클러스터 파일 시스템의 메타데이터 서버를 위한 내용 기반 부하 분산 알고리즘

장 준 호[†] · 한 세 영[††] · 박 성 용[†††]

요 약

클러스터 파일 시스템의 성능을 결정짓는 주요 요소 중 하나는 메타 데이터 서비스의 성능이다. 본 논문에서는 메타 데이터 연산의 종류에 따라 적절한 메타 데이터 서버에게 클라이언트의 요청을 동적으로 분배할 수 있는 내용 기반의 부하 분산 알고리즘을 제안한다. 이 알고리즘에서는 메타 데이터 서버 사이에 데이터의 이동을 유발 시키는 대신 메타 데이터를 복제하고 수정 메시지를 로깅하게 함으로써, 기존의 알고리즘에 비해 클라이언트의 요청을 서버들에게 균등하게 분배하여 응답 지연 시간을 현저히 줄일 수 있었다.

키워드 : 클러스터 파일 시스템, 부하 분산, 메타데이터 서버

A Content-based Load Balancing Algorithm for Metadata Servers in Cluster File System

Junho Jang[†] · Saeyoung Han[††] · Sungyong Park[†††]

ABSTRACT

A metadata service is one of the important factors to affect the performance of cluster file systems. We propose a content-based load balancing algorithm that dynamically distributes client requests to appropriate metadata servers based on the types of metadata operations. By replicating metadatas and logging update messages in each server, rather than moving metadatas across servers, we significantly reduced the response time and evenly distributed client's requests among metadata servers.

Key Words : Cluster File System, Load Balancing, Metadata Server

1. 서 론

고성능 컴퓨터에 대한 벤치마크 기관인 SPEC의 연구 결과에 따르면 클러스터 파일 시스템에 대한 트래픽의 60% 이상이 메타 데이터에 대한 요청(request)에 집중되고 있다[1]. 메타 데이터는 파일의 크기, 생성 날짜, 소유자, 접근 권한, 물리적 저장 위치 등의 정보를 보관하고 있는 10 KB 미만의 구조체이다. 그러나 파일 시스템의 클라이언트가 요청하는 파일이나 디렉터리에 대한 생성, 읽기, 정보 수정 등을 수행하기 위해서는 메타 데이터에 대한 접근이 필수적으로 선행되어야 하기 때문에, 실제 파일의 데이터에 비해 차지하는 크기는 작지만 접근 횟수는 빈번하다.

초기 클러스터 파일 시스템에서는 메타 데이터를 파일 데이터와 동일한 물리적 공간에 분할하여 저장하는 방법을 사용 하였으나, 디스크와 같은 디바이스가 파일 서비스뿐만 아니라 메타 데이터 서비스를 함께 수행해야 하기 때문에 파일 서비스에 대한 성능 저하를 가져왔다[2, 3]. 따라서 성능 향상을 위해 메타 데이터의 저장 공간을 데이터 저장 공간과 분리하고 독립된 메타 데이터 서버가 서비스 하도록 하는 클러스터 파일 시스템 구축 기법이 제시되었다[4]. 그러나 이와 같은 기법은 메타 데이터 서버의 장애 발생이 전체 파일 시스템의 장애를 가져올 수 있고, 모든 메타 데이터 서비스 요청이 한 서버로 집중됨에 따라 확장성의 측면에서 한계가 있다.

최근에는 클러스터 파일 시스템의 메타 데이터 서비스를 위하여 여러 대의 서버를 이용하여 클러스터를 구축하려는 시도를 하고 있다 [5-8]. 공유 저장 장치나 고성능 네트워크를 이용하여 전체 메타 데이터 공간을 동시에 사용하거나 메타 데이터를 복제 (replication)하는 대칭형(symmetric) 클러스터링 기법도 시도되었으나, 보다 성능이 뛰어난 비대칭

※ 본 연구는 서강대학교 산업기술연구소의 지원으로 수행되었음.
† 정 회 원 : 삼성전자 메모리 사업부
†† 정 회 원 : 서강대학교 컴퓨터학과 박사과정
††† 정 회 원 : 서강대학교 컴퓨터학과 부교수
논문접수 : 2005년 8월 23일, 심사완료 : 2006년 6월 1일

(asymmetric) 클러스터링 기법이 현재 메타 데이터 서버 연구의 주류를 이루고 있다. 비대칭 클러스터링 기법에서는 메타 데이터 공간을 분할하여 각 메타 데이터 서버에게 담당 영역을 할당한다. 메타 데이터에 대한 접근이 특정 디렉터리에 집중되거나 메타 데이터 연산마다 다른 계산량을 가지는 파일 시스템 연산의 특성상 서버 간 부하의 불균형과 과부하가 발생이 불가피하다. 또한 메타 데이터 서버의 클러스터링을 구축하고 유지하기 위한 추가적인 네트워크 트래픽의 발생을 초래하기도 한다[6, 8]. 따라서 클러스터 파일 시스템의 성능을 결정짓는 중요 요소인 메타 데이터 서비스의 성능을 일정하게 유지하기 위해서는 각 서버들에게 메타 데이터를 공정하게 배분하는 합리적인 부하 분산 기법이 필수적이다.

비대칭 메타데이터 클러스터에서 각 서버에게 할당될 메타 데이터를 분할하는 방법은 디렉터리 구조에 따른 분할(Directory Sub-Tree Partitioning), 단순 해싱을 이용한 부할(Pure Hashing), 그리고 지연 정책을 수반하는 혼합 분할 방식(Lazy Hybrid)으로 나눌 수 있다. 디렉터리 구조에 따라 메타 데이터 공간을 분할하는 방식과 단순 해싱을 이용하는 방식은 파일 시스템 연산의 종류에 따라 과도한 부하와 네트워크 트래픽을 유발하기도 하므로, 그 두 방식의 장점을 모으고 단점을 보완한 혼합 분할 방식이 연구되었다. 대표적인 혼합 분할 방식인 *LH3*의 경우, 파일의 전체 경로를 입력 값으로 하는 해시 함수를 사용하여 메타 데이터를 각 메타 데이터 서버에 분산하여 저장하는데, 모든 클라이언트에 메타 데이터 검색 테이블(Metadata Lookup Table)을 구축하고, 모든 메타 데이터 서버에 디렉터리 계층 구조를 유지함으로써 복잡한 파일 시스템 연산 수행 시 발생하는 과부하를 해결하였다[6].

본 논문에서 제안하는 내용 기반의 동적 부하 분산 알고리즘은 요청 분배 서버가 클라이언트의 요청 내용을 분석하고 요청하는 연산에 따라 해당 메타 데이터 서버에게 요청을 전달하는 내용 기반 부하 분산 정책을 사용한다. 또한 모든 서버에서 쓰기 연산을 동시 수행하게 하고 로그 정보를 기록하고 추후 실제 데이터 이동을 수행하게 함으로써 *LH3*에서 해결하지 못했던 과도한 브로드캐스트 발생 및 메타 데이터 이동에 따른 성능 저하를 해결하였다. 또한 같은 메타 데이터에 대한 간접 검색 테이블을 모든 서버와 요청 분배 서버에서 유지하여, 요청 분배 서버가 각 메타 데이터 서버의 부하를 실시간으로 반영하여 클라이언트의 요청을 공평하게 배분할 수 있도록 함으로써, 기존의 *Vesta*나 *LH3*에 비해 월등히 좋은 연산 성능을 나타낸다.

본 논문의 나머지 구성은 다음과 같다. 2장에서 비대칭 메타 데이터 서버 클러스터를 이용한 메타 데이터 관리를 위한 기존 연구와 문제점을 기술하고 3장에서는 메타 데이터 서버 클러스터에서 내용을 기반으로 하는 동적인 요청 분산을 이용하여 메타 데이터 서비스의 성능을 유지하도록 하는 알고리즘을 구체적으로 제시한다. 4장에서는 제시하는 알고리즘의 성능을 시뮬레이션을 통해 측정, 평가하고 마지막으로 5장에서 결론을 내린다.

2. 관련 연구

비대칭 메타 데이터 서버 클러스터에서 각 서버로 할당할 메타 데이터를 분할하는 방법은, 디렉터리 구조에 따른 메타 데이터의 분할(Directory Sub-Tree Partitioning), 단순 해싱을 이용한 메타 데이터의 분할(Pure Hashing), 그리고 지연 정책을 수반하는 혼합 분할 방식(Lazy Hybrid)으로 분류할 수 있다.

2.1 디렉터리 구조에 따른 메타 데이터의 분할

디렉터리 구조에 따라 메타 데이터를 분할하는 경우 한 디렉터리를 담당하는 메타 데이터 서버가 그 하위 파일들과 하위 디렉터리들을 같이 관리한다. 한 파일에 대한 경로를 탐색하거나 파일의 접근 권한을 조회할 때 그 경로 상의 각 디렉터리에 대해 메타 데이터를 서버에게 요청해야하는데, 이 경우 여러 개의 서버를 접속하지 않아도 되므로 효율적이라고 할 수 있다. 그러나 특정 파일이나 디렉터리에 클라이언트의 요청이 집중될 경우, 그 메타 데이터를 저장하고 있는 서버로 부하가 집중되므로 메타 데이터 서버들 사이에 효율적인 부하의 분배가 불가능하다.

2.2 단순 해싱을 이용한 메타 데이터의 분할

메타 데이터 서버 사이에 부하를 보다 효율적으로 분배하기 위한 방법으로 단순 해싱 기법이 제안되었다. 이 방법은 각 파일과 디렉터리의 이름이나 ID, 또는 다른 속성을 입력 값으로 하여 해시 함수를 계산하고, 그 결과 값에 해당하는 메타 데이터 서버로 파일과 디렉터리의 메타 데이터를 할당하므로, 디렉터리 구조에 따라 분할 방법에 비해 훨씬 공정하게 메타 데이터 서버들 사이에 부하를 분산시킬 수 있다.

단순 해싱 방법을 사용하는 파일 시스템으로는 *Vesta* 파일 시스템이 있다[8]. 이 파일 시스템에서는 요청하는 파일의 전체 경로를 해시 함수의 입력 값으로 사용하고, 48비트의 출력 값을 생성한다. 이 출력 값 중 상위 16 비트와 하위 16비트는 각각 다시 한 번 해시 함수를 통해 변환을 거치게 되고, 상위 16 비트의 결과 값은 메타 데이터가 저장된 입출력 노드의 위치를, 하위 16비트의 결과 값은 그 입출력 노드 내에서 메타 데이터 엔트리(entry)의 위치를 나타낸다[8]. 이 때, 각 입출력 노드는 파일의 64비트 식별자(Internal ID)와 파일의 이름, 그리고 분산저장 관련 정보 등의 정보를 저장하기 위하여 객체 테이블(Object Table)을 구축하고 관리하는데, 64비트 식별자에는 서로 다른 파일이 같은 48비트 결과 값을 산출하는 경우에 대비하여 7비트의 추가 필드가 포함된다.

해시 함수의 사용은 파일 경로 상의 모든 조상 디렉터리들에 대한 메타 데이터를 요청할 필요 없이 해당 파일의 메타 데이터를 직접 요청할 수 있지만, 디렉터리 관리 및 파일의 접근 권한 조회에 어려움이 있다. *Vesta*의 경우 디렉터리 구조를 관리하기 위해서 *Xref*라는 구조체에 해당 파일과 다른 *Xref*들의 내부 식별자 리스트를 저장하는데, 이는

사용자 편의를 위한 파일의 묶음만을 제공할 뿐 디렉터리 권한을 고려한 파일 접근 권한에 대한 관리는 제공하지 못한다. 또한 이 해싱을 이용한 메타 데이터 분할 방법에서 디렉터리 이름이 변경되면 해시 함수의 결과 값이 달라지므로 해당 디렉터리 뿐 아니라 모든 하위 파일과 디렉터리의 메타 데이터가 재배치되어야 하고, 더욱이 입출력 노드의 추가나 삭제와 같이 하드웨어 설정이 변경될 경우 모든 메타 데이터가 재배치되어야 하므로 상당한 과부하를 발생시키게 된다. 또한, 한 디렉터리의 내용을 조회 할 경우에도 여러 번의 해시 함수의 계산과 여러 입출력 서버로의 접근이 필요하므로 클라이언트의 요청에 대한 응답 시간이 떨어지게 된다.

2.3 지연 정책을 수반하는 혼합 분할 방식

단순 해싱을 이용한 메타 데이터 분할 방법에서의 과부하 발생과 응답시간 지연을 줄이기 위해서 *LH3*와 같은 지연 정책을 수반하는 혼합 분할 방식이 제안되었다[6]. *LH3*에서도 역시 파일의 전체 경로를 입력 값으로 하는 해시 함수의 결과 값을 이용하여 메타 데이터를 다수의 메타 데이터 서버에 분산하여 저장한다. 그러나 각 클라이언트는 메타 데이터 검색 테이블(Metadata Lookup Table)을 구축 하고 유지함으로써, 해시 함수의 결과 값을 직접적으로 메타 데이터 서버를 선택하는데 사용하지 않고 검색 테이블의 인덱스 값으로 사용하여 해당 메타 데이터 서버를 선택한다. 이를 통해 서버의 추가나 삭제가 발생할 경우 메타 데이터를 재배치하는 과부하 없이 메타 데이터 검색 테이블의 조정만으로 하드웨어 설정을 변경할 수 있다. 한편 각 메타 데이터 서버들은 모든 메타 데이터의 디렉터리 계층 구조에 대한 정보를 저장하고 유지함으로써 디렉터리의 내용 조회 시 발생되는 응답 시간 지연을 방지한다.

*LH3*는 또한 글로벌 로깅(Global Logging)과 지연 정책(Lazy Update Policy)를 사용하여 디렉터리 연산의 성능을 효율적으로 향상시킨다. 즉, 한 디렉터리의 정보 수정 요청이 발생하면 그 디렉터리를 담당하는 메타 데이터 서버는 클라이언트의 요청을 로깅한 후 바로 응답을 클라이언트에게 보내고, 다른 메타 데이터 서버들에게도 그 수정 내용을 브로드케스트 하여 디렉터리에 대한 수정된 정보를 공유한다. 디렉터리의 정보 수정에 따른 실질적인 메타 데이터의 이동은 추후에 수행한다. 즉, 메타 데이터 정보에 대한 클라이언트의 요청을 받게 되면, 해당 메타 데이터 서버는 자신의 로그 내용을 검색하여 메타 데이터의 실제 위치를 파악하고, 메타 데이터가 다른 메타 데이터 서버에 있으면서 현재 서버로의 이동이 아직 일어나지 않았다면 메타 데이터의 이동을 먼저 수행한 후 클라이언트의 요청을 처리한다. 클라이언트의 요청을 받기 전이라도 후위 작업(Background Job)에 의해서 메타 데이터 서버들 간의 메타 데이터의 이동을 수행시키기도 한다. 이런 방법을 통해 디렉터리 내용 변경에 따른 대량의 메타 데이터 이동이나 삭제, 또는 동시에 수행되는 메타 데이터 검색 연산에 의해 지연되는 응답 시간 지연 등을 방지한다.

*LH3*의 경우, 메타 데이터의 저장 공간을 공평하게 사용할 뿐 아니라 단순 해싱을 이용한 분할 방법에서의 여러 가지 문제점을 해결한다. 그러나 이 방법으로도 메타 데이터 서버들 사이의 부하를 실시간으로 조정하지 못한다. 더욱이 *LH3*는 모든 메타 데이터 서버의 로깅과 주기적인 브로드케스트에 의한 로그 정보의 공유에 의존한 메타 데이터 관리 방법이므로, 추가적인 네트워크의 부하를 감수해야 한다. 또한 메타 데이터의 검색 연산을 위해서 항상 로그 정보의 추가적인 검색이 필요하고, 이는 실제 메타 데이터를 보관하고 있는 서버로 클라이언트 요청을 집중될 경우 응답 시간의 지연을 야기한다. 마지막으로 글로벌 로깅 기법은 메타 데이터 서버가 삭제되는 경우에는 적용할 수 없다.

3. 내용 기반의 동적 부하 분산을 지원하는 메타 데이터 관리 기법

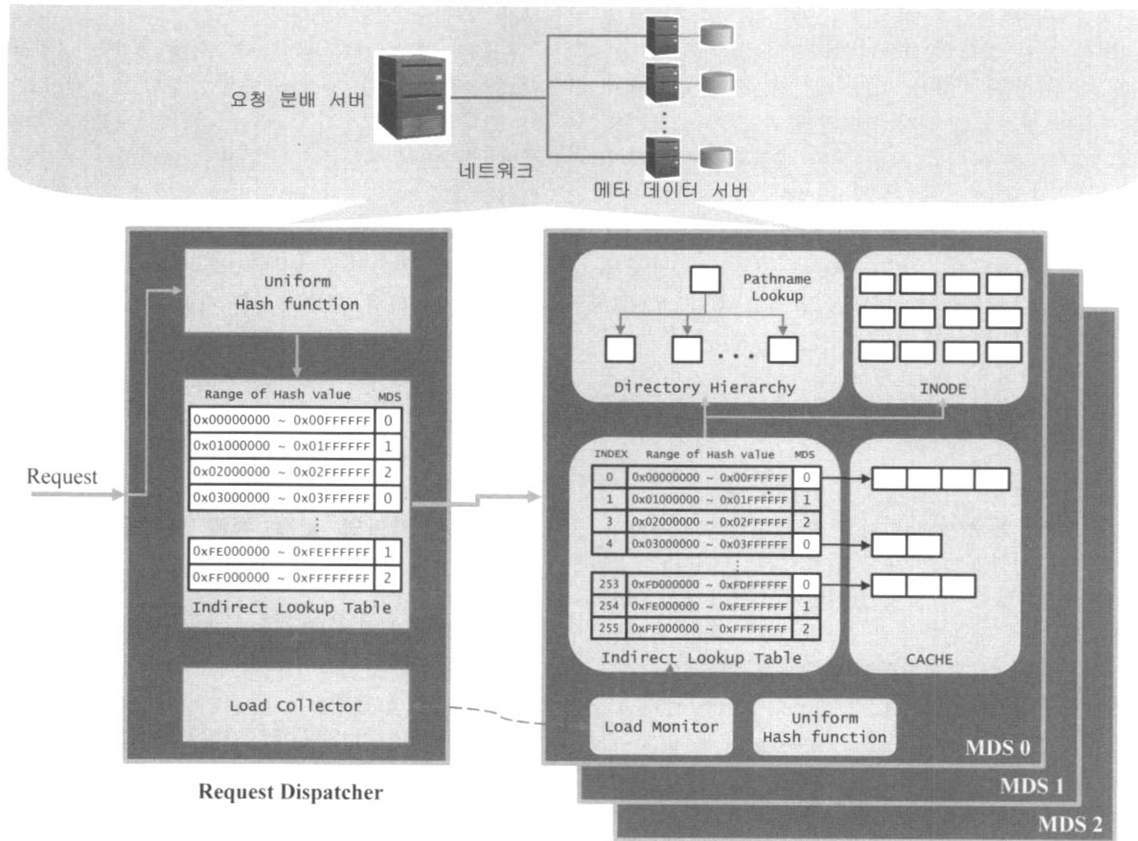
본 장에서는 지연 정책을 수반한 메타 데이터의 혼합 분할 방식에서의 단점을 보완하여, 메타 데이터 서버들의 부하 변동 상황을 동적으로 반영하여 클라이언트의 메타 데이터 연산 요청을 효율적으로 분산시키는 방법을 제안하고자 한다. 메타 데이터 서버 클러스터를 위한 전용 요청 분배기를 사용하여 클라이언트의 각 요청에 대한 내용을 파악하고, 적절한 메타 데이터 서버에게 그 요청을 전달하여 처리하도록 하는 내용 기반의 분산 알고리즘을 사용하고, 더 나아가 각 메타 데이터 서버들의 부하를 동적으로 반영하여 요청을 분배 하도록 하는 동적 부하 분산 알고리듬을 제안하고자 한다.

3.1 메타 데이터 서버 클러스터의 구성

제안하는 비대칭 메타 데이터 서버 클러스터는 메타 데이터 서버, 전용 요청 분배 서버, 해시 함수, 간접 검색 테이블(Indirect Lookup Table), 로컬 캐시, 그리고 디렉터리 구성 정보로 구성된다((그림 1) 참조).

클라이언트의 메타 데이터 연산 요청은 먼저 요청 분배 서버로 전달되고, 요청 분배 서버는 요청된 파일의 정보를 입력 값으로 한 해시 함수의 결과 값을 사용해서 간접 검색 테이블에서 해당 파일의 메타 데이터를 저장하고 있는 메타 데이터 서버를 결정한다. 요청 분배 서버는 요청된 연산의 종류에 따라 결정된 메타 데이터 서버 또는 클러스터의 모든 메타 데이터 서버에게 클라이언트의 요청을 전달하여 적절한 연산이 수행되도록 한다. 구체적으로 연산이 수행되는 과정은 3.2 절에서 자세히 설명하고자 한다.

이와 같이 동작하기 위해서 각 메타 데이터 서버들과 요청 분배 서버는 동일한 해시 함수를 사용해야 하고, 동일한 내용의 간접 검색 테이블을 유지할 수 있어야 한다. 또한 메타 데이터 서비스 성능의 향상을 위하여 각 메타 데이터 서버에 메타 데이터 정보의 캐시(cache)를 저장하고, 파일 경로 검색 또는 파일 권한 검사 등 디렉터리 계층 구조 관



(그림 1) 메타 데이터 서버 클러스터의 구성

련 연산의 속도 향상을 위하여 동일한 디렉터리 계층 구조를 각 메타 데이터 서버에 별도로 저장하고 관리한다.

3.1.1 해시 함수(Hash Function)

제안하는 기법에서도 *LH3*와 마찬가지로 파일의 전체 경로를 입력 값으로 하여 해쉬 함수의 결과 값을 계산하는데, 이 결과 값으로 메타 데이터를 저장할 서버를 바로 지정하는 것이 아니라, 간접 검색 테이블의 엔트리를 결정하여 그 엔트리에 저장된 메타 데이터 서버가 담당 서버로 결정된다. 공유 저장 장치를 사용하지 않는 비대칭 클러스터 환경에서, 각 메타 데이터 서버는 자기가 담당할 메타 데이터에 저장하기 위하여 로깅(Logging)과 동시 쓰기 연산(Concurrent Update)을 수행한다. 즉, 쓰기 연산이 필요한 경우, 요청 분배 서버로부터 그 내용이 모든 메타 데이터 서버들에게 전달되어 로깅이 수행되고, 추후 모든 메타 데이터 서버에서 동시에 쓰기 연산이 수행되어 서버 간 메타 데이터의 이동에 따른 과부하를 방지한다.

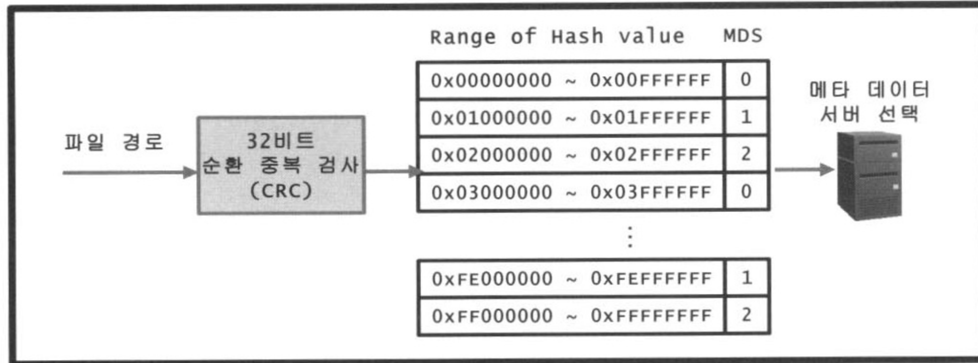
모든 메타 데이터 서버와 요청 분배 서버는 동일한 해시 함수와 간접 검색 테이블을 사용하여 자체적으로 자신의 해당 메타 데이터의 담당 여부를 결정하게 되는데, 본 논문에서는 파일의 전체 경로를 입력 값으로 하는 테이블 방식의 32비트 순환 중복 검사(Table-driven Cyclic Redundancy Check) 알고리즘을 해시 함수로 사용하였다. 32비트 순환 중복 검사 알고리즘은 입력 문자열에 수학적 연산을 수행하여 32비트의 유일한 숫자 값을 생성한다. 이 알고리즘은 원래 통신상에서 전달되는 메시지의 원본 데이터의 변경 여부를 검사하기 위하여 고안된 것으로, 그 계산이 매우 단순하고, 입력 문자열에서 한 비트만 변경되더라도 결과 값이 차이가 나므로 비슷한 이름을 갖는 파일이나 디렉터리를 갖는 상황에서도 효율적이다.

3.1.2 간접 검색 테이블(Indirect Lookup Table)

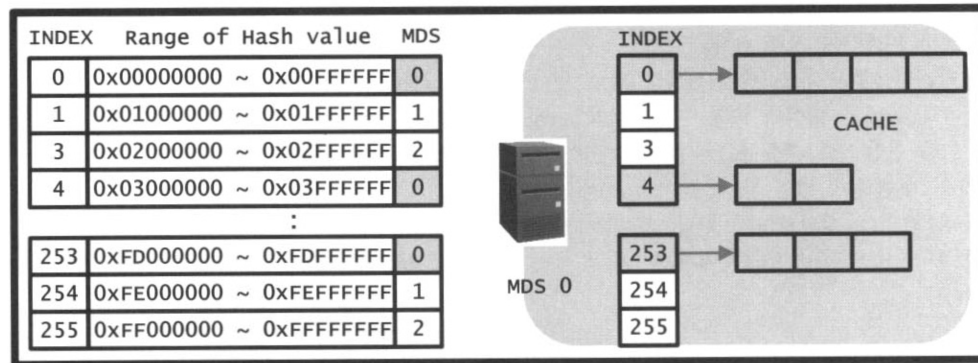
각 메타 데이터의 32 비트 순환 중복 검사 알고리듬을 사용한 결과 값은 0x00000000~0xFFFFFFFF 사이의 값을 갖게 되는데, 각 메타 데이터 서버는 이 중 특정 범위를 할당받게 되고 관련 정보는 간접 검색 테이블로 관리된다. 따라서 클라이언트가 특정 메타 데이터를 요청하면 요청 분배 서버가 이 간접 검색 테이블을 검색하여 담당하는 메타 데이터 서버를 찾아낸다.

(그림 2)는 메타 데이터의 영역을 256개로 분할하고 메타 데이터 서버를 할당한 간접 검색 테이블을 나타낸 것이다. 처음에는 간접 검색 테이블의 각 엔트리에 담당할 메타 데이터 서버를 균등하게 차례로(Round Robin) 할당하여 각 메타 데이터 서버가 하나 이상의 영역을 담당하도록 한다.

서비스가 진행됨에 따라 메타 데이터 서버에 장애가 발생하거나 특정 서버에 부하가 집중되는 상황이 발생하게 되면, 해시 함수의 변경이나 저장된 메타 데이터의 이동 없이, 테이블 엔트리에 담당 메타 데이터 서버를 수정함으로써 서버



(그림 2) 해시 함수와 간접 검색 테이블을 이용한 메타 데이터 서버 선택



(그림 3) 메타 데이터 서버의 캐시 관리

들 간의 부하를 동적으로 조정할 수 있다. 메타 데이터들 간의 동적인 부하 분산을 위해서 각 서버는 자신의 부하 정보를 요청 분배 서버에게 주기적으로 보내고, 요청 분배 서버는 각 서버들의 부하 정보를 통합하여 간접 검색 테이블의 엔트리를 조정한 후 다시 각 서버에게 분배하여, 전체 클러스터 내에서 다시 동일한 간접 검색 테이블을 가지도록 한다. 자세한 메타 데이터 서버들 간의 부하 조정 방법은 3.3 절에서 자세히 다루도록 하겠다.

3.1.3 캐시(Cache)

디스크에 메타 데이터를 읽어 오거나 저장하는 것은 저장 장치의 낮은 성능으로 인하여 메타 데이터 서버의 성능을 저해하는 요인이 된다. 따라서 각 메타 데이터 서버는 일반 파일 시스템에서와 마찬가지로 접근했던 메타 데이터에 대한 캐시를 메모리에 저장하여 저장 장치로의 접근 횟수를 최소화하고 응답 시간을 향상시킨다. 메타 데이터 서버 사이에 공유 캐시(Cooperative Cache)를 지원하기 위해서, (그림 3)과 같이 각 메타 데이터 서버는 자기가 담당하고 있는 간접 검색 테이블의 엔트리에 해당하는 메타 데이터에 대한 캐시만을 저장한다. 메타 데이터 서버 사이에 담당하는 메타 데이터가 변경됨에 따라 해당 캐시도 새로 담당하는 서버로 이동시키는 공유 캐시는 3.3 절에서 자세히 기술하겠다.

3.1.4 디렉터리 계층 구조(Directory Hierarchy Structure)

디렉터리의 검색, 이동, 그리고 삭제와 같은 디렉터리 연산은 계산 량이 많고 응답 시간이 긴 메타 데이터 서비스에 해당한다. 디렉터리 검색과 삭제 연산은 해당 디렉터리 뿐 아니라 모든 하위 파일들과 디렉터리들의 메타 데이터에 대한 추가적인 접근을 필요로 한다. 또한 디렉터리 이동의 경우는 목적지 디렉터리에 대한 타당성 검사 수행과 해당 디렉터리와 그 하위의 모든 파일과 디렉터리에 대한 메타 데이터의 이동을 필요로 한다.

담당하는 메타 데이터의 영역이 분할되어 있는 비대칭 메타 데이터 서버 클러스터에서 이런 연산은 다른 메타 데이터 서버와의 추가적인 통신을 발생시키므로 여러 서버의 부하를 증가시키고 클라이언트 요청에 대한 응답 시간을 지연시킨다. 따라서 이런 디렉터리 연산이 수행될 때 메타 데이터의 이동 시간을 최소화 하고 다른 메타 데이터 서버와의 추가적인 통신을 방지하기 위하여, 모든 메타 데이터 서버는 로컬에 동일한 디렉터리 계층 구조를 저장한다.

3.2 내용 기반의 요청 분배 과정

클라이언트의 메타 데이터 서비스 요청은 그 내용에 해당하는 메타 데이터 연산의 종류에 따라 요청 분배 서버에 의해 적절히 서버들에게 분배된다. 리눅스 가상 파일 시스템(VFS)에 따르면 inode 연산 리스트 중에서 메타 데이터와 관련 있는 연산은 디렉터리 정보 검색, 파일 정보 검색, 디렉터리 생성, 디렉터리 삭제, 디렉터리 이름 변경, 디렉터리 정보 수정, 파일 생성, 파일 삭제, 파일 이름 변경, 파일 정보 수정, 그리고 파일 경로 검색 등이다[9].

앞 절에서 살펴보았듯이 간접 검색 테이블이 변경되더라도 모든 메타 데이터 서버와 요청 분배 서버가 동일한 메타 데이터를 갖도록 하기 위해서는 메타 데이터의 수정 연산에 대해서는 모든 메타 데이터 서버가 그 내용을 로깅하고 수정을 동시에 수행해야 한다. 메타 데이터의 수정 연산에는 디렉터리나 파일을 생성하거나 삭제하고, 또는 그 이름이나 정보를 수정하는 등의 연산이 있다. 그러나 메타 데이터의 검색 연산을 수행하는 경우에도 메타 데이터의 "last access time" 필드를 현재 시간으로 수정해야 하므로, 사실상 디렉터리 정보 검색이나 파일 정보 검색의 일부도 수정 연산에 속한다고 볼 수 있다. 따라서 효율적인 메타 데이터 연산의 수행을 위하여 파일 정보 검색과 디렉터리 정보 검색은 일반적인 검색 연산과 "last access time" 필드를 수정하는 수정 연산으로 세분화하여 진행 한다.

본 절에서는 클러스터 파일 시스템의 연산들을 메타 데이터 검색 연산, 디렉터리 메타 데이터 수정 연산, 그리고 파일 메타 데이터 수정 연산으로 분류하고, 각각에 해당하는 파일 시스템 연산을 수행하기 위한 메타 데이터 서버들과 요청 분배기의 동작 과정을 살펴보고자 한다. 또한 마지막 절에서 각 연산의 수행 시간과 부하량을 이론적으로 분석하고 비교하고자 한다.

3.2.1 메타 데이터 검색 연산

파일이나 디렉터리의 내용을 읽기 위해서 클라이언트의 파일 시스템은 디렉터리 정보 검색, 파일 정보 검색, 그리고 파일 경로 탐색 등의 메타 데이터 검색 연산을 메타 데이터 서버에게 요청한다. 디렉터리 정보 검색과 파일 정보 검색의 경우 그 파일이나 디렉터리를 담당하는 메타 데이터 서버에 의해 상수 시간 안에 수행을 완료할 수 있다. 그러나 디렉터리 정보 검색과 파일 정보 검색 연산 중 "last access time" 필드를 수정해야 하는 경우에는 모든 메타 데이터 서버들이 해당 메타 데이터의 "last access time" 필드의 수정 요청을 로깅하는 시간이 추가로 요구된다. 또한 파일 경로 탐색의 경우는 모든 메타 데이터 서버가 디렉터리 계층 구조를 관리하므로, 담당 메타 데이터 서버의 로컬에서 디렉터리 계층 구조의 트리 순회(traverse) 방법으로 수행을 완료할 수 있다.

3.2.2 디렉터리 메타 데이터 수정 연산

메타 데이터 서버는 디렉터리 생성, 디렉터리 제거, 디렉터리 이름 변경, 그리고 디렉터리 정보 수정 등의 클라이언트 요청을 수행하기 위하여 해당 디렉터리의 메타 데이터를 수정한다. 이 연산들 중 디렉터리 정보 수정을 제외한 다른 연산들은 메타 데이터 서버가 저장하고 있는 디렉터리 계층 구조의 변경을 필요로 하므로, 디렉터리 메타 데이터의 수정 뿐 아니라 디렉터리 계층 구조의 검색과 수정도 수행 되어야 한다.

디렉터리 삭제의 경우 대상 디렉터리 뿐 아니라 하위의 모든 파일과 디렉터리의 메타 데이터를 삭제해야 한다. 또한 디렉터리 이름 변경의 경우 대상 디렉터리 뿐 아니라 모든 하위 파일과 하위 디렉터리의 메타 데이터에 대한 해시 함수의 결과 값을 변화시키기 때문에 메타 데이터 서버들 간 메타 데이터의 재배치가 수행되어야 한다. 반면 디렉터리 정보 수정 요청은 디렉터리 계층 구조에 영향을 미치지 않고 단순히 메타 데이터의 수정만을 필요로 한다.

디렉터리 계층 구조의 변경을 요구하는 연산들의 경우 서버 사이에 메타 데이터의 재배치가 필요하므로, 이는 대량의 메타 데이터 이동을 수반하게 되어 그 수행 시간이 너무 길어진다. 그러나 제안하는 시스템에서는 모든 메타 데이터 서버들이 각각 동일한 디렉터리 계층 구조를 관리하므로, 디렉터리 계층 구조를 변경하려는 요청을 받으면 모든 서버에서 각각 해당 연산을 수행하여 새로운 동일한 디렉터리 계층 구조를 생성한다. 이는 담당이 아닌 나머지 메타 데이터 서버들에게 부하를 가중시키지만, 서버 사이의 메타 데이터 이동이 필요 없게 함으로써 클라이언트의 요청에 대한 응답 시간 지연을 방지한다.

3.2.3 파일 메타 데이터 수정 연산

파일 생성, 파일 삭제, 파일 이름 변경, 그리고 파일 정보 수정 등의 요청을 받은 클라이언트의 파일 시스템은 해당 파일의 메타 데이터를 수정하도록 메타 데이터 서버에게 요청하게 된다. 이런 파일의 메타 데이터 수정 연산은 디렉터리 메타 데이터 수정 연산과는 달리 디렉터리 계층 구조의 변경을 필요로 않음으로 비교적 짧은 연산 시간을 갖는다. 그러나 디렉터리 수정 연산에서와 같이 모든 메타 데이터 서버들이 같은 메타 데이터 정보를 갖기 위해서는 모든 메타 데이터 서버들이 동시에 파일의 메타 데이터 수정 관련 연산을 수행해야 한다.

한편 파일 이름 변경 연산의 경우 디렉터리 계층 구조 변경 연산처럼 메타 데이터 서버들 사이에 메타 데이터의 이동을 유발시키지만 해당 파일의 메타 데이터로 그 범위가 국한되므로 부하는 크지 않다. 따라서 디렉터리 계층 구조와 같은 별도의 정보를 관리하지는 않는다.

3.2.4 연산의 수행 시간과 부하량 비교

디렉터리 정보 검색, 파일 정보 검색, 그리고 파일 메타 데이터 수정 연산은 메타 데이터의 정보만 수정하고 연산의 수행을 완료할 수 있다. 그러나 디렉터리 경로 탐색과 디렉터리 메타 데이터 수정 연산은 각 메타 데이터 서버가 저장하고 있는 디렉터리 계층 구조의 탐색과 수정을 추가적으로 요구한다.

디렉터리 메타 데이터 수정 연산, 파일 메타 데이터 수정 연산, 그리고 디렉터리와 파일의 "last access time" 필드 수정 연산은 모든 메타 데이터 서버에서 동시에 수행 된다. 이런 연산들은 각 메타 데이터 서버의 관점에서 보면 그 계산량이 O(Depth of Tree) 또는 O(1)이지만 모든 메타 데이터 서버에서 발생하는 종합적인 부하량으로 보면 서버의 개수만큼 부하가 증가한다. 따라서 메타 데이터 서버의 개수

〈표 1〉 메타 데이터 연산의 수행 시간과 부하량 비교

	수행 시간	부하량
디렉터리 정보 검색	O (1)	O (1) or O (N)
파일 정보 검색	O (1)	O (1) or O (N)
디렉터리 경로 탐색	O (Depth of Tree)	O (Depth of Tree)
디렉터리 생성	O (Depth of Tree)	O (N * Depth of Tree)
디렉터리 삭제	O (Depth of Tree)	O (N * Depth of Tree)
디렉터리 이름 변경	O (Depth of Tree)	O (N * Depth of Tree)
디렉터리 정보 수정	O (1)	O (N)
파일 생성	O (1)	O (N)
파일 삭제	O (1)	O (N)
파일 이름 변경	O (1)	O (N)
파일 정보 수정	O (1)	O (N)

를 N이라 할 때 각 연산 별 필요한 연산 수행 시간과 전체 메타 데이터 서버에서 발생하는 부하량을 정리하면 <표 1>과 같다.

3.3 동적 부하 분산 알고리즘

파일 시스템의 동시 사용자 수가 증가함에 따라 특정 메타 데이터 서버로의 접근에 병목 현상이 일어나게 되고 이는 전체 파일 시스템의 성능을 저하 시키게 된다. 따라서 제안하는 시스템에서는 동시 사용자 수가 증가하더라도 클러스터 파일 시스템의 성능을 일정하게 유지하도록 하기 위해서 해시 함수와 간접 검색 테이블의 동적인 조정을 통하여 메타 데이터 서버들 사이에 부하를 균등하게 분산시키고 공유 캐시를 사용하여 성능을 저하를 최소화 하도록 한다.

3.3.1 간접 검색 테이블 조정 알고리즘

모든 메타 데이터 서버에게 메타 데이터를 균등하게 분할하여 할당하더라도 파일 시스템 연산마다 계산량이 다르고 파일마다 접근 빈도가 다르므로 각 서버의 부하는 불균등하게 발생된다. 따라서 이는 특정 서버에 과도한 부하를 유발하여 메타 데이터 서비스에 대한 응답 시간을 지연시키고 결국 클러스터 파일 시스템 전체의 성능을 저하시킨다. 따라서 서버의 성능과 부하 상태에 따라 간접 검색 테이블을 조정하여 서버들 사이의 부하 불균형을 동적으로 재조정하도록 하는 알고리듬이 필요하다. 즉, 모든 메타 데이터 서버들이 평균 부하량에 가까운 부하량을 가지도록 간접 검색 테이블을 조정하되 발생되는 과부하를 최소화하기 위하여 엔트리를 담당하는 서버의 변경을 최소화해야 한다.

먼저 전체 메타 데이터 서버 클러스터의 평균 부하량을 계산하고 평균을 초과한 부하를 갖는 서버와 그 초과한 부하량을 결정해야 한다. 한 메타 데이터 서버 mds_i에 대하여 현재 부하량을 $Load(mds_i)$로 나타낼 때, 메타 데이터 서버들의 평균 부하량 $Average(MDS)$는 <식 1>과 같다.

$$Average(MDS) = \frac{\sum Load(mds_i)}{l}, \text{ l is the number of servers.} \quad \text{<식 1>}$$

또한 서버의 초과 부하량은 <식2>와 같이 현재 부하량에서 평균 부하량을 뺀 값이 된다.

$$Extra(mds_i) = Load(mds_i) - Average(MDS) \quad \text{<식 2>}$$

이 때 초과 부하량 $Extra(mds_i)$가 양수인 서버 mds_i는 담당하는 간접 검색 테이블의 엔트리 개수를 줄여서 부하를 줄여야 하고, 음수인 서버는 테이블의 엔트리를 추가로 할당받아 부하를 추가로 부담할 수 있다. 따라서 평균 부하량을 초과하는 부하를 가진 서버는 초과한 부하량에 해당하는 테이블의 엔트리 개수를 계산하여 다른 서버에게 배분해야 한다. <식 3>와 같이 한 서버 mds_i의 현재 부하량을 담당하고 있는 엔트리의 개수로 나눈 값을 엔트리 당 부하량 $Load_e(mds_i)$이라고 할 때, $Extra(mds_i)$가 양수인 mds_i, 즉 평균 부하량보다 많은 부하를 가진 서버가 줄일 수 있는 최대 엔트리 수 E_i는 <식 4>를 만족시키는 시키는 음이 아닌 정수 중에서 최대값이 된다.

$$Load_e(mds_i) = \frac{Load(mds_i)}{m},$$
$$m = \text{서버 mdsi의 담당 엔트리 수} \quad \text{<식 3>}$$

$$Extra(mds_i) - Load_e(mds_i) \times E_i \le 0,\ 0 \le E_i \le m \quad \text{<식 4>}$$

메타 데이터 서버의 개수가 4개, 각 서버별 담당 엔트리 개수가 64개, 부하 허용 한계치가 70%, 그리고 평균 부하량이 55%일때 $Load_e(mds_i)$, $Extra(mds_i)$, E_i는 <표 2>와 같다.

모든 메타 데이터 서버가 동일한 성능을 가지지 않을 경우 엔트리 당 부하량이 각 서버마다 다를 것이므로 간접 검색 테이블의 담당 엔트리 조정 시 이를 고려하여야 한다. 예를 들어 간접 검색 테이블의 k번째 엔트리의 담당 서버를 mds_i에서 mds_j로 변경시키는 경우, mds_i와 mds_j의 성능을 각각 $Power(mds_i)$와 $Power(mds_j)$라 하면, mds_i가 담당하고 있는 k번째 엔트리에 대한 부하량은 $Load_e(mds_i)$이므로, mds_j가 새로 담당하게 될 엔트리 k에 대한 부하량은 <식 5>과 같다.

$$Load_e(mds_j) = \frac{Power(mds_j)}{Power(mds_i)} \times Load_e(mds_i) \quad \text{<식 5>}$$

평균 부하량 보다 적은 부하를 가진 서버 mds_j는 평균 부하량 이상의 부하를 가진 mds_i의 담당 엔트리를 대신 할당받아야 하는데, 이 때 할당받게 될 엔트리의 개수 I_j는 E_i를 넘지 않는 범위 안에서 mds_j의 초과 부하량이 음수를 유

〈표 2〉 메타 데이터 서버별 부하량 정보의 예

$MDS(i)$	$Load(mds_i)$	$Extra(mds_i)$	$Entries$ #	$Load_e(mds_i)$	E_i
0	50 %	5 %	64	0	0
1	50 %	5 %	64	0	0
2	70 %	-15 %	64	1.09	13
3	50 %	5 %	64	0	0

〈표 3〉 부하 분산 알고리즘 적용 결과의 예

MDS (i)	Extra (mdsi)	Entries #	Loade (mdsi)	Ei	New Entries #	Expected Load(mdsi)
0	5 %	64	0	0	69	55.49 %
1	5 %	64	0	0	69	55.49 %
2	-15 %	64	1.09	13	51	53.65 %
3	5 %	64	0	0	67	53.27 %

지할 수 있는 최대 엔트리 수이다. 거기에 mds_j에 하나의 엔트리를 더 추가 했을 때 그 초과 부하량이 mds_j의 엔트리 당 부하량의 1/2 넘지 않는 경우, 추가적으로 한 개의 엔트리를 더 할당 한다. 이를 식으로 나타내면 <식 6>과 같다.

$$Extra(mds_j) + Load_e(mds_j) \times I_j - \frac{1}{2}Load_e(mds_j) \le 0,$$

$$I_j \le E_i \qquad \text{<식 6>}$$

메타 데이터 서버들의 성능이 동일하다고 가정하고, <표 2>의 서버들에 대하여 위의 알고리즘을 적용한 결과는 <표 3>과 같다.

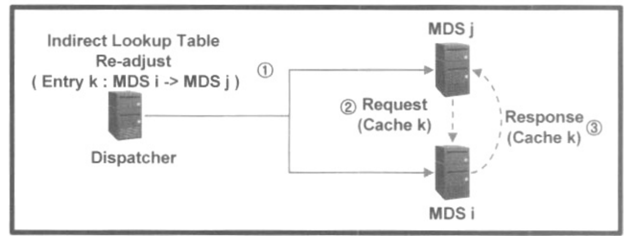
지금까지 논의한 부하 균등을 위한 검색 테이블 조정 알고리즘은 (그림 4)와 같다.

1. 각 메타 데이터 서버의 평균 부하량 *Average(MDS)* 와 초과 부하량 *Extra(mds)* 를 계산한다.
2. 평균 부하량을 초과하는 부하를 가진 서버들을 *Y*, 평균 부하량 이하의 부하를 가진 서버들을 *Z* 로 분류한다.
$$Y = \{mds_i | Extra(mds_i) > 0, \text{for all } i\}$$
$$Z = \{mds_i | Extra(mds_i) \le 0, \text{for all } i\}$$
3. *Y* 의 모든 서버에 대하여 엔트리당 부하량 $Load_e(mds_i)$ 와 줄일 수 있는 최대 엔트리 수 E_i 를 계산한다.
4. 엔트리 당 부하량이 가장 큰 서버 mds_i 를 선택한다.
5. *Z* 에서 하나의 서버 mds_j 를 선택한다.
6. 서버 mds_j 의 엔트리 당 부하량 $Load_e(mds_j)$ 를 계산하여 추가할 수 있는 최대 엔트리 수 I_j 를 결정한다.
7. 간접 검색 테이블에서 mds_i 가 담당하는 엔트리 중 I_j 개를 mds_j 가 담당하도록 조정한다.
8. 서버 mds_j 의 초과 부하량 $Extra(mds_j)$ 와 E_i 를 다음과 같이 재계산 한다.
$$Extra(mds_j) \leftarrow Extra(mds_j) + Load_e(mds_j) \times I_j$$
$$E_i \leftarrow E_i - I_j$$
9. E_i 가 *0* 이 되면 끝내고, 아니면 5 ~ 8의 과정을 *Z* 의 각 서버에 대하여 수행한다.
10. *Y* 에 속한 서버가 없으면 끝내고, 아니면 4 ~ 9의 과정을 *Y* 의 각 서버에 대하여 수행한다.

(그림 4) 간접 검색 테이블 조정 알고리즘

3.3.2 메타 데이터 서버들의 공유 캐시 알고리즘

간접 검색 테이블의 재조정은 각 메타 데이터 서버가 담당하고 있는 메타 데이터 영역의 변화를 일으킨다. 메타 데이터 서버들은 성능 향상을 위하여 담당하는 메타 데이터에 대한 캐시를 유지하므로, 테이블 조정이 일어나면 이전에 담당했던 영역에 대한 캐시는 삭제하고 새로 담당하게 되는 영역의 메타 데이터를 로컬 디스크로부터 읽어 와야 한다. 이와 같은 테이블 조정에 따르는 과부하를 줄이기 위하여,



(그림 5) 간접검색테이블 조정에 따른 메타데이터 서버 간의 선택적 공유캐시

간접 검색 테이블 엔트리의 담당 서버만 변경하는 것이 아니라 그에 해당하는 캐시를 서버 사이에 교환하는 공유 캐시를 사용한다. 이는 서버 사이의 네트워크가 저장 장치 보다 넓은 대역폭을 가진 환경에서 효율적이다.

각 메타 데이터 서버는 (그림 3)과 같이 캐시를 간접 검색 테이블의 엔트리별로 나누어 저장한다. 간접 검색 테이블의 k 번째 엔트리의 담당 서버가 mds_i에서 mds_j로 변경되는 경우 공유 캐시를 위한 알고리즘은 (그림 5)와 같다. 즉, mds_j가 mds_i에게 간접 검색 테이블의 k 번째 엔트리에 대한 캐시를 요청하면, 요청을 받은 mds_i는 보유하고 있던 k 번째 엔트리에 대한 캐시를 네트워크를 통해 mds_j에게 보내게 된다. 이런 과정을 통하여 mds_j는 기존의 캐시 정보를 재활용하여 새로 담당하게 된 엔트리에 대한 메타 데이터를 저장 장치에서 다시 읽어오지 않고도 빠르게 캐시를 구축하여 부하 조정에 따르는 과부하를 최소화한다.

4. 성능 평가

본 장에서는 제안한 클러스터 파일 시스템의 메타 데이터 서버의 동적 부하 분산 알고리즘 성능을 측정하기 위하여 프로세서 기반 이산 사건 시뮬레이터인 Mesquite Software 사의 CSIM19 시뮬레이터를 사용하였다[10].

비대칭 클러스터 파일 시스템에서 클라이언트 요청이 특정 메타 데이터 서버로 집중되어 과부하가 발생하게 되면 클러스터 파일 시스템 전체의 성능이 저하되어 클라이언트 요청에 대한 응답 시간이 길어지게 된다. 따라서 제안하는 알고리즘이 기존의 대표적인 클러스터 파일 시스템 관리 방법인 *Vesta*와 *LH3*에 비해 서버 사이에 부하를 공평하게 분배하고 파일 시스템의 성능을 일정하게 유지하고 있음을 보이기 위하여 시뮬레이션을 통해 성능을 측정하고 그 결과를 비교하였다.

시뮬레이션은 짧은 시간에 클라이언트 요청을 집중적으로 발생시켜 진행하였고, 성능 판단을 위하여 두 가지 평가 기준을 사용하였는데, 첫 번째는 클라이언트 요청의 집중 발생 시간 동안 각 서버의 최대 부하량이고 두 번째는 클라이언트의 각 요청에 대한의 응답 시간이다.

클러스터 파일 시스템의 경우 일반적으로 읽기와 쓰기 연산의 비율이 약 9:1이다[11]. 따라서 먼저 이 비율의 연산을 수행하여 성능을 측정하고, 추가로 쓰기 연산의 비율을 증가시키면서 그 성능의 변화를 측정하고 분석하였다. 마지막

으로 부하 분배 서버의 성능에 따른 전체 클러스터 파일 시스템의 성능 변화를 측정하고 분석하였다.

4.1 시뮬레이션 모델

시뮬레이션을 위한 메타 데이터 서버 모델은 하드웨어 사양이 같은 서버들로 가정하였다. 서버는 인텔 펜티엄 III - 800Mhz CPU, PC100 타입의 메모리, 그리고 Ultra ATA - 66과 5600 RPM을 지원하는 하드 디스크로 구성 되었으며, 리눅스 커널 2.6을 운영체제로 사용하는 서버로 모델링하였다. 연속적인 20 KB의 데이터를 접근할 때 메모리와 하드 디스크의 이론적인 최대 전송 속도는 각각 0.0244 ms와 12.5035 ms이다. 메모리와 하드 디스크의 구체적인 하드웨어 사양은 <표 4>와 같다.

클러스터 파일 시스템에서 각 메타 데이터의 크기는 디스크의 섹터 당 바이트 수를 고려하여 2의 배수로 결정되어야 한다. 현재 리눅스 Ext2 파일 시스템은 inode의 크기가 128 바이트지만 클러스터 파일 시스템은 그 구현 방식에 따라 해시 값, 파일 데이터의 분산 저장 정보 등을 추가로 포함하여야만 한다. 따라서 본 성능 평가에서는 메타 데이터의 크기를 256 바이트로 가정하고 수행하였다.

메타 데이터의 수정 관련 연산은 서버 간의 가용성과 일관성을 위하여 직접 디스크에 쓴다고 가정하고, 읽기 연산은 성능을 고려하여 메모리 캐시를 사용한다고 가정한다. 리눅스 커널의 추가적인 연산 시간을 포함한 실질적인 메모리와 디스크의 평균 접근 시간은 직접 서버에서 측정하였다. 메타 데이터 크기를 256 바이트라고 했을 때, 10 MB의 메모리 캐시 평균 검색 시간은 0.155 ms이며, 리눅스 커널 캐시를 고려하지 않은 상황에서 메타 데이터 1개에 대한 디스크 평균 접근 시간은 1.561 ms였다. 미국의 SUN사는 기술 문서를 통해 NFS의 성능 유지를 위하여 inode의 캐시 적중 비율(hit ratio)을 90% 이상으로 유지할 것을 추천하고 있다. 따라서 본 논문에서도 대용량 메모리를 이용한다고 가정하고 캐시 적중 비율을 90%로 유지하는 것으로 하였다.

서버 사이의 통신은 대부분 3 바이트 미만의 컨트롤 메시지와 메타 데이터의 이동이 차지하기 때문에 컨트롤 메시지의 통신 시간은 무시하였다. 메타 데이터의 이동만을 고려하여 한 번 통신에 필요한 메시지 크기를 256 바이트로 가정하였다. 100 Mbps 대역폭을 갖는 네트워크에서 하나의 메시지 전달에 필요한 이론적인 시간은 0.02048 ms이다. 그러나 실제 측정된 시간은 TCP 프로토콜의 슬로우 스타트(slow start)와 프로토콜 추가 비용(overhead) 등으로 인하여 10배 정도인 0.209 ms로 였다. 시뮬레이션에 사용한 파라미터는 <표 5>에 정리하였다.

〈표 4〉 하드웨어 사양

PC100 타입 메모리	Access Time	10 ns
	대역폭	64 bit
5600 RPM 하드 디스크	외부 최대 전송 속도	66 MB/sec
	내부 최대 전송 속도	445 Mbit/sec
	평균 탐색 시간	12.5 msec
	섹터 당 바이트	512 byte

〈표 5〉 시뮬레이션 파라미터

메타 데이터 서버의 수	8
메타 데이터의 크기	256 Byte
평균 메모리 검색 시간	10 MB 당 0.155 ms
메모리 캐시 적중율	90 %
디스크 접속 시간	메타 데이터 당 1.561 ms
네트워크 전송 시간	메타 데이터 당 0.209 ms

4.2 실험 결과 분석

앞서 언급한 시뮬레이션 모델 및 파라미터 값에 따라 *Vesta*, *LH3*, 그리고 본 논문이 제안한 알고리즘, 세 가지 기법에 대한 성능을 평가하고 결과를 분석하였다. 시뮬레이션은 20000 ms 동안 클라이언트 요청을 집중적으로 발생시키면서 성능 판단을 위하여 두 종류의 측정을 실시하였다. 첫 번째는 클라이언트 요청의 집중 발생 시간 동안 서버의 최대 부하량을 측정하였고, 두 번째는 각 클라이언트 요청에 대한 응답 시간을 측정하였다.

클라이언트 요청에 대하여 읽기와 수정 연산의 비율을 9:1로 하여 성능을 측정하였다. 또한 쓰기 연산 비율이 제안 기법의 성능에 미치는 영향을 알아보기 위하여 쓰기 연산 비율 증가에 따른 성능 변화를 측정하였고 마지막으로 부하 분배기의 성능이 제안 기법의 성능에 미치는 영향에 대하여 실험하였다.

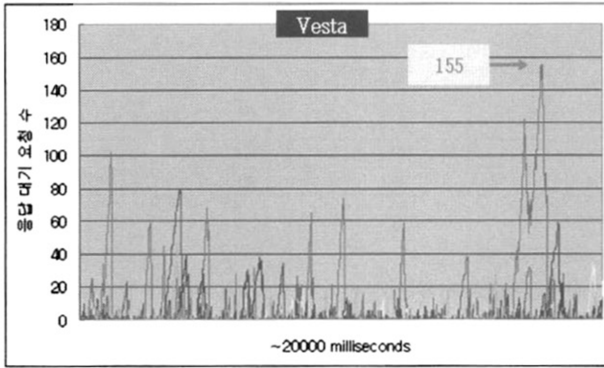
4.2.1 파일 시스템 성능 비교

(그림 6), (그림 7), (그림 8)에서 읽기와 쓰기 연산의 비율이 9:1일 때, 각 기법의 성능을 나타내는 그래프를 나타내었다. 그래프는 클라이언트 요청의 집중 발생 기간 동안 시간의 흐름에 따라 각 서버에서 처리를 기다리는 대기 요청 수의 변화를 나타낸다. *Vesta*와 *LH3*의 경우 한 서버의 요청 대기 수가 나머지 서버에 비하여 많고 그 기간이 어느 정도 지속되고 있음을 볼 수 있는데, 이것은 특정 서버에게 부하가 집중되고 있음을 의미한다. 그러나 제안 기법에서는 특정 서버로 대기 요청 수가 집중되는 현상이 거의 없거나 있더라도 그 기간이 상당히 짧음을 알 수 있다. 이는 제안 기법이 다른 기법에 비하여 부하를 효과적으로 분산하고 있음을 나타낸다.

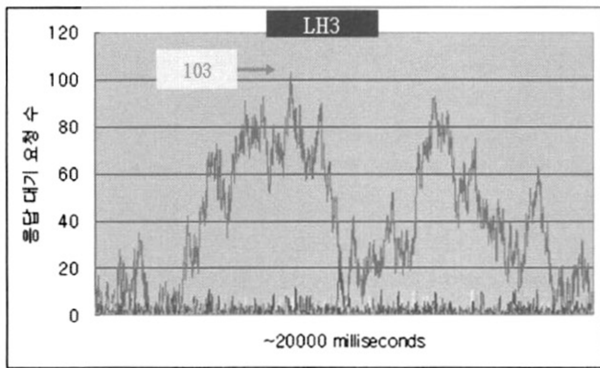
각 서버의 평균 대기 요청 수와 그 분산을 <표 6>에서 정리하였다. 제안 기법의 경우 평균 대기 요청 수가 3.13으로 *Vesta*의 2.11보다는 조금 높은 것으로 나타난다. 그러나 서버 간 분산은 제안 기법은 0.14로 *Vesta*의 10.36보다 분포가 훨씬 균일함을 알 수 있다. 즉 제안 기법은 평균 요청 개수를 상향 평준화하지만, 특정 서버로 요청이 몰리지 않도록 균등하게 요청과 부하를 분배함을 알 수 있다.

〈표 6〉 대기 요청 수의 평균 및 분산

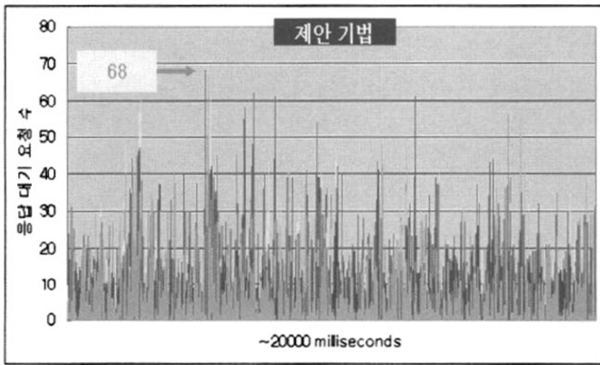
	Vesta	LH3	제안 기법
평균 대기 요청수	2.11	4.35	3.13
분산	10.56	95.10	0.14



(그림 6) 응답 대기 요청 수 - Vesta



(그림 7) 응답 대기 요청 수 - LH3



(그림 8) 응답 대기 요청 수 - 제안 기법

4.2.2 요청 분배 서버 성능의 영향

큐잉 이론의 시스템 모델 중 요청 분배 서버를 M/M/1 모델로 가정하고, C는 요청 분배 서버의 용량, 즉 서비스 비율을 나타내고, λ는 클라이언트 요청의 발생 비율을 나타내면, 요청 분배 서버의 성능이라 할 수 있는 평균 응답속도 R은 다음과 같이 표현할 수 있다[12].

$$R = \frac{1}{C - \lambda}$$

이때 클라이언트 요청 발생 비율 λ가 일정하다고 하면, 이 요청 분배 서버의 평균 응답 시간은 요청 분배 서버의 용량 C에 의하여 결정되는데, C가 λ에 비해 월등히 큰 경우, C가 λ와 비슷하지만 큰 경우, 그리고 C가 λ보다 작거나 같은 경우로 구분하여 그 상황을 고려해야 한다.

먼저, C가 λ에 비하여 훨씬 큰 경우, 즉 C >> λ인 경우, R ≈ 1 / C가 되어 C의 값이 늘어날수록 그 평균 응답 시간은 기하급수적으로 빨라진다. 따라서 요청 분배 서버가 일정 수준 이상의 용량을 갖는다고 가정하면 충분히 빠른 평균 응답 시간을 갖는다고 볼 수 있다.

두 번째 C가 λ보다 크기는 하지만 비슷한 경우, 즉 클라이언트의 요청 발생 비율이 거의 요청 분배 서버의 서비스 비율에 가까운 경우, C가 λ에 가까워질수록 평균 응답 시간 R은 무한히 커지게 된다.

$$\lim_{C \to \lambda} \frac{1}{C - \lambda} = \infty$$

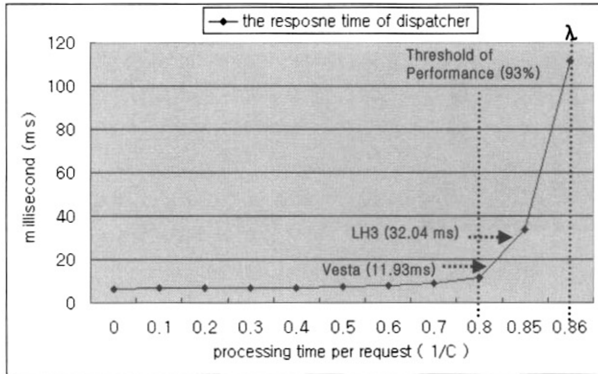
마지막으로 λ가 C 보다 크거나 같은 경우 요청 분배 서버가 처리할 수 있는 용량과 같거나 더 빠른 속도로 클라이언트의 요청이 도착하므로 시스템 내에서 요청이 대기하는 시간이 무한대가 되고 따라서 평균 응답시간은 무한대가 된다.

앞의 실험과 같은 읽기 90%, 수정 10%로 이루어진 클라이언트의 요청이 있는 환경에서, 요청 분배 서버의 용량이 제안 기법의 성능에 끼치는 영향을 알아보는 실험을 실시하였다. 요청 분배기의 용량을 C라고 할 때 요청 분배기가 클라이언트의 요청을 처리하는 평균 서비스 시간은 1/C이 된다. 클라이언트 요청의 평균 도착 시간 1/λ이 0.86 ms일 때, 요청 분배 서버의 성능에 따른 제안 기법의 응답 시간은 (그림 9)와 같이 변화한다.

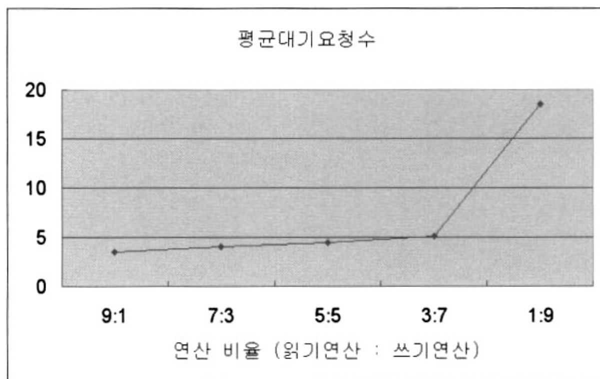
1/C가 0.8 ms 이하일 때는 완만하게 응답 시간이 증가하고, 0.8 ms를 넘어서는 시점에서부터 응답 시간이 급격히 증가하고 있다. 또한 1/C이 0.8 ms 이하일 때 제안 기법의 성능이 기존의 *Vesta*나 *LH3*의 성능보다 우수하다는 것을 알 수 있다. 다르게 표현하면, 클라이언트 요청의 발생 비율 λ에 비해 요청 분배기의 처리 시간 1/C이 93% 수준일 때까지는 제안 기법의 응답 시간이 완만하게 증가하고, 기존의 다른 기법과 비교하여도 우수한 응답 시간을 가진다고 할 수 있다.

4.2.3 읽기와 수정 비율에 따른 성능 변화

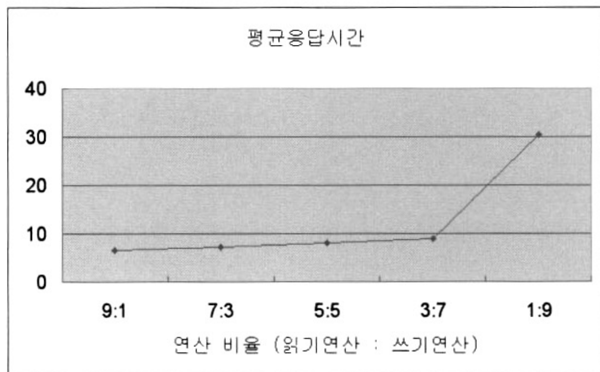
(그림 10)과 (그림 11)은 수정 관련 연산의 비율이 변할 때 제안 기법의 성능 변화를 측정하여 나타낸 그래프이다. 전체 클러스터 파일 시스템 연산 중 수정 관련 연산의 비율을 10%, 30%, 50%, 70%, 90%로 변화시키면서 평균 대기 요청수와 응답 시간을 측정하였으며, 이 때 요청 분배서버의 처리 시간 1/C은 0으로 가정 하였다. (그림 10)은 모든 서버에서 수행을 기다리는 클라이언트 요청의 평균 개수를 나타낸 그래프이고, (그림 11)은 클라이언트 요청에 대한 평균 응답 시간을 측정하여 나타낸 그래프이다. 수정 연산의 비율이 증가함에 따라 디렉터리 계층 구조 수정에 대한 부담 등으로 제안 기법의 성능이 저하되고 있으며, 수정 연산의 비율이 70%를 넘으면, 10%일 경우와 비교하여 80% 이상 성능이 저하되고 있음을 볼 수 있다.



(그림 9) 서버 성능에 따른 제안기법의 응답 시간 변화



(그림 10) 연산 비율에 따른 평균 대기요청 수



(그림 11) 연산 비율에 따른 평균 응답시간

5. 결 론

본 논문에서는 메타 데이터 서비스 성능에 영향을 미치는 파일 시스템 연산의 특징을 분석하여 비대칭 메타 데이터 서버 클러스터에 적합한 내용 기반의 동적 부하 분산 알고리즘을 제안하였다. 클라이언트 요청의 분산 과정에서 발생하는 메타 데이터의 이동을 최소화하기 위해서 모든 서버가 동시에 수정 관련 연산을 수행하고 그 사항을 로깅하도록 하였다. 특히, 읽기 연산 중에서도 메타 데이터의 "last access time" 필드의 수정이 필요한 경우 역시 수정 연산과 같은 과정을 거쳐 수행된다.

일반적인 파일 시스템에서의 수정 관련 연산 비율인 10%에서, 기존의 비대칭 메타 데이터 서버 클러스터 관리 기법인 *Vesta*와 *LH3*를 제안하고자 하는 알고리즘과 비교하기 위하여 그 성능을 측정하고 결과를 비교, 분석하였다. 이를 통해, 클라이언트 요청의 집중적으로 발생하는 기간 동안 제안 알고리즘은 다른 알고리즘에 비해 평균 부하 량이 60~89%, 평균 응답 시간은 64~80%의 성능 향상을 보였다. 이는 제안 알고리즘이 부하를 모든 서버에게 공평하게 분배하여 각 서버의 부하량을 비슷한 수준으로 유지함으로써 성능을 향상시키고 있다는 것을 보여준다.

또한, 수정 연산의 비율을 증가시키면서 제안 기법의 성능이 얼마나 저하되는지를 실험하였는데, 수정 연산의 비율이 90%인 경우, 10%인 경우에 비해 80% 이상의 성능 저하를 보임을 알 수 있었다.

제안 기법의 성능을 결정하는 주요 요소 중 하나는 요청 분배 서버의 성능이다. 요청 분배 서버의 용량이 클라이언트의 요청의 도착 비율 보다 빠른 경우, 제안 기법의 성능 향상을 기대할 수 있다. 실험을 통해, 클라이언트 요청의 평균 도착 시간이 0.86 ms일 때, 요청 분배 서버의 평균 처리 시간이 평균 요청 도착 시간의 93% 수준일 때까지는 제안 기법의 응답 시간이 완만하게 증가하며 기존의 다른 기법들과 비교하여도 우수한 응답 시간을 가짐을 알 수 있었다.

참 고 문 헌

[1] SPEC, "SFS 3.0 Documentation Version 1.0," Standard Performance Evaluation Corporation, 2001.

[2] K. W. Preslan et al., "A 64 Bit, Shared Disk File System for Linux," Proceedings of the 16th IEEE Mass Storage Systems Symposium, pp.22-41, 1999.

[3] http://www.macroimpact.com

[4] P. H. Carns et al, "PVFS: A Parallel File System For Linux Clusters," Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, pp.317-327, Oct., 2000.

[5] Peter J. Braam et al., "The Lustre Storage Architecture," Cluster File System. Inc, Mar., 2003.

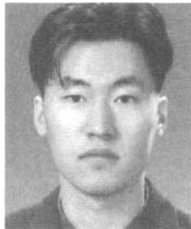
[6] Scott A. Brandt et al., "Efficient Metadata Management in Large Distributed Storage Systems," Proceedings of the 11th IEEE NASA Goddard Conference on Mass Storage Systems and Technologies, Apr., 2003.

[7] Jin Xiong et al., "Design and Perforamance of the the Dawning Cluster File System," IEEE International Conference on Cluster Computing(Cluster'03), Dec., 2003.

[8] Peter F. Corbett et al., "The Vesta parallel file system," ACM Transactions on Computer Systems(TOCS), Vol.14, No.3, pp.225-264, Aug., 1996.

[9] Daniel P. Bovet et al, Understanding the Linux Kernel, O'Reilly and Associates, Sebastopol, 2003.

[10] http://www.mesquite.com
[11] L. mummert and M. Satyanarayanan. "Long term distributed file reference tracing: Implementation and experience". Software-Practice and Experience (SPE), Vol.26, No.6, pp.705-736, June, 1996.
[12] Kishor Shridharbhai Trivedi, Probability and Statistics with Reliability, Queuing, and Computer Science Applications, John Wiley & Sons, Inc., New York, 2002.



장 준 호
e-mail : junho.jang@gmail.com
1998년 서강대학교 컴퓨터학과(공학사)
2004년 서강대학교 컴퓨터학과(공학석사)
2004년~현재 삼성전자 메모리 사업부
관심분야 : 클러스터 파일시스템, 분산처리 시스템

한 세 영
e-mail : syhan@sogang.ac.kr
1991년 포항공과대학교 수학과(이학사)
2003년 서강대학교 정보통신대학원(공학석사)
2004년~현재 서강대학교 컴퓨터학과 박사과정 재학중
관심분야 : 피어투피어 컴퓨팅, 분산처리 시스템

박 성 용
e-mail : parksy@sogang.ac.kr
1987년 서강대학교 컴퓨터학과(공학사)
1994년 미국 Syracuse University 대학원(공학석사)
1998년 미국 Syracuse University (공학박사)
1998년~1999년 미국 Bell Communication Research 연구원
1999년~현재 서강대학교 컴퓨터학과 부교수
관심분야 : Autonomic Computing, Peer to Peer Computing, High Performance Cluster Computing and System