논문 2004-41CI-6-1

간접 분기의 타형태 타겟 주소의 정확한 예측

(Accurate Prediction of Polymorphic Indirect Branch Target)

백 경 호*, 김 은 성**

(Kyoung-Ho Paik and Eun-Sung Kim)

요 약

현대적인 프로세서들은 그 성능을 높이기 위해서 분기 예측과 같은 투기적인 방식으로 가용한 ILP 즉 명령어 수준의 병렬성을 추구한다. 전통적으로, 분기 방향은 2-단계 예측기를 사용하여 아주 높은 비율의 정확도로 예측이 가능하고, 분기 타겟 주소는 BTB를 사용하여 예측한다. 간접 분기를 제외한 모든 분기들은 그 자신의 타겟 주소가 유일하기 때문에 BTB로 거의 정확하게 예측되지만, 간접 분기는 그 타겟 주소가 동적으로 수시로 달라지기 때문에 예측하기가 매우 어렵다. 일반적으로, 분기 방향을 예측하는 기술을 간접 분기의 타겟 주소를 예측하는데 적용하여 전통적인 BTB 보다 훨씬 좋은 정확도를 얻고 있다. 본 논문에서는 간접 분기 명령과 이와 데이터 종속적인 관계를 갖고 있는 이 간접 분기 명령 보다 훨씬 앞서 수행되는 명령어의 레지스터 내용을 결합하여 간접 분기의 타겟을 예측하는 전혀 새로운 방법을 제안한다. 제안된 방식의 효율성을 검증하기 위해 심플스칼라 시뮬레이터 상에서 제안된 예측기를 구현하고 SPEC 벤치마크를 시뮬레이션하여, 수시로 바뀌는 간접 분기의 타겟을 거의 완벽하게 예측할 수 있음을 보이고, 기존의 다른 어떤 방법보다도 우수한 결과임을 보인다.

Abstract

Modern processors achieve high performance exploiting avaliable Instruction Level Parallelism(ILP) by using speculative technique such as branch prediction. Traditionally, branch direction can be predicted at very high accuracy by 2-level predictor, and branch target address is predicted by Branch Target Buffer(BTB). Except for indirect branch, each of the branch has the unique target, so its prediction is very accurate via BTB. But, because indirect branch has dynamically polymorphic target, indirect branch target prediction is very difficult. In general, the technique of branch direction prediction is applied to indirect branch target prediction, and much better accuracy than traditional BTB is obtained for indirect branch. We present a new indirect branch target prediction scheme which combines a indirect branch instruction with its data dependent register of the instruction executed earlier than the branch. The result of SPEC benchmark simulation which are obtained on SimpleScalar simulator shows that the proposed predictor obtains the most perfect prediction accuracy than any other existing scheme.

Keywords : Instruction Level Parallelism, BTB, speculative technique, indirect branch target prediction

I. 서 론

최근의 프로세서는 보다 좋은 성능을 위해서 깊은 파이프라인을 통해 다수 개의 명령어를 동시에 이슈하고 처리한다. 연속적인 명령어의 흐름을 유지시켜 프로세서의 파이프라인이 끊기지 않고 수행되게 하려면 가용한 ILP 즉 명령어 수준의 병렬성을 최대한 이용할 수 있도록 해 주어야하지만, 프로그램의 제어 흐름을 변경시키는 제어 종속성의 분기 명령은 분기될 타겟 주소와/또는 분기 조건 결과 생성의 지연으로 인해 ILP를 유용하는데 주요 장애 요인이 된다. 이를 극복하여 연속적인 명령어 흐름을 최대한 보장해 주기 위해 분기 결과를 미리 예측하여 예측된 방향의 타겟 주소의 명령들을 미리 페치하여 투기적으로 수행한다. 잘못 예측되면 투기적으로 수행한 명령들을 취소하고 원상태로 복구하여야 한다. 따라서 분기 명령의 분기 방향과 타겟 예측의 정확성은 투기적 실행 방식에서는 고성능을 보장하기 위해 절대적으로 필요하다.[1]

프로그램에서 나타나는 분기는 네 종류, 즉 직접 무

* 학생회원, ** 종신회원, 순천향대학교 정보기술공학부 (Div. of Information Technology Engineering, Soon Chun Hyang University)

※ 본 논문은 순천향 대학교 대학 자체 학술 연구비 지원(20030067)으로 수행 되었음.

접수일자: 2004년4월1일, 수정완료일: 2004년10월25일

조건/조건 분기, 간접 무조건/조건 분기로 분류할 수 있다. 직접 분기는 항상 그 타겟 주소가 유일한 반면, 간접 분기는 동적으로 그 타겟이 수시로 바뀐다. 또한 조건 분기는 분기 조건에 따라 분기 방향이 결정된다. 따라서 무조건 분기는 타겟 주소의 예측만이 필요하지만, 조건 분기는 타겟 주소와 함께 분기 방향을 예측하여야 한다. 초기 연구의 결과로 분기 방향과 타겟 주소를 함께 예측하는 BTB 구조가 널리 사용되었으나,[2, 3] 좀 더 정교한 예측을 통한 성능 향상을 위해 소결합 분기 구조는 조건 분기가 일어나는지 일어나지 않는지를 예측하기 위해 방향 예측기를 사용하고, 이와는 별도로 일어나는 분기의 타겟을 예측하기 위해 타겟 주소 예측기를 사용한다.[4] 과거의 많은 연구들이 분기 방향 예측 방식들에 집중되어 아주 높은 예측 정확도가 가능하게 되었으나,[5~11] 분기 타겟 예측을 위해서는 BTB를 사용하는 방식을 여전히 사용하여 오다가 간접 분기의 타겟 주소는 단순한 BTB로는 정확하게 예측할 수 없음을 발견하고, 이를 향상시킬 수 있는 연구들이 꾸준히 진행되어 왔다.[12~17]

직접 분기의 타겟 주소 예상은 통상적으로 이 명령이 페치될 때 행해지고, 디코드 단계에서 실제로 결정된다. 예상이 잘못된 경우에는 페치에서 디코드 단계까지의 사이클만큼 명령어 흐름이 잘못되기 때문에 이를 취소하고 프로세서에 이로 인한 부작용이 발생하지 않도록 원상복구 시키고 새로운 명령을 페치해야 하므로 필요한 시간만큼 사이클을 허비하게 되는데, 이를 잘못된 명령 페치에 따른 페널티(instruction misfetch panalty)라 한다. 직접 분기는 페치에서 디코드까지의 사이클 구간이 짧기 때문에 이 페널티로 인한 프로세서 성능에 미치는 영향은 조건 분기의 방향 예측 실패 및 간접 분기의 타겟 예측 실패에 따른 페널티에 비하면 작다. 그러나 조건 분기의 경우에는 그 방향을 예측하여 예측된 방향의 명령들을 미리 페치하여 처리하게 되는데, 실제적인 분기 방향의 결정은 이 분기의 수행이 완결되는 시점에서나 확정되기 때문에 방향 예측 실패 페널티(direction misprediction penalty)가 커지게 된다. 마찬가지로 간접 분기의 타겟 주소를 예측하여 투기적으로 미리 명령을 처리하는 경우에도 그 타겟 주소가 결정되기 위해서는 이전 명령들과의 데이터 종속적인 관계로 인하여 이 간접 분기가 페치된 이후로 타겟 주소를 알게 될 때까지 많은 사이클 지연이 일어나기 때문에 타겟 예측 실패 페널티(target misprediction penalty)가 커지게 된다. 그러므로 조건 분기의 방향 예측뿐만 아니라 간접 분기의 타겟 예측의 정확도를 높이는 것이 프로세서 성능 향상에 아주 중요하게 된다.

대부분의 방향 예측기는 어떤 한 분기의 방향은 이전에 수행되었던 분기 방향의 이력(history)과 밀접한 상관관계가 있다는 전제 하에 이를 이용할 수 있도록 구현되었다. 지금까지 발표되었던 간접 분기 타겟 예측기도 동적으로 다형태의 특성을 갖는 타겟 주소는 이전에 수행되었던 분기 타겟들의 이력들과 상관 관계가 있으므로 방향 예측을 위해 사용되는 방식을 차용하여 이를 적용함으로써 예측 정확도를 높일 수 있도록 하고 있다. 그러나 분기 방향과 마찬가지로 분기 타겟 모두가 이러한 상관관계에 따른 예측 가능한 패턴을 보이지 못하는 경우가 상당하게 존재하는 한계가 있다.

본 논문에서는 기존의 예측기들과는 근본적으로 다른 새로운 방식의 예측기를 제안한다. 즉, 간접 분기 명령과 이와 데이터 종속 관계를 갖고 있는 이 간접 분기 명령 보다 훨씬 앞서 수행되는 명령의 레지스터 값을 결합한 정보를 이용하여 분기 타겟을 예측하는 메카니즘이다. 제안된 데이터 종속 특성을 이용한 간접 분기 예측기의 우수성을 검증하기 위해 심플스칼라 시뮬레이터[18] 상에서 제안된 예측기를 구현하고 이를 통하여 SPEC 벤치마크를 수행하여 동적으로 수시로 바뀌는 간접 분기의 타겟을 거의 완벽하게 예측할 수 있음을 보이고, 기존의 다른 어떤 방법보다도 그 결과가 우수함을 보인다.

Ⅱ. 이론적 배경

예측기 구조 중에서 가장 간단한 것이 그림 1과 같은 분기 타겟 버퍼(BTB)이다.[2, 3] 분기 주소를 인덱스로 사용하여 각 분기에 대한 가장 최근의 타겟 주소를 저장해 둔 타겟 버퍼를 액세스한다. 분기 주소 비트 수가

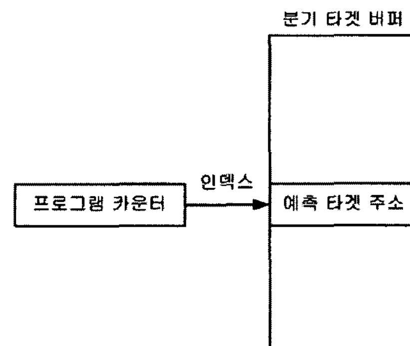


그림 1. 분기 타겟 버퍼
Fig. 1. Branch Target Buffer.

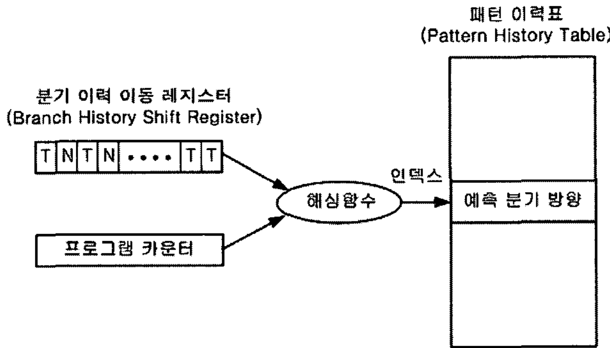


그림 2. 2-단계 적응형 분기 예측기
Fig. 2. 2-level adaptive branch predictor.

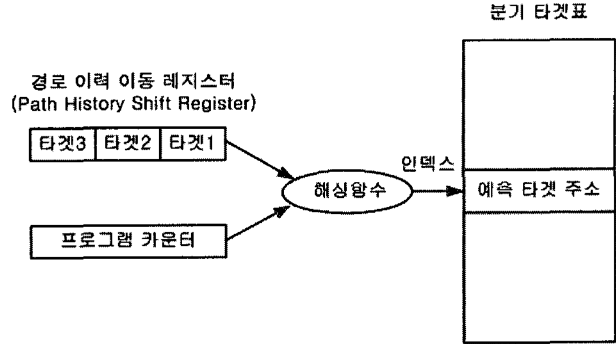


그림 3. 2-단계 간접 분기 예측기
Fig. 3. 2-level indirect branch predictior.

버퍼의 모든 엔트리들을 인덱스하기 위한 것보다 크면, 그 크기만큼의 하위 비트를 인덱스 비트로 사용하고 나머지 상위 비트는 태그로 타겟 버퍼에 저장한다. BTB에 타겟이 없거나 예측이 잘못되면 새로운 타겟 주소로 해당 엔트리 내용을 교체한다. 동적으로 수시로 바뀌는 간접 분기의 타겟 주소를 예상하는 어려움을 완화하기 위해 포화형 2-비트 계수기를 사용하여 연속해서 2회 틀리는 경우에만 교체하는 방식이 제안되기도 하였다.[4]

분기 방향을 예측하기 위한 다양한 형태의 2-단계 적응형 분기 예측기가 제안되었다.[5] 그림 2에서 보는 바와 같이 이 분기 예측기는 이전에 수행되었던 분기의 방향에 따른 이력 패턴을 인덱스로 사용한다. 예상 분기 방향을 패턴 이력표에 저장하는데, 이를 골고루 분산시키기 위해 분기 이력 패턴과 분기 주소를 혼합시키는 해싱 함수를 사용하여 인덱스 주소를 만들어낸다. 일반적으로 해싱 함수로는 xor 함수를 사용하는 것이 좋은 성능을 보이는 것으로 알려져 있다.[6] 좀 더 좋은 성능을 위해 단순한 구조의 예측기부터 복잡한 예측기까지 여러 형태의 예측기로 구성되는 혼합형 예측기도 널리 이용된다. 각각의 독특한 특성을 갖는 예측기마다 서로 보완적인 장점을 갖고 있기 때문에 선택 논리를 두어 이를 경우에 따라 골라서 사용할 수 있도록 하는 것이다.[7~9] 이는 과거의 분기 방향 이력이 지금 예측하려는 분기의 방향과 밀접한 상관관계가 있다는 전제하에 상황에 따라 서로 다른 상관관계를 이용하려는 것이다. 그러나 나타나는 패턴들이 일정하지 않기 때문에 복잡한 패턴으로부터 의미있는 형태의 패턴을 추출하기 위해 데이터 압축 알고리즘이나 신경망 알고리즘을 분기 예측하는데 적용하는 방법도 제안되었다.[10, 11] 과거의 분기 방향을 이력 패턴으로 사용하는 일반적인 패턴 기반의 방식과는 달리, Nair는 예측하려는 분기에 이르는 경로 상에 있는 분기 타겟 주소들을 이력 패턴으로 사용하는 경로 기반의 분기 예측기를 제안하여, 패턴에 포함되는 최근에 수행된 타겟 수로 정의되는 경로 길이를 다양하게 변화시켜가면서 나타나는 성능을 측정하였다.[19] 간접 분기의 타겟이 다형태의 특성을 갖기 때문에 예측하기가 매우 어려워, 경로 기반의 분기 방향 예측 방법을 적용하여 이를 해결하기 위한 방법이 제안되었다.[13] 그림 3에 나타난 바와 같이 제한된 크기의 분기 타겟표의 엔트리 수로 인하여 인덱스로 사용할 경로 이력 크기가 제한되므로 적절한 경로 길이를 선택하는 것이 필요하다. 특히 경로에 포함되는 타겟은 매우 길기 때문에 이 중 일부만을 사용할 수밖에 없게 된다. 따라서 한정된 경로 이력에 포함되는 경로 길이가 길어지면 질수록 타겟 주소는 그만큼 짧아진다. 기존의 분기 방향 예측기에서 사용되는 방식을 적용한 다양한 혼합형 예측기나 복잡한 알고리즘의 간접 분기 예측기가 제안되었다.[14~17]

간접 분기 중에 서브루틴 복귀 명령은 수행된 호출 명령의 반대 순서로 수행되고, 호출 명령이 수행될 때 복귀 주소를 알 수 있기 때문에 이를 별도의 복귀 주소 스택(RAS)에 저장하는 메카니즘이 제안되어 정확하게 예측해낸다.[20] 간접 분기 중에 일부로서 C++ 프로그램에 존재하는 가상 함수 호출(virtual function call)의 타겟을 미리 계산하는 엔진을 사용하는 방법도 제안되었으나, 이는 간접 분기 모두에 적용할 수도 없고 또한 계산 엔진을 별도로 마련해주어야 하기 때문에 하드웨어 부담이 커지게 된다.[21]

III. 간접 분기의 특성

이 절에서는 분기 명령의 특성 특히 간접 분기의 여러 가지 특성을 살펴본다. 모든 간접 분기를 완벽하게 예측할 수 있다면 평균 IPC를 10.8% 까지도 향상시킬

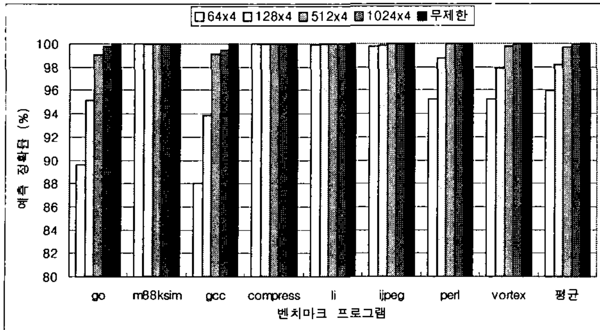


그림 4. BTB의 크기 변화에 따른 직접 분기의 예측 정확도

Fig. 4. Prediction accuracy for direct branches varying with BTB size.

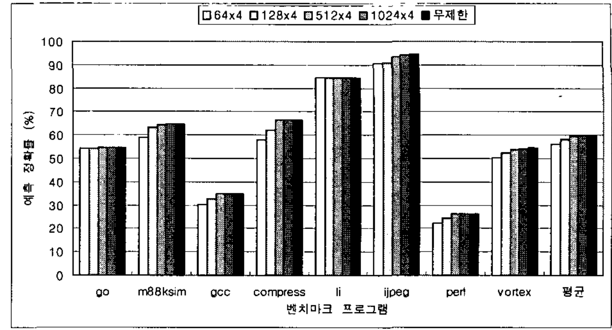


그림 5. BTB의 크기 변화에 따른 return을 제외한 모든 간접 분기의 예측 정확도

Fig. 5. Prediction accuracy for all indirect branches excluding returns, varying with BTB size.

표 1. 벤치마크 프로그램의 동적 특성

Table 1. Dynamic characteristics of benchmark program.

벤치마크	입력	수행 명령 수 (백만개)	분기 당 명령 수	간접 분기 당 명령 수	간접 분기 당 조건 분기 수	평균 타겟 수 (간접 점프)	평균 타겟 수 (간접 호출)
go	2stone9.in(train)	548	6.83	89.18	10.10	5.20	1.00
m88ksim	ctl.raw(ref)	245	4.39	25.36	3.44	5.44	1.40
gcc	cccp.i(ref)	1,263	5.03	51.45	7.93	8.97	14.33
compress	80000 e 2231	239	5.41	35.36	4.10	2.39	1.00
li	train.lsp(train)	183	4.38	25.27	3.34	2.50	23.90
ijpeg	vigo.ppm(train)	1,465	12.29	493.20	33.08	3.48	1.12
perl	jumble.pl(train)	2,392	5.18	39.46	5.42	8.80	1.00
vortex	vortex.raw(train)	2,500	6.09	46.21	5.50	6.20	2.12

표 2. 간접 분기의 타겟 수

Table 2. Number of targets for indirect branch classes.

벤치마크	타겟 수 = 1		타겟 수 = 2		2 < 타겟 수 ≤ 5		5 < 타겟 수	
	간접 점프	간접 호출	간접 점프	간접 호출	간접 점프	간접 호출	간접 점프	간접 호출
go	3	131	0	0	110,788	0	6,664	0
m88ksim	363	601	2	0	704	0	695	25
gcc	16,320	16,271	52,640	5,130	2,941,260	49,311	4,072,277	315,879
compress	0	318	155	0	101	0	0	0
li	3	48	973,774	0	973,686	4,083	0	110,607
ijpeg	789	368,853	0	49,088	8,203	0	0	0
perl	7	3,689,517	23	0	531,250	0	12,867,875	0
vortex	89,893	11,543	87,620	3	137,601	1,272	587,581	2,009

수 있다는 보고[22]에서처럼 간접 분기를 효율적으로 예측하는 것이 ILP 향상에 매우 중요하다는 것을 알 수 다.

그림 4는 분기 방향이 항상 올바르다고 했을 때, 다양한 갈래 연관(set associativity)으로 구성된 1K 엔트리의 BTB와 최적의 결과를 보이는 무제한의 자원을 사용하는 BTB에 대한 각 벤치마크의 직접 분기 타겟 예측 정확도의 변화를 보여주고 있다. 그림에서 보는 바와 같이 직접 분기의 타겟 예측은 BTB의 크기에 제한을 받을 뿐이지 충분한 자원만 마련되면 거의 정확하게 예측할 수 있음을 알 수 있다. 즉, 최초에 수행된 경우를 제외하면 그 외의 것은 항상 정확하게 예측할 수 있게 된다. BTB의 크기에 제한을 두지 않을 때의 평균 예측 정확도는 99.998%에 이르며, 실용적인 측면에서 합리적인 자원 사용을 고려해 보더라도 1K 엔트리로 구성되는 128 × 4와 512 × 4의 집합-연관 사상의 경우

에 각각 98.12%, 99.74%로 아주 높은 정확도를 보인다.

그림 5는 *return* 명령을 제외한 모든 간접 분기의 예측 정확도를 보여 준다. 간접 분기의 타겟은 BTB 자원을 충분히 지원해도 정확하게 예측할 수 없음을 보인다. 즉 무제한의 BTB를 사용한다하더라도 평균 예측 정확도는 60.08%에 지나지 않는다. 따라서 간접 분기의 타겟 예측을 위해서는 별도의 방도를 강구해야 한다.

표 1은 실험에 적용한 SPEC95int 벤치마크 프로그램의 동적 특성을 보여준다. 시뮬레이션 시간을 줄이기 위해서 수행 명령 수를 2,500M로 제한하였다. 각 벤치마크에 대해 사용된 입력 파일, 수행 명령 수, 한 분기 당 수행된 명령 수, 한 간접 분기 당 수행된 명령 수, 한 간접 분기 당 수행된 조건 명령 수 및 간접 점프(*jr* 명령어)와 간접 호출(*jalr* 명령어)의 평균 타겟 수를 나타내었다. 간접 분기는 평균 100개의 명령 수행 중 1개 정도로 나타난다. 벤치마크에 따라 수행된 간접 분기의 빈도수는 편차가 많이 나는데, 많게는 ijpeg의 경우와 같이 간접 분기 당 명령 수가 493개로부터 적게는 m88ksim과 compress에서 25개가 된다. 간접 분기 당 평균 조건 분기 수는 9.1로서 간접 분기의 빈도가 조건 분기보다 낮게 나타났다. 일반적으로 간접 점프 당 타겟 수가 간접 호출의 타겟 수보다 많다. 간접 호출 중의 많은 경우가 타겟이 단 하나인 라이브러리 함수 호출을 위한 것이기 때문에 평균적으로 간접 호출의 수가 1 ~ 2 개정도 밖에 되지 않는데, 예외적으로 gcc와 li는 타겟 수가 많은 특정한 하나의 간접 호출 명령이 특이하게 많이 수행되는 이유로 평균 타겟 수가 많은 것으로 나타났다. 표 2를 보면 타겟 수에 대한 보다 상세한 결과를 볼 수 있다. 위에서 언급한 것처럼 대다수의 간접 호출 명령은 그 타겟 수가 1이고, 간접 점프 명령은 비교적 타겟 수가 적은 것으로부터 많은 것까지 다양하게 나타나는데, 그 이유는 간접 점프 명령은 여러 case 문장으로 향하게 되는 switch 문장을 구현하는데 보통 사용되기 때문이다.

Ⅳ. 데이터 종속적인 특성을 이용한 간접 분기 예측기의 메카니즘

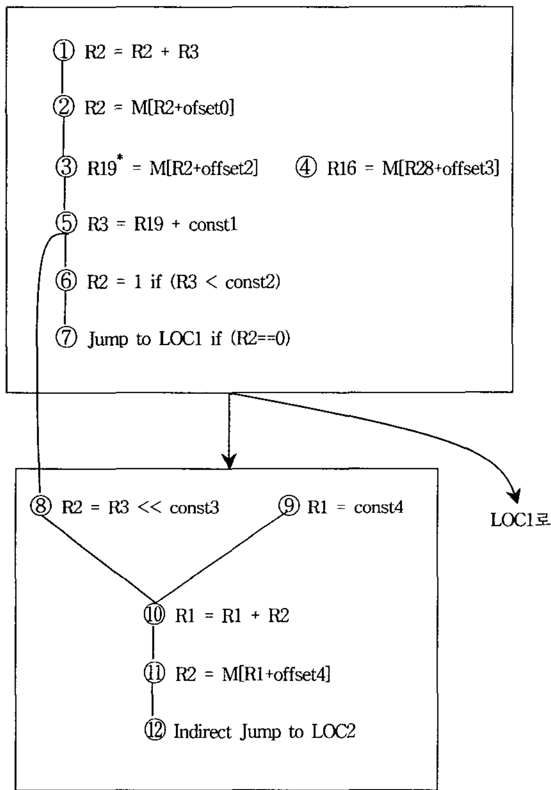
본 논문이 제안한 간접 분기의 타겟 주소 예측 방식은 간접 분기 명령과 이와 데이터 종속적인 관계를 갖고 있는 이 간접 분기 명령 보다 훨씬 앞서 수행되는 명령어의 레지스터 내용을 결합하여 간접 분기의 타겟을 예측하는 메카니즘이다.

먼저 간접 분기의 타겟 주소 결정 과정과 특성을 설명한다. 간접 분기 명령은 switch 문장이나 수행 중 동적으로 생성되는 run-time table을 다루기 위해 사용되고, Global Offset Table(GOT)을 통해 만들어지는 라이브러리 함수 호출과 함수 포인터의 의한 함수 호출 및 C++에서 나타나는 가상 함수 호출을 구현하기 위해 사용된다. 프로그램 수행 중 동적으로 변화하는 간접 분기 명령의 타겟 주소들은 해당 메모리 장소에 각각 저장되어 있으며, 이 메모리 내용 즉 간접 분기의 타겟은 일정하며 변화하지 않는다, 프로그램 수행 중 조건에 따라 적절한 타겟 주소가 저장되어 있는 메모리 주소를 계산하고 결정하여 이를 사용하게 된다. 예를 들어, switch 문장을 수행할 때, 이 문장의 switch 변수 값에 따라 해당 타겟 주소가 있는 메모리의 주소를 계산하여 사용한다. 가상 함수 호출 메카니즘의 경우, 가상 함수 테이블이 주어진 어떤 한 객체 클래스가 액세스할 수 있는 모든 메소드의 주소를 갖고 있으며 미리 결정된 옵셋을 통해 알 수 있다. 따라서 switch 변수 값을 결정하거나 객체를 액세스하기 위한 주소를 결정하는데 영향을 주는 명령어 흐름 중에서 타겟 주소와 아주 밀접한 관계가 있는 명령의 수행 결과를 알면 이에 따라 앞으로 수행될 분기 명령어의 타겟 주소를 미리 예측해 낼 수 있다. 또한 타겟 주소에 결정적인 명령들 중에서 가급적 간접 분기와 멀리 떨어져 있는 명령을 선택하는 것이 좋은데, 슈퍼스칼라 프로세서와 같이 여러 명령들을 동시에 페치하고 이슈하는 경우에 misfetch penalty를 제거하거나 최소화시키기 위한 것이다.

제안된 메카니즘의 특성을 그림 6의 예를 통하여 자세히 설명한다. 그림 6(a)는 SPEC95int 중의 gcc 내에 있는 tree.c의 줄번호 806에 나타나는 switch 문장에 대한 어셈블리 프로그램의 일부이다. 그리고 이에 대한 동적인 제어 및 데이터 종속 그래프를 그림 6(b)에 나타내었다. 명령 12인 간접 분기의 타겟 주소는 명령 11의 메모리 M[R1 + offset4]를 참조하여 알 수 있는데, 레지스터 R1의 값은 이와 데이터 종속적인 이전의 명령들에 따라 수시로 변하게 된다. 따라서 간접 분기 명령의 타겟 주소는 여러 기억 장소에 분산되어 저장된다. 그러므로 R1의 값을 미리 알 수 있다면 타겟이 저장되어 있는 메모리 장소도 미리 알 수 있어 정확한 타겟을 미리 확보할 수가 있다. 그러나 이러한 방식은 R1 값이 어떻게 변화하는지를 미리 알아 두어야하는데, 이를 계산하고 지원하기 위한 복잡한 하드웨어 메카니즘이 별도로 있어야 하고 또한 계산된 값을 실제로 사용

```
 1: addu  $2, $2, $3          // R2 = R2 + R3
 2: lw    $2, 0($2)           // R2 = M[R2 + offset0]
 3: lb    $19, 0($2)          // R19 = M[R2 + offset2]
 4: lw    $16, -20764($28)    // R16 = M[R28 + offset3]
 5: addiu $3, $19, -49        // R3 = R19 + const1
 6: sltiu $2, $3, 72          // R2 = 1 if (R3 < const2)
 7: beq   $2, $0, 004488d0    // jump to LOC1 if (R2 == 0)
 8: sll   $2, $3, 0x2         // R2 = R3 << const3
 9: lui   $1, 4097            // R1 = const4
10: addu  $1, $1, $2          // R1 = R1 + R2
11: lw    $2, -12792($1)      // R2 = M[R1 + offset4]
12: jr    $2                  // indirect jump to LOC2
```

(a) 어셈블리 코드의 예
(a) An example of assembly code.



(b) 어셈블리 코드에 대한 동적 종속 그래프
(b) Dynamic dependence graph for the assembly code.

그림 6. 어셈블리 코드의 예
Fig. 6. An assembly code and the dependence graph.

하기 전에 빠르게 제공해 주어야 하는 어려움과 한계가 있다.[21]

간접 분기의 타겟 주소에 대해 한 가지 주목해야 할 흥미로운 사실은 그 각각의 기억 장소의 내용은 변하지 않고 고정되어 있다는 것이다. 이러한 특성을 이용하면 타겟이 저장되어 있는 메모리 장소를 정확하게 알 필요

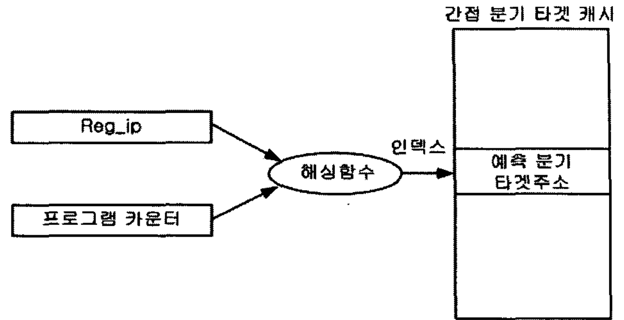


그림 7. 제안한 간접 분기 예측기
Fig. 7. The proposed indirect branch predictor.

가 없이도 타겟 주소를 정확하게 알아낼 수가 있다. 간단하게 설명하기 위해서 레지스터 R1이 100일 때의 메모리 주소가 1100이고 이에 저장된 타겟 주소는 5000이며, R1이 200일 때 메모리 주소는 1200이 되고 저장된 타겟 주소가 6000이 된다고 가정해 보자. 이 명령들이 각 한번씩 수행되어 레지스터 내용의 타겟 주소로의 사상 관계를 알고 있다면, 분기 명령을 다시 반복해서 수행되는 경우에 R1이 100인 것을 안다면 바로 타겟 주소가 5000임을 정확하게 알 수 있으며, R1이 다른 값을 갖고 있는 경우도 마찬가지가 된다. 이를 확장해서 생각해 보면 이 레지스터 R1과 데이터 종속 관계를 갖는 명령을 거슬러 올라가 보면 명령 3의 R19의 변화에 따라 이와 연동하여 명령 12의 R1이 결정된다는 것을 쉽게 알 수 있다. 따라서 명령 3의 R19의 값을 알면 바로 이에 대응하는 명령 12의 타겟 주소를 미리 정확하게 알 수 있게 되는 것이다. 프로그램 수행 과정 중에서 명령 10은 2개의 레지스터 R1과 R2를 사용하여 그 결과가 생성된다. 만약 이 두 개의 레지스터 값이 모두 변한다고 하면 위에서 언급한 것과 같은 특성을 이용하기 어렵게 된다. 그러나 다행스럽게도 프로그램 분석 결과, 항상 그 중의 하나는 반드시 상수로 고정되어 있다. 주어진 어셈블리 코드의 예에서 R1은 명령 9에서 어떤 상수 값으로 일정하다. 그러므로 명령 3의 R19 값을 미리 알고 있으면, 이와 예측하려는 간접 분기인 명령12의 분기 주소와 연계하여 그 타겟 주소를 정확하게 미리 알 수 있게 되는 것이다. 비록 이 제어 흐름 중간에 있는 조건 분기 명령 7의 결과로 그 흐름이 바뀐다할지라도 전혀 문제가 되지 않는다. 왜냐하면 미리 파악해 둔 R19의 값을 사용하지 않으면 그만이고, 나중에 명령 3이 다시 수행될 때 새로운 값을 알면 되기 때문이다. 명령 2와 3은 메모리로부터 필요한 값을 적재해 오기 때문에 데이터 종속 관계를 이를 넘어 분석하는데는 세심한 주의가 필요하다. 왜냐하면 간접 분기의 타겟 주소

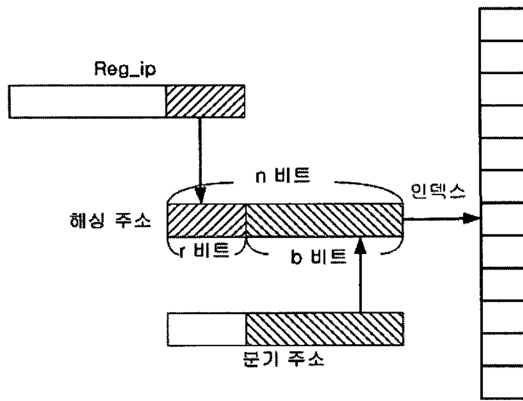


그림 8. concatenate 방식을 사용한 인덱싱
Fig. 8. Indexing by concatenation.

를 메모리로부터 적재해오는 명령 11이 적재해 오는데 사용하는 메모리 주소에는 간접 분기의 타겟 주소가 저장되어 있고, 이는 변하지 않고 하나의 값으로 고정되어 있는데, 명령 2와 3의 경우는 반드시 그렇다고 확신할 수 없기 때문에 컴파일러의 보수적인 관점을 유지해야 한다. 만약 그 저장 값이 변하지 않는다는 보장이 선결되지 않는 한 더 이상의 데이터 종속 관계를 찾는 것은 무의미한 것이 된다. 본 논문에서는 간단한 종속 관계 분석을 위해 타겟 주소를 적재해 오는 명령 이외의 적재 명령은 컴파일러의 보수적인 관점에 따르고, 데이터 종속 관계는 하나의 분기 명령을 넘는 범위로 한정시켰다.

본 논문은 이러한 특성을 이용한 간접 분기의 타겟 주소를 정확하게 예측할 수 있는 메카니즘을 그림 7과 같이 제안한다. 먼저 수행할 프로그램 코드에 해당 간접 분기와 이러한 데이터 종속 관계를 갖는 명령임을 밝히기 위한 구분을 해둔다. 이러한 구분을 위해서는 기존의 명령의 레지스터 명시 부분에 한 비트만 추가시켜주면 되므로 부담이 되지 않는다. 본 논문에서는 심플스칼라의 annotation field를 이용하여 이를 표현해 놓았다.[18] 이 명령을 디코드하여 수행하면서 명시된 레지스터의 내용을 Reg_ip로 불러온다. 간접 분기의 타겟 예측을 위해 해당 간접 분기와 데이터 종속적인 레지스터 값을 동적으로 저장하는 Reg_ip는 기존의 2-단계 예측기의 패턴 이력 레지스터와 유사한 역할을 담당하는 것이다. 이 Reg_ip와 간접 분기의 주소를 사용하여 해싱 함수에 의해 예측 분기 타겟 주소를 저장해 놓은 간접 분기 타겟 캐시를 인덱스하기 위한 인덱스 주소를 생성한다. 해싱 함수는 두 개의 값을 concatenate하거나 xor를 시켜 사용할 수 있다. 이들에 대한 상세한 성능 비교는 다음 절에서 다루기로 한다.

V. 성능 분석

본 논문에서 제안한 예측기의 성능 측정을 위해서 심플스칼라 3.0 툴셋[18]의 분기 예측을 위한 시뮬레이터인 sim-bpred에 전절에서 보인 그림 7과 같이 제안한 예측기를 구현하여 평가하였다. 시뮬레이션을 위해 3절에서 언급한 표 1에서와 같은 SPEC95int 벤치마크와 이를 위한 입력 데이터 셋을 사용하였다.

먼저 그림 8에서 보는 바와 같이 간접 분기 주소와 이와 데이터 종속적인 관계를 갖는 레지스터 값과의 상호 연관 관계를 이용하기 위해 이 두 값을 concatenate하여 얻을 수 있는 성능을 살펴본다. 간접 분기 타겟 캐시의 크기가 2^n이라면 이에 필요한 인덱스는 n 비트가 된다. 따라서 Reg_ip의 일부 r 비트와 분기 주소 중의 일부 b 비트를 concatenate할 때, $r + b = n$이 되어야 한다. Reg_ip와 분기 주소의 일부를 사용하는 경우, LSB로부터 시작한 비트들을 사용하는 가장 효과적인 것으로 나타났다. 그러나 분기 주소는 4 바이트 기계인 경우에는 2 비트의 LSB는 제외시켜야 한다. 심플스칼라에서는 annotation field를 포함하여 명령어가 3 바이트로 구성되어 있기 때문에 3 비트의 LSB를 제외시킨다. r이 0이면 기존의 BTB와 같이 분기 주소로만 간접 분기 타겟 캐시를 인덱싱하게 되는 것이고, r이 n이 되면 분기주소는 사용하지 않고 인덱싱을 위해 Reg_ip 값만을 사용하는 것이다.

그림 9는 벤치마크 중 gcc의 경우로, 직접 사상의 간접 분기 타겟 캐시의 다양한 크기에 대해서 Reg_ip의 비트 수와 분기 주소의 비트 수를 가변시키면서 concatenate하여 얻은 예상 정확도를 나타낸 것이다. 그림에서 알 수 있듯이 Reg_ip와 분기 주소의 비트 수를 대략 절반씩 사용하여 concatenate하였을 때 가장 높은 예측 정확도를 보인다. 다른 벤치마크 프로그램도 거의 유사한 특성을 보이기 때문에 그 결과는 특별히 나타내지 않았다. 모든 데이터 결과를 지면 관계상 보일 수 없어서, 그림 10에 1K 엔트리의 직접 사상의 간접 분기 타겟 캐시에 대해 각 벤치마크 프로그램이 갖는 정확도를 대표적으로 나타내었다. 분기 주소만 사용(x축 값이 0)한 경우나 Reg_ip만 사용(x축 값이 10)한 경우는 그 두 가지 값을 연결하여 사용한 경우보다도 결과가 훨씬 나쁘다는 것을 확인할 수 있다. 각각의 벤치마크는 Reg_ip 중에서 10 비트로 구성된 인덱스의 절반인 5 비트를 포함시켰을 때 최상의 예측 정확도를 보였다.

표 3은 동일한 엔트리마다 간접 분기 타겟 캐시를 구

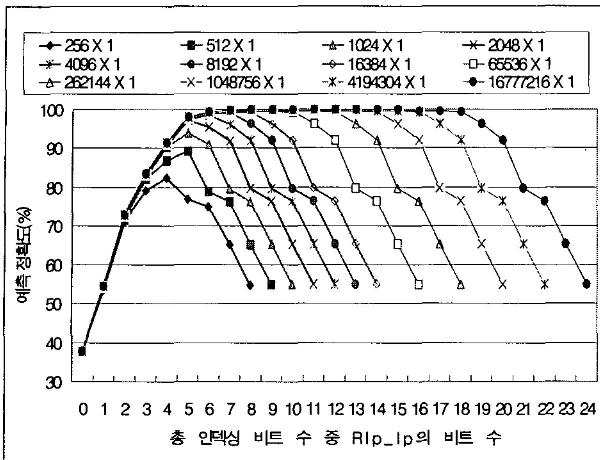


그림 9. concatenate 방식을 사용하였을 때, 인덱싱에 포함되는 Reg_ip의 비트 수에 따른 예측 정확도의 변화율

Fig. 9. Prediction Accuracy rates corresponding to bits of Reg_ip in the indexing in the case of using concatenation scheme.

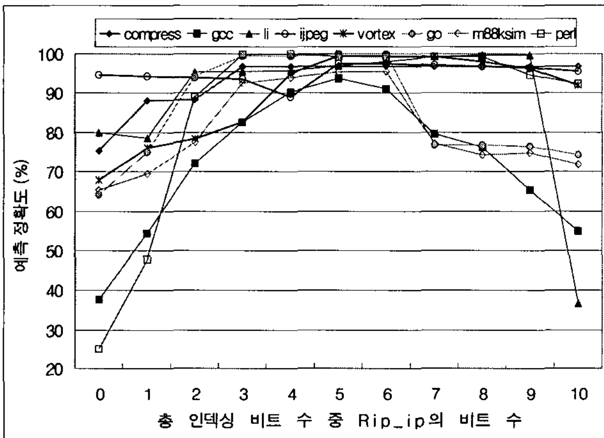


그림 10. 1024 × 1의 간접 분기 타겟을 사용하였을 때, 각 벤치마크 프로그램이 갖는 예측 정확도

Fig. 10. Prediction Accuracy rates of each benchmark program in the case of using 1024 × 1 indirect branch target cache.

성할 수 있는 4 가지의 집합-연관 사상에 따라 갖게 되는 예측 정확도를 보여준다. 각 벤치마크에 대해 2개의 항목을 나타내었는데, 첫 번째 항목의 위에 나타낸 값이 concatenate 방식으로 인덱싱하는 경우의 예측 정확도이고 그 아래쪽에 나타낸 값은 사용한 총 인덱싱 비트 중에서 Reg_ip가 차지하는 비트 수로서, 각 집합-연관 사상에서 가장 정확도가 높은 경우만을 나타내었다. 16K 크기의 간접 분기 타겟 캐시를 사용한 경우의 정확도는 평균 99.02%로 직접 분기의 타겟 예측을 위해 사용하는 BTB에 근접하는 아주 높은 정확도를 보인다. 또한 16K 이상의 캐시를 사용하더라도 예측 정확도의 변화는 거의 무시할 정도로 미미하였다. 표에서 예측 정확도가 100.00%로 나타나는 것은 소수점 3자리 이하를 반올림한 결과로서 예측율이 99.995% 이상이 된다는 의미이다. 1K의 그리 크지 않은 규모를 사용한다하더라도 평균 정확도가 98.52%에 달한다. 이는 다른 논문들에서 제안한 방식을 적용하고, 무제한의 자원을 할당한다하더라도 약 95%의 정확도밖에 거두지 못하고, 4 갈래 연관 사상으로 구성된 8K 및 1K 엔트리의 예측기가 각각 94.05%와 91.02%의 정확도를 보이는 것과 비교해 보면 본 논문이 제안한 예측기의 성능이 매우 우수하다는 것을 알 수 있다.[13, 17] SPEC95int 벤치마크 중에서 m88ksim과 compress가 다른 벤치마크에 비해 상대적으로 정확도가 낮은데, 그 이유는 이를 제외한 다른 벤치마크들의 간접 분기당 수행 회수가 약 6,200 ~ 427,300에 이르는데에 반해, 이 벤치마크들의 간접 분기당 수행 횟수는 각각 100 및 44 밖에 되지 않는다. 즉, 간접 분기 타겟 버퍼에 저장된 학습 내용을 충분히 이용할 기회를 갖지 못하기 때문에 발생하는 자연스러운 결과이다. 이를 제외한 1K 크기의 정확도를 고려해 보면 각 집합-연관 사상에 대한 평균 예측율이 99.02%에 이르게 된다. 한편, 가장 높은 정확도를 갖기 위해 인덱싱에 사용하는 총 비트 중에서 Reg_ip가 차지하는 비트 수는 절반 보다 1 비트 정도가 많은데, 그 주된 이유는 li가 상대적으로 Reg_ip의 비트 수를 많이 사용하기 때문이다. li는 다형태의 타겟을 갖는 간접 분기 수는 많지 않은 반면, 한 간접 분기당 나타나는 타겟 수가 많은 특성을 보인다. 집합-연관 사상을 사용할 때 갈래 수가 많아짐에 따라 예측 정확도도 좋아진다. 그러나 그 증가율이 그다지 크지 않기 때문에 갈래 수가 많아질수록 캐시의 구조가 복잡해지고 액세스 시간도 길어지는 단점을 고려한다면, 가장 단순한 구조의 직접 사상 방식을 선택하는 것도 좋을 것이다. 특히 집합-연관 사상을 사용하는 경우에 인덱싱에 포함되는 Reg_ip의 비트 수는 인덱싱 비트 수의 절반이 아니라, 간접 분기 타겟 캐시의 총 엔트리 비트 수의 절반 정도라는 흥미로운 사실도 발견할 수 있다.

표3의 각 벤치마크의 두 번째 항목은 인덱스를 위한 해싱 함수로 분기 주소와 Reg_ip를 xor하는 방식을 사용한 경우의 예측 정확도를 나타내고 있다. 캐시의 크기에 상관없이 상당히 고른 결과를 보인다. 갈래 수가 작을 경우에는 xor 해싱 방식이 concatenate 해싱 방식에 비해 약간 성능이 떨어지는 반면, 갈래 수가 많아지게 되면 오히려 그 결과가 반대가 된다. 이는 같은 크기

표 3. 다양한 간접 분기 타겟 캐시의 크기에 대한 concatenate 방식 및 xor 방식의 예측 정확도
Table 3. Prediction accuracy of concatenate and xor scheme for varying sizes of indirect branch target cache.

	간접 분기 타겟 캐시의 크기															
	256				1024				4096				16384			
	집합-연관 사상 (사용한 총 인덱싱 비트 수)															
	256×1 (8)	128×2 (7)	64×4 (6)	32×8 (5)	1k×1 (10)	512×2 (9)	256×4 (8)	128×8 (7)	4k×1 (12)	2k×2 (11)	1k×4 (10)	512×8 (9)	16k×1 (14)	8k×2 (13)	4k×4 (12)	2k×8 (11)
bench-mark	concatenate 방식의 예측 정확도(%) (총 인덱싱 비트 중 Rip_IP의 비트 수)															
	xor 방식의 예측 정확도(%)															
go	99.48	99.84	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95
	4	6	5	5	5,6	5~7	5~7	5~7	5~8	5~9	5~9	5~9	5~10	5~11	5~11	5~11
	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95	99.95
m88ksim	93.85	93.97	95.36	95.19	95.65	95.69	95.69	95.69	95.69	95.69	95.69	95.69	95.69	95.69	95.69	95.69
	4	4	6	5	6	6	6	6	6	6	6	6	6	6	6	6
	94.39	95.36	95.69	95.40	95.19	95.48	95.56	95.61	95.48	95.48	95.48	95.48	95.56	95.56	95.56	95.56
gcc	82.25	88.91	93.97	95.22	93.76	96.80	98.28	98.67	98.55	99.21	99.49	99.64	99.35	99.82	99.85	99.86
	4	5	5	5	5	5	6	6	6	6	7	7	7	7	7	7
	94.45	97.56	97.97	96.97	97.82	99.23	99.44	99.46	99.10	99.84	99.91	99.95	99.70	99.95	99.95	99.95
compress	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86
	3~8	3~7	3~6	3~5	3~10	3~9	3~8	3~7	3~12	3~11	3~10	3~9	3~14	3~13	3~12	3~11
	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86	96.86
li	99.37	99.46	98.12	96.97	99.58	99.67	99.67	99.46	99.90	99.99	99.67	99.67	99.90	100.00	100.00	100.00
	7	7	6	5	9	8,9	8	7	11	11	8~10	8,9	11~13	11,12	11,12	11
	99.65	99.46	98.12	96.97	99.65	99.67	99.67	99.46	100.00	100.00	99.67	99.67	100.00	100.00	100.00	100.00
ijpeg	96.20	96.32	96.50	96.37	97.59	97.76	98.18	98.12	99.21	99.68	99.96	99.96	99.95	99.92	99.96	99.96
	5	5	6	5	6	6	6	6	5	5	5	5,6	5	6	5~7	5~8
	98.44	99.22	99.47	99.90	99.38	99.96	99.96	99.96	99.85	99.96	99.96	99.96	99.94	99.96	99.96	99.96
perl	99.30	99.96	99.98	99.96	99.93	100.00	100.00	100.00	99.97	100.00	100.00	100.00	100.00	100.00	100.00	100.00
	4	5	6	5	4	7	7,8	7	6	7~9	7~10	7~9	7,8	7~11	7~11	7~11
	99.93	99.98	99.98	99.96	99.95	100.00	100.00	100.00	99.97	100.00	100.00	100.00	99.99	100.00	100.00	100.00
vortex	99.03	99.57	99.76	99.81	99.54	99.88	99.88	99.91	99.83	99.92	99.92	99.96	99.91	99.96	99.96	99.96
	5	5	5	5	6	6	6	7	5	8	8	9	7	9	9,10	9,10
	99.21	99.75	99.88	99.83	99.81	99.96	99.92	99.91	99.81	99.96	99.96	99.96	99.96	99.96	99.96	99.96
평균	95.79	96.86	97.56	97.54	97.98	98.22	98.56	98.58	98.75	98.91	98.94	98.97	98.95	99.02	99.03	99.03
	4.8	5.3	5.4	4.9	6.0	6.3	6.3	6.3	6.6	7.3	7.3	7.1	7.6	8.1	8.1	8.0
	97.86	98.52	98.49	98.23	98.58	98.89	98.92	98.40	98.88	99.00	98.97	98.99	99.00	99.03	99.02	99.02

의 캐시를 사용할 때, 갈래 수가 많아질수록 인덱스에 사용되는 비트 수가 적어지므로 캐시내에서 불필요한 간섭이 발생하기 때문이다. 소규모의 캐시를 사용하는 경우에 xor 방식을 사용하는 것이 concatenate 해싱 방식을 사용하는 것보다 훨씬 유리해진다.

VI. 결 론

간접 분기의 타겟은 동적으로 수시로 바뀌는 다형태의 특성을 갖기 때문에 미리 예측하기가 매우 어렵다. 본 논문에서는 이러한 간접 분기의 타겟을 예측하기 위한 기존의 경로 이력 기반의 예측기와 근본적으로 다른 전혀 새로운 예측기를 제안하였다. 즉, 간접 분기 명령과 이와 데이터 종속적인 관계를 갖고 있는 이 간접 분기 명령 보다 훨씬 앞서 수행되는 명령어의 레지스터 내용을 결합하여 간접 분기의 타겟을 예측하는 메카니즘으로써, 실험을 통한 결과는 기존의 다른 어떤 예측기보다도 월등히 좋은 성능을 나타내었으며, 간접 분기 타겟을 거의 완벽하게 예측할 수 있게 되어 직접 분기의 타겟만을 예측하는데 사용되는 BTB의 예측 정확도에 버금가는 정확도를 보였다.

4 갈래 연관 사상을 사용할 때, 16K 엔트리의 간접

분기 타겟 캐시가 갖는 예측 정확도는 concatenate 방식의 해싱 함수를 적용하면 평균 99.03%이고, 4K 및 1K의 경우는 각각 98.94%와 98.56%의 정확도를 보였다. 직접 사상의 간접 분기 타겟 캐시가 갖는 예측 정확도는 4K 및 1K 엔트리에 대하여 각각 98.75%와 97.98%를 나타내었는데, 복잡한 구조의 집합-연관 사상 구조에 비해 성능이 크게 떨어지지 않음을 보였다. 또한 제안된 예측기에 xor 방식의 해싱 함수를 적용하였을 때 얻게 되는 예측 정확도는 타겟 캐시의 크기에 상관없이 상당히 고른 결과를 보였다. 256 엔트리의 경우, concatenate 방식으로 얻는 정확도가 95.79%인데에 반해 xor 방식은 97.86%의 정확도를 얻을 수 있었다. 소규모의 캐시를 사용하는 경우에 xor 방식을 사용하는 것이 concatenate 해싱 방식을 사용하는 것보다 훨씬 유리할 것이다.

참 고 문 헌

[1] D. W. Wall, "Limits of Instruction-Level Paralle-lism", 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 176-188, Santa Clara, U.S.A., Apr. 1991.

[2] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design", IEEE Computer Magazine, 17(1), Jan. 1984.

[3] C. Perleberg and A. J. Smith, "Branch Target Buffer Design and Optimization", IEEE Transac-tions on Computers, 42(4), pp. 396-412, Apr. 1993.

[4] B. Calder and D. Grunwald, "Fast & Accurate Instruction Fetch and Branch Prediction", 21th Int'l Symp. on Computer Architecture, pp. 2-11, Chicago, U.S.A., Apr. 1994.

[5] T. Y. Yeh and Y. N. Patt, "Alternative Implemen-tation of Two Level Adaptive Training Branch Predictions", 19th Int'l Symp. on Computer Architecture, pp. 124-134, Gold Coast, Australia, May 1992.

[6] S. McFarling, "Combining Branch Predictions", TN 36, DEC-WRL, June 1993.

[7] E. Sprangle, R. S. Chappell, M. Alsup and Y. N. Patt, "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference", 24th Int'l Symp. on Computer Architecture, pp. 284-291, Denver, U.S.A., June 1997.

[8] C. -C. Lee, I. K. Chen and T. N. Mudge, "The Bi-Mode Branch Predictor", 30th Int'l Symp. on Microarchitecture, pp. 4-13, Research Triangle Park, U.S.A, Dec. 1997.

[9] A. N. Eden and T. N. Mudge, "The Yags Branch Predictor", 31th Int'l Symp. on Microarchitecture, pp. 69-77, Dallas, U.S.A., Dec. 1998.

[10] D. A. Jimenes and C. Lin, "Dynamic Branch Prediction with Perceptron", 7th Int'l Symp. on High Performance Computer Architecture, pp. 197-206, Monterrey, Mexico, Jan. 2001.

[11] A. Seznec, S. Felix, V. Krishnan and Y. Sazei-des, "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor", 29th Int'l Symp. on Computer Architecture, pp. 295-306, Anchora-ge, U.S.A., May 2002.

[12] P. Y. Chang, E. Hao and Y. N. Patt, "Target Prediction for Indirect Jumps", 24th Int'l Symp. on Computer Architecture, pp. 274-283, Denver, U.S.A., June 1997.

[13] K. Driesen and U. Holzle, "Accurate Indirect Branch Prediction", 25th Int'l Symp. on Computer Architecture, pp. 167-178, Barcelona, Spain, July 1998.

[14] K. Driesen and U. Holzle, "The Cascaded Predictor: Economical and Adaptive Branch Target Prediction", 31th Int'l Symp. on Micro-architecture, pp. 249-258, Dallas, U.S.A., Dec. 1998.

[15] K. Driesen and U. Holzle, "Multi-Stage Cas-caded Prediction", 5th Int'l Euro-Par Conf. on Parallel Processing, pp. 1312-1321, Toulouse, France, Aug. 1999.

[16] J. Kalamatianos and D. R. Kaeli, "Improving the Accuracy of Indirect Branch Prediction via Branch Classification", Technical Report ECE-CEG-98-008, Northeastern University, Boston, Mar. 1998.

[17] J. Kalamatianos and D. R. Kaeli, "Predicting Indirect Branches via Data Compression", 31th Int'l Symp. on Microarchitecture, pp. 272-281, Dallas, U.S.A., Dec. 1998.

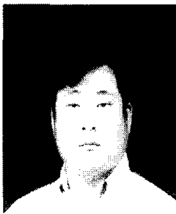
[18] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report CS-RT-97-1342, University of Wisconsin, Madison, June 1997.

[19] R. Nair, "Dynamic Path-Based Branch Correla-tion", 28th Int'l Symp. on Micro architecture, pp. 15-23, Ann Arbor, U.S.A., Nov. 1995.

[20] D. R. Kaeli and P. G. Emma, "Branch History Table Prediction of Moving Target Branches due to Subroutine Returns", 18th Int'l Symp. on Computer Architecture, pp. 43-42, Toronto, Canada, May 1991.

[21] A. Roth, A. Moshovos and G. S. Sohi, "Improving Virtual Function Call Target Predic -tion via Dependence-Based Pre- Computation", 13th Int'l Conf. on Super computing, pp. 356 -364, Rhodes, Greece, June 1999.
[22] O. J. Santana, A. Falcon, E. Fernandez, P. Medina, A. Ramirez and M. Volero, "A Compre -hensive Analysis of Indirect Branch Prediction", 4th Int'l Symp. on High Performance Computing, pp. 133-145, Kansay Science City, Japan, May 2002.

저 자 소 개



백 경 호(학생회원)
1999년 순천향 대학교 전자공학과 공학사
2001년 순천향 대학교 전자공학과 공학석사
2001년 3월~순천향 대학교 전자공학과 박사과정
<주관심분야: 컴퓨터 구조>

김 은 성(종신회원)
1985년 한양대학교 전자공학과 공학석사.
1989년 한양대학교 전자공학과 공학박사.
2000년~2002년 미국 노스이스턴 대학교 객원교수.
1989년~현재 순천향대학교 정보기술공학부 교수
<주관심분야: 고성능 컴퓨터, 마이크로프로세서 응용, Ad-Hoc 및 센서 네트워크>