분산 공유 메모리 시스템에서 메모리 참조 패턴에 근거한 거짓 공유 감소 기법

조 성 제†

요 약

분산 공유 메모리 시스템에서 거짓 공유는 두 개의 다른 처리기에 의해 접근되는 두 개의 서로 다른 데이터가 동일한 블록에 위치하기 때문에 발생하며, 시스템 성능 저하의 큰 요인으로 작용한다. 본 논문에서는 먼저 공유 메모리가 동적으로 할당되는 병렬 프로그램들을 대상으로 공유 데이터 객체의 할당 및 접근 패턴을 분석하였다. 공유 객체들은 순차적으로 할당되며 서로 다른 참조 패턴을 보일 때가 많다. 동일 크기의 객체들이 처리기 수만큼 연속적으로 요청되는 경우에는 각 객체는 특정 처리기에 의해서만 참조되며, 동일 크기의 객체들이 처리기의 수보다 훨씬 더 많이 연속적으로 할당되는 경우에는 연속적인 2개 이상의 객체가 특정 처리기에 의해서만 참조된다. 이를 근거로, 참조 패턴이 다른 객체들을 서로 분리하여 서로 다른 페이지에 배치함으로써 거짓 공유를 줄일 수 있는 메모리 할당 방식을 제시하고, 기존의 거짓 공유 제거 방식들과 비교 평가한다. 제시한 기법은 익간의 메모리 공간을 추가로 요구하며 몇몇 응용에 대해서는 거짓 공유 폴트의 수를 많이 감소시켜 준다.

Reducing False Sharing based on Memory Reference Patterns in Distributed Shared Memory Systems

Seong-Je Cho†

ABSTRACT

In Distributed Shared Memory systems, false sharing occurs when two different data items, not shared but accessed by two different processors, are allocated to a single block and is an important factor in degrading system performance. The paper first analyzes shared memory allocation and reference patterns in parallel applications that allocate memory for shared data objects using a dynamic memory allocator. The shared objects are sequentially allocated and generally show different reference patterns. If the objects with the same size are requested successively as many times as the number of processors, each object is referenced by only a particular processor. If the objects with the same size are requested successively much more than the number of processors, two or more successive objects are referenced by only particular processors. On the basis of these analyses, we propose a memory allocation scheme which allocates each object requested by different processors to different pages and evaluate the existing memory allocation techniques for reducing false sharing faults. Our allocation scheme reduces a considerable amount of false sharing faults for some applications with a little additional memory space.

1. 서 론

분산 공유 메모리(distributed shared memory : DSM) 시스템은 상호연결망에 접속된 여러 노드(node)들로 구성되며, 각 노드는 하나의 처리기와 지역 메모리(local memory)를 가진다. DSM의 장점은 병렬 응용이 수행될 때 처리기들 간에 갱신된 내용들의 전송이나 관리가 운영체제, 또는 그 위의 소프트웨어 계층에서 이루어지기

† 정 회 원 : 단국대학교 자연과학부 교수
논문접수 : 1999년 3월 4일, 심사완료 : 2000년 1월 31일

때문에 응용 프로그래머에게 손쉬운 프로그래밍 기법을 제공한다는 것이다[1-4]. 참조 국지성(locality)의 활용과 동시 수행의 지원을 위해 DSM 시스템은 데이터 이주(migration)와 데이터 복제(replication)를 도입하고 있다[5]. 이는 임의 프로세스가 원격 처리기(remote processor)에 있는 공유 데이터를 참조할 때 원격 처리기로부터 그 데이터 블록(메모리 일관성 유지의 기본 단위로 페이지나 캐쉬 라인을 의미)을 참조한 처리기의 지역 메모리로 이동(또는 복사)시켜 접근하는 방법으로 통신 지연을 줄여 준다. 블록 이주는 병렬 응용이 강한 참조 국지성을 보일 경우에는 큰 성능 향상을 보장해 준다. 그러나 블록 이주는 block- bouncing이라는 역효과를 낳을 수 있다. block-bouncing은 이주된 블록이 곧바로 또다른 처리기에 의해 요청되어 그 블록이 한 처리기에서 또다른 처리기로 계속 이주되는 현상으로 성능 저하의 원인이다 block-bouncing은 블록의 복사본을 해당 처리기에 복제시킴으로써 해결될 수 있다. 블록이 복제될 경우, 동일 블록이 여러 처리기에서 동시 접근될 수 있어 시스템의 병렬성이 향상되지만, 어떤 처리기가 공유 데이터의 로컬 복사본을 수정했을 경우 그 데이터의 복사본을 보유하고 있는 모든 처리기에게 그 사실을 알려 메모리 일관성을 유지해야 하는 문제가 발생한다.

DSM 시스템에서 블록 이주와 복제 기법의 구현을 더욱 복잡하게 만드는 요인으로 거짓 공유(false sharing) 현상이 있다. 거짓 공유는 블록이 이주 및 복제, 그리고 메모리 일관성 유지의 기본 단위라는 사실에서 발생한다. 즉, 병렬 응용에서 서로 관련이 없는 '공유 데이터 객체'(또는 공유 변수)들이 동일한 메모리 블록에 공존하기 때문에 거짓 공유가 빌생하며, 블록의 크기가 커질수록 더 많은 거짓 공유기 발생한다[6-11]. 특히 공유 데이터 일관성을 유지하기 위해 무효화 프로토콜(invalidation protocol)을 사용하는 페이지 기반 DSM 시스템에서 페이지의 한 워드만 수정될 경우에도 그 페이지의 다른 복사본들을 무효화시켜야 하기 때문에 시스템 성능이 크게 저하되기도 한다. 이러한 거짓 공유는 기존의 동적 공유 메모리 할당자(dynamic shared memory allocator)나 컴파일러가 데이터를 적절히 정렬(alignment)하지 않고 서로 다른 참조 패턴을 가지는 데이터들을 같은 블록에 위치시키기 때문에 더욱 많이 유발된다[9, 11]. 거짓 공유로 인해 block-bouncing 현상은 더욱 심화되며, 또한 블록이 거짓 공유되면 메모리 일관성 프로토콜은 더 이상 좋은 해결책을 갖지 못한다[12].

DSM 시스템의 성능을 향상시키는 한 방법은 거짓 공유의 제거이다. 본 논문에서는 동적 공유 메모리 할당자에 의해 공유 데이터 객체가 생성되는 병렬 응용들을 대상으로 하여 공유 객체들의 할당 패턴과 처리기들의 공유 객체 참조 패턴을 거짓 공유와 관련지어 분석한다. 분석한 공유 메모리 참조 패턴을 반영하여 거짓 공유를 줄일 수 있는 새로운 메모리 할당 방식도 제시한다 거짓 공유를 줄이기 위한 기존의 동적 공유 메모리 할당 방식으로는 "처리기별 할당 방식"(allocation scheme per processor)[13], "다중 페이지 걸침 최소화 방식"(minimization scheme of multiple page span)[8, 13], 그리고 "객체 크기 별 할당 방식"(sized allocation scheme)[9, 24] 등이 있는데, 이러한 방식들이 거짓 공유 감소에 기여하는 정도도 분석한다. 본 논문의 2장에서는 거짓 공유를 줄이기 위한 기존의 연구들에 대해 기술하고, 3장에서는 실험 환경 및 병렬 응용들의 공유 메모리 참조 특성을 설명한다. 4장에서는 거짓 공유를 줄일 수 있는 새로운 메모리 할당 방식을 제시하면서 성능을 평가하고, 5장에서 결론을 맺는다

2. 관련 연구

거짓 공유 오버헤드를 줄이기 위해 Torrellas 등은 여러 가지의 데이터 배치 기법들을 제시하였다[13]. 즉, 거짓 공유 현상을 보이는 변수들을 서로 다른 캐쉬 블록에 배치하는 기법, lock 변수와 lock 변수로 보호되는 데이터를 가능한 한 동일한 캐쉬 블록에 배치하는 기법, 한 레코드의 크기를 캐쉬 라인 크기에 맞게 패딩(padding)하는 방법 등을 제시하였다. 이러한 기법들이 거짓 공유를 줄이는 데 큰 도움을 주지만, 향후에는 컴파일 시나 수행 중에 자동적으로 지원되어야 한다는 점도 지적하였다 Jeremiassen 등은 거짓 공유 미스를 최소화하기 위하여 병렬 응용을 분석하고 공유 데이터를 재구성하는 컴파일러 알고리즘을 개발하였다[14]. 이 알고리즘은 프로세스별로 공유 데이터에 대한 참조 패턴을 분석한 다음, 이 정보를 사용하여 거짓 공유를 유발하는 자료 구조를 찾아내고 적당한 변환 기법을 사용하여 거짓 공유를 줄인다. 이들이 제시한 변환기법은 메모리에서 자료 구조의 위치를 물리적으로 바꾸어 공간 국지성을 개선하는 기법, 자료 구조를 재배치하기 어려운 경우에는 자료 구조를 처리기별로

따로 배치한 다음 포인터를 두어 자료 구조를 간접적으로 참조하는 기법, 데이터를 패딩하는 기법 등이다. 이들은 이러한 변환 기법을 통해 거짓 공유 미스를 평균 64% 정도 줄인 결과를 제시하고 있다. 그러나 병렬 응용을 정적으로 방대하게 분석해야 한다는 단점이 있다.

Rothman[14]등은 페이지 단위가 아닌 CPU 캐쉬 블록 단위로 여러 응용의 공유 메모리 미스에 대해 분석하면서, 거짓 공유는 블록의 크기가 클수록 증가하며 특정 블록에 집중되어 발생한다는 결과를 제시하였다[15]. 그리고, 프로그래머가 주의하여 일부 공유 변수들을 재구성한다면(미사용 배열들을 추가하여 처리기별 데이터를 서로 분리시킨다면) 거짓 공유도 감소되고 시스템 성능도 향상될 수 있다고 주장한다. 그러나, 이를 적용하려면 응용 프로그램을 수정해야하며 수정에 따라 명령어 수가 증가한다는 단점이 있다. 거짓 공유를 감소시키기 위한 다른 기법으로, 각 객체에 대한 처리기들의 참조 횟수를 고려하여 실행 중간에 그 객체를 자주 참조하는 처리기로 이주하거나 복사하는 기법[16, 17]이나, 한 블록 내의 갱신된 워드를 전송하기 위해 순차적 일관성 프로토콜이 아닌 느슨한 일관성 프로토콜을 사용하는 기법[18, 19]을 생각할 수도 있지만, 이는 공유 객체 할당과는 다른 분야의 문제이다.

DSM에서 주목할 점은, 많은 병렬 응용들이 공유 데이터를 필요에 따라 동적으로 할당한다는 것과 병렬 응용에서 필요로 하는 공유 메모리를 하나의 주 처리기가 총괄하여 할당한다는 것이다[20, 21]. 병렬 처리 환경에서 제공하는 동적 공유 메모리 할당자를 이용하여 공유할 공간을 획보하는 방법이 다중 경량 프로세스(*light-weight process*)를 이용한 주소 공간 기법보다 쉽고 호환성이 좋기 때문에 대부분의 병렬 응용들은 동적 공유 메모리 할당자를 사용한다. 기존의 연구에서 제시된 기법들은 보통 동적으로 할당되는 공유 데이터에 대해서는 적용할 수 없다. 기존 동적 공유 메모리 할당 루틴은 기본적으로 순차적 할당 기법을 사용하여 공유 메모리를 할당하기 때문에, 한 페이지에 여러 자료 구조가 섞여서 할당될 가능성이 많다. 이 같은 할당 방식은 메모리 공간을 낭비 없이 효율적으로 사용할 수 있고 구현이 간단하다는 장점이 있지만 서로 관계없는 여러 데이터 객체가 한 페이지에 공존하기 때문에 거짓 공유가 많이 발생한다. 거짓 공유 문제를 해결하기 위한 기존의 동적 메모리 할당 방식들은 다음과 같다.

2.1 기존 동적 메모리 할당 방식

2.1.1 처리기별 할당 방식

이 방식에서는 거짓 공유를 줄이기 위해 공유 메모리 할당을 요청한 처리기 별로 별도의 페이지를 사용함으로써, 하나의 페이지에 서로 다른 처리기가 요구한 데이터 객체들이 섞이지 않도록 한다[13]. 이 방식은 각 처리기가 자신이 필요로 하는 객체를 직접 요청할 수 있는 응용에서는 처리기 국지성을 활용할 수 있는 장점을 지닌다. 그러나, 실제 벤치마크로 사용되는 병렬 응용들의 대부분이 주 처리기가 공유 메모리 할당과 반환을 담당한다[20, 21]. 최근 연구들[15, 18, 19]에서 사용된 SPLASH2[21] 병렬 응용[1]도 처리기 0에서 수행되는 주 프로세스가 모든 공유 메모리 할당을 담당하기 때문에, 처리기 별 할당 방식이 도움이 되지 못한다. 본 논문에서는 SPLASH2 응용들의 거짓 공유를 줄이고자 하는 것이 목표인데, SPLASH2에는 처리기별 할당 방식을 적용할 수 없다.

2.1.2 다중 페이지 걸침 최소화 방식

이 방식에서는 하나의 객체가 두 개 이상의 페이지에 걸쳐서 할당되는 것을 최소화시켜 거짓 공유를 줄인다. 이는 데이터를 처리기 캐쉬 경계 내에 배치시키는 기법과 일치한다[8, 13]. 이 방식은, 첫째로 페이지보다 작은 객체들이 두 페이지에 걸치지 않도록 하는 것이고, 둘째로 큰 객체들이 여러 페이지에 걸치는 것을 최소화하는 것이다. 이렇게 하면 한 객체가 여러 개의 페이지에 걸치지 않게 되어 거짓 공유가 감소될 수도 있지만, 경우에 따라서는 객체 내의 워드 배치가 달라지게 되어 워드 단위의 거짓 공유 수는 증가할 수도 있다(4.2절 참조). 이 방식은 독자적으로 사용될 수도 있지만 타 방식과 함께 적용될 때도 있다. 본 방식에서도 이 방식을 채택한다.

2.1.3 객체 크기별 할당 방식

이 방식은 요청된 데이터 객체의 크기별로 공유 객체들을 별도의 페이지를 할당함으로써, 크기가 서로 다른 데이터 객체가 동일 페이지에 섞이는 것을 방지하여 성능을 향상시키고자 하는 방식이다[9, 24]. 그러

1) SPLASH2의 응용에서, 모든 공유 객체들은 주 처리기인 0번 처리기에 의해 0번 처리기의 지역 메모리에 일단 할당 - 배치된 다음, 실제 참조하는 처리기의 지역 메모리로 복사/이주된다.

나 3.2절의 <표 2>를 보면 SPLASH2 응용에서는 동일 크기의 객체들이 대부분 연속적으로 할당되니 서로 다른 참조 패턴을 보인다. 또한, 같은 크기의 객체들이 동일 페이지나 연속된 페이지들에 할당되는 경우가 많기 때문에, SPLASH2의 6개 응용에서 크기별 방법은 순차적 할당 기법과 유사하며 성능 향상에 큰 영향을 끼치지 않을 수 있다. 본 논문에서 제시하는 방식은 순차적으로 할당되는 객체들을 서로 다른 페이지에 배치시킴으로써 참조 패턴이 다른 객체가 동일 페이지에 섞이는 것을 줄여 준다.

3. 공유 메모리 할당 및 참조 패턴

3.1 실험 환경

본 논문에서는 프로그램 구동형 시뮬레이터(program-driven simulator)인 Mint[22, 23]를 Solaris 2.5.1/Ultra Sparc에 설치하여 페이지기반 DSM 시스템을 실험하였다. Mint는 메모리 참조 발생기(memory reference generator)와 목표 시스템 시뮬레이터라는 두 부분으로 구성되어 있다. 메모리 참조 발생기는 프로그램이 여러 개의 처리기 상에서 수행되는 결과를 생성하면서, 프로그램이 특정 연산을 수행하면 그 사실을 사건(event)으로 목표 시스템 시뮬레이터에 알려준다. 목표 시스템 시뮬레이터는 시스템의 연결 망과 메모리 계층 구조를 모델링하며, 메모리 참조 발생기가 전송한 사건을 분석하여 적절한 연산을 수행한 다음 메모리 참조 발생기가 다시 프로세스를 진행할 수 있도록 신호를 보낸다. Mint는 병렬 프로세스들을 직접 스케줄링하면서 각 프로세스에 관련된 정보를 명령 단위로 추적하기 때문에[23] 실제 시스템에서 실험하는 것과 동일한 효과를 얻을 수 있다.

<표 1>은 페이지 크기가 4KB이고 처리기 수가 32개일 때 여러 병렬 응용들의 특성들을 보여준다. 표에 나타난 병렬 응용들은 모두 SPLASH2[21] 사이트에서 가져온 것들이다. Cholesky는 희소 행렬에서 Cholesky 인수 분해를 수행하는 응용이고, fft는 여섯 단계의 FFT 알고리즘을 수행하는 응용이다. lu는 밀집 행렬을 하위 삼각 행렬과 상위 삼각 행렬로 인수 분해하는 응용이고, ocean은 거대한 해양의 유동 상태를 시뮬레이션하는 응용이다. radix는 기수 정렬을 수행하는 응용이고, water-spatial은 액체 상태에서의 물분자의 힘과 전위를 측정하는 응용이다. Cholesky의 경우, 공유 데이터 객체들로 구성된 공유 메모리의 총 크기는 20.74MB(5,311개의 페이지)이며 공유 데이터들이 읽기나 쓰기 연산에 의해 참조되는 횟수는 308.64×10^6임을 나타낸다. 이들 응용들의 특성은 [15, 18]에도 잘 설명되어 있다. Mint를 이용하여 실행한 SPLASH2의 응용들의 명령 인자는 다음과 같다. 공유 메모리 참조 횟수와 크기는 프로그램 수행 시 주어지는 명령 인자들에 의해 차이가 있다.

<표 1> 병렬 응용들의 특성

병렬 프로그램	문제 크기	공유 메모리 참조 회수 ($\times 10^6$)	공유 메모리 크기 (MB 바이트)
cholesky	tk15.O	308.64	20.74
fft	1M complex doubles	212.88	48.40
lu	512×512 matrix	277.38	2.13
ocean	258×258 grid	278.64	23.01
radix	256K keys, 1024 radix	6.02	4.63
water-spatial	512 molecules	84.87	0.33

- cholesky/CHOLESKY -p32 cholesky/inputs/tk15.0
- fft/FFT -m20 -p32 -n65536 -l4
- lu/contiguous_blocks/LU -n512 -p32 -b16
- ocean/non_contiguous_partitions/OCEAN -n258 -p32 -e1e-07 -r20000 -t28800
- radix/RADIX -p32 -n262144 -r1024 -m524288
- water-spatial/WATER-SPATIAL < water-spatial/input.32

Fft의 경우 -m16 (65536 complex doubles) 인자와 -n1048576 (1M cache lines) 인자 등을 이용하여 수행시켜보았으며 인자들에 따라 할당되는 객체의 크기가 많이 다르다. LU의 경우, 가상 메모리에 각 데이터 블록을 연속되게 할당하는 "conuguous" 버전을 사용했다. Ocean의 경우 두 개의 버전이 있는데, 본 논문에서는 "contiguous" 버전이 에러가 발생하여 "noncontiguous" 버전을 사용했다. 위의 6개 이외의 다른 응용들은 구축한 본 시스템에서 에러가 나고 실행되지 않았다.

각 응용의 공유 객체들은 Mint의 mint_shmalloc()/sim_shalloc()에 의해 40001_{16}[2] 가상 페이지부터 순차적으로 할당된다. 그리고, spin lock과 barrier events, sema-

2) 공유 페이지들은 가상주소 0x40001000에서 시작되며, 시작 가상 페이지 번호는 $262145(=40001_{16})$이다. 본 논문에서는 262145번 페이지를 0번 공유 페이지로 칭한다.

phore 등의 동기화 변수가 synch_alloc()/ sim_lockalloc()에 의해 40000_{16} 가상 페이지에 할당된다. 본 실험에서 구현된, DSM 시스템을 모델링하는 목표 시스템 시뮬레이터에서 sim_shalloc()을 이용하여 동기화 객체를 제외한 일반 객체들의 할당 패턴을 분석하였다. 그리고 두 번째 목표 시스템 시뮬레이터를 개발하여 공유 객체별 처리기 참조 패턴과 공유 페이지 별 처리기 참조 패턴, 여러 폴트 수 등을 분석하였다. 끝으로, 본 논문에서 제시된 기법을 적용하기 위하여 메모리 참조 발생기에 포함된 공유 메모리 할당자를 수정하였고, 앞서 구현된 목표 시스템 시뮬레이터를 이용하여 성능을 평가하였다. 실험에서 구현한 DSM 시스템은 메모리 일관성 유지 프로토콜로서 무효화 기반 프로토콜을 사용하고, 페이지 소유자를 찾기 위하여 Li 등이 고안한 '동적 분산 관리자 알고리즘'[1]을 사용한다.

3.2 공유 객체들의 할당 및 참조 패턴

처리기 수가 32개이고 페이지의 크기가 4K 바이트일 때 여섯 개의 병렬 응용의 공유 메모리 할당 요청 패턴을 분석한 결과를 <표 2>에 제시하였다. 공유 메모리 할당 패턴을 분석한 결과 동일한 크기의 객체들은 대부분 할당 요청이 연속적으로 이루어진다. <표 2>에서 '요청 크기'는 병렬 응용이 수행될 때 동적으로 요청되어 할당된 데이터 객체의 크기로, 태그(tag) 정보를 제외한 각 객체 자체의 크기만을 나타낸다. 태그는 각 객체들이 반환될 때 사용되는 메모리 제어 블록으로, 해당 데이터 객체와 연속된 위치에 놓이는 것이 일반적이며 그 크기는 4바이트~12바이트 정도이다. '연속 요청 회수'는 동일한 크기의 데이터 객체가 연속적으로 몇 번 요청되었는가를 가리킨다. 예를 들면, cholesky에서는 262,152 바이트 크기의 데이터 객체가 총 76번 요청되는데, 2번, 50번, 24번씩 연속적으로 요청된다는 것을 의미한다.

6개 응용의 공유 메모리 참조 패턴은 다음 세 가지로 요약할 수 있다.

참조 패턴 1 : 대부분 순차적으로 할당되는 공유 객체들은 서로 다른 참조 패턴을 보인다. 특히, 동일 크기의 객체들이 처리기 수만큼 연속 할당되는 경우에는 특정 객체가 특정 처리기(들)에 의해서만 참조되었다. cholesky에서 8,264 바이트 크기의 객체와 radix의 8,192 바이트 크기의 객체들은 각각이 한 처리기에 의

〈표 2〉 할당 요청 크기와 요청 횟수

요청 크기 (바이트)	요청 회수	연속 요청 회수	요청 크기 (바이트)	요청 회수	연속요청 회수
cholesky			ocean		
16	1		24	1	
72	33	33	32	1	
152	1		80	1	
664	1		164,208	1	
8,264	32	32	532,512	2	
262,152	76	2, 50, 24	534,576	1	
357,632	1		1,065,024	2	
487,064	1		1,065,032	1	
715,264	1		1,597,536	2	2
			2,130,048	3	
			9,585,360	1	
fft			radix		
40	1		136	2	2
128	2	2	256	3	3
16,384	1		8,192	32	32
16,912,384	3	2	131,200	1	
			1,048,576	2	2
			2,361,024	1	
lu			water-spatial		
56	1		16	69	2, 64
128	1		24	32	32
256	5	5	32	16	4,4,4,4
4,096	2	2	128	2	
69,632	32	32	176	1	
			680	512	512

해서만 참조된다. cholesky의 147개 공유 객체들 중에서 8,264 바이트 크기의 객체는 92번[3]째부터 123번째까지 32번 연속적으로 할당되는데, 각 객체는 하나의 처리기에 의해서만 참조된다. O_{92}(92번 객체)는 1번 처리기에 의해서만 참조되며, O_{93}은 2번 처리기에 의해서만 참조되며, O_{123}은 0번 처리기에 의해서만 참조된다. cholesky의 72 바이트 크기의 객체들은 4번째부터 36번째까지 33번 연속적으로 할당되며, 모두 공유 페이지 0에 배치된다. 이들 중 O_4와 O_{36}은 0번 처리기에 의해서만 참조되며, O_5는 0번과 1번 처리기에 의해서만 참조되며, O_6은 0번과 2번 처리기에 의해서만 참조되며, O_{35}는 0번과 31번 처리기에 의해서만 참조된다. cholesky의 O_5~O_{34}들 각각은 0번 처리기에 의해서 12% 정도 참조되며 다른 하나의 처리기에 의해서 87% 이상 참조된다. water-spatial의 24 바이트 크기의 객체들은 2번째부터 33번째까지 32번 연속적으로 할당된다

3) 편의상 할당되는 객체는 1번(O_1), 두 번째로 할당되는 객체는 2번(O_2) 등과 같이 번호를 매겼다

이들 중 O_2는 0번 처리기에 의해서만 참조되며, O_3은 0번과 1번 처리기에 의해서만 참조되며. O_4는 0번과 2번 처리기에 의해서만 참조되며, O_{32}은 0번과 31번 처리기에 의해서만 참조되지만, 모두 공유 페이지 0에 배치된다 이처럼 cholesky의 72 바이트 크기의 객체들과 water-spatial의 24 바이트 크기의 객체들은 연속적으로 할당되며 각각이 한 개, 또는 두 개의 처리기에 의해서만 참조되는 서로 다른 참조 패턴을 보인다 기존의 순차 할당이나 '크기별 할당 기법'[9]은 이들 객체들을 일관성 유지 단위인 하나의 페이지에 배치시켜, 처리기들이 빈갈아 접근할 경우에는 page-bouncing 현상을 유발하게 되며 결과적으로 거짓 공유 현상을 더욱 심화시키게 된다.

참조 패턴 2 : 동일 크기의 객체들이 처리기 수보다 훨씬 많이 연속 할당되는 경우에는 연속된 2개 이상의 객체가 한 그룹이 되어 특정 처리기에 의해서만 참조된다. water-spatial의 680 바이트 크기의 객체들과 16 바이트 크기의 객체들이 대표적인 예이다. 680 바이트 크기의 객체들은 57번째부터 568번째까지 512번 연속적으로 할당되며, 각 객체마다 전체 참조의 85% 이상이 한 처리기에 의해서 이루어진다. 이들 중 O_{57}과 O_{58}은 0번 처리기에 의해서 집중적으로 참조되고, O_{59}와 O_{60}은 1번 처리기에 의해 주로 참조되고, O_{61}과 O_{62}는 2번 처리기에 의해서 주로 참조된다. 16 바이트 크기의 객체들은 569번째부터 632번째까지 64번 연속적으로 할당되며, 연속한 두 객체가 동일한 참조 패턴을 보인다. 즉, O_{569}와 O_{570}은 1번 처리기에 의해서만 참조되며, O_{571}과 O_{572}는 2번 처리기에 의해서만 참조되며, O_{631}과 O_{632}는 0번 처리기에 의해서만 참조된다 순차 할당 기법에서는 680 바이트 객체들의 경우 최대 6개의 객체가 한 페이지에 공존하게 되며, 16 바이트 객체들의 경우 모두 85번 공유 페이지에 공존하게 되므로, 처리기들이 번갈아 참조할 경우 거짓 공유로 인한 성능 저하가 발생할 수 있다. 이는 크기별 할당 기법에서도 마찬가지이다.

참조 패턴 3 . 페이지 크기보다 매우 큰 객체의 경우, 객체 전체로 보면 여러 처리기들에 의해 참조되나 객체 내의 특정 페이지는 특정 처리기에 의해서만 참조되는 페이지별 참조 패턴을 보인다. cholesky의 262,152 바이트 크기의 객체와 lu의 69,632 바이트 객체, ocean의 1,065,024와 1,597,536 바이트 객체 등이 대표적인 예이다 이들 객체에 대해서는 객체 할당 기법만으로는 거짓 공유 현상을 완화시킬 수 없다 본 논문에서는 객체 단위로 공유 메모리를 할당하는 기법에 대한 연구이며, 페이지보다 매우 큰 객체에 대해서는 다중 페이지 절침 최소화 기법을 적용하여 거짓 공유를 감소시키고자 한다.

3.3 기존 결과와의 비교

Cholesky의 경우, 참고문헌 [24]에 나타난 할당 패턴과 상이한 패턴을 보이는데, [24]의 자료는 SPLASH2가 아닌 SPLASH의 cholesky를 수행시켰기 때문이라고 분석된다. 이를 확인하기 위해 SPLASH2에서 tk14.O, tk15.O, tk23.O, lshp O, wr10 O 등의 입력을 이용하여 cholesky를 수행시켜 보았다. 이들 입력들에 대해서도 <표 2>에서처럼 동일 크기의 객체들이 처리기 개수만큼 또는 그 이상 할당되는 경우가 존재하였으며, 동일 크기의 객체들이 연속적으로 할당되었다. 처리기 수가 4개, 8개, 16개일 때도 유사한 할당 패턴을 보였다. Cholesky의 경우, 객체 할당 패턴에 영향을 줄 수 있는 인자는 처리기 수와 입력 데이터뿐이며, SPLASH2에서는 [24]의 결과가 나오지 않는다. SPLASH2는 SPLASH의 제한점이 개선된 응용이므로 SPLASH보다는 본 실험에 적합하다고 판단되며, 최근 연구들[15, 18, 19]에서도 많이 이용되고 있다. [24]에 나타난 cholesky의 할당 패턴은 <표 2>의 fft와 ocean의 할당 패턴과 유사하며, 페이지 크기보다 큰 객체들이 한번씩 주로 할당되는 경우에는 본 논문에 인급된 어떤 할당 기법도 좋은 성능을 내지 못할 것이라 생각된다(4.2절 참조).

본 실험에서는 처리기 수를 4, 8, 16개로 하여 객체들의 할당 및 참조 패턴도 분석해 보았다. 처리기 수가 증가할수록 공유 객체에 대한 전체 할당 요청 횟수도 증가하며, 여러 번 할당되는 동일 크기의 객체들은 대부분 연속되어 할당되었다 각 응용에서 여러 번 할당되는 객체들 중 어떤 객체는 처리기 수에 비례하여 할당 요청 수가 증가하였다. 처리기 수가 증가할수록 객체들의 평균 할당 요청 크기는 감소하였으며, 하나의 페이지에 공존하는 객체의 수가 증가하고, 같은 페이지를 참조하는 처리기 수가 많아지 거짓 공유 돌트의 수도 증가하였다 그리고, 페이지 크기가 클수록 어러 데이터 객체가 한 페이지에 공존할 확률이 늘아지므로 거짓 공유도 증가한다. 현재 대부분의 처리기들이 4KB 또는 8KB 페이지를 채택하고 있으며 본 논문에서는 주로 4KB 페이지에 대해 실험하였다. 8KB 페

이지의 경우, 객체 할당 패턴에는 차이가 없으며 거짓 공유가 증가하였다.

4. 동적 공유 메모리 할당 방식

4.1 메모리 참조 패턴에 기반한 공유 메모리 할당

본 절에서는 메모리 객체 할당 및 참조 패턴에 기반하여 거짓 공유를 줄이기 위한 새로운 동적 공유 메모리 할당 기법을 제시한다. 기존의 연구들에서 제안한 거짓 공유의 척도 및 측정 방법들[9, 11]을 보면 서로 다른 참조 패턴을 갖는 객체를 서로 다른 페이지에 배치시키면 거짓 공유 폴트를 감소시킬 수 있다. 서로 관련이 없거나 참조 패턴이 다른 데이터 객체들이 한 페이지에 섞이지 않도록 하기 위해, 다음과 같은 메모리 할당 정책을 채택하였다.

① 3.2절의 "참조 패턴 1"과 "참조 패턴 2"에서 기술한 것처럼 객체들이 순차적으로 할당되는 경우, 다른 처리기에 의해 참조되는 객체들이 동일 페이지에 공존하는 경우가 많게 되어 거짓 공유가 많이 발생한다. 본 기법에서는 할당되는 객체들을 순서적으로 각각 다른 페이지에 배치시켜, 참조 패턴이 다른 객체가 동일 페이지에 공존하는 것을 줄인다. 즉, 공유 객체들을 할당하기 위한 공유 페이지 풀(pool)을 관리하면서, 객체의 할당 순서를 반영하여 동일 페이지에 서로 다른 처리기가 참조하는 데이터 객체들이 섞이는 것을 감소시켜 거짓 공유를 줄인다. 본 방식에서는 처리기의 수에 비례하게 페이지 풀을 미리 예약하여 이들 페이지에 차례로 요청되는 공유 데이터 객체를 할당한다.

② 추가로 페이지 걸침 최소화[8, 9]를 적용하여 한 객체가 두 개의 페이지에 걸쳐서 할당되지 않도록 한다. 특히, "참조 패턴 3"에서처럼 데이터 객체의 크기가 페이지 크기보다 큰 경우에는 데이터 객체를 페이지 경계에 정렬시켜 할당함으로써 하나의 데이터 객체가 여러 페이지에 걸쳐서 할당되는 경우를 최소화하여 거짓 공유의 가능성을 줄인다.

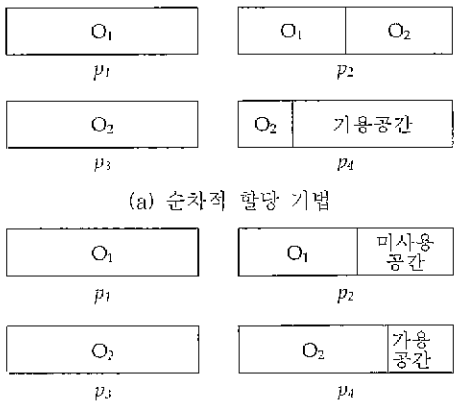
제시한 할당 기법에서는 연속적으로 할당되는 객체들이 일반적으로 서로 다른 참조 패턴을 보인다는 사실에 입각하여, 순차적으로 할당되는 객체들을 온-라인(on-line)으로 각각 다른 페이지에 배치시킴으로써 서로 관련없는 객체들이 동일 페이지에 섞이지 않도록 하였다. 먼저, 전체 처리기 수의 1/4~1배만큼의 페이지를 풀(pool)로 유지하여 객체들이 서로 다른 페이지에 번갈아 가며 배치되도록 하였다. 8개의 페이지로 구성된 풀일 경우, 객체번호를 8로 나누어 나머지가 1이면 첫 번째 페이지에 배치하고, 나머지가 2이면 두 번째 페이지에 배치한다. 본 기법은 주 처리기가 공유 객체를 총괄하여 할당하는 병렬 응용에서 거짓 공유를 감소시킬 수 있는 한 가지 대안이 될 수 있다. 제시된 기법에서도 동일 페이지에 다른 처리기에 의해 참조되는 객체들이 섞일 수 있지만, 3.2절에서 기술한 참조 패턴에 의하면 순차적 할당 기법에서보다는 줄어들 수 있다.

다중 페이지 걸침을 최소화하기 위한 예는 (그림 1)에 나타나 있다. O_1과 O_2의 참조 패턴이 다를 경우, 순차 할당 기법에서는 O_2 객체가 세 개의 페이지에 걸치게 되고 p_2 페이지에는 O_1과 O_2가 공존하게 되어 거짓 공유가 많이 발생하게 된다. 그러나 제시한 기법에서는 O_2가 두 개의 페이지에 걸치게 되며, p_2에서의 거짓 공유는 줄어든다. 이를 다음과 같은 모델링[9, 11]을 이용해 확인할 수 있다. 메모리 일관성의 단위가 하나의 페이지(p)라고 할 때, 페이지 p_j내의 특정 워드 w_i에 대한 거짓 공유 발생 정도는 다음과 같이 표현될 수 있다.

$$FS(j) = 1 - \frac{|W_i|}{|P_j|} \quad (1)$$

$|W_i|$: 프로그램 실행 시, 페이지 p_j내의 워드 w_i를 참조하는 처리기 집합

$|P_j|$: 워드 w_i가 속한 페이지 p_j를 참조하는 처리기 집합



(그림 1) 다중 페이지 걸침의 최소화

서로 관련없는 여러 객체가 페이지 j에 공존할 경우

페이지 j를 참조하는 처리기들의 수가 많아져 $|P_j|$는 증가할 것이고, 그 페이지 내의 워드 w_i를 참조하는 처리기 수는 변화가 없을 것이므로 $FS(j)$는 커진다. O_1과 O_2가 서로 관련있다 하더라도, 제시한 방법이 기존 할당 방법보다 거짓 공유의 정도가 작기나 같다.

〈표 3〉 여러 객체가 공존하는 페이지에 대한 정보

응용	공존 객체 수	페이지수	응용	공존 객체 수	페이지수
cholesky	2	110	ocean	2	11
	37	1		3	2
fft	2	4	radix	2	34
	3	1		7	1
lu	2	34	water-spatial	7	83
	7	1		8	1
				60	1
				67	1

페이지 걸침 최소화 방식을 적용하기 위해, 참고로 순차 할당에서 하나의 페이지에 몇 개의 객체가 공존하는가를 <표 3>에 나타내었다. Cholesky의 경우 147개의 공유 객체들이 총 5,311개의 페이지를 차지하는데, 그 중 110개의 페이지에는 2개의 객체가 공존하고 1개의 페이지에는 37개의 객체가 공존하며 나머지 5,200개의 페이지에는 한 개의 객체(객체가 커서 두 개 이상의 페이지에 걸쳐있는 경우가 대부분임)만 존재한다. 표에서 두 개 이상의 객체를 포함하는 페이지의 수가 cholesky는 111개, fft는 5개, lu는 35개, ocean이 13개, radix가 35개, water-spatial이 86개이다. 반대로, 한 개의 객체만을 포함하는 페이지 수가 cholesky가 5,200 페이지(전체 공유 페이지의 98%), fft가 12,387 페이지(99.9%), lu가 512 페이지(94%), ocean이 5,879 페이지(99.8%), radix가 1,150 페이지(97%), water-spatial이 0 페이지(0%)이다. 즉, 평균적으로 공유 객체가 커서 두 페이지 이상에 걸쳐 있는 경우가 많다 한 페이지에 할당되는 평균 객체 수가 water-spatial의 경우 7.35개이나, 다른 응용들의 경우 0.1개 미만이다 본 논문에서는 한 객체가 두 개 이상의 페이지에 걸치는 것을 줄이고자 노력하였다. 본 논문에서 제시된 할당 기법을 라이브러리 내의 공유 메모리 동적 할당자에 구현하였으며, 사용자 인터페이스는 전혀 변경되지 않았다 따라서 프로그래머는 병렬 응용을 컴파일할 때에 기존 메모리 할당자 대신 새로 구현된 루틴이 포함되어 있는 라이브러리만 링크시키면 된다 실제 제시한 할당 기법을 Mint에 구현하였으며, 기존의 병렬 응용들은 수정없이 그대로 수행된다

4.2 동적 메모리 할당 방식들의 성능 평가

동적 메모리 할당 방식들이 거짓 공유에 미치는 영향을 분석하기 위해서, 순차적 할당 방식, 다중 페이지 걸침 최소화 방식, 다중 페이지 걸침을 최소화한 크기별 할당 방식, 그리고 본 논문에서 제시한 방식 등에서 발생한 cold 폴트, 참 공유 폴트, 거짓 공유 폴트의 수를 비교하였다. 실험한 결과가 <표 4>에 나타나 있다. 실험에서 사용한 페이지 풀의 크기는 32이다 전체 폴트 중에서 거짓 공유 폴트가 큰 비중을 차지함을 알 수 있다. 제시한 기법은 cholesky, radix, water-spatial에서 거짓 공유 폴트의 수를 크게 줄여준다 이는 cholesky와 radix에는 '32절의 '참조 패턴 1'을 보이는 객체들이 많고 water-spatial에는 '참조 패턴 2'를 보이는 객체들이 많기 때문이다 즉, 순차 할당에서는 참조 패턴이 다른 여러 객체가 한 페이지에 공존하는 경우가 많은

〈표 4〉 동적 메모리 할당 방식들의 폴트 수

병렬 응용	폴트 유형	순차 할당	크기별 할당	페이지걸침 최소화	제시한 방식
choles-ky	cold	29,175	29,806	29,621	29,402
	TS	26,098	25,941	26,004	19,819
	FS	457,933	465,628	452,943	407,953
	Total	513,506	521,375	508,571	457,174
fft	cold	89,191	89,160	89,160	88,902
	TS	4,321	4,321	4,321	4,130
	FS	30,626	30,626	30,626	30,624
	Total	124,138	124,107	124,107	123,656
lu	cold	4,394	4,397	4,395	4,272
	TF	3,603	3,603	3,603	1,527
	FS	5,070,891	5,070,891	5,070,891	5,070890
	Total	5,078,888	5,078,891	5,078,889	5,076,689
ocean	cold	16,634	16646	16,550	16,417
	TS	238,799	238,838	238,867	199,999
	FS	50,012,876	50,114,147	50,125,397	50,104,906
	Total	50,268,309	50,369,631	50,380,811	50,321,322
radix	cold	16,923	16,936	16,963	16,587
	TS	1,430	1,427	1,428	790
	FS	819,654	510,407	510,413	492,280
	Total	838,007	528,797	528,804	509,657
water-spatial	cold	2,554	2,670	2,873	3,094
	TS	11,274	11,314	12715	12,030
	FS	877,574	869,701	765,490	618,177
	Total	891,402	883,685	781,078	633,301

※ cold(cold 미스), TS(참 공유 폴트), FS(거짓 공유 폴트), Total(전체 폴트)

대 비해, 제시한 기법에서는 참조 패턴이 다른 객체들이 한 페이지에 공존하는 것을 많이 감소시켰기 때문이다. 크기별 할당 기법은 cholesky에서 참조 패턴이 다른 객체를 동일 페이지에 많이 공존시키기 때문에 순차적 할당 기법보다 거짓 공유를 많이 유발시켰다.

fft, lu, ocean 등에서는 페이지 크기보다 작은 객체들이 거의 요청되지 않기 때문에, 풀에서의 할당 정책이 별로 효과가 없다. 이들 응용의 경우에는 '참조 패턴 3'을 보이는 객체들이 공유 메모리 참조에서 큰 비중을 차지하기 때문이다. 이 경우 페이지 걸침 최소화를 적용할 수 있으나 그 효과가 크지 않다. ocean의 경우, 순차 할당이 가장 좋은 결과를 보이며 다른 방식들에서는 거짓 공유가 0.22% 정도 증가하였다. ocean에서는 페이지보다 매우 큰 객체들이 특히 많으며, 그 객체들은 여러 처리기들에 의해 번갈아 참조된다(이러한 경우에는 공유 메모리 할당 기법만을 이용해서 거짓 공유를 줄이는 것이 매우 어렵다). 그리고 순차 할당보다 제시된 방식이 객체들을 공유 메모리에 할당하면서 참조 패턴이 다른 워드(거짓 공유의 측정 단위는 기존 연구들[11, 24]에서처럼 워드 단위이며, 하나의 큰 객체 내의 수많은 워드들의 참조 패턴이 다를 경우가 많다)들을 동일 페이지에 공존시키는 경우가 더 많기 때문에, ocean에서는 거짓 공유가 증가되었다. 이는 크기별 할당과 페이지 걸침 최소화 기법에서도 마찬가지이다.

끝으로, 본 기법이 추가로 사용한 메모리 양을 순차적 할당 기법과 비교해 보았다. <표 5>에 나타나 있듯이 본 기법은 순차적 할당 기법보다 138,968 바이트(fft)~346,752 바이트(cholesky) 정도 더 많은 공간을 요구한다. 추가로 드는 공간들은 풀에서 사용되지 않은 공간과 페이지 정렬로 인해 추가로 더 사용된 것들이다. 이는 전체 공유 데이터 크기에 비해 상당히 적은 공간이며 현재 다중 처리기 시스템들이 장착하는 메모리 양의 증가 추세로 보아 큰 낭비는 아니다.

<표 5> 제시한 방식에서 사용된 메모리 공간

병렬 응용	공유 데이터 크기 (바이트)	할당된 메모리 크기 (바이트)	추가로 사용된 메모리 (바이트)
cholesky	21,751,168	22,097,920	346,752
fft	50,753,832	50,892,800	138,968
lu	2,237,880	2,437,120	199,240
ocean	24,129,760	24,285,184	155,424
radix	4,852,560	5,058,560	206,000
water-spatial	350,976	524,288	173,312

5. 결 론

DSM 시스템에서 메모리 블록 관리비용을 줄이고 거짓 공유를 제거하기 위해서는 먼저 데이터 객체 접근 패턴에 기반한 객체 할당 및 메모리 일관성 정책이 수립되어야 한다. 이를 위해 본 논문에서는 동적 공유 메모리 할당자에 의해 데이터 객체가 생성되는 병렬 응용을 대상으로 데이터 객체들의 할당 및 접근 패턴을 거짓 공유와 관련지어 분석하였다. 또한 분석한 결과를 이용하여 거짓 공유를 줄일 수 있는 새로운 메모리 할당 방식을 제시하였으며, DSM 시스템 성능을 향상시키기 위해 거짓 공유를 제거하는 기존 방식들과 비교 평가하였다. 제시한 방식은 사용자 인터페이스 변경없이 기존의 라이브러리 일부만을 수정하여 구현하였으며 일부 응용들의 거짓 공유의 수를 상당량 감소시켜 준다. 공유 객체가 페이지 크기보다 매우 커서 여러 처리기에 의해 참조되나 객체 내의 특정 페이지는 특정 처리기에 의해서만 참조되는 경우에는 객체 할당 방식으로는 거짓 공유를 줄이기가 어렵기 때문에, 추가적인 성능 향상을 위해서 응용 실행 중 페이지 이주나 복제 기법에 대해 연구할 계획이다.

참 고 문 헌

[1] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," ACM Transactions on Computer Systems, Vol 7, pp.321-359, Nov. 1989.

[2] M. Stumm and S Zhou, "Algorithms Implementing Distributed Shared Memory," IEEE Computer, pp.54-64, May 1990

[3] A. S Tanenbaum, Distributed Operating Systems, Chapter 6, Prentice-Hall, 1995.

[4] O. E. Theel and B D Fleisch, "A Dynamic Coherence Protocol for Distributed Shared Memory Enforcing High Data Availability at Low Costs," IEEE Trans on Parallel and Distributed Systems, 7(9) : 915-930, September 1996.

[5] R. Chow and T. Johnson. Distributed Operating Systems & Algorithms, Chapter 7, Addison-Wesley, 1997.

[6] R. Bianchini, T. J LeBlanc, and J Veenstra, "Eliminating Useless Message in Write-Update Protocols on Scalable Multiprocessors," Technical report, 539, Univ. of Rochester, Dept. of Computer Science, October 1994.

[7] W. J. Bolosky and M. L. Scott, "False Sharing and its Effect on Shared Memory Performance," Proceedings of the USENIX Symposium on Experience with Distributed and Multiprocessor Systems(SEDMS IV), pp.57-71, September 1993.

[8] S. J. Eggers and T. E. Jeremiassen, "Eliminating False Sharing," Proceedings of the 1991 Int'l Conf. on Parallel Processing, Vol.I(Architecture), pp.377-381, August 1991.

[9] J. W. Lee and Y. Cho, "An Effective Shared Memory Allocator for Reducing False Sharing in NUMA Multiprocessors," Proceedings of 1996 IEEE 2nd Int'l Conf. on Algorithms & Architectures for Parallel Processing(ICA3PP '96), pp.373-382, June 1996.

[10] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy and P. Stenstrom, "The Detection and Elimination of Useless Misses in Multiprocessors," In Proc. of the 20th Int'l Symp. on Computer Architecture, pp.88-97, 1993

[11] V. Khera, R. P. LaRowe Jr., and C. S. Ellis, "An Architecture-Independent Analysis of False Sharing," Technical Report CS-1993-13, Duke Univ., Dept. of Computer Science, October 1993

[12] E. Markatos and C. Chronaki, "Trace-Driven Simulation of Data-Alignment and other Factors affecting Update and Invalidate Based Coherent Memory," Proc. of Int'l Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems, pp.44-52, January 1994

[13] J. Torrellas, M. S. Lam, and J. L. Hennessy, "Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates," Proceedings of the 1990 Int'l Conf. on Parallel Processing, Vol.II (Software), pp.266-270, August 1990.

[14] T. E. Jeremiassen and S. J. Eggers, "Reducing False Sharing on Shared Multiprocessors through Compile Time Data Transformations," Fifth ACM SIGPLAN Symposium Principles and Practice of Parallel Programming, pp.179-188, July 1995.

[15] J. B. Rothman and A. J. Smith, "Analysis of Shared Memory Misses and Reference Patterns," Technical Report UCB/CSD-99-1064, U. C. Berkeley, Computer Science Division, Sept. 1999.

[16] D. L. Black et al., "Competitive Management of Distributed Shared Memory," Proceedings of the 34th IEEE Computer Society International Conference, pp.184-190, March 1989.

[17] W. J. Bolosky et al., "Numa Policies and Their Relation to Memory Architecture," Proceedings of the 4th International Conference on Architectural Support for Programming Language and Operating Systems, pp.212-221, April 1991

[18] Y. Zhou et al., "Relaxed Consistency and Coherence Granularity in DSM Systems : A Performance Evaluation," Proc. of the 6th ACM Sym. on Principles and Practice of Parallel Programming, June 1997. (IEEE TPDS에 제출 중)

[19] L. Iftode et al., "Shared Virtual Memory with Automatic Update Support," Proc. of the Int'l Conf. on Supercomputing, June, 1999

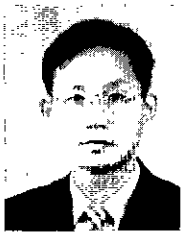
[20] J. P. Singh, W. Weber, and A. Gupta, "SPLASH : Stanford Parallel Applications for Shared-Memory," ACM SIGARCH Computer Architecture News, Vol.20, No.1, pp.5-44, March 1992

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs : Characterization and Methodological Considerations," Proceedings of the 22nd Annual Int'l Symposium on Computer Architecture, pp.24-36, June 1995

[22] J. E. Veenstra and R. J. Fowler, "Source code of shared memory allocator in MINT," University of Rochester.

[23] J. E. Veenstra, "MINT Tutirial and User Manual," Technical report, TR452, Computer Science Department, Univ. of Rochester, July 1993

[24] 이종우, 김문희 등, "분산 공유 메모리 시스템에서 동적 공유 메모리 할당 기법이 거짓 공유에 미치는 영향", 정보과학회논문지(A), 24(12), pp.1257-1269, December 1997.



조 성 제

e-mail sjcho@dankook.ac.kr

1989년 서울대 전자계산기공학과 졸업(학사)

1991년 서울대 대학원 컴퓨터공학과 졸업(공학석사)

1996년 서울대 대학원 컴퓨터공학과 졸업(공학박사)

1996~1997년 서울대 컴퓨터 신기술연구소 연구원

1997년~현재 단국대학교 자연과학부 조교수

관심분야 : 시스템 소프트웨어, 분산 시스템, 멀티미디어, 시스템 보안 등.