


연구논문/작품 중간보고서

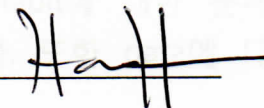
2018 학년도 제 2학기

제목	비동기 Prefetch를 이용한 Alluxio 최적화	○ 논문(√) 작품() ※해당란 체크
GitHub URL	https://github.com/dlehddms1115/alluxio_prefetch	
평가등급	지도교수 수정보완 사항	탐원 명단
A, B, F중 택1 (지도교수가 부여)	○ ○ A ○	이 동 은  (학번: 2015313106)

2018 년 9월 21일

지도교수 : 한 환 수

서명



■ 요약

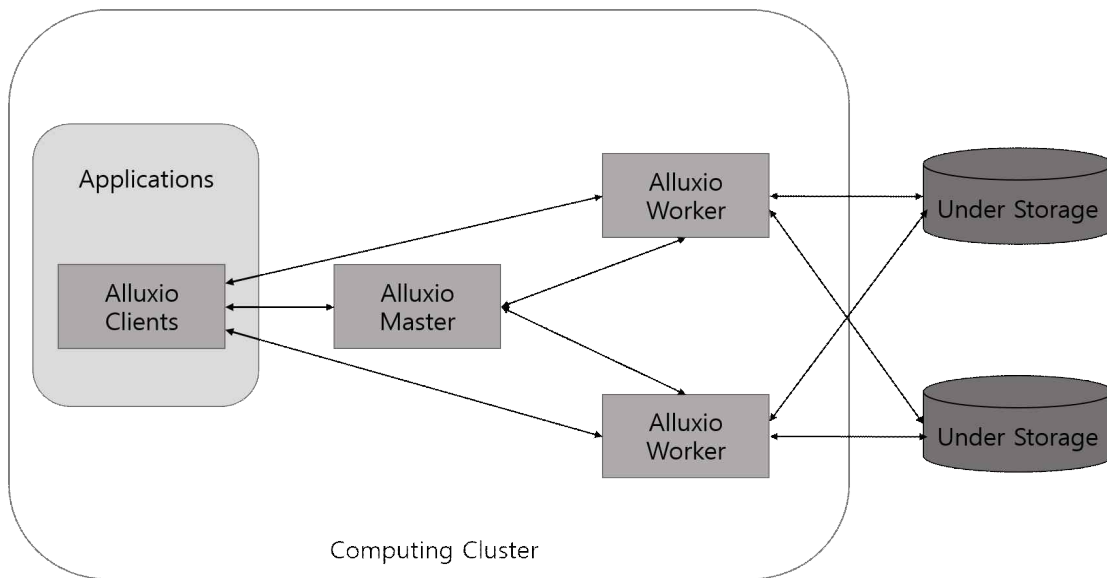
빅데이터 처리 프레임워크인 Apache Spark의 수요가 높아짐에 따라 처리 성능 향상에 대한 연구가 활발히 이루어지고 있다. 그 결과로 Alluxio라고 하는 중간 스토리지가 개발됨에 따라 공간 확장성이 뛰어나고 처리 속도가 빠른 cache layer의 역할이 더욱 중요해졌다. 그러나 여전히 메모리 자원이 제한된 상황에서 효율적인 데이터 저장을 위해 본 논문에서는 Spark Application의 요청에 따른 비동기적 prefetch를 활용하여 Alluxio를 최적화하는 기법을 제안하고자 한다.

■ 서론

가) 제안배경 및 필요성

Alluxio는 Big Data를 위한 빠른 가상 스토리지 이다. Tachyon으로도 알려진 Alluxio는 메모리 중심적 가상 분산 스토리지 오픈 소스 시스템으로, 데이터 접근을 메모리 속도로 가능하도록 해주는 안정적인 스토리지이다. Alluxio는 파일 API와 커맨드라인 toolkit을 사용해 host의 RAM 공간에 접근할 수 있도록 한다. 분산 프레임워크인 Apache Spark는 Alluxio를 사용해서 데이터 접근 속도를 증가시킬 수 있다. 분산 프레임워크인 Apache Spark의 경우 많은 양의 데이터를 처리하는 데에 성능이 매우 중요하다. 때문에, Spark의 성능을 도와주는 Alluxio를 데이터 prefetching을 통해서 더 좋은 성능을 내도록 하는 작업이 매우 중요하다.

지금은 Big Data의 시대이다. 우리는 수많은 데이터를 생성하며, 이러한 데이터는 컴퓨팅 중심 플랫폼에서 급진적인 변화를 일으킨다. 계속 증가하는 데이터를 처리하기 위해 cluster computing이 널리 보급되고 있으며 map reduce 및 dyrad와 같은 데이터 집약적인 프레임워크가 제안된다. 그러나 흔히 사용되는 HADOOP map reduce는 복잡한 multi-stage의 처리와 interactive하고 ad-hoc한 쿼리들을 잘 처리하지 못하고 느리다는 단점이 있다. 이러한 파일 기반 데이터 통신은 과도한 I/O를 생성하여, 높은 성능의 병목의 원인이 된다. 이런 단점을 해결하기 위해 HDFS를 거치지 않고 RAM을 사용하는 SPARK라는 분산 프레임워크가 제안되었다. SPARK는 RAM을 read-only로 사용하여 fault에 대한 문제를 해결하고, HDFS를 거칠 때보다 빠른 속도를 자랑한다. 또한 data에 대해

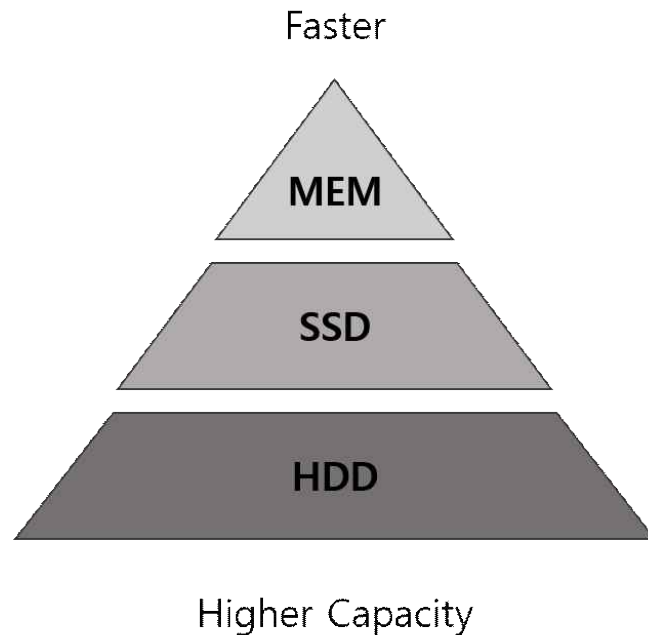


[그림 1] Alluxio 구조

수행되는 계산 순서를 나타낸 그래프인 DAG를 이용해 lazy-execution으로 효율적인 계산이 가능하게 한다. Lineage(계보)로 대강의 execution plan이 다 정해진 뒤 계산을 시행하므로 자원이 배치되는 상황 등을 미리 고려해서 효율적인 계산이 가능하다는 장점이 있다.

Apache Spark는 Alluxio를 사용하여 데이터 접근 속도를 증가시켜 전체적인 성능을 증가시킬 수 있다. Alluxio는 스토리지 시스템과 computation framework와 어플리케이션 사이에 위치하여 메모리 중심적 디자인을 구성한다. Alluxio는 computational framework와 분산 스토리지가 분리되었을 때 가장 큰 효율을 보인다. Alluxio는 데이터 access의 공통 인터페이스를 제공하는 동시에 빅데이터 어플리케이션의 속도를 최대로 증가시키는 역할을 한다. Alluxio가 under 스토리지 시스템들을 어플리케이션이 모르도록 하기 때문에 모든 under 스토리지가 모든 어플리케이션과 framework를 커버할 수 있다.

Alluxio는 tiered 스토리지를 제공하여 메모리뿐만이 아닌 다른 스토리지 타입도 커버할 수 있도록 한다. 현재 Alluxio는 [그림 2]와 같은 Mem, SSD, HDD의 3가지 유형의 스토리지를 제공한다. Alluxio의 높은 tier일수록 데이터를 읽어오는 속도가 빠르나, 용량이 적다. 많은 양의 데이터를 처리하는 데에 데이터를 읽어오는 속도는 전체 성능에 큰 영향을 끼치기 때문에 Alluxio내에서 사용할 데이터를 미리 높은 tier로 prefetch하는 것은 전체 어플리케이션의 성능 향상에 큰 도움이 될 것이다. Data를 읽는 시간이 오래 걸리는 SSD나 HDD에 저장된 데이터를



[그림 2] Alluxio Tiered Storage

어플리케이션과 비동기적으로 MEM으로 읽어오도록 코드를 구성하여 성능의 증가를 확인할 것이다.

나) 연구논문/작품의 목표

MEM에서 data를 읽어오는 속도와 SSD나 HDD에서 data를 읽어오는 속도는 현저하게 큰 차이가 난다. 때문에 자주 사용하는 data를 SSD나 HDD에서 MEM으로 prefetch를 하도록 수정한 Alluxio 코드를 통한 실험에서 기존의 Alluxio로 실험하였을 때의 성능 차이가 위에서 언급한 MEM에서 data를 읽어오는 것과 스토리지에서 data를 읽어오는 것과 같은 정도의 성능 차이가 나는 것을 기대한다.

Spark가 Alluxio를 사용함에 있어서 더 좋은 성능을 낼 수 있는 방법으로 성능최적화를 하기 위해 data prefetching 기법을 사용하는 것이다. Data prefetching 없이 Alluxio를 사용했을 때와의 성능 비교를 통해 Data prefetching이 Spark의 성능을 얼마나 증가시키는 지를 실험을 통해 알아낼 것이다.

Spark에서 application을 실행할 때 각 job에 필요한 data들을 Alluxio에 미리 prefetching을 해 놓음으로써, Spark의 성능을 최적화한다. Alluxio에서 data prefetching을 쉽게 구현할 수 있으면 좋겠지만, Alluxio가

data prefetching에 대한 염두 없이 제작된 시스템이기 때문에, Alluxio 코드 내에 data prefetching을 구현하는 데에 어려움이 예상된다는 한계가 있다. 하지만, 기존의 코드를 잘 분석하여 prefetching이 가능하도록 코드를 수정한다면 후에 자주 사용하는 data를 고르는 방식을 더 최적화하여 좋은 성능이 나올 수 있을 것으로 기대한다.

다) 연구논문/작품 전체 overview

빅데이터 분석 프레임워크인 Apache Spark의 경우 여러 분산 노드에 걸쳐서 많은 양의 데이터를 처리해야하기 때문에, 대부분의 Spark 애플리케이션들은 오랫동안 수행해야하는 작업을 처리한다. 따라서 Spark 애플리케이션은 작업 수행 성능이 매우 중요하다. 이를 위해 데이터 접근을 메모리 속도로 가능하도록 해주는 안정적인 스토리지인 Alluxio가 제안되었다. Alluxio는 저장장치로 메모리 외에도 SSD와 HDD를 사용할 수 있는데, SSD나 HDD에서 데이터를 가져올 경우 메모리 대비 아주 낮은 지연시간을 갖는다. 이때, 만약 Spark에서 어떤 일을 수행할지, 즉 어떤 데이터가 다음 입력으로 사용될지 알 수 있다면 입력으로 사용될 데이터를 SSD나 HDD에서 미리 Alluxio 메모리로 가져올 수 있다. 그러나 이러한 작업을 위해 몇 가지 해결해야하는 문제가 존재한다.

기존 Alluxio의 블록 복사 메커니즘은 동기적 복사로, 데이터가 필요할 때마다 SSD나 HDD에 저장된 데이터를 메모리로 복사해오기 때문에 성능 저하의 요인이 된다. 또한, 마스터가 워커로 명령할 수 있는 직접적인 통신 채널이 없어 클라이언트가 원하는 데이터 블록의 복사를 직접적으로 명령할 수 없다. 위의 문제들을 해결하기 위해 본 논문에서는 Alluxio Prefetching 기법을 제안한다. 제안하는 Prefetching 기법을 통해 실험한 결과 Spark 캐시 대비 최대 1.41배의 성능 향상을 보였다.

라) 섹션 소개

서론 세션에서는 과제의 제안배경 및 필요성과 연구논문/작품의 목표와 연구논문/작품 전체 Overview에 대해 설명한다. 관련연구 세션에서는 주제와 관련된 논문들과 그 내용에 대해 간략히 설명한다. 제안 작품 소개 세션에서는 이론적 배경과 함께 실제 작품의 구현 및 시스템 구성을 자세히 설명한다. 구현 및 결과분석 세션에서는 논문의 결과를 소개하고 결과를 분석한다. 결론 및 소감 세션에서는 결론과 연구논문을 진행하며 겪었던 소감에 대해 설명한다.

■ 관련연구

가) C. Chen, T. Hsia, Y. Huang, and S. Kuo, "Scheduling-Aware Data Prefetching for Data Processing Services in Cloud", 2017 IEEE International Conference on Advanced Information Networking and Applications, 2017

클라우드 컴퓨팅 서비스는 많은 양의 데이터들을 처리할 수 있는 유연한 스토리지 자원을 제공한다. 인메모리 기술에서는 데이터 처리 성능을 향상시키기 위해 자주 사용되는 데이터를 더 빠르고 비싼 스토리지에 저장한다. 데이터 Prefetching은 성능 증가를 위해 지연 시간이 적은 스토리지로 데이터를 이동시키는 데에 목적이 있다. 그러나 기존의 Prefetching 메커니즘은 같은 데이터를 자주 access하지 않는 특징을 갖는 애플리케이션의 경우를 고려하지 않는다. 또한 실행중인 다른 애플리케이션에 영향을 주지 않고 메모리 자원을 회수하는 방법이 없다는 문제가 있다. 본 연구에서는 클라우드 데이터 센터의 데이터 처리 서비스를 위한 Scheduling-Aware Data Prefetching(SADP) 메커니즘을 제안한다. SADP 메커니즘은 데이터 Prefetching 및 데이터 축출 방법을 포함한다. SADP 메커니즘은 다른 데이터 블록을 캐싱하기 위해 메모리에서 데이터를 제거한 다음 곧 사용될 데이터를 캐싱한다. 실제 실험을 통해 SADP 메커니즘이 효과적임을 확인하였다.

나) O. Yildiz, A. Zhuo, and S. Ibrahim, "Eley: On the Effectiveness of Burst Buffers for Big Data Processing in HPC Systems", 2017 IEEE International Conference on Cluster Computing, 2017

Burst Buffer(BB)는 HPC 시스템에서 데이터 전송 시간과 I/O 간섭을 줄이기 위한 효과적인 해결책이다. BB는 빅데이터 애플리케이션의 대용량 데이터 인풋과 HPC 시스템의 일급 객체인 HPC 애플리케이션의 성능보장을 고려해야 한다. 때문에, 빅데이터 애플리케이션을 처리하기 위해 BB를 확장하는 데에 많은 어려움이 있다. 기존의 BB는 빅데이터 애플리케이션의 중간 데이터에만 초점을 맞추었기 때문에 빅데이터와 HPC 애플리케이션 둘 모두의 성능 저하를 초래한다. 본 연구에서는 HPC 애플리케이션의 성능을 보장하면서 빅데이터 애플리케이션의 성능을 가속화하는 BB인 Eley를 제안한다. Eley는 빅데이터 애플리케이션의 성능 증가를 위해 애플리케이션의 인풋 데

이터를 가져와 컴퓨팅 노드에 저장하여 데이터 인풋을 읽는 시간을 단축시키는 Prefetching 기법을 사용한다. 또한 Eley는 HPC 시스템에서 독립적으로 실행되어 HPC 애플리케이션의 성능을 보장하는 전체 지연 연산자를 갖추고 있다. 실험을 통해 Eley가 HPC 애플리케이션의 성능을 보장하면서 빅 데이터 애플리케이션의 실행시간을 단축시킬 수 있다는 것을 확인하였다.

다) Y. Chen, H. Zhu, H. Jin, and X.-H. Sun, "Algorithm-level Feedback-controlled Adaptive data prefetcher: Accelerating data access for high-performance processors", *Parallel Comput.*, vol. 38, no. 10-11, pp. 553-551, Oct. 2012.

멀티 코어 아키텍처의 급속한 발전과 칩의 컴퓨팅 기능의 발전으로 인해 메모리 시스템 속도의 압력이 증가하고 있다. 많은 애플리케이션이 점점 더 많은 데이터를 사용하고 있다. 프로세서 속도가 아닌 데이터 access의 지연이 고성능 컴퓨팅의 주요 성능 병목 현상의 원인이 된다. 데이터 Prefetching은 애플리케이션의 데이터 access 속도를 높이고 컴퓨팅 속도와 access 속도 사이의 격차를 줄이는 효과적인 방법이다. 하지만 기존의 Prefetching 방법은 일반적으로 전력 소모 문제가 존재한다. 본 연구에서는 이러한 문제를 해결하기 위해 Algorithm-level Feedback-controlled Adaptive(AFA) 데이터 Prefetcher를 제안한다. AFA Prefetcher는 access 속도 증가를 위해 특별히 설계된 하드웨어 구조인 Data-Access History Cache를 사용한다. Data-Access History Cache는 알고리즘 레벨 adaptation을 제공하며 런타임동안 적절한 Prefetch 알고리즘에 동적으로 사용할 수 있다는 장점이 있다. SimpleScalar 시뮬레이터를 통해 AFA Prefetcher가 효과적이고 21개의 SPEC-CPU 벤치마크에 상당한 Instruction Per Cycle(IPC) 개선을 달성하였음을 확인하였다.

라) Y. Chen, S. Byna, and X. H. Sun, "Data access history cache and associated data prefetching mechanisms", in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007. SC '07, 2007, pp. 1-12

데이터 Prefetch는 프로세서와 메모리 간의 성능 격차를 줄이는 효과적인 방법이다. 컴퓨팅 성능이 메모리 성능보다 훨씬 빠르게 증가함에 따라 데이터 access 기록을 저장하고 데이터 access 지연 시간을 효과적으로 사용하기 위해 prefetching을 제공하는 전용 cache가 필요해졌다. 이에 따라 본 연구에서는 Data Access History Cache(DAHC)라는 새로운 cache 구조를 제

안하고 관련 Prefetch 메커니즘을 제안한다. DAHC는 instruction이나 데이터에 따라 캐싱을 하는 기존의 cache와 다르게 최근 참조 정보에 따라 캐싱을 한다. 이론적으로, DAHC는 기존의 잘 알려진 히스토리 기반의 여러 Prefetching 알고리즘들을 적용할 수 있다. 본 연구에서는 DAHC 설계 및 DAHC 기반 데이터 Prefetching 방법을 검증하고 성능 향상을 입증하기 위한 시뮬레이션 실험을 수행하였다. DAHC는 데이터 Prefetch의 효과를 얻는 실질적인 접근법을 제공하며 관련 Prefetching 메커니즘은 기존의 접근법보다 효과적임을 입증하였다.

■ 제안 작품 소개

가) 이론적 배경

1) Spark

Spark는 파일 기반 데이터 통신으로 과도한 I/O를 생성하여 성능이 저조한 Apache Hadoop의 단점을 보완하기 위해 메모리만을 사용하도록 제안된 분산 프레임워크이다. Spark는 데이터를 처리하는 계산 순서를 나타낸 그래프인 Directed Acyclic Graph(DAG)를 이용해 lazy-execution으로 효율적인 계산이 가능하다. RDD는 Spark 내에서 여러 분산 노드들에 저장되는 데이터의 집합을 말한다. Spark에는 3가지 작업이 있는데, 새로운 RDD를 생성하거나, 기존의 RDD를 변형하거나, RDD에서 연산을 호출하는 작업이다. RDD는 읽기 전용 데이터이기 때문에 한번 생성된 후로 업데이트가 불가능하다. Spark는 한 개의 RDD를 처리할 때 한 개의 Task를 하나의 Executor에 할당하여 데이터를 처리한다. Task가 처리하는 데이터의 기본 단위는 128MB의 블록이다.

2) Alluxio

Tachyon으로도 알려진 Alluxio는 메모리 중심적 가상 분산 스토리지 오픈 소스 시스템으로 캐시로 사용할 수 있는 계층적 스토리지를 제공한다. Alluxio는 모든 애플리케이션이 스토리지 시스템에 저장된 모든 데이터와 메모리 속도로 상호 작용이 가능하도록 해준다. Alluxio는 HDFS, S3, GlusterFS 등의 분산 파일 시스템과 transparent한 파일시스템을 제공한다. 이를 통해 Spark나 Hadoop 등의 기존 분산 처리 프레임워크가 별도의 애플리케이션 소스 코드 수정 없이 Alluxio

파일시스템에 접근할 수 있다.

Alluxio는 계층적인 스토리지를 제공하여 메모리뿐만이 아닌 다른 유형의 스토리지도 사용할 수 있도록 한다. 현재 Alluxio는 메모리, SSD, HDD의 3가지 유형의 스토리지를 제공한다. 계층이 높을수록 데이터를 읽어오는 속도가 빠르나, 용량이 적다. 계층화된 스토리지에 저장된 데이터는 Alluxio에서 관리하며 직접 접근은 불가능하다.

3) Alluxio Heartbeat 전송 구조

Alluxio는 마스터, 워커, 클라이언트의 세 가지의 요소들로 구성된다. 클라이언트는 Spark 클러스터와 Alluxio 서버의 통신을 담당한다. 마스터는 워커에서 어떤 명령을 실행할지를 응답하고, 워커는 살아있다는 상태 정보를 마스터로 전달한다. 클라이언트와 마스터는 RPC 통신을 통해 네트워크로 코드나 데이터를 전송하고 수신한다.

4) Spark의 데이터 요청에 따른 Alluxio의 동작

애플리케이션이 클라이언트를 통해 데이터를 요청하면 클라이언트는 마스터를 통해 데이터를 가진 워커의 위치를 확인한다. 클라이언트가 워커에게 데이터를 요청하면 워커는 Alluxio의 계층적 스토리지에서 데이터를 가져와 반환한다. Alluxio 공간이 부족해 데이터를 수용할 수 없게 되어 스토리지에서 캐시 미스가 발생하면 워커는 언더 스토리지에서 데이터를 가져와 클라이언트로 반환하고, Alluxio 스토리지에 데이터를 캐싱한다.

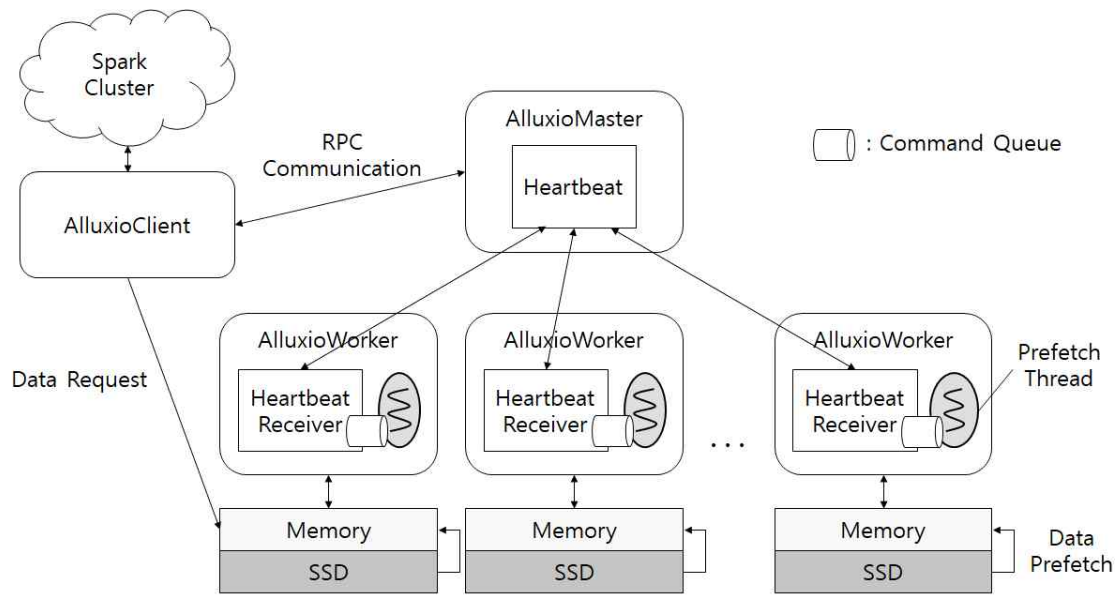
나) 시스템 구성

1) Prefetching 명령 전달

마스터에서 워커로 Prefetching 명령과 필요한 정보를 전달해주어야 하는데, 마스터가 워커로 명령할 수 있는 직접적인 통신 채널이 존재하지 않는다는 문제점이 있다. 이를 해결하기 위해 워커의 변경 정보를 마스터에 적용하는 Heartbeat를 통해 명령과 필요한 정보를 전달한다.

2) Alluxio Prefetching

Prefetching 과정은 앞으로 사용할 데이터 블록을 낮은 스토리지 계층에서 더 높은 스토리지 계층으로 복사 및 캐싱을 하는 과정이다. Alluxio는 데이터가 필요할 때마다 저장장치에서 메모리로 데이터를



[그림 3] 제안하는 Alluxio 개념도

복사하는 동기적 복사를 사용하는데, 이는 Spark와 같은 애플리케이션의 성능 저하의 원인이 된다. 복사 시 애플리케이션이 멈추지 않도록 비동기적으로 Prefetching을 수행한다.

3) Prefetching Thread

워커는 비동기적으로 블록 복사를 수행하기 위해 Prefetch Thread와 Command 큐를 생성한다. Prefetch Thread는 Command 큐에 작업이 할당될 때까지 대기하며, 큐에 작업이 할당되면 깨어난다. Command 큐는 FIFO로 작업을 수행한다. 큐에 작업이 할당되었지만 아직 차례가 돌아오지 않아 Prefetch가 진행되지 않은 상태에서 애플리케이션으로부터의 데이터 요청이 먼저 오는 경우에는 기존의 Alluxio에서의 데이터 요청에 따른 동작과 같게 동작한다.

다) 연구 내용

기존의 Alluxio 블록 복사 메커니즘을 Prefetching 기법으로 대체하여 프레임워크를 구성한다. Prefetching 기법을 통해 Alluxio는 Spark에서 어떤 데이터가 다음 입력으로 사용될지에 대한 정보를 얻어와 Alluxio 저장장치의 낮은 계층에 저장된 데이터들을 미리 Alluxio 메모리로 가져온다. 그림 3에서 보듯이, 클라이언트에서 Prefetch 명령을 받으면, Prefetch 해야 할 데이터 블록의 정보를 얻기 위해 마스터로 해당 명령을 전달한다. 이때, 마스터는 Heartbeat를 통해 워커로 Prefetch 명령을 전달한다. 마스터는 BlockMaster를

통해 Prefetch해야 할 블록 아이디 등의 메타데이터를 갱신하고, 워커에서는 갱신된 블록 아이디가 존재하면 데이터 Prefetch를 진행한다. 워커에서는 데이터 Prefetch를 진행할 때 애플리케이션이 멈추지 않도록 BlockMasterSync 모듈이 비동기적인 PrefetchThread와, Command 큐를 생성한다. 해당 Prefetch를 명령받은 블록 아이디들은 Command 큐로 관리된다. 큐는 생산자-소비자 모델로 BlockMasterSync 모듈이 생산자, PrefetchThread가 소비자에 해당한다. PrefetchThread는 Command 큐에 할당된 작업이 있을 때까지 대기하도록 설계되었으며, 워커로 Prefetch 명령이 들어오면 큐에 작업을 할당하고 PrefetchThread를 깨워 블록을 복사하는 작업을 실행한다. 해당 Prefetch 작업은 FIFO로 수행된다. 큐에 작업이 할당되었지만 아직 차례가 돌아오지 않아 Prefetch가 진행되지 않은 상태에서 애플리케이션으로부터 데이터 요청이 먼저 오는 경우에는 기존의 Alluxio에서 데이터 요청에 따른 동작과 같게 동작한다.

■ 구현 및 결과분석

가) 실험환경

[표 1] Spark-Alluxio 설정별 실험 환경

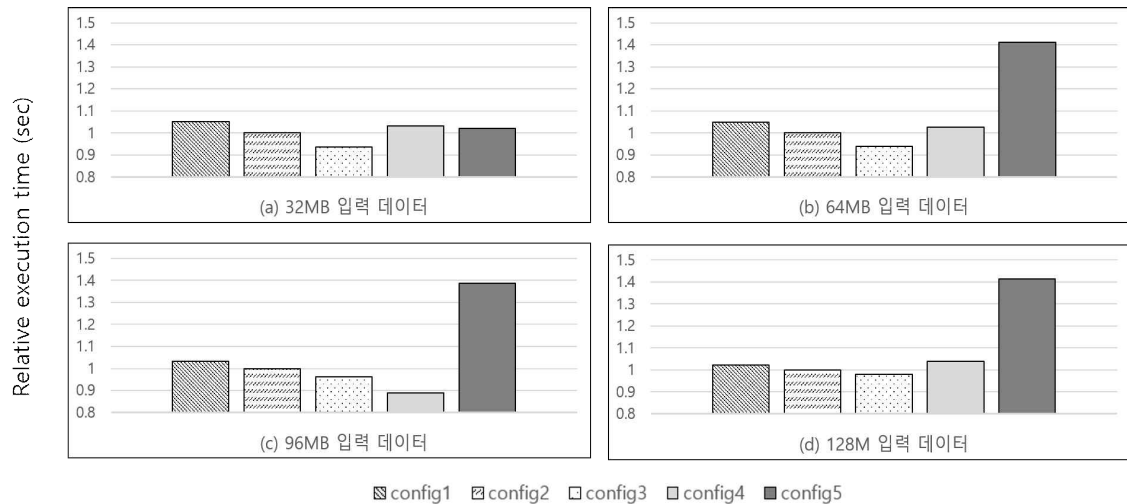
	Spark Mem	Alluxio Mem	Initial data	Prefetch	Cache
Config1	7GB	5GB	SSD	X	X
Config2				O	X
Config3			MEM	X	X
Config4	12GB	0GB	SSD	X	X
Config5				X	O

Prefetch의 유무에 따른 성능을 분석하기 위해 Spark 2.1.0 버전과 Alluxio 1.6.1 버전을 사용하여 [표 1]과 같은 환경에서 성능을 측정하였다. Spark 메모리와 Alluxio 메모리의 총량은 12GB로 통일하도록 구성되었다. Config1, 2, 3의 경우 보통 Spark는 총 메모리의 40% 가량을 캐시로 사용함을 고려하여 Alluxio 메모리를 5GB로 설계하였다. 실험 벤치마크로는 PageRank를 사용하였다. 이와 함께 데이터 크기별로 나타나는 성능 패턴을 분석하기 위해 32MB, 64MB, 96MB, 128MB의 4가지 입력 데이터를

사용하였다.

나) 실험 결과 및 분석

[그림 4]는 데이터 크기 및 Configuration에 따른 PageRank의 총 수행시간을 config2를 기준으로 상대적으로 비교한 그래프이다. 해당 그래프에 표기된 config1, 2, 3, 4, 5는 표 1에 제시한 configuration들이다.



[그림 4] 데이터 크기 및 configuration에 따른 상대 성능 그래프

1) Config 1, 2, 3의 비교 분석

전체적으로 입력 데이터의 크기에 상관없이 config2의 수행시간은 config1의 수행시간보다는 짧으나 config3의 수행시간보다는 긴 경향을 보인다. Config2의 수행시간이 config1보다 짧은 이유는 Prefetching 기법을 통해 SSD에서 읽어 와야 했던 데이터를 메모리로 복사해 데이터를 읽어오는 시간을 단축할 수 있기 때문이다. 하지만 Prefetching을 수행하는 시간에 대한 오버헤드가 존재하기 때문에 config2의 수행시간은 config3보다 길다.

2) 입력 데이터의 크기에 따른 config 1, 2의 비교 분석

입력 데이터의 크기가 커질수록 config1에 대한 config2의 상대 성능이 줄어드는 경향을 보인다. 이는 데이터 크기가 증가할수록 총 실행시간에서 실제 Spark 연산에 걸리는 시간이 차지하는 비율은 높게 증가하지만 Prefetching은 Spark가 데이터를 읽어오는 속도만을 증가시키기 때문이다.

3) 입력 데이터의 크기에 따른 config 4, 5의 비교 분석

Config4에서는 config5에 비해 32MB의 입력 데이터에서는 느리나, 64MB 이상의 입력 데이터에서는 빠른 경향을 보인다. Config5는 PageRank를 실행하기 전에 캐시 함수를 실행해 SSD에 저장되었던 데이터를 메모리에 저장하기 때문에 속도가 더 빨라야 한다. 하지만 PageRank의 특성상 중간에 생성되는 RDD가 많기 때문에 64MB 이상의 입력 데이터에서는 Spark의 캐시 공간이 부족해지고 캐시에서 빠진 중간 RDD를 다시 만들어야 하기 때문에 config4에서 config5에서보다 수행시간이 짧다.

4) 96MB에서의 config4 분석

96MB의 입력 데이터에서만 config4의 수행시간이 config3에 비해 짧은 경향을 보이는데, 이는 실험상의 오차로 보인다.

5) Spark 캐시 대비 분석

제안한 Prefetching 기업을 적용한 config2는 Spark 캐시를 실행한 config5 대비 최대 1.41배의 성능 향상을 보인다.

■ 결론 및 소감

가) 결론

Spark의 성능 증가를 위해 캐시 메모리로 Alluxio를 사용하는 것이 제안되었다. 하지만 Alluxio의 메모리의 크기 역시 한정적이며, 메모리 캐시 미스가 발생하여 Alluxio SSD에서 데이터를 읽어오는 경우 상당한 시간지연이 발생한다. 이에, 본 논문에서는 Alluxio Prefetching 기법을 제안하고, 성능의 증가를 분석하였다. 분석 결과, Alluxio에서 Prefetching 기법을 사용하면 데이터를 읽어오는 시간을 단축하여 Spark 캐시 대비 최대 1.41배의 성능 향상을 확인하였다.

나) 향후 연구

추후 본 연구에 이어 데이터가 필요한 순서에 따라 Prefetch 작업을 진행하도록 Command 큐의 입출력방식을 다르게 구성하여 최적의 성능을

발휘할 수 있는 프레임워크 연구하고자 한다. 또한 애플리케이션이나 입력 파일이 많을 때의 연구를 진행해보고자 한다.

다) 개선 사항

연구실 내 연구결과 공개발표회를 통해 교수님과 연구실 석사 및 박사 연구과정 학생들에게 연구결과에 대한 피드백을 받을 수 있었으며 이는 다음과 같다.

- 1) 사용한 PageRank에 대해 더 자세한 명시가 필요하다.
- 2) 사용한 Input 파일에 대한 더 자세한 명시가 필요하다.
- 3) 그래프의 configuration에 대해 config 1, 2, 3, 4, 5보다 더 명확한 명시가 필요하다.
- 4) 상대 성능 그래프에서 config2를 기준으로 하기보다는 config5를 기준으로 하는 것이 더 유의미해 보인다.
- 5) 실험 결과 분석의 원인에 대한 검증을 위해 더 정확한 실험이 필요하다.

최종 보고서 작성 시 위의 개선 사항을 참고할 예정이다.

라) 소감

본 연구를 진행하면서 실제 연구실에서 실험 및 연구를 어떤 방식으로 진행하고 수행하는지를 느끼고 체험해 볼 수 있었다. 또한, 논문을 작성해보며 논문의 목차를 구성하는 법, 요약물 작성하는 법, 서론을 작성하는 법, 논문에 첨부될 그림을 만드는 방법 등을 배울 수 있었다. 또한 저의 연구 능력을 강화하고, 실제 연구실이 어떻게 돌아가는지를 체험해 볼 수 있는 뜻깊은 기회였다.

■ 참고문헌

- [1] C. Chen, T. Hsia, Y. Huang, and S. Kuo, "Scheduling-Aware Data Prefetching for Data Processing Services in Cloud", 2017 IEEE International Conference on Advanced Information Networking and Applications, 2017

- [2] O. Yildiz, A. Zhuo, and S. Ibrahim, "Eley: On the Effectiveness of Burst Buffers for Big Data Processing in HPC Systems", 2017 IEEE International Conference on Cluster Computing, 2017
- [3] Y. Chen, H. Zhu, H. Jin, and X.-H. Sun, "Algorithm-level Feedback-controlled Adaptive data prefetcher: Accelerating data access for high-performance processors", *Parallel Comput.*, vol. 38, no. 10-11, pp. 553-551, Oct. 2012.
- [4] Y. Chen, S. Byna, and X. H. Sun, "Data access history cache and associated data prefetching mechanisms", in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007. SC '07, 2007, pp. 1-12
- [5] Apache Spark – Lightning-Fast Cluster Computing. [Online]. Available: <http://spark.apache.org/>
- [6] Alluxio – Open Source Memory Speed Virtual Distributed Storage. [Online]. Available: <http://www.alluxio.org/>
- [7] Apache Hadoop. [Online]. Available: <http://hadoop.apache.org>
- [8] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.
- [9] L. Page, S. Brin, R. Motwani and T. Winograd, "The PageRank citation ranking: Bringing order to the web", Stanford InfoLab, 1999.