F2FS 파일시스템의 버전별 변화 및 성능 영향 분석

이준호○, 곽현호, 신동군
성균관대학교 반도체시스템공학과
crow6316@skku.edu, gusghrhkr@skku.edu, dongkun@skku.edu

Performance Analysis of F2FS File System Versions

Junho Lee○, Hyunho Gwak, Dongkun Shin
Department of Semiconductor Systems Engineering, Sungkyunkwan University

요 약

리눅스 커널 v3.8에서 처음 소개된 F2FS 파일시스템은 플래시 메모리의 특성을 고려하고 이에 최적화된 LFS 기반의 파일시스템이다. F2FS가 소개된 이후 커널 패치를 통해 점차적으로 개선되어오고 있는데, 지금까지 이러한 패치 내역 및 변화 과정에 대해 정리한 연구는 존재하지 않았다. 그러므로 각 커널 버전에 따라 F2FS의 어떤 부분이 변화하고 어떤 기능들이 추가되었는지를 체계적으로 정리할 필요성이 있었다. 본 연구에서는 F2FS 파일시스템이 소개된 시점부터 최신 리눅스 커널 버전인 v4.16까지 어떠한 패치들이 있었는지를 파악한다. 이 과정에서 특히 성능과 관련 있는 부분에 중점을 두고, 버전별로 추가되는 속성, ioctl 명령어 그리고 기타 개선사항에 대한 약 100가지의 패치 내용과 관련된 커널 코드를 조사하였다. 분석 결과, 성능에 직접적으로 영향을 미치는 패치 내역들을 찾을 수 있었고, 몇몇 패치들에 대해서는 실험을 진행하여 변화에 따른 성능 영향과 그 원인을 분석할 수 있었다.

1. 서 론

최근 Solid State Drive (SSD)나 Universal Flash Storage (UFS)같은 플래시 메모리 기반 저장장치들의 사용이 빠르게 대중화되고 있다. 낮은 전력소모와 높은 내구성 그리고 높은 성능 등의 특성을 가진 플래시 메모리는 기존 HDD 저장장치에 비해 많은 장점을 가진다. 현재 리눅스에서 가장 많이 사용되는 파일시스템 중 하나인 EXT4 파일시스템은 이러한 HDD 저장장치에 최적화 되어있다. 그러므로 플래시 메모리 기반 저장장치의 발전에 따라 해당 특성을 반영하고 이에 적합한 파일시스템을 사용한다면 높은 성능 향상을 기대할 수 있다. 특히 Log-structured File System (LFS) [1] 종류 중 하나인 Flash-Friendly File System (F2FS) [2]는 플래시의 특성을 활용하고 LFS의 다른 이슈들을 해결하여 플래시 메모리 기반 저장장치에서 큰 성능 향상을 보인다.

리눅스 커널 v3.8에서 처음 소개된 F2FS 파일시스템은 플래시 메모리의 특성을 고려하고 이에 최적화된 LFS 기반 파일시스템이다. 이는 리눅스의 대표적인 기본 파일시스템인 EXT4 파일시스템과 종종 비교된다. 본 연구의 목적은 이러한 F2FS 파일시스템에 대한 심층적인 분석이다. 특히 F2FS가 리눅스 커널에 소개된 이후 수년째 점차적으로 개선되어오고 있는데, 지금까지 각 커널 버전에 따라 F2FS의 어떤 부분이 변화하고 어떤 기능들이 추가되었는지를 정리한 연구는 존재하지 않았다. 그러므로 본 연구에서는 F2FS가 변화해온 과정을 추적하며, 그 중 성능과 관련된 변화 이슈들을 정리한다. 이를 위해 먼저 최신 리눅스 커널 버전인 v4.16까지 어떤 속성들과 ioctl 명령어들이 추가되었는지를 정리하고, 각 속성들이 생긴 이유와 역할을 조사하였다. 또한 F2FS에 대한 패치 내용을 분석하고 각 내용을 버전별로 파일시스템의 어떤 영역에 해당되는지를 분류하였다. 이후 그 중에서 성능과 관련이 있다고 판단되는 항목들을 따로 정리하고, 각각에 대해 어떠한 상황 또는 워크로드가 주어질 때 F2FS의 성능이 향상될 수 있는지를 실험하였다.

이 논문은 2016년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. 2016R1A2B2008672)

2. 배경 지식

2.1 Log-Structured File System (LFS)

LFS는 저장장치를 세그먼트 단위로 분할하고, 각 세그먼트를 다시 블록으로 분할한다. 그리고 쓰기 요청을 처리할 때 이 블록들을 순차적으로 할당하여 임의 쓰기 요청들을 순차 쓰기로 변환시킨다. 최근, 이러한 LFS의 순차 쓰기 방식은 임의 쓰기에 취약한 플래시 메모리에 적합하기 때문에 플래시 메모리를 위한 파일시스템으로써 널리 연구되고 있다.

하지만 이 순차 쓰기 방식 때문에 LFS는 invalid 처리된 블록을 회수하는 Garbage Collection (GC)이라는 작업이 필요하다. 기존의 많은 연구들은 LFS에서 GC 오버헤드를 줄이기 위해 In-Place Update (IPU) 정책을 같이 사용하는 등 다양한 노력을 해왔지만, 여전히 LFS에서 GC는 파일시스템 성능에 큰 영향을 미치는 요소이다.

2.2 Flash-Friendly File System (F2FS)

F2FS는 LFS를 바탕으로 만들어진 파일시스템이다. 플래시 메모리는 기존에 사용하던 HDD 저장장치와는 다른 성질을 가지고 있기 때문에, 이러한 성질에 맞고 잘 활용할 수 있도록 하기 위한 목적을 가지고 개발된 파일시스템이 바로 F2FS이다.

덮어쓰기가 불가능하고 로그 형태로 추가만 가능한 LFS에서 데이터를 저장할 공간이 부족해질 때, 더 이상 사용하지 않는 공간을 모아 빈 공간으로 할당하는 것을 클리닝이라고 한다. F2FS에서는 Hot/Cold 데이터를 구분하여 배치하므로 효율적인 클리닝이 가능하며, 높은 공간 사용량에 대비하여 오버헤드를 발생시키는 클리닝의 양을 줄이고자 SSR [3]이라는 기법을 함께 사용한다. SSR은 클리닝 되지 않은 비 활성화된 공간에 새로운 데이터를 덮어쓰는 기법이다. 이 밖에도 Wandering-Tree 문제를 해결하는 등 효율적인 구조를 구성하였다.

3. 파일시스템 기능별 변화 분석

리눅스 커널 버전이 올라감에 따라 F2FS 파일시스템 또한 지속적으로 변화하였다. 새로운 ioctl 명령어, 조정 혹은 읽기 가능한 속성, 마운트 옵션 등을 통해 기능을 더하고, 기존의 정책 및 기법을 개선하여 편의성, 안정성을 높이기도 하였다. 본 연구에서는 F2FS 파일시스템이 처음 소개된 리눅스 커널 v3.8부터 최신 버전인 v4.16까지 F2FS 파일시스템에 어떠한 변화가 있었는지를 파악하였다. 이 과정에서 특히 성능과 관련 있는 부분에 중점을 두고, 버전별로 추가되는 41개의 속성, 25개의 ioctl 명령어 그리고 기타 개선사항에 대한 약 100가지의 패치 내용 및 관련된 커널 코드를 분석하였다. 표 1은 조사한 내용 중 성능과 관련 있는 중요한 버전별 패치 내용들을 F2FS 파일시스템의 4가지 주요 영역(Read, Write, GC, Trim)별로 나누어 정리한 표이다.

Read와 관련된 패치들은 대부분 캐싱을 통해 읽기 지연시간을 줄여 성능을 향상시키는 기법과 관련이 있다. Readahead 기법은 F2FS 파일시스템에서 노드 페이지를 읽을 때, 주변의 노드 페이지들도 미리 읽어서 캐시에 올려놓아 다음 읽기 요청에 대비하여 읽기 성능을 향상시키는 기법이다. *ram_thresh*는 F2FS의 메타데이터가 사용하는 메모리 사용량을 제어하는 속성이다. 기존에는 이를 동적으로 적용할 수 없었는데, 패치를 통해 사용자가 직접 임계값을 조절하며 사용할 수 있게 되었다. *extent_cache*는 최근에 접근한 많은 페이지-블록 변환 정보를 rb-tree 기반 확장 캐시에 추가하는 것을 가능하게 하는 새로운 마운트 옵션이다. 이를 사용하면 연속적인 논리 주소와 inode별 실제 주소 사이의 변환 정보를 캐싱할 수 있으므로 캐시 적중률이 증가할 것이고, 이러한 수치들을 *status*를 통해 확인할 수 있다.

F2FS 파일시스템에 대한 성능과 관련 있는 패치 중 가장 많은 부분을 차지한 것은 Write에 관한 사항이었다. 먼저, *inline_data*는 작은 데이터(기본적으로 3692B 이하)를 inode 블록 자체에 포함하도록 하는 마운트 옵션이다. 이를 통해 공간 요구사항을 줄이고 쓰기 성능을 향상시킬 수 있다. *ipu_policy*는 로그 형태로 Out-of-place 쓰기를 진행하는 F2FS에서 부분적으로 In-place 쓰기가 가능하도록 하는 속성이다. 이를 활성화할 여러 정책이 있으며, 상황에 따라 적용함으로써 추가 노드 블록 쓰기를 줄일 수 있게 되었다. v4.12에서는 F2FS의 Hot/Cold 분리 정책이 바뀌었다. 기존에는 디렉토리와 파일의 형식에 따라 구분했다면, 패치 이후에는 작은 크기(기본적으로 64KB 이하)의 I/O를 Hot한 노드 및 데이터로 취급하는 정책을 사용하고 있다. 이를 통해 작은 I/O와 큰 I/O를 분리하며, 더 큰 연속적인 쓰기를 얻을 수 있게 되었다. 마지막으로 *min_ssr_segments*는 F2FS가 GC 오버헤드를 줄이기 위해 활용하는 Slack Space Recycling (SSR) 기법을 활성화하는 프리 블록 임계값을 사용자가 조절할 수 있도록 한 속성이다.

표 1. F2FS 파일시스템의 영역별 주요 패치 내용

영역	주요 패치 내용
Read	Readahead mode (v3.10) *ram_thresh* (v3.15) *extent_cache* (v4.1)
Write	*inline_data*, *ipu_policy* (v3.14) Write small sized I/O to hot log (v4.12) *min_ssr_segments* (v4.15)
GC	*gc_sleep_time*, *gc_idle* (v3.12) *gc_urgent mode* (v4.14)
Trim	*max_small_discards* (v3.14) Asynchronous discard (v4.11) Enable small discard by default (v4.12) *discard_granularity* (v4.14)

GC에 관한 패치는 GC를 언제 얼마만큼의 주기로 쓰레드를 활성화할 것인지에 대한 변화가 많았다. v3.12에서는 GC 쓰레드의 sleep 시간을 조절하는 *gc_sleep_time* 속성들이 추가되었다. 이를 통해 얼마나 적극적으로 GC를 수행할 것이지를 조절할 수 있게 되었다. 또한 *gc_idle* 속성을 통해 두 가지 GC 정책(Greedy, Cost-Benefit) 중 하나를 선택할 수 있게 되었다. *gc_urgent_mode*는 미리 설정한 절대적인 시간 간격으로 GC를 활성화시킬 수 있는 속성이다. 이를 통해 상황에 따라 백그라운드 GC를 좀 더 공격적으로 실행하도록 설정할 수 있게 되었다.

마지막으로, Trim은 버릴 영역의 크기 조정에 관한 패치가 많았다. v3.14에서는 *max_small_discard* 속성을 제공하고 이를 기본적으로 0을 세팅함으로써 작은 크기(2MB 이하)의 Trim을 억제하였는데, v4.12에서는 심각하게 단편화된 공간을 위해 기본 값을 최댓값으로 바꾸어 이를 허용하였다. 또 v4.14부터는 이 둘의 대안으로써 Trim을 가능하게 할 최소 크기(기본적으로 64KB)를 조정할 수 있도록 *discard_granularity* 속성을 제공하였다. Hot/Cold 분리가 되어있으면 GC를 통해 무효 영역이 곧 확장될 것을 기대할 수 있으므로, 이를 활용하는 것이 도움이 될 수 있다. 또한 v4.11부터는 이러한 Trim 명령을 비동기적으로 변경하여 성능을 향상시켰다.

4. 실험 결과

4.1 실험 환경

실험은 Intel Core i7-8700, RAM 16GB, Samsung SSD 850 evo 250GB의 데스크탑 환경에서 Ubuntu 18.04.1(커널 버전 Linux 4.16.0)를 사용하였다. 원활한 실험을 위해 SSD를 10GB의 파티션으로 나누어 사용하였다. 각 실험은 F2FS의 마운트 옵션을 변경하거나, *sysfs* 속성 값을 바꾸어 성능을 측정하는 방식으로 진행하였다.

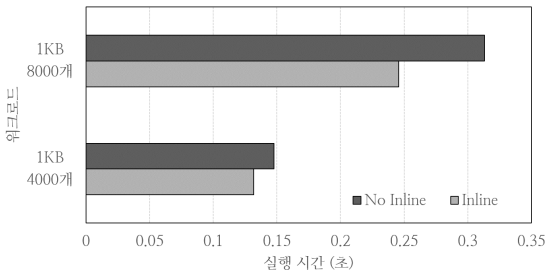


그림 1. *inline_data* 마운트 옵션 활용 실험 결과

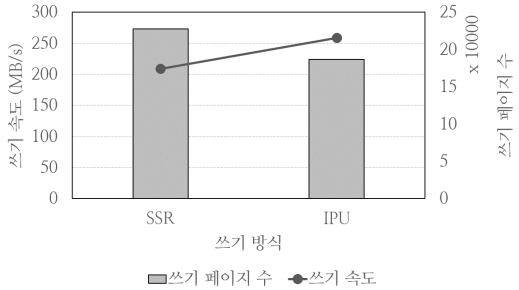


그림 2. *ipu_policy* 속성 활용 실험 결과

4.2 *inline_data* 마운트 옵션 활용

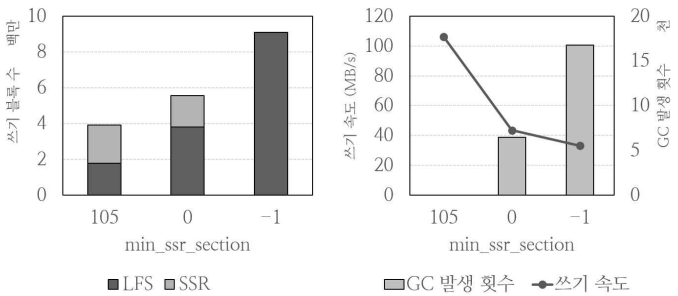
*inline_data*는 작은 크기의 쓰기가 들어올 때, 해당 쓰기를 처리하기 위해 별도의 데이터 블록을 만들지 않고, inode 블록 자체에 포함하게끔 해주는 마운트 옵션이다. 파일을 생성할 때, 모든 데이터가 inode 안으로 들어간다면 그렇지 않을 때보다 필요한 블록은 절반정도가 될 것이다. 이를 확인하기 위해 쓰기가 In-line될 수 있는 작은 크기(1KB)의 파일 여러 개를 생성하는 비교 실험을 하였다. 그림 1의 그래프는 *inline_data* 마운트 옵션 여부에 따른 쓰기 시간을 나타낸다. 해당 옵션을 사용하지 않았을 경우, 파일 수가 4000개일 때는 약 12%, 8000개일 때는 약 28%의 시간이 더 소요됨을 확인할 수 있다.

4.3 *ipu_policy* 속성 활용

최신 LFS 기반의 파일시스템은 필요에 따라 부분적으로 In-place 쓰기를 한다. 그에 따라서 임의 쓰기가 발생하지만, 추가 노드 블록 쓰기를 줄임으로써 성능 향상을 기대할 수 있기 때문이다. 본 실험에서는 FIO [4] 임의 쓰기에서 프리 블록이 줄어들어 SSR 기법이 활성화될 때, SSR 대신 IPU 방식의 쓰기를 함으로써 생기는 성능 향상을 관찰하였다. 그림 2는 두 방식을 각각 사용할 때 발생하는 페이지 쓰기 수와 전체 쓰기 속도를 나타낸다. IPU 방식을 사용할 때는 노드 블록 쓰기의 양이 줄어들기 때문에 전체 쓰기 양 또한 줄어듦을 확인할 수 있다. 또한 임의 쓰기 범위가 넓고 촘촘하여 I/O 스케줄러의 최적화 효과로 인해 쓰기 속도가 어느 정도 보정되어 최종적으로 약 24%의 성능 향상을 확인하였다.

4.4 *min_ssr_section* 속성 활용

*min_ssr_segments*는 프리 블록의 수가 줄어들어 GC가 일어나기 전, 이를 지연시키기 위해 기존의 비활성 블록을 재사용하는 SSR 기법의 임계값을 조절하는 속성이다. 이 임계값을 변경함으로써 생기는 성능 변화를 측정하고자 파일시스템 전체 사용량이 90% 상태에서 FIO 임의 쓰기를 진행하였다. 실험 환경에서 해당 속성의 기본 값은 GC를 위한 reserved 수와 같은 105였다. 이를 증가시켰을 때, 성능 향상이 있었지만 크게 의미 있는 수준은 아니었다. 그림 3은 기본 상태, 해당 속성 값을 0으로 최소화한 상태 그리고 SSR이 아예 일어나지 않도록 하는 별도의 마운트 옵션을 준 상태에서 실험한 결과를 보여준다. 그림 3(a)를 통해 임계값을 줄일수록 SSR 방식의 쓰기 양이 줄어듦을, 이와 반대로 그림 3(b)를 통해 GC



(a) 쓰기 종류 (b) GC 발생 횟수 및 속도

그림 3. *min_ssr_section* 속성 활용 실험 결과

발생 횟수는 크게 증가함을 확인할 수 있다. GC는 valid 페이지의 복사로 인해 추가적인 쓰기를 유발하므로, 이에 따라 LFS 방식의 쓰기 또한 증가함을 그림 3(a)를 통해 확인할 수 있다. GC 발생 횟수가 증가함에 따라 쓰기 속도는 줄어들었고, 이는 GC 오버헤드가 굉장히 크고 SSR 기법이 GC를 지연시킴으로써 쓰기 성능 향상에 큰 기여를 한다는 것을 확인할 수 있다.

5. 결론

이처럼 F2FS 파일시스템이 버전에 따라 개선되면서, 많은 기능들이 추가 및 변경되었다. 기존에 어떤 정책이나 알고리즘이 개선되어 성능을 향상시키거나 메모리 사용량을 감소시킨 패치도 많지만, 속성 등을 추가하여 전체적인 파일시스템의 유연성을 높이는 패치들은 사용자가 자신의 워크로드와 환경에 맞게 F2FS에서 지원하는 속성을 잘 조절함으로써 성능 개선이 가능하게 하였음을 확인하였다.

참고문헌

[1] M. Rosenblum and J. K. Ousterhout, “The Design and Implementation of a Log-structured File System,” *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 26-52, 1992.

[2] C. Lee, D. Sim, J. Hwang, and S. cho, “F2FS: A New File System for Flash Storage,” *Proc. of the USENIX Conference on File and Storage Technologies*, pp. 273-286, 2015.

[3] Y. Oh, J. Choi, E. Kim, and Sam. H. Noh, “Optimizations of LFS with Slack Space Recycling and Lazy Indirect Block Update,” *Proc. of Annual Haifa Experimental Systems Conference*, pp. 1-9, 2010.

[4] J. Axobe, “Fio-flexible io tester,” URL http://freecode.com/projects/fio, 2014.