값 예측 오류를 위한 순차적이고 선택적인 복구 방식

(Sequential and Selective Recovery Mechanism for Value Misprediction)

이 상 정 † 전 병 찬 ††
(Sang-Jeong Lee) (Byung-Chan Jeon)

요 약 고성능 슈퍼스칼라 프로세서에서 값 예측(value prediction) 방식은 명령의 결과 값을 미리 예측하고, 이 후 데이타 종속 관계가 있는 명령들에게 값을 조기에 공급함으로써 이들 명령들을 모험적으로 실행하여 성능을 향상시키는 방식이다. 값 예측으로 성능을 향상시키기 위해서는 예측 실패 시에 효율적으로 복구하는 과정이 필수적이다.

본 논문에서는 값 예측 실패 시에 잘못 예측된 값을 사용하여 모험적으로 수행된 명령들만을 순차적으로 취소하고 복구한 후에 재이슈하는 값 예측 실패 복구 메커니즘(value misprediction recovery mechanism)을 제안한다. 제안된 복구 방식은 한번에 모든 종속명령들을 검색하지 않음으로써 파이프라인을 정지시키지 않는다. 즉, 파이프라인이 진행되는 순서에 따라 순차적으로 값 예측이 틀린 종속명령만을 선택적으로 취소하고 재이슈하여 불필요한 취소와 재이슈를 줄임으로써 값 예측 실패 시에 손실을 줄인다.

키워드 : 슈퍼스칼라 프로세서, 모험적 실행, 값 예측, 값 예측 실패 복구

Abstract Value prediction is a technique to obtain performance gains by supplying earlier source values of its data dependent instructions using predicted value of a instruction. To fully exploit the potential of value speculation, however, the efficient recovery mechanism is necessary in case of value misprediction.

In this paper, we propose a sequential and selective recovery mechanism for value misprediction. It searches data dependency chain of the mispredicted instruction sequentially without pipeline stalls and adverse impact on clock cycle time. In our scheme, only the dependent instructions on the predicted instruction is selectively squashed and reissued in case of value misprediction.

Key words : superscalar processor, speculative execution, value prediction, value misprediction recovery

1. 서 론

고성능 슈퍼스칼라 프로세서에서 높은 성능에 도달하기 위해서는 명령어 수준 병렬성(Instruction Level Parallelism, ILP)을 이용하여 다수의 명령을 동시에 이슈하고 처리해야 한다. ILP를 이용하는 주요장애는 명령어 간의 종속 관계인 제어종속(control dependences)과 데이타종속(data dependences) 관계이다. 제어종속 관계는 프로그램의 제어흐름이 변경되는 분기 명령에 의해 발생한다. 즉, 분기될 타겟주소와 분기조건 결과 생성의 지연으로 발생하는 문제이다. 제어종속 관계를 극복하는 방식으로는 조건분기의 결과를 미리 예측하여 제어종속관계가 있는 명령들을 미리 페치하여 모험적으로 실행(speculative execution)한다[1,2]. 데이타 종속관계는 현재 명령이 이전 명령의 수행결과를 참조할 때 발생하고, 이전 명령의 결과가 생성될 때까지 현재의 명령은 실행할 수가 없다. 데이타종속 관계는 와이드 이슈 프로세서에서 명령 간에 빈번히 발생하기 때문에 명령어 수준 병렬처리의 주요 장애가 되어 고성능 슈퍼스칼라 프로세서의 성능 저하의 주된 요인이 되고 있다. 따라서, 최근에는 실행되는 명령의 결과 값을 미리 예측하고, 이 후 데이타종속 관계가 있는 명령들에게 값을 조

· 본 연구는 한국과학재단 목적기초연구(R05-2000-000-00283-0) 지원으로 수행되었음
† 정 회 원 : 순천향대학교 정보기술공학부 교수
sjlee@sch.ac.kr
†† 비 회 원 : 청운대학교 컴퓨터과학과 교수
jbc66@cwunet.ac.kr
논문접수 : 2003년 4월 2일
심사완료 : 2003년 9월 22일

기에 공급하고 이들 명령들을 모험적으로 실행하여 성능향상을 꾀하는 값 예측 방식에 관하여 활발히 연구가 진행되고 있다[3-12]. 슈퍼스칼라 값 예측기로는 여러가지 기법들이 제안되었고 대표적인 예측기로 최근값 예측기(last value predictor)[6], 스트라이드 예측기(stride value predictor)[10,11], 2-단계(2-level value predictor) 및 혼합형(hybrid value predictor)[11] 등이 있다. 그러나 값 예측은 분기예측 보다도 예측이 어려워 예측 정확도가 70-80% 수준이어서 예측 실패가 빈번히 발생한다. 따라서 값 예측으로 성능을 향상시키기 위해서는 예측 실패 시에 효율적으로 복구하는 과정이 필수적이다. 본 논문에서는 값 예측 실패 시에 잘못 예측된 값을 사용하여 모험적으로 수행된 명령들만을 순차적으로 취소하고 복구한 후에 재이슈하는 값 예측 실패 복구 메커니즘을 연구 제안한다. 제안된 복구 방식은 값 예측이 틀린 종속명령만을 선택적으로 재이슈하여 불필요한 재이슈를 줄임으로써 값 예측 실패 시에 손실을 줄인다. 또한 잘못 예측된 명령에 종속적인 명령들을 한번에 병렬로 검색하지 않고 명령들의 종속체인을 따라 순차적으로 검색함으로써 프로세서의 클럭 사이클에 영향을 미치지 않으면서 하드웨어의 구현의 복잡성을 줄인다. 제안된 순차적 복구 메커니즘은 심플스칼라(Simple-Scalar) 시뮬레이터[13] 상에 기존의 대표적인 값 예측기와 함께 구현하고, 다양한 SPECint 벤치마크 프로그램에 대해 성능을 측정 분석하여 제안된 방식의 효율성 및 타당성을 입증한다.

2. 관련연구

최근값 예측기는 가장 최근에 일어난 값을 근거로 한 명령의 결과 값을 예측하는 기법으로 고정된 상수형 값을 생성하는 명령의 결과를 예측하는데 적합하다[6]. 스트라이드 값 예측기는 최근값 예측기를 확장하여 명령의 결과값이 상수형 값 뿐 만 아니라 고정된 값(스트라이드)으로 변하는 값을 예측하기 위해 개발되었다[10,11]. 스트라이드는 한 명령에 대해 최근에 두 번 수행된 결과 값들의 차이다. 스트라이드를 구한 후에 바로 이전에 수행된 결과 값과 스트라이드를 더하여 값을 예측한다. 2-단계 값 예측기[11]는 일정한 패턴으로 변하는 값들을 예측하기 명령의 수행결과로 생성된 일련의 결과값들을 테이블에 저장하고 저장된 패턴을 기준으로 다음에 생성될 값을 예측하였다. 이들 예측기들은 특정 패턴의 명령들에 대해서는 잘 동작하지만 다른 패턴에 대해서는 성능이 저하되어 최근에는 이들 예측기들을 통합한 혼합형 값 예측기를 개발하고 있는 추세이다[3,8,10,11].

이와같이 기존의 많은 논문에서 효율적인 값 예측기의 구현 및 성능에 관해서는 많은 연구가 있었지만 효과적인 값 예측 실패 복구 방식과 이의 성능에의 영향에 관한 연구는 상대적으로 적은 편이다. Lipasti는 값 예측 실패 시 분기예측의 복구와 같이 예측이 잘못된 명령 이 후 모든 명령들을 취소하고 다시 페치/이슈하는 명령 재페치/재이슈 방식과 종속관계가 있는 명령만을 재이슈하는 선택적 재이슈(selective instruction reissue) 방식을 언급하였다[7]. 그리고 일반적으로 값 예측기의 정확도는 70-80% 정도로 분기예측기의 정확도보다 떨어지고, 값 예측되는 명령들은 일반 명령들로써 조건분기에만 적용되는 분기예측 보다도 더 빈번히 예측을 시도하여 명령 재페치/재이슈 방식은 오히려 값 예측을 하지 않는 경우보다도 성능이 더 저하될 수도 있음을 지적하였다. 따라서 선택적인 재이슈 방법을 값 예측에 적용하였지만 구체적인 방법에 대해서는 기술하지 않았다. Rychlik 등은 PowerPC 스타일의 슈퍼스칼라 프로세서 상에서 값 예측 실패 시 선택적 재이슈를 위해 선택적 재이슈 코어(selective reissue core)를 설계 제안하였다[8]. 즉, 모험적으로 이슈된 명령들의 모든 소스 오퍼랜드 값을 생성하는 명령들이 실행 완료되어 이 소스 오퍼랜드들의 값이 최종적으로 올바른 비모험적 상태(non-speculative state)에 도달할 때까지 명령들은 완료(commit)되지 않고 명령 윈도우에 남아 있는다. 그리고 이들 소스의 값 예측이 실패로 판명된 경우 다시 재페치하거나 디스패치(dispatch)하지 않고 명령 윈도우 상에서 즉시 재이슈하였다. 그러나 Rychlik의 방식은 설계의 간소화를 위해 예측된 명령의 값과 직접 종속관계(direct dependency)가 있는 명령이 수행된 후 이 결과 값을 참조하는 종속관계가 있는 명령 즉,예측된 명령과 간접 종속관계(indirect dependency)가 있는 명령에 비모험적 상태가 간접적으로 계속 전파되는 것을 허용하지 않았다. 모험적 상태의 수행 결과에 대해서는 더 이상 결과값을 전파하지 않아서 올바른 값 예측인 경우 성능 향상을 제한하였다. Zhou 등 명령의 디코드 후 디스패치 시 명령들 간에 종속관계를 표시하는 종속 매트릭스(dependency matrix)를 구성하고 예측실패 시 이 매트릭스를 참조하여 한번에 병렬로 재이슈될 명령을 탐색하여 복구하였다[12]. 종속 매트릭스의 구성이 재이슈될 명령을 찾는데 도움이 되지만 명령의 디스패치 스테이지에서 명령 간의 직접 종속관계 뿐만 아니라 간접 종속관계를 포함하여 종속 매트릭스를 구성해야 하는 부가 작업이 요구되어 하드웨어 구성이 복잡해진다. 또한 다수의 명령이 이슈되는 와이드 이슈 프로세서에서는 명령들 간의 종속관계가 수십 개 이상 되는 경우도 있어 프로세서의 클럭 사이클이 길어져야 하는 문제점이 있다. 또한 Zhou의 방식은 명령의 수행 후 결과

값이 비모험적 상태로 생성된 경우에는 모험적 수행 후 윈도우 상에서 대기하고 있는 모든 직접 및 간접 종속 명령들에 대해 병렬로 비모험적 상태를 전파해야 한다. Sato는 값 예측 실패로 잘못 수행되어 재이슈될 명령들의 병렬로 한번에 처리하지 하지 않고 순차적으로 재이슈하는 방식을 제안하였다[9]. 즉, 종속 명령들이 예측 값을 사용하여 모험적 이슈된 후에 예측된 명령이 잘못 예측된 경우로 판명된 경우 이의 모든 종속명령을 한 클럭에 병렬로 검색하여 재이슈하지 않고 직접 종속관계만 있는 명령을 재이슈하였다. 그리고 재이슈된 명령이 수행 완료된 후에 다시 이의 종속명령을 재이슈하는 순차적인 재이슈 방식을 제안하였다. Sato의 방식은 종속명령의 순차적인 검색으로 하드웨어 및 프로세서의 클럭 사이클의 부담을 줄였지만 다수의 명령이 이슈되어 실행되는 다중이슈(multiple issue)가 아닌 하나의 명령이 이슈되는 단일 이슈 모델에만 적용되고 다중 이슈인 경우에 대해 명확한 모델을 제시하지 못하였다. 즉, 한번에 여러 개의 명령이 이슈되고 동시 예측되면서 한 명령이 이들 명령에 모두 종속관계가 있어서 발생하는 복합적인 재이슈 상황을 대처하지 못했다.

본 논문에서 제안된 값 예측 실패 복구 방식은 다중 이슈되어 비모험적 상태로 전파된 명령들을 저장하는 상태 큐(non-speculation queue)와 이 큐로부터 명령들을 읽어들이고 종속명령들에게 비모험적 상태를 전파하는 파이프라인 스테이지를 추가함으로써 다중 이슈 모델에 대하여 순차적이고 선택적인 재이슈를 지원한다 (4.2절 참조).

3. 머신 모델 및 값 예측 실패 복구 방식

3.1 머신모델

본 논문에서는 심플스칼라(Simplescalar) 프로세서 모델[13]에 스트라이드와 2-단계 예측기를 통합한 혼합형 예측기[11]에 대해 값 예측 실패 복구 기법을 적용한다. 제안된 복구방식에 특정 머신 모델이나 값 예측기에 크게 종속되지 않기 때문에 일부 수정으로 다른 머신 모델과 예측기에도 쉽게 적용할 수 있다.

심플스칼라 모델은 RUU(register update unit)를 명령 윈도우로 사용한다. RUU는 비순차 이슈를 지원하기 위해 디스패치된 명령을 저장하는 명령 윈도우의 기능과 명령 리카버리를 위해 프로그램의 순서를 유지하는 리오더 버퍼(reorder buffer)의 기능을 통합한 큐 구조의 장치이다. 파이프라인은 5개의 스테이지로 구분되어 동작한다. 먼저 명령페치(Instruction Fetch, IF) 스테이지에서 명령들이 L1 캐시로부터 명령들을 페치하고 분기예측기는 다음 명령의 주소를 예측한다. 명령 디코드 및 디스페치(Instruction Decode & Dispatch, ID) 스테이지에서는 페치된 명령을 디코드하고 소스 오퍼랜드의 가용여부를 체크하고 RUU에 디스페치한다. 그리고 명령 이슈 및 수행(Instruction Issue & Execution, EX) 단계에서는 명령들의 소스 오퍼랜드가 가용한지를 검사하여 모든 소스 오퍼랜드가 가용하면 명령들을 이슈하여 수행한다. 결과저장(Writeback, WB) 단계에서는 수행이 종료된 명령들에 대해 RUU에서 이 결과 값을 기다리고 있는 종속 명령들에게 전파(broadcast)한다. 그리고 분기예측의 결과를 검증하고 틀린 경우 분기예측 실패 복구과정을 시작한다. 마지막으로 완료(commit, CM) 단계에서는 RUU의 선두에 도달하고 수행이 완료된 명령에 대해서 결과값을 해당 레지스터 파일에 저장하고 RUU에서 제거된다.

명령의 값 예측을 이용한 파이프라인 단계는 다음과 같다. IF 스테이지에서 예측기를 참조하여 결과값을 예측하고 ID 스테이지에서는 이 예측값을 사용하는 종속 명령들의 소스 오퍼랜드를 가용한 것으로 표시하고 디스페치한다. EX 단계에서는 예측값을 사용한 종속명령들을 모험적으로 이슈(speculative issue)하고 수행한다. 그리고 결과 값이 예측된 명령이 수행 완료된 경우 WB 스테이지에서 예측의 성공 여부를 검증하여 예측이 성공한 경우에는 계속 진행하고 예측이 실패한 경우에는 잘못된 예측명령을 사용하여 모험적으로 이슈되어 수행된 종속명령들의 결과를 취소하고 다시 이슈하는 복구 과정을 수행한다.

3.2 값 예측 실패 복구방식

IF 스테이지에서 예측된 값은 WB 스테이지에서 예측의 올바름이 검증된다. 이때 검증되는 명령은 비모험적 상태(non-speculative state) 즉, 모든 소스 오퍼랜드가 모험적으로 수행된 값이 아닌 비모험적 최종 값으로 수행된 상태로 수행이 완료된 상태가 되어 있어야 한다. 모험적 상태로 수행이 완료된 경우 명령의 결과 값이 잘못된 값인 경우도 발생하여 검증 자체가 모험적인 판정(speculative resolution)이 되기 때문이다. 수행이 완료된 명령이 모험적 상태인지 비모험적 상태인지를 판별하기 위해서는 각 오퍼랜드 값에 대한 비모험적 상태 여부를 추적하고 한 명령의 수행 결과 값의 상태를 종속명령의 소스 오퍼랜드에 전달하는 비모험적 상태 전파가 이루어져야 한다.

예측된 명령이 비모험적 상태로 최종 완료되고 예측이 실패한 것으로 판명된 경우 이 예측 값을 사용하여 이미 모험적으로 이슈된 명령들의 수행을 취소시키고 다시 올바른 값으로 재이슈하는 값 예측 실패 복구과정을 수행해야 한다. 값 예측 실패를 복구하는 기법으로는 예측 실패 이 후 모든 명령을 재이슈하는 방식과 예측이 실패한 명령에 데이타 종속적인 명령만을 선택적으

로 재이슈하는 방식이 있다[7-9,12]. 모든 명령을 재이슈하는 방식은 잘못 예측된 명령에 데이타 종속되지 않고 올바르게 수행된 명령도 취소하고 다시 수행해야 하기 때문에 복구 시 사이클 손실이 커서 오히려 값 예측을 하지 않는 경우보다도 성능이 더 저하되는 경우가 빈번히 발생한다.

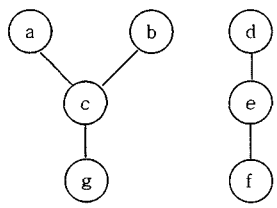
선택적인 재이슈 방식은 예측 실패한 명령 이후의 모든 명령을 취소하지 않고 예측 값에 데이타 종속적인 명령만을 선택하여 재이슈하는 방식이다. 재이슈되는 명령은 명령 윈도우에서 제거되지 않고 다시 이슈함으로써 명령을 다시 페치하지 않기 때문에 명령 재이슈 방식 보다도 복구 손실이 크게 줄어든다. 그러나 복구하기 전에 예측 실패한 명령에 데이타 종속적인 명령을 명령 윈도우 상에서 검색해야 하는 부담이 있다. 데이타 종속적인 명령의 검색에는 병렬 또는 순차적인 검색 방법이 있다. 병렬 검색방식은 한번에 명령 간의 직접 데이타 종속관계 뿐만 아니라 간접 종속관계를 포함하여 검색하는 방식이다. 이 방식은 한번에 잘못 수행된 명령을 취소하여 이들 명령의 파급을 끊을 수 있다는 장점은 있으나 한 번에 많은 명령을 검색해야 하므로 복잡한 하드웨어가 요구되고 종속명령이 많은 경우 검색하는데 많은 사이클이 소비된다. 값 예측 복구 시에는 파이프라인이 중지되므로 다른 명령들의 수행도 중지되어 프로세서 성능에도 영향을 미친다. 순차적인 방식은 예측 실패 시 한 번에 모든 종속명령을 검색하지 않고 예측 실패 명령에 직접 데이타 종속관계만 있는 명령만을 재이슈하고 간접 데이타 종속명령에 대해서는 파이프라인이 진행되는 과정에서 순차적으로 취소하고 재이슈하는 방식이다. 즉, 잘못 수행된 명령을 한번에 취소되지 않음으로써 수행의 파급을 조기에 끊을 수 없다는 단점이 있지만 대신 파이프라인이 중지되지 않고 계속 수행이 되며 부가되는 하드웨어도 적다는 이점이 있다.

그림 1의 (a)는 심플스칼라 PISA 명령어 세트로 표시된 임의의 코드를, (b)는 데이타 종속관계를 나타낸다. (c)는 4개의 명령을 각각 페치,이슈 및 수행할 수 있는 머신에서 열의 명령에 대해 행의 클럭 사이클에서 각 파이프라인 단계의 진행 상황을 보여주는 그림이다. IF*에서는 페치 스테이지에서 값 예측을 , WB*에서는 예측 결과의 검증을 한다. EX$, WB$는 모험적으로 수행된 상태를 표시한다.

그림 1에서 명령 a,b,d가 레지스터 r3,r5,r7의 결과 값을 예측하고 이 중 명령 a만 예측이 틀리고 나머지 b,d는 올바른 예측을 한 경우를 가정하자. 또한 메모리 로드명령(lw)은 2 클럭이 소요된다고 가정한다. 클럭 3에서 명령 c는 명령 a,b의 예측 값을 이용하여 모험적으로 수행되고 명령 e도 d의 예측 값을 이용하여 모험적

```
a.  lw    r3,4(r2)          ; r3 = M[r2+4]
b.  sll   r5,r4,1           ; r5 = r4 << 1
c.  and   r6,r3,r5          ; r6 = r3 & r5
d.  lw    r7,8(r2)          ; r7 = M[r2+8]
e.  andi  r8,r7,8           ; r8 = r7 & 8
f.  addiu r10,r8,16         ; r10 = r8 + 16
g.  beq   r6,r0,next        ; if (r6 == 0) goto next
```

(a) 프로그램 코드



(b) 데이타 종속 관계 그래프

명 령

사이클	a	b	c	d	e	f	g
1	IF*	IF*	IF	IF*			
2	ID	ID	ID	ID	IF	IF	IF
3	EX	EX	EX$	EX	ID	ID	ID
4	EX	WB*	WB$	EX	EX$		
5	WB*			WB*	WB$		EX$
6	CM	CM				EX$	WB$

(c) 명령 a는 예측 실패하고 명령 b d는 예측 성공 시 파이프라인 수행

그림 1 값 예측 코드 예

으로 이슈된다. 클럭 5에서 명령 a가 수행이 완료되어 예측이 틀린 것으로 검증된다. 이때 명령 재이슈 방식은 a의 이후 모든 명령 b,c,d,e,f,g에 대해 수행을 취소하고 다시 이슈한다. 선택적 명령 재이슈 방식은 명령 a에 종속적인 c와 g 명령 만을 취소하고 재이슈하며 나머지 올바르게 명령들은 취소하지 않고 계속 진행한다.

4. 순차적이고 선택적인 값 예측 실패 복구 기법

4.1 RUU 엔트리 확장

값 예측과 선택적인 재이슈 지원을 위해 명령 윈도우인 RUU의 엔트리에 부가적인 필드가 추가된다. 그림 2는 값 예측과 순차적이고 선택적인 재이슈를 지원하는 RUU 엔트리 구조를 보여주는 그림이고 밑줄 친 부분인 spec, reissued, squashed 필드가 값 예측과 선택적 복구를 위해 추가된 필드이다(숫자는 비트 수를 표시하며 tag 필드는 RUU 크기, data 필드는 지원되는 오퍼랜드 값의 크기에 따라 크기가 정해진다).

그림 2에서 각 소스 오퍼랜드에서 ready 비트는 소스 오퍼랜드의 가용 여부를 표시하며 이전 명령의 수행으로 소스 오퍼랜드가 이미 수행되었거나 또는 값 예측된 경우 TRUE로 세트된다. spec(speculation) 비트는 소

스 오퍼랜드가 예측된 값을 참조한 경우 TRUE로 세트되고, 데스티네이션 오퍼랜드인 경우에는 예측된 값이거나 소스 오퍼랜드 중의 하나의 spec 비트가 TRUE이면 TRUE로 세트되어 명령의 생성된 결과가 예측 값을 이용해 모험적 실행으로 생성된 결과임을 표시한다. data 필드에는 각 오퍼랜드의 값으로 최종 값이거나 예측 값 또는 모험적 실행으로 생성된 결과 값이 저장된다. issued 비트는 명령의 이슈여부를 표시하고 completed 비트는 명령의 수행이 완료되었음을 표시하는 비트이다. reissued 비트는 잘못된 예측 값을 사용하여 명령이 다시 이슈되어야 함을 나타내는 비트이다. squashed 비트는 잘못된 예측 값을 가지고 현재 수행 중인 명령을 취소하기 위해 사용되는 비트이다.

4.2 값 예측 실패 복구 기법

그림 3-5는 확장된 RUU 엔트리와 예측된 값을 이용한 모험적 실행 과정을 C 스타일의 코드로 보여주는 그림이다.

먼저 IF 스테이지에서 값 예측기를 참조하여 명령의 예측 값을 구한 후 그림 3과 같이 디스패치 및 이슈 과정을 수행한다. 만약 명령 i의 데스티네이션 값이 예측되었으면 RUU의 데스티네이션 필드의 data 필드에 예측 값을 저장하고 spec 비트를 TRUE로 세트한다(10-13행). 그리고 명령 i의 소스 오퍼랜드가 예측된 명령에 데이타 종속되어 이 값을 사용하는 경우 해당 소스 오퍼랜드 필드의 data 필드에 예측 값을 저장하고 ready, spec 비트를 TRUE로 하여 오퍼랜드가 가용하지만 모험적으로 생성된 값임을 표시한다(14-18행). 그리고 i의 모든 오퍼랜드들이 가용한 명령이거나 예측이 실패된 값을 참조하여 재이슈되어야 할 명령이면 이슈한다(20-26행).

명령이 이슈되고 수행 후에는 이 결과 값을 WB 스테이지에서 윈도우 상에 기다리고 있는 명령들에게 전파해야 한다. 예측 값이 틀린 경우로 판명된 경우 이 후 잘못 수행된 명령을 재이슈한다. 또한 수행이 완료된 명령이 모험적 상태인지 비모험적 상태인지를 판별하기 위해서는 각 오퍼랜드 값에 대한 비모험적 상태 여부를 추적하고 한 명령의 수행 결과 값의 상태를 종속명령의 소스 오퍼랜드에 전달하는 비모험적 상태 전파(non-speculative state propagation)가 이루어져야 한다. 본 연구에서는 최종 값으로 판명된 명령에 종속적인 명령

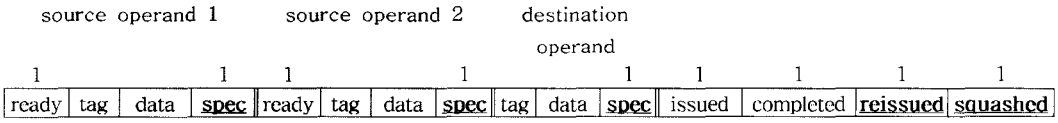


그림 2 확장된 RUU 엔트리

```
1   // instruction i is fetched from the instruction cache in the fetch stage.
2   // dispatch instruction i into ruu_i entry;
3   // ruu entry is initialized when dispatched
4   ruu_i.dest.ready = FALSE; ruu_i.dest.spec = FALSE;
5   ruu_i.src_1.ready = FALSE; ruu_i.src_1.spec = FALSE;
6   ruu_i.src_2.ready = FALSE; ruu_i.src_2.spec = FALSE;
7   ruu_i.issued = FALSE; ruu_i.completed = FALSE;
8   ruu_i.squashed = FALSE;
9
10  if (instruction i predicts a value) {
11        ruu_i.dest.data = predicted value;
12        ruu_i.dest.spec = TRUE;
13  }
14  if (instuction i uses the source operands which are predicted) {
15        ruu_i.src_n.ready = TRUE;    // ,where n=1 or 2
16        ruu_i.src_n.data = predicted value;
17        ruu_i.src_n.spec = TRUE;
18  }
19
20  // all the operands of instruction i are ready
21  // or it is a candidate for reissue
22  if ((ruu_i.src_1.ready && ruu_i.src_2.ready) || ruu_i.reissued) {
23        ruu_i.issued = TRUE;
24        ruu_i.reissued = FALSE;
25        issue the instruction i to the functional unit;
26  }
```

그림 3 명령 디스패치 및 이슈 스테이지에서의 값 예측 동작

```
1   // assume the execution of instruction i is completed
2   // and its execution result is result_i
3
4   // squash the incorrect issued instruction
5   if (ruu_i.squashed) {
6       squash the execution of instruction i.
7       ruu_i.squashed = FALSE;
8       return;
9   }
10  ruu_i.dest.data = result_i;
11  ruu_i.completed = TRUE;
12  if (ruu_i.src_1.spec || ruu_i.src_2.spec)
13      ruu_i.dest.spec = TRUE;
14  if (instruction i is the mispredicted branch instruction && !ruu_i.dest.spec){
15      recover mispredicted branch prediction;
16      return;
17  }
18  // assume instuction j in ruu is data dependent on the instruction i
19  // and its source n(n=1,2) uses the result value of i
20  // broadcast the result to the waiting instruction j
21  if (!ruu_j.issued && !ruu_j.reissued) {
22      ruu_j.src_n.ready = TRUE;
23      ruu_j.src_n.data = result_i;
24      if (ruu_i.dest.spec)          // if i was executed speculatively
25         ruu_j.src_n.spec = TRUE;   // propagate the speculative flag of i into j
26  }
27  // if j is already issued and i was executed non-speculatively
28  else if (!ruu_i.dest.spec) {
29      // propagate the non-speculative flag of i into j
30      ruu_j.src_n.spec = FALSE;
31      // if j referenced the mispredicted or incorrect value of i
32      if (ruu_j.src_n.data != result_i) {
33          ruu_j.issued = FALSE;
34          ruu_j.reissued = TRUE;         // for reissue
35          ruu_j.src_n.data = result_i;   // result broadcast
36          if (!ruu_j.completed)          // if j is is executing and not comleted
37             ruu_j.squashed = TRUE;     // increment squash counter for squashing
38      }
39      // if j referenced correct value of i
40      else {
41          if (!ruu_j.src_1.spec && !ruu_j.src_2.spec) {
42             ruu_j.dest.spec = FALSE;
43             // non-speculation flag will propagate through data dependeny
44             // chain in non-speculation queue
45             insert instruction j into NS queue;
46             if (instruction j is a mispredicted branch)
47                  recover mispredicted branch prediction;
48          }
49      }
50  }
```

그림 4 WB 스테이지에서의 값 예측 복구 동작

들에게 비모험적 상태 전파를 위해 한 번에 병렬로 모든 명령들을 검색하지 않고 이들 명령에 직접 종속관계에 있는 명령들만을 비모험적 상태 큐(non-speculation queue, NS 큐)에 저장한다. 그리고 다음 클럭에서 이들 큐에 저장된 명령들에 대해 비모험적 상태를 전파하고 다시 이들 명령에 직접적 종속적인 명령들만을 NS 큐에 저장한다. 이와 같은 과정을 순차적으로 반복하여 데이타 종속 체인 끝까지 비모험적 상태를 순차적으로 전파한다. 본 논문에서 제안한 방식은 한번에 많은 명령을 병렬로 검색하고 전파를 위해 파이프라인을 중지할 필요가 없다.

그림 4는 WB 스테이지에서의 복구 동작을 나타낸 그림이다. 먼저 수행이 완료된 명령 i의 squashed 비트를 조사하여 TRUE이면 잘못된 값으로 수행된 명령이므로 수행 결과 값을 취소하고 squashed 비트를 FALSE로 리셋한다(4-9행). 그리고 RUU에 결과값과 completed 비트 세트한다(10-11행). 수행 완료된 명령 i의 소스 오퍼랜드 중 하나가 모험적으로 수행된 값이면 명령 i의 데스티네이션의 spec 비트를 TRUE로 세트하여 모험적으로 실행된 결과임을 표시한다(12-13행). 그리고 명령 i가 잘못 예측된 분기 명령이고 비모험적으로 수행되어 완료된 명령이면 잘못된 경로의 수행된 명령

을 취소하고 올바른 경로의 명령들을 패치하여 실행하는 분기실패 복구 과정을 수행한다(14-17행). 명령 i가 이미 이슈되었거나 재이슈된 명령이 아니면 이의 결과를 사용하는 종속 명령 j의 소스 오퍼랜드에 결과 값을 전파하고 명령 i의 결과 값이 모험적 상태이면 j의 해당 소스에도 모험적 상태로 표시한다(21-26행).

만약 명령 j가 이미 예측 값을 사용하여 모험적으로 이슈되었고 명령 i의 결과 값이 비모험적 상태로 완료되었으면 예측 값의 올바름 여부를 검증한다. 예측이 틀린 경우 이미 이슈된 명령 j를 취소하고 재이슈한다(31-38행). 예측이 올바르고 j의 모든 소스가 비모험적인 상태이면 다음 클럭의 비모험적 상태 전파 스테이지(non-speculative state propagation stage, NS 스테이지)에서 다시 j에 종속된 명령들에게 비모험적 상태를 전파하기 위해 명령 j를 NS 큐에 삽입한다. 그리고 j가 잘못 예측된 분기명령이면 분기예측 실패 복구과정을 수행한다(39-49행).

그림 5는 NS 스테이지에서의 값 예측 복구 동작을 보여주는 그림으로 기본적으로는 그림 4의 WB 스테이지에서의 복구 동작과 비슷하다. 먼저 큐로부터 구해진 비모험적 상태로 완료된 명령 j와 직접 종속관계가 있는 명령 k에 대해 예측이 틀린 경우 결과 값을 전파하고 명령을 재이슈한다(6-18행). 예측이 올바른 경우 다시 k에 종속된 명령들에게 비모험적 상태를 전파하기 위해 명령 k를 NS 큐에 삽입한다. 그리고 k가 잘못 예측된 분기명령이면 분기예측 실패 복구과정을 수행한다(19-31행).

그림 6은 앞의 그림 1(a)의 코드 예에서 명령 a,b,d가 값을 예측하고 명령 a,b는 예측이 실패하고 명령 d만 값 예측이 성공한 경우를 가정하여 제안된 값 예측 실패 복구 기법을 적용한 예이다. 그림에서 EX!는 재이슈되어 수행됨을 표시한다. 먼저 (1,a) (1,b) (1,d) - (i,j)는 클럭 사이클 i에서의 명령 j의 파이프라인 상태를 표시 - 에서 값을 예측한다. (3,c)에서 a, b의 예측값을 사용하여 명령 c가 모험적 이슈 수행되고, (4,b)에서 b의

명 령

사이클	a	b	c	d	e	f	g
1	IF*	IF*	IF	IF*			
2	ID	ID	ID	ID	IF	IF	IF
3	EX	EX	EX$	EX	ID	ID	ID
4	EX	WB*	WB$	EX	EX$		
5	WB*		EX$!	WB*	WB$		EX$
6	CM	CM	EX!		NS	EX$	WB$
7			WB			WB$	
8			CM	CM	CM	CM	EX!
9							WB
10							CM

그림 6 그림 1의 (a)의 코드에 대한 순차적이고 선택적인 값 예측 복구 예(명령 a b는 예측 실패하고 명령 d는 예측 성공 시 파이프라인 수행)

```
1   get instruction j from the NS queue;
2
3   // assume instuction K in ruu is data dependent on the instruction j
4   // and its source n(n=1 or 2) uses the result value of j
5
6   // if K referenced the mispredicted or incorrect value of j
7   if (ruu_k.src_n.data != ruu_j.dest.data) {
8      if(!ruu_k.issued)
9           ruu_k.src_n.ready = TRUE;
10     else {
11          ruu_k.issued = FALSE;
12          ruu_k.reissued = TRUE;
13     }
14     // result broadcast
15     ruu_k.src_n.data = ruu_j.dest.data
16     // propagate the non-speculative flag of i into j
17     ruu_k.src_n.spec = FALSE;
18  }
19  // if k referenced correct value of j
20  else {
21      // propagate the non-speculative flag of i into j
22      ruu_k.src_n.spec = FALSE;
23      if (!ruu_k.src_1.spec && !ruu_k.src_2.spec) {
24          ruu_k.dest.spec = FALSE;
25          // non-spec. flag will propagate through data dependency chain in
26          // insert instruction K into non-speculation queue;
27          insert instruction K into NS queue;
28          if (instruction K is a mispredicted branch)
29                recover mispredicted branch prediction;
30      }
31  }
```

그림 5 NS 스테이지에서의 값 예측 복구 동작

예측 실패가 검증 되어 c의 재이슈 비트 세트하고 (5,c)에서 c는 재이슈된다. (4,e)에서 d의 예측값을 사용하여 명령 e가 모험적으로 이슈 수행된다. (5,a)에서 a의 최종 값이 예측 실패로 검증되어서 실행 중인 c를 취소하고 명령 c의 재이슈를 위해 reissued 비트를 TRUE로 세트하고 (6,c)에서 재이슈된다. (5,g)에서 명령 g는 (4,c)에서 모험적으로 수행된 명령 c의 결과가 전파되어 모험적으로 수행된다. (6,e)에서는 (5,d)에서 NS 큐에 삽입된 e에 대해 비모험적 상태 플래그를 f에 전파한다. (6,f)에서 명령 f는 (5,e)에서 전파된 모험적 수행 값을 가지고 모험적으로 이슈를 시작하고 (7,f)에서 수행을 완료한다. 그리고 잘못 수행된 종속명령 g를 다음 사이클에 재이슈하도록 reissued 비트를 세트한다. 명령 g는 (9,g)에서 재이슈되어 수행된다. 만약 분기명령 g가 잘못 분기예측된 명령이라면 (9,g)에서 분기예측 실패 복구 과정을 시작한다. 즉, 모험적 수행 상태인 (6,g)에서는 분기예측 실패 복구를 하지 않고 비모험적 상태를 판명될 때 복구를 한다.

이상과 같이 제안된 값 예측 복구 방식은 한번에 병렬로 예측 실패한 명령을 검색하지 않고 파이프라인 진행에 따라 순차적으로 복구함으로써 파이프라인을 중지하지 않으면서 사이클 타임의 손실을 가져오지 않는다. 또한 다중 이슈된 명령들에 대해서도 효과적으로 적용할 수 있고, 예측 실패 명령 이후 모든 명령을 다시 페치하여 이슈하지 않고 종속된 명령 만을 선택적으로 이슈함으로써 값 예측 실패로 인한 성능저하를 최소화하였다.

5. 성능측정 및 분석

5.1 실험방법

성능측정을 위해 기존의 대표적인 슈퍼스칼라 프로세서의 사이클 수준 시뮬레이터인 SimpleScalar 3.0 tool set에 혼합형 값 예측기와 제안된 값 예측 실패 복구방식을 추가하였다. 표1은 시뮬레이션된 8,16 이슈 프로세서 구성에 대한 각 머신 파라미터를 보여준다. 각각 64KB의 크기를 갖는 명령 및 데이타 캐시와 분기예측기로는 8K 엔트리를 갖는 혼합형 예측기를 사용하였다(gshare와 bimodal의 혼합형 예측기)[1,2]. 값 예측기로는 가장 보편적인 스트라이드와 2-단계 예측기를 통합한 혼합형 예측기를 사용하였다[11].

표 2는 적용된 벤치마크 프로그램의 특성을 보여주는 그림이다. SPEC2000의 경우 시뮬레이션 시간을 줄이기 위해 기준 입력 세트를 수정하여 시뮬레이션 하였다[14]. 각 벤치마크 프로그램에 대해 입력값, 실행된 명령수, IPC(Instrucrions Per Clock Cycle), 분기예측 정확도 및 값 예측 정확도를 나타낸 표로써 8-이슈인 경우를 기준으로 측정된 값이다. 표 2에 나타난 바와 같이 분기 예측 정확도는 평균 93%인데 반하여 값 예측의 정확도는 79%임을 알 수 있다.

5.2 성능분석

그림 7은 8, 16 이슈 머신에 대해 각 방식의 복구방식을 적용할 때 값 예측 시의 성능향상을 보여주는 그림이다(Mean은 기하평균(geometric mean)을 의미한다). 성능향상은 값 예측을 적용하지 않았을 경우의 IPC와 비교해서 각 복구방식의 값 예측을 적용한 경우 IPC가 향상된 비율이다. 그림에서 reissue 방식은 값 예측의 실패가 판명되었을 때 예측 실패 명령 이후의 모든 명령들을 취소하고 재이슈하는 방식이다. serial은 본 논문에서 제안한 순차적이고 선택적인 값 예측 실패 복구 방식이다. parallel은 값 예측 실패 판명 시 이후 종속적

표 1 머신 파라미터 구성

	parameter	value
Processor Core	instruction window (RUU) load-store queue (LSQ) fetch/issue/commit width	256 entries 64 entries 8-16 instructions/cycle 8-16 integer units, 1-cycle latency 4-8 floating-point units, 2-cycle latency 4-8 integer multiply/divide units, 3/12-cycle latency 2-4 floating-point multiply/divide units, 4/12-cycle latency
Memory	120-cycle latency for the first 8 bytes, 2 cycle-latency for each 8 bytes thereafter	
Branch Predictor	branch target buffer (BTB) hybrid branch predictor return address stack (RAS)	2K entries, 2-way gshare with 8K entries,14-bit history, bimodal with 8K entries 32 entries
Caches	level-one data cache level-one instruction cache level-two unified cache	64KB, 4-way, 32 bytes/block, 2-cycle latency 64KB, 2-way, 64 bytes/bock, 2 basic blocks/access 1MB, 128 bytes/block, 12-cycle latency
Value Predictor	hybrid value predictor	8K entries two-level + stride table, 4 history data values

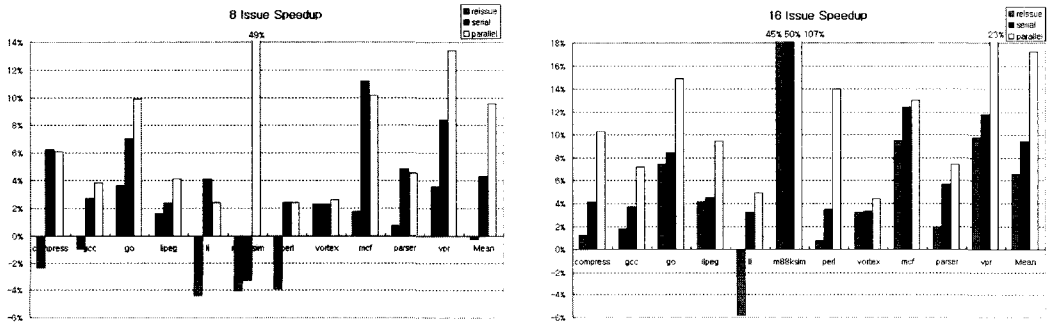
표 2 벤치마크프로그램

Program	Input	Exec. Instr. (million)	IPC	Branch Accuracy	Value Accuracy	Benchmark Set
compress	10000 e 2231	67	2.72	0.92	0.76	
gcc	jump.i	74	1.67	0.92	0.77	
go	5 9	214	1.48	0.84	0.74	
ijpeg	tinyrose.ppm	101	3.39	0.95	0.86	SPEC95
li	queen6.lsp	92	2.20	0.95	0.70	integer
m88ksim	dcrand.little	491	2.68	0.97	0.95	
perl	scrabbl.pl	75	2.45	0.96	0.73	
vortex	persons.250	73	2.45	0.98	0.87	
mcf	smred.in, place	230	2.74	0.96	0.85	SPEC2000
parser	smred.in	484	2.20	0.95	0.77	integer
vpr	smred.in	46	1.98	0.86	0.69	
Average		162	2.36	0.93	0.79	

인 모든 명령들을 병렬로 검색하여 선택적으로 취소하고 복구하는 방식이다(한 클럭에 모든 종속명령들을 검색, 취소하는 것으로 가정하였다. 이는 비현실적이지만 값 예측 실패 복구방식이 도달할 수 있는 성능의 상한으로 볼 수 있다). 그림 7에 나타난 바와와 같이 reissue, serial, parallel 방식에 대해 값 예측 성능이 8 이슈인 경우 각각 평균 -0.2%, 4.3%, 9.5%로 향상되었고 16 이슈인 경우 각각 평균 6.5%, 9.4%, 17.2%로 향상되었다. 즉, 제안된 serial 방식은 reissue 방식 보다 8 이슈인 경우 약 4%, 16 이슈인 경우는 약 3% 정도 성능이 더 되었음을 알 수 있다. 8 이슈인 경우 reissue 방식은 값 예측으로 인해 성능 향상 보다 재이슈되는 명령들로 인한 손실이 더 커서 값 예측으로 오히려 성능이 더 저하됨을 알 수 있다(16 이슈인 경우 명령의 이슈율이 커지면서 값 예측의 성능이 효과가 있음을 알 수 있다). parallel과 비교하여 제안된 serial 방식이 8 이슈인 경우 약 5%, 16 이슈인 경우 약 7% 정도 성능이 저하되었다(그러나 m88ksim을 제외하면 8 이슈인 경우 평균 1%, 16 이슈인 경우 평균 4% 정도 성능이 저하되었다). 즉 m88ksim을 제외하면 제안된 순차적이고 선택적인 값 예측 실패 복구 방식의 성능이 parallel에 근접함을 알 수 있다. m88ksim 벤치마크는 프로세서 시뮬레이터 프로그램으로 값 예측이 효과적으로 적용되어 예측률이 95%나 되어 parallel인 경우와 16 이슈인 경우 인상적인 성능향상을 보였다. 8 이슈 parallel 방식에서 compress, li, parser 등의 성능이 serial 보다 더 저하되었다. 이는 값 예측 이외에 분기예측 실패, 캐시 미스 등의 복합적인 요인이 영향을 미쳐서 발생된 현상이다(예를들어 분기예측 실패로 인한 잘못 수행되는 경로 상에서는 parallel 방식과 같은 신속한 복구는 오히려 성능에 악영향을 미친다).

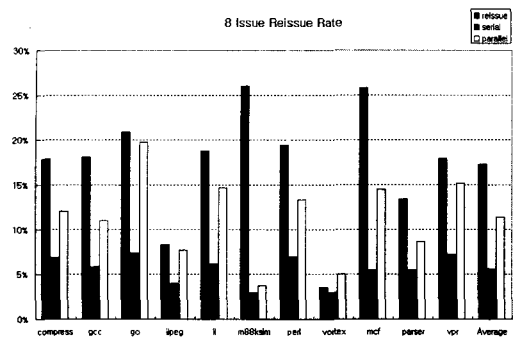
5.3 명령 재이슈율

그림 8은 각 복구 방식을 적용한 경우에 전체 수행된 명령을 기준으로 재이슈되는 명령의 비율을 측정한 그

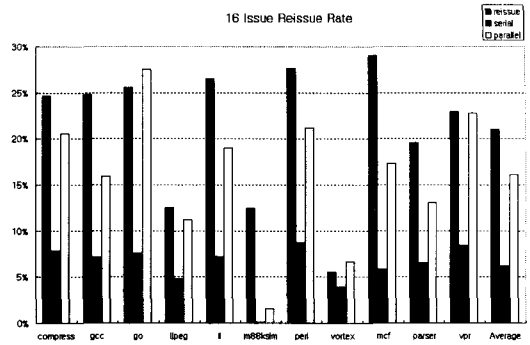


(a) 8 이슈 머신의 성능 향상 (b) 16 이슈 머신의 성능 향상

그림 7 값 예측 실패 복구 방식에 대한



(a) 8 이슈 머신의 명령 재이슈율



(b) 16 이슈 머신의 명령 재이슈율

그림 8 값 예측 실패 복구 방식에 대한

림이다. 명령 재이슈율이 reissue, serial, parallel 방식에 대해 8 이슈인 경우 17%, 5%, 11% 이고, 16 이슈인 경우 21%, 6%, 16% 임을 알 수 있다. 제안된 serial 방식이 재이슈되는 명령의 수가 가장 적음을 알 수 있다. reissue 방식은 예측 실패 이후 모든 명령을 재이슈하여 가장 높은 재이슈율을 갖는다. parallel인 경우에는 한 클럭에 병렬로 모든 재이슈될 명령들을 조기에 검색하는데 이는 분기예측 실패로 인한 잘못 수행된 경로인 경우에 상황을 악화시킨다. 즉, 분기예측 실패로 판명되기 전에 먼저 명령들을 재이슈함으로써 제안된 순차적인 방식보다도 높은 재이슈율을 보였다.

5.4 병렬검색 명령

그림 9는 값 예측 복구 시 병렬검색(parallel) 적용 시에 한 사이클 당 평균 검색되는 데이타 종속명령의 수를 보여주는 그림으로 8 이슈인 경우 평균 2.20개이고, 16 이슈인 경우 2.68개이다. 즉 매 사이클 당 2개 이상의 종속 명령이 매번 검색됨을 알 수 있다. 그림 10은

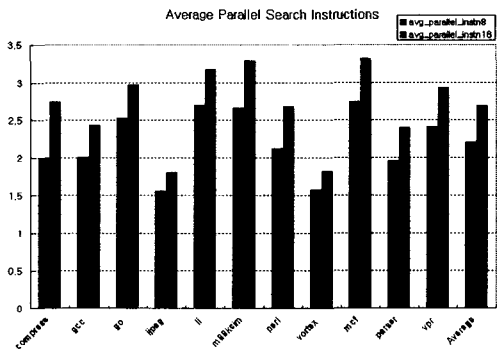


그림 9 병렬검색 적용 시 사이클 당 평균 데이타종속 명령의 수

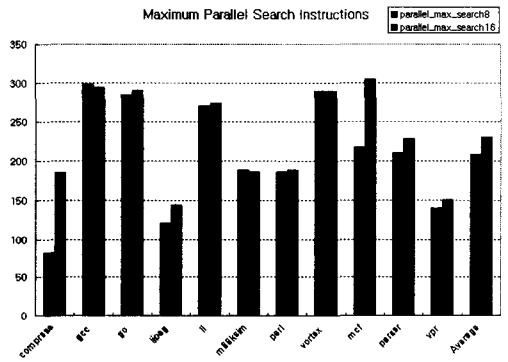


그림 10 병렬검색 적용 시 사이클 당 최대 데이타종속 명령 수

병렬검색 적용 시 한 사이클에 최대로 검색되는 종속 명령의 수를 보여주는 그림으로 8 이슈인 경우 140개 명령이, 16 이슈인 경우 151개의 명령이 병렬로 검색될 수 있음 보여준다(일부 벤치마크의 경우 검색되는 명령의 수가 RUU의 크기 256 보다도 큰 데 이는 한 사이클에 값 예측 실패한 명령이 한 개 이상 되어 중복 검색되었기 때문이다). 즉, 데이타종속 명령 검색을 위해 많은 사이클이 소모되고 또 이를 위한 하드웨어도 무시할 수 없게 된다.

6. 결 론

본 논문에서는 값 예측 실패 시에 잘못 예측된 값을 사용하여 모험적으로 수행된 명령들만을 순차적으로 취소하고 복구한 후에 재이슈하는 값 예측 실패 복구 메커니즘을 연구 제안하였다. 제안된 복구 방식은 값 예측이 틀린 종속명령만을 선택적으로 재이슈하여 불필요한 재이슈를 줄임으로써 값 예측 실패 시에 손실을 줄였다.

또한 잘못 예측된 명령에 종속적인 명령들을 한번에 병렬로 검색하지 않고 명령들의 종속체인을 따라 순차적으로 검색함으로써 프로세서의 클럭 사이클에 영향을 미치지 않으면서 하드웨어의 구현의 복잡성을 줄였다. 제안된 순차적 복구 메커니즘은 심플스칼라 시뮬레이터 상에 구현하고 SPECint 벤치마크 프로그램에 대해 성능을 측정 분석하였다. 실험 결과 제안된 방식이 값 예측 실패 후 모든 명령을 재이슈하는 복구 방식보다 평균 4% 이상 성능 향상을 보였으며 한 클럭에 모든 명령을 병렬검색하는 방식과도 크게 성능 차이를 보이지 않았다.

참 고 문 헌

[1] S.McFarling, "Combining Branch Predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[2] T.Yeh and Y.Patt, "Two-level Adaptive Branch Prediction," Proceedings of the 24th International Symposium Microarchitecture (MICRO-24), Nov. 1991.

[3] B.Calder, G.Reinman and D.Tullsen, "Selective Value Prediction," Proceedings of the 26th International Symposium on Computer Architecture (ISCA-26), May 1999.

[4] Sang-Jeong Lee and Pen-Chung Yew, "On Table Bandwidth and Its Update Delay for Value Prediction on Wide-Issue ILP Processors," IEEE Transaction on Computers, Vol.50 No.8, pp.847~852, Aug. 2001.

[5] Sang-Jeong Lee and Pen-Chung Yew, "On Augmenting Trace Cache for High-Bandwidth Value Prediction," IEEE Transaction on Computers, Vol.51 No.9, pp.1074~1088, Sept. 2002.

[6] M.Lipasti and J.Shen, "Exceeding the Limit via Value Prediction," Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29), Dec. 1996.

[7] M.Lipasti, *Value Locality and Speculative Execution*, Ph.D. Thesis in Electrical and Computer Engineering, Carnegie Mellon University, 1997.

[8] B.Rychlik, J.Faistl, B.Krug, and J.Shen, "Efficacy and Performance Impact of Value Prediction," Parallel Architectures and Compilation Techniques (PACT98), Paris, Oct. 1998.

[9] T.Sato, "Evaluating the Impact of Reissued Instructions on Data Speculative Processor Performance," Microprocessors and Microsystems, Vol. 25, Issue 9-10, pp.469~482, Elsevier, Jan. 2002.

[10] Y.Sazeides and J.Smith, "The Predictability of Data Values," Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30), Dec. 1997.

[11] K.Wang and M.Franklin, "Highly Accurate Data Value Predictions using Hybrid Predictor," Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30), Dec. 1997.

[12] H.Zhou, C.Fu, E.Rotenberg, and T.Conte, *A Study of Value Speculative Execution and Misspeculation Recovery in Superscalar Microprocessors*, Technical Report, ECE Department, N.C. State University, Jan. 2001.

[13] D.Burger and T.Austin, *The SimpleScalar Tool Set, Version 2.0*, Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[14] A.KleinOsowski, J.Flynn, N.Meares, and D.Lilja, "Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research," Workshop on Workload Characterization held in conjunction with International Conference on Computer Design, Sept., 2000.

이 상 정

1983년 2월 한양대학교 공과대학 전자공학과 공학사. 1985년 2월 한양대학교 대학원 전자공학과 공학석사. 1988년 8월 한양대학교 대학원 전자공학과 공학박사 1988년 9월~현재 순천향대학교 공과대학 정보기술공학부 교수. 1999년 2월~2000년 1월 미국 University of Minnesota 방문교수. 관심분야는 고성능 프로세서 설계, 컴파일러, 네트워크 응용

전 병 찬

1992년 2월 한밭대학교 공과대학 전자계산학과 공학사. 1994년 2월 수원대학교 대학원 전산계산학과 이학석사. 2001년 2월 순천향대학교 대학원 전산학과 공학박사. 2002년 3월~현재 청운대학교 컴퓨터과학과 강의전담교수. 관심분야는 컴퓨터 구조, 마이크로프로세서 응용, 모바일 네트워크