JAIST Repository
https://dspace.jaist.ac.jp/

Title	動的コード生成を用いた適応的移動コードに関する研究
Author(s)	川崎，大輔
Citation	
Issue Date	1999-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1234
Rights	
Description	Supervisor:渡部 卓雄，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

動的コード生成を用いた
適応的移動コードに関する研究

指導教官　渡部卓雄 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻 言語設計額講座

710032 川崎大輔

1999年2月15日

Copyright © 1999 by Daisuke KAWASAKI

要 旨

本稿では、静的に決定される情報を元に実行環境に適応したコードを、実行時に生成することで、記述性を確保しながら、処理速度の向上を得るための手法の提案を行う。

移動コード等のモバイルコンピューティングの分野において、ソフトウェアが計算機環境に動的適応することの必要性が指摘されており、様々な角度から研究が行われはじめている。本研究では、移動コードに動的コード生成の手法を取り入れることにより、これに対処する。また、実際に処理系のプロトタイプを作成し有効性の検証を行う。

目 次

1 はじめに **1**
1.1 本研究の背景 1
1.2 本研究の目的 2
1.3 本論文の構成 3

2 移動コードについて **4**
2.1 移動コード 4
2.1.1 従来のアプローチ 4
2.1.2 新しいアプローチ 5
2.2 移動コードを用いることの利点 6
2.2.1 通信回線の効率的な利用 6
2.2.2 カスタマイズが可能 7
2.3 既存の移動コードについて 8

3 計算機環境に適応可能なソフトウェア **9**
3.1 適応の必要性 9
3.2 適応の記述 10
3.3 環境の変化 11

4 動的コード生成 **13**
4.1 動的コード生成 13
4.1.1 VCODE 13
4.1.2 FABIUS 14
4.1.3 Auslander96 16
4.2 まとめ 17

5 処理系の構成 **19**
5.1 適応的移動コードについて 19
5.2 全体像 19
5.3 言語拡張 20
5.4 処理系の動作原理 21
5.5 ディスパッチの仕組 23

6 結果 **27**
6.1 実験 27
6.2 考察 27

7 まとめ **32**
7.1 まとめ 32
7.2 今後の課題 32

謝辞 **34**

付録１ **37**

付録２ **39**

図 目 次

2.1 リモートプロシージャコール 5
2.2 リモートプログラミング 6

3.1 適応の例 10
3.2 記述の分離 11

4.1 移動コードの構造 18

5.1 全体の処理の流れ 20
5.2 class の構造 25
5.3 KS1_MachineArc_ScreenWidth が参照するコンスタントプール 26

6.1 PC/AT 互換機 29
6.2 Ultra Enterprise 3000 30

表 目 次

5.1 取得できる計算機環境 22

6.1 評価に使用した環境 28
6.2 計測結果 31

第 1 章

はじめに

1.1 本研究の背景

Telescript[1]、Agent Space[2]、Java[3] 等に見られるように、移動コードはネットワーク上のアプリケーションに新しいアプローチをもたらした。これまでのネットワークアプリケーションは、リモートプロシージャコール(RPC) を主に用いている。RPC パラダイムは1970 年代に発明され、コンピュータ間の通信を、一方のコンピュータから他方のコンピュータのプロシージャを呼び出すようなものだと考えた。ネットワークを通じて運ばれるメッセージは、プロシージャへのリクエスト、もしくはプロシージャの実行結果のレスポンスである。リクエストにはプロシージャと、引数としてのデータが含まれる。レスポンスには実行結果のデータが含まれており、プロシージャそのものはそれを実行するコンピュータ側に有った。リモートプロシージャコールでは、１回のコンピュータのインタラクションに２回の通信が必要になるのが大きな特徴である。最初にサーバに対してプロシージャ実行を依頼し、つぎにサーバから動作結果が送られてくる。この間、「リモートプロシージャコールの処理中ずっとコンピュータ同士が接続されていなければならない」のである。これに対し、リモートプログラミング(RP) では、コンピュータ間の通信において、あるコンピュータから他のコンピュータに対してプロシージャをコールするだけではなく、実行されるプロシージャそのものを転送してしまうことができる。リモートプログラミングの顕著な特徴として、ユーザーのコンピュータとサーバとの間でネットワークを介し、１度移動コードを送れば、その後ネットワークを切断しても処理が継続される。つまり「処理中にコンピュータ同士が接続されている必要がない」のである。これが従来なかなかできなかったことであり、移動コードのもっとも顕著な利点である。

移動コードは、様々な計算機環境でアプリケーションの実行を可能とした。しかし、様々

な計算機環境において実行することはできるようになったが、様々な計算機環境に適応した振る舞いを行うという点について考慮されていない。移動先の環境は、ほとんどの場合、その計算機の使用目的によって実に様々である。例えば、OS、CPU、ネットワーク環境、mpeg エンコーダ等の特殊なデバイス等があげられる。この時、移動先の環境に沿った動作を行うことで、実行効率の改善や、容易なオペレーションが得られる場合がある。このような場合、環境の変化にアプリケーションが対応していくことは有効である。

しかし、環境の変化に対応できるアプリケーションを作成するのは容易ではない。様々な環境の変化に対応するためには、あらゆる環境を想定し、その環境に対応した複数のコードを書かなくてはいけない。また、適応に関する部分を混在することは、プログラムの可読性と拡張性を低下させ、その保守を困難にしたり、適応の記述の共有化や、再利用を困難にする。

そこで、本研究では、環境の変化に着目した。環境の変化は、「常に変化する可能性のある環境」と「マシン固有の環境」に大別できる。環境の変化に動的に応じて、ディスパッチを行ない、現在の環境に最適なメソッドを呼び出すことで、どちらの環境の変化にも対応することはできる。しかし「マシン固有の環境」で、毎回ディスパッチを行なうことは無駄である。マシン固有の環境としては、OS、CPU、ネットワーク環境、特殊なハードウェアプロセッサといったものが上げられる。また、最近は画面サイズや、色数が動的に変更できる機構も OS に備わってきているが、実際動的に変更することは希である。つまり、移動先において適応の対象となる環境情報は、移動コードが実行環境に移動した時点で、殆ど静的に決定することができる。そこで、移動コードを実行する前に、実行環境に応じた新しいコードを生成することで、ディスパッチを行うオーバーヘッドを無くすことができる。

この事を上の例に当てはめて考えてみる。実行時に静的に決定される情報を用いて、実行環境に応じたコードを生成することにより、例で述べた両方の長所を保ちながら、欠点を取り除くことが可能である。

1.2 本研究の目的

本研究では、環境の変化に注目し、変化する環境の中で静的に決定される情報を元に、実行環境に適応したコードを実行時に生成することで、記述性を確保しながら、処理速度の向上を得るための手法の提案を行う。また、Java を用いてプロトタイプとして作成し、有効性の検証を行う。

1.3　本論文の構成

２章で移動コードの有効性についてまず述べる。その上で、３章において移動コードに対して計算機環境に適応する機構の必要性について考察する。４章では、移動コードに適応機構を持たせるにあたって非常に有効な手段であると考えられる、動的コード生成について述べる。５章で、今まで述べてきた方法を用いて実際に処理系の設計を行っていく。６章で、作成した処理系について考察する。

第 2 章

移動コードについて

2.1 移動コード

これまでの通信技術に関する簡単な説明を行い、移動コードの特徴と、これまでの通信技術との比較を行う。

2.1.1 従来のアプローチ

現在のコンピュータ通信ネットワークは、リモートプロシージャコール(RPC)を基本としている。RPCパラダイムは1970年代に発明され、コンピュータ間の通信を、一方のコンピュータから他方のコンピュータのプロシージャを呼び出すようなものだと考えた。ネットワークを通じて運ばれるメッセージは、プロシージャへのリクエスト、もしくはプロシージャの実行結果のレスポンスである。リクエストにはプロシージャと、引数としてのデータが含まれる。レスポンスには実行結果のデータが含まれており、プロシージャそのものはそれを実行するコンピュータ側にある(図2.1)。

リモートプロシージャコールに従って通信を行う2つのコンピュータは、リモートアクセスするプロシージャがシステムにどういう影響を及ぼすか、またプロシージャへの引数および、どのような結果を返すかということについて、あらかじめ決定しておかなければならない。この取り決めが「プロトコル」である。

ユーザーのコンピュータは、サーバに対して一連のリモートプロシージャコールを行うことで仕事を行う。プロシージャがコールされる毎に、毎回ユーザーからサーバに対するリクエストが送られ、サーバからユーザにはその応答が返される。たとえば、あるサーバに存在するファイルに対して、2ヶ月以上前のタイムスタンプを持つファイルを削除

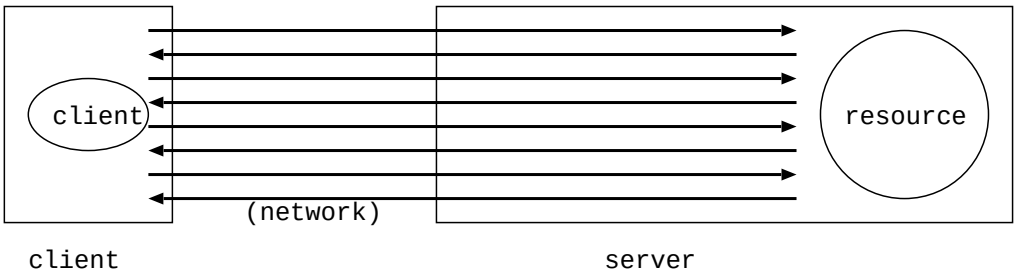

図 2.1: リモートプロシージャコール

する場合を考えてみる。まず、ユーザーのコンピュータからプロシージャコールを行い、ファイルと日付をサーバから返してもらう。そして、次のプロシージャコールで古いファイルを削除することができる。この方法では、どのファイルを削除するかという判断が、ユーザー側のコンピュータで行われている。もし、n 個のファイルを削除するのであれば、ユーザーのコンピュータはサーバと $2 \times (n+1)$ 回のメッセージのやり取りが行われることになる。

リモートプロシージャコールでは、1回のコンピュータのインタラクションに2回の通信が必要になるのが大きな特徴である。最初にサーバに対してプロシージャ実行を依頼し、つぎにサーバから動作結果が送られてくる。この間、「リモートプロシージャコールの処理中ずっとコンピュータ同士が接続されていなければならない」のである。

2.1.2 新しいアプローチ

リモートプロシージャコールに代わる方法として、リモートプログラミング(RP)がある。リモートプログラミングでは、コンピュータ間の通信において、あるコンピュータから他のコンピュータに対してプロシージャをコールするだけではなく、実行されるプロシージャそのものを転送してしまうことができる(図2.2)。

ネットワーク間でやり取りされる各メッセージには、受け手のコンピュータで処理が行われるプロシージャそのものと、それに与えられる引数としてのデータが含まれている。

RP パラダイムによる2つのコンピュータの通信には、プロシージャの中にどういう命令やデータタイプがあるかということについての取り決めが必要である。そして、その取り決めは「言語」の形で与えられる。その言語を使うことで、プロシージャで判断を行ったり、状態を変更したりすることができ、更に重要なことには受手側のコンピュータでプロシージャを実行することができるのである。この場合プロシージャコールは、リモート

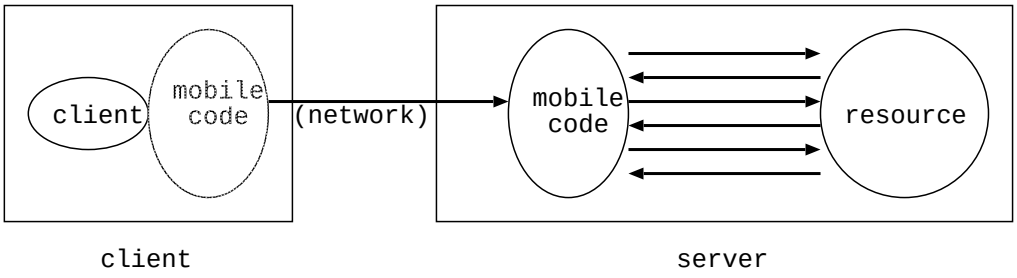

図 2.2: リモートプログラミング

ではなくローカルに行われる。このようなプロシージャを移動コードと呼ぶ。

たとえば２ヶ月以上前のファイルを削除する例では、ユーザーのコンピュータはサーバに、サーバで実行されるプロシージャ（たとえば「削除」）と内部状態（たとえば「２ヶ月以前のファイル」）をもつ移動コードを送り込むことになる。該当するファイルが複数あったとしても、１つの移動コードを送るだけでよい。

リモートプログラミングの顕著な特徴として、ユーザーのコンピュータとサーバとの間でネットワークを介し、１度移動コードを送れば、その後ネットワークを切断しても処理が継続される。つまり「処理中にコンピュータ同士が接続されている必要がない」のである。これが従来なかなかできなかったことであり、移動コードのもっとも顕著な利点である。

2.2 移動コードを用いることの利点

移動コードを用いることの利点、つまり、リモートプログラミングがリモートプロシージャコールに比べて本質的に優れている点として次の二つがあげられる。

- 通信回線の効率的な利用
- カスタマイズが可能

2.2.1 通信回線の効率的な利用

通信回線の効率的な利用とは、ユーザーのコンピュータでサーバにやらせたい仕事が合った場合、ネットワーク上でコマンドのやり取りを行う代わりに、移動コードを送るだけでよい。そのため仕事をローカルにまとめることができ、ネットワークに流れるメッ

セージも少なくてすむ。仕事が大きくなればなるほど、リモートプログラミングでメッセージを節約することができる。

リモートプログラミングによる回線利用効率の利点は、使用するネットワークによって変わってくる。ネットワークのスループットが低かったり、混雑していたり、コストが高いといった場合には、この利点は大きくなる。イーサネットよりも公衆電話回線の方が、このパラダイムの恩恵を大きく受け取ることができる。また、今日PHSなどの無線通信のほとんどでは、特に有効であると考えられる。パーソナルコミュニケータのように、パソコンに比べてスピードが遅くコストの高いネットワークの機器にはリモートプログラミングは特に適している。また、家庭のパソコンのように、通常の電話と回線を共有している場合にも有効である。

ネットワークに常時接続されていないコンピュータの例としては、家庭のコンピュータがある。リモートプログラミングを使えば、そのようなコンピュータからでもユーザーは移動コードを送ることで数多くの仕事を行わせることができる。コンピュータは、その移動コードを送り、結果を受け取るときだけネットワークにつながっていればよい。移動コードが仕事を行っているときは、コンピュータはネットワークにつながっていなくてもよいのである。

2.2.2 カスタマイズが可能

リモートプログラムの特徴として、カスタマイズ（拡張）が可能であることがあげられる。前のファイルの例で見てみる。サーバにユーザーファイルをリストするプロシージャと、ファイルの名前を与えて削除するというプロシージャがあった場合、ユーザは、ある特定の年月を経たファイルを削除するというプロシージャを追加することができる。

リモートプログラミングは、ソフトウェア作成者の仕事の分野を変えるだけでなく、出来上がったソフトウェアのインストールも簡単にすることができる。パソコンのスタンドアローンアプリケーションとは異なり、パーソナルコミュニケータのための通信型アプリケーションは、サーバ上にもそのアプリケーションの一部がなければならない。リモートプロシージャコールを使うアプリケーションのサーバ側の部分については、管理者があらかじめインストールしておく必要がる。それに対し、リモートプログラミングベースのアプリケーションでは、そのアプリケーションだけでサーバ側の部分をインストールすることができる。

2.3 既存の移動コードについて

現在移動コードというカテゴリに含まれる言語は、Java、Self-Tcl、Safe-Python、Scheme48、Oblique、Guile、Logicware、April、Joule、Penguin といったものがある。

このように、移動コードは今後更に発展が期待される技術である。

第 3 章

計算機環境に適応可能なソフトウェア

２章において移動コードの利点について述べた。３章では移動コードの問題点を指摘し、その問題点を解決するために必要な方法について述べる。

3.1　適応の必要性

近年の急速な技術革新により、計算機の小型化、高性能化が進み、形態可能な小型計算機の実用性が非常に高まった。この技術革新は計算機だけにとどまらず、携帯電話の高性能化をも促した。その結果、形態可能な計算機と、無線通信を用いた新しいコンピューティングスタイルがうまれた。このように移動可能な小型計算機をネットワークに接続し、分散環境と融合させた移動計算機環境がさかんに行われるようになった。

２章で述べた移動コードは、こうした分散環境において非常に注目されている技術である。しかし、このような移動計算機環境では、その時々の計算機の使用形態、使用場所、使用目的に応じて、計算機システムの構成や環境は様々に変化するが、こういった環境の上で動作する移動コードを含むソフトウェアは、環境に応じて変化するわけではない。簡単な例を用いてみることにする。

今、形態端末であるPDA上において、画像をJPEGへエンコードしたいとする。JPEGのエンコーディングには普通大変な計算機資源を必要とするため、PDAでエンコードを行うことはあまり得策とはいえない。そこで、比較的大きな計算機資源を持った計算機環境でエンコードを行うという考えは分散コンピューティングの立場からも自然なことである。

3.2 適応の記述

計算機環境に動的に適応するソフトウェアを記述する場合、適応の記述方法が問題になる。そこで、先程の例を用いて考えてみる。PDA においてエンコードを行うのはあまり得策ではないので、ワークステーション等の計算機で移動コードを用いてエンコードを行う(図3.1)。このとき、移動先の計算機がもし、ハードウェアエンコーダを持っていた場合、もちろんハードウェアエンコーダを用いて処理を行いたいが、ハードウェアエンコーダを持っていなかった場合には、移動コード自身もしくは、その計算機環境に用意されている API 等を用いてエンコードを行わせたいとする。このように、移動先の計算機環境

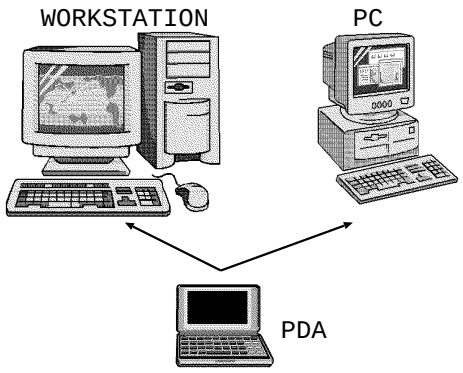

図 3.1: 適応の例

に従って、動的に適応可能なソフトウェアを記述するとき、実直なプログラムを記述した場合(図3.2-(a))、適応の手続きの記述と、適応動作の記述が混在することになる。これらの記述の混在は以下の問題を引き起こす。

- プログラムの可読性と拡張性を低下させ、その保守を困難にする。
- 適応の記述の共有、再利用を困難にする。

このため本研究では、適応の手続きの記述と、適応動作の記述を分離独立させる(図3.2-(b)) ことにより、上記の問題を回避する。また、適応の記述を容易かつ安全に拡張することができるようになる。

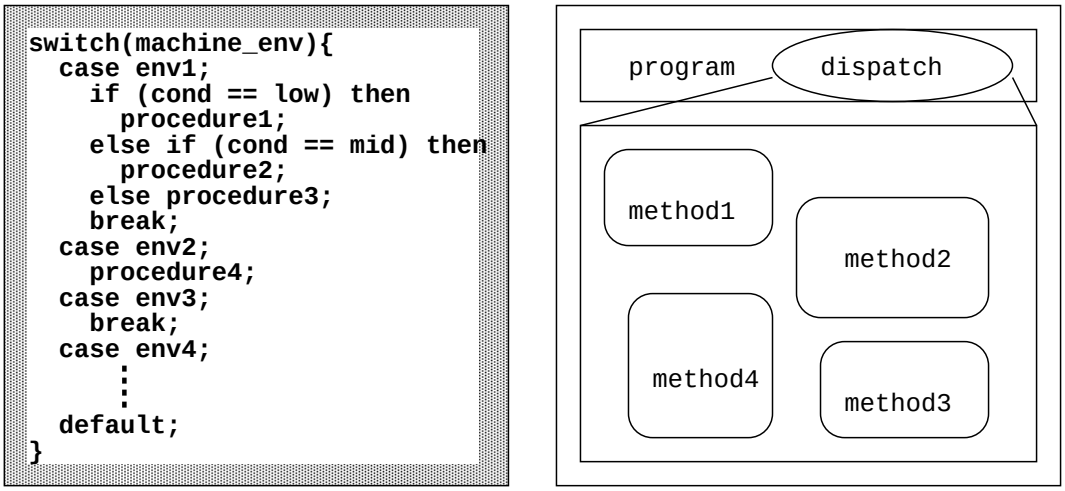

図 3.2: 記述の分離

3.3 環境の変化

計算機環境の変化に適応に付いて述べてきたが、ここで、「環境の変化」に注目してみる。環境の変化は、

- 常に変化する可能性のある環境
- マシン固有の環境

に大別できる。LEAD++は環境の変化に応じて、ドラスティックにディスパッチを行ない、現在の環境に最適なメソッドを呼び出すことで，どちらの環境の変化にも対応することができる。しかし「マシン固有の環境」の状況で、毎回ディスパッチを行なうことは無駄である。なぜなら、アプリケーションに実行中に環境の変化はあり得ないにもかかわらず、毎回「マシン固有の環境」に沿ったメソッドを選択し、実行するからである。

例えば，描画命令を多量に含む動的適用可能な移動コードを作成したいとする。そして環境によっては描画命令を高速に実行できるハードウェアが利用可能であるとする。ドラスティックな動的適応を用いて、対処するためには、次の2種類の方法が考えられる。

ディスパッチ経由で描画命令を実行するようにする方法 この方法は、適応の記述が完全に独立したモジュールであり、プログラムの可読性、拡張性、保守性に富んでいる。つまり、アプリケーションの本体や他の適応の手続き、適応メソッドに何等影響を

与えることなく、適応メソッドの修正、追加が可能である。このように、記述面において非常に優れているが、ドラスティックにディスパッチを行うため実行が遅くなってしまうという欠点がある。

環境毎に大きなメソッドを用意する方法 ハードウェア支援を用いないメソッド全体およびハードウェア支援を用いたメソッドを記述し、ディスパッチによってメソッドを起動し分ける方法。これは、ドラスティックな動的適応を用いながらも、最初の方法の「実行速度」の問題点を改善している。しかし、実行速度が速いという長所がある反面、冗長な記述を強いられるという短所がある。

本研究では、「マシン固有の環境」によって環境が静的に決定される場合において、記述性を確保しながら、実行速度の向上を行うための手法の提案を行う。そこで、実行前に静的に決まる環境を用いて、実行時にディスパッチを行い、現在の環境にもっとも適応したコードを生成し、プログラムを実行させることによって上記の問題に対応できる。実直な方法としては、実行時にソースコードから環境に適応したコードをコンパイルによって得るという方法が考えられる。しかし、動的なディスパッチのオーバーヘッドに比べて、コンパイルにかかる時間は非常に長く、コンパイルを行う利点が得られない。そこで、コンパイラに変わるものとして、動的コード生成という方法がある。動的コード生成はコンパイラに比べて高速にコード生成を行うことが出来るため、コード生成にかかるオーバーヘッドを大幅に減らすことができる。動的コード生成については次章で詳しく述べる。

第 4 章

動的コード生成

３章で、計算機環境に対する適応の必要性を述べた。４章では、３章で立てた仮定をもとに、移動コードに有効であると考えられる動的コード生成について述べる。

4.1 動的コード生成

動的コード生成とは、実行時にプログラムが変化する場合に、高速にその変化に応じたプログラムを生成する技術である。動的コード生成を用いることで効率的な実行が得られるが、その基本となる考えについて述べる。

動的コード生成に関して多くの研究があるが、ここでは代表的な中間コードを用いた方法、部分評価を用いたもの、テンプレートを用いたものを以下にあげ、その特徴についてまとめておく。

4.1.1 VCODE

VCODE[7] はリターゲットできる高速のコード生成システムである。中間コードとして理想的なRISC マシンの命令形式を取り、呼び出し系列やレジスタ区分など機能毎対象にあわせて変えられる拡張性を持っている。また、‘C[8] はlcc[9] を元に作られており、‘C はVCODE を生成する。基本的にVCODE は入力にC を用いる。VCODE の仕様は、lcc の中間コードの影響を強く受けている。‘C では、動的コード生成する部分を明示的に記述する。

VCODE のコード生成系は大域的最適化や命令スケジューリングを行わない。レジスタは呼び出しを越えて保存されるか否か（caller save, callee save）の２種類に分けて、種類別に優先度をつけ、一時変数に割り付ける。VCODE のレジスタは実マシンのレジスタと

ほぼ１対１に対応されることが多く、あふれレジスタに対応する変数はスタックに置かれる。これはVCODEからコード生成うる処理系にまかされている。Callee save, caller save, 引数レジスタの間でのレジスタ融通、末端手続き（リーフ）の特別扱いもできる。命令選択とバイナリコード生成の際にジャンプ先のバックパッチを行う。

コード生成は関数ごとに次のように行う。

- まずv_lambda関数を呼び、引数の型とそのレジスタ、末端か否かを指定し、生成するコードの入れ場所を与える。
- スタックにある引数はレジスタにロードし、prologの入れ場所を用意する。
- 利用プログラムで、レジスタと局所変数、ラベルを決定し、VCODEマクロを使ってコードを作って行く。
- prolog, epilog命令をバックパッチし、生成した命令列の番地を返す。命令生成時に簡単な遅延分岐処理も行う。

リターゲットするには、与えられたレジスタなどを変換して、対象機種の機械語を合成するマクロを作り、VCODE命令をそれらのマクロで表現し、呼び出し系列とアクティベーションレコード管理の部分を作る。そのためのプリプロセッサも作ってあり、MIPS, SPARC, Alphaに対しては４０～１００行で記述できる。機種間で流用できる部分も多く、x86との親和性も悪くなく、x86のレジスタの少なさはメモリオペランド命令でカバーできるので、仮想レジスタをメモリに対応させることができる。スタックマシンとの親和性は悪い。

4.1.2 FABIUS

FABIUS[10]はMLのサブセットで書かれたプログラムに対して、部分評価に基づいて動的コンパイルを行う。FABIUSは、MLもinteger, real, vectorとユーザ定義型を含む１階サブセットを入力する。このサブセットでは評価順序依存性がなく、別名解析も不要である。

行列の積の計算ルーチンの最内ループであるベクタの内積計算

```
fun dotprod (v1, v2) = loop (v1, v2, 0, length v1, 0)
and loop (v1, v2, i, n, sum) = if i = n then sum
   else loop (v1, v2, i+1, n, sum + (v1 sub i)  *
             (v2 sub i))
```

において、計数行列が定数であると、これは展開できて、(vi subi) の計算や添字範囲チェックも不要となる。さらに、疎行列だと (vi sub i) が 0 の部分の計算も不要になる。定数引数（early argument）に関数を作用させると、定数でない引数（lateargument）でパラメータ化された関数が得られる。これをレジスタトランスファー言語に変換し、early, late の区別を付記した MIPS アセンブリコードを生成する。末尾再帰はループに変換する。実行時には、early 命令はそのまま実行し、late 命令にオペランドの定数化や変位の定数化などを行って MIPS 命令を生成する。ループ制御部が early 命令だと、ループは完全に展開される。　生成する機械語を入れる場所はコードポインタ ($cp) で示す。これを使って、

```
add $sum,$sum,$prod     ここでadd, $sum, $prodはlate表示部
```

という late 命令は、

```
li    $t0,x852020   ; instruction encoding
sw    $t0,($cp)     ; write to code segment
addiu $cp,$cp,4     ; advance code pointer
```

というコード列に変換して実行するので、実行時には中間語表現を使っていない。1 命令生成するのに約 6 命令しかかからない。MIPS の即値は 16 ビットまでなので、32 ビット値をロードするには 2 サイクルかかるが、動的コンパイラでそれをメモリからロードしようとすると、少し複雑になる。変位が 16 ビット以内か否かなどで異なる命令列を生成した方が良いことがある。その場合は、どちらの命令列を生成するかを実行時に選択するが、中間語を使わないので、そのコストはわずかである。命令を生成してすぐ実行しようとすると、命令キャッシュとデータキャッシュが分かれていて、データとして命令を書き換えても、命令キャッシュが無効化されないマシンがある。DEC station5000/200 で命令キャッシュをフラッシュすると、トラップ時間＋0.8 nsec/B かかる。FABIUS では、既存コードの書き換えはしないので、コード領域をＧＣする時にのみフラッシュする。命令のプリフェッチが障害とならないように、生成する命令列は命令キャッシュラインに合わせるようにしている。インライン展開や末尾再帰のループ化の他に、インライン展開に適さない長い関数は、動的コンパイルした時にその引数の値を記録しておき、すでに出現した引数が再度与えられると、以前動的コンパイルしたコードを使う。実行時の最適化は難しい。ジャンプへのジャンプ省略や、命令スケジューリングを高速に行うのは困難である。コンパイル時に early 命令と late 命令で使うレジスタの生存区間を分離し、両者で同じレジスタを使っても干渉し合わないようにしている。しかし、プログラムによっては、生成するコードに無駄なスピル命令が入ることがある。

4.1.3 Auslander96

Auslander[11] の方法は、動的コンパイルの対象部分（動的領域）と実行時定数となる変数を注記したソースから、静的コンパイラで、空欄付き機械語テンプレートと、実行時定数算定用の準備コード、実行コード生成用の指示文を含む前コンパイラコードを生成する。動的コンパイラは(sticher)、指示文に従って、テンプレートに複写して空欄を埋める、入力にはCを用い、制限は無い。静的コンパイラでは、実行時定数の検出と、その到達範囲の解析を十分に行う。

プログラムに対する注記は以下のようなものがある。

dynamicRegion 関数本体などのブロックの先頭において、そのブロックが動的領域であることを示す。そこにきたときに、実行時定数となる変数も指定できる。

dynamic 実行時に値が不変となる変数を指定する。この変数は、ポイント先や添字付きであっても良く、ポインタが可変でもポイント先が不変なら不変と指示することができる。

unrolled 反復条件が実行時定数となり、ループ展開できるループを指定する。

1つの動的領域から複数の機械語列を生成しても良い。その対応関係はdynamicRegionのkeyとして指定した実行時定数となる変数の値で決まる。

静的コンパイラは、動的領域に対して次の4段階の処理を行う。

実行時定数となる変数の摘出 dynamicで指定すろ他に、実行時定数となる変数を求める。

```
x := y;
x := y op z;
x := f(y1, ..., yn);
x := *p;
```

上記の時、オペランドが実行時定数で、fが副作用無しなどいくつかの条件を満たす時、xも実行時定数となる。どの経路を通ってもいつも同じ値であるとか、条件式が実行時定数となり経路がかぎられてしまうことによって、いつも同じ値になる変数も実行時定数である。到達可能性解析は実行時定数解析と同時に行い、入り口から制御の流れに沿って進める。構造化されていないプログラムも対象とする。

準備コードとテンプレートの部分グラフに分割 準備コードは一度だけ実行され、実行時定数を全部計算し、表に記録する。その表は通常前もって割り付けるが、展開する数

を実行時に決定するループでは、ループの展開を行う時に割り付ける。テンプレートは、実行時定数の部分を空欄にした機械命令列である。

制御フローグラフの最適化 最適化処理は、準備コードとテンプレートコードに分離する前でも後でも行うことができる。テンプレートの穴は値の定まっていない定数として扱うが、「穴を含む命令はテンプレートの外には移動できない」、「穴の値は、動的領域の外では意味を持たない」、「ループ帰納変数に対応する穴の値は反復のたびに値が変わること」等に留意しなければならない。

機械語とsticher向けの指示文の生成 sticherへの指示文は、動的領域の開始、終了。展開されたループの開始、終了、帰辺、命令の空欄と分岐の空欄、PC相対分岐とその分岐先、を表現する。

対象言語を限定していないので、中間語は抽象構文木ではなく、3アドレス命令を用いている。

テンプレートを初期設定する時、sticherは指示文に従うだけでよい。その時、コードのコピーや分岐先オフセットの設定、実行時定数の命令へのパッチといった事を行う。大きな値や浮動小数点、ポインタ値に対してはロード命令を使い、必要であるとわかっているロード命令は静的コンパイラで生成し、スケジューリングしておく。sticherは定数の簡約などの簡単な覗き穴最適化を行う。sticherの処理を準備コードの中に織り交ぜると、更に処理を高速化できると思われるが、現時点では行っていない。しかし、現在の方法は処理に少し時間がかかる代わりに、単純で柔軟性がある。

4.2 まとめ

動的コード生成を用いた計算機環境に適応可能な処理系には、テンプレート方式を適応する。４章で述べた適応の方法を移動コードに用いたイメージは図4.1である。計算機環境に適応する同じような仕事を行う様々なプログラムを呼び出す場所を「穴」とみなし、その穴にあてはまるプログラムを「プログラム片」とすることで、テンプレートを用いた動的コード生成の手法が適用できる。

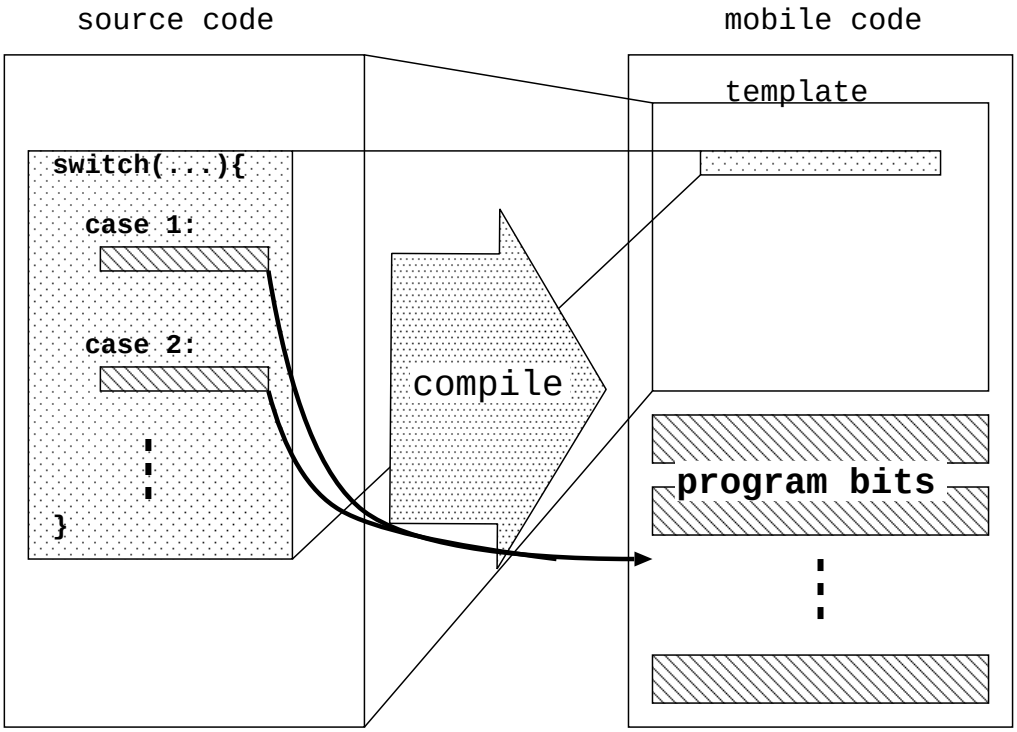

図 4.1: 移動コードの構造

第 5 章

処理系の構成

これまで、移動コードに関して適応の必要性と、適応の実現方法としての動的コード生成について述べてきた。5章では、これまで述べてきたことを実現するための具体的な実現方法について述べる。

5.1 適応的移動コードについて

移動コードに、移動先の計算機環境に適応する機構をもたせたものを、適応的移動コードと呼ぶ。

本論文では、適応的移動コードを実現するために次の点に注目した。

- 静的に決まる環境に適応したプログラムを、実行時に生成する
- 記述性を重視し、ディスパッチ部を分離する

上に述べた機構を実現する実行時コード生成を用いた効率のよい等価な命令への置換え、および不要コードの除去を行なえるデーモンをプロトタイプとして作成した。プロトタイプはJavaを用いて作成した。これは、JavaがRMI等の機構を持っており、移動コードに関するプロトタイプの作成に非常に適していることと、JavaのVMコードレベルでの操作は、そのままネイティブコード生成への転用が容易であるためである。

5.2 全体像

処理系の全体の流れは図5.1のようになる。本稿において移動コードは、実行先の様々な環境に応じた同じような動作を行う細かいプログラム片と、プログラム片を埋めこむ

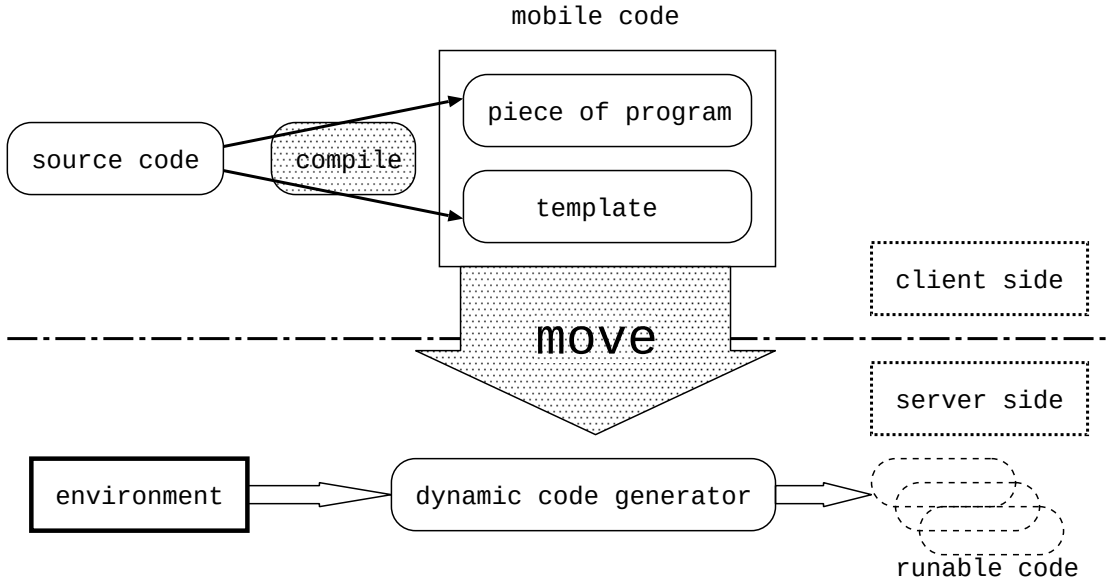

図 5.1: 全体の処理の流れ

テンプレートの２つから成る。これは、テンプレートを用いた方法によって動的コード生成を行うためである。この方法は、動的コード生成時に再コンパイルをする必要がないので、高速なコード生成が可能である。

このプログラム片と、テンプレートの２つから成る移動コードが、クライアントからサーバへ移動することになる。移動コードはサーバにある動的コード生成器へ送られる。動的コード生成器は、計算機の実行環境を参照し、移動コード中のプログラム片から適切なプログラム片を選び、テンプレートに空いた穴へコードを埋めこむ。このにようして、移動コードは異動先の実行環境に適応した実行可能コードとなる。

5.3 言語拡張

動的コード生成を行うプログラムの記述として、実行時コード生成する箇所を指定するものと、ユーザが意識しなくてもよいものとがあるが、本研究では明示的に指定する方法を用いた。これは、プログラムの動作予想が立てやすく、複雑な環境適用に対応することができる。

動的コード生成を行う部分の指定方法にはAuslander96や‘Cのように、言語の拡張を行う方法がある。本研究では処理系の作成にJavaを用いているため、Java本来の目的や、

Javaの性質を崩すことを極力避けておきたので、Javaの言語拡張、およびJavaの処理系には一切の手を加えない。言語拡張と、処理系に手を加えずに動的コード生成部分を指定するために以下の方法を用いる。

- 動的コード生成部分は必ずメソッド呼び出しで記述する
- 動的生成の対象となるメソッド郡は別クラスに記述する
- 動的コード生成を行うメソッドおよびクラスは、それが動的コード生成を行う部分であると判る特殊な名前付ける
- 動的コード生成を行わないメソッドおよびクラスは、動的コード生成を行うメソッドおよびクラスと同じ規則でメソッド名およびクラス名を付けてはならない

今回作成した処理系では条件、つまり取得できる環境を表??にあげる。動的コード部分を別クラスに分離することは、前述した、記述性、保守性、可読性の向上になる。動的コード生成部分の指定方法として、クラス名、およびメソッド名を用いる。指定方法を以下に示す。

_KS_[クラス識別子].[メソッド識別子]_[条件](引数1,引数2,…)

クラス名の先頭が_KS_で始まるクラスを動的コード生成の対象として、処理する。つまり、このクラスの中に含まれるメソッドはすべて動的コード生成時のプログラム片となる。メソッド名は「メソッド識別子」と「条件」からなる。メソッド識別子は、OSや、画面サイズに応じた複数鵜存在するメソッドに対して、同じ郡をなすものに同一の識別子を付ける。条件は、ディスパッチの条件を付ける。処理系はこの条件によって、環境に適応したメソッドを探すことになる。

[メソッド識別子]_[DCGが選ぶメソッド名](引数1,引数2,…)

処理系は、条件から得られる環境情報を用いて環境に適応したメソッドを選択する。選択されるメソッドは、表の「DCGが選ぶメソッド名」に合致するメソッドを選択する。

5.4 処理系の動作原理

実際の動作は例を挙げながら説明していく。まず、テンプレート部から見ていく。テンプレートには環境に適応した様々なコードを埋めるための「穴」が空いている。この穴

表 5.1: 取得できる計算機環境

条件	取得できるる環境	DCG が選ぶメソッド名
osarch_user_language	仕様言語	ja,en, etc.
osarch_java_version	Java のバージョン	1_1_6,1_1_5, etc.
osarch_line_separator	テキストの改行コード	CR,LF,CRLF
osarch_file_encoding	文字コードの種類	SJIS,EUCJIS,JIS
osarch_user_timezone	タイムゾーン	GMT,JST,ECT, etc.
osarch_user_name	ユーザー名	kawasaki, etc.
osarch_os_arch	CPU の種類	x86,SPARC, etc.
osarch_os_name	OS 名	Windows_95, SunOS etc.
osarch_os_version	OS のバージョン	4_10, 4_1_x, etc.
osarch_java_class_version	Java class のバージョン	45_3, etc.
MachineArc_ScreenWidth	画面の横幅	640,800,1024, etc.
MachineArc_ScreenHight	画面の高さ	400,600,768, etc.

は、メソッド呼び出しという形で実現している。「穴」は下のように指定する。処理系はこのメソッド呼び出しを「穴」と解釈し、環境に適応したメソッドを呼び出すようにコードを生成する。

```
_KS_scrnWidth.KS1_MachineArc_ScreenWidth();
```

クラス名が_KS_で始まっているのでこのメソッド呼び出しが、動的コード生成の対象であることがわかる。次に呼ばれているメソッドを見てみる。「KS1」という識別子をもっており、環境適応の条件は「MachineArc_ScreenWidth」で、表6.2から、これは画面の幅が条件であることがわかる。つぎに、コード片にあたる_KS_scrnWidth クラスを見てみる。

```
public class _KS_scrnWidth {
  public int width;

  void KS1_MachineArc_ScreenWidth(){}
  private void KS1_1024()   {width = 1024;}
  private void KS1_800()    {width = 800; }
  private void KS1_default(){width = -1 ; }

  public int getWidth(){return width;}
```

```
}
```

処理系はコード片のメソッド中にある識別子「KS1」を持つメソッド郡に対して処理を行う。識別子「KS1」を持つメソッド郡は

```
  void KS1_MachineArc_ScreenWidth(){}
  private void KS1_1024()   {width = 1024;}
  private void KS1_800()    {width = 800; }
  private void KS1_default(){width = -1 ; }
```

である。

次に、環境適応の条件であるMachineArc_ScreenWidthから得られる結果を元に、メソッドを選択する。実行環境の画面サイズが800 × 600であるとすると、MachineArc_ScreenWidthの条件から得られる結果は800であり、処理系は「KS_800」というメソッドを選択する。選ばれたメソッドは、MachineArc_ScreenWidthとすり替えられる。つまり、処理後_KS_scrnWidthクラスは下のようになる。

```
public class _KS_scrnWidth {
  public int width;

  void KS1_MachineArc_ScreenWidth(){width = 800; }

  public int getWidth(){return width;}
}
```

識別子KS1を持つメソッドは、選択されたメソッド以外全部削除される。さらに、プログラム片のクラスは名前を変えて、元からあるクラスはそのままに、別の新しいクラスを生成する。これは、オリジナルのクラスを書き換えてしまうと、プログラムの再利用性が損なわれてしまう為である。新しく作られるクラスは、元のクラス名の後ろに「__」をつける。これに伴って、メソッド呼び出し部、テンプレートの「穴」にあたる部分もクラス名の書換を行う。

5.5 ディスパッチの仕組

ディスパッチの動作自体はリンカとさして違いは無い。ここでは、JavaVMコードにおいて、どのようにクラス名の差し替えを行っているかについて説明する。差し替えを行うにあたって、まずJavaのクラスファイルの構造を知っておく必要が有る。Javaのクラスファイルは大まかに図5.2のような構造になっている。図5.2を見てもらうと判るように、

Javaのclassはメソッド毎に管理されている。また、class名やメソッド名はすべてコンスタントプールエリアで管理されている。そこで、今回作成した処理系では、バイトコード自体には全く手を触れず、コンスタントプールエリアに格納されているメソッド名と、メソッド名とメソッド本体とのリンクだけを操作する。Javaには、公式のアセンブラコードは存在しない。そこで、便宜上JasminというJavaのアセンブラコード表記を用いる。

JavaVMにおいて、メソッド呼び出しはすべてinvoke命令によって行われる。invoke命令には、invokeinterface、invokespecial、invokestatic、invokevirtulaがあり、何れも引数としてCONSTANT_Methodrefをとる。このCONSTANT_Methodrefはコンスタントプールのエントリ番号をさしており、このエントリ番号から呼びだされるメソッドを特定できる。例として、付録２に挙げたプログラムと、バイトコードと、その解析結果を用いる。詳しくは付録２を参照されたい。コンスタントプールエントリに関する参照関係を図5.3に示す。Javaのコンスタントプールエントリはこのような構造になっている。つまり、コンスタントプールエントリを書き換えるだけで、目的の処理系を実装することができる。

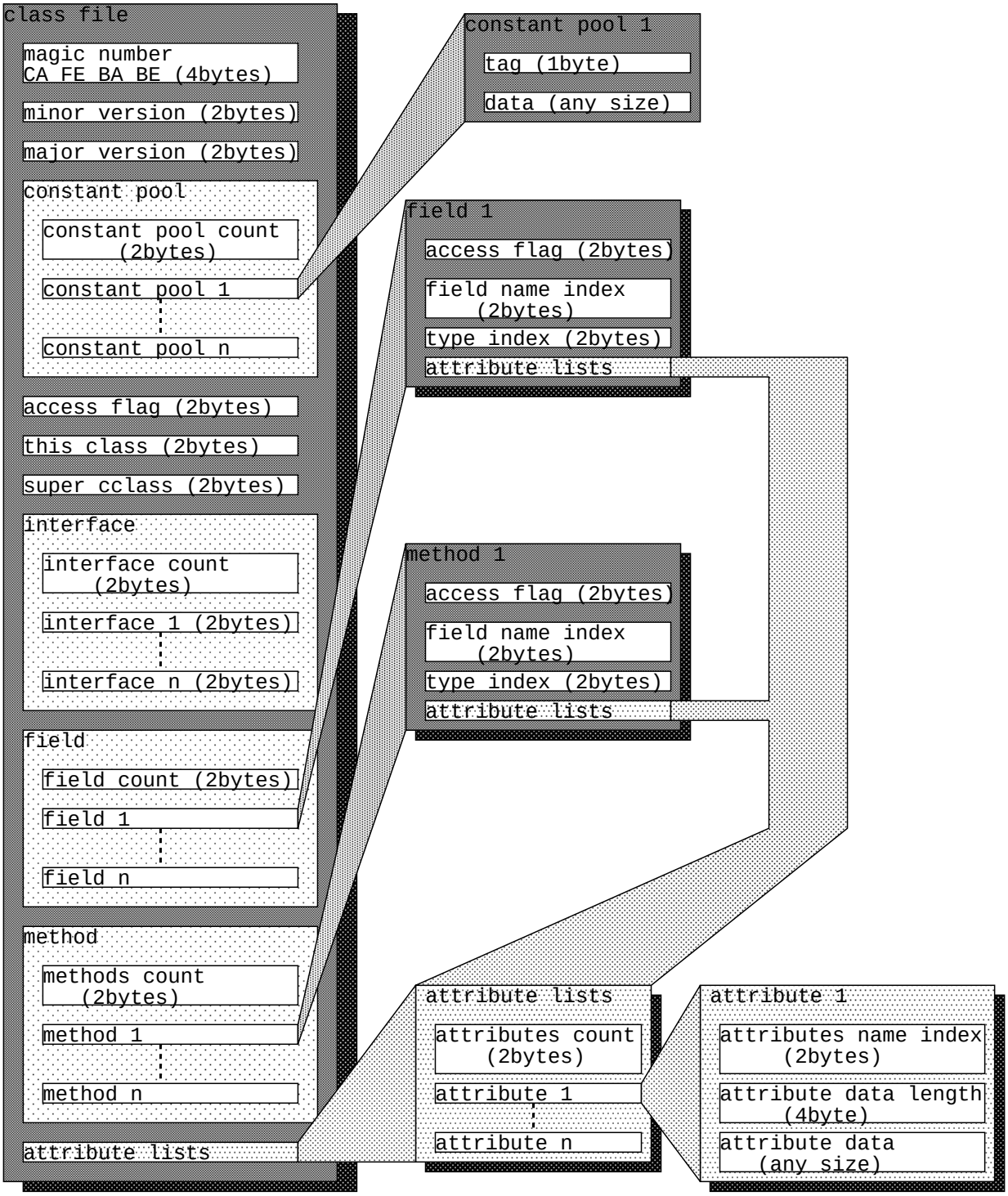

図 5.2: class の構造

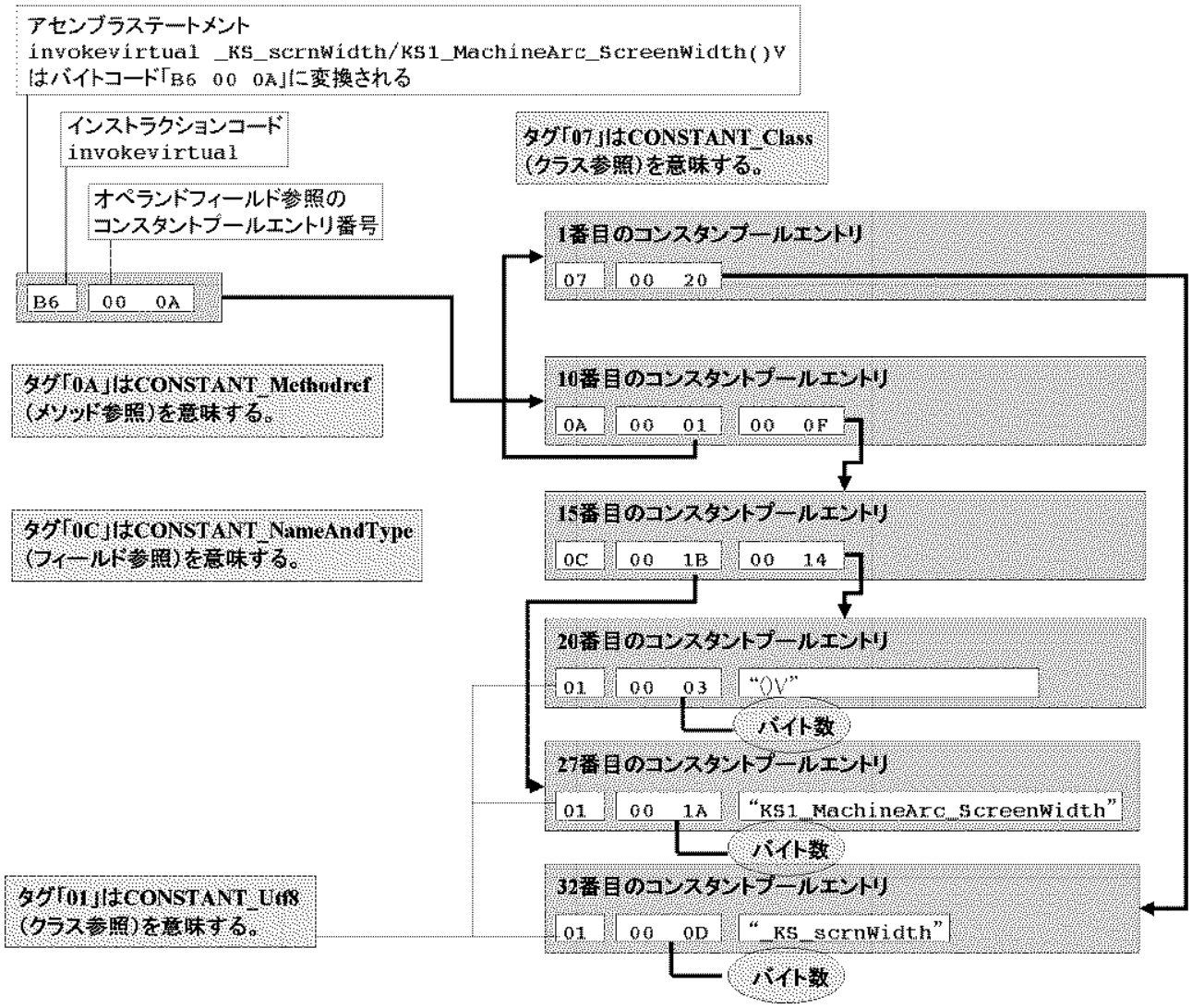

図 5.3: KS1_MachineArc_ScreenWidth が参照するコンスタントプール

第 6 章

結果

6.1 実験

本研究では、静的に決定される情報を元に、実行環境に適応したコードを、実行時に生成することで、効率の良い等価な命令への置き換え、および不要コードの除去を行える処理系を、プロトタイプとして作成した。これによってどの程度、効率の改善ができたのかをベンチマークにより評価を行う。

評価にはSPARC アーキテクチャを使用したものと、x86 アーキテクチャを使用したものの2 通りの環境を用いた。使用した環境は以下の表6.1である。

コンパイラはJDK1.1.6 を使用している。評価には実行時間を用いる。実行時間の計測にはJava の関数Date を用いた。

今回実験を行ったベンチマークは以下の通りである。

- ディスパッチを伴う関数呼び出しにかかる時間
- ディスパッチ動作を削除した時の関数呼び出しにかかる時間
- 作成した処理系の実行時間

これらの計測結果を図6.1, 図6.2, 表6.2に示す。実験に用いたプログラムのリストは付録に載せた。

6.2 考察

結果から述べると、ディスパッチのオーバーヘッドがなくなっている分だけ、確実に速くなっている。1 つづつ見ていく。図6.1はPC/AT 互換機の結果であるが、ループ回数が

表 6.1: 評価に使用した環境

マシン名	Sun Ultra Enterprise 3000
CPU	UltraSPARC-II
クロック	250MHz
メモリ	512MB
OS	SunOS 5.6
マシン名	PC/AT 互換機
CPU	Celeron
クロック	333MHz
メモリ	256MB
OS	Windows 98

増える毎に、ほぼ線形変化であることが判る。この場合、ディスパッチを行う場合と、行わない場合とで約１２倍速くなっていることがわかる。

図6.2のUltra Enterprise3000においては、４１倍の速度改善が見られる。この事から、ディスパッチには大きなオーバーヘッドが伴うことを示していると考えられる。逆にディスパッチを削除することで良好な速度改善が得られる。図6.1と図6.2の間には速度改善率にかなりの開きがある。これは、Ultra Enterprize 3000で採用されているSPARCアーキテクチャのパイプライン処理は条件分岐命令を多用するとSPARC本来の性能が引き出せないためであると考えられる。つまり、本研究のアプローチはSPARCのようなプロセッサには非常に有効であると考えられる。

次に表6.2についてみてみる。「環境の取得」は、計算機環境（OS、CPU、画面サイズ等）を取得するのにかかる時間である。PC/AT互換機に比べて、Ultra Enterprise 3000やSSparc tation 5は非常に時間がかかっている。環境の取得は、処理系の動作速度との対比のためにのせてある。PC/AT互換機ではコード生成の処理にやや時間がかかっているが、EnterpriseやStation5では十分に速い結果が得られた。

表と図から、PC/AT互換機では約18000回以上ディスパッチを減らすことで、動的コード生成にかかる時間よりも実行速度の向上が期待される。Ultra Enterprise 3000では、約1200回以上ディスパッチを減らせば、動的コード生成にかかる時間よりも実行速度の向上が期待できる。

今回実験に用いたプログラムは、メソッド呼び出しをループしているだけの非常に簡単

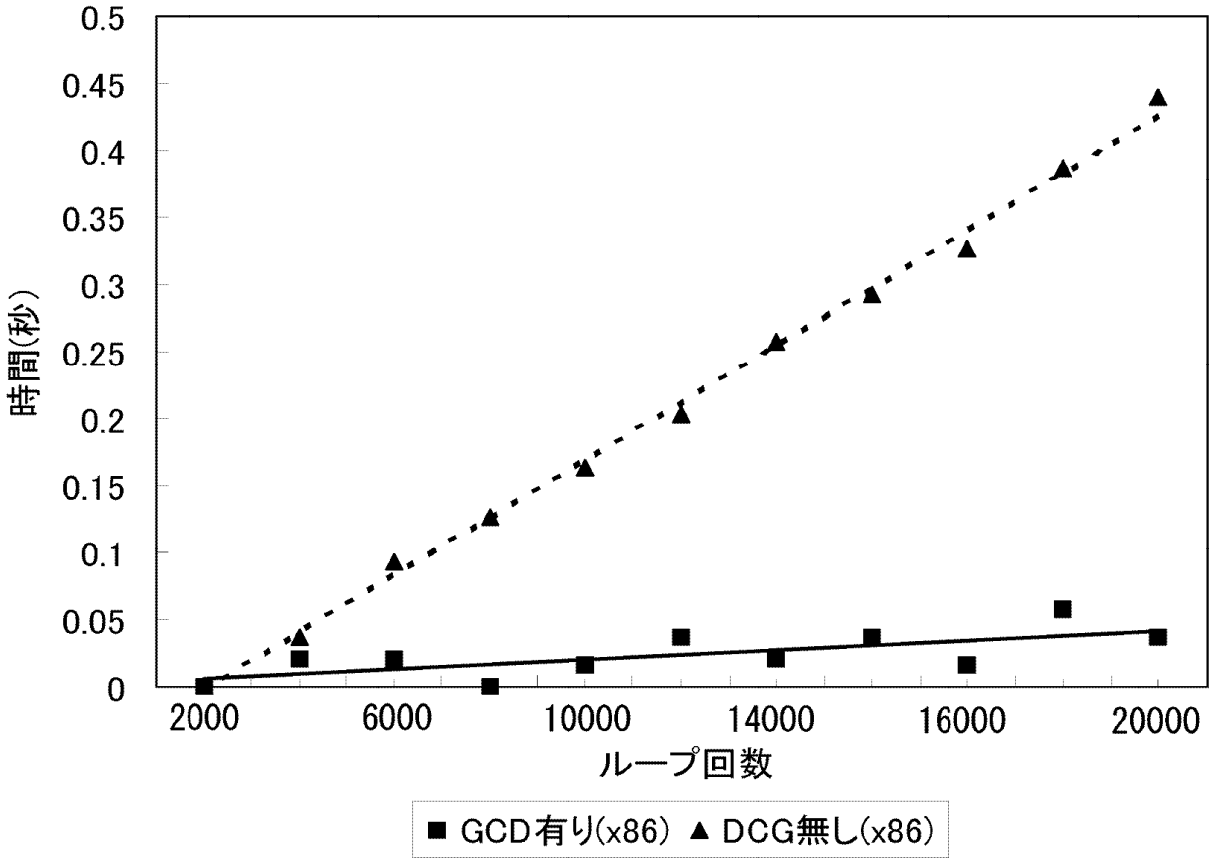

図 6.1: PC/AT 互換機

なものであったが、Ultra Enterprise 3000 では非常に良好な結果が得られた。

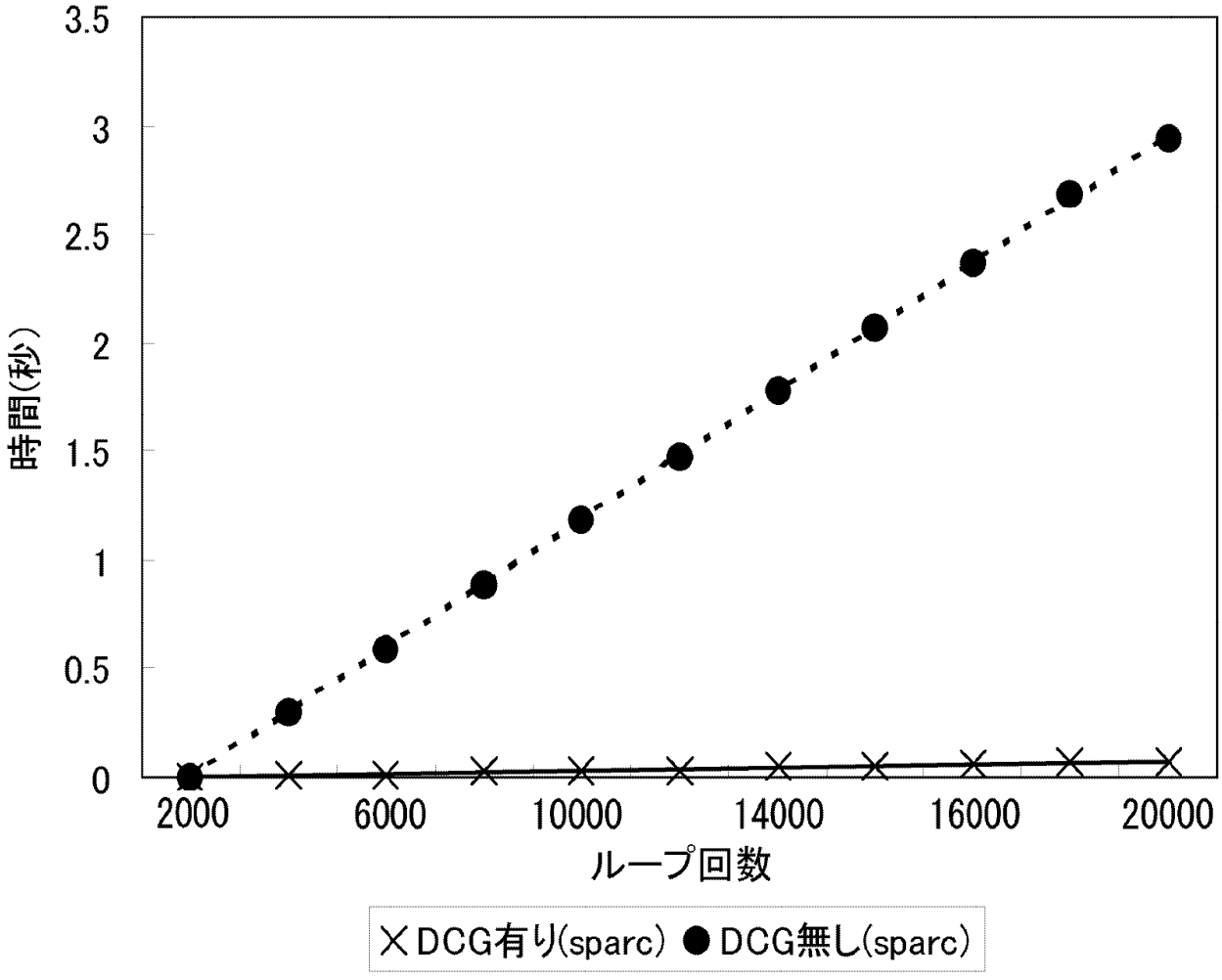

図 6.2: Ultra Enterprise 3000

表 6.2: 計測結果

	PC/AT 互換機	Enterprise 3000	Sparc Station 5
環境の取得	0.35	1.934	3.872
テンプレートの解析 ファイルの読み込み	0.05	0.006	0.038
テンプレートの解析 処理	0.17	0.034	0.058
テンプレートの解析 ファイルの書き出し	0.05	0.059	0.049
ディスパッチ プログラム片の読み込み	計測不能	0.003	0.010
ディスパッチ 処理	0.11	0.067	0.079

第 7 章

まとめ

7.1 まとめ

本稿では、現在非常に注目されている移動コードの技術に、環境適応の必要性を述べ、適応的移動コード構成の方針について提案をした。また、プロトタイプとして、簡単な処理系を構成した。

提案した方針は、以下の通りである。

- 環境の変化が静的なものに対して、実行時に最適なコードを生成する
- 動的適応部を分離することによる、記述性の向上
- 動的コード生成による高速なコード生成

提案に従い、簡単な処理系を構成し、いくつかのサンプルプログラムを書き、「記述性を確保しながら、処理速度を向上できる」ことを確認した。

7.2 今後の課題

本研究で実装した処理系はあくまで実験の為プロトタイプである。今後更なる改良のために以下の課題が挙げられる。

ネイティブコードへの対応 現在の実装では、JavaVM コードを用いて、環境に適応した JavaVM コードを生成している。承知のとおり Java は VM もしくは、JIT によって実行されているために、実行速度の面において劣っている。動的コード生成は直接

ネイティブコードを生成する技術も既に幾つか研究されており、この技術を適用することで更に速度の向上が期待できる。

適応の戦略 環境に適応したメソッドを選択する戦略について、あまり考慮していない。柔軟かつ、高い動的適応可能性を行うためには、適応の戦略に工夫を行う必要が有る。

動的適応を行う場所の記述 本研究では、クラス名と、メソッド名に意味を持たせることにより動的コード生成を行っている。しかし、この方法は、プログラマが不意に動的コード生成の対象となるクラス名や、メソッド名を用いてしまう可能性があるが、そのような場合の対処機構は備えていない。この方法では、不意に問題の記述を行ったときに、動作保証が得られないために、プログラマの混乱を招く恐れがある。プリプロセッサを用いるなどの対処が必要である。

更なる効率改善 今回作成した処理系は、不要コードの除去と、動的なリンカとしての機能しか持ち合わせていない。動的コード生成には、部分評価などの最適化の技法が数多く用いられている。これらの機構を導入することにより更なる速度向上が期待できる。

QoS への転用 QoS[12] はシステム構成、環境に応じて動的にデータ等の質を調整し、サービス全体としての質を保証することを意味する。例えば、リアルタイムに動画の転送を行なうとき、ネットワーク資源の確保を行ない、サービス全体としての環境が決定する。このときネットワーク資源が確保されていることを仮定して、ほん研究を適用することができる。

謝辞

本研究をおこなうにあたり，終始ご指導していただいた情報科学研究科情報システム学専攻言語設計学講座 助教授渡部 卓雄博士に深く感謝いたします。また、有益な助言をしていただいた講座教授 二木 厚吉博士に御礼を申し上げます。最後に、研究に関する議論につきあっていただいた言語設計額講座の諸氏に感謝いたします。

参考文献

[1] James E. White, Telescript Technology-Mobile Agents, American Association for Artificial Intelligency, 1991.

[2] 佐藤一郎, AgentSpace 高階モーバイルエージェントシステム, 電子情報通信学会技術研究報告,Vol97,No627,pp 41-48, 電子情報通信学会, 1998.

[3] Java,http://java.sun.com

[4] Toshikazu Akagi, Tatsuo Nakajima, A Framework for Building Adaptive Applications in Mobile Computing Environments, IPSJ MBL, November , 1996.

[5] Noriki Amano, Takuo Watanabe, LEAD++: An Object-Oriented Reflective Language for Dynamically Adaptable Software, OOPSLA'98 International Workshop on Reflective Programming in C++ and Java, Technical Report of Center for Computational Physics, University of Tsukuba 98-4, pp.91-95, Oct, 1998

[6] Akihiro Hokimoto, Tatsuo Nakajima, An Approach for Constructing Dynamic Adaptive Software in Heterogeneous Computing Environments, DiCoMo Workshop, July, 1997.

[7] Dawson R. Engler, VCODE:A Retargetable, Extensible, Very Fast Dynamic Code Generation System,Proc. ACM PLDI '96 pp. 160-170,1996

[8] DawsonR.Engler. et al, 'C:A Language for Hight-Level,Efficient,and Machine-Independent Dynamic Code Generation,Proc. POPL '96,pp 1-14,1996

[9] C.W.Fraser and D.R. Hanson., A retargetable compiler for ANSI C. SIGPLAN Notices,26(10), October 1991.

[10] Lee, P., Leon, M. (CMU), Optimizing ML with Run-Time Code Generation, Proc. PLDI'96, pp. 137-148, May, 1996.

[11] Auslander, J., Philipose, M., Chambers, C., Eggera, S. J., Bershad, N., Fast, Effective Dynamic Compilation, Proc. PLDI '96, pp. 149-159, May, 1996.

[12] 栗原 邦彰, 中島 達夫, サービスプロキシを用いた移動計算機環境の構築法, 第１１回ソフトウェア科学会全国大会, 1994.

付録１

実験で用いたプログラムのリストを載せる。

テンプレート template.java

```
import java.io.*;
import java.util.*;

public class template {
  public static void main(String[] args){

    getEnvironment ge = new getEnvironment();

    for(int i1=0; i1 <= 50; i1++){
      System.out.print(i1*2000+" loop ");
      for(int i2=0; i2 < 3; i2++){
        Date d1 = new Date();
        for(int i=0; i< i1*2000; i++){
          _KS_scrnWidth width = new _KS_scrnWidth();
          width.KS1_MachineArc_ScreenWidth();
        }
        Date d2 = new Date();
        long dl2 = d2.getTime() - d1.getTime();
        System.out.print(" "+((double)dl2/1000)+", ");
      }
      System.out.println("");
    }
  }
}
```

プログラム片 _KS_scrnWidth.java

```
public class _KS_scrnWidth {
  public int width;

  public _KS_scrnWidth() {
    KS1_MachineArc_ScreenWidth();
  }

  void KS1_MachineArc_ScreenWidth(){
    if(getEnvironment.findCond("MachineArc_ScreenWidth")){
      if(getEnvironment.findEnv("1600")){KS1_1600();}
      if(getEnvironment.findEnv("1152")){KS1_1152();}
      if(getEnvironment.findEnv("1024")){KS1_1024();}
      if(getEnvironment.findEnv("800")){KS1_800();}
    } else {KS1_default();}
  }
  private void KS1_1600()   {width = 1600;}
  private void KS1_1152()   {width = 1152;}
  private void KS1_1024()   {width = 1024;}
  private void KS1_800()    {width = 800; }
  private void KS1_default(){width = -1 ; }

  public int getWidth(){return width;}
}
```

付録２

ここでは、処理系の実装において例題として用いたプログラムの、「ソースコード」、「クラスファイルの一六進ダンプ」、「逆アセンブル」、「クラスファイルの構造解析」をのせておく。

ソースコード test3.java

```
public class test3 {
  public static void main(String[] args){
    getEnvironment ge = new getEnvironment();
    _KS_scrnWidth width = new _KS_scrnWidth();
    width.KS1_MachineArc_ScreenWidth();
    System.out.println(width.getWidth());
  }
}
```

test3.class の一六進ダンプ test3.class

```
0000  CA FE BA BE 00 03 00 2D-00 2B 07 00 20 07 00 21   .......-.+.. ..!
0010  07 00 23 07 00 24 07 00-25 07 00 29 0A 00 01 00   ..#..$..%..)....
0020  0E 0A 00 02 00 0E 0A 00-04 00 0E 0A 00 01 00 0F   ................
0030  0A 00 01 00 10 09 00 05-00 11 0A 00 03 00 12 0C   ................
0040  00 17 00 14 0C 00 1B 00-14 0C 00 22 00 13 0C 00   ...........".....
0050  27 00 1D 0C 00 28 00 15-01 00 03 28 29 49 01 00   '....(.....()I..
0060  03 28 29 56 01 00 04 28-49 29 56 01 00 16 28 5B   .()V...(I)V...([
0070  4C 6A 61 76 61 2F 6C 61-6E 67 2F 53 74 72 69 6E   Ljava/lang/Strin
0080  67 3B 29 56 01 00 06 3C-69 6E 69 74 3E 01 00 04   g;)V...<init>...
0090  43 6F 64 65 01 00 0D 43-6F 6E 73 74 61 6E 74 56   Code...ConstantV
00A0  61 6C 75 65 01 00 0A 45-78 63 65 70 74 69 6F 6E   alue...Exception
00B0  73 01 00 1A 4B 53 31 5F-4D 61 63 68 69 6E 65 41   s...KS1_MachineA
00C0  72 63 5F 53 63 72 65 65-6E 57 69 64 74 68 01 00   rc_ScreenWidth..
00D0  0F 4C 69 6E 65 4E 75 6D-62 65 72 54 61 62 6C 65   .LineNumberTable
00E0  01 00 15 4C 6A 61 76 61-2F 69 6F 2F 50 72 69 6E   ...Ljava/io/Prin
```

```
00F0  74 53 74 72 65 61 6D 3B-01 00 0E 4C 6F 63 61 6C   tStream;...Local
0100  56 61 72 69 61 62 6C 65-73 01 00 0A 53 6F 75 72   Variables...Sour
0110  63 65 46 69 6C 65 01 00-0D 5F 4B 53 5F 73 63 72   ceFile..._KS_scr
0120  6E 57 69 64 74 68 01 00-0E 67 65 74 45 6E 76 69   nWidth...getEnvi
0130  72 6F 6E 6D 65 6E 74 01-00 08 67 65 74 57 69 64   ronment...getWid
0140  74 68 01 00 13 6A 61 76-61 2F 69 6F 2F 50 72 69   th...java/io/Pri
0150  6E 74 53 74 72 65 61 6D-01 00 10 6A 61 76 61 2F   ntStream...java/
0160  6C 61 6E 67 2F 4F 62 6A-65 63 74 01 00 10 6A 61   lang/Object...ja
0170  76 61 2F 6C 61 6E 67 2F-53 79 73 74 65 6D 01 00   va/lang/System..
0180  04 6D 61 69 6E 01 00 03-6F 75 74 01 00 07 70 72   .main...out...pr
0190  69 6E 74 6C 6E 01 00 05-74 65 73 74 33 01 00 0A   intln...test3...
01A0  74 65 73 74 33 2E 6A 61-76 61 00 21 00 06 00 04   test3.java.!....
01B0  00 00 00 00 00 02 00 09-00 26 00 16 00 01 00 18   .........&......
01C0  00 00 00 45 00 02 00 02-00 00 00 1D BB 00 02 B7   ...E............
01D0  00 08 BB 00 01 59 B7 00-07 4C 2B B6 00 0A B2 00   .....Y...L+.....
01E0  0C 2B B6 00 0B B6 00 0D-B1 00 00 00 01 00 1C 00   .+..............
01F0  00 00 16 00 05 00 00 00-06 00 06 00 07 00 0E 00   ................
0200  08 00 12 00 09 00 1C 00-05 00 01 00 17 00 14 00   ................
0210  01 00 18 00 00 00 1D 00-01 00 01 00 00 00 05 2A   ...............*
0220  B7 00 09 B1 00 00 00 01-00 1C 00 00 00 06 00 01   ................
0230  00 00 00 04 00 01 00 1F-00 00 00 02 00 2A         .............*
```

逆アセンブル 逆アセンブルツールD-Javaを用いてtest3.classを逆アセンブルした結果をのせる。JavaVMアセンブラ言語のうちの1つであるJasmin形式で出力した。

```
;>> test3.class <<
;
; Output created by D-Java (May 12 1997)
; mailto:umsilve1@cc.umanitoba.ca
; Copyright (c) 1996-1997 Shawn Silverman
;

;Classfile version:
;    Major: 45
;    Minor: 3

.source test3.java
.class  public test3
; ACC_SUPER bit is set
.super  java/lang/Object
```

```
; >> METHOD 1 <<
.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 2
.line 6
    new getEnvironment
    invokespecial getEnvironment/<init>()V
.line 7
    new _KS_scrnWidth
    dup
    invokespecial _KS_scrnWidth/<init>()V
    astore_1
.line 8
    aload_1
    invokevirtual _KS_scrnWidth/KS1_MachineArc_ScreenWidth()V
.line 9
    getstatic java/lang/System/out Ljava/io/PrintStream;
    aload_1
    invokevirtual _KS_scrnWidth/getWidth()I
    invokevirtual java/io/PrintStream/println(I)V
.line 5
    return
.end method

; >> METHOD 2 <<
.method public <init>()V
    .limit stack 1
    .limit locals 1
.line 4
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method
```

test3.class の構造 test3.class は一六進ダンプを見ても判るように、合計574bytes のバイト列で構成されている。このバイト列の構造をJavaVM の仕様に従ってバイト列に切れ目と簡単なコメントを入れたものを以下に示す。

```
Class file length=574bytes
```

```
// マジックナンバー
CA FE BA BE

// マイナーバージョン
00 03

// メジャーバージョン
00 2D

// ここからコンスタントプールエントリ

00 2B  // コンスタントプールエントリ数[42]

    // エントリ 1:Class参照 name_index(32)
    07 00 20

    // エントリ 2:Class参照 name_index(33)
    07 00 21

    // エントリ 3:Class参照 name_index(35)
    07 00 23

    // エントリ 4:Class参照 name_index(36)
    07 00 24

    // エントリ 5:Class参照 name_index(37)
    07 00 25

    // エントリ 6:Class参照 name_index(41)
    07 00 29

    // エントリ 7:Method参照 name_index(1) descriptor_index(14)
    0A 00 01 00 0E

    // エントリ 8:Method参照 name_index(2) descriptor_index(14)
    0A 00 02 00 0E

    // エントリ 9:Method参照 name_index(4) descriptor_index(14)
    0A 00 04 00 0E

    // エントリ 10:Method参照 name_index(1) descriptor_index(15)
    0A 00 01 00 0F
```

```
// エントリ 11:Method参照 name_index(1) descriptor_index(16)
0A 00 01 00 10

// エントリ 12:Field参照 name_index(5) descriptor_index(17)
09 00 05 00 11

// エントリ 13:Method参照 name_index(3) descriptor_index(18)
0A 00 03 00 12

// エントリ 14:名前と型 name_index(23) descriptor_index(20)
0C 00 17 00 14

// エントリ 15:名前と型 name_index(27) descriptor_index(20)
0C 00 1B 00 14

// エントリ 16:名前と型 name_index(34) descriptor_index(19)
0C 00 22 00 13

// エントリ 17:名前と型 name_index(39) descriptor_index(29)
0C 00 27 00 1D

// エントリ 18:名前と型 name_index(40) descriptor_index(21)
0C 00 28 00 15

// エントリ 19:Utf8 ("()I")
01 00 03 28 29 49

// エントリ 20:Utf8 ("()V")
01 00 03 28 29 56

// エントリ 21:Utf8 ("(I)V")
01 00 04 28 49 29 56

// エントリ 22:Utf8 ("([Ljava/lang/String;)V")
01 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 29 56

// エントリ 23:Utf8 ("<init>")
01 00 06 3C 69 6E 69 74 3E

// エントリ 24:Utf8 ("Code")
01 00 04 43 6F 64 65

// エントリ 25:Utf8 ("ConstantValue")
01 00 0D 43 6F 6E 73 74 61 6E 74 56 61 6C 75 65
```

```
// エントリ 26:Utf8 ("Exceptions")
01 00 0A 45 78 63 65 70 74 69 6F 6E 73

// エントリ 27:Utf8 ("KS1_MachineArc_ScreenWidth")
01 00 1A 4B 53 31 5F 4D 61 63 68 69 6E 65 41 72 63 5F 53 63 72 65 65 6E 57 69 64 74 68

// エントリ 28:Utf8 ("LineNumberTable")
01 00 0F 4C 69 6E 65 4E 75 6D 62 65 72 54 61 62 6C 65

// エントリ 29:Utf8 ("Ljava/io/PrintStream;")
01 00 15 4C 6A 61 76 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 65 61 6D 3B

// エントリ 30:Utf8 ("LocalVariables")
01 00 0E 4C 6F 63 61 6C 56 61 72 69 61 62 6C 65 73

// エントリ 31:Utf8 ("SourceFile")
01 00 0A 53 6F 75 72 63 65 46 69 6C 65

// エントリ 32:Utf8 ("_KS_scrnWidth")
01 00 0D 5F 4B 53 5F 73 63 72 6E 57 69 64 74 68

// エントリ 33:Utf8 ("getEnvironment")
01 00 0E 67 65 74 45 6E 76 69 72 6F 6E 6D 65 6E 74

// エントリ 34:Utf8 ("getWidth")
01 00 08 67 65 74 57 69 64 74 68

// エントリ 35:Utf8 ("java/io/PrintStream")
    01 00 13 6A 61 76 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 65 61 6D

// エントリ 36:Utf8 ("java/lang/Object")
01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74

// エントリ 37:Utf8 ("java/lang/System")
01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 53 79 73 74 65 6D

// エントリ 38:Utf8 ("main")
01 00 04 6D 61 69 6E

// エントリ 39:Utf8 ("out")
01 00 03 6F 75 74

// エントリ 40:Utf8 ("println")
```

```
    01 00 07 70 72 69 6E 74 6C 6E

    // エントリ 41:Utf8 ("test3")
    01 00 05 74 65 73 74 33

    // エントリ 42:Utf8 ("test3.java")
    01 00 0A 74 65 73 74 33 2E 6A 61 76 61

// コンスタントプールエントリはここまで

// クラスのアクセスフラグ（このクラスはpublic）
00 21

// this class のコンスタントプールエントリ番号 name_index(6)
00 06

// super class のコンスタントプールエントリ番号 name_index(4)
00 04

// このクラスで宣言されているインターフェースの個数（0個）
00 00
    // インターフェースの個数が0なので、
    // インターフェース配列のバイト列は現れない

// フィールドの個数（0個）
00 00
    // このクラスには、フィールドは1つも宣言されていないので、
    // フィールドの配列は現れない

// このクラスで宣言されているメソッドの個数（2個）
00 02
    // メソッドの始まり
    // main() メソッドの始まり
    // main() メソッドのアクセスフラグ(public static)
    00 09

    // メッソド名("main")の
    // コンスタントプールエントリ番号 name_index(38)
    00 26

    // メソッドの引数と返り値の型("([Ljava/lang/String;)V")の
    // コンスタントプールエントリ番号（22）
    //（引数がjava.lang.Stringの1次配列で、返り値がvoid）
    00 16
```

```
// メソッド属性の個数（1 個）
00 01

    // Code 属性の始まり
    //属性の名前（"Code"）
    // のコンスタントプールエントリ番号 name_index(24)
    00 18

    // 属性データのバイト長（69bytes）
    00 00 00 45

    // インストラクションコード列の始まり
        // オペランドスタックの上限値（2 個）
        00 02

        // ローカル変数の上限値（2 個）
        00 02

        // インストラクションコードのバイト数（29bytes）
        00 00 00 1D

        // new getEnvironment
        BB 00 02

        // invokespecial getEnvironment/<init>()V
        B7 00 08

        // new _KS_scrnWidth
        BB 00 01

        // dup
        59

        // invokespecial _KS_scrnWidth/<init>()V
        B7 00 07

        // astore_1
        4C

        // aload_1
        2B

        // invokevirtual _KS_scrnWidth/KS1_MachineArc_ScreenWidth()V
```

```
            B6 00 0A

            // getstatic java/lang/System/out Ljava/io/PrintStream;
            B2 00 0C

            // aload_1
            2B

            // invokevirtual _KS_scrnWidth/getWidth()I
            B6 00 0B

            // invokevirtual java/io/PrintStream/println(I)V
            B6 00 0D

            // return
            B1
        // インストラクションコードの終わり

        // 例外ハンドラの個数（0個）
        00 00
          // 1例外ハンドラが1つも無いので、
          // 例外ハンドラ配列は現れない

            // メソッドの属性であるCode属性の属性配列の始まり
            // 属性の個数（1個）
            00 01

                // 行番号属性の始まり
                // ここにはソースコードの行番号情報が格納されている
                // 本質ではないので一六進のみの表記
                00 1C 00 00 00 16 00 05 00 00 00 06 00 06 00 07
                00 0E 00 08 00 12 00 09 00 1C 00 05
                // 行番号属性の終わり
            // メソッドの属性であるCode属性の属性配列の終わり
        // Code属性の終わり
    // main()メソッドの終わり

//////////////////////////////
    // <init>() メソッドの始まり
    // <init>() メソッドのアクセスフラグ(public)
    00 01

    // メッソド名("<init>")の
    // コンスタントプールエントリ番号 name_index(23)
```

```
00 17

// メソッドの引数と返り値の型("()V")の
// コンスタントプールエントリ番号 name_index(20)
//(引数が無く、返り値がvoid)
00 14

// メソッド属性の個数(1個)
00 01

    // Code 属性の始まり
    //属性の名前("Code")の
    // コンスタントプールエントリ番号 name_index(24)
    00 18

    // 属性データのバイト長(29bytes)
    00 00 00 1D

    // インストラクションコード列の始まり
        // オペランドスタックの上限値(1個)
        00 01

        // ローカル変数の上限値(1個)
        00 01

        // インストラクションコードのバイト数(5bytes)
        00 00 00 05

        // aload_0
        2A

        // invokespecial java/lang/Object/<init>()V
        B7 00 09

        // return
        B1
    // インストラクションコードの終わり

    // 例外ハンドラの個数(0個)
    00 00
      // 1例外ハンドラが1つも無いので、
      // 例外ハンドラ配列は現れない

        // メソッドの属性であるCode属性の属性配列の始まり
```

```
            // 属性の個数（1 個）
            00 01

                // 行番号属性の始まり
                // ここにはソースコードの行番号情報が格納されている
                // 本質ではないので一六進のみの表記
                00 1C 00 00 00 06 00 01 00 00 00 04
                // 行番号属性の終わり
            // メソッドの属性である Code 属性の属性配列の終わり
        // Code 属性の終わり
    // <init>() メソッドの終わり

// クラス属性の個数（1 個）
00 01
    // クラス属性の始まり
    // SourceFile 属性の始まり
    //属性の名前("SourceFile") の
    // コンスタントプールエントリ番号 name_index(31)
    00 1F

    // 属性のバイト長(2bytes)
    00 00 00 02

    // ソースファイル名("test3.java") の
    // コンスタントプールエントリ番号 name_index(42)
    00 2A
    // SourceFile 属性の終わり
    // クラスの属性配列の終わり
```