논문 2006-43CI-4-1

효율적인 데이터 종속 기반의 간접 분기 예측기

(Efficient Indirect Branch Predictor Based on Data Dependence)

백 경 호*, 김 은 성**

(Kyoung-Ho Paik and Eun-Sung Kim)

요 약

간접 분기 명령은 현대적인 고성능 프로세서의 ILP를 제한하는 가장 심각한 장애 요인 중 하나이다. 다른 분기 명령들과는 다르게 간접 분기는 그 타겟 주소가 동적으로 다형태로 변하기 때문에 이를 예측하기 매우 어려우며, 투기적 실행 방식을 사용하는 대부분의 현대적인 고성능 프로세서에서는 예측이 잘못되는 경우에 많은 수행 사이클 지연이 일어나게 되어 프로세서의 성능이 크게 떨어지게 된다. 우리는 예측 정확도가 아주 뛰어난 새로운 개념의 간접 분기 예측 방식 즉, 간접 분기 명령과 이와 데이터 종속 관계를 가진 이 명령어 보다 훨씬 앞서 수행되는 명령어의 레지스터 내용을 결합시켜 간접 분기의 타겟을 예측해내는 방식을 제안하였다. 1K의 예측기를 사용하는 경우에 98.92%의 예측 정확도를 보이고, 8K의 크기를 사용하면 거의 완벽한 99.95%의 정확도를 보인다. 그러나 지금까지 제안된 모든 예측기가 그러하듯이 예상 타겟 주소와 함께 앨리어싱 문제를 완화시키기 위한 태그를 저장하기 위한 하드웨어 오버헤드가 크다는 단점을 안고 있다. 그러므로 본 논문에서는 예측 정확도의 손실없이도 예측기의 하드웨어 오버헤드를 최소한으로 줄이는 방법을 제안한다. 실험 결과로써 태그 저장에 따른 하드웨어를 성능 손실 없이 약 60%를 줄일 수 있으며, 0.1%의 손실을 감수하면 약 80%까지 줄일 수 있다. 또한 부분 타겟 저장으로 인한 성능 손실 없이 타겟 주소 저장에 따른 하드웨어를 약 35% 절약할 수 있으며, 1.11%의 손실을 감수하면 약 45%까지 절약할 수 있다.

Abstract

The indirect branch instruction is a most substantial obstacle in utilizing ILP of modern high performance processors. The target address of an indirect branch has the polymorphic characteristic varied dynamically, so it is very difficult to predict the accurate target address. Therefore the performance of a processor with speculative methodology is reduced significantly due to the many execution cycle delays in occurring the misprediction. We proposed the very accurate and novel indirect branch prediction scheme so called data-dependence based prediction. The predictor results in the prediction accuracy of 98.92% using 1K entries, and 99.95% using 8K. But, all of the proposed indirect predictor including our predictor has a large hardware overhead for restoring expected target addresses as well as tags for alleviating an aliasing. Hence, we propose the scheme minimizing the hardware overhead without sacrificing the prediction accuracy. Our experiment results show that the hardware is reduced about 60% without the performance loss, and about 80% sacrificing only the performance loss of 0.1% in aspect of the tag overhead. Also, in aspect of the overhead of storing target addresses, it can save the hardware about 35% without the performance loss, and about 45% sacrificing only the performance loss of 1.11%.

Keywords: ILP, Polymorphism, Indirect Branch Prediction, Minimizing Hardware Overhead

I. 서 론

고성능의 프로세서는 명령어 수준의 병렬성(ILP)을 최대한 이용하고자 꾸준히 발전되어 왔다.[1~3] 특히 프로그램의 제어 흐름을 변경시키는 분기 명령은 분기될 타겟 주소와 분기 조건 결과 생성의 지연으로 인해 ILP를 유용하는데 주요 장애 요인이 되기 때문에, 분기 방향 및 분기 타겟을 예측하고 해당 타겟 주소의 명령어들을 미리 인출하여 투기적으로 수행하는 방식을 적용하여 연속적인 명령어 흐름을 최대한 연속적으로 유지시켜 줌으로써 성능 향상을 꾀하여 왔다. 그러나 예측

* 학생회원, ** 종신회원, 순천향대학교 정보기술공학부
(Division of Information Technology Engineering, Soonchunhyang University)

※ 본 논문은 순천향대학교 연구과제(20050059) 지원으로 수행되었음.

접수일자: 2006년4월20일, 수정완료일: 2006년6월29일

이 잘못되는 경우에는 투기적으로 수행한 명령들을 취소하고 원상태로의 복구가 필요하기 때문에 성능에 나쁜 영향을 크게 미치게 되어 분기 명령의 방향 및 타겟의 정확한 예측은 투기적 실행 방식에서는 고성능을 보장하기 위해 절대적으로 필요하다.

조건 분기 명령의 방향 예측을 잘못하는 경우에 초래되는 방향 예측 실패 페널티가 매우 크기 때문에, 이에 대한 많은 연구들로 인해 아주 높은 예측 정확도가 가능하게 되었다.[4~8] 타겟 주소 예측은 통상적으로 분기 명령이 인출될 때 행해지고, 해독 단계에서 실제로 결정되는데, 예측이 잘못되면 이에 따른 명령 인출 실패 페널티가 주어진다.[9] 직접 분기 명령에 대한 명령 인출 실패 페널티는 인출에서 해독 단계까지의 사이클 구간이 1~2 사이클로 짧기 때문에 방향 예측 실패 페널티에 비해 부담이 적다. 그러나 레지스터를 참조하여 타겟 주소가 결정되는 간접 분기 명령을 투기적으로 이슈하고 수행하려면, 이 간접 분기를 인출한 이후로 자신의 타겟 주소를 알게 될 때까지 이전 명령들과의 데이터 종속 관계로 인하여 많은 사이클 지연이 일어나기 때문에 타겟 예측 실패 페널티가 매우 커지게 된다.[10] 또한 예측을 하지 않는 통상적인 방법으로는 이슈 대역폭이 크게 떨어지게 되는데, 완벽한 간접 분기 예측으로는 IPC를 평균 10.8%까지 향상시킬 수도 있다.[11] C와 같은 절차적 언어에서 switch/case 문과 동적으로 링크되는 호출 및 함수 포인터를 통한 호출을 위해 간접 분기가 사용되고, C++나 자바와 같은 객체지향 언어에서 동적으로 디스패치되는 가상 함수를 호출하거나 가상머신 명령어를 인터프리트하는데 간접 분기가 사용된다. 특히 이러한 객체지향 언어의 사용이 꾸준히 증가하는 추세이기 때문에 간접 분기가 프로세서의 성능에 미치는 영향은 더욱 커지게 될 것이다.[12~14]

전통적으로 분기 타겟 예측을 위하여 BTB를 사용하여 왔으나 특히 간접 분기의 타겟 주소의 예측 정확도가 크게 떨어지므로, 이를 향상시키려고 하는 많은 연구들이 꾸준히 진행되어 왔다. 지금까지 발표되었던 간접 분기 타겟 예측 방식의 대부분은 간접 분기 명령의 타겟 주소가 동적으로 달라지는 특성이 이전에 수행되었던 분기 타겟들의 이력들과 상관관계가 있는 것으로 파악하여 이에 근거한 이른바 경로 기반 방식으로 예측 정확도를 높일 수 있도록 하고 있지만, 분기 타겟 모두가 이러한 상관관계에 따른 예측 가능한 패턴을 보이지 못하는 경우가 상당하게 존재하는 한계가 있다.[15,16] 그러므로 우리는 이러한 한계를 뛰어넘는 새로운 간접 분기 타겟 주소 예측 방식을 제안하였다.[17] 이 예측 방식은 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.

분기 방향 예측기가 예측하려는 분기 방향의 정보를 1~2 비트로 저장하고 이를 액세스하여 방향 예측에 사용하는 반면, BTB를 포함한 모든 분기 타겟 예측기는 예측하려는 정보가 타겟 주소이므로 저장해야 할 데이터량이 많아진다. 더군다나 분기 방향 예측기는 제한된 크기의 패턴 이력표를 사용할 때 서로 다른 분기 명령이 동일한 인덱스 정보를 공유하여 발생하게 되는 앨리어싱(aliasing) 문제로 인해 예측에 해를 입기도 하지만 득이 되기도 하고 또는 아무런 영향도 받지 않게 되기도 하지만, 분기 명령에는 대부분 각기 고유의 타겟 주소가 주어지기 때문에 분기 타겟 예측기에서의 앨리어싱 발생은 바로 예측 실패로 이어지게 된다. 집합 연관 방식을 사용하여 앨리어싱 문제를 완화시킬 수 있지만, 이러한 경우에 타겟 주소와 더불어 태그를 저장해야 하므로 분기 타겟 예측기의 하드웨어 부담이 분기 방향 예측기에 비해 아주 커지게 된다.

본 논문에서는 간접 분기 타겟 예측기의 하드웨어 부담을 줄이기 위한 여러 가지 방법들을 제안된 예측기에 적용하고, 이에 따른 예측 정확도에 미치는 영향을 분석한다. 앨리어싱 위험을 감수하는 태그를 사용하지 않는 방법과 앨리어싱을 가급적 피하려는 여러 가지 인덱싱 방식에 따른 태그를 사용하는 방식들을 상세히 조사하고 분석한다. 또한 성능에 영향을 주지 않으면서 태그와 타겟 주소를 최소한으로 줄여서 사용할 수 있는 방법도 제안하고 그 특성을 분석한다. 본 논문은 다음과 같이 구성되었다. Ⅱ장은 관련 연구에 대한 것으로서, 분기 예측에 관련된 대표적인 연구들, 특히 간접 분기 예측에 관한 기존의 연구들에 대해 다룬다. Ⅲ장에서는 간접 분기와 관련된 데이터 종속관계의 이론적 배경을 설명한다. Ⅵ장에서는 Ⅲ장의 이론을 바탕으로 제안된 간접 분기 예측기에 대한 설명과 예측기의 여러 가지 인덱싱 방식에 따른 예측 정확도에 미치는 영향들을 평가하고 비교 분석한다. Ⅴ장에서는 제안된 방식과 여러 인덱싱 방법에 따른 하드웨어 오버헤드를 줄일 수 있는 방법들을 모색한다. 예측기에 저장되는 태그와 타겟 주소의 비트 수를 줄여 하드웨어 부담을 최소화시키고, Ⅵ장에서 결론을 맺는다.

Ⅱ. 관련 연구

전통적으로 사용하는 예측기 구조로는 그림 1과 같은 분기 타겟 버퍼(BTB)가 있다. 분기 타겟 테이블의 크기가 한정되어 있기 때문에 테이블의 인덱스로 분기 주소의 하위 일부분을 테이블의 엔트리 수에 맞는 만큼을 선택하여 사용한다. 그리고 서로 다른 분기가 동일한 엔트리를 사용하는 것을 방지하기 위해 나머지 상위 부분의 분기 주소를 태그로 테이블 엔트리에 저장하고, 이 패턴에 대해 가장 최근에 결정되었던 타겟 주소도 저장한다. 분기 주소의 일부분을 테이블의 인덱스로 사용하기 때문에 발생할 수 있는 앨리어싱 문제를 더욱 완화시키기 위해 집합 연관 사상의 테이블을 구성할 수도 있으며, 앨리어싱으로 인한 불이익을 감수하면서 하드웨어의 부담을 줄이기 위해 태그가 없는 테이블을 구성하여 사용하기도 한다. BTB는 분기 당 하나의 타겟 주소만을 저장할 수밖에 없기 때문에 동적으로 타겟 주소가 수시로 변하는 간접 분기의 타겟을 예측하는 것이 매우 어렵게 된다. 무제한 크기의 BTB를 사용하면 직접 분기의 타겟은 거의 완벽하게 예측할 수 있다. 맨 처음 버퍼에 타겟 주소를 저장할 때를 제외하면 항상 정확하게 타겟 주소를 버퍼를 참조하여 알 수 있다는 의미이다. 그런 반면, 간접 분기의 타겟 주소는 동적인 변화로 인해 그 예측 정확도가 60.08% 밖에 되지 않는다.

BTB의 이러한 단점을 향상시키기 위해 분기 당 하나 이상의 타겟을 저장할 수 있는 2-단계 간접 분기 예측기가 제안되었다(그림 2).[15,16] 동적으로 다형태의 특성을 갖는 간접 분기의 타겟 주소는 이 분기에 이르기까지의 명령어 흐름 경로 상에 있는 이전에 수행되었던 분기 타겟들의 이력들과 상관관계가 있는 것으로 파악하여, 이 두 가지 정보를 조합하여 분기 타겟 테이블의 인덱스로 사용한다. 제한된 크기의 타겟 테이블로 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 포함되는 분기 당 타겟 주소는 그만큼 짧아진다. 또한 경로 이력 패턴과 더불어 분기 주소도 함께 인덱스를 위해 사용되어야 하므로 이들을 조합하여 인덱스로 사용하여야 한다. 인덱스를 생성해 주는 해싱 함수로는 경로 이력 패턴과 분기 주소를 XOR하는 것을 가장 보편적으로 사용한다. 이 방식은 분기 타겟 모두가 이러한 상관관계에 따른 예측 가능한 패턴을 보이지 못하는 경우가 상당하게 존재하는 한계가 있다.

타겟 주소를 예측 가능케 하는 패턴을 정확하게 알아내기가 불가능하기 때문에 확률적으로 높은 패턴을 찾아내려는 연구들이 꾸준히 진행되어 왔다. 먼저 포함되는 경로 길이에 따라 예측 정확도가 달라지는 특성을 이용하기 위해 다양한 경로 길이에 따라 각각 인덱스되는 여러 개의 예측기를 두고, 이 중에서 예측 정확도가 가장 높을 것 같은 예측기를 동적으로 선택하는 방법이 제안되었다.[18] 이 방식은 각각의 독특한 특성을 갖는 예측기마다 서로 보완적인 장점을 갖고 있기 때문에 어떤 예측기를 사용할 지를 결정하는 선택 방법이 어렵다. 또 다른 방식으로 어떤 의미있는 패턴을 좀 더 정확하게 찾아내기 위해 경로 이력에 복잡한 데이터 압축 알고리즘을 적용하여 하는 방식도 제안되었다.[19] m 차수의 PPM(Prediction by Partial Matching) 알고리즘이 적용된 m + 1 개의 Markov 예측기를 구성하여 차수가 낮은 예측기로부터 찾고자하는 패턴이 있는지를 차례로 조사하여 사용한다. 이 방식은 알고리즘 구현이 매우 복잡하고, 여러 개의 예측기로 구성되기 때문에 하드웨어 부담이 커지게 된다.

분기 특성에 따라 분류하여 예측하기 어려운 특성을 가진 간접 분기에 대해 보다 집중하여 성능을 높이려는

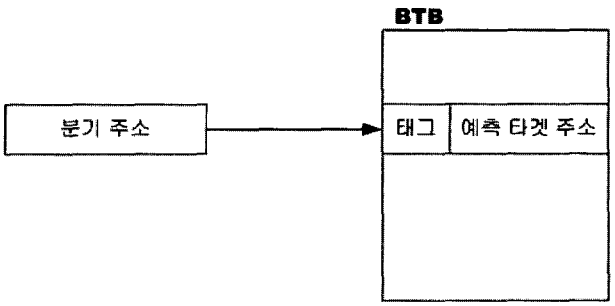


그림 1. 분기 타겟 버퍼
Fig. 1. Branch Target Buffer.

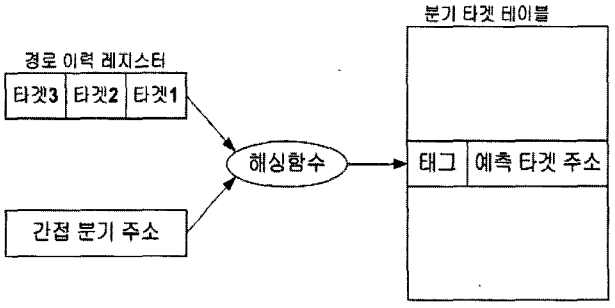


그림 2. 2-단계 간접 분기 예측기
Fig. 2. 2-level indirect branch predictor.

시도도 있다. 간접 분기 중에는 직접 분기처럼 타겟이 하나인 경우도 다수 있기 때문에, 이러한 간접 분기를 정적으로 즉, 컴파일 시에 프로파일 정보를 참고하여 분류시켜 놓았다가 이를 직접 분기와 같은 방법으로 처리하고, 타겟이 여러 개인 간접 분기에 대해서만 별도로 복잡한 예측기를 적용하는 방식이다.[20] 이러한 분류를 동적으로 하는 방식도 있다.[21] 필터링 메카니즘을 두어 예측하기 어려운 분기만을 따로 분류하고 위와 마찬가지로 좀 더 복잡하고 정교한 예측 방식을 이에 대해 적용하는 방식이다. 이러한 방식들은 여러 개의 예측기가 사용되는 방식에 비해 하드웨어 부담이 적다는 장점을 갖고 있으나 예측 정확도가 상대적으로 떨어지는 단점도 보인다.

타겟 주소를 예측 가능케 하는 패턴을 확률적인 방식으로 사용할 수밖에 없는 한계를 극복하기 위해, 간접 분기 중에 일부로서 C++ 프로그램에 존재하는 가상 함수 호출의 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.[22]

정교한 컴파일러 최적화 기법을 개발하여 C++나 자바에서 나타나는 가상 함수 호출이나 가상머신 명령어로 인한 간접 분기의 발생을 감소시키거나 부담이 상대적으로 작은 다른 분기로 대체하려는 노력들이 오랫동안 지속되어 왔다. 그러나 이러한 연구의 결과로는 간접 분기로 인한 부담을 궁극적으로 해결할 수가 없다.[23,24]

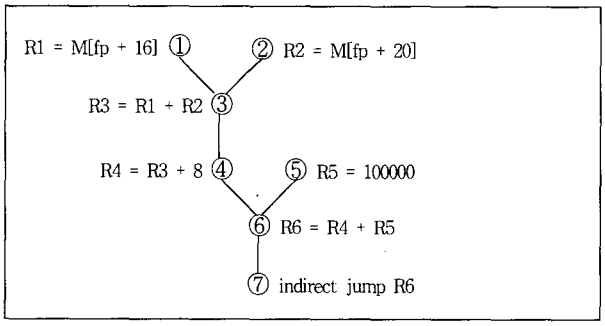
III. 데이터 종속적인 간접 분기의 특성

1. 복합 및 단순 데이터 종속관계

어떤 한 명령이 수행되려면 먼저 자신의 소스 오퍼랜드의 값이 미리 결정되어야 하고, 이러한 소스 오퍼랜드의 값은 앞서서 수행되는 일련의 명령어들의 수행 결과로 주어진다. 그림 3에서 보는 바와 같이, 명령어 *I3*의 레지스터 *R3*은 명령어 *I1*과 *I2*에서 각각 계산된 *R1*과 *R2*를 사용하여 수행이 된 후에나 결정된다. 여기서 명령어 *I3*은 명령어 *I1*과 *I2*에 대해 복합 데이터 종속적이라 정의한다. 또는 레지스터 *R3*은 *R1*과 *R2*에 대해 복합 데이터 종속적이라고 말하고, 역으로 *R1* 및 *R2*는 *R3*에 대해 상대적 상관관계를 갖는다고 말한다. 왜냐하면 *R3*은 두 가지 요인 즉, *R1*과 *R2*가 변하는 데에 따라 복합적인 영향을 받기 때문이다. 또한 명령어 *I4*는

```
I1:  lw     $1, 16($fp)     // R1 = M[fp + 16]
I2:  lw     $2, 20($fp)     // R2 = M[fp + 20]
I3:  addu   $3, $1, $2      // R3 = R1 + R2
I4:  addiu  $4, $3, 8       // R4 = R3 + 8
I5:  la     $5, $L10        // R5 = 100000
I6:  addu   $6, $4, $5      // R6 = R4 + R5
I7:  jalr   $6              // indirect jump
```

(a) 어셈블리 코드의 예



(b) 데이터 종속 그래프

그림 3. 어셈블리 코드와 이에 대응하는 데이터 종속 그래프

Fig. 3. An assembly code and the corresponding data dependence graph.

명령어 *I3*에 대해 단순 데이터 종속적이라 정의한다. 또는 *R4*는 *R3*에 대해 단순 데이터 종속적이라 말하고, 역으로 *R3*은 *R4*에 대해 절대적 상관관계를 갖는다고 말한다. 왜냐하면 *R4*는 *R3*의 변화에 의해서만 유일하게 영향을 받기 때문이다. 마찬가지로 명령어 *I6*은 명령어 *I4*에 대해 단순 데이터 종속적이 되고, 명령어 *I7*도 명령어 *I6*에 대해 단순 데이터 종속적이 된다. *R3*은 *R6*에 대해 절대적 상관관계를 갖고 있으나, *R1*이나 *R2*는 *R6*에 대해 상대적 상관관계를 갖는다. 이 말의 의미는 *R3*의 어떤 값이 계산되고 이에 따라 정해지는 *R6*의 값을 미리 알고 있다면, *R6*의 계산에 필요한 일련의 명령어 시퀀스를 수행하지 않더라도 *R3*의 값을 아는 즉시 미리 그 값을 정확하게 예측해 낼 수 있다는 것이다. 그러나 *R1*이나 *R2*는 *R6*에 대해 상대적 상관관계를 갖고 있기 때문에 *R1*이나 *R2*의 값을 알고 있다고 해도 *R6*의 값을 이에 따라 미리 알아낼 수가 없다. 왜냐하면 *R1*과 *R2*의 변화가 보수적인 관점에서 상호간에 어떤 규칙성을 갖고 있다고는 볼 수 없기 때문이다. 따라서 어느 한 레지스터와 이와 절대적 상관관계를 갖는 한 레지스터를 알고 있다면, 이 관계를 갖는 명령어 시퀀스가 한번만 수행되면 이 관계를 이용하여 절대적 상관관계를 갖는 레지스터의 값이 결정되자마자 이와 단순 종속적인 관계를 갖고 있는 레지스터의 값을 미리 정확하게 예측할 수가 있는 것이다. 그리고 간접 분기 타겟을 갖고 있는 레지스터와 절대적 상관관계를 갖고 있는

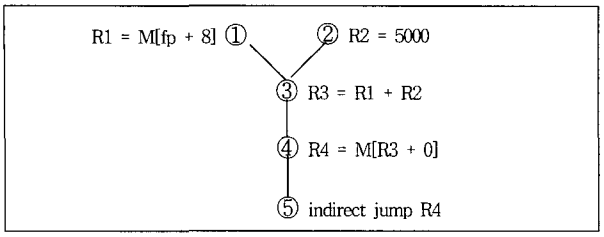


그림 4. 메모리와 관련된 레지스터 간의 데이터 상관 관계

Fig. 4. Data correlation between registers in reference to memory location.

레지스터가 여러 개인 경우에는 가급적 거리가 먼 것을 최종 절대적 상관관계를 갖는 레지스터로 선택한다.

메모리와 관련된 좀 더 복잡한 예의 데이터 종속 그래프를 그림 4에 보여준다. 이 그림에서 명령어 *I4*와 명령어 *I3*과의 데이터 종속관계를 살펴보자. 레지스터 *R4*의 레지스터 *R3*에 대한 데이터 종속관계는 메모리 주소 [*R3* + 0]에 저장된 데이터의 변화 여부에 따라 달라진다. 일반적으로 M[*R3* + 0]의 내용이 변하는지의 여부를 알기 어렵다. 왜냐하면 데이터 종속 그래프는 주로 레지스터에 대한 데이터 종속관계를 알려주고, 메모리에 대한 데이터 종속관계는 극히 보수적으로 표현하기 때문이다. 따라서 M[*R3* + 0]의 내용의 변화 여부를 정확히 알 수 없는 경우에는 그 내용이 변하는 것으로 보수적으로 가정하게 되므로 *R4*는 *R3*에 대해 단순 데이터 종속적이라 말할 수 없다. 다만 M[*R3* + 0]의 내용이 변하지 않는다는 것이 확실할 때에만 *R4*는 *R3*에 대해 단순 데이터 종속적이 되고, 이 경우에 *R3*의 값에 따라 *R4*의 내용을 미리 알아낼 수 있게 된다.

2. 간접 분기 타겟 결정을 위한 데이터 종속 특성

C와 같은 절차적 언어나 C++ 또는 자바와 같은 객체지향 언어로 작성된 프로그램에 대해 생성된 머신 코드에 포함되는 여러 가지의 간접 분기에 대해 살펴본다.

절차적 언어에서 볼 수 있는 switch/case 문과 동적으로 링크되는 호출 및 함수 포인터를 통한 호출이 간접 분기를 사용하여 구현된다. 그리고 객체지향 언어에서, 동적으로 디스패치되는 가상 함수를 호출하기 위해 간접 분기가 사용된다. 함수 호출을 위한 유효 주소는 리시버의 동적 타입에 의존하기 때문에 컴파일 시에는 알 수 없고 수행 시에나 결정된다.

먼저 switch/case 문을 위해 생성되는 간접 분기의 타겟 주소의 결정 과정은 다음과 같다.

(1) 선택 파라미터가 옵션 범위내의 값인지를 조사
(2) 점프 테이블의 주소를 계산
(3) 이 테이블에서의 인덱스를 사용할 옵셋을 계산
(4) 테이블에서 타겟 주소를 적재
(5) 간접 분기를 수행

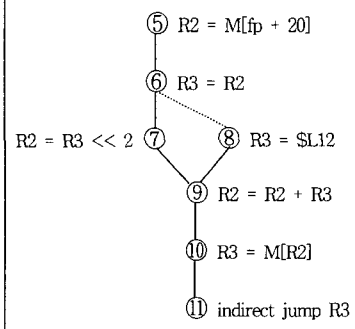
그림 5(a)는 switch/case 문을 포함한 간단한 C 소스 코드이며, 그림 5(b)는 이 코드에 대한 어셈블리 코드(심플스칼라의 MIPS)의 일부로서, 점프 테이블을 이용

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main() {
  int i, j;  char c;

  srand(time(NULL));
  for (i = 0; i < 100; i++) {
    j = rand() % 5;
    switch (j) {
      case 0:  c = 'a';  break;
      case 1:  c = 'b';  break;
      case 2:  c = 'c';  break;
      case 3:  c = 'd';  break;
      case 4:  c = 'e';  break;
    }
  }
}
```

(a) swich/case 문을 포함한 C 코드

```
        ...
I1: sw   $2, 20($fp)   //switch 문장의 5 가지의 선택 파라미터 중
I2: lw   $2, 20($fp)   //하나를 결정.
I3: sltu $3, $2, 5     //선택 파라미터 값이 옵션 수(5) 보다 크면
I4: beq  $3, $0, $L13 //switch/case 구조를 빠져 나옴.
I5: lw   $2, 20($fp)   //switch/case 문의 베이스 주소를 위한 점프
I6: move $3, $2        //테이블의 기본 옵셋을 계산.
I7: sll  $2, $3, 2     //최종 옵셋 결정(각 subcase 주소는 4 바이트).
I8: la   $3, $L12      //점프 테이블의 베이스 주소를 $3에 적재.
I9: addu $2, $2, $3    //유효 엔트리 위치 = 베이스 주소 + 옵셋
I10: lw  $3, 0($2)     //올바른 유효 명령어 주소를 적재.
I11: j   $3            //간접 분기한 위치에서 부터 해당 case에 대해
                         //수행해야 할 코드 시퀀스가 시작.
    .rdata  3
    .align  2
    .align
$L12:
    .word  $L7     //switch/case 문에 대한 점프 테이블의 베이스
    .word  $L8     //주소와 각 case에 대해 수행해야 할 코드
    .word  $L9     //시퀀스를 위한 5개의 주소를 저장
    .word  $L10
    .word  $L11
```

(b) (a)에 대응하는 어셈블리 코드의 일부



(c) (b)에 대응하는 데이터 종속 그래프의 일부

그림 5. switch/case 문과 이를 위해 생성된 간접 분기가 포함된 어셈블리 코드

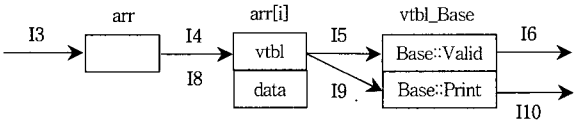
Fig. 5. Assembly code with indirect branch matching to a switch/case statement.

```
class Base
    virtual  void  Print( ) { ... }
    virtual  int   Valid( ) { ... }

class Derived : public Base
    void Print( ) { ... }
    int Valid( ) { ... }

Base * arr[S];

for (i = 0;  i < S;  i++)
    if (arr[i]->Valid( ))
        arr[i]->Print( );
```

(a) C++ 소스 코드의 일부

```
  I1: lw    $16, arr
  I2: move  $17, $0
$L12
  I3: lw    $4, 0($16)  //$4 <- 객체의 베이스 주소
  I4: lw    $3, 0($4)   //$3 <- vtbl 테이블의 베이스 주소
  I5: lw    $3, 0($3)   //$3 <- 호출되는 함수의 주소
  I6: jalr  $3          //함수 Base::Valid로 분기
  I7: bne   $2, $L13
  I8: lw    $3, 0($4)   //$3 <- vtbl 테이블의 베이스 주소
  I9: lw    $3, 4($3)   //$3 <- 호출되는 함수의 주소
 I10: jalr  $3          //함수 Base::Print로 분기
$L13
 I11: addiu $16, $16, 4
 I12: addiu $17, $17, 1
 I13: blt   $17, S, $L12
```

(b) (a)에 대응하는 어셈블리 코드의 일부



(c) (b)에 대응하는 데이터 종속 그래프의 일부

그림 6. 객체지향언어에서 가상함수를 동적으로 디스패치하기 위한 동적 바인딩 사용 예
Fig. 6. An example of dynamic binding for dynamically dispatched virtual function in object oriented languages.

한 간접 분기를 수행하여 이를 구현한다. 이 예는 5 개의 임의수를 생성하여 이 값을 switch 문의 선택 파라미터로 사용하는데, 이 값에 따라 명령어 *I11*의 간접 분기 타겟이 달라진다. 그러므로 이 간접 분기가 수행될 때마다 수시로 달라지는 타겟 주소를 기존의 경로 기반의 예측 방식으로는 미리 알기가 거의 불가능하다. 그림 5(b)의 일부를 그림 5(c)와 같이 데이터 흐름 그래프로 나타내었다. 명령어 시퀀스 *I5*에서 *I9*까지의 수행으로 필요한 타겟 주소가 저장된 점프 테이블 내의 메모리 주소를 계산해 낸다. 명령어 *I10*으로 올바른 유효 타겟 주소를 적재한 후, 명령어 *I11*은 이 타겟 주소를 사용하여 해당 case에 대해 수행해야 할 코드 시퀀스의 처음으로 분기한다. 여기서 주목해야 할 것은 점프 테이블에 저장된 각각의 타겟 주소는 변하지 않는다는 사실이다. 그림 5(c)의 데이터 흐름 그래프에서와 같이, 명령어 *I5*에서 결정된 레지스터 *R2*의 내용에 따라 명령어 *I9*의 레지스터 *R2*의 내용 즉, 올바른 타겟 주소가 저장되어 있는 점프 테이블의 해당 위치(메모리 주소)가 결정된다. 그러므로 명령어 *I5*의 *R2*와 명령어 *I11*의 *R3* 사이에는 단순 데이터 종속관계가 있기 때문에 이 관계를 이용하여 타겟이 저장되어 있는 메모리 장소를 정확하게 알 필요가 없이도 타겟 주소를 미리 정확하게 예측해낼 수가 있다. 글로벌 옵셋 테이블이나 프로시저 링키지 테이블을 통해 만들어지는 라이브러리 함수 호출과 함수 포인터의 의한 함수 호출의 특성도 switch/case 문을 위해 생성되는 점프 테이블을 이용한 간접 분기 루틴과 아주 유사하나, 이 보다 훨씬 단순하다.

객체지향 언어에서 가상 함수를 동적으로 디스패치하기 위해 동적(또는 늦은) 바인딩을 사용하는 간단한 예를 그림 6에서 나타내었다. 컴파일러는 각 클래스에 대한 가상 함수 테이블(vtbl)을 구축하고, 개개의 클래스 실체(객체)에 해당 클래스의 vtbl 테이블을 가리키는 포인터를 저장한다. vtbl의 각 메소드의 주소가 저장된 위치는 미리 정의된 옵셋으로 결정되며, vtbl에는 정해진 객체 클래스가 액세스할 수 있는 모든 메소드들의 주소를 저장해둔다. 그림 6(b)는 그림 6(a)의 소스 코드에 대한 어셈블리 코드의 일부로서, 각 객체가 자신의 클래스에 대응하는 vtbl에 대한 포인터로 초기화되어, 이 포인터로 해당 함수를 액세스한다. 명령어 *I3*은 객체의 베이스 주소를 액세스한다. 명령어 *I4*는 객체의 베이스 주소를 사용하여 자신의 vtbl을 액세스한다. 그런 다음 명령어 *I5*는 간접 호출하는데 필요한 함수 주소를 가져온다. 명령어 *I6*은 arr[i]->Valid()를 위한 코드 시퀀스의 첫 번째 명령어로 분기한다. 여기서 명령어 *I3*의 수행 결과로 *R4*의 내용이 결정되면, 해당 명령어 시퀀스 *I3* ~ *I6*의 수행으로 간접 분기의 타겟 주소가 결정된다. 따라서 명령어 *I3*의 *R4*와 명령어 *I6*의 *R3*

의 절대적 상관관계를 미리 알고 있으면, 이와 관련된 명령어 시퀀스의 수행에 앞서 간접 분기의 필요한 타겟 주소를 미리 정확하게 알아낼 수 있게 된다. 이와 마찬가지로 arr[i]->Print()를 위한 코드 시퀀스를 수행하기 위해서는 명령어 체인 *I3, I8, I9, I10*이 수행되어야 하는데, 명령어 *I3*의 수행 결과에 따라 명령어 *I10*의 타겟 주소도 결정되므로, 그 절대적 상관관계로 타겟 주소를 미리 알아낼 수 있다.

IV. 데이터 종속 특성을 이용한 간접 분기 예측

1. 데이터 종속 기반의 간접 분기 예측기

우리가 제안한 간접 분기의 타겟 주소 예측 방식은 간접 분기 명령과 이와 단순 데이터 종속적인 관계를 갖고 있는 이 명령어 보다 훨씬 앞서 수행되는 명령어의 레지스터 내용을 결합하여 간접 분기의 타겟을 예측하는 메카니즘이다. 전절의 그림 5의 예에서와 같이 간접 분기 명령어 *I11*과 이 타겟 주소와 절대적 데이터 상관관계를 갖고 있는 명령어 *I5*의 레지스터 *R2*를 결합시켜 이 정보를 간접 분기 타겟 캐시(IBTC)에 미리 저장해 둔다. 그러면 절대적 데이터 상관관계를 갖는 명시된 특정 레지스터 값에 따른 간접 분기가 수행될 때 자신의 타겟 주소를 결정하기 위한 명령어 시퀀스를 수행하지 않더라도 이 캐시를 참조하여 타겟 주소를 미리 알아낸다.

이를 위해 먼저 수행할 프로그램 코드에서 간접 분기와 절대적 데이터 상관관계를 갖는 명령어의 해당 레지스터가 어떤 것인지를 명확히 명시해 둔다. 이러한 구분을 위해서는 기존의 명령의 레지스터 명시 부분에 한 비트만 추가시켜주면 되므로 부담이 되지 않는다. 본 논문에서는 심플스칼라의 annotation field를 이용하여 이를 표현해 놓았다. 이 명령을 해독하여 수행할 때 명시된 레지스터 즉 최종 절대적 상관관계를 갖는 레지스터 내용을 레지스터 FAR(Final Absolutely-correlated Register)로 불러온다. 그런 다음 그림 7에서와 같이 이 레지스터 FAR과 간접 분기의 주소를 사용하는 해싱 함수에 의해 생성된 인덱스를 주소로 예측 분기 타겟 주소를 저장해 놓은 IBTC를 액세스한다.

2. 간접 분기 타겟 캐시의 해싱 방법과 구조

현재 가장 많이 사용하고 있는 32 비트 프로세서는 메모리 주소와 레지스터가 32 비트로 구성되어 있다. 따라서 IBTC를 액세스하기 위해 사용할 수 있는 정보는 레지스터 FAR과 간접 분기 주소를 합하여 64 비트에 이른다(대부분의 프로세서는 4 바이트 메모리 정렬 방식을 사용하기 때문에 명령어 주소는 30 비트면 충분하다). 하드웨어 제약으로 인해 간접 분기 타겟 캐시를 1K로 구성하는 경우를 생각해 보자. 그러면 이 캐시를 인덱스하기 위해 필요한 비트 수는 10 비트면 충분하다. 따라서 가급적 예측 정확율의 손상이 없도록 하면서 FAR과 분기 주소에 포함된 비트들을 압축하여 좀 더 짧은 비트 패턴을 만들어 사용해야 하고, 이렇게 됨으로써 발생할 수 있는 앨리어싱을 최소화하기 위한 태그 사용 방식도 마련해 주어야 한다. 현재 상용화되어 그 사용이 급격히 증가할 것으로 예상되는 64 비트 프로세서의 경우에는 그 중요성이 더욱 커진다.

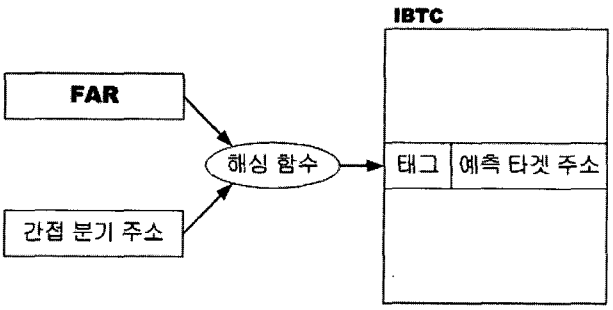
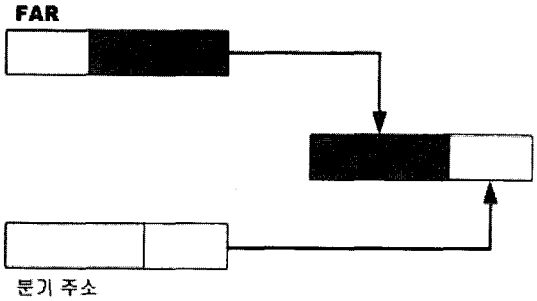
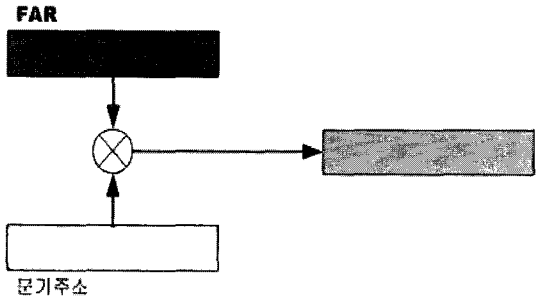


그림 7. 제안한 간접 분기 예측기
Fig. 7. The proposed indirect branch predictor.



(a) 연쇄연결 방식



(b) XOR 방식

그림 8. IBTC의 해싱 방법
Fig. 8. Hashing of IBTC.

IBTC를 해싱하는 방법으로는 크게 두 가지를 선택한다. 하나는 그림 8(a)와 같이 적절한 엔트리를 선택하기 위해 분기 주소와 레지스터 FAR을 연쇄연결(concatenate)하는 것이다. 앨리어싱을 가급적 피하기 위해 인덱스에 분기 주소와 FAR의 비트를 각각 얼마만큼 포함시키는 가에 따라 성능에 다양한 편차를 보이게 될 것이다. 둘째는 그림 8(b)와 같이 가급적 분산시켜서 IBTC에 정보를 저장하기 위해 분기 주소와 FAR를 XOR시키는 것이다.

IBTC를 액세스하기 위해 사용되는 해싱 방법과 더불어 IBTC의 구성도 성능에 영향을 미치게 된다. 제한된 IBTC의 크기에 맞게 인덱스 비트를 사용해야 하기 때문에 분기 주소와 FAR의 정보를 압축할 수밖에 없게 되어 IBTC에서의 간섭이 일어나기가 쉽다. 2-단계 분기 방향 예측기와 달리, IBTC는 BTB처럼 분기 타겟 주소를 저장하므로 간섭이 발생하면 성능에 해만 끼치게 된다. 개개의 간접 분기마다 서로 다른 타겟을 사용하는 것이 거의 대부분이기 때문에 간섭이 일어나게 되면 곧바로 예측 실패로 이어지게 된다. 따라서 이러한 간섭을 피하기 위해 IBTC의 구조는 태그를 저장하도록 만든다. 타겟 주소와 함께 태그 정보를 저장해야 하기 때문에 IBTC의 하드웨어 부담이 커지게 되므로 성능 손실을 최소화하면서 동시에 태크 매칭에 사용되는 정보를 최적화하는 방법이 필요하다. 그 여러 가지 태그 매칭 방식과 이에 대한 성능 평가를 다음 절에서 다룬다. 또 다른 방법으로 태그 저장에 따른 하드웨어 부담을 없애기 위해 태그가 없는 IBTC를 사용하는 것이다. 간섭이 발생되는 경우에 아주 취약하기는 하지만 적절한 해싱 방식을 사용하여 앨리어싱의 발생을 최대한으로 억제시킨다면 성능에 큰 도움이 될 것이다.

3. 태그 구성 방식에 대한 성능평가

가. 실험 환경

본 논문에서 제시한 실험 결과는 명령어 수준의 시뮬레이터인 심플스칼라를 사용한 이벤트 구동 시뮬레이션을 통하여 얻었다.[25] 심플스칼라 시뮬레이터에 데이터 종속 기반의 간접 분기 예측기를 구현하고 다양한 IBTC의 크기에 대해 제안된 두 가지 해싱 방법과 여러 가지 태그 사용 방식을 적용하여 간접 분기 예측율을 측정하였다.

벤치마크 프로그램으로는 표 1과 같은 특성을 갖는 SPEC95int를 사용하였다. 시뮬레이션 시간을 줄이기 위해서 한 프로그램에서 수행되는 명령어 수를 최대 2,500M로 제한하였다. 벤치마크 프로그램에 나타나는 간접 분기의 빈도수는 평균 100 개의 명령어 수행 당 1 개 정도로 나타났으며, 그 빈도수는 프로그램에 따라 많은 편차를 보였다. 간접 분기의 타겟 수는 평균 4.8 개로 나타났다. ijpeg과 compress는 간접 분기의 타겟 수가 1 개인 경우가 각각 86.6%, 55.6%로 아주 많이 존재하는데, 이는 타겟이 단 하나뿐인 라이브러리 함수 호출이 많이 포함되어 있기 때문이다. 그러나 많은 경우의 프로그램들에 포함되어 있는 간접 분기는 타겟이 여러 개로 동적으로 변하는 특성을 보이고 있다.

나. 태그를 사용하는 IBTC

IBTC를 액세스하는 해싱 방법과 더불어 IBTC에 저장되는 태그 정보도 성능에 크게 영향을 미치게 된다. 본 절에서는 이에 따라 영향을 받는 IBTC의 성능을 살펴본다. IBTC의 전체 크기를 1K 엔트리로 고정하고, 이에 대한 여러 가지의 집합 연관에 따른 성능을 측정하였다. 다른 크기의 엔트리에 대해서도 거의 비슷한 결과를 보이기 때문에 지면 관계상 생략하였다.

표 1. 벤치마크 프로그램
Table 1. Benchmark program.

벤치마크	입력 데이터	수행된 명령어수 (백만개)	간접 분기 당 명령어 수	타겟 수(n)						
				n = 1		2 ≤ n ≤5		5 < n		평 균
go	2stone9.in(train)	548	89.18	134	0.1%	110,788	94.2%	6,664	5.7%	5.2
m88ksim	ctl.raw(ref)	245	25.36	964	40.3%	706	29.5%	720	30.1%	4.4
perl	jumble.pl(train)	2,392	39.46	3,689,524	21.6%	531,273	3.1%	12,867,875	75.3%	7.1
li	train.lsp(train)	183	25.27	51	0.0%	1,951,543	94.6%	110,607	5.4%	3.7
compress	80000 e 2231	239	35.36	318	55.4%	256	44.6%	0	0.0%	1.6
vortex	vortex.raw(train)	2,500	46.21	101,436	11.1%	226,496	24.7%	589,590	64.3%	6.1
ijpeg	vigo.ppm(train)	1,465	493.20	369,642	86.6%	57,291	13.4%	0	0.0%	1.2
gcc	cccp.i(ref)	1,263	51.45	32,591	0.5%	3,048,341	40.8%	4,388,156	58.8%	9.2

먼저 연쇄연결 방식의 해싱을 사용하는 경우에, IBTC의 크기를 1K로 했기 때문에 직접사상의 IBTC의 인덱스 $I(n)$에는 10 비트가 사용된다(n = 10). 전절의 그림 8(a)에서처럼 $I(n)$은 $FAR(r)$과 $BA(p)$로 구성되는데, 이전의 실험 결과로서 r과 p를 같은 정도로 사용하는 것이 대략 가장 최적의 결과를 보이므로 p은 $n/2$ = 5로, r는 $(n - p)$로 각각 설정하였다. IBTC의 전체 크기가 고정되어 있으면서 연관도가 커지게 되면, 인덱스의 비트 수는 역으로 줄어들게 된다. 따라서 인덱스에 포함되는 비트수가 5보다 큰 경우에는 p를 5로 고정시켜두고 r을 줄이는 것이 좋은 성능을 보인다. 인덱스 비트수가 5 보다 작은 경우는 r은 사용하지 않고 p만을 인덱스 비트 수만큼 사용한다. IBTC에 저장되는 태그는 일반적으로 인덱스 비트로 사용되는 것을 제외한 그 나머지 비트들이 된다. 즉 그림 8(a)와 그림 9에서 나타낸 바와 같이 태그 $T_{C1}(m)$은 FAR의 r 비트와 분기 주소의 p 비트를 제외한 $FAR(r') + BA(p')$로 구성된다. 여기서 $m = r' + p' = (W - r) + (W - p)$가 된다. W는 프로세서의 주소 용량을 의미한다. 즉, 4 바이트 정렬방식의 메모리 주소를 사용하는 32 비트 프로세서인 경우는 W가 30이 된다(실험에 사용한 심플스칼라 시뮬레이터에서는 명령어가 8 바이트 메모리 정렬이 되기 때문에 W 값이 실제보다 하나가 작다). 그러나 태그로 사용되는 비트가 너무 많아지기 때문에 분기 주소 또는 FAR 중 하나만을 선택하여 그 일부만을 태그로 사용하는 방식인 T_{C2}과 T_{C3}에도 적용하여 성능 분석을 하였고, 그 결과를 그림 10에 나타내었는데, 지면관계상 각 벤치마크에 대한 결과는 생략하고 8 개의 벤치마크의 전체 평균값만을 나타내었다.

그림에서 알 수 있듯이 T_{C1}은 연관도가 1인 직접 사상부터 연관도가 1024인 완전 사상까지 예측 정확도가 거의 차이가 없이 높게 나온다. 반면에 T_{C2}는 집합연관 사상의 연관도가 64 미만일 때는 가장 좋은 정확도를 보이다가 연관도가 이보다 커지면 급격히 그 정확도가 떨어지는데, 이는 FAR에 포함된 데이터 종속 특성을 반영할 수 없기 때문이다. 즉, 연관도가 64 이상이 되면 T_{C2}는 일반적인 BTB와 같은 동작을 하기 때문에 당연히 성능이 나빠지는 것이다. T_{C3}은 연관도가 커질수록 정확도가 나빠지는데, 분기 주소의 정보 없이 절대상관관계 데이터만으로는 서로간의 분별력이 떨어지기 때문에 나타나는 현상이다.

XOR 방식의 해싱의 경우, 분기 주소와 FAR를 XOR 한 후에 하위 x 비트를 인덱스에 사용한다. XOR 해싱 방식에서 사용하는 태그 $T_{X1}(m)$은 FAR의 모든 비트들과 분기 주소의 모든 비트들 즉, $FAR(W) + BA(W)$로 구성된다. 그러므로 이 경우에는 m이 $2W$가 된다. T_{X2}

해싱 방식	인덱스 $I(n)$	태그 $T(m)$
연쇄연결	$FAR(r) + BA(p)$	$T_{C1}(m) = FAR(r') + BA(p')$
		$T_{C2}(m) = BA(p')$
		$T_{C3}(m) = FAR(r')$
XOR	$I(x)$	$T_{X1}(m) = FAR(W) + BA(W)$
		$T_{X2}(m) = BA(W)$
		$T_{X3}(m) = FAR(W)$
		$T_{X4}(m) = I(x')$

그림 9. IBTC의 다양한 태그 방식
Fig. 9. Various tagged schemes of IBTC.

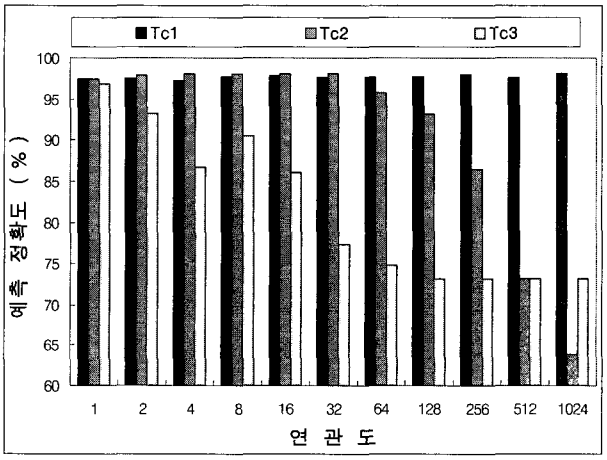


그림 10. 연쇄연결 해싱 방식 사용 시의 다양한 태그 방식에 대한 예측 정확도(%)
Fig. 10. Prediction accuracy for various tagged schemes using concatenation hashing function.

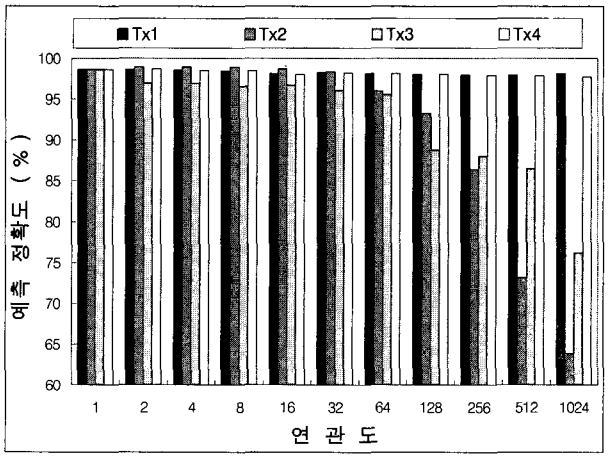


그림 11. XOR 해싱 방식 사용 시의 다양한 태그 방식에 대한 예측 정확도(%)
Fig. 11. Prediction accuracy for various tagged schemes using XOR hashing function.

와 T_{X3}의 태그는 T_{C2}과 T_{C3}와 같이 각각은 분기주소와 FAR 중 하나만으로 구성된다. T_{X4}는 태그로서 분기 주소와 FAR를 XOR한 결과의 하위 x 비트를 제외한 나머지 x' 비트를 사용한다. XOR 해싱방식을 사용하는 경우, 이에 적용한 네 가지 태그 방식에 대한 예측 정확도를 그림 11에서 보여준다.

그림 10과 11에 보인 바와 같이 해싱 방법으로 연쇄연결방식이나 XOR 방식을 사용하는 것과 상관없이 가장 좋은 성능 결과는 여러 가지 태그 구성 방법 중 분기주소로만 태그를 구성하는 경우에 나타난다. 특히 분기주소와 FAR의 내용을 모두 사용해야하는 T_{C1}과 T_{X1}보다 분기주소만을 태그로 사용하는 T_{C2}와 T_{X2}가 IBTC에 저장하는 정보가 훨씬 적을 뿐만 아니라 연관도를 작게 하여 하드웨어 복잡도를 줄여줄 수 있기 때문에 고무적인 결과이다. 그리고 같은 태그 구성 방식을 사용하는 경우, XOR 해싱 방식이 연쇄연결 해싱 방식에 비해 좋은 성능을 보인다. 연쇄연결 방식 중 연관도가 32인 T_{C2}가의 예측 정확도가 98.12%로 가장 좋았으며, XOR 방식 중에는 연관도가 2인 T_{X2}가 98.92%의 예측 정확도를 보였다. 이는 XOR 해싱 방식이 제한된 크기의 IBTC를 좀 더 분산적으로 사용하기 때문에 나타나는 결과이다.

결론적으로, 해싱 방식으로는 XOR를 사용하고 태그 방식으로는 분기주소만으로 구성하는 것이 가장 좋은 성능을 보인다.

나. 태그/무태그 사용 방식의 IBTC 비교

태그를 사용하는 IBTC는 태그 비교를 하기 위해 해당 태그를 저장하는 구조로 되어 있으나, 태그를 사용하지 않는 IBTC는 태그 저장이 필요없게 되므로, 태그를 사용하지 않는 만큼의 IBTC 크기를 줄일 수 있거나 같은 크기의 태그를 사용하는 방식 보다 엔트리 수를 늘릴 수 있다. 그런 반면 IBTC에서의 간섭이 커지게 되어 태그를 사용하는 방식보다 성능이 떨어질 수 있다.

그림 12는 128 엔트리 크기의 태그를 사용하는 방식 중에 가장 성능이 좋은 T_{C2}와 256 엔트리 크기의 태그를 사용하지 않는 연쇄 연결 방식 및 XOR 방식이 갖는 예측 실패율을 보여준다. 무태그 연쇄연결 방식의 예측 실패율은 6.26%이고, 무태그 XOR 방식은 2.14%의 예측 실패율을 보인다. 그리고 태그 사용 방식인 T_{C2}는 연관도 4일 때 가장 낮은 2.24%의 실패율을 보여준다. IBTC의 크기를 작게 하는 경우에는 집합 연관 사상을 사용하는 것보다 엔트리 수를 보다 늘린 직접 사상을 사용하는 것이 더 유리하다.

또한 그림 13은 엔트리 크기를 태그 사용 방식에는 2K로 하고, 태그를 사용하지 않는 방식에는 4K로 했을 때의 성능을 보여준다. 무태그 연쇄 연결 방식의 예측 실패율은 1.54%이고, 무태그 XOR 방식은 1.12%의 예측 실패율을 보인다. 반면에 태그 사용 방식인 T_{C2}는 연관도 8일 때 가장 낮은 1.03%의 실패율을 보여준다. 따라서 충분한 크기의 IBTC를 사용할 수 있는 경우는 적절한 연관도를 가진 집합 연관 사상 방식으로 IBTC를 구성하는 것이 좋다.

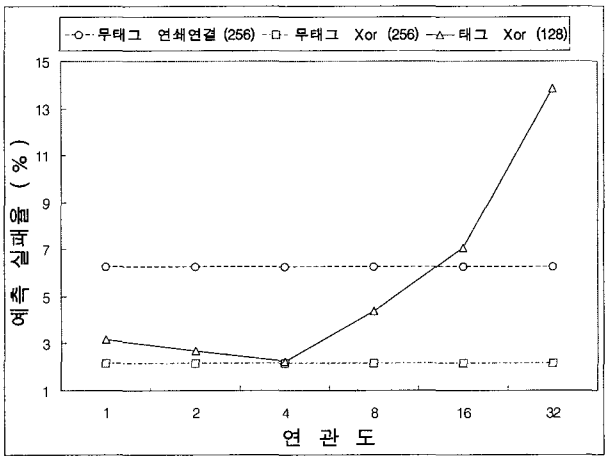


그림 12. 128 엔트리 크기의 태그 사용 방식과 256 엔트리 크기의 태그를 사용하지 않는 방식의 비교

Fig. 12. Comparison of 128 entry tagged and 256 entry tagless IBTC.

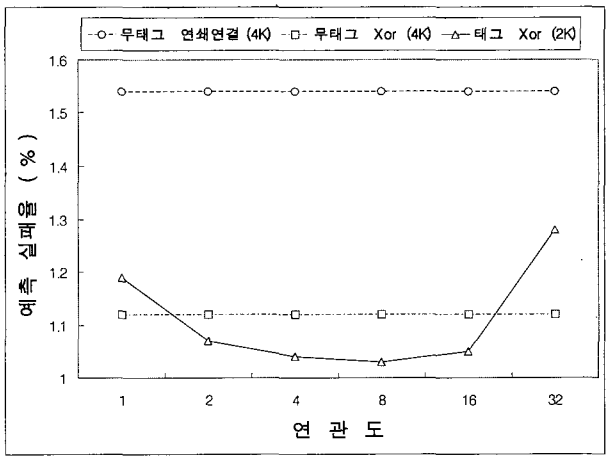


그림 13. 2K 엔트리 크기의 태그 사용 방식과 4K 엔트리 크기의 태그를 사용하지 않는 방식의 비교

Fig. 13. Comparison of 2K tagged and 4K tagless IBTC.

V. IBTC의 최소화

간접 분기 예측기는 기본적으로 분기 타겟 주소를 저장해야 하기 때문에 하드웨어 부담이 크다. 전절에서 알 수 있듯이 어느 정도의 충분한 엔트리로 구성되는 간접 분기 예측기의 IBTC는 집합 연관 사상 방식으로 구성하는 것이 가장 성능이 우수하다. 그러나 집합 연관 방식으로 구성하는 경우에는 분기 타겟 주소뿐만 아니라 해당 태그도 저장해야 하므로 하드웨어 부담이 훨씬 커지게 된다. 따라서 본 절에서는 예측 정확도의 손실 없이도 예측기의 오버헤드를 줄이도록 한다.

1. 저장되는 태그의 최소화

전장에서 살펴보았듯이 데이터 종속성을 이용한 간접분기 예측기는 XOR 방식으로 IBTC를 인덱싱하고, 집합연관 사상으로 구성된 IBTC의 태그로서 분기주소를 사용하는 것이 가장 좋은 성능을 보인다는 것을 알았다. 본 절에서는 상기 구성의 예측기를 하드웨어 오버헤드 측면에서 최소화하기 위해 저장되는 태그 비트 수를 줄일 수 있는 방법을 모색한다.

XOR 방식이 연쇄연결 방식의 인덱싱 보다 성능이 좋게 나오는 것은 IBTC에 저장되는 정보가 보다 잘 분산되기 때문이고, 집합연관사상의 구조에 태그를 사용하는 것도 같은 이유에서이다. 따라서 성능에 손실이 가지 않는다면 이 저장되는 태그 비트 수를 줄여주어도 무방할 것이다. 본 논문에서는 IBTC의 크기를 1K, 4K 및 8K로 했을 경우의 각 벤치마크에 대한 성능을 표 2에 나타내었다. gcc와 ijpeg를 제외한 모든 벤치마크 프로그램들은 분기 주소의 하위 2 ~ 6 비트만을 저장해도 분기 주소 모두를 태그로 사용하는 것과 동일한 성능을 보여주고 있다. 전체 평균적으로 IBTC의 크기와 연관도에 따라 분기 주소의 하위 10 ~ 12 비트만을 태그로 사용해도 성능에 손실이 전혀 없게 된다. 즉, 이것은 성능 손실 없이도 IBTC의 태그 저장에 필요한 하드웨어 오버헤드를 약 60% 정도 줄일 수 있다는 의미이다.

그림 14는 1K 크기의 IBTC에 부분 태그를 사용했을 때의 완전 태그 사용 방식에 대한 커버리지율을 보여준다. 6 비트의 부분 태그만을 사용했을 경우, 512 × 2, 256 × 4 및 128 × 8의 각 커버리지율은 100%, 99.91% 및 99.90%가 된다. 그러므로 0.1% 이하의 성능 손실을 감수한다면 태그 저장에 필요한 하드웨어 오버헤드를 약 80%까지 줄일 수 있게 된다. 4K나 8K에 대해서도

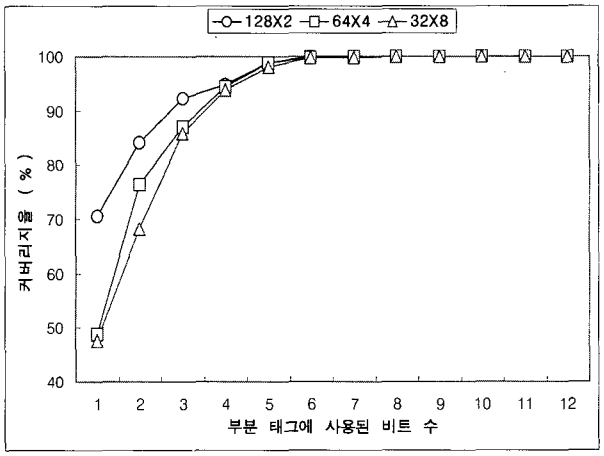


그림 14. 1K 엔트리 크기의 IBTC에서의 부분 태그 사용 방식의 완전 태그 사용 방식에 대한 커버리지율

Fig. 14. Coverage rate of partial tagged scheme for full tagged scheme with 1K entry IBTC.

표 2. 전체 태그 사용 방식의 예상 정확도와 부분 태그 사용 방식의 특성

Table 2. Prediction Accuracy of full tagged schemes and characteristic of partial tagged schemes.

엔트리 크기	집합 연관	벤치마크																	
		전체 태그 사용 방식의 예상 정확도(%) / 부분 태그 사용 방식 적용시의 성능 손실 없는 최소 사용 비트 수																	
		go		m88ksim		gcc		compress		li		ijpeg		perl		vortex		전체 평균	
1K	512×2	99.95	5	95.48	4	99.23	10	96.86	6	99.67	2	99.96	10	99.99	4	99.96	5	98.89	10
	256×4	99.95	5	95.56	4	99.44	9	96.86	6	99.67	2	99.96	10	99.99	4	99.92	4	98.92	10
	128×8	99.95	5	95.61	4	99.46	9	96.86	6	99.46	2	99.46	11	99.99	4	99.91	5	98.90	11
4K	2048×2	99.95	5	95.48	4	99.84	12	96.86	6	99.99	2	99.96	10	99.99	4	99.96	5	99.00	10
	1024×4	99.95	5	95.48	4	99.91	12	96.86	6	99.67	2	99.96	10	99.99	4	99.96	5	98.97	12
	512×8	99.95	5	95.48	4	99.95	12	96.86	6	99.67	2	99.96	10	99.99	4	99.96	4	98.98	12
8K	4096×2	99.95	5	95.48	4	99.93	12	96.86	6	99.99	2	99.96	10	99.99	4	99.96	5	99.01	10
	2048×4	99.95	5	95.48	4	99.95	12	96.86	6	99.99	2	99.96	10	99.99	4	99.96	5	99.02	12
	1024×8	99.95	5	95.48	4	99.95	12	96.86	6	99.67	2	99.96	10	99.99	4	99.96	5	98.98	12

비슷한 결과가 나오지만, 그 결과는 1K 보다도 좋다.

2. 저장되는 타겟 주소의 최소화

궁극적으로 필요한 것은 정확한 분기 타겟 주소를 미리 아는 것이다. 그렇기 때문에 타겟 주소를 정확하게 예측할 수 있는 방식이 필요한 것이고 또한 이를 효과적으로 구현하는 방법이 필요한 것이다. 그러나 요구되는 타겟 주소는 매우 크기 때문에 이를 저장하는 하드웨어 부담이 크다. 따라서 하드웨어 부담을 최소화하면서 효율적으로 타겟 주소를 이용할 수 있는 방법을 제안한다.

분기 주소와 이와 대응하는 타겟 주소의 거리가 그리 멀지 않다면 2^n 경계 상에 있지 않는 한, 이 두 주소의 상위 비트들은 같을 것이다. 그러므로 타겟 주소를 저장해 두어야하는 IBTC에는 필요한 타겟 주소의 하위 일부 비트들만을 저장해 두고 이를 액세스하여 분기 주소의 그 나머지 상위 비트와 연쇄연결을 하면 완전한 타겟 주소를 만들어 낼 수 있게 된다.

표 3은 분기 주소와 타겟 주소의 비트 수가 얼마나 차이가 나는지를 각 벤치마크 별로 나타낸 것이다. 이 표에서 알 수 있듯이 IBTC에 타겟 주소의 하위 19 비트만을 저장해 두고 이를 액세스하여 간접 분기 주소의 그 나머지 상위 비트들과 연쇄연결시키면, 모든 타겟 주소를 저장하는 방식과 성능면에서 전혀 차이가 없다.

그림 15는 타겟 주소를 IBTC에 부분적으로 저장했을 때의 완전 타겟 주소 저장 방식에 대한 커버리지율을 보여준다. IBTC에 타겟 주소의 하위 19 비트 이상

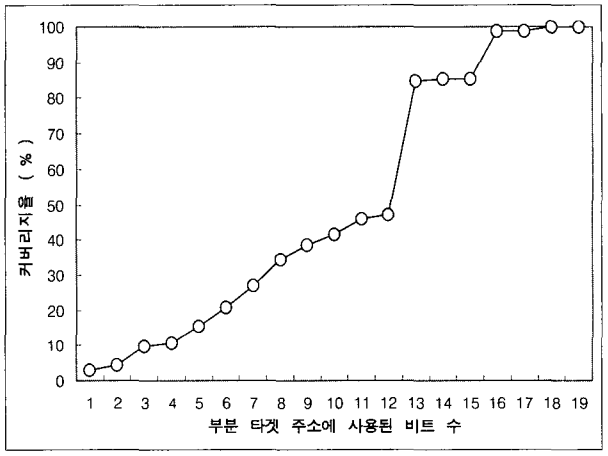


그림 15. IBTC에 부분 타겟 주소를 저장하는 방식의 완전 타겟 주소 저장 방식에 대한 커버리지율

Fig. 15. Coverage rate of partial target scheme for full target scheme.

표 3. 간접 분기 주소와 타겟 주소의 비트 수 차이

Table 3. Difference of bits between indirect branch addresses and its target addresses.

비트수 차이	벤치마크								합계	분포도 (%)
	go	m88ksim	gcc	compress	li	ijpeg	perl	vortex		
1	161	301	675,773	101	631	1,664	57	148,148	826,836	2.94
2	4,462	22	318,940	0	94,560	0	23,169	1,056	442,209	1.57
3	10,920	59	349,641	0	879,126	1,792	178,023	33,190	1,452,751	5.17
4	4,094	103	242,761	0	1	0	14,708	2,561	264,228	0.94
5	14,906	444	262,890	76	973,090	149	1,101	82,234	1,334,890	4.75
6	27,695	118	554,291	26	53	4,481	869,184	66,607	1,522,455	5.42
7	43,667	340	1,085,167	0	0	128	390,519	216,158	1,735,979	6.18
8	5,707	0	1,546,635	1	1	0	525,024	39,453	2,116,821	7.54
9	5,681	162	725,820	0	0	1,537	178,030	224,636	1,135,866	4.04
10	221	108	588,806	0	6	149,529	178,714	90,801	1,008,185	3.59
11	64	42	348,456	56	0	15,299	704,329	5,396	1,073,642	3.82
12	3	108	291,148	310	34,013	384	77	0	326,043	1.16
13	0	27	2,680	0	75,484	207,104	10,339,104	954	10,625,353	37.83
14	5	538	108,845	4	5,222	14,456	3,036	0	132,106	0.47
15	0	18	6,274	0	14	2,752	1	2,619	11,678	0.04
16	0	0	52,075	0	0	27,648	3,685,540	425	3,765,688	13.41
17	0	0	13,728	0	0	0	0	3,284	17,012	0.06
18	0	0	294,635	0	0	0	0	0	294,635	1.05
19	0	0	533	0	0	0	0	0	533	0.00
합계	117,586	2,390	7,469,098	574	2,062,201	426,923	17,090,616	917,522	28,086,910	

을 저장해도 성능에는 변함이 없다. 따라서 하위 19 비트만을 저장하는 경우에는 타겟 주소를 저장하는데 드는 하드웨어 오버헤드를 동일한 성능을 유지하면서도 약 35%를 줄일 수 있게 된다. 만약 1.11%의 성능 손실을 감수한다면, 즉 타겟 주소를 하위 16 비트만을 저장한다면, 하드웨어 오버헤드는 45%까지 절약할 수 있다. 이 그림에서 부분 타겟 주소로 사용된 비트 수를 12 비트에서 13 비트로 올릴 때 커버리지율이 급격히 변화하는 이유는 표 3에서 보는 바와 같이 perl은 이 부분에서 집중적으로 수행되기 때문이다.

VI. 결 론

타겟 주소가 동적으로 다형태로 변하는 간접 분기는 그 타겟 주소 예측이 매우 어렵기 때문에, 고성능의 ILP 기술을 사용하는 대부분의 현대적인 프로세서에서의 성능을 크게 떨어뜨리는 주요 장애요인이 된다. 우리는 예측 정확도가 아주 뛰어난 새로운 개념의 간접 분기 예측 방식 즉, 간접 분기 명령과 이와 데이터 종속 관계를 가진 이 명령어 보다 훨씬 앞서 수행되는 명령어의 레지스터 내용을 결합시켜 간접 분기의 타겟을 예측해내는 방식을 제안하였다. 제안한 간접 분기 예측기를 심플스칼라 시뮬레이터에 구현하고, SPEC 95 벤치마크를 수행하여 얻은 실험 결과로서, 1K의 예측기를 사용하는 경우에 98.92%의 예측 정확도를 보이고, 8K의 크기를 사용하면 거의 완벽한 99.95%의 정확도를 보인다. 그러나 지금까지 제안된 모든 예측기가 그러하듯이 예상 타겟 주소와 함께 앨리어싱 문제를 완화시키기 위한 태그를 저장하기 위한 하드웨어 오버헤드가 크다는 단점을 안고 있다. 그러므로 본 논문에서는 예측 정확도의 손실없이도 예측기의 하드웨어 오버헤드를 최소한으로 줄이는 방법을 제안한다. 실험 결과로써 태그 저장에 따른 하드웨어를 성능 손실 없이 약 60%를 줄일 수 있으며, 0.1%의 손실을 감수하면 약 80%까지 줄일 수 있다. 또한 부분 타겟 저장으로 인한 성능 손실 없이 타겟 주소 저장에 따른 하드웨어를 약 35% 절약할 수 있으며, 1.11%의 손실을 감수하면 약 45%까지 절약할 수 있다.

우리가 제안한 간접 분기 예측기는 기존의 어떤 예측기보다도 예측 정확도가 뛰어날 뿐 아니라, 본 논문에서 제안한 예측기의 하드웨어 오버헤드를 크게 줄일 수 있는 방법을 실제 적용하면, 경제적이면서 실제적인 고성능의 CPU 설계가 가능하게 될 것이다.

참 고 문 헌

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

[2] E. Sprangle and D. Carmean, "Increasing Processor Performance by Implementing Deeper Pipelines", 29th Int'l Symp. on Computer Architecture, pp. 25-34, Anchorage, U.S.A., May 2002.

[3] B. A. Fields, R. Bodik, M. D. Hill and C. J. Newburn, "Using Interaction Costs for Microarchitectural Bottleneck Analysis", 36th Int'l Symp. on Microarchitecture, pp. 228-242, San Diego, U.S.A, Dec. 2003.

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

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

[6] G. H. Loh and D. S. Henry, "Predicting Conditional Branches with Fusion-based hybrid Predictors", 11th Conf. on Parallel Architectures and Compilation Techniques, pp. 165-176, Charlottesville, U.S.A., Sep. 2002.

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

[8] D. A. Jimenes, "Piecewise Linear Branch Prediction", 32nd Int'l Symp. on Computer Architecture, pp. 382-393, Madison, U.S.A., June 2005.

[9] 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.

[10] K. Driesen and U. Holzle, "Limits of Indirect Branch Prediction", Technical Report TRCS97-10, Univ. of California Santa Barbara, June 1997.

[11] O. J. Santana, A. Falcon, E. Fernandez, P. Medina, A. Ramirez and M. Volero, "A Comprehensive Analysis of Indirect Branch Prediction", 4th Int'l Symp. on High Performance Computing, pp. 133-145, Kansay Science City, Japan, May 2002.

[12] B. Calder and D. Grunwald, "Reducing Indirect Function Call Overhead in C++ Programs", 21st Symp. on Principles of Programming Languages, pp. 397-408, Portland, U.S.A., Jan. 1994.
[13] K. Driesen and U. Holzle, "The Direct Cost of Virtual Function Calls in C++", 11th Conf. on Object-Oriented Programming Systems, Languages and Applications, pp. 306-323, San Jose, U.S.A., June 1996.
[14] O. Zendra and K. Driesen, "Stress-testing Control Structures for Dynamic Dispatch in Java", Proc. of the 2nd Java Virtual Machine Research and Technology Symp., pp. 105-118, San Francisco, U.S.A., Aug. 2002.
[15] 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.
[16] K. Driesen and U. Holzle, "Accurate Indirect Branch Prediction", 25th Int'l Symp. on Computer Architecture, pp. 167-178, Barcelona, Spain, July 1998.
[17] 백경호, 김은성, "간접 분기의 다형태 타겟 주소의 정확한 예측", 대한전자공학회논문지, 제41권 CI편 제6호, pp. 511-521, 2004년 11월.
[18] K. Driesen and U. Holzle, "The Cascaded Predictor: Economical and Adaptive Branch Target Prediction", 31th Int'l Symp. on Microarchitecture, pp. 249-258, Dallas, U.S.A., Dec. 1998.
[19] 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.
[20] 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.
[21] K. Driesen and U. Holzle, "Multi-Stage Cascaded Prediction", 5th Int'l Euro-Par Conf. on Parallel Processing, pp. 1312-1321, Toulouse, France, Aug. 1999.
[22] A. Roth, A. Moshovos and G. S. Sohi, "Improving Virtual Function Call Target Prediction via Dependence-Based Pre- Computation", 13th Int'l Conf. on Super computing, pp. 356-364, Rhodes, Greece, June 1999.
[23] M. A. Ertl and D. Gregg, "Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters", Conf. on Programming Language Design and Implementation, pp. 278-288, San Diego, U.S.A., June 2003.
[24] M. A. Ertl and D. Gregg, "The Structure and Performance of Efficient Interpreters", Journal of Instruction-Level Parallelism, Vol. 5, Nov. 2003.
[25] T. Austin, E. Larson and D. Ernst, " SimpleScalar: An Infrastructure for Computer System Modeling", IEEE Computer Society, pp. 59-67, Feb. 2002.

저 자 소 개

백 경 호(학생회원)
1999년 순천향대학교 전자공학과 공학사
2001년 순천향대학교 전자공학과 공학석사
2001년~현재 순천향대학교 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>