분산 파일시스템을 위한 효율적인 협력캐쉬 알고리즘

Efficient Cooperative Caching Algorithm for Distributed File Systems

박새미, 이석재, 유재수
충북대학교 정보통신공학과

Park Sae-Mi, Lee Seok-Jae, Yoo Jae-Soo
Dept. of Computer and Communication Eng.,
Chungbuk National University

요약

분산 파일시스템 환경에서는 디스크 접근 비용을 줄이기 위해 각 노드에 캐쉬된 데이터를 서로 공유하는 협력캐쉬 기법이 사용된다. 협력캐쉬 기법은 분산되어있는 시스템들의 캐쉬정보를 서로 공유하여 가상으로 더 큰 캐쉬를 형성함으로써 캐쉬 히트율을 높이고 디스크 접근을 줄이는 방법이다. 기존에 제안된 협력캐쉬 기법들은 캐쉬에 대한 근사정보를 이용하여 메시지 비용을 줄이고, 로컬캐쉬영역과 글로벌 캐쉬영역을 가변적으로 사용하여 캐쉬히트율을 높이고 있다. 또한 캐시교체시 교체된 블록을 비활동적인 노드로 보내어 계속 캐쉬에 유지하도록 하여 글로벌 캐시히트율을 높이는 장점을 갖는다. 그러나 잘못된 근사정보가 성능을 저하시킬 수 있으며 일관성 유지를 위한 메시지교환 비용이 많이 든다는 단점을 갖고 있다. 또한 비활동적인 노드를 선정하기 위해 사용되는 각 노드의 에이지정보 관리비용이 많이드는 단점을 갖고 있다. 본 논문에서는 정확한 캐쉬정보를 유지하며 일관성 유지비용과 캐시에이지정보 관리비용을 최소화시키는 협력캐쉬 알고리즘을 제안한다. 그리고 성능평가를 통해 기존의 협력캐쉬 기법과 비교하여 제안하는 알고리즘의 우수성을 보인다.

Abstract

In distributed file-systems, cooperative caching algorithm which owns the data cached at each node jointly is used to reduce an expense of disk access. Cooperative caching algorithm is the method that increases a cache hit-ratio and decrease a disk access as it holds the cache information of distributed systems in common and makes cache larger virtually. Recently, several cooperative caching algorithms decrease the message costs by using approximate information of the cache and increase the cache hit-ratio by using local and global cache fields dynamically. And they have an advantage that increases the whole field hit-ratio by sending a replaced block to the idel node on cache replacement in order to maintain the replaced block in the cache field. However the wrong approximate information deteriorates the performance, the concistency maintenance goes to great expense to exchange messeges and the cost that manages Age-information of each node to choose the idle node increases. In this thesis, we propose a cooperative cache algorithm that maintains correct cache information, minimizes the maintance cost for consistency and the management cost for cache Age-information. Also, we show the superiority of our algorithm through the performance evaluation.

1. 서론

최근 개인용 컴퓨터의 활발한 보급과 네트워크 컴퓨팅 기술의 발전으로 컴퓨터 사용자수가 급증하면서 서로 다른 컴퓨터 사용자끼리의 데이터 공유가 반드시 필요하게 되었다.[1] 보통 컴퓨터 사용자의 데이터는 파일이라고 불리는 하나의 추상화된 단위로 구별되어 처리되어지며, 파일시스템은 이러한 파일을 컴퓨터상에서

보관하는 방법을 제공하는 운영체제의 일부분으로 보통 하나의 컴퓨터에는 하나의 파일시스템 형태의 로컬 파일시스템이 존재한다. 그런데 로컬 파일시스템으로는 사용자끼리의 정보 공유가 디스켓을 이용한 직접 복사나 네트워크를 이용한 파일 전송에 의해서만 가능하기 때문에 당연히 사용자의 불편과 생산성의 저하를 초래하게 된다. 분산 파일시스템은 이러한 로컬 파일시스템의 문제를 해결하고자 하는 노력의 결과라 할 수 있겠다. 즉 분산 파일시스템이란, 물리적으로 서로 다른 컴퓨터끼리 네트워크로 연결하여, 사용자에게 동일하게 보이는 파일 접근 공간을 제공해 주는 시스템으로 서로 다른 컴퓨터를 사용하는 많은 사용자들에게 네트워크를 통하여 공통된 파일시스템을 제공해 주는 시스템이다.

이러한 분산 파일시스템 환경에서는 데이터를 보다 효과적으로 관리하기 위해서 캐시라는 파일시스템의 핵심적인 기술을 사용한다. 이 기술은 자주 접근되는 데이타를 원격 데이타 서버로부터 프로그램이 수행되는 로컬 사이트로 캐시하는 것으로 같은 정보에 대한 반복된 접근은 부가적인 네트워크의 전송 없이도 로컬사이트에서 처리될 수 있다.

단일캐시 알고리즘의 경우, 로컬 사이트로 캐시된 데이터를 검색할 때 다른 사이트와의 정보교환 없이 로컬 사이트에서만 검색하는 알고리즘으로 제한된 캐시의 크기로 인해 페이지 교체정책을 이용하여 캐시 데이터를 변경시켜 사용한다. 따라서 단일캐시 알고리즘은 캐시 크기의 제한으로 자주 디스크에 접근해야하는 단점이 발생한다. 이에 반해, 협력캐시 알고리즘은 분산되어 있는 시스템들의 캐시간에 캐시정보를 상호협력을 통해 교환함으로써 원하는 데이터를 로컬사이트뿐 아니라 다른 사이트로부터 검색하여 가져올 수 있으므로 단일캐시 알고리즘보다 훨씬 좋은 성능을 보인다.[2]

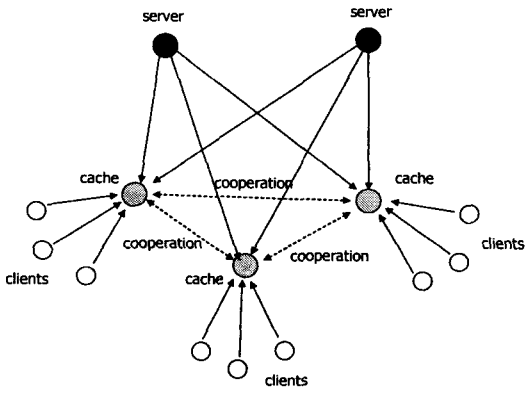
전통적인 분산 파일시스템에서는 이러한 협력캐시 알고리즘 사용시에 모든 캐시 정보를 중앙 서버가 제공해 주고 있는데 이러한 형태의 분산 파일시스템의 경우 서버에 병목현상을 초래할 수 있다는 단점을 가진다.

그러므로 본 논문에서는 서버의 과부하를 막기 위해 중앙에서 관리하는 서버 없이 전체 시스템의 캐시 효율성을 향상시킬 수 있는 협력캐시 알고리즘을 제안한다. 또한 캐시를 분산 관리하게 되면 캐시에 대한 정보를 관리하기 위한 비용도 고려해야 하는데 캐시관리 비용을 최소화하는 알고리즘을 제안한다.

본 논문의 구성은 다음과 같다. 우선 2장에서는 기존에 제안된 분산 파일시스템에서의 협력캐시 알고리즘에 대해서 기술하고, 3장에서 본 논문에서 제안하는 협력캐시 알고리즘에 대해 설명한다. 그리고 4장에서 결론을 맺고 향후 연구방향을 제시한다.

Ⅱ. 관련 연구

협력캐시 알고리즘은 그림 2.1과 같이 분산되어있는 시스템들의 캐시간에 캐시정보를 상호협력(cooperation)을 통해 교환함으로써 다른 클라이언트의 메모리들을 이용하여 가상으로(virtual) 더 큰 캐시를 형성한다[3]. 따라서 글로벌 캐시히트율이 증가하여 디스크를 좀더 적게 접근하므로 로컬 캐시영역으로만 데이터를 캐시하여 자주 디스크에 접근하는 단일캐시 알고리즘에 비하여 네트워크 통신 속도가 훨씬 빠르다. 또한 서버가 데이터를 전송하는 대신에 데이터가 있는 클라이언트를 알려주므로 서버에 부하가 적게 걸리며, 비용면에서 비교할 때 100개의 클라이언트에 값싼 16MB씩 가지는 것은 1개의 서버가 비싼 1600MB를 가진 것보다 훨씬 효과적이다.



▶▶ 그림 2.1 일반적인 협력캐시 알고리즘의 구조

이 장에서는 기존에 제안된 분산 파일시스템에서의 협력캐시 알고리즘인 Hint-based 알고리즘[4], GMS

(Global Memory Service) 알고리즘[5] 그리고 Serverless Network FileSystem[6]에 대하여 살펴보고 본 논문의 접근 방향에 대해 기술한다.

1. Hint-based 협력캐시 알고리즘

Hint-based 협력캐시 알고리즘은 캐시 정보에 대해서 중앙에서 관리하는 서버 없이 캐시를 분산 관리하는 알고리즘으로 캐시에 대한 정확한 정보를 이용하기보다 이에 따르는 메시지 통신 비용을 줄이기 위한 목적으로 캐시에 대한 근사(approximate) 정보인 힌트(hint)정보만을 가지고 처리하는 알고리즘이다.

블록 검색(Block Lookup)시 클라이언트가 파일 관리자로부터 파일에 대한 토큰을 받으면서 그 파일에 대한 모든 블록의 힌트정보를 함께 받아 그 정보를 이용하여 요청된 블록을 검색한다. 비록 잘못된 힌트정보로 인하여 부분적으로 캐시히트율이 크게 감소될 수 있으나 성능평가 결과 전체적인 캐시히트율을 비교했을 때 정확한(accurate) 캐시정보를 위하여 관리자와 메시지를 교환하는 경우가 오히려 성능이 더 떨어지는 것으로 나타났다. 교체정책(Replcement)으로는 항상 서버로부터 가장 처음 복사된 블록(master copy)만을 전송시키는 방법인 Best-Guess 교체정책을 사용한다. 그리고 각 노드마다 가장 오래된 블록에 관한 정보를 저장하고 있는 리스트(Oldest Block List)를 유지함으로써 교체시에 이 리스트 정보를 이용하여 가장 오래된 블록을 버린다. 이 교체정책은 중앙에서 관리하는 관리자를 두지 않음으로써 블록에 대한 정보를 교환하기 위한 부가적인 메시지를 줄이고 있으므로 기존의 알고리즘보다 높은 성능을 보이고 있다.

Hint-based 협력캐시 알고리즘은 기존 알고리즘에서 발생했던 부가적인 메시지를 효과적으로 줄이고 있으나 각 노드가 블록 정보에 대한 리스트를 유지하고 관리하기 위한 비용이 많이 든다는 단점을 가지고 있다.

2. GMS 알고리즘(Global Memory Service)

GMS(Global Memory Service)는 캐시를 관리하는 방법에 있어서 Hint-based 협력캐시 알고리즘과 같이 중앙에서 관리하는 서버를 따로 두지 않고 캐시를 분산 관리한다. 또한 캐시를 효율적으로 이용하기 위하여 로컬캐시영역과 글로벌 캐시영역을 따로 구분하여 사용하지 않고 가변적으로 이용함으로써 캐시히트율을 높이는 특징을 가지고 있다.

블록 검색을 위해서 여러 가지 데이터구조가 사용되는데 기본적으로 노드가 가지고 있는 페이지에 대한 자료구조인 UID(Unique Identifier), 캐시된 블록을 저장하고 있는 노드의 위치 자료구조를 해쉬 테이블 형태로 만든 GCD(Global Cache Directory)를 사용한다. 그리고 GCD를 저장하고 있는 노드와 UID를 매핑시켜 주는 자료구조인 POD(Page Ownership Directory)를 모든 노드에 똑같이 복사시켜서 가지고 있다.

GMS는 캐시된 블록에 대해 한번 사용되고 나서 다시 사용하기까지의 시간간격을 에이지(age) 정보라고 정의하고 그 정보를 이용하여 전체 캐시에서 가장 자주 사용되는 블록을 우선적으로 유지하는 교체정책을 사용한다. 따라서 기존의 알고리즘에 비하여 전체적인 시스템 성능은 향상시키고 있으나, 데이터의 가치판단을 위한 정보관리 및 데이터관리 비용이 상당히 높다는 단점을 가진다.

3. Serverless Network FileSystem

Serverless Network FileSystem은 여러 개의 관리자를 두어 모든 자원들을 분산시켜 관리하는 네트워크 파일시스템으로서, 저장서버들을 다시 그룹화하여 그룹내의 서버들에 데이터를 분산시켜 저장하고 관리함으로써 중앙 병목 현상을 제거한다.

세 가지 주요 구성 요소에는 클라이언트(client)와 관리자(manager) 그리고 저장서버(storage server)가 있다. 클라이언트는 사용자들로부터 파일에 대한 요구를 받고 데이터를 보내주며 페이지 폴트(page fault)가 발생했을 경우, 관리자에게 원하는 데이터 정보를 얻어 저장서버들이나 다른 클라이언트들로부터 응답을 받는다. 관리자의 역할은 파일 데이터의 위치를 검색하여 클라이언트로의 요구를 적절한 목적지로 보내는 것이며, 저장서버들은 공동으로 데이터를 그룹단위로 분산시켜 저장하는 역할을 한다.

Serverless Network FileSystem은 데이터를 읽고

쓰는 동작시에 다수의 디스크들에 나누어 동시에 처리하므로 입출력 성능을 높이고 있다. 그리고 여러 노드에 분산 저장할 때 블록에 대한 패리티 블록을 계산하여 하나의 디스크에 저장함으로써 특정 노드의 결함이 발생하여도 나머지 노드들에 저장된 내용을 가지고 결함이 발생한 노드의 디스크에 저장된 정보를 복구할 수 있어 신뢰성을 높일 수 있다.

그러나 정확한 캐시정보 관리에 있어 비용이 많이 들며, 캐시 교체정책에 있어서는 특별하게 고려하지 않고 있어 다른 알고리즘에 비해 특별히 높은 성능을 보이고 있지 않다.

Ⅲ. 협력캐시 알고리즘 설계

본 논문에서는 중앙에서 관리하는 서버 없이 캐시를 분산 관리하며 전역 캐시히트율과 로컬캐시히트율을 동시에 만족시키는 효율적인 캐시 알고리즘을 설계한다. 중앙에서 관리하는 서버 없이 캐시를 분산 관리하게 되면 캐시에 대한 정보를 관리하기 위한 비용이 많이 들고 블록접근시간도 지연되지만, 중앙서버가 캐시를 관리함으로써 생기게 되는 서버의 병목현상이 발생하지 않는다.

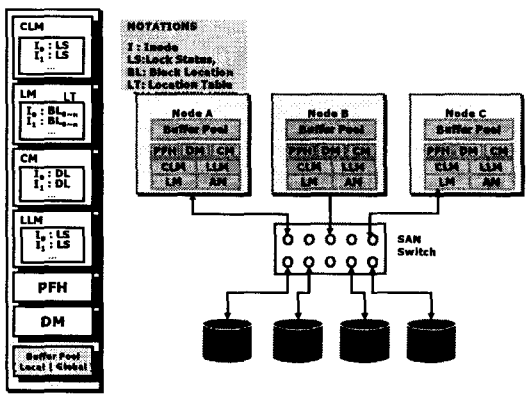
이에 본 논문에서는 서버 없이 캐시를 분산관리하여 서버의 과부하를 막고 데이터 관리비용을 최소화시키는 방법에 초점을 맞추어 설계한다. 또한 효율적인 글로벌 캐시히트율과 로컬 캐시히트율을 위하여 글로벌 캐시영역과 로컬 캐시영역을 따로 구분하여 사용하지 않고 동적으로 유연하게 이용함으로써 캐시히트율을 높인다. 그리고 글로벌 캐시히트율을 향상시키기 위하여 한 노드에서 교체된 캐시를 모든 노드중 가장 비활동적인(idle) 노드로 전송하는 방법으로 설계하며, 비활동적인 노드의 선정방법은 전체 알고리즘의 성능에 큰 영향을 주므로 메시지 교환을 최소로 하며 CPU 점유가 적은 노드 선정방법을 제안한다.

본 장의 구성은 먼저 전체 시스템의 구성도에 대해서 기술하고 논문에서 제시하는 캐시의 메타정보 관리방법에 대해서 기술한다. 그리고 노드가 블록에 접근하기 위해 블록의 위치정보를 검색하게 되는데 이와 관련된 로컬 히트(local hit)처리방법과 페이지 폴트(page fault) 처리방법에 대해서 기술하고, 캐시히트율을 최대화하기 위한 캐시의 교체정책에 대해서 기술한다. 마지막으로 임의의 노드에서 발생한 블록접근 요청에 대해서 항상 최신의 데이터를 제공하기 위해 필요한 캐시의 일관성 유지에 대한 설계 내용을 기술한다.

1. 전체 구성도

그림 3.1은 본 논문에서 제안하는 알고리즘의 전체구성도이다. 그림 3.1과 같이 여러 개의 노드가 SAN[7]에 부착된 저장장치를 공유한다. SAN은 서버에 개별적으로 연결되던 저장 시스템을 광 채널과 같은 고속의 전용 네트워크에 직접 연결하여 중앙 집중적인 저장 시스템의 관리를 가능하게 하고, 서버를 거치지 않고 네트워크에 연결된 저장 장치를 직접 접근하는 환경으로 좀더 효과적으로 저장 장치에 저장된 데이터를 접근하여 변경 및 읽기를 수행한다.

그림 3.1의 가장 왼쪽의 구성도는 하나의 노드에 대한 구성도로써 각 노드는 전체 노드의 잠금(Lock)을 관리하고 있는 CLM(Cluster Lock Manager)과 자신 노드의 잠금을 관리하는 LLM(Local Lock Manager),



▶▶ 그림 3.1 전체 구성도

그리고 제안하는 알고리즘의 주요소인 LM(Location Manager), CM(Consistency Manager), DM (Delibery Manager), PFH(Page Fault Handler), AM(Age Manager)으로 구성된다.

LM(Location Manager)은 각 노드에 위치하여 노드

에 할당된 블록들의 위치정보를 관리하는 관리자이다. 노드는 블록요청시나 블록정보를 강제로 무효화 시켜야 할때(invalidate), 최신블록 요청시, 혹은 블록의 중복 체크시에 블록의 위치를 검색하기 위해서 LM에 질의를 내린다. CM(Consistency Manager)은 캐시의 일관성을 유지하는 일을 수행하는 관리자이다. 제안하는 알고리즘에서는 블록 변경시에 내용을 곧바로 변경시키지 않고 변경된 블록을 접근하게 되었을 때 이를 변경시키는 방법인 LWI(Lazy Write Invalidate) 알고리즘을 사용한다. DM(Delibery Manager)은 노드의 블록을 요청한 노드에게 전송시키는 작업을 수행한다. PFH (Page Fault Handler)는 페이지 폴트가 일어났을 때, 즉 노드가 원하는 블록을 로컬 캐시영역에서 검색하는데 실패하게 될 경우 이를 해결하는 모든 절차를 수행해주는 관리자이다. 페이지 폴트시에 원하는 블록의 위치를 검색하기 위해 LM에게 질의를 내리고 위치정보를 받은 후에 DM에게 원하는 블록을 전송하도록 요청하는 일을 한다. AM(Age Manager)는 가장 비활동적인 노드를 선택하기 위한 정보를 제공하는 관리자이다. 제안하는 알고리즘에서는 캐시 교체시에 가장 비활동적인 노드 즉, 오랫동안 참조되지 않은 블록의 수가 많은 노드를 찾아서 교체하게 될 블록를 전송하는데 이에 대한 정보를 관리한다.

마지막으로 각 노드의 캐시는 글로벌 캐시와 로컬 캐시를 구분하여 사용하는데 정적으로 할당된 것이 아니라 가변적으로 이용한다. 로컬 캐시는 노드가 사용중이거나 사용한 블록들을 의미하며, 글로벌 캐시는 캐시 히트율을 높이기 위해 캐시 교체시에 다른 노드에 의하여 전송된 블록들을 의미한다.

2. 캐시의 메타정보 관리

제안하는 알고리즘에서는 블록의 정보들을 기술하는 메타정보를 관리한다. 메타정보에는 블록의 현재위치, 블록의 변경 여부, 블록의 중복 여부, 블록의 최신본 여부의 정보들이 있다. 그리고 이 메타정보는 LM에 의해서 파일단위로 관리가 되며 LM은 각 파일이 담당하는 블록들의 정보들을 위치 테이블(Location Table)을 이용하여 관리한다. 또한 모든 파일은 라운드 로빈(round robin)을 통해서 각 노드에 분산시켜 관리된다.

2.1 위치 테이블 (Location Table)

제안하는 알고리즘에서는 캐시의 메타정보를 위치 테이블을 이용하여 관리한다. 위치 테이블의 구조는 각 파일에 대한 정보를 기억하기 위해 만들어진 inode번호와 inode에 따른 블록 번호 그리고 그 블록들의 위치정보들로 구성된다. 블록들의 위치정보는 각 노드마다 2비트씩 할당하여 처음 비트는 해당 블록이 있는지 여부를 나타내고, 두 번째 비트는 해당 블록이 최신인지를 나타낸다.

그림 3.2는 세 개의 노드에 대한 블록들의 위치테이블이다. 예를 들어 블록42를 소유하고 있는 노드는 블록42에 대해서 노드의 처음 비트가 1로 세팅된 노드로써 노드N1, 노드N2, 노드N3가 되겠다. 그리고 두 번째 비트가 1로 세팅된 노드N1만이 최신의 블록을 가지고 있는 노드라 할 수 있다.

Inode 번호 / 블록 번호 / 블록 소유 노드 / 최신 블록 여부

Inode Nos	Block IDs	Location					
		N1		N2		N3	
2	12	1	0	1	1	1	1
	13	0	0	0	0	1	1
	14	1	1	1	1	0	0
5	32	1	0	1	1	0	0
	33	0	0	1	1	0	0
	40	1	1	0	0	0	0
	42	1	1	1	0	1	0

▶▶ 그림 3.2 블록의 위치 테이블 (Location Table)]

2.2 수정 리스트 (Dirty List)

쓰기 잠금(write lock)을 가진 노드가 블록에 새로운 정보를 써서 블록의 메타정보가 수정되어야 할 경우, 제안하는 알고리즘에서는 수정된 정보를 바로 위치 테이블에 반영시키지 않고 우선 수정 리스트에 정보를 유지한다. 블록의 정보가 변경될 때마다 모든 복사본들을 모두 변경시키기 위해서는 많은 메시지 전송이 필요하다. 수정 리스트는 이러한 메시지 전송을 줄이기 위한 것으로 수정 리스트의 구조는 간단하게 1바이트로 구성되며 변경된 블록만을 저장한다. 따라서 블록 검색시에 원하는 블록을 찾은 후에 항상 수정 리스트를 참조하여 블

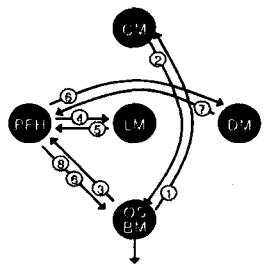
록의 상태 정보를 확인한다.

3. 페이지 검색

각 노드는 원하는 블록에 접근하기 위하여 먼저 로컬 캐시영역을 검색한다. 이때 블록 검색이 로컬 캐시영역에서 성공적으로 완료되면 로컬 히트되었다고 하고, 블록 검색이 로컬 캐시영역에서 실패하면 페이지 폴트가 발생되었다고 한다. 페이지 폴트시 다른 노드의 캐시영역이나 디스크를 검색하여 원하는 블록에 접근한다.

3.1 로컬 히트(local hit)처리

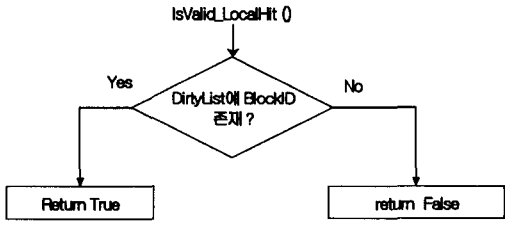
노드가 원하는 블록을 로컬 캐시영역에서 검색하게 되어 로컬 히트가 되었을 경우, 그림 3.3과 같이 로컬 히트를 처리한다.



- 1 : hit 된 캐시의 validity를 local의 CM에 확인
- 2 : valid 또는 invalid(다른 노드에서 변경) 통보
  invalid일 경우 page fault와 동일한 처리 진행
- 3 : invalid 경우, PFH에 page fault 처리 시작 지시
- 4 : 블록 위치 질의
- 5 : 블록 위치 응답(nodeID or NULL)
- red 6 : 블록 전송 요청
- blue 6 : disk I/O 수행 지시
- 7 : 블록 전송
- 8 : 블록을 캐시영역에 위치

▶▶ 그림 3.3 로컬 히트처리

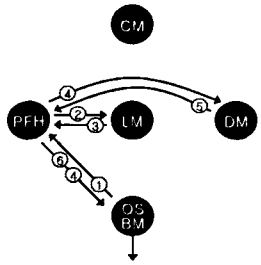
우선 검색된 블록의 수정여부를 확인하기 위한 유효성 검증을 위해 노드의 CM에게 질의를 내리고 CM은 그림 3.4와 같이 블록의 유효성을 검증한다. 수정리스트에 그 블록이 있는지를 확인하여 수정 리스트에 없는 경우 블록이 쓰기 연산에 의하여 변경된 것이 아님을 알고 유효하다고 판단하고, 반대로 수정리스트에 블록 ID가 있는 경우 블록이 변경된 것으로 판단하고 다시 최신의 블록을 찾기 위해서 페이지폴트 처리과정을 수행한다.



▶▶ 그림 3.4 로컬 히트시 캐시의 유효성 검증 알고리즘

3.2 페이지 폴트(page fault) 처리

노드가 원하는 블록을 로컬 캐시영역에서 검색하는데 실패한 경우, 즉 페이지 폴트가 발생하게 되면 PFH는 그림 3.5와 같이 처리한다.



- 1 : page fault 처리 지시
- 2 : 블록 위치 검색
- 3 : 블록 위치 응답 (NodeID or NULL)
- blue 4 : 3의 응답이 NULL, disk I/O 지시
- read 4 : 블록 전송 요청
- 5 : 블록 전송
- 6 : DM에 해당 블록이 없으면 blue 4 와 동일
  그렇지 않으면 블록을 캐시영역에 위치

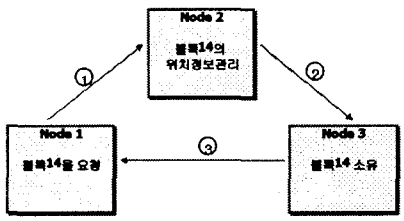
▶▶ 그림 3.5 페이지 폴트 처리

PFH는 우선 블록의 위치를 검색하기 위해 LM에게 질의를 내린다. LM은 블록 위치를 검색하여 응답을 하는데 블록이 다른 노드의 캐시영역에 존재하지 않는 경우에 디스크로부터 블록의 전송을 지시함으로써 페이지 폴트 처리를 수행한다. 하지만 블록이 다른 노드의 캐시영역에 존재하는 경우, 블록의 위치정보를 PFH에게 응답하고 PFH는 다시 블록의 위치정보를 가지고 DM에게 블록 복사본의 전송을 요청한다. 그리고 DM는 LM로부터 받은 블록의 위치정보를 가지고, 블록이 위치한 노드의 캐시영역으로부터 블록 복사본을 전송한다. 이

때 블록이 그 노드의 캐시영역에 있지 않아서 전송하지 못하는 경우가 발생할 수 있다. 이것은 PFH가 블록의 위치정보를 받고나서 블록을 요청하는 사이에 블록의 위치가 교체정책에 의해 변경되었다는 것을 의미한다. 따라서 이럴 경우에 블록을 가져올 수 없으므로 디스크로부터 원하는 블록을 요청함으로써 페이지 폴트처리를 완료한다.

3.3 블록 연산과 블록의 위치정보 유지

제안하는 알고리즘에서는 쓰기와 읽기의 연산으로 변경된 블록의 위치정보에 대해서 추가적인 메시지 없이 위치정보를 유지시키는 방법을 사용한다. 그림 3.6은 노드N1이 블록14를 요청한 경우의 처리 흐름도이다. 노드N1에서 페이지 폴트가 발생했을 경우, 우선 원하는 블록의 위치정보를 알아내기 위해 그 블록을 관리하는 노드N2에게 질의를 내린다. 노드N2는 블록을 가지고 있는 노드N3를 검색하고 이 정보를 노드N1에게 알려준다. 이때 나중에 따로 메시지를 보내서 블록에 대한 위치정보를 변경시키는 것이 아니라, 노드N1이 블록을 검색하고 있다는 것은 결국 그 블록을 찾아서 블록의 복사본을 노드N1으로 가져오는 것을 의미하기 때문에 검색과 동시에 블록의 위치정보를 노드N1에 중복으로 변경시켜준다. 그리고 블록전송이 실패하는 경우 노드N1는 노드N2에 전송실패를 알리고, 노드N2는 블록의 위치를 재조정하여 작업을 완료한다.



▶▶ 그림 3.6 노드의 블록 요청

그림 3.7은 노드N1이 블록14를 읽기 연산을 수행하기 전과 읽기 연산을 수행한 후의 LT를 나타낸다. 읽기 잠금을 가진 노드가 블록을 검색하는 경우, 블록이 있는 파일에 대한 잠금을 얻어온 후 위치 테이블에서 읽기작업을 수행시킬 블록의 위치정보를 검색한다. 그리고 검색과 동시에 위치 테이블에 요청한 노드가 블록을 가지고 있는 것으로 블록상태를 변경시킨다. 즉 해당 블록이 있는지 여부를 나타내는 처음비트와, 해당 블록이 최신인지를 나타내는 두 번째 비트를 모두 1로 바꾼다.

□ 블록 요청 수행전의 위치테이블

Inode 번호 블록 번호 블록 소유 노드 /최신 블록 여부

Inode Nos	Block IDs	Location					
		N1		N2		N3	
2	12	1	0	1	1	1	1
	13	0	0	1	1	1	1
	14	0	0	0	0	1	1

□ 블록 요청 수행후의 위치테이블

Inode 번호 블록 번호 블록 소유 노드 /최신 블록 여부

Inode Nos	Block IDs	Location					
		N1		N2		N3	
2	12	1	0	1	1	1	1
	13	0	0	0	0	1	1
	14	1	1	0	0	1	1

▶▶ 그림 3.7 읽기 연산과 위치테이블 변경

그림 3.8은 노드 N1이 블록 14를 쓰기 연산을 수행하기 전과 쓰기 연산을 수행한 후의 LT를 나타낸다. 쓰기 잠금을 가진 노드가 블록을 변경시키게 되는 경우에는 읽기 작업과 동일하게 우선 위치 테이블에서 쓰기 작업을 수행시킬 블록의 위치정보를 검색한다. 그리고 쓰기 작업 후에는 그 블록정보만 변경되므로 요청한 노드만이 최신의 블록을 가지고 있는 것으로 변경시킨다. 즉, 해당 블록이 있는지 여부를 나타내는 처음비트와, 해당 블록이 최신인지를 나타내는 두 번째 비트를 모두 1로 바꾸고, 블록을 가진 나머지 노드들에 대해서는 최신인지를 나타내는 두 번째 비트를 모두 0으로 바꾼다. 그리고 수정 리스트에 수정된 블록 ID를 넣어 블록의 변경을 표시한다.

□ 블록 요청 수행전의 위치테이블

Inode 번호 블록 번호 블록 소유 노드 /최신 블록 여부

Inode Nos	Block IDs	Location					
		N1		N2		N3	
2	12	1	0	1	1	1	1
	13	0	0	1	1	1	1
	14	0	0	0	0	1	1

□ 블록 요청 수행후의 위치테이블과 수정리스트

Inode 번호 블록 번호 블록 소유 노드 /최신 블록 여부

Inode Nos	Block IDs	Location					
		N1		N2		N3	
2	12	1	0	1	1	1	1
	13	0	0	0	0	1	1
	14	1	1	0	0	1	0

수정리스트 (dirty list)	16	35	13	..

수정리스트 (dirty list)	16	35	13	14

▶▶ 그림 3.8 쓰기 연산과 위치테이블 변경

4. 교체정책(Replacement)

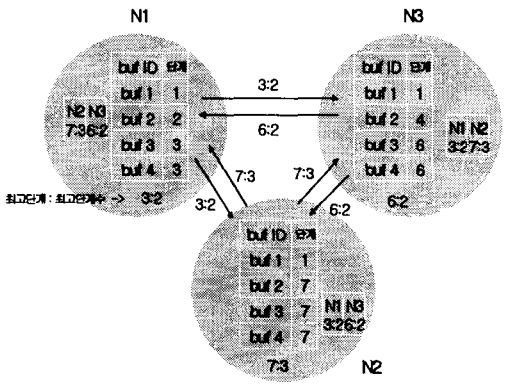
블록을 요청하여 전송받은 블록을 캐시에 저장하려 할 때, 캐시에 블록이 가득 차 있으면 블록을 교체해주어야 하는데 최적의 캐시 교체 정책은 캐시 용량이 정해져있을 때 캐시 적중률을 최대화할 수 있도록 데이터를 교체하는 정책이다. 따라서 가장 이상적이라고 할 수 있는 캐시 교체 정책은 가장 오랜 시간동안 참조하지 않을 데이터를 교체하는 정책이다. 하지만 이것은 현실적으로 구현이 어렵기 때문에 근사한 방법으로 가장 오랫동안 참조되지 않은 데이터를 교체하는 LRU(Least Recently Used) 정책을 사용한다.

제안하는 알고리즘에서는 캐시 교체시에 비활동적인 노드, 즉 오랫동안 참조되지 않은 캐시의 수가 많은 노드를 찾아서 그 노드에서 가장 오랫동안 사용하지 않은 블록과 교체한다. 비활동적인 노드를 선택할 때에는 대체적으로 비활동적인 노드를 선택하는 정책을 사용한다. 물론 가장 정확하게 비활동적인 노드를 선택하는 것이 최선을 방법으로 훨씬 더 좋은 성능을 기대할 수도 있으나, 이를 위한 정보유지 비용이 오히려 성능을 저하시킬 수 있다. 따라서 정보유지 비용을 고려하여 성능에 크게 영향을 주지 않을 정도의 대체적으로 비활동적인 노드를 선택하는 정책을 사용한다.

4.1 캐시의 Age정보

AM(Age Manager)는 비활동적인 노드를 선택하기 위해 다음과 같은 Age정보를 제공한다. 각 노드는 일단 블록이 처음 사용이 되고 다시 사용되기까지의 경과된 시간을 N단계로 나누는데, 단계가 높을수록 경과시간이 오래됨을 의미한다. 그리고 가장 높은 단계에 해당하는 블록의 개수를 유지하여 다른 노드들과 이에 대한 정보를 서로 교환한다. 이때, 부가적으로 메시지를 보내어 정보를 교환하는 것이 아니라, 블록 검색이나 블록 전송시에 메시지를 교환하게 될 때 Age정보를 첨부함으로써 부가적인 메시지수를 줄인다. 그리고 일정시간동안 메시지를 주고받지 않은 노드에 대해서는 강제적으로 메시지를 전송시킴으로써 Age정보를 수집하게 된다.

AM은 메시지 교환시 얻어낸 각 노드에 대한 Age정보를 Age-List에 저장하고, 캐시 교체시에 이 정보를 이용하여 높은 단계에 해당하는 블록의 개수가 많을수록 비활동적인 노드로 간주한다.



▶▶ 그림 3.9 노드간의 에이지정보 교환

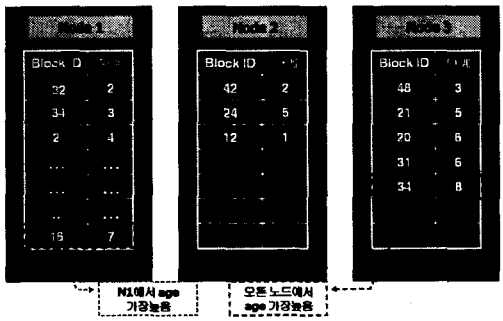
그림 3.9는 세 개의 노드가 각각의 Age정보를 서로 교환하고 있는 그림이다. 노드N1의 경우 가장 높은 단계의 블록은 buf3과 buf4로 이 블록들의 단계는 3이 되겠고, 따라서 노드N1의 Age정보는 최고단계:최고단계의 블록수 = 3:2 이다. 노드N2의 경우 역시 가장 높은 단계의 블록은 buf2, buf3, buf4로 이 블록들의 단계는 7이므로 노드2의 Age정보는 최고단계:최고단계의 블록수 = 7:3이다.

4.2 교체 알고리즘

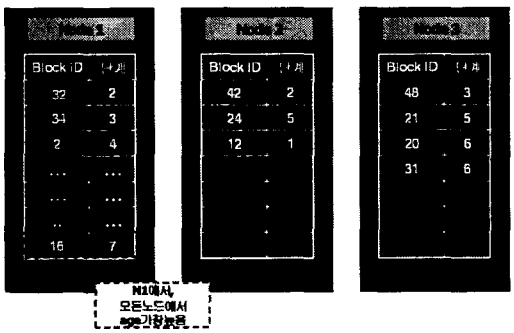
블록을 요청하여 다른 노드에서 블록을 가져오거나 디스크로부터 블록을 가지고 왔을 때 노드의 캐시영역에 블록이 가득 차 있는 경우 캐시를 교체한다. 먼저 노드에서 Age가 가장 높은 블록을 우선순위로 하여 교체

할 블록을 선택한다. 그리고 AM에 의해 수집된 Age-List를 참조하여 블록을 전송할 노드를 선택한다. 노드 선택의 기준은 먼저 Age가 가장 높은 단계를 가지고 있는 노드를 우선순위로 하며, 여러 노드가 선택될 시에는 다시 그 노드들 중에서 높은 단계의 블록수가 가장 많은 노드를 선택한다. 선택된 노드에 블록을 전송하고 블록의 위치를 관리하는 노드에 변경된 위치를 알려주는 메시지를 보낸다. 마지막으로 수정리스트를 이용하여 선택된 노드의 가장 오래된 블록의 유효성을 검증하여 블록을 버리거나 디스크에 쓰고 전송된 블록은 그 블록의 자리로 위치시킴으로써 캐시교체를 완료한다.

그림 3.10은 노드N1이 캐시를 교체하는 그림이다. 먼저 노드N1에서 Age가 가장 높은 블록16을 선택하고 블록16을 전송할 노드를 선택하기 위해 Age-List를 참조한다. 노드N3가 8:1로 가장 비활동적인 노드이므로 노드N3에서 가장 Age가 높은 블록34의 자리에 블록16을 위치시킨다. 그리고 블록34에 대한 유효성 검증을 하여 블록을 버리거나 디스크에 쓰는 작업을 한다.



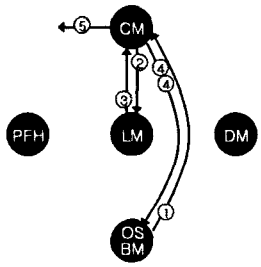
▶▶ 그림 3.10 가장 비활동적인 노드가 다른 노드인 경우의 캐시 교체



▶▶ 그림 3.11 가장 비활동적인 노드가 자신 노드인 경우의 캐시 교체

그림 3.11은 노드N1이 캐시를 교체하는 그림으로 가장 비활동적인 노드가 자신의 노드인 경우이다. 따라서 노드N1에서 Age가 가장 높은 블록16은 전체 노드들에서도 Age가 가장 높은 블록이므로 블록16에 대한 유효성 검증을 한다.

교체된 블록에 대한 유효성 검증은 먼저 수정리스트에 블록이 있는지 여부를 조사한다. 블록이 수정 리스트에 없는 경우, 블록은 읽기 연산에 의해서만 수행된 것이므로 디스크에 쓸 필요 없이 버린다. 그리고 블록이 수정 리스트에 있는 경우, 블록이 쓰기 연산에 의해 변경되었으므로 그림 3.12와 같이 블록의 유효성을 검사하여 처리한다. 블록이 유효한지를 판단하기 위해 CM에게 질의를 내리고 CM는 블록의 유효성, 즉 블록의 상태정보가 최신본인지를 알아보기 위해 LM에게 질의를 내린다. 최신본인 경우 블록을 디스크에 쓴 뒤에 버리면서 LM은 위치 테이블에서 블록에 대해 첫 번째 비트가 표시되어 있으면서 최신비트가 표시되지 않는 블록들로 구성된 Invalidating List를 CM에게 함께 반환한다. CM는 이 Invalidating List를 보고 해당 노드에 강제로 invalidating 메시지를 전송한다. 그리고 블록이 최신본이 아닌 경우 더 이상 그 블록은 필요하지 않으므로 블록을 버리는 작업을 수행한다.



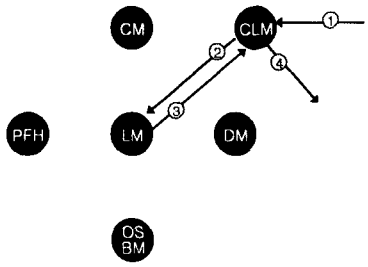
- 1 : 블록이 valid 한지 CM에 질의
- 2 : LM 에 질의하여 블록의 valid 여부 확인
- 3 : valid 여부 응답, valid 경우 invalidating list를 같이 반환
- red 4 : invalidating list 를 보고 강제로 해당 노드에 invalidating 메시지 전송
- blue 4 :블록이 valid 하므로 disk I/O 할 것을 지시
- 5 : 블록이 invalid 하므로 disk I/O 하지 않고 버릴 것을 지시

▶▶ 그림 3.12 수정 리스트에 있는 블록 처리

5. 잠금(lock) 연산

잠금은 한번에 하나의 접근만을 허용하는 동시성 제어기술로 파일시스템에서 블록에 대한 동시 접근의 문

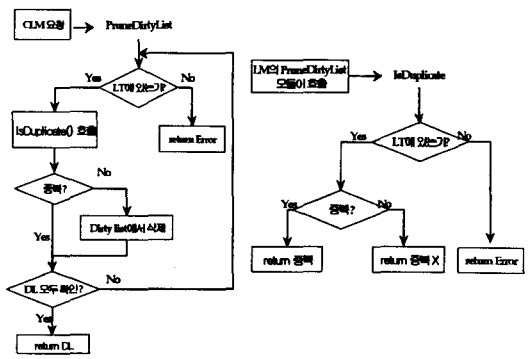
제를 해결하기 위하여 사용한다. 제안하는 알고리즘에서는 원하는 블록에 접근하기 위해서 파일단위의 잠금을 얻어 접근한다.



- 1 : CLM이 어떤 노드에서 파일에 대한 잠금을 얻으면서 그 파일에 대한 수정 리스트를 그 노드의 CM에서 가져옴
- 2 : LM에 수정 리스트에 대한 prunning 요청
- 3 : prunning 된 수정 리스트를 CLM에 전달
- 4 : 잠금을 요청한 다음 노드에 잠금 허용과 동시에 수정리스트 전달

▶▶ 그림 3.13 잠금 연산

그림 3.13은 잠금 연산을 나타낸다. 먼저 노드가 파일에 대한 잠금을 요청하게 되면 전체 노드의 잠금을 관리하고 있는 CLM(Cluster Lock Manager)이 그 파일에 대한 잠금을 가지고 있는 노드로부터 잠금을 얻는다. 그리고 그 파일에 대한 수정 리스트를 그 노드의 CM으로부터 함께 가져온다. 그리고 나서 LM에게 수정리스트에 대한 pruning을 요청한다.



▶▶ 그림 3.14 수정 리스트 pruning처리

그림 3.14는 prunning처리 알고리즘을 나타낸다. prunning 작업은 수정 리스트에서 필요없는 블록ID를 제거하는 일로 복사본이 없는 유일한 블록들의 경우 수정 리스트에 저장할 필요가 없으므로 위치테이블을 검색하여 복사본이 있는 블록들만을 수정 리스트에 유지하고 복사본이 없는 블록들은 수정 리스트에서 삭제한다.

LM은 pruning작업을 마친 후 CLM에게 다시 수정 리스트를 전달하고 CLM는 잠금을 요청한 노드에게 잠금 허용과 동시에 prunning작업을 마친 수정 리스트도 함께 보낸다.

6. 일관성 유지 (Consistency)

노드가 항상 최신의 데이터를 사용할 수 있기 위해서는 캐시된 데이터의 일관성이 유지되어야 한다. 일관성이 유지되지 않아 블록 할당이 잘못될 경우 디스크로부터 데이터를 액세스할 가능성이 높아지게 되며, 부가적인 페이지 전송요청을 위한 메시지가 빈번하게 일어남으로 시스템 성능이 크게 저하될 수 있다.

일반적으로 일관성 유지의 가장 큰 관건은 복사본의 변경이다. 사용자의 관점에서 파일의 복사본은 동일한 논리적인 개체를 가지며, 따라서 블록의 변경내용은 모든 다른 복사본에까지 반영되어야 한다.

제안하는 알고리즘에서는 블록의 내용이 변경될 때마다, 모든 다른 복사본의 내용을 곧바로 변경시키지 않고, 우선 수정 리스트에 변경 정보를 저장한 뒤, 변경된 블록을 접근하게 되었을 때 수정 리스트를 참조하여 변경시키는 방법인 LWI(Lazy Write Invalidate)알고리즘을 사용한다. LWI알고리즘은 어느 시점에서 논리적으로 보았을 때 서버가 최신의 파일 자료를 가지지는 못하기 때문에 캐시된 데이터들의 일관성이 유지되지 않은 듯 보일 수 있지만, 변경된 블록을 접근하게 되었을 때 변경된 정보가 반영되기 때문에 전체적으로 데이터의 일관성을 유지하고 있다. 이 방법은 블록의 내용이 변경될 때마다 일일이 다른 복사본들의 내용을 변경시켜주기 위해서 발생되는 추가적인 메시지 교환을 줄일 수 있다는 장점을 가진다.

Ⅳ. 결론

본 논문에서는 중앙에서 관리하는 서버 없이 전체 시스템의 캐시 효율성을 향상시킬 수 있는 분산 파일시스템의 협력캐시 알고리즘을 설계하였다. 기존의 협력캐시 알고리즘 중에 모든 캐시정보를 중앙 서버가 제공해

주고 있는 알고리즘의 경우 서버에 병목현상을 초래하고 있고, 캐시정보를 여러 노드로 분산시켜 관리하고 있는 알고리즘의 경우 캐시정보를 유지하는데 비용이 많이 드는 단점을 가지고 있다.

제안한 협력캐시 알고리즘에서는 중앙에서 관리하는 서버 없이 캐시정보를 관리하여 전체 시스템의 캐시 효율성을 향상시키고 있으며, 글로벌 캐시히트율과 로컬 캐시히트율을 동시에 만족시킨다. 또한 비용은 감소시키면서 가치있는 데이터를 우선적으로 유지하기위한 방법에 초점을 두어 캐시교체 수행을 효율적으로 처리하도록 설계하였고, 노드가 항상 최신의 캐시 데이터를 사용하도록 일관성을 유지하는 알고리즘으로 설계하였다.

▌참고문헌▐

[1] 정보통신부 홈페이지, http://www.mic.go.kr/.

[2] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. "Cooperative Caching : Using Remote Client Memory to improve File System Performance," In Proceedings of the 1st Symposium in Operating System Design and Implementation, pages 267-280, November 1994.

[3] W. H. Ahn, S. H. Park, and D. Park. Efficient Cooperative Caching for File Systems in Cluster-Based Web Servers. In Proc. of IEEE International Conference on Cluster Computing, pages 326-334, Chemnitz, Germany, November 2000

[4] Prasenjit Sarkar, John Hartman "Efficient Cooperative Caching using Hints" In Proceedings of the 2nd Symposium on Operating Systems Design and Implementation. USENIX, October 1994

[5] Michael J. Feeley, William E, Morgan, Frederic H.Pighin, Anna R. Karlin, and Henry M.Levy "Implementing Globall Memory Management in a Workstation Cluster," In Proceedings of the 15th Symposium on Operating System Principles, pages 201-212, December 1995

[6] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang "Severless Network File Systems," In Proceedings of the 15th ACM Symposium in Operating Systems Principles , pages 109-126, December 1995

[7] 김정환, 강희일, 이동일, "SAN 기술 및 시장동향," 자통신동향분석, pp.24-37, 2000.]

[8] Thomas Ruwart, "Disk Subsystem Performance Evaluation: From Disk Drives to Storage Area Networks," IEEE Symposium on Mass Storage Systems, pp.1-24, 2000.

[9] Maru G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K.Ousterhout. "Measurements of a Distributed File Systems," In Proceedings of the 13th Symposium on Operating Systems Principles, pages 198-212, October 1991.

[10] T. Cortes and J. Labarta. Linear Aggressive Prefetching: A Way to Increase the Performance of Cooperative Caches. In Proc. of the 10th Symposium on Parallel and Distributed Processing, pages 46-54, San Juan, Puerto Rico, April 1999

[11] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. "Caching in the Sprite Network File System" ACM Transactions of Computer Systems, 11(2):228-239, February, 1993.