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

東京大学理学系研究科情報科学専攻
113-8654 東京都文京区本郷 7-3-1
e-mail: {cocoa, yonezawa}@is.s.u-tokyo.ac.jp

概要　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、MIC コードから SPARC と x86 コードへの変換器が実装されている。加えて我々は C 言語と C++ 言語から MIC へのコンパイラを実装した。これにより C, C++ 言語を使ったプログラムでも異機種間モーバイル環境での利用が容易になる。モーバイル計算において我々が理想的と考えるコード表現が持つべき性質とは、速い実行速度、速い翻訳速度、言語非依存性、機種非依存性、安全性の 5 点である。この論文では MIC をこれらの観点から具体的に検証した結果も示す。

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) と時には実行状態までもが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 [22]。おそらく最も普及しているモーバイル計算はワールドワイドウェブ (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。

モーバイル計算はインターネット上で主として行われる。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 [6, 8, 21] では機種共通のバイトコードをインタープリタ実行していた。

しかし、インター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 [10, 16] 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来ている。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。そのような手法の中で最近注目されているモーバイル計算には次のような著しい特徴がある。それはコードが実行される時になって初めてそのコードが実行環境に送られて来るという点である。したがってコードをできるだけ速く実行可能にするのが望ましい。この意味で Java 言語のバイトコードは理想的なコード表現とは言えない。Java 言語のバイトコードから良質の機械本来のコードを生成するためには高価な解析を行う必要があり、そのような解析を避けるためにあらかじめ Java 言語のバイトコードに解析の結果を表す注釈を付けておく研究 [2] が存在する。

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 [11] の機能を持つ言語

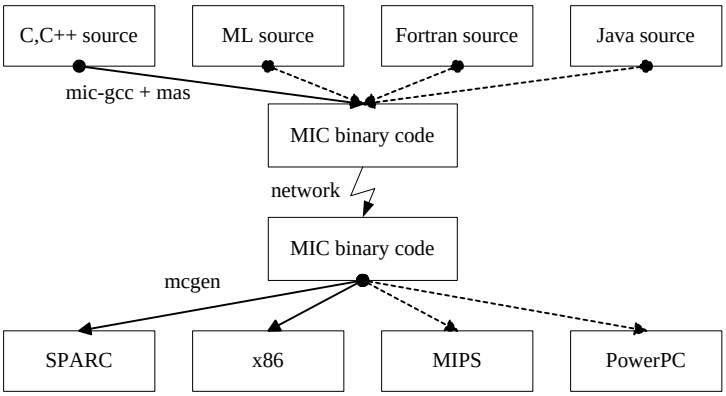


図 1: 機械独立コードを利用したモーバイルシステム

の実装 [7, 13] で行われて来ている。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ない。

本研究の意義は機械独立コード MIC と呼ばれる次のような特徴を持つコード体系を設計し、その言語処理系を実装したことにある。

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。

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時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。

機種非依存性 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。

実行速度の速さという第一番目の意義は言語一般で重視される観点である。第二番目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。この観点でよ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。第三番目の観点もモーバイル計算に関係するものである。インターネットには様々な特徴を持つ計算機がつなげら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。

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 [23] や、SFI [20] 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。

我々は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、機械独立コードから SPARC プロセッサと x86 プロセッサ用のコードに変換する変換器を実装した。

この論文の構成は次のようになっている。第 2 節

では機械独立コードを使ったモーバイルシステムの概要について説明する。第 3 節では機械独立コードの設計について議論する。第 4 節では我々の機械独立コードを評価する。第 5 節は簡単なまとめを行う。

2 機械独立コードを使ったモーバイルシステム

この節では我々の設計した機械独立コードを利用してどのようにモーバイルシステムを構築するのかについて説明する。同時に我々が行った実装についても説明する。

図 1 に機械独立コード MIC を利用したモーバイルシステムの概念図を示す。図中、実線は実装が終了しているものを意味し、破線はまだ構想段階にあるものを意味する。

機械独立コードは様々なプログラミング言語のコンパイルの標的であると想定されている。ネットワーク上を MIC binary code と呼ばれる統一のフォーマットにより移動し、実行される環境に到達すると、変換器によって機械本来のコードに変換され、実行される。

C 言語と C++ 言語から MIC へのコンパイラが実装されている。このコンパイラは二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 2.7.2.3 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 x86 プロセッサ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。

3 機械独立コードの設計方針

この節では機械独立コードの設計について議論する。

3.1 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充分条件を次のようなものと考える。なお一部の項目については第 1 節で解説されている。

- 実行速度の速さ
- 翻訳速度の速さ
- 言語非依存性
- 機種非依存性
- 安全性

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 [12, 20] が存在している。

3.2 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である。そのように設計されたのは次のような理由からである。

- 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既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。

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。

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。

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 [14, 17] 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。また抽象機械の方式はメモリを安全に管理するためにしばしばガーベージコレクションの概念を含む。

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時に重要な効果を持つ。

レジスタの数 機械独立コードでのレジスタの数はいくつであるのがよいのだろうか。またレジスタの数と密接に関係する問題としてレジスタ割り当てを翻訳時に行うべきかどうかという問題がある。

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時に重要な効果を持つ。

プロセッサの状態 機械独立コードはフラグなどのプロセッサの状態を持つべきだろうか。

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。

翻訳時の最適化 翻訳時にどの程度の最適化を行うべきだろうか。

これらの問題に対する我々の考え方を次に示す。ただしこれらの設計に関する選択は、評価する基準が複数、存在し、また相互に深い関係にあるので決定が容易ではない。そこである一つの原則を天下り的に設定することとする。それは

> 翻訳時の最適化では関数内部の最適化を行い、関数をまたがる最適化は行わない

というものである。

3.3 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, 4] では RISC を採用している。文献 [1] によれば x86 アーキテクチャの実装の多くは RISC に基づいたスーパースカラであり、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一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 [3] ではプロファイリングやモジュール間インライン展開によって実行効率を向上させている。しかし、コードが一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で済ませるという原則に反する。

3.4 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 [17] 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 [14] 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、メモリの割り当て、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 [20] や PLANET [12] など少ない実行時オーバー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。(例えば EmacsLisp のバイトコードなど。)

3.5 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、2-アドレス形式のレジスタマシン、3-アドレス形式の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。すなわちスタックトップのトレースとデータフロー解析を行わなければならない。我々は高価なデータフロー解析を避けるため MIC をレジスタマシンとして設計した。

同じレジスタマシンであっても SPARC は 3-アドレス形式であり、x86 は 2-アドレス形式である。我々は 2-アドレス形式と 3-アドレス形式を相互に変換する方式について調べ、2-アドレス形式から 3-アドレス形式に変換する方式ではしばしば局所的な情報だけでは効率良く変換するのが難しいことを発見した。例えば次のような変換例を考える。

$$\texttt{mov}\ r_0, r_2 \Longrightarrow \texttt{add}\ r_0, r_1, r_2$$
$$\texttt{add}\ r_1, r_2$$

ただし 2-アドレス形式は第一オペランドを入力とし、演算結果を第二オペランドに書き込むとする。3-アドレス形式は第一オペランドと第二オペランドの間で演算を行い、結果を第三オペランドに格納する。左のような命令列は 3-アドレス形式では一命令に変換することができ、そうするのが望ましい。しかしこのように変換するのは一般的には難しく、mov 命令と add 命令の間に無関係な命令が挿入されていた場合などに対応するためには命令列の解析が必要になる。

一方、逆方向の変換を考えると、

$$\texttt{add}\ r_0, r_1, r_2 \Longrightarrow \begin{array}{l}\texttt{mov}\ r_0, r_2 \\ \qquad\texttt{add}\ r_1, r_2\end{array}$$

これは容易である。以上のような理由から MIC は 3-アドレス形式のレジスタマシンとして設計された。またデータフローに関する解析を避けるためにレジスタには生存期間や最後に使用される出現などの情報が付加的されている。

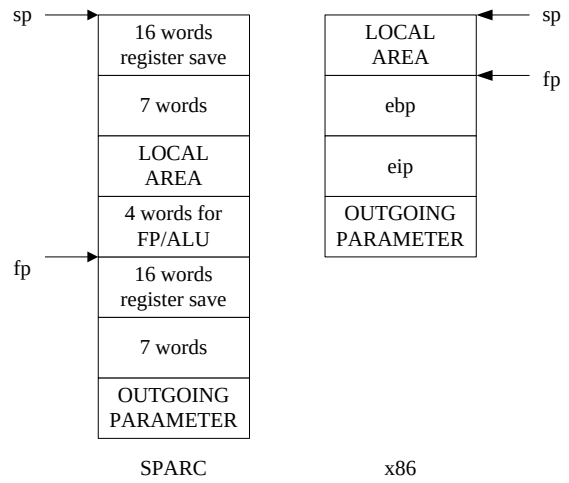


図 2: SPARC と x86 のスタックフレーム

3.6 レジスタの数

機械独立コードの仮想レジスタの数がプロセッサのレジスタの数よりも少ないときには本来不必要な溢れが発生する。溢れを含むようなコードを解析して溢れたデータフローを物理レジスタにマップさせるような解析を行うのは困難である。

Omniware では一般レジスタの数は 16 になっている。文献 [1] によるとこの理由はこれ以上レジスタの数を増やしても実行効率の向上が見られないからである。ただし彼らがどんなアプリケーションでこの測定を行ったのかは記述されていない。Omniware では翻訳時のレジスタ割り当てを行っていない。

VCODE でのレジスタの数は無限大であり、翻訳時にレジスタ割り当てを行うことも選択できる。レジスタ割り当てを行わない場合にはレジスタにあらかじめ割り当てられている優先順位の順番に物理レジスタを割り当てていき、足りなくなったらメモリの特定の場所を割り当てるようになっている。

現在ところ MIC では一般レジスタの数は 16 であるが、無限個にするべきであると考える。インターネットに接続されているコンピュータのプロセッサの持つレジスタの数は機種によって様々に変化する。したがって我々はそれぞれの機種の能力を最大限に引き出すためには機械本来のコードに変換する段階でレジスタ割り当てを避けることはできないと考える。

3.7 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 x86 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良く使われるアドレスの形式である。CISC プロセッサとして x86 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一つ用意し、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時には、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 x86 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一つ割り当てて、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一つ取られることである。x86 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時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時に、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、x86 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時に、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 を採用していないが x86 のための最適化も行っていない。これを行えば x86 の場合の効率が改善するのではないかと思われる。

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。

現在の MIC では SIA を採用しており SPARC のための最適化を行っている。このために MIC は SPARC よりも少ないレジスタを持ちながら SPARC 上の翻訳でもレジスタ割り当てが必要になる。

3.8 プロセッサの状態

文献 [1] でも指摘されているがプロセッサの状態の一つであるフラグの意味論はプロセッサによって大きく変化する。RISC プロセッサのほとんどはフラグを持たないか、フラグに影響を与えない算術命令が存在するかのいずれかである。機械独立コードがフラグを持つ場合に、フラグの意味を正確かつ効率的に実装するのは難しい。そのためプロセッサの状態と、副作用として状態に影響を与える命令は他に支障がない限り、少ない方が良い。

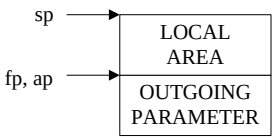


図 3: MIC のスタックフレーム

3.9 関数呼び出し規約

3.9.1 レジスタ保存戦略

あるレジスタが関数呼出し命令を越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) とを一対一に対応させる方式がある。この方式は、極めて高速にレジスタ割り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。

柔軟なレジスタ割り当てを行うために仮想レジスタと物理レジスタを一対一に対応させない時には、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。

柔軟なレジスタ割り当て 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り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。

3.9.2 スタックフレームの構造

関数のスタックフレームは一時的に値を退避させたり、呼び出す関数に値を受け渡したりするのに用いられる。スタックフレームの扱い方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。この節では SPARC と x86 プロセッサの両方のスタックフレームを機種非依存コードから統一的に扱う方法について記述する。

この節に記述されている方法では残念ながらすべてのプラットフォーム上の機械本来の関数呼び出しの慣習と完全に一致させることはできない。それにも関わらず統一的な方法を用意するのは次のような利点のためである。

- スタックフレームの特定の箇所はあらかじめプロセッサやオペレーティングシステムで定められた目的に使用されるために予約されている。機械独立コードからスタックフレームを統一的に扱えるためにはそれらの予約領域を避けて扱うようにしなければならない。その結果個別の実行環境では未使用の領域が存在していることになる。我々の方式ではこのような領域を省き、使用するスタックフレームの大きさを小さくすることができる。

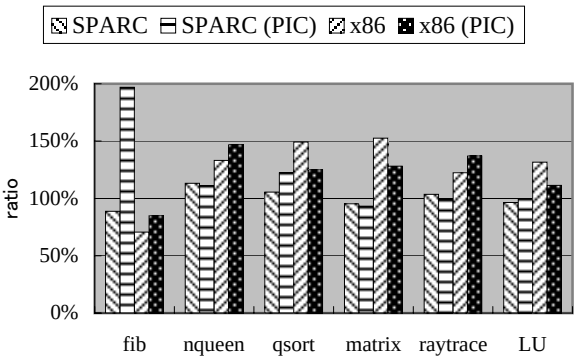


図 4: 実行時間の比率

- 状態の移動を行うモーバイル言語では関数の状態であるスタックフレームを移動させる場合がある。スタックに対して機種独立に扱えなければスタックフレームの符号化、復号化を実装するのが難しい。

図 2 に SPARC と x86 におけるスタックフレームの構造を示す。ただし fp と sp はそれぞれのプロセッサでのフレームポインタとスタックポインタである。スタックフレームを統一的に扱う時に生じる問題とは局所使用領域とスタック上の引数の領域へのオフセットが SPARC と x86 で異なっていることである。例えば SPARC では最初の局所領域のアドレスは $[\mathrm{fp}-20]$ であり最初の引数のアドレスは $[\mathrm{fp}+92]$ であるが、x86 ではそれぞれ $[\mathrm{fp}-4]$ と $[\mathrm{fp}+8]$ となる。局所領域と引数領域へのアクセスでフレームポインターに加算されるオフセットが異なっている。局所領域へのアクセスは必ずしも $[\mathrm{fp}-X]$ という形を取らず、一般レジスタに fp の値をコピーしてからアクセスすることもありうるため、あるメモリアクセスが局所領域を触っているかどうか判別するのは難しい。したがって異なるプロセッサのスタックフレームを機械独立コードから統一的に扱うことは困難である。

MIC ではレジスタごとに明確に役割を定め、アクセスする場所ごとに専用のレジスタを使用することで機械独立コードからスタックフレームを統一的に

扱っている。図 3 に MIC が想定しているスタックフレームの構造を示す。ap とはスタック上の引数のアドレスを指す特殊なレジスタを表す。fp と ap は概念上は全く同じアドレスを指している。各レジスタの役割を次のように定める。

sp レジスタ 関数を呼び出す時にスタックに引数を積むのに用いる。引数は [sp], [sp + 4], ... と続く。

fp レジスタ 関数の局所使用領域にアクセスするのに用いる。局所使用領域は [fp − 4], [fp − 8], ... と続く。

ap レジスタ 関数自身の引数にアクセスするのに用いる。引数は [ap], [ap + 4], ... と続く。

このように役割を決めてしまえば各レジスタを機械本来のレジスタに対応させる時に適当なオフセットを加えることで機械本来の慣習と一致させることができる。具体的には次のように変換する。

MIC	SPARC	x86
[sp]	[sp + 92]	[sp]
[fp]	[fp − 16]	[fp]
[ap]	[fp + 92]	[fp + 8]

もちろんレジスタにオフセットが付けられた状態で使う時にはそのオフセットも加えることになる。例えば [fp − 4] は SPARC では [fp − 20] となる。

局所使用領域に配列などを確保した場合に fp レジスタを一般レジスタにコピーしてアクセスする場合がある。この時にも fp レジスタに適切なオフセットを加えて正しい領域を指すように補正を行う。

3.10 翻訳時の最適化

翻訳時にどこまで最適化をすればいいのであろうか。関連研究を参照すると Omniware では以下のような最適化を行っている。

- MIPS と PowerPC では局所的命令スケジューリングを行う。
- SPARC では大域的ポインタと遅延スロットの充填を行う。ただし大域的ポインタとは 32 bit の即値のアドレスに関する最適化である。
- x86 では浮動小数点パイプラインスケジューリングと除き穴最適化を行う。

また vcode は極めて高速なコード生成を目指しており原則として最適化は行わない。文献 [1] によると Omniware は今後、大域的命令スケジューリングと機械依存除き穴最適化を行うと述べられている。

我々は、関数をまたがる大域的な最適化はとりわけ計算量が増大するため行わず、関数内部の局所的な最適化は行うという原則を持っている。しかし、現在の MIC の翻訳器の実装ではレジスタ割り当てと遅延スロットの充填を除く最適化は行っていない。

4 評価

この節では MIC の設計と実装についての評価を行う。評価は SPARC プロセッサを利用する場合は UltraSPARC 167MHz (Solaris) 上で行った。x86 プロセッサを利用する場合は Pentium Pro 200MHz (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。コンパイルだけを行ったプログラムには C++ 言語で書かれたものがあった。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び変えるものである。matrix は縦横のサイズが 256 の double 型の行列同士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。

最初に MIC から機械本来のコードに変換されたコードの実行効率を評価する。これは 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その機

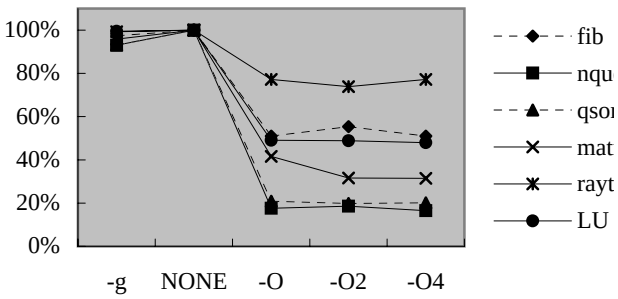
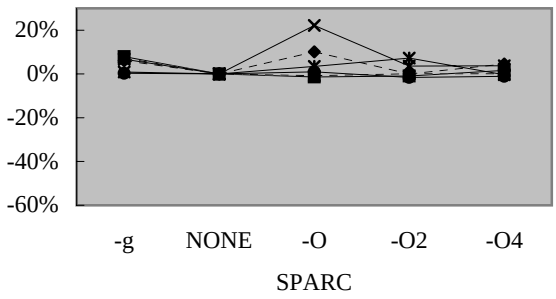


図 5: gcc の最適化による実行時間の比率 (SPARC 上)

種の C 言語コンパイラ (gcc を使用) でコンパイルしたコードの実行時間 (B) を比較することによって行う。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。図 4 がその比率 (A ÷ B) を表にしたものである。PIC は位置非依存のコードを意味する。fib において SPARC-PIC の場合に MIC が極端に遅くなっているがそれ以外は遅くても実行時間の増加は 6 割未満であり、中には MIC を使った方が速くなっている場合もある。一般的に x86 プロセッサの場合に遅くなる傾向があり、これは MIC が x86 プロセッサの特徴を充分に把握していないことを表していると思われる。また現在の実装では x86 コードを生成する変換器にやや問題があり、多くの場合 x86 コードを生成する時に 6 つの一般レジスタしか使わない。これを 8 つすべて使うようにすれば更に効率は上がると思われる。

次に MIC が、最適化が意味を持つコード体系であるかを判定する予備的な実験を行う。最適化がコード体系の上で意味があるかどうかを検証するのは難しく、この実験はあくまでも予備的なものである。図 5 は SPARC 上の gcc を使って最適化のレベルによってプログラムの実行時間がどのように変わるかを調べたものである。ただし最適化オプションを付けなかった時の実行時間を 1 として正規化してある。この図によれば gcc の最適化アルゴリズムは nqueen に対して最も有効であり、raytrace に対してはあまり有効でないことが分かる。



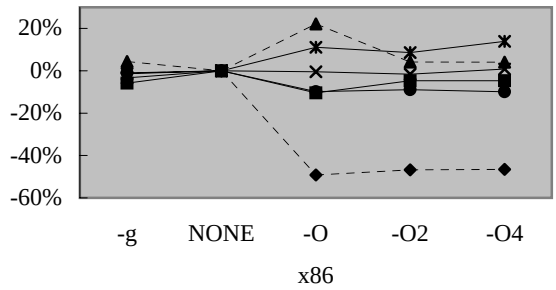


図 6: MIC と gcc の最適化による実行時間の比率の差

この同じグラフを MIC を経由して生成された SPARC のコードと x86 のコードに対して作成し、それらと機械本来の gcc が生成したコードとの差分を求めたグラフが図 6 である。グラフの下方向に行くほど MIC の最適化が gcc に勝り、上方向に行くほど gcc の最適化が勝っていることを意味する。これらの図によれば MIC 上で行った最適化は機械本来のコードで行った最適化とほぼ同程度の有効性を持っており、悪くとも 20 % 未満の効率低下で済んでいることが分かる。

最後に MIC から機械本来のコードを生成する時間を測定した。これを図 7 に示す。ただし測定結果にはファイルを生成する時間も含まれている。対象とした MIC コードは 10 種類の C 言語または C++ 言語のプログラムを 10 通りの異なったやり方でコンパイルしたものである。それは 5 通りのコンパイルオプション (-g, なし, -O, -O2, -O4) と PIC モードか否かの組み合わせである。横軸はデータフローのノードの数であり、MIC コードに現れるレジスタ

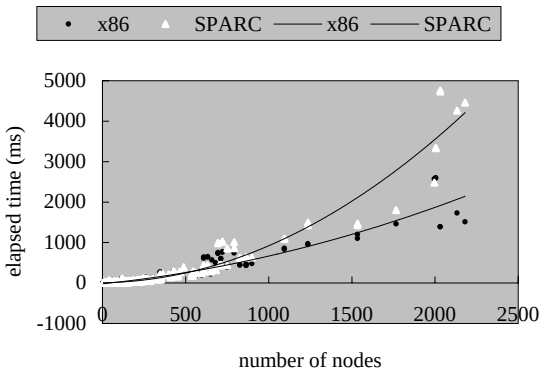


図 7: MIC から機械本来のコードへの変換時間

の出現にほぼ対応する。参考までに LU 分解を行うプログラム全体でノード数は 617 であり、2000 個のノードを含む別の関数において基本ブロックの数は 200 前後であった。

この測定によると 1000 ノードの MIC コードから機械本来のコードを生成するのにおよそ 0.6 〜 1 秒ほどかかっている。別に行った測定によれば機械本来のコードを生成する時間のうちのおよそ 6 割以上はレジスタ割り当てに費されている。現在、コード生成器の中のレジスタ割り当ての方式はグラフ彩色アルゴリズムをナイーブに実装している。これをより洗練されたレジスタ割り当てのアルゴリズムに置き換えることによって更に高速化できるものと思われる。

5 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 x86 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比肩しうる速度で動作し、また MIC コードから機械本来のコードの生成も容易である。

我々と同等の目的を持った研究として Omniware [1] と vcode [4, 5] 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。vcode は極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。また彼らは機械独立なスタックフレームの表現を考慮しておらず実行状態のマイグレーションを彼らのシステム上に実装するにはやや問題がある。

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 [15] を行うシステム [9, 18, 19] が存在する。この中で文献 [19] 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。

本研究では異機種間で同じ意味を持つコードを動作させる手法について述べたが、外部環境へのインターフェースを同じコードから扱う仕組みを用意しなければモーバイルコードから意味のある計算を行うことはできない。本研究ではベンチマークを行うために libc の一部の関数を同じモーバイルコードから呼び出せるスタブ関数群の実装を行っているが、実用的なモーバイル言語システムを作るためには安全性を考慮した上でより本格的なものを作成する必要があると思われる。

今後の研究として次のような項目を考えている。第一に 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。第二に MIC から生成されたコードの実行速度が機械本来のコンパイラで生成されたコードよりも遅くなる原因を解明し、より高速化につとめる。第

三に MIC から機械本来のコードを生成する時間を短縮する。第四に VCODE などとの関連研究と詳細な比較を行う必要があると考える。

参考文献

[1] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and Language-Independent Mobile Programs. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 127–136, 1996.

[2] Ana Azevedo, Alex Nicolau, and Joe Hummel. Java Annotation-aware Just-in-Time (AJIT) Compilation System. In *Proceedings of ACM 1999 Java Grande Conference*, 1999.

[3] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Creig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *Proceedings of OOPSLA Conference on Object-Oriented Programming Languages and Systems*, pages 83–100, 1996.

[4] Dawson R. Engler. VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System. In *Proceedings of the 23rd Annual ACM Conference on Programming Language Design and Implementation*, 1996.

[5] Dawson R. Engler and Todd A. Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generation System. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 263–273, 1994.

[6] James Gosling and Henry McGilton. The Java Language Environment, 1995.

[7] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: An Expressive Annotation-Directed Dynamic Compiler for C. Technical report, Department of Computer Science and Engineering, University of Washington, 1998. UW-CSE-97-03-03.

[8] Robert S. Gray. Agent Tcl: A Transportable Agent System. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management*, 1995.

[9] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997.

[10] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler. In *Proceedings of ACM 1999 Java Grande Conference*, 1999.

[11] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993.

[12] Kazuhiko Kato, Katsuya Matsubara, Yuuichi Someya, Kazumasa Itabashi, and Yutaka Moriyama. PLANET: An Open Mobile Object System for Open Network. In *Proceedings of IEEE First International Symposium on Agent Systems and Applications / Third Internaltional Symposium on Mobile Agents*, pages 274–275, 1999.

[13] Peter Lee and Mark Leone. Optimizing ML with Run-Time Code Generation. In *Proceedings of the 23rd Annual ACM Conference*

on Programming Language Design and Implementation, 1996.

[14] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *Conference Record of POPL'98: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[15] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary Translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[16] Colusa Software. Omniware: A Universal Substrate for Mobile Code, 1995. white paper.

[17] Raymie Stata and Martín Abadi. A Type System for Java Bytecode Subroutines. In *Conference Record of POPL'98: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 149–160, 1998.

[18] SunSoft. Wabi, 1994.

[19] David Ung and Cristina Cifuentes. Machine-Adaptable Dynamic Binary Translation. In *Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 30–40, January 2000.

[20] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, 1993.

[21] James E. White. Telescript Technology: An Introduction to the Language, 1995. General Magic white paper.

[22] James E. White. Mobile Agents. In Jeffrey Bradshaw, editor, *Software Agents*. The MIT Press, 1996.

[23] Frank Yellin. Low Level Security in Java. In *Fourth International World Wide Web Conference*, 1995.