2001년도 대한전기학회 하계학술대회 논문집 2001.7.18-20

Intel486 병렬시스템의 Cache Coherence를 위한 Central Directory Unit의 설계

유 준 복*, 정 태 상
중앙대학교 전자전기공학부

Design of Central Directory Unit for Cache Coherence of Multiprocessor based on Intel486 Microprocessor

You Jun-Bok*, Chung Tae-Sang
School of Electronics and Electrical Engineering, Chung-Ang University

Abstract - In order to utilize cache in multiprocessor system, cache coherence problem must be handled. Central directory scheme is one of hardware-assisted cache coherence solutions. The goal of this paper was not only to propose some special methods needed to apply central directory scheme to the specific multiprocessor system based on Intel486 microprocessors but also to design central directory unit for cache coherence of the target system. The problems of arbitrating several requests from processors, storing the cache information, and generating control signals for cache line fill and snoop cycle were solved.

1. 서 론

Cache coherence 문제는 서로 메모리 데이터를 공유하는 병렬시스템에서 cache의 사용으로 인하여, 하나의 공유데이터가 전체시스템에서 동일한 값을 유지하지 못하는 상황이 발생하고, 그 결과로 전체 시스템이 정상동작을 할 수 없게 되는 것을 의미한다. 본 논문은 병렬시스템에서의 cache coherence 문제를 하드웨어적으로 해결하는 방법인 central directory cache coherence 방법을 직접 구현해 보는 것을 그 목적으로 하였다. 대상이 된 시스템은 두 개의 Intel486 프로세서 보드가 시스템 버스로 연결되어 서로의 메모리를 공유해서 동작하고, 각 프로세서는 내부적으로 가지고 있는 cache을 사용하는 시스템이다. Cache coherence 방법으로는 directory based cache coherence 방법을 채택하였고, 그 구현을 위해서 Intel486 프로세서와 내부 cache의 특성을 고려한 central directory unit을 설계하였다.

2. Cache Coherence 문제

메모리 공유 병렬시스템은 다수의 프로세서가 하나의 메모리 영역을 같이 사용하는 시스템을 의미한다. 공유 메모리 영역에 존재하는 데이터는 항상 모든 프로세서가 read 혹은 write 동작이 가능한 데이터이다. 메모리 공유 병렬시스템에서 프로세서가 cache를 가지고 있을 경우, 어떤 공유 데이터의 값이 전체 시스템에 여러 개 존재할 수 있으며, 이 경우 coherence 문제가 발생하게 된다. 그림 1과 같이, 프로세서 A가 공유 메모리에서 데이터 X를 읽으면, 자신의 cache에 데이터 X를 저장하게 된다(①). 만일, 프로세서 A의 cache가 write-back cache이면, 프로세서 A의 데이터 X에 대한 write 동작이 발생했을 때, cache 데이터에 대해서만 실행되기 때문에(②), 데이터 X의 값은 X와 X′ 두 개가 존재한다. 이 때, 프로세서 B에서 공유 데이터 X에 read 동작이 발생한다면, 프로세서 B는 변경된 데이터 X′가 아닌, 잘못된 데이터 X를 저장하게 된다(③).

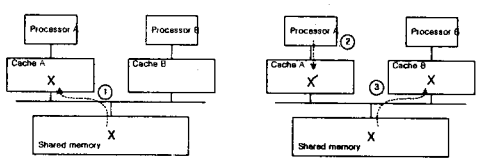

그림 1 Cache Coherence Problem

이와 같은 문제를 해결하기 위해서 여러 가지 방법들이 사용되어 지고 있으며, 하드웨어적인 해결 방법으로 central directory cache coherence 방법이 있다.

2.1 Central Directory Cache Coherence 방법

Central directory 방법은 모든 프로세서의 cache 정보를 directory에 저장하고, 저장된 cache 정보를 이용해서, 공유 데이터가 coherence를 유지할 수 있도록 한다. Central directory unit은 전체 시스템에서 발생하는 모든 동작을 감시하고, 동작이 발생하면, directory에 가지고 있는 cache 정보를 이용해서, 관련된 프로세서의 cache에 필요한 신호를 보내주고, 동작을 발생한 프로세서에 올바른 동작을 할 수 있는 정보를 제공한다. 발생한 동작의 결과로 변하는 cache 정보를 다시 directory에 저장한다.

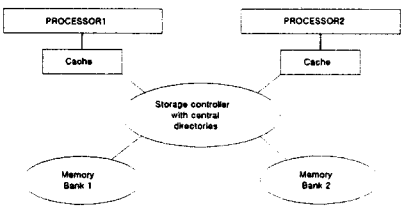

그림 2 Central Directory Cache Coherence

3. Intel486 DX4 프로세서의 내부 Cache

Intel486 DX4 프로세서는 16Kbyte의 크기를 가지는 unified cache이다. Line size는 16bytes이고, 4-way set associative 구조를 가지고 있다. Cache coherence를 위하여 snoop cycle을 지원하고, Modified MESI protocol을 사용한다.

3.1 Snoop Cycle

내부 cache를 사용하는 경우, cache에 저장된 데이터와 메모리에 저장된 데이터가 항상 일치하지 않는다(write-back cache). 따라서, 다른 버스 마스터에 의한 메모리 접근 동작이 있을 경우, 잘못된 데이터 전송을 방지하기 위한 동작이 필요하다. Snoop cycle은 프로세서의 내부 cache에 어떤 line의 존재 여부를 판단하거나, 그 line을 어떤 이유로 인하여 invalidate 시킬 필요가 있을 경우 실행되는 동작이다. Central directory unit은 DX4 프로세서의 snoop 기능을 이용해서, 저장된 line을 invalidate 시키거나, 변경된 line을 메모리에 write-back 하도록 한다.

3.2 Write-Back Cache Coherence Protocol

DX4 프로세서 내부cache는 cache coherence를 위한 protocol로 "Modified MESI protocol"를 사용한다. 저장되는 각 line들은 M, E, S, I, 4개 중 하나의 상태를 가지게 되며, 각각의 상태가 의미하는 것은 다음과 같다.

- **M(Modified)**: Line의 내용이 메모리와 틀리다. Read와 write 동작이 모두 메모리 접근 없이 cache에서만 실행된다.
- **E(Exclusive)**: Write-back line. 메모리와 동일한 내용을 가지고 있다, Write 동작이 발생했을 경우, 상태가 M으로 변하고, 메모리에 데이터를 쓰지 않는다.
- **S(Shared)**: Write-through line. Read 동작의 경우에는 메모리 접근 없이 cache 내부에서만 이루어지고, write 동작의 경우에는 cache와 메모리 모두에 write를 한다. 메모리와 내용이 항상 일치한다.
- **I(Invalid)**: Cache에 저장되어 있지 않음을 의미한다. Read 동작이 발생하면, cache line fill 동작이 발생하고, 메모리에서 한 line을 읽어온다. write 동작의 경우는 메모리에만 write 동작을 한다.

Central directory unit은 DX4의 MESI protocol을 기준으로 directory 메모리에 cache 정보를 저장하게 된다.

4. Central Directory Unit의 설계

대상 시스템은 두 개의 Intel486 DX4 프로세서 보드로 구성된 NUMA 시스템이다. 모든 메모리 접근 동작은 항상 central directory unit을 통해서 적당한 동작 신호의 생성 후에 다시 동작하게 된다. Central directory unit은 항상 모든 프로세서 보드에서 발생되는 메모리 접근 신호를 감시한다. 만일 어느 보드에서 메모리 접근 신호가 발생되면, 해당 보드의 동작을 정지시키고, directory check cycle을 시작하게 된다. Directory check cycle은 directory 메모리로부터 요구되는 line의 cache 정보를 읽고, 현재의 발생한 메모리 접근 신호와 조합하여, 동작에 필요한 신호들을 생성한다. 그 후에 정지되어 있던 프로세서 보드를 동작시켜서, 동일한 동작을 다시 시작하도록 한다. 다시 시작되는 동작은 이미 directory check cycle을 통하여, 필요한 동작 신호를 생성하고, directory 메모리에 저장될 정보들을 변경했기 때문에 다시 central directory unit을 거치지 않고, 정상 동작을 하게 된다. 정상동작은 directory check cycle을 끝낸 후 프로세서가 발생했던 동작을 끝마치는 동작을 의미한다.

실험에 사용된 시스템에서 가능한 메모리 접근 동작은 크게 4가지로 나누어 볼 수 있다.

REQ_00: 보드 B0에서 local 메모리(M0)를 접근
REQ_01: 보드 B0에서 remote 메모리(M1)를 접근
REQ_11: 보드 B1에서 local 메모리(M1)를 접근
REQ_10: 보드 B1에서 remote 메모리(M0)를 접근

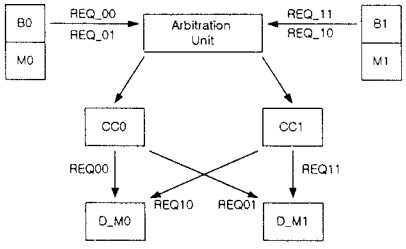

그림 3 실험시스템 블록도

4.1 Central Directory Unit

Central directory unit은 arbitration unit, coherence controller, directory 메모리, replacement controller, replacement 메모리로 구성된다. Arbitration unit은 각 보드에서 발생한 REQ신호들을 중재하여 어드레스 버스의 사용, directory 메모리 사용에 있어서 충돌을 막는 기능을 하고, coherence controller는 directory check cycle을 수행하여, 각 보드에서의 메모리 접근 동작이 cache coherence를 유지하면서 동작할 수 있도록 신호들을 만들고, directory 메모리에 저장될 정보를 변경한다. Coherence controller는 각 보드에 하나씩 할당되어 다른 coherence controller와는 독립적으로 동작하게 된다. 현재의 시스템은 2개의 보드로 구성된 병렬시스템을 모델로 설계했으므로, CC0, CC1, 두 개의 coherence controller가 있다. Directory 메모리에는 보드의 local 메모리를 기준으로 할당된다. D_M0는 M0의 모드 데이터에 대한 cache 정보를, D_M1은 M1의 모든 데이터의 cache 정보를 저장하고 있으며, replacement controller와 replacement 메모리는 프로세서 보드 내에서 발생되는 replacement동작으로 인하여 발생 가능한 문제를 처리하기 위해 설계되었다.

4.2 Cache State

Central directory unit의 모든 동작은 보드에서 발생한 동작(read 혹은 write)과 directory 메모리에 저장된 정보에 의해서 결정된다. 저장되는 cache 정보는 다음과 같은 기준으로 정하였다.

State	b1b0
I	0 0
E	1 1
S0	0 1
S1	1 0

표 1 Line의 상태 정의

표 1과 같이 가능한 line은 상태는 I, E, S0, S1, 4가지로 정하고, 각 상태를 2bit을 이용해서 directory 메모리에 저장하였다. 앞서 설명한 M상태는 프로세서 외부에서 그 상태를 파악할 수 없기 때문에 제외되었다.

전체 시스템에서 프로세서 보드가 2개이므로, 어떤 line에 대한 cache 정보는 모두 4bit이 된다. D_M0의 관점으로 설명한다면, b1과 b0는 B0의 내부 cache에 저장된 line 상태를 b2와 b3는 B1의 내부 cache의 line 상태를 나타낸다.

NUMA 시스템의 특성상, 프로세서는 local 메모리에 대한 동작이 remote 메모리에 대한 동작보다 많을 것이다. 따라서, line 상태를 결정함에 있어서 local 메모리에 대한 동작에 우선권을 주면, 전체 시스템의 성능향상을 기대할 수 있다. 다음과 같은 조건을 정해서 local 메모리에 대한 동작을 효과적으로 실행할 수 있도록 하였다. D_M0에 저장되는 값들을 기준으로 설명하겠다.

조건 1. E 상태는 local 메모리 접근 경우에만 가능

B0의 M0에 대한 동작과, B1에서 M0을 접근하는 동작을 서로 구분하였다. ST0, [I I]상태의 line을 B0에서 최초로 read 동작을 하면, 그 line은 E 상태, 즉 write-back 형태로 저장되고, B1에서 최초로 read 동작을 하면, S 상태로 저장하도록 하였다. E 상태의 line은 read 와 write 동작이 모두 내부 cache에 대해서만 이뤄지므로, 빠른 실행 속도와 메모리에 대한 접근을 줄일 수 있다. B1의 경우도 동시에 E상태로 저장한다면, 그 변화를 알 수 없기 때문에 coherence 유지를 할 수 없다. 따라서, 가능한 상태는 ST1, [I E], 만 가능하다.

조건 2. Local 메모리에 대해서는 S1 상태가 존재한다.

표 1에서 같이 S 상태를 S0와 S1, 두 가지로 구분하였다. S0상태로 저장된 local 메모리의 line에 대

해 write 동작이 발생하면, S1 상태로 변하고, 동일 line에 다시 write 동작이 발생하면, I 상태로 변한다. 어떤 line에 대해서 write 동작이 두 번 발생한 것은 B0에서 이 line에 대한 동작이 빈번하다는 것을 말해 주고, line의 상태를 고의로 I 상태로 만들어 다음 read동작에 다시 E 상태로 저장할 수 있는 조건을 만들어 주는 것이다.

State	Line State		b3	b2	b1	b0
ST0	I	I	0	0	0	0
ST1	I	E	0	0	1	1
ST2	S0	I	0	1	0	0
ST3	S0	S0	0	1	0	1
ST4	I	S1	0	0	1	0
ST5	S0	S1	0	1	1	0

표 2 가능한 Cache 정보 상태

M0에 대한 동작(operation)은 B0에서의 read (B0_R) 와 write (B0_W), B1에서의 read (B1_R) 와 write(B1_W), 4가지 경우로 나누어 볼 수 있다. 요구되는 동작에 의한 현재 cache 정보의 상태 변화는 표 3과 같다. Central directory unit은 표 3을 기준으로 해서 각각의 동작과 해당 line의 상태에 따른 동작 신호들을 생성하게 된다. 이미 저장된 line에 대한 read 동작은 외부적으로 어떤 신호도 발생하지 않기 때문에, 그러한 동작들은 제외하고, 발생 가능한 동작들은 모두 16가지이다. ①, ③, ⑦, ⑬은 read 동작으로 인한, line fill 동작이 발생하는 경우이다. Line fill은 16bytes 크기의 line을 cache에 저장하는 동작으로서, central directory unit은 동작에 필요한 신호들을 생성해서 해당 보드로 전송하게 된다. ①의 경우는 local 메모리에 대한 line fill 동작이므로, E 상태로 저장되고, ③의 경우는 remote 메모리에 대한 동작으로 S 상태로 저장되어진다. 이와 같은 구분은 line fill 동작 시 DX4의 WB/WT#의 상태에 따라서 결정되는데, central directory unit은 상황에 따라 WB/WT#신호를 만들어 주게 된다. ⑥, ⑧, ⑩, ⑪, ⑫, ⑭, ⑮, ⑯의 경우들은 write동작에 의해서 해당 line을 cache에 저장하고 있는 다른 프로세서에게 invalidate 동작에 필요한 신호들을 보내준다. 현재 발생한 write동작에 의해서, 해당 line의 데이터는 공유메모리와 동일한 값을 유지 할 수 없으므로, 가지고 있는 line을 invalid상태로 만들도록 신호를 보내준다. ⑫의 경우는 S1 상태의 line에 다시 write동작이 발생한 것으로, 동작을 발생한 해당 보드를 invalidate 시키는 것으로 다른 invalidate 동작들과는 차이를 가진다. 이와 같이, 각 프로세서에서 발생한 동작들은 cache 정보에 따라서, 동일한 동작일지라도 서로 다른 동작들을 실행하게 되며, 그 동작의 결과는 다시 directory 메모리에 변경된 정보로 저장되는 것이다. 표 3에서 N_S는 next state를 의미하며, 현재 발생된 동작의 결과로 변경되는 각 프로세서의 내부 cache에 저장된 line의 상태를 반영해서 directory 메모리에 저장되는 정보이다.

4.3 Coherence Controller

Coherence controller는 directory check controller, control signal generator, operation generator, next state generator, line fill controller, invalidate controller로 구성되어 있다. coherence controller는 arbitration unit으로부터 directory check cycle의 시작 신호를 받아서 directory check cycle을 실행한다. 프로세서 보드로부터 전송된 어드레스를 가지고 해당 데이터의 cache 정보를 directory 메모리로부터 가져오고, 이것을 이용해서, next state generator는 동작에 의해 변경되는 cache 정보를, line fill controller는 cache line fill에 필요한 신호들을, invalidate controller는 snoop cycle에 필요한 동작 신호들을 생성하게 된다. 프로세서 보드는 directory check cycle을 통해서 생성된 신호들을 이용해서 정상동작을 실행하게 된다. 변화된 cache정보는 다시 directory 메모리에 저장된다.

P_S		Operation	N_S	
ST0 : I I	①	B0_R	I	E
	②	B0_W	I	I
	③	B1_R	S0	I
	④	B1_W	I	I
ST1 : I E	⑤	B1_R	I	E
	⑥	B1_W	I	I
ST2 : S0 I	⑦	B0_R	S0	S0
	⑧	B0_W	I	I
	⑨	B1_W	S0	I
ST3 : S0 S0	⑩	B0_W	I	S1
	⑪	B1_W	S0	I
ST4 : I S1	⑫	B0_W	I	I
	⑬	B1_R	S0	S1
	⑭	B1_W	I	I
ST5 : S0 S1	⑮	B0_W	I	S1
	⑯	B1_W	S0	I

표 3 Cache 정보의 상태 변화

5. 결론

Directory based cache coherence 방법을 실제로 구체적인 시스템에 적용하기 위해서, Intel486 메모리 공유 시스템을 대상으로, Intel486 프로세서의 내부 cache의 동작을 분석하였고, arbitration unit, coherence controller, directory 메모리 등으로 구성된 central directory unit을 설계하고, 시뮬레이션을 통해서 그 동작을 검증하였다. 여러 보드에서 발생한 메모리 접근 요구들의 중재, cache 정보를 directory에 저장하는 방법, cache coherence를 위해 필요한 신호를 만드는 방법과 같은, directory based 방법이 병렬시스템에 적용되기 위해서 고려되어야 할 문제점들에 대한 해답을 제시하였고, 그러한 방법들을 디지털 회로로 구현하는 것을 보여주었다.

본 논문은 directory based cache coherence 방법을 Intel486 병렬시스템에 적용하기 위한 방법을 제시하고, 설계하는데 목적을 두었으며, 실제로 시스템에 설치하여 동작해 봄으로써 설계의 문제점을 파악하고 해결해야 할 것이며, 다양한 응용프로그램을 동작해 봄으로써, 응용프로그램의 특성에 따른 시스템의 성능 변화에 대한 분석과 연구가 필요하다고 할 수 있다.

[참 고 문 헌]

[1] "Embedded Intel486 Processor Family Developer's Manual", 1997, Intel

[2] "Embedded Write-Back Enhanced IntelDX4 Processor" ,1997, Intel

[3] Gergory F. Pfister, "In Search of Clusters", 1998 Prentice Hall

[4] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal, "Directory-Based Cache-Coherence in Large-Scale Multiprocessors", IEEE Computer, June 1990.

[5] M. Papamarcos and J. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories", In Proceedings of the 11th ISCA, pages 348--354, 1984.