자바 성능 분석기의 설계 및 구현

이 종 동, 정 민 수, 이 재 동, 진 민, 박 규 석

경남대학교 컴퓨터공학과

Design and Implementation of the Java's Performance Analyzer

Jong-Dong Lee, Min-Soo Jung, Jae-Dong Lee, Min Jin, Kyoo-Seok Park

Dept. of Computer Engineering, Kyungnam University

요 약

본 논문은 자바의 성능을 분석하기 위한 것이다. 실행 중인 자바의 프로그램의 메쏘드에서 메쏘드가 속한 클래스, 바이트코드 정보와 메쏘드의 호출관계, 호출 횟수, 실행 시간, 메모리 사용에 관한 정보를 분석하여 시각화 도구를 통하여 그 결과를 쉽게 알 수 있도록 구현하였다. 자바 성능 분석기의 분석을 통하여 프로그램 개발자는 프로그램의 성능에 문제를 일으키는 부분을 개선시켜 보다 나은 성능의 프로그램 개발할 수 있다.

1. 서론

자바(Java)는 플랫폼 독립적인 특성으로 인하여 인터넷과 같은 네트웍 환경에서 동작하는 프로그램 작성에 널리 사용되고 있다. 뿐만 아니라 C++ 언어보다 더 객체 지향적이기 때문에 더욱 주목을 받고 있으며, 현재 많은 응용 프로그램들이 자바로 작성되어지고 있다. 자바의 실행 형태는 두 가지로 나뉠 수 있는데, 자바 컴파일러에 의해 생성된 자바 바이트코드(bytecode)들이 로컬로 전송되어 실행되거나, 바이트코드들이 직접 로컬 환경에서 실행된다. 보통 전자의 형태를 애플릿(applet) 프로그램이라 하고, 후자의 형태를 응용(application) 프로그램이라 한다. 자바 바이트코드는 자바 가상 기계(Java Virtual Machine:JVM)가 구현된 어떠한 플랫폼에서도 소스코드의 수정 없이 잘 동작되도록 설계되었다[1, 2].

일반적으로 자바 가상 기계를 구현하는 방식은 하드웨어 상에서 직접 바이트코드를 실행하기 위해 설계된 자바 칩(Java Chip)을 이용하는 방법과 자바 칩을 이용하지 않는 범용 마이크로프로세서에서 소프트웨어로 구현하는 방법으로 나눌 수 있다. 소프트웨어로 구현하는 방법은 다시 몇 가지로 구분할 수 있다. 먼저 바이트코드를 차례로 읽어 해석해 가면서 바로 수행하는 해석(interpreting)방식이 있고, 다음은 각각의 바이트코드를 해당 마이크로프로세서의 고유코드(native code)로 대응시켜 생성된 코드를 실행하는 컴파일 방식이 있다.

본 연구에서는 자바의 성능을 분석하여 프로그래머로 하여금 자바가 좀 더 빨리 실행될 수 있도록 하기 위하여 '자바 성능 분석기'라는 특별한 시스템을 설계 및 구현하였으며, 또한 자바 성능 분석기가 분석한 결과를 사용자들이 알기 쉽게 보여주는 시각화(visualization) 도구도 구현하였다.

2. 자바 성능 분석기

2.1 자바 성능 분석기의 역할

자바 성능 분석기의 역할은 자바 프로그램의 실행 시간에 자바 메쏘드들을 분석하는 것이다. 자바 성능 분석기는 자바 가상 기계의 내부에 있는 해석기(interpreter)와 연계되어 모든 정보를 해석기로부터 얻게 된다. 그 정보는 메쏘드의 호출 횟수, 실행 시간, 메모리 사용량(스택, 지역 변수 크기), 바이트코드, 호출자 등이다.

2.2 Kaffe 의 주요 모듈 분석

자바 성능 분석기는 자바의 실행 시간에 정보를 수집하고 분석한다. 따라서 자바 성능 분석기를 만들기 위해서는 자바 가상 기계가 필요하게 된다. 본 논문에서는 자바 가상 기계를 직접 구현하지는 않고 소스가 공개된 Kaffe 자바 가상 기계를 사용하여 설계 하였다.

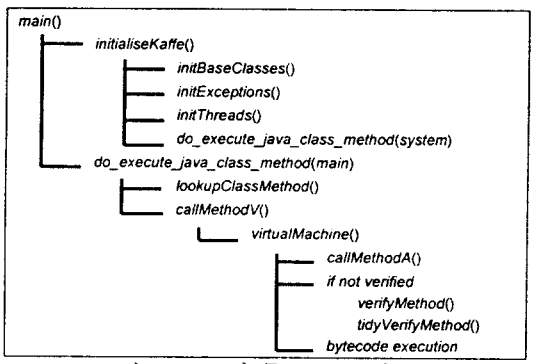

그림 1. Kaffe 의 주요 모듈 리스트

그림 1 에 나타난 Kaffe 의 주요 모듈들을 설명하면 다음과 같다.

- *initialiseKaffe*() - 시스템 클래스, 예외처리, 쓰레드에 관련된 초기화 작업을 수행한다.
- *do_execute_java_method* () - 고유코드로부터 자바 메쏘드를 호출한다.
- *do_execute_java_class_method* () - 고유코드로부터 클래스의 정적 자바 메쏘드를 호출한다.
- *callMethodA*() - 인자로 주어진 고유코드나 자바 메쏘드를 호출한다(시그너처가 배열인 경우).
- *callMethodV*() - 인자로 주어진 고유코드나 자바 메쏘드를 호출한다(시그너처가 가변인자인 경우).
- *virtualMachine*() - 실제 바이트코드를 해석하고 실행한다.

2.3 자바 성능 분석기의 구성

자바 성능 분석기는 크게 호출 관계 분석부, 호출 횟수 분석부, 실행 시간 분석부, 메모리 사용량 분석부, 바이트코드 분석부로 이루어진다. 각 분석부는 서로 각각 실행되지 않고 동시에 동작하도록 되어 있다.

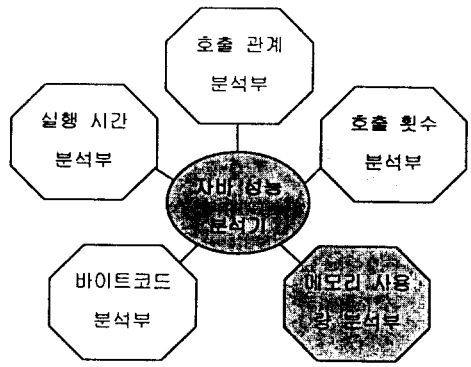

그림 2. 자바 성능 분석기 시스템의 구성

- **호출 관계 분석부**
  실행 시 메쏘드 호출 관계를 분석하여 메쏘드가 실행 될 때마다 해당 메쏘드가 고유코드를 사용하는 메쏘드 인지 아닌지도 분석한다. 분석 결과는 *jvm~0.prof* 라는 파일에 저장한다.
- **호출 횟수 분석부**
  프로그램 실행 시 각각의 메쏘드가 몇 번 호출되는지 분석한다. 분석 결과는 *jvm_method.lst* 파일에 저장한다.
- **실행 시간 분석부**
  프로그램 실행 시 각각의 메쏘드가 실행된 시간을 밀리 초단위로 계산한다. 분석 결과는 *jvm_time.lst* 파일에 저장한다.
- **메모리 사용량 분석부**
  프로그램 실행 시 각각의 메쏘드가 사용할 메모리 공간의 용량을 계산한다. 분석 결과는 *jvm_mem.lst* 파일에 저장한다.
- **바이트코드 분석부**
  바이트코드를 분석 결과 파일인 *jvm_bcode.lst* 에 저장한다.

2.4 메쏘드 호출 관계 그래프의 설계

대부분의 자바 가상 기계는 모든 작업의 처리를 메쏘드 단위로 하므로 자바 가상 기계에서 메쏘드 호출 관계를 파악하는 것은 쉽다.

Kaffe 소스에서 메쏘드의 실행과 관련된 *callMethodA*(), *callMethodV*() 함수의 내부에서 *virtualMachine*() 함수를 호출하면 인자로 넘어온 메쏘드 정보를 파악하면 메쏘드에 관련된 모든 정보를 알 수 있다. 메쏘드 호출 관계를 정확하게 나타내기 위하여 메쏘드가 호출되면 호출 깊이를 저장하는 변수의 값을 하나 증가 시키고, 메쏘드가 리턴되면 호출 깊이를 저장하는 변수의 값을 하나 감소시킨다.

2.5 메쏘드 분석부의 설계

메쏘드 분석부는 프로그램의 실행 중에 메쏘드에 관련된 정보를 수집, 분석하는 역할을 하는 것으로 호출 관계 그래프, 호출 횟수, 실행 시간, 메모리 사용량, 바이트코드 등에 관한 정보를 분석한다.

메쏘드 분석을 위해 먼저 Kaffe 에서 사용되는 자바 메쏘드 관련 자료 구조와 본 논문에서 구현을 위해 추가로 사용하는 자료 구조를 각각 그림 3, 4 에 나타내었다.

```
struct _methods {
    Utf8Const              name;          //메쏘드 이름
    Unf8Const              signature;     //시그너쳐
    accessFlags            accflags;      //접근 플래그
    short                  idx;           //디스패쳐 테이블의 클래스
                                           인덱스
    u2                     stacksz;       //스택 크기
    u2                     localsz;       //지역 변수 크기
    nativecode             ncode;         //고유 코드
    union {
        struct {
            nativecode*    ncode_start;   //고유코드 시작 포인터
            nativecode*    ncode_end;     //고유코드 끝 포인터
        } ncode;
        struct {
            unsigned char* code;          //바이트코드
            int            codelen;       //바이트코드 길이
        } bcode;
    } c;
    Hjava_lang_Class*      class;         //클래스
    struct _lineNumbers*   lines;         //라인번호 속성
    struct _jexception*    exceptiontable; //예외처리 테이블
}
```

그림 3. Kaffe 의 자바 메쏘드 정보 저장 구조

```
struct _methodPool {
    int                    count;         //호출 횟수
    long                   time;          //실행 시간
    int                    local;         //지역 변수 크기
    int                    stack;         //스택 크기
    char                   name[256];     //메쏘드 이름
    unsigned char*         bcode;         //바이트코드
    int                    bcode_len;     //바이트코드 크기
    MethodPool*            next;          //다음 포인터
}
```

그림 4. 자바 성능 분석기의 자바 메쏘드 정보 저장 구조

메쏘드 관련 정보를 효율적으로 관리하기 위하여 Kaffe 소스에 *profilerAnalysis*() 함수를 새로이 작성하여 중복되는 정보의 제거하고 세련된 정보를 그림 3 의 저장 구조에 저장 할 수 있도록 하였다. 모든 메쏘드 관련 정보들은 프로그램의 실행 중에는 연결 리스트(linked list) 구조로 저장되어 있다가 프로그램의 종료 시 파일로 저장된다.

3. 자바 성능 분석기의 구현

3.1 구현 환경

본 논문에서 구현한 자바 성능 분석기의 구현 환경은 다음과 같다. 유닉스 운영 체제인 솔라리즈(Solaris) 2.3 하에서 GNU-C 2.7.2.3 을 이용하여 C 언어로 구현하였다.

자바 가상 기계의 소스는 공개된 Kaffe 0.10.0 버전이며, 여기에 자바 성능 분석기의 기능을 수행할 수 있도록 본 논문의 구현에서 소스를 일부 수정하였다.

3.2 구현과 관련된 자원

자바 성능 분석기의 구현에 관련하여 사용되는 자원(resource)에 대한 내역이다.

자바 성능 분석기는 모든 정보를 분석하여 미리 정해진 파일로 분석 정보들을 저장한다. 각 파일의 구조를 표현하면 그림 5 와 같다. 각 파일의 항목 사이에는 항목 구분자로써 문자 "-?-" 을 사용한다.

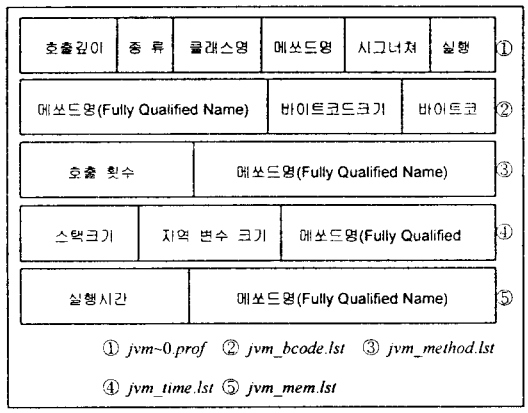

그림 5. 분석 결과 파일 구조

3.3 메쏘드 호출 그래프의 구현

메쏘드 호출그래프의 구현과 관련한 함수 관계를 간단하게 나타내었다.

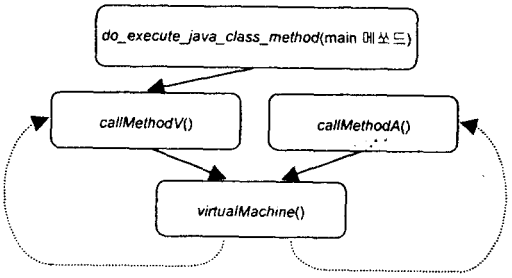

그림 6. 메쏘드 호출 그래프의 전체 흐름도

메쏘드 호출 그래프는 실제 자바 프로그램의 *main*() 메쏘드부터 출발하도록 구현하였다. 따라서 자바 가상 기계의 처음 초기화 부분은 생략되어 있다. 그림 6에서 알 수 있듯이 *main*() 메쏘드를 인자로 *do_execute_java_class_method*() 함수로부터 출발하여 메쏘드의 호출 그래프를 출력한다. 메쏘드가 호출 될 때 마다 *virtualMachine*()이 호출되므로 *virtualMachine*() 함수 내에서 인자로 넘어온 메쏘드 정보를 파악하여 메쏘드의 모든 정보를 파악한다. 호출의 관계를 나타내기 위하여 *callDepth* 변수를 사용한다. 메쏘드가 호출되면 *callDepth* 변수의 값을 하나 증가시키고, 리턴되면 *callDepth* 변수 값을 하나 감소시킨다.

함수 *virtualMachine*() 에서 실제 바이트코드를 실행하므로 여기서 다른 메쏘드를 호출하는 바이트코드(*invokevirtual*, *invokestatic*, *invokespecial*, *invokeinterface*)가 있으면 간접적으로 *callMethodA*(), *callMethodV*() 함수를 호출하도록 되어있다.

분석된 결과는 직접 *jvm~0.prof*, *jvm~1.prof* 파일에 저장된다. *jvm~0.prof* 파일은 시각화 도구의 입력으로 사용되며, *jvm~1.prof* 파일은 호출 그래프를 들여쓰기(indentation)하여 직접 볼 수 있도록 하기 위해 사용된다.

3.4 메쏘드 분석부의 구현

메쏘드 분석부는 호출 횟수, 실행 시간, 메모리 사용량, 바이트코드에 대한 분석을 수행하는 부분으로, *profilerAnalysis*()함수가 이 역할을 수행하도록 구현하였다.

void profilerAnalysis(char* args, long targ, int larg, int sarg, register bytecode* bc, int cl)		
설명	메쏘드에 대한 분석 정보를 연결 리스트에 저장한다.	
인자	args :	메쏘드 이름(Fully Qualified Name)
	targ :	메쏘드의 실행 시간
	larg :	메쏘드의 지역 변수 크기
	sarg :	메쏘드의 스택 크기
	bc :	메쏘드의 바이트코드
	cl :	바이트코드 크기

그림 7. *profilerAnalysis*() 함수

다음은 메쏘드 분석 저장에 관한 알고리즘이다.

```
연결 리스트가 비어(empty) 있으면
  - 첫 리스트에 분석 정보를 기록
연결 리스트가 비어 있지 않으면
  - 연결 리스트에서 현재 주어진 메쏘드가 존재하는지 검색
  - 만일 존재하면
    - 해당 리스트의 기존의 정보를 변경하여 저장
  - 만일 존재하지 않으면
    - 새로운 리스트 정보를 만들고 연결 리스트의 맨 끝 노드에 연결
```

그림 8. 메쏘드 분석 저장 알고리즘

4. 실험 및 실험 결과

4.1 시각화 도구

본 논문에서 실험 결과를 좀 더 쉽게 알 수 있도록 하기 위하여 시각화 도구를 구현하였다. 그림 9는 자바 프로그램의 실행에서부터 시각화 도구까지의 과정을 요약한 것이다.

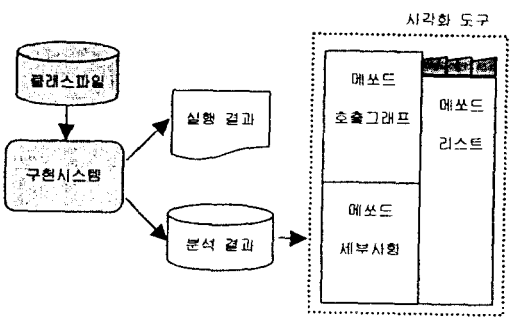

그림 9. 시각화를 위한 구성도

시각화 도구는 *main*()메쏘드에서 부터 호출되는 메쏘드의 순서를 트리 형식으로 나타내며, 메쏘드 호출 그래프의 특정 메쏘드를 선택하면 그 메쏘드를 분석한 결과를 세부 항목 부분에 표시하게 된다. 표시되는 세부 항목은 메쏘드가 속해있는 클래스, 메쏘드의 시그니처, 실행 시간, 자신을 호출한 메쏘드, 바이트코드 등이다. 또한, 메쏘드 리스트 부분에는 메쏘드를 호출 횟수, 실행 시간, 메모리 사용량별로 출력할 수 있다.

4.2 시각화 도구의 설계 및 구현

시각화 도구는 메쏘드 호출 그래프 출력기, 메쏘드 세부 항목 표기기, 메쏘드 리스트기의 세 부분으로 구성되어 있으며, 자바 언어로 작성하였다.

시각화 도구의 각 부분의 역할을 기술하면 다음과 같다.

- 메쏘드 호출 그래프 출력기 - 메쏘드의 호출 관계를 트리 형식으로 표시한다.
- 메쏘드 세부 항목 표시기 - 메쏘드가 속해있는 클래스, 메쏘드의 시그너처, 실행 시간, 자신을 호출한 메쏘드, 바이트코드 등을 표시한다.
- 메쏘드 리스트기 - 메쏘드를 호출 횟수, 실행 시간, 메모리 사용량별로 출력한다.

4.2.1 메쏘드 호출 그래프 출력기

본 논문에서 구현된 메쏘드 호출 그래프는 자바 프로그램이 실행된 순서를 의미한다. 분석 파일 *jvm~0.prof* 을 이용하여 트리 형식으로 호출 그래프를 표시한다.

메쏘드 호출 그래프 트리를 구성하는 방법은 다음과 같다.

```
① 모든 메쏘드의 연결 리스트를 만든다.
② 연결 리스트의 맨 처음 항목을 가져온다.
    - 가져온 항목을 가지고 루트 노드를 생성한다. (노드를 생성
    할 때 마다 호출 메쏘드의 정보를 나타내는 CallGraphInfo 와
    트리뷰(TreeView) 패널에서 하나의 노드를 나타내는
    TreeNode 인스턴스 객체를 각각의 스택에 넣는다)
③ 연결 리스트에서 다음 항목을 가져온다.
    - 연결 리스트에서 가져온 항목의 호출 깊이와 스택 꼭대기
    노드의 호출 깊이를 비교하여 가져온 항목의 값이 작거나
    같으면 계속 반복해서 스택에서 항목을 꺼낸다.
    - 연결 리스트에서 가져온 항목을 스택의 꼭대기에 있는 항
    목을 부모(parent)로 하여 새로운 노드를 만든다.
④ 연결 리스트의 마지막까지 ③을 반복한다.
```

그림 10. 트리 구성 알고리즘

4.2.3 메쏘드 세부 항목 표시기

메쏘드 세부 항목 표시기는 메쏘드 호출 그래프 표시기에서 특정 메쏘드를 마우스로 더블 클릭(double click)하면 메쏘드에 관련된 세부 항목을 출력한다.

출력되는 항목들은 *CallGraphInfo* 클래스의 멤버(member)로 되어 있다.

```
public class CallGraphInfo {
    int     callDepth;          // 호출 깊이
    int     methodKind;         // 코드 종류
                                // (1:고유코드, 0:바이트코드)
    String  parentName;         // 현 메쏘드의 호출자
    String  className;          // 클래스 명
    String  methodName;         // 현 메쏘드 명
    String  methodSig;          // 시그너처
    String  execTime;           // 메쏘드 실행 시간(ms)

}
```

그림 11. *CallGraphInfo* 클래스의 구성

4.2.4 메쏘드 리스트기

메쏘드 리스트기는 탭 컨트롤(tab control)을 사용하여 메쏘드에 관련된 정보를 나타낸다.

메쏘드 호출 횟수 리스트와 관련된 클래스는 *MethodInfo* 이고, 메쏘드의 실행 시간을 리스트와 관련된 클래스는 *TimeInfo* 이고, 메쏘드 메모리 사용량과 관련된 클래스는 *MemoryInfo* 이다.

4.3 실험 결과 분석

실제적으로 자바 프로그램을 선정하여 그 결과를 분석한다. 그림 12 의 *TestProfiler.java* 소스 프로그램을 컴파일하여 본 논문에서 구현한 자바 성능 분석기 시스템에서 실행시키면 분석 결과가 생성된다.

```
class TestProfiler {
static void test1(int n){
   for(int i=0;i < n;i++){
      double x = Math.log(i);
   }
}

static void test2(int n){
   for(int i=0;i < n;i++){
      double y = test3(i);
   }
}

static double test3(double x){
   return Math.sin(x);
}

static double test4(double x,int i){
   if(i > 0)
      return    Math.sin(test5(x,i-
      1));
   else
      return x;
}

static double test5(double x,int i) {
   if(i > 0)
      return Math.sin(test4(x,i-1));
   else
      return x;
}

public static void main(String[] args) {
   long t1 = System.currentTimeMillis();
   test1(100);
   long t2 = System.currentTimeMillis();
   test2(500);
   long t3 = System.currentTimeMillis();
   int [] mem = new int[100];
   long t4 = System.currentTimeMillis();
   double x = test4(1.,50);

   long t41 = System.currentTimeMillis();
   System.out.println("test1: "+(t2-t1)+
      " millis\n test2: "+(t3-t2)+" millis\n"+
      "test4: "+(t41-t4)+" millis");
   }
}
```

그림 12. 실행 예제 – *TestProfiler.java*

다음은 실제로 자바 소스 프로그램을 컴파일하고 실행하여 시각화 도구까지의 단계를 나타낸 것이다.

```
% javac TestProfiler.java    // 예제 파일을 컴파일
% kaffe TestProfiler         // 예제 파일을 실행
test1: 0 millis              //
test2: 0 millis              //
test4: 0 millis              // 예제 파일의 실행 결과
% java jprof                 // 시각화 도구를 실행
```

그림 13. 실험 과정의 예

Kaffe 자바 가상 기계를 통하여 자바 프로그램을 실행시키고 나면 아래의 그림들과 같은 형식의 파일

들이 생성되는데 이 파일들이 자바 성능 분석기가 생성한 자바 프로그램 분석 파일이다.

```
0-?-0-?-TestProfiler-?-main-?-([Ljava/lang/String;)V-?-1900.00
1-?-1-?-java/lang/System-?-currentTimeMillis-?-()J-?-1900.00
1-?-0-?-TestProfiler-?-test1-?-(I)V-?-1900.00
2-?-1-?-java/lang/Math-?-log-?-(D)D-?-1910.00
```

그림 14. 분석 파일의 예 – *jvm~0.prof*

4.3.1 시각화 도구를 통한 결과 브라우징

다음은 *TestProfiler.java* 프로그램의 실행을 분석한 결과이다.

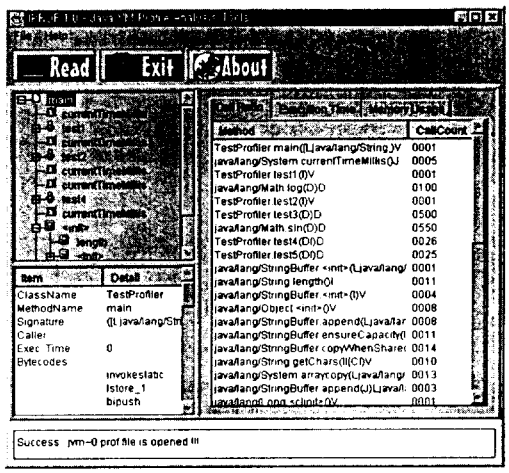

그림 15. 시각화 도구를 이용한 결과

5. 결론 및 향후 연구 방향

본 논문에서 구현한 '자바 성능 분석기'는 실행 시간에 자바 프로그램의 분석을 통하여 자바의 성능 문제를 파악하고 이에 대한 분석 자료를 제공한다. 또한 분석 자료를 좀 더 쉽게 파악할 수 있도록 시각화 도구도 제공한다.

앞으로는 본 논문에서 구현된 시스템을 바탕으로 보다 향상된 '고성능 자바 가상 기계'의 개발에 연구 방향을 맞추고자 한다. 또한, '자바 성능 분석기'의 처리 능력을 향상시키는 데에도 연구를 주력하고자 한다.

본 논문에서 구현된 '자바 성능 분석기'를 사용함으로써 보다 향상된 성능의 자바 프로그램을 작성하는데 도움을 제공할 것으로 기대한다.

참고 문헌

[1] G. McGraw and E. Felten, *Java™ Security*, Wiley, 1997

[2] P. van der Lindon, *Just Java 2/E*, SunSoft Press, 1997

[3] T. Lindholm and F. Yellin, *The Java™ Virtual Machine Specification* , Addison-Wesley, 1996

[4] J. Meyer and T. Downing, *Java™ Virtual Machine*, O'Reilly, 1997

[5] B. Venners, *Inside the Java Virtual Machine*, McGraw-Hill, 1998

[6] G. Cornell and C. S. Horstmann, *Core Java 2/E*, SunSoft Press, 1977

[7] J. Gosling, *The Java Language Specification*, Addison-Wesley, 1996

[8] D. Kramer, *The Java Platform A White Paper*, Java Soft, 1996

[9] A. Adl-Tabatabai and M. Cierniak and Huei-Yuan Lueh and V. M. Parikh and J. M. Stichnoth, "Fast, Effective Code Generation in a Just-In-Time Java Compiler", *Proceedings of the ACM SIGPLAN*, pp. 280~290, 1998

[10] A. Krall and R. Grafl, "CACAO-A 64 bit Java VM Just-in-Time Compiler", *PPoPP'97 Workshop on Java for Science and Engineering Computation*, 1997

[11] D. Flanagan, *Java in a Nutshell 2/E*, O' Reilly, 1997

[12] M. Campione and K. Walrath, *The Java™ Tutorial, Object-Oriented Programming for the Internet*, Addison-Wesley, 1996

[13] http://www.transvirtual.com/kaffe.html, Kaffe home page

[14] 이종동, 정민수, 이수진, 진민, "동적 컴파일링 기법을 이용한 자바 가상 기계의 설계" 한국정보과학회 '98 가을 학술발표논문집(I), 제 25 권 제 1 호, 1998. 10, pp. 425~427