Vol. 43 No. SIG 8(PRO 15) 情報処理学会論文誌：プログラミング Sep. 2002

家電向け Java JIT コンパイラの構成方法とその評価

川 本 琢 二† 春 名 修 介†† 金 丸 智 一††

携帯電話などの家電機器への Java 普及が進みつつある中で，Java プログラムの高速化が求められ，Java JIT コンパイラが要望されている．家電機器向けの JIT コンパイラの実装では，実行性能追求よりも家電機器の持つ 2 つの特性「メモリ資源の制約」,「機種展開時の CPU 変更に対応する移植性」を考慮する必要がある．PC 用 JIT コンパイラは，レジスタなど，CPU に依存した構造に最適化することによって，高速化に注力しているが，コンパイラが複雑化してメモリ資源が増え，CPU の移植性が低下するため，家電機器にはそのままでは適用できない．本論文で提案する方式では，1) 最適化を実装しないことでコンパイラサイズを縮小し，2) GCC の利用とインタプリタ C ソースコードからのコンパイラ自動生成による移植性の向上，を実現させた．また，本方式では，bytecode 実行と nativecode 実行が同じ Java オペランドスタックを共有するので，部分コンパイルが容易に実現でき，nativecode を格納するメモリを削減するうえでも有効である．インテル x86 用の実装評価では，コンパイラ自体のメモリ量が 17K バイト，インタプリタの最大 8.7 倍の高速化を実現できた．

A Construction Scheme of Java JIT Compiler for Consumer Electronics and Its Evaluation

Takuji Kawamoto,† Shusuke Haruna††
and Tomokazu Kanamaru††

In Java having wide spread through the consumer electronics appliances, e.q. a cellular phone, the Java JIT compiler is necessary to improve the execution speed of Java programs. In the implementation of the JIT compiler for cunsumer electronics, the characteristics that have “restrictions of memory resources” and “the portability to CPU change at the time of model deployment” need to be taken into more consideration, rather than the improvement in its speed. The JIT compiler for PC has concentrated in improvement of the execution speed by optimizing to the structure depending on CPU, such as registers. While a compiler becomes complicated, memory resources increase and the portability of CPU goes down, as a result, it is inapplicable to consumer electronics. In the proposed scheme by this paper, we reduced the compiler size without adopting the optimization depending on CPU, and raised portability with using GCC and carrying out automatic generation of the compiler from an interpreter source code. Moreover, since byte code and native code share the Java operand stack at execution time, partial compilation is easily realized. Therefore, this system is effective for reducing the memory which stores native codes. Our evaluation results show that compiler memory size is 17 K bytes for Intel x86 processor, and that the execution speed of native codes is a maximum of 8.7 times of an interpreter.

1. はじめに

携帯電話をはじめとして，家電機器への Java の普及が進みつつある．普及にともない Java で書かれたアプリケーションの流通が始まっている．たとえば，携帯電話でのゲームなどのアプリケーションの流通が進展すると，次に，Java 仮想マシンの実行速度向上の要求が高まってくる．従来から，Java 実行速度の向上のためには，ハードウェアアクセラレータや nativecode への変換実行を可能とする JIT コンパイラの実装が行われているが，ハードウェアアクセラレータの実装は，LSI コストが増加し，製品コストの増大につながる．コスト削減の要求が強い家電機器への導入は，敬遠されがちである．

Java が導入されている機器には，通常，ブラウザが搭載されていることが多く，ブラウザが使用するデータ（HTML データなど）格納用メモリを Java の実行用に適用することが可能である．そのため，変換され

† 株式会社松下電器情報システム名古屋研究所
Matsushita Information System Research Laboratory Nagoya Co., Ltd.
†† 松下電器産業株式会社ソフトウェア開発本部
Corporate Software Development Division, Matsushita Electric Industrial Co., Ltd.

た nativecode の格納用に，このメモリを用いることができる JIT コンパイラの導入要望が高まっている．しかし，家電分野での Java 搭載に関しては，家電機器特有の特性のため，パーソナルコンピュータ（以下，PC）に代表されるコンピュータ分野での技術をそのまま用いることはできない[1)]．

本論文では，少ない資源しか持たない家電機器の特性を反映した JIT コンパイラの構成方法を提案し，実装について詳細に説明し，その評価について述べる．以下，本論文の構成を示す．

2 章で本研究の適用領域の特性を明らかにし，3 章で従来の JIT コンパイラを家電領域に適用する際の問題点をあげ，この問題点を解決する家電向け JIT コンパイラの構成方法を述べる．そして，4章〜5 章でシステムの各構成要素について説明し，6 章でシステムの評価結果を述べる．また，7 章で関連研究との差異を明らかにし，8 章で本研究のまとめと今後の課題について述べる．

2. 適用領域の特性

本章では，JIT コンパイラが適用される領域である家電機器の特性を明らかにする．

2.1 資 源 制 約

Java の家電適用を考えると，冷蔵庫・電子レンジといった家庭用電化機器も考慮しなければならない[2)]．これらの機器では，現在においてもプロセッサ（以下，CPU）性能やメモリ容量の資源制約が厳しい．また，携帯電話では，メモリ量の制約は緩やかになっているが，消費電力の制約は現在でも厳しく，内蔵するプロセッサ性能の飛躍的な向上は，望めない．このような資源制約から，JIT コンパイラの使用メモリおよび変換処理量は，極力おさえなければならない．

また，一般に，bytecode は nativecode に比べ数倍規模に展開される．全 bytecode を展開する方法では，使用メモリの増大を引き起こす．使用メモリ削減の観点からは，必要部分のみ nativecode に変換できる部分コンパイルが重要である[12)]．

2.2 要求される性能目標

本節では，家電向け JIT コンパイラに要求される性能目標について議論する．資源が潤沢な PC での JIT コンパイラに関する研究は，性能指向の研究がほとんどである．しかし，家電では，性能とコストのバランスが重要となる．近年，JIT コンパイラと対抗する手法として，ハードウェアアクセラレータ手法が発表されている[3),4)]．これら手法では，数倍〜10 倍以下の実行性能向上を達成しているものがほとんどであり，Java 実行性能の向上に関する 1 つの指標と考えられる．

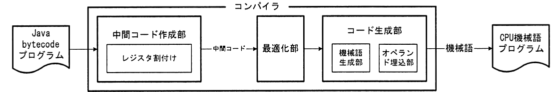

図 1 従来の JIT コンパイラの構造
Fig. 1 A structure of JIT compiler for PC.

2.3 対象プロセッサの変更

家電機器は，コスト要求が厳しく，無駄を極力排除するため，垂直統合型の開発形態をとっており，PC のような標準プラットフォームを規定しにくい．そのため，コスト低減のため毎年のモデルごとに CPU が変更されることもしばしば発生する．業務提携や海外事業との整合などの事業方針によっても使用する CPU は，変更される．また，家電組み込みソフトウェアの急激な規模拡大により，複数の協力会社との共同開発となることが多く，開発者の技術・知識に大きなばらつきが生じる．そのため，必ずしも担当分野に明るい担当者が開発にあたらない場合があり，CPU の変更を前提とした高い移植性が求められる．

3. 本研究のアプローチ

本章では，従来の JIT コンパイラの問題をあげ，この問題を解決する家電向け JIT コンパイラの構成方針について述べる．

3.1 従来の JIT コンパイラの問題

一般的な JIT コンパイラの構成を**図 1** に示す．bytecode をいったん中間コードに変換し，レジスタや CPU スタックフレームなど CPU に依存した構造に最適化を行い，中間コードと対象 CPU の機械語との対応を記入したテンプレートを用いて，nativecode を生成し，bytecode 内のオペランド情報を生成した nativecode に埋め込む[5)]．

最適化は，実行性能の向上が望める[8)〜10)]反面，以下のような問題を引き起こす．

- 必要メモリ資源が増加する
  中間コード生成部，最適化処理部，および中間コード格納のためのメモリが増大し，家電のメモリ制約を満たさない．
- コンパイル時間がかかる
  中間コード生成，最適化処理に CPU パワーを必要とし，家電の CPU 能力制約を満たさない．また，実行のための起動時間が長くなり，ユーザに受け入れられない．
- 部分コンパイルが制限される
  bytecode 環境と nativecode 環境が異なるので，

任意位置での部分コンパイルができない．家電ではメモリ制約のため細かい単位で bytecode と nativecode の切替えが必要である．

- 他 CPU への移植が難しくなる
  Java JIT コンパイラは，Java 言語のスタック構造などがテンプレートに反映されるため，CPU に依存した構造に最適化すると，テンプレート記述が複雑化し，移植性を損なう．

3.2　家電向け JIT コンパイラの構成方針

すでに述べた問題を考慮し，家電向け JIT コンパイラの構成方針は，以下のとおりとする．

(1)　性能追求のアプローチは採用しない．
(2)　実行系の負担を極力おさえる．
(3)　異なる CPU への移植を自動化する．

(1)，(2) のために，性能向上目標を数倍〜10 倍程度とした．これは，ハードウェアアクセラレータ手法と同程度の性能向上をハードコストの増加なしに実現することにある．

このことにより実行系内の最適化処理を排除でき，必要資源量を削減できるが，生成コード量が増大するという欠点がある．反面，CPU 特有のレジスタやスタック構造などに最適化しないので，bytecode と nativecode が同一オペランドスタックを共有することになり任意位置での bytecode，nativecode の混在実行ができ，必要な部分のみをコンパイルする部分コンパイルの実現が容易となる．この部分コンパイルを前提とすることにより，JIT コンパイラでの生成コード増加の欠点をシステム全体として補うことができる．

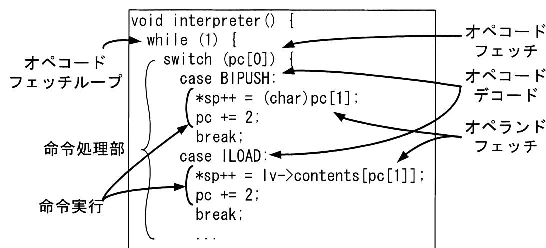

図 2　インタプリタ C ソースコード
Fig. 2　An interpreter source code.

(3) の実現のため，我々は，インタプリタのソースコードに着目した．**図 2** に示すように C 言語で記述されたインタプリタの命令処理部には，各 bytecode に対応した処理素片が含まれており，**図 3** に示すように，以下の手順でテンプレートを自動生成可能である．

i)　インタプリタソースコードを再構成し，C 言語のテンプレートを生成．
ii)　生成された C 言語テンプレートを対象 CPU 用 GCC でコンパイル．

また，オペランド情報の埋め込み処理は，機械語の形式に依存するため，一般には対象 CPU ごとに作成する必要があり，移植の自動化は困難である．本システムでは，テンプレート生成時の差分検出の手法を用いて，対象 CPU ごとのオペランド埋め込みに関する情報☆の自動検出を図っている．

JIT コンパイラのコード生成時の負担を軽減し，かつ移植性を高める手法として，文献 6) にあるように，抽象的なアセンブリ言語を使用してテンプレートを記述しておき，実行時に抽象的なアセンブリ言語から nativecode に変換するというアプローチがある．この手法では，軽量な処理ではあるが，抽象的なアセンブリ言語から nativecode に変換するアセンブル処理

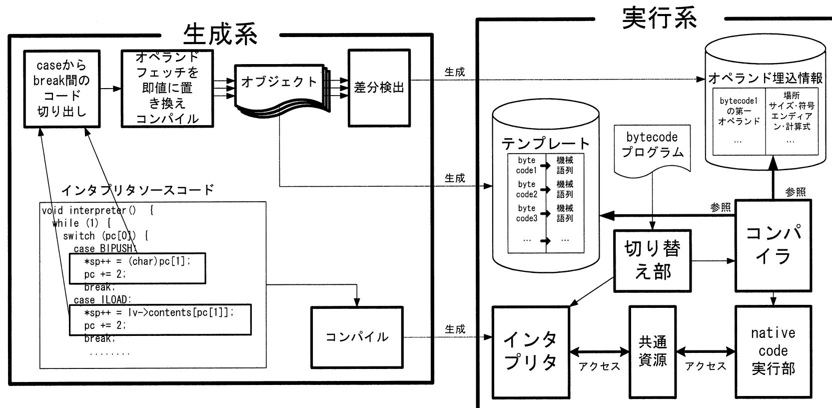

図 3　本 JIT コンパイラのシステム概念図
Fig. 3　A system of our JIT compiler system.

☆ 位置，サイズ，エンディアンなどが含まれる．

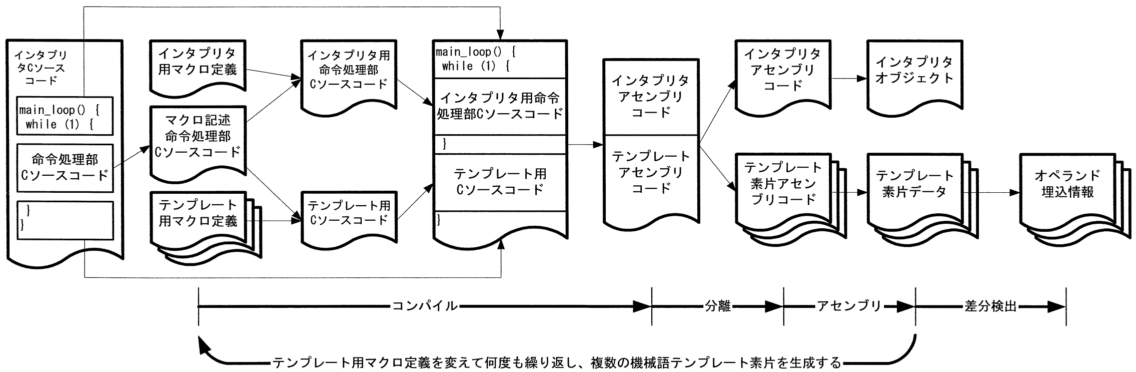

図 4 Java JIT コンパイラ生成システム工程図
Fig. 4 A process of a Java JIT compiler generator.

が実行時に必要である．また，このアセンブル処理も対象 CPU ごとに用意する必要がある．本研究では，実行系内の処理を極力削減することに加えて，テンプレート生成の自動化および移植の自動化の観点から，図 3 に示すように生成系に比重を置いたアプローチを採用した．

本システムは，大きく分けて生成系と実行系から構成される．以下，各構成要素の詳細な説明を行う．

4. 生 成 系

図 4 に示すように，生成系のシステム工程は，コンパイル工程，分離工程，アセンブリ工程の 3 工程と，これら 3 工程を繰り返す差分検出工程からなる．

生成系の処理の概要を説明する．

インタプリタソースコードは，C 言語マクロを用いて書き換える．インタプリタ用マクロとテンプレート用マクロを用意し，コンパイル時に切り替えることにより両ソースコードを生成する．生成されたソースコードは，分離工程・アセンブリ工程においてアセンブルされ，インタプリタ用オブジェクトコードとテンプレート用オブジェクトコードが生成される．差分検出工程において，複数のテンプレートを比較することにより，CPU の機械語形式に依存するオペランド埋め込み情報を抽出する．以下，マクロ記述および，各工程の詳細について述べる．

4.1 マクロ記述

テンプレートをインタプリタのソースコードから自動生成するためには，インタプリタとテンプレートとで，異なった要求を記述し分けなければならない．

たとえば，インタプリタの命令処理部ではオペコードフェッチ処理，オペコードデコード処理が必要であるが，テンプレートにはこれらは不要である．

また，インタプリタではオペランドフェッチ処理が必要であるが，テンプレートには不要である．これは，フェッチされたオペランド値（以下，オペランド即値）は，nativecode 展開時に，即値ロード命令に置き換えられるためである．

さらに，処理が複雑な bytecode 命令の処理部は，展開するよりインタプリタの処理を呼び出すほうが，メモリ資源の節約になる．

これらの実現のために，図 4 の左端のように，インタプリタ C ソースコードをメイン関数 main_loop() から命令処理部の手前までの部分と，命令処理部と，命令処理部の後残りの部分を別のファイルに分離する．それぞれを前処理部 C ソースコード・命令処理部 C ソースコード・後処理部 C ソースコードと呼ぶことにする．

命令処理部 C ソースコードは，インタプリタとテンプレートで処理の異なる部分を**表 1** にあげたマクロで記述しなおす．マクロにはテンプレート切り出し関係マクロ・Java プログラムカウンタ関係マクロ・インタプリタ内関数呼び出し関係マクロがある．

テンプレート切り出し関係マクロは，3 種類ある．

そのうちのスイッチマクロはテンプレート生成時には不要なので削除される．ケースマクロとブレークマクロで，テンプレートの各 bytecode に対応する処理（以下，テンプレート素片）の区切りを示す．

ケースマクロは，テンプレート生成時にはテンプレート素片先頭を示すラベルに置き換えられる．ラベル名としては，引数に書かれた bytecode オペコード値を一部に持つユニークラベルを生成するよう実装した．

ブレークマクロは，インタプリタ生成時にはそのまま出力され，テンプレート生成時にはテンプレート素片終了を示す記号として働く．実装上はダミーラベルへの goto 文により実装した．

表 1 マクロ一覧
Table 1 A macro list.

マクロ種別		マクロ記述	インタプリタ生成	テンプレート生成
テンプレート切り出し関係マクロ	スイッチマクロ	SWITCH(val)	switch(val)	-
	ケースマクロ	CASE(ope_code)	case ope_code:	テンプレート素片先頭を示すラベル
	ブレークマクロ	BREAK()	break	テンプレート素片終了を示す記号
Javaプログラムカウンタ関係マクロ	オペコードマクロ	OPE_CODE(val)	pc[0]	-
	数値オペランドマクロ	OPE_SCHAR(n) 等	(char)pc[n]	オペランド即値
	アドレスオペランドマクロ	OPE_ADDR2(n) 等	pc+(short)(pc[n]*256+pc[n+1])	オペランド即値
	プログラムカウンタインクリメントマクロ	INCREMENT_PC(val)	pc += val	-
インタプリタ内関数呼び出し関係マクロ (関数 int invoke_virtual(int index) 呼び出しの場合)		INVOKE_VIRTUAL(index)	invoke_virtual(val)	invoke_virtual の間接参照呼び出し

Javaプログラムカウンタ関係マクロは，4種類ある．

オペコードマクロは，インタプリタ生成時には pc[0] に展開され，テンプレート生成時には削除される．

数値オペランドマクロは，符号あり 1 バイト（OPE_SCHAR(n)），符号なし 1 バイト（OPE_UCHAR(n)），符号あり 2 バイト（OPE_SSHRT(n)），符号なし 2 バイト（OPE_USHRT(n)），符号あり 4 バイト（OPE_SLONG(n)），符号なし 4 バイト（OPE_ULONG(n)）の 6 種類である．引数は，そのオペランドの先頭が bytecode 命令中に占めるバイト位置（以下，オペランドオフセット）である．インタプリタ生成時，n はプログラムカウンタのインデックスになる．たとえば表 1 の OPE_SCHAR(n) の場合は (char)pc[n] に展開され，OPE_SSHRT(n) の場合は (short)(pc[n]*256+pc[n+1]) に展開される．また，テンプレート生成時にはオペランド即値に展開される．

アドレスオペランドマクロは 2 バイトサイズ（OPE_ADDR2(n)）と 4 バイトサイズ（OPE_ADDR4(n)）がある．インタプリタ生成時には，OPE_ADDR2(n) は pc+(short)(pc[n]*256+pc[n+1]) に展開され，テンプレート生成時にはオペランド即値に展開される．

プログラムカウンタインクリメントマクロは，インタプリタ生成時には pc+=val に展開され，テンプレート生成時には削除される．

インタプリタ内の関数呼び出し関係マクロは，呼び出したいインタプリタ内関数をすべて用意した．表 1 は，メソッド呼び出し関数 int invoke_virtual(int index) の例である．

インタプリタ内の関数呼び出しを実現するには，関数ラベルへの参照をテンプレート内に未解決なまま残し，nativecode ロード時にラベル解決を行う方法が考えられるが，コンパイル速度低下の一因となるうえ，コンパイラの構造が複雑になり望ましくない．

そこで，後述するように，インタプリタの自動変数とテンプレートの自動変数は共有されることを利用し，関数ラベルをインタプリタの自動変数に保持して，テンプレートからこれを間接参照呼び出しを行うことで，ロード時のラベル解決を回避した．表 1 の例では，インタプリタメイン関数 main_loop() 内に int (*)(int) 型の自動変数 ind_invoke_virtual を設けて，テンプレート側からは (*ind_invoke_virtual)(index) を呼び出す．ind_invoke_virtual の初期化は，インタプリタ内の前処理部 C ソースコードで行う．

```
SWITCH(OPE_CODE())
{
CASE(0x00 /* nop */)
  INCREMENT_PC(1);
  BREAK();
CASE(0x01 /* aconst_null */)
  *sp++ = 0;
  INCREMENT_PC(1);
  BREAK();
  ...
CASE(0x10 /* bipush */)
  *sp++ = OPE_SCHAR(1);
  INCREMENT_PC(2);
  BREAK();
  ...
CASE(0x15 /* iload */)
  *sp++ = lv->contents[OPE_UCHAR(1)];
  INCREMENT_PC(2);
  BREAK();
  ...
```

図 5 マクロ記述命令処理部 C ソースコード例
Fig. 5 An example of C source codes written by macro.

```
unsigned char *pc;
#define SWITCH(value) switch(value)
#define CASE(case_value) case case_value:
#define BREAK() break
#define INCREMENT_PC(value) pc+=(value)
#define OPE_CODE() pc[0]
#define OPE_SCHAR(n) (char)pc[n]
#define OPE_UCHAR(n) pc[n]
#define OPE_SSHRT(n) (short)((pc[n]<<8)+pc[n+1])
#define OPE_USHRT(n) ((pc[n]<<8)+pc[n+1])
```

図 6 インタプリタ用マクロ定義の例
Fig. 6 An example of macro definitions for interpreter.

図 5 は，マクロ記述命令処理部 C ソースコード例

```
void *template_table[256] = {
&&template_label_0x00,&&template_label_0x01,
....
&&template_label_0xfe,&&template_label_0xff };

#define SWITCH(value) goto *template_table[0];
#define CASE(case_value) template_label_##case_value:
#define BREAK() goto *template_table[0]
#define INCREMENT_PC(value)
#define OPE_CODE()
#define OPE_SCHAR(n) n+0x70
#define OPE_UCHAR(n) n*2+0xf0
#define OPE_SSHRT(n) n*3+0x7fe0
#define OPE_USHRT(n) n*4+0xffd0
```

図 7 テンプレート用マクロ定義の例
Fig. 7 An example of macro definitions for template.

```
switch(pc[0])
{
case 0x00 /* nop */:
  pc+=1;
  break;
case 0x01 /* aconst_null */:
  *sp++ = 0;
  pc+=1;
  break;
  ...
case 0x10 /* bipush */:
  *sp++ = (char)pc[1];
  pc+=2;
  break;
  ...
case 0x15 /* iload */:
  *sp++ = lv->contents[pc[1]];
  pc+=2;
  break;
  ...
```

図 8 インタプリタ用 C ソースコード生成例
Fig. 8 An example of source code generated for interpreter.

```
void *template_table[256] = {
&&template_label_0x00,&&template_label_0x01,
....
&&template_label_0xfe,&&template_label_0xff };
goto *template_table[0];
{
template_label_0x00 /* nop */:
  goto *template_table[0];
template_label_0x01 /* aconst_null */:
  *sp++ = 0;
  goto *template_table[0];
  ...
template_label_0x10 /* bipush */:
  *sp++ = 1+0x70;
  goto *template_table[0];
  ...
template_label_0x15 /* iload */:
  *sp++ = lv->contents[1*2+0xf0];
  goto *template_table[0];
  ...
```

図 9 テンプレート用 C ソースコード生成例
Fig. 9 An example of source code for template.

である．**図 6** に示すインタプリタ用マクロ定義を適用することで，**図 8** に示すインタプリタ用命令処理部 C ソースコードを得る．また，**図 7** に示すテンプレート用マクロ定義を適用することにより，**図 9** に示すテンプレート用 C ソースコードを得る．

図 7 の先頭の `template_table` は，テンプレート素片の先頭を示すラベルテーブルで，後述する分離工程において，各テンプレート素片の先頭を検出するために使用される．

4.2 コンパイル工程

任意位置での bytecode，nativecode の混在実行を実現するために，インタプリタの自動変数とテンプレートの自動変数を共有化する必要がある．自動変数のオフセットはコンパイル時に GCC が決定するため，インタプリタとテンプレートを同一ファイル内でコンパイルする必要がある．

コンパイル工程では，それぞれのマクロ定義を適用した後，図 4 のように，前処理部 C ソースコード，インタプリタ用命令処理部 C ソースコード，後処理部 C ソースコード，テンプレート用 C ソースコードの順に 1 つのファイルにまとめ，テンプレート用 C ソースコードがインタプリタ C ソースコードのメイン関数 main_loop() 内に同居するようにして GCC を用いてコンパイルし，アセンブリソースコードを出力する．その結果，インタプリタの自動変数とテンプレートの自動変数は，同名であれば同じオフセットに割り当てられることになる．

4.3 分 離 工 程

コンパイル工程で出力されたアセンブリソースコードは，コード共有などの GCC の最適化によって，インタプリタ部分とテンプレート部分が入り組んだ状態になる．分離工程では，このように混在しているインタプリタアセンブリコードとテンプレートアセンブリコードを分離し，テンプレート素片を切り出す．たとえば，iload と iload_1 は，**図 10** にあるように，テンプレート用出力の処理の後半は共通であり，コードが共有されている．

まず，図 7 の変数 `template_table` を検出し，すべてのテンプレート素片の先頭を示すラベルを得る．

次に，各テンプレート素片ごとに，先頭ラベルからテンプレート素片の終了まで，アセンブリソースコードを，ジャンプ先も含めてトレースし，テンプレート素片アセンブリコードをすべて分離し，各テンプレート素片アセンブリコードを別々のファイルに格納する．そして，残った部分を集めてインタプリタアセンブリコードとする．

なお，変数 `template_table` は GCC の最適化処理で削除されないため，これに関連するテンプレート素片の削除をも防いでいる．

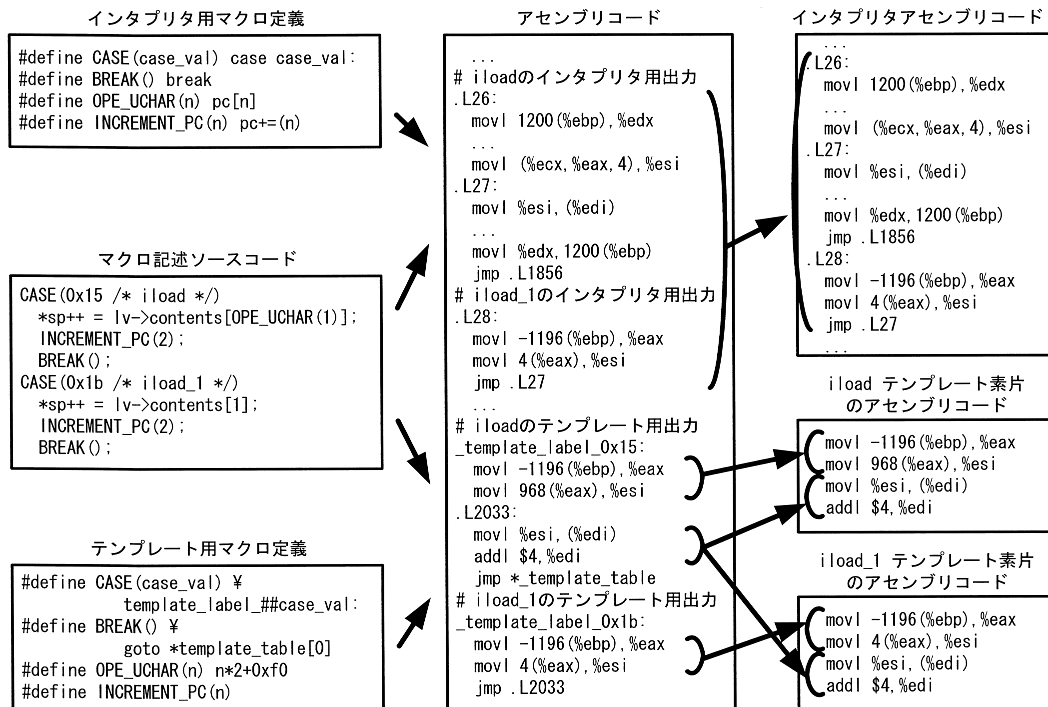

図 10 分離工程の出力例
Fig. 10 An example of separation process.

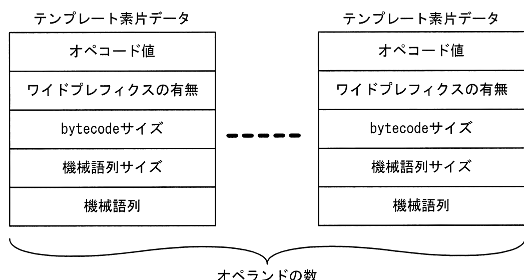

図 11 テンプレート構造
Fig. 11 A structure of the template.

4.4 アセンブリ工程

アセンブリ工程では，インタプリタアセンブリコードをアセンブルしてインタプリタオブジェクトを得る．また，各テンプレート素片アセンブリコードをアセンブルしてテンプレート素片のオブジェクトを生成し，コードセクションから機械語列とそのサイズを抽出する．そして，**図 11** に示すように，オペコード値・bytecode のサイズなどの情報を付加し，各テンプレート素片データを生成する．各テンプレート素片データをまとめ，テンプレートとする．

なお，テンプレート素片の bytecode サイズには，固定長命令の場合はその命令長を格納し，可変長命令 tableswitch，lookupswitch については実行時に命令長を計算するためダミーを格納する．

4.5 差分検出工程

最後の差分検出工程では，表 1 の数値オペランドマクロ・アドレスオペランドマクロを異なる即値で置き換え，コンパイル工程以降を繰り返すことにより得られたテンプレートの差分を調べることにより，オペランド埋め込み情報を生成する．

差分検出工程において，以下の項目を検出する．

(1) エンディアン
(2) オペランド埋め込み場所と，そこに埋め込まれているオペランドオフセット☆
(3) オペランド埋め込みの計算方法
bytecode のオペランドは，数値計算のオペランドに使われる場合と，Java 言語のローカル変数などの配列のインデックスに使われる場合がある．後者の場合，機械語のメモリアドレッシングモードのインデックスに埋め込まれるが，GCC による機械語生成の過程において，オペランド即値が，配列の 1 要素のバイト数倍され

☆ オペランドオフセットとは，インタプリタソース中に記述されている pc[n] の n のことを指す．

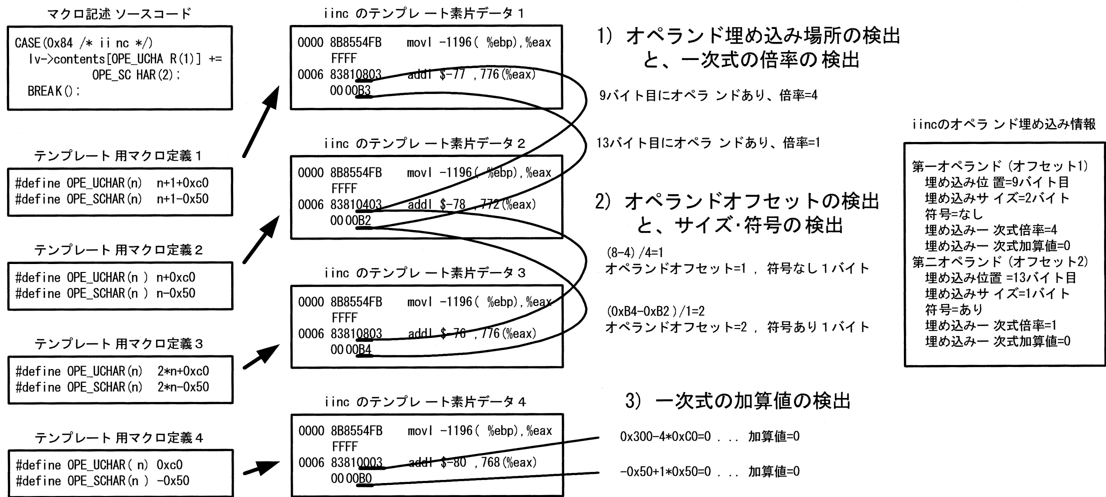

図 12　差分検出工程の出力例
Fig. 12　An example of difference detection process.

たり，配列の先頭オフセットが足し込まれたりする．機械語の即値として埋め込まれる値は，**n*倍数+加算値**の一次式で表される．このような埋め込み計算の一次式を検出する．

(4)　オペランド埋め込みサイズ

これらの検出方法を，**図 12** の iinc の例を使って説明する．iinc は，int 型のローカル変数に整数を加算する bytecode である．ローカル変数のインデックスは，bytecode のオフセット 1 にある符号なし 1 バイトオペランドで指定され，加算する即値はオフセット 2 にある符号付き 1 バイトオペランドで指定される．

エンディアンの検出方法については後述し，ここではリトルエンディアンであると検出できたものとする．説明用の図は，インテル x86 の例である．

1) まず，図 12 のテンプレート用マクロ定義 1，2 を使って生成した iinc のテンプレート素片データの差分を調べ，オペランドが埋め込まれた場所と，計算方法の一次式の倍数を検出する．

0xc0 を加算しているのは，後述する短縮形を回避するためであるが，差分を計算する場合，加算の影響は相殺されるので，ここの検出では無視できる．マクロ定義 1，2 は値が 1 違うので，この比較によって違いが現れたところがオペランドが埋め込まれた場所であり，その差は計算方法の一次式の倍数である．図 12 の場合は，先頭から 9 バイト目と 13 バイト目にオペランドが埋め込まれており，倍数はそれぞれ 4，1 と検出される．

2) 次にテンプレート素片データ 2，3 の差分を調べ，各埋め込み位置にあるオペランドオフセット・符号の有無・サイズを検出する．マクロ定義 2，3 は n に対する増分が 1 違うので，テンプレート素片データ間で差のある場所の数値を，先に検出した倍数で割るとオペランドオフセットが得られ，iinc の仕様からオペランドの符号の有無やオペランドサイズが検出できる．図 12 の場合は先頭から 9 バイト目はオフセットが 1 と検出できるので，iinc の仕様から符号なし 1 バイトと分かる．13 バイト目はオフセットが 2 と検出でき，同様に符号あり 1 バイトと分かる．

3) 最後にテンプレート素片データ 4 を調べ，一次式の加算値と埋め込みサイズを検出する．差分検出方法では加算値は相殺されるため，1 つのテンプレート素片データから検出する．この段階ではエンディアン・埋め込み場所・倍数・符号の有無・bytecode 中のオペランドサイズがすでに分かっているので，これらの情報から加算値を計算できる．

図 12 の場合は，9 バイト目は倍数が 4 であり，符号なし 1 バイトのオペランドであることから，この埋め込み計算式は 0xc0*4+**加算値**と考えられ，埋め込みサイズは 2 バイトと予想される．そこで 9 バイト目と 10 バイト目から 0x300 を取り出し，加算値 0x300-4*0xc0=0 を検出する．この例では，機械語内の埋め込みサイズは 4 であり，予想した 2 とは異なる．しかし，符号なしオペランドの場合は，初期値として 0 が埋め込まれていれば，長さの短い即値を埋め込んでも上位 2 バイトはつねに 0 になるので実用上問題はない．

13 バイト目は，-0x50(0xB0) であり，倍数が 1 の場所であるので，埋め込み計算結果が 1 バイトに収ま

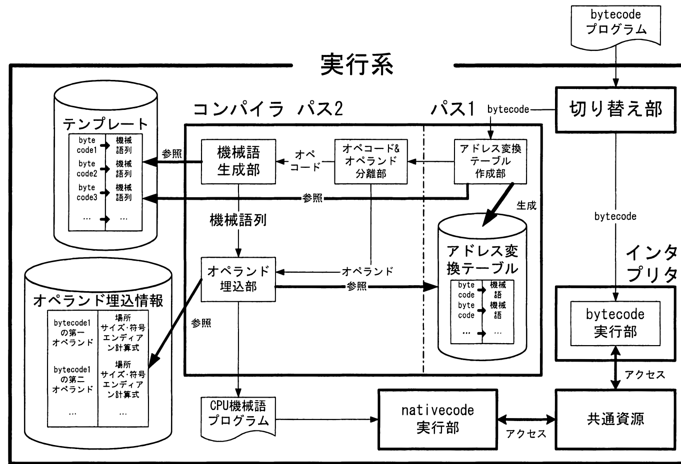

図 13　実行系の構成図

Fig. 13　A structure of an execution system.

ることに注意して，同様に計算すると加算値が 0 であることを検出できる．符号ありオペランドの場合，埋め込みサイズを誤って短く検出すると符号拡張の問題が発生するため，この後，正の数 0x50 を定義したテンプレート素片データとの差分を調べて，正しい機械語命令上の埋め込みサイズを検出する必要がある．

4) これらの検出結果を元に，図 12 の右のような iinc のオペランド埋込情報を生成する．

なお，CPU によっては短縮形の即値オペランドを備えるものがあり，そのアセンブラは値の大きさにより出力を短縮形に切り替えることがある．たとえば，符号なし 8 ビット幅のオペランドで 7 ビット以下に収まる即値を使用すると，短縮形が出力される恐れがある．差分検出工程で，誤って通常形と短縮形の出力を比較しないために，図 12 のテンプレート用マクロ定義で 0xc0 などの数値を加算している．差分を計算する場合，加算の影響は相殺される．

さらに，エンディアン検出については，4 バイトアドレスオペランドの埋め込み位置を検出したあと，そのオペランドマクロに 0xf0e0d0c0☆を定義し，生成したテンプレート素片データを使って検出可能である．このオペランドは，アドレスとして扱われるため，機械語中には，一次式の計算を行わずそのまま埋め込まれる．そのため，テンプレート素片データには 0xf0, 0xe0, 0xd0, 0xc0 がエンディアンに従った順序で現れるので，これを他の検出項目に先立って検出する．

☆ この値に限るものではなく，エンディアンの検出に便利な数値であれば何でもよい．

5. 実　行　系

図 13 に実行系の構成図を示す．

コンパイラは 2 パス構成である．パス 1 のアドレス変換テーブル作成部は，入力された bytecode 列の各命令のオペコードをキーにテンプレートを参照して，各オペコードに対応する機械語列の長さを計算し，各命令の bytecode アドレスと機械語アドレスの組をアドレス変換テーブルに設定する．固定長命令の場合は，テンプレート素片に書かれている機械語列サイズを使用し，可変長命令の場合は，命令に書かれている情報を元に機械語列の長さを求め，機械語アドレスを算出する．

パス 2 の機械語生成部は，同様にテンプレートを参照して，各命令に対応する機械語列を生成する．オペランド埋め込み部は，オペランド埋め込み情報を参照して，数値オペランドを機械語列の適切な位置に埋め込む．アドレスオペランドはアドレス変換テーブルを参照して機械語アドレスに変換してからオペランド埋め込み情報の示す位置に埋め込む．

このようにして生成された機械語列は nativecode 実行部で実行される．

6. 評　　価

評価のため，他の JIT コンパイラとの比較が容易なインテル x86 プロセッサ用に実装を行った．対象とするインタプリタは，ソースコードの入手の容易性から，我々の所有するインタプリタ[1)]を用いた．このインタプリタのコードサイズは，72 K バイトである．

実行環境は，FreeBSD 4.1.1-RELEASE，Pentium

表 2 コンパイラコードサイズ
Table 2 Compiler code sizes.

	サイズ
テンプレート	9,908 byte
オペランド埋込情報	971 byte
コンパイラコード	6,571 byte

III 550 MHz である．

評価用アプリケーションは，組み込み用ベンチマークプログラム Embedded CaffeineMark 3.0☆を，評価しやすいように 10,000 回繰り返すよう修正して使用した．

評価用アプリケーションは，SpecJVM☆☆を用いるのが一般的であるが，SpecJVM は，インタプリタが実行されるプラットフォーム（OS・入出力・ファイルシステム・表示など）を含めたシステムの評価を行うベンチマークであり，インタプリタそのものの性能改善を見るには不適当である．また，本研究の適用領域で想定するメモリ規模に合致しない．

Embedded CaffeineMark 3.0 は，プラットフォームに依存する表示や入出力などを含まないアプリケーション群から構成されているため，インタプリタそのものの性能改善評価が可能なベンチマークとして採用した．

我々のインタプリタは，静的リンカにより，class ファイル中のシンボル参照関係を事前に解決後，実行するプレリンク方式を採用しているため，評価用アプリケーションの bytecode を静的リンカにより処理した後の bytecode をコンパイル対象としている．

以下，コンパイラコードサイズ・コード生成時間・実行速度・移植性の評価結果について述べる．

6.1 コンパイラコードサイズ

コンパイラコードサイズの評価結果を**表 2** に示す．テンプレートのサイズは 9,908 byte，オペランド埋め込み情報のサイズは 971 byte, コンパイラのコードサイズは 6,571 byte であり，全コードサイズは 17 KB になる．x86 用 JIT コンパイラを備える kaffe version 1.0.6☆☆☆のコンパイラ部 JIT3 のコードサイズ 74 KB に比べて本システムは約 4 分の 1 であり，実行系の負担を極力おさえる本研究の目的に合致したものとなっている．これは，kaffe JIT3 のようなレジスタ割付けなどの CPU に依存する最適化を実施しておらず，コンパイラの構造が簡単になっているためである．

☆ http://www.pendragon-software.com/pendragon/cm3
☆☆ http://www.specbench.org/osg/jvm98/
☆☆☆ http://www.kaffe.org/

表 3 コード生成時間
Table 3 Code generation time.

	本技術の JIT	kaffe JIT3	比
logic	959 μ 秒	4,102 μ 秒	0.23 倍
loop	587 μ 秒	1,006 μ 秒	0.58 倍
method	141 μ 秒	478 μ 秒	0.29 倍
sieve	376 μ 秒	885 μ 秒	0.42 倍
string	284 μ 秒	778 μ 秒	0.37 倍

表 4 実行速度
Table 4 Execution speed.

logic	loop	method	sieve	string
8.7 倍	4.5 倍	2.1 倍	3.1 倍	1.0 倍

6.2 コード生成時間

コード生成時間の評価結果を**表 3** に示す．評価用アプリケーションの主要メソッドである execute() の全 bytecode に対応する nativecode を生成するのにかかった時間を計測した．コード生成時間は，kaffe JIT3 比，平均 0.32 倍と高速である．この結果は，最適化処理が排除されている本コンパイラの特長を反映したものとなっている．

6.3 実 行 速 度

インタプリタによるアプリケーション実行速度と nativecode によるアプリケーション実行速度の比を**表 4** に示す．

string テストは，多くの bytecode でインタプリタ内処理を呼び出すコードが生成されているため，コンパイラによる速度改善はない．一方，logic テストでは，ほとんどの bytecode は nativecode に展開されているため，今回の評価アプリケーション中では，最大の性能改善 8.7 倍を示した．このようにインタプリタ内処理を呼び出す頻度の差により，性能改善の差が生じている．

文献 1) の記載にあるように，我々のインタプリタは，前述のプレリンクの効果により，インタプリタの実行性能が通常の JavaVM より 3 倍程度向上しているため，通常の JavaVM に適用した場合は，表 4 に示す以上の速度改善が期待できる．

6.4 移 植 性

対象プロセッサ変更への対応性評価として，Motorola 社の M68K シリーズ，32 ビット組込みマイコン[7)]への移植を行った．移植作業は，GCC を変更するだけで完了し，移植の自動化が確認できた．

以下，他プロセッサへの対応について，考察を行う．

RISC 型プロセッサは，その特性から機械語のオペランドとして 32 ビット即値を使えるものが少ない．この即値を必ずしもバイト境界でない 2 つの値に分割し

て使用するタイプ☆や，即値をデータとして配置して間接参照するタイプ☆☆がある．

前者のタイプの CPU については，分割された即値のビット幅を差分検出前に指定することで，対応が可能である．

後者のタイプの CPU については，1 つの bytecode のコード生成ごとに即値を生成し，それを飛び越すようなジャンプ命令を生成していたのでは，メモリ効率も実行効率も悪くなる．これを避けるには，実行系のパス 1 で，即値を生成コードから参照可能な領域内にまとめて配置し，配置した即値への参照コードをパス 2 で生成することにより対応が可能である．

7. 関連研究

3 章で述べたように，従来の JIT コンパイラの研究のアプローチは，資源投入型の速度追求アプローチであった[8)〜10)]．我々の研究のアプローチは，性能追求ではなく，少ない実装資源の中での実装と CPU 変更に際する移植の自動化にある．

Java ポータブル JIT コンパイラとして，kaffe がある．移植に関しては，プロセッサ依存部分を明確にし，この部分を対象プロセッサごとに修正することで対応している[11)]．この方法では，移植にコンパイラの知識が必要であるとともに移植の自動化が難しい．

JIT コンパイラの実装規模の縮小を目的とした研究に文献 12) がある．この文献では，インタプリタの実行イメージから switch 文の各 case 部分をテンプレートとして抽出する方式が論じられているが，オペランドフェッチや Java プログラムカウンタ操作処理も nativecode に展開されるため，生成コードの実行性能低下およびサイズの増加を引き起こす．本システムは，インタプリタのソースコードからコード生成用テンプレートを生成するので，生成系の中でオペランドフェッチや Java プログラムカウンタ操作処理を削除することが可能であり，生成コードの実行性能およびコード生成効率が高い．また，文献 12) では，移植性に関しては，考慮されていない．

小型機器として，PDA 向きの Java 実行の高速化についての研究として，文献 13) がある．この文献では，PC との接続を前提として，PC 上でプロファイル情報に基づく nativecode への変換を行い，PDA が PC と接続されたとき，PDA に対して，nativecode をダウンロードする．PC の存在を前提とできない家電機器に適用することは，難しい．また，PDA に対して，bytecode のままでのダウンロードができないため，Java に期待されているマルチプラットフォーム性を生かしたサービスプログラムの配信に適用できない．

☆ SPARC では，10 ビット即値と 22 ビット即値に分ける．
☆☆ 日立社 SH や ARM 社 ARM がこのタイプに属する．

8. おわりに

本研究では，近年，家電機器への適用が始まっている Java プログラムの高速化手法として，家電向き Java JIT コンパイラの構成方法について提案を行った．本手法は，メモリおよび CPU 能力などの資源制約が厳しく，かつ商品企画などによる CPU の変更が発生する家電領域に適用することを主眼としている．本手法では，インタプリタソースコードからコンパイラのコード生成用テンプレートと CPU 依存のオペランド埋め込み情報の自動生成を行うことにより，コンパイラの必要資源を削減し，CPU 変更時のコンパイラの移植の自動化を実現できた．インテル x86 用のコンパイラの試作・評価では，コンパイラ自体のメモリ量が 17 K バイト，インタプリタの最大 8.7 倍の高速化を実現できた．また，移植性に関して，複数の組込み CPU に対して，自動的に移植できることを確認した．

今後，家電向けの簡素化された例外処理機構・ガーベージコレクション機構の研究に取り組み，家電向けの Java 実行環境全体の完成度を高めていきたいと考える．

謝辞　本研究をまとめるにあたり貴重なご意見をいただいた神戸大学工学部情報知能工学科鎌田十三郎助手に謝意を申し上げます．また，本研究の機会を与えていただき日頃ご指導いただくソフトウェア開発本部杉本豊三参事，今井良彦所長をはじめとする開発や議論に参加いただいた諸氏に深く感謝いたします．

参考文献

1) 春名修介，金丸智一，吉田　力，和氣裕之，富永宣輝：家電向け仮想マシンアーキテクチャ，情報処理学会論文誌：プログラミング，Vol.41, No.SIG4 (PRO7), pp.32–41 (2000).
2) 浅部　勉，西川　宏，長光左千男，宮部義幸：家庭の情報化：家電業界が考える家庭情報化へのアプローチ，情報処理学会誌，Vol.42, No.11, pp.1070–1076 (2000).
3) 足立直大：既存のプロセッサ・アーキテクチャを拡張して高速処理—ARM プロセッサの Java 拡張 “Jazelle”，*Design Wave Magazine*, No.39, pp.95–100 (2001).
4) Levy, M.: アクセラレータ LSI で Java バイトコードを変換，日経エレクトロニクス，No.797,

pp.168–176 (2001).
5) 中田育男：コンパイラの構成と最適化，朝倉書店.
6) Engler, D.R.: VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System, *Proc. Conference on Programming Language Design and Implementation* (*PLDI'96*), pp.160–170 (1996).
7) 松下電子工業株式会社マイコン事業部：MN-10300 シリーズ命令説明書.
8) 志村浩也，木村康則：Java JIT コンパイラの試作，情報処理学会計算機アーキテクチャ研究会報告，No.120-007, pp.37–42 (1996).
9) Ishizaki, K., Kawahiro, M., Yasue, T, Takeuchi, M., et al.: Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler, *ACM Java Grande Conference*, pp.119–128 (1999).
10) Arnold, M., Fink, S., Grove, D., Hind, M. and Sweeney, P.: Adaptive Optimization in the Jalapeno JVM, *OOPSLA '00*, pp.47–65 (2000).
11) Ganapathi, M., Fisher, C. and Hennesy, J.: Retargetable Compiler Code Generation, *ACM Computing Surveys*, Vol.14, No.4, pp.573–592 (1982).
12) Manjunath, G. and Krishnan, V.: A small Hybrid JIT for Embedded Systems, *ACM SIGPLAN Notices*, Vol.35, No.4, pp.44–49 (2000).
13) 有馬　啓，並木美太郎：PDA における Java 実行の高速化の一方式，情報処理学会論文誌，Vol.42, No.6, pp.1535–1544 (2001).

(平成 14 年 2 月 19 日受付)
(平成 14 年 5 月 16 日採録)

川本　琢二

昭和 37 年生．平成 2 年名古屋大学大学院理学研究科博士後期課程満了．理学博士．同年（株）松下電器情報システム名古屋研究所入社．プログラミング言語処理系，ソフトウェア開発環境に関する研究開発に従事．日本数学会会員．

春名　修介（正会員）

昭和 29 年生．昭和 52 年神戸大学工学部電子工学科卒業．同年松下電器産業（株）入社．現在，ソフトウェア開発本部勤務．マイクロコンピュータアーキテクチャ，プログラミング言語処理系，ソフトウェア開発環境に関する研究開発に従事．平成 14 年神戸大学大学院自然科学研究科博士後期課程修了．博士（工学）．電子情報通信学会，ACM 各会員．

金丸　智一

昭和 47 年生．平成 9 年早稲田大学大学院理工学研究科情報科学専攻修士課程修了．同年松下電器産業（株）入社．現在，ソフトウェア開発本部勤務．プログラミング言語処理系の研究開発に従事．