

석 사 학 위 논 문

HLRC를 위한 효율적인 락 프로토콜

윤 희 철 (尹 熙 澈)

전자전산학과 (전산학전공)

한국 과학 기술 원

2001

HLRC를 위한 효율적인 락 프로토콜

An Efficient Lock Protocol for Home-based
Lazy Release Consistency

An Efficient Lock Protocol for Home-based Lazy Release Consistency

Advisor : Joon-Won Lee

by

Hee-Chul Yun

Division of Computer Science

Department of Electrical Engineering & Computer Science

Korea Advanced Institute of Science and Technology

A thesis submitted to the faculty of the Korea Advanced Institute of Science and Technology in partial fulfillment of the requirements for the degree of Master of Engineering in the Department of Electrical Engineering & Computer Science

Taejon, Korea

2000. 12. 18

Approved by

Joon-Won Lee

Major Advisor

HLRC를 위한 효율적인 랙 프로토콜

윤 희 철

위 논문은 한국과학기술원 석사 학위논문으로 학위논문심사위원회에서 심사 통과하였음.

2000년 12월 18일

심사위원장 이 준 원 (인)

심사위원 권 용 래 (인)

심사위원 이 윤 준 (인)

MCS 윤 희 철. Hee-Chul Yun. An Efficient Lock Protocol for Home-
993355 based Lazy Release Consistency . HLRC를 위한 효율적인 락 프
 로토콜. Department of Computer Science. 2001. 26p. Advisor:
 Joon-Won Lee.

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. In these applications, the number of page faults inside critical sections is reduced by 56% on the average and by up to 98% compared with base HLRC.

사랑하는 부모님에게 이 글을 바칩니다.

목 차

제 1 장 서론	1
제 2 장 배경지식	4
제 1 절 LRC	4
제 2 절 HLRC	6
제 3 절 HLRC에서 락 사용의 문제점	7
제 3 장 HLRC를 위한 향상된 락 프로토콜	9
제 1 절 프로토콜	9
제 2 절 장점과 단점	10
제 3 절 홈 페이지에 대한 diff생성	12
제 4 절 메모리 오버헤드	12
제 5 절 알고리즘	13
5.1 락 요청	13
5.2 락 해제	14
5.3 락 허가	14
5.4 페이지부재 처리	15
제 4 장 성능 측정	16
제 1 절 플랫폼	16
제 2 절 응용프로그램	16
제 3 절 성능 측정 결과	18
3.1 성능향상(speedup)	19
3.2 임계구역 내의 원격 페이지 요청	20
3.3 메시지 전송량	21
3.4 실행시간 분할	22

제 5 장	관련연구	24
제 1 절	Lazy Hybrid(LH) [12]	24
제 2 절	ADSM [13]	25
제 3 절	Amza의 프로토콜 [14]	25
제 6 장	결론	26
참고 문헌		27

제 1 장

서론

최근 고성능 마이크로 프로세서와 고속 네트워크의 등장으로 인해서 NOW (Networks Of Workstations)와 같은 클러스터 시스템을 병렬 처리에 사용하고 자 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 대표적인 프로토콜이다. 이 둘은 모두 다중 기록자 프로토콜을 채용하는데 이것은 twin과 diff를 이용해 공유메모리상의 변경내용을 찾아내며 쓰기 가능한 페이지가 동시에 여러개 존재할 수 있다. LRC와 HLRC의 차이는 메모리 구조와 diff의 전달 및 관리방법 다르다는 점이다. LRC에서 모든 공유 페이지는 캐쉬 처럼 취급된다. diff의 생성 및 전달은 다른 프로세스가 diff를 요청할때에 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 무효화 되지

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 락의 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 페이지의 갱신을 diff를 전송하여 이루기 때문에 페이지 전체를 얻어오는 것에 비해 메시지 양을 크게 줄인다.

이 프로토콜은 KDSM(Kaist Distributed Shared Memory) [5] HLRC 버전상에서 구현되었다. KDSM의 HLRC 프로토콜은 Princeton의 HLRC [3] 프로토콜

을 구현한 것이다. 성능측정은 Linux 운영체제를 사용하며 100Mbps switched fast ethernet으로 연결된 8대의 P-III 500Mhz cluster상에서 5개의 응용 프로그램을 수행하여 성능을 측정하였다. 성능측정 결과 제안한 프로토콜은 5개의 응용 프로그램 중 4개에서 기본 HLRC 프로토콜에 비해 2% ~ 60%의 성능향상을 보였으며 나머지 1개에서도 성능의 감소가 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.

제 2 장

배경지식

본 장에서는 SDSM 에 있어 대표적인 두 프로토콜인 LRC와 HLRC에 대해 설명하고 이들의 장단점을 살펴본다. 그후 본 논문에서 초점으로 하고 있는 HLRC가 왜 락의 사용에 있어 나쁜 성능을 보이는지를 설명한다.

제 1 절 LRC

Lazy Release Consistency (LRC) [2] 는 Release Consistency (RC) [6] 메모리 모델의 한 구현이다. RC는 동기화 시점에서만 메모리 일관성을 보장해주는 메모리 모델이다. 동기화 연산은 acquire, release, barrier 등이 있으며 프로그램 내에서 이들을 적절하게 사용하여 동기화를 보장한다. 적극적 방식의 (Eager Release Consistency) 프로토콜에서는 공유 메모리에 대한 모든 변경 내용이 각 release 연산마다 시스템 전체에 알려지게 된다. 반면 LRC는 이러한 변경 내용을 다음 락 요청자에게만 알려줌으로서 불필요한 일관성 정보의 전달을 줄이고 있다.

LRC에서 프로그램의 수행은 *interval*로 구분된다. 이들 *interval* 간에는 *happen-before* 관계를 유지시킴으로서 RC [6] 메모리 모델을 만족시킨다. 이를 위해 각 프로세스는 자신이 알고 있는 다른 프로세스들의 시간에 대한 정보를 나타내는 *vector timestamp* 를 유지한다. 쓰기-쓰기 거짓 공유 (write-write false sharing) 를 줄이기 위해 LRC는 다중 기록자 기법과 함께 사용된다. 소프트웨어적인 다중 기록자 (multiple-writer) 방식은 TreadMarks 시스템 [7] 에서 처음 사용되

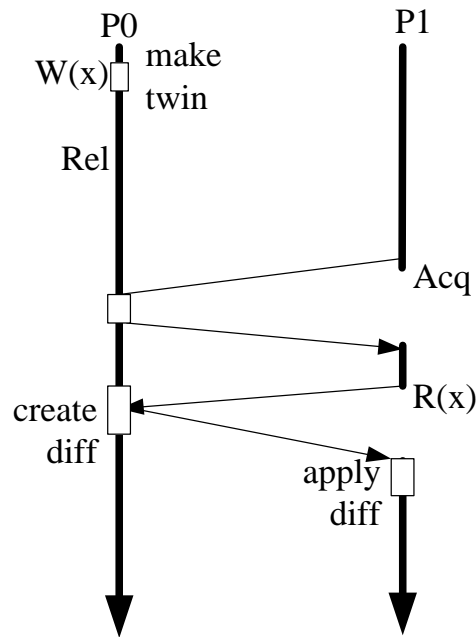


그림 2.1: LRC에서의 락의 동작

었다. TreadMarks 시스템에서 모든 기록자 (writer) 는 공유 메모리에 대한 매 인터벌 마다의 변경내용을 기록한다. 어떤 페이지에 대해 최초로 쓰기가 발생했을 때 실제 쓰기가 일어나기 전의 페이지를 저장하는데 이것을 *twin* 이라고 한다. Release시에 프로세스는 현재의 페이지와 앞서 저장하였던 *twin* 페이지를 비교하여 변경내용을 찾아 저장하며 이를 *diff* 라 한다. 이 *diff* 는 매 인터벌 끝에 적극적으로 생성할 수도 있지만 *diff* 의 요청이 있을 때에 생성하도록 더욱 늦출 수 있다. 락을 획득한 프로세스는 락 허가 메시지에 포함된 (piggyback) 변경된 페이지에 대한 정보인 *write-notice* 를 참조하여 해당 페이지를 무효화시킨다. 무효화 된 페이지에 대한 접근으로 인해 페이지 부재가 발생하면 페이지 부재 처리기는 (page fault handler) *write-notice* 정보를 통해 해당 페이지에 대한 모든 기록자에게 *diff* 를 요청하며 전송받은 *diff* 들을 순서 (casual order) 에 맞게 적용하여 새로운 페이지를 구성한다. 그림 2.1 는 LRC에서의 락의 동작을 나타내고 있다.

제 2 절 HLRC

HLRC는 Princeton Univ [3]에서 제안된 페이지 기반의 다중 기록자 프로토콜로 앞서 설명한 LRC와 유사하나 변경내용이 전달되는 방법에 있어 차이가 있다. HLRC에서 모든 공유 메모리 페이지는 항상 최신의 내용을 가지는 지정된 홈 노드가 있다. 이를 위해 홈이 아닌 노드는 인터벌 끝에서 홈으로 diff를 전송하며 홈은 이것을 즉시 적용한다. 페이지 부재가 발생하면 해당 프로세스는 단순히 이 홈으로부터 페이지를 얻어옴으로서 최신의 복사본을 얻는다. 그림 2.2는 이러한 HLRC의 동작을 보여주고 있다.

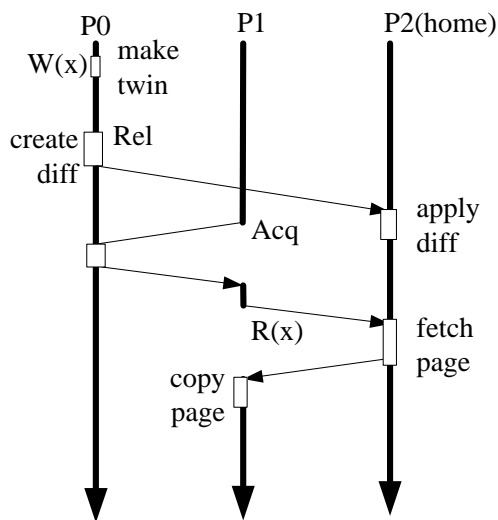


그림 2.2: HLRC에서의 락의 동작

HLRC는 LRC에 비해 다음과 같은 장점이 있다. [8] (1)페이지 부재가 발생하였을 때 홈 노드로 한번의 왕복 메시지만이 필요하다. (2)홈 페이지의 대한 쓰기는 diff및 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 할당

되어 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.

제 3 절 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 두가지 문제가 모두 나타나게 된다. 첫째 홈 할당에 있어 이동공유 패턴의 특성상 적절한 홈할당을 하는 것이 불가능하므로 락의 흐름과 상관없이 매 release마다 홈으로 diff를 전송해야 한다. 둘째 페이지 부재 발생시 항상 페이지 전체를 얻어오지만 실제로 필요한 부분은 매우 작다.

그림 2.3 은 HLRC 에서 락사용시 발생하는 프로토콜의 동작 예로, P2 (홈노드) 는 실제 락의 전달과 상관없음에도 diff의 처리및 페이지 요청에 대한 처리를 해야 하며 P1은 실제 필요한 데이터는 4byte크기지만 4Kbyte의 페이지를 전달받아야 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.

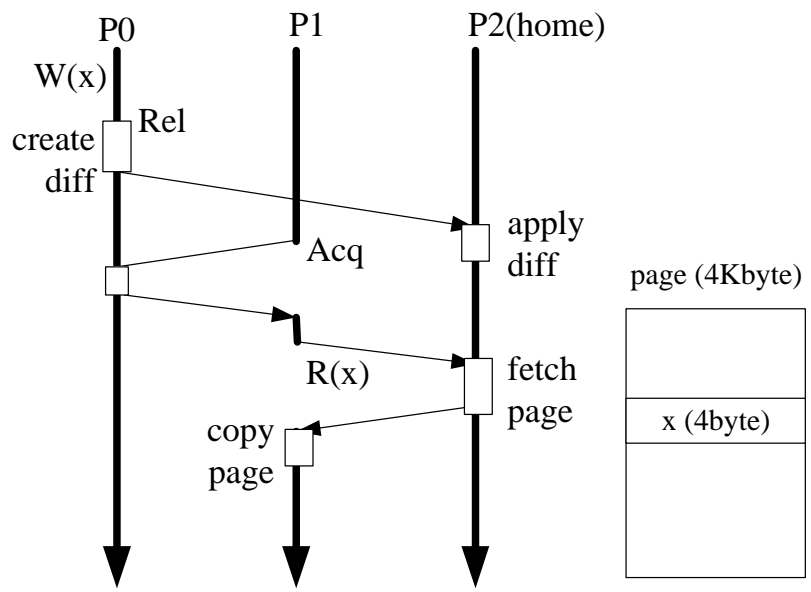


그림 2.3: 비효율적인 HLRC의 락 동작

제 3 장

HLRC를 위한 향상된 락 프로토콜

제 1 절 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 전송받은 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 락요청, 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 접근 기록을 바탕으로 예측한다. 락의 경우 임계구역 내의 메모리 접근 패턴이 매우 일정하므로 이전 기록을 참조하는 것으로도 좋은 예측이 가능하다.

- 락 허가

락 허가자는 락 요청 메시지에 포함된 페이지 목록중 실제로 diff를 전달할 페이지를 선택한다. 선택의 기준은 해당 페이지를 갱신하기 위해 전달해야 할 diff의 크기의 합이 페이지 크기보다 작아야 한다는 것이다. 이것은 diff 누적현상 [8]으로 인한 문제를 방지하기 위함이다. diff 누적현상이란 하나의 페이지를 갱신하기 위해 전달되어야 할 diff가 여러개가 되는 현상을 말하며 누적된 diff의 크기의 합이 페이지 크기보다 커질경우 diff를 전달하는 것이 오히려 페이지 전체를 전달하는 것보다 커다란 메시지를 발생시켜 성능의 저하를 가져오게 된다. 선택된 페이지들에 대한 diff는 write-notice 와 함께 락 허가 메시지에 포함 (piggyback) 하여 전송한다.

- 페이지 부재 처리

전달된 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 3.1는 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 허가시 y에 대한 diff들을 전송하지 않고 P2는 y에 대해 기본 HLRC 프로토콜에서처럼 홈으로부터 페이지를 얻어온다.

제 2 절 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 것

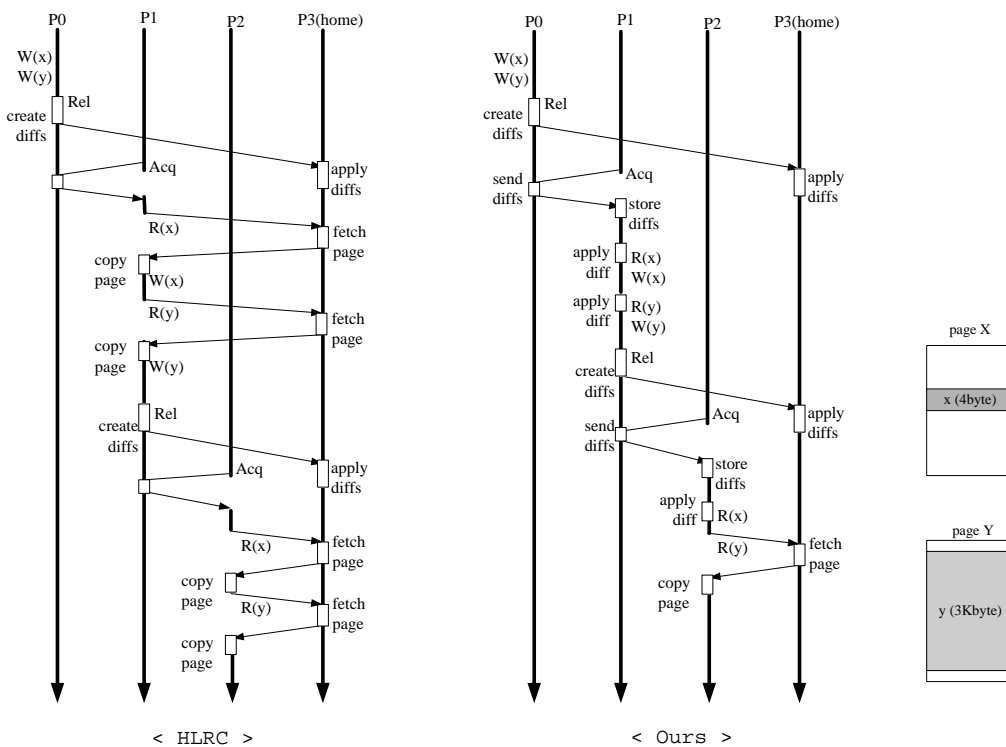


그림 3.1: 락의 동작 예

만으로 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 적은 크기를 가진다. 따라서 전달되는 메시지의 양은 기본 HLRC 프로토콜에 비해 줄어든다. 또한 누적된 diff의 크기를 페이지 크기로 제한하기 때문에 diff의 누적으로 인해 메시지양이 늘어나는 현상은 일어나지 않는다.

셋째, 추가적인 메시지가 발생하지 않는다. 제안한 프로토콜을 위한 모든 정보는 기존의 메시지에 덧붙여서 보내어질 수 있다. 따라서 전달되는 메시지

의 갯수는 기본 HLRC 프로토콜과 동일하다.

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

제 3 절 홈 페이지에 대한 diff생성

제안한 프로토콜이 효율적으로 동작하기 위해서는 홈페이지에 대해서도 diff를 생성해야 한다. 하지만 그 대상이 임계구역 내에서 접근되는 페이지만으로 한정되므로 그 수는 매우 적다. 임계구역 밖에서 접근되는 페이지들은 기본 HLRC 처럼 홈에서 diff를 생성하지 않는다. 임계구역 내의 페이지여서 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) 이와같은 제한은 성능에 좋은 영향을 끼친다. 현재의 구현에서 이 값은 256Byte 로 하였다. 이상의 방법으로 홈 페이지의 diff 생성으로 인한 부하를 최소화 하였으며 성능 측정 결과 거의 무시할 수 있는 시간만이 걸리는 것을 확인하였다.

제 4 절 메모리 오버헤드

제안한 프로토콜은 LRC처럼 diff를 메모리에 유지할 필요가 있다. 하지만 홈페이지에 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.

제 5 절 알고리즘

본 프로토콜은 HLRC를 기반으로 하고 있기 때문에 많은 부분이 기존의 HLRC 프로토콜과 유사하다. 본 논문에서 기반하고 있는 HLRC 프로토콜의 설계 및 구현에 대해서는 [5]에 상세하게 기술되어 있다. 이 장에서 다루는 내용은 대부분의 경우 기존의 HLRC와 다른 부분에 대한 것이므로 여기서 다루지 않은 부분에 대해서는 [5]을 참조하기 바란다.

5.1 락 요청

락 요청의 전달은 일반적으로 널리쓰이는 방식인 락 매니저를 이용한 큐-기반(queue-based) 락 방식을 사용한다. 각 락은 round-robin으로 선출된 정적인 관리자를 가지고 있으며 락요청은 이 관리자를 통해서 일어난다. 락 요청을 받으면 락 관리자는 최후로 락을 요청했던 프로세스에게 이 요청을 전달(forwarding) 한다. 락 요청을 받은 프로세스는 자신이 락을 사용(held)하고 있지 않으면 즉시 락을 허가하고 그렇지 않으면 락 변수의 next 필드에 요청한 프로세스를 기록함으로써 분산된 큐를 구성한다.

procedure Requesting of lock L

```
create new interval
if ( L is local )
    set L as held
else
    send vector timestamp and update list
    wait for reply
    // now L is local and held
    store early diffs
    invalidates each page p in write notices
```

그림 3.2: 락 요청

본 프로토콜에서는 기존의 HLRC와 달리 임계구역에서의 메모리 접근 기록에 대한 정보를 추가적으로 전달한다. 임계구역에서의 메모리 접근 기록은 각 락 변수마다 락 해제시에 수정된다. 그림 3.2는 본 프로토콜에서의 락 요청 가

상코드를 나타내고 있다. 락 요청 메시지에는 *vector timestamp*와 *update list*가 포함된다. *update list*는 락 변수에 저장되어 있는 임계구역에서의 메모리 접근 기록을 나타낸다. 락 허가를 받으면 프로세스는 *early diffs* 와 *write notices*를 전달받게 된다. *early diffs*는 *update list*를 통해 요청된 페이지들을 갱신하기 위해 전달받은 diff들이다. *early diffs*의 각 diff들에 대한 포인터는 해당 페이지의 페이지 구조체에 연결되어 후에 페이지 부재 처리시 diff 에 대한 접근이 용이하도록 한다.

5.2 락 해제

락 해제시 임계구역 내에서 접근되었던 페이지 목록을 기록하여 락 구조체에 저장한다. 이 정보는 후에 *update list* 로서 락 요청시 이용된다.

5.3 락 허가

락 허가가 일어나는 시점은 락 해제시 락을 기다리고 있는 프로세스가 있거나 락을 가지고 있으나 사용하지 않는 상태에서 다른 프로세스가 락을 요청한 경우이다. 락 허가시에는 우선 자신의 벡터 타임스탬프를 통해 전송할 *write notices*를 결정한다. 이에 덧붙여 본 프로토콜에서는 갱신을 위한 diff를 전송한다.

procedure Granting of lock *L* to process *Q* from process *P*

```

set L as not local
make write notices from P's vector timestamp
calculate diffsize of each page p in write notices
foreach page p in update list
    if ( diffsize < PAGESIZE and homepid != Q )
        add diffs of page p to early diffs
send early diffs
send write notices

```

그림 3.3: 락 허가

그림 3.3에서와 같이, 본 프로토콜에서는 락 요청을 통해 전송받은 *update*

*list*와 이 리스트에 포함되어 있는 페이지이면서 이를 갱신하기 위한 diff들의 크기가 페이지 크기보다 작으며 락을 전송받을 프로세스(Q)가 페이지의 홈이 아닌 경우 이들 diff들을 *early diffs*에 포함시킨다. 페이지의 홈이 Q 일때 diff를 포함하지 않는 이유는 본 프로토콜이 기본 HLRC와 같이 dirty page를 처리하는 과정에서 이미 홈으로 diff를 전송하였기 때문이다.

5.4 페이지부재 처리

캐쉬 페이지에 대한 부재 발생시 기본 HLRC에서는 항상 홈으로 페이지 요청을 했으나 본 프로토콜에서는 페이지를 갱신하기 위한 모든 diff를 미리 전달 받을 수 있으므로 이 경우 이미 가지고 있는 diff들을 메모리에 적용함으로써 페이지를 갱신한다. 그림 3.4는 이러한 캐쉬페이지에 대한 페이지 부재 처리 과정을 나타내고 있다.

```
procedure page fault handling for cache page  $p$ 
.
.
if ( all diffs are ready to make up-to-date page  $p$  )
  apply diffs
else
  get remote page  $p$  from the home
.
.
.
```

그림 3.4: 페이지 부재 처리

제 4 장

성능 측정

제 1 절 플랫폼

성능측정은 switched 100Mbps ethernet으로 연결된 8노드의 500Mhz PIII 클러스터에서 행하였다. 각 노드는 256 Mbyte의 메모리를 가지며 Linux 2.2.13을 운영체제로 사용하였다. 또 통신계층으로는 TCP/IP를 사용하였다. 이 시스템에서의 KDSM의 기본 작동에 대한 오버헤드는 그림 4.1과 같다.

Operation	Cost (μ s)
fetch a 4KB page	1047
lock acquire	259
barrier (8 processors)	1132

표 4.1: KDSM의 기본 동작 성능

제 2 절 응용프로그램

성능측정은 락을 사용하는 응용 프로그램인 TSP, Water, IS, Raytrace를 이용하였다. TSP, Water, IS는 cvm [9] 배포판에 포함된 것을 이용하였고 Raytrace는 SPLASH2 [10] 에서 포팅한 Raytrace_{orig}와 [11]에서 지적한 대로 상태정보를 위한 락을 제거한 버전인 Raytrace_{rest} 버전을 사용하였다. 각 응용 프로그램에 대

해 간략히 기술하면 다음과 같다.

- TSP

분기-경계 알고리즘을 이용하여 여러 도시들을 최소의 비용으로 한번씩만 방문하며 마지막엔 다시 출발하였던 곳으로 돌아올 수 있는 경로를 구하는 프로그램이다. 동기화 수단으로서 락만을 사용하며 쓰기 크기는 매우 작다.

- Water

물 분자계의 시간에 따른 분자간의 힘과 에너지를 구하는 프로그램이다. 힘과 에너지는 매 시간 단위마다 계산된다. 락의 사용이 빈번한 것이 아니며 락보다 배리어에 의한 동기화가 더 많은 비중을 차지한다.

- Raytrace

3차원 이미지를 빛 추적(ray tracing) 기법을 사용하여 렌더링하는 응용 프로그램이다. octree와 유사한 계층적 단일 그리드가 이미지를 표현하기 위해 사용되었다. 2가지 버전이 사용되었는데 원래의 버전은 4바이트 상태정보를 위한 락을 매 빛 추적마다 사용한 것이고 다른 버전은 이 락을 제거한 버전이다. 전자의 경우는 락의 사용이 매우 많지만 후자의 경우는 락의 사용이 그다지 빈번하지 않다.

- IS

버킷정렬방식을 이용하여 키값을 정렬하는 정렬 프로그램이다. 키들은 프로세스별로 분할되어 있다. 각 프로세스는 자신의 버킷에 있는 키의 숫자를 센다. 다음 단계에서 버킷의 키값들은 합산된다. 공유 버킷은 락을 통해 보호되며 버킷의 내용은 락을 획득한 프로세스에 의해 완전하게 덮어쓰여지게 된다. 버킷의 크기가 크기 때문에 일반적으로 페이지 전체가 쓰여진다.

표 4.2 은 사용된 문제 크기, 사용된 락 과 배리어의 수, 그리고 단일 프로세스일때의 실행시간을 보여준다.

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 4.2: 응용 프로그램의 특징

제 3 절 성능 측정 결과

성능측정은 원래의 HLRC 프로토콜 (*orig*)와 제안한 프로토콜 (*new*) 을 비교한다. 8프로세스에서의 수행하였을때의 성능향상, 임계구역 내에서의 페이지 요청 횟수, 메시지 전송량의 차이를 비교하였고 마지막으로 실행시간 분할을 비교하였다.

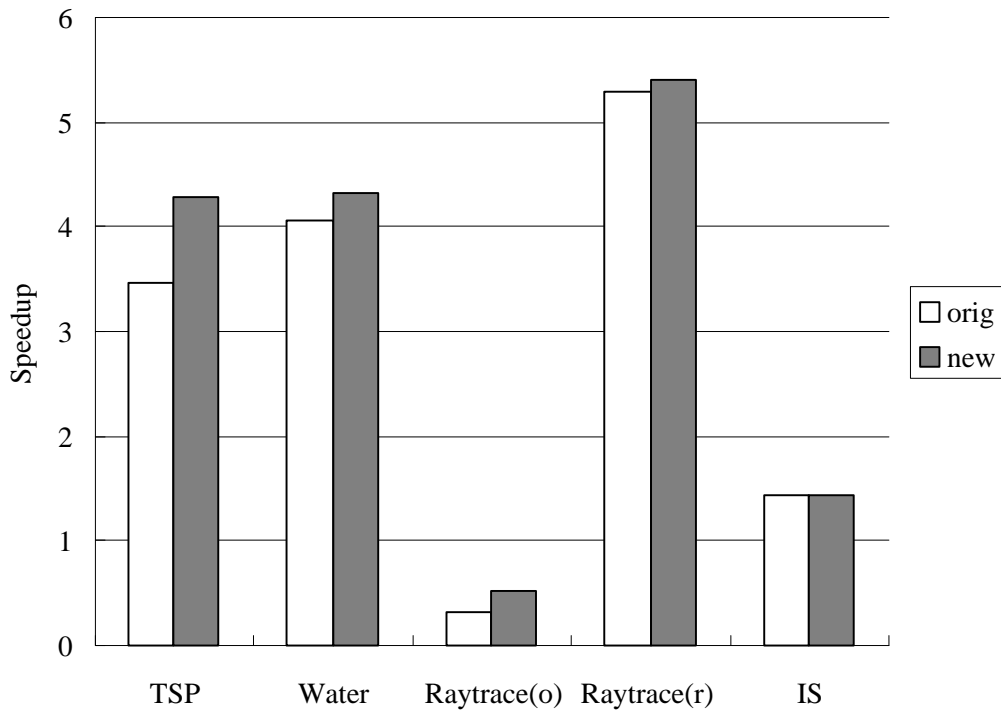


그림 4.1: 성능향상(speedup)

3.1 성능향상(speedup)

그림 4.1은 8프로세스에서 *orig*와 *new*를 이용하여 수행한 각 응용 프로그램의 성능향상 그래프이다. IS를 제외한 모든 응용 프로그램에 있어 *new*는 기대한 대로 *orig*에 비해 성능이 향상되었다. IS의 경우에는 락을 주로 사용함에도 불구하고 *new*로 인한 성능 개선이 전혀 없었는데 이것은 IS 임계구역 내에서의 쓰기 크기가 매우 크기 때문이다. 앞서 설명한 바와 같이 쓰기 크기가 클 경우 *new*에서는 diff의 누적현상으로 인한 성능저하를 막기 위해 diff를 전달하지 않으며 *orig*와 동일하게 동작한다.

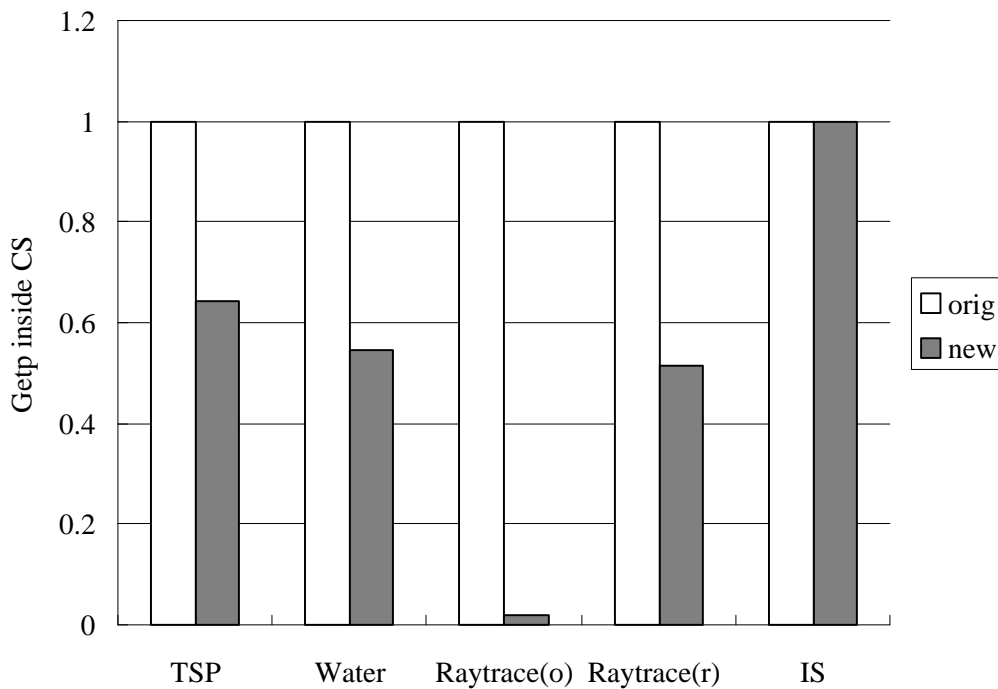


그림 4.2: 임계구역 내의 원격 페이지 요청 횟수

3.2 임계구역 내의 원격 페이지 요청

그림 4.2는 임계구역내에서의 페이지 요청 횟수의 차이를 보여준다. 페이지 요청 횟수는 *orig*의 것을 기준으로 나타내었다. 이 그림은 *new*의 특징을 가장 잘 표현하는 것으로서 기대한 대로 갱신을 통해 임계구역 내의 원격 페이지 요청을 효율적으로 줄이고 있음을 확인할 수 있다. IS의 경우는 앞서 언급했듯이 *new*가 *orig*와 동일하게 동작하기 때문에 페이지 요청횟수의 차이가 없다. $Raytrace_{orig}$ 는 가장 많은 페이지 요청의 감소가 일어났는데 이것은 $Raytrace_{orig}$ 에서 사용하는 대부분의 락이 4바이트의 상태 정보를 갱신하기 위한 것이기 때문에 *new*에서는 갱신을 통해 이를 위한 모든 페이지 요청을 없앨 수 있었기 때문이다.

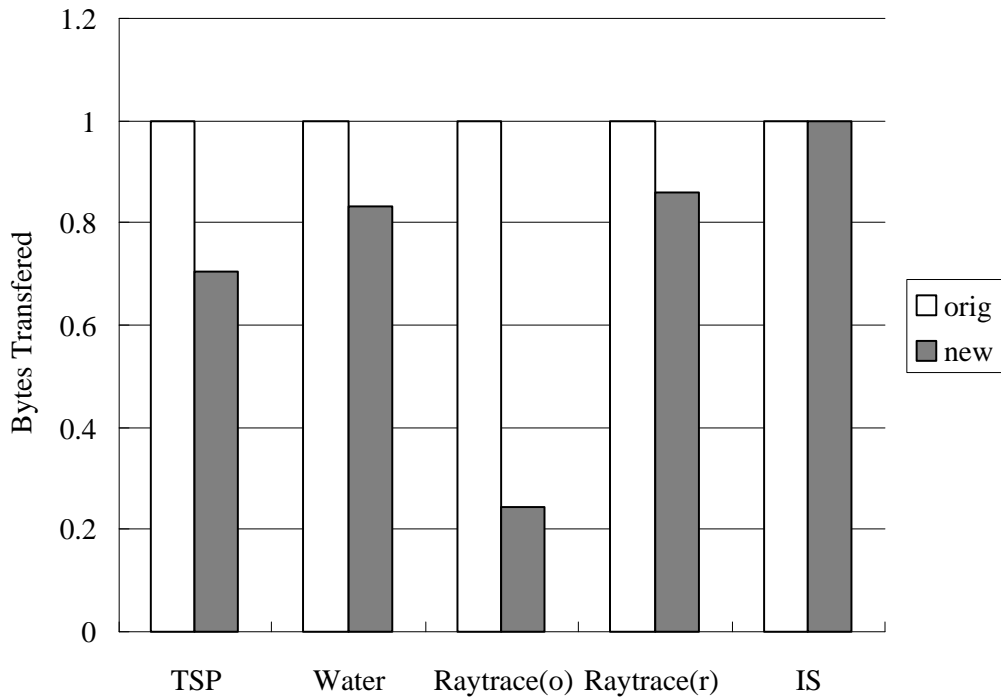


그림 4.3: 메시지 전송량

3.3 메시지 전송량

그림 4.3는 전체 메시지 전송량의 차이를 보여준다. 기대한 대로 IS를 제외하고는 대부분의 경우에 있어 임계구역내의 쓰기 크기가 작기 때문에 페이지 대신 diff가 전달됨으로서 메시지의 양이 줄어드는 것을 알 수 있다.

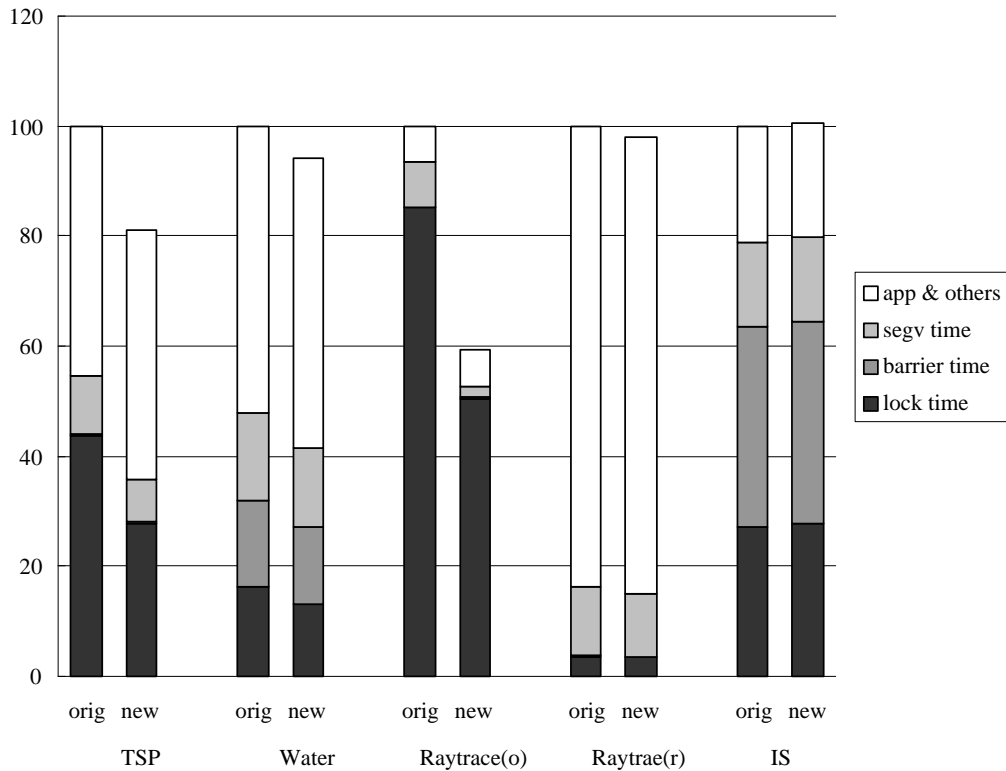


그림 4.4: 실행 시간 분할

3.4 실행시간 분할

그림 4.4은 *orig*와 *new*에서의 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 있을때 락 시간의 감소로 이어지며 락에 대한 경쟁이 심할수록 그 효과는 프로세스의 수에 비례하여 늘어난다. TSP와 $Raytrace_{orig}$ 는 모두 락을 주로 사용하는 프로그램으로 이같은 효과로 인해 페이지 부재 처리시간보다 더 큰 락 처리시간의 감소를 얻었다. IS는 diff를 통한

갱신이 일어나지 않기 때문에 락 수행시간만 *new*의 오버헤드로 인해 약간 늘어났다. 하지만 이 오버헤드는 매우 작았다.

제 5 장

관련연구

본 장에서는 lock의 성능을 향상시키기 위해 제안된 혹은 이와 관련이 있는 다양한 연구들에 대해 소개한다. 본 논문에서는 HLRC라는 특정 프로토콜에 대해 초점이 맞추어 있지만 이러한 연구들과도 많은 연관이 있으므로 이들을 설명함과 동시에 제안한 프로토콜과의 정성적으로 비교를 해 본다.

제 1 절 Lazy Hybrid(LH) [12]

LH는 diff를 이용하여 갱신을 한다는 점에서 본 논문에서 제안한 프로토콜과 유사하다. 하지만 기본적으로 LRC에 기반하고 있으며 갱신 대상의 선택방법이 매우 다르다. LH에서는 각 프로세스는 페이지 별로 근사복사집합 (approximate copyset) 이라는 정보를 통해 갱신 대상을 결정한다. 이것은 해당 페이지에 대해 다른 프로세스가 요청한 적이 있는지의 여부를 나타내는 것으로 한번 등록 되면 지워지지 않는다. 락 해제시 write-notice를 전달하는데 이때 락을 획득하는 프로세스가 해당 페이지의 근사 복사집합에 포함되어 있는 경우 diff를 전달하여 갱신한다. 이 근사복사집합은 락의 사용과 무관하게 관리되므로 적절한 부적절한 갱신이 일어날 가능성이 크다. 따라서 갱신으로 인한 이득은 매우 제한적이며 프로그램에 따라 매우 심각한 성능저하가 생길 수 있다.

제 2 절 ADSM [13]

ADSM은 LRC에 기반한 적응 프로토콜로 페이지 별로 공유되는 패턴을 분류하여 각 패턴별로 적절하게 적응을 시도한다. 이러한 패턴중 락의 사용과 관련이 깊은 이동 공유 패턴에 대해서는 단일 기록자 모드로 전환하며 동시에 쓰기 전달 방법을 갱신방식으로 전환한다. 따라서 만약 어떤 페이지가 임계구역으로 보호되고 있을 때 ADSM은 이를 단일 기록자 모드로 전환하여 페이지에 대한 twin 및 diff의 생성을 막으며 락 해제시에는 락을 획득하는 프로세스에게 해당 페이지를 전달하여 갱신을 하도록 한다. 이 방법의 가장 큰 단점은 단일 기록자 모드로 동작할 경우 반드시 페이지 전체가 전달되기 때문에 페이지에 대한 쓰기 크기가 작은 경우 (락으로 보호되는 영역은 쓰기 크기가 일반적으로 작다) diff를 전달하는 것에 비해 메시지 부담이 크다는 것이다. 때문에 메시지의 부하로 인해 갱신의 효과가 제한적일 수 있다. 또한 하나의 페이지에 여러 개의 다른 락으로 보호되는 메모리 영역이 할당되어 있다면 이들은 이동 공유 패턴이 아닌 것으로 판단하므로 적절하게 대처를 하지 못한다. 반면 본 논문에서 제안하는 프로토콜은 이러한 경우에도 diff만 전달하면 되기 때문에 갱신이 가능하다.

제 3 절 Amza의 프로토콜 [14]

Amza에 의해 제안된 프로토콜도 ADSM 과 유사하나 쓰기 크기에 대한 적응을 하는 점이 다르다. 이 프로토콜에서는 쓰기 크기가 큰 경우에만 ADSM과 같은 방법으로 단일 기록자 모드 및 갱신방식으로의 전환이 일어나며 그렇지 않은 경우 기존의 무효화 방식의 다중 기록자 모드로 동작하도록 함으로서 메시지 부하를 줄이고 있다. 그러나 무효화 방식으로 동작할 경우 페이지 부재가 발생하게 되므로 임계구역 내의 지연시간이 늘어나게 된다.

제 6 장

결론

본 논문에서는 HLRC를 위한 효율적인 락 프로토콜을 제시하였다. 기존의 HLRC에서는 임계구역 내에서 페이지 부재가 발생하면 홈으로 페이지를 얻어오는 반면 제안한 프로토콜은 락 허가시 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 성능향상을 얻음을 확인하였다. 제안한 프로토콜을 위한 오버헤드는 사용한 모든 응용 프로그램에 있어 무시할 수 있을 정도로 작았다.

참고 문헌

- [1] P. Keleher, *Distributed Shared Memory Using Lazy Release Consistency*. PhD thesis, Rice University, December 1994.
- [2] L. Monnerat and R. Bianchini, “Efficiently adapting to sharing patterns in software dsms,” 1998.
- [3] C. Amza, A. Cox, S. Dwarkadas, L. Jin, K. Rajamani, and W. Zwaenepoel, “Adaptive protocols for software distributed shared memory,” 1999.
- [4] K. Li and P. Hudak, “Memory coherence in shared virtual memory systems,” in *Proc. 5th Annual ACM Symp. Principles of Distributed Computing*, pp. 229–239, Aug 1986.
- [5] P. Keleher, A. L. Cox, and W. Zwaenepoel, “Lazy Release Consistency for Software Distributed Shared Memory,” in *Proceedings of ISCA*, 1992.
- [6] Y. Zhou, L. Iftode, and K. Li, “Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems,” in *Proceedings of USENIX OSDI*, October 1996.
- [7] H. Li, S. Dwarkadas, A. Cox, and W. Zwaenepoel, “Message passing versus distributed shared memory on networks of workstations,” in *In Proceedings of Supercomputing '95*, December 1995.
- [8] . 이상권, “소프트웨어 분산 공유메모리 시스템의 설계 및 구현,” tech. rep., 한국과학기술원, 2000.
- [9] K. Gharachorloo, D. Lenoski, P. Gibbons, A. Gupta, and J. Hennessy, “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors,” in *Proceedings of ISCA*, 1990.

- [10] C. Amza, S. Dwarkadas, P. Keleher, A. Cox, and Z. Zwaenepoel, "Treadmarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, vol. 29, February 1996.
- [11] A. Cox, E. de Lara, Y. Hu, and W. Zwaenepoel, "A performance comparison of homeless and home-based lazy release consistency protocols in software shared memory," in *Fifth International Symposium on High Performance Computer Architecture*, Feb 1999.
- [12] P. Keleher, "CVM: The Coherent Virtual Machine," tech. rep., University of Maryland, Department of Computer Science, 1996.
- [13] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of ISCA*, 1995.
- [14] D. Jiang, H. Shan, and J. Singh, "Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors," in *6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 217–229, 1997.

감사의 말

본 논문이 완성되기까지 지도해주시고, 저의 진로에 대해서도 세심한 부분까지 신경을 너무나 많이 써주신 이준원 교수님께 깊은 감사를 드립니다. 또한 석사 생활에서 많은 관심을 가져주신 조정완 교수님, 맹승렬 교수님, 윤현수 교수님께도 감사를 드립니다. 심사를 맡아주시고, 좋은 조언을 아끼지 않으신 권용래 교수님, 이윤준 교수님께 감사드립니다.

이 논문을 위해 관심을 갖고 지도해 주신 많은 실험실 학형들에게 감사를 드립니다. 논문에 대한 지도와 조언을 아끼셨으며 논문 이외에 여러면에서 모범이 되어주신 상권형께 진심으로 감사드립니다. 저의 질문에 언제나 친절하게 답변해 주셨으며 논문에 대한 날카로운 지적과 조언을 해주셨던 규환이형, 영철이형에게도 진심으로 감사드립니다. 논문을 손봐주시고 생활에 모범을 보여주셨던 병찬이형과 민희누나 에게 감사드립니다. 여러방면으로 저의 랩생활을 즐겁고 풍요롭게 해주셨던 영배형, 승택이형, 호중이형, 소연이누나에게 진심으로 감사드립니다. 2년간 즐거움과 노동을 함께 나눈 실험실 동기들, 원진이형, 병천이형, 우성이형, 선영, 인철, 진성, 정록이에게도 고마움을 전합니다. 각자 흩어져 자신의 길을 가겠지만 사회에 나가서도 서로 만나 가끔 술한잔을 기울일 수 있기를 바랍니다. 비록 많은 시간을 나누지는 못했지만 진원이를 비롯한 석1들에게도 고마움을 전하며 내년에 좋은 논문 쓰기를 바랍니다.

마지막으로 언제나 저를 믿고, 큰 힘을 주시는 아버지,어머니에게 한없는 감사를 드립니다.

이 력 서

성 명 : 윤 희 철

생 년 월 일 : 1977년 11월 22일

출 생 지 : 원주

본 적 : 강원도 원주시 명륜동 453번지

학 력

1995.3-1999.2 한국과학기술원 전산학과 졸업 (B.S.)

1999.3-2001.2 한국과학기술원 전산학과 졸업 (M.S.)