하이브리드 플래시-디스크 저장장치용 Flash Translation Layer의 성능 개선을 위한 순차패턴 마이닝 기반 2단계 프리패칭 기법

Improving Flash Translation Layer for Hybrid Flash-Disk Storage through Sequential Pattern Mining based 2-Level Prefetching Technique

장재영(Jaeyoung Chang)*, 윤언근(Un-keum Yoon)**, 김한준(Han-joon Kim)***

초 록

본 논문은 플래시 메모리와 하드디스크로 구성되는 하이브리드 저장장치의 성능을 높이기 위한 프리패칭 기법을 제안한다. 하이브리드 저장장치에 포함된 플래시 메모리는 하드디스크에 비해 쓰기/읽기 연산 속도가 상대적으로 빠르기 때문에 이를 캐시 공간처럼 활용하여 성능을 높일 수 있다. 프리패칭을 위한 기본 전략은 순차패턴 마이닝을 이용하는 것이며, 이를 이용하면 시간적 흐름을 가지는 과거 객체 참조열로부터 반복되는 객체 접근 패턴을 추출할 수 있다. 프리패칭 기법을 사용하여 하이브리드 저장장치의 성능을 최대화하기 위하여 본 논문은 두 가지 방법을 사용하였다. 첫 번째는 플래시 메모리 매핑을 위하여 기존의 FAST 알고리즘을 개선하였고, 두 번째는 제한된 플래시 메모리의 공간을 효율적으로 사용하기 위하여 프리패칭 단위로 파일 수준과 블록 수준을 동시에 고려하였다. 제안 기법의 효용성을 평가하기 위해 참조 지역성을 가지는 합성 데이터와 UCC 데이터를 활용하여 실험을 실시하여 제안된 방법의 우수성을 증명하였다.

ABSTRACT

This paper presents an intelligent prefetching technique that significantly improves performance of hybrid fash-disk storage, a combination of flash memory and hard disk. Since flash memory embedded in a hybrid device is much faster than hard disk in terms of I/O operations, it can be utilized as a 'cache' space to improve system performance. The basic strategy for prefetching is to utilize sequential pattern mining, with which we can extract the access patterns of objects from historical access sequences. We use two techniques for enhancing the performance of hybrid storage with prefetching. One of them is to modify a FAST algorithm for mapping the flash memory. The other is to extend the unit of prefetching to a block level as well as a file level for effectively

본 연구는 한국과학재단 특정기초연구(R01-2007-000-20649-0) 지원으로 수행되었으며, 또한 한성대학교 교내연구비 지원과제임.

* 주저자, 한성대학교 컴퓨터공학과

** 공동저자, 텔로드

*** 교신저자, 서울시립대학교 전자전기컴퓨터공학부

2010년 09월 27일 접수, 2010년 10월 16일 심사완료 후 2010년 11월 05일 게재확정.

utilizing flash memory space. For evaluating the proposed technique, we perform the experiments using the synthetic data and real UCC data, and prove the usability of our technique.

키워드 : 하이브리드 저장장치, 플래시 메모리, 순차패턴, 프리패칭, 데이터 마이닝
Hybrid Storage, Flash Memory, Sequential Pattern, Prefetching, Data Mining

1. 서 론

최근 들어 수많은 휴대용 전자기기들이 출시됨에 따라 이들 기기의 저장장치로 사용되는 플래시 메모리(flash memory)의 수요가 급증하고 있다. 특히 최근 시장을 확대하고 있는 스마트폰은 더 이상 휴대전화로서의 기능을 벗어나 전자상거래, 게임, 교통, 인터넷 등 그 효용성이 날로 확대되고 있다. 따라서 이들 기기에서 사용되고 있는 플래시 메모리의 용량도 날로 대형화되고 있는 추세이다. 플래시 메모리는 비휘발성 저장장치로 많은 장점을 지니고 있다. 우선 읽기속도는 DRAM과 거의 근접한 수준이며, 쓰기속도도 일반적인 하드 디스크에 비해서 매우 빠르다. 또한 크기가 작고 단순한 셀 구조로 인해 DRAM에 비해 확장성도 뛰어나다. 하지만 하드 디스크를 대체하기에는 여전히 용량 대비 고비용이라 아직까지는 이를 완전히 대체하지 못하고 있다. 또한 플래시 메모리는 섹터(sector)에 대해서 직접적인 덮어쓰기(overwrite) 연산이 불가능하고, 섹터를 완전히 소거(erase)한 후에만 쓰기연산이 가능하다는 태생적인 한계를 지니고 있다[6, 8].

따라서 하드 디스크와 상대적으로 작은 용량의 플래시 메모리를 결합한 하이브리드(hybrid) 방식의 저장장치가 제안되어 왔으며, 플래시 메모리의 많은 장점으로 인해 일반적인 하드 디스크의 대안이 되고 있다[3, 11, 17]. 하이브리드 플래시 디스크(hybrid flash-disk)는 일반적인 HDD와 플래시 메모리를 결합한 저장장치로 최근 들어 성능향상을 위한 새로운 형태의 보조기억장치로 주목 받고 있다. 예를 들어 삼성의 HM12HII/DOM 플래시 저장장치는 120Gbytes의 HDD와 256Mbytes의 플래시 메모리로 구성된다. 하이브리드 저장장치에 포함된 플래시 메모리는 일종의 캐시(cache) 역할을 한다. 즉, 하드 디스크에서 빈번히 접근(access)되는 데이터를 임시로 플래시 메모리에 저장하여 보조기억장치의 읽기 쓰기 연산의 속도를 현저히 줄일 수 있다. 하이브리드 저장장치의 성능을 결정짓는 중요한 요소는 어떠한 데이터를 플래시 메모리에 적재하는 가에 달려있다. 즉, 가까운 미래에 접근될 데이터를 정확히 예측하여 플래시 메모리에 미리 저장하면 그 만큼 디스크의 접근 빈도를 줄여 I/O 성능을 획기적으로 향상시킬 수 있다.

본 논문에서는 하이브리드 저장장치의 성능향상을 위한 새로운 프리패칭(prefetching) 기법을 제안한다. 프리패칭 기법이란 하드 디스크에 저장된 데이터 중에서 가까운 미래에 접근될 것으로 예상되는 데이터를 미리 플래시 메모리에 적재(load)하는 기법을 의미한

다. 이를 위해 본 논문에서는 FAST(Fully Associative Sector Translation) 기법[9]을 사용하는 플래시 메모리를 가정한다. FAST는 플래시 메모리의 FTL(Flash Translation Layer) 기법 중에서 최근에 제안된 것으로 다른 방식에 비해 높은 성능을 보이는 것으로 알려져 있다. 이 기법은 플래시 메모리 공간을 데이터 블록(data block) 영역과 로그 블록(log black) 영역으로 나누어 사용하며, 로그 블록은 데이터 블록 내의 특정 블록에 덮어쓰기 연산이 발생할 경우, 해당 블록을 소거하지 않고 대신에 로그 블록에 새로운 데이터를 기록하기 위해 사용된다. 특히 FAST는 로그 블록을 이용하여 플래시 메모리 내에서 순차 접근이 빈번히 발생하는 상황에 효율적으로 대처할 수 있도록 설계되었다.

순차패턴(sequential pattern)은 플래시 메모리를 함유한 저장장치에서 흔하게 발생하는 접근 패턴이다. 이러한 성질을 감안하여, 본 논문에서는 프리패칭을 위해 순차패턴 마이닝(sequential pattern mining) 기법을 활용한다[1]. 순차패턴 마이닝은 데이터마이닝(data mining) 기법 중 연관규칙(association rule)의 일종으로 대규모의 순차 데이터로부터 순차패턴을 추출하는 기법을 의미한다. 따라서 순차패턴을 적절히 활용하여 프리패칭할 데이터를 탐색한 후, 가까운 미래에 사용될 데이터를 플래시 메모리에 미리 적재함으로써 하이브리드 저장장치의 성능을 향상시킬 수 있게 된다.

데이터의 접근패턴은 단위에 따라 파일수준(file-level)과 블록수준(block-level)의 순차패턴으로 나눌 수 있다. 파일수준의 패턴은 파일들을 접근하는 순서를 의미하며, 블록수준 패턴은 특정 파일 내에서 블록들을 접근하는 패턴을 의미한다. 따라서 본 논문에서는 순차패턴의 유무에 따라 파일 또는 블록 수준의 두 단계로 프리패칭하는 알고리즘을 제시한다. 반면에 기존의 연구에서는 단순한 휴리스틱(heuristics)에 의존한 블록 단위의 프리패칭에 의존하고 있다[10].

본 논문에서 제안한 알고리즘의 효율성을 검증하기 위해 다양한 실험을 실시하였다. 실험은 지역성(locality)를 고려하여 생성된 합성 데이터와 실제 UCC(User Created Contents) 데이터를 대상으로 실시하였으며, 본 논문에서 제안한 프리패칭 기법이 하이브리드 저장 장치의 성능을 획기적으로 향상시킨다는 것을 검증하였다.

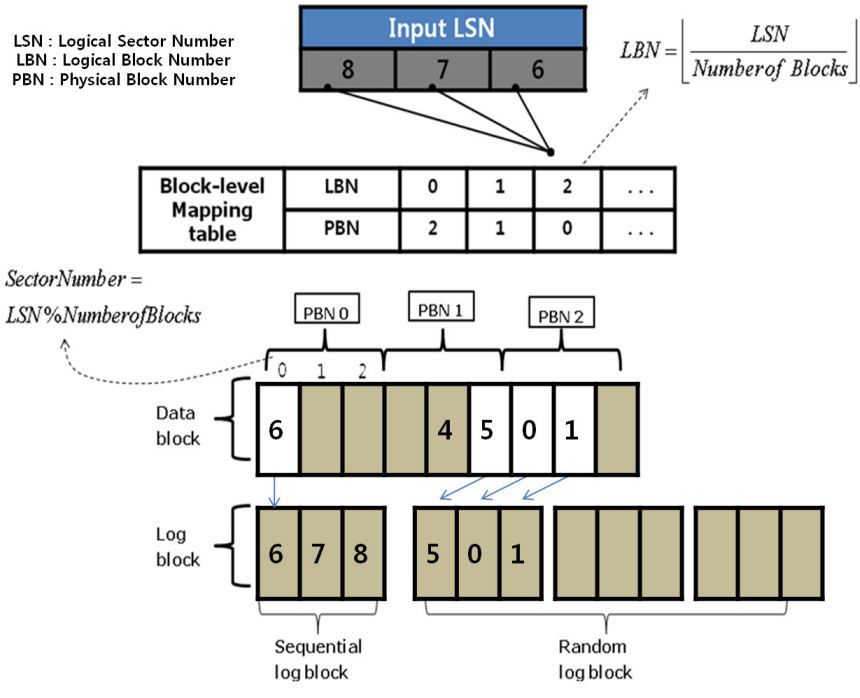
2. 관련 연구

2.1 FAST 알고리즘

본 논문에서는 플래시 메모리의 관리 기법 중에서 소거 연산을 최소화하기 위해 고안된 FTL의 한 종류인 FAST 기법[9]을 가정하였다. FAST 알고리즘은 플래시 메모리를 데이터 블록과 로그 블록으로 나누어서 사용한다. 로그 블록은 다시 순차쓰기용 로그 블록(sequential write log block)과 임의쓰기용 로그 블록(random write log block)으로 나누어 사용한다<그림 1>. 로그 블록을 두 영역으로 나눠서 사용하는 이유는 순차쓰기용 로그 블록을 사용함으로써 합병(merge) 연산보다 교환(switch) 연산을 많이 발생시키기 위해서

다. 일반적으로 교환 연산은 합병 연산보다 더 효율적인 것으로 알려져 있다[9].

FAST 알고리즘은 하이브리드 매핑(hybrid mapping) 방식을 사용한다. 하이브리드 매핑은 블록 매핑(block mapping) 방식과 섹터 매핑(sector mapping)을 혼용하여 사용하는 방식이다. 하이브리드 매핑 방식은 덮어쓰기 연산을 최소화하기 때문에 현재 가장 많이 사용되고 있다. 만약 덮어쓰기 연산이 발생한다면 FTL은 두 가지 경우를 생각할 수 있다. 첫 번째는 순차쓰기 연산(sequential writes)이고 두 번째는 임의쓰기 연산(random writes)이다. 우선 순차쓰기인 경우, 특정 데이터 블록의 첫 번째 섹터에 덮어쓰기가 발생하면 순차쓰기로 가정하고 순차쓰기용 로그 블록을 할당한다. 이후의 블록내의 연속된 섹터에 대한 쓰기 연산에 대해서는 순차쓰기용 로그 블록에 연속적으로 기록된다. 순차쓰기용 로그 블록의 모든 섹터가 사용되면 매핑 테이블의 원래 데이터 블록을 지우고, 순차쓰기용 로그 블록을 데이터 블록으로 교환연산을 수행한다. 그 이외의 쓰기연산에 덮어쓰기 연산이 발생하면 임의쓰기용 로그 블록에 차례로 쓰기연산을 수행하는데 이때는 순차쓰기와는 달리 각 임의쓰기용 로그 블록이 특정 데이터 블록과 매핑되지 않고 섹터 단위로 매핑된다. 따라서 임의쓰기용 로그 블록에 대해서는 섹터단위의 매핑 정보가 필요하게 된다. 또한 임의쓰기용 로그 블록들이 모두 사용될 경우 상대적으로 비용이 발생하는 합병연산으로 빈 블록을 할당 받게 된다.



〈그림 1〉 FAST 내부 구조

<그림 1>은 FAST 알고리즘의 데이터 처리에 관한 구조를 나타낸다. 이 그림에서 데이터 블록과 로그 블록의 섹터들 중에서 음영으로 처리된 섹터들은 현재 비어 있음을 나타낸다. 예를 들어 <그림 1>에서 쓰기연산에 대한 논리적 섹터 번호, 즉 LSN이 6, 7, 8의 순으로 차례로 요청되었다고 가정하자. 그러면 우선 FTL이 입력된 LSN 6의 정보를 처리하기 위하여 논리적 블록 번호(logical blcok number : LBN)를 6/3(한 블록의 섹터수)=2와 같이 계산하고, 논리적 블록 번호를 사용하여 매핑 테이블(block-level mapping table) 정보로부터 물리적 블록 번호(Physical Block Number : PBN) 0을 찾는다. 다음으로 FTL은 이 정보를 이용하여 LSN 6의 정보를 첫 번째 데이터 블록에 기록하게 된다. 그러나 이때 PBN 0의 첫 번째 섹터에 데이터가 기록되어 있을 경우 덮어쓰기 연산이 발생한다. 이 경우에는 블록의 첫 번째 섹터이므로 순차쓰기용 로그 블록에 이 정보를 기록하게 된다. 이후의 연속된 LSN 7, 8의 섹터에 대해 쓰기 연산이 발생하면, 새로운 데이터가 로그 블록에 기록되어 있으므로 데이터 블록이 아닌 <그림 1>과 같이 로그 블록에 기록하게 된다. 그 이유는 추후에 합병연산이 발생할 경우 이 로그 블록을 데이터블록으로 단순히 교환함으로써 합병비용을 절약하기 할 수 있기 때문이다.

다음으로 임의쓰기용 로그 블록을 사용하는 예를 보자. <그림 1>에서 LSN이 5, 4, 1, 0의 순으로 쓰기 연산이 발생한다고 가정하자. 우선 LSN 5는 덮어쓰기 연산이 필요하지만, 블록의 첫 번째 섹터가 아니므로 임의쓰기용 로그 블록에 기록하게 된다. 다음으로 LSN 4의 경우는 이에 대응되는 PBN 1의 두 번째 섹터가 비어 있으므로 데이터 블록에 직접 기록된다. LSN 1, 0에 대해서는 모두 임의쓰기용 로그 블록에 기록된다. 여기서 LSN 0는 블록의 첫째 섹터임에도 임의쓰기용 로그 블록에 기록된 이유는 이미 같은 블록에 있는 LSN 1이 임의쓰기용 로그 블록에 기록되어 있기 때문이다. 임의쓰기용 로그 블록의 공간이 모두 차있을 경우, 여기에 쓰기 연산이 다시 발생하면 가장 처음의 로그 블록에 저장된 섹터들의 해당 블록들에 대해 합병연산을 하고 로그 블록에 대해서는 소거 연산을 실행한 후에 다시 사용하게 된다.

본 논문에서는 FAST 방식이 순차쓰기용 로그 블록을 활용하여 순차쓰기에 대한 성능 향상을 가져왔다는데 착안하여, 순차패턴을 이용한 프리패칭 기법에 이 구조를 활용하는 기법을 제안한다.

2.2 순차패턴 마이닝

순차패턴은 연관규칙에 시간 개념을 첨가하여 시간의 흐름에 따라 반복적으로 나타나는 행위 또는 연속적인 사건의 집합을 의미하며, 순차패턴 마이닝은 대용량 순차 데이터베이스로부터 순차패턴 탐색하는 것이다[1]. 예를 들어, 마켓에서 “CANON 디지털 카메라를 구입한 사람은 곧이어 삼성 컬러 프린터를 구입할 가능성이 높다”와 같이 구매자들의 장바구니를 분석하여 구매 패턴을 발견하는 것이 순차패턴 마이닝의 주된 역할이다.

순차는 사건들의 순서화된 목록으로, 순차

$s = < e_1, e_2, \cdots e_n >$과 같이 나타낸다. 여기서 e_i는 하나의 사건이며, e_i, e_j에 대해서 $i > j$이면 e_i가 e_j보다 시간적으로 먼저 발생한 사건임을 나타낸다. 주어진 순차 데이터에 대해서 순차패턴 마이닝은 높은 지지도(support)를 갖는 순차패턴들의 집합을 찾아낸다. 여기서 지지도란 순차 데이터베이스 내에서 해당 패턴을 갖는 순차 데이터들의 레코드 수로, 지지도가 높을수록 해당 패턴이 강하게(또는 높은 확률로) 나타난다는 것을 의미한다. 이러한 패턴을 이용하면 특정 사건이 발생하였을 경우 어떠한 사건이 가까운 미래에 발생할 것인가를 예측할 수 있다. 따라서 이러한 순차패턴 마이닝 기법을 이용하면, 하이브리드 저장장치에서 프리패칭할 데이터의 결정에 응용될 수 있다. 즉, 기존의 파일이나 블록 단위의 참조 기록을 바탕으로 순차패턴을 탐색하고 이 정보를 이용하여 특정 데이터 집합이 플래시 메모리에 적재되었을 때, 프리패칭할 대상을 결정할 수 있다.

순차패턴 마이닝 기법은 크게 두 가지 부류로 나눌 수 있는데, 그 기준은 연관 규칙의 Apriori 알고리즘에 기반하고 있는가 여부이다. Apriori 알고리즘에 기반한 순차패턴 마이닝 기법은 AprioriAll[2], SPADE(Sequential Pattern Discovery using Equivalent Class)[16], GSP(Generalized Sequential Patterns)[2] 등이 있으며, 그렇지 않은 기법은 PSP(Prefix tree for Sequential Pattern) [13], FreeSpan(Frequent Pattern-Projected Sequential Pattern Mining)[7], WAP(Web Access Pattern Tree)[14] 등이다. Apriori 기반 알고리즘은 기본적으로 매우 큰 후보 패턴 집합을 생성해야 하는 부담을 안고 있어 수행시간을 줄이는데 한계가 있다. 이에 비해 WAP 알고리즘은 자주 발생하는 사건을 탐색하여, 그것들을 트리구조로 재구성하여 suffix frequent pattern을 찾아냄으로써 수행시간을 대폭 줄일 수 있다. 본 연구에서는 WAP의 한 종류인 Pre-order linked WAP tree (PLWAP) 기법[5]을 채택하였으며, 이는 트리의 각 노드에 이진 포지션 코드를 할당하여 Pre-Order 방식으로 노드를 연결시킴으로써 기존 WAP 기법의 성능을 한층 더 개선하였다.

3. 프리패칭 기법

3.1 문제정의

본 논문의 목표는 하이브리드 저장장치에서 향후에 접근될 것으로 예측되는 파일이나 블록을 캐시 역할을 하는 플래시 메모리에 프리패칭함으로써 전반적인 I/O 시간을 줄이는 것이다.[1] 이를 해결하기 위해서는 다음의 두 문제를 해결해야 한다. 첫째는 프리패칭할 객체를 선정하는 방법을 결정해야하며, 둘째는 선정된 객체를 FAST 구조에 어떠한 방법으로 적재할지를 결정해야 한다.

우선 첫 번째 문제를 해결하기 위해 본 논문에서는 앞서 언급한 바와 같이 순차패턴 마이닝 기술을 활용한다. 예를 들어 <표 1>과 같은 데이터 접근 순차를 가정하자. 이 표

1) 본 논문의 이후부터는 프리패칭 대상이 되는 파일이나 블록을 '객체'라고 부른다.

〈표 1〉 데이터 접근 순차 예($S_1^{F_5} \sim S_5^{F_5}$는 파일 F_5내의 블록들에 대한 순차를 의미함)

SID	파일접근 순차	SID	블록접근 순차
S_1	$< \boldsymbol{F_5}, \boldsymbol{F_4}, F_3, \boldsymbol{F_1} >$	$S_1^{F_5}$	$< \boldsymbol{B_2}, \boldsymbol{B_1}, B_3 >$
S_2	$< F_2, \boldsymbol{F_5}, F_{10}, \boldsymbol{F_4}, F_{15}, F_4, F_{15}, \boldsymbol{F_1} >$	$S_2^{F_5}$	$< \boldsymbol{B_2}, \boldsymbol{B_1}, B_3, B_4 >$
S_3	$< \boldsymbol{F_5}, F_{10}, \boldsymbol{F_4}, \boldsymbol{F_1} >$	$S_3^{F_5}$	$< B_2, B_3 >$
S_4	$< F_2, \boldsymbol{F_5}, F_5, \boldsymbol{F_4}, F_2, \boldsymbol{F_1} >$	$S_4^{F_5}$	$< B_1, \boldsymbol{B_2}, \boldsymbol{B_1}, B_4 >$
S_5	$< F_2, \boldsymbol{F_5}, F_6, \boldsymbol{F_4}, F_9, \boldsymbol{F_1} >$	$S_5^{F_5}$	$< B_1, \boldsymbol{B_2}, B_3, \boldsymbol{B_1}, B_1, B_5, B_1 >$

에서 왼편은 파일들에 대한 순차 레코드를 나타내며, 오른편은 파일 F_5에 대해서 파일 내부 블록들의 순차 레코드를 나타낸다. 우선 파일들에 대한 순차 접근 레코드를 보면 $< F_5, F_4, F_1 >$의 접근 패턴을 발견할 수 있다. 이 패턴의 의미는 F_5가 접근되었을 때 F_4, F_1의 순으로 접근될 확률이 매우 높음을 나타낸다. 따라서 실제 환경에서 F_5가 접근되었을 때 F_4와 F_1을 플래시 메모리에 미리 적재시키면 I/O 연산의 성능을 크게 향상시킬 수 있다. 또한 <표 1>의 오른편 블록 순차를 보면 $< B_2, B_1 >$과 같은 순차패턴을 발견할 수 있다. 따라서 B_2가 접근되었을 경우, B_1을 사전에 프리패칭함으로써 I/O의 성능향상을 기대할 수 있다.

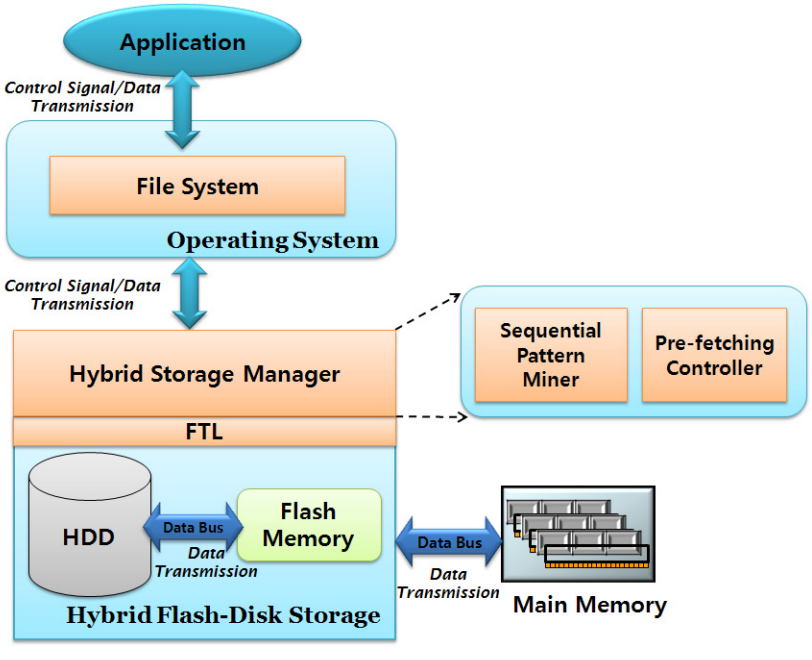
일반적으로 파일 접근 순차의 레코드들은 장기간 운용하게 되면 그 양이 매우 방대해진다. 따라서 이들로부터 추출될 수 있는 순차패턴의 수 또한 매우 커질 수 있다는 것도 예상할 수 있다. 이 경우 너무 많은 객체들이 프리패칭되어 오히려 성능을 저하시키는 결과를 낳게 된다. 이를 해결하기 위한 하나의 방법으로 순차패턴을 선정하기 위한 최소 지지도(minimum support)를 높이는 방법도 고려해 볼 수 있지만 근본적인 해결책이 되기에는 부족하다. 따라서 본 논문에서는 가중치 그래프(weighted graph)를 이용하여 특정 객체가 접근되었을 경우 접근 가능성이 가장 높은 하나의 인접 객체만을 프리패칭하는 방식을 사용한다.

두 번째로 프리패칭 기법을 FAST 기반 하이브리드 저장장치에 적용하기 위해, 본 논문에서는 기존의 FAST 구조를 확장하여 본래의 기능을 유지한 상태에서 효율적인 프리패칭을 위한 방법을 제안한다. FAST 기법에서는 순차 접근을 위한 순차쓰기용 로그 블록을 운용한다. 그러나 기존 방법에서는 순차쓰기용 로그 블록을 하나만 운용함으로써 순차접근이 자주 발생하고, 프리패칭할 객체들이 많은 환경에서는 적응력이 떨어지게 된다. 따라서 본 논문에서는 이 순차쓰기용 로그 블록이 하나 이상이 되도록 FAST 구조를 확장하여 프리패칭에 최대한 활용될 수 있는 방법을 제시한다.

3.2 시스템 구조

본 논문에서 제안하는 하이브리드 저장장치의 구조는 <그림 2>와 같다. 이 그림은 응용 프로그램에서 요구하는 객체가 디스크로부터 주기억장치로 적재되는 과정을 보여준다. 우선 응용 프로그램이 하이브리드 저장소 관리기(hybrid storage manager)에 특정 객체의 접근을 요구하면, 이 모듈에서는 FTL의 도움을 받아 객체가 플래시 메모리 공간에 존재하는지 조사한다. 만약 플래시 메모리에 존재하게 되면 하드 디스크를 참조하지 않고 곧바로 플래시 메모리로부터 데이터를 주기억장치로 적재하게 된다. 반대로 플래시 메모리에 존재하지 않을 경우에는 디스크로부터 객체를 읽어 플래시 메모리에 저장하고, 마지막으로 동일한 내용을 주기억장치에 적재한다.

하이브리드 저장소 관리기의 역할 중 하나는 가까운 미래에 접근될 가능성 높은 객체를 플래시 메모리에 미리 적재하는 프리패칭 기능이다. 이를 위해 하이브리드 저장소 관리기는 순차패턴 마이너(sequential pattern miner)와 프리패칭 컨트롤러(prefetching controller)의 두 가지 특별한 서브 모듈을 포함한다. 순차패턴 마이너는 과거의 객체 참조 순차(object reference sequence)를 대상으로 순차패턴을 추출하는 역할을 한다. 프리패칭 컨트롤러는 순차패턴 마이너에서 생성된 순차패턴을 기반으로 프리패칭할 객체를 결정하고, 이 객체를 시스템 유휴 시간(idle time)에 플래시 메모리에 프리패칭하는 역할을 수행한다.

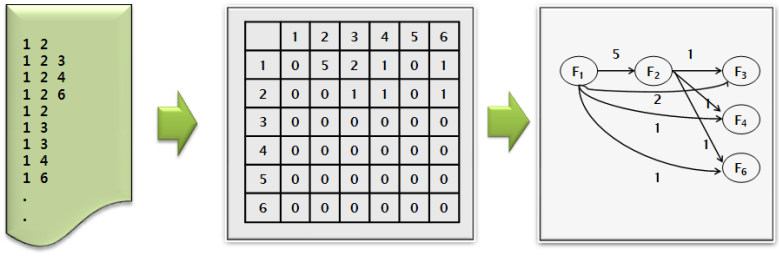


〈그림 2〉 시스템 구조도

3.3 순차패턴의 추출 및 프리패칭 객체의 결정

<그림 2>의 순차패턴 마이너는 주어진 객체들이 참조된 순차 데이터로부터 패턴을 탐색하는 역할을 한다. 앞서 예를 든 바와 같이 <표 1>의 순차 리스트로부터 이 모듈은 $< F_5, F_4, F_1 >$과 같은 패턴을 생성한다. 이때 주어진 패턴으로부터 어떠한 객체를 프리패칭할 지를 결정해야한다. 예를 들어 $< F_5, F_4, F_1 >$과 같은 패턴이 주어지고, F_5에 대한 참조가 이루어졌을 때, F_4와 F_1을 동시에 프리패칭할 수도 있고, F_5와 F_4가 참조되었을 때 비로소 F_1을 프리패칭할 수도 있다. 이를 결정하기 위해서는 두 가지 사항을 고려해야한다. 첫째는 너무 많은 순차패턴으로 인해 프리패칭이 지나치게 자주 발생하는 것을 피해야하며, 둘째는 객체가 참조되었을 때 매번 과거의 참조된 순차들과 패턴들을 일일이 대조해야하는 오버헤드를 피해야한다. 이를 위해 본 논문에서는 순차패턴을 이용하여 프리패칭되는 객체를 바로 다음에는 접근될 객체를 하나만 프리패칭하는 것을 허용함으로써 이러한 문제를 해결한다. 다시 말해서 $< \cdots, F_i, F_j, \cdots >$와 같은 순차패턴에 대해서 F_i가 참조되었을 때, 하나의 F_j만을 프리패칭하고 그 이후의 객체에 대해서는 고려하지 않는다.

프리패칭할 객체를 결정하기 위해 본 논문에서는 <그림 3> (c)와 같은 그래프 구조를 이용하여 해결하였다. <그림 3> (a)는 순차패턴 마이너를 통해 생성된 순차패턴들이라고 가정하자. 이 패턴 집합은 <그림 3> (b)와 같이 2차원 배열로 표현 할 수 있다. 이 배열에서 i행 j열의 값이 k라는 것은 모든 순차패턴 중에서 $< \cdots, F_i, F_j, \cdots >$형태와 같이 F_i와 F_j가 연속적으로 나타나는 빈도수가 k번이라는 것을 의미한다. 따라서 최종적으로 이 배열은 <그림 3> (c)와 같이 가중치 그래프로 표현 가능하다. 이 그래프에서 노드(node)는 객체들을 의미하며, 에지(edge)의 숫자는 노드들의 쌍으로 나타나는 패턴들의 빈도수를 의미한다. 이러한 빈도수는 본 논문에서는 프리패칭할 객체를 선정하기 위한 가중치 정보로 활용한다. 예를 들어 <그림 3> (c)에서 객체 F_1이 참조되었을 경우 F_1 다음



(a) 순차패턴 리스트 (b) 순차패턴의 가중치배열 (c) 가중치 그래프

〈그림 3〉 순차패턴의 데이터 구조

에 참조될 가능성이 가장 높은 것은 F_2, F_3, F_4(또는 F_5)의 순이므로, 프리패칭할 하나의 객체만을 선정한다면 F_2가 되고, 그 이상일 경우에는 위의 순서대로 해당 객체들을 프리패칭하게 된다. 이러한 가중치를 사용하는 이유는 플래시 메모리의 저장 공간이 상대적으로 하드 디스크보다 적기 때문이다. 만약 가중치를 사용하지 않고 발견된 모든 패턴을 적용하여 미래에 사용될 객체를 플래시메모리에 적재한다면 페이지 부재(page fault)가 빈번히 발생할 것이다. 이러한 페이지 부재는 실제 사용될 객체에 대해서 페이지 교체(page replacement)를 발생시킬 수 있으며, 결과적으로 전체적인 성능이 저하되는 결과를 초래한다.

페이지 교체의 빈도는 프리패칭할 객체를 선정하기 위해 가중치 정보의 임계점(threshold)을 조정함으로써 해결할 수 있다. 즉, 모든 객체를 프리패칭하는 것이 아니고 일정 수준의 가중치를 갖는 객체에 대해서만 프리패칭을 수행한다. 예를 들어, 위의 예에서 임계값이 1이라면 F_2만을 프리패칭하게 되고, 임계값이 2라면 F_2와 F_3를 프리패칭하게 된다.[2)]

프리패칭의 대상이 되는 객체는 파일이나 블록이 될 수 있다. 따라서 상황에 따라 파일 전체를 프리패칭할지 아니면 파일 내의 블록 단위로 프리패칭할지를 결정해야한다. 이를 해결하기 위한 알고리즘이 <그림 4>에 제시되어 있다. 우선 응용 프로그램이 특정 파일 F내의 블록 b를 요청하였다고 가정하자. 그러면 <그림 3>의 그래프 구조로부터 프리패칭할 대상되는 파일 F내의 블록들(이를 <그림 4>에서 $prefetch(b)$로 정의함)을 결정하고 프리패칭을 수행한다. 만약 파일 F의 블록 b가 이 파일 내에서 최초로 접근된 블록이라면 동시에 파일단위 패턴으로부터 파일 F 다음에 프리패칭할 다음 파일들(이를 <그림 4>에서 $prefetch(F)$로 정의함)을 선별한다. 이때 $prefetch(F)$내의 각 파일 F_i에 대해서, F_i내에 블록 수준의 순차패턴이 존재하는지 조사한다. 블록 수준의 순차패턴이 존재하게 되면 블록 단위로 프로패칭을 수행한다. 이때 블록 단위의 프리패칭을 위해서 파일 F_i내의 블록 단위 패턴 중 시작패턴의 빈도수가 가장 큰 객체-예를 들어 <그림 3> (a)에서 1번 블록-를 프리패칭한다. 만약 파일 F_i 내에 블록 단위 패턴이 존재하지 않을 경우에는 파일 전체를 프리패칭할지를 결정한다. 이때 판단해야할 중요한 사항이 파일의 크기이다. 만약 프리패칭할 파일의 크기가 특정 크기보다 클 경우에는 상대적으로 용량이 플래시 메모리의 특성을 고려하여 프리패칭 대상에서 제외된다.

3.4 프리패칭을 위한 FAST 구조의 개선

본 절에서는 FAST 구조에서 객체가 프리패칭되는 과정을 설명한다. <그림 5>는 변경된 FAST 구조를 나타낸다. 이 구조와 <그림 1>에 나타난 본래의 FAST 구조의 가장 큰 차이점은 순차쓰기용 로그 블록의 수를 한개 이상 설정할 수 있다는 점이다. 본 논문에서 순차쓰기용 로그 블록을 한 개 이상 가정한

2) 본 논문에서 실험한 결과로는 임계점이 2일 경우 최적의 성능을 나타내는 것으로 나타났다.

```
PROCEDURE Select_Objects_to_be_Prefetched
INPUT: A requested block b in file F
BEGIN
   Search a set of blocks prefetch(b) to be prefetched after b is requested
   if(prefetch(b) is not empty)
       Prefetch all blocks in prefetch(b)
   if(b is the first block requested F)
   {
        Search a set of file prefetch(F) to be prefetched after F is requested
        if(prefetch(F) is not empty)
        {
            for each F_i in prefetch(F)
            {
                  if(There are block-level patterns in F_i)
                      Prefetch a block with a highest frequency among initial blocks in the patterns
                                                          // block-level prefetching
                  else if(Size of the file F_i < α)  // for example, α is 1024kbytes
                      Prefetch all blocks in F_i                  // file-level prefetching
            }
        }
   } }
END
```

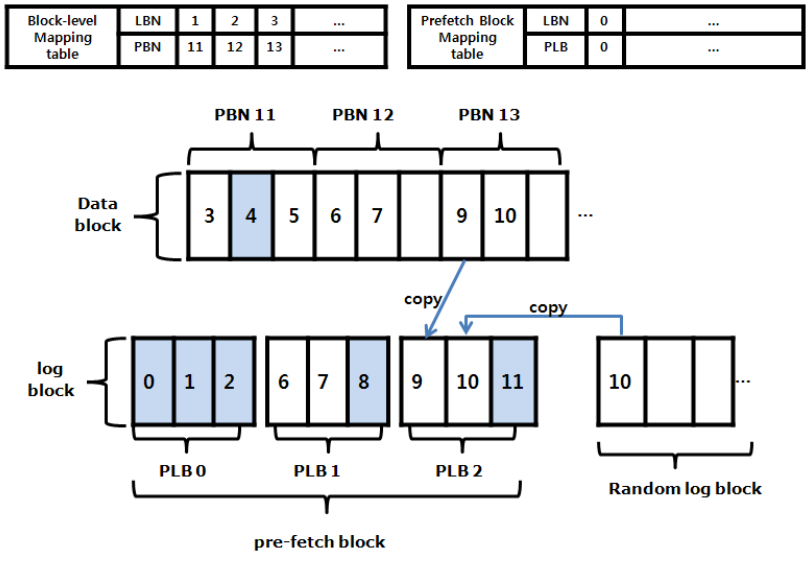
〈그림 4〉 프리패칭 알고리즘

이유는 순차쓰기용 로그 블록을 프리패칭을 위한 블록으로 동시에 사용하여 효율성을 높이기 위해서다. 따라서 본 논문의 이후부터는 순차쓰기 로그 블록과 프리패칭 블록(prefetching block)을 같은 의미로 사용한다. 프리패칭 하기로 결정된 각 블록에 대해서 프리패칭하는 과정은 현재 플래시 메모리에 저장된 기존의 데이터에 따라 달라진다. 우선 프리패칭할 블록이 이미 프리패칭 블록 영역에 적재되어 있다면 아무 일도 하지 않고 종료된다. 그렇지 않은 경우에 대해서는 다음과 같이 4가지 경우로 나누어서 처리된다.

첫째로 프리패칭할 블록의 어떠한 섹터들도 데이터 블록과 로그 블록에 존재하지 않는 경우를 보자. 이 경우는 블록 전체를 디스크에서 플래시 메모리로 적재하게 된다. 이때 적재하는 위치는 데이터 블록 영역이 아닌 프리패칭 블록 영역이 된다. 해당 블록을 프리패칭 블록으로 적재하는 이유는 프리패칭되는 블록이 접근될 수 있다는 확률만 존재하고 이전에 접근된 흔적이 없어 그 가능성을 상대적으로 낮게 취급하기 위해서다. 따라서 실제 접근이 이루어져 현재 데이터 블록 영역에 적재된 블록들에는 영향을 미치지 않게 된다. 실제 이후의 나머지 경우에 대해서는 프리패칭될 블록의 일부 섹터들이 플래시 메모리에 이미 적재되어 있으므로 첫 번째 경우보다는 접근 확률을 높게 보아 모두 데이터 블록 영역으로 이동된다. 만약 모든 프리패칭 블록 영역이 사용 중일 경우는 기존

의 버퍼 관리 알고리즘이나 FAST에서 사용한 원형 큐(circular queue)와 같은 방식으로 교체할 블록을 선정하여 빈 블록을 확보한 후 사용한다. 예를 들어 <그림 5>에서 LBN 0을 프리패칭할 경우 블록매핑 테이블에서 LBN 0에 해당하는 PBN이 존재하지 않는다. 따라서 프리패칭 블록 영역의 한 블록을 할당받아 LSN 0, 1, 2를 적재하게 된다. 이 그림에서 프리패칭 블록 영역에 적재된 섹터들 중에서 음영으로 표시된 섹터들은 프리패칭 과정에서 디스크로부터 새롭게 적재된 섹터를 의미한다. 프리패칭 블록 영역의 매핑을 위해서는 <그림 5>와 같이 블록 매핑 테이블 이외에 프리패칭 블록 매핑 테이블(pre-fetch block mapping table)이 별도로 필요하다. 그 이유는 기존의 FAST에서는 순차쓰기용 로그 블록이 하나이므로 이에 대한 매핑 테이블이 필요 없었지만 <그림 5>와 같이 개선된 구조에서는 하나 이상의 프리패칭 블록을 둘 수 있으므로 LBN과 프리패칭 블록의 매핑을 위한 별도의 테이블이 요구된다. 이 그림에서 프리패칭 블록 매핑 테이블의 상태는 LBN 0를 프리패칭 블록 영역의 PLB (Prefetch Log Block) 0에 적재한 후의 모습을 나타낸다.

두 번째 경우는 블록의 일부 또는 모든 섹터들이 데이터 블록 영역에 존재하고 로그 블록 영역에는 존재하지 않는 경우이다. 이때는 아직 적재되지 않은 일부 섹터들을 비어있는 해당 데이터 블록 영역으로 적재하고, 모든 섹터가 이미 적재되어 있다면 아무 일도 하지 않고 종료된다. 예를 들어 <그림 5>에서 LBN 1을 프리패칭 할 경우에 이와 매핑되는 PBN 11에는 이미 LSN 3과 LSN 5가 저장되어 있다고 가정하자. 이때에는 이 블록의 비어있는 섹터인 LSN 4만을 디스크로부



〈그림 5〉 개선된 FAST 구조에서의 프리패칭 과정

터 프리패칭하면 된다.

세 번째 경우는 블록의 일부 섹터가 데이터 블록 영역과 프리패칭 블록 영역(즉, 순차쓰기용 로그 블록)에 동시에 존재하는 경우이다. 프리패칭 블록에 전체가 아닌 일부 섹터가 존재한다는 것은 프리패칭된 블록이 아니라 기존 FAST 알고리즘에서 덮어쓰기 연산에 의해 순차쓰기용 로그 블록에 기록된 블록임을 의미한다. 이 경우에는 데이터 블록 영역의 최신 섹터를 프리패칭 블록 영역에 확보된 블록에 복사하고 나머지 섹터들을 디스크로부터 적재한다. 이렇게 하나의 프리패칭 블록 영역에 채워진 블록은 데이터 블록과의 교환연산을 통해서 데이터 블록으로 이동하게 된다. 예를 들어 <그림 5>에서 LBN 2를 프리패칭할 때 이와 매핑되는 PBN 12에는 LSN 6과 7이 이미 적재되어 있다. 하지만 LSN 6과 7은 덮어쓰기 연산이 이미 발생하여 프리패칭 블록에 최신 데이터가 쓰여진 상태이다. 따라서 LSN 8를 프리패칭 블록에 적재한다.

마지막으로 네 번째는 블록의 일부 섹터가 데이터 블록 영역과 임의쓰기용 로그 블록에 동시에 존재하는 경우이다. 이때는 프리패칭 블록 영역의 한 블록을 할당받아 여기에 데이터 블록 영역과 로그 블록에 있는 최신 섹터만을 복사하고 나머지는 디스크로부터 적재한다. 임의쓰기용 로그 블록에 남아있는 섹터들에 대해서는 FAST에서와 마찬가지로 해당 섹터를 무효화(invalid) 시킨다. 이렇게 채워진 블록은 세 번째 경우와 마찬가지로 데이터 블록과의 교환연산을 통해서 데이터 블록으로 이동하게 된다. 예를 들어 <그림 5>에서 LBN 3을 프리패칭할 때 PBN 13에는 LSN 9와 10이 이미 저장되어 있고, 또한 임의쓰기용 로그 블록에도 최신 데이터인 LSN 10이 저장되어 있다. 따라서 이 경우에는 프리패칭 블록 영역에서 하나를 할당받아 PBN 13에 저장된 LSN 9와 임의쓰기용 로그 블록의 최신 데이터인 LSN 10을 여기에 복사하고, 아직 플래시 메모리에 적재되지 않은 LSN 11을 프리패칭한다. 마지막으로 이렇게 완성된 블록을 데이터 블록과 교환연산을 수행한다.

4. 성능 평가

4.1 실험 환경

본 논문에서 제안한 프리패칭 기법의 성능을 평가하기 위해서 512Mbytes 용량의 플래시 메모리를 장착한 하이브리드 저장장치에서의 객체 프리패칭 시뮬레이터를 구현하였다. 실험을 위해 가정한 시스템의 상세한 구성은 <표 2>와 같다. 또한 패턴 추출을 위한 순차패턴 마이너의 기본 방법으로는 PLWAP 알고리즘을 사용하였다. PLWAP은 지금까지 제안된 여러 알고리즘 중에서 가장 좋은 성능[3]을 나타내는 것으로 알려져 있다[5].

3) PLWAP의 성능은 순차 데이타의 수와 최대빈발순차 데이터의 길이의 곱에 비례한다. 구체적으로, PLWAP의 시간복잡도는 O(N+n)(L+l)이다. 여기서 N은 전체 순차(sequence) 데이터의 수, L은 최대빈발순차(longest frequent sequence) 데이터의 길이, n은 유입데이터(incremental data)에 존재하는 순차 데이터의 수, l은 유입데이터에 존재하는 최대길이순차(longest sequence) 데이터의 길이를 의미한다.

〈표 2〉 실험 환경

플래시 메모리		FAST	
플래시 메모리 용량	512Mbytes	데이터 블록 크기	192-416Mbytes
페이지 크기	512bytes	임의쓰기용 로그 블록 크기	64Mbytes
읽기 속도	200μs	프리패칭 블록의 크기	32-256Mbytes
쓰기 속도	210μs		
소거 속도	1.2sec		

본 논문에서 제안한 기법의 성능을 상대적으로 비교하기 위해서 Top-N 프리패칭 기법을 채용한 FAST 기법을 이용하였다. Top-N 프리패칭 기법은 상위 N개의 자주 사용되는 객체를 주기적으로 프리패칭하는 단순한 방법을 의미한다[11]. 본 실험에서는 이 기법을 FAST+Top-N이라고 명명한다. [11]에 따르면, N값이 100까지는 급격히 적중률(hit ratio)이 증가하지만 그 이후부터는 프리패칭의 효과가 크지 못하다. 게다가 본 실험에서 프리패칭 블록의 전체 크기가 최대 256Mbytes로 제한되어 있기에 N값을 100으로 설정하였다. 또한 본 논문에서 제안한 방법 중에서 파일 수준으로만 프리패칭 기법을 FAST+File-Level이라고 하며, <그림 4>와 같이 파일과 블록을 모두 고려한 프리패칭 기법을 FAST+2-level이라고 한다. 즉, FAST+2-level은 <그림 4>의 알고리즘을 그대로 적용한 기법을 의미하며, FAST+File-Level은 <그림 4>의 알고리즘에서 블록 수준 프리패칭하는 부분을 생략하고 모든 $prefetch(F)$에 대해서 파일 수준의 프리패칭만을 수행하는 기법을 의미한다.

본 실험에서는 두 가지 유형의 데이터를 사용하였다. 한 유형은 유명 포털사의 UCC 데이터이고 다른 한 유형은 지역성을 포함하는 합성 데이터(synthetic data)이다. 첫 번째 유형인 유명 포털사의 UCC 데이터는 1개월에 걸쳐 각 사용자들의 UCC에 대한 정보를 담고 있다. UCC 데이터에 1개월 간 사용자들이 접근한 총 파일의 개수는 약 600만 개이며 이러한 데이터를 사용한 사용자의 수는 약 200만 명이다. 파일의 크기는 4kbytes에서 2Mbytes까지 다양하게 분포되어 있다. 두 번째 유형인 합성 데이터는 인위적으로 생성된 데이터이다. 본 실험에서는 합성 데이터의 생성을 위해 [4, 8, 9]에서 제시한 데이터 분포에 따르고 있으며, 이는 플래시메모리를 사용하는 로그기반 파일시스템의 성능을 측정하기 위해 지역성을 반영한 것이다. 본 실험에서 사용한 데이터의 크기는 1Kbyte-10Mbytes범위에서 랜덤하게 1,000개의 파일을 생성하였다. 합성 데이터는 실험을 위해 인위적으로 지역성을 가지도록 생성하였다. 지역성은 'x : y'의 방식으로 표기할 수 있는데. 예를 들어 '2 : 8'은 80%가 전체 데이터의 20%에 집중되고 그 나머지 20%는 80%의 다른 데이터 블록 영역에 접근함을 의미한다[4]. UCC 데이터와 합성데이타에 대한 성능 측정을 위해 접근한 객체 수는 각

각 150,000개와 50,000개로 고정하였다

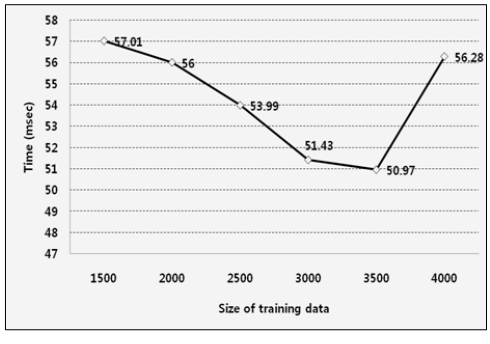
4.2 실험결과

본 실험에서는 네 가지 관점에서 실시하였다. 첫째는 순차패턴을 생성하기 위해서 최적의 학습 데이터량을 결정하기 위한 실험을 실시하였다. 둘째는 합성 데이터에 대해서 지역성의 변화에 따른 성능 변화를 측정하였다. 다음으로는 패턴 가중치의 임계점 변화에 따른 성능을 측정하였고 마지막으로 프리패칭 블록 영역의 크기에 따른 변화를 평가하였다.

4.2.1 학습 데이터의 크기 변화에 따른 성능 평가

본 실험에서는 학습 데이터의 크기에 따른 성능을 비교한다. 상식적으로 학습 데이터의 크기가 커질수록 세분화된 패턴들이 생성될 것으로 기대할 수 있다. 예를 들어 학습 데이터가 2,000개일 경우 추출된 패턴이 $< F_2, F_1 >$이라면 4,000개에서 추출되는 패턴은 더 세분화된 $< F_2, F_3, F_5, F_1 >$이 될 것이다. 그러나 일반적으로 기계학습 알고리즘에서 예측 모델을 생성할 때 학습 데이터가 크다고 해서 항상 더 나은 모델을 생성하는 것은 아닌 것으로 알려져 있다[9].

<그림 6>은 학습 데이터 크기에 따라 다르게 생성되는 순차패턴에 따른 성능 결과이다. 이 그림에서 한 명의 사용자가 하나의 파일을 접근하는 것을 하나의 학습 데이터로 간주하며, 실행시간은 응용 프로그램이 요청한 파일이 주기억장치로 업로드 되는 시간을 의미한다. 이 그림에서 볼 수 있듯이 초기에는 학습 데이터의 크기가 커짐으로써 패턴이 세분화 되면 성능이 향상된다는 것을 알 수 있었다. 합성 데이터와 UCC 데이터는 학습 데이터의 크기가 각각 3500개, 40000개 일 때 최적의 성능 향상을 보였다. 반면에 학습 데이터의 크기가 그 이상 커질 경우 성능이 현저히 떨어지는 것을 확인할 수 있다. 이유는 많은 객체를 플래시 메모리로 프리패칭하다 보니 페이지 부재가 많이 발생하기 때문이다. 즉 성능 향상의 효과를 갖기 위해서는 학습 데이터의 크기를 어느 정도 확보해야 되지만 지나치게 큰 경우 오히려 성능 향상 비율이 떨어지게 된다. 또한 본 실험의 결과에서 알



(a) 합성 데이터

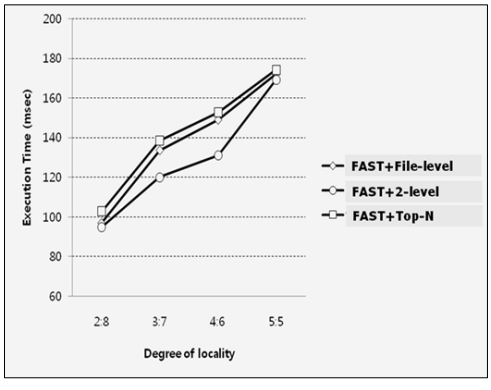
(b) UCC 데이터

〈그림 6〉 학습 데이터의 크기에 따른 성능 비교

수 있듯이 최적의 학습 데이터 크기는 데이터의 종류에 따라 달라질 수 있다. 따라서 도메인별 최적의 학습 데이터크기는 다양한 환경에서의 실험을 통하여 통계적으로 확보해야한다. 본 논문에서는 나머지 실험을 위하여 합성 데이터와 UCC 데이터의 학습 데이터 크기를 각각 3,500개와 40,000개로 설정하였다.

4.2.2 합성 데이터의 지역성에 따른 성능평가

<그림 7>은 합성 데이터에 대해서 지역성의 변화에 따라 성능이 어떻게 변하는 지를 나타낸다. 이 실험을 위해 프리패칭 블록의 크기를 128MB로 고정하였고, FAST+Top-N, FAST+File-level, FAST+2-level에 대해서 성능을 측정하였다. 이 그림에서 보는 바와 같이 지역성이 증가할수록 성능이 선형적으로 증가한다는 것을 알 수 있다. 그 이유는 지역성이 증가할수록 순차패턴이 강하게 나타나므로 프리패칭의 효과가 커지기 때문이다. 그러나 지역성이 지나치게 강할 경우에는 성능 차이가 크게 나타나지 않는다. 그 이유는 극히 일부분 데이터에 대해서만 패턴이 생성되므로 프리패칭하는 블록의 수가 적어서 전반적인 성능에는 크게 영향을 미치지 않기 때문이다. 또한 FAST+Top-N보다 순차패턴에 근거한 FAST+File-level와 FAST+2-level이 더 좋은 성능을 보인다는 것도 확인할 수 있다. 특히 FAST+2-level이 FAST+File-level보다 더 좋은 성능을 보인다. 그 이유는 FAST+File-level의 경우 파일 단위로 프리패칭하므로 파일 내에 실제 참조되지 않는 블록들까지 동시에 프리패칭되어 페이지 부재를 자주 발생시키기 때문이다. 본 실험에서는 지역성이 '4 : 6'일 때 FAST+File-level와 FAST+2-level을 비교했을 때 가장 큰 차이가 발생하였다. 따라서 본 논문에서의 나머지 실험을 위해 참조 블록의 지역성을 '4 : 6'으로 설정하였다.

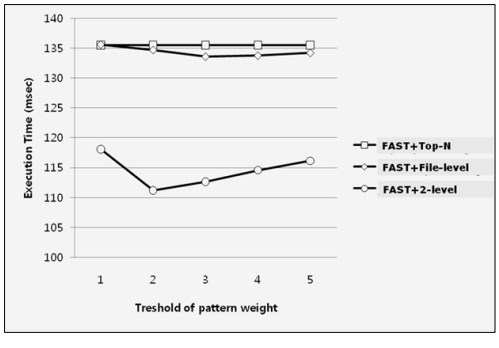


〈그림 7〉 합성 데이터의 지역성 변화에 따른 성능 비교

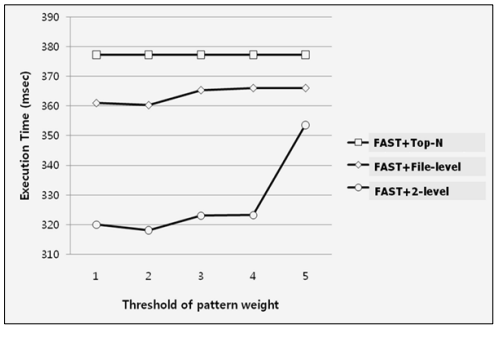
4.2.3 패턴 가중치의 임계점 변화에 따른 성능평가

하이브리드 저장장치의 성능 향상을 결정짓는 중요한 요소 중 하나는 순차패턴을 적용하기 위한 최적의 가중치 임계점을 알아내는 것이다. 가중치 임계값이 너무 낮으면 너무 많은 순차패턴이 발생되어 프리패칭 연산이 빈번히 발생되고 너무 크게 되면 프리패칭 수가 적어 그 효과가 미미하게 된다. <그림 8>은 합성 데이터와 UCC 데이터에 대해서 가중치의 임계값 변화에 따른 성능의 변화를 보여주고 있다. 이 그림에서 보는 바와 같이 FAST+2-level의 경우 가중치의 임계점에 큰 영향을 받으나 FAST+File-level은 임계점의 영향이 크기 않은 것을 알 수 있다. 이

러한 현상은 합성 데이터와 UCC 데이터 모두에서 발견된다. 이와 같이 FAST+2-level의 성능이 가중치 임계점에 민감한 이유는 사용자의 데이터 접근 패턴이 파일단위 보다는 상대적으로 규모가 작은 블록단위에서 더욱 분명히 나타나기 때문이다. 그림에서 보는 바와 같이, 합성 데이터, UCC 데이터의 경우에 가중치가 2일 때 성능이 가장 좋은 것으로 나타났다. 가중치가 너무 클 경우에는 프리패칭되는 객체가 많지 않기 때문에 최적의 가중치와 비교했을 때 성능 차이가 발생한다는 것을 알 수 있다. 본 실험에서는 패턴 가중치의 임계점이 2일 경우 가장 좋은 성능을



(a) 합성 데이터



(b) UCC 데이터

〈그림 8〉 패턴 가중치의 임계점 변화에 따른 성능 비교

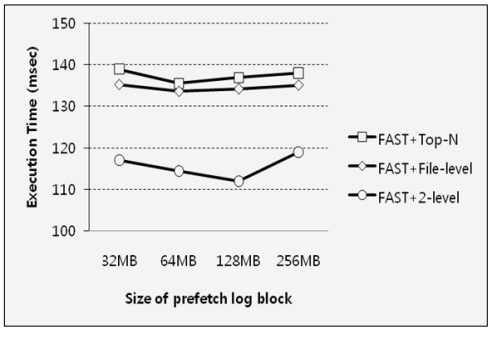
나타냈으나, 데이터의 종류, 특성, 메모리의 크기, 블록의 설정 등에 따라 그 결과가 다양하게 변할 수 있다. 따라서 실제 환경에 적용하기 위한 임계점의 결정은 다양한 환경에서의 실험을 통해서 결정해야할 것으로 판단된다.

4.2.4 프리패칭 블록 영역의 크기 변화에 따른 성능평가

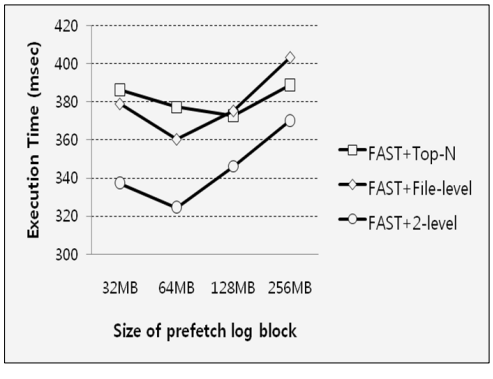
마지막으로 가중치 임계점을 2로 고정한 가운데 프리패칭 블록 영역의 크기의 변화에 따른 성능을 측정하였다. 만약 프리패칭 블록 영역의 크기가 커진다면 프리패칭된 객체를 더 많이 저장할 수 있을 것이다. 하지만 데이터 블록의 크기는 프리패칭 블록의 크기에 반비례한다. 이 때문에 프리패칭 블록의 크기가 너무 클 경우 데이터 블록의 페이지 교체에 따른 연산이 자주 발생할 것이다. 플래시 메모리는 읽기, 쓰기에 비해 지우기 속도가 현저히 떨어진다. 이 때문에 성능 향상을 위해서는 이러한 페이지 교체로 발생되는 지우기 연산을 최소로 하는 동시에 되도록 많은 객체가 프리패칭될 수 있는 프리패칭 블록의 크기를 찾아야 할 것이다.

<그림 9>는 합성 데이터와 UCC 데이터에 대해서 프리패칭 블록 영역의 크기에 따른 성능의 변화를 보여준다. 이 그림에서 보는 바와 같이 합성 데이터와 UCC 데이터는 각각 128MB와 64MB에서 가장 좋은 성능을 보인 것으로 나타났다. 이 성능 결과에 가장 중요한 영향을 미친 것은 페이지 교체 횟수가 이 크기에서 가장 적게 나타났기 때문이다. 단순히 생각할 때에는 프리패칭 블록은 합병

연산보다는 교환연산이 발생하므로 크기가 클수록 성능이 좋을 것으로 볼 수 있다. 하지만 프리패칭 블록 영역이 커지면 데이터 블록 영역이 상대적으로 작아지므로 성능이 오히려 떨어지게 된다. 따라서 제한된 플래시 메모리 크기에 대해서 프리패칭 블록 영역의 크기를 얼마로 결정하느냐가 최적의 성능을 보이는데 중요한 역할을 하게 된다.



(a) 합성 데이터



(b) UCC 데이터

〈그림 9〉 프리패칭 블록 크기 변화에 따른 성능 비교

5. 결 론

본 논문에서는 플래시 메모리를 함유한 하이브리드 저장장치의 성능 향상을 목적으로 순차패턴 마이닝을 이용한 프리패칭 기법을 제안하였다. 제안 기법의 기본 아이디어는 가까운 미래에 접근될 것으로 예측되는 객체를 순차패턴 마이닝 기법으로 예측한 후, 이 객체를 사전에 플래시 메모리로 프리패칭하는 것이다. 하이브리드 저장장치 내부에서 캐시 공간 역할을 하는 플래시 메모리의 입출력 데이터 매핑을 위해 기존 FAST 알고리즘을 변형하였고, 제한된 플래시 메모리의 공간을 효율적으로 사용하기 위하여 프리패칭 단위로서 파일 수준과 블록 수준을 동시에 고려하였다. 즉 파일 수준과 블록 수준에서 순차패턴을 추출하였으며, 프리패칭을 위해 2단계로 구성된 순차패턴에 대해 세밀한 패턴매칭 절차를 수행함으로써 최선의 대상 파일 및 블록을 결정한다. 제안 기법의 효용성을 평가하기 위해 지역성을 가지는 합성 데이터와 UCC 데이터를 활용하였고 실험을 통하여 파일과 블록을 모두 고려한 프리패칭 기법이 우수한 성능을 나타내는 것을 확인하였다. 최적의 성능을 얻기 위해서는 프리패칭 대상을 결정하기 위한 패턴 가중치 임계점, 학습 데이터의 양, 프리패칭 블록의 크기에 있어서 최적치를 결정하는 것이 중요하며, 향후 이를 효과적으로 결정하는 방안을 연구할 것이다.

참 고 문 헌

[1] Agrawal R., Srikant R., “Mining sequential patterns,” Proceedings of the

11th International Conference on Data Engineering (ICDE'95), 1995, pp. 3-14.

[2] Agrawal R. S. R., "Mining sequential patterns : Generalizations and performance improvements," Proceedings of the Fifth International Conference On Extending Database Technology(EDBT '96), 1996, pp. 3-17.

[3] Bae Y. H., "Design Technique of High Performance Flash Memory SSD," Journal of Korean Institute of Information Scientists and Engineers, Vol. 25, No. 6, 2007, pp. 18-28.

[4] Chiang M. L., Lee Paul C. H. and Chang R. C., "Using data clustering to improve cleaning performance for flash memory," Software Practiceand Experience, Vol. 29, No. 3, 1999, pp. 267-290.

[5] Ezeife C. I. and Lu Y., "Mining web log sequential patterns with position coded pre-order linked wap-tree," International Journal of Data Mining and Knowledge Discovery, Vol. 10, No. 1, 2005, pp. 5-38.

[6] Gal E. and Toledo S., "Algorithms and Data Structures for Flash Memories," ACM Computing Surveys, Vol. 37, No. 2, 2005, pp. 138-163.

[7] Han J., Pei J., Mortazavi-Asl B., Chen Q., Dayal U., and Hsu M., Freespan : Frequent pattern-projected sequential pattern mining, Proceedings of the 2000 Int. Conference on Knowledge Discovery and Data Mining(KDD'00), 2000, pp. 355-359.

[8] Kim H. J. and Lee S. G., "An Effective Flash Memory Manager for Reliable Flash Memory Management," IEICE Transactionson Information and Systems, Vol. 85, No. 6, 2002, pp. 950-964.

[9] Lee S. W., Park D. J., Ching T. S. Lee D. H., Park S. W. and Song H. J., "A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation," ACM Transactions on Embedded Computing Systems, Vol. 6, No. 3, 2007, pp. 18-44.

[10] Li Z., Chen Z., Srinivasan S. M. and Zhou Y., "C-Miner : Mining Block Correlations in Storage Systems," Proceedings of the 3rd USENIX Conference on File and Storage Technology (FAST'04), 2004, pp. 173-186.

[11] Markatos E. P. and Chronaki C., "A Top-10 Approach to Prefetching on the Web," Proceedings of the INET 98 Conference, 1998.

[12] Min S. L. and Nam E. H., "Current trends in flash memory technology : invited paper," Proceedings of the 2006, conference on Asia South Pacific design automation, 2006, pp. 332-333.

[13] Masseglia F., Poncelet P. and Cicchetti R., "An efficient algorithm for web usage mining," Networking and Information Systems Journal, Vol. 2, No. 5-6, 1999, pp. 571-603.

[14] Pei J., Han J., Mortazavi-Asl B. and Zhu H., "Mining access patterns efficiently from web logs," Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'00), 2000, pp. 396-407.

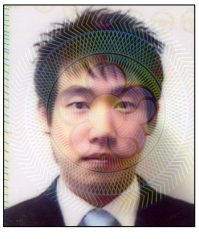
[15] Tan P. N., Steinbach M. and Kumar V., "Introduction to Data Mining," Addison -Wesley, 2006.

[16] Zaki M. J., “SPADE : An efficient algorithm for mining frequent sequences,” Machine Learning, Vol. 42, No. 1/2, pp. 31-60.

[17] Hybrid drive : Wikipedia, http://en.wikipedia.org/wiki/Hybrid_drive.

저 자 소 개

장재영 (E-mail : jychang@hansung.ac.kr)
1992년 서울대학교 계산통계학과 (이학사)
1994년 서울대학교 계산통계학과 (이학석사)
1999년 서울대학교 계산통계학과 (이학박사)
2000년~현재 한성대학교 컴퓨터공학과 부교수
관심분야 데이터베이스, 정보검색, 데이터마이닝



윤언근 (E-mail : ds5eqe@hanmain.net)
2000년 경기대학교 전자계산학과 (공학사)
2010년 서울시립대학교 전자전기컴퓨터공학부 대학원 (공학석사)
2010년~현재 텔로드 연구원
관심분야 데이터베이스, 스토리지 시스템, 데이터마이닝

김한준 (E-mail : khj@uos.ac.kr)
1994년 서울대학교 계산통계학과 (공학사)
1996년 서울대학교 전산과학과 (공학석사)
2002년 서울대학교 컴퓨터공학부 (공학박사)
2002년~현재 서울시립대학교 전자전기컴퓨터공학부 부교수
관심분야 정보검색, 텍스트마이닝, 데이터베이스, 기계학습, e-비즈니스 기술