공학석사 학위논문

PCRAM controller의 hardware prefetcher를 위한 data buffer 최적화

Optimization of a data buffer for a hardwired prefetcher in a PCRAM controller buffer

2021 년 02 월

서울대학교 대학원

전기정보 공학부

심 석 보

PCRAM controller의 hardware prefetcher를 위한 data buffer 최적화

Optimization of a data buffer for a hardwired prefetcher in a PCRAM controller buffer

지도 교수 이 혁 재

이 논문을 공학석사 학위논문으로 제출함
2020 년 12 월

서울대학교 대학원
전기 정보 공학부
심 석 보

심석보의 공학석사 학위논문을 인준함
2020 년 12 월

위 원 장 _____ 김 태 환 _____ (인)

부위원장 _____ 이 혁 재 _____ (인)

위 원 _____ 류 수 정 _____ (인)

초 록

본 논문에서는 PCRAM based storage의 cache buffer 성능 향상을 위한 prefetcher 구조에 대한 연구를 수행한다. Hdd, Nand-SSD를 위한 일반적인 software prefetcher가 아닌 PCRAM hard-wired controller를 위한 경량화된 hardware prefetching 구조를 제안함으로서 NVM stroage cache buffer의 최적화를 수행한다.

이 때 hardware prefetcher의 implementation시 단점으로 부각되는 history buffer의 area overhead, prefetching algorithm의 hardware complexity를 개선하기 위해 application id polling과 address boundary detector를 이용한 필터 구성으로 history buffer를 경량화하였다. application id poller는 단위 시점에서의 populated application id를 선정하고 해당 appication id의 cache miss address만을 sequential boundary detector로 인가한다. Sequential boundary detector는 miss address의 sequentiality를 detect하여 history buffer에 기록하고 이를 바탕으로 유형별 prefetch request를 생성한다

Real-life storage workload로 controller의 average latency를 측정하였고, 약 14%의 read latency개선으로 동일 성능 필요 cache buffer size의 50%만을 필요케끔 cache size가 최적화 됨을 확인하였다

주요어 : NVM Storage, PCRAM, Data Buffer, Prefetcher, Latency, Cache Size
학 번 : 2019-25565

목 차

표 목차

그림 목차

제 1 장 서 론

1.1 연구 배경 및 목표

Memory hierarchy에서 storage는 main memory, 즉 DRAM과의 큰 성능 gap을 지니고 있다. NAND-SSD의 등장 이후 DRAM-HDD로 이어지던 Memory hierarchy는 DRAM-SSD-HDD로 개편되어 전체 system의 큰 성능 개선을 이루었지만 NAND-SSD는 msec 수준의 latency를 지니고 있어 DRAM 대비 여전히 큰 성능 gap을 보인다. 비휘발성(non-volatile) new memory로서 NAND보다 60x이상 빠르고 신뢰성이 좋은 Phase Change DRAM 은 뛰어난 성능과 고직접이 가능한 구조적 장점으로 DRAM과 SSD 사이의 성능 차이를 완화해줄 performance gap filler로서 각광받고 있다.[1] 그러나 여전히 DRAM 대비 상대적으로 느린 read latency 및 write latency로 인해 fast buffer cache가 필요하며 적정size의 cache를 적용시 약 20%의 성능개선을 보이는 것으로 연구되고 있다.[2] HDD, SSD 등 일반적인 storage는 latency gap을 완화해줄 buffer cache외에도 prefetching unit을 추가로 지닌다. HDD는 플래터간 이동을 최소화하여 pick-up의 mechanical 신뢰성을 최대한 보장해주기 위한 목적으로, SSD는 buffer cache size를 최적화하여 latency를 개선하기 위해 prefetcher를 구비한다. HDD, SSD 등 coventional storage에 사용되는 prefetch scheme은 모두 software 기반으로 firmware에 기반한 storage controller unit과 같이 구동된다.

한편 Intel사의 Optane으로 대표되는 PCRAM based storage의 또다른 특징은 hard-wired controller로 구성되어있단 점이다.

J.Yang et al.[3][4]은 2018년 그리고 2020년 2차례의 Optane based SSD storage의 성능평가를 통해 Optand SSD가 NAND SSD대비 high-pressure I/O workload 동작중 매우 낮은 I/O access latency를 보임을 확인하였고 이는 Firmware based controller로 인한 성능 overhead 부담을 최소화 하기 위함임을 분석하였다.

즉, performance 지향의 설계로 access latency variation을 최소화하기 위해 PCRAM controller는 SSD와 달리 hardware only write/read path를 갖는다. 따라서 PCRAM based storage에 fast data cache buffer를 최적화하기 위해서는 software가 아닌 hardware prefetcher가 필요하다.

이 논문에서는 hard-wired PCRAM controller를 위한 hardware prefetcher scheme을 제안하고, hardware prefetcher의 단점인 large history buffer의 area overhead와 prefetching algorithm complexity를 개선하기 위한 방법으로 application id poller와 address boundary detector 방식의 pattern recognition 방식을 제안한다. 이를 통해 counter방식의 small history buffer를 구성함으로서 경량화한 효율적인 hardware-prefetcher 구성을 제안한다

1.2 논문의 구성

본 논문의 구성은 다음과 같다. 2장에서는 prefetching method의 이론과 hardware & software prefetcher에 대해 알아본다. 또한 NVM을 위한 prefetching algorithm에 대해서도 알아본다. 3장에서는 NVM storage 방식의 문제점과 이를 해결하기 위한 hardware prefetcher 구조에 대해 제안한다. 4장에서는 구현 방법과 실험 결과에 대해 확인하고 5장에서 결론을 맺는다.

제 2 장 관련 연구

2.1 Prefetching

Prefetching은 lower level memory의 high latency를 개선하기 위해 실제 request가 이뤄지기 전 data를 미리 fetch시키는 것을 의미한다. 이를 통해 cache의 miss rate을 낮추어 system의 전체 성능을 향상시킨다. Prefetching은 prefetch instruction을 주는 주체에 따라 크게 hardware prefetching과 software prefetchig으로 나뉘며 input pattern을 monitoring하여 pattern을 recognition하고 고유의 prefetching algorithm으로 prefetch instruction을 generation 하는 것은 동일하다

2.1.1 Hardware Prefetching

Hardware prefetching기술은 multi-level cache를 중심으로 발전되어 왔다. hardware가 processor의 access를 모니터링하여 pattern과 stride를 찾아내고 prefetch address를 자동으로 생성한다. Modern hardware prefetching의 특징은 global history buffer[5] 라 불리는 pattern recognition을 위한 별도의 저장공간을 두고 prediction을 위한 prefetching algorithm을 연산한다. 또한 buffer

cache의 cache replacement policy와의 pollution을 방지하기 위해 prefetching된 instruction을 흔히 stream buffer[6]라 불리는 별도의 공간에 저장하여 둔다. 그러나 대부분의 hardware prefetching method는 높은 prefetching hit ratio를 얻기위해 지나치게 큰 history buffer 공간과 prefetching algorithm의 hardware complexity로 인해 실제 implementation에 부담이 되는 경우가 많은 단점을 지닌다.

그림1은 hareware prefetcher의 position을 보여주는 대표적인 형태이며 Cache pollution을 막기위해 LLM와 cache사이에 병렬로 위치한 hardware prefetcher의 모습을 보여준다.

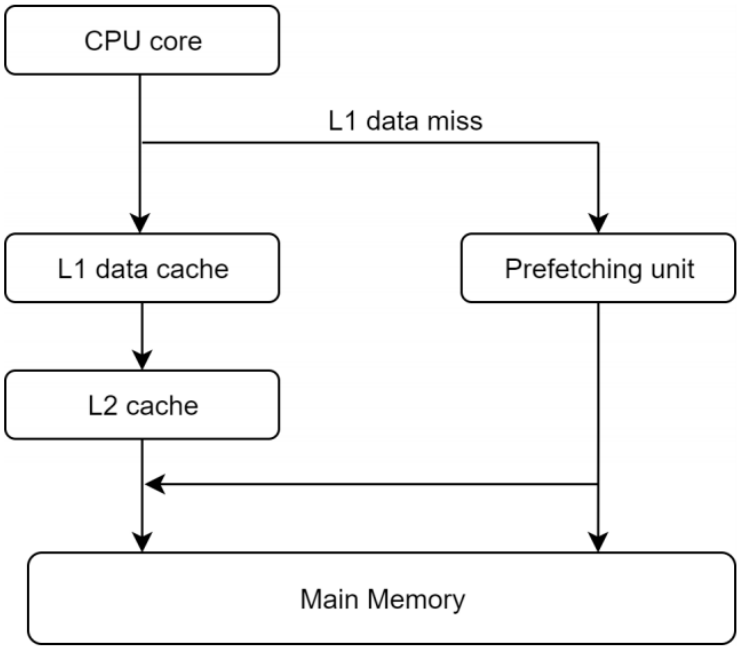

그림 1. Hardware prefetcher in memory subsystem

B.Tishuk, S.Bykovskii

2.1.2 Prefetching for storage

HDD, SSD등의 storage도 고유의 prefetching scheme을 갖는다. Storage에 사용되는 prefetching은 software 방식을 사용하는데, 이는 패턴 인식을 위해 매우 큰 공간이 필요하고 msec order의 media latency에 적합한 사유에 기인한다. Nand-SSD는 신뢰성 확보를 위해 firmware기반의 memory controller를 가지고 있어 software prefetching은 Nand-SSD를 중심으로 발전되어 왔다. Modern Nand-SSD prefetching은 user-space에 pattern recognition을 저장하고 다수개의 state machine으로 application마다의 prefetching algorithm을 구현하여 각 application에 최적화된 prefetching을 수행한다.[7] 그림2는 SSD software prefetcher의 대표논문인 Flashy의 Architecture를 나타내었다.

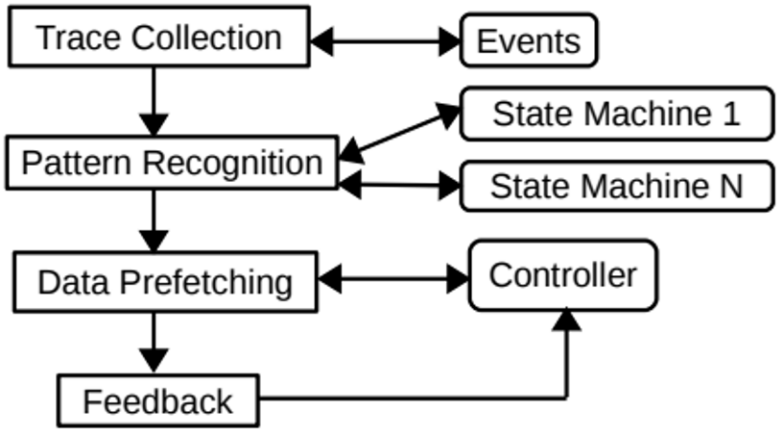

그림 2. Software prefetcher : Flashy Prefetching Architecture

2.2 Previous prefetching work

본 장에서는 storage prefetcher의 대표적 연구인 ReadAhead[8]와 NVM storage prefetcher인 Speculative Paging for Future NVM Storage @SPAN[9]을 알아본다.

2.1.2 ReadAhead

Storage의 prefetching 연구에 자주 인용되는 Readahead framwork에서는 sequential pattern을 category화하여 OS level에 효율적인 prefetching 방법론을 제시하였다. Cache miss된 page의 offset을 분석하여 miss page의 classification을 수행하여 그 data를 기반으로 prediction을 수행한다. Sequentiality Criteria라 불리는 3가지의 category (trivial, unaligned, retried) 로 sequential 패턴을 분류하여 pattern recognition을 하였지만 modern storage input은 제시된 3가지의 category로 분류되기엔 복잡도가 훨씬 더 상승하였다.

그림3은 Readahead에서 제시된 Sequential patttern의 category를 나타낸다. Controller의 최소 transaction 단위로 stride가 지속되는 trivai, 일정한 time interval과 stride간격을 갖는 unaligned, negative stride를 지속적으로 지니는 retried가 그것이다. 본 논문의 제안 방식에서는 확장된 sequeality category를 통해 readahead category 방식을 개선한다.

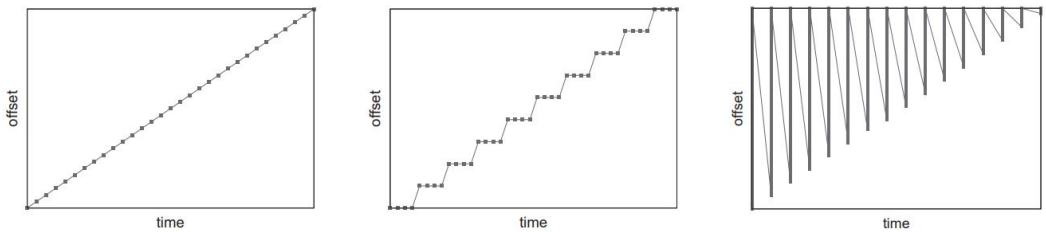

그림 3. Readahead : The trivial / unaligned / retried sequential reads

2.1.2 SPAN

Conventional software prefetching method가 NVM storage에 잘 동작하지 않는 점을 지적하며 Viacheslav외 는 SPAN 이란 논문에서 NVM Storage를 위한 software only prefetching scheme을 제안하였다. 이는 앞서 지적하였던. HDD,SSD 기반으로 발전해 온 prefetching method가 parallelism이 극대화된 방식으로 동작하는 NVM과 부합하지 않기 때문으로 분석하였다. HDD는 플래터 control motor의 신뢰성이 생명이므로 인접 address의 data를 한번에 많이 가져오는 형태의 prefetching method를 지니고 있고 , NAND의 경우는 block 단위 erase동작의 횟수를 최소화하여 신뢰성을 높이기 위한 prefetching을 구사하게끔 최적화 되어 왔다. 이에 반해 PCRAM은 byte addressable 특징을 지니고 있으므로 DRAM과 같이 Random access memory로서 장점을 지니고 있으므로 기존의 prefetching method를 그대로 NVM에 적용시 효율이 떨어지는 이유가 분석되었다.

SPAN에서는 pattern recognition은 software 방식의 prefething method를 이용하지만 prefetch request를 generation하고 저장하는 방식은 hardware의 그것을 따른다. 즉 prefetch request는 별도의 secondary queue에 저장되고 mux되는 모습을 보인다. 그림4은

SPAN의 Block diagram을 나타낸다.

SPAN은 모든 OS의 page fault에 대해 application id별로 stride를 계산하여 signature table (ST)를 update하고 이 data의 delta pattern을 Pattern Table(PT)에 기록시켜 둔다. ST에는 data의 유효성을 위해 과거 4회 까지의 signature를 기록해두고 이 정보의 delta값들은 모두 PT에 기록된다. 단, ST와 PT의 data가 비대해질것을 우려해 ST는 압축 알고리즘을, PT는 hashing 알고리즘이 사용되어 size를 줄이려 하였다. SPAN은 per process id, per page 연산 방식을 통해 NVM의 특징에 부합하는 prefetching을 algorithm을 제시하였고 약 18%의 running time 개선을 이루었다고 한다.

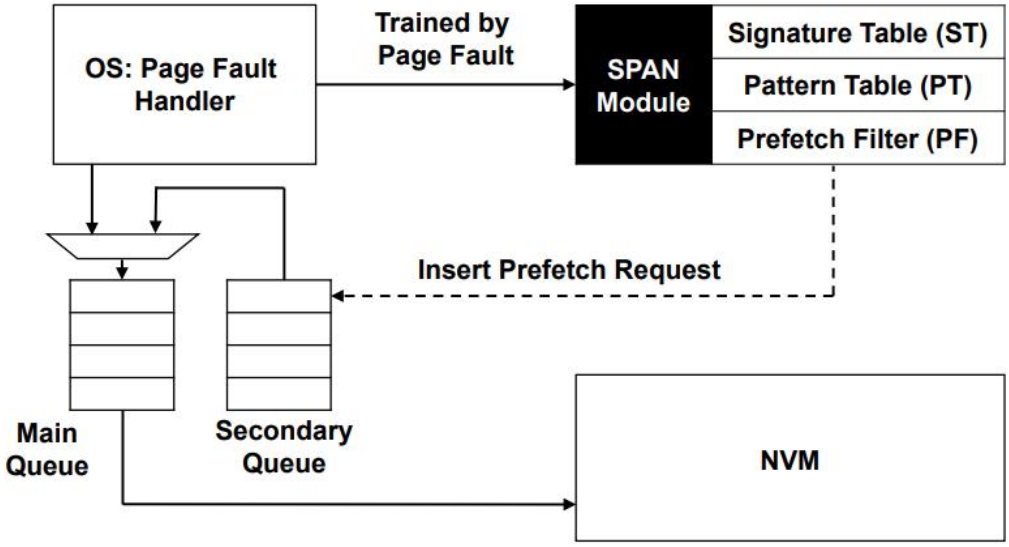

그림 4. SPAN Design Overview

제 3 장 Application polling과 Sequential boundary detector를 이용한 hardware prefetching 방식 제안

3.1 기존 연구의 문제점

이번 장에서는 기존 연구들을 다시 한 번 살펴보고 문제점을 되짚어 본 후 이를 개선하기 위한 방식을 제안하고자 한다.

2장에서 살펴본 SPAN은 기존 storage를 위한 software prefetcher의 단점인 pattern 저장 area overhead 극복을 위해 hashing과 compression 기법을 사용하여 pattern table과 signature table을 생성하였다. 그러나 여전히 모든 page fault 에 대해 table을 update하므로 높은 prefetching hit rate를 얻으려면 큰 size의 table이 필요하고 또한 computing calculation cost가 매우 크다. 무엇보다도 storage를 위한 conventional software prefetching method는 hard-wired controller에 적합하지 않다. prefetching method를 통해 latency를 개선하고 buffer cache의 size를 최적화 하는 것보다 prefetching hardware구성시의 overhead가 상대적으로 너무 크기 때문이다.

prefething의 목적이 cache size를 최적화시키는 것에 있다고 보면 complexity가 높은 prefetching algorithm을 hardware에 implementation 시키는 것은 큰 부담이기 때문이다.

본 논문에서는 이러한 문제점을 보완하여 경량화한 pattern recognition method와 focusing된 prefetching alogorithm으로

PCRAM hard-wired controller에 적합한 hardware prefetching method를 제안하고자 한다

3.2 관측

그림 5는 real workload인 umass-storage workload [10] 중 하나인 finacial1의 패턴을 plot한 것이다. x축은 read request의 순서이고 y축은 logical address이다. 색깔은 ASU (application specific unit)으로 storage로 input되는 application id를 의미한다. 단위 시간 동안 십수개의 application이 random mix되어 들어오지만 x축을 일정 단위로 quantization할 경우 실제로 populated id는 3-4개 수준으로 dominant한 특성을 보이는 것을 볼 수 있다. 즉, 전체 application을 모두 calculation하지 않고 수 개의 populated application을 선택하여 prefetching하면 성능개선 폭은 다소 줄어들더라도 pattern recognition을 위한 area overhead와 calculation overhead를 모두 줄일 수 있다.

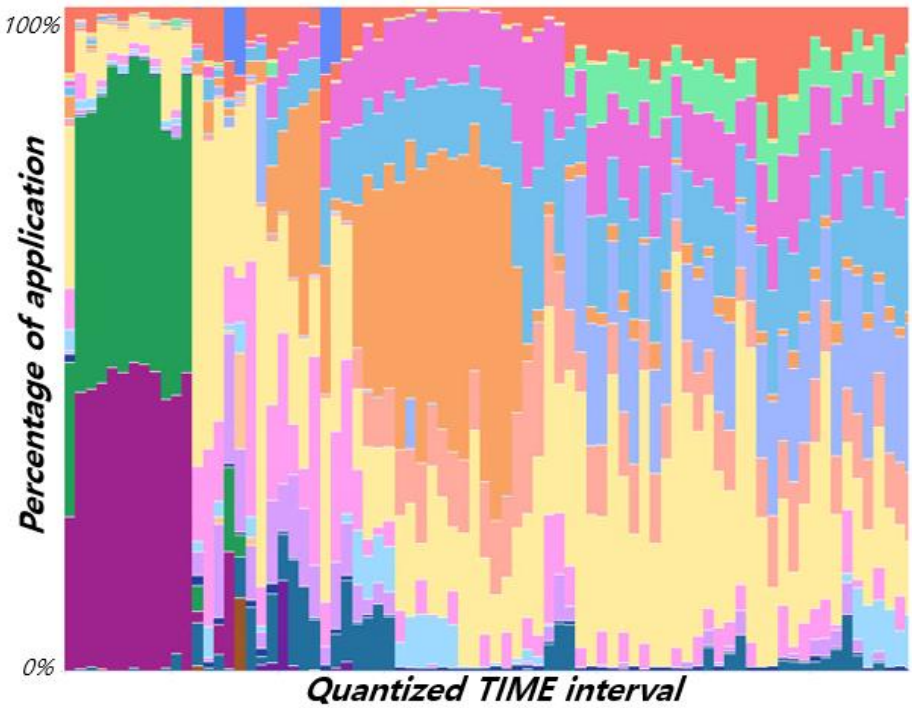

그림 5. Applicaion ID percentage plot per time interval

그림 6는 그림5 중 단일 application 단위로 address stream을 plot한 그림이다. Sequential pattern과 temporal & spatial random 패턴이 모두 mix되어 인가됨을 확인할 수 있고 sequential 패턴이라 하더라도 다양한 time interval과 stride과 뒤섞여서 인가됨을 볼 수 있다. 즉, 거시적으로는 sequential 패턴이더라도 미시적으로는 random 패턴으로 보여질 수 있다. 기존의 prefetching method들은 sequential 패턴이라도 복잡한 input의 반복되는 rule을 파악하기 위해 hardware complexity가 매우 높아졌다. 그러나 sequential pattern의 경우 random에 가까운 다양한 stride를 보이더라도 일정 크기 이상을 넘지않는 것을 확인할 수 있다. 다만 단일 application만 놓고 보아도 sequential 패턴과 random address가 mix되어 있으므로 random pattern의 filtering 필요성도 동시에 관측된다. 상기의 관측으로부터

populated application만을 선택하여 일정 boundary내의 cache miss address (그중에서 특히 sequential 패턴만을)를 집중적으로 prefetching한다면 history buffer의 area overhead를 줄임과 동시에 prefetching algorithm의 complexity도 낮출 수 있음을 알 수 있다.

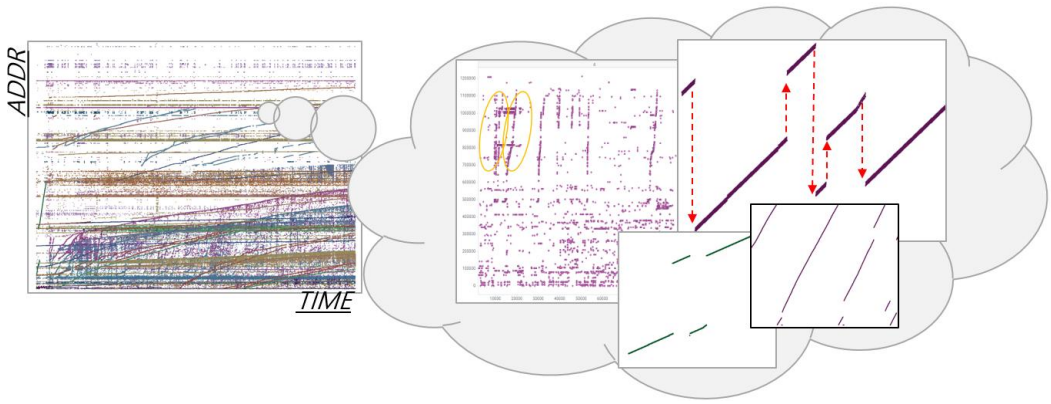

그림 6. 단일 Application Address vs Time plot

3.3 Prefetching method 제안

그림7은 제안된 hardware prefetcher의 block diagram을 간단히 도시하였다. Conventional buffer cache외 hardware prefetcher의 기본 구성인 stream match와 stream buffer외 Application ID polling machien과 Sequential Boundary Detector 그리고 Hitory buffer가 구성되었다. Prefetch generation engine과 prefetch request queue및 mux unit의 이해의 편의상 생략되었다.

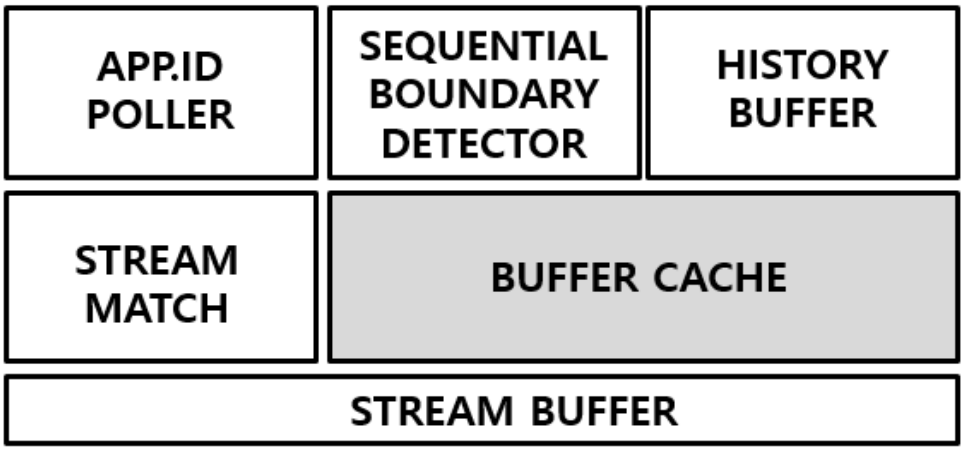

그림 7. Proposed Prefetching architecture

모든 Read request는 크게 2가지의 process를 따른다. 첫번째로 stream buffer tag match stage, 두번째로 data buffer tag match stage이다. prefetching된 read는 별도의 stream buffer에 저장되는데 인가된 read가 stream buffer에 있을 경우 prefetch hit status로 해당 read는 direct response된다. 첫번째 tag match stage에서 miss될 경우 , prefetch miss 로 두번째 data buffer tag match stage로 넘어간다. 이는 통상의 cache match stage로 hit될 경우 direct response되며 miss될 경우 media로 read request가 전달된다. 이때 모든 read request는 application id polling machine으로 인가되어 populated application id를 update하게 되며, 이 정보는 sequential boundary detector (SBD) 와 hitory buffer로 전달된다. data buffer miss시의 cache miss address는 SBD로 향한후 boundary를 check하고 in-boundary일 경우에만 history buffer에 update된다.

Applicaion id poller와 SBD는 2단계의 filter 역할을 수행하므로 patttern recognition을 위한 history buffer 의 size를 줄이는데

일조한다. 누적된 history buffer정보로부터 prefetching algorithm이 수행되고 생성된 prefetching request는 media로 인가되며 일정 latency후 전달된 read정보는 stream buffer에 저장되어 prefetch hit/miss를 판가름하게 된다.

3.3.1 Application ID polling

앞선 관측으로 일정 time interval로 storage의 input을 quantization할 경우 수 개의 populated id로 input id를 선정할 수 있음을 확인하였다. Read의 application id 정보는 id polling machine의 queue로 인가되고 미리 mapping된 application id map의 counter 값을 update한다. 이때 queue의 depth는 설정된 time interval에 해당된다. 새로 들어온 app.id는 해당 app.id map의 count값을 1만큼 증가시키고 제일 오래된 queue의 마지막 id는 빠져나가게 되며 해당 app.id map의 count값을 1만큼 감소시킨다. 그림8-1은 application id poller의 동작을 도시화 한것으로 1의 id가 인가되었을때 app.id map의 count정보가 갱신되는 모습을 보여준다 그림8에서 선정된 populated id는 3,1,0 순으로 점유율이 높은 것을 알 수 있다. update된 populated id 정보는 sequential boundary detector로 전달된다.

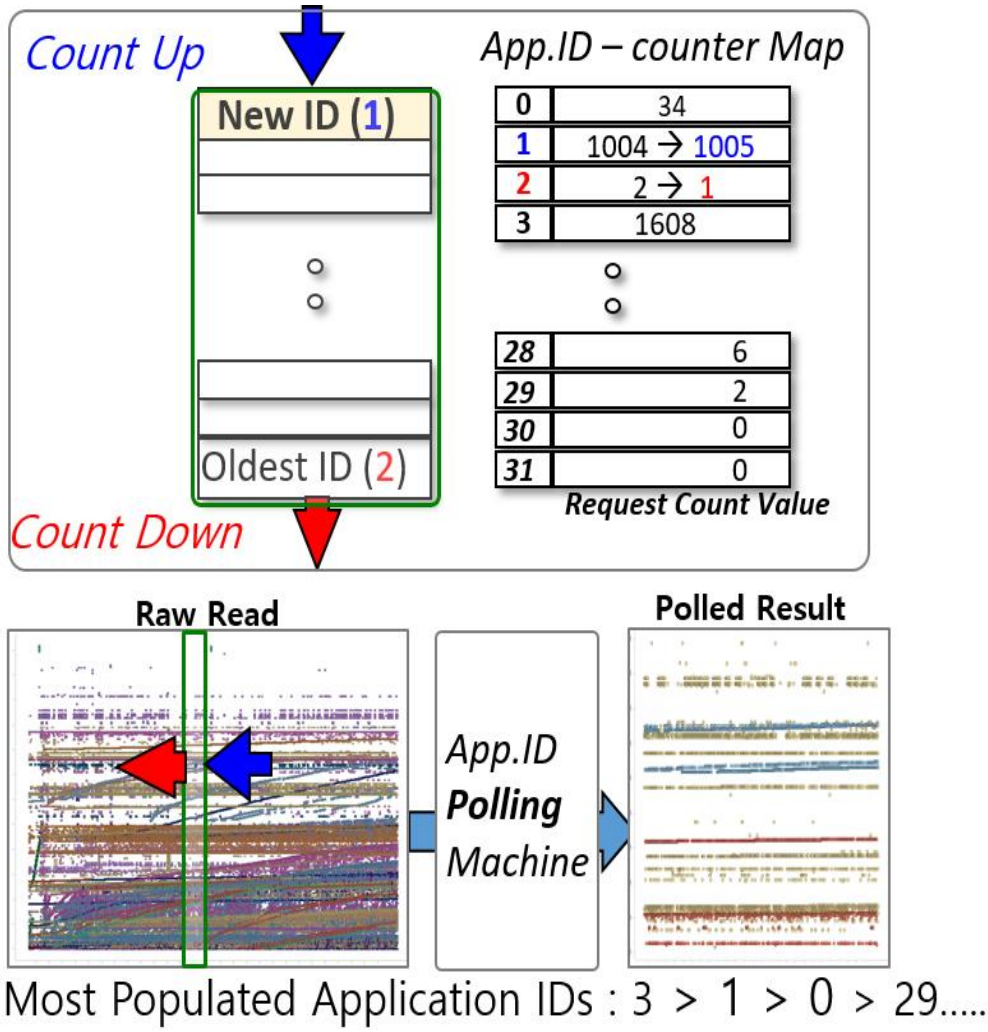

그림 8-1. Application id polling machine

그림 8-2은 상기 기술된 application polling machine을 사용해 storage workload 샘플의 상위 3개의 application을 polling한 모습을 도시하였다. vertical축은 logical address를, horizontal축은 time축을 표현하였다. 일정 interval에 총 20여개의 application이 random input되고 있고 최상단 trellis에 dominant application을 도시하였다. 1, 10,12,6,19,20에 해당하는 populated application이 다른 다수의

application과 함께 input되고 있다. 2nd trellis에는 Populated application sector1의 최다빈도의 application인 1, 10, 19가 위치되고 sector2, sector3에 차순위 application이 ranking되었다. 이때 sector 2와 sector3의 application을 보면 sector 와 overlay되며 19번에서 20번으로 populated application이 변화하는 것을 볼 수 있으며 sector3의 경우 굉장히 짧은 간격으로 application이 변화함을 볼 수 있다. 즉 3~4개의 populated application에만 집중하여도 전체 application request의 상당부분을 모니터링이 가능함을 짐작할 수 있다.

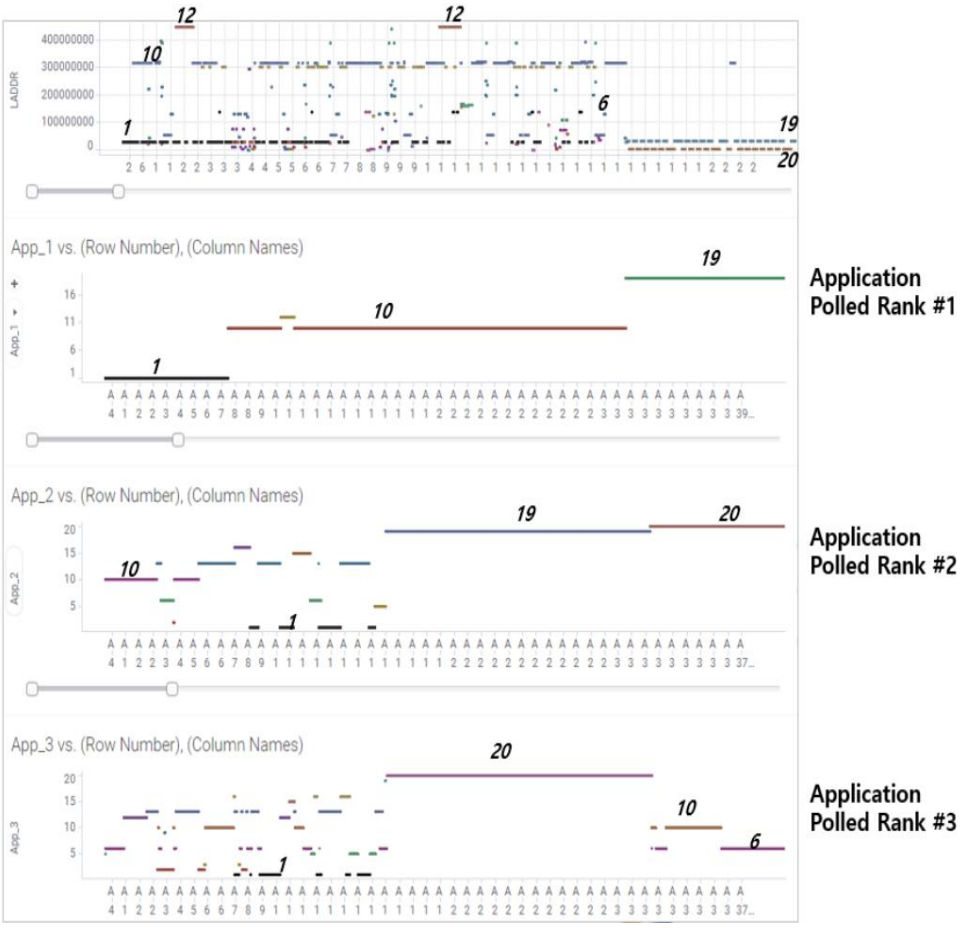

그림 8-2. Application id polled result

그림 8-3은 UMASS Storage Workload 대표 3개 workload에 대해 polling window size를 변화시켜 가며 최적의 polling window size를 평가해본 결과이다.

이때 사용한 대표 Workload financial1, financial2, websearch는 random이 특히 강하며 다양한 application이 섞여서 들어오므로 application polling을 평가하기에 적합한 workload이다

좌측 그림에서는 most populated application의 polling window size에 따른 최다 application number의 변화 count 값을 도시한 것으로 약 2000개 정도의 request count range를 polling할시 최다 application의 변화 빈도가 saturation됨을 볼 수 있다. window size가 너무 작을 경우 최다 빈도의 application number은 지나치게 자주 바뀔 것이고 적정 window로 구성될 시 satuartion을 보일 것이다. 3개의 application candidate을 구성한다고 가정시 마지막 populated application id는 최다빈도 출현 application이 saturation될때 오히려 반대의 satuartion을 보일 것이다. 8-3의 우측 그림에서는 2000개의 count결과에서 financial1의 결과가 최대변곡을 보이고 이후부터 줄어드는 모습을 볼 수 있다. financial1보다 random성이 덜한 financial2의 경우 최다빈도 application은 2000에서 수렴되지만 최빈 빈도를 표현한 우측 그림에서는 변화 빈도가 오히려 줄어드는 모습을 보인다. application숫자가 4~5개 정도로 수개밖에 해당하지 않는 websearch의 경우 polling window range에 영향을 크게 받지 않음을 볼 수 있다.

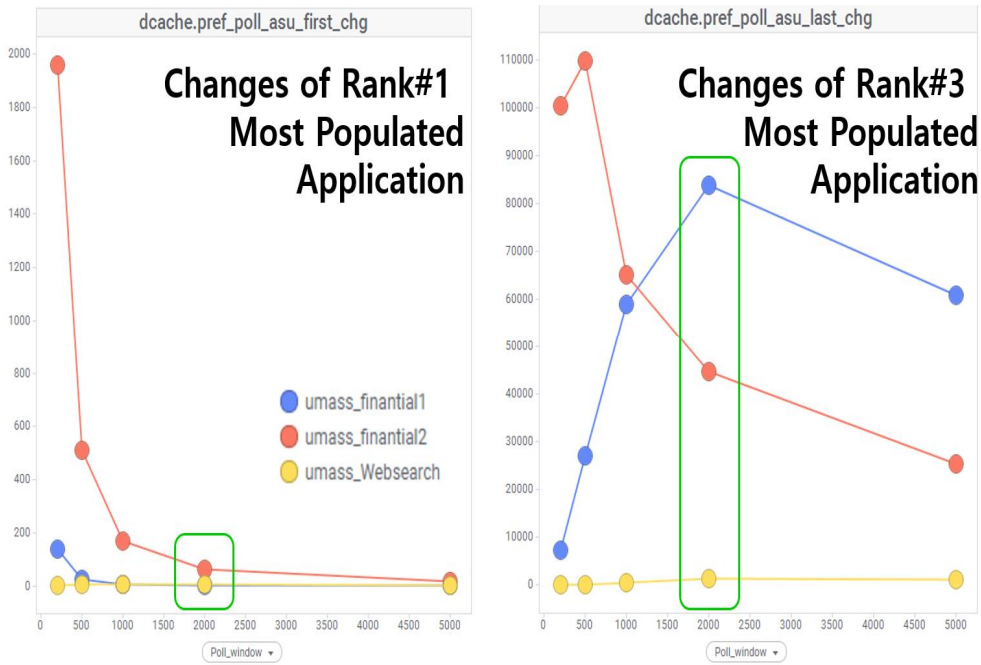

그림 8-3. Polling window size evaluation result

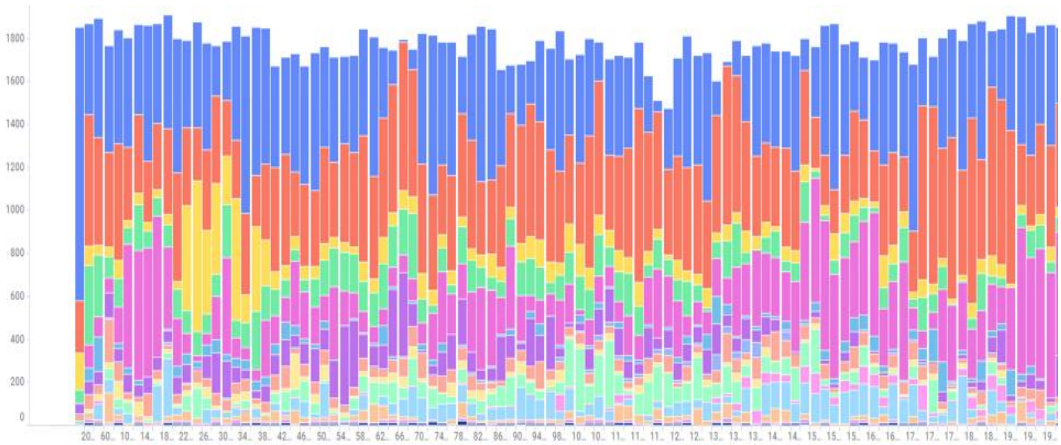

그림 8-4. Applicaion # plot via application count window of 2000

그림 8-4는 financial1과 상이한 결과를 보인 financial2의 request를 2000ea의 polling window로 나누어 도식한 결과이다. 최빈 appplication이 숫자대비 잔여 application의 변화가 크지 않으므로 상대적으로 random성이 떨어지는 특성을 확인할 수 있다.

상기 실험으로 부터 Polling window range는 input workload의 특성을 모니터링후 가변하는 것이 최적임을 확인할 수 있다. 본 논문의 실험에서는 storage workload 20개에서 최적 성능을 보이는 average window값을 parameter로 설정하여 실험을 진행하였다

3.3.2 Sequential Boundary Detector

그림9는 sequentail boundary detector의 operation flow chart를, 그림10은 sequential boundary detector의 동작을 시간에 따른 address로 plot하여 표시한 것이다. input된 cache miss read address는 application id poller에서 선정한 populated id 여부를 먼저 판단받는다. 해당 populated id일 경우 history buffer에 마지막으로 기억된 last address를 기준으로 설정된 up-boundary와 dn-boundary사이에 input address가 위치하는지 판단한다. Input address가 설정된 boundary내에 있을 경우 history buffer의 last address를 현재의 input address로 update하고 이전 last address와의 차이를 계산해 stride도 update한다. 이때 stride의 값에 따라 sequential pattern의 type 또한 history buffer에 기록된다. 만약 input address가 설정된 boundary 내에 위치하지 않을경우 boundary-out counter값을 증가시키고 해당 miss address는 disgard된다. Boundary

out counter가 일정 임계를 넘을 경우 history buffer는 flush되며 seed address는 마지막input address로 대치된다.

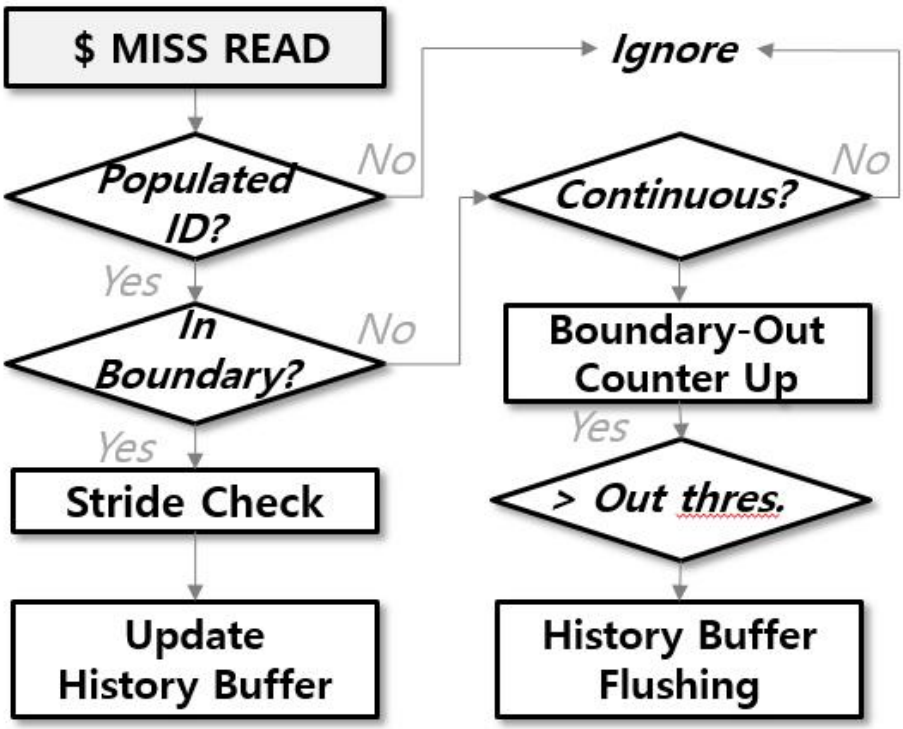

그림 9. Sequential Boundary Detector Operation flow

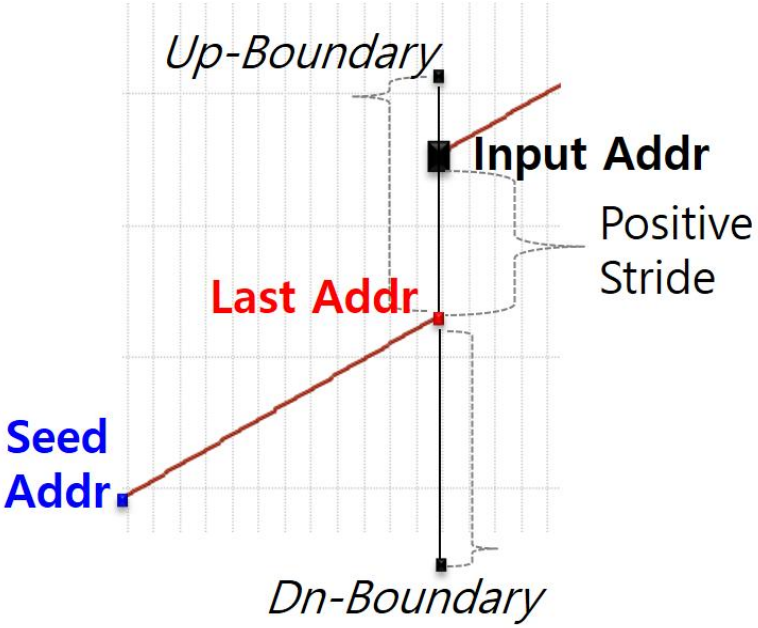

그림 10. SBD: Time vs Address plot

SBD는 일정 boundary를 유지시켜 input address를 필터링함으로서 다변하는 stride를 모두 품을 수 있다. 설정한 boundary를 초과하는 input이 지속된다는 것은 address가 다른 block으로 이동되었거나 sequential 패턴사이에 섞인 random read가 해당하게 된다. 이경우 SBD는 Boundary-Out counter를 구비하여 일정 임계이상 Out-count가 지속될 경우 Seed address를 바꿔줌으로서 천이된 address의 tracking을 지속시켜 줄 수 있도록 update된다.

즉 boundary 임계값은 빈도가 높은 sequnetial pattern의 stride를 대부분 품을 수 있도록 설정하여여만 한다.

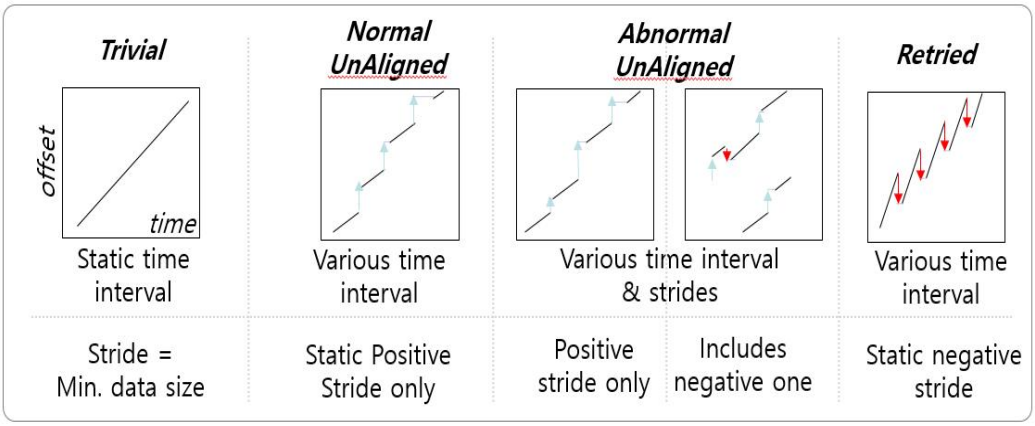

그림 11. SBD: Sequentiality Categorization

그림11은 SBD에서 stride calculation시 사용되는 sequentiality category를 나타낸다. 앞서 살펴본 ReadAhead 방식에서는 3가지의 sequential category를 나누었지만 SBD에서는 이보다 확장된 5가지의 sequentiality category를 가진다. Last stride정보와 현재 input address로부터 계산된 stride로부터 현재 입력된 address의 sequentiality는 trivial, normal unaligned, abnormal unaligned with non-negative stride, with negative stride, retried 5가지로 분류된다. input address의 가변되는 time interval은 application id poll 결과가 유지되는한 SBD에서는 고려할 이유가 없다. 오직 stride만을 계산하게 되며 계산 결과는 history buffer의 각각의 stride counter에 저장된다.

SBD는 이같은 필터링 동작을 통해 sequential 패턴에 집중하여 prefeching을 위한 recognition 정보를 history buffer에 기록할 수 있다.

SBD의 성능과 hardware 구성 area overhead의 tradeoff에 영향을 주는 가장 중요한 parameter는 boundary의 크기이다. request i/o size가 64KB를 넘어갈 정도로 크고 반복되는 address의 stride크기가

크지 않을 경우 적정 수준의 up & dn boundary의 설정으로 대부분의 request의 sequentiality 결과를 Trivial과 Normal unaligned pattern에 묶어 둘 수 있어 prefetching alorithm의 복잡도를 크게 낮출 수 있다. 반면 큰 boundary를 설정할 경우 abnormal unaligned pattern의 detection 빈도가 높아지며 이를 prefetching하기 위한 algorithm의 복잡도도 상승하게 된다. 본 논문의 prefetcher에서는 hardware 복잡도를 최소한으로 낮추기 위해 negative stride를 가지는 abnormal unaligned pattern은 오히려 prefetching filter out의 요소로 보았다. 즉, abnormal unaligned pattern에서 기반한 prefetching이 높은 hit-rate를 갖기 위해서는 보다 복잡한 correlation 기반의 prefetching algorithm을 구비해야하며 이는 경량화한 prefetching scheme구성 목적과는 반하는 것으로 판단하였기 때문이다. 본 논문의 실험에서는 일반적인 memory controller의 request queue size를 가정하고 64B 단위로 split된 request parsing이 전제됨에 기반해 trivial 과 normal unaligned pattern에 집중한 prefetching을 구사함으로서 algorithm hardware의 경제성을 우선시한 실험을 진행하였다. pattern의 SBD result monitoring결과에 기반한 boundary size을 adaptive 가변할 수 있는 prefetcher 구성이 further work으로 고려할 수 있다.

3.3.3 History Buffer

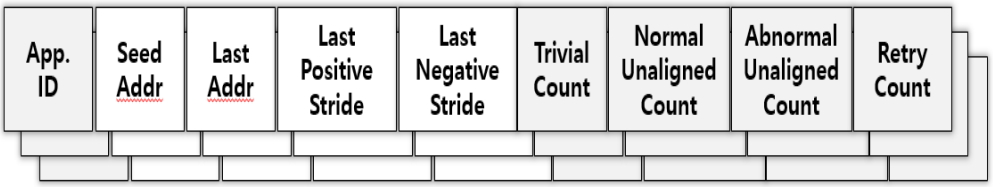

그림 12. History buffer data field

그림12는 history buffer의 data table 구조를 나타내고 있다. Polling의 결과인 application id, seed address, last address, last positive stride, last negative stride로 총 5개의 register와 4개의 counter로 구성된 이 history buffer는 address 자체를 모두 기록하지 않고 오직 seed address와 last address만을 기록한다. 대신 last address와 input address로부터 계산된 stride값 그 자체와 분류된 sequential category의 counting value만을 기록함으로서 history buffer의 area overhead를 현저히 낮출 수 있다. history buffer는 application id poller에서 설정한 id의 개수만큼을 구성한다. 예를 들어 총 3개의 populated application id만을 monitoring한다면 history buffer table은 3개만 필요하다. Monitoring되는 address의 SBD boundary밖으로의 천이가 일어나면 history buffer는 flushing되며 해당 flushing기준시의 input address로 seed address를 update한다.

각각의 counter value가 일정 threshold를 넘을 경우 prefetch request가 생성되며 prefetch request queue에 저장하게 되고 Data buffer module이 idle일때 media read를 수행하게 된다. 즉 normal cache miss request가 우선권을 지니게 되며 prefetch request는 그 다음 순위의 priority를 갖는다.

3.3.1 Overall implementation

그림13은 앞서 설명한 Application ID polling, Sequential boundary detector 및 History buffer로 pattern recognition부를 구성한 prefetching 방식의 overall implementation을 도시하였다. Conventional data cache를 중심으로 prefetching unit들은 일반적인 hardware prefetching의 구성으로 따라 병렬 구성되었다. Data cache의 전단과 후단에 위치한 mux unit으로 request와 response는 prefetch request와 normal request의 구별을 위해 mux 처리된다.

Data Cache와의 replacement policy pollution을 방지하기 위해 Tag matching은 2단계로 분류되며 첫번째 prefetch hit 여부 판단을 위한 Stream Search 부와 Data Buffer cache부의 conventional tag match stage 가 그 두번째이다. Application ID poller에서 설정한 populated application id candidate 갯수에 따라 동일한 갯수의 histroy buffer table이 구비된다. application id polling 정보는 갱신될 시 SBD와 Stream Search engine, History buffer 3개의 block에 각각 전달되며 각 block들은 input의 application id가 populated id와 동일할 시에만 동작하여 total miss read count number대비 prefetching calculation cost를 낮춰주게 된다.

history buffer의 update값에 기반한 prefetching algorithm을 구현한 prefetch generator와 generation된 prefetch request를 저장하는 prefetch queue가 history buffer와 sequential 구성되었다. prefetch queue의 depth는 conventional hardware prefetcher에서도 지적되었듯이 depth의 크기와 성능과는 무관한 결과를 보였다.

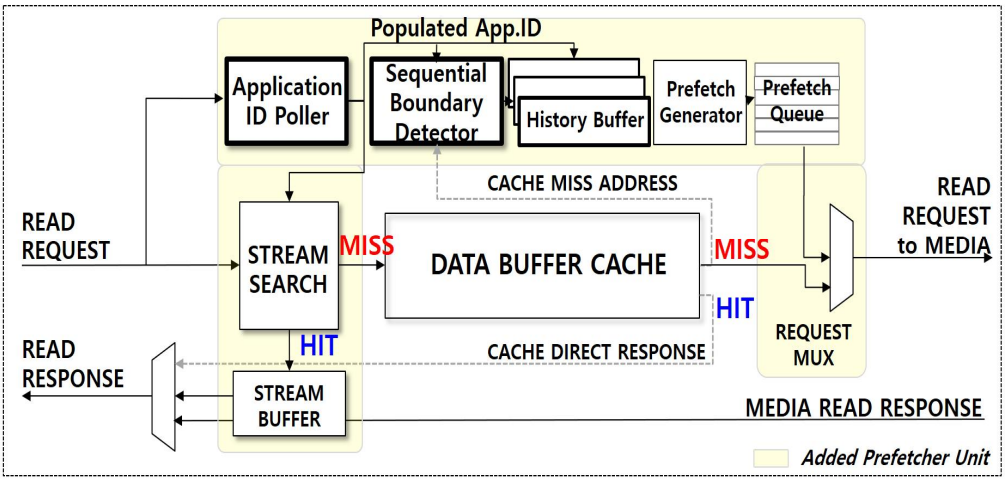

그림 13. Proposed Prefetching Architecture

Prefetcher의 성능을 결정하는 또 다른 요소는 prefetching request를 generation하는 시점을 결정하는 일이다. prefetcher와 함께 구성된 data cache는 stream search를 위한 tag matching sequence와 일반cache의 tag matching의 2가지 delay를 가진다. 즉 최소의 조건으로 tag matching delay + prefetching detecting & generation delay + media scheduling latencdy + media access delay (read latency) 의 총합보다 상회하는 scheduling delay를 지닌 address를 prefetching하여야 올바른 pretching sequence를 지님으로서 hit-rate를 높일 수 있다. 이를 위해 PCRAM controller의 request queue의 average popping out delay와 tag miss delay 를 고려하여 prefetching address stride를 계산하였고 data cache의 output port가 idle하고 prefetch queue가 비어 있지 않으면

prefetching request가 이루어 지도록 구성하였다. 그러나 이런 고려에도 불구하고 input logical address의 physical mapping 결과에 따라 bank parallelism이 효율적이지 못한 경우 , 즉 bank의 idle을 지나치게 기다리는 경우 media scheduling의 변폭은 작지 않았고 이경우 prefetching의 timeliness는 깨어지는 경우가 적지 않았으므로 stream buffer의 구성은 2가지로 구성하였다. 즉 prefetching request는 data cache output port가 idle이면 바로 진행되며 이때의 address는 곧바로 stream buffer에 기입하고 valid와 standby란 2개의 data field를 구성하였다. prefething request가 response되기전 input miss address가 stream match단계에 도달하지 않았다면 standby는 false상태를 유지시킨다. 반면 prefetching request가 이루어진 상태에서 input address가 동일한 request가 data cache에 도달하였고 해당 address가 stream buffer에 있을 경우 standby flag를 true로 바꿔줌으로서 아직 미도달된 prefetch request가 있음을 알 수 있게 하였다. 바꿔말해 prefetch request가 timeliness를 유지할 수 있는 range를 늘려줌으로서 정상적인 prefetch hit이외에 기 진행된 prefetch가 다시 중복되지 않도록 filtering역활을 stream buffer가 가질 수 있도록 구성하였다. 본 논문의 실험에서의 prefetching hit ratio는 두가지의 prefetching hit case,

1. Request input after prefetching response (Normal Hit)
2. Request input after prefetching request but not yet responsed (Standby-Hit)

를 모두 합산한 결과를 계산하였다. Prefetchig total hit rate의 합산

결과의 비중은 standby-hit ratio의 비율이 낮을수록 timeliness가 잘 맞는 prefetching이 이루어진다는 반증이며 반대로 standby-hit ratio가 높다는 것은 prefetching이 too late 일어난다는 것을 의미한다 . 본 논문의 실험에서는 구비한 Workload 평가의 standby-hit ratio의 average가 가장 낮은 parameter를 구성하였다. 그렇지만 standby-hit ratio rate을 모니터링하여 adaptive한 prefetching stride를 계산하여 더 높은 normal prefetching hit ratio를 얻는 further work을 구상할 수 있다.

제 4 장 실험 결과 및 분석

4.1 실험 구현 방식

제안된 hardware prefetcher for hardwired pcram controller는 general purpose의 가변 bit_set와 way number를 가지는 set-associative cache와 최신 optance 성능 평가[1]에 기반한 multi-rank, multi-bank 지원의 media scheduler, 그리고 pcram가상 media로 구현되었다. Pcram media의 interface는 1600Mbps의 ddr4-like interface로 가정되었고 density는 512GB, write latency 500ns, read latency 300ns로 가정하였고 이를 위한 timing controller는 FCFS(first-come, fisrst-service) 방식의 bank parallelism을 지원하는 scheduler로 구성되었다. 단 이때 system contoller의 clock당 data 전송 단위는 Optane 측정에 기반해 64B[12]로 설정하였다. 표1는 simulation 구현에 사용된 주요 parameter를 표시하였다.

Input workload로는 storage 성능평가에 자주 인용되는 real life workload인 umass-storage[10]외에 microsoft research center의 msr-cambridge workload[11]를 사용하였다. 표2는 사용된 workload의 패턴이름과 Read/Write ratio, request i/o size를 나타내었다. umass storage의 workload는 다양한 read/write 비율, 다양한 io request size, application id별 뚜렷한 pattern 특징으로 storage 성능평가에 사용하기 적합하다. MSR-Cambridge workload는

64KB까지의 큰 io size등 modern storage io에 가장 근접한 공개 workload이다.

Section	Parameter	Value
Controller	Data Cache	Bit_Sets ~15, Way ~32 Max 256MB
	Scheduling Policy	FCFS : First-Come Fisrt-Service
Media	Density	512GB
	Interface	x8 org, 8-burst DDR4-like, 64B line [12]
	Write Latency	500ns [11]
	Read Latency	300ns [11]

표 1. Controller 구성 Parameters

Category	Pattern	R/W ratio	I/O Size
UMASS	Financial1	24/76	0.5~4k
	Financial2	82/18	0.5~3k
	WebSearch	99/01	8k
MSR-Camb.	msr-hm	44/56	4k
	msr-mds	58/42	4k~64k
	msr-prn	54/46	4k~64k
	msr-prxy	61/39	4k, 8k
	msr-rsrch	16/84	4k
	msr-usr	12/88	4k~64k
	msr-web	79/21	4k~64k

표 2. Input Workload characteristics

4.2 제안 방식에 따른 결과

그림14는 구성된 Prefetcher의 pattern별 hit ratio를 나타내었고 Overall average prefetch hit ratio는 82%를 보인다.

그림 14. Hit Ratio of proposed hareware-prefetcher

Prefetching의 경우 통상 70% under의 hit ratio를 가질 경우 redundant한 prefetch request의 증가로 오히려 system 성능을 degrade시키므로 80~90%에 가까운 hit rate를 보여야만 한다. 패턴별로 확인하면, websearch와 같이 8k이상의 큰 request size를 갖고 sequential read가 많은 패턴은 90%이상의 높은 hit rate을 보이는 반면 read가 많더라도 random pattern이 주를 이루는 financial2 , prxy의 경우 60~70%의 낮은 hit-rate를 보인다. rsrch의 경우 낮은 hit ratio를 보이지만 이는 rsrch의 낮은 Read비율에 기인한다.

그림 14는 Miss Read의 computation ratio를 표현하였다. 모든 miss read의 count를 1로 보았을때 SBD에서의 stride check 횟수를 count하여 비율로 표시하였다.

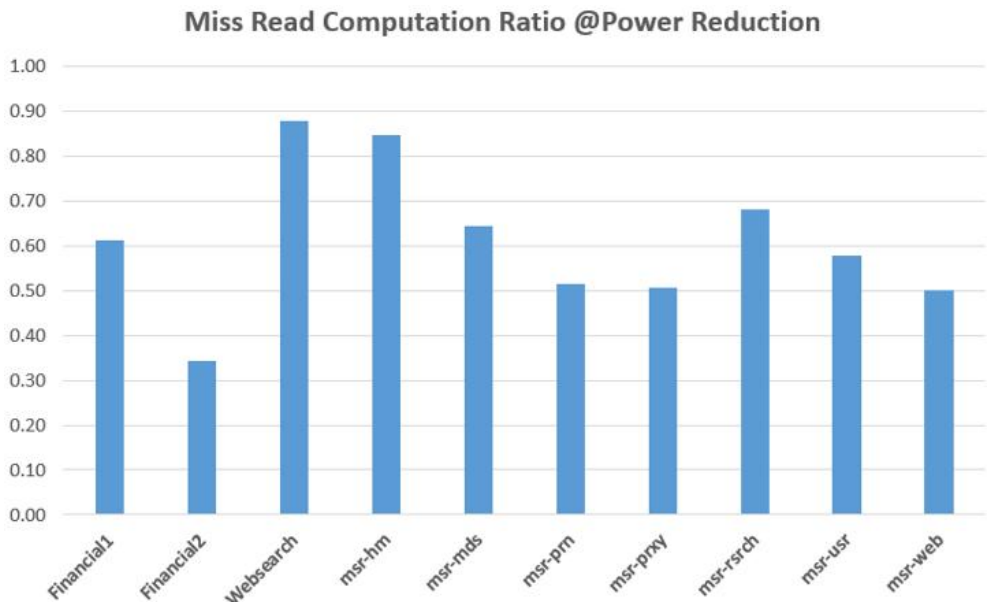

그림 15. Miss Read computation ratio of proposed hareware-prefetcher

Application ID polling과 boundary check라는 2번의 필터작업 이후의 cache miss read에 대해서만 stride check를 하였기에 prefetching을 위한 calculation 횟수는 average 61%로 줄어들었고 이는 곧 prefetching unit의 power consumption이 개선되었음을 의미한다. Financial2의 경우 Random Read pattern이 주를 이루므로 가장 큰 폭의 감소를 보이며 Websearch의 경우 높은 hit-ratio를 보인만큼 가장 적은 Calculation 감소를 보인다

그림15는 PCRAM controller의 overall read latency를 나타내었다. Trace driven simulator의 경우 실제 running time을 계산하기는 어려우므로 prefetcher off 기준의 total read count에서 prefetch hit count를 차감하여 latency 개선값을 계산하였다. Average latency는 total 14%가 개선되었으며 이는 saturated way number (32)를 사용하는 set-associative 기준 50%이하의 cache size를 필요로하는 것과 같다. 그림 16은 Saturated way인 32 way기준 tag set의 bits에 따른 latency 변화를 도시하였다.

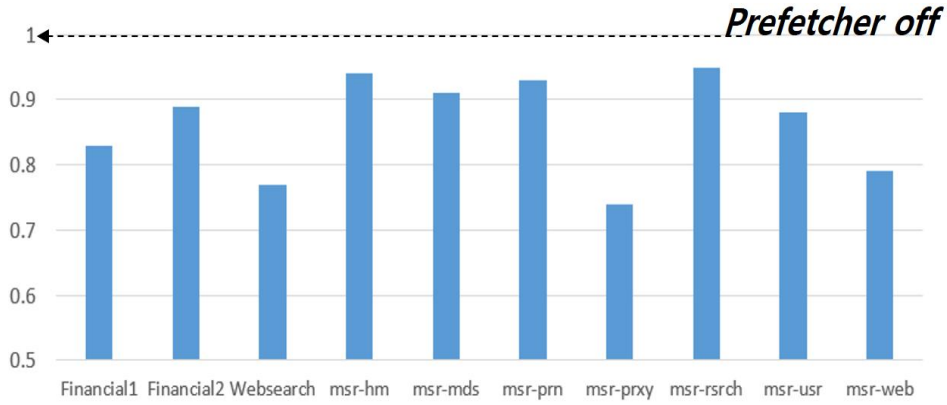

그림 16. Overall Read latecny of proposed hareware-prefetcher

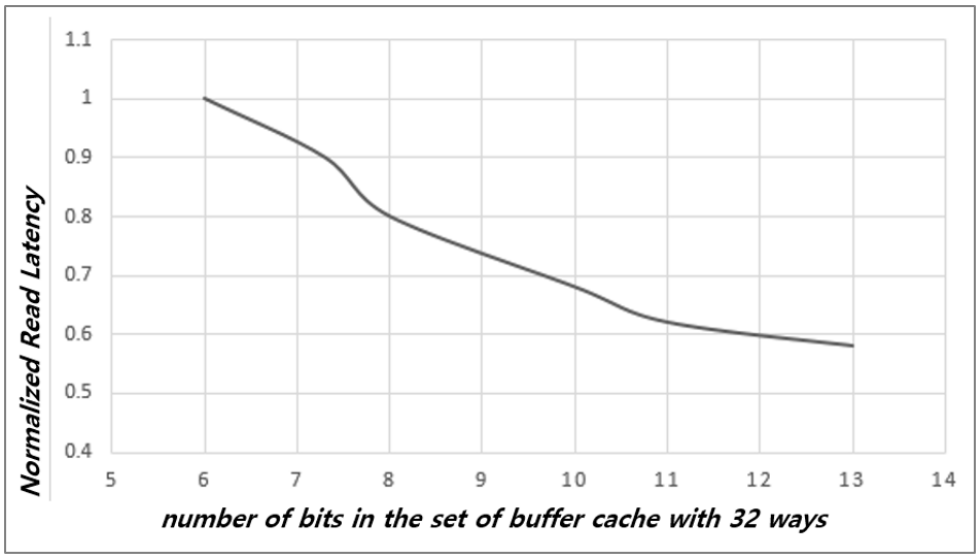

그림 17. Read Latency vs # of bits of cache set with 32 ways

제 5 장 결론

본 논문에서는 PCRAM based storage hardwired controller의 cache buffer size 최적화를 위한 hardware precfetching 방식을 제안하였다. Populated application polling, sequential boundary detecting, 그리고 categorized stride counting방식의 경량화된 history bufffer를 통해 energy efficient하고 area overhead가 적은 hardware prefetching 방식을 통해, 평균 61%의 power를 절약하면서 약 14%의 avreage read latency를 개선하였다. 이는 동일 성능의 cache size대비 1/2 이하의 cache size를 필요로 함으로서 nvm storage를 위한 cache size optimization을 가능하게 한다.

향후 storage의 지속적인 speed 개선시 hardware방식의 prefetching 분야 연구에 적용될 수 있을 것으로 기대한다.

참고 문헌

[1] J Izraelevitz, J Yang. “Basic performance measurements of the intel optane DC persistent memory module” in arXiv, Mar, 2019

[2] K Wu, A Arpaci-Dusseau, “Exploiting Intel Optane SSD for Microsoft SQL Server” in DaMoN'19: Proceedings of the 15th International Workshop on Data Management on New Hardware, July 2019 Article No.15 Pages 1–3

[3] S Kim, JS Yang, “Optimized I/O determinism for emerging NVM-based NVMe SSD in an enterprise system” in DAC '18: Proceedings of the 55th Annual Design Automation Conference , June 2018 Article No. 56 ,Pages 1–6

[4] Frank T. Hady; Annie Foong, “Platform Storage Performance With 3D XPoint Technology” in Proceedings of the IEEE Volume: 105, Issue: 9, Sept. 2017, Pages 1822-1833

[5] K.J. Nesbit; J.E. Smith, “Data Cache Prefetching Using a Global History Buffer” in 10th International Symposium on High Performance Computer Architecture (HPCA'04), Feb 2004,

[6] Norman P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers” in ACM SIGARCH Computer Architecture News, May 1990

[7] AJ Uppal, RC Chiang, “Flashy prefetching for high-performance flash drives” in 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)

[8] WU Fengguang, XI Hongsheng, “On the design of a new linux readahead framework” in ACM SIGOPS Operating Systems Review, July 2008

[9] V Fedorov, J Kimfas, “Speculative paging for future NVM storage” in MEMSYS '17: Proceedings of the International Symposium on Memory Systems, October 2017, Pages 399–410

[10] UMassTraceRepository, http://traces.cs.umass.edu/

[11] MSR Cambridge Traces, http://iotta.snia.org/traces/388

[12] FT Hady, A Foong, “Platform storage performance with 3D XPoint technology” in Proceedings of the IEEE (Volume: 105, Issue: 9, Sept. 2017)

Abstract

Optimization of a data buffer for a hardwired prefetcher in a PCRAM controller buffer

Seokbo Shim
Electrical and Computer Engineering
The Graduate School
Seoul National University

I In this paper, we study the prefetcher structure to improve the cache buffer performance of PCRAM based storage. We perform optimization of the NVM stroage cache buffer by proposing a lightweight hardware prefetching structure for PCRAM hard-wired controller, not a general software prefetcher for HDD and Nand-SSD.

At this time, in order to improve the area overhead of the history buffer and hardware complexity of the prefetching algorithm, which is a drawback when implementing the hardware prefetcher, the history buffer is lightened by configuring a filter using application id polling and sequential address boundary detector. The application id poller selects the populated application id at the unit time point and applies only the cache miss address of the application id to the sequential boundary detector. Sequential boundary detector detects the sequentiality of miss addresses, records them in the history

buffer, and creates prefetch requests for each type based on this.

The average latency of the controller was measured with a real-life storage workload, and it was confirmed that the cache size was optimized so that only 50% of the cache buffer size required the same performance by improving the read latency of about 14%

Keywords : NVM Storage, PCRAM, Data Buffer, Prefetcher, Latency, Cache Size

Student Number : 2019-25565