https://doi.org/10.5392/JKCA.2018.18.04.178

멀티코어 및 고성능 플래시 SSD 환경에서 저널링 파일 시스템의 성능 평가 및 최적화

Performance Evaluation and Optimization of Journaling File Systems with Multicores and High-Performance Flash SSDs

한혁
동덕여자대학교 컴퓨터학과

Hyuck Han(hhyuck96@dongduk.ac.kr)

요약

최근 클라우드 컴퓨팅, 슈퍼컴퓨팅, 기업용 스토리지/데이터베이스 등의 분야에서 멀티코어 CPU와 고성능 플래시 메모리 기반 저장 장치(플래시 SSD)를 장착한 컴퓨터 시스템에 대한 수요가 크게 증가하고 있다. 이러한 고성능 시스템에서 구동되고 있는 대표적인 운영체제 파일 시스템인 저널링 파일 시스템은 저장 장치의 입출력 성능을 최대로 활용하고 있지 못하다. 본 논문에서는 고성능 플래시 SSD와 멀티코어 CPU 기반의 컴퓨터 시스템에서 리눅스 운영체제의 EXT4 저널링 파일 시스템의 성능을 평가하고 분석하고자 한다. 성능 평가에 사용된 72-코어 컴퓨터 시스템은 인텔의 고성능 NVMe 기반 플래시 SSD를 장착하고 있으며 이 장치의 연속 읽기/쓰기 성능은 2800/1900 MB/s 이다. 실험 결과는 EXT4 파일 시스템의 체크포인팅 연산이 성능상의 큰 오버헤드임을 보여준다. 이 결과를 바탕으로 체크포인팅을 여러 쓰레드가 수행할 수 있는 최적화 기법을 제안하였고, 최적화된 EXT4 파일 시스템은 기존 EXT4 파일 시스템 대비 최대 92%의 성능 향상을 보여준다.

■ 중심어 : | 플래시 SSD | 멀티코어 | 저널링 파일 시스템 |

Abstract

Recently, demands for computer systems with multicore CPUs and high-performance flash-based storage devices (i.e., flash SSD) have rapidly grown in cloud computing, super-computing, and enterprise storage/database systems. Journaling file systems running on high-performance systems do not exploit the full I/O bandwidth of high-performance SSDs. In this article, we evaluate and analyze the performance of the Linux EXT4 file system with high-performance SSDs and multicore CPUs. The system used in this study has 72 cores and Intel NVMe SSD, and the flash SSD has performance up to 2800/1900 MB/s for sequential read/write operations. Our experimental results show that checkpointing in the EXT4 file system is a major overhead. Furthermore, we optimize the checkpointing procedure and our optimized EXT4 file system shows up to 92% better performance than the original EXT4 file system.

■ keyword : | Flash SSD | Multicore | Journaling File System |

* 이 논문은 2017년도 동덕여자대학교 학술연구비 지원에 의하여 수행된 것임.
접수일자 : 2018년 02월 22일
심사완료일 : 2018년 03월 14일
수정일자 : 2018년 03월 14일
교신저자 : 한혁, e-mail : hhyuck96@dongduk.ac.kr

I. 서 론

최근 고성능 플래시 메모리 기반 저장 장치(플래시 SSD)와 멀티코어 CPU를 구비한 서버 시스템은 데이터센터, 사회 관계망 서비스, 기업용 데이터베이스/스토리지 영역에 널리 채택되어 사용되고 있다. 플래시 SSD 기반 스토리지는 기존의 하드디스크에 비해 수십~수백 배 빠른 지연시간과 향상된 대역폭을 보여주며, 멀티코어 CPU는 동시에 수행할 수 있는 작업의 수를 증가시켜 시스템의 성능을 향상시켜준다.

그러나 최근 연구에 [1-3] 따르면 이러한 고성능의 컴퓨터 시스템에서 동작하고 있는 운영체제의 저널링 파일 시스템들이 저장 장치의 입출력 성능을 최대로 활용하지 못한다. 이러한 성능 문제를 분석하기 위해 본 논문은 PCIe 기반의 고성능 NVMe 플래시 SSD와 18코어 CPU 4개를 (총 72-코어) 탑재한 서버 시스템에서 리눅스 EXT4의 성능을 평가하고 분석한다. 본 연구에서 사용한 저장 장치는 Intel의 고성능 NVMe 플래시 SSD이며 4KB 입출력 요청 크기일 때 최대 연속 읽기/쓰기 성능 2800/1900 MB/s이다. 이는 기존 SATA 인터페이스 기반 플래시 SSD의 성능 (최대 읽기 성능 540MB/s, 쓰기 성능 520MB/s)에 비해 읽기는 5.3배, 쓰기는 3.6배 좋은 성능을 보인다. 실험 결과는 멀티코어 CPU와 고성능 플래시 SSD를 이용하더라도 EXT4 파일 시스템의[4] 체크포인팅 연산이 성능상의 큰 오버헤드임을 보여준다. 이것은 EXT4 파일 시스템이 체크포인팅을 수행할 때 서버 시스템이 멀티코어 CPU를 탑재하고 있음에도 하나의 쓰레드만이 저장 장치와의 체크포인팅 입출력 연산을 수행하기 때문이다.

이러한 실험 결과를 바탕으로 EXT4 파일 시스템의 체크포인팅 수행 부분을 최적화하였다. 기존 EXT4 파일 시스템은 체크포인팅 연산을 수행할 때 하나의 쓰레드가 체크포인트 버퍼 블록 리스트에 접근하여 입출력 연산을 수행하지만 최적화된 EXT4 파일 시스템은 여러 쓰레드가 입출력 연산을 수행할 수 있게 하여 성능 향상을 달성한다. 실험을 통해 제안된 방법이 안전하게 체크포인팅을 수행하면서 기존 EXT4 파일 시스템 대비 최대 92%의 성능 향상을 보여준다.

본 논문의 나머지는 다음과 같이 구성된다. 2장은 본 연구와 관련된 연구와 본 논문의 배경이 되는 리눅스 저널링 및 체크포인팅에 대해 설명한다. 3장은 리눅스 EXT4 파일 시스템의 성능 평가 결과를 분석과 함께 설명한다. 4장에서는 3장의 결과를 바탕으로 최적화된 체크포인팅 기법과 그 효과를 설명한다. 마지막으로 5장에서는 논문의 결론을 내린다.

II. 관련 연구 및 배경

1. 관련 연구

최근에 플래시 메모리 기반 저장 장치의 성능을 평가하고 소프트웨어 수준에서 최적화하는 연구들이 진행되어 왔다. [5]의 연구에서는 플래시 SSD에 최적화된 트랜잭션 처리 방법을 제안하였고, 이를 통해 하드디스크 시스템 대비 수십 배 이상의 트랜잭션 처리 성능 향상을 가져올 수 있음을 보였고, [6]에서는 빅데이터 처리 시스템에서 사용되는 하둡 파일 시스템을 플래시 SSD과 하드 디스크를 이용하여 평가하였다. [7]의 연구에서는 리눅스 파일 시스템을 NVMe 기반 플래시 SSD를 탑재한 컴퓨터 시스템에서 성능을 평가하였다. 그러나 멀티코어 CPU까지 고려한 성능 분석은 이루어지지 않았다.

Silo[8] 및 SiloR[9]은 멀티코어 CPU에 최적화된 메인메모리 데이터베이스 시스템이며, 데이터베이스 시스템 내부의 모든 소프트웨어 컴포넌트들이 멀티코어에 확장적으로 설계되었다. [10-12]의 연구에서는 멀티코어 CPU 환경에서 운영체제 내부의 (가상메모리, 가상머신 등) 공유 자료구조의 경합을 최적화하였다. 이러한 연구들은 데이터베이스/운영체제 시스템을 멀티코어에 확장적으로 최적화하였다는 점에서 본 연구와 유사하지만 본 연구는 저널링 파일 시스템에 초점이 맞추어져있다. [3]의 연구에서는 리눅스 파일 시스템의 멀티코어 상의 성능 병목을 분석하였고, [1]의 연구에서는 멀티코어 및 고성능 플래시 SSD의 성능을 최대로 활용하기 위해 복수 개의 가상의 스토리지를 생성한다. 이러한 연구들은 스토리지 소프트웨어 스택의 성능 병목

을 분석하거나 최적화하였다는 점에서 본 연구와 유사하다. 그러나 본 연구는 성능 병목을 해소하기 위해 새로운 종류의 파일 시스템 및 스토리지 시스템을 설계하기보다는 기존의 EXT4 파일 시스템에 이식 가능하며, 적용 가능성이 높은 방법을 제안한다.

2. 저널링 파일 시스템

파일 시스템의 트랜잭션은 원자적 및 영속적이어야 하는 파일 시스템 변경 사항의 집합을 의미한다[13][14]. 트랜잭션 처리의 원자성 및 영속성을 보장하기 위해 많은 파일 시스템들이 저널링 방법을 사용한다. 저널링 파일 시스템들은 파일 시스템의 변경 사항을 디스크 상의 원래 위치에 반영하기 전에 로그라는 단위로 만들어 변경 사항들을 저널 영역에 쓴다. 이러한 작업이 끝나면 (트랜잭션이 커밋되면), 변경 사항을 디스크 상의 원래 위치에 쓰며 이것을 체크포인팅이라고 한다. 이러한 방법을 이용하여 저널링 파일 시스템들은 시스템의 장애 상황이 발생하더라도 크래쉬-일관성을 응용에 제공할 수 있다[14].

본 논문은 EXT4 파일 시스템이 다른 저널링 파일 시스템들보다 일반적으로 많이 사용되기 때문에 EXT4 파일 시스템에 초점을 맞춘다. EXT4 파일 시스템은 트랜잭션을 처리하기 위해 저널링 블록 디바이스2(JBD2)를 사용한다. EXT4 파일 시스템이 제공하는 저널링 모드는 write-back, ordered 및 data 저널링 모드이다. JDB2는 단독 트랜잭션 모델에 기반하고 있다. 즉, 유일한 1개의 활성 트랜잭션이 파일 시스템의 변경 사항을 처리하는 구조이다. 활성 트랜잭션은 변경된 블록을 가리키는 포인터들을 리스트로 유지하고 있다. 주기적인 커밋 연산 혹은 fsync() 연산이 호출되면 활성 트랜잭션의 상태는 “커밋중”이라는 상태로 변경되며 트랜잭션에 유지하고 있는 변경된 블록들을 모두 저널 영역에 쓴다. 이 작업이 끝나면 트랜잭션의 상태는 체크포인트가 가능한 상태로 변경되며, 이후에 저널 영역 확보를 위해 변경된 블록들을 디스크 상의 원래 위치에 쓴다 (체크포인팅).

III. 문제점 분석

1. 실험 환경

이번 장에서는 고성능 플래시 SSD 및 멀티코어 CPU를 탑재한 컴퓨터 시스템에서 EXT4 파일 시스템의 성능을 평가하기 위한 실험 환경에 대해 설명한다. 성능 평가를 위해 4개의 Intel Xeon CPU E7-8870 CPU를 장착한 서버를 사용하였다. 이 서버 시스템은 72개의 코어와 16GB 메인 메모리를 가지고 있으며, 이 시스템에 Linux 커널 4.9.1 버전에 기반하는 Ubuntu 16.04.01 LTS를 구동하였다. 본 실험에서 사용되는 저장 장치는 최근 Intel이 개발한 NVMe 기반 플래시 메모리 SSD이며 이 장치는 용량이 800GB이며, 4KB요청 크기일 때 최대 읽기/쓰기 성능이 2800/1900 MB/s를 보여준다.

EXT4의 저널링 모드는 ordered와 data 저널링 모드로 하여 각각 실험을 진행하였고, 실험에 사용한 벤치마크는 Tokubench, Sysbench, Filebench의 Varmail, Filebench의 Fileserver이며 각 벤치마크의 파라미터는 [표 1]과 같다. Tokubench와 Varmail은 ordered 모드에서, Sysbench와 Fileserver는 data 저널링 모드에서 실험을 수행하였다. Ordered 모드 상에서는 파일 시스템의 메타데이터 변경만을 트랜잭션으로 관리하여 저널링을 수행하며, data 저널링 모드에서는 메타데이터와 데이터 모두의 변경을 트랜잭션으로 관리하여 저널링을 수행한다. 각각의 실험은 CPU 코어의 수를 변경해 가면서 수행하였고, 코어의 수와 동일한 수의 응용 쓰레드가 벤치마크를 수행하도록 하였다.

표 1. 벤치마크 파라미터

벤치마크	파라미터
Tokubench (microbenchmark)	파일 : 30,000,000개, 입출력 단위 : 4KB
Sysbench (microbenchmark)	파일: 1GB 파일 72개 입출력 단위 : 4KB
Filebench Varmail (macrobenchmark)	파일 : 300,000개 디렉토리 : 10,000
Filebench Fileserver (macrobenchmark)	파일 : 1,000,000개 디렉토리 : 10,000

2. 성능 평가 결과 및 분석

[그림 1]은 EXT4 파일 시스템의 저널링 모드를 ordered로 하였을 때의 결과를 보여준다. 두 개의 벤치마크 모두 코어의 수에 확장적이지 않음을 확인할 수 있다. 특히 Varmail의 경우에는 36 코어에서는 690MB/s이지만 72 코어에서는 450MB/s로 성능이 오히려 하락한다. Tokubench의 경우에는 전체적으로 플래시 SSD의 최대 성능의 5.6% 정도만 사용하고 있으며 4 코어일 때 107MB/s의 최대 성능을 보이며 4 코어 이상에서 더 이상 증가하지 않는다.

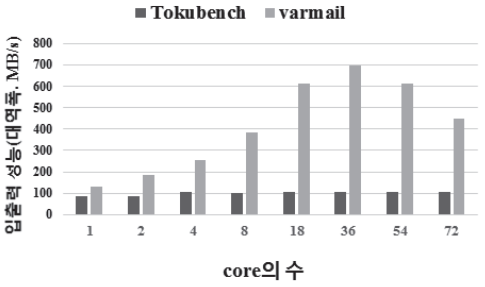


그림 1. Ordered 모드 실험 결과

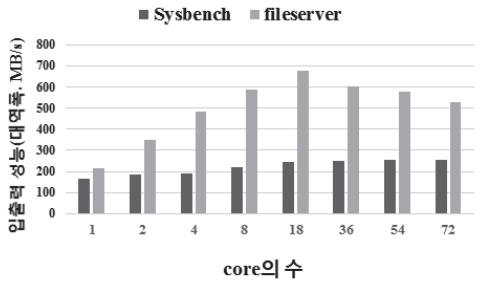


그림 2. Data 저널링 모드 실험 결과

[그림 2]는 EXT4 파일 시스템의 data 저널링 모드에서의 벤치마크 실험 결과를 보여준다. Fileserver의 경우에는 18 코어일 때 680MB/s 정도의 성능을 보이지만 72 코어일 때는 530MB/s 정도로 성능 역전 현상이 나타난다. Sysbench의 경우에는 36 코어일 때 최고 성능을 (255MB/s) 보인다.

위와 같은 현상의 주요 원인은 1개의 쓰레드가 체크포인팅 연산을 처리하는 것이다. [그림 3]은 EXT4 파일 시스템의 체크포인팅 처리 과정을 보여준다. Thread 1/2/3은 응용 프로그램의 쓰레드이며 커널 모드에서 시스템 호출 처리 중에 로그 공간을 확보하기 위해 체크포인팅을 수행하려고 하는 쓰레드들이며, Thread 4는 저널 영역에는 이미 반영된 파일 시스템 변경을 취소하기 위해 체크포인트 리스트에 접근하는 커널 쓰레드이다. 체크포인팅을 수행하고자하는 쓰레드들은 모두 j_checkpoint_mutex를 얻으려고 하고 여러 쓰레드들 중에서 하나의 쓰레드만이 체크포인팅을 수행한다 (그림 예에서는 Thread 2). Thread 4는 체크포인팅을 수행하는 쓰레드가 아니기 때문에 j_checkpoint_mutex를 얻지 않고 체크포인트 리스트에 접근하려고 한다. 체크포인트 리스트는 j_list_spinlock의 보호 하에 접근이 원자적으로 이루어지며, [그림 2]에서는 Thread 2와 Thread 4가 경합하여 체크포인트 리스트에 접근할 수 있다. 최종적으로 Thread 2만이 체크포인트 리스트에서 변경된 블록을 지시하는 포인터를 읽어서 파일 시스템의 변경 사항들을 실제 디스크 위치에 쓴다. 이러한 처리 방법은 서버 시스템이 멀티코어 CPU를 탑재하였을지라도 하나의 입출력 쓰레드만이 체크포인팅 연산을 수행하므로 고성능 플래시 SSD의 성능을 최대로 이용할 수 없게 한다.

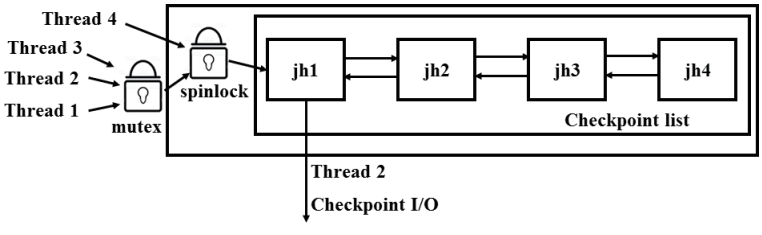


그림 3. EXT4 파일 시스템의 체크포인팅 처리 (jh: 변경된 블록을 지시하는 포인터)

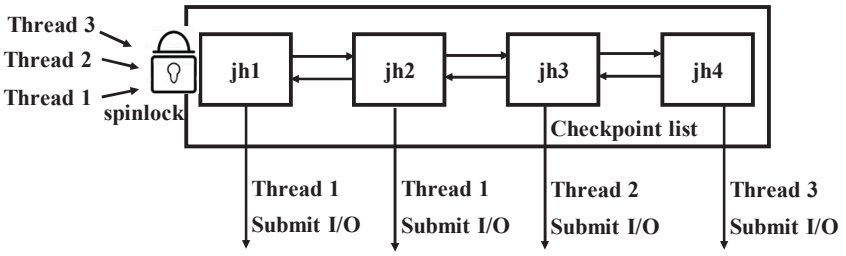


그림 4. 최적화된 체크포인팅 쓰기 연산 처리 (jh: 변경된 블록을 지시하는 포인터)

이를 검증하기 위해 Sysbench 벤치마크의 실행 시간을 분석하였다. 72 코어일 때 전체 벤치마크 수행시간은 52220초이며 그 중 j_checkpoint_mutex로 인한 대기 시간이 17940초 정도로 전체 수행 시간의 34% 정도를 차지하였다. 즉, 72개의 Sysbench 응용 쓰레드들이 전체 수행 시간의 34% 정도를 체크포인팅을 수행하거나 수행이 끝나기를 기다리는데 소비한다. 이와 같은 결과는 멀티코어 CPU와 고성능 플래시 SSD를 탑재한 컴퓨터 시스템에서 저널링 파일 시스템의 성능 향상을 위해서 체크포인팅 연산의 최적화가 필수적임을 알 수 있다.

IV. 최적화 기법 및 성능 평가

1. 최적화된 체크포인팅

이 장에서는 앞서 분석한 것과 같이 고성능 플래시 SSD와 멀티코어 CPU를 탑재한 시스템에서 EXT4 파일시스템의 성능 상의 큰 오버헤드인 체크포인팅 처리를 최적화하는 기법에 대해 설명한다. 기존 EXT4 파일 시스템에서 j_checkpoint_mutex에 의해 보호되어 하나의 쓰레드만이 체크포인팅을 위한 입출력 연산을 수행하는 것이 고성능 저장 장치의 최대 성능을 활용하지 못하는 원인이 된다.

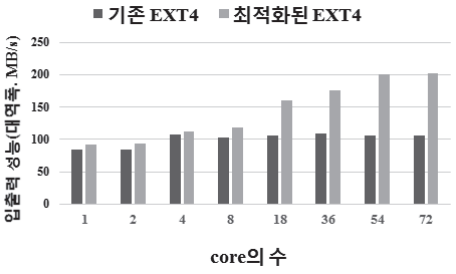
최적화된 체크포인팅 처리에서는 고성능 저장 장치의 성능을 최대로 활용하기 위해 단일 쓰레드가 아닌 여러 쓰레드가 체크포인팅 입출력 연산을 수행하도록 한다. 즉, j_checkpoint_mutex로 보호되는 구간을 가능한 여러 쓰레드가 수행하도록 하는 것이다. 이러한 설계를 구현하기 위해서는 크게 다음과 같은 두 가지 사항을 고려하여야 한다. 첫째, 단일 쓰레드가 입출력 연산을 처리하는데 적합하도록 설계된 기존의 중앙 집중적인 자료구조를 여러 쓰레드가 처리할 수 있도록 해야 한다. 둘째, 체크포인팅 연산을 여러 쓰레드가 동시에 수행하더라도 체크포인팅 연산 후에 체크포인트 트랜잭션 및 리스트 관리는 안전하게 원자적으로 해야 한다.

위에서 제시한 고려 사항을 충족하기 위해서 [그림 4]와 같이 최적화된 체크포인팅 처리에서는 여러 쓰레드가 체크포인팅 처리를 할 수 있도록 j_checkpoint_mutex를 제거하지만 checkpoint list에 대한 접근을 원자적으로 해주는 spinlock (j_list_lock)은 그대로 유지하여 체크포인팅을 위한 입출력 버퍼를 여러 쓰레드가 안전하게 얻을 수 있게 한다. 그리고 체크포인팅에 참여하는 각각의 쓰레드들이 동시에 쓰기 연산을 요청하고 (알고

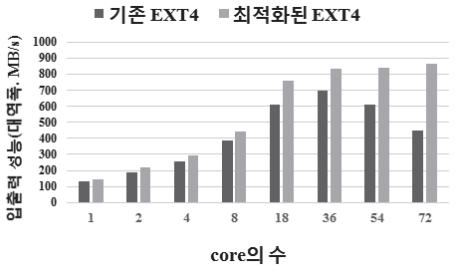
```
1: jbd2_log_wait_for_space(journal){
2:     if((transaction = journal->j_checkpoint_transactions) == NULL)
3:         return;
4:     atomic_add(transaction->num_threads, 1);
5:     create_wait_list(local_wait_list); // create a local wait list per thread
6:     spin_lock(journal->j_list_lock);
7:     while((jh = transaction->t_checkpoint_list) != NULL){
8:         l_bh = jh->bh;
9:         transaction->t_checkpoint_list=jh->next;
10:        spin_unlock(journal->j_list_lock);
11:        submit_bh(WRITE, l_bh);
12:        add_wait_list(local_wait_list, l_bh);
13:        spin_lock(journal->j_list_lock);
14:    }
15:    spin_unlock(journal->j_list_lock);
16:    wait_cp_io(local_wait_list);
17:    if(atomic_sub(transaction->num_threads, 1) == 0){
18:        spin_lock(journal->j_list_lock);
19:        <set the next transaction to be checkpointed
20:                in the checkpoint transaction list>
21:        spin_unlock(journal->j_list_lock);
22:    }
23: }

24: wait_cp_io(local_wait_list){
25:    while(!wait_list_empty(local_wait_list){
26:        bh = list_entry(local_wait_list.next, ...);
27:        wait_on_buffer(bh);
28:    }
29: }
```

알고리즘 1. 최적화된 체크포인팅 알고리즘

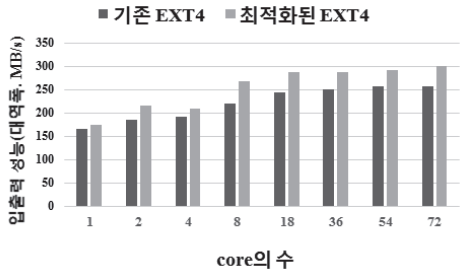


(a) Tokubench 결과

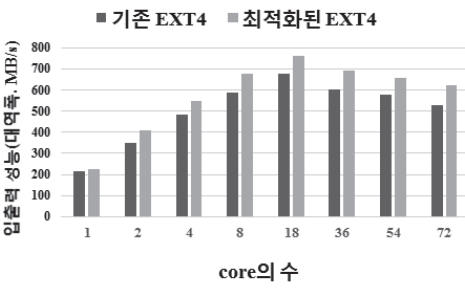


(b) Varmail 결과

그림 5. Ordered 모드 결과



(a) Sysbench 결과



(b) Fileserver 결과

그림 6. Data 저널링 모드 결과

리즘1의 11번째 줄) 쓰기 연산에 대한 응답을 기다려주는 wait list를 쓰레드마다 유지하게 하여 쓰기 연산에 대한 병렬성을 높여준다. (알고리즘 1의 5, 12번째 줄 및 wait_cp_io 함수)

또한, 모든 쓰기 연산이 끝난 후에 처리해야할 체크포인팅을 해야 할 다음 트랜잭션 지정과 같은 일은 가장 마지막에 쓰기 연산을 수행한 쓰레드가 수행하도록 한다. 이와 같이 처리하게 하면 기존 체크포인팅 처리가 제공하는 논리적 의미를 훼손하지 않으면서 저장 장치의 성능을 최대로 이용할 수 있다. 이것을 구현하기 위해서는 체크포인팅에 참여하는 모든 쓰레드들이 jbd2_log_wait_for_ space 함수에 진입하면 체크포인트에 참여하고 있는 쓰레드의 수를 지시하는 변수를 원자적으로 증가키시고 (알고리즘 1의 4번째 줄) 참여 쓰레드들이 쓰기 연산을 완료하면 이 값을 다시 원자적으로 감소시킨다 (알고리즘1의 17번째 줄). 그리고 그 결과가 0인 쓰레드가 마지막으로 쓰기 연산을 완료한 쓰레드이기 때문에 이 경우에만 체크포인팅을 해야 할 다음 트랜잭션 지정과 같은 후 처리 작업을 수행한다.

2. 성능 평가

체크포인팅 처리 최적화 기법의 효과를 알아보기 위해 3절의 실험 환경에서 기존 EXT4와 최적화된 EXT4 파일 시스템 상에서 벤치마크들을 수행하였다. [그림 5]는 Tokubench와 Varmail 프로그램을 ordered 모드에서 수행하였을 때의 성능 변화를 보여준다. [그림 5(a)]와 같이 Tokubench에서는 최적화된 체크포인팅을 수행하는 EXT4 파일 시스템이 72 코어일 때 202 MB/s의 최대 성능을 보이며 기존 EXT4 파일 시스템 대비 90% 정도의 성능 향상을 보였다. 또한 기존 EXT4 파일 시스템의 경우 4코어 이후에 성능이 수렴하는 데 비해 최적화된 EXT4 파일 시스템은 54 코어 이후에 성능이 수렴하여 전체적으로 코어의 수에 따라 성능이 확장적으로 증가하는 양상을 보여주었다. [그림 5(b)]는 Varmail의 결과이다. 최적화된 EXT4 파일스템은 72 코어일 때 864 MB/s의 최대 성능을 보이며 기존 파일 시스템 대비 92%의 성능 개선을 보여주었다. 기존 EXT4 파일 시스템의 36 코어 이상에서 나타났던 성능 역전 현상이 최적화된 EXT4 파일 시스템에서는 나타나지 않아 코

어의 수에 확장적으로 성능이 개선되었다. 그러나 쓰레드의 수가 적은 경우 성능 향상 효과가 작다. 벤치마크를 수행하는 쓰레드의 수가 적으면 벤치마크를 수행하는 중에 처리해야하는 입출력 연산의 양이 적어지게 되고, 이로 인한 체크포인팅을 수행하는 횟수가 줄어들기 때문이다.

[그림 6]은 data 저널링 모드에서의 Sysbench 및 Fileserver 벤치마크 실험 결과이다. [그림 6(a)]와 같이 Sysbench에서는 최적화된 체크포인팅을 수행하는 EXT4 파일 시스템이 72 코어일 때 300 MB/s의 최대 성능을 보이며 기존 EXT4 파일 시스템 대비 16% 정도의 성능 향상을 보였다. [그림 6(b)]는 Fileserver 벤치마크 실험 결과이다. 최적화된 EXT4 파일스템은 18 코어일 때 620 MB/s의 최대 성능을 보이며 기존 파일 시스템 대비 17%의 성능 개선을 보여주었다. Sysbench 및 Fileserver 벤치마크 모두 ordered 모드 실험 결과보다는 적은 성능 향상 폭을 보여주었다. 이것은 최적화된 체크포인팅 처리에서 j_ checkpoint_mutex는 제거하여 여러 쓰레드가 동시에 체크포인팅 작업을 수행하도록 하였지만 j_list_lock은 여전히 남겨져 있다. 3절의 분석 결과와 유사하게 72 코어일 때, 최적화하지 않은 시스템의 경우에 전체 시간의 33% 정도를 체크포인팅을 하는데 소비하고 10% 정도를 j_list_lock을 얻는데 소비하고 있다. 즉, 블록 버퍼 리스트 접근을 보호하고 있는 j_list_lock이 활성 트랜잭션 처리, 트랜잭션 커밋 연산에서도 사용되어 락 경쟁을 유발하고 있기 때문에 큰 폭의 성능을 보이지 못 하며 Fileserver의 결과와 같이 18 코어 이상일 때 최적화된 EXT4 파일 시스템 역시 성능 역전 현상을 보여주는 것으로 분석되었다.

V. 결론

본 논문에서는 여러 IT 분야에서 많이 활용되고 있는 고성능 플래시 SSD와 멀티코어 CPU를 탑재한 컴퓨터 시스템에서 저널링 파일 시스템의 성능을 평가하고 분석하였다. 현재 산업계 및 학계에서 가장 많이 사용되고 있는 저널링 파일 시스템인 EXT4 파일 시스템을 ordered 및 data 저널링 모드에서 4개의 벤치마크를 이용하여 성능을 평가하였다. 실험 결과는 멀티코어 CPU를 탑재한 시스템이라도 기존 EXT4 파일 시스템의 체크포인팅 연산이 단일 쓰레드에 의해 수행되기 때문에 고성능 저장 장치의 성능을 최대로 이용하지 못하는 것으로 분석되었다. 이를 해결하기 위해 다수의 쓰레드가 체크포인팅 연산에 참여하여 쓰기 연산을 수행하는 최적화 기법을 제안하고 구현하였으며 이 기법이 최대 92% 정도 성능을 향상시킬 수 있음을 보였다. 향후 연구에는 활성 트랜잭션 처리, 트랜잭션 커밋, 체크포인팅에서 발생되는 블록 버퍼 리스트 보호로 인한 락 경쟁을 완전히 제거할 수 있는 기법을 제안하고자 한다.

참 고 문 헌

[1] Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai, “MultiLanes: Providing Virtualized Storage for OS-level Virtualization on Many Cores,” In Proceedings of the 12th USENIX Conference on File and Storage Technologies(FAST 14), 2014.

[2] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai, “SpanFS: A Scalable File System on Fast Storage Devices,” In Proceedings of the 2015 Annual Technical Conference (ATC 15), 2015.

[3] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim, “Understanding Manycore Scalability of File Systems,” In Proceedings of the 2016 Annual Technical Conference (ATC 16), 2016.

[4] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, “The new ext4 filesystem: Current status and future plans,” In Proceedings of the Linux Symposium, 2007.

[5] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim, “A case

for flash memory ssd in enterprise database applications," In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08), 2008.

[6] Sangwhan Moon, Jaehwan Lee, Xiling Sun, and Yang-Suk Kee, "Optimizing the Hadoop MapReduce Framework with high-performance storage devices," Journal of Supercomputing, Vol.71, No.9, pp.3525-3548, 2015.

[7] Yongseok Son, Hara Kang, Hyuck Han, and Heon Young Yeom, "An empirical evaluation and analysis of the performance of NVM express solid state drive," Cluster Computing, Vol.19, No.3, pp.1541-1553, 2016.

[8] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden, "Speedy transactions in multicore in-memory databases," In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP 13), 2013.

[9] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov, "Fast Databases with Fast Durability and Recovery Through Multicore Parallelism," In Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI 14), 2014.

[10] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang, "A case for scaling applications to many-core with OS clustering," In Proceedings of the sixth conference on Computer systems (EuroSys 11), 2011.

[11] 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), 2010.

[12] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich, "RadixVM: scalable address spaces for multithreaded applications," In Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys 13), 2013.

[13] A. R. Hagmann, "Reimplementing the Cedar file system using logging and group commit," SIGOPS Oper. Syst. Rev., Vol.21, No.5, pp.155-162, 1987.

[14] Daejun Park and Dongkun Shin, "iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call," In Proceedings of the 2017 Annual Technical Conference (ATC 17), 2017.

저 자 소 개

한 혁(Hyuck Han) 정회원

- 2003년 8월 : 서울대학교 컴퓨터공학부(공학사)
- 2006년 2월 : 서울대학교 컴퓨터공학부(공학석사)
- 2011년 2월 : 서울대학교 컴퓨터공학부(공학박사)
- 2011년 3월 ~ 2012년 8월 : 서울대학교 컴퓨터공학부 박사후 연구원
- 2012년 9월 ~ 2014년 2월 : 삼성전자 메모리 사업부 책임연구원
- 2014년 3월 ~ 현재 : 동덕여자대학교 컴퓨터학과 조교수

<관심분야> : 데이터베이스 시스템, 병렬 프로그래밍, 분산 시스템