차 례

그림 차례

표 차례

제 1 장 서론

1.1 개요

성능에 대한 확장성(Scalability)는 100코어 또는 1000 이상으로 구성된 매니코어(Many-core) 시스템에서 가장 중요한 요소이다. 전체 시스템의 성능 확장성은 일반적으로 운영체제의 커널(Kernel) 때문에 제한을 받는다. 이러한 운영체제 커널 중 가장 많이 사용되는 것은 리눅스(Linux) 커널이다. 그 이유는 리눅스 커널은 멀티코어에 최적화가 되어 있기 때문이다. 하지만 이렇게 멀티코어에 최적화된 리눅스 커널도 매니코어 시스템에서는 여전히 성능에 대한 확장성에 문제가 있다 [9] [36]. 확장성 문제중 가장 큰 문제는 커널의 자료구조 중 업데이트 락(Lock) 경쟁에 대한 문제 때문이다 [32] [29].

이처럼 업데이트 직렬화(Serialization) 문제를 해결하기 위해 여러 동시적 업데이트(Concurrent Update) 방법 들이 연구되고 있다 [7] [29] [20]. 동시적 업데이트 방법을 사용하여 업데이트 직렬화 문제를 해결하는 연구들은 업데이트 비율에 따라 많은 성능 차이를 보인다. 이러하는 방법들은 높은 업데이트 비율을 가진 자료 구조 때문에 발생하는 확장성 문제에 대해서는 여전히 효율적이지 않다. 높은 업데이트 비율을 가진 자료에 대한 해결책 중 하나는 캐쉬 통신 병목(cache communication bottleneck) 현상을 줄인 로그 기반(log-based) 알고리즘 [25] [10]을 사용하는 것이다. 로그 기반 알고리즘은 업데이트가 발생하면, 자료구조의 업데이트 명령(update operatin)을 퍼코어(per-core) 또는 원자적(atomic)하게 로그로 저장하고 읽기 명령(read operation)을 수행하기 전에 저장된 로그를 수행하는것이다. 따라서, 리더(reader)는 최신 데이터를 읽게 되며, 이것은 마치 CoW(Copy On Write)와 유사하다 [31] [37].

S. Boyd-Wickizer et al.는 동기화된 타임스탬프 카운터(synchronized timestamp counters) 기반의 퍼코어 로그를 활용하여 높은 업데이트 비율을 가진 자

료구조를 대상으로 동시적 업데이트 문제를 해결함과 동시에 캐쉬 커뮤니케이션 병목현상(cache communication bottleneck)을 줄였다 [10]. 동기화된 타임스탬프 카운터 기반의 퍼코어 로그를 활용한 동시적 업데이트방법은 업데이트 부분만 고려했을 때, 퍼코어에 데이터를 저장함으로 굉장히 높은 성능 확장성을 가진다[]. 하지만 퍼코어 기반의 동기화된 타임스탬프 카운터를 사용한 방법은 결국 타임스탬프 병합(timestamp merging) 과 정렬(ordering) 작업을 야기한다. 만약 코어 수가 늘어 날 경우, 로그를 자료 구조에 적용하는 과정에서 타임스탬프(timestamp) 때문에 발생하는 추가적인 순차적 프로세싱sequential processing)이 요구된다. 이것은 결국 확장성과 성능을 저해한다.

본 논문은 동기화된 타임스탬프 카운터를 이용함에 따라 생기는 추가적인 순차적 프로세싱(sequential processing) 문제를 해결하기 위해 공유 메모리 시스템을(shared memory system) 위한 새로운 LDU(Lightweitgh log-based Deferred Update)를 개발하였다. LDU는 타임스탬프 카운터가 필요한 명령어 로그(operation log)를 업데이트 순간 지우고, 매번 로그를 생성하지 않고 재활용하는 방법이다. 이로인해 동기화된 타임스탬프 카운터 문제와 캐쉬 커뮤니케이션 병목현상에 대한 문제를 동시에 해결하였다. 해결 방법은 분산 시스템(distributed system)에서 사용하는 동기화된 타임스템프 로그 기반의 동시적 업데이트 방식과 최소한의 공유 메모리 시스템의 하드웨어 기반 동기화(hardware-based synchronization) 기법(compare and swap, test and set, atomic swap)을 조합하여 동시적 업데이트 문제를 해결하였다.

이처럼 동기화된 타임스탬프 카운터를 제거함과 동시에, 캐쉬 커뮤니케이션 병목 현상을 줄인 LDU는 기존 로그 기반 알고리즘들의 장점들을 모두 포함할 뿐만아니라 추가적인 장점을 가진다. 첫째로, 업데이트가 수행하는 시점 즉 로그를 저장하는 순간에는 fine-grain synchronization이 필요가 없다. 따라서 동기화(synchronization) 오버헤드 없이 동시적 업데이트를 수행할 수 있다 둘째로, 저장된 업데이트 명령어 로그를 coarse-grained lock과 함께 하나의 코어에서 수행하기 때문에, 캐쉬 효율성이 높아진다 [25]. 다음으로, 기존 여러 자료구조에 쉽게 적용할 수 있는 장점이 있다. 게다가 마지막으로, 로그를

저장하기 전에 로그를 삭제하므로 보다 빠르게 로그의 수를 줄일 수 있다.

우리는 위와 같은 장점을 가지는 LDU를 리눅스 커널에서 높은 업데이트 비율 때문에 성능 확장성 문제를 야기시키는 익명 역 매핑(anonymous reverse mapping)과 파일 역 매핑(file reverse mapping)에 적용하였다. 또한 우리는 LDU를 리눅스 커널 버전 4.5.rc4에 구현하였고, Fork가 많이 발생하는(fork-intensive) 워크로드인 AIM7 [1], MOSBENCH [4]의 Exim [5], Lmbench [34]를 대상으로 성능 개선을 보였다. 개선은 기존 리눅스 커널에 비해 120코어에서 각각 1.5x, 2.6x, 2.7x 배 성능 향상을 가진다.

1.2 논문의 기여

본 논문은 다음과 같은 기여를 하였다.

- 우리는 높은 업데이트 비율를 가지는 자료구조를 위한 새로운 로그 기반 동시적 업데이트 방법인 LDU를 개발하였다. LDU는 동기화된 타임스탬프 카운터를 이용함에 따라 생기는 시간 정렬과 머징에 의한 추가적인 순차적 프로세싱 문제를 최소한의 하드웨어 동기화 기법을 사용하여 해결한 방법이다. LDU는 하드웨어 동기화 기법을 이용하여 LDU는 로그를 업데이트 순간 지우고 로그를 재활용한다.
- 우리는 LDU을 현실적인(practical)한 매니코어 시스템인 인텔(Intel) 제온(XEON) 120코어 위에 동작하는 리눅스 커널의 2가지 역 매핑 (익명, 파일)에 적용하여, 리눅스 fork 성능 확장성 문제를 해결하였다. Fork 관련 벤치마크 성능은 워크로드 특성에 따라 1.6x부터 2.2x까지 개선되었다.

1.3 논문에서 해결하고자 하는 구체적인 문제

운영체제 커널의 병렬화(paralleism)는 시스템 전체의 병렬화(parallesim)에서 가장 중요하다. 만약에 커널이 확정성이 있지 않으면, 그 위에 동작하는

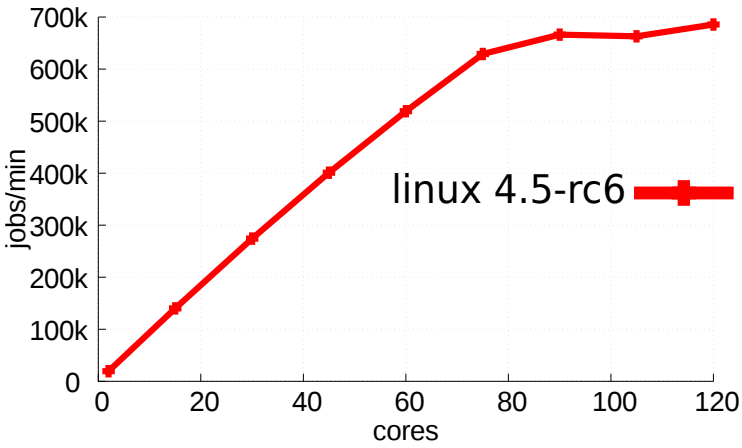


그림 1.1: AIM7-multiuser scalability

응용프로그램들도 역시 확장성이 있지 않는다 [15] [8]. 우리는 이처럼 중요한 부분인 운영체제 커널 중 멀티코어에 최적화된 리눅스의 성능 확장성을 분석하기 위해, AIM7-multiuser를 가지고 성능 확장성을 실험해보았다. AIM7은 최근에도 성능 확장성을 위해 연구(Research) 진영과 리눅스 커널 커뮤니티 진영에서도 활발히 사용되고 있는 벤치마크 중 하나이다 [12] [11]. AIM7-multiuser 워크로드는 동시에 많은 프로세스(Process)를 생성하며 수행되며, 디스크 파일(Disk File) 오퍼레이션, 가상 메모리(Virtual Memory) 오퍼레이션, 파이프(Pipe), I/O(Input/Output) 그리고 수학 연산과 함께 수행한다. 우리는 파일 시스템의 성능 확장성(File system scalability)를 최소화 하기 위해 tempfs(Temp file system)를 사용하였다. 실험 결과 7코어 까지는 확장성이 있으나 그 이후에는 확장성이 떨어져 완만한 그래프를 보여준다.

우리는 성능 확장성에 근본적인 문제를 분석하기 위해, 문제가 있는 120코어에서 리눅스의 Lockstat[]를 이용하여 락 경합을 분석하였다. Lockstat는 리눅스 커널에 있는 락 프로파일러(profiler)이며, 얼마나 쓰레드(Thread)가 락을 보유하고 락을 얻기 위해 쓰레드가 얼마나 기다리고 있지를 보여준다. 먼저 멀티 프로세스 기반의 벤치마크인 AIM7을 동작시키고 동시에 120코어 대해

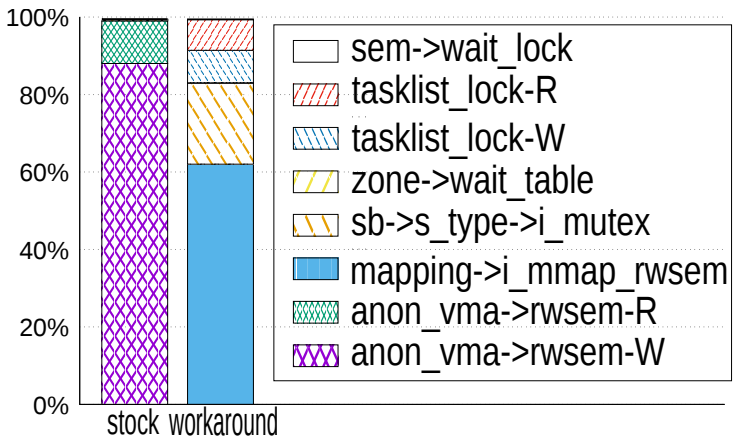


그림 1.2: Lock wait time on 120 core

서 락 경합을 분석하면 그림 1.2(b)과 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 익명 역 매핑 세마포어(semaphore) (`anon_vma->rwsem`) 수많은 fork에 의해 프로세스를 생성하면서 발생하는 락 경합 문제이다. 이러한 역 페이지 매핑(revsers page mapping)은 리눅스가 fork(), exit(), and mmap() 시스템 콜(system call)을 사용할 때 페이지(page) 정보를 업데이트한다. 다음으로 우리는 익명 역 매핑의 락 경합을 줄이기 위해, 임시로 fork에서 익명 역 매핑를 호출하는 부분과 읽기와 관련있는 페이지 스왑(page swap)이 안되도록 하고, 120코어를 대상으로 다시 락 경합을 분석하여 보았다. 이 때 부터 그동안 상대적으로 가려졌던 파일 역 매핑에서 많은 락 경합이 발생되었다. 본 연구의 분석 결과 둘 중 하나가 아니라 두 가지 락 모두 fork의 성능 확장성 문제를 야기 시킨다.

이러한 익명 역 페이지 매핑은 리눅스 커뮤니티에서 잘 알려진 락 경합 문제 [6]이고, 파일 페이지 역 매핑에 대한 락 경합 문제는 S.Boyd-Wickizer [10] OpLog 논문을 통해 fork의 확장성 문제의 중요한 원인으로 제시한 부분이다. 즉 두가지 모두 개선해야지 fork의 성능 확장성이 향상 된다.

이러한 높은 업데이트 비율 때문에 발생하는 업데이트 직렬화 문제에 대한 해결 방법들은 그동안 여러 방법들이 제안되었다. 해결 방법들은 동시적 업데이트를 위한, 논블락킹 자료구조(non-blocking data structure)를 이용하는 방식과 로그 기반(log-based) 알고리즘을 사용하는 방법이 있다. 논블락킹 알고리즘들은 하드웨어 동기화 원자적(hardware synchronized atomic) 연산들을 활용하여 동시적으로 업데이트와 읽기 명령어를 수행하게 만든 자료구조이다. 예를들어, 논블락킹 알고리즘들은 업데이트 명령어 수행전에 전역 변수를 지켜본 후, 업데이트를 원자적인 CAS(Compare And Swap) 명령으로 전역 변수가 변경되었는지 확인과 저장하는 읽을 원자적으로 수행한다. 이때 해당 전역변수가 수정되었다면, 업데이트 명령어는 다시 처음부터 수행하여 다른 쓰레드가 변경을 안할때까지 같은 일을 수행하는 방법이다. 하지만 이러한 방법도 결국 전역 공유 메모리 주소에 다수의 CAS로 접근하여 병목현상이 생긴다. 이것은 결국 캐쉬 커뮤니케이션 오버헤드를 만든다 [10]. 최근에는 전역 공유 메모리 주소에 다수의 CAS(Compare And Swap)로 접근하여 발생하는 병목현상을 줄인 로그 기반 방법들이 연구되고 있다. 우리의 LDU도 이러한 로그 기반 방법을 활용하였다.

로그 기반 알고리즘은 업데이트 비율이 많은 자료구조에 적합한 알고리즘이다. 로그 기반 알고리즘은 락을 피하기 위해 업데이트가 발생하면, 자료구조의 업데이트 명령어(삽입 또는 삭제)를 함수 인자(argument)와 함께 저장하고, 주기적 또는 읽기 명령이 수행하기 전에 applies the updates in all the logs to the data structure, so reader can read up to date data structure. 이러한 로그 기반 방법은 마치 CoW(Copy on Write)와 유사하다. 즉, 읽기 전에 저장된 쌓여있는 로그가 수행됨으로 읽기가 간헐적으로 수행되는 자료구조에 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.

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.

1.4 논문 구성

본 논문의 구성은 다음과 같다.

The rest of this paper is organized as follows. Section 1.3 describes the background and the Linux scalability problem. Section 2.1 describes the design of the LDU algorithm and Section 2.2 explains how to apply the LDU to Linux kernel, and Section 2.3 shows the results of the experimental evaluation. Section **??** describes related work with our research. Finally, section 4.1 concludes the paper.

제 2 장 로그기반 동시적 업데이트 방법

2.1 설계

LDU는 리눅스 커널의 높은 업데이트 비율을 가진 자료구조의 성능 확장성 문제를 해결하기 위한 로그 기반 방법 중에 하나이다. 동기화된 타임스탬프 카운터 기반의 퍼코어 로그를 활용한 동시적 업데이트방법은 결국 타임스탬프 정렬 및 머징 작업을 야기한다. 특히 코어 수가 늘어 날 경우, 퍼코어 로그를 자료 구조에 적용하는 과정에서 추가적인 순차적 프로세싱이 요구된다. 이것은 확장성과 성능을 저해한다. 이러한 문제를 해결하기 위해, LDU는 로그기반 방식의 동시적 업데이트 방법과 원자적 동기화 기능을 최소한으로 이용하도록 설계하였다. 따라서, LDU는 동기화된 타임스탬프 카운터를 사용하는 방식의 타임스탬프를 제거함과 동시에 캐쉬 커뮤니케이션 오버헤드를 최소화 하였다. 이번 장에서는 LDU의 알고리즘적인 디자인 측면에 대해서 설명한다.

2.1.1 Approach

동기화된 타임 스탬프 카운터가 필요한 근본적인 이유는 특정한 명령어들은 반드시 순서가 지켜져야 한다. 예를 들어 프로세스가 삽입 명령어를 퍼코어 메모리에 저장한 후 그 프로세스가 다른 코어로 이동했을 때, 만약 다음 오퍼레이션이 삭제 오퍼레이션이면 반드시 앞의 삽입 명령어 다음에 수행되어야 한다 [10]. 이러한 시간에 민감한 로그를 더 구체적으로 설명하기 위해 우리는 논문 [15]에서 사용한 심볼 방식으로 설명한다. 우리는 삽입 명령은 원형으로된 플러스 모양인 ⊕로 표시하고, 삭제 명령은 원형으로된 마이너스 모양인 ⊖으로 표시하고 오브젝트들은 색(Ⓑ(object B))으로 구별하였다. 서로 다른 색깔과 다른 높낮이는 다른 CPU를 의미한다. 예를 들어

⊕Ⓐ, ⊕Ⓑ, ⊕Ⓒ,⊖Ⓐ, ⊖Ⓒ, ⊕Ⓐ, ⊕Ⓒ,⊖Ⓒ

이것은 5개의 삽입 명령들과 3개의 삭제 명령 그리고 3개의 CPU들 그리고 3개의 오브젝트를 의미한다. 이 예제에서 ⊕A와 ⊖A는 시간에 민감한 로그이며 반드시 시간 순서로 실행되어야 한다. LDU는 이러한 시간에 민감한 명령어들을 업데이트 순간에 삭제한다. 그렇게 함으로써 동기화된 타임 스탬프 카운터는 제거가 된다. 한가지 더 중요한 사실은 이러한 시간에 민감한 로그들은 최적화 단계에서 제거가 된다는 것이다. 예를 들어, 삽입-삭제 명령어 또는 삭제-삽입 명령어 경우, ⊕A⊖A, ⊕C⊖C 그리고 ⊕C⊖C들은 리더가 수행하기 전에 취소되어도 상관없는 명령어들이다. 따라서 남은 로그인

⊕B, ⊕A

로그들은 시간에 민감하지 않는 로그들이다. 업데이트 명령어가 발생되면 LDU는 이러한 타임 민감한 명령어를 업데이트 측면에서 제거하는 방법을 사용하여 제거한다.

LDU는 시간에 민감한 로그를 제거하기 위해 업데이트 측면에서의 삭제(update-side removing)이라는 방법을 사용한다. 이 방법은 만약 같은 오브젝트(object)에 대해서 삽입(insert)와 삭제(remove)가 발생하였으면, 같은 오브젝트에 대해서, 삽입 명령과 삭제 명령에 대한 로그를 업데이트 시점에 바로 바로 삭제하는 방법이다. 동기화된 타임 스탬프 카운터(syncronized timestamp counters) 기반의 OpLog도 이러한 로그 삭제 방법 수행하여 최적화를 하였으나, 명령어에 대한 로그가 서로 다른 코어에 존재하는 로그 같은 경우에 로그를 반드시 병합한 후 삭제를 해야한다. 동기화된 타임 스탬프 카운터 방법은 워크로드에 따라, 최적화를 위해 또 다른 순차적 프로세싱이 요구된다. 하지만 LDU는 개별적 오브젝트(indivisual object)를 대상으로 스왑(swap) 원자적 명령을 사용하여 공유된 로그를 삭제하는 방법을 사용해서 이런 문제가 없다.

업데이트 순간 로그를 지우는 방법은 공유 메모리 시스템의 스왑(swap) 명령어를 사용한다. 이를 위해, LDU는 모든 오브젝트에 삽입과 삭제의 마크(mark) 필드를 추가해서 업데이트 시점에 로그를 삭제하였다. 예를 들어 만약 같은 오브텍트에 삽입-삭제(insert-remove) 명령이 수행될 경우 처음 삽입 명령어는 삽입에 대한 마크 필드에 마킹하고 큐(queue)에 저장한다. 다음 삭제

명령 부터는 로그를 큐에 저장하지 않고 삽입에 표시한 마크 필드에 표시한 값만 원자적으로 지워주는 방식으로 진행된다. 다음으로 LDU는 로그를 적용할 때, 큐안에 로그가 존재하더라도, 마크 필드가 표시된 로그만 실제로 실행한다. 이것은 스왑이라는 상대적으로 가벼운 연산과 상대적으로 덜 공유하는 개별적인 공유 오브젝트의 마크 필드(mark filed)를 사용해서 시간에 민감한 명령을 제거할 뿐만 아니라, 동시에 실제 명령을 수행하지 않고 로그를 지워주는 효과를 가져주므로 성능이 향상된다.

LDU의 또 다른 최적화 기법은 업데이터 시점에 지우는 기능 때문에 취소된 가비지(garbage) 로그를 재활용하는 것이다. 이러한 가비지 로그일 경우 다음 업데이트 명령에 대해서는 해당 로그를 새로 많들어 넣지 않고 기존 로그를 재활용하는 방법이다. 예를 들어 같은 오브젝트에 대해서 삽입-삭제-삽입(insert-remove-insert) 순서로 업데이트가 수행될 경우, 세번째 삽입 명령어는 큐에 들어가 있지만 업데이트 시점에 지워진 로그이기 때문에 최소되어 삽입에 대한 마크 필드가 FALSE로 표지된다. 이런 경우가 가비지 오비젝트(garbage object)이다. 이러한 경우, 다음 삽입 명령에 대해서는 새로운 로그를 큐에 다시 넣지 않고, 해당 오브젝트의 마크 필드만 변경하여 로그를 재활용하는 방법을 사용한다.

LDU는 로그 때문에 불필요하게 메모리 낭비를 방지하고, 끝없이 증가되는 것을 방지하기 위해 로그를 주기적으로 시간 순서대로 적용한다. 따라서 LDU는 주기적으로 로그를 적용함으로 로그가 쌓여서 발생하는 메모리 낭비를 줄인다. 이것은 OpLog의 배치 업데이트(batching updates)와 Flat combining의 병합 쓰레드(combiner thread)와 같이, 1개의 쓰레드가 업데이트 명령을 수행하므로 캐쉬 지역성(cache locality)이 높아지는 장점을 가진다.

LDU는 보다 다양한 데이터 구조를 지원하기 위해, 퍼코어또는 전역 큐(global queue) 모두 이용할 수 있게 설계하였다. 그 이유는 데이터 구조가 특성에 따라 퍼코어 구조에 적당한 자료구조가 있고, 전역 큐 구조에 적당한 데이터 구조가 있기 때문이다. 먼저 LDU의 퍼코어 큐는 큐에 로그를 저장할 때, 글로벌 헤드(global head) 포인터에 대한 CAS 명령어를 완전히 제거한 장점을

가진다. LDU의 퍼코어 큐의 대한 단점으로는 시간에 민감한 로그가 제거되었더라도, 앞에서 설명한 예와 같이 남은 로그들인 ⊕B, ⊕A는 순서가 바뀔 수 있다. 이러한 경우 트리와 같은 자료구조에는 문제가 없으나, 스택과 같이 남은 명령의 순서도 민감한 자료구조는 사용하지 못하는 문제점이 있다. 또한 전형적인 퍼코어 큐를 방법의 단점으로 메모리 관리 코드에 대한 복잡도가 증가되고, 메모리 사용량도 퍼코어에 추가적으로 할당을 받으므로 증가된다. 이를 보완하기 위해 LDU는 전역 큐도 같이 지원한다. LDU의 전역 큐의 장점은 광장히 간단하여 어떻한 자료구조라도 쉽게 적용이 가능하다. 또한 퍼코어 처럼 글로벌 헤드 포인터에 대한 CAS연산을 완전히 제거하지 못하지만, LDU의 업데이트 시점에 지우는 방법과 가비지 로그를 재활용하는 방법으로 글로벌 큐에 로그를 저장하는 횟수를 상당히 줄여서, 캐쉬 커뮤니케이션 오버헤드를 상당히 줄일 수 있다.

LDU는 논블락킹 큐를 이용하여 로깅한다. 그 이유는 전역 또는 퍼코어 락 없이 수행될 수 있기 때문이다. 논블락킹 큐 중에 LDU는 헤드 포인터에 대한 CAS 연산을 최대한 줄인 multiple producers and single consumer에 활용될 수 있는 자료구조[]를 이용하였다. 이 큐는 다른 논블락킹 리스트들[][][]과 다르게, 삽입 명령을 항상 처음 노드에 삽입함에 따라, CAS가 발생되는 횟수를 상대적으로 줄일 수 있다. 게다가 로그를 적용하는 부분에서 단일 소비자(single consumer)만 고려 했기 때문에, 삭제를 위한 복잡한 알고리즘이 필요 없다. 예를 들어 단일 소비자(single consumer)는 명령 로그 전체를 얻기 위해, 스왑 명령을 사용하여 헤드 포인터(head pointer)를 널(NULL)로 원자적으로 제거한다.

2.1.2 Example

그림 2.1는 LDU가 퍼코어 큐와 전역 큐를 사용하여 수행되는 예를 보여준다. 우리는 높은 업데이트 비율을 가지는 자료구조와 함께 동시적 지연 업데이트 방법을 설명하기 위해서, 7개의 업데이트 명령어가 리드 명령 전에 어떻게 동시적으로 수행되는지를 설명한다. 업데이트 명령어의 순서는 아래와 같다.

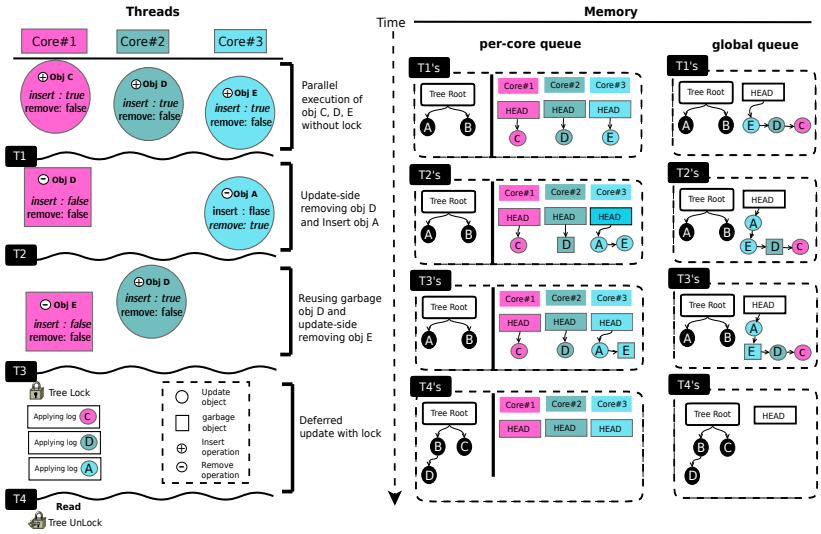


그림 2.1: The LDU example showing seven update operations and one read operation.

$\oplus C, \oplus D, \oplus E, \ominus D, \ominus A, \oplus D, \ominus E.$

. 우리는 LDU를 설명하기 위해서 앞절에서 사용한 심볼에 사각형 모양의 가비지 심볼인 D를 추가하여 설명한다.

이 그림에서 실행 순서는 위에서 부터 아래로 이루어진다. 그림 왼쪽에 있는 것은 CPU의 명령어들을 보여주고 오른쪽에 있는 것은 특정한 시간에 메모리에 있는 자료구조의 내용에 대해서 보여준다. 초기 트리 자료구조에는 오브젝트 A와 B가 들어 있고 큐는 비어 있다. 그림의 위쪽에는 `Core1`, `Core2` 그리고 `Core3`은 동시적 업데이트 명령을 수행한다. 따라서 $\oplus C$, $\oplus D$ 그리고 $\oplus E$는 락이 없이 동시적으로 수행된다.

LDU는 명령어 로그를 저장하기 위해 논블락킹 큐를 사용하기 때문에, 이 작업에는 업데이트 락이 필요없다. 따라서 모든 쓰레드는 락에 대한 경쟁 없이 동시적으로 수행이 가능하다. `T1` 시점이 되면, 트리는 오브젝트 A과 B이

존재한다. 그리고 퍼코어 큐와 전역 큐는 오브젝트 ⊕C, ⊕D 그리고 ⊕E가 보관되고, 퍼코어 큐는 퍼코어 메모리에 각각 구별되어 저장된다.

다음 명령어들은 ⊖D과 ⊖A이며, 먼저 ⊖D 명령어가 실행이 되면, LDU는 새롭게 큐에 로그를 넣지 않고, 원자적으로 오프젝트에 있는 마크 필드를 수정한다. 그리고 ⊖A 명령어는 새로운 명령어이기 때문에 큐에 바로 저장한다. T2 시점이 되면, 퍼코어 큐와 전역 큐에는 ⊕C, ⊕D, ⊕E 그리고 ⊖A 로그들이 저장된다. 이 시점에서는 오브젝트 D의 삽입에 대한 마크 필드는 FALSE이다. 이것은 업데이트 시점에 삭제되는 방법 때문에 취소된 로그이며, 우리는 가비지 오브젝트라 부른다.

마지막 명령어들은 ⊕D, ⊖E 이다. LDU는 원자적 스왑을 사용하여 큐에 있는 로그를 새롭게 생성하지 않고 다시 재활용한다. 그리므로, 업데이트 측면에서 지우는 기술로 오브젝트 D는 D로 바뀌고, 그리고 오브젝트 E는 E로 바뀐다. T3 시점이 되면, 퍼코어 큐에는 ⊕C, ⊕D, ⊖A 그리고 ⊕E가 저장된다. 리드 함수가 수행되기 전에 이것은 트리의 명령어를 보호하기 위해서 상호배재 기반의 원본 트리의 락이 필요하다. LDU는 큐에서 트리로 각각의 명령어를 옮긴다. 이 때 옮길 오브젝트는 마크가 표시된 오브젝트 로그들이다. 그러므로, 명령어 ⊕C, ⊕D 그리고 ⊖A 들은 가비지 로그인 ⊕E를 제외하고 옮겨진다. T5시점이 되면, 트리는 B, C 그리고 D를 가지게 되어, 최종적으로 리더는 결국 일관성있는 같은 데이터를 읽게 된다.

2.1.3 The Algorithm and Correctness

이번 장은 알고리즘의 중요한 부분에 대해서 보여준다. 우리는 LDU의 로깅하는 큐에 대한 알고리즘과 자세한 자료구조에 대해서는 간략한 설명을 위해 배재하였다.

```
1  bool ldu_logical_insert(struct object_struct *obj, void *head) {
2    // Phase 1 : update-side removing logs
3    if (SWAP(&obj->ldu.remove.mark, false) == false){
4      ASSERT(obj->ldu.insert.mark);
5      obj->ldu.insert.mark = true;
6      // Phase 2 : reusing garbage log
7      if (!TEST_AND_SET_BIT(LDU_INSERT,
8          &obj->ldu.used)){
9        // Phase 3(slow-path): insert log to queue
10       // ... : save argument and operation
11       ldu_insert_queue(head, log);
12     }
13   }
14 }
```

그림 2.2: The LDU concurrent insert algorithm.

inserting logs

그림 2.3는 동시적 업데이트를 수행하는 함수에 대해 보여준다. 이러한 동시적 업데이트 함수는 3가지 단계로 구분된다. 첫째 단계는 체크단계이며, 이 오브젝트가 취소 가능한 오브젝트인지 확인을 한다(Line 4, 20). 이코드가 수행되면, synchronize 함수는 리더 또는 주기적인 함수에 의해 호출된다. 따라서 첫번째 단계에서는 원자적 명령어가 필요하다. 만약 그에 상응하는 마크 필드가 TRUE라면 그것에 대한 마크 필드는 FALSE로 수정된다. 두번째 단계에서는 로그가 이미 큐에 들어가 있는 로그인지 아닌지 체크를 수행한다(Line 8, 24). 만약 그렇다면, 마크 필드는 이미 TRUE로 마크가 됐기 때문에(Line 6, 22), 이 함수는 바로 종료한다. 마지막 단계에서는, 만약 명령어 로그가 처음 사용된 로그라면(Line 12, 28) 명령어 로그는 논블락킹 큐에 저장된다.

이 함수는 항상 옳게 동작한다. 그 이유는 리눅스 커널은 독특한 명령어 순서를 가지고 있기 때문이다. 예를 들어, 만약 같은 오브젝트에 대해서 삽입

```
1  bool ldu_logical_remove(struct object_struct *obj, void *head) {
2    // Phase 1 : update-side removing logs
3    if (SWAP(&obj->ldu.insert.mark, false) == false){
4      ASSERT(obj->ldu.remove.mark);
5      obj->ldu.remove.mark = true;
6      // Phase 2 : reusing garbage log
7      if (!TEST_AND_SET_BIT(LDU_REMOVE,
8          &obj->ldu.used)){
9        // Phase 3(slow-path): insert log to queue
10       //... : save argument and operation
11       ldu_insert_queue(head, log);
12     }
13   }
14 }
```

그림 2.3: The LDU concurrent remove algorithm.

명령이 발생하면, 다음 명령은 반드시 삭제 명령이 발생한다. 왜냐하면, 리눅스 업데이트 함수는 검색(search), 동적할당(alloc) 그리고 해재(free) 함들들과 구별되어 있기 때문이다. 리눅스 커널의 삭제-삭제 순서 또는 삽입-삽입 명령 순서는 금지된다. 만약에 삭제-삭제 명령어 순서가 발생하면, 두번째 삭제 명령어는 크레쉬(crash) 만날 수 있다. 그 이유는 첫번째 삭제 명령어 다음에 이 오브젝트는 바로 동시에 메모리 해제가 될 수 있기 때문다. 따라서 우리는 그에 상응하는 마크 필드를 체크하였다(Line 5, 21).

applying logs

그림 2.4는 명령어 로그들을 적용하는 지연 업데이트 함수를 보여준다. `synchronize` 함수는 리드 전에 호출되거나 주기적으로 호출되는 타이머 핸들러(timer handler)에 의해 호출 된다. 그 이유는 무한정 커지는 로그를 방지하기 위해서이다. `synchronize` 함수가 수행하기 전에, `synchronize` 함

```
1  void synchronize_ldu(void *head)
2  {
3    entry = SWAP(&head->first, NULL);
4    //iteration all logs
5    for_each_all_logs(log, entry, next) {
6      //... : get log's arguments
7      //atomic swap due to update-side removing
8      if (SWAP(&log->mark, false) == true)
9        ldu_apply_log(log->op_num, log->args);
10     CLEAR_BIT(log->op_num, &obj->ldu.used);
11     // once again check due to reusing garbage logs
12     if (SWAP(&log->mark, false) == true)
13       ldu_apply_log(log->op_num, log->args);
14   }
15 }
```

그림 2.4: The LDU applying logs algorithm.

수는 오브젝트의 락을 사용하여 반드시 락이 걸려 있어야 한다. 따라서 이 함수는 단일 소비자로 수행되어야 한다. 즉 이 방법은 OpLog의 배치 업데이트(batching updates)와 FC의 컴파이너 쓰레드(combiner thread)와 비슷하다고 볼수 있다. 처음으로 synchronize 함수는 큐의 헤드 포인터를 원자적인 스왑 명령(Line 3)을 이용하여 얻는다. LDU는 주기적으로 로그의 큐를 적용하기 때문에, LDU의 업데이트 명령어는 synchronize 함수와 동시에 수행될 수 있다. 그러므로 원본 자료구조에 적용하기 전에 마크필드는 FALSE로 수정된다(Line 8,9). 가비지 로그를 위한 사용 플래그(used flag)는 FALSE로 수정된다. 이것은 이 오브젝트가 큐에 포함되어 있지 않는다는 것을 의미한다(Line 10). synchronize 함수는 한번 더 마크 필드가 적용하는 과정(Line 8)과 가비지 비트를 초기화 하는 과정(Line 10)에서 수정되었는지 다시 체크를 한다(Line 12,13).

2.2 Applying Linux kernel

This section shows how to apply the LDU to the complex Linux virtual memory system to solve the update serialization problem;it deals with more practical one.

The Linux reverse page mapping(rmap), a kernel memory management mechanism, consists of anonymous rmap and file rmap, are a update-heavy data structure. These two rmaps maintain virtual address(VMAs) to translate physical addresses to virtual address [30], and the rmaps are a shared global resource between processes. These global resource of rmap are managed by using a interval tree. To protect these shared tree, Linux kernel uses the reader-writer semaphore, and simultaneous creation of many processes becomes bottlenecks because not only the rmap's update operations can not run in parallel but also update's lock brings about the cache invalidation traffic. On the contrary, the rmap rarely reads the interval tree when it swaps a physical page out to disk, migrates other cpu, or truncates a file.

2.2.1 Anonymous mapping

Figure 2.5 shows the anonymous rmap data structure. When a process spawns, the parent's anonymous vma chain(AVC) are copied to a child, and then a new anonymous vma(`struct anon_vma`) is created. When a process simultaneously spawns, the more complex anonymous rmap data structures are created;the anonymous ramp is one of the complex data structure in Linux kernel [17]. The anonymous rmap uses the root lock since the AVCs are shared with child processes, so this root lock causes a lock contention problem [6].

To eliminate this lock contention problem, we add the insert and remove mark field in the individual object(`struct anon_vma`), and then we implement the update-side removing logs scheme. Understanding the log's position of

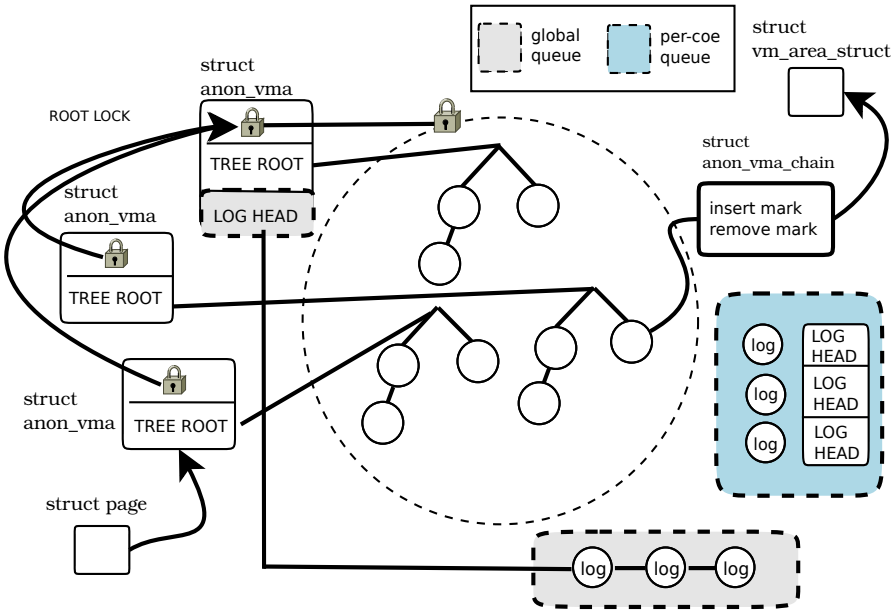


그림 2.5: An example of applying the LDU to anonymous reverse mapping.

queue header is important. As noted earlier, since the anonymous rmap uses the root lock, the per-core queue version of the LDU logs into a per-core memory with root information, or the global queue logs into a root data structure(`struct anon_vma`). Consequently, the LDU does not largely modify the original data structure, which shows why the LDU is a lightweight method.

2.2.2 File mapping

Figure 2.6 shows the rmap for file. In order to translate physical addresses to virtual address, the page(`struct page`) indicates the address space object(`struct address_space`), and the address space object manages the VMAs by using the interval tree. This interval tree is a shared resource between processes. Because

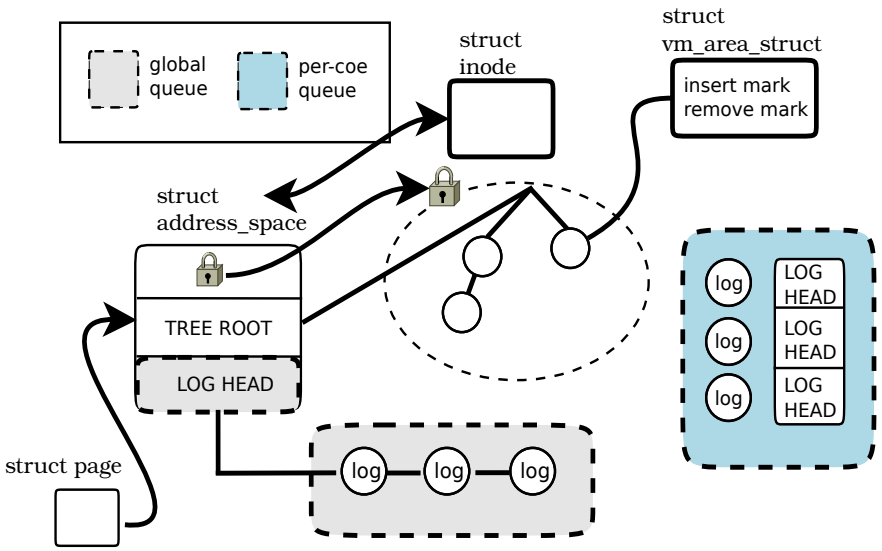


그림 2.6: An example of applying the LDU to file reverse mapping.

the system calls such as `fork()`, `exit()` and `mmap()` entail concurrent updating VMAs into the shared resource. when the processes simultaneously invoke these system calls, the file rmap can be serialized at the update operations.

The LDU can easily be applied to the file ramp data structure. For instance, to use the LDU, a developer adds log's queue header into the per-core memory or into the original data structure(`struct address_space`), and then adds mark field to the individual object(`struct vm_area_struct`). Then, the developer modifies update function to logging function without a lock. Finally, the developer creates `synchronize` function and calls the `synchronize` function before the read.

This figure clearly shows why the LDU additionally supports the global

queue because it is a simpler and easier scheme because the log's head pointer is located in the interval tree's data structure. On the other hand, the per-core queue may need an additional per-core queue management scheme due to the its isolated memory location.

2.2.3 Detail Implementation

Because the log's head pointer for the per-core queue is separated with the original data structure, the implementation of per-core queue uses a per-core hash table method that can allow each object to distinguish. The per-core hash table implemented as a direct-mapped cache, which one bucket only has an object because recently used objects will be in the hash table in a way similar to the OpLog's per-core hash table. When this hash table is met a hash conflict, the LDU evicts the object in the hash slot. Moreover, this method reduces additional tasks of programmers because it can minimize code modifications and does not need an additional lock. The per-core hash table, however, incurs a hash conflict overhead. This method is useful when a number of the root objects are infrequently created like the file rmap(`struct address_space`). On the other hand, since the anonymous rmap severely creates many root objects(`struct anon_vma`), it causes a hash conflict overhead. Therefore, in the case of the anonymous ramp, we did not distinguish object headers, but it needs additional tasks with global lock.

We have implemented the new deferred update algorithm in Linux 4.5-rc6 kernel, and our modified Linux is available as open source. The implementation is stable enough and has passed the testing related with virtual memory, scheduler, and file in the Linux Test Project [2].

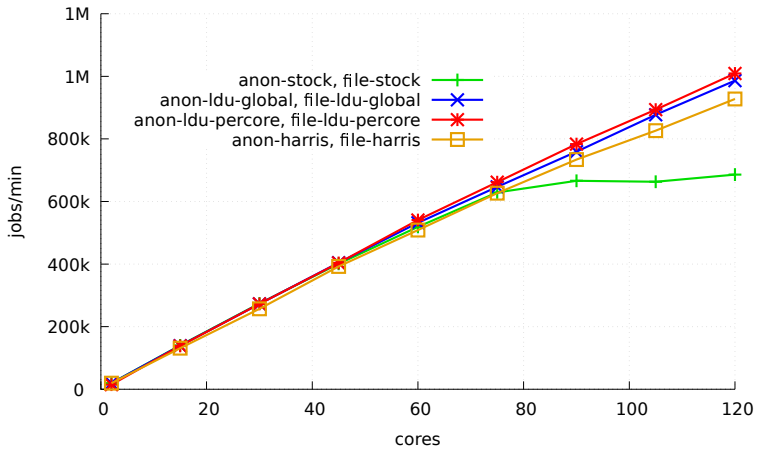


그림 2.7: Scalability of AIM7-multiuser.

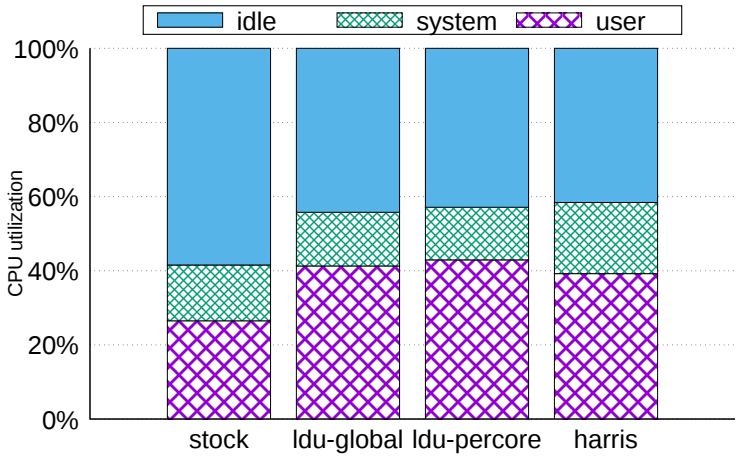
2.3 Evaluation

2.3.1 Experimental setup

For the purpose of performance evaluation of the proposed LDU technique, we performed experiments using a Linux kernel where LDU technique is implemented compared to a lock-free list version of Linux proposed by Harris [24]. The reason why we used Harris algorithm for our comparison purposes is that the algorithm is considered as the representative concurrent non-blocking algorithm. The basic algorithms of Harris linked list are from sysnchrobench [23] and ASCYLIB [19], and we slightly converted the Harris linked list to be adopted in Linux kernels.

The hardware specification we used for our experiments are a 120 core machine with 8-socket, Intel E7-8870 chips(15 cores per socket) equipped with 792 GB DDR3 DRAM.

We selected benchmark programs with fork-intensive applications since

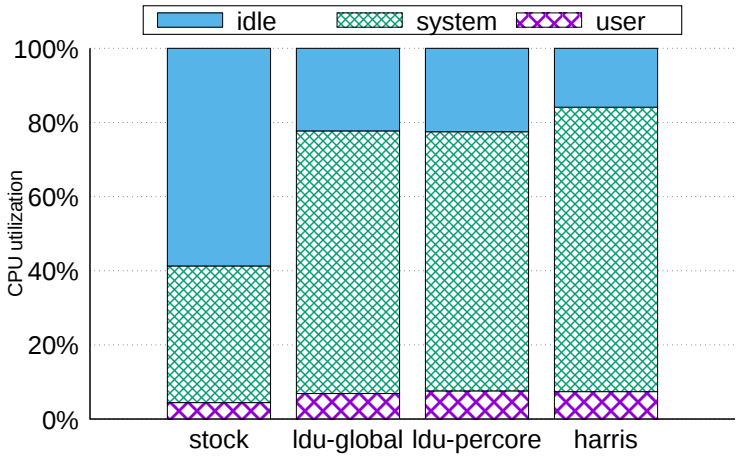


(a) AIM7 - 120core

그림 2.8: AIM7 CPU utilization on 120 core.

the fork-intensive update-heavy data structure accesses could maximally benefit from the proposed technique. The benchmark programs are AIM7, a Linux scalability benchmark, Exim, an email server in MOSBENCH, and Lmbench, a micro benchmark. The workloads exhibit the high lock contentions because of the two reverse mappings. Moreover, the AIM7 benchmark is widely used in the Linux community not only for testing Linux kernel but also for improving the scalability. The Exim is a real world application, but it has scalability bottlenecks caused by the Linux fork. Finally, in order to only focus on the fork performance and scalability, we selected the Lmbench.

We used four different experiment settings. First, we used the stock Linux as the baseline reference. Second, we used the LDU version of Linux kernel that used global queue. Next, we used the per-core queue version of the LDU. Finally, we used Harris lock-free list version of Linux kernel as we mentioned earlier. Unfortunately, direct comparison experiments between the LDU and the



(a) Exim - 120core

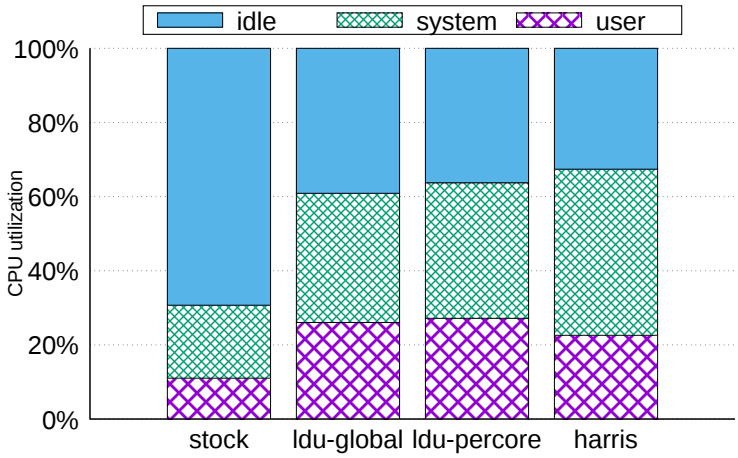
그림 2.9: EXIM CPU utilization on 120 core.

OpLog was not possible for a few implementation-related issues (e.g., we could not obtain the detailed implementation of the OpLog).

2.3.2 AIM7

We used AIM7-multiuser, which is one of fork-intensive workload in AIM7. The multiuser workload simultaneously creates many processes with various operations(see section 1.3), and we used the temp filesystem to minimize the file system bottleneck. We increased a number of users in proportion to number of cores.

The results for AIM7-multiuser are shown in Figure 2.7. Up to 75 core, the stock Linux scales linearly and then serialized updates become bottlenecks. However, up to 120 core, the Harris and our LDU scale well because these workloads can concurrently execute update operations without the reader-writer semaphores(`anon_vma->rwsem`, `mapping->i_mmap_rwsem`). The per-core queue



(a) Lmbench - 120core

그림 2.10: Lmbench CPU utilization on 120 core.

version of the LDU shows the best performance and scalability outperforming stock Linux by 1.5x and Harris by 1.1x. In addition, although the global queue version of the LDU has the global CAS operation, it also has high performance and scalability because the global CAS operations are mitigated by two LDU techniques;it had 2% performance degradation compared with per-core queue version of the LDU. Furthermore, the stock Linux shows the highest idle time(58%)(see figure 2.8) since it waits to acquire semaphore(i.e., `anon_vma->rwsem`, `mapping->i_mmap_rwsem`). Surprisingly, although two LDU have higher idle time than the Harris version, the throughput shows higher than the Harris because of our efficient algorithm.

2.3.3 Exim

To measure the performance of the Exim, we used the MOSBENCH, a many-core scalability benchmark. The Exim email server is designed to scale

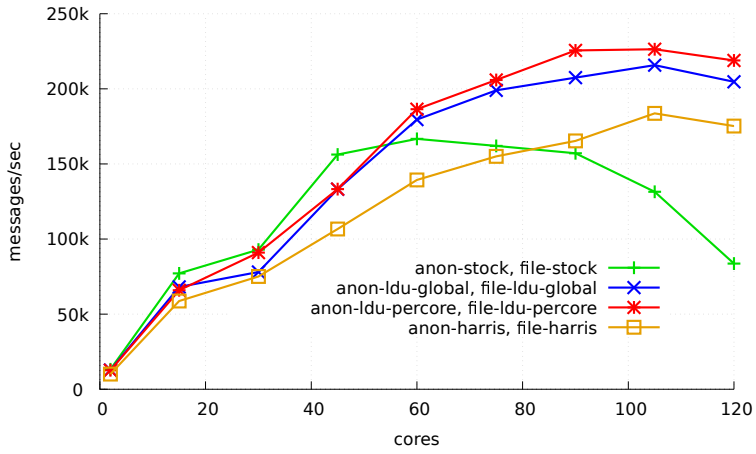


그림 2.11: Scalability of Exim.

because the Exim delivers messages to mail boxes in parallel using the Linux process;the Exim is a fork-intensive workload. Clients ran on the same machine and each client sent to a different user to prevent contention on user mail file. The Exim was bottlenecked by the filesystem [9] since the message body appends to the per-user mail file, so we used the separated tmpfs to reduce filesystem bottlenecks.

Results shown in figure 2.11 indicate that the Exim scales well up to 60 core, but the stock Linux performance decreases near 60 core. Both the Harris and the LDU increase up to 105 core because they can execute concurrent updates without the semaphores. The per-core queue version of the LDU performs better due to the fact that it can reduce cache coherence-related overheads outperforming stock Linux by 2.6x and Harris by 1.2x. Even though we applied the scalable solutions, the Exim shows the limitation near 105 core since the Exim process has a relatively large size of virtual address mapping that leads to the clearing virtual address mappings overheads and many soft page fault during the process

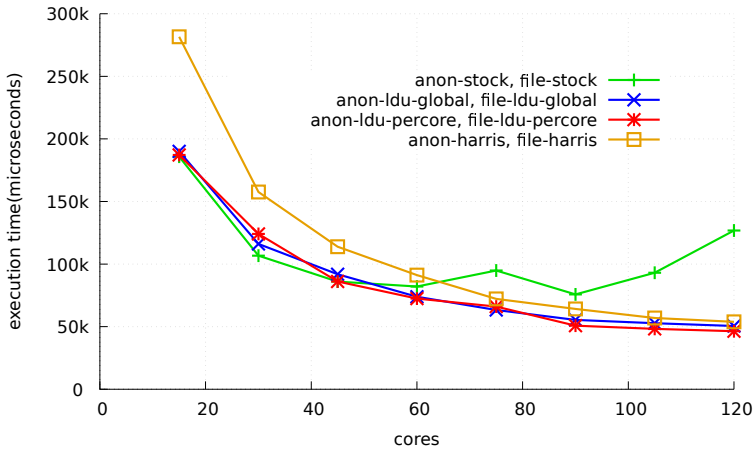


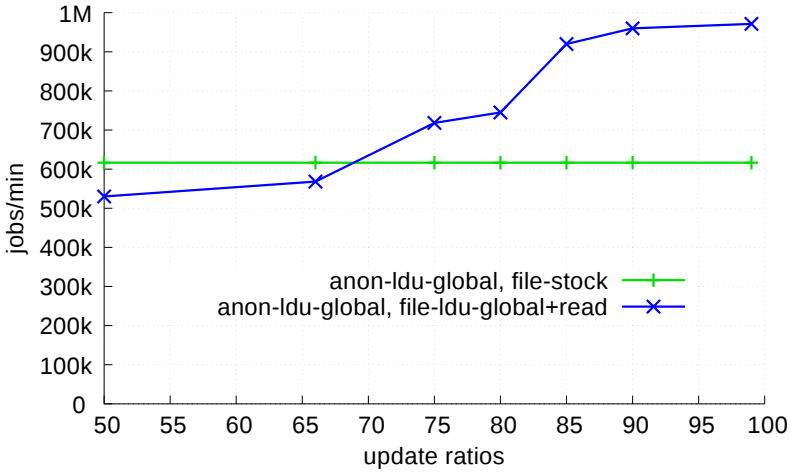
그림 2.12: Execution time of the process management workload in the Lmbench.

destruction which executes more slowly with more remote socket memory access. The Harris has 15% idle time, whereas per-core queue version of the LDU has 22% idle time because the LDU has the efficient algorithm(see figure 2.9).

2.3.4 Lmbench

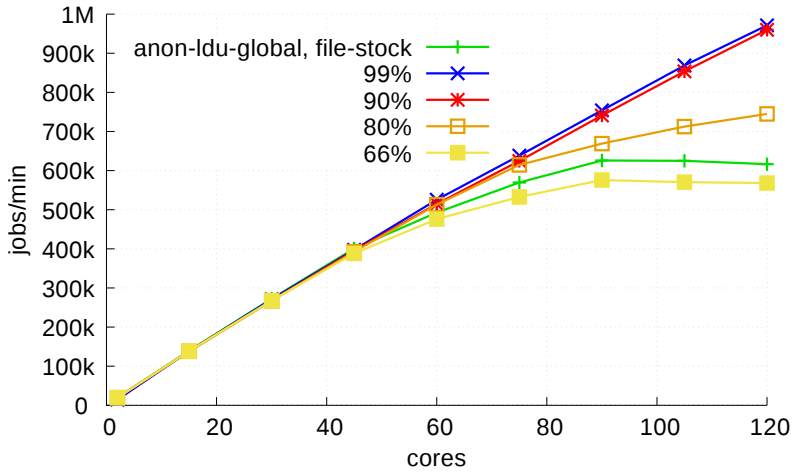
The Lmbench has various micro benchmarks including process management workloads. We used the process management workload in the Lmbench. This workload is used to measure the basic process primitives such as creating a new process, running a program, and context switching. We configured process create workload to enable the parallelism option(the value was 1000).

The results for the Lmbench are shown in Figure 2.12, and the results show the execution times. Up to 45 core, the stock Linux scales linearly and then the execution time goes up to grow. The per-core version of the LDU outperforms the stock Linux by 2.7x and the Harris by 1.1x at 120 core. While the



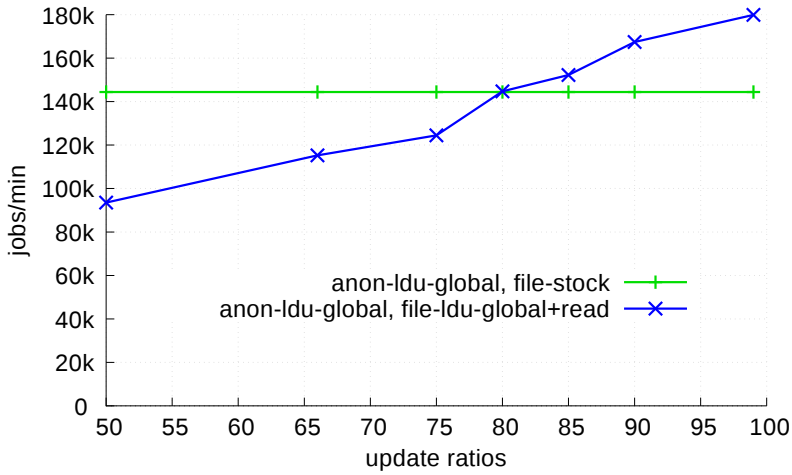
(a) AIM7 - 120core

그림 2.13: Performance depending on update ratios and scalability.



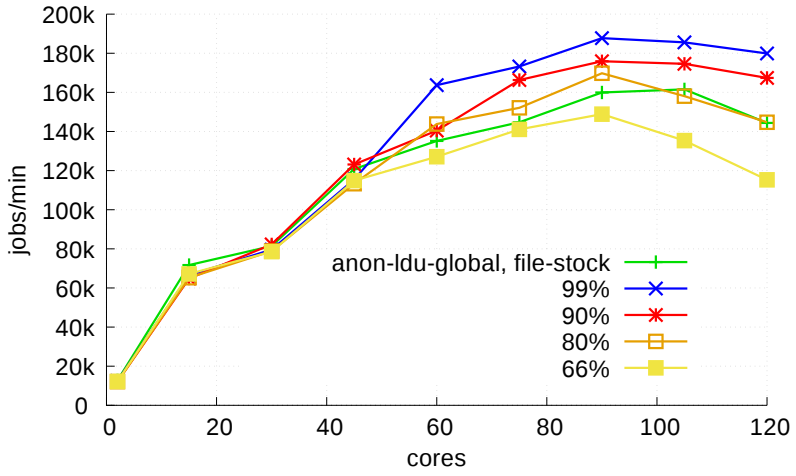
(a) AIM7 - scalability

그림 2.14: Performance depending on update ratios and scalability.



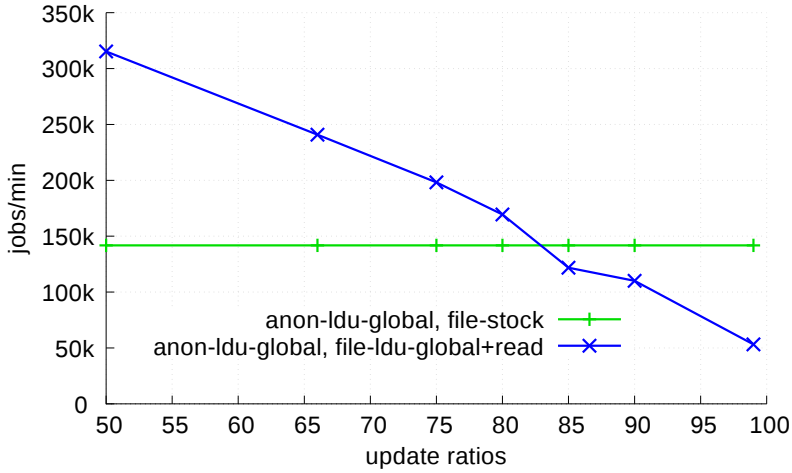
(a) AIM7 - 120core

그림 2.15: Performance depending on update ratios and scalability.



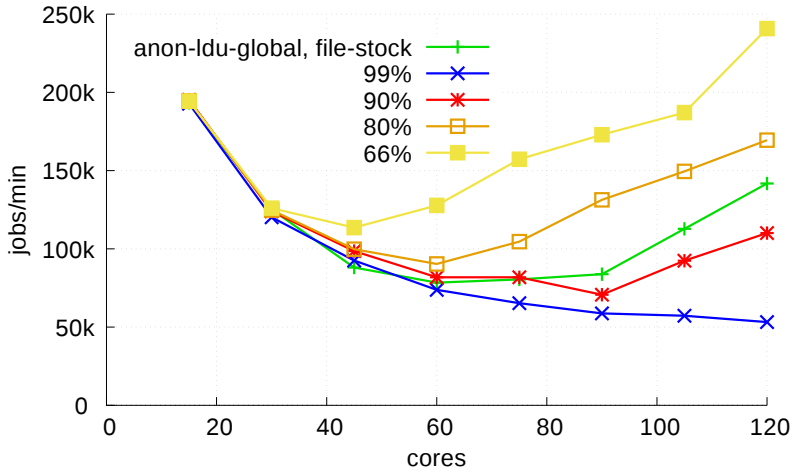
(a) AIM7 - scalability

그림 2.16: Performance depending on update ratios and scalability.



(a) AIM7 - 120core

그림 2.17: Performance depending on update ratios and scalability.



(a) AIM7 - scalability

그림 2.18: Performance depending on update ratios and scalability.

stock Linux has 69% idle time, other methods have approximately 35% idle time since the stock Linux waits to acquire two rmap semaphores(`anon_vma->rwsem`, `mapping->i_mmap_rwsem`)(see figure 2.10). Indeed, our main motivation in the LDU is to improve the performance and scalability on the many-core systems, so we did not consider a low cores performance(under 30 cores). However, up to 30 core, our LDU is similar to the stock Linux performance, but the Harris has low performance up to 60 core.

2.3.5 Updates ratio

One question that could be raised regarding the proposed LDU scheme would be how the performance scalability is affected by the frequency of read operations since the proposed technique has only focused on update-heavy data structures with very low read ratios. Even read operations could be slower since read operation should perform codesynchronize function to apply logs.

To understand the effect of read operations, we performed another experiment with intentionally adding the read operations in proportion to update operations. The anonymous rmap used the global queue version of the LDU, and then we sequentially increased read (`lock`, `synchronize`) ratios regarding the file rmap. The upper graphs of figure 2.13 ,2.14 , 2.15, 2.16, 2.17 and 2.18 shows the performance on 120 core depending on its update ratios, and the lower graphs represent the scalability.

Since the AIM7 has less fork-intensive workload than other ones, the read operations are invoked relatively infrequently. As a result, although the data structure uses 75% update rates(3 update, 1 read), the LDU version of Linux has outstanding performance than stock Linux. The scalability of AIM7 shows that the LDU has substantially high scalability at the the 90% and the 99% update rates as well as the 80% update rates.

The Exim and the Lmbench show the extremely high fork-intensive work-

load. As a result, the stock Linux outperforms the LDU by approximately 80%, but the LDU outperforms the stock kernel after 85%. This explains the LDU has outstanding performance even when the read operations frequently occur.

제 3 장 배경 이론 및 관련 연구

3.1 Hardware

3.1.1 Cache

3.1.2 Atomic Operations System Archtecture

3.1.3 Memory Barriers

3.2 Operating systems scalability

To improve the scalability, researchers have attempted to create new operating systems [8] [40] or have attempted to optimize existing operating systems [9] [14] [16] [10] Our research belongs to optimizing existing operating systems in order to solve the Linux fork scalability problem. However, previous research did not deal with the anonymous reverse mapping, which is one of the fork scalability bottleneck.

3.2.1 Corey

3.2.2 fOS

3.2.3 Barrelfish

3.2.4 Tessellation

3.2.5 HeliOS

3.2.6 Bonsai

3.2.7 RadixVM

3.2.8 OpLog

3.3 Scalable lock

Scalable locks have been designed by the queue-based locks [35] [28], [**?**], [**?**] [11] [12] hierarchical locks [38] [13] and [**?**] [**?**] delegation techniques [25] [21] [26]. Our research is similar to the delegation techniques because the LDU's `synchronize` function runs as a combiner thread;it improves cache locality. However, our approach not only can improve cache locality but also can eliminate synchronization methods during updates due to using a lock-free manner.

3.3.1 Locking

3.3.2 Ticket Lock

3.3.3 Queued Lock

3.3.4 Flat Combining

3.4 Scalable data structures

Many scalable data structures with scalable schemes show different performances depending on their update ratios. In low and middle update rate, researchers have attempted to create new scalable schemes [33] [29] [24] [22] [39] or have attempted to adapt these scheme to data structures [7] [20] [14]. In high update rate, the OpLog shows significant improvement in performance scalability for update-heavy data structures in many core systems, but suffers from limitation and overhead due to time-stamp counter management. We substantially extend our preliminary work [27] not only to support per-core algorithm but also to apply the LDU to anonymous rmap due to improving the Linux kernel scalability.

3.4.1 RCU

3.4.2 Harris

3.4.3 RLU

제 4 장 결론 및 향후 연구

4.1 결론

We proposed and evaluated a novel concurrent update method, LDU, to maximally improve the performance scalability of Linux kernel in many core systems. Such improvement was possible through eliminating the synchronized time-stamp counters management overhead found in a previous well-known scheme, OpLog. Our experiments using a Linux kernel with our LDU implementation revealed that the proposed LDU shows better performance up to 2.7 times stock Linux kernel on the 120 core machine.

4.2 향후 연구

While the proposed technique achieves significant improvement in performance scalability through eliminating time-sensitive logs, there still remain the data structures to consider to further improve the scalability (i.e., stack and queue data structures). Future direction of research is to create a new synchronization scheme by combining two techniques(the LDU and the OpLog) to support the stack and queue.

The LDU is implemented on to Linux kernel 4.5-rc6 and available as open-source from `https://github.com/manycore-ldu/ldu`.

참 고 문 헌

[1] AIM Benchmarks. http://sourceforge.net/projects/aimbench.

[2] Linux test project. https://github.com/linux-test-project/ltp.

[3] LOCK STATISTICS. https://www.kernel.org/doc/Documentation/locking/lockstat.txt.

[4] MOSBENCH. 2010. https://pdos.csail.mit.edu/mosbench/mosbench.git.

[5] Exim Internet Mailer. 2015. http://www.exim.org/.

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

[7] 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, 2014.

[8] 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, 2008.

[9] 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 *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, 2010.

[10] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. OpLog: a library for scaling update-heavy data structures. In *Technical Report MIT-CSAIL-TR2014-019*, 2014.

[11] 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.

[12] Davidlohr Bueso. Scalability Techniques for Practical Synchronization Primitives. volume 58, pages 66–74, December 2014.

[13] Milind Chabbi and John Mellor-Crummey. Contention-conscious, Locality-preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 22:1–22:14, 2016.

[14] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, 2012.

[15] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, 2013.

[16] Austin T. Clements, M. Frans Kaashoek, and Nickolai and Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, 2013.

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

[18] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[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, 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, 2015.

[21] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. *SIGPLAN Not.*, 47(8):257–266, February 2012.

[22] 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.

[23] 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, 2015.

[24] 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, 2001.

[25] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. SPAA '10, pages 355–364, 2010.

[26] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Delegation Locking Libraries for Improved Performance of Multithreaded Program. In *Euro-Par 2014 Parallel Processing*, pages 572–583, 2014.

[27] Joohyun Kyong and Sung-Soo Lim. LDU: A Lightweight Concurrent Update Method with Deferred Processing for Linux Kernel Scalability. In *Proceedings of the IASTED International Conference, Parallel and Distributed Computing and Networks (PDCN 2016)*, 2016.

[28] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171. IEEE Computer Society, 1994.

[29] 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, 2015.

[30] Dave McCracken. Object-based reverse mapping. In *In Proceedings of the 2004 Ottawa Linux Symposium*, 2004.

[31] Paul McKenney. Some more details on Read-Log-Update. 2016. https://lwn.net/Articles/667720/.

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

[33] 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, October 1998.

[34] 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.

[35] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[36] 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, June 2016.

[37] Adam Morrison. Scaling synchronization in multicore programs. *Queue*, 14(4):20:56–20:79, August 2016.

[38] Zoran Radovic and Erik Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 241–. IEEE Computer Society, 2003.

[39] 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, 2012.

[40] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 3–14, 2010.