実行時コンパイラによる Java 仮想マシンの分散オブジェクト対応 *

首藤 一幸　村岡 洋一
{shudoh,muraoka}@muraoka.info.waseda.ac.jp
早稲田大学 理工学部

1　はじめに

Java 仮想マシン (JVM) は、実行時に Java バイトコードをプロセッサのネイティブコードに変換する Just In Time コンパイラ (JIT) を持つことが一般的である。通常、JIT にはバイトコードの高速実行しか期待されていない。JIT は JVM の仕様通りの動作をするネイティブコードを生成する必要がある。

生成されたネイティブコードの動作がすなわち JVM の動作なので、JIT がバイトコードの解釈を決めていると言える。JIT が生成するコードを変更できれば、JVM によるバイトコードの解釈を変えられる。

我々は JIT のこの性質を利用して分散透明な分散オブジェクトシステム JITDO(Just In Time Distributed Object) を研究、開発している。JITDO は、通常の JVM ではローカルな操作となるバイトコード命令を、ネットワークの先の遠隔オブジェクトに対する操作が可能なネイティブコードにコンパイルする。操作対象のオブジェクトが遠隔オブジェクトであった場合、その操作は遠隔の JVM にリダイレクトされる。

プログラマは、遠隔オブジェクトの生成、遠隔メソッド呼び出しを、ローカルオブジェクトに対する通常の文法で行える。また、既存のシステム [3][5] では行えなかったフィールドアクセスも行える。つまり、遠隔オブジェクトをローカルオブジェクトとほぼ同様に扱える。

2　shuJIT – x86 用 JIT

JITDO は、我々が開発した shuJIT[2] を元に開発されている。shuJIT は Sun の JVM(JDK,JRE)、Linux および FreeBSD、Intel x86 プロセッサ用の JIT コンパイラである。

JIT はプログラムの実行中にコンパイルを行うのでコンパイル時間の短さも重要である。shuJIT は中間言語への変換を行わない。3 パスのコンパイルでほぼ直接バイトコードをネイティブコードに変換する。

*Adaptation of Java Virtual Machine
for Distributed Object with Just In Time Compiler
Kazuyuki SHUDO, Yoichi MURAOKA
School of Science and Engineering, Waseda Univ.

中間言語を利用せずにプロセッサのレジスタを有効に活用するのは難しい。shuJIT と同様にほぼ直接コード生成を行う JIT TYA[4] では複数のバイトコード命令をまたいで利用できているレジスタはひとつだけである。shuJIT では複数のレジスタを利用するため、stack states という考え方を導入した (図 1)。コード生成を高速に行うため、あらかじめ各バイトコード命令とそれぞれの stack states に応じたネイティブコードが用意してある。

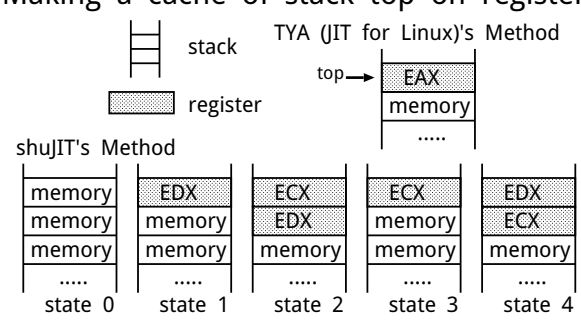

図 1: stack states

CaffeineMark 3.0[1] で、バイトコードインタプリタ、TYA、shuJIT の性能を比較した。環境は Pentium with MMX tech./233MHz、Linux 2.0.36、Linux 用 JDK 1.1.7v1a である。TYA と同等以上の項目が多いが、メソッド呼び出し性能が String、Method にあらわれている。

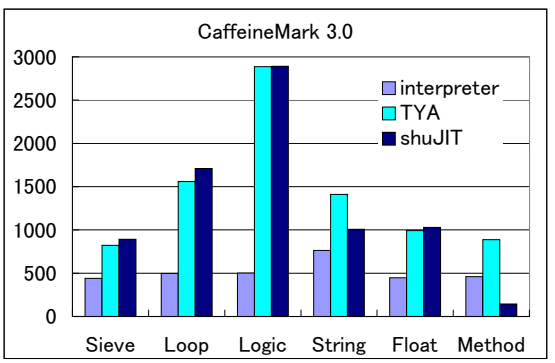

図 2: shuJIT の性能評価

3　構造

既存システムでは、遠隔オブジェクトの代理オブジェクトがローカルに存在し、遠隔オブジェクトと

signature(セレクタ) が同じメソッドを持ち、メソッド呼び出しを遠隔オブジェクトに中継する。JITDO では中継されるのはメソッド呼び出しではなくバイトコード命令である。メソッド呼び出し、フィールドアクセスなどオブジェクトに対する操作を行うバイトコード命令が、JITDO が生成したネイティブコードによって遠隔オブジェクトに中継される。

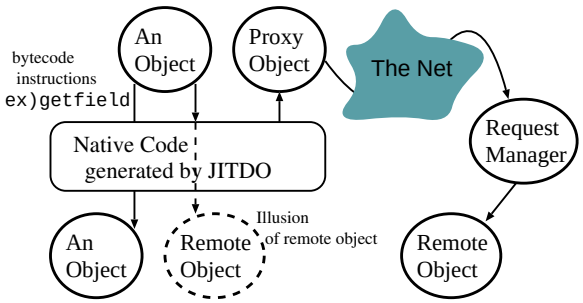

図 3: オブジェクトに対する操作のリダイレクト

JITDO は、操作対象オブジェクトがローカルオブジェクトなのか代理オブジェクトなのかを判断して挙動を変えるネイティブコードを生成する。次の各バイトコード命令について、通常の JIT とは異なるコードを生成する必要がある。

- オブジェクトの生成
  - 配列以外 – `new`
  - 配列 * – `newarray`, `anewarray`
- アクセス
  - フィールド – `putfield`, `getfield`
  - 配列 * – `[ilfdabcsa]aload`, `[ilfdabcsa]astore`
- メソッド呼び出し – `invoke{virtual,special,interface}`
- 型チェック – `checkcast`, `instanceof`

遠隔オブジェクトがさもローカルに存在するかのように見せるため、オブジェクトの型をチェックする命令の挙動も変える必要がある。

現在の設計では配列 * は遠隔オブジェクトにはせず、必ずローカルオブジェクトとしている。配列を遠隔オブジェクトとしたい局面もあるので、妥当な制御法を考えている。

4 プログラミングインタフェース

通常の Java 言語に加えて唯一必要なのは、オブジェクトの生成先 JVM を指定する手段である。

```
VMAddress addr =
    new VMAddress("foo.bar.com", 10000);
JitdoController.setInstantiationVM(addr);
```

その時点で制御を握っているスレッドに生成先 JVM が対応付けられる。以後、このスレッドからのオブジェクト生成はここで指定した JVM 上で行われる。

5 現状および他手法との比較

現在実装を進めていて、ごく簡単なプログラムが動いている段階である。

JITDO は既存システムより分散透明である。遠隔オブジェクトの生成を通常の Java 言語の文法で行え、フィールドアクセスが可能で、遠隔参照と実際の遠隔オブジェクトの型の不一致がない。

反面、ローカルオブジェクトの操作が遅くなることが予想される。操作対象のオブジェクトがローカルか遠隔の判断がオブジェクトへの操作ごとに動的に為されるためである。

オブジェクトの位置を実行時に判断するから実行速度が低下する。現在のプログラミングインタフェースではオブジェクト生成先 JVM は Java プログラム中で指定されるので、Java プログラムをコンパイルする際に静的に前処理を行い、遠隔オブジェクトの操作を適当なコードに展開してしまう手法も考えられる。しかしこの手法にはバイトコードの量が爆発しかねないという問題がある。メソッドの引数それぞれについて遠隔、ローカル双方を考慮し、組合せごとの展開結果を用意する必要がある。Java のコンパイル単位であるクラスについてこの組合せ数は爆発し得る。インライン展開で軽減できるものの本質的な解決にはならない。

オブジェクト生成先 JVM を Java プログラムの外から動的に変更、制御することを計画している。JVM ごとのメモリ使用量を調整するといった応用が考えられる。この場合オブジェクトの存在場所の動的な判断が必要であり、静的な前処理では対応できない。

6 まとめ

我々が設計、実装している分散オブジェクトシステム JITDO について、ベースとしている shuJIT のコード生成手法と性能、JITDO の構造とプログラミングインタフェースを述べ、既存システムや他の手法との比較を行った。

今後は実装、性能評価を進めていく。

参考文献

[1] CaffeineMark 3.0. http://www.pendragon-software.com/pendragon/cm3/.

[2] shuJIT: JIT compiler for Sun JVM/x86. http://www.shudo.net/jit/.

[3] Sun Microsystems. *Java^{TM} Remote Method Invocation Specification*, 1997.

[4] TYA Archive. http://tya.home.ml.org/.

[5] ObjectSpace: Products: Voyager. http://www.objectspace.com/products/Voyager/.