다중처리기 시스템에서 거짓 공유 완화를 위한 메모리 할당 기법

(Memory Allocation Scheme for Reducing False Sharing on Multiprocessor Systems)

한 부 형 † 조 성 제 ††
(Boohyung Han) (Seongje Cho)

요 약 공유 메모리 다중처리기 시스템에서 거짓 공유는 서로 다른 처리기에 의해 참조되는 데이타 객체들이 동일한 일관성 유지 블록에 공존하기 때문에 발생하는 현상으로 메모리 일관성 유지비용을 증가시키는 주요 원인이다. 본 논문에서는 주 처리기가 공유 데이타 객체를 총괄하여 할당하는 병렬 응용들을 대상으로 거짓 공유를 감소시켜 주는 새로운 메모리 할당 기법을 제시한다. 제시한 기법에서는 일단 공유 객체를 임시 주소공간에 할당한 다음, 나중에 각 객체를 처음으로 참조한 처리기의 주소공간으로 정식 배치한다. 이렇게 함으로써 각 객체를 요청한 처리기별로 별도의 페이지에 각 객체가 할당되며, 서로 다른 처리기에서 요구한 데이타 객체들이 동일 공유 페이지에 섞이지 않게 된다. 본 기법의 효용성을 검증하기 위해 실제 병렬 응용을 사용하여 실행-기반 시뮬레이션을 수행하였다. 실험 결과 제시한 기법은 적은 오버헤드로 기존의 기법들에 비해 거짓 공유 현상을 적게 유발한다는 것을 확인하였다.

Abstract In shared memory multiprocessor systems, false sharing occurs when several independent data objects, not shared but accessed by different processors, are allocated to the same coherency unit of memory. False sharing is one of the major factors that may degrade the performance of memory coherency protocols. This paper presents a new shared memory allocation scheme to reduce false sharing of parallel applications where master processor controls allocation of all the shared objects. Our scheme allocates the objects to temporary address space for the moment, and actually places each object in the address space of processor that first accesses the object later. Its goal is to allocate independent objects that may have different access patterns to different pages. We use execution-driven simulation of real parallel applications to evaluate the effectiveness of our scheme. Experimental results show that by using our scheme a considerable amount of false sharing faults can be reduced with low overhead.

1. 서 론

대규모 다중처리기 시스템에서는 공유 메모리 개념(paradigm)을 효과적으로 지원하기 위해 일반적으로 데이타를 캐싱한다. 각 처리기의 지역 메모리로 참조한 데이타를 캐싱함으로써 메모리 참조 지연이 감소되며 병렬성도 향상된다. 일반적으로 공유 데이타를 캐싱할 때 문제점 중의 하나는 여러 복사본을 일관성있게 유지하는데 드는 비용으로, 서로 다른 처리기에 의해 참조되는 워드들이 동일 일관성 유지 블록 내에 유지될 때 이 비용은 더욱 더 커진다. 즉, 다중 처리기 시스템에서 불필요한 일관성(coherency) 오버헤드의 주요 원인은 거짓 공유(false sharing)라고 알려져 있으며, 거짓 공유는 실제 응용에서 일관성 유지비용의 아주 큰 부분을 차지한다[1, 2, 3]. 특히, 메모리 일관성 유지가 하드웨어적인 캐쉬 라인이 아닌 소프트웨어적인 페이지 단위로 이루어지는 경우에는 거짓 공유의 정도가 더 심해진다. 페이

† 학생회원 : 서울대학교 컴퓨터공학과
bhhan@ssrnet.snu.ac.kr
†† 정 회 원 : 단국대학교 자연과학부 교수
sjcho@dankook.ac.kr
논문접수 : 1999년 6월 14일
심사완료 : 2000년 1월 21일

지처럼 큰 블록은 지역성(locality)을 높여주지만 일관성 유지를 위해 다른 처리기로 페이지를 전송하는 경우에는 네트워크 통신량을 과도하게 증가시키기 때문이다.

Khera 등은 실험을 통해 무효화 기반 일관성 프로토콜뿐만 아니라 갱신 기반 일관성 프로토콜에서도 거짓 공유가 시스템 성능에 큰 영향을 미치는 것을 보였다[1]. 그들은 또 거짓 공유를 체계적으로 정의하는 것의 어렵다는 점과 나름대로의 거짓 공유 측정 방법을 제시하였다. 거짓 공유는 워드(word) 단위로 측정될 수 있으며, 서로 다른 처리기에 의해 참조되는 워드들이 동일 일관성 유지 블록에 존재할 경우 거짓 공유의 정도가 더 커진다[1, 4]. 거짓 공유는 병렬 시스템의 성능에 매우 큰 오버헤드로 작용하기 때문에 여러 연구들에서 거짓 공유 완화 방안을 제시하였다[5, 6, 7]. 이들 연구들은 서로 다른 타입의 데이타 객체들이 동일 페이지에 섞이지 않도록 하는 방법, 한 레코드의 크기를 캐쉬 라인 크기에 맞게 패딩(padding)하는 방법, 데이타를 재배치하는 방법, 데이타를 간접적으로 참조하는 기법 등을 적용하여 거짓 공유를 완화시켰다. 이러한 연구들에서는 주로 정적 데이타를 대상으로 하였으며 하드웨어 캐쉬들에 대해서 수행되었다. 따라서 공유 메모리 할당자(Shared Memory Allocator)에 의해 동적으로 데이타 객체나 페이지 단위로 가상 메모리를 관리하는 많은 병렬 응용들을 고려하지 않았다. 그리고 거짓 공유 완화에 드는 노력의 대부분을 응용 프로그래머에게 부담시켰다.

본 논문에서는 NUMA(Non-Uniform Memory Access) 시스템이나 분산 공유 메모리(Distributed Shared Memory: DSM) 시스템에서 발생되는 거짓 공유를 줄이기 위한 방법으로, 동적으로 공유 메모리를 할당하는 병렬 응용들을 대상으로 처리기 요구를 반영한 공유 데이타 객체 할당 기법을 제시한다. 페이지 단위로 메모리 일관성을 유지하는 페이지 기반 다중 처리기 시스템에서는 요청된 데이타를 어느 공유 페이지에 배치시키느냐가 매우 중요하다. 서로 관련이 없거나 처리기들에 의한 참조 패턴이 다른 데이타가 같은 페이지에 할당되는 경우에 일관성 유지를 위한 무효 폴트나 불필요한 갱신이 많이 발생한다. 이를 해결하기 위하여 주처리기가 총괄하여 공유 객체를 할당하는 병렬 응용을 대상으로, 각 처리기 요청에 기반한 처리기별 공유 객체 할당 기법을 도입한다. 본 기법은 처음 할당되는 객체들을 일단 임시 영역에 유지하다가 각 객체를 처음으로 참조한 처리기의 페이지로 실제 배치시킨다. 이렇게 하면 참조 패턴이 다른 객체들이 동일 페이지에 섞이는 것을 방지할 수 있으며 거짓 공유도 줄어들 수 있다. 제안한 기법을 라이브러리 수준에서 구현하여 프로그래머에게 어떠한 부담도 지우지 않는다.

본 논문의 구성은 다음과 같다. 2장에서는 거짓 공유에 대한 모델을 기술하고 폴트(fault)에 기반한 처리기별 할당 방식을 제안한다. 3장에서는 공유 데이타 객체들의 할당 패턴 및 참조 패턴을 분석하고, 4장에서는 제안한 메모리 기법의 특징을 설명한다. 5장에서는 거짓 공유를 감소시키기 위한 기존의 방식들과 제안한 방식들의 성능을 비교하고, 6장에서 결론을 맺는다.

2. 거짓 공유 완화 기법

거짓 공유 완화 기법을 모델링하기 위해, 기존 연구에서와 같이 거짓 공유를 메모리 참조의 기본 단위인 워드(w) 단위로 정의한다[1, 4]. 여기서 워드는 실제 크기에 상관없이 모든 원자적(atomic) 메모리 참조를 나타내는 용어로 사용된다. 참조되지 않는 워드는 마치 존재하지 않는 것처럼 취급하여 메모리 일관성 비용에 아무 영향을 미치지 않게 한다. 한 처리기 입장에서 **거짓 공유 폴트**란 거짓 공유를 유발하는 메모리 참조를 말한다. 페이지 기반 다중 처리기 시스템에서 메모리 일관성 유지 단위를 워드들로 구성된 페이지(p)라 할 때 다음과 같이 표현될 수 있다.

$$p_j = \{\ w_i \mid \text{word } i \text{ is part of page } j\ \}$$

$$p_j \cap p_k = \varnothing,\quad j \neq k$$

p_j에 w_i와 w_{i+1}이 공존할 때 w_i를 접근한 적이 있는 처리기 T_a 입장에서, "다른 처리기 T_b가 w_{i+1}을 수정하여 p_j가 무효화된 이후, T_a가 w_i을 참조할 때 발생하는 폴트"는 **거짓 공유 폴트**이며, "처리기 T_b가 w_{i+1}을 수정하여 p_j가 무효화된 이후, T_a가 w_{i+1}을 참조할 때 발생하는 폴트"는 **참 공유 폴트**이다. 좀 더 자세한 정의는 [4, 8]에 나타나 있다.

한 워드에 대한 처리기 집합 W_i는 특정 시간동안 그 워드를 참조한 처리기들의 집합이라고 정의하고, 한 페이지에 대한 처리기 집합 P_j는 그 페이지 내의 워드들의 처리기 집합들을 합한 것이라고 정의한다. 이러한 정의를 사용하여 p_j내의 특정한 w_i에 대한 거짓 공유의 정도 $F(i, j)$를 다음 식으로 유도할 수 있다.

$$F(i, j) = 1 - \frac{|W_i|}{|P_j|} \tag{1}$$

$$W_i = \{\ \text{processors referencing } w_i\ \}$$

$$P_j = \cup W_i = \{\ \text{processors referencing } p_j\ \}$$

(식 1)로부터 특정 워드의 처리기 집합 크기와 페이지의 처리기 집합 크기의 차가 클수록, 그 워드에 대한 거짓 공유의 정도가 커진다는 것을 알 수 있다. 어떤 응용의 p_i에 관련없는 여러 워드가 공존할 경우에 p_i를 참조하는 처리기들의 수는 많아질 것이지만 p_j페이지 내의 한 워드 w_i를 참조하는 처리기 수는 변화가 없을 것이므로 $F(i, j)$는 커진다. 페이지 크기가 커질수록 $|P_j|$는 커질 것이므로 $F(i, j)$ 역시 커지게 된다. 페이지 내에서 참조되지 워드의 경우에는 위 식을 더 이상 적용하지 않는다. 위 식에서 알 수 있듯이 거짓 공유를 감소시키기 위해서는 서로 관련없는 워드들을 서로 다른 페이지에 배치시키는 것이 중요하다.

본 논문에서는 프로그래머에게 투명하게 거짓 공유를 감소시키기 위해 다음의 설계 목표를 설정하였다. 첫째, 객체 공존성을 최소화(minimal co-location)하기 위해 가능한 한 서로 관련없는 또는 참조 패턴이 다른 공유 데이타가 동일 페이지에 섞이지 않도록 하였다. 둘째, 기존 공유 메모리 동적 할당자의 사용자 인터페이스를 수정없이 사용할 수 있게 함으로써, 거짓 공유 완화를 위한 어떠한 부담도 프로그래머에게 부과하지 않는다. 첫 번째 목표인 비공존성을 지원하기 위해서는 먼저, 병렬 응용들의 메모리 할당 방식을 살펴볼 필요가 있다.

많은 병렬 응용들에서는 데이타 객체 단위로 여러 처리기들이 공유하는 메모리를 할당하기 때문에[9, 10], 워드 단위로 공유 메모리를 배치하는 것은 오버헤드가 너무 크다. 적은 오버헤드로 거짓 공유를 감소시키기 위해 본 논문에서는 병렬 응용에서 객체 단위로 공유 메모리를 동적으로 할당하는 문제로 한정시킨다. 일반적으로 동적 공유 메모리 할당 루틴은 기본적으로 순차적 할당(sequential allocation) 방식에 의해 공유 객체를 할당한다[11, 12]. 따라서 각기 다른 처리기에 의해 요청된 데이타 객체라 하더라도 요청 순서대로 할당되기 때문에, 한 페이지에 여러 데이타 객체가 섞여서 할당될 가능성이 많다. 이러한 할당 방식은 메모리 공간을 낭비없이 사용할 수 있고 구현이 간단하지만, 거짓 공유로 인한 폴트가 많이 발생할 수 있다. 이에 대한 해결책으로 Torrellas 등은 다중처리기 캐쉬 연구에서 프로그래머가 직접 공유 메모리 할당을 요청한 프로세스 별로 별도의 공유 페이지를 사용하는 방법을 제시하였다[5]. 이 '프로세스별 할당 기법(또는 처리기별 할당 기법)'은 동일 공유 페이지에 서로 다른 프로세스가 요구한 데이타 객체들이 섞이지 않도록 하여 거짓 공유를 감소시켜 준다. 그러나 SPLASH-2와 같은 병렬 응용에서 공유 메모리를 하나의 주 처리기에서 총괄하여 할당하는 경우에는 효과가 없다. 실제로 많은 병렬 벤치마크 응용들의 대부분이 공유 메모리의 효율적 관리를 위해 주 처리기가 공유 메모리 할당과 반환을 담당한다[9, 10].

본 논문은, 주 처리기가 총괄하여 공유 메모리 할당과 반환을 담당하는 병렬 응용에서 '처리기별 객체 할당 기법'의 실현에 대한 것이다. 이를 위해 각 공유 객체에 대한 첫 번째 폴트(fault)를 힌트(hint)로 사용하여 관련없는 객체가 동일 페이지에 섞이지 않게 한다. 주 처리기가 공유 메모리를 순차적으로 할당하는 응용의 경우, 주 처리기가 아닌 다른 처리기에 의해서 주로 참조되는 경우에도 주 처리기의 메모리 공간에 할당되어 결과적으로 거짓 공유를 많이 유발시키게 된다. 이에 대한 해결책으로, 본 논문에서는 주 처리기에 의해 할당되는 객체들을 임시 공간에 유지하면서 각 객체가 처음 참조되면서 폴트가 발생될 때 폴트를 처음 유발시킨 처리기의 페이지로 해당 객체를 정식 배치시킨다. 이는 해당 객체를 요청한 처리기가 그 객체를 처음으로 접근한다는 '처리기별 할당 기법'과 객체에 대한 처리기 친화성(affinity)을 그대로 반영하는 것이다. 이렇게 하면, 기존의 순차적인 메모리 할당 방식보다 $|P_j|$가 작아지므로 $FS(i, j)$도 감소된다. 본 논문에서는 이를 **FFalloc** (First Fault-based Memory Allocation) 방식이라 칭한다.

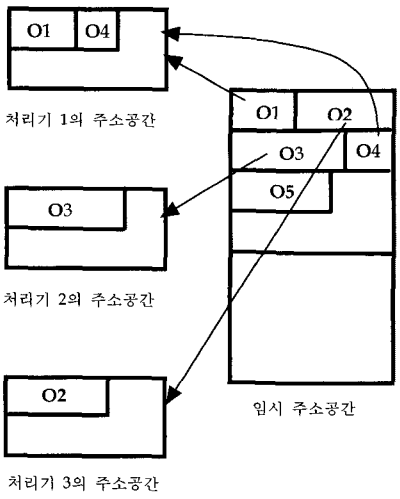

그림 1 처리기별 할당 방식

제안한 기법이 그림 1에 나타나 있다. 객체들은 일단 임시 주소공간에 할당되었다가 첫 번째 폴트를 유발시

킨 처리기의 주소공간으로 정식 배치된다. 그림에서 객체 O1과 O4는 처리기 1에 의해 처음 접근되었으며, 객체 O2는 처리기 3에 의해, 객체 O3는 처리기 2에 의해 처음 접근되었다. 이 기법에서 문제가 되는 것은 소요되는 메모리 공간이다. 당연히 순차적 할당 방식보다 처리기 수에 비례하여 더 많은 공간이 요구된다. 공유 메모리가 할당 요청되고 난 후, 각 객체에 대해 폴트가 발생하기 전까지는 임시 주소공간을 위해 많은 메모리가 사용된다. 그러나 폴트 발생 후 정식 배치가 되면 각 처리기 공간의 마지막 페이지에서 단편화(fragmentation)가 발생할 수 있으므로, 처리기 수가 N일 경우 순차적 방식보다 평균 (N-1)/2 개의 페이지가 더 사용된다. 실제 예로 처리기 수가 32개이고 4KB 페이지라면 평균 31/2 ×4 KB 공간이 추가로 요구된다는 것이다. 이는 보통 다중처리기 시스템의 가상 주소 공간이 충분하다는 가정에 비추어 큰 문제는 아니다.

3. 공유 데이타 할당 및 참조 패턴

FFalloc 기법을 구현하기 전에 실제 병렬 응용들의 공유 메모리 할당 및 참조 패턴을 분석하여 FFalloc 기법의 타당성을 조사한다. 먼저, 실험 환경을 기술하고 SPLASH-2 응용들의 공유 데이타 객체의 할당 패턴 및 참조 패턴에 대해 언급한다.

3.1 실험 환경

본 논문에서는 프로그램 구동형 시뮬레이터(program-driven simulator)인 Mint[12, 13]를 사용하여 다중 처리기 시스템을 실험하였다. Mint는 메모리 참조 발생기(memory reference generator)와 목표 시스템 시뮬레이터(target system simulator)라는 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[13] 실제 시스템에서 실험하는 것과 동일한 효과를 얻을 수 있다. 본 논문에서 제시된 기법을 적용하기 위하여 메모리 참조 발생기에 포함된 공유 메모리 할당자를 수정하였고, 페이지 기반 다중 처리기 시스템을 모델링하기 위하여 목표 시스템 시뮬레이터를 새로 구현하였다. 실험에서 구현한 시스템은 메모리 일관성 유지 프로토콜로서 무효화 기반 프로토콜을 사용하고, 페이지 소유자를 찾기 위하여 Li 등이 고안한 '동적 분산 관리자 알고리즘'[14]을 사용한다.

표 1 병렬 응용들의 특성

병렬 응용	문제 크기	메모리 참조 회수 ($\times 10^6$)	공유 데이타 크기 ($\times 10^6$ byte)
cholesky	tk15.O	399.0	21.8
fft	1048576 points	392.6	50.8
lu	512×512 matrix	302.5	2.2
ocean	258×258 grid	233.6	24.1
radix	256K keys, 1K radix	30.1	4.9
water-spatial	512 molecules	334.0	0.4

표 1은 처리기 수가 32개일 때 여러 병렬 응용들의 특성들을 보여준다. 표 1에 나타난 병렬 응용들은 모두 Stanford 대학의 SPLASH-2[10] 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.

3.2 공유 데이타 할당 및 참조 패턴

처리기 수가 32개이고 페이지 크기가 4KB일 때 병렬 응용들에 대해 공유 객체와 동기화 객체의 할당 요청 패턴을 분석한 결과를 표 2와 표 3에 제시하였다. 표에서 '요청 크기'는 병렬 응용이 수행될 때 동적으로 요청되어 할당된 데이타 객체의 크기로, 태그(tag) 정보를 포함한 크기를 나타낸다. 태그는 각 객체들이 반환될 때 사용되는 메모리 제어 블록으로 해당 데이타 객체와 연속된 위치에 놓이는 것이 일반적이다. 표 2와 3의 결과에서 '요청 크기'는 기본 객체의 크기에다 태그 8바이트를 합하여 표현하였다. '연속 요청 회수'는 동일한 크기의 데이타 객체가 연속적으로 몇 번 요청되었는가를 나타낸다. 예를 들면, cholesky에서는 262,160 바이트 크기의 데이타 객체가 총 76번 요청되는데, 그 요청 중 2번, 50번, 24번이 각각 연속적으로 이루어진다는 것을

의미한다. 공유 메모리 할당 패턴을 보면 동일한 크기의 객체들은 대부분 할당 요청이 연속적으로 이루어진다. MINT에서 공유 메모리 영역은 0x40,000,000 주소에서 시작하는데, 첫 페이지에는 별도로 동기화 객체들(spin lock, barrier events, semaphore 등)이 배치되며 일반 공유 객체는 0x40,001,000 주소에서부터 순차적으로 배치된다. FFalloc 기법에서는 동기화 객체들에 대한 할당은 고려하지 않으며 일반 공유 객체에 대한 할당만을 다룬다.

다음으로 공유 객체의 참조 패턴을 분석하였다. 공유 메모리 참조 특성은 다음 세 가지로 요약할 수 있다. **첫째**, 순차적으로 할당되는 공유 객체들은 대부분 서로 다른 참조 패턴을 보인다. 특히, 동일 크기의 객체들이 처

표 2 할당 요청 크기와 요청 횟수

요청 크기 (바이트)	요청 회수	연속 요청 회수	요청 크기 (바이트)	요청 회수	연속 요청 회수
cholesky			ocean		
24	1		32	1	
80	33	33	40	1	
160	1		248	1	
672	1		164,216	1	
8,272	32	32	532,520	2	
262,160	76	2, 50, 24	534,584	1	
357,640	1		1,065,032	2	
487,072	1		1,065,040	1	
715,272	1		1,597,544	2	2
			2,130,056	3	
			9,585,368	1	
fft			radix		
48	1		144	2	2
136	2	2	264	3	3
16,392	1		8,200	32	32
16,912,392	3	2	131,208	1	
			1,048,584	2	2
			2,361,032	1	
lu			water-spatial		
64	1		24	69	2, 64
136	1		32	32	32
264	5	5	40	16	4×4
4,104	2	2	136	2	
69,640	32	32	184	1	
			688	512	512

표 3 동기화 객체 할당 패턴

요청 크기 (바이트)	요청 회수	연속 요청 회수	요청 크기 (바이트)	요청 회수	연속 요청 회수
cholesky			ocean		
32	1		32	20	
36	68	3, 32	36	26	6
fft			radix		
32	1		32	130	33×2
36	2	2	36	100	34
lu			water-spatial		
32	1		32	3	
36	2	2	36	73	16×4, 7

리기 수만큼 연속 할당되는 경우에는 특정 객체가 특정 처리기(들)에 의해서만 참조된다 (<참조 패턴 1>). Cholesky에서 8,272 바이트 크기의 객체와 radix의 8,200 바이트 크기의 객체들은 각각 한 처리기에 의해서만 참조된다. Radix의 8,200 바이트 크기의 객체는 10번[1]째부터 41번째까지 연속적으로 할당되는데, 10번 객체는 1번 처리기에 의해서만 참조되며, 11번 객체는 2번 처리기에 의해서만 참조되며, 12번 객체는 3번 처리기에 의해서만 참조된다. Cholesky의 80 바이트 크기의 객체들(4번부터 36번까지 연속 33번 할당됨)과 water-spatial의 32 바이트 크기의 객체들(2번부터 33번까지 연속 32번 할당됨)은 각각이 한 개, 또는 두 개의 처리기에 의해서만 참조되며, 연속된 객체들의 참조 패턴이 다르다. Cholesky의 4번 객체는 0번에 의해서만 참조되며, 5번 객체는 0번과 1번 처리기에 의해서만 참조되고, 6번 객체는 0번과 2번 처리기에 의해서만 참조된다.

둘째, 동일 크기의 객체들이 처리기 수보다 훨씬 많이 연속 할당되는 경우에는 연속된 2개 이상의 객체가 한 그룹이 되어 동일한 참조 패턴을 보인다(<참조 패턴 2>). Water-spatial의 24 바이트 크기의 객체들과 688 바이트 크기의 객체들이 대표적인 예이다. 688바이트 객체의 경우 연속한 두 객체가 그룹이 되어 하나의 처리기에 의해서만 85%이상 집중적으로 참조된다. 즉, 57-58번은 처리기 0에 의해서, 59-60번은 처리기 1에 의해서 집중적으로 참조된다. 그리고, water-spatial의

1) 처음으로 할당되는 객체는 1번, 두 번째로 할당되는 객체는 2번 등과 같이 번호를 매겼다.

24 바이트 크기의 객체 569—570번은 처리기 1에 의해서만 참조되며, 571—572번은 처리기 2에 의해서만 참조되며, 573-574번은 처리기 3에 의해서만 참조된다. <참조 패턴 1, 2>의 경우 순차적 할당 기법에서는 참조 패턴이 다른 객체들이 같은 페이지에 공존하게 되어 거짓 공유가 많이 발생할 수 있다.

셋째, 페이지 크기보다 매우 큰 객체의 경우, 객체 전체로 보면 여러 처리기들에 의해 참조되나 객체 내의 특정 페이지는 특정 처리기들에 의해서만 참조되는 페이지별 참조 패턴을 보인다(<참조 패턴 3>). Cholesky의 262,160 바이트 크기의 객체와 lu의 69,640 바이트 크기의 객체, *ocean*의 1,597,544 바이트 객체들이 대표적인 예이다. Cholesky의 262,160 바이트 객체는 37-38번, 42-91번, 124-147번까지 총 76번 할당된다. 이중 37번 객체는 전체적으로 0-64번 공유 페이지들에 걸쳐 배치되어 모든 처리기에 의해 참조되나 1-20번 페이지는 처리기 0에 의해서만 참조된다. 그림 2는 ocean의 1,597,544 바이트 객체에 대한 참조 패턴을 가상 시간(해당 객체에 대한 첫 번째 참조를 가상 시간 1로 보며, 두 번째 참조를 가상시간 2로 봄)에 따라 보여준다. 이중 5번 객체는 820-1210번 공유 페이지들에 걸쳐 배치되어 모든 처리기에 의해 참조되나, 821-835번 페이지는 처리기 1-4번에 의해서만 참조되며 838-851번 페이지는 처리기 5-8번에 의해서만 참조된다. 그림 2에서는 두 객체에 대한 처음 16,000번째까지의 메모리 참조 패턴만 나타내었다.

4. FFalloc 기법과 SSalloc 기법

이상에서 보면 처리기 0이 총괄적으로 공유 객체를 할당하지만, 처리기 0에 의해서는 참조되지 않는 객체가 많이 있다. FFalloc 기법에서는 이러한 객체들을 대상으로 그 객체를 실제 참조하는 처리기의 페이지로 분류하여 거짓 공유를 줄이고자 하는 것이다. 거짓 공유를 줄이고자 하는 다른 연구로는, 요청되는 객체의 크기별로 공유 객체들을 별도의 페이지에 할당함으로써 크기가 서로 다른 객체가 동일 페이지에 섞이지 않도록 하는 '크기별 할당 방식'[4, 8]이 있다. 그리고 연속적으로 할당되는 객체들이 서로 다른 참조패턴을 가질 수 있다는 특성을 이용하여 순서적으로 번갈아 가면서 객체를 다른 페이지에 배치하는 '페이지 풀 기반 교대 할당 방식(Page Pool based Alternate Memory Allocation: 이하 PPalloc이라 칭함)'[15]도 있다. 3.2절의 <참조 패턴 1>과 <참조 패턴 2>의 특성을 갖는 객체의 경우, 동일 크기의 객체들이 서로 다른 참조 패턴을 보이므로 '크기별 할당 기법'은 거짓 공유 감소에 도움을 주지 않을 수 있다. PPalloc 기법도 <참조 패턴 2>의 특성을 보이는 같은 그룹의 객체들을 서로 다른 페이지에 배치시켜 거짓 공유 발생을 증가시킬 수 있다. 또, PPalloc에서 같은 처리기의 페이지에 할당될 수 있는 1번, 33번, 65번, 97번 객체가 있을 때 1번과 65번 객체는 처리기 1에 의해서만 참조되고, 33번과 97번 객체는 처리기 2에 의해서만 참조되는 경우에, PPalloc은 이를 구별하지 못한다. 이에 반해 FFalloc은 처리기별 참조 특성을 잘 반영하여 패턴이 다른 객체를 다른 페이지에 배치시켜 준다. <참조 패턴 3>의 특성을 갖는 객체들에 대해서는 '크기별 할당 방식', PPalloc, FFalloc 모두가 해결책이 되지 못하며, 다중 페이지 걸침 최소화 방식에 의해서 거짓 공유가 감소될 가능성이 있다. 그러나 다중 페이지 걸침 최소화 방식도, 페이지 내의 워드 단위로 거짓 공유를 측정할 경우 각 페이지에 배치되는 워드의 순서를 재배치시켜 오히려 거짓 공유가 증가시킬 수도 있다. 이에 대한 실험 결과가 5장에 기술되어 있다.

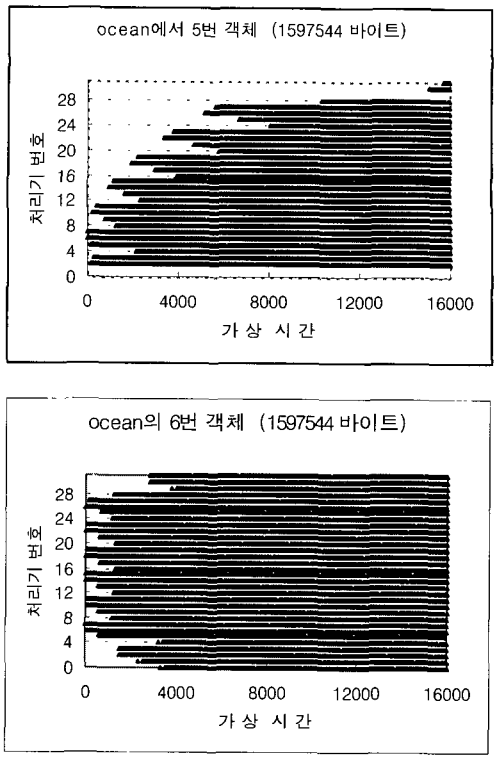

그림 2 ocean에서 5번과 6번 객체의 참조 패턴

표 2에서 '요청 회수'가 1인 객체들과 '요청 회수'가 2

이상이더라도 동일 크기가 연속적으로 할당되지 않는 객체는 특정 처리기에 의해서만 참조되는 것이 아니라 대개가 여러 처리기들에 의해 번갈아 가며 참조된다. 따라서, 본 논문에서는 동일 크기의 객체가 2번 이상 연속적으로 할당되는 경우에만 '처리기별 할당 방식'을 적용시켜 보았다. 이를 **SSalloc**(Same Size and First Fault based Memory Allocation)이라 칭한다. 이 방식은 공유 객체가 O_1, O_2, O_3, ... 순서로 할당 요청될 때, 각 객체가 할당될 때마다 그림 3의 알고리즘을 차례차례 수행한다. 그림에서 **일반 주소공간**이란 요청 회수가 1인 객체들과 연속적으로 할당되지 않는 '동일 크기 객체'들을 배치시키기 위한 공간을 말한다.

① 객체 O_i가 할당 요청되면, **임시 주소공간**에 일단 배치된다.
② O_i와 O_{i-1}의 크기가 다르면, 즉시 O_{i-1}을 **일반 주소공간**에 순차적으로 정식 배치한다.
③ O_i와 O_{i-1}의 크기가 같다면, O_{i-1}과 O_i는 처음 폴트를 유발시킨 **처리기 주소공간**으로 정식 배치된다.

그림 3 객체 O_i가 할당될 때 SSalloc 기법의 수행 순서

SSalloc은 동일 크기의 객체가 연속 할당되는 객체의 경우에, 특히 참조 패턴이 다르다는 분석 결과를 반영한 것이다. 그 외의 객체들은 순차적으로 할당된다. 크기별 할당 방식과 PPalloc에서는 다중 페이지 걸침(multiple-page spanning) 최소화 방식도 적용하여 하나의 객체가 두 개 이상의 페이지에 걸쳐서 할당되는 것을 최소화시켰지만, FFalloc과 SSalloc에서는 이 방식을 적용하지 않는다.

4.1 처리기 수와 페이지 크기에 따른 패턴

처리기 수가 2, 4, 8, 16개일 때도 객체들의 할당 및 참조 패턴을 조사해 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 공유 페이지 전체 할당 수는 증가하였으며(water-spatial만 87개로 동일), 객체들의 평균 할당 요청 크기는 감소하였다. 따라서 처리기 수가 많을수록 하나의 페이지에 공존하는 객체의 수가 증가하고, 같은 페이지를 참조하는 처리기 수가 많아져 거짓 공유 폴트의 수도 증가한다. 그리고, 페이지 크기가 클수록 여러 데이타 객체가 공존할 확률이 높아지기 때문에 거짓 공유 폴트의 빈도는 높아진다. 이는 2장에서 설명된 바와 같다. 페이지 크기가 작을 경우, 다량의 페이지 테이블이 요구되며 요구 페이징에서는 프로그램 수행 시 페이징 사상도 과도하게 발생한다. 또한 메모리와 디스크 사이의 데이타 전송을 효율적으로 하기 위해 디스크 블록의 크기와 같은 크기의 페이지를 사용한다. 이러한 이유로 현재 많은 시스템들은 4KB나 8KB 페이지를 채택하고 있으며 본 논문에서도 페이지 크기를 4KB를 선택하였다. 현재 고속 네트워크 기술의 발전으로 많은 컴퓨터가 연결되어 공동작업을 하는 경우는 일반화되었기 때문에, 본 논문에서는 처리기 수를 32개로 설정하였다.

5. 공유 메모리 할당 방식의 성능

프로그래머에게 투명성을 보장하기 위해 본 연구에서는 라이브러리 수준에 있는 동적 메모리 할당자를 수정하여 FFalloc와 SSalloc 기법을 구현하여 성능을 평가하였다. 이들 기법에서는 임시 페이지에 대한 페이지 폴트 시, 어떤 객체에 대한 폴트인지만 구별하여 해당 객체를 정식 배치하면 되므로 공유 메모리가 더 필요하며 다른 오버헤드는 아주 적다. 제시한 기법의 성능을 평가하기 위해 순차적 할당 기법뿐만 아니라 다른 연구진이 제시한 기법도 같이 실험하였다. 실험한 다른 기법으로는, '크기별 할당 방식'[4, 8]과 '페이지 풀 기반 교대 할당 방식(PPalloc)'[15] 등이 있다. 그리고 한부형 등은 [15]에서 거짓 공유의 개수를 객체 단위로 측정하였는데, 본 장에서는 워드 단위로 측정하였다.

본 논문에서는 다섯 가지 메모리 할당 방식들을 구현하여 성능을 서로 비교하였는데, 성능 비교가 그림 4에 나타나 있다. 그림에서 성능을 비교하기 위해 2장에서 정의한 거짓 공유 폴트와 참 공유 폴트의 수 외에 cold miss의 수도 측정하였다. cold miss란 어떤 워드에 대해 임의처리기에 의해 발생하는 첫 번째 폴트를 의미한다. Cholesky와 fft를 제외하고는 거짓 공유 폴트가 전체 폴트의 95%을 차지하기 때문에 그림에서는 cold miss와 참 공유 폴트의 수가 거의 나타나지 않는다. Cholesky의 경우 '크기별 할당 방식'을 제외하고는 순차적 방식에 비해 거짓 공유가 감소하였다. 크기별 할당에서는 서로 다른 참조 패턴을 보이는 동일 크기의 객체를 동일 페이지에 공존시키기 때문에 폴트가 7,869회(1.5%) 증가하였다. SSalloc에서는 폴트가 83,814회(16.3%) 감소하였는데, 이는 동일 크기의 객체가 연속적

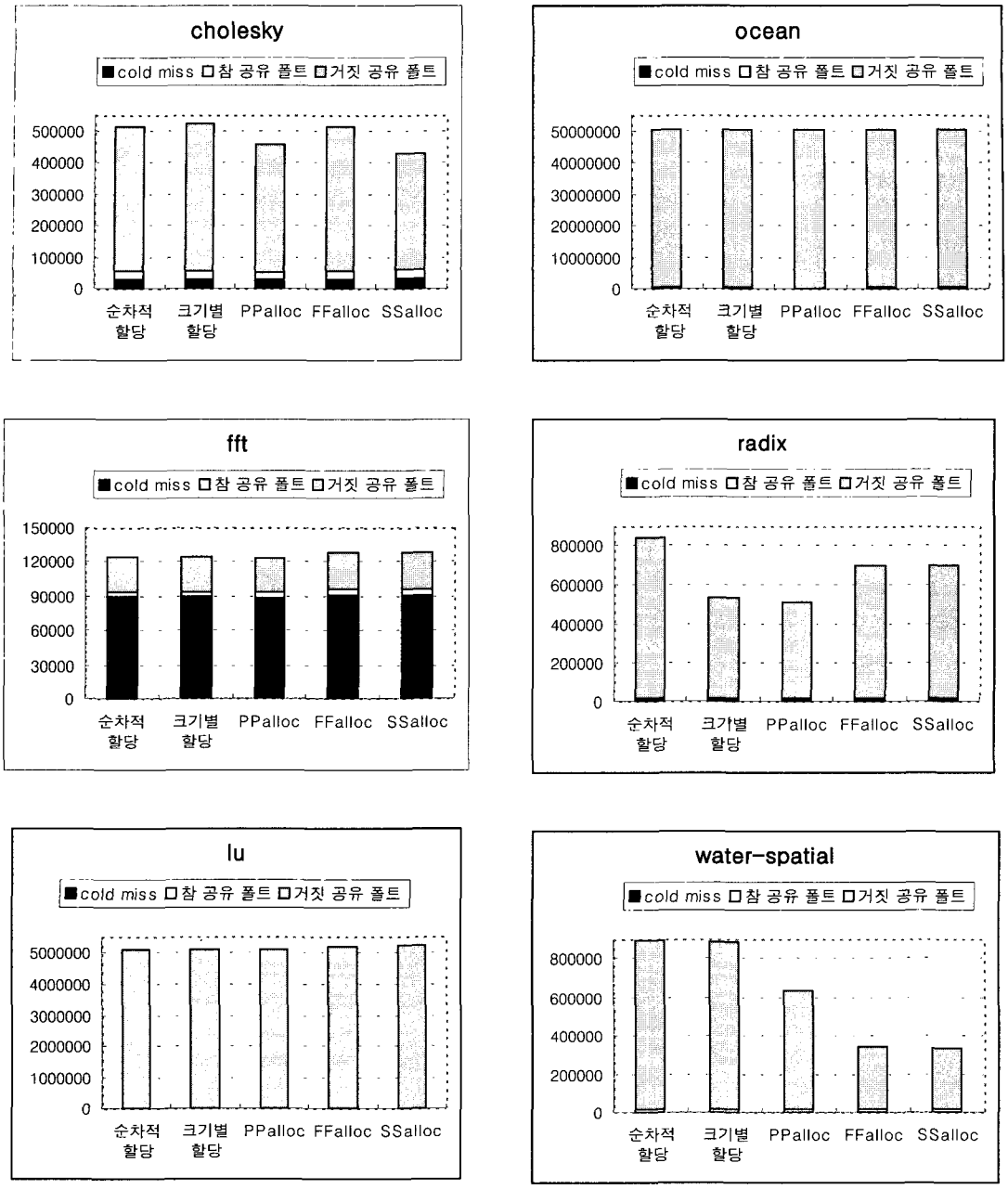

그림 4 동적 메모리 할당 방식들에서 발생하는 폴트 수

으로 할당되는 경우 서로 다른 참조 패턴을 보이기 때문에 이들을 서로 공존하지 않도록 배치한 결과이다.

Fft와 lu, ocean의 경우에는 각 방식마다 성능의 차이가 거의 없으며 순차적 방식보다 FFalloc와 SSalloc 방식에서 거짓 공유가 조금 증가하였다. 이는, 세 응용의 경우 페이지 크기보다 훨씬 큰 객체들이 주류를 이루며 이들 객체들이 여러 처리기들에 의해 골고루 참조되기 때문이다. 그리고 첫 번째 폴트가 해당 객체에 대한 메모리 참조 패턴이나 서로의 연관성을 정확히 나타내지 않기 때문이다. lu의 경우 69,640 바이트 객체가 32번 연속적으로 할당되는데(2번부터 33번까지 할당됨), 이들 객체는 처음 폴트를 유발한 처리기보다는 다른 처리기에 의해서 많이 참조된다. 예를 들어 lu의 2번, 3번, 4번 객체의 경우 모두 처리기 0에 의해서 첫 폴트가 유발되지만, 각각 처리기 1, 처리기 2, 처리기 3에 의해 주로 참조된다.(2번 객체의 경우 처리기 0에 의해서는 32,800번 참조되나 처리기 1에 의해서 4,750,000 정도 참조된다.) Fft와 ocean의 경우에는 참조 패턴이 다른 객체들이 아주 적기 때문에 제안된 기법이 성능을 향상시키지 못했다. Fft의 경우 cold miss의 비중이 매우 커 성능 향상의 여지가 별로 없다. 크기별 할당과 PPalloc에서는 다중 페이지 걸침 최소화 방식에 의해 fft에서는 거짓 공유가 다소 줄었다. Ocean의 경우에는 순차적 할당이 가장 좋은 성능을 보였다. FFalloc의 경우 폴트 수가 fft에서는 3,463회(2.8%), lu에서는 86,994회(1.7%), ocean에서는 114,495회(0.2%) 증가하였다.

Radix의 경우, 전체 폴트의 수가 FFalloc에서는 141,838회(16.9%), SSalloc에서는 141,415회(16.9%) 감소되었으며 크기별 할당과 PPalloc이 아주 좋은 성능을 보였다. 이는 페이지 크기보다 큰 객체들이 다른 객체들에 비해 아주 많은 거짓 공유를 유발시키며, 객체들이 할당되는 순서에 따라 서로 다른 참조 패턴을 보이기 때문이다. Radix의 144 바이트 객체와 264 바이트 객체들은 각각 100번 이하 참조되지만 131,208 바이트 객체는 1,246,000번 정도 참조된다. Water-spatial의 경우에도 모든 방식들에서 성능이 향상되었으며 FFalloc과 SSalloc에서는 전체 폴트의 수가 각각 550,261회(61.7%), 555,681회(62.3%) 감소되었다. 이는 모든 객체들이 페이지 크기보다 작고 또 동일 크기의 객체들이 연속적으로 할당되면서 서로 다른 참조 패턴을 보이기 때문이다. 즉, 어떤 객체는 하나의 처리기에 의해서만 참조되며, 어떤 객체는 처음 참조한 처리기에 의해 많이 참조된다.

또한, 다중 페이지 걸침 최소화 방식만을 적용하여 실험하여 보았는데 cholesky, fft, radix, water- spatial의 경우에는 거짓 공유를 각각 4,935회, 31회, 309,203회, 110,324회 감소시켰으나, lu와 ocean의 경우에는 오히려 거짓 공유를 각각 1회, 112,505회 증가시켰다. 따라서, radix에서 크기별 할당과 PPalloc이 좋은 성능을 보이는 것은 다중 페이지 걸침 최소화 방식을 적용하였기 때문이다. FFalloc과 SSalloc에서는 다중 페이지 걸침 최소화 방식을 적용하지 않았다. 실험 결과에서 FFalloc과 SSalloc이 거짓 공유를 감소시킬 수 있음을 알 수 있으며, 특히 페이지 크기보다 작은 동일 크기의 객체들이 연속적으로 할당될 때 효과가 뛰어났다. 그러나 할당 요청되는 객체의 개수가 적거나 페이지보다 큰 객체들이 주로 할당되는 경우에는 그 효과가 크지 않다. 이는 객체의 크기가 페이지 크기보다 작을 때는 객체 단위로, 클 때는 페이지 단위로 메모리 할당이나 메모리 일관성을 유지하는 것이 좋음을 의미한다. 새로운 할당 기법을 사용하였을 때 추가로 필요한 메모리 공간을 분석한 결과가 표 4에 나타나 있다. 표에서 제시된 공간은 동기화 변수를 제외한 공유 데이타 객체들이 차지하는 전체 메모리 공간을 페이지 수로 나타낸 것이다. '공유 메모리 크기'는 순차적 할당 기법에서 사용된 공유 메모리 공간을 나타낸다. 어떤 처리기는 자신에 의해 폴트가 처음으로 유발되는 객체가 전혀 없기 때문에, 자신의 주소공간을 가지지 않는 경우가 있다. 따라서 대부분 응용의 경우 처리기 개수보다 적은 수의 페이지가 추가로 든다. SSalloc에서는 FFalloc에 비해 일반 주소공간을 위해 하나의 페이지가 더 요구된다. 크기별 할당과 PPalloc에서는 다중 페이지 걸침을 최소화하기 위해서 추가의 메모리를 요구한다.

표 4 각 메모리 할당 방식에서 사용된 메모리 공간

(단위: 페이지 수)

병렬 응용	공유 메모리 크기	크기별 할당	PPalloc	FFalloc	SSalloc
cholesky	5,311	5,318	5,395	5,342	5,343
fft	12,392	12,396	12,425	12,413	12,414
lu	547	551	595	559	560
ocean	5,892	5,905	5,929	5,918	5,919
radix	1,185	1,191	1,235	1,216	1,217
water-spatial	87	91	128	118	119

5.1 구현시의 메모리 공간

Mach의 가상 메모리 구조[16]와 같은 가상 메모리

시스템에서 프로세스가 동적으로 메모리를 할당하기 위해 malloc/sbrk 호출을 수행할 경우에 커널은 주소 맵 엔트리(*vm_map_entry*)만을 할당하거나 주소 맵 엔트리가 지정할 수 있는 가상 주소의 범위만 확장하고 주기억장치의 자유 페이지 프레임(free page frame)은 할당하지 않는다. 나중에 해당 주소에서 실제로 페이지 폴트가 발생하면 자유 페이지 프레임을 할당하여 heap 영역의 페이지 테이블에 사상시킨다. 이는 요구 페이징 기법을 사용하는 대부분의 가상 메모리 시스템에서도 동일하다. 이와 같은 구조에서 FFalloc을 적용한다면 '임시 주소공간'을 위해서는 물리적인 메모리가 전혀 필요하지 않다. 즉, 새로운 객체 할당이 요청되는 경우에는 그 객체에 대한 임시 가상 주소공간만을 일단 확보한다. 후에, 해당 객체가 실제로 참조될 때 폴트가 발생할 것이며, 이때 폴트를 유발시킨 처리기의 물리적 메모리에 객체를 실제 배치하면 된다. 32비트 CPU의 경우 전체 가상 주소공간이 4GB이므로 50MB(표 1에서 공유 메모리 전체 크기는 fft가 50MB 정도로 가장 크다) 정도의 임시 가상주소 공간을 확보하는 것은 어렵지 않다.

6. 결 론

공유 메모리를 사용하는 대규모 다중처리기 시스템에서는 병렬성을 효과적으로 지원하기 위해 공유 데이타를 지역 메모리로 캐싱하여 참조한다. 데이타를 캐싱하면 데이타에 대한 접근 시간은 감소되지만, 여러 복사본들에 대해 메모리 일관성을 유지하는 비용이 증가하게 된다. 메모리 일관성 유지비용을 증가시키는 주요 원인은 거짓 공유이다. 거짓 공유는 서로 다른 처리기에 참조되는 워드들이 동일한 일관성 유지 블록에 공존하기 때문에 발생하는 현상으로 메모리 일관성 유지비용을 증가시키는 주요 원인이다. 본 논문에서는 주 처리기가 공유 데이타 객체를 총괄하여 할당하는 병렬 응용들을 대상으로 거짓 공유를 감소시켜 주는 새로운 메모리 할당 기법(FFalloc)을 제시하였다. 제시한 기법에서는 해당 객체를 실제 요청한 처리기가 그 객체에 대해 첫 번째 폴트를 유발시킨다는 것을 이용하여, 각 객체에 대해 첫 번째 폴트를 유발시킨 처리기 영역에 그 객체를 배치한다. 이렇게 함으로써 각 객체 할당을 요청한 처리기별로 별도의 페이지를 사용하게 되며, 서로 다른 처리기에서 요구한 데이타 객체들이 동일 공유 페이지에 섞이지 않게 된다.

또한, 동일 크기의 객체가 연속적으로 할당되는 경우에만 해당 객체를 첫 번째 폴트를 유발시킨 처리기의 페이지에 배치하는 기법(SSalloc)도 제시하였다. 본 기법의 효용성을 검증하기 위해 실제 병렬 응용에서 공유 데이타 객체들의 할당 및 참조 패턴을 분석하였으며, 병렬 응용을 사용하여 실행-기반 시뮬레이션을 수행하였다. 이를 통해 제시한 기법들이 적은 오버헤드로 기존의 기법들에 비해 거짓 공유 현상을 적게 유발한다는 것을 확인하였다. 메모리가 가지는 시간적인 지역성(temporal locality)에 비추어 볼 때 한 객체에 대한 특정 처리기의 폴트는 그 객체를 그 처리기가 당분간 계속 참조할 것임을 추정할 수 있다. 더욱 큰 성능 향상을 얻기 위해서, 향후에는 페이지(또는 데이타 객체)의 참조 패턴에 따라 동적으로 페이지(또는 객체)를 이동하거나 복사하는 분야를 연구할 계획이다.

참 고 문 헌

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

[2] E. P. Markatos and C. E. Chronaki, "Trace-Driven Simulation of Data-Alignment and other Factors affecting Update and Invalidation Based Coherent Memory," Proceedings of Int'l Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems(MASCOTS '94), pp. 44-52, January 1994.

[3] 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, pp. 57-71, 1993.

[4] 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.

[5] J. Torrellas, M. S. Lan, and J. L. Hennessy, "Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates," Proceedings of the 1990 International Conference on Parallel Processing, Vol. II(Software), pp. 266-270, August 1990.

[6] S. J. Eggers and T. E. Jeremiassen, "Eliminating False Sharing," Proceedings of the 1991 International Conference on Parallel Processing, Vol. I(Architecture), pp. 377-381, August 1991.

[7] 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, 1995.
[8] 이종우 등, "분산 공유 메모리 시스템에서 동적 공유 메모리 할당 기법이 거짓 공유에 미치는 영향", 정보과학회 논문지, 24(12):1257-1269, 1997년 12월.
[9] J. P. Singh, W. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared- Memory," ACM SICARCH Computer Architecture News, vol. 20, no. 1, pp. 5-44, March 1992.
[10] 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.
[11] J. E. Veenstra and R. J. Fowler, "MINT: A Front End for Efficient Simulation of Shared- Memory Multiprocessors," Proc. of the 2nd Int'l Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems(MASCOTS '94), pp. 201-207, Jan.-Feb. 1994.
[12] J. E. Veenstra and R. J. Fowler, "Source code of shared memory allocator in Mint," University of Rochester.
[13] J. E. Veenstra, "MINT Tutirial and User Manual," Technical report, TR452, Computer Science Department, Univ. of Rochester, July 1993.
[14] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," ACM Transactions on Computer Systems, vol. 7, pp. 321-359, Nov. 1989.
[15] 한부형 등, "분산 공유 메모리 시스템에서 거짓 공유 제거 및 통신량 감소 기법", 정보과학회 논문지, 25(10):1100-1108, 1998년 10월.
[16] Open Software Foundation, Design of the OSF/1 Operating System, Release 1.2., Prentice-Hall, 1993.

한 부 형
1991년서울대 컴퓨터공학과 학사. 1993년 서울대 컴퓨터공학과 석사. 1993년 ~ 현재 서울대 컴퓨터공학과 박사과정. 관심분야는 운영체제, 분산 시스템, 분산 공유 메모리 등임.

조 성 제
정보과학회논문지 : 시스템 및 이론
제 27 권 제 3 호 참조