KIPS Trans. Comp. and Comm. Sys.
Vol.12, No.7 pp.209~216
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)

https://doi.org/10.3745/KTCCS.2023.12.7.209

Optimizing LRU Lock Management in the Linux Kernel for Improving Parallel Write Throughout in Many-Core CPU Systems

Eun-Kyu Byun† · Gibeom Gu†† · Kwang-Jin Oh†† · Jiwoo Bang†††

ABSTRACT

Modern HPC systems are equipped with many-core CPUs with dozens of cores. When performing parallel I/O in such a system, there is a limit to scalability due to the problem of the LRU lock management policy of the Linux system. The study proposes an improved FinerLRU to solve this problem. Our new FinerLRU improves the parallel write performance of file systems using the buffer cache through granular lock management by increasing the number of LRU locks upto the maximum number of cores. The proposed method was implemented in Linux 5.18.11, and the performance was measured on two types of CPUs, Intel Icelake Xeon and Intel Knights landing, with different characteristics, and it was found that a performance improvement of about two times can be obtained in both types of systems.

Keywords : Manycore CPU, Linux Kernel, LRU, Parallel I/O

매니코어 CPU 시스템의 병렬 쓰기 성능 향상을 위한 리눅스 커널의 LRU 관리 최적화 기법

변 은 규† · 구 기 범†† · 오 광 진†† · 방 지 우†††

요 약

최신 HPC 시스템은 수십 개의 코어를 가진 매니코어 CPU를 탑재하고 있다. 이런 시스템에서 병렬 I/O를 수행할 경우 리눅스 시스템의 LRU락 관리 정책의 문제로 인해 확장성에 한계를 가지고 있음을 확인하였다. 본 연구에서는 이 문제를 해결하기 위한 개선된 FinerLRU를 제안한다. LRU락을 최대 코어 개수만큼 증가시키는 것을 골자로 한 세분화된 Lock 관리를 통해 페이지 기반 버퍼 캐시를 사용하는 파일 시스템의 병렬 쓰기 성능을 향상시키는 것을 목적으로 한다. 리눅스 5.18.11에 제안한 방법을 구현하였으며, 서로 다른 특성을 가진 2종류의 CPU인 Intel Icelake Xeon과 Intel Knights landing에서 성능을 측정하였고 두 종류의 시스템 모두에서 두 배 전후의 성능 향상이 발생함을 확인하였다.

키워드 : 매니코어 CPU 시스템, 리눅스 커널, LRU, 병렬 I/O

1. 서 론

HPC(High Performance Computing)시스템은 전통적인 과학계산 응용에 뿐만 아니라 빅데이터, AI 등 대규모 데이터 처리에도 광범위하게 사용되고 있다. 이러한 추세에 따라 최근 제공되는 서버용 CPU는 수십 개의 코어를 가지는 것이 일반적인 상황이다[1]. 이런 종류의 매니코어 CPU를 이용하여 강력한 병렬 연산 성능을 제공하는 것 뿐 아니라 스토리지 장치에의 병렬 접근도 가능하게 한다.

본 연구에서는 매니코어 CPU에서의 병렬 쓰기 성능 향상을 목표로 Linux 운영체제의 LRU 매커니즘의 병목을 해결하는 방법을 제시한다. 일반적인 리눅스 파일시스템에서 쓰기 연산이 실행되면, 성능 향상을 위해 버퍼 캐시를 이용하며 이때 캐시로 이용된 메모리 페이지들을 관리하기 위해 LRU 리스트를 가진다. 본 연구팀은 이전의 연구에서 Linux의 LRU리스트의 Lock관리 메커니즘이 병렬화된 매니코어 CPU 시스템의 성능을 고려하지 않고 설계되어 병렬 쓰기의 경우 확장성이 제한됨을 확인하고 FinerLRU를 설계하여 Linux 버전 5.2.8에서 병목을 일부 해결한 연구를 진행하였다[2]. 그러나 실험을 통해 초기 버전의 FinerLRU는 특정 상황이나 설정에서 비효율이 존재함을 확인하였고 추가로 이후

※ 본 연구는 2023년도 한국과학기술정보연구원(KISTI) 기본사업 과제 및 대한민국 정부(과학기술정보통신부)의 재원으로 한국연구재단 슈퍼컴퓨터개발선도사업(과제번호:2020M3H6A1084853)의 지원을 받아 수행되었음.
※ 이 논문은 2022년 한국정보처리학회 ACK 2022의 우수논문으로 "매니코어 CPU 시스템에서의 병렬 I/O성능 향상을 위한 LRU최적화 기법 연구"의 제목으로 발표된 논문을 확장한 것임.
† 정 회 원 : 한국과학기술정보연구원 슈퍼컴퓨팅기술개발센터 선임연구원
†† 비 회 원 : 한국과학기술정보연구원 슈퍼컴퓨팅기술개발센터 책임연구원
††† 비 회 원 : 서울대학교 컴퓨터공학부 석·박사통합과정
Manuscript Received : December 20, 2022
Accepted : January 26, 2023
* Corresponding Author : Eun-Kyu Byun(ekbyun@kisti.re.kr)

버전이 많이 올라간 리눅스에 맞춘 구현본이 필요하게 되었다. 본 연구팀은 이전 버전의 FinerLRU보다 단순하지만 효율적인 방식을 개발하여 리눅스 5.18.11에 구현하였고 추가적인 성능 향상을 확인하여 발표하였다[3]. 본 논문에서는 기존의 알고리즘의 새로운 설명을 추가하고 범용 매니코어 프로세서에서의 성능 분석 수행한 결과를 추가로 제공한다.

리눅스 커널에서 공유되는 자원에 대한 락 경합을 줄이기 위한 연구는 많이 진행되었다. D. Zheng 등은 전역 캐시를 여러개의 독립된 페이지 집합으로 나누어 멀티코어 시스템에서의 lock 경합을 감소시키는 방식을 제안하였다[4]. Y. Zhang 등은 응용 레벨에서 자동으로 coarse-grained lock을 fine-grained lock으로 변환해주는 리팩터링 도구를 제안하였다[5]. 이런 연구들의 결과로 응용 단위에서의 I/O 성능을 향상시키는 효과를 얻을 수 있었다. 이와 다르게 우리의 연구는 커널 수준에서의 작업을 통해 경쟁이 심한 lock 구조를 세분화된 finer-grained lock로 변환함으로써 I/O 성능이 크게 향상될 수 있음을 보였다. 또 다른 접근 방식으로 다양한 밀집도의 잠금 방법을 사용한 연구들이 있었다. K. Ganesh 등[6]은 각 쓰레드의 밀집도 요구량에 따라 fine-grained와 coarse-grained lock을 섞어서 사용하는 방안을 제안하였다. 그러나 이 방식은 실행시간에 스레드별 최적의 lock 방법을 찾아내는데 추가적인 오버헤드를 발생시켜서 성능향상에 한계가 있었다.

2. 본 론

2.1 기존 리눅스 LRU방식의 확장성 한계

연구수행당시 가장 버전이 높은 안정화 버전이였던 Linux 버전 5.18.11에서 매니코어 CPU의 I/O성능 확장성이 어떤지를 실험하였고 그 결과가 Fig. 1에 정리되어 있다. 이 실험은 64개의 물리코어와 256개의 논리 스레드를 가지는 Intel Xeon Phi 7210 (Knights Landing, KNL)프로세서[7]를 장착한 서버에서 진행하였으며, IOR 벤치마크[8]를 이용하여 동시에 복수의 MPI프로세서가 각각 file-per-process방식으로 순차 쓰기를 수행할 때의 각 쓰기 성능(Throughput)의 총합을 측정하였다. 결과가 명확히 보여주는 바와 같이 동시에 쓰기를 실행하는 쓰레드의 수가 증가함에 따라 성능이 증가하다가 32~64개의 스레드에서 그 증가폭이 둔화되고, 그 이상의 스레드가 동시에 수행 시 성능이 크게 줄어들게 된다.

이러한 성능 저하는 버퍼 캐시를 관리하는 LRU의 Lock매커니즘 때문이다. Fig. 2와 같이 LRU는 lruvec 구조체 내부에 LRU_INACTIVE_ANON, LRU_ACTIVE_ANON, LRU_INACTIVE_FILE, LRU_ACTIVE_FILE, LRU_UNEVICTABLE의 리스트와 이를 수정할 때 사용하는 하나의 lock 변수를 가지고 있다. 수정 요청은 pagevec구조체에 저장되었다가 16개까지 모이게 되면 LRU lock을 잡고 전부 처리된 후

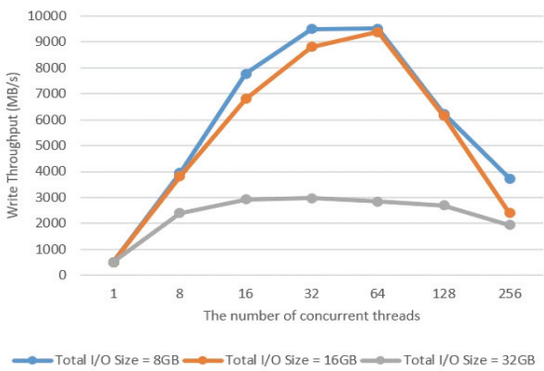


Fig. 1. Parallel Write Performance According to the Number of Concurrent Threads in the Original Linux System

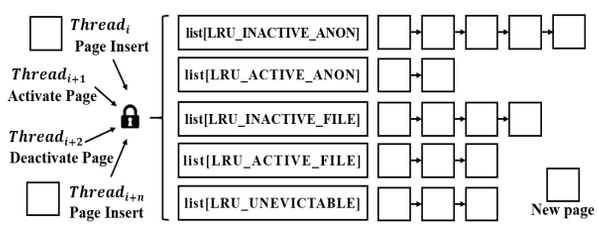


Fig 2. The Original LRU list Locking Mechanism

lock을 풀게 된다. 문제는 lruvec 구조체 소켓당 혹은 메모리 cgroup당 하나씩만 존재한다는 특징이다. 즉, 이 실험의 경우 최대 256개의 MPI프로세스가 동시에 쓰기를 수행하기 위해 단 하나의 lock을 잡기 위해 경쟁하였다는 것이다. 이는 동시 쓰기의 확장성을 크게 저해하게 된다.

2.2 FinerLRU : 다수의 Lock을 이용한 확장성 향상 기법

이를 해결하기 위한 첫 번째 해결책으로 본 연구팀은 FinerLRU를 제안하였다. lruvec 구조체에서 관리하는 5종류의 리스트들 각각을 관리하는 Lock을 따로 두고 추가로 FINER_FACTOR라는 환경 변수를 이용하여 lruvec 구조체당 관리하는 리스트의 개수를 그 수만큼 늘렸다. 예를 들어 FINER_FACTOR가 8인 경우 하나의 lruvec 구조체에 5종류의 리스트 각각이 8개씩 총 40개가 존재하며 lock의 개수도 그에 맞게 40개가 된다. 이런 변화에 대응하여 LRU 리스트를 수정하는 함수들도 자신이 접근하는 LRU리스트에 해당하는 lock만을 사용하도록 수정되었다. 또한 page 구조체를 수정하여 자신이 어떠한 LRU 리스트에 소속되어 있는지를 나타내는 finer_index를 저장하기 위한 멤버 변수와 이를 계산하기 위한 함수가 추가되었다. 따라서 다른 finer_index에 해당하는 list들의 갱신이 동시에 수행이 가능해 지고 이에 따라 lock 경합이 전체적으로 줄어들어서 병렬 쓰기 성능이 향상 될 것으로 기대하였다. 반대로 lock이 많아지면 리스트를 갱신하는 연산 마다 lock을 잡고 푸는 작업이 많아짐에 따라

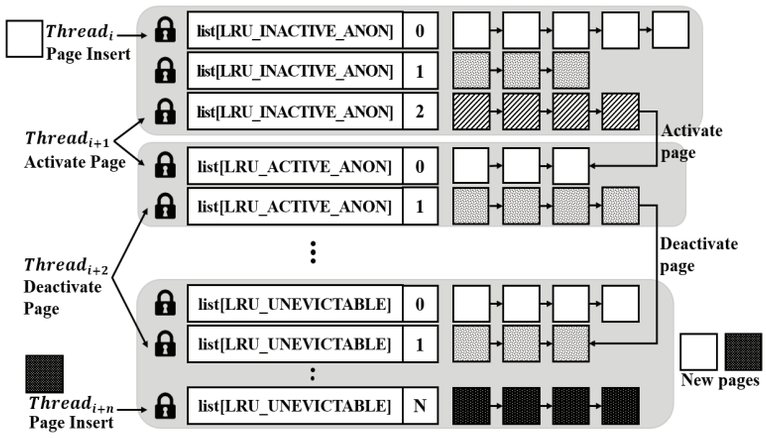


Fig. 3. LRU List Locking Mechanism of the First Version of Finer-LRU

오버헤드가 발생할 수 있다. LRU리스트를 갱신하는 작업은 1개의 lock 혹은 2개의 lock을 잡게 된다. 이러한 오버헤드를 최소한으로 줄이기 위해 lazy lock release를 도입하였다. 즉, lock을 바로 풀지 않고 잡고 있다가 다음번 LRU갱신 작업이 필요로 하는 lock과 비교를 한 후 동일한 lock에 대한 연산일 경우 lock을 다시 잡지 않는다.

2.3 FinerLRU 최적화

위에서 설명한 최초의 FinerLRU는 FINER_FACTOR를 8까지 증가시켰을 때까지는 성능이 향상되었으나, 그 이상의 경우 빈대로 성능이 저하되는 것으로 측정되었다. 더구나 새로운 버전의 리눅스 커널이 발표되면서 LRU와 관련된 소스코드에 변화가 생겨서 직접적인 적용이 어려웠다. 이에 본 연구에서는 개선된 FinerLRU를 개발하였다.

가장 큰 변화는 기존에 FINER_FACTOR개수 만큼 각 LRU리스트 개수를 늘리고 리스트마다 lock을 두었던 것과 다르게, lock의 개수는 FINER_FACTOR만큼만 두고 각 lock이 4종류의 리스트(5개 중 LRU_UNEVICTABLE은 논리적으로는 존재하나 실제로는 쓰이지 않음)에 대한 접근을 한 번에 보호하도록 변경한 것이다. Page/Folio구조체의 가속한LRU가 변하는 연산의 경우, 항상 같은 finer_index를 가지는 리스트들 사이에서 이동/추가/제거가 이루어지는 것을 확인하였기 때문에, 하나의 lock으로 모든 리스트 수정 함수들에 대해 참여하는 모든 리스트들의 보호가 가능하다. 이 방식으로도 충분한 이유는 리눅스 최신 버전에서 LRU변경이 코어당 하나의 스레드에서만 수행되도록 강제되기 때문에, 서로 다른 finer_index간의 간섭이 발생할 여지가 많이 줄어들었기 때문이다. 이런 구조는 Fig. 4에 표현되어 있다.

이전 FinerLRU와 마찬가지로 각 페이지는 해당하는 finer_index를 가지게 된다. 기존과 다르게 page구조체와 folio 구조체에 (최근 linux에서 도입된 page 구조체의 랩퍼) 멤버 변수를 추가하여 finer_index를 저장하는 대신 finer_

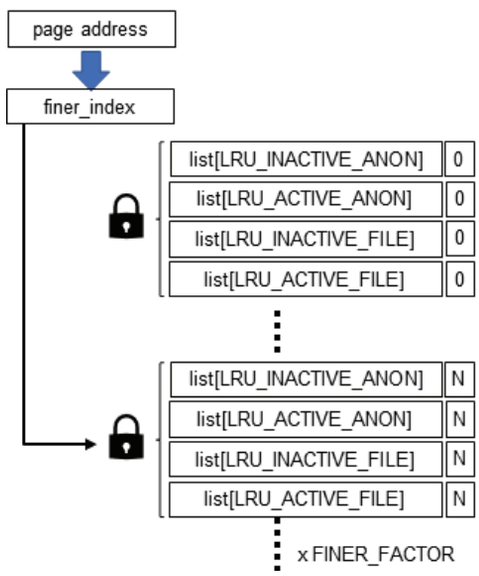


Fig. 4. LRU List Locking Mechanism of the Advanced Version of Finer-LRU

index를 계산하는 방법을 설계하였다. finer_index는 다음 Equation (1)과 같이 folio의 주소 값을 이용하여 계산된다.

$$(\text{unsigned long})\ \text{folio} \gg 10\ \%\ \text{FINER\_FACTOR} \quad (1)$$

위 식에 따르면 10자리 shift를 하는데, 그 중 6자리는 folio 구조체가 64byte 크기로 정렬되어 있어서 각자 다른 값을 가지게 위함이고, 4자리는 pagevec구조체에서 LRU갱신 연산을 16개씩 모아서 처리하므로, 연속된 16개의 folio가 동일한 finer_index를 가지게 하여 순차쓰기 연산이 발생할 경우 같은 finer_index를 가진 작업이 pagevec구조체에 한꺼번에 모일 가능성을 높히기 위해서이다.

lock의 개수가 줄어들었기 때문에 lazy lock release의 구현도 단순화 되었다. 초기 버전의 FinerLRU의 경우에는

하나의 LRU갱신 작업에서 2개의 lock을 관리해야하는 경우가 있었지만, 개선된 버전의 경우에는 하나의 lock만을 잡는 것만으로 모든 LRU갱신 작업의 처리가 가능하다. 이러한 작업들은 pagefec_lru_move_fn, __pagevec_lru_add 를 통해 수행 되며 공통적으로 folio_lruvec_relock함수 군을 호출한다. 이 함수는 이미 lruvec의 lock이 잡혀져 있는 상태에서 다음 연산이 다음 lruvec의 lock을 잡아야 할 때, 즉 연속된 LRU갱신 작업을 처리할 때, 이전의 lruvec와 동일한 lruvec에 대한 연산의 경우 lock을 풀고 다시 잡는 것을 방지하는 역할을 한다. 원래 lruvec당 하나의 lock만 존재했는데 FinerLRU의 경우 lruvec마다 FINER_FACTOR만큼의 lock을 가지도록 변경하였기 때문에, folio_lruvec_relock 함수들이 lock을 새로 잡기 위한 판단을 할 때 lruvec자체 뿐 아니라 이전에 작업에 사용한 finer_index를 동시에 비교하도록 수정하였다. Algorithm 1에는 LRU갱신 함수들의 표준 수행 방식에 대한 pseudo-code가 정리되어 있다.

3. 실험 결과

3.1 Intel Knights Landing CPU에서의 성능 평가

실험은 KNL 7210프로세서를 장착한 시스템에서 IOR 벤치마크를 활용해 총 8GB의 데이터를 쓰는 동안의 성능을 측정하였다. KNL CPU는 최대 72개의 물리코어와 288개의 논리스레드를 갖는 Many-core CPU로 여전히 세계 상위권의 슈퍼컴퓨터 중 여러 시스템에서 활용되고 있다[9].

첫 번째로 IOR 벤치마크를 이용하여 복수의 MPI 프로세스가 쓰기를 file-per-process방식으로 병렬로 수행할 때의 총 쓰기 대역폭을 참여 스레드의 개수와 FINER_FACTOR를 바꿔가며 측정하였고 그 결과는 Fig. 5에 정리되어 있다. FinerLRU의 적용으로 성능이 향상되고 특히 FINER_FACTOR를 최대 논리 스레드 개수인 256으로 설정했을 경우 성능 향상이 가장 크다. 그 비율은 동시 접속 스레드의 수가 증가할수록 더 커져서 64~256스레드의 경우 70%에서 140%까지 성능향상을 보인다.

Fig. 5에 따르면 성능 향상에도 불구하고 스레드 수가 128개 이상이 되면 그보다 적은 스레드를 사용할 때 보다 성능이 하락하는 것을 확인할 수 있다. 이것이 LRU lock을 기다리는 것에 의한 것인지 확인하기 위하여 추가 실험을 수행하였다. 커널에서 쓰기 연산이 실행될 때 실제로 LRU리스트의 갱신은 folio_add_lru함수에서 처리된다. 프로파일링을 통해 전체 folio_add_lru 함수 수행시간 중 lruvec lock을 기다리는 시간의 비율을 측정하였고 그 결과가 Fig. 6에 나타나 있다. 기존의 리눅스에서는 스레드 수가 많아짐에 따라 lock대기 시간이 전체 수행의 대부분을 차지하는 것과는 다르게 개선된 FinerLRU를 적용한 경우 256스레드의 경우에도 그 비율이 20%미만만을 차지하는 것을 볼 수 있다. 이는 LRU lock경합이 병목의 원인이었던 문제는 해결되었음을 의미한다. 추가적인 실험 결과 write 시스템 콜의 후반부에 수행되는 실제로 메모리나 디스크에 에 데이터가 쓰이지는 부분에서 수행시간이 증가되는 것을 확인하였다. 향후 연구를 통해 이런 부분에 대한 병목을 해결하여 확장성을 추기로 확보해야 할 것으로 보인다.

Algorithm 1. Pseudo-code of LRU Modification

```
modifyLRU(new_page, LRU_mod_action)
new_idx = (address of new_page)>>10 % FINER_FACTOR
add(pagevec[new_idx], page)
prev_lruvec = NULL
prev_index = NULL
prev_lock = NULL
if (size of pagevec[new_idx] >= 16)
  for every page in pagevec[new_idx]
    lruvec = get_lruvec_from_page(page)
    index = (address of page)>>10 % FINER_FACTOR
    if ( lruvec != prev_lruvec || index != prev_index)
      if (prev_lock != NULL) unlock(prev_lock)
      this_lock = lruvec->FinerLock[index]
      lock(this_lock)
      prev_lock = this_lock
    end
    do(LRU_mod_action)
  end
  if (prev_lock != NULL) unlock(prev_lock)
end
```

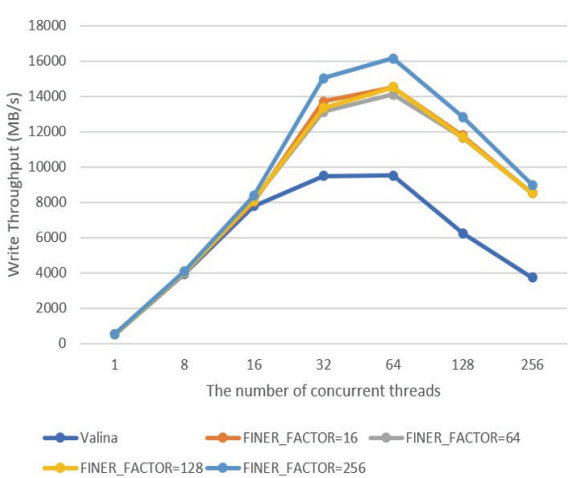


Fig. 5. Parallel Sequential Write Performance of KNL CPU Server as the Number of Concurrent Threads Increases in Various FINER_FACTOR Settings

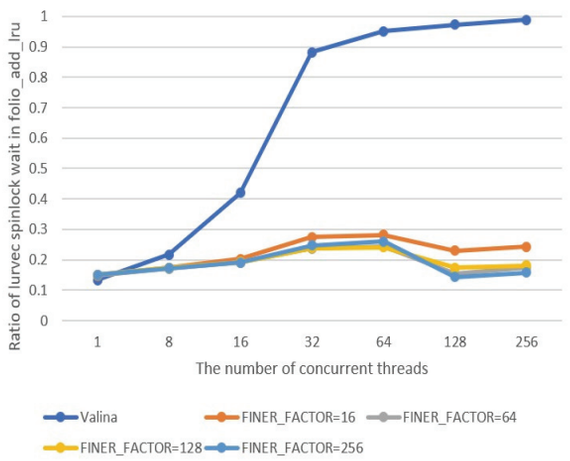


Fig. 6. The Ratio for Waiting Lruvec Spin Lock in folio_lru_add Funtion in KNL CPU Server

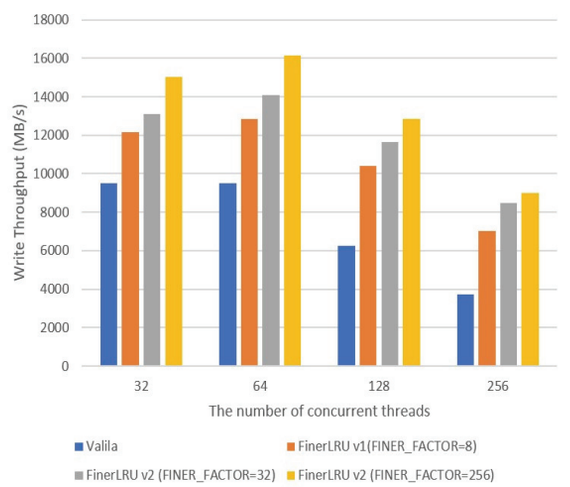


Fig. 7. Comparison Between the First Version finerLRU and the Advanced One

다음으로 개선된 FinerLRU가 초기의 FinerLRU보다 대비 어느 정도의 성능 향상이 있는지를 비교한 결과를 Fig. 7에서 정리하였다. 기존의 FinerLRU에서 가장 성능이 좋았던 것은 FINER_FACTOR를 8로 두었을 때이고, 이때 실사용되지 않는 LRU_UNEVICTABLE리스트를 제외하면 lruvec 구조체에는 32개의 lock과 리스트를 가진다. 이와 비교하기 위하여 개선된 FinerLRU의 FINER_FACTOR를 32와 256로 각각 설정하여 측정한 결과와 비교를 하여 보았다. 결과에 따르면 개선된 버전이 모든 동시 접속 스레드 개수의 경우에서 기존보다 성능이 추가로 향상되었음을 확인하였다.

FinerLRU는 본래 병렬 쓰기 성능을 향상시키기 위해 개발되었고 FINER_FACTOR 만큼 lock의 개수와 LRU 리스트의 개수를 증가시켰다. 이러한 오버헤드가 쓰기가 아닌 읽기 연산의 경우에 얼마나 큰 문제가 되는지를 확인하기 위해

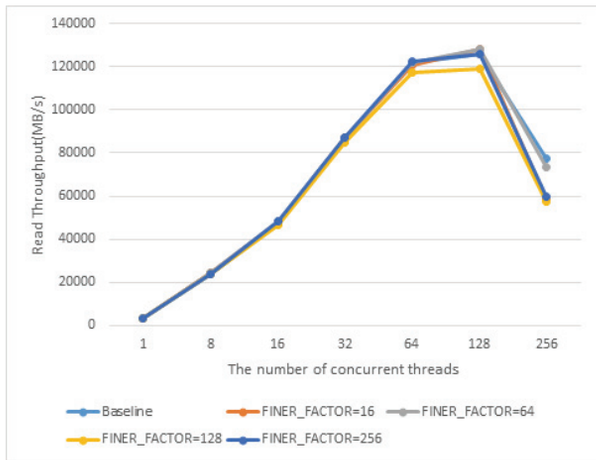


Fig. 8. Parallel Read Performance of KNL CPU Server as the Number of Concurrent Threads Increases in Various FINER_FACTOR Settings

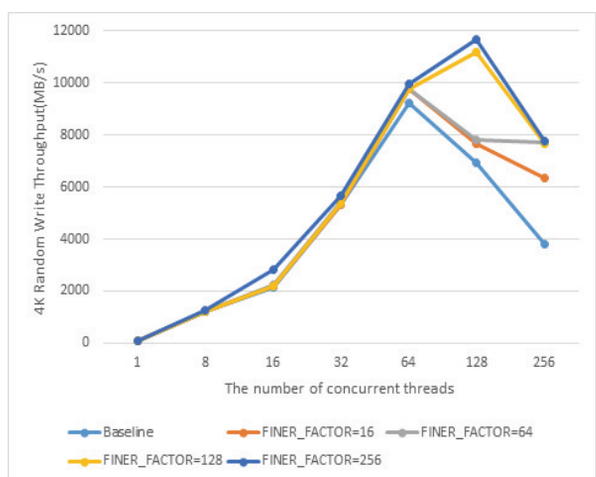


Fig. 9. Parallel Random Write Performance of KNL CPU Server as the Number of Concurrent Threads Increases in Various FINER_FACTOR Settings

병렬 읽기 연산에 대해서도 실험을 수행하였다. Fig. 8에서 볼 수 있는 것과 같이 읽기 성능은 baseline과 비교하여 크게 저하되지 않음을 확인하였다. 다만 쓰레드의 개수가 128,256으로 증가했을 때, 읽기 성능 또한 저하되는 것으로 보아 다른 하드웨어적인 병목이 원인이 있는 것으로 보인다.

마지막으로 여러 쓰레드가 동시에 4K 임의 쓰기를 수행할 때의 총 성능을 측정하였다. 기존 리눅스(baseline)의 경우 연속 쓰기와 마찬가지로 성능이 향상되다가 동시에 참여하는 쓰레드의 개수가 증가하자 성능이 다시 감소되는 것이 관측되었다. FinerLRU를 적용한 경우, 특히 쓰레드의 개수가 128, 256개로 매우 많은 경우 성능이 크게 증가함을 확인하였다. 다만 연속쓰기 때와 마찬가지로 256쓰레드가 동시에 쓰기를 수행할 때에는 성능이 감소하였다.

3.2 Intel Icelake Xeon CPU 에서의 성능 평가

최초에 FinerLRU의 개발은 KNL시스템에서의 I/O성능에 대한 연구를 진행하다가 병렬 쓰기 성능이 쓰레드의 개수가 증가할 때 오히려 성능이 감소하는 것을 확인한 것에서 시작하였다. 실험을 통해 KNL 시스템에서는 개선된 FinerLRU가 유의미한 성능 향상을 가져옴을 확인하였으나, KNL의 특성상 각 코어가 가지는 연산 능력이 범용 CPU의 코어보다 부족하기 때문에 발생하는 것일 수도 있다. 따라서 매니코어 시스템에서 병렬 I/O의 성능저하가 LRU lock 때문에 발생하고 FinerLRU를 적용하여 성능을 개선할 수 있는 것이 일반적인 것인지를 확인하지 위한 추가적인 실험을 수행하였다. 실험에 사용한 매니코어 CPU는 인텔 Icelake Xeon 프로세서로, 단일 코어의 성능은 KNL에 비해 훨씬 뛰어나고 범용이다. 실제 실험에 사용한 프로세서는 Interl Xeon Gold 6338 프로세서도 32개의 물리 코어와 64개의 논리 쓰레드로 구성되어 있고 각 코어는 2.0Ghz~3.2Ghz의 연산 속도를 가진다.

다른 매니코어 CPU를 가진 시스템에서 KNL에서 진행한 것과 동일하게 FINER_FACTOR와 동시에 I/O를 수행하는 쓰레드의 개수를 바꾸어가면서, 연속 쓰기의 성능 합, 전체 folio_add_lru 함수 수행시간 중 lruvec lock을 기다리는 시간의 비율, 병렬 읽기의 성능 합, 임의 쓰기의 성능 합을 측정하였다. 대신 쓰레드의 개수는 최대 64개로 하였고, 최신 리눅스의 특성상 LRU 갱신에 참여할 수 있는 쓰레드는 코어의 개수를 넘을 수 없기 때문에 FINER_FACTOR도 최대 64개로 설정하였다.

Fig. 10에서 나타난 것처럼 baseline의 경우 KNL과 마찬가지로 Icelake Xeon의 경우에도 쓰레드의 개수가 많아지면 병렬 쓰기의 성능이 올라가지 않고 오히려 저하되었다. 또한 FinerLRU를 적용하였을 때 병렬 I/O성능을 크게 향상 시킴을 확인하였다. 그리고 마찬가지고 논리 쓰레드 전체까지 사용하였을 때 FinerLRU를 적용해도 성능 향상에 한계가 있음을 확인하였다. 이는 KNL에서 측정한 것과 매우 유사한 패턴이다.

Fig. 11에서 나타난 것과 같이 전체 folio_add_lru 함수 수행시간 중 lruvec lock을 기다리는 시간의 비율 또한 KNL과 유사한 패턴을 보였다. 다만 최대 참여 쓰레드의 개수라 KNL이 훨씬 많았기 때문에 전체 비율로는 적어 보이나, 64 쓰레드가 참여한 경우 lock을 기다리는데 거의 대부분의 시간을 소모하는 것은 동일하였고, FINER_FACTOR를 크게 한 경우에는 그 비율이 10%남짓에 불과하였다. KNL과 마찬가지로 64쓰레드의 경우 lock대기 시간이 적은데 성능이 저하되는 것을 보면 다른 병목 지점이 있는 것으로 보인다.

읽기 성능의 경우 Fig. 12에서 나타나는 것처럼 거의 차이가 보이지 않았다. 즉 FinerLRU로 인한 오버헤드가 다른 I/O에 크게 영향을 미치지 않음을 나타낸다고 볼 수 있다.

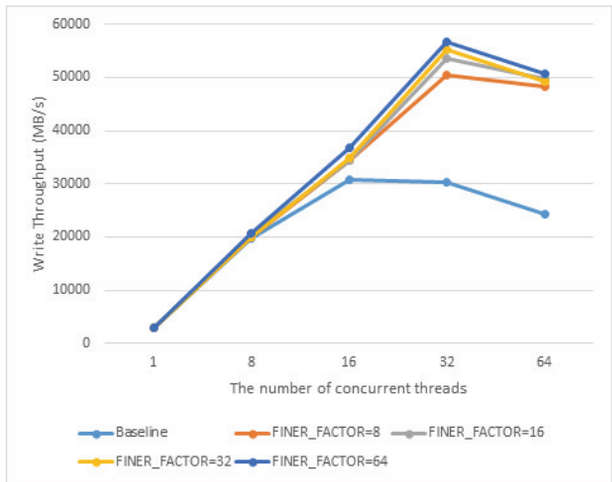


Fig. 10. Parallel Sequential Write Performance of Icelake Xeon server as the Number of Concurrent Threads Increases in Various FINER_FACTOR Settings

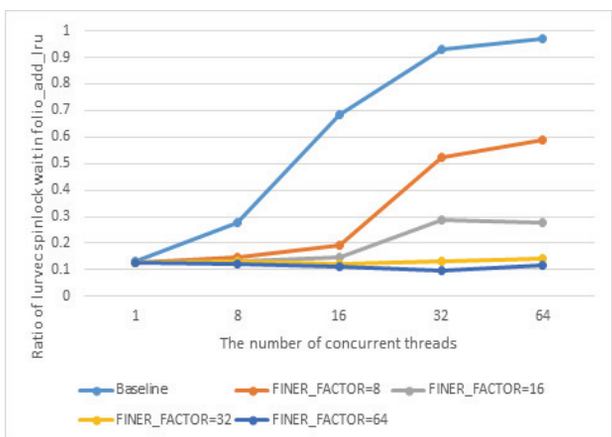


Fig. 11. The Ratio for Waiting Lruvec Spin Lock in folio_lru_add Funtion in Icelake Xeon CPU Server

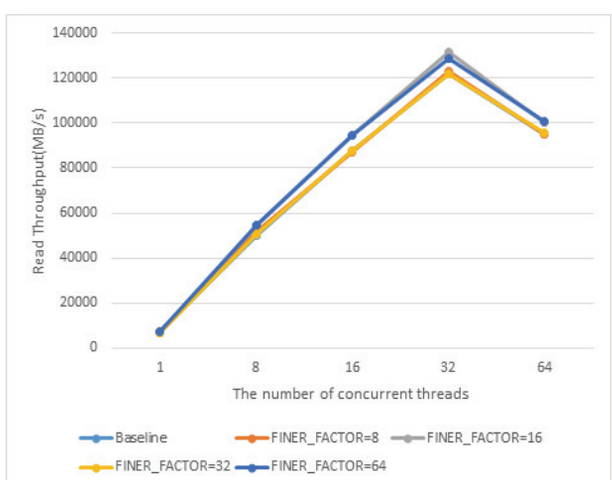


Fig. 12. Parallel Read Performance of Icelake Xeon Server as the Number of Concurrent Threads Increases in Various FINER_FACTOR Settings

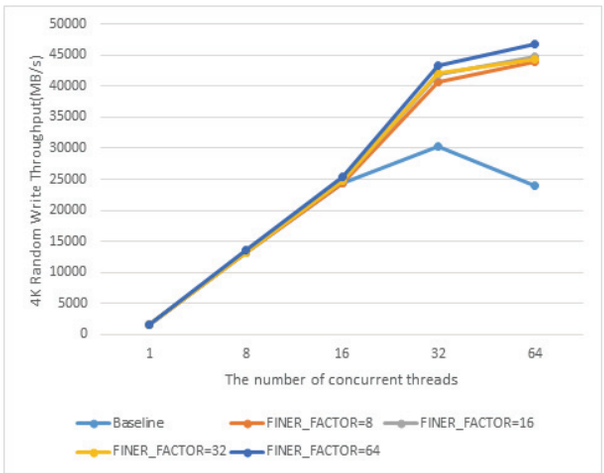


Fig. 13. Parallel Random Write Performance of Icelake Xeon CPU Server as the Number of Concurrent Threads Increases in Various FINER_FACTOR Settings

임의 쓰기의 경우에도 Fig. 13에서 보여지는 바와 같이 KNL과 비슷한 경향이 나타났다. baseline의 경우 스레드 증가에 따라 성능이 저하되고 FinerLRU를 사용한 경우 더욱 명확하게 병렬 쓰기 성능이 향상된다. 이 경우에는 쓰레드 64의 경우에도 성능이 저하되지 않았는데, 64개 이상의 쓰레드를 사용할 경우에도 성능이 저하되지 않는다고는 볼 수 없고 실험장비의 한계로 확인하지 못 하였다.

4. 결 론

이번 연구를 통해 Many-core CPU 시스템에서 수십 개 이상의 스레드가 병렬 쓰기를 수행하는 경우 기존 리눅스의 LRU lock방식이 성능 확장성을 보장하지 못하는 점을 보완하기 위해 개선된 FinerLRU를 제시하였다. 이를 리눅스 버전 5.18.11에 구현하였으며, 특성이 다른 두 가지 Many-core CPU, KNL CPU 탑재 서버와 Icelake Xeon CPU 탑재 서버에서 IOR 벤치마크 툴을 이용한 실험을 통해 큰 오버헤드 없이 병렬쓰기의 성능이 2배가량 증가시킬 수 있음을 확인하였다.

References

[1] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *ACM Computing Surveys*, Vol.48, No.3, 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] E. -K. Byun et al., "A study on optimizing LRU lock for improving parallel I/O throughout in manycore CPU systems," *Proceedings of the Annual Conference of Korea Information Processing Society Conference (KIPS) 2022*, Vol.29, No.2, pp.2-4, 2022.

[4] D. Zheng, R. Burns, and A. S. Szalay, "A parallel page cache: Iops and caching for multicore systems," in *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems, ser. HotStorage'12*. USA: USENIX Association, p.5, 2012.

[5] Y. Zhang, S. Shao, J. Zhai and S. Ma, "FineLock: Automatically refactoring coarse-grained locks into fine-grained locks," *In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp.565-568, 2020.

[6] 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*, pp.546-559, 2018.

[7] A. Sodani et al., "Knights landing: Second-generation intel xeon Phi product," in *IEEE Micro*, Vol.36, No.2, pp.34-46, 2016.

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

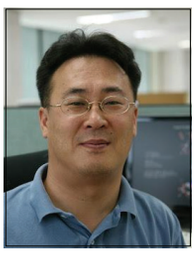
[9] TOP500, [Internet], https://www.top500.org/lists/top500

변 은 규

https://orcid.org/0000-0002-1811-9136
e-mail : ekbyun@kisti.re.kr
2003년 한국과학기술원 전산학과(학사)
2011년 한국과학기술원 전산학과(박사)
2011년 ~ 2012년 SK텔레콤 매니저
2013년 ~ 현 재 한국과학기술정보연구원 슈퍼컴퓨팅기술개발센터 선임연구원
관심분야 : 분산병렬시스템, 빅데이터, HPC, 스토리지 시스템

구 기 범

https://orcid.org/0000-0001-5313-123X
e-mail : gibeom.gu@kisti.re.kr
1997년 서강대학교 컴퓨터공학과(학사)
1999년 서강대학교 컴퓨터공학과(석사)
2002년 ~ 현 재 한국과학기술정보연구원 슈퍼컴퓨팅기술개발센터 책임연구원
관심분야 : 고성능컴퓨팅, 가속기 기반 알고리즘, 컴퓨터 그래픽스



오 광 진

https://orcid.org/0009-0005-0029-4015
e-mail : koh@kisti.re.kr
1992년 고려대학교 화학과(학사)
1996년 고려대학교 화학과(석사)
2000년 University of Nebraska – Lincoln 화학과(박사)
2003년 University of Pennsylvenia 화학과(박사후연구원)
2003년 ~ 현 재 한국과학기술정보연구원 슈퍼컴퓨팅기술개발센터 책임연구원
관심분야 : 슈퍼컴퓨팅, 계산과학

방 지 우

https://orcid.org/0000-0002-3556-2535
e-mail : cilioh14@gmail.com
2017년 서울대학교 컴퓨터공학부(학사)
2017년 ~ 현 재 서울대학교 컴퓨터공학부 석・박사통합과정
관심분야 : HPC, 분산파일시스템, 병렬컴퓨팅