メタコンピュータ構成方式の研究

A Construction Method of Metacomputer
as Network Transparent Programming System

2001 年 3 月

首 藤 一 幸

メタコンピュータ構成方式の研究

A Construction Method of Metacomputer as Network Transparent Programming System

2001 年 3 月

首 藤 一 幸

Kazuyuki Shudo

目次

図目次

表目次

第 1 章

序論

メタコンピュータとは，単一目的に利用できる，ネットワーク上に分散したコンピュータ群を一台の仮想コンピュータに見立てて呼んだ言葉である．本研究の対象は，メタコンピュータを構成する基盤ソフトウェアのあり方と実装法である．特に，プログラマにどのようにプログラムを書かせるか，すなわちプログラミングモデルに注力する．分散処理の目的は，プログラム全体の処理能力向上や高性能計算であることが多いので，各コンピュータの処理能力を最大限活用することもまた重要な目標である．

1.1 研究の背景と関連研究

1.1.1 インターネット上での分散処理

インターネット接続をもつコンピュータは数千万台を数え，パーソナルコンピュータ (PC)，スーパーコンピュータはネットワークに接続されることがごく当たり前となった．携帯電話器，Personal Digial Assistant(PDA) に代表される，より小規模なコンピュータのネットワーク接続も進んでいる．ネットワーク接続されたコンピュータの著しい増大にともなってその利用方法は多様化しており，ファイル交換，電子メイルの配送といった基本的な応用から始まったインターネットは，今では広域分散処理，すなわちグローバルコンピューティング [22] の基盤としてみなされるようになった．

分散処理とは，単独でも稼働可能な複数のコンピュータをネットワーク接続してひとつの目的に用いる処理を指す．その形態も様々である．例えば，データベース管理システムや World Wide Web(WWW) に代表されるクライアントサーバ方式は，古くから用いら

れ，今なお応用範囲の広い重要な方式である．このように，一対一，または一対少数で通信が行われる，単純で小規模な分散処理が普及すると同時に，昨今では，より複雑で大規模な分散処理方式が現実に適用，実装，運用されるようにもなってきた．

中，大規模分散処理への取り組みを地理的な分散の度合いで分類すると，一方の局には，ローカルネットワークを前提とするクラスタコンピューティングがあり，もう一方には，インターネットを通信基盤とするグローバルコンピューティングがある．クラスタは，それ単体で稼働できる単位計算機，Processing Element(PE) を集めた並列計算機であり，PE 群は何らかのネットワークで相互接続されている．接続網としては，低遅延，高帯域で特定目的向けに設計された System Area Network(SAN) [9] が用いられることもあるが，Local Area Network(LAN) 接続に用いるイーサネットの性能が向上したことで，日常品であるイーサネットと TCP/IP を用いた並列，分散処理が広く行われるようになっている．

PE 間の通信にインターネットのプロトコルである TCP/IP を用いると，PE 群が同一の LAN に接続されていようと別の LAN に分散して配置されていようと，並列分散処理ソフトウェアはそれを関知する必要がない．PE 群を分散配置すれば期待できる通信性能は低くなるが，それでも充分な性能が得られる場合や応用はあり，また，地理的に離れた複数のクラスタを単一の目的に利用することも可能となる．クラスタのハードウェアとソフトウェアを用いながら，PE の分散配置を考え始めると，それはグローバルコンピューティングと明確には区別できなくなる．これは，TCP/IP，つまりインターネットプロトコルのスケーラビリティを反映した結果である．つまり，イーサネットに代表される日常品や事実上の標準である TCP/IP を採用することで，LAN 上での並列処理からグローバルコンピューティングまで，あらゆる規模の分散処理で，同じソフトウェアを用いることができるようになったのである．

実際には，いくら TCP/IP がスケーラブルだからといって，クラスタ向けのソフトウェアをそのまま広域に分散したコンピュータ群に適用したところで，分散処理の手間に見合う実行効率が得られるとは限らない．そのため，クラスタで用いるソフトウェアを広域に適用しようという研究 [21][1] がなされる一方で，広域分散処理ならではの問題への取り組みも盛んになされている．広域ならではの問題とは，PE の数が多いことや分散していることに起因する，処理能力やネットワーク帯域幅の非均質さ，大きな通信遅延，ネットワークの低い信頼性，PE の所有者が様々であることなどの諸問題である．また，広域分散処理の一典型であるボランティアコンピューティングでは，悪意を持った PE への対処

法といった問題も発生している．現在のボランティアコンピューティング [13][45][16] では，数百から数百万の PE を扱うために，パラメータサーチのようにデータ並列性を利用することが普通であり，各 PE に広大な探索空間の一部を順次割り当てていく．その際に，悪意を持った PE が割り当てを受けつつ処理を行わなかったとしたら? 故意に正しくない結果を返したら?といった事態を想定しなければならない．これらはまさに，広域分散によって発生した新しい問題である．

1.1.2 分散処理のプログラミングモデル

分散処理とは PE 群を単一目的に活用することなので，単一 LAN 上だろうと広域分散処理だろうと，PE 間で何かしらの情報を共有する必要が生じる．分散処理のハードウェアは，各 PE がそれぞれ独立したローカルメモリを持つ分散メモリ型並列計算機とみなすことができるので，情報共有のために PE 間の通信が必要となる．そのため，プログラムを記述する際には，単一のコンピュータに対する逐次プログラミングとは異なった作法，つまり，プログラミングモデル，API に沿うことが要求される．もちろん，逐次プログラムの分散処理プログラムへの自動変換も考えられるが，分散した計算能力の有効利用や，遠隔にある目的の資源へのアクセスを達成するためには，プログラマが明示的な指示をする必要がある場合が多い．指示の方法は様々で，メッセージの送受信を明に記述する場合もあれば，ヒントを記述するだけの場合もある．いずれにせよ，プログラマの労力と，性能を含めた実行効率の間にトレードオフがある．

プログラマが PE 間の情報交換，通信をどのように記述するか，これがすなわち分散処理のプログラミングモデルである．これまで，目的に応じて多くのモデルが提案されてきた．いくつかのモデルは，ハードウェアやオペレーティングシステム (OS) の提供してきた通信方式をそのまま提供し，プログラマが通信を直接記述する．TCP が提供する仮想回線や，並列計算機で利用可能なメッセージ通信，また，共有メモリはこの部類である．例えば，PVM [25] はメッセージ通信モデルの TCP/IP 上の実装であるし，MPI [19][20] は下位の通信方式は TCP/IP 他様々であるものの，提供するプログラミングモデルはメッセージ通信である．このように，どのような通信方式の上であっても，ソフトウェア，ハードウェアシステムによって，任意のプログラミングモデルを提供できる．分散共有メモリは，ハードウェアとしての共有メモリを持たない PE 群に共有メモリプログラミングモデルを提供するものであり，この種の，下位通信方式とプログラミングモデル

が一致しないシステムの代表である．

メッセージ通信や仮想回線は，ある問題領域，特定のプログラミング言語とは独立した概念であり，どのような目的の分散処理にも用いることができる．また，プログラマが通信を直接記述するので，通信に関する最適化をプログラマ自身の手で細かく行うことができる．送受信のタイミングを調整することで各 PE の通信待ち時間を減らしたり，ハードウェアが通信と計算の同時処理可能なものならば，通信と計算を重ね合わせたりできる．その反面，また，通信の内容，送受信の整合など，通信に関する責任をプログラマが負わねばならないため，プログラムが想定した動作に至るまでの労力は非常に大きい．

プログラマの労力を軽減するために特定の問題領域に特化した，目的指向のプログラミングモデルもある．例えば，WWW サーバやデータベース管理システムが行う通信は，問い合わせを受けて返答を返すだけであり，メッセージ通信のサブセットととらえることができる．プログラマは，メッセージ通信や仮想回線上での通信を直接記述してもよいが，WWW，データベースアクセスといった目的に応じた API やライブラリを用いることもできる．その場合，プログラマが意識する必要があるのは問い合わせの発行と返答だけとなり，換言すれば，問い合わせ–返答に基づく単純化されたプログラミングモデルが提供されたと言える．目的指向のプログラミングモデルは，問題に対する汎用性を敢えて捨てることでプログラミングを単純化し，プログラマの労力を減らしているのである．

目的指向ではなく，問題に対する汎用性を失わずに分散プログラミングの労力を減らす試みの中で，プログラミング言語の機能をネットワーク経由で働かせようというというプログラミングモデルは一定の成功を収めている．Remote Procedure Call(RPC) は，ネットワーク経由で別のコンピュータに対して行う関数，手続き呼び出しを指し，その単純さゆえに広く用いられている．Sun RPC [63] は Network File System(NFS) といったコンピュータ群の利用に欠かせないサービスの基盤として世界中で実用されているし，また，RPC を分散高性能計算に適用する試み [10][38] もなされている．オブジェクト指向言語での RPC は Remote Method Invocation(RMI)，つまり遠隔メソッド呼び出しと呼ばれ，分散オブジェクトシステムのごく基本的な機能となっている．

分散オブジェクトもまた RPC と同様に，言語の機能に着目して考えられたプログラミングモデルである．オブジェクト指向言語の概念であるオブジェクトを，コンピュータ群に分散させる単位としている．オブジェクトに対する操作，主にメッセージ送信をネットワーク経由で行いつつ分散処理を進める．分散オブジェクトには商業的な需要があったため，CORBA/IIOP [40] という，各種言語に応じたプログラミングインタフェースおよび

TCP/IP 上のプロトコルの標準も作成された．製品，無償ソフトウェア合わせて，数十以上の実装がある．CORBA 準拠ではない独自プロトコルも用意している分散オブジェクトシステムもあり [59][41][34]，これらは，独自プロトコル上では CORBA では利用できない機能を提供している．

1.1.3 Java 言語

分散オブジェクトシステム RMI [59] こそ言語や標準 API の範囲外ではあるものの，Java は，ネットワークを強く意識して設計された言語として初めて開発者の間で広く受け入れられた．Java は 1995 年に Sun Microsystems 社が発表したプログラミング言語であり，言語仕様 [8] と標準 API の仕様，また仮想マシン仕様 [37] で規定されている．

Java 言語および Java 仮想マシン (Java Virtual Machine, JVM) は，プログラムコードの**安全な実行**，**機種非依存性**，そして**高速実行**の 3 点を同時に達成したという理由で，分散処理研究において重要な位置を占める．

Java 言語は必ずしも仮想マシンの存在，仮想マシン上での実行を仮定していないが，ネットワーク経由でプログラムコードを移動させるためのいくつかの機能は Java 仮想マシンが実現している．まず，ネットワークの先にあるプログラムコードを安心して実行するための安全性は，Java 仮想マシンを用いないと達成できない．Java 言語のプログラムを特定プロセッサの実行形式にコンパイルすることは可能だが，変換後の実行形式をネットワーク経由で入手したとして，手もとのコンピュータで何をされるか判らないため，誰しもが安心して実行することはできない．手もとの私的な情報を読まれたり，ファイルを消去されてしまう不安は拭えない．プログラムコードに電子署名がされていれば，署名元を信頼する限りは安心できるかというとそうでもなく，署名元が意図しないプログラム誤りによって問題が起こる可能性は残されている．一方，Java 仮想マシンは，ネットワーク経由で入手したコードの安全性を実行前に検証する機能を持っている．プログラマが意図しようとするまいと，セキュリティ的に問題のある処理が行われないかどうか，事前に検出できるのである．また，プログラムの実行中も安全性のチェックは行われる．つまり，そのプログラムに読み書きの許されないメモリ領域へのアクセスや，許されない処理，例えばファイルアクセスなどを行わせないことができる．

機種非依存性も Java 仮想マシンが実現している．Java 仮想マシンが実行する中間コードである Java バイトコードは特定のプロセッサに依存しないので，バイトコードを格納

しているクラスファイルは機種非依存であり，どのコンピュータの上でも実行が可能である．機種非依存性だけならば，他の手段でも達成できるのだが，Java 仮想マシンが新たに設計されたことには理由がある．ソースコードを各マシン上でコンパイル，もしくはインタプリタで実行することが考えられるが，実行前にコンパイルや構文解析に時間がかかってしまう．特定プロセッサ，OS の実行形式を別機種ではエミュレートするという手段もあるが，安全性のためのコード検証を考慮して設計されたわけではないため，コード検証がとても繁雑になるか行えないかのどちらであろう．それに対して，クラスファイルはコード検証を考慮して設計されている．例えば，それぞれのメソッドが読み書きし得るスタックの最大深さがクラスファイルに記録されていて，コード検証時に利用される．また，ファイルアクセスといった安全性チェックが必要な処理は特定のクラスやメソッドに集められていて，それらが実行時に安全性チェックを行うようになっているので，漏れなくチェックすることが可能である．

Java 仮想マシンがスタックマシンとなっているのも，ネットワーク上でのコード移動のためである．スタックマシンではレジスタマシンと比較して，命令のオペランドを少ないビット数で表現できる．レジスタマシンでは多くの場合オペランドはレジスタを指すので，レジスタ数 n とするとオペランドひとつが $\log_2 n$ ビットとなる．しかしスタックマシンでは，多くの命令ではオペランドは暗黙のうちにスタックトップであり，オペランド自体が不要である．そのため，クラスファイルのサイズを小さくすることができ，ネットワーク経由の転送に有利となる．

Java 仮想マシンは，上記の安全性と機種非依存性に加えて，高速実行も達成している．仮想マシン内の実行時コンパイラがプログラムの実行中にバイトコード命令列をネイティブコードにコンパイルすることで，仮想マシンによる恩恵を損なうことなく，同時に，高い性能を達成する．もちろん，Java 仮想マシンを用いずに，実行前にネイティブコードにコンパイルしてしまえば，言語仕様が原因であるいくつかのペナルティ [32] を除いては，C/C++言語で記述されたプログラムに性能面で劣る理由はない．しかし Java 仮想マシンを用いる場合でも，実行時コンパイラが C 言語のコンパイラと同等かそれ以上の最適化を行い，C/C++のプログラムに近いかそれ以上の性能が得られる．あらゆる Java 仮想マシンがそれだけ高い性能を達成しているわけではないとはいえ，C/C++と同等の性能を達成した実装があるという事実が，Java 仮想マシンの仕様，設計が元からそれだけの高性能を達成し得る枠組であったことを示している．

図 1.1: メタコンピュータの概念

1.2 研究の目的

プログラマに対してメタコンピュータ (metacomputer)(図 1.1) を提供することが研究の目標である．メタコンピュータとは，単一目的に利用できる，ネットワーク上に分散したコンピュータ群を指す．つまり，分散プログラミングを単一コンピュータに対するプログラミングと同様に行えるようにすることを目指す．分散したコンピュータ群があたかも一台のコンピュータに見えるということは，分散処理のための通信に由来する，プログラマの労力が不要になるということである．なので，研究の目的は，ネットワーク透過なプログラミングシステムの構成と換言することもできる．

メタコンピュータを構成するためには，Java 仮想マシンに機能を付加するという方針を採る．分散処理に非常に適している Java 仮想マシンの性質，つまり安全性，機種非依存性，高速実行を保ちながら，機能を付加することで分散 Java 仮想マシン (図 1.2) を構成する．具体的には，**分散オブジェクト**と**実行状態の移送**の機能を付加する．ネットワーク透過な分散オブジェクト機能は，Java バイトコードの実行時コンパイラを元にして達成できる．

分散オブジェクトはネットワークの先にあるオブジェクトを操作する機能で，オブジェクト指向言語から複数コンピュータを活用するための一般的な手段である．実行状態の移送は実行中のプログラムをコンピュータ間で移動させる機能で，複数コンピュータの性能

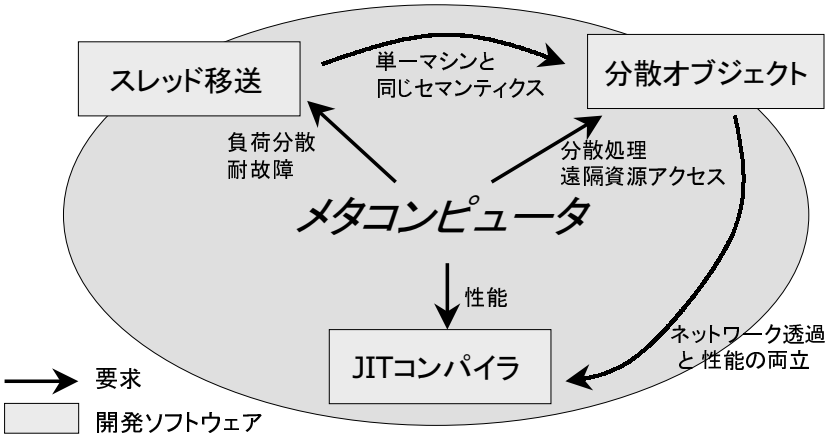

図 1.2: 分散 Java 仮想マシン

を活用するための動的負荷分散には欠かせない．また，実行状態の移送によってあらゆる種類の分散処理を行うことが可能なので，分散オブジェクトの代替ともなり得る．どちらも，分散 Java 仮想マシンに欠かせない機能である．

1.2.1 分散オブジェクト

Java 仮想マシンは，ネットワーク経由で入手したプログラムコードであっても安心して実行できる安全性，Java バイトコードと呼ばれるコンパクトな実行形式の機種非依存性，そして，バイトコードの高速実行を同時に達成した．とはいえ，分散処理のために Java の標準クラスライブラリが提供している機能は非常に単純で，TCP/IP で通信を行うための BSD socket programming interface，通称ソケットという API が用意されているだけである．TCP が提供する仮想回線上のバイト列送受信が行えるだけなので，通信に関する責任を負うプログラマの負担が大きい．これを解決するために，アプリケーションを限定しない分散処理向け機能を提供するソフトウェアとして，多くの分散オブジェクトシステム [59][41][27] が提供されてきた．しかし，これらを利用しての分散処理ソフトウェアの開発は，単一コンピュータ上で稼働するソフトウェアの開発とは多くの点で異なる．開発の際に独特の作法に従う必要があるだけでなく，ネットワークの先にある遠隔オブジェクトに対して行える操作に制約もある．つまり，メソッド呼び出し以外の操作を行えない．また，遠隔メソッド呼び出しと通常の単一コンピュータ上での呼び出しでは，引数渡しのセマンティクスがコピー渡しと参照の値渡しというように異なるので，この違いも意識する必要がある．これらのシステムは，C 言語で利用されてきた Remote

Procedure Call(RPC) の機能を Java 言語で利用できるようにしたものに過ぎない.

これら RPC システムを用いるだけで，確かに，標準クラスライブラリのソケット API を直接用いるよりは，分散プログラミングが容易になる．しかしそれも，コンピュータ一台を対象とした通常のプログラミングとはかなり異なったものである．プログラマの労力，負担を減らすという観点からは，分散プログラミングであってもコンピュータ一台に対するものになるべく近いことが望ましい．そのためにはまず，分散オブジェクトであれば，ネットワークの先にある遠隔オブジェクトと同一コンピュータ上のローカルオブジェクトの間にプログラマから見た違いを無くす必要がある．

メタコンピュータの構成という観点から，スレッド移送 (後述) を行う際にネットワーク透過な分散オブジェクト機能が役立つ．移送の際には，複数のスレッドから参照されているオブジェクト，つまり共有オブジェクトの扱いが問題となる．分散オブジェクトの提供する遠隔参照機能が利用できない場合，共有オブジェクトをスレッドと共に移送すると，元のコンピュータに留まるスレッドからは参照できなくなり，移送しないとすると移送対象スレッドからは参照できなくなる．参照不能を避けるために共有オブジェクトをスレッド移送先にコピーしたとすると，本来単一であるはずのオブジェクトの複製が作られることで，コンピュータ一台上での実行と，移送を伴う分散実行とでプログラムのセマンティクス，実行結果が変わってしまうことがある．ファイルなど特定コンピュータに強く結び付けられたオブジェクトについても同様の問題が起きることが判っている．分散オブジェクト機能が利用できるならば，この問題を解決できるのである．

本研究では，メタコンピュータを実現するための要素技術として，プログラマが**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備えた分散オブジェクトシステムの構成方法に取り組む.

1.2.2 実行状態の移送

実行状態 (execution context, activation record) は，アプリケーションを限定しない分散処理向け機能として非常に有用なものである．**実行中のプログラムを**，実行中の状態を保ったまま**コンピュータ間で移送する**ことを指し，関数型言語 Scheme で言えば継続 (continuation) のコンピュータ間移送に相当する．分散オペレーティングシステムでは，プロセス移送 (process migration) [17] として研究されてきた．Java 仮想マシンではスレッド (群) の移送にあたる.

実行状態の移送は動的負荷分散のために必須の機能である．実行状態の移送機能がなくともアプリケーション自身が自らの実行状態を保存，復帰するという方法もあるが，この方法ではプログラム上の限られた時点でしか移送できず，また，アプリケーション開発者の負担が非常に大きい．当然，移送可能なアプリケーションは限定される．そのため，従来より，アプリケーションの外，例えば言語処理系や OS での，実行状態移送に対する支援方法が研究されてきたのである．もちろん，メタコンピュータを構成しようという際も，システムソフトウェアの側で支援することが望ましい．

実行状態の移送は応用範囲が広い．分散オブジェクトで記述できる分散処理はすべて移送でも記述可能だという事実がそれを示している．例えば，遠隔コンピュータで何か処理を行いたければ，対象コンピュータに移動，処理，移動して戻る，というプログラムを記述すればよい．逆に，動的負荷分散は分散オブジェクトでは行えない．また，実行状態はネットワーク経由で送受信できるようにバイト列に変換されるので，そのバイト列を保存することでプログラムのスナップショットを得ることもできる．これは，長時間実行されるプログラムでは，コンピュータやネットワークの故障に対する備えとして有効である．耐故障にも応用できるのである．

プログラミングモデルの観点からは，移動エージェントの記述を容易にするという点が特徴的である．実行状態の移送が可能だと，単一コンピュータ向けのプログラムに移動の指示を挿入するだけで移動エージェントを記述できる [49]．それに対して，実行状態の移送機能なしで移動エージェントを記述しようとすると，プログラマは単一コンピュータ向けのプログラミングとは異なった作法に従う必要がある [36]．これは，プログラマの負担が大きい．

実行状態は，Java 仮想マシンではスレッドが該当するので，実行状態の移送とはスレッド移送 (thread migration) を指す．設計，実装にあたって，分散処理に重要な Java 仮想マシンの機能，すなわち安全性，機種非依存性，高速実行を損なわないことが望ましい．特に，ネットワーク上のコンピュータ群はプロセッサ，OS ともに様々で均質であると仮定するわけにはいかないので，機種非依存性を損なうわけにはいかない．

本研究では，Java 仮想マシンの分散処理向きの諸性質を損なうことなく，実行状態，すなわちスレッドの移送を達成することを目的とする．

1.2.3 実行時コンパイラ

分散オブジェクトシステムのネットワーク透過性は，Java バイトコードの実行時コンパイラ，つまり Just-in-Time(JIT) コンパイラを利用して達成できる．Java 仮想マシンは Java バイトコードを実行するとはいえ，実際は実行時コンパイラが生成したネイティブコードが実行されるので，実行時コンパイラはバイトコードの解釈を決められる，と見ることができる．この点に着目し，本研究では**実行時コンパイラによる解釈変更**を提案する．そして，これを実際に分散オブジェクトシステムの構成に応用してその有効性を示す．

また，プログラムの実行時間短縮は分散処理の大きな目的のひとつなので，実行時コンパイラによる性能向上は，単純に，重要である．解釈変更による分散オブジェクトの支援が第一の目的であるものの，極力高い性能を達成することも目的とする．さらに，実行時コンパイラではコンパイルに費される時間もプログラムの実行時間となってしまうので，いかにコンパイル時間を短くするか，いかにコンパイルの対象となるメソッドを絞るかも重要である．

本研究では，実行時コンパイラによるバイトコード命令の解釈変更を提案すると共に，その手法をネットワーク透過な分散オブジェクトシステムの構成に応用することでその有効性を示す．実行時コンパイラの設計，開発においては，短いコンパイル時間でいかに高い性能を達成できるかを目的とする．また，分散オブジェクトの支援を筆頭に，研究の基盤として利用し易いことも，重要な性質である．

1.2.4 本論文の構成

本章では，研究の背景を述べ，それに基づく本研究の方針と目的を述べた．続く章ではメタコンピュータの構成要素について述べて行く．すなわち，第 2 章で実行状態の移送について，第 3 章でネットワーク透過な分散オブジェクト機能について，そして，第 4 章で Java バイトコードの実行時コンパイラについて述べる．最後に，第 5 章では，本論文で述べた研究成果をまとめる．

第 2 章

実行状態の移送

この章では，メタコンピュータを構成する上で重要な一機能である実行状態の移送手法と，移送システムの構成法を述べる．本研究によって，Java 仮想マシン間での**異機種間**，**非同期**スレッド移送が可能であることが実証された．

2.1 まえがき

実行状態のコンピュータ間移送は，プロセス移送 (process migration) またはスレッド移送 (thread migration) として動的負荷分散や遠隔実行に応用されてきた [33][17][12]．また，近年ではその応用分野に移動エージェントの記述も加わった [58][46]．長い間取り組まれてきたテーマであるにも関わらず，異機種間移送と高速実行の両立，任意の時点での移送などの問題が残されている．

ここで述べる方法で構成した Java スレッドの移送システムは，異機種間での (heterogeneous) 非同期な (asynchronous) 移送を達成している．非同期という語は，移送対象スレッドの協力なしに移送が可能なことを意味する．移送対象のプログラム中に移送の記述がなくとも，別のスレッドが移送を指示することが可能である．別スレッドから移送を指示できることは，動的負荷分散や，複数スレッドの同時移送に必要な機能である．プログラム変換で実行状態の移送を実装 [12][55][46] したのでは非同期移送は行えず，応用が移動エージェントの記述に限定されてしまう．

異機種間移送とは，異なる種類のプロセッサや OS を持つコンピュータ間での移送が可能なことを意味する．コンピュータ群が非均質であることは移送を難しくする．本手法では，一般のプロセス移送とは異なり，プログラムカウンタや仮想メモリといったプロセッサと OS 上の実行状態ではなく Java 仮想マシン上の実行状態を対象とすることで，異機

システム	言語	実行状態の移送	実行方式	異機種間移送	非同期移送
MOBA(本研究)	Java バイトコード	可	インタプリタ	可	可
Sumatra	Java バイトコード	可	インタプリタ	可	不可
JavaGO	Java 言語	可	JIT 可	可	不可
Voyager,Aglets	Java バイトコード	不可	JIT 可	可	不可
Telescript	Telescript	可	インタプリタ	可	不可 †
Emerald	Emerald	可	native code	不可	可
Arachne	C, C++	可	native code	可	不可

† 原理的には非同期移送も可能だが,
別スレッドを移送させる API が用意されていない.

表 2.1: 各移送システムの特性

種間移送を可能にしている.

異機種間，非同期移送を実現しているものの，現在の実装はプログラムの高速実行は達成できていない．Java 仮想マシンは通常，Just-in-Time(JIT) コンパイラと呼ばれる Java バイトコードの実行時コンパイラを持ち，これによって高い性能を達成している．しかし，JIT コンパイラが生成したネイティブコードの実行中は Java 仮想マシン上の実行状態を得ることが難しいため，現実装ではインタプリタでの実行を仮定してしている．しかし，JIT コンパイラを使用しつつ Java 仮想マシン上の実行状態を取得する手法はすでに存在するので，JIT コンパイラ，つまり高速実行との両立はすでに，実装する労力の問題でしかない.

2.2 関連研究

ここでは，本研究で実装したスレッド移送システム MOBA を関連研究と比較し，その特徴を示す．実行状態の移送が可能なソフトウェアシステムを表 2.1 に挙げる．「言語」列に「Java バイトコード」とあるのは，Java 仮想マシン上のプログラムが移送対象ではあるものの，Java 言語で書かれたプログラムに限らないことを示している．移送対象のプログラムが Java 以外の言語で記述されていたとしても，それが Java バイトコードにコンパイルされていれば移送が可能である．一方「Java 言語」とあるシステムでは，移送対象は Java 言語で記述されている必要がある.

ここで挙げたシステムの中で，Sumatra [44] と Telescript [58] が MOBA に最も近い

性質を持つ．MOBA と Telescript は Java バイトコードが対象で，Telescript は独自の Telescript 言語が対象という違いはあるが，移送の方法は類似している．プログラムは仮想マシン内のインタプリタで実行され，仮想マシン上の実行状態を移送する．後述するように，手法はあるのだが，JIT コンパイラとの共存が難しいので (第 2.4.3 項) 実行性能が低くなる．これが難点である．これら 3 システムの内，非同期移送が可能なのは MOBA のみである．Telescript には非同期移送ができない理由はないが，移送対象プログラムの外から移送を指示する API は用意されていない．Sumatra では，スレッドが自分自身を移動させる `go()` メソッドが用意されている．他のスレッドを移動させる方法は用意されていない．原理的に非同期移送が可能かどうかは，論文に記述がなく，またソフトウェアが未公開であるために，不明である．

Fünfrocken の方法 [23] と JavaGo [46] もまた，実行状態の保存および移送，移送先コンピュータでの実行再開を達成している．Arachne [12] は，C，C++言語を対象としている点がこれらとは異なるが，同様の手法を用いている．これらが用いる手法はプログラム変換に基づいていて，移送対象の Java プログラムを実行前に変換する．変換後のプログラムも Java 言語であり，移送のために Java 仮想マシンの支援は一切不要なので，JIT コンパイラと問題なく共存できる．実行性能の点で有利である．しかし，移送対象が Java 言語で記述されている必要があるので，Java 仮想マシンで動作する任意のプログラムを移送できるわけではない．また，任意の時点で移送では移送できず，プログラム中に移送の指示が記述してある時点，つまり JavaGo であれば `go(`**移動先**`)` と書かれた時点でのみ，移送が可能である．そのために，複数スレッドを同時に移送させることができない．他のスレッドから移送を指示することもできないため，動的不可分散への応用もできない．実際，JavaGo は移動エージェントの記述を目的としたソフトウェアである．

Aglets [36]，Voyager [41] は，実行状態の移送は行えないものの，移動エージェントを記述，動作させるための基盤ソフトウェアである．実行状態を保持したままの移動は行えないため，オブジェクトの移動を指示する際に，移動後にそのオブジェクトに対して呼び出して欲しいメソッドを指定する．システム側は，オブジェクトを，それが持つデータ，つまりインスタンス変数の値ごと移動先にコピーし，その後，指定されたコールバックメソッドをそのオブジェクトに対して呼ぶ．移動エージェントを記述する際，もし実行状態も移送されるならば，移動前の処理，移動の指示に続けて，移動後の処理を記述でき，通常のプログラミングとさほど変わらない作法で記述できる．しかしこれらのシステム向けに移動エージェントを記述すると，移動しないプログラムとはかなり異なったプログラム

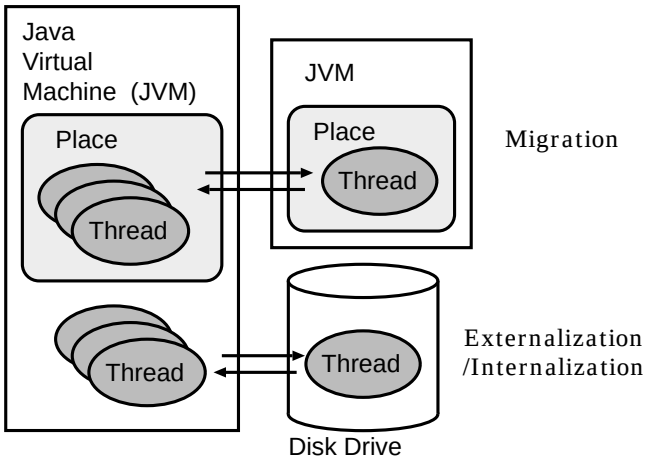

図 2.1: プレイス (place)

となってしまう．

2.3 システムの概要

スレッド移送システム MOBA は，Java 仮想マシンへのプラグインとして実装されている．正確には，Sun 社による Java 仮想マシンの参照実装である classic VM が対象である．classic VM は Sun 社が配付している開発キット，Java Development Kit(JDK) に含まれるので入手しやすく，動作する OS も多い．プラグインであるため，Java 仮想マシンの再コンパイルも不要で，利用できるようにソフトウェアを整える手間が小さい．この性質は，広く使用されるためには重要である．また，プログラムからスレッド移送だけでなく classic VM と JDK の全機能を利用できるので，実用的なプログラムを開発することができる．本システムが研究目的の試作品とは異なる点である．

2.3.1 プレイス

プレイス (place) は，移送スレッドが動作する場である．スレッドはプレイスに対して移送できる．移送先は必ずプレイスだが，プレイス上で動作していないスレッドも移送させることはできる．

一台のコンピュータ上で複数のプレイスを稼働させることができる．その際，各プレイスは TCP のポート番号で識別される．そのため，プレイスのアドレス，識別子は，ポート番号と，ホスト名か IP アドレスの組である．

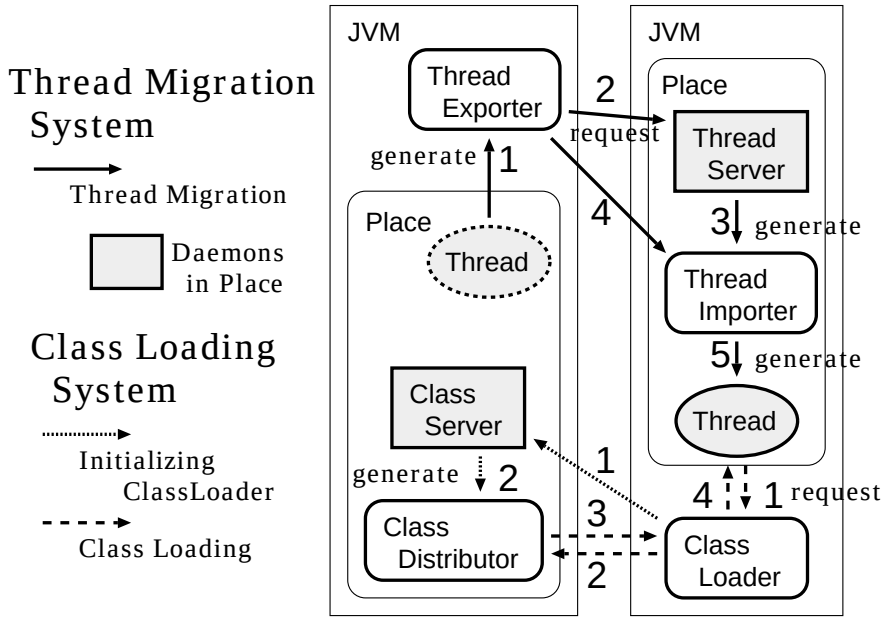

図 2.2: 2 つのサブシステム

2.3.2 サブシステム

スレッドの移送とは，実行状態およびプログラムコードを含めたクラスの移送を指すので (第 2.4.1 項)，本システムは次の 2 つのサブシステムを持つ.

スレッド移送系 スレッド移送を行う系.

クラス転送系 移送スレッドにクラス定義を提供する系.

スレッド移送系は，移送時に次の処理を行う (図 2.2)．ただし，これは移送スレッドが自身で移送を指示した場合の手順である.

1. 移送スレッドがエクスポータを生成.
2. エクスポータが移送先に対してインポータの生成を要求.
3. 移送先でインポータが生成される.
4. エクスポータがインポータに，スレッドを外部化して渡す.
5. インポータがスレッドを内部化.
6. 移送元でスレッドを停止.

クラス転送系とは，具体的には次を指す.

- 移送スレッドが使用する**クラスローダ**
- デーモンである**クラスサーバ**
- クラスサーバがクラスローダごとに生成する**クラスディストリビュータ**

移送スレッドは，自身が生成されたコンピュータからクラス定義，つまりクラスファイルを取得する (第 2.4.2 項).

2.3.3　プログラミングおよびユーザインタフェース

メタコンピュータの構成が目的であるため，プログラミングインタフェースは極力単純なものとした．移動しない通常のプログラムに対する変更はごくわずかで済む．また，非同期移送，つまり他スレッドからの移送の指示が可能であるため，移送されることを想定せずに記述されたプログラムであっても移送できるように変更することは非常に容易である．

スレッドが自らを他のコンピュータに移送するためには, 次のメソッド呼び出しを行う．

`MobaThread.goTo(`**移送先**`)`

ここで移送先は，移送先コンピュータのホスト名または IP アドレスで指定する．移送先では，この呼び出しの直後の文から実行が再開される．もちろん，ローカル変数などの実行状態は移送の際にも値が保たれる．

他のスレッドを非同期移送する場合は，次の形式で `moveTo` メソッドを呼び出す．

＜**移送対象スレッド**＞`.moveTo(`**移送先**`)`

この呼び出しによって，移送対象スレッドが移送先コンピュータへ移送される．

もう一点だけプログラマが知る必要があるのは, 移送したいプログラムは通常の `Thread` クラスではなく，`MobaThread` クラスを用いて記述しておくことである．だからと言って，このクラスは，移送を達成するために原理的に必要なものではなく，上記，非同期移送のプログラミングインタフェースを提供するために用意されたものである．

移送はまた，プログラムから指示できるだけではない．本システムは，他のツールがシステムの外から分散オブジェクト経由で本システムとやりとりするためのインタフェースを用意している．これを利用すると，外部プログラムがスレッド群の状態を調べたり，移送を指示することができる．この外部プログラムインタフェースを利用した

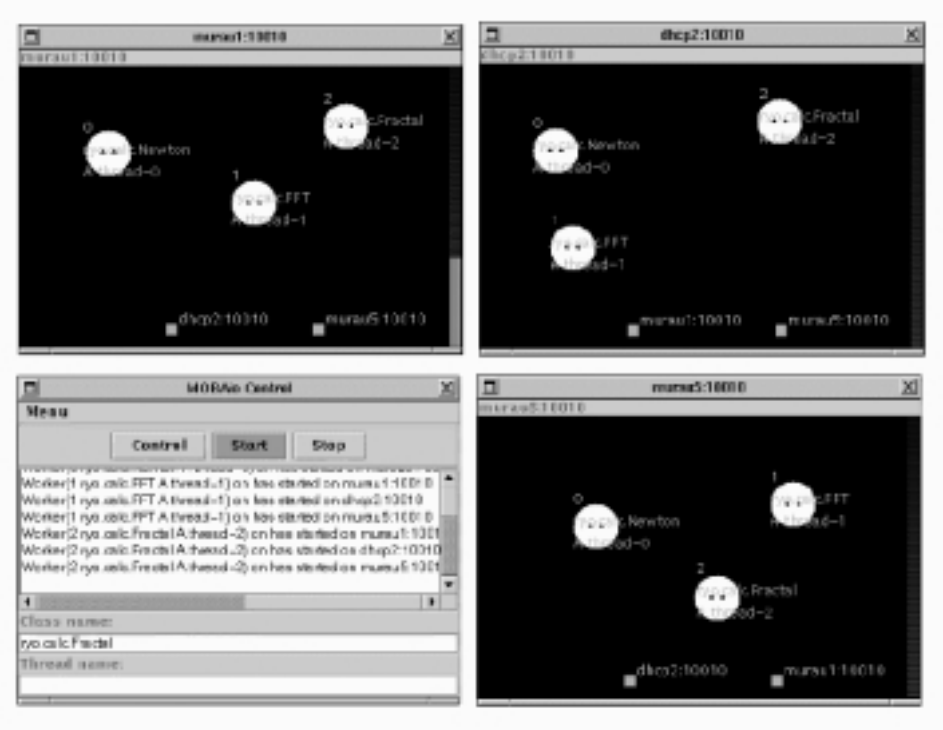

図 2.3: スレッド移送の GUI

GUI(Graphical User Interface) も用意されている (図 2.3)．利用者はこのツールを利用して，どのコンピュータでどのプログラム，スレッドがいくつ稼働しているかを知ることができ，簡単かつ直観的な操作でスレッドの移送を指示できる．

2.4 移送機能の設計

スレッド移送は，以下の機能を提供するライブラリの組み合わせで達成されている．

- スレッドの外部化 (externalization) と内部化 (internalization)
- オブジェクトのマーシャリング (marshaling, バイト列化)
- イントロスペクション (introspection, 内省)

図 2.4 は，各機能の依存関係を表している．それぞれの機能は独立性が高いため，スレッド移送以外の目的にも利用できる．例えば，マーシャリング機能を用いてオブジェクトの永続化を行えるだろう．

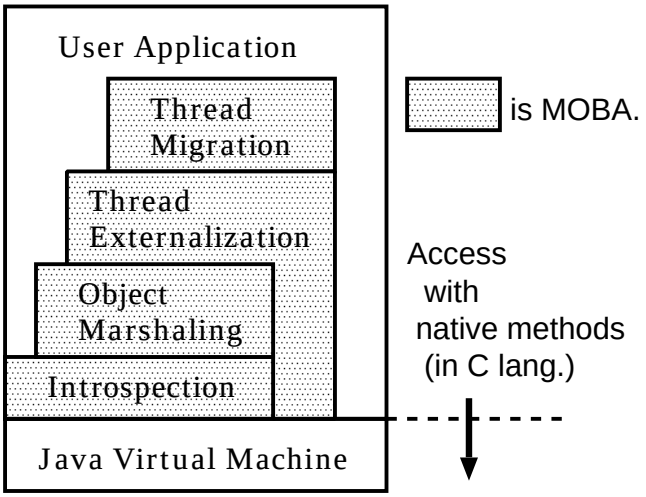

図 2.4: 移送機能の構成

2.4.1 スレッド移送

スレッドの移送とは次の 2 つを移送することである.

- **実行状態**

  スタックフレームごとの
  - スタックフレームに対応するメソッド
  - プログラムカウンタ (PC) — 最後に実行した命令を指す
  - スタックポインタ
  - ローカル変数
  - スタック上の要素 (即値, インスタンスへの参照)
- スレッドが必要とするクラス群の定義

スレッドの移送とは，実行状態の移送だけでなく，クラス定義の移送，つまり型の定義と**コードの移動**も意味する.

実行状態の移送

実行状態の移送は次の手順で行う (図 2.5).

1. 実行状態をバイト列化 (**外部化**)

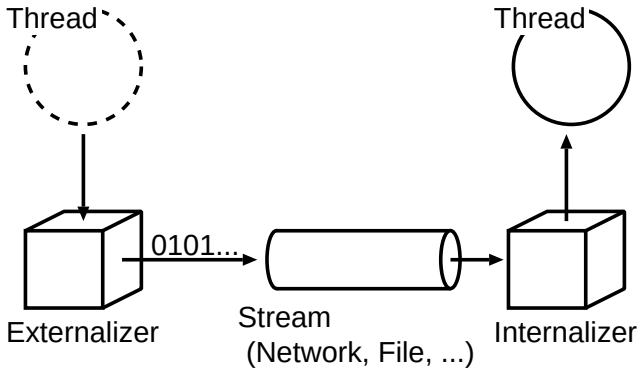

図 2.5: スレッドの外部化および内部化

2. バイト列を移送先に転送
3. バイト列からスレッドを再構成 (**内部化**)
4. スレッドの実行を再開

転送はバイト列化と並行して行われ，再構成はバイト列を受け取りつつ行われる．外部化，転送，内部化の各処理はオーバラップ可能である．

本研究では，実行状態のバイト列化を**外部化** (*externalization*)，そのバイト列を元に JVM 内に実行状態を再構成することを**内部化** (*internalization*) と呼ぶ．スレッドの外部化，内部化手法は，第 2.4.3 項で述べる．

クラス定義の転送

実行状態だけを移送したのではスレッドの移送は達成できない．移送先の Java 仮想マシン内には存在しないクラスをスレッドが必要とするかもしれない．スレッドが必要とするクラス定義も移送する必要がある．クラス定義の移送手法は第 2.4.2 項で述べる．

スレッド間の共有資源

JVM 上のスレッド群はインスタンスを共有し得る．スレッド移送にあたって，複数スレッドに共有されている，つまり複数スレッドから参照を辿って到達し得るオブジェクトをどう扱うかを考えねばならない．移送されるスレッドと留まるスレッドの双方から到達し得るインスタンスがあった場合，双方からそれまで通りローカルに参照することはできない．

個々のインスタンスごとに次の選択肢がある．

1. **移動させず**，移送対象スレッドから遠隔参照する．
2. **移動させ**，留まるスレッドから遠隔参照する．
3. **コピーして**，双方からローカルに参照する．
   （a）コピー元，先の一貫性に配慮する．(遠隔参照のキャッシュとしてのコピー)
   （b）配慮しない．

また，分散共有メモリを利用または実装し，その上で JVM を動作させるという選択肢もある．

実装は，3(b)，1，2 の順に困難になっていく．コピーしてしまってその後の一貫性にも配慮しないという実装が最も容易である．1 と 2 を比較して 2 の実装が困難である理由は，ローカル参照を遠隔参照に置き換えなければならないことである．1 では，移送の際に新たに遠隔参照を用意すればそれで済み，置き換えは不要である．とはいえ，この置き換えの困難さは，Java 仮想マシンと分散オブジェクトシステムの実装に依る．ライブラリ，ツールとして実装されている通常の分散オブジェクトシステムでは，遠隔参照はローカル参照とは異なる型で表されるため，置き換えてしまうことはできない．しかし，本研究の分散オブジェクトシステム (第 3 章) では遠隔参照は型までもローカル参照であるかのように見えるため，置き換えも可能である．この場合，1 と 2 の実装の困難さは同じである．

3(b) のように，共有されたインスタンスを単純に移送先にコピーしてしまうと，単一の Java 仮想マシン上での実行とはプログラムのセマンティクスが変わってしまう場合がある．例えば，移送先にコピーされたインスタンス変数の値を書き換えてもコピー元インスタンス変数の値は変わらない．これは，コンピュータ群に対して単一コンピュータと同じようにプログラムを書かせる，という本研究の目的にそぐわない．ただし，この方法は遠隔参照を必要としないため，遠隔参照を辿ってのネットワーク経由の変数アクセスに伴うオーバーヘッド，性能低下がないという利点はある．

1, 2 のようにプログラムのセマンティクスを保てる方法は，遠隔参照を必要とする．セマンティクスを保ちつつも，可能な限り，遠隔参照による性能低下は少ないことが望ましい．プログラム実行中に遠隔参照を辿らねばならない回数をなるべく少なくするべきである．この回数は，スレッド移送の際にどのインスタンスをスレッドと共に移送するかの選択に大きく影響を受ける．

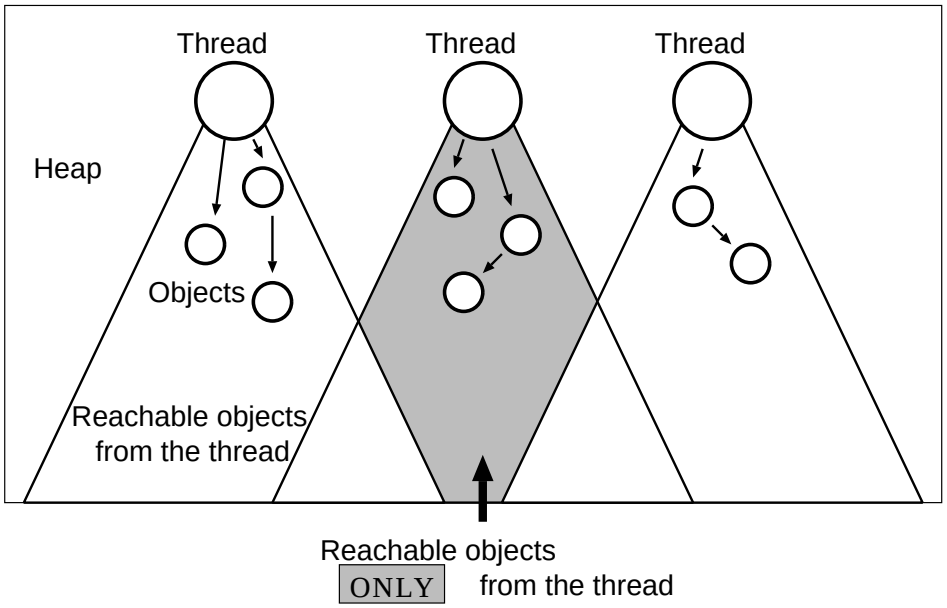

図 2.6: 単一スレッドからのみ到達し得るインスタンス集合

移送対象スレッドと留まるスレッドから共有されているインスタンスについて，移送すべきかどうかを判定することは難しい．プログラマに何らかの形式で指示させることも考えられる．この自動判定法は，残された研究課題である．

一方，スレッド間で共有されていないインスタンス，つまり移送対象スレッドのみから到達し得るインスタンス (図 2.6) なら，スレッドと共に移動させてしまって何も問題ない．あるインスタンスがあるスレッドから参照経由で到達可能か否か，この判定は原理的には可能である．マークアンドスイープ方式のガーベジコレクション (GC) のマーク処理と同様に参照を辿っていくことで調べられる．問題となるのは，実装と処理のコストである．もし Java の標準ライブラリに含まれるリフレクション API にクラスへのアクセス制限が課せられていなかったら，Java 言語で実装することも可能だったが，実際は制限があり，実装できない．Java 仮想マシンがこの判定機能を提供していればよいが，仮想マシン仕様 [37] に含まれていない以上，まったく期待できない．ネイティブメソッド (native methods)，つまり C や C++言語で Java 仮想マシン内に直接アクセスして実装するという手段はある．いずれにせよ，参照を辿って調べるという判定方法を実装する限り，最悪の場合，マークアンドスイープ方式 GC 一回分の処理コストがかかる．

■目標とする手法 移送対象スレッドから到達し得るインスタンスは，スレッド移送にあたって次のように扱うことが望ましい．

- 移送対象スレッドだけから到達し得るインスタンス (図 2.6) は，スレッドと共に移送する．
- 他のスレッドからも到達し得るインスタンスを移送するかしないかは，各スレッドから当該インスタンスへのアクセス頻度に応じて判断する．さもなくば，移送のコストを避けるために移送しない．

■現在の実装　ところが，現在の実装は次のようになっている．

- 移送対象スレッドから到達し得るインスタンスはすべて，移送先に**コピー**する．
- 移送元と移送先でのオブジェクトの一貫性は考慮しない．

一旦，スレッド移送によってインスタンスが移送先にコピーされると，コピーされたインスタンスはコピー元とは別の独立したインスタンスとなる．コピー元が他のスレッドからも到達し得た場合，本来単一であるべき実体が複数生成されてしまう．これによって，移送されるスレッドが直接，間接に参照するインスタンスを他スレッドも参照するプログラムでは，単一 Java 仮想マシン上での動作とセマンティクスが異なってしまうことがある．

現実装はその特殊なセマンティクスを意識して利用する必要があるので，単一 Java 仮想マシンとの透過性，つまり，本研究の目的という点で不満が残る．

目標とする手法の達成には以下が必要である．

- 指定したスレッドからのみ到達可能なインスタンスへの印付け
  あるインスタンスがあるスレッドに参照されているか否かの判定は，実装してある．ただし，処理コストは大きい．
- 遠隔参照
  本研究の分散オブジェクトシステム (第 3 章) との統合．

分散オブジェクトシステムと統合することで，移送を行う場合と単一 Java 仮想マシン上での実行のセマンティクスの違いを無くすことができる．これは残された課題である．

2.4.2　クラス定義の転送

移送されるスレッドが移送先で必要とするクラスの定義が，移送先には存在しないかもしれない．必要なクラス定義が移送先に存在しないと，移送後に実行を続けることができ

	自動	手動
移送先	○	× 移送先が限定される
手間	○ かからない	× かかる
実行時転送コスト	× かかる	○ かからない
安全性に対する配慮	要	不要

○: 良い　×: 悪い

表 2.2: クラス定義の手動，自動転送それぞれの得失

ない．移送システムはクラス定義の転送も支援する必要がある．

ここでは，まず，自動転送と手動インストールについて，特に安全性に関して論じる．続いて，転送のタイミングを，静的転送，動的転送，それらの混合とに分類し，それぞれの得失を述べる．また，移送スレッドのクラス名空間について考察する．

手動と自動の得失

移送先に必要なクラス定義が存在することを保証するために 2 通りの方法が考えられる．移送先となり得るすべてのコンピュータに対して

1. プログラマが**手で**インストールしておく．
2. システムが**自動で**転送する．

それぞれの得失を表 2.2 に挙げる．

非均質な異機種混在環境でのクラス定義の自動転送は，Java 仮想マシンのプラットフォーム非依存性に負うところが非常に大きい．例えば，ネイティブコードが実行されている状況では，移送元と移送先のプロセッサや OS が異なると実行状態の非同期移送は行えない．クラス定義，プログラムコードの自動転送も難しい．異機種混在環境でも，移送先でソースコードから再コンパイル，再リンクによって実行状態を移送しようという研究 [55] はあるが，非同期移送は行えず，移送のコストも大きくなる．

この，クラス定義の配置問題は，スレッド移送に限らず分散処理を行うシステム共通の問題である．Object Request Broker(ORB) を含めた分散オブジェクトシステムもまた例外ではない．システムの設計によっては，遠隔メソッド呼び出しの際に，呼び出し先コンピュータ上でレシーバや引数のクラス定義が存在しない状況が起こり得る．ある種の

- 負荷分散
  相対的に負荷の高い計算機から低い計算機へスレッドを移送させる．
- 移動エージェント
  巡回　いくつかの計算機をあらかじめ決められた経路で巡回する．
  自律的な移動先決定　自ら，移送を受け入れるコンピュータを探したり，移送先を決定したりする．
  …

表 2.3: スレッド移送の典型的な応用

ORB では，呼び出し先コンピュータ上にクラス定義をインストールしておくことは利用者の責任である．手間がかかる上，利用者の配置ミスによって容易に，分散プログラムは正常に動作しなくなる．

とはいえ，ORB では手でのインストールの手間もあまり問題とならない場合も多い．ORB の応用はその多くが，データベースアクセスといった，呼び出し先コンピュータが一台またはせいぜい数台に限定されるクライアントサーバモデルの分散処理である．台数は少数に限られ，呼び出し元，呼び出し先コンピュータは固定なので，インストールの手間はひどく大きくはない．

ところが，スレッド移送では移送先でも実行が継続されるだけで，移送元と移送先に ORB のような上下関係はない．移送先となるすべてのコンピュータ上に，移送スレッドが必要とするクラスの定義が必要となる．さらに ORB とは異なるのは，スレッド移送の典型的な応用 (表 2.3) では多くのコンピュータをまたいで処理が行われる点である．クラス定義に変更があるたびに，すべての移送先へインストールする労力はとても大きい．

以上の理由から，メタコンピュータの構成という目的のためには，自動転送が必須である．プログラマが手でクラス定義をインストールしていたのでは，あらかじめインストールしたコンピュータに対してしか移送できない．また，そういった面倒をシステム側で看てコンピュータ間の垣根をなるべく意識させないことが，メタコンピュータのそもそもの役割である．

反面，自動転送によって，セキュリティに対する配慮が必要となる．クラス定義とは型と振舞いの定義であり，プログラムコードを含む．クラス定義の転送とはつまり**コードの移動**を含むのである．移送の受け入れ側で，あらゆる利用者からのあらゆるスレッドを受

け入れ，そしてあらゆる行動を許してしまっては，受け入れ側コンピュータ上の情報，資源が危険にさらされる．悪意を持ったプログラマによる攻撃だけでなく，プログラマの過失による事故も防がねばならない．受け入れ側を安全に保つための対策としては，例えば下に挙げる処理が考えられる．

- 受け入れスレッドの認証
  - スレッドの生成元コンピュータ，移送元コンピュータ，スレッドを作成した利用者の認証
- 受け入れスレッドの行動制限
  - 実行時間やステップ数の制限
  - 資源へのアクセス制限
    - ファイルの作成，削除，読み書き，リンク
    - 他コンピュータへの接続，接続の受け付け，マルチキャストの使用
    - 他スレッドの操作，プロセスの起動，停止
    - 環境変数など，実行環境に関する情報へのアクセス
    - クラスライブラリ，動的リンクライブラリの利用
    - 印刷
    - ...
  - 再移送の制限
- 処理履歴の記録
  - 資源へのアクセス記録
  - ...

これらのうち一部は，`SecurityManager` といった Java 仮想マシンの機能で達成できる．しかし，実行時間の制限など，Java 仮想マシンは提供していない機能もある．

静的, 動的, 混合

クラス定義を転送するタイミングは，移送したスレッドの実行性能に影響を与える．転送のタイミングとしては次が考えられる．

- **静的な転送**　移送時に，移送先で必要となるクラスを列挙，転送する．
- **動的な転送**　移送後の実行中，必要になったクラスを転送する．

	静的	動的
完全な転送	×	○
実行性能への影響	○	×

○: 良い　×: 悪い

表 2.4: クラス定義の静的，動的転送それぞれの得失

• **混合**　静的，動的転送の混合．

■静的な転送　静的な転送では，プログラムによっては移送先で必要となるクラスを完璧に挙げられない場合がある．例えば，プログラム自身が動的にクラスをロードする場合である．Java の標準 API には，プログラム自身がクラスをロードするメソッド (`java.lang.Class#forName()` 他) が用意されている．クラス名を表す文字列がプログラム中で静的に与えられている場合には対応できるかもしれないが，必ずしもそうとは限らない．移送先で必要となるクラス群を移送時に静的に完全に挙げることは難しい．また，別の問題として，転送はしたものの，実際は移送先で利用されないといった無駄も生じ得る．

■動的な転送　動的な転送では，クラス名が実行時に与えられる場合に対応でき，無駄な転送も生じない．反面，必要になった時点で転送を始めるため，ネットワーク経由での転送が完了するまでプログラムは待たされ，実行性能は悪影響を受ける．ただし，プログラムが必要とするクラスを早めに予測するといった方法でこの問題を軽減できるかもしれない．

■静的, 動的転送の混合　実行性能への悪影響のない静的転送，クラス定義の不在を起こし得ない動的転送，それぞれの利点を活かすためには，両者を混合することが考えられる．いくつかのクラスについてはスレッド移送と同時に静的に転送しておき，移送先に存在しないクラスが必要となったら動的に転送を行う．

クラス名空間

クラス定義がファイルシステム上に置かれているクラスだけを利用している限りは，Java 仮想マシン上のすべてのスレッドは単一のクラス名空間を共有する．すなわち，ど

のスレッドにとっても同じ名前のクラスは同じ定義を持つ.

スレッド移送が可能な場合，移送の前後でスレッドから見えるクラス名空間が変わってはならない．つまり，移動前は利用できたクラスが移動後には存在しなくなったり，同じ名前のクラスの定義が移動後は違うものになったりしてはいけない.

スレッド移送システムは，スレッドごとに適切なクラス名空間を提供する必要がある．これは，実装としては Java 仮想マシンの一機能であるクラスローダで達成できる.

本研究のアプローチ

クラス定義に関するいくつかの選択肢と，それぞれの得失を挙げてきた．ここで，本研究での選択，および現実装について述べる.

■自動転送, 定義 利用者の手によるクラス定義の各コンピュータへのインストール，更新は，分散システムの透過性を著しく下げるため，これを避けた．クラス定義のスレッド生成元からの自動転送を実現している．現実装では移送先コンピュータの安全について配慮していないが，`SecurityManager` を用いた処理制限の実装は容易である．受け入れスレッドの認証，実行時間の制限は残された課題である.

■動的転送 クラス定義は，スレッド移送先での必要に応じて動的に，スレッド生成元から移送先へ転送される．現実装では静的な転送も可能であり，動的か静的かを利用者が選択できる．混合方式は未実装である．移送先で確実に必要になるクラスについてはスレッド移送に先だって静的に転送することで，通信遅延による実行性能の低下を抑えることを計画している.

■一貫性のあるクラス名空間 クラス名空間は，スレッド生成元のコンピュータごとに分離してある．移送スレッドが必要とするクラス定義は，そのスレッドの生成元から取得される．生成元コンピュータが同一であるスレッド群は同じクラス名空間を共有し，生成元が異なれば異なるクラス名空間を持つ.

これにより，別のコンピュータ上で生成されたスレッド群が同一の Java 仮想マシン上に存在してもクラス名とクラス定義の不整合が起きないことを保証している．つまり，生成元コンピュータが異なる 2 つのスレッドが同名で異なるクラス定義を要求した場合でも，双方の要求が満たされる.

2.4.3　スレッド外部化

スレッド移送は，スレッドの外部化，転送，内部化という手順で行われる (第 2.4.1 項)．ここでは外部化，内部化の方法を述べる．

外部化によって，移送だけでなく，スレッドをファイルやデータベースに保存することも可能となる．これは耐故障性の向上に応用できる．仮にコンピュータやネットワークの障害で実行が中断されても，保存したおいた時点から実行を継続できる．数日から数ヵ月といった長い間実行されるプログラムには，自らの実行状態を保存，復帰する機能を持たせることが一般的である．スレッド外部化を利用することで，このような，本来の処理ではない非本質的な開発作業は不要となる．

本項ではまず，外部化の手順を述べ，続いて，外部化に必要な，スタック上の基本型と参照の識別について述べる．最後に位置依存の資源の扱いを述べる．

状態

内部化後に実行を継続するためには以下の状態を外部化しなければならない．

- Java スレッドの持つ実行状態
  - レジスタ

    プログラムカウンタ, スタックポインタ
  - スタックフレーム

    ローカル変数, オペランドスタック, ···
- スレッドから辿り得るオブジェクト群

手順

これらの外部化，内部化の手順を述べる．

■排他制御, 一時停止　まず，外部化対象スレッドのモニタを取得して，実行を一時停止させる．外部化の途中でスレッドの状態が変化してしまうと，そのバイト列は内部化できないからである．

一時停止の前にスレッドオブジェクトのモニタを取得するのは，スレッドを排他的に扱

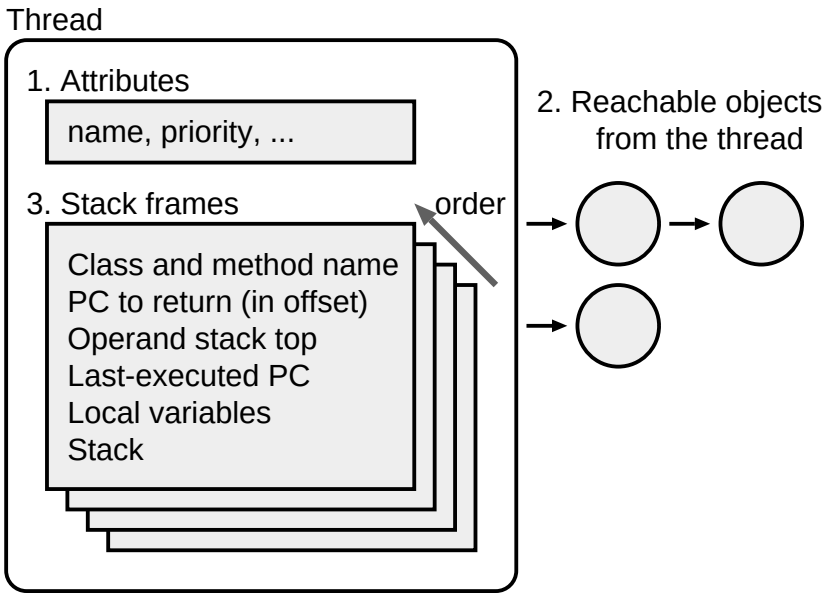

図 2.7: スレッドの外部化手順

うためである．しかし，モニタを取得しても，完全な排他は達成できない．スレッドの実行開始，停止にはモニタの取得が必要なので，モニタの取得によって他スレッドからの操作を抑制できる．しかし，一時停止，実行再開といったモニタを通さずに行える操作がある．また，`Thread` クラスの一時停止，実行再開用メソッドをオーバライドして，モニタを通じた操作としてしまうこともできない．それらは `final` メソッドとしてオーバライドが禁じられている．このように，現実装では外部化時の排他が完全ではない．完全な排他制御は残された課題である．

■自身の外部化のための前処理　外部化対象スレッドの実行は一時停止させねばならないので，自分自身，つまり現在実行中のスレッドは外部化できない．自スレッドの状態，少なくともプログラムカウンタが外部化の過程で必ず変化してしまうからである．自スレッドの外部化は，別にスレッドを生成してそのスレッドに任せる．外部化のためのスレッドを生成したら，自身は一時停止する．

■バイト列化　バイト列化の手順を示す (図 2.7)．もし実行状態を表現する標準的な形式があればそれに従ったが，そういった標準形式は存在しないため，独自の形式を定義した．

1. スレッドの属性

そのスレッドが生成された計算機の IP アドレス,

スレッドの名前，優先度，デーモンスレッドかどうか

2. スレッドから辿り得るオブジェクト群

   マーシャリング (第 2.4.4 項) でバイト列化

3. スレッドの状態

   メソッド呼び出しで作られるフレームの連鎖を，呼び出しが古い方から順に辿り，各フレームの次の属性をバイト列化

   (a) クラス，メソッド名

   (b) リターン先 (バイトコード命令列の先頭からのオフセット)

   (c) オペランドスタックの先頭位置

   (d) 最後に実行された命令のアドレス (オフセット)

   (e) ローカル変数

      メソッドが持つローカル変数表の各要素を辿り，各変数について次の属性をバイト列化

      i. その時点で該当変数が生存していれば変数の値

         基本データ型であれば即値,

         オブジェクトであればマーシャリング結果のバイト列

   (f) スタック

      オペランドスタック上の各要素を辿る

      i. 要素の値

クラス定義はメソッドごとにローカル変数表を持ち，それぞれの変数の有効期間をプログラムカウンタの範囲という形で保持している．これを参照することで，ある時点でそのローカル変数が生存しているかどうかを判定できる．

■スレッドの再構成　内部化では，外部化で生成されたバイト列を元に，外部化とは逆の手順でスレッドを再構成する．

まず，ヒープ上にスレッドのインスタンスを再構成する．これはイントロスペクション (第 2.4.5 項) の機能を用いて実装されている．

スレッドは実行状態を含む特殊なオブジェクトなので，ヒープ上のインスタンスを構成しただけでは内部化は済まない．少なくとも現在のソフトウェア実装の Java 仮想マシンでは，スレッドオブジェクトは，OS やその上のライブラリに実装されたシステムレベル

のスレッド (以下システムスレッド) と対応付けられている．Java 仮想マシンが利用するシステムレベルのスレッドライブラリは，OS が提供するものであったり，Sun 社が提供するユーザ空間の Green Threads であったりする．

インスタンスとしてのスレッドに続いては，対応するシステムスレッドを作成し，Java スレッドの実行を再開する．

基本型の値とオブジェクトの判定

ローカル変数の値やオペランドスタック上の要素は

- 整数，浮動小数点数，真偽値などの**即値**
- インスタンスへの**参照**

をとり得る．バイト列化の際，即値はネットワーク上表現に変換するだけで済むが，オブジェクトはマーシャル (marshal) する必要がある．従って，ローカル変数やスタック上の要素を外部化するときは，それが即値なのか参照なのかを判定しなければならない．

ローカル変数については Java 仮想マシン中の表に型情報があるので，それを参照すれば済む．問題はオペランドスタック上の要素である．少なくとも，Sun 社の Java 仮想マシンは，スタック上に要素の型情報はない．Sumatra [2] では，スタックを操作する際に値だけでなく型も積み降ろしすることで，常にスタック上要素の型を知ることができるようにしている．また，原理的には，メソッドのコードを先頭から辿ってスタックの状態を追跡するという方法で型を調べることもできる．しかし，この手法は辿るコードの長さに応じた時間がかかり，外部化への応用は現実的ではない．また，Sun で開発された Java 仮想マシン，Research VM [5] のように，あらかじめコード中の何ヶ所かについてスタック上要素の型を調べておく方法も考えられる [3][4]．Research VM は，スタック上の型情報を，保守的 (conservative)GC を避けて precise GC(または exact GC) を行うために用いているが，この型情報はスレッド移送にも応用できる．

現在の実装では，OS のメモリ保護機能を利用して即値と参照を識別している．まず，スタック上の 32bit 値を参照であると仮定して，それに対して参照のメモリ上の構造に従ってアクセスしていく．もしその 32bit 値が即値であればアクセス違反の Segmentation Violation シグナル (SIGSEGV) が発生する可能性があり，参照であれば発生し得ない．発生しなければ参照だと判定する．この手法で確実な判定ができない．即値の 32bit 値が

偶然ある参照のアドレスと一致していたら，それは参照であると判定されてしまう．しかし，参照のメモリ構造をよくアクセスすることで，現実的な精度での判定は可能である．誤判定の起こり得ない手法の実装は残された課題である．

位置依存の資源

別のコンピュータ上では意味を持たない，ネットワーク上の位置に依存する種類のインスタンスが存在する．例えばファイルを表す `File` クラスのインスタンスである．現実装は，これら位置依存のインスタンスをスレッド内部化の際に無効化している．詳細はマーシャリングについての第 2.4.4 項で述べる．

Just-in-Time コンパイラ対応

現在，スレッド外部化と Just-in-Time(JIT) コンパイラ双方の同時利用は実現できていない．JIT コンパイルとは Java バイトコードの実行時コンパイルを指す．Java 仮想マシンはこれによって高速実行を達成している (第 1.1.3 項)．

スレッドの外部表現が OS やプロセッサに依存しないためには，プロセッサの実行状態ではなく，Java 仮想マシンの実行状態を外部化しなければならない．実際，本研究では Java 仮想マシンの実行状態を移送の対象としており，例えば，プログラムカウンタ (PC) はプロセッサのそれではなく，バイトコード命令列を指すものを外部化している．また，これによって，異機種間移送を達成している．

通常の JIT コンパイラは異機種間移送と共存できない．JIT コンパイラが生成したネイティブコードの実行中は Java 仮想マシン上の実行状態を得ることができないためである．得られるのは，プロセッサ上の実行状態のみである．共存できない理由はそれだけではない．たいてい，Java バイトコードの一命令は，プロセッサの命令複数に対応する．スレッド移送の際に，その複数命令の途中でスレッドが一時停止されたとしたら，対応するバイトコード命令は未実行とも実行済みともつかない状態となる．これでは移送はできない．

JIT コンパイラと異機種間移送を両立させるためには，Java 仮想マシンと JIT コンパイラに次の機能が必要である．

1. JIT コンパイラが生成したネイティブコードの実行中でも Java 仮想マシン上の実行

状態を得られること．

2. Java 仮想マシンの状態が不整合とならない時点でスレッドを一時停止できること．

1 は Sun 社の Java 仮想マシンである HotSpot VM が実装している「on stack replacement」という機能で達成することが可能である．これは，あるメソッドをインタプリタが実行している最中に，JIT コンパイラが生成したコードの実行に切替えたり，その逆を行うための機能である．Java 仮想マシン上の実行状態と，プロセッサ上の実行状態を相互に変換できる．これによって HotSpot VM とその JIT コンパイラは，インタプリタで実行中のメソッドをコンパイルして，メソッド実行中にも関わらずコンパイル結果のネイティブコードの実行に移ることができる．HotSpot VM は別にスレッド移送のためにこの機能を用意しているわけではない．インタプリタでの実行中にそのメソッドが長時間実行されることが判明した際に，ネイティブコードの実行に移れることは性能のために重要なのである．

2 は，Research VM や HotSpot VM が実際に行っている．Research VM は precise GC のために行っていて，手法は文献 [3] で説明されている．HotSpot VM も，on stack replacement に必要となる機能なので，行っていると推測される．

2.4.4 マーシャリング

本システムではスレッド外部化の際にインスタンス群もバイト列に変換する．マーシャリング (marshaling) は，その，インスタンスのバイト列化を行う機能である．逆の変換をアンマーシャリング (unmarshaling) と呼ぶ．

■Sun 社の提供する Object Serialization Java の標準 API でも，Object Serialization[53] という同等の機能が提供されている．マーシャリングを独自に実装したのは，スレッド移送のために，標準 API が持つ制約を越える必要があったからである．例えば，標準 API では，インスタンスがバイト列化可能であることを，クラスを記述する時点で明示的に表明しておく必要がある (`implements Serializable`)．ところが，スレッド移送では任意のインスタンスをバイト列化する必要がある．標準 API の課す制約は一般的な利用においては妥当であるが，スレッド移送では越える必要があり，独自の実装が必要となった．

アルゴリズム

インスタンスは参照関係で任意のグラフを構成する．インスタンスのバイト列化とは，インスタンスのなすグラフのバイト列化である．アルゴリズムを次に示す．インスタンスのフィールドを順次走査し，基本型であれば定められた形式に変換し，参照であれば再帰的にバイト列化する．循環グラフを扱うために，バイト列化済みインスタンスの表を用意しておき，バイト列化対象のインスタンスが表に見つかった場合は，インスタンス自体をバイト列化する代わりに表中の位置，つまり添字を書き出す．

マーシャリングのアルゴリズムは，詳しくは次の通りである．

```
METHOD marshaling(対象オブジェクト O)
IF（O は marshal 済み）{
  marshal 済みの表中の index を出力
  RETURN
}
marshal 済みの表に登録
IF（O は配列）{
  配列の要素すべてを出力
}
IF（O のクラスが自前の外部化メソッドを持つ）{
  外部化メソッドを呼ぶ
  RETURN
}
FOR クラス C IN O のクラスのスーパークラスを根から順に {
  IF（クラス C が自前のマーシャリングメソッドを持つ）{
    マーシャリングメソッドを呼ぶ
    CONTINUE
  }
  FOR フィールド F IN O の C 中のフィールド {
    IF（F はオブジェクトか配列型）{
      CALL marshaling(F)
```

```
      }
      ELSE { // F は基本型
        F の値を出力
      }
    }
  }
```

ここで，自前の外部化メソッド，マーシャリングメソッドとは次のようなメソッドである．

- **マーシャリングメソッド**　自クラスのバイト列化を行う．スーパークラスには一切配慮しない．
- **外部化メソッド**　自クラスと，直接，間接のスーパークラスのバイト列化を行う．

マーシャリングの過程で，クラスが持つフィールドの型や属性，値を取得したり，アンマーシャリングの過程でフィールドに値を設定する必要がある．そのためにイントロスペクション (第 2.4.5 項) を利用する．

■プロトコルの頑強性　スレッド外部化 (第 2.4.3 項)，マーシャリングに関して，現在のプロトコルや内部化部は頑強さが充分ではない．プロトコルの頑強性とは，ストリームの様々な違法性を早く確実に検出しやすい性質を指す．処理部の頑強性とは，違法なストリームに対して誤動作しにくい性質を指す．

現実装は形式が違法なストリームを受け取った際に，検出できずにで誤動作する可能性がある．より頑強なプロトコルおよび処理部の構成は残された課題である．

データの外部表現

マーシャリングでは整数，浮動小数点数，真偽値など基本型の外部表現を読み書きできる必要がある．

32bit 整数を 8bit 単位で扱う場合を考えよう．8bit 要素の順によって 4 の階乗，つまり 24 通りの表現が考えられる．書き出した値をきちんと読み込むためには，書き出し側と読み込み側で何かしらの合意が必要となる．方針はふたつある [63]．

- プラットフォーム独立表現をひとつ定める
- 書き出し側と読み込み側で，どの表現を用いるかネゴシエーションする

前者はネゴシエーションが不要で，実装もシンプルになる．後者は外部表現の扱いをより高速に行える可能性がある．例えば，読み書き双方のプロセッサがリトルエンディアンであった場合に，プロセッサ上表現を外部表現としてそのまま採用することに双方が同意すれば，バイトオーダを変換する処理が不要になるので，性能的に有利である．

知られた外部データ表現はすでにいくつかある．

- Sun 社の Remote Procedure Call(RPC) ライブラリが利用している External Data Representation(XDR)
- Open Software Foundation(OSF) が採用した Netorking Computing System(NCS) の Network Data Representation(NDR)
- Xerox の Courier
- ISO 標準の Open Systems Interconnection(OSI) Abstract Syntax Notation One(ASN.1), Basic Encoding Rules(BER) and Distinguished Encoding Rules(DER)

NCS の NDR では，書き出し側がデータ表現，つまりバイトオーダを決定，プロトコルによって読み込みがわにデータ表現を伝える．他はすべて，単一の，プラットフォーム独立外部表現を定めている．

本システムのマーシャリングでは各基本型にプラットフォーム独立表現を定めている．Sun XDR にならっている．

- ネットワークバイトオーダ (MSB first) に揃える．
- 浮動小数点数の表現は IEEE754 floating-point standard [29] 準拠．

位置依存の資源

別のコンピュータ上では意味を持たない，ネットワーク上の位置に依存する種類のインスタンスが存在する．例えばファイルを表す `File` クラス，ソケットを表す `Socket` クラスなど，特定のコンピュータ上の資源と結びつけられたインスタンスである．

マーシャリングの際，これらに配慮しないと，アンマーシャルしたインスタンスが意図せぬ資源と結び付いて不正な操作をしてしまう，といった問題が起こり得る．これら位置依存のオブジェクトに対しては，内部化の際に無効化または補整処理を施せば，こういっ

た問題は防げる．

本システムが提供するクラスであれば，クラス定義自体に，無効化，補整処理を組み込むことが可能である．ところが，標準 API が含むクラスに新たな処理は追加できない．標準 API が含む位置依存クラスについては，マーシャル，アンマーシャルの際にチェックし，無効化や補整処理を施すという方法を採っている．

無効化や補整が不要な場合もある．マーシャル後，同じスレッドがそれをアンマーシャルする場合である．ただし，ある資源に対応するインスタンスの存在がひとつしか許されない場合は，アンマーシャル時の補整処理は避けられない．

位置依存性に着目すると，クラスを次の区分に分類できる．

- 位置依存
  - 同一スレッドによるマーシャル，アンマーシャルでは特別扱い不要
  - 常に無効化または補整処理が必要
- 位置非依存

標準 API に含まれないクラスであれば，無効化，補整処理はクラスを記述するプログラマが責任を負うべきである．そのクラスが上記のどの分類に属するかはプログラマのみが知っているからである．標準 API 中のクラスについては，すべて本システム側で無効化，補整処理を施す必要がある．現在の実装は，標準 API 中のいくつかのクラスについて無効化が可能となっている．

ファイルを表すインスタンスは移送先では無効化するしかないが，ソケットの場合，プログラムによっては移送後に再接続することでそのソケットを使った処理を続けることができるだろう．この再接続処理はプログラマに記述させる方法も考えられるが，システム側で通常の `Socket`，`ServerSocket` クラスの代替を用意しておくことで，プログラマがいちいち記述せずとも再接続するようにできるだろう．こういった，補整処理で継続しての使用が可能なクラスについては，代替クラスの提供がプログラミングの容易さの点で有効である．この用意もまた，残された課題である．

2.4.5 イントロスペクション

本システムはイントロスペクションという名前で，Java の標準 API のリフレクション (reflection)API と同様の機能を提供している．リフレクションとは一般には

計算システムが，自分自身の構成や計算過程・計算方式に関する計算を行うこと

を指すが，リフレクション API の機能はこの一部で，インスタンスが自身のフィールドや自クラスのメソッドについて属性や値を取得，設定する機能が提供されている．

本システムと標準 API の違いは次の通りである．標準 API だけが提供している機能は

- メソッド，コンストラクタの呼び出し

であり，本システムでのみ提供いている機能は

- クラス定義中の定数表 (constant pool) へのアクセス

である．

イントロスペクションの機能はオブジェクトのマーシャリングに利用されている．マーシャリングの過程で，クラスが持つフィールドの型や属性，値を取得したり，アンマーシャリングの過程でフィールドに値を設定する必要があり，これらをイントロスペクションで実現している．

標準 API に含まれるリフレクション API を利用せずに独自に実装したのは，マーシャリングと同様に (第 2.4.4 項)，標準 API が持つ制約を越える必要があったからである．スレッド移送では任意のオブジェクトをバイト列化する必要がある．リフレクションを利用しようが，本来アクセスできないフィールドには当然アクセスできない．標準 API Reflection の課すこの制約は一般的な利用においては妥当であるが，スレッド移送では越える必要があり，独自実装が必要となった．

2.4.6　クラス外部化

これはスレッド移送に必要な機能ではない．ここでクラス外部化 (class exportation) と呼ぶのは，クラス定義，つまりクラスファイルを，Java 仮想マシン内の情報から再構成する機能である．クラス定義の転送処理を高速化する目的に利用している．

クラスファイルはファイルシステム上に存在するため，ファイルから読み込むことができるのだが，Java 仮想マシン内から再構成することでファイルアクセスが不要となり，高速化される．

手法

イントロスペクション (第 2.4.5 項) を応用して，Java 仮想マシン内からクラスに関する情報を得てバイト列化，クラスファイルを復元する．この機能のために，イントロスペクション側にクラスの定数表 (constant table) へアクセスする機能を用意した．

課題

■quick 命令 Sun 社の Java 仮想マシンでは，実行中にバイトコード命令を書き換えるという最適化を行っている．Sun はこの手法の特許を取得している．一度実行された命令がある条件を満たした場合，“**<命令名>**_quick” という別の命令 (以下 quick 命令) に変換する．これによって，一度目の実行の際にのみ行う必要がある処理を二度目以降の実行では省くことができる．例えば，クラスの初期化などの処理である．

具体的には，定数表中から即値，オブジェクトを取り出す `ldc` 命令，インスタンスのフィールドを取得する `getfield` 命令，メソッド呼び出しの `invokevirtual` 命令，オブジェクトを生成する `new` 命令などが quick 命令に変換される．

現在のクラス外部化機能は quick 命令に対応していない．quick 命令をそのままクラスファイル中に含めてしまう．初回の実行から quick 命令となっていると，次の理由で Java 仮想マシンが異常動作することがある．

- 一度目の実行では必ず必要な処理がなされない．
- quick 命令生成時の Java 仮想マシンの状態に依存したオペランドで不正な処理を行う．

quick 命令はその時点の Java 仮想マシンの状態に依存したオペランドを取る．例えばメソッド呼び出しの quick 命令では，Java 仮想マシン内のメソッド表中の添字が引数として書き込まれる．メソッド表の構成は，クラス定義の変更などを理由に実行毎に変化し得る．

このように，quick 命令は Java 仮想マシンの外に出すべきものではない．クラス外部化の際は，quick 命令は通常命令に変換する必要がある．quick 命令から通常命令への変換は残された課題である．

2.5 応用

2.5.1 移動エージェントの記述

エージェントという語の定義は文脈に依存する．ここでは移動エージェントを，コンピュータ間を移動可能な処理主体，と定義する．

既存システム

移動エージェントの記述を目的としたソフトウェアは多数あるが (表 2.5)，これらは実行状態を移送できない．移動エージェントを表すインスタンスを移動させ，その後，そのインスタンスに対してメソッドを呼び出すことで実行を続ける方式を採っている．

Aglets Software Development Kit [28]　日本 IBM 東京基礎研究所で開発された，移動エージェントのプログラミング環境
Kafka [18]　富士通研究所によるマルチ・エージェント記述ライブラリ
Voyager [41]　ObjectSpace 社の分散システム開発基盤
Odyssey [26]　General Magic 社の移動エージェント

表 2.5: 既存の移動エージェント

移動によって，それまで実行していたメソッドとは別のメソッドに制御が移るため，プログラマはそれを意識してコードを書く必要がある．その結果，単一の Java 仮想マシン上で動作する移動しないプログラムとは異なる作法が要求される．プログラミングインタフェースの透過性が低い．

MOBA

本システム MOBA の移送スレッドは移動エージェントとみなすことができる．スレッドは実行状態を含めて移動し，移動先で実行を継続できる．このため，移動エージェントと移動しない通常のプログラムの違いは小さく，既存システムのように移動させるために特殊な作法に従う必要がない．

プログラミングインタフェースの比較

MOBA のプログラミングインタフェースの透過性を示すために，移動エージェントの記述法を MOBA と Aglets とで比較する．

■Aglets Aglet クラスのサブクラスとして記述する．移動は，移動先と，移動先で呼び出したいメソッド名を引数に go() メソッドを呼ぶことで行う．

```
public class MobileAgent extends Aglet {
  SimpleItinerary itinerary = null;

  public void onCreation(Object init) {
    itinerary = new SimpleItinerary(this);
    itinerary.go("atp://destination", "callback");
  }

  public void callback(Message msg) {
    System.out.println("arrived.");
  }
}
```

オブジェクトの移動と，移動したオブジェクトに対するメッセージ送信である．移動のために go() の呼び出すと，移動後は go() の引数として与えたコールバックメソッド callback() に制御が移る．

■MOBA 通常のスレッドの記述では Thread クラスを使用するところ，MobaThread クラスを使用する．それ以外は通常のスレッドと同様に記述する．

次のようにスレッドを生成し

```
MobaThread t = new MobaThread(...);
t.start();
```

次のようにスレッド自身で移動する．

```
...
MobaThread.goTo(移動先計算機);
...
```

2.5.2 プログラムの耐故障性向上

スレッド外部化 (第 2.4.3 節) を利用して，実行中のスレッドをファイルやデータベースに保存することが可能である．適当なタイミングで実行中の状態を保存しておけば，コンピュータの停止や故障時にも保存時の状態から実行を継続できる．また，もし緊急の電源断などコンピュータの停止や故障をあらかじめ検知することが可能ならば，先だって状態を保存したり，他のコンピュータ上へ移動することで，それまでの処理結果を失わずに済む．

実行に数週間以上かかるプログラムには，不慮の事故に備えて，実行途中の状態を保存する仕掛け，保存した状態から復帰する仕掛けを用意することが一般的である．スレッド外部化を利用することで，このような非本質的な部分に労力を割く必要がなくなる．

■課題　プログラムにとっての異常事態のうち，OS のシャットダウンなど，あらかじめ検知することが原理的に可能な事象も多い．ところが，Java のプログラムからそういった事態を検知する手段はない．C 言語ではある種のシグナルをとらえる方法はあるものの，標準的ではないので，多くの環境で広く使える手法ではない．

Java のプログラムから Java 仮想マシンの異常終了を検知する方法を検討している．将来は Java 言語の仕様の一部として提供されることが望ましい．

2.5.3 負荷分散

従来より研究されてきたプロセス移送の主な応用である．負荷分散の手段がプロセス移送だけというのは効率が悪いが，タスクの配置 (task placement) で静的負荷分散を行いつつ，プロセス移送を動的負荷分散利用することでさらなる性能向上を図り得る．

■課題　動的負荷分散のためには，いつ，どこからどこへ移送を行うかをスケジュールせねばならない．このスケジューリングは本研究では対象としていないが，効果的なスケジュールのためには移送システムからの情報提供も欠かせない．その点，現状の本システムや Java 仮想マシン自体には足りないものが多い．

- コンピュータの能力，負荷の表現，取得法.
  様々な側面を持つ計算能力の表現法，およびその取得，見積り法.
- 負荷分散のポリシー.
  能力，負荷情報から移送対象，移送先を決めるポリシー，アルゴリズム.

2.5.4 遠隔メソッド呼び出しのエミュレーション

スレッド移送を用いて，ORB で実装されている遠隔メソッド呼び出しと同等の処理が可能である．呼び出し先コンピュータへ移動し，目的の処理を行い，元のコンピュータへ移動して戻ればよい.

ただし，意味的に同等の処理とはいえ，一般にスレッド移送は遠隔メソッド呼び出しより重い処理である．遠隔メソッド呼び出しが可能な場合に，あえてそれをスレッド移送でエミュレートする理由はない.

2.6 性能評価

本システムの目的は，メタコンピュータを構成するために実行状態の移送を達成することである．しかし，移送ができればそれでよいというものではなく，性能が問われる局面も多い.

代表的な応用である動的負荷分散 (第 2.5.3 項) の目的は，そもそも，コンピュータが暇な時間を減らすことで分散処理全体の時間を短縮することである．いつ，どこからどこへ移送するか，効果的にスケジュールするためには，移送にかかる時間の見積りが重要である．移送に長い時間がかかると，移送による負荷分散の効果を相殺してしまいかねない．効果的な負荷分散のためには，素早く移送できることが求められる.

実装した移送システム MOBA について，以下を評価した.

- 移送の遅延
- 転送のスループット

実験は，2 台のコンピュータを 100Mbps のイーサネットで接続して行った．1 台は UltraSPARC-II 167 MHz を搭載，もう 1 台は UltraSPARC-II 296 MHz を搭載したコンピュータで，どちらも SunOS 5 で動作している．以下，前者をコンピュータ 1，後者

をコンピュータ 2 と呼ぶ.

MOBA 以外のシステムでは，Java Development Kit(JDK) 1.1.7 と，それに含まれる Research VM および JIT コンパイラを利用した．一方 MOBA では，JDK 1.1.8 の参照実装，つまり classic VM を利用し，JIT コンパイラは用ずにインタプリタを用いた．これは，MOBA が classic VM を必要とすることと，また，いまだスレッド移送に必要な機能を提供している JIT コンパイラがないことが理由である.

2.6.1 遅延

MOBA と Voyager [41] について，移送にかかる時間を計測した．プログラムは，コンピュータ 1 と 2 の間を決められた回数往復して，移送 1 回あたりの移送にかかる時間を計測するという単純なものである．そのコードは MOBA では以下のようなものであり，

```
開始時刻を取得
for (i = 0; i < 往復の回数; i++) {
  コンピュータ 1 へ移動
  コンピュータ 2 へ戻る
}
終了時刻を取得
```

Voyager では次のようなコードとなる.

```
開始時刻を取得
for (i = 0; i < 往復の回数; i++) {
  対象インスタンスをコンピュータ 2 へ移送
  対象インスタンスをコンピュータ 1 へ移送して戻す
}
終了時刻を取得
```

表 2.6 に，それぞれのシステムで 1 回の移送にかかった時間を挙げる．1 回の往復にかかる時間は MOBA の方が短いが，往復の回数を増やしていくと，1 回あたりにかかる時間は Voyager の方が短くなっていく.

Voyager ではインスタンスが移送されるのに対し，MOBA ではそれに加えてスレッドの実行状態も移送される．この点で原理的には MOBA が不利である．実行状態を移送で

往復の回数	1	10	20	50	
MOBA	191.0	109.3	105.53	105.32	
Voyager	292.5	57.05	44.00	37.08	(ミリ秒)

表 2.6: 移送にかかった時間

きることの利点と，性能面での代償をどう評価するかは，応用や目的に依ろう．

とはいえ，これらの通信を行うソフトウェアでは，細かい実装の違いや通信に関するパラメタの調整が性能に大きく影響を与える．例えば，バッファの利用でスループットが十倍変化したり，逆にそれによって遅延が増大したりする．こういった実装，調整の違いが結果に与えている影響の大きさは，今回も正確に評価できていない．原理，アルゴリズムの比較というよりも，個々の実装の比較になっているというのが現状である．

2.6.2 スループット

移送対象のスレッドが大きなデータを持つ場合，移送時のスループットが移送時間に大きく影響を与える．ここでは，分散処理の基盤ソフトウェアを比較するという意味で，Object Request Broker(ORB) と MOBA を比較した．

ORB としては Java RMI [59] と HORB [27] を用い，次のようなコードで転送のスループットを計測した．

```
// 準備
double[] argument = new double[配列のサイズ];
remoteRef = 遠隔参照;

// 計測
開始時刻を取得
remoteRef.aMethod(argument);     // 遠隔呼び出し
終了時刻を取得
```

ここで，`aMethod()` は，64bit 浮動少数点数の配列 (`double[]`) を引数にとり，返り値を返さないメソッドである．メソッドの中身は空で何も行わない．これを呼び出すことで，引数を呼び出し先コンピュータまで転送するのにかかる時間を計測できる．一方 MOBA

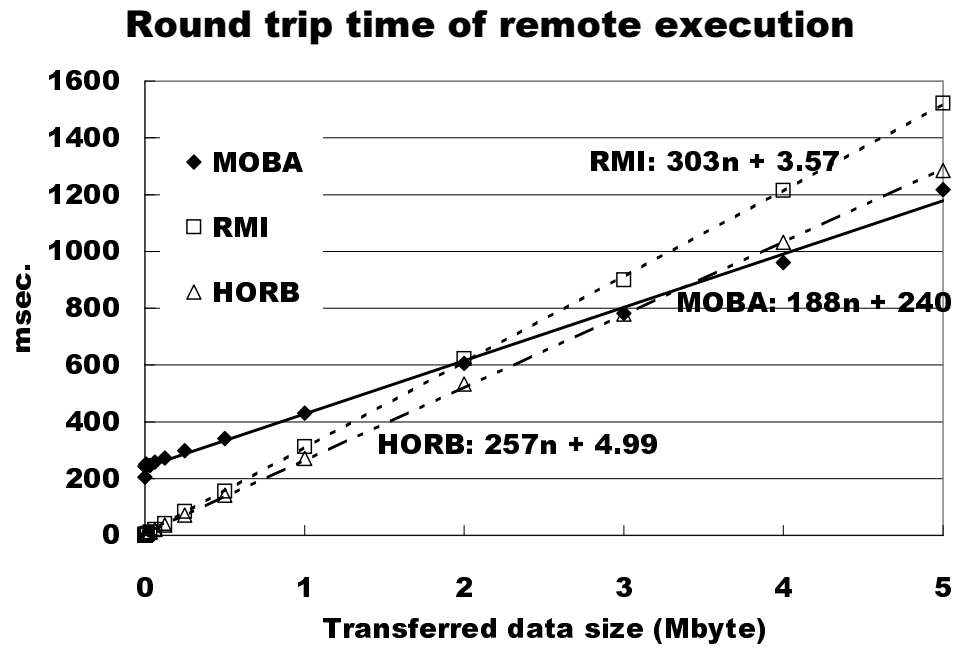

図 2.8: 移送にかかった時間

では次のようなコードで，遠隔呼び出しをエミュレートした．

```
// 準備
double[] argument = new double[配列のサイズ];

// 計測
開始時刻を取得
移動
argument = null;          // 引数を捨てる
元のコンピュータに移動して戻る
終了時刻を取得
```

計測結果を図 2.8 に，また，結果を元に算出したスループットを図 2.9 に示す．ORB では，転送されるのは引数と呼び出し要求であり，引数以外のサイズはたかが知れているが，MOBA ではスレッドの移送に時間がかかっている．転送量が多い場合は MOBA の結果が良く，スループットの高さがあらわれている．

この遠隔呼び出し，またはそのエミュレーションにかかった時間 (ミリ秒) を一次式で近似すると次の式が得られる．

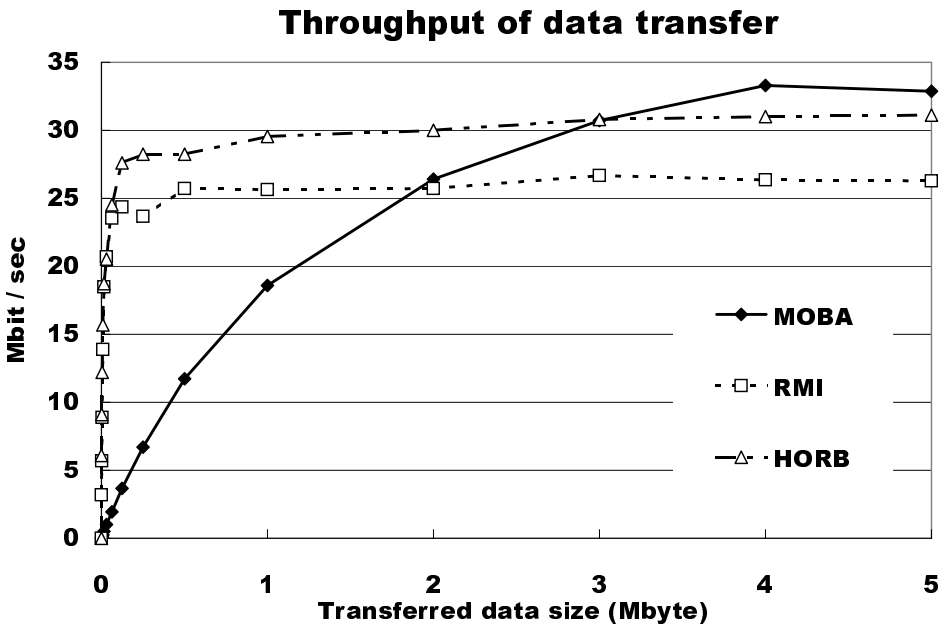

図 2.9: 移送のスループット

$$
\begin{array}{lrcr}
\text{MOBA:} & 188n & + & 240 \\
\text{HORB:} & 257n & + & 4.99 \\
\text{RMI:} & 303n & + & 3.57
\end{array}
$$

ここで n はメガバイト単位の転送データサイズである．ここで，定数は遠隔呼び出しごとのオーバヘッド，n の係数は 1 メガバイトを転送するためにかかる時間 (ミリ秒) と考えられる．

MOBA は引数だけでなく実行状態も移送しているため，オーバヘッドが大きい．このように，単に遠隔呼び出しをエミュレートする目的にスレッド移送を用いるのは効率が悪い．しかし，遠隔にあるディスク等の資源に頻繁にアクセスしつつ処理を進める場合は，スレッド移送が有利になり得る．遠隔呼び出しではアクセスのたびに通信遅延によるオーバヘッドがかさむのに対し，スレッド移送でアクセス先に移動してしまうことで，アクセスごとに通信遅延の影響を受けることはなくなる．

2.7 むすび

この章では，Java 仮想マシン間でのスレッド移送手法とそれを実装したシステムについて述べた．メタコンピュータの構成という目的には欠かせない異機種間移送と非同期移送を Java 仮想マシンの上で行えることを実証し，また，性能上重要な JIT コンパイラと

の共存手法を示した．JIT コンパイラとの共存以外にも，スレッド移送機能を設計，実装する際の一般的な問題を明らかにし，考えられる手法と本研究の手法を説明した．問題とはすなわち，移送すべきインスタンスの選択，位置依存資源の扱い，スタック上要素の型識別である．

メタコンピュータの構成には，これらの問題に対するより良い解決手法および分散オブジェクト (第 3 章) との統合に取り組むべきであろう．

第 3 章

分散オブジェクト

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．本章では，メタコンピュータの構成要素として構築した，ネットワーク透過な分散オブジェクトシステムについて述べる．本システムを用いることで，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にコンピュータ群をほぼ単一コンピュータ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達成された．

3.1 まえがき

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コンピュータ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 [40][52][41] [59][39] には，別コンピュータ上にあるオブジェクト（遠隔オブジェクト）を同一コンピュータ上のオブジェクト（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．

本研究では，これら非透過性の多くを解消した，MetaVM と呼ばれる Java 仮想マシン上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，コンピュータ群をほぼ単一コンピュータ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 [40][52][59] 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．

本研究では，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し，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 [37] 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．

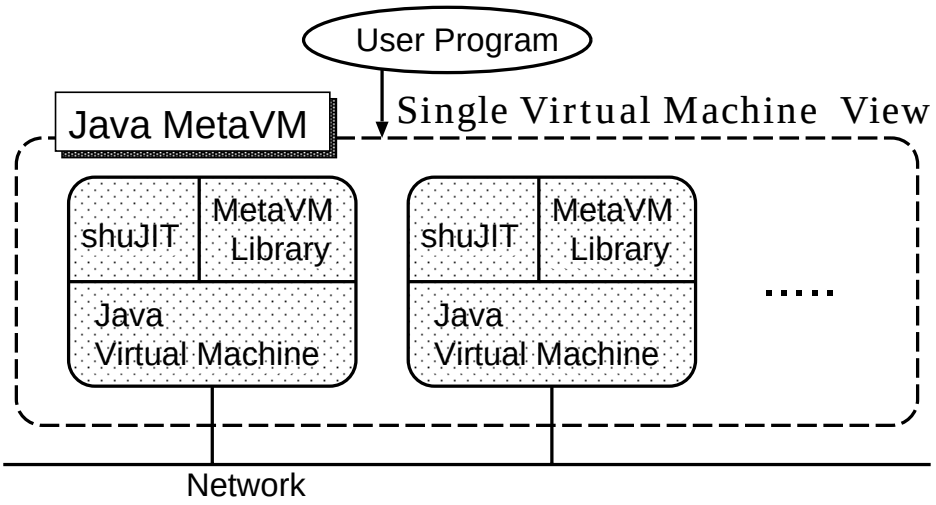

図 3.1: MetaVM の構成

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 [62]．本研究では，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．

本省では，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．

3.2 MetaVM の概要

ネットワーク接続されたコンピュータ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，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 3.1 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 [47] 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 3.5 節で述べる．

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．遠隔操作のためには，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 3.4 節で，通常の実行時コンパイラとしての手法は第 4 章で述べる．

Java **仮想マシン**　shuJIT が対応しているもの，つまり，Sun Microsystems 社の classic VM を利用する．これは，一般向けに配布されている Java Development Kit（JDK）や Java Runtime Environment（JRE）に含まれている．

3.2.1 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．メソッド呼び出し，フィールドアクセスなどの操作を `obj` に施した場合，MetaVM がその操作を遠隔に転送する．そのため，プログラマはオブジェクトが遠隔にあるのかローカルにあるのかを意識する必要はない．

以下，プログラミングインタフェースを説明する．

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.*;
```

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実行され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．

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.addressOf(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．

クラス Registry　プログラマは，オブジェクトへの参照にキーとなる文字列を結びつけて Registry に登録できる．登録されているオブジェクトをキーを元に検索，参照を取得できる．

3.3 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，ネットワーク上のコンピュータ群をほぼ単一コンピュータとみなせるという，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．この節

では，透過であるとは何を指すのかを，既存の C++言語，Java 言語用分散オブジェクトシステム [40][52][41] [59][39] と比較して述べる．また，いくつか残っている，単一コンピュータとの違いにも言及する．

3.3.1　MetaVM の提供する透過性

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 [40] では Interface Definition Language （IDL）で，RMI [59] 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 [27] 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 [41]，RyORB [39] 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．

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．

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．

引数渡しのセマンティクス

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，参照を値渡しすることが基本である．MetaVM ではプログラマの指示（第 3.2.1 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（第 3.3.2 項）．

既存の Java 用システムでは，遠隔に渡されるのは基本的にオブジェクトのコピーである．ローカルな呼び出しでは参照が渡されるので，ローカル呼び出しと遠隔呼び出しで結果や副作用が異なり得る．

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，コピーではなく参照を渡せるシステムもある．しかし，それらのシステムでは配列オブジェクトを遠隔参照できないので（第 3.3.1 項），配列はコピーを渡さざるを得ない．また，そもそもこの方式では，他人が設計して自分は変更を加えられないクラスはコピー渡しせざるを得ないという限界もある．

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．

3.3.2 非透過性 — 単一コンピュータとの差異

MetaVM の目標はコンピュータ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であるが，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．

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．

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．MetaVM では，遠隔参照への操作がバイトコードから行われる限りは問題は起きないが，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，既にあるネイティブメソッドは遠隔参照を考慮していないため，注意が要る．

具体的には，遠隔参照が引数としてネイティブメソッドに渡る場合に問題が起き得る．レシーバが遠隔参照である場合，つまり，遠隔参照に対してネイティブメソッドを呼び出すことには何ら問題がない．その呼び出し要求は遠隔参照が指すオブジェクトへとネットワーク経由で中継されるため，ネイティブメソッドにレシーバへの参照が渡ることはない．

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．

引数渡しのセマンティクスの例外

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．

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．

3.4 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 [47] 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 3.3.2 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 3.6.2 節で示す．また，動的に判定するこの手法の是非を，第 3.7.1 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 3.6.3 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

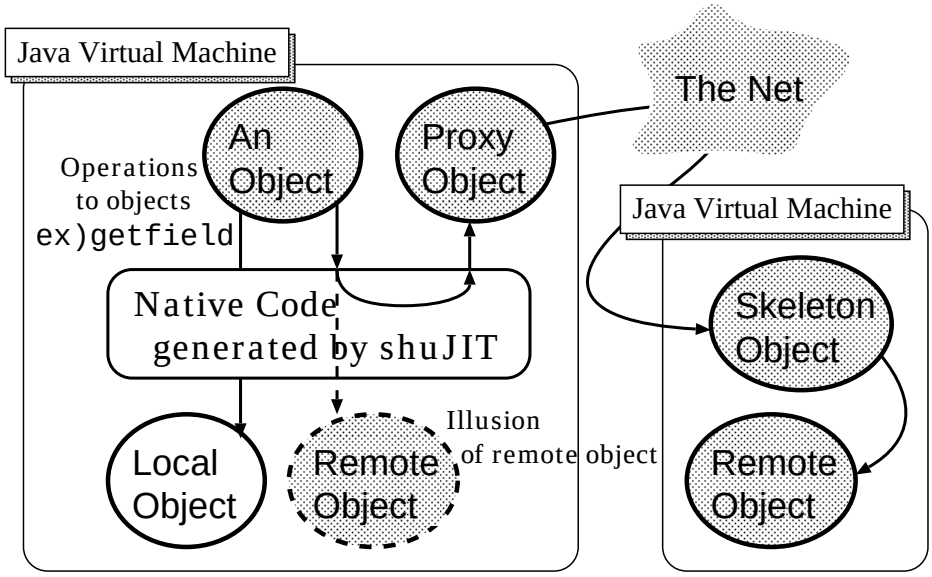

図 3.2: 遠隔操作の中継

`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．

3.5　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スやメソッド（第 3.2.1 節）を含む．

3.5.1　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

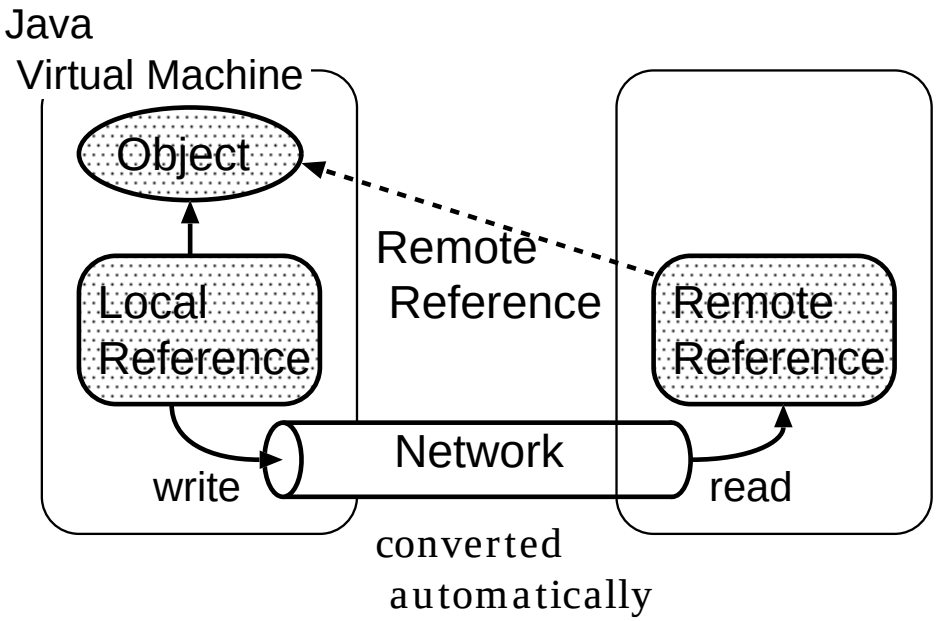

図 3.3: ネットワークを経由した参照の受渡し

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 3.2）．

3.5.2 参照の受け渡し

プログラムが分散実行される際も単一コンピュータ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 3.3.2 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 3.3）．

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 [39][41] 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

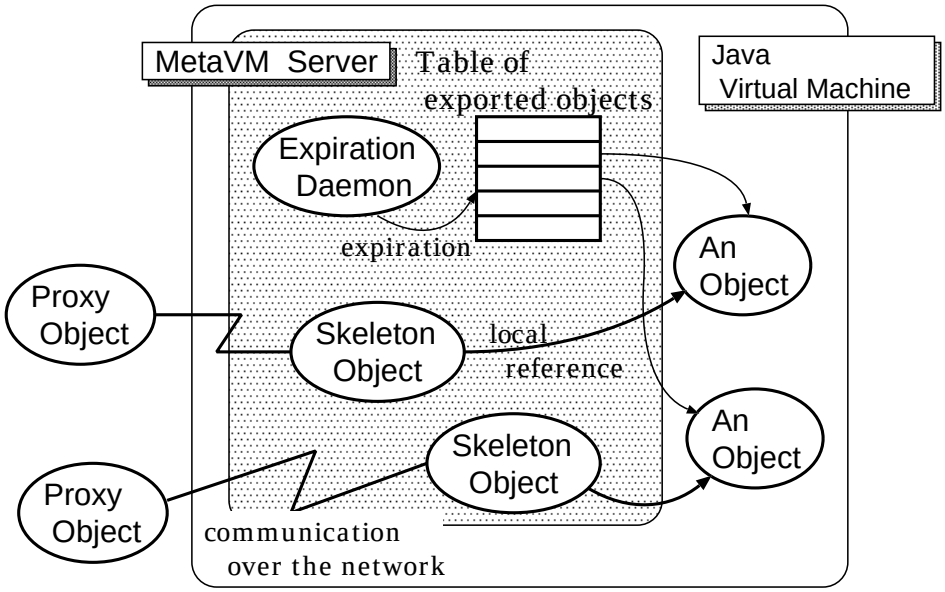

図 3.4: MetaVM サーバ

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．

3.5.3　MetaVM サーバ

MetaVM サーバ（図 3.4）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コンピュータ上であらかじめ 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 3.2.1 節）の最初の呼び出し時に，自動的に起動される（図 3.5）．

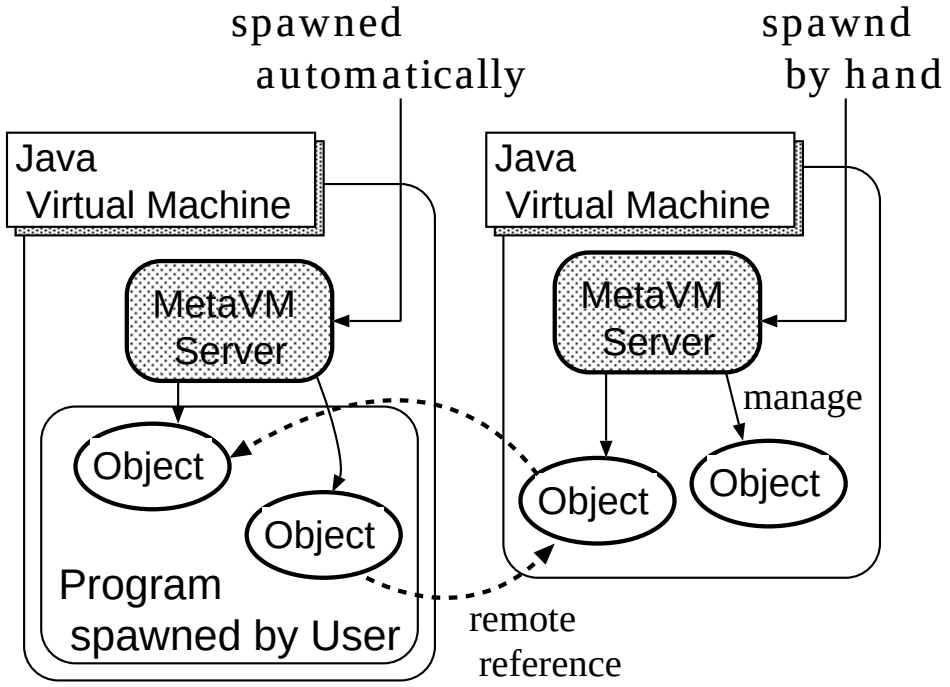

図 3.5: 自動的に起動される MetaVM サーバ

3.6 性能評価

性能評価の結果を示す．実験には次のコンピュータを用いた．

PC1 Pentium III 600 MHz, Linux 2.4.0, JDK 1.2.2

PC2 K6-2 400 MHz, Linux 2.2.16, JDK 1.2.2

1 台での実験には PC1 を用い，2 台での実験では PC1 から PC2 に対して遠隔操作を行った．ネットワークは 100 Mbps Ethernet(100base-TX) であり，2 台のコンピュータは 1 台のスイッチングハブを介して接続した．

すべての実験で JIT コンパイラとして shuJIT を用いた．MetaVM では MetaVM 対応機能を有効にしてコンパイルした shuJIT を用い，MetaVM 以外のシステムでは MetaVM 対応機能を無効にした，つまり非分散版の shuJIT を用いた．

MetaVM は 2001 年 1 月 15 日版，RyORB は 1999 年 3 月 8 日版，Voyager は 2.0.2，HORB は 1.3 beta4，RMI は JDK 1.2.2 に付属のものを使用した．

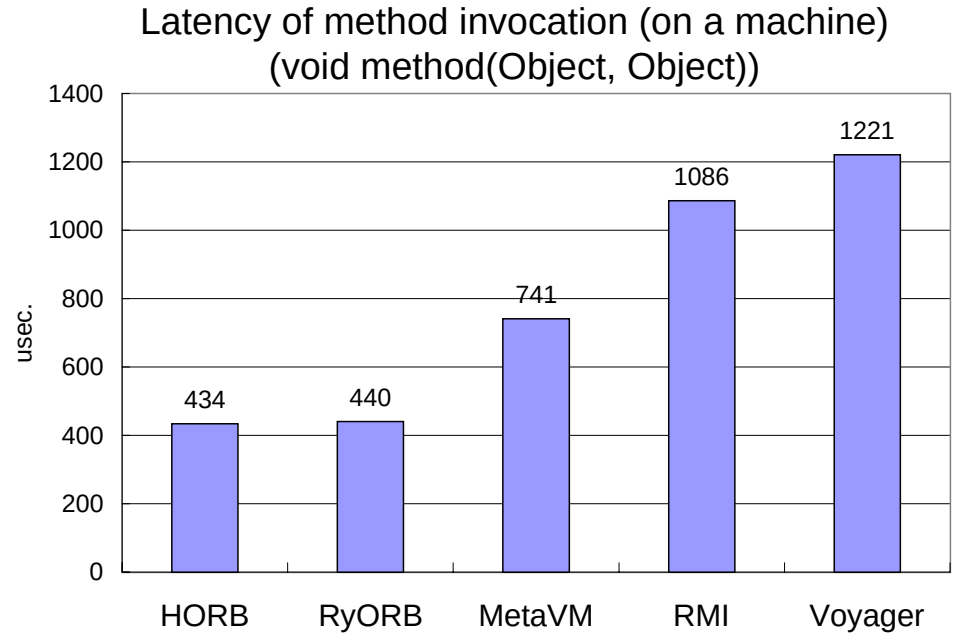

図 3.6: 遠隔メソッド呼び出しの遅延 (1 台上)

3.6.1 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 台のコンピュータ上での実験と，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．

メソッド呼び出し

図 3.6，3.7 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 3.6 がコンピュータ 1 台上での結果，図 3.7 がネットワークを介した 2 台での結果である．

HORB 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

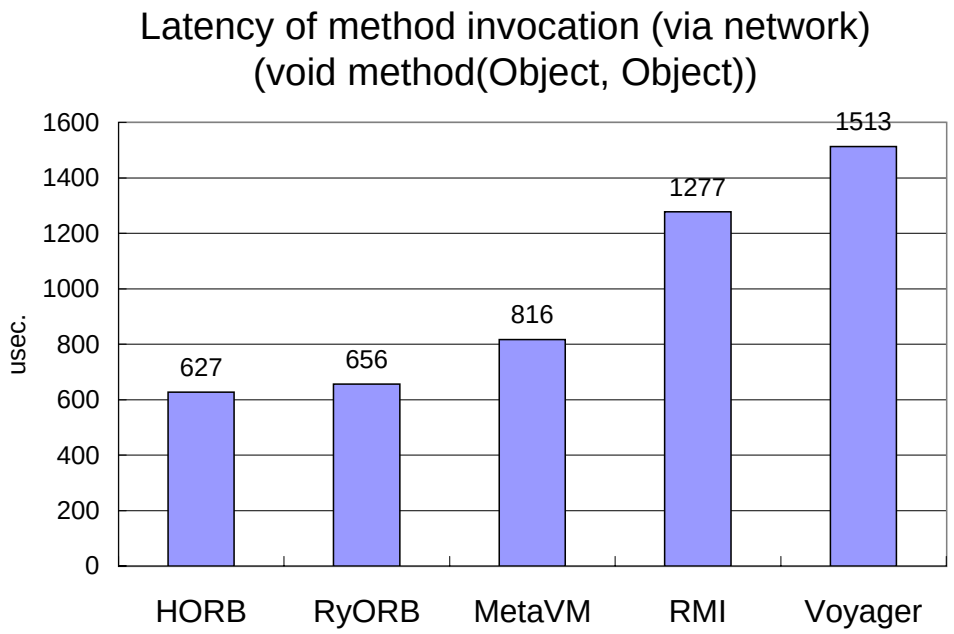

図 3.7: 遠隔メソッド呼び出しの遅延 (ネットワーク経由)

ることがわかる．RyORB と HORB には及んでいないものの，RMI と Voyager よりは速い．

フィールドアクセス

図 3.8，3.9 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．

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 側コンピュータの処理性能が 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 3.1 に，read 操作でのこの性能向上比を示す．MetaVM では結果が向上し，他のシステムでは低下している．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．

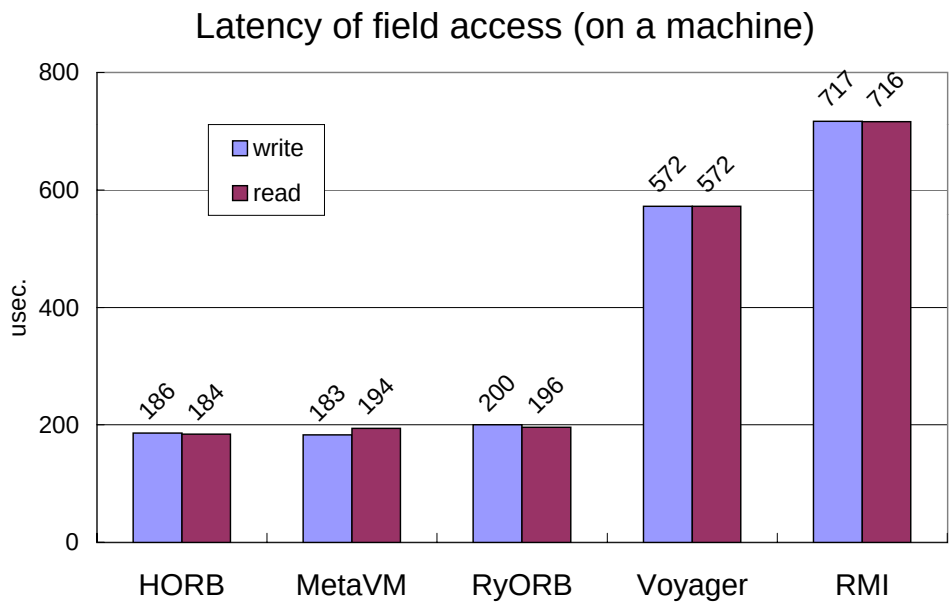

図 3.8: 遠隔フィールドアクセスの遅延 (1 台上)

- caller，callee の処理のオーバラップ量の大きさ．

MetaVM は，caller 側と callee 側の処理が極力オーバラップするように実装した．つまり，何かを送受信する際にタイミングを調整できる場合，送信はなるべく先に，受信は逆に後で行うようにしてある．2 台での実験では，1 台上での実験では必要のなかった，

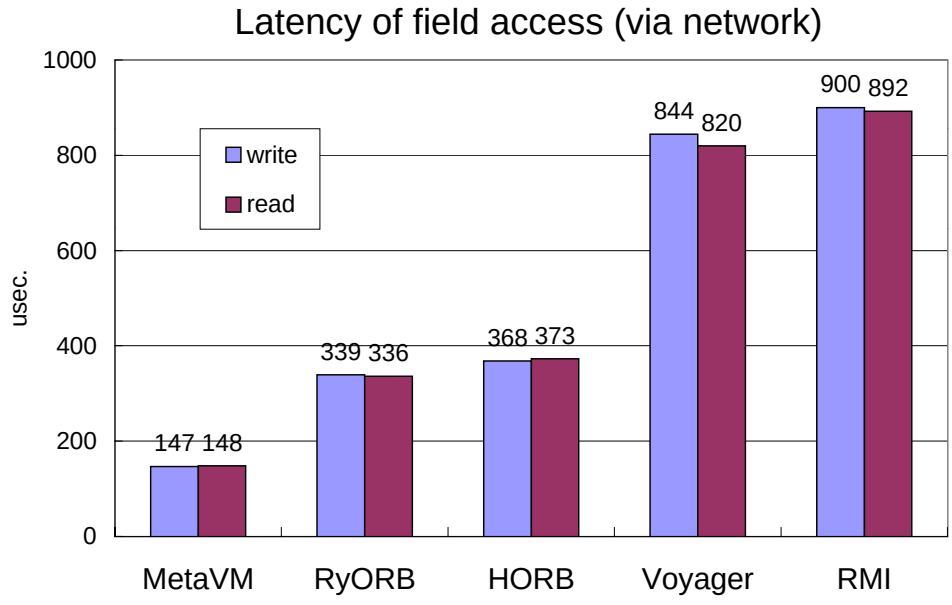

図 3.9: 遠隔フィールドアクセスの遅延 (ネットワーク経由)

システム	性能向上比
MetaVM	1.28
RMI	0.80
Voyager	0.69
RyORB	0.60
HORB	0.50

表 3.1: 2 台での性能向上比

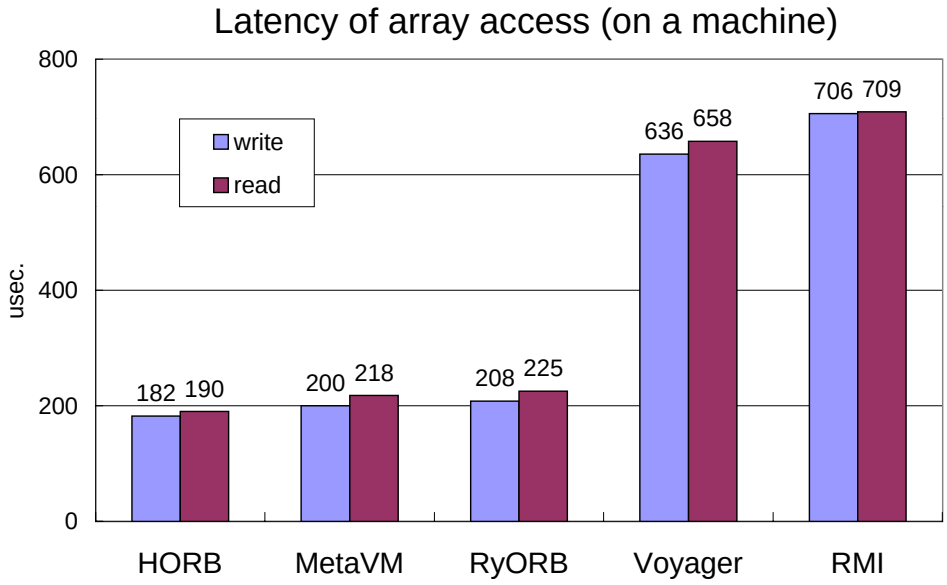

図 3.10: 遠隔配列アクセスの遅延 (1 台上)

ネットワーク経由の通信が起こる．それにも関わらず，MetaVM では結果が向上しているのは，送受信の処理がよくオーバラップできていることのあらわれである．

配列アクセス

図 3.10，3.11 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機能を利用したが，他のシステムはその機能を持たない．そのため，配列アクセスのみを目的としたメソッドを遠隔呼び出しすることで，遠隔配列アクセスをエミュレートした．

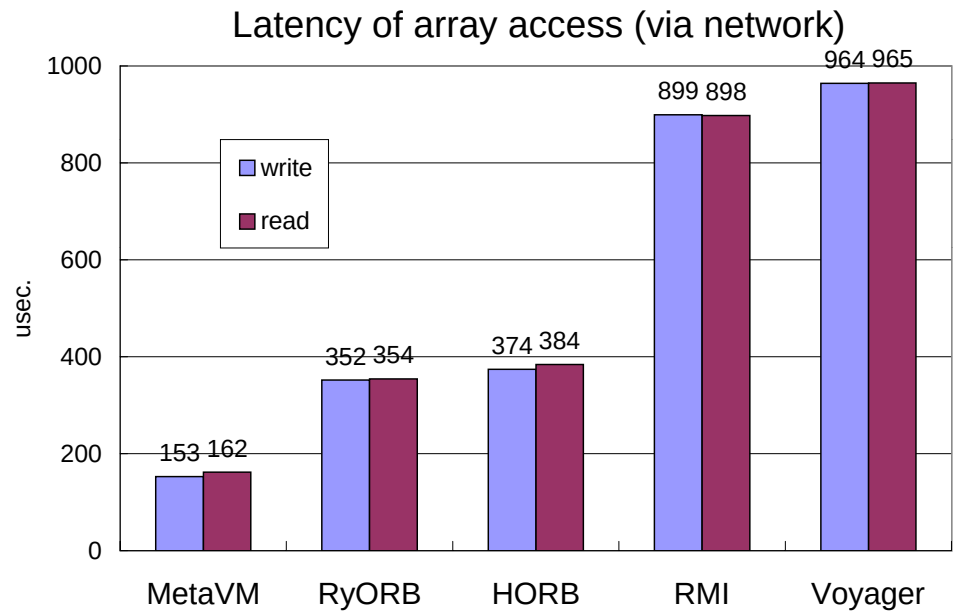

図 3.11: 遠隔配列アクセスの遅延 (ネットワーク経由)

3.6.2　ローカルな処理の性能

第 3.4 節で述べたように，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 3.12，3.14 はそれぞれ，
SPEC JVM98[50]，Linpack benchmark [15]（$n = 500$）の結果である．SPEC JVM98 の結果は，ある Java 処理系を基準とした相対性能，つまり，実行時間の逆数を定数倍したものであり，Geometric Mean は SPEC JVM98 の全ベンチマークプログラムの幾何平均（n 個の値をすべて乗じたものの n 乗根）である．Linpack benchmark の結果は Mflops/秒である．どちらも数値は大きい方が良い結果を表す．

4 通りの条件で計測した．グラフの左から，インタプリタ（without 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．

図 3.13 は，非分散版 shuJIT での結果に対する MetaVM 対応 shuJIT の結果の比である．MetaVM 対応が shuJIT の性能にどの程度の影響を与えたかを示している．100%に

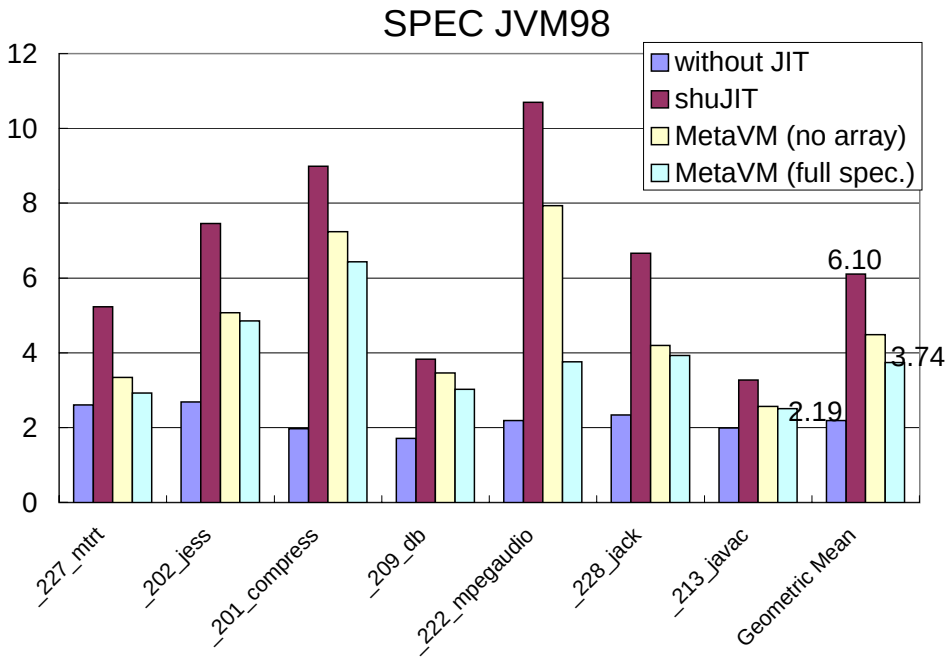

図 3.12: ローカル実行の性能 – SPEC JVM98

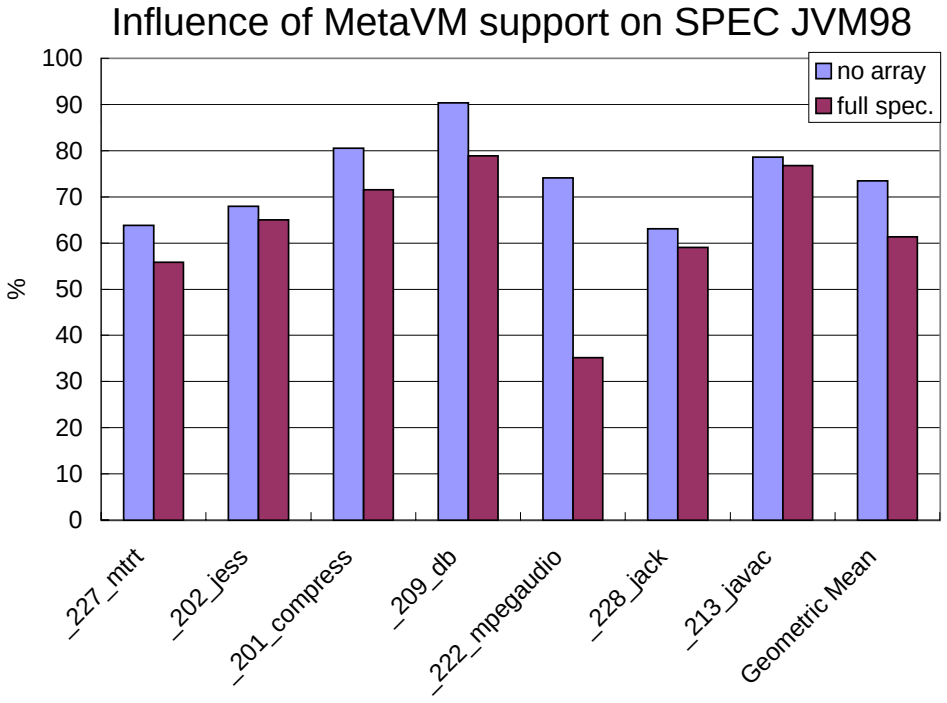

図 3.13: 性能への影響 – SPEC JVM98

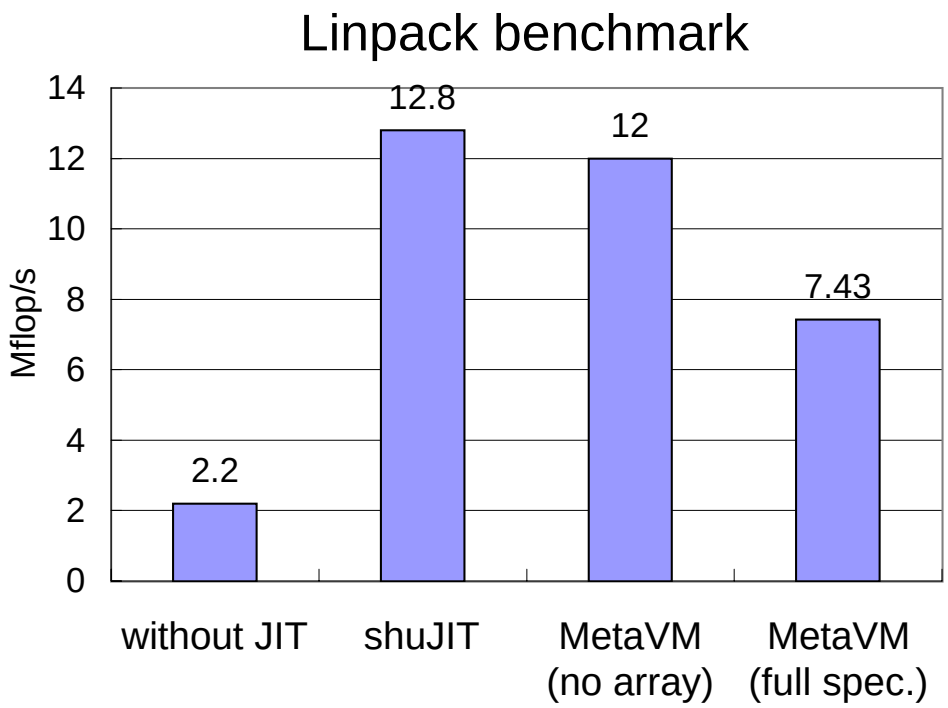

図 3.14: ローカル実行の性能 – Linpack ベンチマーク ($n = 500$)

近いほど影響が小さい，つまり良い結果である．

全機能を有効にした MetaVM でも，すべてのベンチマークプログラムでインタプリタより良い結果が示された．MetaVM によって最も結果が下がったベンチマークは SPEC JVM98 の_222_mpegaudio で，非分散版 shuJIT の 35.1%に低下している．配列の遠隔操作を無効にした場合（no array）は 74.1%への低下にとどまっていることから，配列アクセスが多いプログラムであることがわかる．

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，参照の値渡しというセマンティクスが崩れ，プログラムによっては分散実行とローカル実行とで実行結果が変わり得る．no array と full spec. を比較することで，このような透過性の低下を許せる場合に性能低下をどの程度防げるのかが判る．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，性能低下をほとんどなくすことができている（図 3.14）．

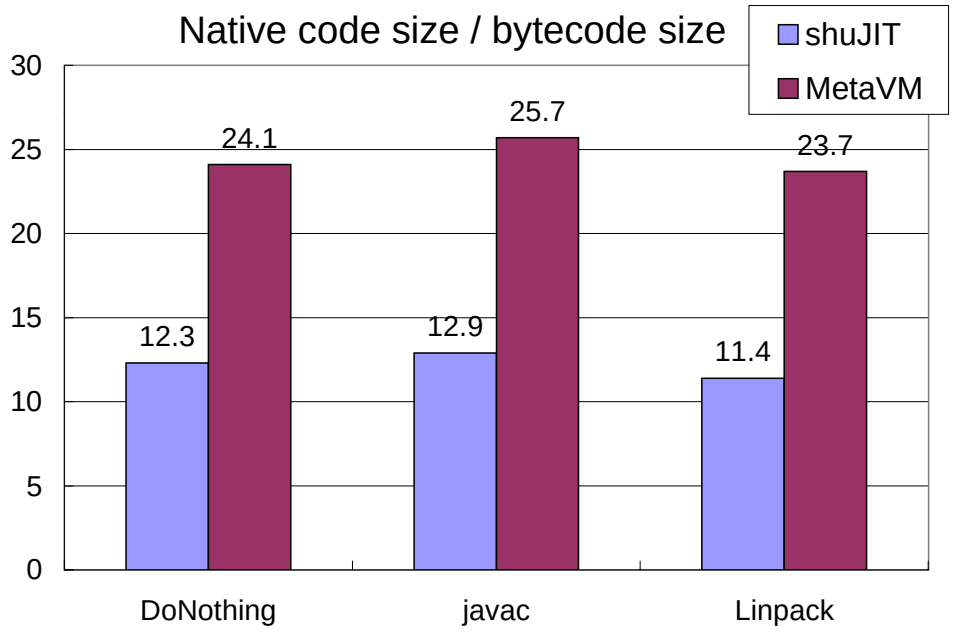

図 3.15: ネイティブコードとバイトコードのサイズの比

3.6.3 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 3.4 節）．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 3.15 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 3.16 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．MetaVM 対応 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である．

```
class DoNothing {
```

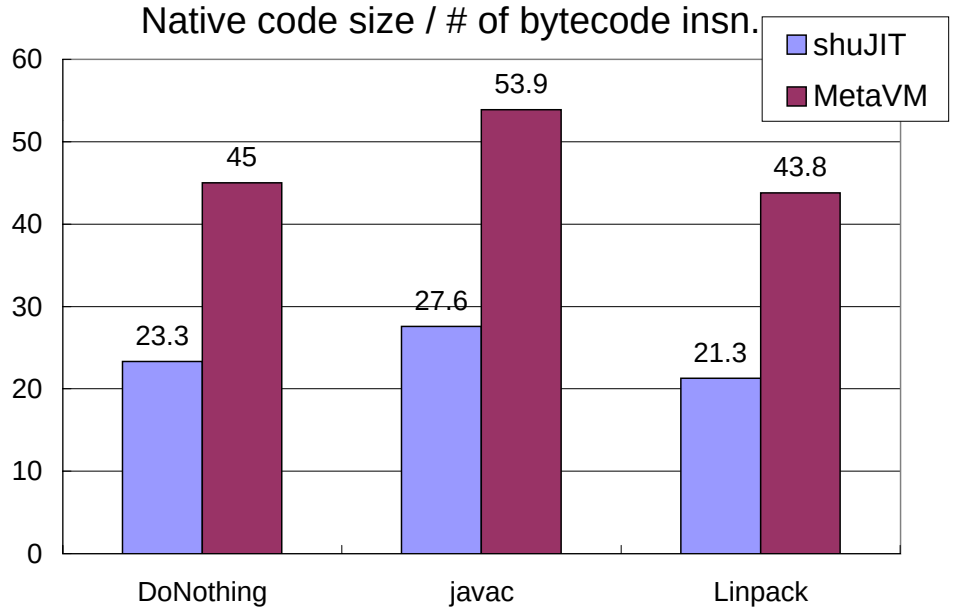

図 3.16: ネイティブコードのサイズとバイトコード命令数の比

```
  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，メソッドの先頭と末尾に shuJIT が付加するプロローグ，エピローグのコードを含んでいる．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．Linpack より javac の方が悪い結果，つまりネイティブコードが大きくなっているのは，一般に，コンパイラなどのシステムソフトウェアの方が，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ことを反映しているのであろう．

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．

3.7 他の手法および関連研究

3.7.1 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という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 3.2.1 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コンピュータ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．

3.7.2　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．現に，cJVM [7] はそのようにして，複数コンピュータで稼働する単一の Java 仮想マシンを構成している．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対応させる必要がある．はじめから実行時コンパイラで対応した方が手間は少なくて済む．

3.7.3　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 [62] 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．

カリフォルニア大学バークレイ校では，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，メソッド呼び出しをネットワークインタフェースの直接操作に変換する目的で shuJIT が利用された．

3.8 むすび

この章では，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，コンピュータ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コンピュータ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．

現在の API では，インスタンスを生成先コンピュータはプログラマが指定する必要がある．これが唯一，プログラミングインタフェースに関する単一コンピュータと MetaVM の違いである．メタコンピュータを構成するという目的のためには，適切な生成先を自動的に選択するアルゴリズムを開発する必要があろう．

第 4 章

実行時コンパイラ

本研究で設計，実装した分散オブジェクトシステム（第 3 章）のネットワーク透過性は，Java バイトコードの実行時コンパイラ shuJIT によって達成されている．この章では，実行時コンパイルを応用した研究の基盤として用いるべく開発したコンパイラ shuJIT の，コンパイル手法，最適化，性能について述べていく．

4.1 はじめに

本研究の方針は，Java 仮想マシンによって達成されている分散処理向けの諸性質（第 1.1.3 項）を利用しつつ，さらにいくつかの機能を付加しようというものである．Java 仮想マシンを使いつつ高い実行性能を得るためには，実行時コンパイラ（Just-in-Time コンパイラ,JIT コンパイラ）が必須である．また，第 3 章でその設計を述べた分散オブジェクトシステムでは，実行時コンパイラがバイトコード命令の解釈を変更することで高いネットワーク透過性を達成している．こういった研究には，基盤として利用できるコンパイラが必要である．

研究基盤として利用するために，Java バイトコードの実行時コンパイラ shuJIT [47] の開発に取り組んできた．shuJIT は Intel 社の IA-32，通称 x86 プロセッサ [11][61] 用の実行時コンパイラで，OS として Linux と FreeBSD に対応している．Sun Microsystems 社（Sun 社）が開発した Java 仮想マシン classic VM と共に動作する．

設計，開発には次の方針で臨んだ．

- **実用的**なソフトウェアとする．
- **研究基盤**として利用しやすいものとする．

- **少ない労力で短期間で開発**する．ひとりで，長くとも数ヵ月程度の期間で作る．

shuJIT の開発では，上記の方針に基づいて種々の工学的選択を行ってきた．例えば，shuJIT 側であらかじめ用意したネイティブコードをつなぎ合わせていくというコード生成手法は，研究基盤としての扱いやすさと開発期間の短縮を同時に狙った選択である．この手法を採用したことで，コンパイラ内にアセンブラを持つ必要がなくなり，その分，開発の手間を削減できた．それと同時に，shuJIT 開発者にとっては，コンパイラが生成するネイティブコードを直接手で記述できるという利点もある．これによって，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（第 3 章）の開発が比較的容易に，そして現実的になった．

このコード生成方式には，コンパイルに要する時間を短くできるという利点がある．実行時コンパイラではコンパイル時間がそのままプログラムの実行時間に加わってしまうので，これは大きな利点である．その半面，このコード生成方式では，複数のレジスタを活用するコードを生成することが難しい．近年の高性能プロセッサはロードストアアーキテクチャであり，レジスタの効率的な利用なくして高い性能を達成することはできない．この問題を解決するために，**スタック状態（*stack state*）**という考えを導入した．これにより，高速なコード生成と複数レジスタの利用を両立することができ，実用的な性能を達成することができた．

研究基盤として利用しやすいものとするという方針は，実際に他の研究に利用されるという形で結実した．自身では，分散オブジェクト（第 3 章）以外に，`strictfp` [54] の効率的な実装を探る目的に利用した [48][60]．また，自ら利用しただけではなく，他の研究者によって，高性能通信，入出力の研究を行う基盤として利用された [57]．彼らは，研究に利用する実行時コンパイラを選ぶ際，TYA [35] と shuJIT を比較している．その結果 shuJIT が選択された．

実用的なソフトウェアに仕立てるという方針は，当然ながら，自分以外にも役立つコンパイラにする，という目標を達成するためのものである．しかしそれだけではなく，多くの利用者を獲得することでバグ，問題の報告数を増やし，ソフトウェアの品質を上げるという狙いもある．現に，shuJIT の公開からおよそ 2 年半の間にソースコード約 7000 回，バイナリ約 8000 回のダウンロードがあり，受けた問題報告も数十に及び，これが品質向上に多いに役立った．

本章では，実行時コンパイラ shuJIT について，コンパイル手法，実装した各種の最適

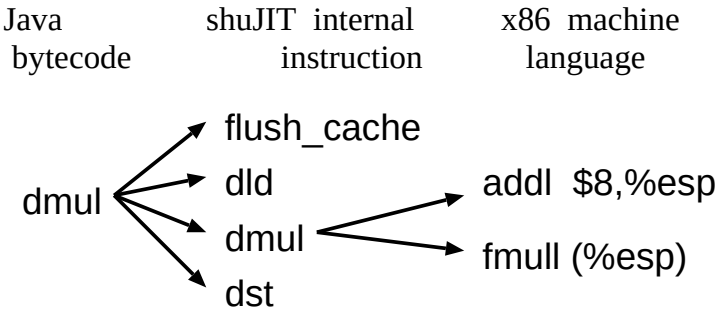

図 4.1: コード生成手法

化，また，性能評価の結果を述べていく．

4.2 コード生成手法

コンパイラは通常, コンパイル対象のコードを一旦何らかの中間表現に変換し, それに対して各種の最適化を施した後，コード生成を行う．中間表現としては四つ組（quadruple）や RTL（Register Transfer Language）が一般的に用いられている．

shuJIT が用いる中間表現，すなわち shuJIT 内部表現は，Java バイトコード命令に非常に近い．たいていのバイトコード命令はひとつの shuJIT 内部命令に置き換えられる．それぞれの内部命令には，対応するネイティブコード列（pre-assembled コード）が shuJIT 側であらかじめ用意されていて，内部命令からネイティブコードへの変換は，直接の置き換えで行われる（図 4.1）．ただし，内部命令列に対して数種類のピープホール最適化は施す．

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研究の材料として使いやすくするための方式である．アセンブリ言語で記述しておいたコードが直接にコンパイル結果の一部となるので，コンパイラに生成させたいコードを shuJIT 開発側で直接記述しておくことができる．

■軽いコンパイル処理　このコード生成方式の長所として，コンパイル過程の各パスでの処理はすべて，バイトコード命令列長 n として $O(n)$ のコストで済むことが挙げられる．多くの中間表現が対象とするレジスタマシンモデルでは，データフロー解析などに $O(n^2)$ 以上のコストがかかってしまう．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．

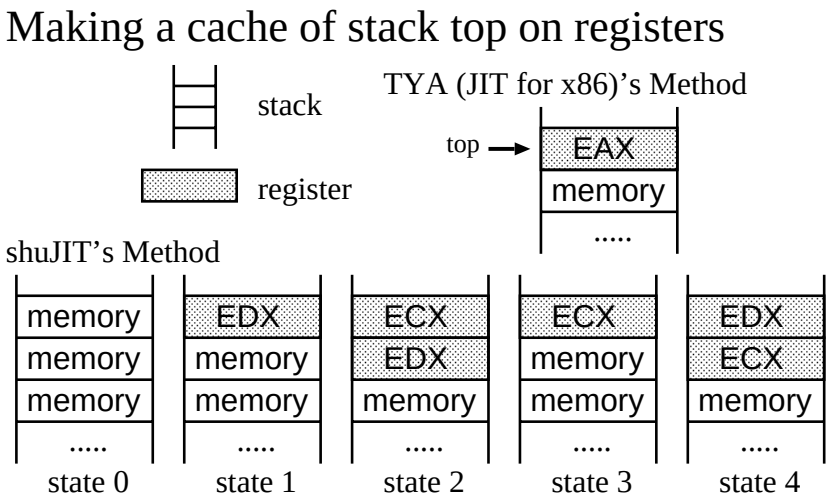

図 4.2: スタック状態

コンパイル時間を短く抑えるためには，コンパイル時間を短くする以外にも，コンパイルの対象を絞り込んで極力減らすという方針もあり，多くの実行時コンパイラが採用している．プログラム中で頻繁に実行される部分はそう多くないという経験則に基づき，頻繁に実行される部分，すなわちホットスポット（hot spot）のみをコンパイルしようという方法である．ホットスポットのコンパイルには時間を費して性能の良いコードを生成する代わりに，コンパイル対象は減らすことで，性能は向上させつつもコンパイル時間全体を長くせずに済む．

かと言って，処理が軽いコンパイル方式は不要かと言えば，そうではない．反対の性質を持つ，重いコンパイル処理で質の高いコードを生成する JIT コンパイラと補い合うことができる．Intel 社の Java 仮想マシン Open Runtime Platform [30] や IBM 社の Jalapeño 仮想マシン [6] はインタプリタを持たず，その代わりに処理の軽い実行時コンパイラを持つ．ホットスポットとして検出した部分に対してだけ処理が重い最適化コンパイラを適用し，それ以外の，コンパイルに時間を費す価値がないか，もしくは価値があるか判らない部分は処理が軽いコンパイラを適用するのである．このように，仮にホットスポットの検出がうまくいったとしても，処理の軽いコンパイル手法が依然重要であることに変わりはない．

本手法には，処理が軽い以外にもうひとつ，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．これがコンパイラ開発の労力削減に大きく貢献している．

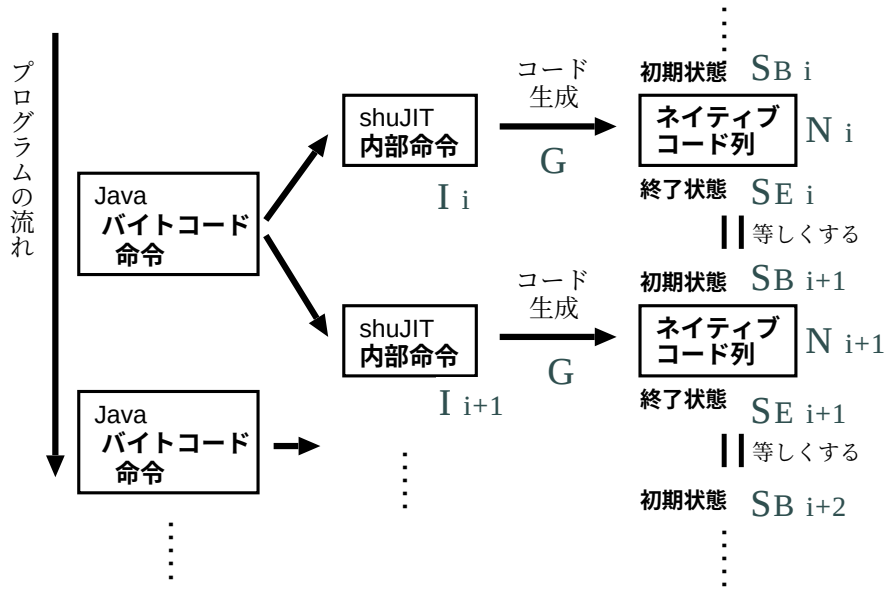

図 4.3: スタック状態の整合

■スタック状態 しかし，このコード生成方式でプロセッサのレジスタを有効に活用するのは難しい．例えば，実行時コンパイラ TYA [35] は，shuJIT と同様に，用意しておいたネイティブコード列をつないでいくことでコード生成を行う．この TYA が割り付け対象としているレジスタは EAX ただひとつある（図 4.2）．バイトコード命令一命令分の pre-assembled コードの中では複数のレジスタを活用しているものの，pre-assembled コード間で利用されるレジスタはひとつだけである．この問題を解決すべく，shuJIT ではスタック状態という考え方を導入した（図 4.2）．まず，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．内部命令と状態の組に対してそれぞれ pre-assembled コードを用意しておく．コード生成処理では，基本的に内部命令を pre-assembled コードに置き換えていくのだが，その際に，直前の pre-assembled コードの実行終了時点での状態に対応した pre-assembled コードを選び，生成していくのである（図 4.3）．

shuJIT では，図 4.2 の通り状態数を 5 とした．2 つのレジスタで，Java 仮想マシンのスタックトップ付近をキャッシュする．このコード生成方式では，内部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．200 を超える内部命令に対し 5 状態を設けたので，1000 以上の pre-assembled コードの用意が必要となった．とはいえ実際には，多くの内部命令で，全 5 状態について同じ pre-assembled コードを使い回せることもあり，用意した pre-assembled コードの数は 1000 より少なくなった．

キャッシュとして使用するレジスタが 2 つとは少なく見えるかもしれない．レジスタ割り付けを行えば，より多くのレジスタを有効に使えるだろう．しかし，IA-32 は汎用の整数レジスタを 8 つしか持たない．shuJIT では，キャッシュに使用する 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使用している他，その他のレジスタも pre-assembled コード内では活用している．IA-32 はたいていの RISC プロセッサよりレジスタが少ないために，レジスタ割り付けを行うか否かの差が小さいのである．また，Pentium Pro 以降の IA-32 プロセッサは，実際は内部に数十のレジスタを持ち，8 つの汎用（論理）レジスタをそれらに割り付けている（レジスタリネーミング）．多くの RISC プロセッサでは，レジスタ群の活用はレジスタ割り付けを行う立場にあるコンパイラの仕事であるが，IA-32 では，その仕事の多くをプロセッサが引き受けているのである．レジスタ群の活用をプロセッサに任せるという shuJIT のコード生成手法は IA-32 に向いた手法である．

以下，shuJIT のコード生成手法を形式的に説明する．次の記号を用いる．

I_i	shuJIT 内部命令
$N(I, S)$	pre-assembled コード列
$S_{B\ i}$	初期状態
$S_E(N)$	pre-assembled コード列 N の終了状態
G	コード生成処理

ここで，‘i’ は i 番目の shuJIT 内部命令に対応していることを表している．この時，i 番目の内部命令についてのコード生成処理は以下の式で表せる．

$$(N(I_i, S_{B\ i}), S_E(N(I_i, S_{B\ i}))) = G(I_i, S_{B\ i}) \tag{4.1}$$

つまり，内部命令と初期状態から，対応するネイティブコード列と，そのコード列の終了状態を得る．次の $i+1$ 番目の内部命令に対する処理では，まず，次のように初期状態を直前のネイティブコード列の終了状態と等しくとる．

$$S_{B\ i+1} = S_E(N_i) \tag{4.2}$$

続いて，式 4.1 に従い，$i+1$ についても同様にコードを生成する．以下，$i+1, i+2, \ldots$ についても同様の処理となる．

4.3 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．

4.3.1 プロセッサのスタックの利用

Sun 社の classic VM 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，インタプリタはこれを利用している．shuJIT はこの仮想マシン用スタックを使わない．IA-32 プロセッサのスタック操作命令を用いることで，高速にスタック操作を行う．

仮想マシン用スタックへの push 操作は，値のメモリへの書き込み，仮想マシンのスタックポインタのインクリメントという手順を要する．それに対して，プロセッサのスタック操作命令を用いることで，push 操作をプロセッサの一命令で済ませることができる．また，仮想マシンのスタックポインタが置かれているメモリへのアクセスも不要である．

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，また逆の場合は，引数と返り値を積み直す必要がある．

レジスタ割り付けを行うコンパイラにこの工夫は無用であるが，shuJIT のコード生成手法との組み合せは有効である．

4.3.2 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．一度きり，そのコードが初めて実行されたときだけ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権を持っているのかどうかチェックしなければならない．このチェックは，そのコードが初めて実行されたときだけ行えば充分である．また，メソッド呼び出し命令では，呼び出すメソッドが定義されているクラスが初期化済みか否かのチェック，および初期化処理を 1 度きりにするために，自己書き換えコードを生成する．

4.3.3 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操作を行った場合に，例外 `NullPointerException` が発生する．当然，実行時コンパイラが生成するコードにもこの動作が要求される．操作対象が `null` かそうでないかの検査を行わねばならない．

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 番地をアクセスしようとし，プロセッサのメモリ保護機能が働き，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，shuJIT はこの SIGSEGV を利用する．シグナルハンドラで捕えて，NullPointerException を発生させるのである．このようにシグナルを利用して null 検査を行うことで，明に検査するコードとその検査時間を省ける．オブジェクトへの操作では，NullPointerException が発生しない場合がほとんどなので，高速になる．

4.3.4　末尾再帰の除去

メソッドの再帰呼び出しが末尾再帰（tail recursion）であった場合，呼び出しをメソッド先頭へのジャンプに変換する．これによって新たなスタックフレームの作成を防げるので，作成に要する時間とスタックの消費を節約できる．プログラミング言語を問わず，有名な最適化である．例えば Scheme 言語では実装が義務付けられている．

shuJIT が除去できる末尾再帰は，自分自身に対する静的（static，private）メソッド呼び出しである．IBM JDK に含まれる，東京基礎研究所で開発された JIT コンパイラは，devirtualize [31] できた動的呼び出しも末尾再帰除去の対象とする．

4.3.5　ジャンプ先コードの整列

shuJIT は，ループの先頭を 16 バイト境界に整列（align）させる．正確には，後方ジャンプと後方分岐の先をループの先頭とみなして整列させる．

近年の IA-32 プロセッサ，Pentium Pro（II，III，Celeron）では，命令が 16 バイト境界で始まっていないと，デコーダで小さなペナルティが発生する．これを避けるための方策である．

しかし，この最適化は生成コード量を増やす．整列させるためだけに，NOP 命令やジャンプ命令を詰めた無駄なコード領域を設けねばならないからである．生成コード量の増加はキャッシュのヒット率に悪影響を与えるので，過度の整列は性能低下を招きかねない．shuJIT では，ジャンプの中でも頻繁に実行されるであろう後方へのジャンプ，分岐について，ジャンプ先のコードを整列させることとした．

4.3.6 ピープホール最適化

shuJIT 内部命令の列に対してピープホール最適化を施す．

例えば，スタック上の値をローカル変数に対して pop し，そのローカル変数を再び push するというバイトコード命令列がよく見られる．この操作は単に，スタックの一番上の値をローカル変数に格納すれば済むので，そのように変換するのである．スタックポインタの操作や，ローカル変数からの読み込みは無駄な処理である．

また，ローカル変数をスタックに push し，それを浮動小数点演算器（FPU）のレジスタにコピーするといった無駄なコピーも，ピープホール最適化で除去する．

4.3.7 インライン展開

内部命令列に対してインライン展開を施す．末尾再帰の除去と同様，静的メソッド呼び出し（`static`，`private`，`final`，スーパクラスのコンストラクタ呼び出し）が対象である．

shuJIT が展開するのは，ジャンプ命令を含まない，`catch` 節を含まないメソッドである．これは，コンパイル処理を軽くするための選択である．また，再帰的な展開は 2 段階までとし，展開するメソッドの長さは内部命令で 20 命令以下のものとした．2 段階，20 内部命令という値は，様々な条件をベンチマークプログラム SPEC JVM98 [50] で試して決定した値である．

4.4 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．

比較対象として用いた実行時コンパイラは次の通りである．すべて Linux 用を用いた．

IBM JDK 1.3.0　IBM 社製の処理系．`build cx130-20000815` である．東京基礎研究所で開発された JIT コンパイラ [51][64]，JITC 3.6 を含む．

IBM JDK 1.1.8　IBM 社製の処理系．`build l118-20000713` である．JITC 3.0 を含む．

Sun JDK 1.3.0　Sun 社製の処理系．`build 1.3.0rc1-b17` である．2 種類の Java 仮想

マシン，HotSpot Client VM と HotSpot Server VM を含み，それぞれ特性の異なる JIT コンパイラを持つ．

Kaffe 1.0.6　Transvirtual Technology 社製の処理系 [56]．JIT コンパイラ JIT v3 を含む．

sunwjit.so　Blackdown 製 JDK 1.2.2 FCS に含まれる JIT コンパイラ．

JBuilder Java 2 JIT 1.2.15　Borland 社製 JIT コンパイラ．メソッドが初めて呼ばれた際はインタプリタで実行し，2 度目に呼び出された時点でコンパイルする，という方式を採っている．

OpenJIT 1.1.14　東京工業大学松岡研究室と富士通，富士通研究所で開発された JIT コンパイラ [42]．ほとんどの部分が Java 言語自身で記述されている．

TYA 1.7v2　Albrecht Kleine 氏が開発した JIT コンパイラ [35]．

インタプリタ（interpreter）　Blackdown 製 JDK 1.2.2 FCS が持つバイトコードインタプリタ．

GCJ　バイトコード，または Java 言語のプログラムをネイティブコードに変換するスタティックコンパイラ [24]．gcc 2.95.2 に含まれるものを用いた．

これらのうち，インタプリタと GCJ 以外はすべて JIT コンパイラである．GCJ はスタティックコンパイラであり，プログラムは，実行に先だってあらかじめネイティブコードにコンパイルしておく必要がある．

shuJIT を含めて，sunwjit.so，JBuilder JIT，OpenJIT，TYA は Sun 社の classic VM 用の JIT コンパイラである．これらの JIT コンパイラとインタプリタの性能測定には，Blackdown というプログラマグループが Linux に移植した Java Development Kit（JDK） 1.2.2 FCS を使用した．この JDK には Java 仮想マシンとして classic VM が含まれている．IBM 社の JDK には，classic VM を元に IBM が変更を加えた Java 仮想マシンが含まれている．また，Kaffe は独自の Java 仮想マシンおよびクラスライブラリを持っている．クラスライブラリの違いが測定結果に影響を与えることも考えられる．

4.4.1　SPEC JVM98

SPEC JVM98 [50] は，SPEC（Standard Performance Evaluation Corporation）の OSG（Open Systems Group）が作成，提供している，Java 仮想マシンのベンチマークプログラムである．SPEC が提供する他のベンチマークプログラムと同様，世の中で実

際に利用されているアプリケーションプログラムを元にしたベンチマークである．SPEC JVM98 には，次の 8 種のベンチマークプログラムが含まれる．

_200_check　Java 仮想マシンの機能を確認するプログラム．性能の指標としては使われない．

_201_compress　LZW 法（Lempel-Ziv 法の亜種）の圧縮プログラム．SPEC CPU95 の 129.compress ベンチマークを Java 言語に移植したもの．

_202_jess　NASA の CLIPS エキスパートシステムの Java 言語版．推論エンジンは，実行に従って大きくなるルールセットを探索していく．

_209_db　IBM 社によって書かれた，データ管理のベンチマーク．メモリ上の住所データベースの上で複数の関数，つまり，追加，削除，検索，ソートを実行していく．

_213_javac　JDK 1.0.2 の Java コンパイラ．

_222_mpegaudio　MPEG audio Layer-3（いわゆる MP3）のデータを伸長するアプリケーション．4 メガバイトの音楽データを扱う．

_227_mtrt　レイトレーシングのプログラム．340 キロバイトの入力ファイルから恐竜を描く．2 つのスレッドが走る．

_228_jack　パーザジェネレータ．現在の JavaCC の初期の版である．jack 自身を生成する処理を行う．

各ベンチマークプログラムのスコアは，ある環境（PowerPC 604 133 MHz，AIX 4.1.5.0，JDK 1.1.4，インタプリタ）での実行時間を基準として，その何倍速く実行できたかを表す．つまり，数値が大きいほど良い結果を表す．総合スコアである Geometric Mean は，全ベンチマークプログラムの相乗平均，すなわち，n 個の値をすべて乗じたものの n 乗根である．

■総合スコアの算出方法　総合スコアとして相乗平均を用いるのは，各ベンチマークが総合スコアに与える影響を均一化するためである．SciMark 2.0 は，各ベンチマークについて得られた Mflop/秒の値の相加平均を総合スコアとするため，一部のベンチマークがほとんど総合スコアに影響を与えないという問題がある（第 4.4.2 項参照）．このように，複数のベンチマーク結果を元に算出される総合スコアは，各ベンチマークプログラムの選び方と総合スコアの求め方に大きく左右されるので，あくまで目安だと考えるべきである．

SPEC JVM98	mtrt	jess	compress	db	mpeg-audio	jack	javac	Geometric Mean
IBM JDK 1.3.0	26.0	23.7	14.8	12.4	33.5	30.6	13.2	20.5
HotSpot Server VM	24.2	27.4	13.9	9.53	36.5	35.9	11.6	19.2
HotSpot Client VM	16.1	22.6	13.0	9.19	20.5	29.3	11.3	16.2
IBM JDK 1.1.8	11.7	11.7	13.1	7.56	23.8	16.2	7.20	12.1
OpenJIT	8.13	8.52	10.3	3.35	11.1	8.64	4.31	7.18
JBuilder JIT	8.88	8.32	10.4	3.45	13.9	2.74	3.81	6.28
shuJIT	5.23	6.86	8.71	3.97	10.8	6.78	3.44	6.10
TYA	5.79	6.69	7.67	3.39	8.75	6.66	3.69	5.79
sunwjit.so	2.30	2.90	9.26	N/A	7.94	5.82	3.09	4.55
Kaffe	3.27	2.46	16.3	4.28	7.66	2.15	2.72	4.25
インタプリタ	2.60	2.68	1.97	1.71	2.19	2.34	1.99	2.19

表 4.1: SPEC JVM98 の結果

■測定結果　表 4.1, 図 4.4 は SPEC JVM98 の結果である．相乗平均（Geometric Mean）の値が大きい順に並べてある．測定は，動作周波数 400 MHz の K6-2 プロセッサで行った．ヒープの初期サイズおよび最大サイズは，32 メガバイトに設定した（オプション `-Xms32m -Xmx32m` を指定）．表 4.1 中で，sunwjit.so の_209_db が計測不能となっている

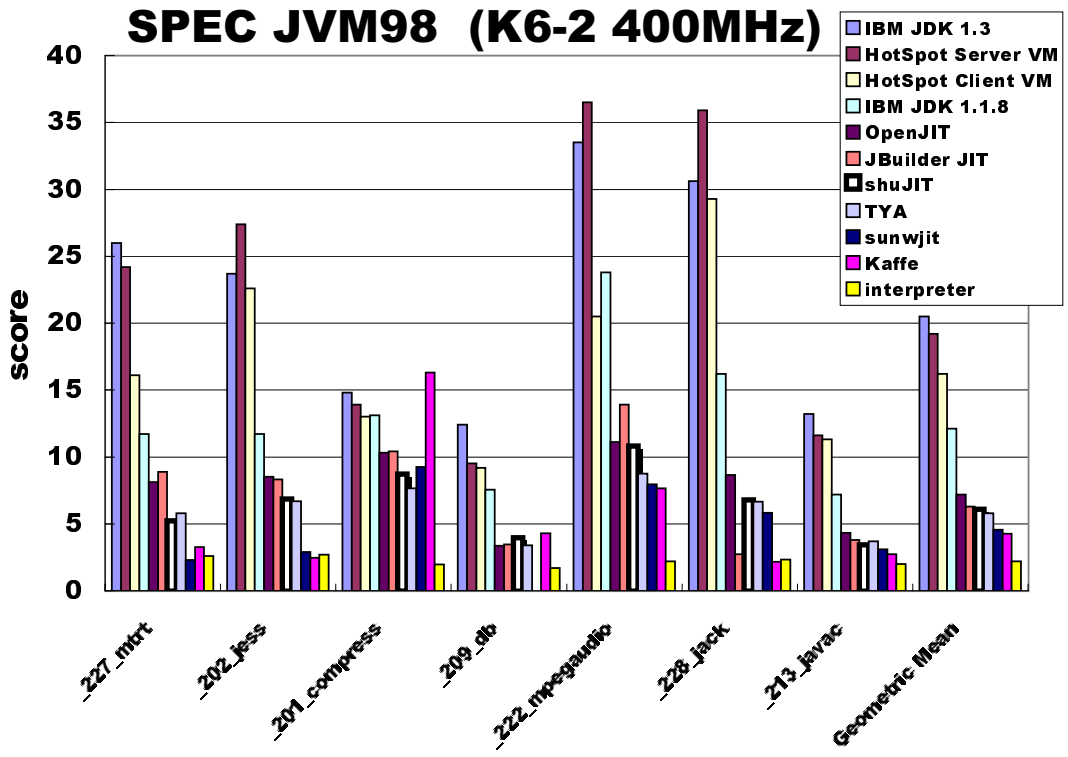

図 4.4: SPEC JVM98 の結果

のは，このベンチマークプログラムが sunwjit.so では正しく動作しなかったことを示す．

総合スコアは，IBM JDK 1.3.0 が最も良い結果を示した．結果が良かった一群，すなわち IBM JDK および 2 種の HotSpot VM とそれ以外の差が大きい．shuJIT の結果は，IBM JDK と HotSpot VM を除いた，他の JIT コンパイラとほぼ同じである．処理が軽いコード生成手法（第 4.2 節）で，他の JIT コンパイラと同程度の性能を達成できることを確認できた．

shuJIT を，同じ classic VM で動作する 4 種の JIT コンパイラ，OpenJIT，JBuilder JIT，TYA と比較すると，_209_db では shuJIT の結果が最も良く，逆に_227_mtrt では最も悪い．これは，インタフェース呼び出しが多い_209_db，解析によって一意決定できる動的呼び出しが多い_227_mtrt という，各ベンチマークプログラムの性質 [64] を反映していると考えられる．shuJIT は，これらの JIT コンパイラと比較してインタフェース呼び出しの効率は良く，逆に，動的呼び出しの効率は悪いという結果となった．

4.4.2 SciMark 2.0

SciMark 2.0 [43] は，米国 NIST（National Institute of Standards and Technology）の Roldan Pozo と Bruce Miller が作成したベンチマークプログラムで，科学技術計算によく現れる処理の性能を計測する．5 種類のベンチマークプログラム，一次元の高速フーリエ変換，SOR 法，モンテカルロシミュレーションによる π の計算，疎行列の乗算，密行列の LU 分解から成る．このそれぞれについて，何 Mflop/秒で処理できたかを計算し，それらの相加平均を ‘Composite Score’ という総合スコアとする．

■測定結果 表 4.2，図 4.5 は SciMark 2.0 の結果である．総合スコアが高い順に並べてある．計測は，動作周波数 333 MHz の Pentium II で行った．

表と図の中で，GCJ に括弧で但し書きが付けてあるのは，SciMark 2.0 をネイティブコードにコンパイルした際の条件を表している．‘–O2’ と ‘–O0’ は GCJ に与えたオプションで最適化レベルを表し，‘.class’ はクラスファイルを入力として与えたことを表す．この表中にはないが，‘.java’ はソースファイルを入力したことを表す．SciMark 2.0 の場合，ソースコードを与えて得られた実行形式は正常に実行を完了することができなかったため，評価できなかった．

特に良い結果を示した IBM JDK と HotSpot VM を除くと，shuJIT の結果は他の JIT コンパイラ並である．他の JIT コンパイラと同程度の性能を達成していることを確認で

SciMark 2.0	FFT	SOR	Monte Carlo	Sparse matmult	LU	Composite Score
IBM JDK 1.3.0	36.6	91.5	9.85	44.5	85.7	53.6
IBM JDK 1.1.8	36.5	84.3	7.04	41.8	58.8	45.7
HotSpot Client VM	18.4	84.7	8.32	30.0	50.7	38.4
HotSpot Server VM	19.2	71.3	10.1	24.1	50.7	35.1
GCJ （–O2 with .class）	17.1	42.1	2.29	20.8	26.4	21.7
OpenJIT	8.69	32.7	1.94	10.6	19.4	14.7
JBuilder JIT	10.1	19.7	2.02	12.9	16.8	12.3
shuJIT	6.13	27.5	1.57	12.0	11.1	11.6
sunwjit.so	9.19	32.9	1.13	10.9	3.00	11.4
Kaffe	5.39	23.5	2.16	10.8	13.1	11.0
GCJ （–O0 with .class）	6.53	20.8	1.63	11.1	10.2	10.0
TYA	4.21	14.7	1.59	8.97	11.9	8.27
インタプリタ	1.21	3.69	0.642	2.36	2.95	2.17

表 4.2: SciMark 2.0 の結果

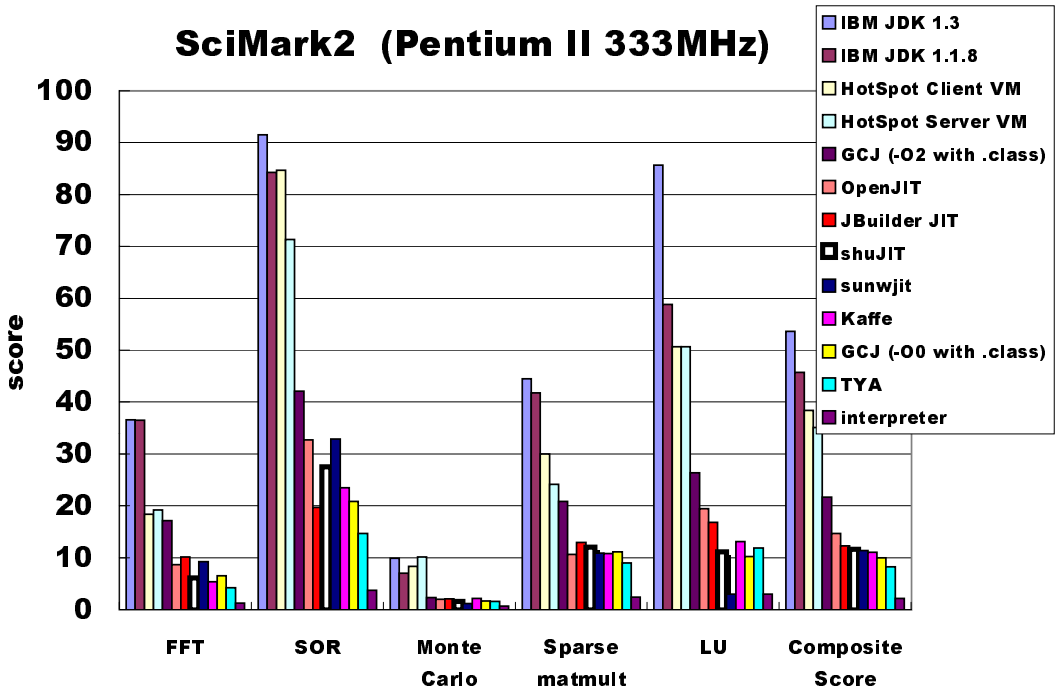

図 4.5: SciMark 2.0 の結果

きた.

■総合スコアの算出方法 SciMark 2.0 の結果を見ると，総合スコアの求め方に問題があることが判る．SOR と Monte Carlo の結果に大きな開きがあるにも関わらず，単純に相加平均を総合スコアとしてしまっている．ここで評価した全ての処理系で，SOR の数値が最も大きく，Monte Carlo の数値が最も小さい．そしてその開きは，最も小さいインタプリタであっても 5.75 倍，最も大きい sunwjit.so に至っては 29.1 倍となっている．これでは，Monte Carlo の結果は総合スコアにほとんど影響を与えない．総合スコアを求める際は，Mflop/秒といった数値を直接用いるのではなく，各ベンチマークが総合スコアに与える影響が極力均一になるよう工夫すべきであろう．例えば，相加平均を用いるのであれば，各ベンチマークについて得られた数値に対して一般的な処理系での結果に対する比を計算する．または，相乗平均を用いるという方法もある．このようにすることで，各ベンチマークが総合スコアに与える影響を均一に近づけることができる．影響の均一化を考えた後でなら，各ベンチマークに対して，その重要性に応じた重み付けを検討してもよいだろう.

4.4.3 Linpack benchmark

Linpack benchmark [14] は，ガウスの消去法で連立一次方程式を解くベンチマークプログラムである．浮動小数点演算の性能を測定することを目的とし，結果は flop/秒で得られる.

2 通りの問題サイズ，500×500 と 200×200 で計測した．表 4.3 は両方の結果，図 4.6 は 500×500 の結果をグラフにしたもの，図 4.7 は 200×200 の結果をグラフにしたものである.

shuJIT は，IBM JDK，HotSpot VM，GCJ に続く良い結果を示した.

4.5 コンパイル時間の評価

各 JIT コンパイラについてコンパイル処理に要する時間を計測したいのだが，ソースコードが公開されていない処理系について計測することは不可能か非常に困難である．そこで，Linpack benchmark での，2 通りの問題サイズ 200×200 と 500×500 の結果の違いに着目した.

Linpack benchmark （表 4.3，図 4.6，図 4.7）では，生成されたネイティブコードの

500 × 500	Mflop/秒	秒
IBM JDK 1.3.0	23.675	3.54
IBM JDK 1.1.8	17.994	4.66
HotSpot Server VM	16.783	5.00
HotSpot Client VM	15.053	5.58
GCJ （–O2 with .java）	14.284	5.87
GCJ （–O2 with .class）	13.813	6.07
shuJIT	11.277	7.43
JBuilder JIT	10.734	7.81
Kaffe	10.584	7.92
OpenJIT 1.1.14	9.455	8.87
TYA	9.449	8.87
GCJ （–O0 with .java）	9.18	9.13
GCJ （–O0 with .class）	7.96	10.53
sunwjit.so	2.362	35.49
インタプリタ	2.361	35.51

200 × 200	Mflop/秒
GCJ （–O2 with .java）	26.410
GCJ （–O2 with .class）	24.524
HotSpot Client VM	16.349
IBM JDK 1.1.8	14.928
shuJIT	14.300
OpenJIT 1.1.13	13.464
GCJ （–O0 with .java）	12.716
TYA	11.444
JBuilder JIT	11.257
OpenJIT 1.1.14	10.899
Kaffe	9.810
GCJ （–O0 with .class）	9.671
IBM JDK 1.3.0	5.675
HotSpot Server VM	3.540
インタプリタ	2.384
sunwjit.so	2.266

表 4.3: Linpack benchmark の結果

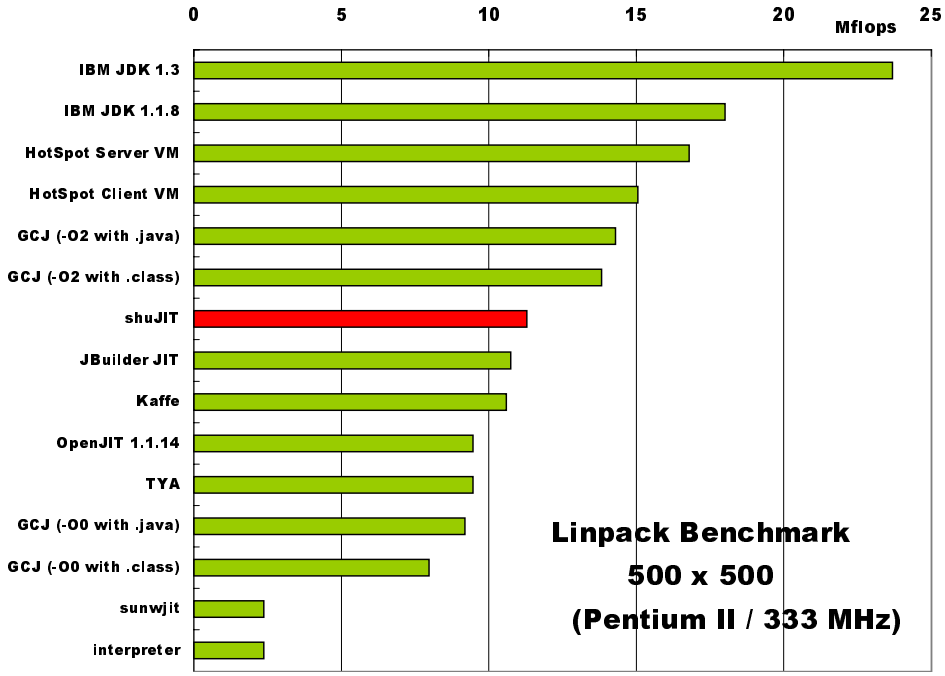

図 4.6: Linpack benchmark（500 × 500）の結果

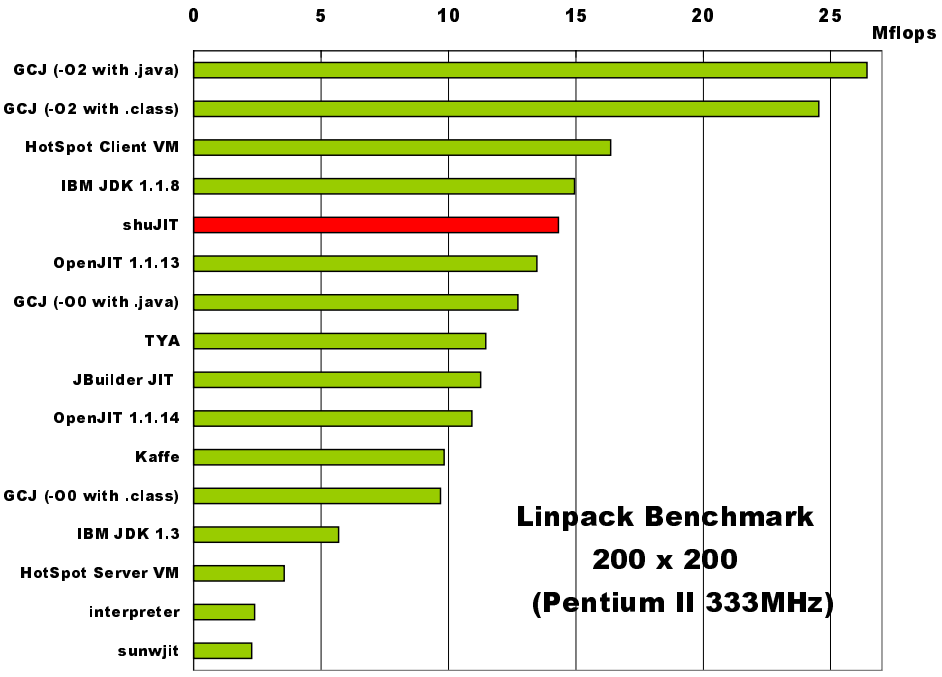

図 4.7: Linpack benchmark（200 × 200）の結果

ピーク性能は 500 × 500 での結果に表れている．200 × 200 では，どの JIT コンパイラでも実行はわずか数十ミリ秒で完了した．そのため，ピーク性能よりもむしろ，コンパイル処理に要した時間が表れている．例えば，500 × 500 では 23.675 Mflop/秒という最も良い結果を示した IBM JDK 1.3.0 が，200 × 200 では 5.675 Mflop/秒という結果に終わっている．また，実行前にコンパイルを済ませてしまう，つまり実行中はコンパイル処理が不要である GCJ が，200 × 200 で良い結果を示していることからもうかがえる．

もし，コンパイル処理に要した時間に対して，生成されたコードが充分長い時間実行されれば，コンパイル時間は隠れて，生成コードのピーク性能が計測される．生成コードの実行時間が短ければコンパイル時間が計測結果に表れてしまう．ここで，500 × 500 を前者，200 × 200 を後者であると考えた．すると，その比はコンパイル時間の長さを反映しているはずである．

表 4.4 に，500 × 500 に対する 200 × 200 の結果の比を示した．この比が大きいほどコンパイル時間は短いという相関，傾向があると考えている．実行中にコンパイルを行わない GCJ の値が軒並大きく，コンパイル時間が長いことで知られる HotSpot Server VM の値がとても小さいことからも，相関があると考えることが妥当であろう．

shuJIT の結果はとても良い．shuJIT に近いコード生成方式（第 4.2 節）を採っている

GCJ （–O2 with .java）	1.849
GCJ （–O2 with .class）	1.775
GCJ （–O0 with .java）	1.385
shuJIT	1.268
GCJ （–O0 with .class）	1.215
TYA	1.211
OpenJIT 1.1.14	1.153
HotSpot Client VM	1.087
JBuilder JIT	1.049
インタプリタ	1.010
sunwjit.so	0.9594
Kaffe	0.9269
IBM JDK 1.1.8	0.8296
IBM JDK 1.3.0	0.2397
HotSpot Server VM	0.2109

表 4.4: 500×500 の結果に対する 200×200 の結果の比

TYA の結果も良い．このコード生成処理の軽さが表れている．

比が 1 より大きい，つまり 200×200 の方が結果が良い処理系が多いのは，データセットの大きさを反映した結果であろう．行列の大きさは，200×200 では 300 キロバイト強，1.5 メガバイトである．データセットが小さい方がキャッシュメモリに載りやすい．また，この Linpack benchmark のアルゴリズムはブロック化されていないごく単純なものなので，200×200 で 1600 バイト，500×500 で 4000 バイトという一行の大きさも性能に影響を与え，200×200 の結果が良いことに結び付いていると考えられる．インタプリタでは比がおよそ 1，つまりデータセットの大きさが性能に影響を与えていない．これは，インタプリタの性能ではメモリアクセス以外がボトルネックになっているためだと推測している．

4.6 むすび

研究基盤として開発してきた実行時コンパイラ，shuJIT について述べた．特徴のひとつは，コンパイル時間の短さにも関わらず，複数のレジスタを利用できるコード生成方式であった．また，研究基盤として有用であるだけでなく，高い実用性を達成している点も，大きな特徴であった．

第 5 章

結論

本論文は，**メタコンピュータ**を構成するための一方式をまとめたものである．メタコンピュータとは，ネットワーク上に分散しつつ，かつ単一目的に利用できるコンピュータ群を指す．本研究では特に，プログラマに対してメタコンピュータを提供することを目標とし，そのためのシステムソフトウェアの構成法に取り組んだ．

Java 仮想マシンが分散処理向けの優れた性質を持つことに着目して，それらの性質，すなわち安全性，機種非依存性，高速実行を損なうことなしに，メタコンピュータの構成に不可欠な機能を付加する方法を提案，実証した．不可欠な機能とは，ネットワーク透過な分散オブジェクトおよび実行状態の移送である．

本研究の成果をまとめる．

- Java 仮想マシン間で，メタコンピュータの構成に不可欠な**異機種間，非同期スレッド移送**が可能であることを実証した．性能の点で重要である実行時コンパイラとの共存については手法を示した．また，スレッド移送を設計，実装する際の一般的な問題を明らかにし，採り得る手法を考察した．問題とは，移送すべきインスタンスの選択，位置依存資源の扱い，およびスタック上要素の型識別である．
- ネットワーク上のどこにあるオブジェクトも同一コンピュータ上のオブジェクトであるかのように扱える，**ネットワーク透過な分散オブジェクト**システムを開発した．この透過性を達成するために，実行時コンパイラを用いてバイトコード命令の解釈を変更するという手法を提案し，実装，評価した．
- 研究基盤として利用しやすい**実行時コンパイラ**を開発した．実際に分散オブジェクトに応用することで，有用性を示した．また，コストの小さいコード生成手法で実用的な性能を達成し得ることを示した．

本研究は，プログラマに対して広域分散計算のための便宜を提供した．しかし，分散計算，特に広域計算には，プログラミングモデル，プログラミングインタフェース以外にも数多くの問題が残されている．具体的には，コンピュータやネットワークといった資源の属性および情報管理，利用者情報の管理，利用者の認証と権限の付与，資源と利用者双方についてのセキュリティ，資源へのプログラムやタスクの割り当て方法，プログラムから実行環境情報へのアクセスなどが挙げられる．今後は，これら分散システムについての研究成果と，本研究をはじめとするプログラミングに関する成果が統合されることで，種類，数共に多くのメタコンピュータが構築されていくことだろう．

謝辞

本論文は，私が早稲田大学 大学院理工学研究科 村岡洋一研究室に在籍していた期間に行われた研究をまとめたものです．研究を進める過程で多くの方々のご指導，ご支援を賜りました．心から感謝致します．

村岡洋一教授は，私が教授の研究室に所属して以来，6 年間にわたってご指導頂きました．先生は常に科学技術と社会の向かう未来を鋭敏な感覚でとらえて私に示し，また，それを自らデザインしようという姿勢でもって，私の研究者としてあり方を決定付けました．そして，一流を見なさい，目標を高く設定しなさい，というご助言と，実際に一流を見せてくださるそのご指導によって，私を含めた学生の前には広大な可能性が開けていることを教えて下さいました．先生のご指導，ご支援なしに，本研究はあり得ませんでした．

筧捷彦教授，中島達夫助教授，山名早人助教授には，本論文の審査をして頂きました．また，大石進一教授，後藤滋樹教授からは，審査の場にて貴重な助言や示唆を頂きました．上田和紀教授には，学部生の頃より研究環境などの面でお世話になりました．頂いた助言や示唆は，本論文をまとめるためだけでなく，私の今後の研究活動においても有益な，大変貴重なものでした．

村岡研究室の諸氏には，議論，助言，示唆という形で，また，研究室での生活でも非常にお世話になりました．水野裕識氏，後藤真孝氏には，多くの助言を頂いただけでなく，公私両面にわたり様々な相談をさせても頂きました．福盛秀雄氏，三浦敏孝氏には，早稲田大学メディアネットワークセンター助手として業務でお世話になった他，並列処理やコンピュータアーキテクチャについて多くの有意義な議論をさせて頂きました．内野聡氏には，私が研究活動を始める際にご指導頂きました．氏の並列化コンパイラが私のスタート地点です．菅原健一氏，浜中征志郎氏，根山亮氏，秋岡明香氏とは，分散並列処理の研究に共に取り組ませて頂きました．興梠正克氏からは，私と共に研究室に所属した 6 年間にわたり，同僚としてよい刺激を頂きました．園田智也氏の積極的に産業社会に関わろうと

する姿勢は，研究者と社会，経済の関係について考えるきっかけを与えて下さいました．高橋里恵氏は，私を含めた村岡研究室の活動を常に支えて下さいました．

太田和夫氏，青木和麻呂氏は，NTT 情報通信研究所（現情報流通プラットフォーム研究所）に 2 度にわたり短期滞在する機会を与えて下さいました．日常の研究対象とは異なり，情報セキュリティに取り組むという稀な時間でした．また，社会における研究活動の意義について考えるきっかけを与えて下さいました．

本研究の一部は，日本学術振興会 科学研究費補助金，早稲田大学 特定課題研究助成金によって遂行されました．また，ある部分は，日本学術振興会 未来開拓学術研究推進事業の一部として遂行されました．未来開拓「知能情報・高度情報処理」の分散・並列スーパーコンピューティングのソフトウェアの研究 プロジェクトに参加された方々には，プロジェクトの活動を通じて様々にご支援を頂きました．

他にも，多くの方々からご支援，ご助言を頂きました．私の研究活動を可能にして下さった多くの方々に，深く感謝致します．

最後に，今日まで私の研究生活を見守って下さった横須賀と静岡の両親，そして，あらゆる面から私を支え続けてくれる妻の麻里に感謝致します．

参考文献

[1] David Abramson, John Giddy, and Lew Kotler. Nimrod/g: Killer application for the global grid? In *Proc. International Parallel and Distributed Processing Symposium (IPDPS) 2000*, pp. 520–528, May 2000.

[2] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A language for resource-aware mobile programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems*, Vol. 1222, pp. 111–130. Springer Verlag, 1997.

[3] Ole Agesen. GC points in a threaded environment. Technical Report SMLI TR-98-70, Sun Microsystems, Inc., December 1998. http://www.sun.com/research/jtech/pubs/.

[4] Ole Agesen and David Detlefs. Finding references in Java^{TM} stacks. In *Proc. of OOPSLA97 Workshop on Garbage Collection and Memory Management*, October 1997.

[5] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java^{TM} Virtual Machines. In *Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI'98)*, pp. 269–279, June 1998.

[6] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journals, Java Performance Issue*, Vol. 39, No. 1, February 2000.

[7] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: a single system image

of a jvm on a cluster. In *Proc. of 1999 International Conference on Parallel Processing (ICPP-99)*, September 1999.

[8] Ken Arnold, James Gosling, and David Holmes. *Java Language Specification, Third Edition*. Addison Wesley, 2000.

[9] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, Vol. 15, No. 1, pp. 29–36, 1995.

[10] Henri Casanova and Jack Dongarra. NetSolve: A network server for solving computational science problems. In *Proc. Supercomputing '96*. IEEE, November 1996.

[11] John H. Crawford, Patrick P. Gelsinger(岩谷宏訳). 80386 プログラミング (原題 Programming The 80386). 工学社, July 1988.

[12] Bozhidar Dimitrov and Vernon Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transaction on Parallel and Distributed Systems*, Vol. 9, No. 5, pp. 459–469, May 1998.

[13] Distributed Computing Technologies, Inc. distributed.net. http://www.distributed.net/.

[14] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS–89–85, Computer Science Department, University of Tennessee, January 2001. http://www.netlib.org/benchmark/performance.ps.

[15] Jack J. Dongarra and Reed Wade. Linpack benchmark — Java version. http://www.netlib.org/benchmark/linpackjava/.

[16] Entropia, inc. http://www.entropia.com/.

[17] M. Raşit Eskicioğlu. Design issues of process migration facilities in distributed system. *IEEE Technical Comittee on Operating Systems Newsletter*, Vol. 4, No. 2, pp. 3–13, Winter 1989. Reprinted in *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press.

[18] Kafka: Java 用マルチ・エージェント記述ライブラリ. http://www.fujitsu.co.jp/hypertext/free/kafka/jp/.

[19] Message Passing Interface Forum. *Document for a standard message-passing*

interface, March 1994.

[20] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[21] Ian Foster and Nicholas T. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proc. SC98: International Conference for High Performance Networking and Computing*, November 1998.

[22] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.

[23] Stefan Fünfrocken. Transparent migration of Java-based mobile agents. In *Proc. of 2nd Int'l Workshop on Mobile Agents 98(MA'98)*, pp. 26–37, September 1998.

[24] The gnu compiler for the java programming language. http://gcc.gnu.org/java/.

[25] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[26] General Magic, Inc. Odyssey information. http://www.genmagic.com/technology/odyssey.html.

[27] Satoshi Hirano. HORB: Distributed execution of Java programs. In *Proceedings of World Wide Computing and Its Applications*, March 1997.

[28] IBM Aglets Software Development Kit. http://www.trl.ibm.co.jp/aglets/index-j.html.

[29] IEEE standard 754-1985 for binary floating-point arithmetic, 1985.

[30] Intel Microprocessor Research Labs. Open runtime platform, September 2000. http://intel.com/research/mrl/orp/.

[31] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, , and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proc. of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, October 2000.

[32] Java Grande Forum Numerics Working Group. Improving Java for numerical computation, October 1998. http://math.nist.gov/javanumerics/.

[33] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transaction on Computer Systems*, Vol. 6, No. 1, pp. 109–133, February 1988.

[34] Katarzyna Keahey and Dennis Gannon. PARDIS: CORBA-based architecture for application-level parallel distributed computation. In *Proceedings of SC97: High Performance Networking & Computing*, November 1997.

[35] Albrecht Kleine. TYA Archive. http://sax.sax.de/~ adlibit/.

[36] Danny Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley Longman, Inc., 1998.

[37] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Addison Wesley, 1997.

[38] Hidemoto Nakada, Mitsuhisa Sato, and Satoshi Sekiguchi. Design and implementations of Ninf: towards a global computing infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 1999.

[39] Ryo Neyama. RyORB—Ryo's object request broker for Java, 1998. http://www.shogi.ne.jp/RyORB/.

[40] Object Management Group, Inc. CORBA/IIOP 2.4 specification, October 2000.

[41] ObjectSpace, Inc. Voyager. http://www.objectspace.com/products/Voyager/.

[42] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An open-ended, reflective JIT compile framework for java. In *Proc. of 14th European Conference on Object-Oriented Programming (ECOOP '2000)*, June 2000.

[43] Roldan Pozo and Bruce Miller. SciMark 2.0, 1999. http://math.nist.gov/scimark2/.

[44] M. Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware mobile programs. In *Proceedings of USENIX'97*, January 1997.

[45] SETI@home. http://setiathome.ssl.berkeley.edu/.

[46] Tatsuro Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In *Proc. 3rd Int'l Conf. on Coordination Models and Languages(Coordination99)*, April 1999.

[47] Kazuyuki Shudo. shuJIT—JIT compiler for Sun JVM/x86, 1998. http://www.shudo.net/jit/.

[48] Kazuyuki Shudo and Yoichi Muraoka. Efficient implementation of strict floating-

point semantics. In *Proc. of 2nd Workshop on Java for High-Performance Computing (in conj. with ICS'00)*, pp. 27–38, May 2000.

[49] Kazuyuki Shudo and Yoichi Muraoka. Asynchronous migration of execution context in Java virtual machines. *Future Generation Computer Systems (FGCS)*, 2001.

[50] Standard Performance Evaluation Corporation. SPEC JVM98, 1998. http://www.spec.org/osg/jvm98/.

[51] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the ibm java just-in-time compiler. *IBM Systems Journals, Java Performance Issue*, Vol. 39, No. 1, February 2000.

[52] Inc. Sun Microsystems. Java^{TM} IDL Documentation. http://www.javasoft.com/products/jdk/idl/.

[53] Sun Microsystems, Inc. *Java^{TM} Object Serialization Specification*, 1997.

[54] Sun Microsystems, Inc. Updates to the Java language specification for JDK release 1.2 floating point, 1999. http://java.sun.com/docs/books/jls/strictfp-changes.pdf.

[55] Marvin M. Theimer and Barry Hayes. Heterogeneous process migration by recompilation. In *Proc. IEEE 11th International Conference on Distributed Computing Systems*, pp. 18–25, 1991. Reprinted in *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press.

[56] Transvirtual Technologies, Inc. Kaffe. http://www.transvirtual.com/kaffe.htm.

[57] Matt Welsh and David Culler. Jaguar: Enabling efficient communication and I/O from Java. In *Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications*, December 1999.

[58] James E. White. *Telescript Technology: The Foundation of the Electronic Marketplace.* General Magic, Inc., 1994.

[59] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java^{TM} system. In *Proc. The Second Conference on Object–Oriented Technology and Systems (COOTS)*, pp. 219–231, 1996.

[60] 首藤一幸. strictfp の実装, 1999.

http://www.shudo.net/java-grandprix99/strictfp/.

[61] 小鷲英一. MMX テクノロジ 最適化テクニック. アスキー出版, July 1997.

[62] 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, August 1998.

[63] John Bloomer(森島晃年訳). RPC プログラミング. アスキー, 1995.

[64] 石崎一明, 川人基弘, 今野和浩, 安江俊明, 竹内幹雄, 小笠原武史, 菅沼俊夫, 小野寺民也, 小松秀昭. Java Just-In-Time コンパイラにおける最適化とその評価. 電子情報通信学会技術研究報告, CPSY99-64, pp. 17–24, August 1998.

研究業績

論文

種別	題名	発行年月	掲載誌名	著者
論文誌				
1 ◯	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	1999 年 8 月	情報処理学会論文誌: プログラミング, VOl.40, No.SIG 7 (PRO 4), pp.66–79	首藤一幸 根山亮 村岡洋一
2 ◯	Asynchronous Migration of Execution Context in Java Virtual Machines	掲載決定	Future Generation Computer System (FGCS)	Kazuyuki Shudo Yoichi Muraoka
国際会議				
3	Evaluation of Gigabit Ethernet with Java/HORB	1998 年 8 月	Computing in High Energy Physics (CHEP'98)	Y. Yasu H. Fujii Y. Igarashi E. Inoue H. Kodama A. Manabe Y. Watase Y. Nagasaka M. Nomachi S. Hirano H. Takagi K. Shudo T. Arai L. Sarmenta C. Thomborson R. Nicolescu M. Duc

種別	題名	発行年月	掲載誌名	著者
4 ◯	Noncooperative Migration of Execution Context in Java Virtual Machines	1999 年 6 月	Proc. 1st Annual Workshop on Java for High-Performance Computing, pp.49–57	Kazuyuki Shudo Yoichi Muraoka
5 ◯	Efficient Implementation of Strict Floating-Point Semantics	2000 年 5 月	Proc. 2nd Workshop on Java for High-Performance Computing, pp.27–38	Kazuyuki Shudo Yoichi Muraoka
6 ◯	MetaVM: A Transparent Distributed Object System	2000 年 6 月	Proc. The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), VOL.II, pp.879–882	Kazuyuki Shudo Yoichi Muraoka
7	A Model for Stream Calculation	2000 年 6 月	The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), VOL.II, pp.737–743	Hirobumi Sugino Kazuyuki Shudo Masayoshi Sekiguchi Yoichi Muraoka
8	Grid-based Asynchronous Migration of Execution Context in Java Virtual Machines	2000 年 8 月	Proc. European Conference on Parallel Computing 2000 (Euro-Par 2000), pp.22–33	Gregor von Laszewski Kazuyuki Shudo Yoichi Muraoka

講演

種別	題名	発行年月	掲載誌名	著者
研究会				
1	隣接 S-box の影響を一部考慮した DES の経路探索	1996 年 7 月	電子情報通信学会技術研究報告 ISEC-96-13, pp.61–72	青木和麻呂 首藤一幸
2 ◯	分散環境を対象とした並列プログラミング環境 MC	1997 年 8 月	情報処理学会研究報告 97-HPC-67, pp.91–96	首藤一幸 菅原健一 浜中征志郎 村岡洋一
3 ◯	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	1998 年 5 月	電子情報通信学会技術研究報告 CPSY98, pp.39–46	首藤一幸 村岡洋一
4 ◯	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	1999 年 3 月	情報処理学会 プログラミング研究会	首藤一幸 根山亮 村岡洋一
5	使いやすさ・透過性・拡張性の高い分散オブジェクトシステム Ry-ORB	1999 年 3 月	情報処理学会 プログラミング研究会	根山亮 首藤一幸 村岡洋一
6	グローバルコンピューティングのためのストリーム計算実行時システム	2000 年 8 月	情報処理学会研究報告 2000-HPC-82, pp.149–154	関口真良 首藤一幸 杉野博文 村岡洋一
一般講演				
7 ◯	分散環境をターゲットとした自動並列化コンパイラ及び実行環境	1996 年 3 月	情報処理学会第 52 回全国大会講演論文集 (6), pp.87–88	首藤一幸 菅原健一 浜中征志郎 村岡洋一
8	分散環境でのストリーム処理のための並列コード生成	1996 年 9 月	情報処理学会 第 53 回全国大会	浜中征志郎 首藤一幸 菅原健一 村岡洋一
9	並列化コンパイラによるタスク並列とデータ並列の統合処理	1996 年 9 月	情報処理学会 第 53 回全国大会	菅原健一 首藤一幸 浜中征志郎 村岡洋一

種別	題名	発行年月	掲載誌名	著者
10 ◯	分散・並列化コンパイラ MC の開発基盤 —中間表現—	1997 年 9 月	情報処理学会第 55 回全国大会講演論文集 (1), pp.339–340	首藤一幸 村岡洋一
11	実行環境の情報を利用した並列化コンパイラと実行時システム	1997 年 9 月	情報処理学会第 55 回全国大会講演論文集 (1), pp.343–344	浜中征志郎 首藤一幸 菅原健一 村岡洋一
12 ◯	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	1999 年 3 月	情報処理学会第 58 回全国大会講演論文集 (1), pp.363–364	首藤一幸 村岡洋一
13	疎なネットワーク環境における自律分散型・動的負荷分散システム	1999 年 3 月	情報処理学会第 58 回全国大会講演論文集 (3), pp.357–358	秋岡明香 首藤一幸 根山亮 村岡洋一
14 ◯	Java 実行時コンパイラへの strictfp の実装	2000 年 3 月	情報処理学会第 60 回全国大会講演論文集 (1), pp.267–268	首藤一幸 村岡洋一

その他

種別	題名	発行年月	掲載誌名	著者
ポスター発表				
1 ◯	Java 上のスレッド移送システムと移動エージェントへの応用	1998 年 6 月	並列処理シンポジウム JSPP'98 ポスターセッション	首藤一幸 村岡洋一
2 ◯	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	1999 年 6 月	並列処理シンポジウム JSPP'99 ポスターセッション	首藤一幸 村岡洋一