중복 데이터 관리 기법을 통한 저장 시스템 성능 개선

(Storage System Performance Enhancement Using Duplicated Data Management Scheme)

정 호 민 † 고 영 웅 ††
(Ho Min JUNG) (Young Woong KO)

요 약 기존의 전통적인 저장 서버는 중복 데이터 블록에 의해서 저장 공간과 네트워크 대역폭의 낭비가 발생되고 있다. 이와 같은 문제를 해결하기 위하여, 다양한 중복 제거 메커니즘이 제시되었으나, 대부분 Contents-Defined Chunking (CDC) 기법을 사용하는 백업 서버에 한정되었다. 왜냐하면 CDC 기법은 앵커를 사용하여 중복 블록을 쉽게 추적할 수 있기 때문에 파일의 업데이트를 관찰하기 유리한 백업 시스템에서 널리 사용되고 있는 것이다.

본 논문에서는 저장 시스템 성능을 개선하기 위하여, 새로운 중복 제거 메커니즘을 제시하고 있다. 범용적인 중복제거 서버를 구축하기 위한 효율적인 알고리즘에 초점을 맞추고 있으며, 이를 통하여 백업 서버, P2P 서버, FTP 서버와 같은 다양한 시스템에 활용이 가능하게 하는 것을 목표로 한다. 실험 결과 제안한 알고리즘이 중복 영역의 블록을 찾아내는 시간을 최소화하고 효율적으로 저장 시스템을 관리하는 것을 보였다.

키워드 : 파일 지문, 스트라이드, 해시, 중복, 저장 서버

Abstract Traditional storage server suffers from duplicated data blocks which cause an waste of storage space and network bandwidth. To address this problem, various de-duplication mechanisms are proposed. Especially, lots of works are limited to backup server that exploits Contents-Defined Chunking (CDC). In backup server, duplicated blocks can be easily traced by using Anchor, therefore CDC scheme is widely used for backup server.

In this paper, we propose a new de-duplication mechanism for improving a storage system. We focus on efficient algorithm for supporting general purpose de-duplication server including backup server, P2P server, and FTP server. The key idea is to adapt stride scheme on traditional fixed block duplication checking mechanism. Experimental result shows that the proposed mechanism can minimize computation time for detecting duplicated region of blocks and efficiently manage storage systems.

Key words : File Fingerprint, Stride, Hash, Duplication, Storage Server

· This work was supported by the Korea Research Foundation(KRF) grant funded by the Korea government(MEST)(No.2009-0076520)

† 학생회원 : 한림대학교 컴퓨터공학과
chorogyi@hallym.ac.kr
†† 종신회원 : 한림대학교 컴퓨터공학과 교수
yuko@hallym.ac.kr

논문접수 : 2009년 4월 21일
심사완료 : 2009년 10월 14일

정보과학회논문지: 시스템 및 이론 제37권 제1호(2010.2)

1. 서 론

컴퓨터를 이용한 문서 작업이 증가하고, 인터넷을 이용하여 데이터를 교환하는 것이 일반화되면서 저장 시스템(storage system)의 성능이 중요해지고 있다. 대부분의 저장 시스템은 사용자의 요구에 의해서 다양한 종류의 파일을 일정한 크기의 블록으로 변환하여 저장을 하게 되는데, 최근 들어 중복 데이터에 대한 처리가 주요한 관심이 되고 있다. 중복 데이터는 동등 계층 통신(P2P: peer to peer) 시스템, 백업(backup) 시스템, FTP 미러(mirror) 그리고 가상화(virtualization)[1] 시스템 등에서 높은 비율로 발생되고 있다[2]. 예를 들면 리눅

스 ftp 미러의 저장 서버에서 동일한 파일을 다양한 종류의 미디어 포맷으로 변환(CD이미지, DVD이미지, RPM 파일 등)하여 서비스하고 있으며, 리눅스 ftp 미러에서 데이터 중복이 50% 이상 발생하고 있다[2].

이와 같이 저장 시스템에 존재하는 중복된 파일 및 블록에 대한 효율적인 데이터 관리를 위하여 저장 시스템을 설계하는 단계에서 중복 제거를 위한 방안이 제공되어야 할 필요성이 있다. 일부에서는 저장 미디어에 대한 기술 발전으로 하드 디스크 및 플래시 미디어에 대한 단위 공간 대비 가격 비율이 점점 낮아지고 있기 때문에 복잡한 소프트웨어/하드웨어 설계 보다 간단한 설계 방식이 성능 면에서 좋을 것이라고 문제를 제기할 수 있다. 그러나 중복 제거 기능이 들어가면 여러 면에서 이득을 볼 수 있다. 첫째, 파일의 중복과 블록의 중복을 제거하여 저장 공간을 줄일 수 있다. 둘째, 파일 시스템에서 파일간의 중복된 데이터를 같은 디스크 블록으로 공유함으로써 캐싱(caching) 효과를 일으켜 성능을 향상 시킬 수 있다. 셋째, 모바일 환경에서 중복 데이터의 전송을 줄임으로서 에너지 소모와 네트워크 비용을 절감할 수 있다.

본 논문에서는 중복 제거를 지원하는 저장 시스템을 설계함에 있어서 기존에 널리 사용되는 Content-Defined Chunking(CDC) 기법의 단점을 극복하기 위한 방안을 제시한다. 기존의 중복 제거 시스템은 파일 내에 앵커(anchor)를 삽입하여 중복 블록을 쉽게 찾아낼 수 있는 가변 분할 기법을 널리 도입하고 있다. 하지만, 앵커를 사용하는 방식은 파일의 업데이트에 대한 정보를 파악하기 쉬운 백업 서버에서 이득을 볼 수 있으며 임의의 파일 간에 존재하는 중복 블록을 처리하는데 적합하지 않다. 이를 위하여 본 연구에서는 앵커를 사용하지 않는(anchor-free) 시스템 설계로 접근하여 기존의 방법들과 상세한 비교를 통하여 제안한 시스템의 유용성을 밝히고자 한다. 이를 위하여 본 논문에서는 다양한 서버에 사용할 수 있는 개선된 중복 블록 제거 기법으로 DBC_SS(Duplicate Block Check with Stride Scheme)과 DBC_ISS(Duplicate Block Check with Improved Stride Scheme)을 제안한다. 제안하는 방식의 주요 아이디어는 고정 분할 기법에 스트라이드(stride) 기법을 적용하는 방식이다.

2장에서는 중복 제거와 관련된 최근 연구 동향에 대해서 알아보고, 3장에서는 제안하는 중복 제거 저장 시스템의 설계 원리와 중복 제거 알고리즘의 동작에 대해 설명한다. 4장에서는 제안하는 시스템의 성능 평가 결과에 대해 기술하였으며 마지막으로 5장에서는 결론 및 향후 연구방향을 제시한다.

2. 관련 연구

저장 시스템이나 버전 컨트롤 프로그램 등 다양한 영역에서 중복 데이터를 제거해 네트워크 대역폭과 저장 효율을 향상시키려는 연구가 진행되었다. Chord[3], Pastiche[4] 같은 P2P 오버레이 네트워크에서 라우팅을 위해 그리고 파일의 중복 저장을 막기 위해 MD5[5], SHA1[6] 해시 함수를 사용하고 특히 파일 저장 시 파일의 해시를 만들고 서로 비교하여 같은 해시이면 중복으로 처리하여 저장을 막는다.

잘 알려진 Rsync[7]는 네트워크로 연결된 디렉토리의 데이터를 동기화 시켜주는 프로그램이다. 한쪽의 디렉토리에서 데이터의 삭제나 수정 같은 변화가 생길 경우 파일의 복사가 일어나는데 이때 롤링 체크섬(Rolling Checksum)이라는 중복 데이터를 검색하는 알고리즘을 사용해 새로운 데이터의 복사만 일어나게 하는 프로그램이다. 롤링 체크섬은 원본 파일을 일정하게 나누어 블록단위의 해시값을 생성하고 비교하려는 파일은 슬라이딩(Sliding) 기법을 이용하여 바이트 단위별로 모든 블록을 중첩(overlapping) 시켜 해시를 만들고 원본파일의 해시 리스트와 비교하여 중복된 부분을 찾는다. Rsync 방식은 모든 중복 데이터를 찾을 수 있어 중복제거 성능에서 우수하지만 수행시간이 길어 대용량 데이터를 처리하는 저장 시스템에서 사용이 어렵다.

Plan9[8]의 Venti[9]는 네트워크 저장 시스템에서 중복 데이터를 제거하여 저장하는 저장 시스템이다. Venti는 데이터를 저장할 경우 파일을 고정된 크기(8Kbyte)의 블록으로 나누고 각 블록에 SHA1 해시를 적용하여 160bit 크기의 해시를 만들고 전송한다. 만약 블록의 해시가 저장 서버에 있을 경우 중복으로 간주하여 블록의 저장을 피한다. Venti는 이전 저장 정보가 없이도 델타(delta) 백업의 효과를 낼 수 있으며 실험 결과에서는 다른 스냅샷(snapshot) 시스템과는 달리 30%의 저장 공간을 줄이는 것으로 나타났다. Venti의 단점으로는 중복 데이터를 놓치는 경우가 발생하는 것을 들 수 있다. 파일의 수정이 일어나게 되면 수정이 일어난 뒷부분의 블록들의 위치가 달라지며 Venti는 수정이 일어난 뒷부분의 블록을 다른 해시값으로 계산하여 중복을 인식하지 못한다.

LBFS[10]는 자주 끊기거나 품질이 좋지 않은 네트워크 환경을 위해 설계된 네트워크 파일 시스템이다. 네트워크 연결이 중단되고 다시 연결되어도 데이터 전송이 가능하도록 프로토콜을 설계하였고 네트워크 대역폭을 절약하기 위해 동일한 데이터의 재전송을 막는 기법을 고안하였다. LBFS에서는 CDC(Content-defined Chunks) 방식을 사용한다. Rabin Fingerprint[11]로 해시하여 특

별한 값을 반환하는 앵커(Anchor) 블록을 파일에 삽입하고 데이터 전송 전에 앵커 사이의 블록을 SHA1, MD5같은 해시 함수를 사용해 해시를 만들고 그 해시들을 전송하여 서버에서 캐싱된 해시 룩업(Lookup) 테이블과 비교하여 중복을 검색한다. CDC는 다양한 응용에 사용되었으며 클러스터 시스템에서 서버의 메모리 블록 이전에 사용되어 메모리 압축 효과와 데이터 전송효과의 증대로 메모리 성능을 향상시켰으며[12] 그 외에도 웹 캐싱, 전송(delivery) 네트워크, 소프트웨어 소스 배포 등에 사용되었다. CDC 알고리즘은 Venti와는 달리 파일의 수정이 일어나도 다른 블록의 해시값에 영향을 주지 않는다. 수정 시에 앵커 블록의 위치가 수정된 상황에 맞게 변하기 때문에 앵커 사이의 블록은 변함이 없게 되고 수정된 데이터만 변하게 되는 것이다. CDC에서 파일이 빈번하게 수정되는 상황이 발생하면 앵커들의 위치가 고르게 분포되지 않아서 블록 크기가 아주 작아지거나 커질 수 있고 특히 블록 크기가 작아지는 문제가 생긴다. 이를 해결하기 위해 블록의 최소/최대 크기를 설정하여 문제를 해결하고 있다. 그러나 수정이 많이 일어나면 결국 최소 크기를 가지는 블록들이 많아지며 Venti에서 사용하는 중복제거 방법보다 잦은 파일 입출력과 높은 통신 오버헤드를 일으키게 된다.

IDE[13]에서는 CDC에서 발생하는 각 블록 크기의 편차가 커지는 문제를 해결하기 위해 계층적 해시데이터 구조와 희소 행렬(sparse matrix)을 설계하여 블록 크기를 고르게 하였다. 또한 HP에서 TTTD(Two Thresholds, Two Divisors) 알고리즘[14]을 설계하여 앵커 사이의 블록 크기의 편차를 최소화했으며 이를 통해 클라이언트와 서버간 통신에 필요한 오버헤드가 줄어든 것을 확인하였다. DRED[15] 시스템은 델타 인코딩 전략을 사용해 웹 페이지, 전자메일과 같은 데이터의 중복을 효율적으로 제거하였다. 델타 인코딩은 시간적인 순서로 연관된 두 개의 데이터 집합의 파일 이름, 크기 등의 차이를 통해 압축하는 방식이다. DRED에서 사용하는 델타 인코딩 방식과 CDC, 압축, 슈퍼 핑거프린팅을 사용한 중복탐색 등의 다양한 기술을 적용해 REBL[16]을 만들었으며 중복제거와 수행속도에서 더 뛰어난 성능을 보였다.

이외에도 최근의 연구로 Data Domain의 Summary Vector, Stream-Informed Segment Layout 기법[17] 및 HP의 Sparse Indexing[18] 등이 제시되고 있다. 이와 같은 연구는 주로 대용량의 데이터를 처리하는데 있어서 문제가 되고 있는 해시 데이터를 효율적으로 처리하는데 초점을 맞추고 있다.

3. 중복 제거 저장 시스템 설계

본 연구에서 제안하는 시스템의 형태는 그림 1에서 보이는 것과 같이 클라이언트와 중복 제거 서버로 구성되어 있다. 클라이언트는 사용자가 저장할 파일을 선택할 수 있도록 인터페이스를 제공하고 파일을 전송할 수 있도록 설계하였으며 중복 제거 파일 서버는 클라이언트로부터 받은 파일들의 중복을 제거하고 저장 서버에 저장하는 작업을 수행한다.

클라이언트에서는 미리 저장할 파일들의 해시 리스트를 만들고 파일을 보내기 전에 해시 리스트를 서버에 전송한다. 파일 해시는 SHA1 해시 함수에 파일 스트림을 입력해 얻은 결과로 파일을 대표하는 값이다. 중복제거 서버에서는 기존에 저장된 파일에 대한 해시 데이터를 관리하고 있으며, 클라이언트에서 전송된 파일 해시를 파일 서버의 해시 리스트와 비교하여 중복된 파일을 찾아낼 수 있다. 중복된 파일을 탐색하는 작업은 해시 계산 비용만 소요되고 중복된 파일이 존재하면 재전

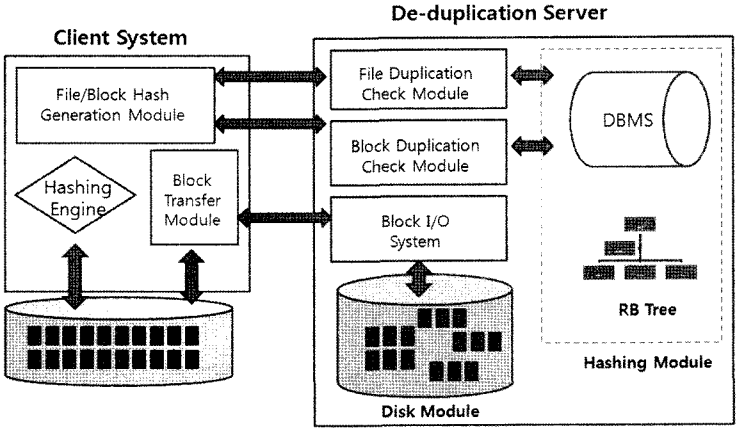

그림 1 시스템 레이아웃

송을 피할 수 있어 저장 비용과 전송 비용을 획기적으로 줄 일 수 있다. 클라이언트는 중복되지 않는 파일에 대해서 블록에 대한 해시 정보를 계산하여 중복 제거 서버로 전송한다. 중복 제거 서버는 블록에 대한 해시 정보를 전송 받아서 서버에 저장되어 있는 블록들의 해시 정보와 비교하는 작업을 수행하고 중복된 블록이 존재하는 경우에 클라이언트에게 그 정보를 돌려주게 된다. 클라이언트는 중복된 블록에 대해서는 실제 데이터 블록을 전송하지 않고 서버의 메타데이터만 업데이트 하는 작업을 수행하게 된다. 따라서 본 연구에서는 클라이언트에서 어떠한 해싱 방법으로 블록 해시 정보를 생성하여 서버로 전송하는지, 서버에서 블록의 해시 정보를 어떤 방식으로 탐색하는지가 매우 중요하다. 다음 절에서 해시의 개념 및 관리 방법에 대해서 설명한다.

3.1 해시 함수(Hash Function) 및 해시 충돌(Hash Collision)

데이터를 바이트 단위로 비교하여 중복 데이터를 찾는 일은 많은 시간을 소모한다. 데이터가 많을수록 파일의 개수가 증가 하는데 파일의 개수를 n이라고 했을 경우 비교 횟수가 n(n+1)/2으로 나타나므로 n이 커질수록 비교 횟수가 점점 증가한다. 또한 바이트 단위의 비교는 액세스 속도가 느린 디스크의 빈번한 입출력을 유발시키기 때문에 오버헤드가 크다. 그러나 직접적인 바이트 비교 대신 해시 함수를 사용한다면 큰 오버헤드 없이 중복 데이터를 찾을 수 있다. 일정한 크기의 데이터를 MD5, SHA1과 같은 해시 함수를 사용하면 해시값(checksum)이 생성되는데, 해시값이 서로 같다면 동일한 데이터로 보는 것이다. 해시값은 트리 구조와 같은 자료구조로 저장 할 수 있기 때문에 파일의 개수를 n이라고 했을 때 비교 횟수가 log n로 나타나므로 자료개수가 늘어날수록 비교 횟수의 상승폭이 직접적인 바이트비교 보다 작다.

해시 함수는 임의의 길이의 메시지를 고정된 길이의 메시지로 변환하는데 있어 해시 함수의 치역 집합이 정의역 집합보다 큰 범위이기 때문에 서로 다른 메시지가 동일한 해시값으로 매핑되어 해시 함수의 충돌(collision)이 일어날 수 있다. 해시 함수의 충돌이 일어난다면 해시값이 같아도 입력 메시지인 데이터가 서로 다르므로 해시 함수를 사용하여 중복데이터를 찾는 일은 의미가 없어진다. 그러나 해시 함수의 충돌 확률은 중복적으로 해시 함수를 사용하던지 블록의 몇 자리를 패리티로 구성하면 해시 충돌을 일상에서 발견할 수 없을 정도의 확률을 얻을 수 있다. 해시 함수의 충돌확률은 입력 데이터의 양과 해시 함수가 표현할 수 있는 해시값의 범위를 가지고 확률을 구할 수 있다.

해시 함수의 충돌 확률을 구하는 식은 표 1과 같다. n이 입력 데이터의 양이고 k가 해시값이 표현할 수 있는 범위이다. 만약 국가적인 사업을 위해 Exa byte를 담을 수 있는 저장이 필요하고 해시 함수에 입력되는 평균 크기를 8Kbyte라고 가정해보자. 이때 사용하는 해시 함수로 SHA1를 사용한다면 입력 데이터는 $2^{60}/2^{13}$, 즉 n은 2^{47}로 표현할 수 있고 k는 SHA1 해시값의 크기 160bit로 설정하여 계산하면 해시 충돌이 일어날 확률이 약 10^{-17}로 계산되는 것을 확인할 수 있다. 이와 같은 수치는 디스크 읽기 에러로 인해 데이터를 복구할 수 없는 확률(10^{-13}-10^{15})[19]보다 작은 것이며 시스템의 크기가 작아지거나 해시 충돌 확률을 작게 만드는 기법을 적용한다면 기하급수적으로 충돌 확률은 작아질 것이다.

표 1 해시 충돌 확률

$$1-\frac{n!}{(n-k)!n^k}>1-e^{-\frac{k(k-1)}{2n}} \qquad k=160, prob=10^{-17}$$

3.2 해시 데이터 관리

중복 제거 서버는 파일 해시들과 블록 해시들을 데이터베이스를 통해서 관리한다. 데이터베이스 테이블에 파일과 블록을 저장할 때 해시값을 주키로 설정하고 파일 이름, 순서와 같은 속성들을 추가하며 다음과 같은 구조로 되어 있다.

Table <File>

File Hash	Position	Block Hash
0xFA12..	0	0x12A3..
0xFA12..	8192	0x12A3..
0xFA12..	16384	0x12A3..
0xFA12..	24576	0x12A3..
0xFA12..	32768	0x12A3..
...	...	...
0xFA12..	1024000	0x12A3..

Table <Block>

Block Hash	Rabin	Size	Location
0x1212..	158741..	8192	110000000
0x3812..	234141..	8192	110016384
0xE112..	616215..	8192	110024576
0x311F..	258741..	424	110032768
0x2341..	431584..	8192	110008192
...	...	...	...
0x2353..	587434..	8192	111024000

그림 2 데이터베이스 저장되는 File, Block 테이블

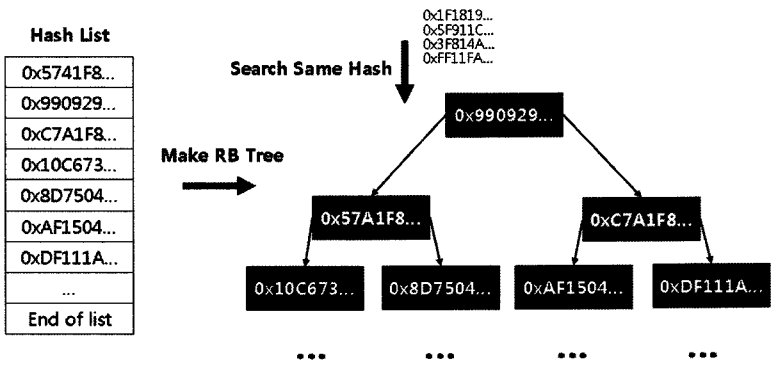

그림 3 RB Tree를 이용한 해시 데이터 관리

중복 제거 과정 전에 데이터베이스의 파일, 블록 해시를 각각의 RB(Red-Black) 트리에 로드하고 중복 제거 과정에서 해시들을 비교할 때 RB 트리의 비교함수를 사용한다. 본 논문에서는 데이터베이스의 SQL 쿼리를 사용하여 해시를 비교하면 디스크를 접근하는 오버헤드가 생기기 때문에 메모리에서 처리할 수 있도록 그림 3과 같이 RB 트리를 이용하고 있다. 이와 같은 방법은 추가적인 메모리 사용이 있지만, 빈번한 데이터 입력과 비교에 매우 효과적이다. RB 트리를 이용한 검색 기법을 사용하기 위해서, 본 연구에서는 RB 트리에 라빈 해시(64비트 정수)와 SHA1 해시(문자열)를 저장하고 비교할 수 있는 함수를 추가하였다. 중복 제거 과정이 완료되면 클라이언트에서 받은 파일은 중복 블록 집합과 나머지 블록 집합으로 나누어지고, 중복 블록이 아닌 경우는 디스크에 블록을 저장하고 블록의 해시, 저장된 위치들의 정보를 데이터베이스에 저장한다.

본 논문에서 사용하는 방법은 앞에서 지적했듯이 대용량 저장 서버를 구성하는 해시 검색에 있어서 성능을 향상시킬 수 있으나, 추가적인 메모리 사용이 문제로 제기될 수 있다. 예를 들어 블록의 크기가 8Kbyte라고 했을 때 저장 용량 1 Gbyte당 RB 트리에서 사용하는 메모리는 약 2.5 Mbyte가 필요하다. 1 Terabyte를 저장하게 될 때에는 메모리 용량은 무려 2 Gbyte 이상을 사용해야 한다. 하지만, 데이터베이스를 이용하는 경우에도 해시 데이터 처리를 위한 메모리 문제가 발생된다. SQL 쿼리를 통하여 해시를 비교하는 경우 데이터베이스가 운영체제의 버퍼 캐쉬를 활용하기 때문에 유용한 페이지, 디스크 블록 등이 페이징 아웃(paging out)이 되는 상황이 발생된다. 따라서 RB 트리를 활용하는 경우에 대한 정확한 오버헤드 문제는 추가적인 연구가 필요한 것으로 판단된다.

3.3 기존의 중복 제거 알고리즘 분석

기존의 중복 제거 알고리즘은 Venti의 Duplicate Block Check with Fixed-size Blocking(DBC_FS), Rsync의 Duplicate Block Check with Byte Shift(DBC_BS), LBFS의 (CDC:Contents-Define Chunking), DRED의 델타 인코딩(Delta-Encoding) 등으로 분류할 수 있다. LBFS에서 사용하는 CDC 알고리즘은 앵커(Anchor), 델타 정보가 있어야 하기 때문에 클라이언트 시스템에서 파일의 업데이트가 일어나는 순간을 감시해서 앵커 노드를 삽입하는 방법을 사용하고 있다. 따라서 백업 시스템과 같이 파일을 수정/저장하는 시스템에서 매우 유용하게 사용될 수 있다. 하지만, 다수의 파일 집합에서 중복된 블록이 다수 존재하는 P2P 시스템, FTP 미러 등에 있어서는 파일을 업데이트 하지 않고 단순히 저장만 하기 때문에 LBFS 알고리즘을 적용하기 어렵다. 따라서 본 연구에서는 CDC와 같은 가변 분할 방식을 제외하고 고정 분할 기법에 기반한 알고리즘에 대해서만 분석 및 비교를 하였다.

DBC_BS 기법은 중복제거에 필요한 데이터가 해시 데이터 외에는 없고 어떠한 파일 조작도 일어나지 않기 때문에 P2P 저장 서버, FTP 미러 같은 시스템에서 중복 제거 용도로 사용하기 적합하다. 본 연구에서는 기존에 알려진 DBC_FS, DBC_BS기법을 구현하였으며 본 논문에서 제안하는 스트라이드 기반 방식(DBC_SS, DBC_ISS)과 상호 비교하였다.

3.3.1 DBC_FS 알고리즘 분석

DBC_FS 알고리즘은 파일을 일정한 크기의 블록으로 나누어 SHA1, MD5와 같은 해시 함수를 적용하여 블록을 대표하는 해시를 생성한다. 생성된 해시가 기존의 저장에 저장되었던 해시 리스트와 비교를 통해 중복된 해시가 발견되면 연관된 블록도 중복 처리하는 것이다. 알고리즘에서는 파일 스트림(fd)을 입력 인자로 넣는다. Seek함수를 사용하여 원하는 파일의 위치로 이동한 뒤 SubString 함수를 사용하여 고정된 크기의 블록을 얻는다. SHA1 해시 함수를 사용하여 블록의 해시값(ShaHash)

```
Algorithm 1: DBC_FS
  Input: FileStream fd
  Output: MetaStruct
  begin
    fdsize ← Length(fd);
    offset ← 0;
    while fileoffset < fdsize do
      offset ← seek(fd, seek_cur);
      Block ← SubString(fd, BlockSize);
      Sha1Hash ← Sha1(Block);
      if CompareSha1Hash( Sha1Hash ) > 0 then
        MetaStruct ∪ MS(offset, Sha1Hash, false);
      else
        offset ← seek(fd, seek_set);
        MetaStruct ∪ MS(offset, Sha1Hash, false);
      end
    end
    return MetaStruct;
  end
```

그림 4 DBC_FS 알고리즘

을 얻고 CompareSha1Hash함수를 사용하여 RB 트리에 있는 SHA1 해시값들과 비교하여 메타데이터(MetaStruct)의 위치와 중복 여부를 저장한다. DBC_FS 기법은 다른 종류의 중복제거 알고리즘보다 블록을 나누는 기법이 간단하기 때문에 단위 시간당 처리하는 블록의 개수가 증가한다. 그러나 DBC_FS는 파일이 수정되는 경우에, 새로운 정보가 추가된 이후의 블록들이 해시 정보가 달라지기 때문에 중복 블록을 찾을 수 없다. 따라서 DBC_FS는 원본 파일에서 수정된 부분이 앞쪽에 위치할수록 중복을 찾을 확률이 낮아지게 되고, 파일의 수정이 일어나지 않거나 뒷부분에 덧붙여지는 경우에는 최고의 성능을 나타낸다.

3.3.2 DBC_BS 알고리즘 분석

DBC_BS 방식은 모든 중복 데이터를 찾을 수 있다는 장점을 가지고 있다. DBC_BS 방식은 모든 구간을 바이트 단위로 해시하기 때문에 SHA1 같은 해시 함수를 사용하는 것은 매우 비효율적이다. 따라서 일반적으로 라빈 핑거프린트(Rabin Fingerprint)와 같은 슬라이드 해시 함수를 사용한다. 여기서 슬라이드 해시 함수는 byte shift를 수행하면서 해시값을 계산하는 함수를 의미한다. 그러나 슬라이드 해시 함수는 해시 충돌이 일어날 확률이 SHA1에 비해 매우 높고 빈번하게 충돌이 발견되기 때문에 비신뢰적이다. 그러므로 슬라이드 해시 함수를 통해 1차적으로 동일한 해시가 있는지 찾고 만약 동일한 해시가 발견되면 SHA1과 같이 해시 충돌의 위험성이 없는 해시 함수를 사용하여 중복된 블록인지 확인하는 방법을 사용한다. 또한 수행 성능을 높이기 위해서 중복된 해시를 발견하면 슬라이드 하지 않고 다음 블록을 바로 SHA1 해시 함수로 검사하고 중복이 아니라면 다음 슬라이드 해시 함수를 사용해 연속적으로 해시를 생성하고 중복을 검사한다.

```
Algorithm 2: DBC_BS
  Input: FileStream fd
  Output: MetaStruct
  begin
    saveoffset ← 0;
    fileoffset ← 0;
    fdsize ← Length(fd);
    while fileoffset < fdsize do
      if saveoffset = fileoffset then
        ...;
        if MatchBlock(RabinHash, Sha1Hash, Block) > 0 then
          saveoffset ← fileoffset;
        end
      else
        ...;
        if fileoffset = ( saveoffset + 2 × BlockSize ) then
          MetaStruct ∪ MS(saveoffset, newblock);
          saveoffset ← saveoffset + BlockSize;
          Block ← SubString(fd, BlockSize);
        end
        if CompareRabinHash( RabinHash) > 0 then
          Block ← SubString(fd, BlockSize);
          Sha1Hash ← Sha1(Block);
          if CompareSha1Hash( RabinHash) > 0 then
            MetaStruct ∪ MS(saveoffset, oldblock);
            saveoffset ← fileoffset;
          end
        end
      end
    end
    return MetaStruct;
  end
```

그림 5 DBC_BS 알고리즘

그림 6은 DBC_BS의 동작 원리를 보이고 있다. 기존의 파일은 각 블록에 대해서 라빈 해시 함수와 SHA1 해시 함수의 값이 저장 서버에 보관 되어 있다. 수정된 파일의 블록에 대해서 라빈 해시 함수를 이용하여 해시값을 얻은 후에 저장 서버의 라빈 블록 해시값과 비교한다. 동일한 라빈 해시값(3474)이 존재하는 경우에 라빈 해시값과 매핑되어 있는 SHA1 해시값(을 저장 서버에서 읽어온다. 현재 계산중인 블록의 SHA1 해시값(82a3)을 저장 서버에서 읽어온 SHA1 해시값(82a3)과 비교하여 동일한 값으로 판정이 된 경우에 중복된 블록이 저장 서버에 존재하는 것으로 판단한다.

3.4 개선된 중복 제거 알고리즘

3.4.1 DBC_SS 알고리즘

DBC_BS의 수행시간은 DBC_FS에 비해 8~9배의 시간이 걸리기 때문에 실제 대용량의 파일을 처리하는 경우에 매우 오랜 시간이 소요된다. 그리고 저장하려는 데이터와 저장 서버에 저장된 데이터간의 중복이 없을 경우에 수행시간이 더욱 많이 필요하다. 이와 같은 문제를 해결하기 위하여 본 논문에서는 Duplicate Block Check with Stride Scheme(DBC_SS)을 제안하고 있다. DBC_SS는 파일을 일정한 크기의 블록으로 나눈 후에, 중복 탐지 구간과 건너뛰기 구간(stride)으로 나누어서 중복 블록을 탐색한다. 중복 탐지 구간에서는 DBC_BS 방식으로 중복 블록을 탐지하며 만약 중복 블록이 없는 경우에는 일정한 블록을 건너뛰기하고 DBC_BS 방식으로

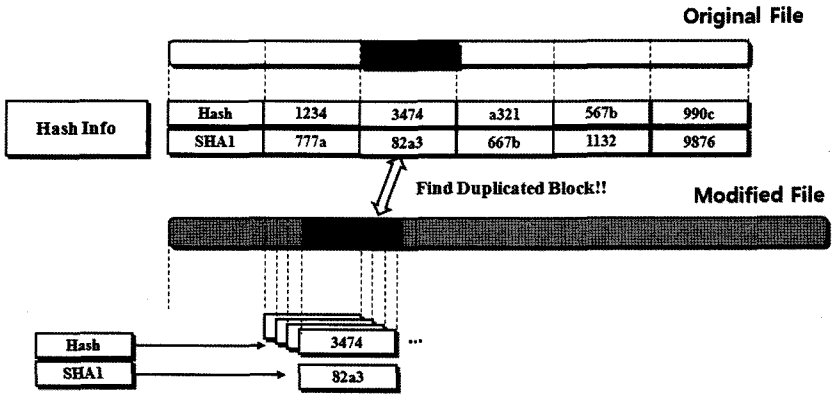

그림 6 DBC_BS 수행 과정

표 2 해시 수행 성능 (AMD athlon 3600+ 1.90GHz)

	Output Len.	1M bytes	50M bytes	100M bytes
SHA1	160 bits	608.7 Mbps	922.7 Mbps	905.9 Mbps
RMD160	160 bits	304.2 Mbps	306.1 Mbps	301.9 Mbps
MD5	128 bits	1258.1 Mbps	1488.9 Mbps	1484.7 Mbps
Rabin	64 bits	180.4 Mbps	189.3 Mbps	198.2 Mbps

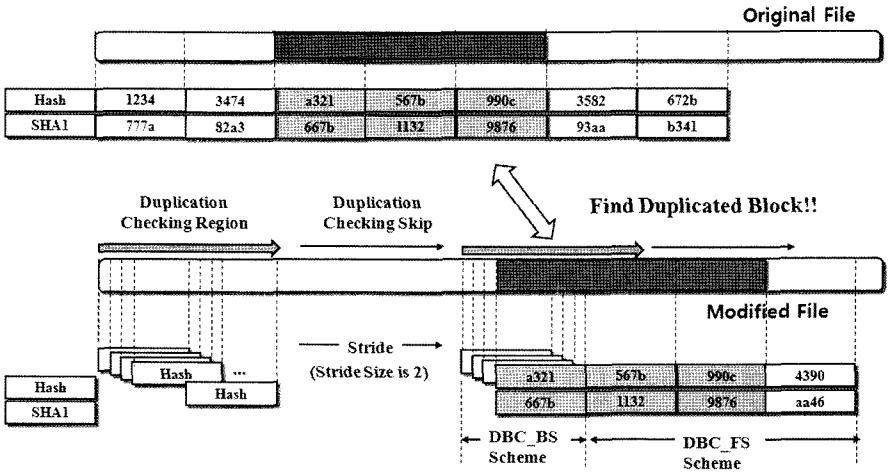

그림 7 DBC_SS 기법 동작 원리

중복을 탐지한다. 일단 중복 블록이 발견되는 경우에는 주변에 연속적으로 중복 블록이 발견될 확률이 높기 때문에 DBC_FS 방식으로 다음 블록의 중복 여부를 탐색한다. 일정 구간의 중복 블록이 발견되고 중복이 아닌 블록이 나타나는 경우에 다시 건너뛰기와 DBC_BS 방식을 반복하여 중복 블록을 찾는 과정을 계속한다.

본 방법은 연속적으로 중복이 있는 블록에 경우 중복 제거에 매우 효과적이며 중복이 없는 파일일 경우에도 DBC_BS 보다 빠른 수행속도를 보장 받을 수 있다. 실험 결과에서 알 수 있듯이 DBC_BS 기법과 비교하여 중복 탐지 시간은 급격히 줄이고 중복 블록은 대부분 찾아내는 결과를 보여주고 있다. 슬라이드의 크기에 따라서 DBC_BS에 비해 중복제거 성능이 줄어들 수가 있지만 시스템에 맞게 슬라이드 크기를 조절한다면 성능을 DBC_BS 만큼 올릴 수 있다.

그림 7은 DBC_SS의 동작을 보이고 있다. 중복 탐지 구간에서 동일한 해시값을 갖는 블록을 발견하지 못하고 2개의 블록을 스트라이드한 후에 DBC_BS 방식으로 해시값 비교를 수행하고 있다. 라빈 해시값(a321) 그리고 SHA1 해시값(667b)에서 중복된 블록을 확인하고 이

```
Algorithm 3: DBC_SS
Input: FileStream fd, StrideSize
Output: MetaStruct
begin
    matchbit leftarrow true;
    saveoffset ← 0;
    fileoffset ← 0;
    fdsize ← Length(fd);
    while fileoffset < fdsize do
        ...;
        if matchbit = ture then
            ...;
            if CompareSha1Hash( Sha1Hash ) > 0 then
                MetaStruct ∪ MS(saveoffset, oldblock);
                saveoffset ← fileoffset;
            end
            matchbit = false;
        end
        ...;
        if fileoffset = ( saveoffset + 2 × BlockSize ) then
            MetaStruct ∪ MS(saveoffset, newblock);
            saveoffset ← saveoffset + BlockSize;
            Block ← SubString(fd, BlockSize);
            MakeNewBlock(fileoffset, BlockSize×StrideSize);
            saveoffset ← fileoffset;
        end
        ...;
    end
    return MetaStruct;
end
```

그림 8 DBC_SS 알고리즘

```
Algorithm 4: DBC_ISS
Input: FileStream fd, StrideSize
Output: MetaStruct
begin
    matchbit ← true;
    saveoffset ← 0;
    fileoffset ← 0;
    fdsize ← Length(fd);
    while fileoffset < fdsize do
        ...;
        if fileoffset = ( saveoffset + 2 × BlockSize ) then
            MetaStruct ∪ MS(saveoffset, newblock);
            saveoffset ← saveoffset + BlockSize;
            Block ← SubString(fd, BlockSize);
            MakeNewBlock(fileoffset, BlockSize×StrideSize);
            saveoffset ← fileoffset;
        end
        ...;
        if CompareRabinHash( RabinHash) > 0 then
            ...;
            if CompareSha1Hash( Sha1Hash ) > 0 then
                MetaStruct ∪ MS(saveoffset, oldblock);
                saveoffset ← fileoffset;
                OldBlockSearch(fileoffset, BlockSize×StrideSize);
                saveoffset ← fileoffset;
                matchbit ← true;
            end
            ...;
        end
    end
    return MetaStruct;
end
```

그림 9 DBC_ISS 알고리즘

후에는 DBC_FS 방식으로 연속된 블록을 추가로 2개 더 찾아내는 것을 보이고 있다. 이 방식에서는 첫 번째 발견된 블록의 이전 블록도 중복된 블록일 확률이 있으나 확인을 하고 있지 않다.

DBC_SS 알고리즘은 다음과 같이 동작한다. 먼저 블록을 해시(Rabin Hash, SHA1)하여 중복 블록이 존재하는지 확인한다. 만약 중복 블록이라면 메타데이터(MetaStruct)에 중복 여부와 블록의 구간을 저장하고 Fixed scheme 방식으로 다음 블록에 대해서 중복을 체크한다. 그러나 중복 블록이 발견되지 않는 경우에는 2 * BlockSize 만큼의 블록을 건너뛰기 한 후에 다음에 나오는 일정 크기의 영역에 대해서 동일한 해시(Rabin Hash, SHA1)가 있는지 검색한다. 반복적으로 위의 동작을 수행하다가 파일 끝을 만나면 함수를 종료하고 메타데이터를 반환한다.

3.4.2 DBC_ISS 알고리즘

Duplicate Block Check with Improved Stride Scheme(DBC_ISS)은 DBC_SS에서 중복 블록을 찾았을 경우에 스트라이드 크기만큼 건너뛴 부분에 대해서 중복 블록이 존재하는지 체크하는 기능이 추가된 알고리즘이다. 본 알고리즘은 알고리즘 수행 도중 중복 블록을 만나는 경우, 현재 블록의 위치에서 뒷부분으로 스트라이드 크기만큼 구간을 재탐색하는 것이다. 그러므로 스트라이드만 했을 때 보다 더 많은 중복을 발견할 가능성이 있으나 재탐색 비용 때문에 수행 시간이 DBC_SS에 비해서 조금 더 늘어나게 된다.

그림 10은 개선된 DBC_SS 기법을 보이고 있다. 앞에서 설명한 DBC_SS에서 3개의 중복 블록을 발견한 이후에, 스트라이드 크기만큼 중복 블록 탐지를 스킵한 부분에 대해서 역으로 DBC_FS 기법을 적용하여 중복 블록을 1개 찾아내는 것을 보이고 있다. DBC_SS와 마찬가지로 중복 블록이 연속적으로 발견될 것으로 예측을 하고 알고리즘을 설계하였다.

4. 중복 제거 시스템 성능 평가

다음 표 3에서는 본 연구의 개발 및 실험 플랫폼을 보이고 있다. 클라이언트 개발은 Window XP에서 수행되도록 하였으며, 중복 제거 서버는 리눅스 기반의 Fedora Core 9 운영체제에서 수행되는 응용 프로그램으로 개발하였다.

실험에 사용된 데이터는 리눅스 배포 데이터(Linux Distribution Data)와 Vmware[20] 이미지 데이터가 사용되었다. 리눅스 배포 데이터는 CentOS 5.2 i386 패키지[21] 데이터를 사용하였으며 CD 이미지 6장, DVD 이미지 1장으로 구성되었으며 총용량은 7,052,120,064 byte이다. CD 이미지와 DVD 이미지의 데이터는 크게 차이가 없다. Vmware 이미지 데이터는 가상 환경 시스템인 Vmware에서 리눅스 환경인 Fedora Core 4와

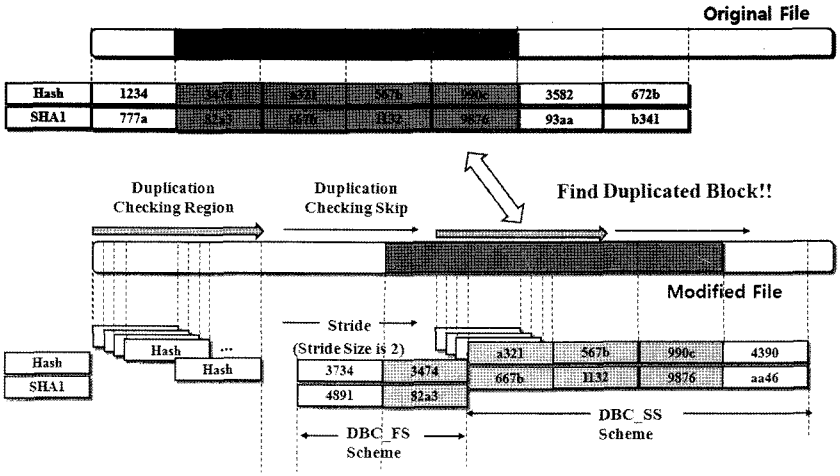

그림 10 DBC_ISS 동작원리

표 3 개발 및 실험 플랫폼

S/W 플랫폼	Client	운영체제	Window XP
		개발도구	Visual Studio 2008
	De-Duplication Server	운영체제	Fedora Core 9
		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	

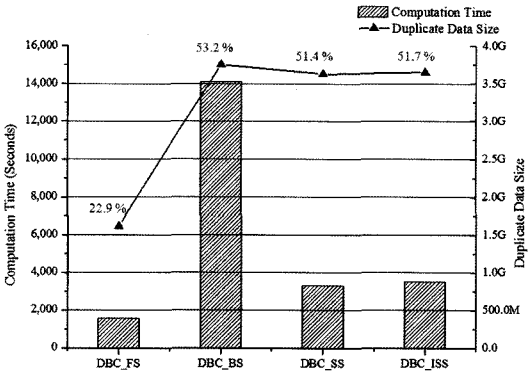

그림 11 리눅스 배포 데이터 중복 제거 결과

Fedora Core 5[22]를 각각 설치하였으며 VMware 운영에 쓰이는 가상 디스크를 실험 데이터로 사용하였다. Fedora Core 4의 전체 설치 크기는 5,235,474,432 byte 이며 Fedora Core 5의 전체 설치 크기는 8,232,566,784 byte를 사용하고 있다.

실험의 내용은 리눅스 배포 데이터를 중복 제거 서버에 저장할 때 DBC_FS, DBC_BS, DBC_SS, DBC_ISS 별로 수행 시간과 중복 제거 성능을 측정하였으며 DBC_SS, DBC_ISS의 기법은 스트라이드 크기에 변화를 주면서 성능을 살펴보았다. 실험을 진행할 때 모든 블록 크기 단위는 8 KByte를 사용하였으며, Vmware 데이터의 실험도 리눅스 배포 데이터 실험과 동일한 기법으로 수행하였다.

그림 12는 리눅스 배포 데이터를 중복제거 서버에 저장하였을 때 중복 제거 알고리즘별로 수행 결과를 출력한 그래프이다. 수행 속도에서는 DBC_FS 기법이 빠른 성능을 보여주고 있으며 중복 제거 부분에서는 DBC_BS 기법이 가장 많은 중복 블록을 탐지하였다. DBC_FS은 다른 기법보다 중복 제거 성능이 절반에도 못 미치는 결과를 보이고 있고 DBC_BS은 DBC_SS나 DBC_ISS의 기법보다 수행시간이 5배 이상이 걸리는 것을 볼 수 있다. 그리고 DBC_SS와 DBC_ISS 기법은 중복 데이터를 제거하는 성능이 DBC_BS와 거의 차이가 나지 않으면서 수행 시간은 DBC_FS에 근접하는 결과를 보여주고 있다. 따라서 리눅스 배포 데이터 같은 데이터를 중복 제거하여 저장할 때에는 DBC_SS, DBC_ISS의 기법을 사용하는 것이 효율적임을 알 수 있다.

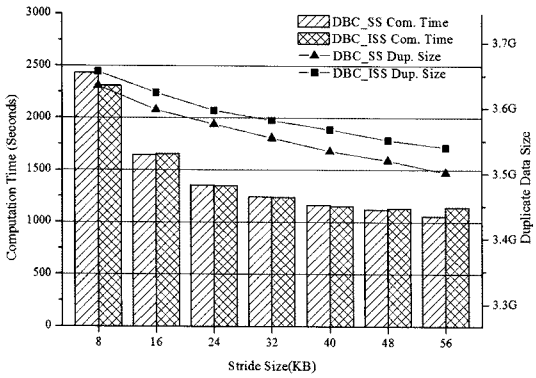

그림 12 스트라이드 크기 변화에 대한 리눅스 배포 데이터 중복 제거 결과

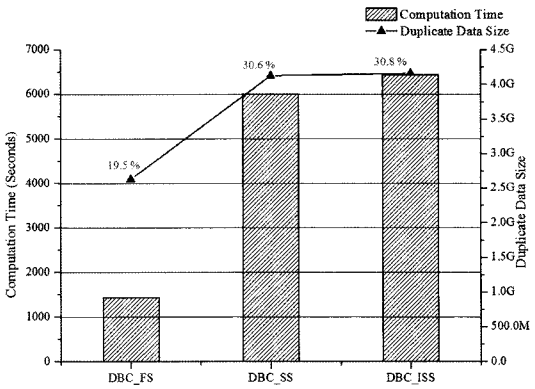

그림 13 알고리즘별 Vmware 데이터 중복제거 결과

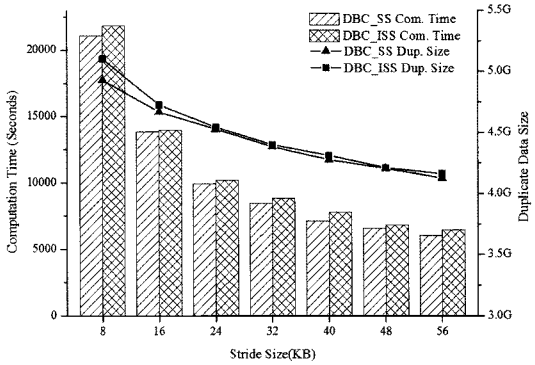

그림 14 스트라이드 크기변화에 대한 Vmware 데이터 중복 제거 결과

그림 12는 DBC_SS, DBC_ISS의 기법에서 스트라이드 크기를 증가시키면서 관찰한 결과이다. 두 기법 모두 스트라이드 크기가 커질수록 중복 제거 성능이 떨어지지만 계산 속도가 향상되는 것을 볼 수 있다. 또한 중복 제거 성능도 급격히 감소하는 것이 아니라 선형적으로 감소하는 것으로 나타나고 있다. 수치적으로 따지면 중복 제거 용량은 스트라이드 크기가 8일 때 와 56일 때 100MB 정도의 차이를 보이고 있으며 DBC_ISS가 DBC_SS 보다 수행시간이 수초정도 더 걸리고 50~70MB 크기만큼 중복을 더 찾아내고 있다.

두 번째 실험은 Vmware 데이터를 저장했을 때 수행 속도와 중복제거 성능을 관찰한 것이다. 블록 크기는 첫 번째 실험과 마찬 가지로 8Kbyte를 사용하였으며 DBC_SS, DBC_ISS 알고리즘에서의 스트라이드 크기를 56으로 하여 실험한 것이다. 그림 13에서 DBC_BS이 제외된 이유는 수행시간의 차이(수치적으로 10배 가까운 성능차이가 났음)가 크게 나타나므로 비교가 불가능했기 때문이다. 이 실험에서는 DBC_FS가 중복 데이터를 2.6Gbyte 정도 찾아냈으며 수행 시간은 1400초 정도 걸렸으며, DBC_SS와 DBC_ISS는 4.1Gbyte 정도의 중복을 찾아내는데 6000초 정도 시간이 소요되었다.

그림 14에서는 스트라이드 크기 변화에 따른 중복 제거 결과를 보이고 있다. Vmware 데이터에서의 수행시간은 리눅스 배포 데이터에서 수행했을 때 보다 비율적으로 많이 증가하였는데 중복 데이터의 비율이 높지 않기 때문에 그만큼 탐색 시간이 길어졌다. 또한 스트라이드의 크기가 커질수록 수행시간이 줄어드는 폭이 리눅스 배포 데이터를 저장했을 때 보다 커졌으며 DBC_SS, DBC_ISS의 중복제거 성능의 차이가 줄어들었다. 중복 제거 용량은 스트라이드 크기가 8일 때와 56일 때의 차이가 800MB정도 나는 것을 볼 수 있어 스트라이드 크기가 커질수록 중복 데이터 제거 성능이 떨어짐을 확인할 수 있었다.

5. 결론 및 향후연구

본 논문에서는 해시를 사용하여 데이터 중복을 찾는 기법에 대한 기존 연구의 한계를 분석하고 다양한 파일 서버에서 범용적으로 사용 가능한 개선된 중복제거 알고리즘을 제시하였다. 기존의 접근 방법에서 가변 분할 기법에 해당되는 CDC, 델타 인코딩 방식은 수행 시간이나 중복제거에서 좋은 성능을 나타내지만 시간적 전후 관계에 있는 경우에만 사용할 수 있기 때문에 일반적인 아카이브 저장에 적용하기 어렵다. 또한 고정 분할 비교 방식을 사용하는 Venti는 수행 속도는 빠르지만 중복제거 성능이 다른 방식보다 떨어진다는 한계가 있다.

본 논문에서는 고정 분할 기법에 스트라이드 방식을 적용한 DBC_SS 기법으로 수행속도의 성능과 중복 탐지 기능을 향상시킬 수 있었다. 또한 DBC_SS 기법이 스트라이드 구간에 대하여 중복 블록을 놓치는 문제점을 보완하는 DBC_ISS 기법을 제안하여 더 나은 성능 향상을 보여주고 있다. 실험을 통하여 고정 분할 기법의

기본적인 알고리즘인 DBC_FS, DBC_BS와 본 논문에서 제시하는 DBC_SS, DBC_ISS에 대한 성능 비교를 수행하였으며, 처리 속도 및 중복 탐지 부분에서 의미 있는 결과를 도출한 것을 확인하였다.

향후 연구로는 제안된 기술의 성능을 개선하기 위해 다양한 압축 기법과 해싱 방식을 혼합하는 방법을 연구할 계획이며, 다양한 형태의 파일 유형에 적합한 중복 제거 알고리즘을 세분화할 계획이다.

참 고 문 헌

[1] J.S. Robin and C.E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In Proceedings of the 9th USENIX Security Symposium, Denver, CO, August 2000.

[2] 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.

[3] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, ACM SIGCOMM 2001, San Deigo, CA, August 2001, pp.149-160.

[4] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. 5th USENIX OSDI*, Boston, MA, Dec. 2002.

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

[8] plan9 home page, http://plan9.bell-labs.com/plan9/

[9] 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.

[10] Athicha Muthitacharoen, Benjie Chen, and David Mazieres, "A Low-Bandwidth Network File System," In *Proceedings of the Symposium on Operating Systems Principles (SOSP'01)*, pp.174-187, 2001.

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

[12] Constantine P. Sapuntzakis, Ramesh Chandra, BenPfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[13] D. Bobbarjung, Suresh Jagannathan, C. Dubnicki. Improving Duplicate Elimination in Storage Systems, ACM Transactions on Storage, November 2006.

[14] K. Eshghi and H.K. Tang, A Framework for Analyzing and Improving Content-Based Chunking Algorithms. Hewlett-Packard Labs Technical Report TR 2005-30.

[15] Fred Douglis and Arun Iyengar. Application-specific Delta-encoding via Resemblance Detection. In *Proceedings of 2003 USENIX Technical Conference*, pp.113-126, San Antonio, Texas, USA, 2003.

[16] Purushottam Kulkarni, Fred Douglis, Jason La Voie, and John M. Tracey, "Redundancy Elimination Within Large Collections of Files," In *Proceedings of 2004 USENIX Technical Conference*, Boston, Massachusetts, USA, 2004.

[17] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST)*, pp.269-282, 2008.

[18] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Campbell, "Sparse Indexing, Large Scale, Inline Deduplication Using Sampling and Locality," In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST) 2009*, San Francisco, CA.

[19] Jim Gray, Catharine van Ingen, "Empirical Measurements of Disk Failure Rates and Error Rates," Microsoft Research Technical Report MSR-TR-2005-166, 2005.

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

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

[22] fedoraproject home page, http://www.fedoraproject.org/

정 호 민

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

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