# 매니코어 CPU 시스템에서의 병렬 I/O 성능 향상을 위한 LRU 최적화 기법 연구

변은규 [1], 방지우 [2], 구기범 [1], 오광진 [1]
[1] 한국과학기술정보연구원 국가슈퍼컴퓨팅본부
[2] 서울대학교 컴퓨터공학부
ekbyun@kisti.re.kr, cilioh14@gmail.com, gibeom.gu@kisti.re.kr, koh@kisti.re.kr

# A Study on Optimizing LRU lock for Improving Parallel I/O Throughout in Manycore CPU Systems

Eun-Kyu Byun[1], Jiwoo Bang[2], Gibeom Gu[1], Kwang-Jin Oh[1]
[1]Div. of National Supercomputing, Korea Institute of Science and Technology Information
[2]Dept. of Computer Science and Engineering, Seoul National University

**요 약**

매니코어 CPU 시스템에서의 병렬 I/O 는 현재의 리눅스 시스템의 LRU 관리 방법의 한계로 확장성에 문제를 가지고 있다. 본 연구에서는 이 문제를 해결했던 하기 위한 개선된 FinerLRU 를 제안한다. LRU 락을 최대 코어 개수만큼 증가시키고 세분화된 Lock 관리를 통해 버퍼 캐시를 사용하는 파일 시스템의 병렬 I/O 성능을 향상시킨다. 리눅스 5.18.11 에 제안한 방법을 구현하였으며, 64 개의 물리적 코어와 256 개의 논리적 코어를 가지는 Intel Knights Landing 프로세서를 이용한 실험을 통해 두 배 가량의 성능 향상을 얻을 수 있음을 확인하였다.

## 1. 서론

HPC(High Performance Computing)시스템은 전통적인 과학계산 응용에 더하여 빅데이터, AI 등 대규모 데이터 처리에도 널리 사용되고 있다. 이러한 필요에 의해 최근 제공되는 서버 용 프로세서들은 수십개의 코어를 탑재하는 것이 일반적인 상황이다. [1] 이러한 매니코어 CPU 는 강력한 병렬 연산 성능을 제공하는 것 외에도 메모리나 스토리지 장치에의 병렬 접근도 가능하게 만들어준다.

본 연구에서는 매니코어 CPU 에서의 병렬 I/O 성능 향상을 목표로 기존 리눅스 운영체제의 LRU 병목을 해결하는 방안을 제안한다. 대부분의 파일시스템에서 쓰기와 읽기를 수행할 경우, 성능 향상을 위해 버퍼 캐시를 이용하며 이때 사용된 메모리 페이지를 관리하기 위한 LRU 리스트가 존재한다. 본 연구팀은 이전의 연구를 통해 이 LRU 리스트의 Lock 메커니즘이 병렬 쓰기의 확장성을 크게 저해함을 발견하고 FinerLRU 를 제안하여 리눅스 5.2.8 에서 병목을 일부 해결하였다. [2] 이는 기존의 몇 가지 선행 연구들 보다 안전성과 성능에서 장점을 보였다. [3][4]

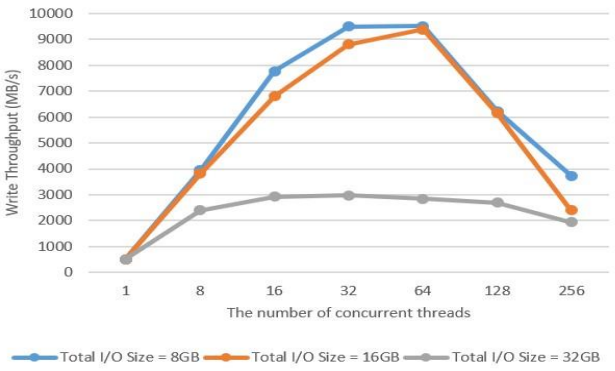
기존의 FinerLRU 는 특정 상황이나 설정에서 비효율이 존재하였고, 추가로 이후 버전이 많이 올라간 리눅스에 직접적인 적용이 어려운 상황이 되었다. 이에 본 연구에서는 이전 버전의 FinerLRU 보다 단순하지만 효율적인 방식을 개발하여 리눅스 5.18.11 에 구현하였고 추가적인 성능 향상을 확인하였다.

## 2. 개선된 FinerLRU 의 작동방식

현재의 리눅스(버전 5.18.11, 연구수행당시 최신의 stable 판)에서 매니코어 CPU 의 I/O 성능 확장성이 좋지 않음을 보여주는 결과가 그림 1 에 나타나 있다. 실험은 64 cores, 256 threads 를 가지는 Intel Xeon Phi 7210 (Knights Landing, KNL)프로세서[5]를 탑재한 서버에서 진행하였으며, IOR 벤치마크[6]를 이용하여 여러 개의 MPI 프로세서가 각각 file-per-process 방식으로 동시에 순차 쓰기를 진행할 때의 총 Throughput 을 측정하였다. 결과에서 명확히 나타나는 것처럼 동시에 쓰기를 수행하는 쓰레드의 수가 증가함에 따라 성능이 증가하다가 32~64 개의 스레드에서 그 증가폭이 둔화되고, 그 이상이 되면 오히려 성능이 크게 줄어들게 된다.

앞서 소개한 지난 연구[2]에서 분석한 결과 이러한 성능 저하의 원인은 버퍼 캐시를 관리하는 LRU 의 Lock 매커니즘 때문이다. LRU 는 lruvec 구조체 내부에 LRU_INACTIVE_ANON, LRU_ACTIVE_ANON,

LRU_INACTIVE_FILE, LRU_ACTIVE_FILE, LRU_UNEVICTABLE 의 리스트와 이를 갱신할 때 사용하는 하나의 Lock 변수를 가지고 있다. 갱신 요청은 pagevec 구조체에 저장되었다가 16 개가 쌓이면 lock 을 잡고 전부 처리된 후 lock 을 풀게 된다. 문제는 lruvec 구조체가 소켓당 혹은 메모리 cgroup 당 하나만 존재한다는 점이다. 즉, 이번 실험의 경우 최대 256 개의 프로세스가 동시에 쓰기를 수행하기 위해 단 하나의 lock 을 잡기 위해 경쟁하였다는 것이다.




(그림 1) 리눅스 시스템에서 병렬 쓰기를 수행하였을 때 쓰레드 수 증가에 따른 성능 변화

이를 해결하기 위해 이전 연구에서 FinerLRU 를 제안하였다. lruvec 구조체가 관리하는 5 개의 리스트 각각에 대하여 Lock 을 두고 추가로 FINER_FACTOR 라는 전역 변수를 설정하여 lruvec 구조체가 관리하는 리스트의 개수를 각각 그 수만큼 증가시켰다. 예를 들어 FINER_FACTOR 가 8 인 경우 하나의 lruvec 구조체에는 각 리스트가 8 개씩 40 개 존재하며 lock 의 개수도 그에 맞게 40 개가 존재하게 된다. 이에 따라 LRU 리스트를 갱신하는 함수들도 자신이 접근하는 LRU 리스트에 해당하는 lock 만을 잡도록 수정하였다. 또한 page 구조체에 어떠한 LRU 리스트에 소속되는지를 나타내는 finer_index 를 저장하기 위한 멤버 변수와 이를 계산하기 위한 함수가 추가되었다. 또한 lock 이 많아져서 발생하는 오버헤드를 줄이기 위해 lazy lock release 를 통해 동일한 lock 에 대한 연산이 연속될 경우 lock 을 다시 잡는 일이 없도록 하였다.

기존 FinerLRU 는 FINER_FACTOR 를 8 까지 증가시켰을 때에는 성능이 좋아졌으나, 그보다 더 큰 경우 오히려 성능이 떨어지는 현상이 발생하였다. 더구나 리눅스 커널의 버전이 올라가면서 LRU 와 관련된 부분에 일부 변화가 생겨서 직접 적용이 불가능하다. 이에 본 연구에서는 개선된 FinerLRU 를 개발하였다.

가장 큰 변화는 기존에 FINER_FACTOR 개수 만큼 각 LRU 리스트 개수를 늘리고 리스트마다 lock 을 두었던 것과 달리, lock 의 개수는 FINER_FACTOR 만큼만 두고 각 lock 이 4 개 리스트(LRU_UNEVICTABLE 은 논리적으로는 존재하나 실제로는 쓰이지 않음)에 대한 접근을 한번에 보호하도록 변경한 것이다. Page 구조체의 소속 LRU 가 변경될 때, 항상 같은 finer_index 를 가지는 리스트들 사이에서 이동/추가/제거가 이루어 지기 때문에, 하나의 lock 으로 모든 리스트 변경 함수들에 대해 참여하는 모든 리스트들의 보호가 가능하다. 이 방식이 효율적인 이유는 리눅스 최신 버전에서 LRU 변경이 코어당 한 스레드에서만 수행 가능하기 때문에, 서로 다른 finer_index 간의 간섭이 발생할 여지가 많이 줄어들었기 때문이다.

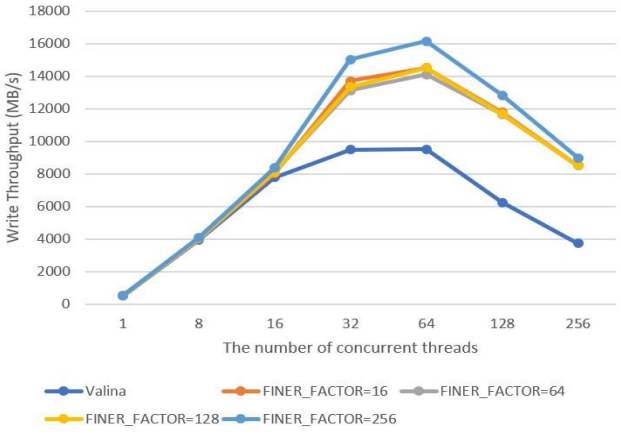
추가로 page 구조체와 새롭게 도입된 page 구조체의 wrapper 인 folio 구조체에 finer_index 를 저장하는 변수를 추가하지 않고 finer_index 를 계산하는 방법을 설계하였다. finer_index 는 다음과 같이 folio 의 주소를 이용하여 계산된다.

*(unsigned long)folio>>10 % FINER_FACTOR* (식 1)

10 자리 shift 를 하는 이유는 6 자리는 folio 구조체가 64byte 로 정렬되어 있기 때문에 각자 다른 값을 가지게 하기 위함이고, 4 자리는 16 개씩 모아서 LRU 갱신 함수를 호출하기 때문에, 연속된 16 개의 folio 가 동일한 finer_index 를 가지게 하여 연속쓰기 연산에서 lazy lock release 의 효과를 향상시키기 위함이다.

## 3. 실험 결과

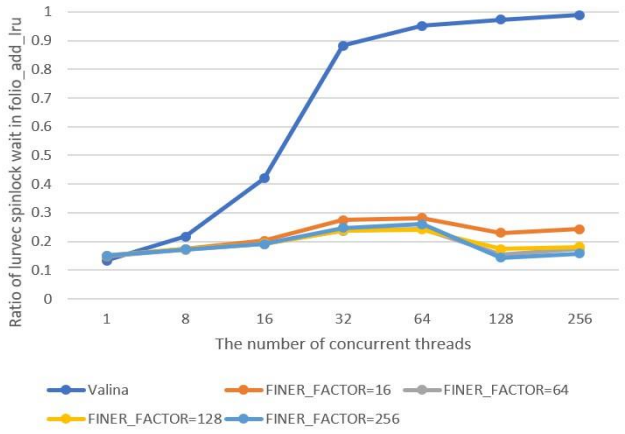
실험은 KNL 7210 프로세서를 탑재한 서버에서 IOR 벤치마크를 이용하여 총 8GB 의 데이터를 쓸 때의 성능을 측정하였다. KNL 프로세서는 최대 72 개의 코어와 288 개의 스레드를 가지는 매니코어 CPU 로 여전히 세계 상위 50 위권의 슈퍼컴퓨터 중 여러 시스템에서 활용되고 있는 CPU 이다. [7]




(그림 2) 개선된 FinerLRU 적용 후 FINER_FACTOR 를 256 까지 증가할 때 동시 접속 쓰레드에 따른 쓰기 대역폭 변화

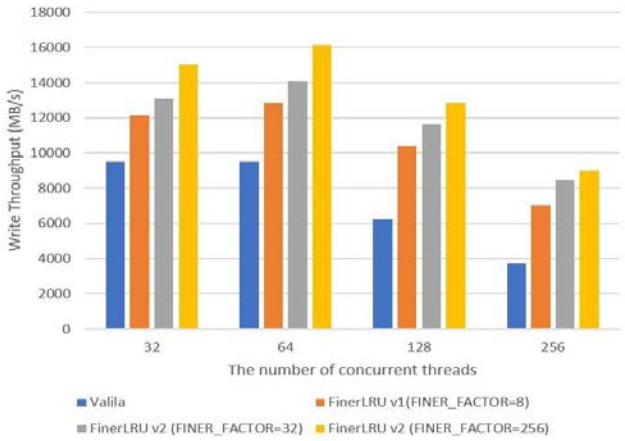
첫 번째로 그림 1 에서 측정한 것처럼 IOR 벤치마크를 이용하여 여러 MPI 프로세스가 병렬 쓰기를 file-per-process 방식으로 수행할 때의 총 쓰기 대역폭을 참여하는 스레드의 개수와 FINER_FACTOR 를 바

꿔가며 측정해 보았다. 그림 2 에 그 결과가 나타나 있다. FinerLRU 를 적용시키는 것으로 성능향상이 발생하지만 특히 FINER_FACTOR 의 값을 최대 논리 코어개수인 256 으로 설정했을 경우 성능 향상이 최대화된다. 그 비율은 동시 접속 스레드의 수가 증가할수록 더 커져서 64~256 스레드의 경우 70%에서 140%까지 성능향상을 보인다.




(그림 3) folio_lru_add 중 lruvec lock 을 얻기 위해 기다리는 시간의 비율

그림 2 에 따르면 성능 향상에도 불구하고 스레드 수가 128 개 이상이 되면 그보다 적은 스레드를 사용할 때 보다 성능이 떨어지는 것을 확인할 수 있다. 이것이 여전히 LRU lock 을 기다리는 것 때문인지 확인하기 위하여 추가적인 실험을 진행하였다. 커널에서 쓰기 연산이 수행될 때 실제로 LRU 리스트의 갱신은 folio_add_lru 함수에서 진행된다. 프로파일링을 통해 전체 folio_add_lru 함수 수행시간 중 lruvec lock 을 기다리는 시간의 비율을 측정하였고 그 결과가 그림 3 에 나타나 있다. 기존의 리눅스에서는 스레드 수가 많아 짐에 따라 lock 대기 시간이 전체 수행의 대부분을 차지하는 것과 달리 개선된 FinerLRU 를 적용한 경우 256 스레드의 경우에도 그 비율이 20%미만만 차지 하는 것을 확인할 수 있다. 이는 LRU lock 경합이 병목이었던 문제는 해결되었으며, 하드웨어적 혹은 다른 소프트웨어 부분이 병목이 되어 확장성을 가로막고 있는 것으로 보인다.




(그림 4) 기존 FinerLRU 와 개선된 FinerLRU 의 성능 비교

마지막으로 새롭게 구현된 FinerLRU 가 기존의 FinerLRU 보다 어느정도 성능이 향상되었는지를 비교한 결과를 그림 4 에서 정리하였다. 기존의 방식에서 가장 성능이 좋았던 것은 FINER_FACTOR 를 8 로 두었을 때이고, 이때 실사용 되지 않는 LRU_UNEVICTABLE 리스트를 제외하면 lruvec 구조체에는 32 개의 lock 과 리스트를 가지게 된다. 이와 비교하기 위하여 새로운 FinerLRU 의 FINER_FACTOR 를 32 와 256 로 각각 설정한 결과와 비교하였다. 결과에 따르면 개선된 버전이 모든 동시 접속 스레드 개수의 경우에서 기존보다 성능이 추가로 향상되었음을 알 수 있다.

## 4. 결론

본 논문에서는 매니코어 CPU 시스템에서 수십 개 이상의 스레드가 병렬 I/O 를 수행할 때 기존의 리눅스의 LRU Lock 방식이 충분한 성능 확장성을 제공하지 못하는 것을 보완하기 위하여 개선된 FinerLRU 를 제안하였다. 이를 리눅스 5.18.11 에 구현하였으며, KNL CPU 기반 시스템에 IOR 을 이용한 실험을 통해 병렬쓰기의 성능이 2 배 가량 증가함을 확인하였다.

**본 논문은 대한민국 정부의 재원으로**
**한국과학기술정보연구원 주요사업의 지원을 받아**
**수행된 연구임 (과제번호: K-22-L02-C06-S01)**

## 참고문헌

[1] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," ACM Comput. Surv., vol. 48, no. 3, Feb. 2016.

[2] J. Bang et al. "Finer-LRU: A Scalable Page Management Scheme for HPC Manycore Architectures," 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 567-576, 2021.

[3] Y. Zhang et al. "FineLock: automatically refactoring coarse-grained locks into fine-grained locks," In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020). Online, 2020, pp. 565–568.

[4] K. Ganesh, S. Kalikar, and R. Nasre, “Multi-granularity Locking in Hierarchies with Synergistic Hierarchical and Fine-Grained Locks,” in Euro-Par 2018: Parallel Processing, Turin, Italy, 2018, pp.546–559.

[5] A. Sodani et al., "Knights Landing: Second-Generation Intel Xeon Phi Product," in IEEE Micro, vol. 36, no. 2, pp. 34-46, 2016.

[6] IOR Benchmark, https://github.com/hpc/ior

[7] TOP500, https://www.top500.org/lists/top500