プログラマに単一マシンビューを提供する分散オブジェクトシステムの実現

首藤一幸[†] 根山亮[††] 村岡洋一[†]

分散オブジェクトシステムの目標のひとつは，オブジェクト指向言語のプログラミングモデルを分散処理に適用できるようにし，プログラマにネットワークを意識させないようにすることである．我々はネットワーク透過な分散オブジェクトシステムを構築した．本システムでは，ローカルオブジェクトとネットワークの先にあるオブジェクトをまったく同様に扱うことができ，プログラミングの際はマシン群をほぼ単一マシンとみなすことができる．この，ネットワークおよびプログラミングモデルの透過性は，Javaバイトコードの実行時コンパイラに，オブジェクトの遠隔操作が可能なネイティブコードを生成させることで実現された．

Design and Implementation of Distributed Object System presenting a Single Machine View to Programmers

Kazuyuki Shudo,[†] Ryo Neyama[†] and Yoichi Muraoka[†]

One of important goals of distributed object system is allowing programmers object oriented programming for network of machines and being not aware of the existence of network. We've designed and developed a network transparent distributed object system which satisfy the goal well. The system lets programmers handle remote objects in the same way as local ones and regard network of machines as a machine. Such transparency is achieved by getting Java bytecode runtime compiler generating native code can deal with remote objects.

1. はじめに

分散オブジェクトシステムは分散システムをオブジェクト指向プログラミング言語(OOPL)で記述するための道具である．その出現理由，重要な目的のひとつは，ネットワーク経由の情報交換についてプログラマの負担を低減することである．プログラマは OOPL のプログラミングモデルをネットワーク接続されたマシン群に対して適用でき，通信を明に記述する際の，プロトコル設計，デバッグなど煩雑な作業はかなり軽減される．

負担が軽減されるとはいえ，依然としてプログラマがネットワークの存在を意識しなければならない局面は残されている．例えば，従来システム [1)2)3)4)5)] には，別マシン上にあるオブジェクト(遠隔オブジェクト)を同一マシン上のオブジェクト(ローカルオブジェクト)と同様には扱えないという制約があったり，遠隔参照の対象とできるクラスの種類に制限があるなど，ネットワークに関する非透過性が見られる．さらにシステムによっては，遠隔から呼び出すことのできる関数，メソッドについて特殊な宣言が必要であったり，プログラムにプリプロセスが必要であるなど，プログラマが考えねばならないことが多く残されている．

我々は，これら非透過性の多くを解消した，MetaVM と呼ばれる Java 言語用の分散オブジェクトシステムを設計，実装した．本システムを利用することで，プログラマは遠隔オブジェクトをローカルオブジェクトとまったく同様に扱うことができる(ネットワーク透過)．これによって，マシン群をほぼ単一マシンとみなしてプログラミングすることが可能となった(プログラミングインタフェースが透過)．

既存の分散オブジェクトシステムではプリプロセッサまたはスタブクラスジェネレータが重要な役割を担っている．これによって生成されるスタブと呼ばれるクラスが，遠隔操作の対象としたいクラスとセレクタ(Java の用語では signature)が同じ関数，メソッド群を持ち，関数，メソッド呼び出し要求を遠隔に中

† 早稲田大学理工学部
School of Science and Engineering, Waseda University
†† 日本アイ・ビー・エム株式会社東京基礎研究所
IBM Research, Tokyo Research Labolatory, IBM Japan Ltd.

継する．プログラマは，ローカルな呼び出しと同じ記述で遠隔呼び出しを行える．記述が同じであっても，OOPL の機能である動的結合によって，遠隔呼び出しとローカルな呼び出しが適切に選択される．つまり，操作対象がスタブクラスのオブジェクト (スタブオブジェクト) であった場合に，呼び出し要求が遠隔に中継される．

遠隔呼び出しをローカルなそれと同様に記述できるという透過なプログラミングインタフェースを実現した一方，この手法にはいくつかの限界もある．まず，動的結合を利用しているため，可能な遠隔操作は関数，メソッド呼び出しだけである．また，遠隔参照を表すスタブオブジェクトの型が実際に操作対象となる遠隔オブジェクトの型とは異なるため，遠隔参照を扱う際は遠隔参照であることを意識して記述する必要がある．さらに，スタブを動的に生成できないシステム [1)2)4)] では，プログラマがあらかじめスタブジェネレータを使って生成しておく必要がある．

我々は，こういった特殊な準備や，遠隔オブジェクトを扱う際のローカルオブジェクトにはない制限を取り除くために，実行時コンパイルを応用する手法を開発，実証した．まず，研究素材としての扱いやすさを重視して実行時コンパイラを開発し，それを基盤に，プログラマおよびユーザプログラムに対して遠隔オブジェクトをローカルオブジェクトであるかのように見せるコンパイラを開発した．

Java 仮想マシン [6)] は，Java バイトコードをプロセッサネイティブなコードに変換する，Just In Time コンパイラと呼ばれる実行時コンパイラを持つことが一般的である．実際にプロセッサが実行するのはバイトコードではなく，実行時コンパイラによって生成されたネイティブコードである．つまり，Java 仮想マシンによるバイトコードの解釈は実行時コンパイラが決めていると言える．通常，実行時コンパイラには Java 仮想マシンの仕様通りに高速に動作するコードの生成だけが期待されていて，仕様を逸脱したコードが生成されるべきではない．しかし，実行時コンパイラが生成するコードを積極的にカスタマイズ，変更することが有効な応用がある [7)]．我々は，実行時コンパイラのこの権限，能力を，透過性の高い分散オブジェクトシステムを構成するために利用した．

本稿は，実行時コンパイラを利用した分散オブジェクトシステムの構成手法を提案し，構成したシステムの概要を述べ，その基本性能の評価結果を報告するものである．また，素材として開発した実行時コンパイラについて，そのコード生成手法，最適化手法を報告する．

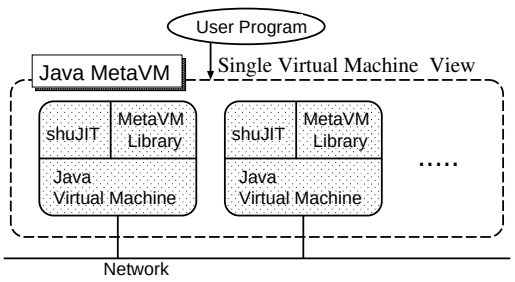


図 1 MetaVM の構成
Fig. 1 Structure of MetaVM

2. MetaVM の概要

ネットワーク接続されたマシン群に対する単一マシンビューをプログラマに提供することを目標に，我々は Java 言語用分散オブジェクトシステム MetaVM を開発した．プログラマはオブジェクトを生成する場所を指定でき，遠隔で生成されたオブジェクトをローカルオブジェクトとまったく同様に扱うことができる．

図 1に MetaVM の構成を示す．MetaVM はネットワーク上に分散した複数の Java 仮想マシンから構成される．それぞれの Java 仮想マシンと共に動作する MetaVM ライブラリと shuJIT[8)] が，複数の Java 仮想マシンに対する単一マシンビューをプログラマおよびユーザプログラムのバイトコードに提供する．

MetaVM ライブラリ Java で実装されたライブラリである．通信やエクスポート表の管理などシステムの機能のほとんどを受け持つ．詳細を第 5章で述べる．

shuJIT Java バイトコードの実行時コンパイラである．遠隔オブジェクトを扱うことのできるネイティブコードを生成する．遠隔操作のために，適宜 MetaVM ライブラリを利用する．MetaVM の一部としてだけではなく，通常の実行時コンパイラとしても利用できる．MetaVM のためのコード生成手法を第 4章で，通常の実行時コンパイラとしての手法を第 6章で述べる．

Java 仮想マシン Sun Microsystems 社(Sun 社)のもの，つまり Java Development Kit (JDK) や Java Runtime Environment (JRE) を利用する．

2.1 プログラミングインタフェース

プログラマが，Java 言語以外に，遠隔オブジェクトを扱うために知っておく必要があるのは，オブジェクト生成先となる Java 仮想マシンを指定する方法だけである．ネットワーク上の位置を指定する．以下に，

遠隔にオブジェクトを生成するコード例を示す．

```
VMAddress addr = new VMAddress("hostname");
MetaVM.instantiationVM(addr);
  // オブジェクト生成先の指定
Object obj = new Object();
  // オブジェクトの遠隔生成
MetaVM.instantiationVM();
  // 今後の生成はローカルに，という指定
```

MetaVMでは，オブジェクト生成先の指定以外は，オブジェクトの生成も含めて，すべて通常のJava言語の構文，機能を用いることができる．上記のように生成した遠隔オブジェクト`obj`は，以後，ローカルオブジェクトとまったく同様に扱える．

以下，プログラミングインタフェースを説明する．

2.1.1 準　　備

クラス定義の前に次の宣言を記述しておく．

```
import NET.shudo.MetaVM.*;
```

2.1.2 オブジェクト生成先の指定

Java仮想マシンのネットワーク上の位置はIPアドレスとTCPポート番号の組で特定され，`VMAddress`クラスのオブジェクトで表される．`VMAddress`のオブジェクトをメソッド`MetaVM.instantiationVM()`に与えることで，オブジェクト生成先仮想マシンを指定する．引数を与えないと，生成先としてローカルのJava仮想マシンを指定したことになる．指定されたVMアドレスはその時点で制御を握っているスレッドに対応付けられ，その後のそのスレッドからのオブジェクト生成は指定された仮想マシン上で行われる．

2.1.3 そ の 他

分散システムの記述に有用なクラス，メソッドをいくつか提供している．

メソッド `MetaVM.address(Object obj)`　引数として与えたオブジェクトが存在しているJava仮想マシンの位置を`VMAddress`型で返す．

インタフェース `ByValue`　これを実装したクラスのオブジェクトは，遠隔呼び出しの引数となった場合，遠隔に参照ではなくコピーが渡される．

3. 透 過 性

MetaVMは，遠隔オブジェクトをローカルオブジェクトと同様に扱えるという高いネットワーク透過性を達成し，ネットワーク上のマシン群をほぼ単一マシンとみなせるという，プログラミングインタフェースの高い透過性を達成している．本章では，我々が何をもって透過であると主張するのかを，既存のC++言語，Java言語用分散オブジェクトシステム[1)2)3)4)5)]と比較して述べる．また，いくつか残っている，単一マシンとの違いにも言及する．

3.1 MetaVMの提供する透過性

3.1.1 特殊な準備，プリプロセス

MetaVMでは，ユーザプログラムは通常のJavaコンパイラでコンパイルしておくだけでよい．これは，一部の既存システムでは実現されていた．

多くの既存システムでは，遠隔から呼び出したい，エクスポートする関数，メソッドのセレクタ(Javaの用語ではsignature)をあらかじめ記述，宣言しておく必要がある．CORBA[1)]ではInterface Definition Language (IDL)で，RMI[4)]ではJava言語の`interface`として記述しておく．また，RMIではコンパイル後のクラスをスタブコンパイラ`rmic`で処理する必要がある．HORB[9)]でも`horbc`による前処理が必要となる．CORBAでもプリプロセスが必要である．

Voyager[3)]，RyORB[5)]では上記のような特殊な前処理は不要であり，スタブなど必要なクラスはプログラム実行時に動的に生成，コンパイル，ロードされる．この点ではMetaVMと同等の透過性を達成している．

3.1.2 遠隔参照の型

MetaVMでは，プログラマやユーザプログラムのバイトコードは，遠隔参照を実際の遠隔オブジェクトとまったく同じ型として扱うことができる．これは既存システムでは実現できていない．

RMIでは，あらかじめ記述しておいた`interface`の型で遠隔参照を扱う．Voyagerでは`interface`ジェネレータ`igen`で生成しておいた`interface`の型で扱う．HORBでは，RMIと同様に`interface`を記述しておくか，もしくは，実際の遠隔オブジェクトとは異なる“クラス名_Proxy”型で扱うかを選択できる．いずれにせよ，遠隔オブジェクトであることを意識して，ローカルオブジェクトとは異なる扱いをする必要がある．

RyORBは他の既存システムよりも透過的である．遠隔参照を表すスタブクラスは遠隔オブジェクトのクラスのサブクラスとなるので，遠隔オブジェクトをそれ自身の型で扱うことが可能となっている．反面，サブクラスの作成を許さない(`final`)クラスへの遠隔参照に制約が生じる．その`final`クラスが何かしらの`interface`を実装(`implements`)していれば，その`interface`の型として遠隔参照できる．しかし何も`interface`を実装していない`final`クラスのオブジェクトは遠隔参照できない．プログラミングインタフェースの透過性を重視した設計の代償である．

3.1.3 配列への遠隔参照

MetaVM では配列も遠隔参照できる．Java 言語の配列はオブジェクトであるにも関わらず，既存の Java 用システムでは配列を遠隔参照できない．

3.1.4 call by reference

Java では，オブジェクトへの参照をメソッド呼び出しの引数とした場合，callee に渡るのは参照であり，call by reference が基本である．MetaVM ではプログラマの指示（第 2.1節）がない限り，遠隔呼び出しであってもこのセマンティクスが守られる．ただし一部のクラスについてはオブジェクトのコピーが遠隔に渡される．この例外については第 3.2.3項で述べる．

既存の Java 用システムでは，遠隔呼び出しの基本は call by value である．ローカルな呼び出しでは参照渡しなので，ローカル呼び出しと遠隔呼び出しで結果や副作用に違いが出る場合がある．

引数オブジェクトのクラスに特別な指定をしておくことで，引数オブジェクトの遠隔へのコピーではなく，参照を渡せるシステムもある．しかし，配列オブジェクトを遠隔参照できないので（第 3.1.3項），配列は参照として渡すことはできない．また，他人が設計して自分は変更できないクラスを参照渡しできない，という限界もある．

3.1.5 配列，フィールドへのアクセス

MetaVM では遠隔オブジェクトのフィールド（メンバ変数）や，遠隔の配列の要素に，通常の Java 言語の構文でアクセスできる．C++，Java 用の既存システムではできない．

3.2 非透過性 — 単一マシンとの差異

MetaVM の目標はマシン群に対する単一マシンビューをプログラマに提供することだが，単一 Java 仮想マシンとの違いがいくつか残っている．その相違点を述べる．

3.2.1 遍在するクラスオブジェクト

クラスの定義は各 Java 仮想マシンごとにロードされるので，名前と定義が同じクラスのクラスオブジェクト（`java.lang.Class` クラスのオブジェクト）が Java 仮想マシンごとに存在してしまう．するとクラスオブジェクトに属する変数，つまりクラス変数の値が，どの Java 仮想マシン上で参照するかによって異なってしまう．

3.2.2 オブジェクトへのネイティブメソッドからのアクセス

Java 言語では C 言語または C++で記述したメソッド（ネイティブメソッド）を呼び出すことができる．Java の標準ライブラリもいくつかのネイティブメソッドを含んでいる．遠隔参照への操作がバイトコードから行われる限りは，実行時コンパイラが遠隔操作のためのコードを生成するので問題は起きない．しかし，遠隔参照を考慮しないで書かれたネイティブメソッドは遠隔参照を扱えない．標準ライブラリ中のものも含めて，世の中のネイティブメソッドは遠隔参照の存在を想定して書かれてはいないからである．

ネイティブメソッドが遠隔参照をアクセスした場合，結果は予測できない．例えば，Java の標準 API の一部である reflection API は，その内部でネイティブメソッドを多用している．したがって，reflection API が提供する機能のほとんどは遠隔参照に対しては利用してはいけない．

3.2.3 call by reference の例外

MetaVM での遠隔呼び出しでは，ローカル呼び出しと同様に，引数として callee に渡るのは基本的に参照である．フィールド，配列に対する遠隔アクセスの際も，オブジェクトのコピー，つまり値ではなく，参照がネットワーク経由で渡される．ところが例外的に，いくつかのクラスについては，システム実装の都合，または性能上の理由から，コピーが渡されるようにしてある．次に挙げるクラスである．

- 実装の都合
  - **Throwable** 遠隔操作で発生した例外を caller に返すため．ネットワークの障害が原因で例外が発生した場合，その例外は遠隔参照できないだろう．
  - **InetAddress** IP アドレスを表すクラス．遠隔参照が指す先を表すためにこのクラスを用いているので，InetAddress 自体は遠隔参照にできない．
  - **Number,Boolean,Character** 整数，浮動小数点数などの基本型の wrapper クラス．遠隔呼び出し時に基本型の値を wrapper クラスで包んで遠隔に渡すため，参照渡しにできない．
- 性能上の理由
  - **String** Java プログラムで頻繁に使われるため．

3.2.4 プログラムの終了条件

Java 仮想マシンのソフトウェア実装では，daemon 属性が付けられていないスレッドが存在しなくなった時点で，Java 仮想マシン自体が実行を終了する．MetaVM で遠隔にスレッドを生成した場合，遠隔にスレッドが残っていても，ローカルに非 daemon スレッドがなくなった時点でローカルの Java 仮想マシンは実行を終了してしまう．これによって，単一 Java

仮想マシン上で実行する場合とはプログラムの終了条件が変わる場合がある．

4. 実行時コンパイラによる遠隔オブジェクト対応ネイティブコードの生成

MetaVM は，プログラマに対してだけではなく，ユーザプログラムがコンパイルされて生成された Java バイトコードに対しても，遠隔オブジェクトをローカルオブジェクトと同様に見せる．これは，バイトコードの実行時コンパイラ shuJIT[8] が，遠隔参照を扱えるネイティブコードを生成することで実現されている．

shuJIT は，オブジェクトに対する操作を行うバイトコード命令に対して，操作対象が遠隔参照だった場合は遠隔操作を行う，というネイティブコードを生成する．遠隔操作は MetaVM ライブラリの呼び出しで実現されている．ここで生成されるネイティブコードの処理内容は，次に示すものに相当する．

```
if ((obj instanceof Proxy) && remote_flag)
```
遠隔参照 obj を MetaVM ライブラリに処理させる
```
else
```
ローカル参照 obj に対して通常の処理を行う

ここで obj は操作対象への参照，Proxy は遠隔参照を表す MetaVM ライブラリ中のクラス，つまりスタブクラスである．remote_flag は，Proxy クラスのオブジェクトで表された遠隔参照を，ユーザプログラムに対して Proxy オブジェクトとして生のまま見せるか，それとも遠隔参照として見せるか，のフラグであり，各スレッドに属する．つまり，バイトコード命令の操作対象が遠隔オブジェクトであり，かつフラグが立っているならば遠隔操作を行う，というネイティブコードが生成される．

オブジェクトを生成するバイトコード命令に対しては，次に示す処理に相当するネイティブコードを生成する．

```
if (remote_flag && !(clazz は値渡しのクラス))
```
遠隔にオブジェクトを生成する
```
else
```
ローカルにオブジェクトを生成する．

ここで clazz はオブジェクトを生成するクラス，値渡しのクラスとは，第 3.2.3項で挙げた，ネットワーク越しにコピーが渡されるクラスである．

オブジェクトに対する操作では，操作対象が遠隔オブジェクトかどうかの判断がプログラムの実行時に動的になされる．ローカルオブジェクトへの操作時も，この判断は行われ，オーバヘッドとなる．MetaVM に対応しない通常の実行時コンパイラとしてコンパイルした非分散版 shuJIT と比較すると，MetaVM は，遠隔参照をまったく扱わない完全にローカルに実行されるプログラムの実行性能が低くなっている．変化量の評価結果は第 7.2節で示す．また，動的に判定するこの手法の是非を，第 8.1節で他手法と比較して論じる．

また，非分散版 shuJIT と比較して，生成されるネイティブコードの量が多い．コード量の評価結果を第 7.3節で示す．

MetaVM 対応の shuJIT が，非分散版 shuJIT とは異なるネイティブコードを生成するバイトコード命令を挙げる．ここで [...] は [] 中のどれか一文字，{...} は {} 中の “,” で区切られたどれか一語に相当する．

- オブジェクトの生成
  - 配列以外のクラス
    `new`
  - 配列
    `newarray`, `anewarray`, `multianewarray`
- アクセス
  - フィールド
    `getfield`, `putfield`
  - 配列の要素
    `[ailfdbcs]aload`, `[ailfdbcs]astore`
- 配列長の取得
  `arraylength`
- メソッド呼び出し
  `invoke{virtual, special, interface}`
- 型チェック
  `checkcast`, `instanceof`
- モニタの扱い
  `monitorenter`, `monitorexit`

遠隔参照に対応したネイティブコードを生成するためには，200 強あるバイトコード命令のうち，たかだか，ここに挙げた 30 命令に対して生成するコードを変えればよい．さらに，複数の命令に対して共通に生成されるネイティブコード列も多い．

5. MetaVM ライブラリ

MetaVM ライブラリは，遠隔操作，つまり，遠隔オブジェクトに対するフィールドアクセス，配列アクセス，メソッド呼び出しなどを行うための，Java で記述されたライブラリである．ネットワーク越しの要求を処理する MetaVM サーバ，shuJIT が生成するネイティブコードから呼び出されるメソッド群，そして，ユーザプログラムから利用されるいくつかのクラ

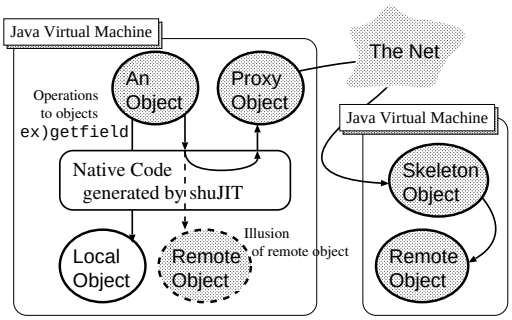


図 2 遠隔操作の中継
Fig. 2 Redirection of operations to the remote object

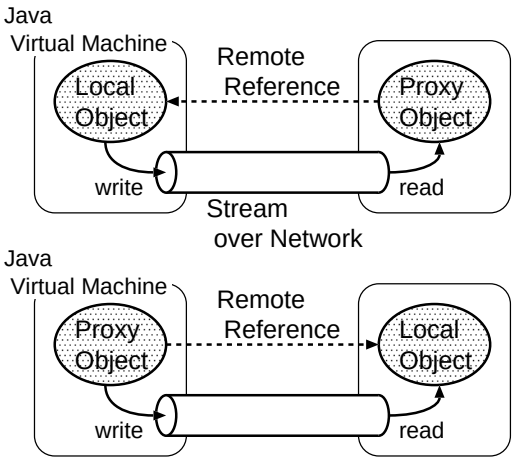


図 3 ネットワークを経由した参照の受渡し
Fig. 3 Reference passing via the network

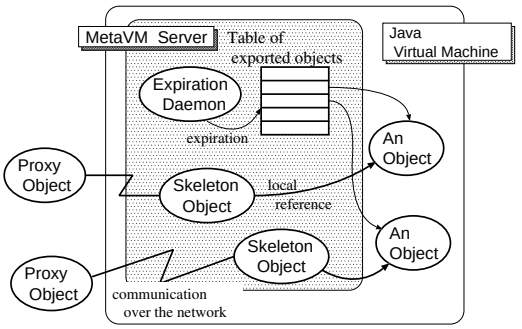


図 4 MetaVM サーバ
Fig. 4 MetaVM server

スやメソッド (第 2.1節) を含む .

5.1 Proxy と Skeleton

多くの既存システムでは , スタブクラスを , そのインスタンスを遠隔参照するクラスごとに生成する . しかし MetaVM ではそれとは異なり , Proxy クラスがあらゆるクラスのスタブとなっている . 遠隔参照は Proxy クラスのオブジェクト (Proxy オブジェクト) への参照で表現される . shuJIT が生成するネイティブコードによって , ユーザプログラムのバイトコードからは , Proxy オブジェクトは遠隔オブジェクトそのものに見える . 遠隔の仮想マシン上には , Proxy オブジェクトに対応する , Skeleton クラスのオブジェクト (Skeleton オブジェクト) があり , Proxy オブジェクトからのネットワーク越しの要求を本当の操作対象である遠隔オブジェクトに中継する (図 2) .

5.2 参照の受け渡し

プログラムが分散実行される際も単一マシンでの実行と同じセマンティクスを保つため , MetaVM では基本的にネットワーク越しにオブジェクトのコピーは行わない . ネットワーク越しには参照が渡される . ただし例外として , 第 3.2.3項で挙げた一部クラスのオブジェクトは , 遠隔にコピーが渡される .

コピーではなく参照を渡すためには , ローカル参照を適切に遠隔参照に変換する必要がある . 一方 , ネットワーク越しに受け取った遠隔参照がローカルオブジェクトを指していた場合 , 操作の際に Proxy , Skeleton オブジェクトを経由するオーバヘッドを回避するため , 遠隔参照を , 対応するローカル参照に変換する (図 3) .

遠隔参照のローカル参照への完全な変換は MetaVM に固有の特徴である . いくつかの既存システム [5)3)] では , 遠隔参照がローカルオブジェクトを指していた場合は , スタブオブジェクトが直接目的のメソッドを呼び出すことで通信を削減し , メソッド呼び出しの性能向上を図っているが , スタブオブジェクトへの呼び出しは避けられない . このような場合 MetaVM では完全にローカル呼び出しとなり , スタブオブジェクト (Proxy オブジェクト) のメソッド呼び出しは起きない .

5.3 MetaVM サーバ

MetaVM サーバ (図 4) は Proxy オブジェクトからの接続要求に対して Skeleton オブジェクトを生成する . Proxy オブジェクトは Skeleton オブジェクトに対して最初に , オブジェクトの生成か , ID に対応するオブジェクトの検索を要求する . ID は , あるオブジェクトに対応する , Java 仮想マシン内で固有の 32bit 整数値である . Meta サーバ内のエクスポート表には , ID をキーに , ネットワーク越しに参照され得るオブジェクトが登録されている .

MetaVM のユーザは , 分散処理に利用したいマシン上であらかじめ MetaVM サーバを動作させておく必要がある . いったんユーザプログラムの実行を開始すると , そのために起動された Java 仮想マシン内のオブジェクトに対する , ネットワーク越しの遠隔参照

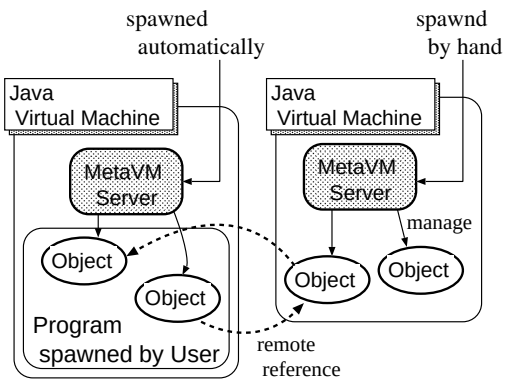


図 5 自動的に起動される MetaVM サーバ
Fig. 5 MetaVM server spawned automatically

も生じ得る．そのため，ユーザプログラムのために起動した Java 仮想マシン内にも MetaVM サーバが必要である．とはいえ，ユーザプログラムの中で明示的に MetaVM サーバを起動する必要はない．オブジェクト生成先を指定する `instantiationVM()`（第 2.1節）の最初の呼び出し時に，自動的に起動される（図 5）．

6. 実行時コンパイラ shuJIT

実行時コンパイラによる，遠隔オブジェクトに対応したコードの生成については第 4章で述べた．本章では，我々が実行時コンパイルを応用した研究の素材として開発したコンパイラ，shuJIT の概要を述べる．

6.1 概 要

shuJIT[8] は，Sun 社の Java 仮想マシンと，x86 と呼ばれる，Intel 社の 32bit アーキテクチャ IA-32 用の Java バイトコード実行時コンパイラである．Linux と FreeBSD で動作している．商用のコンパイラとは異なりソースコードが公開されているので，研究，教育目的に利用しやすい．

6.2 コード生成手法

実行効率を最優先する実行時コンパイラは，いわゆる高級言語のコンパイラの多くと同様に，中間表現を利用して，レジスタを有効に活用できるようレジスタ割り付けを行う．FJIT[10] やそれを元にした OpenJIT[7] は，GNU C コンパイラの中間表現である RTL を利用している．それに対して shuJIT は中間表現を利用しない．我々があらかじめ記述しておいた，各バイトコード命令に対応するネイティブコード（pre-assembled コード）をつなぎ合わせていくことでコード生成を行う．この単純なコード生成方式は，実行時コンパイルを応用した，MetaVM のような研究の素材として使いやすいことを最優先している．アセンブリ表現で記

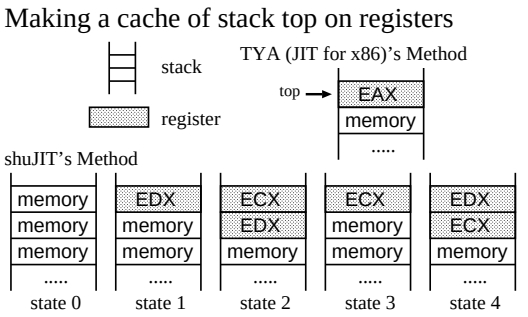


図 6 スタック状態
Fig. 6 stack states

述しておいたコードが直接に生成結果の一部となるので，生成させたいコードを直接記述しておくことができる．このように，改造を柔軟に行えるので，研究素材として扱いやすい．shuJIT はまた，異機種間のプリエンプティブな移送が可能なスレッド移送システム[11] が実行時コンパイラと両立することを実証するためにも，基盤として利用されている．

コード生成手法が単純なので，コンパイル時間が短いことが期待できる．実行時コンパイラによるコード生成はプログラムの実行中に行われるので，生成されるコードの実行効率と共に，コンパイル時間の短さも重要である．さらに，中間表現を利用する手法とは異なり，実行時コンパイラがアセンブラを持つ必要がなくなるという利点もある．

しかし，中間表現を利用せずにレジスタを有効に活用するのは難しい．例えば，shuJIT と同様に，バイトコード命令を対応するネイティブコードにほぼ直接変換するコンパイラ TYA[12] では，バイトコード命令間で値の受け渡しに利用できているレジスタはひとつだけである．shuJIT では，バイトコード命令からの直接変換と複数レジスタの利用を両立させるために，スタック状態という考え方を採用した（図 6）．Java 仮想マシンのスタックトップ付近がどのレジスタでキャッシュされているかに対応したいくつかの状態を定義する．pre-assembled コードは，バイトコード命令だけではなく各状態にも対応したものを用意し，そのコードの実行終了時点での状態も表に記録しておく．コード生成は，直前の pre-assembled の終了時点での状態，そしてバイトコード命令に対応した pre-assembled コードを連結していくことで進んでいく．

shuJIT では，レジスタを 2 つ使って Java 仮想マシンのスタックトップ付近をキャッシュするために，状態数を 5 とした．このコード生成法の問題は，バイトコード命令の種類に状態数を乗じた数の pre-assembled コー

ドを用意しておかねばならないという点である．コードはかなり使い回せるとはいえ，200 を超える命令に対し 5 状態設けたので，1000 以上の pre-assembled コードを用意しておく必要があった．

キャッシュに利用できるレジスタが 2 つとは少なく見えるかもしれない．中間表現を利用すれば，より多くのレジスタを有効に使えるだろう．しかし，IA-32 は汎用レジスタを 8 つしか持たない．shuJIT では，キャッシュに利用する 2 つ以外にも，Java の局所変数のベースアドレスをキャッシュする目的にひとつ，バイトコード命令中で雑用として 2 つのレジスタを活用している．スタックポインタ，ベースポインタをその本来の用途に使ってしまっているとはいえ，空いているレジスタはない．レジスタ数が少ないゆえに，中間表現を利用するか否かの差は小さい．shuJIT のコード生成手法は IA-32 に向いていると言える．

6.3 その他の最適化

コード生成手法以外にもいくつかの工夫を施している．

6.3.1 プロセッサのスタックの利用

Sun 社の Java 仮想マシンは，正方向に成長する仮想マシン用スタックを用意していて，Java インタプリタはこれを利用している．このスタックへの push 操作の手順は，値のメモリへの書き込み，メモリ上にある，仮想マシンのスタックポインタのインクリメント，となる．shuJIT が生成するコードは，このスタックの代わりにプロセッサのスタック機構を利用する．push 操作はプロセッサの push 命令だけで済む．ただし，IA-32 のスタックは負方向に成長するので，インタプリタが実行しているメソッドから，shuJIT が生成したコードが呼び出される場合，またその逆の場合は，引数と返り値の積み直しが必要となる．

中間表現を利用するコード生成手法では，もともとこの工夫は無用である．Java 仮想マシンはスタックマシンであるとはいえ，たいていの中間表現は仮想レジスタマシンのコードだからである．

6.3.2 ネイティブコードの自己書き換え

生成されたネイティブコードが，自分自身を書き換えることが有効な局面がある．一度きり，そのコードが初めて実行されたときに実行されれば充分な処理を 2 度目以降の実行では省くために，shuJIT は自己を書き換えるコードを生成する．例えばバイトコードの new 命令では，インスタンスが生成されるクラスに対して実行中のコードがアクセスする権限を持っているのかどうかのチェックを 1 度きりにするために，自己を書き換えるコードを生成する．また，メソッド呼び出し命令では，呼び出すメソッドを持っているクラスが初期化されているかどうかのチェックと初期化処理を 1 度きりにするために，自己書き換えコードを生成する．

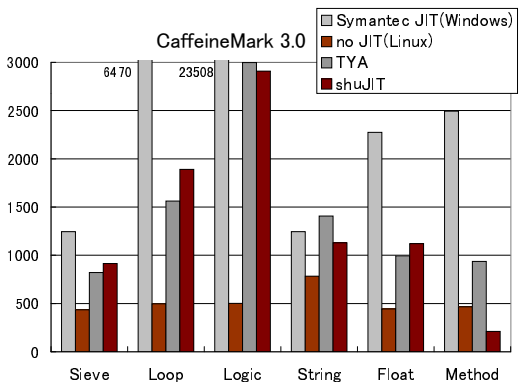


図 7 各種コンパイラの CaffeineMark 3.0 ベンチマークの結果
Fig. 7 CaffeineMark 3.0 benchmark results of runtime compilers

6.3.3 シグナルの利用

Java 言語では，null に対してメソッド呼び出しなどの許されない操作を施した場合に，例外 NullPointerException が発生する．実行時コンパイラが生成するコードにも同じ動作が要求される．操作対象が null かそうでないかの検査が必要である．ここで敢えて，検査を行うコードを生成しない．すると，Java 仮想マシン内で null を表す 0 番地をアクセスしてしまい，オペレーティングシステム（OS）のシグナル SIGSEGV が発生する．SIGSEGV を受け取ったアプリケーションは異常終了するのが通常であるが，SIGSEGV をシグナルハンドラで捕え，NullPointerException が発生したとみなせば，Java 仮想マシンはその処理を続けられる．このように null かどうかの検査を行うことで，明に検査するコードを省け，NullPointerException が発生しないたいていの場合を高速化できる．

6.4 shuJIT の性能評価

shuJIT の生成するコードの性能を，他の Java バイトコード実行時コンパイラと比較して示す．実験環境は，プロセッサが Pentium with MMX technology 233MHz，OS は特に断らない限り Linux 2.2.1，Java 実行系は JDK 1.1.7 である．

図 7は，Pendragon Software 社のベンチマークプログラム，CaffeineMark 3.0（CM3）[13] の結果である．このスコアは一定時間内にある処理繰り返せた回数なので，数値が大きい方が良い結果を表している．今回は，Java バイトコードの実行性能以外が支配的と

なるベンチマーク，つまり Graphics，Image，Dialog を除外した．

4 種の Java 実行系で実験した．左から，Symantec 社の JIT コンパイラ，Linux 用の JDK（Java インタプリタ），Linux，FreeBSD 用 JIT コンパイラである TYA[12]，shuJIT と並んでいる．Symantec 社の JIT コンパイラは Windows 用のみ提供されているので，Windows95 で実験した．それ以外は Linux 上での結果である．

shuJIT の性能は Symantec 社の製品には及ばない．TYA と比較すると，3 項目で TYA がよい結果を示し，別の 3 項目では shuJIT がよい結果を示している．結果に表れているように，現 shuJIT の性能上の問題はメソッド呼び出しのコストが高いことである．

このように，shuJIT と TYA は特性が異なるので，アプリケーションによって，どちらでよい性能が示されるかは異なる．AES[14] 候補の 15 の暗号方式の Java optimized implementation について，1 ブロック暗号化のスループットを計測したところ，8 つのアルゴリズムは TYA でよりよい結果が示され，残り 7 つでは shuJIT でよりよい結果が示された．

7. MetaVM の性能評価

性能評価の結果を示す．以下すべての実験で，実験環境は Pentium with MMX technology 233MHz，Linux 2.2.1，JDK 1.1.7 である．また，すべての実験で，MetaVM 以外では JIT コンパイラ TYA を用いている．shuJIT を必要とする MetaVM は別として，Linux 上でのこれらのベンチマークでは，TYA を使った場合に最も良い結果が示されたからである．

MetaVM は 1999 年 3 月 6 日版，RyORB は 1999 年 3 月 8 日版，Voyager は 2.0.2，HORB は 1.3 beta4，RMI は JDK 1.1.7 に付属のものを使用した．

7.1 遠 隔 操 作

MetaVM を用いた遠隔操作の性能を，他の Java 用分散オブジェクトシステムと比較する．ひとつの評価プログラムについて，1 台のマシン上での実験と，2 台での実際にネットワークを介しての実験の双方を行った．1 台での実験では，ネットワークを介することによる遅延の影響を除外して，ソフトウェアによるオーバヘッドのみを測定できる．反面，本来は並列性がありオーバラップされるはずの，操作を行う側と施される側の処理が 1 台で行われることになる．仮に，処理をオーバラップさせるという実装上の工夫がなされていたとしても，それは結果に表れない．

2 台での実験には，上述のマシンに加えて，PentiumII 350 MHz，Linux 2.2.9，JDK 1.1.7 という構成のマシンを用い，上述のマシンからこのマシンに対して遠隔操作を行った．ネットワークは 10Mbps Ethernet であり，2 台のマシンは 3 台のリピータハブを介して接続された．

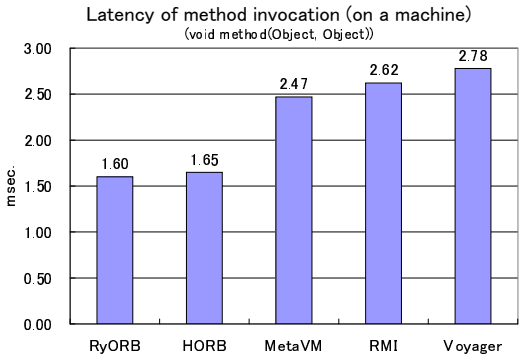


図 8 遠隔メソッド呼び出しの遅延 (1 台上)
Fig. 8 Latency of remote method invocation(on a machine)

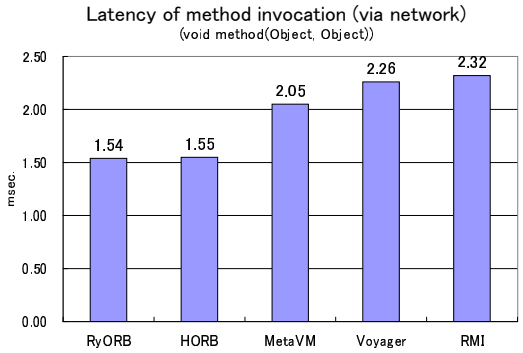


図 9 遠隔メソッド呼び出しの遅延 (ネットワーク経由)
Fig. 9 Latency of remote method invocation(via network)

7.1.1 メソッド呼び出し

図 8，9は，参照 2 つを引数 p にとり，返り値を返さないメソッド（`void method(Object obj1, Object obj2)`）を遠隔呼び出しした場合の，呼び出し 1 回あたりの遅延を示している．図 8がマシン 1 台上での結果，図 9がネットワークを介した 2 台での結果である．

RyORB が最も小さい遅延を示している．MetaVM は他のシステムと同程度の性能であることがわかる．

7.1.2 フィールド アクセス

図 10，11は，32bit 整数（`int`）型のフィールドに対する遠隔書き込み，読み出しの遅延を示している．MetaVM では遠隔フィールドアクセス，他のシステムでは，フィールドアクセスのみを目的としたアクセサメソッドの遠隔呼び出しを行った．

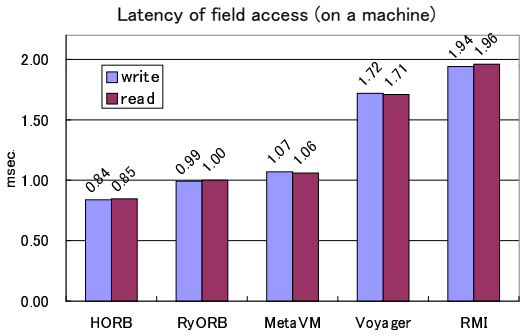


図 10 遠隔フィールドアクセスの遅延 (1 台上)
Fig. 10 Latency of remote field access(on a machine)

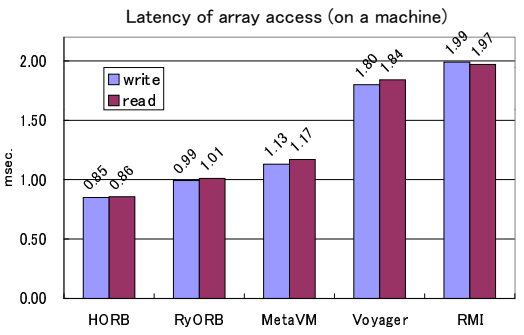


図 12 遠隔配列アクセスの遅延 (1 台上)
Fig. 12 Latency of remote array access(on a machine)

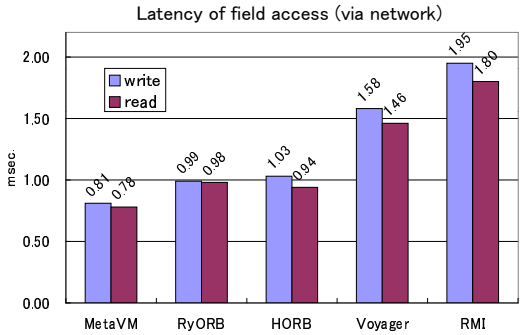


図 11 遠隔フィールドアクセスの遅延 (ネットワーク経由)
Fig. 11 Latency of remote field access(via network)

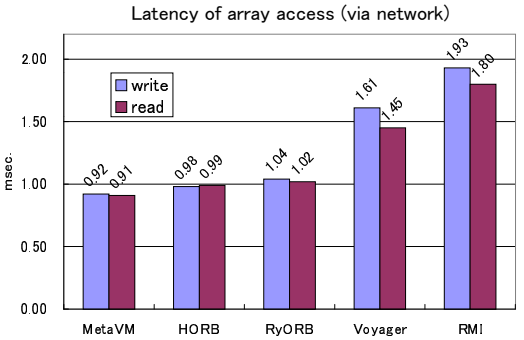


図 13 遠隔配列アクセスの遅延 (ネットワーク経由)
Fig. 13 Latency of remote array access(via network)

システム	性能向上比
MetaVM	1.36
Voyager	1.17
RMI	1.09
RyORB	1.02
HORB	0.90

表 1 2 台での性能向上比
Table 1 The ratio of performance improvement with two machines

メソッド呼び出しと比較して，MetaVM が相対的に良い結果を出しているのは，MetaVM だけが遠隔フィールドアクセスの機能を持っていて，遠隔呼び出しよりオーバヘッドが小さいためと考えられる．

この実験では特に，1 台での実験と 2 台での実験の結果の比が，システムによって大きく異なることが興味深い．2 台での実験で用いた callee 側マシンの処理性能が caller 側より高いため，ほとんどのシステムでは，1 台での実験よりも結果が良くなっている．表 1 に，read 操作でのこの性能向上比を示す．MetaVM，Voyager，RMI では結果が向上し，RyORB はほぼ変化せず，HORB の結果は逆に 1 割悪くなっている．この比はシステムが持つ次の性質に左右される．

- ネットワークを介した通信を行う際のオーバヘッドの小ささ．
- caller，callee の処理のオーバラップ量の大きさ．

caller，callee の処理をなるべくオーバラップさせることは，システム実装上の工夫である．HORB は，これらの性質が比較的良くないと推測できる．

7.1.3 配列アクセス

図 12，13は，32bit 整数(`int`)型の配列に対する遠隔書き込み，読みだしの遅延を示している．MetaVM では遠隔配列アクセス，他のシステムでは，配列アクセスのみを目的としたメソッドの遠隔呼び出しを行った．

7.2 ローカルな処理の性能

第 4章で述べたように，MetaVM ではオブジェクトに対する操作時に動的に，操作対象が遠隔参照かローカル参照かの判定が行われる．この判定は操作対象がローカル参照であった場合にも行われ，オーバヘッドとなる．どの程度の性能低下が起こるのかを調べた．

図 14，15はそれぞれ，CaffeineMark3.0[13]，Linpack benchmark[15] での結果である．CM3 の結果は，ある処理を一定時間内の繰り返せた回数，Linpack benchmark の結果は Mflops で，どちらも数値は大きい方が良い結果を表す．

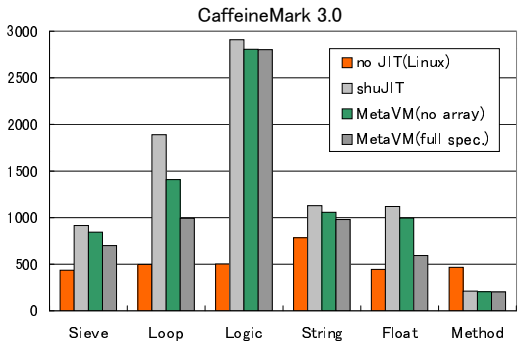


図 14 ローカル実行の性能 – CaffeineMark 3.0
Fig. 14 Performance of local execution – CaffeineMark 3.0

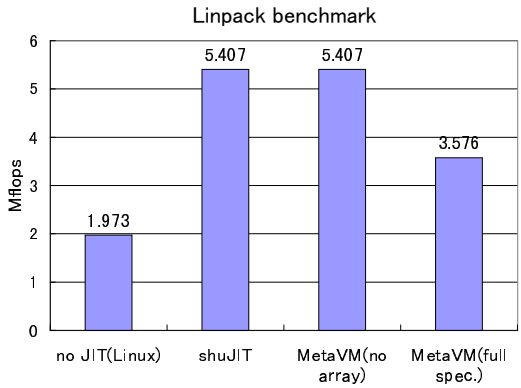


図 15 ローカル実行の性能 – Linpack ベンチマーク
Fig. 15 Performance of local execution – Linpack benchmark

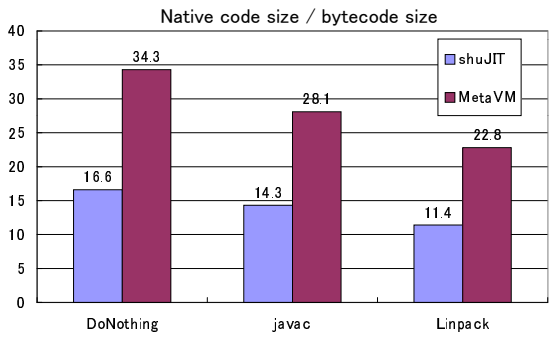


図 16 ネイティブコードとバイトコードのサイズの比
Fig. 16 The ratio of native code size to bytecode size

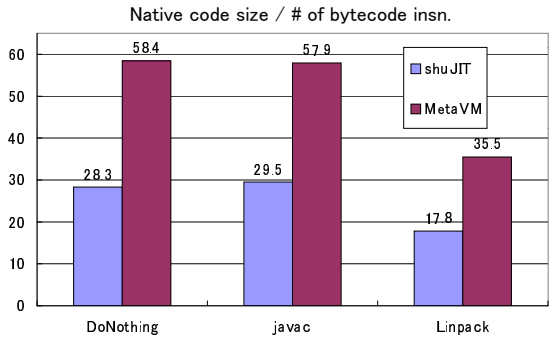


図 17 ネイティブコードのサイズとバイトコード命令数の比
Fig. 17 The ratio of native code size to the number of bytecode instructions

4 通りの条件で計測した．グラフの左から，インタプリタ (no JIT)，非分散版の素の shuJIT，配列を遠隔操作する機能を無効にした MetaVM (no array)，全機能を有効にした MetaVM (full spec.) と並んでいる．

MetaVM によって最も結果が下がったベンチマーク (CM3 の Float) でも，非分散版 shuJIT の 53%への低下に抑えられている．また，全機能を有効にした MetaVM でも，もともと非分散版 shuJIT が Java インタプリタより遅かったベンチマーク (CM3 の Method) を除いては，インタプリタより悪い結果は出ていない．

配列を遠隔参照する機能を無効にしてしまうと，ネットワーク越しに配列を渡そうとした際に，参照ではなく配列のコピーが渡ることになる．すると，メソッド呼び出しの際の call by reference の原則が崩れ，プログラムによっては分散実行とローカル実行とでセマンティクスが変わり得る．このような透過性の低下を許容できる場合に性能低下をどの程度防げるのかも，実験結果は示している．例えば，そのカーネルに配列ではないオブジェクトの操作を含まない Linpack benchmark では，性能低下をなくすことができている (図 15)．

7.3 生成されるネイティブコードの大きさ

MetaVM では，通常の実行時コンパイラが生成するコードに加えて，遠隔参照を扱うためのコードも同時に生成される (第 4章)．そのため，通常の実行時コンパイラよりも，生成されるコードのサイズが大きくなる．どの程度大きくなるのかを，非分散版の shuJIT と MetaVM に対応した shuJIT で比較した．

図 16は，バイトコードのサイズに対する，生成されたネイティブコードのサイズ (バイト) の比を，図 17は，バイトコード命令の数に対するネイティブコードのサイズの比を示している．どちらも，ある処理を行った際にコンパイルされた全メソッドについて，ネイティブコードのサイズ，バイトコードのサイズ，命令数を合計し，その比を求めたものである．shuJIT では，ネイティブコードへのコンパイルはそのメソッドが初めて呼び出されたときに行われるので，一度でも実行されたメソッドについての合計を計測したこと

になる．

3通りの処理で実験した．左の “DoNothing” は，次に示す，何も処理のないプログラムを実行した場合，

```
class DoNothing {
  public static void main(String[] argv) {}
}
```

つまり，どのようなプログラムを実行する際も，Java 仮想マシンを起動しさえすれば実行されるメソッド群についての実験である．真中の “javac” は，そのクラス DoNothing を JDK 付属の Java コンパイラ（Java 言語で記述されている）でコンパイルした際の結果である．右の “Linpack” は，Linpack benchmark[15] を実行した際の結果である．

この比を計算する元になったネイティブコードは，メソッドの頭と末尾に shuJIT が付加するプロローグ，エピローグのコード，また，たいていは実行されない例外処理のコードも含んでいる．そのため，バイトコードが小さいメソッドが多いほど，それに対するネイティブコードサイズの比は大きくなる．一般に，コンパイラや，プログラムの起動準備のためのシステム寄りのコードの方が，Linpack のような計算集約的なプログラムよりもメソッドあたりのコード量が小さいため，ネイティブコードサイズの比が大きくなっていると推測できる．

MetaVM 対応の shuJIT が生成するコードは，非分散版の shuJIT のほぼ倍の大きさになってしまっている．これは MetaVM の問題のひとつである．

8. 他の手法および関連研究

8.1 静的な実現手法

分散オブジェクトシステムの構成に実行時コンパイラを利用する本手法の抱える問題のひとつは，オブジェクトへの操作時に，操作対象が遠隔参照であるかローカル参照であるかの判定を動的に行うことによるオーバーヘッドである．動的な判定が問題となるなら，ソースコードに対するプリプロセッサを用意して，遠隔参照を扱えるコードを静的に生成しておくことが考えられる．現在の MetaVM 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は静的にコード中で指定されるだけであり（第 2.1節），これはこの静的なアプローチの障害にはならない．

とはいえ，プログラム実行中に起こるオブジェクト操作のすべてについて，操作対象がローカル参照であるか遠隔参照であるかを静的には解析し尽くせない場合がある．メソッド中のすべてのオブジェクト操作について，その対象がローカルか遠隔かが，もしメソッドの引数だけに依存するならばよい．引数のうち参照型のものすべてについてローカル参照，遠隔参照それぞれの場合を考慮してすべての組み合わせに応じたメソッドを生成すればよい．例えば以下のコードを

```
class Foo {
  void method(Bar o1, Baz o2) { ... }
}
```

次のように変換する．

```
class Foo {
  void method(Bar o1, Baz o2) { ... }
  void method_1(Bar_Stub o1, Baz o2) { ... }
  void method_2(Bar o1, Baz_Stub o2) { ... }
  void method_3(Bar_Stub o1, Baz_Stub o2) {
    ...
  }
}
```

ここで “クラス名_Stub” は遠隔参照を表すスタブクラスである．ところが実際は，操作の対象がローカルか遠隔かは，その操作を含むメソッドの引数だけでは決まらない．自身以外のオブジェクトのフィールドをアクセスしたり，メソッド呼び出しの返り値として参照を受け取ることがある．このように得られる参照についての解析は難しい．さらに Java ではスレッドがどのようにスケジュールされるかは実行時に決まる．つまり，引数以外から得られる参照がローカルか遠隔かがスケジューリングに依存する場合もあるので，完全に静的に解析することはできない．

また，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は静的に指定されるだけだが，今後，生成先を動的に計画し，メモリ空間の使用量をマシン間で調整することを計画している．この目的には，操作対象を動的に判定する必要がある．

8.2 Java インタプリタの改造

実行時コンパイラに生成させるコードを変更して実現できることなら，バイトコードインタプリタを改造することでも実現できる．しかし，インタプリタを改造したところで，改造内容は実行時コンパイラが生成するネイティブコードにまでは反映されない．実行時コンパイラを利用したければ，結局，実行時コンパイラも改造する必要がある．はじめから実行時コンパイラを改造した方が手間がかからない．

8.3 実行時コンパイラが生成するコードの変更

MetaVM 以外にも，実行時コンパイルを実行性能の向上以外にも応用しようという研究はいくつかなされている．

松岡らの OpenJIT[7] は，Java 言語自身で記述された Java の実行時コンパイラである．プログラマが，

Java 言語で実行時コンパイラの挙動を制御，変更することのできるインタフェースを提供することを目標としている．プログラミングインタフェースの一般的な性質として，仮想マシンやコンパイラの内部構造に深く依存せず，可搬性が高く，利用し易いことと，コンパイラをカスタマイズする能力の高さ，自由度は相反する．この双方の性質を兼ね備えたインタフェースをいかに設計するかが課題であろう．

UCB では，Java 言語から Myrinet を利用した低遅延通信を行うための研究がなされている．そこでは，Java のプログラムから直接ネットワークインタフェースを制御するために，ネットワークインタフェース上のメモリを Java の配列にマップするために我々の shuJIT が利用されている．

9. おわりに

本稿では，実行時コンパイラを利用してネットワーク透過な分散オブジェクトシステムを構成する手法を提案し，それに基づいて設計，開発したシステムの概要を述べ，遠隔操作およびローカル実行の性能，そして透過性の評価結果を示した．提案した手法により，遠隔オブジェクトに対するローカルオブジェクトと同様の操作が実現され，わずかな制約のもとで，マシン群を単一マシンとみなしてプログラミングすることが可能となった．高い透過性を達成しつつ，遠隔操作の遅延は既存システムと同程度に抑えられている．

また，実行時コンパイルを応用した研究の素材とすべく開発し，本研究の基盤として利用した実行時コンパイラについて，コード生成手法を述べ，基本性能の評価結果を示した．

MetaVM には，オブジェクト移送のサポート，高速化などの課題が残されている．Java 用分散オブジェクトシステムの応用を広げるためには，遅延のオーダーを減らす必要がある．現在の Java 用システムでは，遠隔操作の遅延が同一マシン上ですら数百マイクロ秒から数ミリ秒と，ハードウェアの持つ性能に比して非常に高い．

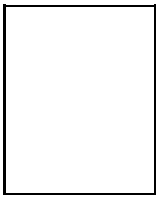
MetaVM を分散共有メモリシステムとして発展させることも考えられる．MetaVM では配列の遠隔参照が可能なので，分散したプログラム，スレッド群が共通の配列への遠隔参照を持ち，アクセスすることが可能である．つまり，原始的ではあるが分散共有メモリを提供している．ただし，現在の通信プロトコルはキャッシュを考えていないため共有メモリとしての性能は低い．計算集約型アプリケーションのために，配列中のブロック単位の転送や，キャッシュプロトコルの設計，実装は興味深い問題である．

参考文献

1) Object Management Group, Inc.: CORBA/IIOP 2.2 Specification. http://www.omg.org/corba/corbaiiop.html.
2) Sun Microsystems, Inc.: Java^{TM} IDL Documentation. http://www.javasoft.com/products/jdk/idl/.
3) ObjectSpace, Inc.: Voyager. http://www.objectspace.com/products/Voyager/.
4) Wollrath, A., Riggs, R. and Waldo, J.: A Distributed Object Model for the Java(tm) System, *The Second Conference on Object–Oriented Technology and Systems (COOTS) Proceedings*, pp. 219–231 (1996).
5) 根山亮: RyORB—Ryo's Object Request Broker for Java. http://www.info.waseda.ac.jp/muraoka/ryo/RyORB/.
6) Lindholm, T. and Yellin, F.: *The Java^{TM} Virtual Machine Specification*, Addison Wesley (1997).
7) 松岡聡, 小川宏高, 志村浩也, 木村康則, 堀田耕一郎, 高木浩光: OpenJIT —自己反映的な Java JIT コンパイラ—, 電子情報通信学会技術研究報告, CPSY98-67, pp. 49–56 (1998).
8) 首藤一幸: shuJIT—JIT compiler for Sun JVM/x86. http://www.shudo.net/jit/.
9) Hirano, S.: HORB: Distributed Execution of Java Programs, *Proceedings of World Wide Computing and Its Applications* (1997).
10) 志村浩也, 木村康則: Java JIT コンパイラの試作, 情報処理学会研究報告, 96-ARC-120, pp. 37–42 (1996).
11) 首藤一幸, 村岡洋一: Java 言語環境におけるスレッド移送手法と移動エージェントへの応用, 電子情報通信学会技術研究報告, CPSY98-32, pp. 39–46 (1998).
12) Kleine, A.: TYA Archive. ftp://gonzalez.cyberus.ca/pub/Linux/java/.
13) Pendragon Software Corporation: CaffeineMark 3.0. http://www.pendragon-software.com/pendragon/cm3/.
14) U.S. Department of Commerce, National Institute of Standards and Technology: Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES), *Federal Register*, Vol. 62, No. 177, pp. 48051–48058 (1997). http://csrc.nist.gov/encryption/aes/aes_home.htm.
15) Dongarra, J. and Wade, R.: Linpack Bench-

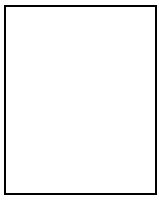
mark — Java Version. http://www.netlib.org/benchmark/linpackjava/.

(平成 11 年 3 月 10 日受付)
(平成 11 年 4 月 28 日採録)



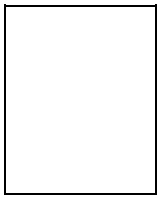
首藤　一幸 (学生会員)

1996 年早稲田大学理工学部情報学科卒業．1998 年同大大学院修士課程修了．同年同大メディアネットワークセンター助手．現在，同大大学院博士後期課程に在学中．分散処理方式，プログラミング言語，言語処理系，情報セキュリティ等に興味を持つ．IEEE–CS 会員．



根山　　亮 (正会員)

1997 年早稲田大学理工学部情報学科卒業．1999 年同大大学院修士課程修了．同年，日本アイ・ビー・エム (株) 東京基礎研究所に入所し，現在に至る．分散並列処理，音楽情報処理等に興味をもつ．日本ソフトウェア科学会会員．



村岡　洋一 (正会員)

1965 年早稲田大学理工学部電気通信学科卒業．1971 年イリノイ大学電子計算機学科博士課程修了．Ph.D．この間，Illiac-IV プロジェクトで並列処理ソフトウェアの研究に従事．同学科助手ののち，日本電信電話公社 (現 NTT) 電気通信研究所に入所．1985 年より早稲田大学理工学部教授．現在同大学メディアネットワークセンター所長．並列処理，マンマシンインタフェース等に興味を持つ『コンピュータアーキテクチャ』(近代科学社) など著書多数．