바이트코드 최적화기
(The Bytecode Optimizer)

이 야 리 † 홍 경 표 † 오 세 만 ††
(Yari Lee) (Kyungpyo Hong) (Seman Oh)

요 약 자바언어는 인터넷 및 분산 환경 시스템에서 효과적으로 응용 프로그램을 개발할 수 있도록 설계된 프로그래밍 언어로써 객체지향 패러다임 특성 및 다양한 개발 환경을 지원하고 있다. 그러나, 자바는 클래스 파일이 이동하여 JVM 환경에서 인터프리팅 되는 시스템이므로, 성능의 저하 없이 실행되기 위해서는 효율적인 최적화와 실행 시스템이 요구된다. 본 논문은 네트워크 상에서 동적으로 다운로드 되는 클래스 파일을 바이트코드 수준에서 최적화 하였다. 최적화된 바이트코드들이 인터프리팅 되는 시스템에서 적은 네트워크 로드를 가지고 실행할 수 있도록 하며, 효율적인 실행 속도를 보이도록 하는 것이다.

본 논문에서 구현된 바이트코드 최적화기에서는 내부적으로 바이트코드 최적화기와 클래스 파일 생성기를 이용하여 실행시간을 개선하고 전체 클래스 파일의 크기를 줄이게 된다. 바이트코드 최적화기는 바이트코드를 클래스사이의 계층 분석과 제어 흐름의 분석을 통하여 클래스들간의 연관 관계를 분석한 후 그래프를 구성하고, 패턴 담색 결과 기본 블록 분리를 통하여 전역 최적화를 이루고, 기본 블록 안에서의 연산강도 경감, 그리고 도달할 수 없는 코드 블록의 제거를 수행한다. 바이트코드 최적화 단계를 수행한 클래스 파일은 부분적으로 클래스 파일의 최적화를 가져와 전체 클래스 파일의 크기를 줄이고, 인터프리터를 통하여 실행될 때 수행 속도 면에서 좀더 빠른 실행속도를 가지게 된다.

키워드 : 최적화기, 바이트코드, 클래스 파일, 패턴매칭 최적화

Abstract The Java programming language is designed for developing effective applications in a heterogeneous network environment. Major problem in Java is its performance. Many attractive features of Java make the development of software easy, but also make it expensive to support; applications written in Java are often much slower than their counterparts written in C or C++. To use Java's attractive features without the performance penalty, sophisticated optimizations and runtime systems are required. Optimizing Java bytecode is a widespread topic of many ongoing researches. Developing an optimizer for the Java bytecode has several advantages. First, the bytecode is independent of any compiler that is used to generate it. Second, the bytecode optimization can be performed as a pre-pass to Just-In-Time(JIT) compilation. Many attractive features of Java make the development of software easy, but also make it expensive to support.

The goal of this work is to develop automatic construction of code optimizer for Java bytecode. We' ve designed and implemented a Bytecode Optimizer that performs the peephole optimization, bytecode-specific optimization, and method-inlining techniques. Using the Classfile optimizer, we see up to 9% improvement in speed and about 20% size reduction in java class files, when compared to average code using the interpreter alone.

Key words : Optimizer, Bytecode, Class File, Pattern Matching Optimization

· 본 연구는 한국과학재단 특정기초연구(과제번호 : 1999 1 30300 3)지원으로 수행되었음.
† 비 회 원 : 동국대학교 컴퓨터공학과
yaree@dongguk.edu
hongah@yahoo.co.kr
†† 종신회원 : 동국대학교 컴퓨터공학과 교수
smoh@dgu.ac.kr
논문접수 : 2002년 7월 8일
심사완료 : 2002년 10월 31일

1. 서 론

자바 언어는 이질적인 네트워크 환경에서 효율적인 응용 프로그램을 개발하기 위해 설계되었으며, 다양한 개발환경과 객체지향 속성을 지원한다. 자바 프로그래밍 언어는 이식성과 보안성의 요구에 맞는 언어로써 점점 더 각광을 받고 있다. 최근 연구들에서는 자바언어의 확장과 바이트코드 최적화를 통한 실행 속도 향상에 많은 관심을 두고 있다.

자바 가상기계에서의 구현은 다른 프로그래밍 언어와 비교해 보면 상대적으로 쉽다. 자바는 작은 인터프리티브 언어이기 때문이다. 자바 컴파일러는 JVM 환경에서 실행될 수 있는 바이트코드와 상수 풀이 포함된 중간 형태의 클래스 파일을 생성한다. 이 바이트코드와 클래스 파일의 최적화를 통하여 JVM 환경에서뿐만 아니라 컴파일러 후단부(Back-end)에서 유용하게 사용할 수 있으며, 최적화를 통하여 네트워크 환경에 효율적인 클래스 파일을 제공할 수 있다.

본 논문에서는 자바 클래스 파일이 인터넷 및 분산 환경 시스템에서 효율적으로 실행되기 위해서 자바 컴파일러가 생성한 바이트코드에 대해 최적화를 수행하는 코드 최적화기를 설계하고 구현하였다. 바이트코드에 대한 최적화기 모델을 제시하고, 존재하는 최적화 방법론에 관한 연구를 통하여 자바 바이트코드의 특성을 고려하여 적합한 방법론을 적용하였고, 바이트코드를 위한 코드 최적화기의 자동적 생성에 관하여 설계 및 구현하였다.

2. 관련연구

자바에서의 관심사는 자바 바이트코드와 클래스 파일이다. 바이트코드는 플랫폼에 독립적으로 실행될 수 있지만 인터프리팅되어 실행되는 특징 때문에 다른 언어에 비해 실행속도가 느리다는 단점을 갖는다. 또한, 이들이 네트워크 상에서 동적으로 빠른 전송이 가능하도록 좀더 작은 크기의 클래스 파일이 되어야한다. 이와 같은 문제점을 해결하기 위하여 최적화하는 방법이 시도되는데 직접 바이트코드를 조작하여 최적화하거나 클래스 파일을 조작하는 툴을 사용하는 방법이 있다.

최적화를 위한 여러 가지 툴 중에서 DashO[2]는 자바 응용 프로그램을 위해서 압축기법을 사용하지 않고 작은 크기의 클래스 파일을 만드는 방법을 사용하고 있다. 클래스 파일 내에서 참조되는 패키지(package)를 탐색해 나가면서 오직 필요한 패키지만을 사용하도록 하여 크기를 줄이는 방법을 사용한다. 전체 클래스 파일의 크기를 줄여서 하나의 압축된 파일로 만들어 내어 사용자가 다운로드 하여 사용할 수 있도록 하는 방법을 제공한다.

JOIE[3]는 클래스를 동적으로 로딩하기 위한 기능을 개선하였다. 동적인 클래스 로딩을 위하여 클래스 로더는 자바 프로그램이 실행되는데 중요한 역할을 하고 있다. 하나의 자바 프로그램에서 다른 자바 프로그램을 사용하기 위하여 JVM에서 사용될 수 있도록 로드되어야 하면서 동시에 해제되어야 한다. 이러한 동작을 효율적으로 할 수 있는 방법을 제시하고 있다. 바이트코드 내에서 사용되는 메소드의 분석을 용이하게 하는 방법도 제시되어 있다. 이렇게 사용자가 바이트코드 내의 어디서나 메소드의 분석을 요구할 수 있게 하는 툴에는 Bytecode Instrument Tools[4]가 있다.

바이트코드를 분석하고 최적화하기 위해서 어셈블리 코드와 같은 코드의 분석이 필요하다. Jasmin[5] 어셈블러는 JVM 명령어들을 사용하여 의사 어셈블리 코드를 컴파일하고 바이트코드의 실행을 분석하는데 사용되고 있다. 가와(Kawa)는 바이트코드를 생성하기 위해서 바이트코드를 생성하는 gnu.bytecode package를 포함하고 있다. 또한 이들을 이용한 BCEL[9]의 JavaClass는 바이트코드를 분석한 후 바이트코드의 변형을 통하여 클래스 파일을 생성한다. 컴팩에 의해서 진행된 SWIFT 컴파일러는 입력으로 자바 바이트코드들을 받아서 효율적인 바이트코드를 생성하여 플랫폼에 독립적이고 가능한 한 빠른 코드를 생성한다.

본 논문에서는 자바의 특징적 요소를 사용하기 위하여 여러 가지 접근방법으로서 일반적으로 최적화에 사용되는 최적화 기법과 핍홀 최적화, 전역 최적화 방법들을 사용하여 직접 바이트코드만을 분석 최적화하는 방법을 사용하고 있으며, 두 번째 단계에서는 클래스 상수 풀을 최적화는 하는 방법을 사용하여 최종적으로 JVM에서 실행될 최적화된 클래스 파일을 만들어 낸다.

3. 최적화 바이트코드의 생성

JVM 위에서 실행되는 바이트코드는 기계 독립적인 특징을 가지고 있으므로 바이트코드의 패턴을 분석하여 좀더 바이트코드의 크기를 줄이면서, 실행 효율을 높일 수 있다. 본 절에서는 하나의 클래스 파일로부터 새로운 최적화된 클래스 파일을 생성하는데 있어서 바이트코드 최적화기의 시스템 구성 모델을 제시한다.

바이트코드 최적화기는 자바 클래스 파일을 받아들여 최적화된 클래스 파일을 생성해내는 시스템이다. 첫 번째 단계에서는 바이트코드 최적화기를 통하여 좀더 작은 크기의 바이트코드를 생성하기 위한 최적화를 진행

한다. 두 번째 단계에서는 바이트코드 추출기에 의해 추출된 바이트코드 명령어들과 추가 코드 정보로 JVM에서 실행될 최적화된 클래스 파일을 만드는 과정을 거치게 된다.

바이트코드 최적화기 시스템은 바이트코드 추출기, 코드 최적화기, 그리고 클래스 파일 생성기의 세 부분으로 구성되어 있다.

3.1 최적화 바이트코드 생성 모델

바이트코드를 최적화하기 위해서, 그림 1과 같은 모델을 제안하였으며 바이트코드 최적화와 상수 풀의 최적화에 대한 두 단계로 구성되어 있다. 첫 번째 최적화 단계에서는 바이트코드의 최적화를 통해서 일반적인 최적화와 바이트코드 의존적 최적화를 수행한다. 바이트코드의 길이가 경감되고 실행 속도 면에서 최적화되는 바이트코드를 생성해낸다. 두 번째 단계에서는 생성된 바이트코드들과 상수 풀을 분석하여 실행시간에 사용되지 않는 디버깅 정보를 찾아내어 제거하거나 축약하여 클래스 파일의 크기를 줄이게 된다.

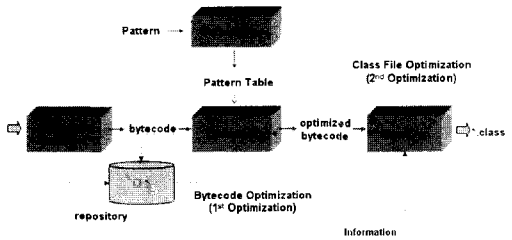

그림 1 바이트코드 최적화기의 구성

그림 1에서는 프로그래머의 자바 프로그램을 자바 컴파일러를 이용해서 생성해낸 클래스 파일을 최적화기의 입력으로 사용되는 것을 보이고 있다. 입력으로 사용된 클래스 파일은 최적화를 위한 자료로 분리해 내기 위하여 바이트코드 추출기와 임시 저장장소(repository)인 CFS(ClassFileStrcture)를 거친다.

3.2 바이트코드 추출기

바이트코드 최적화기에 사용되는 바이트코드 추출기는 최적화기를 위한 전 처리기처럼 사용된다. 모듈의 이름이 명명된 것처럼 바이트코드 추출기는 입력된 클래스 파일의 상수 풀을 분석하여 바이트코드 명령어를 추출해 낸다.

바이트코드 명령어는 자바 명세서(Java Specification)에 정의된 Opcode Mnemonic들의 집합이며, 바이트코드 추출기는 기존의 Jasmin 문법[10]과 유사하게 바이트코드 명령어 집합으로 분리해 내는 모듈이다. 그림 2의 바이트코드 추출기는 Jasmin 문법과 유사한 바이트코드 명령어 집합과 상수 풀 정보를 나타낸다.

그림 2에서 보여주는 것과 같이 바이트코드 추출기는 바이트코드 명령어 집합과 상수 풀의 정보를 다음 단계의 입력으로 사용할 수 있도록 바이트코드 추출기로부터 생성하게 된다.

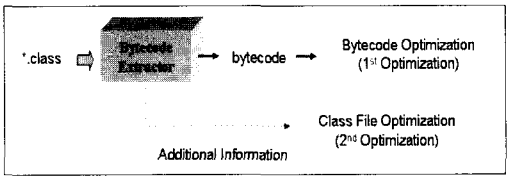

그림 2 바이트코드 추출기

바이트코드 추출기의 입력으로써 클래스 파일을 받아들이며, 그림 3에서 보이는 형태대로 바이트코드 명령어 집합을 생성해 낸다. 추출된 바이트코드 명령어들의 집합은 Opcode Mnemonic을 따르고 있다. 하지만 추출된 자료는 최적화기의 입력으로 사용되는 것이므로 Jasmin Syntax와는 같지 않은 정수 상수 값을 사용하며 그림 3의 2번째 열과 같은 형태의 출력이 요구된다.

Jasmin Syntax	From BytecodeExtractor
ifnull	198
iconst_0	6
goto	167
nop	0
ifeq	167

그림 3 바이트코드 추출기로부터 생성된 코드

추출기에서 생성된 바이트코드 명령어들로부터 실제 코드를 분석하고 최적화될 블록을 찾아내는 부분은 최적화기에 있어서 매우 중요한 부분이다. 바이트코드 추출기로부터 생성된 코드는 패턴 테이블을 생성할 때 사용되는 패턴묘사(pattern description)의 형태와 유사하다. 이는 바이트코드 최적화기의 최적화 구현시 바이트코드 명령어의 최적화될 블록 탐색을 위하여 구성된 것이기 때문이다.

또한 바이트코드 추출기로부터 클래스 파일 생성기에서 필요한 상수 풀 내의 메소드 이름과 변수 이름에 대한 정보를 추가로 추출해 낸다. 이와 같은 방법은 기존 바이트코드 추출기 모델과 다른 형태로 최적화 패턴의 확장성이 가능하여 전체 클래스 파일의 크기와 효율성

```
CONSTANT_Methodref[10] (class_index = 6, name_and_type_index = 15)
CONSTANT_Fieldref[9]   (class_index = 16, name_and_type_index = 17)
CONSTANT_String[8]     (string_index = 18)
CONSTANT_Methodref[10] (class_index = 19, name_and_type_index = 20)
CONSTANT_Class[7]      (name_index = 18)
CONSTANT_Class[7]      (name_index = 21)
CONSTANT_Utf8[1]       ("<init>")
CONSTANT_Utf8[1]       ("()V")
CONSTANT_Utf8[1]       ("Code")
CONSTANT_Utf8[1]       ("LineNumberTable")
CONSTANT_Utf8[1]       ("main")
CONSTANT_Utf8[1]       ("([Ljava/lang/String;)V")
CONSTANT_Utf8[1]       ("SourceFile")
CONSTANT_Utf8[1]       ("HelloWorld.java")
CONSTANT_NameAndType[12](name_index = 7, signature_index = 8)
CONSTANT_Class[7]      (name_index = 22)
CONSTANT_NameAndType[12](name_index = 23, signature_index = 24)
CONSTANT_Utf8[1]       ("HelloWorld")
CONSTANT_Class[7]      (name_index = 25)
CONSTANT_NameAndType[12](name_index = 26, signature_index = 27)
CONSTANT_Utf8[1]       ("java/lang/Object")
CONSTANT_Utf8[1]       ("java/lang/System")
CONSTANT_Utf8[1]       ("out")
CONSTANT_Utf8[1]       ("Ljava/io/PrintStream;")
CONSTANT_Utf8[1]       ("java/io/PrintStream")
CONSTANT_Utf8[1]       ("println")
CONSTANT_Utf8[1]       ("(Ljava/lang/String;)V")
```

그림 4 추출된 상수 풀의 정보

에서 좀더 최적화를 이룰 수 있는 모델이다.

그림 4는 간단한 클래스 파일을 바이트코드 추출기를 통해서 추출해 낸 상수 풀 내의 정보들이다.

3.3 패턴 테이블

패턴 테이블은 입력된 바이트코드 명령어 집합을 향상된 바이트코드로 교체하기 위한 수단으로써 사용한다.

패턴 테이블을 생성하기 위하여 패턴 생성기의 입력으로 패턴 정의 언어가 입력으로 사용되고 그 결과로 바이트코드 최적화기 모듈의 입력으로 사용되는 패턴 테이블이 생성된다. 이 패턴 정의 언어는 바이트코드 명령어의 연속적인 바이트코드 열에서 찾아낸 패턴을 나열한 것으로 다음과 같이 EBNF 형태로 정의한다.

```
if_statement  = ifnull, iconst_0, goto, nop, ifeq | ifnull
```

그림 5 패턴 예(if statement)

위의 패턴은 if_statement에 해당하는 패턴으로 바이트코드 추출기로부터 생성된 바이트코드 명령어 집합으로부터 찾아내며 Opcode number를 열거한 값들의 집합을 패턴의 이름으로 정의한다.

그림 6은 패턴의 묘사를 하고 있으며 찾아낸 패턴의 이름과 변경할 바이트코드 명령어로 구성된다. 패턴 탐색은 바이트코드 패턴 문법으로부터 패턴을 찾아낸다.

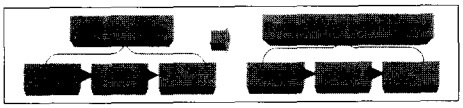

그림 6 패턴 묘사 형태

그림 7은 최적화에 사용될 바이트코드 패턴 테이블을 보여주고 있으며, 그림 8은 바이트코드를 생성하기 위하여 바이트 패턴 코드 문법을 정의하였다.

```
type_push %2  type_push %2          ::= type_push %2 dup
ldc %2 ldc %2                       ::= ldc %2 dup
ldc_w %2 ldc_w %2                   ::= ldc_w %2 dup
ldc2_w %2 ldc2_w %2                 ::= ldc2_w %2 dup2
aload%1 getfield %2 Type[B|C|Z|A] aload%1 getfield %2 B
                                    ::= aload%1 getfield %2 Type [B|C|Z|A] dup
aload%1 getfield %2 L%3; aload%1 getfield %2 L%3;
                                    ::= aload%1 getfield %2 L%3; dup
aload%1 getfield %2 Type[S|Z] aload%1 getfield %2 S
                                    ::= aload%1 getfield %2 Type[S|Z] dup
aload%1 getfield %2 [%3 aload%1 getfield %2 [%3
                                    ::= aload%1 getfield %2 [%3 dup
aload%1 getfield %2 D aload%1 getfield %2 D
                                    ::= aload%1 getfield %2 D dup2
aload%1 getfield %2 J aload%1 getfield %2 J
                                    ::= aload%1 getfield %2 J dup2
                                    ...
```

그림 7 바이트코드 패턴 테이블

```
<bytecode>  ::=     <instructions>
<instructions> ::=  <instruction> <instructions>
<instruction>  ::=  [<stack_op>] [<arth_op>][<arth_op>][<ctrl_flw>]
                    [<ldst_op>]  [<fld_acc>][<method_invoc>][<obj_alloc>]
                    [<conv_type_chk>][<misc>]
<stack_op>  ::=     [<const>] ...
<arth_op>           ::=[<add>] [<sub>] [<mul>] [<div>]
<ctrl_flw>          ::=[<if>] [<goto>] [<jsr>] [<ret>] [<return>][<table>]
<ldst_op>           ::=[<load>][<store>] [<aload>][<astore>]
<fld_acc>           ::=[<get>][<put>]
<method_invoc>      ::=[<invkvtbl>][<invksp>][<invkst>][<vkint>]
<obj_alloc>         ::=[<new>][<newarray>][<anewarray>][<arraylenght>]
<conv_type_chck>    ::=[<conv>][<type>]
<misc>              ::=<nop>
```

그림 8 바이트코드 패턴 문법

패턴 집합들 안의 패턴 이름들은 각각 연산자 Add/Sub/Multiple/Div/Shift/Dec/Inc의 산술연산자와 Assign/New의 연산자, If/If_Else/Do_While/While/Goto/For 등의 제어흐름 문장, Refer와 같은 참조 문장, 그리고 Exceptions와 같은 예외 문장의 이름들이다. 이들 각 패턴들의 이름은 바이트코드 명령어들의 집합 이름이다.

컴파일러는 일관된 바이트코드를 생성하기 때문에 최적의 코드를 생산해 낼 수 없으며 전통적인 최적화 방법과 바이트코드 의존적인 최적화 기술을 통하여 최적화가 이루어진다. 이를 위하여 컴파일러가 생성해 낸 코드의 패턴을 분석하는 단계가 중요하다.

3.4 바이트코드 최적화기

바이트코드 최적화 단계는 바이트코드 최적화기에서 클래스 파일과는 독립적으로 바이트코드만을 가지고 패턴을 검색하여 최적화하는 단계이다. 아래의 그림 19에서 보이는 바이트코드 최적화기는 지역 최적화와 전역

최적화의 코드 최적화가 진행되는 부분과 바이트코드들을 블록으로 분리하여 최적화를 수행할 수 있도록 임시 저장장소 역할을 하는 CFS부분으로 구성된다.

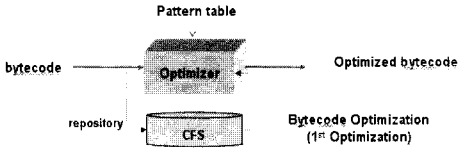

그림 8 1단계 최적화

최적화기의 부분적인 목적은 기존 최적화 방법인 펍홀 최적화에 의하여 자바 컴파일러가 일관성 있게 생성해 내는 코드들을 조작하여 좀더 작은 크기의 바이트코드 집합과 좀더 실행 효율이 높은 코드들로 바꾸어 생성하는데 있다. 바이트코드 의존적 최적화는 본 논문에서 제안하는 CFS 내에서 각 클래스 파일의 구조적 속성들을 분석하여 그림 10과 같은 구조로 만들어서 수행한다.

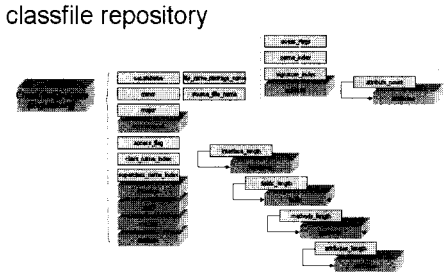

그림 9 CFS 구성도

최적화기의 펍홀 최적화의 경우는 아래의 예 그림 11과 같은 다양한 바이트코드의 경험적 분석을 통하여 패턴을 찾아낼 수 있으며, 이후에 반복적인 실험을 통해서 더 많은 패턴을 정의하여 많은 패턴을 찾아낼 수 있다.

그림 11은 연속된 바이트코드 명령어를 구분해 내어서 최적화 되는 바이트코드 명령어들의 예이다. 스택에 동일한 상수를 삽입(push)하거나 변수를 삽입하는 바이트코드들을 분석하여 최적화된 코드로 변환한 것을 보여주고 있다.

여기서, 코드 의미와 상관없이 바이트코드의 일정한 패턴만을 가지고 그 패턴을 찾아낸 후 좀더 효율성 있는 코드로 바꿀 수 있음을 알 수 있다. 또한, 스트링 연결(string concatenation)의 경우는 자바 컴파일러가 생성한 바이트코드에 생성자가 불필요하게 생성되는 것을 볼 수 있다. 이들을 제거함으로써 컴파일러가 생성한 코드보다 작은 코드를 생성할 수 있다. 하지만 이들은 실제 JVM의 검증기에서 까다로운 검증과정을 거치므로 완전한 바이트코드의 나열이라고 할 수는 없다.

	Original Bytecode code	Modified Code
Constant duplicate	aload_0 bipush bipush invokevirtual pushC/dupTest(II)V	aload_0 bipush_8 dup invokevirtual pushC/dupTest(II)V
Variable duplicate	aload_0 aload_0 getfield pushV/xI aload_0 getfield pushV/kI	aload_0 dup getfield pushV/xI dup
Fetch and store	bipush 9 isotre 4 iload 4 istore_1	bipush 9 Dup istore 4 istore_1

그림 10 최적화되는 바이트코드 명령어 집합

바이트코드 최적화 모듈에 적용되는 최적화 알고리즘은 패턴을 탐색하는 부분과 기존 최적화 알고리즘을 적용하는 부분으로 구성된다. 이 최적화가 진행될 때는 바이트코드의 생성 규칙을 벗어나지 않도록 검증기에서 별다른 문제없이 스택을 조정하면서 바이트코드를 생성해 낼 수 있도록 하는 모듈이 필요하다. 그림 12의 코드 최적화기의 알고리즘은 코드 최적화의 순서를 보여주고 있다. 바이트코드 최적화기의 입력으로 바이트코드 명령어 집합을 사용한다. 또한 패턴 탐색을 통하여 입력된 명령어 집합들을 기본블록으로 구성하고, 찾아낸 기본블록간에는 전역 최적화를 행하고 각 블록 단위의 최적화를 적용한다.

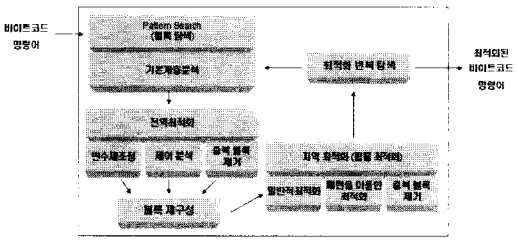

그림 11 코드 최적화 알고리즘

바이트코드 최적화기 모듈의 주요 행동은 일반적인 최적화의 적용 후에 독립적인 코드들의 분석 최적화(최적화 패턴으로의 교체)와 지역 변수의 재조정이다. 독립

적인 코드들의 분석 최적화에 있어서는 사용되는 바이트코드를 다른 코드로 변경함으로써 실행시간의 단축과 바이트코드의 길이를 줄이게 된다. 이러한 것들은 최적화될 코드를 미리 분석하여 테이블을 가지게 된다. 예를 들면 생성된 NOP 명령어는 제거하여도 실행에는 관계되지 않으므로 제거하여 최적화한다. 지역 변수의 조정에서는 스택 기반에서 운용되는 바이트코드들이 명령어가 변수 슬롯을 사용하는 위치에 따라 바이트코드의 길이에 영향을 미치는 것을 이용하여 자주 사용하는 변수의 할당을 0~3번지로 옮겨서 짧은 길이의 코드를 사용할 수 있다.

이렇게 최적화된 바이트코드가 생성되면 이들은 검증과정을 거치게 되어 불합리한 참조가 이루어 질 수 있다. 이러한 문제점을 해결하고 상수 풀의 최적화를 위하여 두 번째 단계의 최적화 루틴을 둔다.

바이트코드 최적화기는 바이트코드를 클래스사이의 계층 분석과 제어 흐름의 분석을 통하여 클래스들간의 연관 관계를 분석한 후 그래프를 구성하고, 패턴 탐색 결과 기본 블록 분리를 통하여 전역 최적화를 이루고, 기본 블록 안에서의 연산강도 경감, 그리고 도달할 수 없는 코드 블록의 제거를 수행한다. 또한, 바이트코드 의존적인 최적화로서 변수 할당과 같은 최적화와, dup 명령어의 사용을 통한 최적화를 행하게 된다.

3.5 클래스 파일 생성기

클래스 파일 생성기는 전체 바이트코드 최적화기의 두 번째 최적화 단계이다.

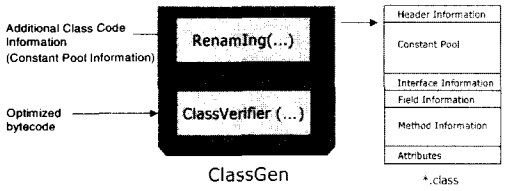

그림 12 클래스 파일 생성기

그림 13에서는 JVM의 명령어 집합들인 최적화된 바이트코드를 입력으로 바이트코드 명령어가 1:1로 코드 맵핑을 하게 되면서 순서대로 클래스 파일을 생성해 나간다. 바이트코드 추출기에서 추출된 추가 코드정보는 변수 이름과 메소드 이름의 테이블을 가지고 있으며 Renaming의 방법에 있어서는 알파벳 문자를 오름차순으로 정리하여 명명하고, 클래스 파일 검증기에서 클래스 파일 생성 규칙에 어긋나지 않도록 검사한다.

그림 14는 바이트코드 최적화기를 거친 클래스 파일을 나누어 보여준다. 바이트코드로부터 생성된 클래스 파일 자신은 클래스 파일을 생성하는데는 문제가 없다. 하지만, 추가 코드 정보를 통해서 바이트코드에서 클래스 파일로 변환하는 동안 변수명을 축약하고 클래스 파일을 생성하여 전체 클래스 파일의 크기를 줄여 나가게 된다.

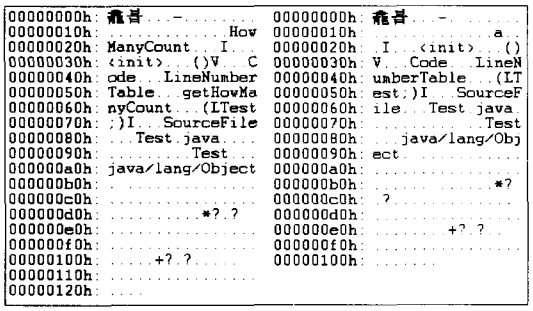

그림 13 원시 클래스 파일과 최적화된 클래스 파일

4. 실험 결과

본 논문의 바이트코드 최적화기의 실험은 실행속도와 바이트코드 크기의 변화를 실험 결과로 사용한다. 실험 환경으로는 Intel Pentium-IV 1.8A, RAM 512MB, Windows XP, j2sdk version1.4를 사용하였다.

실험 대상 데이타는 스탠포드 벤치마크 프로그램(Stanford Benchmark program)과 기타 일상적인 자바 애플리케이션 프로그램들을 무작위로 추출하여 약 200여 개의 예제 데이타들을 사용하였다. 전체 데이타들 중 그림 14에서 나타낸 실험 데이타 perfectTest.java, bubbleTest.java, matmulTest.java 등은 완전수와 버블 소팅, 행렬 곱을 실행하는 대표적 벤치마크 테스트용 프로그램으로 실험을 위하여 *millisec*단위로 측정할 수 있는 타이머 클래스(Timer class)를 구성하였다. 그밖에 다른 데이타들에서도 바이트코드 크기의 경감과 클래스 파일 전체 크기의 경감을 통한 비교 실험을 하였다. 특히, 그림 15에서 사용된 데이타들은 비교적 고른 최적화 경감율을 나타내었다.

그림 14과 그림 15의 비교테이블은 자바 컴파일러를

Stanford Benchmark(*.java)	Code size[byte]		Execution Time[msec.]	
	Original	Optimized	Original	Optimized
perfectTest	488	474	4,887	4,814
bubbleTest	848	624	5,778	5,781
matmulTest	716	652	1,932	1,913

그림 14 코드크기와 실행속도 비교테이블

(*.java)	Code size[byte]		Execution Time[msec.]	
	Original	Optimized	Original	Optimized
ExplicitTypeConversion	92	49	21.1	19.7
InfinityArithmetic	161	65	1178.6	977.3
LogicalOperators	102	102	17	19
LosePrecision	38	18	9	8
NarrowingTypeConversion	103	103	88	78.2
RelationalOperators	154	98	36.9	22.1
RemainderOperator	58	24	17	11.1
BreakSt	81	81	38	39
Calculator	272	272	35.2	44
BitOperators	147	137	44.1	43
ShiftOperators	116	36	134.4	128.1
RuntimeClass	88	88	30	26.2
ThreadExample	77	35	101.2	99

그림 15 코드크기와 실행속도 비교테이블

사용하여 생성된 클래스 파일과 최적화기를 통해 생성된 최적화된 파일을 분석한 것이다. 분석 결과를 막대 그래프로 표현한 그림 16과 그림 17은 바이트코드 최적화를 통하여 최적화된 실험 결과로써 소스 파일과 최적화된 파일을 함께 나타냄으로써 그림 16은 바이트코드 크기 경감율, 그림 17은 실행속도 경감율의 향상을 보여준다.

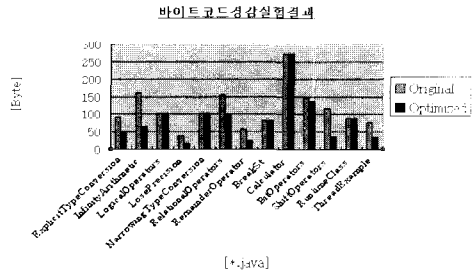

그림 16 바이트코드 크기 경감율

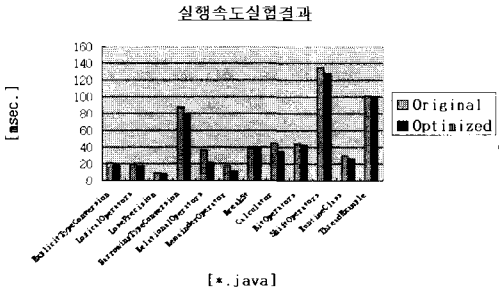

그림 17 실행속도 경감율

실험 결과 분석에서 실험에 이용되었던 최적화된 클래스 파일들은 코드 크기 면에서 약 20~30% 경감의 결과를 보여주었다. 그림 18의 분석 사례는 본 논문의 최적화 바이트코드 생성기를 통한 바이트코드 크기 경감(1단계 최적화)과 클래스 Obfuscation(2단계 최적화) 즉, 클래스 파일명과 변수명을 생략하여 얻은 것이고, 또한 실행에 관계하지 않는 클래스 정보인 'Source filename', 'LineNumberTable'의 제거를 통한 결과를 보여 주었다.

실행 속도 면에서도 각 단위 예제에 대한 100회 반복 실험을 통하여 약 9~10% 정도의 향상을 나타내었다.

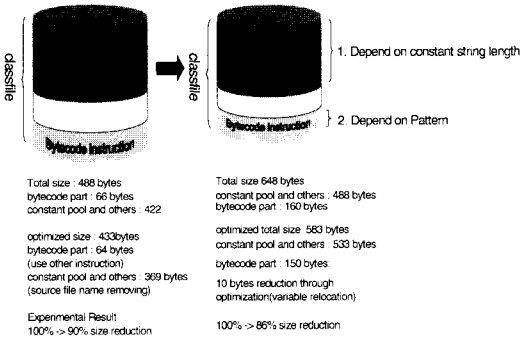

그림 18 바이트코드 최적화기를 통한 실험 결과 분석

5. 결론 및 향후 연구 방향

본 논문은 바이트코드에 대한 최적화기를 구현하기 위하여 두 단계로 최적화 단계를 구분하였다. 첫 번째 단계인 바이트코드에 대한 최적화기에서는 바이트코드에 대한 각 명령어의 특성 및 JVM에서 실행되는 동작을 분석하고, 바이트코드에 대한 최적화 기법들을 이용하여 공통식의 제거, 연산 강도 경감, 루프 불변 코드 이동 등과 같은 최적화를 진행하였다. 특히 바이트코드에 의존적 최적화 기법인 변수 할당, 스택에 관련된 중복을 행하는 dup 최적화를 이용하였다. 이 최적화 단계에서는 코드에 대한 최적화 패턴을 찾아내고, 코드 최적화 자동화에 관한 알고리즘을 제안하여 바이트코드 최적화를 이룬다. 바이트코드 최적화기에는 바이트코드를 최적화 하기 위해 패턴을 탐색하는 알고리즘과 패턴을 정의한 테이블을 이용한다. 더 많은 패턴을 찾아낼 수록 더 많은 최적화된 바이트코드로써 나타난다.

두 번째 단계인 바이트코드 최적화기에서는 상수 풀에서 사용되지 않는, 또는 참조되지 않는 테이블과 엔트리를 제거하여 코드 사이즈 경감에 있어서 최적화를 시도하였다.

본 논문의 바이트코드 최적화기는 자바 프로그램의 성

능을 향상시키기 위해서 보편적으로 사용되는 코드 최적화 알고리즘과 바이트코드에만 있는 특정 바이트코드로의 교체를 통하여 최적화하였고, 상수 풀 중에 실행에 관계되지 않는 디버깅 정보를 제거하여 최적화 하였다.

실험결과에 사용되는 실험 데이타가 되는 사용자 자바 프로그램의 클래스 파일에는 데이타로써 여러 가지 한계가 있다. 원시적으로 최적화하여 가며 코딩을 하는 로직을 가진 프로그래머의 프로그램이 실험 데이타로써 사용될 때와 그렇지 않은 프로그램 사이에는 현저한 성능 차이가 있다. 현재 바이트코드 최적화기는 총 120여 개의 최적화를 위한 패턴을 탐색하여 테이블에 기술하며 상수 풀에 대한 분석을 통하여 고정적으로 제거 대상이 될 수 있는 클래스 정보 중 소스 파일 이름을 삭제한다. 동적 링크되는 클래스 파일들간의 연결에 관하여서는 연구를 진행 중이다.

본 논문의 향후 연구 방향은 바이트코드 최적화기에 사용되는 코드 최적화 알고리즘을 보완하고 최적화된 바이트코드에 있어서 사이즈 경감과 성능(실행속도)의 trade off에 대한 정량적 분석을 이루어서 클래스 파일을 최적화하는데 있다.

참 고 문 헌

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, 1985.

[2] DashO, preEmptive solution Inc. http://www.preemptive.com/

[3] Geoff Cohen,Prof. Jeff Chase,Prof. Sid Chatterjee (UNC) John Bley , Geoff Berry, Dr. David Kaminsky (IBM), JOIE, The Java Object Instrumentation Environment.

[4] Han Bok Lee and Benjamin G. Zorn BIT: A Tools for Instrumenting Java Bytecodes. In Proceedings USENIX Symposium on Internet Technologies and Systems,1998.

[5] Jonathan Meyer, Jasmin - Jasmin Assembler, http://www.cat.nyu.edu/meyer/jasmin/guide.html

[6] Ken Arnold, James Gosling, The Java Program ming Language, Second Edition., Addison Wesley, 1998.

[7] Mahadevan Ganapathi, Charles N. Fisher, and John L. Hennessy, Retargetable Compiler Code Generation, ACM Computing Surveys, vol.14, no.4, pp.573-593, 1982.

[8] Make Java fast:Optimize!. http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html.

[9] M. Dham. Byte Code Engineering with the JavaClass API. University Berlin, 1998 http://bcel.sourceforge.net/

[10] OpenJIT homepage http://www.openjit.org.

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

[12] Yari Lee, Jungsook Kim, Kyongpyo Hong, Seman Oh, Design and Implementation of a Java Bytecode Optimizer, Proceedings of the IASTED International Conference Internet and Multimedia Systems and Applications, Hawaii, pp.62-66, August 2001.

[12] 고광만, 오세만, 트리 패턴 매칭 코드 생성 알고리즘, 한국정보과학회 논문지(B), 25권 10호, pp.1566-1574, 1998. 10.

[13] 이야리, 오세만, 자바 바이트코드 최적화기, 한국정보과학회 프로그래밍언어연구회 논문지, 제15권 제2호, pp.9-18, 2001.11.

[14] 홍경표, 이야리, 오세만, 자바 클래스 파일 최적화, 한국정보과학회 봄 학술 발표논문집, 제28권 제1호, pp.55-57, 2001.4.

이 야 리

1986년~1990년 고려대학교 전자전산공학과(학사). 2000년~2001년 경인여자대학 멀티미디어 정보전산학부 전임강사. 1999년~2002년 8월 동국대학교 컴퓨터공학과(박사). 관심분야는 컴파일러, 프로그래밍 언어, XML

홍 경 표

1993년~1997년 동국대학교 컴퓨터공학과(학사). 2000년~2002년 동국대학교 컴퓨터공학과(석사). 관심분야는 모바일 언어, 프로그래밍 언어, XML

오 세 만

1993년~1999년 동국대학교 컴퓨터공학과 대학원 학과장. 1985년~현재 동국대학교 컴퓨터공학과 교수. 관심분야는 모바일 컴퓨팅, XML, 프로그래밍 언어, 컴파일러