내장형 자바 시스템을 위한 클래스파일의 프리레졸루션

강두진, 맹혜선, 이영민, 한탁돈, 김신덕
연세대학교 미디어 시스템 연구실

Class File Pre-Resolution for Embedded Java System

Doo-Jin Kang, Hye-Seon Maeng, Young-Min Lee, Tack-Don Han, Shin-Dug Kim
Media System Lab, Yonsei Univ.

요 약

내장형 기기에서 자바를 제어 및 응용 프로그램으로 사용하는 경향이 증대하고 있다. 내장형 자바가상머신에서는 응용 프로그램과 관련 자바 API를 로딩된 형태로 롬에 적재하는 형태를 가진다. 따라서 응용프로그램은 필터링을 거쳐서 관련된 자바 API를 선별한 후 롬릿으로 만들어진다 본 논문에서는 내장형 자바가상머신에 적재될 롬릿을 만드는 과정에서 사용될 수 있는 프리레졸루션 방법을 제시하고 이를 위한 컨스턴트풀 데이터 구조를 제안하였다. 프리레졸루션은 롬릿을 만드는 과정에서 미리 레졸루션을 수행한 결과를 저장하여, 실행 시 발생되는 레졸루션 시간을 제거할 수 있도록 한다 또한 프리레졸루션은 메모리 접근 횟수를 감소시켜 내장형 기기에서 필요한 저전력 요구를 지원한다 본 논문에서 제안한 컨스턴트풀 데이터 구조는 공용체 구조의 미사용 부분에 레졸루션 결과를 저장할 수 있도록 구성함으로써 추가적인 메모리 비용없이 프리레졸루션의 이득을 얻을 수 있다.

1. 서 론

내장형 기기에서 자바[1] 수행 환경을 이용하는 경향이 늘어나고 있다. 내장형 기기의 제어 및 응용 프로그램을 자바 언어로 작성하는 것은 프로그램의 생산성 및 이식성을 높게 한다. 내장형 기기에서 자바 프로그램을 수행시키기 위해서는 내장형 자바가상머신과 내장형 API로 구성된 내장형 자바 환경[2]이 필요하다.

내장형 자바 환경에서는 엔터프라이즈 자바 환경이나 퍼스널 자바 환경과는 달리 수행할 응용 프로그램이 런타임 이전에 정해진다. 따라서 내장형 기기를 이용하는 경우에는 롬 메모리에 롬릿의 형태로 수행시간 중에 필요한 프로그램과 API를 정적 로딩하는 방법을 사용한다

본 논문에서는 내장형 기기를 위한 응용 프로그램과 API를 정적 로딩하는 과정에서 프리레졸루션을 수행하므로써 수행 시간 중의 레졸루션 비용을 줄일 수 있는 방법을 제시하도록 한다. 일반적으로 동적 로딩을 수행하는 경우에는 간접 참조 정보를 직접 참조 정보로 대치하는 레졸루션 작업이 수행 시간 중에 이루어지게 된다.

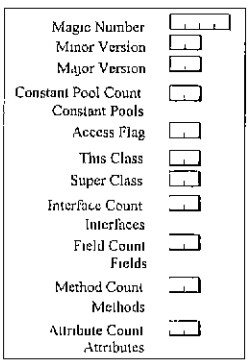
본 논문에서는 내장형 자바 환경에서 응용 프로그램의 수행 범위를 고려하여 관련된 API를 선별하는 필터링 방법과 롬 메모리에 적재될 응용 프로그램과 API를 프리레졸루션 과정을 거쳐서 롬릿 이미지를 구성하는 방법을 제시하도록 한다

본 논문의 구성은 다음과 같다 2 장에서는 메모리에 로딩되어야 하는 클래스 파일의 구조를 살펴보도록 한다 특히 간접 참조의 대상이 되는 컨스턴트풀 구조를 살펴보도록 한다

3 장에서는 내장형 자바 환경에서 프로그램이 적재되는 형태인 롬릿을 설명하고 이를 구성하기 위한 필터링 알고리즘을 제시한다. 4장에서는 컨스턴트풀 프리레졸루션을 수행하기 위한 방법과 이를 적용하기 위한 컨스턴트풀 자료 구조를 제시한다. 또한 제안된 방법을 통하여 얻을 수 있는 수행 이득을 설명한다. 마지막으로 결론과 향후 계획을 제시하도록 한다

2. 클래스 파일 구조와 레졸루션

자바 언어로 작성된 소스 프로그램을 컴파일함으로써 이진 파일 형태의 클래스 파일을 얻을 수 있다

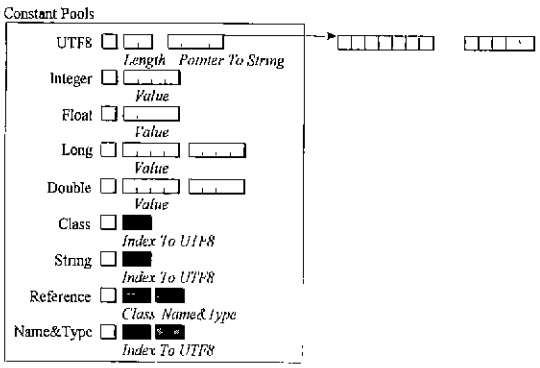


[그림 1] 클래스 파일 구조

자바 환경에서 수행될 프로그램과 API는 모두 클래스 파일 형태를 가져야 한다. 클래스 파일의 구조는 자바가상머신 명세[3]를 따라서 [그림 1]과 같이 구성된다. [그림 1]에서 사각형의 개수는 각각의 구성 요소를 저장하는데 필요한 바이트 수를 의미한 것이며, 컨스턴트 풀, 인터페이스, 필드, 메소드, 속성에 대한 정보는 생략하였다.

클래스 파일은 그 구조가 간단하지만, 내부적으로 간접 참조가 많이 이루어지는 인덱스 중심의 참조를 하고 있다. 클래스 파일은 내부적으로 값을 참조하기 위해서 컨스턴트풀을 이용하는데, 컨스턴트풀의 인덱스를 참조함으로써 필요한 정보를 참조한다. 이러한 인덱스 중심의 참조는 런타임 시에 직접 참조로 변환되어야 하는데, 이를 레졸루션이라 한다.

클래스 파일에서 인덱스 중심의 간접 참조는 컨스턴트풀에서 두드러지게 나타나는데, 컨스턴트풀의 구조는 [그림 2]와 같다. 컨스턴트풀은 테이블 형태로 구성되어서 각각의 엔트리들은 인덱스를 가진다. 컨스턴트풀 내의 많은 엔트리들은 컨스턴트풀의 다른 엔트리에 대한 인덱스 값을 자신의 내용으로 가진다. [그림 2]에서 Class, String, Reference, Name & Type형 엔트리는 모두 다른 컨스턴트풀 엔트리에 대한 인덱스를 참조하는 구조를 가지고 있다. [그림 2]의 회색 부분이 간접 참조되는 부분이다.



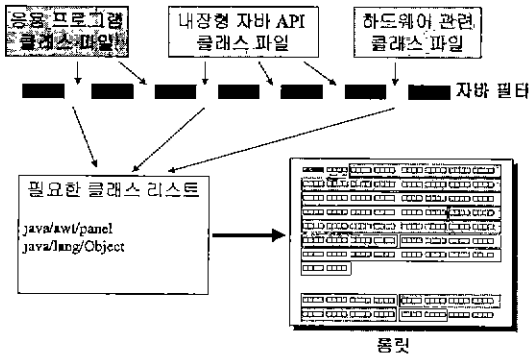
[그림 2] 컨스턴트풀 구조

클래스 파일을 통틀어서 컨스턴트풀 엔트리는 컨스틴트풀 리스트에서 각각의 엔트리의 위치를 알려주는 인덱스에 의해서 참조된다. 실제로 클래스 파일 내에서 클래스나, 필드, 메소드 정보를 접근할 때에는 각각의 이름과 기술자에 대한 컨스턴트풀 인덱스 정보를 가지고 접근하게 된다. 각각의 이름과 기술자에 대한 실제 컨스턴트 값은 일반적으로 UTF8 형 엔트리에서 가지게 된다 이러한 작업은 컨스턴트풀에 대하여 2 회 이상의 접근을 요구한다.

3. 롬릿과 클래스 필터링

내장형 자바 환경에서 자바 코드는 롬릿의 형태로 내장형 기기에 탑재된다. 롬릿은 로딩된 자바 코드를 메모리 덤프 시킨 것이다 롬릿을 사용하므로써 자바 코드의 중복된 메모리 로딩을 줄일 수 있다

롬릿을 구성하기 위해서는 [그림 3]과 같이 응용 프로그램, 내장형 API, 그리고 하드웨어에 관련 클래스 파일 중에서 필요한 클래스 파일에 대한 리스트를 자바 필터를 통해서 얻어낸 후, 이를 메모리에 로딩한다.



[그림 3] 롬릿의 구성

클래스 필터링은 정적 로딩을 수행하기 위해서 응용 프로그램을 수행하는데 필요한 API 리스트를 찾아내기 위한 작업이다. 특정한 응용 프로그램을 수행하기 위한 클래스 파일이 필요로 하는 클래스파일 리스트를 얻기 위해서는 클래스 파일 안에 존재하는 컨스턴트풀이 포함하는 클래스 파일 정보를 참조한다.

컨스턴트풀 안에 있는 Class 형 컨스턴트 정보를 바탕으로 응용 프로그램이 필요로 하는 클래스 파일들의 리스트를 생성하고 또 각각의 클래스 파일들을 수행하는데 필요한 클래스들의 리스트를 재귀적으로 생성함으로서 응용 프로그램 수행에 필요한 모든 클래스 파일의 리스트를 얻어낸다. [그림 4]는 Hello.class라는 클래스 파일에 대하여 자바 필터링을 수행하기 위한 알고리즘을 보여준다.

```
(1) Hello.class를 확인한다. 없다면 에러를 발생한다.
(2) Hello.class를 로딩하고 loadedList에 추가한다.
(3) 로딩된 클래스파일의 컨스턴트를 안에서 참조 클래스 이름
    들을 모두 LoadingQueue에 넣는다
(4) while ( ! IsEmptyQueue ( Queue = LoadingQueue) )
    {
     LoadingQueue에서 클래스파일을 하나 remove() 한다.
     remove()된 클래스파일을 로딩하고 LoadedList에 추가한다.
     컨스턴트풀에서 클래스를 찾아 LoadingQueue에 넣는다.
    }
(5) LoadedList를 프린트한다
```

[그림 4] 클래스 필터링 알고리즘

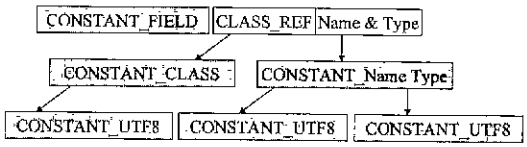
4. 컨스턴트풀 프리레졸루션

컨스턴트풀은 11가지 종류의 형식을 가지는 엔트리로 구성된다. 이 중에서 가장 큰 크기를 차지하는 것은 UTF8 형이다.(문자열 길이 정보 2바이트와 문자열을 가르키는 포인터 4바이트, 총 6바이트) Long과 Double 형 컨스턴트 엔트리는 8바이트를 차지하지만, 이들은 두 개의 컨스탄트 엔트리를 차지하므로 4바이트로도 충분하다.

본 논문에서는 컨스턴트풀의 모든 엔트리를 포함하는 컨스턴트 정보에 대한 데이터 구조를 공용체로 정의한다 그리고 그 공용체의 크기는 메모리 경계정렬을 고려하여 6바이트가 아닌 8바이트로 고정시

킨다. 컨스턴트풀의 모든 엔트리를 하나의 공용체에 포함시킴으로서 컨스턴트풀을 고정 크기의 배열로 나타낼 수 있다. 또한 컨스턴트풀의 종류를 나타내는 태그는 메모리 경계정렬을 고려해 별도의 바이트 배열에 저장한다.

프리레졸루션의 대상이 되는 타입은 Class 컨스턴트 정보와 Field와 Method에 대한 Reference 정보이다. 이들은 궁극적으로는 클래스 파일, 필드, 메소드에 대한 접근을 위한 것이지만 컨스턴트풀 내에는 컨스턴트풀에 대한 인덱스 정보만을 가지기 때문에 여러 번의 간접 참조를 수행해야 한다. [그림 5]는 필드 컨스턴트에 대한 레졸루션을 수행하기 위한 간접 참조 단계를 보여준다.



[그림 5] 필드 컨스턴트의 레졸루션

필드 참조 타입의 컨스턴트풀 엔트리를 접근해서 실제로 필드가 속한 클래스의 이름과 필드 이름 그리고 타입을 얻어내는 과정에서 5번의 컨스턴트풀 접근이 요구되었다 또한 이 정보를 가지고 필드가 저장된 위치를 찾는 작업을 다시 수행하므로서 레졸루션 작업을 마치게 된다 이외 같은 방법으로 간접 참조 횟수를 계산하면 [표 1]과 같은 결과를 얻게 된다. 여기서의 결과는 컨스턴트풀 접근 횟수만 고려하였으며, 로딩된 클래스, 필드, 메소드를 찾아가는데 필요한 메모리 접근 횟수는 고려하지 않았으므로, 실제로는 더 많은 메모리 접근이 이루어지게 된다

[표 1] 컨스턴트 풀 간접 참조 횟수

컨스턴트 종류	간접참조 감소 횟수
Class 형	1회
Refernce 형	5회
Name & Type 형	4회

본 논문에서는 클래스 파일에 대한 정적 로딩을 수행하면서 프리레졸루션을 수행하도록 하고 그 정보를 저장할 수 있는 컨스턴트풀에 대한 자료 구조를 [그림 6]과 같이 제안하였다.

제안된 구조에서 공용체의 구성요소가 되는 각 타입의 컨스턴트 정보는 기본적으로 클래스파일에서 읽어들인 컨스턴트 값을 저장하며, 그외에 컨스턴트 값이 참조를 포함한다면 이에 대한 레졸루션 결과를 함께 저장하도록 하였다. 직접 참조에 대한 정보를 포함하는 타입에 대한 설명은 다음과 같다

▣ CONSTANT_Class

처음 4바이트는 CONSTANT_UTF8 의 스트링에 대한 직접 참조 포인터를 저장하며 두번째 4바이트는 실제로 로딩된 클래스에 대한 포인터를 저장한다

▣ CONSTANT_String

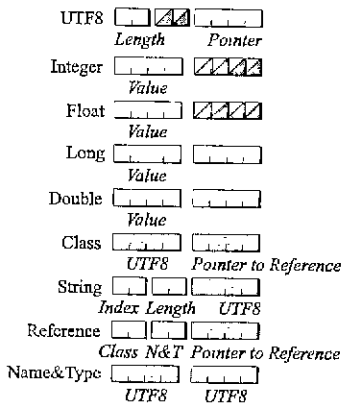
남는 메모리 영역에 CONSTANT_UTF8 타입이 저장된 메모리를 직접 참조할 수 있는 포인터를 저장한다.

▣ CONSTANT_*ref

남는 메모리 영역에 필드, 메소드, 인터페이스 메소드를 직접 참조할 수 있는 포인터를 저장한다.

▣ CONSTANT_NameAndType

컨스턴트풀의 CONSTANT_UTF8 인덱스를 직접 참조할 수 있는 포인터를 저장한다



[그림 6] 제안된 컨스턴트풀 구조

제안된 컨스턴트풀 구조를 사용하면서 프리레졸루션을 수행하므로써 메모리를 더 사용하지 않으면서도, 간접참조를 완전히 없앨 수 있다. 클래스, 필드, 메소드로 직접 찾아갈 수 있으므로, [표 1]에서 계산된 간접 참조 횟수 외에 훨씬 더 많은 메모리 참조를 줄일 수 있다. [그림 6]에서 하얀색 부분은 자바가상머신 명세에 의한 데이터이며, 회색 부분으로 나타나는 것이 레졸루션의 결과 추가 또는 변경된 부분이다. 사선으로 된 부분은 사용되지 않는 영역이다.

5. 결론 및 향후 계획

본 논문에서는 내장형 자바 시스템에서 수행 시간을 단축하기 위한 프리레졸루션 방법과 이를 위한 컨스턴트풀 자료 구조를 제안하였다 본 방법은 컨스턴트풀을 공용체의 고정된 크기의 배열로 설정하며, 사용되지 않는 메모리 부분을 직접 참조로 활용함으로써, 메모리의 추가사용 없이 완전한 레졸루션을 수행할 수 있으며, 수행 시간시 레졸루션 비용을 감소시킬 수 있다 또한 메모리 참조 횟수를 줄임으로써 내장형 시스템에서 필요한 저전력 요구를 지원한다. 향후 벤치마크 프로그램 수행을 통하여 실제 수행 시간이 감소되는 비율을 실험을 통해서 입증할 계획이다

참고 문헌

[1] James Gosling, Bill Joy, Guy Steel, *The Java™ Language Specification* Addison Wesley, 1996

[2] "EmbeddedJava Application Environment Specification." http://java.sun.com/80/products/embeddedjava/spec/, 1999 1

[3] Tim Lindholm, Frank Yellin, *The Java Virtual Machine Specification Second Edition*, Addison Wesley, 1999.