잠금 해제 지연 일관성 모델을 기반으로 하는 분산 공유 메모리 시스템에서의 효과적인 로깅기법

(An Efficient Logging Scheme based on Lazy Release Consistent Model for Distributed Shared Memory System)

박 태 순 † 염 헌 영 ††
(Taesoon Park) (Heon Y. Yeom)

요 약 본 논문은 잠금 해제 지연 메모리 모델을 기반으로 하는 분산 공유 메모리 시스템을 위한 효과적이고 안전한 로깅 기법을 제안한다. 제안된 기법에서는 프로세스들 사이의 종속 관계가 추적되어, 실제로 종속 관계가 발생하는 경우에만 안전한 로깅을 수행하는데, 이는 프로세스들 사이에 정보 전달이 일어나는 경우 무조건 로깅을 수행하던 기존의 방법들과 비교 해볼 때, 로깅 횟수가 크게 줄어드는 효과를 낸다. 더욱이, 제안된 기법에서는 각 프로세스가 사용한 데이타를 모두 안전한 저장장소에 로깅 하는 대신, 필요한 데이타들은 그 데이타를 생성한 프로세스의 휘발성 메모리에 로깅하고, 그 데이타의 사용 정보만을 로깅 한다. 프로세스내의 결함 발생 후, 복구 과정에서, 각 프로세스는 로깅 된 사용 정보만을 이용하여 알맞은 버전의 데이타를 효과적으로 찾을 수 있다. 결과적으로 각 프로세스에서 저장되는 로그의 양 또한 줄어든다.

Abstract This paper presents an efficient stable logging scheme for the distributed shared memory system based on the lazy release consistent memory model. In the proposed scheme, inter-process dependency is traced and stable logging is performed when the dependency relation between processes actually happens. With the dependency tracking, the proposed scheme requires much less frequency of stable logging, comparing with the previous schemes in which stable logging is performed whenever any information transfer happens between processes. Also, in the proposed scheme, every data item accessed by a process is not logged, but only the access information is logged in the stable storage. For the recovery from a failure, the correct version of the accessed data items can be effectively traced by using the logged access information. As a result, the amount of logged information is also reduced.

1. 서 론

분산 공유 메모리 (Distributed Shared Memory, 이하 DSM) 시스템[11]은 네트워크로 연결된 워크스테이션들을 위한 효과적인 병렬 프로그래밍 기법을 지원하므로, 기존의 병렬 컴퓨터를 대체할 수 있는 수단으로 각광받고 있다. 이러한 DSM 시스템의 실제적인 사용을 위해서는, 결함 내성 기능을 갖추는 것이 매우 중요한데, 그 이유는, DSM 시스템에 참여하는 워크스테이션의 수가 증가함에 따라, 시스템내의 결함 발생 확률이 높아지게 되고, 실행시간이 긴 프로그램에겐 시스템내의 부분적 결함이 치명적이 될 수 있기 때문이다. 따라서, DSM 시스템의 효과적인 구현을 위해서는 시스템내의 결함 발생 후에도 프로그램을 처음부터 다시 시작하지 않도록 하는 복구기능을 갖추는 것이 매우 중요하다[20].

· 본 연구는 학술진흥재단의 자유공모과제(1997-001-E00409) 지원을 받아 수행되었음.
† 종신회원 : 세종대학교 컴퓨터공학과 교수
tspark@cs.sejong.ac.kr
†† 종신회원 : 서울대학교 전산과학과 교수
yeom@arirang.snu.ac.kr
논문접수 : 1998년 7월 2일
심사완료 : 1999년 11월 10일

DSM 시스템에 결함 내성을 제공하는 한 방법은 검사점(checkpointing) 기법과 롤백(rollback) 복구 기법을 사용하는 것이다. 검사점은 프로그램을 수행중인 시스템 내의 중간 상태를 결함에 의하여 영향을 받지 않는 안전한 저장장소(stable storage)에 저장하는 연산이다. 주기적으로 검사점을 취함으로써, 시스템 내에 결함이 발생할 경우, 프로그램을 초기 상태로부터 다시 수행시키는 대신, 저장된 중간 상태로 복구할 수 있다. 이와 같이 저장된 검사점을 이용하여 시스템을 재구성하고 그 상태로부터 다시 연산을 수행하는 것을 롤백이라고 부른다.

DSM 시스템에서는 공유 데이타의 접근에 의해, 한 프로세스의 연산 상태가 다른 프로세스의 상태에 종속되게 된다. 프로세스간의 종속관계를 특징짓는 중요한 요인은 DSM 시스템의 메모리 일관성 모델이다. 순차적 일관성 모델[11]을 기반으로 하는 DSM 시스템은 시스템 구현이 간단하다는 장점을 가지고 있지만, 공유 데이타 접근에 따른 오버헤드가 커서 성능을 떨어뜨리는 요인이 되고 있다. 이러한 이유로 많은 완화된 메모리 일관성 모델들이 개발되었다[2, 9]. 이 모델들은 전통적인 순차적 모델과는 달리, 여러 노드에 동일한 페이지의 복사본을 둠으로써 일시적으로 일관성이 깨지는 현상을 허용하는 등의 메모리 모델 완화로 성능을 향상시킨다. 이러한 모델에서는 프로세스간의 종속관계가, 순차적 모델에서 데이타를 읽을 때마다 그 데이타를 쓴 프로세스에 종속되는 것과 다르게, 잠금(lock)에 대한 해제(release)와 얻기(acquire) 연산을 통해 명시적으로 동기화가 된 쓰기 연산과 읽기 연산에 대한 경우에만 발생한다.

이러한 DSM 시스템내의 프로세스간의 종속관계 때문에, 한 프로세스가 결함 발생으로 인하여 롤백을 수행할 경우, 종속관계에 있는 다른 프로세스들도 일관된 복구선을 찾아 함께 롤백을 하여야 하는 경우가 발생한다. 만약, 검사점이 종속 관계에 있는 프로세스들 사이에서 독립적으로 설정이 된다면, 일관된 복구선을 찾는 과정에서 도미노 현상(domino effect)[15]이라고 불리는 연쇄적 롤백 현상이 나타날 수 있는데, 최악의 경우에는 초기 상태 이외의 일관된 복구선을 찾을 수가 없어 검사점을 취하였음에도 불구하고, 연산결과를 모두 잃어버리게 되는 경우가 발생할 수 있다.

이러한 도미노 현상에 대처하기 위한 한가지 방법은 조정된(coordinated) 검사점을 취하는 것이다. 이 방법은 한 프로세스가 검사점을 설정할 때마다, 관련된 프로세스들과의 조정을 통해 일관된 복구선을 만들어 내도록 함께 검사점을 설정하는 방법이다[1, 3, 6, 10, 12]. 검사점 조정을 통해 각 프로세스들의 가장 최근 검사점이 항상 일관된 복구선을 형성함으로, 도미노 현상은 나타나지 않는다. 그러나, 이 방법의 단점은 검사점 조정을 수행하는 동안 프로세스들의 정상적인 수행이 정지(blocking)되어야 한다는 것이다.

다른 방법으로는 프로세스들간에 종속관계가 생길 때마다 검사점을 취하는, 데이타 전달에 기반한 검사점(communication-induced checkpointing) 기법이 있다[7, 20]. 이 검사점 기법은 한 프로세스가 다른 프로세스로부터 데이타를 전달받아 새로운 종속관계가 생성될 때마다 검사점을 설정하므로, 롤백 파급을 제한하여 도미노 현상을 방지하는 방법이다. 따라서, 시스템내의 한 노드에 결함이 발생할 경우, 다른 노드의 프로세스들에게는 제한된 롤백 파급만이 요구되며 도미노 효과는 발생하지 않는다. 그러나, 이 방법은 데이타 전달에 따른 검사점 설정 횟수의 급격한 증가로 시스템 성능이 저하될 수 있다는 문제점이 있다.

또 다른 방법으로는 독립적인 검사점 설정과 함께 데이타 로깅을 이용하는 방법이 있다[16]. 이 방법에서는 프로세스가 연산에 사용한 모든 공유 데이타 값들을 안전한 저장 장소에 순서대로 저장함으로써, 시스템 복구시에 저장되어 있는 데이타들을 결함 발생 이전과 동일한 순서로 재사용하여 동일한 연산 결과들을 만들어낸다. 이와 같이, 결함 발생 후에도 프로세스들이 결함 발생 전과 동일한 상태를 재구성할 수 있으므로, 한 프로세스의 롤백이 다른 종속된 프로세스들에게 영향을 미치지 않는다는 장점이 있으나, 데이타 로깅에 따른 시스템 부하를 줄이는 것이 구현상의 중요한 문제가 되고 있다.

데이타 로깅의 효과적인 구현을 위한 많은 방법들이 제안되었다. 먼저, 순차적 메모리 모델을 사용하는 DSM 시스템을 위해서, 사용한 데이타를 반복적으로 로깅 하는 것이 아니라, 데이타 값은 한번만 로깅하고, 그 데이타가 사용된 시점들을 로깅 함으로써 저장하여야할 데이타의 양을 줄이는 방법이 제안되었다[19]. 또한, 각 프로세스가 사용한 데이타 값들이 무효화 될 때에만 로깅을 수행함으로써, 로깅 되는 데이타의 양을 줄이는 방법도 제안되었다[13]. [8]에서는 데이타를 매번 읽을 때마다 로깅 하는 것이 아니라, 데이타가 쓰여지는 프로세스에 의해서 로깅 되는 방법을 제시하였으며, [14]에서는 각 프로세스에 의해서 쓰여진 데이타 값과 그 데이타의 사용 구간을 해당 데이타가 무효화될 때, 한꺼번에 안전한 저장장소에 로깅 함으로써, 로깅 오버헤드를 줄

이는 방법을 제안하였다.

완화된 일관성 모델의 일종인 잠금 해제 지연 일관성 (Lazy Release Consistent) 메모리 모델을 사용하는 DSM 시스템을 위하여, [19]에서는 새로운 종속관계가 생길 때마다, 자신이 사용한 데이타 값과, 사용 구간을 안전한 저장장소에 로깅 하는 기법이 제시되었는데. 이 방법에서는 프로세스가 자신이 보유하고 있던 잠금을 해제하거나, 자신이 쓴 데이타 값을 다른 프로세스에게 전달할 때, 로깅을 수행한다. [5]에서 제안하는 로깅 방법은 데이타 값과 사용된 구간을 데이타를 생성한 프로세스의 휘발성 메모리에 저장하는 방법인데, 이 경우 안전한 저장장소로의 로깅에 필요한 부하를 줄였다는 장점은 있으나, 휘발성 메모리만을 이용함으로써, 데이타를 읽은 프로세스와 데이타를 생성한 프로세스에 동시에 결함이 발생하면 시스템 복구가 불가능하다는 단점이 있다.

본 논문에서는 잠금 해제 지연 메모리 모델을 사용하는 DSM 시스템을 위한 새로운 로깅 기법을 제시한다. 이 방법은 [16,19]에서 제시된 방법에 비해 로깅 부하가 적으며, [5]에서 제시된 방법과는 달리 여러 노드에서 동시에 발생하는 결함들을 견딜 수 있다. 먼저, 잠금 해제 지연 메모리 모델의 경우, 종속 관계는 쓰기와 읽기 연산이 잠금의 얻기, 해제 연산에 의해 동기화 된 경우에만 발생한다. 따라서, 쓰기 연산이 수행된 후에 잠금이 해제되고, 다른 프로세스가 잠금을 얻은 후 그 프로세스 내에서 읽기 연산이 수행되는 등의 종속 관계 추적을 통해 꼭 필요한 경우에만 로깅을 수행하는 기법을 제안한다. 이 방법은 불필요한 로깅을 줄임으로써, 로깅 횟수를 크게 줄일 수 있다. 또한, 사용된 데이타 값을 안전한 저장장소에 로깅 하는 대신, 생성한 데이타는 휘발성 메모리에 로깅을 하고, 사용 구간을 나타내는 정보만을 안전한 저장 장소에 로깅 하는 방법을 택한다. 그 이유는 데이타 자체는 생성한 프로세스의 롤백에 의해 언제나 재생성 될 수 있기 때문이다. 결과적으로, 안전한 저장 장소에 로깅 되는 정보는 데이타 복구에 필요한 사용 구간을 나타내는 정보뿐이므로 로깅 되는 정보의 양도 크게 줄일 수 있다.

본 논문의 구성은 다음과 같다. 2절에서는 DSM 시스템 모델과 일관된 시스템 복구선에 관한 정의가 주어지고, 3절에서는 제안된 기법과 이에 따른 복구기법을 설명한다. 4절에서는 제안된 로깅기법의 성능을 실험을 통해서 분석, 평가하고 5절에서 결론을 맺는다.

2. 연구배경

2.1 시스템 모델

본 논문에서는 네트워크로 연결된 여러 노드들로 구성된, 소프트웨어 기반의 분산 공유 메모리 시스템을 가정한다. 각 노드는 프로세서와 휘발성의 주 메모리, 그리고 비휘발성의 보조 메모리를 가지며, 프로세서들 사이의 통신은 메시지 전달 방식에 의해 이루어진다. 메시지 전달은 에러가 없고, 메시지 손실이 없다고 가정한다. 그러나, 메시지 전달 순서에 관해서는 특별한 가정을 하지 않는다.

각 프로세서들은 결함 중단(fail-stop)[17] 모델을 따른다고 가정한다. 즉, 프로세서에 결함이 발생하면 즉시 멈춤으로써 더 이상의 잘못된 연산은 수행하지 않는다. 시스템에서 고려되는 결함은 일시적 결함이라고 가정한다. 즉, 프로세서가 결함 복구 후 재실행을 시작하면, 동일한 결함이 계속해서 발생하지는 않는다. 노드에 결함이 발생하면 레지스터와 주 메모리의 내용은 손실되나 노드내의 보조 메모리의 내용은 보존되며, 보조 메모리를 시스템 결함에 영향을 받지 않는 안전한 저장장소라고 가정한다. 또한, 여러 노드에서 동시에 결함이 발생할 수 있는 모델을 가정한다.

논리적으로 DSM 시스템은 각 프로세서 상에서 수행중인 프로세스들의 집합으로 정의되며, 프로세스들은 논리적인 공유 메모리상의 데이타에 대한 읽기와 쓰기 연산을 통해 통신을 한다고 볼 수 있다. 각 프로세스는 초기 상태에서 마지막 상태에 이르는 일련의 상태들로 정의된다. 각 상태들은 서로 다른 사건들에 의해 발생하며 프로세스내의 상태 전이를 일으키는 일련의 사건들을 연산이라 정의한다. 프로세스의 연산이 주어진 입력 값, 외부로부터의 입력 값 또는 다른 프로세스들로부터의 메시지의 내용에 따라 결정적인 결과 값을 보일 때, 주어진 연산을 구간 결정적(piece-wise deterministic) 이라고 하며, 본 논문에서는 각 프로세스들이 연산에 사용한 공유 데이타 값에 따라 결정적인 쓰기 작업을 수행하는 구간 결정적 연산을 가정한다.

공유 메모리 공간은 일정한 크기의 데이타 페이지들로 구성되며, 메모리 일관성 모델로는 쓰기 무효화를 기반으로 하는 잠금 해제 지연 메모리 모델을 가정한다[9]. 이 모델에서는 동일한 데이타 페이지에 대한 여러 개의 복사본이 존재하여, 여러 프로세스들에 의해 동시에 읽기/쓰기 연산이 수행될 수 있다. 이렇게, 공유 데이타에 대한 연산 순서의 예측이 어려워짐에 따라, 충돌 가능한 연산들(conflicting operations)간의 올바른 실행 순서를 맞추기 위하여, 잠금 해제 지연 메모리 모델에서는 잠금 얻기(lock acquire), 잠금 해제(lock release),

배리어(barrier)등의 동기화 연산을 제공하며, 쓰기 연산에 의한 무효화는, 쓰기 수행 후 해제된 잠금을 얻는 시점까지 지연된다. 또한, 이 메모리 모델은, 두 프로세스가 동일한 페이지에 대한 쓰기 연산을 경쟁적으로 수행하는 핑퐁(ping-pong) 효과를 막기 위해, 다중 쓰기(multiple-writer)를 허용한다.

2.2 일관된 복구선

먼저, S(i,a)를 프로세스 Pi의 a번째 동기화 연산이라고 하고, S(i,a)와 S(i,a+1) 사이의 연산을 나타내는 상태 구간(state interval)을 I(i,a)로 표시한다. 단, I(i,0)은 시스템의 시작부터 첫 번째 동기화 연산 사이의 상태 구간을 나타낸다. 잠금 해제 지연 메모리 모델을 사용하는 DSM 시스템에서는 I(i,a)에서 수행된 쓰기 연산의 효과가 다른 프로세스 Pj에게 보여지려면, Pi가 I(i,a) 이후에 해제한 잠금을 Pj가 얻거나, 아니면, 두 프로세스가 I(i,a)의 연산 다음에 같은 배리어를 통과하여야 한다. 따라서, 시스템내의 상태 구간들 사이의 종속 관계는 다음과 같이 정의될 수 있다. 정의에 사용된, acquire(x), release(x), barrier(x)는 잠금 x에 대한 잠금 얻기, 잠금 해제, 배리어 연산을 나타낸다.

정의 1: 다음의 조건 중 한가지가 만족되면, 상태 구간 I(i,a)는 I(j,b)에 종속되었다고 한다.

(1) i=j 이고 a=b+1 이다.

(2) I(j,b)가 release(x)로 끝나고, I(i,a)는 acquire(x)로 시작하며, Pi는 I(i,a)에서 Pj가 I(j,b) 에서 생성한 데이타 값을 사용한다.

(3) I(j,b)가 barrier(x)로 끝나고, I(i,a)는 barrier(x)로 시작하며, Pi는 I(i,a)에서 Pj가 I(j,b) 에서 생성한 데이타 값을 사용한다.

(4) I(i,a)는 I(k,c)에 종속되고, I(k,c)는 I(j,b)에 종속된다. □

이와 같은 상태 구간들 사이의 종속 관계는 한 프로세스가 롤백을 수행한 후, 연산을 다시 시작하는 경우에 일관성 문제를 발생시킬 수 있다. 그림 1은 일관성 없는 롤백 복구선의 전형적인 예를 보여주고 있다[4]. 그림에서 R(X1)과 W(X1)은 페이지 X1에 대한 읽기와 쓰기 연산을 각각 나타내며, acquire(A)와 release(A)는 잠금 A에 대한 얻기와 해제연산을 각각 나타낸다.

그림 1에서 프로세스 Pi는 해당 노드의 결함 발생으로, 마지막 검사점 Ci로 롤백 하였다. 이때, 만약 프로세스 Pi가 결함 전에 수행하였던 W(X1) 연산을 복구하지 못한다고 가정하자. 그러면, Pi와 Pj사이의 일관성이 깨어지게 되는데, 왜냐하면, Pj의 현재 수행중인 연산은 Pi의 롤백에 의해 잃어버린 연산에 종속되기 때문이다. 이러한 경우, 고아 메시지(orphan message) 사례가 발생하였다고 하며, 시스템내의 롤백 복구 과정 후에 고아 메시지 사례가 발생되지 않는 경우, 시스템은 일관된 복구선으로 복구되었다고 한다[14].

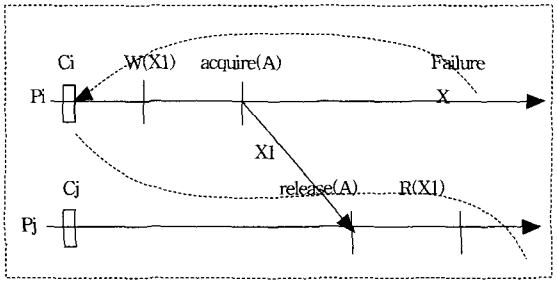


그림 1 일관성 없는 복구선의 예

3. 제안된 기법

3.1 로깅 기법

프로세스가 구간 결정적인 연산을 수행하는 경우에는, 사용된 공유 데이타 값들을 순서대로 로깅 하였다가, 롤백 복구 시 동일한 연산 지점에서 다시 사용한다면, 그 프로세스는 동일한 연산 결과를 다시 만들어 낼 수 있다[16]. 따라서, 일관된 복구선으로 롤백하기 위해서, 각 프로세스는 두 가지 종류의 정보를 로깅 하면 되는데, 하나는 자신이 사용한 데이타 페이지의 내용이고, 다른 하나는 그 페이지를 사용한 연산 시점에 관한 정보이다.

먼저, 데이타 페이지의 내용에 관한 로깅은 그 페이지를 사용한 프로세스가 할 수도 있고, 그 페이지를 생성해 낸 프로세스가 할 수도 있다. 페이지를 사용한 프로세스가 로깅을 할 경우에는, 프로세스의 결함 발생 후에도, 해당 페이지를 재사용할 수 있도록 하기 위해, 안전한 장소에 로깅을 해야 한다. 그러나, 페이지를 생성한 프로세스가 로깅 하는 경우에는, 휘발성 메모리를 이용할 수 있다. 왜냐하면, 만약 페이지를 생성한 프로세스에 결함이 발생하여 페이지의 내용이 손실되더라도, 올바른 복구 과정을 통해, 동일한 내용의 페이지를 다시 생성해 낼 수 있기 때문이다. 더욱이 한 프로세스가 생성한 데이타는 여러 프로세스에 의해 사용되어지므로, 데이타를 생성한 프로세스가 로깅을 수행하는 것이 사용한 프로세스가 로깅 하는 것보다 효율적이다. 특히, 잠금 해제 지연 메모리 모델의 경우에는 각 프로세스가 자신이 갱신한 데이타 페이지의 내용을 휘발성 메모리

에 저장 해두는 *diff*라는 자료 구조를 사용하는데, 이 *diff*를 이용하면, 별도의 로깅 작업 없이, 데이타를 로깅 한 효과를 얻을 수 있다.

잠금 해제 지연 메모리 모델에서는 프로세스가 데이타 페이지에 쓰기 연산을 수행할 경우, 먼저 *twin*이라 불리는 데이타 페이지의 복사본을 만든 후, 쓰기 연산을 수행한다. 나중에 다른 프로세스가 그 데이타 페이지의 사용을 요청할 때, 수정된 데이타 페이지와 *twin*의 내용을 비교하여, 수정된 부분만을 요청한 프로세스에게 보내 주는데, 이 부분을 *diff*라 부른다. 데이타 페이지를 요청한 프로세스는 해당 페이지에 대해 쓰기 연산을 수행한 모든 프로세스들로부터 이러한 *diff*들과 그 생성 시간을 전달받아, *diff*를 생성 시간에 따라 적용하여, 가장 최신의 데이타 페이지를 만들어 낸다. 이 과정에서 만들어진 *diff*는 시스템이 주기적으로 수행하는 쓰레기 수집(garbage collection)을 통해 제거되기 전 까지 *diff*를 생성한 프로세스의 휘발성 메모리에 *diff*의 생성 시간과 함께 저장되어 진다. 따라서, 본 논문에서는 데이타 페이지의 내용에 대해서는 별도의 로깅 작업을 수행하지 않고, 이렇게 휘발성 메모리에 저장되어 있는 *diff*를 이용한다.

이렇게 저장된 *diff*를 프로세스가 결함 회복 후 재실행 과정에서 정확한 연산 시점에 사용하도록 하기 위해서는 *diff*의 생성 시점과 사용 시점에 관한 정보가 또한 로깅 되어져야 한다. 이러한 정보를 효과적으로 표현하기 위하여, 본 논문에서는 잠금 해제 지연 메모리 모델에서 제공하는 벡터 시간(vector time)을 사용한다. 시스템내의 모든 프로세스 Pi는 자신의 벡터 시간 Ti를 관리하는데, Ti는 N개의 정수 원소를 가지는 배열로 표현되며, 초기값은 모든 엔트리에 대해 0의 값을 갖는다. 여기서, N은 시스템내의 프로세스의 수를 나타낸다. Ti의 i 번째 원소, Ti[i]는 Pi가 쓰기 연산을 수행한 후 잠금을 해제하는 시점에서 1 씩 증가하며, Ti의 k 번째 원소, Ti[k]는 Pi가 또 다른 프로세스 Pj로부터 해제된 잠금을 얻는 시점에서 Tj[k]와 현재 Ti[k]중 큰 값으로 정해진다. 결과적으로, 벡터 시간은 프로세스들의 각 동기화 연산 사이의 인과적 순서(causal order)[9]를 나타내게 되며, 정의 1의 종속 관계도 벡터 시간에 의해 표현될 수 있다. 즉, 모든 k에 대하여, $Ti[k] \leq Tj[k]$ 의 관계를 만족하면, Tj 시점의 Pj의 연산은 Ti 시점의 Pi의 연산에 종속된다.

이와 같은 벡터 시간은 잠금 해제 지연 메모리 모델에서 *diff*를 데이타 페이지에 적용시키는 과정에서 사용된다. 즉, 프로세스가 자신의 벡터 시간 T에 어떤 데이타 페이지를 사용하려면, 벡터 시간 T 이전에 수행된 쓰기 연산에 의해 생성된 *diff*들만이 적용된 데이타 페이지를 생성해내야 한다. 따라서, 각 *diff*들이 생성된 벡터 시간과 연산을 수행하려는 프로세스의 벡터 시간을 비교하여 해당 *diff*들만을 벡터 시간 순서대로 적용 시켜 최신 버전의 페이지를 생성해내는 것이다. 이와 같은 데이타 페이지 생성 과정은 시스템 복구 시에도 같은 방식으로 이용될 수 있다. 즉, 각 *diff*가 생성된 벡터 시간과 각 프로세스가 *diff*를 사용한 벡터 시간을 저장 해 둔다면, 정상적인 수행 시와 같은 방법으로 결함 발생 후에도 공유 데이타 페이지를 정확한 연산 시점에서 사용할 수가 있다.

동기화 연산과 관련된 벡터 시간을 효율적으로 로깅하기 위하여, 시스템내의 각 프로세스 Pi는 자신이 수행한 동기화 연산의 횟수를 나타내는 동기 연산 번호(synchronization operation number, 이하 Syn_i)를 관리하며, 각 동기화 연산에서, Pi의 벡터 시간이 바뀐 경우, 현재의 Syn_i와 Pi의 변경된 벡터 시간을 로깅 한다. 그림 2는 3개의 프로세스로 이루어진 시스템에서의 로깅 예를 보여주고 있다. 그림 2에서 Ti:a=(i,j,k)는 Syn_i의 값이 a인 동기화 연산 시점에서의 Pi의 벡터 시간을 나타낸다. 벡터 시간이 변경된 경우에만 로깅이 수행되므로, 시스템에서는 4번의 로깅이 수행된다.

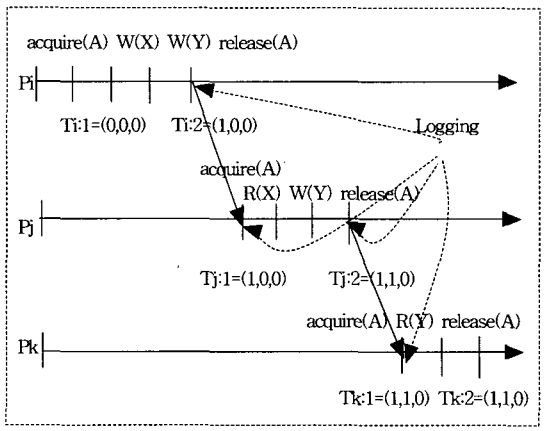


그림 2 로깅 연산의 예

여기서 주목할 점은 잠금 해제 지연 메모리 모델의 경우 벡터 시간은 동기화 연산에 의해서만 변경된다는 점이다. 따라서, 임의의 데이타 페이지가 생성된 시간은 페이지에 대한 쓰기 연산이 수행되기 직전의 동기화 연산 시 결정되며, 데이타 페이지를 사용한 시간도 그 직전의 동기화 연산을 수행한 시간으로 결정된다. 따라서,

동기화 연산과 관련된 벡터 시간만 저장하면, 모든 읽기/쓰기 연산과 관련된 벡터 시간을 유추해낼 수 있다. 예를 들어, 그림 2의 Pk가 결함 발생 후, release(A)까지 복구를 해야 하는 상황이라고 가정하자. 이때, Pk는 자신의 첫 번째 동기화 연산에 대한 벡터 시간 (1,1,0)을 로깅 해두고 있다. 따라서, 다음 상태 구간 동안에 수행할 R(Y) 연산을 위한 데이타 페이지는 (1,1,0) 이전, 즉, (0,0,0)에서 Pi에 의해 생성된 페이지 Y의 *diff*와 (1,0,0)에서 Pj에 의해 생성된 페이지 Y의 *diff*가 적용된 데이타 페이지를 사용하면 된다. 이와 같이 동기화 연산과 관련된 벡터 시간만을 로깅 하여도, 프로세스는 결함 복구 시 정확한 데이타를 추적하여 사용할 수 있다.

그러나, 데이타 페이지의 내용, 즉 *diff*들과는 달리 벡터 시간은 프로세스에 결함이 발생하면 재생성 해낼 수가 없기 때문에 프로세스의 휘발성 메모리에 저장될 수 없다. 따라서, 각 프로세스는 벡터 시간을 안전한 저장 장소에 로깅 하여야 하는데, 이 경우 비록 로깅 되어지는 정보의 양은 작지만 빈번한 로깅으로 시스템의 성능이 저하될 우려가 있다. 로깅의 빈도 수를 줄이기 위하여, 본 논문에서는 Syn_i와 벡터 시간의 값을 일단 휘발성 메모리에 로깅 하였다가, 프로세스들 간의 새로운 종속관계가 발생하는 경우에만 안전한 저장장소로 옮기는 방법을 제안한다. 만약, 프로세스가 결함으로 인하여 연산 결과를 잃어버린 경우, 손실된 상태 구간에 종속되어 있는 프로세스들은 고아 메시지 사례를 없애기 위해 같이 롤백을 수행하여야 한다[4]. 그러나, 만약, 손실된 상태 구간에 종속된 프로세스가 없다면, 프로세스 복구시의 임의의 재실행은 시스템 상태를 일관성 없는 상태로 만들지 않는다. 따라서, 로깅이 필요한 경우는 종속 관계가 발생하는 경우뿐이다.

정의 1에 따르면, Pi의 상태 구간 I(i,a)는 Pj의 상태 구간 I(j,b)에 다음과 같은 경우에만 종속된다. I(j,b)에서 W(X)를 수행하고, release(A)(혹은 barrier(A))를 한 후, I(i,a)에서 acquire(A)(or barrier(A))를 수행하고, I(i,a)에서 R(X) 또는 W(X)를 수행한 경우이다. 따라서, Pj의 입장에서는 쓰기 연산, 잠금 해제 연산, 다른 프로세스의 잠금 얻기 요청에 의한 잠금 전달(lock transfer), 그리고 diff 요청에 의한 diff 전달이 연속적으로 발생할 경우에만 종속 관계의 가능성이 있다고 볼 수 있다.

안전한 저장 장소로의 효율적인 로깅을 위해서, 제안된 기법에서는 각 프로세스 Pi가 *DSLi* (dependency-since-last-logging)라는 변수를 관리하는데, 이것은 Pi가 마지막으로 벡터 시간을 안전한 저장 장소에 로깅한 이후에 새로운 종속 관계 발생 가능성이 있는지를 나타낸다. *DSLi*는 초기에 0으로 선언되고, *DSLi* 값이 1이면 마지막 안전한 저장 장소로의 로깅 이후 쓰기 연산이 수행되었음을, 2이면 쓰기 연산 수행 후, 잠금 해제 연산까지 수행되었음을 의미한다. 따라서, Pi는 다른 프로세스로부터 잠금 얻기 요청을 받은 후, *DSLi*가 2인 경우에만 현재 휘발성 로그의 내용을 안전한 저장 장소에 로깅 한 후 *diff*를 전달한다. 결과적으로, 그림 2에서 휘발성 메모리로의 로깅은 4번 수행되지만, 안전한 저장 장소로의 로깅은 Pi로부터 Pj로 잠금 A가 전달되는 시점과 Pj로부터 Pk로 잠금 A가 전달되는 시점에서 2번만 수행된다. 제안된 기법은 그림 3에 요약되어 있다.

3.2 검사점 설정

시스템내의 각 프로세스는 결함 발생 후 재실행하여야 하는 연산의 양을 줄이기 위하여, 주기적으로 검사점을 취한다. 검사점은 프로세스의 중간 상태(context), 현재의 벡터 시간, *Syn*, *DSL*의 값과, 현재 프로세스가 가지고 있는 데이타 값(*diff*들)과 메모리 일관성 모델에서 사용되는 여러 정보들을 포함한다. 종속관계가 있는 프로세스들이 조정된 검사점을 취할 필요는 없지만, 검사점 설정이 배리어 연산이나 쓰레기 수집 작업과 연관되어 수행된다면, 별도의 통신 부담 없이 조정된 검사점을 취할 수 있다.

```
Initially, Syni=DSLi=0;

When Pi Acquires a Lock :
If (Any Change in Ti) Then{
        Volatile_Log=Volatile_Log U (Ti:Syni);
}
Syni++;

When Pi Writes on a Data Page :
If (DSLi==0) Then
        DSLi++;

When Pi Releases a Lock or Reaches a Barriers :
If (Any Changes in Ti) Then{
        Volatile_Log=Volatile_Log U (Ti:Syni);
}
Syni++;
If (DSLi==1) Then
        DSLi++;

When Pi Receives a Lock Acquire Request :
If (Volatile_Log ≠ ∅ and DSLi ==2) Then{
        Stable_Log = Stable_Log U Volatile_Log;
        DSLi=0;
}
```

그림 3 제안된 로깅 기법

3.3 롤백 복구

프로세스 Pi의 결함 발생 후 복구 작업은 그림 4와 같이 수행된다. 먼저, 가장 최근에 저장된 검사점을 읽어들여 프로세스의 상태를 검사점 취득시의 상태와 동일하게 설정하고, 새롭게 설정된 상태로부터 다음과 같은 과정을 거쳐 프로세스의 재실행을 시작한다.

잠금 얻기 연산 시점 : 프로세스는 *Syn* 값을 증가시키고, 저장된 로그에서 현재의 *Syn*값과 같은 값을 가지는 로그 항목(entry)이 있는지 검색한다. 만약, 존재한다면, 현재의 벡터 시간을 해당 로그 항목의 벡터 시간으로 설정한다.

데이타 페이지 요청 시점 : 프로세스는 자신이 필요한 데이타 페이지가 자신의 로컬 메모리에 존재하지 않으면(data page access miss 시), 해당 페이지에 대한 요청 메시지와 함께, 현재의 벡터 시간을 다른 프로세스들에게 방송한다. 다른 프로세스들은 방송된 벡터 시간보다 먼저 만들어진 *diff*를 보관하고 있는 경우, 이를 모두 보내 준다. 만약, 해당하는 *diff*가 없는 경우에는 NULL 메시지를 보내준다. 복구중인 프로세스는 전달받은 *diff*들을 시간 순서대로 정렬하고, 페이지에 적용하여 최신 버전의 데이타 페이지를 만들어 낸다. 이 과정은 결함

```
State(Pi)=Last_Checkpoint(Pi);
Syni=Last_Checkpoint(Pi).Syni;
Ti=Last_Checkpoint(Pi).Ti;
Next_Entry=Stable_Log.First;
Next_Syni=Next_Entry.Syni;
Resume(Normal_Computation) as Follows;

When Pi Acquires a Lock A :
Syni++;
If (Syni==Next_Syni) Then{
    Ti=Next_Entry.Ti;
    Next_Entry=Stable_Log.Next;
    For (any X ∈ A.Write_Notice)
            Invalidate(X);
}

When Pi Reads a Data Page X :
If (Access_Miss) Then{
    Broadcast(Page_Request(X),Ti);
    Wait_for(Reply from Every Pj);
    Create(X) by Applying diffs from Pj;
}
Read(X);

When Pi Writes on a Data Page X :
If (Access_Miss) Then{
    Broadcast(Page_Request(X),Ti);
    Wait_for(Reply from every Pj);
    Create(X) by Applying diffs from Pj;
}
Write_on(X);

When Pi Releases a Lock or Reaches a Barriers :
Syni++;
If (Syni==Next_Syni) Then{
    Ti=Next_Entry.Ti;
    Next_Entry=Stable_Log.Next;
}

When Pi Receives (Page_Request(X),Tj) from Pj :
If (Ti<Tj) Then
    Hold the Request until Ti[j] ≥ Tj[j];
Else If (diff with T ≤ Ti exists) Then
    Sends(diff,T);
Else
    Send(NULL);
```

그림 4 롤백 복구 기법

이 발생하지 않은 경우의 수행과 동일한 방식이다. 새롭게 만들어진 데이타 페이지는 벡터 시간이 변하기 전까지 사용되며, 그 후 무효화된다.

이렇게 생성된 데이타 페이지가 불필요하게 무효화되는 것을 방지하기 위하여, 잠금을 얻는 시점에서 잠금과 함께 전달되는 write notice들을 이용할 수도 있다. write notice는 어느 프로세스에서 해당 페이지에 쓰기 연산을 수행했는가에 관한 정보를 포함하고 있으므로, 잠금을 얻는 프로세스는 자신이 가지고 있는 데이타 페이지들 중 write notice에 포함된 페이지만 무효화시키면 된다. 이러한, write notice들을 벡터 시간과 함께 로깅 하면, 시스템 복구 중 페이지 무효화 횟수를 줄일 수 있다. write notice의 로깅은 로깅 되는 정보의 양을 약간 증가시키나, 로깅의 횟수와는 무관하다.

잠금 해제 연산 또는 배리어 연산 시점 : 프로세스는 *Syn* 값을 증가시키고, 저장된 로그에서 현재의 *Syn* 값과 같은 값을 가지는 로그 항목을 검색한다. 만약, 존재한다면, 현재의 벡터 시간을 해당 항목의 벡터 시간으로 설정한다.

이와 같은 방법에 의해, 각 동기화 연산 시점에서, 프로세스는 결함 발생 이전과 동일한 벡터 시간을 설정할 수 있으므로, 다른 프로세스로부터 동일한 *diff*들을 전달받을 수 있다. 이것은, 읽기/쓰기 연산을 결함 발생 이전과 동일하게 수행할 수 있음을 의미하며, 또한 결함 발생 이전과 동일한 연산 결과를 만들어 냄을 의미한다.

그러나, 만약 프로세스 Pi의 롤백 복구 중, 또 다른 프로세스 Pj가 시스템 결함으로 인한 복구 작업중이라

면, Pi의 데이타 페이지 요청에 대해 Pj는 부적절한 응답을 보낼 수가 있다. 예를 들어, Pi가 벡터 시간 T 전에 만들어진 데이타 페이지 X의 *diff*들을 요청하였다고 하자. 이때, 만약 Pj는 벡터 시간 T 전인 T'에 X의 *diff*를 만든 적이 있으나, 현재는 롤백 복구 중이며, 벡터 시간 T'까지의 복구가 끝나지 않았다면, Pi의 요청에 대해 적절한 diff의 내용을 보내 줄 수가 없을 것이다. 이 경우 Pi는 Pj가 T'까지의 복구를 마치기를 기다려 해당하는 *diff*를 얻은 후에야 정확한 복구 작업을 수행할 수 있다. 따라서, 롤백 복구 중인 프로세스는 데이타 요청을 방송한 후 모든 프로세스들로부터 응답을 받을 때까지 기다리고, 데이타 요청을 받은 프로세스는 먼저 자신의 현재 벡터 시간과 요청된 벡터 시간을 비교한다. 이때 만약, 현재 벡터 시간이 요청된 벡터 시간 보다 작다면, 해당 프로세스 역시 롤백 복구 중이므로, 데이타 요청 메시지의 처리를 해당 벡터 시간이 커질 때까지 지연시킨다. 이 경우에도 잠금 해제 지연 메모리 모델의 특성상 프로세스간의 순환적 종속 관계는 발생하지 않으므로, 교착상태의 발생 가능성은 없다.

다음의 정리들은 본 논문에서 제안하는 로깅 기법과 롤백 복구 기법의 정확성을 증명한다.

정리 1 : 시스템내의 모든 유효한 상태 구간 I(i,a)에 대해서, 만약 I(i,a)에 종속적인 임의의 상태 구간 I(j,b)가 존재한다면, I(i,a)를 시작하는 잠금 얻기 또는 배리어 연산의 벡터 시간은 롤백 후 복구 가능하다.

증명 : 상태 구간 I(j,b)가 I(i,a)에 종속되기 위해서는, I(i,a)에서 생성된 *diff*가 I(j,b)에서 읽기나 쓰기 연산에 사용되어져야만 하고, 그러기 위해서는 I(i,a) 구간이 끝나는 잠금 해제 연산 수행 시 *DSLi*의 값은 2가 되어야 따라서, Pi는 해당 잠금을 Pj에게 넘겨주기 전에, I(i,a) 구간이 끝나는 잠금 해제 연산에 대한 벡터 시간을 안전한 저장 장소에 로깅 하였을 것이다. I(i,a) 구간이 시작하는 지점의 잠금 얻기 연산 시의 벡터 시간은 로깅된 벡터시간의 I-번째 엔트리의 값을 하나 감소하여 얻을 수 있다. 따라서, 시스템 내에 하나 혹은 다중의 결함 발생 시에도 상태 구간 I(i,a)의 시작 지점의 벡터 시간은 로그로부터 얻을 수 있다. □

정리 2 : 시스템내의 모든 유효한 상태 구간 I(i,a)에 대해서, 만약 I(i,a)에 종속적인 임의의 상태 구간 I(j,b)가 존재한다면, I(i,a) 내에서 사용된 모든 *diff*들은 롤백 후 복구 가능하다.

증명 : 잠금 해제 지연 메모리 모델에서는 각 상태 구간의 벡터 시간이 그 구간에서 생성된 데이타 값에 대한 식별자로 사용되며 또한, 그 구간에서 사용한 데이타에 대한 식별자로도 이용된다. 제안된 로깅 기법에서는 이 벡터 시간이 각 상태 구간을 나타내는 번호와 함께 안전한 저장 장소에 로깅 되어 있으므로, 결함 발생 후에도, 동일한 구간에 대한 동일한 벡터 시간을 지정할 수가 있다 (정리 1 참조). 따라서, 만약, Pi를 제외한 시스템내의 모든 프로세스가 살아 있는 상태라면, *diff* 요청 메시지에 해당 벡터시간을 함께 보냄으로 해서 결함 발생전과 동일한 데이타 페이지를 만들 수 있다. 만약, Pi 이외의 임의의 프로세스 Pk가 역시 롤백 복구 중으로 Pi가 요청한 *diff*를 아직 생성하지 못 했다고 하자. 이때, Pk의 상태 구간 I(k,c)에서 *diff*가 생성될수 있다면, I(k,c)의 벡터 시간 Ti(k,c)는 I(i,a)의 벡터 시간 Ti(i,a)보다 작다. 최악의 경우의 *diff* 생성을 기다리는 상태 구간들이 I(i,a) --> I(k,c) --> ..... --> I(k',c') 의 관계를 형성한다면 (여기서, I(i,a) --> I(k,c)는 Pi가 I(i,a)를 복구하기 위해 I(k,c)에서 생성될 *diff*를 기다리고 있음을 나타낸다.), 각 해당 벡터시간은 Ti(i,a) > Tk(k,c) > ..... > Tk'(k',c')의 관계를 형성한다. 벡터 시간의 각 엔트리는 항상 0보다 크므로, 이와 같은 대기 체인은 무한할 수 없으며, 따라서, 한정된 시간 내에 Pi는 모든 *diff*를 전달받을 수 있게된다. □

정리 3 : 제안된 로깅 기법과 롤백 복구 기법은 일관성 있는 복구선을 형성한다.

증명 : 시스템내의 모든 유효한 상태 구간 I(i,a)에 대해서, 만약 I(i,a)에 종속적인 임의의 상태 구간 I(j,b)가 존재한다면, I(i,a) 내에서 사용된 모든 *diff*들은 롤백 후 복구 가능하다 (정리 2 참조). 따라서, I(i,a)는 결함 발생전과 동일한 연산 결과를 생성할 것이며, 고아 메시지 사례는 발생하지 않는다. 결과적으로, 시스템은 일관된 복구선을 형성하게 된다. □

4. 성능 평가

본 논문에서 제안하는 로깅 기법의 성능을 모의 실험과 실제 시스템 구현을 통해 기존의 기법들[16, 19]과 비교, 분석한다. 두 가지 성능 척도가 이용되는데, 그 하나는 각 프로세스에서 안전한 저장 장소에 로그 되어야 하는 *diff*의 양이고, 다른 하나는 안전한 저장 장소로의 로깅 횟수이다. 비교, 분석의 대상이 되는 로깅 기법은 다음과 같다.

◉ Shared Read Logging(SRL) [16] : 각 프로세스는 자신이 사용한 데이타 값을 휘발성 메모리에 저장하고 있다가, 자신이 생성한 데이타 값을 다른 프로세서에게 전달할 때, 휘발성 메모리에 있는 로그의 내용을 안전한 저장 장소로 옮긴다.

◉ Shared Access Tracking(SAT) [19] : 각 프로세스는 자신이 사용한 데이타 값과 전달받은 write notice의 내용들을 휘발성 메모리에 저장하고 있다가, 다른 프로세스에게 자신이 생성한 데이타를 전달하거나, 또는 다른 프로세스로부터 잠금 요청을 받는 경우, 안전한 저장 장소로 옮긴다.

◉ Reduced Stable Logging(RSL) : 본 논문에서 제안된 방법으로, 각 프로세스는 벡터 시간과 write notice만을 선택적으로 휘발성 메모리에 저장하고 있다가, 새로운 종속관계가 생기는 경우에만, 휘발성 로그의 내용을 안전한 저장장소로 옮긴다.

4.1 모의 실험 결과

16개의 프로세스와 500개의 데이타 페이지로 이루어진 시스템을 시뮬레이션 하였다. 각 프로세스는 데이타 페이지에 대한 읽기와 쓰기 연산의 연속으로 이루어진다. 쓰기 연산에 대한 읽기 연산의 비율을 읽기/쓰기 비율로 나타낸다. 읽기/쓰기 비율이 0.9이면 연산의 90%는 읽기이고, 10%는 쓰기임을 나타낸다. 지역성은 각 프로세스가 지역 메모리내의 데이타 페이지를 사용하는 비율을 나타낸다. 지역성이 0.9이면 전체 페이지 사용 연산의 90%가 자신의 지역 메모리에 있는 페이지에 대해 이루어짐을 의미이다. 동기화 연산은 쓰기나 읽기 연산이 10번 수행될 때마다, 한 번씩 수행되는 것으로 가정되었다. 즉 잠금 얻기 연산 후에 10번의 읽기나 쓰기 연산이 나오고, 다음에 잠금 해제 연산을 수행한다. 동기화 연산의 시뮬레이션을 위해서는 8개의 서로 다른 잠금이 사용되었다. 하나의 인공 프로그램은 10,000개의 읽기, 쓰기 연산으로 구성되고, 읽기/쓰기 비율과 지역성의 값을 변화 시켜 다양한 인공 프로그램들을 생성해 낸다.

먼저, 그림 5는 읽기/쓰기 비율과 지역성이 안전한 저장 장소에 로깅 되는 데이타의 양에 미치는 영향을 보여준다. SRL 방법에서는 읽기나 쓰기 연산에 사용된 모든 데이타들이 로깅 되며, 쓰기 연산에 지역적 데이타가 사용될 경우에만 로깅이 면제된다. 따라서, 읽기 비율이 100%일 때는, 사용된 데이타가 100% 로깅 되며, 쓰기 비율이 증가할수록 로깅의 양이 줄어든다. SAT 방법에서는 사용하려는 데이타 페이지를 다른 노드에서 가져와야 하는 경우에만 데이타 페이지의 로깅을 수행한다. 따라서, 쓰기 비율이 증가할수록, 데이타 페이지의 무효화 횟수가 증가하고, 다른 노드로부터 새로운 데이타를 가져와야 하는 경우가 많아져 로깅의 양이 증가한다. 두 기법 모두에서, 지역성이 감소하면 로깅의 양이 늘어나는데, 이는 적은 지역성은 다른 노드에 있는 데이타 페이지를 읽는 비율이 상대적으로 증가함을 의미하기 때문이다. 반면, 제안된 기법의 경우 안전한 저장장소로 로깅 되는 데이타의 양은 항상 0임을 알 수 있다.

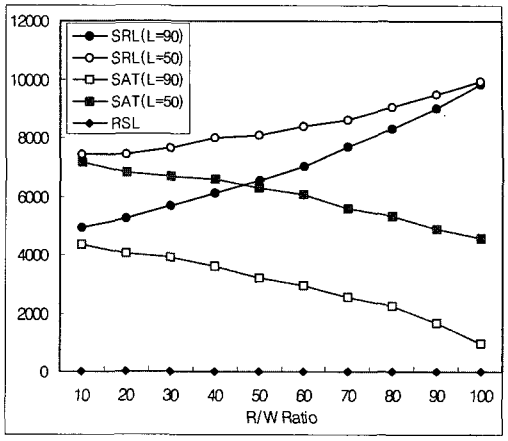
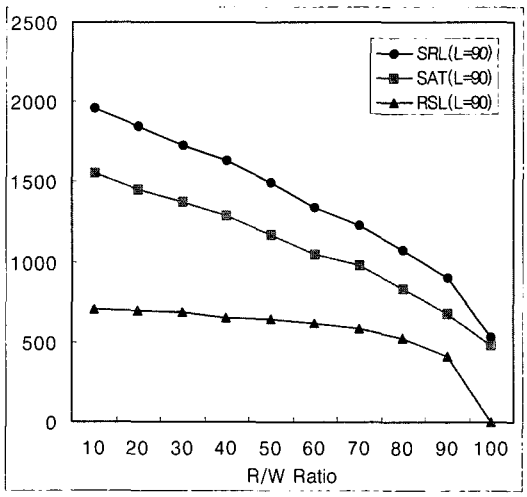


그림 5 읽기/쓰기 비율과 지역성에 따른 로깅의 양

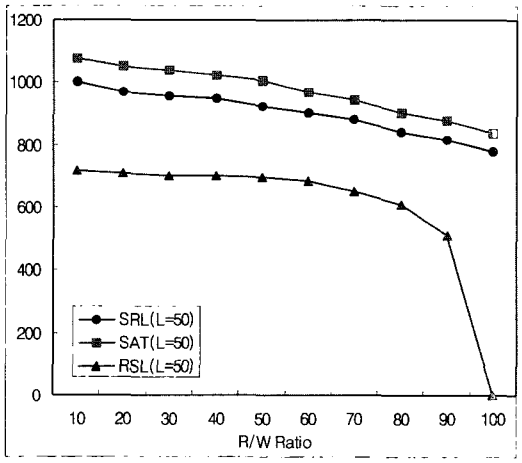
그림 6은 읽기/쓰기 비율과 지역성에 따른 안전한 저장장소로의 로깅 횟수를 보여주는데, 여기서는 데이타 값뿐만 아니라, 데이타의 사용 정보를 위한 안전한 저장장소의 접근 횟수를 포함한다. 모든 방법에 있어서, 안전한 저장장소로의 로깅은 새로운 종속관계가 형성되었다고 판단되는 시점에서 이루어지는데, 예를 들면, 데이타 값이나 write notice등이 다른 프로세스로 전달되는 시점이다. 쓰기 연산은 기존 데이타의 무효화와 새로운 데이타 전송을 의미하므로, 로깅 횟수는 모든 로깅 기법에서 쓰기 비율이 증가할수록 증가한다.

세 가지 방법을 비교해 보면, 먼저 SRL 방법은 데이타 전송 시마다 로깅을 수행하므로 쓰기 비율의 증가에 따라 로깅 횟수가 매우 빠른 증가를 보여준다. SAT 방법은 데이타 전송 시와 write notice 전송 시에 로깅을 수행하는데, 이때 휘발성 메모리에 저장되어 있는 로그의 양은 SRL 방법의 경우 보다 적으므로, 매번 로깅이 시도될 때마다의 로깅 성공 확률은 SRL 방법보다 작다. 따라서, SAT 기법은 지역성이 큰 경우, 즉 로깅 되는 정보의 양이 작은 경우에는 SRL 기법보다 적은 로깅 횟수를 보이지만, 지역성이 적은 경우에는 SRL 기법보다 로깅 횟수가 더욱 커진다. 제안된 기법에서의 로깅은 write notice 전송 시에만 수행되는데, 이는 데이타 값의 전송에 비해 그 횟수가 매우 적으며, 더욱이, write notice 전송 시 매번 로깅이 수행되는 것이 아니라, 쓰기 연산이 선행된 경우에만 로깅을 수행하므로,

다른 기법들에 비해 로깅 횟수가 매우 작아짐을 볼 수 있다.



(a)



(b)

그림 6 읽기/쓰기 비율과 지역성에 따른 로깅 횟수

4.2 시스템 구현 실험 결과

모의 실험에 의한 성능 평가 결과를 뒷받침하고, 특히 시스템의 정상 작동 중 로깅에 의한 부하가 프로그램의 실행 시간에 미치는 영향을 측정하기 위해, 본 논문에서 제안된 기법(RSL 기법)과 SAT 기법을 각각 CVM(Coherent Virtual Machine)상에 구현하고, 실행 시간과 로깅 횟수 등의 성능을 측정하였다. CVM은 다중의 메모리 모델을 지원하는 분산 공유 메모리 시스템으로, 이중 다중 쓰기를 지원하는 잠금 해제 지연 메모리 모델 상에서 로깅 기법을 구현하였다. 구현된 시스템 상에서, Barnes-hut, FFT, Traveling Salesman's Problem 등 3개의 병렬 프로그램을 수행한 후 측정된 프로그램 실행시간과 로깅 횟수에 관한 결과가 그림 7과 그림 8에 보여진다. 이때, 사용된 프로세서의 개수는 8개이다.

그림 7은 로깅이 수행되지 않는 경우와 RSL과 SAT 기법에 의해 로깅이 수행되는 각 경우의 실행 시간을 비교하고 있다. 실행된 프로그램에 따라, SAT 기법은 27.3%- 49.9%의 실행 시간 증가를 보이고 있는데 반해, 본 논문에서 제안한 RSL 기법은 단지 2.7%-10.7%의 실행 시간 증가만을 보인다. 그림8은 RSL과 SAT 기법에서의 로깅 횟수를 보여주는데, 역시 RSL 기법은 SAT 기법 보다 11.2%- 68.4% 정도 줄어든 로깅 횟수를 보여준다.

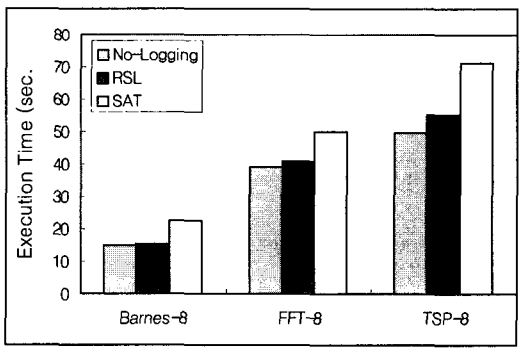


그림 7 병렬 프로그램의 실행시간

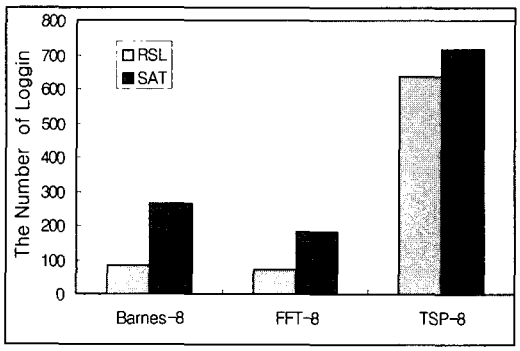


그림 8 병렬 프로그램의 로깅 횟수

5. 결 론

본 논문은 잠금 해제 지연 메모리 모델을 사용하는

DSM 시스템을 위한 새로운 로깅기법을 제안하고 있는데, 이 기법은 적은 로깅 오버헤드로 다수의 결함으로부터 시스템을 복구할 수 있다. 제안된 기법에서는 쓰기 연산에 의해 생성된 데이타 값은 쓰기 연산을 수행한 프로세스의 휘발성 메모리에 저장되고, 각 데이타 값의 복구를 위한 식별자인 벡터 시간만을 안전하게 로깅 한다. 따라서, 데이타를 생성한 프로세스의 결함 발생 후, 이 식별자를 이용하면 데이타 값들은 정확하게 복구될 수 있다. 또한, 데이타를 사용한 프로세스의 결함 시에도 벡터 시간을 이용하면, 같은 방법으로 안전한 복구가 가능하다. 로깅 되는 정보의 양뿐 아니라, 로깅의 횟수도 줄이기 위해, 프로세스간의 종속 관계를 추적하여, 종속 관계의 발생 가능성이 있는 시점에서만 로깅을 수행함으로써 로깅 횟수를 현저히 줄이고 있다. 모의 실험과 실제 구현된 시스템에서의 성능 측정 결과에 의하면, 제안된 기법이 다른 기법에 비하여 로깅 부하가 매우 적음을 알 수 있다.

참 고 문 헌

[1] R.E. Ahmed, R.C. Frazier and P.N. Marinos. Cache-aided Rollback Error Recovery(carer) Algorithms for Shared-memory Multiprocessor Systems. In *Proc. of the 20th Symp. on Fault-Tolerant Computing*, pp. 82--88, Jun. 1990.

[2] S.V. Adve and M.D. Hill. Weak Ordering -- A New Definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture*, pp. 2--14, May 1990.

[3] G. Cabillic, T. Priol and I. Puaut. The Performance of Consistent Checkpointing in Distributed Shared Memory Systems. In *Proc. of the 14th Symp. on Reliable Distributed Systems*, pp. 95--105, Sep. 1995.

[4] M. Chandy and L. Lamport. Distributed Snapshot: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, Vol. 3, No. 1, pp. 63--75, Feb. 1985.

[5] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. In *Proc. of the USENIX 2nd Symp. on Operating Systems Design and Implementation*, Oct. 1996.

[6] G. Janakiraman and Y. Tamir. Coordinated Checkpointing Rollback Error Recovery for Distributed Shared Memory Multicomputers. In *Proc. of the 13th Symp. on Reliable Distributed Systems*, pp. 42--51, Oct. 1994.

[7] B. Janssens and W.K. Fuchs. Relaxing Consistency in Recoverable Distributed Shared Memory. In *Proc. of the 23rd Annual Int'l Symp. on Fault-Tolerant Computing*, pp. 155--163, Jun. 1993.

[8] S. Kanthadai and J.L. Welch. Implementation of Recoverable Distributed Shared Memory by Logging Writes. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems*, pp. 116--123, 1996.

[9] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 18th Annual Int'l Symp. on Computer Architecture*, pp. 13--21, May 1992.

[10] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability. In *Proc. of the 25th Int'l Symp. on Fault-Tolerant Computing Systems*, pp. 289--298, Jun. 1995.

[11] K. Li. Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD thesis, Department of Computer Science, Yale University, Sep. 1986.

[12] N. Neves, M. Castro, and P. Guedes. A Checkpoint Protocol for an Entry Consistent Shared Memory System. In *Proc. of the 13th Annual ACM Symp. on Principles of Distributed Computing*, Aug. 1994.

[13] T. Park, S.B. Cho, and H.Y. Yeom. An Improved Logging and Checkpointing Scheme for Recoverable Distributed Shared Memory. In *Proc. of the 2nd Asian Computing Science Conference*, pp. 74--83, Dec. 1996.

[14] T. Park, S.B. Cho, and H.Y. Yeom. An Efficient Logging Scheme for Recoverable Distributed Shared Memory Systems. In *Proc. of the 17th Int'l Conf. Distributed Computing Systems*, May 1997.

[15] B. Randell, P.A. Lee and P.C. Treleaven. Reliability Issues in Computing System Design. *ACM Computing Surveys*, Vol. 10, No. 2, pp. 123--165, Jun. 1978.

[16] G. G. Richard III and M. Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. In *Proc. of the 12th Symp. on Reliable Distributed Systems*, pp. 58--67, Oct. 1993.

[17] R.D. Schlichting and F.B. Schneider. Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems. *ACM Trans. on Computer Systems*, Vol. 1, No. 3, pp. 222--238, Aug. 1983.

[18] R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Technical Report TR #SFB124-1592, Department of Computer Science, University of Kaiserslautern, 1992.

[19] G. Suri, B. Janssens, and W. K. Fuchs. Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory. In *Proc. of the 25th*

Annual Int'l Symp. on Fault-Tolerant Computing, Jun. 1995.

[20] K.-L. Wu and W. K. Fuchs. Recoverable Distributed Shared Memory. *IEEE Transactions on Computers*, Vol. 39, No. 4, pp. 460--469, Apr. 1990.

박 태 순

1983년 ~ 1987년 서울대학교 전자계산기공학과 학사. 1988년 ~ 1989년 Texas A&M University 석사. 1990년 ~ 1994년 Texas A&M University 박사. 1995년 ~ 1997년 세종대학교 정보처리학과 전임강사. 1998년 ~ 현재 세종대학교 컴퓨터공학과 조교수. 관심분야는 분산 시스템, 결함내성 시스템, 분산 데이타베이스 시스템.

염 헌 영

1980년 ~ 1984년 서울대학교 계산통계학과 학사. 1985년 ~ 1986년 Texas A&M University 석사. 1986년 ~ 1992년 Texas A&M University 박사. 1992년 ~ 1993년 삼성데이타시스템 선임연구원. 1993년 ~ 1998년 서울대학교 전산과학과 조교수. 1998년 ~ 현재 서울대학교 전산과학과 부교수. 관심분야는 분산 시스템, 결함내성시스템, 멀티미디어 시스템.