실시간 클래스 생성에 의한 해석적 자바 코드의 성능 향상 기법

(A Performance Tuning Technique for Interpretable Java Code by Realtime Class Generation)

김 윤 호*
(Yun-Ho Kim)

요 약 본 논문에서는 실시간에 많은 회수의 해석처리를 요하는 문제에 대한 자바 코드 성능 향상을 위한 실시간 클래스 생성 방식을 제안한다. 먼저 해석 처리한 결과를 실시간에 동적으로 자바 소스로 생성하고 컴파일한 뒤 메모리에 적재시키며, 이후에는 생성된 클래스의 메소드 호출에 의해 계산이 처리되도록 한다. 이렇게 함으로써 매번 해석에 따른 시간 부담을 감소시켜 반복 계산의 성능의 향상시켰다.

Abstract This paper presents a realtime class generation technique for Java performance tuning of enormously repeated codes needing interpreting. Based on interpreted results, a Java source program for a Java class including a method with the corresponding function is generated, compiled, and loaded on the memory. And then and next time, the newly generated method is called for processing the original codes. As the results, overall times of repetition processing is reduced.

1. 서론

자바 프로그래밍 언어 (Java Programming Language) [1]가 1995년 발표된 이래 자바는 프로그래머나 기업체에 급속히 전파되어 왔다. 한편, 자바는 초창기부터 C나 C++와 같은 언어로 작성된 코드에 비하여 속도가 느리다는 비평을 들어왔다. 그러나, 이러한 속도의 문제는 자바언어의 특성에 대한 이해의 부족에 기초한 코딩에 의해 초래된 경우도 많았으며 또한 자바 가상 머신의 급속한 성능 발전에 따라 현재는 이에 대한 비평은 수그러들고 있다. 하지만, 특정 언어에 기반하지 않고도 소프트웨어에서의 성능 향상 (performance tuning)의 문제는 소프트웨어 공학적인 관점에서 항상 제기되어 온 문제이다. 자바에서는 코드의 최적화에 활용할 수 있는 강력하면서도 독특한 기능들을 가지고 있다 (예: Class, ClassLoader, Runtime, Reflection 등). 따라서, 이러한 기능들에 기반하여 자바 코드의 성능 개선을 위한 여러 기법들이 제안되고 있다[2,3].

본 논문에서는 무수히 반복 실행될 것이 요구되는 문제에서 해석과 계산의 분리가 가능한 경우에 자바 코드의 성능을 향상시킬 수 있는 방법을 제안하고자 한다. 즉, 어떠한 문제의 해결을 위하여 이에 대한 해석을 하고난 뒤 계산을 수행하게 되는 성격을 가지는 자바 코드들에 있어서, 이들 코드들이 반복적으로 수행이 될 때에는 매번 해석을 하는 과정과 계산을 하는 과정을 반복하게 된다. 이러한 경우에 있어서 해석을 하는 과정과 계산을 하는 과정을 분리하여, 해석하는 과정에서 해석된 결과를 보존하고 이후의 수행에서는 그 결과에 의거하여 계산을 하게되면 매번 해석하는 시간을 절감함으로써 성능을 향상시킬 수 있다. 즉, 최초에 해석한 결과를 보존하는 부분과 이에 의거하여 결과를 계산하는 부분을 가지는 클래스를 설계하고 이 클래스를 실행시간에 생성시킴으로써, 이후에는 이 클래스에서 계산하는 부분의 반복적 실행을 통해서 결과를 얻도록 하는 것이다.

2. 해석적 자바코드의 처리

* 안동대학교 전자정보산업학부

본 논문에서는 실시간 클래스 생성에 의한 자바 코드의 성능 향상을 위한 기법을 제시하고자 한다. Richards[4]에서의 예에서와 같이 후위 연산식 (postfix expression)의 처리 문제와 같은 경우에 있어서 자바 프로그래밍 언어는 중위 연산식(infix expression)만을 구문적으로 지원하므로 후위 연산식을 바로 자바의 코드로 표현하여 처리할 수 없다. 따라서, 후위 연산식을 처리하기 위해서는 중위 연산식으로 변환하여 코딩을 하거나 아니면 후위 연산식을 해석해서 바로 처리하는 방법 즉, 연산자를 만날 때마다 연산항 두개를 가져와서 연산을 하는 과정을 반복하여 처리를 하도록 코딩을 하는 식으로 해결한다. 이와 같이 문제의 해결을 위하여 바로 자바 코트로 표현할 수 없고 문제의 내용을 분석하여 처리하는 즉, 해석하는 코드를 '해석적 코드(interpretable code)'라 부르기로 한다.

이러한 해석적 코드는 실행시에 반복적 수행이 될 경우에는 프로그램 성능에 부정적인 영향을 미치게 된다. 해석적 코드의 한 예로서 매개변수를 가지는 후위 연산식을 입력으로 받는 문제에 대하여 고려해 보기로 한다. 후위 연산식을 처리하는 방법으로서는 스택을 이용하여 후위 연산식을 계산하는 방법 또는 후위 연산식에 대한 파스트리를 생성하고 이것에 의거하여 연산하는 방식이 일반적이다. 그러나, 이러한 알고리즘은 매번 계산할 때마다 후위 연산식에 대한 전체 처리를 반복하게 된다. 특히, 하나의 연산식에 대하여 매개변수의 값만이 변경되어 반복될 경우에는 하나의 후위 연산식에 대한 계산구조는 하나로 일정하므로, 매번 계산구조를 분석하는 과정은 처리 중복의 의미가 있다.

따라서, 이러한 문제의 한 해결방법으로서 처음 후위 연산식에 대한 파스트리 생성시에 각 중간단계마다 (두 연산항과 하나의 연산자를 가질 때마다) 그 결과를 중위연산식으로 표현을 하여 전체 연산식을 획득하고 그 식에 대하여 연산하여 계산값을 얻으며, 이후에는 바로 이 연산식에 대하여 연산을 함으로써 중복되는 처리를 회피할 수 있는 방법을 생각할 수 있다. 즉, 처음에는 후위 연산식의 해석 결과를 중위 연산식 형태로 나타내고, 이 식에 대하여 연산하며, 이후에는 이 문제를 중위 연산식의 문제로 보고 바로 연산만을 하는 것이다. 이렇게 하면, 스택을 이용하여 매번 후위 연산식에 대한 연산을 하거나, 매번 파스트리를 생성하고 연산을 하는 방법에 비해 처리시간을 줄일 수 있다. 처음 한 번 중위 연산식으로 변환을 하는 시간 외에는 바로 중위 연산식에 의한 연산을 수행하기만 하면 되기 때문이다.

여기서 문제가 되는 것은 실행시간에 코드가 변경되는 것인데, 이것은 프로그램에서 기본적으로는 불가능한 일이다. 그러나, 이러한 상황은 '코드 생성 기법'[3,5]을 이용하여 해결하는 것이 가능하다. 따라서, 본 논문에서는 '동적 코드 생성 기법'[4]을 보완 확장하여, 해석된 결과를 반영하고 처리할 수 있도록 클래스를 실행시간에 생성하고 이 클래스가 매번 실행시간에 호출되도록 함으로써 자바 코드의 성능을 개선할 수 있는 방법을 제시하고자 한다.

2.1 해석적 코드

이 절에서는 해석적 코드의 한 예로서 다음의 예제에서 보이는 바와 같은 매개변수를 가지는 후위 연산식과 일반적인 (해석을 수반하는) 후위 연산식 계산 알고리즘에 대하여 기술한다.

[예제 1] 여기서 후위 연산식은 상수 또는 매개변수를 연산항으로 하는 후위 연산식에 대해서 고려한다. 매개변수의 표기는 $\#i$ $(i = 1, 2, 3, \ldots)$의 형태를 가지는데, 여기서 #은 매개변수임을 나타내고, i는 매개변수의 식별자를 나타낸다. 예를 들어 2개의 매개변수를 가진 입력 후위 연산식이 "9.0 #1 + #2 * #2 -"라면, 이것은 중위 연산식 표현으로 "(((9.0 + #1) * #2) - #2)"에 대응되며, 2개의 매개변수는 #1과 #2로 표현된다. 매개변수에 대한 입력은 배열의 형태로 매개변수의 순서에 대응되게 주어진다. 즉, 두 매개변수 #1과 #2의 값이 각각 '1.0', '2.0'라면, double[] paramValue = {1.0, 2.0}과 같이 주어지는 것으로 한다.

예제 1에 대한 스택을 이용한 처리 알고리즘의 예를 예제 2에서 보인다. 후위 연산식의 스트링 형태의 입력에 대한 토큰분석은 후위 연산식의 처리를 위한 구현 방식에 독립적으로 필요한 부분이다. 따라서, 토큰의 분석부를 클래스의 생성자에 두고, 그 이후의 처리부를 compute() 메소드로 구현한다. 이렇게 함으로써, 각 알고리즘 간의 수행시간의 비교는 compute() 메소드의 수행시간의 비교에 의해 이루어 질 수 있도록 한다.

[예제 2] 예제 1과 같은 형태의 후위 연산식을 처리하기 위한 스택 기반 알고리즘의 주요 부분을 보이면 다음과 같다. main() 메소드에서는 이 클래스에 대한 테스트의 한 예를 보여준다.

```
public class PostfixStack {
 String[] tokens = null;

 public PostfixStack(String expression) {
  // expression의 어휘분석 결과를 tokens에 저장.
```

```
}
public double compute(double[] values) {
 Stack stack = new Stack();
 // 후위 연산식에 대한 해석 및 계산
 for (int i=0;i<tokens.length;i++) {
  String token = tokens[i];
  if (token.startsWith("#")) {
   int id = Integer.parseInt(token.substring(1));
   stack.push(new Double(values[--id]));
  } else {
   char opChar = token.charAt(0);
   int opCode = "+-*/".indexOf(opChar);
   if (opCode == -1) {
    //연산항인 경우 스택에 저장.
    stack.push(Double.valueOf(token));
   } else {
    //연산자인 경우 연산항 두개를 스택에서
    //가져와서 계산하고 그 값을 스택에 저장.
    double opr2 = ((Double) stack.pop()).doubleValue();
    double opr1 = ((Double) stack.pop()).doubleValue();
    switch(opCode) {
     case 0:
       stack.push(new Double(opr1+opr2)); break;
     case 1:
       stack.push(new Double(opr1-opr2)); break;
     case 2:
       stack.push(new Double(opr1*opr2)); break;
     case 3:
       stack.push(new Double(opr1/opr2)); break;
     default: // 유효하지 않은 연산자에 대한 예외처리.
    }//endSwitch
   }//endInnerIf
  }//endOuterif
 }//endFor
 // 최종값의 반환.
 return ((Double) stack.pop()).doubleValue();
}//compute()

public static void main(String[] args) {
 // 입력 후위 연산식과 매개변수값.
 String inputExpr = "9 #1 + #2 * #2 -";
 double[] paramValue = {1.0,2.0};

 // PostfixStack 클래스의 생성과 계산.
 PostfixStack ps = new PostfixStack(inputExpr);
 double result = ps.compute(paramValue);
 System.out.println("Result: " + result);
}
}
```

□

2.2 코드의 해석과 처리 클래스의 생성

이 절에서는 코드를 해석하고 해석된 결과에 의거하여 문제를 처리하는 클래스를 자동 코딩하고 컴파일한 뒤 클래스를 로딩하는 내용을 기술한다.

2.2.1 해석된 수식의 생성

후위 연산식을 해석한 결과를 중위 연산식 형태의 수식으로 산출하는 것은 다음과 같이 하여 이루어진다.

```
String InterpretExpr() {
// String[] tokens에서 어휘 분석된 결과를 가져옴
 Stack stack = new Stack();

 for(int i=0;i<tokens.length;i++) {
  String token = tokens[i];
  if (token.startsWith("#")) {
   int idx = Integer.parseInt(token.substring(1));
   idx--;
   stack.push("paramValues[" + idx + "]");
  } else {
   int opChar = "+-*/".indexOf(token.charAt(0));
   if (opChar == -1) {
    stack.push(token);
   } else {
    String opr2 = (String) stack.pop();
    String opr1 = (String) stack.pop();
    stack.push("(" + opr1 + " " + token.charAt(0) +
     " " + opr2 + ")");
   }//endInnerIf
  }//endOuterIf
 }//endFor

 return ("return " + (String) stack.pop() + ";");
}
```

[예제 3] 후위 연산식 "9.0 #1 + #2 * #2 -"에 대하여 위의 메소드를 수행한 결과로 "(((9.0 + #1) * #2) - #2)"를 얻게 된다. 여기서 후위 연산식을 해석하여 그 결과를 중위 연산식으로 표현하는 이유는 중위 연산식으로 표현되어야 그 결과를 그대로 자바의 수식으로 자바 소스 파일에 포함시킬 수 있기 때문이다. □

2.2.2 실시간 클래스 소스의 생성

후위 연산식을 해석한 결과를 처리하기 위한 자바 클래스 소스파일을 생성하기 위한 과정은 다음과 같다.

```
void buildSrc(File srcFile, String classname,
    String IntprExpr) throws IOException {
```

```
 FileOutputStream out =
  new FileOutputStream(srcFile);

 String classHead = "public class " +
    classname + " extends Computation {";
 String method = " public double " +
    "compute(double[] paramValues) {";
 String classEnd = " } }";

 String source = classHead + method +
    IntprExpr + classEnd;
 out.write(source.getBytes());
 out.flush();out.close();
}
```

여기서 compute() 메소드는 추상 클래스 (abstract class) Computation의 구상 클래스 (concrete class) 내에서 정의되도록 설계하였으며, 이렇게 함으로써 실행시간에 생성되는 클래스의 이름에 독립적으로 compute() 메소드를 호출할 수 있게 된다. Computation 추상 클래스는 다음과 같이 설계된다.

```
public abstract class Computation {
    abstract double compute(double[] arguments);
}
```

[예제 4] 예제 3에 대하여 위의 메소드를 적용한 결과 얻게되는 자바 소스파일의 예는 다음과 같다. (아래의 코드에서 "..."은 클래스 이름을 나타내며, 클래스 이름은 임의로 지정하여 주면 되므로 여기서는 "..."으로 표현하였다.)

```
public class ..... extends Computation {
 public double compute(double[] paramValues) {
  return (((9.0 + paramValues[0])*
   paramValues[1]) - paramValues[1]);
 }
}
```

□

이 예에서 보는 바와 같이 입력된 후위 연산식에 대하여 위와 같은 클래스를 얻게 되면, 이 후에는 다른 인자값들이 주어질 때 예제 2에서와 같이 다시 천체적인 처리를 할 필요가 없이 *compute*() 메소드를 인자값들을 주어 호출하면 된다 (예제 5의 main()메소드 부분 참고).

2.2.3 클래스 소스의 컴파일

새로이 생성된 자바 클래스 파일에 대한 컴파일은 다음과 같이 하여 이루어진다. 즉, 컴파일이 실시간에 이루어지므로 자바의 Runtime.getRuntime().exec() 메소드를 사용하여 javac 컴파일러를 새로이 생성할 파일의 이름을 매개변수로 전달받아 실행시키게 된다.

```
void compileSrc(File srcFile) throws IOException {
 String _compiler = "javac";
 String _classpath = ".";

 String[] cmd = { _compiler, "-classpath", _classpath,
  srcFile.getName());

 Process process = Runtime.getRuntime().exec(cmd);
 try {
  process.waitFor();
 } catch (InterruptedException e) {}
}
```

2.2.4 컴파일된 클래스의 로딩

이전 과정에서 생성된 자바 소스에 대해 컴파일된 클래스를 읽어들이는 과정은 다음과 같이 하여 이루어진다. 이것은 클래스 파일의 내용을 바이트 스트림으로 읽어들여, 자바의 java.lang.ClassLoader 클래스의 defineClass() 메소드와 java.lang.Class 클래스의newInstance() 메소드를 이용하여 메모리에 적재시켜 새로이 생성된 클래스를 실시간에 사용할 수 있게 한다.

```
Computation loadGenClass(String classname)
    throws IOException {
 File classfile = new File(classname + ".class");
 byte[] buf = new byte[(int) classfile.length()];

 FileInputStream in = new FileInputStream(classfile);
 in.read(buf);
 in.close();

 Class c = defineClass(classname, buf, 0, buf.length);

 try {
  return (Computation) c.newInstance();
 } catch (IllegalAccessException e) {
  throw new RuntimeException(e.getMessage());
 } catch (InstantiationException e) {
  throw new RuntimeException(e.getMessage());
 }
}
```

2.2.5 처리 과정의 총괄

앞서의 여러 절에서 주어진 후위 연산식에 대해 해석

된 결과를 생성하고 이 결과를 수행하기 위한 클래스의 소스파일을 실시간에 생성하고 컴파일하며 생성된 클래스 파일을 메모리에 적재하는 각 과정을 기술하였다. 이 절에서는 이들을 총괄하여 전체적인 처리가 이루어질 수 있도록 하는 메소드를 보인다.

```
public Computation generate() {
 String expr = "";
 String filename = "";
 String classname = "";

 try{
  File javafile = File.createTempFile(
   "generated_", ".java", new File("."));
  filename = javafile.getName();
  classname = filename.substring(0,
   filename.lastIndexOf(".");

  expr = InterpretExpr();
  buildSrc(javafile, classname, expr);
  compileSrc(javafile);
  return loadGenClass(classname);

 } catch (IOException e) {}
}
```

[예제 5] 이상에서 보인 모든 메소드를 통합하여 하나의 클래스로 설계한 결과는 다음과 같으며, 이에 대한 클래스 다이어그램은 그림 1과 같다. main() 메소드에서는 이 클래스에 대한 테스트의 한 예를 보여준다.

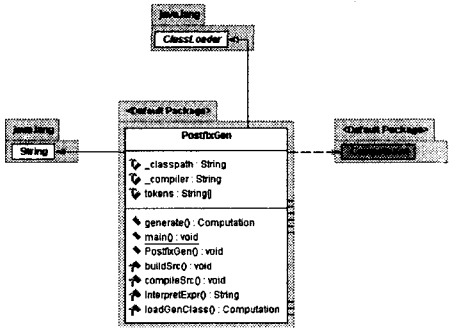

그림 1 PostfixGen 클래스 다이어그램

```
public class PostfixGen extends ClassLoader {
 String[] tokens;

 public PostfixGen(String expression) {
  super(ClassLoader.getSystemClassLoader());
  ArrayList list = new ArrayList();
  StringTokenizer tokenizer =
   new StringTokenizer(expression);
  while (tokenizer.hasMoreTokens()) {
   list.add(tokenizer.nextToken());
  }
  tokens = (String[]) list.toArray(
   new String[list.size()]);
 }

 public Computation generate() { ..... }

 Computation loadGenClass(String classname) { ..... }

 void buildSrc(File srcFile, String classname,
  String IntprExpr) throws IOException { ..... }

 void compileSrc(File srcFile) throws IOException { ..... }
  String InterpretExpr() {.....}

 public static void main(String[] args) {
  String inputExpr = "9 #1 + #2 * #2 -";
  double[] paramValue = {1.0,2.0};

  POstfixGen pg = new PostfixGen(inputExpr);
  Computation g = pg.generate();
  double result = g.compute(paramValue);
  System.out.println("Result: " + result);
 }
}
```

□

3. 수행시간의 측정

3.1 Stopwatch 클래스

이 절에서는 예제 2와 예제 5의 성능평가를 위하여 수행시간의 측정을 위한 Stopwatch 클래스에 대하여 기술한다. 시간의 측정은 자바의 java.lang.System의 currentTimeMillis() 메소드의 반환값에 기초하여 설계하며[2], 이 클래스에 대한 클래스 다이어그램은 그림 2와 같다.

그림 2에서 보는 바와 같이 Stopwatch 클래스는 getElapsedTime(), reset(), start(), stop()의 4개의 메소드로 구성되며 각 메소드에 대한 기능은 다음과 같다.

■ start(): 측정을 시작. currentTimeMillis()의 반환값을 startTime 변수에 저장.

■ stop(): 측정을 종료. currentTimeMillis()의 반환값

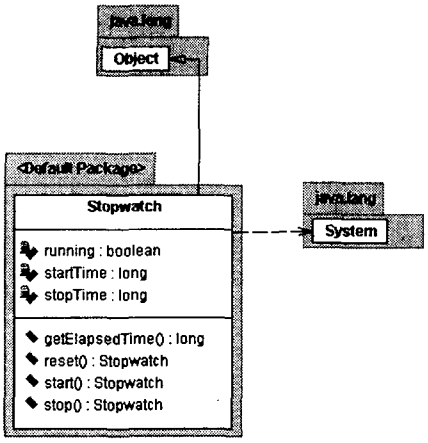

그림 2 Stopwatch 클래스 다이어그램

을 stopTime 변수에 저장.

■ getElapsedTime(): 경과시간을 측정. stopTime에서 startTime을 뺀 값을 반환.

■ reset(): 측정값을 초기화. stopTime와 startTime의 값을 초기화.

본 논문에서 측정 구간에 대한 측정값은 자바 가상 머신 (Java Virtual Machine)의 작동 환경에 따른 편차를 고려하여 측정 구간에 대한 1,000,000회 반복 측정하여 얻어진 값 또는 그 평균을 최종 측정값으로 한다 [2,6].

3.2 성능의 비교

예제 2와 예제 5에서 입력되는 후위 연산식에 대한 토큰 분석은 공히 두 경우 모두에 전제되는 과정이므로 각 클래스의 생성자에서 토큰 분석이 이루어지도록 하였다. 그러나, 각 클래스의 생성 시간은 클래스에 따라 다르므로 클래스의 생성 시점부터 측정을 시작하여야 한다. 또한, 예제 2에서 클래스 생성 시에 토큰 분석이 이루어지므로 같은 입력식에 대하여는 매번 계산시마다 클래스를 생성하지 않아도 된다. 따라서, 예제 2에서의 성능 측정을 위하여 클래스 생성을 하는데 소요되는 시간 t_a (time_a)와 이후 계산을 위해 소요되는 시간 t_b (time_b)로 나누어 측정한다. 다음과 같이 측정 구간을 설정한다. 여기서 얻어진 총 수행시간의 값을 $T^S(=t_a+t_b)$라 부르기로 한다.

```
.....
double result = 0.0;
long time_a = 0;
long time_b = 0;

Stopwatch sw = new Stopwatch();
sw.start();
PostfixStack ps = new PostfixStack(inputExpr);
sw.stop();
time_a = sw.getElapsedTime();
sw.reset();

for(int i=0;i<1000000;i++) {
sw.start();
result = ps.compute(paramValue);
sw.stop();
time_b = time_b + sw.getElapsedTime();
}
.....
```

한편, 예제 5에서의 성능 측정은 두 가지 부분에 대하여 이루어지게 된다. 즉, 입력되는 후위 연산식에 대하여 계산 구조를 해석하여 처리 클래스를 생성, 컴파일, 로딩하는 과정에 대한 측정을 하는 부분과 이후에 계산을 수행하는 과정에 대하여 측정을 하는 부분으로 나뉘어 측정이 이루어진다. 결과적으로 두 측정값의 합이 전체 처리를 위해 경과되는 시간이 된다. 이 시간을 T^G라 한다. 앞 부분에 대하여 성능 측정을 위한 구간의 설정은 다음과 같이 된다.

```
.....
double result = 0.0;
long time_g = 0;
Stopwatch sw = new Stopwatch();

for(int i=0;i<1000000;i++) {
 sw.start();
 PostfixGen pg = new PostfixGen(inputExpr);
 Computation g = pg.generate();
 sw.stop();
 time_g = time_g + sw.getElapsedTime();
}

result = g.compute(paramValue);
.....
```

위에서 얻어진 time_g의 값을 t_g라 부르기로 한다. 한편, 두번째 경우의 성능 측정을 위한 구간의 설정은 다음과 같이 된다.

```
.....
double result = 0.0;
long time_c = 0;
Stopwatch sw = new Stopwatch();

PostfixGen pg = new PostfixGen(inputExpr);
Computation g = pg.generate();

for(int i=0;i<1000000;i++) {
sw.start();
result = g.compute(paramValue);
sw.stop();
time_c = time_c + sw.getElapsedTime();
}
.....
```

위에서 얻어진 time_c의 값을 t_c라 부르기로 한다. 따라서, 예제 5의 방법에 의한 처리 시간 T^G는 $T^G = t_g + t_c$가 된다.

[예제 6] 예제 2와 예제 5에 대하여 t_a, t_b, t_g, t_c를 측정한 결과는 다음과 같다. 실험 입력 후위연산식 (1)은 "9 #1 +#2 * #2 -", 식 (2)는 "#1 #2 + #1 + #2 + #1 + #2 +#1 +#2 + #1 + #2 + #1 + #2 + #1 + #2 + #1 + #2 + #1 + #2 +"이며, 반복실행 회수는 10^6번으로 한다. (실험환경: OS WindowsXP, CPU 650MHz, Java J2SDK 1.4.0_01)

입력식	T_n^S		T_n^G	
	t_a	$\Sigma_{i=1}^n t_b$	t_g	$\Sigma_{i=1}^n t_c$
식 (1)	0.0013	7562	1342	281
식 (2)	10	30294	1382	471

한편, 어떠한 문제에서 예제 2 또는 예제 5의 알고리즘이 n번 반복될 것이 요구되는 경우에 대하여 고려하자. 이때 예제 2의 방법에 의한 총 수행 시간 T_n^S은 $T_n^S = t_a + \Sigma_{i=1}^n t_b$가 되며, 예제 5의 방법에 의한 총 수행 시간 T_n^G은 $T_n^G = t_g + \Sigma_{i=1}^n t_c$가 된다. 따라서, $(t_a + \Sigma_{i=1}^n t_b) > (t_g + \Sigma_{i=1}^n t_c)$의 조건을 만족하는 n에 대하여 예제 5의 방법이 더 좋은 성능을 보이게 됨을 알 수 있으며, 또한 웹 서버 환경과 같은 상황에서 선적재 (preload)가 요구되는 경우[7]에 있어서는 선적재 시에 해석과정에 대한 처리가 이루어지고 그 이후 클라이언트에 대한 서비스가 실시되므로 예제 5번의 방법이 더 유리하다는 결론을 얻을 수 있다.

4. 결 론

본 논문에서는 실시간에 수많은 반복 처리를 요하는 해석적인 문제가 입력되었을 때 이의 처리에 대한 성능 개선(Tuning)을 위한 한 방법으로서 실시간에 해석된 결과를 처리하기 위한 클래스를 생성하여 처리하는 기법을 제시하였다. 매번 수행될 때마다 해석이 반복 수행되는 것을 개선하기 위하여 해석된 결과를 보존하고 이를 처리하기 위한 메소드를 포함한 자바 클래스 소스파일을 실시간에 생성하며, 컴파일하고, 메모리에 적재함으로써 이 후부터는 새로이 생성된 클래스의 메소드를 호출하기만 하면 되도록하였다. 이렇게 함으로써, 반복 수행될 때마다 매번 해석하는 과정을 생략할 수 있어 처리를 위한 반복 계산 시간을 줄일 수 있었다. 본 논문에서 제시하는 방법을 더욱 향상시키기 위하여 소스파일을 생성하고, 컴파일, 메모리에 적재하는 시간의 비중이 메소드를 호출하여 계산하는 시간에 비해 크므로, 실시간 클래스 생성 시간을 보다 적게 하는 연구가 계속 수행되어야 한다.

참 고 문 헌

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

[2] S. Wilson and J. Kesselman, Java Platform Performance: Strategies and Tactics, Addision-Wesley, 2000.

[3] A. Hunt, The Pragmatic Programmer, Addison-Wesley, 1999.

[4] N. Richards, "Dynamic code generation," Java Developer's Journal, vol. 7, no. 2, pp. 54-62, 2002.

[5] M. J. Rettig and M. Fowler, "Reflection vs. code generator," JavaWorld Magazine, vol. 6, no. 11, 2001.

[6] J. Shirazi, Java Performance Tuning, O'Reilly, 2000.

[7] D. Bulka, Java Performance and Scalability: Server-side Programming Techniques, Addison-Wesley, 2000.

김 윤 호 (Yun-Ho Kim)

1983. 2. 경북대학교 전자공학과 공학사
1993. 2. 경북대학교 대학원 컴퓨터공학과 공학석사
1997. 2. 경북대학교 대학원 컴퓨터 공학과 공학박사
1997. 8~현재 안동대학교 전자정보산업학부 조교수
관심분야 : 분산병렬처리, 객체지향 분석/설계/프로그래밍, 인터넷 컴퓨팅