분산 공유 메모리 시스템에서 거짓 공유를 줄이는 호출지 추적 기반 공유 메모리 할당 기법

(Call-Site Tracing-based Shared Memory Allocator for False Sharing Reduction in DSM Systems)

이 종 우 †

(Jongwoo Lee)

요 약 거짓 공유는 공유 메모리 다중 처리기 시스템에서 여러 처리기들이 일관성 유지의 단위 메모리 영역을 공유함으로 인해 발생하는 현상으로써, 메모리 일관성 유지의 정확성에는 아무런 도움을 주지 못하면서 그 비용만 증가시키는 주요 요인이다. 특히 메모리 일관성 유지의 단위가 커질수록 그 피해가 더 커진다고 할 수 있다. 페이지-기반 분산 공유 메모리 시스템에서 거짓 공유를 줄이기 위해서는 공유 페이지에 할당되는 객체들의 특성을 미리 예측하여 참조 패턴이 상이한 객체들이 하나의 공유 페이지에 섞이는 것을 방지하는 것이 필수적이다. 본 논문에서는 병렬 응용 프로그램의 코드 내에서 공유 메모리 할당자를 호출한 위치를 추적하여 서로 다른 호출지에서 요청된 공유 객체가 같은 공유 페이지에 할당되는 것을 방지하는 호출지-추적 기반 거짓 공유 감소 기법(CSTallocator)을 제시한다. CSTallocator는 서로 다른 코드 위치에서 할당 요청된 공유 객체들은 각각 상이한 참조 패턴을 보일 것이라는 가정에 기반하고 있다. 이 기법의 효용성을 검증하기 위해 기존 거짓 공유 감소 할당 기법들의 성능과 비교한 결과 기존 방식에 비해 훨씬 더 많은 거짓 공유 폴트를 감소시킨다는 것을 알 수 있었다. 실험은 실제 병렬 응용에 기반한 실행-기반 시뮬레이션 기법을 사용하였다.

키워드 : 거짓 공유, 분산 공유 메모리, 거짓 공유, 분산 공유 메모리, 동적 메모리 할당, 성능 평가, 호출지 추적

Abstract False sharing is a result of co-location of unrelated data in the same unit of memory coherency, and is one source of unnecessary overhead being of no help to keep the memory coherency in multiprocessor systems. Moreover, the damage caused by false sharing becomes large in proportion to the granularity of memory coherency. To reduce false sharing in a page-based DSM system, it is necessary to allocate unrelated data objects that have different access patterns into the separate shared pages. In this paper we propose call-site tracing-based shared memory allocator, shortly CSTallocator. CSTallocator expects that the data objects requested from the different call-sites may have different access patterns in the future. So CSTallocator places each data object requested from the different call-sites into the separate shared pages, and consequently data objects that have the same call-site are likely to get together into the same shared pages. We use execution-driven simulation of real parallel applications to evaluate the effectiveness of our CSTallocator. Our observations show that by using CSTallocator a considerable amount of false sharing misses can be additionally reduced in comparison with the existing techniques.

Key words : False Sharing, Distributed Shared Memory, Dynamic Memory Allocation, Call Site Tracing

1. 서 론

분산 공유 메모리(distributed shared memory, 이하 DSM) 시스템에서는 메모리 모듈들이 여러 노드들에 분산되어 존재하기 때문에 데이타의 효과적 캐싱(caching)

· 본 연구는 숙명여자대학교 2005년도 교내연구비 지원에 의해 수행되었음

† 종신회원 : 숙명여자대학교 멀티미디어과학과 교수
bigrain@sookmyung.ac.kr

논문접수 : 2005년 1월 24일
심사완료 : 2005년 6월 3일

이 전체 시스템의 성능에 큰 영향을 미친다. 지역(local) 메모리 참조 보다 원격(remote) 메모리 참조에 훨씬 많은 비용이 들기 때문에 효과적인 캐싱을 통해 원격 메모리 참조 횟수를 줄일 수 있다면 결과적으로 메모리 참조의 평균 비용을 줄일 수 있게 되므로 시스템 전체의 성능을 향상시킬 수 있다[1]. DSM 시스템에서는 효과적 캐싱을 위해 주로 데이타 복사(replication)나 이주(migration) 기법을 이용하는데, 이는 각 처리기들이 참조하는 데이타 중 자주 참조하는 것들을 자신의 지역 메모리에 위치시켜 메모리 참조 비용을 줄이고자 하는 것이 그 목적이라고 할 수 있다[2]. 그러나 데이타 복사/이주 기법은 메모리 일관성 유지비용을 증가시키는 요인으로 작용하기도 하는데, 특히 데이타 복사 기법의 경우에는 동일 데이타의 여러 복사본들이 각 처리기의 지역 메모리에 상존하게 되므로 일관성 유지비용을 크게 증가시킨다(그림 1).

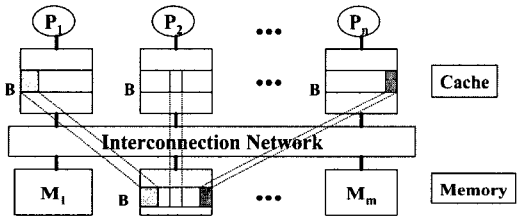


그림 1 DSM 시스템에서 메모리 복사의 예

거짓 공유(false sharing)는 공유 메모리 다중 처리기 시스템에서 여러 처리기들이 일관성 유지의 단위 메모리 영역을 공유함으로 인해 발생하는 현상으로써, 메모리 일관성 유지의 정확성에는 아무런 도움을 주지 못하면서 그 비용만 증가시키는 주요 요인이다[3-6]. 특히 PC-NOW DSM 시스템처럼 메모리 일관성 유지의 단위가 큰(일반적으로, 하나의 가상 페이지) 경우에는 그 피해가 더 커진다고 할 수 있다. [3-6]에 의하면 페이지 기반 분산 공유 메모리 시스템에서 발생하는 공유 메모리 폴트 중 거짓 공유 폴트가 차지하는 비중이 병렬 응용별로 차이는 있지만 대게 80% 안팎임을 알 수 있는데, 이는 거짓 공유가 분산 공유 메모리 시스템에서 메모리 성능을 떨어뜨리는 주 요인이라는 것을 의미한다. 본 논문에서는 동적 공유 메모리 할당자(dynamic shared memory allocator)를 통해 공유 데이타 영역을 생성하는 병렬 응용들을 대상으로 DSM 시스템에서 거짓 공유 감소에 도움을 주기 위한 동적 공유 메모리 할당 기법을 제시한다. 거짓 공유를 줄이는 동적 공유 메모리 할당을 위해서는 "향후 데이타 객체에 가해질 참조 패턴을 미리 예측"하는 것이 필수적인데, 본 논문에서는 **병렬 프로그램 내에서 데이타 객체를 요청한 위치를 추적하여 서로 다른 위치에서 요청된 데이타 객체가 같은 공유 페이지에 섞이지 않도록** 하였다. 이는 같은 위치에서 요청된 데이타 객체는 향후에 유사한 참조 패턴을 보일 것이라는 점에 착안한 것이다. 비록 이 참조 패턴 예측이 항상 정확하다고는 할 수 없으나 기존 기법들과 비교한 결과 거짓 공유 현상을 적게 유발한다는 것을 확인하였다. 한편 거짓 공유로 인해 발생하는 페이지 폴트(이하 거짓 공유 폴트)의 횟수를 측정하기 위해 본 논문에서는 SPLASH와 SPLASH II의 병렬 응용 프로그램들과 다중 처리기 구조 시뮬레이터인 MINT를 이용하였다.

본 논문의 구성은 다음과 같다. 2절에서는 기존 연구들을 살펴보고, 3절에서는 호출지 추적 기반 공유 메모리 할당 기법을 설계하고 구현한 내용을 제시한다. 4절에서는 성능 평가 결과를 보이며, 끝으로 5절에서는 결론과 향후 연구 과제를 제시한다.

2. 기존 연구

본 논문에서 기본 환경으로 설정하고 있는 DSM 시스템은 시스템 가상 페이지 단위로 메모리 일관성이 유지되는 페이지 기반 DSM 시스템인데, 이러한 시스템을 위한 동적 공유 메모리 할당자가 우선적으로 고려해야 할 점은 **요청된 데이타 객체를 어느 공유 페이지에 배치시키느냐** 하는 것이다. 만약 요청된 공간에 놓이게 될 데이타 객체의 특성 및 참조 패턴 등을 동적 공유 메모리 할당자가 미리 알 수 있다면 할당자는 앞서 언급한 거짓 공유의 원인을 최대한 없애는 방식으로 데이타 객체들을 공유 페이지에 배치하면 될 것이다. 예를 들어, 참조 패턴이 현저히 다른 데이타 객체들을 같은 공유 페이지에 배치하지 않거나, 또는 서로 관련이 없는 데이타 객체들이 같은 공유 페이지에 섞이지 않도록 하는 등의 기법을 이용하면 거짓 공유의 정도를 상당 부분 완화할 수 있을 것이다. 그러나 동적 공유 메모리 할당자가 요청된 공간에 놓일 데이타 객체의 특성 및 참조 패턴 등을 미리 알 수는 없으므로 [7]에서는 사용자가 제공한 힌트에 기반한 *typed allocation* 기법을 제안하기도 하였다. 이 방식에서는 프로그래머가 공유 메모리 요청 시 그 공간의 타입을 지정하도록 하였다. *Read-Only* 타입과 *Write-Mostly* 타입, 그리고 *Lock* 타입 등과 같이 할당받을 공간의 예상되는 참조 패턴을 사용자가 지정하도록 함으로써 서로 다른 타입의 데이타 객체들이 같은 공유 페이지에 섞이지 않도록 하였다. 그러나 이 방법에서는 동적 공유 메모리 할당자의 사용자 인터페이스가 달라지므로 기존 응용들도 소스 코드 수준에서 수정되어야 한다는 추가 비용이 발생한다. 본

연구에서는 동적 공유 메모리 할당자의 인터페이스는 변경시키지 않는다는 것을 가정하고 있기 때문에 이 연구는 본 연구와는 차이가 있다 하겠다. 게다가 아무리 숙달된 프로그래머라 할지라도 프로그램에서 요청하는 공유 공간에 대한 여러 처리기들의 참조 패턴을 미리 알기가 쉽지 않다는 단점도 존재한다.

*프로세스별 할당 방식*은 서로 다른 프로세스에 의해 요청된 데이타 객체들이 같은 캐쉬 라인에 섞이지 않도록 할당하는 방식이다[3]. 이 기법에서는 서로 다른 프로세스가 요청한 데이타 객체들을 별도의 캐쉬 라인에 배치함으로써 관련 없는 데이타 객체나 참조 패턴이 다른 데이타 객체들이 같은 캐쉬 라인에 섞일 가능성을 줄일 수 있다고 가정하고 있으며, 실험을 통해 효과가 있음을 보이고 있다. 이 기법은 공유 메모리 할당을 여러 처리기가 골고루 요청하는 경우에는 효과가 있지만, 공유 메모리 할당을 어떤 한 프로세스가 전담하는 경우에는 당연히 아무런 효과가 없다[8]. 본 연구의 실험에 사용된 병렬 응용프로그램들에서도 역시 한 프로세스가 공유 메모리 할당을 전담하고 있기 때문에 이 기법 또한 본 연구와의 비교 대상으로는 적절치 않다고 할 수 있다.

다음은 *객체-크기별 할당 방식*을 들 수 있다[5,6,8]. 객체-크기별 할당 방식이란 하나의 공유 페이지 내에 크기가 다른 데이타 객체가 섞이지 않도록 할당하는 방식을 의미한다. 즉, 한 공유 페이지 내에는 같은 크기의 데이타 객체들만 할당되게 함으로써 이질적인 데이타 객체들이 하나의 공유 페이지에 섞이는 것을 최대한 방지하고자 한 것이다. 객체의 특성이나 향후 참조 패턴을 예측하는 근거로 객체-크기 정보를 이용함으로써 할당자 인터페이스의 사용자 투명성(transparency)을 유지함과 동시에 거짓 공유 폴트도 줄일 수 있다는 것이 이 연구들의 결론이라 하겠다. 특히, [8]에서는 객체-크기별 할당 방식 외에도 태그 분리 할당 방식, 다중 페이지 걸침 최소화 방식 등도 거짓 공유 감소에 도움을 준다는 결과를 보이고 있다. 하지만 객체의 크기가 향후의 참조 패턴을 완전하게 대변하는 것은 아니므로 이를 보완할 수 있는 추가적인 예측 기법이 필요하다 할 것이다. 본 연구에서는 이 기법을 주요 비교 대상으로 설정하였는데 그 이유는 이 연구에서 가정하고 있는 것이 본 연구의 가정과 거의 유사하기 때문이다. 즉, 동적 공유 메모리 할당자의 인터페이스를 수정하지 않아야 하고 아울러 한 프로세스가 공유 메모리 할당을 전담하든 그렇지 않든 효과적이어야 한다는 가정이 본 연구와 일치한다.

기존 연구에서 제시된 여러 기법들을 조합한 할당 방식을 제시한 연구들도 있었는데, [9]에서는 *프로세스별 할당 방식과 다중 페이지 걸침 최소화 기법을 조합한 공유 메모리 할당 방식*을 제안하고 있다. 이 연구에서는 데이타 객체의 크기가 가상 페이지 크기보다 작은 경우와 큰 경우를 구분한다. 즉, 객체가 페이지 크기보다 작은 경우에는 동일 페이지에 서로 다른 처리기가 요청한 객체들이 섞이지 않도록 하고, 데이타 객체가 페이지 크기보다 큰 경우에는 객체를 여러 페이지 경계에 걸치지 않도록 정렬함으로써 하나의 객체가 여러 페이지에 걸쳐서 할당되는 경우를 최소화하고자 하였다. 기존 기법들을 조합만하더라도 거짓 공유 감소 효과가 좋을 수 있음을 보였다는 점에서 의미 있는 연구이기는 하지만 객체에 대한 향후 참조 패턴을 예측하는데 필요한 새로운 기법 제시에는 미흡했다고 할 수 있다.

또한, *사용될 병렬 응용프로그램의 메모리 참조 패턴을 미리 분석한 후 그에 가장 잘 부합하는 공유 메모리 할당 정책을 제시한 연구*도 있었는데, [10]이 그 예이다. [10]에서는 실험에 사용될 병렬 응용프로그램들의 메모리 참조 패턴을 분석한 결과, 공유 객체들은 순차적으로 할당되며 서로 다른 참조 패턴을 보일 때가 많음을 발견하였다. 아울러, 동일 크기의 객체들이 처리기 수만큼 연속적으로 요청되는 경우 각 객체는 특정 처리기에 의해서만 참조되며, 동일 크기의 객체들이 처리기 수보다 훨씬 더 많이 연속적으로 할당되는 경우에는 연속적인 2개 이상의 객체가 특정 처리기에 의해서만 참조된다는 것도 발견하였다. 이를 토대로 참조 패턴이 다른 객체들을 서로 분리하여 서로 다른 페이지에 배치함으로써 거짓 공유를 줄이고자 한 것이다. 특정 병렬 응용프로그램에 특화된 정책을 고안했으므로 가장 좋은 성능을 보일 것으로 판단되기는 하지만, 이 방식에서는 실험에 사용된 병렬 응용 프로그램의 실행 궤적을 미리 추출한 후에 이에 대한 최적 할당 방식을 찾아내기 때문에 실험에 사용되지 않은 다른 특성의 병렬 응용프로그램들에 대해서는 효과적이지 못할 수도 있을 것이다.

DSM 환경 하에서의 연구는 아니지만 [11]에서는 처리기 별 할당 방식을 응용하여 할당된 메모리 공간에 대해 첫 페이지 폴트가 발생하는 순간 페이지 폴트를 유발한 처리기의 로컬 메모리로 할당된 메모리를 옮긴 후 고정시키는 기법을 제시하였다. 이 기법은 일종의 "fault and pinning" 방식으로써 해당 메모리를 실제로 사용할 처리기의 로컬 메모리에 할당하여 메모리 참조 시간을 줄이고자 한 것이 목적이라 하겠다. 하지만 이 연구는 거짓 공유 감소에 초점을 맞춘 것이 아니라 다중처리기 시스템에서 메모리 성능 향상을 위한 것이어서 커널의 수정까지 필요하게 되므로 본 연구와는 그 방향이 다소 다르다고 할 수 있다.

한편 [12-14]에서는 다중스레드 환경에서 동적 공유 메모리 할당자의 확장성 향상 연구를 진행하는 과정에

서 처리기 별로 힙(heap)을 분리하면 거짓 공유도 줄일 수 있다는 부수적인 결과를 제시하고 있다. 이들 연구는 비교적 최신 연구이긴 하지만 거짓 공유 감소를 주 목표로 하고 있지 않을 뿐만 아니라 처리기별 할당 기법이 거짓 공유 감소에 도움이 된다는 것은 이미 밝혀진 사실이기 때문에 본 연구와 비교하기에는 적절치 않을 연구들이라 하겠다.

이와 같은 기존 연구들에서 공통적으로 발견할 수 있는 사항은 공유 객체에 가해질 미래의 참조 패턴을 얼마나 효과적으로 예측할 수 있느냐가 거짓 공유 감소에 큰 영향을 미친다는 점이다. 서로 상이한 참조 패턴을 보일 것으로 예상되는 공유 객체들을 가능한 한 서로 다른 메모리 일관성 유지 단위 블록에 배치해야 하는 것이다. 본 논문에서 제시하는 호출지 추적(cal-site tracing) 기법(이후 CSTallocator라 칭함)에서는 공유 객체를 요청한 코드 상의 위치, 즉 병렬 응용프로그램 내에서 공유 객체를 요청한 곳의 명령 포인터(instruction pointer) 값을 기준으로 향후 참조 패턴을 예측한다. 이는 요청 호출지가 서로 다른 객체는 향후에도 서로 다른 참조 패턴을 보일 것이라는 가정에 기반을 두고 있다. 이 기법은 프로그래머에게 추가적인 부담을 주지 않는다는 점에서는 기존 기법들과 유사하지만 공유 메모리 할당 함수에 주어지는 명시적 정보가 아닌 프로그램에 내재되어 있는 묵시적 정보를 사용한다는 점에서 기존 기법들과 차별된다. 또한, 공유 객체가 요청된 프로그램 내에서의 위치는 그 객체에 대한 향후 참조 패턴을 유추하기에 유용한 힌트가 되는데, 그 이유는 대게의 병렬 응용프로그램에서는 공유 객체의 용도별로 서로 다른 위치에서 할당 함수를 호출하는 것이 일반적이기 때문이다. 물론 공유 메모리 할당 함수 내에서 자신이 호출된 프로그램 내에서의 위치를 추적하는 데 드는 비용은 인자로 주어진 할당 크기나 함수를 호출한 처리기 식별자 같은 정적 정보를 얻는데 드는 비용보다 크다. 하지만 호출지 추적은 객체 별로 최초 할당 시 한번만 수행하므로 초기화 비용 외에 런타임에 추가로 드는 비용은 기존 방식들과 마찬가지로 없다고 할 수 있다.

3. CSTallocator 설계 및 구현

어떤 한 마스터 프로세스가 공유 메모리 할당을 전담하건 아니건 간에 서로 다른 용도의 공유 객체는 대게의 경우는 그 크기가 서로 다른 것이 일반적이다. [5,6,8]은 이러한 특성을 활용한 공유 메모리 할당 기법이라고 할 수 있다. 하지만 객체-크기보다 그 객체의 용도를 좀 더 정확하게 유추할 수 있는 기준이 있는데, 그 중 하나가 바로 그 공유 객체가 코드 상의 어느 위치에서 할당 요청되었는가 하는 것이다. 대부분의 병렬 응용프로그램에서 여러 프로세스(또는 스레드)가 공유할 데이타 영역을 할당할 때에는 그 영역의 용도 별로 서로 다른 코드 상의 위치에서 할당 요청 함수를 호출하기 때문이다. 설령 그 크기가 같다 하더라도 서로 다른 위치에서 할당 요청된 공유 객체에 가해지는 향후의 참조 패턴이 서로 차이가 날 것이라는 것은 쉽게 예상할 수 있다. 이 예상이 빗나가려면 같은 위치에서 요청된 여러 데이타 객체가 향후에 전혀 다른 용도로 사용되어 그 참조 패턴이 서로 다르게 되는 경우가 자주 발생해야 하는데, 이는 프로그램 코드의 어느 특정 부분의 용도가 실행 시에 동적으로 변하지 않는 한 자주 발생하기 힘든 상황이라 하겠다.

다음 그림 2는 병렬 응용프로그램 내에서 공유 객체 요청 함수를 호출한 위치, 즉 호출지 별로 공유 객체를 별도의 페이지에 위치시킨 예를 보이고 있다. 결국 서로

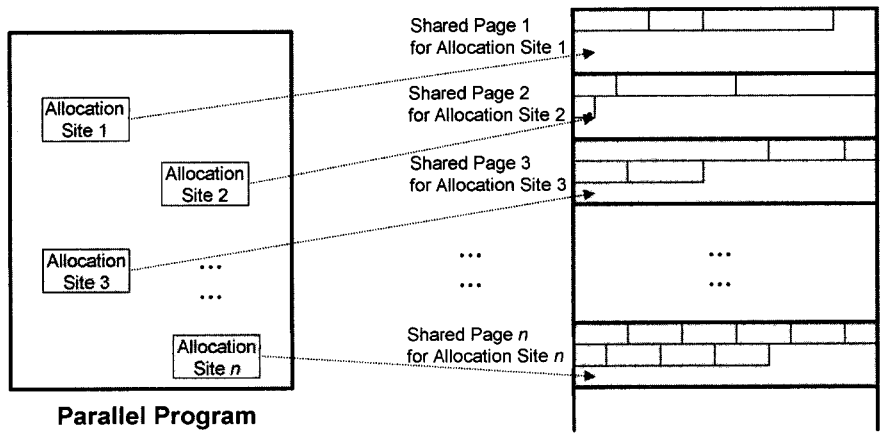


그림 2 CSTallocator에서 호출지 별 공유 객체 배치 방법

다른 호출지에서 요청된 공유 객체가 같은 공유 페이지에 섞이지 않게 하는 것이 핵심 아이디어라고 할 수 있는데, 이 그림에서도 알 수 있듯이 그 크기는 다르더라도 같은 호출지에서 요청된 객체들이 서로 섞이는 것은 허용하고 있다. 이는 기존 연구 중 객체-크기별 할당 방식과의 정확한 비교를 위해 의도적으로 허용한 것이다. 즉, 호출지 추적 기법만 사용했을 때와 객체-크기별 할당 기법만을 사용했을 때의 성능을 비교하기 위해 이 두 방식을 조합한 방식은 실험에 포함시키지 않았다. 물론 호출지와 객체-크기를 모두 고려하여 하나라도 다르면 섞이지 않도록 하는 할당 방식을 사용한다면 당연히 호출지 추적만을 사용한 기법이나 객체-크기별 할당 방식만을 사용한 기법보다 좋을 것으로 예상되지만, 본 논문에서는 호출지 추적 기법이 성능에 어떤 영향을 보이는지가 관심이므로 조합한 방식에 대한 더 이상의 언급은 생략하고 향후 연구로 남겨두도록 한다.

3.1 호출지 추적 기법

호출지 별 공유 메모리 할당을 위해서는 우선 병렬 응용프로그램 내에 서로 다른 호출지가 어디 어디인지를 규명해야 하는데, 물론 이는 실행 중에 동적으로 이루어져야 하고 아울러 공유 메모리 할당 함수 내에서 추가적인 형식인자 없이 투명하게 이루어져야 한다. 이를 위해 본 논문에서는 *함수 호출 경로 역추적(call path back tracking)* 기법을 사용하였다. 즉, 공유 메모리 할당 함수에서 자신을 호출한 곳이 어디인지를 알아내기 위해 프로세스(또는 스레드) 스택에 쌓여있는 활성 레코드(activation record)를 역추적함으로써 프로그램의 시작점인 main() 함수로부터 현재 자신까지의 함수 호출 경로를 알아내는 것이다. 함수 호출과 복귀를 위해서는 스택에 복귀 주소가 저장되기 마련이고, 함수 내에서 사용하는 지역 변수나 인자를 위한 스택 영역 크기를 알아내면 이 복귀 주소를 얻어낼 수 있다. 이러한 스택 역추적을 계속하다보면 결국 main() 함수를 만나게 되는데 main() 함수를 만나면 스택 추적을 종료하게 된다. 어느 호출지에서 함수 호출 역추적을 한 결과 "share_malloc() ← B() ← A() ← main()"의 순서로 추적되었다면 이 호출지의 식별자는 "A→B"라는 표현으로 정의된다. 호출지 ID에 main() 함수와 공유 메모리 할당 함수인 share_malloc()은 생략해도 상관없는데 이는 모든 *호출지 ID*에 공통으로 들어갈 수밖에 없기 때문이다. 그림 3은 호출지 추적의 예를 보이고 있다. 그림에서 볼 수 있듯이 이 추적을 통해 S_N이라는 호출지가 규명되고 S_N은 "A→B"라는 호출지 ID로 기록된다. 그런 후 공유 페이지 할당 시 호출지 ID가 서로 다른 객체들과 섞이지 않게 할당한다.

위의 예에서 한 가지 고려해야 할 점은 *함수 호출 경로 역추적의 깊이*를 어느 정도로 해야 할 것이냐 이다. 위에서는 main() 함수까지 추적한 후 호출지 ID를 부여한다고 설명했으나 병렬 응용프로그램에 따라서는 main()까지 가지 않아도 모든 호출지를 분명하게 구분할 수 있는 경우가 있을 것이다. 즉, *깊은 추적(deep tracing)*과 *얕은 추적(shallow tracing)*을 동적으로 선택할 수 있다면 불필요한 스택 추적으로 인한 오버헤드를 막을 수 있다는 것인데, 이는 구현할 수 없다는 단점이 있다. 동적 공유 메모리 할당 함수 내에서 함수 호출 경로 역추적을 할 때 어디까지 해야 모든 호출지 구별이 가능한지를 미리 알 방법이 없기 때문이다. 다만, 성능 평가 시 추적의 깊이(depth)를 정적으로 변화시켰을 때 거짓 공유 감소에 어떤 영향을 미치는지는 알 수 있을 것이다. 이를 위해 본 논문에서는 호출 *체인 길이 N(length-N call chain)*을 정의하였는데, 호출 체인 길이란 공유 메모리 할당 함수에서 시작하여 main() 함수로 가는 완전 함수 호출 경로 중 처음 N 개의 함수 호출까지의 경로를 의미한다. 예를 들어, 호출 체인 길이 1이란 share_malloc()을 호출한 함수 B() 하나만을 구분하겠다는 의미이다. 또한, 호출 체인 길이 2란 share_malloc()을 호출한 함수 B()와 B()를 호출한 A()까지를 호출지 ID에 포함시키겠다는 의미이다. 호출지 체인 길이의 의미에서 보면 호출지 체인 길이가 길수록 깊은 추적을 수행한다는 의미가 되며 이 경우 얕은 추

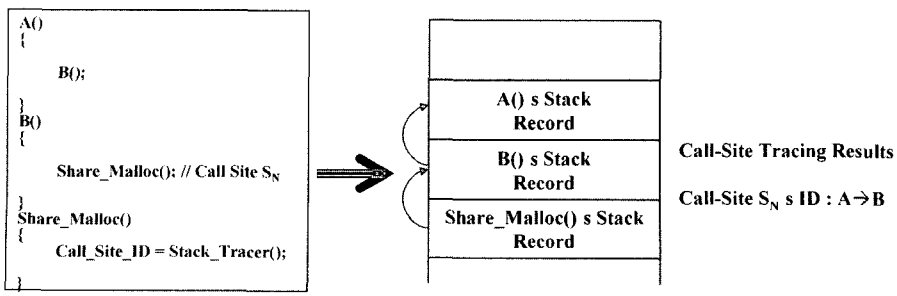
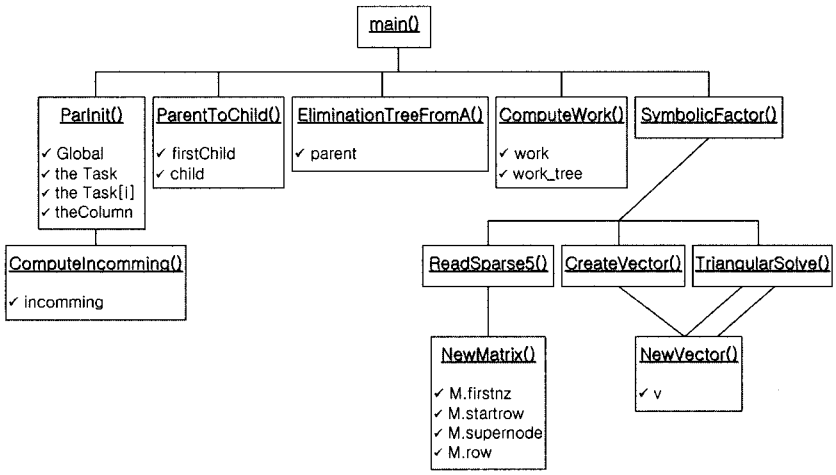


그림 3 호출지 추적을 통해 호출지 ID를 찾는 예

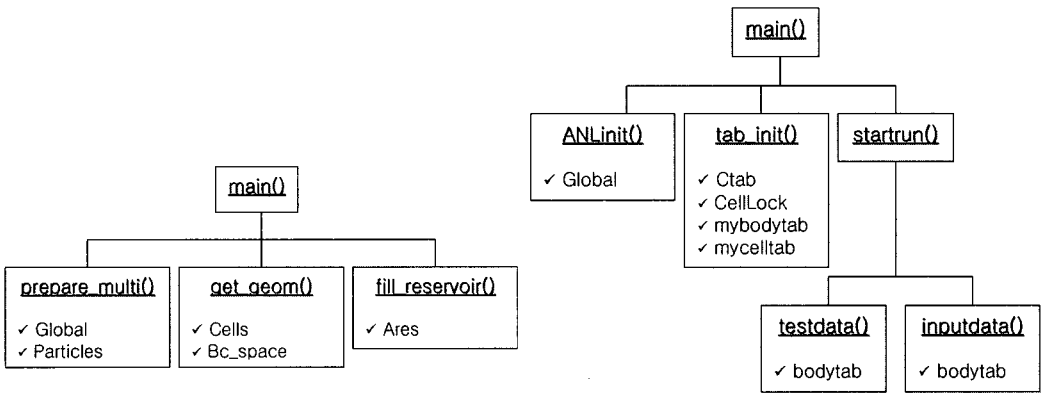
적보다 세세한 구분을 할 수 있게 되므로 거짓 공유의 가능성을 좀 더 줄일 수 있을 것으로 예상할 수 있다.

3.2 병렬 응용프로그램 별 호출지 추적 예

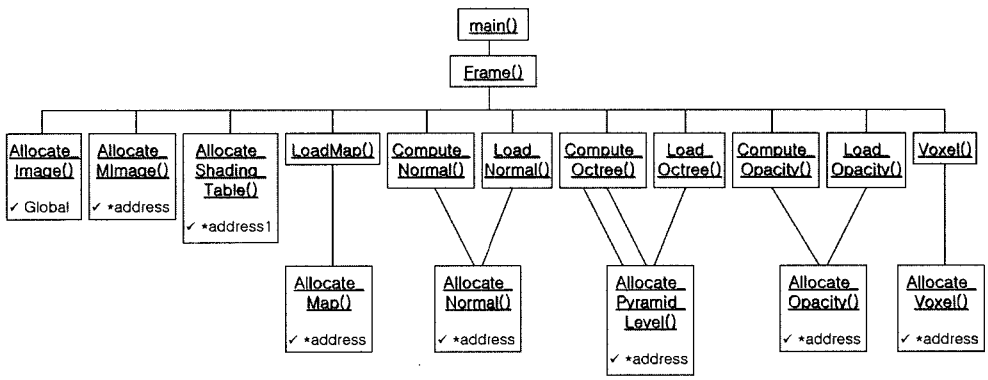
지금까지 설명한 호출지 추적 기법에 따라 본 논문에서 사용한 병렬 응용프로그램들의 공유 메모리 할당 함수 호출 궤적을 추적한 결과는 그림 4와 같다. 그림 4에



(a) Cholesky의 호출지 추적 결과(최대 호출 체인 길이 = 2)



(b) Mp3d의 호출지 추적 결과(최대 호출 체인 길이 = 1) (c) Barnes의 호출지 추적 결과(최대 호출 체인 길이 = 1)



(d) Volrend의 호출지 추적 결과(최대 호출 체인 길이 = 2)

그림 4 실험에 사용된 4개의 병렬 응용프로그램에 존재하는 호출지 추적 결과

보인 추적 결과는 실행 중에 동적으로 발견되는 호출지 ID들을 함수 별로 정리해놓은 것인데, 어떤 한 함수 내에서 공유 메모리 할당 함수를 호출하는 곳이 여러 곳인 경우도 있고 한 곳만 있는 경우도 있음을 알 수 있다. 편의상 호출지 ID를 포함하고 있지 않은 함수는 표시하지 않았다.

그림 4에 있는 호출지 추적 다이어그램에서 알 수 있는 점은 대부분의 병렬 응용프로그램이 다양한 코드 상의 위치에서 공유 메모리를 요청하고 있다는 점이다. 특히, 어떤 함수 내에서 공유 메모리 할당 요청을 전담하고 있다고 해도 그 함수를 호출하는 곳이 여러 군데라면 그 각각은 서로 다른 용도의 공유 객체를 할당하려 했을 가능성이 높다는 것을 쉽게 예상할 수 있다. 이처럼 다양한 호출 궤적 상에 놓여 있는 공유 객체 할당 요청들을 서로 분리하여 처리해줌으로써 프로그래머가 다른 의도를 가지고 할당받은 여러 공유 객체들을 서로 다른 공유 페이지에 할당할 수 있게 한 것이 본 논문이 제시하는 핵심적인 아이디어라고 할 수 있다. 결국 이는 상이한 참조 패턴을 가진 여러 공유 객체들이 같은 공유 페이지에 섞이는 것을 방지할 수 있게 해주기 때문에 거짓 공유의 가능성을 많이 줄여줄 것으로 기대된다. 또한, 이미 언급했듯이 추적이 깊이가 깊어지면 호출지 추적에 드는 비용이 커질 수 있는데 그림 4에서 볼 수 있듯이 대부분의 응용에서 발견된 호출지의 깊이가 그리 깊지 않음을 알 수 있다. 실험에 사용된 4개의 병렬 응용프로그램들에서 발견된 호출 체인 길이의 최대값이 2를 넘지 않기 때문에 비교적 얕은 추적만으로도 모든 호출지를 구분할 수 있었다.

4. 성능 평가

이번 절에서는 실험 환경에 대해 설명하고, 3절에서 제시한 호출지 추적 기반 공유 메모리 할당 기법이 거짓 공유에 어떠한 영향을 미치는지에 대해 기존 객체-크기별 할당 기법과 비교한 실험 결과를 보인다. 이 실험에서 객체-크기별 할당 기법과 비교하는 이유는 객체-크기별 할당 기법이 동적 공유 메모리 할당 기법에서는 가장 널리 사용되는 투명(별도의 프로그래머 부담이 없는) 할당 기법이기 때문이기도 하지만 투명성을 유지하면서도 동적 공유 메모리 할당자 수준에서 거짓 공유 감소를 시도한 유일한 기존 연구이기 때문이다. 그렇기 때문에 2절에서 제시한 대부분의 기존 연구들에서도 자신들의 기법을 객체-크기별 할당 기법의 성능과 비교하고 있는 실정이다.

4.1 실험 환경

본 논문에서는 16개의 노드로 구성된 DSM 시스템을 시뮬레이션하기 위해 실행-구동형(execution-driven) 시뮬레이션 기법을 이용하였다. 사용된 시뮬레이터는 크게 전반부(front-end) 시뮬레이터와 후반부(back-end) 시뮬레이터로 구성되어 있는데, 전반부 시뮬레이터는 주어진 병렬 응용 프로그램의 실행 코드를 해석하여 각 처리기들의 실행을 시뮬레이션 하는 기능을 한다. 우리는 전반부 시뮬레이터로 MINT(Mips INTerpreter)[15, 16]를 이용하였다. 후반부 시뮬레이터는 MINT의 출력을 바탕으로 메모리 관리 시스템의 여러 정책들을 시뮬레이션 한다. MINT는 주어진 실행 코드를 해석하여 데이타 참조를 위한 모든 메모리 참조 시마다 후반부 시뮬레이터에서 제공한 일정 함수들을 호출하기 때문에 시뮬레이션 하고자 하는 메모리 관리 정책이나 메모리 일관성 프로토콜은 후반부 시뮬레이터에 의해 구현된다.

한편 본 실험에서 사용된 병렬 응용 프로그램들은 Cholesky와 Mp3d, Barnes, 그리고 Volrend 인데, 이들은 스탠포드 대학의 SPLASH 사이트[17]와 SPLASH II 사이트[18]에서 얻을 수 있는 병렬 벤치마크 프로그램들에서 임의적으로 선택한 것이다. 이들 병렬 응용프로그램에 대한 자세한 내용은 [17,18]를 참조하면 된다. 실험은 기존의 객체-크기별 할당 방식을 사용했을 때와 CSTallocator를 사용했을 때 발생하는 거짓 공유 폴트의 횟수를 비교하는 방식으로 실시하였다. 아울러 CSTallocator를 사용할 경우 호출 체인 길이 N 값을 변화시켜가며 거짓 공유 폴트 횟수에 어떤 영향을 주는지도 측정하였다.

4.2 실험 결과

실험에 사용된 4 개의 병렬 응용프로그램 각각에 대해 본 논문에서 제시한 CSTallocator를 사용했을 때와 기존 객체-크기별 할당 방식을 사용했을 때 발생하는 거짓 공유 폴트 횟수 측정 결과는 표 1과 같다. 이 표의 두 번째 컬럼에 있는 "버켓의 개수"는 공유 메모리 할당 함수가 반복적으로 호출되는 과정에서 발견한 고유한 할당 슬롯의 개수를 의미하는데, 객체-크기별 할당 방식에서는 프로그램에서 요청한 객체-크기가 몇 가지 종류인지를 나타내고 CSTallocator에서는 할당자가 추적해낸 호출지 ID의 개수를 의미한다. 공유 메모리 할당자에서는 이 버켓 별로 별도의 포인터를 사용해 메모리 객체를 연결 리스트로 관리하게 된다. 즉, 같은 버켓 포인터에 연결된 공유 페이지에는 호출지 ID가 같거나 요청 크기가 같은 데이타 객체들만이 할당된다. 따라서 버켓의 개수가 많다는 것은 그 만큼 세밀하게 객체의 특성을 구분했다는 의미가 된다. 대개의 경우 구분 기준이 합리적이기만 하다면 버켓의 개수가 많을수록 거짓 공유 감소 가능성은 커질 것이다. 하지만 응용에 따라서는 버켓의 개수가 다른 방식보다 많다 할지라도 거짓 공유 폴트의 횟수가 줄지 않을 수도 있는데, 이는 할당

자가 사용한 버켓 구분 알고리즘이 그 응용에 대해서는 별다른 효과를 보지 못했음을 의미한다.

표 1에 나타난 실험 결과는 실험에 사용된 4개의 병렬 응용프로그램 모두에 대해 기존 객체-크기별 할당 방식에 비해 본 논문에서 제시한 CSTallocator가 거짓 공유 감소에 훨씬 더 효과적이라는 것을 보여주고 있다. 이는 객체-크기보다 객체를 요청한 프로그램 내에서의 위치가 그 객체에 가해질 향후 참조 패턴을 좀 더 효율적으로 예측할 수 있게 해주는 힌트가 될 수 있음을 의미한다. 버켓의 개수가 객체-크기별 할당 방식에 비해 줄어든 Mp3d나 Barnes의 경우에도 거짓 공유 폴트 횟수가 감소했다는 사실이 이를 입증하고 있다. 또한,

표 1 CSTallocator와 객체-크기별 할당 방식의 성능 비교 (페이지 크기는 4KB를 사용)

(a) Cholesky

구분 / 적용 기법	버켓의 개수	거짓 공유 폴트 횟수	감소율 (%)
객체-크기별 할당 방식	10	44,717	
CSTallocator (호출 체인 길이 1)	15	40,921	8.5
CSTallocator (호출 체인 길이 2)	17	36,599	18.2

(b) Mp3d

구분 / 적용 기법	버켓의 개수	거짓 공유 폴트 횟수	감소율
객체-크기별 할당 방식	8	6,147,589	
CSTallocator (호출 체인 길이 1)	5	5,754,143	6.4

(c) Barnes

구분 / 적용 기법	버켓의 개수	거짓 공유 폴트 횟수	감소율
객체-크기별 할당 방식	27	5,805,705	
CSTallocator (호출 체인 길이 1)	7	5,104,413	12.1

(d) Volrend

구분 / 적용 기법	버켓의 개수	거짓 공유 폴트 횟수	감소율
객체-크기별 할당 방식	11	953	
CSTallocator (호출 체인 길이 1)	8	931	2.3
CSTallocator (호출 체인 길이 2)	12	883	7.3

Cholesky와 Volrend에서 볼 수 있듯이 예상대로 호출 체인 길이가 길수록 거짓 공유 감소율이 증가하고 있음을 알 수 있는데, 앞서 언급했지만 이는 호출지를 좀 더 세세하게 구분할수록 객체에 대한 향후 참조 패턴을 좀 더 정확하게 예측하는 데에 도움을 주기 때문인 것으로 판단된다. 한편 버켓의 수가 줄었음에도 거짓 공유 감소율이 높아졌다는 것은 CSTallocator가 거짓 공유 감소 성능 측면 뿐만아니라 공간 효율성 측면에서도 객체-크기별 할당 방식에 뒤지지 않는다는 것을 의미한다.

4.3 공간 효율성 분석

지금까지의 실험 결과는 CSTallocator가 거짓 공유 폴트를 줄이는데 효과적이라는 것을 보이고 있다. 그러나 CSTallocator가 유발하는 공간 오버헤드가 어느 정도인지를 살펴보는 것도 필수적일 것이므로 본 절에서는 CSTallocator와 기존 객체-크기별 할당 기법이 보이는 공간 오버헤드를 분석한다. 여기서 공간 오버헤드란 제시된 기법이 기존 기법에 비해 추가적으로 사용하는 메모리 양을 의미한다. 좀 더 정확한 효율성 분석을 위해서는 시간 효율성 측정도 필요하나 본 논문에서는 시뮬레이션을 사용했기 때문에 각 할당 기법들의 실제 실행 시간을 측정하는 것이 불가능하였으므로 언급하지 않기로 한다.

먼저 객체-크기나 호출지 ID 같은 버켓을 사용하지 않는 일반 공유 메모리 할당자, 즉 할당 요청 순서대로 공유 페이지에 섞이도록 할당하는 경우의 공간 효율성을 분석하면 다음과 같다. 일반 할당 방식에서 i 번째 할당 요청의 크기를 s_i라고 할 때 할당 요청 스트림 S는 다음과 같이 표현될 수 있다[5,6].

$$S = \{ s_1, s_2, \cdots, s_n \} \quad (1)$$
$$s_i = \text{requested size of } i-th \text{ allocation} (1 \le i \le n).$$
$$n = \text{total \# of requests.}$$

이와 같은 할당 요청 스트림을 기존 할당 방식으로 처리하기 위해 필요한 페이지 수는

$$\#\ \text{of pages required} = \left\lceil \frac{\sum_{i=1}^{n} s_i}{\text{pagesize}} \right\rceil \quad (2)$$

이 된다. 반면 CSTallocator나 객체-크기별 할당 방식에서 각 할당 요청 스트림은 요청 순서를 무시할 경우 다음과 같이 표현될 수 있다.

$$S = \{ S_{bucket_1}, S_{bucket_2}, \cdots, S_{bucket_k} \} \quad (3)$$
$$S_{bucket_k} = \text{set of allocations with bucket ID } bucket_k.$$
$$S_{bucket_1} \cap S_{bucket_2} \cap \cdots \cap S_{bucket_k} = \varnothing$$
$$BS = \{ bucket_1, bucket_2, \cdots, bucket_k \}$$
$$: \text{set of unique bucket IDs}$$

이와 같은 할당 요청 스트림을 버켓 별로 할당하기

위해 필요한 페이지 수는

$$\#\ \text{of pages required} = \sum_{bucket_k \in BS} \left\lceil \frac{|S_{bucket_k}| \times AvgSize_{bucket_k}}{\text{pagesize}} \right\rceil$$

$$Here,\ AvgSize_{bucket_k} = Average\ size\ of\ each\ allocation\ request\ heading\ \text{for}\ bucket_k \tag{4}$$

이 된다.

식 (2)와 (4)를 비교해 보면 그 차이가 단지 올림 함수(ceiling)의 횟수에 의존한다는 것을 알 수 있다. 즉, 식 (2)에서는 올림 함수가 한번만 적용되지만 식 (4)에서는 집합 BS의 크기($|BS|$)에 해당하는 횟수만큼 적용된다는 것을 알 수 있다. 이는 일반 할당 방식에 비해 추가로 요구되는 페이지의 수가 객체-크기별 할당 방식에서는 객체-크기 종류의 수 이하이고, CSTallocator에서는 호출지 ID 개수 이하라는 것을 의미한다. 따라서 다음 식이 성립한다.

$$\text{Space Overhead} = \left(\sum_{bucket_k \in BS} \left\lceil \frac{|S_{bucket_k}| \times AvgSize_{bucket_k}}{\text{pagesize}} \right\rceil \right) - \left\lceil \frac{\sum_{i=1}^{n} s_i}{\text{pagesize}} \right\rceil \leq |BS| \tag{5}$$

위의 식에서 알 수 있는 중요한 사실은 사용한 기법이 무엇이건 간에 최대로 구분 기준에 의한 버켓 개수만큼의 공유 페이지가 공간 오버헤드로 작용하게 된다는 사실이다. 식 (5)를 근거로 CSTallocator와 객체-크기별 할당 방식의 공간 효율성을 비교해보면 표 2와 같다.

표 2 CSTallocator의 공간 효율성
(페이지 크기 : 4KB, CSTallocator에서는 최대 호출 체인 길이를 기준으로 함)

병렬 응용프로그램 이름 (괄호 안은 일반 할당 방식에서의 총 페이지 수)	추가로 소요되는 페이지 수 (괄호 안은 공간 오버헤드(%))	
	객체-크기별 할당 방식	CSTallocator
Cholesky (738)	10 (1.36)	17 (2.30)
Mp3d (553)	8 (1.45)	5 (0.90)
Barnes (308)	27(19.85)	7 (5.15)
Volrend (441)	11 (2.49)	12 (2.72)

표 2에서 볼 수 있듯이 CSTallocator의 공간 효율성은 Mp3d와 Barnes에서는 객체-크기별 할당 방식보다 더 좋았지만 Cholesky와 Volrend에서는 다소 떨어지는 등 객체-크기별 할당 방식과 큰 차이를 보이고 있지 않음을 알 수 있었다. 또한 이 정도의 공간 오버헤드는 현재 보통의 컴퓨터 시스템들에서 지원하고 있는 메모리 사양으로 볼 때 그리 크지 않다고 할 것이다.

5. 결 론

본 논문에서는 DSM 시스템에서 동적으로 할당된 공유 메모리를 통해 통신하는 병렬 응용들을 대상으로 거짓 공유를 줄이는 효과적인 동적 공유 메모리 할당 기법을 제시하였다. 공유 메모리 할당자의 사용자 인터페이스를 수정하지 않으면서도 기존 객체-크기별 할당 방식보다 더욱 효과적으로 거짓 공유를 줄이는 호출지 추적 기반 할당 기법을 제안하였는데, 이 기법에서는 객체-크기 대신 병렬 응용프로그램 내에서 객체를 요청한 위치를 추적하여 이를 기반으로 호출 경로가 다른 객체들이 같은 공유 페이지에 섞이지 않도록 하였다. 이는 프로그램 내에서 개발자의 의도를 유추하기 위한 힌트로 호출지를 사용한 것이라고 할 수 있으며, 실험 결과 객체-크기보다 훨씬 더 정확하게 객체에 가해질 미래의 참조 패턴을 예측할 수 있음을 확인하였다. 아울러 제시된 기법으로 인해 추가적으로 소요되는 메모리 양을 분석한 결과 프로그램 내에 존재하는 호출지 ID 개수만큼의 페이지 밖에는 더 소요되지 않음을 증명하였다. 결국 기존 객체-크기별 할당 방식과 공간 오버헤드는 비슷하면서도 거짓 공유는 더 많이 줄일 수 있었다. 본 논문에서 제시한 기법을 통해 거짓공유가 감소하면 DSM 시스템에서 불필요한 메모리 일관성 유지비용이 줄어들 것으로 예상된다.

향후에는 객체-크기별 할당 방식과 본 논문에서 제시한 CSTallocator, 그리고 그 외의 기타 여러 할당 방식들을 조합할 경우 어느 정도의 거짓 공유 감소 효과가 있는지 실험할 계획이며, 실험 환경을 시뮬레이션이 아닌 실제 DSM 시스템으로 개선함으로써 거짓 공유 폴트의 감소가 병렬 응용들의 응답 시간 감소에 어느 정도 기여하는지를 측정할 예정이다.

참 고 문 헌

[1] Andrew S. Tanenbaum. *Distributed Operating Systems*, chapter 6, pages 333-345. PRENTICE HALL, 1995.

[2] 이 종우, 조 유근. NUMA 다중 처리기에서 조정 가능한 지연 카운터를 이용한 페이지 복사 기법. 전자공학회논문지, 33(6):23-33, June 1996.

[3] Josep Torrellas, Monica S. Lam, and John L. Hennessy. Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II(Software),

pages 266-270, August 1990.

[4] Susan J. Eggers and Tor E. Jeremiassen. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I(Architecture), pages 377-381, August 1991.

[5] 이 종우, 조 유근. NUMA 다중 처리기에서 거짓 공유를 줄이는 공유 메모리 할당 기법. 정보과학회논문지, 23(5):487-497, May 1996.

[6] JongWoo Lee and Yookun Cho. An Effective Shared Memory Allocator for Reducing False Sharing in NUMA Multiprocessors. In *Proceedings of 1996 IEEE 2nd International Conference on Algorithms & Architectures for Parallel Processing(ICA^{3}PP '96)*, pages 373-382, June 1996.

[7] Roger L. Adema and Carla Schlatter Ellis. Memory Allocation Constructs to Complement NUMA Memory Management. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, December 1991.

[8] 이종우, 김문희, 한장희, 지대구, 윤종완, 김장선. 분산 공유 메모리 시스템에서 동적 공유 메모리 할당 기법이 거짓 공유에 미치는 영향. 정보과학회논문지, 24(12): 1257-1269, December 1997.

[9] 한부형, 조성제, 조유근. 분산 공유 메모리 시스템에서 거짓 공유 제거 및 통신량 감소 기법. 정보과학회논문지, 25(10):1100-1108, October 1998.

[10] 조성제. 분산 공유 메모리 시스템에서 메모리 참조 패턴에 근거한 거짓 공유 감소 기법. 정보처리논문지, 7(4):1082-1091, April 2000.

[11] 한부형, 조성제. 다중처리기 시스템에서 거짓 공유 완화를 위한 메모리 할당 기법. 정보과학회논문지, 27(4): 383-393, April 2000.

[12] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117-128, November 2000.

[13] Emery D. Berger. *Memory Management for High-Performance Applications*. PhD thesis, University of Texas at Austin, August 2002.

[14] Maged M. Michael. Scalable Lock-Free Dynamic Memory Allocation. In Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation(PLDI'04), June 2004.

[15] J. E. Veenstra. MINT Tutorial and User Manual. Technical Report TR452, Computer Science Department, University of Rochester, July 1993.

[16] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems(MASCOTS '94)*, pages 201-207, January-February 1994.

[17] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5-44, March 1992.

[18] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24-36, June 1995.

이 종 우

1990년 서울대학교 컴퓨터공학과 졸업(학사). 1992년 서울대학교 컴퓨터공학과 대학원 석사과정 졸업(석사). 1996년 서울대학교 컴퓨터공학과 대학원 박사과정 졸업(박사). 1996년~1998년 현대전자㈜ 정보시스템사업본부 과장. 1998년~1999년 현대정보기술㈜ 책임연구원. 1999년~2002년 한림대학교 정보통신공학부 조교수. 2002년~2003년 광운대학교 컴퓨터공학부 조교수. 2003년~2004년 아이닉스소프트㈜ 개발이사. 2004년~현재 숙명여자대학교 정보과학부 멀티미디어과학전공 조교수. 관심분야는 storage systems, computational finance, cluster computing, parallel and distributed operating systems, and embedded system software