공학석사학위논문

데이터 집약적 응용을 위한 프로그램 컨텍스트 기반의 I/O 최적화

2019 년 8 월

서울대학교 대학원

컴퓨터공학부

진 용 석

초 록

오늘날에는 다양한 형태의 데이터 집약적인 응용이 활용되고 있다. 이러한 응용들은 대용량의 데이터를 분석하거나, 데이터를 구조화하여 스토리지에 저장하는 등 많은 I/O를 발생시켜, 시스템이 I/O를 수행하는 속도에 따라 성능에 큰 영향을 받게 된다.

운영체제는 메인 메모리보다 성능이 크게 떨어지는 저장 장치로의 접근을 최소화하여 파일 I/O의 성능을 극대화하고자 메인 메모리의 일부를 페이지 캐시로 할당한다. 하지만 메모리의 크기는 저장 장치에 비해 크게 제한되어 있어, 파일 I/O의 성능을 높이기 위해서는 앞으로 참조되는 데이터를 잘 보관하고 참조되지 않을 데이터를 캐시로부터 내보내며 효율적으로 관리하는 것이 매우 중요하다. 하지만 어떤 데이터가 앞으로 참조될지, 그리고 어떤 데이터가 참조되지 않을지에 대해서 시스템이 자체적으로 완벽하게 예측하는 것은 불가능하다. 따라서, 시스템보다 상위 계층에서의 최적화를 위한 노력 없이는 I/O 최적화에 있어 명백한 한계가 존재한다.

본 논문에서는 응용이 I/O를 수행하는 맥락, 즉 프로그램 컨텍스트를 기반으로 I/O가 발생하는 시점과 그 패턴을 자동으로 파악하여 분석하는 기법과, 이를 통해 분석한 결과를 기반으로 하여 각각의 I/O가 발생한 프로그램 컨텍스트에 적용할 최적화 방안 추천을 자동화하는 기법을 제안한다. 이를 통해 시스템에서 자체적으로 파

악할 수 없는 다양한 힌트를 사전에 제공하고, 이 정보를 시스템이 적극적으로 활용하여 이전보다 효율적인 I/O를 수행할 수 있도록 한다.

주요어 : I/O 최적화, 프로그램 컨텍스트, 페이지 캐시, 버퍼 관리, 운영체제
학 번 : 2017-25886

목 차

표 목차

그림 목차

제 1 장 서 론

제 1 절 연구의 배경

오늘날 페이스북(Facebook)에서 개발한 스토리지 엔진 RocksDB, 대용량의 데이터를 입력받고 중간 과정을 저장하며 연산을 수행하는 GraphChi 등의 데이터 집약적인 응용들이 많이 활용되고 있다. 대용량 그래프를 입력받아 요청받은 정보를 연산하는 GraphChi는 수 GB에 달하는 입력 파일을 입력받고, 수백 MB 이상의 중간 결과물을 수시로 파일 형태로 저장하는 만큼 많은 I/O를 수행하게 된다. 이러한 응용들은 시스템이 I/O를 얼마나 빠르게 처리하는가가 성능 향상에 큰 영향을 미친다[1].

운영체제에서는 I/O를 효율적으로 처리하기 위해 메모리의 일부를 페이지 캐시(Page Cache)로 활용한다[2]. 페이지 캐시는 메모리에 비해 상대적으로 느린 스토리지로 인한 성능 저하를 최소화하고자 고안된 공간으로, 기본적으로는 한 번 읽거나 쓴 데이터는 페이지 캐시에 저장이 된다. 하지만 페이지 캐시를 위해 활용할 수 있는 메모리는 한계가 있기에, 읽고 쓰는 데이터가 많아질수록 페이지 캐시에 머무는 데이터의 비율은 점점 줄어들게 된다. 일반적으로 DRAM과 SSD의 성능 차이가 최소 5배에서 10배 이상까지도 차이

가 난다는 점을 고려하면[3], I/O를 효율적으로 수행하는 데에 있어서 페이지 캐시를 최대한 효율적으로 활용하는 것이 매우 중요하다.

페이지 캐시는 I/O에 특별한 설정이 없는 한 시스템에 접근하는 모든 데이터를 일단 캐시에 저장하게 된다. 하지만, 이 중에서 실제로 다시 재참조되는 데이터의 비율은 상당히 낮다[4, 5]. RocksDB 벤치마크 수행 시 다시 참조되는 데이터의 비율은 약 20%를 넘지 못하는데, 응용과 workload의 특성에 따라서도 달라지지만, 많은 응용에서 일관성을 유지하고 각종 문제를 해결하고자 남기는 로그와 같이 한 번 쓰이고 다시 참조되는 일이 드문 데이터 또한 많이 생성된다는 점 또한 이에 일조한다.

또한, 응용이 수행하는 I/O의 특성에 따라서도 최적화의 여지가 있다. GraphChi와 같이 큰 파일을 입력받아 순차적으로 읽는 경우, 시스템에서 요청한 크기보다 미리 더 읽어서 페이지 캐시에 저장한다면 큰 성능 향상 효과를 얻을 수 있다. 4KB의 페이지를 순차적으로 파일을 읽고 처리하는 간단한 예제를 통해 실험한 결과, 시스템에서 4KB씩 읽을 때에 비해 8KB씩 읽었을 때 약 61%, 32KB씩 읽었을 때 약 70%의 성능 향상이 이루어졌다. 이에 운영체제 내부에서도 일부를 더 읽으며 성능을 최적화하고자 하는 readahead 정책을 도입하고 있다.

이와 반대로, 지정된 크기만큼만 임의의 데이터만을 필요로 하는

경우가 있다. 이 경우에는 앞서 언급한 readahead를 자체적으로 수행하게 되면 스토리지로서는 필요 없는 데이터를 추가로 읽어오는 부담이 더해짐과 동시에 페이지 캐시로서는 필요하지 않은 데이터를 저장하게 되는 문제가 생기기 때문에, 운영체제의 readahead를 억제하는 것이 도움이 된다. 리눅스에서는 이를 억제하기 위해 파일의 설정을 변경할 수 있으나, 일반적인 응용에서 자주 활용되는 기능은 아니다.

제 2 절 연구의 목적 및 기여

응용은 각각의 I/O를 수행하는 맥락에 따라 일정한 패턴을 가진다[7]. 예를 들어, 입력 파일을 읽고 처리하는 경우에는 순차적으로 읽는 패턴을 보인다. 응용이 실행되는 중간에 시스템 일관성을 유지하기 위해 로그를 남기는 경우, 응용이 이후 비정상적인 현상으로부터 복구하는 상황이 오지 않는 한 이 데이터는 다시 읽히게 될 확률이 매우 드물다. 하지만 시스템은 응용이 발생시키는 I/O의 특성을 모두 알 수는 없기에, 이러한 모든 상황에 대한 최적화는 어려운 상황이다. 따라서 응용 입장에서 현재 수행하는 I/O의 특성에 대해 시스템에게 힌트를 제공할 수 있다면 I/O 성능 향상에 큰 도움을 줄 수 있을 것이다.

POSIX 기반의 시스템은 fadvise[14]라는 인터페이스를 지원한다. fadvise는 system call으로 구현된 인터페이스로, I/O를 수행하는 파일에 대해 시스템에게 힌트를 제공한다. 순차적으로 액세스하는 파일이나 그렇지 않은 파일에 대해 I/O 수행 이전에 fadvise 인터페이스를 통해 힌트를 제공하면 시스템은 이에 맞게 파일의 설정을 변경한다. 또한, 해당 페이지가 앞으로 계속 참조될지, 아니면 더 이상 참조되지 않는지에 대한 힌트도 제공할 수 있다. I/O 수행 이후에 해당 힌트를 제공하면 시스템은 요청한 페이지를 제공받은 힌트에 따라 캐시에 더 오래 머물도록 설정하거나 즉시 내보낸다. 페이지

캐시는 어떤 데이터가 더 이상 필요하지 않을지 자체적으로 판단하기 어렵기 때문에 기본적으로 읽거나 쓴 모든 데이터를 우선 저장한 뒤 공간이 모자랄 때만 비운다는 점을 고려하면, 필요하지 않은 데이터를 빠르게 내보내는 것은 메모리를 효율적으로 사용하는 데에 큰 도움을 줄 수 있다.

하지만 POSIX를 지원하는 리눅스가 널리 활용되고 있음에도, fadvise 인터페이스는 현재 많이 활용되고 있지 않은 상황이다. RocksDB에는 POSIX 환경에서의 I/O 설정에 fadvise를 활용할 수 있도록 'Hint'와 'InvalidateCache' 와 같은 인터페이스가 구현은 되어 있으나, 실제로 이를 적극적으로 수행하고 있지는 않다.

fadvise를 쉽게 적용하지 못 하는 이유는 우선 코드 레벨에서 사용자가 직접 적용해야 한다는 불편함이 있을 것이다. 코드의 복잡도가 커질수록 다양한 맥락에서 I/O가 발생하는데, 모든 맥락에 대해 완전히 이해하고 그에 적합한 최적화를 적용하기에는 다소 어려움이 따른다.

응용의 I/O 패턴을 간편하게 분석하기 위해 본 논문은 PCStat이라는 I/O 프로파일러를 제안한다. PCStat은 응용이 발생시키는 I/O를 프로그램 컨텍스트(PC)[6, 7] 단위로 분류하고, 각각에 해당하는 I/O 패턴을 분석한다. PC란 특정 시점까지 프로그램이 실행되어 온 실행 경로를 말한다[6]. 예를 들어 응용이 read 시스템 콜을 호출하

기까지 [main()→readInputFile()→read()] 총 3개의 함수를 수행했고 read() 함수에서 read 시스템 콜을 호출하였다면, 이 실행 경로가 곧 해당 read 시스템 콜을 호출한 PC가 되는 것이다. 서로 같은 실행 경로 하에서 발생한 I/O는 곧 같은 맥락에서 발생하였음을 뜻한다. 따라서, PC를 기준으로 I/O 수행 기록을 분류한다면 각 맥락에서 어떤 패턴의 I/O를 발생시키는지 간단하게 분석할 수 있다.

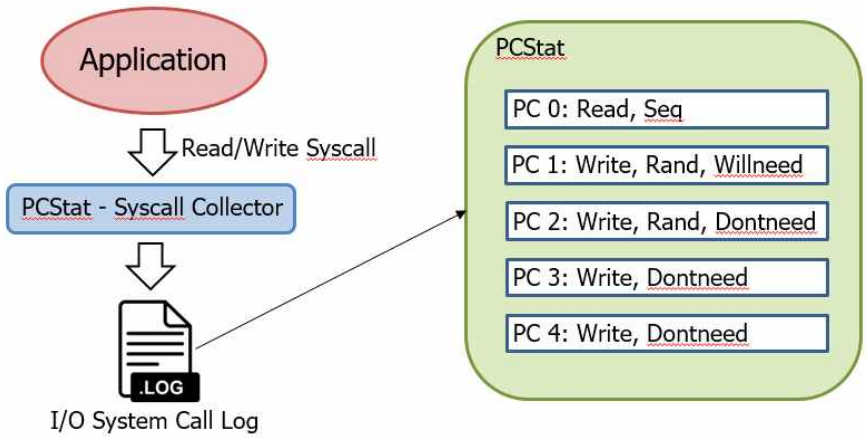

그림 1 PCStat의 구조

PCStat은 응용의 실제 I/O 수행 기록을 캡쳐하여 분석하기 때문에, 기존에 수행된 PC 기반의 시스템 레벨에서의 최적화 관련 연구보다[7] 각각의 맥락에서 발생한 I/O의 패턴에 대해 정확하게 분석하고 조언할 수 있다. 특히 기존 방식의 가장 큰 한계점은 on-line

으로 운영되는 시스템의 입장에서 어떤 데이터가 앞으로 참조되지 않을지에 대한 구분이 사실상 불가능하다는 점이다. PCStat은 응용이 수행하며 액세스한 모든 기록을 토대로 분석하기 때문에, 앞으로 참조되지 않을 데이터를 발생하는 PC를 정확하게 탐지할 수 있다. 따라서 PCStat을 활용한 분석 결과를 기반으로 각각의 I/O 요청 시점에 따라 fadvise 등의 적절한 힌트를 배치하여, 시스템이 해당 응용이 요청하는 I/O를 더 효율적으로 처리하도록 개선할 수 있도록 하는 것이 본 연구의 목적이다.

제 3 절 논문 구성

본 논문의 구성은 다음과 같다. 2장에서는 프로그램 컨텍스트를 활용한 각종 관련 연구에서 제안한 기법을 소개한다. 3장에서는 응용이 I/O를 발생시킬 때 프로그램 컨텍스트를 추출하는 방식과 본 논문에서 제안하는 프로그램 컨텍스트 기반의 I/O 분석 기법과 과정에 대해 설명한다. 4장에서는 3장에서 제안한 응용의 I/O 분석 결과를 기반으로, 각각의 주요 프로그램 컨텍스트에 대해 최적화를 적용하는 방식을 소개한다. 5장에서는 3장과 4장에서 제안한 I/O 분석 및 최적화를 적용한 실험의 환경과 결과를 보인다. 마지막으로 6장에서는 결론을 통해 앞서 설명한 내용들을 정리하고, 향후 확장될 수 있는 연구에 대해 다룬다.

제 2 장 관련 연구

제 1 절 프로그램 컨텍스트를 활용한 버퍼 캐싱

운영체제가 버퍼를 관리하는 데에 있어 가장 중요한 것은 hit ratio를 높이는 것이다[15]. 이를 높이기 위한 많은 연구가 있었는데, I/O 패턴에 중점을 두어 진행한 연구가 그중 하나이다. 이 연구에서는 프로그램 컨텍스트 단위의 I/O 액세스 패턴을 파악한다. 각각의 프로그램 컨텍스트는 다음과 같이 총 4가지 패턴으로 분류된다.

패턴 이름	특징
One-shot	1번만 접근함
Looping	특정 블록들의 집합을 돌아가며 접근함
Temporally Clustered	최근에 접근한 블록에 접근할 확률이 높음
Others	그 외

표 1 I/O 패턴 분류 기준[7]

One-shot의 경우에는 단순히 한 번 접근한 블록은 다시 접근하지 않는 패턴으로 구분에 큰 어려움이 없다. 나머지 3개의 패턴은 내부적으로 'average reference recency'[7]를 계산하여 구분하는데, 그

계산법은 다음과 같다.

특정 PC가 i번째 I/O 액세스를 하는 순간에, 가장 오래전에 액세스한 블록 순으로 나열한 리스트를 L_i라고 가정한다. 만약 특정 PC가 [4 2 3 1 2 3] 순으로 액세스를 했다면, 6번째로 블록 3에 액세스하는 순간인 L_6 = {4, 3, 1, 2}가 될 것이다. $|L_i|$ 가 L_i의 길이라고 하고, 리스트 L_i 내에서 i번째로 액세스할 블록의 현재 위치를 p_i라고 하면, i번째의 'reference recency' R_i는 다음과 같이 계산한다.

$$R_i = \begin{cases} p_i/(|L_i| - 1), & |L_i| > 1 \\ 0.5, & |L_i| = 1 \\ \perp, & undef\ on\ first\ access \end{cases}$$

예를 들어, [1 2 3 4 4 3 4 5 6 5 6] 와 같이 최근에 접근한 블록이 다시 접근될 확률이 높은 경우, R의 평균값은 0.79로 상당히 높게 계산될 것이다. 반대로 [1 2 3 1 2 3] 와 같이 루프를 도는 패턴에 대해서는 R의 평균값은 0으로 굉장히 낮게 계산된다. 이 연구에서는 T=0.4로 설정하여, T보다 R의 평균값이 낮은 PC에 대해서 looping 패턴을 보이며, 높은 경우에는 temporally clustered 패턴을 보인다고 구분한다.

위와 같이 PC의 패턴이 파악되고 나면 공간을 할당할 때 해당

PC의 특성에 맞추어 할당하게 된다. 예를 들어 looping 패턴을 가지는 PC로부터 I/O 요청이 발생했고 이 과정에서 cache miss가 발생하게 된다면, 이전에 파악한 loop size에 기반한[16] 확률로 임의의 MRU 파티션으로부터 블록을 내보내어 공간을 확보한다. 이와 같은 기법을 통해 기존의 버퍼 관리 기법인 LRU나 ARC[20]에 비해 cache miss rate를 그림 2와 같이 최대 50%까지 줄여 크게 성능을 개선하였다.

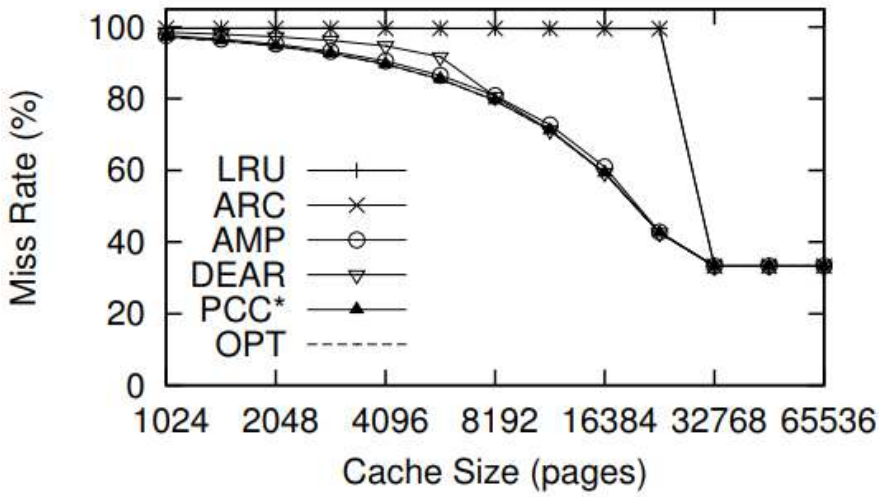

그림 2 기존 기법 대비 AMP의 Cache Miss 절감 효과[7]

하지만 위 연구에서도 결국 캐시에서 방출하는 데이터는 시스템의 부담을 줄이기 위해 무작위로 결정한다. 어느 데이터를 방출할

것인지는 시스템이 자체적으로 판단하기 어려운 영역에 있어, 이를 더 완벽하게 예측하기 위해서는 상위 레벨에서의 조언이 불가피한 상황이다.

제 2 절 프로그램 컨텍스트 기반의 데이터 분리 기법

낸드 플래시 메모리는 저장 장치로서 오늘날 널리 사용되고 있다. 하지만 낸드 플래시 메모리의 특성상 이를 기반으로 하는 스토리지 장치는 유효하지 않은 페이지를 수거하는 가비지 컬렉션 작업이 요구된다[8, 9]. 가비지 컬렉션 작업은 유효한 페이지를 읽고 새 공간에 쓰며, 블록을 지우는 과정을 거치기 때문에 많은 시간이 걸리게 된다. 따라서 이로 인한 오버헤드를 최소화하고자 많은 연구가 진행되었다[10].

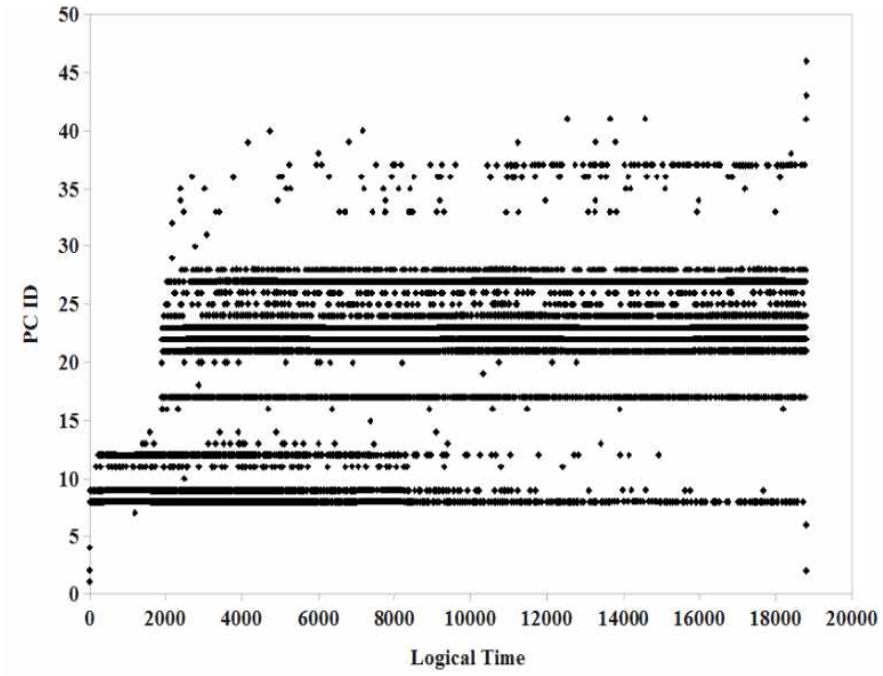

그림 3 TPC-R 벤치마크의 PC에 따른 데이터 갱신 주기[10]

가비지 컬렉션은 대상 블록에 유효한 페이지가 있을 때마다 이를 읽고 새 공간에 복사하는 작업을 거쳐야 하기에, 유효한 페이지가 많을수록 이로 인한 오버헤드가 커지게 된다. 이는 즉 무효한 페이지들과 유효한 페이지들을 최대한 분리하여 각각 다른 블록에 모이도록 한다면 가비지 컬렉션으로 인한 오버헤드를 줄일 수 있다는 것을 의미하게 된다. 따라서 이를 위해 데이터를 각각의 특성에 따라 분리하는 여러 기법이 이전부터 연구되었다.

가장 흔히 사용되는 데이터 구분 기법 중 하나는 데이터가 갱신되는 주기를 이용한 것이다[11]. 자주 갱신되는 데이터가 쓰인 페이지는 곧 무효화될 확률이 높고, 반대로 드물게 갱신되는 데이터가 쓰인 페이지는 무효화되지 않고 오래 남을 확률이 높다. 이를 간단하게는 hot data와 cold data로 나누어, SSD에서 쓰기 요청이나 가비지 컬렉션으로 인해 데이터를 쓰게 될 때 hot data가 저장될 블록과 cold data가 저장될 블록을 각각 구분하여 저장한다. Hot data가 저장된 블록은 곧 대부분의 페이지가 무효화될 것이므로 가비지 컬렉션 과정에서 복사할 유효한 페이지의 수가 줄어들게 되고, 따라서 이로 인한 오버헤드가 줄어들게 된다.

하지만 가장 큰 문제는 각각의 데이터의 갱신 주기를 판단하기가 어렵다는 점이다. 이를 예측하기 위해 여러 연구가 진행되어왔는데, 그중 하나는 프로그램 컨텍스트를 힌트로 삼은 연구이다. 같은 문맥

에서 발생한 데이터는 유사한 수명을 가질 것이라는 아이디어를 기반으로 한 이 연구는 사용자로부터 I/O 요청이 올 때 그에 해당하는 프로그램 컨텍스트를 계산하고, 그에 해당하는 PC signature를 스토리지로 전달하여 데이터 구분에 참조할 수 있도록 한다.

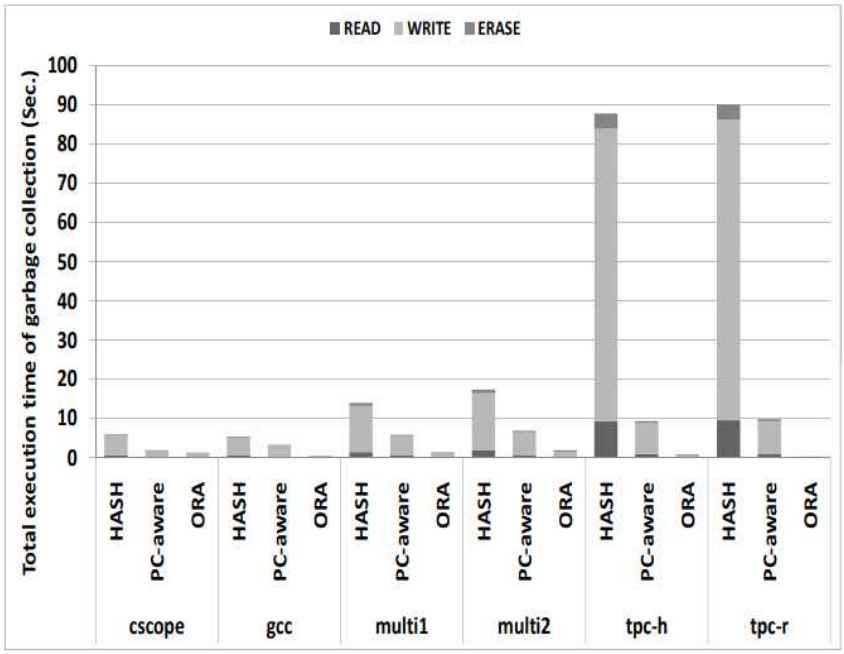

그림 4 PC-aware FTL의 가비지 컬렉션 효율 비교[10]

그림 3는 어떤 PC에서 언제 데이터가 갱신되었는지를 나타낸다. 대부분의 PC에서는 데이터가 쓰인 이후 갱신이 많이 일어나지 않는데, 일부 PC에서만 굉장히 빈번하게 데이터가 갱신되고 있는 것을

확인할 수 있다. 이에 주목하여 PC 단위로 Update Group을 결성하고, 해당 묶음 단위로 SSD 내에서 블록을 구분하여 할당하는 방식으로 프로그램 컨텍스트를 활용한 연구가 있다. 그 결과, 그림 4와 같이 기존의 hash 기반의 데이터 분리 기법 대비 가비지 컬렉션 수행 시간이 평균적으로 약 58% 감소하였다.

최근에 SSD에 기존의 FTL이 주관하던 공간 할당에 사용자가 개입할 수 있는 Multi-Stream이라는 인터페이스가 고안되었다, Multi-Stream을 지원하는 SSD는 쓰기 요청을 받을 때 별도로 stream ID를 입력받을 수 있으며, 서로 다른 stream ID에게는 서로 분리된 공간을 제공한다. 사용자가 직접 공간 할당에 힌트를 제공할 수 있다는 점에 착안하여, 이를 프로그램 컨텍스트를 통한 데이터 구분과 접목하는 기법도 제안되었다.[12]

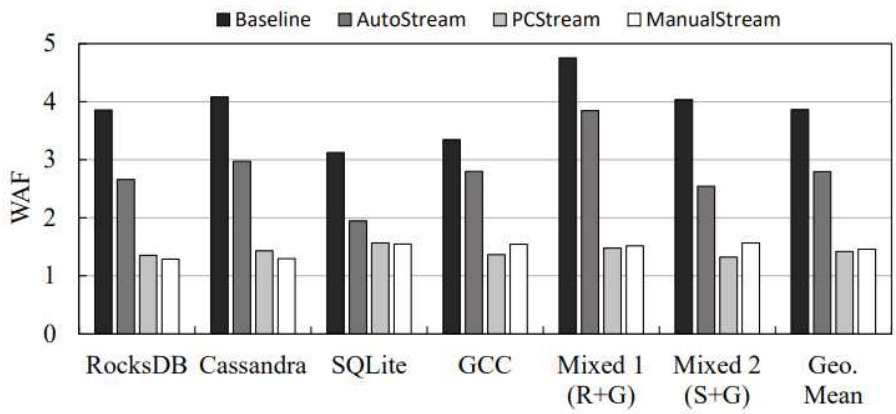

그림 5 PCStream의 WAF 비교[12]

서로 다른 stream에게는 다른 블록이 할당된다는 것이 보장된다는 점을 이용하여, 커널에서는 해당 PC에서 발생하는 데이터의 수명에 따라 stream을 할당한다. 하지만 같은 PC에서 발생하는 데이터라 할지라도 수명이 크게 갈릴 가능성도 존재한다. RocksDB의 경우 데이터가 저장된 LSM Tree 상의 레벨에 따라 수명이 크게 갈릴 수 있다. 낮은 레벨일수록 갱신될 확률이 높고, 데이터가 쌓임에 따라 레벨이 높아질수록 갱신될 확률은 굉장히 낮아진다. 하지만, RocksDB의 compaction은 레벨에 따라 실행 경로가 변하지 않기 때문에 서로 다른 레벨이더라도 같은 PC 상에서 I/O를 수행하게 되고, 이에 따라 같은 PC 내에서도 데이터의 수명이 크게 갈리게 된다.

가비지 컬렉션 과정에서 복사되는 데이터가 자주 갱신되지 않는 데이터일 확률이 높다는 점과 너무 많은 stream을 활용하게 되면 스토리지의 성능이 떨어진다는 점을 활용하여, 해당 연구에서는 'Internal Stream'을 제안한다. 일부 stream에 대해 사용자의 직접적인 할당을 제한하고, 가비지 컬렉션을 수행할 때에만 내부적으로 공간을 할당하여 데이터를 저장할 수 있도록 운영한다. 이와 같이 데이터를 한 번 더 분류할 수 있는 추가 장치를 도입하여 같은 PC를 가지는 데이터 사이에도 기존보다 더 세밀하게 데이터를 분류하도록 하였다.

실험 결과 그림 5와 같이 일반 SSD와 기존의 AutoStream 대비 평균 WAF를 각각 63%, 49% 감소하였으며, stream 할당을 자동화 하였음에도 사용자가 직접 모든 stream을 할당한 ManualStream과 거의 유사한 성능을 보였다.

제 3 장 프로그램 컨텍스트에 기반한 응용 I/O 분석

제 1 절 프로그램 컨텍스트의 정의와 추출 기법

프로그램 컨텍스트(PC)는 특정 시점까지 프로그램이 실행되어 온 실행 경로를 말한다.[6] 응용이 읽기나 쓰기와 같은 I/O를 요청할 때, 읽거나 쓰게 될 데이터는 이전의 실행 경로에 따라 결정된다. 예를 들어 RocksDB의 경우, 일반적으로 읽기 요청에는 사용자가 'Get' method를 통해 데이터를 요청하는 경우와, LSM Tree의 compaction 과정에서 기존에 쓰여진 데이터를 읽는 경우가 있으며, 쓰기 요청에는 로그를 남기는 경우와 LSM Tree의 compaction 과정에서 데이터를 쓰는 경우, 사용자로의 데이터로 구성된 SST를 디스크에 작성하는 경우 등이 있다. 이 실행 경로의 흔적은 프로세스의 stack 영역에 남게 되는데, 각각의 함수를 호출하는 과정에서 함수가 return하고 나면 다시 돌아갈 주소를 stack 영역에 저장하기 때문이다[17].

가령 RocksDB가 로그를 작성하게 된다면, [WriteImpl()→WriteToWAL()→AddRecord()]의 실행 흐름을 거쳐 write() 함수를 호출하게 된다. 따라서 write() 함수를 호출한 시점에서, stack 영역

에는 순서대로 AddRecord, WriteToWAL, WriteImpl 함수 내의 return 할 주소가 저장되어 있다. 이 주소들의 집합을 PCStat은 해당 I/O 시스템 콜의 PC로 간주한다.

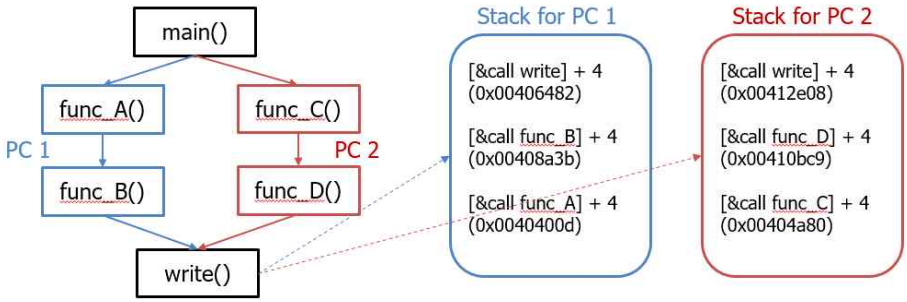

그림 6 write I/O 수행 시 각 PC의 stack 상태의 예시

PC의 추출 방식은 다음과 같다. 가령 write() 시스템 콜이 호출되었다고 했을 때, 사전에 추가된 모듈이 동작하여 사용자 영역의 stack을 stack pointer로부터 탐색한다. 만약에 stack에 저장된 값이 사용자 영역의 code 범위 안에 포함된다면, stack 내에 저장된 그 값을 PC의 일부로 보고 해당 값을 저장한다. 이렇게 user stack으로부터 모은 code 영역 내 주소의 개수가 특정 임계값을 넘어가거나 stack을 끝까지 탐색하고 나면 발견한 주소값들을 해당 I/O의 PC로 본다.[12]

하지만, Intel Broadwell 등 신형 컴퓨터 아키텍처에서는 메모리

내 사용자 영역에 대한 접근을 SMAP(Supervisor Mode Access Prevention)[18]이라는 방어 기법을 통해 막고 있다. 다양한 컴퓨터 아키텍쳐 하에서도 PC를 원활하게 계산하기 위해서는 단순 연산이 아닌, 리눅스에서 제공하는 사용자 영역에 안전하게 접근할 수 있도록 하는 copy_from_user[13] 등의 API를 사용하였다.

제 2 절 PCStat: 프로그램 컨텍스트에 기반한 I/O 패턴 분석

본 논문에서는 응용이 실행하면서 발생한 I/O 시스템 콜 기록을 수집하여 저장하고, 이를 각각의 PC 단위로 구분하여 분석하는 PCStat을 제안한다. PCStat은 총 두 단계를 거쳐 응용이 I/O를 발생하는 PC를 구분하고 그에 해당하는 I/O 패턴을 분석한다.

첫 번째로, 분석하고자 하는 응용이 실행 도중에 발생하는 I/O 시스템 콜의 기록을 수집한다. 이를 위해 분석자가 임의로 지정한 응용이 I/O 시스템 콜을 호출할 때마다 분석에 필요한 정보를 수집하는 PC module을 개발하였다. 해당 모듈을 활성화하면 대상 응용 호출한 각각의 I/O 시스템 콜이 발생한 시간, 이를 완료하는 데에 걸린 지연 시간, I/O 타입(read, write, readv, writev), 파일명, offset, 크기, 그리고 3장 1절에서 다룬 해당 I/O 시스템 콜이 발생한 PC 총 7가지를 수집한다. 수집된 정보들은 로그 파일에 별도로 저장되며, 다음의 분석 단계에서 활용된다. 3장 2절에서 언급한 별도의 I/O 시스템 콜을 통해 수집된 데이터는 위의 7가지 중 PC를 수집하는 부분이 빠지게 되고, 대신 추가 인자를 통해 전달받은 PC signature를 가지게 된다.

두 번째로, 응용이 실행하며 수집한 로그를 분석하는 과정을 거친다. 앞선 과정을 통해 시간의 흐름에 따라 수집된 기록을 분석하는

데, 이를 위해 총 3가지 과정을 거치게 된다. 앞서 수집한 PC를 실제 함수명으로 변환하고, 각각의 시스템 콜 기록들을 PC 단위로 분류하며, 마지막으로 분류된 각각의 PC에서 수행된 시스템 콜들의 패턴을 분석한다.

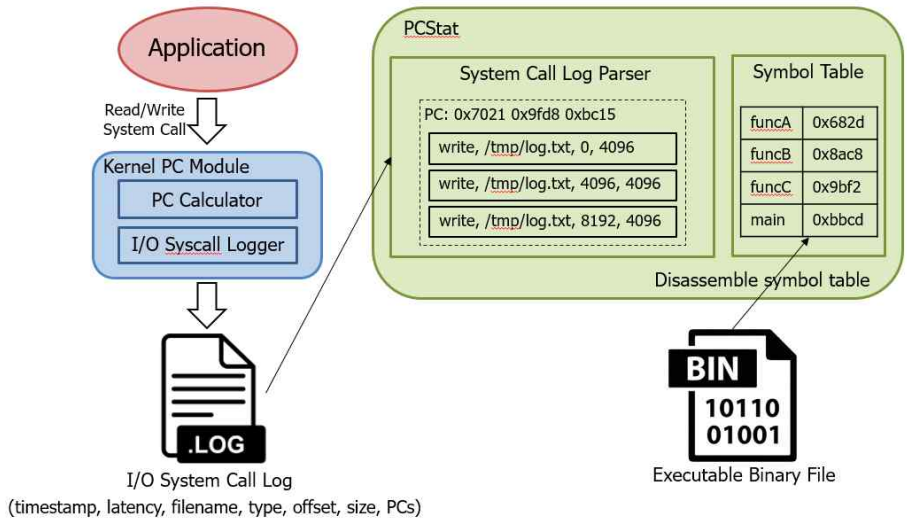

그림 7 PCStat의 응용 I/O 수집 및 분류 과정

우선 사용자가 코드 레벨에서의 최적화를 진행할 수 있도록 실제 함수명으로 변환하는 작업을 거친다. 이 작업을 위해, 앞선 단계에서 stack에 저장된 code 영역의 주소를 가져올 때는 그 주소를 그대로 저장하지 않고 해당 주소의 값에서 code 영역의 시작 주소의 값을 뺀 offset 값을 저장한다. 이렇게 저장하는 이유는 런타임에 있는

프로세스의 code 영역의 시작 주소가 분석할 때의 정적 바이너리가 가지는 시작 주소와 다를 수 있기 때문이다. 각각의 주소를 실제 함수로 변환하기 위해 우선 해당 바이너리의 심볼 테이블을 디스어셈블하고, 이를 통해 바이너리의 시작 주소와 각각의 함수들의 주소를 파악한다. 그 뒤, 로그에 저장된 offset 값들에 바이너리의 시작 주소를 더한 뒤 심볼 테이블과 비교하여 해당 주소가 어느 함수에 속하는 주소인지 찾아낸다.

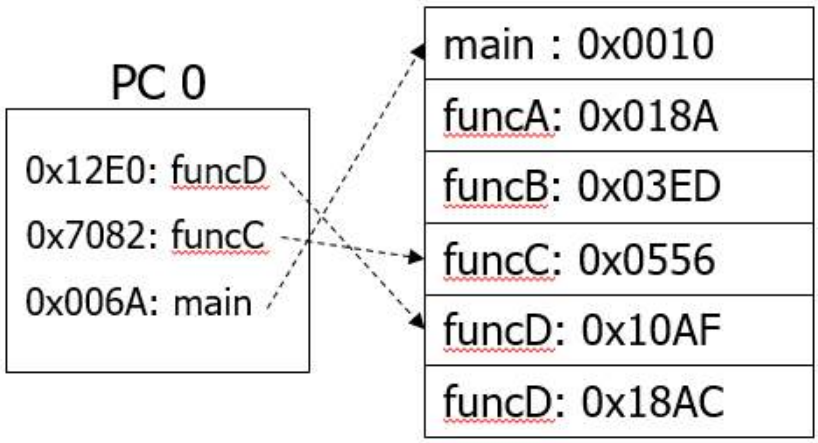

그림 8 각 PC를 실제 함수명으로 변환하는 예시

예를 들어 0x003A 라는 offset을 로그로부터 읽었고 바이너리 시작 주소가 0x1000이라면 이 둘을 더한 0x103A가 찾고자 하는 문맥의 주소가 된다. 심볼 테이블을 통해 함수 A의 주소가 0x1000, B의

주소가 0x1030, C의 주소가 0x104E라는 정보를 얻었다면, 0x103A는 이보다 작은 주소 중 가장 큰 값인 0x1030에 해당하는 B 함수를 수행한 기록이라는 것을 알 수 있다. 단, 이 과정에서 함수의 주소와 파악하고자 하는 주소가 같은 경우에는 해당 PC에서 배제한다. 컴퓨터 시스템의 특성상 함수가 시작하는 지점으로 return 하는 것이 불가능하기에, 이 경우에는 프로그램의 실행 맥락이 아닌 함수 포인터를 stack 영역에 변수로서 저장한 경우이기 때문이다.

다음으로, 주어진 시스템 콜들을 각각의 PC에 따라 분류한다. 단순히 모든 PC 주소가 겹치는 경우만 고려해서는 정확한 분류가 어려워지는데, 그 이유는 컴퓨터 시스템이 stack 영역을 사용한 뒤에 남은 찌꺼기 값들을 0으로 되돌리지 않기 때문이다. 그렇기에 같은 맥락에서 발생한 I/O라도 시점에 따라 다른 함수와 관련된 주소가 포함되어 있을 수 있다. 다만 이로 인해 같은 맥락에서 수행하였음에도 다른 주소값을 가지는 경우가 연속적으로 발생하지는 않을 것이라는 가정 하에, 1개의 다른 주소를 가지고 그 이외에는 특정 개수 이상의 같은 실행 흐름을 가지고 있는 PC라면 같은 PC로 분류하도록 설계하였다.

예를 들어, 서로 시점은 다르지만 같은 A()→B()→C()→write() 의 과정을 거친 2개의 I/O가 발생했다고 가정한다. 시스템 콜 1은 PC 추출 시 A→B→C 로 계산되었는데, 같은 맥락에서의 시스템 콜 2가

발생하기 이전에 D 함수가 실행된 적이 있었다면, 이에 대한 PC가 A→B→D→C 와 같이 추출되는 경우가 생길 가능성이 있다. 하지만 D를 제외하면 둘 다 A→B→C 의 3개 이상의 공통된 실행 흐름을 가지고 있으므로 시스템 콜 2에서 계산된 PC에서 D를 없애고, 둘은 같은 PC라고 판단한다. 이러한 방식으로 사전에 응용이 수행하는 도중 수집한 I/O 시스템 콜 기록을 각각의 PC 단위로 분류하여, 어느 PC에서 어떤 패턴의 I/O를 수행하는지에 대한 분석이 가능하도록 하였다.

패턴	특징
SEQUENTIAL	순차적으로 발생하는 PC
RANDOM	순차적으로 발생하지 않는 PC
DONTNEED	재참조되지 않는 block에만 접근하는 PC
WILLNEED	자주 참조되는 block에 접근하는 PC

표 2 PCStat이 분류하는 PC의 패턴

마지막으로 각각의 PC 단위로 분류된 시스템 콜들의 패턴을 분석한다. 각각의 PC에서는 어떤 종류(읽기, 쓰기)의 I/O가 발생하는지, 각각의 I/O 요청의 평균 크기는 어느 정도인지, 평균 지연 시간은 어느 정도인지를 이를 통해 확인할 수 있다. 또한, 각 PC의 I/O 패

턴을 표 2와 분류한다. 해당 PC에서 액세스한 파일명과 offset, size를 참조하며 해당 PC가 순차대로 접근하는지, 아니면 임의의 위치에 접근하는지에 대한 정보를 얻을 수 있다. 또한, PCStat은 각 I/O 수행 시 액세스한 block과 그 시간을 알 수 있기에 특정 block 단위로 액세스한 시간을 계산하며 어느 block이 자주 액세스되었는지, 또는 해당 block이 응용을 실행하는 동안 전혀 재참조되지 않았는지를 확인할 수 있다. 이를 통해 자주 참조되는 block에 액세스하는 PC, 또는 재참조되지 않는 block만에 액세스하는 PC를 분류할 수 있다.

재참조되지 않는 데이터를 명확하게 알 수 있다는 점은 시스템 내부에서 예측하는 기법과 대비했을 때의 가장 큰 장점이다. 자체적으로 예측하는 경우 자주 참조되어 계속 가지고 있어야 하는 블록은 판별할 수는 있지만, 이후의 요청에 대해서는 알 수 없기에 어느 블록을 내보내는 것이 최선일 지에 대한 예측이 불가능하며 이에 근접한 시도만으로도 큰 연산 오버헤드를 발생할 것이다. PCStat을 활용하면 어느 PC가 다시는 참조되지 않는 데이터를 발생시키는지 파악할 수 있으며, 이를 토대로 효율적인 페이지 캐시 관리에 크게 기여할 수 있다.

제 3 절 I/O 쓰레드 환경을 위한 프로그램 컨텍스트의 추출 기법

GraphChi[19]와 같은 데이터 집약적인 응용은 메인 쓰레드로부터 전달받은 I/O 요청만 수행하는 I/O 쓰레드를 따로 두어 활용하고 있다. 이러한 경우에서는 I/O 수행 시 기존의 쓰레드의 PC가 아닌 I/O를 수행하고 있는 쓰레드의 PC가 계산되어, 실제로 찾고자 하는 PC를 찾을 수 없게 된다.

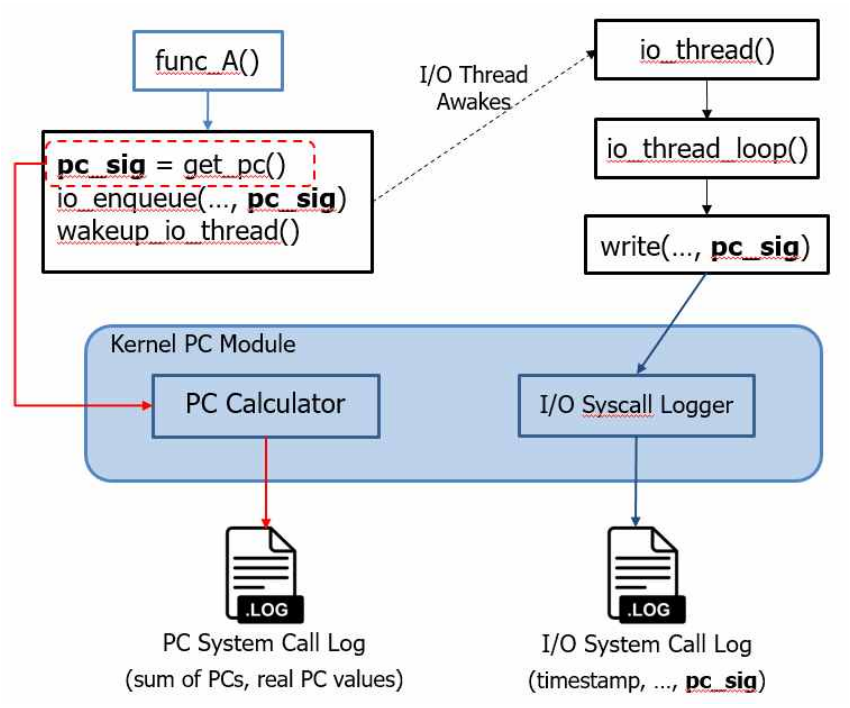

그림 9 I/O 쓰레드 환경을 위한 PC 추출 기법

이를 해결하기 위해, 사용자가 지정한 시점에서 PC를 파악하고 저장하는 시스템 콜인 PC 시스템 콜과 이와 연동하기 위한 기존 I/O 시스템 콜의 변형을 구현하였다. 그림 7와 같이, PC 시스템 콜을 호출하면 현재의 PC 값 전체와 그 합을 PC signature로서 계산하여 별도의 로그 파일에 저장한다. 이후 I/O 쓰레드에서 I/O를 수행할 때, I/O 쓰레드의 프로그램 컨텍스트가 아닌 이전에 저장된 것과 연동시키기 위해 변형한 I/O 시스템 콜을 호출한다. 이 시스템 콜은 추가 인자로서 사전에 계산된 PC signature를 받고, 이를 기존의 PC 시스템 콜을 통해 저장된 PC 로그와 연동하는 데에 사용한다.

제 4 장 프로그램 컨텍스트에 기반한 응용 I/O 최적화 적용

제 1 절 페이지 캐시에 제공하는 힌트

일반적으로 POSIX 표준을 따르는 리눅스에서는 fadvise[14]라는 API를 지원한다. fadvise는 사용자가 파일을 다루는 데에 있어 운영체제에 힌트를 제공한다. fadvise는 표 2에 해당하는 힌트를 운영체제에 제공하여 더 효율적인 I/O를 수행하도록 돕는다.

이름	역할
POSIX_FADV_NORMAL	일반적인 I/O와 같음
POSIX_FADV_SEQUENTIAL	최대 readahead 크기 확장
POSIX_FADV_RANDOM	readahead 억제
POSIX_FADV_WILLNEED	지정된 페이지를 버퍼에 잔류
POSIX_FADV_DONTNEED	지정된 페이지를 버퍼에서 방출

표 3 fadvise가 제공하는 기능[14]

fadvise를 적절하게 활용한다면 운영체제가 I/O를 효율적으로 수행하는 데에 많은 도움이 될 수 있다. 하지만 POSIX API 중에서

잘 알려지지 않았고, 단순 API인 만큼 대규모의 복잡한 코드에서 적절한 위치에 직접 fadvise를 적용해야 한다는 불편함이 있어 일부 연구에서만 언급될 뿐 실제로 적절히 활용하고 있는 응용은 많지 않다. RocksDB에서는 POSIX 환경에 대해 'Hint'와 'InvalidateCache'라는 이름으로 사용할 수 있도록 제공하고 있으나, 현재까지는 RocksDB 또한 API로서 사용자에게 맡길 뿐 이를 적절히 활용하고 있지는 않은 것으로 파악되었다.

제 2 절 fadvise 적용을 통한 프로그램 컨텍스트 기반의 I/O 최적화

3장 3절에서 제안한 PCStat을 통해 엔지니어는 분석하고자 하는 응용이 I/O를 발생시키는 문맥의 실제 코드 위치와 해당 문맥에서 발생하는 I/O 패턴, 그리고 해당 문맥에서 접근하는 데이터가 자주 참조되는 데이터인지에 대한 정보를 알 수 있다. 이는 즉 PCStat에서 제공하는 정보를 통해 즉시 코드 레벨에서 fadvise를 통한 최적화를 적용할 수 있음을 말한다.

PCStat은 전체 PC를 분석한 뒤 어느 PC에 어떤 최적화 기법을 적용할지를 사용자에게 추천한다. 기본적으로 I/O 총량이 적은 PC의 경우 최적화의 의미가 많이 떨어지므로 기본적으로는 100MB 이상의 I/O를 수행한 PC에 대해서만 최적화 힌트를 제공하도록 하고 있으며, 이 값은 필요에 따라 사용자에 의해 변경될 수 있다.

PCStat으로 분류한 각각의 PC의 특성 중 I/O의 공간적 지역성에 따라서 최적화를 적용할 수 있다. PCStat은 시간 순서대로 정렬된 system call의 파일명과 offset, size를 확인하여 연속된 공간을 요청하는지 파악한다. 연속적으로 수행된 system call 개수의 평균을 계산하여, 임계값을 넘어서는 경우 해당 PC는 순차적으로 액세스하는 패턴을 가진다고 분류하고, SEQUENTIAL 힌트를 적용할 것을 추천한다. RocksDB 등의 응용 수행을 통해 분석한 결과, 약 4 이상의

길이를 가지는 경우 순차적으로 액세스한다고 볼 수 있다. 반대로 해당 값이 1에 가까운 경우에 해당하는 PC에는 RANDOM 힌트를 적용하여 운영체제의 readahead를 막을 것을 추천한다.

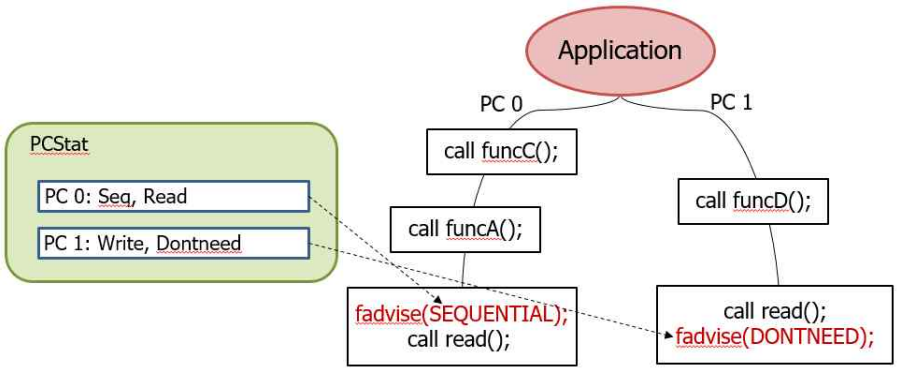

그림 10 PCStat 분석 결과를 바탕으로 한 fadvise 적용 방식 예시

또한, 각각의 PC가 얼마나 자주 참조되는 블록에 접근하는지를 계산하여 자주 참조되는 블록에 접근하는 PC와 다시 참조되지 않는 블록에 접근하는 PC를 구분한다. 자주 참조되는 경우에는 WILLNEED 힌트를 적용하여 버퍼에서 해당 PC로부터 발생하는 데이터를 오래 보관하도록 추천하고, 참조되지 않는 경우 즉시 버퍼에서 내보내는 DONTNEED 힌트를 적용하도록 추천한다.

SEQUENTIAL과 RANDOM 힌트의 경우에는 I/O 수행 전에 주어져야 하는 힌트이기 때문에 read나 write 함수를 수행하기 전에

배치하며, WILLNEED와 DONTNEED 힌트의 경우에는 데이터가 페이지 캐시에 적재된 이후에 주어져야 하는 힌트이기 때문에 read 나 write 함수를 수행한 후에 배치한다.

제 3 절 PCAdvisor: 프로그램 컨텍스트 기반의 I/O 최적화 자동화

4장 2절에서 다룬 최적화 추천은 정교하게 최적화를 적용할 수 있다는 장점이 있지만, 코드를 직접 수정하는 것은 매우 복잡하고 오래 걸리는 일이 될 수 있다. 특히 C++와 같은 객체지향적 언어로 개발된 응용이라면 복잡한 상속 관계로 인해 코드 수정이 크게 복잡해지는 상황이 발생할 수 있다. 따라서 다양한 응용에서 PCStat을 통한 분석 결과를 손쉽게 적용할 수 있도록 본 논문에서는 I/O 최적화를 자동화하는 PCAdvisor를 제안한다.

PCAdvisor는 시스템 내부에 내장되는 커널 모듈이다. PCAdvisor 모듈이 가동되면 사전에 PCStat을 통해 생성한 설정 파일을 읽는다. 이후 I/O 발생 시, PC를 계산하여 설정 파일에 명시된 PC를 가지는 I/O가 발생하면 이에 해당하는 최적화 옵션을 자동으로 적용한다.

PCAdvisor를 지원하기 위해서 PCStat은 두 가지 모드를 가진다. 첫 번째는 코드 최적화용 모드로, 해당 모드는 3장 3절에 서술한 내용과 같다. 다른 하나는 PCAdvisor용 모드로, 이 경우에는 코드를 수정할 필요가 없어 심볼명을 알 필요가 없기 때문에 함수 심볼을 추출하기 위한 바이너리 디스어셈블 및 심볼 매칭과 같은 과정들이 생략된다. PCAdvisor 모드로 동작하는 PCStat은 계산된 PC signature 값과 해당 PC에 적용할 최적화 옵션을 기재한 파일을 생

성하고, 이 파일은 PCAdvisor의 설정 파일이 된다.

PCAdvisor가 커널에 등록되면 우선 설정 파일을 읽고, 최적화하고자 하는 PC의 signature 값과 각각의 최적화 옵션을 읽고 내부의 해시 테이블 구조에 저장한다. 그 후, 지정된 응용이 I/O를 발생시키면 해당 I/O의 PC signature를 계산한다. 만약 그림 11과 같이 PCStat에 의해 최적화가 제시된 PC signature 값을 가지고 있다면, 이 I/O에 대해 PCAdvisor가 자동으로 최적화 옵션을 적용하게 된다.

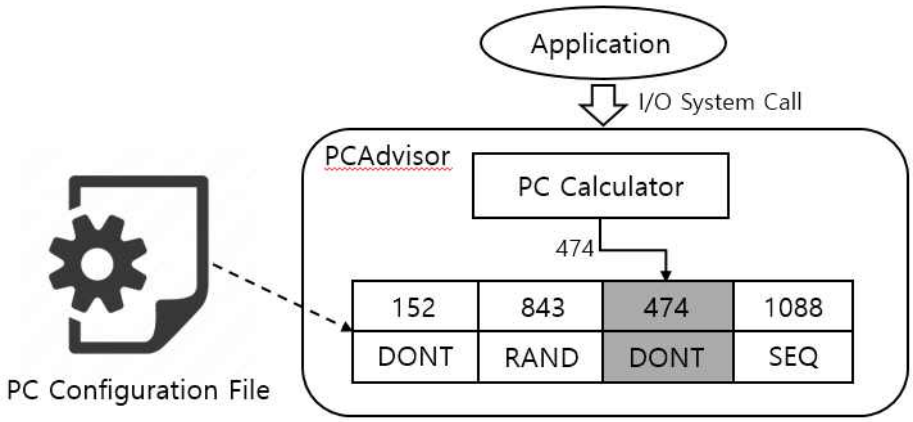

그림 11 PCAdvisor의 동작 방식

예를 들어 PC의 주소값이 각각 10, 20, 30, 65인 I/O가 수행되었다면, 이 모든 값을 합한 125가 PC signature가 된다. 만약 125에

RANDOM과 DONTNEED 옵션을 적용하도록 사전에 제시되었다면, I/O가 수행되기 전에 random I/O에 맞는 설정을 적용한 뒤 해당 I/O가 끝난 뒤 페이지 캐시로부터 해당 데이터를 내보내는 작업까지 모두 자동으로 진행하게 된다.

바이너리와 관련된 주소를 계산하지 않더라도, PC signature를 계산할 때에는 반드시 code 영역의 시작 주소를 뺀 offset으로 계산해야 한다. 같은 프로그램이라 하더라도 해당 프로그램이 실행되며 프로세스로 생성되는 시점에서 주소 공간이 달라질 수 있고, 그렇게 되면 PC signature 매칭이 불가능해지기 때문이다.

제 5 장 평가 실험

제 1 절 실험 환경

실험은 Intel Core i7-2500 8-core processor, 16GB DRAM, Samsung 840 PRO SSD, WD10EALX HDD의 환경에서 진행하였다. 실험에 활용된 운영체제 및 응용의 버전은 다음과 같다.

소프트웨어	버전
Ubuntu	14.04
Linux kernel	3.16.43
RocksDB	2.8_fb
GraphChi	0.101.2
Redis	4.0.14_stable
ClamAV	2.0

표 4 실험에 사용된 소프트웨어의 버전

제 2 절 실험 결과

우선, 버퍼 관리로 인한 성능 변화를 확인하고자 FIO[21]를 개량한 마이크로벤치마크의 총 수행시간을 비교하였다. 쓰기만 수행하는 쓰레드와 쓰고 난 뒤 그 데이터를 읽는 쓰레드가 각각 8개씩 존재하고, 이에 대해 DONTNEED와 WILLNEED 힌트를 제공한 경우와 제공하지 않은 경우에 대해 비교하였다. 그 결과, 그림 12와 같이 최적화를 적용한 이후 SSD에서는 총 수행 시간이 약 10%, HDD에서는 약 25% 감소한 것으로 나타났다.

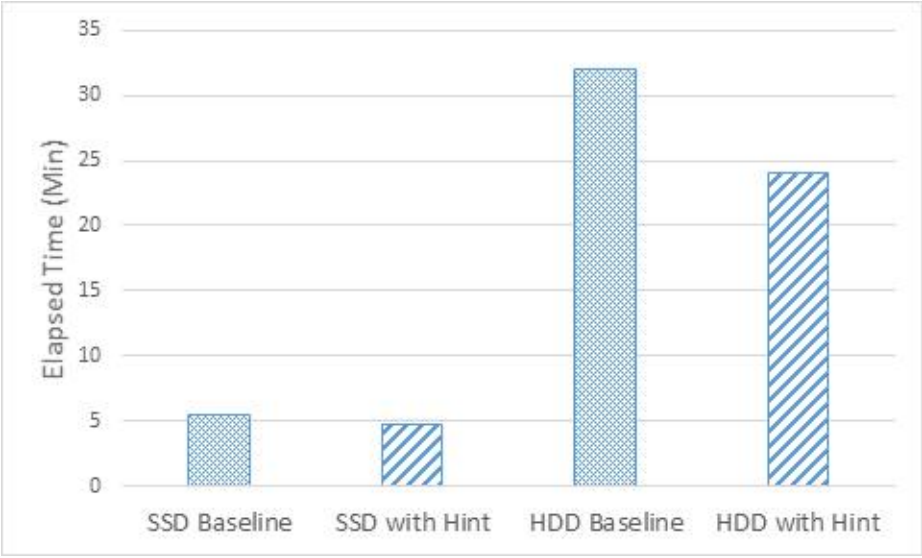

그림 12 마이크로벤치마크 수행 결과

PCStat과 PCAdvisor를 통한 버퍼 관리의 효과를 확인하고자 GraphChi가 약 2GB의 edge 데이터에 대해 pagerank 연산을 수행하는 과정에서의 남은 메모리의 총량을 측정하였다. 그림 13와 같이, 수행 시간이 길어짐에 따라 PCAdvisor를 적용하기 전과 후의 남은 메모리의 총량이 크게 차이가 남을 확인할 수 있다. 큰 입력 파일들을 읽으면서도 PCStat의 분석 결과를 통해 이 중 일부는 다시 읽히지 않을 것을 알기에, PCAdvisor가 자체적으로 페이지 캐시에서 해당 데이터를 방출시키는 과정에서 이와 같은 메모리 잔량의 차이가 발생하게 된다.

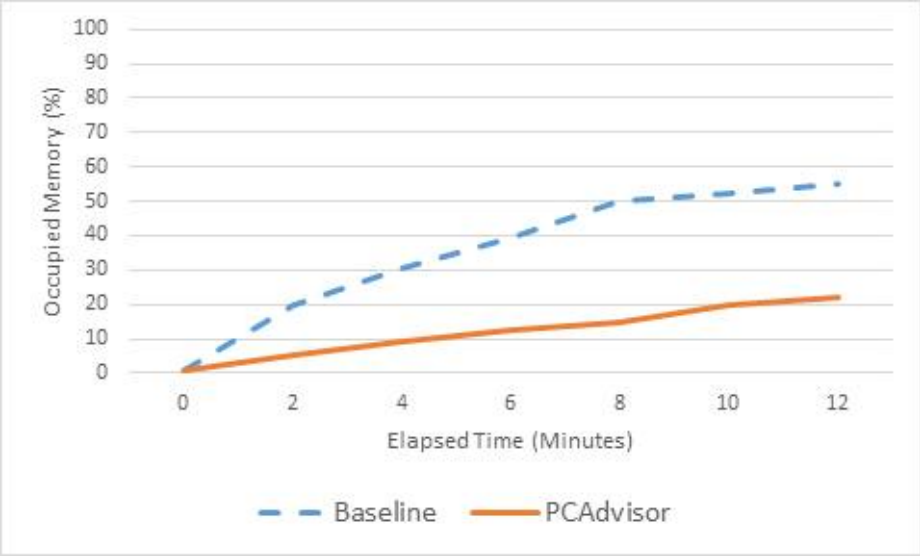

그림 13 GraphChi 수행 도중 메모리 점유율 비교

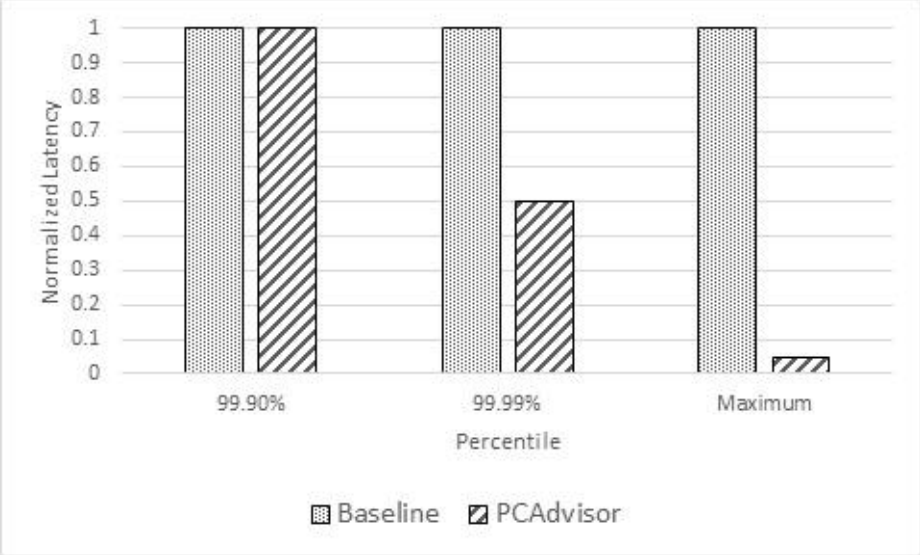

그림 14 ClamAV 가동 중 Redis INCR 연산의 꼬리응답시간

페이지 캐시 관리 효과를 확인하기 위해 ClamAV와 함께 실험을 진행하였다. ClamAV는 안티바이러스 응용으로, 디스크 전체를 탐색하며 바이러스를 탐지한다. 많은 읽기 요청을 발생하지만, 한 번 스캔한 파일은 다시 읽지 않는다는 특성을 가지기 때문에, 일찍 페이지 캐시에서 방출될수록 메모리 확보 면에서 기존의 방식 대비 이득을 취할 수 있다.

ClamAV의 바이러스 스캔 모드와 함께 Redis라는 in-memory 데이터베이스 벤치마크를 수행하며 성능 변화를 측정하였다. Redis의 전반적인 성능 차이는 거의 없었으나, ClamAV에 PCAdvisor를 적용하지 않았을 때는 메모리 부족으로 인해 꼬리응답시간이 크게 발

생하는 것으로 파악되었다. Redis가 지원하는 다양한 operation에 대해 벤치마크를 수행한 결과, PCAdvisor를 적용하지 않았을 때는 적용한 때에 비해 그림 14과 같이 최대 21배에 달하는 큰 꼬리응답시간이 발생하는 것으로 측정되었다.

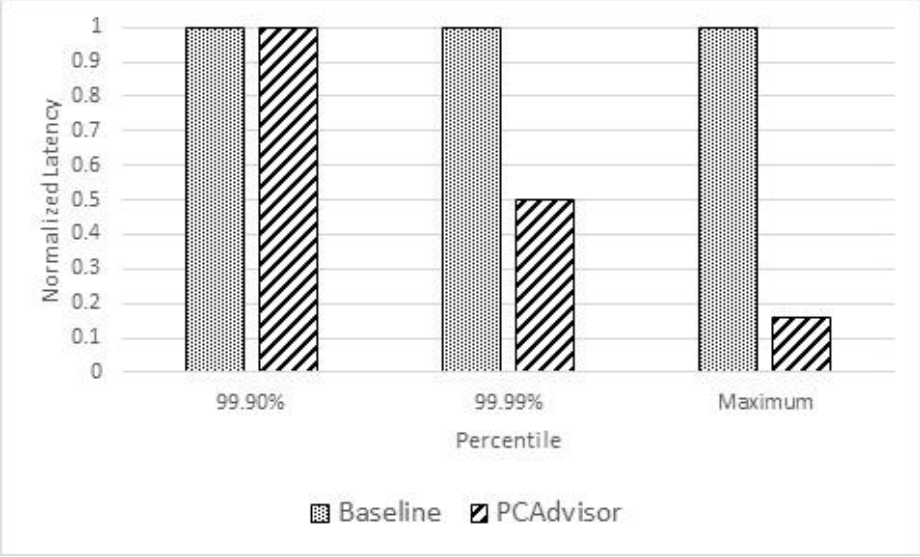

그림 15 ClamAV 가동 중 Redis SET 연산의 꼬리응답시간

PCAdvisor의 RANDOM 힌트를 통한 읽기 증폭 억제 효과를 확인하기 위해, RocksDB와 GraphChi 각각의 워크로드를 수행한 뒤 실제로 디스크에서 읽기를 요청한 데이터의 양을 비교하였다. 그 결과, 그림 16과 같이 GraphChi에서는 거의 변화가 없는 반면 RocksDB에서는 PCAdvisor 적용 이후 실제 읽은 데이터가 약 20%

정도 감소하였다. 대용량의 데이터를 읽고 쓰기에 임의 읽기 패턴이 잘 나타나지 않는 GraphChi와는 달리, RocksDB는 'Get' 메서드와 같은 임의 읽기 패턴을 보이는 경우가 비교적 많기에 PCAdvisor로부터 RANDOM 힌트를 통한 readahead 억제를 받았기 때문이다.

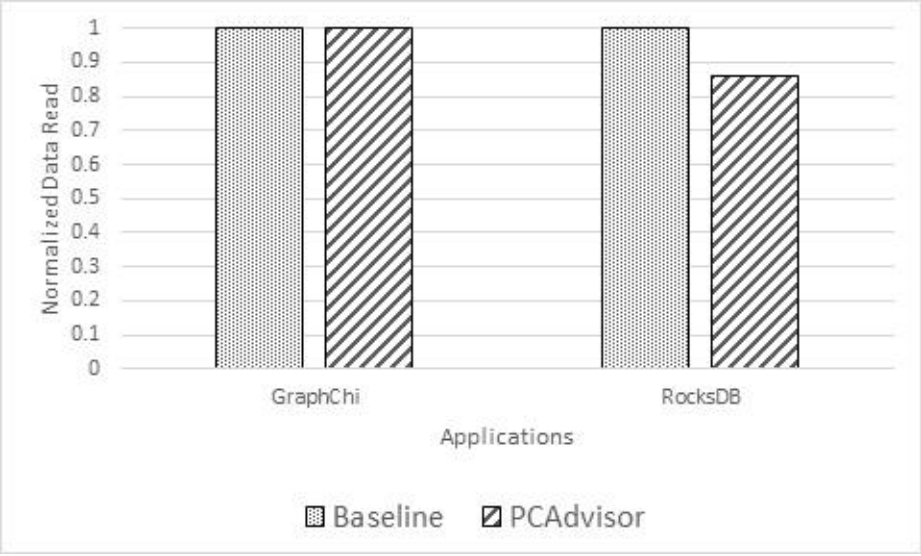

그림 16 PCAdvisor 적용 전후의 읽은 데이터의 양 비교

제 6 장 결 론

제 1 절 결론 및 향후 계획

본 논문에서는 응용이 I/O를 발생시키는 프로그램 컨텍스트를 기반으로 I/O를 발생시키는 위치와 특성을 간편하게 분석하는 PCStat을 제안하였으며, PCStat이 응용이 발생시키는 I/O를 적절하게 구분한다는 것을 보였다.

PCStat을 통해 분석한 자료를 토대로 응용의 I/O를 최적화하는 두 가지 방안 또한 함께 제안하였다. 첫째로 PCStat의 함수 심볼 변환을 통해 코드 레벨에서 fadvise를 적용하는 최적화 추천, 둘째로 이를 자동화한 커널 모듈 PCAdvisor를 활용한 I/O 최적화 자동화를 제시했다. 또한, PCStat을 통한 분석 결과를 바탕으로 각각의 프로그램 컨텍스트에서 발생하는 I/O의 패턴에 따라 페이지 캐시 공간을 더욱 효율적으로 사용할 수 있음을 실험을 통해 검증하였다.

본 논문에서 제안한 최적화 방안은 데이터의 연속성 여부와 재참조의 빈도 두 가지의 변수를 중점으로 각각의 프로그램 컨텍스트에 대한 최적화 기법을 제안하였다. 이후에는 위 두 가지 이외에도 각각의 프로그램 컨텍스트에서 가질 수 있는 고유한 패턴과 그 패턴에 대한 최적화 방안에 대한 연구를 진행할 계획이다.

이외에도 프로그램 컨텍스트는 다양한 방면에서 활용할 수 있는 여지가 많다. PCStat과 같은 정밀한 분석을 통해 자주 참조되는 데이터와 자주 참조되지 않을 데이터에 대해 알 수 있다면 스토리지 시스템에서는 다양한 기법을 통해 I/O 최적화의 힌트로 삼을 수 있다. 페이지 캐시 이외의 영역에서 프로그램 컨텍스트를 통한 최적화를 향후 연구로 남긴다.

참 고 문 헌

[1] C. L. P. Chen and C. Zhang, “Data-intensive Applications, Challenges, Techniques and Technologies: A Survey on Big Data,” Information Sciences, vol. 275, pp. 314-347, 2014.

[2] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware,” in Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operational Systems, pp. 37-48, 2012.

[3] Y. Zhang and S Swanson, “A Study of Application Performance with Non-volatile Main Memory,” in Proceedings of 31st Symposium on Mass Storage Systems and Technologies, pp. 1-10, 2015.

[4] D. Muntz and P. Honeyman, “Multi-level Caching in Distributed File Systems or Your Cache Ain’t Nuthin’ But Trash,” in Proceedings of the USENIX Winter 1992 Technical Conference, pp. 305-313, 1992.

[5] T. M. Wong and J. Wilkes, “My Cache or Yours? Making Storage More Exclusive,” in Proceedings of USENIX Annual Technical Conference, pp. 161-175, 2002.

[6] C. Gniady, A. Butt, and Y. Hu, “Program-Counter-based Pattern Classification in Buffer Caching,” in Proceedings of the Symposium on Operating System Design and Implementation, p. 27, 2004.

[7] F. Zhou, J. Behren, and E. Brewer, “AMP: Program Context Specific Buffer Caching,” in Proceedings of USENIX Annual Technical Conference, pp. 371-374, 2005.

[8] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, “Design Tradeoffs for SSD Performance,” in Proceedings of USENIX Annual Technical Conference, pp. 57-70,

2008.

[9] X. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, “Write Amplification Analysis in Flash-based Solid State Drives,” in Proceedings of the ACM International Systems and Storage Conference, p. 10, 2009.

[10] K. Ha and J. Kim, “A Program Context-aware Data Separation Technique for Reducing Garbage Collection Overhead in NAND Flash Memory,” in Proceedings of the International Workshop on Storage Network Architecture and Parallel I/O, 2011.

[11] L. Chang and T. Kuo, “An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems,” in Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 187-196, 2002.

[12] T. Kim, D. Hong, S. S. Hahn, M. Chun, S. Lee, J. Hwang, J. Lee, and J. Kim, “Fully Automatic Stream Management for Multi-streamed SSDs Using Program Contexts,” in Proceedings of the USENIX Conference on File and Storage Technologies, pp. 295-308, 2019.

[13] The Linux Kernel Module Programming Guide, https://www.tldp.org/LDP/lkmpg/2.6/html/lkmph.html

[14] fadvise - Linux Man Page, https://linux.die.net/man/2/fadvise

[15] D. Kroft, “Lockup-free Instruction Fetch/Prefetch Cache Organization,” in Proceedings of the 8th Annusl Symposium on Computer Architecture, pp. 81-87, 1981.

[16] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim, “A Low-overhead High-performance Unified Buffer Management Scheme that Exploits Sequential and Looping Reference,” in Proceedings of the Symposium on Operating System Design and Implementation, pp. 119-134, 2000.

[17] C. Lindig, “Random Testing of C Calling Converntions,” in

Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging, pp. 3-12, 2005.

[18] Spervisor Mode Access Prevention, https://lwn.net/Articles/517475

[19] A. Kyrola, G. Blelloch, and C. Guestrin, “GraphChi: Large-Scale Graph Computation on Just a PC,” in Proceedings of the Symposium on Operating Systems Design and Implementation, pp. 31-46, 2012.

[20] N. Megiddo and D. S. Modha, “Outperforming LRU with an Adaptive Replacement Cache Algorithm,” Computer, vol. 37, no. 4, pp. 58-65, 2004.

[21] 1. fio - Flexible I/O tester rev. 3.14, https://fio.readthedocs.io/en/latest/fio_doc.html

Abstract

Program Context based I/O Optimizations for Data Intensive Applications

Yongseok Jin
Department of Computer Science & Engineering
College of Engineering
The Graduate School
Seoul National University

Many kinds of data intensive applications are broadly utilized nowadays. These applications generate a lot of I/O such as analyzing a large amount of data, structuring the data and storing it in the storage, and the performance is greatly influenced by the speed of the I/O the system performs.

The operating system allocates a portion of main memory to the page cache to maximize the performance of file I/O by minimizing access to the storage device which is much lower in performance than main memory. However, since the size of memory is limited compared to the size of the storage device, it is very important to keep the data to be referenced to in future

and to export the data not to be referenced from the cache and to manage efficiently to improve the performance of the file I/O. However, it is impossible for the system to predict perfectly about which data will be referenced in the future and which data will not be. Thus, without I/O optimization at the application level, there is a clear limit to performance improvement.

In this thesis, we propose a method to automatically detect and analyze I/O characteristics based on I/O program contexts of which an application executes I/O. We propose a technique to automate the optimization recommendation to be applied to the program context in which I/O occurs. Through this, the application can provide various hints to the system that can not be grasped by the system itself, and the system actively reflects this information so that I/O can be performed faster and resources can be used more efficiently than before.

Keywords: I/O Optimization, Program Context, Page Cache, Buffer Management, Operating System
Student Number: 2017-25886