논문 2011-48CI-3-7

RAS 오염 방지를 통한 함수 복귀 예측 정확도 향상

(Prediction Accuracy Enhancement of Function Return Address via RAS Pollution Prevention)

김 주 환*, 곽 종 욱**, 장 성 태***, 전 주 식****

(Ju-Hwan Kim, Jong Wook Kwak, Seong Tae Jhang, and Chu Shik Jhon)

요 약

조건 분기 명령어의 예측 정확도가 매우 높아짐에 따라 상대적으로 무조건 분기 명령어의 예측이 중요해지고 있다. 그 중 RAS(Return Address Stack)를 사용하는 함수 복귀 예측은 이론적으로 오버플로가 발생하지 않는 한도 내에서 100%의 정확도를 보여야 한다. 하지만 투기적 실행을 지원하는 현대 마이크로프로세서 환경 하에서는 잘못된 실행 경로로의 수행 결과를 무효화 할 때 RAS의 오염이 발생하며, 이는 함수 복귀 주소의 예측 실패로 이어진다. 본 논문에서는 이러한 RAS의 오염을 방지하기 위하여 RAS 재명명 기법을 제안한다. RAS 재명명 기법은 RAS의 스택을 소프트 스택과 하드 스택으로 나누어 투기적 실행에 의한 데이터의 변경을 복구할 수 있는 소프트 스택에서 투기적 실행에 의한 데이터를 관리하고, 소프트 스택의 크기 제한으로 겹쳐쓰기가 일어나는 데이터 중 이후에 사용될 데이터를 하드 스택으로 옮기는 구조로 구성된다. 또한 이러한 구조의 문제점을 파악하여, 본 논문에서는 RAS 재명명 기법의 추가적 개선법을 소개한다. 제안된 기법을 모의실험 한 결과, RAS 오염 방지 기법이 적용되지 않은 시스템과 비교하여 함수 복귀 예측 실패를 약 1/90로 감소시켰으며, 최대 6.95%의 IPC 향상을 가져왔다. 또한 기존의 RAS 오염 방지 기법이 적용된 시스템과 비교하여 함수 복귀 예측 실패를 약 1/9로 감소 시켰다.

Abstract

As the prediction accuracy of conditional branch instruction is increased highly, the importance of prediction accuracy for unconditional branch instruction is also increased accordingly. Except the case of RAS(Return Address Stack) overflow, the prediction accuracy of function return address should be 100% theoretically. However, there exist some possibilities of miss-predictions for RAS return addresses, when miss-speculative execution paths are invalidated, in case of modern speculative microprocessor environments. In this paper, we propose the RAS rename technique to prevent RAS pollution, results in the reduction of RAS miss-prediction. We divide a RAS stack into a soft-stack and a hard-stack and we handle the instructions for speculative execution in the soft-stack. When some overwrites happen in the soft-stack, we move the soft-stack data into the hard-stack. In addition, we propose an enhanced version of RAS rename scheme. In simulation results, our solution provide 1/90 reduction of miss-prediction of function return address, results in up to 6.85% IPC improvement, compared to normal RAS method. Furthermore, it reduce miss-prediction ratio as 1/9, compared to previous technique.

Keywords : 분기 예측, 함수 복귀 예측, 간접 분기, RAS 오염, 명령어 인출

Ⅰ. 서 론

분기 목적지(branch target)가 계산되기 전에 유용한 명령어들을 인출(fetch)하여 투기적 실행(speculative execution)을 지원하는 프로세서는 분기 예측(branch predict)의 결과에 시스템 성능이 크게 의존한다[1~2]. 그러므로 정확한 분기 예측기는 투기적 실행을 지원하고,

* 정회원, 삼성전자 SOC 개발실
(Samsung Electronics)
** 평생회원-교신저자, 영남대학교 컴퓨터공학과
(Yeungnam University)
*** 정회원, 수원대학교 컴퓨터학과
(The University of Suwon)
**** 평생회원, 서울대학교 컴퓨터공학부
(Seoul National University)
접수일자: 2010년12월16일, 수정완료일: 2011년5월12일

깊은 파이프라인(pipeline)을 가진 마이크로프로세서에서 고성능을 유지하기 위한 필수적인 부분이라 할 수 있다[3]. 이를 위하여 많은 연구들이 고성능 분기 예측기를 제안하였으며, 이러한 고성능 분기 예측기에 의해 조건 분기(conditional branch)의 예측은 매우 정확해졌다[4~8]. 조건 분기의 예측 정확도가 매우 높아짐에 따라, 상대적으로 무조건 분기(unconditional branch)의 예측 정확도가 점차 중요해지고 있다[9]. 또한 직접 분기(direct branch) 뿐만 아니라 간접 분기(indirect branch)의 예측 정확도에 관한 연구들도 이루어지고 있다[10~11].

본 논문에서는 무조건 분기 중 특히 함수 복귀(function return)의 예측 정확도를 향상시키기 위한 새로운 기법을 제안한다. 함수 복귀 예측은 복귀 주소 스택(RAS, return address stack)을 사용한다. RAS를 사용하는 함수 복귀 예측은 이론적으로는 RAS의 오버플로(overflow)가 발생하지 않는 한도 내에서 100%의 예측 정확도를 보인다. 함수 호출 시에 복귀 주소(return address)를 푸시하고, 함수 복귀 시에 이 주소를 다시 팝 하여 사용하며, 이와 같은 푸시와 팝 동작은 완벽하게 상호 일대일 대응이 되기 때문이다[9]. 하지만 투기적 실행을 지원하는 현 마이크로프로세서 환경 하에서는 잘못된 경로의 실행(miss-speculative execution path)을 무효화할 때, 잘못된 경로 안에 포함된 RAS의 푸시와 팝에 의해 RAS의 오염(pollution)이 발생할 수 있다. 또한 파이프라인의 깊이가 깊어지고 단위시간당 제공되는 명령어의 수가 늘어날수록 잘못된 경로에 포함된 RAS의 푸시와 팝 또한 점차 증가하고 있어, RAS의 오염 빈도 또한 점차 증가하고 있다.

이를 위해 본 논문에서는 RAS의 오염을 방지하기 위한 새로운 RAS의 구조를 제안한다. 제안된 방식은 RAS의 오염으로 인한 예측 실패 상황을 줄이기 위하여, 기존의 RAS 구조를 두개의 스택으로 분리한다. 이는 투기적으로 실행된 함수 호출/복귀 명령어들의 정보가 섞여 있는 소프트 스택과 커밋(commit)된 함수 호출/복귀 명령어들의 정보만 들어있는 하드 스택이다. 소프트 스택은 투기적 실행이 무효화될 때 스택 내부의 값들을 복구할 수 있지만 공간을 효율적으로 사용하지 못하는 구조로 되어 있고, 하드 스택은 스택 내부의 값들은 복구할 수 없지만 공간을 효율적으로 사용하는 구조로 이루어져 있다. 이러한 구조로 RAS의 오염을 효율적으로 방지함으로써 함수 복귀 예측의 정확도를 높인다. 이러한 RAS의 오염을 방지하기 위한 기법들은 이미 다수 제안되어 있지만, RAS의 스택을 두 개로 분리하는 시도는 본 논문이 최초이며, 기존 연구들과 비교하여도 본 논문에서 소개하는 방식이 RAS의 오염을 매우 효율적으로 방지하였다. RAS의 오염을 방지하기 위한 여러 기법들은 관련 연구에서 상세히 언급한다.

이하 본 논문의 구성은 다음과 같다. Ⅱ장에서 RAS의 오염을 방지하기 위해 제안되었던 기존의 연구 사례들을 소개한다. Ⅲ장에서는 하드 스택과 소프트 스택을 사용하여 함수 복귀 예측을 수행하는 새로운 기법을 제시한다. Ⅳ장에서는 모의실험을 통해 본 논문에서 제안한 기법의 성능을 검증하며, 끝으로 Ⅴ장에서는 결론을 맺는다.

Ⅱ. 관련 연구

RAS의 오염을 방지하기 위해서 Skadron 등은 투기적 실행을 시작할 때마다 RAS 전체를 검사점(checkpoint)에 기록하는 기법을 제안하였다[12]. 하지만, 전체 RAS를 매번 검사점에 기록하는 방법은 너무나 방대한 공간과 시간을 필요로 하기에 현실적으로 구현이 불가능하다. 이에 대한 대안으로써, TOS(Top of Stack) 포인터가 가리키는 엔트리만을 검사점에 기록하는 기법이 제안되었는데, TOS 포인터 값과 함께 스택의 최상위 엔트리의 값을 검사점에 기록하여 대부분의 RAS의 오염을 방지할 수 있게 하였다. 이 기법은 효과적으로 상당수의 RAS 오염을 복구하여 주지만, 동시에 2개 이상의 엔트리가 오염되었을 경우에는 RAS의 오염을 복구하지 못한다[12].

유사 연구로, self-checkpoint RAS 기법도 제안되었다. 이 기법은 RAS 구조에 TOS 포인터 외에 NEXT 포인터를 유지한다. NEXT 포인터는 항상 다음 푸시가 사용할 엔트리를 가리킨다. 팝이 발생했을 시에 팝 된 엔트리를 스택에서 제거하지 않고 TOS 포인터만 하향 조정된다. 이후 푸시가 발생하면, NEXT 포인터가 가리키는 엔트리에 복귀 주소값을 저장하고, TOS 포인터는 해당 엔트리를, NEXT 포인터는 다음 엔트리를 가리키게 된다. 그리고 새로 푸시 된 엔트리의 NOS(Next on Stack) 필드가 이전 최상위 엔트리, 즉 논리적으로 바로 아래 엔트리를 가리키게 된다. 이처럼 이미 팝 되어 사라진 엔트리를 물리적으로 스택 내에 유지하고, 연결 목록(linked list) 방식으로 논리적 스택 구조를 유지함으로써 RAS의 오염은 막을 수 있다. 하지만 이 방식은

팝 되어 더 이상 쓰이지 않을 엔트리들이 물리적으로 계속 스택 내부에 쌓이게 되기 때문에, 심각한 공간 낭비가 발생한다. 더욱이 낭비되는 공간은 잘못된 경로의 실행을 통해서 쌓이는 것이 아니라, 정상적으로 함수 호출/복귀가 수행되는 과정에서 계속 쌓여가기 때문에, 스택의 크기가 아무리 크더라도 결국 겹쳐쓰기(overwrite)가 발생하여 유효한 데이터가 지워지게 된다. 이는 RAS의 오염이 줄어든 만큼 스택 오버플로가 더 많이 발생하게 된 것이다. 본 논문에서 제안하는 기법은 이 기법의 단점을 보완한 기법이라고 할 수 있다[13].

Hans 등은 추가적인 테이블을 사용하여 스택의 오염과 스택 오버플로가 발생한 엔트리를 찾아내는 기법을 소개하였다. 함수 호출/복귀 시에 테이블에 정보를 기록하여 스택 내의 오염된 엔트리를 판별한다. 스택의 오염이나 오버플로가 발생한 것으로 판별된 엔트리를 사용한 함수 복귀 예측은 RAS를 이용한 예측값을 버리고 대신 BTB를 이용한 예측값을 사용한다. 이 기법은 RAS의 오염 자체를 막는 것이 아니라, 오염을 탐지 하여 오염된 값 대신에 차선책을 사용하는 방법으로, 함수 복귀 예측의 정확도를 높이는 점에서는 본 논문에서 제안하는 방식과 유사한 목표를 가지고 있지만, 본 논문에서 제안하는 기법과는 그 범주가 다르다[9].

복귀 스택 휴지통(RST, Return Stack Trashcan) 기법도 RAS의 오염을 막아주는 기법으로 제안되었다. 이 방식은 투기적 실행에 사용된 모든 함수 호출과 복귀의 정보를 RST에 저장하고 있는 방식이다. 이는 확실히 RST에 저장된 정보를 사용하여 RAS의 오염을 수정할 수 있지만, RST 엔트리를 하나하나 살피면서 RAS의 정보를 수정해야함으로 복구에 많은 시간이 소요된다는 단점이 있다[14]. 한편, 최근에 제안된 명령어 인출 기법 중에는, 잘못된 실행 경로를 무효화 할 때 전체 시스템의 무효화가 끝나기 전에 분기 예측 모듈의 복구를 끝내고 분기 예측을 먼저 시작하는 기법도 있다. 하지만, RST 기법은 복구에 시간이 너무 오래 소요되어 이러한 기법에는 적용이 불가능하다[15]. 또한 RST에 RAS에서 사라진 복귀 주소를 모두 저장해야하기 때문에, RST에 상당량의 하드웨어 공간을 할당해야 하며 또한 투기적 실행에 포함된 함수 호출/복귀 횟수가 RST의 크기를 넘어서는 경우는 복구할 수 없다는 문제점이 있다.

Ⅲ. RAS 오염 방지 기법

1. RAS 오염

투기적 실행(speculative execution)을 지원하는 프로세서 환경에서의 RAS는 잘못된 경로로의 실행을 무효

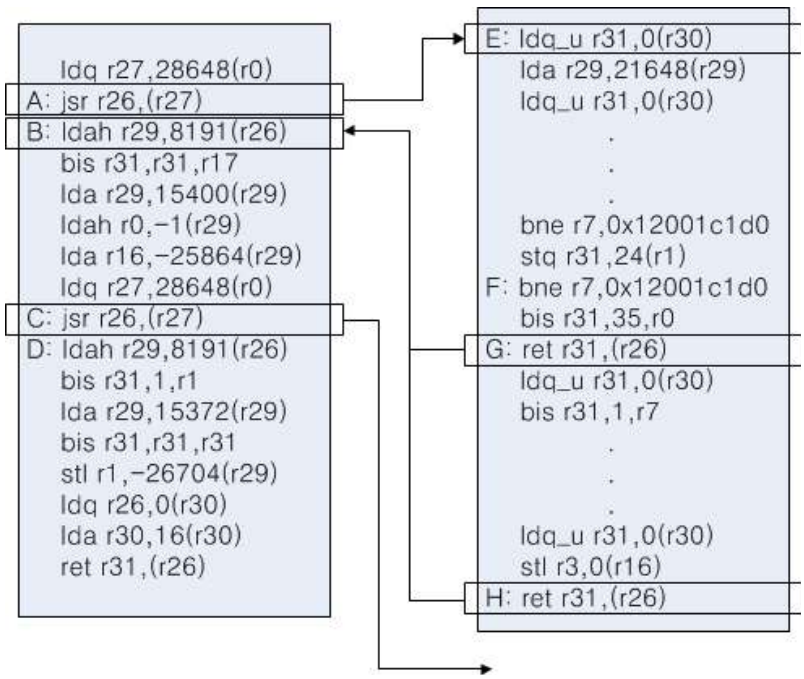

그림 1. 256.bzip2 코드의 투기적 실행 경로 사례
Fig. 1. Speculative execution path example of 256.bzip2.

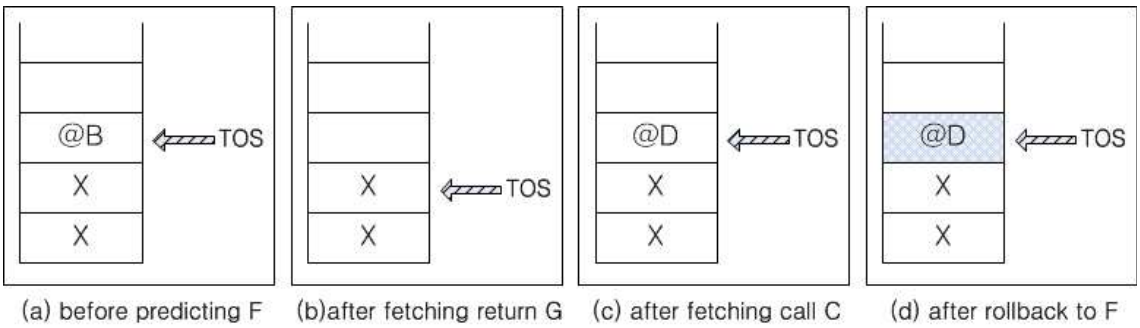

그림 2. RAS 오염 사례 예제
Fig. 2. Example of RAS pollution.

화할 때 RAS를 이전 상태로 올바르게 되돌려야 한다. 이를 위해 현재는 TOS 포인터만을 백업(backup) 하는 방식을 사용한다. 이 방식은 투기적 실행을 시작하는 명령어가 인출될 때, 그 시점의 TOS 포인터의 값을 검사점에 기록한다. 이 때 해당 경로의 실행 무효화가 발생 할 경우, 투기적 실행을 시작하는 명령어가 검사점에 기록한 TOS 포인터의 값을 현재의 RAS의 TOS에 덮어쓴다[16~17]. 이러한 방법은 TOS 포인터의 값은 복구할 수 있지만, RAS의 각 엔트리에 저장된 값이 변경되었다면 이 값들은 복구가 불가능하다. 다음과 같은 사례가 있을 수 있다. 그림 1은 SPEC 벤치마크 프로그램 가운데 하나인 256.bzip2 코드의 일부분이며, 그림 2는 그림 1의 코드를 실행하는 동안 발생하는 RAS의 상태 변화를 보여준다.

우선, 함수 호출 명령어인 명령어 A가 인출되면 그림 2의 (a)와 같이 함수 호출 명령어 A의 복귀 주소인 명령어 B의 주소를 RAS에 푸시 한다. 이후 호출된 함수로 넘어가서 명령어 E를 시작으로 이하 명령어들의 실행을 진행한다. 그 과정에서 분기 명령어인 명령어 F가 인출되게 되면, 분기 예측기의 예측 결과값을 사용하여 투기적 실행이 시작된다. 투기적 실행이 발생할 때 마다 TOS의 값을 체크 포인팅 하며, 주어진 사례에서는 그림 2의 (a)에 해당하는 TOS 값을 검사점에 기록한다. 일단 분기 명령어 F의 예측값이 NotTaken이라고 가정하면, 이후 투기적 실행을 진행하는 과정에서 함수 복귀 명령어인 명령어 G를 인출하게 되고, RAS의 최상위에 있는 값인 "@B*"를 팝하고 그 값을 사용하여 복귀 주소를 예측한 후, 다시 명령어 B부터 명령어 인출을 계속 진행한다. 그 이후 함수 호출 명령어인 명령어 C가 인출되면 다시 RAS에 명령어 C의 복귀 주소인 "@D"를 푸시 하여 그림 2의 (c)와 같은 상태가 된다.

이 시점에서 투기적으로 실행된 분기 명령어 F의 예측값이 틀렸다는 것을 알게 되었다고 가정하자. 이 경우, 명령어 F 이후에 인출된 명령어들은 모두 무효화되어야 한다. 이 과정에서 RAS의 TOS 포인터가 명령어 F의 검사점에 기록되어 있는 TOS의 값으로 복구되어 최종적으로 (d)와 같은 상태가 된다. 무효화 이후의 RAS는 그림 2의 (a)와 같은 상태가 되어야 하지만, 실제로는 (d)와 같은 상태가 된다. 즉, "@B"대신에 "@D" 값이 잘못 저장되어 있다. 분기 명령어 F의 분기 예측 실패를 복구하고 다시 명령어 인출을 진행하면 함수 복귀 명령어 H가 인출되게 된다. 함수 복귀 명령어 H는 명령어 B의 주소로 복귀하여 계속 인출을 진행하여야 하지만 RAS의 최상위 엔트리에 잘못된 값인 "@D"가 저장되어 있기 때문에, 결과적으로 복귀 명령어 H의 복귀 주소를 잘못 예측하게 된다.

2. RAS 오염 방지를 위한 RAS 재명명 기법

본 논문에서는, 주어진 사례에서 확인된 바와 같은 RAS의 오염을 방지하기 위해 RAS 재명명 기법을 제안한다. 제안하는 RAS 재명명 기법은 두 개의 스택을 가진다. 하나는 기존의 RAS와 유사한 방식으로 동작하는 하드 스택(hard-stack)이며, 또 다른 하나는 새롭게 추가된 소프트 스택(soft-stack)이다. RAS 외부에서의 푸시는 소프트 스택으로만 가능하다. 그러면 후입선출 버퍼인 스택의 특성상 팝 또한 소프트 스택에서 주로 발생하게 된다. 하드 스택은 소프트 스택의 공간 부족으로 지워지는 데이터 중 이후에 사용될 데이터만을 저장한다.

하드 스택은 기존의 원형 스택과 동일한 구조를 가지

* 본 논문에서는 특정 "명령어"의 주소값을 "@명령어"와 같이 표기한다.

```
FETCH FUNCTION CALL INSTRUCTION:
BEGIN
SoftStack[SoftBottom] <= Return Address
Softstack[SoftBottom].next <= SoftTOS
SoftTOS <= SoftBottom
increase SoftBottom
END

FETCH FUNCTION RETURN INSTRUCTION:
BEGIN
if used_Stack = Soft then
   Predicted Value <= SoftStack[SoftTOS]
   SoftTOS <= SoftStack[SoftTOS].next
end if
if used_Stack = Hard then
   Predicted Value <= HardStack[HardTOS]
   decrease HardTOS
end if
END
```

```
ROLL BACK WRONG PATH INSTRUCTION:
BEGIN
if CheckPoint.used_Stack = Hard then
   used_Stack <= Hard
   recover HardTOS
end if
if CheckPoint.used_Stack == Soft then
   used_Stack <= Soft
   SoftTOS <= CheckPoint.SoftTOS
   recover HardTOS
end if
END
```

그림 3. 소프트 스택의 동작과 next 포인터 사용 알고리즘
Fig. 3. Algorithm for operation of soft-stack and use fo next pointer.

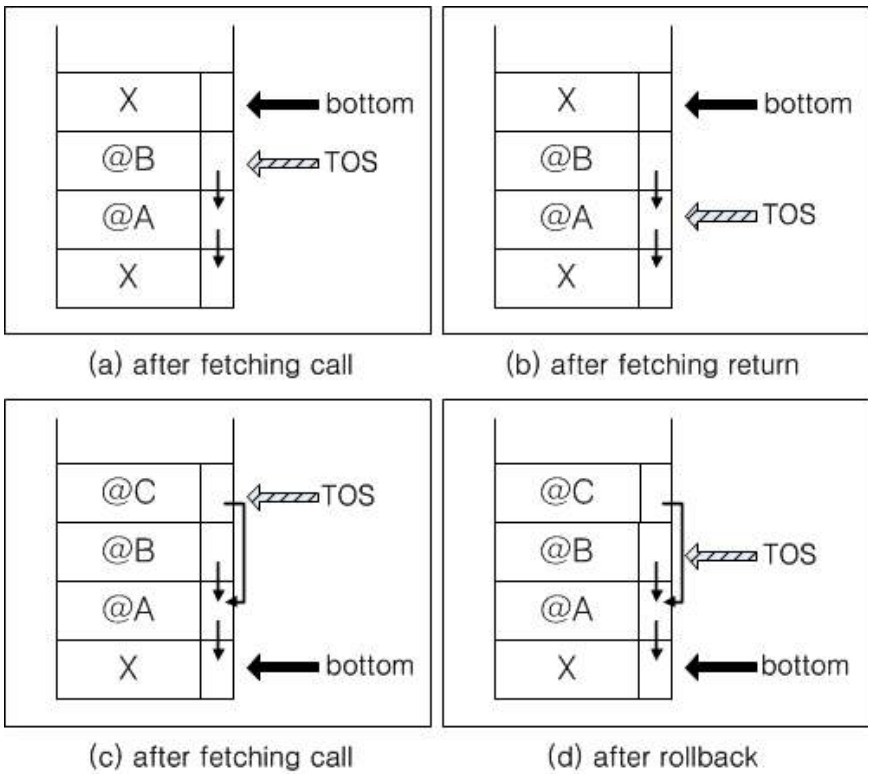

그림 4. 소프트 스택 동작 및 next 포인터의 사용 예제
Fig. 4. Example for peration of soft-stack and use of next pointer.

고 있으며, TOS 포인터와 함께 스택 오버플로를 탐지하기 위하여 bottom 포인터를 가지고 있다. 소프트 스택은 잘못된 실행 경로에 대한 무효화가 발생했을 때 스택 내의 값들을 복구하기 위하여 사용된다. 이를 지원하기 위한 소프트 스택의 구조는 원형 스택의 형태를 가지며, 각각의 엔트리를 가리키기 위하여 TOS 포인터와 bottom 포인터가 존재한다. 그리고 각각의 엔트리는 복귀 주소값, next 포인터, CC 비트, RC 비트, T 비트로 구성된다. 기존의 원형 스택 구조의 RAS와 비교하여 추가된 각 요소들의 기능은 다음과 같다.

우선 기존의 TOS 포인터와 함께 bottom 포인터가 추가되었다. 이 bottom 포인터는 스택의 바닥, 즉 다음 푸시에서 겹쳐쓰기 할 엔트리를 가리키고 있다. 그리고 소프트 스택의 각각의 엔트리에 next 포인터, CC 비트(call commit bit), RC 비트(return commit bit), T 비트(toggle bit)가 추가되었다. next 포인터는 논리적인 아래 엔트리를 가리키기 위해 사용되고, CC 비트와 RC비트는 겹쳐쓰기가 발생할 때 해당 엔트리가 이후에 사용될 데이터인지를 판별하기 위해 사용되며, T 비트는 논리적 아래 엔트리가 하드 스택으로 옮겨졌는지를 판별하기 위해 사용된다. 그림 4와 그림 6 그리고 그림 8은 각각 소프트 스택의 next 포인터의 동작 예제, CC/RC 비트의 동작 예제, 그리고 T 비트의 동작 예제를 보여준다.

소프트 스택은 next 포인터를 사용하여 연결 목록처럼 구현된다. 물리적인 아래 엔트리와 논리적인 아래 엔트리가 서로 다른 것이다. 논리적인 아래 엔트리는 next 포인터가 가리키는 엔트리가 된다. 그림 3은 소프트 스택의 동작과 next 포인터의 사용을 알고리즘으로 나타낸 것이다.

그림 4는 위의 알고리즘의 동작 예제를 나타낸 것이다. 그림 4는 소프트 스택의 구조를 나타내고 있다. 스택의 각각의 엔트리 오른쪽에 추가된 필드가 next 포인터를 나타내며, next 포인터 필드에 그려진 화살표들은 각각의 next 포인터가 가리키는 엔트리를 나타낸다. 함수 복귀 주소 “@A”와 “@B”가 소프트 스택에 이미 들어가 있다고 가정하면, (a)와 같은 형태가 된다. 이 시점에서 투기적 실행이 시작되었다고 하면 (a)의 TOS 값을 검사점에 기록하게 된다. 여기서 함수 복귀 명령어가 인출되어 “@B”값을 팝 하게 되면 (b)와 같은 형태가 된다. bottom 포인터는 이동하지 않으며, TOS 포인터의 위치에 있던 “@B”값을 가져가고 TOS 포인터는 next 포인터가 가리키는 자리로 이동하게 된다. 그 이후, 만약 복귀 주소가 “@C”인 함수 호출 명령어가 인출되게 되면, (c)와 같은 형태가 된다. TOS 포인터가 (b)에서 가리키고 있던 엔트리 바로 윗자리인 “@B”가 기록되어 있는 자리가 아니라, bottom 포인터가 (b)에서 가리키고 있던 자리에 “@C”값을 푸시 한다. 그리고 새롭게 푸시 된 “@C”가 기록된 엔트리의 next 포인터를 TOS 포인터가 (b)에서 가리키고 있었던 “@A”가 기

```
COMMIT FUNCTION CALL INSTRUCTION:
BEGIN
SoftStack[CheckPoint.SoftTOS].CC <= set
END

COMMIT FUNCTION RETURN INSTRUCTION:
BEGIN
SoftStack[CheckPoint.SoftTOS].RC <= set
END
```

```
FETCH FUNCTION CALL INSTRUCTION:
BEGIN
if CC = set & RC = unset then
  increase HardTOS
  HardStack[HardTOS] <= SoftStack[SoftBottom]
endif
SoftStack[SoftBottom] <= Return Address
SoftStack[SoftBottom].CC <= unset
SoftStack[SoftBottom].RC <= unset
Softstack[SoftBottom].next <= SoftTOS
SoftTOS <= SoftBottom
increase SoftBottom
END
```

그림 5. CC/RC 비트 사용 알고리즘
Fig. 5. Algorithm for use of CC/RC bits.

록되어 있는 자리를 가리키게 만든다. 물리적으로 "@C" 아래에는 "@B"가 있지만, 논리적으로는 "@C" 아래에 "@A"가 있다는 것을 기록해 두는 것이다. 그리고 bottom 포인터는 하나 위의 엔트리를 가리키게 되지만 원형 버퍼로 구현되어 있기 때문에 (c)에서 보이는 바와 같이 물리적으로는 가장 아래의 엔트리를 가리키게 되었다.

여기서 투기적 실행이 잘못되었다는 것이 발견되어 "@C" 값을 푸시 한 것과 "@B" 값을 팝 한 것을 무효화하여 (a)와 같은 상태로 돌아가야 하는 상황이 발생하면 (d)에서 보이는 바와 같이, 검사점에 기록되어 있던 TOS 값을 이용하여 TOS 포인터의 위치만을 변경한다. 물리적인 소프트 스택의 상태를 보면 (a)와 (d)는 서로 다르다. "@C"가 겹쳐쓰기 되어 있으며 이로 인해 bottom 포인터의 위치가 다르지만, "@C"의 겹쳐쓰기로 인해 지워진 데이터는 이후에 설명할 방법을 사용하여 하드 스택으로 옮겨져 있다. 이를 고려하면, TOS 포인터가 가리키는 엔트리부터 next 포인터를 따라가는 논리적인 스택은 동일한 형태를 가지고 있다. 이와 같이 본 논문에서 소개된 소프트 스택은, 투기적 실행으로 인하여 팝 된 데이터를 물리적으로 유지하기 때문에, 무효화가 발생 했을 때 이전의 상태로의 정확한 복귀가 가능하다.

CC/RC 비트는 소프트 스택에서 겹쳐쓰기가 발생할 때 해당 엔트리를 하드 스택으로 옮길 것인지 삭제할 것인지를 결정하기 위해 사용된다. 하드 스택으로 옮길 것으로 결정되면, 겹쳐쓰기가 발생한 엔트리, 즉 소프트 스택의 bottom 포인터가 가리키고 있던 엔트리의 복귀 주소값만을 임시 버퍼(buffer)에 복사한다. 해당 엔트리의 나머지 필드의 값들은 하드 스택으로 옮기는 과정에

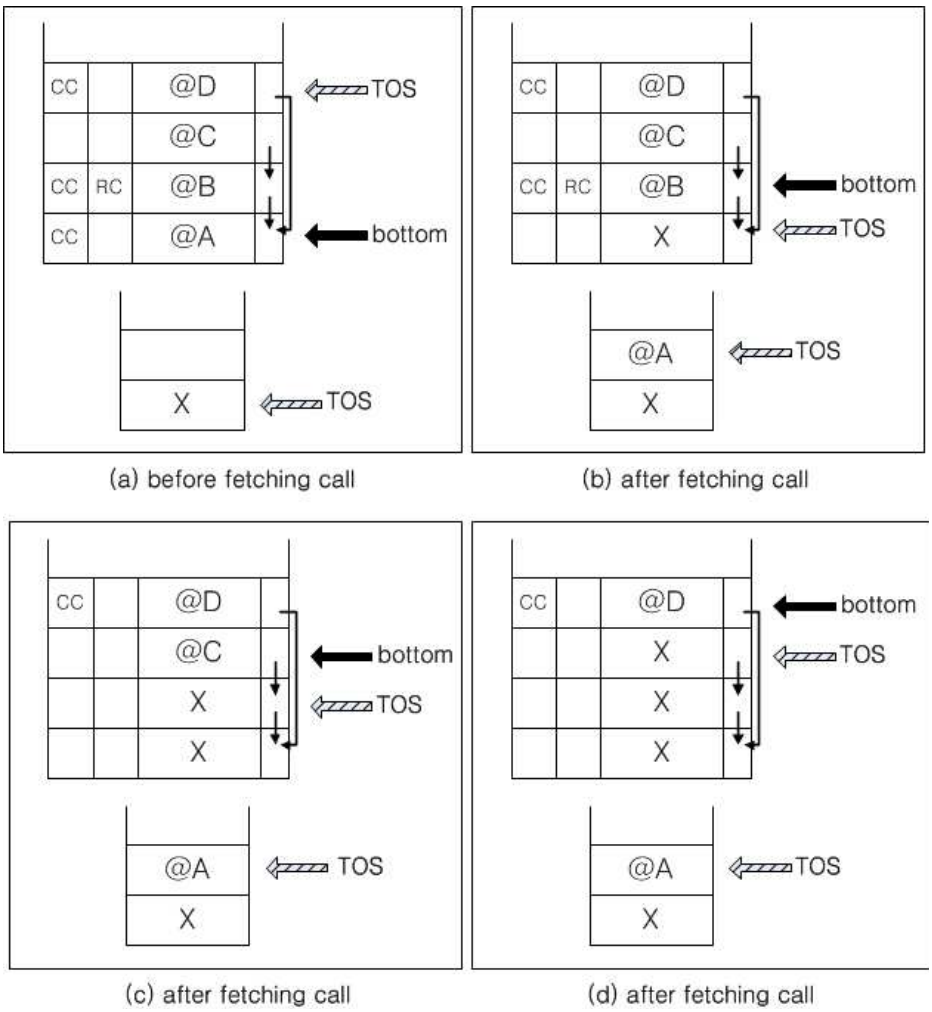

그림 6. CC/RC 비트 사용 예제
Fig. 6. Example for use of CC/RC bits.

서는 사용되지 않는다. 그리고 소프트 스택에서 겹쳐쓰기가 진행됨과 동시에 하드 스택의 TOS 포인터의 값을 하나 증가시키고, 증가된 TOS 포인터가 가리키는 엔트리에는 앞서 임시 버퍼에 복사해둔 복귀 주소값을 기록한다. CC 비트와 RC 비트는 각각 함수 호출과 함수 복귀 명령어가 커밋 될 때 자신이 사용한 RAS의 엔트리에 자신이 커밋 되었다는 것을 표시해주는 비트이다. 기존 RAS 구조에서 투기적 실행을 시작하는 명령어들이 인출 될 때 RAS의 TOS 포인터 값을 검사점에 기록한다. 함수 호출과 복귀 명령어 또한 예측값을 이용하여 투기적 실행을 시작하는 명령어들이므로, 인출 시점의 TOS 포인터의 값을 검사점에 기록한다. 이 경우 검사점에 기록되는 TOS 포인터의 값은 자신이 사용한 RAS 엔트리를 가리키고 있다. 한편, BTB와 분기 예측기의 갱신을 위하여 함수 호출/복귀 명령어를 포함한 모든 분기 명령어는 커밋 시점에 분기 예측 모듈에 재접근하게 된다. 따라서 소프트 스택의 CC/RC 비트는 추가적인 오버헤드 없이 함수 호출/복귀 명령어가 커밋되는 시점에 충분히 기록이 가능하다. 주어진 CC/RC 비트의 정보를 활용하여 해당 엔트리의 값이 이후에 사용될지의 여부를 판별할 수 있다. CC 비트가 표시되어 있고, RC 비트가 표시되어 있지 않은 경우, 해당 엔트리는 함수 호출 명령어에 의해서 푸시 되었지만, 함수 복귀 명령어가 아직 팝 하지 않은 엔트리이다. 따라서 이는 추후 사용될 엔트리임으로 하드 스택으로 복사한다. CC 비트와 RC 비트가 모두 표시되어 있는 경우, 해당 엔트리는 함수 복귀 명령어가 이미 사용한 엔트리로, 이는 다시 사용되지 않는다. CC 비트와 RC 비트가 모두 표시되어 있지 않은 경우는 해당 엔트리를 푸시한 함수 호출 명령어가 잘못된 경로에 포함되어 무효화되어 버린 엔트리이다. 마찬가지로 이미 무효화 된 엔트리이기 때문에 앞으로 사용되지 않을 엔트리이다. 마지막으로 RC 비트만 표시 되고 CC 비트는 표시 되지 않는 상황은 발생하지 않는다. 그림 5는 이를 알고리즘으로 나타낸 것이다.

그림 6은 위의 알고리즘을 사용하는 과정을 상세히 보여주고 있다. 그림 6의 위쪽 스택은 소프트 스택이고, 아래쪽 스택은 하드 스택이다. (a)의 상태에서 시작하여 (b), (c), (d)에서 겹쳐쓰기가 발생할 때 각각의 CC/RC 비트의 상태에 따라서 해당 엔트리를 하드 스택으로 이동시키거나 삭제하는 것을 보여준다.

마지막으로 T 비트는 next 포인터가 가리키고 있는 논리적 아래 엔트리가 이전에 포인터를 설정할 때와 동일한 엔트리인지 아니면 겹쳐쓰기가 발생하여 다른 엔트리로 교체되었는지를 구분하기 위해 사용된다. 만약 겹쳐쓰기에 의해 다른 엔트리로 교체되었다면 이전의 엔트리는 하드 스택으로 이동되어 있다. T 비트는 해당 엔트리에 값을 기록할 때마다 토글 된다. 그리고 next

```
FETCH FUNCTION CALL INSTRUCTION:
BEGIN
 SoftStack[SoftBottom] <= Return Address
 toggle SoftStack[SoftBottom].T
 Softstack[SoftBottom].next <= SoftTOS
 Softstack[SoftBottom].next.T                    <=
Softstack[SoftTOS].T
 SoftTOS <= SoftBottom
 increase SoftBottom
END
```

```
FETCH FUNCTION RETURN INSTRUCTION:
BEGIN
if used_Stack = Soft then
   Predicted Value <= SoftStack[SoftTOS]
   tempT <= SoftStack[SoftTOS].next.T
   SoftTOS <= SoftStack[SoftTOS].next
   if tempT != SoftStack[SoftTOS].T then
      used_Stack = Hard
   end if
end if
if used_Stack = Hard then
   Predicted Value <= HardStack[HardTOS]
   decrease HardTOS
end if
END
```

그림 7. T 비트 사용 알고리즘
Fig. 7. Alogorithm for use of T bit.

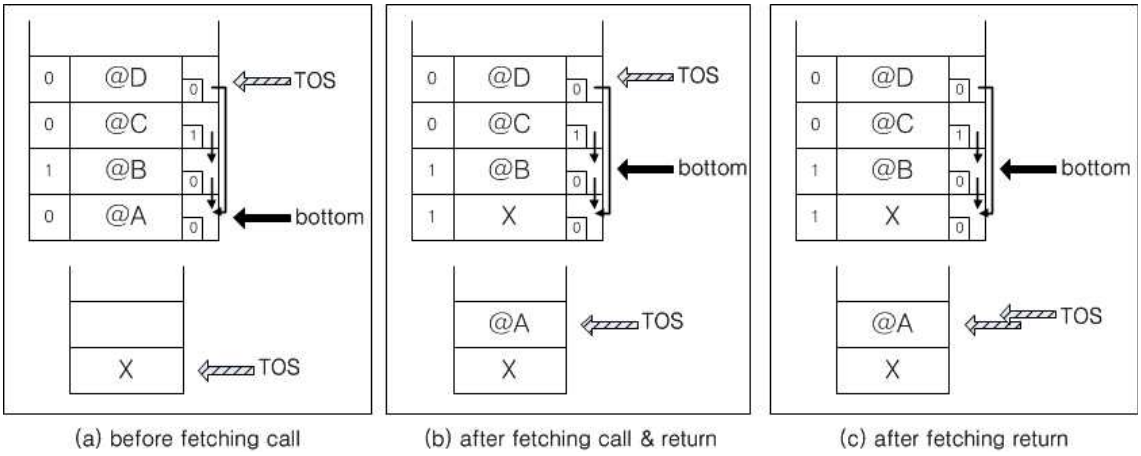

그림 8. T 비트 사용 예제
Fig. 8. Example for use of T bit.

포인터에도 1비트가 추가되어 next 포인터가 가리키는 엔트리의 T 비트의 값을 저장하고 있다. 이를 비교하여 next 포인터가 가리키는 엔트리가 하드 스택으로 이동하였는지를 구분한다. 그림 7은 이를 알고리즘으로 나타낸 것이다.

구체적인 동작 예제는 그림 8에 나타나있다. 그림 8의 위쪽 스택은 소프트 스택이며 아래쪽 스택은 하드 스택이다. 소프트 스택의 각각의 엔트리 왼쪽에 추가된 필드가 T 비트이며, 오른쪽에 있는 필드는 next 포인터 필드이다. next 포인터 필드 안의 작은 박스는 next 포인터에 추가된 1비트를 의미한다. (a)와 같이 소프트 스택과 하드 스택이 구성되어 있다고 가정하자. 새로운 엔트리가 푸시 될 때, 새로운 엔트리의 next 포인터가 이전 최상위 엔트리를 가리키게 하며, 이전 최상위 엔트리의 T 비트의 값을 새로운 엔트리의 next 포인터에 추가된 1비트에 기록해둔다. (a)에서는 아직 겹쳐쓰기가 발생하지 않았으므로 next 포인터 필드에 추가된 1비트는 next 포인터가 가리키는 엔트리의 T 비트와 일치하는 것을 볼 수 있다. 함수 호출 명령어가 인출되어 소프트 스택의 “@A” 값이 기록되어 있던 엔트리에 겹쳐쓰기를 하게 되면, (b)와 같이 “@A” 값은 하드 스택으로 옮겨지고 해당 엔트리의 T 비트 값이 0에서 1로 토글하게 된다. 이후 함수 복귀 명령어가 인출되어 소프트 스택의 현재의 최상위인 “@D” 값을 팝 하게 되면, TOS 포인터는 “@D”가 기록되어 있는 엔트리의 논리적 아래 엔트리로 이동해야 한다. 논리적 아래 엔트리는 현재 엔트리의 next 포인터가 가리키는 엔트리로써, 이 경우 “@D”가 기록된 엔트리의 논리적 아래 엔트리는 “X”가 기록된 엔트리이다. 하지만 이 “X”가 기록되어 있는 엔트리는 겹쳐쓰기로 인하여 교체된 다른 엔트리이며, “@D”가 기록된 엔트리의 진짜 논리적 아래 엔트리인 “@A”는 하드 스택으로 이동되어 있다. “X”가 기록된 엔트리는 겹쳐쓰기 시점에 T 비트의 값이 토글되어 0에서 1로 바뀌었으므로, “@D”가 기록되어 있는 엔트리의 next 포인터에 추가된 1비트에 기록되어 있는 값인 0과 일치하지 않는다. 이것은 다음 팝에 사용해야 하는 현재 최상위 엔트리의 논리적 아래 엔트리가 하드 스택으로 옮겨졌다는 뜻이므로, 이후의 함수 복귀 예측에는 하드 스택을 사용해야 한다. 그림 8의 (c)에서 소프트 스택의 TOS가 하드 스택의 TOS와 같은 곳을 가리키고 있는 것은 이후의 팝에 사용될 엔트리는 하드 스택의 최상위 엔트리라는 의미를 도식적으로 나타낸 것이다.

3. RAS 재명명 기법의 개선

위에서 설명한 바와 같이 RAS 재명명 기법을 사용하면 RAS 오염을 방지하기 위하여 소프트 스택과 하드 스택을 구분하여 사용한다. 소프트 스택을 이용하여 잘못된 실행 경로의 무효화가 발생하면 스택의 논리적 상태를 이전 상태로 복구하고, 소프트 스택에 발생하는 겹쳐쓰기로 인한 데이터의 유실을 하드 스택을 이용하여 방지한다. 하지만 근본적으로 함수 호출/복귀 명령어의 실행과 투기적 실행이 원자적 연산(atomic operation)이 아니기 때문에, 다음과 같은 세 가지 문제 상황이 발생한다. 본 논문에서는 이를 함수 호출 오류와 함수 복귀 오류, 무효화 오류라 명명한다. 다음에서

이 세 가지 문제의 원인과 해결법을 소개하였다.

가. 함수 호출 오류

이는 함수 호출 명령어가 커밋 되기 전에 소프트 스택내의 해당 엔트리가 겹쳐쓰기 되는 경우에 발생한다. 정상적으로 함수 호출 명령어가 커밋 되면 CC 비트에 해당 정보를 표시해야 하지만, 아직 실행중이기 때문에 CC 비트에 표시 되어 있지 않다. 겹쳐쓰기가 발생하면, 함수 호출 명령어는 무효화되지 않고 현재 실행 중임에도 불구하고, 소프트 스택은 해당 함수 호출 명령어가 무효화되어 CC 비트에 표시가 없다고 잘못된 판단을 하여, 해당 엔트리를 삭제하게 된다.

하지만 이와 같은 상황은 소프트 스택의 크기가 일정 크기 이상이 되면 발생하지 않는다. 함수 호출 명령어가 인출되면서 기록한 엔트리에 겹쳐쓰기가 발생하려면 소프트 스택의 전체 엔트리를 모두 한번 씩 겹쳐쓰기 한 이후라야 가능하다. 또한, 명령어 창(instruction window)의 크기가 제한되어 있고, 동시에 실행 가능한 명령어의 숫자도 한계가 있기 때문에 스택 크기가 일정 이상이면 함수 호출 오류는 발생이 불가능 하다. 본 논문의 모의실험에 의하면 소프트 스택의 크기가 8 일 때 해당 상황의 발생 비율은 0.02%이며, 소프트 스택의 크기가 16 이상일 때는 전혀 발생하지 않았다.

나. 함수 복귀 오류

이는 함수 복귀 명령어가 커밋 되기 전에 소프트 스택 내의 해당 엔트리에 겹쳐쓰기가 되는 경우 발생한다. 마찬가지로 함수 복귀 명령어가 아직 커밋 되지 않았기 때문에 RC 비트는 아직 표시되어 있지 않다. 이로 인해 함수 복귀 명령어는 이미 해당 엔트리의 데이터를 사용하였음에도 불구하고, 해당 함수의 복귀 명령어가 아직 인출되지 않아서 RC 비트가 표시되어 있지 않다고 잘못된 판단을 한다. 이로 인해 해당 엔트리를 삭제하지 않고, 이를 하드 스택으로 옮기게 된다.

하지만 이러한 문제 상황은 함수 복귀 명령어의 커밋 시점에 간단히 발견할 수 있다. 명령어 인출과 명령어 커밋은 항상 순차적으로 진행되어야만 하기 때문에, 함수 복귀 명령어와 대응 되는 함수 호출 명령어는 함수 복귀 명령어보다 항상 일찍 커밋 되며, 해당 엔트리에 겹쳐쓰기 한 함수 호출 명령어는 현재 커밋 된 함수 복귀 명령어보다 반드시 늦게 커밋 된다. 그러므로 함수 복귀 명령어의 커밋 시점에 해당 엔트리의 CC 비트가 표시되어 있지 않다면, 해당 엔트리는 이미 하드 스택으로 옮겨지고 다른 엔트리가 겹쳐쓰기 된 것으로 판단할 수 있다. 삭제해야 할 엔트리가 하드 스택으로 옮겨졌으므로, 해당 엔트리를 찾아서 삭제해야 한다. 본 논문에서는 해당 엔트리가 항상 하드 스택의 최상위에 있다고 가정하고 하드 스택의 최상위 엔트리를 삭제하였다. 모의실험에 의하면 하드 스택의 최상위에 해당 엔트리가 있는 경우가 95.35%이다.

다. 무효화 오류

이는 잘못된 실행 경로에 대한 무효화가 발생했을 경우, 복구되는 소프트 스택의 TOS 포인터가 가리키는 엔트리가 겹쳐쓰기 되는 경우 발생한다. 잘못된 예측값을 기반으로 한 투기적 실행으로 인해 무효화가 발생했을 때, RAS는 검사점에 기록되어 있던 TOS 값을 복구한다. 하지만, 복구된 TOS 포인터가 원래 가리켜야 했던 엔트리가 하드 스택으로 옮겨졌을 경우, 복구된 소프트 스택의 TOS 포인터는 잘못된 엔트리를 가리키게 된다.

이러한 상황을 방지하기 위해 소프트 스택의 TOS 값을 검사점에 기록할 때 TOS 포인터가 가리키는 엔트리의 T 비트도 같이 검사점에 기록한다. 무효화로 인해 TOS 값을 복구 할 때, T 비트의 값을 함께 비교하여 서로 다를 경우 해당 엔트리는 이미 하드 스택으로 옮겨졌다고 판단한다. 복구된 TOS 포인터가 가리켜야 할 진짜 엔트리는 이미 하드 스택으로 옮겨졌으므로, TOS 포인터가 올바른 엔트리를 가리키도록 하기 위해, 하드 스택에서 해당 엔트리를 찾아 그 엔트리를 가리키게끔 TOS 값을 수정해야 한다. 본 논문에서는 항상 해당 엔트리가 하드 스택의 최상위에 있다고 가정하고 TOS 포인터가 하드 스택의 최상위를 가리키도록 하였다. 본 논문의 모의실험에 의하면 해당 엔트리가 하드 스택의 최상위에 있는 경우가 99.62%이다.

Ⅳ. 모의실험 및 성능 분석

1. 모의실험 환경 및 벤치마크 프로그램

본 논문에서의 모의실험은 구동 기반 시뮬레이터인 SimpleScalar로 진행되었다[18]. 이벤트 구동형 시뮬레이터(event-driven simulator)인 SimpleScalar는 빠른 모의실험과 높은 정확도의 결과를 보장하는 강력한 모의실험 환경으로, 비순차적 실행(out-of-order execution),

표 1. 모의실험 인자
Table 1. Argument of experiments.

Parameter	Value
Fetch Queue	16 entries
Fetch, Decode Width	8 instructions
ROB entries	64 entries
LSQ entries	32 entries
Functional Units(integer)	8 ALUs, 2 Mult/Div
Functional Units(floating point)	8 ALUs, 2 Mult/Div
Branch Predictor	gshare, 4M entries
BTB entries	4096(1024 X 4-way) entries
RAS entries	64 entries
Branch Misprediction Penalty	11 cycles
L1 I-Cache	32 KB, direct map, 32B line, 1 cycle
L1 D-Cache	32 KB, 4-way, 32B line, 1 cycle
L2 Cache(unified)	2 MB, 4-way, 64B line, 6 cycles
Memory Latency	first_chunk=80 cycles, inter_chunk=12 cycles

비중단 캐시(non-blocking cache), 투기적 실행(speculative execution)등과 같은 최신 프로세서 기술을 지원함과 아울러, 가장 널리 사용되는 gshare 예측기를 포함하여 다양한 분기 예측 기법의 사용을 가능하게 한다[19]. 표 1에 본 논문에서 사용된 실험 환경이 제시되어 있다.

한편 모의실험에 사용된 벤치마크 프로그램은 SPEC에서 제공하는 CPU 성능 측정 프로그램인 SPEC CINT 프로그램들이며, 이 가운데 실험 환경에서 지원하는 10개의 벤치마크를 사용하였다. 일반적으로 SPEC에서 제공하는 CPU 성능 평가 프로그램은 정수형 프로그램인 CINT와 실수형 프로그램인 CFP로 구분되는데, CFP 프로그램의 경우는 과학 계산 형태의 응용 프로그램이 주를 이루며, 이들은 매우 정형화되어 있다. 이 같은 경우, 분기 예측의 정확도가 매우 높게 나타나기 때문에, 분기 예측의 개선으로 인한 성능 향상의 정도를 효율적으로 관찰하기에는 곤란하다. 따라서 CFP 프로그램들은 함수 복귀 예측을 포함하여 분기 예측과 관련된 연구에서 일반적으로 제외된다[20].

2. 실험 결과 및 분석

우선 표 2는 RAS 재명명 기법이 적용된 시스템에서의 MPKI(Miss Per Kilo Instructions)과 IPC(Instructions Per Cycle)를 나타낸다. “base”는 RAS 오염 방지 기법이 적용되지 않은 시스템을 의미하며, “CT_ras”는 관련 연구에서 언급한 기법으로 Skadron이 제안한 RAS의 TOS를 검사점에 기록할 때 RAS의 최상위 엔트리도 같이 검사점에 기록하는 기법을 구현한 것이다. “Rren”은 본 논문에서 제안하는 RAS 재명명 기법을 구현한 시스템을 의미하며, “**adv_Rren**”은 “Rren”에서 발생하는 3가지 문제 상황을 수정한 개선된 RAS 재명명 기법을 구현한 시스템을 의미한다.

먼저 함수 복귀 예측의 예측 정확도를 살펴보면 평균적으로 “**adv_Rren**” 기법은 “base” 기법의 실행과의 비교에서 MPKI 값이 약 1/90로 줄였으며, “CT_ras”와 비교해도 약 1/9로 줄어들었다. MPKI는 평균적으로 1000개의 명령어를 실행했을 때 발생하는 예측 실패의 횟수를 나타내며 일반적으로 예측 성능을 평가할 때 사용되는 평가요소이다. MPKI 값은 작을수록 예측 정확도가

표 2. MPKI와 IPC
Table 2. MPKI and IPC.

	base		CT_ras		Rren		**adv_Rren**	
	MPKI	IPC	MPKI	IPC	MPKI	IPC	MPKI	IPC
164.gzip	0.826	2.139	0.000	2.182	0.000	2.182	0.000	2.182
175.vpr	1.176	2.089	0.085	2.154	0.042	2.157	0.000	2.161
176.gcc	0.550	2.047	0.098	2.071	0.010	2.077	0.000	2.077
181.mcf	1.115	1.042	0.221	1.054	0.016	1.056	0.013	1.056
186.crafty	0.553	2.492	0.014	2.532	0.049	2.530	0.001	2.534
197.parser	2.595	2.118	0.162	2.258	0.163	2.257	0.062	2.265
252.eon	1.053	2.174	0.175	2.229	0.022	2.239	0.000	2.241
254.gap	0.057	1.641	0.015	1.642	0.041	1.641	0.013	1.642
255.vortex	0.123	2.488	0.005	2.496	0.002	2.496	0.000	2.496
256.bzip2	0.029	2.184	0.000	2.186	0.056	2.182	0.000	2.185
average	0.808	2.041	0.078	2.080	0.040	2.082	0.009	2.084

뛰어나다는 것을 뜻한다.

더욱이 “164.gzip”, “175.vpr”, “176.gcc”, “252.eon”, “255.vortex”, “256.bzip2” 벤치마크 프로그램들은 함수 복귀 예측의 정확도가 100%(소숫점 4째 자리에서 반올림한 MPKI 값이 0)를 보여주고 있다. 이는 개선된 RAS 재명명 기법이 효과적으로 RAS의 오염을 방지하고 있음을 보여준다. 벤치마크 프로그램들 가운데 “181.mcf”, “197.parser”, “254.gap”은 스택 오버플로가 많이 발생한다. 스택 오버플로는 하드웨어의 공간적인 제약으로 발생하는 문제이므로 스택의 크기를 늘리는 것 외에는 확실한 해결책이 없다. 이러한 하드웨어 자원의 제약에서 비롯되는 스택 오버플로 및 이로 인해 야기되는 예측 실패를 제외한다면, 본 논문에서 제안된 기법이 사실상 완벽하게 RAS의 오염을 막아주고 있다고 판단된다.

다음으로 IPC를 살펴보면 다음과 같다. “base” 기법과 비교했을 때 평균 2.07%의 IPC 향상이 있으며, “197.parser”의 경우 최대 6.95%의 성능 향상을 보인다. “CT_ras”와 비교했을 때는 평균 0.16%의 성능 향상이 있으며, “252.eon”의 경우 0.52%의 성능 향상을 보인다.

표 3은 각각의 벤치마크에서 오버플로의 발생 횟수를 보여준다. RAS 재명명 기법은 소프트 스택에 이미 팝 되었거나 잘못 푸시 된 엔트리를 유지하기 때문에 소프트 스택의 공간 사용 효율이 좋지 않다. 그렇기 때문에 표 3에서 보이는 것과 같이 오버플로의 발생 횟수가 증가한다.

표 3. 오버플로 발생 횟수
Table 3. Number of overflows.

	base	CT_ras	Rren	**adv_Rren**
164.gzip	1	1	3	494
175.vpr	0	0	1	371
176.gcc	1	1	1282	2900
181.mcf	4	4	2426	13051
186.crafty	10	10	24	19
197.parser	322	419	25433	33159
252.eon	15	15	15	11
254.gap	95	92	4987	4946
255.vortex	8	8	8	10
256.bzip2	3	3	3	10563
average	45.9	55.3	3418.2	6552.4

전술한 바와 같이 오버플로가 많이 발생하는 몇몇 벤치마크 프로그램은 이로 인해 함수 복귀 예측 정확도가 감소한다. “256.bzip2”는 예외적으로 “adv_Rren”에서 오버플로가 많이 발생했지만 함수 복귀 예측 정확도는 거의 100%를 보여주고 있다. “256.bzip2”에서는 함수의 재귀 호출이 오버플로를 발생시키기 때문이다. 재귀 호출

은 함수 복귀 주소가 전부 동일하기 때문에 쉽게 예측이 가능하다. 실제로 “256.bipz2”의 오버플로가 발생한 함수 복귀 예측의 예측 정확도는 99.9%이다.

표 4는 소프트 스택의 크기에 따른 함수 호출 오류의 발생 비율을 보여준다. 응용 프로그램별로 다소간 차이를 보이지만, 본 논문에서의 실험 결과에 의하면, 소프트 스택의 크기가 8일 때 평균 0.023%의 함수 호출 오류가 발생하며, 소프트 스택의 크기가 16 이상일 때는 함수 호출 오류는 전혀 발생하지 않는다. 이는 소프트 스택의 크기는 최소 8 이상이 되어야 한다는 것을 뜻한다. 표 4의 고딕체는 각 응용 프로그램별로 함수 호출 오류 발생 비율이 0%인 최소 소프트 스택의 크기이다.

표 4. 소프트 스택 크기에 따른 함수 호출 오류 발생 비율
Table 4. Ratio of function call error for size of soft-stack.

	2	4	8	16
164.gzip	0.016%	**0.000%**	0.000%	0.000%
175.vpr	0.001%	**0.000%**	0.000%	0.000%
176.gcc	3.442%	0.039%	**0.000%**	0.000%
181.mcf	1.290%	0.440%	0.016%	**0.000%**
186.crafty	0.760%	0.002%	**0.000%**	0.000%
197.parser	5.675%	0.644%	0.123%	**0.000%**
252.eon	13.215%	0.001%	**0.000%**	0.000%
254.gap	14.199%	0.020%	**0.000%**	0.000%
255.vortex	10.697%	0.014%	**0.000%**	0.000%
256.bzip2	0.058%	**0.000%**	0.000%	0.000%
average	6.477%	0.175%	0.023%	**0.000%**

마지막으로 표 5는 소프트 스택과 하드 스택의 크기 변화에 따른 MPKI의 변화를 보여준다. 소프트 스택과 하드 스택의 합은 64 엔트리이며, 표 5의 각 열은 소프트 스택의 크기를 의미한다. 평균적으로 소프트 스택의 크기가 24일 때 가장 작은 MPKI값을 보여주며, 각각의 벤치마크 별로 최적의 크기 비율을 고딕체로 표기하였다.

하드 스택은 최대한 커밋 된 엔트리만을 저장하려고 노력하였지만, 하드 스택으로 옮겨 가는 엔트리가 100% 커밋 된 엔트리인 것은 아니다. 함수 호출은 소프트 스택의 크기가 16이상 일 경우 100%의 커밋을 보장하지만, 함수 복귀와 무효화로 인한 복구의 경우 소프트 스택의 크기가 32일 때 각각 평균 0.038%와 0.072%의 확률로 커밋 되지 않은 엔트리가 하드 스택으로 이동한다. 소프트 스택의 크기가 커지면 이 수치는 줄어들지만 절대 0%로 만들 수는 없다. 함수 호출은 소프트 스택을 물리적으로 차례대로 사용하지만, 함수 복귀와 무효화로 인한 복구는 소프트 스택을 물리적으로 차례대로 사용하지 않기 때문이다. 반면에 소프트 스택의 크기가 커지고 하드 스택의 크기가 작아지면 오버플로의 발생 횟수가 늘어나게 된다. 소프트 스택은 RAS의 오염 상태를 복구 가능한 구조를 유지하기 위해 하드 스택에 비해서 저장 공간을 비효율적으로 사용하기 때문이다. 이러한 이유로 각각의 벤치마크 프로그램의 특성에 따라 최적의 하드 스택과 소프트 스택의 크기 비율이 달라진다.

표 5. 소프트 스택과 하드 스택의 크기 변화에 따른 MPKI
Table 5. MPKI for varying size of soft-stack and hard-stack.

	2	4	8	16	24	32	40	48	56
164.gzip	0.002	0.001	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
175.vpr	0.035	0.001	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
176.gcc	0.454	0.049	0.010	0.002	0.002	**0.000**	**0.000**	**0.000**	0.001
181.mcf	0.772	0.548	0.061	0.026	0.017	**0.013**	**0.013**	0.015	0.037
186.crafty	0.269	0.028	0.007	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
197.parser	3.176	0.467	0.180	0.080	**0.056**	0.062	0.068	0.172	0.468
252.eon	3.152	0.506	0.039	**0.000**	**0.000**	**0.000**	0.002	0.001	0.001
254.gap	2.788	0.038	0.007	**0.006**	0.008	0.013	0.027	0.072	0.180
255.vortex	3.528	0.036	0.032	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
256.bzip2	0.212	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
average	1.439	0.167	0.034	0.012	**0.008**	0.009	0.011	0.026	0.069

V. 결 론

RAS를 사용하는 함수 복귀 예측은 이론적으로 오버플로가 발생하지 않는 한도 내에서 100%의 예측 정확도를 보여야 한다. 하지만 투기적 실행을 지원하는 현대 마이크로프로세서 환경 하에서는, 잘못된 경로에 대한 실행을 무효화 할 때 RAS의 오염이 발생하여, RAS의 예측 정확도를 떨어뜨린다.

본 논문에서는 이러한 RAS의 오염을 방지하기 위하여 RAS 재명명 기법을 제안하였다. 제안된 기법은 RAS 스택을 소프트 스택과 하드 스택으로 구분한다. 이를 바탕으로 투기적 실행에 의한 데이터의 변경을 복구할 수 있는 소프트 스택에서는 투기적 실행에 의한 데이터를 관리하고, 소프트 스택의 크기 제한으로 겹쳐쓰기가 일어나는 데이터 중 이후 사용될 데이터는 하드 스택으로 옮기는 구조를 소개하였다.

모의실험 결과 본 논문에서 제안된 기법은, RAS 오염 방지 기법이 적용되지 않은 기존 시스템과 비교하여, 함수 복귀 예측 실패를 약 1/90로 감소시켰으며, 평균 2.07%의 성능 향상을 가져왔다. 또한 기존의 RAS 오염 방지 기법이 적용된 시스템과의 비교에서는 함수 복귀 예측 실패를 약 1/9로 감소 시켰다. 명령어 인출 단계와 명령어 실행 단계의 능력이 개선되어 동시에 실행되는 명령어의 숫자가 계속 증가하는 현 추세를 감안할 때, RAS 재명명 기법을 통한 성능 향상의 기여도는 추후 더욱 증가할 것이라 기대된다.

참 고 문 헌

[1] D. A. Patterson and J. L. Hennessy, "*Computer architecture: a quantitative approach*", Morgan Kaufman, 2007, 4th Edition.

[2] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines", *In Proc. 29th Int'l Symp. on Computer Architecture*, pp.25-34, 2002.

[3] G. H. Loh, "Revisiting the Performance Impact of Branch Predictor Latencies", *IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 59-69, 2006.

[4] H. Gao and H. Zhou, "PMPM: Prediction by Combining Multiple Partial Matches" , *Journal of Instruction-Level Parallelism*, Vol. 9, May, 2007.

[5] Y. Ishii, "Fused Two-Level Branch Prediction with Ahead Calculation", *Journal of Instruction-Level Parallelism*, Vol. 9, May, 2007.

[6] D. Jimenez, "Piecewise liner branch prediction" *In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec, 2003.

[7] A. Seznec, "The L-TAGE Branch Predictor", *Journal of Instruction-Level Parallelism*, Vol. 9, May, 2007.

[8] K. Aasaraai, A. Baniasadi and E. Atoofian, "Computational and storage power optimizations for the O-GEHL branch predictor", *Proceedings of the 4th international conference on Computing frontiers*, pp. 105-112, May, 2007.

[9] H. Vandierendonck and A. Seznec, "Speculative return address stack management revisited", *ACM Transactions on Architecture and Code Optimization (TACO)*, Volume 5 Issue 3, November 2008.

[10] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt and R. Cohn, "VPC prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization", Proceedings of the 34th annual international symposium on Compoter architecture, pp. 424-435, 2007.

[11] G. Lee, Y. Shi and H. Lin, "Indirect Branch Validation Unit", Microprocessors and Microsystems, Vol. 33, Issues 7-8, pp. 461-468, October-November 2009.

[12] K. Skadron, P. S. Ahuja, M. Martonosi, D. W. Clark, "Improving prediction for procedure returns with return-address-stack repair mechanisms", *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, p.259-271, November 1998.

[13] S. Jourdan, T.-H. Hsing, J. Stark, Y. N. Patt, "The Effects of Mispredicted-Path Execution on Branch Prediction Structures", *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, p.58, October 20-23, 1996.

[14] G.-Y. Chiu, H.-C. Yang, W. Y.-H. Li, Chung-Ping Chung, "Mechanism for return stack and branch history corrections under misprediction in deep pipeline design", *Computer Systems Architecture Conference*, p.1-8, 2008.

[15] J. H. Kim, J. W. Kwak, S. Yang, S. Y. Shin and C. S. Jhon, "Branch Pre-Prediction: A Method for Hiding Branch Prediction Latency", *Information an International Interdisciplinary journal*, vol. 13, No. 2, 2010.

[16] T.-Y. Yeh, “Return address predictor that uses branch instructions to track a last valid return address”, United State Patent Number 6,253,315. June 2001.
[17] V. Desmet, Y. Sazeides, C. Kourouyiannis and K. D. Bosschere, “Correct alignment of a return-address-stack after call and return mispredictions”, In Workshop on Duplicating, Deconstructing and Debunking. 25--33., 2005.
[18] D. Burger, T. M. Austin and S. Bennett, “Evaluating future micro-processors: the SimpleScalar tool set”, Tech. Report TR-1308, Univ. of Wisconin-madison Computer Science Dept., 1997.
[19] S. McFarling, “Combining branch predictors. Tech. Rep. TN-36m”, Digital Western Research Lab., June, 1993.
[20] SPEC CPU Benchmarks, http://www.specbench.org

저 자 소 개

김 주 환(정회원)
2001년 서울대학교 컴퓨터공학과 학사 졸업
2011년 서울대학교 컴퓨터공학과 박사 졸업
2011년～현재 삼성전자 반도체 사업부 SOC 개발실 책임 연구원
<주관심분야 : 고성능 컴퓨터, 병렬 구조, 모바일 SOC 설계>

장 성 태(정회원)
1986년 서울대학교 전자계산기 공학과 공학사.
1988년 서울대학교 대학원 컴퓨터공학과 석사.
1994년 서울대학교 대학원 컴퓨터공학과 박사.
1994년 3월～현재 수원대학교 정보공학대학 컴퓨터학과 교수
<주관심분야 : 다중 프로세서 시스템, 컴퓨터 구조, 병렬 처리, 캐쉬 구조, 메모리 모델>

곽 종 욱(평생회원)-교신저자
1998년 경북대학교 컴퓨터공학과 학사 졸업
2001년 서울대학교 컴퓨터공학과 석사 졸업
2006년 서울대학교 전기·컴퓨터 공학부 박사 졸업
2006년～2007년 삼성전자 SOC 연구소 책임 연구원
2007년～현재 영남대학교 컴퓨터공학과 조교수
2011년～현재 서울대학교 컴퓨터연구소 객원 연구원
<주관심분야 : 컴퓨터 구조, 저전력 내장형 시스템, 모바일 SOC 설계, 고성능 병렬 처리>

전 주 식(평생회원)
1974년 서울대학교 수공학과 학사
1976년 한국과학기술원 전산학 석사
1982년 University of Utah 전산학 박사
1982년～1984년 University of Iowa 연구원
1985년～현재 서울대학교 전기·컴퓨터공학부 교수
<주관심분야 : VLSI, CAD, 병렬 컴퓨터 구조>