http://dx.doi.org/10.3745/KTCCS.2013.2.10.413

A Fast Parity Resynchronization Scheme for Small and Mid-sized RAIDs

Sung Hoon Baek† · Ki-Wong Park††

ABSTRACT

Redundant arrays of independent disks (RAID) without a power-fail-safe component in small and mid-sized business suffers from intolerably long resynchronization time after a unclean power-failure. Data blocks and a parity block in a stripe must be updated in a consistent manner, however a data block may be updated but the corresponding parity block may not be updated when a power goes off. Such a partially modified stripe must be updated with a correct parity block. However, it is difficult to find which stripe is partially updated (inconsistent). The widely-used traditional parity resynchronization manner is a intolerably long process that scans the entire volume to find and fix inconsistent stripes. This paper presents a fast resynchronization scheme with a negligible overhead for small and mid-sized RAIDs. The proposed scheme is integrated into a software RAID driver in a Linux system. According to the performance evaluation, the proposed scheme shortens the resynchronization process from 200 minutes to 5 seconds with 2% overhead for normal I/Os.

Keywords : Storage, RAID, Operating System, Parallel I/O, Reliability

중소형 레이드를 위한 빠른 패리티 재동기화 기법

백 승 훈† · 박 기 웅††

요 약

정전 방지 장치가 없는 중소형 레이드 (RAID: redundant arrays of independent disks)는 갑작스런 정전 또는 오류로 인한 종료 이후에 수 시간의 긴 재동기화 시간을 요구한다. 레이드에서는 데이터 블록과 패리티 블록이 일관성 있게 갱신되어야 하는데, 데이터를 기록하다가 정전이 되면 데이터 블록은 갱신되었는데 패리티 블록은 갱신되지 않거나 반대인 경우가 발생할 수 있다. 이렇게 부분적으로 갱신된 스트라이프를 반드시 올바른 패리티로 갱신해야 하나 어떤 스트라이프에 이런 문제가 발생하였는지 찾기가 매우 어려웠다. 기존에는 전 저장공간을 검색하고 오류 있는 스트라이프를 수정하는, 수 시간을 요구하는, 패리티 재동기화 방법이 사용되어 왔다. 본 논문은 중소형 레이드에서 낮은 오버헤드를 갖는 고속의 재동기화 기술을 제안한다. 제안하는 기술은 리눅스의 소프트웨어 레이드에서 구현되었다. 성능 실험 결과에 따르면, 제안하는 기법은 재동기화 과정을 200분에서 5초로 단축시키고, 일반 입출력에서 22%의 오버헤드를 2%로 낮추었다.

키워드 : 스토리지, 레이드, 운영체제, 병렬입출력, 신뢰성

1. 서 론

한 개 또는 두 개의 디스크 오류로부터 데이터를 보호하면서 대용량과 높은 처리량을 제공하기 위해 여러 가지 형태의 레이드(RAID: Redundant arrays of independent disks)가 제안되고 있다[12], [2], [3].

소호(SOHO: small office and home office) 환경에서 사용되는 중소형 레이드 시스템들은 사용자의 소중한 데이터를 안전하게 저장하기 위해서 널리 사용되어지고 있으며, 가정 또는 사무실의 내부 네트워크에서 네트워크 결합 스토리지(NAS: Network-Attacked Storage) 기능을 제공한다.

고사양의 레이드 시스템은 신뢰성과 고성능을 위해서 무정전전원공급장치 또는 배터리를 가지는 램(RAM)을 이용한다[4]. 그러나 이런 정전에 안전한 장치는 많은 비용을 유발시켜, 저비용의 중소 레이드 장치에서는 사용되지 않는다.

어떤 쓰기가 레이드-5 장치에 인가될 때에, 패리티와 데이터는 두 개 이상의 디스크들이 일관성을 유지하는 방법을 통하여 동시에 갱신되어야 한다. 만약 패리티는 저장되었는데 데이터는 갱신되지 않은 상태에서 갑작스럽게 정전이 발생하면 쓰기 중이던 그 스트라이프는 일관성을 잃은

† 종신회원 : 중원대학교 컴퓨터시스템공학과 조교수
†† 비 회 원 : 대전대학교 해킹보안학과 조교수
논문접수 : 2013년 6월 20일
수 정 일 : 1차 2013년 8월 27일
심사완료 : 2013년 8월 28일
* Corresponding Author : Ki-Wong Park(woongbak@dju.kr)

상태(불일치 상태)로 남겨지게 된다. 이 경우에, 레이드 시스템은 정전 후에 불일치 상태의 스트라이프를 쉽게 찾을 수가 없다.

불일치 스트라이프들을 찾아서 패리티를 바르게 수정하는 행위를 패리티 재동기화라고 한다. 디스크가 고장 나면 불일치 스트라이프에서 데이터들을 복원할 수 없다. 그래서 크래시 뒤에 재시작을 하자마자 불일치 스트라이프들을 찾기 위해서 전체 저장 공간에 대해 검색하는 패리티 재동기화가 필요하다[5]. 모든 스트라이프를 스캐닝하는 것은 수시간이 소요되고, 사용자에게 불편을 야기한다.

일반 성능을 희생하고 고비용의 재동기화 시간을 획기적으로 감소하기 위해 레이드 드라이버인 리눅스 커널의 MD(multi-device)는 의도 비트맵 (intent bitmap) 기법을 이용한다[6]. 하지만 의도 비트맵은 일반 쓰기 성능을 상당히 저하시킬 수 있다.

다른 재동기화 기법으로서 파일시스템의 저널링 기법을 수정하여 작은 오버헤드로 시간 소모적인 재동기화 과정을 제거하는 기법이 제안되었다[7]. 그러나 이 기법은 파일 시스템과 fsck, mkfs같은 관리 도구들을 변경해야한다. 더구나, 이 기술은 기존 파일시스템과 호환성이 없으며, iSCSI 타겟, SCSI 저장장치, 및 파이버채널 스토리지 같은 블록 장치로 서비스할 수 없다.

본 논문에서 제안하는 기법은 중소형 저비용의 레이드 시스템에 목표를 두고 있으며, 기존의 파일 시스템을 변경하지 않고, 일반 입출력에 대해서 무시할만한 비용으로 수 초 내에 완료할 수 있는 빠른 재동기화 과정을 제공한다. 그러므로 본 논문에서 제안하는 기법은 고비용의 무정전전원 장치를 필요로 하지 않기 때문에 레이드 시스템의 가격을 낮출 수 있는 획기적인 돌파구를 제공한다.

2. 문제점과 기존 기술

2.1 문제점

하나의 스트라이프를 이루는 데이터 블록들과 패리티 블록은 일관성을 유지하는 방법으로 갱신되어야 한다. 만약 데이터가 수정되면 해당 패리티 블록도 동시에 갱신되어야 한다. 또한, 각 디스크의 회전판은 서로 동기화되어 있지 않으므로, 각 디스크 헤드가 목표 섹터에 도달하는 회전 지연 시간은 디스크마다 다르다. 그러므로 다수의 디스크들이 같은 시간에 같은 쓰기요청을 받더라도 각 디스크가 그 요청을 수행하는 시간이 다르다. 그래서 시스템 오류 또는 정전으로 멈출 때에, 패리티와 데이터의 상관관계가 훼손된 불일치 스트라이프들이 발생되는 것을 피할 수 없다.

레이드-5 장치의 디스크들이 자성 매체에 어떤 스트라이프를 기록하는 동안 전원이 꺼지면 그 스트라이프는 불일치 상태가 될 수 있다. 왜냐하면 스트라이프의 일부분은 기록이 되었으나 다른 부분은 기록이 되지 않을 수 있기 때문이다.

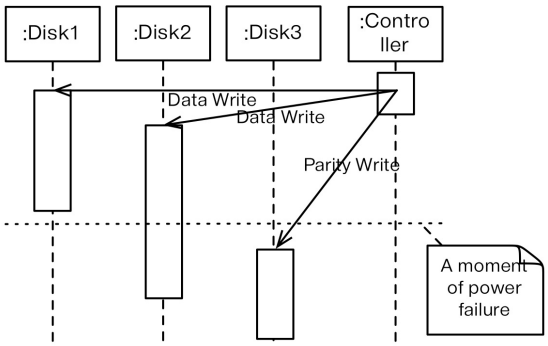

Fig. 1. A data block in disk1 is updated but disk2 and disk3 fail to update their parity block and data block due to a sudden power-off, The old parity of the stripe is inconsistent with the updated data of disk1

Fig. 1에 나타난 예에서, 디스크1의 데이터 블록은 갱신되고, 갑작스런 정전 때문에 디스크2와 디스크3의 패리티 블록과 다른 데이터 블록은 갱신되지 않았다. 이로 인해, 불일치 스트라이프가 존재하게 된다.

불일치 스트라이프가 레이드 시스템에 존재하는데 디스크가 고장 나면, 불일치 스트라이프에서는 잘못된 데이터들을 복원한다. 왜냐하면 불일치 스트라이프에서는 패리티와 데이터의 상관관계가 없으며, 또한 디스크가 고장난 후에는 레이드 시스템은 그곳이 불일치 스트라이프인지도 인지할 수 없기 때문이다.

정전 또는 오류로 재시작할 때에는 중지되는 순간에 계류중이거나 완료된 쓰기 동작에 대한 기록이 없다. 따라서 어느 스트라이프가 불일치 상태인지를 간단히 알 수 있는 방법은 없다.

정전 또는 오류로 재시작한 후에는 불일치 스트라이프가 존재할 수 있으므로 재시작을 하자마자 불일치 스트라이프들을 찾아 그곳의 패리티를 바르게 수정하는 패리티 재동기화를 수행해야만 한다.

2.2 기존 기술

성능과 비용 측면에서 전원이 예기치 않게 꺼질 경우에 스트라이프의 불일치를 피하기 위해서 다양한 방법이 제시되었다.

1) 하드웨어 해결법

신뢰성과 성능을 동시에 달성하기 위해서 하드웨어 레이드는 배터리가 달린 (Battery-backed) RAM, 비휘발성 RAM, 또는 무정전전원장치(UPS)를 사용한다.

비휘발성 메모리에서 버퍼링된 데이터는 시스템 오류 또는 정전에 관계없이 안전하게 보관될 수 있다[4]. 오류 발생 순간에 비휘발성 메모리에서 아직 디스크로 전송되지 않은 데이터 블록들을 재시작한 후에 디스크로 안전하게 전송할 수 있다.

하드웨어 해결법은 오버헤드 없이 최고의 성능을 제공한다. 하지만, 저가의 레이드 장치에는 사용되지 않는다.

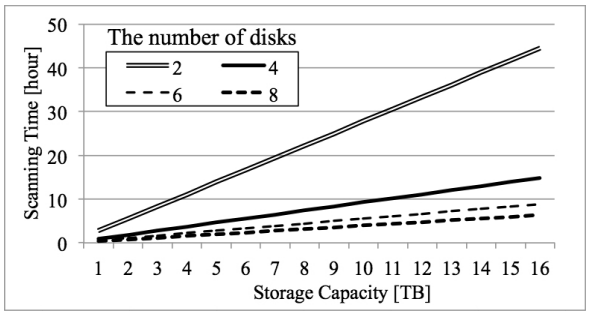

Fig. 2. Scanning time by varing the capacity: full scanning takes tens of hours

2) 전체 스캐닝

패리티 일관성을 유지하기 위해서 널리 사용되는 기법은 불일치 스트라이프를 찾아서 고치기 위해서 전체 저장공간을 스캔하는 방법이다[5].

Fig. 2는 전체 스캐닝에 필요한 시간을 보여준다. 전체 스캐닝을 수행하는데 필요한 시간은 사용자 입출력이 없어도 수 시간 또는 하루이다.

만약 전체 스캐닝 중에 IO가 요청된다면, 스토리지 시스템은 동시에 스캐닝과 사용자 요청을 서비스하기 위해서 나쁜 성능을 제공하게 된다.

3) 의도 비트맵

성능을 희생하면서 짧은 재동기화 과정을 위해서 의도 비트맵이 제안되었다[6]. 의도 비트맵은 어떤 데이터 블록이 비휘발성 저장장치와 휘발성 버퍼 사이에 동기가 되어 있지 않은지 계속 기록한다. 리눅스 커널의 Multi-device(MD)는 의도 비트맵 기법을 지원한다[6].

의도 비트맵의 각 비트는 버퍼에 있는 해당 데이터 블록이 디스크로 갱신되지 않았다면 1로 세트된다. 이 비트 값은 버퍼에 있는 해당 데이터가 디스크와 동기화될 때에 0으로 지워진다.

쓰기가 진행 중에 있는 블록들에 대해서만 해당 비트가 1로 세트되므로 전체 비트들 중에서 1로 세트된 비트는 전체 비트맵 중에서 아주 작은 일부이다.

의심스러운 종료 이후에 의도 비트맵 중에서 1로 세트된 비트에 해당하는 스트라이프들만 검사하면 되기 때문에 재동기화시간을 획기적으로 단축시킬 수 있다.

비트맵의 각 비트는 전체 저장 공간의 블록 또는 스트라이프 등에 해당된다. Fig. 3은 비트맵의 각 비트에 대한 상태도를 보여준다. 데이터 업데이트가 끝난 후에 비동기적으로 해당 비트는 0으로 지워진다.

의도 비트맵은 재동기화 과정을 가속화하지만 일반 입출력과정에서 상당한 수준의 부하를 발생시키는 단점을 가지고 있다.

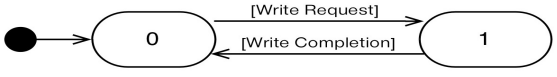

Fig. 3. The state diagram for each bit of the intent bitmap

4) EXT3 선언 모드

EXT3 선언모드(declared mode)는 EXT3 순서모드(ordered mode)의 변형이며, 계류 중인 쓰기에 대한 의도 정보를 저널에 기록하는 방식이다. 이 기법은 저널 트랜잭션을 시작할 때에 저널링 영역에 추가적인 선언 블록을 기록한다.

선언 블록은 현 저널링 트랜잭션의 데이터 블록들이 어디로 기록될지에 대한 위치의 목록을 담고 있다. EXT3 선언모드는 다음과 같이 처리된다[7].

(1) 쓰기 장벽으로 저널링 영역에 선언 블록을 기록한다.
(2) 저널 영역으로 descriptor 블록과 메타데이터를 기록하고 플러시를 수행한다.
(3) 데이터 블록을 홈 위치에 기록하다.
(4) 쓰기 장벽으로 저널영역에 commit 블록을 기록한다.

쓰기 장벽을 가지는 쓰기는 모든 계류 중인 입출력들이 자성 매체로 기록될 때까지 기다리고, 이것이 완료 된 후에 그 쓰기를 디스크로 요청한다. 그리고 이 쓰기가 비휘발성 매체에 기록되기 까지 뒤따라오는 입출력들을 블록시킨다.

EXT3 선언 모드는 작은 부하를 갖지만 빠른 재동기화 시간을 제공한다. 그러나 EXT3 선언 모드를 이용하는 레이드 시스템은 파일 시스템에 의존적이다. 사용자는 레이드에 설치될 파일 시스템의 선택권이 없으며, 더구나 이 기술을 탑재한 레이드 시스템은 파일시스템 없는 iSCSI, SAS, FC 같은 블록 장치를 제공할 수 없다.

3. 의도 로그 기반 재동기화

본 장에서는 새로운 의로 로그 기반 재동기화 기법을 제안한다. 이것은 빠른 패리티 재동기화 과정과 작은 부하를 제공할 뿐만 아니라 파일시스템을 변경할 필요가 없다. 그러므로 이 기술은 일반 쓰기 성능을 저하시키지 않고 고비용의 무정전전원장치 없는 저비용 레이드 시스템을 실현할 수 있다.

제안하는 기술은 저널링 파일 시스템의 쓰기 장벽(write barrier)[13]을 이용하여 무정전전원장치 없이도 휘발성 쓰기 캐시를 사용하는 새로운 의도 로그 기법과 이중 쓰기 캐시로 구성되어 있다.

1절에서 의도 로그와 이중 쓰기 캐시를 통한 새로운 재동기화 기법을 제시하고, 2절에서 무정전전원장치 없는 휘발성 쓰기 캐시에 대해서 설명한다.

3.1 이중 쓰기 캐시와 의도 로그

1) 일반 IO를 위한 동작

본 논문은 앞으로 저장될 블록들의 주소들을 미리 저장하는 의도 로그를 제안한다. 이 로그는 전원이 꺼질 때에 어떤 스트라이프가 저장 중에 있었는지에 대한 힌트를 제공한다.

기본 아이디어는 다음과 같다. 시스템은 쓰기 캐시에서 다수개의 쓰기 요청들을 버퍼링하고, 그 쓰기 요청이 디스크로 전달되기 전에 버퍼링된 데이터의 위치를 담고 있는

의도 로그를 안전한 곳에 저장한다. 그리고 버퍼링된 데이터들을 디스크로 기록한다. 그러므로 쓰여 지고 있는 블록들의 위치는 이전에 의도 로그에 저장이 되었다.

의도 로그는 정전 순간에 기록되고 있던 불일치 스트라이프의 위치들의 전체 집합이다. 그러므로 의도 로그에 적힌 스트라이프만을 대상으로 패리티 재동기화 검색을 수행해도 된다. 따라서 전체 검색보다 시간이 획기적으로 감소한다.

데이터를 버퍼링하고, 버퍼링한 다음 의도 로그를 만든 다음에, 디스크로 버퍼링된 데이터를 기록하는 단계적 과정은 성능에 비효율적이다. 버퍼링과 저장이 동시에 발생해야만 성능에 효율적이다. 본 논문은 버퍼링과 저장이 동시에 연속적으로 처리되도록 이중 쓰기 캐시를 제안한다.

Fig. 4는 어떻게 의도 로그와 이중 쓰기 캐시를 처리하는지를 보여준다. 제안하는 이중 쓰기 캐시는 버퍼링 쓰기 캐시(Buffering Write Cache) 및 저장 쓰기 캐시(Destaging Write Cache)로 구성되어 있다.

이중 쓰기 캐시를 이용해 의도로그를 생성하고 데이터를 저장하는 과정은 3단계로 구성된다.

1단계에서, 저장 쓰기 캐시에 있는 데이터를 디스크로 이동하면서 호스트로부터 받은 쓰기 데이터는 버퍼링 쓰기 캐시(BWC)로 전송된다. 저장 쓰기 캐시(DWC)의 모든 데이터가 디스크로 이동하여 저장 쓰기 캐시가 비워지면 2단계를 수행한다.

2단계에서, 버퍼링 쓰기 캐시와 저장 쓰기 캐시를 서로 맞교환한다. 즉 버퍼링 쓰기 캐시에 있던 모든 데이터가 저장 쓰기 캐시로 이동한다.

3단계에서, 저장 쓰기 캐시에 있는 모든 데이터들의 위치 정보를 담은 의도 로그를 안전한 비휘발성 저장장치에 기록한다. 의도 로그 기록이 완료되면 3 단계 다음인 1단계를 수행한다.

다음 1단계에서는 저장 쓰기 캐시에 있는 데이터들을 디스크로 저장하는 동안 버퍼링 쓰기 캐시는 호스트로부터 새로운 쓰기 데이터를 받는다. 즉, 저장과 버퍼링은 동시에 발생한다.

저장 쓰기 캐시에 있는 모든 스트라이프들을 디스크로 저장하기 전에 의도 로그에 등록된다. 그리고 저장 쓰기 캐시에 있는 스트라이프만이 디스크로 저장된다. 호스트로부터 전송받은 데이터 블록은 버퍼링 쓰기 캐시로만 버퍼링된다.

저장 쓰기 캐시로 데이터 블록을 추가하는 유일한 방법은 2단계에서 버퍼링 쓰기 캐시와 저장 쓰기 캐시를 서로 맞교환하는 것뿐이다. 그러므로 데이터 위치가 의도 로그로 기록되기 전까지는 그 데이터는 저장될 수 없다. 그리고 버퍼링과 저장이 동시에 처리될 수 있다.

저장 쓰기 캐시에 있는 스트라이프의 위치만이 마지막 의도 로그에 저장이 되었기 때문에 저장 쓰기 캐시에 있는 스트라이프만을 디스크로 저장할 수 있다.

LRU같은 캐시 정책을 적용하여, 저장 쓰기 캐시의 데이터에 캐시 히트가 발생할 때에 저장 쓰기 캐시에서 버퍼링 쓰기 캐시로 이동할 수 있다.

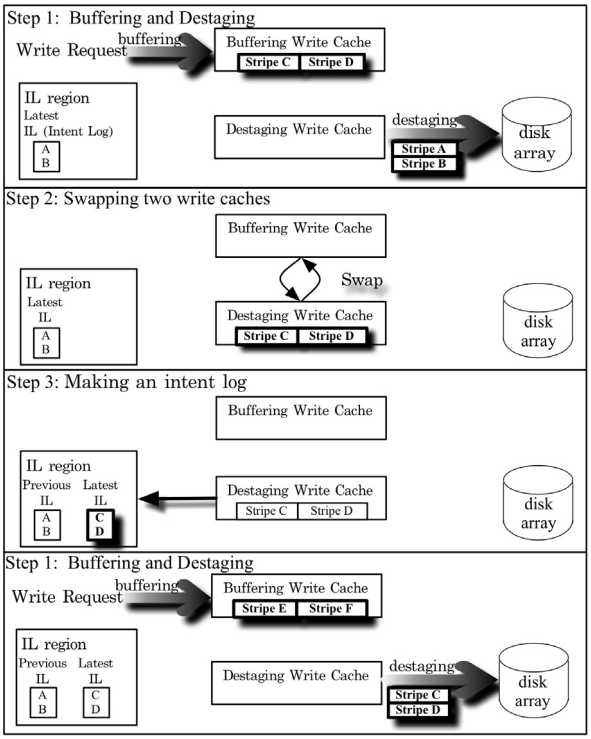

Fig. 4. The dual write caches processes both buffering and destaging simultaneously while no data is destaged until its location is journaled in the intent log region. In Step 1, Stripe C and Stripe D are buffered in the buffering write cache (BWC). Stripe A and Stripe B in the destaging write cache (DWC) are destaged to disks. In Step 2, the BWC is swapped with the DWC. In Step 3, the locations of Stripe C and Stripe D are logged as an intent log (IL). In the second Step 1, Stripe C and Stripe D in the DWC are destaged to disks while new requests (Stripe E and Stripe F) are buffered in the BWC

버퍼링 쓰기 캐시와 저장 쓰기 캐시는 같은 메모리 자원을 공유한다. 저장 쓰기 캐시의 크기가 감소하면 버퍼링 쓰기 캐시의 크기가 증가할 수 있다. 즉, 저장 쓰기 캐시에서 퇴출된 메모리는 버퍼링 쓰기 캐시로 할당될 수 있다.

의도 로그는 디스크의 가장 바깥 트랙에 위치한다. 왜냐하면 바깥 트랙이 안쪽 트랙보다 빠르기 때문이다. 의도 로그는 모든 디스크에 걸쳐 라운드 로빈 방식으로 기록된다. 가장 최신의 의도 로그만이 유효하기 때문이 의도 로그의 버전 번호를 통하여 최신의 의도 로그를 디스크들 사이에서 찾아낼 수 있다.

2) 재동기화

부팅과정에서 시스템이 의심스러운 종료를 감지하면 재동기화 과정을 수행해야 한다. 먼저, 모든 멤버 디스크들에 저장된 모든 의도 로그를 읽는다. 그리고 버전을 비교하여 가장 마지막에 저장된 의도 로그를 고른다. 가장 마지막 의도 로그는 전원이 꺼지는 순간에 갱신되고 있던 스트라이프들의 위치를 알고 있다.

```
1:  write_cache *BWC; // buffering write cache
2:  write_cache *DWC; // destaging write cache
3:  write_cache WC1, WC2
4:  boolean run ← true

5:  swap_write_caches()
6:  begin
7:      write_cache *temp
8:      temp  ←  BWC
9:      BWC  ←  DWC
10:     DWC  ←  temp
11: end

12: // destaging thread
13: raid_destage()
14: begin
15:         ordered_list L  // Intent Log
16:     while (run)
17:         If DWC is empty and BWC is not empty then
18:             swap_write_caches ()
19:             for each stripe S of DWC
20:                 Add the location of S to L
21:             end for
22:             save L
23:         end if
24:         record the stripe cache entry in DWC to the
            disk
25:         wait_event(…)  // A new request or an IO
                           completion wakes up this line
26:     end while
27: end

28: // a function that processes a write request
29: raid_write(write_request *wr)
30: begin
31:     Search a stripe cache entry in the write caches for
        the requests wr.
32:     if the stripe cache entry is found in a cache then
33:         if the stripe cache entry is in DWC then
34:             Remove the stripe cache entry from DWC
35:         end if
36:     else
37:         Allocate a new stripe cache entr, where if no
            free memory is left, wake up the destage kernel
            thread to get a free stripe cache entry, and
            wait until there is a free memory.
38:     end if
39:     Add the stripe cache entry to BWC
40:     …
41: end
```

Fig. 5. The pseudo code of the proposed scheme

의도 로그에 기술된 각 스트라이프에 대해서, 시스템은 데이터 블록들과 패리티 블록을 읽어서 패리티가 이 데이터 블록에 대해서 맞는지 검사한다. 만약 틀리다면, 읽은 데이터로 새로 만든 패리티 블록을 그 스트라이프로 저장한다.

재동기화 중에 검사되는 스트라이프의 개수는 제한적이다. 그래서 재동기화 과정이 예측될 수 있는 짧은 시간 내에 종료한다.

의도 로그는 수천 스트라이프 당 추가적인 한 개의 쓰기를 요구한다. 그러나 의도 비트맵은 최악의 경우 각 블록 갱신마다 비트를 셋하고 클리어하기 위한 두 번의 쓰기 접근을 필요로 한다. 그러므로 의도 로그 기법은 의도 비트맵 기법보다 훨씬 낮은 오버헤드를 필요로 한다.

3) 가상 코드

Fig. 5는 본 기술의 의사코드를 보여준다. 쓰레드인 저장 함수 raid_destage()는 저장 쓰기 캐시(DWC)에 있는 데이터를 디스크로 저장한다(24행). 만약 저장 쓰기 캐시가 비어 있고 버퍼링 쓰기 캐시(BWC)가 비어 있지 않으면 (17행) 이 두 쓰기 캐시를 맞교환한다 (18행). 그다음 저장 쓰기 캐시에 있는 각 스트라이프 캐시 엔트리의 위치를 의도 로그에 추가하고 (20행), 완성된 의도 로그를 비휘발성 저장장치에 저장한다 (22행). 의도로그 저장이 완료되면 DWC에 있는 스트라이프 캐시 엔트리를 디스크로 저장한다 (24행).

호스트에서 전달받은 새로운 쓰기 요청을 처리하는 함수 raid_write()는 이중 쓰기 캐시에 쓰기 요청을 버퍼링하고 (31~39행), 나머지 쓰기 동작을 수행한다 (40행).

호스트가 쓰기 요청을 스토리지 시스템으로 보내면, 스토리지 시스템은 요청받은 블록이 저장 쓰기 캐시에 있는지 검사하고 (33행), 그렇다면 저장 쓰기 캐시에서 해당 스트라이프 캐시 엔트리를 제거하고 (34행), 그것을 버퍼링 쓰기 캐시에 추가한다 (39행). 요청 받은 데이터가 쓰기 캐시에 없으면 (36행) 새로운 스트라이프 캐시 엔트리를 위한 메모리를 할당받아 (37행). 그것을 버퍼링 쓰기 캐시에 추가한다 (39행).

3.2 휘발성 쓰기 캐시

제시한 이중 쓰기 캐시는 무정정전전원장치 없이 구현될 수 있다. 기존 저널링 파일 시스템의 쓰기 장벽과 기존 디스크 프로토콜 기술로써 휘발성 쓰기 캐시를 구현할 수 있다.

1) 저널링 파일 시스템의 쓰기 장벽

저널링 파일 시스템의 쓰기 장벽은 휘발성 쓰기 캐시를 실현하는 핵심요소이다. EXT3[9], EXT4[10], NTFS[11], ZFS[12] 같은 저널링 파일 시스템은 휘발성 쓰기 캐시를 허용하기 위해서 쓰기 장벽을 이용한다[8]. 즉, 저널링 파일 시스템은 정전에 취약한 휘발성 쓰기 캐시로 지연쓰기(write-back)를 하여도 파일시스템의 일관성을 유지할 수 있다.

저널링 파일 시스템의 순서 모드 (ordered mode) 처리과정은 다음과 같다.

(1) 홈으로 데이터 블록들을 기록하고 플러시를 수행한다.
(2) 저널 영역에 descriptor 블록과 메타데이터들을 기록한다.
(3) 쓰기 장벽을 이용하여 저널로 커밋 (commit) 블록을 기록한다.

저널링 파일 시스템은 커밋 블록을 가지는 트랜잭션을 보증한다. 만약 커밋 블록 없이 descriptor 블록 또는 메타데이터가 발견되면 저널링 파일 시스템은 이 불완전한 트랜잭션을 버리고 바로전의 완전한 트랜잭션으로 롤백한다.

리눅스는 커밋 블록을 기록하기 위해서 쓰기 장벽을 이용한다. 이때의 쓰기 장벽은 커밋 블록을 기록하기 전에 이 트랜잭션의 모든 버퍼링된 블록들이 자성 매체로 저장되었음을 보장한다. 다르게 말하면, 스토리지 시스템이 휘발성 쓰기 캐시를 사용하더라도 커밋 블록이 확인된 트랜잭션은 안전하게 기록되었음을 보장할 수 있다[13].

입출력 요청과 함께하는 쓰기 장벽은 그 입출력을 수행하기 전에 쓰기 캐시와 디스크를 동기화시킨다. 동기화 후에

그 입출력을 디스크로 전달한다. 이 입출력이 완료되기 전까지는 모든 새로운 요청들은 블록된다.

저널링 파일시스템에서 커밋 블록을 기록할 때 사용하는 쓰기 장벽은 휘발성 쓰기 캐시를 사용해도 파일 시스템 일관성에 문제가 없도록 한다.

본 논문에서 제안하는 의도로그는 휘발성 쓰기 캐시를 이용하면서 파일시스템의 일관성을 유지하는 쓰기 장벽을 이용한다.

2) 디스크 프로토콜과 레이드의 쓰기 캐시

휘발성 쓰기 캐시를 지원하는 레이드 시스템은 쓰기 장벽을 갖는 입출력 요청을 다음과 같이 처리해야한다. 레이드는 쓰기 캐시에 있는 모든 데이터를 디스크로 플러시 하고, 디스크의 캐시와 자성 매체 사이의 동기화를 디스크로 요청한다. 이 동기화 완료 후에 이 입출력 요청을 바로 쓰기 (write through) 정책으로 수행한다.

SCSI(small computer system interface)는 휘발성 쓰기 캐시를 위한 인터페이스를 제공한다. SCSI CDB(command descriptor block)에 있는 ForceUnitAccess (FUA) 비트는 타겟이 데이터를 디스크매체 표면으로 즉시 전송하도록 하고, 지연쓰기를 허락하지 않는다[14], [15]. 만약 쓰기 명령에 있는 FUA 비트가 0이면, 쓰기 데이터는 스토리지 장치의 내부 캐시에 버퍼링될 수 있다.

SCSI의 SYNCHRONIZE CACHE 명령은 버퍼링된 블록들을 비휘발성 매체로 강제로 플러시하도록 한다. 윈도우즈 8의 NTFS는 FUA 대신에 SYNCHRONIZE CACHE를 사용하여, AHCI (advanced host controller interface)와 SCSI 장치에서 개선된 신뢰성과 성능을 제공한다[16].

파일 시스템의 슈퍼 블록 같은 몇몇 특별한 블록들은 바로 쓰기 정책을 통해 동기적으로 갱신되어야 한다. 동기식 쓰기는 리눅스에서 SYNCIO 플래그로 SCSI 장치에서는 FUA비트로써 인식된다, 그래서 레이드 시스템은 비휘발성 쓰기 캐시를 가지기 위해서 동기식 쓰기 및 비동기식 쓰기를 구분할 수 있어야 한다.

제안하는 의도 로그 기법은 무정전전원장치 없이 쓰기 장벽을 지원하는 휘발성 쓰기 캐시를 이용한다. 쓰기 장벽은 휘발성 쓰기 캐시를 사용하여 파일 시스템이 일관성을 유지할 수 있게 한다.

4. 실험 결과

이 실험에서는 선행 기술들 중 declared EXT3 모드는 배제하였다. 이것은 이것을 위한 새로운 파일 시스템에서 동작하지만. 본 기술은 기존의 파일 시스템에서 동작할 수 있고 블록 장치를 서비스할 수 있기 때문에 블록 장치 서비스가 가능한 의도 비트맵(IB: Intent Bitmap)과 전체 스캐닝으로 제안하는 의도 로그(IL: Intent Log) 기술을 비교한다. 평가 결과는 다양한 벤치마크에서 IL이 IB보다 월등히 우수하다는 것을 보여준다.

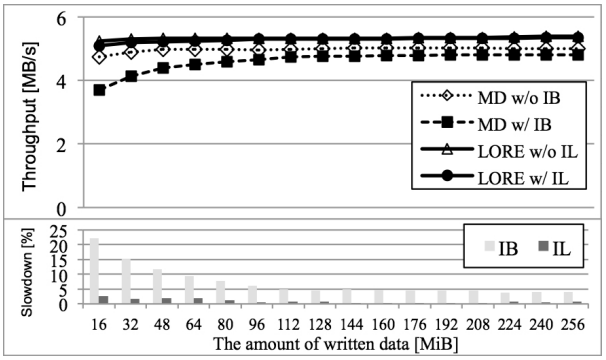

Fig. 6. Random write: the top graph plots random write performance as the amount of data written is increased along the x-axis. The bottom graph shows the slowdown of IB and IL

4.1 실험 환경

LORE라는 RAID 드라이버에서 이중 쓰기 캐시를 갖는 IL기법을 구현하였다. LORE는 여러 선행 연구문서에 소개되었다[17], [18], [19]. 이 레이드 드라이버는 Multiple Device (MD)[20]와 같은 소프트웨어 레이드이며, 리눅스 커널 2.6.35 x86_64에서 구현되었다.

테스트베드는 3.2GHz i7 프로세서와 2GB의 메인 메모리와 다섯 개의 7200rpm SATA3 2TB 하드디스크(ST2000 DM001)을 사용한다. 이 다섯 개의 하드디스크는 LORE 또는 MD 드라이버를 통해 레이드-5로 구성되었다. MD는 리눅스 커널에 포함된 기존 소프트웨어 레이드 드라이버이다. EXT4 파일 시스템을 설치하여 실험하였다.

LORE는 IL기법을 포함하고 있으며, MD는 IB기술을 구현하고 있다. 이 장의 모든 그림의 실험에서는 'IB 없는 MD (MD w/o IB)' 또는 'IB 있는 MD (MD w/ IB)' 또는 'IL 없는 LORE (LORE w/o IL)' 또는 'IL 있는 LORE (LORE w/ IL)'로 레이드-5가 구성되었다. 모든 그래프는 여섯 번의 실험결과의 평균을 보여준다.

4.2 마이크로 벤치마크

이절에서는 임의 쓰기 및 순차 쓰기로 성능을 평가한다. 먼저 Fig. 6은 512MB파일에 임의 쓰기의 성능을 보여준다. 각 실험의 종료시점에 sync()를 호출하여 모든 블록들이 디스크로 플러시(flush)되도록 하였다.

위쪽 그래프는 x축으로 기록한 데이터의 양이 증가함에 따른 임의 쓰기 성능을 보여준다. 아래 그래프는 IB와 IL의 각 상대적인 성능 저하를 보여준다.

아래 성능저하 그래프에서 IB의 성능저하 비율은 'IB 없는 MD'와 'IB 있는 MD'를 통하여 구하였고, IL의 성능저하 비율은 'IL 있는 LORE'와 'IL 없는 LORE'에서 구하였다. IB는 임의 쓰기에서 비트맵의 많은 비트들의 수정을 요구하여 비트맵에 대한 많은 쓰기를 유발한다. IL은 패턴에 관계없이 많은 쓰기에 대해서 한 번의 로그 저장만을 요구한다.

Fig. 6에서 IL의 오버헤드는 0.5%에서 2.5%이고 IB의 오버헤드는 4%에서 22%에 걸쳐있다.

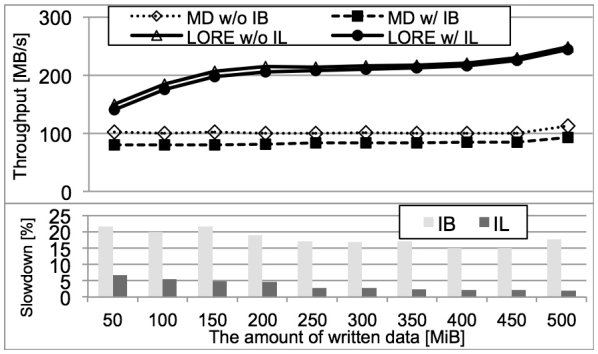

Fig. 7. Sequential write: the top graph plots sequential write performance as the amount of data written is increased along the x-axis. The bottom graph shows the slowdown of IB and IL

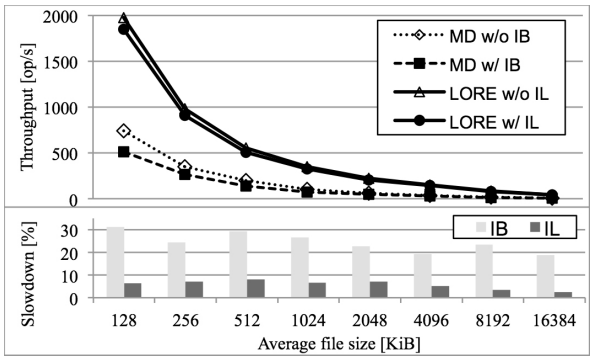

Fig. 8. Filebenchmark throughput. The top graph shows the throughput of Filebenchmark as the size of files served by a file server increases along the x-axis. The bottom graph shows the relative throughput of IB and IL

Fig. 7은 500MB 크기의 파일에 x축에 따라 기록한 데이터량에 따른 순차 쓰기 성능을 보여준다. 실험의 종료시점에 sync()를 호출하여 모든 블록들이 디스크로 플러시되도록 하였다.

임의 쓰기뿐만 아니라 순차 쓰기에서도 IL은 IB보다 더 좋은 성능을 제공한다. Fig. 7에서 IL의 오버헤드는 2%에서 7%이고 IB의 오버헤드는 15%에서 22%에 걸쳐있다. 기록하는 데이터가 많을수록 하나의 의도 로그 기록 당 더 많은 쓰기를 발생하므로 더 낮은 성능저하를 나타낸다.

4.3 매크로 벤치마크

더 실제적인 작업부하를 위하여, Filebench 버전 1.4.8의 'file server' 워크로드를 선택하였다. 이 워크로드는 다수개의 쓰레드로 파일을 생성하고 읽고 지우는 작업을 한다. Fig. 8과 Fig. 9와 Fig. 10에는 x축을 따라 평균 파일이 증가 하면서 변하는 Filebench의 처리량, 대역폭, 지연 시간이 나타나있다.

이 결과에서 IL이 처리량, 대역폭, 지연시간 측면에서 IB보다 월등히 우월하다는 것을 알 수 있다.

IL의 처리량 성능저하는 2%에서 8%에 달하지만 IB의 성능저하는 19%에서 31%에 달한다. 대역폭에 있어서, IL의

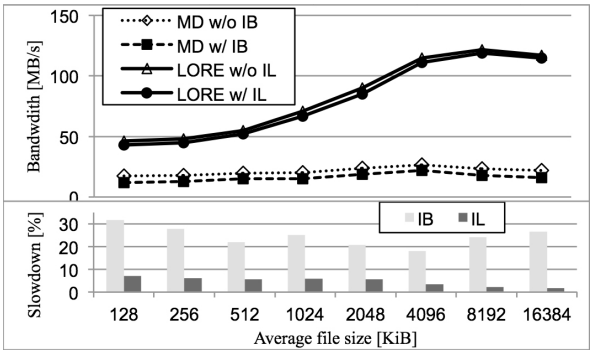

Fig. 9. Filebenchmark bandwidth. The top graph shows the bandwidth of Filebenchmark. The bottom graph shows the relative bandwidth of IB and IL

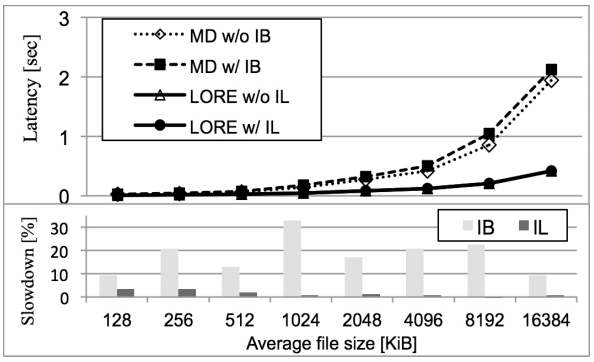

Fig. 10. Filebenchmark latency. The top graph shows the latency of Filebenchmark. The bottom graph shows the relative latency of IB and IL

성능저하는 2%에서 7%사이이나 IB의 성능저하는 18%에서 32%에 이른다. 지연 시간에 있어서, IL의 성능 저하는 1%에서 4%이고, IB는 9%에서 33%에 이른다.

MD와 LORE의 주요 성능 격차는 쓰기 정책에 있다. MD는 바로 쓰기 정책을 사용하고 LORE는 쓰기 장벽을 이용한 지연 쓰기 정책을 지원하기 때문이다.

Table 1은 불완전 종료 후에 불일치 스트라이프를 찾아서 수정하기 위한 재동기화 시간에 대해서 전체 스캐닝과 의도 로그를 비교하고 있다.

Table 1. Resynchronization time

	Full scanning	Intent log
Resynchronization time	200 분	5 초

기존 기법은 10TB의 저장 공간을 재동기화 하는데 200분의 시간이 소요된다. 사용자 또는 관리자가 긴 재동기화 과정이 완료될 때까지 기다리는 것은 상당히 불편한 일이다. 이중 쓰기 캐시를 갖는 의도 로그 기법은 재동기화 시간을 200분에서 5초로 단축할 수 있다.

의도 로그 기법의 재동기화 시간은 쓰기 캐시 크기에 의해서 제한된다. 의도 비트맵의 재동기화 시간은 1로 셋트된 비트의 개수를 제한함으로써 재동기화 시간을 일정 시간 이내로

제한할 수 있다. 그러므로 의도 비트맵도 의도 로그와 마찬가지로 짧은 재동기화 시간을 제공한다. 의도 비트맵과 의도 로그의 재동기화 시간은 환경설정값에 따라서 변경될 수 있다.

5. 결 론

버퍼링된 데이터를 보호하기 위해서 무정전전원장치를 사용하는 고사양의 레이드 시스템과는 달리, 제안하는 기법은 정전 방지 장치 없는 중소형 레이드 시스템에 목표를 두고 있다. 기존 저가형 레이드 시스템의 핵심 문제는 매우 긴 재동기화 시간 또는 비싼 쓰기 오버헤드이다.

전체 저장공간을 스캐닝하는 방법은 좋은 성능을 제공하지만, 부주의한 종료의 대가로 사용자는 수 시간동안의 재동기화 과정을 기다려야 한다. 하지만 IL은 짧은 재동기화 시간을 보이면서도 일반 쓰기 성능을 크게 희생하지 않는다.

제안하는 기술인 이중 쓰기 캐시를 갖는 의도 로그는 무시할만한 성능 저하를 보이고 기존 기술에 비해서 상당히 짧은 재동기화 시간을 요구한다. 제안하는 기술은 중소형 레이드 시스템의 중요 문제를 해결하는 돌파구를 제공한다.

참 고 문 헌

[1] P.M. Che, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson, “RAID: High-Performance, Reliable Secondary Storage,” ACM Computing Surveys, Vol.26, No.2, pp.145-185, June, 1994.

[2] J.S. Plank, “The RAIOD-6 Liberation Codees”, The 6th USENIX Conf. on File and Storage Technologies, pp.97-110, Feb., 2008.

[3] W. Zheng and G. Zhang, “FastScale: Accelerate RAID Scaling by Minimizing Data Migration”, The 9th USENIX Conf. on File and Storage Technologies, Feb., 2011.

[4] J. Menon and J. Cortney, “The Architecture of a Fault-Tolerant Cached RAID Controller”, Proc. of the 20th Annual International Symp. on Computer Architecture, Vol.21, issue.2, pp.76-87, May, 1993.

[5] D. Teigland and H. Mauelshagen, “Volume Managers in Linux”, In Proc. of the USENIX Annual Technical Conference, June, 2002.

[6] P. Clements and J. Bottomley, “High Availability Data Replication”, In Proc. of the 2003 Linux Symposium, June, 2003.

[7] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Journal-guided Resynchronization for Software RAID”, In Proc. of the 4th USENIX Conf. on File and Storage Technologies”, Dec., 2005.

[8] T. Heo. “I/O barriers”, Linux Kernel 2.6 Documentation, July, 2005.

[9] S. Tweedie, “EXT3, journaling filesystem,” in Proceedings of the 2000 Ottawa Linux Symposium, 2000.

[10] 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 2007 Ottawa Linux Symposium, 2007.

[11] M. Russinvovich, “Inside Win2k NTFS, Part 1”, Microsoft Developer Network. 2002.

[12] D. Robbins, “Common threads: Advanced filesystem implementer’s guide, Part 9, Introducing XFS”, Developer Works, IBM. Jan., 2002.

[13] J. Bacik, K. Dudka, H. Goede, D. Ledford, D. Novotny, N. Straz, and et. al., “Write Barriers”, in Red Hat Enterprise Linux 6 Storage Administration Guide, pp.149-151, 2011.

[14] T10, “Information technology - SCSI Primary Commands -3 (SPC-3)”, T10 Committee on SCSI Storage Interfaces, ISO/IEC 14776-313, May, 2005.

[15] Seagate, “SCSI Commands Reference Manual, Rev. A”, Seagate Technology LLC, Publication Number: 100293068, Feb., 2006.

[16] Neal Christiansen, “Windows 8 File System Performance and Reliability Enhancements in NTFS”, Storage Developer Conference, Sept., 2011.

[17] S.H. Baek and K.H. Park, “Striping-aware sequential prefetching for independency and parallelism in disk array with concurrent accesses”, IEEE Trans. on Computers, Vol.58, No.8, Aug., 2009, pp.1146-1152.

[18] S.H. Baek and K.H. Park, “Matrix-Stripe-Cache-Based Contiguity Transform for Fragmented Writes in RAID-5”, IEEE Trans. on Computers, Vol.56, No.8, August, 2007.

[19] S.H. Baek and K.H. Park, “Prefetching with Adaptive Cache Culling for Striped Disk Arrays”, In Proc. of the 2008 USENIX Annual Technical Conference, pp.363-376, June, 2008.

[20] D. Vadala, “Managing RAID on Linux”, O’Reilly & Associates, Inc., 2003.

[21] R. McDougall, J. Mauro, “Filebench: File system micro benchmarks”, http://www.opensolaris.org/os/community/performance/filebench. 2006.

백 승 훈

e-mail : shbaek@jwu.ac.kr

1997년 경북대학교 전자공학과(학사)

1999년 한국과학기술원 전기및전자공학과 (공학석사)

1999년~2005년 한국전자통신연구원 컴퓨터 시스템연구부 연구원

2008년 한국과학기술원 전기및전자공학전공(공학박사)

2008년~2011년 삼성전자 메모리사업부 책임연구원

2011년~현 재 중원대학교 컴퓨터시스템공학과 조교수

관심분야 : 컴퓨터 저장장치, 운영체제, 가상 데스크톱, 시스템 보안, 클라우드 컴퓨팅

박 기 웅

e-mail : woongbak@dju.kr

2005년 연세대학교 전산학과(공학사)

2007년 한국과학기술원 전기및전자공학과 (공학석사)

2012년 한국과학기술원 전기및전자공학과 (공학박사)

2012년~현 재 대전대학교 해킹보안학과 조교수

관심분야 : 시스템 보안, 클라우드 컴퓨팅, 디지털 포렌식