자바 바이트코드 최적화기†
(A Java Bytecode Optimizer)

이 야 리, 오 세 만
동국대학교 컴퓨터공학과
yaree@dongguk.edu, smoh@dgu.ac.kr

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, 인터프리티브 방식으로 실행되기 때문에 성능상 많은 제약이 따르게 된다. 즉 자바는 클래스 파일이 이동하여 JVM 환경에서 인터프리팅 되는 시스템이므로, 성능의 저하 없이 실행되기 위해서는 효율적인 최적화와 실행 시스템이 요구된다. 본 논문의 목적은 네트워크 상에서 동적으로 다운로드 되는 클래스 파일의 최적화에 있다. 클래스 파일이 인터프리팅 되는 시스템에서 보다 적은 네트워크 로드를 가지고 실행할 수 있도록 하며, 효율적인 실행 속도를 보이도록 하는 것이다.

본 논문에서 구현된 자바 바이트코드 최적화기에서는 내부적으로 바이트코드 최적화기와 클래스 파일 생성기를 이용하여 실행시간을 개선하고 전체 클래스 파일의 크기를 줄이게 된다. 바이트코드 최적화기는 핍홀 최적화를 수행하고, 바이트코드 의존적 최적화, 그리고 전역 최적화를 행하게 된다. 클래스 파일 생성기는 클래스 파일의 포맷에 따라 바이트코드를 분석하고 본래의 클래스 파일보다 작은 크기의 클래스 파일을 생성하게 된다. 최적화된 클래스 파일은 부분적으로 클래스 파일의 최적화를 가져와 전체 클래스 파일의 크기를 줄이고, 인터프리터를 통하여 실행될 때 수행 속도 면에서 좀더 빠른 실행속도를 가지게 된다.

1. 서론

자바언어는 인터넷 및 분산 환경 시스템에서 플랫폼에 독립적인 바이트코드를 사용한다. 네트워크를 통해 전송된 바이트코드를 수행하는 플랫폼의 후단부(Back-End)에서 바이트코드의 실행 속도는 상당히 중요하다. 그러나, 자바 컴파일러는 각 플랫폼에 독립적인 중간코드 표현에 제약을 받기 때문에 효율적인 코드를 생성하는데 한계가 있을 뿐만 아니라, 바이트코드는 스택을 기반으로 하는 가상기계 언어로서의 단점 또한 가지고 있다. 따라서 바이트코드가 네트워크와 같은 실행 환경에서 효과적으로 실행되기 위해서

† 이 논문은 한국과학재단의 특정기초연구(과제번호:1999-1-30300-3)지원에 의한 것임.

는 효율적인 바이트코드에 대한 최적화 변환이 필요하다.

본 논문에서는 자바 바이트코드가 인터넷 및 분산환경 시스템에서 효율적으로 실행되기 위해서 자바 컴파일러가 생성한 바이트코드에 대한 최적화기를 설계 및 구현하였고 실험결과와 향후 연구에 대하여 논한다.

2. 관련연구

자바 컴파일러는 JVM 환경에서 실행될 수 있는 바이트코드와 상수 풀이 포함된 중간 형태의 클래스 파일을 생성한다. 바이트코드는 플랫폼에 독립적으로 실행될 수 있지만 인터프리팅 되어 실행되는 특징 때문에 다른 언어에 비해 실행속도가 느리다는 단점을 갖는다. 또한 이들이 네트워크 상에서 동적으로 빠른 전송이 가능하도록 좀더 작은 크기의 클래스 파일이 되어야 한다. 이와 같은 문제점을 해결하기 위하여 최적화하는 방법이 시도되는데 직접 바이트코드를 조작하여 최적화하거나 클래스 파일을 조작하는 도구를 사용하는 방법이 있다.

자바 바이트코드가 인터프리티브 방식으로 실행됨으로써 발생하는 실행 속도 문제를 해결하기 위한 연구에는 JIT 컴파일링[10] 방식이 있다. 이 방식은 처음 실행하는 코드에 대해서 동등하게 변환된 네이티브(native) 코드를 실행함으로써 실행 속도를 높여 준다. 또 다른 연구로써 Hotspot[11] 컴파일링 방식에서는 프로그램의 임계 코드 부분만을 탐지하여 최적화를 실행하는 방법을 사용하여 실행 속도를 기존의 C/C++와 유사한 정도로 개선하였다.

최적화를 위한 여러 가지 도구 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. 즉, 전체 클래스 파일의 크기를 줄여서 하나의 압축된 파일로 만들어 내어 사용자가 다운로드 하여 사용할 수 있도록 하는 방법을 제공한다. 또한, 동적인 클래스를 위한 클래스 로더를 증대하는데 사용한 JOIE[3]가 있으며 사용자가 바이트코드 내의 어디서나 메소드의 분석을 요구할 수 있는 Bytecode Instrument Tools[5]가 있다. Jasmin[6] 어셈블러는 의사 어셈블리 코드를 컴파일하는데 사용되고, Kawa는 자바 바이트코드를 생성하는 gnu.bytecode package를 포함하고 있다. 또한 이들을 이용한 BCEL[9]의 JavaClass는 바이트코드의 분석을 통하여 나온 바이트코드를 변형시켜서 클래스 파일을 생성한다.

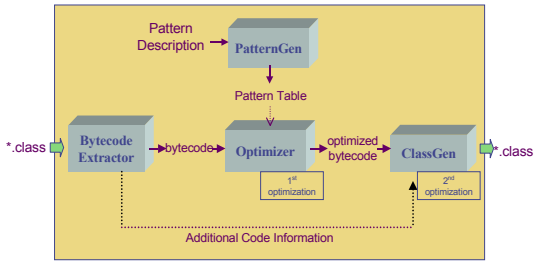
본 논문에서 적용한 최적화 방법은 두 단계이다. 첫 번째 단계의 최적화는 일반적으로 최적화에 사용되는 최적화 기법과 핍홀 최적화, 전역 최적화 방법들을 사용하여 직접 바이트코드를 분석, 최적화하는 방법을 사용하였으며, 두 번째 단계에서는 클래스 파일 정보를 최적화하여 최종적으로 JVM에 로드될 최적화된 클래스 파일을 만들어 낸다.

3. 자바 바이트코드 최적화기

자바 바이트코드 최적화기 시스템은 자바 클래스 파일을 받아들여 최적화된 클래스 파

일을 생성해내는 시스템이다. 첫 번째 단계의 바이트코드 최적화기를 통하여 좀더 작은 크기의 바이트코드를 생성하기 위한 최적화를 진행한다. 두 번째 단계에서는 바이트코드 추출기에 의해 추출된 바이트코드 명령어들과 클래스 파일 정보로 JVM에서 실행될 최적화된 클래스 파일을 만드는 과정을 거치게 된다. [그림 1]은 자바 바이트코드 최적화기의 구성을 나타내며, 바이트코드 추출기(Bytecode Extractor), 최적화기(Optimizer), 그리고 클래스 파일 생성기(ClassGen)의 세 부분으로 구성되어 있다.

바이트코드 최적화기는 바이트코드와 패턴 테이블을 입력으로 사용하며, 바이트코드 명령어들의 집합을 탐색하면서 바이트코드의 패턴을 분석한다. 클래스 파일 생성기는 바이트코드 최적화기를 통하여 생성된 최적화된 바이트코드와 바이트코드 추출기에서 추출된 클래스 파일 정보를 입력으로 클래스 파일 생성 규칙을 통하여 보다 작은 클래스 파일을 생성하게 된다.



[그림 1] 자바 바이트코드 최적화기의 구성

3.1 바이트코드 추출기

바이트코드 추출기는 최적화기를 위한 전처리기(Preprocessor)처럼 사용되며, 입력된 클래스 파일을 분석하여 바이트코드 명령어를 추출해 낸다.

바이트코드 명령어는 Java Specification에 정의된 Opcode Mnemonic들의 집합이며, 바이트코드 추출기는 기존의 Jasmin Syntax[5]와 유사하게 바이트코드 명령어 집합과 클래스 파일 정보를 추출한다.

[그림 1]에서 보여주는 것과 같이 바이트코드 추출기의 입력은 클래스 파일이며, [그림 2]에서 보이는 형태대로 바이트코드 명령어 집합을 생성해 낸다. 추출된 바이트코드 명령어들의 집합은 Opcode Mnemonic을 따르고 있다. 하지만 추출된 자료는 최적화기의 입력으로 사용되는 것이므로 Jasmin Syntax와는 다른 정수 상수 값을 사용한다.

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

[그림 2] 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 패턴 묘사(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 클래스 파일 정보를

추가로 추출해 낸다. 이와 같은 방법은 기존 바이트코드 추출기 모델과 다른 형태로 최적화 패턴의 확장성이 가능하여 전체 클래스 파일의 크기와 효율성 면에서 좀 더 많은 최적화를 이룰 수 있는 모델이다.

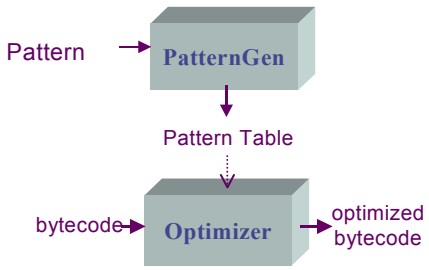
[그림 3]은 간단한 클래스 파일을 바이트코드 추출기를 통해서 추출해 낸 상수 풀 내의 정보들이다.

```
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 3] 바이트코드 추출기를 통해 추출한 상수 풀의 정보

3.2 바이트코드 최적화기

바이트코드 최적화기는 클래스 파일과는 독립적으로 바이트코드를 가지고 패턴을 검색하여 최적화하는 단계이다. 아래의 [그림 4]에서 보이는 바이트코드 최적화기는 지역 최적화와 전역 최적화의 코드 최적화가 진행되는 부분이다. 최적화기의 부분적인 목적은 기존 최적화 방법인 핍홀 최적화에 의하여 자바 컴파일러가 일관성 있게 생성해 내는 코드들을 조작하여 좀더 작은 크기의 바이트코드 집합과 좀더 실행 효율이 높은 코드들로 바꾸어 생성하는데 있다.



[그림 4] 바이트코드 최적화기

최적화기의 핍홀 최적화의 경우는 아래의 예 [그림 5]와 같은 다양한 바이트코드의 경험적 분석을 통하여 패턴을 찾아낼 수 있으며, 이후에 반복적인 실험을 통해서 더 많은 패턴을 정의하여 보다 다양한 패턴을 찾아낼 수 있다.

[그림 5]는 연속된 바이트코드 명령어를 구분해 내어서 최적화되는 바이트코드 명령어들의 예이다.

	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

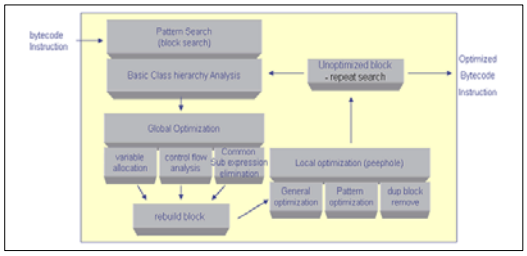
[그림 5] 최적화되는 바이트코드 명령어 집합

위의 [그림 5]는 스택에 동일한 상수를 push하거나 변수를 push하는 바이트코드들을 분석하여 최적화된 코드로 변환된 것을 보여주고 있다. [그림 5]에서는 코드 의미와 상관없이 일정한 바이트코드 패턴을 가지고 그 패턴을 찾아낸 후 좀더 효율성 있는 코드로 바꿀 수 있음을 알 수 있다. 자바 컴파일

러가 생성한 바이트코드는 프로그램에서 특정 패턴이 사용되었을 때 바이트코드에 생성자가 불필요하게 생성되는 것을 볼 수 있다. 바이트코드 최적화기에서 이들을 제거함으로써 컴파일러가 생성한 코드보다 작은 코드를 생성할 수 있다. 하지만 제거된 바이트코드는 실제 자바 가상 머신의 검증기에서 까다로운 검증 과정을 거치므로 완전한 바이트코드의 나열이라고 할 수는 없다.

바이트코드 최적화기는 패턴을 탐색하는 부분과 기존 최적화 알고리즘을 적용하는 부분으로 구성된다. 또한 바이트코드 최적화가 진행될 때는 바이트코드의 생성 규칙에 위배되지 않도록 검증기에서 별다른 문제없이 스택을 조정하면서 바이트코드를 생성해 낼 수 있도록 하는 모듈이 필요하다.

[그림 6]의 코드 최적화 알고리즘은 코드 최적화의 순서를 보여주고 있다. 바이트코드 최적화기의 입력으로 바이트코드 명령어 집합을 사용한다. 또한 패턴 탐색을 통하여 입력된 명령어 집합들을 기본 블록으로 구성하고, 찾아낸 기본 블록간에는 전역 최적화를 행하고 각 블록 단위의 최적화를 적용한다.



[그림 6] 코드 최적화 알고리즘

바이트코드 최적화기의 주요 행동은 일반적인 최적화의 적용 후에 독립적인 코드들의 분석 최적화(최적화 패턴으로의 교체)와 지역 변수의 재조정이다. 독립적인 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 옮겨놓는 것이다.

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

바이트코드 최적화기는 바이트코드를 클래스 사이의 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.

바이트코드 최적화기는 클래스 파일과는 독립적으로 바이트코드를 입력으로 패턴을 검색하여 최적화하는 단계이다. 컴파일러는 일관된 바이트코드를 생성하기 때문에 최적의 코드가 될 수 없으며 전통적인 최적화 방

법과 바이트코드 의존적인 최적화 기술을 통하여 최적화가 이루어진다.

3.3 패턴 테이블

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

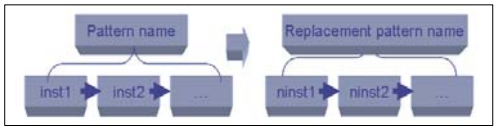
패턴 테이블을 생성하기 위하여 패턴 생성기의 입력으로 패턴 정의 언어가 입력으로 사용되고 그 결과로써 바이트코드 최적화기 모듈의 입력으로 사용되는 패턴 테이블이 생성된다. 이 패턴 정의 언어는 바이트코드 명령어의 연속적인 바이트코드 열에서 찾아낸 패턴을 나열한 것으로 다음 [그림 7]의 예와 같이 EBNF 형태로 정의한다.

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

[그림 7] 패턴 예(if statement)

위의 패턴 예는 if_statement에 해당하는 패턴으로 바이트코드 추출기로부터 생성된 바이트코드 명령어 집합으로부터 찾아내며 Opcode number를 열거한(enumeration) 값들의 집합을 패턴의 이름으로 정의한다.

[그림 8]은 패턴의 묘사를 하고 있으며 찾아낸 패턴의 이름과 변경할 바이트코드 명령어로 구성된다. 패턴 탐색은 바이트코드 패턴 문법으로부터 패턴을 찾아낸다.



[그림 8] 패턴 묘사 형태

[그림 9]는 최적화에 사용될 바이트코드 패턴 테이블을 보여주고 있으며, [그림 10]은 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 9] 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>          ::=[<invkvtl>][<invksp>][<invkst>][<ivkint>]
<obj_alloc>             ::=[<new>][<newarray>][<anewarray>][<arraylenght>]
<conv_type_chck>        ::=[<conv>][<type>]
<misc>                  ::=<nop>      ...
```

[그림 10] 바이트코드 패턴 문법

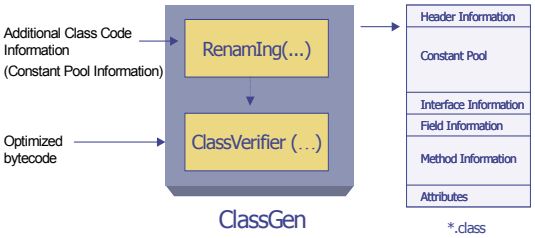
패턴 집합들 안의 패턴의 이름들은 각각 연산자 : Add / Sub / Multiple / Div / Shift / Dec / Inc의 산술연산자와 Assign / New의 연산자, If / If_Else / Do_While / While / Goto / For등의 Control Flow, Refer와 같은 Reference, Exceptions 문장들의 이름들이다. 이들 각 패턴들의 이름은 바이트코드 명령어들의 집합 이름이다.

컴파일러는 일관된 바이트코드를 생성하기 때문에 최적의 코드를 생산해 낼 수 없으며 전통적인 최적화 방법과 바이트코드 의존적인 최적화 기술을 통하여 최적화가 이루어

진다. 이를 위하여 컴파일러가 생성해낸 코드의 패턴을 분석하는 단계가 중요하다.

3.4 클래스 파일 생성기

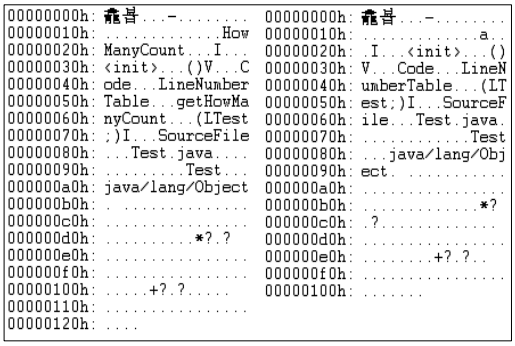
클래스 파일 생성기는 바이트코드 최적화기의 두 번째 최적화 단계를 실행한다.



[그림 11] 클래스 파일 생성기

[그림 11]은 클래스 파일 생성기를 보여주고 있다. 클래스 파일 생성기는 바이트코드 최적화기로부터 생성된 바이트코드 명령어 집합과 클래스 파일 정보를 입력으로 사용한다. 입력된 클래스 파일 정보와 바이트코드 명령어들로부터 클래스 파일 생성 규칙에 위배되지 않도록 클래스 파일을 생성한다. 바이트코드 추출기에서 추출된 클래스 파일 정보는 변수 이름과 메소드 이름의 테이블을 가지고 있으며 클래스 파일 생성기는 참조되지 않는 속성(Attribute)을 제거함으로써 제거된 속성이 클래스 파일 검증기에서 클래스 파일 생성 규칙에 어긋나지 않는지 검사한다.

[그림 12]는 실험에서 사용한 원시 클래스 파일과 바이트코드 최적화기를 거친 클래스 파일을 나누어 보여준다. 클래스 파일 생성기는 클래스 파일 정보를 통하여 바이트코드에서 클래스 파일로 변환하는 동안 필드명을 축약하고 실행에 관련 없는 디버깅 정보를 제거하여 전체 클래스 파일의 크기를 줄여 나가게 된다.



[그림 12] 원시 클래스 파일과 최적화된 클래스 파일

4. 실험 결과

본 논문의 자바 바이트코드 최적화기의 실험은 실행속도와 클래스 파일 크기의 변화를 실험결과로 사용한다. 실험 환경으로는 Intel Pentium-III 750 MHz, RAM 512MB, Windows XP, JDK1.3.1을 사용하였다.

실험 대상 데이터로써는 스탠포드 벤치마크 프로그램과 일반적인 자바 프로그램들(perfectTest.java, bubbleTest.java, matMulTest.java)을 사용하였다. 본 논문의 실험에서 사용한 프로그램 perfectTest.java, bubbleTest.java 등은 완전수와 버블 소팅을 하는 프로그램으로써 실험을 위하여 *millisec* 단위로 측정할 수 있는 Timer class를 구성하였다. 각각 자바 컴파일러를 사용하여 생성된 클래스 파일과 최적화기를 통해 생성된 최적화된 파일을 분석한 것이다.

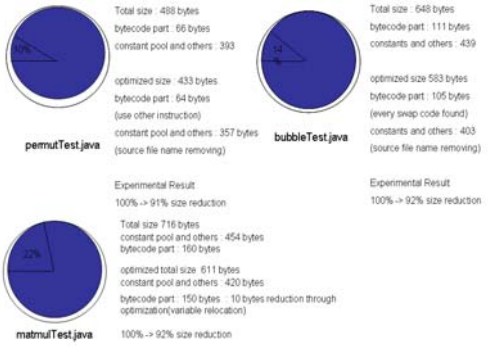
[그림 13]은 자바 바이트코드 최적화기를 통하여 최적화된 실험 결과를 보여준다.

Benchmark program	Code size (byte)		Execution time(sec)		Etc.
	Original	optimized	Original	optimized	
perfectTest.java	488	433	4.887	4.814	(2000이하의 수)
bubbleTest.java	648	583	5.778	5.761	(data개수:2000)
matmulTest.java	716	611	1.932	1.913	(300*300 행렬)

[그림 13] 바이트코드 최적화기를 통한 실험 결과표

[그림 13]의 실험 결과표에서는 바이트코드 최적화기의 실험 결과를 나타내었다. 바이트코드 최적화기로부터 생성된 프로그램을 인터프리터에서 실행했을 때 실행 속도는 미세한 차이의 결과를 보여주었으나 파일 크기면에서 좋은 결과를 보여주었다.

[그림 14]는 [그림 13]의 실험 결과 데이터를 분석한 것이다.



[그림 14] 바이트코드 최적화기를 통한 실험 결과 분석

5. 결론 및 향후 연구 방향

본 논문은 자바 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 자바 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 사용한다. 더 많은 패턴을 찾아낼 수록 더 많은 최적화된 바이트코드들이 나타난다.

두 번째 단계인 클래스 파일 생성기에서는 사용되지 않는, 또는 참조되지 않는 테이블과 엔트리를 제거하여 코드 사이즈 경감에 있어서 최적화를 시도하였다.

본 논문의 자바 바이트코드 최적화기는 자바 프로그램의 성능을 향상시키기 위해서 보편적으로 사용되는 코드 최적화 알고리즘과 바이트코드 의존적 최적화를 하였고, 클래스 파일 정보 중에 실행에 관계되지 않는 디버깅 정보를 제거하여 최적화 하였다.

실험결과에 사용되는 실험 데이터가 되는 사용자 자바 프로그램의 클래스 파일에는 데이터로써 여러 가지 한계가 있다. 소스 코드 단계에서 최적화된 프로그램이 실험 데이터로써 사용될 경우와 그렇지 않은 프로그램이 사용될 경우 사이에서 바이트코드 최적화기는 현저한 성능 차이를 보인다. 현재 자바 바이트코드 최적화기는 총 120여 개의 최적화를 위한 패턴을 탐색하여 테이블에 기술하며 클래스 파일의 속성에 대한 분석을 통하여 실행에 관계되지 않는 디버깅 정보를 삭

제한다.

본 논문의 향후 연구 방향은 자바 바이트코드 최적화기에 사용되는 코드 최적화 알고리즘을 보완하고 최적화된 바이트코드에 있어서 사이즈 경감과 성능(실행속도)의 Trade off에 대한 정량적 분석을 통하여 바이트코드 최적화기의 성능 향상을 이루는데 있다.

참고문헌

[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 Programming 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

http://www.openjit.org.

[11] The Java HotSpotTM Virtual Machine

(Technical White Paper)

http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html

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

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

이 야리

1986년~1990년 고려대학교 전자전산공학과(학사)

2000년~2001년 경인여자대학 멀티미디어 정보전산학부 전임강사

1999년~현재 동국대학교 컴퓨터공학 박사 과정

관심 분야는 컴파일러, 프로그래밍 언어, XML

오 세만

1993년~1999년 동국대학교 컴퓨터공학과 대학원 학과장

1985년~현재 동국대학교 컴퓨터공학과 교수

관심 분야는 프로그래밍 언어, 컴파일러, XML, 모바일 언어