내장형 자바 시스템을 위한 클라이언트 선행 컴파일 기법을 이용한 코드 캐시 확장

(Expanding Code Caches for Embedded Java Systems using Client Ahead-Of-Time Compilation)

홍 성 현 † 김 진 철 ††
(Sunghyun Hong) (Jin-Chul Kim)
신 진 우 ††† 권 진 우 †††
(Jin Woo Shin) (Jin-Woo Kwon)
이 주 환 † 문 수 묵 ††††
(Joohwan Lee) (Soo-Mook Moon)

요 약 많은 내장형 자바 시스템들이 제한된 메모리를 가지고 있으므로 JITC를 위해 충분한 코드 캐시가 주어지지 않아 자바의 수행 성능이 떨어질 수 있다. 본 논문에서는 이를 극복하고자 수행 중에 코드 캐시 공간이 부족하면 일부 메소드의 머신 코드를 영구적 메모리의 파일 시스템에 저장해 두었다가 재호출 때에 다시 코드 캐시로 읽어와서 재활용하는 코드 캐시를 확장하는 수행 방식을 제안한다. 이는 기존의 클라이언트 선행 컴파일 기법을 수행 중에 코드 캐시 확장을 위해 적용한 것이다. 우리가 제안한 자바 수행 구조는 코드 캐시가 반으로 줄었을 때의 일반적인 자바 수행 방식보다 1.6배 좋은 성능을 보여주고 있다.

키워드 : 코드 캐시, 자바 가상 머신, 적시 컴파일, 클라이언트 선행 컴파일, 코드 재배치

Abstract Many embedded Java systems are equipped with limited memory, which can constrain the code cache size provided for Java just-in-time compilation, affecting the Java performance. This paper proposes expanding the limited code cache when it is full, by saving the machine code for some methods in the code cache into the file system of the permanent storage and reloading it to the code cache when they are re-invoked later. This is applying the client ahead-of-time compilation during the execution time for the purpose of enlarging the code cache. Our experimental results indicate that the proposed execution method can improve the performance by as much as 1.6 times compared to the conventional method, when the code cache size is reduced by half.

Key words : Code cache, Java Virtual Machine, JITC, client-AOTC, relocation

1. 서 론

최근의 많은 내장형 시스템들은 소프트웨어 플랫폼으로 자바를 선택하고 있다. 예를 들어서 휴대폰, 디지털 TV, 홈 네트워크, 블루레이 디스크, 텔레매틱스(무선통신으로 차량과 서비스센터를 연결하여 각종 정보와 서비스를 제공) 등에서 자바를 기본 소프트웨어 플랫폼으로 사용하고 있다[1]. 내장형 시스템들이 자바를 선택하는 이유는 첫째로, 가상 머신을 사용함으로써 CPU, OS, 하드웨어 기반이 다양한 내장형 플랫폼에서 일관된 실행 환경을 제공한다는 점이다. 두 번째는 보안상의 이점으로 자바 플랫폼은 바이러스나 해킹 자바코드로 인해 전체 시스템이 다운될 가능성이 적다. 마지막으로 성숙하고 풍부한 API(Application Program Interface)를 제공하고 프로그램의 안정성을 높여주는 언어 기능(쓰레기 수집기, 예외 처리)이 소프트웨어 콘텐츠 개발을 쉽게 하기 때문이다. 특히 "Write once, run everywhere"라는 호환성의 장점으로 독립 플랫폼 환경을 제공한다.

독립 플랫폼 환경의 장점은 자바가 가상머신 (Java virtual machine, JVM)을 사용함으로써 얻어진다. 즉 자바는 하드웨어에서 직접 수행될 수 있는 머신 코드로 컴파일되는 것이 아니라 바이트코드[2]라고 불리는 중간 코드로 컴파일되고 배포되어 JVM이 설치된 곳이면 어디에서나 인터프리터에 의해 수행된다. 이러한 소프트웨

· 본 연구는 지식경제부 및 한국산업기술평가관리원의 산업원천기술개발사업의 일환으로 수행하였음[2009-S-036-01, 고성능 가상머신 규격 및 기술 개발]
· 이 논문은 제36회 추계학술발표회에서 '내장형 자바 시스템을 위한 클라이언트 선행 컴파일 기반의 코드 캐시 확장'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 서울대학교 전기.컴퓨터공학부
hongshow@snu.ac.kr
joohwan@aces.snu.ac.kr
†† 비 회 원 : SAP R&D Center Korea
jckim@altair.snu.ac.kr
††† 비 회 원 : 서울대학교 전기.컴퓨터공학부
jinwoosh@usc.edu
jiniarin@naver.com
†††† 종신회원 : 서울대학교 전기.컴퓨터공학부 교수
smoon@snu.ac.kr

논문접수 : 2009년 12월 24일
심사완료 : 2010년 4월 12일

어에 의한 실행은 하드웨어에 의한 실행보다 느릴 수밖에 없으므로 적시 컴파일러(Just-In-Time Compiler, JITC)[3-5]나 선행 컴파일러(Ahead-Of-Time Compiler, AOTC)[6-8]처럼 바이트코드를 머신 코드로 번역하는 컴파일 기법을 사용하여 가속한다. JITC은 수행 시간 중에 핫스팟(프로그램에서 자주 반복적으로 사용되는 코드의 일부 부분)으로 판명된 메소드의 바이트코드를 머신 코드로 번역하여 수행하며, AOTC는 번역을 서버에서 미리 수행하고 번역된 코드를 클라이언트 머신에 장착하여 실행 중 사용한다. 또한, 클라이언트에서 JITC를 이용하여 AOTC를 미리 수행하는 클라이언트 AOTC도 사용할 수 있다[9].

내장형 시스템에서는 JITC의 수행 중 번역 오버헤드 때문에 AOTC가 더 적합하다고 할 수 있다. 하지만, 내장형 시스템은 내려 받은 프로그램을 사용하는 용도로 개발되는 경우가 많으며, 내려받은 프로그램에는 AOTC를 적용하기가 어려워서 여전히 JITC을 채택해야 한다. 그런데 내장형 시스템은 수행 시간에 사용되는 메모리 자원은 부족하여 JITC가 생성한 머신 코드를 저장하는 메모리 공간인 코드 캐시(code cache)가 부족할 수가 있다. 이 경우 두 가지 간단한 해결 방법이 있는데 우선 더 이상의 JITC를 포기하는 것으로 코드 캐시가 가득 찬 이후로는 핫스팟으로 선택된 메소드라 할지라도 JITC되지 않고 인터프리터로 계속 수행하는 것이며 이로 말미암은 성능 저하가 있을 수 있다. 또 다른 방법은 몇몇 메소드의 머신 코드를 새롭게 JITC될 핫스팟 메소드를 위해 코드 캐시에서 선택해 삭제하는 것이다[10]. 삭제된 메소드가 다시 수행되면 인터프리터로 수행하거나 다시 JITC으로 번역해야 하는 데, 이 또한 인터프리터가 가져온 성능 저하와 반복되는 JITC의 수행이 가져오는 번역 오버헤드가 발생한다. 따라서 코드 캐시의 부족은 직접적인 성능 저하를 가져올 수 있다.

본 논문에서는 자바 프로그램 수행 중 코드 캐시가 부족할 때 저장된 일부 메소드의 머신 코드를 플래시메모리 같은 영구 메모리의 파일 시스템으로 옮겨서 코드 캐시 부족을 해결하고 파일에 저장된 메소드가 다시 호출되면 이를 코드 캐시에 로딩하여 바로 사용하는 방식을 제안한다. 이를 통해 코드 캐시 부족시 JITC를 포기할 필요가 없으며 또한 같은 메소드를 반복적으로 JITC하지 않아도 된다. 이는 파일 시스템을 이용하여 코드 캐시를 확장시키는 효과를 얻는 것이며 클라이언트 AOTC에서처럼 JITC를 이용한 AOTC를 수행 중에 적용하는 효과가 있다고 볼 수도 있다. 이때 코드 캐시에서 파일로 이동하는 머신 코드를 어떤 휴리스틱을 통해 선택하는 것이 최적의 성능을 나타내는 지가 중요한 연구 문제인데 파일 I/O의 속도가 느려서 이동 오버헤드로 작용하고 따라서 이동 횟수를 줄여야 하기 때문이다.

본 논문의 나머지는 다음과 같다. 2장에서는 파일 시스템에 머신 코드를 저장하는 방식을 사용한 코드 캐시의 확장 방법을 설명하고, 3장에서는 실험결과를 보여서 효과를 확인한다. 4장에서는 결론을 내리도록 하겠다.

2. 파일 시스템을 이용한 코드 캐시의 확장

2.1 코드 캐시 확장을 위한 자바 수행 과정의 수정

그림 1은 일반적인 자바 프로그램을 JITC을 이용한 VM이 수행할 때의 간단한 수행 과정을 보여주고 있다. 수행하려는 메소드가 코드 캐시에 있으면 그 머신 코드를 수행하고, 없으면 핫스팟인지를 확인하여 핫스팟이 아니면 인터프리터를 이용해 수행하고, 핫스팟이라면 JITC을 통해 번역 후 머신 코드를 수행하게 된다. 인터프리터에서의 누적 정보는 핫스팟 선택의 기준이 된다.

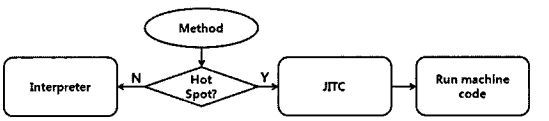

그림 1 자바 프로그램의 수행 과정

만약 코드 캐시가 부족하다면 우리가 제안하는 코드 캐시에서 선택된 메소드를 파일 시스템에 저장하는 방법을 구현하여 마치 코드 캐시가 확장된 것 같은 효과를 만들어서 성능 저하를 막으려고 한다. 그림 2는 머신 코드를 파일로 저장하는 방법을 구현함으로써 변경된 자바 프로그램의 수행 과정이다.

수행하려는 메소드가 코드 캐시에 없을 때 핫스팟인지 확인하여 핫스팟일 때는 파일에 저장된 머신 코드가 있는지 확인한다. 저장된 코드가 존재하면 코드 캐시가 꽉 찼는지를 확인한다. 코드 캐시가 가득 차지 않았다면, 머신 코드를 코드 캐시로 로딩하여 수행하면 된다. 코드 캐시가 다 차서 부족하다면 휴리스틱에 의해 코드 캐시에 존재하는 메소드들 중에 일부를 선택하여 파일 시스템에 저장하고 코드 캐시에서는 제거한다. 비워진 코드 캐시의 공간에 수행할 메소드의 머신 코드를 로딩하여 수행한다. 이 과정은 그림 2의 왼쪽 부분에 있다.

만약 핫스팟 메소드에 대해 파일에 저장된 머신 코드가 없다면, JITC을 통해 새로운 머신 코드를 생성한다. JITC을 수행할 때도 코드 캐시에 공간이 있는지 확인해야만 한다. 코드 캐시가 비어 있다면, 머신 코드를 생성해서 바로 수행하면 되지만, 공간이 충분치 않다면 코드 캐시에 존재하는 메소드들 중에 휴리스틱을 통해 선택된 메소드들의 머신 코드를 파일에 저장하고 JITC된 머신 코드를 생성하여 코드 캐시에 저장하고 수행하게 된다.

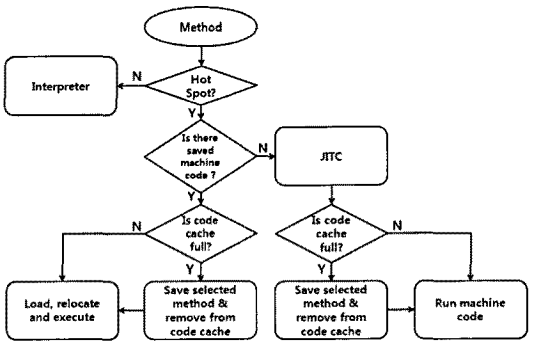

그림 2 수정된 자바 프로그램 수행 과정

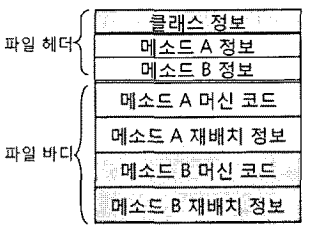

그림 3 머신 코드 저장 파일의 형식

이 과정은 그림 2의 오른쪽 부분에 있다.

파일에서 머신 코드를 코드 캐시로 읽어 들일 때 주소 재배치(relocation)가 필요하다. 재배치란 저장된 머신 코드를 재사용할 때 머신 코드에서 사용하는 주소값들을 현재 수행의 상황에 맞도록 수정하는 작업이다. 저장된 머신 코드를 읽어 들일 때, 데이터와 코드가 메모리에 있는 곳이 머신 코드를 저장할 때와 다를 수 있다. 저장된 코드를 그대로 사용하게 되면 코드에 있는 메모리 주소값은 전혀 다른 곳을 가리키게 돼서 오작동을 일으키게 된다. 이것을 수정하는 과정이 재배치이다. 예를 들어 invokestatic, getstatic/putstatic, lwc, new, anewarray, checkcast, instanceof 등의 바이트코드의 머신 코드에 대해 주소 재배치가 필요하며 클라이언트 AOTC에서 사용한 재배치 방식을 채용하였으며[9] 이로 인한 오버헤드가 파일 I/O의 오버헤드에 더해진다.

머신 코드가 저장되는 파일은 하나의 클래스마다 하나씩 생성한다. 그림 3에서처럼 파일은 헤더 부분과 바디 부분으로 나눌 수 있다. 헤더 부분에는 클래스에 대한 정보와 파일에 저장된 머신 코드의 메소드에 대한 정보가 저장되어 진다. 이 정보들은 JITC에 의해 생성된 머신 코드들이 이미 파일에 저장하는지를 확인할 때 사용하게 된다. 바디 부분은 메소드의 머신 코드와 재배치를 위한 정보들로 이루어지게 된다.

2.2 제거 머신 코드 선택 휴리스틱

그림 2의 과정에서 새로운 머신 코드가 들어갈 자리를 만들기 위해서 코드 캐시에서 제거할 메소드를 선택해야 한다. 최선의 선택은 가장 오랫동안 사용되지 않을 메소드를 찾아내는 것이겠지만, 이것은 앞으로의 동작을 정확히 예상해야 하기 때문에 불가능하다. 따라서 우리는 다음과 같은 두 가지 휴리스틱을 이용하여 제거할 메소드를 선택하는 작업을 수행하였다.

3.2.1 FIFO(First In First Out)

FIFO는 머신 코드가 생성되는 순서를 기억하여 가장 먼저 저장된 머신 코드를 선택하게 된다. FIFO는 가장 단순한 휴리스틱이지만, 가장 오래전에 생성된 머신 코드가 현재 수행되지 않을 확률이 높다는 가정에 따라 사용될 수 있다.

3.2.2 LFU(Least Frequently Used)

LFU는 메소드가 호출될 때마다 그 수(call count)를 저장하여, 가장 적은 횟수가 호출된 머신 코드를 선택하여 제거하게 된다. 이 알고리즘은 프로그램의 수행 동안 자주 수행되는 메소드는 계속하여 자주 호출되어 사용될 것이라는 판단을 기반으로 한다. Java VM에서 이런 판단은 JITC의 대상이 되는 핫스팟 메소드를 선택할 때도 사용되는 전제이다.

3. 실험 결과

우리는 지금까지 코드 캐시의 머신 코드를 파일에 저장하고 읽어들여서 코드 캐시가 확장된 것처럼 사용하는 방법을 소개하였다. 코드 캐시가 부족하여 새로 생성된 머신 코드가 저장될 공간이 부족하면 코드 캐시에 이미 존재하는 머신 코드를 선택하여 파일에 저장함으로써 새로 생성된 머신 코드를 저장할 코드 캐시를 확보하고, 저장된 머신 코드가 이후에 호출되면 파일에서 머신 코드를 읽어들여 재활용함으로써 성능 손실이 없도록 한다. 이번 장에서 우리가 구현한 코드 캐시 확장 방법의 성능을 평가하도록 하겠다.

3.1 실험 환경

이 실험은 SUN의 CVM RI[11] version build 1.01_fcs-std-b12에 우리가 구현한 JITC에서 수행되었다. 우리가 구현한 JITC은 안정성 검사에서 대부분을 통과했으며, JVM 규격을 따르고 있다.

실험에 사용한 머신은 MIPS 기반의 SoC(System on Chip)로 300MHz의 clock 속도를 가진다. I-cache로 16KB, D-cache로 16KB, 메인 메모리로 128MB를 사용하고 있다. 우리는 클라이언트 AOTC 파일을 저장하기 위해서 40GB의 하드디스크를 설치했다. OS는 내장형 리눅스(kernel v2.4.18)를 사용하고 있다. 벤치마크는 EEMBC [12]중에서 crypto와 png를 사용하였다. 성능 측정을 위해서 우리는 각각의 벤치마크를 10번씩 수행하여 그중 중간값을 가지는 5개를 선택하여 그 평균값을 얻

었다. 벤치마크의 결과에 다소 편차가 나타나기 때문에 이런 방식을 취하고 있다. 이런 편차는 내장형 시스템을 사용함으로써 발생하는 것으로 일반적인 PC나 서버환경보다 민감한 것으로 보인다. 벤치마크를 10번씩 수행하는 것은 CVM 자체를 처음부터 10번씩 수행하는 것을 의미한다.

각 벤치마크를 수행할 때, JITC이 생성하는 머신 코드의 양을 측정하여 벤치마크에 주어지는 코드 캐시의 양을 조절하여 코드 캐시가 모자란 상황을 연출하여 실험하였다. crypto는 138KB만큼의 머신 코드를 생성하며, png는 45KB만큼의 머신 코드를 생성한다.

3.2 성능 평가

벤치마크가 사용할 수 있는 코드 캐시를 벤치마크별로 100%, 90%, 75%, 50%로 제한하여 수행 시간을 측정해 보았다. 벤치마크의 성능은 벤치마크의 score가 아닌 직접 수행되는 시간을 측정하였다.

실험에서 휴리스틱 JITC은 코드 캐시가 허용하는 한 메소드를 번역하지만, 코드 캐시가 부족하면 이후에 핫스팟 메소드로 선택되더라도 번역하지 못하는 방식이다. 따라서 JITC은 코드 캐시가 100%일 때 최선의 성능을 가진다. 휴리스틱 FIFO와 LFU는 코드 캐시 확장을 위해 수정된 자바 수행 과정에서 코드 캐시에서 제거할 메소드를 선택하는 휴리스틱으로 각각 FIFO와 LFU를 적용한 것이다.

그림 4와 표 1은 crypto의 성능이다. 그림 4의 y축의 값은 표 1의 값을 100% JITC의 시간인 8,559ms로 나눈 비율을 그래프로 표현한 것이다. 따라서 그래프에서 1은 JITC이 100%의 코드 캐시를 가지는 경우이며, 이 숫자가 커질수록 수행 시간이 길어짐을 의미한다.

그림 4에서 성능을 비교해 보면, 코드 캐시가 줄어듦에 따라 JITC의 성능 저하가 가장 큼을 알 수 있다. 실제로 코드 캐시 크기에 따라 JITC에 의해 컴파일되는 메소드의 개수는 표 2와 같은 데 캐시 크기가 감소하면 컴파일되는 메소드의 수가 감소한다. 코드 캐시가 90%일 때 JITC은 100%의 코드 캐시와 비교해서 수행 시간에 큰 차이가 없는데, 이것은 번역을 포기한 10%의 메소드(Chess 2개, Png 1개)는 벤치마크 수행의 거의 마지막에 JITC으로 번역되며, 이들은 핫스팟으로 선택되지만 실제로 머신 코드를 생성한 이후에 많은 횟수의 수행을 하지 않기 때문이다. 그러나 75% 및 50%에서 컴파일되지 않는 메소드들의 경우 크기가 비교적 커서 수행 시간을 많이 차지하는 경우가 많아 성능이 급락하고 있다(대체로 수행 초기에 컴파일되어 75%나 50%의 캐시에 들어간 메소드들은 크기가 작은 경우가 많음).

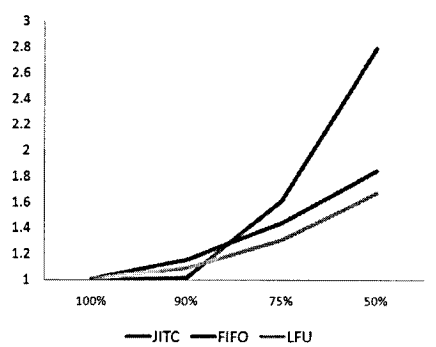

그림 4 crypto 수행 시간 비율 그래프

표 1 crypto 성능(ms)

휴리스틱	코드 캐시		
	90%	75%	50%
JITC	8,745	13,788	23,887
FIFO	9,905	12,316	15,774
LFU	9,366	11,220	14,322

FIFO와 LFU의 수행 시간은 JITC이 100%의 코드 캐시에서 수행되는 시간에 코드 캐시와 파일 시스템 간에 일어나는 머신 코드의 파일 I/O가 일어나는 시간을 합한 것과 같다고 볼 수 있다. 따라서 코드 캐시가 적을 때 발생하는 파일 I/O의 오버헤드가 핫스팟 메소드가 JITC으로 번역된 머신 코드로 수행되지 못하고 인터프리터로 수행되는 시간보다 작다고 할 수 있다.

표 3은 crypto를 수행할 때 코드 캐시와 파일 시스템 간에 머신 코드가 저장된 횟수와 로딩된 횟수를 보여주고 있다. crypto에서 JITC에 의해 머신 코드로 번역되는 메소드는 39개로 저장이 39개 된 경우는 번역된 모든 메소드가 파일에 저장되는 것이다. 표 3에서 코드 캐시가 적어지면 저장과 로딩의 횟수가 매우 증가하여 그림 4에서의 성능 저하의 원인이 되었다.

그림 5와 표 4는 png의 수행 시간의 그래프와 수행 시간 자료이다. 그림 5는 표 4의 값을 100% JITC의 시간인 8,269ms로 나누어 그 비율을 그래프로 표현한 것이다. png에서도 crypto와 마찬가지로 코드 캐시가 줄어듦에 따라 JITC이 가장 느려지고, FIFO와 LFU는 코드 캐시가 줄어들면서 제한된 코드 캐시를 사용하는

표 2 코드 캐시 크기에 따라 JITC 되는 메소드 수

벤치마크	코드 캐시			
	100%	90%	75%	50%
crypto	39	37	35	32
png	16	15	14	12

표 3 crypto에서 머신 코드가 저장, 로딩된 횟수

휴리스틱	코드 캐시					
	90%		75%		50%	
	저장	로딩	저장	로딩	저장	로딩
FIFO	29	14	39	169	39	488
LFU	4	47	8	175	17	396

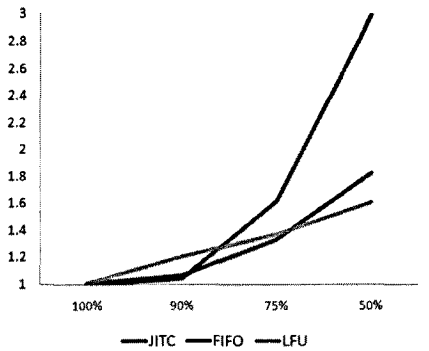

그림 5 png 수행 시간 비율 그래프

표 4 png 성능(ms)

휴리스틱	코드 캐시		
	90%	75%	50%
JITC	7,573	11,733	21,717
FIFO	7,737	9,706	13,247
LFU	8,769	9,975	11,677

표 5 png에서 머신 코드가 저장, 로딩된 횟수

휴리스틱	코드 캐시					
	90%		75%		50%	
	저장	로딩	저장	로딩	저장	로딩
FIFO	7	4	16	150	16	504
LFU	3	97	4	193	6	354

JITC보다 훨씬 빠르게 수행됨을 보이고 있다.

표 5는 png를 수행할 때 코드 캐시와 파일 시스템 간에 머신 코드가 저장된 횟수와 로딩된 횟수를 보여주고 있다. png에서 JITC에 의해 머신 코드로 번역되는 메소드는 16개이다. crypto와 같은 형태로 표 5에서 코드 캐시가 적어지면 저장과 로딩의 횟수가 매우 증가하여 그림 5에서의 성능 저하의 원인이 되었다.

4. 결 론

본 논문에서는 코드 캐시가 확장되는 효과를 얻기 위해 코드 캐시에 저장된 머신 코드를 파일로 저장하고 저장된 머신 코드는 이후에 주소 재배치와 로딩하여 재활용할 수 있는 구조를 구현하였다. 또한, 효과적인 수행을 위해 코드 캐시에서 제거할 메소드를 선택할 수 있는 휴리스틱을 구현했다. 우리가 수정한 자바 수행 구조는 부족한 코드 캐시 때문에 포기하게 되는 JITC의 성능 이득을 포기하지 않도록 한다. 이때 발생하는 파일 I/O의 오버헤드는 JITC으로 얻는 성능 이득보다 훨씬 적다. 만약 코드 캐시가 필요 치의 절반 수준을 줘서 JITC을 포기하는 구조에서는 수행 시간이 3배 느려지게 되지만, 코드 캐시를 확장하는 효과를 내면 수행 시간은 1.8배 정도 느려지게 된다. 즉, 파일 시스템을 이용해 코드 캐시를 확장하면 기존 JITC보다 1.6배 빨리 수행할 수 있다. 코드 캐시에서 제거할 머신 코드를 선택하는 휴리스틱은 다양하게 존재할 수 있으며, 우리가 구현한 단순한 휴리스틱만으로도 효과적으로 동작함을 보였다.

참 고 문 헌

[1] Sun Microsystems, "CDC: An Application Framework for Personal Mobile Devices," White Paper, Sun Microsystems.

[2] J. Gosling, B. Joy, and G. Steele, "The Java Language Specification," Addison-Wesley, 1996.

[3] J. Aycock, "A Brief History of Just-in-Time," ACM Computing Surveys, vol.35, no.2, pp.97-113, Jun. 2003.

[4] A. Krall, "Efficient JavaVM Just-in-Time Compilation," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp.54-61, 1998.

[5] A. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. Parikh, J. Stichnoth, "Fast, Effective Code Generation in a Just-in-Time Java Compiler," *Proceedings of ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pp.280-290, 1998.

[6] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, S. A. Watterson, "Toba: Java for Applications A Way Ahead of Time (WAT) Compiler," *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, p.3, 1997.

[7] G. Muller, B. Moura, F. Bellard, C. Consel, "Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code," *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, p.1, 1997.

[8] M. Weiss, F. de Ferrière, B. Delsart, C. Fabre, F. Hirsch, E. A. Johnson, V. Joloboff, F. Roy, F. Siebert, X. Spengler, "TurboJ, a Java Bytecode-to-Native Compiler," *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp.110-130, 1998.

[9] S. Hong, J. Kim, S. Moon, J. Shin, J. Lee, H. Oh, H. Choi, "Client ahead-of-time compiler for embedded Java platforms," *Software-practice & Experience*, vol.39, no.3, pp.259-278, 2009.

[10] L. Bak, J. R. Andersen, K. V. Lund, "Non-intrusive Gathering of Code Usage Information to Facilitate Removing Unused Compiled Code," US Patent, 6738969.

[11] Sun Microsystems, Java ME Connected Device Configuration (CDC), http://java.sun.com/products/cdc/

[12] The Embedded Microprocessor Benchmark Consortium, http://eembc.org