韓國IT서비스學會誌
第6卷 第2號
2007年 8月

자바 프로그램의 런타임 특성 분석을 위한 Kaffe의 확장*

신범주** · 이창우*** · 이완직****

The Extension of Kaffe to Analyse Runtime Characteristics of a Java Program*

Bum Joo Shin** · Chang Woo Lee*** · Wan Jik Lee****

■ Abstract ■

This paper describes an extension of Kaffe JVM which enables to extract runtime characteristics of a Java program. The runtime characteristics include frequency of method call instruction, ratio of library method call and ratio of method whose runtime is less than compile time. It also represents ratio of method called only once, polymorphicity of virtual call and distribution of method size called in runtime. This paper analyses runtime features of the SciMark benchmark suite using the extended Kaffe.

Keyword : JVM, Benchmark, JIT Compiler, Kaffe

* 이 논문은 부산대학교 자유과제 학술연구비(2년)에 의하여 연구되었음.
** 부산대학교 바이오정보전자전공 교수
*** 군산대학교 컴퓨터정보과학과 교수
**** 부산대학교 바이오정보전자전공 교수, 교신전자

1. 서 론

Java 언어[5]는 간결하면서도 완전한 객체지향 프로그래밍을 지원할 뿐 아니라 뛰어난 이식성과 안정성을 지닌 응용 프로그램을 작성할 수 있는 장점을 가지기 때문에 다양한 분야에서 활용되고 있다. 특히 'write once, run anywhere'라는 슬로건처럼 한번 컴파일된 클래스 파일은 표준 사양을 준수하는 JVM[8]을 지원하는 모든 시스템에서 수행할 수 있기 때문에 인터넷을 기반으로 하는 응용에서는 더욱 진가를 발휘할 수 있다.

그러나 인터프리터를 사용하는 구조를 채택함으로써 Java는 C 및 C++에 비해 성능 측면에서 상대적인 열세에 있다. 이 같은 단점을 극복하기 위하여 Ahead of Time 컴파일러를 사용하거나, JIT 또는 HotSpot 컴파일러 등과 같은 바이트 코드를 머신 코드로 변환 후 수행하는 동적 컴파일러(Dynamic Compiler) 기술을 사용하고 있다[4]. 동적 컴파일러에서 사용하는 최적화 방법들은 응용 프로그램의 수행 특성에 종속적이기 때문에 응용에 따라 다른 성능을 보일 수 있다. 따라서 응용의 동적 특성을 분석하는 것은 최적의 성능을 지원할 수 있는 기반을 제공한다.

Kaffe는 가장 널리 사용되고 있는 오픈 소스 JVM 중 하나이며, 다양한 프로세서를 지원하며 GNU GPL 라이센스를 적용하기 때문에 소스 프로그램의 사용 및 변경이 자유롭다. 본 논문에서는 동적인 방법을 사용하여 Java 프로그램을 분석할 수 있도록 Kaffe를 확장하고, 이를 이용하여 SciMark 벤치마크 프로그램[9]을 분석한다. 분석 내용에는 메쏘드 호출의 종류, 라이브러리 객체 함수와 네이티브 함수 호출 비율, 수행 시간이 JIT 컴파일 시간 보다 작 은 함수의 비율, 한번 호출되는 함수들의 특성, 버추얼 콜의 다형성 정도, 호출되는 메쏘드의 사이즈별 분산도가 포함된다.

본 논문의 구성은 다음과 같다. 제 2장에서는 본 연구와 관련된 Kaffe와 SciMark를 소개하고, 관련 연구를 기술한다. 제 3장은 동적 분석을 위하여 Kaffe를 확장한 내용을 다루며, 제 4장에서는 확장 구현된 Kaffe를 이용하여 성능 벤치마크 프로그램인 SciMark의 런타임 특성을 분석한 내용을 다룬다. 그리고 마지막으로 결론 및 향후 계획을 기술한다.

2. 배경 및 관련 연구

본 장에서는 본 논문에서 사용하고 있는 Kaffe JVM과 수치 계산에 기반을 둔 성능 벤치마크 프로그램인 SciMark 그리고 관련 연구에 대해 기술한다.

2.1 Kaffe JVM

Kaffe는 C 언어로 구현되어 있으며, 가장 널리 사용되고 있는 오픈 소스 JVM 중 하나이다. 다양한 프로세서를 지원하는 Kaffe는 클래스 라이브러리로 GNU Classpath를 사용하며, GNU GPL 라이센스를 적용받기 때문에 소스의 활용 및 변경이 자유롭다. bytecode verifier의 일부 기능 부족 등 JVM의 표준을 완벽하게 지원하지 못하지만, Tomcat, Ant, Eclipse 등과 같은 거의 모든 응용 프로그램들을 수행할 수 있는 환경을 제공한다. Kaffe는 불필요한 컴파일을 최소화하기 클래스가 로드될 때 모든 메쏘드를 컴파일하지 않고 실제 메쏘드가 호출되는 시점에서 컴파일할 수 있게 하는 Trampoline 기반 JIT 컴파일러를 지원한다[7].

2.2 SciMark

SciMark 2.0은 NIST에서 개발한 과학 및 수치 계산을 기반으로 하는 Java 벤치마크이다. 본 벤치마크 프로그램의 최종 목표는 수치 계산 측면에서 여러 Java 플랫폼의 JVM/JIT 컴파일러의 동작을 좀 더 잘 이해하는 것이며, 이 분야에서 아주 유용한 것으로 증명된 Matlab과 Java Linpack 벤치마크와 유사하다. 다섯 개의 계산 커널은 Fast Fourier Transform (FFT), Jacobi Successive Over-relaxation (SOR), Monte Carlo 적분, Sparse ma-

trix multiply, dense LU matrix factorization로 구성된다. 이들 커널은 JVM/JIT 컴파일러가 이들 형태의 알고리즘을 이용하는 응용 프로그램에서 얼마나 잘 수행되는 가에 대한 지표를 제공하기 위해 선택되었다. 문제의 크기는 의도적으로 메모리 구조의 효과를 분리시키기 위해 작게 선택되고 내부 JVM/JIT와 CPU 이슈에 초점을 두고 있다[9].

2.3. 관련 연구

D. Gregg 외 2인은 Java 프로그램의 메쏘드 레벨 런타임 분석과 관련된 연구 결과를 발표하였다[6]. 이 연구는 Kaffe VM의 인터프리터를 확장하고, SPEC JVM98과 Java Grande를 분석하였다. Gregg의 연구는 본 연구의 기초를 제공하고 있지만, 인터프리터 모드에서 수행함으로써 JIT와 관련된 분석이 없다는 점에서 본 논문과 차별된다. 본 논문은 JIT 컴파일러를 확장하여 관련 데이터를 분석한다.

3. Kaffe의 확장

본 논문의 런타임 분석을 위해 최신 버전인 Kaffe 1.1.7을 확장하였다. 확장된 부분은 크게 두 부분으로 나누어진다. 하나는 JIT 컴파일러의 확장이며, 다른 하나는 클래스 로더의 기능을 확장하는 것이다. 본 절에서는 Kaffe를 확장한 내용에 대해 기술한다.

3.1 JIT 컴파일러의 확장

본 논문은 Java 응용 프로그램의 런타임 특성을 분석할 수 있는 자료를 얻기 위하여 Kaffe의 JIT 컴파일러를 확장하였다. JIT 컴파일러의 확장은 바이트 코드 분석 단계와 네이티브 코드 생산 단계의 두 가지 측면에서 이루어졌다. 바이트 코드 분석 단계에서의 확장은 주로 정적인 자료를 얻기 위함이다. 대표적인 것으로 각 메쏘드의 사이즈 및 메쏘드의 특징과 같은 자료이다. 이들은 정적으로 얻을 수 있는 자료이기 때문에 분석 단계를 확장하여 추출하였다.

정적으로 발췌할 수 있는 자료와 달리 런타임 자료를 발췌하기 위해서는 네이티브 코드 생산 모듈을 확장하여야 한다. 이를 위하여 Kaffe의 프로파일 기능을 참조하였다. Kaffe는 Java 응용 프로그램의 런타임 특성을 추출할 수 있도록 프로파일 기능을 지원한다. 프로파일 기능은 JIT 컴파일 시에 프로파일을 수행하는 네이티브 인스락션들을 추가하는 방법을 사용한다. 각 메쏘드들의 네이티브 코드에는 관련 자료를 발췌할 수 있는 인스트락션을 포함된다. 본 논문의 자료를 발췌하기 위하여 이 기능을 확장하였다. 즉 메쏘드 호출 바이트코드에 대한 네이티브 코드 생산 시에 관련 자료를 추출할 수 있도록 추가적인 인스트락션을 포함시키는 것이다.

3.2 클래스 초기화 모듈 확장

정적 초기화 메쏘드(static initializer) 또는 정적 필드 초기화는 바이트 코드를 통해 호출되는 것이 아니라 JVM이 클래스를 최초로 로드할 때 JVM에 의해 호출된다[8]. Kaffe는 이를 지원하기 위한 함수 processClass()를 제공한다. 본 논문에서는 클래스 초기화 함수에 대한 정보를 발췌하기 위하여 본 함수를 확장하였다. 클래스의 정적 초기화 메쏘드는 한번만 수행됨이 보장되기 때문에 동적인 자료 수집이 필요치 않다. 따라서 네이티브 코드의 추가가 이루어질 필요가 없다.

3.3 확장 모듈의 검증

Kaffe는 설치가 완료된 후 제대로 동작하는 가를 점검할 수 있는 테스트 프로그램을 제공한다. 210여 개의 Java 응용 프로그램으로 이루어진 테스트 프로그램은 JVM에서 제공해야 할 다양한 기능들에 대한 시험을 진행한다. 본 논문에서 확장한 Kaffe의 동작 과정을 검증하기 위하여 테스트 과정을 수행하였고, 모두 정상적으로 수행함을 보였다. 이에

덧붙여 다음 장에서 다룰 SciMark 벤치마크 프로그램을 정상적으로 수행하는 것으로 검증하였다.

4. SciMark 수행 특성

본 장에서는 확장된 Kaffe를 이용하여 SciMark 벤치마크의 런타임 특성을 메쏘드 수준에서 분석한다.

4.1 메쏘드 호출 분석

Java 프로그램의 메쏘드 호출은 크게 두 가지로 이루어진다. 바이트 코드의 메쏘드 호출 인스트락션(invokevirtual, invokestatic, invokespecial, invokeinterface)에 의해 호출되는 경우와 JVM에 의해 호출되는 것으로 구분된다. 객체 메쏘드 및 일반적인 클래스 메쏘드의 호출은 전자에 해당되는 반면 정적 초기화 메쏘드(static initializer)의 호출은 후자에 해당된다. 본 절에서는 두 종류의 메쏘드 호출의 분석에 대해 기술한다.

4.1.1 메쏘드 호출 형태

JVM 표준[8]은 메쏘드 호출에 사용하는 4 가지의 인스트락션을 제공하며, 각 인스트락션이 사용되는 경우에 대해 명확하게 기술하고 있다. <표 1>은 메쏘드 호출 인스트락션에 따라 SciMark의 메쏘드 호출 형태를 분석한 것이다. FFT와 LU의 경우 거의 대다수 호출이 static 메쏘드임을 보여주며, 이는 객체지향 프로그래밍의 특성을 반영하지 못함을 알 수 있다. <표 1>에서 static initializer는 클래스 또는 인터페이스 초기화 함수로 바이트 코드를 통해 호출되는 것이 아니라 JVM이 클래스를 최초로 로드할 때 수행하는 메쏘드이다. 이 메쏘드는 클래스가 처음 JVM에 로드되는 시점에 단 한 번 만 수행되기 때문에 네이티브 코드로 변환하는 것이 오히려 성능을 저하시킬 수 있기 때문에 바람직하지 않을 수 있다. 이와 관련된 부분은 다음 절에서 보다 상세하게 다룬다.

〈표 1〉 메쏘드 호출 형태 분석

	invoke virtual (%)	invoke special (%)	invoke static (%)	invoke inter-face (%)	static initi-alizer (%)	total (106)
FFT	1.84	0.42	97.58	0.16	0.01	0.75
LU	0.42	0.06	99.50	0.02	0	5.20
Monte Carlo	99.92	0.02	0.05	0.01	0	16.8
SOR	62.11	8.93	25.32	3.47	0.16	0.04
Sparse	57.22	10.08	28.60	3.92	0.18	0.03
Total	73.90	0.07	26.01	0.03	0	22.82

4.1.2 메쏘드 호출 타겟

자바 프로그램의 수행에서 라이브러리 클래스의 사용은 불가피하며, 라이브러리 메쏘드를 많이 사용할수록 보다 나은 성능을 가져올 가능성이 높다. <표 2>는 프로그램 수 행중 동적으로 호출되는 메쏘드들의 분포를 나타낸다. <표 2>에서 total method는 수행된 전체 메쏘드의 수를 나타내며, 메쏘드가 중복해서 수행될 경우 수행 횟수만큼 카운트된다. MonteCarlo의 경우 라이브러리 메쏘드의 수행이 타 커널에 비해 현저히 떨어짐을 볼 수 있다. 이는 대다수의 메쏘드가 응용 수준에서 작성되었으며, 메쏘드가 수행되는 횟수도 타 커널에 비해 상대적으로 규모가 매우 크기 때문이다. 이 같은 점은 다음 장에서 다룰 메쏘드 사이즈 별 분석에 많은 영향을 끼침을 볼 수 있다.

〈표 2〉 수행된 메쏘드의 분포

	Library (%)	Native Lib.(%)	Total Method
FFT	90.94	0.42	0.75e6
LU	99.76	0.06	5.20e6
Monte Carlo	0.14	0.02	16.80e6
SOR	70.73	8.91	0.04e6
Sparse	79.93	10.06	0.03e6
Total	26.06	0.07	22.82e6

4.2 JIT 관련 분석

JIT 컴파일러를 사용하여 바이트 코드를 네이티브 코드로 변환하여 수행함으로써 여러 번 수행되는 메쏘드의 경우 성능을 향상시킬 수 있다. 그러나 단 한번만 수행하는 메쏘드의 경우나 컴파일 시간이 수행 시간보다 많이 소요되는 메쏘드의 경우에는 그 효과가 줄어들거나 오히려 성능이 저하되는 경우가 발생할 수 있다. 본 장에서는 이와 관련하여 일회성 호출 또는 컴파일 시간이 수행 시간보다 긴 함수에 대해 분석한다.

4.2.1 일회 호출 메쏘드

정적 초기화 메쏘드(static initializer)는 바이트 코드를 통해 호출되는 것이 아니라 JVM이 클래스를 최초로 로드할 때 JVM에 의해 호출된다. 따라서 이 메쏘드는 단 한번 만 수행되기 때문에 네이티브 코드로 변환하는 것이 바람직하지 않을 수 있다. <표 3>은 정적 초기화 메쏘드의 전체 호출 메쏘드에 대한 비율 그리고 함수를 컴파일하는 데 소요된 시간과 메쏘드의 수행 시간 비를 나타낸다. 이 같은 함수의 컴파일에 소요되는 오버헤드를 보다 정확하게 분석하기 위해서는 인터프리터로 수행할 때 소요되는 시간을 추출하고, JIT 컴파일 시간과 네이티브 코드 수행 시간을 비교할 수 있어야 한다. 인터프리터를 확장에 많은 시간이 소요되므로 이 부분은 향후 연구에서 다루도록 한다.

〈표 3〉 정적 초기화 메쏘드

	method/all(%)	jit/exec(%)
FFT	0.03	42.5
LU	0	58.4
Monte Carlo	0	57.7
SOR	0.69	56.4
Sparse	0.08	58.9

4.2.2 짧은 런타임 메쏘드

본 절에서는 메쏘드의 런타임 시간이 JIT 컴파일 시간보다 짧은 메쏘드에 대해 분석하였다. 컴파일 시간보다 짧은 수행 시간을 갖는 경우, JIT 컴파일은 오히려 성능을 저하시킬 수 있는 요인이 된다. <표 4>는 컴파일 시간보다 짧은 수행 시간을 갖는 메쏘드가 전체 수행되는 메쏘드의 0.08%를 라는 것을 보여주고, 이들 함수 중 약 40% 정도가 한번만 수행됨을 나타낸다. 비록 이들이 성능을 저하시킬 수 있는 요인이긴 하지만, 전체 수행되는 메쏘드에 비해 아주 미미한 정도이므로 무시할 수 있는 크기임을 알 수 있다.

〈표 4〉 JIT 시간 미만의 런타임을 갖는 메쏘드

	1 회(%)	2 회(%)	3 회(%)	비율(%)
FFT	40.67	9.77	2.13	0.08
LU	40.28	9.72	2.12	0.02
Mon	39.36	9.75	2.13	0
SOR	40.21	9.79	1.96	1.62
Sparse	40.07	9.75	2.13	1.83
Total	40.13	9.76	2.09	0.01

4.3 다형성 분석

객체 지향 프로그래밍에서의 다형성(polymorphism)은 상속 관계에 있는 객체들에 따라 달라지는 메쏘드를 구분하지 않고 사용하게 함으로써 생산성과 확장성을 제공한다. Java에서의 다형성은 method overriding으로 제공되며, 호출되는 메쏘드는 런타임에 결정된다. 따라서 method overriding이 없는 경우 호출되는 메쏘드를 정적으로 결정할 수 있도록 최적화시킴으로써 성능을 향상시킬 수 있다. 본 절에서는 다형성 정도가 어떻게 분포되는가를 파악하기 위하여 invokevirtual의 타겟을 분석하였다. <표 5>, <표 6> 그리고 <표 7>에 나타난 바와 같이 SciMark의 다형성 정도는 그렇게 높지 않음을 보이기 때문에 타겟 메쏘드의 호출을 컴파일 타임에 결정하는 최적화를 통해 성능을 시킬 수 있음을 보인다.

〈표 5〉 전체 메쏘드의 다형성 분포

	1 회(%)	2 회(%)	3 회(%)	4 회(%)	5 회(%)
FFT	99.06	0.90	0.03	0	0
LU	99.93	0.07	0	0	0
Monte	99.98	0.02	0	0	0
SOR	88.26	11.37	0.33	0.04	0.01
Sparse	86.66	12.92	0.37	0.04	0.01

〈표 6〉 라이브러리 메쏘드의 다형성 분포

	1 회(%)	2 회(%)	3 회(%)	4 회(%)	5 회(%)
FFT	98.89	1.08	0.03	0	0
LU	99.93	0.07	0	0	0
Monte	83.15	16.32	0.47	0.05	0.01
SOR	83.07	16.40	0.47	0.05	0.01
Sparse	83.10	16.37	0.47	0.05	0.01

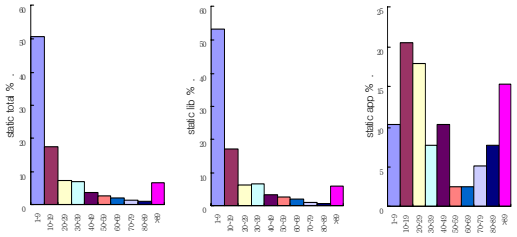
〈표 7〉 응용 메쏘드의 다형성 분포

	1 회(%)	2 회(%)	3 회(%)	4 회(%)	5 회(%)
FFT	100.00	0	0	0	0
LU	99.99	0.01	0	0	0
Monte	100	0	0	0	0
SOR	99.99	0.01	0	0	0
Sparse	99.98	0.02	0	0	0

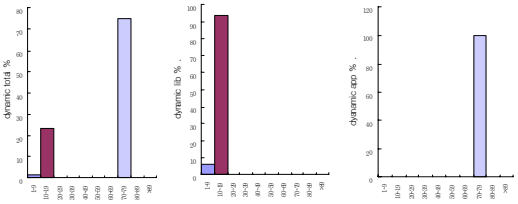
4.4 메쏘드 사이즈 분석

작은 사이즈의 메쏘드는 많은 JVM에서 사용하는 최적화 기술인 코드 code inlining의 주요 타겟이 될 수 있으며, 객체 지향 프로그래밍의 표준으로 간주된다. 또한 code refactoring은 결과적으로 작은 사이즈의 메쏘드를 초래하기 때문에 객체지향 프로그래밍에서의 메쏘드의 사이즈는 객체지향 프로그래밍 및 성능 최적화의 매우 중요한 측도가 된다. 본 절에서는 메쏘드 사이즈의 분포에 대해 분석한다.

[그림 1]은 정적인 방법으로 벤치마크 프로그램에서 호출되는 메쏘드들의 사이즈에 따른 분포를 나타낸다. 그림에서 전체 메쏘드들의 분포는 작은 사이즈에 많이 분포되어 있음을 보여주고 있다. 이 같은 분포는 순수 응용 프로그램의 수준에서는 다소 차이가 있을 것으로 판단된다.

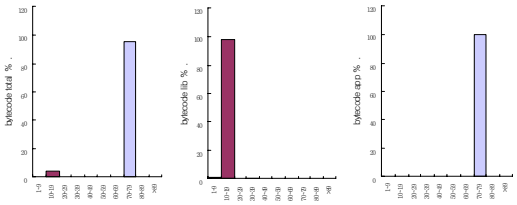


[그림 1] 정적 분석을 통한 메쏘드 호출 분포



[그림 2] 동적 분석을 통한 메쏘드 호출 분포

[그림 2]는 프로그램 수행 중에 동적으로 호출되는 메쏘드의 사이즈별 분포도를 나타낸다. 정적인 분석 때와는 달리 특정 규모의 메쏘드에 집중적인 호출이 발생함을 보여준다. 그러나 이들 분석을 5개의 커널 별로 분석하면, 라이브러리의 경우 정적인 분석과 유사한 형태를 보이나, 응용은 특정 사이즈에 집중되는 현상을 보인다. 그럼에도 불구하고 동적 분석의 결과가 특이하게 나타나는 것은 <표 1>과 <표 2>에서 볼 수 있듯이 MonteCalro의 경우 호출되는 메쏘드의 수가 전체 호출의 약 74%를 차지할 정도로 크며, 메쏘드의 호출이 응용에 집중되어 있기 때문이다. 또한 메쏘드 호출의 23%를 점유하는 LU 커널의 경우 거의 100%가 라이브러리 메쏘드를 호출함으로써 라이브러리의 분포의 특성을 결정하고 있다. 이 같은 점을 고려하더라도 [그림 2]에 나타난 특성은 응용 측면에서 객체 지향성 및 JVM의 성능 최적화의 특성을 나타내기에 부족한 부분이 있다.



[그림 3] 바이트 코드 계산을 통한 메쏘드 사이즈 분포

[그림 3]은 [그림 2]의 경우처럼 메쏘드의 사이즈에 따른 분포를 나타낸다는 점에서 유사하나 전체 수행된 바이트코드의 비율을 나타낸다는 점에서 다르다. 즉 전체 수행된 바이트 코드 중 라이브러리는 10~19 규모의 바이트코드로 구성된 메쏘드가 거의 100%에 가까운 율을 보이고 있음을 볼 수 있으며, 응용의 경우 70~79 규모의 크기를 갖는 메쏘드가 전체 수행되는 응용 바이트코드를 제공하고 있음을 보인다. 이 같은 특성은 이미 전 절에서 언급한 것과 같이 SciMark는 많은 JVM이 사용하는 최적화 기능을 제대로 반영한 성능을 보여주기에 부족한 면이 많음을 보여준다.

5. 결 론

본 논문은 Java 응용 프로그램의 런타임 특성들을 추출하기 위하여 Kaffe를 확장하였으며, 이를 이용하여 SciMark 벤치마크 프로그램의 런타임 특성을 분석하였다. 분석한 결과를 보면 SciMark는 많은 JVM 들이 성능 향상을 위하여 사용하는 최적화 기능을 제대로 반영하기 어려운 런타임 특성을 보인다. 특히 메쏘드 사이즈에 따른 분포를 보면 특정 커널 모듈이 특정 규모의 메쏘드를 대규모로 수행함으로써 일반적인 응용에 나타날 수 있는 성능을 측정할 수 있다고 보기 어렵다고 판단된다.

본 연구는 메쏘드 수준의 런타임 특성을 분석할 수 있는 기반을 제공하고 있으나, JIT 컴파일러를 사용함으로써 발생하는 성능 저하 부분에 대해 보다 심도있는 연구가 필요하다. 이는 향후 연구에서 다룰 예정이다. EEMBC GrinderBench[3]와 SPEC JVM98[10]의 런타임 특성을 분석하고, CDC 기반 JVM 들의 성능을 비교 분석하는 것도 향후 계획의 하나이다.

참 고 문 헌

[1] Arnold, M., S. Fink, V. Sarkar and P. Sweeney, “A comparative study of static and dynamic heuristics for inlining”, *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, (2000), pp.52-64.

[2] Cavazos, J. and M. O’Boyle, “Method-specific dynamic compilation using logistic regression,” *OOPSLA,* (2006), pp.229-240.

[3] EEMBC, *Calculating GrinderBench*, http://www.eembc.org/techlit/datasheets/calculatingmark.pdf.

[4] Goetz, B., *Dynamic compilation and performance measurement,* http://www-128.ibm.com/dev eloperworks/library/j-jtp12214, 2004.

[5] Gosling, J., B. Joy, G. Steele and G. Bracha, *Java Language Specification, 3rd Edition*, Prentice Hall, 2005.

[6] Gregg, D., J. Power and J. Waldron, “A method-level comparison of the Java Grande and SPEC JVM98 benchmark suite”, *Concurrency and Computation : Practice and Experience*, Vol.17, No.7-8(2005), pp.757-773.

[7] Kaffe.org, *Document of Kaffe 1.1.7 release*, http:// www.kaffe.org/document

[8] Lindholm, T. and F. Yellin, *The Java(TM) Virtual Machine Specification,* 2nd Edition, Prentice Hall, 1999.

[9] Pozo, R. and B. Miller, *How fast is your Java platform for number crunching?*, http://math.nist. gov/scimark2.

[10] Standard Performance Evaluation Corporation, *SPEC JVM98 Benchmarks*, http://www.spec.org/jvm98.

◈ 저 자 소 개 ◈

신 범 주 (bjshin@pusan.ac.kr)

경북대학교 전자공학과를 졸업하고, 동 대학의 컴퓨터공학과에서 공학석사 및 박사를 취득하였다. LG정보통신(주) 및 한국토지공사 전산실에 근무한 바 있으며, 한국전자통신연구원 시스템소프트웨어연구실 책임연구원으로 근무하였다. 밀양대학교 컴퓨터공학부 교수로 재직하다 밀양대학교가 부산대학교에 통합된 후 부산대학교 바이오정보전자전공 교수로 재직하고 있다. 주요 연구분야는 u-health, 메디컬 분야와 연계되는 인프라 소프트웨어이다.

이 창 우 (leecw@kunsan.ac.kr)

1996년 경일대학교 컴퓨터공학과 공학사를 취득하였고, 1998년과 2004년에 경북대학교 컴퓨터공학과 공학석사 및 공학박사 학위를 취득하였다. 1998년부터 2001년까지 포항 1대학 전산정보처리학과 전임강사로 근무하였고, 2004년부터 현재까지 군산대학교 컴퓨터정보과학과 조교수로 근무하고 있다. 관심 연구분야는 인공지능, 텔레매틱스, 패턴인식, 컴퓨터비젼이다.

이 완 직 (wjlee@pusan.ac.kr)

경북대학교 통계학과를 졸업하고, 동 대학의 컴퓨터 공학과에서 석사 및 박사 학위를 취득하였다. 1997년부터 밀양대학교 정보통신공학과에서 재직하였으며, 대학교 통합에 의해 2006년부터 부산대학교 바이오시스템 공학부 바이오정보전자전공에 재직 중이다. 주요 연구분야는 센서 네트워크, USN 응용, 시스템소프트웨어 등이다.