A Lightweight Log-based Deferred Update for Linux Kernel Scalability

***Abstract*—We propose a novel light weight concurrent updates method, LDU, to improve performance scalability for Linux kernel on many-core systems through eliminating lock contentions for update-heavy global data structures during process spawning and optimizing update logs. The proposed LDU is implemented into Linux kernel 4.5 and evaluated using representative benchmark programs. Our evaluation reveals that the Linux kernel with LDU shows performance improvement by ranging from Xx through Xx on a 120 core system.**

***Keywords*-component; formatting; style; styling;**

I. Introduction

최근 코어수가 증가하고 있다. 따라서 멀티코어에서 매니코어 시스템으로 바뀌고 있다. 매니코어 시스템에 대한 운영체제 커널의 parallelism은 시스템 전체의 parallelism에서 가장 중요하다. 만약 커널이 scale하지 않으면, 그 위에 동작하는 응용프로그램들도 역시 scale하지 않는다[]. 이처럼 중요한 운영체제 커널 중 멀티코어 또는 매니코어 환경에서 많이 사용되는 운영체제가 리눅스 커널이다[]. 하지만 리눅스 커널은 아직 확장성 문제가 있다 [1] [2]. 확장성 문제 중 하나는 락 경합 때문에 발생하는 업데이트 직렬화 문제이다 [3] [4]. 그 이유는 업데이트 오퍼레이션은 여러 쓰레드가 동시에 수행되지 못하기 때문이다 [5].

이처럼 업데이트 직렬화 문제를 해결하기 위해 여러 동시적 업데이트 방법 들이 연구되고 있다 [6] [3]. 이러한 동시적 업데이트 방법들은 워크로드 특성인 업데이트 비율에 따라 많은 성능 차이를 보인다 [3]. 이 중 높은 업데이트 비율을 가진 자료 구조 때문에 발생하는 학장성 문제를 해결하기 위한 여러 방법이 연구 되고 있다. 그 중 하나는 cache communication bottleneck을 줄인 log-based 알고리즘 [7] [8] [9]을 사용하는 것이다. Log-based 알고리즘은 업데이트가 발생하면, data structure의 업데이트 operation을 per-core 또는 atomic하게 log로 저장하고 read operation을 수행하기 전에 저장된 로그를 수행하는것이다 . 이것은 마치 CoW(Copy On Write)와 유사하다 [10].

S. Boyd-Wickizer et al.는 동기화된 타임스탬프 카운터(synchronized timestamp counters) 기반의 per-core log를 활용하여 update-heavy한 자료구조를 대상으로 동시적 업데이트 문제를 해결함과 동시에 cache communication bottleneck을 줄였다 [9]. 동기화된 타임스탬프 카운터 기반의 per-core log를 활용한 동시적 업데이트방법은 업데이트 부분만 고려했을 때, per-core에 데이터를 저장함으로 굉장히 높은 scalability를 가진다[]. 하지만 per-core 기반의 동기화된 타임스탬프 카운터를 사용한 방법은 결국 timestamp merging and ordering 작업을 야기한다. 만약 코어 수가 늘어 날 경우, 로그를 자료 구조에 적용하는 과정에서 timestamp 때문에 발생하는 추가적인 sequntial 프로세싱이 요구된다. 이것은 결국 확장성과 성능을 저해한다.

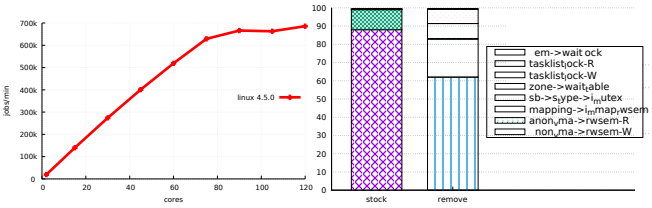
본 논문은 동기화된 타임스탬프 카운터를 이용함에 따라 생기는 추가적인 sequencial processing 문제를 해결하기 위해 shared memory system을 위한 새로운 LDU(Lightweitgh log-based Deferred Update)를 개발하였다. LDU는 타임스탬프 카운터가 필요한 operation log를 업데이트 순간 지우고, 매번 로그를 생성하지 않고 재활용하는 방법이다. 이로인해 synchronized timestamp counter 문제와 cache communication bottleneck 문제를 동시에 해결하였다. 해결 방법은 분산 시스템에서 사용하는 synchronized timestamp log기반의 concurrent updates 방식[]과 최소한의 shared-memory system의 hardware-based synchronization 기법(compare and swap, test and set, atomic swap)을 조합하여 동시적 업데이트 문제를 해결하였다.

이처럼 동기화된 타임스탬프 카운터를 제거함과 동시에, cache communication bottleneck 줄인 LDU는 기존 log-based 알고리즘들의 장점들을 모두 포함할 뿐만아니라 추가적인 장점을 가진다. 첫째로, update가 수행하는 시점 즉 로그를 저장하는 순간에는 lock이 필요가 없다. 따라서 lock에 대한 오버헤드 없이 concurrent updates를 수행할 수 있다 둘째로, 저장된 update operation log를 coarse-grained lock과 함께 하나의 코어에서 수행하기 때문에, cache 효율성이 높아진다 [8]. 셋째로, 기존 여러 자료구조에 쉽게 적용할 수 있는 장점이 있다. 게다가 마지막으로, log를 저장하기 전에 로그를 삭제하므로 보다 빠르게 log의 수를 줄일 수 있다.

우리는 위와 같은 장점을 가지는 LDU를 리눅스 커널에서 high update rate 때문에 scalability 문제를 야기시키는 anonymous reverse mapping과 file reverse mapping에 적용하였다. 또한 우리는 LDU를 Linux 4.5.rc4에 구현하였고, fork-intensive 워크로드인 AIM7 [11], Exim [12] from MOSBENCH [13], lmbench [14]를 대상으로 성능 개선을 보였다. 개선은 stock 리눅스 커널에 비해 120코어에서 각각 x,x,x 배이다.

Contributions. This paper makes the following contributions:

- 우리는 high update rate를 가지는 data structuer를 위한 새로운 log-based concurrnet updates 방법인 LDU를 개발하였다. LDU는 동기화된 타임스탬프 카운터를 이용함에 따라 생기는 시간 정렬과 머징에 의한 추가적인 sequencial processing 문제를 최소한의 hardware-based synchronization 기법을 사용하여 해결하였다. LDU는 hardware-based synchronization



(a) aa (b) aa

Figure 1: Scalability of AIM7 multiuser. This workload simultaneously create many processes. Up to 60 core, the stock Linux scale linearly, then they flattens out.

기법을 이용하여 LDU는 로그를 업데이트 순간 지우고 로그를 재활용한다.

- 우리는 LDU을 practical한 manycore system인 intel xeon 120코어 위에 동작하는 리눅스 커널의 2가지 reverse mapping(anonymous, file)에 적용하여, fork scalability 문제를 해결하였다. Fork 관련 벤치마크 성능은 워크로드 특성에 따라 1.6x부터 2.2x까지 개선되었다.

The rest of this paper is organized as follows. Section 2 describes the background and Linux scalability problem. 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 8 concludes the paper.

II. Background and Problem

운영체제 커널의 paralleism은 시스템 전체의 parallesim에서 가장 중요하다. 만약에 커널이 scale하지 않으면, 그 위에 동작하는 응용프로그램들도 역시 scale하지 않는다 [15]. 우리는 이처럼 중요한 부분인 운영체제 커널중 multi-core에 최적화된 리눅스의 scalabiliy를 분석하기 위해, AIM7 multiuser[]를 가지고 scalability를 실험해보았다. AIM7은 최근에도 scalability를 위해 reserach 진영과 리눅스 커널 진영에서도 활발히 사용되고 있는 벤치마크 중 하나이다 [16] [17]. File system scalability를 최소화 하기 위해 temp filesystem [18]을 사용하였다. This workload simultaneously create many processes. Up to 60 core, the stock Linux scale linearly, then they flattens out.

우리는 Scalability에 문제가 있는 120코어에서 리눅스의 Lockstat[]를 이용하여 락 경합을 분석하였다. 먼저 멀티 프로세스 기반의 벤치마크인 AIM7을 동작시키고 동시에 120코어 대해서 락 경합을 분석하면 그림 3과 같은 결과를 가진다. AIM7 벤치마크의 경우 상당히 많은 부분이 anonvma에서 쓰기 락 경합이 발생한다. 이는 리눅스 역 매핑(reverse mapping)을 효율적으로 수행하기 위한 자료구인 anonvma가 수많은 fork에 의해 프로세스를 생성하면서 발생하는 락 경합 문제이다. 다음으로 우리는 anonvma의 lock 경합을 줄이기 위해, 임시로 fork에서 anonvma를 호출하는 부분과 read와 관련있는 pageswap이 안되도록 하고, 120코어를 대상으로 다시 Lock 경합을 분석하여 보았다. 이 때 부터 그동안 상대적으로 가려졌던 file reverse mapping에서 많은 락 경합이 발생되었다. 이러한 anonvma reverse page mapping은 리눅스 커뮤니티에서 잘 알려진 락 경합 문제 [19] [20]이고, file mapping에 대한 락 경합 문제는 Silas wikizer가 OpLog 논문을 통해 fork의 scalability 문제의 원인으로 제시한 부분이다. 결론적으로 본 연구의 분석 결과 둘 중 하나가 아니라 두 가지 락 모두 fork의 scalabililty 문제를 야기 시킨다. 즉 두가지 모두 개선해야지 fork의 scalalbility가 향상 된다.

두가지 revserse page mapping의 근본적인 문제는 hight update operation에 대한 serialization 때문에 발생하는 문제이다. 이러한 revser page mapping은 page frame reclaiming을 위해 존재하며, 리눅스가 fork(), exit(), and mmap() 시스템콜을 사용할 때 rmap을 update한다. 리눅스 커널은 reader들은 parallal 하게 동작할 수 있는 RW-lock 또는 RCU 같은 락 메카니즘이 있으나, 이러한 락은 결국 high update rate 앞에서는 serialization된다. Update는 execulucive lock을 통해 보호해야하기 때문에, update rate 높은 상황이 발생하면 리눅스 커널은 결국 lock에 의해 serialized되어 scalability가 떨어진다.

이러한 high update rate이 발생하는 상황의 update serialization 문제에 대한 해결 방법들은 존재한다. 해결 방법은 concurrent updates를 위한, non-blocking data structure와 log-based 알고리즘을 사용하는 방법이 있다. Non-blocking algorithms들은 hardware synchronized atomic 연산들을 활용하여 current 하게 update와 read를 수행하게 만든 data structure이다. 하지만 shared memory global value를 multipul CAS로 접근하여 bottlenecks이 생긴다. due to inter-core communication overheads [9]. 최근에는 Deu to the multipul CAS, inter-core communication overheads를 줄인 log-based 방법들이 연구되고 있다. 우리의 LDU도 이러한 log-based 기반 방법 활용하였으며, log-based 방법에 대한 설명은 다음 장에서 자세히 다룬다.

III. LDU Design

LDU는 리눅스 커널의 high update rates를 가진 data strcuture의 scalability를 해결하기 위한 log-based 방법중에 하나이다. 동기화된 타임스탬프 카운터 기반의 per-core log를 활용한 동시적 업데이트방법은 결국 timestamp ordering and merging 작업을 야기한다. 특히 코어 수가 늘어 날 경우, per-core 로그를 자료 구조에 적용하는 과정에서 추가적인 sequntial 프로세싱이 요구된다. 이것은 확장성과 성능을 저해한다. 이러한 문제를 해결하기 위해, LDU는 log기반 방식의 concurrent updates 방법과 atomic synchronization 기능을 최소한으로 사용하도록 설계하였다. 따라서, LDU는 synchronized timestamp counter를 사용하는 방식의 timestamp를 제거함과 동시에 cache communication overhead를 최소화 하였다.

A. *Log-based Concurrent updates*

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

Update heavy한 구조를 위한 Log-based 방법은 총 4가지의 장점을 가진다. 첫째로, update가 수행하는 시점 즉 로그를 저장하는 순간에는 lock이 필요가 없다. 따라서 update를 concurrent하게 수행할 수 있을 뿐아니라, lock 자체가 가지고 있는 overall coherence traffic is significantly reduced. 둘째로, 저장된 sequantial update operation log를 corse-grain lock과 함께 하나의 코어에서 수행하기 때문에, cache 효율성이 높아진다. 셋째로, 큰 수정 없이 기존 여러 데이터(tree, queue) structure에 쉽게 적용할 수 있는 장점이 있다. 마지막으로 저장된 log를 실제 수행하지 않고, 여러가지 optimization 방법을 사용하여 적은 operation으로 Log를 줄일 수 있다. LDU도 log-based approach를 따른다. 그러므로 앞에서 설명한 log-based 방법의 장점을 모두 가짐과 동시에 업데이트 순간 삭제 가능한 log를 지움으로 성능을 향상시킨다.

B. *Approach*

Synchronized timestamp가 근본적으로 필요한 이유는 when 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[]. LDU는 update 순간 time-sensitive log를 제거함으로 synchronized timestamp counter 때문에 발생하는 문제를 해결하였다. time-sensitive log란 순서가 바뀌서는 안되는 operation log이다. 본 논문에서는 이러한 time-sensitive log를 설명하기 위해, [15]에서 사용한 심볼방법을 이용하였다. 먼저 ⊕와 ⊖는 각각 insert와 remove update operation을 의미한다. 바로 따라오는 심볼은 object의 명을 의미하고 색깔과 높낮이는 서로 다른 CPU를 의미한다.

$$\oplus_A,\ \oplus_B,\ \oplus_C, \ominus_A,\ \ominus_C,\ \oplus_A,\ \oplus_C, \ominus_C$$

즉 위와 같이 수행되었을 경우 time-sensitive log인 $\oplus_A$과 $\ominus_A$에 대해서는 항상 timestamp에 맞게 수행되어야 한다. 여기서 중요한 사실은 time-sensitive operation log은 삭제해도 괜찮은 operation log들 이다. insert-remove operation 또는 remove-insert operation을 가지는 $\ominus_B$, $\oplus_C$ 삭제되도 괜찮으며, 결국 남은 operation log은 같은 object에 대해서 insert 또는 remove operation만 남게 된다. 위의 log는 아래와 같이 non-time-sensitive한 operation만 남게 된다.

$$\oplus_B,\ \oplus_A$$

LDU는 update-side removing이라는 방법으로 이러한 time-senstivive한 log를 update 순간 바로 지운다.

LDU는 time-senstivive한 log를 제거하기 위해 update-side removing이라는 방법을 사용한다. 이 방법은 만약 같은 object에 대해서 insert와 remove가 발생하였으면, 같은 object에 대해서, insert oepration과 remove operation에 대한 log를 update시점에 바로 바로 삭제하는 방법이다. 이처럼 log가 삭제될 수 있는 이유는 operation log가 deferred로 수행하기 문에 가능하다. syncronized timestamp counters 기반의 OpLog도 이러한 log 삭제 방법 수행하여 최적화를 하였으나, operation log가 서로 다른 코어에 존재하는 log 같은 경우에 log를 merge와 검색한 후 삭제를 해야한다. syncronized timestamp counters 방법은 워크로드에 따라, 최적화를 위해 또 다른 sequntial 프로세싱이 요구된다. 하지만 LDU는 indivisual object를 대상으로 swap atomic operation을 사용하여 shared log 삭제하는 방법을 사용해서 이런 문제가 없다.

업데이트 순간 로그를 지우는 방법은 shared memory system의 swap operation을 사용한다. 이를 위해, LDU는 모든 object에 insert와 remove의 mark 필드를 추가해서 update-side absorbing을 수행 하였다. 예를 들어 만약 A라는 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 atomic하게 지워주는 방식으로 진행된다. 다음으로 LDU는 log를 적용할 때, queue안에 log가 존재하더라도, mark 필드가 표시된 log만 실행한다. 이것은 swap이라는 상대적으로 가벼운 연산과 상대적으로 덜 share 하는 indivisaul global object의 mark filed를 사용해서 time-sensitive한 operation을 제거할 뿐만 아니라, 동시에 실제 operation을 수행하지 않고 log를 지워주는 효과를 가져주므로 성능이 향상된다.

이처럼 update-side absorbing으로 log를 지울 수 있는 근본 이유는 리눅스 커널(freeBSD 커널 역시) search가 two update operations(one to insert and one to remove an element)이 함수 외부해서 수행하는 구조이기 때문에 가능하다. 즉 리눅스의 data structure의 경우에는 search와 update가 분리되어 있어, 같은 update operation이 연달아 호출되지 않는다. LDU는 이러한 사실을 이용하여 update-side absorbing log를 개발 하였다. 즉 리눅스의 data structure 구조는 search가 update operaton 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) [21]의 방법들은 search가 update operation 내부에 존재하는 구조로 되어 있다. 따라서 같은 key값의 노드에 대해서 insert-insert 또는 remove-remove 순서로 동작할 수 있다. 이처럼 CSDS 알고리즘일 경우에는 LDU를 적용하지 못하는 limitation이 있다. 하지만 본 연구는 리눅스 커널과 같은 practical한 data structure에 초점을 맞추었기 때문에, research 분야에 많이 연구되고 있는 CSDS 알고리즘은

고려하지 않았다. 아직 CSDS의 non-blocking 알고리즘들은 garbage collector [22], iteration [23], ordering [24]여러 practical한 이유로 C언어로 구현된 리눅스 커널 등에 적용하기에는 아직 한계가 있다.

LDU의 또 다른 최적화 기법은 update-side removing 때문에 취소된 garbage log를 재활용하는 것이다. 이 garbage log는 queue에는 존재하지만 update-side aborsrbing 때문에 취소 되어 garbage가 되어 queue에 보관되어 있는 log이다. 이러한 garbage log일 경우 다음 update operation에 대해서는 해당 로그를 새로 많들어 넣지 않고 기존 로그를 재활용하는 방법이다. 예를 들어 A라는 오브젝트에 대해서 insert-remove-insert 순서로 update가 수행될 경우, 세번째 insert 명령어는 queue에 들어가 있지만 update-side absorbing 때문에 최소되어 insert mark filed가 zero인 경우(garbage object)이다. 이러한 경우, 다음 insert operation에 대해서는 새로운 log를 list에 연산으로 다시 넣지 않고, 해당 object의 mark 필드만 변경하여 log를 재활용하는 방법을 사용한다.

LDU는 queue의 위치에 대한 의존성을 없애기 위해 log를 항상 non-blocking queue에 저장한다. LDU는 head 포인터에 대한 CAS 연산을 최대한 줄인 multiple producers and single consumer에 기반하는 non-blocking queue를 이용하였다. 이러한 큐는 이미 Linux 커널에 Lock-less list [25]라는 이름으로 구현되어 있으며, 이미 커널에 많은 부분에 사용되고 있다. 이 큐는 다른 non-blocking list[][][]와 다르게, insert operation을 항상 처음 노드에 삽입함에 따라, CAS가 발생되는 횟수를 상대적으로 줄일 수 있다. 게다가 log를 적용하는 single consumer만 고려 했기 때문에, remove를 위한 복잡한 알고리즘이 필요없다. 예를 들어 single consumer는 operation log 전체를 얻기 위해, xchg 명령을 사용하여 head pointer를 NULL로 atomic하게 제거한다. 결론적으로 LDU는 queue의 위치에 따른 의존성을 제거하고, head pointer의 CAS fail 때문에 발생하는 multiple CAS를 최소화 하기 위한 non-blocking queue를 이용하였다.

LDU는 log를 저장하기 위해, global 또는 per-core queue 모두 이용할 수 있게 설계하였다. 즉 저장하는 방법이 global queue이면 global non-blocking queue를 이용하고, per-core queue이면 per-core 메모리에 non-blocking queue를 저장하여 사용하였다. 이러한 두가지 queue를 지원하도록 한것은 서로 장단점을 가지고 있기 때문이다. 예를 들어 일반적으로 per-core 방법이 굉장히 높은 scalability를 가진다. 하지만 data structure의 head를 가지는 root object가 굉장히 많이 생성되는 워크로드일 경우, per-core 방법은 너무 많은 메모리를 요구한다.

이와 같이 LDU의 두가지 queue는 서로 장단점을 가지고 있다. 먼저 global queue의 장점은 광장히 simple하여 어떻한 data structure간에 쉽게 적용이 가능하다. global queue를 사용한 방법은 LDU의 light-weight한 queue사용과 update-side absorbing log와 reuse garbage log 기법등으로 global head 포인터에 대한 CAS operation 사용을 줄였지만, 여전히 global header에 CAS가 사용되고 있다. 따라서 cache coherence traffic issue가 여전히 남아있어서 scalability 측면으로는 단점을가지고 있다. Per-core queue일 경우 global head 포인터에 대한 CAS operation을 완전히 제거한 장점을 가진다. 하지만 LDU의 per-core queue 버전은 특정 data structure에는 사용 하지 못하는 data structure의 dependency가 있는 문제가 있다. 즉 LDU를 사용하면 time-senstivive log가 제거되었더라도 stack과 queue와 같이, 같은 operation에 대한 순서도 중요한 data structure는 사용하지 못한다.

LDU는 log 때문에 불필요하게 메모리 낭비를 방지하고, 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 flush해줌으로써 Log가 쌓여서 발생하는 메모리 낭비를 줄인다.

C. LDU example

먼저 LDU를 보다 정확하게 설명하기위해, 로그를 global queue 저장한 방법과 per-core queue에 저장한 두가지 방법에 대해서 예를 들어 설명한다. 그림 2 LDU의 하나의 예에 대해서 보여준다. 그림 2은 예로서 queue를 global queue를 사용하였다. 7개의 update operation과 1개의 read operation을 LDU에서 어떻게 처리하는지 보여준다. update operation에 대해서는 앞에서 설명한 심볼을 사용했고 garbage object을 위한 심볼이 추가 되었다. garbage object는 D과 같이 rectangle로 표시한다. 이 그림의 실행 순서는 위에서 부터 아래이다. 그림의 왼쪽은 CPU의 operation 순서를 보여주고 오른쪽은 특정 시간에 메모리에 저장된 자료구조의 내용을 보여준다. 초기 data structure의 트리에는 A and B 두개의 오브젝트가 들어가고, global queue에는 아무것도 없다.

그림 2는 시간 순서대로 각단계에 대해서 LDU의 수행 흐름을 보여준다. 먼저 그림에서 concurrent updates 단계에서는 `Core0`, `Core1` and `Core2`가 각각 ⊕C, ⊕D and ⊕E에 해당하는 3개의 updates oepration을 log로 저장한다. Log를 저장하는 queue를 non-blocking queue를 사용하기 때문에, 이 단계에서의 update는 lock이 필요없다. 즉 모든 thread는 lock에 대한 contention없이 병렬적으로 실행된다. `T1` 시점이 되면 메모리의 트리는 A and B의 노드를 가지게 되고, queue에는 ⊕C, ⊕D and ⊕E가 insert mark 필드가 true로 된 상태로 저장되어 있다. update-side absorbing 단계에서는 ⊖D명령이 수행되면, 새로운 object를 queue에 넣지 않고, mark 필드만 atomic하게 true로 수정한다. 만약 처음 수행되는 ⊖A operation이면 queue에 넣는다. `T2` 시점이 되면 메모리의 트리는 변함없고, queue에는 ⊕C, ⊕D, ⊕E and 가 ⊖A 저장된다. 여기서 D는 mark필드가 false로 되어 garbage가 된 object이다. reuse garbage log 단계에서는 D와 같이 queue에는 들어가 있지만, garbage object인 경우 새롭게 queue에 operation log를 넣지 않고, mark 필드 변경으로 재사용하는 것이다. 따라서 D는 D 상태로 변경된다. `T3` 시점이 되면 queue에는 ⊕C, ⊕D, ⊕E and 가 ⊖A 저장된다. 마지막 단계인 저장된 log를 실행하는 단계에서는 tree에서 사용한 방법으로 lock를 건후 하나의 코어에서 전체 로그를 update한다. 여기서 garbage log인 ⊕D을 제외하고 queue에 저장된 순서대로 ⊕C,⊕E and

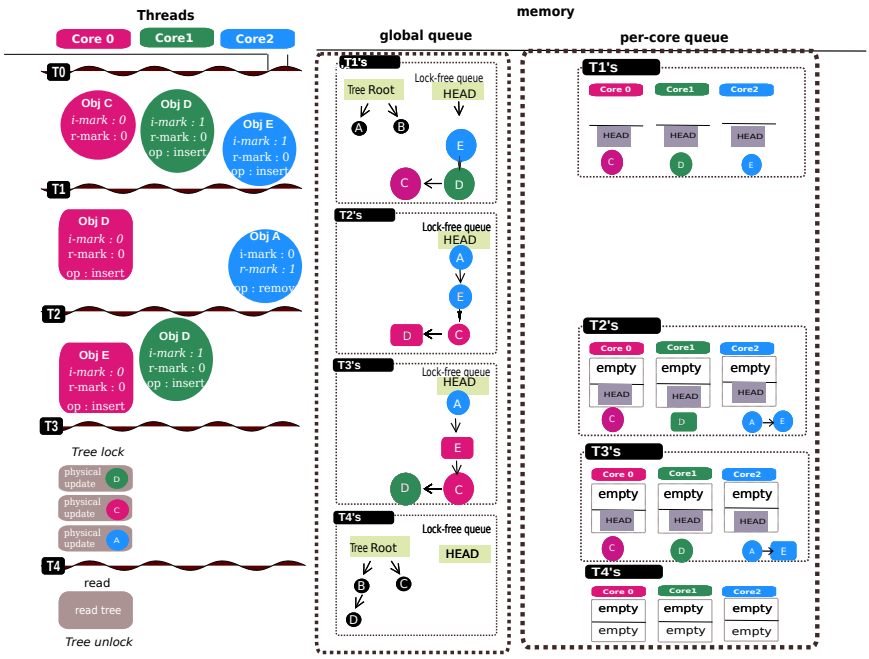


Figure 2: LDU example showing six update operations and one read operation. The execution flows from top to bottom. Memory represents original data structure and logging queue at T1, T2 and T3, respectively.

가 ⊖Ⓐ의 operation을 수행한다. T4 시점이 되면 tree에는 모든 operation이 수행되어, Ⓑ, Ⓒ and Ⓓ 값이 존재하게 된다. 결국 로그를 모두 수행하면 마지막 reader는 같은 tree의 값을 읽게 된다.

그림 **??**는 per-core queue에 log를 저장되는 흐름을 설명한다. global queue와의 차이점만 설명하면,

D. The LDU Algorithm

1) logical update: inserting logs: 그림 3는 concurrent updates를 수행하는 코드와 앞에서 설명한 update-side absorbing logs와 reuse garbage logs에 대한 코드이다. `logical_insert`과 `logical_remove`의 코드에서 가장 먼저하는 것은 이미 update operation이 수행되어 log가 queue에 들어 있는지에 대해서 체크를 한다. 이것은 update함수가 수행하는 도중 어느때나 log를 flush하는 `synchronize` 함수가 수행 될 수 있기 때문에, `xchg` 함수를 통해 atomic하게 체크를 한다. 또한 `xchg`를 통해 old mark 필드가 true라면 이미 queue에 log가 있다는 뜻이므로, atomic하게 false로 수정을 한다. 이것은 앞에서 설명한 update-side absorbing log 기능이며, 1개의 atomic operation으로 update를 수행할 수 있다. 만약 mark 필드가 체크되지 않았다면, 두번째로 queue에 존재하는 로그인지 체크를 한 후, 존재하는 로그라면 이미 앞에서 mark필드를 설정하여 이미 로그를 재활용했으므로 queue에 로그를 넣지 않는다. 만약 처음 사용된 log라면 non-blocking queue에 넣는다. 이 코드는 항상 옳게 동작한다. 그 이유는 리눅스 update operation이 항상 insert-remove 순서나 remove-insert 순서로 실행되기 때문이다.

2) physical update: applying logs: 그림 4는 concurrent updates를 통해 저장된 log들을 적용하는 deferred update 함수에 대한 코드이다. `synchronize_ldu`는 마치 CoW처럼 read 전에 호출되기도 하지만, 로그가 쌓이는것을 방지하기 위해 주기적으로 호출되어 로그를 비워준다. 이 함수는 항상 object의 lock이 걸린 상황에서 수행된다. LDU는 single consumer가 독점적으로 log를 처리한다. 따라서 제일 먼저 하는일은 queue에 저장된 log를 얻기 위해 atomic xchg operation으로 log의 head 포인터를 얻는다. 그리고 이 순간 부터는 single consumer가 독점적으로 log를 처리하게 된다. LDU는 주기적으로 log를 비워줘야 하므로, update operation이 수행되는 도중에도 `synchronize_ldu`가 호출될 수 있다. 따라서 항상 origianl data structure에 적용하기 전에 mark 필드를 초기화해야한다. 또한 log를 사용했으면, used 비트를 초기화 하여 해당 log가 queue 없음을 표시한다. 하지만 mark 필드 초기화 과정과 used 비트를 초기화 하는 과정에서, reuse garbase logs 기능 때문에, 언제든지 다시 concurrent updates operation을 통해 mark 필드가 설정될 수 있다. 따라서 한번 더 mark 필드를 체크하여 log를 비워준다.

3) LDU queues: LDU의 로그를 저장하기 위한 2가지 queue에 대한 코드는 그림 5와 같다. 앞에서 설명했듯이 LDU는 어떤 queue를 사용해도 문제 없도록 설계하였다. 따라서 queue 대한 의존성이 없다. global queue를 사용할 경우 LDU는 항상 head의 first 포인터에 CAS연산으로 성공할 때까지 반복 수행 후 데이터를 넣는다. per-core

```
bool ldu_logical_insert(struct object_struct *obj,
    void *head)
{
  ...
  // Phase 1 : update-side removing logs
  // atomic swap due to synchronize update's logs
  if(!xchg(&obj->remove->mark, 0)){
    BUG_ON(obj->insert->mark);
    insert->mark = 1;
    // Phase 2 : reuse garbage log
    if(!test_and_set_bit(LDU_INSERT,
        &obj->ldu.used)){
      // Phase 3 : insert log to queue
      //...save argument and operation
      ldu_insert_queue(root, insert);
    }
  }
  ...
}

bool ldu_logical_remove(struct object_struct *obj,
    void *head)
{
  ...

  // Phase 1 : update-side removing logs
  // atomic swap due to synchronize update's logs
  if(!xchg(&obj->insert->mark, 0)){
    BUG_ON(obj->remove->mark);
    remove->mark = 1;
    // Phase 2 : reuse garbage log
    if(!test_and_set_bit(LDU_REMOVE,
        &obj->ldu.used)){
      // Phase 3 : insert log to queue
      //...save argument and operation
      ldu_insert_queue(root, insert);
    }
  }
  ...
}
```

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.

queue를 사용할 경우는 per-core의 hash 테이블을 가지고 온 후 hash 테이블이 넣으려는 log의 data structure와 같은지 체크하고, 다르면 해당 코어에 저장된 log를 lock과 함께 update를 수행한다. 전체 코어의 log를 flush할 필요없이 해당 코어에 저장된 log만 flush하면 되는 이유는 이미 update-side absorbing에 의해 time-sencitive한 operation이 삭제되었기 때문이다.

IV. Concurrent updates for Linux kernel

A. Case study:reverse mapping

리눅스 커널의 프로세스간 공유자원 중 하나인 reverse page mapping(rmap)은 fork, exit, mmap이 수행될 때 update가 많이 발생하는 data structure이다. Linux's reverse mapping reads walked through each process's pages mappings and selected pages to unmap when it swaps a pyhsical page out to disk, migrates other cpu, or turncates

```
void synchronize_ldu(struct obj_root *root)
{
  ...

  //atomic remove first, lock-less list
  entry = xchg(&head->first, NULL);

  //iterate all logs
  llist_for_each_entry(dnode, entry, ll_node) {
    //get log's arguments
    ...
    //atomic swap due to update-side removing
    if (xchg(&dnode->mark, 0))
      ldu_physical_update(dnode->op_num, arg,
          ACCESS_ONCE(dnode->root));
    clear_bit(dnode->op_num, &vma->dnode.used);
    // one more check due to reuse garbage log
    if (xchg(&dnode->mark, 0))
      ldu_physical_update(dnode->op_num, arg,
          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.

a file. After all a page's mappings were removed could it be selected for pageout [**?**]. Rmap의 anonymous page와 file page의 관리는 최근 interval trees로 되어 있으며, 이것은 reverse page 성능 향상으로 위해 지속적으로 최적화가 [26] [27] 이루어지고 있다.

리눅스는 rmap의 interval tree를 보호하기 위해 rw semapore를 사용한다. 따라서, 프로세스들이 fork, exit, mmap를 simultaneosly 수행하면 interval tree를 보호하는 read-write semapore 때문에 scalability가 떨어진다. anonymous page를 위한 rmap과 file mapped page을 위한 rmap 모두 문제가 있다. 이번 장은 이러한 문제를 해결하기 위해, 어떻게 LDU를 리눅스의 rmap에 적용했는지에대한 내용을 설명하며, 보다 practial한 내용을 다룬다.

B. file mapping

file을 위한 rmap의 구조는 그림 x과 같다. reverse page map을 위한 page는 inode가 가지고 있는 address_space를 가리키며, 이것은 vm_area_struct(vma)을, containg the virtual address of the mapping, interval tree를 이용하여 관리한다. 이것은 fork가 되면 부모의 address_space의 interval tree에 추가된다. Tree를 보호하기 위한 lock은 프로세스마다 공유하는 데이터인 address_space의 lock을 사용하여 보호되며, anonymous ramp의 lock보다 contention이 덜 발생 하지만, file ramp역시 fork, exit과 mmap을 자주하는 워크로드 일 경우, scalability의 문제를 만든다[].

concurrent updates 때문에 발생하는 lock contention을 제거하기 위해 LDU를 사용하려면, 각각의 object에 insert와 remove에 해당하는 mark필드가 추가되야하며,

```
bool insert_log_global_queue(struct obj_root *root,
    struct ldu_node *dnode)
{
  ...
  //do {
  //  new_last->next = first = head->first;
  //} while (cmpxchg(&head->first, first, new_first) != first);
  llist_add(&dnode->ll_node, &p->list);
  ...
}

bool insert_log_per_core_queue(struct obj_root *root,
    struct ldu_node *dnode)
{
  ...

  slot = &get_cpu_var(obj_root_slot);
  p = &slot->obj[hash_ptr(root, HASH_ORDER)];
  empty = READ_ONCE(p->list.first);
  // is empty list?
  if (!empty) {
    ldu = llist_entry(first, struct ldu, ll_node);
    // is hash complict?
    if (ldu->root != dnode->root) {
      ...
      lock = READ_ONCE(ldu->lock);
      entry = xchg(&p->list->head->first, NULL);
      llist_add(&dnode->ll_node, &p->list);
      // flush log as a direct mapped cache
      object_lock(&lock);
      synchronize_ldu(entry);
      object_unlock(&lock);
      goto out;
    }
  }

  llist_add(&dnode->ll_node, &p->list);
out:
  ...
}
```

Figure 5: LDU physical update algorithm. `synchronize_ldu` may be called by reader and converts update log to original data structure traversing the lock-less list.

queue 종류에 따라 head 포인터를 추가해야 한다. 만약 global queue를 사용할 경우에는 address_space object에 global queue에 대한 head pointer를 추가하고, percore queue를 이용할 경우에는, 각각의 per-core memory에 추가하여 로그를 저장한다.

C. anonymous mapping

Anonymous rmap의 공유데이터는 서로 상당히 복잡하게 연결되어 있다. 그림 x-x는 이렇게 복잡하게 연결된 공유데이터를 보여준다. fork가 수행되면 부모의 anon_vma_chain(avc)는 복사가 되며 해당 avc를 관리하는 새로운 anon_vma가 생성된다. 여기서 Anonymous rmap의 file rmap과 다른점은 자식에 해당하는 anon_vma에 대한 tree operation을 수행할 때 사용되는 lock은 root가 가지고 있는 lock을 이용한다. 따라서 자식이 많아지는 경우 root의 lock 때문에 심한 lock contention이 발생한다[].

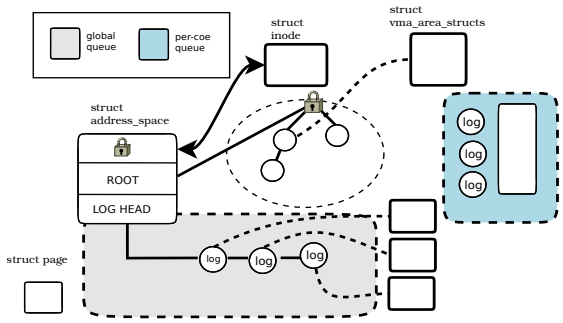


Figure 6: An example of applying the LDU to file reverse mapping.

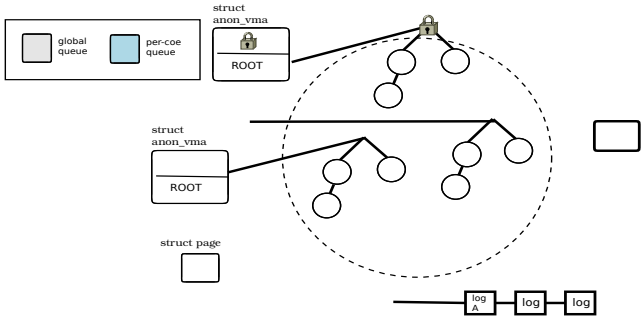


Figure 7: An example of applying the LDU to file reverse mapping.

이러한 lock contention을 제거하기 위해 LDU는 각각의 anon_vma_chain object에 대해서 mark 필드를 추가하여, update-side removing을 수행한다. anonymous rmap역시 global queue를 사용한다면, lock을 root의 lock을 사용하기 때문에, log의 위치는 그림 x-x와 같이 root의 anon_vma에 해당하는 object에 로그를 저장한다. 만약 percore queue를 사용하면, percore에 위치에 root 포인터 정보와 함께 operation log를 저장한다.

V. Implementation

We implemented the new deferred update algorithm in Linux 4.5.rc6 kernel, and our modified Linux is available as open source. global queue를 사용한 버전일 경우 코드의 수정량은 xx-xx을 가지며 덜 복잡한 구조이며, percore를 사용할 경우 코드 수정량은 xx-xx을 가진다.

LDU percore queue는 2가지 방법으로 구현되었다. 첫번째 방법은 percore hash table로 구현하는 방법이다. percore hash table 방법은 direct-mapped cache 처럼 구현하여, 버켓에는 하나의 object만 존재하도록 만든 방법이다. 만약 hash 충돌이 발생하면 기존 해당 코어의 log를 flush하는 일을 수행한다. 이것은 file reverse mapping 처럼 object가 많이 생기지 않을 때 유용하다. 이 방법은 기존 코드에 대한 수정없이 적용이 가능하고 추가적인 lock이 필요없다. 하지만 이 방법은 anonymous reverse mapping 처럼 object가 많이 생기는 경우, hash 충돌로 인

해 성능상 오버헤드가 생긴다. 따라서 anonymous reverse mapping의 percore queue는 percore data에 object별로 구분하지 않고 로그를 저장한 후 실제 percore log를 수행할 때 global lock을 사용하여 보호하였다.

VI. Evaluation

This section answers the following questions experimentally:

- Does LDU's design matter for applications?
- Why does LDU's scheme scale well?
- What about LDU's read-write ratio?

A. 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.

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 V). 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 [28];therefore, we implement the Harris linked list to Linux kernel. The code refers from sysnchrobench [29] and ASCYLIB [21], 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.

We ran the three benchmarks on Linux 4.5.rc6 with stock Linux. All experiments were performed on a 120 core machine with 8-socket, 15-core Intel E7-8870 chips equipped with 792 GB DDR3 DRAM.

B. 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.

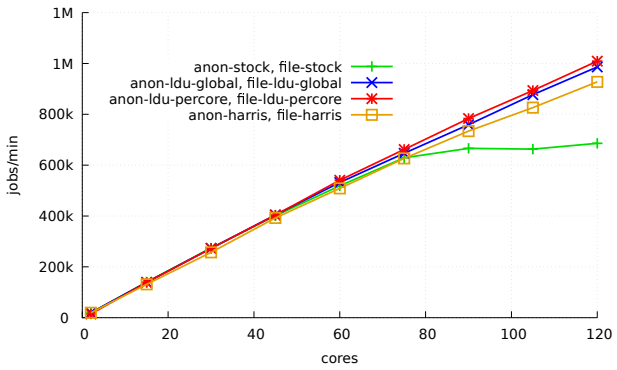
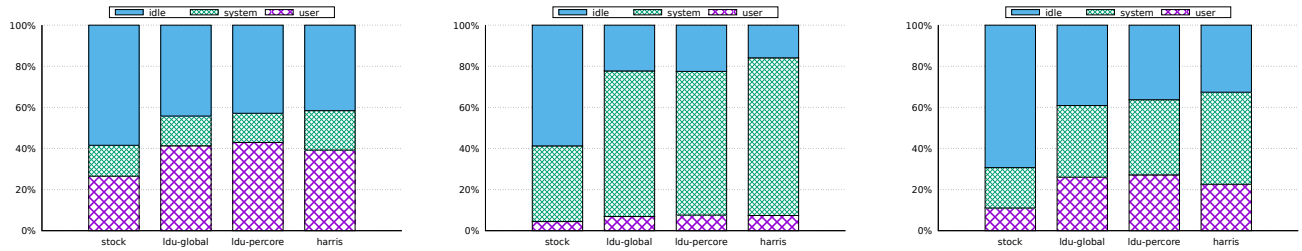


Figure 8: Scalability of AIM7 multiuser for different method. The combination LDU with unordered harris list scale well;in contrast, up to 60 core, the stock Linux scale linearly, then it flattens out.

The results for AIM7-multiuser are shown in Figure 8, 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 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 **??**), 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.

C. Exim

To measure the performance of Exim, shown in Figure 9, 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 [1]. Therefore, although we eliminate the fork problem, the Exim may suffer from contention on spool directories.



(a) AIM7 - 120core (b) Exim - 120core (c) Lmbench - 120core

Figure 10: Read-write ratio from 50:50 to 1:99 percent

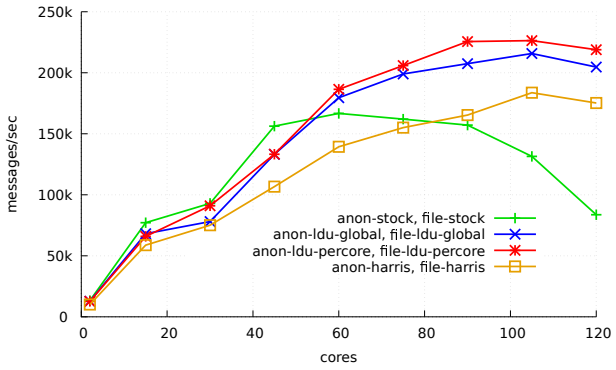


Figure 9: Scalability of Exim. The stock Linux collapses after 60 core;in contrast, both unordered harris list and our LDU flatten out.

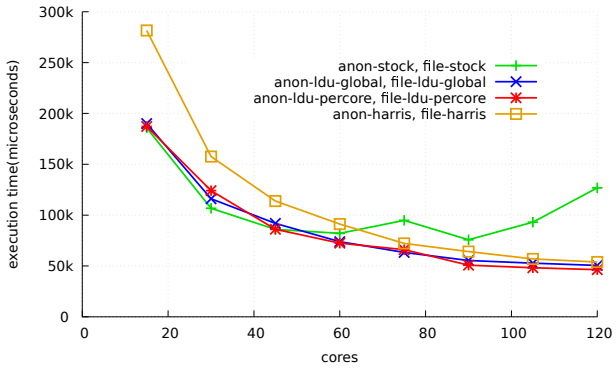


Figure 11: Execution time of lmbench's fork micro benchmark. The fork micro benchmark drops down for all methods up to 15 core but either flattens out or goes up slightly after that. At 15 core, the stock Linux goes up;the others flattens out

Results shown in Figure 9 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.

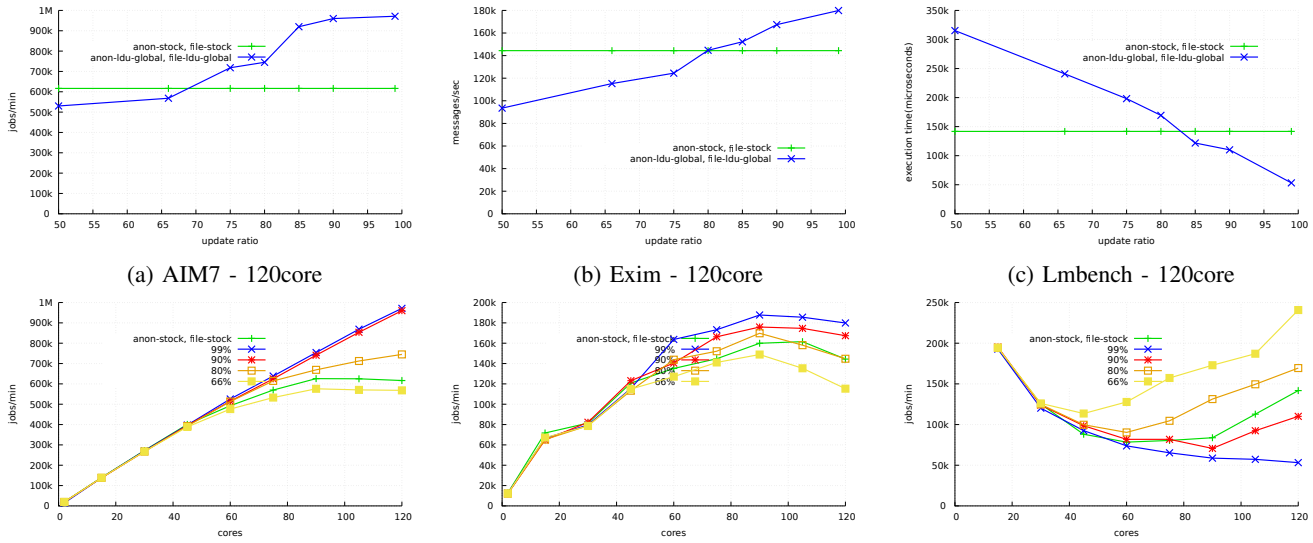
D. Lmbench

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 [14]; we used 100 processes.

The results for lmbench are shown in Figure 11, 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`.

E. Updates ratio

LDU는 high update rate operation가지는 update-heavy한 data structure를 위한 방법이다. 즉 linux kernel의 rmap과 같이 read가 간혈적으로 발생하는 data structure를 대상으로 사용될 때 굉장히 높은 scalabilty를 보여준다. 우리는 어느 정도의 update rate을 가진 data structure에 LDU를 적용하면 효율적인지 판단하기 위해, 리눅스의 rmap을 대상으로 log를 적용하는 read operation을 추가하여 성능과 확장성을 비교하는 실험을 하였다. 보다 범위를 좁혀 구체적인 실험 결과를 보기 위해, anonymous rmap은 ldu를 적용한 버전을 대상으로 실험을 하였고, file ramp에서 update operation(insert or remove)이 수행할 때 read operation(lock + synchronize)을 비율에 맞게 추가



(a) AIM7 - 120core (b) Exim - 120core (c) Lmbench - 120core

(d) AIM7 - scalability (e) Exim - scalability (f) Lmbench - scalability

Figure 12: Read-write ratio from 50:50 to 1:99 percent

하여 실험하였다.

그림 ¡x-x¿의 위의 그래프들은 120코어에서 update rate에 따른 성능을 보여주고 아래 그림은 rmap의 update rate에 따른 확장성을 보여준다. AIM7은 EXIM과 Lmbench와 다르게 보다 덜 fork-intensive한 특징을 가지므로, log를 적용하는 read operation 상대적으로 덜 호출된다. 따라서 75프로(3 update operation + 1 read)의 update rate을 가지는 data structure에도 stock 리눅스보다 높은 성능과 확장성을 가진다. Scalability를 보면 90프로 이상의 update rate를 가질 때는 상당히 높은 scalability을 가지며 80프로 이상의 update rate를 가질 경우에도 scalability가 약간 떨어지나, stock 리눅스보다는 높은 성능을 가진다.

Exim과 Lmbench 경우에는 AIM7 보다 더 fork-intensive한 워크로드 특징을 가진다. Log를 적용하는 read operation이 상당히 짧은 주기로 호출됨에 따라, AIM7보다 높은 85프로 이상의 update rate를 가질 때 디폴트보다 높은 성능을 가진다. Lmbench의 scalability의 경우 log를 merge하는 read operation이 호출되는 주기가 짧아짐에 따라, 비록 update rate이 90프로를 가져도 코어 수가 90 core 이상에서 scalability가 떨어지나, stock 리눅스 보다는 높은 성능을 가진다.

VII. Discussion

우리는 리눅스 커널과 같이 update operation이 insert-insert또는 remove-remove와 같이 같은 operation에 대해서 발생하지 않는 data structure의 경우에 대해서만 사용가능하도록 LDU를 구현하였다. 이러한 방법은 굉장히 practical한 방법으로 리눅스와 freeBSD와 같은 커널에는 적용 할 수 있으나 보다 연구 중심적인 data structure인 CSDS 알고리즘과 비교 실험하는데는 무리가 있다. 따라서 LDU 또는 Oplog와 같이 log-based 방법을 보다 연구 중심적인 data structure와 비교 실험이 가능하도록 보다 일반화할 필요가 있다.

VIII. Related work

Operating system scalability. [15]In order to improve Linux scalability, researchers have been optimized memory management in Linux by finding and fixing scalability bottlenecks [30] [31]. 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 [32] solved this address space problem by using the RCU; RadixVM [33] 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 [1].

Scalabe data structure. [4]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 [6] 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 [3] 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.

Scalable lock. [34] [16] [17]One method for the concurrent update is using the non-blocking algorithms [28] [35] [36], which are based on CAS.

MCS [37], a scalable exclusive lock, is used in the Linux kernel [38]. 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 [39] 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 [40] 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.

IX. 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 4.5 and available as open-source from https://github.com/manycore-ldu/ldu.

References

[1] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of Linux scalability to many cores," in *9th USENIX Symposium on Operating System Design and Implementation*, Vancouver, BC, Canada, October 2010, pp. 1–16.

[2] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding manycore scalability of file systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 71–85. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/min

[3] A. Matveev, N. Shavit, P. Felber, and P. Marlier, "Read-log-update: A lightweight synchronization mechanism for concurrent programming," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, New York, NY, USA, 2015, pp. 168–183.

[4] M. Dodds, A. Haas, and C. M. Kirsch, "A scalable, correct time-stamped stack," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 233–246. [Online]. Available: http://doi.acm.org/10.1145/2676726.2676963

[5] P. E. McKenney, "Is parallel programming hard, and, if so, what can you do about it?" 2011.

[6] M. Arbel and H. Attiya, "Concurrent updates with rcu: Search tree as an example," in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC '14, New York, NY, USA, 2014, pp. 196–205.

[7] O. Shalev and N. Shavit, "Predictive log-synchronization," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 305–315, Apr. 2006. [Online]. Available: http://doi.acm.org/10.1145/1218063.1217965

[8] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 355–364. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810540

[9] S. Boyd-Wickizer, "Optimizing communications bottlenecks in multiprocessor operating systems kernels," in *PhD thesis, Massachusetts Institute of Technology*, 2013.

[10] P. McKenney, "Msome more details on read-log-update," 2016, https://lwn.net/Articles/667720/.

[11] "Aim benchmarks," http://sourceforge.net/projects/aimbench.

[12] "Exim internet mailer," 2015, http://www.exim.org/.

[13] "Mosbench," 2010, https://pdos.csail.mit.edu/mosbench/mosbench.git.

[14] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[15] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, pp. 10:1–10:47, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699681

[16] D. Bueso, "Scalability techniques for practical synchronization primitives," *Commun. ACM*, vol. 58, no. 1, pp. 66–74, Dec. 2014. [Online]. Available: http://doi.acm.org/10.1145/2687882

[17] D. Bueso and S. Norto, "An overview of kernel lock improvements," 2014, http://events.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf.

[18] C. Rohland, "Tmpfs is a file system which keeps all files in virtual memory," 2001, git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/tmpfs.txt

[19] T. C. Andi Kleen, "Scaling problems in fork," in *Linux Plumbers Conference, September*, 2011.

[20] D. H. Tim Chen, Andi Kleen, "Linux scalability issues," in *Linux Plumbers Conference, September*, 2013.

[21] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, New York, NY, USA, 2015, pp. 631–644.

[22] S. Al Bahra, "Nonblocking algorithms and scalable multicore programming," *Commun. ACM*, vol. 56, no. 7, pp. 50–61, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2483852.2483866

[23] E. Petrank and S. Timnat, "Lock-free data-structure iterators," in *DISC '13 Proceedings of the 27th International Conference on Distributed Computing, Jerusalem, Israel*, 2013.

[24] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. Spear, "Practical non-blocking unordered lists," in *DISC '13 Proceedings of the 27th International Conference on Distributed Computing, Jerusalem, Israel*, 2013.

[25] H. Ying, "Lock-less list," 2011, https://lwn.net/Articles/423366/.

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

[27] ——, "The case of the overly anonymous anon_vma," 2010, https://lwn.net/Articles/383162/.

[28] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01, London, UK, UK, 2001, pp. 300–314.

[29] V. Gramoli, "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015, New York, NY, USA, 2015, pp. 1–10.

[30] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 43–57. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855745

[31] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous." in *In Proceedings of the Linux Symposium*, 2012.

[32] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scalable address spaces using RCU balanced trees," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, London, UK, February 2012, pp. 199–210.

[33] ——, "Radixvm: Scalable address spaces for multithreaded applications," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, New York, NY, USA, 2013, pp. 211–224.

[34] T. Wang, M. Chabbi, and H. Kimura, "Be my guest: Mcs lock now welcomes guests," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 21:1–21:12. [Online]. Available: http://doi.acm.org/10.1145/2851141.2851160

[35] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '04, 2004, pp. 50–59.

[36] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank, "Wait-free linked-lists," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, New York, NY, USA, 2012, pp. 309–310.

[37] J. M. Mellor-Crummey and M. L. Scott, "Scalable reader-writer synchronization for shared-memory multiprocessors," in *Proceedings of the Third PPOPP*, Williamsburg, VA, April 1991, pp. 106–113.

[38] J. Corbet., *MCS locks and qspinlocks*, 2014.

[39] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with "readers" and "writers"," *Communications of the ACM*, vol. 14, no. 10, pp. 667–668, October 1971.

[40] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, Las Vegas, NV, October 1998, pp. 509–518.