YAFFS를 위한 파일 연산 최적화 기법

이태훈°, 박송화, 정기동
부산대학교 컴퓨터공학과
e-mail:{withsoul°, downy25}@melon.cs.pusan.ac.kr, kdchung@pusan.ac.kr

File Operation Optimization Technique for YAFFS

Tae-Hoon Lee°, Song-Hwa Park, Ki-Dong Chung
Dept. of Computer Engineering, Pusan National University

요 약

본 논문은 임베디드 시스템에서 효율적인 파일 연산을 위한 메타 데이터의 구조와 파일 연산 최적화 기법을 제안한다. 플래시 메모리는 비휘발성이며 크기가 작고 전력소모도 적으며 내구성이 높아 임베디드 시스템에 널리 사용되고 있다. 하지만 제자리 덮어쓰기(update-in-place)가 불가능하고 메모리 셀에 대한 초기화 횟수가 제한되어 있으며 바이트 단위의 입출력이 불가능하다. 이러한 하드웨어적 특성 때문에 NAND 플래시 메모리 전용 파일 시스템으로 YAFFS(Yet Another Flash File System)가 개발 되었지만 비효율적인 파일 연산 과정의 문제가 존재한다. 본 논문은 YAFFS의 파일 연산을 분석하여 이를 개선시켜 파일 연산 최적화 기법을 제시하고, YAFFS에 적용하여 성능 평가를 한다.

1. 서 론

임베디드 시스템은 크기가 작아 휴대에 용이하고 충격에 강하며 전력소모가 적은 저장 매체를 요구한다. 다음과 같은 특징을 가지고 있는 NAND 플래시 메모리가 많이 사용되고 있다.

그러나 플래시 메모리를 저장 시스템으로 사용하기 위해서는 두 가지 단점이 있다. 첫째, 플래시 메모리는 데이터의 제자리 덮어쓰기(update-in-place)가 불가능하다. 플래시 메모리의 각 비트가 단방향으로만 토글링(toggling)되기 때문에 쓰기 연산 시 초기화 연산을 선행해야 한다. 둘째, 플래시 메모리의 각 블록은 초기화 연산의 횟수가 제한되어 있기 때문에 플래시 메모리의 전체 공간이 균등하게 사용되지 못하는 경우에는 사용가능한 메모리 공간이 급격히 줄어들게 된다[1].

이와 같은 플래시 메모리의 특성으로 인하여 기존의 파일 시스템을 플래시 메모리에 바로 적용할 수 없으므로 플래시 메모리용 파일 시스템의 연구가 활발히 진행되어 왔다. 대표적인 예로 FTL(Flash Translation Layer)[2], TrueFFS[3], JFFS(Journaling Flash File System)[4], YAFFS(Yet Another Flash File System)[5] 등이 있다. FTL은 순차적인 플래시 공간이 디스크의 섹터처럼 보이도록 하기 위해 매핑(mapping) 관리를 수행하는 드라이버 형식으로 구현되어 있다. TrueFFS는 VxWorks RTOS에 맞게 구현한 시스템으로 플래시에 대한 블록 디바이스 인터페이스를 제공하기 위해 Block-to-Flash 전환 시스템을 사용한다. JFFS2(Journaling Flash File System)는 플래시 공간을 순차적으로 저장하는 LFS(Log-structured File System)[6]처럼 플래시 메모리에 대한 갱신 연산을 추가 연산으로 변형하여 처리하며, 이를 통해서 플래시 메모리의 제자리 덮어쓰기가 허용되지 않는 문제를 해결하였다. 하지만 플래시 메모리의 이용률이 커지면서 쓰기 속도가 저하되고, 마운트 시간이 오래 걸리며, 메인 메모리를 많이 사용하는 단점이 있다. 이런 JFFS2의 단점을 해소하기 위해 개발된 것이 YAFFS이다. YAFFS는 NAND 플래시 메모리 전용 파일 시스템으로 마운트 속도와 NAND 유형 플래시 메모리의 입출력 속도에서 JFFS2보다 성능 면에서 우수하다. 하지만 YAFFS는 Tnode를 사용한 파일 연산에서 비효율적인 문제가 있다.

본 논문에서는 기존의 대표적인 NAND 플래시 메모리 전용 파일 시스템인 YAFFS의 파일 연산 과정을 분석하여, 효율적인 NAND 플래시 메모리 파일 시스템을 위한 파일 연산 최적화 기법을 제안하고, 성능 평가를 한다.

본 논문의 구성은 다음과 같다. 2장에서는 YAFFS의 파일 연산에 대한 상세한 내용을 기술하고, 3장에서는 메타 구조와 파일 연산 최적화 기법에 대해 제안한다. 4장에서는 제안된 파일 연산 최적화 기법의 성능을 평가하고 5장에서 이 논문의 결론과 향후 과제를 제시한다.

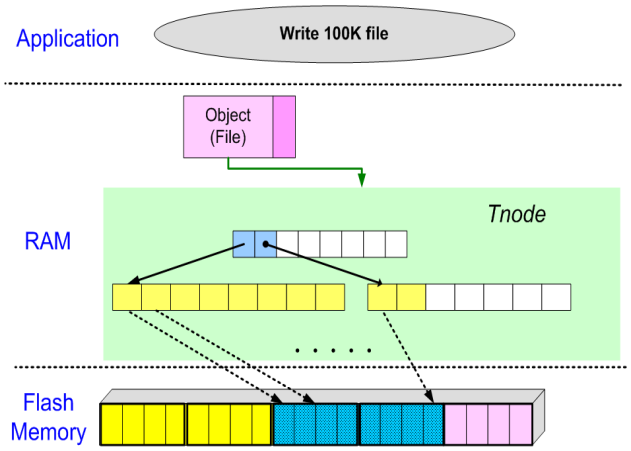
이 논문은 2단계 두뇌한국21사업에 의하여 지원되었음.

2. YAFFS

YAFFS는 2002년 5월 Aleph One사에서 개발하였으며 JFFS2의 높은 메인 메모리 사용률, 느린 마운팅 속도를 해결하기 위해 개발된 NAND 플래시 전용 파일시스템이다.

일반적으로 Small 블록의 NAND 플래시 메모리는 512B의 페이지와 16B의 스페어 영역으로 구성되며 32개의 페이지가 모여 블록을 구성한다. 플래시 메모리에서 읽기와 쓰기는 페이지 단위로 수행되며, 삭제는 세그먼트 단위로 수행된다. 페이지에는 파일의 정보를 관리하는 메타 데이터(yaffs_ObjectHeader)와 파일의 데이터가 저장된다. 메타 데이터는 파일의 이름과 파일의 크기, 수정 시간, 상위 디렉터리에 포인터 등으로 구성된다. tag에는 페이지의 내용에 대한 정보가 저장된다.

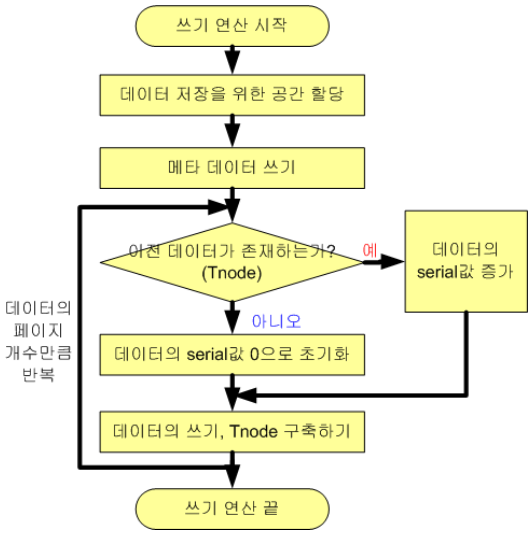
YAFFS는 파일을 생성할 때마다 yaffs_Obejct 객체를 메인 메모리에 만들며, 이 객체들은 서로 연결되어 파일 시스템의 트리 구조를 형성한다. 또한 각 yaffs_Object 객체는 자신이 관리하는 파일에 속하는 데이터들의 위치를 yaffs_Tnode라는 트리로 관리한다. 각 파일의 데이터에 대한 트리를 구성하며, 파일의 데이터에 접근하기 위해서는 매번 yaffs_Tnode에서 물리주소를 읽어야 한다.



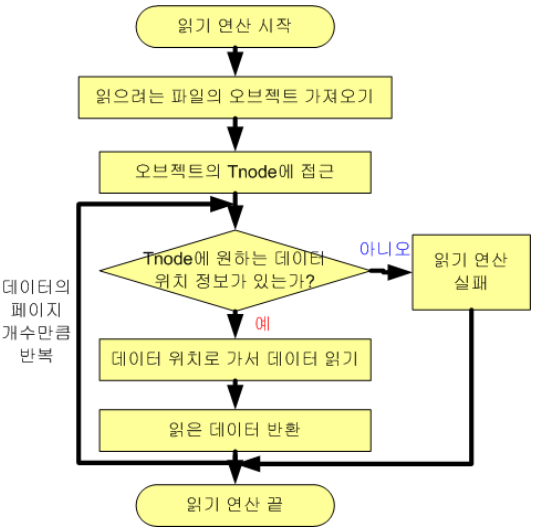
[그림 3] YAFFS의 메타 구조

YAFFS의 파일 쓰기 연산의 과정은 먼저 플래시 메모리에 데이터를 쓰기 위한 빈 블록을 할당받는다. 그리고 파일에 대한 수정이 발생한 것이기에 파일의 메타데이터에 대한 갱신을 수행한다. 메타의 갱신이 수행된 후, 실제 데이터의 쓰기 연산이 수행된다. 이 과정에서 이미 존재하는 데이터에 관한 갱신 여부에 관해서 yaffs_Tnode를 통해서 알아본다. 이미 존재하는 데이터라면 쓰려는 데이터가 더 최신임을 나타내기 위해 serial을 증가시켜 쓰기 연산을 수행하고, 새로 추가되는 데이터라면 serial을 0으로 초기화해서 쓰기 연산을 해주게 된다. 쓰기 연산을 해준 후 Tnode를 재구축해준다. yaffs_Tnode를 통해 데이터의 존재 여부를 확인하여 쓰기를 수행하고 Tnode의 재구축을 해주는 과정을 데이터의 개수만큼 반복하여 파일 쓰기 연산을 수행해준다.

YAFFS의 파일 읽기 연산의 과정은 먼저 읽으려는 파일의 yaffs_Object 객체를 찾고, yaffs_Object를 통해 파일의 위치를 저장하고 있는 yaffs_Tnode로 접근한다. yaffs_Tnode에서 해당 위치에 데이터가 실제 존재하는지를 확인한다. 존재하는 데이터라면 플래시 메모리로 접근하여 해당 위치에서 데이터를 읽어주게 되고, 데이터가 없다면 읽기 연산은 실패한다. 쓰기 연산과 마찬가지로 yaffs_Tnode를 통해서 데이터가 있는지를 확인하고 데이터를 플래시 메모리에서 읽어오는 과정을 필요한 데이터의 개수만큼 반복해준다.



[그림 4] YAFFS의 쓰기 연산 과정



[그림 5] YAFFS의 읽기 연산 과정

이와 같이 YAFFS는 파일의 쓰기 연산과 읽기 연산에서 필요한 데이터의 개수만큼 yaffs_Tnode를 통해 확인하는 작업을 반복해 주고 있다. 이는 트리 구조의 yaffs_Tnode를 매 페이지마다 검색해야하기 때문에 부하가 된다. 이를 최소화 해주기 위해 메타 구조와 파일 연산 최적화 기법을 제안한다.

3. 메타 데이터 구조 및 파일 연산 최적화 기법

3.1 메타 데이터 구조

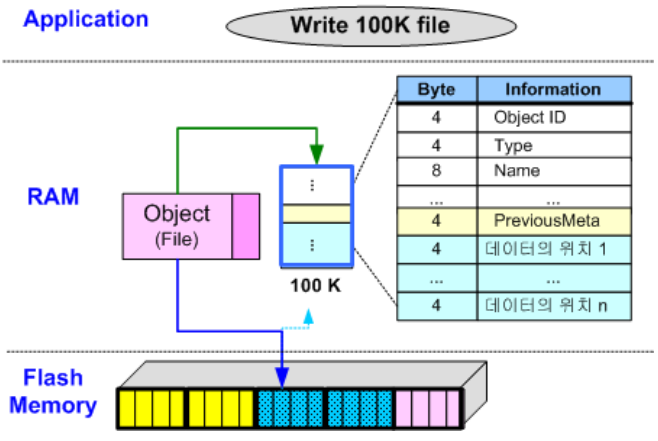
yaffs_Tnode를 사용하고 있는 파일 연산 과정을 바꾸기 위해 사용하려는 메타 데이터 구조는 RFFS[7]에서 이미 제안한 구조이며, [표 1]과 같다. 기존의 메타 데이터의 정보인 58 Byte에 링크의 수, 이전 메타 데이터의 위치, 파일의 데이터 위치 정보를 추가함으로써 yaffs_Tnode를 사용하지 않고도 파일의 데이터의 위치를 메타를 이용해서 찾을 수 있게 해주었다. 메타 데이터는 하나의 페이지에 기록되므로 하나의 메타 데이터가 나타낼 수 있는 데이터의 크기는 한정되어, 결과적으로 지원하는 파일의 크기에 제한을 두게 된다. 이러한 문제점을 해결하기 위해 하나의 파일에 대한 메타 데이터들을 링크드 리스트(linked-list) 구조로 유지한다.

[표 2] 메타 데이터의 구조

Byte	정보
66	기존 메타 데이터의 정보
1	링크(link) 수
4	이전 메타 데이터의 위치
4	파일의 데이터 위치 정보 1
…	…
4	파일의 데이터 위치 정보 n

3.2 파일 연산 최적화 기법

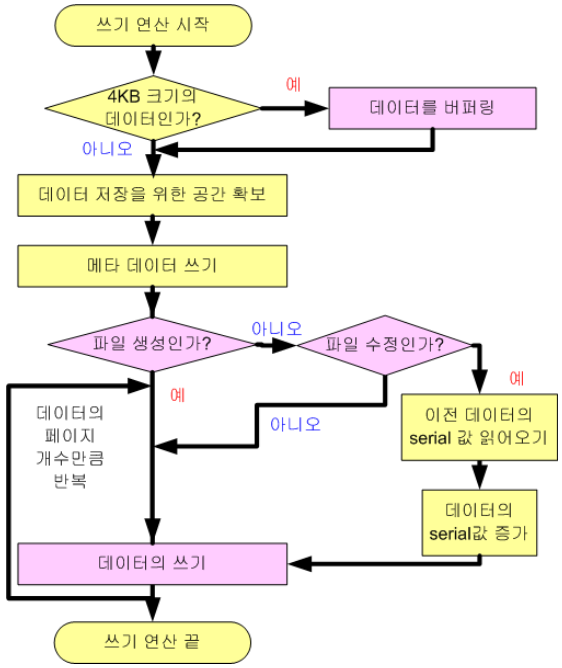
[그림 4]는 새로운 메타 데이터 구조를 이용할 경우 파일 연산의 구조를 나타내고 있다. 메타 데이터에 데이터의 위치가 행렬로 포함이 되어 있기에 이를 이용해 파일에서 해당 데이터로 접근 할 수 있다.



[그림 6] 새로운 메타 구조

3.2.1 쓰기 연산

[표 1]의 메타 구조를 이용하여 최적화 시킨 쓰기 연산 과정은 먼저 버퍼링 과정을 통해 데이터를 모아준다. 버퍼링 과정에서는 데이터가 4KB보다 큰 경우 버퍼링해주며 16KB까지 데이터를 모아준다. 이렇게 모아진 데이터의 쓰기 연산은 먼저 플래시 메모리에 데이터를 쓰기 위한 빈 블록을 할당받는다. 그리고 파일에 대한 수정이 발생한 것이기에 파일의 메타데이터에 대한 갱신을 수행한다. 메타의 갱신이 수행된 후, 실제 데이터의 쓰기 연산이 수행된다. 이 과정에서 YAFFS의 yaffs_Tnode 대신 메타 데이터에 포함되어 있는 데이터의 위치 정보를 이용하여 알아보게 된다. 그리고 페이지의 개수만큼 쓰기 연산을 수행하게 된다. 이미 존재하는 데이터에 관한 갱신 여부에 관해서 YAFFS는 모든 페이지에 대해서 yaffs_Tnode를 통해 알아보지만 제안하는 연산 최적화 기법에서는 최대 16KB의 버퍼링한 데이터에 대해서 한번의 비교 과정을 통하여 알 수 있기에 부하를 줄 일 수 있다. 그리고 YAFFS는 쓰기 연산을 해준 후 yaffs_Tnode를 재구축해주는 과정을 필요로 하지만 제안하는 구조에서는 메타 데이터의 행렬을 업데이트하는 과정은 메타데이터 쓰기 과정에서 같이 수행하고 있으며, 행렬에 대한 갱신이기에 트리 구조보다는 부하가 적다.



[그림 7] 최적화 쓰기 연산 과정

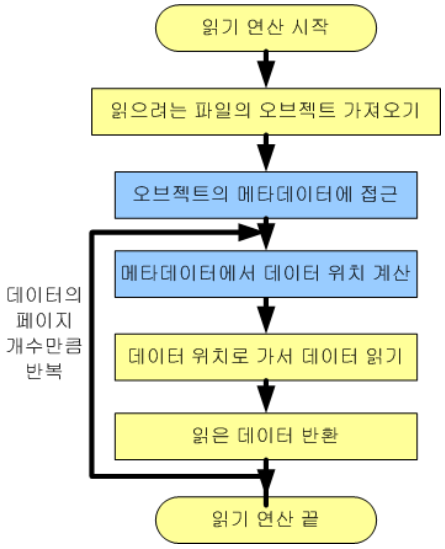
3.2.2 읽기 연산

3.2.2.1 메타 데이터를 통한 읽기 연산

메타 데이터를 통한 읽기 연산과정은 먼저 읽으려는 파일의 Object 객체를 찾고, Object를 통해 파일의 위치를 저장하고 있는 메타 데이터로 접근한다. 메타 데이터에서 데이터의 위치를 계산하고, 데이터의 페이지 개수만큼 반복하여 읽기를 수행한다.

YAFFS는 매번 yaffs_Tnode를 통해 해당 데이터의 위치를 확인하는 과정을 거치는 대신 메타 데이터의 행렬을 사용한다. 행렬을 사용하는 경우 yaffs_Tnode 구조체를 사용하지 않기에 메모리 사용량이 줄어들며, 연산의 복잡도가 낮아지게 된다. 하지만 성능 평가를 하는 과정에서 메타 데이터를 통한 읽기 연산이 yaffs_Tnode를 사용한 읽기 연산에 비해 더 느린 속도를 보였다. 따라서 yaffs_Tnode와 메타 데이터를 적응적으로 사용하는 읽기

연산 기법을 제안한다.



[그림 8] 메타 데이터를 통한 읽기 연산 과정

3.2.2.2 적응적 Tnode 생성 기법을 통한 읽기 연산

적응적 Tnode 생성 기법은 자주 접근하는 K개의 Hot 파일에 대해서만 yaffs_Tnode를 생성하여 사용하는 기법이다. 일반 파일에 대한 접근을 할 때는 메타 데이터의 데이터 위치 정보를 이용해서 접근을 하고, Hot 파일은 yaffs_Tnode를 생성해서 접근한다.

파일 시스템은 마운트 과정 후, 3 번 이상 접근한 파일을 Hot 파일로 간주하고 yaffs_Tnode를 생성한다. 이 과정을 통해 Hot 파일은 K 개까지 만들어지게 된다. 이후 파일들은 (식 1)을 통해 계산되는 가중치로써 Hot 파일이 된다.

$$W = nAccessed \times lastAccessTime \qquad \text{(식 1)}$$

*W*는 가중치이며, *nAccessed*는 접근된 횟수, *lastAccessTime*은 가장 최근에 접근한 시간을 뜻한다. 초기에 3 번 이상 접근한 k개의 파일들이 모인 이후부터는 (식 1)을 통한 가중치를 이용해서 Hot 파일을 결정한다. 일반 파일에 대한 연산이 발생하면, Hot 파일 중 제일 낮은 가중치를 가진 파일과 비교하여 가중치가 더 클 경우, Hot 파일이 되고 yaffs_Tnode를 생성하게 된다. 이와 같이 Hot 파일에 대해서만 yaffs_Tnode를 생성하기에 빠른 연산 속도와 적은 메모리 사용량을 지원하기에 효율적인 NAND 플래시 메모리 파일 시스템에 적당하다.

적응적 Tnode 생성 기법을 통한 읽기 연산은 Hot 파일에 대해서는 YAFFS의 읽기 연산을 사용하고, 일반 파일은 메타 데이터를 통한 읽기 연산한다.

4. 실험 환경 및 성능 평가

4.1 실험 환경

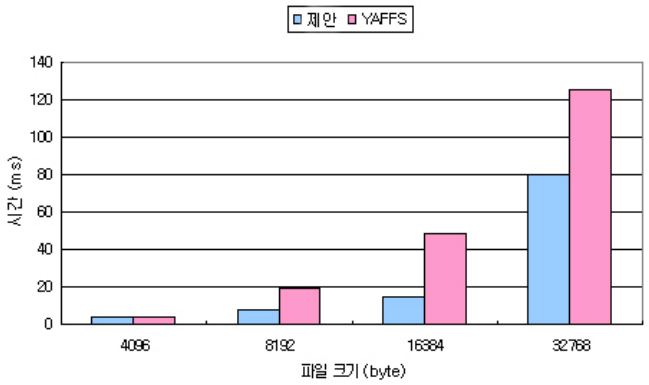
[표 3] PXA255-PRO III 보드

항목		명세
CPU		Intel PXA255
SDRAM		128MB(100MHz)
Flash Memory	NOR	32MB(Intel)
	NAND	64MB(Samsung)

제안하는 파일 연산 최적화 기법의 성능은 [표 2]의 임베디드 보드를 사용하여 기존의 YAFFS와 기법을 적용한 YAFFS의 쓰기 연산 속도와 읽기 연산 속도를 비교, 평가한다.

4.2 쓰기 연산 성능 측정

쓰기 연산의 성능을 측정하기 위해 파일의 크기를 4KB에서 32KB로 변경하면서 쓰기 연산을 수행하였다.



[그림 10] 파일 크기에 따른 쓰기 연산 시간

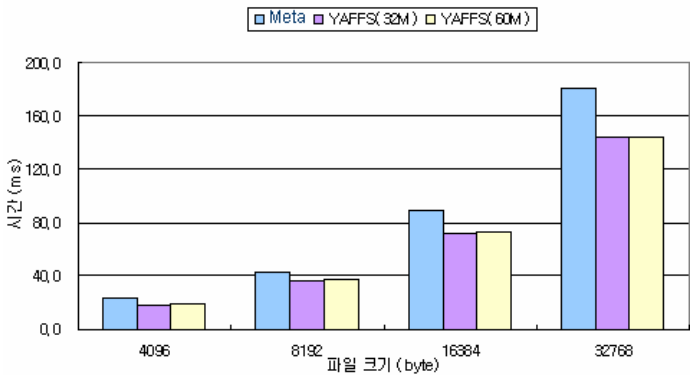
(그림 7)은 파일 크기에 따른 쓰기 연산 시간의 변화를 보여준다. 4KB의 경우는 제안한 기법과 YAFFS가 거의 비슷한 성능을 보여주고 있지만 파일의 크기가 커질 수록 차이가 더 커지는 것을 확인할 수 있다. 이는 YAFFS는 페이지 단위로 yaffs_Tnode를 통해서 확인해주고 쓰기 이후에도 yaffs_Tnode의 재구축을 해야 하기에 파일의 크기가 커질 수 록 성능의 차이가 커지게 된다.

4.3 읽기 연산 성능 측정

읽기 연산의 성능을 측정하기 위해 파일의 크기를 4KB에서 32KB로 변경하면서 읽기 연산을 수행하였다.

4.3.1 메타 데이터

먼저 메타 데이터를 통한 읽기 연산 성능을 측정하였다. yaffs_Tnode의 경우 플래시 메모리의 크기에 따라 트리의 깊이가 달라지기 때문에 32MB와 60MB인 경우에 대해 비교하였다.



[그림 11] 메타 데이터를 통한 읽기 연산 시간

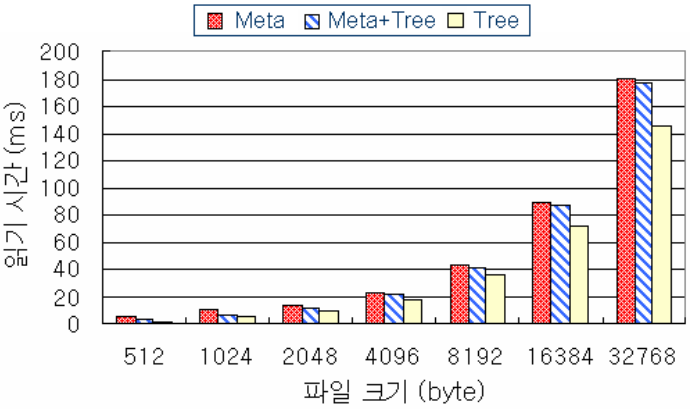
(그림 9)는 파일 크기에 따른 메타 데이터를 통한 읽기 연산 시간을 보여준다. YAFFS의 경우는 32MB보다 트리의 깊이가 깊어지기에 약간 더 느려지는 것을 확인할 수 있다. 메타 데이터를 이용한 경우 32MB의 플래시 메모리에서 실험한 YAFFS보다 약 40% 느리고, 60MB의 플래시 메모리에서 실험한 YAFFS보다는 약 35% 느리다. 이는 트리를 통한 파일의 위치 탐색이 행렬을 통한 위치 탐색보다 빠르기 때문이다.

4.3.2 적응적 Tnode 생성 기법

적응적 Tnode 생성 기법을 통한 읽기 연산 성능을 측정하기 위해 아래의 3가지 경우에 대한 성능을 비교하였다.

- Meta: 메타데이터만 이용 (일반 파일)
- Meta+Tree: 메타데이터를 이용하여 yaffs_Tnode를 구축하여 yaffs_Tnode를 이용
- Tree: yaffs_Tnode만 이용 (Hot 파일)

각 경우에 대해 파일의 크기를 512B에서 32KB까지 변경하며 읽기 연산의 성능을 측정하였다.



[그림 12] 적응적 Tnode 생성 기법의 성능 측정

[그림 10]은 적응적 Tnode 생성 기법의 성능을 측정한 그래프이다. yaffs_Tnode가 생성되어 있는 Hot 파일의 경우가 가장 빠른 읽기 연산 속도를 보여주며, Meta를 이용한 일반 파일의 경우가 가장 느림을 알 수 있다. 특이한 점은 메타 데이터를 읽어 yaffs_Tnode를 구축하여 구축한 yaffs_Tnode를 이용하여 읽기 연산을 한 경우가 메타데이터를 사용한 읽기 연산보다 빠른 속도를 얻은 점이다.

5. 결론 및 향후과제

본 논문에서는 효율적인 NAND 플래시 메모리 파일 시스템을 위한 파일 연산 최적화 기법을 제안하였다. 기존의 대표적인 NAND 플래시 메모리 전용 파일 시스템인 YAFFS의 파일 연산 과정을 분석하여, 쓰기 연산에서는 yaffs_Tnode 대신 메타의 데이터 위치 정보를 이용하여 속도를 향상시켰고, 읽기 연산에서는 yaffs_Tnode대신 메타를 이용하여 연산의 복잡도 및 메모리의 사용량이 감소되는 기법을 제안하였지만 읽기 성능이 떨어져 yaffs_Tnode와 메타를 적응적으로 사용하는 적응적 Tnode 생성 기법을 제안하였다. 이 기법은 모든 파일에 대해yaffs_Tnode를 생성하는 것이 메모리 사용량을 증가시키기 때문에 이를 최소화하면서 읽기 속도를 유지하기 위한 기법이다.

향후에는 좀 더 효율적인 읽기 기법에 대한 연구가 진행되어야 할 것이다.

참고문헌

[1] 김한준, 이상구, "신뢰성 있는 플래시메모리 저장시스템 구축을 위한 플래시메모리 저장 공간 관리 방법", 정보과학회 논문지(시스템 및 이론), 27(6), pp.567-582, 2000

[2] Understanding the Flash Translation Layer(FTL) specification. Intel: 1997.

[3] WindRiver, "TrueFFS for Tornado Programmer's Guide 1.0", 1999

[4] David Woodhouse, "JFFS : The Journaling Flash File System", Technical Paper of RedHat inc. Oct. 2001.

[5] YAFFS Spec, http://www.aleph1.co.uk/yaffs/yaffs.html

[6] M.Resenblum and J.K.Ousterhout, "The Design and Implementation of a Log-Structured File System", ACM Transaction on Computer Systems, Vol.10, pp.26-52, 1992

[6]이태훈, 박송화, 김태훈, 이상기, 이주경, 정기동 “임베디드 시스템을 위한 신뢰성 있는 NAND 플래시 파일 시스템의 설계, 정보처리학회 논문지 A, pp. 571-582, 2005