Javaにおける静的コンパイル済みコードのリンク方法

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．従来の Java 向け静的コンパイラは，動的ロードへの対応が不十分なため，コンパイル作業が煩雑になり，たとえば更新されうるクラスを手動で指示する必要があった．しかし，バイトコード形式で配布された Java アプリケーションを利用するユーザにとって，どのクラスが更新されうるか指示することは困難である．この問題の解決を目的として，Java 実行環境 JeanPaul を開発している．JeanPaul では，静的コンパイル済みコードをリンクする方法の工夫により，ユーザが更新されうるクラスを指示しなくても，Java アプリケーションを静的コンパイルし，なおかつ正常に実行できる．本論文では，JeanPaul におけるリンク方式について詳述する．また，JeanPaul のリンク方式が Java アプリケーションの性能に与える影響について考察する．

A Linking Method for Statically Compiled Code for Java

Yuji Chiba†

Traditional static Java compilers, that accelerate performance of Java applications, so far compelled users troublesome compilation process to support dynamic loading: for example, users had to manually choose the classes for static compilation which are not updated during runtime. To cope with this, a Java runtime environment JeanPaul has been developed, whose linking method for statically compiled code enables Java applications that are compiled without user's direction to support dynamic loading. This paper shows the implementation of the linking method for statically compiled code in JeanPaul and the effect that the linking method makes on the performance of Java applications.

1. はじめに

オブジェクト指向プログラミング言語 Java™,☆は，生産性の高さや機種非依存性などの利点を持つ一方で，実行速度が遅いなど問題をかかえている．Java は機種非依存性を実現するため，アプリケーションをバイトコードで表現するが，このことは次の 2 つの点で実行速度に悪影響を与える．

起動オーバヘッド C 言語や C++などで開発し，ネイティブコードにコンパイルしたアプリケーションは，コンピュータで直接実行できるが，バイトコードは直接実行できない．現在一般に用いられる Java 実行環境では，Just In Time（JIT）コンパイラ[4)]でバイトコードを動的にネイティブコードに変換してから実行する．しかし，変換には時間がかかり，アプリケーションの起動が遅くなる．

最適化レベル プログラムの実行を高速化するには，コンパイル時に最適化を施すのが一般的である．しかし，複雑な最適化には長い時間がかかる．動的コンパイラでは，最適化に長い時間を費やすとアプリケーションの起動が遅くなるので，適当なところで最適化作業を切り上げなければならない．これに対し，C 言語や C++では，プログラムを起動する前にあらかじめコンパイルしてしまう静的コンパイラが一般的である．静的コンパイラはプログラムの起動時間を気にせず，長い時間をかけて十分な最適化を実施できる．

Java アプリケーションを高速実行する手段として，バイトコード形式の Java アプリケーションを，C 言語や C++と同様に，静的にコンパイルしてネイティブコードに変換する方法がある．バイトコードは機種非依存性などを確保するための方便であり，ローカルにインストールするアプリケーションがバイトコード形式である必要はない．実際，Java アプリケーションを

† 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．

高速化するために，バイトコードをネイティブコードに変換する静的コンパイラが商品として存在する[10]．

Java 向け静的コンパイラには，Java アプリケーションを高速実行できる利点がある反面，単独では動的ロードを実現できないという問題もある．動的ロードは Java が提供する機能の 1 つであり，プログラムを構成するクラスファイルを実行時に読み込む．動的ロードを使うとプログラムの実行中にクラスファイルを生成して読み込むことが可能になるが，実行中に生成する，すなわち実行開始前に静的コンパイルできないクラスファイル中のメソッドを実行するには，JIT コンパイラやインタプリタが必要になる．

従来の Java 向け静的コンパイラには，動的ロードをサポートしないものもあるが，これでは Java の言語仕様[8),13)] との互換性が問題になる．静的コンパイラと JIT コンパイラあるいはインタプリタを組み合わせることで，動的ロードの機能を提供するシステムも存在する．しかし，静的コンパイラと JIT コンパイラを組み合わせたシステムでは，その実装によっては，動的ロードの機能を実現するためにコンパイル作業が複雑になることがある．

すなわち，Java の言語仕様は，アプリケーションを構成するすべてのクラスファイルを実行時に動的ロードし，その内容に従ってプログラムを実行するよう規定している．このとき静的コンパイラで考慮すべき問題は，静的コンパイル後にクラスファイルの更新が起きた場合，古いクラスファイルから作成した静的コンパイル済みコードが利用不能になることである．この問題の解決策の 1 つは，更新の対象になりうるクラスファイルを静的コンパイル対象から除外することである．更新の対象になりうるクラスファイルを静的コンパイル対象から除外すれば，利用不能になるコードは発生しない．除外したクラスについては，動的ロードしたクラスファイルを JIT コンパイラでコンパイルするかインタプリタで解釈実行すればよい．

ただし，更新の対象になりうるクラスを静的コンパイル対象から除外する方法には問題もある．すなわち，どのクラスが更新の対象となるかコンパイラが解析することは困難なので，更新の対象となるクラスをコンパイル対象から除外する作業は，コンパイラのユーザが実施しなくてはならない．このことは，バイトコード形式で配布した Java アプリケーションを，アプリケーションユーザが静的コンパイルすることを困難にする．なぜなら，アプリケーションユーザが Java アプリケーションの詳細な動作に関する情報（どのクラスが更新の対象になるか）を知ることは困難だからである．

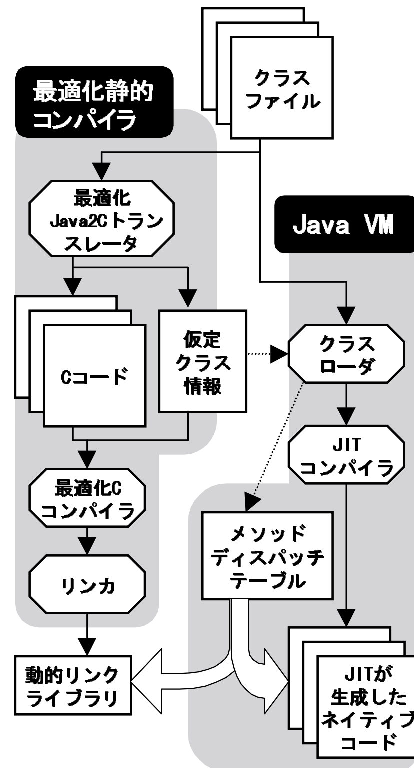

図 1 JeanPaul の構成
Fig. 1 Organization of JeanPaul.

そこで本研究では，動的ロードのサポートなど Java 言語仕様との互換性を確保し，なおかつバイトコード形式で配布された Java アプリケーションを，JIT コンパイラと同様に，アプリケーションユーザが実行するだけでコンパイルできる Java 向け静的コンパイラを目標とし，Java 実行環境 JeanPaul を開発している．JeanPaul では次の手順でアプリケーションを静的コンパイルする．まず，ユーザがアプリケーションをテスト実行する．テスト実行時には JIT コンパイラで生成したコードを使用する．テスト実行終了後，JeanPaul は静的コンパイラを起動して実行時に使用したクラス（のうち重要なもの）を静的コンパイルし，生成したコードを保存する．保存したコードは次にアプリケーションを実行するときから使用する．動的ロードについては，静的コンパイラと JIT コンパイラの併用によりサポートする（**図 1**）．

JeanPaul が従来の静的コンパイラと JIT コンパイラを組み合わせたシステムと異なる点は，静的コンパイル後にクラスファイルの更新が起きた場合への対処方法である．更新の対象になりうるクラスファイルを静的コンパイル対象から除外する方法は，除外する作業の自動化が困難なので，JeanPaul では利用しない．

その代わり JeanPaul では，静的コンパイル時にどのクラスファイルを参照したか記録しておく．そして実行時に，動的ロードした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か確認し，確認できた場合のみ静的コンパイル済みコードをリンクし，プログラムの実行に使用する．この方法によれば，テスト実行時にどのクラスを使用したという情報のみからアプリケーションを自動的に静的コンパイル可能になる．

動的ロードしたクラスファイルと静的コンパイル時に参照したクラスファイルの同一性を確認した後で静的コンパイル済みコードをリンクするシステムの実装は必ずしも単純でない．たとえばクラスファイル `foo.class` について同一性を確認できても，`foo.class` 内のすべてのメソッドについて静的コンパイル済みコードが使用可能になるとは限らない．これは静的コンパイル時にクラス間最適化☆を施すことによる．たとえば `foo.class` 内のメソッド `boo()` 内の呼出点に `woo.class` 内のメソッド `buz()` をインライン展開すると，`boo()` の静的コンパイル済みコードは `foo.class` だけでなく `woo.class` についても同一性を確認した後でないと使用可能にならない．

本論文の 2 章では，JeanPaul における，動的ロードしたクラスファイルと静的コンパイル時に参照したものの同一性を確認した後で静的コンパイル済みコードをリンクするシステムの実装について詳述する．さらに，3 章で同一性の確認が済むまで静的コンパイル済みコードのリンクを遅延することが実行速度に与える影響や，静的コンパイラと JIT コンパイラを組み合わせたシステムである JeanPaul が Java アプリケーションをどれだけ高速化できるか評価する．4 章では関連研究について述べる．5 章は結論である．

2. JeanPaul における静的コンパイル済みコードのリンク方法

メソッドの静的コンパイル済みコードが実行時に使用可能なるための条件は次の 2 つである．

(1) 静的コンパイル時に参照したすべてのクラスファイルの内容について，実行時に動的ロードしたものと内容が同一であると確認が済むこと．

(2) 静的コンパイル時に参照したクラスの一部について，初期化が完了すること．

条件 (1) はクラスファイルの更新への対処を目的とし，条件 (2) はクラス変数参照の高速化などの最適化を実施したとき発生する．クラス変数参照など初めてクラスを参照しうる動作については，実行する前に，まず，参照するクラスが初期化済みか検査し，未初期化であれば初期化する必要がある．これは，Java の言語仕様が，初めてクラスを参照するとき初期化するよう規定しているためである．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．この問題を解決するため，JeanPaul では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，クラス変数を定義するクラスの初期化が完了するまで遅延する．このため，条件 (2) が必要になる．

条件 (2) はクラスを初期化するまで満たされないが，Java の言語仕様上，クラスを初期化するタイミングをプログラムの実行開始時などに移すことはできない．したがって JeanPaul では，静的コンパイル済みコードはプログラムの実行当初から使用できるとは限らず，リンクを遅延しているメソッドを実行するためには JIT コンパイラが生成したコードを使う．

JeanPaul では，条件 (1) のクラスファイルの同一性を確認する作業についても，条件 (2) が満たされたか確認する作業と同じくクラスを初期化する際に，個々のクラスごとに実行することにした．同一性の確認作業については，Java の言語仕様がクラスファイルをロードするタイミングを規定していないため，プログラムの実行開始時に静的コンパイル対象の全クラスをロードし，一括して実行することもできる．しかし，次の 2 つの理由から JeanPaul では実行開始時に静的コンパイル対象の全クラスをロードすることはしない．

- 標準クラスライブラリ `java.lang.Class` が提供するメソッド `forName()` を使用するプログラムが，プログラマの意図にそぐわない挙動をしうる．`forName()` は引数に与えた名前を持つクラスを返戻するメソッドで，目的のクラスが未ロードならばロード，初期化のうえで返戻する．

  実行時にクラスファイル `foo.class` を生成し，その後に `forName()` でクラス `foo` を取得するプログラム P について考える．プログラム P のプログラマの意図は実行時に生成した `foo.class` から生成したクラスを取得することにあると考える．しかし，実行開始時に `foo.class` を含む静的コンパイル対象のクラスをロードすると，プログラム P において `forName()` は実行開始前に存在した `foo.class` から生成したクラスを返戻する．この挙動は Java の言語仕様的には問題ない

☆ 複数のクラスを参照する最適化をクラス間最適化と呼ぶ．

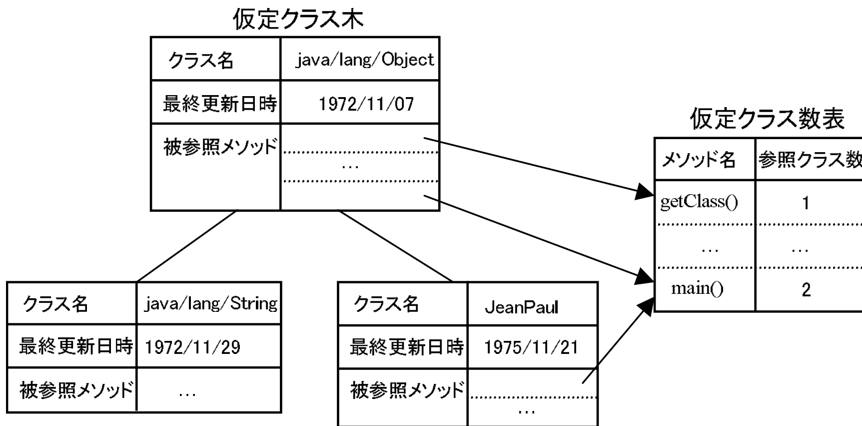

図 2 仮定クラス情報
Fig. 2 Supposed Class Information.

が，プログラマの意図にそぐわない．

調査した限りでは，本論文執筆時点で市場に存在し，動的ロードをサポートする Java 実行環境で，プログラム P についてプログラマの意図にそぐわない動作をするものはなかった．

- 実行開始時に静的コンパイル対象のクラスをすべてロードすると，ロード時の I/O オーバヘッドが実行開始時に集中し，起動オーバヘッドが増える．

JeanPaul における Java アプリケーションの静的コンパイルから実行に至る一連の流れの概要を具体的に示す．まず，テスト実行時に使用したクラスをコンパイル対象として，アプリケーションを静的コンパイルする．このとき，個々のメソッド $m()$ を静的コンパイルする際に参照したクラスの集合 S_m を仮定クラス情報というデータ構造に記録し，静的コンパイル済みコードとともに動的リンクライブラリに収めておく．次に，アプリケーションの実行にあたり，JVM（Java Virtual Machine，Java 仮想機械）は最初，JIT コンパイラを使ってプログラムをコンパイル，実行しつつ，プログラムを構成する個々のクラスを順次動的ロード，初期化する．初期化の際，動的ロードしたクラスファイルと静的コンパイル時に参照したものが同一か，仮定クラス情報を参照して確認する☆．

同一性の確認がとれた場合，それによって使用可能になった静的コンパイル済みコードがあるか仮定クラス情報を参照して調べる．使用可能な静的コンパイル済みメソッドの算出方法については 2.1 節で詳述するが，大まかには次のとおり．メソッド $m()$ の静的コンパイル時に参照したクラスの集合 S_m に関し，すべての $c \in S_m$ についてクラスファイルの同一性を確認し，初期化が完了したならば，メソッド $m()$ の静的コンパイル済みコードを使用可能と見なす．

静的コンパイル済みコードが使用可能になったメソッドについては，プログラムの実行に使用するコードを JIT コンパイラが生成したものから静的コンパイラが生成したものに切り替える．コードの切替えはメソッドディスパッチテーブルに登録してある個々のメソッドのアドレスを書き換えることで実施する．メソッドディスパッチテーブルは，プログラムがメソッドを呼び出す際に，呼出先のメソッドのアドレスを検索するために使う表であり，したがって，この表に登録してあるアドレスを静的コンパイル済みコードのアドレスに書き換えれば，次のメソッド呼出しからは静的コンパイル済みコードがプログラムの実行に使われる．

2.1 使用可能な静的コンパイル済みメソッドの算出

使用可能な静的コンパイル済みメソッドを探すには，仮定クラス情報の構成要素である仮定クラス数表と仮定クラス木を用いる（**図 2**）．仮定クラス数表は個々のメソッドを静的コンパイルした際に参照したクラスの数を収める表である．仮定クラス木は静的コンパイル時に参照したクラスを表すノードからなる木構造のデータ構造であり，個々のノードにはクラス間の継承関係を表すエッジと，クラスファイルに関する情報（ファイルの最終更新日時），参照メソッド表を収める．参照メソッド表は，静的コンパイル時にノードが表すクラスを参照した全メソッドの，仮定クラス数表におけるエントリへのポインタを保持する．使用可能な静的コンパイル済みメソッドを探す手順を次に示す．

(1) 動的ロードしたクラスに対応する仮定クラス木

☆ 現在のところ，ファイルの同一性の確認にはファイルの最終更新日時を利用している．

上のノードを求める☆.

（2） ノードに記録してあるクラスと動的ロードしたクラスが同一か調べる（更新日時を比較する).

（3） 同一ならば，ノードの参照メソッド表中のポインタが参照するすべての仮定クラス数表上のエントリについて，参照クラス数から 1 を引く．その結果，参照クラス数が 0 になったエントリが表すメソッドについては，静的コンパイル時に参照した全クラスについて，クラスファイルの同一性確認と初期化が完了しているので，静的コンパイル済みコードを使用可能と見なす.

2.2 仮定クラス情報の算出

仮定クラス数表と仮定クラス木は，静的コンパイラが次の手順により算出する.

（1） すべての静的コンパイル対象のメソッド $m()$ について，それぞれクラスの集合 S_m とメソッドの配列 N_m を用意する．S_m はメソッド $m()$ を静的コンパイルしたときに参照したクラス，すなわち実行時にメソッド $m()$ の静的コンパイル済みコードを使用する前に静的コンパイル時に参照したクラスファイルと動的ロードしたクラスファイルが同一か確認すべきクラスを収め，初期値は $\{C\}$ とする．ここで C はメソッド $m()$ を定義するクラスである.

配列 N_m は集合 S_m を求めるために一時的に用い，実行時にメソッド $m()$ より先に静的コンパイル済みコードが使用可能になる必要があるメソッド（メソッド $m()$ から静的コンパイル済みコードを直接呼び出すメソッド）を収める．メソッド $m()$ の中にメソッド $n()$ への呼出点が複数あるとき，その数だけ $n()$ を格納する必要があるため，N_m は集合でなく配列とする.

（2） すべてのメソッドをコンパイルする．メソッド $m()$ にクラス間最適化を施すとき，その内容に応じて集合 S_m，配列 N_m にクラスやメソッドを追加する．静的コンパイラが提供する最適化が集合 S_m，配列 N_m にどのようなクラスやメソッドを追加するかについては，2.3 節で述べる.

（3） すべてのメソッド $m()$ について，配列 N_m を集合に変換する．すなわち，配列 N_m について，配列中に同一の要素が複数存在するとき，1 つを残して残りは捨てる.

（4） すべてのメソッド $m()$ について，集合 S_m を，次の手順で求める.

（a） 集合 T を用意し，初期値を ϕ とする.

（b） 集合 N_m 中のメソッドがなくなるまで次の操作を繰り返す.

（i） 集合 N_m からメソッドを 1 つ取り出し，$n()$ とする.

（ii） $T \leftarrow T \cup \{n()\}$

（iii） $S_m \leftarrow S_m \cup S_n$

（iv） $N_m \leftarrow N_m \cup (N_n \cap T)$

（5） 求めた集合 S から仮定クラス数表と仮定クラス木を生成する．すなわち，すべてのメソッド $m()$ について，集合 S_m の要素数を仮定クラス数表のメソッド $m()$ の参照クラス数の欄に収め，集合 S_m が収めるすべてのクラスに対応する仮定クラス木のノードの参照メソッド表にメソッド $m()$ を登録する.

☆ ノードの検索はクラス名を鍵にハッシュ表を検索することで実施する．ハッシュ表は静的コンパイル時に仮定クラス情報の一部として生成する．Java では実行時に同名のクラスが，クラスローダごとに複数存在しうるので，一般には名前のみから対応するノードを取得できない．しかし，現在の JeanPaul ではシステムクラスローダがロードしたクラスのみ仮定クラス木上のノードと対応付けるので，クラス名のみで対応するノードを求めることができる．システムクラスローダ以外がロードしたクラスについては，対応するノードは存在しないことにしている．この結果，現在の JeanPaul ではシステムクラスローダがロードしたクラスしか静的コンパイル済みコードを利用できない.

表 1 JeanPaul の静的コンパイラが実施する最適化一覧
Table 1 Optimizations in the static compiler for JeanPaul.

クラス間最適化	メンバ，メソッドのオフセットの静的解決 クラス変数参照の高速化 クラスメソッドの直接呼出し クラスフロー解析を利用した ・冗長な CheckCast の削除 ・インスタンスメソッドおよびインタフェースメソッド呼出しの I-call if 変換[2] インライン展開
クラス内最適化	冗長な Null チェックの削除[1] 冗長な配列添字チェックの削除[12] コピー伝播 不要な定義点の削除 インタフェース呼出しの検索キャッシュ[18] 別名解析によるメンバ参照の集約

2.3 クラス間最適化と仮定クラス情報の計算

JeanPaul の静的コンパイラの最適化機能（**表 1**）のうち，クラス間最適化が仮定クラス情報の算出の過程で集合 S や配列 N にどのようなクラスやメソッドを追加するか示す.

2.3.1 メンバ変数，インスタンスメソッドのオフセットの静的解決

インスタンス変数参照（**図 3** 上段）を行うには，イ

```
// Java ソース
int val = obj.member;
```
```
// 最適化なし
offset = resolveInstanceField(
                        ClassName,
                        FieldName,
                        FieldDescriptor);
val = obj[offset];
```
```
// 最適化あり
val = obj[CONSTANT_MEMBER_OFFSET];
```

図 3 インスタンス変数参照
Fig. 3 Instance Variable Reference.

```
// Java ソース
int val = Class.member;
```
```
// 最適化なし
address = resolveStaticField(
                        ClassName,
                        FieldName,
                        FieldDescriptor);
val = *address;
```
```
// 最適化あり
val = *StaticFieldLinkTable[FIELD_OFFSET];
```

図 4 クラス変数参照
Fig. 4 Class Variable Reference.

ンスタンス変数のオフセット（インスタンス変数がインスタンスの何番目のフィールドに入っているか）を知る必要がある．Java ではインスタンス変数のオフセットは実行時に定まる．

静的コンパイラが生成しうるインスタンス変数参照のコードのうち，クラス間最適化なしで生成できるものを図 3 中段に示す．図 3 中段の関数 `resolveInstanceField()` はインスタンス変数のオフセットを検索する．クラスファイルの変更によりインスタンス変数が消失していた際には例外を発生する．

図 3 中段のコードでインスタンス変数参照を実現すると，インスタンス変数参照のたびに関数呼出しがおこり，プログラムの実行速度が低下する．インスタンス変数参照を高速化するにはオフセットを静的に計算し，図 3 下段のコードでインスタンス変数参照を実現する．しかし，オフセットを静的に計算した図 3 下段のコードは，実行時に定まったオフセットと静的コンパイル時に計算したオフセットが一致しないと正常に動作しない．そこで，オフセットの計算のために参照したすべてのクラス（インスタンス変数を定義するクラスとそのすべての親クラス）を集合 S に加える．

インスタンスメソッド呼出しについても，メソッドディスパッチテーブルにおけるオフセットに関して同様の問題が発生する．インスタンスメソッドのオフセットを静的に計算する最適化を施した場合，メンバ変数参照の場合と同様に，オフセットの計算のために参照したすべてのクラスを集合 S に加える．

2.3.2 クラス変数参照の高速化

Java ではクラス変数はつねに参照可能であると保証できない．たとえば，クラス変数を定義するクラスの内容を更新した結果，クラス変数がなくなる場合がある．Java の言語仕様は，クラス変数参照に失敗した場合，例外を発生するよう求めている．また，クラス変数参照時に，クラス変数を定義するクラスが未初期化であれば初期化する必要があるが，クラス変数参照のたびにクラスが未初期化か検査するのは冗長である．

静的コンパイラが生成しうるクラス変数参照のコードのうち，クラス間最適化なしで生成できるものを**図 4** 中段に示す．図 4 中段の関数 `resolveStaticField()` はクラス変数のアドレスを検索する．検索の過程で目的のクラス変数を定義するクラスが未初期化と判明した場合には初期化する．初期化に失敗した場合などには例外を発生する．

図 4 中段のコードではクラス変数を参照するたびに関数呼出しがおこり，プログラムの実行速度が低下する．クラス変数参照を高速実行するには，クラス間最適化を施した図 4 下段のコードを利用する．図 4 下段のコードは，動的リンクテーブル `StaticFieldLinkTable` を介してクラス変数を参照するコードで，動的リンクライブラリから外部モジュールの大域変数を参照する場合など，アドレスが静的に定まらない変数を参照するとき一般に利用するコードである．`StaticFieldLinkTable` は実行時にクラスを初期化し，クラス変数のアドレスが定まった時点で初期化する．図 4 下段のコードはクラス変数を定義するクラスを初期化したあとでないと正常に動作しないので，クラス参照変数を図 4 下段のコードに最適化した場合には集合 S にクラス変数を定義するクラスを加える．

2.3.3 クラスメソッドの直接呼出し

JeanPaul では，クラスメソッドについても，実行時にコードが JIT コンパイラが生成したコードから静的コンパイラが生成したコードにすりかわる．すりかわりうる（実行時にアドレスが変わりうる）メソッドを呼び出すため，クラスメソッド呼出しを間接呼出しで実現する方法もあるが，間接呼出しは直接呼出しよ

り実行時間がかかる．また，クラスメソッド呼出時には，クラスメソッドを定義するクラスが未初期化ならば初期化する必要があるが，クラスメソッド呼出しのたびにクラスが未初期化か検査するのは冗長である．

クラスメソッド呼出しは，静的コンパイラが生成したコードへの直接呼出しとして実現すれば高速化できる．しかし，この最適化を施すと，呼出し元のコードは呼出し先のメソッドの静的コンパイル済みコードをリンクしたあとでリンクしないと正常に動作しなくなる．そこで，この最適化を実施した場合，配列 N に呼出先のクラスメソッドを追加する．

2.3.4 インスタンスメソッドおよびインタフェースメソッド呼出しの I-call if 変換

インスタンスメソッド呼出しは間接呼出しでも実現できるが，間接呼出しは実行に時間がかかり，また最も有効な最適化の1つであるインライン展開を適用できない．インスタンスメソッド呼出しをより高速に実現する方法に I-call if 変換[2] がある．I-call if 変換では，インスタンスメソッド呼出しを次の手順で実行する（**図 5**）．まず，呼出し対象のオブジェクトのクラスをクラス A と比較する．比較の結果，オブジェクトのクラスが A なら，クラス A に対応するメソッドを直接呼出しする．そうでなければ，次に，クラス B との比較を試みる．どちらにも該当しなければ，間接呼出しによってメソッド呼出しを行う．ここでクラス A，B は呼出し対象のオブジェクトがとりうるクラスである．一般には if 文の段数は任意で，呼出対象のオブジェクトがとりうるクラスの候補数などに応じて決める．比較と直接呼出し（あるいはインライン展開後のメソッドの実行）にかかるコストが間接呼出しのコストより小さい場合には，I-call if 変換によりメソッド呼出しを高速化できる可能性がある．インタフェースメソッド呼出しも同様に最適化できる．

図 5 は JeanPaul の静的コンパイラでインスタンスメソッド呼出しを I-call if 変換した結果である．図 5 のコード中の `ClassLinkTable` は，実行時に動的ロードしたクラスを表すデータ構造のアドレスを収める表である．静的コンパイラが生成したコードは `ClassLinkTable` を介してクラスを表すデータ構造を参照する．`ClassLinkTable` は仮定クラス情報の一部として静的コンパイラが生成し，表の各エントリは実行時にクラスを初期化するとき初期化する（初期化したクラスを表すデータ構造のアドレスを `ClassLinkTable` に収める）．

I-call if 変換の結果，静的コンパイラが生成するコードへの直接呼出しが発生した場合，直接呼出対象のメソッド（たとえば図 5 の *ClassA_method*() と *ClassB_method*()）を配列 N に収める．こうすることで，呼出元のメソッドを直接呼出先のメソッドより後にリンクすることを保証し，まだ使用不能なはずの直接呼出先のメソッドを，呼出元のメソッド経由で呼び出してしまう事態を避ける．また，I-call if 変換で候補のクラスを決めるのにクラスフロー解析[18] を利用した場合，クラスフロー解析時に参照したクラスを集合 S に加える．これはクラスファイルの更新によってクラス階層に変化が起きた場合，クラスフロー解析の結果が無効になりうるためである．冗長な CheckCast の除去にクラスフロー解析を利用した場合も同様に，参照したクラスを集合 S に加える．

```
// object.method() の I-call if 変換後のコード
target_class = object->class;
if (target_class ==
          ClassLinkTable[CLASS_A_OFFSET]){
   // クラスが A なら A の method() を直接呼出し
   ClassA_method(object);
}
else if (target_class ==
          ClassLinkTable[CLASS_B_OFFSET]){
   // クラスが B なら B の method() を直接呼出し
   ClassB_method(object);
}
else{
   // どちらでもなければ間接呼出し
   mdt = object->method_dispatch_table;
   (mdt[METHOD_OFFSET])(object);
}
```

図 5 I-call if 変換
Fig. 5 I-call if Conversion.

2.3.5 インライン展開

静的コンパイラが生成するコードを直接呼出しするメソッド呼出しはインライン展開できる．メソッド $m()$ 中のメソッド $n()$ への直接呼出しをインライン展開する場合，メソッド $n()$ のコンパイルにあたって参照したクラスをメソッド $m()$ のコンパイルにあたって参照したクラスに追加する．具体的には，集合 S_m，配列 N_m に次の操作を施す．

- $S_m \leftarrow S_m \cup S_n$
- 配列 N_m 中のメソッド $n()$ を1つ捨てる．
- 配列 N_m に配列 N_n の要素をすべて追加する．

表 2 SPECjvm98 実行時間
Table 2 Execution time of SPECjvm98.

項目名	HiJIT (sec)	JeanPaul (sec)	
_200_check	0.533	0.409	(1.30)
_201_compress	1144.479	249.04	(4.60)
_202_jess	649.379	400.486	(1.62)
_209_db	945.25	761.886	(1.24)
_213_javac	573.366	335.131	(1.71)
_222_mpegaudio	655.059	291.985	(2.24)
_227_mtrt	990.747	392.775	(2.52)
_228_jack	448.875	304.998	(1.47)
_999_checkit	52.42	30.333	(1.73)
相加平均			(2.05)

※括弧内は HiJIT に対する実行速度比

3. 性能評価

JeanPaul の性能を評価するうえで検証すべき項目は，静的コンパイラと JIT コンパイラの混成システムである JeanPaul がどれだけ静的コンパイラの利点を残しているかだと考える．静的コンパイラの利点は，充実した最適化による実行の高速化と，JIT 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削減にある．本章では，これらの項目について，SPECjvm98 [3] の実行結果から検証する．SPECjvm98 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．

3.1 実行速度

JeanPaul で SPECjvm98 を実行した結果を**表 2** に示す．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 に指定した．実験環境には HITACHI3500/540MP（CPU：PA7200 100 MHz，メモリ：256 MByte，OS：HI-UX/WE2）を用いた．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を指定した．実行速度の比較対象には，日立製作所製の JIT コンパイラのみを利用する Java 実行環境（HiJIT☆☆）を用いた．表 2 から，最大で 4.60 倍，相加平均で 2.05 倍 JeanPaul が HiJIT より高速に動作することが分かる．

図 6 には，JeanPaul で静的コンパイル済みコードのリンクを遅らせる（使用可能か確認した後でリンクする）ことが実行性能に及ぼす影響を示すことを目的として，JeanPaul で通常どおり，静的コンパイル済みコードを使用可能か確認してからリンクしつつベンチマークを実行した結果のほかに，あらかじめすべての静的コンパイル済みコードをリンクしてからベンチマークを実行した結果を示す．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にした全クラスをベンチマーク実行開始前にあらかじめロード，初期化することで実施した．リンクを遅らせた結果，ベンチマーク実行中に静的コンパイル済みコードが使用できない場合，静的コンパイル済みコードの代わりに 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ので，2 つの実行結果の間に差が現れる．

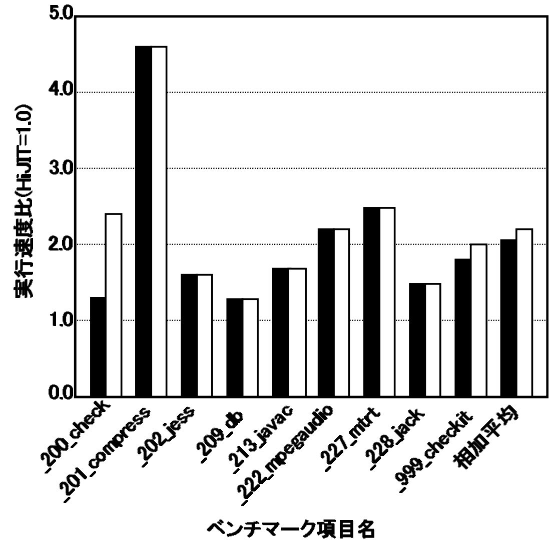

図 6 遅延リンクの影響
Fig. 6 Effect of delayed linking.

図 6 の 2 つの実行結果の比較から，多くのベンチマーク項目で実行速度がほぼ同一であることが分かる．このことから，静的コンパイル済みコードを使用可能か確認した後でリンクする方式が，実行速度に及ぼす影響は小さいといえる．_200_check に限っては静的コンパイル済みコードの事前リンクにより実行性能が約 90%，時間にして 0.162 秒向上したが，高速化原因の 68%（0.11 秒）はクラスを先行してロード，初期化したため，これらの処理をベンチマーク実行中に実施する手間が省けたことによるもので，静的コンパイル済みコードが使用可能になったことの影響はそれほど大きくない．

表 2 の結果を得るにあたり，JeanPaul の静的コン

☆ どのクラスをロードしたかログを出力するオプション．
☆☆ HiJIT を含む JDK for HI-UX/WE2 は日立製作所ソフトウエア開発本部より入手可能である．

表 3 I-call if 変換におけるメソッド候補数の影響
Table 3 Effect of number of the method candidates on I-call if conversion.

項目名	I-call if 変換を施す呼出し点のメソッド候補数の上限				
	2	3	4	5	6
_200_check	1.32	1.30	1.30	1.29	1.31
_201_compress	4.56	4.60	4.55	4.56	4.54
_202_jess	1.62	1.62	1.61	1.54	1.63
_209_db	1.15	1.24	1.27	1.28	1.22
_213_javac	1.74	1.71	1.74	1.72	1.37
_222_mpegaudio	2.25	2.24	2.24	2.24	2.25
_227_mtrt	2.50	2.52	2.45	2.52	2.42
_228_jack	1.47	1.47	1.44	1.38	1.30
_999_checkit	1.70	1.73	1.72	1.87	1.90
相加平均	2.03	2.05	2.04	2.04	1.99

※値は JeanPaul の HiJIT に対する実行速度比

パイラは表 1 の全最適化を実施しており，個々のベンチマーク項目ごとに最適化オプションを変更してはいない．プログラム中のどこにどう最適化をかけるかは静的コンパイラが全自動で決定している．最適化箇所を決定するアルゴリズムのうち，最も調整を必要としたのは I-call if 変換[2] のアルゴリズムだった．I-call if 変換はインライン展開との組合せにより，オブジェクト指向プログラミング言語で記述したアプリケーションを最も高速化できる最適化の 1 つだが，JeanPaul では無条件に I-call if 変換を実施すると，静的コンパイル済みコードが実行時に使用できなくなり，むしろ実行速度が遅くなる．これは，I-call if 変換を実施すると，メソッド呼出しが直接呼出しになる代わり，参照クラス数が増え，静的コンパイル済みコードが使用可能になる時期が遅くなる（実行時により多くのクラスをロード，初期化するまで静的コンパイル済みコードが使用できなくなる）ためである．

試行錯誤の結果，クラスフロー解析により，呼び出しうるメソッドの候補数を 3 つ以下に絞り込めた呼出点に限り I-call if 変換を施すことにした．**表 3** に，I-call if 変換を実施するか否かを定めるメソッド候補数の上限と，JeanPaul の HiJIT に対する実行速度比の関係を示す．表 3 からメソッド候補数の上限が 3 のとき，実行速度比の相加平均がピーク値をとることが分かる．表 3 の _213_javac について，メソッド候補数の上限が 2〜5 の範囲でおおむね一定である実行速度比が，6 になると突然低下するが，この原因は過剰な I-call if 変換により静的コンパイル済みコードが使用不能になることにある．なお，本論文執筆時点では JeanPaul の最適化系は実行時プロファイルを利用しておらず，実行時プロファイルを利用した I-call if 変換の比較順序の調整などにより，まだ高速化できる余地はある．

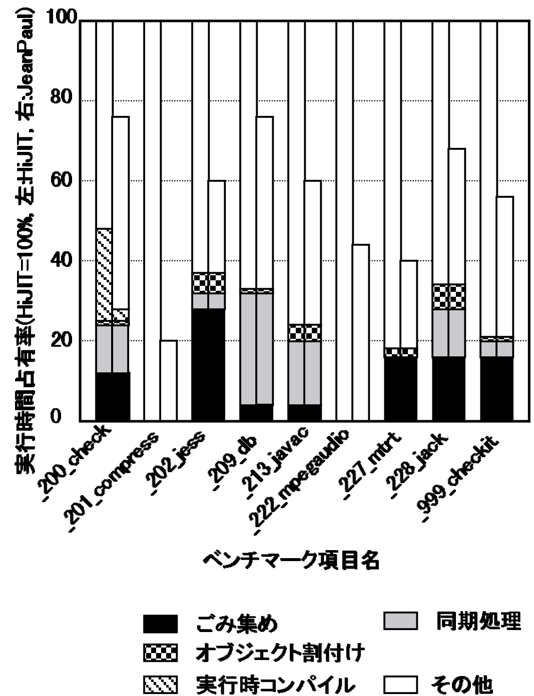

図 7 ごみ集めと同期処理，実行時コンパイルのオーバヘッド
Fig. 7 Overheads of Garbage Collection, Synchronization and Dynamic Compilation.

もっとも，最適化機能の強化だけでは性能向上に限界がある．JeanPaul の静的コンパイラが高速化できるのはプログラムのバイトコードで記述した部分のみであり，たとえばごみ集めや同期処理，オブジェクト割付けなど，バイトコードで記述していない部分（Java 実行環境が提供する C 言語で記述したライブラリ）の実行速度は変わらない．したがって，プログラムの実行時間の多くを非バイトコード部分が占めるベンチマークは，静的コンパイラだけではあまり高速化できない．

静的コンパイラだけで高速化できる範囲を検討するための資料として，SPECjvm98 を構成する各ベンチマークについて，ごみ集めと同期処理，オブジェクト割付け（非バイトコードで記述してある処理のうち，オーバヘッドが大きい 3 項目），それに実行時コンパイルが実行時間のどの程度の割合を占めるか調査した結果を**図 7** に示す．図 7 は HiJIT で各ベンチマーク項目を実行するのにかかる時間を 100%とし，それぞれの処理が HiJIT と JeanPaul の実行時間のどれだけを占めるかを表す．図 7 の「その他」の項目には，静的コンパイラで高速化可能なバイトコード部分のほかに，静的コンパイラで高速化不能な部分（ごみ集め，

同期処理，オブジェクト割付けおよび実行時コンパイル以外の，もともと C 言語などで記述してある部分）の実行コストを含む．図 7 のデータは図 6 と同じ実行条件で取得した．ごみ集めの実行時間は各ベンチマークを `-verbosegc` オプション☆をつけて実行し，実行ログから算出した．同期処理およびオブジェクト割付けの実行時間は，ベンチマーク実行中にそれぞれの処理を実行した回数に，それぞれの処理にかかる平均時間をかけて算出した推定値である．同期処理およびオブジェクト割付けにかかる平均時間は，それぞれの処理を 100 万回（100 回 × 10000 ループ）実行するのにかかった時間を 100 万で割った値とした．

図 7 から，`_213_javac` では，ごみ集めと同期処理，オブジェクト割付けのオーバヘッドが HiJIT で実行した場合の実行時間の 25%を占めており，`_201_compress`，`_222_mpegaudio` を除くその他のベンチマーク項目でも 20〜30%程度を占めることが分かる．ごみ集めと同期処理，オブジェクト割付け以外にも静的コンパイラで高速化できない部分があることを考慮し，仮に静的コンパイラで高速化できない部分の処理にかかる時間が全体の 30%程度であるとすると，静的コンパイラだけでは最大でも HiJIT の 3 倍程度までしか性能向上できないことが分かる．JeanPaul は相加平均で HiJIT の 2.05 倍高速にベンチマークを実行するが，これ以上の高速化には，最適化の強化だけでなく，ごみ集めや同期処理など JVM 内部の処理の高速化が必要だと考える．なお，`_201_compress`，`_222_mpegaudio` では，ごみ集めと同期処理，オブジェクト割付けのオーバヘッドがほぼ 0 だが，これらは JeanPaul が最も高速化できているベンチマークである．

3.2 JIT コンパイル時間

SPECjvm98 の多くのベンチマーク項目が大量のデータ処理に時間を費やすため，図 7 では目立たないが，JeanPaul が高速化する処理の 1 つに JIT コンパイラのオーバヘッドがある．JeanPaul では，始めから静的コンパイル済みコードが使用できる場合には JIT コンパイルを省略できる．各ベンチマーク項目について，JeanPaul がどの程度 JIT コンパイルを省略できるか測定した結果を**図 8** に示す．

図 8 から，JeanPaul を利用することで JIT コンパイル時間を概ね 20%程度に軽減できることが分かる．JIT コンパイルに消費する時間の軽減は，起動オーバヘッドの改善に役立つ可能性があるものの，多くのベンチマーク項目にとって，実行時間全体に与える影響は軽微である．ただし，`_200_check` では HiJIT で実行した場合の 24%を占める（図 7）．図 6 によると JeanPaul は `_200_check` の実行速度を 21%高速化するが，高速化した時間の 92%は動的コンパイルにかかるオーバヘッドの削減から生じている．

☆ ごみ集めの実行ログを出力するオプション．

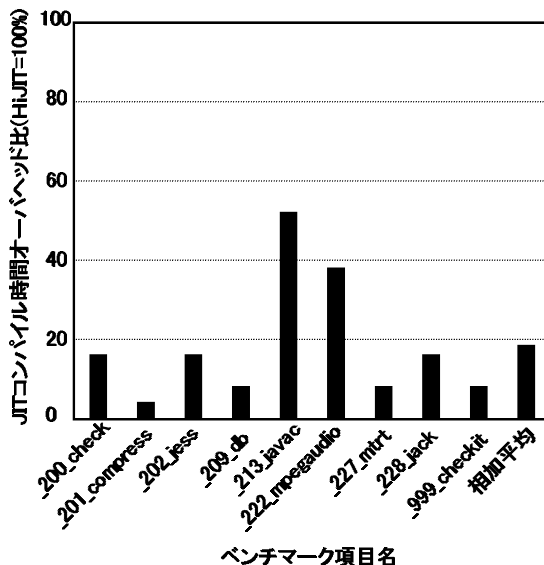

図 8 JIT コンパイル時間
Fig. 8 Overhead of JIT Compilation.

4. 関 連 研 究

JeanPaul の特徴は，メソッドの静的コンパイル済みコードを，実行時に使用可能か確認したうえでプログラムにリンクし，実行速度の向上を図ることにある．

JeanPaul と同様にソースファイルの更新に対応するシステムは，make コマンド[7] など，古くから存在する．make では更新され，再コンパイルが必要なソースファイルを，コンパイル開始時に一括して検出する．これに対し，JeanPaul では `Class.forName()` の挙動などに配慮し，更新されたクラスファイルの検出を実行時に個々のクラスを初期化する際まで遅延する．

オブジェクト指向の分野では Smalltalk などの実装に，ソースファイルの更新に応じて動的コンパイラが生成したコードを破棄・更新するものがある[6),9)]．これらのシステムは，動的コンパイル済みコードの有効期限を定めるために，ソースファイルとの依存関係を保持する．コンパイル済みコードとソースファイル（クラスファイル）の依存関係を保持する点は JeanPaul も同様である．しかし，JeanPaul は静的コンパイル済みコードが使用可能になる時点を定めるために依存関係を利用するという点で，これらのシステムと異なる．

Java 向け静的コンパイラに関しては，これまで数多くの発表があり[5),10),11),14)~17)]，なかには動的ロードのサポートなどを目的として，インタプリタと静的コンパイラを組み合わせたものや[14)]，JeanPaul 同様に，静的コンパイラと JIT コンパイラを組み合わせたものもある[10)]．しかし，静的コンパイル済みコードを，実行時に使用可能か確認したうえでプログラムにリンクして実行速度の向上を試みるシステムを実装し，そのシステムが実用的なアプリケーションをどれだけ高速化するか評価を示している文献や，JeanPaul のように Java アプリケーションを容易に静的コンパイルできるシステムは，これまでに知られていない．

5. 結 論

Java 向け静的コンパイラの使い勝手の向上を目的として，Java 実行環境 JeanPaul を開発している．本論文では，Java 向け静的コンパイラに動的ロードが及ぼす問題を示し，その解決策として，利用可能と確認がとれた場合に限って静的コンパイル済みコードをリンクし，そうでない場合には JIT コンパイラをもちいてプログラムをコンパイル・実行する方法を提案した．提案したシステム JeanPaul を実装し，SPECjvm98 を用いて評価した結果，HiJIT より平均で 2.05 倍ベンチマークを高速実行できることが分かった．

参 考 文 献

1) Budimlic, Z. and Kenedy, K.: Optimizing Java – Theory and Practice, *ACM 1997 Workshop on Java for Science and Engineering Computation* (1997).
2) Calder, B. and Grunwald, D.: Reducing Indirect Function Call Overhead In C++ Programs, *POPL 94*, pp.397–408 (1994).
3) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks (1998). http://www.spec.org/osg/jvm98/
4) Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R. and Wolczko, M.: Compiling Java Just In Time, *IEEE Micro*, Vol.17, No.3, pp.36–43 (1997).
5) Dean, J., DeFouw, G., Grove, D., Litvinov, V. and Chambers, C.: Vortex: An Optimizing compiler for object-oriented languages, *OOPSLA 96*, pp.83–100 (1996).
6) Deutsch, P. and Schiffman, A.: Effiecient Implementation of the Smalltalk-80 System, *POPL 84*, pp.297–302 (1984).
7) Feldman, S.: Make – A Program for Maintaing Computer Programs, *Software-Practice and Experience*, Vol.9, No.4, pp.255–265 (1979).
8) Gosling, J., Joy, B. and Steele, G.: *The Java Lenguage Specification*, Addison Wesley, Reading, MA (1996).
9) Hölzle, U. and Ungar, D.: A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance, *OOPSLA 94*, pp.229–243 (1994).
10) Howard, R.: Developing and Deploying Server-hosted Applications with Java (1997). http:// www.twr.com/java/white-paper.html
11) Hsieb, C.-H., Conte, M., Johnson, T., Gyllenbaal, J. and mei Hwu, W.: OPTIMIZING NET Compilers for Improved Java Performance, *IEEE COMPUTER*, Vol.30, No.6, pp.67–75 (1997).
12) Kolte, P. and Wolfe, M.: Elimination of Redundant Array Subscript Range Checks, *PLDI 95*, pp.270–278 (1994).
13) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, Addison Wesley, Reading, MA (1996).
14) Muller, G., Moura, B., Bellard, F. and Consel, C.: Harrisa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code, *USENIX Conference on Object-Oriented Technologies and Systems*, pp.1–20 (1997).
15) Proebsting, T., Townsend, G., Bridges, P., Hartman, J., Newsham, T. and Watterson, S.: Toba: Java For Applications A Way Ahead of Time (WAT) Compiler, *USENIX Conference on Object-Oriented Technologies and Systems*, pp.41–53 (1997).
16) Siram, K.: Jolt (1996). http://substance.blackdown.org/~kbs/jolt.html
17) 堀田拡二，渡辺雄二：Java 高速実行環境：HBC, *Fujitsu*, Vol.48, No.2, pp.164–167 (1997).
18) 小野寺民也：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).

(平成 12 年 5 月 26 日受付)
(平成 12 年 9 月 8 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．現在，システム開発研究所にてコンパイラの研究開発に従事．ソフトウェア科学会会員．