부분 압축 명령어를 위한 캐쉬 구조의 설계 및 평가
(Design and Evaluation of Cache Structure for Semi-packed Instruction)

홍 원 기 † 이 승 엽 †† 김 신 덕 †††
(Won-Kee Hong) (Seung-Yup Lee) (Shin-Dug Kim)

요 약 VLIW에서는 프로그램 코드를 병렬화 하는 작업이 모두 컴파일러에 의해서만 이루어진다. 따라서 병렬로 수행될 연산어들을 명시적으로 나타내 주어야 하며, 이를 위한 명령어 인코딩 방식으로 전개 인코딩 방식과 압축 인코딩 방식이 사용되어 왔다. 각 인코딩 방식들은 명령어의 적재 및 검색을 위해 서로 다른 캐쉬 구조를 필요로 하는데, 전개 인코딩 방식으로 비압축 캐쉬를 압축 인코딩 방식으로 압축 캐쉬를 사용하고 있다. 그러나 이들은 각각 무효 연산어로 인한 메모리 활용 효율 저하와 복원 과정으로 인한 명령어 인출 오버헤드의 증가라는 문제점을 안고 있다. 본 논문에서는 부분적으로 명령어 길이를 일정하게 유지하는 부분 압축 인코딩을 사용해 메모리 활용 효율을 높이는 동시에 명령어 인출 오버헤드를 줄일 수 있는 분할 캐쉬 구조를 제안한다. 각 캐쉬 구조를 구현하는데 필요한 칩 영역을 계산하여, 분할 캐쉬가 비교적 비용 효율적인 캐쉬 구조임을 확인하였다. 모의 실험을 통한 메모리 활용 효율 측정 결과 하드웨어 비용의 증가를 고려하더라도 분할 캐쉬는 비압축 캐쉬에 비해 최고 약 3배의 메모리 활용 효율을 얻을 수 있었다. 각 캐쉬 구조를 일차 캐쉬로 하는 VLIW 시스템들의 성능 측정 결과는 TCSC(블록 집중형 분할 캐쉬)를 사용한 시스템이 비용 대비 성능 면에서 가장 우수한 것으로 나타났다.

Abstract In VLIW architectures, the optimization and scheduling process for parallelizing program codes totally rely on an ILP compiler. The static scheduling requires that an instruction, which is a collection of operations in parallel, must be encoded with an explicit specification of parallel operations. Up to now, two instruction encoding schemes have been used to separate instructions from one another, which are the unpacked encoding scheme for the decompressed cache and the packed encoding scheme for the compressed cache. However, these caches have an adverse influence on the VLIW memory subsystem in the aspects of the memory utilization and instruction fetch. In this paper, a new cache structure, called the section cache is proposed in order to enhance memory utilization with reduced instruction fetch overhead by partially fixing the length of an instruction. The computation of on-chip cache area reveals that the section cache is comparatively a cost-effective cache structure. The simulation results confirm that the memory utilization of the section cache is three times higher than that of the decompressed cache, even though the increased cost of the section cache is considered. Consequently, the VLIW systems with the TCSC(Tightly Coupled Section Cache) as a first-level cache can achieve the better performance/cost than any other systems.

1. 서 론

집적 회로 기술의 발전으로 3D 그래픽, 화상 회의, 영상 압축, 음성 인식, 애니메이션 등 다양한 멀티미디어 응용 프로그램들의 수요가 최근 급성장하고 있는 추세이다. 이들 멀티미디어 응용 프로그램들은 주로 실시간에 많은 양의 미디어 데이터 처리를 요구하기 때문에 고성능의 미디어 프로세서를 요구한다[1]. 미디어 프로세서들은 멀티미디어 응용 프로그램들이 가지고 있는

* 본 논문은 정보통신부에서 지원하는 대학기초 연구지원 사업으로 수행된 과제임.
† 비 회 원 : 연세대학교 컴퓨터학과
wkhong@kurene.yonsei.ac.kr
†† 학생회원 : LG전자 DigitalTV 연구원
sylee@lge.com
††† 종신회원 : 연세대학교 기전공학부 정보산업전공 교수
sdkim@kurene.yonsei.ac.kr
논문접수 : 2000년 3월 16일
심사완료 : 2001년 3월 15일

많은 명령어 및 데이터 병렬성의 특징을 적극 활용할 수 있는 구조들, 즉, MMX, 벡터 프로세싱, SIMD, VLIW 등, 다양한 구조들을 채택하여 높은 성능을 얻고자 하였다. 특히, VLIW 구조는 비교적 간단한 하드웨어로 높은 명령어 병렬성을 지원해줄 수 있다는 점에서 최근 내장형 미디어 프로세서로써 각광을 받고 있다[2, 3, 4].

VLIW는 코드를 병렬화하고 스케쥴 하는 일련의 모든 작업들을 전적으로 병렬화 컴파일러에 의존하고 있다. 프로그램들은 통상 루프 전개(loop unrolling), 테일 복재(tail duplication), 프로시저 삽입(procedure inlining), 보상 코드(compensation code)의 삽입 등, 병렬성을 증진시키기 위한 컴파일 과정을 통해 변형되고 스케쥴 된다. 그러나 이러한 컴파일 기법들은 주로 코드 복제를 통해 이루어지기 때문에 코드 크기의 증가라는 문제점을 안고 있다. 뿐만 아니라, 함께 수행될 연산어들을 가려내기 위해 삽입되는 무효 연산어(No Operation, NOP)는 이러한 코드 크기 증가를 더욱 가속화시키는 역할을 한다. 기존 문헌에 따르면 무효 연산어의 비율이 전체 VLIW 코드 중에 약 50~80%를 차지하고 있는 것으로 나타나고 있다[5, 6, 7]. 이는 메모리 활용 효율을 떨어뜨려 전체적인 메모리 접근 지연 시간을 증가시키는 결과를 낳는다. 이러한 문제를 해결하기 위해 최근까지 개발되거나 제안된 VLIW 시스템들은 크게 두 가지 전략 중 하나를 사용하고 있다. 첫째는 명령어에서 무효 연산어를 완전히 제거하여 메모리에 저장하는 방법이다. 이것은 메모리 활용도면에 있어서 높은 성능을 얻을 수 있으나 명령어를 수행할 때마다 무효 연산어를 이용하여 명령어를 복구해야 하기 때문에 명령어 인출 오버헤드가 증가하는 문제점을 안고 있다. 특히, 이로 인해 파이프라인 분기 비용이 늘어나게 되는데 이것은 VLIW나 수퍼스칼라와 같이 한번에 여러 개의 연산어를 수행하는 구조에서는 상당한 성능 손실을 가져온다. 더욱이, 최근 VLSI 기술의 발전으로 온 칩 캐쉬의 크기가 증가했을 뿐만 아니라 다단계 온 칩 캐쉬가 가능해 짐으로써 이러한 문제는 더욱 심각해진다. 즉, 단순히 하위 메모리에 대한 접근 빈도를 줄여주기만 해서는 전체적인 시스템 성능을 올리기 어려워 졌는데, 이는 캐쉬 용량의 증가로 인해 캐쉬 접근 실패율이 줄어들고 다단계 캐쉬로 인해 접근 실패 비용이 줄어들면서, 명령어 인출 오버헤드가 시스템 성능에 미치는 영향이 상대적으로 늘어났기 때문이다. 무효 연산어의 영향을 약화시키기 위한 또 다른 전략으로 메모리 계층 구조상의 메모리 단계에 따라 사용하는 명령어 인코딩 기법을 다르게 하는 방법을 들 수 있다. 즉, 하위 메모리에는 압축 명령어 코드들을 저장하는 반면 프로세서와 가까운 상위 메모리에는 명령어 복구가 필요 없는 비압축 명령어 코드들을 저장함으로써 명령어 인출 오버헤드를 줄여주며 캐쉬 접근 실패 비용도 어느 정도 줄여주려는 방법이다. 그러나 비압축 인코딩 기법 및 이를 위한 캐쉬 메모리 구조는 메모리 활용이 매우 떨어지기 때문에 크기가 비교적 작은 상위 메모리에서 사용하기에는 부적합하다.

본 논문에서는 메모리 계층 구조상의 메모리 단계에 따라 명령어 인코딩 기법을 달리하는 메모리 서브시스템을 위한 명령어 압축 인코딩 기법 및 캐쉬 구조를 설계한다. 상위 메모리에 사용될 명령어 압축 인코딩 기법으로써 적절한 메모리 활용도를 얻으면서 동시에 파이프라인 분기 비용을 줄여줄 수 있는 부분 전개 인코딩 기법(semi-packed encoding scheme)을 제안한다. 이것은 일정한 길이의 부분 명령어(sub-instruction)들로 구성하여 명령어 인출 오버헤드를 줄여 주고 있다. 또한, 부분 전개 명령어를 효과적으로 검색·저장하기 위하여 분할 캐쉬(section cache)라는 새로운 캐쉬 구조를 제안한다. 효과적인 캐쉬 설계를 위해서는 수행 성능과 함께 하드웨어 복잡도(hardware complexity)에 대한 고려가 있어야 한다. 이를 위하여 각 캐쉬 구조에서 요구하는 칩 영역을 계산하여 섹션 캐쉬의 하드웨어 복잡도를 살펴보았다. 섹션 캐쉬 및 이를 채택한 VLIW 시스템의 성능 측정을 위해 전개 인코딩 기법(unpacked encoding scheme)을 위한 비압축 캐쉬(decompressed cache)와 압축 인코딩 기법(packed encoding scheme)의 압축 캐쉬(compressed cache)를 비교 대상으로 하여 모의 실험을 하였다. 모의 실험을 위한 실험 도구로 Trimaran[8]을 사용하였으며 VLIW 구조는 PlayDoh 구조를 가정하고 있다. PlayDoh는 명령어 병렬성을 활용하기 위한 다양한 프로세서 구조 연구를 위해 1993년 휴렛 팩커드 연구소에서 발표된 구조이다[9]. 특히, 이것은 VLIW를 위한 다양한 기능들을 제공하고 있으며 현재 많은 연구, 사업단체에서 사용하는 대표적인 VLIW 구조로 알려져 있다. 실험 결과 분할 압축 인코딩 기법은 전개 인코딩 기법에 비해 코드 크기를 최고 $\frac{2}{5}$까지 줄일 수 있었으며 메모리 활용 측면에 있어서 분할 캐쉬는 비압축 캐쉬에 비해 최고 3.4배가 높은 활용 효율을 보여주었다. 또한 명령어 인출 오버헤드의 효과적인 제거로 인해 분할 캐쉬를 채택한 메모리 서브시스템은 다른 시스템에 비해 최고 15%의 성능 향상을 가져올 수 있음을 확인하였다.

본 논문의 구성은 다음과 같다. 제2장에서는 VLIW를 위한 명령어 인코딩 기법과 캐쉬 구조에 대한 기존 연

구들을 소개한다. 제3장에서는 분할 캐쉬 성능을 예측하기 위한 척도로 임계 접근 실패율을 소개하며 이를 통해 메모리 활용도와 명령어 인출 오버헤드 간의 관계를 살펴본다. 제4장에서는 본 논문에서 제안하는 부분 압축 인코딩과 분할 캐쉬 구조를 설명한다. 제5장에서는 각 캐쉬 구조의 하드웨어 복잡도를 계산하며 이를 통해 각 캐쉬 구조의 장단점을 살펴본다. 제6장에서는 각 캐쉬 구조들의 성능을 측정하기 위한 방법론 및 실험에 사용된 벤치마크 프로그램들의 특징을 살펴보며 실험결과를 비교・분석한다. 마지막으로 제7장에서 결론을 맺는다.

2. VLIW 명령어 인코딩과 캐쉬 구조

슈퍼스칼라 구조의 경우 병렬 수행 연산어 추출이 하드웨어에 의해 동적으로 수행되는 반면 VLIW 구조에서는 이러한 작업이 오직 컴파일러에 의해서만 이루어지기 때문에 명령어와 명령어를 구분해 줄 수 있는 새로운 명령어 인코딩 방식이 필요하다. 일반적으로 VLIW 구조에서 사용되고 있는 명령어 인코딩 방식은 무효 연산어의 삽입 여부에 따라 크게 압축 인코딩과 전개 인코딩 방식으로 나눌 수 있다. 압축 인코딩 방식은 메모리 공간의 효과적인 활용을 위하여 유효 연산어들만으로 명령어를 구성하고 있으나 동시에 수행될 연산어의 수가 일정하지 않으므로 명령어와 명령어를 구분할 수 있는 정보를 별도로 유지해야 한다. 또한 명령어 인출 시마다 해당 연산어들을 검색해야 하므로 인출 과정이 복잡해지는 문제점을 안고 있다. 반면에 전개 인코딩 방식은 유효 연산어의 수에 상관없이 모든 명령어에 일정한 수의 연산어 슬롯들을 할당하여 명령어를 구분하고 있다. 이것은 메모리로부터 명령어 인출을 간단하게 하지만 메모리 활용도가 떨어지는 단점이 있다.

이와 같이 VLIW 구조는 명령어 인코딩 방식에 따라 서로 다른 명령어 검색 및 인출 과정이 필요하고 슈퍼스칼라 구조와 달리 순차 명령어의 주소 계산을 위한 명령어 길이 정보가 요구되기 때문에 이를 지원할 수 있는 새로운 메모리 구조와 명령어 인출 방식이 필요하다. 본 장에서는 각 인코딩 명령어를 저장하기 위한 두 가지 대표적인 캐쉬 구조에 대해 설명한다.

2.1 압축 캐쉬

압축 캐쉬는 압축 명령어 —압축 인코딩 기법으로 인코딩 된 명령어— 를 저장・검색하기 위한 캐쉬 구조이다[10]. 압축 명령어는 그 길이가 명령어마다 서로 다르기 때문에 임의의 명령어가 두 개의 블록에 걸쳐 저장될 수 있다. 이런 경우 명령어를 읽어 오기 위해 두 번의 캐쉬 메모리 접근이 필요하며, 따라서 명령어 인출 시간이 늘어나게 된다. 이러한 문제를 해결하기 위해 압축 캐쉬는 두 개의 뱅크로 구성되어 있다. 이는 연속한 캐쉬 블록들을 서로 다른 캐쉬 뱅크에 번갈아 저장해두고 명령어 인출시 두 개의 뱅크를 동시에 접근하게 함으로써 해당 명령어를 한 번에 읽어올 수 있게 한다.

압축 캐쉬를 검색하기 위한 주소 기법으로 전통적인 직접 사상 캐쉬에서 사용되는 직접 사상 주소 기법(direct mapped addressing scheme)을 사용할 수 있다. 이를 위해 각 캐쉬 블록마다 주소 태그를 가지며, 캐쉬에 저장된 명령어를 구분하고 순차 명령어 주소를 계산하기 위해 연산어 단위로 명령어 길이 정보를 유지하고 있다. 그러나, 이것은 전체적인 저장 오버헤드를 증가시키는 하나의 요인이 된다.

VLIW 컴파일러는 메모리 연산어와 관련된 코드 수행 길이의 최적화를 위하여, 명령어 내의 위치에 따른 연산어들 간의 우선 순위를 보장함으로써 서로 종속관계에 있는 메모리 연산어들을 같은 명령어에 둘 수 있게 하고 있다[11]. 그러나 압축 캐쉬로부터 명령어를 인출하기 위해 각 캐쉬 뱅크로부터 읽어온 두 개의 블록을 순서에 맞춰 조정한 후 해당하는 연산어들을 추출해야 한다. 이러한 조절・추출 과정을 위해 명령어 파이프라인의 명령어 인출 단계와 디코드 단계 사이에 복원 단계를 추가하게 된다. 그러나 이것은 조건 분기 명령어로 인한 예측 오류 비용이 늘어나는 결과를 초래하게 되는데, 복원 단계가 없을 경우 조건 분기 명령어를 인출하고 나서 1사이클이 지나면 목적 명령어의 위치를 계산할 수 있지만, 복원 단계가 포함될 경우 2사이클이 지난 후에야 목적 명령어의 위치를 알 수 있게 되기 때문이다. 특히, VLIW 구조는 임의의 연산어에 예외 상황(exception)이 발생하였을 경우 명령어 전체가 지연되는 동기 수행 연산(lock-step operation)을 수행하기 때문에 이러한 파이프라인 분기 오류 비용은 더욱 심각한 성능 저하 원인이 되고 있다.

2.2 비압축 캐쉬

비압축 캐쉬는 전개 명령어를 적재하기 위한 캐쉬 구조로써 모든 명령어에 일정한 크기의 캐쉬 공간을 할당한다. 이 때, 캐쉬 폭을 전개 명령어 길이에 맞춰줌으로써 명령어 인출을 간단하게 하고 있다. 즉, 캐쉬 접근 시마다 블록 교환 및 연산어 선택과 같은 복원 과정이 필요 없기 때문에 파이프라인 분기 오류 비용을 줄여줄 수 있다. 그러나 명령어를 구성하는 유효 연산어의 수에 상관없이 일정한 캐쉬 공간을 할당하기 때문에 메모리 활용도가 떨어지는 문제점이 있다.

비압축 캐쉬는 압축 명령어를 저장하고 있는 하위 메

모리로부터 명령어를 읽어오기 위해 주소 모호성의 문제를 해결해 줄 수 있어야 한다. 주소 모호성이란 명령어 주소의 태그와 인덱스만으로 비압축 캐쉬에 저장된 명령어를 구분할 수 없게 되는 것을 말한다. 이는 상위 메모리와 하위 메모리가 사용하는 명령어 인코딩 기법이 서로 다르기 때문에 발생한다. 예를 들어 그림 1과 같이 명령어 A와 B를 담고 있는 압축 메모리 블록을 가정해 보자. 직접 사상 주소 기법을 사용할 경우 그림 1(a)와 같이 오프셋을 제외한 명령어 A와 명령어 B의 태그, 인덱스, 워드가 모두 일치한다. 따라서 기존의 직접 사상 주소 기법을 사용하기 위해서는 각 캐쉬 블록마다 태그와 함께 명령어들의 오프셋을 저장해 두어 같은 블록내의 명령어들을 구분해 줄 수 있어야 한다. 그러나, 전개 명령어를 저장하는 캐쉬 블록은 해당 압축 메모리 블록의 모든 명령어들을 수용할 수 없기 때문에 캐쉬 충돌로 인한 접근 실패가 늘어나게 되는 문제점이 발생할 수 있다.

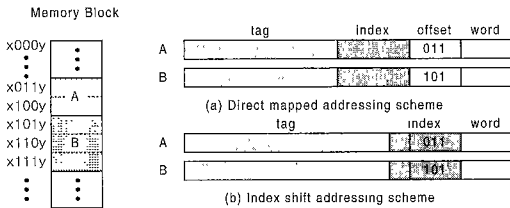


그림 1 VLIW를 위한 캐쉬 주소 기법

주소 모호성을 해결하기 위한 또 다른 방법으로써 그림 1(b)와 같이 명령어 주소의 인덱스 부분을 오프셋 크기만큼 우측 이동함으로써 A와 B의 캐쉬 내 주소를 다르게 잡아주는 인덱스 이동 주소 기법(index shift addressing scheme)을 들 수 있다. 이것은 연속된 명령어를 캐쉬의 서로 다른 위치에 저장하게 하여 캐쉬 충돌로 인한 캐쉬 접근 실패를 줄여 줄 수 있다. 그러나 인덱스를 우측 이동한 만큼 저장해야 할 태그가 늘어나게 되며, 캐쉬 블록 크기를 하나의 명령어 크기로 밖에 할 수 없기 때문에 모든 명령어마다 태그를 유지하고 있어야 한다.

3. 임계 접근 실패율을 통한 성능 예측

2장에서 살펴보았듯이 메모리 활용 효율과 명령어 인출 오버헤드 중 어느 한쪽도 소홀히 해서는 VLIW를 위한 효율적인 캐쉬를 설계하기 어렵다. 즉, 압축 캐쉬는 메모리 활용 효율을 높여 줄 수 있었지만 분기 예측 오류 비용의 증가로 인해 성능 개선에 장애가 되는 반면, 비압축 캐쉬에서는 명령어 길이를 일정하게 유지해줌으로써 명령어 인출 오버헤드를 줄여줄 수 있으나 낮은 메모리 활용 효율로 인해 성능 개선이 어려운 문제를 안고 있다. 본 장에서는 메모리 활용 효율과 명령어 인출 오버헤드의 관계를 살펴보기 위하여 캐쉬 접근 실패와 분기 예측 오류 비용을 기반으로 한 임계 접근 실패율을 소개한다. 임계 접근 실패율이란 분기 예측 오류 비용에 있어우위를 유지하기 위해 허용되는 캐쉬 접근 실패율을 말한다.

메모리 계층 구조가 두 단계의 캐쉬와 주 메모리로 구성된 VLIW 프로세서에서 명령어 당 평균 수행 시간, T_{ex}은 다음과 같이 계산할 수 있다. 단, 파이프라인 지연은 오직 제어 헤저드(control hazard)에 의해서만 발생한다고 가정하였다.

$$T_{ex} = (1 + P_b P_{bt} b)\tau_1 + P_{miss_1}(\tau_2 + \tau_3 P_{miss_2}) \quad (1)$$

표 1은 식(1)에서 사용된 변수들을 보여주고 있다. 이때 분기 명령어란 분기 연산어를 포함하고 있는 명령어를 의미한다.

표 1 변수 정의

변수	정의
P_b	분기 명령어를 수행할 확률
P_{bt}	분기 명령어가 프로그램 수행 순서를 바꿀 확률
b	조건 분기 명령어 예측 오류 비용
P_{miss_1}, P_{miss_2}	일차, 이차 캐쉬의 캐쉬 접근 실패율
τ_1, τ_2, τ_3	일차, 이차 캐쉬, 주 메모리 접근 시간

이제 서로 다른 일차 캐쉬 구조를 채택한 두 가지 VLIW 시스템 A, B를 가정해 보자. 시스템 A의 일차 캐쉬는 메모리 활용도 측면에 있어서는 시스템 B에 비해 떨어지나 파이프라인 분기 비용을 줄여주도록 설계되었다. 즉, 시스템 B는 명령어 파이프라인 상에 명령어 복원을 위한 복원 단계가 명령어 인출 단계와 디코드 단계 사이에 삽입되는 반면, 시스템 A에서는 명령어를 복원한 상태로 캐쉬에 적재해 둠으로써 복원 단계가 필요 없도록 하였다. 단, 이차 캐쉬 및 주 메모리는 모두 같은 인코딩 방식과 메모리 구조를 채택하고 있다고 가정한다. 그렇다면, 시스템 A와 시스템 B의 분기 비용을 각각 b^A와 $b^B(b^A < b^B)$로 나타낼 수 있다. 이차 캐쉬 접근 실패율은 일차 캐쉬 접근 실패율에 따라 달라지긴

하지만 시스템 A와 B 모두 같은 구조의 이차 캐쉬를 사용하고 있고, 전체 시스템의 성능에 미치는 영향이 상대적으로 미미하므로 이차 캐쉬 접근 실패율의 차이는 충분히 무시될 수 있다. 따라서, 시스템 A와 시스템 B의 캐쉬 접근 실패율은 P_{miss_2}로 같다고 가정할 수 있다. 시스템 A와 시스템 B의 수행 시간을 각각 T^A_{ex}, T^B_{ex}라 하고 일차 캐쉬의 접근 실패율을 각각 $P^A_{miss_1}$, $P^B_{miss_1}$라고 한다면 다음과 같은 식을 얻을 수 있다.

$$\begin{aligned} T^A_{ex} &= (1+P_bP_{bt}b^A)\,\tau_1 + P^A_{miss_1}(\tau_2+\tau_3P_{miss_2}) \\ T^B_{ex} &= (1+P_bP_{bt}b^B)\,\tau_1 + P^B_{miss_1}(\tau_2+\tau_3P_{miss_2}) \end{aligned} \quad (2)$$

따라서 시스템 B에 대한 시스템 A의 성능 증대, S는 다음과 같이 나타낼 수 있다.

$$\begin{aligned} S &= \frac{T^B_{ex}}{T^A_{ex}} \\ &= \frac{(1+P_bP_{bt}b^B)\,\tau_1 + P^B_{miss_1}(\tau_2+\tau_3P_{miss_2})}{(1+P_bP_{bt}b^A)\,\tau_1 + P^A_{miss_1}(\tau_2+\tau_3P_{miss_2})} \end{aligned} \quad (3)$$

이 때, 시스템 A가 시스템 B보다 높은 성능을 얻기 위해서 식(3)은 항상 1보다 큰 값을 유지해야 한다. 따라서, 식(3)으로부터 다음과 같은 식을 유도할 수 있다.

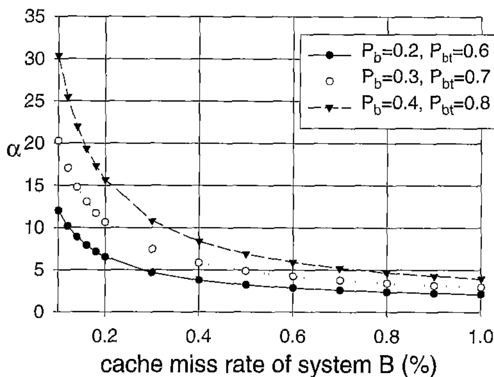
$$\begin{aligned} & S > 1 \\ \Leftrightarrow\ & (1+P_bP_{bt}b^A)\,\tau_1 + P^A_{miss_1}(\tau_2+\tau_3P_{miss_2}) \\ & < (1+P_bP_{bt}b^B)\,\tau_1 + P^B_{miss_1}(\tau_2+\tau_3P_{miss_2}) \\ \Leftrightarrow\ & (P^A_{miss_1}-P^B_{miss_1})(\tau_2+\tau_3P_{miss_2}) \\ & < (b^B-b^A)P_bP_{bt}\tau_1 \end{aligned} \quad (4)$$

$P^A_{miss_1}$를 제외한 모든 변수를 식(4)의 우 항으로 옮기면 다음과 같다.

$$P^A_{miss_1} < \left\{1+\frac{(b^B-b^A)P_bP_{bt}\tau_1}{P^B_{miss_1}(\tau_2+\tau_3P_{miss_2})}\right\}\cdot P^B_{miss_1} \quad (5)$$

이 때, 식(5)의 우항은 시스템 A가 시스템 B에 대해 성능 우위를 유지하기 위한 캐쉬 접근 실패 상한 값 즉, 임계 접근 실패율을 나타낸다. 이것은 만일 시스템 A가 낮은 메모리 활용 효율로 인해 캐쉬 접근 실패율이 증가하더라도 그 값이 임계 접근 실패율 보다 낮다면 시스템 B에 대한 시스템 A의 성능 개선을 기대할 수 있음을 의미한다.

그림 2에서는 $\alpha = 1+\frac{(b^B-b^A)P_bP_{bt}\tau_1}{P^B_{miss_1}(\tau_2+\tau_3P_{miss_2})}$라 할 때, 시스템 B의 일차 캐쉬 접근 실패율에 대한 α의 변화를 보여주고 있다. 각 단계의 메모리 접근 지연 시간 τ_1, τ_2, τ_3를 각각 1CPU 사이클, 5CPU 사이클, 59CPU 사이클로 가정하였다. 이들은 성능 평가를 위한 모의실험을 위해 가정한 값들이며 자세한 내용은 6장에서 설명하겠다. 명령어 파이프라인 상에 복원 단계가 없는 시스템 A의 경우 분기 오류 비용이 1CPU 사이클인 반면 시스템 B의 경우 복원 단계가 삽입됨으로써 분기 오류 비용이 2 CPU 사이클로 증가한다. 이차 캐쉬 접근 실패율은 시스템 A와 시스템 B 모두 0.1로 일정하다고 가정하였다. 일반적으로 프로그램내의 분기 명령어는 20~40%를 차지하고 있으며 이중 분기가 일어날 확률은 0.6~0.8 정도로 나타나고 있다[11, 12]. 따라서 P_b와 P_{bt}의 값이 각각 0.2와 0.6, 0.3과 0.7, 그리고 0.4와 0.8일 경우 α의 변화를 살펴보았다. 그림 2로부터 시스템 B의 일차 캐쉬 접근 실패율이 0.1% ~ 1.0%을 나타낼 때 α는 최소 2.1에서 최대 30의 값을 갖는 것을 알 수 있다. 이것은 시스템 A의 메모리 활용도가 떨어져 시스템 B의 캐쉬 접근 실패율에 비해 최소 2.1 배에서 최대 30배의 캐쉬 접근 실패율을 보이더라도 분기 오류 비용의 감소로 인해 성능 개선을 얻을 수 있음을 의미한다. 반면, 시스템 B의 일차 캐쉬 접근 실패율이 1.0% 이상일 경우 α가 급격히 낮아지지만 프로그램 내의 조건



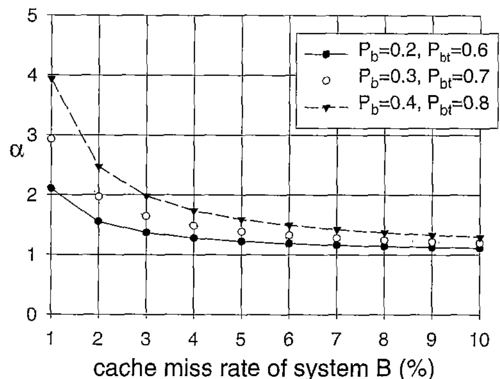


그림 2 시스템 B의 일차 캐쉬 접근 실패율에 따른 α의 변화

분기 명령어가 차지하는 비율에 따라 시스템 B의 캐쉬 접근 실패율이 1.5%~3%를 유지한다면 시스템 A가 2배 이상의 캐쉬 접근 실패율을 보이더라도 성능 증대 효과를 거둘 수 있음을 보여주고 있다.

4. 부분 압축 인코딩과 분할 캐쉬

3장에서 살펴보았듯이 캐쉬 메모리의 성능을 높이기 위해서는 메모리 활용 효율과 함께 명령어 인출 오버헤드에 대한 개선이 있어야 한다. 본 장에서는 명령어 인출을 간단히 하여 파이프라인 분기 오류 비용을 줄이면서 동시에 적절한 메모리 활용 효율을 얻을 수 있는 새로운 명령어 인코딩 방식인 부분 압축 인코딩 방식을 제안한다. 또한, 이를 효과적으로 저장·검색하기 위한 분할 캐쉬 구조에 대하여 설명한다. 본 장에서는 임의의 VLIW 명령어 인코딩 방식에 대해 VLIW 명령어를 구현하기 위해 필요한 연산어 슬롯의 수를 LOF(*length of format*)로 나타내고 있다.

4.1 부분 압축 인코딩(semi-packed encoding)

부분 압축 인코딩 기법은 VLIW 명령어를 여러 개의 부분 명령어(sub-instruction)로 구성하는 방법이다. 이 때, 부분 명령어의 LOF는 VLIW 명령어의 최대 LOF의 약수들 중 하나로 항상 일정하게 유지된다. 예를 들어, VLIW 명령어의 최대 LOF가 8이라면 부분 명령어의 LOF는 1, 2, 4 혹은 8 중 하나가 될 수 있다. 이와 같이 부분 명령어의 LOF를 제한함으로써 명령어 인출뿐만 아니라 캐쉬 접근을 위한 주소 계산을 간단하게 할 수 있다. VLIW 명령어의 최대 LOF를 8이라 하고 부분 압축 인코딩에 사용되는 부분 명령어의 LOF를 2라 한다면, 3개의 유효 연산어를 갖는 명령어는 2개의 부분 명령어로 인코딩할 수 있다. 즉, 전개 인코딩을 사용한다면 5개의 유효 연산어를 저장할 수 있는 공간이 낭비되는 반면, 부분 압축 인코딩의 경우 이를 $\frac{1}{5}$로 줄여줄 수 있다.

일반적으로, 부분 명령어의 길이가 짧을수록 높은 메모리 활용 효율을 얻을 수 있으나 연산처리기로의 제어 경로가 복잡해지며 캐쉬 테이블을 위한 저장 오버헤드가 증가하게 된다.

4.2 분할 캐쉬(section cache)

분할 캐쉬는 분할 압축 명령어를 효율적으로 적재·검색하기 위한 캐쉬 구조이다. 분할 캐쉬는 기본적으로 여러 개의 캐쉬 섹터로 구성되며 하나의 명령어는 각 캐쉬 섹터에 부분 명령어 별로 나뉘어 저장된다. 분할 캐쉬를 구성하는 캐쉬 섹터의 수는 VLIW 명령어의 최대 LOF와 부분 명령어의 LOF에 의해 결정된다. 즉, VLIW 명령어의 최대 LOF를 l이라 하고 부분 명령어의 LOF를 l_{SI}라 하면 분할 캐쉬는 총 l / l_{SI}의 캐쉬 섹터로 구성된다. 태그, 블록 상태, 명령어 길이에 대한 정보를 갖고 있는 캐쉬 테이블 엔트리는 각 캐쉬 섹터의 블록(섹터 블록)마다 유지하고 있어야 한다.

그림 3은 부분 명령어의 LOF가 VLIW 최대 명령어의 $\frac{1}{4}$이라 할 때, 이를 위한 분할 캐쉬의 기본 모델을 보여주고 있다. 분할 캐쉬는 집합 연관 캐쉬와 매우 유사한 구조를 갖고 있으나 캐쉬 테이블뿐만 아니라 명령어 인출 방식에 있어서 차이가 있다. 즉, 집합 연관 캐쉬는 각 way에서 읽어 온 블록 중 태그가 일치하는 하나의 블록만이 선택되어 명령어 파이프라인으로 들어가지만, 분할 캐쉬의 경우 각 캐쉬 섹터에서 읽어 온 부분 명령어들 중 명령어를 구성하는 하나 이상의 부분 명령어들이 동시에 명령어 파이프라인으로 들어간다. 이 때, 해당 부분 명령어를 고르기 위해 각 캐쉬 섹터에 해당되는 테이블 엔트리를 사용된다. 명령어를 구성하는 부분 명령어들은 각 캐쉬 섹터에 순서대로 저장되기 때문에 따로 순서를 맞춰주기 위한 조정 과정을 필요로 하지 않으며 따라서 파이프라인으로부터 복원 단계를 제거할 수 있다.

분할 캐쉬는 압축 명령어를 저장하고 있는 하위 메모

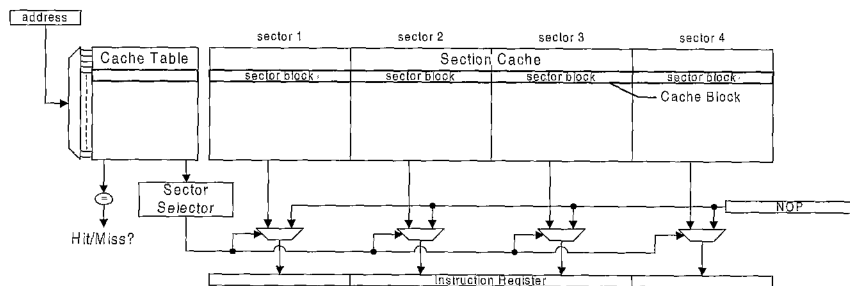


그림 3 분할 캐쉬의 기본 모델

리로부터 명령어를 읽어올 경우 주소 모호성의 문제가 발생한다. 이러한 문제는 비압축 캐쉬와 마찬가지로 직접 사상 주소 기법과 인덱스 이동 주소 기법을 사용하여 해결할 수 있다. 본 논문에서는 인덱스 이동 주소 기법을 사용한 블록 분산형 분할 캐쉬(LCSC, Loosly Coupled Section Cache)와 전통적인 직접 주소 사상 기법을 적용한 블록 집중형 분할 캐쉬(TCSC, Tightly Coupled Section Cache)을 소개한다.

4.2.1 LCSC

LCSC는 주소 모호성의 문제를 해결하기 위해 인덱스 이동 주소 기법을 사용하는 분할 캐쉬이다. LCSC의 섹터 블록은 하나의 부분 명령어만을 수용할 수 있는데, 이것은 인덱스 이동 주소 기법에서 주소 계산을 블록 단위가 아닌 연산어 단위로 수행하기 때문이다.

그림 4는 *s*개의 캐쉬 섹터를 갖는 LCSC의 캐쉬 테이블 엔트리 구조를 보여주고 있다. 캐쉬 테이블 엔트리는 캐쉬 블록마다 할당되며 각 섹터 블록이 수용하고 있는 명령어의 태그 및 길이에 대한 정보와 함께 비어있는 섹터 블록에 대한 정보(*r_sec*)를 가지고 있다.

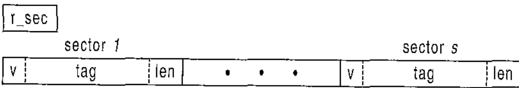


그림 4 LCSC 캐쉬 테이블 엔트리 구조(r_sec: remained sector field, v: valid bit field, tag: tag field, len: length field)

*r_sec*은 하위 메모리로부터 읽어온 명령어를 해당 캐쉬 블록에 저장하기 전에 명령어를 저장하기 위해 필요한 섹터 블록의 수와 비교하여 다른 명령어들과 함께 해당 캐쉬 블록을 공유할 수 있는지를 결정하는데 사용된다. 즉, 만일 적재될 명령어가 필요로 하는 섹터 블록의 수가 *r_sec* 보다 작다면 해당 캐쉬 블록이 명령어를 수용할 수 있는 빈 섹터 블록들을 가지고 있음을 의미하며 따라서 이미 캐쉬 블록을 일부 점유하고 있는 다른 명령어들과 함께 해당 캐쉬 블록을 공유할 수 있다. 반대로 만일 요구하는 섹터 블록의 수가 *r_sec* 보다 크다면 해당 캐쉬 블록은 기존 명령어들을 제거하지 않고서는 명령어를 적재할 수 없음을 의미한다. 따라서 섹터 블록들을 확보하기 위해 해당 캐쉬 블록을 차지하고 있는 기존 명령어들을 제거해야 하는데 이때 필요한 만큼의 섹터 블록만을 확보함으로써 메모리 활용 효율을 높여 줄 수 있는 방안을 생각해 볼 수 있지만 이것은 매우 복잡한 적재 알고리즘을 요구한다. 따라서 본 논문에서는 필요한 섹터 블록이 모자라는 경우에 해당 캐쉬 블록을 점유하고 있는 모든 명령어들을 제거하는 적재 알고리즘을 사용한다.

CPU로부터 임의의 명령어에 대한 요구가 발생하여 압축 명령어를 갖고 있는 메모리로부터 명령어를 읽어와야 할 때 해당 명령어뿐만 아니라 명령어를 포함하고 있는 메모리 블록이 모두 LCSC에 저장되게 된다. 예를 들어, 그림 5와 같이 CPU로부터 명령어 B에 대한 요구가 있을 때, A, B, C 라는 3개의 명령어를 갖는 메모리 블록을 LCSC에 적재하게 된다. LCSC는 2개의 캐쉬 섹터로 구성되어 있으며 메모리 블록이 적재될 해당 캐쉬 블록들에는 이미 X, Y, Z라는 명령어가 캐쉬 블록의 일부 혹은 전부를 점유하고 있다. 명령어 A는 명령어 X와 마찬가지로 한 개의 부분 명령어만으로 구성되므로 이 두 명령어는 해당 캐쉬 블록을 공유할 수 있다. 그러나, 두개의 부분 명령어로 인코딩 되는 명령어 B의 경우는 비록 해당 캐쉬 블록의 명령어 Y가 하나의 섹터 블록을 차지하고 있더라도 명령어 B가 두 개의 섹터블록을 필요로 하므로 캐쉬 블록을 함께 공유할 수 없다. 반대로, 명령어 Z는 캐쉬 블록을 모두 차지하고 있으므로 적재할 명령어 C의 크기에 상관없이 자리를 양보해야 한다.

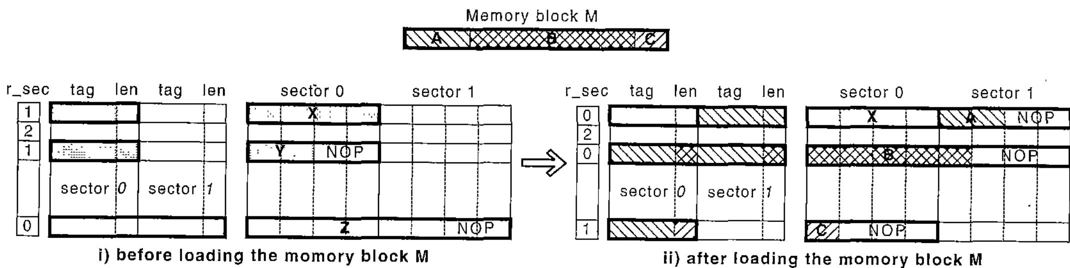


그림 5 메모리 블록의 LCSC 적재 과정(명령어 A, B, C의 인덱스 주소는 각각 명령어 X, Y, Z의 인덱스 주소와 일치한다)

4.2.2 TCSC

TCSC는 직접 사상 주소 기법을 기반으로 한 캐쉬 구조이다. LCSC와는 달리 TCSC의 주소 계산은 연산어 단위가 아닌 블록 단위로 수행되므로 각 섹터 블록마다 한 개 이상의 부분 명령어를 가질 수 있다. 그러나 같은 블록 내의 명령어들을 구분하기 위해 각 부분 명령어마다 명령어의 길이와 함께 주소 오프셋에 대한 정보를 유지하고 있어야 한다. 메모리 활용 효율을 높이기 위해 LCSC처럼 부분 명령어마다 태그 정보를 갖게 할 수 있지만 이것은 명령어의 적재 과정을 매우 복잡하게 한다. 본 논문에서는 캐쉬 블록 단위로 태그 정보를 유지하게 함으로써 메모리 활용 효율 측면에서는 떨어지지만 명령어 적재 과정을 간단하게 하고 태그를 저장하는데 필요한 하드웨어를 줄여 줌으로써 이를 보완하고 있다. TCSC는 캐쉬 블록마다 캐쉬 테이블 엔트리를 가지고 있는데 그림 6은 캐쉬 블록이 $s \cdot n$개의 부분 명령어를 수용할 경우의 캐쉬 테이블 엔트리 구조를 보여주고 있다.

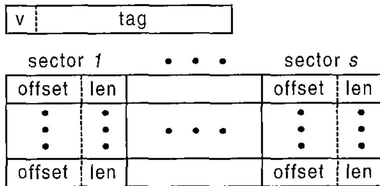


그림 6 TCSC의 캐쉬 테이블 엔트리 구조(v: valid bit field tag, tag: tag field, offset: offset field, len: length field)

명령어는 연산어의 순서를 해당 캐쉬 블록의 어느 곳에나 저장할 수 있다. 즉, 압축 명령어를 저장하고 있는 메모리 블록을 적재할 때 메모리 블록내의 위치에 상관없이 명령어 내의 연산어 순서를 유지할 수만 있다면 캐쉬 블록내의 임의의 위치로 저장될 수 있다. 그러나 메모리 블록의 크기가 캐쉬 블록과 같을 경우, 캐쉬 블록은 최대 $s \cdot n$개의 명령어 밖에 저장할 수 없지만, 메모리 블록은 최대 $s \cdot n \cdot l_{SI}$의 명령어를 저장할 수 있다. 따라서 캐쉬 블록이 메모리 블록의 명령어들을 모두 수용할 수 없는 상황이 발생할 수 있다. 이러한 문제를 해결하기 위한 방안으로 주소 블록(address block)의 크기를 전송 블록(transfer block) 보다 크게 잡아주는 방법이 있다. 주소 블록이란 TCSC 내의 캐쉬 블록 할당 단위를 말하며 전송 블록은 캐쉬 접근 실패가 일어났을 때 메모리로부터 전송 받는 블록 단위를 말한다. 예를 들어 CPU가 명령어 B를 요구하여 해당 명령어뿐만 아니라 명령어 A와 C를 포함한 블록을 하위 메모리로부터 전송 받아야 하는 경우를 가정해 보자. 이 때, 명령어 B는 두 개의 부분 명령어로, 명령어 A와 C는 각각 한 개의 명령어로 인코딩 되어 있다. 만일 주소 블록의 크기와 전송 블록의 크기가 같다면 명령어 B 밖에 저장할 수 없지만 주소 블록이 전송 블록의 두 배라면 그림 7(i)와 같이 명령어 B와 함께 각각 하나의 부분 명령어로 구성되는 명령어 A와 C를 함께 저장할 수 있다. 또 다른 방법으로 TCSC의 연관도를 증가시켜주는 방법을 생각할 수 있다. 즉, 그림 7(ii)에서 보여주듯이 연관도를 2로 늘려주어 첫 번째 way에서 명령어를 모두 수용하지 못할 경우 나머지 명령어를 다른 way에 저장하게 하는 것이다. 그러나 주소 블록의 크기나 연관도를 늘려주는데 있어서 메모리 활용도를 떨어뜨리거나 캐쉬 접근 시간이 늘어나지 않도록 유의해야 한다.

5. 하드웨어 비용 분석

효과적인 캐쉬 설계를 위해서는 수행 성능과 함께 하드웨어 복잡도에 대한 고려가 있어야 한다. 하드웨어 복

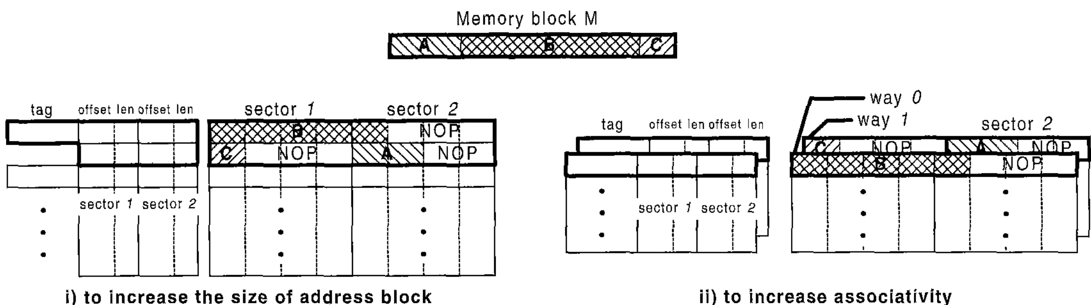


그림 7 TCSC로 메모리 블록을 적재하는 방법

잡도를 측정하기 위한 방법으로 여러 가지 기준을 적용할 수 있겠지만, 본 논문에서는 분할 캐쉬의 구현에 드는 하드웨어 비용을 측정하기 위하여[13]에서 정의한 온 칩 집합 연관 캐쉬를 위한 영역 모델을 이용하였다. 이것은 캐쉬 구현에 필요한 칩 영역을 수식으로 표현하기 위해 데이터 메모리뿐만 아니라 캐쉬 제어기, 태그와 상태 비트 등을 저장하는 캐쉬 테이블, 비교 회로, 증폭 회로, 멀티플렉서 등에 필요한 칩 영역을 포함하고 있다. 캐쉬 영역 단위로 **rbe**(*register bit equivalents*)를 사용하고 있으며 캐쉬 영역은 다음과 같이 표현할 수 있다.

$$
\begin{aligned}
Area = \; & 130 + 0.6(blksize \times assoc + 6)\left(\frac{\#entries}{assoc} + 6\right) \\
& + 0.6((\#tagbits + \#statusbits \\
& + \#lenbits) \times assoc + 6)\left(\frac{\#entries}{assoc} + 12\right) \quad \text{(rbe)}
\end{aligned}
$$

이때, *blksize* 캐쉬 블록의 크기를 말하며, *assoc*는 캐쉬 연관도, *#entries*는 총 캐쉬 엔트리의 수, *#tagbits*는 명령어 주소의 태그 비트 수, *#statusbits*는 상태 비트 수, *#lenbits*는 길이 비트 수를 의미한다.

우선, 위 수식 모델을 이용해 캐쉬 구조의 칩 영역을 계산하기 위해, 각 캐쉬 구조를 집합 연관 캐쉬의 구조로 매핑하였다. 즉, 비압축 캐쉬 및 압축 캐쉬의 경우 집합 연관 캐쉬와 유사한 구조를 띄고 있으므로, 각각 연관도가 1, 2인 집합 연관 캐쉬로 가정하여 칩 영역을 계산할 수 있다. 반면, 섹션 캐쉬의 경우 구조 매핑을 위해 각 캐쉬 섹터를 집합 연관 캐쉬의 way로 가정할 수 있다. 따라서, 섹터 블록은 집합 연관 캐쉬의 캐쉬 블록에 해당된다. 특히, TCSC의 경우 태그는 섹터 블록이 아닌 캐쉬 블록 단위로 할당되기 때문에 집합 연관 캐쉬로의 매핑을 위해 태그를 캐쉬 섹터의 수로 나눈 값을 집합 연관 캐쉬의 태그로 하여 계산하였다.

표 2는 캐쉬 크기를 8KB로 하였을 때 각 캐쉬 구조의 칩 영역 계산을 위해 가정한 변수 값들과 이들을 통해 계산한 하드웨어 영역을 보여주고 있다. 64비트 크기의 연산자와 32비트 주소를 기반으로 하였으며 VLIW 명령어의 최대 LOF를 8로 가정하였다. 분할 캐쉬의 경우 캐쉬 테이블의 크기는 부분 명령어의 LOF에 따라 달라지므로 부분 명령어의 LOF를 각각 1, 2, 4로 다르게 하였을 때 LCSC와 TCSC의 칩 영역의 변화를 알아보았다. 제어 회로 구현에 필요한 하드웨어 영역은 캐쉬 구조에 상관없이 모두 130으로 일정하다고 가정하였다. 사실, 보다 정확한 계산을 위해서는 각 캐쉬 구조의 인출 하드웨어 영역 및 적재 알고리즘에 따른 제어 회로 영역의 변화가 계산에 포함되어야 하지만 비교 회로, 유도 회로, 증폭기 등 그 이외의 많은 상세한 부분들을 포함하고 있기 때문에 비교적 정확한 하드웨어 비용을 얻을 수 있다.

표 2 8KB 크기의 캐쉬를 위한 칩 영역(rbe)(CM: 압축 캐쉬, DC: 비압축 캐쉬)

		blksize	*assoc*	*#tagbits*	*#statusbit +#lenbits*	*#entries*	table area	total area
CM		512	2	19	40	128	5654	49044
DC		512	1	22	4	128	2688	44465
LCSC	l_{SI}=1	64	8	22	4	1024	17976	59753
	l_{SI}=2	128	4	22	4	512	9240	51017
	l_{SI}=4	256	2	22	4	256	4872	46649
TCSC	l_{SI}=1	64	8	5.38	4	1024	6807	48585
	l_{SI}=2	128	4	7.75	4	512	4452	46229
	l_{SI}=4	256	2	12.5	4	256	3276	45053

캐쉬 구현에 가장 많은 비용 차이가 발생하는 캐쉬 테이블을 위한 칩 영역을 살펴보면, 비압축 캐쉬가 다른 캐쉬에 비해 적게는 16%에서 많게는 84%의 영역 감소를 얻을 수 있음을 알 수 있다. 이는 블록 단위로 유지해야 하는 길이 정보가 상대적으로 적을 뿐만 아니라 캐쉬 영역에 많은 영향을 주는 연관도와 엔트리 수 역시 매우 낮기 때문이다. 반면, LCSC는 부분 명령어마다 캐쉬 테이블 엔트리를 유지하고 있기 때문에 부분 명령어의 LOF가 1, 2일 경우 압축 캐쉬보다도 많은 영역을 차지하고 있다. TCSC의 경우 부분 명령어마다 길이와 상태 정보를 갖고 있기는 하나 캐쉬 블록단위로 태그에 대한 정보를 유지함으로써 하드웨어 비용을 많이 줄여줄 수 있었다. 특히, 부분 명령어의 LOF가 2나 4의 경우는 압축 캐쉬보다도 적은 영역을 차지하고 있음을 알 수 있다. 캐쉬 테이블 영역만을 고려할 경우 이와 같이 캐쉬 구조간에 많은 차이가 나지만, 전체적인 캐쉬 영역을 살펴보면 분할 캐쉬는 비압축 캐쉬에 비해 1.3~34%의 영역을 더 필요로 하는 것으로 나타난다. 그러나, 분할 캐쉬의 성능 개선을 고려할 때 어느 정도의 비용 손실은 감수할 수 있다. 특히, 부분 명령어의 LOF가 2 또는 4인 TCSC의 경우 6장에서 설명할 성능 개선을 고려할 때 1~4% 내외의 영역 확장은 무시할 만하다.

6. 성능 평가

본 논문에서는 모의 실험을 통하여 부분 압축 인코딩 기법의 압축률을 살펴보았으며, 분할 캐쉬의 메모리 활용 효율 분석과 성능평가를 수행하였다. 이를 위한 실험 도구로 Trimaran을 사용하였으며 성능 비교 대상으로

각각 압축 인코딩과 전개 인코딩, 압축 캐쉬와 비압축 캐쉬를 선정하였다. Trimaran은 크게 컴파일러와 시뮬레이터로 구성되는데 컴파일러는 다시 IMPACT라고 하는 컴파일러 선단부와 ELCOR라고 하는 후단부로 구성되어 있다. Trimaran 컴파일러는 임의의 프로그램 소스 코드를 입력받아 다양한 최적화 기법과 스케줄링 기법들을 통해 Rebel이라고 하는 중간코드를 생성한다. Trimaran 시뮬레이터는 이 중간코드를 해석하여 모의 수행할 수 있는 새로운 코드를 만들어 주게 되는데 본 논문에서는 이 시뮬레이터를 수정하여 실험하고자 하는 VLIW 파이프라인을 모델링하였으며 각 캐쉬 구조들을 위한 모듈들을 구현하였다.

6.1 VLIW 시스템 모델

실험에 사용된 프로세서 모델은 PlayDoh 구조를 기반으로 한 VLIW 프로세서를 가정하였다. 이것은 500 MHz로 구동되며 최대 8개의 연산어를 동시에 처리할 수 있는 구조이다. 연산 처리기는 총 8개가 있는데 각각 3개의 정수 연산 처리기와 2개의 부동 소수점 연산 처리기, 2개의 메모리 연산 처리기 그리고 1개의 분기 연산 처리기로 구성되어 있으며 모든 연산은 1CPU 사이클에 처리된다고 가정하였다. 명령어들은 일련의 명령어 파이프라인을 통해 처리되는데 일차 캐쉬로 어떤 캐쉬 구조를 사용하는가에 따라 파이프라인 단계가 달라진다. 즉, 비압축 캐쉬와 분할 캐쉬의 경우 명령어 인출 단계, 해석 단계, 실행 단계, 데이터 메모리 접근 단계, 재기록 단계 등의 다섯 단계로 구성된 파이프라인을 사용한다. 따라서 분기 예측 오류 비용으로 1CPU 사이클이 요구된다. 반면, 압축 캐쉬의 경우 인출 단계와 해석 단계 사이에 명령어 복원 단계가 추가된 여섯 단계의 파이프라인이 사용되므로 2CPU 사이클로 가정하였다. 실험에 사용된 분기 예측 기법으로 비분기 예측(predict-not-taken) 기법을 채택하였다.

한편, 메모리 시스템 모델은 일차 캐쉬, 이차 캐쉬 그리고 주 메모리로 구성된다. 일차 캐쉬는 명령어 캐쉬와 데이터 캐쉬가 따로 구성되며 명령어 캐쉬의 경우 압축 캐쉬, 비압축 캐쉬, 분할 캐쉬를 사용할 수 있다. 본 연구에서는 일차 명령어 캐쉬의 종류와 크기를 달리하여 다양한 VLIW 시스템 모델들을 실험하였다.

표 3은 실험에 사용된 VLIW 시스템 모델들의 규격을 요약한 것이다. 명령어 캐쉬 크기를 8KB, 16KB, 32KB로 변화시켜가며 실험하였으며 데이터 캐쉬의 크기는 무한하다고 가정하였다. 5장에서 살펴보았듯이 비압축 캐쉬는 칩 영역에 있어서 다른 캐쉬들에 비해 최대 30% 이상의 영역 감소를 보여주고 있다. 따라서 보다 공정한 성능 비교를 위해 비압축 캐쉬의 크기를 다른 캐쉬의 두 배로 한 VLIW 시스템 모델도 함께 실험하였다. 일차 캐쉬의 블록 크기는 64바이트, 이차 캐쉬의 블록 크기는 256바이트이며 모두 온 칩 캐쉬로 가정하였다. 일차, 이차 캐쉬의 접근 시간은 각각 1CPU 사이클과 5CPU 사이클로 하였다. 2차 캐쉬와 주 메모리는 모두 명령어를 압축 인코딩 형태로 저장하고 있으며 주 메모리는 64비트 외부 버스로 연결되어 있다고 가정하였다. 주 메모리 접근 시간은 166MHz SDRAM을 근거로 하였으며, 따라서 64바이트 크기의 블록 전송 시 59CPU 사이클이 걸리게 된다.

표 3 실험에 사용된 다양한 VLIW 시스템 모델

		VLIW System Model						
		I	II	III	IV	V	VI	VII
CPU		500 MHz 8-issue						
L1 Cache	type	CM	DC		LCSC	TCSC	LCSC	TCSC
					$l_{si}=2$		$l_{si}=4$	
	size	8, 16, 32 KB		×2 KB	8, 16, 32 KB			
	block size	64 bytes						
L2 Cache	size	512KB 온 칩 캐쉬						
	block size	256 bytes						
Main Memory		166 MHz SDRAM						

6.2 워크로드 분석

실험에 사용할 벤치마크들로 Trimaran의 paraffins, SPEC92의 li, go, compress와 MediaBench의 epic을 선정하였다. 표 4는 각 벤치마크들의 수행 명령어 및 연산어 수와 조건 분기 명령어의 분포를 보여주고 있는데, 하나의 명령어 당 평균 2.2 개의 유효 연산어를 가지고 있음을 알 수 있다. VLIW에서는 임의의 연산처리기에서 지연이 발생했을 경우 모든 프로세서가 지연되는 동기 연산 수행(lock-step operation)을 하므로 조건 분기 명령어의 분포는 연산어가 아닌 명령어를 단위로 계산한다. 전체적으로 조건 분기 명령어는 전체 수행 명령어의 약 24 %를 차지하고 있다. 따라서, 전체 수행 명령어 중에 분기되는 조건 분기 명령어가 차지하는 비율은 벤치마크에 따라 약 13~26%로 나타나고 있음을 알 수 있다.

각 명령어 인코딩 방식들이 프로그램 코드를 어느 정도 압축할 수 있는지 알아보기 위해 다음 식에 따라 압축률을 계산하였다.

표 4 벤치마크들의 수행 명령어 분포

benchmarks	paraffins	li	go	compress	epic
수행 명령어 수	152,762	5,936,639	1,354,878	37,570,202	52,141,582
수행 연산어 수	432,284	10,379,545	2,354,980	70,866,090	144,880,303
조건 분기 명령어 수	30,014	1,507,063	421,928	6,691,432	13,342,121
분기 연산어의 분기율(%)	81.91	64.78	83.55	73.00	74.58
분기가 일어나는 명령어 점유율(%)	16.09	16.44	26.02	13.00	19.08

$$compression\ rate = \frac{compressed\ size}{original\ size}$$

표 5는 각 명령어 인코딩 방식에 따른 벤치마크의 명령어 코드 크기와 압축률을 분석한 내용이다. 압축 인코딩을 사용한 벤치마크들에 대해서는 22~26%의 명령어 코드 압축률을 얻을 수 있는 반면, 부분 압축 인코딩은 부분 명령어의 LOF가 2일 때 40~44%, 4일 때 52~55%의 압축률을 보이고 있다. 이것은 부분 명령어를 사용함으로써 무효 연산어에 의해 낭비되는 부분을 크게 줄여 줄 수 있었기 때문이다. 특히, 부분 명령어의 LOF가 4일 경우 최대 $\frac{1}{2}$의 압축률을 얻을 수 있을 수 있는 것에 비춰볼 때 거의 모든 명령어가 한 개의 부분 명령어만으로 인코딩 될 수 있음을 확인할 수 있다.

표 5 명령어 인코딩 기법에 따른 벤치마크들의 명령어 코드 크기(bytes)와 압축률

벤치마크 \ 명령어 인코딩		전개	압축	부분 압축 $l_{si}=2$	부분 압축 $l_{si}=4$
paraffins	코드 크기	33,088	7,552	13,296	17,248
	압축률	1	0.23	0.40	0.52
li	코드 크기	313,792	72,768	126,240	162,816
	압축률	1	0.23	0.40	0.52
go	코드 크기	1,268,416	284,288	512,848	659,040
	압축률	1	0.22	0.40	0.52
compress	코드 크기	66,240	15,936	27,296	34,912
	압축률	1	0.24	0.41	0.53
epic	코드 크기	199,616	51,264	87,040	109,280
	압축률	1	0.26	0.44	0.55

일반적으로, 메모리 활용 효율은 명령어가 사용중인 메모리 공간에 대한 유효 연산어가 차지하는 메모리 공간의 비로 계산할 수 있다. 그러나 5장에서 설명한 것처럼 캐쉬 구조에 따라 구현에 필요한 칩 영역이 달라지므로 보다 정확한 메모리 활용 효율을 계산하기 위해 하드웨어 비용에 대한 고려가 있어야 할 것이다. 이를 위해 순수 메모리 활용 효율(E)와 함께 압축 캐쉬의 하드웨어 비용으로 정규화 된 메모리 활용 효율, E_c을 계산하였다. 이때, E_c는 다음과 같이 정의할 수 있다.

$$E_c = E \times \frac{C_{CM}}{C}$$

여기서 C는 메모리 활용 효율을 계산하고자 하는 캐쉬 구조의 하드웨어 비용을, C_{CM}은 압축 캐쉬의 하드웨어 비용을 나타낸다. 메모리 압축 캐쉬의 경우 모든 명령어들은 유효 연산어로만 구성되므로 순수 메모리 활용 효율과 E_c는 항상 1이 된다. 비압축 캐쉬의 순수 메모리 활용 효율은 전체 정적 코드 크기를 정적 유효 연산어 크기로 나눈 값으로 간단히 계산할 수 있다. 그러나 분할 캐쉬의 경우는 비압축 캐쉬처럼 단순히 정적 코드의 크기로 순수 메모리 활용 효율을 계산할 수 없다. 예를 들어, 4개의 캐쉬 섹터로 이루어진 분할 캐쉬에 3개의 부분 명령어로 구성된 명령어가 임의의 캐쉬 블록을 차지하고 있다고 하자. 만일 2개의 부분 명령어로 인코딩 된 명령어를 새로 읽어 올 경우 이들은 캐쉬 블록을 함께 공유할 수 없으므로 메모리 활용 효율이 떨어지게 된다. 따라서 분할 캐쉬의 순수 메모리 활용 효율은 임의의 프로그램 수행 시점에서 명령어 코드들이 사용하고 있는 메모리 양에 대한 유효 연산어의 점유 메모리 양의 비로 계산해야 한다. 또한 분할 캐쉬는 사용한 주소 기법에 따라 사용하는 적재 알고리즘이 다르기 때문에 메모리 활용 효율이 달라질 수 있으므로 LCSC와 TCSC의 순수 메모리 활용 효율을 알아보았다. 이렇게 계산된 순수 메모리 활용 효율과 표 2의 하드웨어 비용을 이용해 각 캐쉬 구조의 E_c를 계산할 수 있다.

표 6은 비압축 캐쉬와 LCSC, 그리고 TCSC의 순수 메모리 활용 효율과 E_c를 보여주고 있다. 비압축 캐쉬와 TCSC, 그리고 부분 명령어의 LOF가 4인 LCSC는 압축 캐쉬에 비해 차지하는 칩 영역이 작으므로 이를 고려할 경우 메모리 활용 효율이 4~10%씩 증가하고 있음을 알 수 있다. 비압축 캐쉬의 경우 E_c는 평균 0.26에 그치는 반면 비교적 압축 효율이 좋은 분할 캐쉬의

표 6 캐쉬 메모리 활용 효율

Benchmarks \ Cache type	DC E	DC E_C	LCSC $l_{si}=2$ E	LCSC $l_{si}=2$ E_C	LCSC $l_{si}=4$ E	LCSC $l_{si}=4$ E_C	TCSC $l_{si}=2$ E	TCSC $l_{si}=2$ E_C	TCSC $l_{si}=4$ E	TCSC $l_{si}=4$ E_C
paraffins	0.23	0.25	0.72	0.69	0.43	0.45	0.53	0.56	0.37	0.40
li	0.23	0.25	0.76	0.73	0.43	0.45	0.54	0.57	0.41	0.45
go	0.22	0.24	0.75	0.72	0.42	0.44	0.57	0.60	0.40	0.44
compress	0.24	0.26	0.72	0.69	0.48	0.50	0.54	0.57	0.39	0.42
epic	0.26	0.29	0.79	0.76	0.49	0.52	0.57	0.66	0.43	0.47

E_c는 최소 0.44에서 최대 0.72에 이르고 있다. 특히, 분할 캐쉬는 부분 명령어의 LOF가 작을수록, 낭비되는 메모리의 양이 줄어들게 되므로 메모리 활용 효율이 높게 나오고 있음을 확인할 수 있다. 부분 명령어의 LOF가 같을 경우 TCSC에 비해 LCSC의 메모리 활용 효율이 높아지는 것을 볼 수 있는데, 이것은 TCSC의 경우 명령어들이 임의의 캐쉬 블록을 공유하기 위해서는 그들이 모두 같은 메모리 블록에 있는 것들이어야 하지만, LCSC에서는 명령어들의 메모리 블록이 다르더라도 캐쉬 블록을 함께 공유할 수 있기 때문이다.

6.3 수행 시간 결과 및 분석

캐쉬 접근 실패의 증가를 비용으로 하여 명령어 인출 오버헤드를 줄이고자 한 캐쉬 구조를 일차 캐쉬로 채택한 VLIW 시스템이 압축 캐쉬를 일차 캐쉬로 하는 시스템에 대해 성능 우위를 유지하기 위해 허용 가능한 캐쉬 접근 실패율을 알아보았다. 이를 위해 3장에서 설명한 임계 접근 실패율을 사용하였다. 임계 접근 실패율은 각 벤치마크들의 수행 특성, 즉, 분기 명령어의 코드 점유율, 분기율, 이차 캐쉬 접근 실패율 등을 사용하여 계산할 수 있다. 그림 8은 VLIW 시스템 모델 II-V 에서 발생한 일차 캐쉬 접근 실패율과 임계 접근 실패율을 보여준다. paraffin, li, compress에서, 비압축 캐쉬의 경우 캐쉬 블록 크기를 두 배로 늘려주어도 일차 캐쉬 접근 실패율이 임계 접근 실패율 보다 낮아지지 않는 것을 알 수 있다. 즉, 메모리 활용 저하로 인한 전체적인 메모리 접근 지연 시간의 증가가 너무 크기 때문에, 명령어 인출 오버헤드를 줄이더라도 성능 증대를 기대하기가 어렵다는 것을 보여준다. 반면 분할 캐쉬의 캐쉬 접근 실패율은 모든 벤치마크에서 임계 접근 실패율보다 낮게 나오고 있음을 알 수 있다. 특히, go와 epic의 경우 분할 캐쉬뿐만 아니라 비압축 캐쉬에서도 캐쉬 접근 실패율이 임계 접근 실패율 보다 훨씬 낮게 나오고 있는 것을 알 수 있는데, 이것은 벤치마크의 특성상 매우 높은 지역성로 인해 캐쉬 접근 실패율이 낮게 나오고 또한 전체 코드에서 분기 명령어의 비율이 높기 때문이다. 이전 절에서 살펴보았듯이 일반적으로 TCSC의 메모리 활용도는 LCSC에 비해 낮다. 그럼에도 불구하고, 일부 벤치마크들 — paraffin, li, go — 에서 TCSC가 LCSC보다 낮은 캐쉬 접근 실패율을 보여주는데, 이것은 TCSC의 구조상 LCSC에 비해 충돌형 접근 실패를 많이 줄여 줄 수 있었기 때문이다.

그림 9는 VLIW 시스템 모델 I-V에서 각 벤치마크 프로그램들을 수행하였을 때 연산어 당 소요되는 CPU 사이클(CPO)을 캐쉬 크기 별로 보여준 그래프이다. 전체적으로 캐쉬 크기의 증가에 따른 비압축 캐쉬나 분할 캐쉬의 성능 향상 폭이 압축 캐쉬의 성능 향상 폭보다 큰 것을 볼 수 있다. 이것은 캐쉬 크기가 커질수록 캐쉬 접근 실패로 인한 성능 영향은 줄어들고 상대적으로 명령어 인출 오버헤드가 시스템에 미치는 영향이 증가하기 때문이다. 전체적으로, 부분 명령어의 크기가 2인 분할 캐쉬를 일차 캐쉬로 사용한 VLIW 시스템은 압축 캐쉬를 일차캐쉬로 한 VLIW 시스템에 비해 최고 15 %(평균 10%)의 성능 향상이 있었으며 같은 크기의 비압축 캐쉬를 사용한 시스템에 대해서는 평균 14 %의 성능 향상이 있었다. 또한, 캐쉬 크기가 두 배인 비압축 캐쉬와 비교하더라도 평균 6 %의 성능 향상이 있음을 알 수 있

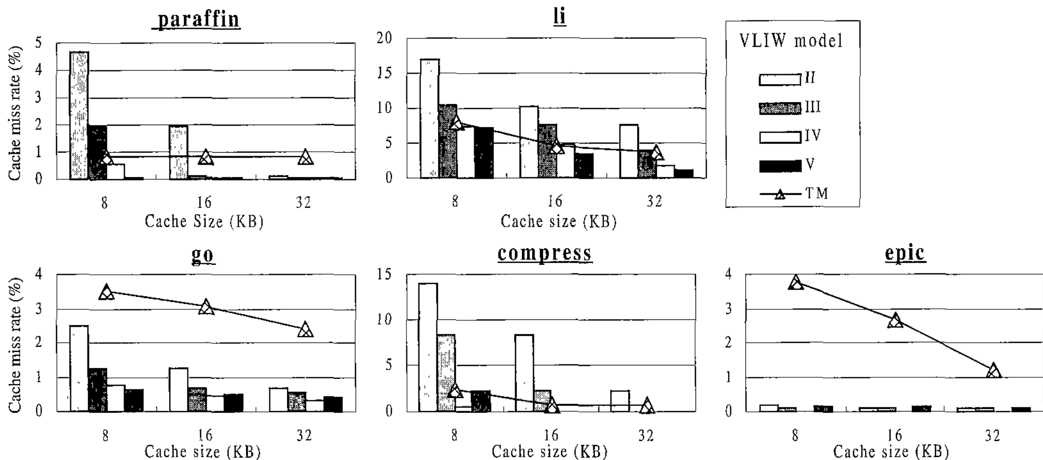


그림 8 일차 캐쉬 접근 실패율과 임계 접근 실패율(TM)

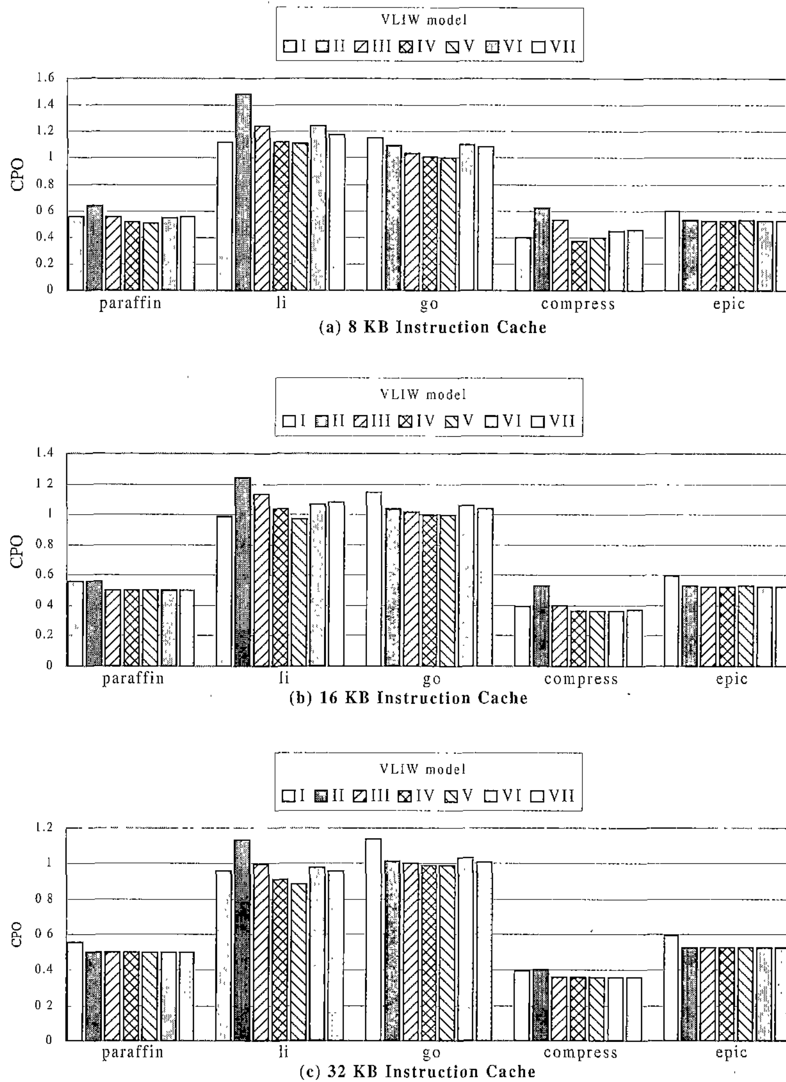


그림 9 VLIW 시스템 모델들의 연산어 당 수행 사이클 수(CPO)

다. 분할 캐쉬의 부분 명령어 크기를 4로 증가할 경우 그 성능이 떨어지긴 하지만, 비압축 캐쉬에 비해서 좋은 성능을 나타내며 비교적 캐쉬 접근 실패가 낮은 벤치마크들에 대해서는 압축 캐쉬보다 높은 성능을 보이고 있다. 특히, TCSC의 경우 낮은 하드웨어 비용으로도 비교적 높은 성능을 얻을 수 있음을 확인하였다.

7. 결 론

VLIW의 명령어들을 구분하기 위해 사용되는 인코딩 방법 중의 하나인 압축 인코딩은 메모리 활용 측면에서 매우 좋은 성능을 보이지만 명령어 인출 오버헤드로 인해 분기 예측 오류 비용이 증가하는 문제점을 안고 있다. 특히, VLIW의 동기 수행 연산으로 인해 분기 예측 오류 비용이 성능 손실에 미치는 영향은 매우 심각하다. 이를 보완하기 위해 메모리 계층 구조의 단계에 따라 하위 단계에는 압축 인코딩을 사용하고 상위 단계에는 전개 인코딩을 사용하는 방법이 제안되었으나 전개 인코딩의 메모리 활용도가 매우 낮기 때문에 성능 개선을

기대하기가 매우 어렵다.

본 논문에서는 임계 접근 실패율을 통해 메모리 활용 효율이 높을 수록 명령어 인출 오버헤드가 전체 시스템 성능에 미치는 영향이 증대하고 있음을 확인하였다. 또한, 명령어 인출 오버헤드를 최소화하면서 메모리 활용 효율을 높이기 위해 전개 인코딩과 압축 인코딩의 중간 형태인 부분 압축 인코딩과 부분 압축 명령어의 효과적인 적재·검색을 위한 분할 캐쉬를 제안하였다. 부분 압축 인코딩은 여러 개의 부분 명령어를 사용하여 낭비되는 연산어 슬롯을 줄여주었으며 또한 부분 명령어의 길이를 일정하게 유지함으로써 캐쉬로부터의 명령어 인출을 간단하게 할 수 있었다. 실험 결과 부분 압축 인코딩의 코드 압축률은 전개 인코딩에 비해 1.5배에서 3배가 높았으며 메모리 활용 측면에서는 분할 캐쉬의 증가되는 하드웨어 비용을 고려하더라도 약 2배에서 3배의 활용 효율을 보였다. 특히, TCSC보다 LCSC가, 부분 명령어의 LOF가 클 때 보다 작을 때, 높은 명령어 활용 효율을 얻을 수 있었다. 그러나, 전체적인 수행 성능을 파악하기 위해 측정한 CPO에 대해서는 TCSC가 비교적 낮은 하드웨어 비용으로 높은 성능을 얻을 수 있음을 확인하였다.

참 고 문 헌

[1] K. Dienfendorff, P. K. Dubey, "How multimedia workloads will change processor design," *IEEE Computers,* pp. 43-45, Sep. 1997.

[2] M. Awaga, T. Ohtsuka, H. Yoshizawa, S. Sasaki, "3D Graphics processor Chip Set," *IEEE Micro,* pp. 37-45, Dec. 1995.

[3] A. K. Riemens and et. al., "TriMedia CPU64 Architecture," *Int. Conf. on Computer Design,* pp. 586-592, Oct. 1999.

[4] Texas Instruments, Inc. "TMS320C62x/C67x Programmer's Guide," 1998.

[5] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Tr. on Computers,* Vol. 37, No. 8, Aug. 1988.

[6] J. H. Moreno, "Dynamic translation of tree-instructions into VLIWs," *IBM Research Report RC20661,* Dec. 1996.

[7] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 Departmental Supercomputer," *IEEE Computer,* pp. 12-35, Jan. 1989.

[8] Compiler and Architecture Group(Hewlett Packard Lab.), ReaCT-ILP Group(New York University), and IMPACT Group(University of Illinois), "Trimaran : An Infrastructure for Compiler Research in Instruction Level Parallelism," New York University, 1998

[9] V. Kathail, M. Schlansker, and B. R. Rau, "HPL PlayDoh Architecture Specification: Version 1.0," Tech. HPL-93-80, Hewlett-Packard Laboratories, Feb. 1994.

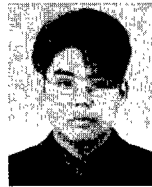
[10] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes and S. W. Sathaye, "Instruction fetch mechanisms for VLIW architectures with compressed encodings," *Int. Symp. on Microarchitecture,* pp. 201-211, Dec. 1996.

[11] K. Hwang, "Advanced Computer Architecture: Parallelism, Scalability, Programmability," McGraw-Hill, Inc., 1993.

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

[13] J. M. Mulder, N. T. Quach, M. J. Flynn, "An Area Model for On-Chip Memories and its Applications," *IEEE Journal of Solid State Circuits,* Vol. 26, No. 2, pp. 98-106, Feb. 1991.

홍 원 기
1995년 연세대학교 이과대학 전산과학과(학사). 1997년 연세대학교 공과대학 컴퓨터과학과(석사). 1997년 ~ 현재 연세대학교 컴퓨터과학과(박사과정). 관심분야는 고성능 컴퓨터, 저전력 컴퓨터 구조, 메모리 시스템, 병렬처리 등



이 승 엽
1996년 한양대학교 공과대학 전자계산학과(학사). 2001년 연세대학교 공과대학 컴퓨터과학과(석사). 2001년 ~ 현재 LG전자 Digital TV(연) 연구원. 관심분야는 고성능 컴퓨터, 3D 그래픽스, 메모리 시스템 등

김 신 덕
1982년 연세대학교 공과대학 전자공학과(학사). 1987년 University of Oklahoma 전기공학(석사). 1991년 Purdue University 전기공학(박사). 1993년 ~ 1995년 광운대학교 컴퓨터공학과 조교수. 1995년 ~ 현재 연세대학교 공과대학 기전공학부 정보산업전공 부교수. 관심분야는 고성능 컴퓨터 구조, 병렬처리 시스템, 메모리 시스템, 인터넷 컴퓨팅 등