特輯

확장성 다중처리 시스템 구조

李 俊 元
韓國科學技術院 情報 및 通信工學科

Ⅰ. 서 론

다중처리 컴퓨터(MP:Multiprocessor)는 오랫동안 많은 분야의 관심을 끌어왔는데 그 이유로는 기존의 프로그램을 병렬처리하여 더 빨리 수행할 수 있고, 멀티유저 환경에서 시스템 전체의 성능을 향상할 수 있기 때문이다. 후자의 경우는 각 유저 또는 트랜잭션이 독립적으로 수행되므로 운영체계의 병렬성 이외에는 시스템의 확장을 제한하는 요소가 비교적 적은 반면 전자의 경우는 병렬화된 응용 프로그램의 각 쓰레드(thread)들이 자료의 공유나 동기화(synchronization)를 필요로 하여 확장성 시스템의 구성을 어렵게 한다. 자료관리시스템(DBMS)을 포함한 많은 응용프로그램들이 최근 들어 병렬화되는 추세에 따라 다중처리 시스템 구성시 확장가능성(scalability)의 중요성은 더욱 강조되고 있다.

확장가능성의 중요성은 공유메모리 다중처리 시스템에서 더욱 강조되는데 그 이유는 하이퍼큐브와 같은 메시지 패싱 구조는 각 프로세싱 요소들이 서로 공유하는 것이 없으므로 확장이 쉽기때문이다. 확장성 구조의 가장 중요한 장점은 다양한 성능의 다중처리 시스템을 같은 구조로 설계 및 구현이 가능하며 사용자의 요구에 따라 시스템 성능을 확장할 수 있다는 것이다.

현재의 컴퓨터 구조상 확장성을 제약하는 요소들로는 캐쉬 일관성 방법(cache coherence protocol), 접속망 구성(interconnection network), 메모리 모델 및 동기화를 들 수 있다. 본 논문에서는 이러한 문제점들이 어떻게 확장성을 제한하며 최근 학계 및 산업계의 연구 경향과 새로운 방법을 제안한다. 또한 확장성 시스템의 구성 예와 확장성 접속망의 표준 예를 포함하여 구현시의 문제점도 제시하였다.

Ⅱ. 확장가능성의 의미(Notions of Scalability)

'확장가능성'이라는 용어에 대한 의미는 매우 직관적이지만 많은 사람이 동의하는 정확한 의미를 정의하기는 어렵다. 직관적 의미로는 전체 시스템 규모가 커지면 전체 성능도 향상해야 한다는 조건을 포함하지만 응용 문제나 그 문제를 해결하려는 알고리즘에 따라 성능의 현저한 차이가 존재하므로 특정한 응용분야를 지칭하지 않고는 어떤 시스템이 일반적으로 확장가능하다고 이야기하기가 힘들다.

M.D.Hill은 [6] 에서 speedup의 의미를 이용하여 확장가능성을 다음과 같이 정의하였다.

"A system is scalable if its speedup is linearly proportional to n for all algorithms, number of processors n, and problem sizes."

이 정의에 따르면 어떤 시스템도 확장가능하지 않기 때문에 이 정의는 시스템 설계시의 목적으로 사용될 뿐 확장성의 정도를 표시하는데는 부적합하다

A.Agarwal은 [18] 에서 확장가능성을 다음과 같이 정의하였다.

$$\text{scalability} = \frac{\text{asymptotic speedup on real machine}}{\text{asympototic speedup on an EREW PRAM}}$$

여기서 asympototic speedup은 주어진 알고리즘에서 어떤 시스템이 달성할 수 있는 최대의 speedup을 말하며 EREW(exclusive read and exclusive

write) PRAM은 메모리 작동이 병렬적으로 일어나지 않는 병렬 시스템을 말하며 이러한 가상의 이론적 시스템을 기준으로 하였기 때문에 이 정의 역시 신용성이 결여되어 있다

MIPS(million instruction per second)와 같이 일반적 의미를 갖는 '확장가능성'을 정의하는 것이 여러 다중처리 구조들을 비교하는데 유용하겠지만 아직은 이 분야가 초보단계임을 고려할 때 전체 시스템의 확장가능성보다는 각 기능별로 상대적인 확장가능성을 비교할 수 있는 order 함수(보통 대문자 O로 표시함)를 본 논문에서는 사용하였다. 그 예로는 수행 시간, 하드웨어의 복잡도 등이며 그것이 시스템의 비용이나 부담(overhead)일때 선형 함수 형태인 O(n)으로 나타나면 양호하다고 말할 수 있다.

Ⅲ. 확장성의 제약요인과 해결 방법

다중처리 시스템의 프로세서가 늘어 날때 전체 성능이 선형적으로 증가하지 못하는 이유는 알고리즘 자체에 순차적인 부분이 많거나, 병렬화 작업시 소프트웨어의 비효율성 등도 있겠지만, 본 논문에서는 이보다는 하드웨어 측면에서 그 요인들을 분석하였다.

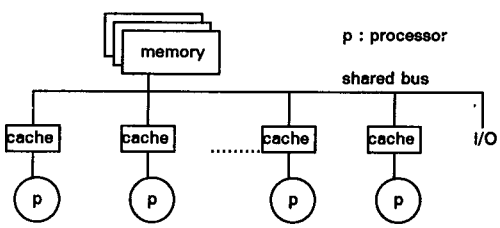

그림 1. 공유버스를 이용한 다중처리 컴퓨터

프로세서의 속도는 메모리에 비해 현저하게 빠르며 이는 현 메모리 제작 기술인 DRAM의 특성에 원인이 있으며 DRAM이 사용되는 한 이 차이는 계속 존재할 것이다. 이 두 부분의 속도차이를 완화하기 위해 캐쉬 메모리가 오래전부터 사용되어 왔다. 확장성 다중처리 시스템에서는 또 다른 종류의 메모리 지연이 발생하는데 이는 주로 다중처리 시스템을 연결하기 위해 사용되는 접속망(interconnection network)에 기인하며, 그 예로 공유버스(shared bus)로 여러 프로세서들을 연결하면(그림 1 참조), 각 프로세서들은 공유버스를 통해 메모리를 사용하므로 프로세서의 개수가 확장됨에 따라 공유버스는 쉽게 포화 상태에 이르며 이로 인해 메모리 지연은 더욱 악화될 것이다. 이러한 문제점을 해결하기 위해 MIN(multistage interconnection network)을 사용하는 경우가 많은데(그림 2 참조) MIN은 프로세서들과 메모리들 사이에 여러개의 경로를 제공하여 공유버스의 포화문제는 제거하였지만 MIN의 여러 단계들(stages)을 거쳐서 메모리 접근을 해야 하므로 이 역시 상당한 메모리 지연이 발생하게 된다.

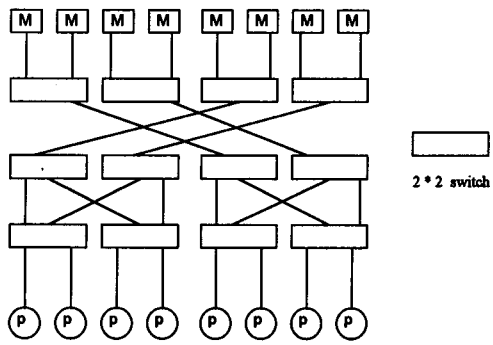

그림 2. MIN을 이용한 다중처리 컴퓨터

이러한 메모리 지연이 다중처리 시스템의 확장성을 제한하는 주 요인이며 이를 해결하기 위한 새로운 방법들이 다음의 분야에서 강구되고 있다.

1. 공유버스를 이용한 시스템

공유버스를 이용하여 프로세서와 메모리를 그림 1처럼 연결하는 것은 가격도 저렴하고 버스 구조에 대한 기존의 경험과 지식을 활용할 수 있어 소,중형 규모의 다중처리기에 많이 이용되고 있다. 그러나 앞서 지적하였듯이 공유버스는 다중처리기를 확장하는데는 부적합하며 그 이유로는 버스의 전기적 특성도 있지만 그 보다는 버스 사용 빈도가 증가함에 따라 각 버스 사용의 평균대기시간이 증가하여 전체 성능을 저하시키는데 원인이 있다.

이러한 버스 사용을 줄이기 위해 각 프로세서에는 그림 1과 같이 캐쉬 메모리가 주로 사용되는데 이는 각 프로세서의 메모리 접근 빈도를 줄여준다. 그러나 이와 같이 여러개의 캐쉬 메모리가 존재하면 캐쉬 일관성(cache coherence)문제가 발생하는데 이는 같

은 데이타가 여러 곳에 존재하기 때문에 어떤 캐쉬에서 그 공유하는 데이타를 수정할 경우 두개의 다른 값이 존재할 수 있다. 이런 문제를 방지하기 위해서 캐쉬 일관성 방법(cache coherence protocol)이 필요하게 되는데 다른 캐쉬에 있는 copy를 무효화(invalidate)하거나 수정(update)함으로서 일관성을 유지할 수 있다.

이 두가지 방법에 기초한 여러 캐쉬 일관성 방법들의 성능을 모의 실험을 통해 비교한 결과를 보면 [4] 대부분의 방법들은 대동 소이하며 성능 향상을 위해 다음의 문제점들에 주안점을 두고 설계되었다

1) 어떤 캐쉬 라인이 공유되고 있는지 여부

이 사항은 캐쉬 일관성 방법의 성능에 가장 중요한 항목으로 어떤 캐쉬 라인을 수정하려 할때 이 캐쉬 라인이 공유되어 있는지 아닌지 모를 경우 무조건 공유된 것으로 가정해야 함으로 비효율적이다. 가장 쉬운 방법은 주 기억장치에서 이 데이타가 공유되고 있는지를 나타내는 방법이지만 주 기억장치의 모든 메모리 블럭마다 어떤 프로세서가 이 블럭을 캐쉬에 저장하고 있는지에 관한 정보가 필요함으로 비효율적이다. 공유버스를 이용한 시스템에서는 이보다 효율적으로 처리하고 있다. [4]

2) 캐쉬 라인의 오우너쉽(ownership)

처음 캐쉬 메모리에 어떤 데이타가 존재하지 않으면 그 데이타를 메모리나 다른 캐쉬에서 가져오게 되는데 이 때 누가 필요한 데이타를 보내 주어야 하는지 결정해야 한다. 이를 위해 어떤 데이타의 오우너쉽(ownership) 개념이 필요하는데 물론 한 데이타의 오우너쉽은 가변적이며 이를 표시하기 위해 캐쉬 라인의 상태가 하나 더 필요해 진다. 주 기억장치가 오우너쉽을 갖게되면 캐쉬와 캐쉬 사이의 전송과는 달리 많은 시간이 필요해지는데 이는 주 기억장치를 구성하는 DRAM 기술상의 문제로 보통 10배 정도 캐쉬보다 느리다. 이 때 만일 주 기억장치 사용시간동안 공유버스를 놔 주지 않으면 현저한 성능 저하가 예견되므로 반드시 주 기억장치를 이용하는 일은 한 트랜잭션(transaction)을 분리하여 주 기억장치에 주소와 읽기/쓰기 정보를 알려준 후 공유버스를 다른 프로세서가 사용하도록 할 필요가 있다

3) 무효화 대 갱신

일관된 캐쉬의 값을 유지하기 위해 공유된 데이타를 수정하기위해서는 데이타를 공유하는 다른 캐쉬의 라인들을 무효화하거나 갱신시켜 주어야 하는데 이의 선택은 소프트웨어의 메모리 참조 패턴에 달려 있다. 가령 어느 공유 데이타가 서로 다른 캐쉬에서 자주 갱신된다면 갱신 방법이 유리하겠지만 이 공유 데이타가 한 캐쉬에 오랫동안 머문다면 무효화 방법이 우수할 것이다. 이 둘 사이의 비교를 용이하기 위해 S. Eggers는 [12] 에서 write-run이라는 변수를 정의했는데 이는 어느 공유 데이타가 한 프로세서에서 다른 프로세서의 간섭없이 연속적으로 수정되는 횟수로 이 값이 길수록 무효화 방법이 유리하다. 또한 무효화는 전체 캐쉬 라인의 이동을 유발할 가능성이 높지만 갱신은 주로 한 워드의 전송이므로 캐쉬 라인 크기도 이 둘을 선택할 때 고려해야 한다. 이러한 변수들의 자세한 분석은 [15] 에서 참조할 수 있다

캐쉬 일관성 방법을 설계할 때는 위의 사항들을 결정해야 하는데 시스템 설계시 목표로 하는 성능과 주어진 하드웨어 기술을 고려하여야 한다. 대부분의 공유버스를 이용한 시스템에서는 스누핑캐쉬(snooping cache)라 하여 각 캐쉬 컨트롤러가 공유버스의 모든 활동 상황을 감시하여 필요시 버스에 반응을 보내거나 자체 캐쉬 라인의 상태를 변경하게 된다. 스누핑 캐쉬를 설계할 때 또 다른 주의점은 공유버스에서 관찰되는 모든 메모리 주소와 자체 캐쉬에 존재하는 캐쉬 라인들의 주소의 비교가 빈번함으로 자체 프로세서의 캐쉬 접근을 방해할 가능성이 높다(그림 3 참조). 따라서 적어도 캐쉬의 택(tag) 필드는 동시에 둘(버스와 프로세서)의 접근을 허용하여야 한다.

앞에서도 기술하였듯이 공유버스를 기초로 하는 시스템은 공유버스 자체가 병목 현상을 야기하며 버스 전체 길이의 제한 때문에 확장성 시스템으로는 부적합하지만 위에서 기술한 캐쉬 프로토콜 관련 문제와 뒤에 기술할 메모리 모델의 여러 문제점들을 해결함으로써 어느 정도(20 프로세서 정도)까지는 확장 가능하다.

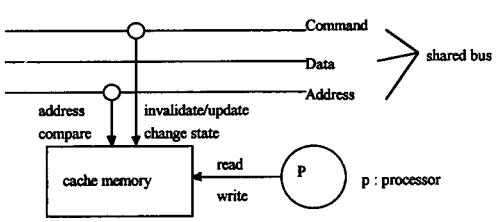

그림 3. 프로세서와 스누핑 캐쉬의 경쟁

2. MIN을 이용한 시스템

공유버스와는 달리 MIN은 프로세서와 메모리 사이 또는 프로세서와 프로세서 사이에 여러 경로를 제공하여 공유버스의 병목 현상을 제거하였지만 스누핑 캐쉬의 사용 불가능과 다단계 특성에서 기인한 메모리 지연 문제가 새로이 제기되고 있다.

MIN에서는 빠른 브로드캐스트(broadcast)방법의 구현이 불가능하므로 Point-to-Point 통신에 의해 캐쉬 일관성 방법을 설계한다. 대표적인 방법은 디렉토리(directory)를 이용한 방법으로 주 기억장치의 각 블럭들은 어떤 프로세서의 캐쉬에 그 블럭이 저장되어 있고 그것이 쓰기를 목적으로 하는지 읽기를 목적으로 하는지에 대한 정보를 갖고 있다(그림 4).

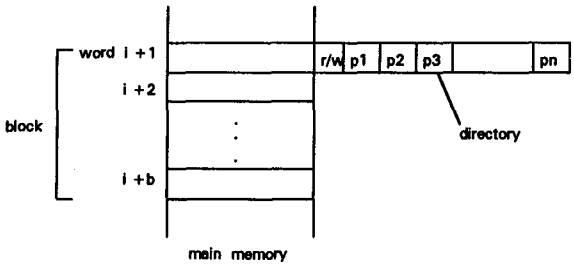

그림 4. 디렉토리 구조(1 메모리 블럭 = b words, n개의 프로세서 가정)

물론 각 캐쉬 라인들도 그 캐쉬 라인의 상태(state)를 표시할 수 있는 필드를 마련하여 불필요한 주 기억장치 접근을 줄이고 있다. 어떤 프로세서(Pk)에서 메모리 i+1에 write를 하려면 디렉토리에서 그 블럭을 공유하고 있는 프로세서들을 찾아서 무효화나 갱신의 신호를 보내고 그 모든 프로세서에서 확인 메시지를 접수한 뒤에 디렉토리의 r/w bit를 w로,Pk의 bit를 1로, 나머지는 모두 0으로 설정한 뒤 write를 수행한다. 또한 계속된 write에 대해 이러한 작업의 반복을 피하기 위해 자체 캐쉬 라인에 modified bit를 설치하여 write가 자기 캐쉬의 modified된 캐쉬 라인에 대한 것이면 위의 작업 수행없이 자체적으로 수행할 수 있다.

위와 같은 방법상의 문제점은 디렉토리를 위한 메모리 공간(O(n2))과 무효화 메시지와 응답 메시지가 MIN통과시 소요되는 시간이다. 디렉토리 공간을 줄이기 위해 IEEE에서 정의한 SCI(Scalable Coherent Interface) [8] 에서는 디렉토리 정보를 각 캐쉬 라인에 분산하여 linked-list로 구성하였다(그림 5). 이 방법은 확장성에서 매우 우수하지만 write 작업시 무효화 메시지가 linked-list를 따라 마지막 프로세서까지 전달되는데 많은 시간이 걸려 전체 성능이 데이타의 공유되는 정도에 따라 많은 차이를 보일 것이다.

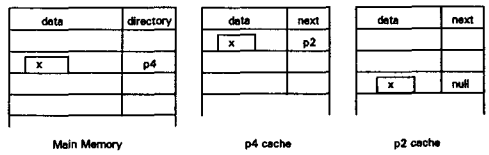

그림 5. SCI Protocol하에서 데이타 X를 P2와 P4가 공유한 상태

메시지가 MIN을 따라 전달될 때 발생하는 지연을 줄이기 위해서는 단계의 숫자를 줄이거나(switching degree를 높임으로 가능함), 다음 절에서 논술할 메모리 모델을 통해 가능하다.

3. 메모리 모델(Memory Model)

단일 프로세서 시스템의 메모리 작동순서에서 read의 의미는 '가장 최근에 쓰여진 값을 읽어옴'으로 비교적 오해의 소지가 없으나 MP에서는 동시에 여러 곳에서 같은 데이타가 쓰여질 가능성이 있으므로 '가장 최근'이라는 의미가 정확하지 않다. L. Lamport는 [9] 에서 이러한 MP 메모리 모델의 부정확성을 시정하기 위해 sequential consistency를 정의하였는데 이는 database의 serializability의 의미와 유사하며 어떠한 버퍼링(buffering)도 허용하지 않은 모델이다. 따라서 이 모델을 기초로 한 시스템은 긴 메모리 지연을 감수해야 한다.

이러한 sequential consistency는 필요 이상으로 메모리 접근 순서를 제한함으로서 시스템의 성능을 저하시키므로 최근에는 메모리 모델을 완화하여 무순서 실행(out-of-order execution)이나 캐쉬 미스시의 부담을 줄이려는 시도가 활발해지고 있다. [2,10,16] 이러한 시도 중 분산환경을 겨냥한 PRAM(pipelined RAM) [17] 이나 slow memory [7] 는 프로그래머가 이 메모리 모델에 주의하여 프로그램을 개발해야 하므로 sequential consistency를 가정한 프로그램을 실행할 경우는 틀린 결과가 발생할 수도 있다

본 논문에서 제안하는 모델은 기존의 동기화 요구사항을 만족하는 모든 프로그램은 항상 옳바른 결과

를 생산하도록 되어 있다. 동기화 요구 사항이란 일관성이 요구되는 공유 데이타를 사용하기 위해서는 critical section과 같은 동기화 작업을 요구하는 것으로 이것은 이미 multiprogramming 환경하에서 각 process들이 지키고 있는 사항이므로 별도로 프로그램의 수정이나 새로운 요구 사항이 필요 없다

이 메모리 모델의 특징을 critical section의 예를 들어 설명하면 다음과 같다.

```
read A
write A
lock
read B          (1)
write B         (2)
write C         (3)
unlock
read D
write D
```

그림 6. critical section의 예

위의 critical section은 공유 데이타 B,C의 일관성을 위한 것으로 이를 제외한 모든 데이타 A,D는 서로의 의존성(dependency)만 존재하지 않는다면 어떤 순서로 이용해도 수행 결과는 항상 옳다. 그러나 B를 읽기 전에 lock은 반드시 시스템 전체적으로 관련된 모든 작업(필요하다면 무효화 메시지 및 응답 메시지의 전달)이 완료되어야 한다. 그러나 (3)은 (2)의 완료를 기다릴 필요가 없으므로 (2)에서 야기된 무효화 메시지가 다른 프로세서에 전달되어 필요한 캐쉬 라인을 무효화하고 응답 메시지가 돌아올 때까지 기다릴 필요가 없다

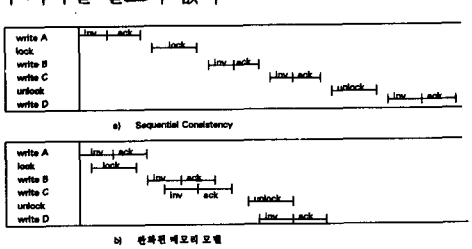

그림 7. 각 메모리 모델에서 critical section의 수행 과정

A,B,C,D 모두가 공유되어 있다고 가정할 때 sequential consistency와 새로 완화된 메모리 모델이 디렉토리를 이용한 시스템에서 위의 코드 중 write문을 수행하는 상태를 도표로 나타내면 그림 7과 같다.

전체 시스템에 N개의 프로세서를 가정하면 보통 한 메시지가 MIN을 통과하는데 logN 시간이 걸린다. 결과적으로 sequential consistency의 경우 8 * logN + 5 + 2 * synch의 시간이 걸리는데 여기서 5는 각 인스트럭션을 수행하는데 필요한 시간(1 clock/instruction)이며 synch는 동기화 작업에 필요한 시간이다. 반면 완화된 메모리 모델에서는 대부분 앞의 문장이 끝나는 시점을 기다리지 않고 다음 문장을 수행하여 전체 수행시간을 줄일 수 있다. 이것은 write문 사이에 상관 없는 read문이나 다른 것들이 있어도 같은 결과이다. 전체 수행시간은 2 * logN + 5 + synch로 sequential consistency와 비교하여 현저한 차이를 보이며 이 차이는 MP뿐만 아니라 싱글 프로세서에서도 나타난다. 이는 캐쉬 미스 발생시 프로세서를 멈추지 않고 계속 수행 가능하기 때문이다.

위와 같은 완화된 메모리 모델에 관한 정확한 정의는 [2,11,16] 에서 찾아볼 수 있다

4. 동기화 작업(synchronization)

대부분의 병렬시스템에서는 각 쓰레드들이 동기화 작업을 빈번히 수행하는데 이는 주로 각 쓰레드의 수행을 순서화하거나 공유데이타의 일관성을 유지하기 위해서이다. 기존의 단일 프로세서 시스템에서는 대부분 test_and_set을 사용하여 여러 종류의 동기화 명령어, 즉 lock, semaphore, monitor 등을 구현하는데 MP환경에서는 test_and_set이 기존의 캐쉬 일관성 방법하에서 다음과 같은 문제를 야기한다.

일반적으로 lock명령어는 while (test_and_set(A) = = 1); 와 같이 busy-wait방법을 사용하는데 이 때 'set' 부분은 쓰기와 동일한 작업이므로 여러 프로세서가 이를 수행하게 되면(무효화 방법일 경우) A와 같은 lock variable을 포함한 캐쉬 라인이 끊임없이 프로세서들사이에서 전송하게 된다. 이를 완화하기 위해 캐쉬 일관성이 필요한 MP에서는 다음과 같이 wait상태에서는 쓰기가 일어나지 않도록 하고 있다.

```
repeat
    while ( A = = 1);
until (test_and_set(A) = = 0);
```

이러한 방법을 사용하여 busy-wait시 캐쉬 무효화를 방지하더라도 lock variable A가 unlock되는 순간 기다리던 모든 프로세서들이 test_and_set을 수행함으로 이 역시 상당한 부담이 되고 있다.

위와 같은 문제를 해결하기 위한 소프트웨어 방법은 lock을 수행하는 모든 프로세서들을 순서화하여 어떤 프로세서가 busy-wait해야할 variable을 각 프로세서별로 서로 다르게 할당하여 한 variable에 한 프로세서만이 busy-wait하게 한다. 이 variable들은 주로 array로 구현하는데 주의할 점은 각 array element들이 서로 다른 캐쉬 라인에 mapping해야지 그렇지 않으면 서로 다른 variable을 사용하면서도 마치 이 variable들이 공유된 것처럼 처리될 것이다(false sharing).

하드웨어적 해결 방법은 주로 캐쉬 라인 상태에 있는 그 라인이 lock variable을 소유할 경우 이것이 lock된 상태인지를 표시하여 캐쉬 일관성 방법에 lock 명령에 대한 적절한 동작을 포함시킨다. [2,13] 대부분의 확장성 시스템은 캐쉬 일관성에 관한 정보를 분산시키며 이 때 lock에 관한 정보 역시 포함시킴으로서 lock에 관련된 사항을 캐쉬 일관성과 함께 정의할 수가 있다. 그 예로 SCI의 경우 같은 변수를 읽기로 공유하는 경우 linked-list로서 해당 프로세서들이 연결하는데 이 같은 구조는 read-lock에서도 필요하므로 쉽게 다음과 같이 read-lock을 구현할 수 있다.

```
read-lock(A)
    if (mem.lock = = unlocked) then
    {       mem.lock = read-lock;
            mem.tail = my-id;
    }
    else  /* A is unlocked */
    {
            mem.tail->next = my-id;
            mem.tail = my-id;
            cache.state = read-lock
    }
```

위에서 메모리 디렉토리는 lock의 상태와 linked-list의 끝(tail)에 관한 정보를 갖고 있으며 wait상태의 모든 프로세서들은 해당 캐쉬 라인의 next필드를 이용하여 linked-list를 구성하여 기다린다. 이 때 새로운 lock 요구에 대해서는 tail 프로세서가 응답함으로서 lock에 관한 업무가 모든 프로세서 사이에 균등하게 분할된다. 더우기 이 방법은 새로운 하드웨어가 거의 필요 없이 SCI표준에 맞는 모든 시스템에 쉽게 적용 가능하다.

이러한 캐쉬 라인을 이용한 동기화 작업의 문제는 해당 캐쉬 라인이 다른 데이타로 교환(replace)되었을때 linked-list를 유지하는데 있다. 프로세서가 busy-wait할 때는 계속 해당 캐쉬 라인을 사용하므로 이런 일이 발생하지 않지만 blocked-wait인 경우나 lock을 획득한 프로세서는 해당 캐쉬 라인을 교환할 가능성이 있다. 이 두 가지의 경우 다 언젠가는 (lock의 경우 unlock, wait인 경우 wakeup) 해당 캐쉬 라인을 동기화 작업에 사용하므로 next 필드 및 동기화에 이용하는 정보를 그대로 캐쉬 라인에 유지할 수 있다. 이 경우 물론 컴파일러는 한 캐쉬 라인에 하나 이하의 lock variable을 할당해야 한다

5. 확장성 시스템의 예 : DASH

스탠포드 대학에서 구현된 확장성 시스템인 DASH [2]의 예를 보면 전체 시스템은 여러 개의 cluster들로 구성되며 이들의 연결을 현재 Mesh접속망으로 되어 있지만 다른 종류의 MIN으로도 대체할 수 있도록 되어 있다(그림 8 참조).

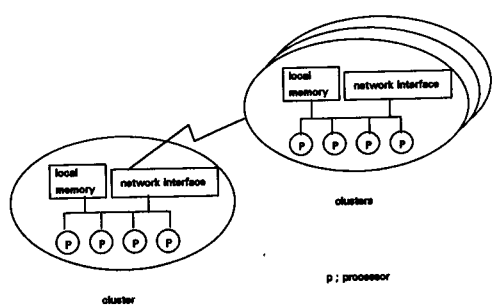

그림 8. DASH의 구조

앞에서도 지적했듯이 공유버스의 경우 캐쉬 일관성 유지 및 여러 부분에서 장점이 있지만 확장성면에서는 많은 취약점이 있다. 이를 해결하기 위해 한 cluster에 존재하는 4개의 프로세서는 공유버스를 이용한 스누핑 캐쉬로 구성하고 각 cluster사이에는 디렉토리를 이용하여 캐쉬 일관성을 유지했다. 이러한 구성은 communication locality 즉 프로세서간의 통신이 국부적이라는 가정에 기반을 두고 있다. 만일 프로세서간의 통신이 여러 cluster에 걸쳐 자주 발생

하는 경우 위의 구성 방법은 심각한 성능 저하 현상을 보일 것이다.

DASH 시스템은 그 밖에 앞에서 제안한 메모리 모델과 캐쉬 라인을 이용한 동기화 방법 등을 실현하여 전체 성능면에서 지금까지 제안된 다른 어떤 시스템보다 우수하지만 디렉토리 구조상의 문제점, 즉 O(n2)의 공간을 요구하는 점,때문에 확장성에 문제가 있다. 이를 위하여 SCI에서 제안된 방법과 같이 디렉토리의 구조를 변경할 필요가 있다.

Ⅳ. 결론 및 기술 동향

본 논문에서는 확장성 다중처리 컴퓨터의 여러 현안에 대해 깊이보다는 폭 넓게 소개하며 새로운 메모리 모델과 동기화 방법을 제안하였다. 앞에서도 언급했듯이 모든 확장성에 관련된 문제는 메모리의 공유 데이타를 사용하는데 있다. 여기서 발생하는 memory latency를 해결하기 위한 하드웨어 방법들이 본 논문에서 거론되었지만 보다 나은 성능 향상과 확장성을 위해서는 소프트웨어의 도움이 절실한 실정이다. 예를 들어 어떤 데이타가 공유되고 있는지를 소프트웨어가 알 수 있다면 앞에서 소개한 복잡한 캐쉬 일관성 방법들이 필요 없다. 그러나 현재 대부분의 프로그래밍 환경은 컴파일러가 이러한 기능을 수행할 수 없게 되어있다.

앞으로의 연구는 하드웨어에 의존한 해결보다는 새로운 병렬처리용 프로그래밍 언어, 새로운 컴파일러, 새로운 프로그래밍 도구를 이용하여 본 논문에서 지적한 확장성의 문제점들을 해결하는 것이 바람직하다. 그 이유는 아직까지 알려진 하드웨어 방법들이 뚜렷한 한계를 보이고 있으며 앞으로 현저하게 발전하리라고 기대하기 어려운 실정이기 때문이다. 더욱이 병렬처리용 소프트웨어는 이제 초보적 단계에 있으므로 확장성을 위한 특정한 프로그래밍 패러다임(paradigm)을 강요할 수 있기 때문이다

參考文獻

[1] S.V. Adve and M.D.Hill, "Weak ordering - a new definition", *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp.2-11, May 1990.

[2] D.Lenoski, J.Laudon, T.Joe, D. Nakahira, L.Stevens, A.Guita, and J. Hennessy, "The DASH Prototype: Implementation and Performance", *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp.92-103, May 1992.

[3] T.E.Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, 1(1): 6-16, Jan. 1990.

[4] J.Archibald and J.Baer, "Cache coherence protocols: evaluation using a multiprocessor model", *ACM Transactions on Computer Systems*, pp. 278-298, Nov. 1986.

[5] M.Dubois, C.Scheurich, and F.Briggs, "Memory access buffering in multiprocessors", *In Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 434-442, June 1986.

[6] Mark D.Hill, "What is scalability?", *Computer Architecture News*, Dec. 1990.

[7] P.Hutto and M.Ahamad, "Slow memory: Weakening consistency to enhance concurrency in distributed shared memory", Technical Report GIT-ICS-89/39, Georgia Institute of Technology, Oct.1989.

[8] IEEE P1596-SCI Coherence Protocols, *Scalable Coherent Interface*, version 1.0 edition, Jan.1991.

[9] L.Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE Transaction on Computers*, C-28(9):690-691,

Sep.1979.

[10] C.Scheurich and M.Dubois, "Correct memory operation of cache-based multiprocessors", *In Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 234-243, June 1987.

[11] J.Torrellas and J.Hennessy, "Estimating the performance advantages of relaxing consistency in a shared memory multiprocessor", *In Proceedings of the 1990 International Conference on Parallel Processing*, pp.I:26-33, 1990.

[12] S.J.Eggers and R.H.Katz, "A characterization of sharing in parallel programs and its application to coherency protocol evaluation", *In Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp.373-382, June 1988.

[13] J.R.Goodman, M.K.Vernon, and P.J. Woest, "Efficient synchronization primitives for large scale cache-coherent multiprocessor", Technical Report TR-814, Univ. of Wisconsin at Madison, Jan.1989.

[14] G.Graunke and S.Thakkar, "Synchronization algorithms for shared-memory multiprocessors", *IEEE Computer*, 23: 60-69, June 1990.

[15] A.R.Karlin, M.S.Manasse, L.Rudolph, and D.D.Sleator, "Competitive snoopy caching", *Algorithmica*, 3:79-119, 1988.

[16] Joonwon Lee and Umakishore Ramachandran, "Architectural primitives for a scalable shared memory mutiprocessor", *In ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[17] R.J.Lipton and J.S.Sandberg, "PRAM: A scalable shared memory", Technical Report CS-TR-180-88, Princeton University, September 1988.

[18] D.Nussbaum and A.Agarwal, "Scalability of parallel machines", *CACM*, 34(3): 57-61, March 1991.

[19] S.Thakkar, P.Giffold, and G.Fielland, "The Balance multiprocessor system", *IEEE Micro*, pp.57-69, February 1988.

筆 者 紹 介

李 俊 元

1958年 9月 7日生

1983年 7月 서울대학교 계산통계학과 졸업(학사)

1990年 6月 Georgia Tech 전산학 석사

1991年 6月 Georgia Tech 전산학 박사

1983年 8月 ~ 1986年 7月 (주) 유공

1991年 6月 ~ 1992年 8月 미국 IBM 연구원

1992年 9月 ~ 현재 한국 과학 기술원 정보및 통신공학과 교수

주관심 분야 : 전산기 구조, 병렬 처리, 운영 체계