time since stock Linux waits to acquire reverse mapping locks such as `anon_vma's rwsem` and `mapping's i_mmap_rwsem`.

7 Discussion and future work

8 Related work

In order to improve Linux scalability, researchers have been optimized memory management in Linux by finding and fixing scalability bottlenecks.

Shared address spaces in multithreaded applications easily become scalability bottlenecks since kernel operations including `mmap` and `munmap` system calls and `page faults` handling require per-process locks for synchronization. Multithreaded application, for example, can become bottleneck by kernel operations on their shared address space, whose operations are the `mmap` and `munmap` system calls and `page faults`. These operations are synchronized by a single per-process lock. BonsaiVM [**?**] solved this address space problem by using the RCU; RadixVM [**?**] created a new VM using refcache and radix tree, which enable `munmap`, `mmap`, and `page fault` on non-overlapping memory regions to scale perfectly. Alternatively, to avoid contention caused by shared address space locking, system programmers change their multithreaded applications to use processes [**?**].

Though sufficient level of performance scalability has been achieved for reader intensive operations through RCU and Hazard pointer, solutions to scalability for update-heavy operations has not been satisfiable. A recent paper by Arbel and Attiya [**?**] shows a new design of concurrent search tree called the Citrus tree. The Citrus tree combines RCU and fine-grained locks, and it supports concurrent write operations that traverse the search tree by using RCU concurrently. When increasing the update rate, Citrus tree still suffers from bottlenecks. RLU [**?**] presents a new synchronization mechanism that allows unsynchronized sequences of reads to execute concurrently with updates. In high update rate, Oplog can achieve substantially multi-core scaling for

update-heavy data structures. Our work focus on update-heavy data structures and uses non-blocking method to store the operation log instead of per-core processing.

One method for the concurrent update is using the non-blocking algorithms [**?**] [**?**] [**?**], which are based on CAS. In non-blocking algorithms, each core tries to read the values of shared data structures from its local location, but has possibility of reading obsolete values. CAS is performed at the time of reading values that are not the current values and CAS fails and requires retrials sometimes when the values have been overwritten. These algorithms execute optimistically as though they read the value at location in their data structure;they may obtain stale data at the time. When they observed against the current value, they execute a CAS to compare the against value. The CAS fails when the value has been overridden, and they must be retried later on. Consequently, both repeated CAS operation and their iteration loop caused by CAS fails cause bottlenecks due to inter-core communication overheads [**?**]. Moreover, none of the non-blocking algorithms implements an iterator, whose data structure just consists of the insert, delete and contains operations [**?**]. The Linux, however, commonly uses the iteration to read, so when applying non-blocking algorithms to the Linux, they may meet this iteration problem. Petrank [**?**] solved this problem by using a consistent snapshot of the data structure; this method, however, may require a lot of effort to apply its sophisticated algorithms to Linux. For evaluation purposes, we implemented Harris linked list [**?**] to Linux, and we sometimes have failure where reading the pointer that had been deleted by updater concurrently result of the problem of the iteration.

MCS [**?**], a scalable exclusive lock, is used in the Linux kernel [**?**]. to avoid unfairness at high contention levels, so this scalable exclusive lock can be used for fine grained locking in Linux. However, in MCS, since only one thread may hold the lock at a time, it can cause low scalability in case of long critical regions. Reader-writer lock [**?**] allows either number of readers to execute concurrently or single writer to execute. Thus, readers-writer locks allow better scalability in case of read-mostly objects.

In read-mostly data structures, RCU [**?**] can be quite useful since it allows read operations to proceed without read locks, and delays freeing of data structures to avoid races. One drawback is that as the update rate increases, their performance and scalability decrease due to a single writer and their synchronization function. Consequently, scalable exclusive lock, reader-writer lock and RCU require serialization for updates and thus show significant limitation on scalability.

9 Conclusion

We propose a concurrent update algorithm, LDU, for update-heavy data structures scalable for many-core systems. To achieve the scalability during process spawning, we applied deferred log processing with global log queue and update-side absorbing to Linux reverse mapping. Evaluation results using the AIM7, Exim and lmbench reveal that LDU shows better performance up to 2.2 times compared to existing solutions. LDU is implemented on to Linux kernel 3.19 and available as open-source from `https://github.com/KMU-embedded/scalablelinux`.

10 Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (14-824-09-011, “Research Project on High Performance and Scalable Manycore Operating System”)