(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2017-0024111
(43) 공개일자 2017년03월06일

(51) 국제특허분류(Int. Cl.)
G06F 9/455 (2006.01)
(52) CPC특허분류
G06F 9/4552 (2013.01)
(21) 출원번호 10-2017-7003198
(22) 출원일자(국제) 2015년07월23일
심사청구일자 2017년02월03일
(85) 번역문제출일자 2017년02월03일
(86) 국제출원번호 PCT/US2015/041847
(87) 국제공개번호 WO 2016/014863
국제공개일자 2016년01월28일
(30) 우선권주장
62/029,383 2014년07월25일 미국(US)
14/806,169 2015년07월22일 미국(US)

(71) 출원인
인텔 코포레이션
미합중국 캘리포니아 95054 산타클라라 미션 칼리지 블러바드 2200
(72) 발명자
압달라 모하메드
미국 캘리포니아주 95132 산 호세 선크레스트 애비뉴 3868
(74) 대리인
제일특허법인

전체 청구항 수 : 총 20 항

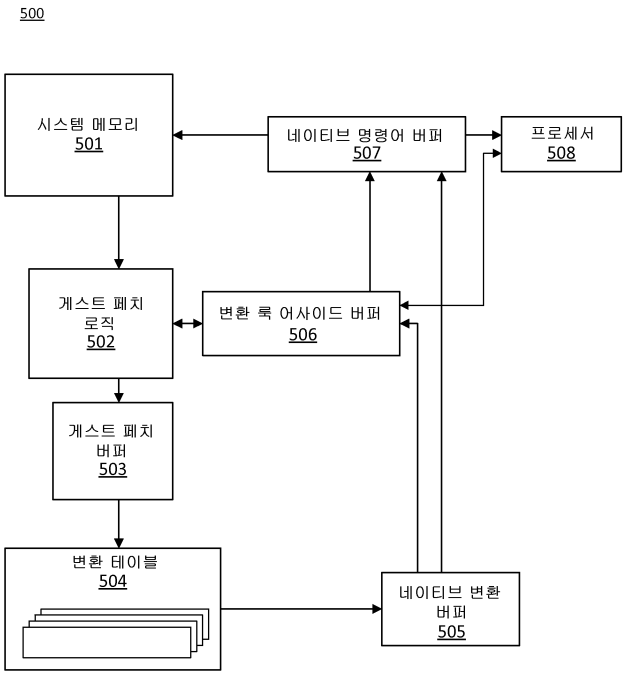
(54) 발명의 명칭 **복수의 변환 테이블을 사용한 명령어 집합의 애그노스틱 런타임 아키텍처 구현**

(57) 요 약

애그노스틱 런타임 아키텍처용 시스템이 개시된다. 시스템은 시스템 에뮬레이션/가상화 변환기와, 애플리케이션 코드 변환기와, 시스템 변환기를 포함하며, 시스템 에뮬레이션/가상화 변환기 및 애플리케이션 코드 변환기는 시스템 에뮬레이션 프로세스를 구현하고, 시스템 변환기는 게스트 이미지로부터의 코드를 실행하기 위한 시스템 변

(뒷면에 계속)

대 표 도 - 도9



환 프로세스를 구현한다. 시스템 변환기는 복수의 게스트 명령어에 액세스하기 위한 게스트 페치 로직 컴포넌트와, 게스트 페치 로직 컴포넌트 및 분기 예측 컴포넌트에 연결되어 복수의 게스트 명령어를 게스트 명령어 블록으로 조립하기 위한 게스트 페치 버퍼와, 게스트 페치 버퍼에 연결되어 게스트 명령어 블록을 대응하는 네이티브 변환 블록으로 번역하기 위한 제 1 레벨 변환 테이블 및 제 2 레벨 변환 테이블을 포함하는 복수의 변환 테이블을 포함한다. 시스템은 변환 테이블에 연결되어 대응하는 네이티브 변환 블록을 저장하기 위한 네이티브 캐시와, 네이티브 캐시에 연결되어 게스트 명령어 블록에서 대응하는 네이티브 변환 블록으로의 맵핑을 저장하기 위한 변환 룩 어사이드 버퍼를 더 포함한다. 게스트 명령어의 후속 요청 시, 변환 룩 어사이드 버퍼는 히트가 발생하였는지를 판정하기 위해 인덱스되며, 맵핑은 게스트 명령어가 네이티브 캐시 내에서 대응하는 변환된 네이티브 명령어를 갖고 있음을 표시하고, 히트에 응답하여 번역된 네이티브 명령어를 실행을 위해 포워딩한다.

명 세 서

청구범위

청구항 1

애그노스틱 런타임 아키텍처(agnostic runtime architecture)용 시스템으로서,

시스템 에뮬레이션/가상화 변환기와,

애플리케이션 코드 변환기와,

시스템 변환기 - 상기 시스템 에뮬레이션/가상화 변환기 및 상기 애플리케이션 코드 변환기는 시스템 에뮬레이션 프로세스를 구현하고, 상기 시스템 변환기는 게스트 이미지로부터의 코드를 실행하기 위한 시스템 변환 프로세스를 구현함 - 를 포함하며,

상기 시스템 변환기는,

복수의 게스트 명령어에 액세스하기 위한 게스트 페치 로직 컴포넌트와,

상기 게스트 페치 로직 컴포넌트 및 분기 예측 컴포넌트에 연결되어 상기 복수의 게스트 명령어를 게스트 명령어 블록으로 조립하기 위한 게스트 페치 버퍼와,

상기 게스트 페치 버퍼에 연결되어, 상기 게스트 명령어 블록을 대응하는 네이티브 변환 블록으로 번역하기 위한 제 1 레벨 변환 테이블 및 제 2 레벨 변환 테이블을 포함하는 복수의 변환 테이블과,

상기 변환 테이블에 연결되어 상기 대응하는 네이티브 변환 블록을 저장하기 위한 네이티브 캐시와,

상기 네이티브 캐시에 연결되어 상기 게스트 명령어 블록에서 대응하는 네이티브 변환 블록으로의 맵핑을 저장하기 위한 변환 룩 어사이드 버퍼를 더 포함 -

게스트 명령어의 후속 요청 시, 상기 변환 룩 어사이드 버퍼는 히트(a hit)가 발생하였는지를 판정하기 위해 인덱스되며, 상기 맵핑은 상기 게스트 명령어가 상기 네이티브 캐시 내에서 대응하는 변환된 네이티브 명령어를 갖고 있음을 표시함 - 하며,

상기 히트에 응답하여 상기 변환 룩 어사이드 버퍼는 상기 번역된 네이티브 명령어를 실행을 위해 포워딩하는

애그노스틱 런타임 아키텍처용 시스템.

청구항 2

제 1 항에 있어서,

상기 제 1 레벨 변환 테이블 및 상기 제 2 레벨 변환 테이블은 시스템 메모리에 저장된 변환 테이블 버퍼와 일관성 있게 유지되는 고속의 저 지연 캐시로서 구현되는

애그노스틱 런타임 아키텍처용 시스템.

청구항 3

제 1 항에 있어서,

상기 제 1 레벨 변환 테이블 및 상기 제 2 레벨 변환 테이블은 네이티브 명령어를 게스트 명령어로 대체하기 위한 대체 필드, 및 상기 제 1 레벨 변환 테이블과 제 2 레벨 변환 테이블을 제어하기 위한 제어 필드를 포함하는

애그노스틱 런타임 아키텍처용 시스템.

청구항 4

제 1 항에 있어서,

상기 제 1 레벨 변환 테이블은 상기 게스트 명령어에 대해 제 1 레벨 번역을 수행하도록 사용되는

애그노스틱 런타임 아키텍처용 시스템.

청구항 5

제 4 항에 있어서,

상기 제 1 레벨 변환 테이블은 복수의 게스트 명령어를 인식하도록 사용되는 해시를 포함하는

애그노스틱 런타임 아키텍처용 시스템.

청구항 6

제 5 항에 있어서,

상기 복수의 게스트 명령어는 적어도 하나의 프리픽스 게스트 명령어 및 상기 프리픽스 게스트 명령어와 연관된 복수의 오피코드(opcode)를 포함하는

애그노스틱 런타임 아키텍처용 시스템.

청구항 7

프로세서를 위한 명령어를 번역하기 위한 시스템으로서,

시스템 에뮬레이션/가상화 변환기와,

애플리케이션 코드 변환기와,

시스템 변환기 - 상기 시스템 에뮬레이션/가상화 변환기 및 상기 애플리케이션 코드 변환기는 시스템 에뮬레이션 프로세스를 구현하고, 상기 시스템 변환기는 게스트 이미지로부터의 코드를 실행하기 위한 시스템 변환 프로세스를 구현함 - 를 포함하며,

상기 시스템 변환기는,

복수의 게스트 명령어에 액세스하기 위한 게스트 페치 로직 컴포넌트와,

상기 게스트 페치 로직 컴포넌트 및 분기 예측 컴포넌트에 연결되어 상기 복수의 게스트 명령어를 게스트 명령어 블록으로 조립하기 위한 게스트 페치 버퍼와,

상기 게스트 페치 버퍼에 연결되어, 상기 게스트 명령어 블록을 대응하는 네이티브 변환 블록으로 번역하기 위한 제 1 레벨 변환 테이블 및 제 2 레벨 변환 테이블을 포함하는 복수의 변환 테이블과,

상기 변환 테이블에 연결되어 상기 대응하는 네이티브 변환 블록을 저장하기 위한 네이티브 캐시와,

상기 네이티브 캐시에 연결되어 상기 게스트 명령어 블록에서 대응하는 네이티브 변환 블록으로의 맵핑을 저장하기 위한 변환 룩 어사이드 버퍼를 더 포함 -

게스트 명령어의 후속 요청 시, 상기 변환 룩 어사이드 버퍼는 히트가 발생하였는지를 판정하기 위해 인덱스되며, 상기 맵핑은 상기 게스트 명령어가 상기 네이티브 캐시 내에서 대응하는 변환된 네이티브 명령어를 갖고 있음을 표시함 - 하며,

상기 히트에 응답하여 상기 변환 룩 어사이드 버퍼는 상기 번역된 네이티브 명령어를 실행을 위해 포워딩하는

프로세서를 위한 명령어를 번역하기 위한 시스템.

청구항 8

제 7 항에 있어서,

상기 제 1 레벨 변환 테이블 및 상기 제 2 레벨 변환 테이블은 시스템 메모리에 저장된 변환 테이블 버퍼와 일관성 있게 유지되는 고속의 저 지연 캐시로서 구현되는

프로세서를 위한 명령어를 번역하기 위한 시스템.

청구항 9

제 7 항에 있어서,

상기 제 1 레벨 변환 테이블 및 상기 제 2 레벨 변환 테이블은 네이티브 명령어를 게스트 명령어로 대체하기 위한 대체 필드, 및 상기 제 1 레벨 변환 테이블과 제 2 레벨 변환 테이블을 제어하기 위한 제어 필드를 포함하는

프로세서를 위한 명령어를 번역하기 위한 시스템.

청구항 10

제 7 항에 있어서,

상기 제 1 레벨 변환 테이블은 상기 게스트 명령어에 대해 제 1 레벨 번역을 수행하도록 사용되는

프로세서를 위한 명령어를 번역하기 위한 시스템.

청구항 11

제 10 항에 있어서,

상기 제 1 레벨 변환 테이블은 복수의 게스트 명령어를 인식하도록 사용되는 해시를 포함하는

프로세서를 위한 명령어를 번역하기 위한 시스템.

청구항 12

제 11 항에 있어서,

상기 복수의 게스트 명령어는 적어도 하나의 프리픽스 게스트 명령어 및 상기 프리픽스 게스트 명령어와 연관된 복수의 오피코드를 포함하는

프로세서를 위한 명령어를 번역하기 위한 시스템.

청구항 13

명령어를 번역하는 방법을 구현하는 마이크로프로세서로서,

시스템 에뮬레이션/가상화 변환기와,

애플리케이션 코드 변환기와,

시스템 변환기 - 상기 시스템 에뮬레이션/가상화 변환기 및 상기 애플리케이션 코드 변환기는 시스템 에뮬레이션 프로세스를 구현하고, 상기 시스템 변환기는 게스트 이미지로부터의 코드를 실행하기 위한 시스템 변환 프로

세스를 구현함 - 를 포함하며,

상기 시스템 변환기는,

복수의 게스트 명령어에 액세스하기 위한 게스트 페치 로직 컴포넌트와,

상기 게스트 페치 로직 컴포넌트 및 분기 예측 컴포넌트에 연결되어 상기 복수의 게스트 명령어를 게스트 명령어 블록으로 조립하기 위한 게스트 페치 버퍼와,

상기 게스트 페치 버퍼에 연결되어, 상기 게스트 명령어 블록을 대응하는 네이티브 변환 블록으로 번역하기 위한 제 1 레벨 변환 테이블 및 제 2 레벨 변환 테이블을 포함하는 복수의 변환 테이블과,

상기 변환 테이블에 연결되어 상기 대응하는 네이티브 변환 블록을 저장하기 위한 네이티브 캐시와,

상기 네이티브 캐시에 연결되어 상기 게스트 명령어 블록에서 대응하는 네이티브 변환 블록으로의 맵핑을 저장하기 위한 변환 룩 어사이드 버퍼를 더 포함 -

게스트 명령어의 후속 요청 시, 상기 변환 룩 어사이드 버퍼는 히트가 발생하였는지를 판정하기 위해 인덱스되며, 상기 맵핑은 상기 게스트 명령어가 상기 네이티브 캐시 내에서 대응하는 변환된 네이티브 명령어를 갖고 있음을 표시함 - 하며,

상기 히트에 응답하여 상기 변환 룩 어사이드 버퍼는 상기 번역된 네이티브 명령어를 실행을 위해 포워딩하는

명령어를 번역하는 방법을 구현하는 마이크로프로세서.

청구항 14

제 13 항에 있어서,

상기 제 1 레벨 변환 테이블 및 상기 제 2 레벨 변환 테이블은 시스템 메모리에 저장된 변환 테이블 버퍼와 일관성 있게 유지되는 고속의 저 지연 캐시로서 구현되는

명령어를 번역하는 방법을 구현하는 마이크로프로세서.

청구항 15

제 13 항에 있어서,

상기 제 1 레벨 변환 테이블 및 상기 제 2 레벨 변환 테이블은 네이티브 명령어를 게스트 명령어로 대체하기 위한 대체 필드, 및 상기 제 1 레벨 변환 테이블과 제 2 레벨 변환 테이블을 제어하기 위한 제어 필드를 포함하는

명령어를 번역하는 방법을 구현하는 마이크로프로세서.

청구항 16

제 13 항에 있어서,

상기 제 1 레벨 변환 테이블은 상기 게스트 명령어에 대해 제 1 레벨 번역을 수행하도록 사용되는

명령어를 번역하는 방법을 구현하는 마이크로프로세서.

청구항 17

제 16 항에 있어서,

상기 제 1 레벨 변환 테이블은 복수의 게스트 명령어를 인식하도록 사용되는 해시를 포함하는

명령어를 번역하는 방법을 구현하는 마이크로프로세서.

청구항 18

제 17 항에 있어서,

상기 복수의 게스트 명령어는 적어도 하나의 프리픽스 게스트 명령어 및 상기 프리픽스 게스트 명령어와 연관된 복수의 오피코드를 포함하는

명령어를 번역하는 방법을 구현하는 마이크로프로세서.

청구항 19

제 13 항에 있어서,

상기 제 1 레벨 변환 테이블 및 상기 제 2 레벨 변환 테이블은 복수의 마스크 및 복수의 태그를 포함하며, 상기 태그는 패턴 매치를 결정하고 마스크는 패턴의 비 관련 비트(non-relevant bit)를 숨기는

명령어를 번역하는 방법을 구현하는 마이크로프로세서.

청구항 20

제 19 항에 있어서,

상기 복수의 마스크 및 상기 복수의 태그는 구성 가능한 디코딩을 가능할 수 있게 하도록 소프트웨어 적재 가능한

명령어를 번역하는 방법을 구현하는 마이크로프로세서.

발명의 설명

기 술 분 야

[0001] 본 출원은 Mohammad A. Abdallah에 의해 2014년 7월 25일자로 출원된 "A RUNTIME ARCHITECTURE FOR EFFICIENTLY OPTIMIZING AND EXECUTING GUEST CODE AND CONVERTING TO NATIVE CODE(게스트 코드를 효과적으로 최적화하여 실행하고 네이티브 코드로 변환하기 위한 런타임 아키텍처)"라는 명칭의 공동으로 양도된 계류 중인 미국 가출원 제62/029383호의 우선권을 주장하며, 이 가출원은 그 전체가 본 출원에서 참조문헌으로 인용된다.

[0002] 기술 분야

[0003] 본 발명은 일반적으로 디지털 컴퓨터 시스템에 관한 것으로, 특히, 명령어 시퀀스를 포함하는 명령어를 선택하기 위한 시스템 및 방법에 관한 것이다.

배 경 기 술

[0004] 프로세서는 종속적이거나 아니면 완전히 독립적인 여러 작업을 처리해야 한다. 그러한 프로세서의 내부 상태는 통상 프로그램 실행의 각 특정 순간에 상이한 값을 보유할 수 있는 레지스터로 구성된다. 프로그램 실행의 각 순간에, 내부 상태의 이미지는 프로세서의 아키텍처 상태라고 부른다.

[0005] 다른 함수 (예를 들면, 다른 스레드, 프로세스 또는 프로그램)를 실행하기 위해 코드 실행이 전환될 때, 머신/프로세서의 상태는, 새로운 함수가 내부 레지스터를 이용하여 그 함수의 새로운 상태를 구축할 수 있도록, 저장되어야 한다. 새로운 함수가 종료되면 새로운 함수의 상태는 파기될 수 있으며, 이전 컨텍스트(context)의 상태가 복구되어 다시 실행될 것이다. 이러한 스위치 프로세스는 컨텍스트 스위치(context switch)라 불리며, 특히 복수의 레지스터(예를 들면, 64, 128, 256) 및/또는 비순차적(out of order) 실행을 채용하는 특히 현대 아키텍처에서는 통상적으로 수십 또는 수백 사이클을 포함한다.

[0006] 스레드 인식(thread-aware) 하드웨어 아키텍처에서, 하드웨어는 제한된 수의 하드웨어 지원 스레드에 대해 여러

컨텍스트 상태를 지원하는 것이 일반적이다. 이 경우, 하드웨어는 지원되는 각 스레드에 대한 모든 아키텍처 상태 요소를 복제한다. 이렇게 하면 새로운 스레드를 실행할 때 컨텍스트 스위치가 필요하지 않다. 그러나 이것은 여전히 많은 단점, 말하자면, 하드웨어에서 지원되는 각 추가 스레드에 대한 모든 아키텍처 상태 요소(레지스터)를 복제하기 위한 공간, 전력 및 복잡성이 있다. 또한, 만일 소프트웨어 스레드의 수가 명시적으로 지원되는 하드웨어 스레드의 수를 초과하면, 컨텍스트 스위치는 계속 수행되어야 한다.

[0007] 많은 수의 스레드를 요구하는 미세하게 작은 세부 단위에 병렬 처리가 필요하기 때문에 이러한 일이 흔해진다. 중복 컨텍스트 상태 하드웨어 저장소를 갖는 하드웨어 스레드 인식 아키텍처는 비-스레드방식의(non-threaded) 소프트웨어 코드를 지원하지 않으며 단지 스레드 처리된 소프트웨어에 대해 컨텍스트 스위치의 수만을 줄인다. 그러나 이러한 스레드는 통상 큰 그레인(coarse-grain) 병렬 처리를 위해 구축되며, 시작 및 동기화에 대한 소프트웨어 오버헤드를 과중하게 만드는 결과를 초래하여, 효율적인 스레드 시작/자동 생성 없이, 함수 호출 및 루프 병렬 실행과 같은 미세 그레인(fine grain) 병렬처리를 하지 못한다. 이러한 오버헤드로 인해 비명시적인/용이하게 병렬화된/스레드 처리된 소프트웨어 코드에 대해 최신의 컴파일러 또는 사용자 병렬화 기술을 사용하여 그와 같은 코드를 자동 병렬화하는 것이 더불어 어려워진다.

발명의 내용

과제의 해결 수단

[0008] 일 실시예에서, 본 발명은 애그노스틱 런타임 아키텍처용 시스템으로서 구현된다. 시스템은 시스템 에뮬레이션/가상화 변환기와, 애플리케이션 코드 변환기와, 시스템 변환기를 포함하며, 시스템 에뮬레이션/가상화 변환기 및 애플리케이션 코드 변환기는 시스템 에뮬레이션 프로세스를 구현하고, 시스템 변환기는 게스트 이미지로부터의 코드를 실행하기 위한 시스템 변환 프로세스를 구현한다. 시스템 변환기는 복수의 게스트 명령어에 액세스하기 위한 게스트 페치 로직 컴포넌트와, 게스트 페치 로직 컴포넌트 및 분기 예측 컴포넌트에 연결되어 복수의 게스트 명령어를 게스트 명령어 블록으로 조립하기 위한 게스트 페치 버퍼와, 게스트 페치 버퍼에 연결되어 게스트 명령어 블록을 대응하는 네이티브 변환 블록으로 번역하기 위한 제 1 레벨 변환 테이블 및 제 2 레벨 변환 테이블을 포함하는 복수의 변환 테이블을 포함한다.

[0009] 시스템은 변환 테이블에 연결되어 대응하는 네이티브 변환 블록을 저장하기 위한 네이티브 캐시와, 네이티브 캐시에 연결되어 게스트 명령어 블록에서 대응하는 네이티브 변환 블록으로의 맵핑을 저장하기 위한 변환 룩 어사이드 버퍼를 더 포함한다. 게스트 명령어의 후속 요청 시, 변환 룩 어사이드 버퍼는 히트가 발생하였는지를 판정하기 위해 인덱스되며, 맵핑은 게스트 명령어가 네이티브 캐시 내에서 대응하는 변환된 네이티브 명령어를 갖고 있음을 표시하고, 히트에 응답하여 번역된 네이티브 명령어를 실행을 위해 포워딩한다.

[0010] 전술한 내용은 개요이며, 필요에 따라, 세부 사항의 단순화, 일반화, 생략을 포함하며, 따라서, 본 기술에 통상의 지식을 가진 자라면 개요는 단지 예시적일 뿐이며 어떤 식으로든 제한하려는 것이 아니라는 것을 인식할 것이다. 본 발명의 다른 측면, 발명의 특징 및 장점은 특허 청구 범위에 의해 정의된 것처럼 아래에서 설명되는 비제한적인 상세한 설명에서 자명해질 것이다.

도면의 간단한 설명

[0011] 본 발명은 첨부 도면의 도면에서 예의 형태로 예시되는 것이지, 제한하는 형태로 예시되지 않으며, 첨부 도면에서 동일한 참조 부호는 유사한 구성요소를 지칭한다.

도 1은 본 발명의 일 실시예에 따른 아키텍처 애그노스틱 런타임 시스템의 개요도를 도시한다.

도 2는 본 발명의 일 실시예에 따른 하드웨어 가속된 변환/JIT 계층을 묘사하는 다이어그램을 도시한다.

도 3은 본 발명의 일 실시예에 따른 하드웨어 가속된 런타임 변환/JIT 계층의 더 상세한 다이어그램을 도시한다.

도 4는 본 발명의 일 실시예에 따른 시스템 에뮬레이션 및 시스템 변환을 구현하기 위한 컴포넌트를 묘사하는 다이어그램을 도시한다.

도 5는 본 발명의 일 실시예에 따른 게스트 플래그 아키텍처 에뮬레이션(guest flag architecture emulation)을 묘사하는 다이어그램을 도시한다.

도 6은 본 발명의 일 실시예에 따른 통합된 레지스터 파일(unified register file)의 다이어그램을 도시한다.

도 7은 본 발명의 일 실시예에 따른 추측 아키텍처 상태(speculative architectural state) 및 비영속성 아키텍처 상태(transient architectural state)를 지원하는 통합된 새도우 레지스터 파일 및 파이프라인 아키텍처(unified shadow register file and pipeline architecture)(1300)를 도시한다.

도 8은 본 발명의 일 실시예에 따른 런 어헤드 배치/변환 프로세스(run ahead batch/conversion process)를 묘사하는 다이어그램을 도시한다.

도 9는 본 발명의 일 실시예에 따라 게스트 명령어 블록 및 그의 대응하는 네이티브 변환 블록이 캐시 내에 저장되는 방식을 예시하는 예시적인 하드웨어 가속된 변환 시스템의 다이어그램을 도시한다.

도 10은 본 발명의 일 실시예에 따른 하드웨어 가속된 변환 시스템의 더 상세한 예를 도시한다.

도 11은 본 발명의 일 실시예에 따른 이중 스코프 사용(dual scope usage)을 포함하는, 제 2 사용 모델의 다이어그램을 도시한다.

도 12는 본 발명의 일 실시예에 따라 비영속성 컨텍스트(transient context)로부터 복귀할 때 이전 컨텍스트를 저장 및 복원할 필요가 없는 비영속성 컨텍스트 스위칭을 포함하는 제 3 사용 모델의 다이어그램을 도시한다.

도 13은 본 발명의 일 실시예에 따라 후속 코드에 대한 번역이 필요하기 때문에 명령어 시퀀스에서 예외가 있는 경우를 묘사하는 다이어그램을 도시한다.

도 14는 본 발명의 일 실시예에 따라 비영속성 컨텍스트로부터 복귀할 때 이전 컨텍스트를 저장 및 복원할 필요 없는 비영속성 컨텍스트 스위칭을 포함하는, 제 4 사용 모델의 다이어그램을 도시한다.

도 15는 본 발명의 일 실시예에 따라 명령어를 분기(branch)의 앞으로 최적화하게 스케줄링하는 것을 예시하는 다이어그램을 도시한다.

도 16은 본 발명의 일 실시예에 따라 로드(load)를 스토어(store)의 앞으로 최적화게 스케줄링하는 것을 예시하는 다이어그램을 도시한다.

도 17은 본 발명의 일 실시예에 따른 스토어 필터링 알고리즘의 다이어그램을 도시한다.

도 18은 본 발명의 일 실시예에 따라 순차적으로 메모리로부터 읽는 로드를 구성하는 메모리 일관성 모델(memory consistency model)에서 로드가 비순차적(out of order)인 세마포어(semaphore) 구현을 도시한다.

도 19는 본 발명의 일 실시예에 따라 JIT 최적화를 통한 재정렬 프로세스의 다이어그램을 도시한다.

도 20은 본 발명의 일 실시예에 따라 JIT 최적화를 통한 재정렬 프로세스의 다이어그램을 도시한다.

도 21은 본 발명의 일 실시예에 따라 JIT 최적화를 통한 재정렬 프로세스의 다이어그램을 도시한다.

도 22는 본 발명의 일 실시예에 따라 JIT 최적화를 통한 스토어 앞에 재정렬된 로드를 예시하는 다이어그램을 도시한다.

도 23은 본 발명의 일 실시예에 따라 로드 및 스토어 명령어의 분할(load and store instruction splitting)에 관한 제 1 다이어그램을 도시한다.

도 24는 본 발명의 일 실시예에 따라 메모리 내에 저장된 코드 캐시 및 게스트 명령어 대 네이티브 명령어의 맵핑과 협력하여 CLB가 기능을 수행하는 방식을 예시하는 예시적인 흐름도를 도시한다.

도 25는 본 발명의 일 실시예에 따른 런 어헤드 런 타임 게스트 명령어 변환/디코딩 프로세스(run ahead run time guest instruction conversion/decoding process)의 다이어그램을 도시한다.

도 26은 본 발명의 일 실시예에 따른 게스트 명령어 시퀀스를 갖는 변환 테이블 및 네이티브 명령어 맵핑을 갖는 네이티브 맵핑 테이블을 묘사하는 다이어그램을 도시한다.

발명을 실시하기 위한 구체적인 내용

[0012] 본 발명은 하나의 실시예와 관련하여 기술되지만, 본 발명은 본 명세서에서 언급된 특정한 형태로 제한되게 하려는 의도는 아니다. 이와 반대로, 본 발명은 첨부의 특허청부범위에 의해 정의된 바와 같이 본 발명의 범위 내에 합당하게 포함될 수 있는 것과 같은 대안 예, 수정 예, 및 균등물을 망라하고자 한다.

[0013] 다음 상세한 설명에는 특정 방법 순서, 구조, 요소 및 연결과 같은 여러 특정 세부 사항이 언급되어 있다. 그러

나 이러한 특정 세부 사항 및 다른 특정 세부 사항은 본 발명의 실시예를 실시하기 위해 이용될 필요가 없다는 것을 이해해야 한다. 다른 상황에서는 널리 공지된 구조, 요소 또는 연결이 생략되거나, 본 설명을 불필요하게 모호하게 하는 것을 피하기 위해 특별히 상세히 설명되지 않는다.

[0014] 명세서 내에서 “일 실시예” 또는 “실시예”라고 말하는 것은 실시예와 관련하여 기술된 특정한 특징, 구조 또는 특성이 본 발명의 적어도 하나의 실시예에 포함된다는 것을 의미하는 것으로 의도된다. 명세서 내의 여러 곳에서 "일 실시예에서"라는 문구가 출현한다 하여 모두 동일한 실시예를 말하는 것은 아니며, 또한 다른 실시예의 상호 배타적으로 별개 또는 대안의 실시예라는 것도 아니다. 또한, 일부 실시예에 의해 나타날 수 있지만 다른 실시예에 의해 나타나지 않는 다양한 특징이 기술된다. 마찬가지로, 일부 실시예의 요구 조건이 될 수 있지만 다른 실시예에서는 그렇지 않은 다양한 요구 사항이 기술된다.

[0015] 다음에 설명되는 상세한 설명의 일부는 컴퓨터 메모리 내의 데이터 비트에 대한 절차, 단계, 로직 블록, 처리, 및 연산의 다른 부호적 표면의 측면에서 제공된다. 이러한 설명 및 표현은 데이터 처리 기술에서 통상의 지식을 가진 자가 자기들의 작업의 실체를 본 기술에서 통상의 지식을 가진 다른 자에게 가장 효과적으로 전달하기 위해 사용되는 수단이다. 본 명세서에서 절차, 컴퓨터 실행된 단계, 로직 블록, 프로세스 등은 일반적으로 원하는 결과에 이르는 자체 일관성 있는 단계 또는 명령어의 시퀀스라고 이해된다. 이러한 단계는 물리적인 양의 물리적 조작을 요하는 단계이다. 통상적으로, 필수적인 것은 아니지만, 물리적인 양은 컴퓨터 판독 가능한 저장 매체의 전기 또는 자기 신호의 형태를 취하며, 저장, 전달, 결합, 비교될 수 있고, 그렇지 않으면, 컴퓨터 시스템에서 조작될 수 있다. 이러한 신호를 비트, 값, 요소, 심볼, 문자, 용어, 또는 숫자 등으로서 지칭하는 것이 주로 공통으로 사용한다는 이유 때문에 편리하다고 입증되었다.

[0016] 그러나 이러한 용어 및 유사한 용어는 모두 적당한 물리적인 양과 연관되고, 이들 양에 적용되는 단지 편리한 표시라는 것을 명심해야 한다. 다음과 같은 설명으로부터 명백해지는 것처럼 달리 구체적으로 명시되지 않는 한, 본 발명의 전체에서, "처리" 또는 "액세스" 또는 "기록" 또는 "저장" 또는 "복제" 등과 같은 용어를 사용하는 설명은 컴퓨터 시스템의 레지스터와 메모리 및 다른 컴퓨터 판독 가능한 매체 내에서 물리적인 (전자) 양으로서 표현되는 데이터를 컴퓨터 시스템 메모리 또는 레지스터 또는 다른 그러한 정보 저장, 전송 또는 디스플레이 디바이스 내의 물리적인 양으로서 유사하게 표현되는 다른 데이터로 조정 및 변환하는 컴퓨터 시스템, 또는 유사한 전자 계산 디바이스의 동작 및 프로세스를 말하는 것으로 인식된다.

[0017] 본 발명의 실시예는 범용의(universal) 애그노스틱 런타임 시스템(universal agnostic runtime system)의 구현에 관련된다. 본 명세서에서 사용되는 바와 같이, 본 발명의 실시예는 "VISC ISA 애그노스틱 런타임 아키텍처"라고 지칭되기도 한다. 다음의 상세한 설명의 도 1 내지 도 30은 범용의 애그노스틱 런타임 시스템을 구현하는데 사용되는 메커니즘 프로세스 및 시스템을 예시한다.

[0018] 본 발명의 실시예는 소프트웨어 산업에서의 추세, 즉, 새로운 시스템 소프트웨어가 런타임 컴파일, 최적화, 및 실행을 향해 점차적으로 지향되는 추세를 이용하는 것에 관련된다. 전통적인 이전의 소프트웨어 시스템은 정적인 컴파일에 적합하다.

[0019] 유리하게 본 발명의 실시예는 런타임 조작을 지향하는 경향이 있는 새로운 시스템 소프트웨어에 관련된다. 예를 들면, 처음에는 자바(Java) 가상 머신 런타임 구현이 인기가 있었다. 그러나 이러한 구현은 네이티브 실행보다 네 배 내지 다섯 배 더 느려지는 단점이 있다. 최근에, 구현은 (예를 들면, 2 내지 4배 더 느린) 자바 가상 머신 구현 및 네이티브 코드 캡슐화를 지향하고 있다. 더 최근에 구현은 (네이티브보다 두 배 더 느린) 크롬(Chrome) 및 저 레벨의 가상 머신 런타임 구현을 지향하고 있다.

[0020] 본 발명의 실시예는 확장적인 런타임 지원을 하고 이를 사용할 아키텍처를 구현할 것이다. 본 발명의 실시예는 (예를 들면, 런 타임 게스트 코드(run time guest code)를 포함하는) 게스트 코드를 효율적으로 실행하는 능력을 가질 것이다. 본 발명의 실시예는 게스트/런타임 명령어를 효율적으로 네이티브 명령어로 변환할 수 있다. 본 발명의 실시예는 변환된 게스트/런타임 코드 대 네이티브 코드를 효율적으로 맵핑할 수 있다. 또한, 본 발명의 실시예는 런타임 시 게스트 코드 또는 네이티브 코드를 효율적으로 최적화할 수 있다.

[0021] 이러한 능력은 본 발명의 실시예를 아키텍처 애그노스틱 런타임 시스템의 시대에 매우 적합하게 만들어 줄 수 있다. 본 발명의 실시예는 레거시 애플리케이션 코드를 실행하는 능력으로 완전히 이식 가능할 것이며, 그러한 코드는 빠르게 또는 다른 아키텍처보다 두 배 빠르게 실행하도록 최적화될 수 있다.

[0022] 도 1은 본 발명의 실시예에 따른 아키텍처 애그노스틱 런타임 시스템의 개요도를 도시한다. 도 1은 가상 머신 런타임 JIT(예를 들면, 저스트-인-타임 컴파일러(just-in-time compiler))를 도시한다. 가상 머신 런타임 JIT는

도시된 바와 같이 자바 유사 바이트 코드, 저 레벨의 내부 표현 코드, 및 가상 머신 JIT를 포함한다. 가상 머신 JIT는 저 레벨의 내부 표현 코드 및 자바와 유사한 바이트 코드(Java like byte code)를 처리한다. 가상 머신 JIT의 출력은 도시된 바와 같이 ISA 특정 코드이다.

[0023] 자바 코드는 머신에 독립적이다. 프로그래머는 하나의 프로그램을 작성할 수 있으며 이 프로그램을 다른 많은 머신에서 실행하여야 한다. 자바 가상 머신은 ISA 특정 머신이며, 각 머신 아키텍처는 자체의 머신 특정 가상 머신을 가지고 있다. 가상 머신의 출력은 런타임 시 동적으로 생성된 ISA 특정 코드이다.

[0024] 도 1은 또한 프로세서에 밀접하게 연결된 하드웨어 가속된 변환/JIT 계층을 도시한다. 런타임 JIT/변환 계층은 프로세서가 가상 머신에 의해 처리될 필요가 없는 전 처리된 자바 바이트 코드를 사용할 수 있게 하며, 그렇게 함으로써 코드 수행이 상당히 빨라진다. 런타임 JIT/변환 계층은 또한 프로세서가 가상 머신/JIT에 의해 처리될 필요가 없는 (예를 들면, 가상 머신 런타임 JIT 내에서 보이는) 자바 바이트 코드의 저 레벨의 내부 표현을 사용할 수 있게 한다.

[0025] 도 1은 또한 정적의 이진 실행 코드를 생성하는 오프라인 컴파일러(예를 들면, x86, ARM 등)에 의해 처리되는 C++ 코드(예를 들면, 또는 기타 등등)를 도시한다. C++은 머신 독립적 프로그래밍 언어이다. 컴파일러는 머신에 특정(예를 들면, x86, ARM 등)한 컴파일러이다. 프로그램은 머신 특정 컴파일러를 사용하여 오프라인에서 컴파일되며, 그럼으로써 머신에 특정한 정적의 이진 코드를 생성한다.

[0026] 도 1은 ISA 특정 코드가 어떻게 통상의 프로세서상의 통상의 운영 시스템에 의해 실행되는지를 도시하며, 한편으로는 (예를 들면, 저 레벨의 내부 표현으로부터 유래되는) 이식 가능한 코드, (예를 들면, 가상 머신 런타임 JIT로부터 유래되는) 전 처리된 자바 유사 바이트 코드, 및 (예를 들면, 컴파일러로부터 유래되는) 정적의 이진 실행 코드가 모두 어떻게 하드웨어 가속된 변환 JIT 계층 및 프로세서를 통해 처리될 수 있는지를 제대로 보여주고 있다.

[0027] 하드웨어 가속된 변환/JIT 계층은 본 발명의 실시예의 장점을 성취하기 위한 주요 메커니즘이라는 것을 알아야 한다. 다음의 도면은 하드웨어 가속된 변환/JIT 계층의 동작 방식을 예시한다.

[0028] 도 2는 본 발명의 일 실시예 따른 하드웨어 가속된 변환/JIT 계층을 묘사하는 다이어그램을 도시한다. 도 2의 다이어그램은 가상 머신/고 레벨의 런타임/로드 타임(load time) JIT가 어떻게 가상 머신 고 레벨 명령어 표현, 저 레벨 가상 머신 명령어 표현, 및 게스트 코드 애플리케이션 명령어를 생성하는지를 도시한다. 이들 모두는 런타임/로드 타임 게스트/가상 머신 명령어 표현 대 네이티브 명령어 표현의 맵핑을 위한 프로세스에 제공된다. 다음 차례로 이 프로세스는 도시된 바와 같이 하드웨어 가속된 변환/JIT 계층으로 넘어가며, 이 계층에서 런타임 네이티브 명령어 표현 대 명령어 어셈블리 컴포넌트에 의해 처리된 다음 코드 캐시 할당 및 메타데이터 생성 컴포넌트의 하드웨어/소프트웨어에 의한 동적 시퀀스 기반의 블록 구성/맵핑으로 전달된다. 도 2의 다이어그램에서, 하드웨어 가속된 변환/JIT 계층은 동적으로 변환된 시퀀스를 저장하는 시퀀스 캐시를 가진 프로세서에 연결된 것으로 도시된다. 도 2의 다이어그램은 또한 네이티브 코드가 런타임 네이티브 명령어 시퀀스 형성 컴포넌트에 의해 어떻게 직접 처리될 수 있는지를 보여주는데, 이 컴포넌트는 결과로 생긴 출력을 코드 캐시 할당 및 메타데이터 생성 컴포넌트의 하드웨어/소프트웨어에 의한 동적 시퀀스 기반의 블록 구성/맵핑으로 전달한다.

[0029] 도 3은 본 발명의 일 실시예에 따른 하드웨어 가속된 런타임 변환/JIT 계층의 더 상세한 다이어그램을 도시한다. 도 3은 하드웨어 가속된 런타임 변환/JIT 계층이 어떻게 시스템 에뮬레이션 및 시스템 변환을 용이하게 하는 하드웨어 컴포넌트를 포함하는지를 보여준다. 분산된 플래그 지원(decentralized flag support), 및 CLB/CLBV 등과 같은 이들 컴포넌트는 시스템 에뮬레이션과 시스템 변환의 두 가지를 모두 지원하려 작동하는 맞춤형 하드웨어(customized hardware)를 포함한다. 이들 컴포넌트는 런타임 소프트웨어 실행을 통상의 프로세서의 속도의 다섯 배 또는 그 이상으로 실행시킨다. 시스템 에뮬레이션 및 시스템 변환은 아래에서 논의된다.

[0030] 도 4는 본 발명의 일 실시예에 따른 시스템 에뮬레이션 및 시스템 변환을 구현하기 위한 컴포넌트를 묘사하는 다이어그램을 도시한다. 도 4는 또한 애플리케이션 코드와 OS/시스템 특정 코드를 둘 다 갖는 이미지를 도시한다.

[0031] 본 발명의 실시예는 애플리케이션 코드 및 OS/시스템 특정 코드를 실행하기 위해 시스템 에뮬레이션 및 시스템 변환을 사용한다. 시스템 에뮬레이션을 이용하는 머신은 하드웨어가 지원하는 아키텍처와 상이한 (시스템 및 애플리케이션 코드를 둘 다 갖는) 게스트 시스템 아키텍처를 에뮬레이트하고/가상화한다. 에뮬레이션은 (예를 들어, 시스템 코드를 처리하는) 시스템 에뮬레이션/가상화 변환기 및 (예를 들어, 애플리케이션 코드를 처리하는) 애플리케이션 코드 변환기에 의해 제공된다. 애플리케이션 코드 변환기는 베어 메탈(bare metal) 컴포넌트로 묘

사되는 것으로 도시된다.

[0032] 시스템 변환을 사용하는 머신은 게스트 아키텍처와, 하드웨어를 지원하지만 아키텍처의 시스템 이외의 부분(즉, 애플리케이션 명령어)이 상이한 아키텍처와의 사이의 유사한 시스템 아키텍처 특성을 갖는 코드를 변환한다. 시스템 변환기는 게스트 애플리케이션 변환기 컴포넌트 및 베어 메탈 컴포넌트를 포함하는 것으로 도시된다. 시스템 변환기는 또한 멀티-패스 최적화 프로세서(multi-pass optimization process)를 잠재적으로 구현하는 것으로 도시된다. 시스템 변환 및 에뮬레이션이라는 용어를 언급함으로써, 본 명세서의 다음의 설명은 도 4에 도시된 바와 같이 시스템 에뮬레이션 경로 또는 시스템 변환 경로 중 어느 한 경로를 사용할 수 있는 프로세스를 말하고 있는 것임을 알아야 한다.

[0033] 다음의 도 5 내지 도 26의 다이어그램은 범용의 애그노스틱 런타임 시스템/VISC ISA 애그노스틱 런타임 아키텍처를 지원하기 위한 시스템 에뮬레이션 및 시스템 변환을 모두 구현하는데 사용되는 각종 프로세스 및 시스템을 도시한다. 다음의 다이어그램에서 프로세스 및 시스템을 사용하면, 런타임 코드에 하드웨어/소프트웨어 가속이 제공되며, 그 결과로 아키텍처의 성능이 증가된다. 그러한 아키텍처 가속은 분산된 플래그, CLB, CLBV, 하드웨어 게스트 변환 테이블 등의 지원을 포함한다.

[0034] 도 5는 본 발명의 일 실시예에 따른 게스트 플래그 아키텍처 에뮬레이션을 묘사하는 다이어그램을 도시한다. 도 5의 좌측은 다섯 개의 플래그를 갖는 집중형 플래그 레지스터(centralized flag architecture)를 도시한다. 도 5의 우측은 플래그가 레지스터들 자체 사이에 분산되어 있는 분산된 플래그 레지스터를 갖는 분산형 플래그 아키텍처(distributed flag architecture)를 도시한다.

[0035] 아키텍처 에뮬레이션(예를 들면, 시스템 에뮬레이션 또는 변환) 동안, 분산형 플래그 아키텍처가 집중형 게스트 플래그 아키텍처의 동작을 에뮬레이트하는 것이 필요하다. 분산형 플래그 아키텍처는 또한 데이터 레지스터와 연관된 플래그 필드와 대조적으로 복수의 독립 플래그 레지스터를 사용하여 구현될 수 있다. 예를 들면, 데이터 레지스터는 R0 내지 R15로서 구현될 수 있는데 반해, 독립 플래그 레지스터는 F0 내지 F15로서 구현될 수 있다. 이 경우 이들 플래그 레지스터는 데이터 레지스터와 직접적으로 연관되지 않는다.

[0036] 도 6은 본 발명의 일 실시예에 따른 통합된 레지스터 파일(unified register file)(1201)의 다이어그램을 도시한다. 도 5에 도시된 바와 같이, 통합된 레지스터 파일(1201)은 2개의 부분(1202-1203) 및 엔트리 선택기(1205)를 포함한다. 통합된 레지스터 파일(1201)은 하드웨어 상태 업데이트를 위한 아키텍처 추측(architecture speculation)의 지원을 구현한다.

[0037] 통합된 레지스터 파일(1201)은 최적화된 새도우 레지스터 및 커밋된(committed) 레지스터 상태 관리 프로세스의 구현을 가능하게 한다. 이러한 프로세스는 하드웨어 상태 업데이트를 위한 아키텍처 추측을 지원한다. 이러한 프로세스 하에서, 본 발명의 실시예는 레지스터 메모리 사이에서 어떠한 교차 복사를 요구하지 않고 새도우 레지스터 기능성 및 커밋된 레지스터 기능성을 지원할 수 있다. 예를 들면, 일 실시예에서, 통합된 레지스터 파일(1201)의 기능성은 대체로 엔트리 선택기(1205)에 의해 제공된다. 도 5의 실시예에서, 각 레지스터 파일 엔트리는 각기 부분 1 및 부분 2로부터의 2 쌍의 레지스터(R 및 R')로 구성된다. 어느 시간에서도, 각 엔트리로부터 읽어오는 레지스터는 부분 1 또는 부분 2로부터의 R 또는 R'이다. 엔트리 선택기(1205)에 의해 각 엔트리마다 저장된 x 및 y 비트의 값에 기초하여 레지스터 파일의 각 엔트리에는 4 개의 상이한 조합이 있다.

[0038] 도 7은 본 발명의 일 실시예에 따라 추측 아키텍처 상태 및 비영속성 아키텍처 상태를 지원하는 통합된 새도우 레지스터 파일 및 파이프라인 아키텍처(unified shadow register file and pipeline architecture)(1300)의 다이어그램을 도시한다.

[0039] 도 7의 실시예는 아키텍처 추측 상태를 포함하는 명령어 및 결과를 지원하며, 비영속성 상태를 포함하는 명령어 및 결과를 지원하는 아키텍처(1300)를 구성하는 컴포넌트를 도시한다. 본 명세서에서 사용된 바와 같이, 커밋된 아키텍처 상태는 프로세서 상에서 실행하는 프로그램에 의해 액세스(읽기 및 쓰기)될 수 있는 가시적 레지스터 및 가시적 메모리를 포함한다. 대조적으로, 추측 아키텍처 상태는 커밋되지 않으며, 이에 따라 전역적으로 가시적(globally visible)이지 않은 레지스터 및/또는 메모리를 포함한다.

[0040] 일 실시예에서, 아키텍처(1300)에 의해 인에이블되는 네 개의 사용 모델이 있다. 제 1 사용 모델은 하드웨어 상태 업데이트를 위한 아키텍처 추측을 포함한다.

[0041] 제 2 사용 모델은 이중 스코프 사용(dual scope usage)을 포함한다. 이 사용 모델은 하나의 스레드는 추측 상태에서 실행하고 다른 스레드는 비 추측 상태에서 실행하는, 2개의 스레드를 프로세서로 페치하는데 적용된다. 이

러한 사용 모델에서, 두 스코프는 모두 머신으로 페치되어 동시에 머신에 존재한다.

[0042] 제 3 모델은 명령어를 하나의 형식에서 다른 형식으로 JIT(just-in-time) 번역 또는 컴파일하는 것을 포함한다. 이러한 사용 모델에서, 아키텍처 상태의 재정렬은 소프트웨어, 예를 들면, JIT를 통해 달성된다. 제 3 사용 모델은 예를 들면, 게스트 대 네이티브 명령어 번역, 가상 머신 대 네이티브 명령어 번역, 또는 네이티브 마이크로 명령어를 보다 최적화된 네이티브 마이크로 명령어로의 재맵핑/번역하는 것에 적용될 수 있다.

[0043] 제 4 사용 모델은 비용속성 컨텍스트로부터 복귀할 때 이전 컨텍스트를 저장 및 복구할 필요 없이 비영속성 컨텍스트 스위칭하는 것을 포함한다. 이러한 사용 모델은 여러 가지 이유로 발생할 수 있는 컨텍스트 스위치에 적용된다. 그러한 하나의 이유는 예를 들면, 예외 처리 컨텍스트를 통한 정확한 예외 처리일 수 있다.

[0044] 다시 도 7을 참조하면, 아키텍처(1300)는 전술한 제 4 사용 모델을 구현하기 위한 복수의 컴포넌트를 포함한다. 통합된 새도우 레지스터 파일(1301)은 제 1 부분의 커밋된 레지스터 파일(1302), 제 2 부분의 새도우 레지스터 파일(1303), 및 제 3 부분의 최신 표시자 어레이(latest indicator array)(1302)를 포함한다. 추론적 퇴거 메모리 버퍼(speculative retirement memory buffer)(1342) 및 최신 표시자 어레이(1340)가 포함된다. 아키텍처(1300)는 비순차적 아키텍처를 포함하며, 따라서, 아키텍처(1300)는 재정렬 버퍼 및 퇴거 윈도우(1332)를 더 포함한다. 재정렬 버퍼 및 퇴거 윈도우(1332)는 또한 머신 퇴거 포인터(1331), 준비 비트 어레이(ready bit array)(1334), 및 표시자(1333)와 같은 명령어 별 최신 표시자를 포함한다.

[0045] 하드웨어 상태 업데이트를 위한 아키텍처 추측인 제 1 사용 모델은 본 발명의 일 실시예에 따라서 더 자세하게 설명된다. 전술한 바와 같이, 아키텍처(1300)는 비순차적 아키텍처를 포함한다. 아키텍처(1300)의 하드웨어는 비순차적 명령어 결과(예를 들면, 비순차적 로드(load) 및 비순차적 스토어(store) 및 비순차적 레지스터 업데이트)를 커밋할 수 있다. 아키텍처(1300)는 통합된 새도우 레지스터를 이용하여 커밋된 레지스터와 새도우 레지스터 사이에서 추측 실행을 지원한다. 또한, 아키텍처(1300)는 추론적 로드 스토어 버퍼(1320) 및 추론적 퇴거 메모리 버퍼(1342)를 이용하여 추측 실행을 지원한다.

[0046] 아키텍처(1300)는 이러한 컴포넌트를 재정렬 버퍼 및 퇴거 윈도우(1332)와 함께 사용하여, 머신이 이들을 내부적으로 비순차적 방식으로 통합된 새도우 레지스터 파일 및 퇴거 메모리 버퍼에다 퇴거시켰더라도, 아키텍처의 상태가 커밋된 레지스터 파일(1302) 및 가시적 메모리(1350)에 정확하게 퇴거할 수 있게 할 것이다. 예를 들면, 아키텍처는 통합된 새도우 레지스터 파일(1301) 및 추측 메모리(1342)를 사용하여 예외가 발생하는지 또는 발생하지 않는지에 따라 롤백(rollback) 및 커밋 이벤트를 구현할 것이다. 이러한 기능성은 레지스터 상태가 통합된 새도우 레지스터 파일(1301)에 비순차적으로 퇴거할 수 있게 하며 추론적 퇴거 메모리 버퍼(1342)가 비순차적으로 가시적 메모리(1350)에 퇴거할 수 있게 한다. 추측 실행이 진행되고 비순차적 명령어 실행이 진행됨에 따라, 어떤 분기(branch)도 잘못 예측(missed predicted)되지 않고 아무 예외도 발생하지 않으면, 머신 퇴거 포인터(1331)는 커밋 이벤트가 트리거될 때까지 진행한다. 커밋 이벤트는 통합된 새도우 레지스터 파일이 커밋 포인트를 앞당김으로써 그의 콘텐츠를 커밋하게 하며, 추론적 퇴거 메모리 버퍼가 머신 퇴거 포인트(1331)에 따라 그의 콘텐츠를 메모리(1350)에 커밋하도록 한다.

[0047] 예를 들면, 재정렬 버퍼 및 퇴거 윈도우(1332) 내에서 보이는 명령어 1-7을 고려해 보면, 준비 비트 어레이(1334)는 "X" 옆의 명령어가 실행할 준비가 되어 있는 것을 표시하고, "/" 옆의 명령어는 실행할 준비가 되지 않은 것을 표시한다. 따라서, 명령어 1, 2, 4, 및 6은 비순차적으로 진행하게 된다. 그 결과, 명령어 6의 분기가 잘못 예측되는 것과 같은 예외가 발생하면, 명령어 6에 후속하여 발생하는 명령어는 롤백될 수 있다. 대안으로, 만일 아무 예외도 발생하지 않으면, 그에 따라서 모든 명령어 1-7는 머신 퇴거 포인터(1331)를 이동시킴으로써 커밋될 수 있다.

[0048] 최신 표시자 어레이(1341), 최신 표시자 어레이(1304) 및 최신 표시자(1333)는 비순차적 실행을 가능하게 하는데 사용된다. 예를 들면, 명령어 2가 명령어 5의 앞에 레지스터 R4를 로드하더라도, 명령어 5가 발생할 준비가 되면 명령어 2로부터의 로드는 무시될 것이다. 최신의 표시자에 따라서 최신의 로드는 이전의 로드에 우선시 될 것이다.

[0049] 분기 예측 또는 예외가 재정렬 버퍼 및 퇴거 윈도우(1332) 내에서 발생하는 경우, 롤백 이벤트가 트리거된다. 전술한 바와 같이, 롤백의 이벤트에서, 통합된 새도우 레지스터 파일(1301)은 그의 최종 커밋된 포인트로 롤백할 것이며, 추론적 퇴거 메모리 버퍼(1342)는 지워(flush)질 것이다.

[0050] 도 8은 본 발명의 일 실시예에 따른 런 어헤드 배치/변환 프로세스(run ahead batch/conversion process)를 묘사하는 다이어그램을 도시한다. 이 도면은 게스트 코드가 변환 프로세스를 거쳐 네이티브 코드로 변환되는 방식

을 다이어그램으로 도시한다. 이러한 네이티브 코드는 CLB를 채우기 위해서도 사용되는 네이티브 코드 캐시에 채워진다. 도면은 게스트 코드가 이전에 변환되지 않았던 어드레스(예를 들면, 5000)로 어떻게 점프하는지를 보여준다. 변환 프로세스는 이 게스트 코드를 도시된 바와 같은 (예를 들면, 게스트 분기(8000) 및 추측 분기(guess branch)(6000)를 포함하는) 대응하는 네이티브 코드로 바꾼다. 추측 분기는 코드 캐시에서 네이티브 분기(예를 들면, 네이티브 분기(g8000) 및 네이티브 분기(g6000))로 변환된다. 머신은 네이티브 분기에 대한 프로그램 카운터가 추측 분기에 대한 프로그램 카운터와 다르게 진행 중이라는 것을 자각하고 있다. 이것은 네이티브 코드 캐시에서 기호(예를 들면, X, Y 및 Z)로 표시된다. 이러한 번역이 완료되면, 결과의 번역은 향후의 사용을 위해 CLB에 저장된다. 이러한 기능성은 게스트 코드의 네이티브 코드로의 번역 속도를 크게 높여준다.

[0051] 도 9는 본 발명의 일 실시예에 따라 게스트 명령어 블록 및 그의 대응하는 네이티브 변환 블록이 캐시 내에 저장되는 방식을 예시하는 예시적인 하드웨어 가속된 변환 시스템(500)의 다이어그램을 도시한다. 도 9에 예시된 바와 같이, 변환 룩 어사이드 버퍼(conversion look aside buffer)(506)는 게스트 블록과 네이티브 블록 사이의 어드레스 맵핑을 캐싱하는데 사용되며, 그래서 가장 빈번히 접하는 네이티브 변환 블록은 낮은 지연 가용성으로 프로세서(508)에 액세스된다.

[0052] 도 9의 다이어그램은 빈번히 접하는 네이티브 변환 블록이 고속의 저 지연 캐시, 즉, 변환 룩 어사이드 버퍼(506) 내에 유지되는 방식을 예시한다. 도 9에 도시된 컴포넌트는 하드웨어 가속된 변환 처리를 구현하여 훨씬 높은 수준의 성능을 산출한다.

[0053] 게스트 페치 로직 유닛(502)은 시스템 메모리(501)로부터 게스트 명령어를 페치하는 하드웨어 기반의 게스트 명령어 페치 유닛으로서 기능한다. 특정 애플리케이션의 게스트 명령어는 시스템 메모리(501) 내에 상주한다. 프로그램이 시작되면, 하드웨어 기반의 게스트 페치 로직 유닛(502)은 추측 명령어를 게스트 페치 버퍼(503)로 프리페치(prefetch)하기 시작한다. 게스트 페치 버퍼(507)는 게스트 명령어를 누적하고 이를 게스트 명령어 블록으로 조립한다. 이러한 게스트 명령어 블록은 변환 테이블(504)을 사용하여 대응하는 네이티브 변환 블록으로 변환된다. 변환된 네이티브 명령어는 네이티브 변환 블록이 완성될 때까지 네이티브 변환 버퍼(505) 내에 누적된다. 그런 다음 네이티브 변환 블록은 네이티브 캐시(507)로 전송되며 맵핑은 변환 룩 어사이드 버퍼(506)에 저장된다. 네이티브 캐시(507)는 실행을 위해 네이티브 명령어를 프로세서(508)에 공급하는데 사용된다. 일 실시예에서, 게스트 페치 로직 유닛(502)에 의해 구현되는 기능성은 게스트 페치 로직 상태 머신에 의해 생성된다.

[0054] 이렇게 프로세스가 계속됨에 따라, 변환 룩 어사이드 버퍼(506)는 게스트 블록과 네이티브 블록과의 어드레스 맵핑으로 채워진다. 변환 룩 어사이드 버퍼(506)는 하나 이상의 알고리즘(예를 들면, 최근 최소 사용(least recently used) 등)을 사용하여, 더욱 빈번히 접하는 블록 맵핑이 버퍼 내에 유지되게 하는 반면에, 드물게 접하는 블록 맵핑이 버퍼에서 축출되도록 보장한다. 이러한 방식으로, 사용이 많은(hot) 네이티브 변환 블록 맵핑은 변환 룩 어사이드 버퍼(506) 내에 저장된다. 또한, 네이티브 블록 내에서 잘 예측된 멀리 떨어진 게스트 분기는 이들 분기의 타겟 블록이 단일 맵핑된 네이티브 블록 내에 합쳐지기(stitch) 때문에 새로운 맵핑을 CLB에 삽입할 필요가 없으며, 그래서 CLB 구조에는 용량 효율성을 작게 보전한다는 것을 주목하여야 한다. 또한, 일 실시예에서, CLB는 엔딩 게스트 대 네이티브 어드레스 맵핑만을 저장하도록 구성된다. 이러한 측면은 또한 CLB의 작은 용량 효율성을 보전한다.

[0055] 게스트 페치 로직(502)은 변환 룩 어사이드 버퍼(506)를 검색하여 게스트 명령어 블록으로부터의 어드레스가 이미 네이티브 변환 블록으로 변환되었는지를 결정한다. 전술한 바와 같이, 본 발명의 실시예는 변환 처리를 위한 하드웨어 가속을 제공한다. 따라서, 게스트 페치 로직(502)은 새로운 변환을 위해 시스템 메모리(501)로부터 게스트 어드레스를 페치하기 전에 변환 룩 어사이드 버퍼(506)를 검색하여 기존의 네이티브 변환 블록 맵핑을 찾을 것이다.

[0056] 일 실시예에서, 변환 룩 어사이드 버퍼는 게스트 어드레스 범위(guest address range) 또는 개별 게스트 어드레스에 의해 인덱스된다. 게스트 어드레스 범위는 네이티브 변환 블록으로 변환된 게스트 명령어 블록의 어드레스의 범위이다. 변환 룩 어사이드 버퍼에 의해 저장된 네이티브 변환 블록 맵핑은 대응하는 게스트 변환 블록의 대응하는 게스트 어드레스 범위를 통해 인덱스된다. 따라서, 게스트 페치 로직은 게스트 어드레스를 변환된 블록의 게스트 어드레스 범위 또는 개별 게스트 어드레스와 비교할 수 있으며, 그의 맵핑은 변환 룩 어사이드 버퍼(506)에서 유지되어 기존의 네이티브 변환 블록이 네이티브 캐시(507)에 또는 도 6의 코드 캐시에 저장된 것 중에 존재하는지를 결정한다. 만일 기존의 네이티브 변환 블록이 네이티브 캐시 또는 코드 캐시 중 어느 한 곳에 있으면, 대응하는 네이티브 변환 명령어는 이들 캐시로부터 프로세서에 직접 포워딩된다.

[0057] 이러한 방식으로, 사용이 많은 게스트 명령어 블록(예를 들면, 빈번히 실행되는 게스트 명령어 블록)은 고속의 저 지연 변환 룩 어사이드 버퍼(506) 내에서 유지되는 그의 대응하는 사용이 많은 네이티브 변환 블록 맵핑을 갖는다. 블록이 찾아지면, 적절한 대체 정책은 변환 룩 어사이드 버퍼 내에 사용이 많은 블록 맵핑이 확실하게 남아 있게 한다. 따라서, 게스트 페치 로직 유닛(502)은 요청된 게스트 어드레스가 이미 변환되었는지를 신속하게 식별할 수 있으며, 이전에 변환된 네이티브 명령어를 프로세서(508)에 의한 실행을 위해 네이티브 캐시(507)로 직접 포워딩할 수 있다. 이러한 양상은, 시스템 메모리로의 여정이 40 내지 50 사이클 또는 그 이상 소요할 수 있기 때문에, 다수의 사이클을 절감할 수 있다. 이러한 속성(예를 들면, CLB, 게스트 분기 시퀀스 예측, 게스트 및 네이티브 분기 버퍼, 이전 것의 네이티브 캐싱)은 본 발명의 실시예의 하드웨어 가속 기능성이 게스트 애플리케이션의 애플리케이션 성능을 비교 가능한 네이티브 애플리케이션의 애플리케이션 성능의 80% 내지 100% 내까지 성취할 수 있게 한다.

[0058] 일 실시예에서, 게스트 페치 로직(502)은 프로세서(508)로부터 게스트 명령어 요청과 무관하게 변환을 위한 게스트 명령어를 계속하여 프리페치한다. 네이티브 변환 블록은 덜 빈번하게 사용된 블록에 대한 시스템 메모리(501)의 변환 버퍼 "코드 캐시" 내에 누적될 수 있다. 변환 룩 어사이드 버퍼(506)는 또한 가장 빈번하게 사용된 맵핑을 유지한다. 따라서, 만일 요청된 게스트 어드레스가 변환 룩 어사이드 버퍼 내 게스트 어드레스에 맵핑되지 않으면, 게스트 페치 로직은 시스템 메모리(501)를 체크하여 게스트 어드레스가 시스템 메모리에 저장된 네이티브 변환 블록에 대응하는지를 결정할 수 있다.

[0059] 일 실시예에서, 변환 룩 어사이드 버퍼(506)는 캐시로서 구현되며 캐시 일관성 프로토콜을 이용하여 상위 레벨의 캐시 및 시스템 메모리(501)에 저장된 훨씬 더 큰 변환 버퍼와의 일관성을 유지한다. 변환 룩 어사이드 버퍼(506)에 저장된 네이티브 명령어 맵핑은 상위 레벨의 캐시 및 시스템 메모리(501)에도 또한 라이트 백(write back)된다. 시스템 메모리로의 라이트 백은 일관성을 유지한다. 이런 이유로, 캐시 관리 프로토콜은 사용이 많은 네이티브 변환 블록 맵핑이 변환 룩 어사이드 버퍼(506) 내에 저장되고 사용이 덜한(cold) 네이티브 변환 맵핑 블록이 시스템 메모리(501)에 저장되는 것을 보장하는데 사용될 수 있다. 그러므로 훨씬 큰 형태의 변환 룩 어사이드 버퍼(506)가 시스템 메모리(501)에 상주한다.

[0060] 일 실시예에서, 예시적인 하드웨어 가속된 변환 시스템(500)은 복수의 상이한 가상 저장 방식을 구현하는데 사용될 수 있다는 것을 알아야 한다. 예를 들면, 게스트 명령어 블록 및 그의 대응하는 네이티브 변환 블록이 캐시 내에 저장되는 방식은 가상의 저장 방식을 지원하는데 사용될 수 있다. 유사하게, 게스트 블록과 네이티브 블록 간의 어드레스 맵핑을 캐싱하는데 사용되는 변환 룩 어사이드 버퍼(506)는 가상의 저장 방식(예를 들면, 가상 대 물리 메모리 맵핑의 관리)를 지원하는데 사용될 수 있다.

[0061] 일 실시예에서, 도 9의 아키텍처는 복수의 상이한 명령어 아키텍처를 입력으로서 수신할 수 있는 유연한 변환 프로세스를 사용하는 가상 명령어 집합 프로세서/컴퓨터를 구현한다. 그러한 가상 명령어 집합 프로세서에서, 프로세서의 프론트 엔드는 이것이 훨씬 상위 레벨의 성능을 내놓기 위해 하드웨어 가속된 변환 프로세싱을 활용하면서, 소프트웨어로 제어될 수 있도록 구현된다. 그러한 구현을 사용하면, 여러 게스트 아키텍처는 각 아키텍처가 하드웨어 가속의 장점을 이어 받아 훨씬 높은 레벨의 성능을 누리면서 처리되고 변환될 수 있다. 게스트 아키텍처의 예는 자바(Java) 또는 자바스크립트(JavaScript), x86, MIPS, SPARC 등을 포함한다. 일 실시예에서, "게스트 아키텍처"는 (예를 들어, 네이티브 애플리케이션/매크로 연산으로부터의) 네이티브 명령어일 수 있으며, 변환 프로세스는 최적화된 네이티브 명령어(예를 들면, 최적화된 네이티브 명령어/마이크로 연산)을 생성한다. 소프트웨어로 제어되는 프론트 엔드는 프로세서상에서 실행하는 애플리케이션에 대해 상당한 정도의 유연성을 제공할 수 있다. 전술한 바와 같이, 하드웨어 가속은 게스트 애플리케이션의 게스트 명령어의 실행에 필요한 네이티브 하드웨어 속도를 거의 성취할 수 있다.

[0062] 도 10은 본 발명의 일 실시예에 따른 하드웨어 가속된 변환 시스템(600)의 더 상세한 예를 도시한다. 시스템(600)은 전술한 시스템(500)과 실질적으로 동일한 방식으로 수행한다. 그러나 시스템(600)은 예시적인 하드웨어 가속 프로세스의 기능성을 기술하는 추가적인 세부 사항을 보여준다.

[0063] 시스템 메모리(601)는 게스트 코드(602), 변환 룩 어사이드 버퍼(603), 최적화기 코드(604), 변환기 코드(605), 및 코드 캐시(606)를 구비하는 데이터 구조를 포함한다. 시스템(600)은 또한 게스트 명령어 및 네이티브 명령어가 모두 다 인터리빙되고 공유될 수 있는 공유된 하드웨어 캐시(607)를 포함한다. 게스트 하드웨어 캐시(610)는 공유된 하드웨어 캐시(607)로부터 가장 빈번하게 접촉되는 게스트 명령어를 캐치한다.

[0064] 게스트 페치 로직(620)은 게스트 코드(602)로부터 게스트 명령어를 프리페치한다. 게스트 페치 로직(620)은 게스트 어드레스를 대응하는 물리 게스트 어드레스로 변환하는 변환 룩 어사이드 버퍼로서 기능하는 TLB(609)와

인터페이스한다. TLB(609)는 히트(hit)를 직접 게스트 하드웨어 캐시(610)에 포워딩한다. 게스트 페치 로직(620)에 의해 페치된 게스트 명령어는 게스트 페치 버퍼(611)에 저장된다.

[0065] 변환 테이블(612 및 613)은 대체 필드 및 제어 필드를 포함하며, 게스트 페치 버퍼(611)로부터 수신된 게스트 명령어를 네이티브 명령어로 번역하는 다중레벨 변환 테이블로서 기능을 한다.

[0066] 멀티플렉서(614 및 615)는 변환된 네이티브 명령어를 네이티브 변환 버퍼(616)로 전송한다. 네이티브 변환 버퍼(616)는 변환된 네이티브 명령어를 누적하여 네이티브 변환 블록을 조립한다. 그 다음, 이들 네이티브 변환 블록은 네이티브 하드웨어 캐시(600)로 전달되며, 맵핑은 변환 룩 어사이드 버퍼(630)에서 유지된다.

[0067] 변환 룩 어사이드 버퍼(630)는 변환된 블록 엔트리 포인트 어드레스(631), 네이티브 어드레스(632), 변환된 어드레스 범위(633), 코드 캐시 및 변환 룩 어사이드 버퍼 관리 비트(634), 및 동적 분기 바이어스 비트(dynamic branch bias bit)(635)를 위한 데이터 구조를 포함한다. 게스트 분기 어드레스(631) 및 네이티브 어드레스(632)는 대응하는 어느 네이티브 변환 블록이 변환된 블록 범위(633) 내에 상주하는 것을 표시하는 게스트 어드레스 범위를 포함한다. 캐시 관리 프로토콜 및 대체 정책은 사용이 많은 네이티브 변환 블록 맵핑이 변환 룩 어사이드 버퍼(630) 내에 상주하게 하지만, 사용이 덜한 네이티브 변환 블록 맵핑은 시스템 메모리(601) 내 변환 룩 어사이드 버퍼 데이터 구조(603) 내에 상주하는 것을 보장한다.

[0068] 시스템(500)과 마찬가지로, 시스템(600)은 사용이 많은 블록 맵핑이 고속의 저 지연 변환 룩 어사이드 버퍼(630) 내에 상주하는 것을 보장하도록 모색한다. 그래서, 일 실시예에서, 페치 로직(640) 또는 게스트 페치 로직(620)이 게스트 어드레스를 페치하기 위해 탐색할 때, 페치 로직(640)은 대응하는 네이티브 변환 블록이 코드 캐시(606) 내에 상주하는지를 결정하기 위해 게스트 어드레스를 먼저 체크할 수 있다. 이렇게 하면 요청된 게스트 어드레스가 코드 캐시(606) 내에 대응하는 네이티브 변환 블록을 갖고 있는지에 대해 결정될 수 있다. 만일 요청된 게스트 어드레스가 버퍼(603 또는 608) 또는 버퍼(630) 중 어느 한 곳에도 상주하지 않으면, 게스트 어드레스 및 복수의 후속 게스트 명령어가 게스트 코드(602)로부터 페치되며, 변환 프로세스가 변환 테이블(612 및 613)을 통해 실시된다. 이러한 방식으로, 본 발명의 실시예는 런 어헤드 게스트 페치 및 디코드, 테이블 검색 및 명령어 필드 조립을 구현할 수 있다.

[0069] 도 11은 본 발명의 일 실시예에 따른 이중 스코프 사용(dual scope usage)을 포함하는, 제 2 사용 모델의 다이어그램을 도시한다. 전술한 바와 같이, 이 사용 모델은 하나의 스레드가 추측 상태에서 실행되고 다른 스레드가 비 추측 상태에서 실행되는 2개의 스레드를 프로세서로 페치하는 것에 적용된다. 이러한 사용 모델에서, 두 스코프는 모두 머신으로 페치되고 동시에 머신에 존재한다.

[0070] 다이어그램(1400)에서 도시된 바와 같이, 2개의 스코프/트레이스(1401 및 1402)는 머신으로 페치되었다. 이 예에서, 스코프/트레이스(1401)는 비 추측적인 현재의 스코프/트레이스이다. 스코프/트레이스(1402)는 추측적인 새로운 스코프/트레이스이다. 아키텍처(1300)는 2개의 스레드가 실행을 위해 추측 및 스크래치(scratch) 상태를 사용하게 하는 그와 같은 상태를 가능할 수 있게 한다. 하나의 스레드(예를 들면, (1401))은 비 추측 스코프에서 실행하며, 다른 스레드(예를 들면, 1402)는 추측 스코프를 사용한다. 두 스코프는 모두 머신으로 페치될 수 있고 동시에 존재할 수 있으며, 이때 각 스코프는 그의 각 모드를 상이하게 설정한다. 첫 번째 스코프는 비 추측이며 다른 스코프는 추측이다. 그래서 첫 번째 스코프는 CR/CM 모드에서 실행하고 다른 스코프는 SR/SM 모드에서 실행한다. CR/CM 모드에서, 커밋된 레지스터가 읽고 쓰여지며, 메모리 쓰기는 메모리에 전달된다. SR/SM 모드에서, 레지스터 쓰기는 SSSR에 전달되며, 레지스터 읽기는 최신의 쓰기로부터 이루어지는 반면, 메모리는 퇴거 메모리 버퍼(SMB)에 쓴다.

[0071] 하나의 예는 순서가 정해진 현재 스코프(예를 들면, 1401) 및 추측되는 다음 스코프(예를 들면, (1402))일 것이다. 두 스코프는 다음 스코프가 현재 스코프 다음에 페치되기 때문에 의존 관계가 지켜지므로 머신에서 실행될 수 있다. 예를 들면, 스코프(1401)에서, "SSSR을 CR에 커밋"에서, 이 지점까지의 레지스터 및 메모리는 CR 모드에 있지만, 코드는 CR/CM 모드에서 실행한다. 스코프(1402)에서, 코드는 SR 및 SM 모드에서 실행하며, 예외가 발생한다면, 롤백될 수 있다. 이러한 방식으로, 두 스코프는 동시에 머신에서 실행되지만, 각각은 상이한 모드에서 실행 중이고 그에 따라 레지스터를 읽고 레지스터에 쓴다.

[0072] 도 12는 본 발명의 일 실시예에 따라 비영속성 컨텍스트(transient context)로부터 복귀할 때 이전 컨텍스트를 저장 및 복원할 필요가 없는 비영속성 컨텍스트 스위칭을 포함하는 제 3 사용 모델의 다이어그램을 도시한다. 전술한 바와 같이, 이 사용 모델은 여러 이유 때문에 발생할 수 있는 컨텍스트 스위치에 적용된다. 하나의 그러한 이유는 예를 들면, 예외 처리 컨텍스트를 통한 정확한 예외 처리일 수 있다.

[0073] 제 3 사용 모델은 머신이 번역된 코드를 실행하고 있고 컨텍스트 스위치를 접했을 때(예를 들면, 번역된 코드의 내부에서 예외 시, 또는 후속 코드의 번역이 필요한 경우) 발생한다. (예를 들어, 예외 이전의) 현재 스코프에서, SSSR 및 SMB는 아직 이들의 추측 상태를 게스트 아키텍처 상태로 커밋하지 않았다. 현재상태는 SR/SM 모드에서 실행 중이다. 예외가 발생할 때, 머신은 예외 핸들러(예를 들면, 변환기(convertor))로 스위칭하여 예외를 정확히 처리한다. 롤백이 삽입되어, 레지스터 상태를 CR로 롤백하게 하며, SMB는 지워진다. 변환기 코드는 SR/CM 모드에서 실행될 것이다. 변환기 코드의 실행 중에, SMB는 커밋 이벤트를 기다리지 않고 그의 콘텐츠를 메모리에 퇴거시킨다. 레지스터는 CR을 업데이트하지 않고 SSSR에 기록된다. 이어서, 변환기가 종료되고 변환된 코드를 실행하려 다시 스위칭하기 전에, 변환기는 SSSR을 롤백한다(예를 들면, SSSR은 CR로 롤백된다). 이러한 프로세스 동안, 마지막 커밋된 레지스터 상태는 CR에 있다.

[0074] 이것은 이전의 스코프/트레이스(1501)가 SSSR로부터 CR로 커밋된 다이어그램(1500)에서 도시된다. 현재 스코프/트레이스(1502)는 추측이다. 레지스터와 메모리 및 이 스코프는 추측이며 실행은 SR/SM 모드 하에서 발생한다. 이 예에서, 예외는 스코프(1502)에서 발생하며 코드는 번역되기 전의 원래 순서대로 재실행되어야 한다. 이 시점에서, SSSR은 롤백되며 SMB는 지워진다. 그 다음 JIT 코드(1503)가 실행된다. JIT 코드는 SSSR을 스코프(1501)의 끝으로 롤백하고 SMB를 지운다. JIT의 실행은 SC/CM 모드 하에서 수행된다. JIT가 종료될 때, SSSR은 CR로 롤백되며 그런 다음 현재 스코프/트레이스(1504)는 원래의 번역 순서대로 CR/CM 모드에서 재실행된다. 이러한 방식으로, 예외는 바로 그 현재 순서에서 정확히 처리된다.

[0075] 도 13은 본 발명의 일 실시예에 따라 후속 코드에 대해 변환이 필요하기 때문에 명령어 시퀀스에서 예외가 있는 경우를 묘사하는 다이어그램(1600)을 도시한다. 다이어그램(1600)에서 도시된 바와 같이, 이전 스코프/트레이스(1601)는 번역되지 않은 목적지로 멀리 점프하는 것으로 끝난다. 먼 점프 목적지로 점프하기에 앞서, SSSR은 CR로 커밋된다. 그런 다음 JIT 코드(1602)는 (예를 들면, 네이티브 명령어의 새로운 트레이스를 구축하기 위해) 먼 점프 목적지에서 추측 명령어의 번역을 실행한다. JIT의 실행은 SR/CM 모드 하에서 수행된다. JIT 실행이 끝날 때, 레지스터 상태는 SSSR로부터 CR로 롤백되며, JIT에 의해 번역되었던 새로운 스코프/트레이스(1603)가 실행을 시작한다. 새로운 스코프/트레이스는 SR/SM 모드에서 이전 스코프/트레이스(1601)의 마지막 커밋된 지점으로부터 실행을 이어간다.

[0076] 도 14는 본 발명의 일 실시예에 따라 비영속성 컨텍스트로부터 복귀할 때 이전 컨텍스트를 저장 및 복원할 필요 없는 비영속성 컨텍스트 스위칭을 포함하는, 제 4 사용 모델의 다이어그램(1700)을 도시한다. 전술한 바와 같이, 이 사용 모델은 여러 이유로 인해 발생할 수 있는 컨텍스트 스위치에 적용된다. 하나의 그러한 이유는 예를 들면, 예외 처리 컨텍스트를 통한 프로세싱 입력 또는 출력일 수 있다.

[0077] 다이어그램(1700)은 CR/CM모드 하에서 실행 중인 이전 스코프/트레이스(1701)가 함수 F1의 호출로 끝나는 경우를 도시한다. 그 시점까지의 레지스터 상태는 SSSR로부터 CR로 커밋된다. 그 다음, 함수 F1 스코프/트레이스(1702)는 SR/CM 모드 하에서 추측적으로 실행하기 시작한다. 이후 함수 F1은 메인 스코프/트레이스(1703)로 복귀하여 종료한다. 이 시점에서, 레지스터 상태는 SSSR 로부터 CR로 롤백된다. 메인 스코프/트레이스(1703)는 CR/CM 모드에서 실행을 다시 시작한다.

[0078] 도 15는 본 발명의 일 실시예에 따라 명령어를 분기(branch)의 앞으로 최적화하게 스케줄링하는 것을 예시하는 다이어그램을 도시한다. 도 15에 예시된 바와 같이, 하드웨어 최적화된 예는 기존의 저스트-인-타임 컴파일러의 예와 함께 도시된다. 도 15의 좌측에는 바이어스가 이루어지지 않은 분기(branch biased untaken)를 포함하는 원래의 최적화되지 않은 코드인 "Branch C to LI"가 도시된다. 도 15의 가운데 열(column)에는 레지스터가 리네임되고 명령어가 분기보다 앞으로 옮겨진 기존의 저스트-인-타임 컴파일러 최적화를 도시한다. 이 예에서, 저스트-인-타임 컴파일러는 분기의 바이어스 결정이 잘못된 경우(예를 들면, 분기가 선택되지 않은 것과 반대로 실제로는 선택된 경우)를 감안하기 위해 보상 코드를 삽입한다. 대조적으로, 도 15의 우측 열에는 하드웨어 언롤링된 최적화(hardware unrolled optimization)가 도시된다. 이 경우, 레지스터는 리네임되고 명령어는 분기의 앞으로 옮겨진다. 그러나 아무 보상 코드도 삽입되지 않은 것을 주목하여야 한다. 하드웨어는 분기 바이어스 결정이 참(true)인지 아닌지를 추적한다. 잘못 예측된 분기의 경우, 올바른 명령어 시퀀스를 실행하기 위해 하드웨어는 자동으로 그의 상태를 롤백한다. 하드웨어 최적화기 해법은 분기가 잘못 예측되는 그런 경우에, 하드웨어는 잘못 예측된 명령어 시퀀스를 지우면서 메모리에 있는 원래 코드로 점프하여 그곳에서부터 올바른 시퀀스를 실행하기 때문에 보상 코드의 사용을 피할 수 있다.

[0079] 도 16은 본 발명의 일 실시예에 따라 로드(load)를 스토어(store)의 앞으로 최적화하게 스케줄링하는 것을 예시하는 다이어그램을 도시한다. 도 16에 예시된 바와 같이, 하드웨어 최적화된 예는 기존의 저스트-인-타임 컴파

일러의 예와 함께 도시된다. 도 16의 좌측에는 스토어를 포함하는 원래의 최적화되지 않은 코드인 "R3 <- LD [R5]"가 보인다. 도 16의 가운데 열은 레지스터가 리네임되고 로드가 스토어의 앞으로 옮겨진 기존의 저스트-인-타임 컴파일러 최적화를 도시한다. 이 예에서, 저스트-인-타임 컴파일러는 로드 명령어의 어드레스가 저장 명령어의 어드레스를 엘리어싱(aliasing)하는 그러한 경우(예를 들면, 스토어의 앞에 로드를 옮긴 것이 적절하지 않은 경우)를 감안하기 위해 보상 코드를 삽입한다. 대조적으로, 도 16의 우측은 하드웨어 언롤링된 최적화를 도시한다. 이 경우, 레지스터는 리네임되며 로드는 또한 저장의 앞으로 옮겨진다. 그러나 아무 보상 코드도 삽입되지 않았다는 것을 주목하여야 한다. 스토어의 앞으로 로드를 옮기는 것이 잘못된 경우, 하드웨어는 올바른 명령어 시퀀스를 실행하기 위해 그의 상태를 자동으로 롤백한다. 하드웨어 최적화기 해법은 어드레스 엘리어스-체크 분기(address alias-check branch)가 잘못 예측되는 경우에, 하드웨어는 잘못 예측된 명령어 시퀀스를 지우면서 메모리에 있는 원래 코드로 점프하여 그곳에서부터 올바른 시퀀스를 실행하기 때문에, 보상 코드의 사용을 피할 수 있다. 이 경우, 시퀀스는 아무 엘리어스싱도 추정하지 않는다. 일 실시예에서, 도 16에서 다이어그램으로 예시된 기능성은 명령어 스케줄링 및 최적화기 컴포넌트에 의해 구현될 수 있다는 것을 알아야 한다. 유사하게, 일 실시예에서, 도 16에서 다이어그램으로 예시된 기능성은 소프트웨어 최적화기에 의해 구현될 수 있다는 것을 알아야 한다.

[0080] 또한, 동적으로 언롤링된 시퀀스와 관련하여, 명령어는 리네이밍을 사용함으로써 이전 경로 예측된 분기(예를 들면, 동적으로 구성된 분기)를 지나갈 수 있다는 것을 유의하여야 한다. 비 동적으로 예측된 분기의 경우, 명령어의 이동은 분기의 스코프라고 간주하여야 한다. 루프는 원하는 범위까지 언롤링될 수 있으며 최적화는 전체 시퀀스에 적용될 수 있다. 예를 들면, 이것은 분기를 거쳐 이동하는 명령어의 목적지 레지스터를 리네임함으로써 구현될 수 있다. 이러한 특징의 장점 중 하나는 아무 보상 코드도 없거나 분기의 스코프를 광범위하게 분석하지 않아도 된다는 것이다. 그래서 이러한 특징은 최적화 프로세스를 크게 가속화시키고 단순화한다.

[0081] 도 17은 본 발명의 일 실시예에 따른 스토어 필터링 알고리즘의 다이어그램을 도시한다. 도 17의 실시예의 목적은 모든 스토어가 로드 큐(load queue)에 있는 모든 엔트리에 대해 체크받지 않게 필터링하는 것이다.

[0082] 스토어는 일관성을 유지하도록 어드레스 매치(address match)를 위해 캐시를 스누핑(snoop)한다. 만일 스레드/코어(thread/core) X 로드가 캐시 라인으로부터 판독하면, 데이터를 로드했던 캐시 라인의 부분을 표시한다. 다른 스레드/코어 Y 스토어가 캐시를 스누핑할 때, 임의의 그러한 스토어가 그 캐시 라인 부분과 겹치면, 스레드/코어 X의 그 로드에 대해 잘못 예측이 유발된다.

[0083] 이러한 스누프를 필터링하는 하나의 해결책은 로드 큐 엔트리의 참조를 추적하는 것이다. 이 경우, 스토어는 로드 큐를 스누핑할 필요가 없다. 만일 스토어가 액세스 마스크(access mask)와 매치한다면, 참조 추적기로부터 가져온 그 로드 큐 엔트리는 그 로드 엔트리가 잘못 예측하게 할 것이다.

[0084] (참조 추적기가 없는) 다른 해결책에서, 스토어가 액세스 마스크와 매치한다면, 그 스토어 어드레스는 로드 큐 엔트리를 스누핑할 것이며 매칭된 로드 엔트리가 잘못 예측하게 할 것이다.

[0085] 두 가지 해결책에 따르면, 로드가 캐시 라인으로부터 읽기 중이면, 로드는 각 액세스 마스크 비트를 세트한다. 그 로드가 퇴거될 때, 로드는 그 비트를 리셋한다.

[0086] 도 18은 본 발명의 일 실시예에 따라 순차적으로 메모리로부터 읽는 로드를 구성하는 메모리 일관성 모델(memory consistency model)에서 로드가 비순차적(out of order)인 세마포어(semaphore) 구현을 도시한다. 본 명세서에서 사용된 바와 같이, 세마포어라는 용어는 여러 스레드/코어에게 공통의 자원으로의 액세스 제어를 제공하는 데이터 구성을 말한다.

[0087] 도 18의 실시예에서, 액세스 마스크는 여러 스레드/코어에 의한 메모리 자원으로의 액세스를 제어하는데 사용된다. 액세스 마스크는 캐시 라인의 어느 워드가 로드를 보류 중인지를 추적함으로써 기능한다. 비순차적 로드는 캐시 라인의 워드를 액세스할 때 마스크 비트를 세트하고, 로드가 퇴거될 때 그 마스크 비트를 클리어한다. 마스크 비트가 세트된 동안 다른 스레드/코어로부터 스토어가 그 워드에 쓰면, 스토어는 (예를 들면, 추적기를 통해) 그 로드에 대응하는 로드 큐 엔트리가 잘못 예측될 것/지워질 것이라고 신호하거나 그의 종속 명령어로 다시 시도될 것이라고 신호할 것이다. 액세스 마스크는 또한 스레드/코어를 추적한다.

[0088] 이러한 방식으로, 액세스 마스크는 메모리 일관성 규칙이 올바르게 구현되는 것을 보장한다. 메모리 일관성 규칙은 스토어가 순차적으로 메모리를 업데이트하는 것과 로드가 이러한 세마포어가 두 개의 코어/스레드를 거쳐 작동하도록 순차적으로 메모리로부터 읽는 것을 지시한다. 그러므로 코어 1 및 코어 2에 의해 실행되는 코드는, 이들이 모두 메모리 위치 "플래그" 및 "데이터"에 액세스하는 경우, 올바르게 실행될 것이다.

[0089] 도 19는 본 발명의 일 실시예에 따라 JIT 최적화를 통한 재정렬 프로세스의 다이어그램을 도시한다. 도 19는 메모리 일관성 정렬(예를 들면, 로드 정렬 이후 로드(loads before loads ordering))를 도시한다. 로드는 동일한 어드레스로 오는 다른 로드의 앞으로 디스패치(dispatch)될 수 없다. 예를 들면, 로드는 동일한 스레드로부터 후속 로드의 동일한 어드레스에 대해 체크할 것이다.

[0090] 일 실시예에서, 후속하는 모든 로드는 어드레스 매치에 대해 체크된다. 이러한 해결책을 작동하게 하기 위해, Load C 체크는 원래 Load C 위치의 지점까지 퇴거한 후 스토어 큐(예를 들면, 또는 그의 확장(extension))에 그대로 있어야 한다. 로드 체크 확장 크기는 재정렬된 로드(예를 들면, Load C)가 앞으로 점프할 수 있는 로드의 수에 제한을 둠으로써 결정될 수 있다. 이러한 해결책은 부분 스토어 정렬 메모리 일관성 모델(partial store ordering memory consistency model)(예를 들면, ARM 일관성 모델)하고만 작동한다는 것을 주목하여야 한다.

[0091] 도 20은 본 발명의 일 실시예에 따라 JIT 최적화를 통한 재정렬 프로세스의 다이어그램을 도시한다. 로드는 동일한 어드레스로 오는 다른 로드의 앞으로 디스패치될 수 없다. 예를 들면, 로드는 동일한 스레드로부터 후속 로드의 동일한 어드레스에 대해 체크할 것이다. 도 20은 다른 스레드 스토어가 전체 로드 큐 및 모니터 확장을 어떻게 체크하는지를 보여준다. 모니터는 원래 로드에 의해 세트되고 원래 로드 위치 다음에 오는 후속 명령어에 의해 클리어된다. 이러한 해결책은 전체 및 부분 스토어 정렬 메모리 일관성 모델(예를 들면, x86 및 ARM 일관성 모델) 모두와 작동한다는 것을 주목하여야 한다.

[0092] 도 21은 본 발명의 일 실시예에 따라 JIT 최적화를 통한 재정렬 프로세스의 다이어그램을 도시한다. 로드는 동일한 어드레스로 오는 다른 로드의 앞으로 디스패치될 수 없다. 본 발명의 일 실시예는 로드 퇴거 확장을 구현한다. 이 실시예에서, 다른 스레드 스토어는 전체 로드/스토어 큐 (및 확장)에 대해 체크한다.

[0093] 이러한 해결책의 구현 시, 퇴거된 모든 로드는 원래 Load C 위치의 지점까지 퇴거된 후 로드 큐(예를 들면, 또는 그의 확장)에 그대로 있어야 한다. 스토어가 다른 스레드(Thread 0)로부터 올 때, 스토어는 (예를 들어, 확장을 포함한) 전체 로드 큐를 CAN 매치할 것이다. 확장 크기는 재정렬된 로드(Load C)가 (예를 들면, 8 개의 엔트리 확장을 사용함으로써) 앞으로 점프할 수 있는 로드의 수에 제한을 둠으로써 결정될 수 있다. 이러한 해결책은 전체 및 부분 스토어 정렬 메모리 일관성 모델(예를 들면, x86 및 ARM 일관성 모델) 모두와 작동한다는 것을 주목하여야 한다.

[0094] 도 22는 본 발명의 일 실시예에 따라 JIT 최적화를 통한 스토어 앞에 재정렬된 로드를 예시하는 다이어그램을 도시한다. 도 22는 동일한 스레드 내에서 스토어 투 로드 포워딩 정렬(store to load forwarding ordering)(예를 들면, 스토어로부터 로드로의 데이터 의존관계(data dependency))을 이용한다.

[0095] 동일한 스레드 내에서 스토어의 동일한 어드레스로의 로드는 JIT를 통해 그 스토어 앞에 재정렬될 수 없다. 일 실시예에서, 퇴거한 모든 로드는 원래 Load C 위치의 지점까지 퇴거한 후 로드 큐 (및/또는 그의 확장)에 그대로 남아 있어야 한다. 재정렬된 각 로드는 다음에 오는 스토어와 관련하여 머신 순서(예를 들면, IP)에서 그 로드의 초기 위치를 표시할 옵셋을 포함할 것이다.

[0096] 일 구현 예는 초기 명령어 위치를 옵셋 표시자에 포함시키는 것이다. 스토어가 동일한 스레드로부터 올 때, 스토어는 (확장을 포함한) 전체 로드 큐를 CAM 매칭하여 이 스토어가 매칭된 로드로 포워딩할 것이라는 것을 표시하는 매치를 찾을 것이다. 스토어가 Load C 앞으로 디스패치되었던 경우, 그 스토어는 스토어 큐 내에 엔트리를 예약할 것이며, 로드가 나중에 디스패치되면, 로드는 스토어의 어드레스에 대해 CAM 매치할 것이고 또한 그의 IP를 사용하여 스토어 중 임의의 스토어로부터 그 로드로 데이터 포워딩을 결정짓는 머신 순서를 결정할 것이라는 것을 알아야 한다. 확장 크기는 재정렬된 로드(Load C)가 (예를 들면, 8개의 엔트리 확장을 사용함으로써) 앞으로 점프할 수 있는 로드의 수를 제한함으로써 결정될 수 있다.

[0097] 다른 해결책은 원래 로드 대신에 체크 스토어 명령어를 두는 것이다. 체크 스토어 명령어가 디스패치될 때, 이 명령어는 로드 큐에 대해 어드레스 매치를 체크한다. 유사하게, 로드가 디스패치될 때, 로드는 체크 스토어 명령어에 의해 점유된 스토어 큐 엔트리에 대해 어드레스 매치를 체크한다.

[0098] 도 23은 본 발명의 일 실시예에 따라 로드 및 스토어 명령어 분할(load and store instruction splitting)의 제 1 다이어그램을 도시한다. 본 발명의 하나의 특징은 로드가 두 개의 매크로명령어로 분할된다는 사실이며, 제 1 매크로명령어는 어드레스 계산 및 임시 위치(로드 스토어 큐)로의 페치를 수행하며, 제 2 매크로명령어는 메모리 어드레스 콘텐츠(데이터)의 레지스터 또는 ALU 목적지로의 로드이다. 본 발명의 실시예가 로드 및 스토어 명령어를 두 개의 각 매크로명령어로 분할하고 이들을 재정렬하는 맥락에서 기술되지만, 로드 및 스토어 명령어를 두 개의 각 매크로명령어로 분할하고 이들을 마이크로코드 컨텍스트 내에서 재정렬함으로써 동일한 방법

및 시스템이 구현될 수 있다는 것을 알아야 한다.

[0099] 스토어에 대해 기능성은 동일하다. 스토어는 또한 두 개의 매크로명령어로 분할된다. 제 1 명령어는 스토어 어드레스 및 페치이고, 제 2 명령어는 그 어드레스에 데이터의 스토어이다. 스토어의 분할 및 두 개의 명령어는 로드에 대해 아래에서 기술되는 것과 동일한 규칙을 따른다.

[0100] 로드를 두 개의 명령어로 분할하면 런타임 최적화기는 주어진 명령어 시퀀스 내에서 어드레스 계산 및 페치 명령어를 훨씬 일찍 스케줄할 수 있게 된다. 이렇게 하면 데이터를 캐시 계층과 별개인 임시 버퍼로 프리페치함으로써 메모리 미스로부터 일찍 복구가 가능해진다. 임시 버퍼는 LA/SA 및 LD/SD 사이에서 일대일 대응 관계로 프리페치된 데이터의 가용성을 보장하기 위해 사용된다. 로드 어드레스와 로드 데이터 사이의 윈도우에 있는 이전 스토어와 관련된 엘리어싱이 있는 경우(예를 들면, 포워딩 사례가 이전 스토어로부터 검출되는 경우), 또는 어드레스 계산과 관련된 임의의 결함 문제(예를 들면, 데이터 틀림(page fault)가 있는 경우, 대응하는 로드 데이터 명령어는 재발행(reissue)할 수 있다. 또한, 로드를 두 개의 명령어로 분할하면 정보를 두 개의 명령어로 복제하는 것을 또한 포함할 수 있다. 그러한 정보는 어드레스 정보, 소스 정보, 및 다른 부가 식별자 등일 수 있다. 이러한 복제로 말미암아 LA/SA가 없을 때 두 명령어의 LD/SD를 독립적으로 디스패치하는 것이 가능하다.

[0101] 로드 어드레스 및 페치 명령어는 로드 데이터가 다시 돌아오기를 기다리지 않고 실제 머신 퇴거 윈도우로부터 퇴거할 수 있고, 그럼으로써 그 어드레스(예를 들면, 단락의 시작부분에서 언급한 로드 어드레스)에 대해 캐시 미스가 있는 경우에도 머신은 포워딩을 진행시키게 할 수 있다. 예를 들면, 그 어드레스(예를 들면, address X)에 대해 캐시 미스가 생기면, 머신은 메모리 계층으로부터 데이터가 페치되기를 기다리는 수백 사이클 동안 아마도 정지될 수 있다. 데이터가 다시 돌아오기를 기다리지 않고 로드 어드레스 및 페치 명령어를 실제 머신 퇴거 윈도우로부터 퇴거시킴으로써, 머신은 포워딩을 계속 진행시킬 수 있다.

[0102] 명령어를 분할하면 본 발명의 실시예의 핵심적인 장점은 LA/SA 명령어를 명령어 시퀀스의 LD/SD로부터 일찍 그리고 더 멀리 재정렬할 수 있게 하여 로드 및 스토어를 더 일찍 디스패치하고 실행할 수 있게 한다는 것을 주목하여야 한다.

[0103] 도 24는 본 발명의 일 실시예에 따라 메모리 내에 저장된 코드 캐시 및 게스트 명령어 대 네이티브 명령어의 맵핑과 협력하여 CLB가 기능을 하는 방식을 예시하는 예시적인 흐름도를 도시한다.

[0104] 전술한 바와 같이, CLB는 코드 캐시 메모리 내에 저장된 대응하는 변환된 네이티브 어드레스를 갖는 게스트 어드레스의 맵핑(예를 들면, 게스트 대 네이티브 어드레스 맵핑)을 저장하는데 사용된다. 일 실시예에서, CLB는 게스트 어드레스의 일부를 가지고 인덱스된다. 게스트 어드레스는 인덱스, 태그, 및 옵셋(예를 들면, 청크 크기)으로 분할된다. 이 게스트 어드레스는 인덱스에 대응하는 CLB 엔트리 내 매치를 식별하는데 사용되는 태그를 포함한다. 태그에서 히트가 있으면, 대응하는 엔트리는 코드 캐시 메모리(806)에서 대응하는 변환된 네이티브 명령어 청크(예를 들면, 변환된 네이티브 명령어의 대응하는 블록)가 발견될 수 있는 곳을 표시하는 포인터를 저장할 것이다.

[0105] 본 명세서에서 사용된 것으로서 "청크"라는 용어는 변환된 네이티브 명령어 블록의 대응하는 메모리 크기를 말한다는 것을 알아야 한다. 예를 들면, 청크는 변환된 네이티브 명령어 블록의 여러 크기에 따라서 크기가 다를 수 있다.

[0106] 코드 캐시 메모리(806)와 관련하여, 일 실시예에서, 코드 캐시는 (예를 들어, 각 청크 유형마다 다른 크기를 갖는) 고정 크기의 청크 세트에 할당된다. 코드 캐시는 시스템 메모리 및 모든 하위 레벨 HW 캐시(예를 들면, 네이티브 하드웨어 캐시(608), 공유된 하드웨어 캐시(607))에서 세트(set) 및 웨이(way)로 논리적으로 분할될 수 있다. CLB는 게스트 어드레스를 사용하여 코드 캐시 청크에 대해 웨이 태그(way tag)를 인덱스하고 태그 비교할 수 있다.

[0107] 도 24는 게스트 어드레스 태그를, way x 및 way y로서 묘사된 2 웨이로 저장하는 CLB 하드웨어 캐시(804)를 도시한다. 일 실시예에서, CLB 구조를 사용한 게스트 어드레스 대 네이티브 어드레스의 맵핑은 구성된 웨이에서 (예를 들면, 게스트 대 네이티브 어드레스 맵핑으로부터) 네이티브 코드 청크로의 포인터를 저장함으로써 이루어질 수 있다는 것을 알아야 한다. 각 웨이는 태그와 연관된다. CLB는 (태그를 포함하는) 게스트 어드레스(802)로 인덱스된다. CLB에서 히트가 있으면, 태그에 대응하는 포인터가 반환된다. 이 포인터는 코드 캐시 메모리를 인덱스하는데 사용된다. 이것은 도 24에서, 코드 청크의 네이티브 어드레스가 포인터와 세그먼트 번호와의 함수라는 사실을 표현하는 라인 "코드 청크의 네이티브 어드레스=Seg#=+F(pt)"에 의해 표시된다. 본 실시예에서, 세그먼트는 포인터 스코프가 가상적으로 맵핑되는 (예를 들면, 포인터 어레이가 물리 어드레스 내

임의의 구역에 맵핑되게 하는) 메모리 내 포인트에 대한 베이스를 말한다.

[0108] 대안으로, 일 실시예에서, 코드 캐시 메모리는, 도 24에서 라인 "코드 청크의 네이티브 어드레스=seg#+Index*(청크의 크기)+way#*(청크 크기)"에 의해 표시된 바와 같은, 제 2 방법을 통해 인덱스될 수 있다. 이러한 실시예에서, 코드 캐시는 그의 웨이-구조가 CLB의 웨이와 코드 캐시 청크의 웨이 사이에서 1:1 매칭이 존재하도록 구성하는 CLB 웨이와 매칭하도록 조직된다. 특정 CLB 웨이에서 히트가 있을 때, 코드 캐시의 대응하는 웨이에서 대응하는 코드 청크는 네이티브 코드를 갖는다.

[0109] 계속 도 24를 참조하면, CLB의 인덱스가 미스이면, 메모리의 상위 계층(예를 들면, L1 캐시, L2 캐시 등)에서 히트가 있는지가 체크될 수 있다. 만일 이들 상위 캐시 레벨에서 히트가 없으면, 시스템 메모리(801) 내의 어드레스가 체크된다. 일 실시예에서, 게스트 인덱스는, 예를 들면, 64개의 청크를 포함하는 엔트리를 가리킨다. 64개 청크 각각의 태그가 읽혀지고 게스트 태그에 대비하여 비교되어 히트가 있는지를 결정한다. 이와 같은 프로세스는 도 24에서 점선 박스(805)로 도시된다. 시스템 메모리에서 태그와 비교 이후 히트가 없으면, 메모리의 어느 계층 레벨에서도 변환이 존재하지 않으며, 게스트 명령어는 변환되어야 한다.

[0110] 본 발명의 실시예는 게스트 대 네이티브 명령어 맵핑을 캐시와 유사한 방식으로 저장하는 각 계층 레벨의 메모리를 관리한다는 것을 알아야 한다. 이것은 본질적으로 캐시 기반의 메모리(예를 들면, CLB 하드웨어 캐시, 네이티브 캐시, L1 및 L2 캐시 등)로부터 비롯된다. 그러나 CLB는 또한 시스템 메모리(801) 내에서 게스트 대 네이티브 명령어 맵핑을 위한 최근 최소 사용(least recently used, LRU) 대체 관리 정책을 구현하는데 사용되는 "코드 캐시 + CLB 관리 비트"를 포함한다. 일 실시예에서, CLB 관리 비트(예를 들면, LRU 비트)는 소프트웨어 관리된다. 이러한 방식으로, 모든 계층 레벨이 메모리는 최근 최대로 사용된 가장 빈번히 접하는 게스트 대 네이티브 명령어 맵핑을 저장하는데 사용된다. 따라서, 이것은 모든 계층 레벨의 메모리가 가장 빈번히 접하는 변환된 네이티브 명령어를 유사하게 저장하게 한다.

[0111] 도 24는 또한 CLB에 저장된 동적 분기 바이어스 비트 및/또는 분기 이력 비트를 도시한다. 이들 동적 분기 비트는 게스트 명령어 시퀀스를 조립할 때 사용되는 분기 예측의 동작을 추적하는데 사용된다. 이들 비트는 어느 분기 예측이 가장 자주 정확히 예측되는지 그리고 어느 분기 예측이 가장 자주 부정확하게 예측되는지를 추적하는데 사용된다. CLB는 또한 변환된 블록 범위에 필요한 데이터를 저장하기도 한다. 이 데이터는 프로세서로 하여금 (예를 들면, 자체 수정 코드에서와 같이) 대응하는 게스트 명령어가 수정되었던 코드 캐시 메모리에서 변환된 블록 범위를 무효화할 수 있게 한다.

[0112] 도 25는 본 발명의 일 실시예에 따른 런 어헤드 런 타임 게스트 명령어 변환/디코딩 프로세스(run ahead run time guest instruction conversion/decoding process)의 다이어그램을 도시한다. 도 25는 게스트 코드의 온-디맨드 변환/디코딩하는 동안, 메인 메모리로부터 게스트 코드를 가져오는 것(예를 들면, 이것은 비용이 드는 동작일 수 있음)을 회피하려는 목적을 도시하는 다이어그램을 예시한다. 도 25는 게스트 코드가 명령어 시퀀스에서 게스트 분기의 타겟으로부터 프리페치되는 프리페칭 프로세스를 도시한다. 예를 들면, 명령어 시퀀스는 추측 분기 X, Y, 및 Z를 포함한다. 이것은 어드레스 X, Y, 및 Z에서 게스트 코드의 프리페치 명령어의 발행을 유발한다.

[0113] 도 26은 본 발명의 일 실시예에 따른 게스트 명령어 시퀀스를 갖는 변환 테이블 및 네이티브 명령어 맵핑을 갖는 네이티브 맵핑 테이블을 묘사하는 다이어그램을 도시한다. 일 실시예에서, 메모리 구조/테이블은 저 레벨의 저 지연 캐시와 유사한 캐시로서 구현될 수 있다.

[0114] 일 실시예에서, 가장 빈번히 접하는 게스트 명령어 및 이들의 맵핑은 저 레벨 캐시 구조에 저장되어서, 런타임이 이들 구조에 빠르게 액세스하여 게스트 명령어에 동등한 네이티브 명령어를 가져오게 할 수 있다. 맵핑 테이블은 검색된 게스트 명령어 포맷에 동등한 명령어 포맷을 제공할 것이다. 또한 일부 제어 값을 사용하면 제어 필드로서 이들 맵핑 테이블에 저장하여 게스트 명령어의 특정 필드를 네이티브 명령어의 동등한 필드로 신속하게 대체할 수 있다. 여기서의 아이디어는 빈번하지 않은 다른 게스트 명령어가 변환하는데 시간이 더 오래 걸릴 수 있지만, 변환을 신속히 할 수 있도록 가장 빈번히 접하는 게스트 명령어만을 저 레벨 (예를 들면, 캐시)에 저장하는 것이다.

[0115] 이제 본 발명의 실시예에 따른 CLB/CLBV/CLT라는 용어가 논의된다. 일 실시예에서, CLB는 게스트 분기의 목적지에 맵핑하는 코드의 어드레스를 얻기 위해 네이티브 코드를 실행하는 동안 네이티브 게스트 분기를 만날 때 검색되는 메모리 구조로서 유지되는 변환 룩 어사이드 버퍼이다. 일 실시예에서, CLBV는 CLB의 희생 캐시 이미지(victim cache image)이다. 엔트리가 CLB로부터 축출될 때, 이들 엔트리는 정규 L1/L2 캐시 구조에서 캐싱된다.

CLB가 미스를 만나면, CLB는 미스의 타겟을 찾기 위해 하드웨어 액세스에 의해 L1/L2를 자동 검색할 것이다. 일 실시예에서, CLT는 미스의 타겟이 CLB 또는 CLBV에서 발견되지 않을 때 사용되며, 메인 메모리 내 CLT 테이블에 있는 엔트리를 검색하기 위해 소프트웨어 핸들러가 트리거된다.

[0116] 이제 본 발명의 실시예에 따른 CLB 카운터가 논의된다. 일 실시예에서, CLB 카운터는 변환 시간에 설정되어 변환된 명령어 시퀀스/트레이스에 관련된 메타데이터와 함께 저장되는 값이다. 이 카운터는 명령어 시퀀스/트레이스가 실행될 때마다 감분되며 활발한 작동을 위한 트리거로서 역할을 한다. 이 값은 모든 CLB 레벨(예를 들면, CLB, CLBV, CLT)에 저장된다. 이 값이 문턱치에 이를 때, 이 값은 명령어 시퀀스/트레이스를 최적화하기 위해 JIT 컴파일러를 트리거한다. 이 값은 하드웨어에 의해 유지되고 관리된다. 일 실시예에서, 명령어 시퀀스/트레이스는 CLB 카운터와 소프트웨어 카운터의 복합체를 가질 수 있다.

[0117] 이제 본 발명의 일 실시예에 따른 백그라운드 스레드(background thread)가 논의된다. 일 실시예에서, 활발한 작동이 트리거되면, 하드웨어 백그라운드 스레드가 개시되어 소프트웨어에 보이지 않는 백그라운드 하드웨어 작업으로서 역할을 하며 그 자체 하드웨어 자원, 통상적으로는 최소의 자원(예를 들면, 소수의 레지스터 파일 및 시스템 상태)을 갖는다. 백그라운드 스레드는 우선순위가 낮고 실행 자원이 이용 가능할 때 실행 자원을 저장하는 백그라운드 스레드로서 실행을 지속한다. 백그라운드 스레드는 하드웨어 스레드 ID를 갖고 소프트웨어에 보이지 않지만 저 레벨 하드웨어 관리 시스템에 의해 관리된다.

[0118] 이젠 본 발명의 일 실시예에 따른 JIT 프로파일링 및 런타임 모니터링/동적 체킹(JIT profiling and runtime monitoring/dynamically checking)이 논의된다. JIT는 시간 간격으로 프로파일링/모니터링/스위핑(sweeping) 명령어 시퀀스/트레이스를 시작할 수 있다. JIT는 이를테면 분기 프로파일링을 사용함으로써 최적화와 관련된 특정 값을 유지할 수 있다. 분기 프로파일링은 코드 인스트루먼트(code instrumentation)와 함께 분기 프로파일링 하드웨어 명령어를 사용하여, 이것이 특정 어드레스로부터 명령어를 페치하기 시작하여 이들 명령어를 머신 프론트 엔드를 통과시키도록 분기의 시맨틱을 갖는 명령어를 구현함으로써 그리고 이들 명령어를 실행하지 않고 하드웨어 분기 예측자를 검색함으로써 명령어 시퀀스/트레이스 내에서 분기에 대한 분기 예측 값/바이어스를 찾을 수 있다. 그런 다음 JIT는 이들 하드웨어 분기 예측 카운터의 값을 누적하여 하드웨어가 제공하는 것보다 큰 카운터를 생성한다. 이로써 JIT는 분기 바이어스를 프로파일링할 수 있게 된다.

[0119] 상시 프로파일링(constant profiling)은 변경되지 않는 값을 검출하고 이 정보를 사용하여 코드를 최적화하도록 프로파일링하는 것을 말한다.

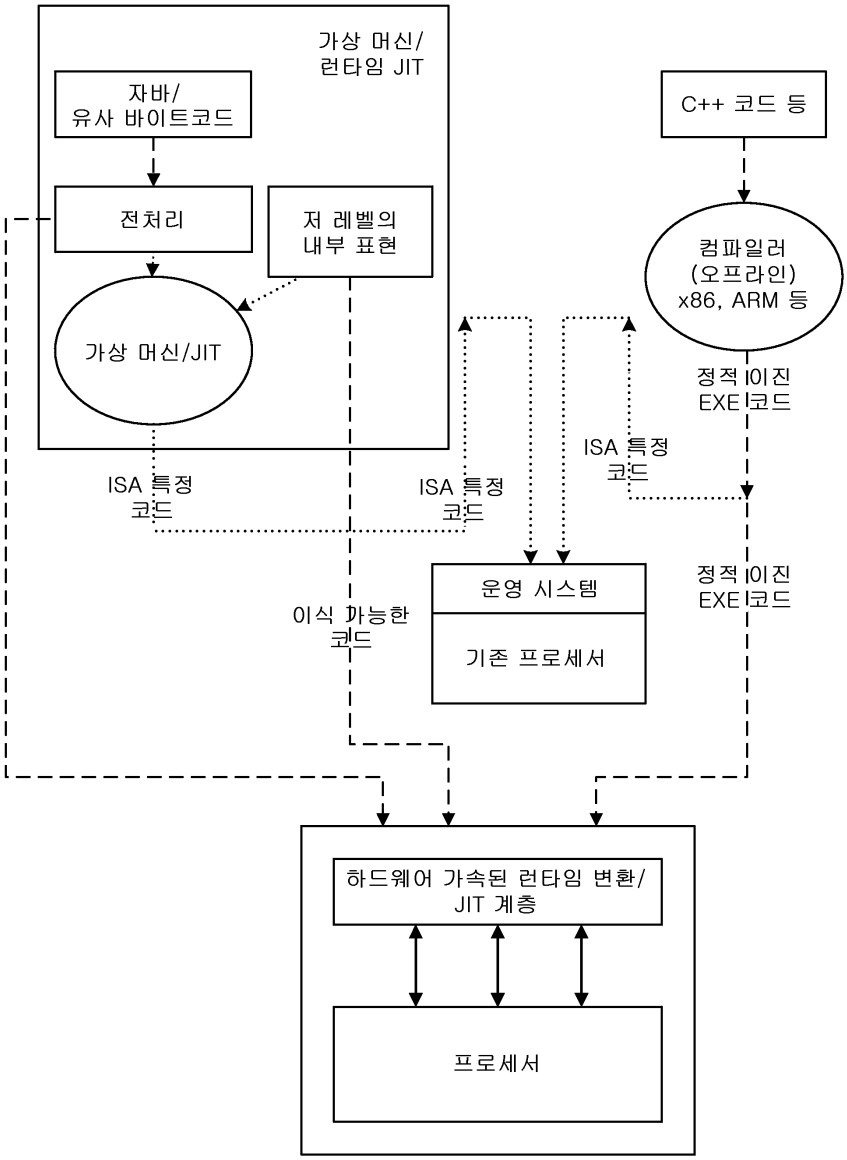
[0120] 로드 스토어 엘리어싱의 체킹(checking for Load store aliasing)은 이것이 로드와 스토어 사이의 어드레스 엘리어싱을 동적으로 체크함으로서 때때로 스토어 투 로드 포워딩이 발생하지 않는 것을 체크하는 것이 가능하기 때문에 사용된다.

[0121] 일 실시예에서, JIT는 코드를 인스트루먼트할 수 있거나 분기 프로파일링 명령어 또는 체크 로드 명령어 또는 체크 스토어 명령어와 같은 특수한 명령어를 사용할 수 있다.

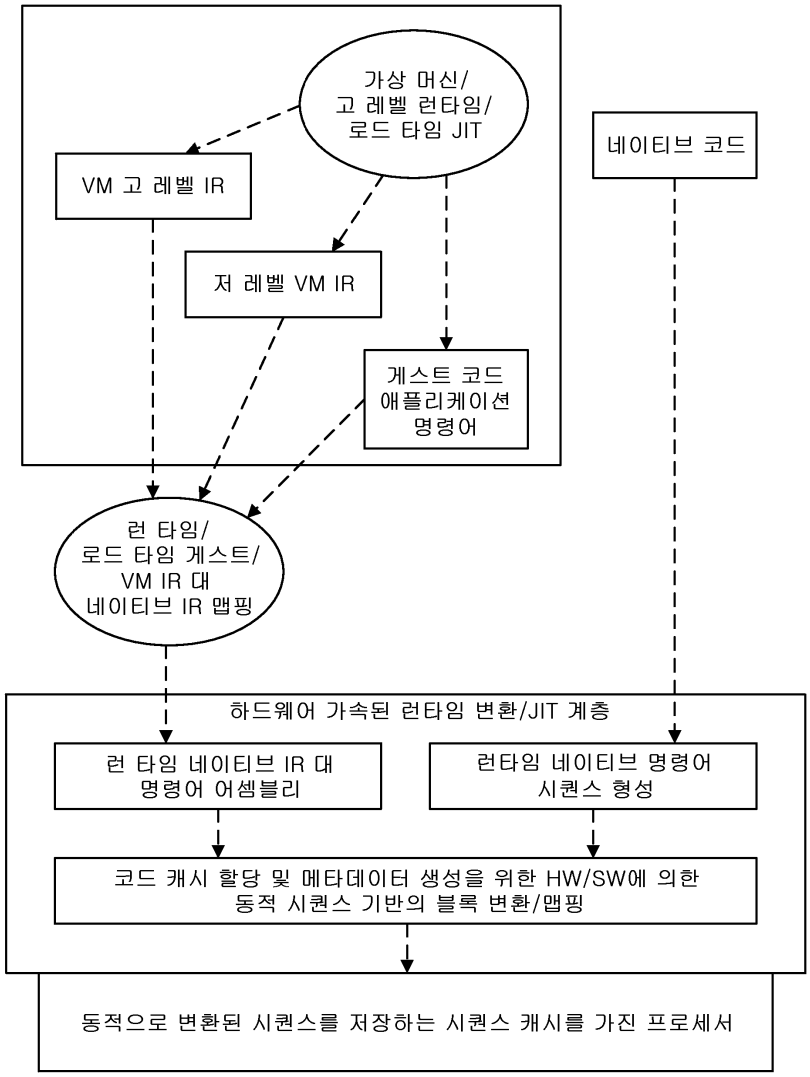
[0122] 설명 목적 상, 전술한 설명은 본 발명을 총망라하거나 제한하려는 의도가 아닌 특정 실시예를 언급한다. 전술한 가르침과 일관하는 많은 수정과 변형이 가능하다. 실시예는 본 발명의 원리 및 그의 실제 응용을 가장 잘 설명하기 위해 선택되고 기술되었으며, 그리하여 본 기술에서 통상의 지식을 가진 자가 본 발명과 본 발명의 다양한 실시예를 그 특정 용도에 적합해질 수 있는 다양한 변형으로 가장 잘 활용할 수 있게 한다.

도면

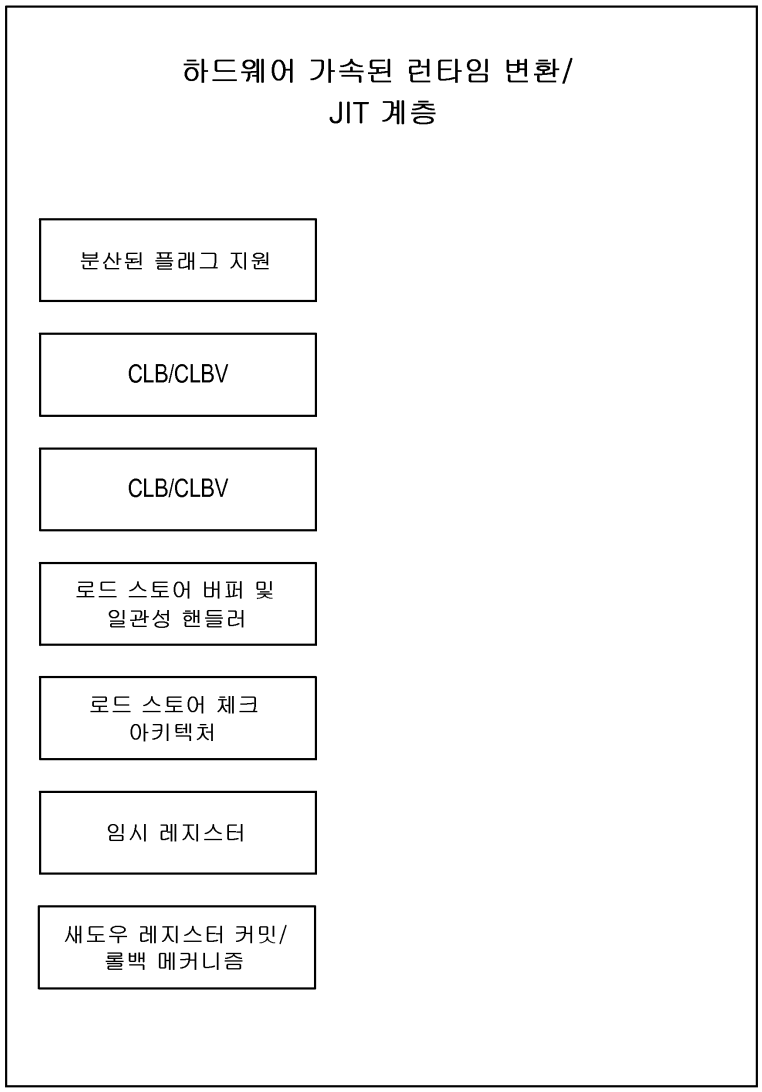
도면1


가상 머신/
런타임 JIT
자바/
유사 바이트코드
전처리
저 레벨의
내부 표현
가상 머신/JIT
C++ 코드 등
컴파일러
(오프라인)
x86, ARM 등
정적 이진
EXE 코드
ISA 특정
코드
ISA 특정
코드
ISA 특정
코드
운영 시스템
기존 프로세서
정적 이진
EXE 코드
이식 가능한
코드
하드웨어 가속된 런타임 변환/
JIT 계층
프로세서

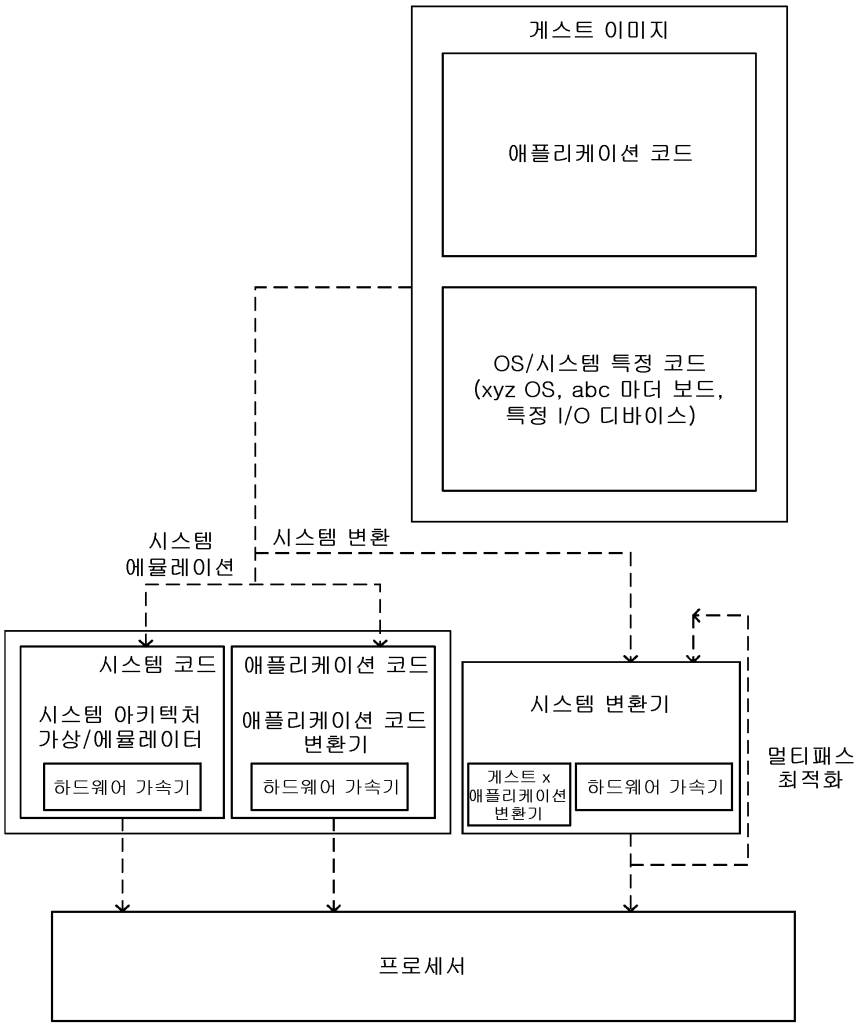
도면2


가상 머신/
고 레벨 런타임/
로드 타임 JIT
네이티브 코드
VM 고 레벨 IR
저 레벨 VM IR
게스트 코드
애플리케이션
명령어
런 타임/
로드 타임 게스트/
VM IR 대
네이티브 IR 맵핑
하드웨어 가속된 런타임 변환/JIT 계층
런 타임 네이티브 IR 대
명령어 어셈블리
런타임 네이티브 명령어
시퀀스 형성
코드 캐시 할당 및 메타데이터 생성을 위한 HW/SW에 의한
동적 시퀀스 기반의 블록 변환/맵핑
동적으로 변환된 시퀀스를 저장하는 시퀀스 캐시를 가진 프로세서

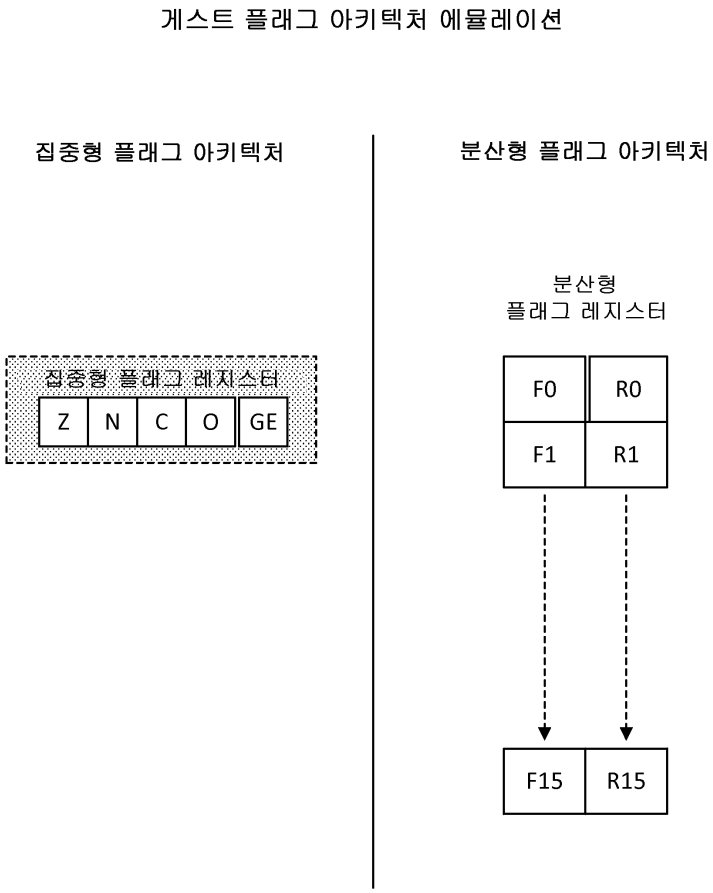
도면3


하드웨어 가속된 런타임 변환/
JIT 계층
분산된 플래그 지원
CLB/CLBV
CLB/CLBV
로드 스토어 버퍼 및
일관성 핸들러
로드 스토어 체크
아키텍처
임시 레지스터
새도우 레지스터 커밋/
롤백 메커니즘

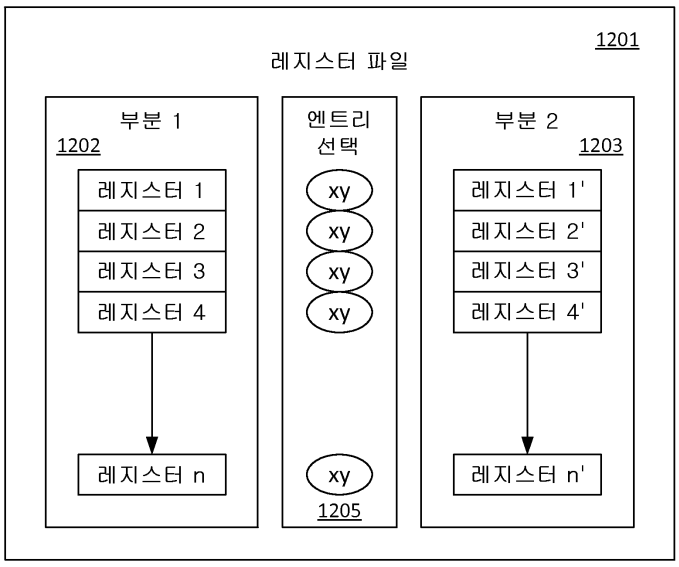
도면4


게스트 이미지
애플리케이션 코드
OS/시스템 특정 코드
(xyz OS, abc 마더 보드,
특정 I/O 디바이스)
시스템
에뮬레이션
시스템 변환
시스템 코드
시스템 아키텍처
가상/에뮬레이터
하드웨어 가속기
애플리케이션 코드
애플리케이션 코드
변환기
하드웨어 가속기
시스템 변환기
게스트 x
애플리케이션
변환기
하드웨어 가속기
멀티패스
최적화
프로세서

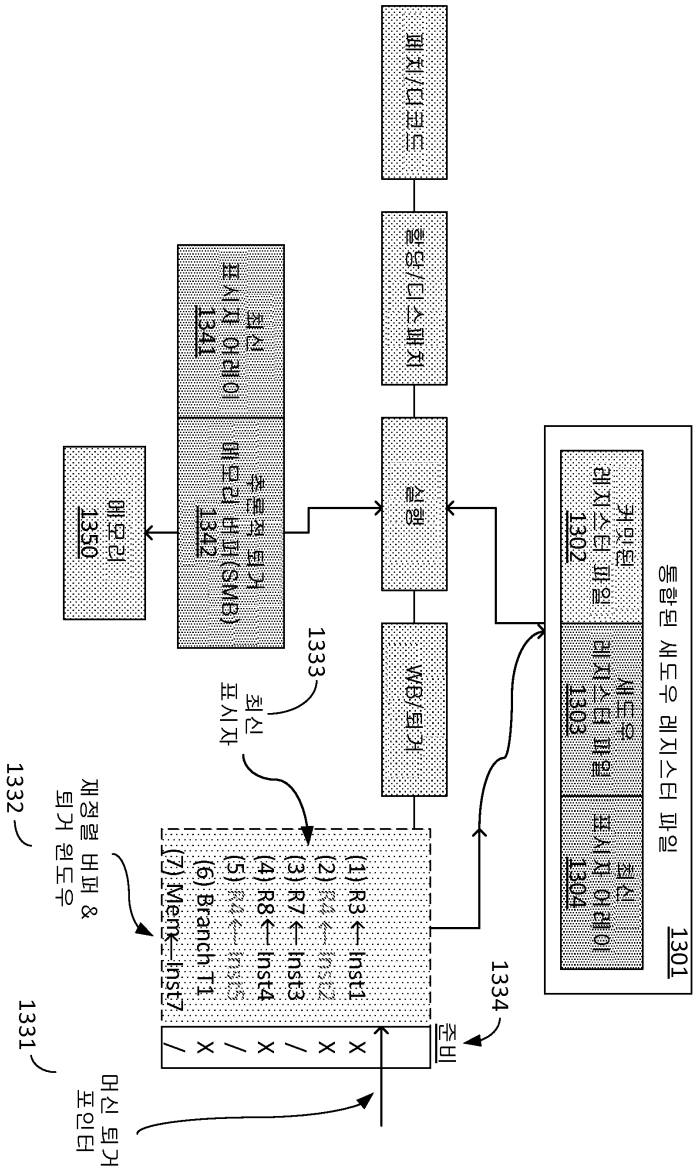
도면5


게스트 플래그 아키텍처 에뮬레이션
집중형 플래그 아키텍처
분산형 플래그 아키텍처
집중형 플래그 레지스터
Z
N
C
O
GE
분산형
플래그 레지스터
F0
R0
F1
R1
F15
R15

도면6


1201
레지스터 파일
부분 1
1202
레지스터 1
레지스터 2
레지스터 3
레지스터 4
레지스터 n
엔트리
선택
xy
xy
xy
xy
xy
1205
부분 2
1203
레지스터 1'
레지스터 2'
레지스터 3'
레지스터 4'
레지스터 n'

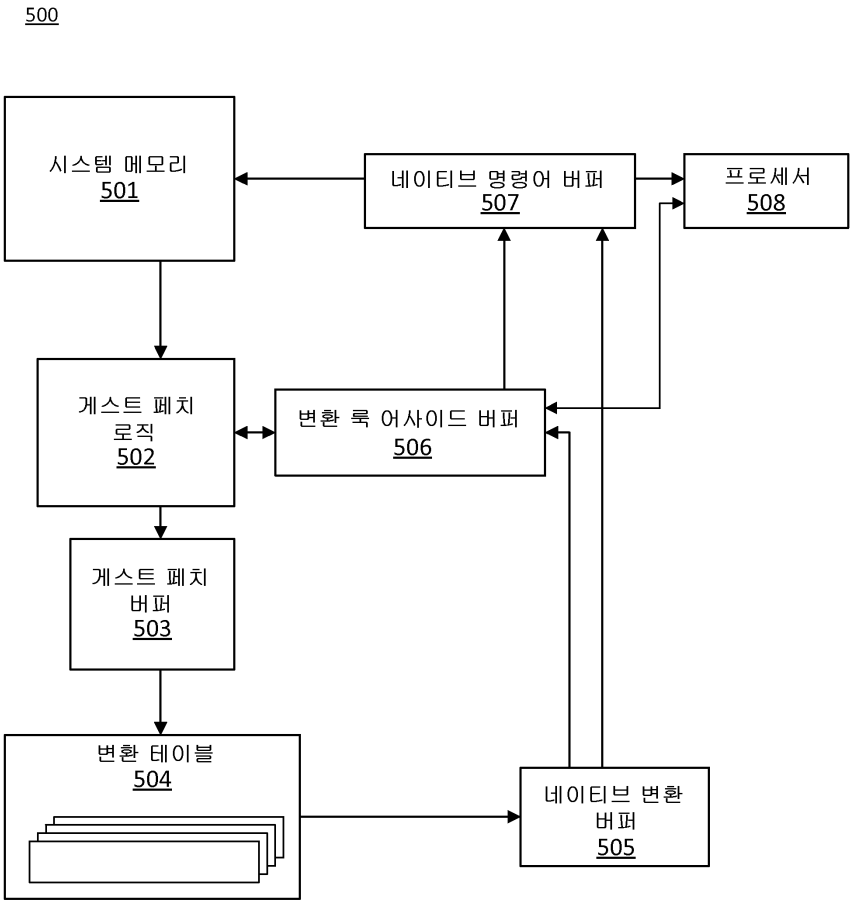
도면7


페치/디코드
할당/디스패치
실행
WB/퇴거
통합된 재도우 레지스터 파일 1301
커밋된 레지스터 파일 1302
재도우 레지스터 파일 1303
최신 표시자 어레이 1304
최신 표시자 어레이 1341
추론적 퇴거 메모리 버퍼(SMB) 1342
메모리 1350
1333
최신 표시자
1332
재정렬 버퍼 & 퇴거 윈도우
1331
머신 퇴거 포인터
1334
준비
(1) R3 ← Inst1
(2) R4 ← Inst2
(3) R7 ← Inst3
(4) R8 ← Inst4
(5) R4 ← Inst5
(6) Branch T1
(7) Mem← Inst7

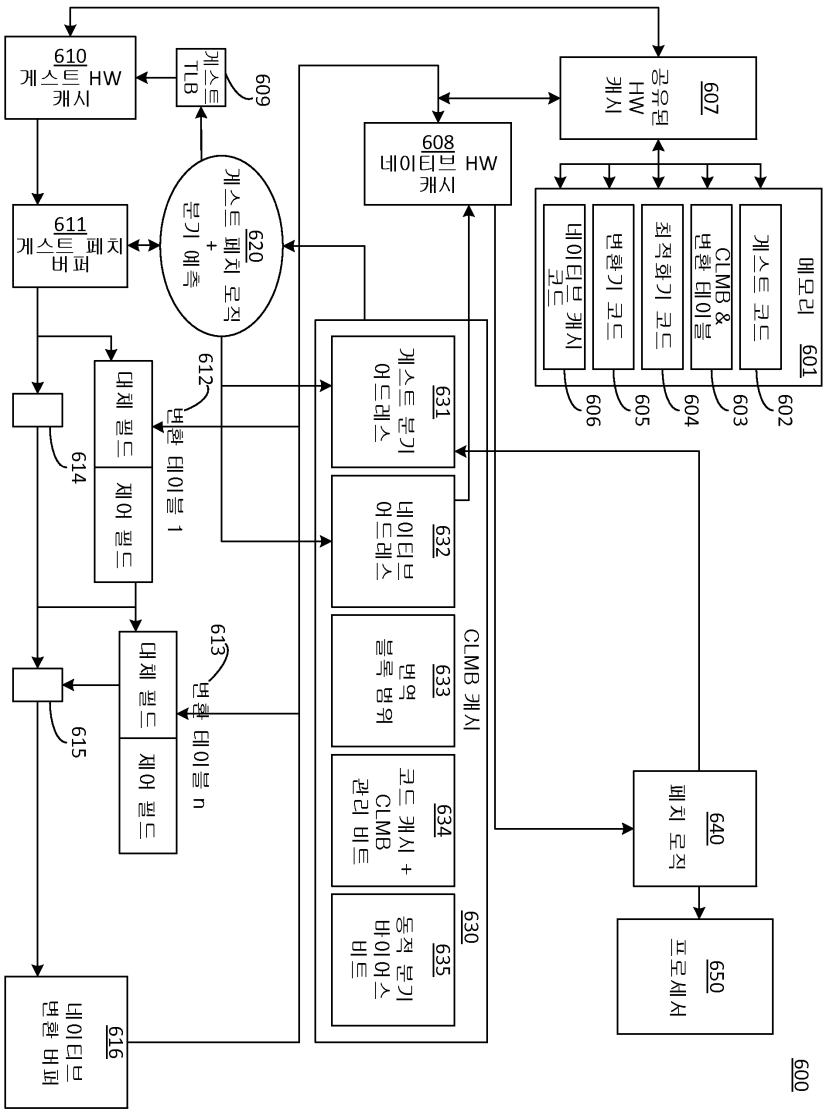
도면8

런 어헤드 페치/변환
게스트 어드레스
5000
6000
8000
점프
변환
게스트 분기 8000
게스트 분기 6000
코드 캐시 내 네이티브 어드레스
네이티브 분기 g(8000)
네이티브 분기 g(6000)
CLB
게스트
네이티브
5000
X
6000
? Y
8000
? Z
네이티브 코드 캐시
X
{
Y
{
Z
{
6000
8000

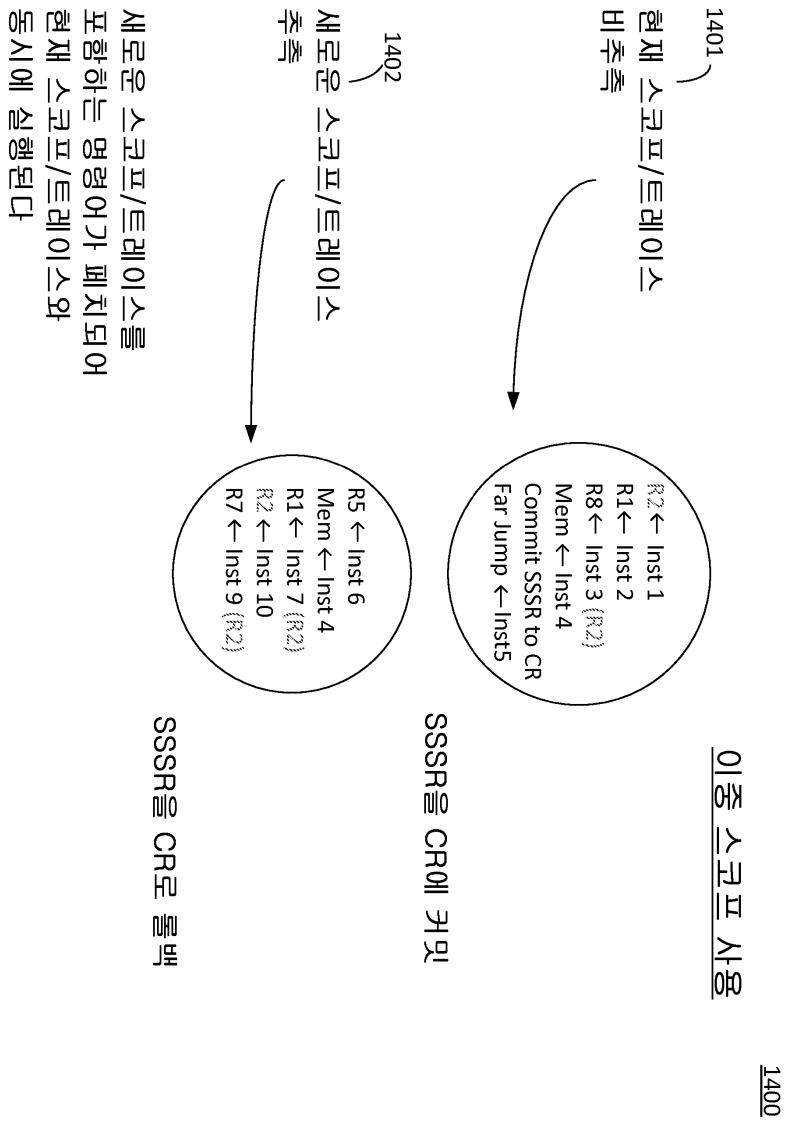
도면9


500
시스템 메모리
501
네이티브 명령어 버퍼
507
프로세서
508
게스트 페치
로직
502
변환 룩 어사이드 버퍼
506
게스트 페치
버퍼
503
변환 테이블
504
네이티브 변환
버퍼
505

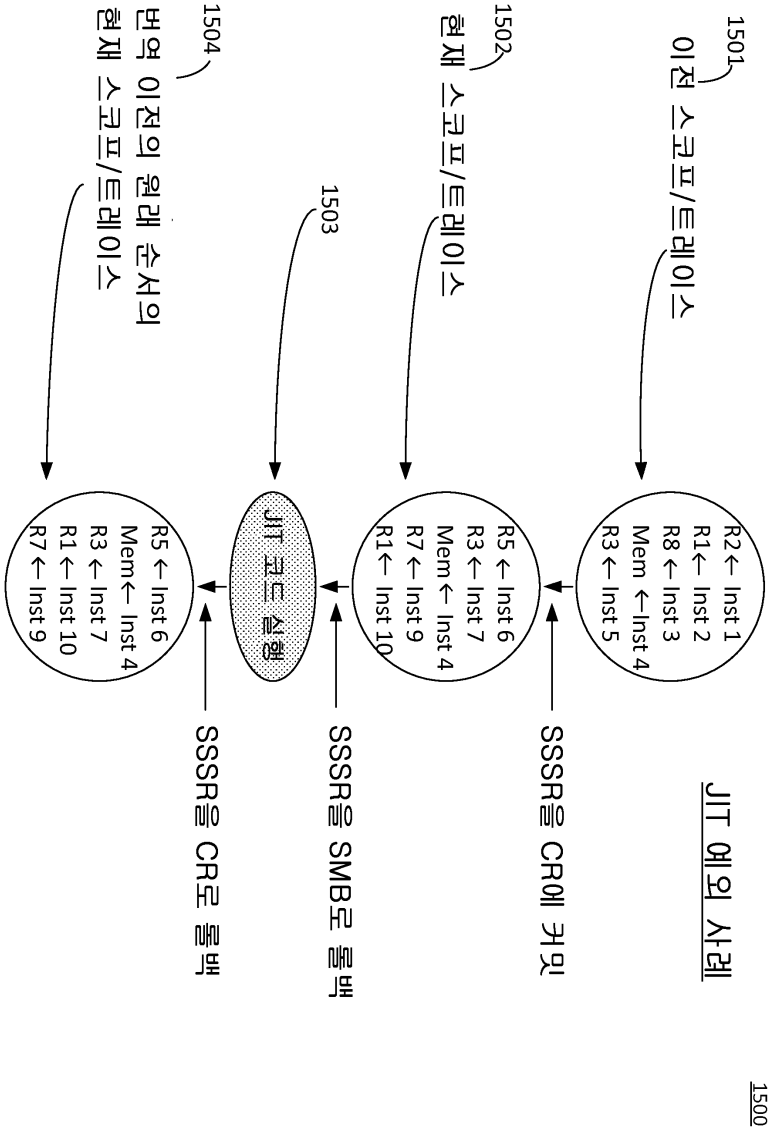
도면10


610
게스트 HW 캐시
게스트 TLB
609
611
게스트 페치 버퍼
게스트 페지 로직 + 분기 예측
620
607
공유된 HW 캐시
608
네이티브 HW 캐시
메모리 601
네이티브 코드
606
변환기 코드
605
최적화기 코드
604
CLMB & 변환 테이블
603
게스트 코드
602
612
변환 테이블 1
대체 필드
제어 필드
614
613
변환 테이블 n
대체 필드
제어 필드
615
CLMB 캐시
630
631
게스트 분기 어드레스
632
네이티브 어드레스
633
번역 블록 범위
634
코드 캐시 + CLMB 관리 비트
635
동적 분기 바이어스 비트
640
페치 로직
650
프로세서
616
네이티브 변환 버퍼
600

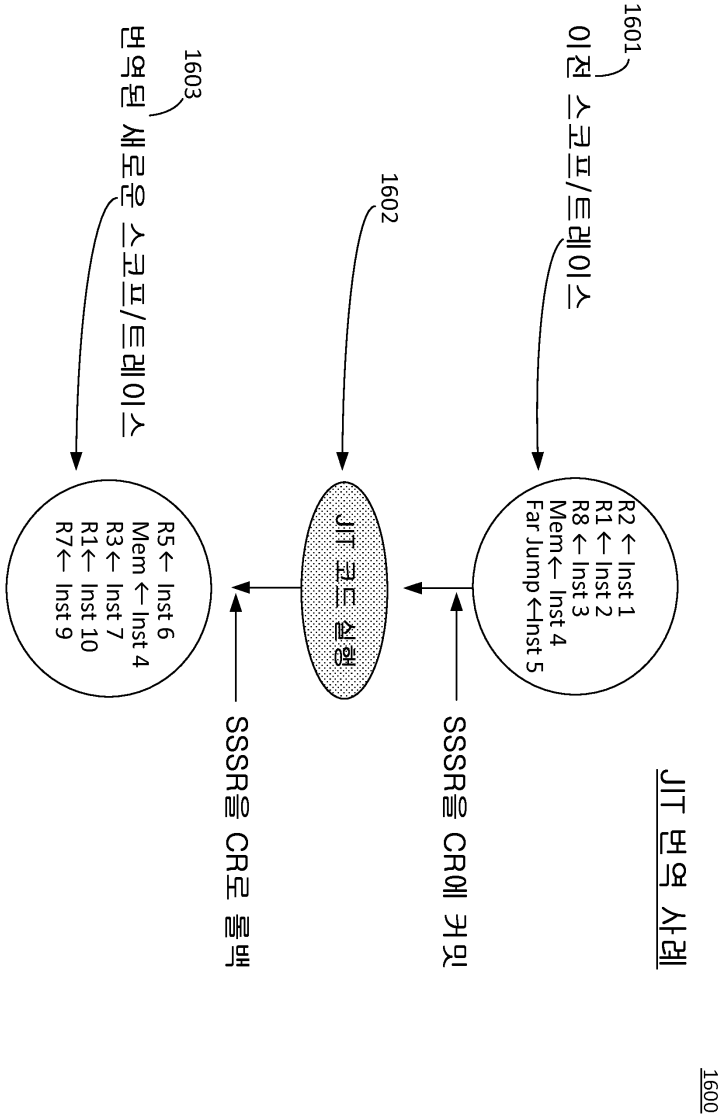
도면11


1400
이중 스코프 사용
1401
현재 스코프/트레이스
비추측
R2 ← Inst 1
R1← Inst 2
R8← Inst 3 {R2}
Mem ← Inst 4
Commit SSSR to CR
Far Jump ← Inst5
SSSR을 CR에 커밋
1402
새로운 스코프/트레이스
추측
R5 ← Inst 6
Mem ← Inst 4
R1← Inst 7 {R2}
R2 ← Inst 10
R7 ← Inst 9 {R2}
SSSR을 CR로 롤백
새로운 스코프/트레이스를
포함하는 명령어가 페치되어
현재 스코프/트레이스와
동시에 실행된다

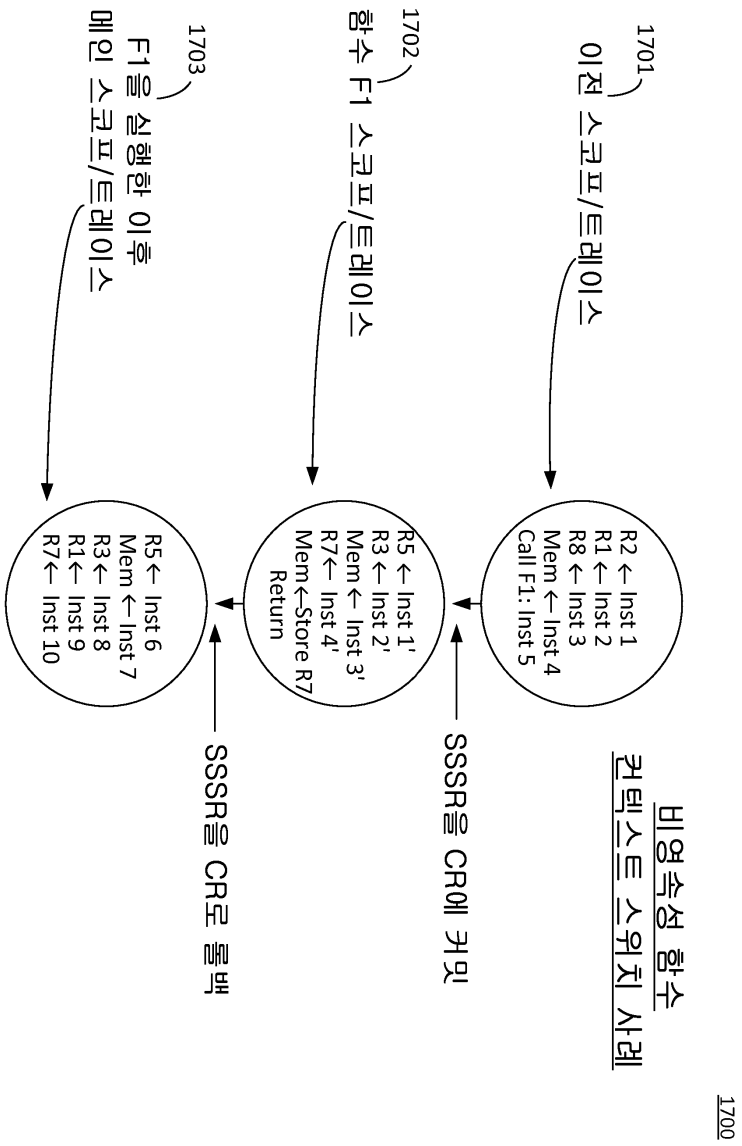
도면12


1500
JIT 예외 사례
1501
이전 스코프/트레이스
R2← Inst 1
R1← Inst 2
R8 ← Inst 3
Mem ←Inst 4
R3 ← Inst 5
SSSR을 CR에 커밋
1502
현재 스코프/트레이스
R5 ← Inst 6
R3 ← Inst 7
Mem ← Inst 4
R7 ← Inst 9
R1← Inst 10
SSSR을 SMB로 롤백
1503
JIT 코드 실행
SSSR을 CR로 롤백
1504
번역 이전의 원래 순서의
현재 스코프/트레이스
R5 ← Inst 6
Mem ← Inst 4
R3 ← Inst 7
R1 ← Inst 10
R7 ← Inst 9

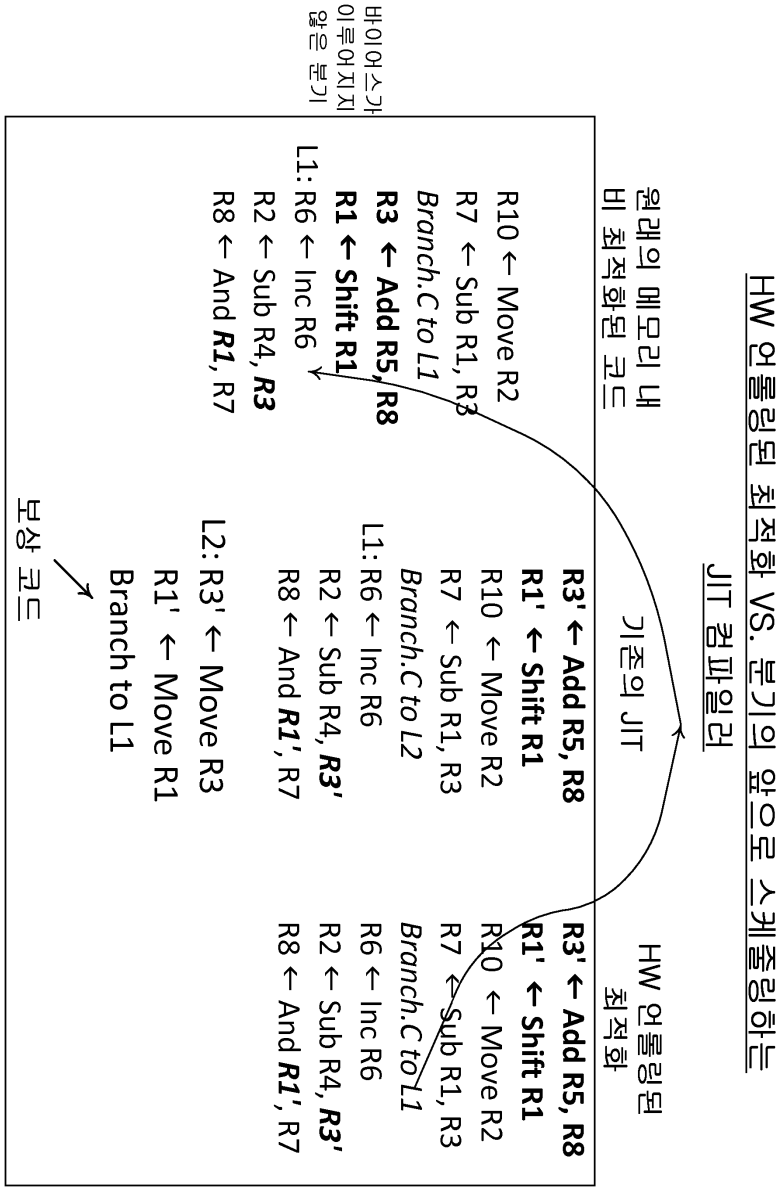
도면13


JIT 번역 사례
1600
SSSR을 CR에 커밋
SSSR을 CR로 롤백
1601
이전 스코프/트레이스
R2 ← Inst 1
R1 ← Inst 2
R8 ← Inst 3
Mem← Inst 4
Far Jump ←Inst 5
1602
JIT 코드 실행
1603
번역된 새로운 스코프/트레이스
R5← Inst 6
Mem ← Inst 4
R3← Inst 7
R1← Inst 10
R7← Inst 9

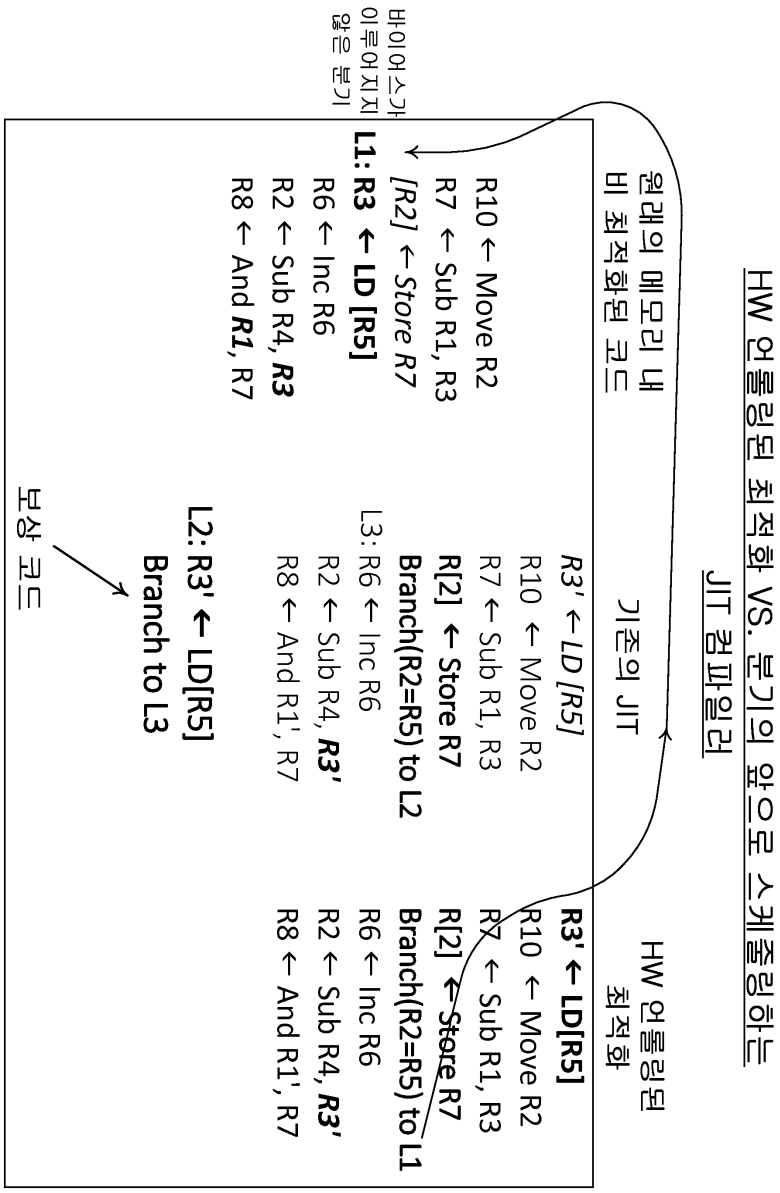
도면14


1700
비영속성 함수
컨텍스트 스위치 사례
1701
이전 스코프/트레이스
R2 ← Inst 1
R1 ← Inst 2
R8 ← Inst 3
Mem ← Inst 4
Call F1; Inst 5
SSSR을 CR에 커밋
1702
함수 F1 스코프/트레이스
R5 ← Inst 1'
R3 ← Inst 2'
Mem ← Inst 3'
R7← Inst 4'
Mem←Store R7
Return
SSSR을 CR로 롤백
1703
F1을 실행한 이후
메인 스코프/트레이스
R5 ← Inst 6
Mem ← Inst 7
R3 ← Inst 8
R1 ← Inst 9
R7 ← Inst 10

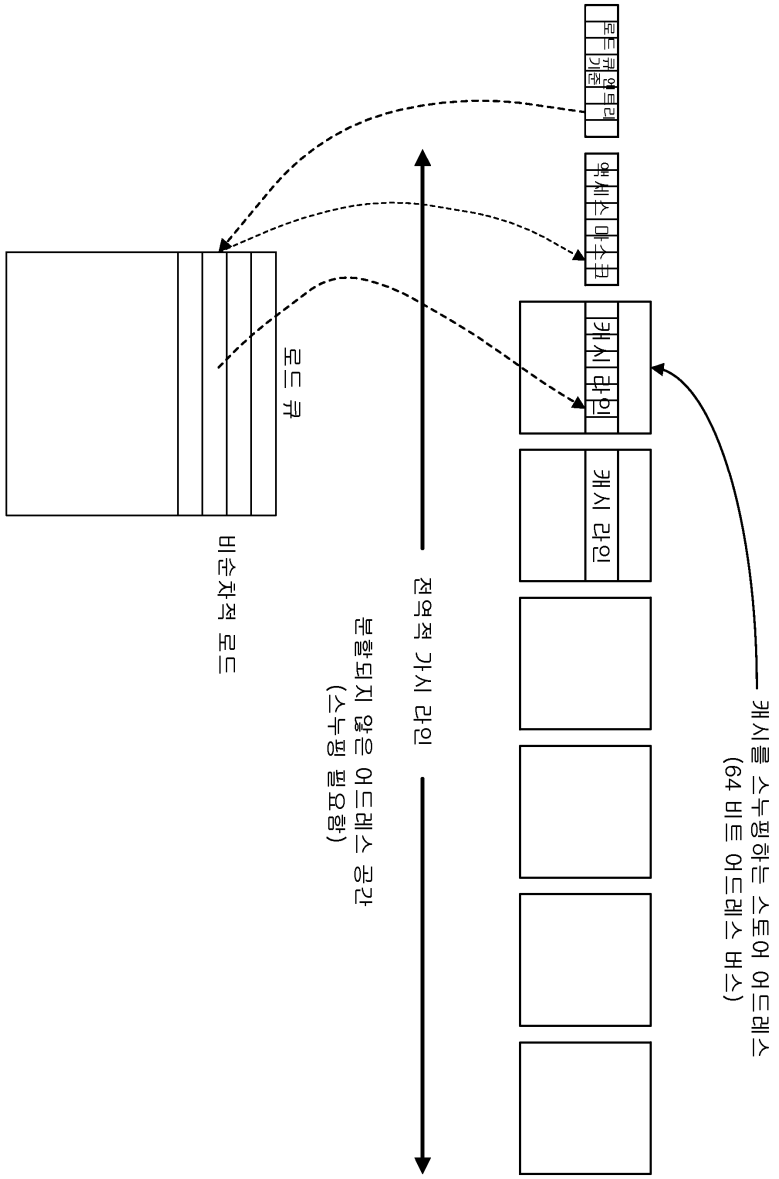
도면15


HW 언롤링된 최적화 VS. 분기의 앞으로 스케줄링하는 JIT 컴파일러
원래의 메모리 내 비 최적화된 코드
기존의 JIT
HW 언롤링된 최적화
바이어스가 이루어지지 않은 분기
R10 ← Move R2
R7 ← Sub R1, R3
Branch.C to L1
R3 ← Add R5, R8
R1 ← Shift R1
L1: R6 ← Inc R6
R2 ← Sub R4, R3
R8 ← And R1, R7
L2: R3' ← Move R3
R1' ← Move R1
Branch to L1
보상 코드
R3' ← Add R5, R8
R1' ← Shift R1
R10 ← Move R2
R7 ← Sub R1, R3
Branch.C to L1
R6 ← Inc R6
R2 ← Sub R4, R3'
R8 ← And R1', R7

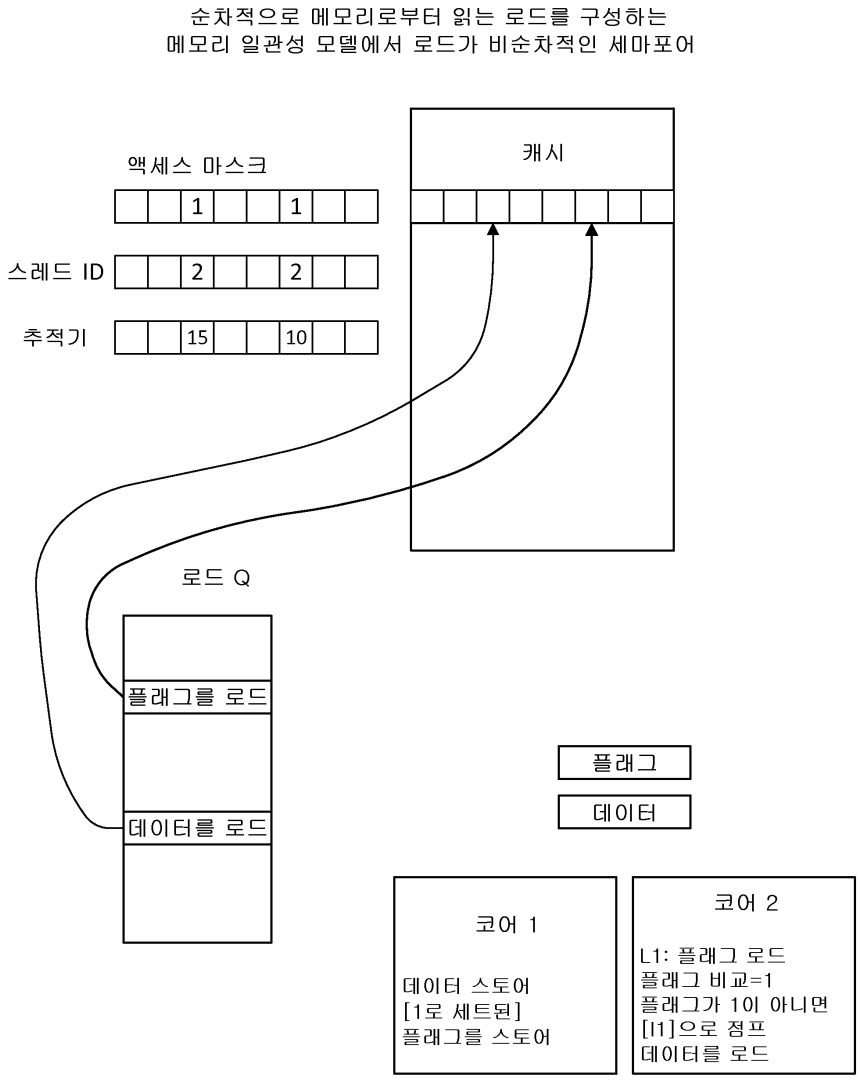
도면16


HW 언롤링된 최적화 VS. 분기의 앞으로 스케줄링하는 JIT 컴파일러
원래의 메모리 내 비 최적화된 코드
기존의 JIT
HW 언롤링된 최적화
바이어스가 이루어지지 않은 분기
R10 ← Move R2
R7 ← Sub R1, R3
[R2] ← Store R7
L1: R3 ← LD [R5]
R6 ← Inc R6
R2 ← Sub R4, R3
R8 ← And R1, R7
L2: R3' ← LD[R5]
Branch to L3
보상 코드
R3' ← LD [R5]
R10 ← Move R2
R7 ← Sub R1, R3
R[2] ← Store R7
Branch(R2=R5) to L2
L3: R6 ← Inc R6
R2 ← Sub R4, R3'
R8 ← And R1', R7
R3' ← LD[R5]
R10 ← Move R2
R7 ← Sub R1, R3
R[2] ← Store R7
Branch(R2=R5) to L1
R6 ← Inc R6
R2 ← Sub R4, R3'
R8 ← And R1', R7

도면17

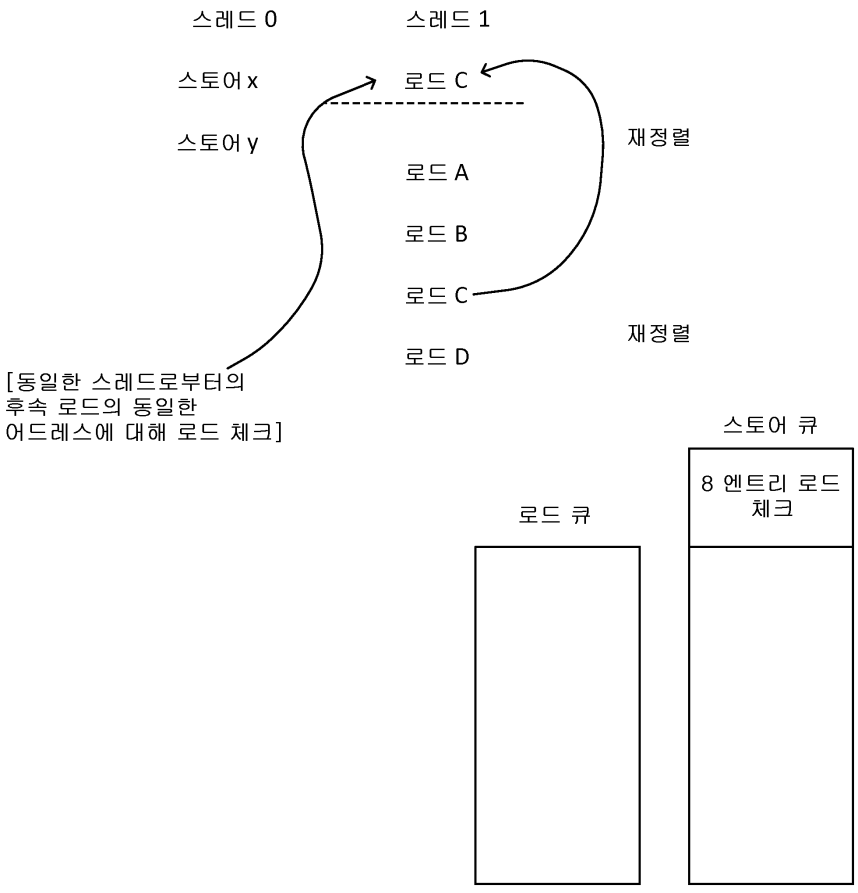

로드 큐 엔트리
액세스 마스크
캐시 라인
캐시 라인
캐시를 스누핑하는 스토어 어드레스
(64 비트 어드레스 버스)
전역적 가시 라인
분할되지 않은 어드레스 공간
(스누핑 필요함)
로드 큐
비순차적 로드

도면18


순차적으로 메모리로부터 읽는 로드를 구성하는
메모리 일관성 모델에서 로드가 비순차적인 세마포어
캐시
액세스 마스크
1
1
스레드 ID
2
2
추적기
15
10
로드 Q
플래그를 로드
데이터를 로드
플래그
데이터
코어 1
데이터 스토어
[1로 세트된]
플래그를 스토어
코어 2
L1: 플래그 로드
플래그 비교=1
플래그가 1이 아니면
[l1]으로 점프
데이터를 로드

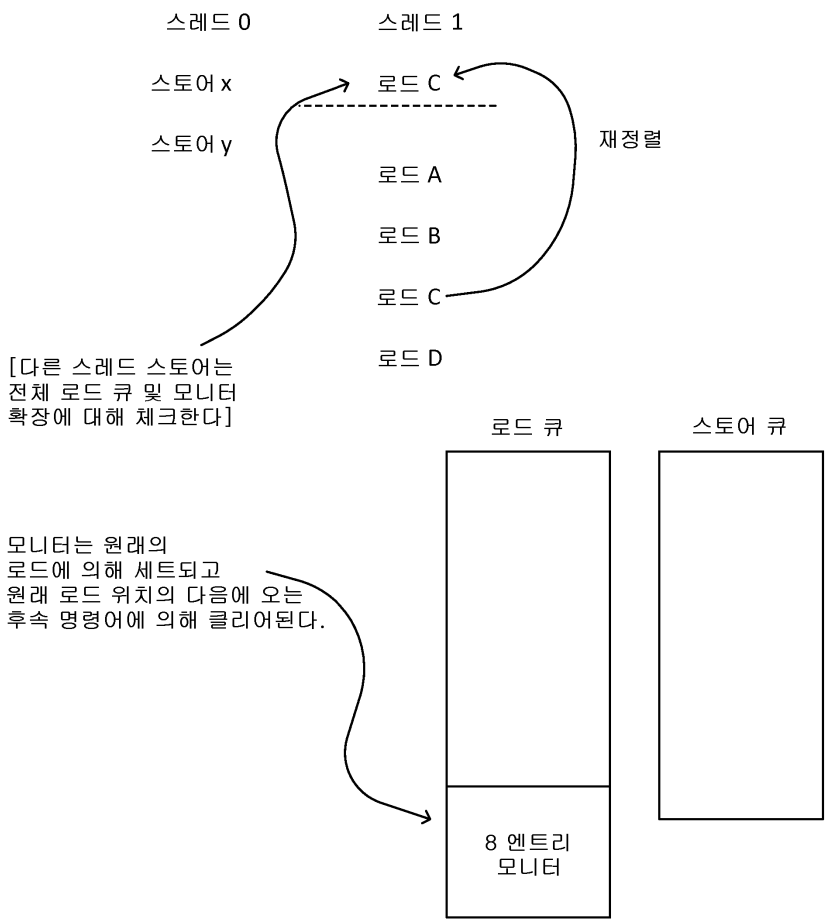
도면19

JIT 최적화를 통한 재정렬 프로세스

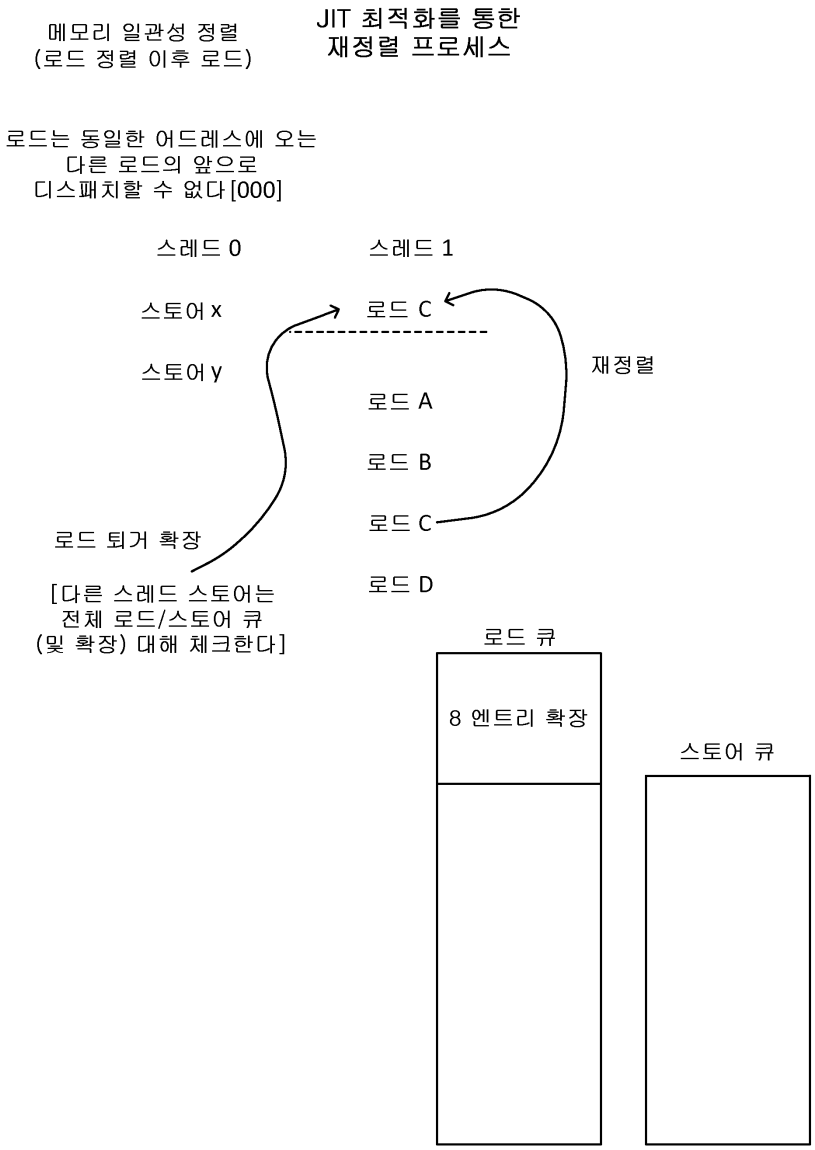


도면20

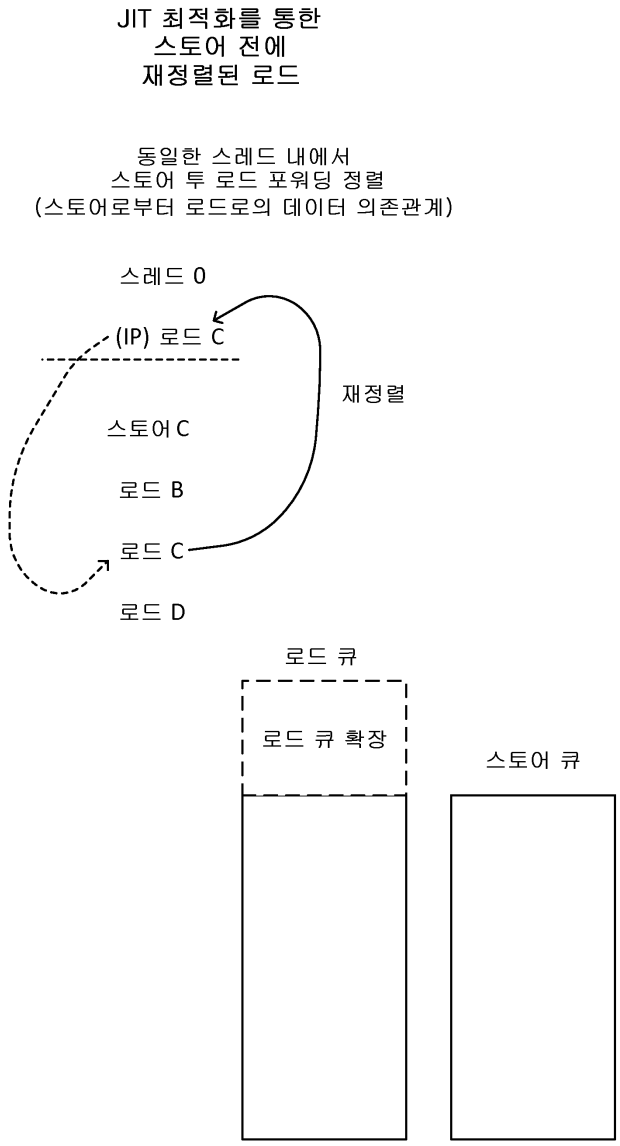
JIT 최적화를 통한 재정렬 프로세스



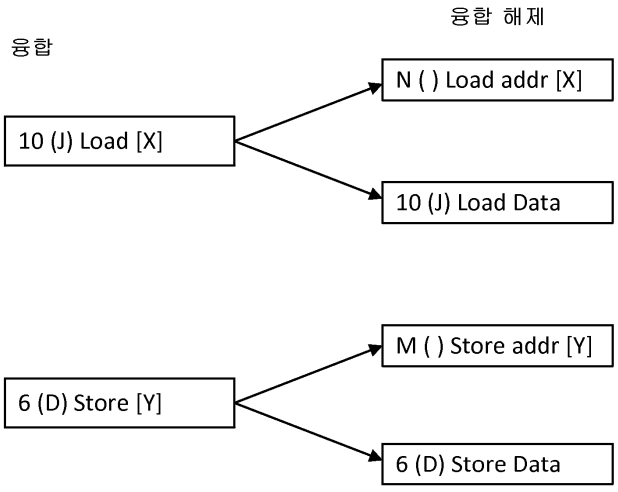
도면21


메모리 일관성 정렬
(로드 정렬 이후 로드)
JIT 최적화를 통한
재정렬 프로세스
로드는 동일한 어드레스에 오는
다른 로드의 앞으로
디스패치할 수 없다[000]
스레드 0
스레드 1
스토어 x
로드 C
스토어 y
재정렬
로드 A
로드 B
로드 C
로드 퇴거 확장
로드 D
[다른 스레드 스토어는
전체 로드/스토어 큐
(및 확장) 대해 체크한다]
로드 큐
8 엔트리 확장
스토어 큐

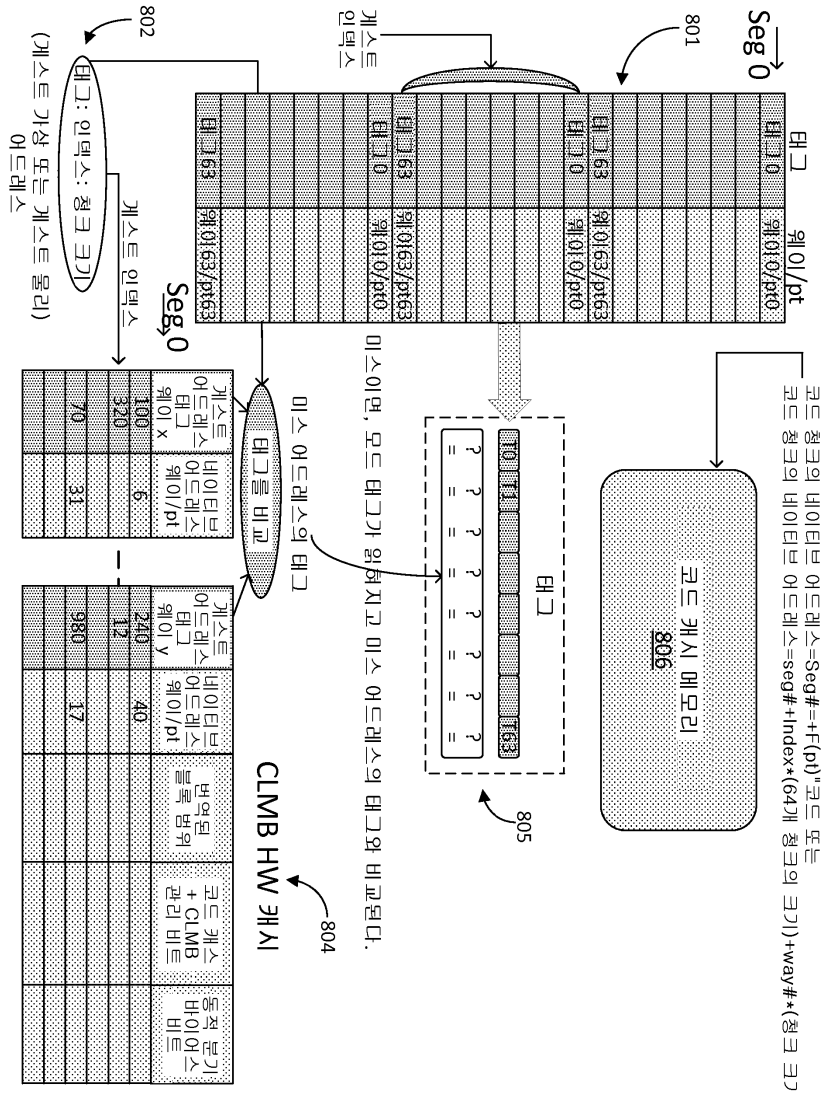
도면22


JIT 최적화를 통한
스토어 전에
재정렬된 로드
동일한 스레드 내에서
스토어 투 로드 포워딩 정렬
(스토어로부터 로드로의 데이터 의존관계)
스레드 0
(IP) 로드 C
재정렬
스토어 C
로드 B
로드 C
로드 D
로드 큐
로드 큐 확장
스토어 큐

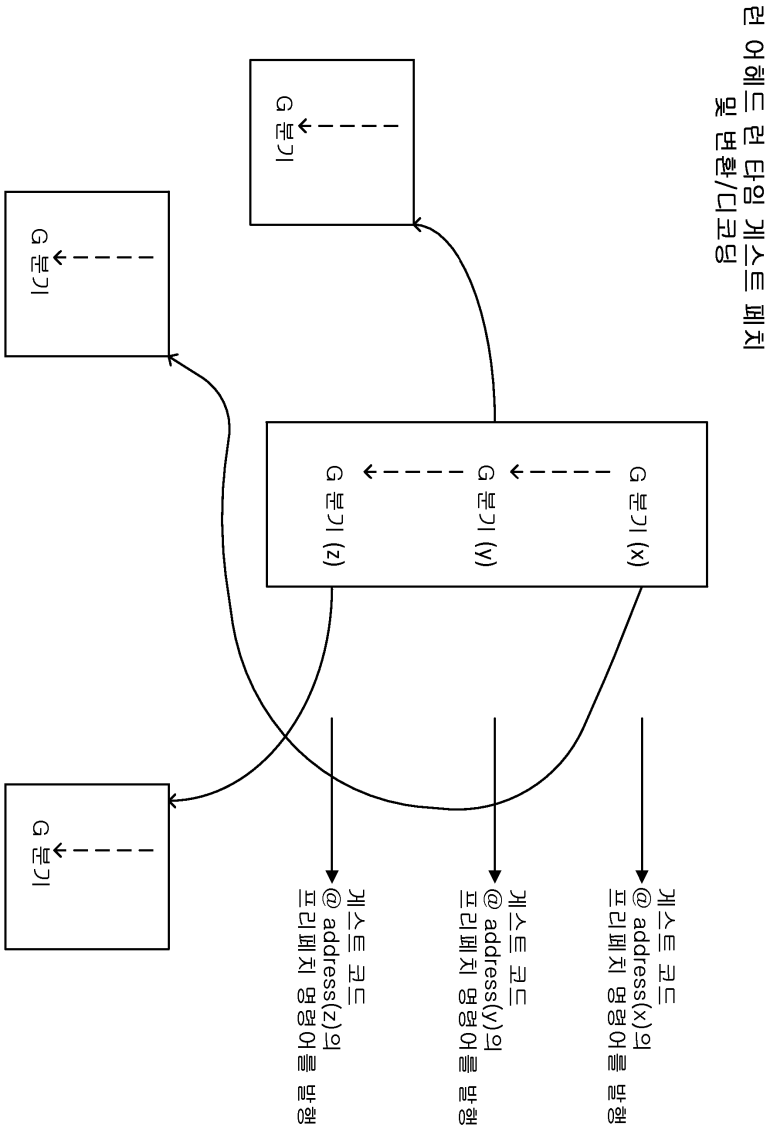
도면23


융합
융합 해제
10 (J) Load [X]
N ( ) Load addr [X]
10 (J) Load Data
6 (D) Store [Y]
M ( ) Store addr [Y]
6 (D) Store Data

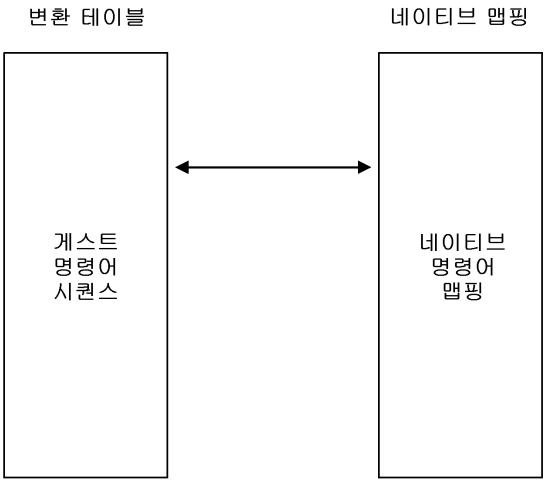
도면24


802
(게스트 가상 또는 게스트 물리) 어드레스
태그: 인덱스: 청크 크기
게스트 인덱스
801
Seg 0
태그
웨이/pt
태그_0
웨이0/pt0
웨이0/pt63
태그_63
웨이63/pt63
게스트 인덱스
Seg 0
게스트 어드레스 태그
웨이 x
네이티브 어드레스 웨이/pt
100
320
70
31
6
게스트 어드레스 태그
웨이 y
네이티브 어드레스 웨이/pt
240
12
980
17
40
태그를 비교
미스 어드레스의 태그
CLMB HW 캐시
804
805
태그
? ? ? ? ? ? ? ?
= = = = = = = =
10 11 163
미스이면, 모드 태그가 앞쳐지고 미스 어드레스의 태그와 비교된다.
코드 캐시 메모리
806
코드 청크의 네이티브 어드레스=Seg#=+f(pt)"코드 또는
코드 청크의 네이티브 어드레스=seg#+Index*(64개 청크의 크기)+way#*(청크 크기)
벙어둠 블록 범위
코드 캐쉬 + CLMB 관리 비트
동적 분기 바이어스 비트

도면25


런 어헤드 런 타임 게스트 페치
및 변환/디코딩
G 분기
G 분기
G 분기 (x)
G 분기 (y)
G 분기 (z)
G 분기
게스트 코드
@ address(x)의
프리페치 명령어를 발행
게스트 코드
@ address(y)의
프리페치 명령어를 발행
게스트 코드
@ address(z)의
프리페치 명령어를 발행

도면26


변환 테이블
네이티브 맵핑
게스트
명령어
시퀀스
네이티브
명령어
맵핑