다중 프로세서 시스템을 위한 세그먼트 디렉토리 방식의 설계 및 성능 분석

(Design and Performance Analysis of Segment Directory Method for Multiprocessor Systems)

최 종 혁 † 이 창 규 †† 박 규 호 †††

(Jong Hyuk Choi) (Chang Kyu Lee) (Kyu Ho Park)

요 약 본 논문에서 우리는 전체 벡터 디렉토리와 포인터 디렉토리의 중간 형태를 가지는 새로운 디렉토리인 세그먼트 디렉토리를 제안한다. 이는 대부분의 포인터 기반 디렉토리 방법들에서 디렉토리 저장 효율을 높이기 위하여 사용될 수 있다. 포인터가 단지 하나의 프로세서만을 가리키는 데 비하여, 세그먼트 디렉토리 요소는 포인터와 거의 같은 수의 비트들을 가지고 여러 개의 프로세서들을 동시에 가리킬 수 있다. 본 논문에서는, 세그먼트 디렉토리를 기존의 네 가지 한정 디렉토리 방법들에 적용하고, 이렇게 얻은 성능 개선을 측정, 분석하였다. 세그먼트 디렉토리는 한정 디렉토리 방법들의 성능을 저하시키는 요인인 디렉토리 넘침을 71% 까지 제거시킴으로써, 이 네 가지 방법들 상에서 수행된 모든 벤치마크 프로그램들에 대해 대역폭 요구량과 디렉토리 제어기 점유도, 메모리 접근 지연을 감소시켜서 프로그램의 수행을 가속시켰다. 게다가, 세그먼트 디렉토리는 추가적인 하드웨어 부담이나 프로토콜 복잡도 없이 간단하게 구현될 수 있다.

Abstract In this paper, we propose a new directory element called the segment directory, which is a hybrid of the full map vector and the pointer. It can be used in place of the pointer in most pointer-based directory schemes to improve directory storage efficiency: a segment directory element can point to several processors with almost the same number of bits as the pointer which can point to only one. In this paper, the segment directory is applied to four of the limited directory schemes, and performance improvement is then evaluated. The segment directory eliminates directory overflows by up to 71 % and reduces the traffic amount, the occupancy of directory controllers, and the latency of memory accesses. Therefore it can reduce the overall execution time of all the workloads run on the four schemes. Moreover, the segment directory does not introduce additional hardware overhead and protocol complexity.

1. 서 론

대부분의 공유 메모리 다중 프로세서 (shared memory multiprocessors) 들은 성능을 향상시키기 위하여 단위 프로세서들에 연결되어 있는 지역 캐쉬 (local cache) 들이 원격 공유 데이타를 캐쉬 (caching)하도록 허용한다. 그런데, 단위 프로세서들이 공유 데이타 개체의 많은 캐쉬 복사본들에 대한 쓰기 동작을 동시에 자유롭게 수행할 수 있다면, 이 복사본들의 일관성이 유지될 수 없게 된다.

일반적으로, 소규모의 다중 프로세서는 공동의 방송 매체, 즉 버스 (bus) 를 통하여 캐쉬 일관성 (cache coherence) 을 유지한다. 캐쉬 제어기가 버스를 감시하여 (snoop), 다른 프로세서들이 캐쉬 내에 복사본이 존재하는 메모리 블록 (memory block) 들을 접근하는지 관찰한다. 이러한 접근이 발생한다면, 사용되는 프로토콜에 따라 캐쉬 내의 복사본이 무효화되거나 갱신된다 [1].

† 정 회 원 : LG종합기술원 시스템 IC센터 연구원
jhchoi@acm.org
†† 비 회 원 : 한국과학기술원 전자전산학과
clee75@intizen.com
††† 종신회원 : 한국과학기술원 전자전산학과 교수
kpark@core.kaist.ac.kr
논문접수 : 2000년 1월 24일
심사완료 : 2000년 9월 28일

대규모 다중 프로세서들은 확장성을 향상시키기 위하여 버스를 사용하지 않고 점대점 상호 연결 망을 사용한다. 공유 메모리는 분산되어 있고, 각 노드 컴퓨터는 공유 메모리의 한 구역을 가지고 있다. 이러한 종류의 시스템을 분산 공유 메모리 다중 프로세서 (DSM : Distributed Shared Memory Multiprocessor) 또는 확장성이 우수한 공유 메모리 다중 프로세서 (SSMP : Scalable Shared Memory Multiprocessor) 라고 부른다. 이러한 시스템에서 방송이란 매우 값비싼 수행과정이다. 만약 일관성 유지를 위한 메시지들이 매번 방송된다면, 큰 시스템 대역폭을 가지는 점대점 상호 연결 망을 사용하는 것이 아무런 의미가 없어질 것이며, 시스템의 확장성도 크게 떨어진다.

DSM에서 캐쉬 일관성을 유지하는 방법들 중 가장 일반적인 것이 디렉토리 캐쉬 일관성 유지 방법 (directory-based cache coherence scheme) 이다 [2,3]. 디렉토리 방법은 점대점 통신을 사용하여 캐쉬 일관성을 유지한다. 디렉토리에는 어떤 프로세서들이 메모리 블록의 복사본들을 가지고 있는가에 대한 정보가 저장된다. 이 정보를 이용하여, 메모리 블록을 캐싱하고 있는 프로세서들에게만 선택적으로 무효화 또는 갱신 정보를 전달한다. 이러한 방법으로, 상호 연결 망의 포화를 피할 수 있고 시스템의 확장성을 증가시킬 수 있다.

전체 디렉토리 방법 (full-map directory scheme)[4]은 모든 메모리 블록들에 대하여 시스템 내의 모든 N 프로세서들의 완전한 캐싱 정보를 정확하게 유지한다. 메모리 블록 당 N-비트의 전체 벡터 (full-map vector)를 두어, 전체 벡터의 각 비트가 해당 프로세서의 블록 캐싱 정보를 나타내도록 한다. 이 방법은 가장 작은 일관성 유지 통신량을 발생시키지만, 대규모 다중 프로세서에서 너무 많은 양의 디렉토리 메모리를 필요로 한다.

이러한 메모리 추가 부담을 줄이기 위하여, 포인터 기반 방법 (pointer directory scheme) 들은 하나의 캐쉬 복사본 위치를 가리키는 식별자인 포인터를 사용한다. 메모리에 전체 캐싱 정보를 항상 정확하게 기록하지는 않는다. 한정 디렉토리 방법 (limited directory scheme) 들 [5,6,7] 에서는 포인터가 메모리에만 존재한다. 메모리 블록에 있는 포인터가 모자라게 되면, 각 방법들마다 고유한 방법을 통하여 이를 해결한다. 체인 디렉토리 방법 (chain directory scheme) [8] 에서는, 캐쉬 블록들에도 포인터들이 존재한다. 이 포인터들을 사용하여, 캐쉬 복사본들을 분산된 연결 리스트로서 관리한다. 포인터 기반 방법들의 성능은 일부 응용 프로그램들에서 매우 저하되는 것이 관찰된다.

본 논문에서 우리는 보다 더 효과적인 디렉토리 비트들의 구성을 제시한다. 만약, 제시된 구성을 사용하여, 포인터와 거의 같은 수의 비트들을 가지고, 포인터가 가리키는 것 보다 더 많은 프로세서들을 가리킬 수 있다면, 포인터를 기반으로 하는 디렉토리 방법들의 성능을 향상시킬 수 있게 된다. 제시된 새로운 디렉토리는 다음과 같은 관찰을 바탕으로 하고 있다 - 포인터는 전체 벡터보다 비트들을 덜 효율적으로 사용하지만, 반면에 포인터를 기반으로 하는 디렉토리 방법들은 전체 디렉토리 방법이 사용하는 것 보다 작은 양의 메모리를 사용한다.

본 논문은 세그먼트 디렉토리 (segment directory) [9,10] 라는 새로운 형태의 디렉토리를 설명한다. 세그먼트 디렉토리는 전체 벡터와 포인터의 혼합된 형태를 가진다. 실제로, 전체 벡터와 포인터는 세그먼트 디렉토리의 특별한 경계 경우들이다. 하나의 세그먼트 디렉토리 요소는 전체 벡터의 한 세그먼트인 세그먼트 벡터와, 전체 벡터 내에서 그 세그먼트의 위치를 결정하는 세그먼트 포인터로 구성된다. 세그먼트 디렉토리 요소는 포인터와 거의 같은 수의 비트들을 가지고 더 많은 프로세서들을 가리킬 수 있다. 게다가, 세그먼트 디렉토리는 포인터와 동일한 방법으로 사용 가능하여, 대부분의 포인터 기반 디렉토리 방법들에서 포인터를 대치할 수 있다. 본 논문에서는 세그먼트 디렉토리를 4개의 기존 한정 디렉토리 방법에 적용하여 디렉토리 넘침 (directory overflow) 을 줄이고, 결과적으로 프로그램 수행시간을 48 %까지 줄이게 된다. 디렉토리의 넘침은 한정 디렉토리 방법에 따라서 통신량을 증가시키거나 운영체제를 호출하는 등의 방법으로 처리하게 되는데, 이러한 방법들은 매우 느린 방법이므로 전체 시스템의 성능을 저하시킨다. 따라서 디렉토리 넘침을 줄이게 되면 전체 시스템의 성능이 향상되게 된다. 디렉토리에 관한 기존의 연구와는 달리 디렉토리 넘침을 줄이는 방법은 [9,10]에서 처음으로 시도되었다.

본 논문은 다음과 같이 구성되어 있다. 2 절에서는 기존의 디렉토리 요소들인 전체 벡터와 포인터의 장단점을 분석하고, 다중 프로세서 시스템의 성능평가에 대한 연구 배경을 설명한다. 3 절에서는 세그먼트 디렉토리를 소개한다. 세그먼트 디렉토리의 구조를 설명하고, 세그먼트 디렉토리가 가지는 성질들을 분석하며, 세그먼트 디렉토리가 높은 저장 효율을 가진다는 것을 입증한다. 또한 세그먼트 디렉토리가 효율적으로 구현될 수 있음을 설명하고, 기존의 한정 디렉토리 방법들에 쉽게 적용될 수 있음을 밝힌다. 4 절에서는 세그먼트 디렉토리

의 성능을 측정하고 분석한다. 본 연구에 사용된 시뮬레이션 (simulation) 방법과 환경을 설명하고 워크로드 (workload) 로 사용된 벤치마크 (benchmark) 들을 소개한 후, 세그먼트 디렉토리를 사용하여 한정 디렉토리 방법들의 성능을 얼마나 향상시킬 수 있는지를 정량적으로 설명한다. 얻어진 결과에 따라, 네 개의 한정 디렉토리 방법들의 성능을 자세히 분석하고, 서로 간의 차이점과 유사점에 대해서도 자세히 설명한다. 또한 세그먼트 디렉토리에 의한 성능 향상 요인들을 정성적으로 분석한다. 마지막으로, 5 절에서 결론을 맺는다.

2. 연구 배경

2.1 전체 벡터 대(對) 포인터

전체 벡터와 포인터는 기존 디렉토리 방법들에서 디렉토리를 구성하는 기본 요소들이다. 하나의 전체 벡터는 하나의 메모리 블록에 대한 N 개의 프로세서들의 캐슁 상태를 나타내는 N 비트들로 구성되어 있다. 전체 벡터의 i 번째 비트가 설정되어 있으면, 프로세서 i의 지역 캐쉬가 메모리 블록의 복사본을 가지고 있다는 것을 의미한다. 전체 벡터는 모든 메모리 블록들에 대한 정확한 캐슁 정보를 유지하는 전체 디렉토리 방법 [4] 에서 사용된다.

전체 디렉토리 방법의 메모리 추가 부담은 중간 규모의 다중 프로세서에서조차 너무 크다. 예를 들어, 128 개의 프로세서로 구성된 다중 프로세서 시스템에서는, 각각의 메모리 블록마다 16 바이트의 디렉토리 메모리가 요구된다. 메모리 블록의 길이가 32 바이트라고 하면, 디렉토리 메모리의 크기가 주 메모리 크기의 절반이 된다. 이것은 매우 과도한 메모리 부담이며, 게다가 전체 벡터의 크기는 프로세서 수 N 에 비례하여 증가하므로, 프로세서 수가 많아질수록 메모리의 부담은 계속 증가한다.

많은 병렬 응용 프로그램들에서 대부분의 메모리 블록들은 한 순간에 작은 수의 프로세서들에 의해서만 캐슁된다는 사실이 이전의 연구들 [5,6] 에서 밝혀졌다. 이 결과를 이용하여, 한정 디렉토리 방법들은 한정된 소수의 포인터들만을 각 메모리 블록마다 두었다. 그리고, 블록을 캐슁하는 프로세서의 개수가 사용 가능한 포인터들의 개수를 넘을 때 (디렉토리 넘침; directory overflow) 를 위한 특정 해결 방법을 제공하였다. 포인터의 크기는 N에 로그 급수로 증가하므로, 대규모 다중 프로세서에서도 디렉토리 메모리 추가 부담이 작게 유지될 수 있다.

하나의 포인터로 하나의 프로세서를 가리키는 데에는 $1+\log_2 N$ 비트[1]가 필요하다. 반면에, 전체 벡터로는 한 비트만 가지고도 충분히 하나의 프로세서를 가리킬 수 있다. 따라서, 전체 벡터가 포인터보다 비트들을 더 효율적으로 사용하고 있다고 볼 수 있다. 하지만, 어느 한 순간에서, 전체 벡터의 많은 비트들이 실제로는 사용되지 않는다. 전체 벡터가 포인터보다 더 큰 정적 저장 효율[2]을 가지고 있다고 할지라도, 포인터는 더 큰 동적 저장 효율[3]을 가지고 있는 것이다. 포인터의 동적 저장 효율을 이렇게 높게 만드는 주 요인은 포인터가 작은 단위로서, 필요한 만큼 임의의 개수가 사용될 수 있다는 점이다. 반면에, 전체 벡터는 전체로서만 사용될 수밖에 없는 나뉘어질 수 없는 큰 단위이다.

그러나, 한정 디렉토리 방법은 디렉토리 메모리 추가 부담이 작은 대신 디렉토리 넘침이라는 또 다른 문제를 야기한다. 많은 공유 읽기 요구가 메모리 블록에 도착하면, 많은 디렉토리 넘침이 일어난다. 이러한 디렉토리 넘침들과, 디렉토리 넘침이 일어났던 메모리 블록으로의 쓰기 요구들은 메모리 접근 지연과 통신 요구량, 디렉토리 제어기 점유도를 매우 크게 증가시킬 수 있고, 결과적으로 시스템의 성능을 저하시킬 수 있다.

2.2 다중 프로세서 시스템의 시뮬레이션 방법

다중 프로세서 메모리 계층 구조의 성능을 측정하는 데에는 주로 실제 프로그램 수행을 입력으로 하는 시뮬레이션이 사용된다. 캐쉬의 hit ratio가 성능에 미치는 영향이 매우 크기 때문에 해석적인 모델을 사용하거나, 시뮬레이션 모델의 입력을 통계적으로 생성하여 사용하면 성능 분석이 정확하지 않게 된다. 따라서, 실제 응용 프로그램의 실행 중 발생하는 메모리 참조를 입력으로 하여 시뮬레이션 함으로써 정확한 결과를 얻는다. 이러한 방법에는 트레이스 구동 시뮬레이션(trace-driven simulation), 실행 구동 시뮬레이션 (execution-driven simulation), 그리고 프로그램/인스트럭션 구동 시뮬레이션(program/instruction-driven simulation) 이 있다 [14]. 트레이스 구동 시뮬레이션은 메모리 참조 트레이스를 미리 얻은 후, 오프 라인으로 트레이스를 입력으로

1) 추가적인 한 비트는 포인터의 유효(valid) 비트이다. 이는 값을 갖지 않은 포인터에 하나의 값을 예약해 줌으로써 없앨 수 있다. 하지만, 최대 2의 지수 승 개의 프로세서까지 지원하려면 유효 비트는 필요하다. 한 메모리 블록에 대한 유효 비트들은 사용 중인 포인터의 개수로 변환될 수도 있다 [12].

2) 디렉토리 요소의 각 원소들이 다 사용된다고 가정하였을 때 단위 메모리 (비트) 로 가리킬 수 있는 캐쉬 복사본 위치의 개수

3) 실제 프로그램 수행 중 단위 메모리로 가리키게 되는 캐쉬 복사본 위치의 개수

시뮬레이션을 행한다. 트레이스의 생성과 시뮬레이션이 결합되어 있지 않기 때문에 트레이스를 생성한 시스템(traced system) 과 시뮬레이션의 대상이 되는 시스템(target system) 이 다름으로써 생기는 편차가 존재한다. 프로그램 구동 시뮬레이션은 CPU를 포함한 컴퓨터의 모든 부분을 모델링하여 응용 프로그램을 모델 상에서 실행시킨다. 세 가지 시뮬레이션 방법 중 가장 정확하게 성능 측정을 할 수 있으나, 시뮬레이션 시간이 매우 오래 걸린다. 실행 구동 시뮬레이션은 응용 프로그램의 비 메모리 참조 명령들 (산술, 논리 연산 등) 은 컴퓨터 상에서 직접 실행시키고, 메모리 참조 명령들(load, store, 동기화 명령 등) 은 시뮬레이터 상에서 수행시킨다. 따라서 빠른 시간 내에 비교적 정확하게 다중 프로세서 계층 메모리 시스템의 성능을 측정할 수 있다. 본 연구에서는 실행 구동 시뮬레이션 방법을 선택하여 제안된 세그먼트 디렉토리의 성능을 분석하였다.

3. 세그먼트 디렉토리

3.1 세그먼트 디렉토리의 구조

세그먼트 디렉토리는 전체 벡터와 포인터의 혼합된 형태를 갖는다. 그림 1은 여러 가지 구성의 세그먼트 디렉토리 구조를 보여주고 있다. 세그먼트 디렉토리 요소는 세그먼트 벡터와 세그먼트 포인터라는 두 개의 부분으로 구성된다. 세그먼트 벡터는 전체 벡터 내의 K-비트[4] 세그먼트이다. 그리고, 세그먼트 포인터는 전체 벡터 내에서 K 비트 단위로 정렬된 세그먼트 벡터의 위치를 결정하는 $\log_2(N/K)$-비트 필드이다. SDEK로 표기되는 세그먼트 벡터는 연속적인 K 프로세서들로 구성된 한 세그먼트에 대한 캐쉬 정보를 정확하게 저장한다. 세그먼트 포인터는 N/K 개의 전체 세그먼트들로부터 특정한 하나의 세그먼트를 지칭하며, 세그먼트 벡터는 하나의 세그먼트에 대한 전체 벡터라 할 수 있다. SDEK에 필요한 비트 수 (NoB : Number of Bits) 는 다음 식으로 표현된다.

$$NoB(SDE_K) = K + \log_2(N/K). \quad (1)$$

그림 1에 도시되어 있는 것과 같이, 32 프로세서로 구성된 시스템을 예로 들어보자. SDE4는 4-비트의 세그먼트 벡터와 3-비트($= \log_2(32/4)$)의 세그먼트 포인터로 구성된다. 따라서, SDE4는 일곱 비트들을 소비하면서 최대 네 개까지의 프로세서들을 가리킬 수 있다. 그림 1의 SDE4의 세그먼트 포인터 값은 1이다. 따라서, 그림 1의 SDE4는 1 번 세그먼트 내의 네 개의 프로세서들의 캐쉬 상태를 저장한다. 이들은 프로세서 4, 5, 6, 7이다. 이 SDE4의 세그먼트 벡터는 이들 중 프로세서 6, 7이 해당 메모리 블록을 캐쉬하고 있다는 것을 나타낸다.

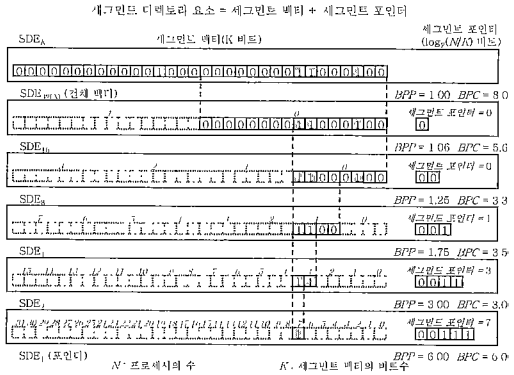


그림 1 세그먼트 디렉토리의 구조

보조 정리 1. SDEN은 전체 벡터이다.

증 명. K가 N일 때, 세그먼트 벡터는 전체 벡터가 된다. 전체 벡터 내에서 이 세그먼트 벡터가 자리할 수 있는 위치가 하나밖에 없기 때문에, 세그먼트 포인터는 불필요하다. □

보조 정리 2. SDE1은 포인터이다.

증 명. K가 1일 때, 세그먼트 벡터는 한 비트로만 구성되고, 세그먼트 포인터는 $\log_2 N$ 비트들로 구성된다. 세그먼트 포인터는 한 비트 세그먼트 벡터가 가질 수 있는 N 개의 위치들로부터 하나를 가리킨다. 세그먼트 포인터는 포인터가 되고, 한 비트 세그먼트 벡터는 포인터의 유효 비트이다. □

정 리 1. 세그먼트 디렉토리는 기존의 전체 벡터와 포인터를 포함하는 통합 디렉토리이다.

증 명. 보조 정리 1과 보조 정리 2로부터 세그먼트 디렉토리의 디렉토리 구성 체제가 기존의 전체 벡터와 포인터를 모두 포함하는 것을 알 수 있다. 즉, 전체 벡터와 포인터는 세그먼트 디렉토리의 특수한 경우들이다. 세그먼트 디렉토리는 이 두 가지 경우를 포함하고, 이들 사이에 새로운 구성의 디렉토리를 제시하는 통합된 디렉토리 구성 방법이다. □

보조 정리 3. SDE2의 정적 저장 효율은 포인터의 정적 저장 효율의 두 배이다.

증 명. SDE2가 가지는 비트 수는 포인터의 비트 수

4) $1 \le K \le N$ 이고, 어떤 정수 i 에 대해 $K = 2^i$ 이 만족됨을 가정한다.

와 동일하다 (식 (2) 참조).

$$NoB(SDE_2) = 2 + \log_2(N/2)$$
$$= 1 + \log_2 N = NoB(SDE_1) \quad (2)$$

포인터가 하나의 프로세서만을 가리킬 수 있는 반면, SDE2는 두 개의 프로세서들을 가리킬 수 있다. 따라서, SDE2는 포인터에 비해 두 배의 정적 저장 효율을 가진다. □

정 의 1. 디렉토리 요소의 프로세서 당 비트 수 (BPP: Bits Per Processor) 는 그 디렉토리 요소를 가지고 하나의 프로세서를 가리키는 데 필요한 비트 수이다. 실제로, BPP 값은 디렉토리의 정적 저장 효율의 역수가 된다. SDEK의 BPP는 다음과 같다.

$$BPP(SDE_K) = \frac{K + \log_2(N/K)}{K} \quad (3)$$

세그먼트 디렉토리 요소의 정적 저장 효율은 K 값에 따라 매우 급격히 증가한다. 정의 1 에서 주어진 세그먼트 디렉토리 요소의 BPP 값을 K 값에 따라 그려보면, BPP가 K 값에 따라 아주 급격히 감소하는 것을 알 수 있다. 그림 2는 하나의 디렉토리가 세그먼트 디렉토리 요소를 하나만 가지고 있다고 했을 경우의 BPP를 나타내었다. 그림 1의 SDE4의 경우를 보면 하나의 세그먼트 디렉토리 요소가 7개의 비트로 이루어져 있으며 최대 4개 노드를 나타낼 수 있으므로, 이 경우 BPP = 7/4 = 1.75 가 된다. 디렉토리의 정적 저장 효율은 BPP의 역수이므로, 디렉토리의 정적 저장 효율은 K 값에 따라 급격히 증가한다. 예를 들면, 보조 정리 3에서 보인 것과 같이 SDE2의 정적 저장 효율은 포인터에 비해 두 배이다. SDE4는 포인터 보다 한 비트만을 더 사용하면서 ($2+\log_2 N$ 비트) 네 개까지의 프로세서들을 가리킬 수 있다. SDE8은 $5+\log_2 N$ 비트를 사용하면서 여덟 개까지의 프로세서들을 가리킬 수 있다.

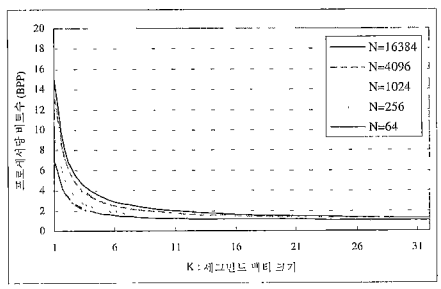


그림 2 세그먼트 디렉토리의 BPP (Bits Per Processor)

정 의 2. 캐쉬된 복사본당 비트 수 (BPC : Bits Per Cached Copy) 는 디렉토리 블록의 전체 비트 수를 현재 해당 블록을 캐쉬하고 있는 노드의 수로 나누어준 값이다. □

디렉토리의 정적 저장 효율을 높이는 데에 꼭 큰 K 값을 갖는 세그먼트 디렉토리를 써야 할 필요는 없고, 작은 K 값의 세그먼트 디렉토리를 사용해도 충분하다. 또한, 작은 K 값을 가지는 SDE는 포인터가 그러하듯이 작은 단위이고, 임의 개수의 SDE를 사용할 수 있으므로, 이렇게 작은 K 값을 가지는 세그먼트 디렉토리의 동적 저장 효율 또한 크다. 동적 저장 효율은 BPC의 역수로서, 세그먼트 디렉토리의 동적 저장 효율은 전체 디렉토리 방법이나 포인터 방법보다 크다는 것을 성능 분석 (4 절) 에서 보일 것이다.

3.2 세그먼트 디렉토리의 해석

SDE와 프로세서 식별자 (PID) 간의 변환 방법은 포인터와 전체 벡터에 필요한 변환 방법과는 다르다. 그림 3은 PID와 이들 세 가지 디렉토리 요소간의 변환 방법을 설명한다. 포인터는 PID를 바로 저장하므로, 포인터 값을 읽어 내어 직접 복사하면 된다. 전체 벡터에서는 1로 설정된 비트의 위치가 바로 PID이다. 이들 간의 변환에는 하나의 복호기 (decoder) 와 하나의 우선 순위 부호기 (priority encoder) 가 필요하다. 전체 벡터에서 1로 설정되어 있는 비트들을 우선 순위 부호화하여 순차적으로 PID들로 변환하고, 반대로 PID를 복호하여 전체 벡터의 해당 비트를 1로 설정한다.

세그먼트 디렉토리의 변환에는 포인터와 전체 벡터의 변환 방법이 모두 사용된다 : 세그먼트 포인터와 PID의 상위 $\log_2(N/K)$ 비트들은 직접 복사되고; 세그먼트 벡터는 PID의 하위 $\log_2 K$ 비트들로 우선 순위 부호기를 통해 변환되고, PID의 하위 비트들은 세그먼트 벡터로 복호기를 통하여 변환된다. 세그먼트 디렉토리에 사용되는 복호기와 우선 순위 부호기는 전체 벡터에 쓰이는 것 보다 훨씬 크기가 작고 빠를 수 있다. 이는 세그먼트 벡터의 크기가 전체 벡터의 크기보다 훨씬 작기 때문이다. 게다가, 전체 벡터의 크기는 프로세서의 수와 함께 선형적으로 증가하는 데 비해, 세그먼트 벡터의 크기는 고정시킬 수 있다. 결과적으로, 세그먼트 디렉토리의 해석에 필요한 하드웨어는 매우 간단하다. 특히 대규모 다중 프로세서에서는, 전체 벡터의 해석에 필요한 하드웨어보다 더 효율적이라고 할 수 있다.

3.3 한정 디렉토리 방법들에서 세그먼트 디렉토리의 사용

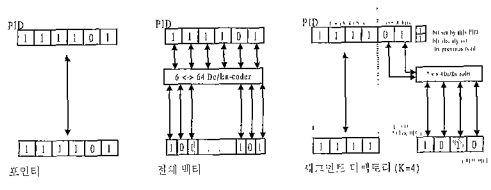


그림 3 포인터, 전체 벡터, 세그먼트 디렉토리의 해석

세그먼트 디렉토리는 대부분의 포인터 기반 디렉토리 방법들에서 포인터를 대신하여 사용될 수 있다. 본 논문에서, 우리는 세그먼트 디렉토리를 우선 네 개의 기존 한정 디렉토리 방법들에 적용하였다. 이들은 방송/비방송 한정 포인터 방법 (LimB/LimNB) [5], 성긴 벡터 디렉토리 방법 (CV) [6], 그리고 LimitLESS 디렉토리 방법 (LimLESS) [6] 이다.

이 디렉토리 방법들은 디렉토리 넘침이 일어나기 전까지는 동일하게 동작한다. 이들은 어느 메모리 블록을 캐싱하는 프로세서들의 PID들을 그 메모리 블록과 연결된 포인터들로 저장한다. 일단 디렉토리 넘침이 발생하면, 이들은 서로 다른 동작을 취한다. LimB는 디렉토리 넘침 발생 시 메모리 블록에 있는 방송 비트를 설정하고, 이후 읽기를 요구한 프로세서들은 포인터에 기록하지 않는다. 이러한 블록으로 이후 쓰기 요구가 도착하면, 무효화는 시스템 내의 모든 프로세서들에게로 방송될 것이다. CV는 디렉토리 넘침 시 디렉토리의 사용 방법을 복수의 포인터들로부터 성긴 벡터로 바꾸어 버린다. 성긴 벡터는 전체 벡터와 같은 비트 벡터인데, 각 비트가 하나의 프로세서를 가리키는 것이 아니라, 몇 개의 프로세서들을 집단적으로 (즉, 몇 개의 프로세서들로 구성된 영역을) 가리킨다. 디렉토리 넘침 이후, 이러한 메모리 블록에 쓰기가 발생한다면, 무효화는 성긴 벡터가 설정된 영역들에 속하는 모든 프로세서들로 부분 방송 (multicast) 된다. LimNB는 포인터들 중 하나를 사용 가능하게 만듦으로써 디렉토리 넘침을 일어나지 않도록 한다. 이는 선택된 포인터가 가리키고 있던 캐쉬 복사본을 무효화시킴으로써 이루어진다. LimLESS에서는, 디렉토리 넘침 시 프로세서에게 인터럽트를 요청한다. 해당 인터럽트 처리 루틴이 하드웨어 포인터들의 내용을 주 메모리 상의 디렉토리 데이타 구조로 복사하여, 모든 하드웨어 포인터들을 다시 사용 가능하게 만든다. 이러한 블록에 이후 도착한 쓰기 요구 역시 인터럽트를 발생시켜서, 인터럽트 처리 루틴이 주 메모리에 저장된 디렉토리를 사용하여 무효화를 전송할 수 있도록 한다.

이러한 모든 한정 디렉토리 방법들은 디렉토리 넘침이 일어난 후 수행되는 동작에 있어서 구별되며, 이러한 동작의 차이로 성능의 차이가 생긴다. 반면에, 한정 디렉토리 방법들에 세그먼트 디렉토리를 적용하는 것은, 디렉토리 넘침 자체를 줄이기 위한 것이다. 따라서, 기존의 한정 디렉토리 방법들과는 완전히 다른 차원으로의 접근 방법이라 할 수 있다. 기존의 한정 디렉토리 방법들에 아주 조금의 수정만 가하여, 포인터 대용으로 세그먼트 디렉토리를 사용할 수 있다. 아래에서 한정 디렉토리 방법들을 적용하는데 필요한 두 가지 수정 사항을 설명하였다.

1. exclusive / read-exclusive 요구 [13] 에 반응하여 무효화를 전송할 때, 포인터는 단지 하나의 무효화만을 생성하지만, SDEK는 K 개까지의 무효화들을 발생시킬 수 있다. 하나의 SDE는 전술한 디렉토리 해석 방법을 통해 복수의 PID들로 변환된다. 무효화를 전송하는 것은 그 자체가 순차적인 동작이고, 앞 절에서 살펴본 것과 같이 세그먼트 디렉토리의 해석은 전체 벡터의 해석보다도 효율적이므로, exclusive / read-exclusive 요구의 접근 지연이 증가하지 않는다.

2. read-nonexclusive 요구 [13] 에 반응하여 새로운 PID를 세그먼트 디렉토리에 저장하기 위해, 해당 메모리 블록에 연관된 디렉토리를 검색하여 동일한 세그먼트 포인터 값을 가진 세그먼트 디렉토리 요소가 이미 존재하는지를 검사해야 한다. 만약 그러한 디렉토리 요소가 존재한다면, 새로운 디렉토리 요소를 사용하지 않고, 기존 해당 디렉토리 요소의 세그먼트 벡터 중 (PID mod K) 번째 비트를 1로 설정하기만 하면 된다. 그러한 디렉토리 요소가 존재하지 않는다면, 새로운 세그먼트 디렉토리 요소를 할당받아 PID를 저장해야 한다. 각 노드 당 하나의 메모리 블록에 연관된 디렉토리 요소의 개수만큼의 비교기 (comparator) 들을 사용함으로써 동일한 세그먼트 포인터를 갖는 디렉토리 요소를 한 번에 검색할 수 있다. 한정 디렉토리 방법들에서는 일반적으로 하나의 메모리 블록에 작은 수의 디렉토리 요소들이 연관되어 있기 때문에, 추가적인 하드웨어 부담은 미미하다. 게다가, 이러한 검색은 회신 (reply) 데이타 전송과 시간적으로 중복될 수 있다 (미리 읽기 (read-ahead) 최적화 [12]). 따라서, read-nonexclusive 요구의 접근 지연을 증가시키지 않고 디렉토리 요소들을 검색하는 것이 가능하다.

세그먼트 디렉토리를 한정 디렉토리 방법들에서 사용하는 데에는 이러한 두 가지 미미한 수정만이 필요할 뿐이다. 다음 절에서는 세그먼트 디렉토리가 한정 디렉

토리 방법들의 성능을 어떻게 얼마나 향상시키는가를 살펴본다.

4. 성능 측정 및 분석

본 절에서는 세그먼트 디렉토리를 한정 디렉토리 방법에 적용시켰을 경우의 성능을 시뮬레이션을 통해 측정하고, 그 결과를 분석한다. 우선 본 논문에서 사용된 시뮬레이션 접근 방법에 대해 간략히 설명한 후, 시뮬레이션의 입력으로 사용된 워크로드와 시뮬레이션 대상 시스템의 구조에 대해 살펴본다. 그 후, 각 한정 디렉토리 방법들에서 얼마나 성능 향상이 이루어졌는지를 디렉토리 넘침 횟수, 대역폭 요구량, 그리고 수행 시간을 척도로 살펴보고, 성능 향상 요인을 분석한다.

4.1 시뮬레이션 방법

세그먼트 디렉토리의 효율성을 보이기 위하여 Stanford 대학의 Tango Lite [15] 실행 구동 시뮬레이션 환경을 사용하였다. Tango Lite는 쓰레드를 이용하여 구현된 실행 구동 시뮬레이션 환경으로 수십 개의 프로세서들로 구성된 다중 프로세서 시스템의 시뮬레이션을 매우 빠른 시간 내에 수행할 수 있다.

시뮬레이션 대상 컴퓨터 (target system) 는 64 개의 노드들이 메쉬 웜홀 라우팅 네트워크로 연결된 DSM 시스템이다. LimNB 를 위하여, Simoni [13] 의 연구에서 상세하게 설명된 비방송 한정 디렉토리를 기반으로 한, MSI 프로토콜 [13] 을 사용하였다. LimB, CV, 그리고 LimLESS 들도 Simoni 의 MSI 프로토콜을 수정하여 구현하였다. 일관성 유지 미스들과 통신량에 집중하기 위해 무한 크기 캐쉬를 가정하였다[5]. 캐쉬와 메모리 블록의 길이는 32 바이트이다. 노드 컴퓨터는 프로세서 (= 수행되는 프로그램), 캐쉬 모듈, 메모리 및 디렉토리 모듈, 네트워크 인터페이스로 구성된다. 노드 컴퓨터는 [13] 에서 제시된 구성을 취하고, 상호 연결 망은 Agarwal의 네트워크 모델[16]에 기반을 두고 있다. 시뮬레이션 대상 시스템에서 가정된 메모리 일관성 모델 (memory consistency model) 은 약한 일관성 (weak ordering) [18] 이다. 시뮬레이션 대상 시스템의 대표적인 메모리 접근 지연을 표 1에 나타내었다.

표 1 대상 시스템의 read-nonexclusive 메모리 접근 지연 (단위: 프로세서 사이클).

접근 위치	접근 지연	접근 위치	접근 지연
캐쉬 hit	1	지역 메모리	39
원격 메모리 (SHARED)	107	원격 메모리 (MODIFIED)	205

5) replacement hint가 없다고 가정하였으므로 무한 크기 캐쉬가 성능을 왜곡하지 않는다.

표 2 시뮬레이션 된 4 가지 벤치마크 프로그램

응용 프로그램	설 명	데이타 집합
LU	밀 행렬 LU 팩터 화	256x256 행렬
Radix	기수(radix) 정렬	256K keys, radix 1K
Barnes	3 차원 상호 작용 계산	4K bodies
FMM	2 차원 상호 작용 계산	4K bodies

시뮬레이션의 작업 부하 (workload) 로는 SPLASH-2 [17] 벤치마크 집합 중 네 개의 프로그램을 사용하였다. 표 2에 사용된 프로그램, 프로그램에 대한 설명, 그리고 사용된 데이타 집합을 나타내었다. LU는 밀 (dense) 행렬을 하위 행렬(lower matrix)과 상위 행렬 (upper matrix)로 분리하는 LU-decomposition 프로그램이고, Radix는 정수의 기수(radix) 정렬 프로그램이다. Barnes는 3 차원에서 n 개 물체들 간의 상호 작용을 barnes-hut 계층적 N-객체 방법을 사용하여 계산하는 프로그램이다. FMM도 Barnes와 마찬가지로 n 개의 물체들간의 상호 작용을 계산하지만, 2 차원에서의 계산이고 빠른 다극 방법을 사용한다.

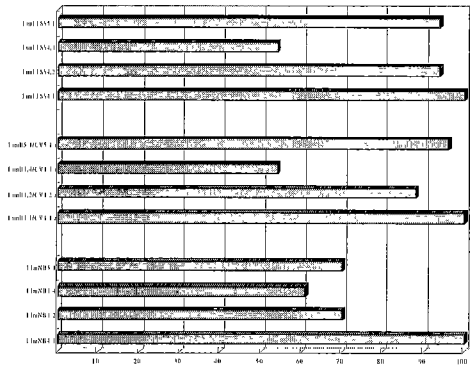
4.2 세그먼트 디렉토리를 적용한 한정 디렉토리 방법들의 성능

LimNB, LimB, LimLESS 는 각각 LimNBi,k, LimBi,k, 그리고 LimLESSi,k로 표기된다. 여기에서 i는 한 메모리 블록에 연관된 디렉토리 요소의 개수이고 k는 세그먼트 벡터의 크기이다. CV는 CVi,k,r로 표기되는데, r은 거친 벡터의 한 비트가 가리키는 영역의 크기이다. k = 1인 방법들은 포인터를 사용한 원래의 한정 디렉토리 방법들이고, k > 1인 방법들이 세그먼트 디렉토리를 적용한 방법들이다. 전체 디렉토리 방법은 FM으로 표기된다.

본 논문은 (4,1(,r)), (4,2(,r)), (4,4(,r)), 그리고 (5,1(,r)) 방법의 성능을 비교한다. (4,1(,r)) 방법은 네 개의 포인터들을 사용한다. 성능을 향상시키기 위해, (4,2(,r)) 방법은 네 개의 SDE2들을 사용하고 (4,4(,r)) 방법은 네 개의 SDE4를 사용한다. 반면에, (5,1(,r)) 방법은 성능을 향상시키기 위해 (4,1(,r)) 방법의 각 메모리 블록에 하나의 포인터를 더 추가한다.

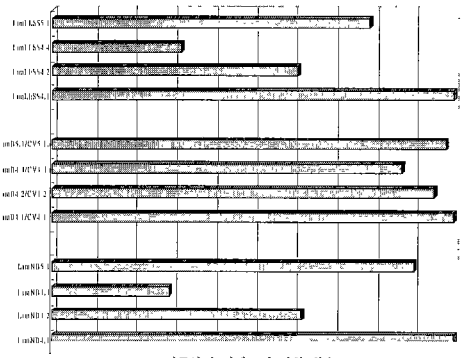
그림 4는 디렉토리 넘침 횟수의 감소를 보여준다. LimB와 CV는 동일한 디렉토리 넘침 특성을 가진다.

사실, LimB는 r = N 인 CV로 생각할 수 있다. (4,2(,r)) 방법들의 디렉토리 넘침은 (4,1(,r)) 방법들의



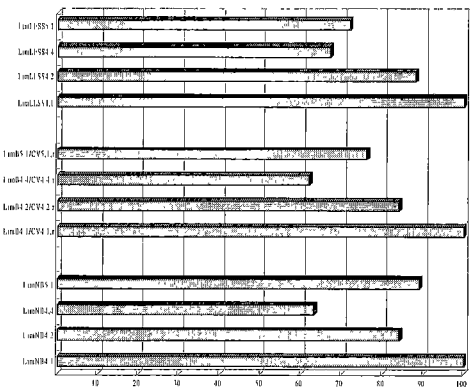
정규화된 디렉토리 넘침 횟수

(a) LU



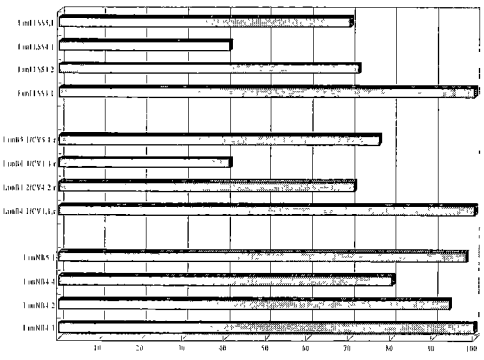
정규화된 디렉토리 넘침 횟수

(b) Radix



정규화된 디렉토리 넘침 횟수

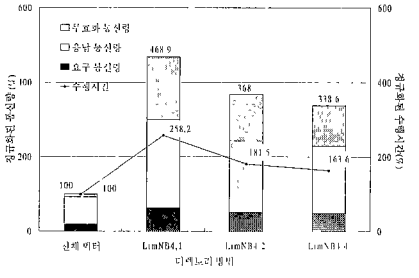
(c) Barnes



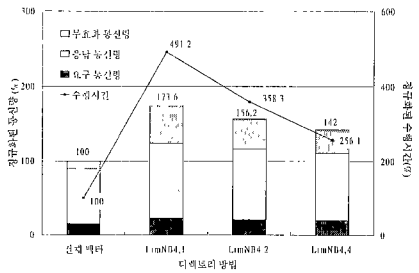
정규화된 디렉토리 넘침 횟수

(d) FMM

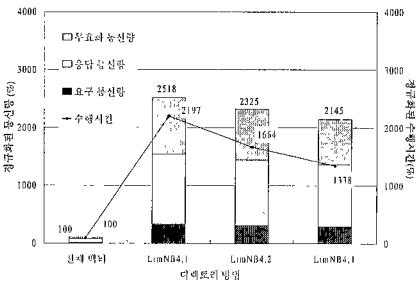
그림 4 디렉토리 넘침 횟수



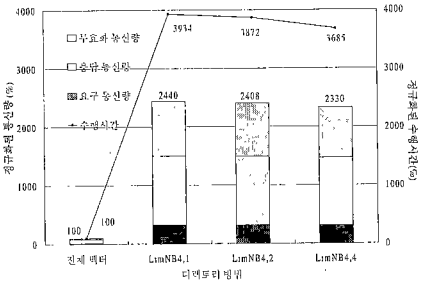
(a) LU



(b) Radix



(c) Barnes



(d) FMM

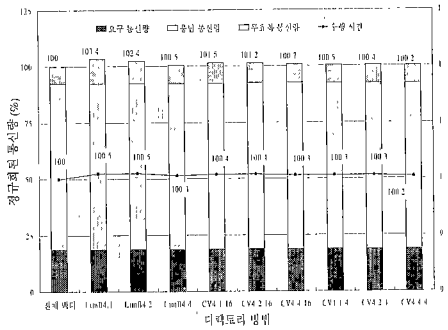
그림 5 LimNB를 적용한 시뮬레이션 결과

디렉토리 넘침에 비하여 39 % 까지 감소되었다. 즉 SDE2를 사용하여 (4,1(,r)) 방법들의 디렉토리 넘침을 39 % 까지 제거한 것이다. (4,1(,r)) 방법들과 (4,2(,r)) 방법들 모두 메모리 블록 당 28 비트의 디렉토리 메모리를 가지고, 모든 응용 프로그램에서 디렉토리 넘침이 감소하였으므로, 세그먼트 디렉토리를 사용하여 디렉토리의 동적 저장 효율을 증가시킨 것이다. (4,4(,r)) 방법들은 (4,1(,r)) 방법들에 비해 메모리 블록 당 4 비트의 메모리를 더 소비하면서, 디렉토리 넘침이 71 % 까지 감소되었다. 이에 비해, (5,1(,r)) 방법들은 (4,1(,r)) 방법들에 비해 블록 당 7 비트의 메모리를 더 소비하면서도, 기껏해야 30 % 의 디렉토리 넘침을 감소시켰을 뿐이다. 모든 경우에 있어서, (4,4(,r)) 방법들이 (5,1(,r)) 방법들에 비해 더 적은 디렉토리 넘침 횟수를 보여주고 있다. 따라서, 세그먼트 디렉토리를 사용하는 것이 단지 포인터 하나를 더하는 것보다 훨씬 효과적이다.

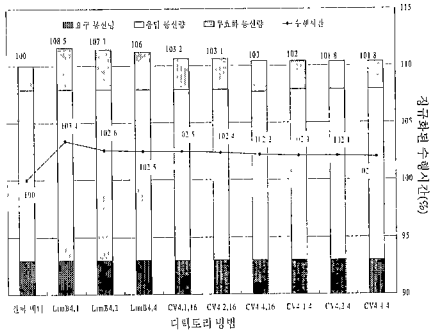
그림 5, 6, 및 7은 네 개의 한정 디렉토리 방법들에 세그먼트 디렉토리를 적용했을 경우의 성능 개선을 보여준다. 전체 통신량은 요구 (writeback을 포함한 프로세서 요구), 응답 (data / exclusive reply, copyback / flush request and reply), 그리고 무효화 (invalidation, invalidation acknowledgment, invalidations done) 통신량으로 나뉘어져 있다 [13]. 통신량은 바이트 수로 측정되었고, 수행 시간은 프로세서 사이클로 측정되었다.

그림 5는 LimNB 방법들에서 수행한 벤치마크 프로그램들의 통신 요구량과 수행 시간을 FM 방법에 대해 정규화시켜 나타내었다. LimNB의 성능은 매우 불안정하다는 것을 본 연구에서 정확한 실험을 통하여 입증하였다. 세그먼트 디렉토리가 LimNB의 통신 요구량과 수행 시간을 줄이기는 하지만, 그래도 LimNB는 좋지 않는 성능을 보여주고 있으며 실용적이지 못하다.

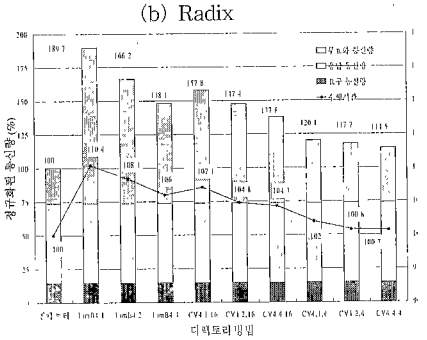
그림 6는 LimB와 CV 방법들에서 수행한 벤치마크



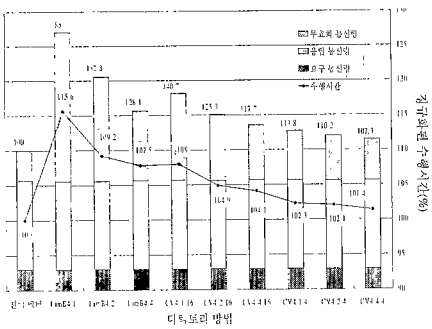
(a) LU



(b) Radix



(c) Barnes



(d) FMM

그림 6 세그먼트 디렉토리를 적용한 LimB / CV 성능

프로그램들의 정규화된 통신 요구량과 수행 시간을 보여주고 있다. 모든 방법들의 요구 통신량과 응답 통신량이 동일함을 볼 수 있다. LimB와 CV가 캐쉬 실패율을 증가시키지 않으므로, FM에 비해 무효화 통신량만이 증가된 것이다.

LU와 Radix의 무효화 분포를 살펴보면 96%의 무효화 유발 쓰기 요구가 단지 하나의 무효화만을 생성하는 것을 알 수 있다. 세그먼트 디렉토리가 LU와 Radix에 대해 각각 LimB4,1 / CV4,1,r의 디렉토리 넘침을 46 % 와 13 % 까지 제거했음에도 불구하고, 디렉토리 넘침이 발생했던 메모리 블록들에 대한 무효화 유발 쓰기가 매우 적다. 따라서, 세그먼트 디렉토리를 사용하여 얻어지는 성능 향상은 그리 크지 않다.

Barnes와 FMM은 이와 다른 특성을 보여준다. 많은 무효화들을 유발하는 다수의 무효화 유발 쓰기가 존재한다. 이런 종류의 응용 프로그램들에 대해서는, 한정 디렉토리 방법의 성능을 사용했을 경우 매우 큰 성능저하가 있을 수 있다.

FM에 대한 LimB4,1의 통신 요구량 증가는 85 % 에까지 이르고, 수행 시간의 증가는 15.6 % 에 까지 달한다. 세그먼트 디렉토리는 통신 요구량을 47 % 까지, 수행 시간을 8 % 까지 감소시킨다.

CV에 대해서는, 우선 64 프로세서 시스템을 염두에 두고 영역 크기 r을 4로 놓았다. CV4,1,4의 성능이 이미 FM의 성능에 근접하므로, 세그먼트 디렉토리를 사용하였을 경우의 성능 향상은 그리 크지 않다. 통신 요구량은 6 % 감소하였고 수행 시간은 2 % 정도 줄어들었다. 하지만, 512 프로세서 시스템에서라면, r = 4 인 거친 벡터는 128 비트, 즉 32 바이트 메모리 블록의 절반을

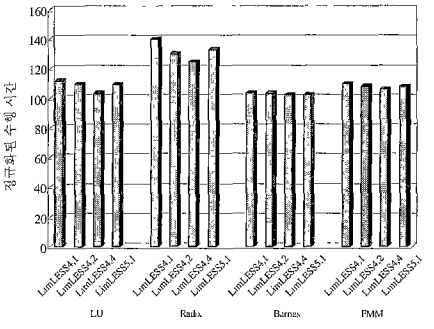


그림 7 세그먼트 디렉토리를 사용한 LimLESS의 성능(수행 시간은 각각의 응용 프로그램의 전체 디렉토리 방법의 수행 시간에 정규화되어 있다.)

소모할 것이다. 이러한 상황에서는, 메모리 추가 부담을 줄이기 위해, r을 증가시켜야 한다. r = 16 인 거친 벡터는 32 비트를 소비한다. 64 프로세서 시스템에서의 CV4,k,16의 성능도 그림 5에 함께 나타나 있다. 통신 요구량은 23 % 감소하였고, 수행 시간은 4 % 까지 감소하였다.

그림 7은 세그먼트 디렉토리를 적용한 LimLESS의 성능을 보여준다. LimLESS의 통신량은 FM의 통신량과 동일하기 때문에, 수행 시간만 나타내었다. LimLESS에서 수행 시간의 증가는 디렉토리 넘침으로 인한 프로세서 인터럽트와 인터럽트 처리 지연[6]에서부터 유래되므로, LimLESS의 수행 시간은 무효화 유발 쓰기의 회수에 보다는 디렉토리 넘침 횟수에 더 연관되어 있다. 따라서, LimB / CV 에서와는 달리, LU와 Radix의 수행 시간 증가가 작지 않다. 세그먼트 디렉토리를 사용하여 Radix를 제외한 모든 응용 프로그램들의 수행 시간을 106 % 이하로 떨어뜨렸다. 그림 4는 또한 단지 포인터 하나를 추가하는 것보다 세그먼트 디렉토리를 사용하는 것이 더 효과적이라는 사실을 확인시켜 준다. LimLESS4,4가 LimLESS5,1보다 더 적은 메모리를 사용하고 있음에도 불구하고, LimLESS4,4가 LimLESS5,1보다 항상 좋은 성능을 보여주고 있다.

결과적으로, 세그먼트 디렉토리는 한정 디렉토리 방법들을 전체 디렉토리 방법의 성능에 보다 더 근접하도록 하여, 한정 디렉토리 방법들의 경쟁력을 높인다. 즉, 훨씬 적은 양의 메모리를 사용하면서 전체 디렉토리 방법의 성능에 가까운 성능을 낼 수 있도록 한다.

4.3 세그먼트 디렉토리의 성능 향상 요인

앞에서 살펴보았듯이, 세그먼트 디렉토리는 기존의 한정 디렉토리 방법들에서 주된 성능 저하 요인인 디렉토리 넘침을 크게 감소시키고 있다. 네 개의 한정 디렉토리에서 수행된 네 개의 벤치마크 응용 프로그램에서 모두 디렉토리 넘침이 감소되었다. 세그먼트 디렉토리는 포인터를 사용한 기존 방법들에서 존재하던 디렉토리 넘침을 5 % 에서 71 % 까지 제거하였다. 반면에, 포인터를 하나 더 더하는 것은, 더 많은 메모리를 사용함에도 불구하고, 2 % 에서 30 % 까지의 디렉토리 넘침을 제거하였을 뿐이다. 모든 경우에 대해, 세그먼트 디렉토리를 적용하는 것이 포인터를 하나 더하는 것에 비해 효과적이었다.

이렇게 현저한 성능 향상은 세그먼트 디렉토리의 메

6) 시뮬레이션에서 인터럽트 처리 지연은 Alewife 멀티 프로세서 [12] 를 참조하여 결정하였다.

모리 저장 효율이 증가되었다는 것에 기인한다. 하지만, 이러한 저장 효율의 증가 외에, 특별한 형태의 메모리 참조 지역성이 성능 향상을 더욱 크게 하고 있다. 만약, 메모리 참조에 지역성이 전혀 없고, 메모리 참조가 모든 프로세서들로부터 균등하게 도착한다면, 위의 성능 분석 결과에서와 같은 큰 성능 향상을 얻을 수 없었을 것이다. 이러한 메모리 참조 지역성은 세그먼트 디렉토리가 연속적인 PID를 갖는 K 개의 프로세서들을 한 번에 가리킬 수 있다는 사실과, 연속적인 PID를 갖는 프로세서들이 시스템에서 서로 인접할 가능성이 아주 높다는 사실에 기인한다. 이러한 점에 따라, 이하의 두 가지 메모리 참조 지역성이 존재한다.

1. 프로그램 내에 존재하는 참조 지역성 : 효율적인 병렬 프로그램에서는 프로세서가 참조하는 데이타를 되도록 그 프로세서에 가깝게 위치시킨다. 따라서, 메모리 블록을 빈번하게 참조하는 프로세서들은 메모리 블록이 위치하는 홈 노드에 가까이 존재하는 프로세서들일 것이다. 홈 노드에 가까이 존재하는 프로세서들은 서로 간에도 역시 인접하고 있기 때문에, 연속적인 PID를 가질 확률이 높다. 따라서, 이렇게 인접하는 프로세서들로부터 메모리 참조 요구가 도착할 경우, 메모리 참조 요구가 모든 프로세서들에 균등하게 분포되었을 경우에 비해, 세그먼트 디렉토리의 동적 저장 효율이 더 높을 것이다.

2. 상호 연결 망의 특성에 의한 참조 지역성 : 메모리 블록에 대한 참조 요구가 전체 시스템에 균등하게 분포되어 있을지라도, 홈 노드에 가까이 존재하는 프로세서들로부터의 메모리 참조가 먼저 도착할 가능성이 높다. 만약, 모든 프로세서들로부터 동시에 메모리 참조 요구가 발생했다면, 이들이 홈 노드에 도착하는 순서는 홈 노드로부터 프로세서들까지의 거리와 연관이 있다. 홈 노드로부터 비슷한 거리에 있는 프로세서들이 연속적인 PID를 가질 확률 역시, 상관 관계가 없는 프로세서들이 연속적인 PID를 가질 확률보다 높다. 게다가, 병렬 프로그램에는 일반적으로 계산의 단계 (phase) 가 존재하고, 각 프로세서들은 서로 동기를 맞추어 계산을 진행하기 때문에, 주어진 기간에 서로 연관있는 데이타들을 참조할 가능성 역시 높다. 따라서, 프로그램에 메모리 참조 지역성이 없을 경우에도, 홈 노드에 도착하는 연속적인 메모리 참조들이 연속적인 PID를 가지는 프로세서들로부터 발생하였을 확률이 높다고 할 수 있다.

그림 8은 이러한 특성을 실험한 결과이다. 실험 환경은 4.2절에서의 실험 조건과 동일하다. 전체 시스템은 64 노드로 이루어져 있으며, 64 그룹은 전체 시스템을 64개의 세그먼트로 분할한 것이다(1 세그먼트 = 1 노드). 32 그룹은 전체 시스템을 32개의 세그먼트로 분할한 것이며(1 세그먼트 = 2 노드), 이 때 연속된 PID를 가지는 두 개의 노드가 하나의 세그먼트를 이루도록 구성하였다. 16 그룹은 전체 시스템을 16개의 세그먼트로 분할한 것이며(1 세그먼트 = 4 노드), 연속된 PID를 가지는 네 개의 노드가 하나의 세그먼트를 이루도록 구성하였다. 프로그램을 수행하는 동안, 하나의 메모리 블록을 몇 개의 세그먼트가 공유하는지를 측정하였다. 그림에서 알 수 있듯이, 세그먼트에 속한 노드의 수가 많아질수록 하나의 메모리 블록을 공유하는 세그먼트의 수는 줄어들게 된다. 이것은 인접한 노드들이 동일한 메모리 블록을 접근하는 경우가 빈번함을 나타내는 결과이다. 즉, 인접한 PID를 가지는 노드들이 동일한 메모리를 접근하는 확률이 높다는 것을 알 수 있다.

그림에서 64 그룹은 포인터를 사용한 경우와 같은 경우이다. 하나의 메모리를 공유하는 세그먼트의 평균수만큼의 포인터를 사용하는 시스템이라면, Radix의 경우 디렉토리의 크기는, 약 7개의 포인터가 요구되므로, 49 bits가 된다. (1 포인터 = 7 bits) 반면에 32 그룹의 경우는 K=2 인 세그먼트 디렉토리를 사용한 경우이며, Radix에서 동일한 조건을 만족하기 위해서는 약 4개의 SDE2가 필요하므로 디렉토리의 크기는 약 28 bits가 된다. (1 SDE2 = 7 bits) 이 결과를 이용하여 3절에서 정의한 BPC를 계산해 보면, Radix의 경우 평균적으로 하나의 메모리 블록을 약 7개의 노드가 공유하고 있으므로, 전체 디렉토리의 경우는 64/7 ≒ 9 가 되고, 포인터의 경우는 49/7 = 7 이 된다. 반면 K=2 인 세그먼트 디렉토리의 경우는 28/7 = 4 가 된다. 즉, 그림 8의 실험결과에서 세그먼트 디렉토리는 전체 디렉토리 방법이나 포인터 방법보다 동적 저장 효율이 좋다는 것을 알 수 있다.

만약, 컴파일러나 운영체제가 이러한 메모리 참조의 지역성을 이용하여 프로그램을 최적화하거나, 태스크를

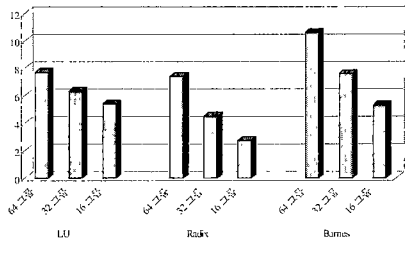


그림 8 64 노드 시스템에서 하나의 메모리 블록을 공유하는 세그먼트의 수(평균)

스케쥴링 및 매핑 한다면, 세그먼트 디렉토리의 저장 효율은 더욱 향상될 것이다. 대표적인 최적화 작업으로는, 전술한 바와 같이, 어떤 데이타를 빈번히 접근하는 프로세서에 가깝게 그 데이타를 위치시키는 것이다. 이러한 간단한 최적화만으로도 세그먼트 디렉토리의 성능은 더욱 향상될 것이다. 더구나, 세그먼트 디렉토리를 위한 최적화가 시스템의 성능에 좋지 않은 영향을 끼치는 것이 아니라, 오히려 시스템 성능을 향상시키는 일반적인 최적화 방법이기 때문에, 이런 최적화와 세그먼트 디렉토리의 사용은 상호 상승 작용을 일으킨다.

5. 결 론

본 논문은 세그먼트 디렉토리를 디렉토리 캐쉬 일관성 유지 방법에서 사용되는 포인터 디렉토리의 개선으로서 제안하였다. 세그먼트 디렉토리는 전체 벡터와 포인터를 양극단으로서 통합하고, 그 사이에 이들의 중간 형태를 갖는 새로운 디렉토리 구성 요소들을 제시한다. 세그먼트 디렉토리 요소는 포인터보다 많은 수의 프로세서들을 가리키면서, 동시에 포인터와 같이 작은 단위로서 사용될 수 있다. 세그먼트 디렉토리는 포인터를 대치하여 사용되어 포인터보다 디렉토리 비트들을 더 효율적으로 이용하도록 한다. 많은 디렉토리 넘침들이 제거되어, 한정 디렉토리 방법을 사용한 다중 프로세서들의 통신 요구량과 메모리 접근 지연, 그리고 디렉토리 제어기 점유도를 감소시킨다. 게다가, 세그먼트 디렉토리의 해석에는 추가적인 비용이나 복잡도가 필요하지 않고, 오히려 전체 벡터의 경우에서보다 간단하고 빠르게 구현할 수 있다.

우선, 세그먼트 디렉토리를 네 개의 기존 한정 디렉토리 방법들에 적용하였다. 세그먼트 디렉토리는 디렉토리 넘침을 감소시켜서 수행된 모든 벤치마크 프로그램에 대해서 모든 한정 디렉토리 방법들의 성능을 향상시켰다. 또한, 세그먼트 디렉토리를 사용하는 것이 포인터들을 더 더하는 것 보다 더 효과적이라는 사실도 입증하였다. 게다가, 세그먼트 디렉토리는 한정 디렉토리 방법들에서 아주 미미한 프로토콜 수정만으로 아주 잘 사용될 수 있다.

시뮬레이션시, 세그먼트 디렉토리에 유리하게 작용할 수 있는 특별한 형태의 데이타 지역성을 이용하도록 벤치마크 프로그램들을 전혀 수정하지 않았다. 만약, 컴파일러나 운영체제가 이러한 데이타 지역성을 이용하여 코드 생성과 태스크 할당을 할 수 있게 된다면, 세그먼트 디렉토리의 효과는 더욱 향상될 것이다.

참 고 문 헌

[1] Archibald, J. K. and Baer, J.-L., "Cache coherence protocols: Evaluation using a multiprocessor simulation model," ACM Trans. on Computer Systems, Vol.4, No.4, pp. 273-298, 1986.

[2] Tomašević, M. and Milutinović, V., "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors: Part 1," IEEE Micro, vol. 14, no. 5, pp. 52-59, Oct. 1994.

[3] Tomašević, M. and Milutinović, V., "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors: Part 2," IEEE Micro, vol. 14, no. 6, pp. 61-66, Dec. 1994.

[4] Censier, L. M. and Feautrier, P., "A New Solution to Coherence Problems in Multicache Systems," IEEE Trans. on Computers, Vol.C-27, No.12, pp. 1112-1118, 1978.

[5] Agarwal, A., Simoni, R., Hennessy, J., and Horowitz, M., "An Evaluation of Directory Schemes for Cache Coherence," in Proc. 15th Int'l Symposium on Computer Architecture, pp. 280-289, 1988.

[6] Gupta, A., Weber, W.-D., and Mowry, T., "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," in Proc. 1990 Int'l Conference on Parallel Processing, pp. I.312-I.321, 1990.

[7] Chaiken, D., Kubiatowicz, J., and Agarwal, A., "LimitLESS Directories: A Scalable Cache Coherence Scheme," in Proc. 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems, pp. 224-234, 1991.

[8] James, D. V., Laundrie, A. T., Gjessing, S., and Sohi, G. S., "Scalable Coherent Interface," IEEE Computer, Vol.23, No.6, pp. 74-77, June 1990.

[9] Choi, J. H. and Park, K. H., "Segment Directory : An Improvement to the Pointer in Directory Cache Coherence Schemes," in Parallel Processing Letters, vol. 8, no. 4, pp. 577-588, Dec. 1998.

[10] Choi, J. H. and Park, K. H., "Segment Directory Enhancing the Limited Directory Cache Coherence Schemes," in Proc. of the Merged Symposium of the 13th Int'l Parallel Processing Symposium & the 10th Symposium on Parallel and Distributed Processing, pp. 258-267, Apr. 1999.

[11] Simoni, R., and Horowitz, M., "Dynamic Pointer Allocation for Scalable Cache Coherence Directories," in Proc. Int'l Symposium on Shared Memory Multiprocessing, pp. 72-81, 1991.

[12] Agarwal, A., Bianchini, R., Chaiken, D., Johnson,

K. L., Kranz, D., Kubiatowicz, J., Lim, B.-H., Mackenzie, K., and Yeung, D., "The MIT Alewife machine : Architecture and performance," in Proc. 22nd Annual Int'l Symposium on Computer Architecture, pp. 2-13, 1995.

[13] Simoni, R., "Cache Coherence Directories for Scalable Multiprocessors," Ph. D. Thesis, Stanford University, 1995.

[14] Covington, R. G., Dwarkadas, J. R., Jump, J. R., Sinclair J. B., and Madala S., "Efficient Simulation of Parallel Computer Systems," Int. Journal in Computer Simulation, Vol. 1, No. 1, pp. 31-58, 1991.

[15] Goldschmidt, S., "Simulation of multiprocessors : Accuracy and performance," Ph. D. Thesis, Stanford University, 1993.

[16] Agarwal, A., "Limits on Interconnection Network Performance," IEEE Trans. on Parallel and Distributed Systems, Vol.2, No.4, pp. 398-412, 1991.

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

[18] Scheurich, C. and Dubois, M., "Correct memory operation of cache-based multiprocessors," in Proc. 14th Annual Int'l Symposium on Computer Architecture, pp. 234-243, 1987.

[19] Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., and Hennessy, J., "The Stanford FLASH Multiprocessor," in Proc. 21st Annual Int'l Symposium on Computer Architecture, pp. 302-313, 1994.

최 종 혁

1991년 서울대학교 공과대학 전기공학과 졸업 (공학사). 1993년 한국과학기술원 전기및전자공학과 졸업 (공학석사). 1999년 한국과학기술원 전기및전자공학과 졸업 (공학박사). 1998년 IBM 왓슨 연구소 초빙 연구원. 1999년 ~ 현재 LG 종합기술원 선임연구원. 관심분야는 멀티프로세서, 마이크로프로세서, 병렬처리, 인터넷 인프라스트럭춰.

이 창 규

1997년 한국과학기술원 전기및전자공학과 졸업 (공학사). 1999년 한국과학기술원 전기및전자공학과 졸업 (공학석사). 1999년 ~ 현재 한국과학기술원 전자전산학과 박사과정. 관심분야는 인터넷 인프라스트럭춰, 멀티프로세서.

박 규 호

1973년 한국과학기술원 전기 및 전자공학과 공학사. 1975년 한국과학기술원 전기 및 전자 공학과 공학 석사. 1975년 ~ 1978년 동양정밀에서 EPABX 개발팀. 1983년 파리 제 11 대학 공학 박사. 1983년 ~ 1999년 한국과학기술원 교수. 현재 한국과학기술원 정교수. 관심분야는 병렬처리, 컴퓨터 구조, 컴퓨터 비젼 등임.