A Lightweight Log-based Deferred Update for Linux Kernel Scalability

Abstract

We propose a novel light weight concurrent update 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.

1 Introduction

최근 코어수가 증가하고 있다. 따라서 멀티코어에서 매니코어 환경으로 바뀌고 있다. 이러한 매니코어 환경에서 리눅스 커널의 확장성에는 문제가 있다 [8] [31]. 확장성 문제 중 하나는 락 경합 때문에 발생하는 직렬화 문제이다 [25] [20]. 여러 락 때문에 발생하는 직렬화 문제 중 하나가 업데이트 오퍼레이션이기 때문에 발생하는 문제이다. 그 이유는 업데이트 오퍼레이션은 여러 쓰레드가 동시에 수행되지 못하기 때문이다 [27]. 예를 들어 리눅스 커널의 프로세스간 공유하는 자료구조인 역 매핑은 프로세스를 생성할 때 높은 업데이트 비율을 가진다 [6]. 따라서 리눅스 커널은 역 매핑에 대한 업데이트 락에 때문에 직렬화되어, 새로운 프로세스를 생성 하는 과정에서 확장성 문제가 생긴다 [4] [35].

이처럼 업데이트 직렬화 문제를 해결하기 위해 여러 동시적 업데이트 방법 들이 연구되고 있다 [5] [25]. 이러한 동시적 업데이트 방법들은 워크로드 특성인 업데이트 비율에 따라 많은 성능 차이를 보인다 [25]. 이 중 리눅스 커널의 역 매핑과 같이 높은 업데이트 비율을 가진 자료 구조 때문에 발생하는 학장성 문제를 해결하기 위한 여러 방법이 연구 되고 있다. 그 중 하나는 캐쉬 coherence traffic을 줄인 log-based 알고리즘 [34] [24] [6]을 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 로깅된 내용을 자료구조에 적용하는 것이다 . 이것은 마치 CoW(Copy On Write)와 유사한 결과를 얻는다 [26].

S. Boyd-Wickizer et al.는 동기화된 타임스탬프 카운터(synchronized timestamp counters) 기반의 per-core log를 활용하여 update-heavy한 자료구조에서 concurrent updates 문제를 해결하였다 [6]. 동기화된 타임스탬프 카운터 기반의 per-core log를 활용한 concurrent updates 방법은 update 부분만 고려했을 때, per-core에 데이터를 저장함으로 굉장히 높은 scalability를 가질 수 있다[]. 하지만 per-core 기반의 synchronized timestamp counters를 사용한 방법은 결국 timestamp ordering and merging 작업을 야기한다. 만약 코어 수가 늘어 날 경우, 로그를 자료구조에 적용하는 과정에서 추가적인 sequntial 프로세싱이 요구된다. 이것은 확장성과 성능을 저해한다. 또한 per-core 방법은 제한된 per-core 메모리 사이즈 때문에 모든 자료구조에 적용되지 않는다. per-core 방법은 자료구조에 따라 심한 메모리 사이즈가 요구된다[].

본 논문은 높은 업데이트 비율을 가진 자료구조를 위한 새로운 방법인 LDU(Lightweitgh log-based Deferred Update)를 제안한다. LDU는 동기화된 타임스탬프 카운터를 이용함에 따라 생기는 정렬과 머징 오버헤드에 대한 문제를 줄이기 위해, 조금 더 간단하고 경량화한 방법을 사용하였다. LDU의 철학은 FC(Flat Combiner)[] 또는 OpLog[]와 같이 분산 시스템에서 사용하는 log기반의 concurrent updates 방식과 shared-memory system의 최소한의 hardware-based synchronization 기법(예를 들어, compare and swap, test and set, atomic swap)을 이용하여 이 문제를 해결 하였다. 먼저 per-core 방식이기 때문에 발생하는 data structure dependency가 있는 문제를 해결하기 위해, log를 global queue에 저장하는 GLDU와 다음으로 global queue 때문에 발생하는 cache coherece traffic을 줄이기 위해, per-core에 log를 저장하는 PLDU 두가지 버전으로 개발하였다. 이것은 서로간 상호보

완적이다. 또한 새로운 update-side absorbing과 reuse garbage log등 2가지 optimization 기술을 사용하여 삭제 가능한 operation log를 지우고 재활용하는 방법으로 성능 최적화를 이뤘다.

이처럼 synchronized timestamp counters를 제거함과 동시에, cache communication bottleneck 줄인 LDU는 전형적인 log-based 알고리즘의 장점을 모두 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 [24]. 셋째로, 기존 여러 자료구조에 쉽게 적용할 수 있는 장점이 있다. 마지막으로 저장된 log를 수행하지 전에 여러 optimization 방법을 사용하여 log의 수를 줄일 수 있다.

우리는 위와 같은 장점을 가지는 LDU를 리눅스 커널의 fork scalability 문제를 야기시키는 2가지 reverse page mapping(anonymous pape mapping, file page mapping)에 적용하였다. 또한 우리는 LDU를 Linux 4.5.rc4에 구현하였고, fork-intensive 워크로드인 AIM7 [1], Exim [3] from MOSBENCH [2], lmbench [29]를 대상으로 성능 개선을 보였다. 개선은 stock 리눅스 커널에 비해 120코어에서 각각 x,x,x 배이다.

Contributions. This paper makes the following contributions:

- 우리는 리눅스 커널을 위한 새로운 log-based concurrnet updates 방법인 LDU를 개발하였다. LDU는 synchronized timestamp counters를 이용함에 따라 생기는 시간 정렬과 머징 overhead와 workload에 따른 interfaces dependency가 있는 문제를 줄였다. 이를 위해 LDU는 조금 더 simple하고 lightweight한 방법과 새로운 update-side absorbing과 reuse garbage의 optimization을 사용하여 log-based concurrent updates를 구현하였다.

- 우리는 LDU을 practical한 manycore system인 intel xeon 120코어 위에 동작하는 리눅스 커널에 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.

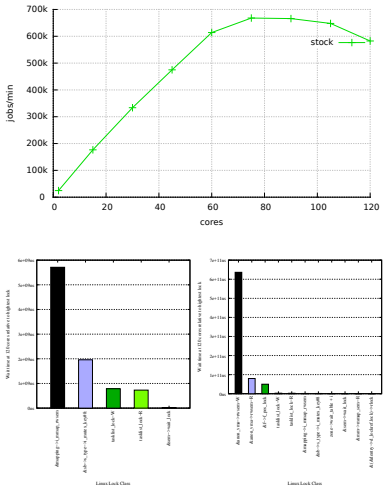

Figure 1: Scalability of AIM7 multiuser. This workload simultaneously create many processes. Up to 60 core, the stock Linux scale linearly, then they flattens out.

2 Background and Problem

운영체제 커널의 paralleism은 시스템 전체의 parallesim에서 가장 중요하다. 만약에 커널이 scale하지 않으면, 그 위에 동작하는 응용프로그램들도 역시 scale하지 않는다 [14]. 우리는 이처럼 중요한 부분인 multi-core에 최적화된 리눅스 커널의 scalabiliy를 분석하기 위해, AIM7 multiuser[]를 가지고 scalability를 실험해보았다. AIM7은 최근에도 scalability를 위해 reserach 진영과 리눅스 커널 진영에서도 활발히 사용되고 있는 벤치마크 중 하나이다 [11] [10]. File system scalability를 최소화 하기 위해 temp filesystem [33]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 Lockstat를 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 anonvma를 수많은 fork에 의해 프로세스를 생성하면서 발생하는 락 경합 문제이다. 다음으로 우리는 anonvma의 lock 경합을 줄이기 위해, fork에서 anonvma를 호출하는 부분을 제거한 후 pageswap이 안되도록 하고, 120코어를 대상으로 다시 Lock 경합을 분석하여 보았다. 이 때 부터 그동안 상대적으로 가려졌던 file reverse mapping에서 많은 락 경합이 발생되었다. anonvma reverse page mapping은 리눅스 커뮤니티에서 잘 알려진 락 경합 문제 [4] [35]이고, file mapping에 대한

락 경합 문제는 Silas wikizer가 OpLog 논문을 통해 fork의 scalability 문제의 원인으로 제시한 부분이다. 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.

이러한 revser page mapping은 page frame reclaiming을 위해 존재하며, 리눅스가 fork(), exit(), and mmap() 시스템콜을 사용할 때 rmap을 update한다. 두 가지 revserse page mapping의 근본적인 문제는 hight update operation에 대한 serialization 때문에 발생하는 문제이다. 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. 근본적인 해결 방법은 update heavy한 상황이 발생하지 않도록 reverse page mapping 알고리즘을 새롭게 디자인하는 방법이 있다. 하지만 이미 오랜 시간 성능과 안정성이 검증된 부분을 새롭게 디자인 하는 일은 쉽지 않다. 보다 현실적인 방법은 concurrent updates를 위한, non-blocking data structure와 log-based 알고리즘을 사용하는 방법이 있다. In non-blocking algorithms, 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. Both repeated CAS operation and their iteration loop caused by CAS fails cause bottlenecks due to inter-core communication overheads [6]. Deu to the multipul CAS, inter-core communication overheads를 줄인 log-based has researched. 이러한 log-based 기반 방법은 다음 장에서 자세히 다룬다.

3 LDU Design

LDU는 리눅스 커널의 high update rates를 가진 data strcuture의 scalability를 해결하기 위한 구조를 가진다. 이러한 LDU의는 log기반 방식의 concurrent updates 방법과 atomic synchronization 기능(i.e., compare-and-swap, test-and-set, etc)을 최소한으로 사용하도록 설계하였다. 이를 통해 concurrent updates가 가능하게 하였으며, 동시에 cache communication overhead를 최소화 하였다. 예를 들어, high update rates data structure의 scalability 문제를 해결하기 위해, LDU는 update operation 순간에는 operation 로그를 atomic하게 저장하여 concurrent updates를 가능하게 한다. 또한, LDU는 global atomic 연산에 따른 cache communication overhead를 줄이기(mitigating) 위해 2가지 최적화 방법(update-side abosrbing, reusing garbage)을 사용하였으며, log를 저장하는 위치에 따라 global queue를 이용한 방식인 GLDU와 per-core queue를 이용한 방식인 PLD로 설계하였다. 본 장에서는 이러한 LDU의 최적화 방법과 log 위치에 따른 LDU 알고리즘의 디자인 측면에 대해서 설명한다.

3.1 Log-based Concurrent updates

Update heavy한 구조 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 간혈적으로 수행되는 data structure에 적합한 방법이다.

Update heavy한 구조를 위해, 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 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.

3.2 Approach

LDU도 log-based approach를 따른다. 그러므로 앞에서 설명한 log-based 방법의 장점을 모두 가진다. LDU는 먼저 lock 없이 concurrent Updates를 수행하기 위해, update를 수행하는 시점에서 atomic하게 operation log를 global queue(GLDU) 또는 per-core queue(PLDU)저장 한다. 따라서 lock이 필요 없어 scalability가 뿐만 아니라 lock 자체의 오버헤를 줄일 수 있다. 저장된 Log는 불필요한 메모리 낭비를 줄이기 위해 주기적으로 flush하거나 또는 read 전에 호출되게 되는데, 이 때 LDU는 FC와 같이 operation Log를 corse grain lock과 함께 single core에서 수행한다. 따라서 cache의 miss를 줄여 성능을 향상시킨다. 다음으로 LDU는 Log-based 방법과 같이 기존 data structure를 많이 수정하지 않고도 다른 data structure에 쉽게 적용할 수 있다.

LDU의 하나의 타입 중 하나인 GLDU(Global queue version of LDU)는 log를 global queue에 저장하는 방법이다. global queue를 사용함에 따라, GLDU는 CAS operations on the head of the queue limit scalability(the bottleneck on the log is similar to that the Michael and Scott queue)[]. GLDU는 global queue의 사용에 대한

cache invalidate 줄이고, 불필요한 연산을 줄이기 위해, CAS를 최소화한 queue를 사용 하였고, update-side absorbing, reuse garbage log과 같은 3가지 최소화 기법을 사용하였다. 먼저 GLDU는 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 list[][][]와 다르게, insert operation을 항상 처음 노드에 삽입함에 따라, CAS가 발생되는 횟수를 상대적으로 줄일 수 있다. 게다가 single consumer를 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. 이러한 큐는 이미 Linux 커널에 Lock-less list라는 이름으로 구현되어 있으며, 이미 커널에 많은 부분에 사용되고 있다. 요약하자면, GLDU는 head pointer의 CAS fail 때문에 발생하는 multiple CAS를 최소화 하기 위해, Linux 커널에 있는 Lock-less list를 활용 하였다 [38].

LDU는 저장된 update log에 대해서 최적화를 수행하기 위해 update-side absorbing이라는 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 근본적인 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 경우 시간 순서되로 log를 merge 후 검색한 후 삭제를 위해야한다. 이것은 추가적인 오버헤드를 가지고 있다. 하지만 LDU는 indivisual object를 대상으로 상대적으로 가벼운 swap operation을 사용하여 shared log 삭제하는 방법을 사용했다.

LDU의 update-side absorbing은 shared memory system의 swap operation을 사용한다. 이를 위해, LDU는 모든 object에 insert와 remove의 mark 필드를 추가해서 update-side absorbing을 수행 하였다. 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 atomic하게 지워주는 방식으로 진행된다. 이것은 swap이라는 상대적으로 가벼운 연산과 상대적으로 덜 share 하는 indivisaul global object의 mark filed를 사용해서 log를 지워주는 효과를 가져주므로, cache communication overhead를 많이 발생시키는 global head 포인터에 CAS연산을 사용을 줄여 준다. 또한 앞으로 설명할 per-core기반의 concurrent updates를 가능하게 한다.

LDU의 또 다른 최적화 기법은 update-side absorbing 때문에 취소된 queue에 저장된 garbage log를 재활용하는 것이다. 이것은 queue에는 존재하지만 update-side aborsrbing 때문에 취소 되어 garbage가 되어 queue에 보관되어 있는 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 global operation list에 들어가 있지만 update-side absorbing 때문에 최소되어 insert mark filed가 zero인 경우(garbage object)이다. 이러한 경우, 다음 insert operation에 대해서는 list에 연산을 다시 넣지 않고, 해당 object의 mark 필드만 변경하여 log를 재활용하는 방법을 사용한다. 따라서 LDU는 두가지 최적화 기법을 사용하여 cache communication overhead를 줄였다.

LDU의 두번 째 타입은 Per-core에 log를 저장하는 PLDU이다. PLDU의 원리는 기본적으로 log를 per-core 메모리에 저장하되 timestamp가 필요한 log만 shared memory system의 atomic 연산으로 처리를 하자는 것이다. 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, so this condition needs the timestamp[].

즉 timestamp 없이 per-core로 merge된 로그가 아래와 같은 상황이 발생하면 문제가 발생한다.

여기서 흥미로운 사실은 같은 object에 대해서 insert-remove operation을 update 할 때 수행해 주면 결국 per-core에 남은 operation은 insert 또는 remove operation만 남게 된다. 즉 per-core log는 timestamp가 없어도 되는 operation만 남게 된다. PLDU는 로그를 update-side absorbing 기술로 로그를 바로 바로 지워줌으로 timestamp를 없이도 per-core log를 사용할 수 있도록 하였다.

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. 즉 LDU는 주기적으로 Log를 적용함으로 Log가 쌓여서 발생하는 메모리 낭비를 줄인다.

3.3 Limitation

PLDU도 일반적으로 per-core 기반의 log-based 방법의 문제점을 가진다. 일반적으로 OpLog를 포함한 Per-core 기반의 log-based 방법의 문제점은 does not work well for a all data structure. Per-core log-based 시스템의 어려운 점은 log의 저장 위치(header)가 update operation이 사용하는 global data structure가 아니라 per-core 메모리 공간과 구별되었기 때문이다. 따라서 log 위치가 global data strucutre와 separated 되었기 때문에 추가적인 hash table등을 사용하여 object별로 구별하여 관리 하거나, 여러 종류의 log를 하나의 per-core 메모리 공간에 저장 한 후 log를 사용할 마다 구별하여 처리해야 하는 방법이 필요하다. 만약

hash table 등을 사용하여 구별하여 저장할 경우는 많은 종류의 오브젝트가 생성되어 hash table 충돌이 많이 발생하는 data structure일 경우 해쉬 충돌로 인해 lock이 필요하므로 오히려 scalability 문제가 생긴다. 이 경우 per-core hash table의 크기를 늘리면 hash 충돌 발생은 줄게 되지만 per-core memory overhead가 큰 문제가 있다. 이와 반대로 일단 하나의 per-core memory에 저장한 후 차후 log를 ojbect별 분리 하는 경우이다. 이 경우는 mix되어 있는 로그를 구별하면서 해당 data structure만 보호하는 lock을 사용하기가 어려우므로, global lock을 사용해야한다. 결국 이 방법도 역시 global lock에 의한 scalability 문제가 있다. 결국 per-core 기반의 log-based 방법은 제한적이다. 즉 per-core 기반 log-based 방법은 does not work well for a all data structure.

PLDU만의 limitation은 reader가 봤을 때 data structure에는 문제 없이 존재하지만, 같은 insert operation간의 또는 같은 remove operation간의 순서가 변경되어 들어가 있기 때문에, operation의 순서를 중요시하는 stack, queue 같은 data structure는 사용하지 못하는 문제가 있다. 이러한 경우는 GLDU나 반드시 timestamp 기반의 방법[][]을 사용해야한다. 예를 들어 아래와 같이 ordering이 변경될 수 있다.

따라서 이런 문제가 있는 data structure일 경우 GLDU를 사용해야한다.

두 GLDU and PLDU의 공통적인 limitation은 리눅스 커널와 freeBSD 커널과 같이 search가 two update operations(one to insert and one to remove an element)이 함수 외부해서 수행하는 구조에서만 적용이 가능하다. 재미있는 사실은 리눅스의 data structure의 경우에는 search와 update가 분리되어 있어, 같은 update operation이 연달아 호출되지 않는다. 이러한 구조는 search가 외부에 있기 때문에 insert operation 다음에는 반드시 remove opration이 발생하고 remove operation 다음에는 반드시 insert operation이 발생하는 특징이 있다(search에서 이미 필터링이 되기 때문). 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. 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 연구되어온 CSDS 알고리즘들은 garbage collector[][], iteration[], ordering[]여러 practical한 이유로 C언어로 구현된 리눅스 커널 등에 적용하기에는 아직 한계가 있다[][][].

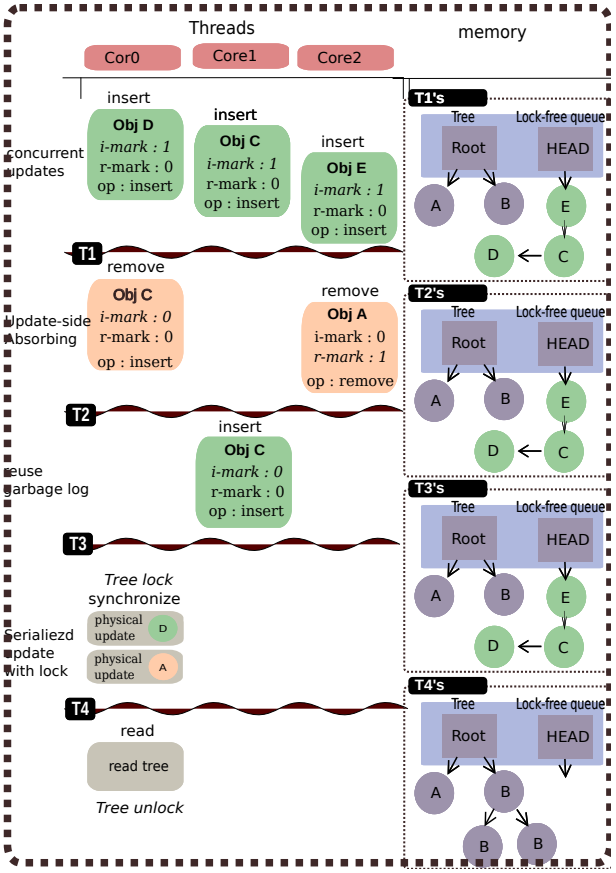

Figure 2: LDU example showing six update operations and one read operation. The execution flows from top to bottom. Memory represents original data structure and logging queue at T1, T2 and T3, respectively.

3.4 GLDU vs. PLDU

GLDU와 PLDU는 서로 상호 보완적인 관계이다. GLUD의 장점은 광장히 simple하여 어떻한 data structure간에 쉽게 적용이 가능하다. 그리고 PLDU와 같은 ordering 이슈도 없다. 단점으로는 global queue를 사용함에 따라 global head pointer를 동시에 수정하므로 cache coherence traffic issue가 있어서 scalability가 PLDU에 비해 약간 떨어진다. 반면에 PLDU는 GLDU보다 Scalability 측면으로는 더 좋지만 stack, queue와 같이 순서가 예민한 data structure에는 사용못하는 data structure의 dependency가 있는 문제가 있다.

3.5 LDU example

3.5.1 GLDU example

Figure 3 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 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.

3.5.2 PLDU example

Figure 3 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 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.

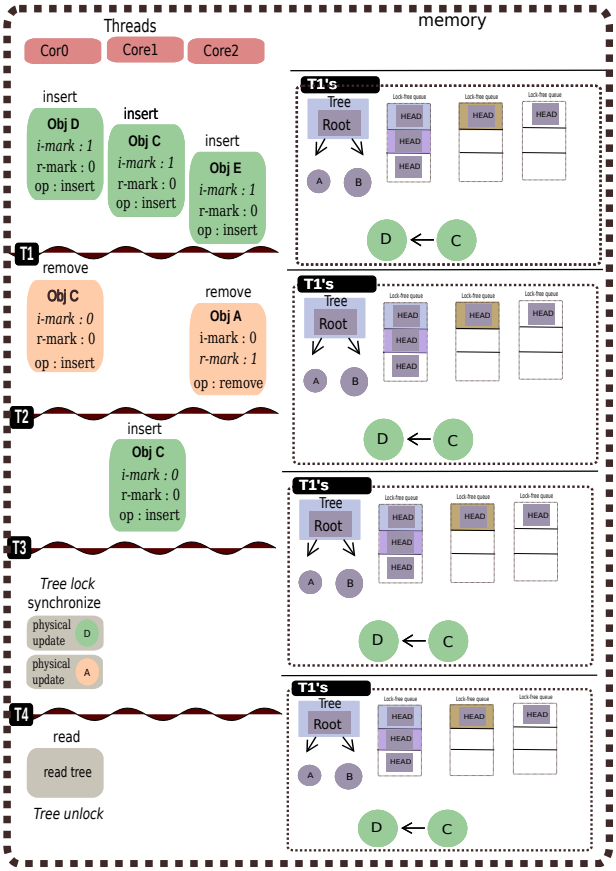

Figure 3: LDU example showing six update operations and one read operation. The execution flows from top to bottom. Memory represents original data structure and logging queue at T1, T2 and T3, respectively.

3.6 The LDU Algorithm

3.6.1 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.

```
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 4: 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.

3.6.2 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.

```
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 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.

3.6.3 The PLDU Algorithm

3.6.4 PLDU logical 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.

3.6.5 PLDU 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

```
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 6: 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 7: LDU physical update algorithm. `synchronize_ldu` may be called by reader and converts update log to original data structure traversing the lock-less list.

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.

4 Concurrent updates for Linux kernel

4.1 Case study:reverse mapping

리눅스 커널의 프로세스간 공유자원 중 하나인 reverse page mapping(rmap)은 fork가 수행될 때 update가 많이 발생하는 data structure이다. Rmap의 anonymous page와 file page는 interval trees로 되어 있으며, 이것은 reverse page 성능 향상으로 위해 지속적으로 최적화가 [16] [17]

따라서, fork를 병렬로 수행하면 reverse page mapping의 lock 문에 scalability가 떨어진다. Rmap의 anonymous page를 위한 rmap과 file mapped page을 위한 rmap 모두 문제가 있다.

이번장은 LDU가 어떻게 리눅스에 적용되었는지에 대한 practial한 내용에 대해서 설명한다.

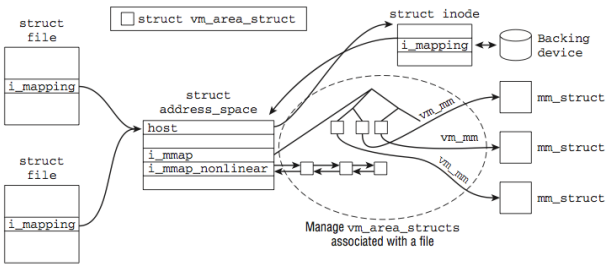

Figure 4-7: Tracking the virtual address spaces into which a given interval of a file is mapped with the help of a priority tree.

Figure 8: An example of applying the LDU to file reverse mapping.

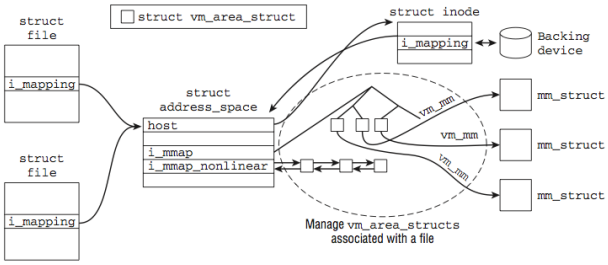

Figure 4-7: Tracking the virtual address spaces into which a given interval of a file is mapped with the help of a priority tree.

Figure 9: An example of applying the LDU to file reverse mapping.

4.2 anonymous page

Anonymous rmap의 공유데이터는 서로 상당히 복잡하게 연결되어 있다.
GLDU. Anonymous rmap을 위한 GLDU는 anon_vma의 root structure에서 log를 저장하도록 하였다. anon rmap은
PLDU.

Anonymous page에는 per-core 방식을 적용하기 힘들다. 그 이유는 log를 저장하는 object가 너무 많은 object를 만들어서 global per-core hash table의 충돌이 너무 많이 발생하기 때문이다. 이것은 또 다른 lock이 필요하므로 오히려 성능이 떨어진다. 만약 per-core 방식을 적용하려면 하나의 per-core memory에 저장한 후 차후 log를 ojbect별 분리 하는 방법이 있으나, 이 경우는 mix되어 있는 로그를 구별하면서 해당 data structure만 보호하는 lock을 사용하기가 어려우므로, global lock을 사용해야한다. 결국 이 방법도 역시 global lock에 의한 scalability 문제가 있다.

4.3 file mapped page

file rmap의 공유데이터는 anonymous rmap보다는 덜 복잡하게 연결되어 있다. 그림 x는 file rmap의 공유데이터를 보여준다.
GLDU. file mapping은
PLDU. file mapping은 PLDU를 적용하는데 문제가 없다. 그 이유는 상대적으로 log의 header를 가지고 있는 address_space가 상대적으로 적게 생성이 되어 hash 충돌이 적게 나타나기 때문에 scalability가 떨어지지 않는다.

5 Implementation

We implemented the new deferred update algorithm in Linux 3.19.rc4 kernel, and our modified Linux is available as open source. LDU's scheme is based on deferred processing, so it needs a garbage collector for delayed free.

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.

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 [23];therefore, we implement the Harris The code refers from sysnchrobench [22]

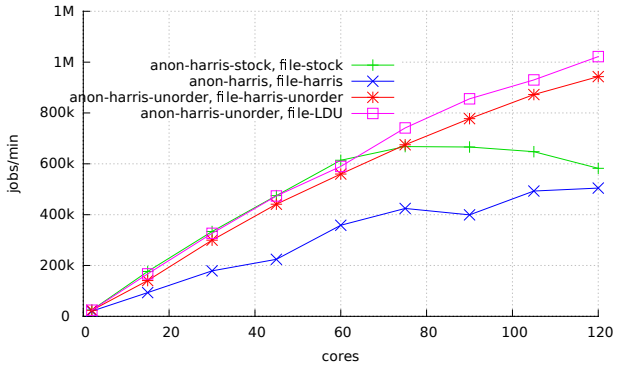

Figure 10: Scalability of AIM7 multiuser for different method. The combination LDU with unordered harris list scale well;in contrast, up to 60 core, the stock Linux scale linearly, then it flattens out.

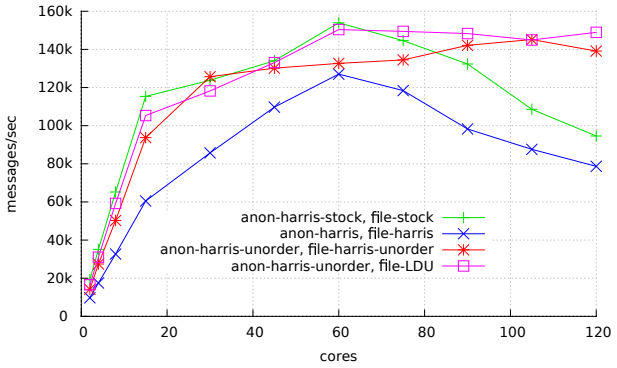

Figure 11: Scalability of Exim. The stock Linux collapses after 60 core;in contrast, both unordered harris list and our LDU flatten out.

and ASCYLIB [19], 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 [39] 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 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 [32]. 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 10, 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 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.

6.3 Exim

To measure the performance of Exim, shown in Figure 11, 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 [8]. Therefore, although we eliminate the fork problem, the Exim may suffer from contention on spool directories.

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.

Results shown in Figure 11 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 [29]; we used 100 processes.

The results for lmbench are shown in Figure 12, 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`.

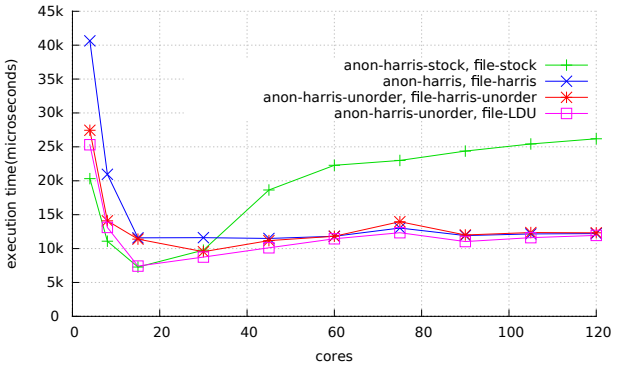

Figure 12: Execution time of lmbench's fork micro benchmark. The fork micro benchmark drops down for all methods up to 15 core but either flattens out or goes up slightly after that. At 15 core, the stock Linux goes up;the others flattens out

7 Discussion and future work

우리는 오직 high update rate를 가진 data structure를 대상으로 LDU를 구현되었다. 하지만 실제 high update rate를 가진 data structure는 워크로드 마다 상황이 틀리다. 앞에서 설명한 두가지 reverse mapping도 역시 fork intencive 워크로드이여야지 high update rate를 가진 data structure라고 말할 수 있다. 결국 log-based approach가 low update rate를 가질때는 오히려 성능이 떨어질 수 있다. 그러므로 log-based approach인 LDU를 high update rate을 가질 때만 적용할 수 있도록 해야 한다. 본 논문은 이 부분에 대해서는 고려하지 않았다. 따라서 임베디드 시스템등 다양한 분야에 사용되는 리눅스 커널에 적용되기 위해서는 update rate를 판단하여 동작하도록 해야지 보다 범용적으로 high update rate 상황과 general학

현재 LDU는 data structure를 보호하기 위해 exclucive lock를 사용하도록 구현하였다. 하지만 concurrent read를 위해 concurrent update와 concurrent read를 고려해서 구현해야 한다. 이 방법은

8 Related work

Operating system scalability. [14]In order to improve Linux scalability, researchers have been optimized memory management in Linux by finding and fixing scalability bottlenecks [7] [9]. 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 [12] solved this address space problem by using the RCU; RadixVM [13] 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 [8].

Scalabe data structure. [20]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 [5] 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 [25] 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. [37] [11] [10]One method for the concurrent update is using the non-blocking algorithms [23] [21] [36], which are based on CAS.

MCS [30], a scalable exclusive lock, is used in the Linux kernel [15]. 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 [18] 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 [28] 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.

References

[1] Aim benchmarks. http://sourceforge.net/projects/aimbench.

[2] Mosbench. 2010. https://pdos.csail.mit.edu/mosbench/mosbench.git.

[3] Exim internet mailer. 2015. http://www.exim.org/.

[4] Tim Chen Andi Kleen. Scaling problems in fork. In *Linux Plumbers Conference, September*, 2011.

[5] 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.

[6] Silas Boyd-Wickizer. Optimizing communications bottlenecks in multiprocessor operating systems kernels. In *PhD thesis, Massachusetts Institute of Technology*, 2013.

[7] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[8] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *9th USENIX Symposium on Operating System Design and Implementation*, pages 1–16, Vancouver, BC, Canada, October 2010.

[9] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *In Proceedings of the Linux Symposium*, 2012.

[10] 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.

[11] Davidlohr Bueso. Scalability techniques for practical synchronization primitives. *Commun. ACM*, 58(1):66–74, December 2014.

[12] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, pages 199–210, London, UK, February 2012.

[13] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013.

[14] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.*, 32(4):10:1–10:47, January 2015.

[15] J. Corbet. *MCS locks and qspinlocks*. 2014.

[16] Jonathan Corbet. The object-based reverse-mapping vm. 2003. https://lwn.net/Articles/23732/.

[17] Jonathan Corbet. The case of the overly anonymous anon_vma. 2010. https://lwn.net/Articles/383162/.

[18] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[19] Tudor David, Rachid Guerraoui, and Vasileios 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*, ASPLOS '15, pages 631–644, New York, NY, USA, 2015.

[20] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 233–246, New York, NY, USA, 2015. ACM.

[21] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, 2004.

[22] Vincent 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*, PPoPP 2015, pages 1–10, New York, NY, USA, 2015.

[23] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001.

[24] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[25] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 168–183, New York, NY, USA, 2015.

[26] Paul McKenney. Msome more details on read-log-update. 2016. https://lwn.net/Articles/667720/.

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

[28] 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, Las Vegas, NV, October 1998.

[29] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[30] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third PPOPP*, pages 106–113, Williamsburg, VA, April 1991.

[31] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, Denver, CO, June 2016. USENIX Association.

[32] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *DISC '13 Proceedings of the 27th International Conference on Distributed Computing, Jerusalem, Israel*. 2013.

[33] 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.

[34] Ori Shalev and Nir Shavit. Predictive log-synchronization. *SIGOPS Oper. Syst. Rev.*, 40(4):305–315, April 2006.

[35] Dave Hansen Tim Chen, Andi Kleen. Linux scalability issues. In *Linux Plumbers Conference, September*, 2013.

[36] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 309–310, New York, NY, USA, 2012.

[37] Tianzheng Wang, Milind Chabbi, and Hideaki Kimura. Be my guest: Mcs lock now welcomes guests. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 21:1–21:12, New York, NY, USA, 2016. ACM.

[38] Huang Ying. Lock-less list. 2011. https://lwn.net/Articles/423366/.

[39] Kunlong Zhang, Yujiao Zhao, Yajun Yang, Yujie Liu, and Michael Spear. Practical non-blocking unordered lists. In *DISC '13 Proceedings of the 27th International Conference on Distributed Computing, Jerusalem, Israel*. 2013.