動的双方向変換技術に基づいた異機種オブジェクトモビリティの実現法

松原克弥[†] 板橋一正[†] 森山豊[†]
染谷祐一[†] 加藤和彦[†,††]
関口龍郎[†††] 米澤明憲[†††]

現在，世界各地で，モーバイルオブジェクトやモーバイルエージェントなどのオブジェクトモビリティを実現するシステムの研究・開発がさかんに行われている．それらの多くは，セキュリティ保全のための機構の実現や異機種性に対する対処が比較的容易であるなどの理由で，特定の機種に依存しないバイトコード・インタプリタ方式をモーバイルオブジェクト実現の要として利用している．我々は，実行時性能およびプログラミング言語との独立性に注目し，ネイティブコードを用いてモーバイルオブジェクトの実行を行う．本実現法では，異機種間でのモーバイルオブジェクト実現のために，オブジェクトに含まれるプログラムコード，データ，実行状態のそれぞれに機種に依存しない正準表現を設定し，ネイティブ表現と正準表現の間の動的な双方向変換技術を実現する．また，変換モジュールをモーバイルオブジェクトとして実現することで，任意の正準表現を動的な選択を可能にする．

Implementing Heterogeneous Object Mobility with the Dynamic Bidirectional Translation Scheme

Katsuya Matsubara,[†] Kazumasa Itabashi,[†]
Yutaka Moriyama,[†] Yuichi Someya,[†] Kazuhiko Kato,[†,††]
Tatsurou Sekiguchi[†††] and Akinori Yonezawa[†††]

Mobile objects or mobile agents is one of the most active research fields in the network-related system software. Most mobile object systems are based on the interpreter-based execution of scripting code or bytecode, and this basis greatly simplifies the management of platform-heterogeneity and security-related issues. We claim that native-code-based execution is more appropriate for execution performance and programming language neutrality. In this paper we propose a framework wherein object mobility is implemented with native machine codes on heterogeneous environments. We have designed and implemented the dynamic bidirectional translation scheme. To handle platform-heterogeneity, the scheme translates between a canonical representation and multiple native representations in the middle of moving objects. Also, the scheme allows multiple canonical representations to coexist in a system.

1. はじめに

現在，世界各地で，モーバイルオブジェクトやモーバイルエージェントの研究開発がさかんに行われている[15),16)]．現在のモーバイルオブジェクト・システムやモーバイルエージェント・システムの多くは，特定の計算機アーキテクチャに依存しない抽象機械コードをインタプリタ・ソフトウェアを用いて解釈実行する**バイトコード・インタプリタ方式**をオブジェクトモビリティ実現の要として利用している．バイトコード・インタプリタ方式は，実装が容易であることや，異機種環境へ対応しやすい利点を持つ．また，インタプリタがオブジェクトのプログラムコードを解釈しながら実行を進めるため，オブジェクトの実行を監視することは比較的容易である．逆に，プログラムコードを解釈するオーバヘッドをともなうため，実行時に計算機性能を最大限に活用することが困難である．また，これらのシステムの多くが，特定のプログラミング言語を使用することを仮定している．Emerald[7)]，Telescript[18)]，Obliq[2)] といったモーバイルオブジェクト・システムでは，それぞれ新たに設計された専用のプログラミング

† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba
†† 科学技術振興事業団
Japan Science and Technology Corporation
††† 東京大学大学院理学系研究科
Department of Information Science, Faculty of Science, University of Tokyo

言語を使用してオブジェクトを記述する．Aglets[11]，Voyager[6]，Concordia[17] に代表される Java 言語システム上のモーバイルオブジェクト・システムは，Java 言語をオブジェクト記述言語として用いる．

我々は，上述とは異なるアプローチでモーバイルオブジェクト・システムの研究開発を進めている[9),10),12)]．現在我々が研究開発を進めている PLANET モーバイルオブジェクト・システムは，モーバイルオブジェクトの実行にネイティブコードを用いる**ネイティブ実行方式**に基づいている．ネイティブコードは，CPU によって直接実行されるため，実行時性能を最大限に引き出すことが可能である．また，ネイティブコードが特定のプログラミング言語に依存しないため，従来のアプリケーションをモーバイルオブジェクト・システム上に移植する際のオーバヘッドを最小限にすることが可能である．

本稿では，PLANET の特徴の 1 つであるネイティブコードによるモーバイルオブジェクトの利点を享受しながら，複数の計算機アーキテクチャが混在する異機種環境においてモーバイルオブジェクトを実現する方法について述べる．本実現法は，以下の 3 つの特徴を持つ．第 1 は，特定のオペレーティングシステムやプログラミング言語に依存しないミドルウェアレベルで実現していることである．**図 1** は，本実現システムの構造を示している．ユーザは，目的に応じて異なるプログラミング言語を用いてモーバイルオブジェクトを記述することが可能となる．第 2 の特徴は，プログラムコード，データ，計算状態のそれぞれに対して，ネイティブ表現と正準表現の動的な双方向変換を実現していることである．第 3 の特徴は，動的双方向変換で用いる正準表現形式を動的に選択できることである．これは，動的双方向変換処理モジュールをモーバイルオブジェクトとして実現することにより可能としている．正準表現のモーバイルオブジェクトをロードした際に，適切な変換処理オブジェクトとともにロードし，ネイティブ表現への変換を行う．ユーザは，用途に応じて正準表現を選択することができ，モーバイルオブジェクトごとに異なる正準表現を用いることも可能である．

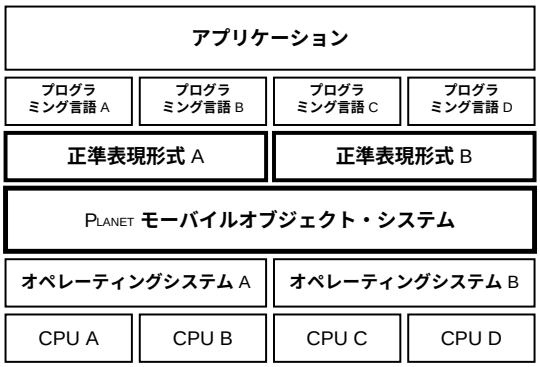

図 1 構造化アーキテクチャ
Fig. 1 Layered architecture.

以下，本稿は次のように構成されている．2 章では，3 章以降の準備として，PLANET システムの概要を簡潔に述べる．3 章では，異機種オブジェクトモビリティの実現方式について述べる．4 章では，3 章で述べた実現方式に基づいて，SPARC アーキテクチャと Intel x86 アーキテクチャ上で行った実装について述べる．5 章では，4 章で実装したプロトタイプシステムを用いた実験について述べる．6 章では，本実現法に関する関連研究について述べる．最後に 7 章で，まとめと今後の研究課題について述べる．

2. PLANET システム

PLANET のシステムモデルは，モーバイルオブジェクト，DSR（Distributed Shared Repository；分散共有格納庫の略），プレース，プロテクションドメイン，オブジェクトポートの 5 つの基本抽象概念を用いて説明される．**図 2** は，PLANET のシステムモデルを示している．

計算処理の対象となるデータおよびデータに対する操作を密閉し，仮想記憶領域から分離可能にしたものをモーバイルオブジェクトと呼ぶ．PLANET システムで扱うモーバイルオブジェクトには 4 種類あり，セグメントの数に応じて，Type I，Type II，Type III，Type IV と呼ぶ．**図 3** に，各モーバイルオブジェクト

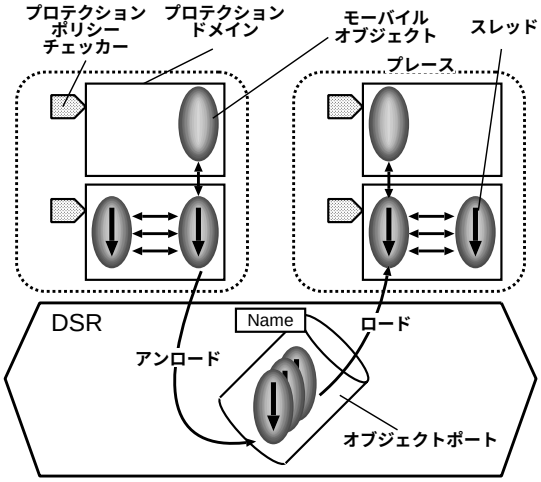

図 2 PLANET システムのシステムモデル
Fig. 2 Basic system model.

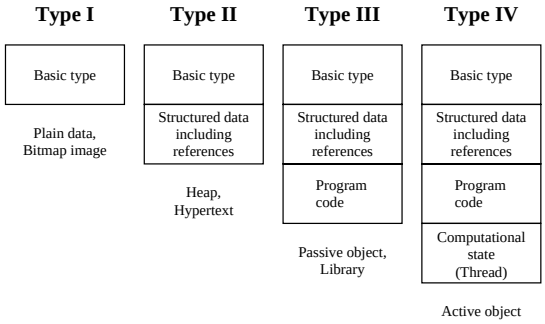

図 3 モーバイルオブジェクトの種類
Fig. 3 Object types.

に含まれるセグメントを示す．Type I モーバイルオブジェクトは，基本データセグメントのみを持つデータオブジェクトである．文書データ，ビットマップデータなどがこのモーバイルオブジェクトとして扱われる．Type II モーバイルオブジェクトは，基本データに加えてデータへの参照（ポインタ）を含む構造化データを持つオブジェクトである．ハイパーテキストなどの構造を持つデータなどがこのモーバイルオブジェクトとして扱われる．Type III モーバイルオブジェクトは，オブジェクト内にスレッドが束縛されていないパッシブ・モーバイルオブジェクトである．内部にテキストセグメントとデータセグメントを持ち，ライブラリなどの実行可能モジュールがこのタイプのオブジェクトで表現できる．Type IV モーバイルオブジェクトは，オブジェクト内にスレッドが束縛されていて，自らが能動的に活動するアクティブ・モーバイルオブジェクトである．テキストセグメント，データセグメントに加えて CPU-State セグメントを持つ．

Planet のモーバイルオブジェクトは DSR またはプレースのいずれかに存在している．DSR は広域ネットワークと永続的記憶空間を抽象化したものであり，プレースはローカルエリアネットワークと揮発的記憶空間を抽象化したものである．モーバイルオブジェクトを DSR からプレースに移動する操作をロード操作，逆にプレースから DSR に移動させる操作をアンロード操作と呼ぶ．ネットワーク環境において DSR 空間は論理的に 1 つしか存在しないのに対し，プレースは数多く存在する．オブジェクトの実行は，オブジェクトがプレース上にあるときにのみ可能であり，DSR 上にあるときは可能でない．

DSR において，モーバイルオブジェクトはオブジェクトポートと呼ばれるキューに格納される．キューは，システム内で一意で位置独立な名前で識別され，モーバイルオブジェクトのロード /アンロード時に指定される．また，キューにはアクセス権が設定されており，アクセス権を持つプレースのみがキューへの操作が可能でなる．

プロテクションドメインは，プレース上の情報保護の単位である．1 つのプロテクションドメインは必ず 1 つのプレースに属し，同時に 2 つ以上のプレースに属することはできない．オブジェクトがあるプレース上にロードされているとき，オブジェクトはそのプレース上の 1 つ以上のプロテクションドメインに必ず関連づけられている必要がある．個々のプロテクションドメインは独立した仮想アドレス空間を持ち，プロテクションドメインを乗り越えた情報アクセスを行うことができないようメモリ管理ハードウェアを用いて厳格に保護されている．各プロテクションドメインに対応して動作しているプロテクションポリシー・チェッカは，ユーザのプロテクション方針に基づいて，システム資源や他のプロテクションドメインへのアクセスを監視する．

Planet システムを用いると，分散処理アプリケーションを構築するうえで必要となる機能を系統的に実現することができる．モーバイルオブジェクトを実行中に移動させることにより，ネットワーク通信のための記述を行うことなしに，分散処理を記述することができる．プログラマは，ネットワーク通信の処理を記述する代わりに，単一の計算機で動作するオブジェクトの適当な箇所に移動のためのプリミティブ呼び出しを挿入するだけでよい．また，オブジェクトの実行状態を含むことができる Type IV モーバイルオブジェクトを使うことで，移動時の実行状態の保存・復元のためのコードを明示的に記述する必要がない．文献 10) では，Planet モーバイルオブジェクトを使って広域ネットワークでの情報収集を効率的に行う，Web サーチロボットの構築法について述べている．

3. 異機種オブジェクトモビリティの実現方式

異機種環境では，オペレーティングシステム API と計算機アーキテクチャの 2 種類の異機種を考える必要がある．

前者の異機種性は，システムコールやオペレーティングシステムが標準で提供しているシステムライブラリの違いを指す．オペレーティングシステム API の異機種に対しては，特定のオペレーティングシステムに依存しない API セット（以下，Planet API と呼ぶ）を提供することで対応する．モーバイルオブジェクトは，システム資源や他のプロテクションドメインのモーバイルオブジェクトへのアクセスを Planet API

を用いて記述する．PLANET API セットは，各計算機アーキテクチャごとに実装し，モーバイルオブジェクトを扱うアプリケーションにリンクする．PLANET API の処理は，システムライブラリ名，引数の型や数，手続き呼び出し方法の違い☆を変換して，各計算機アーキテクチャのシステムコールを呼び出す．プロテクションポリシー・チェッカは，システム資源や他のプロテクションドメインに対する PLANET API 以外のアクセスを制限する（文献 8) 参照）．

後者の計算機アーキテクチャの異機種性は，計算を行う CPU のアーキテクチャが計算機ごとに異なることを指す．以下，本稿では，後者の計算機アーキテクチャの異機種性に対する技術的課題の解決法について述べる．

3.1 基本ソフトウェアアーキテクチャ

本実現法は，特定の計算機アーキテクチャに依存しないオブジェクトの正準表現を設定して，ネイティブ表現と正準表現を動的にかつ双方向に変換する**動的双方向変換**を実現する．移動時は正準表現を用いてモーバイルオブジェクトを表現し，実行時はロードされた計算機のネイティブ表現に変換したモーバイルオブジェクトを使う．また，用いる正準表現を固定せず，上述の動的双方向変換を実現可能な任意の正準表現を選択することが可能となるようにシステム全体を構造化している（図 1 参照）．

異機種オブジェクトモビリティのための正準表現を使った動的双方向変換は，コンパイラバックエンドとオブジェクト表現変換器の 2 つのモジュールにより実現する．**図 4** は，本実現法のソフトウェアアーキテクチャを示している．コンパイラバックエンドは，正準表現のプログラムコードと初期化データを生成するためのモジュールである．オブジェクトのソースプログラムから正準表現のモーバイルオブジェクトを生成するコンパイル時に実行する．オブジェクト表現変換器は，モーバイルオブジェクトに対して，正準表現と各計算機アーキテクチャのネイティブ表現形式の間の動的な双方向変換を行うモジュールである．モーバイルオブジェクトがロードされるときに正準表現からネイティブ表現への変換を行い，アンロード時はモーバイルオブジェクトの状態を表すデータと計算状態の各セグメントを正準表現に変換する．通常変更が行われないテキストセグメントは，ロード時にネイティブ表現

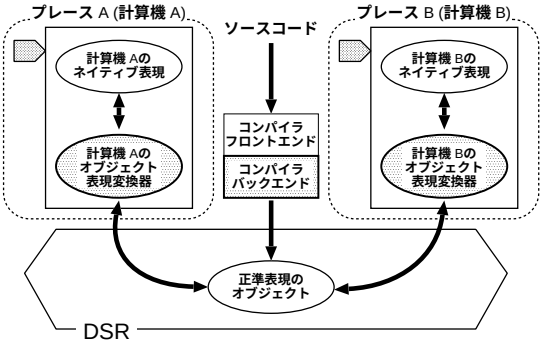

図 4 異機種間オブジェクトモビリティを実現する基本ソフトウェアアーキテクチャ
Fig. 4 Basic software architecture for platform-heterogeneity.

へ変換前の正準表現をキャッシュしておき，アンロード時に利用する．正準表現に変換されたモーバイルオブジェクトは，PLANET が提供する DSR 機能により，オブジェクト実行中に各計算機と永続空間の間でロードおよびアンロードを行うことができる．また，オブジェクト表現変換器自身を PLANET モーバイルオブジェクトとして実現することで，モーバイルオブジェクトのロード時に，適切なオブジェクト表現変換器をプレースにロードする機構を実現する．

本実現法では，上述のオブジェクト表現変換器をモーバイルオブジェクトとして実現する．オブジェクト表現変換器をあらかじめシステム内の全プレースに配布しておくのではなく，モーバイルオブジェクトをロードする際に，ロードしたモーバイルオブジェクトが用いている正準表現形式と実行する計算機アーキテクチャに対応したオブジェクト表現変換器を動的にダウンロードする．

3.2 モーバイルオブジェクトの正準表現形式

モーバイルオブジェクトの移動時に用いる正準表現は，モーバイルオブジェクトに含まれるプログラムコード，データ（ヒープ），計算状態を表すレジスタ，スタックとプログラムカウンタのそれぞれに設定する必要がある．任意の正準表現とネイティブ表現の間を双方向に変換することは，非常に困難である．本実現法では，動的双方向変換を実現するために，正準表現の仕様が以下のような条件を満たしている必要がある．

（条件 1）プログラムコードの非更新性 アンロードするモーバイルオブジェクトは，オブジェクト内のプログラムコードが実行中に変更されていない必要がある．1 つの計算機アーキテクチャのネイティブコードに対して行われた任意の変更を，正準表現コードや他の計算機アーキテクチャのネイ

☆ たとえば，SPARC アーキテクチャでは，引数の最初の 6 つはレジスタを使い，残りはスタックに積んで手続き呼び出しを行う．Intel x86 アーキテクチャでは，すべての引数をスタックに積んで手続き呼び出しを行う．

ティブコードに反映することは困難であるため，この性質が必要となる．

（条件 2）再配置可能性 オブジェクト内のアドレス空間依存情報（ポインタ）を再配置できる必要がある．Planet のモーバイルオブジェクトは，各計算機アーキテクチャの仮想記憶空間内の任意の位置にロードされるため，データ参照に用いられるポインタと分岐と手続き呼び出しに指定されるアドレスを，ロードされた位置に適応するように調整する必要がある．

（条件 3）データ型決定性 オブジェクト内のデータの配置と型に関する情報が取得できる必要がある．データの配置と型に関する情報は，各計算機アーキテクチャで，データ型に基づいた適切なネイティブ表現へ変換するのに必要となる．

（条件 4）レジスタ割当て決定性 計算状態の移動を実現するために，アンロード時に移動すべきレジスタを特定できる必要がある．この条件は，正準表現とネイティブ表現の両方のレジスタの割当て情報が取得できれば解決できる．

（条件 5）スタックの分析可能性 ネイティブ表現のスタックから，スタックの正準表現変換に必要なスタックポインタ，フレームポインタ，リターンアドレス，ローカル変数と引数を含むスタックフレームが取得できる必要がある．スタックポインタとフレームポインタは，スタック内の各フレームを解析するために用いる．リターンアドレスは，各スタックフレームを使っている手続きを特定するために使う．ローカル変数と引数は，手続きの計算状態を再現するために必要である．

（条件 6）コード内位置決定性 正準表現とネイティブ表現の間で，オブジェクト内の手続き呼び出しの位置とアンロードが行われる位置が変換できる必要がある．手続き呼び出し位置の変換は，スタックフレームに含まれるリターンアドレスの変換に必要となる．アンロード位置の変換は，ネイティブ表現されたプログラムカウンタを正準表現コードでの位置に変換するために用いる．この条件を実現するためには，ネイティブ表現に変換されたプログラムコードにおいても，正準表現での手続きの構造やアンロードが行われる位置が維持されている必要がある．

3.3 動的双方向変換機構

本節では，3.2 節で述べた条件を満たす正準表現を用いた動的双方向変換の実現方法について述べる．本実現法は，オブジェクト記述に用いるプログラミング言語のコンパイラバックエンドと，モーバイルオブジェクトを移動する計算機アーキテクチャのオブジェクト変換器を実装することで，動的双方向変換を実現する．

3.3.1 コンパイラバックエンド

コンパイラバックエンドでは，正準表現オブジェクトの生成と動的双方向変換のための正準表現に関する情報の出力の 2 つの処理を実現する必要がある．

生成する正準表現オブジェクトは，3.2 節で述べた再配置可能性（条件 2）を満たさなければならない．本実現では，プログラムコードの再配置のために，データ参照に指定するアドレスと分岐や手続き呼び出しに指定するアドレスで実行時に修飾可能な相対アドレッシングを使う．データの再配置は，プログラミング言語処理系が持つデータの型情報からポインタを特定することで実現する．ポインタに関する位置情報は，生成するオブジェクトに付加して，ロード時にポインタを適切な値に調整するために使用する．

動的双方向変換のために出力する情報は，データの型情報とレジスタの割当て情報である．これらの情報は，データ型決定性（条件 3）とレジスタ割当て決定性（条件 4）を実現するために必要となる．コンパイラフロントエンドが解析したデータ型情報に，コンパイラバックエンドでそれらのデータに割り当てたメモリ領域やレジスタに関する情報を付加して出力する．

3.3.2 オブジェクト表現変換器

オブジェクト表現変換器は，モーバイルオブジェクトがロードされるときに正準表現からネイティブ表現への変換を行い，アンロードされるときにネイティブ表現から正準表現への逆変換を行う．本オブジェクト表現変換器は，モーバイルオブジェクトに含まれるプログラムコード，データ，計算状態の 3 つの表現変換器に分けて実現する（**図 5** 参照）．

プログラムコード表現変換器

プログラムコード表現変換器では，プログラムコードの表現変換とネイティブレジスタに関する割当て情

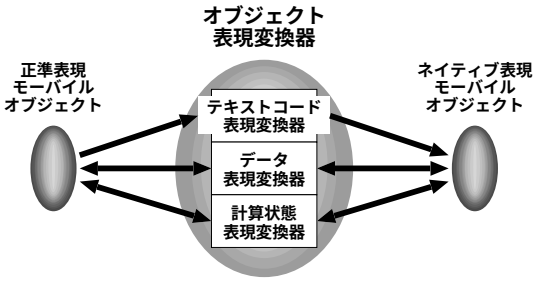

図 5 オブジェクト表現変換器
Fig. 5 Object representation converter.

報の作成を実現しなければならない．

本変換器は，モーバイルオブジェクトをロードするときに正準表現のプログラムコードをネイティブ表現に変換する．その際，コード内位置決定性（条件 6）を満たすために，正準表現とネイティブ表現の間での手続き呼び出し位置とアンロード位置の割当て情報を出力する必要がある．手続き呼び出しの位置は，手続き呼び出し命令の変換時に特定する．アンロード位置は，アンロード・プリミティブ呼び出し位置から特定できる．モーバイルオブジェクトをアンロードするときは，プログラムコードの非更新性（条件 1）の条件より，ロード時の変換が行われる前のプログラムコードを再利用することで正準表現のプログラムコードを生成する．

ネイティブレジスタの割当て情報の出力は，レジスタ割当て決定性（条件 4）を満たすために必要となる．ネイティブコード変換時，正準表現のレジスタをネイティブレジスタへ割り当てる際の情報を基に作成する．

データ表現変換器

データ表現変換器では，整数や浮動小数点などの型付けされたデータの表現方法とバイトオーダーの変換，および表現能力（サイズ）の違いから起こるデータ間の相対位置の変化に対する処理を実現する必要がある．静的に割り当てられるデータは，コンパイラバックエンドで作成するデータの配置情報と型情報を用いて上述の変換を行う．実行中に動的に割り当てられるデータは，Pascal などの型制約の強いプログラミング言語の場合にはコンパイラにより得られる型情報を用いる．C 言語などの型制約の弱いプログラミング言語の場合には，アプリケーションプログラムが実行時に型情報を明に指定する方法を用いることで型情報を取得する．データ位置の変化に対しては，データの参照（ポインタ）の位置および参照先の情報を取得し適切な値に再配置する．

計算状態表現変換器

計算状態表現変換器では，オブジェクトの計算状態を表すプログラムカウンタ，レジスタ，スタックのそれぞれに対し，正準表現とネイティブ表現の双方向変換を実現する．

プログラムカウンタの双方向変換実現の技術的課題は，アンロード時のプログラムカウンタ値が正準表現コードでどの位置を指しているかを特定しなければならないことである．この課題は，コード内位置決定性（条件 6）と Planet のアンロードの仕様を使って解決することができる．現在の Planet では，モーバイルオブジェクトのアンロードを，アンロードするモーバイルオブジェクト内で明にアンロードプリミティブが実行されたときに限定している．この仕様により，ロード時に静的にアンロードプリミティブの位置を解析することで，アンロード時のプログラムカウンタ値から実行されたアンロードプリミティブの位置を特定することができる．特定したアンロードプリミティブの位置は，コード内位置決定性により，正準表現コードで位置に変換することができる．

レジスタの双方向変換実現のためには，アンロード時に正準表現に変換し移動するレジスタを特定しなければならない．この技術的課題は，レジスタ割当て決定性（条件 4）から解決することができる．レジスタ割当て決定性を満たすために，コンパイラバックエンドとプログラムコード表現変換器は，正準表現コードとネイティブコードのそれぞれのレジスタ割当て情報を生成する．アンロードに必要なレジスタの特定は，この 2 つのレジスタ割当て情報から，正準表現でのレジスタに割り当てられているネイティブレジスタを特定することにより実現できる．

スタックの双方向変換では，各計算機アーキテクチャごとに異なるスタック構造から，計算状態の移動に必要なスタックポインタ，フレームポインタ，リターンアドレス，ローカル変数と引数を取得する必要がある．スタックの分析可能性（条件 5）は，これらを取得するためのに必要なスタックの解析を可能にする．アンロード時，ネイティブスタックを解析して，上述の情報を取得する．スタックポインタとフレームポインタは，スタックを解析するために使用する．ローカル変数と引数は，データ表現変換器でデータ型情報に基づいた正準表現への変換を行う．リターンアドレスは，コード内位置決定性（条件 6）により，リターンアドレスをセットした手続き呼び出しの位置を正準表現での位置に変換する．

4. 実　　装

本章では，プログラムコード，データ，計算状態のそれぞれに 3.2 節で述べた条件を満たす正準表現を設定し，Solaris 2.6 オペレーティングシステムが稼働している SPARC および Intel x86 CPU 上での実装について述べる．オペレーティングシステム API の異機種性に対応するための Planet API は現在設計中で，本実装では Solaris 2.6 のシステムコールインタフェースを Planet API として用いている．

プログラムコードの正準表現として，本稿の共著者である関口と米澤が設計した MIC を用いた．MIC 仮想機械コードから各計算機アーキテクチャ用のネイ

ティブコードへの変換器は，SPARC 用と Intel x86 CPU 用が実装されている．MIC 仮想計算機は，16 個の汎用レジスタと 16 個の浮動小数点レジスタ，32 ビットのフラットなアドレス空間，RISC スタイルの命令セットを持つ．データの正準表現は，Sun XDR[14] を用いた．XDR は，計算機アーキテクチャ間のバイトオーダーおよびデータ表現形式と表現能力の違いを吸収する．計算状態の正準表現は，MIC 仮想機械の仕様に基づいたレジスタとスタックを，データの正準表現である XDR を用いて正準化する方法を用いた．

4.1 コンパイラバックエンド

コンパイラバックエンドでは，正準表現オブジェクトの生成，データ型情報の出力と正準表現でのレジスタ割当て情報の出力を実装した．

現在，MIC は，GNU C コンパイラのバックエンドとして実装されており，C 言語および C++ 言語から MIC 仮想機械コードを生成する．3.2 節で述べた再配置可能性（条件 2）を満たす再配置可能コードの生成は，レジスタ相対アドレッシングを用いたコード生成を実装した．レジスタ相対アドレッシングは，PIC（Position Independent Code）技術[5] を用いている．データ内アドレス空間依存情報再配置のためのポインタ位置情報は，フロントエンドコンパイラである GNU C コンパイラが出力するデバッグシンボル情報を解析して，オブジェクト内のポインタの位置を記録したテーブルを作成した．

データ型情報とレジスタ割当て情報は，GNU C コンパイラが出力するデバッグシンボル情報を解析して，データの型と割り当てられたメモリ領域もしくはレジスタに関する情報を記録したテーブルを作成した．

4.2 オブジェクト表現変換器

4.2.1 プログラムコード表現変換器

プログラムコード表現変換器は，正準表現コードからネイティブコードへの変換とネイティブコードにおけるレジスタ割当て情報の出力を実装した．

MIC 仮想機械コードから SPARC アーキテクチャおよび Intel x86 アーキテクチャのネイティブコードへの変換器を，Planet モーバイルオブジェクトとして実装した．実行時は，MIC 仮想機械コードで表現されたモーバイルオブジェクトとともに各計算機アーキテクチャ用のプログラムコード表現変換器をロードして，ネイティブコードに変換してから実行を行う．コード内位置決定性（条件 6）を満たす手続き呼び出しとアンロードプリミティブ・コールの位置情報は，アンロードプリミティブ・コールを含むオブジェクト内の全手続き呼び出し命令の位置を記録したテーブル（以下，手続き呼び出し位置情報テーブルと呼ぶ）を作成した．

用いている MIC の仕様では，正準表現の 32 個のレジスタを静的にネイティブ表現のレジスタに割り当てている．SPARC アーキテクチャでは，i0〜i3，l4〜l7，o0〜o7，fp，sp と f0〜f15 レジスタを正準表現のレジスタに割り当てた．Intel x86 アーキテクチャは，汎用レジスタの数が正準表現のレジスタの数よりも少ないため，ecx，ebx，esi，edi レジスタと残りのレジスタをスタック上のメモリ領域を使用するように割り当てた．ネイティブ表現でのレジスタ割当て情報は，この仕様に従って正準表現のレジスタ割当て情報を変換することで生成した．

4.2.2 データ表現変換器

本実装では，整数，浮動小数，文字型の基本データ型の XDR 表現とネイティブ表現の変換を行うモジュールを作成した．配列や構造体などの複合データ型は，各要素の基本データ型に分解して変換する．静的に割り当てられるデータ領域は，コンパイラバックエンド出力するデータ型情報を基に，各基本型のデータ表現変換モジュールを使って変換を行う．動的に割り当てられる領域は，割当て時にアプリケーションが明に指定したデータ型情報を使って変換を行う．

現在の Planet の実装では，モーバイルオブジェクトの転送にリモートメモリマップ・ファイル機構（文献 12) 参照）を用いている．リモートメモリマップ・ファイル機構では，ロード時にモーバイルオブジェクト全体が転送されるのではなく，実行時にオブジェクト内の必要なデータをセグメント単位で検出し転送する．オブジェクト内の更新されたデータもセグメント単位で管理することで，アンロード時のデータ転送量を最適化している．本データ表現変換器は，リモートメモリマップ・ファイル機構と調和的に統合するために，実行時にセグメント単位でデータ表現変換を行い，転送されるデータと変換オーバヘッドを最小化した．

データ内のアドレス参照に関しては，アドレス空間内のオブジェクトがロードされる位置によって起こるデータの絶対位置の変化と，CPU のワード長などが異なることが原因で起こるデータの相対位置（配置）の変化の 2 種類の技術的課題を解決する必要がある．本実装では，絶対位置の変化を，Planet で実装されている反復的再配置機構（文献 12) 参照）を用いて再配置を行うことで解決した．相対位置の変化については，現在，CPU のワード長が一定であることを仮定することで回避している．

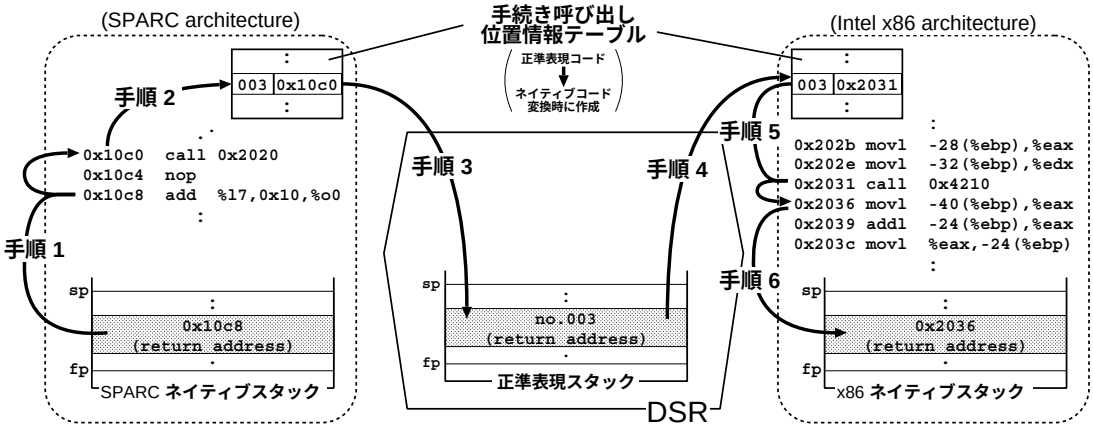

図 6 リターンアドレスの動的双方向変換
Fig. 6 Dynamic bidirectional translation for return address.

4.2.3 計算状態表現変換器

計算状態の正準表現形式は，プログラムコードの正準表現として用いた MIC 仮想機械の仕様に基づいて設計した．計算状態を表すデータは，レジスタ，スタック，プログラムカウンタがある．

レジスタは，プログラムコード表現変換器が生成するレジスタ割当て情報に基づいて変換した．アンロード時は，レジスタ割当て情報に基づいて，正準表現レジスタに割り当てられているネイティブレジスタの値をオブジェクトに保存する．保存したレジスタ値は，データ型情報を使ってデータと同様の XDR 表現変換を行った．

スタックは，スタックを使用している手続きごとにローカル変数とリターンアドレスを含むスタックフレームが積まれている．フレーム内のローカル変数の位置と型情報を得るためには，フレームを使用している手続きを特定する必要がある．本実装では，リターンアドレスからフレームを使用している手続きを特定した．手続きの特定には，リターンアドレスをスタックフレームにセットする命令が手続き呼び出し命令のみである性質を利用する☆．リターンアドレスからプログラムコード中の手続き呼び出し命令を特定し，手続き呼び出し命令のオペランドから呼び出している手続きの先頭アドレスを得た（**図 6** の手順 1 参照）．データ型情報から，関数の先頭アドレスを使って手続き内のローカル変数の型情報を取得し，XDR 表現変換を行う．

リターンアドレスは，各計算機アーキテクチャごとに MIC コードから生成されるネイティブコードの行数が異なるため，再計算する必要がある．本実装では，ネイティブコード生成時にプログラムコード表現変換器で作成した手続き呼び出し位置情報テーブルを使って，正準表現とネイティブ表現の間の変換を行った（図 6 参照）．アンロード時，リターンアドレスから，アドレスをスタックに積んだ手続き呼び出しの場所を特定する（図 6 手順 1）．手続き呼び出しのあるアドレスと上述の手続き呼び出し位置情報テーブルから手続き呼び出しに付けられた番号を得る（図 6 手順 2）．その番号をリターンアドレスの正準表現として用いる（図 6 手順 3）．ロード時は，その計算機アーキテクチャのプログラムコード表現変換器が作成した手続き呼び出し位置情報テーブルを使って，そのネイティブコードでのリターンアドレスを計算する（図 6 手順 4 から 6）．

プログラムカウンタも，リターンアドレスと同様に各計算機アーキテクチャごとに再計算する必要がある．本実装では，プログラムコード表現変換器が作成するアンロードプリミティブの位置情報を用いてリターンアドレスと同様の変換を行った．

5. 実　　験

現在，4 章で述べた方法に基づいて，SPARC アーキテクチャと Intel x86 アーキテクチャ上でプロトタイプシステムが動作している．本章では，プロトタイプシステムを用いた実験について述べる．実験環境は，モーバイルオブジェクトをロードして計算を行うクライアントとして，Sun SPARCstation（UltraSPARC-II 360 MHz，640 MB RAM，Solaris 2.6）と IBM-PC/AT 互換機（PentiumII 300 MHz，

☆ 現実装では，モーバイルオブジェクト内で setjmp などの割込み処理を禁止している．

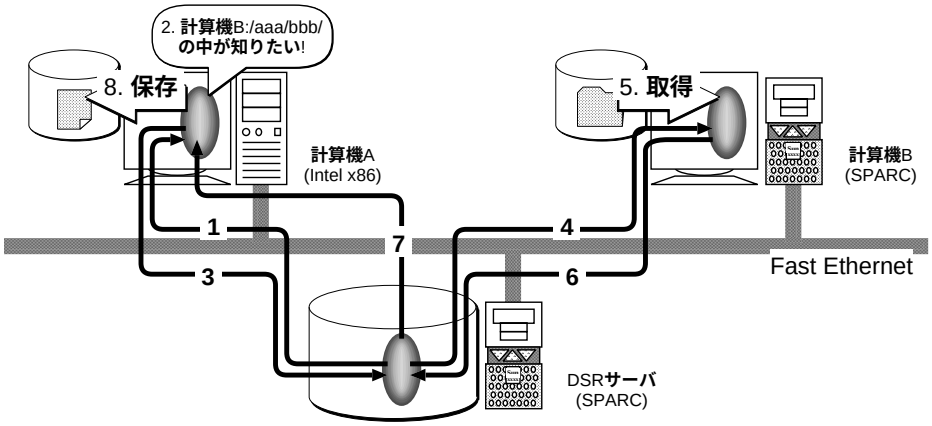

図 7 実験プログラムの動作
Fig. 7 Experimental operation.

128 MB RAM, Solaris 2.6）を用いた．モーバイルオブジェクトを格納する DSR サーバは，Sun SPARCstation（Ultra 360 MHz, 640 MB RAM, Solaris 2.6）を用いた．クライアントと DSR サーバは，Fast Ethernet で接続されている．実験値は，同一の実験を 10 回行った結果を平均している．

本実現法を用いない場合の処理時間の指標を得るために，以下の 4 種類の実験プラットフォームを実行し，応答時間を比較した．

$\mathcal{P}_{UNIX}$ 実験プログラムを，GNU gcc コンパイラで a.out 形式にコンパイルする場合の実験である．あらかじめ，実行する各計算機アーキテクチャのネイティブコードにコンパイルする．

$\mathcal{P}_{MIC}$ 実験プログラムを，MIC 正準表現を介して各計算機の a.out 形式にコンパイルする場合の実験である．実験に先立って，MIC コンパイラバックエンドを使って MIC アセンブラコードにコンパイルした後，各計算機アーキテクチャの a.out ネイティブコードに変換する．

$\mathcal{P}_{Planet}$ 実験プログラムを，PLANET モーバイルオブジェクトとして実装した場合の実験である．DSR サーバ保存時は正準表現を使って表現して，ロード時に本実現法を用いて各計算機アーキテクチャのネイティブコードに変換する．また，実行終了後は，実行結果を含むオブジェクト全体を正準表現に変換して DSR サーバにアンロードする．

$\mathcal{P}_{Java}$ 実験プログラムを，文献 1) で実現されている Java 言語システム上のモーバイルオブジェクト・システム上で動作する Java オブジェクトとして実装した場合の実験である．バイトコード，データと実行状態を含む Java オブジェクトが，DSR サーバと各クライアントで実行している Java 仮想機械上を移動する．使用したモーバイルオブジェクト・システムはオブジェクトの永続処理を実現していないため，DSR サーバは Java 仮想機械プロセス中にオブジェクトを保持する．Java 言語システムは，JDK 1.2.2_05 Standard Edition を使用した．

異機種間のオブジェクト表現変換に要する処理時間とネイティブコード実行方式の影響を測定するために，以下に述べる 2 種類のモーバイルオブジェクトを各実験プラットフォーム上に実装し，実行に要する処理時間を測定した．

5.1 実験 1：ディレクトリ内容取得オブジェクト

実行中に実行状態をともなって異機種間を移動することで計算を行うモーバイルオブジェクトを作成し，処理時間を測定する実験を行った．作成したモーバイルオブジェクトは，指定されたリモート計算機のディレクトリ内のファイル名を昇順にソートして表示する処理を行う．実験アプリケーションの動作を**図 7** に示す．DSR サーバに格納されている正準表現形式のモーバイルオブジェクトが，計算機 A に移動して検索するディレクトリの名前を取得する（図 7 動作 1, 2）．ディレクトリ名をオブジェクト内に保存したモーバイルオブジェクトは，DSR サーバを介して計算機 B に移動して指定されたディレクトリに格納されているファイル名のリストを取得する（図 7 動作 3, 4, 5）．取得したファイル名リストは，クイックソート法を用いて昇順にソートし，その結果を持って再び計算機 A に移動する（図 7 動作 6, 7）．再び計算機 A に移動したモーバイルオブジェクトは，ソートしたファイル名リストをローカルファイルに出力する（図 7 動作 8）．

表 1 実験 1：異機種間オブジェクト移動の処理時間
Table 1 Processing times for moving an object among heterogeneous computers.

		$\mathcal{P}_{UNIX}$	$\mathcal{P}_{MIC}$	$\mathcal{P}_{Planet}$	$\mathcal{P}_{Java}$（JIT あり）	$\mathcal{P}_{Java}$（JIT なし）
計算機 A（Intel x86）	1. 計算機 A へのオブジェクトのロード	—	—	**2768.8**	492.4	410.2
	2. オブジェクトの実行（ディレクトリ名の取得）	0.1	0.0	**0.0**	0.3	0.6
	3. オブジェクトのアンロード	—	—	**184.5**	136.5	104.8
計算機 B（SPARC）	4. 計算機 B へのオブジェクトのロード	—	—	**2493.4**	233.9	230.7
	5. オブジェクトの実行（ファイル名の取得とソート）	25.7	25.7	**60.9**	25.7	212.9
	6. オブジェクトのアンロード	—	—	**173.1**	96.6	246.5
計算機 A（Intel x86）	7. 計算機 A へのオブジェクトのロード	—	—	**2485.1**	56.8	272.4
	8. オブジェクトの実行（ファイル名の保存）	4.5	4.7	**8.0**	4.4	1.9

単位はミリ秒

表 2 実験 1：ロード処理中のオブジェクト表現変換の処理時間
Table 2 Processing times for object translation in the loading process.

	Intel x86	SPARC
オブジェクト表現変換器のロード	340.2	187.3
プログラムコードのネイティブ表現への変換	47.2	36.7
データのネイティブ表現への変換	0.7	0.5
データの正準表現への変換	1.5	3.2
計算状態のネイティブ表現への変換	96.3	137.2
計算状態の正準表現への変換	67.1	57.1

単位はミリ秒

実験結果を**表 1** に示す．表の各列はモーバイルオブジェクトの種類，各行は各計算機における処理内容を示している．$\mathcal{P}_{Java}$（JIT あり）は，ロード時に JDK が提供する JIT コンパイラを用いてネイティブコード表現に変換した後実行を行った場合である．$\mathcal{P}_{Java}$（JIT なし）は，Java バイトコートをインタプリタによって解釈しながら実行を行う．

$\mathcal{P}_{UNIX}$ と $\mathcal{P}_{MIC}$ の結果より，本プログラムでは正準表現を介したネイティブコードも効率的に実行できていることが分かる．MIC を用いて実装した $\mathcal{P}_{Planet}$ では，$\mathcal{P}_{MIC}$ に比べてオブジェクト実行時のオーバヘッドが大きい．これは，オブジェクト転送に用いている Planet システムのリモートメモリマップ・ファイル機構が，オブジェクト実行中にセグメントを単位としてオブジェクトデータをネットワーク転送するためである．さらに，現在のリモートメモリマップ・ファイル機構の実装では，データを持たない空のセグメントに対する参照や更新の場合にも，空のセグメント転送処理が行われる．

$\mathcal{P}_{Planet}$ のロード処理に要する時間が，$\mathcal{P}_{Java}$（JIT なし）の結果を比べて非常に大きい．この原因を調べるために，ロード処理中のオブジェクト表現変換処理に要する処理時間を測定した．**表 2** は，オブジェクト表現変換器のロードにかかる時間とオブジェクトに含まれるプログラムコード，データ，計算状態の各々の表現変換処理に要する時間を示している．DSR サーバからのオブジェクト表現変換器のロード処理に，Intel x86 プロセッサで 340.2 msec.，SPARC プロセッサで 187.3 msec. の時間を必要とした．本実験では，モーバイルオブジェクトをロードするたびにオブジェクト表現変換器をロードしているため，ロードごとに同程度の処理時間を必要とした．一度ロードしたオブジェクト表現変換器をキャッシュするようにシステムを変更することで，2 度目以降のロード処理に要する時間を減らすことができる．

$\mathcal{P}_{Planet}$ でのオブジェクト実行に要する時間は，バイトコード・インタプリタ実行方式である $\mathcal{P}_{Java}$（JIT なし）に比べて，多少の速度向上が見られた．本実験では，オブジェクト実行時間が少なく，扱うデータ量も小さい．ネイティブコード実行方式によるオブジェクト実行時間への利益を確認するために，扱うデータ量とオブジェクト実行処理がより大きいモーバイルオブジェクトを使った実験を行った．

5.2 実験 2：数値計算オブジェクト

ネイティブコード実行方式を用いることによるオブジェクト実行時間への影響を測定するために，5.1 節のモーバイルオブジェクトよりもデータ量と処理内容が大きいモーバイルオブジェクトを使った実験を行った．実験に使用するモーバイルオブジェクトは，各要素に倍精度浮動小数点を含む 50 次の正方行列 A，B に対し，

$$AB^{1000} \quad (= A\underbrace{\cdot B \cdot B \cdots B}_{1000})$$

を計算した結果を行列 A に代入する処理を行う．行列は，データセグメント内に確保してアンロードとともに DSR サーバに永続化される．実験は，DSR サーバと Intel x86 プロセッサの計算機の環境と DSR サーバと SPARC プロセッサの計算機の環境の 2 つの環境で行った．

表 3 実験 2：Intel x86 プロセッサにおける処理時間
Table 3 Ex· 2: Processing times on Intel x86 prossesor.

	$\mathcal{P}_{UNIX}$	$\mathcal{P}_{MIC}$	$\mathcal{P}_{Planet}$	$\mathcal{P}_{Java}$（JIT なし）	$\mathcal{P}_{Java}$（JIT あり）
オブジェクトのロード	—	—	**1999.8**	553.4	565.0
オブジェクトの実行（$A = AB^{1000}$ の計算）	86772.4	113128.9	**115800.0**	667569.8	84866.4
オブジェクトのアンロード	—	—	**128.6**	403.2	208.0

単位はミリ秒

表 4 実験 2：SPARC プロセッサにおける処理時間
Table 4 Ex· 2: Processing times on SPARC prossesor.

	$\mathcal{P}_{UNIX}$	$\mathcal{P}_{MIC}$	$\mathcal{P}_{Planet}$	$\mathcal{P}_{Java}$（JIT なし）	$\mathcal{P}_{Java}$（JIT あり）
オブジェクトのロード	—	—	**1962.8**	331.4	333.7
オブジェクトの実行（$A = AB^{1000}$ の計算）	14611.9	15258.2	**18743.1**	189678.8	11768.7
オブジェクトのアンロード	—	—	**86.7**	244.3	75.2

単位はミリ秒

実験結果を**表 3** と**表 4** に示す．表中の各列はモーバイルオブジェクトを実装した実験プログラムの種類，各行は処理内容を示している．

$\mathcal{P}_{UNIX}$ と $\mathcal{P}_{MIC}$ の結果と比べると，$\mathcal{P}_{MIC}$ のオブジェクト実行時間が増加している．これは，3.2 節で述べた正準表現に対する条件である，レジスタ割当て決定性（条件 4）を満たすために，CPU のレジスタを十分に利用していないためである．特に，MIC 正準表現でのレジスタよりもレジスタの数が少ない Intel x86 プロセッサでは，レジスタに対する命令をメモリで代用しているため，その影響が大きい．

$\mathcal{P}_{Planet}$ の結果は，5.1 節で述べたリモートメモリマップ・ファイル機構によるオブジェクトデータ転送による処理時間の増加があるが，計算処理自体に要する時間が大きいため，$\mathcal{P}_{MIC}$ とほぼ同じ処理時間で計算を実行できた．$\mathcal{P}_{Java}$（JIT なし）と比較すると，ネイティブコード実行方式によるオブジェクト実行速度の向上が見られる．5.1 節の実験の結果と比較すると，オブジェクト実行時間が比較的大きいアプリケーションで本実現法による利点を享受することができることが分かった．

同様のネイティブコード実行方式である $\mathcal{P}_{Java}$（JIT あり）の結果に比べて $\mathcal{P}_{Planet}$ でより多くの計算時間を要した理由は，MIC のコンパイル時およびネイティブ表現変換時の最適化が，Java の最新 JIT 技術に比べて不十分であるためである．現在の実装システムにおいても，ある程度の最適化は可能であるが，レジスタ割当てやコード順序の変更などのよりいっそうの最適化を行うためには，3.2 節で述べた正準表現に対する条件を緩和する必要がある．

6. 関 連 研 究

ネイティブ実行のオブジェクトに対する異機種環境でのモビリティを実現するものとして，Heterogeneous Emerald [13] があげられる．バスストップと呼ばれる移動可能ポイントをコンパイル時に自動生成し，各バスストップでの状態復元のためのコードを自動生成することでモーバイルオブジェクトを実現している．異機種環境への対応は，すべての計算機アーキテクチャ上でバスストップの位置を統一し，各計算機アーキテクチャごとにオブジェクト生成を行うことで実現している．Heterogeneous Emerald は，モーバイルオブジェクト計算用言語 Emerald の仕様に深く依存した設計となっている．本実現法は，特定のプログラミング言語に依存しないように設計を行っている．

異機種間でのプログラムコードのモビリティを実現する技術として，Slim Binaries [4] がある．Slim Binaries は，コンパイル中の構文木の状態を抽出し，圧縮した状態で移動する．移動先では，構文木を基にネイティブコードを生成する．Slim Binaries では，抽象度の高い構文木を正準表現として用いているが，本実現法では，ロード時のネイティブコード生成の高速化に注目し，比較的抽象度の低い仮想機械命令セットを用いている．オブジェクトの計算状態の正準表現形式は，プログラムコードの正準表現形式が深く関係する．構文木を正準表現に用いた場合，プログラムカウンタなどのネイティブ表現の計算状態を生成されるプログラムコードに対応させることは困難である．

異機種分散共有メモリ Mermaid [19] は，異機種環境上で仮想記憶空間上のデータを共有する機構を実現し

ている．Mermaid では，異機種間のページ転送の際に，データ変換器を用いたデータ表現変換を行うことにより，異機種間データ共有を実現している．データ表現変換の際，正準表現を介さず，システムを構成する各機種のネイティブ表現間の変換の全組合せを実現するデータ表現変換器を用いる．変換器は，各計算機アーキテクチャごとにアプリケーションのコンパイル時にリンクする．システム内に新しい計算機アーキテクチャを追加する場合は，その計算機アーキテクチャに対する表現変換器を作成し，すべての計算機アーキテクチャ上でアプリケーションをコンパイルし直す必要がある．本実現法では，オブジェクト表現変換器をモーバイルオブジェクトとして実現しているため，システム内に新しい計算機アーキテクチャを追加する場合や新たな正準表現形式を用いる場合もアプリケーションの再コンパイルは必要ない．アプリケーションロード時は，用いられている正準表現に対して適切なオブジェクト変換器をロードする．

Arachne[3] は，ソースコード変換技術を用いてスレッドの異機種間モビリティを実現している．アプリケーションプログラムに，ソースコード変換を用いてスレッド移動に必要な処理をプログラム内に挿入する．スレッド移動時に実行を再開する位置を特定するために，各関数内を複数のブロックに分けて，各ブロックに状態を示す変数とラベルを挿入する．移動は，実行状態を示す数を用いて各部ロックのラベルへジャンプ文（goto 文）を挿入することで実現する．スタック内のローカル変数は，すべての変数をヒープ領域に確保し，ヒープ領域上のデータをアクセスするようにデータアクセス部分のコード変更を行う．ソースコード変換による実行状態の抽出は，対象とするプログラミング言語に対する深い知識と各プログラミング言語に特化した実現を行う必要がある．本実現法では，特定のプログラミング言語に依存しない実行状態の抽出と移送を実現している．

7. おわりに

ネイティブコードで実行されているオブジェクトに対する異機種環境でのモビリティの実現法について述べた．本実現法では，プログラムコード，データ，計算状態のそれぞれに正準表現を設計し，モーバイルオブジェクト実行時のネイティブ表現と移動時の正準表現を動的かつ反復的に変換する機構を提供する．用いる正準表現に関して，動的双方向変換の実現に必要な正準表現の仕様について検討した．実装では，プログラムコードの正準表現を設計し，データの正準表現として XDR を用いた．計算状態の正準表現では，プログラムコードの正準表現で用いた仮想機械の仕様に基づき，レジスタとスタックを機種非依存な形式に変換する．

今後の課題としては，第 1 に，SPARC CPU と Intel x86 CPU 以外の計算機アーキテクチャ上での実装と，それらの実装を用いた本実現法の有効性の検証がある．現在，PowerPC および Alpha CPU 上での実装を進めている．第 2 に，オペレティングシステム・インタフェースの異機種性に対応するための PLANET API の設計がある．現在，PLANET API の設計を進めるとともに，Solaris オペレーティングシステム以外のオペレーティングシステム・プラットフォームとして，Linux と Windows NT 上で実装を進めている．

謝辞 本稿の執筆にあたり，実験環境の整備に協力していただいた筑波大学阿部洋丈氏に感謝いたします．本研究の一部は，文部省科学研究費補助金奨励研究（A）11780189 の補助を受けて行われた．

参考文献

1) 阿部洋丈，一杉裕志，加藤和彦：ソースコード変換技術を用いた Java 言語におけるスレッドのモビリティの実現法，情報処理学会論文誌プログラミング，Vol.41, No.SIG 2 (PRO 6), pp.29–40 (2000).
2) Cardelli, L.: A language with distributed scope, *Computing Systems*, Vol.8, No.1, pp.27–59 (1995).
3) Dimitrov, B. and Rego, V.: Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms, *IEEE Trans. Parallel and Distributed Systems*, Vol.9, No.5, pp.459–469 (1998).
4) Franz, M. and Kistler, T.: SLIM BINARIES, *Comm. ACM*, pp.87–94 (1997).
5) Gingell, R.A., Lee, M., Dang, X.T. and Weeks, M.S.: Shared Libraries in SunOS, White Paper, Sun Microsystems, Inc (1993).
6) Glass, G.: ObjectSpace Voyager – The Agent ORB for Java, *Worldwide Computing and Its Applications – WWCA '98*, LNCS, Vol.1368, pp.38–55, Springer-Verlag (1998).
7) Jul, E., Levy, H., Hutchinson, N. and Black, A.: Fine-grained mobility in the Emerald system, *ACM Trans. Comput. Syst.*, Vol.6, No.1, pp.109–133 (1988).
8) Kato, K., Matsubara, K., Someya, Y., Itabashi, K. and Moriyama, Y.: Design of an Open Mobile Object System for Open Networks, *Proc. International Workshop on Paral-*

lel and Distributed Computing for Symbolic and Irregular Applications (*PDCSIA '99*) (1999).
9) Kato, K., Narita, A., Inohara, S. and Masuda, T.: Distributed shared repository: A unified approach to distribution and persistency, *Proc. 13th IEEE Int. Conf. on Distributed Computing Systems*, pp.20–29 (1993).
10) Kato, K., Someya, Y., Matsubara, K., Toumura, K. and Abe, H.: An Approach to Mobile Software Robots for the WWW, Special Issue on Web Technologies, *IEEE Trans. Knowledge and Data Engineering*, Vol.11, No.4, pp.526–548 (1999).
11) Lange, D.B. and Oshima, M. (Eds.): *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley (1998).
12) 松原克弥, 加藤和彦：分散永続性を提供するモーバイルオブジェクト・システムの実現, 情報処理学会論文誌, Vol.39, No.8, pp.2494–2508 (1998).
13) Steensgaard, B. and Jul, E.: Object and Native Code Thread Mobility Among Heterogeneous Computers, *SIGOPS*, pp.68–78 (1995).
14) Sun Microsystems: XDR: External Data Representation Standard, Request for Comments (RFC): 1014 (1987).
15) Thorn, T.: Programming languages for mobile code, *ACM Computing Surveys*, Vol.29, No.3, pp.213–329 (1997).
16) Vitek, J. and Tschudin, C. (Eds.): *Mobile Object Systems*, LNCS, Vol.1222, Springer-Verlag (1997).
17) Wang, D., Paciorek, N., Walsh, T., DiCelie, J., Young, M. and Peet, B.: Concordia: An Infrastructure for Collaborating Mobile Agents, *Proc. 1st International Workshop on Mobile Agents* (*MA '97*) (1997).
18) White, J.E.: Mobile Agents, *Software Agents*, Bradshaw, J. (Ed.), MIT Press (1996).
19) Zhou, S., Stumm, M., Li, K. and Wortman, D.: Heterogeneous Distributed Shared Memory, *IEEE Trans. Parallel and Distributed Systems*, Vol.3, No.5, pp.540–554 (1992).

(平成 11 年 12 月 15 日受付)
(平成 12 年 4 月 6 日採録)

松原 克弥

1971 年生．1994 年筑波大学第三学群情報学類卒業．1996 年筑波大学大学院修士課程理工学研究科修了．1998 年より筑波大学電子・情報工学系助手，現在に至る．オープンネットワークのための分散システムソフトウェアに興味を持つ．USENIX 会員．

板橋 一正

1974 年生．1998 年筑波大学第三学群情報学類卒業．2000 年筑波大学大学院修士課程理工学研究科修了．現在,（株）アステック・プロダクツに勤務．システムソフトウェアの設計と実装に興味を持つ．

森山 豊

1976 年生．1998 年筑波大学第三学群情報学類卒業．2000 年筑波大学大学院修士課程理工学研究科修了．現在，日本アイ・ビー・エム（株）に勤務．システムソフトウェアの設計と実装に興味を持つ．日本ソフトウェア科学会会員．

染谷 祐一

1975 年生．1997 年筑波大学第三学群情報学類卒業．同年筑波大学大学院博士課程工学研究科に入学，現在に至る．オペレーティングシステム，プログラミング言語システム，分散システムに興味を持つ．日本ソフトウェア科学会会員．

加藤 和彦（正会員）

1962 年生．1985 年筑波大学第三学群情報学類卒業，1987 年工学修士（筑波大学大学院工学研究科），1992 年博士（理学）（東京大学大学院理学系研究科）．1989 年東京大学理学部情報科学科助手，1993 年筑波大学電子・情報工学系講師，1996 年同助教授，現在に至る．オペレーティングシステム，プログラミング言語システム，データベースシステム，分散システム，モバイルオブジェクト計算に興味を持つ．電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE 各会員．

関口　龍郎

1970 年生．1999 年より日本学術振興会研究員．1999 年東京大学大学院より理学博士取得．主にモーバイル言語システムの研究に従事．

米澤　明憲（正会員）

1947 年生．1977 年 Ph.D. in Computer Science（MIT）．1989 年より東京大学理学部情報科学科教授．超並列・分散ソフトウェアアーキテクチャ等に興味を持つ．共著書「モデルと表現」等（岩波書店），編著書「ABCL」（MIT Press）等がある．1992〜1996 年ドイツ国立情報処理研究所（GMD）科学顧問，ACM Transaction on Programming Languages and Systems 副編集長，IEEE Parallel & Distributed Technology および Computer 編集委員等を歴任，元日本ソフトウェア科学会理事長，ACM Fellow．