Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters

Dongchul Park and David H.C. Du
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, MN 55455, USA
Email: {park, du}@cs.umn.edu

***Abstract*—Hot data identification can be applied to a variety of fields. Particularly in flash memory, it has a critical impact on its performance (due to a garbage collection) as well as its life span (due to a wear leveling). Although the hot data identification is an issue of paramount importance in flash memory, little investigation has been made. Moreover, all existing schemes focus almost exclusively on a *frequency* viewpoint. However, *recency* also must be considered equally with the frequency for effective hot data identification. In this paper, we propose a novel hot data identification scheme adopting multiple bloom filters to efficiently capture finer-grained recency as well as frequency. In addition to this scheme, we propose a Window-based Direct Address Counting (WDAC) algorithm to approximate an ideal hot data identification as our baseline. Unlike the existing baseline algorithm that cannot appropriately capture recency information due to its exponential batch decay, our WDAC algorithm, using a sliding window concept, can capture very fine-grained recency information. Our experimental evaluation with diverse realistic workloads including real SSD traces demonstrates that our multiple bloom filter-based scheme outperforms the state-of-the-art scheme. In particular, ours not only consumes 50% less memory and requires less computational overhead up to 58%, but also improves its performance up to 65%.**

***Index Terms*—Hot Data Idenficiation, Hot and Cold Data, Bloom Filter, Flash Memory, SSD, WDAC**

1) Motivations / Problems (연구의 동기)

2) Key Ideas (핵심 아이디어)

3) Design (제시한 아이디어를
실현하기 위한 디자인이나 구현 방법)

4) Performance (주요 요소의
성능 향상 수치)

5) Personal Opinion (이 논문 아이디어나
디자인에 대한 문제 제기 또는
개선을 위한 본인의 의견)

to read the paper : HERE

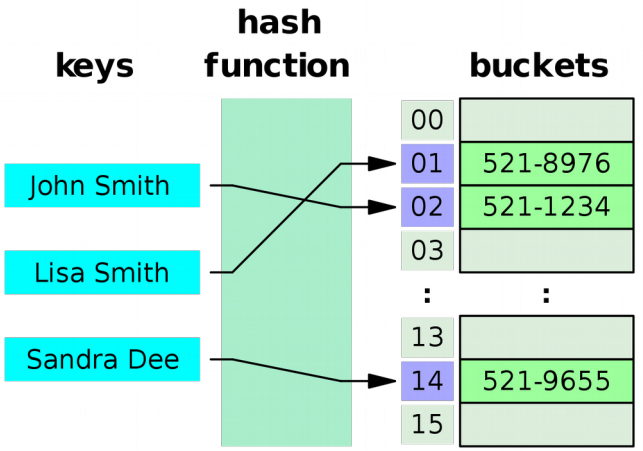
Background

Flash-based Storage System 의 특징
NVM 의 기본적인 특성은 RD, WR 는 page 단위로 실시하지만 ER 는 block 단위로 이루어진다는 사실
본질적으로 flash memory 는 in-place overwrite 가 불가능
이미 존재하는 데이터를 update 요청이 들어오면 write-copy-erase, copy-write-erase 방식으로 진행
GC(garbage collection) 의 필요성이 대두됨
또 한 가지 , 각 셀 (cell) 은 제한적인 P/E(write&erase) cycle 을 지님
이를 극복하기 위해서 wear leveling 의 필요성이 대두됨

FTL(Flash Translation Layer) 이 탄생함
피상적으로 보면 FTL 은 플래시 메모리 외부 시스템에서 인식하는 LBA(Logical Block Address) 를
storage 내부의 PBA(Physical Block Address) 로 바꾸어주는 역할
기존 HDD 인터페이스를 그대로 MTD(Memory Technology Device) 에 적용 가능해진다는 장점
여기에는 다양한 기술과 알고리즘이 적용되어 있고 ,
이를 통해서 wear leveling & garbage collection 를 보다 효율적으로 처리할 수 있게 됨
이 과정에서 hot data – cold data 를 구분할 필요가 생겼다 .
hot 은 write operation 이 다수 발생하는 경우를 의미한다 .
cold 은 한번 저장된 데이터가 새로운 write request 를 받지 않고 오랫동안 저장되는 걸 의미한다 .

Background

Hash Table 은 input 으로 들어온 어떤 값을 임의의 hash function 에 파라미터로 넘겨서
return value 를 얻고 , 이렇게 얻은 값을 key 로 하는 테이블에 저장하는 자료구조
Hash function 예시 - 인풋값을 매개로 복잡한 연산을 거쳐서 얻어낸 값을
‘ 충분히 큰’ 소수 (prime number) 에 대한 mod(%) 연산을 하는 경우
해쉬 함수의 성능은 해쉬 테이블의 모든 key 에 고르게 접근하도록 설계할 수록 성능이 좋다고 평가
이와 같이 구성하면 time complexity O(1) ~ O(N), space complexity O(N) 만으로
데이터를 저장 접근하는 것이 가능
→ 장점 : 성능이 좋다 (속도가 빠르다), 데이터별 고유값 지정 가능
→ 단점
1) hash collision (서로 다른 데이터가 동일한 키에 접근하는 경우)
2) false positive (존재하지 않는 데이터가 존재하는 것으로 판별되는 경우)
3) hash table size (데이터의 양에 비례해서 hash table 의 크기도 증가)



Background

bloom filter 을 이해하기 위해서는 스팸필터링을 상상하면 편리하다 . 스팸에 해당하는 단어나 패턴을 모두 스팸리스트에 집어넣은 뒤에 새로운 메일이 들어올 때마다 필터와 대조해보면서 스팸메일을 걸러낸다 .

Bloom filter 에서의 스팸필터 같은 역할을 하는 것이 hash function 이다 .

블룸 필터는 1 bit 만 저장하는 m 개의 요소로 이루어진 배열이다 . 처음에는 전부 0 으로 세팅한다 .

서로 다른 k 의 해시 함수는 배열의 m 개의 요소 중 하나로 매핑되며 ,

각각의 해시 함수는 전체 블룸 필터에 골고루 접근하도록 설계되어야 한다 .

k 의 값은 어느 정도의 false positive 를 허용하는 지에 따라서 다르게 지정

k = m / n * ln2 일 때 FP 가 최소가 된다 . (m: size of bloom filter, n: total number of inputs)

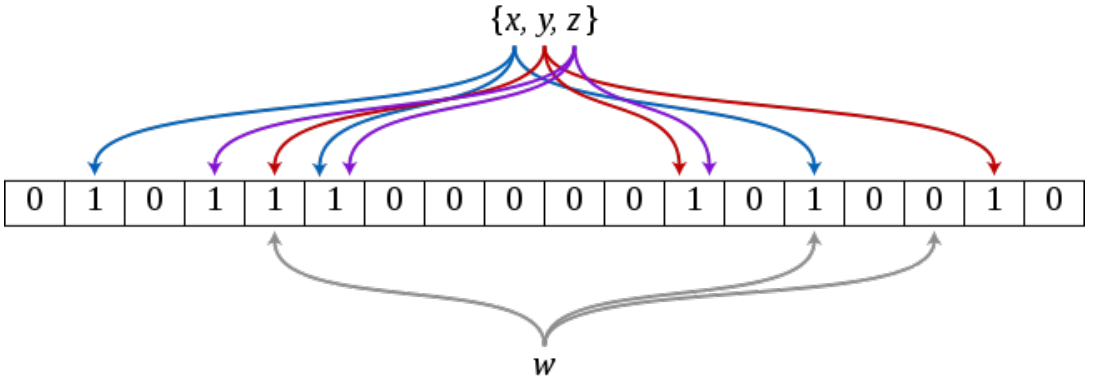
→ 장점 : false positivie 감소 , 메모리 사용량 감소 (smaller hash table)

그러나 여기서 더욱 발전시킬 수는 없을까 ?

단점) BF 를 작동시킨 뒤 세팅을 위한 데이터를 추가적으로 계속 집어넣을 경우 모든 비트가 1 로 바뀜 (high possibility of false positive)

1) 1 bit is Not Enough, we need more information

2) is it possible to decrease the size of bloom filter or decrease the number of hash func?



Background

Counting Bloom Filter 는 배열에 boolean 값 대신에 integer 값을 집어넣는다 .
Hash function 로 매핑되는 인덱스의 값은 +1, 매핑 취소할 때는 -1

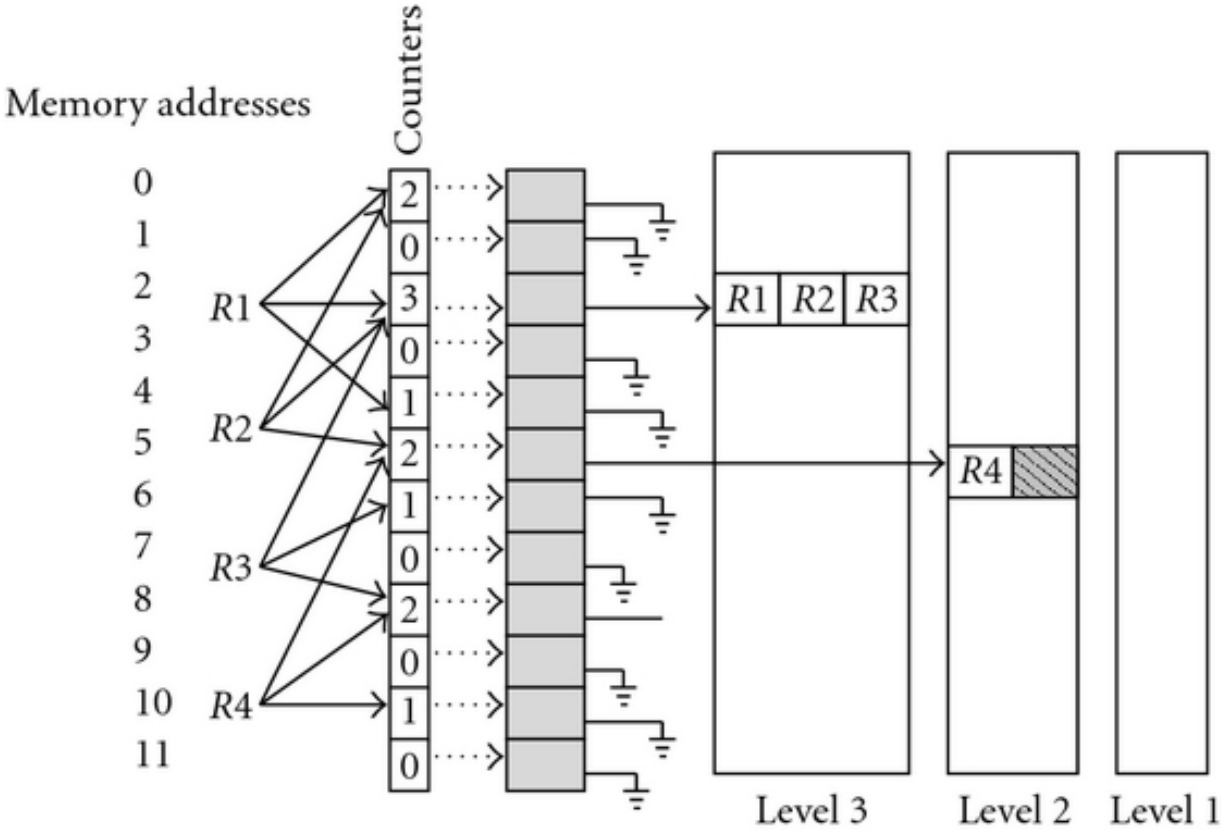


fig. cache architecture with counting bloom filter

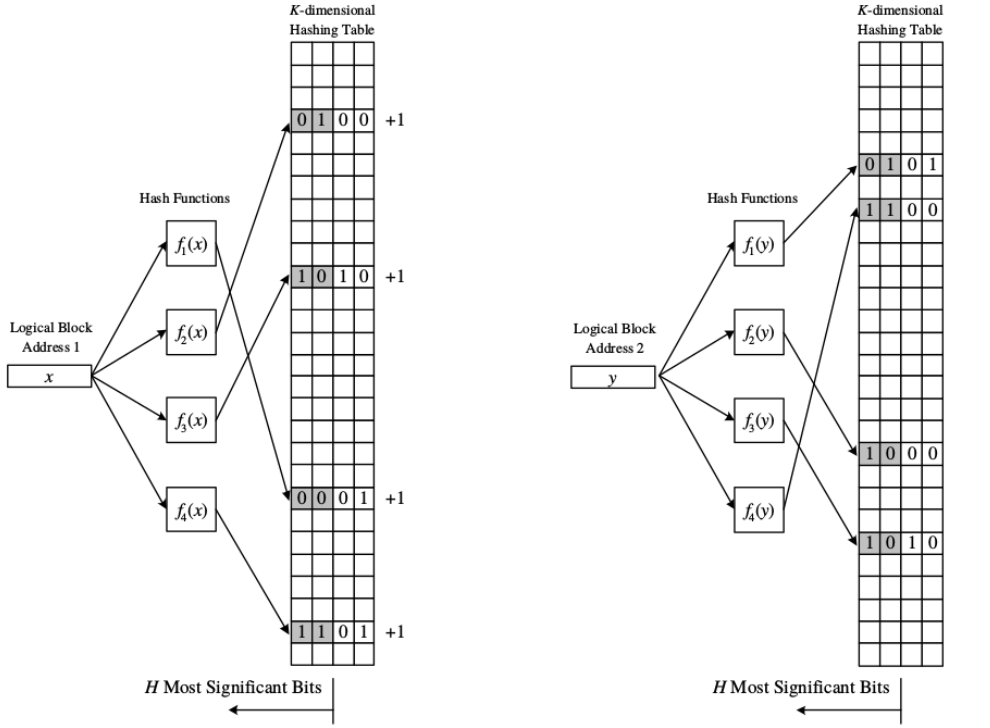
Background

scheme 과 **Baseline-Algorithm** (용어 정리)
scheme : (noun) a large-scale systematic plan or arrangement
baseline : (noun) a minimum or starting point used for comparisons
그러므로 scheme 은 아이디어의 원천이 되는 논리 , baseline algorithm 은 논문에서 내세우려는 알고리즘의 초안 정도로 생각하면 된다 .

기존 논문이라는 말이 나올 때마다
<Efficient identification of hot data for flash memory storage>
본 논문이라는 말이 나올 때마다
<Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters>
라고 생각하면 된다 .

Motivations / Problems

이 논문이 작성되기 이전까지 Hot Data Identification 영역에서 가장 우수한 평가를 받고 있던 최신 논문은 <Efficient Identification of Hot Data for Flash-Memory Storage Systems>(Hsieh et al) 이었다 . 기존 논문인 Hsieh 의 페이퍼에서는 multiple hash functions 와 counting bloom filter 를 이용해서 자신들의 scheme 을 작성한다 . 새로운 WR 요청이 들어올 때마다 그 LBA 를 K 개의 해시 함수를 각각 사용해 M bits 요소를 지니는 CBF 로 매핑한다 . 기존 논문에서 각 요소는 4 bit 값을 지니며 , 매핑된 해당 요소의 값은 +1 된다 . 만약 이 4 bit 중에서 MSB(most significant bits) 즉 상위 2bit 중 하나라도 1 이 있는 경우 Hot Data 로 간주한다 .



(a) The Counter Updating of an LBA (b) The Hot-Data Identification of an LBA

Motivations / Problems

Hot data 는 자주 WR 접근되는 데이터라는 말과 같은데 과거에 자주 접근되는 것보다는 최근에 자주 접근된다는 사실이 보다 의미있는 정보이다 . 이런 논리에 기반해서 기존 논문의 저자는 recency 정보를 고려하려는 의도에서 decay effect 를 추가한다 . 일정한 주기 T
(T <= M / (1 – R), M : size of CBF, R : hot ratio of a workload – here they assumed 20%) 마다 블룸필터에 저장된 모든 값을 right-shift 한다 . R = 0.2 일 때 false identification rate 가 최소가 된다고 분석했다 .

Motivations / Problems

그러나 지금 우리가 살펴보려는 <Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters> 의 저자는 기존 논문에서 제시하는 방법보다 훨씬 더 우아한 방식으로 recency 정보를 포착할 수 있다고 주장한다 . 본 논문의 저자는 hot data identification 을 위한 방안이 지녀야 할 훌륭한 세 가지 속성으로 (1) frequency 정보는 물론이고 recency 정보도 포착할 수 있어야 하고 (2) 메모리 사용량이 적어야 하고 (3) 컴퓨팅 오버헤드가 낮아야 한다는 것을 내세운다 . 그는 이런 관점에서 기존 논문에 보완할 부분이 존재한다고 주장하며 보다 발전된 방안을 제시한다 .

Key Ideas

첫번째 , scheme 에서는 연산 복잡도를 낮추고 공간을 적게 사용하기 위해서 bloom filter 를 사용한다 . Finer-grained recency 정보를 얻어서 hot data identification 정확도를 증가시키기 위해서 여러 개의 BF 를 사용한다 .

두번째 , baseline algorithm 으로 Window-based Direct Address Counting(WDAC) 를 고안했다 . 쓰기 요청이 발생할 때마다 해당 LBA 를 FIFO 윈도우에 집어넣는다 . 모든 LBA 각각에 대해서 서로 다른 recency weight 를 할당한다 .

Design

1) Scheme (= The Framework)

본 논문에서 저자는 CBF 가 아니라 V number of BF(M number of 1 bits array each) 를 사용한다 .

K 개의 서로 다른 해시 함수를 사용한다 . 이를 통해 frequency 와 recency 를 모두 알아낸다 .

기본적인 논리는 간단하다 . FTL 에 새로운 WR 요청이 발생하면 해당 LBA 로 K 개의 해시 함수를 실행 .

첫번째 지정 BF 의 해당 값을 1 로 만든다 .

다음 WR 요청이 도달하면 다음 지정 BF 의 해당 값을 1 로 만든다 . 이런 방식으로 진행된다 .

* configuring Frequency

만약 뒤에 발생한 WR 가 이전에 발생한 WR 와 hash collision 을 일으킬 경우 바로 다음 BF 로 가서 해당 인덱스 값을 1 로 바꾼다 . 여기도 이미 1 이라면 , 0 인 값을 찾을 때까지 모든 BF 를 탐색한다 .

만약 V 개의 BF 가 전부 1 값을 지닌다면 해당 LBA 를 hot data 로 분류한다 .

해당 주기 (T) 종료시까지 해당 LBA 는 항상 hot data 로 인식된다 .

Design

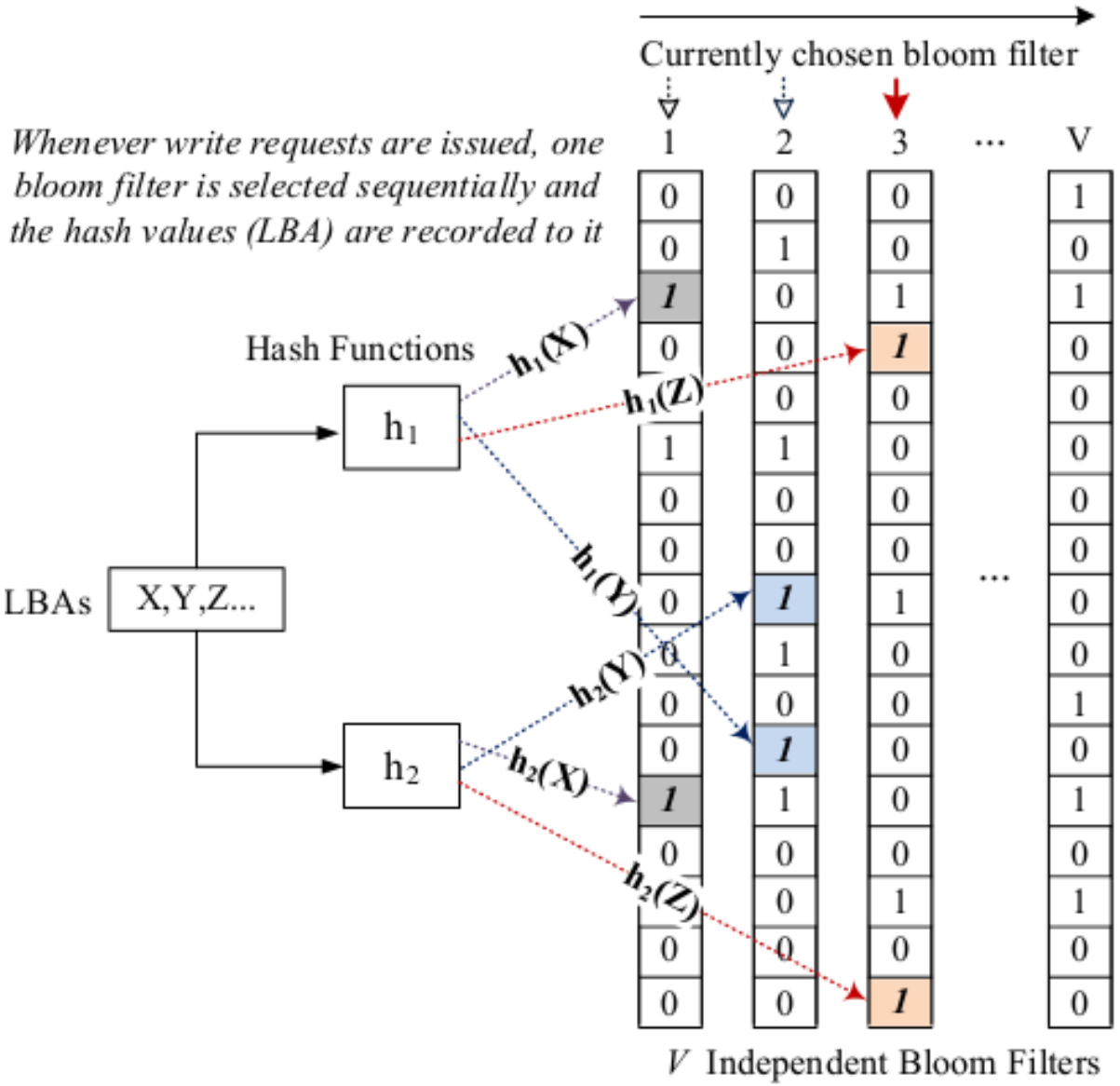


Fig. 3. Our Framework and Its Operations

Design

1) Scheme (= The Framework)

* configuring Recency

동일한 frequency 를 지니는 두 개의 LBA 라 하더라도 과거에 빈번히 업데이트 된 데이터와 최근에 가까운 시점에 자주 WR 요청을 받은 데이터는 서로 다르게 처리되어야 한다 .

T number of WR-Request 를 주기 (interval) 로 해서 M 개의 BF 를 round robin 방식으로 운영한다 .

예를 들어 V 번째 BF 를 reset 한다면 BF_v 는 최신 정보만을 담고 있고

BF_v-1 은 그 다음으로 T+ 최신 정보 , BF_1 은 가장 오래된 정보부터 담고 있는 상태가 된다 .

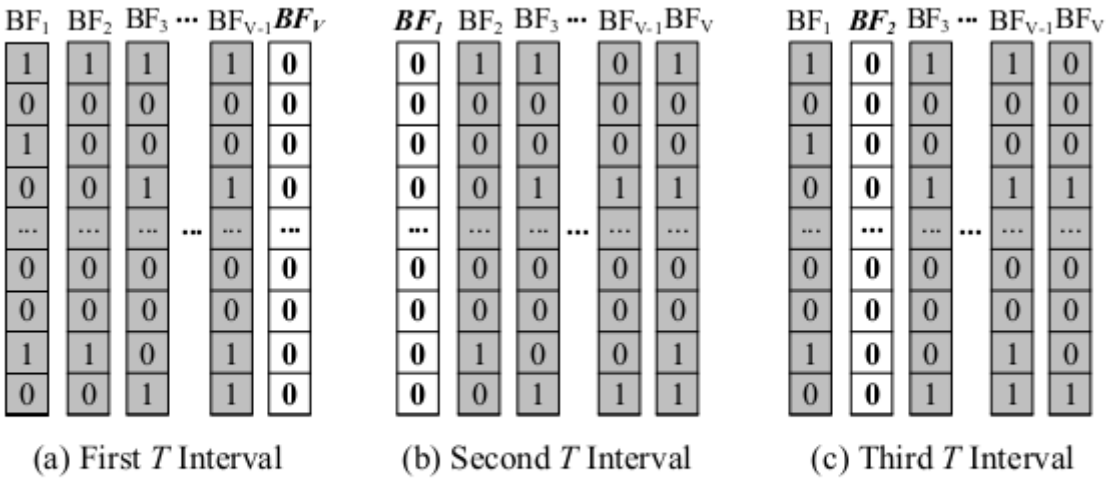
여기에 가중치 정보를 더한다 . 가중치는 이번에 reset 하는 V 번째를 마지막으로 해서 floor[V/2] 번째 BF 에 weight = 1 로 지정한 뒤 각 BF 간의 가중치 간격을 1 / (V – floor[V/2]) 로 설정한다 .

최적의 BF 사이즈 : M = N * K / ln2 (N : element set size, K : number of hash function)

본 논문에서 저자는

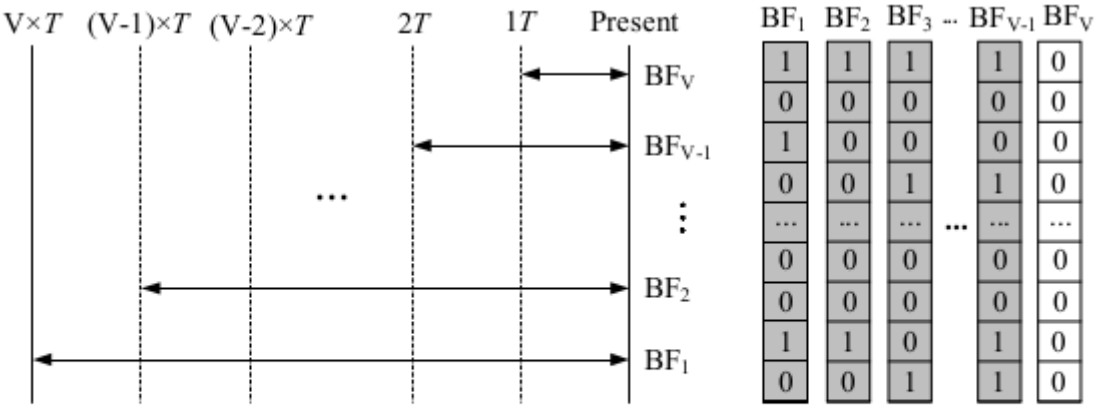
decay interval T = M / V (M : size of each BF, V : number of total Bfs) write requests 와 같이 설정했다 . T 개의 쓰기 요청이 발생할 때마다 V 개의 BF 가 round robin 방식으로 reset to 0.

Design



(a) First T Interval (b) Second T Interval (c) Third T Interval

Fig. 4. Our Aging Mechanism. Here, a white (i.e., not-shaded) bloom filter corresponds to a reset bloom filter



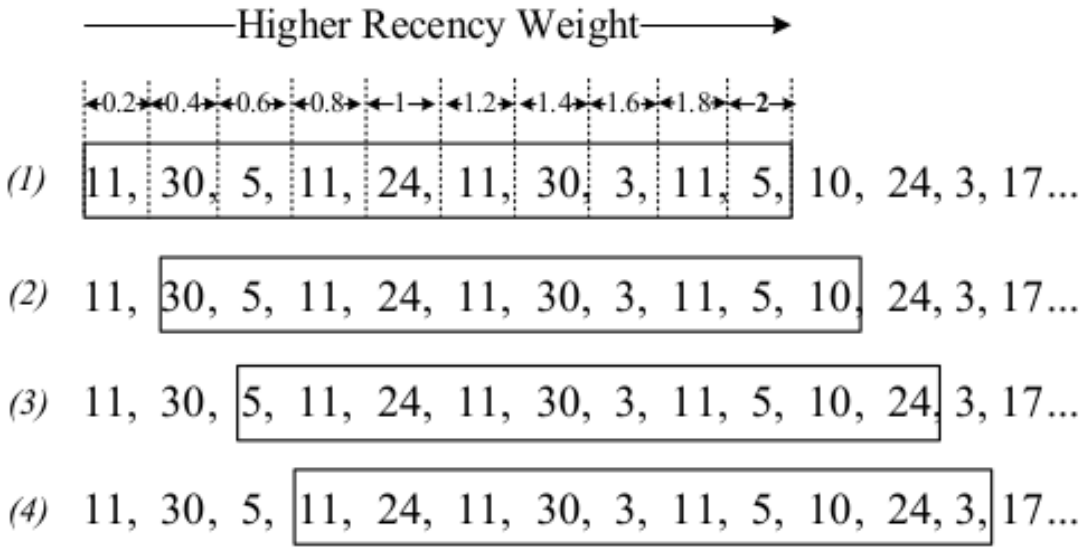
(a) Recency Coverage (b) Bloom Filter Status

Fig. 5. Recency Coverage for Each Bloom Filter

Design

2) Baseline-Algorithm (WDAC : Window-based Direct Address Counting)
기존 논문의 방식에서는 하나의 주기 T 에 속한 WR 요청은 모두 다 동일한 recency value 를 지니기에 정밀도가 떨어진다고 본 논문의 저자는 분석한다 . 여기서는 보다 정밀하게 recency 값을 얻어내는 방법을 소개한다 .
윈도우 크기 W 에 대해서 윈도우에 속한 각 요소 사이의 가중치 차이는 maximum-value / W 가 된다 .
본 논문의 저자는 윈도우 크기를 10, 가장 최근에 enqueue 된 LBA 의 가중치를 2 로 두고 가중치 간격을 0.2 (= 2 / 10) 로 설정했다 . 새로운 WR 요청이 발생하면 윈도우에서 하나의 요소를 dequeue 하고 윈도우에 있는 모든 요소의 가중치를 -0.2 씩해서 재조정한다 . 이런 과정을 보여주는 것이 figure 7(b) 이다 .
WDAC 알고리즘에서 윈도우 크기가 커질수록 계산에 포함되는 LBA 개수가 늘어나므로 hot ratio 도 증가한다 .

Design



(a) LBAs and Sliding Window

(1)

LBA	HDI
11	4.0
30	1.8
5	2.6
24	1.0
3	1.6

(2)

LBA	HDI
11	3.2
30	1.4
5	2.2
24	0.8
3	1.4
10	2.0

(3)

LBA	HDI
11	2.6
30	1.0
5	1.8
24	2.6
3	1.2
10	1.8

(4)

LBA	HDI
11	2.0
30	0.8
5	1.4
24	2.2
3	3.0
10	1.6

(b) Total Hot Data Index for Each LBA

Fig. 7. Working Process of WDAC Algorithm. Here, the window size is 10. HDI corresponds to the total hot data index value.

Performance

실험은 Hsieh et al 의 기존 논문에서 제안한
MHF(Multiple Hash Functions) 방법 & DAM(Direct Address Method) 알고리즘과
본 논문에서 제안하는
MBF(Multiple Bloom Filters) 방법 & WDAC(Window-based Direct Address Counting) 알고리즘 위주로 이루어졌다 .

두 방법 사이의 차이를 고려해서 논리적으로 최대한 같은 조건이 되도록 구성했다 .
기존 방식은 CBF size 4096, 하나의 엔트리마다 4 bits 씩 기록 , interval T = 5117 로 설정했다 .
새로운 방식은 2048 개의 엔트리를 지니는 4 개의 BF, interval T = 512 로 설정했다 .

실험을 위해서 사용한 데이터는 총 네 종류이다 . 실제 상황에 가까운 데이터를 선정하고자 노력했다 .

Finanacial1 : R (22%), W (78%), OLTP 애플리케이션을 실행하고 있는 금융기관 시스템에서 발생한 데이터
MSR : R (4.5%), W (95.5%), 마이크로소프트 연구소의 서버 컴퓨터에 들어온 블록 I/O 요청 데이터
Distilled : R (52%), W (48%), 실제 노트북에서 웹서핑하고 영화보고 게임하고 문서작업하는 사용자 패턴
RealSSD : R (51%), W (49%), 저자들이 가지고 있는 SSD 에서 한 달 동안 발생한 블록 I/O 요청 데이터

RD/WR 요청 개수는 LBA 당 카운트 하나로 친다 . LBA 100 부터 104 까지 쓰기 요청이 들어오면 I/O Request 카운트는 5 가 된다 .

Performance

성능 메트릭스

: hot ratio (이 모든 노력은 Hot Data Identification 을 효율적으로 하기 위함)

: false identification rate (Hot Data 라고 판별했는데 거짓일 경우 효용성이 높지 않다고 판단할 수 밖에)

: memory consumption (아무리 훌륭한 알고리즘이어도 제한적인 MTD 의 SRAM 에서 적용될 수 없다면 활용 가치는 높지 않음)

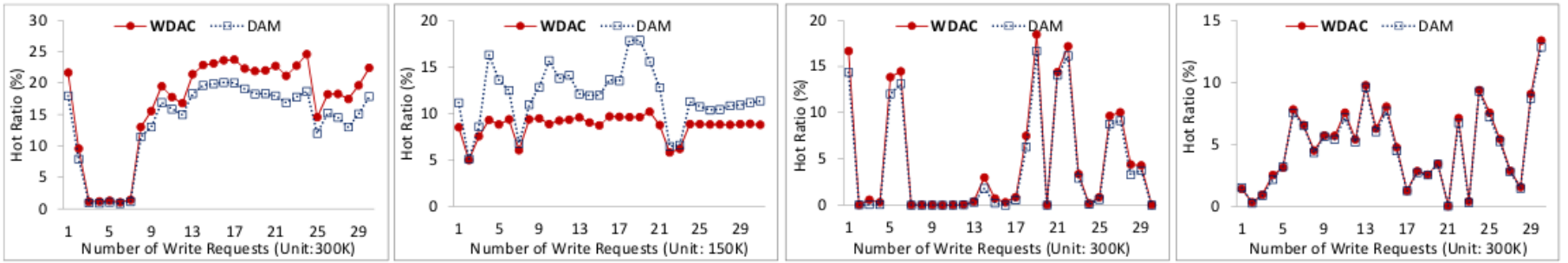
: runtime overhead (CPU clock cycle per operation) (아무리 훌륭한 알고리즘이어도 시스템에 부하를 주거나 빅데이터 상황에 제대로 대응하지 못한다면 활용 가치는 높지 않음)

Performance

A) Baseline Algorithm 비교 (WDAC vs DAM)

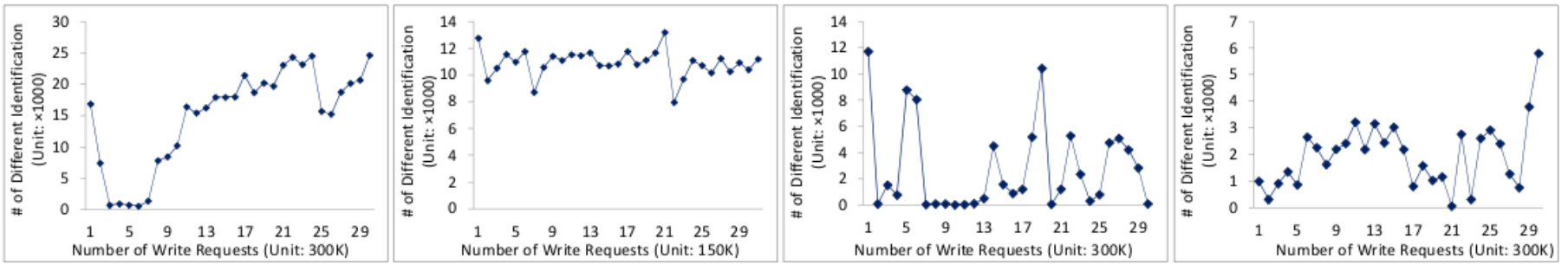
그림 8 에 따르면 hot ratio 측면에서 Financial1 & MSR 는 서로 다르고 , Distilled & ReadSSD 는 아주 유사

이것만 가지고는 알 수 없으므로 그림 9 처럼 false identification rate 를 측정해봤다 . 설령 hot ratio 가 비슷하더라도 WDAC 방식이 보다 넓은 범위의 LBA 를 'hot data' 로 포함시켜서 효율적으로 작동했다 .



(a) Financial1 (b) MSR (c) Distilled (d) RealSSD

Fig. 8. Hot Ratios of Two Baseline Algorithms



(a) Financial1 (b) MSR (c) Distilled (d) RealSSD

Fig. 9. The Number of Different Identification between Two Baseline Algorithms

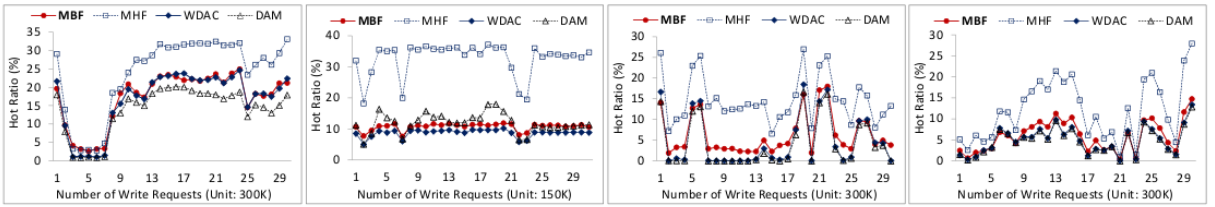
Performance

B) Scheme 성능 비교 (MBF vs MHF)

기존의 MHF 방식은 hash collision 을 막을 수 없기 때문에 시간이 지날수록 false identification rate 가 높아질 수 밖에 없다 . cold data 위주로만 I/O Request 가 일어나더라도 hot ratio 가 점진적으로 증가하기 때문이다 . 이에 대한 해결책으로 해시 테이블을 증가시키거나 일정한 간격으로 decay 를 할지라도 이런 아이디어가 본질적인 해결책이 되지 못한다 .

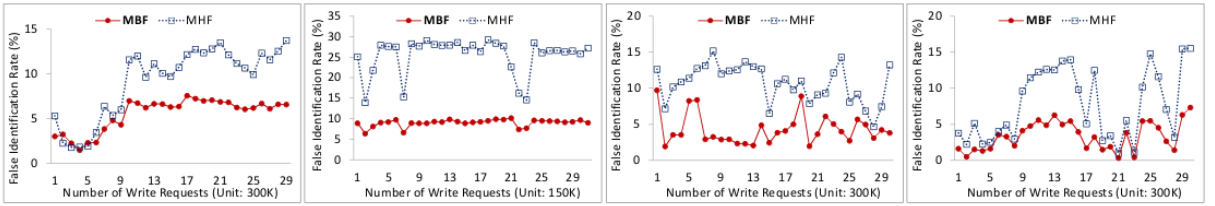
이에 반해 MBF 방식은 threashold value 를 초과한 LBA 의 I/O 요청에 대해서 더 이상 블룸필터의 값을 증가시키지 않으므로 위와 같은 문제는 발생하지 않는 특장점을 지니고 있다 .

실제로 정 결과에서도 새로운 MBF 방식이 네 번의 테스트에서 각각 41%, 65%, 59%, 36% 발전된 성능을 보여준다 . 그림 10 을 보면 알 수 있듯이 MHF 혼자 동떨어진 성능을 보여준다 . 그림 11 을 보면 MBF 가 얼마나 안정된 툴이 될 수 있을지 보다 극명하게 알 수 있다 .



(a) Financial1 (b) MSR (c) Distilled (d) RealSSD

Fig. 10. Hot Ratios of Four Schemes under Various Traces



(a) Financial1 (b) MSR (c) Distilled (d) RealSSD

Fig. 11. False Identification Rates of Both MBF and MHF

Performance

C) Runtime Overhead 성능 비교

AMD X2 3800+ (2GHz) CPU 와 2G 의 RAM 메모리를 장착한 Windows XP Professional 에서 MSR 데이터를 가지고 테스트를 진행했다 . 처음에는 cache-miss 가 발생해서 많은 clock cycle 이 낭비되기도 했지만 데이터가 메모리로 어느 정도 올라온 뒤로는 태스크 처리가 신속해졌다 .

check operation 이란 새로운 I/O 요청이 발생했을 때 Scheme 이나 Baseline Algorithm 으로 진입하기에 앞서 해당 LBA 가 hot data 인지 확인하는 절차를 말한다 . checkup 부분에서 MBF 는 한 비트만 판별하기에 Counting BF 를 사용하는 MHF 에 비해 6% 정도의 적은 오버헤드를 지닌다 .

MBF 방식이 첫번째 BF 부터 마지막 BF 까지 차례로 기록 가능한 비트가 있는지 확인하며 순환하기 때문에 추가적인 오버헤드를 일으키는 것처럼 보일 수도 있다 . 그러나 한번 hot data 로 판별되고 나면 더 이상 위의 작업은 하지 않으므로 오히려 오버헤드가 사라지는 장점이 있다 . 그림 12 를 참고하자 . decay 프로세스는 당연히 더 적은 비트를 조작하는 MBF 가 우월한 성능을 보여준다 .

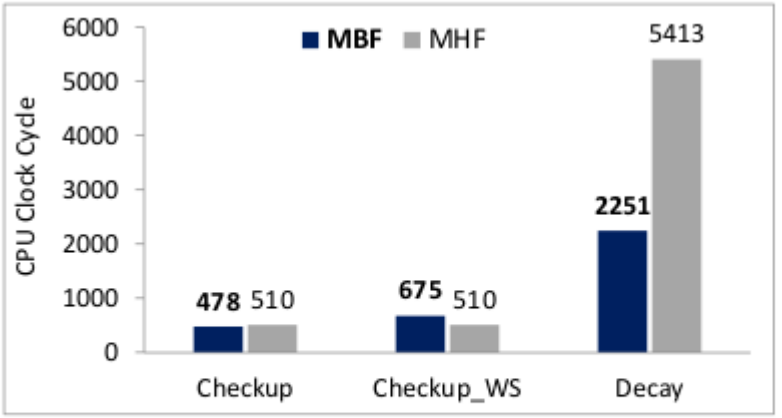


Fig. 12. Average Runtime Overhead per Operations. Here, Checkup_WS means the checkup operation *without* a shortcut decision.

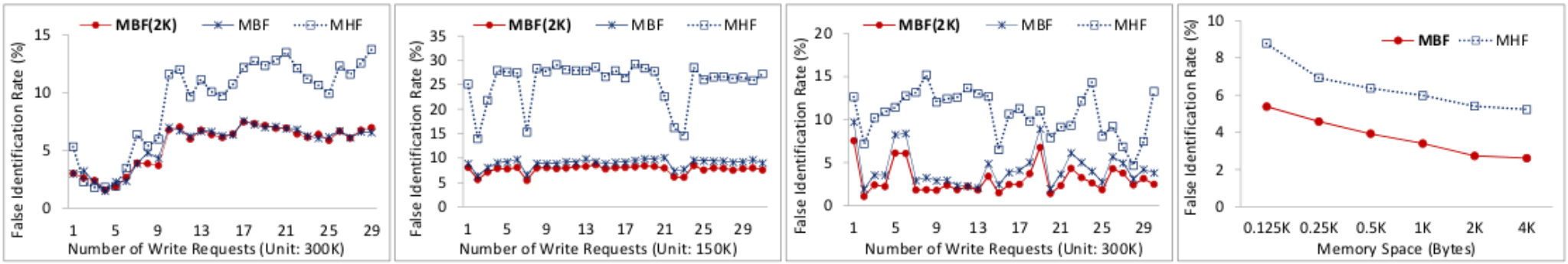
Performance

D) Impact of Memory Size 관점에서

MBF 의 메모리 사용량은 MHF 의 절반이다 .

그림 13 은 동일한 메모리 사용량에서 false rate 가 어떻게 변하는지를 보여준다 .

그래프를 통해서 알 수 있듯이 메모리 사용량이 증가할수록 정확도도 증가한다 .



(a) Financial1 (b) MSR (c) Distilled (d) RealSSD

Fig. 13. False Identification Rates between Two Schemes with Same Memory Space. Here, original MBF and MHF requires 1KB and 2KB respectively. Figure (d) shows the performance change over various memory space under RealSSD traces.

Performance

E) Impact of Window Size

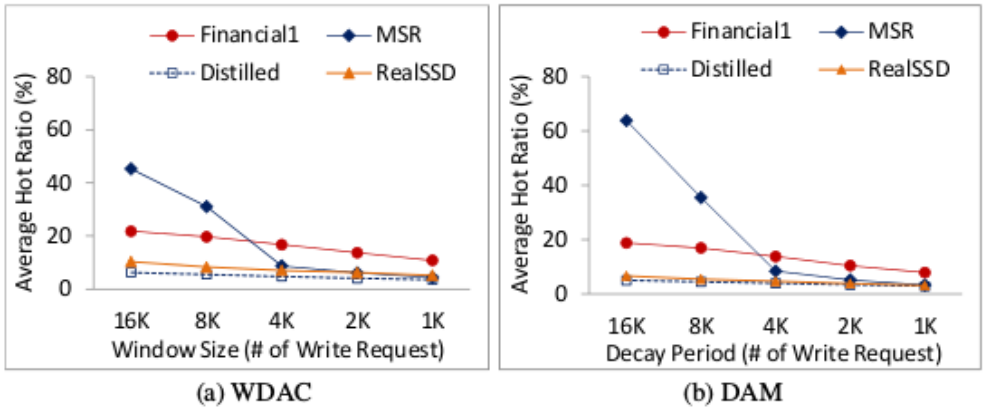
그림 4 를 보면 윈도우 크기를 크게 할수록 그 안에 포함되는 LBA 개수가 증가해서 hot data ratio 도 증가함을 알 수 있다 .

F) Impact of the Number of BF

새롭게 제안하는 MBF-WDAC 에서 BF 의 개수는 recency 정밀도와 밀접한 상관관계를 이룬다 . 이를 수치적으로 확인하기 위해서 BF 사이즈를 다양하게 바꾸면서 실험해보았다 .

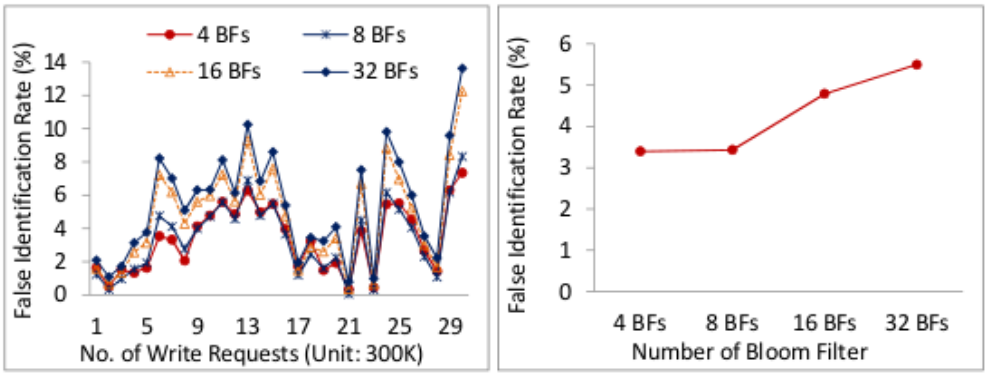
BF 의 개수가 증가하면 false rate 도 증가한다 . 이 실험에서 메모리를 제한하는 조건을 두었기 때문에 나타난 결과이다 . BF 개수를 늘리면 BF 의 크기가 줄어든다 .

메모리 제한을 두지 않고 단순히 Number of BF 를 4 개에서 8 개로 늘리면 원래보다 18% 의 성능이 향상되는 결과를 보인다 . 최신성 정보를 더욱 정밀하게 포착할 수 있었기 때문에 나온 차이이다 .



(a) WDAC (b) DAM

Fig. 14. Changes of Average Hot Ratios over Various Window Sizes in WDAC and Decay Periods in DAM



(a) MHF (b) Change of Total False Identification Rates

Fig. 15. Performance Change over Various Numbers of a Bloom Filter in Our Scheme under RealSSD trace

Personal Opinion

이 논문 아이디어나 디자인에 대한 문제 제기 또는 개선을 위한 본인의 의견
(이 부분은 오류가 있을 수도 있다 . 개인적인 분석이므로)

테스트 성능 결과는 MBF 에 기반한 WDAC 방식이 기존 논문의 알고리즘보다 나은 것으로 사료된다 . 대부분의 상황에서는 이 분석이 옳은 것으로 판명날 것이다 .
그러나 이런 방식으로도 생각해볼 수 있을 것 같다 . 뛰어난 성능을 보이기는 하지만 MBF 방식은 구조적으로 복잡하며 알고리즘을 실행했을 때 수많은 branch 가 생긴다는 것을 알 수 있다 .
반면에 MHF 는 구조적으로 단순하며 알고리즘의 대부분이 loop 로 이루어져 있다 . 예를 들어서 아주 오랜 시간 동안 엄청나게 아주 아주 많은 Write Request 가 집약적으로 발생하는 빅데이터 상황을 가정해보자 . 그런 상황에서는 코드에 대한 Compiler Optimization Level 을 필연적으로 높게 설정할 것이다 . 최적화 관점에서는 branch prediction 보다는 loop optimization 이 더 높은 성능 향상을 보장한다 . 그러므로 위에서 가정한 특수한 상황 등에서는 MHF 에 대한 선호도가 높을 것이다 .

Personal Opinion

추가 사항

*** Loop Optimization : Dynamically Increase Performance

1) code motion/code movement : loop 과 상관없는 코드는 밖으로 빼낸다 .
2) induction variable : 매번 변하는 변수는 최적화 대상이 된다 .
3) loop invariant : 매번 변하는 변수에 영향을 주는 연산 중에서 고정적인 것은 밖으로 빼낸다 .
4) loop unrolling : 동일한 코드를 반복해서 쓰면 jump/test 가 줄어드는 효과
5) loop fusion/loop jamming : 여러개의 (또는 nested) loop 를 합쳐서 하나로 만든다 .

*** Branch Prediction : could be helpful but maybe not

우선 여러 개의 branch 중에서 하나가 True 라고 가정하고 이후의 명령을 미리 실행한다
그 branch 가 선택되지 않을 때는 미리 실행한 명령 전부 Flush 하고 다시 되돌아간다
이는 하드웨어가 지원하는 부분

Questions

~~1. 왜 WR Request 가 발생할 때마다 next BF 를 사용했을까 ? 원래 사용하던 BF 를 사용했어야 recency 정보를 알아내는데에 맞는 게 아니었을까 ?~~

1. Figure 는 반드시 페이지의 맨 위로 보내는 걸까 ?