링 구조 NUMA 시스템에서 적응형 다중 그레인 원격 캐쉬 설계

(Application Behavior-oriented Adaptive Remote Access Cache in Ring based NUMA System)

곽 종 욱 † 장 성 태 †† 전 주 식 †††
(Jong Wook Kwak) (Seong Tae Jhang) (Chu Shik Jhon)

요 약 메모리 병목현상의 완화와 구현상의 용이함으로 인해 NUMA 시스템이 지난 수년 동안 전형적인 다중 프로세서 시스템으로 자리를 잡아 왔다. 하지만 NUMA 시스템은 그 구조의 특성상 원격 메모리로의 접근 비율이 커질수록 응답 속도의 지연이 심화되므로, NUMA 시스템의 구현에 있어서 원격 캐쉬의 효율적인 설계를 요구한다.

본 논문에서는 보다 효율적인 원격 캐쉬의 설계를 목표로 하여, 원격 캐쉬 상에서 실제 응용 프로그램의 공유 단위(Granularity of Sharing)의 패턴을 분석하여 원격 캐쉬의 라인 사이즈를 실행 시간에 가상적으로 변화시킬 수 있는 "다중 그레인 원격 캐쉬" 방식을 제안한다. 그리고 이를 MINT를 통해 모델링한 후 시뮬레이션을 수행하고 그 결과를 분석한다. 시뮬레이션에서는 먼저 Profile-Based 방식을 이용하여 각 응용 프로그램별 최적의 원격 캐쉬 라인 사이즈를 찾아내고, 이를 이용하여 기존의 일반적인 NUMA 시스템에서의 원격 캐쉬와 본 논문에서 제안한 다중 그레인 원격 캐쉬와의 상호 비교를 통해 성능상의 차이점을 비교, 분석한다. 그 후 다중 그레인 원격 캐쉬가 시스템과 응용 프로그램간의 다양한 관계 속에서도 항상 최악의 경우를 피하면서 최적의 경우와 유사한 결과를 가짐을 보인다.

키워드 : 다중 프로세서 시스템, 원격 캐쉬, 다중 그레인 원격 캐쉬, 응용 프로그램, 메모리 참조 패턴

Abstract Due to the implementation ease and alleviation of memory bottleneck effect, NUMA architecture has dominated in the multiprocessor systems for the past several years. However, because the NUMA system distributes memory in each node, frequent remote memory access is a key factor of performance degradation. Therefore, efficient design of RAC(Remote Access Cache) in NUMA system is critical for performance improvement.

In this paper, we suggest Multi-Grain RAC which can adaptively control the RAC line size, with respect to each application behavior. Then we simulate NUMA system with multi-grain RAC using MINT, event-driven memory hierarchy simulator, and analyze the performance results. At first, with profile-based determination method, we verify the optimal RAC line size for each application and, then, we compare and analyze the performance differences among NUMA systems with normal RAC, with optimal line size RAC, and with multi-grain RAC. The simulation shows that the worst case can be always avoided and results are very close to optimal case with any combination of application and RAC format.

Key words : NUMA Sytem, Remote Access Cache, Multi-Grain RAC, Application Behavior

1. 서 론

최근 발표되는 단일 프로세서 시스템의 성능 향상은 가히 획기적이라 할만하다. 하지만 응용 분야의 계산속도와 처리능력에 대한 기대치 또한 더욱 빠르게 증가하는 것이 오늘날의 현실이다. 그리고 이와 같은 상황은 다중 프로세서 시스템의 필요성으로 이어졌다[1].

이러한 다중 프로세서 시스템 중에서 대표적이라 할 수 있는 것이 공유 메모리 구조의 다중 프로세서 시스템이다. 이는 모듈성, 확장성, 신뢰성 등의 측면에서 분

† 비 회 원 : 서울대학교 전기컴퓨터공학부
leoniss@panda.snu.ac.kr
†† 종신회원 : 수원대학교 정보공학대학 컴퓨터학과 교수
stjhang@suwon.ac.kr
††† 종신회원 : 서울대학교 공과대학 전기컴퓨터공학부 교수
csjhon@riact.snu.ac.kr
논문접수 : 2003년 2월 26일
심사완료 : 2003년 6월 13일

산 메모리 구조의 다중 프로세서 시스템에 비해 제약이 있다는 단점이 있으나, 프로세서들이 강 결합성으로 연결되어 있고, 단일 주소 공간을 사용한다는 것에서 기인하는 여러 장점들을 가지고 있다. 특히 각각의 프로세서들은 공유 가능한 모든 메모리 영역에 대해 접근이 가능하고, 공유변수나 메시지에 의해 통신을 수행하는 병렬 프로그램을 작성하기가 쉽다는 상대적인 장점도 있다. 더군다나 병렬 컴파일러, 운영체제 등의 효율적인 지원 여부를 볼 때 공유 메모리 구조 다중 프로세서 시스템이 좀 더 강력한 컴퓨팅 환경을 제공해 준다고 할 것이다[2].

이와 같은 공유 메모리 구조 다중 프로세서 시스템은 메모리의 공유 또는 분산에 따른 응답 지연 시간 즉, 프로세서의 메모리 접근 요청의 발생이 있은 후, 메모리로부터의 결과 획득까지의 시간 지연의 특성에 따라 UMA(Uniform Memory Access) 구조와 NUMA (Non-Uniform Memory Access) 구조로 크게 분류할 수 있다.

UMA 구조의 다중 프로세서 시스템은 공유 메모리로의 접근이 모든 프로세서에서 동일한 시간을 가지게 되므로 프로그래밍하기에는 쉽다는 장점을 가지지만, 메모리로의 접근이 항상 동일한 일정 지연 시간이 소요된다는 점과 특정 순간에 메모리 접근이 집중되면 메모리 병목현상(Bottleneck)으로 인해 시스템의 전체 성능이 급격하게 떨어지는 문제를 가지게 되는 등 시스템의 확장성과 성능 면에서의 한계를 보이게 된다. 반면 NUMA 구조의 다중 프로세서 시스템은 UMA 구조에서 나타났던 메모리 병목현상의 단점을 극복하고자 메모리를 지역적으로 분산시켜 놓았다. 이는 프로세서들을 기준으로 볼 때 각각의 지역 메모리들이 상호 상이한 거리로 연결됨에 따라 가깝게 할당된 지역 메모리로의 접근시간은 상대적으로 먼 거리에 할당된 지역 메모리로의 접근시간에 비해 짧은 지연 시간을 갖게 된다. 이는 NUMA 구조의 중요한 특징으로서, 원격 캐쉬 (Remote Access Cache : RAC)의 필요성을 제기한 동기 부여이기도 하다.

즉 UMA 구조에 비해 NUMA 구조 다중 프로세서 시스템에서는 원격 메모리에 대한 응답 지연을 줄이고 전역 연결망의 트래픽을 감소시키는 것이 시스템의 성능 향상을 위한 필수 조건이다. 이런 연유로 원격 메모리의 접근을 최소화하기 위해 지역 메모리의 영역이 아닌 원격 메모리의 영역만을 캐싱하는 원격 캐쉬가 제안되었다[3]. 그 결과 원격 캐쉬를 가지는 NUMA 구조 다중 프로세서 시스템은 여타의 다중 프로세서 시스템에 비하여 높은 성능과 구현상의 용이도를 가지게 되었다[4].

한편, 구조적으로 우수한 시스템이라 하더라도 시스템의 성능은 주어진 응용 프로그램의 메모리 접근 패턴과 공유의 단위(Granularity of Sharing), 그리고 지역성 (Locality)과 같은 요소들에게서 많은 영향을 받는다. 또한 이와 같은 요소들은 각 응용 프로그램 별로 상이한 특징을 나타낸다[5]. 즉 시스템의 성능 향상에는 메모리에 대한 접근 지연 시간이 시스템 성능 저하의 주된 요소라는 시스템 구조에 대한 관점뿐만이 아니라, 실제 시스템 상에서 수행되는 응용 프로그램별 서로 상이한 메모리 접근 패턴에 대한 분석 역시 중요한 성능 향상의 요소라는 것이다. 이러한 메모리 접근 시간을 감소시키기 위한 원격 캐쉬의 사용과 응용 프로그램별 메모리 접근 패턴과의 관계를 고려해 볼 때, 원격 캐쉬의 라인 사이즈는 캐쉬의 Miss Rate과 메모리 트래픽에 중대한 영향을 미친다.

본 논문에서는 지점간 링크를 이용한 이중 링 스누핑 버스 다중 프로세서 시스템에서, 각 응용 프로그램별 메모리 접근에서 나타난 패턴의 변화를 추적하여 가상적으로 실행 시간에 원격 캐쉬의 라인 사이즈를 변화시킬 수 있는 다중 그레인 원격 캐쉬를 제안한다. 이하 본 논문의 구성은 다음과 같다. 2장에서는 관련연구로 기존의 적응성을 추구해온 여러 방법론들과, 본 논문의 근간이 되는 이중 링을 사용한 NUMA 구조 시스템의 전체 구조와 내부 모듈, 특히 원격 캐쉬에 대해서 알아본다. 3장에서는 다중 그레인 원격 캐쉬(Multi-Grain Remote Access Cache)의 설계 및 모듈의 동작 원리에 대해서 알아보며, 4장에서는 앞서 제시한 다중 그레인 원격 캐쉬의 성능을 모의실험을 통해 그 결과를 분석한다. 끝으로 5장에서는 이를 근거로 결론을 맺고 향후 연구 방향을 제시한다.

2. 관련 연구 및 연구 배경

2.1 동적 캐쉬 블록

본 논문에서 제안하게 될 적응형 캐쉬와 유사한 주제의 연구가 많이 진행되어 왔다. 그 중 하드웨어적인 관점에서 대표적인 것이 스트라이드(Stride) 기반의 프리페칭(Prefetching)일 것이다. 그리고 서브 블록 교체 (Sub-Block Replacement) 정책도 유사한 연구 분야라 할 것이다. 전자의 경우는 Miss Rate을 줄이기 위해서, 하드웨어적으로 스트라이드를 탐지하여 다음 요구가 발생하기 이전에 미리 데이타를 캐쉬에 불러오는 정책이다. 그리고 후자의 경우는 Miss Penality를 줄이기 위한 방법으로, 하나의 캐쉬 라인을 각각의 서브 블록으로 구분하여 Read Miss가 발생할 경우 전체 블록 중에서 해당 서브 블록만을 읽어오게 하는 방식이다.

본 논문에서 제안하게 될 다중 그레인 원격 캐쉬를

전술한 두가지 방법과 비교해 보면 다음과 같다. 우선 스트라이드 기반의 프리페칭은 스트라이드를 계산하기 위한 특별한 모듈이 필요한 것임에 반해, 다중 그레인 원격 캐쉬는 일종의 순차적 프리페칭(Sequential Pre-fetching)과 유사한 방법으로서, 스트라이드가 계산되면 계속해서 해당 간격만큼의 데이타를 불러오는 기존의 방식과 비교하여 그 양을 동적으로 증가시키면서 또한 감소시킬 수도 있다는 것이 특징이다. 다음으로 서브 블록 교체 정책과 비교해 볼 때, 제안하게 될 캐쉬는 캐쉬의 라인 내부적으로 서브 블록을 유지하는 것이 아니라, 이보다 더 큰 단위인 캐쉬의 라인과 라인 사이의 관계를 고려한다는 점에서 그 차이가 있다고 할 것이다.

프리페칭과 서브 블록 교체 정책 이외에도 캐쉬의 라인 사이즈 자체를 동적으로 변화시키고자 하는 기존의 연구도 존재한다. 특히 다중 프로세서 상에서 캐쉬 라인의 동적 변화를 추구하고자 했던 관련 연구에서는[6], 본 논문에서 제안하게 될 Split & Merge 카운터와 유사한 형태의 라인 사이즈 변경 알고리즘이 존재한다. 하지만 이 방법은 Split & Merge 카운터의 구현을 위해서 다중 프로세서 시스템의 복잡도에 핵심이라 할 수 있는 캐쉬 일관성 규약(Cache Coherence Protocol)의 변경까지 초래한다. 하지만 본 논문에서 제안하게 될 방법은 프리페칭의 특징과 동적 라인 사이즈의 요소가 서로 결합된 형태로서 캐쉬 일관성 규약의 변경을 요구하지는 않는다. 그리고 또 다른 관련 연구에서도 유사한 라인 사이즈 변경 알고리즘을 제안하였으나[7], 이는 서브 블록 교체 정책의 변형된 형태로서, 하나의 실질적인 캐쉬 라인 내에서 가상적인 내부 라인 사이즈의 변화를 추구한다. 하지만 이 역시 본 논문에서 제안하게 될 기법과 비교해 보면, 라인 내부적으로가 아닌 라인과 라인 사이에서의 변경, 즉 실질적인 내부 라인 사이즈를 기준으로 가상적인 외부 라인사이즈의 변화를 추구한다는 점에서 차이가 있다고 할 것이다.

그 외에도 캐쉬의 블록 크기를 변화시키고자 하는 동적 캐쉬의 구현에 관한 연구가 많이 시도 되었으나, 기존의 방법이 주로 단일 프로세서 시스템에서의 레벨 1 캐쉬(L1 Cache)와 레벨 2캐쉬(L2 Cache) 상에서 주로 연구되었다는 것에 비하여, 본 논문에서는 링 기반의 다중 프로세서 시스템 상에서 실험이 이루어졌으며, 특히 이와 같은 NUMA 구조상에서 상대적으로 Miss Penalty가 큰 원격 캐쉬(L3 Cache)에 초점이 맞추어졌다는 점에서 그 차이가 있다. 또한 3절에서 논의하게 될 라인 사이즈 결정자 내부의 각 모듈의 구현 방법론적인 측면에서나, 4절에서 논의하게 될 벤치마크 역시 병렬 프로그램(SPLASH-2)이라는 점에서 기존 연구와의 차이가 있다고 할 것이다. 이하 본 절에서는 연구배경(Background)과 관련된 내용으로, 본 논문의 근간이 되는 다중 프로세서 구조인 NUMA 시스템과, 이러한 시스템 상에서의 원격 캐쉬에 관련된 연구를 소개한다.

2.2 NUMA 시스템

본 논문의 시스템 모델은 계층적 버스에 기반 한 NUMA 시스템[8]으로서, 시스템 전체 구조는 그림 1과 같다. 시스템은 메모리 주소를 공유하는 다중 프로세서 시스템으로, 각 연산 노드가 단방향 지점 간 링크 두 개를 사용하여 방향 분리 이중 링(Point-to-Point Dual Ring) 형태를 띈다. 방송용(Broadcasting) 패킷의 경우,

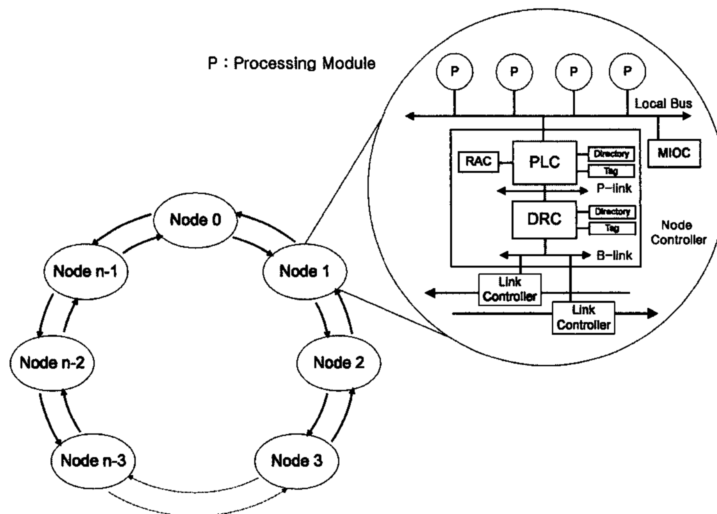

그림 1 NUMA 시스템 모델

하나의 링은 시계방향으로 전송을 하고 다른 하나의 링은 반시계 방향으로 전송을 한다. 단일 전송용(Unicasting) 패킷의 경우에는 두 가지 방향 중에서 가까운 방향을 결정하여 해당 링을 사용하여 전송한다. 특히 메모리 주소의 끝자리를 Odd/Even으로 구분하여 각각의 해당 링을 사용하여 Broadcast를 하게 구성함으로써 패킷의 전송 경로를 분산시켜 평균 응답 지연 시간을 줄였다. 지점간 링크로 사용하는 IEEE 표준 SCI 링크는 16bit 데이타 폭으로 1GByte/Sec(500MHz)의 전송률을 가진다. 이와 같은 지점간 링크는 방송 트랜잭션을 지원하므로 논리적으로 버스와 동일한 동작을 수행하게 된다. 또한 이와 같은 버스구조에 기반 한 시스템은 일반적으로 스누핑(Snooping)[9]에 의한 캐쉬 일관성 유지 방법을 사용한다.

시스템 버스로 연결된 각각의 연산 노드의 구성은 다음과 같다. 지역 버스로 연결된 4개의 프로세서 모듈, 1차 캐쉬와 2차 캐쉬, 원격 캐쉬(RAC), 지역 메모리, 입출력 시스템, 지역 버스와 노드를 연결하는 노드 제어기(PLC), 노드와 전역 링을 연결하는 링 제어기(DRC)로 구성된다. 지역 버스는 요청 트랜잭션과 응답 트랜잭션이 분리될 수 있는 버스, 즉 Split 트랜잭션을 지원하는 버스이며 지역 버스 역시 스누핑 방식에 의한 캐쉬 일관성 유지 방법을 사용한다. 지역 메모리는 분산형 공유메모리로 전체 시스템 메모리의 물리 주소 영역 중 일부를 구성하고 있다. 노드 제어기는 원격 노드로의 접근 지연 시간을 단축하기 위해 원격 캐쉬를 유지하고 있으며, 지역 버스에서 요청이 발생하면 원격 캐쉬나 전역 링크를 통한 원격 노드로부터의 데이타 제공을 담당한다. 또한 전역 링크에서 발생한 요청에 대해서는 링 제어기가 응답의 책임을 가진다. 이밖에 노드 제어기 및 링 제어기는 각각 지역 버스와 전역 링크에서 발생하는 모든 트랜잭션을 스누핑하여 메모리와 캐쉬 사이에 일관성 유지를 위한 디렉터리와 태그(Directory & Tag)를 가진다. 입출력 시스템은 프로세서에서 요구한 입출력 디바이스에 대한 읽기 및 쓰기를 수행한다. 모든 입출력 관련 트랜잭션은 지역 버스에 연결된 PCI 버스를 통해 처리하게 된다.

2.3 원격 캐쉬

그림 1과 같은 전형적인 NUMA 구조, 특히 하드웨어적으로 캐쉬의 일관성을 유지하는 CC-NUMA 구조에서는 UMA 구조에서 나타나는 메모리 병목현상의 단점을 극복하고자 메모리를 지역적으로 분산시켜 놓았다. 이렇게 분산된 여러 지역 메모리들이 모여서 하나의 전역 주소 공간을 이루게 된다. 그리고 메모리를 지역적으로 분산시켜 놓았다는 점에서 COMA 구조가 CC-NUMA 구조와 유사한 형태를 보이지만, COMA 구조에서는 분산된 각 지역 메모리(Attraction Memory : AM)를 캐쉬와 같이 필요에 따라 동적으로 이용한다. 이는 CC-NUMA 구조에서의 단점인 원격 메모리로부터의 긴 접근 지연 시간을 줄일 수 있다는 점에서 장점이 된다. 하지만 참조 미스(Reference Miss)가 발생했을 경우 각 복사본에 대한 위치 추적과, 최후 복사본의 교체시 이에 대한 확인 작업 등 캐쉬 일관성을 유지하기가 어렵고, 주로 계층적 디렉터리 구조를 사용한다는 점에서 구현이 복잡하며, 동적 메모리 공간의 관리 및 할당의 오버헤드(Overhead)를 가지는 단점이 있다[10].

이와 같이 메모리가 지역적으로 집중되어 있는 UMA 구조와 제어상의 복잡도를 가지는 COMA 구조의 단점을 보완하면서 성능상의 우위를 점하기 위한 CC-NUMA 구조의 필수 요소로서, 노드 외부로 나가는 트랜잭션의 수를 줄이기 위해 원격 메모리 주소 영역에 대해 캐싱을 하는 원격 캐쉬가 성능 향상의 필수 조건으로 대두되었다. 따라서 본 논문에서도 이와 같은 원격 캐쉬를 채택하여, 원격 메모리 접근에 대한 응답 지연 시간 감소를 통해 성능상의 향상을 꾀하고자 한다.

3. 다중 그레인 원격 캐쉬

3.1 다중 그레인 원격 캐쉬의 필요성

시스템의 성능은 주어진 응용 프로그램의 공유 단위와 지역성에 많은 영향을 받는다. 이러한 응용 프로그램은 그 특징에 따라서 지역성과 공유 패턴이 비교적 큰 프로그램도 있을 수 있으며, 비교적 큰 크기의 Working Set을 위한 공간을 필요로 할 수도 있다. 같은 논리로, 그 반대의 응용 프로그램 또한 존재 할 수 있다. 그리고 이것은 캐쉬 라인 사이즈와 밀접한 관계를 가진다. 즉 캐쉬의 라인 사이즈가 큰 경우(Coarse Grain)는 공간 지역성(Spatial Locality)적인 측면에서 유리하다고 할 수 있고, 또한 비교적 순차적으로 메모리에 접근하는 응용 프로그램에 있어서 프리페칭의 효과까지 볼 수 있어 우수한 성능을 보인다. 하지만 만약 그 크기가 지나칠 경우 불필요한 데이타까지 캐쉬에 불러들임으로 인해 False Sharing을 유발시킬 수 있으며, 이는 불필요한 캐쉬 라인의 무효화(Invalidation) 트랜잭션을 과도하게 발생시켜 오히려 시스템의 성능을 저하시킬 수 있다. 또한 통신에 의한 동기화 빈도수가 많은 경우에도 불리하다고 할 수 있다. 이와는 반대로 작은 크기의 캐쉬 라인 사이즈를 사용할 경우(Fine Grain)는 위에서 나타난 False Sharing에 의한 과도한 무효화 신호를 줄이고 또한 불필요한 데이타의 전송을 막을 수 있다는 장점을 가진다. 그 외에도 인접 노드와 통신이 발생할 경우, 패킷 자체의 오버헤드는 라인 사이즈가 큰 경우에 비해 상대적으로 크지만 동기화가 빈번히 이루어 질 경우는

오히려 유리하다고 할 수 있다. 하지만 응용 프로그램에 따라 다분히 순차적인 메모리 접근을 나타내는 경우에는 프리페칭의 효과를 볼 수 없어 이 방법이 합리적이지 않을 수 있으며, 주어진 응용 프로그램을 캐쉬에 불러들이기 위한 과도한 버스 혹은 네트워크 상의 트랜잭션을 발생시킬 수 있다는 단점이 있다[11]. 실제 오늘날에는 이런 두 가지 경우의 시스템을 모두 찾아 볼 수 있다. Coarse Grain한 시스템의 경우는 주로 페이지 단위로 구현된 시스템에서 많이 볼 수 있으며, 그 대표적인 예로 페이지 기반 소프트웨어 DSM(Distributed Shared Memory)인 Cashmere를 들 수 있다. Fine Grain한 시스템의 경우는 Load/Store 단위로 캐쉬 라인을 체크하는 Shasta를 그 예로 들 수 있다[12].

결국 문제는, 주어진 응용 프로그램의 공유 단위와 캐쉬 라인 사이즈간의 불균형인 것이다. 하지만 기존의 몇몇 연구에서 보여주듯이[13], 주어진 모든 응용 프로그램에 있어서 최적화된 크기의 캐쉬 라인 사이즈는 존재하지 않는다. 또한 응용 프로그램의 메모리 참조 패턴에 대한 사전 분석을 통해 찾아 낸 최적의 라인 사이즈라 하더라도, 그 크기가 실행 시간 중 계속 최적화 된 라인 사이즈를 유지한다는 사실을 보장하지는 못한다. 따라서 본 논문에서는 주어진 응용 프로그램의 실행 패턴을 분석하여 이에 맞게 실행 시간 중 가상적으로 캐쉬의 라인 사이즈를 변화시킬 수 있는 다중 그레인 원격 캐쉬 기법을 제안한다. 특히 원격 캐쉬의 설계에 초점을 맞춘 이유는 L1, L2 캐쉬에 비해서 L3 캐쉬로서의 역할을 하고 있는 원격 캐쉬의 경우, 상대적으로 상위 레벨 캐쉬에 비해 Miss Rate이 현저히 높고, 이러한 원격 캐쉬에서 Miss가 발생할 경우 시스템 지연의 주된 요소라 할 수 있는 네트워크를 통한 데이타 전송이 이루어져 성능 저하의 주된 요소가 된다는 점을 착안, 원격 캐쉬를 다중 그레인 형태로 설계함으로써 상대적으로 높은 Miss Rate과 이에 대한 Miss Penalty를 줄여 시스템 성능의 향상을 얻고자 함이다.

3.2 다중 라인 사이즈의 결정

그림 1에서 소개된 시스템에서 각 노드의 내부 프로세서들은 P6 버스로 상호 연결되어 있다. P6 버스 프로토콜은 Pentium Pro 프로세서[14] 이후 Intel 社에서 개발된 프로세서가 따르는 버스 규약으로, 이는 조정 단계(Arbitration Phase), 요구 단계(Request Phase), 에러 단계(Error Phase), 스누핑 단계(Snoop Phase), 응답 단계(Response Phase), 그리고 데이타 단계(Data Phase)로의 6단계에 걸쳐 각각의 트랜잭션이 처리된다. 이 경우 실제 데이타는 데이타 단계(Data Phase)에서 전송이 이루어지는데, P6 버스의 경우 그 데이타 버스의 폭이 64bit이며, 데이타 단계에 이런 64bit의 데이타 신호를 최대 4번까지 버스 상에 발생시킨다. 따라서 한 번에 처리 될 수 있는 데이타의 양은 32Byte가 된다. 본 시스템은 이러한 32Byte를 기준으로 해서 캐쉬 일관성 프로토콜이 설계되었으며, 이런 이유로 본 논문에서 제시한 원격 캐쉬의 최소 라인 사이즈는 32Byte이다.

한편, 일반적으로 프로세서의 수행 시간은 식 (1)과 같이 나타낼 수 있다.

$$CPU_{time} = IC \times (CPI_{execution} + \frac{MEM_{IC}}{IC} \times MAT_{avg}) \times Clock \quad (1)$$

IC는 전체 실행 인스트럭션의 수를 나타내며, MAT는 메모리 접근 시간(Memory Access Time), MP는 Miss Penalty를 의미한다. 이때 식 (1)에서의 평균 메모리 접근 시간은 식 (2)와 같이 표현된다.

$$MAT_{avg} = HT \times (HR + MR) + MR \times MP = HT + MR \times MP \quad (2)$$

HT는 Hit Time, HR은 Hit Rate, MR은 Miss Rate을 의미한다. 제안한 시스템에서는 프로세서 캐쉬가 L1, L2 캐쉬로 나누어져 있고 또한 원격 캐쉬까지 존재하므로 식 (2)를 식 (3)과 같이 좀 더 구체화시킬 수 있다.

$$MAT_{avg} = HT_{L1} + MR_{L1} \times (HT_{L2} + MR_{L2} \times (HT_{RAC} + MR_{RAC} \times MP_{RAC})) \quad (3)$$

식 (3)에서 보는 바와 같이 캐쉬상의 Miss가 연속적으로 발생할 경우, 해당 연산은 각 캐쉬의 하위 레벨로 내려가게 되며, 원격 캐쉬에서도 Miss가 발생할 경우 노드 밖으로의 트랜잭션이 생성되어 식 (4)와 같은 원격 캐쉬의 Miss Penalty를 가지게 된다.

$$MP_{RAC} = 2 \times (MAT + \frac{DATA_{tran.}}{Bandwidth}) \times GID \times HOP_{avg} \quad (4)$$

GID(Global Interconnection Delay)는 노드간의 지연 시간을 뜻하며, HOP는 노드간 홉 수를 나타낸다. 원격 캐쉬의 Miss Penalty는 메모리 접근 시간(MAT)과 전송 시간(Transfer Time)을 합하여, 이를 노드간의 지연 시간과 노드간의 평균 홉 수를 곱한 후, Request/Response를 고려하여 이를 두 배한 값이 된다. 식 4에서 보여 주듯이 Miss Penalty는 전송되는 데이타의 양(원격 캐쉬의 라인 사이즈)과 밀접한 관계가 있다. 그리고 이는 식 (1)에 비추어 볼 때, 전체 CPU 수행 시간에 있어서 Miss Penalty가 절대적인 영향을 미친다는 것을 의미한다. 물론 Miss Penalty가 전송되는 데이타의 양(원격 캐쉬의 라인 사이즈)과 정비례의 관계가 있는 것은 아니지만, 아무런 기준 없이 전송되는 데이타의 양을 증가시킬 경우, 오히려 CPU 수행 시간에 좋지 않은 영향을 미칠 수도 있다는 것을 뜻하며, 이는 적절한 Trade-Off 지점이 필요함을 의미한다.

소개된 NUMA 시스템은 SCI 규약[15]을 이용하여 노드간의 데이타를 전송한다. 이때 각 전송 데이타는

SCI 규약에 맞는 패킷 형태로 변환되어 지는데, 응답 패킷의 경우 Target, Command, Source, Control 등의 제어 정보를 Header 부분에 담고 그 뒤에 Data를 담는다. 이 때 Data 부분의 .크기는 SCI 규약 상 0Byte, 16Byte, 64Byte, 256Byte가 될 수 있다. 비록 SCI가 하나의 패킷으로 256Byte의 데이타 전송을 허용하지만, 원격 캐쉬의 최대 라인 사이즈가 256Byte가 될 경우, 실제 데이타 전송량이 SCI 링의 최대 전송량에 가까워져 다중 그레인 원격 캐쉬가 시스템의 병목이 될 수 있는 가능성이 크다. 따라서 적절한 Trade-Off 지점의 필요성과 SCI 규약과의 관계, 그리고 3.3절에서 제시하게 될 다중 그레인 원격 캐쉬 모듈의 설계상의 복잡도를 고려하여, 128Byte를 본 논문에서는 최대 라인 사이즈로 결정하였다.

3.3 다중 그레인 원격 캐쉬의 동작

앞서 논의한 대로 지역적으로 메모리가 각 노드별로 분산되어 있는 NUMA 시스템에서, 원격 캐쉬와 원격 메모리 사이에 데이타를 교환할 수 있는 상황은 그림 2와 같다. 전술한 바와 같이 소개된 시스템은 지역 버스 상에서 P6 버스를 사용하기 때문에, P6 버스 프로토콜의 규약에 의해 최소 32Byte 단위로 일관성을 유지한다. 또한 본 논문에서의 확장 형태인 64Byte와 128Byte 크기로의 데이타 전송도 그림 2의 중/하단부와 같이 표사할 수 있다[16].

각각의 경우는 원격 메모리와 원격 캐쉬 사이에 데이타를 전송함에 있어서 그 순서대로 최소전송/초기전송/최대전송 크기의 캐쉬 라인 사이즈를 나타낸다. 이와 같은 형태를 기존의 시스템에 적용하여 최소 전송의 단위

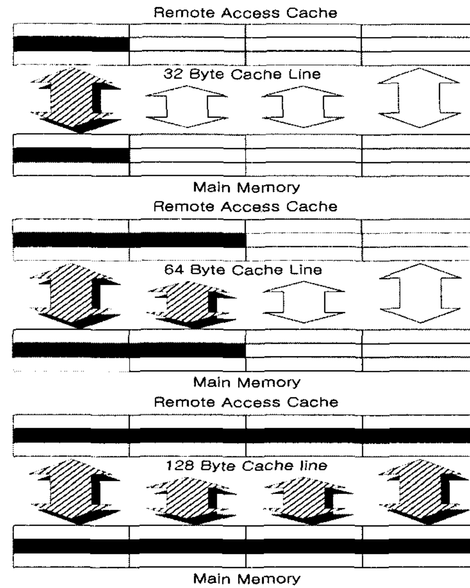

그림 2 메모리와 원격 캐쉬간의 데이타 전송

(Grain)로 32Byte, 최대 전송 단위로 128Byte, 그리고 초기의 전송 단위로 64Byte를 설정한다.

그림 2를 바탕으로 주어진 응용 프로그램의 메모리 참조 패턴을 실행 시간에 동적으로 분석하여(Dynamic Determination) 원격 캐쉬와 원격 메모리 사이에 실제 전송되는 데이타의 양, 즉 캐쉬의 라인 사이즈를 결정하기 위한 기본적 모델은 그림 3과 같다. 그림 3에서 음영으로 표시된 블록이 32Byte 크기의 현재 참조되고 있는 캐쉬 라인(Referenced Cache Line)을 나타낸다. 그리고

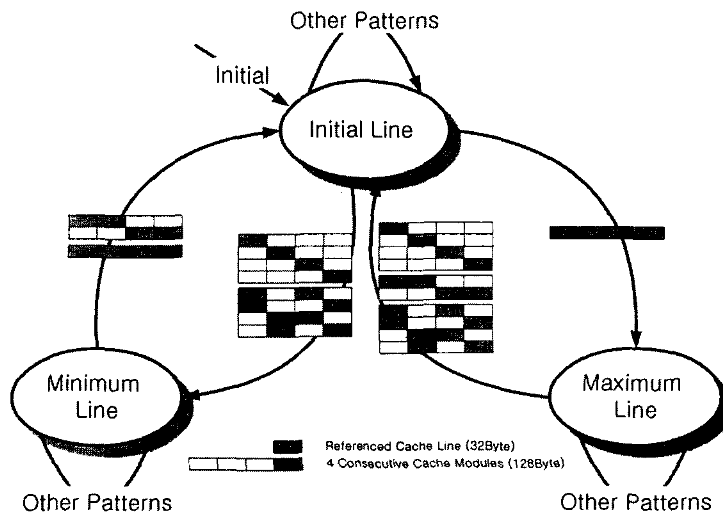

그림 3 캐쉬 라인 사이즈의 상태 변환도

그림 2에서처럼 4개의 연속된 캐쉬 모듈(Consecutive Cache Module : 128Byte)이 논리적으로 존재하게 된다. 이러한 4개의 참조 라인을 기준으로 초기 64Byte 크기로 주어진 응용 프로그램의 실행 패턴을 분석한다. 그리고 이는 다음 원격 메모리에 대한 참조 Miss 발생시, 그 요구에 대해 해당 크기의 라인 사이즈에 부합되는 양만큼의 데이타를 원격 메모리에 요구하게 된다. 그림 3에서 보는 바와 같이 음영으로 표시된 각각의 32Byte 참조 패턴이 순차적 접근, 혹은 인접 라인과의 동시 접근이 빈번하게 발생하는 경우는 그 상태가 점차 최대 라인 쪽으로 옮겨가게 되고, 그 접근이 단편적이거나 무작위적인 패턴을 나타내는 경우는 점차 최소 라인 쪽으로 옮겨가게 된다.

이와 같은 상태 변환을 수행하기 위해 지원되어야 하는 원격 캐쉬의 하드웨어 모델은 그림 4와 같다. 기본적인 구조는 기존의 일반 캐쉬 모듈과 유사하다. 그러나 기존의 고정된 라인 사이즈만을 지원하는 원격 캐쉬의 경우는 캐쉬 모듈이 하나만 존재하였지만, 본 논문에서 제안한 32Byte/64Byte/128Byte의 다중 그레인 원격 캐쉬의 지원을 위해서는 각각 32Byte 크기의 라인 사이즈를 가지는 4개의 캐쉬 모듈이 필요하다. 또한 원격 캐쉬는 Associativity를 가지지 않는다. 만약 Associativity가 있을 경우는, 캐쉬 상에서 연속된 인접 라인이라 하더라도 실제 주소 공간상에서의 연속성을 보장하지 못하기 때문이다. 따라서 원격 캐쉬는 Direct-Map 형식으로 되어 있으며 가로 방향으로의 인접한 4개의 모듈들이 연속된 주소 공간을 가진다.

그림 4에서 보는 바와 같이, 각 캐쉬 모듈의 태그와 참조 여부를 나타내는 R bit는 라인 사이즈 결정자(Granularity Determinator)의 입력으로 대입되며, 이러한 라인 사이즈 결정자가 실제 원격 캐쉬의 전송 단위를 결정하는 역할을 하게 된다. 라인 사이즈 결정자의 내부 구조는 그림 5와 같으며, 크게 라인 사이즈 지정자(Granularity Specifier), 참조 패턴 테이블(Reference Pattern Table), 그리고 Split & Merge 카운터로 구성되어 있다.

원격 캐쉬는 노드 제어기와 인터페이스가 이루어지기 때문에, 노드 제어기로부터 해당 메모리 영역에 대한 주소가 전달된다. 이를 바탕으로 원격 캐쉬는 실제 참조가 발생한 라인에 대해서 데이타를 노드 제어기에게 넘겨줌과 동시에 R bit를 1로 설정하게 된다. 이러한 R bit는 필요에 따라 라인 사이즈 결정자에 의해 사용된다. 즉 해당 캐쉬 라인에 대한 참조가 일어나는 시점에 R bit가 1로 세팅이 되고, 그 후 캐쉬 라인의 교체(Replace)가 발생할 때, 이전까지 인접 라인의 참조 여부를 나타낸 연속된 4개의 R bit의 값을 참조 패턴 테

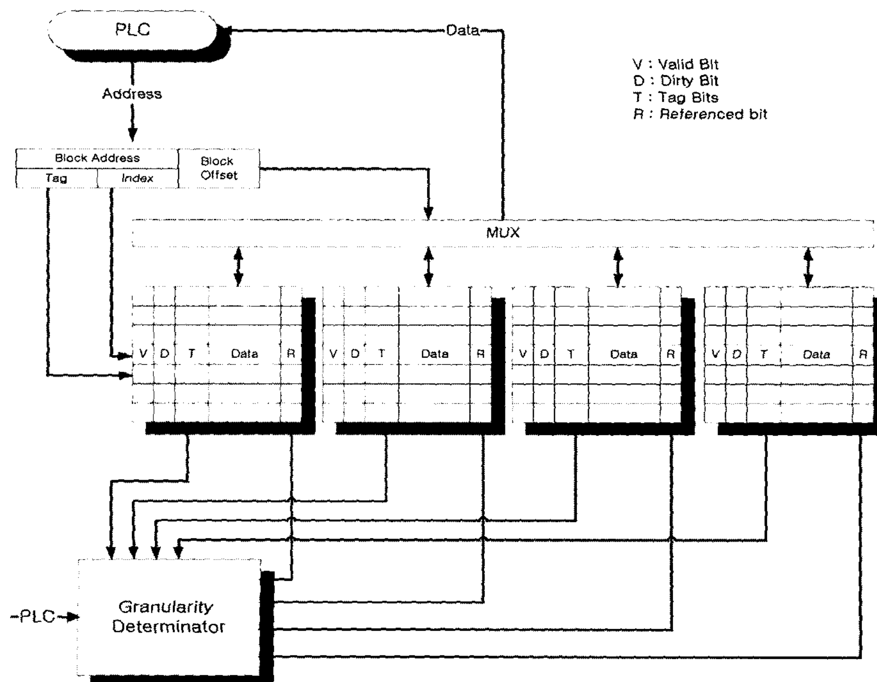

그림 4 다중 그레인 원격 캐쉬의 구조도

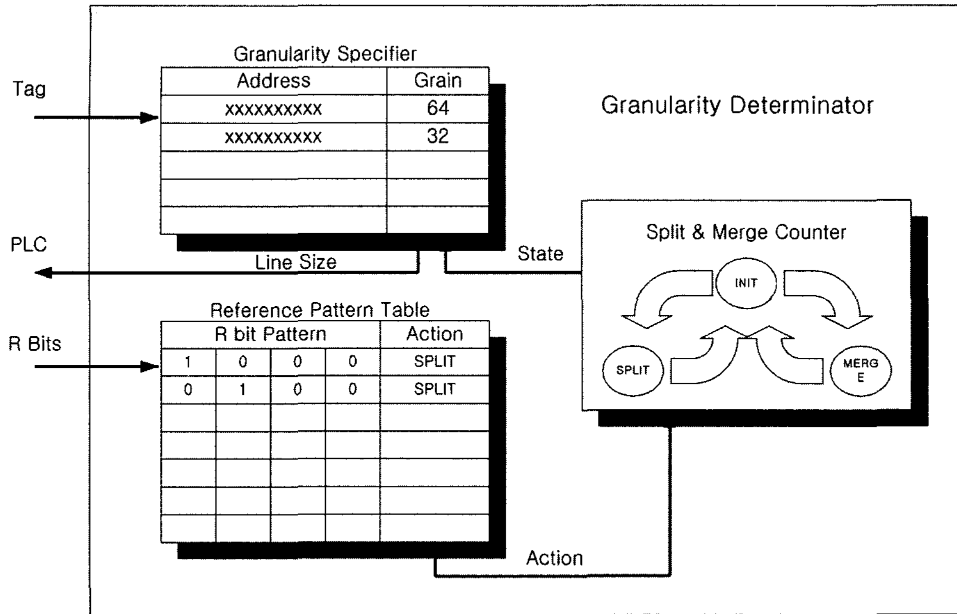

그림 5 라인 사이즈 결정자

이블에 대입하게 된다. 참조 패턴 테이블은 표 1에 따라서 적절한 동작을 출력으로 내어 보내고, 이는 라인 사이즈 지정자 내부의 Split & Merge 카운터의 입력으로 대입된다. 그리고 라인 사이즈 지정자에 저장된 해당 주소 영역에 대해 카운터에서 지정한 상태 변환 알고리즘을 수행한 후 다음 상태를 결정짓고, 라인 사이즈 지정자 내부의 Grain 필드에 그 값이 저장된다. 그리고 이는 해당 주소 영역에 대한 라인의 사이즈를 지정하는 역할을 한다. 즉, 추후 라인 사이즈 지정자는 해당 라인에 대해서 Read Miss 혹은 Write Miss가 발생했을 때, 요청된 라인의 주소를 이용하여 그 값이 라인 사이즈 지정자 내부에 저장되어 있는지 여부를 확인하게 된다. 만약 저장되어진 라인에 대해서는 해당 라인의 Grain 필드 값만큼의 데이타를 읽어오도록 노드 제어기에게 지시한다.

표 1에 연속된 4개의 R bit 입력에 대한 참조 패턴 테이블의 값과 동작이 정의되어 있다. 각각의 조합에 따라서 SPLIT 혹은 MERGE의 동작을 수행하게 되며, 특히 INIT 동작의 경우는 현재 라인이 Maximum Line인 경우는 SPLIT을, Minimum Line인 경우는 MERGE를, 그리고 Initial Line인 경우는 Other Pattern으로 간주하여 동작을 취한다.

그림 6은 참조 패턴 테이블의 출력을 받아들이는 Spit & Merge 카운터의 상태 변환을 나타낸다. 모두 5 상태로 이루어져, 3bit 카운터로의 구현이 가능하다. 초기의 INIT 상태에서 캐쉬 라인의 교체가 발생할 경우,

표 1 참조 패턴 테이블

인접 R bit의 값				동작
1	0	0	0	SPLIT
0	1	0	0	SPLIT
0	0	1	0	SPLIT
0	0	0	1	SPLIT
1	0	1	0	SPLIT
1	0	0	1	SPLIT
0	1	1	0	SPLIT
0	1	0	1	SPLIT
1	1	1	1	MERGE
1	1	0	0	INIT
0	0	1	1	INIT
Other Patterns				No Action

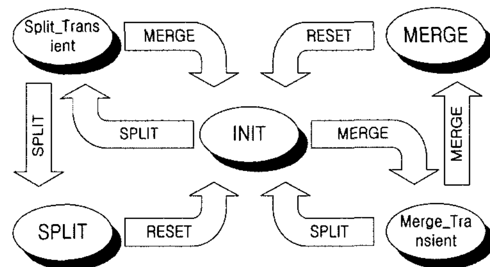

그림 6 SPLIT & MERGE 카운터 상태 변환도

해당 라인이 포함된 연속된 인접한 4개의 R bit를 입력으로 받는다. 이를 표 1에 따라 그림 6과 같이 SPLIT 혹은 MERGE 상태로의 상태 변환을 수행한다. 그리고

이는 다음 해당 라인의 요구가 발생할 경우, 라인 사이즈 지정자에 의해 해당 캐쉬 라인의 사이즈를 결정짓는 역할을 한다. 최종적으로 SPLIT 상태로의 상태 변환을 수행한 경우는 기존의 라인 사이즈가 64(128)Byte인 경우 32(64)Byte로 변환되며, 기존 라인 사이즈가 32Byte인 경우는 더 이상의 변환을 수행하지 않는다. 그리고 해당 캐쉬 라인의 R bit를 다시 초기화시킨다. 이와 반대로 MERGE 상태로 최종적 상태 변환을 수행한 경우는 32(64)Byte에서 64(128)Byte로 변환되며, 기존 라인 사이즈가 128Byte인 경우는 더 이상의 변환을 수행하지 않는다. 이 경우도 해당 캐쉬 라인의 R bit를 초기화시킨다. 즉 R bit의 값이 1인 경우는 Split 혹은 Merge를 수행한 이후 다시 재차 참조가 이루어졌다는 것을 의미하며, 0은 그렇지 않는 경우를 나타낸다. 실제 SPILT과 MERGE 상태로의 상태 변환을 수행하기에 앞서, 중간 상태로 Split_Transient와 Merge_Transient 상태를 둔 이유는 전체 응용 프로그램의 메모리 참조 패턴 분석 도중, 국부적으로 혹은 임시적으로 나타나는 Grain 크기의 변화를 무시하기 위한 수단이다. 이는 과도한 캐쉬 라인 사이즈의 변화를 막기 위함이며, 캐쉬 라인 사이즈의 변경이 시스템 성능에 악영향을 미치지 않게 하기 위해서이다.

이상의 동작을 PLC에서 생성되는 Address bit의 구성 관점에서 생각해 보면 다음과 같다. 예를 들어 256Kbit의 원격 캐쉬를 사용하고 현재 라인 사이즈를 32byte로 가정할 경우, Tag 17bit, Index 10bit, Byte_Offset 5bit로 Address bit(총 32bit)를 구성하게 된다. 이와 같은 상황에서 라인 사이즈가 Split & Merge Counter 알고리즘에 의해서 64byte와 128byte로 변경될 경우, 각각 Tag 17bit, Index 9bit, Byte_Offset 6bit 그리고 Tag 17bit, Index 8bit, Byte_Offset 7bit로 다시 재구성 되게 된다. 이처럼 본 논문에서는 고정된 각각의 32byte 기본 라인에 대해서 이를 최대 4개까지 하나의 Set으로 구성하고, 이와 같이 해당 Set이 미리 정의된 상태에서 최대 128byte까지의 확장을 Address bit 구성의 변경을 통해서 구현하고자 하였다. 그리고 이때 변경된 값은 추후 다시 재 변경되기 전까지 모든 주소영역에 대해 같은 라인 크기 값을 유지하게 된다. 참고로 본 논문에서는 기본 캐쉬 라인(32byte)의 소속 Set이 수행 도중 임의로 변경되거나, 혹은 캐쉬의 세부 주소영역 별로의 서로 다른 라인 크기를 허용하지는 않는다. 이는 구현상의 용이함과 성능상의 이득과의 적절한 Trade-Off를 추구하기 위함이다.

4. 시스템의 성능 분석

이 절에서는 다중 그레인 원격 캐쉬의 성능을 평가하기 위한 모의실험을 수행하고, 그 결과를 분석한다. 다중 그레인 원격 캐쉬의 성능을 평가하기에 앞서, 먼저 고정된 크기의 다양한 원격 캐쉬 라인 사이즈와 서로 다른 원격 캐쉬 사이즈를 인자로 주어 모의실험을 수행하며, 이를 통해 각 응용 프로그램별로 고정된 크기의 최적화 된 원격 캐쉬의 라인 사이즈를 찾아본다. 그 후 다중 그레인 원격 캐쉬의 성능을 모의실험하여, 이를 기존 32Byte 크기의 고정된 원격 캐쉬 라인 사이즈를 가지는 NUMA 시스템과 비교한 후, 다중 그레인 원격 캐쉬가 어느 정도의 성능 향상을 보이는가를 분석한다.

4.1 모의실험 환경 및 인자

본 논문에서는 성능 평가 도구로서 프로그램 구동형 시뮬레이터인 MINT[17]를 사용하여 모의실험을 수행한다. MINT는 다중 프로세서 환경을 메모리 구조와 더불어 프로그램 구동 방식으로 시뮬레이션 할 수 있도록 한 소프트웨어 패키지라 할 수 있다. 모의실험 환경에서 각 노드는 500MHz 클록의 CPU를 가지며, 이 CPU가 100MHz의 지역 버스에 연결되어서 동작을 하게 된다. 또한 각 노드를 연결시켜주는 단방향 지점간 링크는 500MHz로 동작하고, 16bit 전송이 가능하므로 1GByte/Sec의 대역폭을 가진다. 모의실험에 사용된 여러 시스템 인자 값에 대한 요약은 표 2와 같다.

표 2 모의실험 인자

인자	값
프로세서 수	32 개
한 노드내의 프로세서 수	2 개
노드 수	16 개
프로세서 캐쉬의 크기	16KB
원격 캐쉬의 크기	64KB, 128KB, 256KB, 512KB
원격 캐쉬의 라인 크기	32Byte, 64Byte, 128Byte
노드간 링 연결 클록 속도	500MHz
프로세서 캐쉬 접근 시간	2 Processor Clock
프로세서 클록 속도	500MHz
노드내 지역버스 클록 속도	100MHz
노드간 상호 연결 망 데이타 폭	16bit
노드내 지역 버스 데이타 폭	64bit

4.2 벤치마크 프로그램

본 논문의 모의실험 입력 프로그램으로 사용된 것은 SPLASH-2[18]에서 제시된 병렬 프로그램이다. SPLASH-2 벤치마크 프로그램은 과학, 공학, 그래픽 등의 분야에서 사용되는 계산유형을 포함하는 여러 개의 응용 프로그램들을 제공한다. 본 연구에서 사용된 각각의 SPLASH-2 벤치마크 프로그램은 표 3과 같다.

FFT는 Fast Fourier Transformation을 수행하는 프로그램으로, 전치 행렬을 구하는 과정에서 모든 프로세서들 간에 상호적인 통신을 발생시킨다. LU는 N×N 조밀 행렬 A를 하삼각 행렬 L과 상삼각 행렬 U의 곱

표 3 벤치마크 입력 크기

응용 프로그램	인자
FFT	64K Complex Doubles
LU	512×512 Matrix (16×16 Blocks)
BARNES	8K Particles(Bodies)
RADIX	1M Key Integers, Radix 1024

L×U로 나타내기 위해서 분할하는 프로그램이다. BARNES는 Hierarchical N-body Problem 방식을 사용하여 각 입자들 사이에 상호 작용을 구하는 프로그램이다. RADIX는 Radix Sort를 수행하는 프로그램으로, 정렬 과정에서 발생하는 치환이 모든 프로세서 사이에 상호적인 통신을 발생시킨다.

4.3 모의실험 결과 및 분석

이 절에서는 3장에서 제시한 다중 그레인 원격 캐쉬에 대해 각 응용 프로그램별로의 모의실험을 수행한다. 응용 프로그램의 효율적 실행을 위한 원격 캐쉬의 라인 사이즈를 결정짓는 방법은 기본적으로 다음과 같이 나누어 볼 수 있다.

• Profile-Based Determination

주어진 응용 프로그램에 대해서 미리 특정 캐쉬 시뮬레이터를 통한 모의실험을 수행한 후, 그 결과를 바탕으로 최적화된 원격 캐쉬의 라인 사이즈를 구하는 방식이다.

• Compiler-Based Determination

특수 목적용으로 제작된 컴파일러의 도움을 받아서, 소스 코드 레벨에서 메모리 접근 패턴에 대한 사전 분석을 수행한 후, 컴파일러가 적정 크기의 원격 캐쉬의 라인 사이즈를 구하는 방식이다.

• Dynamic Determination

주어진 응용 프로그램을 실행시키고, 이를 특정 하드웨어 모듈이 메모리 접근 패턴을 추적하여 동적으로 원격 캐쉬의 라인 사이즈를 재구성하도록 하는 방식이다.

본 논문에서는 위의 방법들 중에서 Profile-Based Determination과 Dynamic Determination 방법을 사용하여 원격 캐쉬의 라인 사이즈를 결정한다. 4.3.1절에서는 Profile-Based Determination 방식을 사용하여 각 응용 프로그램별 최적의 라인 사이즈를 알아보며, 4.3.2절에서는 Dynamic Determination 방식을 사용하는 다중 그레인 원격 캐쉬의 성능을 분석하여 이를 Profile-Based Determination 방식과 기존의 시스템과 상호 비교한다.

4.3.1 Profile-Based Determination 결과

Profile-Based Determination 방식으로 앞서 제시한 SPLASH-2 응용 프로그램들의 캐쉬 라인 사이즈별 Miss Rate을 나타낸 것이 그림 7에서 그림 10까지 제

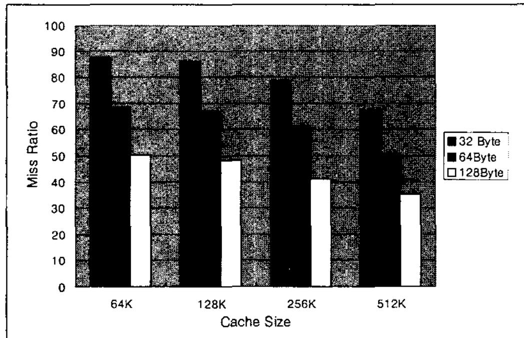

그림 7 FFT Miss Ratio

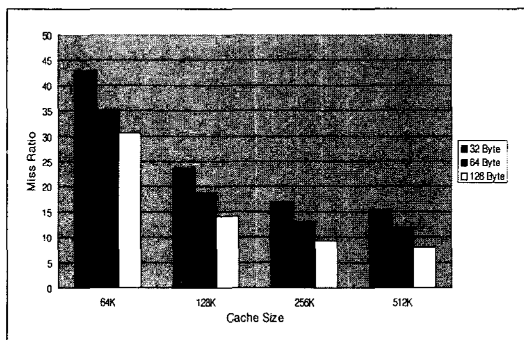

그림 8 LU Miss Ratio

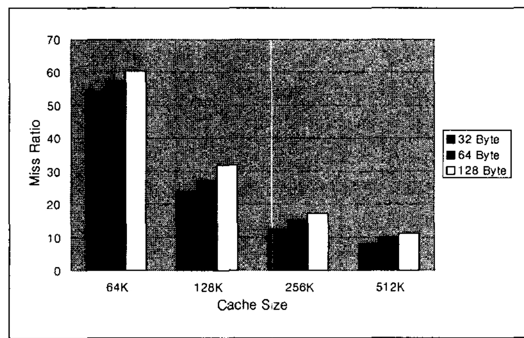

그림 9 BARNES Miss Ratio

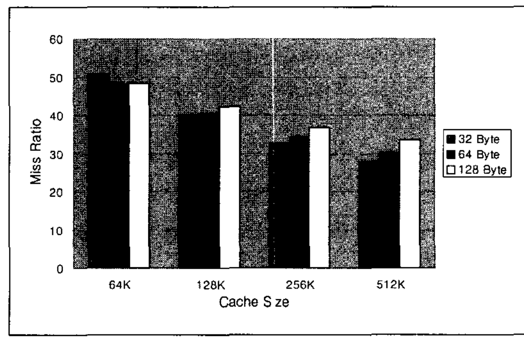

그림 10 RADIX Miss Ratio

시되어 있다. 각각의 경우 가로축은 원격 캐쉬의 크기이며, 세로축은 Miss Rate을 나타낸다. 각 원격 캐쉬 크기에서 32Byte, 64Byte, 128Byte로의 원격 캐쉬의 라인 사이즈에 변화를 주었으며, 그 결과 각 응용 프로그램별 최적의 원격 캐쉬 라인 사이즈를 구하였다.

먼저 그림 7과 그림 8에서 보는 바와 같이, FFT와 LU의 경우는 각각의 크기를 가지는 원격 캐쉬 상에서 원격 캐쉬의 라인 사이즈가 증가할수록 Miss Rate이 점차 줄어드는 경향을 나타낸다는 것을 알 수 있다. 이와는 반대로 그림 9와 그림 10의 BARNES와 RADIX의 경우는 각각의 크기를 가지는 원격 캐쉬 상에서 원격 캐쉬의 라인 사이즈가 증가할수록 Miss Rate이 오히려 증가하는 경향을 가짐을 알 수 있다. 다시 말해 FFT/LU와 BARNES/RADIX는 동일 원격 캐쉬의 크기 하에서 서로 다른 라인 사이즈에 대해 상반된 결과를 나타낸다는 것이다. 이상의 결과에서, FFT/LU의 경우는 응용의 특성 자체가 다분히 Coarse Grain하다는 것을 예상할 수 있으며, 이는 Processor가 메모리에 대해서 한번에 연속적인 라인에 대한 순차적 메모리 접근이 이루어지는 경향이 있고, 따라서 프리패칭으로 인한 Miss Rate 감소의 이득을 보았다고 할 수 있다. 이와는 반대로 BARNES/RADIX의 경우는 응용의 특성 자체가 상대적으로 Fine Grain하다는 것을 알 수 있으며, 또 그러한 Fine Grain한 단위의 Read/Write 연산이 여러 메모리 영역에 걸쳐 흩어져서 발생한다고 예상할 수 있다. 따라서 이런 응용은 라인 사이즈가 커감에 따라 False Sharing에 의한 Miss Rate이 증가했다고도 볼 수 있다.

다음으로 원격 캐쉬의 크기와 라인 사이즈별 Miss Rate의 관계에 대해 고찰해 본다. 그림 7의 FFT 경우는 128Byte 라인 사이즈를 가지는 64K의 원격 캐쉬가 32Byte의 라인 사이즈를 가지는 512K 원격 캐쉬보다 오히려 Miss Rate적인 측면에서 더 우수하다는 것을 알 수 있다. 이는 메모리가 전체 시스템의 중요한 자원이라는 점을 감안할 때, 캐쉬의 크기도 중요하지만 캐쉬의 적절할 라인 사이즈 또한 중요한 고려 대상임을 나타내는 결과라 하겠다. 이에 비해 그림 8의 LU의 경우는 캐쉬 라인의 크기가 증가함에 따라 Miss Rate이 감소하는 경향을 나타내지만, 캐쉬의 라인 사이즈에 많은 영향을 받는 FFT와 비교할 경우 LU는 캐쉬의 라인 사이즈보다는 캐쉬의 실제 크기에 더 많은 영향을 받는다는 것을 알 수 있다. 이는 LU의 경우도 FFT처럼 비교적 순차적인 메모리 접근이 이루어지지만, LU의 특성 자체가 FFT보다 훨씬 더 큰 Working Set을 가지는데서 오는 결과라 하겠다. 이러한 캐쉬의 크기와 라인 사이즈와의 관계에서 나타나는 결과는, 그림 9와 그림 10의 BARNES와 RADIX의 경우도 마찬가지로, BARNES는 캐쉬의 라인 사이즈가 줄어들수록 Miss Rate적인 측면에서 더 좋은 성능을 나타내지만, 그보다는 캐쉬의 실제 크기에 더 많은 영향을 받는다는 것을 알 수 있고, RADIX의 경우는 32Byte/256K 경우와 128Byte/ 512K 경우에서 알 수 있듯이 작은 Working Set의 응용으로, FFT처럼 캐쉬의 크기보다 캐쉬의 라인 사이즈에 보다 더 많은 영향을 받는다는 것을 알 수 있다.

이상의 결과에서, 캐쉬의 라인 크기의 변화가 경우에 따라서 캐쉬의 크기 자체의 변화보다 더 큰 장점이 될 수 있다는 흥미로운 사실과 더불어, FFT/LU의 경우는 128Byte가, BARNES/RADIX는 32Byte가 주어진 환경 하에서의 최적의 라인 사이즈라는 것을 본 모의실험을 통해 알 수 있다.

4.3.2 Dynamic Determination 결과

본 절에서는 다중 그레인 원격 캐쉬를 구현한 Dynamic Determination의 모의실험에 대한 결과를 제시하며 이를 분석한다. 먼저 다음과 같은 분석을 수행한다. 4.3.1절에서 알 수 있듯이 각 응용 프로그램별로, 또한 각각의 캐쉬 라인 사이즈에 따라서 Miss Rate이 현저히 차이가 남을 알 수 있다. 이러한 차이를 시각화 시키기 위해, 식 (5)를 통해 정규화된 Miss Rate의 차이(Normalized Miss Rate Difference : NMD)를 구하였다.

$$NMD(\%) = \frac{(MR_{WC} - MR_{BC})}{MR_{BC}} \times 100(\%) \qquad (5)$$

NMD는 Normalized Miss Rate Difference를 나타내며, MR(Miss Rate)의 BC와 WC는 각각 최선과 최악의 경우를 나타낸다. 이러한 식 (5)에 그림 7에서 그림 10까지의 결과를 적용시켜 도표화한 것이 그림 11이다.

도표에서 보여 지듯이 각 응용별로 최선과 최악의 경우에 Miss Rate의 차이가 FFT/LU의 경우는 평균

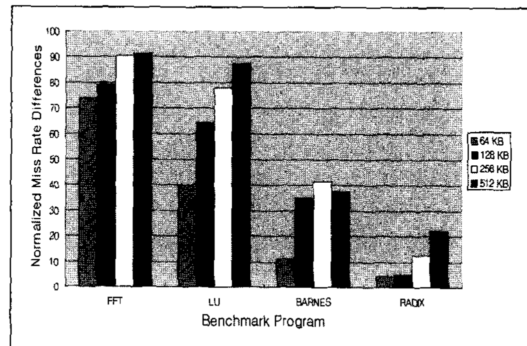

그림 11 각 응용 프로그램별 NMD

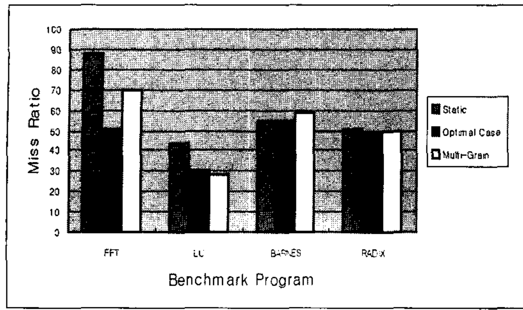

그림 12 64K RAC

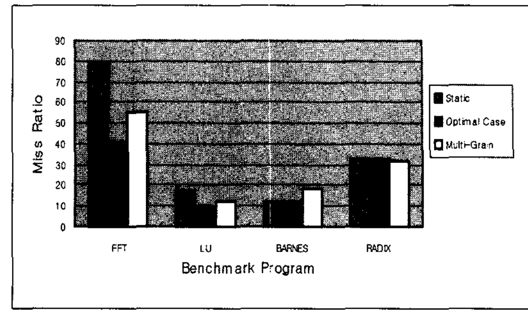

그림 14 256K RAC

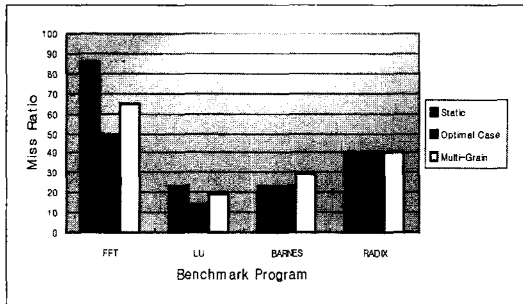

그림 13 128K RAC

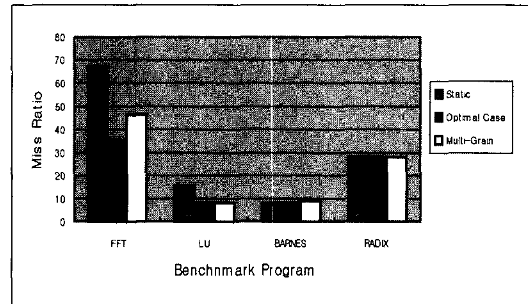

그림 15 512K RAC

75%이상, BARNES/RADIX의 경우는 20%이상 차이가 나타남을 알 수 있다. 또한 각각의 응용별로 원격 캐쉬의 크기가 증가함에 따라 그 % 수치 또한 더욱 증가하고 있다는 것을 알 수 있다. 이는 메모리의 크기가 점점 증가하는 현실과 비추어 보면 그 차이가 앞으로도 더 늘어날 것이라고 예상 할 수 있으며, 이러한 사실은 각 응용 프로그램별 적정 캐쉬 라인 사이즈의 선정이 무엇보다도 중요함을 보여 주는 결과라 할 것이다.

그림 11에서 볼 수 있듯이 캐쉬의 적정 라인 사이즈가 중요함에도 불구하고, 모든 응용에 있어서 최적의 결과를 보이는 라인 사이즈란 존재하지 않는다. 따라서 다중 그레인 원격 캐쉬를 사용함으로 해서, 실행 시간에 동적으로 캐쉬의 라인 사이즈에 변화를 주는 것이 무엇보다 중요하다. 그림 12에서 그림 15까지는 다중 그레인 원격 캐쉬를 사용했을 경우를 모의실험한 도표로서, 가로축은 각각의 응용 프로그램들이며, 세로축은 Miss Rate을 나타낸다. 그리고 이를 64K, 128K, 256K, 512K의 크기를 가지는 원격 캐쉬 상에서 모의실험을 하였다. Static은 32Byte의 고정된 크기의 라인 사이즈를 가지는 기존의 시스템을 의미하며, Optimal Case는 4.3.1절에서 구한 최적의 경우를 의미한다. 끝으로 Multi-Grain은 다중 그레인 원격 캐쉬를 적용했을 경우를 의미한다.

FFT의 경우, 거의 모든 크기에서 Multi-Grain을 사용한 방식이 기존의 Static 시스템 보다 평균 12% 정도의 우수한 성능을 나타내는 것을 알 수 있다. LU의 경우도 마찬가지로, FFT와 비슷한 형태를 보이지만 그 성능상의 이득은 FFT보다 더 우수하다. 또한 LU는 64K/512K 크기를 가지는 원격 캐쉬의 경우 Optimal Case와 상당히 유사하거나 약간 더 우수한 성능을 보인다는 것을 알 수 있다. 이처럼 Multi-Grain 방식이 Optimal Case에 비해 좀 더 우수한 성능을 보이는 이유는, Optimal Case에 비해 Multi-Grain 방식은 응용 프로그램의 실제 수행 시에 나타나는 작업 분배(Job Distribution)나 동기화(Synchronization)와 같은 일시적 Working Set 혹은 지역성의 변화에 대한 추적이 가능하며, 이를 그 결과에 반영 할 수 있는 능력이 있기 때문이다. BARNES와 RADIX의 경우는 그림 9와 그림 10에서 보았듯이 32Byte가 Optimal Case이고 또한 이 값이 기존의 Static 시스템의 값이다. 따라서 Multi-Grain의 결과를 최악의 경우와도 비교해 보아야 한다. 이와 같은 그림 9와 그림 10과의 관계를 고려한 내용이 표 4에 정리되어 있다. 각각의 캐쉬의 크기에 대해서 최적/멀티 그레인/최악의 경우를 나타낸 표이다. 표 4에서 보는 바와 같이 BARNES는 FFT와 유사한 결과로서, 최악의 경우보다 성능의 향상을 보이면서, 최적의 경우

표 4 응용 프로그램의 Miss Rate 분석

구분	BARNES			RADIX		
	Optimal *(Static)*	Multi-Grain	Worst	Optimal *(Static)*	Multi-Grain	Worst
64K	54.1	58.9	60.3	48.1	49.4	50.8
128K	23.7	28.3	31.7	40.2	***40.0***	42.1
256K	12.5	17.9	17.1	32.7	***31.2***	36.8
512K	8.1	9.4	11.1	27.8	28.0	33.6

구분	FFT			LU		
	Optimal	Multi-Grain	Worst *(Static)*	Optimal	Multi-Grain	Worst *(Static)*
64K	50.3	70.0	87.4	30.6	***28.2***	42.9
128K	48.3	65.1	86.0	14.0	19.6	23.6
256K	41.2	55.1	78.5	9.2	12.0	17.0
512K	35.2	46.3	67.6	8.7	***8.3***	15.3

와에 중간정도의 성능을 보였다. RADIX의 경우는 더 우수한 경우로서 LU에서처럼 Multi-Grain 방법이 최악의 경우보다 성능이 우수하면서, Optimal Case와 상당히 유사하거나 경우에 따라서 약간의 성능상의 우수함을 찾을 수 있다. 이 역시 작업 분배나 동기화와 같은 일시적 공유 단위의 변화를 추적 할 수 있는 다중 그레인 원격 캐쉬의 장점을 보여주는 예라 할 것이다. 참고로 표 4의 하단에 FFT와 LU의 경우도 최적의 경우와 최악의 경우와의 상관관계를 비교하여 놓았다. 표 4에서 알 수 있듯이 기존의 Static 시스템이 응용에 따라서 최적의 경우이기도 하고 최악의 경우이기도 하다. 대체로 256K BARNES를 제외한 나머지 모두는 응용 프로그램과 캐쉬의 크기, 그리고 각각의 라인 사이즈와의 다양한 조합에 관계없이 항상 최악의 경우를 회피하고 최적의 경우와 성능의 유사성을 보이며, 4가지 조합(italic)에 있어서는 최적의 경우보다 더 좋은 성능도 올릴 수 있었다.

그림 16에서 그림 19는 각 응용 프로그램의 수행 시간을 나타낸다. 가로축은 각각의 응용 프로그램들이며, 세로축은 수행 시간을 나타낸다. 그리고 이를 64K, 128K, 256K, 512K 크기의 원격 캐쉬에 대해 각각 모의 실험을 수행하였다. 각 응용 프로그램의 실행 시간은 그

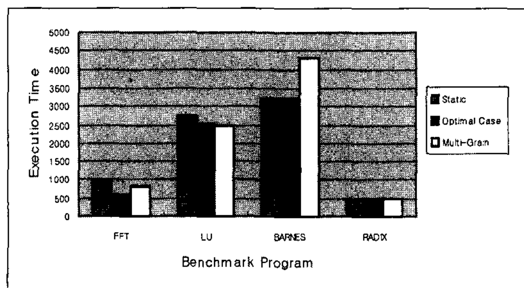

그림 16 64K RAC

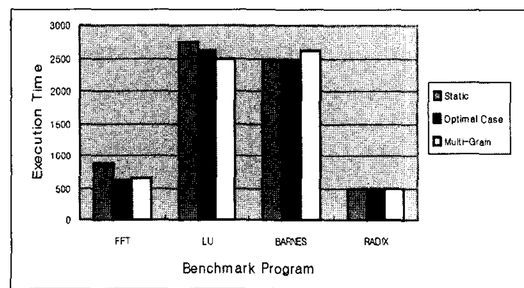

그림 18 256K RAC

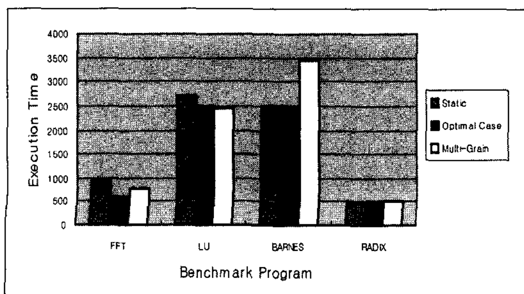

그림 17 128K RAC

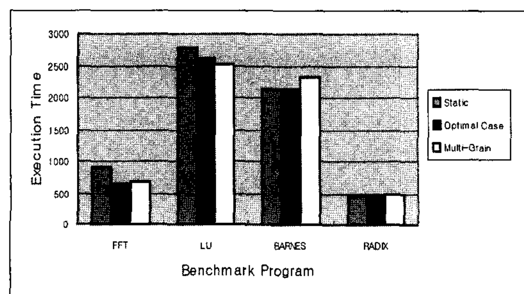

그림 19 512K RAC

림 12에서 그림 15에서 나타난 Miss Rate과 상당히 유사한 패턴을 보인다는 것을 도표를 통해서 알 수 있다. 이는 결국 Miss Rate이 Miss Penalty에 미치는 영향이 지대하며, 그 결과 프로그램 전체의 수행 시간에까지 영향을 미친다는 것을 의미한다.

FFT의 경우는 Multi-Grain의 경우가 Static보다 더 우수하면서 실행 시간이 Optimal Case에 보다 더 근접함을 알 수 있다. 이는 FFT가 Multi-Grain의 효과를 반영하기에 적합한 응용 프로그램임을 뜻한다. 또한 LU의 경우도 Miss Rate에서 나타난 패턴과 유사한 실행 시간의 변화를 나타낸다. 그리고 실행 시간도 Miss Rate에서처럼 Optimal Case보다 더 우수한 성능을 보인다. 이는 전술한 바와 같이 Optimal Case라 하더라도 응용 프로그램의 수행 도중에 항상 최적의 라인 사이즈를 보장하지는 못한다는 것을 의미하며, 상대적으로 Multi-Grain 방식은 이를 반영할 수 있는 능력이 있다는 것을 뜻한다. RADIX의 경우도 Miss Rate적 측면에서 볼 때처럼, 실행 시간에서도 Multi-Grain의 경우가 Static한 경우, 즉 Optimal Case와 상당히 유사함을 알 수 있다. BARNES의 경우도 RADIX와 마찬가지로 기존의 Static한 경우가 이미 최적의 경우였다. 그리고 역시 Miss Rate적 측면에서 볼 때처럼 최적의 경우와 최악의 경우의 중간 정도의 실행 시간이 소요된다는 것을 알 수 있다. 하지만 그림 16과 그림 17의 BARNES를 수행한 RAC의 수행 시간이 그림 12와 그림 13의 BARNES의 Miss Rate과 비교하여 실행 시간에서의 차이가 비율적으로 더 많이 나타남을 알 수 있는데, 이는 다음과 같은 측면에서 분석해 볼 수 있다. 첫째로 Multi-Grain을 유지하기 위한 Dynamic 오버헤드가 작용했다고 볼 수 있다. 하지만 링 구조의 NUMA 시스템에서는 링을 통한 데이타 전송이 시스템 지연의 절대적 요소라 봤을 때, Multi-Grain을 유지하기 위한 캐쉬의 적응 시간(Adaptive Time)의 오버헤드는 전체 수행 시간에서 무시 될 수 있음을 의미한다. 둘째로 Multi-Grain을 유지하기 위한 카운터 Algorithm이 특정 응용 프로그램에서는 효율성이 떨어질 수도 있다는 것이다. 이는 기존의 단순 2 bit 카운터, 2 bit 히스토리 카운터 등 다양한 카운터 적용의 필요성을 의미하기도 한다. 하지만 모든 응용 프로그램 별로 각각의 카운터를 구현한다는 것은 사실상 불가능하므로, 일반적으로 가장 우수한 성능을 나타내는 카운터의 구현이 효과적일 것이다. 끝으로 프리페칭으로 인한 캐쉬의 일관성 충돌(Coherence Conflict)을 원인으로 들 수 있다. 즉 다른 프로세서가 이를 수정(Modified) 상태로 보유하고 있는 캐쉬의 라인을 특정 프로세서가 공유(Shared) 상태의 라인으로 계속 프리페칭해 오는 과정에서, 해당 라인이 추후 다시 무효화 신호를 받음으로 해서 생기는 오버헤드를 의미한다. 이 역시 응용 프로그램별로 다양한 결과를 보일 수 있으나 일반적으로 공유 상태의 라인은 공유상태로, 수정 상태의 라인은 수정 상태로 상태 전환을 하는 것으로 보았을 때[19], 실제 일관성 충돌의 발생 비율은 극히 희박하다고 할 수 있다.

이상의 결과를 종합적으로 분석 해 보면, FFT의 경우는 본 논문의 아이디어를 가장 잘 반영할 수 있는 아주 적절한 응용의 형태로서, 언제나 최악의 경우를 피하면서 알고리즘의 추후 개선으로 인한 성능 향상의 소지가 앞으로도 충분히 있다고 볼 수 있는 응용이다. 그리고 LU의 경우는 오히려 최적의 경우보다 더 성능이 우수하게 나온 응용으로서 최적의 경우가 언제나 실행 시간 전체적인 관점에서 항상 최적의 경우가 아닐 수 있다는 것을 보여준 좋은 예이자, 본 논문에서 제시한 다중 그레인 원격 캐쉬의 우수성을 보여준 좋은 응용의 예라 할 것이다. 이상의 두 응용 프로그램은 전체적으로 2-30% 정도의 성능 향상을 가져왔다. RADIX의 경우는 기존의 배경 시스템으로 사용한 형태인 32Byte 라인 사이즈가 최적의 경우라는 것이 4.3.1절에서 밝혀졌다. 따라서 RADIX의 경우도 다중 그레인 원격 캐쉬가 어떠한 시스템 상에서나 최악의 경우를 피하면서 최적의 경우와 유사하게 혹은 더 우수하게 수행된다는 것을 알 수 있다. BARNES의 경우는 다른 연구에서도 보여 지듯이 응용의 메모리 접근 패턴이 상당히 불규칙함을 알 수 있는 독특한 응용으로서[18], 응용의 접근 단위가 심각하게 자주 변화 할 경우 오히려 적응형 원격 캐쉬를 수행하지 말고 기존의 정적 캐쉬와 같은 방법으로 수행토록 하는 모드 변환을 생각해 볼 수 있을 것이다. 그러나 BARNES 역시 그 결과 값에 있어서는 표4에서처럼 대체로 최적의 경우와 최악의 경우의 사이에서 수행된다는 것을 알 수 있다.

결론적으로 말하자면, 본 논문에서 제시한 방법은 일종의 동적인 기법으로서 대체적으로 최악의 경우를 피하면서, 최적의 경우에 유사하게 수행시킬 수 있다는 점에서 장점이 있다고 할 것이다. 즉 기존의 시스템이 어떤 크기의 캐쉬 라인 사이즈를 가지던, 혹은 어떤 메모리 참조 특성을 지닌 응용 프로그램을 수행시키던 상관없이 항상 최악의 경우를 피하면서 최적의 경우에 근접해 가는 수행 결과를 보인다는 것이다.

5. 결론 및 향후 과제

응용 분야의 계산속도와 처리능력에 대한 기대치의 증가로 인해 다중 프로세서 시스템의 필요성 제기 되었고, 그 결과 NUMA 시스템이 지난 수년간 전형적인 다중 프로세서 시스템으로 자리를 잡아왔다. 그리고 이러

한 NUMA 시스템에 원격 캐쉬의 개념을 도입함으로써, UMA 구조나 COMA 구조에 비해 성능과 구현상의 우위를 점할 수 있게 되었다. 하지만 주어진 응용 프로그램들은 그 특징에 따라서 지역성과 공유 단위가 서로 다르게 나타났고, 이러한 응용 프로그램의 공유 단위와 원격 캐쉬와의 라인 사이즈 불균형은 시스템 전체의 성능 저하의 주된 원인으로 대두되었다. 본 논문에서는 주어진 응용 프로그램의 실행 패턴을 분석하여 이에 맞게 실행 시간 중 가상적으로 캐쉬의 라인 사이즈를 변화시킬 수 있는 다중 그레인 원격 캐쉬 기법을 제안하였다.

먼저 각각의 응용 프로그램별로 최적의 원격 캐쉬 라인 사이즈를 Profile-Based Determination 방식으로 살펴보았다. 또한 이 자료를 바탕으로 NMD를 구하였고, 이는 적정 캐쉬 라인 사이즈 선정의 중요성을 다시 한번 부각시켜 주는 계기가 되었다. 끝으로 본 논문에서 제안한 Dynamic Determination 방식을 통해 각 응용 프로그램들의 메모리 참조 패턴을 분석하였다. 그 결과 Miss Rate과 수행 시간적 측면에서 본 논문에서 사용된 응용 프로그램인 FFT/BARNES의 경우는 최악의 경우를 피하면서, 최적의 경우에 가깝게 수행된다는 것을 보였으며, 그리고 LU/RADIX의 경우 역시 최악의 경우를 피하면서 최적의 경우와 상당히 유사한 혹은 더 우수한 수행 결과를 보였다.

본 논문의 향후 과제로는 다음과 같은 것들을 들 수 있다. 원격 캐쉬의 최소/최대 라인 사이즈를 본 논문에서는 32Byte/128Byte로 정하여 이를 설계하고 성능 평가를 수행하였는데, 이를 보다 더 확장한 형태의 모의실험이 필요하다. 그리고 모의실험을 수행 할 때에 보다 더 다양한 응용 프로그램에 대한 고려와 더불어 각 응용 프로그램의 문제 크기(Problem Size) 또한 모의실험 인자로 주어 그 결과를 분석 할 필요가 있을 것이다.

참 고 문 헌

[1] J. L. Hennessy and D.A Patterson, "Computer Architecture : A Quantitative Approach," Second Edition, Morgan Kaufmann Publishers, Inc, 1996.

[2] Kai Hwang and Zhiwei Xu, "Scalable parallel Computing : Technology, Architecture, Programming," McGraw-Hill, 1998.

[3] Daniel Lenoski, Anoop Gupta et. "The Stanford Dash Multiprocessor," IEEE Computer, March 1992.

[4] Zhang, Z. and J. Torrelas, "Reducing Remote Conflict Misses : NUMA with Remote Cacae versus COMA," In Proc. of the 3rd IEEE Symp. on High performance Computer Architecture (HPCA-3), pp. 272-281, Feb. 1997.

[5] L. Iftode, J. P. Singh, K. Li, "Understanding Application Performance on Shared Virtual Memory Systems," Proceedings of the 23rd annual International symposium on Computer architecture pp. 122-133, 1996.

[6] C. Dubnicki, Thomas J. LeBlanc, "Adjustable Block Size Coherent Cache," International Symposium on Computer Architecture pp. 170-180, 1992.

[7] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, X. Ji, "Adapting Cache Line Size to Application Behavior," International Conference on Supercomputing, June 1999.

[8] 김형호, "지점간 링크를 이용한 스누핑 버스의 설계 및 성능 분석", 서울대학교 석사학위 논문, 1996.

[9] D.J. Lilja. "Cache coherence in large-scale shared-memory multiprocessors : Issues and comparisons," ACM Computing Surveys, 25(3):303-338, Sept. 1993.

[10] Per Stenstrom, *Truman Joe, and Anoop Gupta*, "Comparative Performanc Evaluation of Cache-Coherent NUMA and COMA Architectures," In *the 19th Int'l Symp. on Computer Architecture*, pages 80-91, 1992.

[11] S. J. Eggers and R. H. Katz, "The Effect of *Sharing on the Cache and Bus Performance* of Parallel Programs," In Proc. 3rd ASPLOS, 1989.

[12] S. Dwarkadas 외 6인, "Comparative Evaluation of *Fine- and Coarse-Grain Approaches* for Software Distributed Shared Memory," HPCA, 1998.

[13] D. R. Cheriton, A.Gupta, P. D. Boyle, and H. A. Goosen, "The VMP Multiprocessor : Initial Experience, Refinements, and Performance Evaluation," 15th ISCA, 1988.

[14] Tom Shanley, "Pentium Pro Processor System Architecture," Mind Share, Inc., 1997.

[15] IEEE Computer Society, "IEEE Standard for Scalable Coherent Interface(SCI)," Institute of Electrical and Electronics Engineers, August 1993.

[16] K. Inoue, K. Kai, K. Murakami, "High Bandwidth, Variable Line-Size Cache Architecture for Merged DRAM/Logic LSIs," IEICE TRANS. ELECTRON, 1998.

[17] JACK E. Veenstra, Robert 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), pp. 201-207, 1994.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and Anoop Gupta, "Methodological Considerations and Characterization of the Splash-2 Parallel Application Suite," In Proceedings of the 22th International Symposium on Computer Architecture, pp. 24-36, May 1995.

[19] David E. Culler, Jaswinder P. Singh, A. Gupta, "Parallel Computer Architecture : A Hardware / Software Approach," Morgan Kaufmann Publi-

shers, Inc, 1999.

곽 종 욱

1998년 2월 경북대학교 컴퓨터공학과 공학사. 2001년 8월 서울대학교 대학원 컴퓨터공학과 석사. 2002년 3월~현재 서울대학교 대학원 전기컴퓨터공학부 박사과정. 관심분야는 컴퓨터 구조, 병렬 처리 시스템

장 성 태

1986년 2월 서울대학교 전자계산기공학과 공학사. 1988년 2월 서울대학교 대학원 컴퓨터공학과 석사. 1994년 2월 서울대학교 대학원 컴퓨터공학과 박사. 1994년 3월~현재 수원대학교 정보공학대학 컴퓨터학과 부교수. 관심분야는 다중 프로세서 시스템, 병렬 처리, 캐쉬 구조, 메모리 모델

전 주 식

정보과학회논문지 : 시스템 및 이론 제 30 권 제 4 호 참조