LaTTe: a Java VM Just-in-Time Compiler

이승일 이준표 문수묵
서울대학교 전기공학부

요 약

자바 언어는 네트웍 환경을 위한 언어로 개발되어 embedded system에서 뿐만 아니라 enterprise 서버에 이르기까지 점차 그 영역을 확대하고 있다. 이렇듯 자바 언어의 쓰임이 확대됨에 따라 자바로 만들어진 q프로그램을 빠르게 수행시킬 수 있는 자바 가상머신의 필요성이 증대되고 있다.

본 논문에서는 JIT 컴파일러 방식에 기반한 LaTTe 자바 가상머신을 소개한다. LaTTe에 탑재된 JIT 컴파일러는 효율적으로 복사 명령을 제거할 수 있는 빠른 레지스터 할당기와 중복 연산 제거, 죽은 코드 제거, 메소드 인라이닝과 같은 최적화 기법을 포함하고 있다. LaTTe 자바 가상 머신은 그밖에 효율적인 쓰레기 처리, 빠른 모니터 락 처리, 그리고 필요에 따른 예외 처리를 수행한다. 이와 같이 구성되어 있는 LaTTe 자바 가상 머신은 자바 프로그램을 매우 빠르게 수행할 수 있다.

SPEC jvm98에 대하여 실험한 결과에 따르면, LaTTe는 실행 시간에 있어서 SUN에서 제공하는 JDK 1.1.6과 비교하여 평균 2.3배 정도 향상된 결과를 얻을 수 있었다.

제 1 절 서론

자바 언어는 이식성과 보안성의 장점으로 인하여 최근에 인터넷 프로그래밍 언어로 각광받는 언어이다 [1] . 이러한 자바 언어의 영역은 embedded system에서 뿐만 아니라 enterprise server에 이르기까지 그 영역을 확장시켜나가고 있다. 자바 언어의 가장 뛰어난 장점이라 볼 수 있는 platform independence는 자바 가상 머신을 통해서 자바 프로그램을 수행시킨다는 점에 기인한다 [2] . 이 자바 가상 머신은 자바 프로그램을 컴파일하여 얻어지는 바이트 코드들을 수행시키는 기반이 되는 프로그램이며 이들 바이트 코드를 빠르게 수행하기 위해 개발된 방법이 JIT 컴파일러이다 . JIT 컴파일러는 자바 가상머신의 한 부분으로서 바이트 코드를 해당 기계의 기계어로 변환시켜준다. 변환된 기계어를 통해서 수행하는 방식은 바이트 코드를 직접 해석기를 통해서 수행하는 방식보다 뛰어난 성능을 보인다 [3] . 이는 기계어로 변환하면서 여러가지 최적화 과정을 거쳐서 보다 효율적으로 프로그램의 수행을 지원하고 일단 변환된 기계어는 다음에 같은 메소드를 수행할때 변환된 코드가 불리어서 다시 변환과정을 거치지 않아도 되기 때문이다.

자바 가상 머신에는 JIT 컴파일러 외에도 여러가지 구성요소가 같이 들어있어 자바 프로그램의 수행을

도와준다 [2] . 이중 대표적인 것이 쓰레기 처리기와 예외 처리기, 락을 다루는 모니터들이다.

이 논문에서는 본 연구실에서 개발한 LaTTe 자바 가상 머신을 소개하고 각 구성 요소들을 소개하기로 한다. 제 2절에서는 LaTTe 자바 가상머신이 가지고 있는 JIT 컴파일러와 그밖의 구성요소들에 대해 살펴보고, 제 3절에서는 LaTTe 자바 가상머신의 성능을 SUN JDK 1.1.6과 비교하여 실험한 결과를 살펴보고, 제 4절에서 마무리를 하도록 한다.

제 2 절 본론

2.1 레지스터 할당기

2.1.1 바이트 코드를 기호 레지스터를 사용하는 중간코드로 변환

LaTTe 자바 가상 머신이 가지고 있는 JIT 컴파일러에서 바이트 코드를 기계어로 바꾸기 위해 먼저 하는 일은 대상 메소드 내의 각 바이트 코드를 해당하는 중간코드로 변환하는 일이다. LaTTe가 가지고 있는 JIT 컴파일러가 만들어 내는 중간 코드는 실제 레지스터가 아닌 기호 레지스터를 사용한다는 것을 제외하면 최종코드와 동일하다. 기호 레지스터들은 해당 메소드의 지역 변수들과 연산 스택의 위치를 나타낸다. 이들 기호 레지스터들은 레지스터 할당 과정을 거쳐서 실제 레지스터로 변환된다 [4] .

기호 레지스터들은 이 기호 레지스터가 지역 변수에 속한 것인지 연산 스택에 위치한 것인지 나타내는 정보와 타입을 나타내는 정보, 그리고 서로 다른 기호 레지스터를 구분하는 숫자로 이루어져 있다. 각 메소드들이 사용하는 지역변수의 크기와 연산 스택의 크기는 자바 프로그램을 컴파일할 때 결정되어 바이트 코드와 함께 정보로서 저장되므로 몇개의 기호 레지스터를 사용할지 미리 결정할 수 있다.

2.1.2 기호 레지스터를 실제 레지스터로 변환

일반적인 최적화 컴파일러에서는 그래프 컬러링 알고리즘에 기반한 전역 레지스터 할당기에 의해 기호 레지스터를 실제 레지스터로 변환한다 [5] . 그러나 이 그래프 컬러링 알고리즘은 데이터 흐름 분석과 같이 시간이 오래 걸리는 작업을 필요로 하기 때문에 JIT 컴파일러에서는 사용하기 힘들다. 따라서 LaTTe의 JIT 컴파일러는 빠르게 레지스터를 할당해주면서 효율적으로 복사 명령어들을 제거할 수 있는 새로운 알고리즘을 사용한다.

이 알고리즘은 tree region을 기본 단위로 하여 레지스터를 할당한다. tree region 은 하나의 entry와 여러개의 exit을 가지는 tree모양의 subgraph 이다. 이들 tree region들을 reverse post-order로 traversal하면서 각각에 대해 backward sweep과 forward sweep 알고리즘을 적용한다.

backward sweep backward sweep에서는 region을 post-order로 traversal하면서 각 명령어에 대한 preferred assignment를 정해준다. 즉 각 명령어에서 다음 region 의 레지스터 할당 정보나 calling conven-

tion에 따른 정보를 이용하여 destination 레지스터의 할당 정보를 미리 알려주는 것이다. 이 정보는 forward sweep에서 destination 레지스터를 할당하는데 사용된다. 이 과정을 통해 region 사이에서 레지스터 할당의 불일치로 발생할 수 있는 발생하는 복사 명령어를 제거하고, calling convention을 맞추기 위해 발생하는 복사 명령어를 최소화 한다.

forward sweep forward sweep에서는 region을 depth-first-order로 traversal하면서 실제 레지스터 할당을 수행한다. 레지스터 할당을 할 때 기호 레지스터와 실제 레지스터 사이의 사상을 저장하는 h_map을 이용한다. 각 기호 레지스터가 define이 될때 h_map에 삽입하고 마지막 use될 때 h_map에서 제거 한다. 복사 명령어의 경우 source와 destination에 대해 동일한 map을 사용하도록 하여 복사 명령을 제거한다. 따라서 이 과정을 통해 복사 명령어는 실제 코드 대신 map만 바꾸어줌으로서 레지스터 융합의 효과를 생성한다.

2.2 중복 연산 제거(redundancy elimination)

중복 연산 제거 기법은 대표적인 최적화 기법 중 하나로 같은 결과를 낳는 중복되는 연산을 모두 제거해서 최소한의 연산만을 코드가 실행하도록 하는 기법이다. 이 기법을 구현하기 위한 알고리듬으로는 value numbering과 부분 중복 제거(partial redundancy elimination)가 대표적이다 [5] .

JIT상에서는 최적화에 소요되는 시간역시 최소화 하는 것이 목적이므로 변환 시간에 부과되는 부담과 생성된 코드의 질 사이에서 적절한 trade-off가 필요하다. LaTTe에서는 value numbering 알고리즘을 region 단위로 적용하여 각 명령어를 한번씩만 방문함으로서 최적화에 소요되는 시간을 최소화했다 [6] .

이 알고리듬은 2단계로 이루어져 있는데, 처음 단계에서는 depth-first-order로 제어 흐름 그래프 상의 명령어들을 방문하여, 각 명령어에 value number를 할당하고 중복여부를 표시하고, 두번째 단계에서는 역 depth-first-order로 명령어를 방문하면서 중복된 명령어를 제거하고 대신에 적절한 복사 명령어를 삽입한다. 또한 value numbering 알고리듬을 확장해서 상수 접기(constant folding)과 상수 전달(constant propagation)도 중복 연산 제거와 함께 이루어진다

2.3 죽은 코드 제거(dead code elimination)

죽은 코드 제거는 계산 결과가 사용되지 않는 명령어들을 제거하는 최적화 기법이다. 자바의 바이트코드상에는 죽은 코드가 거의 없기 때문에 별로 효과가 없지만, value numbering을 통해 중복 연산 제거나 상수 접기, 상수 전달등을 수행하면서 죽은 코드가 많이 생기게 되고 따라서 죽은 코드 제거의 효과도 커지게 된다 [6] . 어떤 코드가 죽은 코드인지는 생존 분석(live analysis)를 통해 알 수 있다. 전통적인 생존 분석은 생존 정보를 반복적으로 갱신해서 일정한 값으로 수렴하게 만들지만, JIT 컴파일러상에서 수행하기에는 시간이 너무 오래걸린다. 따라서 LaTTe에서 구현된 생존분석 방식은 모든 명령을 한번씩만 방문하는 방법을

이용하여 여기에 소요되는 시간을 줄이고 있다. 하지만, 한번씩만 명령어들들 방문하기 때문에 생존 정보가 정확하지 않아 실제로는 죽은 코드지만 제거하지 못하는 경우도 생긴다.

그러나 JIT 컴파일러라는 제약조건을 생각할 때 제거되지 않는 죽은 코드의 양이 크지 않으므로 최소한의 시간을 소요시키면서 최대한의 효과를 얻는 방법이라 할 수 있다.

2.4 메소드 인라이닝

메소드 인라이닝이란 메소드 사이의 호출 부담을 줄이기 위해 한 메소드 안에 불려지는 메소드를 같이 넣어서 한꺼번에 코드변환을 적용시키는 것이다. 현재 메소드 인라이닝은 어떤 메소드가 불리는지를 컴파일 시간에 알수 있는 final, private, static 메소드에 대해 시도하고 있다.

2.4.1 메소드 인라이닝을 하는 이유

자바 언어는 객체 지향언어이다. 따라서 어떠한 객체의 필드에 접근하기 위한 getter와 setter를 하나의 메소드 형태로 만들어 놓는 경우가 많다. 이때 getter와 setter가 하는 일은 단순히 해당하는 객체의 필드값을 읽어오거나 필드값을 고쳐주는 일이다. 이러한 단순한 작업을 위해서 메소드를 호출한다면 호출 규칙(calling convention)을 맞춰주기 위해 여러 작업이 필요해진다. 이들 작업중 대표적인 것은 호출 경계(call boundary)에서 caller save register의 살아있는 값을 다른 곳으로 저장하는 일과 호출하는 메소드에 인자를 넘겨주기 위해 값을 복사하는 일, 그리고 호출되는 메소드에서는 메소드에서 사용할 callee save 레지스터의 값들을 저장해주고, 스택 포인터를 고쳐주는 일등이 필요하다. 이러한 일들 때문에 메소드의 크기가 작을 경우 실제 메소드 안에서 해야 하는 일들보다 그 메소드를 호출하기 위한 부담이 더 커지게 된다. 이러한 부작용을 줄이기 위해 크기가 작은 메소드에 대해서 메소드 인라이닝을 하면 전체 프로그램 수행시간을 단축하는데 도움을 줄 수 있다.

2.4.2 메소드 인라이닝의 효과

메소드 인라이닝을 함으로서 얻어지는 이득은 우선 메소드 호출에 들어가는 부담이 최소화가 된다. 메소드 호출에 들어가는 부담은 인자 전달을 위한 레지스터 사이의 복사 명령과 살아있는 caller save 레지스터의 값을 저장하기 위한 작업이다. 메소드를 인라인 할 경우 인자 전달은 기호 레지스터사이의 복사 명령어로 변환이 되어 레지스터 할당기에서 이러한 복사 명령어를 제거해 줄 수 있다. 또한, caller save 레지스터 값을 저장할 이유가 없어지기 때문에 불필요한 복사 명령어가 생성되지 않는다. 인라인된 메소드에서도 스택 포인터를 갱신하는 등의 명령어를 생성할 필요가 없기 때문에 전체 수행되는 명령어의 수를 줄일 수 있다.

메소드 인라이닝을 통해 얻을 수 있는 또하나의 이득은 한 메소드의 크기가 커지게 된다는 점에서 기인한다. 대부분의 최적화 기법이 메소드를 최적화의 단위로 하기 때문에 메소드의 크기가 커질 수록 좀더 많은 최적화 기회가 생기게 된다. 따라서 메소드 인라이닝을 통해 더 효율적인 코드를 생성할 수 있다.

이러한 메소드 인라이닝은 장점과 함께 그에 따르는 부작용도 있다. 그 중 대표적인 것이 한 메소드에서 사용되는 기호 레지스터의 갯수가 인라인 되는 메소드의 기호 레시스터 갯수까지 포함하게 되어 레지스터 할당기의 입력으로 들어가므로 결과적으로 레지스터 압력(register pressure)이 높아진다. 따라서 인라이닝을 하지 않을 경우 버림(spill)이 안 발생하던 메소드가 인라이닝을 하는 경우 버림을 발생시키는 경우가 생길 수 있다. 이때는 버림 때문에 발생하는 메모리 접근 명령이 늘어나므로 오히려 코드의 효율이 떨어져 프로그램 수행시간을 늘어나게 하는 경우가 발생할 수 있다. 또한 인라인 되는 메소드는 코드가 중복되어 코드의 크기가 커지게 되는 현상도 발생한다.

2.4.3 앞으로 해야할 일

지금까지의 메소드 인라이닝은 컴파일 시간에 어떤 메소드를 호출할 것인지 정해지는 private, final, static 메소드만을 대상으로 삼는다. 그러나 대부분의 메소드들은 컴파일 시간에 어떤 메소드를 호출할지 정할 수 없는 virtual 메소드로 이루어져 있다. 따라서 실제로 메소드 인라이닝을 적용할 대상을 많이 찾을 수 없어 그 효과가 뚜렷하게 나타나지 않는다. 여기에 virtual 메소드 호출을 다이나믹 프로파일링기법을 이용하여 어떤 메소드가 호출될지 예측하여 이 예측된 메소드에 대해 인라인을 한다면 virtual 메소드일지라도 부분적으로 인라인이 가능하다 [7] . 이렇게 되면 좀더 많은 메소드가 인라인되어 메소드 인라이닝으로 얻을 수 있는 이득을 최대화 할 수 있다. 또한 메소드 인라이닝의 부작용을 어떻게 하면 최소화할 수 있을 것인가에 대한 연구가 더 필요하다.

2.5 자바의 모니터 구현

자바의 모니터는 프로그램 내의 critical section에 대해 상호 배타적인(mutually exclusive) 수행을 보장해주는 역할을 한다. 이런 상호 배타적인 수행을 보장하기 위해 해당하는 객체에 대해 모니터 락을 걸어줌으로서 동시에 같은 객체를 접근하지 못하도록 막는다 [1] . 이러한 모니터 락은 바이트 코드 명령인 MONITORENTER와 MONITOREXIT으로 이루어 진다.

2.5.1 모니터 명령의 특징

MONITORENTER와 MONITOREXIT을 수행할 때 나타날 수 있는 상황은 다음의 3가지가 있다.

- 다른 어떤 쓰레드도 락을 소유하고 있지 않아 바로 락을 얻는 경우
- 자신이 그 객체에 대한 락을 가지고 있어, 중복되어 락을 얻는 경우
- 다른 쓰레드가 그 객체에 대한 락을 소유하고 있어, 락이 풀릴 때까지 기다려야 하는 경우

대다수의 단일 쓰레드 벤치마크들은 모니터 명령을 수행할 필요가 없다. 그러나 자바 표준 클래스 라이브러리의 프로그램들은 멀티 쓰레드에서도 잘 동작하도록 만들어져 있어서 위의 첫번째와 두번째 경우에 해당하는 모니터 명령을 자주 수행한다. 또한 다중 쓰레드 벤치마크에서도 세번째 경우는 거의 발생하지 않는 것으로 관찰되었다.

2.5.2 Lightweight monitor

LaTTe에서 구현된 모니터 락은 세번째 경우가 매우 드물다는 관찰에 근거해 첫번째와 두번째 경우에 대해 최적화하여 구현하였다 [8] .

빠르게 락 명령을 수행하기 위해서는, 락과 관련된 모든 정보가 객체의 header에 포함되어 있는 것이 좋다. 그러나 메모리의 지나친 낭비를 줄이기 위해 락 정보를 encoding 된 형태로 객체의 header에 저장한다. 세번째 경우에 필요한 정보인 wait queue나 조건 동기화를 위한 queue는 별도의 hash table 을 이용해 관리한다. 관찰된 벤치마크들에 따르면 전체 객체 중에 모니터 명령의 대상이 되는 객체 수의 비율은 가장 큰 것도 30%이하이고, 평균은 5%도 되지 않는다 [8] .

모니터 명령어는 바이트 코드를 변환할 때 최소한의 부담으로 수행하기 위해 함수 호출을 하는 대신 코드를 직접 인라인 시켰다. 각각 소요되는 명령어 수를 살펴보면, 첫번째와 두번째 경우에 대해서 MONITORENTER시에는 각 9와 11, MONITOREXIT시에는 각 5와 7개의 SPARC 명령어로 구현이 가능하다.

2.6 예외 처리

자바의 예외(exception)가 발생하는 경우는 크게 2가지로 나눌 수 있다. 하나는 throw라는 명령어를 통해 사용자가 인위적으로 예외를 발생시키는 경우고 다른 하나는 바이트코드를 수행하다가 예외 발생 조건이 만족되는 경우이다 [2] . 어느 경우든 예외가 발생하면, JVM이 그 예외를 'catch'하는 예외 처리 루틴(exception handler)을 찾아내서 제어권을 넘겨 주어야 한다.

대개의 프로그램의 경우 예외(exception)은 예외적인 경우에 발생한다. 즉 대부분의 예외처리 루틴들은 실제로는 수행되지 않는다. 현재 LaTTe에서는 이러한 가정에 근거하여 예외처리를 구현하고 있다. 예외가 발생하여 제어권이 예외 처리 루틴으로 옮겨져야 하는 경우 일반적인 분기(branch)명령이나 함수 호출 명령을 사용하지 않고 trap을 이용한다. trap 명령은 이를 처리하기 위한 부담이 일반 분기 명령이나 함수 호출 명령보다 크나 이는 예외의 발생 빈도가 적다는 가정에선 크게 문제가 되지않는다. 오히려 trap 명령을 이용함으로써 변환된 코드에서 예외 처리를 위한 명령어의 수를 줄이고 제어 흐름을 단순하게 할 수 있다. 또한 null pointer exception이나 arithmetic exception의 경우는 명시적으로 trap 명령을 집어 넣지 않고도 OS가 발생시키는 신호를 받아 처리할 수 있다.

어떤 메소드를 JIT 컴파일러를 통해 변환하는 경우, 그 메소드내에 포함된 예외 처리 루틴은 변환하지 않고 실제로 그 루틴의 실행이 필요해질 때까지 변환과정을 미루어 둔다. 이는 발생하지 않는 예외 처리루틴을 변환하지 않음으로써 변환과정에 소요되는 시간을 줄이고, 수행되지 않는 명령어를 만들어 내지 않는다는 장점이 있다.

2.7 메모리 관리

2.7.1 자바에서의 메모리 관리자

자바 언어에서는 명시적인 메모리 소거 명령이 존재하지 않는다. 이 때문에, 메모리 소거의 역할을 자바 가상 머신에게 맡기고 사용자는 이에 관여하지 않게 된다. 이는 자바 언어를 이용하여 프로그램을 개발하는데 발생하는 버그를 줄이고 사용자가 메모리를 사용하는데 편하게 하기 위한 방법이다. 그러나 이러한 특징 때문에 자바 가상머신의 메모리 관리자는 메모리의 할당 뿐 아니라 자동적인 소거의 기능도 가지고 있어야 한다 [2] . 일반적으로 쓰레기 처리기(garbage collector)라 불리는 자바 가상 머신의 한 구성요소가 메모리의 자동적인 소거의 기능을 담당하고 있다. 메모리 관리의 효율성을 위해서는 쓰레기 처리 과정에 소요되는 시간과 메모리 할당에 걸리는 시간을 동시에 중시하여야 한다.

2.7.2 현재 사용되고 있는 쓰레기 처리기

현재 LaTTe에서 구현되어 있는 쓰레기 처리기는 기본적으로 mark-and-sweep 알고리듬을 변형한 형태의 알고리즘을 사용한다 [9] . 기본적인 mark-and-sweep 알고리즘에서 메모리 할당에 걸리는 시간을 감소시키기 위해 mark-compact 알고리듬과 유사한 할당 기법을 사용했다. 전통적인 mark-compact 알고리듬에서처럼 객체의 위치를 옮기지는 않지만, 인접한 빈 공간들을 하나의 빈 공간으로 합쳐준다. 빈 공간는 연속된 영역으로 안에서는 copying collector와 유사한 방법으로 할당이 이루어진다 [9] . 하나의 빈 공간이 가득 차게 되면 다음의 빈 공간안에서 다시 할당이 이루어진다.

2.7.3 앞으로의 발전 방향

성능 비교의 주된 대상으로 삼았던 SPEC jvm98 벤치마크에 대해서 객체들의 life time에 대한 조사를 해본 결과에 따르면 대다수의 벤치마크의 경우 최근에 할당된 객체들일 수록 소거될 가능성이 높았다. 즉, 대다수의 객체들은 생성되고 얼마 후면 사라지게 되고, 일부의 살아남는 객체들의 경우는 프로그램이 종료될 때까지 소거되지 않는 경향을 가지고 있다. mark-and-sweep 알고리즘보다 일반적으로 우수하다고 알려진 copying 알고리즘만 사용하면 계속 살아남는 객체들 때문에 오히려 느려질 가능성이 있다. 따라서, 이러한 경향을 잘 이용할 수 있는 generational 쓰레기 처리 알고리즘을 같이 사용하면 쓰레기 처리의 효율을 높일 수 있다 [9] .

이때 고려를 해야할 사항은 다음과 같다.

- 어떻게 스택에서 객체를 구별해 낼 것인가?
- generational 쓰레기 처리기를 구현하기 위한 write barrier를 어떻게 효율적으로 구현할 것인가 ?

처음 문제를 해결하기 위해서는 JVM이 변환하는 메소드에 대해서는 각 시점에서 레지스터 사상을 저장하고 native 메소드에 대해서는 객체를 직접 접근하지 못하게 막아야 한다. native 메소드에 대한 문제는 JNI를 이용하면 객체를 직접 접근하지 못하므로 해결될 수 있다. 첫번째 문제를 포기함으로서 오히려 효율을 높이는 방법도 있다. Mostly copying 알고리즘에서는 스택에 대해서 보수적인(conservative) 가정을 함으로써 위와 같은 복잡한 처리 없이 copying 방법의 흉내를 내고 있다 [10] .

두번째 문제에 대해서는 이미 많은 연구가 되어 있다. 소프트웨어적인 방법에서는 write barrier를 각 쓰기(write) 명령에 대해 2개의 RISC instruction으로 줄이는 방법이 소개되어 있다 [9] . 하드웨어의 도움을 받는 방법은 old generation에 대해 쓰기 방지를 함으로써 이 곳에 write를 하는 경우에만 trap을 발생시켜 처리하는 방법이 있다. 소프트웨어적인 방법은 실제 수행되는 쓰기 명령에 비례해 부담이 발생하고, 하드웨어적인 방법은 old generation에 대해 수행되는 쓰기 명령에 비례한다. trap 명령으로 인한 부담이 상당히 크지만, old generation에 속하는 object들에 쓰는 명령이 그다지 실행되지 않는다면 오히려 하드웨어적인 방법이 더 부담이 적을 수도 있다.

제 3 절 실험결과

벤치마크	LaTTe			JDK 1.1.6		비율(SUN/LaTTe)
	전체	G.C.	TR	전체	G.C.	전체
_201_compress	69.84	2.27	0.09	106.00	0.80	1.52
_202_jess	42.44	3.26	4.69	101.10	14.52	2.38
_209_db	74.05	2.34	2.27	320.31	11.63	4.33
_213_javac	69.65	6.00	12.17	250.61	107.56	3.60
_222_mpegaudio	50.25	3.47	0.03	82.00	0.07	1.63
_227_mtrt	62.22	2.77	4.60	117.26	37.78	1.88
_228_jack	55.80	4.38	2.16	119.61	5.94	2.14
GEOMEAN						2.33

표 1: LaTTe와 SUN JDK 1.1.6의 실행시간 비교

표 1는 LaTTe와 SUN JDK 1.1.6을 SPEC jvm98 벤치마크들에 대하여 실험한 결과이다 [11] . 전체는 프로그램이 수행하는데 걸리는 총 시간을 나타내며 TR은 JIT 컴파일러에서 소요된 시간이며, GC는 쓰레기 처리에 소요된 시간이다. 실험 환경은 UltraSPARC-IIi 270MHz 프로세서를 장착하고 256 MB 메모리를 가지고 있는 Ultra-5 워크스테이션에서 이루어졌다. 각 실행시간을 비교해볼때 LaTTe가 우수한 성능을 가지고 있음을 보여준다. 이는 LaTTe에 있는 JIT 컴파일러가 효율적인 코드를 만들어 주었으며 변환하는데 사용되는 시간도 전체 시간에 비해 적은 비율을 차지하는 것을 알려준다. 또한 쓰레기 처리 시간 역시 SUN JDK 1.1.6보다 빠르게 처리함으로서 전체 시간이 많이 단축되었음을 볼 수 있다.

제 4 절 결론

LaTTe 자바 가상 머신은 JIT 컴파일러에 기반한 자바 가상 머신이다. LaTTe 자바 가상 머신이 가지고 있는 JIT 컴파일러는 빠르면서도 효율적인 레지스터 할당기를 사용하고 중복 코드 제거, 죽은 코드 죽이기나 메소드 인라이닝과 같은 최적화 기법을 통해 빠른 속도로 수행가능한 코드를 만들어 낸다. 또한, LaTTe 자바 가상 머신은 JIT 컴파일러 외에도 자바 프로그램의 효율적인 수행을 위해, 최소화된 부담으로 동작하는 모니터 락과 예외처리기를 가지고 있으며 프로그램의 수행에 부담이 되지 않을 정도로 빠른 쓰레기 처리기를 가지고 있다.

참고 서적

[1] K. Arnold and J.Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

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

[3] M. P. Plezbert and R. K. Cytron. Does Juse in Time = Better Late than Never? In *The 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[4] K. Ebcioğlu, E. Altman, and E. Hokenek. A JAVA ILP Machine Based on Fast Dynamic Compilation. In *MASCOTS '97 - International Workshop on Security and Efficiency Aspects of Java*, 1997.

[5] R. Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

[6] 박성배. 자바 just-in-time 컴파일 환경에서 코드 최적화 기법에 대한 연구. 석사학위 논문, 서울대학교, Feb 1998.

[7] U. Hölzle. *Adaptive Optimization For SELF: Reconciling High Performance With Exploratory Programming*. PhD thesis, Stanford University, Aug. 1994.

[8] B.-S. Yang, J. Lee, J. Park, S.-M. Moon, and K. Ebcioğlu. Lightweight Monitor in Java Virtual Machine. In *the 3rd Workshop on Interaction between Compilers and Computer Architectures*, Oct 1998.

[9] R.Lins R.Jones. *Garbage Collection Algorithms for Automatic Dynamic Memory Management*. JOHN WILEY & SONS, 1996.

[10] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988. Also in Lisp Pointers 1, 6 (April–June 1988), pp.2–12.

[11] SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.