로그 구조 파일 시스템에서 불필요한 저장장치 공간 낭비를 줄이기 위한 이중 로깅 기법

곽현호°, 신동군
성균관대학교 전자전기컴퓨터공학과
gusghrhkr@skku.edu, dongkun@skku.edu

Two-Level Logging Technique to Reduce Storage Space Waste in Log-Structured File Systems

Hyunho Gwak°, Dongkun Shin
Department of Electrical and Computer Engineering, Sungkyunkwan University

요 약

플래시 저장장치의 사용이 대중화되면서 플래시 저장장치에서 많이 사용되는 로그 구조 파일 시스템에 대한 관심도 높아지고 있다. 그런데, 플래시 저장장치는 블록 저장장치이므로, 블록 내의 작은 양의 바이트들이 업데이트 되어도 블록 단위로 저장장치 공간을 소모하여 로그 구조 파일 시스템의 가비지 컬렉션(Garbage Collection)을 가속화시키는 문제가 있다. 본 논문에서 제안하는 이중 로깅 기법은 로그 구조 파일 시스템에서 바이트 로깅을 사용하여 저장장치 공간 낭비를 줄이는 기법이다. 또한, 비 휘발성 메모리를 바이트 로깅 영역으로 활용함으로써 바이트 로깅을 효과적으로 사용하는 기법도 함께 제안한다. 실험 결과 기존 로그 구조 파일 시스템에 비해 저장장치로의 쓰기 양을 35% 감소시켰고, 성능 또한 현저히 증가한 것을 확인하였다.

1. 서 론

최근 SSD나 SD 카드와 같은 플래시 저장장치의 사용이 대중화되고 있다. 플래시 저장장치는 저전력, 높은 내구성 등의 특성을 가지고 있고, 특히 기존 주 저장장치로 사용되었던 하드디스크 대비 높은 성능을 보여준다. 때문에 플래시 저장장치는 임베디드 환경이나 서버 시스템에서 주 저장장치로 많이 사용되고 있다. 하지만, 하드 디스크나 플래시 저장장치 같은 전통적인 블록 저장장치들은 읽고 쓰는 것이 블록 단위로만 가능하다. 일반적으로 블록의 크기는 4KB이기 때문에 몇 바이트의 데이터를 기록하기 위해서도 4KB의 쓰기가 발생하게 된다. 우리는 SQLite를 이용한 mobibench [1]를 사용하여 얼마나 많은 불필요한 쓰기가 발생하는지 관찰하였다. 리눅스 커널의 로그 구조 파일 시스템에서 분석한 결과, 전체 쓰인 블록의 약 45%가 블록의 일부만 변경되었으며 이로 인해 약 32%의 불필요한 쓰기가 발생한 것을 확인하였다. 이러한 불필요한 쓰기는 저장장치의 쓰기 공간을 실제 기록한 데이터의 양보다 더 소모시킨다. 특히, 불필요한 쓰기로 인한 저장장치 공간 소모는 플래시 저장장치 내부의 가비지 컬렉션 (Garbage Collection)을 유발시킬 수 있고, 결과적으로 플래시 저장장치의 성능과 수명을 감소시킨다.

이와 같은 불필요한 쓰기로 인한 문제를 해결하기 위해 바이트 로깅 기법들이 제안되었다. 바이트 로깅은 블록에서 변경된 바이트들만 로깅하여 저장장치에 기록함으로써 불필요한 쓰기를 줄이는 기법이다. 기존의 바이트 로깅 연구들은 EXT4 파일 시스템의 저널링 영역에 바이트 로깅을 지원하는 기법을 사용하였다 [2, 3]. 그런데, 불필요한 쓰기 문제는 EXT4 파일 시스템뿐만 아니라 로그 구조 파일 시스템 [4]에서도 발생한다. 특히, 로그 구조 파일 시스템은 플래시 저장장치와 같이 다른 자리 (Out-Of-Place) 업데이트 방식을 사용하기 때문에 파일 시스템에서 가비지 컬렉션을 수행해야 한다. 따라서, 플래시 저장장치처럼 불필요한 쓰기로 인한 저장장치 공간 소모가 파일 시스템의 가비지 컬렉션을 유발시키는 문제가 있다. 로그 구조 파일 시스템에서 가비지 컬렉션을 수행하는 동안은 사용자의 쓰기 요청이 처리되지 못하고 지연되기 때문에 I/O 성능이 감소하게 된다. 하지만, 로그 구조 파일 시스템은 EXT4 파일 시스템처럼 저널링을 사용하지 않기 때문에 저널링 영역에 바이트 로깅을 적용한 기존 기법 [2, 3]을 사용하여 로그 구조 파일 시스템의 불필요한 쓰기 문제를 해결할 수 없다.

본 논문에서는 앞서 설명한 것과 같은 로그 구조 파일 시스템에서 불필요한 쓰기로 발생하는 문제를 해결하기 위한 이중 로깅 기법을 제안한다. 이중 로깅 기법은 기존 로그 구조 파일 시스템에서 사용하는 블록 로깅에 추가적으로 바이트 로깅을 지원하는 기법이다. 이중 로깅 기법은 작은 크기의 바이트 로그들을 로깅 버퍼에 블록 단위로 모아서 저장장치에 기록하기 때문에 불필요한 쓰기를 감소시킬 수 있다. 또한, 기존 바이트 로깅 기법들은 바이트 로그를 저장하는 장치로 블록 저장장치 [2]또는 비 휘발성인 메모리 (SCM) [3]만을 사용하였지만, 본 논문에서 제안하는 이중 로깅 기법은 플래시 저장장치와 SCM을 모두 바이트 로깅 영역으로 사용하여 바이트 로깅을 효과적으로 사용한다. 2장에서는 SCM과 바이트 로깅 기법을 사용하는 다른 기법들을 설명하며, 3장에서는 이중 로깅 기법이 어떻게 구현되었는지 다루고, 4장에서는 실험을 통해 기존의 파일 시스템과 이중 로깅 기법을 사용한 결과를 비교한다. 그리고 5장에서 결론을 낸다.

2. 관련 연구

최근에 PRAM과 STT-MRAM과 같은 바이트 접근이 가능하면서도 비 휘발성인 메모리 (SCM)이 개발되었으며, 이러한

이 논문은 2013년도 정부(교육부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임 (2013R1A1A2A10013598)

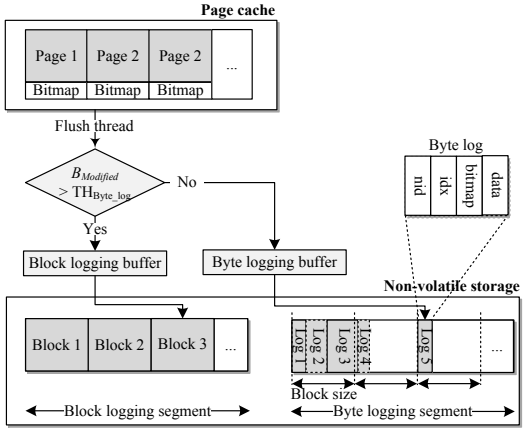

그림 1. 이중 로깅 구조

SCM의 특성을 이용하여 파일 시스템의 성능을 개선하는 기법들이 제시되었다. 먼저, SCM만을 저장 장치로 사용하는 파일 시스템들이 있다 [5]. 하지만, SCM은 플래시 저장장치에 비해 대역폭이 작기 때문에 SCM만을 사용할 경우 성능을 저하시킬 수 있다. 다음으로 플래시 저장장치와 SCM을 함께 사용하는 기법들이 있다 [6]. 일반적으로 파일 시스템에서 메타 데이터의 접근이 많다는 점을 이용하여 메타 데이터를 SCM에 저장하는 기법이다. 본 논문에서 제안하는 이중 로깅 기법은 메타 데이터뿐만 아니라 데이터까지 바이트 단위로 로깅하여 효율을 더 증가시켰다.

Okeanos [2]와 델타 저널링 (Delta Journaling) [3]은 EXT4 파일 시스템의 저널링 기법에 바이트 로깅을 적용한 기법이다. Okeanos는 바이트 로그를 블록 단위로 버퍼링하여 블록 저장장치에 기록하는 기법이다. 그런데, 동기식 쓰기가 발생했을 때는 바이트 로그가 모이지 못하고 불필요한 쓰기가 발생하는 문제가 있다. 델타 저널링은 바이트 로그를 SCM에만 기록하는 기법으로, Okeanos의 동기식 쓰기 문제가 발생하지 않지만 SCM의 대역폭의 한계 때문에 성능이 저하되는 문제가 있다. 또한, 로그 구조 파일 시스템에서는 저널링 기법을 사용하지 않기 때문에 이 기법들을 그대로 적용시킬 수 없다.

3. 이중 로깅

3.1 변경된 부분 블록 추적

우리는 부분적으로 변경되는 블록을 바이트 단위로 추적하기 위해 리눅스의 페이지 캐시를 수정하였다. 수정된 페이지 캐시에서는 페이지마다 비트맵을 가지고 있고, 각 페이지가 변경될 때 마다 대응되는 비트맵의 비트를 표시한다. 비트맵으로 관리하는 변경 단위를 너무 작게 유지하면 페이지당 필요한 비트맵 크기가 늘어나서 메모리 오버헤드가 증가한다. 따라서, 128B 단위로 페이지 변경을 관리하여 페이지당 비트맵의 크기를 4B로 제한함으로써, 메모리 오버헤드를 페이지 캐시 크기의 0.1%로 제한하였다.

3.2 블록 로깅 세그먼트와 바이트 로깅 세그먼트

그림 1은 이중 로깅 기법을 사용하는 로그 구조 파일 시스템의 구조를 보여준다. 페이지 캐시에서 주기적으로 호출되는 플러시 스레드 (Flush Thread)는 더티 상태의 페이지들에 대해 저장장치로 쓰기를 발생시키며, 이때 페이지의 변경된 크기는 페이지 캐시의 비트맵이 변경된 크기, $B_{Modified}$로 계산할 수 있다. 로그 구조 파일 시스템은 저장장치

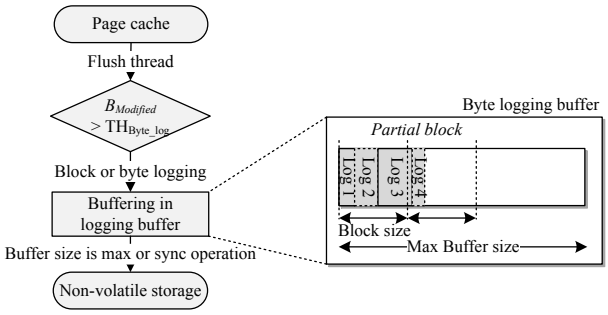

그림 2. 로깅 버퍼에 버퍼링되는 바이트 로깅

공간을 세그먼트 단위로 구분하여 관리하는데, 이중 로깅 기법은 세그먼트를 두 종류로 구분하여 쓰기 요청의 페이지의 $B_{Modified}$값이 미리 설정된 기준점인 TH_{Byte_log}보다 작을 경우 바이트 로깅 세그먼트에 기록하고 클 경우 블록 로깅 세그먼트에 기록한다. 바이트 로깅 세그먼트에는 변경된 일부 바이트만 기록되기 때문에 블록 로깅처럼 이전 블록을 무효화 시키지 않고, 블록 로깅 세그먼트의 가비지 컬렉션과 다른 별도의 가비지 컬렉션을 수행한다.

그림 2는 이중 로깅 기법을 사용하는 로그 구조 파일 시스템에서 쓰기가 처리되는 과정을 나타낸다. 블록 또는 바이트 로그들은 플러시 스레드에서 쓰기를 발생시킬 때 마다 바로 저장장치에 기록되지 않고 로깅 버퍼에 버퍼링된다. 그리고, 버퍼 크기의 부족이나 사용자의 동기식 쓰기 요청이 발생했을 때만 로깅 버퍼에 모인 로그들을 저장장치에 기록한다. 따라서, 블록 크기보다 작은 바이트 로그들도 로깅 버퍼에 블록 단위로 모아서 기록될 수 있으므로 불필요한 쓰기가 발생하지 않는다.

파일 시스템의 안정성을 위해 각 바이트 로그마다 로깅된 페이지를 특정할 수 있는 노드 아이디 *(nid)*와 인덱스 *(idx)*가 같이 기록된다. 파일 시스템의 복구 과정에서는 각 바이트 로그들을 읽어서 대응되는 페이지의 변경된 데이터를 복구 할 수 있다.

3.3 SCM을 사용한 이중 로깅

사용자가 fsync 등의 동기식 쓰기를 요청할 경우 로깅 버퍼에 모인 로그들을 바로 저장장치에 기록해야 하므로 바이트 로그들이 블록 단위로 모이지 못하고 불필요한 쓰기가 발생할 수 있다. 예를 들어, 그림 2와 같이 바이트 로깅 버퍼에서 사용자가 fsync를 요청할 경우 블록 크기보다 작은 부분 블록이 저장장치에 기록되어 불필요한 쓰기가 발생한다. 본 논문에서 제안하는 이중 로깅 기법에서는 SCM을 사용하여 이와 같은 동기식 쓰기 문제를 해결하였다. 동기식 쓰기로 인해 발생한 부분 블록의 크기가 미리 설정된 기준점인 TH_{SCM}보다 작은 경우 바이트 단위 쓰기가 가능한 SCM에 기록함으로써 불필요한 쓰기를 감소시킬 수 있다.

3.4 바이트 로깅된 페이지 관리

바이트 로깅된 페이지에 대한 읽기 요청을 처리할 때 해당 페이지의 바이트 로그들이 서로 다른 블록에 기록되어 있다면 각 바이트 로그가 기록된 모든 블록을 읽어야 한다. 또한, 바이트 로깅 세그먼트에 대해 가비지 컬렉션을 수행할 때 몇

바이트 크기의 바이트 로그를 복사하기 위해서도 한 블록을 복사해야 한다. 이처럼 블록 저장장치에 기록된 바이트 로그에 접근할 때 불필요한 I/O가 발생한다. 이중 로깅 기법에서는 이와 같은 문제를 해결하기 위해 바이트 로깅된 페이지가 페이지 캐시에서 방출될 때 클린 상태라도 블록 로깅 영역에 기록한다. 그 결과 바이트 로그를 포함한 완전한 페이지가 페이지 캐시 또는 블록 로깅 영역 중 하나에 항상 존재한다. 따라서, 바이트 로깅된 페이지에 대한 읽기 요청은 바이트 로그에 대한 접근 없이 처리할 수 있으므로 기존 읽기와 성능 차이가 발생하지 않는다. 이때 바이트 로깅된 페이지를 구분하기 위해 페이지 캐시의 각 페이지마다 로그 비트를 추가하였다. 바이트 로깅 세그먼트에 대한 가비지 컬렉션을 수행할 때도 바이트 로그를 접근하지 않고 페이지 캐시에 존재하는 모든 로그 상태 페이지를 저장장치에 기록함으로써 바이트 로깅 세그먼트를 전부 무효화시키는 방식을 사용한다.

4. 실험 및 평가

실험은 Cortex-A9 667MHz CPU와 512MB의 PRAM, 1GB의 DRAM을 가지는 보드 [7]에서 진행하였으며, 주 저장장치로는 4GB의 micro SD 카드를 사용하였다. 이중 로깅 기법은 F2FS [4]를 기반으로 구현되었으며, 기존 F2FS와 I/O 성능을 비교하였다. 또한, 가비지 컬렉션의 성능을 측정하기 위해 F2FS에서 가비지 컬렉션이 발생하는 기준점을 낮추고 가비지 컬렉션이 시작되기 전까지 저장장치를 채우는 초기화 과정을 거친 후 I/O 성능을 측정하였다. 초기화 과정에서는 파일 시스템의 유효한 블록 비율 (Utilization)이 75%가 되도록 파일을 생성 후 임의 쓰기를 하였다. 그리고, 모든 실험에서 이중 로깅 기법의 TH_{Byte_log}와 TH_{SCM}는 2KB로 설정하였다.

이중 로깅의 성능을 측정하기 위해 SQLite (mobibench)와 tiobench를 사용하였다. 각 벤치마크의 세부사항은 표 1과 같다. 그림 3과 4는 기존 로그 구조 파일 시스템과 이중 로깅 기법을 비교한 결과를 보여준다. *Flash-Only*는 바이트 로깅 영역으로 플래시 저장장치만을 사용한 이중 로깅 기법이며, *Flash & SCM*은 SCM도 함께 사용한 이중 로깅 기법이다. 각 그래프는 블록 단위 로깅만을 사용하는 기존 로그 구조 파일 시스템의 결과값에 정규화한 값을 보여주며, 각 그래프에서 y축의 1은 기존 로그 구조 파일 시스템에서 측정한 결과값을 나타낸다. 그림 3과 4에서 확인할 수 있는 것처럼 모든 워크로드에서 이중 로깅 기법을 적용함으로써 불필요한 쓰기를 감소시킬 수 있었으며, 이로 인해 저장장치 쓰기 공간 소모가 감소하면서 파일 시스템의 가비지 컬렉션으로 발생하는 쓰기 (*GC write*)가 줄어들고, I/O 성능이 증가하였다.

SQLite에서는 빈번하게 사용되는 fsync로 인해 3.3에서 언급한 부분 블록이 많이 발생한다. 때문에 *Flash-Only* 기법을

표 1. 벤치마크 설정

	Workload	I/O size
SQLite	Random writes with frequent fsync	1KB
Tiobench_Big	Random writes with no fsync	1.5KB
Tiobench_Small	Random writes with no fsync and small I/O size	128B

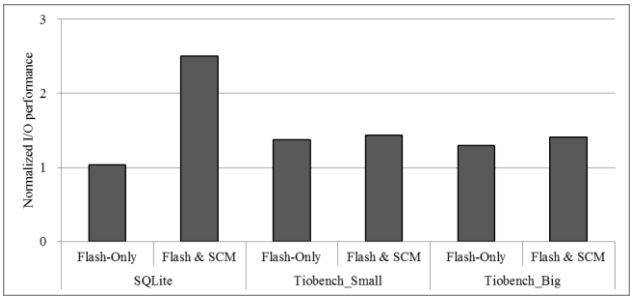

그림 3. 이중 로깅 기법의 성능

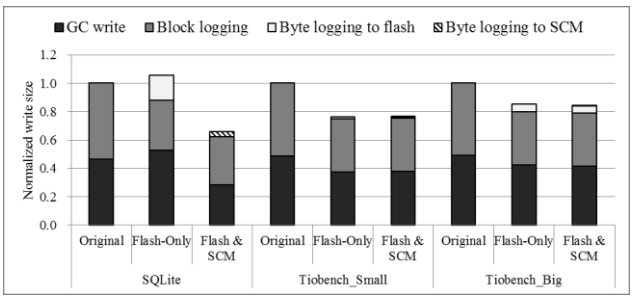

그림 4. 저장장치에 발생한 쓰기 양

사용할 경우 동기식 쓰기 문제가 발생하며, *Flash & SCM* 기법을 사용하여 해당 문제를 해결할 수 있었다. Tiobench에서는 fsync가 사용되지 않기 때문에 *Flash-Only*와 *Flash & SCM*의 결과 차이가 크지 않다. Tiobench_Small과 Tiobench_Big에서는 I/O 크기가 커짐에 따라 바이트 로그의 크기가 커져서 쓰기 양 감소가 줄어들고 I/O 성능 증가도 감소하는 것을 볼 수 있다.

4. 결론

본 논문의 이중 로깅 기법은 불필요한 쓰기로 인한 저장장치 쓰기 공간 소모를 감소시켜 로그 구조 파일 시스템의 가비지 컬렉션 가속화 문제를 해결하였다. 또한, 블록 저장장치와 SCM을 모두 사용하여 바이트 로깅을 효과적으로 사용하는 기법을 제시한다. 실험 결과, 저장장치 공간의 소모를 최대 35% 감소시켰으며 I/O 성능을 최대 150% 향상시킬 수 있었다.

참 고 문 헌

[1] Hatzieleftheriou, et al., “Okeanos: Wasteless Journaling for Fast and Reliable Multistream storage,” *USENIX Annual Technical Conf*, 2011.

[2] Dulloor, et al., “System Software for Persistent Memory,” *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[3] Jeong, et al., “AndroStep: Android Storage Performance Analysis Tool,” *Software Engineering (Workshops)*, Vol. 13, 2013

[4] Lee, et al., “F2FS: A New File System for Flash Storage,” *USENIX Conference on File and Storage Technologies (FAST),* 2015.

[5] Wu, et al., “SCMFS: A File System for Storage Class Memory,” *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[6] Chen, et al., “FSMAC: A File System Metadata Accelerator with Non-Volatile Memory,” *International Conference on Massive Storage Systems and Technology (MSST),* 2013.

[7] Lee, et al., “FPGA-based prototyping systems for emerging memory technologies,” *Rapid System Prototyping* (*RSP),* 2014.