

경량 로그 기반 자연 업데이트 기법을 활용한  
리눅스 커널 확장성 향상

A Lightweight Log-based Deferred Update for  
Linux Kernel Scalability

지도교수 임 성 수

이 論文을 ○○學碩(博)士學位 請求論文으로 提出함

2016 년 10 월 15 일

국민대학교 대학원

컴퓨터 공학과 컴퓨터공학 전공

경 주 현

2016

金國民의

○ ○ ○ 學碩(博)士學位 請求論文을 認准함

2013년 ○월 ○일

(“논문인준일”)

審查委員長  인

審查委員  인

審查委員  인

審查委員  인 박사과정만 삽입표기

審查委員  인 박사과정만 삽입표기

국민대학교 대학원

# 차례

차례	.....	
그림 차례	.....	ii
요약	.....	iv
<b>제 1 장 서론</b>		<b>1</b>
1.1 개요	.....	1
1.2 논문의 기여	.....	4
1.3 논문 구성	.....	5
<b>제 2 장 연구 배경 및 관련 연구</b>		<b>6</b>
2.1 병렬화 운영체제 역사	.....	6
2.2 최근 운영체제 병렬화 연구	.....	11
2.2.1 새로운 운영체제 제안	.....	11
2.2.2 기존 운영체제 최적화	.....	16
2.3 확장성 있는 락 연구	.....	23
2.3.1 큐 기반의 락(Queued Lock)	.....	25
2.3.2 계층적 락	.....	26
2.3.3 Delegation techniques	.....	26
2.4 확장성 있는 자료구조 연구	.....	30
2.4.1 확장성 있는 자료구조를 위한 동기화 기법	.....	30
2.4.2 확장성 있는 자료구조	.....	36
<b>제 3 장 논문에서 해결하고자 하는 구체적인 문제</b>		<b>38</b>

<b>제 4 장</b>	<b>로그기반 동시적 업데이트 방법</b>	<b>44</b>
<b>4.1</b>	설계 . . . . .	44
<b>4.1.1</b>	접근법 . . . . .	44
<b>4.1.2</b>	실행 예 . . . . .	48
<b>4.1.3</b>	알고리즘 . . . . .	51
<b>4.2</b>	리눅스 커널에 적용 . . . . .	54
<b>4.2.1</b>	익명 역 매핑 . . . . .	54
<b>4.2.2</b>	파일 역 매핑 . . . . .	56
<b>4.2.3</b>	자세한 구현 내용 . . . . .	57
<b>4.3</b>	평가 . . . . .	60
<b>4.3.1</b>	실험 환경 . . . . .	60
<b>4.3.2</b>	AIM7 . . . . .	63
<b>4.3.3</b>	Exim . . . . .	64
<b>4.3.4</b>	Lmbench . . . . .	66
<b>4.3.5</b>	업데이트 비율 . . . . .	69
<b>제 5 장</b>	<b>결론 및 향후 연구</b>	<b>73</b>
<b>5.1</b>	결론 . . . . .	73
<b>5.2</b>	향후 연구 . . . . .	74
<b>참 고 문 헌</b>		<b>75</b>
<b>Abstract</b>	. . . . .	<b>84</b>

## 그림 차례

2.1	90년대 공유 메모리 시스템 . . . . .	7
2.2	CPU 발전 동향. . . . .	8
2.3	공유 메모리 시스템 . . . . .	9
2.4	공유 메모리 시스템 . . . . .	10
2.5	corey 운영체제 address space 공유 방법 . . . . .	13
2.6	Barrelfish 구조 . . . . .	14
2.7	FusedOS 구조 . . . . .	15
2.8	linux scalability 분석 연구 . . . . .	16
2.9	Address space 문제와 BosaiVM을 이용한 해결 . . . . .	18
2.10	RadixVM의 해결 방법 . . . . .	20
2.11	Flat combining 방법 . . . . .	27
2.12	OpLog의 업데이트 방법 . . . . .	28
2.13	RCU 예제 . . . . .	31
2.14	RCU의 delayed free의 시점 . . . . .	32
2.15	Non-locking synchronization . . . . .	34
2.16	간단한 Non-blocking 스택 알고리즘. . . . .	35
2.17	Harris 삭제 . . . . .	37
3.1	AIM7-multiuser 성능 확장성 . . . . .	38
3.2	120코어에서 락 때문에 기다리는 시간 . . . . .	39
3.3	anonymous 역 매핑의 문제 . . . . .	40
3.4	파일 역 매핑의 문제 . . . . .	41
3.5	120코어에서의 lock_stat 결과 분석 . . . . .	42
3.6	업데이트 직렬화의 문제 . . . . .	43

3.7	동기화된 타임스탬프 카운터 머징에 따른 오버헤드 . . . . .	43
4.1	7개의 업데이트 명령과 1개의 리드 명령에 대한 LDU 예. . . . .	48
4.2	LDU의 동시적 삽입에 대한 알고리즘. . . . .	50
4.3	LDU의 동시적 삽제에 대한 알고리즘. . . . .	50
4.4	로그를 적용하는 알고리즘. . . . .	52
4.5	리눅스 익명 역 매핑에 LDU를 적용한 그림. . . . .	55
4.6	리눅스 파일 역 매핑에 LDU를 적용한 그림. . . . .	56
4.7	LDU의 전역 큐. . . . .	57
4.8	LDU의 퍼코어 큐. . . . .	59
4.9	실험 환경. . . . .	60
4.10	lockfree 리스트로 변경하기 전 자료구조 . . . . .	61
4.11	lockfree 리스트로 변경 후 자료구조 . . . . .	62
4.12	AIM7-multiuser 확장성. . . . .	63
4.13	120코어에서 AIM7 CPU 사용량. . . . .	64
4.14	120코어에서 EXIM CPU 사용량. . . . .	65
4.15	120코어에서 Lmbench CPU 사용량. . . . .	66
4.16	Exim 확장성. . . . .	67
4.17	Lmbench의 프로세스 관리 벤치마크에 대한 실행시간. . . . .	68
4.18	업데이트 비율에 따른 AIM7 성능. . . . .	68
4.19	업데이트 비율에 따른 AIM7 확장성. . . . .	69
4.20	업데이트 비율에 따른 Exim 성능. . . . .	69
4.21	업데이트 비율에 따른 Exim 확장성. . . . .	70
4.22	Lmbench performance depending on update ratios. . . . .	70
4.23	업데이트 비율에 따른 Lmbench 성능. . . . .	70
4.24	업데이트 비율에 따른 Lmbench 확장성. . . . .	71

## Abstract

# A Lightweight Log-based Deferred Update for Linux Kernel Scalability

*by Kyong, Joohyun*

*Department of Computer Science*

*Graduate School, Kookmin University,*

*Seoul, Korea*

In highly parallel computing systems with many-cores, a few critical factors cause performance bottlenecks severely limiting scalability. The kernel data structures with high update rate naturally cause performance bottlenecks due to very frequent locking of the data structures. There have been research on log-based synchronizations with time-stamps that have achieved significant level of performance and scalability improvements. However, sequential merging operations of the logs with time-stamps pose another sources of scalability degradation.

To overcome the scalability degradation problem, we introduce a lightweight log-based deferred update method, combining the log-based concepts in the dis-

tributed systems and the minimal hardware-based synchronization in the shared memory systems. The main contributions of the proposed method are:(1) we propose a lightweight log-based deferred update method, which can eliminate synchronized time-stamp counters that limits the performance scalability;and (2) we implemented the proposed method in the Linux 4.5-rc6 kernel for two representative data structures (anonymous reverse mapping and file mapping) and evaluated the performance improvement due to our proposed novel light weight update method. Our evaluation study showed that application of our method could achieve from 1.5x through 2.7x performance improvements in 120 core systems.

# 제 1 장 서론

## 1.1 개요

최근 코어 수가 증가함에 따라, 멀티코어에서 매니코어로 변화되고 있다. 매니코어 시스템에서 성능에 대한 확장성(Scalability)은 가장 중요한 요소이다. 이러한 매니코어 시스템의 확장성 중에, 운영체제의 커널(Kernel) 때문에 전체 시스템의 성능이 제한을 받는다. 그리고, 운영체제 커널 중 가장 많이 사용되는 것은 리눅스(Linux) 커널이다. 그 이유는 리눅스 커널은 멀티코어에 최적화가 가장 잘된 운영체제 중에 하나이기 때문이다. 하지만 이렇게 멀티코어에 최적화된 리눅스 커널도 매니코어 시스템에서는 여전히 성능에 대한 확장성에 문제가 있다 [14] [48]. 확장성 문제 중 가장 큰 문제는 커널의 자료구조 중 업데이트 락(Lock) 경쟁에 대한 문제 때문이다 [44] [41].

이처럼 업데이트 직렬화(Serialization) 문제를 해결하기 위해 여러 동시적 업데이트(Concurrent Update) 방법들이 연구되고 있다 [9] [41] [27]. 동시적 업데이트 방법을 사용하여 업데이트 직렬화 문제를 해결하는 연구들은 업데이트 비율에 따라 많은 성능 차이를 보인다. 이러한 방법들은 높은 업데이트 비율을 가진 자료 구조 때문에 발생하는 확장성 문제에 대해서는 여전히 효율적이지 않다. 높은 업데이트 비율을 가진 자료에 대한 해결책 중 하나는 캐시 통신 병목(cache communication bottleneck) 현상을 줄인 로그 기반(log-based) 알고리즘 [33] [15]을 사용하는 것이다. 로그 기반 알고리즘은 업데이트가 발생하면, 자료구조의 업데이트 명령(update operation)을 퍼코어(per-core) 또는 원자적(atomic)으로 로그로 저장하고 읽기 명령(read operation)을 수행하기 전에 저장된 로그를 수행하는 것이다. 따라서, 리더(reader)는 최신 데이터를 읽

게 되며, 이것은 마치 CoW(Copy on Write)와 유사하다 [43] [49].

S. Boyd-Wickizer et al.는 동기화된 타임스탬프 카운터(synchronized timestamp counters) 기반의 퍼코어 로그를 활용하여 높은 업데이트 비율을 가진 자료구조를 대상으로 동시적 업데이트 문제를 해결함과 동시에 캐시 병목현상(cache communication bottleneck)을 줄였다 [15]. 동기화된 타임스탬프 카운터 기반의 퍼코어 로그를 활용한 동시적 업데이트 방법은 업데이트 부분만 고려했을 때, 퍼코어에 데이터를 저장함으로 굉장히 높은 성능 확장성을 가진다[1]. 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 프로세싱(sequential 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(Lightweight log-based Deferred Update)를 개발하였다. LDU는 타임스탬프 카운터가 필요한 명령어 로그(operation log)를 업데이트 순간 지우고, 매번 로그를 생성하지 않고 재활용하는 방법이다. 이로 인해 동기화된 타임스탬프 카운터 문제와 캐시 커뮤니케이션 병목현상에 대한 문제를 동시에 해결하였다. 해결 방법은 분산 시스템(distributed system)에서 사용하는 동기화된 타임스탬프 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, 캐시 커뮤니케이션 병목 현상을 줄인 LDU는 기존 로그 기반 알고리즘들의 장점들을 모두 포함할 뿐만 아니라 추가적인 장점을 가진다. 첫째로, 업데이트가 수행하는

시점 즉 로그를 저장하는 순간에는 파인 그레인드 동기화 기술(fine-grained synchronization)이 필요가 없다. 따라서 동기화(synchronization) 오버헤드 없이 동시적 업데이트를 수행할 수 있다 둘째로, 저장된 업데이트 명령어 로그를 코스 그레인드 동기화 기술(coarse-grained synchronization)과 함께 하나의 코어에서 수행하기 때문에, 캐시 효율성이 높아진다 [33]. 다음으로, 기존 여러 자료구조에 쉽게 적용할 수 있는 장점이 있다. 게다가 마지막으로, 로그를 저장하기 전에 로그를 삭제하므로 더욱 빠르게 로그의 수를 줄일 수 있다.

우리는 위와 같은 장점을 가지는 LDU를 리눅스 커널에서 높은 업데이트 비율 때문에 성능 확장성 문제를 일으키는 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 [46]를 대상으로 성능 개선을 보였다. 개선은 기존 리눅스 커널보다 120코어에서 각각 1.5x, 2.6x, 2.7x 배 성능 향상을 한다.

## 1.2 논문의 기여

본 논문은 다음과 같은 기여를 하였다.

- 우리는 높은 업데이트 비율을 가지는 자료구조를 위한 새로운 로그 기반 동시적 업데이트 방법인 LDU를 개발하였다. LDU는 동기화된 타임스 템프 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 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 향상된다.

### 1.3 논문 구성

본 논문의 구성은 다음과 같다.

??장에서는 리눅스 확장서의 문제점에 대해서 기술한다. 4.1에서는 LDU 설계에 대한 내용과 LDU의 예에 대해서 설명하며 4.2장에서는 LDU를 리눅스 커널에 어떻게 적용하였는지를 설명한다. 4.3장에서는 본 논문에서 제안한 방법에 대한 실험 결과에 대해서 설명한다. 2장에서는 관련 연구와 배경지식에 대해서 설명한다. 마지막으로 5장에서는 본 논문의 결론과 향후 연구에 대해서 기술한다.

## 제 2 장 연구 배경 및 관련 연구

### 2.1 병렬화 운영체제 역사

본 장에서는 현시점에서 운영체제 병렬화가 필요한 이유와 함께 운영체제의 병렬화의 역사에 관해서 설명한다. 그동안 운영체제의 병렬화는 시분할 시스템, 클라이언트(client) 서버(server) 구조, 그리고 SMPs(Shared Memory Processor) 그리고 최근 프로세서에 코어가 많아 지는 멀티코어로 총 4단계에 거쳐 발전해 왔다 [34].

첫 번째 단계에서는 시분할 시스템에서 사용되는 병렬화이다. 60년대부터 70년대에서의 운영체제 병렬성은 시분할(time sharing) 시스템을 가졌다. 즉 컴퓨터 한대에 여러 사용자가 동시에 사용되었고, 대부분이 1개의 프로세서로 이루어졌다. 이 시점에서 병렬 처리 연구는 I/O 병렬화 프로그램에 대한 연구가 진행되었다 [11][CTSS 1962]. 최대한 프로세서를 이용률(utilization) 를 높여서 I/O를 처리하기 위해 즉 커널은 병렬로 I/O를 처리하기 위해 다른 프로그램 커널로 문맥교환되어 실행되도록 만들었다.

초기 컴퓨터 중 일부 프로세서들은 시분할 시스템과 멀티프로세서의 병렬화를 고려하여 만들었다(예를 들어, 버로우스(Burroughs)의 B5000 [42]). 따라서, 병렬화에 대해서 많은 관심과 노력이 이루어졌다. 그 결과 병렬화 관련 초기 많은 이론인 암달의 법칙 [7], 멀티스(Multics)에서의 트래픽 컨트롤 [54], 데드락 발견(deadlock detection) 그리고 락 오더링(locking ordering)등 많은 이론들이 생겨나게 되었다. 70년대 하나의 프로세서 위에서 병렬화를 제공하기 위해 많은 연구 및 개발이 이루어졌고, 실제 단일 프로세스에 여러 유저에게

시분할 기능으로 병렬화를 제공하는 Unix 커널 [53]이 개발되었다.

두번째 단계에서는 80년대와 90년대에는 컴퓨터의 가격이 개인이 구매가 가능할 정도 내려갔으며, 로컬 네트워크로 여러 유저가 협업하면서 작업할 수 있는 환경이 되어 클라인트 서버 환경을 위한 병렬화가 이루어졌다. 문제는 여러 유저가 수행할 서비스(services)에 대한 병렬화가 필요하게 되었고, 따라서 논커널 프로그래머들도 커널의 기능이 필요하여, 서버의 커널이 인터페이스(interface)를 추가하여 유저들에게 병렬화 서비스를 제공하였다.

그 결과 많은 운영체제 병렬화 기술들이 이 시점에 연구 개발되었다. 예를 들어 스레드(Thread), 락(Locks) 그리고 컨디션 변수(Condition variables)등이 이 시점에 많은 연구가 이루어졌다. 이벤트(events)와 스레드(threads)에 대한 논쟁 [50] [57] 그리고 Accent [52], Mach [6] , V [21] 등 새로운 운영체제들이 제안되었다. 이러한 연구들은 마이크로커널(microkernel)에 영감을 주었고, 결국 최근 많은 운영체제가 사용하고 있는 Pthreads[POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)]에 대해 영향을 주었다. 새로운 운영체제 뿐만 아니라 새로운 언어들(예를들어 Mesa [36])도 연구되었고, 결국 가비지 컬렉션등에 대한 연구가 같이 진행되어, 그 결과 것들이 최근 자바(JAVA)와 고(Go) 언어등에 영감을 주었다. 결론으로 커널의 인터페이스를 서버 개발자에게 노출하여 서버를 병렬로 이용 할 수 있게 만들었다.

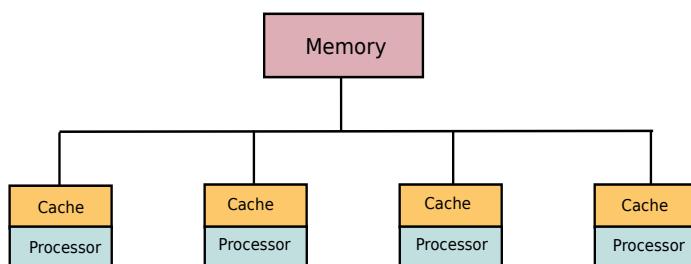


그림 2.1: 90년대 공유 메모리 시스템

다음 단계에서는 90년대 각각의 프로세서가 메모리를 공유하는 개념의 컴퓨터인 SMP(Shared-memory Multi Processors)가 낮은 가격으로 보급이 되어서, 커널 또는 서버 개발자는 이 때부터 심각하게 운영체제 병렬화에 대해서 고려하게 되었다. 예를 들어, 운영체제 커널은 BKL(Big Kernel Lock) 등을 지원하며 병렬화 기능을 제공하기 시작하였다. 이 시점 많은 회사(BBN Butterfly, Sequent, SGI, Sun, Thinking Machines 등)가 운영체제 병렬화에 대해서 연구하기 시작했다. 그 결과 많은 운영체제 성능 확장성에 대해서 새로운 개념 예를 들어, MCS 락 [47], 유저 레벨 쓰레딩 [40], NUMA 메모리 관리 [12], 가상 머신 모니터(virtual machines monitor) [18] 등이 제안되었다.

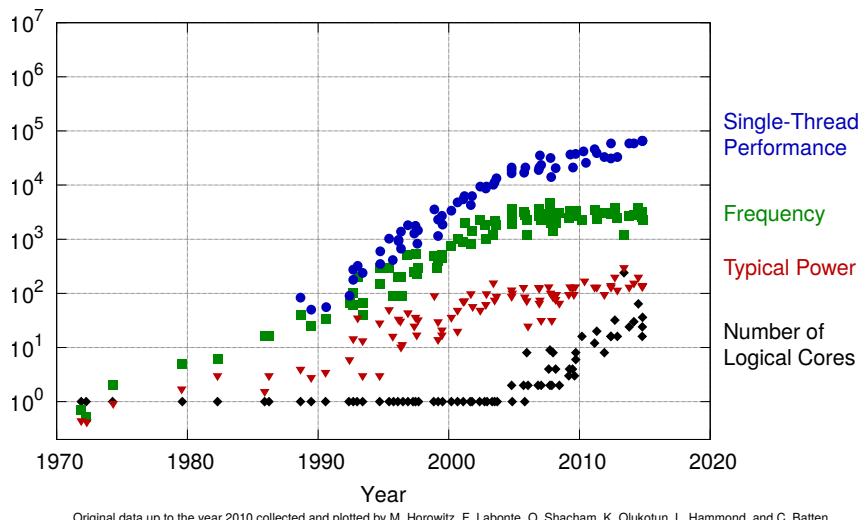


그림 2.2: CPU 발전 동향.

마지막 단계는 멀티코어이다. 그림 4.12과 같이 주파수는 계속 증가하다가, 2000년대 중반 멈추고, 그 때부터 코어수가 증가하고 있다. 따라서 코어수

가 100개 이상의 멀티코어 프로세서들도 등장함에 따라, 공유 때문에 야기하는 새로운 문제가 발생하기 시작하였다.

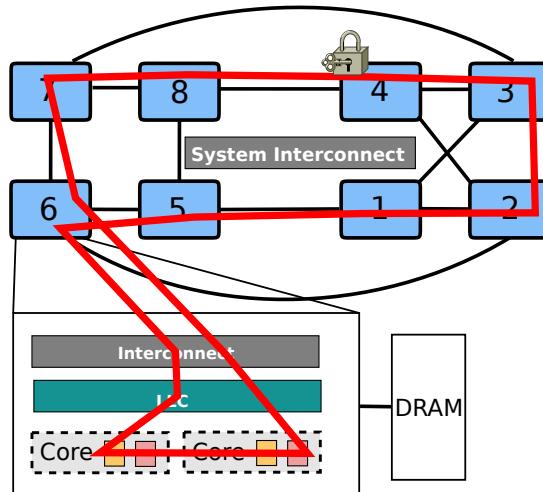


그림 2.3: 공유 메모리 시스템

최근에 발생하는 문제들은 상당 부분이 캐시라인(cache-line)의 공유 때문에 발생하는 문제이고, 이를 해결하기 위해서 최근에는 여러 운영체제(section 2.2), 락 기법(section 2.3), 그리고 자료구조와 알고리즘(section 2.4)들이 개발되고 있다.

20년동안 싱글코어를 대상으로 연구가 되었고, 최근에는 이러한 문제를 해결하기 위해 파티션ning 기법을 활용하여, 파티션ning 기법의 한 예로서는, 그림 x-x와 같이 공유되는 전역 자료 구조를 각자 CPU에서 처리하도록 하는 방법이 있다. 이러한 방법은 시스템 전반에 발생하는 캐시 커뮤니케이션 오버헤드를 줄일 수 있다.

Non-blocking synchronization은 장점은 여러 스레드들이 락 기반으로 자원을 관리함에 따라 발생하는 문제를 해결할 수 있다. 가장 큰 장점은 스레드

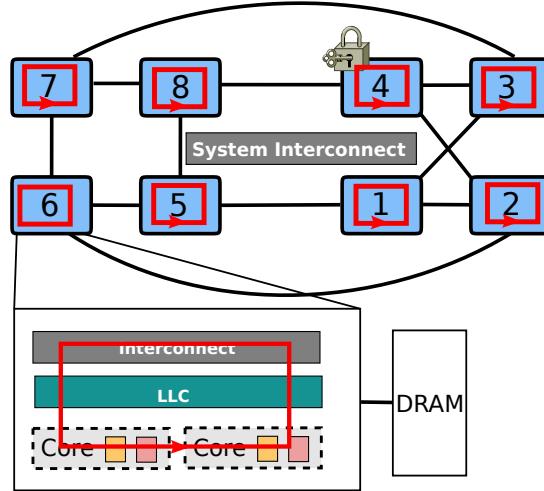


그림 2.4: 공유 메모리 시스템

또는 프로세스가 락 때문에 기다리는 시간을 제거할 수 있다. 이 것은 락을 얻기 위해 기다리는 시간을 최소화 할 뿐만 아니라 무한 루프 때문에 무한정 기다리는 데드락 같은 상황까지 제거 할 수 있다. 다음으로 모든 락은 락 자체의 오버헤드를 가지고 있는데 이것을 제거할 수 있다. 예를 들어 코어 수가 증가 할 수록 락 자체를 얻기 위해 원자적 명령을 이용한는데 이것은 캐시 일관성 트래픽을 발생한다. 이와 같이 Non-blocking 방법은 이러한 락 자체가 가지고 있는 문제점인 데드락(deadlock), 라이브락(livelock), 우선순위 역전현상(priority inversion)등을 제거 할 수 있다. 이러한 Non-blocking synchronization 기법을 사용하는 lock-free 자료 구조들은 성능을 향상 시킬 수 있다. 그 이유는 멀티코어 환경에서 공유되는 데이터를 접근하기 위해 직렬화 되는 부분이 매우 짧기 때문이다.

## 2.2 최근 운영체제 병렬화 연구

코어 수가 증가 되는 상황에서 병렬화가 중요해진 운영체제에 대한 연구는 성능에 대한 확장성을 향상 시키기 위해서, 새로운 확장성 있는 운영체제를 만들거나 [13] [59] [10] [37] [28] 기존 운영체제를 최적화 시키는 방법 [14] [22] [24] [15]을 시도하고 있다.

### 2.2.1 새로운 운영체제 제안

#### Corey

Corey [13]는 MIT의 Parallel and Distributed Operating Systems Group에서 개발하였다. 기본적인 철학은 커널 영역의 공유 데이터를 유저 응용프로그램이 사용할 수 있게 만들어 주어서, 공유 데이터 때문에 발생하는 경합 문제를 유저에게 해결할 수 있도록 공유에 대한 인터페이스를 제공하는 것이다. 그 이유는 프로세서 내의 코어 간의 캐시 일관성 작업 때문에 성능이 저하되는데 이러한 것의 원인을 운영체제가 응용프로그램과는 상관 없이 데이터를 공유하고 하드웨어 역시 응용프로그램에 상관없이 동기화 하는 방식을 사용하기 때문에 기존의 운영체제에서 취하는 방법은 매니코어 환경에서 성능 향상이 어렵다는 것이다. 이러한 문제를 해결하기 위해 응용프로그램의 워크로드에 따라 공유 문제를 해결할 수 있는 방향을 제공해준다. 이것은 exokernel[]의 개념을 가져와서 매니코어 시스템에 적용하였고, 이를 통해 확장성을 개선하였다.

이러한 Corey는 3가지 기본적은 개념을 가지고 있다. 그것은 address ranges, 커널 kernel core 그리고 shares이다. 첫째로, address ranges는 운영체제의 스레드간의 공유하는 데이터인 address space에 대해서 다룬다. 대부분의 운영체제는 address space를 space를 그림 2.5(a)와 같이 single address space로

구성하는 경우와 그림 2.5(b)와 같이 per-core 기반의 separate address space로 구성된다. 만약 single address space를 사용할 경우 모든 코어가 같은 address space를 사용함에 따라 반드시 락이 필요하다. 따라서 예를들어, 맵리듀스와 같은 응용프로그램을 사용할 경우 맵 단계에서 굉장히 많은 락 경합이 발생하게 된다. 또한, 만약 separate address space를 사용할 경우 리듀스 단계에서 공유하지 않은 데이터에 접근함에 따라 soft page fault가 많이 발생하는 문제가 있다. Corey는 이러한 문제를 address rages라는 개념으로 해결하였다. 이것은 separate address space를 제공함과 동시에 중간 결과를 공유할 수 있는 방법을 제공함으로 single address space의 장점을 동시에 취한다. 다음으로, kernel core는 응용프로그램을 공유 메모리를 사용하지 않고, 특정 코어에 독점 할당 시켜주고 공유는 IPC로 하도록 제공하는 기술이다. 따라서 스케줄러와 인터럽트등에 방해를 받지 않고 캐시 지역성을 높여 성능을 향상 시키는 방법이다. 마지막으로, 공유는 어떻게 커널 자료구조를 접근할 수 있는지에 대해서 제공해준다.

## Barrelfish

Barrelfish [10]는 취하리의 ETH와 마이크로 소프트(Microsoft)가 공동 연구하여 만든 운영체제이다. Barrelfish는 멀티커널(multikernel) 운영체제 중 하나이고, 기본적인 철학은 공유 메모리 시스템 기능들을 모두 분산 처리 방식으로 구현하자는 것이다. 예를 들어, 운영체제에서 각 코어는 네트워크로 분산 된 시스템으로 가정하고 메시지 패싱을 통해 분산 된 코어들 간에 통신을 하여, 성능을 향상 시킨 방법이다. 메시지 패싱 방법을 사용한 이유는, 오늘날 사용되는 캐시 구조로 된 시스템의 single shared interconnect가 코어가 증가 할 수록 cache coherence traffic 문제가 있기 때문에 하드웨어 cache coherence protocol을 사용하는 방법보다 메시지 패싱 방법이 오히려 더 좋기 때문이다.

이러한 Barrelfish의 구현은 그림 2.6와 같이 구현되었다. 커널 레벨에서

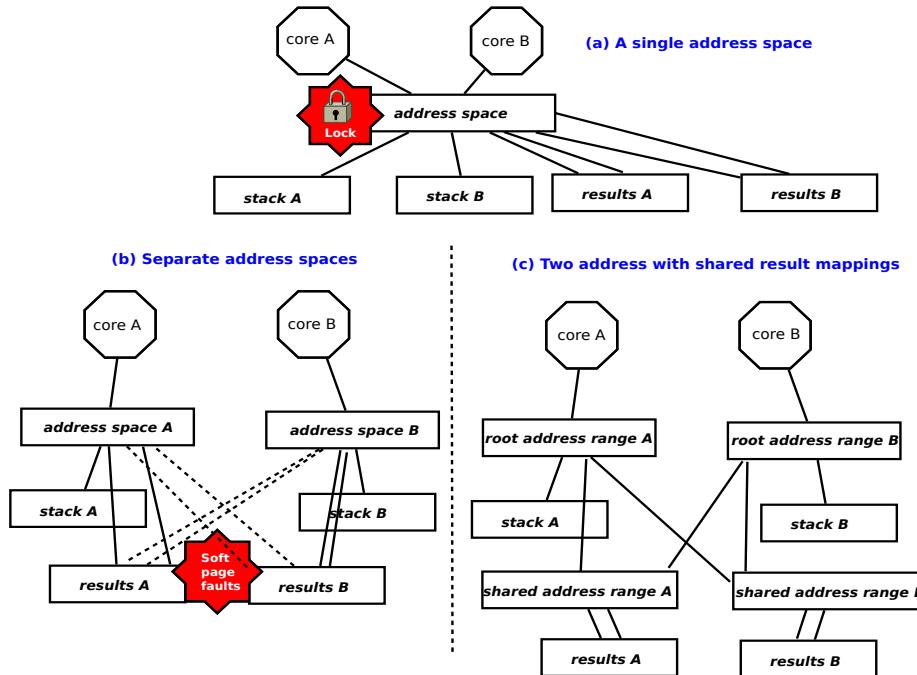


그림 2.5: corey 운영체제 address space 공유 방법

는 하드웨어와 밀접한 CPU driver가 하드웨어 인터페이스를 제공한다. CPU driver는 유저 레벨의 모니터(Monitor)와 함께 하나의 운영체제처럼 동작하며, 각 코어에 하나의 운영체제가 동작하는 것처럼 보이는 멀티커널(multikernel) 방법의 구조를 가진다. 응용프로그램은 여러 코어를 동시에 사용할 수 있는데, 이러한 환경을 제공하기 위해 존재하는 것이 모니터이다. 모니터는 운영체제의 기본 기능을 제공하기 위해 존재하며, 공유 메모리를 사용하기보다는 replication과 IPC를 방법을 사용한다. 모니터는 view라는 상태를 가지고 replication을 수행한다. 이것은 분산 시스템에서 사용하는 방식과 같이 시스템을 구성하여, 캐시 커뮤니케이션 때문에 발생하는 시스템의 interconnect의 로드를 줄일 수 있다.

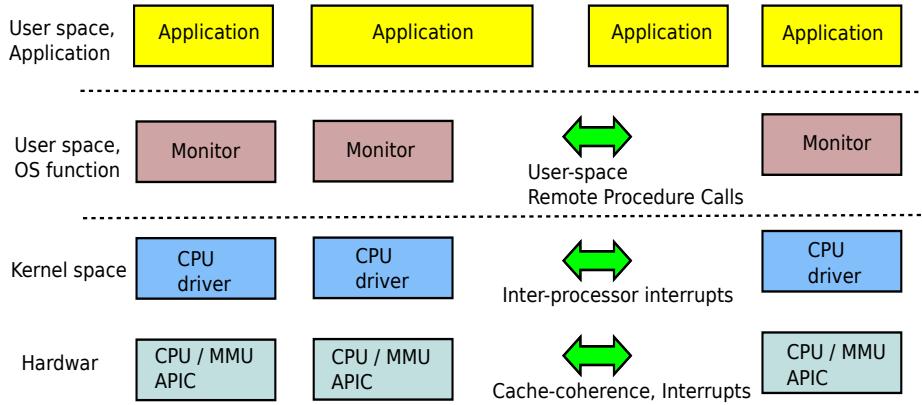


그림 2.6: Barreelfish 구조

Barreelfish의 구조적인 철학은 공유를 최대한 줄이는 것인데, 결국 이것은 로드 밸런싱을 수행할 수 없는 단점을 가진다. 예를 들어 하나의 코어에 많은 스레드들이 같이 돌고 있고, 다른 코어에는 아무런 스레드도 없는 경우 분산 시스템처럼 수행되므로, 동적으로 스레드에 대한 정보들을 다른 코어로 전송할 수가 없다. 즉 분산 시스템의 전형적인 단점인 로드 밸런싱이 어려우므로 응용프로그램에 따라 로드가 한 쪽에 몰리는 경우, 느려진 코어의 일을 기다려야 하기 때문에 전체적인 성능이 저하될 수 있다.

## FusedOS

FusedOS는 IBM 연구소에서 개발되었으며, 모노리티k 구조와 마이크로 구조의 장점을 활용한 운영체제이다. FusedOS의 철학은 기존 연구는 대부분 경량 커널(LWK: Light-Weight Kernel) 또는 정량 커널(FWK: Full-Weight Kernel) 둘 중 하나의 방식으로만 개발되어온 운영체제를 LWK와 FWK를 혼합하여 만든 운영체제이다. 그 이유는 FWK으로 사용하는 리눅스로 인하여

기존 반들어진 라이브러리등을 모두 재사용할 수 있기 때문이다. 또한 LWK를 통하여 FWK이 가지고 있는 근본적인 확장성 문제를 해결할 수 있다는 것과, 과학에서 사용되는 특정한 목적으로 HPC 위에서 개발된 응용프로그램은 커널의 간섭이 없는 LWK에서 더 효율적으로 동작하기 때문이다.

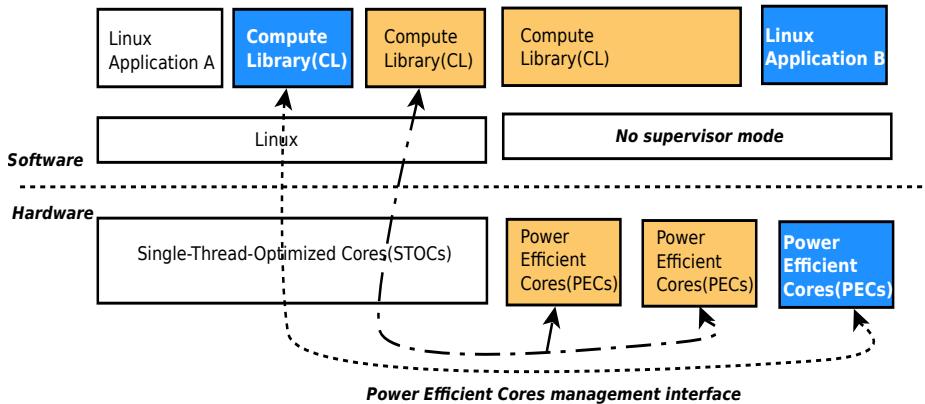


그림 2.7: FusedOS 구조

이러한 FusedOS의 구조는 그림 2.7와 같다. 그림과 같이 FusedOS의 하드웨어는 성능 좋은 코어 그룹(STOCs: Single-Thread-Optimized Cores)과 전력에 효율적인 코어 그룹(PECs: Power Efficient Cores)로 구성되어 있다. PEC는 STOC의 기능 중 하나이고 supervisor 모드를 포함하고 있지 않는다. STOC에는 리눅스 운영체제가 동작하고, 리눅스에 기능을 추가하여 Compute Library(CL)이 PEC에 접근이 가능하도록 설계되었다.

CL은 마치 리눅스 응용프로그램으로 동작하며, 실행을 하면 가벼운 커널을 PEC 메모리에 전달되고 그 후 가벼운 커널은 PEC 코어에서 동작한다. 이를 통해, HPC 응용의 성능을 보여주면서 리눅스 운영체제의 기능을 제공할 수 있는 장점을 가진다. FusedOS의 성능은 HPC 운영체제 코어와 연산을 위한 코어가 분리되었기 때문에, 운영체제의 방해를 받지 않아 기존 리눅스보다 높은

성능을 보인다. 이것은 기존 운영체제의 병목현상 때문에 발생하는 확장성을 문제를 해결할 수 있다. 하지만 FusedOS 역시 독립적으로 코어에 LWK를 할당하여 호출하는 방법을 사용하므로, 응용프로그램을 실행하고 종료하는데 추가적인 시간이 필요하다.

### 2.2.2 기존 운영체제 최적화

#### Linux scalability

MIT PDOS 연구 그룹은 새로운 운영체제가 아닌 리눅스 커널을 대상으로 매니코어 환경에서 확장성을 연구하였다. 실제 사용되는 7가지의 응용프로그램(Exim, memcached, Apache, PostgreSQL, gmake, Psearchy, and MapReduce)을 가지고 MOSBENCH라는 응용프로그램 벤치마크를 만들어 리눅스 커널을 대상으로 성능을 측정하였고, 측정 중 발생되는 여러 문제를 해결하여 리눅스 커널의 확장성을 향상시켰다.

<b>Solution</b>	<b>Issue</b>
<b>Multicore packet processing</b>	<b>Concurrent accept system calls contend on shared socket fields.</b>
<b>Sloppy counters</b>	<b>Resolving preference counts contend (dentry, vfsmount, dst_entry, protocol usage)</b>
<b>Lock-free comparison</b>	<b>Walking file name paths contends on per-directory entry spin lock.</b>
<b>per-core data structures</b>	<b>Resolving path names to mount points contends on a global spin lock</b>
<b>Eliminating false sharing</b>	<b>False sharing in page, net_device and device</b>
<b>DMA buffer allocation</b>	<b>DMA memory allocations contend on the memory node 0 spin lock.</b>
<b>Avoiding unnecessary locking</b>	<b>inode lists, Dcache lists, Per-inode mutex</b>

그림 2.8: linux scalability 분석 연구

그림 2.8와 같이 총 7가지의 기술을 활용과 응용프로그램을 직접 수정하여 커널 확장성을 개선하였다. 먼저 멀티코어 패킷 프로세싱이 있다. 리눅스 커널은 네트워크 멀티코어 패킷을 위해 멀티 큐를 사용하여 성능이 향상되었으나, 만약 클라이언트 연결 주기가 짧을 경우 이러한 패킷 커넥션이 성능을 저해한다. 따라서 이 연구 그룹은 동시 다발적으로 연결을 요청하는 Apache 응용프로그램과 같은 경우는 커널의 소켓 함수 중 accept을 수정하여 per-core 큐에 저장하도록 수정하였다. 이 것은 싱글 listening 소켓을 보호하기 위한 락을 제거할 수 있어 확장성을 향상시킨다.

다음으로 이 연구 그룹은 리눅스의 레퍼런스 카운트 때문에 발생하는 캐시 일관성 트래픽을 제거하기 위한 sloppy counter를 만들었다. 이것을 통해, 디렉토리 엔트리 오브젝트(dentries), 마운트된 파일 시스템 오브젝트(vfsmounts) 그리고 네트워크 프로토콜에서의 메모리 할당을 추적하기 위한 전역 변수를 sloppy counter로 수정하여 성능을 향상 시켰다. 또한, 리눅스 커널은 디렉토리 엔트리 캐시의 이름을 찾는 부분에 per-dentry spin lock 때문에 문제가 있다. 이러한 문제를 해결하기 위해 리눅스의 lock-free page cache lookup protocol과 유사한 방법을 활용하여 전역 spin lock을 제거하여 성능을 향상 시켰다. 또한 per-core 방법을 사용한 방법과 캐시라인의 false sharing 때문에 발생하는 성능 저하 문제, 이더넷 디바이스 DMA 버퍼가 한쪽 노드에만 할당되어 발생하는 확장성 문제와 그리고 불필요한 락을 제거하여 리눅스 커널의 확장성을 향상 시켰다.

## BonsaiVM

BonsaiVM은 MIT Parallel and Distributed Operating Systems Group에서 개발한 리눅스 커널을 위한 가상 메모리 시스템이다. 리눅스의 멀티 스레드들은 하나의 address space를 공유하게 되는데, 이러한 공유된 address space 때문에 mmap과 soft page fault간에 공유된 address space를 보호하기 위해, reader-

writer 세마포어를 사용한다. 하지만, 많은 락 경합 때문에 스레들이 블락 걸리 는 현상이 많이 생기는데, 이 때문에 코어가 많아 지면 성능이 떨어지는 문제가 있다.

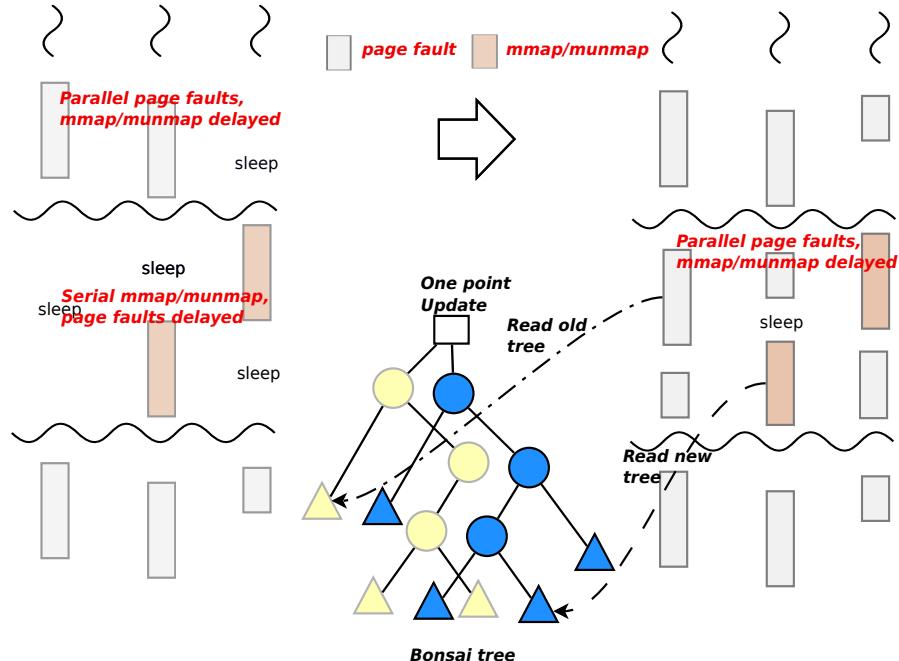


그림 2.9: Address space 문제와 BosaiVM을 이용한 해결

그림 2.9의 왼쪽 부분은 이러한 address space를 문제를 보여준다. 병렬로 수행이 가능한 페이지 폴트 때문에 mmap/munmap 함수는 블락이 걸리고, 병렬로 수행이 불가능한 mmap/munmap 함수가 수행되면 mmap/munmap 함수뿐만 아니라 페이지 폴트까지 블락에 걸린다. 이러한 single address space 문제해결하기 위해서 앞에서 설명한 corey 운영체제를 만드는 등 여러 연구들이 진행되었지만, 이 연구에서는 새로운 운영체제가 아닌 리눅스 커널을 대상 으로 RCU라는 리눅스 커널의 동기화 기법을 사용하여 이 문제를 해결하였다. 이것은 새로운 운영체제를 제안하는 것이 아니라, 리눅스 커널을 대상으로 개

선한 연구이며, 리눅스 커널 중 상당히 복잡한 가상 메모리 시스템에 직접 RCU라는 동기화 기법을 사용하여, 성능 확장성을 향상시킨 연구이다.

BonsaiVM은 총 3가지 기법(fault locking, hybrid locking/RCU, pure RCU)을 통해 single address space 문제를 해결하였다. 이 중 앞의 두가지 방법은 리눅스 커널의 구현 의존 적인 해결방법이고, pure RCU는 기존 리눅스 커널의 레드-블랙 트리를 사용하지 않고 RCU를 사용할 수 있는 새로운 Bonsai 트리 자료구조를 만들어 문제를 해결한 방법이다. Bonsai 트리는 그림 2.9과 같이 이진 트리로 구성되어 있으며, 가장 큰 특징은 루트 노드의 업데이트가 원자적 명령으로 한번에 할 수 있다는 것이다. 그 이유는 Bonsai 트리는 합수형 트리 형식으로 개발되어서, 밸런스를 수행하는 동안에도 리더들은 old 트리의 값을 읽으며 수행할 수 있다. 하지만 병렬로 수행할 수 있는 장점은 있지만, 트리 검색에 기존 레드-블랙 트리에 비해 많은 성능 오버헤드가 존재하여, 실제 리눅스 커널에 적용되지는 못하였다.

## RadixVM

RadixVM은 single address space 때문에 발생하는 확장성 문제를 해결하기 위해, 기존 연구를 위해 개발된 운영체제를 대상으로 가상 메모리에 대한 부분을 수정하여 문제를 해결한 연구이다. 그 이유는 리눅스의 가상 메모리를 수정하는 것은 굉장히 복잡하여, 적용하기 힘들기 때문에 덜 복잡한 운영체제에 새로운 개념을 적용하였다. RadixVM은 BonsaiVM과 같이 VM의 공유되는 address space가 map, unmap, page fault 함수 들로 인해 서로 경쟁함으로 발생하는 문제를 새로운 3가지 접근을 통해 해결하였다.

첫째로, 성능 좋은 밸런스 트리를 사용하지 않고, radix 트리를 이용하는 것이다. RadixVM의 궁극적인 목적은 VM과 관련된 operation에 대해서는 최대한 공유를 피하자는 것이다. 만약 모든 메모리 맵을 배열로서 만들면 모든

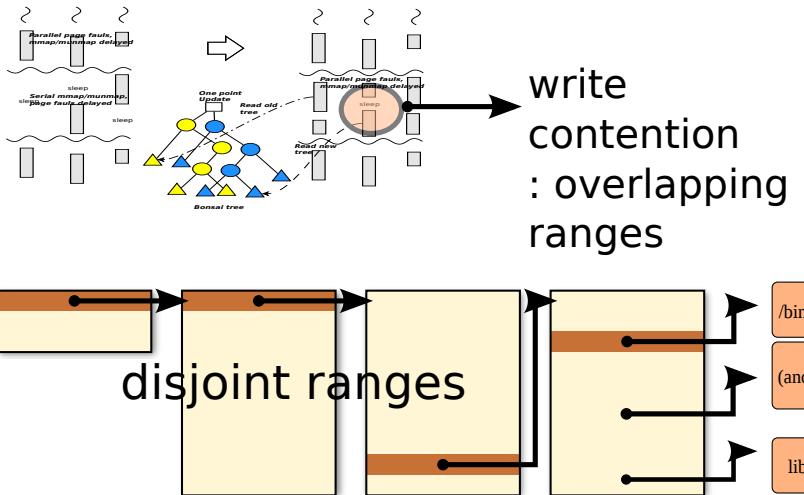


그림 2.10: RadixVM의 해결 방법

VM관련 명령들이 충돌이 일어나지 않는다. 하지만 이러한 방법은 너무 많은 메모리 사용량이 요구된다. 이러한 문제를 해결하기 위한 가장 좋은 방법은 하드웨어 페이지 테이블처럼 동작하는 radix 트리를 이용하는 것이다. radix 트리는 트리의 읽기와 쓰기를 서로 다른 부분에 접근하도록 만들어 준다. 따라서, 배열을 사용한 방법과 같이 충돌 없이 사용가능하므로 동시에 여러 스레들이 접근 할 수 있을 뿐만아니라 메모리 사용량도 밸런스 트리와 비슷한 수준을 유지 할 수 있다.

두번째 기법은 munmap할 때 발생하는 TLB shutdown 문제이다. unmap 함수는 반드시 어떠한 코어의 페이지지도 매핑이 안된 상태로 끝나야 한다. 따라서 unmap 함수는 모든 코어에 TLB shutdown 메시지를 남기며, 이로인해 코어가 증가 할 수록 TLB shutdown이 많이 발생하게 된다. radixVM에서는 이러한 문제를 per-core 페이지 테이블을 이용하여 해결하였다.

RadixVM의 마지막 기법은 새로운 refcache라는 레퍼런스 카운터를 제공

한다는 것이다. 레퍼런스 카운터는 최근 운영체제의 가장 큰 확장성 저해요소 중에 하나이다. 이러한 레퍼런스 카운터는 3가지 종류로 개발되고 있다. 먼저 가장 쉬운 방법인 락을 이용하는 방법이 있다. 즉 모든 증가/감소 명령의 앞 뒤에 락을 호출하는 것이다. 이것은 락 때문에 상당히 많은 캐시-라인 경합이 발생된다. 다른 방법으로는 원자적 증가/감소 명령을 이용하는 것이다. 하지만 이러한 방법도 결국 전역 변수 때문에 캐시-라인 경합이 발생한다. 최근의 방법으로는 per-core 카운터를 이용하는 것이다. 하지만, 단순히 per-core 카운터를 이용하는 것은 코어 수에 비례하여 공간에 대한 오버헤드를 가지게 된다. RadixVM은 이러한 확장성과 메모리 사용량에 대한 오버헤드를 동시에 줄이기 위해, 10ms의 시간(epochs)을 기반으로 per-core에 저장된 델타 카운트를 체크하여 전역 카운터에 적용하는 방법을 사용하였다.

## SC rule

SC rule은 MIT PDOS 연구 그룹에서 연구한 운영체제의 확장성 개선을 새로운 관점으로 바라본 연구이다. 기존 연구들은 대부분 운영체제의 병목 지점을 추출한 후 이러한 병목지점을 해결하기 위해 새로운 동기화 기법을 개발하거나 기존 개발된 동기화 기법을 적용하는 방법을 사용하였다. 하지만, 이러한 방법들은 모두 워크로드가 다름에 따라 서로 다른 양상을 가지고, 또한 문제를 해결하는데 너무 오랜시간이 걸리는 문제점을 가지고 있다. 실제 확장성에 대한 문제는, 대부분 설계 단계에서 인터페이스를 확장성 있게 소프트웨어를 설계하면 해결된다는 것이다. 이러한 가정을 가지고 확장성을 위해서는 확장성있는 설계가 중요하며 그에 대해서 초점을 둔 논문이다. 그 이유는 리눅스등의 운영체제는 확장성 있게 설계 되었으나 응용프로그램의 사용방법에 따라 확장성 문제가 발생하기 때문이다. 예를 들어 원자적으로 값 X를 변수 A에 추가하고, 다른 CPU가 원자적으로 같은 변수에 Y라는 값을 추가하였다면, 두 가지의 명령의 순서는 상관 없고, 결국에는 X + Y의 값이 변수 A에 저장된다는 것이다. 또한, SC rule은 이러한 문제점을 발견하기 위해

메모리의 충돌을 발견하는 새로운 툴을 제안하였다. 이러한 툴을 통해 설계 단계에서 확장성 문제를 발견할 수 있으며, 이 툴을 통해 기존 운영체제(리눅스, sv6)의 문제점을 분석하였다.

저자는 POSIX의 오퍼레이션들은 가환성(commutativity)이 있지 않고, POSIX 오퍼레이션들을 확장성 있게 만드는 것은 어렵다. 하나의 예로 프로세스를 복사하는 `fork()`는 가환성을 가지고 있지 않다. 하지만 저자는 `fork()`를 `posix_spawn()`으로 수정하면 가환성을 가지게 되고, 이것은 결국 확장성을 향상시키게 된다. 또한 `open()`는 사용가능한 낮은 숫자부터 반환하게 되는데 이것을 `open()`과 `O_ANYFD`을 함께 사용하면 가환성을 있게 만들 수 있다고 제안하였다. 이처럼 리눅스 커널은 가환성을 잘 지키도록 호출 해주면, 확장성을 향상 시킬 수 있고 이것은 설계단계에서 처리할 수 있다는 것이다. 저자는 QEMU을 수정하여 리눅스의 가환성을 분석 할 수 있는 Commuter라는 툴을 제공한다.

## 2.3 확장성 있는 락 연구

락은 기본적으로 여러 스레드들을 안전하고 올바르게 동작하도록 만들 어주는 방법이다. 이처럼 여러 스레드를 안전하게 동작시켜주기 위해, 락은 하드웨어 동기화 명령들(CAS(Compare-And-Swap), fetch-and-add, SWAP 등)을 이용하여 구현된다. 기본적으로 락의 구현은 코어들과 RAM간에는 공유하는 버스가 있고, 이러한 버스를 이용하여 원자적으로 처리하기 위해 하드웨어 동기화 명령을 사용하여 구현한다. 예를 들어 x86 시스템 같은 경우 xchg 명령어를 통해 락을 쉽게 구현할 수 있다. 하지만 실제 시스템은 보다 더 복잡한 구조를 점 가지게 되는데, 복잡한 이유는 중간에 캐시 메모리와 일관성을 유지하기 위한 캐시 일관성 프로토콜과 매니코어 NUMA 구조에 최적화 되도록 구현해야 하기 때문이다.

이처럼 전통적으로 락의 primitive들은 기본적으로 두 종류로 구현되어 있는데, 하나는 busy-waiting 방법과 다른 하나는 sleeping 락인 blocking 방법으로 구현된다. 락을 잡고 있는 시간이 적을 때는 보통 busy-waiting 방법을 사용한다. 이 방법은 보통 락이 풀릴 때까지 전역 변수를 CAS연산을 반복적으로 수행하므로 구현된다. 이것은 blocking 방법의 문제점인 블락킹 오버헤드를 줄일 수 있으나, 수행도중 CPU를 계속 사용함에 따라 문제가 있다. 다른 방법인 sleeping 락은 락이 걸린동안 다른 스레드들을 동작시킬 수 있는 장점이 있으나, 스케줄러의 스레드관리에 의존적이고 스케줄링 정책에 의해 락의 공정성등에 영향을 준다.

최근 운영체제는 이러한 두가지 방법을 혼합하여 사용한다. 최근 리눅스 커널의 sleeping 락은 fastpath, optimistic spinning 그리고 slowpath로 구현된 3 가지의 상태의 락을 혼합하여 구현한다. 예를 들어 커널의 rw\_semaphore(reader-writer semaphore)는 아래와 같이 3가지 상태로 구현된다.

- **fastpath.** 아무도 락을 안잡고 있을 때, 원자적인 명령어(fetch\_and\_add)

를 이용하여 카운터를 수정하고 락과 함께 반환하는 부분이다.

- **optimistic spinning.** 다른 스레드가 락을 잡고 있어서 기다려야 하는 상황이다. 하지만 optimistic spinning은 많은 부분이 락을 잡은 후 바로 반환하는 부분이 많아서 이러한 부분은 blocking 오버헤드를 줄이기 위해 busy-waiting 방법으로 수행한다. 최근에는 모든 코어가 같은 전역 변수를 읽을 경우 많은 캐시 일관성 트래픽이 발생하므로, 락을 걸 코어에 등록하고 기다리는 코어에서는 해당 CPU의 로컬 변수만 주기적으로 확인하는 큐기반 락(MCS 기반 락)으로 구현한다.
- **slowpath.** 만약 락을 잡고 있는 시간이 길어지면, 대기 큐에 집어 넣고 sleep을 수행하는 sleeping 락을 수행한다.

이와 같이 이러한 하이브리드한 sleeping 락은 많은 성능 향상을 보인다.

이와 같이 확장성을 위해 락과 관련한 병렬 처리에서 고려해야 할 사항 정리하면 기본적으로 락으로 보호해야 할 임계 영역(critical region)의 길이을 최대한 짧도록 해야하고, 락 자체가 가지고 있는 오버헤드도 또한 고려해야 할 사항이다. 또한 락의 세분화(granularity) 정도에 따라 fine-grained 락을 사용 할지 coarse-grained 락을 사용해야 할지 고려해야한다. 다음으로 읽기-쓰기 비율(read-write ratios)을 고려하여 읽기가 많은 경우에는 reader-writer을 사용하고, 읽기가 많으면 stale 데이터도 괜찮은 자료구조에서는 RCU 같은 락을 사용해야 한다. 또한 약간의 공정성(fairness) 손해보더라도 확장성을 높이는 방법도 고려해야 한다. 마지막으로 최근 락에 대해서 중요한 요소로 생각하고 있는 캐시 일관성 때문에 발생하는 오버헤드를 고려해야 한다.

이러한 고려사항들을 적용하여 확장성있는 락에 대한 연구는 큐 기반의 락 [47] [39], [58], [55] [16] [17]과 계층적 락 [51] [20] [38] [19] 그리고 위임하는 방법(delegation techniques) [33] [29] [35]들이 연구되고 있다.

### 2.3.1 큐 기반의 락(Queued Lock)

모든 코어가 바라보는 전역 변수를 사용하기 때문에 발생하는 캐시 일관성 트래픽 문제는 락 내부에서도 발생한다. 그 동안 락의 구현은 하나의 전역변수를 대상으로 원자적 명령을 이용하여 구현하였다. 따라서 자연히 캐시 일관성 트래픽 문제가 발생하였는데, 이것을 해결하는 방법이 큐 기반의 락을 이용하는 것이다.

락 때문에 발생하는 캐시 일관성 문제를 해결하기 위한 가장 쉬운 접근하는 방법은 각 코어가 모두 다른 캐시-라인의 데이터를 가지고 스피드을 하면 쉽게 해결이 된다. 즉 각각의 코어가 read-only 스피드을 하고, 반환 하는 스레드가 명시적으로 해당 코어의 캐시-라인 데이터를 원자적 명령을 이용하여 락을 반환하는 것이다. 이러한 방법의 문제점은 모든 락이 모든 코어의 중복되지 않은 캐시-라인 데이터를 위한 공간을 확보해야 한다는 것이다. 이것은 현실적으로 메모리 낭비가 심하다. 이러한 문제를 해결한 것이 CLH 큐 기반 락이다. 이러한 큐 기반의 락들의 기본 철학은 모든 코어가 같은 전역 변수를 바라 보며 스피드하지 않고, 각자의 로컬 변수를 바라 보며 스피드하도록 하는 것이다. 대신 락을 반환하는 스레드가 직접 해당 코어의 변수를 수정하는 일을 하는 것이다.

이러한 메모리 낭비가 심한 문제를 해결하기 위해 MCS 락이 개발되었다. 각 락에 대한 waiter들을 링크드 리스트로 보관하자는 것이 기본적인 아이디어이고, 이것은 하나의 스레드는 반드시 하나의 락만 기다린다는 아이디어를 활용하여 리스트로 기다리는 스레드들을 관리하였다. 즉 기존 많은 공간(락 \* 스레드 수)이 필요할 것을 적은 공간(락 + 스레드 수)로도 구현이 가능하게 되었다. 큐 기반의 락들도 여러가지가 있으나 본 논문에서는 그 중 최근 리눅스 커널에서 사용하는

MCS 락의 성능은 적은 코어 즉 코어 수가 2개 일경우 티켓 락이 더 좋은 성능을 보이지만, 코어 수가 많아 질수록 MCS 락은 좋은 성능을 보인다.

이처럼 락을 확장성 있는 락을 사용하면, 락 때문에 발생하는 확장성 문제 즉 성능이 갑자기 떨어지는 현상을 막을 수 있으나, 근본적으로 확장성 있는 시스템을 구축하려면 반드시 임계 영역의 길이를 줄여야한다. 리눅스 커널도 역시 처음에는 티켓 락 기반의 스피드락을 사용했으나, 2013년 이후 티켓-스피드락을 MCS-스피드락으로 변경하였다. 염밀하게 말하면 리눅스 커널은 MCS의 기본적인 자료구조 사이즈가 크기 때문에, 커널은 MCS 락을 커널 크기에 맞도록 수정하여 사용한다. 결국 2016년 부터는 리눅스 커널에서 티켓 락 기반의 스피드락은 사라지게 되었다[33].

### 2.3.2 계층적 락

계층적인 락들은 공통적으로 스케일이 큰 NUMA 환경에서 락의 확장성을 높이는 것에 있다. 특히 NUMA 환경에서는 원격 소켓의 데이터를 접근하는 것 때문에 성능이 저하되는 문제를, 소켓 단위로 lock의 migration을 줄여서 향상시키는 방법이다. 또한 계층적 락에 대한 연구는 여러 소켓을 global lock을 소켓 안에서 관리하는 local lock를 두어 약간의 공정성(fairness)를 손해보더라도, NUMA 노드의 지역성을 향상 시켜서 성능을 향상시키는 방법이다.

### 2.3.3 Delegation techniques

#### Flat Combining

Flat combining(FC) [33]는 여러 코어에서 락을 각자 락을 호출하면서 명령을 수행하는 것 보다, 하나의 코어에서 한 스레드가 해당 명령어들을 모아서 하나의 전역 락을 사용하여 처리하는 것이 보다 효율적이고 이러한 특징을 이용한 방법이다. 그리고 이러한 철학을 하나의 알고리즘화 한 것이 FC 방법이다. FC는 두 가지 장점을 가지는데 가장 먼저 FC는 락을 자주 수행하지

않으므로, 락에 의한 캐시 일관성 트래픽이 상대적으로 덜 발생한다는 것이다. 다음으로 하나의 스레드에서 여러 명령어들을 수행함에 따라, 캐시 지역성이 높아져서 여러 스레드에서 락을 걸며 수행한 방법보다 높은 성능을 가진다.

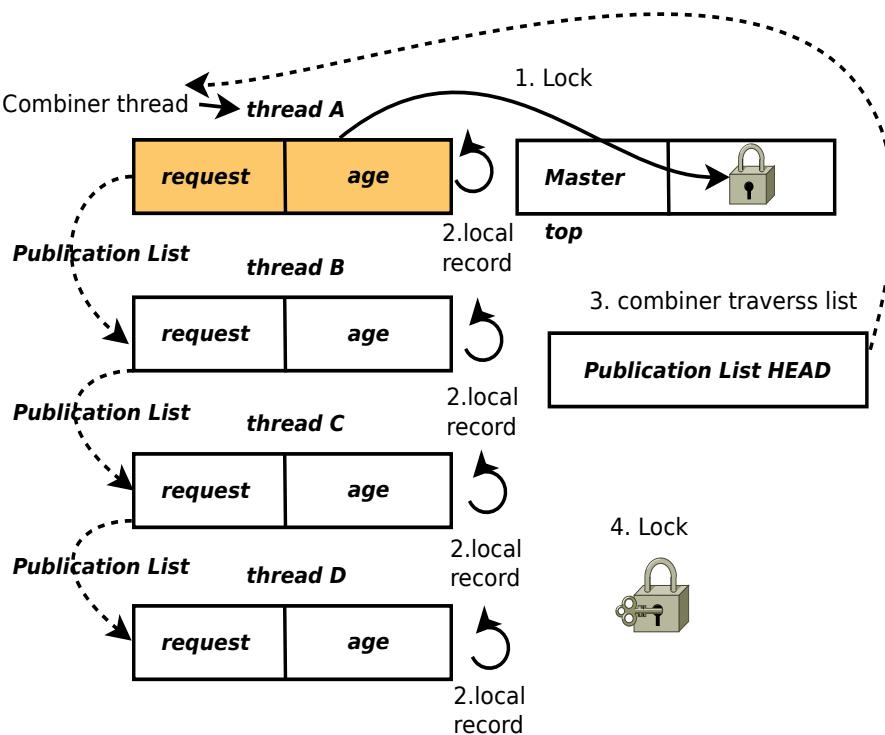


그림 2.11: Flat combining 방법

그림 2.11는 FC 알고리즘의 방법에 대해서 설명한다. 먼저 자료구조에 대한 명령어(예를 들어 스택같은 경우 push 또는 pop)가 도착하면, 처음 받은 스레드는 락을 걸고, 해당 명령어를 수행한다. 동시에 다른 코어의 스레드들은 다른 명령어들을 수행하게 되는데, 각 코어의 스레드들은 받은 명령어들을 코어의 내부 변수에 요청 정보와 age를 저장을 한다. 처음 받은 스레드의 명령이

끝나면 이때 부터 각 스레드에 저장된 명령을 순회하면서 combier 스레드가 수행하고, 마지막으로 락을 푼다.

## OpLog

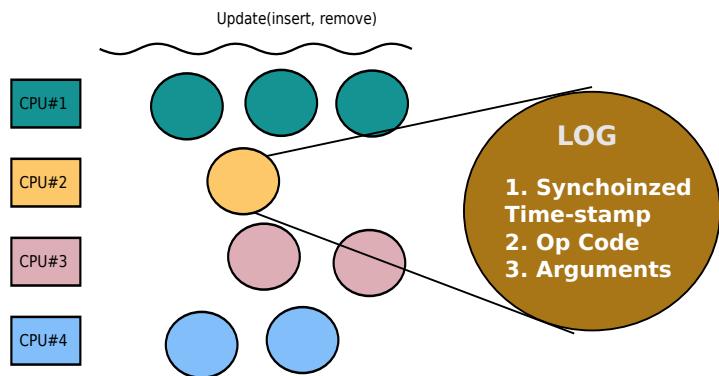


그림 2.12: OpLog의 업데이트 방법

OpLog는 RCU와 반대로 업데이트 비율이 높은 업데이트 헤비(Update heavy)한 자료구조를 위해 만든 동기화 기법 중 하나이다. 업데이트가 많아질 경우, 기존 연구들은 모두 공유 데이터를 접근함에 따라, 이때 발생하는 캐시 일관성 트래픽이 굉장히 많이 발생한다는 것이다. 이러한 문제점을 해결하기 위한 방법 중 하나가 로그 기반으로 처리하는 것인데, OpLog는 이러한 로그 기반 방법을 동기화된 타임스탬프 카운터를 이용하여 해결하였다. 각 코어에 타임스탬프와 함께 명령어를 로그로 저장을 한 다음, 읽기가 수행되기 전에 저장된 로그를 타임스탬프 정보와 함께 처리하는 방법이다. 그림 2.12는 각 코어에 저장된 로그 정보를 보여준다. 업데이트 명령어가 발생하면, 각 코어

는 타임스탬프, 명령어 코드 그리고 명령어를 처리하기 위한 정보들을 함께 저장한다.

## 2.4 확장성 있는 자료구조 연구

많은 확장성 있는 방법과 사용되는 자료구조들은 업데이트 비율에 따라 다른 성능을 가진다. 낮거나 중간 정도의 업데이트 비율에서는 연구자들은 새로운 확장성 있는 기법 [45] [41] [32] [30] [56]을 연구하거나 그 기법을 자료구조에 적용 [9] [27] [22]을 하도록 시도하고 있다. 높은 업데이트 비율에서는 OpLog가 매니코어의 업데이트 비율이 높은 자료 구조에 대해서 상당히 높은 성능 확장성을 가진다.

### 2.4.1 확장성 있는 자료구조를 위한 동기화 기법

#### RCU

확장성을 위한 대표적 동기화 기법인 RCU는 McKenney와 Slingwine에 의해 개발되었고, 동기화 기법 때문에 발생하는 오버헤드를 최소화 시킨 방법이다. 특히 RCU는 리더들을 보호하기 위해 사용하는 동기화 기법의 오버헤드를 최소화 시킨다. 단점으로는 RCU의 라이터가 수행하는 방법은 복잡하고 느리다. 이러한 단점에도 불구하고 리더들이 수행하는 락의 오버헤드가 적고, 여러 리더와 업데이터 하나가 동시에 수행이 가능하므로 RCU는 현재 리눅스 커널에서 상당히 많이 사용되고 있다.

RCU의 기본 철학은 특정 시점에서 오브젝트를 복제해서 처리한다. 그림 2.13은 이러한 RCU의 예를 보여준다. 그림에서 1단계에는 A, B, C, D, E 오브젝트 중 A, B, D 오브젝트를 리더들이 읽는 과정을 보여준다. 만약 이 순간 D 오브젝트를 수정하려 하면, RCU는 복사본을 할당 받고, 새로운 값인 오브젝트 N으로 수정을 한다. 그리고 다음 단계에서는 atomic한 연산을 통해서 오브젝트 C와 N을 연결한다. 이 순간 오브젝트 D를 읽고 이는 리더와 다른 리더들은 아무런 블락 없이 계속 읽기를 수행할 수 있으며, 동시에 업데이트까지

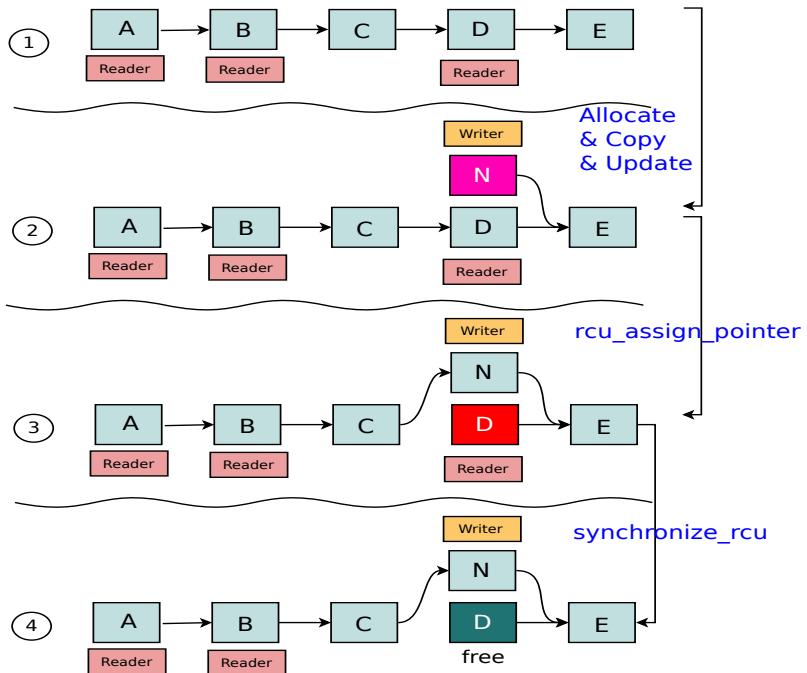


그림 2.13: RCU 예제

수행할 수 있어서 성능이 향상된다. 마지막으로 `syncroinze_rcu()` 함수를 통해 리더가 읽기를 마칠 때 까지 기다리고, 읽기가 끝나면 바로 `free()`를 수행한다. 이 때 마지막 리더가 읽을 때 까지 기다리는 시간을 RCU에서는 grace period라 부른다.

RCU는 기본적으로 3가지 특징을 가진다. 첫 번째로 Lock-free 리더이다. 실제로 RCU의 리더들은 아무런 락 또는 배리어(barrier)를 소유하지 않고 수행되며, 리드 구간에서는 per-core 자료구조에 단순히 enter/exit를 기록하여 수행한다. 따라서, 락 발생하는 캐시 일관성 트래픽이 발생하지 않는다. 두 번째로, 싱글 포인터 업데이트이다. RCU의 writer는 atomic 명령으로 one pointer 업데이트를 수행한다. 이러한 특징으로 인해 여러 리더들과 한가지 업데이터가

동시에 동작할 수 있다. 마지막으로, RCU는 delayed free를 수행한다. 노드를 바로 free를 하지 않고, 모든 리더들이 리드 구역을 벗어난 경우 까지 기다린 후 해당 노드를 free한다. 이를 통해 안전하게 노드를 자원 해제할 수 있다.

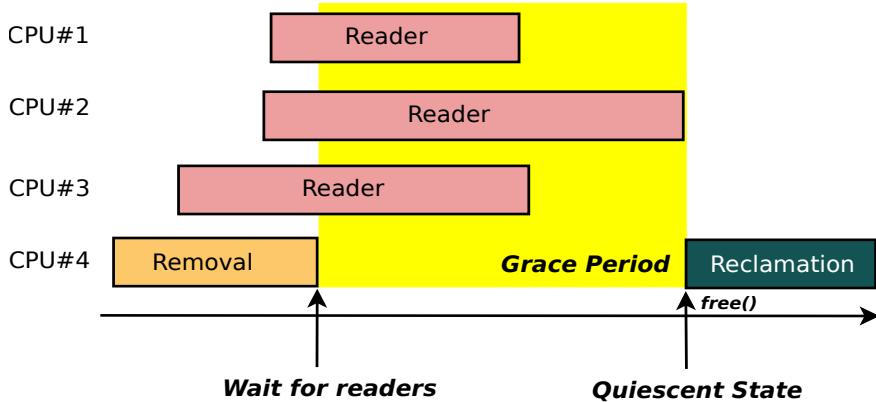


그림 2.14: RCU의 delayed free의 시점

그림 2.14는 RCU의 delayed free의 시점을 보여준다. RCU는 Removal 명령이 도착하면 Removal 명령을 수행하고 바로 `free()`를 수행하지 않는다. 그 이유는 그 동안 해당 데이터는 리더들이 사용하고 있을 가능성이 있기 때문이다. 그 순간부터 grace period 라고 부르는 리더들이 종료되기를 기다리고, quiescent state가 되면 그 때 `free()`를 수행한다. 현재는 리눅스 스케줄러에 의존하여 quiescent state를 판단하나, 이러한 quiescent state를 마지막 리더가 끝나는 순간 바로 판단할 수 있는 방법들이 연구해야 할 과제이다.

## RLU

RCU는 read-mostly 자료구조에 적합한 방법이나 RCU는 사용하기에 프로그래머의 노력이 필요하고, 라이터들이 증가할 수록 많은 오버헤드를 가진

다. 또한 RCU의 delayed free는 모든 리더가 종료했을 때 바로 quiescent state를 찾는 것이 아니기 때문에, 이것을 암시적으로 시간에 민감한 응용프로그램에 문제를 가진다. 이러한 문제를 해결하기 위해 만든것이 RLU [41]이다. RLU는 또 하나의 업데이트 문제를 해결한 로그 기반 알고리즘이나, 이것은 read-mostly 자료구조에서 라이터의 원자적 명령어에 의한 오버헤드가 심한 문제를 싱글 원자적 명령어를 사용한 global clock 변수와 오브젝트 레벨안에 per-core 단위로 로깅을 사용한 방법이다. 따라서 업데이트가 많아지면 여전히 문제가 생긴다.

## Non-locking synchronization

Non-blocking synchronization은 장점은 여러 스레드들이 락 기반으로 자원을 관리함에 따라 발생하는 문제를 해결할 수 있다. 가장 큰 장점은 스레드 또는 프로세스가 락 때문에 기다리는 시간을 제거할 수 있다. 이 것은 락을 얻기 위해 기다리는 시간을 최소화 할 뿐만 아니라 무한 루프 때문에 무한정 기다리는 데드락 같은 상황까지 제거 할 수 있다. 다음으로 모든 락은 락 자체의 오버헤드를 가지고 있는데 이것을 제거할 수 있다. 예를 들어 코어 수가 증가 할 수록 락 자체를 얻기 위해 원자적 명령을 이용한는데 이것은 캐시 일관성 트래픽을 발생한다. 이와 같이 Non-blocking 방법은 이러한 락 자체가 가지고 있는 문제점인 데드락(deadlock), 라이브락(livelock), 우선순위 역전현상(priority inversion)등을 제거 할 수 있다. 또한, Non-blocking synchronization 기법을 사용하는 lock-free 자료 구조들은 성능을 향상 시킬 수 있다. 그 이유는 멀티코어 환경에서 공유되는 데이터를 접근하기 위해 직렬화 되는 부분이 매우 짧기 때문이다.

이러한 non-blocking 동기화 기법은 그림 2.15와 같이 분류된다. 크게 보면 Wait-free방법과 Wait-free으로 구분되고, 이것은 1990대 초에 구분되었다. Wait-free 방법은 가장 구현하기 힘든 알고리즘이며, 모든 스레드가 딜레이 없

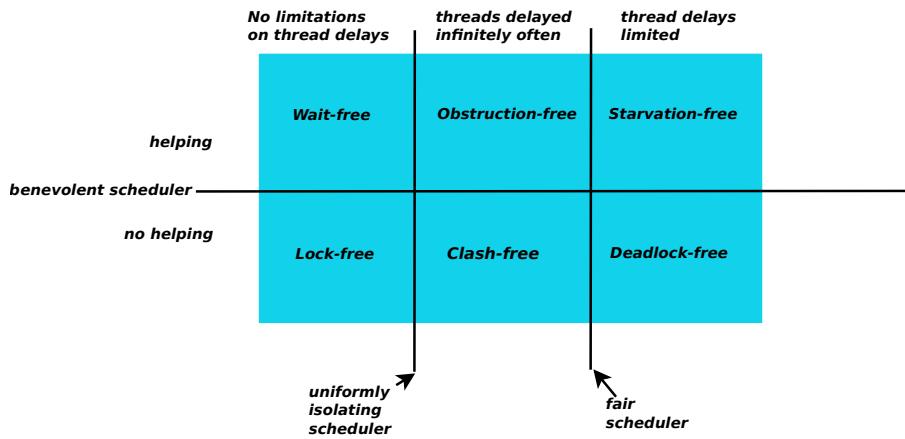


그림 2.15: Non-locking 동기화 기법의 분류

이 바로 종료할 수 있는 알고리즘이다. 다음으로 *Lock-free* 방법은 적어도 하나의 스레드는 딜레이 없이 바로 끝나는 방법이다. 즉 그 이외의 스레드들은 전역 변수를 동시에 접근해서 발생하는 충돌 때문에 다시 순회할 가능성이 있는 알고리즘이다.

이러한 장점을 가진 Non-blocking Algorithm의 한 스택의 예는 그림 2.16과 같이 구현되어 있다. 자료구조의 내용은 value와 다음 노드를 가리키는 포인터가 존재한다(line 4). *push* 함수 같은 경우를 보면, 먼저 새로운 노드에 스택의 *top*에 해당하는 노드를 저장(line 12)하고, CAS 연산을 통해 원자적으로 수정되었는지 체크를 함과 동시에 스택의 *top*에 해당하는 노드를 새로운 노드로 수정한다. 만약 CAS 연산이 실패하였다면(line 13), 즉 다른 스레드가 수정하였다면, 다시 처음부터 앞의 작업으로 이동(line 11) 후 반복하여 수행한다. *pop* 함수는 먼저 스택의 *top*에 해당하는 포인터를 지역 변수에 저장 후(line 20), CAS 연산을 통해 원자적으로 *top* 다음 포인터를 가르키게 한 후(line 21) *top*에 해당하는 노드를 반환한다(line 23). 만약 CAS 연산이 실패하며, *push*처럼 반복 수행한다(line 19).

```

1  struct element {
2      int key;
3      int value;
4      struct element *next;
5  };
6
7  struct element *global;
8
9  void push(struct element *e)
10 {
11     retry:
12     e->next = global;
13     if (cmpxchg(&global, e->next, e) != e->next)
14         goto retry;
15 }
16
17 struct element *pop(void)
18 {
19     retry:
20     struct element *e = global;
21     if (cmpxchg(&global, e, e->next) != e)
22         goto retry;
23     return e;
24 }
```

그림 2.16: 간단한 Non-blocking 스택 알고리즘.

Non-blocking synchronization의 가장 큰 현실적인 문제점은 바로 중간에 메모리를 재사용할 수 있기 때문이다. 예를 들어 스택에 `top->A->B->C` 세 가지 노드가 순차적으로 들어 있을 경우, CPU 1이 `CAS(top, A, B)` 명령어를 수행하고, 동시에 CPU 2가 A와 B를 꺼내고 A, B를 free한 후 다시 A를 스택에 넣으면, CPU 1의 `CAS(top, A, B)` 명령어는 성공하게 되서, 최종으로 `top->B->C` 이 스택에 쌓이게 되고, 처음과 다른 노드 A를 얻게 된다는 것이다. 이러한

상황을 ABA 문제라고 한다. 이러한 문제의 간단한 해결책은 `free()`를 바로 호출하지 않고, 레퍼런스 카운트를 보고 해제하거나, 안전한 시간(모든 프로세서가 작업이 끝날때)까지 기다린 후 호출하는 방법이 있다.

#### 2.4.2 확장성 있는 자료구조

##### Harris Linked List

Non-blocking 알고리즘 중 대표적인 알고리즘 중 하나는 2001년도에 발표된 Harris Linked List이다. Harris 알고리즘은 CAS를 이용한 알고리즘 중 하나이며, 순서대로 정렬된 노드들을 순회하면서 해당 노드의 위치의 오른쪽 노드를 찾은 후 CAS로 해당 오른쪽 노드 앞에 새로운 노드를 삽입하는 방법이다.

그림 2.17는 이러한 Harris 알고리즘 중 삭제하는 방법에 대해서 설명한다. 시간 순서대로 위에서 아래로 진행된다. 제일 위에서는 동시에 CORE1이 노드 B를 삭제하고 CORE2가 노드 C를 삭제한다면, Harris 링크드 리스트에서는 바로 삭제하지 않고 먼저 각 노드에 마킹을 한다. Harris 링크드 리스트 알고리즘에서는 이것을 `logical remove`라고 한다. 다음으로 CORE1과 CORE2가 동시에 CAS 연산으로 삭제를 하면, CORE2의 OLD 값이 CORE1에 의해 변경되었기 때문에 이순간 CAS 실패가 발생하다. 따라서 `logical remove`에 의해 노드는 제거되는 되었지만 아직 노드가 남아 있는 상태가 된다. Harris는 리스트는 CAS가 실패하면 처음부터 다시 순회함으로 마킹된 노드를 제거하게 된다.

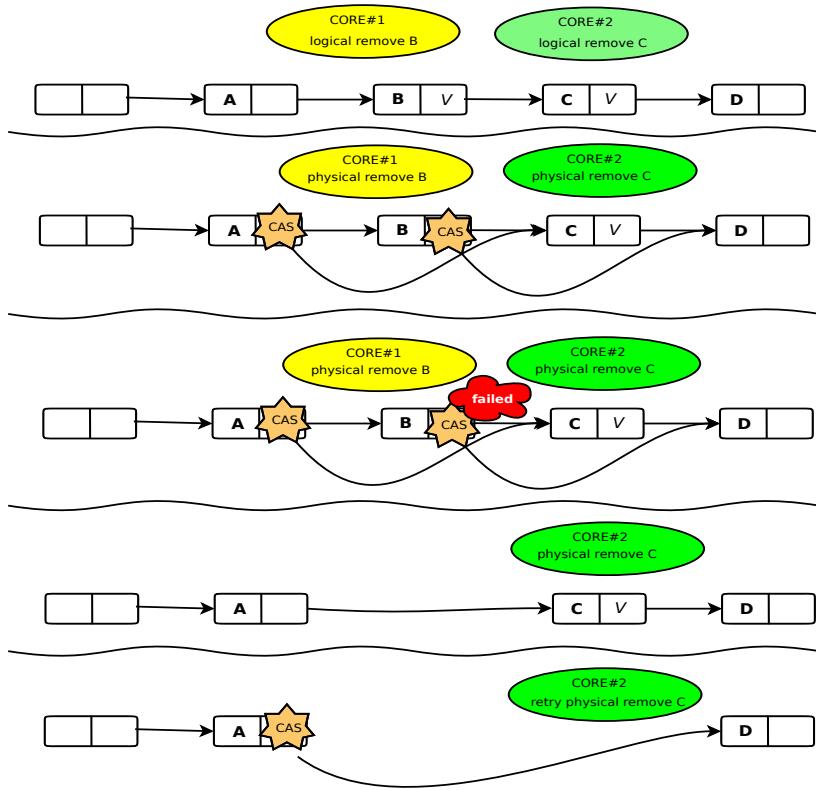


그림 2.17: Harris 삭제

### 제 3 장 논문에서 해결하고자 하는 구체적인 문제

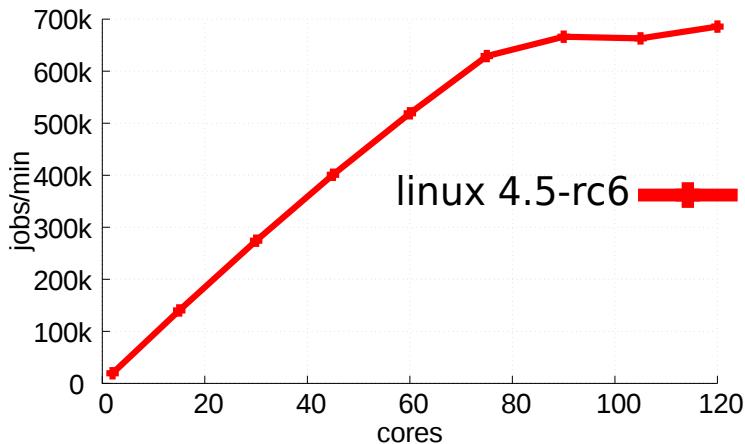


그림 3.1: AIM7-multiuser 성능 확장성

운영체제 커널의 병렬화(parallelism)는 시스템 전체의 병렬화(parallelism)에서 가장 중요하다. 만약에 커널이 확장성이 있지 않으면, 그 위에 동작하는 응용프로그램들도 역시 확장성이 있지 않는다 [23] [13]. 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 [17] [16]. 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 최소화하기 위해

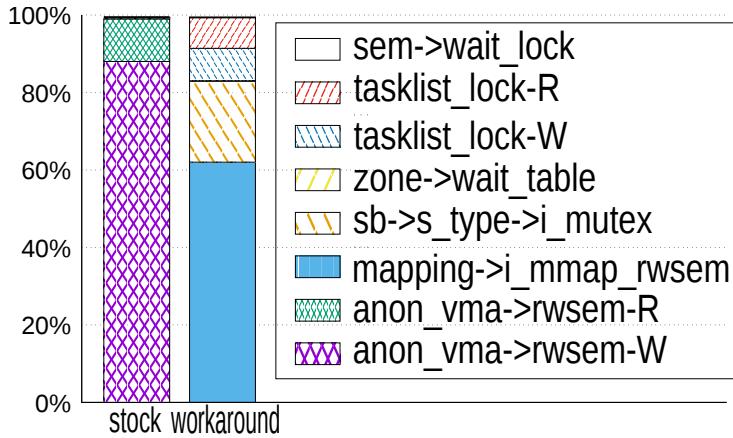


그림 3.2: 120코어에서 락 때문에 기다리는 시간

tempfs(Temp file system)를 사용하였다. 실험 결과 70코어까지 확장성을 가지나 그 이후에는 확장성이 떨어져 완만한 그래프를 보여준다.

우리는 성능 확장성에 근본적인 문제를 분석하기 위해, 문제가 있는 120 코어에서 리눅스의 `lock_stat` [3]를 이용하여 락 경합을 분석하였다. `lock_stat`는 리눅스 커널에 있는 락 프로파일러(profiler)이며, 얼마나 스레드(Thread)가 락을 보유하고 락을 얻기 위해 쓰레드가 얼마나 기다리고 있지를 보여준다. 먼저 멀티 프로세스 기반의 벤치마크인 AIM7을 동작시키고 동시에 120코어 대해서 락 경합을 분석하면 그림 3.2과 같은 결과를 가진다. AIM7 벤치마크의 경우 상당히 많은 부분이 익명 VMA에서 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 `fork`에서

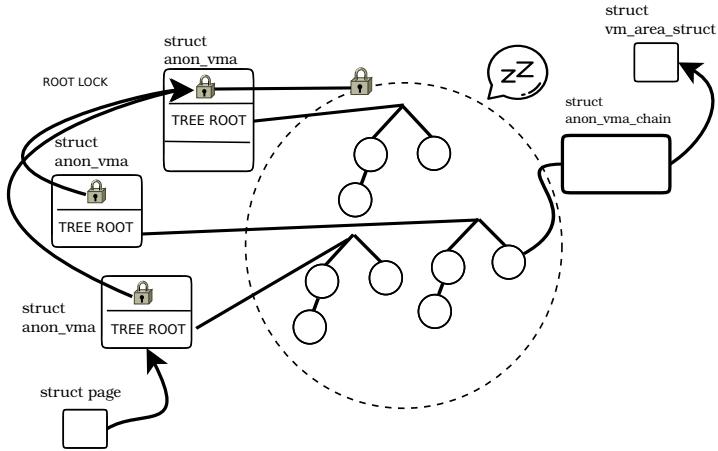


그림 3.3: anonymous 역 매핑의 문제

익명 역 매핑을 호출하는 부분과 읽기와 관련 있는 페이지 스왑(page swap)이 안되도록 하고, 120코어를 대상으로 다시 락 경합을 분석하여 보았다. 이때 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 일으킨다.

이러한 익명 역 페이지 매핑은 리눅스 커뮤니티에서 잘 알려진 락 경합 문제 [8]이고, 파일 페이지 역 매핑에 대한 락 경합 문제는 S.Boyd-Wickizer [15] OpLog 논문을 통해 fork의 확장성 문제의 중요한 원인으로 제시한 부분이다. 즉 두 가지 모두 개선해야지 fork의 성능 확장성이 향상 된다.

이러한 높은 업데이트 비율 때문에 발생하는 업데이트 직렬화 문제에 대한 해결 방법들은 그동안 여러 방법이 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

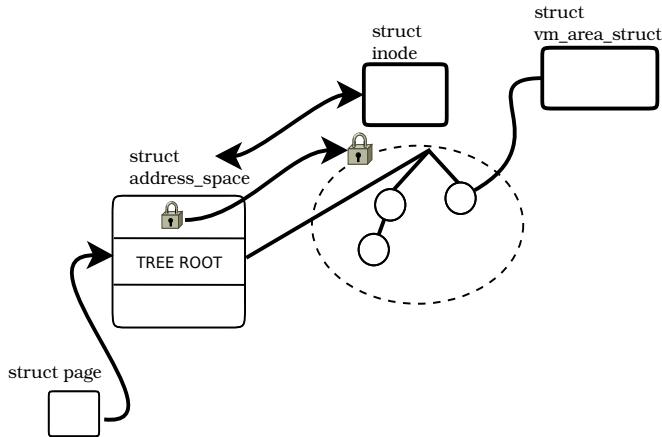


그림 3.4: 파일 역 매핑의 문제

활용하여 동시적으로 업데이트와 읽기 명령어를 수행하게한 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 스레드가 변경을 안 할때까지 같은 일을 수행하는 방법이다. 하지만 이러한 방법도 결국 전역 공유 메모리 주소에 다수의 CAS로 접근하여 병목현상이 생긴다. 이것은 결국 캐시 커뮤니케이션 오버헤드를 만든다 [15]. 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 저장

&anon_vma->rwsem-W:	26070201	26990065	0.06	2004649.76	636301175165.4
&anon_vma->rwsem-R:	358821	569702	0.16	1995132.80	79666356549.93
<b>&amp;anon_vma-&gt;rwsem</b>	<b>8549431</b>	[<fffffff811be9c6>]	unlink_anon_vmas+0x96/0x1d0		
<b>&amp;anon_vma-&gt;rwsem</b>	<b>6022485</b>	[<fffffff811bedae>]	anon_vma_fork+0xde/0x130		
<b>&amp;anon_vma-&gt;rwsem</b>	<b>9424646</b>	[<fffffff811beb93>]	anon_vma_clone+0x93/0x1d0		
&anon_vma->rwsem	2989201	[<fffffff811be4da>]	_put_anon_vma+0x2a/0xa0		
<b>&amp;anon_vma-&gt;rwsem</b>	<b>7487361</b>	[<fffffff811be9c6>]	unlink_anon_vmas+0x96/0x1d0		
&anon_vma->rwsem	12649795	[<fffffff811beb93>]	anon_vma_clone+0x93/0x1d0		
&anon_vma->rwsem	1988738	[<fffffff811bedae>]	anon_vma_fork+0xde/0x130		
&anon_vma->rwsem	1235	[<fffffff811b7a27>]	vma_adjust+0x147/0x7b0		

그림 3.5: 120코어에서의 lock\_stat 결과 분석

하고, 주기적 또는 읽기 명령이 수행하기 전에 그동안 저장된 로그를 수행하는 방법이다. 이러한 로그 기반 방법은 마치 CoW(Copy on Write)와 유사하다. 즉, 읽기 전에 저장된 쌓여있는 로그가 수행됨으로 읽기가 간헐적으로 수행되는 자료구조에 적합한 방법이다.

업데이트 비율이 많은 자료구조를 위한 로그 기반 방법은 총 4가지의 장점을 가진다. 첫째로, 업데이트가 수행하는 시점 즉 로그를 저장하는 순간에는 락이 필요가 없다. 따라서 업데이트를 동시적으로 수행할 수 있을 뿐만 아니라, 락 자체가 가지고 있는 캐시 메모리 오버헤드를 줄일 수 있다. 둘째로, 저장된 순차적인 업데이트 명령을 하나의 코어에서 수행하기 때문에, 캐시 지역성이 높아진다. 셋째로, 큰 수정 없이 기존 여러 데이터(tree, queue) 자료구조에 쉽게 적용할 수 있는 장점이 있다. 마지막으로 저장된 로그를 실제 수행하지 않고, 여러 가지 최적화 방법을 사용하여 적은 명령으로 로그를 줄일 수 있다. LDU도 로그 기반 방법을 따른다. 그러므로 앞에서 설명한 로그 기반 방법의 장점을 모두 가짐과 동시에 업데이트 순간 삭제 가능한 로그를 지원으로 성능이 향상된다.

동기화된 타임스탬프 카운터 기반의 퍼코어 로그를 활용한 동시적 업데이트방법은 결국 타임스탬프 병합 작업을 일으킨다. 특히 코어 수가 늘어 날 경우, 퍼코어 로그를 자료 구조에 적용하는 과정에서 추가적인 순차적인 프로세싱이 요구된다. 이것은 확장성과 성능을 저해한다.

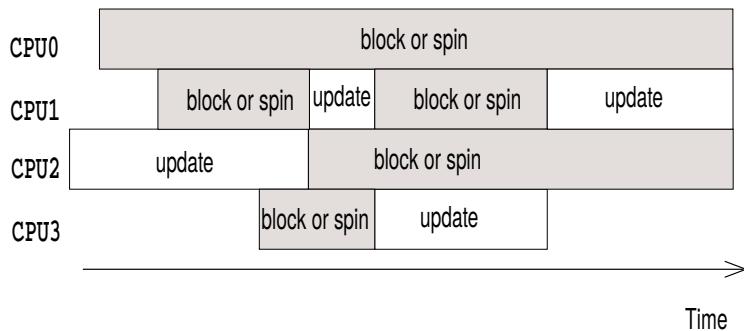


그림 3.6: 업데이트 직렬화의 문제

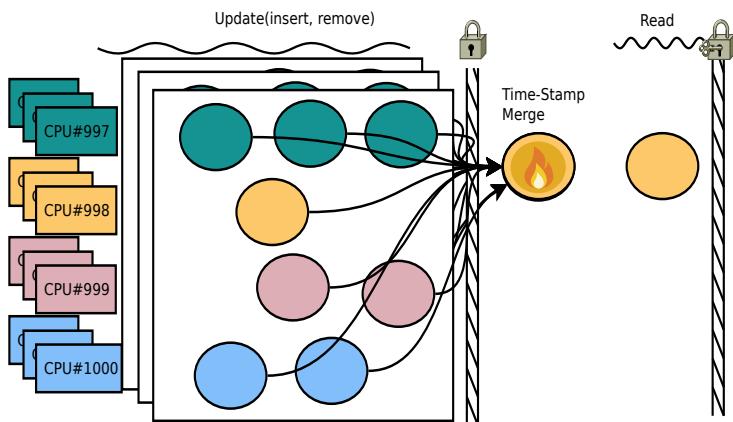


그림 3.7: 동기화된 타임스탬프 카운터 머징에 따른 오버헤드

## 제 4 장 로그기반 동시적 업데이트 방법

### 4.1 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 업데이트 방법은 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 로그 기반 방식의 동시적 업데이트 방법과 원자적 동기화 기능을 최소한으로 이용하도록 설계하였다. 따라서, LDU는 동기화된 타임스탬프 카운터를 사용하는 방식의 타임스탬프를 제거함과 동시에 캐시 커뮤니케이션 오버헤드를 최소화하였다. 이번 장에서는 LDU의 알고리즘적인 디자인 측면에 관해서 설명한다.

#### 4.1.1 접근법

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 수행하여야 한다 [15]. 이러한 시간에 민감한 로그를 더 구체적으로 설명하기 위해 우리는 논문 [23]에서 사용한 심볼 방식으로 설명한다. 우리는 삽입 명령은 원형으로 된 플러스 모양인  $\oplus$ 로 표시하고, 삭제 명령은 원형으로 된 마이너스

모양인  $\ominus$ 으로 표시하고 오브젝트들은 색(**B**(object B))으로 구별하였다. 서로 다른 색깔과 다른 높낮이는 다른 CPU를 의미한다. 예를 들어

$\oplus A, \oplus B, \oplus C, \ominus A, \ominus C, \oplus A, \oplus C, \ominus C$

이것은 5개의 삽입 명령들과 3개의 삭제 명령 그리고 3개의 CPU 그리고 3개의 오브젝트를 의미한다. 이 예제에서  $\oplus A$ 와  $\ominus A$ 는 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,  $\oplus A \ominus A, \oplus C \ominus C$  그리고  $\ominus C \oplus C$ 들은 리더가 수행하기 전에 최소되어도 상관없는 명령어들이다. 따라서 남은 로그인

$\oplus B, \oplus A$

로그들은 시간에 민감하지 않은 로그들이다. 업데이트 명령어가 발생하면 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)과 삭제(remove)가 발생하였으면, 같은 오브젝트에 대해서, 삽입 명령과 삭제 명령에 대한 로그를 업데이트 시점에 바로 삭제하는 방법이다. 동기화된 타임 스탬프 카운터(synchronized 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 해야 한다. 동기화된 타임 스탬프 카운터 방법은 워크로드에 따라, 최적화를 위해 또 다른 순차적 프로세싱이 요구된다. 하지만 LDU는 개별적 오브젝트(individual 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 오브젝트에 삽입-삭제(insert-remove) 명령이 수행될 경우 처음 삽입 명령어는 삽입에 대한 마크 필드에 표시하고 큐(queue)에 저장한다. 다음 삭제 명령부터는 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 field)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 만들어 넣지 않고 기존 로그를 재활용하는 방법이다. 예를 들어 같은 오브젝트에 대해서 삽입-삭제-삽입 (insert-remove-insert) 순서로 업데이트가 수행될 경우, 세 번째 삽입 명령어는 큐에 들어가 있지만 업데이트 시점에 지워진 로그이기 때문에 최소 되어 삽입에 대한 마크 필드가 FALSE로 표시된다. 이런 경우가 가비지 오브젝트 (garbage object)이다. 그러면, 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 증가하는 것을 방지하기 위해 로그를 주기적으로 시간 순서대로 적용한다. 따라서 LDU는 주기적으로 로그를 적용함으로 로그가 쌓여서 발생하는 메모리 낭비

를 줄인다. 이것은 OpLog의 배치 업데이트(batching updates)와 Flat combining의 병합 스레드(combiner thread)와 같이, 1개의 스레드가 업데이트 명령을 수행하므로 캐시 지역성(cache locality)이 높아지는 장점을 가진다.

LDU는 보다 다양한 데이터 구조를 지원하기 위해, 퍼코어 또는 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  $\oplus B$ ,  $\oplus A$ 는 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 증가하고, 메모리 사용량도 퍼코어에 추가로 할당을 받으므로 증가된다. 이를 보완하기 위해 LDU는 전역 큐도 같이 지원한다. LDU의 전역 큐의 장점은 굉장히 간단하여 어떠한 자료구조라도 쉽게 적용할 수 있다. 또한 퍼코어처럼 글로벌 헤드 포인터에 대한 CAS연산을 완전히 제거하지 못하지만, LDU의 업데이트 시점에 지우는 방법과 가비지 로그를 재활용하는 방법으로 글로벌 큐에 로그를 저장하는 횟수를 상당히 줄여서, 캐시 커뮤니케이션 오버헤드를 상당히 줄일 수 있다.

LDU는 논블락킹 큐를 이용하여 로깅한다. 그 이유는 전역 또는 퍼코어 라 없이 수행될 수 있기 때문이다. 논블락킹 큐 중에 LDU는 헤드 포인터에 대한 CAS 연산을 최대한 줄인 다중 생산자 단일 소비자(multiple producers and single consumer)에 활용될 수 있는 자료구조를 이용하였다. 이 큐는 다른 논블락킹 리스트들과 다르게, 삽입 명령을 항상 처음 노드에 삽입함에 따라, CAS가 발생하는 횟수를 상대적으로 줄일 수 있다. 게다가 로그를 적용하는 부분에서 단일 소비자(single consumer)만 고려했기 때문에, 삭제를 위한 복

잡한 알고리즘이 필요 없다. 예를 들어 단일 소비자(single consumer)는 명령로그 전체를 얻기 위해, 스왑 명령을 사용하여 헤드 포인터(head pointer)를 널(NULL)로 원자적으로 제거한다.

### 4.1.2 실행 예

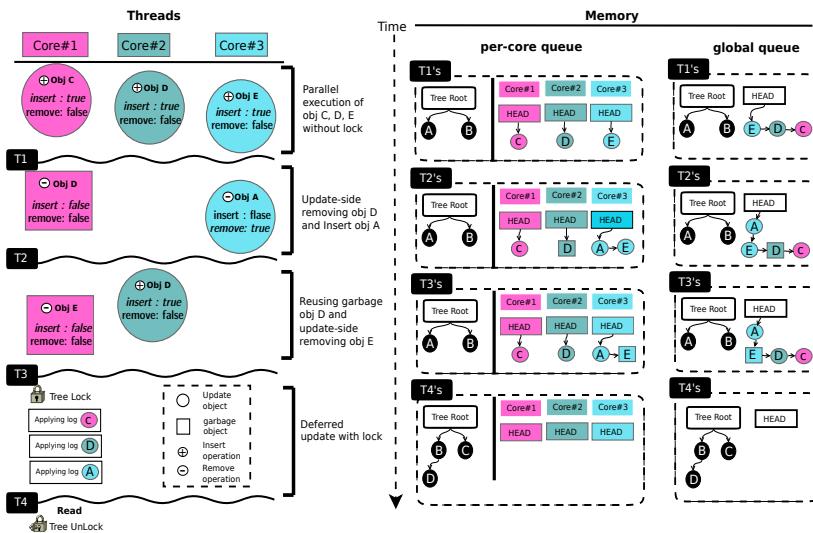


그림 4.1: 7개의 업데이트 명령과 1개의 리드 명령에 대한 LDU 예.

그림 4.1은 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.

$$\oplus \text{C}, \oplus \text{D}, \oplus \text{E}, \ominus \text{D}, \ominus \text{A}, \oplus \text{D}, \ominus \text{E}.$$

- . 우리는 LDU를 설명하기 위해서 앞절에서 사용한 심볼에 사각형 모양의 가비지 심볼인 **D**를 추가하여 설명한다.

이 그림에서 실행 순서는 위에서부터 아래로 이루어진다. 그림 왼쪽에 있는 것은 CPU의 명령어들을 보여주고 오른쪽에 있는 것은 특정한 시간에 메모리에 있는 자료구조의 내용에 대해서 보여준다. 초기 트리 자료구조에는 오브젝트 **A**와 **B**가 들어 있고 큐는 비어 있다. 그림의 위쪽에는 Core1, Core2 그리고 Core3은 동시적 업데이트 명령을 수행한다. 따라서  $\ominus C$ ,  $\ominus D$  그리고  $\ominus E$ 는 락이 없이 동시적으로 수행된다.

LDU는 명령어 로그를 저장하기 위해 논블락킹 큐를 사용하기 때문에, 이 작업에는 업데이트 락이 필요 없다. 따라서 모든 쓰레드는 락에 대한 경쟁 없이 동시적으로 수행 가능하다. T1 시점이 되면, 트리는 오브젝트 **A**과 **B**이 존재한다. 그리고 퍼코어 큐와 전역 큐는 오브젝트  $\ominus C$ ,  $\ominus D$  그리고  $\ominus E$ 가 보관되고, 퍼코어 큐는 퍼코어 메모리에 각각 구별되어 저장된다.

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

마지막 명령어들은  $\oplus D$ ,  $\ominus E$ 이다. LDU는 원자적 스왑을 사용하여 큐에 있는 로그를 새롭게 생성하지 않고 다시 재활용한다. 그리므로, 업데이트 측면에서 지우는 기술로 오브젝트 **D**는 **D**로 바뀌고, 그리고 오브젝트 **E**는 **E**로 바뀐다. T3 시점이 되면, 퍼코어 큐에는  $\oplus C$ ,  $\oplus D$ ,  $\ominus A$  그리고  $\oplus E$ 가 저장된다. 리드 함수가 수행되기 전에 이것은 트리의 명령어를 보호하기 위해서 상호배제 기반의 원본 트리의 락이 필요하다. LDU는 큐에서 트리로 각각의 명령어를 옮긴다. 이 때 옮길 오브젝트는 마크가 표시된 오브젝트 로그들이다. 그러므로, 명령어  $\oplus C$ ,  $\oplus D$  그리고  $\ominus A$  들은 가비지 로그인  $\oplus E$ 를 제외하고

옮겨진다. T5시점이 되면, 트리는 **B**, **C** 그리고 **D**를 가지게 되어, 최종적으로 리더는 결국 일관성있는 같은 데이터를 읽게 된다.

```
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 4.2: LDU의 동시적 삽입에 대한 알고리즘.

```
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 4.3: LDU의 동시적 삭제에 대한 알고리즘.

### 4.1.3 알고리즘

이번 장은 알고리즘의 중요한 부분에 대해서 설명한다. 알고리즘은 로그를 삽입하는 부분과 로그를 적용하는 부분에 대해서만 설명한다.

#### 로그 삽입

그림 4.3은 동시적 업데이트를 수행하는 함수에 대해 보여준다. 동시적 업데이트 함수는 3가지 단계로 구분된다. 첫째 단계는 체크 단계이며, 입력으로 받은 오브젝트가 취소 가능한 오브젝트인지 확인을 한다(Line 4, 20). 이 코드가 수행되면, 동시에 `synchronize` 함수는 리더 또는 주기적인 함수에 의해 호출된다. 따라서 첫 번째 단계에서는 원자적 명령어가 필요하다. 만약 그에 상응하는 마크 필드가 TRUE라면 그것에 대한 마크 필드는 FALSE로 수정된다. 두 번째 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 명령이 발생하면, 다음 명령은 반드시 삭제 명령이 발생한다. 왜냐하면, 리눅스 업데이트 함수는 검색(search), 동적할당(alloc) 그리고 해제(free) 힘들들과 구별되어 있기 때문이다. 리눅스 커널의 삭제-삭제 순서 또는 삽입-삽입 명령 순서는 금지된다. 만약에 삭제-삭제 명령어 순서가 발생하면, 두 번째 삭제 명령어는 크래쉬(crash) 만날 수 있다. 그 이유는 첫 번째 삭제 명령어 다음에 이 오브젝트는 바로 동시에 메모리 해제가 될 수 있기 때문이다. 따라서 우리는 그에 상응하는 마크 필드를 점검하였다(Line 5, 21).

## 로그 적용

```
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 4.4: 로그를 적용하는 알고리즘.

그림 4.8은 명령어 로그들을 적용하는 자연 업데이트 함수를 보여준다. `synchronize` 함수는 리드 전에 호출되거나 주기적으로 호출되는 타이머 핸들러(timer handler)에 의해 호출된다. 그 이유는 무한정 커지는 로그를 방지하기 위해서이다. `synchronize` 함수가 수행하기 전에, `synchronize` 함수는 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 볼 수 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 한 번 더 마크 필드가 적용하는 과정(Line 8)과 가비지 비트를 초기화하는 과정(Line 10)에서 수정되었는지 다시 체크를 한다(Line 12,13).

## 4.2 리눅스 커널에 적용

이번 장은 리눅스 커널의 업데이트 직렬화에 대한 문제를 풀기 위해, 우리가 어떻게 LUD를 복잡한 리눅스 가상 메모리 시스템에 적용했는지에 대해 설명한다. 이번 장은 더욱 현실적인 내용을 다룬다.

리눅스 역 매핑(rmap)은 커널 메모리 관리 메커니즘(mechanism) 중 하나이다. 이것은 익명 역 매핑과 파일 역 매핑으로 구성되며 이것은 업데이트 비율이 높은 자료구조이다. 이러한 두 가지 rmap은 리눅스의 가상 주소(VMAs)를 관리하고, 이것은 물리 주소(physical address)를 가상 주소로 변환하는 일을 하며, rmap은 프로세스 간 공유하는 전역 자원(global resource)이다. 이러한 전역 자원인 rmap은 인터벌 트리(interval tree)에 의해 관리된다. 이러한 공유 자원을 보호하기 위해, 리눅스는 읽기-쓰리 세마포어(reader-writer semaphore)를 이용하고, 동시적으로 많은 프로세스가 생성되면 이것은 결국 병목 현상으로 된다. 그 이유는 rmap의 업데이트 명령어들이 병렬로 실행되지 못할 뿐만 아니라 락 자체가 캐시 커뮤니케이션 오버헤드를 가져오기 때문이다.

이와 반대로, ramp은 물리 페이지(physical page)가 디스크로 옮겨질 때와 다른 CPU로 옮겨질 때 그리고 파일 사이즈를 줄일 때 간헐적으로 인터벌 트리를 읽는다.

### 4.2.1 익명 역 매핑

그림 4.5은 익명 rmap에 대한 자료구조를 보여준다. 프로세스가 자식을 만들 때, 부모의 익명 메모리 체인(AVC)은 자식에게 복사한다. 그리고 새로운 익명 가상메모리(struct anon\_vma)는 생성이 된다. 프로세스가 동시적으로 자식 프로세스를 만들 때, 더 복잡한 익명 ramp의 자료구조는 생성된다. 또한 익명 rmap은 리눅스 커널에서 복잡한 자료구조 중 하나이다 [25]. 익명 ramp

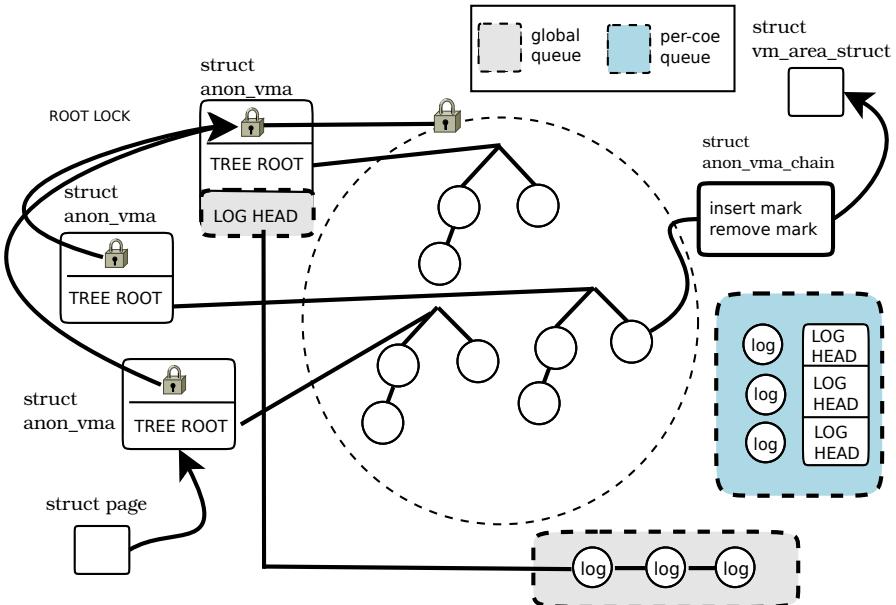


그림 4.5: 리눅스 익명 역 매핑에 LDU를 적용한 그림.

은 자식 프로세스간 AVCs를 공유하기 때문에 루트(root)의 락을 사용한다. 따라서 이러한 루트 락은 락 경합 문제를 일으킨다 [8].

락 경합에 대한 문제점을 제거하기 위해서, 우리는 개별적인 오브젝트 (**struct anon\_vma**)에 삽입과 삭제에 대한 마크 필드를 추가하였다. 그리고 우리는 업데이트 측면에서 지우는 기술을 구현하였다. 로그 큐 헤더에 대한 위치를 이해하는 것은 중요하다. 앞에서 설명한 봐와 같이, 익명 rmap은 루트의 락을 사용하기 때문에, 퍼코어 큐 버전의 LDU는 퍼코어 메모리에 루트의 정보와 함께 저장하거나, 글로벌 큐의 경우에는 루트 자료구조(**(struct anon\_vma)**)에 저장한다. 결론적으로, LDU는 원본 자료구조를 많이 수정하지 않는다. 이것은 LDU가 왜 가벼운 방법인지에 대해서 보여준다.

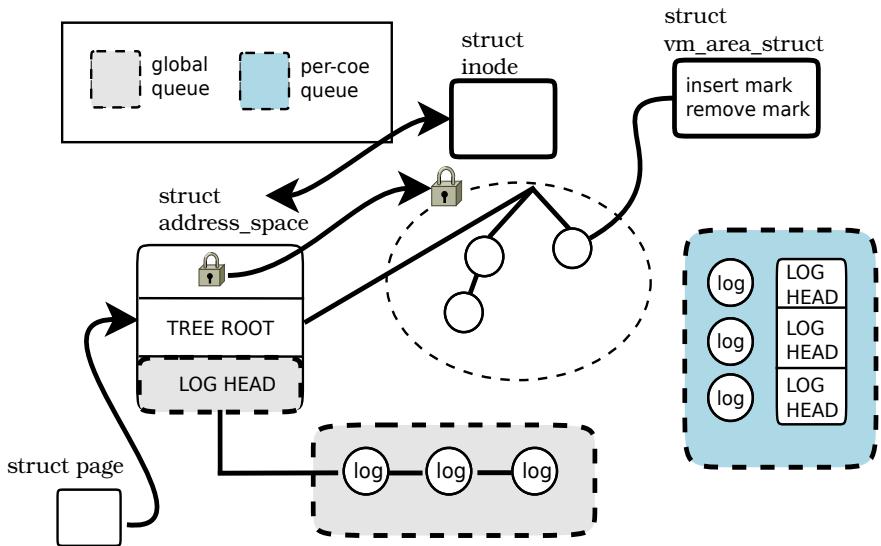


그림 4.6: 리눅스 파일 역 매핑에 LDU를 적용한 그림.

#### 4.2.2 파일 역 매핑

그림 4.6은 파일에 대한 rmap을 보여준다. 물리적 주소를 가상 주소로 변경하기 위해서, 페이지는 address space 오브젝트(`struct address_space`)를 가리키며, address space 오브젝트는 인터벌 트리를 이용하여 VMAs를 관리한다. 이러한 인터벌 트리는 프로세스간 공유하는 자원이다. `fork()` 그리고 `exit()`과 같은 시스템 콜은 VMAs에 대해서 동시적 업데이트를 수반하므로, 프로세스들이 동시에 많은 시스템 콜을 호출할 때, 파일 rmap은 업데이트 명령어 때문에 직렬화가 된다.

LDU는 파일 rmap 자료구조에 쉽게 적용될 수 있다. 예를 들어, LDU를 사용하기 위해서는 개발자는 로그의 큐 헤더를 퍼코어 메모리에 저장하던지,

아니면 원본 자료구조(`struct address_space`)에 저장하면 된다. 그리고 각각의 오브젝트(`struct vm_area_struct`)에 마크필드를 추가하면 된다.

그리고 개발자는 락 없이 업데이트 함수를 로깅 함수로 수정한다. 마지막으로 개발자는 `synchronize` 함수를 만들고, 이것을 리드 전에 호출되도록 수정한다.

이 그림은 왜 LDU가 추가적으로 전역 큐를 사용하는지를 보여준다. 이것은 로그의 헤드 포인터가 인터벌 트리의 자료구조에 위치하기 때문이다. 반면에, 퍼코어 큐는 헤드에 대한 메모리의 위치가 다르므로, 추가적인 퍼코어 큐에 대한 관리 방법이 필요하다.

```
1 bool insert_log_global_queue(struct obj_root *root,
2                             struct ldu_node *log)
3 {
4     ...
5     do {
6         first = head->first;
7         new_last->next = first;
8     } while (CMP_AND_SWAP(&head->first, first, new_first) != first);
9     ...
10 }
```

그림 4.7: LDU의 전역 큐.

#### 4.2.3 자세한 구현 내용

퍼코어 큐를 위한 로그의 헤드 포인트 위치가 큐는 원본 자료구조와 분리되어 있으므로, 퍼코어 큐의 구현은 퍼코어 해시 테이블(per-core hash table) 방법을 이용하였다. 이것은 각각의 오브젝트들을 구분 할 수 있게 해준다. 퍼코어 해쉬 테이블은 직접 사상 캐시(direct-mapped cache)를 구현하였다. 이것은

하나의 해시 버켓(bucket)에는 오직 하나의 오브젝트만 존재하는 방법이다. 그 이유는 최근에 사용된 오브젝트가 다시 사용될 확률이 크기 때문이며, 이것은 OpLog의 퍼코어 해시 테이블 방법과 비슷하다. 만약 해쉬 테이블이 충돌을 만나면, LDU는 충돌난 오브젝트를 해쉬 슬롯에서부터 내보낸다. 더욱이, 이러한 방법은 프로그래머의 추가적인 작업을 줄인다. 그 이유는 이것은 코드의 수정을 최소화할 수 있고 추가적인 락이 필요가 없기 때문이다. 하지만 퍼코어 해시 테이블은 해시 충돌에 따른 오버헤드를 낳는다. 이러한 방법은 파일 rmap(`struct address_space`)과 같이 작은 빈도수로 루트 오브젝트가 생성될 때 효율적이다. 반면에, 익명 ramp은 극심하게 루트 오브젝트(`(struct anon_vma)`들을 생성하기 때문에, 이것은 심한 해쉬 충돌 오버헤드를 발생한다. 그러므로, 익명 rmap과 같은 경우에는 우리는 오브젝트의 헤더를 구별하지 않았다. 그러나 이것은 추가적인 프로그래머의 작업과 전역 락이 필요하다.

우리는 새로운 지역 업데이트 알고리즘을 리눅스 4.5-rc6 커널에 구현하였고, 우리의 수정된 리눅스는 오픈소스로 이용할 수 있다. 우리의 구현은 충분히 안정적이며, 리눅스의 테스트 프로젝트(Linux Test Project) [2] 중에서 가상 메모리, 스케줄러 그리고 파일과 관련된 테스트를 모두 통과하였다.

```

1  bool insert_log_per_core_queue(struct obj_root *root,
2      struct ldu_node *log) {
3      slot = &get_cpu_var(obj_root_slot);
4      p = &slot->obj[hash_ptr(root, HASH_ORDER)];
5      empty = p->list.first;
6      // is empty list?
7      if (!empty) {
8          ldu = log_entry(first, struct ldu, ll_node);
9          // is hash complct?
10         if (ldu->root != dnode->root) {
11             lock = ldu->lock;
12             entry = SWAP(&p->list->head->first, NULL);
13             insert_queue();
14             // flush log as a direct mapped cache
15             object_lock(&lock);
16             synchronize_ldu(entry);
17             object_unlock(&lock);
18             goto out;
19         }
20     }
21     do { first = head->first; log->->next = first;
22     } while (CMP_AND_SWAP(&head->first, first, new_first) != first);
23
24     out:
25 }

```

그림 4.8: LDU의 퍼코어 큐.

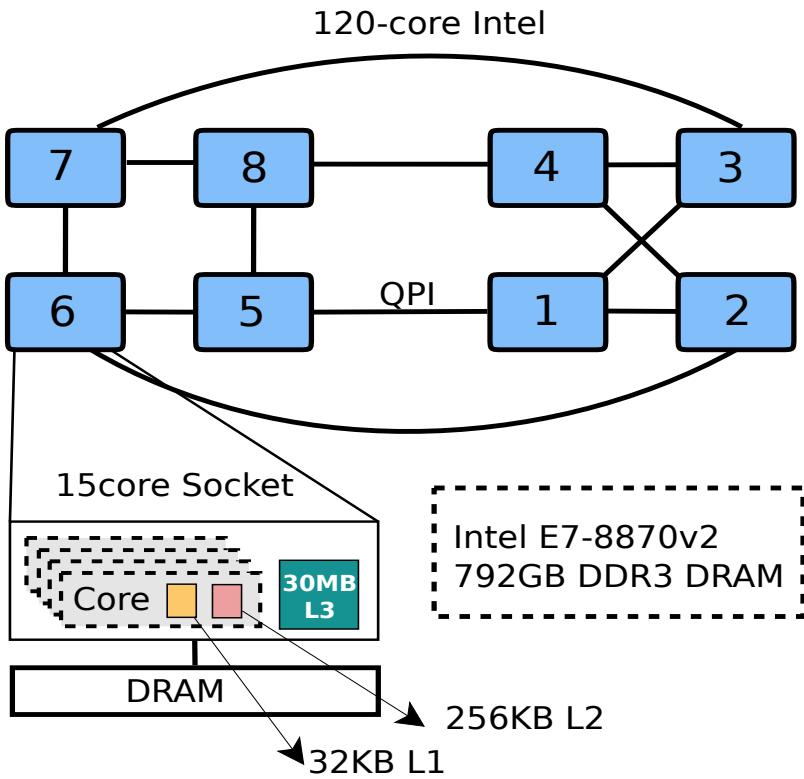


그림 4.9: 실험 환경.

## 4.3 평가

### 4.3.1 실험 환경

우리가 제안한 LDU 기법에 대해서 평가를 하기 위해, 우리는 리눅스 커널에 LDU를 적용하여 비교하였다. 비교 대상으로는 수정하지 않은 리눅스 커널과 Harris의 Lock-free 리스트 [32]를 구현하여 비교 실험을 하였다. Harris

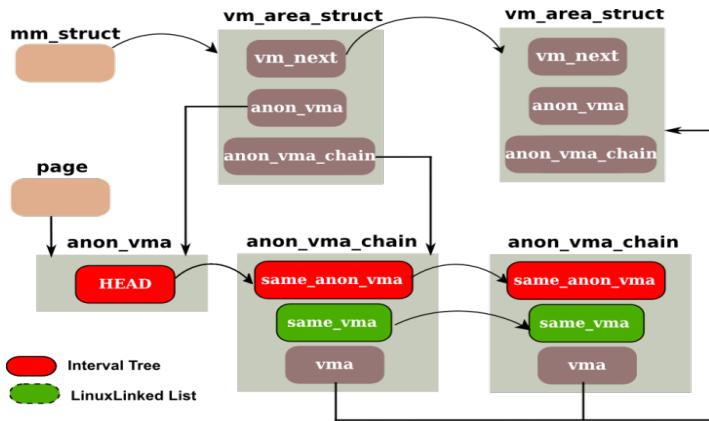


그림 4.10: lockfree 리스트로 변경하기 전 자료구조

알고리즘을 사용한 이유는 논블락킹 알고리즘 중에서 대표하는 알고리즘이기 때문이다. Harris의 링크드 리스트의 기본적인 알고리즘은 sysnchrobench [31]과 ASCYLIB [26]에서 구현된 내용을 사용했으며, 우리는 이러한 Harris 링크드 리스트를 리눅스 커널에 적용하였다.

실험을 위해 우리가 사용한 하드웨어 명세서는 8소켓으로 구성된 120 코어 시스템을 사용했으며, 각각의 코어는 인텔 E7-8870 chips(15 cores per socket)을 사용했다. 메모리는 792기가 바이트 DDR3 DRAM을 사용하였다. 그림 4.9은 우리가 사용한 시스템을 보여준다.

우리는 리눅스 fork에 집약적인 응용프로그램과 업데이트 비율이 많은 자료구조가 우리가 제안한 방법에 효율적이기 때문에 fork에 집약적인 벤치마크를 선택하였다. 이러한 벤치마크 프로그램들은 리눅스의 확장성 벤치마크인 AIM7, 그리고 MOSBENCH에서 이메일 서버 벤치마크인 Exim 그리고 마이크로 벤치마크인 Lmbench이다. 이러한 워크로드들은 두가지 역 매핑 때문에 높은 락 경합을 보여준다. 더욱이, AIM7 벤치마크는 리눅스 커뮤니티에서

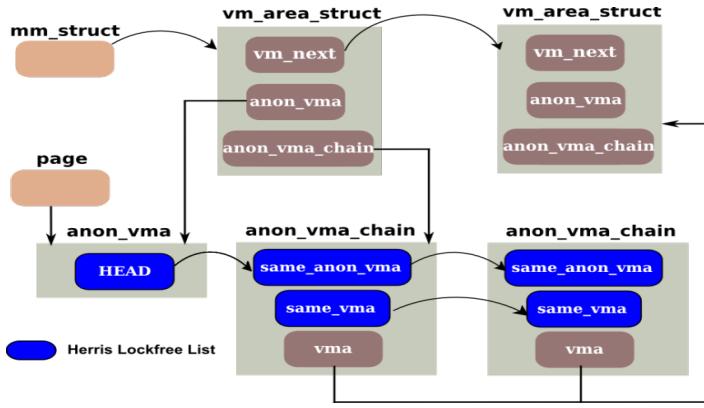


그림 4.11: lockfree 리스트로 변경 후 자료구조

리눅스 커널에 대한 테스팅 뿐만 아니라 확장성을 향상 시키기 위해 많이 사용되는 벤치마크이다. Exim은 실제 사용되고 있는 응용프로그램이다. 하지만 이것 역시 리눅스 fork 때문에 성능에 병목 문제가 생긴다. 마지막으로 우리는 fork에 대한 성능과 확장성에 대해서 보기위해 우리는 Lmbench를 선택하였다.

우리는 비교를 위해 4가지 다른 설정으로 실험하였다. 첫째로, 우리는 기본 점수를 보기 위해, 수정없는 리눅스 커널(stock linux)을 사용하였다. 둘째로, 우리는 전역 큐 버전의 LDU를 사용해서 실험하였다. 다음으로, 우리는 페코어 버전의 LDU를 사용하여 실험하였다. 마지막으로 우리는 앞에서 설명한 Harris의 lock-free 링크드 리스트 버전의 리눅스 커널을 대상으로 실험하였다. 안타깝게도, LDU와 OpLog를 몇몇의 구현 관련 이슈(우리는 OpLog의 자세한 구현을 얻지 못하였다.)로 인해 바로 비교하지 못하였다.

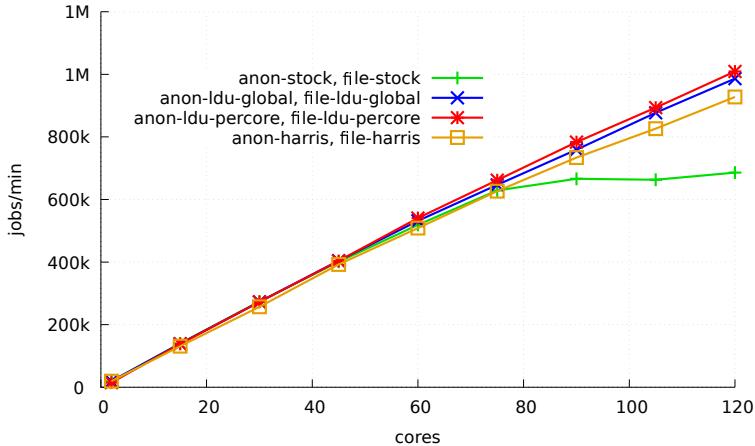


그림 4.12: AIM7-multiuser 확장성.

### 4.3.2 AIM7

우리는 AIM7-multiuser를 사용하였다. 이것은 AIM7의 워크로드 중 리눅스 fork에 집중된 벤치마크이다. 이러한 multiuser 워크로드는 동시에 많은 프로세스를 생성 한 후 다양한 일을 수행한다(see section ??). 또한, 우리는 파일 시스템에 대한 병목현상을 줄이기 위해 리눅스 temp 파일 시스템을 사용하였다. 우리는 코어 수에 비례하여 user들을 증가하였다.

AIM7-multiuser에 대한 실험 결과는 그림 4.12과 같다. 75코어 전 까지는 수정 안한 리눅스는 확장성이 일정하나 그 이후에는 직렬화된 업데이트들 때문에 병목현상이 생긴다. 하지만, 120코어까지 Harris 링크드 리스트와 우리의 LDU는 확장성이 있다. 그 이유는 워크로드들이 업데이트 명령어와 읽기-쓰기 세마포어(anon\_vma->rwsem, mapping->i\_mmap\_rwsem) 없이 동시적으로 실행되기 때문이다. LDU의 퍼코어 큐버전은 가장 좋은 성능을 보여주고, 확장성도 뛰어나며, 수정 안 한 리눅스에 비해 1.5배 빠르고 Harris보다 1.1x 빠른다.

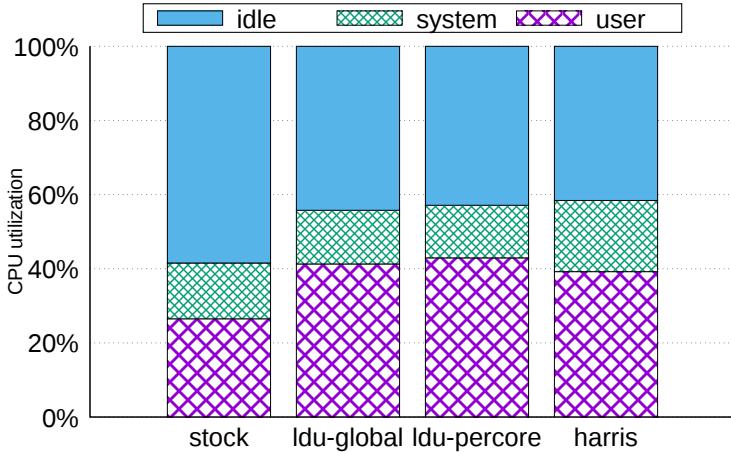


그림 4.13: 120코어에서 AIM7 CPU 사용량.

게다가, 비록 LDU의 전역 큐버전은 전역 CAS 명령어를 실행하지만, 이 방법 역시 높은 성능과 확장성을 가진다. 그 이유는 LDU의 2가지 기법 때문에 전역 CAS에 대한 접근이 완화되었기 때문이다. 이것은 퍼코어 큐 버전에 비해 2% 성능 저하가 생긴다. 더욱이, 수정 안 한 리눅스는 가장 높은 유류(IDLE) 시간(56%)을 가진다(그림 4.13). 그 이유는 2가지 세마포어(i.e., `anon_vma->rwsem`, `mapping->i_mmap_rwsem`)를 얻기 위해 기다리기 때문이다. 놀랍게도 비록 2가지 LDU가 Harris 커널 버전보다 높은 유류시간을 가지지만, 처리량은 Harris 방법보다 더 높다. 이것은 바로 우리의 효율적인 알고리즘 때문이다.

### 4.3.3 Exim

Exim의 성능에 대한 확장성을 측정하기 위하여, 우리는 매니코어 확장성 벤치마크 중 하나인 MOSBENCH를 이용하였다. 이메일(E-mail) 서버인 Exim의 디자인은 확장성이 있게 설계되었다. 그 이유는 Exim의 메시지(message)

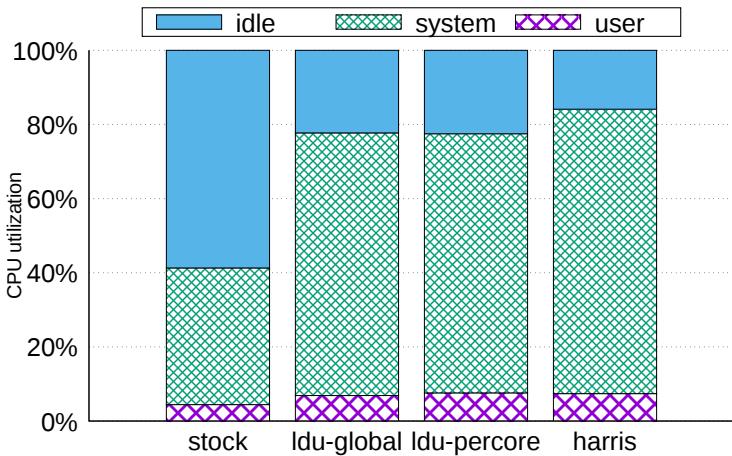


그림 4.14: 120코어에서 EXIM CPU 사용량.

전달자(delivers)는 리눅스의 프로세스를 사용하여 병렬적인 방법을 사용하여 메시지를 메일 박스에 전달한다. 이러한 Exim은 Fork가 많이 발생하는 워크로드 중 하나이다. 클라이언트는 같은 장치에서 실행되고, 각각의 클라인언트는 메일 파일에 대해서 충돌을 막기 위해 다른 여려 유저에게 보낸다 Exim은 파일 시스템에서 병목이 발생한다 [14]. 그 이유는 메시지의 바디(body)는 각각의 유저 메일 파일에 추가되기 때문이다. 따라서 우리는 파일 시스템의 병목 지점을 제거하기 위해 분할된 tmpfs를 사용하였다.

그림 4.16에서 보여주는 Exim의 결과는 수정안한 리눅스 커널의 Exim은 60코어 까지 확장성이 좋게 동작을 하지만 그 60코어 근처부터 성능이 떨어지는 모습을 볼 수 있다. Harris와 우리의 LDU는 105코어까지 성능향상이 보인다. 그 이유는 이 두 방법은 세마포어 때문에 기다리는 현상이 없이 동시에 업데이트가 명령이 수행할 수 있기 때문이다. LDU의 퍼코어 큐 버전은 보다 더 좋은 성능을 가진다. 그 이유는 이것은 캐시 일관성과 관련한 오버헤드를 줄였기 때문이다. 이것은 수정 안한 리눅스보다 2.6배 성능향상을 가지며 Harris 보다 1.2배의 성능 향상을 한다. 비록 우리의 확장성이 있는 기술을 적

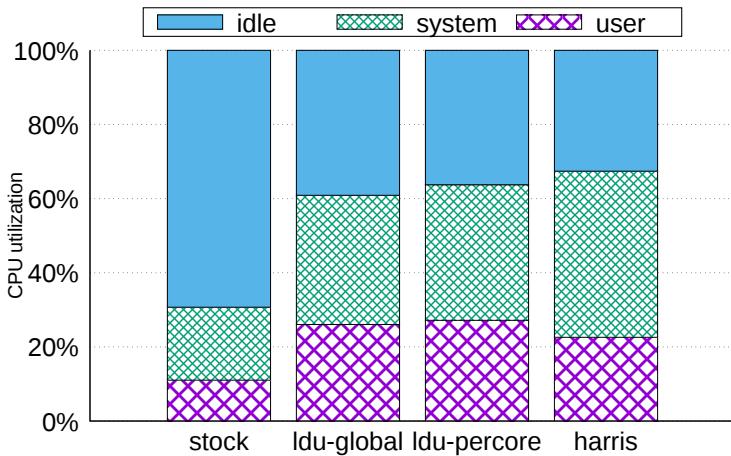


그림 4.15: 120코어에서 Lmbench CPU 사용량.

용하였지만, Exim은 105코어부터 성능 확장성에 대해서 문제가 생긴다. 그 이유는 Exim의 프로세스들은 상대적으로 큰 크기의 가상메모리를 사용하기 때문이다. 이것은 결국 프로세스가 종료될 때 가상메모리에 대한 초기화 오버헤드를 낳으며, 많은 소프트 페이지 폴트(soft page fault)를 야기 시킨다. 이것은 NUMA 구조와 같은 원격 메모리 접근일 때보다 많은 오버헤드가 발생한다. Harris 링크드 리스트는 15%의 유휴 시간을 가진 반면, 퍼코어 큐 버전의 LDU는 22%의 유휴 시간을 가진다. 그 이유는 LUD의 효율적인 알고르즘 때문이다 (그림 4.14)..

#### 4.3.4 Lmbench

Lmbench는 내부에 프로세스 관리에 대한 워크로드를 포함하여 다양한 마이크로(micro) 벤치마크를 포함하고 있다. 우리는 이러한 다양한 마이크로 벤치마크 중 프로세스 관리에 대한 워크로드를 사용하였다. 이러한 워크로드는 기본적인 프로세스 관리에 대한 요소들 예를 들어 프로세스 생성(creating

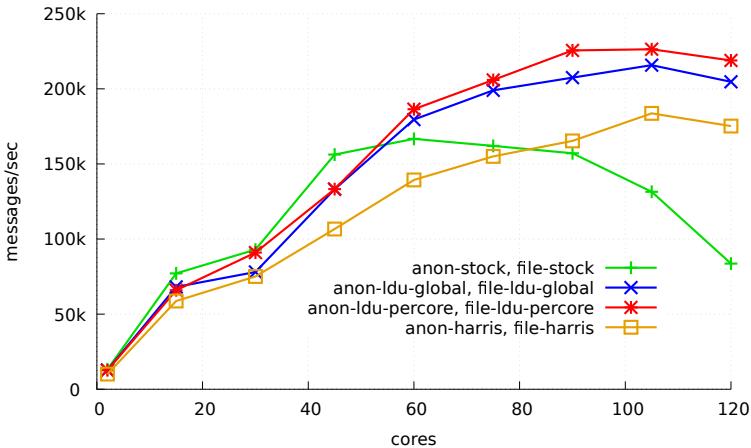


그림 4.16: Exim 확장성.

a new process), 프로그램 시작(running a program) 그리고 문맥교환(context switching)들을 측정한다. 그리고 우리는 프로세스 생성에 대한 워크로드를 병렬 옵션(값은 1000)과 함께 설정하여 수행하였다.

Lmbench의 결과는 그림 4.17에서 보여주며, 세로축에 대한 결과는 실행 시간이다. 45코어까지, 수정 안 한 리눅스 커널은 일정한 확장성을 보이며, 그 후 실행시간을 늘어난다. 퍼코어 버전의 LDU는 120코어에서 수정 안 한 리눅스 커널에 2.7배 성능향상으로 보이며, Harris 리스트에 비해 1.1배의 성능향상을 보인다. 수정 안 한 리눅스는 69%의 유휴 시간을 가진반면 다른 방법들은 약 35% 정도의 유휴시간을 가진다. 그 이유는 수정 안한 리눅스 커널은 2가지 ramp 세마포어(anon\_vma->rwsem, mapping->immap\_rwsem)(figure 4.15)를 기다리기 때문이다. 사실, 우리의 LDU에 대한 개발 동기는 120코어에서 성능과 확장성을 개선하는 것이다. 따라서 우리는 적은 코어(30코어 이내)에서의 성능은 고려하지 않았었다 하지만 30코어까지 우리의 LDU는 수정 안한 리눅스 커널과 비슷한 성능을 보인다. 반면, Harris 링크드 리스트는 60 코어 까지 안 좋은 성능을 보여준다.

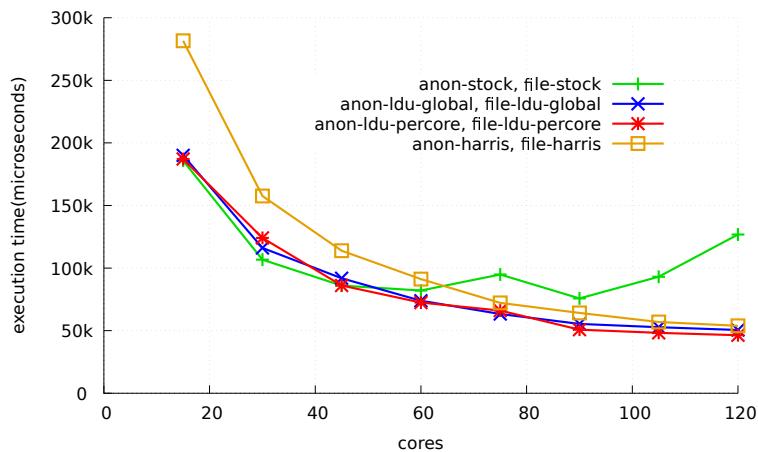


그림 4.17: Lmbench의 프로세스 관리 벤치마크에 대한 실행시간.

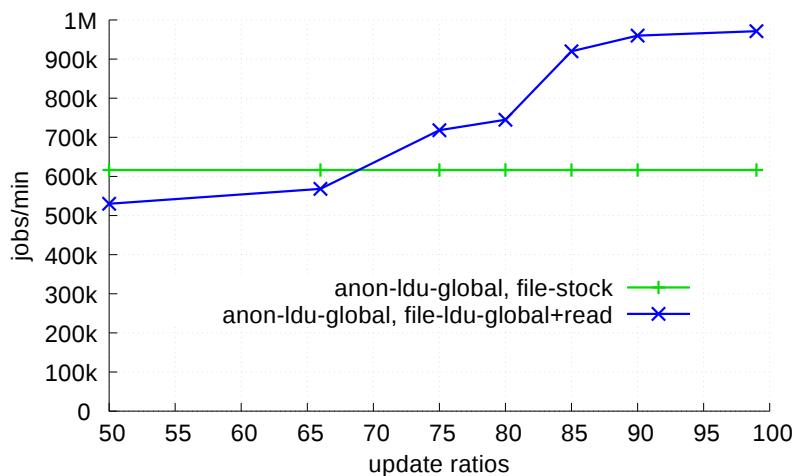


그림 4.18: 업데이트 비율에 따른 AIM7 성능.

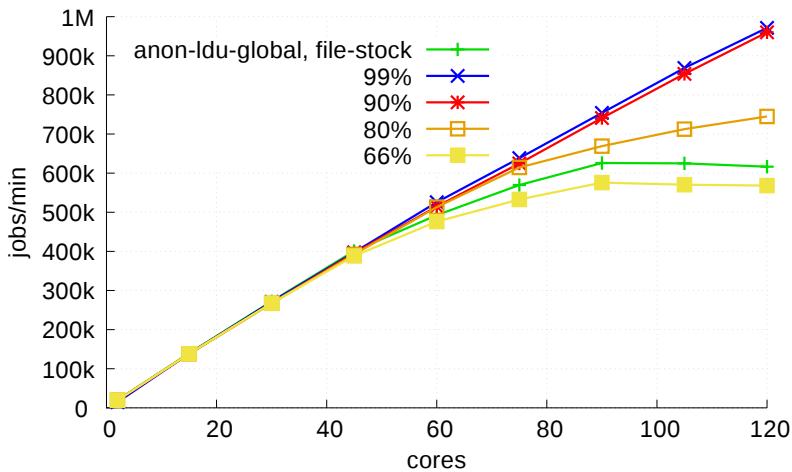


그림 4.19: 업데이트 비율에 따른 AIM7 확장성.

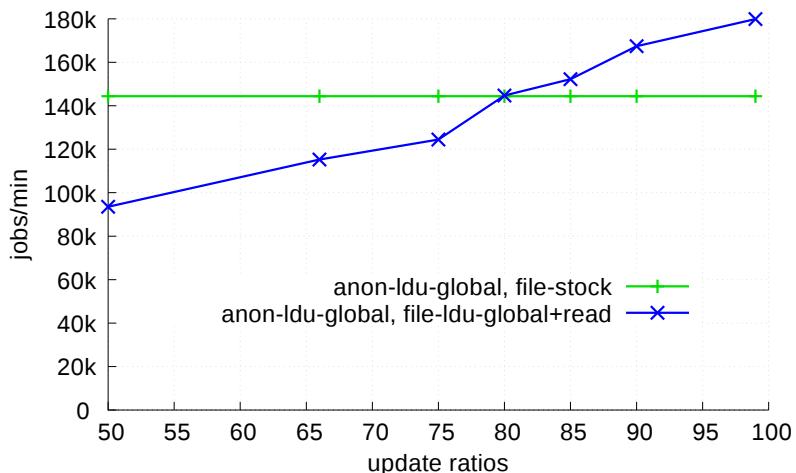


그림 4.20: 업데이트 비율에 따른 Exim 성능.

### 4.3.5 업데이트 비율

우리가 제안하는 LDU에서의 하나의 의문사항은 바로 읽기 명령이 자주 발생할 경우는 성능에 대한 확장성이 어떻게 되는가? 이다. 그 이유는 제안하

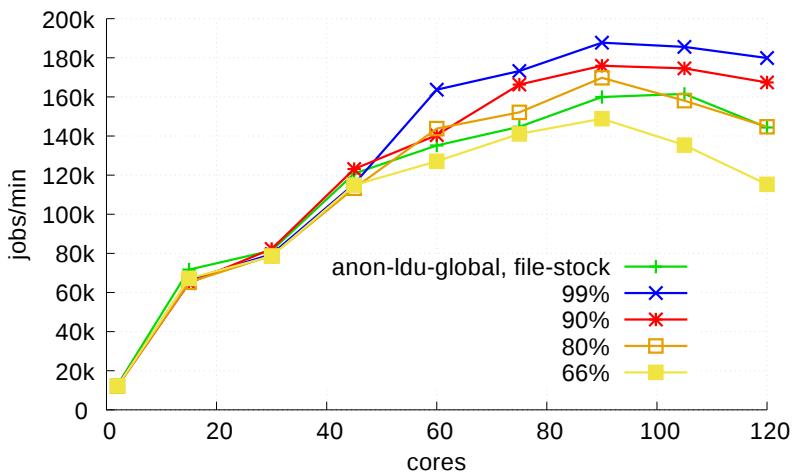


그림 4.21: 업데이트 비율에 따른 Exim 확장성.

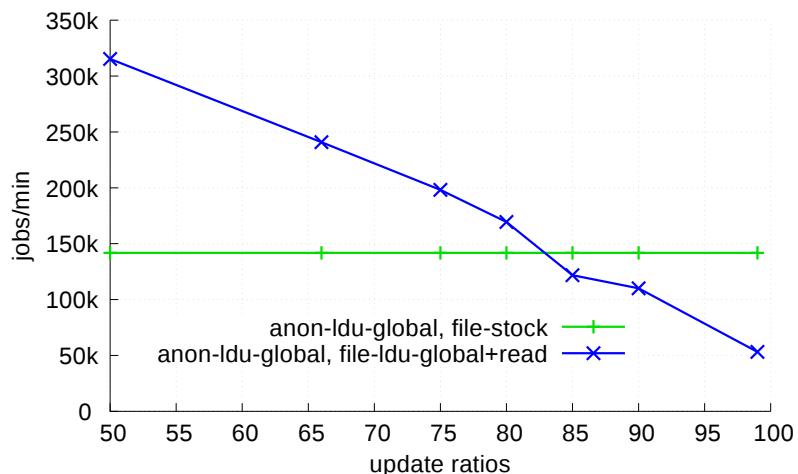


그림 4.22: Lmbench performance depending on update ratios.

그림 4.23: 업데이트 비율에 따른 Lmbench 성능.

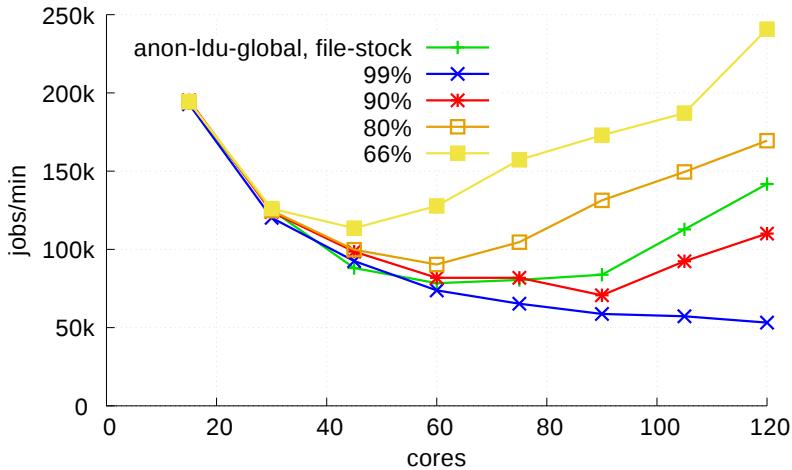


그림 4.24: 업데이트 비율에 따른 Lmbench 확장성.

는 기술이 오직 업데이트 비율이 높고 읽기에 대한 비율이 낮은 자료구조에 적합한 방법이기 때문이다. LDU의 읽기 명령은 로그를 적용하는 `synchronize` 함수를 호출하므로 읽기 명령에 때문에 더 느려지게 된다.

리드 명령에 대한 효과를 이해하기 위해, 우리는 리드 명령을 업데이트 오퍼레이션에 비율에 맞게 추가하여 성능을 측정하는 또 다른 실험을 하였다. 익명 rmap 자료 구조는 LDU의 전역 큐 버전을 이용하였고 우리는 파일 ramp에 대해서 순차적으로 리드(lock, synchronize) 비율을 증가시켰다. 그림 4.18의 위쪽 부분인 ,4.19 , 4.20, 4.21, 4.23 그리고 4.24는 120코에서의 업데이트 비율에 따른 성능을 보여주고, 아래 그래프는 성능에 대한 확장성을 보여준다.

AIM7는 다른 벤치마크 보다는 델 fork에 의존적인 벤치마크이기 때문에, 리드 명령어는 상대적으로 델 발생한다. 그 결과, 비록 자료구조가 75%의 업데이트 비율(3개의 업데이트와 1개의 읽기)을 가지지만, LDU의 버전의 리눅스는 수정 안 한 리눅스에 비해 높은 성능을 가진다. AIM7의 성능에 대한 확장성은 90% 이상의 뿐만 아니라 80%의 업데이트 비율을 가질때에도 수정

안한 리눅스 보다 높은 성능 확장성을 가진다.

Exim과 Lmbench는 매우 fork 집중적인 워크로드이다. 그 결과, 수정안 한 리눅스가 80%의 업데이트 비율을 가질 때 더 좋은 성능을 가진다. 하지만, LDU는 85% 이상의 업데이트 가질 때부터 더 높은 성능을 가진다. 이것은 심지어 LUD는 읽기가 꽤 자주 호출되는 워크로드라도 좋은 성능을 보여준다는 것을 설명한다.

## 제 5 장 결론 및 향후 연구

### 5.1 결론

우리는 새로운 동시적 업데이트 방법인 LDU를 제안하였고, 평가하였고, 이것은 매니코어 시스템에서 리눅스 커널의 성능 확장성을 향상시킨다. 우리의 방법은 사전에 연구되어 잘 알려진 로그 기반 알고리즘인 OpLog의 동기화된 타임 스템프 카운터의 관리에 따른 오버헤드를 제거 할 수 있다. 우리의 LDU를 리눅스 커널에 구현하여 수행한 우리의 실험은 기존 리눅스 커널에 비해 120코어 공유 메모리 기반 컴퓨팅 환경에서 2.7배까지 성능 향상을 이루었다. 우리는 이러한 LDU를 리눅스 커널 4.5-rc6에 구현하였으며, 본 결과물은 아래 사이트에서 오픈소스로 이용할 수 있다.

<https://github.com/manycore-ldu/ldu>

## 5.2 향후 연구

우리가 제안한 기술은 시간에 예민한 로그를 업데이트 순간마다 지움으로 상당한 성능에 대한 향상성을 얻었으나, 이것은 여전히 특정한 자료구조(예를 들어 스택, 큐) 같은 경우에는 남은 명령어들마다 시간 순서가 필요하므로 적용하지 못한다. 향후 연구로는 LDU의 기술과 OpLog의 기술을 통합하여 스택과 큐를 지원학 위해 새로운 동기화 기술을 개발하는 것이다.

## 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] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.
- [7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring)*, pages 483–485, 1967.
- [8] Tim Chen Andi Kleen. Scaling problems in Fork. In *Linux Plumbers Conference, September*, 2011.
- [9] 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.

- [10] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP ’09, pages 29–44, 2009.
- [11] Erich Bloch. The engineering design of the stretch computer. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM ’59 (Eastern), pages 48–58, 1959.
- [12] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. Numa policies and their relation to memory architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 212–221, 1991.
- [13] 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.
- [14] 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.
- [15] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. OpLog: a library for scaling update-heavy data structures. In *Tech-*

nical Report MIT-CSAIL-TR2014-019, 2014.

- [16] 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>.
- [17] Davidlohr Bueso. Scalability Techniques for Practical Synchronization Primitives. volume 58, pages 66–74, December 2014.
- [18] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP ’97, pages 143–156, 1997.
- [19] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 215–226, 2015.
- [20] 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.
- [21] David R. Cheriton and Willy Zwaenepoel. The distributed v kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP ’83, pages 129–140, 1983.
- [22] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the Seven-*

teenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, pages 199–210, 2012.

- [23] 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.
- [24] 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.
- [25] Jonathan Corbet. The case of the overly anonymous anon\_vma. 2010. <https://lwn.net/Articles/383162/>.
- [26] 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.
- [27] 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.
- [28] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Jabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen,

- and Amin Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM ’10, pages 339–350, 2010.
- [29] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. *SIGPLAN Not.*, 47(8):257–266, February 2012.
- [30] 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.
- [31] 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.
- [32] 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.
- [33] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. SPAA ’10, pages 355–364, 2010.
- [34] M. Frans Kaashoek. Parallel computing and the os. In *SOSP History Day 2015*, SOSP ’15, pages 10:1–10:35, 2015.
- [35] 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.

- [36] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa (summary). In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP ’79, pages 43–44, 1979.
- [37] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: Space-time Partitioning in a Manycore Client OS. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar’09, pages 10–10, 2009.
- [38] Victor Luchangco, Dan Nussbaum, and Nir Shavit. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing*, Euro-Par’06, pages 801–810, 2006.
- [39] 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.
- [40] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP ’91, pages 110–121, 1991.
- [41] 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.
- [42] Alastair J. W. Mayer. The architecture of the burroughs b5000: 20 years

- later and still ahead of the times? *SIGARCH Comput. Archit. News*, pages 3–10, 1982.
- [43] Paul McKenney. Some more details on Read-Log-Update. 2016.  
<https://lwn.net/Articles/667720/>.
- [44] Paul E McKenney. Is parallel programming hard, and, if so, what can you do about it? 2011.
- [45] 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.
- [46] Larry W McVoy, Carl Staelin, et al. Imbench: Portable Tools for Performance Analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.
- [47] 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.
- [48] 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.
- [49] Adam Morrison. Scaling synchronization in multicore programs. *Queue*, 14(4):20:56–20:79, August 2016.
- [50] John Ousterhout. Why threads are a bad idea (for most purposes). In *In USENIX Technical Conference (Invited Talk), Austin, TX, January 1996.*, 1996.

- [51] 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.
- [52] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 64–75, 1981.
- [53] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. In *Proceedings of the Fourth ACM Symposium on Operating System Principles*, SOSP '73, 1973.
- [54] J. H. Saltzer. Traffic control in a multiplexed computer. Cambridge, MA, USA, 1966. Massachusetts Institute of Technology.
- [55] Michael L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, 2013.
- [56] 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.
- [57] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 4–4, 2003.
- [58] 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, 2016.

- [59] 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.
- [60] Gerd Zellweger, Simon Gerber, Korniliос Kourtis, and Timothy Roscoe. Decoupling Cores, Kernels, and Operating Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI’14, pages 17–31, 2014.

## Abstract

# A Lightweight Log-based Deferred Update for Linux Kernel Scalability

*by Kyong, Joohyun*

*Department of Computer Science*

*Graduate School, Kookmin University,*

*Seoul, Korea*

In highly parallel computing systems with many-cores, a few critical factors cause performance bottlenecks severely limiting scalability. The kernel data structures with high update rate naturally cause performance bottlenecks due to very frequent locking of the data structures. There have been research on log-based synchronizations with time-stamps that have achieved significant level of performance and scalability improvements. However, sequential merging operations of the logs with time-stamps pose another sources of scalability degradation.

To overcome the scalability degradation problem, we introduce a lightweight log-based deferred update method, combining the log-based concepts in the dis-

tributed systems and the minimal hardware-based synchronization in the shared memory systems. The main contributions of the proposed method are:(1) we propose a lightweight log-based deferred update method, which can eliminate synchronized time-stamp counters that limits the performance scalability;and (2) we implemented the proposed method in the Linux 4.5-rc6 kernel for two representative data structures (anonymous reverse mapping and file mapping) and evaluated the performance improvement due to our proposed novel light weight update method. Our evaluation study showed that application of our method could achieve from 1.5x through 2.7x performance improvements in 120 core systems.