# Home-based Lazy Release Consistency의 락 성능 향상

## (Improving Lock Performance of Home-based Lazy Release Consistency)

윤 희 철 † 이 상 권 †† 이 준 원 ††† 맹 승 렬 †††

(Hee-Chul Yun) (Sang-Kwon Lee) (Joonwon Lee) (Seung Ryoul Maeng)

**요 약** HLRC는 LRC의 변종으로 DSM 분야에 있어 최근에 제안된 모델이다. HLRC는 여러 장점을 가지고 있지만 단점 또한 존재한다.이러한 단점 중 하나는 작은 크기의 영역을 보호하기 위해 락을 사용하는 경우에 매우 성능이 떨어진다는 것이다. 불행히도 이러한 형태의 락 사용은 가장 일반적인 것이기 때문에 락을 사용하는 응용프로그램에 있어 HLRC는 일반적으로 나쁜 성능을 보이게 된다. 본 논문에서는 HLRC를 위한 효율적인 락 프로토콜을 제안한다. 제안하는 프로토콜은 락 허가 시 임계구역 내에서 사용된 페이지에 대해 선택적으로 diff를 전달하여 갱신을 한다. 이때 전송할 diff의 최대 크기를 제한함으로써 diff의 누적현상의 발생을 최소화한다. 제안한 락 프로토콜은 임계구역 내에서의 원격 페이지 요청을 줄이며 이것은 락 대기 시간 및 메시지 전송량의 감소로 이어진다. 5개의 벤치마크 응용 프로그램을 사용하여 성능을 측정한 결과 기존의 HLRC에 비해 2%~40%의 성능향상을 얻었다.

***Abstract*** Home-based Lazy Release Consistency (HLRC) shows poor performance on lock based applications because of two reasons: (1) a whole page is fetched on a page fault while actual modification is much smaller, and (2) the home is at the fixed location while access pattern is migratory. In this paper we present an efficient lock protocol for HLRC. In this protocol, the pages that are expected to be used by acquirer are selectively updated using diffs. The diff accumulation problem is minimized by limiting the size of diffs to be sent for each page. Our protocol reduces the number of page faults inside critical sections because pages can be updated by applying locally stored diffs. This reduction yields the reduction of average lock waiting time and the reduction of message amount. The experiment with five applications shows that our protocol archives 2%~40% speedup against base HLRC for four applications.

## 1. 서 론

최근 고성능 마이크로 프로세서와 고속 네트워크의 등장으로 인해서 NOW(Networks Of Workstations)와 같은 클러스터 시스템을 병렬 처리에 사용하고자 하는 연구들이 활발히 진행중이다. 병렬 처리를 위한 프로그램 모델 중 공유 메모리(shared memory) 모델은 자연스러운 프로그래밍 모델로서 프로그래밍이 용이하다는 장점을 지닌다. 때문에 물리적으로 메모리가 분산되어 있는 클러스터 환경에서 특별한 하드웨어의 지원 없이 이러한 공유 메모리 환경을 효율적으로 제공할 수 있도록 하는 소프트웨어 분산공유메모리(Software Distributed Shared Memory: SDSM)[1] 시스템에 대한 연구가 활발히 진행되고 있다.

LRC[2]와 HLRC[3]는 이러한 소프트웨어 분산공유메모리 시스템에서 가장 대표적인 프로토콜들이다. 이들은 모두 다중 기록자(multiple writer) 프로토콜을 채용하는데 이것은 twin과 diff를 이용해 공유메모리상의 변경내용을 찾아내며 쓰기 가능한 페이지가 동시에 여

† 비 회 원 : 한국전자통신연구원 컴퓨터소프트웨어기술연구소 연구원
hcyun@etri.re.kr
†† 비 회 원 : 한국과학기술원 전산학과
sklee@camars.kaist.ac.kr
††† 종신회원 : 한국과학기술원 전산학과 교수
joon@camars.kaist.ac.kr
maeng@camars.kaist.ac.kr
논문접수 : 2001년 4월 14일
심사완료 : 2001년 8월 2일

러 개가 존재할 수 있도록 함으로서 성능 향상을 꾀한다. LRC와 HLRC의 차이는 메모리 구조와 diff의 전달 및 관리방법 다르다는 점이다. LRC에서 모든 공유 페이지는 캐쉬처럼 취급된다. diff의 생성 및 전달은 다른 프로세스가 diff를 요청할 때에 이루어진다. 페이지 부재가 발생하면 프로세스는 write notice의 기록을 보고 다른 프로세스들에게 diff 요청을 한다. diff는 앞으로 사용되지 않을 것이라는 보장이 있을 때까지 메모리에 유지되어야 한다.

HLRC에서 모든 공유 페이지는 지정된 홈을 가지고 있다. 홈은 항상 최신의 내용을 가지며 절대로 무효화되지 않는다. 홈이 아닌 프로세스는 페이지 부재가 발생하였을 때 홈으로부터 페이지 전체를 얻어온다. 홈이 항상 최신의 내용을 유지하기 위해 홈이 아닌 프로세스에서의 쓰기는 동기화 시점에서 diff를 통해 홈으로 적극적으로 전달된다. 홈으로 전달된 diff는 즉각적으로 적용되며 이후에는 홈에서나 diff를 전송한 프로세스 모두에서 해당 diff를 폐기한다.

HLRC는 홈의 개념을 도입하여 diff로 인한 메모리 부하를 줄이고 최신의 페이지를 얻기 위해 필요한 메시지의 수를 줄이고 있으며, 홈 노드에서의 메모리 접근은 오버헤드가 전혀 없다는 장점이 있다. 하지만 적절한 홈 할당이 중요하며, 변경 내용의 크기와 상관없이 페이지 전체를 얻어와야 하는 문제를 가지고 있다. 이러한 문제점은 특히 락(lock)의 동작에 있어 매우 치명적인 영향을 끼친다. 일반적으로 락으로 보호되는 메모리 영역은 그 크기가 작고 이동 공유 패턴을 보인다. HLRC는 홈이 고정되어 있기 때문에 이와 같은 이동 공유 패턴에 부적절하며 또 항상 페이지 전체를 얻어오기 때문에 메시지의 양과 대기시간이 늘어난다. 이러한 이유로 임계구역 내의 수행시간이 지연되며 락에 대한 경쟁을 유발시켜 전체 시스템의 성능이 심각하게 저하될 수 있다.

본 논문에서는 HLRC를 위한 효율적인 락 프로토콜을 제안한다. 제안하는 방법은 기존의 HLRC의 특징을 그대로 지니지만 락 허가시에 선택적으로 diff를 전송하여 페이지를 갱신한다는 점이 다르다. 갱신대상은 정확성을 높이고 diff의 누적으로 인한 오버헤드[4]를 최소화 하기 위해 락 요청자의 임계구역 내의 메모리 접근 기록과 전송할 diff의 크기를 고려해 결정한다. 제안한 프로토콜은 다음과 같은 장점을 가지고 있다. 첫째 락을 획득하는 시점에서 임계구역 내에서 사용될 가능성이 높은 페이지들을 갱신할 수 있으므로 페이지 부재로 인한 지연시간을 최소화 한다. 이러한 시간 단축은 락에 대한 경쟁을 줄여 락 대기 시간의 감소로 이어진다. 둘째 diff를 얻어와 페이지를 갱신하기 때문에 페이지 전체를 얻어오는 것에 비해 메시지 양을 크게 줄인다.

제안한 프로토콜은 KDSM(KAIST Distributed Shared Memory) HLRC 버전상에서 구현되었다. KDSM의 HLRC 프로토콜은 Princeton 대학의 HLRC[3] 프로토콜을 구현한 것이다. 성능측정은 Linux 운영체제를 사용하며 100Mbps switched fast Ethernet으로 연결된 8대의 P-III 500Mhz cluster상에서 5개의 응용 프로그램을 수행하여 측정하였다. 성능측정 결과 제안한 프로토콜은 5개의 응용 프로그램 중 4개에서 기본 HLRC 프로토콜에 비해 2% - 40%의 성능향상을 보였으며 나머지 하나의 어플리케이션에서는 성능의 차이가 없었다.

논문의 구성은 다음과 같다. 2장에서는 HLRC 프로토콜과 그 문제점에 대해 설명한다. 3장에서는 제안하는 프로토콜에 대해 상세히 기술하고 4장에서는 성능 측정 결과 및 분석을 한다. 5장에서는 관련연구를 기술하고 6장에서는 결론 및 추후 연구에 대해 기술한다.

## 2. 배경지식

이장에서는 HLRC에 대해 기술하고, HLRC에서 락(lock)의 동작이 왜 비효율적인지를 보인다.

### 2.1 HLRC

HLRC[3]는 Princeton 대학에서 제안된 페이지 기반의 다중 기록자(multiple writer) 프로토콜이다. HLRC는 LRC[2]와 마찬가지로 동기화 연산과 그 다음 동기화 연산 사이의 구간을 하나의 *interval* 로 정의하여 이것으로 프로그램의 수행을 구분한다.

HLRC는 이들 interval간의 *happen-before* 관계를 유지함으로써 RC[5]메모리 모델을 만족시킨다. 이를 위해 각 프로세스는 자신이 알고 있는 다른 프로세스들의 interval 정보를 나타내는 *vector timestamp*를 유지한다. HLRC가 LRC와 다른 점은 모든 공유 메모리 페이지에 지정된 홈 노드가 있어 홈 노드에서는 항상 최신의 정보를 가진 페이지가 유지된다는 점이다. 하지만 페이지를 갱신하는 방법은 LRC와 마찬가지로 *diff*를 이용한다. diff는 변경된 부분의 정보만을 가지는 데이터 구조로서 인터벌 종료 후에 변경된 페이지와 인터벌 내에서 해당 페이지에 대한 최초의 쓰기가 발생하였을 때 복사해둔 변경 이전의 페이지인 *twin* 과의 비교를 통해 생성된다. 이렇게 생성된 diff는 홈으로 전송되어 홈이 최신의 페이지를 가질 수 있도록 한다. 또한 인터벌 종료 시에 프로세스는 인터벌 내에서 변경된 페이지들에 대한 번호를 기록하여 저장하는데 이를 *write-notice* 라고 하며 락을 허가하는 프로세스는 락을 요청한 프로세스와의

vector timestamp 비교를 통해 필요한 상대방이 모르고 있는 interval들에 대한 write-notice들을 전달한다. 락을 획득한 프로세스는 write-notice에 따라 페이지를 무효화시키며 실제 페이지 부재가 발생하면 홈 노드로부터 페이지 전체를 얻어옴으로써 최신의 복사본을 얻는다.

HLRC는 LRC에 비해 다음과 같은 장점이 있다. [6] (1)페이지 부재가 발생하였을 때 홈 노드로 한번의 왕복 메시지만이 필요하다. (2)홈 페이지의 대한 쓰기는 diff 및 twin을 생성하지 않는다. (3)공유 페이지에 대한 모든 변경내용은 홈에 반영되기 때문에 diff를 저장할 필요가 없다. 때문에 메모리 요구량이 LRC에 비해 매우 작다. 이러한 장점으로 인해 HLRC는 LRC에 비해 성능이 좋으며 확장성이 우수하다고 알려져 왔다[3].

하지만 HLRC에서는 다음과 같은 단점이 존재한다. (1) 홈이 적절히 할당되어 있지 않은 경우 불필요하게 홈이 갱신되어야 한다. (2) 페이지 부재시 항상 전체 페이지를 얻어와야 하므로 실제 필요한 데이터보다 많은 양의 데이터가 전송될 가능성이 크다.

불행히도 이러한 HLRC의 단점은 락을 사용하는 경우에 매우 치명적인 영향을 끼친다. 다음절에서는 HLRC에서 락을 사용하였을 때 어떠한 문제점이 있는지 보인다.

### 2.2 HLRC에서 락 사용의 문제점

일반적으로 락으로 보호되는 메모리 영역은 매우 작은 크기를 가지며 락의 이동에 따라 데이터가 이동하는 이동 공유 패턴을 보인다. 이러한 특성으로 인해 락의 수행은 앞서 지적하였던 HLRC의 두 가지 문제가 모두 나타나게 된다.

첫째 홈 할당에 있어 이동공유 패턴의 특성상 고정된 홈 할당을 할 경우 락의 흐름과 상관없이 매 release마다 홈으로 diff를 전송해야 한다.

둘째 페이지 부재 발생시 항상 페이지 전체를 얻어오지만 실제로 필요한 부분은 매우 작다.

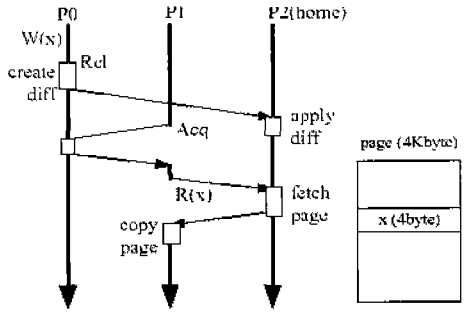


그림 1 비효율적인 HLRC의 락 동작

그림 1은 HLRC에서 락 사용 시 발생하는 프로토콜의 동작 예로 P2(홈 노드)는 실제 락의 전달과 상관없음에도 불구하고 diff의 처리와 페이지 요청에 대한 처리를 해야 하며 P1은 실제 필요한 데이터는 4byte크기지만 4Kbyte의 페이지를 전달 받아야 한다.

이처럼 비효율적인 락 처리로 인해 임계구역의 수행시간이 길어지면 락에 대한 경쟁이 생길 가능성이 더 커진다. 락에 대한 경쟁이란 어떤 프로세스가 락을 잡고 있는 동안에 다른 프로세스가 락을 요청하여 대기하는 상황을 말하며 이 경우 락을 잡고 있던 프로세스의 임계구역의 수행시간이 다른 프로세스의 락 대기 시간에 더해지게 되므로 그 효과는 점점 증폭되어 전체 시스템의 성능을 저하시키게 된다.

## 3. HLRC를 위한 향상된 락 프로토콜

### 3.1 프로토콜

본 논문에서 제안하는 락 프로토콜의 기본 아이디어는 다음과 같다 : 락 허가자는 락 요청자가 사용할 것으로 예상되는 페이지들에 대해서 락 허가 메시지와 함께 해당 페이지들에 대한 diff를 보낸다. 이후에 락을 획득한 프로세스에서 페이지 부재가 발생하면 홈에서 페이지를 가져오지 않고 전송 받은 diff를 사용하여 페이지를 갱신한다. 이를 통해 얻을 수 있는 이득은 다음과 같다. 첫째, 페이지 부재 처리시간이 짧아진다. 둘째, 임계구역 내의 페이지 부재 시간이 짧아지면 한 프로세스가 락을 잡고 있는 시간이 줄어든다. 이것은 락에 대한 경쟁을 감소시켜 락 대기 시간을 줄이는 효과가 있다.

프로토콜의 주요 동작은 크게 락 요청, 락 허가, 페이지 부재 처리의 세 단계에 걸쳐 이루어진다. 각 단계의 세부적인 동작을 살펴보면 다음과 같다.

- 락 요청 락 요청자는 해당 락으로 보호되는 임계구역 내에서 사용될 페이지들에 대한 정보를 락 요청 메시지에 포함 (piggyback) 시킨다. 사용될 페이지들은 해당 락에 의해 보호되는 임계구역 내에서의 이전 메모리 접근 기록을 바탕으로 예측한다. 락의 경우 임계구역 내의 메모리 접근 패턴이 매우 일정하므로 이전 기록을 참조하는 것으로도 좋은 예측이 가능하다.
- 락 허가락 허가자는 락 요청 메시지에 포함된 페이지 목록 중 실제로 diff를 전달할 페이지를 선택한다. 선택의 기준은 해당 페이지를 갱신하기 위해 전달해야 할 diff의 크기의 합이 페이지 크기보다 작아야 한다는 것이다. 이것은 diff 누적현상[6]으로

인한 문제를 방지하기 위함이다. diff 누적현상이란 하나의 페이지를 갱신하기 위해 전달되어야 할 diff가 여러 개가 되는 현상을 말하며 누적된 diff의 크기의 합이 페이지 크기보다 커질 경우 diff를 전달하는 것이 오히려 페이지 전체를 전달하는 것보다 커다란 메시지를 발생시켜 성능의 저하를 가져오게 된다. 때문에 이와 같이 최대 크기를 제한함으로써 이러한 오버헤드의 발생을 막는다. 선택된 페이지들에 대한 diff는 write-notice 와 함께 락 허가 메시지에 포함 (piggyback) 하여 전송한다.

- 페이지 부재 처리전달된 diff는 바로 적용하지 않고 저장하여 둔다. 이는 실제로 접근이 발생하지 않는 페이지에 대한 diff 적용을 피하기 위함이다. diff의 적용은 페이지 부재가 발생하였을 때 일어난다. 페이지 부재가 발생하였을 때 페이지를 갱신하기 위해 필요한 모든 diff들이 있으면 diff를 적용하고 그렇지 않으면 기본 HLRC 프로토콜에서처럼 홈으로부터 페이지를 얻어온다.

그림 2는 제안한 프로토콜이 기본 HLRC와 어떻게 다르며 왜 효율적인지를 보여준다. 제안한 프로토콜은 홈으로의 페이지 요청을 줄임으로서 페이지 부재시의 처리시간을 줄임을 알 수 있다. 제안한 프로토콜에서도 P2에서 한번의 페이지 요청이 발생하는데 이것은 y를 갱신하기 전송되어야 할 2개의 diff의 합이 페이지 크기보다 커졌기 때문이다. 때문에 P1은 락 허가 시 y에 대한 diff들을 전송하지 않고 P2는 y에 대해 기본 HLRC 프로토콜에서처럼 홈으로부터 페이지를 얻어온다.

### 3.2 장점과 단점

제안하는 프로토콜은 기본 HLRC 프로토콜에 비해 다음과 같은 장점을 가진다.

첫째, 임계구역 내에서의 페이지 부재 처리시간이 줄어든다. 페이지를 갱신하는 데 필요한 diff는 미리 전달되기 때문에 페이지 부재시에 이를 적용하는 것만으로 최신의 페이지를 얻을 수 있다. diff의 적용 시간은 홈으로부터 페이지를 얻어오는 시간에 비해 매우 짧으므로 페이지 부재 처리시간은 기본 HLRC 프로토콜에 비해 줄어든다. 이러한 시간 단축은 한 프로세스가 락을 잡고 있는 시간을 줄이기 때문에 락에 대한 경쟁이 감소시켜 락 대기 시간을 줄인다.

둘째, 전송되는 메시지 양이 줄어든다. 최신의 페이지를 얻기 위해 기본 프로토콜은 페이지가 전달되는 반면 제안한 프로토콜은 diff가 전달된다. 일반적으로 임계구역 내에서 변경된 영역의 크기는 작기 때문에 diff는 페이지에 비해 훨씬 작은 크기를 가진다. 따라서 전달되는 메시지의 양은 기본 HLRC 프로토콜에 비해 줄어든다. 또한 누적된 diff의 크기를 페이지 크기로 제한하기 때

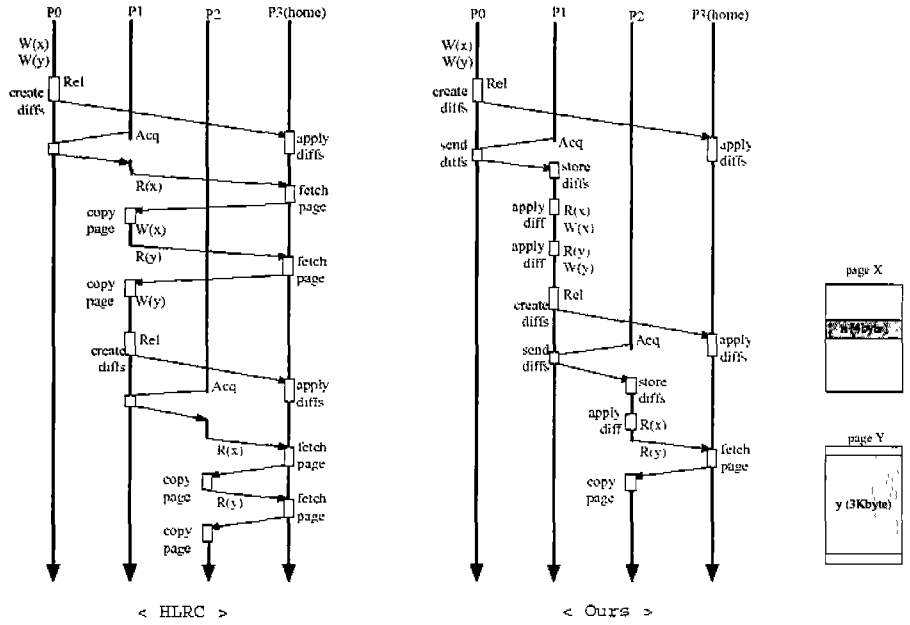


그림 2 락의 동작 예

문에 diff의 누적으로 인해 메시지양이 늘어나는 현상은 일어나지 않는다.

세째, 추가적인 메시지가 발생하지 않는다. 제안한 프로토콜을 위한 모든 정보는 기존의 메시지에 덧붙여서 보내어질 수 있다. 따라서 전달되는 메시지의 개수는 기본 HLRC 프로토콜과 동일하다.

제안한 프로토콜의 오버헤드는 홈에서의 diff생성과 diff의 저장 및 관리를 위한 메모리 부하이다. 이들은 다음절에서 설명하는 방법으로 최소화 할 수 있다.

### 3.3 홈 페이지에 대한 diff생성

제안한 프로토콜이 효율적으로 동작하기 위해서는 홈 페이지에 대해서도 diff를 생성해야 한다. 하지만 그 대상이 임계구역 내에서 접근되는 페이지만으로 한정되므로 그 수는 매우 적다. 임계구역 밖에서 접근되는 페이지들은 기본 HLRC처럼 홈에서 diff를 생성하지 않는다. 임계구역 내의 페이지이기 때문에 diff를 생성하는 경우에도 생성하는 diff의 크기를 제한하여 평균적인 생성시간을 줄일 수 있다. 홈에서의 diff 생성은 프로그램의 올바른 동작과는 무관하며 커다란 diff는 전송될 가능성이 적기 때문에 (제안한 프로토콜은 하나의 페이지에 대해 최대 페이지 크기만큼의 diff 전송만을 허용한다) 이와 같은 제한은 성능에 좋은 영향을 끼친다. 적절한 diff의 크기 제한값은 응용프로그램에 따라 달라질 수 있는데 본 논문에서 사용된 응용 프로그램들에 있어서는 실험을 통해 이 값을 256 Byte로 정하였다. 이상의 방법으로 홈 페이지의 diff 생성으로 인한 부하를 최소화 하였으며 성능 측정 결과 거의 무시할 수 있는 시간만이 걸리는 것을 확인하였다.

### 3.4 메모리 오버헤드

제안한 프로토콜은 LRC처럼 diff를 메모리에 유지할 필요가 있다. 하지만 홈 페이지에 대한 diff 생성의 경우와 마찬가지로 대상을 임계구역 내에서 접근되는 페이지로 한정하므로 diff를 유지해야 할 페이지의 수는 매우 적다. 또 LRC와 달리 홈에서 항상 최신의 정보가 유지되기 때문에 diff를 저장하지 않아도 프로그램의 올바른 수행에는 영향을 주지 않는다.

따라서 LRC와 같은 복잡한 쓰레기 정리 과정 대신 단순히 오래된 인터벌의 diff들을 메모리에서 제거하면 되므로 훨씬 간단하다.

## 4. 성능 측정

### 4.1 KDSM: KAIST Distributed Shared Memory

KDSM(KIAST Distributed Shared Memory) 시스템은 Linux 2.2.13 상에서 실행되는 사용자수준 라이브러리(user-level library)로 구현되었다. 프로세스 간의 통신은 TCP/IP를 통해서 이루어지고, 비동기 메시지의 처리를 위해서 SIGIO 시그널을 가로채는 방식을 사용한다. KDSM은 페이지 기반 무효화 프로토콜(page-based invalidation protocol), 다중읽기 다중쓰기(multiple reader multiple writer) 프로토콜을 바탕으로 HLRC(Home-base Lazy Release Consistency)[3] 프로토콜을 구현한다. KDSM의 주요 연산에 대한 시간은 표 1과 같다.

표 1 KDSM의 기본 연산 비용

| 연산 | 비용(μs) |
|---|---|
| 4KB 페이지 인출 | 1047 |
| Lock을 얻음 | 259 |
| 8 프로세서에서 Barrier | 1132 |

### 4.2 응용프로그램

성능측정은 락을 사용하는 응용 프로그램인 TSP, Water, IS, Raytrace를 이용하였다. TSP, Water, IS는 jiajia 배포판에 포함된 것을 이용하였고 Raytrace는 SPLASH2 [7]에서 포팅한 $Raytrace_{orig}$와 [8]에서 지적한 대로 상태정보를 위한 락을 제거한 버전인 $Raytrace_{rest}$ 버젼을 사용하였다.

표 2 응용 프로그램의 특징

| Appl. | size | locks | barrs | seq.time |
|---|---|---|---|---|
| TSP | 19 cities | 693 | 2 | 24.96 |
| Water | 343 mol | 1040 | 70 | 12.96 |
| $Raytrace_{orig}$ | balls4 | 120945 | 1 | 57.82 |
| $Raytrace_{rest}$ | balls4 | 2081 | 1 | 57.82 |
| IS | $2^{15}$, 10 | 80 | 30 | 7.05 |

표 2는 사용된 문제 크기, 사용된 락과 배리어의 수, 그리고 단일 프로세스에서의 실행시간을 보여준다.

### 4.3 성능 측정 결과

표 3은 8프로세스에서 수행하였을 때의 실행시간, speedup, 페이지 요청 횟수, 임계구역 내에서의 페이지 요청 횟수, 전송된 메시지의 양에 있어 기본 HLRC (orig)와 제안한 프로토콜(new)의 성능 차이를 나타내고 있다.

표 3에서 new는 IS를 제외한 모든 응용 프로그램에서 기대한 대로 페이지 요청 횟수와 전송되는 메시지의 양을 줄임을 알 수 있다. 가장 좋은 성능향상을 보인 $Raytrace_{orig}$는 가장 락을 많이 사용하는 프로그램으로

표 3 실행결과

| Appl. | 8-proc.time | | speedup | | remote getp | | getp in CS | | Msg.amt(MB) | |
|---|---|---|---|---|---|---|---|---|---|---|
| | orig | new | orig | new | orig | new | orig | new | orig | new |
| TSP | 7.19 | 5.83 | 3.47 | 4.28 | 6646 | 4505 | 6047 | 3896 | 27 | 19 |
| Water | 3.19 | 3.00 | 4.06 | 4.32 | 2442 | 2048 | 852 | 464 | 12 | 10 |
| $Raytrace_{orig}$ | 182.15 | 107.88 | 0.31 | 0.53 | 121652 | 18259 | 105309 | 1928 | 526 | 129 |
| $Raytrace_{rest}$ | 10.90 | 10.71 | 5.30 | 5.40 | 8578 | 7424 | 2603 | 1337 | 36 | 31 |
| IS | 4.89 | 4.92 | 1.44 | 1.43 | 4188 | 4188 | 2044 | 2044 | 25 | 25 |

이는 대부분 상태정보를 위한 4byte 정수 값의 갱신을 위한 락 사용이다. new는 이 변수를 접근하기 위한 페이지 요청을 없앰으로써 전체 임계구역 내의 페이지 요청을 대부분 제거하였다.(1053091928. 98% 줄임) 또 페이지 대신 diff가 전송되었기 때문에 메시지의 전송량도 크게 줄었으며(526MB129MB. 75% 줄임) 이것은 40% 이상의 전체 성능향상으로 이어졌다. TSP, Water, $Raytrace_{rest}$도 페이지 부재 횟수와 메시지 양이 줄어들어 성능향상(각각 19%, 6%, 2%)이 있었다. 하지만 Water와 $Raytrace_{rest}$는 줄어든 페이지 요청이 전체 페이지 요청에서 차지하는 비중이 적기 때문에 (각각 16%, 13%) 성능 향상의 폭은 적었다. IS 는 유일하게 성능이 약간 떨어졌는데 이는 임계구역 내에서 생성된 diff의 크기가 커서 new가 diff를 이용한 갱신을 전혀 하지 않기 때문이다. 때문에 orig와 동일하게 동작하며 늘어난 시간은 new를 위한 순수한 오버헤드 즉 홈페이지에 대한 diff생성 및 메모리 관리에 드는 오버헤드다. 그러나 이 오버헤드는 1% 미만으로 매우 작았다.

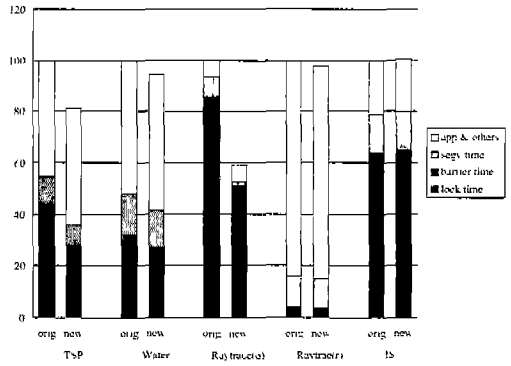


그림 3 실행 시간 분할

그림 3은 orig와 new의 수행시간 분할 그래프이다. 시간 분할은 페이지 부재처리 시간(SEGV 시간), 락 수행 시간, 배리어 수행 시간, 그리고 응용 프로그램 수행 및 기타 시간으로 하였다. 이 그림은 제안한 프로토콜의 주요한 성능향상이 페이지 부재 처리시간의 감소 뿐 아니라 락 시간의 감소로 얻어진다는 것을 보여준다. 앞서 언급했듯이 임계구역 내에서의 페이지 부재 처리시간의 감소는 락에 대한 경쟁이 있을 때 락 시간의 감소로 이어지며 락에 대한 경쟁이 심할수록 그 효과는 프로세스의 수에 비례하여 늘어난다. TSP와 $Raytrace_{orig}$는 모두 락을 주로 사용하는 프로그램으로 이러한 효과로 인해 페이지 부재 처리시간보다 더 큰 락 처리시간의 감소를 얻었다. IS는 diff를 통한 갱신이 일어나지 않기 때문에 락 수행시간만 new의 오버헤드로 인해 약간 늘어났다.

## 5. 관련연구

이동공유 패턴에 적응함으로 락의 성능을 개선하기 위한 다양한 연구[9,10,11]들이 있어왔다. 본 논문에서는 HLRC라는 특정 프로토콜을 기반으로 하고 있으나 본 논문의 몇몇 아이디어는 이들 연구들과 관계가 있다.

Lazy Hybrid[9]는 본 논문의 방법과 같이 락 해제 시 diff를 전송하여 페이지 갱신을 시도한다. 하지만 갱신대상 선택에 있어 한번 이상 접근된 페이지는 무조건 diff를 전송하기 때문에 임계구역 이외의 페이지들에 대한 불필요한 갱신으로 인해 락의 취득 시간이 길어지며 메시지 전송량이 많다. ADSM[10]은 페이지 별로 메모리 접근 패턴을 이동 공유, 생산자-소비자, 거짓공유로 분류하여 각각의 패턴에 대해 다양한 방식으로 적응적으로 동작한다. 이중 이동공유 패턴의 페이지에 대해서는 갱신 방식의 단일 기록자 모드로 전환하여 diff의 생성 및 페이지 부재 발생을 줄인다. 그러나 단일 기록자 모드에서는 페이지 전체가 전달 되므로 작은 데이터를 보호하기 위한 락 사용 시 메시지 전송 부담이 크다. Amza [11]에 의해 제안된 적응 프로토콜도 이와 유사한 방식이지만 변경 부분의 크기가 일정 크기 이상이 되어야만 단일 기록자 모드로 전환하고 그렇지 않은 경우 기존의 무효화 방식의 다중 기록자 프로토콜로 동작

한다. 하지만 무효화 방식에서는 페이지 부재 발생시 원격 노드에 diff 요청을 해야 하므로 지연시간이 문제가 된다. 또 이러한 페이지 단위의 적응 기법은 하나의 페이지 내에 다른 락으로 보호되는 여러 변수들이 있을 경우 사용이 불가능한 문제점을 공통적으로 가지고 있다. 제안한 프로토콜은 diff를 사용하여 갱신을 하며 락 단위로 적응하므로 이와 같은 경우에도 효율적으로 동작할 수 있다.

## 6. 결 론

본 논문에서는 HLRC를 위한 효율적인 락 성능 향상 기법을 제시하였다. 기존의 HLRC에서는 임계구역 내에서 페이지 부재가 발생하면 홈으로부터 페이지를 얻어오는 반면 제안한 프로토콜은 락 허가 시 임계구역에서 사용될 페이지들에 대한 diff를 전달함으로써 페이지 부재시 페이지 요청대신 이미 가지고 있는 diff를 적용하여 새로운 페이지를 구성하도록 하였다. 락을 사용하는 일반적인 응용 프로그램을 이용하여 성능측정을 한 결과 제안한 방법은 전송되는 메시지의 양을 줄이며 갱신으로 인해 임계구역 내의 지연시간을 줄임으로서 성능 향상을 얻음을 확인하였다. 제안한 프로토콜을 위한 오버헤드는 사용한 모든 응용 프로그램에 있어 무시할 수 있을 정도로 작았다.

## 참 고 문 헌


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

[2] P. Kelcher and A. L. Cox and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

[3] Y. Zhou and L. Iftode and K. Li. Perforamnce Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of USENIX OSDI*, October 1996.

[4] H. Li and S. Dvarkadas and A. Cox and W. Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Proceedings of Supercomputing*, December 1995.

[5] K. Gharachorloo and D. Lenoski and P. Gibbons and A. Gupta and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

[6] A. Cox and E. de Lara and Y. Hu and W. Zwaenepoel. A Performance Comparison of Homeless and Home-based Lazy Release Consistency Protocols in Software Shared Memory. In *Proceedings of the fifth High-Performance Computer Architecture*, 1999

[7] S. Woo and M. Ohara and E. Torrie and J. Singh and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, 1995.

[8] D. Jiang and H. Shan and J. Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, 1997.

[9] P. Keleher. Distributed Shared Memory Using Lazy Release Consistency. *PhD thesis, Rice University*, December 1994.

[10] L. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proceedings of the 4th High-Performance Computer Architecture*, Februrary 1998.

[11] C. Amza and A. Cox and S. Dwarkadas and L. Jin and K. Rajamani and W. Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. In *Procedings of the IEEE*, March 1999.



윤 회 철
정보과학회논문지 : 시스템 및 이론
제 28 권 제 9 호 참조

이 상 권
정보과학회논문지 : 시스템 및 이론
제 28 권 제 9 호 참조

이 준 원
정보과학회논문지 : 시스템 및 이론
제 28 권 제 9 호 참조

맹 승 렬
정보과학회논문지 : 시스템 및 이론
제 28 권 제 9 호 참조