경량 동적 코드 변환 기법의 설계 및 구현

김 지 홍, 이 동 우, 김 인 혁, 엄 영 익

성균관대학교 정보통신공학부

{ezjjilong, dwlee08}@gmail.com, {kkojiband, yieom}@ece.skku.ac.kr

Design and Implementation of Light-weight Dynamic Binary Translation Scheme

Jeehong Kim, Dongwoo Lee, Inhyuk Kim, Young Ik Eom

School of Information and Communication Engineering, Sungkyunkwan Univ.

요 약

최근 그린 IT, 클라우드 컴퓨팅 등이 새롭게 주목 받음에 따라 이들의 기반 기술인 가상화 기술이 더욱 활발히 연구되고 있다. 이에 따라 본 논문에서는 다양한 시스템을 손쉽게 운영할 수 있는 전가상화의 장점을 극대화하기 위해 새로운 동적 코드 변환기법에 대하여 제안한다. 이를 위해 동적 주소 변환 기법과 베이직 블록의 특성에 따라 동적 코드를 경량화하는 기법을 설계하였다. 기존의 동적 코드 변환 기법과의 성능 비교를 통해 제안한 기법의 안정성과 경량성를 확인할 수 있었다.

1. 서 론

최근 급변하고 있는 비즈니스 환경의 변화로 변화에 잘 대응할 수 있는 유연성과 변화에 빠르게 대응할 수 있는 속도의 중요성이 부각되면서 차세대 컴퓨팅은 필요한 자원에 대해 추가 개입 없이 원하는 만큼의 IT인프라를 언제 어디서나 손쉽게 얻고 확장할 수 있는 소프트웨어 플랫폼을 추구하고 있으며, 가상화 기술은 이러한 환경을 구축할 수 있는 핵심 기술로써 각광받고 있다.

가상화 기술은 시스템 구성 계층구조 상에서 어느 계층의 추상화를 통해 가상화를 구현하느냐에 따라 OS-level 가상화, 반가상화(para-virtualization), 전가상화(full-virtualization) 그리고 가상화를 지원하는 하드웨어(hardware-assisted virtualization) 기법으로 구분된다. 먼저 OS-level 가상화는 프로세스에게 필요한 자원들을 분배하여 각각의 프로세스들이 독립적인 환경에서 동작할 수 있도록 지원하는 기법이다. 하나의 운영체제를 기반으로 동작하며, 전가상화나 반가상화와 같이 특권 명령어 처리에 대해서 고려할 필요가 없다는 장점이 있는 반면, 가상머신들이 독립된 환경을 갖기 위해 운영체제의 수정이 요구된다. 전가상화 기법은 하이퍼바이저(hypoervisor, virtual machine monitor)가 가상머신의 명령어들을 일정기준[1]에 따라 묶음 단위로 읽어서 수행하는 동적 코드 변환기법(binary translation)을 지원한다. 동적 코드 변환기법을 이용한 전가상화 기법은 가상머신이 동작하면 가상머신 운영체제에 대한 수정이 필요하지 않기 때문에 기존 운영체제를 그대로 가상머신에서 이용할 수 있다. 그러므로 다양한 시스템을 손쉽게 운영할 수 있다는 장점을 갖는다. 하지만 기존 시스템보다 느린 성능을 보인다는 단점을 수반한다. 반가상화 기법은 전가상화의 단점을 개선하기 위해 개발된 기법으로서 가상머신에서 특권 명령어가 발생하면 이를 하이퍼바이저가 처리할 수 있는 형태로 변형시켜서 가상머신이 하이퍼바이저의 영역을 침범하지 못하게 함으로써 가상머신들이 기존 시스템에 가까운 성능을 나타내도록 지원하며, 다양한 운영체제들이 동시에 동작할 수 있도록 설계 할 수 있다. 그러나 가상머신이 하이퍼바이저에 맞게 수정되어야 하는 단점이 있다. 마지막으로 가상화를 지원하는 하드웨어 기법은 하드웨어가 지원하는 가상머신과 하이퍼바이저의 실행명령 권한 계층을 구분함으로써 가상화에 걸림돌이 되는 특권명령어들에 대한 관리를 손쉽게 할 수 있도록 하였다. 하지만 지원 가능한 하드웨어가 수반되어야 하고 다른 가상화 기법들이 추가 반영되어야만 성능을 개선 할 수 있다[1].

이와 같이 여러 가상화 기법 중에서 본 논문에서는 다양한 시스템을 손쉽게 운영할 수 있는 전가상화의 장점을 극대화하기 위해 성능저하의 원인이 되는 동적 코드 변환기법을 경량화하는 새로운 동적 코드 변환기법에 대하여 제안한다.

본 논문의 구성은 다음과 같다. 2장에서 일반적인

본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음 (NIPA-2010-(C1090-1021-0008).

동적 코드 변환기법과 기존의 기술에 대하여 소개하고, 3장에서는 기존의 코드 변환기법을 경량화한 새로운 코드 변환기법에 대해 제안한다. 4장에서 제안한 기법을 기존의 방법과의 비교를 통하여 평가하고, 마지막으로 5장에서는 결론 및 향후 필요한 연구에 대하여 설명한다.

2. 동적 코드 변환 기법

코드 변환 기법은 임의의 명령어 집합(instruction set architecture)을 해당 아키텍처가 아닌 다른 아키텍처에서 실행하기 위해 적절한 명령어 집합으로 기계어 코드수준에서 변환하는 기법이다. 이러한 동작을 하는 시스템을 흔히 '에뮬레이터(emulators)'라고 하고, 동적 코드 변환 기법은 에뮬레이터가 활용하는 효율적인 방법 중 하나다. 현재 대부분의 동적 코드 변환 시스템은 기존의 아키텍처에서 새로운 아키텍처로 변환하기 위한 이주 장치(migration tool)로 사용되고 있다. 코드 변환 기법은 임의의 아키텍처의 모든 바이너리 코드를 새로운 바이너리 코드로 변환하는 정적(static) 코드 변환 기법과 상황에 따라 동적(dynamic)으로 코드 변환을 수행하는 동적 코드 변환 기법으로 나눌 수 있다. 코드와 데이터가 같은 메모리에 상주하는 본 뉴먼(von Newmann) 아키텍처에서 정적 코드 변환 기법은 동적 링킹(dynamic linking)과 자가 수정(self-modifying)과 같은 문제점들 때문에 완벽한 해답이 되지 못했다. 이에 반해 동적 코드 변환 기법은 이러한 문제들을 해결할 뿐만 아니라 새로운 코드를 동시에 생성할 수 있기 때문에 변환 시에 오버헤드가 발생함에도 불구하고 정적 코드 변환 기법보다 빠르게 변환할 수 있다[2].

또한 최근 소프트웨어의 복잡도가 증대되면서 실행중인 애플리케이션에 동적으로 추가의 코드를 삽입할 수 있는 기능이 중요해졌고, 이는 동적 코드 변환 기법을 이용하여 할 수 있게 되었다.

이와 같은 성능상의 장점을 가지고 있는 동적 코드 변환 기법이 가지는 코드 변환 시에 발생하는 오버헤드를 감수해야 하는 단점을 보완하기 위해 다음과 같은 연구가 진행되어 왔다.

이 장에서는 최근 연구가 진행되고 있는 가장 대표적인 동적 코드 변환 기법인 Pin, Valgrind, HDTrans를 소개한다.

2.1 Pin

Pin은 리눅스 운영체제 환경에서의 애플리케이션을 수행하면서 추가하고자 하는 코드를 삽입할 수 있는 소프트웨어 시스템이다. Pin의 목표는 다양한 아키텍처의 다양한 프로그램을 분석할 수 있는 플랫폼을 제공하는 것이다. Pin은 just-in-time(JIT) 컴파일 기법을 이용함으로써 코드를 삽입하고 최적화하는 효율적인 기능을 제공한다[3].

2.2 Valgrind

Valgrind는 동적 재 컴파일 기법을 포함한 just-in-time 컴파일 기법을 이용한 가상머신이다. Valgrind는 중량의 동적 코드를 분석 또는 측정하고 프로파일링하거나, 메모리 분석과 디버깅을 수행하기 위해 정교한 동적 코드 변환 기법을 사용한다. 호스트 프로세서에서는 직접 코드를 수행하지 않는 대신에 중간 코드(intermediate representation, IR) 형식의 간단한 프로그램으로 변환하고, 변환 후에 장치는 Valgrindr가 중간코드를 기계어로 변환하고 호스트 프로세서가 기계어를 수행하기 전에 변환된 코드를 수행하는 변환 순서로 수행된다[4].

2.3 HDTrans

HDTrans는 다른 동적 코드 변환 기법과 달리 IA-32 아키텍처 사이의 동적 변환 기법이다. HDTrans는 가상머신의 에뮬레이션을 위한 목적으로 개발되었다. 현재의 HDTrans는 성능향상과, VMware의 오픈 소스화, 그리고 정적 코드 변환 기법과 동적 코드 변환 기법의 통합을 통해 반가상화 기법이 가지는 재 컴파일의 필요성을 줄이는 데에 목적이 있다[5][6].

3. 경량 동적 코드 변환 기법

그림 1은 본 논문에서 제안하는 동적 코드 변환의 경량화 기법의 흐름도를 나타낸다. 먼저 메모리 영역에 코드 캐시영역을 할당하고 동적 주소 변환기법을 위한 주소 변환 테이블을 생성한다. 그 후, 코드캐시영역과 스택 영역을 적절히 조작하여 베이직 블록의 코드를 변환하기 위한 준비를 하고 해당 베이직 블록의 각 명령어들의 종류에 따라 코드캐시영역에 변환된 코드를 할당한다. 마지막으로 코드캐시영역의 변환된 코드를 수행하고 해당프로세스를 종료한다.

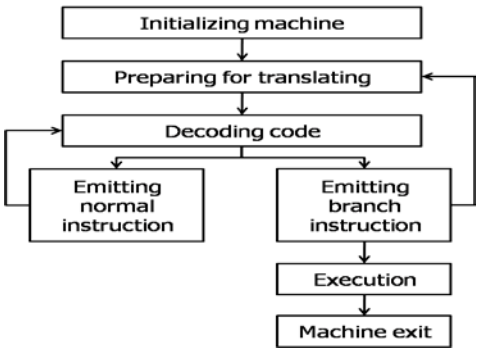


그림 1 제안하는 동적 코드 변환 기법의 흐름도

3장에서는 동적 주소 변환기법과 베이직 블록에 존재하는 분기 명령어 중 'ret', 'call', 'jmp'를 포함하는 베이직블록을 변환하여 코드캐시영역에 할당하고, 실행하는 기법에 대해 자세히 설명한다.

3.1 동적 주소 변환 기법

그림 2는 3단계 변환 테이블을 이용하여, 변환하려는 베이직 블록에 대응하는 코드캐시영역의 시작 주소 값을 반환하는 과정을 나타낸다. 베이직 블록의 주소 값을 인덱스 값으로 하여, 3단계 변환 테이블을 이용한 동적 주소 변환기법을 통해 세 번째 인덱스 테이블이 가리키는 주소의 값인 코드캐시영역의 시작주소 값을 얻게 된다. 인덱스 값의 첫 12비트는 첫 번째 테이블의 오프셋을 나타내고, 다음 10비트는 두 번째 테이블의 오프셋을 나타내며, 마지막 10비트를 세 번째 테이블로 나타냄으로써 모든 메모리 영역의 주소 값을 일대일 대응할 수 있고, 이는 O(1)의 복잡도로 메모리의 변환이 가능하다. 2.3장에서 소개한 HDTrans의 경우, 동적 주소 변환 기법과 유사한 기법으로 헤시 테이블을 이용하는데, 이 기법은 주소충돌(address collision)을 방지할 수 없는 단점이 존재한다. 하지만 제안한 동적 주소 변환 기법은 주소충돌과 같은 기존의 동적 코드 변환 기법이 가지는 문제점을 해결하였을 뿐만 아니라 메모리 영역의 각 주소 값을 일대일 대응함으로써 성능의 향상도 기대할 수 있다.

3.2 'ret'을 포함한 베이직 블록의 동적 코드 변환 기법

그림 3은 'ret'을 포함한 베이직 블록과, 그 베이직 블록을 동적 코드 변환 기법을 통해 변환한 코드캐시영역의 베이직 블록을 나타낸다. 먼저 베이직 블록을 변환하기 위해 초기화 과정을 수행한다. 다음으로 변환의 전처리 과정을 실행하는 과정 중에 베이직블록의 각 명령어를 구분하여 코드캐시영역에 할당하는 과정을 수행한다. 이는 베이직블록의 각 명령어를 종류별로 구분하여 'movl', 'addl'과 같은 일반적인 명령어는 코드캐시영역에 그대로 할당하고, 분기 명령어는 변환 함수를 통해 코드캐시영역에

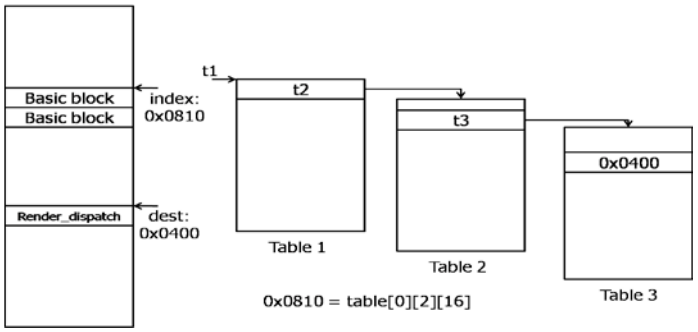
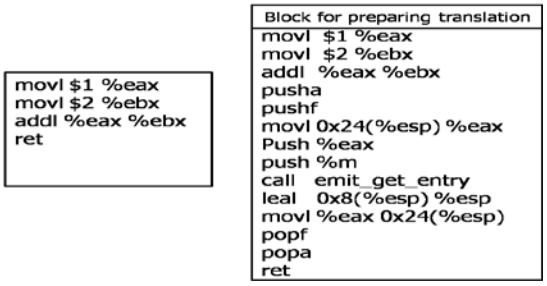


그림 2 동적 주소 변환 기법



(a) (b)

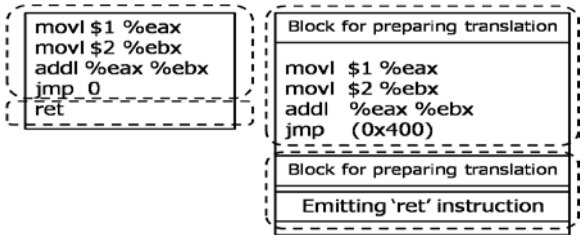
그림 3 'ret'을 포함한 베이직 블록의 동적 코드 변환 기법

(a) 베이직 블록(변환 전), (b) 변환된 베이직 블록

할당하는 과정이다. 이는 베이직블록의 각 명령어를 종류별로 구분하여 'movl', 'addl'과 같은 일반적인 명령어는 코드캐시영역에 그대로 할당하고, 분기 명령어는 변환 함수를 통해 코드캐시영역에 할당하는 과정이다. 그 후 변환을 준비하는 과정의 나머지부분과 변환 함수를 수행하고 프로세스를 종료한다.

3.3 'jmp'을 포함한 베이직 블록의 동적 코드 변환 기법

그림 4는 'jmp'를 포함한 베이직 블록과, 그 베이직 블록을 동적 코드 변환 기법을 통해 변환하여 코드캐시영역에 할당된 베이직 블록을 나타낸다. 먼저 3.2장에서 설명한 방법과 동일한 방법으로 초기화 과정을 수행하고 변환의 전처리 과정 중에 베이직블록의 각 명령어를 구분하여 코드캐시영역에 할당하는 과정을 수행한다. 3.2장에서 설명한 'ret' 명령어의 변환과정과는 달리 'jmp' 명령어는 변환 함수를 호출하고, 3.1장에서 설명한 동적 주소 변환 기법을 이용하여 다음 베이직 블록의 시작 주소를 찾는다. 그 시작 주소의 값이 이전에 변환되었던 명령어의 주소 값인지에 대해 확인한 후에 변환되었던 명령어의 주소 값이면 해당 주소로 분기하여 해당 명령어를 수행하고, 변환되지 않았던 명령어의 주소 값이라면 앞서 3.2장에서 설명했던 동작원리와 같은



(a) (b)

그림 4 'jmp'을 포함한 베이직 블록의 동적 코드 변환 기법

(a) 베이직 블록(변환 전), (b) 변환된 베이직 블록

순서대로 변환을 수행하고 프로세스를 종료한다.

3.4 'call'을 포함한 베이직 블록의 동적 코드 변환 기법

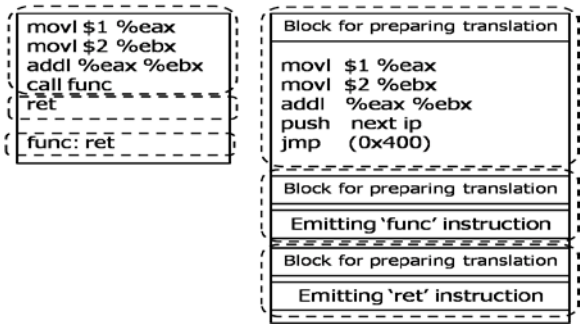
그림 5는 'call'을 포함한 프로그램영역에 존재하는 베이직 블록과, 그 베이직 블록을 동적 코드 변환 기법을 통해 변환하여 코드캐시영역에 할당된 베이직 블록을 나타낸다. 앞서 설명한 방법과 같은 방법으로 초기화 과정과 변환의 전처리 과정을 수행하는 과정 중에 'call'과 관련된 부분을 코드캐시영역에 할당한다. 'call'에 의해 호출되는 함수가 'func: ret' 이므로 3.3장에서 수행했던 과정을 동일하게 수행하고 난 후, 'ret'에 해당하는 변환 과정을 위해 다시 한번 3.3장과 같은 동작과정을 수행하고 프로세스를 종료한다.

4. 평가

본 논문에서는 gzip, bzip, md5sum, ls와 같이 다양한 명령어들에 대해, valgrind, pin, HDTrans와 같은 기존의 동적 코드 변환 기법과의 성능 비교를 통하여 평가하였다. 비교 평가 결과 그림 6과 같은 성능 비교 결과를 확인할 수 있었다. Pin과 valgrind는 인스트루멘테이션 장치이기 때문에 많은 최적화 과정 수행하는 과정에서 오버헤드가 생성되어 제안 기법의 결과와는 큰 차이를 보였다. 또한 HDTrans의 성능이 제안기법보다 좋은 결과를 보였지만, 충분한 최적화 과정이 진행되어온 HDTran의 결과와 비슷한 성능을 보인 것으로 보아 추후 최적화 작업을 통해 제안 기법의 성능 향상을 이룰 수 있을 것으로 생각된다.

5. 결론

본 논문에서는 다양한 시스템을 손쉽게 운영할 수 있는 전가상화의 장점을 극대화하기 위해 새로운 동적 코드 변환기법에 대하여 제안하였다. 기존의 동적 코드 변환 기법과의 성능 비교를 통해 제안한 기법의 안정성과



(a) (b)

그림 5 'call'을 포함한 베이직 블록의 동적 코드 변환 기법
(a) 베이직 블록(변환 전), (b) 변환된 베이직 블록

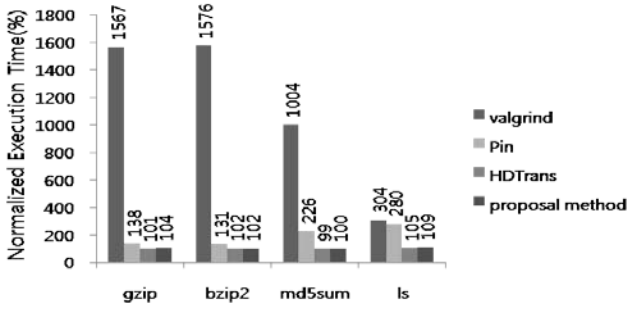


그림 6 기존의 동적 코드 변환 기법과의 성능 비교

경량화를 확인할 수 있었다. 하지만 기존의 방법과 큰 성능차이를 보이지 않았으므로 최적화에 대한 연구가 향후 진행되어야 할 것으로 생각된다.

6. 참고문헌

[1] 김인혁, 김태형, 김정한, 임병홍, 엄영익, "시스템 보안을 위한 가상화 기술 활용 동향", 한국정보보호학회지, 제 19권, 2호, pp. 26-34, 2009. 4

[2] Mark Probst, "Dynamic binary translation", In Proceedings of the UKUUG Linux Developers' Conference, Bristol, United Kingdom, July 2002.

[3] Chi-Keung Luk , Robert Cohn , Robert Muth , Harish Patil , Artur Klauser , Geoff Lowney , Steven Wallace , Vijay Janapa Reddi , Kim Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation", Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, , Chicago, IL, USA, June 2005

[4] Nicholas Nethercote , Julian Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation", Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, San Diego, California, USA, June 2007

[5] Swaroop Sridhar, Jonathan S. Shapiro, Prashanth P. Bungale,"HDTrans: a low-overhead dynamic translator", ACM Computer Architecture News, vol. 35, issue 1, pp. 135-140, March 2007

[6] Swaroop Sridhar, Jonathan S. Shapiro, Prashanth P. Bungale,"HDTrans: an open source, low-level dynamic instrumentation system", In *proc.* 2006 International Conference on Virtual Execution Environments (VEE), pp.175-185, Ottawa, Ontario, Canada, June 2006