다단계 중복 제거 기법을 이용한 클러스터 기반 파일 백업 서버

(A Clustering File Backup Server Using Multi-level De-duplication)

고 영 웅 † 정 호 민 †† 김 진 †††
(Young Woong Ko) (Ho Min Jung) (Jin Kim)

요 약 기존의 상용 저장 시스템은 데이타를 저장할 때 몇 가지 문제점을 가지고 있다. 먼저, 데이타를 저장함에 있어서 실용적인 중복제거 기법이 널리 활용되고 있지 못하기 때문에 저장 장치 낭비를 초래하고 있다. 또한 대규모 데이타 입출력을 처리하기 위해서 고사양의 시스템을 요구한다는 부분도 문제점으로 지적할 수 있다. 이와 같은 문제를 해결하기 위해서 본 논문에서는 블록 수준에서의 중복을 제거하기 위한 방안으로 파일 지문을 이용한 클러스터링 기반 저장 시스템을 제안하고 있다.

본 연구는 기존의 저장 시스템과 몇 가지 부분에서 차이를 보인다. 먼저, 파일 블록의 지문을 이용한 다단계 중복 제거 기법을 통하여 불필요한 데이타에 대한 저장 용량을 효과적으로 줄일 수 있었다. 또한 입출력 시스템 부분에서는 클러스터링 기법을 적용함으로써 데이타 전송 및 입출력 시간을 효과적으로 감소시켰다. 본 논문에서는 제안된 방법을 검증하기 위해서 몇 가지 실험을 수행하였으며, 실험 결과 저장 공간과 입출력 성능이 크게 개선되었음을 보였다.

키워드 : 파일 지문, 클러스터, 백업, 메타 데이타 서버, 해쉬, SHA1

Abstract Traditional off-the-shelf file server has several potential drawbacks to store data blocks. A first drawback is a lack of practical de-duplication consideration for storing data blocks, which leads to worse storage capacity waste. Second drawback is the requirement for high performance computer system for processing large data blocks. To address these problems, this paper proposes a clustering backup system that exploits file fingerprinting mechanism for block-level de-duplication.

Our approach differs from the traditional file server systems in two ways. First, we avoid the data redundancy by multi-level file fingerprints technology which enables us to use storage capacity efficiently. Second, we applied a cluster technology to I/O subsystem, which effectively reduces data I/O time and network bandwidth usage. Experimental results show that the requirement for storage capacity and the I/O performance is noticeably improved.

Key words : File Fingerprinting, Cluster, Backup, Meta Data Server, Hash, SHA1

· 본 연구는 산업자원부와 한국산업기술재단의 지역혁신인력양성사업과 2008년도 한림대학교 교비 학술연구비(HRF-2008-032)에 의하여 수행된 연구결과임

† 종신회원 : 한림대학교 컴퓨터공학과 교수
yuko@hallym.ac.kr
†† 학생회원 : 한림대학교 컴퓨터공학과
chorogyi@hallym.ac.kr
††† 정 회 원 : 한림대학교 컴퓨터공학과 교수
jinkim@hallym.ac.kr

논문접수 : 2008년 3월 19일
심사완료 : 2008년 7월 23일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제14권 제7호(2008.10)

1. 서 론

2006년 3월 세계적인 시장 조사 기관 IDC(International Data Corporation)에서 발표한 보고서에 따르면 2007년 전 세계에서 생산, 유통될 디지털 정보의 양은 162 Exa Byte일 것이라 한다. 데이타 증가 속도로 보면 연평균 51%씩 꾸준히 성장함을 알 수 있다. 디지털 데이타의 증가 요인으로 기업과 정부의 전자 문서 관리 시스템 수요 증가, WEB 2.0의 발전으로 고용량 이미지, 동영상 데이타로 이루어진 사용자 제작 콘텐츠 등으로 볼 수 있으며 이러한 추세로 볼 때 인터넷에서 유통되는 디지털 데이타는 기하급수적으로 증가할 것이다. 또한 장애로 인한 데이타 손실, 소프트웨어 결함, 컴퓨터 고장같이 치명적인 위험에 노출되어 원본 데이타를 복구하지 못한 기업 중 6%만이 생존하는 반면, 43%는 업무를 재개하지 못하고, 51%는 재개가 불가능한 경우가 현실이다[1]. 그러므로 대량의 데이타를 안전하게 백업하고 재난에 대비하는 것은 정보화 사회의 필수 요건이다. 그러나 디지털 데이타의 급격한 증가는 백업 시스템의 저장 용량 부족, 성능 저하, 비용 증대, 서버 성능 저하 등의 여러 가지 문제를 일으키고 있다.

기존 백업 방식에서는 데이타를 백업하기 위해 원본 데이타의 수배, 수십 배에 해당하는 백업 데이타를 Tape 또는 Disk에 저장해야 했다. 기존의 백업 방식인 점진 백업이나 차등백업에서 데이타 중복을 제거하는 방법은 파일 경로, 파일 이름, 파일 수정 시간, 파일 크기 등의 파일 속성을 이용하여 이전의 백업 사본과 비교하는 방식이었으며 이는 주기적인 전체 백업을 요구 했다. 전체 백업은 평균적으로 이전의 백업 사본과 90%이상의 중복을 가지므로 저장 공간의 낭비가 심각하다. 특히 용량이 큰 멀티미디어 데이타는 일부 내용이 수정되었을 경우에, 컨텐츠의 대부분이 중복된 데이타의 집합임에도 불구하고 별개의 파일로 저장되는 문제를 지니고 있다. 또한 최근에는 리눅스 배포(distribution)를 위한 서버 및 가상 머신 이미지(virtual machine image) 등에 있어서도 중복된 데이타의 증가로 인하여 컴퓨터 자원의 낭비가 크다는 연구 결과가 발표되고 있다[2].

본 논문에서는 이러한 문제점을 해결하고자 해시(hash) 함수를 이용하여 파일의 지문(Fingerprint)을 생성하고 생성된 지문들을 비교하여 중복 유무를 판단하고 동일한 블록에 대해서 중복 제거(de-duplication)를 수행하는 방안을 제시하고 있다. 특히 파일에 대한 해시 정보로 동일 파일이 중복 저장되는 것을 방지하고, 파일을 일정 크기의 블록으로 분할한 후에 각 블록에 대해서 해시 정보를 생성하여 블록 수준의 중복 체크를 수행하고 있다. 따라서 중복 파일 및 중복 데이타 블록의 백업을 피할 수 있어 디스크 용량과 네트워크 대역폭을 효율적으로 사용 할 수 있다. 제안된 시스템은 클라이언트, 메타 데이타 서버(MDS: Meta-data Server) 그리고 클러스터 파일 서버로 구성되어 있다. 특히 파일 서버를 클러스터 방식으로 구성하여 확장성 및 성능 향상을 높이고 있는 것이 특징이다.

본 논문의 구성은 다음과 같다. 먼저 2장에서는 백업 기술에 대해서 알아보고, 3장에서는 제안하는 데이타 백업 시스템에 대해 파일 지문을 이용한 백업시스템 설계와 구현에 대해 논한다. 4장에서는 구현 및 성능평가에 대해 기술하였고 5장에서는 관련 연구와 비교를 수행하였다. 마지막으로 6장에서는 결론 및 향후 연구방향을 제시한다.

2. 백업 기술 분석

일반적인 백업 시스템이 갖는 특징을 몇 가지 그룹으로 나누어 생각할 수 있는데, 풀(full) 백업과 증분(incremental)백업, 차등(differential) 백업, 파일 기반의 백업과 물리적 기반의 백업, 데이타 압축 등의 범주로 나눠 볼 수 있다[3]. 풀 백업은 파일시스템 전체를 백업 미디어에 저장하는 방법이다. 백업하고자 하는 파일시스템의 크기에 따라서 소요되는 백업 미디어도 비례해서 증가를 하지만 시스템 장애 발생 시 풀 백업으로 전체 파일시스템을 복구 할 수 있다. 풀 백업은 백업을 할 때 소비되는 백업 미디어의 증가뿐만 아니라 파일시스템 전체를 백업하기 때문에 많은 시간이 소요되고, 저장 공간 또한 많이 차지하게 된다. 이를 보완하고자 제안된 방법이 증분 백업과 차등 백업이라고 볼 수 있다. 증분 백업은 이전 백업 이후에 새로이 생성된 파일이나 변경된 파일만을 백업하는 방법이다. 따라서 풀 백업에 비해서 빠르며 백업 용량도 작게 된다. 차등 백업은 증분 백업과 비슷하거나 백업하는 대상이 다르다. 증분 백업은 이전 백업 이후 변경되거나 생성된 파일만을 백업하는 데에 비해 차등 백업은 이전 백업 이후에 변경된 파일을 백업하는 것에서는 동일하나 백업이 완료 된 후 백업을 했음을 나타내는 속성을 활성화 시키지 않는다. 그래서 현재의 백업 이후인 다음 백업에서도 현재 백업을 했던 파일들이 그대로 모두 다시 백업을 하게 된다. 따라서 백업의 소요되는 미디어도 증분 백업보다는 많다. 하지만 풀 백업에 들어가는 미디어 보다는 작은 용량을 필요로 한다. 차등 백업의 장점은 풀 백업 뒤에 차등 백업을 했다고 가정하면 복구할 때 풀 백업을 먼저 복원하고 여러 번의 차등 백업 중에서 마지막 차등 백업만을 복원하면 전체 파일시스템이 마지막 백업 시점까지 완전히 복구가 된다. 이에 반해 증분 백업은 풀 백업을 복원하고 매 증분 백업에 대해서 차례대로 복원을 해야

만 완전한 파일시스템으로 복구가 된다.

데이타 압축은 파일 기반의 백업에 있어서 필요한 옵션일 수 있다. 파일 기반의 백업은 파일시스템 전체를 백업 미디어에 저장하기 때문에 미디어 소모가 많다. 만약 백업 수행 시 데이타를 압축해서 하게 된다면 어느 정도 백업 미디어를 줄 일 수 있을 것이다.

파일 기반 백업은 파일과 디렉토리 구조, 그리고 이들의 속성을 그대로 보존해서 백업 미디어에 저장을 하게 된다. 백업 프로그램은 백업하고자 하는 파일시스템을 순차적으로 탐색 하면서 백업을 수행하게 된다. 파일 기반 백업의 장점은 파일시스템 하에서 백업을 수행하기 때문에 각각의 파일에 대해서 쉽게 복원이 가능하다. 또한 원하는 파일만을 백업할 수도 있고 반대로 배제하고 싶은 파일을 지정해서 그 파일들만을 배제한 채 백업을 할 수도 있다. 하지만 파일들이 파일시스템 내에서 연속적으로 할당되어 있다고 볼 수 없고 대체로 디스크 내에 여러 곳에 분할되어서 저장되어 있기 때문에 파일단위로 백업을 한다는 것은 디스크 내에 여러 곳을 탐색을 해야 하기 때문에 이로 인한 디스크 오버헤드가 증가하게 된다. 또한 하나의 파일에 대해서 조금의 변화가 있다고 하더라도 그 파일 전체를 백업해야한다. 현재 저장 매체의 기술이 발달하고 저장 공간의 비용이 줄어들었다고 하더라도 수 바이트 또는 수십 바이트의 데이타 수정에 대해서 전체 파일을 백업하는 것은 매우 비효율적이다. 물리적 백업은 블록 레벨에서 증분과 차등 백업을 지원하여 논리적 백업보다 저장매체의 효율이 좋은 편이다. 또한 매체를 순차적으로 탐색하기 때문에 디스크 오버헤드가 적다. 그러나 물리적 백업은 파일별 복구가 불가능한 단점을 가지고 있다[4].

3. 파일 지문 기반 백업 시스템 설계

본 논문에서 제안하는 방식은 백업 데이타 블록에 대해서 파일 지문을 부여하고 파일지문이 동일할 경우 하나의 사본만을 저장한다. 또한 각 파일을 구성하는 데이타 블록에 대해서 블록 지문을 생성/관리하는 방법으로 파일간의 중복되는 블록을 줄일 수 있다. 따라서 다단계의 중복 제거 기법을 도입함으로 저장되는 데이타의 양을 효과적으로 감소시킬 수 있으며 결과적으로 저장 공간을 효율적으로 사용할 수 있다.

3.1 시스템 설계 원칙

본 연구에서 제안하는 백업 시스템의 구조는 확장성 및 효율성을 증대시키기 위하여 시스템을 세 부분으로 나누어 설계하였다. 전체 시스템의 구조는 그림 1에서 보이는 것과 같이 클라이언트, 메타 데이타 서버, 클러스터 파일 서버로 구성되어 있다.

첫 번째 단계에서는 파일 수준에서 중복을 제거하기 위하여 클라이언트는 파일의 해시 리스트를 생성한 후에 중복된 파일 해시가 있는지 MDS에 요청을 한다. MDS는 파일 서버에서 관리하고 있는 파일 해시의 중복 유무를 체크하고 중복된 파일 해시 리스트를 클라이언트에 전송함으로 중복파일 제거를 수행한다. 그림 2에서 볼 수 있듯이, 클라이언트에서 생성된 "File C"의 해시 값 0X67AB"이 MDS 내부의 데이타베이스에 있는 기존 파일 해시 데이타와 동일함으로 "File C"를 백업할 필요가 없는 것이다. 두 번째 단계에서는 백업 대상이 되는 파일에 대해서 블록 중복을 제거하는 작업을 수행한다. 이를 위하여 클라이언트는 첫 번째 단계에서 중복이 아닌 파일로 판명된 대상 파일에 대해서 블록

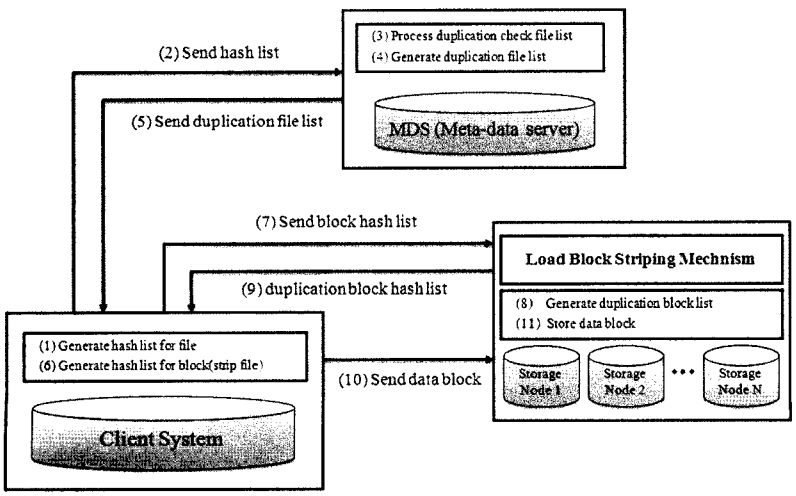

그림 1 전체 시스템 구성도

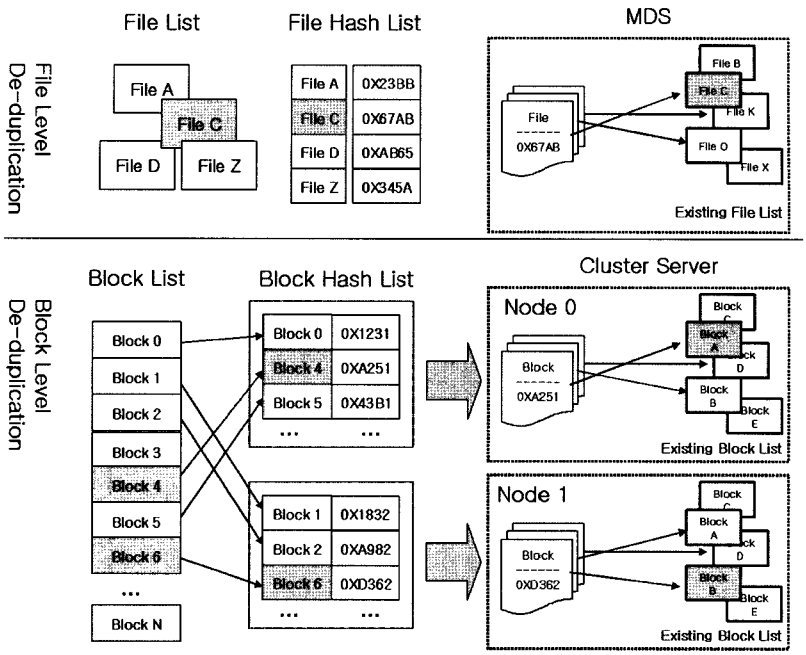

그림 2 다단계 중복 제거 기법

해시 리스트를 생성한다. 블록 해시 리스트는 각 노드에 저장되는 블록 해시의 집합이다. 블록 해시 리스트를 만들기 위해서는 각 노드에 파일의 어떤 블록이 저장되어 있는지를 알 수 있어야 하는데, 본 연구에서는 클러스터 노드 수에 의하여 파일이 스트라이핑(striping)되어 있으므로 클러스터 노드의 수로 모듈러 연산을 수행하여 같은 나머지 값을 갖는 블록 집합으로 블록 해시 리스트를 생성하게 된다. 이때 블록 번호를 클러스터 노드 수로 모듈러 연산을 하는 것이 아니라 블록 해시 값에 대해서 모듈러 연산을 수행하게 된다. 이와 같은 방법을 사용하는 이유는 같은 해시 값을 갖는 중복된 블록들을 단일 노드에서 처리하려는 목적 때문이다. 블록 해시 리스트를 클러스터 서버로 전송하고, 클러스터 서버는 블록 해시의 중복 유무를 체크하여 클라이언트에 중복된 블록 해시 리스트를 전송한다. 중복 블록을 판단한 후에, 클라이언트는 중복이 아닌 블록을 클러스터 서버로 전송하고, 클러스터 서버는 각 노드에 블록을 저장한다. 그림 2에서 볼 수 있듯이 하나의 파일은 다수의 블록의 집합으로 분할이 되고, 같은 노드에 있는 블록의 집합에 대해서 중복 블록이 존재하는지 요청을 하게 되는 것이다. 이와 같은 방법을 통하여 다단계 중복 제거를 수행할 수 있다.

3.2 파일 지문(File Fingerprint)

파일 서버에서 중복된 파일을 제거하기 위하여 서로 다른 파일의 직접적인 비교는 오버헤드가 큰 작업이다. 파일의 처음부터 끝까지 비트단위의 확인을 거쳐야 하고 모든 파일을 서로 비교해야 하기 때문이다. 그러나 파일을 대표하는 몇 개 비트의 비교만으로 작업을 마칠 수 있다면 중복된 데이타를 큰 오버헤드 없이 찾아낼 수 있을 것이다. 몇 개 비트로 파일을 대표 할 수 있는 방법이 해시 함수를 사용한 파일 지문 기법이다. 파일 또는 파일을 구성하고 있는 디스크 블록을 MD5[5], SHA1[6]과 같은 해시 함수를 사용해 축약된 비트 형태로 생성할 수 있으며, 축약된 비트의 비교로 중복된 파일이나 블록을 쉽게 알아낼 수 있다. 그러나 해시 함수는 임의의 길이의 메시지를 고정된 길이의 메시지로 변환하는데 있어 해시함수의 치역 집합이 정의역 집합보다 큰 범위이기 때문에 서로 다른 메시지가 동일한 해시 값으로 맵핑되어 해시 충돌(Collision)이 일어날 수 있다. 해시 충돌이 일어난다면 파일지문의 값이 같아도 입력 메시지인 파일이 서로 다르므로 파일 지문을 사용하는 백업 시스템에 오류를 발생시킬 수 있다.

해시 충돌은 잘 알려진 수학 문제인 생일 패러독스와 유사한 성격을 가지고 있다. 생일 패러독스란 N명 중 특정한 사람을 골라 N명 중 생일이 같은 사람이 나올 확률이 50% 이상이 되려면 N이 183명을 넘으면 되지만 N명중 생일이 같은 두 명이 나올 확률이 50%이상이 되려면 N이 23명만 넘으면 된다는 것을 말한다. 식 (1)은 생일이 같은 두 사람이 나올 확률과 동일하다.

Prob = 1 - (N 명의 생일이 모두 다를 경우) (1)

N명의 생일이 다를 확률을 일반화 하면 M개의 통에 N개의 구슬을 임의로 넣어 N개의 구슬이 모두 다른 통에 들어갈 확률과 같다. 생일이 같은 두 사람이 나올 확률의 수식은 다음과 같다.

$$\epsilon = 1-1\times\left(\frac{M-1}{M}\right)\times\left(\frac{M-2}{M}\right)\times\cdots\times\left(\frac{M-N+1}{M}\right) \quad (2)$$
$$= 1-\prod_{i=1}^{N-1}\left(1-\frac{1}{M}\right)$$

매클로린 급수 전개를 바탕으로 x가 매우 작을 때 $e^{-x}\approx 1-x$로 근사화 할 수 있다. 따라서 식 (2)를 식 (3)으로 바꾸어 나타낼 수 있다.

$$1-\prod_{i=1}^{N-1}\left(1-\frac{1}{M}\right)\approx 1-\prod_{i=1}^{N-1}e^{-\frac{i}{M}} \quad (3)$$

이는 곧 생일이 같은 두 명이 나올 확률이다. 이 수식은 해시 함수의 출력이 고르게 분포한다는 가정을 전제로 하고 있으며, 입력데이타의 수와 표현할 수 있는 해시의 수를 통해서 지문이 일치할 확률을 나타내고 있다.

$$1-\prod_{i=1}^{N-1}e^{-\frac{i}{M}} = 1-e^{-\sum_{i=1}^{N-1}\frac{i}{M}} = 1-e^{-\frac{1}{M}\frac{N(N-1)}{2}} \quad (4)$$

도출된 식 (4)로 해시 함수의 충돌 확률을 구한다고 할 때 시스템의 용량을 Exa Byte ($=2^{60}$), 디스크 블록을 8KByte ($=2^{13}$)라고 가정하면 시스템 전체가 저장할 수 있는 디스크 블록의 수는($=2^{47}$)이다. M의 크기에 MD5, SHA1 해시 함수의 축약 데이타 수(2^{128}, 2^{160})를 대입하고 N의 크기에는 전체 디스크 블록의 수($=2^{47}$)를 대입하면 약 10^{-10}, 10^{-17}의 이하의 확률을 갖는다. 그러므로 대중적인 해시 함수들의 결과값들이 임의적으로 분포한다면 해시 충돌확률은 극히 미세하다고 볼 수 있다.

특히 SHA1은 해시충돌 사례가 아직 발견되지 않았으며 확률이 그냥 작은 것이 아니며 EXA 바이트 단위의 백업 시스템 보다 작을 경우 10^{-17} 보다 더 작을 것이 명백하다. 시스템 운영상에서 일어나는 여타 에러에 비하면 거의 일어나지 않을 것이라고 생각해도 될 정도의 확률이다. 그러나 해시 충돌이 일어날 경우를 대비한다면 두 가지 이상의 해시 함수를 운영하거나 몇 자리의 비트를 저장하여 중복이 발생했을 경우 비트를 비교하여 해시 충돌의 위험을 방지 할 수 있다. 본 논문에서는 해시충돌 사례가 발견되지 않은 SHA1 해시 함수를 사용하고 구현을 용이하기 위해 SHA1 해시 충돌 상황을 고려하지 않았다. SHA1 해시는 add, and, or, xor, rotl 연산을 하며 블록 당 80 라운드의 연산 과정을 거치며, 160bit으로 이루어진 이진 데이타를 생성한다.

3.3 클라이언트 시스템의 동작 원리

클라이언트는 사용자가 백업할 수 있도록 그림 3과 같은 인터페이스를 제공하며 사용자가 인터페이스에서 백업을 수행할 디렉토리를 선택하여 백업을 수행하면 파일 시스템의 디렉토리를 재귀적으로 검색하여 디렉토리와 파일의 리스트를 생성한다.

클라이언트는 생성된 리스트의 정보를 이용하여, 파일 SHA1 해시 리스트와 디렉토리의 SHA1 해시 리스트 그리고 파일 및 디렉토리의 속성을 요약하여 메타데이타를 생성한다. 메타데이타는 중복 파일 유무를 판단하기 위하여 MDS로 전송되며, MDS는 데이타베이스에 저장된 파일 해시의 중복유무를 체크하여 중복된 파일 리스트를 전송한다. 클라이언트에서는 전송받은 중복 파

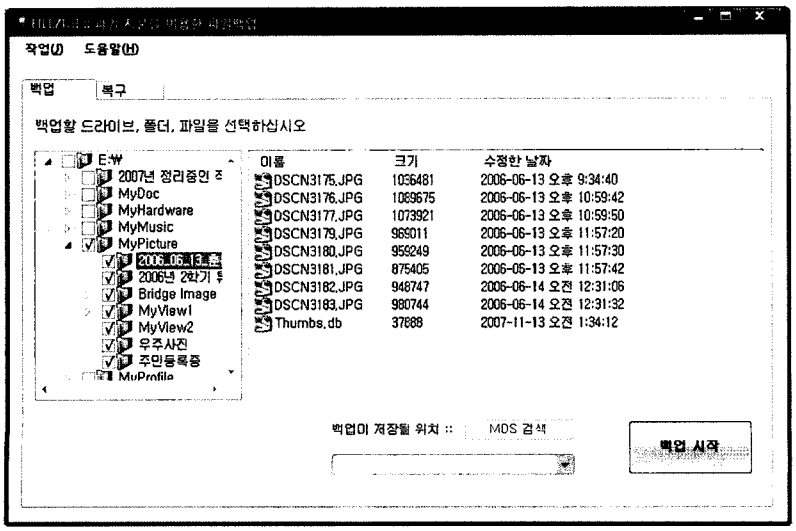

그림 3 클라이언트 화면

```
Algorithm 1: Generate_Block_HashList
  Input: FileStreamList
  Result: HashList
  begin
    Index ← 0;
    for FileIndex ← 0 to FileStreamList.Count do
      while ReadBlock ← Read(FileStream[FileIndex], BlockSize)
      do
        HashValue ← Sha1(ReadBlock);
        LogSave(HashValue, Index, FileStream);
        Index ← Index + 1;
        if HashValue ∉ HashList then
          HashList ← HashList ∪ HashValue;
        end
      end
    end
    return HashList;
  end
```

그림 4 블록 해시 리스트 생성 알고리즘

일을 제외하고 파일 리스트를 생성하며, 그림 4의 알고리즘을 이용하여 블록을 스트라이핑 하고 블록 해시 값을 얻는다.

알고리즘의 동작 원리는 다음과 같다. 파일을 일정한 블록 크기로 분할하여 메모리로 적재한 후에, 블록을 SHA1 알고리즘으로 해시시킨다. 해시 값과 인덱스 그리고 파일 정보에 대한 값을 로그에 기록하고 만약에 해시 값과 동일한 해시 정보가 있는 경우는 파일 내에 중복된 블록이 존재하는 것으로 판단한다. 이와 같이 반복해서 모든 블록에 대하여 해시 값을 생성한 후에, 중복되지 않은 블록에 대해서 해시 리스트를 완료한다.

각 노드에 블록을 스트라이핑하기 위해서는 일정한 규칙에 의해서 블록의 집합을 생성해야 한다. 본 연구에서는 같은 노드에 기록될 블록의 집합을 생성하는 기준으로 해시 데이타를 노드의 수로 모듈러 연산을 수행한 후에 나머지 값을 각 블록이 속하게 될 노드를 결정하는 방식을 취하였다. 그림 5는 이와 같이 블록을 각 노드에 스트라이핑 해주는 알고리즘을 기술하고 있다. 스트라이핑 매커니즘은 넘버 스페이스 상에서 160bit 블록 해시 값을 기준으로 해시리스트를 각 스트라이프 노드에 분배하는 원리이다. 블록 해시 값의 분포가 고르다고 가정했을 때 블록 해시 값에 따라 블록 데이타도 고르게 저장될 것이다. SHA1 해시 함수는 해시 결과의 분포를 봤을 때 그림 6과 같이 고른 분포를 나타내고 있

```
Algorithm 2: Block Striping Algorithm
  Input: NodeList, HashList
  begin
    Modulo ← 2^160 ÷ NodeList.Count;
    for Index ← 0 to HashList.Count do
      HashValue ← HashList[Index];
      NodeIndex ← HashValue ÷ Modulo;
      Node ← NodeList[NodeIndex];
      Node.HashList ← Node.HashList ∪ HashValue;
    end
  end
```

그림 5 스트라이핑 노드 할당 알고리즘

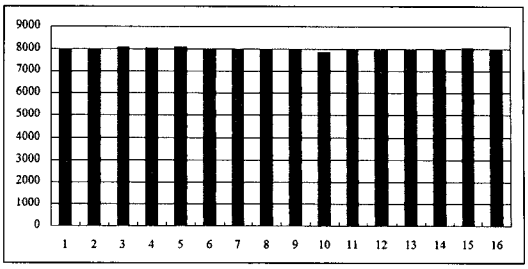

그림 6 해시 데이타를 이용한 블록 할당 결과

다. 따라서 라운드 로빈, 여유 공간 우선순위 등과 같은 다양한 파일 서버의 속성을 필요로 하는 알고리즘을 거치지 않기 때문에 백업 절차가 간소화 될 수 있다.

예비 실험 결과, 제안된 방식이 전체 노드에 블록을 균일하게 분산시키는 효과가 있음을 알 수 있었으며, 그림 5는 실험 결과를 보여주고 있다. 그림에서 X축은 노드 번호이고 Y축은 할당된 블록의 개수를 나타낸다.

클러스터 노드들의 정보를 MDS로 부터 전송받으면 클라이언트는 클러스터 노드들과 연결을 설정하며, 연결 과정이 끝나면 스트라이핑 과정에서 생성된 블록 해시 리스트를 클러스터 노드에 전송하여 블록의 중복을 체크한다. 클라이언트와 1:1통신을 하는 클러스터 노드는 블록 지문과 블록을 관리하며 클라이언트로부터 블록의 저장과 전송을 요청 받는다. 블록 저장과 전송 모두 블록의 해시 값을 검색하는 작업을 하며 해시 테이블을 삽입과 검색에 유리한 자료구조를 사용하는 데이타베이스 테이블에 저장한다.

3.4 메타데이타 관리 서버 동작 원리

MDS 프로그램은 메타데이타 관리, 파일서버 관리, 파일 비교의 기능을 수행한다. 클라이언트에서 전송한 메타데이타는 파일시스템 정보와 중복제거를 위한 파일 지문을 가지며 이를 데이타베이스에 저장되어 있는 기존의 파일지문과 비교하여 파일의 중복을 제거한다. 또한 파일서버들의 용량정보, IP주소 정보를 업데이트하고

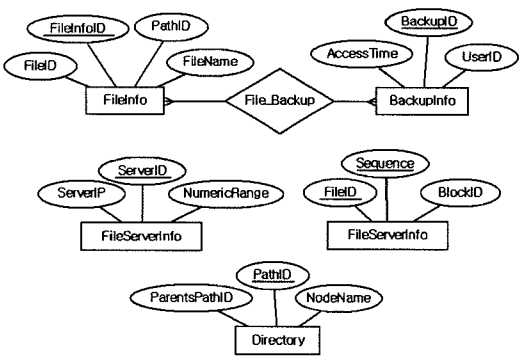

그림 7 MDS를 관리하는 DBMS 스키마

이를 데이타베이스에 저장하는 기능을 수행한다.

메타데이타 저장 스키마는 6개의 테이블로 구성되어 있으며, 각각의 테이블의 내용을 통해서 백업과 복구를 처리 할 수 있다. 테이블의 구성은 파일들의 데이타를 저장하는 FileInfo 테이블, Backup 정보를 저장하는 BackupInfo 테이블, 파일 시스템의 디렉토리를 저장하는 Directory 테이블, 클러스터 노드의 정보를 저장하는 FileServerInfo 테이블, 파일을 구성하는 블록 ID와 순서를 저장하는 BlockInfo 테이블로 구성되어 있다.

다음은 테이블을 이루는 속성의 설명이다.

FileInfo(FileInfoID, FileID, PathID, FileName) : FileInfoID는 파일을 입력 메시지로 하는 SHA1 해시 값이며, PathID는 디렉토리의 절대경로를 입력 메시지로 하는 SHA1 해시 값이다. FileName은 파일이름을 나타내는 파일 속성이다.

Backup(BackupName, accessTime, userID) : BackupName 속성은 유일한 백업 프로젝트 이름을 나타내며, accessTime 속성은 최근 백업의 접속날짜이다.

Directory(PathID, ParentsPathID, NodeName) : PathID 속성은 디렉토리의 절대경로를 160bit로 축약해서 만든 값이다. *ParentsPathID* 속성은 PathID의 부모 디렉토리이다. PathID의 부모 디렉토리를 따라 접근하면 상위 PathID를 추적하면 RootPath에 도달하여 전체 경로를 표현할 수 있다.

File_Server(ServerID, ServerIP, NumericRange) : ServerID 속성은 클러스터 노드에 할당된 160bit 해쉬 값이다. 클라이언트에서 보내는 블록 해쉬 값을 포함하여 디스크 블록을 파일 서버에 전송한다. ServerIP는 IP를 정보 항목이고 NumericRange는 클러스터 노드에서 담당하는 범위이다.

BlockInfo(FileID, Sequence, BlockID) : BlockInfo 테이블은 파일의 블록정보를 저장하는 BlockID와 순서 값인 Sequence를 사용하여 파일 블록 정보를 관리한다.

그림 8은 백업이 수행된 시점에서 MDS에 저장된 테이블에 저장된 파일 해시와 블록 해시의 내용을 보여주고 있다.

3.5 백업의 수행과정

그림 9는 파일 수준의 중복제거를 하는 과정이다.

(1) 클라이언트는 SHA1 해시 함수를 사용하여 파일 해시 리스트를 생성한다.

614334639l7d23d43a0188a37b06ffc857d795e6	12c730e1e8046d344607b79b99f75e4386c7ae34	Thumbs.db
0c6dcffe9149f14c5faf32963d45169829ecb2aa	12c730e1e8046d344607b79b99f75e4386c7ae34	사진0009.jpg
e535ece2cb5bb164ee527959d47bad0db2d891eb	12c730e1e8046d344607b79b99f75e4386c7ae34	사진0017.jpg
4cb313ac6dc6b84ca6cee21dc9b341f3e2a03145	12c730e1e8046d344607b79b99f75e4386c7ae34	사진0018.jpg
3f0e44211ec01b859b453d9fa0e472179d01374b	12c730e1e8046d344607b79b99f75e4386c7ae34	사진0019.jpg
c47b31382fb85da85be3c0573585000d7ff39e0ec	5e737eab3da2eefdf2c4072376c436104ff0c0cd	m51_hallas_big.jpg
aafb23bec849a85157b369297e9f550a4061b20	5e737eab3da2eefdf2c4072376c436104ff0c0cd	s147_gendler_full.jpg
054a6ca5b69a26dd553505cb077a832b813b71c1c	5e737eab3da2eefdf2c4072376c436104ff0c0cd	Thumbs.db
6a1cdbe95fe930d54f195819c00d71c603967649	5e737eab3da2eefdf2c4072376c436104ff0c0cd	다국발명상문.jpg
5b9e1a91b05c07369be6b3564f97c3de8d8fd334	5e737eab3da2eefdf2c4072376c436104ff0c0cd	달마리상문.jpg
8841ba875e4b7d8295614bcb5fc805f5d85639ac	5e737eab3da2eefdf2c4072376c436104ff0c0cd	아이리스상문.jpg
9fa5abfdbdad9e9cb4ed8b99adedd89232bfe367	5e737eab3da2eefdf2c4072376c436104ff0c0cd	오로라.jpg
b87f73f61caf407f7993582ea4844d37b74a9449	5e737eab3da2eefdf2c4072376c436104ff0c0cd	오리온상문.jpg
049807c68ff35585a45a9006af28f02d12867fd3	5e737eab3da2eefdf2c4072376c436104ff0c0cd	궁성자본.jpg
50820dca82b113dffbd537e606a46dd213d97532	5e737eab3da2eefdf2c4072376c436104ff0c0cd	태이사.jpg
70cac5d655b7ac48789950d26d0fbd2e446a5638	90923f4aaa3eb63895760690b0532ca649a9a716	Thumbs.db
7f0dbcbd3aeda9ddeaf391f97dd924121c501535	90923f4aaa3eb63895760690b0532ca649a9a716	김병기_주민등록증.bmp
459b613c3ec582a203e4bde3ef18dc6029b55eac	90923f4aaa3eb63895760690b0532ca649a9a716	김병기_통장.jpg
db518abe0ff81d6ce708361bacd8e2488dd3d824	90923f4aaa3eb63895760690b0532ca649a9a716	통Back.jpg
c79bf1e4eed07a2c4c3002221147a325689e70a0	90923f4aaa3eb63895760690b0532ca649a9a716	통Front.jpg
57825d65fa96f9fbc6c62fe5d5995a8ac96591aa	90923f4aaa3eb63895760690b0532ca649a9a716	통_뒷면.jpg
b2e39bd821df4047cf378a312b41b585a7bd62bf	90923f4aaa3eb63895760690b0532ca649a9a716	통_앞면.jpg

648 rows in set (0.00 sec)

b2e39bd821df4047cf378a312b41b585a7bd62bf	27a1f302139a4f595d750a9ac899927c07684b43	71
b2e39bd821df4047cf378a312b41b585a7bd62bf	e8939d5bdc68fe93e2afec493b828f4f5a3947ea	72
b2e39bd821df4047cf378a312b41b585a7bd62bf	f389d4fae1592c0a4eb71fc3312f35c69935503	73
b2e39bd821df4047cf378a312b41b585a7bd62bf	d1b644bea2cf2897334208883753a322c7481afc0	74
b2e39bd821df4047cf378a312b41b585a7bd62bf	0147ca7692599f9f2faafba481b5543bb45f9338	75
b2e39bd821df4047cf378a312b41b585a7bd62bf	fbf83316d816e24935ec13e8060248b42f01edb4	76
b2e39bd821df4047cf378a312b41b585a7bd62bf	bd1fc54762b3dbbdd2b1eb1f4d2df957756ac4bf	77
b2e39bd821df4047cf378a312b41b585a7bd62bf	6f4d69b5d9e56966ea8465a2a0779448dd3bbcf2	78
b2e39bd821df4047cf378a312b41b585a7bd62bf	b3637f353504d39b415967606e9c03ed9824bb2d	79
b2e39bd821df4047cf378a312b41b585a7bd62bf	9986ec711eaaa484e707bad2f1d27e307d09bc1e	80
b2e39bd821df4047cf378a312b41b585a7bd62bf	95b613cb6388aecbac827b917852792ee9236406	81
b2e39bd821df4047cf378a312b41b585a7bd62bf	86440c1def760c4caa0afe2497228547122l4a0b	82
b2e39bd821df4047cf378a312b41b585a7bd62bf	79c5f402f4f1b24d029b17ef2bcdfd340129c969	83
b2e39bd821df4047cf378a312b41b585a7bd62bf	a4afaff6cc84d18b3ea9c7c742d7a390d5632ec7	84
b2e39bd821df4047cf378a312b41b585a7bd62bf	23928b7a30e37f39fca059879f1c52efdbf9db8	85
b2e39bd821df4047cf378a312b41b585a7bd62bf	256ed6089a9085f80c3d4d8dfb272731e7caa331	86
b2e39bd821df4047cf378a312b41b585a7bd62bf	8a715a043672c804c7ba290e30eef5e180880dd3	87
b2e39bd821df4047cf378a312b41b585a7bd62bf	6bb49455aa4db9ac887d1d080bee96ff92060d41	88
b2e39bd821df4047cf378a312b41b585a7bd62bf	9bdab902650c95e83c9f75f7eaf255d7866db512	89
b2e39bd821df4047cf378a312b41b585a7bd62bf	22d68c93674a18bc1ec08caaad2c530eb932059c0	90
b2e39bd821df4047cf378a312b41b585a7bd62bf	312163e72a65706e65cb4861e540ebec936c395c	91
b2e39bd821df4047cf378a312b41b585a7bd62bf	14b12c67a6d443a7a0c6962965cbe2a171b70fd6	92
b2e39bd821df4047cf378a312b41b585a7bd62bf	09fcebce53cf90d06200343156cb18c9b48dfe79	93

53093 rows in set (0.02 sec)

그림 8 (위)파일 정보, (아래)블록 정보

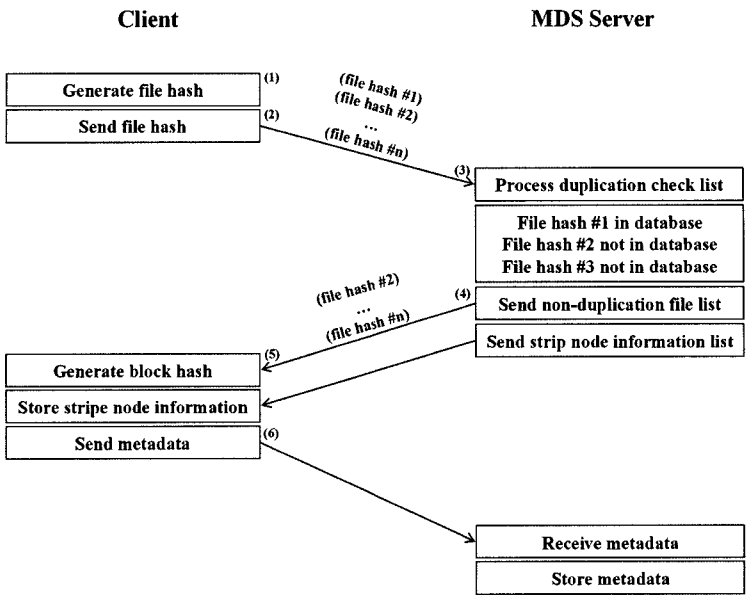

그림 9 파일 수준의 중복제거 과정

(2) 생성한 해시 리스트를 MDS로 전송한다.
(3) 해시 데이타베이스를 검색하여 해시 리스트의 중복을 찾아 제거한다.
(4) 중복이 제거된 파일 리스트와 평소 관리하고 있던 클러스터 노드 리스트를 클라이언트로 전송한다.
(5) 클라이언트는 MDS로부터 전송된 파일리스트의 파일들을 스트라이핑 하여 블록 해시 리스트와 파일 정보를 담은 메타데이타를 생성한다.
(6) 클라이언트는 블록 해시 리스트와 메타데이타 정보를 MDS에 전송하고 MDS는 전송받은 정보를 저장하여 파일 수준의 중복제거 과정을 마친다.

그림 10은 블록 수준의 중복 제거를 하는 과정이다.

(1) 클라이언트는 생성된 블록 해시 리스트를 각 노드별로 생성된 블록 해시리스트로 분류하는 과정을 거친다.

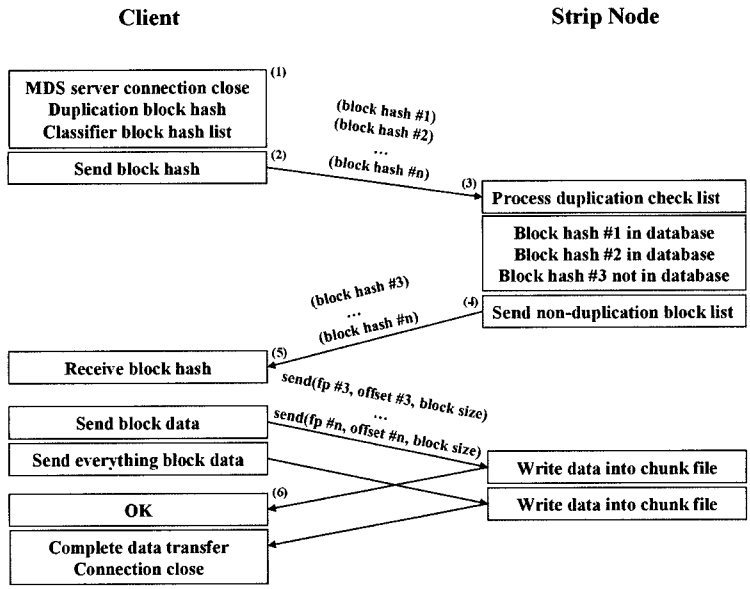

그림 10 블록 수준의 중복 제거 과정

(2) 각 클러스터 노드에 블록 해시 리스트를 전송한다.
(3) 각 클러스터 노드는 로컬 블록 해시 데이타베이스를 검색하여 전송된 블록 해시 리스트의 중복을 제거한다.
(4) 각 클러스터 노드는 중복이 제거된 블록 리스트를 클라이언트로 전송한다.
(5) 클라이언트는 클러스터 노드들로부터 전송된 블록리스트의 데이타들을 읽어 각 클러스터 노드에 전송한다.
(6) 클러스터 노드는 모든 블록데이타를 전송받고 저장하면 클라이언트에 종료패킷을 보낸다. 클라이언트는 모든 클러스터 노드들로부터 종료 패킷을 받으면 블록 수준의 중복제거 과정을 마친다.

4. 구현 및 성능평가

논문에서 제안한 시스템은 표 1과 같은 플랫폼에서 개발 및 운영되었다. 전체 구현 모듈은 세 부분으로 나뉘어진다. 첫 번째, 클라이언트는 Visual Studio 2005를 사용하여 개발하였으며 사용 언어로는 C#을 사용하였다. SHA1함수와 블록 해시를 관리하는 자료구조를 응용프로그램 계층에서 구현하였으며 클러스터링 기법의 구현과 블록 전송을 위해 쓰레드와 비동기 파일 입출력 관련 라이브러리를 사용하였다. 두 번째, MDS는 리눅스 상에서 서버로 수행되는 응용 프로그램이며, 메타데이타를 저장하기 위하여 mysql 데이타베이스를 사용하여 구현하였다. 특히 네트워크 부분에 있어서 성능이 매우 중요하므로, pthread와 event poll을 이용하여 패킷 송수신과 서비스 요청을 처리 하였다. 세 번째, 클러스터 노드는 리눅스에서 수행되는 서버이며, 클라이언트의 블록 요청을 처리하는 기능을 수행한다. 이를 구현하기 위하여 유닉스 파일 시스템에 대용량 크기의 파일을 생성하여 일반적인 파일 입출력 함수를 이용하여 블록 단위의 파일 송수신 루틴을 구현하였다. 향후 RAW DISK I/O 기법을 적용하여 성능 개선을 진행하고 일반적인 파일 입출력을 통한 클러스터링 시스템 구현과 비교를 할 계획을 가지고 있다.

실험에 사용된 컨텐츠는 MP3, JPG, MPEG, AVI 등의 미디어 콘텐츠 파일, 메일 데이타, 그리고 데이타베이스 파일들로 구성하였다. 또한 리눅스 배포본 이미지 등을 이용하여 블록의 중복 제거 기법에 대한 실험을 진행하였다.

첫 번째 실험은 제안된 시스템의 세부적인 구간별 성능을 분석하기 위한 시나리오로 구성되었다. 각 파일을 구성하는 데이타 블록의 크기는 16Kbyte로 고정하였으며, 1.5GByte 파일에 대해서 스트라이핑 노드 수를 최대 4개까지 증가하면서 백업시간을 측정하였다. 실험에서 세부적인 수행 시간을 측정하기 위하여, 실험 측정 구간을 다음과 같이 다섯 구간으로 나누었다.

(1) File Hash : 클라이언트에서 파일을 해시하는데 걸리는 시간
(2) Block Hash : 클라이언트에서 중복되지 않는 파일에 대해서 블록으로 분할하고 블록 해시 리스트를 생성하는데 걸리는 시간
(3) Metadata Save : MDS에 메타데이타를 업데이트하는데 걸리는 시간
(4) Block Hash Verify : 클러스터 노드에 중복된 블록이 있는지 체크하는데 걸리는 시간
(5) Data Transfer : 블록을 전송하는데 걸리는 시간

그림 11에서는 클러스터 시스템에 참여하는 노드의 수를 1에서 4로 증가시키면서 백업 시스템의 성능을 측정하였다. 전체 성능 분포 중에서 파일 해쉬 및 블록 해쉬를 처리하는 부분은 클라이언트와 MDS에서 발생되는 오버헤드이므로 클러스터링 기법을 적용하여도 성능에 큰 변화가 발생하지 않는다. 클러스터링 기법이 적용되었을 때에 데이타 전송 부분이 급격하게 성능이 향상되는 것을 확인할 수 있으며, 블록 해쉬에 대한 검증 부분도 클러스터 노드에서 이루어지기 때문에 성능 향상이 있다.

두 번째 실험은 메일 데이타(OUTLOOK 프로그램의 PST 파일)를 백업한 결과를 보이고 있다(그림 12). 메일 데이타는 1개의 파일로 구성되어 있지만, OUTLOOK 프로그램을 이용하여 메일 데이타를 송수신하거나 메일 삭제 등의 작업을 수행할 때마다 파일의 내용이 계속적으로 변하게 된다. 따라서 본 실험에서는 1개

표 1 개발 및 실험 플랫폼

S/W 플랫폼	Client	운영체제	Window XP
		개발도구	Visual Studio 2005
	MDS 파일서버	운영체제	Fedora Core 4
		Kernel Version	2.6.18
		개발도구	gcc 4.0.2, mysql 4.1.2
H/W 플랫폼	CPU	Pentium 4 3.0 GHz	
	Memory	512 MB	
	Hard Disk	웨스턴 디지털 WD-1600JS(7200/8MB)	
	Network	LAN 100.0 Mbps	

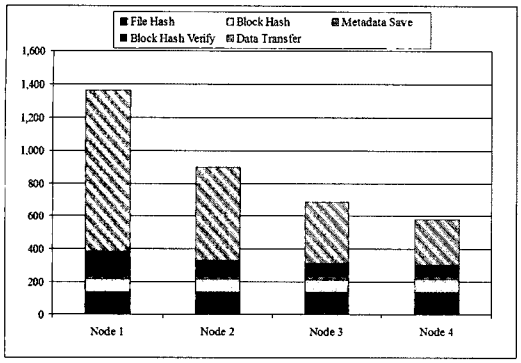

그림 11 마이크로 벤치마크(Micro-benchmark) 결과 (단위: 초)

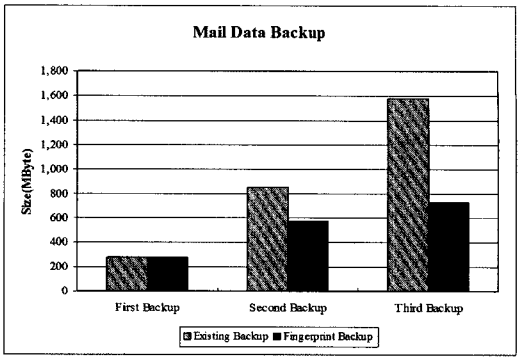

그림 12 메일 데이타 백업 결과

의 파일로 되어 있는 PST 파일에 대해서 백업 시간을 기준으로 버전을 유지하는 것으로 가정하고 실험을 수행하였다. 실험에서 Existing Backup 은 윈도우의 백업 프로그램을 이용한 저장 용량이고 Fingerprint Backup 은 중복제거 방법을 이용한 저장 용량을 의미한다. 실험은 초기 데이타 first에 새롭게 누적된 데이타를 추가하여 각각 second와 third 데이타를 생성하였다. 1차 백업 파일은 262Mbyte 크기였으며, 일주일 정도 메일을 운용하여 추가적으로 293MByte의 데이타가 추가되어 556 MByte의 파일이 생성되었고 추가적으로 메일 시스템을 운용하여 293MByte의 데이타가 추가되어 715MByte의 파일이 생성되었다. 각 시기에 생성된 메일 파일을 백업해야 하기 때문에 First Backup에서는 293Mbyte 파일이 저장되고, Second Backup에서는 293MByte와 556 MByte 파일이 백업되므로 818MByte 크기의 원본 파일이 저장되었다. Third Backup에서는 293MByte, 556 MByte, 715MByte의 각 파일이 백업되어야 하므로 총 1,533MByte 크기의 원본 파일이 저장되었다.

본 논문에서 제안하는 다단계 중복 제거 기법을 사용하는 경우, 파일 수준의 중복 제거는 발생되지 않고 블록 수준에서 중복 제거 기법을 통하여 상당 부분의 블록 중복 제거가 수행되는 것을 알 수 있다. 실험 결과 Third Backup에서 원본 데이타의 45%의 크기로 백업 데이타를 줄인 것을 알 수 있다. 그림 13은 데이타베이스(MySQL) 데이타를 백업한 결과를 보이고 있으며, 메일 데이타 백업과 마찬가지의 결과를 보이고 있다. 실험 결과 데이타베이스의 경우, Third Backup에서 원본 데이타의 42% 크기로 백업 크기를 줄였다.

그림 14는 중복된 블록이 자주 발생되는 것으로 예상되는 리눅스 배포판(CentOS)[7] 및 VMware[8] 이미지(Fedora Core 3, 4, 6)에 대해서 중복 블록 제거를 적용한 결과를 보이고 있다. 리눅스 배포판은 7.6GByte 크기의 이미지 중에서 10% 정도의 블록이 동일한 결과를 보이고 있으며, VMware의 경우도 비슷한 결과를 보이고 있다.

5. 기존 연구와의 비교 분석

본 연구와 관련이 있는 연구로는 컨텐츠 내에서 내용

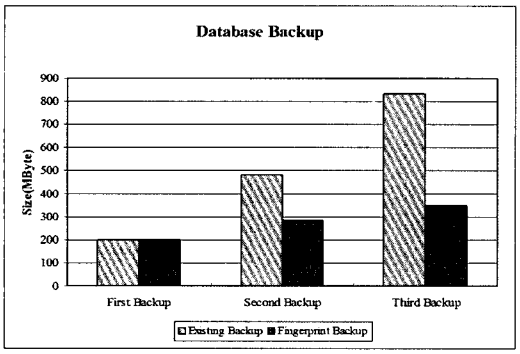

그림 13 데이타베이스 백업 결과

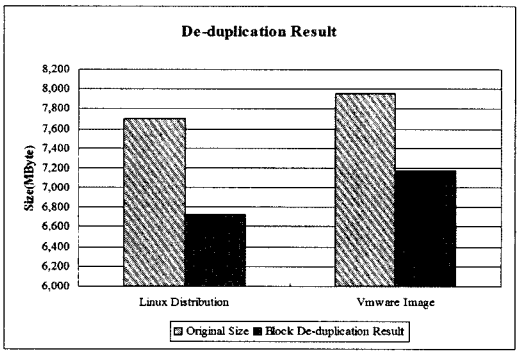

그림 14 블록 수준 중복 제거 결과(리눅스 배포판, VMware 이미지)

을 비교하여 중복된 정보를 찾아주는 Rsync[9], Tivoli [10]가 대표적이며, Rabin fingerprint[11], SHA1[6], MD5[5] 등의 해시 함수로 중복된 정보를 찾아주는 연구 등이 있다. 또한 하이브리드 성격을 가지고 있는 연구도 다수 진행이 되고 있다[12-15]. 이외에도 컨텐츠 전송 네트워크 등에서의 중복 파일 문제를 다루는 논문 [16] 및 파일 서버 관점에서 접근하는 연구 등이 다수 있다[17-21].

저장 오버헤드를 최소화하기 위하여 백업 시스템의 서로 다른 시스템의 파일이나 시스템 내의 다른 파일들의 중복된 내용을 찾을 수 있어야 한다. Rsync[9], Tivoli[10] 같은 버전 관리 툴들은 파일의 서로 다른 버전의 공통된 내용을 찾는 구조를 제공한다. 그러나 Rsync와 Tivoli는 버전정보를 위해 특화되어 있기 때문에 연관 없는 파일에서 공통된 데이타를 공유하는 것이 매우 어렵다. 서로 연관이 없는 파일에서도 공통된 데이타를 공유할 수 있어야 하는데 Rabin fingerprint[11], SHA1[6], MD5[5] 같은 함수들은 블록 정보를 계산된 해시 값으로 바꿀 수 있다. 서로 다른 파일의 블록들을 해시해서 해시 값이 동일하다면 블록들도 동일하다. 그러므로 파일지문을 백업 시스템에 충분히 이용할 수 있는 것이다. 그러나 서로 다른 블록이 같은 해시 값을 가질 수 있다. 이를 해시 충돌(hash collision)이라고 하는데 해시 충돌 확률이 하드웨어가 에러가 일어날 확률보다 작기 때문에 해시 충돌은 백업 시스템에서는 큰 고려사항이 아니다.

중복을 찾아내기 위한 연구로 Contents Chunking 기법이 있는데 Break Marker 라는 48 byte window를 삽입하여 블록의 범위를 설정한다. Break Marker로 범위에 해당하는 블록을 해시해서 해시 값의 중복을 찾는 방법이다. 파일의 수정과 삽입 삭제가 일어날 때 변경된 내용을 고려할 수 있는 것이다. 그러나 파일시스템 레벨에서 Break Marker 삽입을 지원해야 하고 델타 인코딩을 제외하고 저장 용량 상 이득을 얻는 경우가 없다. 저장하는 블록의 크기가 동일하지 않아 백업 시스템 성능 향상에 제약을 주는 단점이 있다[15,19,20].

Venti-Dhash는 두 가지 시스템이 결합된 형태이다. Venti[12]는 하드디스크 상에서 Read-Only 정책을 통하여 저장된 데이타를 효율적으로 사용할 수 있는 시스템이다. 또한 응용프로그램에 독립적으로 사용할 수 있으며 SHA1 해시함수를 사용하여 파일의 중복을 제거할 수 있다. 다만, Venti는 Contents 관리에 신경을 쓰지 않으며 본 연구의 스트라이핑 노드와 같이 Block 관리에만 초점을 두고 있다. 또한 구현 방법에 있어서 Venti는 RPC(Remote Procedure Call)요청을 처리하여 블록 해시 비교와 블록의 저장을 수행한다. RPC를 사용하면 구현상 용이하지만 백업 단위의 처리가 아닌 블록 단위의 처리이기 때문에 백업 시스템 성능 향상에 불리하다. 본 연구에서는 백업 단위의 처리를 고려하였기 때문에 블록 전송 시간과 블록 해시 중복 제거 시간외에 불필요한 패킷 송수신이 일어나지 않아 RPC를 사용한 Venti보다 성능 향상에 유리하다.

DHash[13,14]는 Chord Lookup Protocol에서 동작되는 분산 해시 테이블이다. Chord Protocol는 관리자의 관리 없이 Ring 기반의 노드 토폴로지를 관리할 수 있는 프로토콜이며 DHash는 Chord 위에서 Block 키의 Successor 노드를 찾으며 파일을 Ring 토폴로지에 저장한다. Pastiche[15]는 Pastry 오버레이 네트워크를 이용하는 Peer-To-Peer 백업 시스템으로서 중앙 서버모델이 아닌 분산모델이다. 이는 각 노드로부터 CPU, 디스크용량, 네트워크 대역폭을 공유하여 백업 비용을 각 노드의 사용자들에게 분산 시켜 저렴하게 이용할 수 있다. 또한 서로 공유되는 데이타의 중복을 제거하기 위해 Rabin FingerPrint[11]와 SHA1를 사용하였다. 그러나 Peer-to-Peer 기반 시스템은 노드를 검색하기 위해 빈번한 라우팅이 필요하고 이는 네트워크의 부하를 불러온다. 또한 빈번한 가입과 탈퇴가 일어나는 노드들로 인하여 일관성을 유지하기 어려우며 백업 파일을 검색에 있어 일정 이상의 실패 확률을 불러온다.

6. 결론 및 향후연구

본 논문에서는 백업 시스템에 효율적으로 사용될 수 있는 클러스터 기반 저장 시스템에 대한 설계 및 구현을 기술하고 있다. 주요 아이디어는 파일 내용에 대한 직접적인 비교보다 비용이 적은 파일 지문 비교를 통해 파일 및 파일 블록의 중복을 판단하고 제거하는 방법을 제안하고 있다. 제안된 기법은 동일한 파일, 동일한 데이타 블록의 중복을 제거할 수 있으므로 일반적인 백업 보다 저장 공간의 효율이 높고 중복된 데이타만큼 데이타 전송 및 입출력에 소요되는 오버헤드를 줄일 수 있다.

제안된 저장 시스템은 데이타 전송 시간과 입출력 시간을 줄이기 위하여 데이타 블록을 여러 개로 나눠 각 노드에 효율적으로 저장하는 클러스터링 기술을 사용하였다. 특히 데이타 블록에 대한 해쉬 값을 기준으로 스트라이핑 노드에 블록을 분산시킴으로 개별 노드에서 중복된 블록을 효율적으로 관리할 수 있는 부분이 특징이다. 이와 같은 방식은 데이타 전송 속도를 크게 줄일 수 있어서 백업 시스템에 적합함을 보이고 있다. 제안된 시스템의 유용성을 확인하기 위하여 다양한 실험을 수행하였으며, 실험 결과 클러스터링 기술과 파일지문을 이용한 다단계 중복제거 기법이 저장 공간을 효율적으로 관리하는 것을 보였다. 향후 연구과제로는 제안된 시

스템의 성능을 개선하기 위한 방안으로 다양한 압축 기법 및 비트 수준의 중복 제거 기법을 적용하는 방안에 대해서 연구를 수행할 계획이다.

참 고 문 헌

[1] KyoungSoo Park, Sunghwan Ihm, Mic Bowman, and Vivek S. Pai. "Supporting Practical Content-Addressable Caching with CZIP Compression," In Proceedings of the USENIX Annual Technical Conference, Santa Clara, CA, June 2007.

[2] Storage Networking Industry Association, Backup/Recovery Tutorial, 2001.

[3] A. Tridgell. Efficient algorithms for sorting and synchronization. PhD thesis, The Austrailian National University, 1999.

[4] M. Ajtai, R. Burns, et al. "Compactly encoding unstructured inputs with differential compression," Journal of the Association for Computing Machinery, 2000.

[5] R. L. Rivest, "The MD5 Message Digest Algorithm," Request for Comments(RFC) 1321, Internet Activities Board, 1992.

[6] RFC 3174, "US Secure Hash Algorithm 1 (SHA-1)"

[7] Centos home page, http://www.centos.org

[8] vmware home page, http://www.vmware.com

[9] http://www.samba.org/rsync/

[10] http://www.ibm.com/tivoli

[11] M. O. Rabin. "Fingerprinting by random polynomials," Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[12] QUINLAN, S., AND DORWARD, S. "Venti: a new approach to archival storage," In Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST) (2002).

[13] Josh Cates, Robust and Efficient Data Management for a Distributed Hash Table. Master's thesis, Massachusetts Institute of Technology, May 2003.

[14] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. "OpenDHT: A public DHT service and its uses," In SIGCOMM, 2005.

[15] COX, L. P., AND NOBLE, B. D. "Pastiche: making backup cheap and easy," In Proceedings of the 5th Symposium on Operating Systems Design and Implementation, Dec.2002.

[16] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. "Reliability and security in the CoDeeN content distribution network," In Proceedings of the USENIX Annual Technical Conference, 2004.

[17] S. Annapureddy, M. J. Freedman, and D. Mazires. "Shark: Scaling file servers via cooperative caching," In 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation, Boston, MA, May 2005.

[18] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. "An architecture for internet data transfer," In Proceedings of the 3rd Symposium on Networked Systems Design and Implementation, 2006.

[19] H. Pucha, D. G. Andersen, and M. Kaminsky. "Exploiting similarity for multi-source downloads using file handprints," In Proceedings of the 4th USENIX/ACM Symposium on Networked Systems Design and Implementation, 2007.

[20] C. Policroniades and I. Pratt. "Alternatives for detecting redundancy in storage systems data," In Proceedings of USENIX Annual Technical Conference, 2004.

[21] J. C. Mogul, Y. M. Chan, and T. Kelly. "Design, implementation, and evaluation of duplicate transfer detection in HTTP," In Proceedings of the 1st Symposium on Networked Systems Design and Implementation, 2004.

고 영 웅

1997년 고려대학교 컴퓨터학과(학사). 1999년 고려대학교 컴퓨터학과(석사). 2003년 고려대학교 컴퓨터학과(박사). 2003년~현재 한림대학교 컴퓨터공학과 조교수 관심분야는 운영체제, 실시간 시스템, 임베디드 시스템

정 호 민

2006년 한림대학교 컴퓨터공학과(학사) 2008년 한림대학교 컴퓨터공학과(석사) 2008년~현재 한림대학교 컴퓨터공학과 박사과정. 관심분야는 운영체제, 분산 파일 시스템, 임베디드 시스템

김 진

1984년 고려대학교 물리학과(학사). 1990년 Michigan State University 컴퓨터공학(석사). 1996년 Michigan State University 컴퓨터공학(박사). 1996년~2000년 건국대학교 충주캠퍼스 조교수. 2000년~현재 한림대학교 컴퓨터공학과 교수 관심분야는 생물정보학, 알고리즘, 의료정보