Journal of The Korea Society of Computer and Information
Vol. 18, No. 10, October 2013
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2013.18.10.013

칩 멀티 프로세서 구조에서 온칩 유휴 캐시의 효과적인 활용 방안

곽 종 욱*

Efficient On-Chip Idle Cache Utilization Technique in Chip Multi-Processor Architecture

Jong Wook Kwak*

요 약

최근 들어 칩 멀티 프로세서 상의 코어 개수는 지속적으로 증가하는데 반해, 이를 효율적으로 뒷받침하기 위한 멀티 프로그래밍 혹은 멀티 쓰레딩 기법은 부족한 실정이다. 이로 인해 실제 작업을 수행하지 않는 유휴 코어가 발생하였고, 해당 코어가 소유한 자원들 중 개별 캐시 부분은 유휴 캐시로 낭비되었다. 본 논문에서는 유휴 개별 캐시의 발생이 불가피함을 인지함과 동시에 그것을 칩 내 메모리 공간으로써 효율적으로 활용할 수 있는 기법을 제안한다. 제안된 기법은 유휴 캐시를 희생 캐시로 활용하는 방법이며, 이를 위해 요구되는 새로운 시스템 구성 및 캐시 일관성 프로토콜의 세부 동작을 소개한다. 본 논문에서 제시된 기법은 유휴 캐시를 사용하지 않을 때와 비교하여 4-코어 및 16-코어 기반 칩 멀티 프로세서 환경에서 각각 19.4%와 10.2%의 IPC 향상을 가져왔다.

▸ Keywords : 칩 멀티 프로세서, 개별 캐시, 희생 캐시, 유휴 캐시, NUCA

Abstract

Recently, although the number of cores on a chip multi-processor increases, multi-programming or multi-threaded programming techniques to utilize the whole cores are still insufficient. Therefore, there inevitably exist some idle cores which are not working. This results in a waste of the caches, so-called idle caches which are dedicated to those idle cores. In this research, we propose a methodology to exploit idle caches effectively as victim caches of on-chip memory resource. In simulation results, we have achieved 19.4% and 10.2% IPC improvement in 4-core and 16-core respectively, compared to previous technique.

▸ Keywords : Chip Multi-processor, Private cache, Victim cache, Idle Cache, NUCA

•제1저자 : 곽종욱 •교신저자 : 곽종욱
•투고일 : 2013. 6. 4, 심사일 : 2013. 6. 25, 게재확정일 : 2013. 7. 29.
* 영남대학교 컴퓨터공학과(Dept. of Computer Engineering, Yeungnam University)

I. 서 론

전통적으로 프로세서에 관한 연구는 단일 프로세서의 성능을 극대화 시키는 방향으로 이루어져 왔다. 하지만 꾸준한 반도체 공정 기법의 향상을 통한 칩 트랜지스터 집적도의 증가와 비교적 정체되어 있는 마이크로 프로세서 설계 능력으로 인해 동일한 설계를 복제시키는 디자인 기법의 도입이 불가피해졌다. 그 결과, 최근의 마이크로 프로세서 개발은 단일 칩 내에 다수의 동일한 프로세싱 코어를 탑재하는 칩 멀티 프로세서(CMP, Chip Multi-Processor) 구조로 이루어 왔다. 하지만 이와 같은 프로세싱 코어 수의 증가가 직접적인 성능 향상으로 이어지는 것은 아니다. Intel社에 의하면, 단순히 코어 수의 증가로 얻을 수 있는 성능 향상은 비교적 적은 편이다. 또한 코어 수가 16개를 넘어가는 경우에는, 적절한 하드웨어 스케줄링이나 캐시의 효율적 활용이 지원되지 않을 경우, 오히려 코어 자체의 성능은 감소될 것으로 예상한다. 따라서 다수의 코어를 충분히 활용할 수 있는 멀티프로그래밍, 하드웨어 멀티쓰레딩, 캐시 관리 기법 및 명령어 집합 등의 역할이 점차 중요해지고 있다[1]. 이와 같은 성능 향상의 요소들 가운데 캐시 향상 기법은 다른 인자들과 비교하여 향후 프로세서의 성능 향상에 있어서 가장 큰 역할을 수행 할 것이라 해당 보고서는 기대된다. 따라서 본 논문에서는 칩 멀티 프로세서상의 캐시를 보다 효율적으로 활용하여 프로세서의 성능을 향상 시킬 수 있는 기법을 제안한다.

한편, 프로세서 디자인의 복잡도를 명세한 폴락의 법칙(Pollack's rule)에 따르면 프로세서의 성능 향상은 마이크로 아키텍쳐의 구현 복잡도 증가의 제곱근에 비례하여 이루어진다[2]. 이는 동일한 복잡도 하에서 하나의 큰 프로세서보다는 두 개의 작은 프로세서가 전체 성능 측면에서 약 40% 정도 뛰어남을 의미하며, 다수의 코어를 활용하는 디자인에 대한 효용성을 효과적으로 입증해 준다. 하지만 멀티프로그래밍 혹은 멀티쓰레딩 기법의 부족으로 인해 칩 멀티 프로세서상에 존재하는 다수개의 코어를 100% 활용하기에는 한계가 있다. 다시 말해, 코어 수가 많아지면 많아질수록 코어들 가운데 프로세서 스케줄링 정책에 의해 사용되지 않는 코어가 일시적으로 발생될 수 있으며, 그리고 이러한 유휴 코어(idle core)가 소유하고 있는 자원 또한 사용되지 않을 것이다. 이는 자원 활용 측면에서 낭비임이 분명하다. 일반적으로 멀티 코어의 프로세스 스케줄링은 각 코어에 CPU 사용률의 경계값을 두어 그 이상일 때 추가적으로 코어를 사용하게 된다[3].

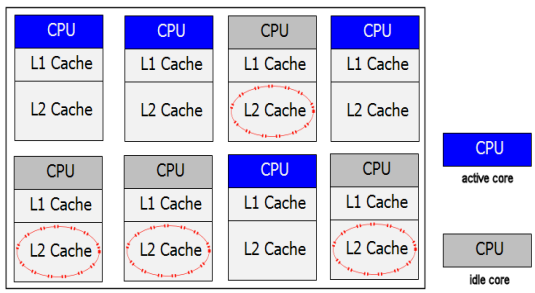

그림 1. 유휴 코어와 캐시
Fig. 1. Idle Core and Cache

한편, 그림 1은 총 8개의 코어를 가진 칩 멀티 프로세서 작업 환경을 예시적으로 보여준다. 이때 4개의 CPU 코어는 업무를 할당 받아 작업을 수행 중이지만, 나머지 코어들은 아무런 작업을 수행하지 않고 있다. 본 논문에서는 전자를 동작 코어(active core), 후자를 유휴 코어(idle core)라 칭한다. 유휴 코어의 경우 해당 코어가 소유한 캐시 자원들 또한 낭비되고 있다. 본 논문에서는 이 점에 착안하여 유휴 코어가 소유한 캐시, 즉 유휴 캐시(idle cache)를 적극적으로 활용함으로써 프로세서의 성능 향상을 꾀할 수 있는 기법을 제안한다.

II. 배경 지식 및 관련 연구

전통적인 캐시 구성은 각각의 프로세서가 L1(Level-1) 데이터 캐시와 L1 명령어 캐시를 개별적으로 소유하며 L2(Level-2) 캐시는 개별 혹은 공유 캐시를 사용하는 형태였다. 따라서 칩 멀티 프로세서 환경에서도 L1 데이터 캐시와 명령어 캐시를 개별 캐시 형태로 사용하는 데에는 별다른 이의가 없다. 하지만 L2 혹은 L3 (Level-3) 캐시를 어떠한 형태로 구성할 것인지에 관한 문제는 많은 논의를 낳았으며, 이와 관련된 개별 캐시와 공유 캐시의 장단점은 익히 알려진 바다. 최근에는 개별 캐시의 빠른 접근 시간과 공유 캐시의 높은 적중률의 장점을 동시에 제공할 수 있는 복합적인 형태의 캐시들이 제안되었다. Beckmann et al.는 L2 캐시를 다수의 뱅크형 공유 캐시 형태로 구성하여 모든 프로세서에게 공유시켰다[4]. Chishti et al.의 연구에서는 뱅크 형 공유 캐시 환경에서 캐시 블록의 태그와 데이터 영역을 분리하여 캐시 간 전송과 이동을 통해 불필요한 사본과 통신 부하를 줄이는 기법을 제안하였다[5]. 한편, Jichuan et al.는 개별적인 L2 캐시를 구성한 뒤, 캐시들 간의 협력을 통해 공유 캐시와 같은 효과를 나타내는 방법을 제안하였다[6].

한편, 반도체 집적도의 향상에 따라 기존의 단일 캐시 구조는 긴 지연 시간으로 인해 점차 비효율적이게 되었다. 이로 인해 다수의 프로세서 즉, 다양한 칩 내 위치 특성들을 가진 칩 멀티 프로세서 환경에서는, 더 이상 균등 캐시 구조(UCA, Uniform Cache Architecture)가 아닌 비 균등 캐시 구조(UNCA, Non-Uniform Cache Architecture의 도입이 제안되었다[7]. NUCA 구조하에서 이루어진 다양한 캐시 환경의 개선과 관련된 연구 가운데 Huh et al.은 다수의 뱅크 형태로 구성된 캐시 환경에서 공유 정도를 조절하여 특정 개수의 캐시 뱅크들을 묶어 하나의 공유 캐시로 사용 하는 기법을 제안하였다[8]. Dybdahl et al.은 지역적으로 고르게 분포되어 있는 뱅크형 L3 캐시에 대해 가까운 곳에 위치한 프로세서에게 소유권을 할당하고 개별 캐시로 사용하도록 하며, 동시에 캐시의 일부분을 공유 캐시형태로 사용하는 방법을 제안하였다[9]. 한편, Zhang et al.은 공유 캐시 블록에 대해 요청 프로세서로부터의 공간적 인접성을 향상시키고자 해당 블록을 복제시켜 가까운 위치로 이동시키는 방식을 제안하였다[10].

III. 칩 멀티 프로세서의 유휴 캐시 활용

1. 칩 멀티 프로세서 모델

본 논문의 대상 환경은 하나의 완전한 프로세서 유닛을 복제시킨 형태의 칩 멀티 프로세서이다. 기본 타일 유닛은 CPU 코어 부분과 함께 L1 캐시 그리고 메모리 접근 장치를 가진다. 각 프로세서의 캐시는 공유 버스에 연결되어 타 프로세서의 캐시와 외부 버스 접근 장치에 접근 할 수 있도록 한다. 본 논문에서 기본적으로 다루고자 하는 대상 플랫폼은 크게 두 가지로, 하나는 4-코어 모델이며, 다른 하나는 이 보다 더 고성능 환경을 지원할 수 있는 16-코어 모델이다.

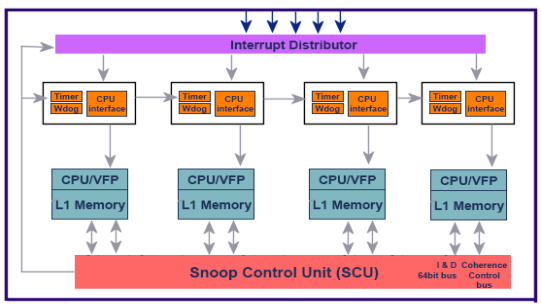

그림 2. ARM11 MPCore 구조
Fig. 2. ARM11 MPCore Architecture

데스크탑 환경에 비해 비교적 적은 메모리 구조를 가지고 있는 내장형 프로세서에서의 캐시 성능은 그것이 전체 성능에 미치는 영향이 매우 크다. 본 연구에서는 그림 2에서 소개된 ARM11 MPCore 모델을 기반으로 하여 대상 환경을 설정하였으며, 이의 상세는 다음과 같다.

(1) 칩 내부의 프로세서는 총 4개의 코어로 구성되며, 각각의 프로세서는 동일한 구성을 가진다.
(2) 각 프로세서의 명령어 수행을 담당하는 CPU/VFP 부분은 ARMv5 코어를 사용한다.
(3) 프로세서의 메모리 부분은 L1 캐시로 구성한다.
(4) 프로세서 내부의 자체적인 구조와 캐시의 동작은 기존 방식을 따른다.
(5) 각 프로세서의 L1 캐시는 스눕 버스를 통해 외부 메모리 혹은 다른 프로세서의 L1 캐시에 접근할 수 있다.

한편, 16개의 코어를 활용하는 16-코어 모델의 기본 구조는 그림 3과 같다. 이는 AMD社의 Opteron Quad 코어의 구조를 기반으로 하여, 이를 16개의 코어를 가진 타일형 칩 멀티 프로세서 형태로 확장시킨 것이다. 그림 3에 소개되어 있는 구조는 총 16개의 타일 프로세서로 이루어져 있으며, 주어진 모델의 상세는 다음과 같다.

(1) 칩 멀티 프로세서는 총 16개의 코어로 구성되어 있으며 각각의 코어는 동일한 구성을 가진다.
(2) 각 프로세서의 주 명령어 수행을 담당하는 CPU/VFP 부분은 ARMv5 코어를 사용한다.
(3) 메모리 부분은 L1 캐시와 L2 캐시로 구성한다.
(4) 프로세서 내부의 자체적인 구조와 캐시의 동작은 기본 방식을 따른다.
(5) 프로세서는 4개가 하나의 세트로 구성되며 이 세트들 사이는 하위 공유 버스를 통해 연결되어 있다. 하위 공유 버스는 버스 접근 장치를 통해 상위 공유 버스에 접근한다.
(6) 버스 인터페이스는 상위 공유 버스를 통해 4개의 하위 공유 버스로부터의 요청을 수집하여 외부 메모리로 전달한다.
(7) 각 프로세서의 L2 캐시는 하위 공유 버스를 통해 외부 메모리 혹은 다른 프로세서의 L2 캐시에 접근할 수 있다.

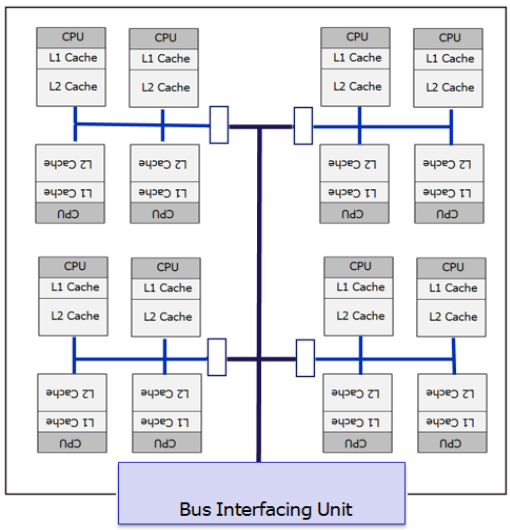

그림 3. 16-코어 칩 멀티 프로세서 모델
Fig. 3. 16-Core CMP Model

2. 희생 캐시로의 유휴 캐시 활용

칩 멀티 프로세서 내의 프로세서 수가 많을수록 유휴 코어의 발생 확률은 높아진다. 이는 유휴 캐시의 발생으로 이어진다. 그림 4를 살펴보면 현재 core1과 core3는 작업을 할당 받아 수행중인 반면 core2와 core4는 그렇지 못하다. 한편, N. Jouppi에 의해 제안된 희생 캐시(Victim Cache)는 캐시와 메인 메모리 사이에 위치하여 캐시로부터 접근 실패에 의해 방출되는 캐시 블록을 저장한 뒤, 차후에 프로세서로부터 해당 블록의 접근 요청이 있을 때 데이터를 공급하는 역할을 한다[11]. 이는 또 다른 캐시 공간을 제공함으로써 프로세서의 성능향상에 직접적으로 기여하게 된다.

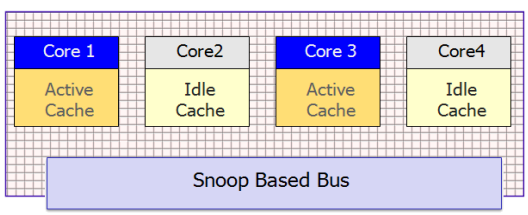

그림 4. 유휴 캐시의 발생
Fig. 4. Occurrence of Idle Caches

그림 5를 살펴보면, core 1은 업무를 수행 중이며 core 2는 작업을 수행하지 않고 있다. 이와 같은 상황에서 core 2의 유휴 캐시는 인접한 core 1에 연관되어 이의 희생 캐시 역할을 수행하게 된다. Core 2의 유휴 캐시는 core 1과 공유한 버스를 통해 core 1과 주 메모리 사이에 발생하는 이벤트를 모두 엿들을 수 있으며, 이를 통해 각 상황에 맞는 적절한 작업을 수행한다. 이에 대한 구체적인 세부 동작은 다음과 같다. 이상의 가. 와 나. 두 가지 작업을 수행하면서 유휴 캐시는 희생 캐시로서의 역할을 수행하게 되며, 이를 통하여 외부 메모리로의 접근 횟수를 줄이고 동시에 연관 코어의 성능 향상에 기여하게 된다.

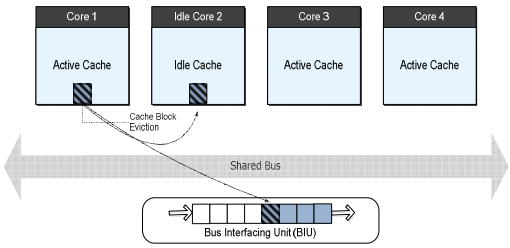

그림 5. 희생 캐시로의 유휴 캐시활용
Fig. 5. Idle Cache Exploration as Victim Cache

가. Core 1에서 캐시 블록 A 방출 시 Core 2 유휴 캐시 작업

i. 유휴 캐시는 공유 버스 상에서 자신에게 연관된 core 1의 캐시 블록 방출 작업을 확인하고, 방출되는 캐시 블록 A의 주소와 데이터를 가져온다.

ii. i.에서 얻은 주소로 자신의 캐시를 탐색하여 성공하면 iii.로 진행하고, 실패하면 iv.로 진행한다.

iii. 자신의 해당 캐시의 데이터를 캐시 블록 A로 갱신한 뒤, v. 로 진행한다.

iv. 해당 캐시 블록을 자신의 캐시에 삽입하며, 동시에 방출되는 캐시 블록의 쓰기 작업을 버스 접근 장치에 요청한다.

v. 버스 접근 장치에 해당 캐시 블록의 쓰기 작업을 취소하는 요청을 보낸다.

나. Core 1에서 캐시 블록 B 요청 시 Core 2 유휴 캐시 작업

i. 유휴 캐시는 공유 버스 상에서 자신에게 연관된 core 1의 캐시 블록 요청 작업을 확인하고, 캐시 블록 B의 주소를 가져온다.

ii. i.에서 얻은 주소로 자신의 캐시를 탐색하여 성공하면 iii.로 진행하고, 실패하면 종료한다.

iii. ii.에서 찾은 캐시 블록의 데이터를 공유 버스를 통해 core 1에게 전달해 주며, 동시에 버스 접근 장치의 버퍼에 저장된 core 1의 B 요청 작업을 취소한다.

3. 구현 및 세부 동작

3.1 코어의 구현

코어의 입장에서는 업무가 할당 되었는지 여부를 확인할 수 있는 정보만을 추가로 필요로 한다. 이는 상태를 나타내는 하나의 비트만으로 표시할 수 있다. 또한, 캐시가 유휴 캐시 상태로 변경될 때에 유휴 캐시가 보조하게 될 동작중인 코어에 대한 식별 정보가 있어야 한다. 이는 연관 코어의 식별자(associated id)를 스케줄러로부터 할당 받아, 자신이 유휴 상태로 변경되었을 때 해당 식별자를 가진 코어의 공유 버스 접근을 관찰하게 된다. 그림 6에 이상에서 소개된 기능을 지원하는 새로운 형태의 코어 구조가 소개되어 있다.

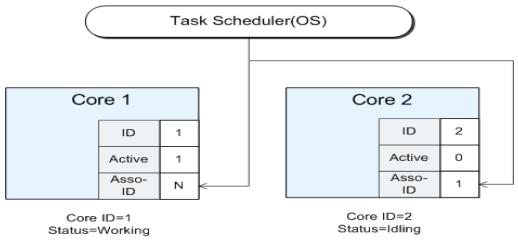

그림 6. 스케줄러를 통한 유휴 코어의 연관 식별자 할당
Fig. 6. Asso-ID Allocation by Task Scheduler

3.2 캐시의 구현

유휴 캐시가 희생 캐시로의 동작을 수행하기 위해 다음과 같은 기능을 수행하는 장치들이 필요하다. 우선, 자신의 core 수행 여부에 따른 캐시 수행 모드 변경 기능이 요구된다. 즉, "프로세서 〈-〉 캐시" 간의 정보 교환 모드와 "연관 코어 〈-〉 캐시" 간의 정보 교환 모드의 구분이 필요하다. 그리고 자신으로부터의 메모리 요청과 공유 버스 상의 메모리 요청을 모두 처리 할 수 있는 형태의 캐시 입출력 장치가 필요하다.

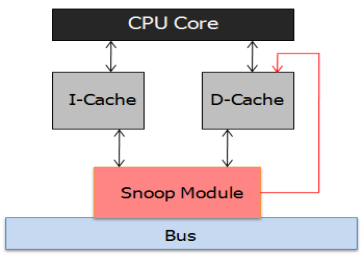

그림 7. CPU, 캐시, 스눕모듈 및 버스 구조
Fig. 7. CPU, Cache, Snoop Module and Bus Structure

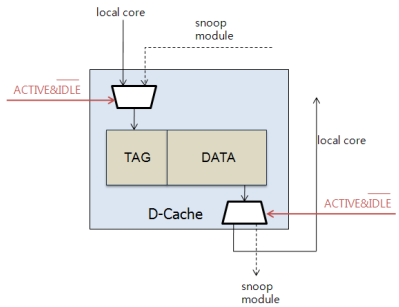

그림 8. 캐시 구조
Fig. 8. Cache Structure

이를 위해 코어의 상태를 나타내는 active, idle 비트를 이용하여 캐시 접근 요청의 소스를 로컬 코어와 스눕 모듈 중 하나로 선택할 수 있는 multiplexer를 사용한다. 또한, 해당 요청 주소로부터 얻어지는 데이터를 보내줄 최종 목적지를 선택할 수 있도록 de-multiplexer를 사용한다. 그림 7과 그림 8에 새로운 캐시의 구조가 소개되어 있다..

3.3 버스 접근 장치의 구현

코어 내부의 공유 버스 접근 장치, 즉 스눕 모듈(snoop module)은 다음과 같은 작업을 수행한다.

가. 코어가 ACTIVE & ~IDLE 모드 일 때

i. 명령어 캐시와 데이터 캐시의 요청을 버퍼에 저장한 뒤, 공유 버스가 사용 가능할 때에 해당 요청을 버스에 인가한다.

ii. 공유 버스를 스누핑 하면서, 로컬 코어의 요청에 대한 응답이 오면 해당 데이터를 버퍼로 가져온 뒤, 명령어 혹은 데이터 캐시로 전달한다.

나. 코어가 ~ACTIVE & IDLE 모드 일 때

i. 공유 버스를 스누핑 하면서, 버스 상에 연관 코어의 식별자를 가진 코어의 요청이 나타나면 해당 요청을 버퍼로 가져온다.

ii. 버퍼를 체크하여, 연관 코어의 캐시 읽기 실패를 확인한 뒤 ii-1을 수행한다. 연관 코어의 캐시 블록 방출을 확인한 경우는 ii-2를 수행한다.

ii-1. 연관 코어의 읽기 실패 주소로 로컬 캐시를 탐색하여 성공하면 해당 데이터를 캐시로부터 받아와 버스를 통해 연관 코어에게 전달한다.

ii-2. 연관 코어로부터 방출 되는 캐시 블록을 로컬 캐시에 쓰며, 동시에 버스 접근 장치의 버퍼에 있는 연관 코어의 캐시 쓰기 작업을 취소한다.

ii-1의 작업은 버스 스눕 모듈의 기능이며 캐시가 일반 상태일 때 수행된다. 한편, 캐시가 유휴 상태일 때 이를 희생 캐시로 동작시키기 위한 ii-2의 작업은 스눕 모듈에 추가된 Victim Caching Engine (VCE)에 의해서 이뤄진다. 그림 9는 새롭게 구성된 스눕 모듈의 구조이다. 한편 외부 메모리로의 접근을 담당하는 외부 버스 접근 장치(External Bus Interface)는 다음과 같은 기능이 추가적으로 필요하다.

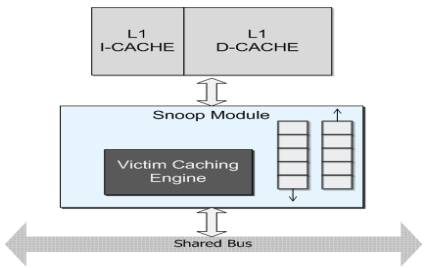

그림 9. 스눕 모듈 구조
Fig. 9. Snoop Module Structure

i. 내부 공유 버스 상의 읽기 혹은 쓰기 요청을 버퍼에 저장한 뒤, 외부 버스를 통해 외부 메모리로 요청한다.
ii. 외부 메모리로부터의 응답을 버퍼에 저장한 뒤, 내부 공유 버스를 통해 해당 코어로 전달한다.
iii. 내부 버스 상의 희생 캐시의 읽기 응답을 인지하면, 그에 대응하는 읽기 요청을 버퍼에서 검색하여 삭제한다.
iv. 내부 버스 상의 희생 캐시로부터의 쓰기 요청 취소를 인지하면, 해당 요청을 버퍼에서 검색하여 삭제한다.

4. CMP 모델에서의 적용

이상에서 소개된 내용을 4-코어 칩 멀티 프로세서에 활용한 내용이 그림 10과 같다. 이 적용에서는 유휴 코어의 L1 캐시를 인접한 동작 코어의 희생 캐시로 사용한다. 이 때 동작 코어 및 연관 코어의 할당은 스케줄러에 의해 이루어진다. 그림 10에서 core 1과 core 3은 동작 코어를, core 2와 core 4는 유휴 코어를 예시적으로 나타낸다. 이때에 core 2의 캐시는 core 1을 연관 코어로 할당 받고, core 4의 캐시는 core 3을 연관 코어로 할당 받아 희생 캐시의 역할을 수행한다. 한편, 16-코어 칩 멀티 프로세서에서의 유휴 캐시 활용 기법의 적용은 그림 11과 같다. 이 적용에서는 유휴 코어의 L2 캐시를 인접한 동작 코어의 희생 캐시로 사용한다. 이 때 동작 코어 및 연관 코어의 할당은 스케줄러에 의해 이루어진다. 그림 11에서 core 1과 core 3은 동작 코어를, core 2와 core 4는 유휴 코어를 예시적으로 나타낸다. 이때에 core 2의 L2 캐시는 core 1을 연관 코어로 할당 받고, core 4의 L2 캐시는 core 3을 연관 코어로 할당 받아 희생 캐시로의 역할을 수행한다. 동작 코어의 메모리 접근 요청은 상위 버스 접근 장치와 외부 버스 접근 장치 두 곳에서 버퍼링 된다. 따라서 희생 캐시 적중에 의한 기존 요청 취소 작업은 상위 버스 접근 장치에서 처리를 시도한 뒤, 실패하면 외부 버스 접근 장치에서 다시 시도한다.

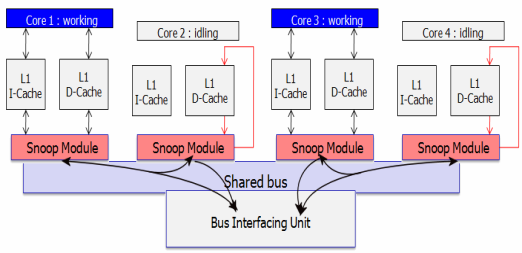

그림 10. 4-코어 모델에서의 유휴 캐시 활용기법 적용
Fig. 10. 4-Core Model with Idle Cache Exploration

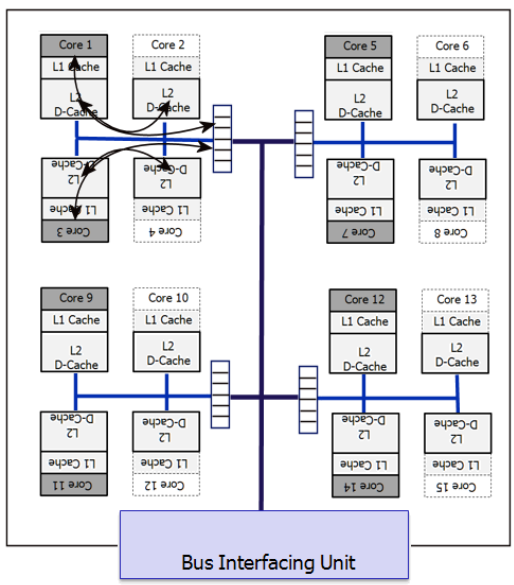

그림 11. 16-코어 모델에서의 유휴 캐시 활용기법 적용
Fig. 11. 16-Core Model with Idle Cache Exploration

IV. 모의실험 및 성능 분석

본 논문에서 제안된 유휴 캐시의 활용에 대한 성능 평가를 위해 모의실험을 수행하였다. 실험 환경으로는 ARMv5에 기반한 Simit-ARM[12]을 시뮬레이터로 사용하고, SystemC를 이용하여 캐시와 버스 및 기타 장치들을 모델링하였다. 표

1은 본 실험에서 사용한 4-코어 모델과 16-코어 모델 각각의 상세 설정 인자를 보여준다. 본 논문의 제안 기법을 평가하기 위해 Mibench의 벤치마크와 SPEC의 벤치마크를 사용하였다. 사용한 벤치마크의 상세 구성은 표 2와 같다.

표 1. 4-코어 및 16-코어 모의실험 환경
Table 1. 4-Core & 16-Core Simulation Environment

4-코어 설정 요소	값
프로세서	ARMv5
블록 크기	32 bytes
L1 명령어 캐시	16 KB, 32-way, 2-cycle
L1 데이터 캐시	16 KB, 32-way, 2-cycle
외부 메모리 접근 지연	80 Cycles
유휴 캐시 접근 지연	8 Cycles

16-코어 설정 요소	값
프로세서	ARMv5
블록 크기	64 bytes
L1 명령어 캐시	16 KB, 32-way, 2-cycle
L1 데이터 캐시	16 KB, 32-way, 2-cycle
L2 명령어 캐시	512 KB, 32-way, 8-cycle
L2 데이터 캐시	512 KB, 32-way, 8-cycle
외부 메모리 접근 지연	250 Cycles

표 2. 벤치마크 구성
Table 2. Benchmark Composition

구분	벤치마크
4-코어용 벤치마크 조합	Mibench(basicmath+ quicksort)
	Mibench(jpeg+dijikstra)
	Mibench(patrica+rsynth)
	Mibench(stringsearch+blowfish)
	Mibench(rijindael+sha)
16-코어용 벤치마크 조합	Mibench(basicmath+quicksort+jpeg+dijikstra+patrica) SPEC2000(twolf+bzip2+vortex)
	Mibench(rsynth+stringsearch+blowfish+rijindael+sha) SPEC2000(parser,mcf,gzip)

본 논문에서 제시된 방식은 코어와 캐시 부분에 있어서 사실상 최소한의 수정 및 오버헤드를 요구한다. 또한 추가적으로 요구되는 기억공간도 없으므로, 대부분 조합회로로 구현이 가능하다. 실행 시간 측면에서의 성능 향상을 살펴 보면 다음과 같다. 그림 12는 유휴 캐시의 활용에 따른 외부 메모리 접근 수의 비교이다. 기존의 방식을 1의 값으로 두고 각 응용 프로그램 별로의 상대적인 접근 감소를 나타내었다. 그림 12에서 보여 지듯이 유휴 캐시를 활용할 경우 외부 메모리로의 접근이 최대 95%, 평균 65% 감소하였다. 위의 메모리 접근 감소는 코어의 읽기, 쓰기 요청을 모두 포함한 것이다.

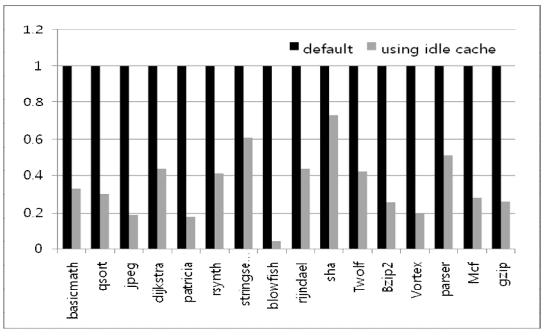

그림 12. 유휴 캐시 활용 전후의 외부 메모리 접근 비교
Fig. 12. The number of Off-Chip Misses

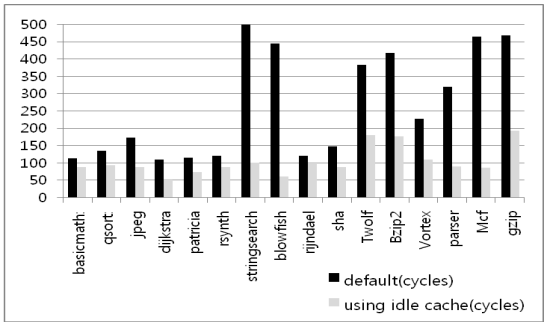

그림 13. 데이터 캐시 블록의 응답 시간 비교
Fig. 13. Response Time for Data Cache Block

한편, 실질적으로 코어의 성능에 영향을 주는 것은 데이터 요청에 대한 응답 시간이다. 이와 같은 응답 시간은 캐시의 적중 시간, 적중 실패율, 적중 실패 비용까지 모두 포함한다. 이에 대한 실험 결과가 그림 13에 제시되어 있다. 응답 시간은 최대 98%, 평균 52% 단축되었다. 또한 그림 12에서처럼, 유휴 캐시의 활용이 외부 메모리로의 접근 횟수를 줄임으로써 명령어 인출의 지연 시간 또한 줄일 수 있어 이로 인한 추가적인 성능 향상이 기대된다. 이 두 가지로 인한 각 벤치마크의 최종적인 IPC 향상은 그림 14와 같다. 평균적으로 19.4%, 최대 63%의 IPC 향상을 보였다. 특히, 비교적 큰 working set을 가진 벤치마크의 성능 향상이 더 크다. 이와 같은 원인을 구체적으로 파악하기 위해 그림 15와 표 3의 결과를 통해 추가적으로 분석해 보면 다음과 같다. 그림 15는 각 벤치마크별 L1 캐시의 적중률이다. 성능이 10% 이상 향상된 벤치마크에 있어서 L1 캐시의 적중률을 살펴보면, 캐시의 적중률이 낮은 벤치마크일수록 유휴 캐시의 활용에 의한 성능 향상이 크다는 것을 알 수 있다. 이는 메모리로의 접근 예측이 어려운 응용 프로그램일수록 유휴 캐시의 활용으로 인해 상대적으로 더 큰 성능 향상을 얻을 수 있다는 것을 의미한다.

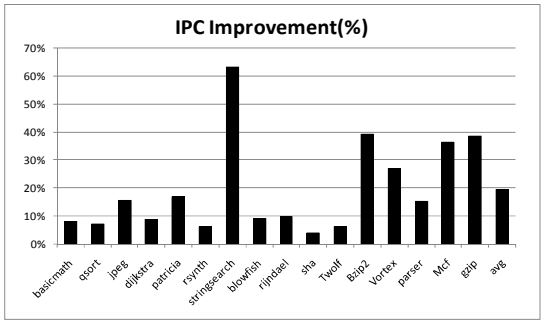

그림 14. IPC 성능 향상 (4-코어)
Fig. 14. IPC Improvement (4-Core)

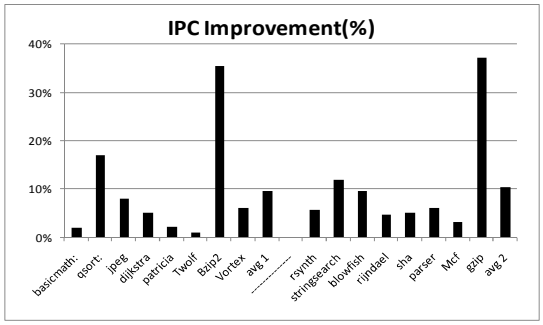

그림 16. IPC 성능 향상 (16-코어)
Fig. 16. IPC Improvement (16-Core)

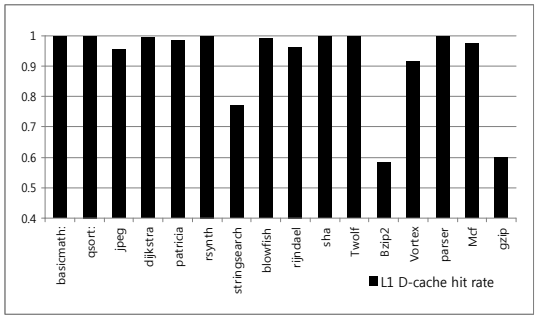

그림 15. L1 데이터 캐시 적중율
Fig. 15. L1 Data Cache Hit Ratio

다음으로 오직 하나의 코어만이 작업을 수행하는 상황을 설정하여 유휴 캐시 기법 자체만의 성능 향상을 살펴본다. 표 3은 하나의 코어가 하나의 유휴 캐시를 희생 캐시로 사용할 때의 캐시 적중률이며, 이와 연관된 IPC의 향상이다. 설명의 편의를 위해 실험 결과에 따라 이를 전형적인 두 그룹으로 분리하였다. 표 3에서 주어진 결과를 살펴보면, 희생 캐시의 적중률과 성능 향상과의 직접적인 상관관계를 확인할 수 있다. 또한 분석 결과 본 논문에서 제안된 기법은 유휴 캐시에서의 캐시 적중 가운데 읽기 적중 비율이 쓰기 적중에 비해 상대적으로 많은 부분을 차지하는 벤치마크에서 보다 더 큰 성능 향상을 확인할 수 있다.

표 3. 희생 캐시 적중률과 성능 향상의 관계 (16-core)
Table 3. Victim Cache Hit Ratio vs. IPC Improvement (16-core)

벤치마크	적중률	IPC (%)	특징
qsort	38.2%	18.4	read hit 〉 write hit
bzip2	31.5%	36.2	
gzip	30.7%	38	
parser	28.3%	7.4	read hit 〈 write hit
mcf	47.1%	4.8	
vortex	37.0%	6.9	

끝으로, 그림 16은 16-코어 모델의 일반적인 실행 환경하에서 IPC 향상에 대한 실험 결과이다. 그림 16의 결과를 살펴보면, 전체적으로 메모리 요구량이 많고, 특히 외부 메모리 접근에 대한 요구치가 높은 벤치마크들(qsort, bzip2, vortex, parser, mcf, gzip)의 경우에 있어서 IPC 향상이 더욱 두드러짐을 확인 할 수 있다. 이처럼 본 실험에서는, 유휴 캐시를 활용하여 많은 외부 메모리 접근을 요구하는 소수 코어들의 메모리 접근을 내부 메모리 접근 요청으로 변경시킴으로 해당 코어 별로 최대 37%, 그리고 칩 멀티 프로세서 전체에서는 10.2%의 IPC 향상을 확인하였다.

V. 결 론

본 논문에서는 다수의 코어를 가진 칩 멀티 프로세서 환경에서, 유휴 캐시를 활용하여 칩 멀티 프로세서의 성능을 높일 수 있는 기법을 모색하였으며, 유휴 개별 캐시를 희생 캐시로 사용하는 방법을 제안하였다. 본 논문에서는 이를 위해 희생 캐시로의 유휴 캐시 활용 방식, 그리고 이에 대한 칩 멀티 프로세서 모델의 구현 및 세부 동작 환경, 그리고 희생 캐시로 유휴 캐시를 활용하기 위한 캐시 일관성 프로토콜을 소개하였다. 실험 결과 멀티 코어 환경에서의 유휴 캐시 활용이 현재 실행 중인 다른 코어의 외부 메모리 접근 횟수를 현저히 낮추며, 동시에 상당한 성능 향상을 가져올 수 있음을 확인하였다. 희생 캐시로의 유휴 캐시 활용 기법은 평균적으로 4-코어와 16-코어의 경우 각각 19.4%와 10.2%의 IPC 향상을 가져올 수 있음을 보였다.

참고문헌

[1] Intel Corporation. "TeraFlops Research Chip", 2007.

[2] S. Borkar, "Thousand Core Chips-A Technology Perspective", pp. 746-749, In Proc of DAC. 2007.
[3] D. Choffines, M. Astley and M. Ward "Migration policies for multi-core fair-share scheduling", ACM SIGOPS OS Review, Vol. 42, Iss. 1, 2008
[4] B. M. Beckmann et al., "ASR: Adaptive Selective Replication for CMP Caches", pp. 443-454, In Proc of MICRO 2006.
[5] Z. Chishti et al., "Optimizing replication, communication and capacity allocation in CMPs", 32nd ISCA, pp. 357-368, 2005.
[6] Jichuan, Chang et al., "Cooperative Caching for Chip Multiprocessors", pp. 264-276, ISCA. 2006.
[7] C. Kim, D. Burger, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches", pp. 211-222, ASPLOS-X. 2002.
[8] J. Huh, et al., "A NUCA Substrate for Flexible CMP Cache Sharing", Trans. Parallel Distrib. Syst, pp. 1028-1040, vol.18, no.8. 2007.
[9] H. Dybdahl et al., "An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors", pp. 2-12, In Proc of HPCA 2007.
[10] M., Zhang and K., Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled CMPs", pp. 336-345, In Proc of 32nd ISCA, 2005.
[11] N., Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", pp. 364-373, In Proc of ISCA, 1990.
[12] W. Qin and S.Malik, "Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation", pp. 10556, In Proc of DATE 2003.

저 자 소 개

곽 종 욱
1998: 경북대학교
컴퓨터공학과 공학사.
2002: 서울대학교
컴퓨터공학과 공학석사.
2006: 서울대학교
전기컴퓨터공학부 공학박사
현 재: 영남대학교
컴퓨터공학과 부교수
관심분야: 컴퓨터구조,
임베디드 시스템,
고성능 컴퓨팅
Email : kwak@yu.ac.kr