리눅스 클러스터 파일 시스템을 위한 메타정보 캐쉬 기법*

홍 재 연**, 김 형 식
충 남 대 학 교 컴 퓨 터 과 학 과
{lh0701, hskim}@cs.cnu.ac.kr

A Caching Scheme of Meta-Information for the Linux Cluster File System

Jae-Youn Hong, Hyong-Shik Kim
Dept. of Computer Science, Chungnam National University

요 약

클러스터 파일 시스템은 고가용성(high availability) 및 결함내성(fault tolerance)을 만족하고 확장성이 뛰어나기 때문에 멀티미디어 서비스 등으로 활용범위를 넓혀왔다. 클러스터 시스템에서 일반적으로 제공되는 단일 시스템 이미지 (single system image) 기술은 저장된 위치에 관계없이 디렉토리나 파일에 접근할 수 있는 장점을 제공하지만 실제 저장된 위치에 따라 접근 시간의 편차가 발생된다.

본 논문에서는 파일이나 디렉토리에 관한 메타정보를 캐쉬에 저장함으로써 클러스터 파일 시스템에서의 접근 시간을 단축하기 위한 방법을 제안한다. 파일과 디렉토리의 접근 형태에 적합한 캐쉬 배치(placement) 기법과 재배치(replacement) 기법을 제시하고 캐쉬 일관성 유지를 위한 알고리즘을 보인다. 제안된 방법은 멀티미디어 서비스 등의 응용에서 효과적으로 접근시간을 단축할 수 있을 것으로 예상된다.

1. 서론

클러스터 파일 시스템은 저장용량 및 처리용량에 대한 확장성을 보장하고 장애가 발생하더라도 지속적인 서비스를 제공할 수 있을 뿐만 아니라 비용 대비 성능 비율에서 우수하기 때문에 멀티미디어 서비스 등의 특정 영역에 적합하다고 알려져 있다[1-2]. 대부분의 클러스터 파일 시스템은 기존의 범용 파일 시스템을 각 노드에서 운용하면서, 단일 시스템 이미지(single system image)를 제공하기 위한 기법을 이용하여 단일 파일 시스템처럼 보이도록 구성된다.

이때 단일 시스템 이미지 기술은 하드웨어 수준, 미들웨어 수준, 혹은 응용 프로그램 수준에서 파일 시스템에 대한 접근 요구를 실제 저장된 위치와 상관없이 일관된 방식으로 서비스하는데 핵심적인 역할을 한다. 그렇지만 실제 저장된 위치에 따라 접근 시간의 편차가 발생할 수 있게 되므로 저장된 노드와 상관없이 빠른 접근 속도를 보장하는 것이 중요한 문제중의 하나로 대두되었다.

클러스터 파일 시스템의 접근 시간은 파일 혹은 디렉토리에 관한 메타정보를 획득하는 시간과 실제 데이터 블록을 획득하는 시간으로 구성된다. 본 논문에서는 리눅스 운영체제 위에서 구현된 클러스터 파일 시스템[3]에서의 접근 시간을 단축하기 위하여 메타정보 캐쉬 기법을 제안한다. 특히 멀티미디어 서비스를 위한 클러스터 파일 시스템[3]의 경우 메타정보의 캐쉬 효과가 데이터 블록의 캐쉬 효과보다 훨씬 우월하다는 점에서 본 논문에서는 캐쉬 대상을 메타정보로 한정하기로 한다.

한편 일반적인 클러스터 파일 시스템은 고가용성(high availability)이나 결함내성(fault tolerance)을 위하여 메타정보 혹은 데이터 블록을 중복 저장하지만 본 논문에서는 설명의 편의를 위하여 메타정보의 중복 저장에 따른 영향은 무시하기로 한다.

2. 문제 정의

2.1 클러스터 파일 시스템

클러스터 파일 시스템에서는 파일의 입출력 시간을 단축하고 결함 허용 기능을 제공하기 위해서 단일 파일을 다중 노드에 분산하여 저장한다. 분산되어 있는 파일을 관리하기 용이하도록 하기 위해서 기존의 파일 시스템과 동일한 디렉토리 구조를 제공해야 한다. 이때, 파일, 디렉토리를 위한 메타 정보가 클러스터 파일 시스템을 구성하는 각각의 노드에 중복되지 않도록 분산 저장된다. 파일과 디렉토리를 위한 메타정보는 크기, 링크 정보 등으로 구성되며 여기서는 f-node와 d-node로 표현하기로 한다[3]. f-node와 d-node 정보는 노드별로 배열의 형태로 분산 저장되며, 파일에 대한 write/read 목적 연산을 위한 handle 정보를 저장하기 위해 handle array 저장구조를 사용한다.

예를 들면, 그림 1과 같이 클러스터 파일 시스템이 두 개의 노드로 구현될 경우 파일과 디렉토리에 대한 메타정보가 f-node와 d-node의 형태로 두 노드에 분산 저장된다. 파일에 대한 메타정보는 'file3'의 저장 형태와 같이 부모 디렉토리와 다른 노드에 저장될 수 있다. 'file3'의 경우 노드 0에서는 파일 자체에 대한 정보를 갖고, 노드 1에서는 해당 파일이 어느 노드의 어느 위치에 존재하는지에 대한 정보를 'dir1'에 저장한다.

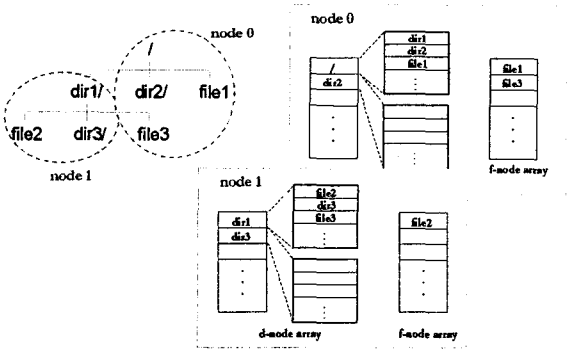

그림 1: 클러스터 파일 시스템의 메타정보 저장구조

파일을 open할 경우는 메타정보의 위치에 상관없이 open된 파일에 대한 메타정보는 open을 요청한 노드의 handle에 복사된다. 그리고 쓰기를 수행한 handle을 close할 때는 필요할 경우 handle의 정보를 이용하여 f-node를 갱신한다.

임의의 시점에서 한 파일에 대한 읽기는 동시에 여러 연산 수행이 가능하지만, 쓰기는 최대 한 연산만 가능하다. 쓰기를 동시에 수행할 경우 메타정보와 데이터 블록에 일관성 문제가 발생하므로 쓰기 목적 open은 최대 하나만 허용된다. 파일에 대한 식별자는 파일에 대한 정보가 저장된 노드번호와 f-node array의 인덱스 쌍으로 구성된다. 디렉토리의 경우도 마찬가지로 노드번호와 d-node array의 인덱스 쌍으로 구별된다.

2.2 접근지연문제

파일과 디렉토리에 접근하기 위해서는 데이터 블록의 위치와 상관없이 메타정보를 획득할 수 있어야 한다. 메타정보가 원격 노드에 저장되어 있는 경우는 로컬 저장 장치에 저장된 메타정보를 획득하는 시간에 비해 과도한 접근 시간이 필요하다. 해결방법으로 원격 노드의

* 이 논문은 정보통신부 지원 선도기반기술개발사업에 의하여 수행된 과제의 결과임.
** BK21 충남대학교 정보통신인력양성사업단의 지원을 받았음.

메타정보를 로컬 저장 장치에 중복 저장 할 수 있다. 그렇지만, 이러한 방법을 사용하면 메타정보에 대한 복사본이 여러 노드에 존재하게 되므로 중복 저장된 데이터 사이의 일관성 유지에 관한 문제가 발생한다.

3. 메타정보 캐쉬 기법

클러스터 파일 시스템에서 원격노드의 메타정보를 획득할 때 발생되는 지연 문제는 자주 참조되는 메타정보의 일부를 캐쉬에 저장함으로써 해결될 수 있다. 먼저 메타정보에 대한 접근 형태를 보면 다음과 같다.

3.1 메타정보에 대한 접근 형태

특정 메타정보를 획득하기 위해서는 f-node나 d-node에 대한 인덱스를 이용하거나 문자열로 표시된 경로 정보를 통하여 접근한다.

인덱스에 의한 접근

파일의 경우 메타정보가 저장된 노드번호와 f-node의 인덱스 쌍으로 접근할 수 있다. 인덱스를 이용하기 때문에 f-node array에서 원하는 f-node에 직접 접근할 수 있다. 디렉토리의 경우는 노드번호와 d-node의 인덱스 쌍을 이용한다.

경로에 의한 접근

디렉토리의 경우 경로를 이용하여 메타정보에 접근할 수 있다. 경로를 이용하여 특정 d-node에 접근하는 경우는 루트 디렉토리부터 원하는 경로까지 하위 디렉토리를 위한 d-node들을 탐색하는 과정을 반복해야 하므로 인덱스에 의한 접근보다 많은 시간이 소요된다. 상대패스가 주어질 때는 현재 작업 디렉토리와 조합하여 절대 경로로 변경한 후 루트 디렉토리부터 탐색할 수 있거나 현재 디렉토리에 해당하는 d-node부터 탐색할 수 있다.

3.2 캐쉬 엔트리 구조

메타정보를 캐쉬에 저장할 때 f-node의 정보와 함께 몇 가지 정보가 추가 저장된다. 캐쉬에 저장된 메타정보를 식별하기 위한 인덱스와 LRU 정보를 저장하기 위한 필드가 추가된다. f-node instance에는 4.2절에서 설명할 grant vector도 추가되어야 한다. d-node의 경우 경로를 통해서도 접근하는 경우가 있으므로 경로에 대한 정보도 추가 저장된다.

캐쉬를 고려하지 않을 경우 파일에 대한 읽기 카운트와 쓰기 카운트 정보는 f-node에 저장된다. 이때, 읽기 카운트/쓰기 카운트는 특정 파일에 대해 전체 노드에서 수행중인 읽기/쓰기 연산의 개수이다. 캐쉬가 사용될 경우는 일관성 문제를 발생시키지 않기 위하여 쓰기 카운트 정보는 홈 노드의 f-node에 그대로 두는 대신, 읽기 카운트 정보를 캐쉬에 저장한다. 홈 노드는 캐쉬가 아닌 fnode array에 f-node를 보유하고 있는 노드를 가리키며 특정 메타정보의 홈 노드는 수시로 바뀔 수 있다. 이때 캐쉬에 저장된 읽기 카운트는 사용되는 현재 노드에서 수행중인 읽기 연산의 개수를 저장한다. 홈 노드의 f-node에 저장된 쓰기 카운트는 전체 노드에서 수행중인 쓰기 연산의 개수를 저장하므로 클러스터를 구성하는 노드들 사이에 동시에 최대 하나의 쓰기 연산을 수행하도록 제한될 수 있기 때문에 일관성 문제를 회피할 수 있다.

예를 들어 그림 2와 같이 시스템이 구성되었고 노드 1의 캐쉬에 파일 (0,1)의 메타정보가 존재한다고 가정하자. 노드 1에서 파일 (0,1)에 대한 읽기 목적 open을 요청하면, 캐쉬의 읽기 카운트를 증가시키고 handle를 생성한다. 쓰기 목적 open인 경우는 캐쉬에 파일 (0,1)에 대한 메타정보가 존재함에도 불구하고 파일 (0,1)의 홈 노드인 노드 0으로 handle 요청을 전송한다. 이때 f-node (0,1)은 캐쉬로부터 제거된다.

3.3 캐쉬 엔트리 교체 방법

캐쉬 탐색에 실패한 경우는 홈 노드로부터 메타정보를 획득한 후 캐쉬에 저장한다. 이때, 캐쉬의 모든 엔트리가 사용 중이면 교체할 엔트리를 선정해야 한다. 본 논문에서는 가장 오래 전에 사용된 엔트리를 교체시키는 LRU(least recently used) 교체 방법을 사용하여 메타정보가 캐쉬의 어느 엔트리에나 적재될 수 있도록 함으로써 캐쉬 실패율을 최소화한다. LRU 교체 방식을 구현하기 위해 메타정보가 사용된 순서대로 참조 순서 리스트를 유지한다. 참조 순서 리스트의 처음(head)은 가장 최근에 사용된 엔트리를 나타내고, 끝(tail)은 가장 오래 전에 사용된 엔트리를 가리킨다. 따라서, 캐시에 새로운 메타정보를 삽입하고자 할 경우는 참조 순서 리스트의 처음에 삽입하며, 캐쉬의 메타정보가 재사용될 경우에도 재사용된 엔트리가 처음이 되도록 참조 순서 리스트를 재구성한다.

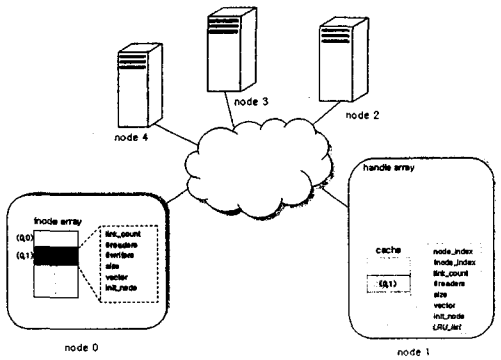

그림 2: 메타정보 캐쉬

3.4 캐쉬 엔트리 탐색 방법

3.3절에서 설명된 것처럼 캐쉬 엔트리 교체 방법으로 LRU를 사용함으로써 임의의 위치에 f-node와 d-node가 저장된다. 즉, 완전 연관(fully associative) 형태의 캐쉬 배치(placement) 방식을 사용하게 되므로, 캐쉬에 저장된 위치를 식별하기 위한 별도의 방법이 필요하다. 본 논문에서 제안된 캐쉬 기법은 인덱스 테이블과 해쉬 테이블을 사용하여 캐쉬 엔트리를 탐색하도록 고안되었다. 물론 캐쉬에 메타정보가 삽입되거나 삭제될 때마다 인덱스 테이블과 해쉬 테이블이 재구성된다.

인덱스 테이블

인덱스를 사용하여 캐쉬에 접근하려 할 경우 인덱스 테이블을 통해 메타정보가 저장된 위치를 알 수 있다. 기본적인 동작방식은 가상메모리의 페이지 테이블과 유사하다. 즉, 인덱스 테이블의 각 엔트리로 하여금 메타정보가 저장된 위치를 저장하도록 한다. 예를 들면, 그림 3에서와 같이 파일 (a, b)에 대한 메타정보가 저장된 캐쉬 엔트리의 위치를 알고 싶다면, 인덱스 테이블에서 (a,b)의 위치에 저장된 x을 이용한다. x가 -1이라면 메타정보가 캐쉬에 존재하지 않는 것을 의미한다.

해쉬 테이블

경로를 이용하여 캐쉬에 접근하려 할 경우는 인덱스 테이블을 사용할 수 없다. 대신, 경로 정보에 해쉬 함수를 적용시켜 획득한 값을 인덱스로 사용하는 테이블에 캐쉬 엔트리의 위치를 저장한다. 해쉬 함수를 통해 획득된 값이 동일하여 충돌(collision)이 발생하는 경우 그림과 같이 메타정보를 서로 연결하여 리스트를 구성한다. 그림 3에서 경로 '/c/d'에 대한 메타정보의 위치는 해쉬 함수를 통해 획득된 y에 저장되며, 동의어(synonym)가 존재할 경우 y가 가리키는 z부터 순차 탐색하여 해당 디렉토리의 메타정보의 위치를 획득할 수 있다.

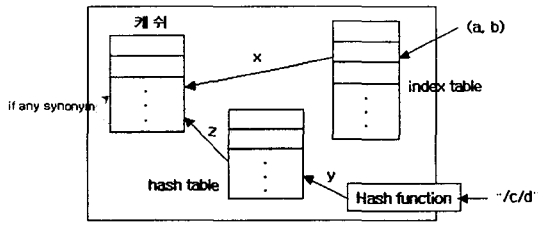

그림 3: 캐쉬 엔트리 탐색 방법

3.5 캐쉬 운용 알고리즘

캐쉬 엔트리 삽입

캐쉬 탐색 실패 시에는 홈 노드로부터 메타정보를 획득한 후 캐쉬에 삽입하고 인덱스 테이블에 반영한다. 삽입한 인덱스에 다른 메타정보가 존재하였다면, 이에 매핑되는 인덱스 테이블의 값을 -1로 변경한다. 그리고 삽입한 인덱스가 LRU 리스트의 처음이 되도록 LRU 정보를 재구성한다. d-node의 경우는 해쉬 테이블도 갱신하여야 한다.

캐쉬 엔트리 삭제

삭제된 엔트리의 메타정보는 더 이상 캐쉬에 존재하지 않으므로, 이를 반영하기 위해 인덱스 테이블의 값을 -1로 변경한다. d-node의 경우는 해쉬 테이블도 갱신하여야 한다.

4. 캐쉬 일관성 유지 기법

메타정보를 캐쉬에 저장할 경우 중복된 데이터가 여러 노드에 존재할 수 있기 때문에, 특정 노드에서 메타정보가 변경될 경우 중복된 데이터 사이에 불일치가 발생할 수 있다. 여기서는 이러한 문제를 해결하기 위한 캐쉬 일관성 유지 기법에 대하여 설명한다.

4.1 고려사항

f-node에 대한 읽기 연산은 일관성 문제가 발생하지 않지만 쓰기 연산에서는 handle이 갱신되므로 일관성 문제가 발생한다. 그리고 쓰기 연산은 데이터 블록의 내용을 변경하므로 쓰기 연산이 동시에 수행되지 않도록 보장하여야 한다. 또한 쓰기에 비해 읽기가 빈번히 발생하기 때문에 읽기 연산이 캐쉬의 성능에 영향을 미치지 않게 하는 것이 중요하다. 한편, d-node에 대해서는 생성, 삭제 등의 연산이 f-node에서의 쓰기 연산과 같은 일관성 문제를 발생시킨다. 다음은 f-node의 경우 캐쉬 일관성 유지를 위한 고려 사항이다.

1. 쓰기 목적 open은 캐쉬에 메타정보가 존재하여도 홈 노드에 직접 접근한다. 반면 읽기 목적 open는 캐쉬 탐색 실패일 경우 홈 노드로부터 메타정보를 획득하여 캐쉬에 저장한 후 open을 수행한다.

2. 홈 노드는 쓰기 목적 open을 수행하기 위해 모든 노드에서 open된 handle을 close했음을 보장해야 한다. 이를 위해 홈 노드는 모든 노드에 invalidation request를 전송한 후 모든 node로부터 허가(grant)를 수신할 때까지 기다린다.

3. 모든 노드는 홈 노드로부터 invalidation request를 수신할 수 있다. 이때 캐쉬에 해당 f-node가 존재하면 읽기 카운트가 0이 된 후에 홈 노드로 허가를 전송한다. 만약 캐쉬에 해당 f-node가 존재하지 않는다면 바로 허가를 전송할 수 있다.

4.2 주요 자료구조

- *open request list*: 홈 노드가 read 혹은 write 목적의 open 요청을 받았을 때, 모든 노드에서 메타정보가 invalidation되었음이 보장될 때까지 그 요청을 유지하기 위해 저장한다.
- *invalidation request list*: 모든 노드에서 읽기 카운트 > 0인 f-node에 대하여 invalidation 요청을 받았을 때 사용된다. 이때 즉시 invalidation 요청을 수행할 수 없으므로 읽기 카운트가 0이 될 때까지 저장하기 위해 사용된다. f-node에 대한 읽기 카운트가 0이 되었을 때 invalidation request list에 존재하면 캐쉬의 메타정보를 invalidation하고 홈 노드에 허가를 전송한다.
- f-node의 *grant vector*: 홈 노드의 f-node는 어떤 노드로부터 invalidation 허가를 받았는지의 여부를 표시하기 위하여 0 혹은 1로 구성된 벡터를 사용한다.

4.3 일관성 보장 알고리즘

읽기 목적 연산

읽기 목적 연산은 일관성 문제를 발생시키지 않지만, 임의의 시점에서 쓰기 목적 연산과 동시에 수행하지 않도록 보장해야 한다.

```
if (캐쉬에 f-node가 존재하지 않음) {
    홈 노드에 f-node 정보를 요구
    if (홈노드의 쓰기 카운트 == 1) {
        open request list에 요청 삽입
        쓰기 카운트가 0이 되면 요청을 삭제하고 f-node를 전달
    }
    f-node를 캐쉬에 삽입
}
캐쉬의 f-node instance를 handle에 복사
캐쉬의 f-node instance의 읽기 카운트 증가
```

쓰기 목적 연산

쓰기 목적 open은 홈 노드로 전달된다. 홈 노드가 쓰기 목적 open 요청을 받았을 때 수행되는 알고리즘은 아래와 같다.

```
open request list에 요청 삽입
if (쓰기 카운트 == 1) {
  쓰기 카운트가 0이 되면 요청을 삭제하고 f-node를 전달
}
else {
  모든 노드에 invalidation request 전송
  do {
      invalidation request를 수신할 때마다 grant vector 갱신
  } while (모든 노드로부터 허가를 수신하지 못함)
  요청을 삭제하고 f-node를 전달
}
f-node의 쓰기 카운트 증가
```

그림 4는 네 개의 노드로 구성된 클러스터 시스템에 대해 노드 1과 노드 2의 캐쉬에 (0,1)의 메타정보가 존재하는 경우를 보인다. 이때 노드 1의 읽기 카운트는 0이고 노드 2의 읽기 카운트는 2라고 가정하고, 설명의 편의를 위하여 f-node (0,1)의 홈 노드에 저장된 쓰기 카운트는 0이라고 가정하자. 노드 1이 쓰기 목적 open을 수행하는 경우 홈 노드인 노드 0으로 요청을 전송한다(①). 홈 노드는 open request list에 삽입(②)하고 쓰기 카운트가 0이므로 모든 노드에 invalidation request를 보낸다(③). 노드 1은 즉시 캐쉬의 메타정보를 invalidation하고 허가를 전송한다(⑤). 반면 노드 2는 읽기 카운트가 2이므로 invalidation request를 invalidation request list에 저장(④)한 후 읽기 카운트가 0이 된 후에 메타정보를 invalidation하고 허가를 전송한다(⑤). 노드 3은 메타정보가 캐쉬에 존재하지 않으므로 즉시 허가를 전송할 수 있다(⑤). 홈 노드는 허가를 수신할 때마다 f-node의 grant vector를 갱신하고(⑥), 모든 노드로부터 허가를 수신하면 노드 1에 f-node 정보를 전달한다.

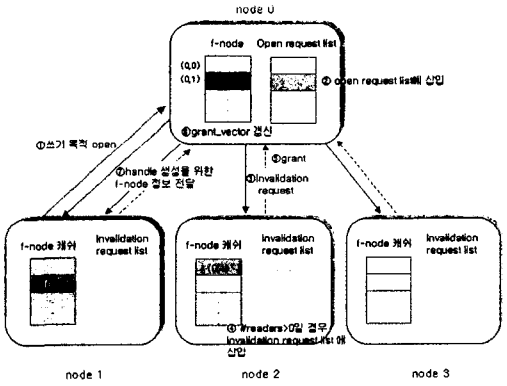

그림 4: 캐쉬 일관성 유지 기법

5. 요약 및 향후 연구 방향

본 논문에서는 리눅스 클러스터 파일 시스템을 위한 메타정보 캐쉬 기법을 제안하고 캐쉬 구조를 구현하기 위하여 필요한 고려사항들에 대하여 설명하였다. 클러스터 시스템이 다양한 응용 분야에 사용되고 있다는 점에서, 본 논문에서 제안된 캐쉬 구조는 멀티미디어 서비스 등의 영역에서 효과적으로 접근 시간을 단축할 수 있을 것으로 판단된다.

향후 캐쉬 기법을 확장하여 캐쉬 접근 시간 초과 문제를 해결할 수 있도록 개선할 계획이며, 데이터 블록에 대한 선반입(prefetching) 기법과 연동하는 방법에 대하여 연구할 예정이다. 또한 클러스터 파일 시스템을 이용하여 멀티미디어 서비스를 제공할 때를 대상으로, 본 논문에서 제안된 캐쉬 기법이 접근시간에 미치는 영향을 정량적으로 분석할 계획이다.

참고 문헌

[1] R. Buyya, *High Performance Cluster Computing, Volume 1: Architectures and Systems*, Prentice-Hall, 1999.

[2] G. F. Pfister, *In Search of Clusters, Prentice-Hall*, 1998.

[3] 강미연, 홍재연, 김형식, "효율적인 멀티미디어 서비스를 위한 리눅스 클러스터 파일 시스템", 한국정보과학회 2002년 춘계 학술발표회, pp. 652-654.