Java向け静的コンパイラによる仮想メソッド呼び出しの高速化

千葉雄司†

Javaアプリケーションを高速実行する手段の1つに，静的コンパイラがある．静的コンパイラの利点は，JITコンパイラより長い時間をかけて最適化を実施できる点にある．しかし，クラス間最適化を施した静的コンパイル済みコードは，プログラムの実行開始当初から利用できるとは限らず，静的コンパイル時に参照したクラスの初期化が済むまでリンクを待たなければならない場合がある．そこで，Java向け静的コンパイラでは，クラス間最適化の実施にあたり，コードの実行速度だけでなく，リンク時期についても配慮する必要がある．本論文では，仮想メソッド呼び出しを高速化するクラス間最適化であるI-call if変換について，リンク時期への配慮を含め，Java向け静的コンパイラで実現する方針について検討する．SPECjvm98による評価結果から，Java向け静的コンパイラではメソッドチェック変換と限定的I-call if変換を積極的に適用すると実行速度が速くなる傾向があり，メソッドチェック変換のみで最適化すると，クラスチェック変換のみで最適化する場合より最大で9.83%高速化できることが分かった．

Optimization of Virtual Method Calls by a Static Compiler for Java

Yuji Chiba†

A static compiler for Java can accelerate performance of Java applications by many kinds of optimizations. But inter-class optimizations can delay linkage of statically compiled code until the classes referred in the static compilation to be initialized at runtime. Therefore, a static compiler for Java should consider not only the execution speed of the code but also the delay of the linkage in making use of inter-class optimizations.This paper describes consideration on implementating I-call if conversion, which is an inter-class optimization for virtual method calls, in a static compiler for Java. The result of SPECjvm98 show that method-check conversion and limited I-call if conversion accelarate performance the most, and method-check conversion improve the performance by up to 9.83% in comparison with class-check conversion.

1. はじめに

Java™☆は教育分野から商用アプリケーション開発現場まで，幅広く普及しつつあるオブジェクト指向プログラミング言語である．Javaが普及する要因として，豊富なライブラリや機種非依存性など，数々のJavaの利点を列挙できる．しかしその一方で，Javaには，C言語やC++で開発した場合に比べ，アプリケーションの実行速度が遅くなりやすいという問題がある．

Javaアプリケーションの実行速度を改善する手段はいくつかあるが，それらのうちの1つに静的コンパイラがある．Javaアプリケーションを構成するクラスは，クラスファイルというファイルに収められている．クラスファイルは，クラスが定義するメソッドをバイトコード形式で保持する．バイトコードは機械非依存の中間語であり，プロセッサで直接実行できない．バイトコード形式のメソッドを実行するには，インタプリタを用いるか，コンパイラでネイティブコードに変換してから実行する．Java向けコンパイラにはJust In Time（JIT）コンパイラと静的コンパイラの2種類があり，前者はプログラムを実行時にコンパイルし，後者は実行開始前にあらかじめコンパイルする．

一般にコンパイラは，最適化を実施してアプリケーションの実行を高速化できる．しかし，最適化を実施するには，数々の情報を収集しプログラムを変換するための時間が必要になる．実行時にコンパイルを行うJITコンパイラでは，最適化に長い時間をかけると，その間アプリケーションの実行が停止するなど問題が起きる．したがって，JITコンパイラでは時間のかかる最適化を実施しにくい．これに対し，プログラムを

† 日立製作所システム開発研究所
Systems Development Laboratory, Hitachi, Ltd.

☆ Javaは米国およびその他の国における米国Sun Microsystems, Inc.の商標です．

実行開始以前にコンパイルする静的コンパイラでは，比較的長い時間をかけて最適化を実施できる．

ただし，Java 向け静的コンパイラでは，クラス間最適化（複数クラスを参照して実行する最適化）を実行しすぎると，生成したコードが実行時に使用できなくなることがある☆．

たとえばクラス変数参照の高速化について考える．Java の言語仕様[5),7)] は，クラスを初めて参照する際に初期化するよう規定している．したがって，クラス変数参照のようにクラスを初めて参照しうる動作については，実行する前にクラスが初期化済みか検査する必要がある．

しかし，クラス変数参照のたびにクラスが初期化済みか検査するのは非効率的である．この問題の解決策として，クラス変数参照を検査抜きで実行するコードに静的コンパイルし，代わりに静的コンパイル済みコードのリンクをクラスの初期化が済むまで遅延する方法がある☆☆．

リンクの遅延はクラス間最適化を施すほど長くなり，最悪の場合ではプログラムの実行が終わるまで静的コンパイル済みコードが使用できずじまいになる．反面，クラス間最適化はインライン展開など多くの有効な最適化を含み，実施しないと実行速度を改善できない．そこで，Java 向け静的コンパイラでは，リンクの遅延とコードの実行速度の両方を考慮してクラス間最適化を実施する必要が生じる．

クラス間最適化のうち，Java 向けコンパイラにとって重要なものの 1 つに，仮想メソッド呼び出し☆☆☆ の高速化がある．仮想メソッド呼び出しは，オブジェクト指向プログラミング言語が提供する機能の 1 つで，手続き型言語の手続き呼び出しに相当する．ただし，手続き呼び出しとは違い，仮想メソッド呼び出しが呼び出すメソッドは，呼び出し対象のオブジェクトのクラスに対応して変化する．仮想メソッド呼び出し（**図 1** 上段）の基本的な実行手順を図 1 中段に示す．すなわち，まず，呼び出し対象のオブジェクトのクラスと，呼び出すメソッドの名前を鍵として，呼び出し対象のメソッドのアドレスを検索し，次に，得られたアドレスに間接ジャンプする．

図 1 中段の関数 `LookUpMethod(`*class*`,` *name*`)` は

```
// Java ソース
obj.m();
```

```
// 基本的な実現
code = LookUpMethod(obj->class, ‘‘m’’);
code(obj);
```

```
// ディスパッチ表法
code = obj->dispatch_table[O_m];
code(obj);
```

図 1 仮想メソッド呼び出しとその実現
Fig. 1 A virtual method call and its implementations.

次の手順でメソッドを検索する．まず，クラス *class* が定義するメソッドに，名前が *name* のものがあるか調べ，あるならば，そのコードのアドレスを返戻し，ないならば，クラス *class* の親クラスに再帰的に同様の操作を適用し，その結果を返戻する．

関数 `LookUpMethod()` の実行手順から想像できるように，図 1 中段の仮想メソッド呼び出しの実現はオーバヘッドが大きく実用的でない．Java をはじめとするオブジェクト指向のプログラミングでは，仮想メソッド呼び出しを頻繁に実行するため，高速に実現する必要がある．仮想メソッド呼び出しの高速化は，オブジェクト指向プログラミング言語の実装の分野で古くから研究が進んおり，これまでに数多くの高速化技法が提案されている[11)]．Java に適用できる高速化技法には，ディスパッチ表法[11)] や I-call if 変換[1)] がある．

本論文の目的は，これらの高速化技法のうち I-call if 変換について，Java 向け静的コンパイラで実現するにあたり，リンクの遅延など固有に考慮すべき問題を示し，その解決策を定量的評価から検討することにある．

本論文の構成について述べる．まず，2 章でディスパッチ表法の実現を例に，静的コンパイル済みコードを実行時にリンクする手順など，Java 向け静的コンパイラ固有の動作について具体的に述べる．次に，3 章で I-call if 変換の実現について述べ，続く 4 章で関連研究を示す．5 章は結論である．

2. ディスパッチ表法

ディスパッチ表法は，仮想メソッドの検索を高速化する．関数 `LookUpMethod()` は子クラスから親クラスへと順次，呼び出し対象のメソッド *m*`()` を定義するか調べてメソッドを検索するが，ディスパッチ表法では個々のクラスについて，あらかじめ検索した結果を収めた表（ディスパッチ表）を用意し，この表を使ってメソッドを検索する．ディスパッチ表法で図 1 上段の仮想メソッド呼び出しを実行するコードを図 1 下段

☆ 動的ロードのサポートが不完全な静的コンパイラでは，この問題を無視する．
☆☆ リンクを遅延しているメソッドについては，JIT コンパイラやインタプリタを使って実行する．
☆☆☆ private でも static でもないメソッド呼び出し．C++ では仮想関数呼び出し，SmallTalk ではメッセージ送信と呼ぶ．

に示す．コードから分かるように，ディスパッチ表法では 2 回☆のメモリ参照のみでメソッドを検索できる．

図 1 下段のコード中の定数 O_m は，呼び出し対象の仮想メソッド $m()$ のアドレスを収める，ディスパッチ表上のエントリのオフセットである．$m()$ を宣言するクラスとそのすべての子クラスのディスパッチ表の O_m 番目のエントリには，そのクラスのインスタンスが実行する $m()$ の実体のアドレスを収めておく．定数 O_m はクラス階層の上位にあるクラスが宣言する仮想メソッドから順に付番して定める．したがって，定数 O_m を求めるためには，$m()$ を宣言するクラス C_0 とその全親クラスを参照する必要がある．

2.1 静的コンパイラによるディスパッチ表法の実現

静的コンパイラもディスパッチ表法を使ってメソッド呼び出し `obj.`m`()` を図 1 下段のコードで実現できる．ただし，最適化の過程で定数 O_m を計算するためにクラス C_0 と，その全親クラスを参照するので，最適化済みのコードは，実行時にそれらすべてのクラスをロードし，クラスファイルが静的コンパイル時に参照したものと同一であると確認するまで使用できなくなる．なぜなら，静的コンパイル後にクラスファイルを更新すると O_m の値が変化して静的コンパイル済みコードを使用不能にしうるからである．

一般に静的コンパイル済みコードは，静的コンパイル時に参照したクラス（仮定クラスと呼ぶことにする）を実行時に動的ロードし，そのクラスファイルが静的コンパイル時に参照したものと同一であると確認し，さらに，一部の仮定クラスについて初期化が完了すれば使用可能になる．静的コンパイラは個々のメソッドをコンパイルする際，仮定クラスを計算し，静的コンパイル済みコードと一緒に実行系に引き渡す．実行系はプログラムの実行時に，この仮定クラスに関する情報を参照し，個々のメソッドについて静的コンパイル済みコードをリンクする時期を定める．

現在開発中の静的コンパイラつき Java 実行環境 JeanPaul[12] の実行系は，すべての仮定クラスについてクラスファイルの同一性を確認し，なおかつ初期化が完了した時点で使用可能と見なしてリンクする．この作業の具体的な手順を次に示す．まず，実行系が個々のクラスを初期化する際，クラスファイルの同一性を確認する．同一と確認した場合，その結果として使用可能になった，すなわちすべての仮定クラスについて同一と確認できた静的コンパイル済みメソッドがあるか調べる．あるならば，そのコードをディスパッチ表に登録することでリンクする☆☆．こうすると，次の呼び出しからは静的コンパイル済みコードが呼び出される．

```
void n()
{
  obj.m();
}
```

図 2 ソースコードの例
Fig. 2 An example of source code.

静的コンパイラにおいて仮定クラスを計算する方法には様々な実装がありうるが，JeanPaul の静的コンパイラにおける実装を次に示す．まず，コンパイル対象のメソッド $n()$（たとえば**図 2**）について，集合 S_n と配列 N_n を用意する．集合 S_n には仮定クラスを収める．たとえばメソッド $n()$ 中のメソッド呼び出し `obj.`m`()` をディスパッチ表法で最適化する場合，メソッド $m()$ を宣言するクラス C_0 とその全親クラスを参照するので，S_n に次の操作を施す（ここで $P(C_0)$ は C_0 とその全親クラスからなる集合を表す）．

$$S_n \leftarrow S_n \cup P(C_0)$$

一方，配列 N_n には，実行時にメソッド $n()$ と同時あるいはそれ以前に，静的コンパイル済みコードをリンクするべきメソッドを収める．原則として，メソッド $n()$ の静的コンパイル済みコードから別のメソッドの静的コンパイル済みコードを直接呼び出しするとき，呼び出し先のメソッドを配列 N_n に追加する．こうすることで，直接呼び出しを経由して，実行時にまだ使用可能になっていない呼び出し先の静的コンパイル済みコードを呼び出すことを防止する．ディスパッチ表法による最適化では，直接呼び出しを生成しないので，N_n には要素を追加しない．N_n に要素を追加する最適化の例には，クラスメソッド呼び出しを，静的コンパイル済みコードへの直接呼び出しに変換する

☆ `obj` からディスパッチ表を取得するためのメモリ参照と，ディスパッチ表からメソッドのアドレスを取得するためのメモリ参照．

☆☆ JeanPaul の実装はクラスファイルの同一性確認のみ必要とするクラスと，初期化まで必要とするクラスを分離しない．しかし，クラスファイルの同一性確認については，JeanPaul のようにクラスの初期化時に逐次実行する必要は必ずしもなく，たとえばプログラムの実行開始時に一括して実行してもよい．このように，クラスファイルの同一性の確認と初期化の完了確認を別個に実施する場合には，分離が必要になる．クラスファイルの同一性をプログラムの実行開始時に一括して確認する方法には，JeanPaul のようにクラスの初期化時に逐次確認する方法より早いタイミングで静的コンパイル済みコードが使用可能になりうるという利点がある．ただし，その反面，標準クラスライブラリ `java.lang.Class` が提供するメソッド `forName()` を使うプログラムの挙動が，現在市場に存在する多くの Java 実行環境と違うものになるなど問題もあり，JeanPaul では互換性重視などの観点から採用しなかった[12]．

最適化や，次章で述べる I-call if 変換がある．

JeanPaul の静的コンパイラは最終的に，配列 N_n を仮定クラスの集合に還元し，集合 S_n に加算する[12]．実行時には集合 S_n 中の全クラスについてクラスファイルの同一性の確認と初期化が完了した場合に，$n()$ の静的コンパイル済みコードを使用可能と見なしてリンクする．本論文では JeanPaul の実装にならい，次章で I-call if 変換が仮定クラスに与える影響について述べる際にも集合 S_n と配列 N_n を用いる．

3. I-call if 変換

I-call if 変換はメソッド呼び出しを，呼び出し対象のメソッドを特定するための if 文と，特定したメソッドを直接呼び出す文に変換する．メソッド呼び出し `obj.`m`()` を I-call if 変換したコードの例を**図 3** に示す．図 3 のコード中でクラス C_1，C_2 は呼び出し対象のオブジェクト `obj` がとりうるクラスであり，コンパイル時にクラス階層解析[3] やクラスフロー解析[8] によって求める．

I-call if 変換の利点はメソッド呼び出しを直接呼び出しで実行することにある．直接呼び出しはインライン展開で高速化できる．間接呼び出しでメソッドを呼び出すディスパッチ表法にはインライン展開を適用できない．

なお，図 3 のコードでは `obj` のクラスを鍵として呼び出し対象のメソッドを検索するが，検索の鍵にはクラスの代わりにメソッドを使うこともできる[4]．検索の鍵にクラスを使う I-call if 変換をクラスチェック変換と呼び，メソッドを使うものをメソッドチェック変換と呼ぶことにする．また，図 3 のコードでは `obj` のクラスが C_1 でも C_2 でもないとき間接呼び出しで仮想メソッド呼び出しを実行するが，静的解析によって `obj` がとりうるクラスが必ず C_1 か C_2 のいずれかであると特定できる場合もある．その場合は，間接呼び出しを省略し，if 文を 1 段節約できる．間接呼び出しを省略した I-call if 変換を限定的であるということにする．

I-call if 変換は，検索の鍵にクラスとメソッドのどちらを使うか，また，限定的か否かという点から 4 種類に分類できる．図 2 のメソッド呼び出し `obj.`m`()` を 4 種類の I-call if 変換で最適化する際，JeanPaul の静的コンパイラが生成するコードの例と，呼び出し元のメソッド $n()$ の仮定クラスを表す集合 S_n と配列 N_n に追加する要素を**図 4** に示す．

図 4 のコード中にある JeanPaul 固有の部分について述べる．クラスチェック変換のコード中にある D_{C_k} はクラス C_k を表す構造体で，静的コンパイラが生成する．JeanPaul の実行系は，動的ロードしたクラス C_k とその全親クラスが仮定クラスと一致する場合に限り，クラス C_k を表す構造体として，構造体 D_{C_k} を利用する．一致しない場合には，クラス C_k を表す構造体を動的に確保した別の空間に配置する．したがって，式 `class == &`D_{C_k} の値は，`class` が C_k であり，なおかつ動的ロードしたクラス C_k とその全親クラスが仮定クラスと一致する場合に限り真になる．

```
class = obj->class;
if (class == C1){
  // クラスが C1 なら C1 の m() を直接呼出し
  C1 :: m(obj);
} else if (class == C2){
  // クラスが C2 なら C2 の m() を直接呼出し
  C2 :: m(obj);
}else{
  // どちらでもなければ間接呼出し
  code = obj->dispatch_table[Om];
  code(obj);
}
```

図 3 I-call if 変換の例
Fig. 3 An example of I-call if conversion.

また，$C :: m$ はクラス C に対応するメソッド $m()$ の静的コンパイル済みコードを参照するものとする．したがって，直接呼び出し $C :: m()$ は静的コンパイラが生成したコードを直接呼び出す．実行時には，静的コンパイラが生成したコードが使用可能になるまでの間，JIT コンパイラが生成したコードを使ってプログラムを実行するが，直接呼び出し $C :: m()$ が JIT コンパイラが生成したコードを呼ぶことはない．また，式 `code == &(`$C :: m$`)` の値は，`code` が C に対応するメソッド $m()$ の静的コンパイル済みコードのアドレスを参照するときに限り真になる．

図 4 では，`obj` がとりうるクラスを $\{C_1, \cdots, C_k\}$ とし，クラスチェック変換の比較候補とした．メソッドチェック変換の比較候補 $\{C'_1 :: m, \cdots, C'_l :: m\}$ は，メソッド $C_1 :: m, \cdots, C_k :: m$ から重複要素を除いたものである．

3.1 追加する仮定クラスの違い

仮想メソッド呼び出しを 4 種類の I-call if 変換のどれを使って最適化するか検討するにあたり，静的コンパイラ固有に考慮すべき問題に，仮定クラスへの影響がある．図 4 に示すように，最適化を施した場合に追加する仮定クラスは，I-call if 変換の種類によって異なる．一般に仮定クラスを追加するほど，実行時にメソッドの静的コンパイル済みコードが使用可能になる時期が遅れ，次の問題が起きやすくなる．

名称	非限定的クラスチェック変換	限定的クラスチェック変換

静的コンパイラ（JeanPaul）が生成するコード	class = obj->class;
	if (class == &D_{C_1}){
	  $C_1 :: m$(obj);
	}else if (class == &D_{C_1}){
	⋮
	}else if (class == &D_{C_k}){
	  $C_k :: m$(obj)
	}else{
	  code = obj->dispatch_table[O_m];
	  code(obj);
	}
		class = obj->class;
		if (class == &D_{C_1}){
		  $C_1 :: m$(obj);
		}else if (class == &D_{C_2}){
		⋮
		}else{
		  $C_k :: m$(obj);
		}

呼び出し元 n() への仮定クラスの追加	$S_n \leftarrow S_n \cup P(C_0)$ $N_n \leftarrow N_n + [C_1 :: m, C_2 :: m, \cdots C_k :: m]$	$S_n \leftarrow S_n \cup P(C_1) \cup \cdots \cup P(C_k)$ $N_n \leftarrow N_n + [C_1 :: m, C_2 :: m, \cdots C_k :: m]$
名称	非限定的メソッドチェック変換	限定的メソッドチェック変換

静的コンパイラ（JeanPaul）が生成するコード	code = obj->dispatch_table[O_m];
	if (code == &($C'_1 :: m$)){
	  $C'_1 :: m$(obj);
	}else if (code == &($C'_2 :: m$)){
	⋮
	}else if (code == &($C'_l :: m$)){
	  $C'_l :: m$(obj);
	}else{
	  code(obj);
	}
		code = obj->dispatch_table[O_m];
		if (code == &($C'_1 :: m$)){
		  $C'_1 :: m$(obj);
		}else if (code == &($C'_2 :: m$)){
		⋮
		}else{
		  $C'_l :: m$(obj);
		}

呼び出し元 n() への仮定クラスの追加	$S_n \leftarrow S_n \cup P(C_0)$	$S_n \leftarrow S_n \cup P(C_0) \cup \cdots \cup P(C_k)$ $N_n \leftarrow N_n + [C'_1 :: m, C'_2 :: m, \cdots C'_l :: m]$

※メソッド m() はクラス C_0 が宣言するものとする．

図 4 I-call if 変換の分類
Fig. 4 Classification of I-call if conversions.

(1) アプリケーションの実行速度を改善できない．
(2) JIT コンパイラを使用頻度が高くなり，静的コンパイラの利点の 1 つである，実行時コンパイルの不要化によるアプリケーション起動オーバヘッドの軽減を達成できない．

追加する仮定クラスが最も少ない I-call if 変換は非限定的メソッドチェック変換である．非限定的メソッドチェック変換では変換結果のコード自身が，呼び出し対象のメソッドのアドレスを比較して静的コンパイル済みメソッドが使用可能か（ディスパッチテーブルに登録済みか）確認する．したがって，この変換を施したコードについては，実行系に，直接呼び出し対象のメソッド m() の静的コンパイル済みコードが使用可能になった後でリンクするよう指示する（すなわち m() を配列 N_n に追加する）必要がない．このため，非限定的メソッドチェック変換では，ディスパッチ表のオフセット計算に必要な要素のみ追加すればよい．

限定的メソッドチェック変換でも呼び出し対象のメソッドのアドレスを比較するが，最後の 1 候補について比較を省略するので，直接呼び出し対象のメソッドを配列 N_n に追加する必要がある．さもないと，たとえば図 4 のコードにおいて，obj のクラスが C'_1 であり，静的コンパイル済みコード $C'_1 :: m$ がまだ使用可能でないとき，本来は JIT コンパイラが生成したコードを呼び出すべきところを，$C'_l :: m$ を呼び出してしまい，プログラムの実行がおかしくなる．

限定的 I-call if 変換ではさらに，集合 S_n に呼び出し対象のインスタンス obj の候補クラス $\{C_1, C_2, \cdots, C_k\}$ とそれらの全親クラスを追加する．この追加もまた，最後の 1 候補について比較を省略するために必要なもので，仮定クラスと一致しないクラスのために，誤ったメソッドを呼び出すことを避ける．非限定的 I-call if 変換では，仮定クラスと一致しないクラスが発生するとディスパッチ表法で仮想メソッド呼び出しをするので，誤ったメソッドを呼び出すことはない．したがって，この追加は不要である．

3.2 仮定クラスの追加による影響の評価

静的コンパイラでは，仮定クラスの追加による影響とコードの実行速度の双方を考えて 4 種類の I-call if 変換を使い分ける．使分けの基準はベンチマークなどの評価結果から定める．ここでは，使分けの基準を定めるために実施したいくつかの評価結果を示す．

```
// クラスチェック変換
LDW         4(0,%r25),%r29              ;%r29 ← obj->class
LDW         T'D_C1(0,%r19),%r31         ;%r31 ← &D_C1
COMBF,=,n   %r29,%r31,NEXT_CANDIDATE    ;if (%r29 != %r31) goto NEXT_CANDIDATE
// メソッドチェック変換
LDW         4(0,%r25),%r22              ;%r22 ← obj->dispatch_table
LDW         -52(0,%r22),%r1             ;%r1 ← dispatch_table[O_m]
LDW         T'$sdp(0,%r19),%r31         ;%r31 ← シンボル表のアドレス
LDW         C'_1 :: m(0,%r31),%r20      ;%r1 ← &(C'_1 :: m)
COMBF,=,n   %r1,%r20,NEXT_CANDIDATE     ;if (%r1 != %r20) goto NEXT_CANDIDATE
```

図 5 クラスおよびメソッドチェック変換のコード
Fig. 5 Code for class- and method- check conversion.

本論文では実行速度の評価に次の共通の環境を利用する．評価対象は SPECjvm98[2] とする．これは javac など中〜小規模の実用的アプリケーションを構成要素とするベンチマークである．SPECjvm98 の問題サイズは 100 とし，_201_compress と _213_javac の 2 項目についてはヒープ不足回避のためヒープ初期サイズ，上限サイズとも 128 Mbyte に指定する．実験機械には HITACHI3500/540 MP（CPU: PA7200 100 MHz，メモリ: 256 MByte，OS: HI-UX/WE2）を用い，Java 実行環境には JeanPaul を用いる．本論文で述べる最適化は JeanPaul の Java 向け静的コンパイラ上に実装する．静的コンパイル対象のクラスは，SPECjvm98 の各ベンチマークを -verbose オプション☆付きで実行した結果から求めた，ベンチマークの終了までにロードした全クラスとする．

3.2.1 クラスチェック vs メソッドチェック

メソッドの候補数 l とクラスの候補数 k については，一般に $l \leq k$ が成り立つ．すなわち，メソッドチェック変換の方がクラスチェック変換より分岐回数が少なくなりうる．なぜなら，メソッドチェック変換では，メソッドの定義を共有する複数のクラスを 1 回の比較で特定できるためである．

分岐回数が少なくなりうるならば，すべての仮想メソッド呼び出しをメソッドチェック変換で最適化すればよいかというと，必ずしもそうでない．メソッドチェック変換では，最初に呼び出し先のメソッドのアドレスを取得するためにディスパッチテーブルを参照するが，この参照はクラスチェック変換では不要であり，その結果クラスチェック変換の方が高速な場合もある．クラスチェック変換とメソッドチェック変換のそれぞれについて，静的コンパイラが生成する，最初の分岐に至るまでのコードを**図 5** に示す．図 5 のコードを比較すると，クラスチェック変換の方が，最初の分岐に至るまでに実行する命令が，ロード命令 2 回分少ないことが分かる．Detlefs はすべての仮想メソッド呼び出しをメソッドチェック変換で最適化するより，クラスチェック変換とメソッドチェック変換を使い分ける方が高速化できると指摘している[4]．

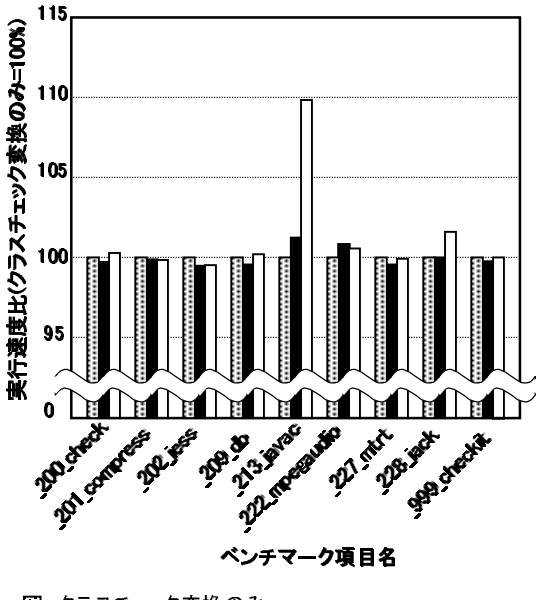

図 6 実行速度によるクラスおよびメソッドチェック変換の比較
Fig. 6 Comparison of class- and method- check conversions by performance.

ただし，静的コンパイラでは純粋なコードの実行速度のほかに，仮定クラスの追加による影響を考慮する必要がある．仮想メソッド呼び出しを静的コンパイラで次に示す 3 つの方針で最適化した場合の実行速度の比較を**図 6** に示す．

(1) クラスチェック変換のみ
(2) クラスチェック変換とメソッドチェック変換の

☆ どのクラスをロードしたかログを出力するオプション

表 1 静的コンパイル済みコードのみによる実行時間
Table 1 Execution time solely by statically compiled codes.

ベンチマーク項目名	最適化方針	静的コンパイル済みコードの事前リンク	
		あり	なし
_213_javac	C	334.983	292.167
	H	330.973	292.443
	M	304.990	292.664
_228_jack	C	317.040	309.858
	H	316.932	310.213
	M	312.038	309.875

※数値は実行時間，単位は秒
※最適化方針の略号 C，H，M は次の最適化方針を表す．
C:クラスチェック変換のみ
H:クラスチェック変換とメソッドチェック変換の使分け
M:メソッドチェック変換のみ

使い分け

(3) メソッドチェック変換のみ

ここでクラスチェック変換とメソッドチェック変換を使い分ける基準として，メソッドとクラスの候補数が同一の場合にはクラスチェック変換を，異なる場合にはメソッドチェック変換を用いた．If 文におけるクラスやメソッドの候補数と比較順序は，比較順序の違いによる速度差が生じることを防ぐために同一とし☆，たとえばクラスチェック変換で C_A，C_B の順で比較する場合，メソッドチェック変換では $C_A :: m$，$C_B :: m$ の順で比較した．限定的 I-call if 変換と非限定的 I-call if 変換の使分けについては，可能な場合にはすべて限定的 I-call if 変換を施した．

図 6 から，_213_javac と _228_jack の 2 項目で実行速度に差が出ていることが分かる．これらの 2 項目ではともに，メソッドチェック変換のみで最適化すると実行速度が最も速くなる．最も遅いクラスチェック変換のみで最適化した場合と比較して，_213_javac では 9.83%，_228_jack では 1.60%高速になっている．メソッドチェック変換のみによる最適化では，クラスチェック変換とメソッドチェック変換を使い分ける最適化よりロード命令を多用する．それにもかかわらずメソッドチェック変換のみの方が実行速度が速くなる原因は，追加する仮定クラスの違いにあると考える．すなわち，非限定的クラスチェック変換と非限定的メソッドチェック変換では，非限定的メソッドチェック変換の方が追加する仮定クラス数が少なく，より早い時期に静的コンパイル済みコードをリンクしてプログラムの実行を高速化できる．リンク時期の違いを除去して静的コンパイル済みコードの実行速度を比較するため，全メソッドについて静的コンパイル済みコードをあらかじめリンクしてから _213_javac と _228_jack を実行した結果を**表 1** に示す☆☆．表 1 から，静的コンパイル済みコードの実行速度はほぼ互角といえる．クラスチェック変換を使う方が，ロード命令の実行回数が少ない分だけ速くなる傾向は特に観測できなかった．これらのことから，図 6 の実行速度の差は静的コンパイル済みコードのリンク時期の違いから生じたと考える．

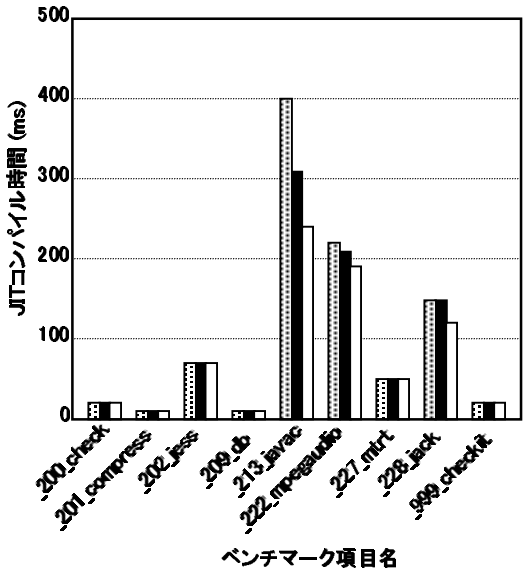

図 7 JIT コンパイル時間によるクラスおよびメソッドチェック変換の比較
Fig. 7 Comparison of class- and method- check conversions by JIT compilation time.

追加する仮定クラスの違いは JIT コンパイル時間にも影響する（**図 7**）．JIT コンパイル時間とは，ベンチマークの実行中に JIT コンパイルに費やした時間を表す．JeanPaul では，呼び出し対象のメソッドについて静的コンパイル済みコードがリンク済みでなければ，JIT コンパイルしてネイティブコードを生成する．静的コンパイル済みコードをリンクする時期は

☆ 比較順序は，候補メソッドを定義するクラスのロード順とする．I-call if 変換は候補メソッド数が 3 以下の場合に適用する．クラスチェック変換では候補メソッドに対応する候補クラスから 1 つを選んで比較対象とする．クラスの選択基準は，メソッドを定義するクラスが抽象クラスでなければ定義するクラスとし，抽象クラスならば候補クラスのうち一番先にロードしたものとする．

☆☆ 静的コンパイル済みコードの事前リンクは，静的コンパイル対象の全クラスをベンチマークの実行開始前にロード・初期化することで実施した．

表 2 仮想メソッド呼び出しにおける誤分岐回数の分布
Table 2 Distribution of misprediction count in virtual method calls.

ベンチマーク 項目名	最適化 方針	実行時間 (秒)	誤分岐回数 0	1	2 以上	合計
_200_check	C	0.426	3,388(83.8%)	579(14.3%)	74(1.8%)	4,041
	H	0.427	3,542(87.7%)	425(10.5%)	74(1.8%)	4,041
	M	0.424	3,691(87.7%)	439(10.4%)	80(1.9%)	4,210
_201_compress	C	247.250	1,620(70.6%)	571(24.9%)	102(4.4%)	2,293
	H	247.730	1,730(75.4%)	461(20.1%)	102(4.4%)	2,293
	M	247.638	1,730(75.4%)	461(20.1%)	102(4.4%)	2,293
_202_jess	C	392.862	20,123,738(74.8%)	6,771,720(25.2%)	294(0.0%)	26,895,752
	H	395.106	26,892,016(100.0%)	3,442(0.0%)	294(0.0%)	26,895,752
	M	394.909	26,892,294(100.0%)	3,452(0.0%)	294(0.0%)	26,896,040
_209_db	C	734.172	14,946,416(100.0%)	3,786(0.0%)	594(0.0%)	14,950,796
	H	737.286	14,947,417(100.0%)	2,785(0.0%)	594(0.0%)	14,950,796
	M	732.726	14,947,418(100.0%)	2,786(0.0%)	594(0.0%)	14,950,798
_213_javac	C	334.983	14,834,421(66.5%)	6,211,718(27.8%)	1,268,556(5.7%)	22,314,695
	H	330.973	17,797,120(77.2%)	5,223,730(22.7%)	39,553(0.2%)	23,060,403
	M	304.990	21,225,152(79.3%)	5,486,978(20.5%)	42,511(0.2%)	26,754,641
_222_mpegaudio	C	277.168	440,265(12.4%)	3,101,643(87.0%)	21,431(0.6%)	3,563,339
	H	274.814	3,493,138(98.0%)	70,196(2.0%)	16(0.0%)	3,563,350
	M	275.558	3,493,139(98.0%)	70,196(2.0%)	17(0.0%)	3,563,352
_227_mtrt	C	402.205	168,541,406(97.5%)	2,875,063(1.7%)	1,390,708(0.8%)	172,807,177
	H	404.065	171,642,879(99.3%)	1,164,040(0.7%)	258(0.0%)	172,807,177
	M	402.533	171,642,886(99.3%)	1,164,042(0.7%)	258(0.0%)	172,807,186
_228_jack	C	317.040	10,389,602(94.0%)	653,960(5.9%)	8,840(0.1%)	11,052,402
	H	316.932	10,548,147(95.4%)	495,415(4.5%)	8,840(0.1%)	11,052,402
	M	312.038	10,665,710(94.8%)	572,026(5.1%)	8,840(0.1%)	11,246,576
_999_checkit	C	28.616	238,256(80.6%)	653,960(18.6%)	2,246(0.8%)	295,462
	H	28.645	277,501(93.9%)	15,717(5.3%)	2,246(0.8%)	295,462
	M	28.610	277,501(93.9%)	15,715(5.3%)	2,246(0.8%)	295,462

※最適化方針の略号 C，H，M の意味は表 1 と共通.

仮定クラス数が少ないほど早く，したがって仮定クラス数が少ないほど JIT コンパイル時間が短く，アプリケーションの起動オーバヘッドが小さくなる．図 7 では，追加する仮定クラス数が最も多いクラスチェック変換のみによる最適化に比べ，最も少ないメソッドチェック変換のみによる最適化では，_213_javac で 40%，_228_jack で 20%，JIT コンパイル時間が短くなっている．

_213_javac と _228_jack 以外の項目で実行速度に差が出ない原因を検討するための資料として，**表 2** に分岐を必要とする仮想メソッド呼び出しの実行回数を，実行時に誤分岐した回数ごとに分類した結果を示す．ここで分岐を必要とする仮想メソッド呼び出しとは，I-call if 変換により 1 段以上の if 文を含む呼び出し文に変換したものを示す．final メソッド呼び出しなど分岐を必要としないものについては，クラスチェック変換とメソッドチェック変換で変換結果が同一なので検討対象から除外する．また，誤分岐回数とは，I-call if 変換後の仮想メソッド呼び出しのコードを実行する過程で通過する if 文において，呼び出し対象のオブジェクトのクラスあるいはメソッドを比較した結果が偽となる回数を表す．

表 2 について，分岐を必要とする仮想メソッド呼び出しの実行回数が最も多く，したがって I-call if 変換の影響を最も受けやすい _227_mtrt に注目して考える．_227_mtrt では，各最適化方針間で誤分岐の比率に大きな差がない．また，実行回数の合計欄の値や，図 7 の JIT コンパイル時間がほぼ同一なことから，静的コンパイル済みコードのリンク時期にも差がないといえる．これらが _227_mtrt で実行速度に差が生じないことの原因だと考える．なお，静的コンパイル済みコードのリンク時期が同一ならば，クラスチェック変換の方がロード命令の実行回数が少ない分だけ実行速度が速くなるはずだが，その傾向は観測できなかった．

誤分岐が実行速度に与える影響について，_227_mtrt について行った実験から考える．表 2 から，_227_mtrt をクラスチェック変換のみで最適化すると，1 度も誤分岐しない確率が 97.5%になることが分かる．実験のため，クラスチェック変換における比較候補クラスの選択アルゴリズムを変更し，これを 66.4% として差

分の 31.1%（67,633,955 回）を 1 回誤分岐した後に間接呼び出しで実行するようにしたところ，実行時間が 9.02 秒余計にかかった．表 2 から，_202_jess をクラスチェック変換のみで最適化すると，1 回誤分岐する仮想メソッド呼び出しが増えることが分かるが，増加数は 6,768,300 回程度であり，実行時間への影響は $9.02 \times \frac{6,768,300}{67,633,955} \simeq 0.90$ 秒程度と見積もることができる．0.90 秒は_202_jess の全実行所要時間の 0.23%と小さな差にすぎない．また，表 2 から静的コンパイル済みコードのリンク時期にも差がないことが分かり，これらの結果として_202_jess では各最適化方針間で実行速度に差が出なかったと考える．_200_check など残りの項目については，分岐を必要とする仮想メソッド呼び出しの実行回数自体が少ないために，各最適化方針間で実行速度に差が出なかったと考える．

結論としては，クラスチェック変換とメソッドチェック変換の比較では，メソッドチェック変換を用いる方が静的コンパイル済みコードのリンク時期を早めることでプログラムの実行速度を高速化でき，それ以外の点では大きな差が出ないといえる．

3.2.2 限定的 vs 非限定的

限定的 I-call if 変換と非限定的 I-call if 変換に関しては，コードの実行速度については，比較段数が少ない限定的 I-call if 変換が勝る．反面，リンクの遅延（追加する仮想クラス数）は，非限定的 I-call if 変換の方が少ない．これらの要素が実行速度に及ぼす影響を評価するため，次に示す 3 つの方針で最適化した場合の実行速度を測定した．測定結果を**図 8** に示す．

(1) 非限定的 I-call if 変換のみを適用する．
(2) final 宣言した仮想メソッドの呼び出しについてのみ限定的 I-call if 変換を適用する．
(3) （クラスフロー解析などによって）限定的 I-call if 変換を適用可能と分かった箇所にはすべて限定的 I-call if 変換を適用する．

最適化方針 (3) は限定的 I-call if 変換を最も多く適用するため，コードの実行速度は速くなるが追加する仮定クラスも多い．なお，メソッドチェック変換とクラスチェック変換の使い分けについては，すべての最適化方針でメソッドチェック変換のみを使用した．

図 8 から，おおむね final 宣言した仮想メソッドには限定的 I-call if 変換を適用する方が実行速度が速くなる傾向があることが分かる．final 宣言した仮想メソッドに限定的 I-call if 変換を適用すると，すべての仮想メソッド呼び出しに非限定的 I-call if 変換を適用する場合に比べ，_201_compress で 12%，他に 4 つの項目で 4〜5% 実行が速くなる．final 宣言した仮想メソッドに限定的 I-call if 変換を適用すると実行速度を改善できる理由を次に示す．

- final 宣言した仮想メソッドでは，限定的 I-call if 変換を施すと単なる直接呼び出しになり，if 文が 1 段もなくなる．このため，たとえば非限定的 I-call if 変換で 3 段の if 文を限定化して 2 段にする場合より，コードの実行速度が大きく改善する．
- final 宣言した仮想メソッドは呼び出し箇所数が多く，仮想メソッド呼び出し箇所数全体の 38〜68% を占める（**表 3**）．

final 宣言した仮想メソッド呼び出しにのみ限定的 I-call if 変換を適用する場合と，可能な限り限定的 I-call if 変換を適用する場合の実行速度差は微妙だが，

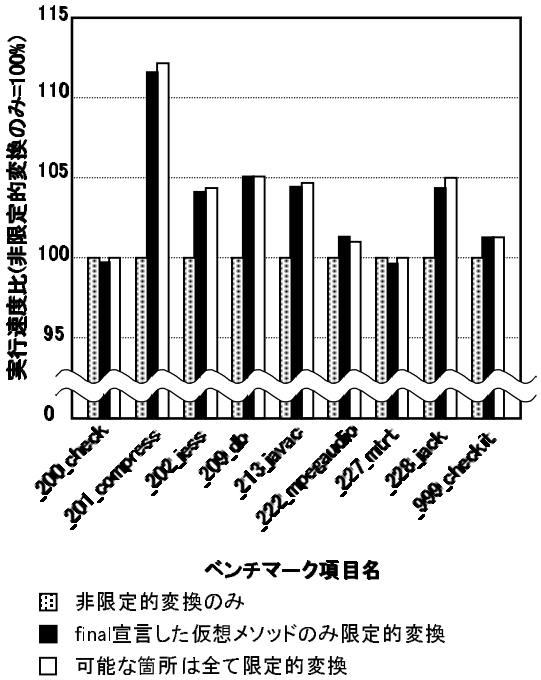

図 8 限定的 I-call if 変換と非限定的 I-call if 変換の比較
Fig. 8 Comparison of limited I-call if conversion and unlimited one by performance.

表 3 final 宣言した仮想メソッドの呼び出し箇所数
Table 3 Static counts of call-sites for final methods.

ベンチマーク項目名	総呼び出し箇所数	final メソッド呼び出し箇所数
_200_check	1,490	842(57%)
_201_compress	1,157	711(61%)
_202_jess	3,317	2,128(64%)
_209_db	1,341	780(58%)
_213_javac	5,620	3,203(57%)
_222_mpegaudio	1,615	1,106(68%)
_227_mtrt	2,129	806(38%)
_228_jack	2,268	1,242(55%)
_999_checkit	1,490	838(60%)

```
    LDWS        4(0,%r26),%r25              ;%r25 ← obj->dispatch_table
    LDWS        -52(0,%r25),%r26            ;%r26 ← dispatch_table[O_m]
    LDIL        LP'C'_1 :: m,%r24           ;%r24 ← &(C'_1 :: m) の上位 bit
    LDO         RP'C'_1 :: m(%r24),%r31     ;%r31 ← &(C'_1 :: m)，即ち%r24 + &(C'_1 :: m) の下位 bit
    EXTRU,=     %r31,31,1,%r23              ;if ((%r23 ← %r31 & 0x80000000) == 0) 次命令を無視
    LDW         -4(0,%r27),%r23             ;（%r31 が関数シンボルなら）%r23 ← オフセット
    COPY        %r26,%r21                   ;%r21 ← %r26
    ADD         %r31,%r23,%r23              ;%r23 ← %r31 + %r23
    BB,>=,N     %r26,30,L1                  ;if (%r26 が共有ライブラリ上にない) goto L1
    BL          $$sh_func_adrs,%r31         ; 関数呼び出し：%r29 ← 補正済みアドレス
    NOP
    COPY        %r29,%r21                   ;%r21 ← %r29
L1
    COPY        %r23,%r29                   ;%r29 ← %r23
    BB,>=,N     %r23,30,L2                  ;if (%r23 が共有ライブラリ上にない) goto L2
    COPY        %r23,%r26                   ; 引数のセット：%r26 ← %r23
    BL          $$sh_func_adrs,%r31         ; 関数呼び出し：%r29 ← 補正済みアドレス
    NOP
L2
    COMBF,=,N   %r21,%r29,NEXT_CANDIDATE    ;if (%r21 != %r29) goto NEXT_CANDIDATE
```

図 9 メソッドチェック変換のコード（その 2）
Fig. 9 Code for method-check conversion (2).

わずかに後者が勝る．したがって，限定的 I-call if 変換と非限定的 I-call if 変換については，可能な限り限定的 I-call if 変換を施す方が実行速度が向上し，リンクの遅延がもたらす影響は小さいといえる．

3.3 メソッドチェック変換と C コンパイラの問題

SPECjvm98 による評価結果から Java 向け静的コンパイラではメソッドチェック変換と限定的 I-call if 変換を積極的に適用すると実行速度が速くなる傾向があることが分かった．ただ，メソッドチェック変換は実装によっては実行速度を改善できない場合もある．JeanPaul の静的コンパイラは Java2C トランスレータと C コンパイラからなり，Java2C トランスレータがバイトコードの仮想メソッド呼び出し命令にメソッドチェック変換を施して図 4 の C コードに変換し，それを C コンパイラがさらに図 5 のコードに変換する．C コンパイラにはいろいろな製品があり，同じ OS（HI-UX/WE2）向けの C コンパイラでも，図 4 のメソッドチェック変換の C コードを**図 9** のコードに変換するものもある（このコンパイラは，クラスチェック変換の C コードは図 5 と同一のコードに変換する）．図 9 のコードは，図 5 のコードが 5 命令で実行する処理に最低 12 命令を必要とし，明らかに実行速度が劣る．したがって，図 9 のコードを出力する C コンパイラを利用する場合，メソッドチェック変換を使うと実行速度が遅くなる☆．

Java2C トランスレータは移植性の高い Java 向け静的コンパイラを実現する手段の 1 つだが，移植にあたっては，この問題が示すように，組み合わせる C コンパイラの性質を考慮して使用する最適化を定める（クラスチェック変換とメソッドチェック変換のどちらを使うか定める）と，より実行速度を改善できる．

4. 関連研究

オブジェクト指向プログラミング言語向けコンパイラにおいて，クラスチェック変換[1] が古くからなじみのある最適化技法であるのに対し，メソッドチェック変換[4] は比較的新しい．これはメソッドチェック変換が，Java のように，ディスパッチ表法などで高速にメソッドのアドレスを検索できるプログラミング言語にしか適用できないためであろう．本論文では 4 種類の I-call if 変換を Java 向け静的コンパイラで適用する場合の得失を具体的評価をもって示したが，このような評価はこれまでに知られていない．

実行時にクラスを検査する動作（クラス検査）に着目し，プログラムの高速化を試みた論文はこれまでにも存在する．Hölzle らはクラス検査と組み合わせて実施する最適化について論じており[6]，その動的ロード存在化における適用については Sreedhar らが述べている[9]．これらの論文はともに最適化によってプログラムの高速化を図るが，本論文では静的コンパイル済

☆ 図 9 と図 5 のコードが異なる原因は，関数アドレスの取得方法の違いにある．図 9 は初期の共有ライブラリの実装に対応するコードだが，実装の改善の結果，図 5 のコードで効率的に関数アドレスを取得可能になった．共有ライブラリの実装が洗練されている近代的な OS とコンパイラでは多くの場合，図 5 に準じたコードを得られ，メソッドチェック変換が有効に働く．しかし，古い OS やコンパイラを使う場合には注意が必要である．

みコードのリンク時期を早めることでプログラムを高速化する方法を示した．

JIT コンパイラを利用する Java 実行環境には，最初はインタプリタで実行し，実行頻度の高いメソッドのみコンパイルするものがある．このような Java 実行環境では JeanPaul 同様に，コンパイル済みコードをリンクする時期が遅くなるとプログラムを高速化しにくい．Suganuma らはこの問題に対処する手段として，インタプリタで実行を開始したメソッドについて，実行途中から JIT コンパイラが生成したコードに遷移する方法を提案している[10]．この手法によれば，次の呼び出しを待たずにコンパイル済みコードを使用してプログラムを高速化できる．現在の JeanPaul にこの機能はないが，今後の検討課題としたい．

5. 結　　論

Java 向け静的コンパイラでは，I-call if 変換を実施するにあたり，コードの実行速度のみならず，最適化が追加する仮定クラスに注意をはらう必要があることを指摘した．SPECjvm98 による評価結果から Java 向け静的コンパイラでは，メソッドチェック変換と限定的 I-call if 変換を積極的に適用すると実行速度が速くなる傾向があることが分かった．メソッドチェック変換のみで最適化すると，クラスチェック変換のみで最適化するより最大で 9.83%高速化できることが分かった．

参 考 文 献

1) Calder, B. and Grunwald, D.: Reducing Indirect Function Call Overhead In C++ Programs, *POPL 94*, pp.397–408 (1994).
2) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/ (1998).
3) Dean, J., Grove, D. and Chambers, C.: Optimization of Object-Oriented Programs Using Static Hierarchy Analysis, *ECOOP 95*, pp.77–101 (1995).
4) Detlefs, D. and Agesen, O.: Inlining of Virtual Methods, *ECOOP 99*, pp.259–278 (1999).
5) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison Wesley, Reading, Mass. (1996).
6) Hölzle, U. and Ungar, D.: Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedbak, *PLDI94*, pp.326–336 (1994).
7) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, Addison Wesley, Reading, Mass. (1996).
8) Pande, H.D. and Ryder, B.G.: Static Type Determination for C++, *USENIX 6th C++ Technical Conference*, pp.85–97 (1997).
9) Sreedhar, V.C., Bruke, M. and Choi, J.-D.: A Framework for Optimization in the Presence of Dynamic Class Loading, *PLDI 2000*, pp.196–207 (2000).
10) Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H. and Nakatani, T.: Overview of the IBM Java Just In Time Compiler, *IBM Systems Journal*, Vol.39, No.1, pp.175–193 (2000).
11) 小野寺民他：オブジェクト指向言語におけるメッセージ送信の高速化技法，情報処理，Vol.38, No.4, pp.301–310 (1997).
12) 千葉雄司：Java における静的コンパイル済みコードのリンク方法，情報処理学会論文誌プログラミング，Vol.42, No.SIG2(PRO9), pp.37–47 (2001).

(平成 12 年 10 月 25 日受付)
(平成 13 年 1 月 22 日採録)

千葉　雄司（正会員）

1972 年生．1997 年慶応義塾大学大学院理工学研究科計算機科学専攻修士課程修了．同年日立製作所（株）入社，システム開発研究所にてコンパイラの研究開発に従事．ソフトウエア科学会会員．