異機種間モーバイル計算のためのコード表現とその実装

関 口 龍 郎† 米 澤 明 憲†

モーバイル計算と呼ばれる，実行時にコードが移動する計算形態がインターネット上で普及しつつある．この論文は異機種間でのモーバイル計算に適したコード体系 MIC の提案を行い，その概要を示す．MIC は 3-アドレス形式の RISC 風命令セットにより構成され，異機種計算機上で，その機種専用の C コンパイラによって生成されたコードと同等の速度で動作させることができる．これを利用すれば，高い実行性能を要求するアプリケーションを異機種間モーバイル環境で実行させることができる．高い実行性能を持つコードは必要最小限度の最適化によって生成が可能であり，コードが実行される段階で初めて転送されるモーバイル計算に適している．MIC コードはアセンブラに近い低レベルな言語表現であり，プログラミング言語に対する依存性が低く，MIC コードを利用したモーバイルシステムでは様々なプログラミング言語をその言語の特徴を保持したまま利用することができる．我々は MIC コードを出力する C，C++ 言語コンパイラと，MIC コードから SPARC と IA32 アーキテクチャのためのコードを生成するコード生成器を実装した．

A Code Representation for Heterogeneous Mobile Computation and Its Implementation

Tatsurou Sekiguchi† and Akinori Yonezawa†

Mobile computation is getting popular on the Internet, which is a form of computation where running programs are moved among distributed computers in a network. This paper proposes a code representation called MIC that is designed for mobile computation on heterogeneous computers. MIC consists of a RISC instruction set of three address format. A MIC code can run on a computer of various architecture as fast as code produced by a native compiler for the architecture. This feature of MIC allows us to run an application on a heterogeneous mobile environment that requires high execution performance. MIC is suitable for mobile computation since a native code with high efficiency can be generated by the least optimizations from a MIC code. In a mobile environment, a code is usually transmitted and available just before the code is executed. MIC is a low-level code representation like assembly languages, which encourages that a program in a various high-level programming language can be compiled into a MIC code without losing important features of the programming language. A translator from a MIC code into SPARC and IA32 native code is implemented. In addition, a C and C++ compiler is implemented that produces MIC code.

1. はじめに

インターネットの新しい利用法が次々に開発されていく中でモーバイル計算と呼ばれるソフトウェアの利用形態が普及してきている．モーバイル計算とはソフトウェアのコード（プログラム）が，それを実行する直前に利用者が使用するコンピュータ上にネットワークを通じて転送されることを特徴とする計算の形態である[41]．ただし「モーバイル計算」という言葉は PDA や小型の携帯コンピュータを扱う技術の総称としても使われることもある．この混乱を回避するために Cardelli は文献 5) においてソフトウェアの移動を扱う分野を *mobile computation* と呼び，移動するハードウェアを扱う分野を *mobile computing* と呼ぶ区別を提案した．ただし mobile computing に mobile computation を利用する研究[20] もあり，両者は密接な関係にある．この論文は mobile computation にかかわる研究である．

モーバイル計算は，多種多様な分散アプリケーションを構築する基盤である．新しい分散アプリケーションの需要が生じてからその専用のシステムを構築するのではなく，プログラムが移動するモーバイル計算では様々な形態の分散計算を柔軟に実行できることにその特徴がある．このようなモーバイル計算の枠組みを利用したアプリケーションとして情報検索[26]，

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Faculty of Science, the University of Tokyo

WWW ページ収集ロボット[23)]，ワークフロー[4)]，移動サーバ[32)] などを含む多くの提案[7),29)] がある．おそらく最も普及しているモーバイル計算は WWW ページにコードを埋め込み，読者とのインタラクティブな操作を可能にする WWW ブラウザの機能であろう．この研究の目的は，以上のようなアプリケーションを動作させるモーバイル計算の基盤システムに適したコード表現を与えることにある．

これらのアプリケーションは主にインターネット上での実行を想定している．インターネットはプロセッサ，クロック，メモリ，ディスク，オペレーティングシステムなどの点で実に様々な種類のホストから構成されている．そのためモーバイル計算の研究においてはその当初からコードが実行されるホストのオペレーティングシステムやプロセッサの不均一さが問題となっていた．この問題を回避するために多くのシステム[16),18),40)] では機種共通のバイトコードをインタプリタ実行していた．

しかし，インタプリタ実行は確かに不均一な実行環境に対応することができたが，その代わりにプログラムの実行速度が遅いという欠点があった．そこでこの問題を解消するために機種共通のバイトコードを機械本来のコード表現に変換して高速に実行する手法[21),34)] が考案された．

同一のコードを異機種のコンピュータ上で動作させることは古くから行われてきている[39)]．1980 年代に UNIX を使用する人々の間ではソフトウェアをソースコードの状態で配付するのが通常であった．しかし，モーバイル計算の場合には，コードが実行されるときになって初めてそのコードが実行環境に送られてくるという著しい特徴がある．したがって，コードをできるだけ速く実行可能な形式にすることが望まれる．

高レベルな言語表現から機械本来のコードを実行時に生成する研究は部分計算[22)] の機能を持つ言語の実装[17),27)] で行われてきている．それらの研究の多くは特定のアーキテクチャの上で実装が行われており，多くのプロセッサとオペレーティングシステムに対応させるという観点はほとんど見られない．

本研究の意義は，モーバイル計算に適したコード表現を設計するのに必要な技術を明らかにし，その処理系を実装，提供したことにある．我々は異機種上で，最小限の最適化によって，その機種本来のコンパイラと同等の実行性能を持つ，言語独立なコード体系 MIC の設計を行った．加えて，この機械独立コード MIC を出力する C 言語，C++ 言語のコンパイラを作成した．

また機械独立コードという仮想的な層を設定することは安全性を確立する方法を支援している．モーバイルコードの安全性を確立するためにはコード検証を用いる手法[42)] や，SFI[38)] のようにコードに動的チェックを挿入する手法などがあるが，機械独立コードという層が設定されていればいずれの手法を実現する場合でも機械独立コードに対して安全性の保証を行えばよい．これに対して機械本来語をモーバイルコードに用いる場合では機械本来語の種類だけコード検証やコード挿入を行う処理を実装する必要がある．

この論文の構成は次のようになっている．2 章では機械独立コードの設計方針について議論する．3 章では機械独立コードを実現するための具体的な問題点とその本研究での解決法について述べる．4 章では MIC コードの例を解説し，また MIC システムの実装の概要についても簡単に述べる．5 章では我々の機械独立コードの評価を行う．6 章では関連研究との比較を行う．7 章は簡単なまとめを行う．

2. 機械独立コードの設計方針

この章では機械独立コードの設計方針について議論する．

2.1 理想的な機械独立コードの条件

我々は，理想的な機械独立コードの十分条件を次のようなものと考える．

実行速度の速さ 機種専用の C コンパイラの出すコードと同等の速度で動作する．

翻訳速度の速さ 機械独立コードから機械本来のコードに変換する段階で高価な解析や最適化を行う必要がなく，高速に機械本来のコードを生成できる．この観点はコードを実行しなければならない段階になって初めてコードが転送されてくるモーバイル計算ならではの特徴である．この観点で良い結果を出すためにはそのコード体系上において高価な最適化が有効でなければならない．

言語非依存性 様々なプログラミング言語から機械独立コードに変換したときに元のプログラミング言語の持つ性質を可能な限り保存する．インターネットは様々な特徴を持つ計算機によって構成されており，それらの計算機の性質を最大限に引き出すことのできる言語をモーバイル計算でも使えるのが望ましい．したがって，特定のプログラミング言語でのみモーバイル計算ができるという状況は望ましくない．そのためモーバイル計算で使われるコード表現は多くのプログラミング言語の性質を反映できるものでなければならない．

アーキテクチャ非依存性 様々なプロセッサ，オペレーティングシステム上で動作させることが可能である．

高い安全性 作成者の不明なコードを実行することが通常であるモーバイル計算では危険かもしれないコードを安全に実行させる技術が必要である．

これらの性質のうち，本論文では安全性については扱わないが，本研究の方式と組み合わせて使用することのできる安全性を確立するための仕組み[24),38)]が存在している．

2.2 機械独立コードの設計方針

我々の機械独立コード MIC はアセンブラ言語に近いコード体系として設計されている．その理由は次の 2 点である．

- 実行速度の速さと翻訳速度の速さを両立させるためには高価な最適化を行わずに生成されたコードが高速に動作しなければならない．つまり機械独立コードはすでに最適化を行った後のようなコード体系である．
- アセンブラ言語レベルでは言語非依存性を達成するのが容易である．

翻訳速度の速さを実現するために，機械本来コード生成時には関数のインライニング，大域的データフロー解析などの関数境界または基本ブロックをまたがる最適化，解析を避けるようにしている．

3. 機械独立コードの実現の技術的課題とその解決

アセンブラに近い機械独立コードを設計するにあたって次のような問題を解決する必要があった．

抽象機械 vs. ポインタ演算 作成者の不明なコードを実行するモーバイル計算においてはコードの実行がシステムを破壊しないことを保証する必要がある．静的な検証[28),35)]によってコードの安全性を保証するためには抽象機械を定義してコードの動的な意味と静的な意味を与える必要がある．この方式ではコードの安全性を高め，動的なチェックを省くことによる実行効率の向上が期待されるが，逆に一部の言語の意味を実装するのが困難になり，またポインタ演算などの効率的な操作を記述できなくなる．

RISC vs. CISC 機械独立コードはどの程度高級な命令を備えるべきだろうか．オブジェクト指向命令やストリングス命令などの命令を備えるべきだろうか．またセグメントやプロテクションなどの機構を備えるべきだろうか．

スタックマシン vs. レジスタマシン 命令のオペランドの形式は機械独立コードから機械本来コードを生成するときに重要な効果を持つ．

レジスタ割当ての方式 レジスタ割当ては高価な最適化技術であるが，高い実行性能を引き出すためには不可欠なものである．どのようなレジスタ割当てを行えばよいのであろうか．

アドレッシングモード 機械独立コードのアドレッシングモードの設計は機械独立コードから機械本来コードを生成するときに重要な効果を持つ．

関数呼び出し規約 関数呼び出し規約はプロセッサ，OS，言語に従って変化する．機械独立コードはどのようにそれらの差異を吸収すべきだろうか．

3.1 抽象機械 vs. ポインタ演算

抽象機械を定義し，静的な検証によってコードの安全性を確立する方式には大きく分けて，機械独立なバイトコードを用いるもの[35)]と既存のアーキテクチャのサブセットとして抽象機械を定義するもの[28)]がある．これらの研究に共通するのは言語システムの統合性を保つために無制限のポインタ演算や型強制，メモリの割当て，開放を禁止していることである．

しかし，このような手法には次のような問題がある．

- モーバイルコードによって表現できるアプリケーションの範囲が制限される．新しい抽象機械を提供するために使えない[1)]．
- C 言語のような意味論を効率的に実装できない．

一方，無制限のポインタ演算が許されている場合には静的に安全性を保証するのは困難であるが，SFI[38)]や Planet[24)]など少ない実行時オーバヘッドで安全性を確立する技術が存在している．また，ポインタ演算や型強制は高速な実行を可能にし，様々なプログラミング言語を効率良く実装するためにも不可欠である．

安全性を保証しない状況でも信頼された閉じた環境で使用されるソフトウェアなどにも機械独立コードを応用することができる（たとえば EmacsLisp のバイトコードなど）．

3.2 RISC vs. CISC

アセンブラレベルでの機械独立コードで任意のポインタ演算を許している関連研究[1),11)]では RISC を採用している．文献 1) によれば IA32 アーキテクチャの実装の多くは RISC に基づいたスーパスカラであり，CISC アーキテクチャを RISC として利用しても問題はないと主張している．本研究もこの考えに従う．

オブジェクト指向言語のための専用の命令（仮想関数呼び出し命令，継承関係判定命令など）を導入することの是非について考える．仮想関数呼び出しを 1 つ

の命令として提供することの利点はより基本的な命令から構成するよりも仮想関数呼び出し命令を検出するのが容易になる点である．その結果，仮想関数呼び出し命令に対して専用の解析，最適化を行う余地が生じる．

オブジェクト指向言語のための専用の命令を備えた中間コード表現で C++，Modula-3，Java などの実装を行っている枠組み[10] によれば，実行効率の向上にとって重要なのはプロファイリングやその結果に基づくモジュール間インライン展開である．しかし，コードが 1 つの実行環境において繰り返し実行されることの少ないモーバイル計算においてはプロファイリングを行うのは非現実的であり，またモジュール間インライン展開は局所的な最適化のみで済ませるという原則に反する．したがって，我々はこのような命令を採用しない．

3.3 スタックマシン vs. レジスタマシン

この節では機械独立コードを設計するにあたり，スタックマシンとレジスタマシンの利害得失について議論する．

モーバイル計算ではコードを実行直前に転送することが多い．そのためコードの転送時間も実行するまでにかかる時間の中で重要な意味を持つ．スタックマシンはコードサイズが小さくなりがちであり，コードの転送時間を短くする効果がある．また値を使用するとスタックから取り除かれるため値の生存期間を確定しやすい効果があり，プログラム解析にも有利である．

しかし，既存のプロセッサのアーキテクチャはほとんどがレジスタマシンであるため，機種本来のコードを生成するためにはスタックマシンからレジスタマシンに変換する必要がある．この変換を行うにはスタックを使って受け渡される値のフローを追跡してそれをレジスタに割り当てることになる．このような変換を高速に行う方式[15] や，またスタックマシンのままでスタックトップにレジスタを割り当てて高速化を図る研究[14] も存在するが，実行性能の向上はそれほどでもなく☆，実行性能を向上させるためには，多くの Java JIT コンパイラが行っているようにデータフロー解析を行わなければならない．

一般にスタックマシンコードはレジスタマシンコードに比べて小さくなる傾向があり，スタックマシンからレジスタマシンに高速に変換することができるのであるから，スタックマシンコードは転送時にのみ使うようにすればよいのではないだろうか．しかし，転送用のコード表現として，コードを圧縮する手法についての研究[13),31] がなされており，これらの手法を使うとバイナリコードを元のサイズの 1 割から 2 割程度にまで圧縮することができる．また Java に限っていえば抽象構文木を転送形態として利用する研究[25] がある．この方式では，コードサイズは未圧縮の状態で Java バイトコードの 4 割程度になる．

3.4 レジスタ割当ての方式

この節ではレジスタ割当ての方式について議論する．

3.4.1 基本的なレジスタ割当ての方式

後の議論で必要になるため，最初にすでに知られている基本的なレジスタ割当ての方式について簡単に説明する．

Chaitin の方式[6] この手法では機械語 1 命令ごとに仮想レジスタを割り当てておく．値が到達する可能性のある仮想レジスタの集合を生存区間（live range）と呼ぶ．生存区間を点として，同時に存在することのある生存区間の間に線を引いたグラフをプログラムから求める．このグラフを干渉グラフと呼ぶ．生存区間ごとに優先度と呼ばれる経験的な指標を与える．優先度の最も高い生存区間から順番に物理レジスタを割り当てていく．このときに線で結ばれた生存区間と同じ物理レジスタを割り当てることのないようにする．もしそのような物理レジスタが存在しないならば，レジスタ割当ては失敗であり，レジスタ溢れが発生する．レジスタ溢れでは，失敗した生存区間に対応するメモリ領域を確保し，その生存区間をいくつかの局所的な生存区間に分割する．その生存区間に含まれる仮想レジスタを使用する命令の直前にメモリから読み込む命令を付加し，その仮想レジスタに書き込みを行う命令の直後にメモリへ書き込む命令を挿入する．以上の動作をレジスタ割当てが成功するまで行う．

Chow の方式[8] この方式は Chaitin の方式と同様であるが，レジスタ割当てに失敗したときにいくつかの命令を挿入して繰り返すということを行わない．その代わりに数個の物理レジスタを余らせた状態でレジスタ割当てを行い，失敗した生存区間にアクセスする命令はこの余っているレジスタを利用する．繰返しを行わないことでレジスタ割当ての時間を大幅に短縮することができる．

3.4.2 レジスタ割当て方式の比較

我々はレジスタ割当ての方式を実際に実装して比較を行った．ただしレジスタ割当ての効果は物理レジス

☆ 文献 14) ではランタイムが使用するメモリ領域が数倍になるのに対して実行速度の向上は 1 割程度にすぎない．

表 1　IA32 上の Omniware の gcc に対する実行時間比
Table 1　Relative elapsed time of Omniware on IA32 compared with gcc.

プログラム	li	compress	alvinn	eqntott
実行時間比	1.09	1.01	1.09	1.05

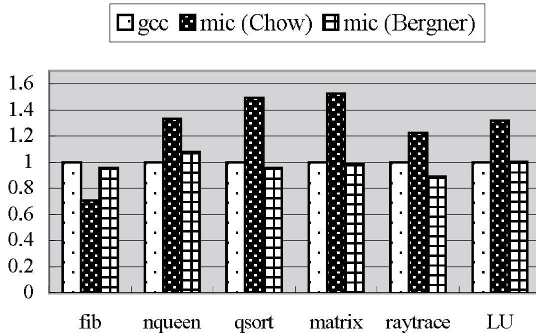

図 1　IA32 上でのレジスタ割当ての効果
Fig. 1　The effect of register allocation on IA32.

タの少ないプロセッサ上で顕著となるためこの項では実行環境を IA32 アーキテクチャに限定している．

表 1 に Omniware の IA32 上での gcc に対する SPEC92 のプログラムの実行時間の比率を示す．これは C 言語のプログラムを中間アセンブラコードに変換し Omniware で実行された実行時間と，gcc によって直接生成したコードの実行時間の比率を表にしたものである[1]．また gcc のバージョンは 2.5.8 である．Omniware は Chow の方式をさらに簡略化したものを採用している．これはあらかじめ 16 の生存区間だけが同時に存在できるようにコードを生成しておいて，物理レジスタを順番にその生存区間に当てはめていく方式である．この方式では生存区間解析もしないで済ますことができる．

図 1 に我々の実験結果を示す．これはレジスタ割当ての方式を変えたときの実行時間の違いを gcc の生成するコードの実行時間との比率として表したものである．図中で mic（Chow）が Chow の方式によりレジスタ割当てを行ったコードを意味する．mic（Bergner）は Chaitin の方式の改良である Bergner の方式[3] によってレジスタ割当てを行ったコードを意味する．また我々が使用した gcc のバージョンは 2.95.2 であり，コンパイルオプションは `-O2 -malign-double` である．実験で使われたプログラムについては 5 章で解説されている．

我々は少なくとも文献 1) に記述されている方式だけでは Omniware と同様の結果を出すことができなかった．Chow の方式ではレジスタに割り当てられなかった生存区間を扱うために IA32 上では 2 個のレジスタをレジスタ割当てから除外しておく必要がある．一般レジスタが 8 つしかない IA32 アーキテクチャではこの除外の効果は甚大であり，最大で 50%を超えるオーバヘッドの原因となる．我々の結論はもし gcc と同等の性能を得たいのであれば Chaitin あるいはそれを改良した Bergner の方式を使うべきである，というものである．

表 1 の Omniware の実行結果は必ずしもレジスタ割当ての性能の計測だけを目的として測定されたものではない．しかし文献 1) によれば，彼らは最適化として浮動小数点パイプラインスケジューリングと自明な覗き穴最適化しか行っていないと主張しているため，特に整数を扱うプログラムでは大きな違いはないと考えられる．我々のコード生成の方式ではレジスタ割当てと自明な覗き穴最適化だけを行っている（SPARC の場合には遅延スロットの充填も行われる）．

VCODE [11),12)] でも高速な機械本来語生成のために Chow の方式を採用している．ただし，彼らは機械独立コードを生成した段階であらかじめ優先順位を計算しておき，機械本来語生成時には順番に物理レジスタを割り当てる．Omniware の文献には優先順位の計算に関する記述は見られない．VCODE に関して高速にレジスタ割当てを行う研究[30)] が存在する．この研究では高速性を重視するため，循環を含む制御構造では循環を 1 点に置き換えて生存区間の解析を行っている．

3.4.3　生存区間解析の高速化

いずれのレジスタ割当ての方式にせよプログラムから干渉グラフを生成する必要がある．そのためには値の生存区間解析を行わなければならない．我々は $O(n^3)$ の計算量のかかる生存区間解析を避けるために機械独立コードにアノテーションを挿入している．ただし n はレジスタの出現数である．コードに挿入されるアノテーションは次の 3 つの情報からなる．

- 生存区間に固有なレジスタ番号をレジスタに与える．
- 基本ブロックの先頭で生存している生存区間の集合を与える．
- 最後に使われるレジスタに印を付ける．

この注釈を利用することで生存区間解析を $O(n^2)$ の時間空間計算量で行えるようになる．

3.5　アドレッシングモード

アドレッシングモードに関する損得を，(1) シンボリックアドレス（リロケータブル 32 bit アドレス）を導入すべきかという問題と，(2) スケール付きインデックスアクセスを導入すべきかどうかという問題で議論する．シンボリックアドレス（以後，SA）とスケール

付きインデックスアクセス（以後，SIA）は IA32 アーキテクチャには備わっているが多くの RISC プロセッサには備わっていない．SIA は配列のアクセスなどによく使われるアドレスの形式である．CISC プロセッサとして IA32 アーキテクチャを，RISC プロセッサとして SPARC アーキテクチャを取り上げる．

機械独立コードが SA を採用すると SPARC 上で不利益が生じる．SPARC では 13 bit の即値しか扱えないため新しいレジスタを 1 つ用意し，そのレジスタに 32 bit のアドレスを設定し，そのレジスタを使って当該のアドレスにアクセスすることになる．メモリ操作命令がループの内側にあり，かつそのアドレスがループ不変であるときには，アドレスを設定する命令をループの外側に移動させる最適化を行わなければ大きなオーバヘッドの原因となる．

一方，機械独立コードが SA を採用しない場合には IA32 上で不利益が生じる．SA を採用していない機械独立コードで SA と同等なことを行うためには新しいレジスタを 1 つ割り当てて，そのレジスタに 32 bit の即値を設定し，そのレジスタを使って当該のアドレスにアクセスするコードが生成されていることになる．この場合に深刻な問題とはレジスタを 1 つ取られることである．IA32 では一般レジスタは 8 個しかなく，フレームポインタ，スタックポインタを除くと 6 個しかない．レジスタの数を減らすためには定数即値を持つレジスタがメモリアクセスに使われているときにはそれを即値で置き換えるような最適化を行わなければならない．

まとめると，SA に関する損得とは

- SA を採用する場合，SPARC 上，ループの内側に SA を含むメモリアクセスがあるときに，可能ならば SA の設定をループの外側に移動させる，
- SA を採用しない場合，IA32 上で SA を保持するレジスタによるメモリアクセスがあるときに，定数をメモリアクセスに直接埋め込む，

という対立である．現在の MIC は SA を採用せずに IA32 のための最適化を行っている．

次に SIA に関する損得であるが，SIA を採用する場合には SPARC 上で不利益が生じる．SPARC には SIA がないためビットシフトなどを使って同等の効果を発生させなければならない．特に SIA がループの内側にある場合には数倍の効率低下が生じる．したがって，スケール付きインデックスの値がループの中で不変であればそれをループの外側に移動させる最適化を行わなければならない．一方，機械独立コードが SIA を採用しない場合には SA と同様の理由によってレジスタを多く使用することになる．

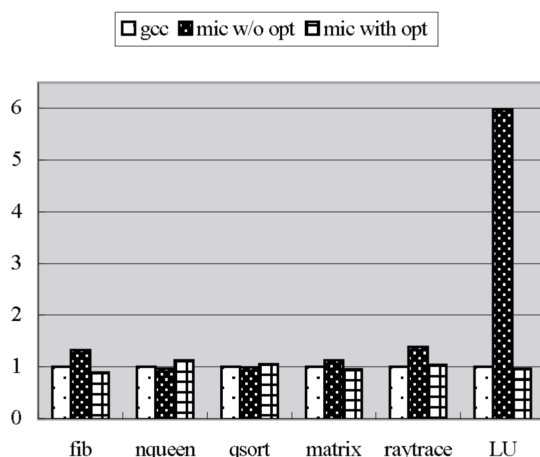

図 2 SPARC 上での SIA に関する最適化の効果
Fig. 2 The effect of optimization concerning SIA on SPARC.

図 2 に SPARC 上で SIA に関する最適化を行った場合と行わなかった場合の実行時間の比率を示す．ただし実行時間は同じプログラムを gcc で直接コンパイルしたコードの実行時間を 1 として正規化してある．gcc のバージョンは 2.95.2 であり，コンパイルオプションは `-O2` である．実験で使われたプログラムについては 5 章で解説されている．この中で SIA を使うプログラムは matrix，raytrace と LU である．matrix の場合は最適化を行わないことの不利益があまり生じてはいないが，LU の場合は最適化をしないと約 6 倍遅くなっている．この違いは SIA を使う回数に起因している．matrix の場合は 1 つのアドレスに対して 1 度しかロードまたストアを行っていない．SIA を持たないプロセッサ上で SIA を行うためにはオフセットの計算を必ず 1 度はやらなければならないためオーバヘッドにならないのである．一方 LU の場合は同じアドレスを指す SIA が何度もプログラム中に現れる．そのためオフセット計算のオーバヘッドが増大したと考えられる．SIA に関する最適化を行わなければ，特に配列演算を含むようなプログラムにおいて極端に実行速度が遅くなる場合が存在する．

3.6 関数呼び出し規約

この節では関数呼び出し規約について議論する．

3.6.1 レジスタ保存戦略

あるレジスタが関数呼び出し命令を超えた生存期間を持っている場合にはそのレジスタの値を保存・復元する必要がある．レジスタを保存・復元するやり方には幾通りかの方式があり，レジスタ保存戦略と呼ばれ

ている．

機械独立アセンブラコードでのレジスタ保存戦略を議論する前に，仮想レジスタを，関数境界を越えて有効にするべきかどうかを考えなければならない．なぜなら関数境界を越えてレジスタを有効でなければレジスタ保存戦略を考える意味がないが，関数境界を越えてレジスタを有効にするのはコストがかかるからである．

仮想レジスタを実装するために，仮想レジスタと物理レジスタ（必要ならばそれに加えてメモリの一部）とを1対1に対応させる方式がある．この方式は，きわめて高速にレジスタ割当てを行うことができる特徴を持ち，関数境界を越えて仮想レジスタが有効になる．この方式で実装している場合には，レジスタ保存戦略を仮想アセンブラコード上に記述することができる．

柔軟なレジスタ割当てを行うために仮想レジスタと物理レジスタを1対1に対応させないときには，関数ごとに仮想レジスタと物理レジスタの対応が異なる場合が生じる．この場合，関数呼び出しと復帰の時点でレジスタの対応を一致させるコードを挿入する必要がある．

しかし，この処理は関数呼び出しと復帰のオーバヘッドを増大させる．また無限個のレジスタを関数境界を越えて有効にする効率的な実装は困難である．そのためMICでは，引数と返り値を渡すのに使われるものを除いてレジスタは関数に対して局所的であると定義している．これにより次のような利点がある．

柔軟なレジスタ割当て　機械独立コードから機械本来コードに変換する段階での関数内部のレジスタ割当ての処理において引数と返り値を渡すレジスタ以外のレジスタを自由に使うことができる．レジスタウィンドウなどを利用して仮想レジスタを実装することも可能である．

レジスタ保存の不必要性　仮想アセンブラコードではレジスタ保存のコードを記述する必要がない．

3.6.2　引数の受け渡し

関数の引数を受け渡す方式はプロセッサやオペレーティングシステムの慣習として定められているが，この慣習はプロセッサやオペレーティングシステムによって大きく変化する．既存のソフトウェア資産をモーバイル環境からでも利用できるようにするためには，機械独立コードから機械本来コードに変換するときに生成された機械本来コードが関数呼び出し規約の慣習を守っているのが望ましい．本研究の方式では関数呼び出し規約の慣習をほぼ遵守するコードを生成している．

自関数へ渡された引数は特殊なレジスタ (`%i0`, `%i1`, ...) によって表現される．このレジスタは添字の番号が，何ワード目の引数であるかを表している．このレジスタには代入することができない．

新しく呼び出す関数へ渡す引数も同様に特殊なレジスタ (`%o0`, `%o1`, ...) によって表現される．このレジスタには1つの制御フローの中で1度だけ代入することができ，関数呼び出し命令で終わる生存区間を持つ．また関数呼び出し命令にはすべての引数が提供されていなければならない．

関数の返り値も同様なレジスタ (`%R0`, `%R1`, ...) によって表現されている．

原則としてある引数が何ワード目の引数であるかが分かれば，その引数の受け渡す方法が分かる．したがって，上で述べた特殊な引数レジスタに対する参照や代入から，引数への関数呼び出し規約に従ったアクセスへ変換するのは可能である．

ただし，この方式では完全に関数呼び出し規約に一致させることはできない．その例を1つあげる．SPARC v8では引数の最初の6ワードをレジスタによって渡し，残りをスタックで渡すことになっている．この規則には例外があり，ちょうど6ワード目に倍精度浮動小数点数が渡されたときに，その最初のワードをレジスタで渡し，残りのワードをスタックで渡すのではなく，値全体をスタックで渡すことになっている．このため6ワード目には空白が生じることになる．SPARC v8のこの規則に対応したコード生成を行うためにはある引数が何ワード目であるかについての情報だけではなく，その引数がどんなデータ型を持っているかという情報も必要になる．

4.　MICコードの例

この章ではMICコードについて，フィボナッチ関数をコンパイルした例をあげて説明する．またMICシステムの実装の概要についても簡単に述べる．

図3にMICコードの例をあげる．図中でピリオドから始まる識別子はラベルまたは疑似命令である．関数は `.function` 疑似命令と `.end` 疑似命令に囲まれていなければならない．`.function` 疑似命令の5つのパラメータは順番に，関数名，関数の最後のラベル，関数フレームの大きさ，型，属性となっている．`.block` 疑似命令は基本ブロックの始まりを表している．その4つのパラメータは順番に，シリアルナンバ，ループのネストレベル，生存しているレジスタ，後に続く基本ブロックのシリアルナンバとなっている．この最後の情報によって制御フロー解析を行わずに済ますことができる．`cble` 命令は比較条件分岐命令である．最

初の 2 つのオペランドを比較した結果によって分岐するかどうか判定している．比較条件分岐命令は，アーキテクチャによって大きく変化するフラグの意味論を吸収し，効率の良い条件分岐を生成するために不可欠である．“/l” が付加されているレジスタの出現は，それがそのレジスタの最後の参照であることを表している．この情報と .block 疑似命令が持つ生存しているレジスタの情報によってレジスタの生存区間解析において繰り返し計算を省くことができるため，$O(n^2)$ の時間空間計算量によってレジスタの生存区間解析を求めることができる．calli 疑似命令は整数を返す関数呼び出しを表す．第 2 オペランドが引数を表し，第 3 オペランドが返り値を持つレジスタを表す．reti 疑似命令は整数を返す関数復帰を表す．

MIC コードを出力する C 言語と C++ 言語コンパイラが実装されている．このコンパイラは 2 つのソフトウェアを組み合わせて構成されている．まず GNU C Compiler 2.95.2 を改造した mic-gcc を利用して C 言語または C++ 言語のプログラムを変換し，その出力を Objective CAML によって記述された変換器 mas を通すことで MIC を出力する．

MIC から機械本来のコードを生成するには mcgen と呼ばれるソフトウェアを利用する．現在 MIC から SPARC プロセッサと IA32 プロセッサ（x86）のコードを出力することができる．この変換器も Objective CAML によって記述されている．

```
        .function fib,.Lfe1,0,(int) -> int,public
        .block 0,0,{%i0},{2,1}
        mov %i0/l,%r1
        cble %r1,1,.L3
        .block 1,0,{%r1},{3}
        add %r1,-1,%o2
        calli fib,{%o2/l},{%R3}
        add %r1/l,-2,%o4
        mov %R3/l,%r5
        calli fib,{%o4/l},{%R6}
        add %r5/l,%R6/l,%r7
        ba .L5
        .block 2,0,{},{3}
.L3:    mov 1,%r7
        .block 3,0,{%r7},{}
.L5:    reti %r7/l
.Lfe1: .endfun
```

図 3 フィボナッチ関数
Fig. 3 Fibonacci function.

5. 評価

この章では MIC の設計と実装についての評価を C 言語のプログラムを MIC コードに変換してから実行された実行時間と gcc によって直接生成されたコードの実行速度とを比較することで行う．評価は SPARC プロセッサを利用する場合は UltraSPARC 167 MHz（Solaris）上で行った．IA32 プロセッサを利用する場合は Pentium Pro 200 MHz（Solaris）上で行った．実行時間を計測したプログラムはすべて C 言語によって記述された．gcc のバージョンは 2.95.2 である．実験に使ったプログラムについて説明する．fib はフィボナッチの 35 を計算する．nqueen は縦横 12 個の版面上のもので重複を余計に計算するものである．qsort は 262144 個のランダムな整数をクイックソートで並べ変えるものである．matrix は縦横のサイズが 256 の double 型の行列どうしの乗算を行うものである．raytrace は周りを 5 面の平面で囲まれた半透明の球をレイトレーシング法により描くものである．LU は縦横のサイズ 256 の double 型の行列をピボット部分選択を行うアルゴリズムで LU 分解するものである．

MIC から機械本来のコードに変換されたコードの実行時間（A）と MIC を使わずに C 言語のプログラムから gcc でコンパイルしたコードの実行時間（B）の比率（A ÷ B）を**図 4** に示す．最適化オプションはすべて O2 で行った．実行時間は 10 回計測した平均値である．MIC コードの実行時間は最大でも 1 割程度の増加であり，半分以上の場合で gcc の性能を上回っている．IA32 アーキテクチャで gcc よりも効率の良いコードになった理由は命令の選び方にあると考えられる．Pentium Pro 以降の実装では 1 サイクルで同時に 3 命令をデコードすることができるが，そのためには単純な命令を使わなければならない．gcc は単純でない命令も多用する傾向にあり，その影響が出

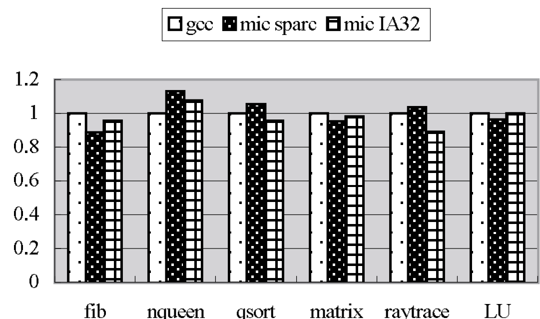

図 4 実行時間の比率
Fig. 4 Relative elapsed time.

たものであると考えられる．

6. 関連研究

この章では，異機種上で同じコードを共有することが可能な関連研究との比較を行う．

6.1 RTL

RTL（Register Transfer Language）は GNU C コンパイラ（gcc）で使われている内部表現であり，無限個のレジスタを持つ，Lisp に基づく手続き型言語である．Gcc の翻訳作業のほとんどすべては RTL のレベルで行われている．RTL を我々の目的に利用する際の問題点とは RTL が Lisp のように表現力に富んだ「式」の記述を許す点にある．したがって，式を機械本来語に変換するときに，新しいレジスタの導入，共通部分式の削除，命令スケジューリングなどの最適化を行う余地が生じる．

6.2 ANDF

OSF（Open Software Foundation Research Institute）が開発を行っている ANDF（Architecture- and language-Neutral Distribution Format）と呼ばれるコード表現[9] がある．ANDF は RTL と共通点は多いが，RTL とは違って整数とポインタを混同することができず，別々の型が与えられており，加えて構造体などのデータ型も用意されている．ANDF は，我々が MIC に対して想定しているよりもより広範なアーキテクチャに対応することを重視して設計されており，アーキテクチャによって変化するパラメータ（アラインメントなど）はインストール時に値が決定する変数または関数によって記述することになっている．したがって，高級言語から ANDF に変換する際には下部のアーキテクチャの自由度が高い分，効率の良いコードを生成するのは難しい．一方，ANDF コードから機械本来語への変換はほとんど C コンパイラを作成するのと同等なコストが生じるであろう．

6.3 Juice

Juice と呼ばれる Java に基づいたシステム[25] では抽象構文木をコード表現として採り入れている．このコード表現は同じプログラムに対して Java のバイトコードの 4 割程度の大きさであり，また情報量が多いため容易にコード検証や最適化を行うことができる．一般に高いレベルのコード表現は実行可能な形式に翻訳するコストが大きい．そのため Juice では動的コード切替えという方式を利用している．この方式では，抽象構文木から自明な方式で高速に機械語を生成し，実行を開始させる．同時にバックグラウンドでコストのかかる翻訳を行い効率の良いコードを生成させる．その後，適切な時期に元のコードを新しい効率の良いコードで置き換えている．この方式では実行環境上でコンパイラが必要となるため Java が当初想定していた実行環境である埋込機器に対応させるのは難しいが，計算資源の豊かなパーソナルコンピュータ上で Java を実行させる方式としては優れている．しかし，コード表現として特定のプログラミング言語を仮定してしまうため言語独立性を著しく損なうことになる．

6.4 Omniware と vcode

我々と同等の目的を持った研究として Omniware[1] と VCODE[11),12] と呼ばれるものがある．Omniware は gcc の出力するコードに対して数%程度のオーバヘッドで動作する機械独立コード体系と機械本来コードへの変換器を作成したが，残念ながら詳細が公開されていない．我々は文献 1) に従って彼らと同じ結果を得ようとしたが，残念ながら実現できなかった．インターネットの普及にともない異機種間でコードを共有したいという需要は多く，そのためにもコード体系とその実装が公開されていることは重要であると我々は考える．VCODE はきわめて高速なコード生成を目指しており，機械独立コードの 1 命令から機械本来のコードを生成するのを平均で機械本来命令の 6 命令によって実装している．我々の研究との違いは彼らは実行速度の速さよりも翻訳速度の速さを重視している点である．また VCODE ではアドレッシングモードとしてスケール付きインデックスアクセスを持っていないが，この場合，我々の経験によれば，特別な最適化をしなければ IA32 アーキテクチャでの実行効率が低下する．Omniware がどのようなアドレッシングモードを持っているのかは不明である．

6.5 AJIT

機械本来語を容易に生成できるようにバイトコードに注釈を付ける研究として AJIT[2] がある．AJIT は Java のバイトコードを対象として，1 命令ごとに 2 種類の情報を付加している．それは (1) オペランドの仮想的なレジスタ番号と，(2) その命令が複数の演算から構成されているときにどの演算を省けるかという情報である．前者の情報は Java バイトコードをレジスタマシンコードとして扱うものであるが，この情報によってコンパイル時に必要になる，スタック上のデータフロー解析を省くことができる．後者の情報は，たとえば，配列の要素を参照する命令において添字の境界チェックを省けるかどうかという情報である．同じ配列の同じ要素に連続して参照するときに 2 回目以降の境界チェックを省くことができる．そこでそのようなアクセスを行うときにはコンパイル時にそれを意味

する注釈を付けてしまう．AJIT ではこのような付加情報によって，彼らがベースにした JIT コンパイラよりも効率の良いコードを出力している．我々の研究との違いは，我々はレジスタに対して最後に参照される位置に印を付けることによって生存区間解析のコストを低下させている点である．

6.6 バイナリ変換

異機種間で同じ意味を持つプログラムを実行させる手法としてバイナリ変換[33)] を行うシステム[19),36),37)] が存在する．この中で文献 37) はペンティアムのコードを SPARC に変換するシステムについて記述してあり，それによると C 言語のプログラムをペンティアムにコンパイルしたコードを SPARC に変換したコードは，最初から SPARC 上でコンパイルしたコードに比べて 2.29 倍から 5.53 倍の実行時間で動作する．

バイナリ変換を行うほとんどのシステムではある特定のアーキテクチャから別のアーキテクチャへの変換（または，それにその逆変換を加えたもの）のみを与えている．複数のアーキテクチャ間での相互の変換を考えるときにある中間表現を経由することは実装の負担を軽減する働きがある．我々のコード体系はそのようなものとしても利用することができる．

7. おわりに

この研究で我々は異機種間でのモーバイル計算に向いたコード表現の設計を行い，機械独立コードから SPARC と IA32 アーキテクチャのためのコードへの変換器を実装した．また C 言語と C++ 言語から機械独立コードへのコンパイラを作成した．我々は MIC コードの性能について検証を行った．MIC コードの実行速度はその機械本来のコンパイラによって生成されたコードと同等の速度で動作し，また MIC コードから機械本来のコードの生成するにはレジスタ割当てと覗き穴最適化のみを行っており，コード生成は容易である．

MIC コードは Planet システムの異機種対応拡張[43)] や異機種対応のソフトウェア DSM システム[44)] のコード表現として利用されている．

今後の研究として次のような項目を考えている．第 1 に MIC から PowerPC や MIPS などのアーキテクチャ用のコードを生成できる生成器の実装を行う．第 2 に MIC から機械本来のコードを生成する時間を測定し，その時間を短縮する．第 3 に，アセンブラ言語上で安全性を保証する枠組みがいくつか提案されているが，それらの研究を我々の体系に適用し，コードを安全に実行する機能を考案，実装する．第 4 に C, C++ 言語以外のプログラミング言語のコンパイラを実装する．

参 考 文 献

1) Adl-Tabatabai, A.-R., Langdale, G., Lucco, S. and Wahbe, R.: Efficient and Language-Independent Mobile Programs, *Proc. ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pp.127–136 (1996).

2) Azevedo, A., Nicolau, A. and Hummel, J.: Java Annotation-aware Just-in-Time (AJIT) Compilation System, *Proc. ACM 1999 Java Grande Conference* (1999).

3) Bergner, P., Dahl, P., Engebretsen, D. and O'Keefe, M.: Spill Code Minimization via Interference Region Spilling, *Proc. ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pp.287–295 (June 1997).

4) Cai, T., Gloor, P.A. and Nog, S.: DartFlow: A Workflow Management System on the Web using Transportable Agents, Technical Report, Dartmouth College, PCS-TR96-283 (1996).

5) Cardelli, L.: Mobile Computation, *Mobile Object System: Towards the Programmable Internet*, Lecture Notes in Computer Science, Vol.1222, pp.3–6, Springer-Verlag (1997).

6) Chaitin, G.J.: Register Allocation and Spilling via Graph Coloring, *SIGPLAN Notices*, Vol.17, No.6, pp.98–105 (1982). Proceedings of the ACM SIGPLAN'82 Symposium on Compiler Construction.

7) Chess, D., Harrison, C. and Kershenbaum, A.: Mobile Agents: Are They a Good Idea?, *Mobile Object System: Towards the Programmable Internet*, Lecture Notes in Computer Science, Vol.1222, pp.25–47 (1996).

8) Chow, F.C. and Hennessy, J.L.: The Priority-Based Coloring Approach to Register Allocation, *ACM Trans. on Programming Languages and Systems*, Vol.12, No.4, pp.501–536 (1990).

9) Currie, I.F.: TDF Specification, Issue 4.0. the Defence Evaluation and Research Agency (June 1995).

10) Dean, J., DeFouw, G., Grove, D., Litvinov, V. and Chambers, C.: Vortex: An Optimizing Compiler for Object-Oriented Languages, *Proc. OOPSLA Conference on Object-Oriented Programming Languages and Systems*, pp.83–100 (1996).

11) Engler, D.R.: vcode: A Retargetable, Extensible, Very Fast Dynamic Code Generation Sys-

tem, *Proc. 23rd Annual ACM Conference on Programming Language Design and Implementation* (1996).

12) Engler, D.R. and Proebsting, T.A.: DCG: An Efficient, Retargetable Dynamic Code Generation System, *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.263–273 (1994).

13) Ernst, J., Evans, W., Fraser, C.W., Lucco, S. and Proebsting, T.A.: Code Compression, *Proc. ACM SIGPLAN'97 Conference on Programming Language Design and Implementation* (1997).

14) Ertl, M.A.: Stack Caching for Interpreters, *Proc. ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pp.315–327 (June 1995).

15) Gosling, J.: Java Intermediate Bytecodes, *ACM SIGPLAN Notices*, Vol.30, No.3, pp.111–118 (1995). ACM SIGPLAN Workshop on Intermediate Representations (IR'95).

16) Gosling, J. and McGilton, H.: *The Java Language Environment* (1995).

17) Grant, B., Mock, M., Philipose, M., Chambers, C. and Eggers, S.J.: DyC: An Expressive Annotation-Directed Dynamic Compiler for C, Technical report, Department of Computer Science and Engineering, University of Washington, UW-CSE-97-03-03 (1998).

18) Gray, R.S.: Agent Tcl: A Transportable Agent System, *Proc. CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management* (1995).

19) Hookway, R.J. and Herdeg, M.A.: Digital FX!32: Combining emulation and binary translation, *Digital Technical Journal*, Vol.9, No.1, pp.3–12 (1997).

20) Hurst, L., Cunningham, P. and Somers, F.: Mobile Agents – Smart Messages, *Mobile Agents MA'97*, Lecture Notes in Computer Science, Vol.1219, pp.111–122 (1997).

21) Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H. and Nakatani, T.: Design, Implementation and Evaluation of Optimizations in a Just-In-Time Compiler, *Proc. ACM 1999 Java Grande Conference* (1999).

22) Jones, N.D., Gomard, C.K. and Sestoft, P.: *Partial Evaluation and Automatic Program Generation*, International Series in Computer Science, Prentice Hall (1993).

23) Kato, K., Someya, Y., Matsubara, K., Toumura, K. and Abe, H.: An Approach to Mobile Software Robots for the WWW, *IEEE Trans. Knowledge and Data Engineering*, Vol.11, No.4, pp.526–548 (1999).

24) Kato, K., Matsubara, K., Someya, Y., Itabashi, K. and Moriyama, Y.: PLANET: An Open Mobile Object System for Open Network, *Proc. IEEE 1st International Symposium on Agent Systems and Applications/3rd Internaltional Symposium on Mobile Agents*, pp.274–275 (1999).

25) Kistler, T. and Franz, M.: A Tree-Based Alternative to Java Byte-Codes. *International Journal of Parallel Programming*, Vol.27, No.1, pp.21–34 (1999).

26) Lange, D. and Oshima, M.: *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley (1998).

27) Lee, P. and Leone, M.: Optimizing ML with Run-Time Code Generation, *Proc. 23rd Annual ACM Conference on Programming Language Design and Implementation* (1996).

28) Morrisett, G., Walker, D., Crary, K. and Glew, N.: From System F to Typed Assembly Language, *Conference Record of POPL'98: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1998).

29) Oliveira, L.A.G., Oliveira, P.C. and Cardozo, E.: An Agent-Based Approach for Quality of Service Negotiation and Management in Distributed Multimedia Systems, *Mobile Agents MA'97*, Lecture Notes in Computer Science, Vol.1219, pp.1–12 (1997).

30) Poletto, M. and Sarkar, V.: Linear Scan Register Allocation, *ACM Trans. Programming Languages and Systems*, Vol.21, No.5, pp.895–913 (1999).

31) Pugh, W.: Compressing Java Class Files, *Proc. ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pp.247–258 (May 1999).

32) Ranganathan, M., Acharya, A., Sharma, S. and Saltz, J.: Network-aware Mobile Programs, Technical Report, University of Maryland (1996).

33) Sites, R.L., Chernoff, A., Kirk, M.B., Marks, M.P. and Robinson, S.G.: Binary Translation, *Comm.ACM*, Vol.36, No,2, pp.69–81 (February 1993).

34) Colusa Software: Omniware: A Universal Substrate for Mobile Code, White Paper (1995).

35) Stata, R. and Abadi, M.: A Type System for Java Bytecode Subroutines, *Conference Record of POPL'98: 25th ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, pp.149–160 (1998).
36) SunSoft: Wabi (1994).
37) Ung, D. and Cifuentes, C.: Machine-Adaptable Dynamic Binary Translation, *Proc. ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pp.30–40 (Jan. 2000).
38) Wahbe, R., Lucco, S., Anderson, T.E. and Graham. S.L.: Efficient Software-Based Fault Isolation, *Proc. 14th ACM Symposium on Operating System Principles*, pp.203–216 (1993).
39) Watanabe, T., Sakuma, K., Arai, H. and Umetani, K.: Essential Language el(α) – A Reduced Expression Set Language for System Programming, *ACM SIGPLAN Notices*, Vol.26, No.1, pp.85–98 (1991).
40) White, J.E.: Telescript Technology: An Introduction to the Language, General Magic White Paper (1995).
41) White, J.E.: Mobile Agents, *Software Agents*, Bradshaw, J. (Ed.), The MIT Press (1996).
42) Yellin, F.: Low Level Security in Java, *4th International World Wide Web Conference* (1995).
43) 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：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，情報処理学会論文誌 (2000). 掲載予定
44) 上田陽平，山本泰宇，関口龍郎，米澤明憲：アセンブリ言語レベルでの異種計算機間のヒープとスタックの共有機構，*SWoPP* (Aug. 2000).

(平成 12 年 5 月 26 日受付)
(平成 12 年 9 月 8 日採録)

関口　龍郎

昭和 45 年生．平成 10 年東京大学大学院理学系研究科博士課程修了．理学博士．平成 10 年より日本学術振興会未来開拓事業特別研究員．主にモーバイル計算等の研究に従事．

米澤　明憲（正会員）

1947 年生．1977 年 Ph.D. in Computer Science（MIT）．1989 年より東京大学大学院理学系研究科情報科学専攻教授．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，元ソフトウェア科学会理事長，ACM Fellow.