Journal of The Korea Society of Computer and Information
Vol. 19, No. 7, July 2014
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2014.19.7.113

BLOCS: 블록 상관관계를 인지하는 시퀀스 패턴 마이닝 기반 하이브리드 스토리지 캐슁 알고리즘

이 성 진*, 원 유 집**

BLOCS: Block Correlation Aware Sequential Pattern Mining based Caching Algorithm for Hybrid Storages

Seongjin Lee*, Youjip Won**

요 약

본 논문은 SSD를 캐쉬로 사용하는 하이브리드 저장장치에서 캐쉬에 저장할 데이터를 찾기 위한 BLOCS 기법을 제안한다. 시퀀스 패턴 마이닝을 사용하는 BLOCS 기법은 파일시스템에서 호출하는 섹터들의 연관성을 발생한 순서를 고려하여 빈번히 요청되는 섹터들의 집합을 생성한다. 비교 분석을 위해 탐색거리(DIST) 기반 기법과 요청 빈도(FREQ) 기반 기법 그리고 빈도와 크기의 곱(F-S) 기반 기법을 제안하였다. 제안한 캐슁 기법을 평가하기 위해 하이브리드 캐슁 시뮬레이터를 개발하여 적중률과 응답시간 정보를 얻는다. 부팅 시 발생하는 I/O의 흐름자료와 10개의 응용프로그램들의 실행 시나리오에서 발생한 I/O 흐름자료를 수집하여 캐쉬 시뮬레이터의 입력으로 사용하였다. 실험 결과 부팅 흐름자료에서 제안한 BLOCS 기법이 61%의 적중률을 나타내서 적중률이 가장 낮았던 거리 우선 기반 기법에 비해 15% 더 높은 적중률을 보였다.

▸ Keywords : 시퀀스 패턴 마이닝, 캐슁 알고리즘, 하이브리드 스토리지, 캐쉬 시뮬레이터

Abstract

In this paper, we propose BLOCS algorithm to find sequence of data that should be saved in cache device of hybrid storage system which uses SSD as a cache device. BLOCS algorithm which uses a sequence pattern mining scheme, creates a set of frequently requested sectors with respect to requested order of sectors. To compare the performance of the proposed scheme, we introduce

•제1저자 : 이성진 •교신저자 : 원유집
•투고일 : 2014. 6. 11, 심사일 : 2014. 6. 20, 게재확정일 : 2014. 7. 9.
* 한양대학교 전자컴퓨터통신공학과(Department Of Electronics and Computer Engineering)
** 한양대학교 컴퓨터소프트웨어학과(Department Of Computer and Software)
※본 연구는 지식경제부 및 한국산업기술평가관리원의 산업원천기술개발사업(정보통신)의 일환으로 수행하였음.
〔No.10035202, 대용량 MLC SSD 핵심기술 개발〕

Distance (DIST) based scheme, Request Frequency (FREQ) based scheme, and Frequency times Size (F-S) based scheme. We measure the hit ratio and I/O latency of different caching schemes using hybrid storage caching simulator. We acquired booting workload along with ten scenarios of launching applications and use the workloads as input to the cache simulator. After experiment with booting workload, we find that BLOCS scheme gives hit ratio of 61% which is about 15% higher than the least performing DIST scheme.

▸ Keywords : Sequence pattern Mining, Caching Algorithm, Hybrid Storage, Cache Simulator

I. 서 론

최근 등장한 낸드플래시 기반의 SSD(Solid State Drive)는 HDD 대비 수배 빠른 고속의 입출력이 가능하여 HDD를 대체할 저장장치로 빠른 I/O 입출력 성능을 원하는 사용자들의 관심을 얻고 있다. HDD[1]의 대역폭은 기계적 제한으로 인해서 입력과 출력에 상관없이 초당 144MB를 처리하는 반면, SSD[2]의 경우 낸드 플래시의 물리적 특성으로 인해 비대칭적인 입출력 속도를 갖고 있다. SSD는 입력과 출력에 따라 초당 530MB와 330MB를 처리할 수 있다. 입출력 처리 속도 뿐만 아니라 소음과 진동, 그리고 소비 전력도 HDD에 비해 낮음에도 불구하고 아직까지 SSD가 대중화가 되고 있지 못한 주요 원인 중에는 용량과 가격이 큰 부분을 차지하고 있다.

SSD의 가격을 HDD와 경쟁할 수 있는 수준으로 낮추면서 용량을 늘리기 위한 방법으로 한 셀에 여러 비트를 저장하는 기술을 들 수 있다. 과거에는 하나의 셀에 하나의 비트를 저장하는 SLC(Single Level Cell)[3] 유형의 낸드 플래시의 구조였지만, 현재 두 개의 비트를 하나의 셀에 저장하는 MLC(Multi Level Cell)[4] 유형의 셀을 사용하도록 바뀌었다. 최근에는 하나의 셀에 세 개의 비트를 저장하는 TLC(Triple Level Cell)기술을 사용하는 SSD도 등장을 하고 있다. 하지만, 한 셀에 더 많은 비트를 저장할수록 셀의 수명은 더욱 짧아지는 문제가 있기 때문에, 최근에는 낸드 플래시를 겹층으로 쌓아 올리는 방식의 3D V-NAND[5]의 등장하여 용량 문제에 대한 대안이 등장하였다.

SSD의 MLC와 TLC 낸드 플래시를 사용하여 용량은 늘어나고 가격은 점차 낮아지고 있다. SSD의 가격은 지속적으로 낮아지고 있는 추세로 2012년 말에는 $0.48/GB까지 가격이 하락하였다[6]. 가격 문제가 해결이 된다고 하더라도 SSD가 HDD를 완전히 대체되지는 않을 것이라고 Gartner는 전망하고 있다[7]. 낸드 플래시는 셀 마모도와 data retention[8] 문제가 있어서 SSD 만을 이용하는 컴퓨팅 환경에 대한 우려가 있기 때문이다.

그림 1은 다양한 구성의 스토리지 시스템을 적용한 컴퓨팅

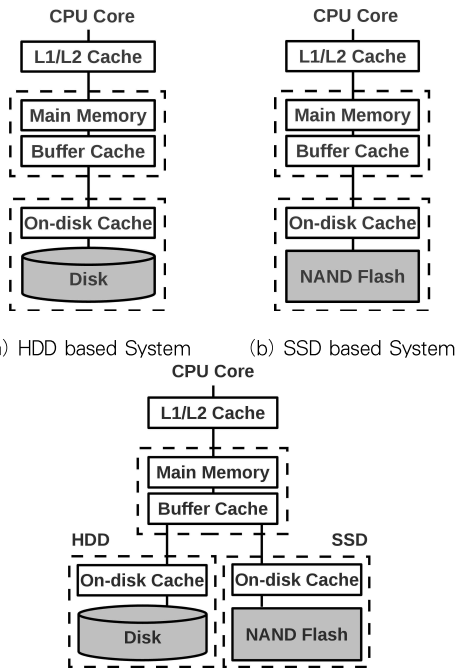

그림 1. HDD, SSD, 그리고 Hybrid 저장 장치를 이용한 시스템의 구성

Fig. 1. System Configuration with HDD, SSD, and Hybrid Storage System

시스템의 예이다. 이제까지 잘 알려진 구성은 HDD 또는 SSD를 사용하는 단일 저장장치의 시스템이다. HDD만으로 구성된 시스템은 성능 면에서는 SSD만으로 구성된 시스템에 비해 낮지만 가격이 저렴하여 보편적으로 사용되고 있다. 낮은 가격과 대용량 저장 공간이라는 두 가지의 장점을 모두 얻는 방법은 그림 1(c)에서 나타낸 것과 같이 HDD와 SSD를 동시에 사용하는 하이브리드 스토리지 시스템을 사용하는 것이다. 하이브리드 스토리지 시스템에 장착된 SSD는 HDD의 캐쉬 디스크로 사용된다. 캐쉬 디스크인 SSD에 사용자가 자주 사용하는 응용프로그램의 핵심 데이터와 실행을 위한 라이브러리들을 저장한다. 이러한 시스템의 장점은 자주 접근되는 데이터가 SSD에 저장이 되어 있어서 HDD의 낮은 입출력 성능을 보완하는 것이다.

본 연구에서는 SSD를 캐쉬로 사용하는 하이브리드 스토리지 시스템을 위한 시퀀스 패턴 매칭 기반의 캐싱 알고리즘인 BLOCS와 캐싱 알고리즘의 성능을 평가할 수 있는 캐쉬 시뮬레이터를 개발하였다. 시퀀스 패턴 마이닝은 연속적으로 발생한 사건들의 상관관계를 분석하여 미래에 일어날 사건을 예측하는 기법이다. BLOCS 캐쉬 알고리즘으로 선정한 데이터는 캐쉬 저장장치로 사용되는 SSD에 기록된다. 캐쉬로 사용되는 저장장치의 크기는 HDD에 비해 한정적이기 때문에 4가지 교체정책(마이닝 기반, 크기 우선, 빈도 우선, 거리 우선) 정책도 제시하여 비교하였다. 시퀀스 패턴 마이닝을 이용하여 Booting 워크로드에서 BLOCS와 마이닝 기반의 교체정책을 사용하였을 때 다른 교체정책보다 5%가 높은 58%의 적중률을 보였다.

2장에서는 관련연구를 소개하고, 3장에서는 패턴 마이닝 기반 캐싱 알고리즘의 동작 방식을 설명한다. 4장과 5장은 구현한 알고리즘을 평가하기 위한 캐쉬 시뮬레이터를 설명하고 캐쉬 시뮬레이터에 입력으로 사용한 워크로드를 분석한다. 6장에서 실험 결과를 분석한 후 7장에서 결론을 맺는다.

II. 관련 연구

HDD를 주로 사용하는 시스템에 SSD를 캐쉬용 저장장치로 사용할 때 얻을 수 있는 장점은 전력 소비 감소와 응용프로그램의 실행시간 단축이다.

캐쉬를 사용하는 시스템의 전력 소비를 줄이는 연구로는 Kgil et al.[9]의 연구가 있다. 읽기와 쓰기를 위한 구분된 영역을 갖는 낸드 플래시 기반의 디스크 캐쉬를 서버에 적용하여 HDD기반 시스템에 비해 전력을 3배 줄일 수 있었다. Hsieh et al.[10]의 연구에서는 해쉬 기반의 LBA 관리 테이블을 사용하는 기법을 사용하여 에너지를 20%를 줄이고 응답시간의 2/3을 줄이는 시스템을 제안하였고 트레이스 기반 시뮬레이션으로 검증 하였다.

낸드 플래시가 HDD보다 쓰기 응답시간이 짧은 점을 이용하여 Bisson et al.[11]은 HDD의 쓰기 지연시간을 70%까지 줄일 수가 있었다. 응용프로그램의 실행시간을 줄이기 위한 노력으로 Joo et al.[12,13]은 응용프로그램이 실행할 때 읽는 파일들의 시퀀스 중 일부를 SSD에 저장하였다. 어떤 파일들을 선택적으로 SSD에 저장할 지는 정수 선형 프로그래밍을 사용하여 최적화하였다.

클라이언트와 서버 모델에서 플래시 메모리를 캐쉬로 사용하는 Koller et al.[14]와 Holland et al.[15]의 연구에서는 플래시 메모리에 적합한 writeback 정책을 분석하였다. 두 연구 모두 synchronous writethrough 방식으로 디스크에 기록할 때 성능이 낮음을 보였다. Koller et al.은 백엔드의 쓰기 요청을 일괄 처리하는 것이 쓰기 처리량을 높이는데 도움이 된다는 것을 보였다. Holland et al.은 플래시로 캐쉬를 사용하는 장점은 쓰기 비율이 높을 때 가장 잘 나타난다고 했다.

캐쉬를 사용할 때 성능에 직접적인 영향을 끼치는 것은 어떤 데이터를 저장을 하느냐의 문제이다. 입출력 성능 개선을 위해 미래에 읽혀질 데이터를 예측하기 위해 시퀀스 패턴 마이닝 기법을 적용한 연구가 있다. Zaki[16]가 제안한 시퀀스 패턴 마이닝 기법인 SPADE 기법은 데이터베이스의 시퀀스가 조합적 특성을 갖고 있는 것을 이용하여 여러 작은 단위 문제로 나누어 시퀀스를 분석하는 방법을 제안하였다. Ayres et al.[17]은 데이터베이스의 원소들의 지지도를 관리하기 위해 비트맵 기법 표현을 사용하였고 이를 이용하여 깊이 우선 검색 방식으로 빈발시퀀스를 찾았다. Yan et al.[18]이 제안한 CloSpan이 사용하는 닫힌 빈발부분시퀀스를 찾는 기법은 지지도가 초시퀀스와 다른 지지도를 갖는 부분시퀀스를 찾기 때문에 결과로 얻게 되는 패턴의 수가 줄어든다.

시퀀스 패턴 매칭 기법을 호스트가 발생시키는 I/O 흐름자료에 적용한 연구도 있다. Li et al.[19]가 제안한 C-miner 기법은 앞서 설명한 CloSpan[18]을 발전시켰다. 또한, 패턴 매칭 기법을 스토리지 시스템이 처리하는 I/O흐름자료에 적용할 수 있도록 cutting window size라는 개념을 도입하였다. Cutting window size는 I/O 흐름자료를 일정크기로 나눈 값을 말한다. Window를 구성하는 방식은 중첩과 비중첩 방식으로 나누는데 Li et al.은 비중첩 방식으로 자르더라도 손실되는 정보가 적다고 주장한다. C-Miner를 Disksim을 이용한 시뮬레이션 환경에서 10000RPM IBM

Ultrastar 36Z15를 모델링한 실험 결과를 보면 응답시간을 워크로드에 따라 12%에서 25% 사이 줄일 수 있었다.

III. 패턴 마이닝 기반 캐슁 알고리즘

본 연구의 캐쉬 알고리즘의 목적은 그림 1(c)과 같은 구조에서 파일시스템에서 호출하는 섹터들의 연관성을 고려하여 요청 빈도가 높은 섹터들의 집합을 얻는 것이다. 개발한 캐쉬 시뮬레이터는 캐쉬 알고리즘을 통해 얻은 섹터 집합의 데이터를 활용할 수 있도록 구현하였다. 섹터들의 연관성을 분석하기 위해 C-Miner[19]에서 사용하는 시퀀스 패턴 마이닝의 빈발부분시퀀스를 근간으로 하였다.

본 논문에서 사용하는 캐쉬 알고리즘은 크게 두 부분으로 나눈다. 첫째 부분은 BLOCS 알고리즘으로 입력 흐름자료를 받아서 자주 요청하는 연관성 있는 섹터들의 집합인 빈발부분시퀀스 L을 생성하는 방법이다. 빈발부분시퀀스 L에 대한 정의는 다음 절에 설명한다. 두 번째 부분은 BLOCS 알고리즘을 통해 선정된 빈발부분시퀀스 L의 우선순위에 따라 캐쉬에 기록하는 방법이다. 먼저 제안하는 BLOCS 알고리즘을 설명하고 캐쉬 시뮬레이터에서 사용할 캐쉬 정책을 설명한다.

1. BLOCS 캐슁 알고리즘

BLOCS의 핵심인 상관관계 분석과 데이터의 재배치는 다음의 과정을 따른다. 먼저, 응용프로그램의 I/O 흐름자료에 시퀀스 패턴 마이닝을 적용하여 I/O들 간의 상관관계를 파악한다. 상관관계 분석으로 얻은 자료를 활용하여 빈번히 발생하는 시퀀스를 하이브리드 저장장치의 SSD와 같은 캐쉬 장치에 재배치한다. 시퀀스 마이닝이 적용되는 분야에 따라 다루는 단위는 다르겠지만, 본질은 빈번히 접근되거나 발생되는 사건 또는 원소의 집합을 발견하는 것이고 또한 그것들의 발생 순서를 찾는 것이다. 이 논문에서는 블록 기반의 흐름자료 수집 도구를 통해 얻은 I/O 흐름자료를 다루고 있기 때문에 이 논문의 시퀀스 마이닝에서 다루는 원소는 응용프로그램이 요청한 I/O의 섹터 주소를 말한다.

그림 2는 BLOCS 알고리즘의 흐름도를 나타내었다. BLOCS의 동작을 설명하기에 앞서 시퀀스 패턴 마이닝에 사용되는 용어들을 정의한다. 집합 F는 원소 f_i로 이루어 졌으며, $F=\{f_1,f_2,...,f_n\}$, 각 f_i는 흐름자료에서 수집한 I/O의 섹터 주소를 나타내고 f_i'로 표기되는 I/O요청 길이에 대한 추가 정보를 갖고 있다. 만일 동일한 섹터 요청 길이에 더 긴 I/O 요청 길이가 존재하면 더 긴 값으로 갱신을 한다.

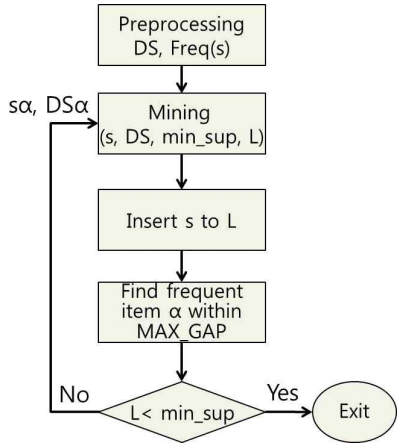

그림 2. BLOCS 알고리즘의 흐름도
Fig. 2. Flow of BLOCS Algorithm

$\sigma=\langle s_1,s_2,...,s_n\rangle$로 표기되는 시퀀스는 순서를 갖는 리스트이며, F의 부분집합이다. $\{s_n \subseteq F\}$.

시퀀스 α가 또 다른 시퀀스 β의 부분시퀀스가 되는 필요충분조건은 시퀀스에 속한 원소 $(i_1,i_2,...,i_m)$의 순서가 변경되지 않으며 $(1 \le i_1 < i_2 <...< i_m \le n)$ 그리고 $a_1 \subseteq b_{i1}$, $a_2 \subseteq b_{i2}$, ... , $a_m \subseteq b_{im}$의 관계가 만족할 때이다. 만약 $\alpha \neq \beta$ 이라고 하면 $\alpha \sqsubset \beta$ 이라고 포함 관계를 표시한다. 만약 시퀀스 α의 원소가 시퀀스 β의 원소와 동일한 순서를 갖는 다면 β를 시퀀스 α의 초시퀀스라고 한다. 그리고 다른 어떤 시퀀스의 원소들 중 순서를 바꾸지 않은 어떤 원소들을 갖는 시퀀스가 있다면 그 시퀀스를 다른 어떤 시퀀스의 부분시퀀스라고 부른다.

시퀀스 데이터베이스, DS는 시퀀스들의 집합이며 $D=\{\sigma_1,\sigma_2,...,\sigma_n\}$와 같다. DS의 시퀀스들의 총 수를 $|D|$로 표현한다. 데이터베이스 D의 시퀀스 σ의 지지도는 σ^+로 표기되며, 이 값은 데이터베이스 D에서 σ를 부분시퀀스로 갖는 시퀀스들의 총 수이다. 빈발부분시퀀스는 어떤 시퀀스의 지지도가 최소 지지도 문턱 값인 min_sup보다 크거나 같은 시퀀스를 말한다.

닫힌시퀀스는 동일한 지지도를 갖는 초시퀀스가 없는 시퀀스를 말한다. 시퀀스 α가 닫힌시퀀스가 되기 위해서는 $\alpha \sqsubset \beta$이면서 두 시퀀스의 지지도가 동일한 시퀀스 β가 없어야 한다. 빈발닫힌시퀀스는 시퀀스 α가 닫힌시퀀스이면서 $\alpha \ge$ min_sup을 만족해야 한다.

응용프로그램의 실행 할 때 읽는 파일들의 집합을 시퀀스로 사용하기 때문에 빈발부분시퀀스가 되기 위한 추가 조건으

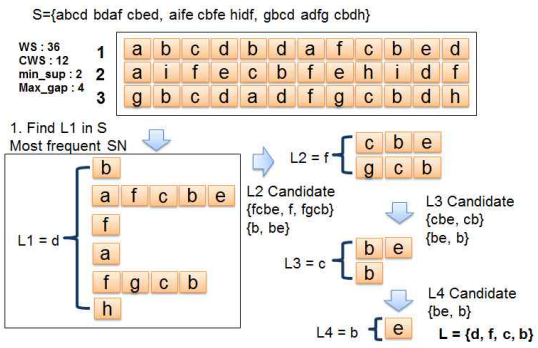

그림 3. 시퀀스 패턴 마이닝의 동작
Fig. 3. Example of Sequence Pattern Mining

로 max_gap이라는 값을 사용한다. 이 값은 현재 선택된 원소와 떨어진 거리를 나타낸다.

그림 3에 보인 것과 같은 S가 있다고 하자. 마이닝 알고리즘에서의 윈도우 크기(WS)는 36이고, max_gap이 4이며 min_sup은 2이라고 가정한다. 윈도우 크기는 빈발부분시퀀스의 집합을 얻기 위해 사용할 원소의 총 개수이다. 흐름자료가 긴 경우 윈도우 크기로 구분된 시퀀스가 만들어 진다. CWS는 윈도우 크기로 나뉜 시퀀스를 일정한 크기로 다시 구분하는 단위이다. 그림에서 사용된 CWS의 크기는 4이고 쉼표로 구분이 되어 있는 S는 다음과 같다.

$$S = \{abcd, bdaf, cbed, aife, cbfe, hidf, gbcd, adfg, cbdh\}$$

먼저 S를 읽어 들인 후 가장 빈번하게 발생된 원소를 찾는다. 이때 선택된 원소는 L1이라고 부르고 L1에 연관되어 최종 빈발부분시퀀스 L에 포함 가능한 다른 원소가 발견되면 순차적으로 L2 그리고 그 이후 n번째를 Ln으로 부른다. L1 이 확정 된 후에는, 찾은 원소를 이용하여 S에서 L2의 대상을 찾는다.

그림 3에서 가장 빈번하게 접근된 값은 "d"이고 이것을 L1으로 사용한다. "d"를 찾는 후에 max_gap 사이에 있는 다른 후보를 검색한다. D에서 "d" 이후에 나타나는 모든 원소를 검색한다. 이때 3번 나타난 "f"가 L2가 된다. 다시 L2를 기준으로 한 D를 검색한다. 이때 2번 발생한 "c"가 L3가 된다. 같은 방식으로 L4를 선택하고 이때 L4는 "b"가 된다. 최종적으로 빈발부분시퀀스 L은 {d, f, c, b} 가 된다.

min_sup은 빈발부분시퀀스 L 에 포함 될 수 있는 원소의 지지도를 나타낸다. L 내의 원소들은 최종적으로 {d-8 회, f-3 회, c-2 회, b-2회}의 지지도를 갖는다. 앞의 예에서 min_sup이 3이라고 하면 지지도가 3인 L2까지만 인정이 되고 지지도가 2인 L3와 L4는 빈발부분시퀀스 L에서 제외가 된다. 최종으로 얻은 L은 결과 파일에 저장이 된다.

2. 탐색거리, 빈도, 빈도 크기 기반 캐싱

시퀀스 σ는 탐색 거리와, 빈도, 크기, 그리고 빈도×크기의 추가 정보를 관리하여 서로 다른 선택 알고리즘의 성능을 비교할 수 있도록 하였다. 추가로 비교하는 마이닝 기법은 다음과 같다: 탐색 길이 기반(DIST), 빈도수 기반(FREQ), 그리고 빈도×크기 기반(F-S) 마이닝 기법.

DIST 기법의 흐름자료의 원소 간의 절대 거리는 선택한 원소와 이전 원소간의 LBA의 차이를 나타낸다. 빈발부분시퀀스의 탐색 길이 σ^D는 각 원소간의 절대 거리의 합으로 정의한다. $i > 1$일 때 i번째 원소의 절대 거리는 $|s_i - s_{i-1}|$로 계산한다.

FREQ 기법의 시퀀스의 빈도는 σ^F로 표기하며 이 값은 빈발부분시퀀스의 모든 원소들의 빈도의 총 합이다. 마지막으로 F-S 기반의 기법은 빈발부분시퀀스의 모든 원소들의 크기와 빈도의 곱의 합으로 정의 된다. 이때 시퀀스의 크기 σ^S는 시퀀스의 추가 정보인 크기 정보를 이용하여 얻은 빈발부분시퀀스의 모든 원소들의 크기의 합으로 정의 된다.

3. 캐싱의 우선순위 선정 방법

HDD는 수TByte의 용량을 지원하기 때문에 캐쉬 디바이스로 사용하는 SSD 간의 용량 차이가 있다. 마이닝을 통해 얻은 모든 캐싱 후보 대상들은 용량제한 때문에 캐쉬에 모두 저장되지 않을 수도 있다. 그렇기 때문에 선택된 캐싱 가능한 데이터의 선별은 중요하다.

제안한 BLOCS 기법이 사용하는 우선순위 선정 방법은 다음과 같다. 그림 4는 본 논문이 제안한 시퀀스 패턴 마이닝 기법을 통하여 얻은 빈발부분시퀀스의 데이터베이스의 예이다. 열 번호와 해당 열의 빈발부분시퀀스 L 에 속해 있는 원소(SN)들의 집합을 길이와 빈도수와 함께 1부터 n까지 표현하고 있다. 흐름자료의 SN과 데이터베이스의 열에서 첫 번째 L1 들을 비교한다. 그리고 비교하는 SN과 L1이 동일하다면 L1을 포함하는 열의 빈발부분시퀀스를 모두 SSD에 등록한다.

{SN, Len, Size}$_1$, {SN, Len, Size}$_2$, ..., {SN, Len, Size}$_n$
1 {34832, 128, 112}, {36696, 128, 16}, ..., {3282, 2, 36}
2 {14898, 18, 32}, {84802, 93, 28}, ..., {78022, 321, 180}
...
n {480208, 1, 257}, {10820, 30, 78}, ..., {923, 11, 99}

그림 4. 빈발부분시퀀스의 데이터베이스
Fig. 4. Frequent Subsequence Database

DIST, FREQ, 그리고 F-S 기반 우선순위 선정 방식은 비교의 기준이 BLOCS와 다르다. DIST 기반 우선순위 선정 기법은 앞 절에서 설명한 것과 같이 시퀀스 내의 원소간의 절대 거리의 합으로 나타낸다. HDD는 느린 탐색지연 시간 때문에 분산된 데이터를 읽을 때 오랜 시간이 걸린다. 탐색거리를 고려하여 멀리 떨어져있는 데이터들을 더 빠른 장치인 SSD에 데이터를 이동해 놓으면 HDD에서 해당 데이터를 읽는 지연시간을 줄일 수 있다. 탐색 지연시간을 줄이기 위한 방법으로 탐색거리가 가장 긴 빈발부분시퀀스 순으로 정렬된 리스트를 관리를 하고 긴 빈발부분시퀀스가 먼저 캐싱 한다.

또 다른 방법으로는 접근 빈도수만을 고려한 FREQ 기반 우선순위 선정방식을 생각해볼 수 있다. 빈번하게 접근되는 데이터가 있다면 SSD에서 해당 데이터를 읽어서 탐색지연시간과 반응속도를 줄일 수 있다. FREQ 기반 우선순위 선정방식은 빈발부분시퀀스 데이터베이스의 모든 빈발부분시퀀스의 빈도수가 큰 순으로 정렬한 리스트를 관리한다. 빈도수가 큰 순으로 먼저 캐싱 한다. F-S 기반은 데이터를 접근하는 횟수와 해당 데이터의 크기를 고려한 우선순위 선정 방식이고 두 곱이 큰 순으로 정렬된 리스트를 관리하고 있다. 빈도×크기가 큰 순으로 먼저 캐싱 한다.

F-S 기반은 데이터를 접근하는 횟수와 해당 데이터의 크기를 고려한 우선순위 선정 방식이고 두 곱이 큰 순으로 정렬된 리스트를 관리하고 있다. 빈도×크기가 큰 순으로 먼저 캐싱 한다.

IV. 캐쉬 시뮬레이터

구현한 캐싱 알고리즘의 평가를 위해서 캐쉬 시뮬레이터를 개발하였다. 구현된 캐쉬 시뮬레이터의 목적은 임의의 흐름자료를 입력받아 다양한 캐쉬 알고리즘을 적용하였을 때 성능 비교뿐만 아니라 각 장치에 기록된 데이터의 양과 적중률과 같은 통계치를 얻는 것이다. 수집된 통계 값을 통하여 다양한 캐쉬 정책을 평가하고 성능을 비교할 수 있는 환경을 만든다.

이 장에서는 캐쉬 시뮬레이터의 구성과 시뮬레이터를 구성하는 모듈을 설명한다. 그리고 입력과 출력 형식을 설명한다. 시스템 디스크와 캐쉬 시뮬레이터가 사용하는 HDD를 구분하였고 SSD를 캐쉬 시뮬레이터가 사용할 수 있도록 하였다. 개발은 윈도우즈에서 Visual Studio 2008 에서 C 언어를 사용하여 개발하였다. 캐쉬 시뮬레이터의 구성은 크게 어드레스 체커 모듈, 디바이스 관리 모듈, 통계 모듈로 되어 있다.

그림 5는 캐쉬 시뮬레이터의 구조를 보인다. 캐쉬 시뮬레이터에는 두 개의 입력 파일이 있다 흐름자료 파일과 마이닝

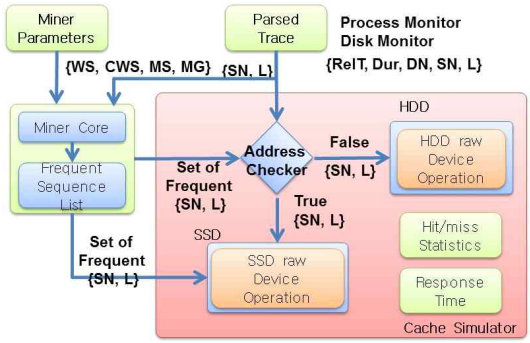

그림 5. 캐쉬 시뮬레이터의 구조
Fig. 5. Structure of Cache Simulator

표 1. 캐쉬 시뮬레이터의 출력 파일의 형식
Table 1. Output Format of Cache Simulator

Value	Description
Index	Sequence Number
Hit Ratio (Count)	Number of SN count divided by total number of read I/O of workloads
Hit Volume (Volume)	Total volume of SN divided by total read I/O volume of workloads
Load Size	Write volume on SSD
Update size	Update volume on SSD
Total Write Volume	Total write volume to HDD and SSD
Average Response Time	Average response time of HDD and SSD using cache algorithms
Average Response Time of SSD	Average response time of SSD using cache algorithms
Average Response Time of HDD	Average response time of HDD using cache algorithms
Load Count	Count of write I/Os to SSD
Read Count	Total number of read I/Os in traces
Total I/O Count	Total number of I/Os in traces
Eviction Count	Number of eviction routine calls
Number of Evicted Sequences	Number of sequences evicted from SSD
Evicted Volume	Total volume evicted from SSD

프로그램에서 생성된 캐쉬 흐름자료 파일이다. 이 두 개의 파일을 캐쉬 시뮬레이터에 입력하면 어드레스 체커 모듈이 동작한다. 이 모듈은 입력된 흐름자료의 섹터의 캐싱 여부와 데이터가 기록된 장치와 위치에 대한 정보를 토대로 해당 장치에서 읽기 또는 쓰기 요청을 처리한다.

어드레스 체커는 입력 흐름자료와 마이닝 알고리즘에서 얻은 빈번하게 접근된 Sector Number(SN)의 집합인 빈발부분시퀀스 L을 파일로 받는다. 어드레스 체커는 입력 흐름자료를 이용하여 얻은 빈발부분시퀀스 L의 데이터베이스를 갖고 있는 파일의 첫 번째 섹터들과 비교하여 SSD에 업데이트를

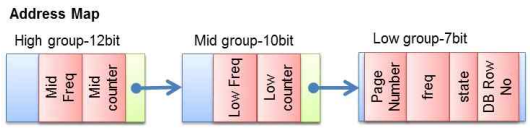

그림 6. Cache Loader가 사용하는 주소 관리 자료 구조
Fig. 6. Data Structure for Address Management in Cache Loader

할 시퀀스들을 선정한다. 입력 받는 흐름자료에서 캐쉬에 등록이 되어 있는 데이터와 일치하는지를 비교하여 SSD에 있는 경우는 SSD의 데이터를 읽어오고, HDD에 있는 경우는 HDD에서 읽어 온다. 요청된 SN이 SSD에 없다면 HDD로 요청을 보내어 처리를 하고, SSD에 있다면 어드레스 체커의 자료구조에서 해당 SN의 접근 횟수를 업데이트한다.

캐쉬 시뮬레이터의 결과로 4개의 파일이 생성이 된다. 생성되는 파일은 요약 정보를 제공하는 파일과 HDD와 SSD의 응답시간을 기록한 두 개의 시계열 파일 그리고 교체 정책으로 삭제된 섹터크기 파일이다. 시계열의 단위는 msec으로 표현된다. 요약정보를 제공하는 파일의 출력 값은 표 1 에 나타나 있다. 캐쉬에 있는 SN의 용량과 개수로 본 캐쉬 적중률과 각 장치의 평균 응답 시간 그리고 낸드에 기록된 Byte단위의 크기와 흐름자료 파일의 읽기 I/O 의 개수와 전체 I/O 의 개수 등의 정보를 기록한다. 또 다른 파일에는 읽기와 쓰기 요청의 응답시간이 기록된다.

1. 어드레스 체커 모듈

어드레스 체커는 입력된 흐름자료와 마이닝 결과 그리고 캐쉬 된 데이터를 서로 비교하는 동작을 한다. 구성 요소로는 Cache Loader 와 Data Mover가 있다. Cache Loader는 HDD의 데이터를 SSD로 이동할 때 SSD page number를 관리하는 사상 테이블을 갖고 있다. Data mover 는 사상된 SSD의 page에 HDD의 데이터를 복사를 관장한다.

Cache Loader에서 사상을 관리하기 위해 사용하는 구조체는 그림 6에 나타나 있다. 이 구조체는 HDD와 SSD 간에 저장하는 최소 단위가 다르기 때문에 주소 변환을 하여 매핑 정보를 관리 한다. 입력 받은 SN은 세 부분으로 나뉜다. SN의 크기는 2TB를 표현할 수 있는 32 bit 이지만, 이 구조체의 합은 총 29 bit 이다. 상위 12 bit는 high 그룹으로 가운데 10bit 는 mid 그룹, 그리고 하위 3 bit를 제외한 7 bit 는 log 그룹으로 지정한다. 하위 3bit를 제외한 이유는 플래시 메모리의 단위가 크기가 4kbyte이고 한 섹터가 512byte 라고 하면 8개 섹터가 1개의 page에 사상되기 때문이다. High와 mid 그룹은 해당 노드에 연결된 하위 노드가 몇 개가 있는지 그리고 몇 번 접근 되었는지에 대한 빈도수를 counter와 Freq에 기록을 한다. 그리고 하위 노드로 향하는 포인터로 이루어져있다. 그리고 Low 그룹은 SN에 대한 page number를 기록 하고 그 SN의 접근 횟수와 상태 값을 저장한다. 상태 값은 Empty, Valid, 그리고 Dirty 로 3가지이다. 그리고 데이터베이스의 열 번호에는 마이닝으로 얻은 빈발부분시퀀스 L의 데이터베이스의 등재 번호를 기록한다. 이 값은 교체 정책에서 참조를 한다.

이 구조체는 모두 int 형으로 선언이 되어 있다. 이 구조체의 최대 크기는 총 49.5 MB이지만 사상이 가능한 영역은 SSD의 크기에 제한을 받기 때문에 실상은 매우 작다. 초기화 과정에서 High 그룹만 생성이 되며 이 때 48KB를 크기를 갖는다. Mid 그룹은 생성이 되면 12KB의 크기를 갖고 Low는 15 KB 의 크기를 갖는다.

2. 장치 관리 모듈

장치 관리 모듈의 목적은 크게 세 가지이다. 먼저, 장치를 열어 핸들을 얻은 후 읽기 또는 쓰기 동작을 처리하는 것이고, 둘째는 캐쉬 해야 할 데이터를 HDD에서 SSD로 복사하는 것이다. 마지막으로 SSD에 저장되지 않은 데이터는 HDD로 요청한다.

캐쉬 시뮬레이터에서 HDD와 SSD의 동작은 실제 장치를 열어서 Read/Write IO를 요청하고 그 소요된 시간을 측정한다. 실제 장치가 I/O를 처리하는데 사용한 시간을 측정하기 위해 먼저 raw device를 열어 장치에 대한 핸들을 얻는다. 입출력 요청을 받으면 섹터 또는 page에 맞게 변환된 주소를 이용하여 해당 저장 장치에서 읽기 또는 쓰기 작업을 한다.

3. 통계 모듈

캐쉬 시뮬레이터의 성능을 평가하기 위해서 사용되는 주요 측정 방법은 Hit Ratio, 응답시간과 캐싱 용량과 같은 정보이다. 통계 모듈은 어드레스 체커 모듈과 장치 관리 모듈과 유기적으로 동작을 한다.

캐쉬의 적중률은 전체 Read IO 와 SSD에서 처리된 Read IO의 비로 계산하는 Eq. (1)을 통하여 얻는다. 이때 $READ^{ALL}$은 총 Read IO의 수이고 $READ^{SSD}$는 SSD에서 처리된 Read IO 의 개수 이다. 캐쉬 적중률은 최종 결과 파일에 기록이 된다.

$$HitRatio = \frac{\sum READ^{SSD}}{\sum READ^{ALL}} \quad (1)$$

캐쉬 시뮬레이터에서 SSD와 HDD에 읽기 또는 쓰기 작업이 시작하기 직전과 직후의 시간 값을 얻어와 IO 작업의 응답 시간을 측정한다. 각 장치의 응답 시간 측정은 Eq. (2)와 Eq. (3)을 통하여 구한다.

$$Resp_{HDD} = \frac{\sum \tau_i^{HDD}}{\sum IO^{HDD}} \quad (2)$$

$$Resp_{SSD} = \frac{\sum \tau_i^{SSD}}{\sum IO^{SSD}} \quad (3)$$

τ_i^{HDD}와 τ_i^{SSD}는 i번째 IO를 처리하는데 HDD와 SSD에서 소요된 시간을 나타내고 각 소요 IO 시간을 각 디바이스의 총 IO 개수의 비로 $Resp_{HDD}$와 $Resp_{SSD}$를 얻는다.

캐쉬를 사용하여 얻는 이득 $Resp_{cache}$는 Eq. (4)로 구한다. HDD와 SSD의 평균 응답시간을 총 IO 중 HDD와 SSD가 차지하는 비율의 곱의 합으로 얻는다.

$$Resp_{cache} = Resp_{HDD}\frac{\sum READ^{HDD}}{\sum READ^{ALL}} + Resp_{SSD}\frac{\sum READ^{SSD}}{\sum READ^{ALL}} \quad (4)$$

4. 교체정책

SSD의 저장 공간이 가득 차면, 어떤 불필요한 데이터를 SSD에서 내릴 것인지를 정하는 것은 중요하다. 캐쉬에 올라온 데이터는 모두 자주 사용되는 데이터들이라고 하더라도 빈도는 차이가 있을 수 있다. Cache Loader에서 사용된 구조체에는 이를 위하여 섹터의 접근 빈도를 기록하고 있다. 교체 정책이 실행 시점과 끝나는 시점을 다음과 같이 정의한다. 교체 동작하는 시점은 SSD의 사용률이 High watermark에 도착 하였을 때이고 이 값에 도달하면 High group의 Freq 값을 순차적으로 정렬을 한다. 이 때 사용하는 사용률은 Eq. (5)에 나타나 있다.

$$U_i = \frac{Freq(Sector)_i^{High}}{Sector_{total}} \quad (5)$$

전체 섹터와 High 그룹의 i 번째 노드의 Freq 값의 비를 구하는 식에서 U_i가 낮은 순으로 지우기 시작한다. High 노드의 값이 모두 동일한 경우 Mid 그룹을 비교하여 가장 낮은 순으로 지운다. Eviction이 종료되는 시점은 총 사용률이

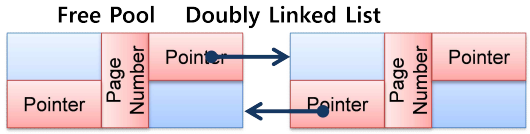

그림 7. FREE POOL을 관리하는 자료 구조
Fig. 7. Data Structure to Manage Free Pool

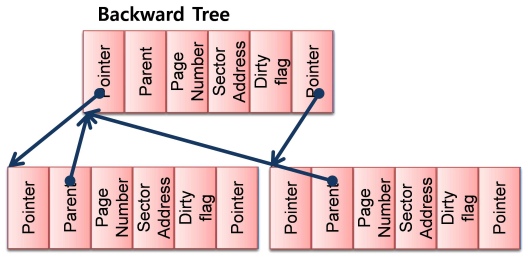

그림 8. Backward Balance Tree 의 자료 구조
Fig. 8. Data Structure of Backward Balance Tree

Low watermark까지 도달할 때이다. 만약 SSD에 새로운 값이 기록이 되고 HDD에는 오래된 값이 있는 경우는 SSD의 값을 HDD로 갱신을 한다.

교체 정책의 성능과 관리를 위해서 두 개의 자료 구조를 사용한다. 두 개의 자료 구조는 Free pool 과 Backward balanced tree이다. 각 자료 구조는 그림 7과 그림 8에 나타내었다. Free pool 의 목적은 SSD에 남아 있는 사용 가능한 페이지를 관리하는 것이다. 쓰고 지우기가 빈번한 Free pool의 관리를 위해서 doubly linked list를 사용하였다 구조체의 각 필드는 모두 int 형이다.

해당 페이지가 HDD의 어디에 기록이 되었는지를 빠르게 검색하기 위해서 Balanced Tree인 Red Black Tree를 사용하였다. 이 Tree를 사용하여 페이지 주소와 섹터 주소 그리고 이 페이지가 업데이트 여부와 같은 정보를 관리하도록 하였다. Dirty flag를 제외한 모든 필드는 int 형의 자료이고 dirty flag는 unsigned char형을 갖고 있다.

HDD와 SSD 간의 주소의 매핑을 관리하는 구조체 중 High group은 교체 동작하는 시점을 정하며 Low group에 있는 데이터베이스의 열 번호 필드는 교체 과정에서 연관성을 유지할 수 있도록 한다. 마이닝 된 결과의 데이터베이스에는 빈발부분시퀀스 L 의 엔트리 번호가 기록되어 있다. 엔트리 번호는 빈발부분시퀀스 L 이 SSD에 삽입될 때 Low group의 데이터베이스의 열 번호에 기록 된다. U_i 가 Highmark를 넘어가면 High group 에서 Freq 값이 가장 작은 노드를 찾는다. 이 노드에 연결되어 있는 모든 SN은 지울 대상이 된다. 캐쉬 적중률을 반영하는 Freq 값이 가장 낮다는 것은 그

만큼 캐쉬에서 중요도가 낮다는 뜻과 동일하기 때문이다. 연관성을 유지하면서 지우기 위해서 SN이 갖고 있는 데이터베이스의 열 번호를 참조해야 한다. 열 번호에 기록된 번호를 사용하여 마이닝된 데이터베이스에서 빈발부분시퀀스 L 을 얻는다. 이 빈발부분시퀀스 L 은 Low group의 SN이 SSD에 등록될 때 같이 등록이 되었던 SN의 집합이므로 같이 찾아 지운다. 이와 같은 방법을 사용하는 것의 장점은 등록을 할 때 사용한 연관성이 유지가 되지만 단점은 검색이 빈번하기 때문에 교체 비용이 커진다.

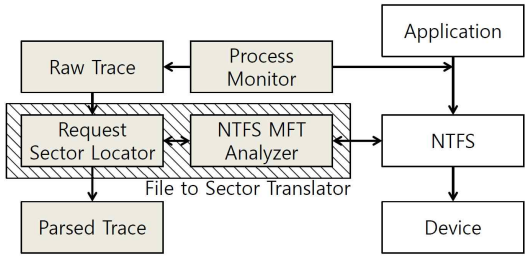

그림 9. 흐름자료 분석기의 구조도
Fig. 9. Structure of Trace Parser

V. 워크로드 분석

이 장에서는 BLOCS 알고리즘의 성능을 평가하기 위해 캐쉬 시뮬레이터에서 사용할 사용자 스토리지 이용 패턴을 분석한다. 흐름자료를 얻기 위해서 사용한 프로그램은 Diskmon[20]이다. 시뮬레이터는 Diskmon으로 수집한 흐름자료 파일을 그대로 사용할 수 없으므로 별도의 프로그램을 구현하였다. 구현한 프로그램은 Diskmon으로 수집한 데이터의 필드의 형을 변환한다. 수집한 흐름자료의 종류와 수집 방법을 설명하고 흐름자료의 특성을 살펴본다.

1. 흐름자료 분석기

Diskmon[20]을 이용하여 얻은 흐름자료를 캐쉬 시뮬레이터에 바로 입력하여 사용할 수가 없다. 그 이유는 흐름자료에서 파일시스템이 요청하는 LBA 정보는 얻을 수 있지만 파일의 실제 물리 주소는 파악 할 수가 없기 때문이다. 이 물리주소를 얻기 위해서 NTFSInfo(NTFS File Sector Information Utility)[21]라는 도구를 사용 할 수 있다.

NTFSInfo를 사용하여 흐름자료를 분석하여 실제 물리주소를 얻을 때 생기는 문제는 세 가지가 있다. NTFSInfo를 사용하면 주소변환에 필요한 시간이 매우 길다. 한 파일의 섹터를 얻어오는데 평균 25msec의 시간이 걸리는데 편차가 심하여 많게는 1.5초까지 소요되는 경우가 있다. 둘째로 Unicode 문자셋은 인식을 못하여 오류 값을 반환하고, 마지막으로 NTFS resident 파일들의 주소를 제대로 반환하지 못한다.

윈도우즈 시스템의 NTFS에는 resident 파일이라는 특별한 형태의 파일이다. 모든 파일들은 MFT(Master File Table)에 등재된 값들과 연관 되어 있다. MFT의 한 엔트리의 크기는 1024 byte를 가지며 저장 장치에 저장된다. 각 엔트리는 파일의 이름, 변경일자, 접근 권한과 같은 파일의 속성 값들을 갖고 있다. MFT의 하나의 엔트리에 저장되는 정보가 1024 byte보다 작은 경우 NTFS는 데이터를 MFT의 엔트리에서 속성 정보를 기록하고 남은 부분에 기록한다. 이렇게 하면 MFT 엔트리만 저장을 하고 추가적인 파일을 할당하지 않기 때문에 스토리지 용량을 절약할 수 있다. 이렇게 저장장치에 별도의 위치에 기록되지 않고 MFT 내에 저장되는 파일을 resident 파일이라 부른다. NTFSInfo로는 Resident 파일의 물리적 위치의 값을 변환해 낼 수가 없다.

Resident 파일의 물리주소를 얻고 Unicode 문자셋 인식을 하고 균일한 섹터주소 획득 시간을 위해 NTFS의 MFT의 정보를 직접 읽는 방법을 선택하였다.

그림 9에 흐름자료 분석기의 구조도를 나타내었다. 가공되지 않은 흐름자료가 분석기에 입력이 되면 File to Sector Translator가 섹터주소를 변환하기 위해서 파일의 경로를 NTFS MFT analyzer에게 전달한다. 전달 받은 경로를 MFT에서 읽어내어 섹터 번호를 얻어 온다. 읽어 온 섹터 주소는 request sector finder에게 전달이 되어 해당 IO 의 offset 값과 합산이 되어 최종 출력 파일에 저장이 된다.

흐름자료 분석기가 입력으로 사용하는 파일은 Diskmon으로 수집한 정보가 포함된다. 입력 파일의 정보는 수집하기 시작한 시간부터의 모든 I/O에 대해 상대시간과 I/O가 처리되는데 걸린 총 시간, 호출한 프로세스의 이름, 읽기/쓰기의 분류가 기록되어 있다. 그리고 실행파일의 경로 정보와 읽기 또는 쓰기를 한 파일에 대한 오프셋과 길이 등의 정보도 제공한다.

흐름자료 분석기는 MFT 정보를 이용하여 섹터 주소를 변환한 후에 캐쉬 시뮬레이터에서 사용할 수 있는 파일을 생성한다. 흐름자료 분석기가 생성하는 최종 파일의 포맷은 순서대로 상대시간, 실행파일의 경로, 바이트 단위로 기록되는 원 파일에 대한 오프셋과 길이, LBA 값, 섹터와 오프셋을 합한 값, 그리고 읽기/쓰기의 분류에 대한 정보이다.

NTFSInfo는 다른 응용 프로그램에서 호출해서 사용할 수

표 2. 흐름자료 분석기의 성능
Table 2. Performance of trace parser

Item	Photoshop	Picasa
Volume of trace	2MB	18MB
I/O Count	13,300	109,000
Time in NTFSInfo	389.04Sec	1934.7Sec
Time in Trace parser	2.55Sec	11.15Sec
% of Improvement	99.3%	99.4%

있는 API를 제공하지 않는다. NTFSInfo가 제공하는 기능은 파일의 경로 정보를 입력 받아 섹터 주소로 변환하여 파일의 시작과 끝 주소 값을 십진수와 16진수로 표현한다. 표 2는 개발된 흐름자료 분석기의 성능이다. NTFS 의 MFT 정보를 직접 변환하기 때문에 NTFSInfo에 비해서 변환 시간을 99% 이상 줄일 수가 있다.

2. 워크로드 시나리오

표 3 은 사용자 스토리지 이용 패턴의 시나리오를 나타낸다. 총 11개의 응용 프로그램 흐름자료를 수집하였다. 응용프로그램의 시작과 종료 사이에는 다른 작업은 없었다. User Data를 생성하기 위해서 사용한 프로그램은 Diskmon이다. 이 프로그램 외에도 Process Monitor라는 프로그램이 있지만 Process Monitor를 사용하지 못한 이유는 File I/O 이외에도 Process, Network, Registry 등의 사용 정보를 같이 누적하기 때문에 파일 크기의 증가량이 너무 크기 때문이다. 1분당 약 400MB정도의 데이터가 누적이 되고 Filtering 하여 저장한다고 하여도 1분당 10메가 분량으로 데이터 증가하여 사용할 수가 없었다.

표 4는 사용한 워크로드에 포함된 전체 명령의 개수와 유일한 파일의 개수, 그리고 읽기/쓰기의 명령 개수와 그 용량을 나타내었다. Bootup을 위해서 722번의 I/O 관련 동작이 있었고 읽기가 쓰기보다 약 17배 더 많이 발생하였다. 엑셀의 경우 응용프로그램이 실행 후 1분간의 유휴 시간 동안 279MB가 디스크에서 읽혔고 142MB가 기록되었다. 오피스 관련 응용프로그램들만으로 읽기와 쓰기 용량의 평균을 구해보면 읽기는 141MB 그리고 쓰기는 약 37MB이지만 아웃룩을 제외하면 쓰기용량은 2MB 이하였다.

그림 10는 워크로드별 읽기 I/O의 위치 분포를 나타낸다. 표 5는 워크로드 별 읽기 I/O의 요청 크기 분포를 나타내었다. 쓰기 I/O 요청의 크기 분포는 표 4와 표 5의 정보만 소개하고 그래프는 지면 관계상 생략하였다. 오피스 프로그램(Excel, Outlook, Powerpoint, Word)의 제3사분위수의 평균 요청크기를 보면 읽기가 384Byte, 쓰기가 556Byte이

표 4. 워크로드에서 사용한 파일의 정보
Table 4. INFORMATION OF ACCESSED FILES ON WORKLOAD

Name	Total I/O Count	Unique File Count	Read I/O Count	Read Size	Write I/O Count	Write Size
Unit	x1000	-	x1000	MB	-	MB
Bootup	722	3870	691	670	22551	40
Excel	140	1112	125	279	14556	142
iTunes Cat.	160	840	156	677	2403	21
iTunes play	86	656	83	98	1756	13
KMplayer	38	400	34	53	2387	5
Outlook	69	682	67	122	1365	2
Photoshop	13	123	11	65	1690	1
Picasa Cat.	109	3452	105	649	2574	32
Picasa	58	1641	52	310	5009	15
Powerpoint	25	1082	23	92	995	1
Word	58	623	55	73	2452	2

표 3. 워크로드 시나리오
Table 3. Workload Scenario

#	Workload	Scenario
1	Boot-up	Win7 32bit Home Premium Cold Boot; Log-in and terminate after 3 minutes
2	Office 2010	Word, Powerpoint, Excel; After launching the application stay idle for 1minute, and terminate
3	Outlook 2010	Launch the application and click "Mail Send/Receive" button; Stay idle for 1 minute, and terminate (Mail DB Size: 6.5GB)
4	Photoshop CS5	Launch the application; Stay idle for 1 minute, and terminate
5	KMplayer	Double click the Movie file; Play the movie for 2 minutes and terminate (MPEG-4 H.263, 23.976 fps, video 1019Kbps , audio 192Kbps)
6	Picasa3 Catalogue	Add picture files to the application, and terminate (Total volume: 8.7GB, File count: 2947, Directories count: 37)
7	Picasa3	Slide five 20MB sized PSD files, and terminate
8	iTunes Catalogue	Add music files to the application, and terminate (Total volume: 6.28GB, File count: 384, Directory count: 35)
9	iTunes Play	Play music file for 2 minutes, and terminate (File size: 5.86MB, Length: 4Min 14sec, 192Kbps)

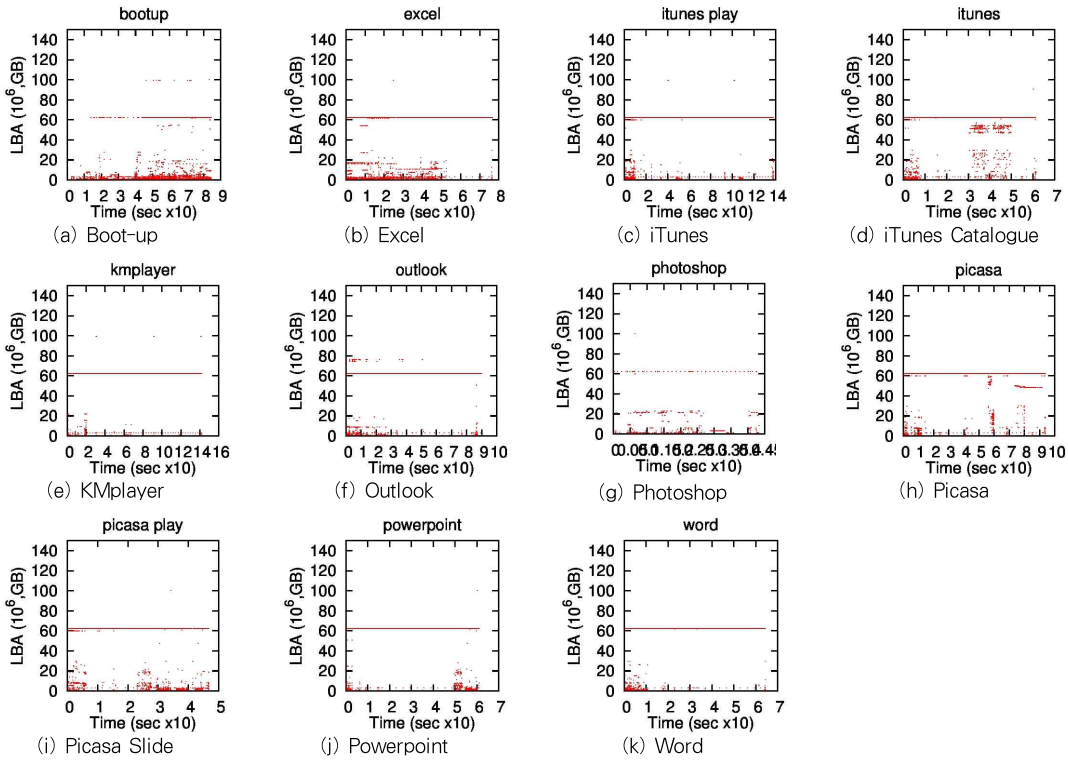

그림 10. 워크로드 별 읽기 I/O 분포
Fig. 10. Read I/O Distribution of Workload

표 5. 워크로드별 읽기/쓰기 I/O 요청 크기 분포
Table 5. Read/Write I/O Size Statistics of Workload

Quantile	Min		1st Quantile		Median		Mean		3rd Quantile		Max	
Unit	Byte		Byte		Byte		Byte		Byte		KByte	
I/O	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write
Bootup	0	0	0	11	48	20	1015	738	256	38	1932	512
Excel	1	1	2	2048	16	2048	2339	5120	256	2048	6777	578
iTunes Cat.	0	1	40	2	256	15	4524	4516	2048	512	728	512
iTunes	0	1	8	4	170	4	1233	3677	512	20	728	398
KMplayer	0	2	64	4	170	4	1604	1016	512	44	3372	256
Outlook	1	1	8	4	170	4	1902	550	512	36	1024	161
Photoshop	1	1	1	1	1	1	12	1	8	2	1094	108
Picasa Cat.	0	2	64	4	512	4	6458	6485	16384	512	273	530
Picasa Slide	0	2	54	25	512	39	6140	1479	5120	119	324	520
Powerpoint	0	2	8	4	170	4	4086	373	512	43	994	75
Word	0	1	144	4	256	61	1369	409	256	97	275	30

다. 평균 읽기/쓰기 요청의 크기를 보면 Excel과 다른 오피스 프로그램들(Powerpoint, Word, Outlook)은 서로 다른 패턴을 보이고 있다. Excel은 읽기 요청의 크기가 평균 쓰기 요청의 약 50%의 크기를 갖고 있다. 반면 아웃룩, 파워포인트 워드는 읽기 요청 크기가 각각 쓰기 요청 크기의 3.5배, 11배, 그리고 3.3배 더 크게 나타났다.

사진, 음악 파일과 동영상을 재생하는 워크로드(iTunes, KMplayer, Picasa Slide)의 경우 미디어 파일을 갱신을 하거나 삭제하는 동작은 하지 않는다. 저장 장치에 기록되어 있는 정보를 읽어 오기 때문에 쓰기 요청의 크기가 작을 것이라

표 6. 실험환경
Table 6. Environment

Device	Model
CPU	AMD Phenom(tm) II X4 925 Processor 2.8 GHz
RAM	DDR3 1333MHz 4G
M/B	Gigabyte Z68X-UD3H-B3
HDD	WD (500GB, SATA2, 7200 RPM, 16M)
O/S	Windows7 32bit Ultimate Eng
Compiler	Visual Studio 2008

표 7. BLOCS 알고리즘의 변수
Table 7. Parameters for BLOCS Algorithm

Parameter	Values
WS	2000, 4000, 6000, 8000, 10000
CWS	100
MS	2,4, 8, 16
MG	30, 40

생각되지만 그렇지 않은 것을 알 수 있다. iTunes 의 경우 평균 677MB를 읽었고 21MB를 기록하였다. iTunes의 경우 기본 설정 음악 라이브러리를 구성할 때 응용 프로그램이 관리하는 새로운 디렉토리에 파일들을 저장을 기록 하도록 되어 있지만 실험에 사용한 구성에서는 원본 데이터는 복사하지 않도록 하였다. Picasa는 약 649MB를 읽었고, 32MB를 기록하였다. 동영상을 재생하는 KMplayer의 경우 53MB를 읽고 약 5MB를 기록하였다.

VI. 실험 결과

최종적으로 구현된 캐쉬 시뮬레이터의 성능을 평가하기 위하여 사용된 환경은 표 6에 나타나 있다. 캐쉬 시뮬레이터는 영문 윈도우7 Ultimate 32 bit 환경에서 C 언어를 이용하여 개발하였고 컴파일러는 Visual Studio 2008을 사용하였다. 먼저 기초 실험을 통하여 마이닝 변수로 사용된 Window Size, Cutting Size, min_sup, 그리고 max_gap 의 영향을 평가한다. 먼저 기초 실험을 통하여 최적 값을 얻고, 실험을 통해 찾은 최적 구성을 이용하여 캐쉬/교체 기법에 따른 성능 차이와 사용하는 캐쉬 용량의 크기 변화에 따른 성능차이를 비교한다. 그리고 마이닝에서 소요되는 시간을 비교하여 실제 시스템에 적용 가능성을 판단한다. 끝으로 마이닝을 적용한 시스템에서 SSD의 수명은 어떤 영향을 받는지 평가한다.

1. BLOCS 알고리즘의 변수 영향

BLOCS 알고리즘에서 성능에 영향을 주는 변수는 크게 4가지가 있다. Window Size, Cutting Window size, min_sup, 그리고 max_gap 이다. 각각을 WS, CWS, MS, 그리고 MG 로 표기 한다. 각 변수의 설명은 BLOCS 캐쉬 알고리즘을 다루면서 하였다. 표 7에는 실험에 사용한 변수의 조합을 나타내었다.

지면 관계상 결과를 도식하지는 않았지만, 표 7의 변수 조합을 사용하여 캐쉬 적중률을 비교한 결과 WS와 CWS는 캐쉬 적중률에 큰 영향을 주지 않는 것으로 나타났다. 초기 실험을 통해서 확인한 결과 CWS는 실험에 큰 영향을 주는 변수는 아니기 때문에 100으로 고정하였다. 가장 영향을 많이 준 변수는 MS와 MG 이다. MS는 작을수록 성능이 좋게 나타나며 MG는 클수록 성능이 좋게 나타나는 것을 확인하였다.

MS와 MG는 빈발부분시퀀스에 포함여부를 결정하는 중요한 변수이다. MS가 작으면 빈발부분시퀀스로 인정되는 원소의 개수가 늘어난다. 제안한 기법은 선정된 빈발부분시퀀스의 모든 원소를 하나의 단위로 캐쉬에 적재하기 때문에 실제 프로그램이 캐쉬에서 읽을 수 있는 원소들이 많아져서 적중률이 높아지는 효과가 있다. MG도 마찬가지이다. 하나의 빈발부분시퀀스로 인정되는 원소들 간의 최대거리를 뜻하기 때문에 이 값이 커지면 원소들 간의 출현 순서에 대한 연관성 기준이 느슨해지는 효과가 있다. 그렇기 때문에 MG만큼 멀리 떨어진 원소가 동일한 빈발부분시퀀스에 포함된다.

가장 성능이 좋았던 조합은 WS에 상관없이 CWS는 100, MS는 2, MG는 40이다. 이후에 실험에서 WS는 2000 그리고 CWS는 100, MS 는 2, MG는 40을 사용하도록 한다. SSD의 크기는 512MB로 설정하여 실험한다.

2. 캐쉬 우선순위 선정 방법에 따른 성능

BLOCS 알고리즘이 선별한 빈발부분시퀀스를 한정된 캐쉬 저장소인 SSD에 이동하는 4가지 방법으로 시퀀스 패턴 마이닝 기반의 캐쉬 우선순위 (BLOCS), 거리우선 기반(DIST), 빈도수 우선 기반(FREQ), 그리고 빈도수와 파일 크기의 곱 기반(F-S)을 사용하였다. 사용한 변수의 조합은 앞서 정한한 것과 같이 WS가 2000, CWS가 100, MS는 2, MG는 40, 그리고 캐쉬 용량은 512MB이다.

그림 11는 워크로드 별 캐쉬 우선순위 선정 방법에 따른 캐쉬 성능을 응답시간과 적중률으로 나타내었다. 통계모듈에서 설명한 Eq. (1)과 Eq. (4)를 사용하여 각 캐쉬 우선순위 선정 방법의 적중률과 평균 응답시간을 구했다. 평균 응답시간을 구하기 위해 필요한 각 장치에서 읽고 쓰기 시간은 파일 시스템을 통과하지 않는 raw 디스크를 접하는 시간을 측정하여 계산하였다.

캐쉬 우선순위 선정 방법에 따른 응답시간과 적중률 모두 워크로드에 상관없이 BLOCS 방식이 가장 좋았다. 가장 안

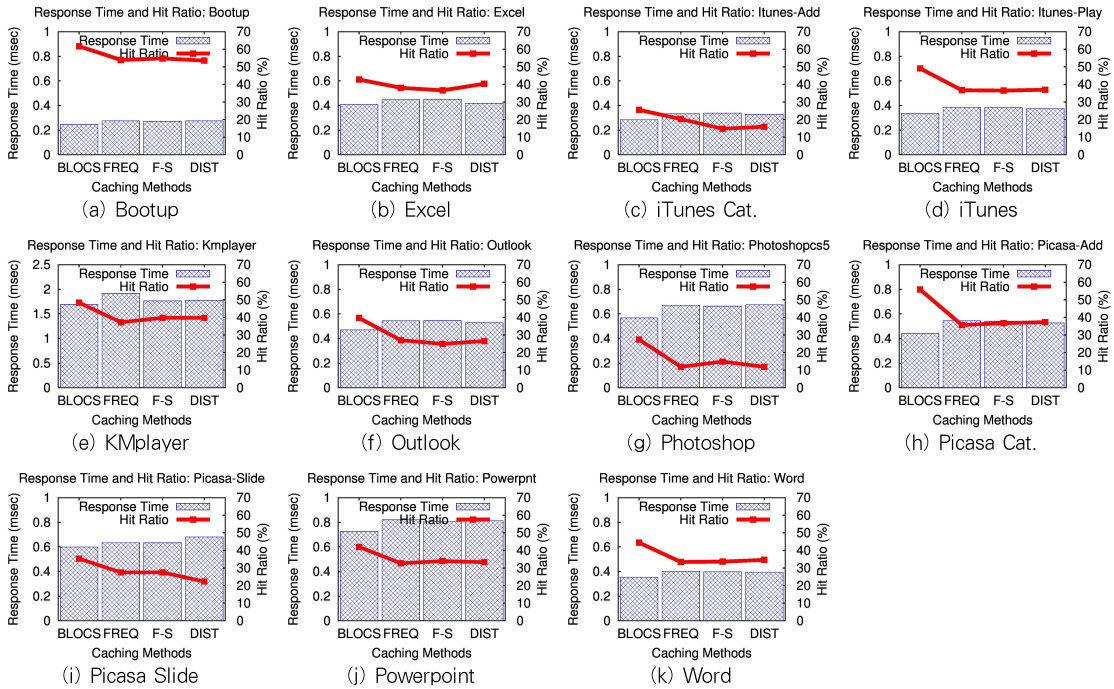

그림 11. 캐싱 우선순위 방법에 따른 성능(WS: 2000, CWS: 100, MS: 2, MG:40)
Fig. 11. Performance of Caching Schemes, (WS:2000, CWS: 100, MS: 2, MG:40)

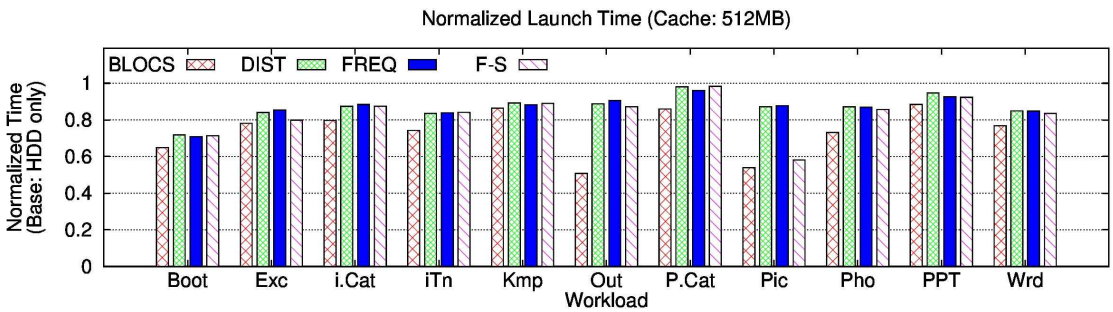

그림 12. 캐싱 우선순위 방법에 따른 정규화된 실행시간(WS: 2000, CWS: 100, MS: 2, MG:40)
Fig. 12. Normalized Execution Time of each Schemes (WS: 2000, CWS: 100, MS: 2, MG:40)

좋은 우선순위 선정 방식은 워크로드에 따라 달랐는데 적중률은 FREQ 방식이 KMplayer, photoshop, Picasa Cat. 그리고 word에서 가장 안 좋았다. F-S 방식이 excel, iTunes Cat. iTunes, 그리고 outlook에서 가장 안 좋은 적중률을 보였으며 DIST방식은 Bootup과 Picasa에서 가장 안 좋은 성능을 보였다. 대체적으로 적중률이 안 좋은 경우가 평균 응답시간도 낮은 것으로 나타났다.

BLOCS기법을 사용하였을 때 61%로 가장 높은 적중률을 보인 Bootup 워크로드의 적중률은 가장 안 좋았던 DIST 방식에 비해 15% 더 높은 적중률을 보였다. iTunes에 파일을 등록하는 워크로드에서는 F-S보다 72%더 좋은 적중률을 보였으며, 사진을 등록하는 Picasa 워크로드는 FREQ 방식 보다 55% 더 좋은 적중률을 보였다. outlook 워크로드는 F-S 방식보다 59% 더 좋은 성능을 보였다. 워크로드의 적중률을 평균적으로 보면 BLOCS 기법이 각 워크로드에서 가장 낮은 적중률을 보인 기법들의 평균에 비해 50% 더 좋은 적중률을 보였다.

그림 12에는 캐싱 우선순위 선정 방식에 따른 정규화된 실

행 시간을 나타내었다. HDD만을 사용하였을 때 워크로드의 전체 수행 시간을 1로 두고 각 기법을 적용하였을 때 소요된 시간을 표현하였다. BLOCS 방식을 사용하였을 때 Bootup, Outlook 그리고 Picasa Slide 워크로드의 경우 실행시간을 35%, 50%, 그리고 47%를 줄일 수가 있었다. DIST 기법을 사용할 때 Picasa 워크로드에서 실행시간을 42% 줄일 수 있었던 것을 제외하면 나머지 워크로드들에서는 큰 성능 향상을 볼 수 없었다.

전체적으로 보면 BLOCS기법을 사용하였을 때 평균 30% 정도의 시간을 줄일 수가 있었다. FREQ와 F-S, 그리고 DIST 기법은 각각 평균 13%, 13%, 그리고 16%의 실행 시간을 줄일 수 있었다. BLOCS 기법이 다른 기법에 비해 약 2배 이상 더 좋은 성능을 보이고 있다.

3. SSD 크기 변화 따른 성능

앞선 실험에서는 512MB의 크기를 갖는 SSD를 캐쉬로 사용하여 캐쉬 우선순위 선정 방식에 대한 성능을 응답시간, 적중률, 총 수행 시간으로 나누어 평가 하였다. 캐쉬의 크기는 가격과 연관이 있기 때문에 비용 문제를 고려하였을 때는 적은 용량의 캐쉬를 효율적으로 사용할 수 있는 기법을 찾는 것이 중요하다. 캐쉬 크기에 따른 성능을 알아보기 위해서 캐쉬 크기를 1MB, 2MB, 4MB, 그리고 8MB 크기로 나누어 실험을 하였다.

적중률 성능 결과를 보면 512MB의 결과와 다르게 BLOCS 기법이 SSD의 크기가 작을 때는 성능이 낮은 것을 볼 수 있다. Bootup 워크로드에서 더 낮은 적중률을 보인다. FREQ 기법에 비해 BLOCS 기법이 약 20% 더 낮은 성능을 보이고 있다. 대체적으로는 캐쉬의 용량이 커질수록 사용한 기법들의 적중률은 높아지는 것을 볼 수 있다. BLOCS 기법은 Bootup 워크로드에서 캐쉬 용량이 커지더라도 적중률이 하락했지만, 나머지 워크로드에서 캐쉬 용량이 8MB일 때 1MB일 때에 비해 평균 44%가 증가하였다. 적중률이 가장 많이 증가한 워크로드는 photoshop으로 3.7배 증가하였다. FREQ와 F-S, 그리고 DIST는 캐쉬 용량이 1MB 대비 8MB가 각각 평균 22%, 21%, 그리고 18% 증가하였다. 모든 워크로드가 photoshop 워크로드에서 적중률이 크게 증가하였다. BLOCS가 3.7배 증가하였고 F-S기법이 2.1배 증가했다. FREQ와 DIST 기법은 둘 다 70% 씩 증가하였다.

4. 마이닝 오버헤드

실제 운영 중인 시스템에 제안한 BLOCS 캐쉬 알고리즘을 적용을 하려면 워크로드가 저장 장치로 내려가기 전 파일 시스템과 디바이스 드라이버 사이에 필터드라이버를 구성해야 한다[22]. 이 필터드라이버는 디바이스가 저장장치로 IO를 처리하기 전에 사용자가 필요로 하는 선행 작업을 수행할 수 있도록 한다. IO를 디스크로 내려 보내기 전에 수행하는 동작이기 때문에 사용자에게 추가 작업으로 인한 오버헤드가 보여서는 안 된다.

표 8. BLOCS 의 마이닝 수행 오버헤드
Table 8. Overhead of Mining with BLOCS Algorithm

	Mining	Total	Overhead
Unit	msec	sec	%
Bootup	4.10	84.42	4.85
Excel	0.67	76.72	0.87
iTunes cat	2.16	61.30	3.53
iTunes	0.02	137.80	0.01
KMplayer	0.32	141.82	0.22
Outlook	0.02	90.12	0.02
Photoshop	0.08	4.34	1.80
Picasa cat	0.79	93.77	0.85
Picasa	0.02	46.76	0.04
Powerpoint	0.02	61.06	0.03

제안한 BLOCS 알고리즘이 수행하기 위해서는 먼저 빈발 부분시퀀스를 선별하여야 한다. 이 선별과정은 최초 1회만 발생되는 오버헤드이지만, 사용자의 워크로드가 변경됨에 따라 주기적으로 선별을 다시 해야 할 수도 있다. 그렇기 때문에 이 시간이 길어서는 안 된다. 마이닝에 소요되는 시간을 분석한 결과를 표 8에 나타내었다.

Bootup, iTunes Cat. photoshop을 제외하면 전체 실행 시간 대비 평균 0.31%의 시간이 소요되었다. iTunes 에 음악 파일을 등록하는 워크로드는 일회성이므로 제외하면 Bootup과 Photoshop 워크로드만 전체 수행 시간의 각각 4.85%와 1.8%의 시간을 마이닝에 사용하였다. Bootup의 경우 캐쉬의 크기가 512MB일 때 실행시간을 40%가까이 줄였으며 61%의 적중률을 보인 것을 감안하면 오버헤드는 큰 의미가 없는 것으로 볼 수 있다. 마찬가지로 Photoshop의 워크로드에서도 가장 낮은 적중률을 FREQ기법에 비해 2배 이상의 높은 적중률과 총 수행시간을 15% 줄일 수 있었기 때문에, 장기적으로 보면 오버헤드를 감수하는 것이 성능측면에서 더 좋은 것으로 보인다.

5. 캐쉬 알고리즘의 SSD에 수명에 미치는 영향

SSD를 캐쉬를 사용하는 이유 중 하나는 HDD의 느린 IO 지연시간을 보완하는 것이다. 그와 더불어 또 하나 고려해야 하는 것은 SSD의 쓰기 수명이 제한되어 있다는 사실이다. 그렇기 때문에 한번 SSD에 기록된 데이터가 갱신되는 빈도와 양을 분석하는 것은 중요하다. 전체 쓰기 요청에 의해 각 저

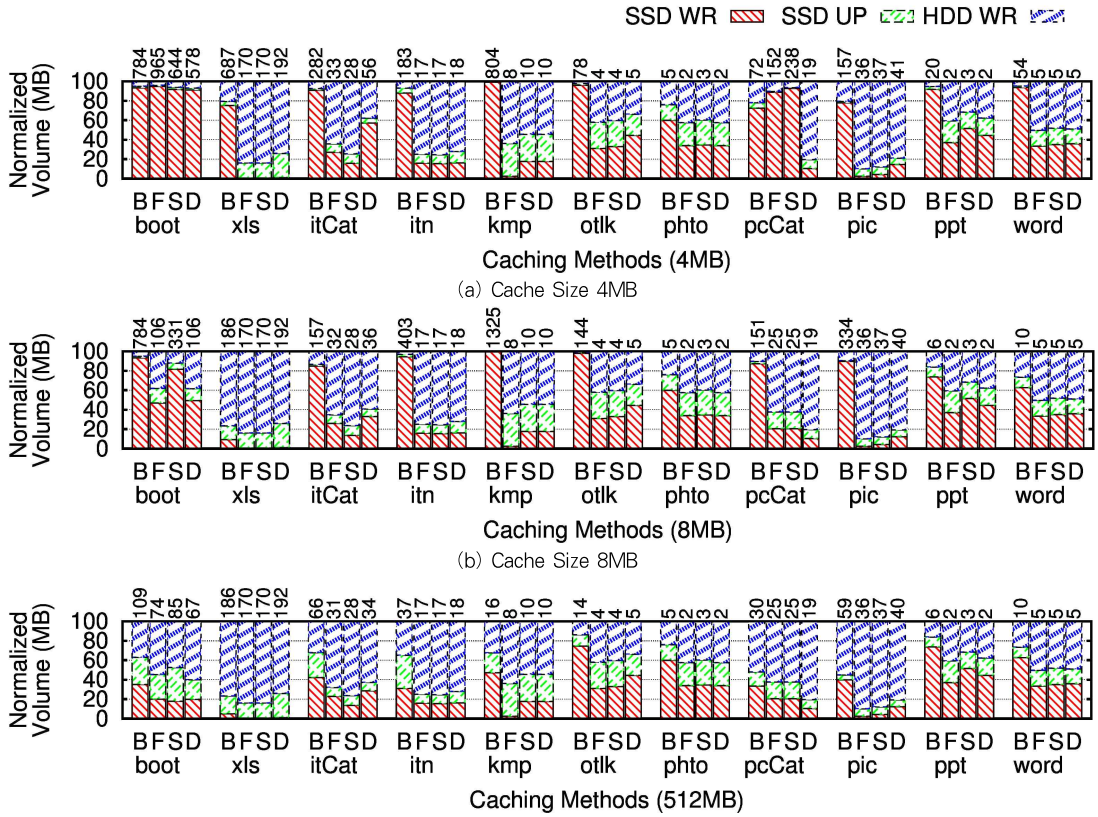

그림 13. 캐쉬 크기의 변경에 따른 캐싱 알고리즘의 저장장치 사용률 (SSD WR: SSD에 기록된 데이터, SSD UP: SSD에 기록된 이후 갱신된 데이터, HDD WR: 캐쉬 되지 않고 HDD에만 기록된 데이터)

Fig. 13. Storage Utilization of Caching Algorithms with Respect to Cache Size (SSD WR: Data Written to SSD, SSD UP: Updated data in SSD After Write, HDD WR: Data Written to HDD)

장장치에 기록된 양과 SSD에 갱신된 용량을 알아보기 위해 그림 13에서는 정규화된 용량을 나타내었다. 캐쉬 용량이 1MB와 2MB일 때는 캐쉬 적중률이 낮고 캐쉬에 쓰고 지우는 량이 너무 많아서 캐쉬 디스크를 쓰는 장점이 없기 때문에 결과에서 제외하였다.

X축은 각 워크로드 별로 사용된 캐싱 알고리즘을 B, F, S, D로 표현하였고, 각각은 BLOCS, FREQ, F-S, 그리고 DIST 기반 교체 알고리즘을 사용한 경우를 나타낸다. Y축은 캐쉬에 기록한 용량, 캐쉬에 기록한 데이터를 갱신한 용량, 그리고 HDD에 쓴 총 용량의 합을 정규화하여 100으로 표현하였다. 그림의 윗부분의 숫자는 SSD와 HDD에서 발생한 쓰기의 총량을 나타낸다. 각 막대그래프는 세 가지의 정보를 표현한다. SSD WR는 캐쉬 알고리즘이 SSD에 쓰기를 수행한 총량을 나타내고 SSD UP은 SSD WR에서 기록된 데이터가 SSD에서 갱신된 양을 나타낸다. 마지막으로 HDD WR은 HDD에서만 쓰기 또는 갱신된 용량의 합을 나타낸다.

그림 13에 나타나 있는 캐쉬 크기 4MB, 8MB, 그리고 512MB의 결과를 보면 SSD의 용량이 작을수록 워크로드에 따라 SSD WR의 비율이 HDD WR의 비율보다 큰 워크로드가 있다. 몇 경우에서는 전체 쓰기 용량이 수 백 MByte가 되는 실험도 있었다. 앞선 실험에서 캐쉬의 용량이 작더라도 대부분 30% 의 적중률을 보였다고 하더라도 캐쉬 용량이 작다면 사용할 수 없음을 나타낸다. 예를 들어 4MB 캐쉬를 사용하였을 때 부팅 워크로드의 경우 모두 500 MB 이상 기록을 하였고 HDD에 기록한 양의 18배를 SSD으로 기록하였다. 캐쉬 용량이 커질수록 boot 워크로드의 경우 SSD WR와 SSD UP 그리고 HDD WR에 기록되는 비율이 비슷하거나 HDD에 기록한 량이 더 많아 졌다. 그림 14에서 보였던 캐쉬

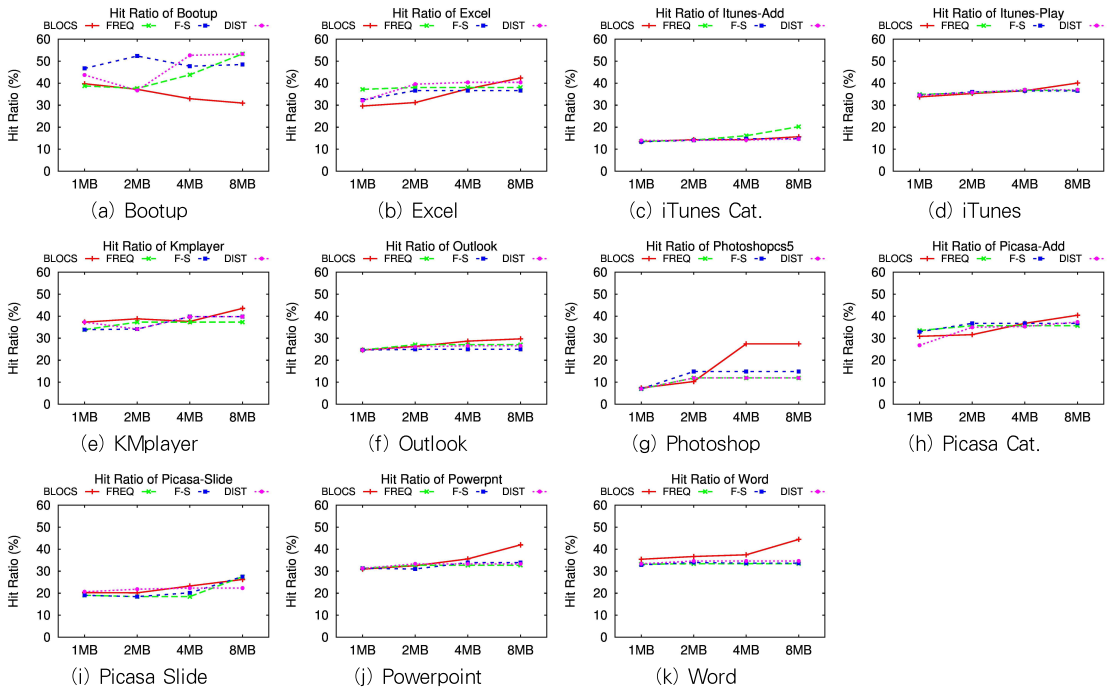

그림 14. 캐쉬 크기에 변화에 따른 캐싱 기법의 성능(WS: 2000, CWS: 100, MS: 2, MG:40)
Fig. 14. Performance of Caching Schemes on different Cache Sizes, (WS:2000, CWS: 100, MS: 2, MG:40)

용량 변경에 따른 적중률 변화 그림과 비교해보면 4MB일 때와 8MB일 때의 적중률은 크게 변화가 없었지만, 그림 13를 보면 캐쉬 디스크에 기록되는 양을 보았을 때는 몇 경우를 제외하고는 기록된 용량이 2배 이상 감소한 것을 알 수 있다.

캐쉬 크기가 8MB일 때와 512MB일 때를 비교해 보면 Excel, Photoshop, Powerpoint, 그리고 Word의 경우는 차이가 없는 것을 알 수 있다. BLOCS기법을 제외하면 iTunes, KMplayer, Outlook, 그리고 Picasa Cat. 워크로드에서도 차이가 없었다. BLOCS 기법은 512MB일 때 총 기록된 용량이 가장 적었다. SSD 쓰기 용량과 SSD 갱신 용량을 보면 캐쉬 저장장치가 받는 부하를 알 수 있다. 워크로드의 갱신 비율이 높을수록 SSD에 기록한 용량이 많기 때문에 SSD의 수명이 짧아지는데, BLOCS 기법이 다른 기법들에 비해 가장 적은 용량을 갱신하였다. BLOCS의 갱신 비율은 평균 72%이고, DIST기법과 F-S기법이 각각 평균 278% 그리고 363% 갱신 비율을 갖고 있다. 워크로드의 전체 평균으로만 보면 FREQ, F-S, 그리고 DIST 기법이 각각 약 7배, 5배, 그리고 4배 정도 캐쉬 저장장치에 더 많이 기록하기 때문에 수명이 BLOCS 보다 더 빨리 줄어들게 된다.

VII. 결 론

SSD의 기술 발전으로 인해서 SSD의 용량이 늘어나고 가격이 하락한 것은 사실이지만 지난 수십 년간 시장을 점유해 온 HDD를 대체하기에는 지속적인 기술 개발이 필요한 것으로 보인다. SSD를 구성하는 낸드플래시는 셀 마모도와 데이터 retention 문제가 있어서 SSD만으로 이루어진 시스템에 대한 신뢰성 문제가 있기 때문이다. 성능과 가격이라는 관점에서 각 장치의 장점을 최대한 활용할 수 있는 방안으로 SSD와 HDD를 동시에 사용하는 하이브리드 스토리지 시스템이 제시되고 있다. 본 연구에서는 SSD를 캐쉬로 사용하는 하이브리드 스토리지 시스템을 구성할 때 데이터를 캐싱하는 방법을 제시하고 하이브리드 캐쉬 시뮬레이터를 개발하여 다양한 캐싱 정책의 성능을 비교한다. 시스템이 부팅할 때 발생되는 워크로드를 포함하여 11개의 응용프로그램 실행 시나리오를 선정하여 실험에 사용할 워크로드를 생성하였다.

본 연구에서 제안하는 BLOCS 기법은 시퀀스 패턴 마이닝을 이용한 것으로 빈번하게 호출되는 데이터의 상관관계를 분석하여 캐쉬에 선택적으로 저장한다. 제안한 기법의 성능을 비교분석하기 위해 거리우선(DIST) 기반, 빈도우선(FREQ)

기반, 빈도와 크기의 곱(F-S) 기반 캐싱 방법을 추가로 구현하였고, 시뮬레이터에서 각 기법의 캐싱 적중률을 분석하였다. Booting 워크로드에서 BLOCS 이 61%의 적중률을 나타내서 최저 성능을 보인 DIST 기법에 비해 15% 더 높은 적중률을 보였다. 캐쉬로 사용하는 SSD의 용량이 큰 경우 적중률과 적중용량을 보았을 때는 BLOCS 알고리즘과 교체정책을 사용하는 것이 좋은 것으로 나타났다. 캐쉬 용량은 커질수록 BLOCS 기법의 적중률이 높아졌다. 캐쉬 크기에 따른 SSD 수명을 살펴보면 FREQ, F-S, 그리고 DIST기법은 캐쉬 크기가 8MB에서 512MB로 변하더라도 쓰이는 양은 변함이 없었지만 적중률은 512MB일 때가 더 좋았다. BLOCS의 경우에는 캐쉬 크기가 512MB일 때가 적중률도 높아지고 SSD에 기록되는 오버헤드도 감소한다. 적재 크기를 전체 워크로드들의 평균으로 으로 보면 FREQ, F-S, 그리고 DIST 기법이 BLOCS기법에 비해 각각 약 7배, 5배, 그리고 4배 정도 더 많이 캐쉬에 기록하기 때문에 캐쉬 저장장치의 수명에는 더 안 좋은 것으로 나타났다.

참고문헌

〔1〕 Seagate. Desktop HDD ST4000DM000 specification. http://www.seagate.com/internal-hard-drives/ desktop-hard-drives/desktop-hdd

〔2〕 Samsung. 512GB 2.5-inch SSD 840 pro series. http://www.samsung.com/us/computer/memory-storage/MZ-7PD512BW-specs

〔3〕 Samsung, *"1g x 8 bit / 2g x 8 bit / 4g x 8 bit NAND flash memory (K9K8G08U0A)."* 2006.

〔4〕 Samsung, *"1g x 8 bit / 2g x 8 bit NAND flash memory (k9lag08u1a),"* 2007.

〔5〕 Samsung. What is V-NAND and how is it different to existing technology? http://www.samsung.com/global/business/semiconductor/html/product/flash-solution/vnand/overview.html

〔6〕 High-Capacity SSDs Finally Match the per-GB Prices of Smaller SSDs. http://dealnews.com/features/High-Capacity-SSDs-Finally-Match-the-per-GB-Prices-of-Smaller-SSDs/622014.html

〔7〕 Solid-State Drives Will Complement, Not Replace, Hard-Disk Drives in Data Centers. https://www.gartner.com/doc/2427717/solidstate-drives-complement-replace-harddisk

〔8〕 Jae-Duk Lee, Sung-Hoi Hur, and J.-D. Choi, "Effects of floating-gate interference on NAND flash memory cell operation," Electron Device Letters, IEEE, vol. 23, pp. 264-266, 2002.

〔9〕 T. Kgil, D. Roberts, and T. Mudge, "Improving NAND Flash Based Disk Caches," in Computer Architecture, 2008. ISCA '08. 35th International Symposium on, pp. 327-338. Beijing, China, June, 2008

〔10〕 J.-W. Hsieh, T.-W. Kuo, P.-L. Wu, and Y.-C. Huang, "Energy-efficient and performance enhanced disks using flash-memory cache," presented at the Proceedings of the 2007 international symposium on Low power electronics and design, Portland, OR, USA, Aug. 2007.

〔11〕 T. Bisson and S. A. Brandt, "Reducing Hybrid Disk Write Latency with Flash-Backed I/O Requests," in Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS '07. 15th International Symposium on, pp. 402-409. Istanbul, Turkey, Oct. 2007

〔12〕 Y. Joo, Y. Cho, K. Lee, and N. Chang, "Improving application launch times with hybrid disks," presented at the Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, pp. 373-382, Grenoble, France, Oct. 2009.

〔13〕 Y. Joo, J. Ryu, S. Park, and K. G. Shin, "FAST: quick application launch on solid-state drives," presented at the Proceedings of the 9th USENIX conference on File and storage technologies, pp. 19-19, San Jose, California, USA. 2011.

〔14〕 R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," presented at the Proceedings of the 11th USENIX conference on File and Storage Technologies, pp. 45-58, San Jose, California,

USA. 2013.
〔15〕 D. A. Holland, E. L. Angelino, G. Wald, and M. I. Seltzer, "Flash caching on the storage client," in USENIX ATC'13 Proceedings of the 2013 USENIX conference on Annual Technical Conference, pp. 127-138, San Jose, California, USA. 2013.
〔16〕 M. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," Machine Learning, vol. 42, issue 1-2, pp. 31-60, Jan, 2001.
〔17〕 J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential PAttern mining using a bitmap representation," presented at the Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, Edmonton, pp. 429-435, Alberta, Canada, 2002.
〔18〕 X. Yan, J. Han, and R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Datasets," in In proceedings of the third SIAM International conference on data mining, pp. 166-177, San Francisco, CA, May. 2003.
〔19〕 Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-Miner: mining block correlations in storage systems," presented at the Proceedings of the 3rd USENIX conference on File and storage technologies, pp. 13-13, San Francisco, CA, Mar. 2004.
〔20〕 DiskMon. http://technet.microsoft.com/en-us/sysin ternals/bb896646.aspx
〔21〕 NTFSInfo. http://technet.microsoft.com/en-us/sys internals/bb897424
〔22〕 California Software Labs, *"I/O file system filter driver for Windows NT,"* CSWL INC Technical Report, Pleasanton, California, 2002.

저 자 소 개

이 성 진

2006: 한양대학교 전자전기컴퓨터공학과 학사.
2008: 한양대학교 전자컴퓨터통신공학과 석사.
현 재: 한양대학교 전자컴퓨터통신공학과 박사과정 재학 중
관심분야: 운영체제, 파일시스템, 스토리지 시스템, 모바일 스토리지
Email : insight@hanyang.ac.kr

원 유 집

1990: 서울대학교 계산통계학과 학사.
1992: 서울대학교 계산통계학과 석사.
1997: University of Minnesota 전산학 박사
현 재: 한양대학교 컴퓨터소프트웨어학과 교수
관심분야: 운영체제, 파일시스템, 스토리지 시스템, 모바일 스토리지
Email : yjwon@hanyang.ac.kr