# オブジェクト指向プログラムの高速化を支援するプロファイラ

神尾貴博[†] 増原英彦[†]

オブジェクト指向プログラムの実行時に，各メソッドが同じ値の引数で何回呼び出されたかを調べるプロファイラを作成した．このプロファイラの情報を利用することでメモ化や部分計算のようなプログラム変換を適用する対象の決定が容易になる．変更可能な状態を持つオブジェクトの同値性を正確に判定するために，プロファイラは各メソッドが参照するフィールドが変化した時刻を記録し，その値によって同値性を判定する．実際に 64,602 行からなるプログラムを人手によるメモ化で高速化した際の経験を基にプロファイル情報の有用性を見積もったところ，検討すべきメソッドの数が 20%以上減るという予測を得た．

# A Profiler for Optimizing Object-oriented Programming by Using Program Transformation

Takahiro Kamio[†] and Hidehiko Masuhara[†]

We present a profiler that enumerates the number of method invocations with the same set of arguments in object-oriented programs. It aims to help identifying methods that would be optimized by program transformations such as memoization and partial evaluation. In order to accurately identify equivalence of mutable objects, the profiler adds a vector of modification times to each object. Based on our experience to manually optimize a 64,602 lines Java program, the information from the profiler is estimated to reduce the number of examined methods by the factor of 20%.

## 1. はじめに

汎用性の高いプログラムは，特定の用途・環境向けに作られたプログラムと比べると実行効率が低いものが多い．その原因の 1 つは，プログラム中の機能をパラメータによって変更できるようにしており，実行時に多くのパラメータを調べて計算を進めているためである．たとえば，オブジェクト指向プログラミングにおけるデザインパターン[2] では，変更可能な機能をオブジェクトとして表すことによって再利用性を高めている．

そのようなプログラムを高速に実行する手法として，部分計算[1),3),5)]，データ特化[7]，メモ化（memoization）などのプログラム変換技術が知られている．これらの技術では，プログラム実行中に同じパラメータを使った計算が繰り返される場合，実際に計算を行う代わりにあらかじめ計算しておいた値を使うことで，高速化を達成する．これらのプログラム変換技術における困難の 1 つは，変換対象の決定である．その理由は**同じ**パラメータが**繰り返し**出現するかどうかはプログラム実行時の性質であり，プログラムの字面から推測するのが困難なためである．特にメモ化のような技術では，実行時にパラメータの同一性を検査するオーバヘッドがあったり，パラメータや結果を保存することによるメモリ消費量の増加があったりするため，適切な対象に適用しないと効率が低下してしまう．

本研究は Java プログラムの実行をプロファイルし，同じパラメータによって繰り返し呼び出されたメソッドを発見するシステムを提案する．このプロファイラと，メソッドの実行時間を計測するプロファイラの結果と合わせることで，プログラムの高速化に貢献するメソッドを容易に発見できるようにすることを目指す．なお，提案するシステムは最適化を適用する箇所のヒントを与えるものであり，実際の最適化を適用するかどうかの判断は人間が行うことを想定している．

Java のようなオブジェクト指向言語では，オブジェクトの状態は代入によって変化しうるため，メソッド呼び出しのパラメータが過去に行われたものと同じであるかどうかは単純には判定できない．本研究では，各オブジェクトに更新時刻ベクトルを持たせることで，より正確なプロファイルを現実的な時間で行うことを

† 東京大学大学院総合文化研究科
Graduate School of Arts and Sciences, The University of Tokyo

可能にした．

実際にクラス数 590 個，64,602 行からなるプログラムを本研究で提案する方式によってプロファイルしたところ，呼び出された 1,513 個のメソッドの中から 1 割以上の速度向上につながるメソッドをより容易に発見することができた．

以降，2 章でプロファイルの対象となるプログラムとその高速化を紹介する．3 章でプロファイラの基本原理と本研究で提案する同値性を，4 章でその実現について述べる．5 章でプロファイラの有用性を評価する．6 章で関連研究にふれ，7 章でまとめを述べる．

## 2. メモ化による高速化

### 2.1 対象プログラム例

本稿では，アスペクト指向言語 AspectJ [6] 1.0.6 版のコンパイラを高速化やプロファイル対象プログラムの例とする．このコンパイラは Java 言語で記述されており，実用レベルの完成度がある一方で，実行効率には改善の余地が残されているといわれている．**図 1** にコンパイラ中の `ASTObject` クラスの定義の一部を示す．このクラスはすべての抽象構文木ノードの親クラスである．4–10 行目で定義されている `getBytecodeTypeDec` メソッドは，構文木の `parent` をたどり，最初に見つかった `TypeDec` 型のオブジェクトを返す．11–13 行目の `getSourceLine` メソッド☆はその抽象構文木ノードが定義されているソースプログラム上の行番号を返す．

```
 1 class ASTObject{
 2   ASTObject parent;
 3   SourceLocation sourceLocation;
 4   TypeDec getBytecodeTypeDec(){
 5     if(getParent() instanceof TypeDec)
 6       return (TypeDec)getParent();
 7     else
 8       return getParent().
 9                   getBytecodeTypeDec();
10   }
11   int getSourceLine(){
12     sourceLocation.getLine();
13   }
14   ASTObject getParent(){return parent;}
15 }
16 class TypeDec extends ASTObject{...}
```

**図 1** プロファイル対象プログラム（一部）
Fig. 1 Profiled program (excerpt).

### 2.2 メモ化による高速化の例

コンパイラの他のクラスの定義を調べると，各 `ASTObject` オブジェクトは `getBytecodeTypeDec` メソッドの呼び出しに対してつねに同じ答えを返すことが分かる．そこで**図 2** のようなメモ化を行うことができる．

```
 1 class ASTObject{
 2   ASTObject parent;
 3   SourceLocation sourceLocation;
 4   static HashMap cache = new HashMap();
 5   TypeDec getBytecodeTypeDec(){
 6     TypeDec r;
 7     if(!cache.containsKey(this)){
 8       if(getParent() instanceof TypeDec)
 9         r = (TypeDec)getParent();
10       else
11         r = getParent().
12                     getBytecodeTypeDec();
13       cache.put(this,r);
14       return r;
15     }else
16       return (TypeDec)cache.get(this);
17   }
18   ...
19 }
```

**図 2** メモ化を行う getBytecodeTypeDec
Fig. 2 getBytecodeTypeDec with memoization.

このプログラムでは，4 行目にレシーバオブジェクトとメソッドの結果の対応を記録するクラス変数 `cache` を定義している．メソッド呼び出しが行われるたびに，すでに結果が記録されているかを調べ（7 行目），存在すればそれを返す（16 行目）．記録されていない場合は変換前と同じ計算を行い，結果を `cache` に記録した後に結果を返す．

実際，図 2 のように変換したコンパイラは，5.2 節に示すように変換前のコンパイラより 1 割以上高速になる．

☆ このメソッドは 3.3 節の説明のために著者が追加したものであり，実際のコンパイラにはない．

## 3. プロファイラの仕組みと問題点

### 3.1 プロファイラの目的

本研究ではプログラムの実行を監視し，各メソッドが同じ値の引数によって何回呼び出されているかを表示する**同値性プロファイラ**を提案する．

前章に示したように，メモ化などのプログラム変換によって高速化されるメソッドは実用的なプログラムの中にも存在するが，どのメソッドが高速化に貢献するかは自明ではない．メソッドが実行される回数や時

**表 1** 同値性プロファイラの出力（一部）
Table 1 Output of equality-based profiler (excerpt).

| シグネチャ | 回数 | 引数 | 割合 (%) |
|---|---|---|---|
| ASTObject.getLexcalType() | 410 | 0 | 10, 10, 3 |
| ASTObject.getBytecodeTypeDec() | 275 | 0 | 25, 4, 3 |
| JavaCompiler.beginSection(String,boolean) | 32 | 0, 2 | 75, 25 |
| NameType.getLegalString() | 10 | 0 | 40, 20 |
| ConstantPool.addMethodRef(NameType, String, String) | 13 | 0, 1 | 54, 38 |

間，同じ引数で呼び出されているかなどは，実際にプログラムを実行してみなければ分からないためである．実行時間を計測するプロファイラ（以降では**時間プロファイラ**と呼ぶ）を用いれば，実行時間や回数の多いメソッドを発見することはできるが，同じ引数で呼び出されているかどうかまでは分からない．

そこで本研究では，同値性プロファイラと時間プロファイラの結果を合わせ，同じ引数で繰り返し実行されるメソッドを特定することで，高速化に貢献するメソッドを効率的に発見することを狙っている．

本研究のプロファイラが提示したメソッドに対して最適化を施すかどうかは最終的に人間が判断する．これは，本プロファイラでは最適化が可能かどうか，最適化を施した場合の性能向上を見積もることが難しいためである．たとえば，外部との入出力を行うメソッドは，プロファイラが「同じ引数で呼び出されている」と判定したとしてもメモ化のような最適化を施すことはできない．

### 3.2 プロファイラの基本原理

同値性プロファイラは，対象プログラムの実行を監視し，メソッド呼び出しが起きるたびに，そのメソッドのシグネチャとそのすべての（レシーバを含む）引数の値の組を記録する．プログラムの終了後，各メソッドについて記録された引数の組のすべての組合せを調べ，同じ値の組合せで呼び出された回数の比率の多い順に表示する．**表 1** はプロファイラの表示の一部を整理したものである．各行は対象プログラム中の各メソッドについて，左からシグネチャ，呼び出された総回数，同じ値の組であることの判定に用いられた引数の番号の集合（0 はレシーバを表す），同じ引数で呼び出された割合を示している．たとえば 1 行目は `getLexicalType` メソッドの呼び出し 410 回のうち，第 0 引数（レシーバオブジェクト）が同じであった呼び出しの割合が 41 回（10%），41 回（10%），12 回（3%）であったことを示している．

メソッドが複数個の引数をとる場合には，原則としてすべての引数が同じであった呼び出しの割合を表示する．ただし，その割合が低かった場合（たとえば，1 つの引数は毎回違う値になっている）には，「同じ呼び出し」が一定の割合以上になるように一部の引数だけを比較する．たとえば `beginSection(String,boolean)` は，第 0，2 引数だけを比較した場合，全 24 回の呼び出しのうちそれぞれ 24 回（75%）と 8 回（25%）は同じ引数の組であったことを示している．

このようにプロファイラは一部の引数を無視して同じ値の組を数える場合もある．その理由は，メモ化や部分計算のような最適化では，すべての引数が同じにならなくても一部の引数が同じであれば，それらが関係する計算を効率化できることが知られているためである．現在は，同じ値の組が一定の割合以上になるように無視する引数を自動的に決定している．

### 3.3 引数の同値性

上述のように同値性プロファイラは，プログラム本体の終了時に実行中に記録した引数の値を調べている．オブジェクト指向言語のように代入によって状態を更新できる言語では，「同じ値」の定義は自明でない．定義によっては，メモ化などの最適化にとっては**不正確**，つまり異なると判定されるべき引数を同じだと判定されてしまったり，**厳密すぎる**，つまり同じと判定すべき引数を異なると判定してしまったりする可能性がある．また，判定をするために必要なメモリ量などのオーバヘッドも異なる．

本研究では，今回のプロファイルの目的に適した精度・効率を与える同値性として**メソッドごと参照・時刻同値**と呼ぶ同値性を提案する．以下では提案手法を含む**表 2** に示した 4 つの同値性の定義と正確さ・厳密さ・効率について説明する．また以下では，ある時点でメソッド $m$ の引数としてオブジェクト $O_1$ が出現し，別の時点で $m$ の引数としてオブジェクト $O_2$ が出現したとして説明する．

#### 3.3.1 参 照 同 値

$O_1$ と $O_2$ が**参照同値**であるとは，$O_1$ と $O_2$ の参照が等しいこととする．参照同値を用いるプロファイラは，1 つの値のために 1 ワードの参照を記録するだけでよいため，メモリ使用量は効率的である．一方，参照同値ではオブジェクトの状態を無視するため，不正確な判定をする問題がある．

たとえば `ast` オブジェクトが**図 3** のように使

**表 2** 同値性判定の種類と特徴（$m$ はプログラム中のメソッドの総数．$k$ はオブジェクトグラフの大きさ）
Table 2 Equality of objects and the feature ($m$ denotes the number of methods in the program. $k$ denotes the size of an object graph).

| 同値性 | 参照同値 | 構造同値 | 参照・時刻同値 | メソッドごと参照・時刻同値 |
|---|---|---|---|---|
| 引数の記録に必要なメモリ（ワード） | 1 | $k$ | 2 | 2 |
| 1 オブジェクトに追加されるメモリ（ワード） | 0 | 0 | 1 | $m$ |
| 状態更新を区別 | × | ○ | ○ | ○ |
| アクセスのないフィールドの更新を無視 | × | × | × | ○ |

```
1 l1 = ast.getSourceLine();
2 typeDec1 = ast.getBytecodeTypeDec();
3 ast.sourceLocation.lineNumber = newLN;
4 l2 = ast.getSourceLine();
5 typeDec2 = ast.getBytecodeTypeDec();
```

**図 3** ASTObject を使う計算の例
Fig. 3 A code fragment that manipulates an ASTObject.

われていたとする．この場合 1 行目と 4 行目の getSourceLine のレシーバは 3 行目の代入で状態が変化しているにもかかわらず，参照同値である．したがって，この結果を元に getSourceLine のメモ化を行うと誤った結果を返すプログラムになってしまう．

#### 3.3.2 構造同値

$O_1$ と $O_2$ が**構造同値**であるとは，$O_1$ と $O_2$ が，(1) 同じクラスのオブジェクトであり，(2) 同名のプリミティブ型フィールドにそれぞれ等しい値を持ち，(3) 同名の参照型フィールドにそれぞれ構造同値な値を持つことである．

構造同値を判定する素朴な方法は，プロファイラが引数 $O_1$ を記録する際に，$O_1$ から参照をたどれるすべてのオブジェクト（つまり $O_1$ を起点とするオブジェクトグラフ）をコピーし，オブジェクトグラフどうしの比較を行えばよい．しかし，オブジェクトグラフはしばしば非常に大きくなるため，この手法は現実的な効率を期待できない．

また，構造同値は，実際には参照されないフィールドも同値であることを要求するため厳密すぎる判定をする場合もある．たとえば図 3 の 2 行目と 5 行目の getBytecodeTypeDec 呼び出しにおける ast オブジェクトの値は，3 行目でフィールドを更新されているため構造同値でない．しかし getBytecodeTypeDec メソッドは parent フィールドしか参照していないため，図 2 に示したようなメモ化を行っても正しい結果を返す．つまり，メモ化のような目的を考えると ast オブジェクトの値は，getBytecodeTypeDec にとっては同値と判定されるべきである．実際のアプリケーションは，1 つのオブジェクトに様々な種類の値を持たせることが多いため，このような状況は頻繁に生じていると考えられる．

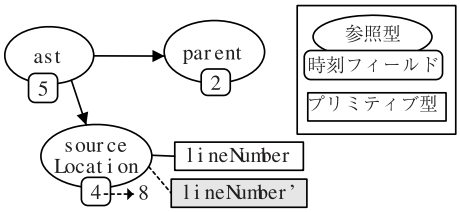


**図 4** オブジェクトグラフと更新時刻
Fig. 4 References and last modified time.

#### 3.3.3 参照・時刻同値

構造同値を効率的に近似する同値性として参照・時刻同値を定義する．

準備として**オブジェクトの更新時刻**を定義する．オブジェクト $O$ の**更新時刻**とは，$O$ のフィールドに代入が行われた最後の時刻とする．また $O$ を起点とする**オブジェクトグラフの更新時刻**とは，$O$ からたどれるすべてのオブジェクトの更新時刻の中で最後のものとする．

$O_1$ と $O_2$ が**参照・時刻同値**であるとは，$O_1$ と $O_2$ の参照が等しく，$O_1$，$O_2$ を起点とするそれぞれのオブジェクトグラフの更新時刻が等しいことである．たとえば **図 4** のように，オブジェクト ast が parent と sourceLocation を参照しているとする．このとき ast を起点とするオブジェクトグラフの更新時刻は 5 である．さらに，時刻 8 に sourceLocation のフィールド lineNumber に代入が起きたとする．このとき sourceLocation の更新時刻は 8 となるので，ast を起点とするグラフの更新時刻も 8 となる．

この同値性は，構造同値による同値性の良い近似になっている．実際，図 3 の例に対しては構造同値と参照・時刻同値のどちらを用いても，同値性について同じ判定をする．参照・時刻同値は，各オブジェクトに 1 ワードの時刻フィールドを追加するだけで実現できる．プロファイラは，(1) オブジェクトのフィールドに代入があったときには，そのオブジェクトの時刻を更新する．(2) メソッドの呼び出しがあったときには，引数の参照とそれを起点としたオブジェクトグラ

フの更新時刻を記録すれば，同値性を調べることができる．しかし，この方法でも構造同値と同様に，あるメソッドが参照しないフィールドが更新された場合でも同値でなくなるという問題が残っている．

#### 3.3.4 メソッドごと参照・時刻同値

構造同値のところで示した問題を解決するため，本研究では**メソッドごと参照・時刻同値**を提案する．直感的にはメソッド $m$ の引数に現れた引数 $O_1$，$O_2$ の同値性は，$m$ の実行中（$m$ から呼び出されたメソッドも含む）に参照されうるオブジェクトのフィールドだけを比較するものである．

準備として，メソッド $m$ の**参照フィールド集合**を，$m$ および $m$ から呼び出されたメソッドの実行中に起こりうるフィールド参照☆のクラスとフィールド名の集合とする．図 1 の例では `getBytecodeTypeDec` の参照フィールド集合は {`ASTObject.parent`} となる．また，**$m$ に関するオブジェクト（グラフ）の更新時刻**とは，オブジェクト（グラフに含まれるすべてのオブジェクト）のフィールドのうち，$m$ の参照フィールド集合に含まれるものが更新された時刻の中で最後のものとする．

このとき $O_1$ と $O_2$ が $m$ に関して参照・時刻同値であるとは，$O_1$ と $O_2$ の参照が等しく，$O_1$，$O_2$ を起点とするオブジェクトグラフの $m$ に関する更新時刻が等しいこととする．

この定義では，あるオブジェクトの $m$ が参照しないフィールドに代入が起きてもオブジェクトの同値性は保たれる．図 3 のプログラムにおける `ast` オブジェクトの `getBytecodeTypeDec` に関する同値性を考える．3 行目での `ast.sourceLocation.lineNumber` への代入は参照フィールド集合に含まれないので，**図 5** に示すように `getBytecodeTypeDec` に関する `ast` の更新時刻を変更しない．したがって 2 行目と 5 行目の `ast` オブジェクトは `getBytecodeTypeDec` に関して参照・時刻同値となる．

一方，`ast` オブジェクトは `getSourceLine` に関して参照・時刻同値ではない．このメソッドの参照フィールド集合は `ASTObject.sourceLocation.lineNumber` を含むため，3 行目の代入は，図 5 に示すように `sourceLocation` オブジェクトの `getSourceLine` に関する更新時刻を変更する．結果として 1 行目と 4 行目の `ast` オブジェクトは `getSourceLine` に関して更新時刻が異なるので参照・時刻同値ではない．

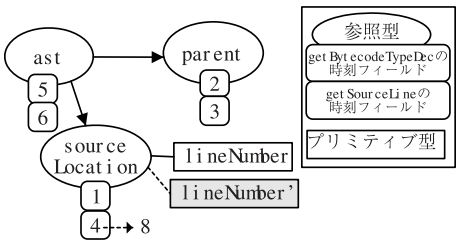


**図 5** メソッドごと参照・時刻同値
Fig. 5 Per-method reference and modification-time equality.

メソッドごと参照・時刻同値の下では，1 つのオブジェクトは，注目するメソッドによって異なった更新時刻をとりうる．素朴には，(1) 各オブジェクトに時刻フィールドを 1 つだけ持たせ，同じ入力の下で各メソッドについてプロファイル実行を一度ずつ行うか，(2) 各オブジェクトにプログラム中のメソッドの数と同数の時刻フィールドを持たせ，1 回のプロファイル実行を行うかのどちらかである．本研究では (2) を改良した方法をとり，参照フィールド集合が同じメソッドは同じ時刻フィールドを使うことで時刻フィールドを減らす．実際，ある 1 つのクラスのフィールドに参照を行うメソッドの個数はプログラム中のすべてのメソッドの個数に対して少ないので，9 割以上のフィールドが削減できている．

プロファイラは，メソッド $m$ の呼び出しが起きると，各引数の参照と $m$ に関する更新時刻をそれぞれ記録する．また，オブジェクトのフィールドへの代入があった場合は，そのフィールドを参照フィールド集合に持つメソッドに関する更新時刻を更新すればよい．

AspectJ のコンパイラを用いた実験では，参照・時刻同値による判定で同値となる引数が出現しなかったメソッドのうち，メソッドごと参照・時刻同値で新たに約 16%のメソッドを同値となる引数が出現すると判定できた．

## 4. 実　　現

同値性プロファイラはアスペクト指向言語 AspectJ を用いて実装した．コードの大きさは 1,102 行である．このプロファイラは対象プログラムを同じ入力を用いて 2 回実行する．1 回目は各メソッドの参照フィールド集合を作成し，2 回目はメソッド呼び出しの引数を記録し同値性を調べる．

メソッド呼び出しの引数の値の記録は，すべてのメソッドの呼び出し時に実行されるアドバイスを定義して実現した．このアドバイスは AspectJ と Java のリ

☆ メソッド $m$ を任意の入力・任意の文脈の下で実行した場合に参照しうるフィールドを含むものとする．ただし，現在のプロファイラの実現では，試し実行の最中に実際に参照されたフィールドのみを調べている．

**表 3** プロファイラの実行時間（秒）
Table 3 Execution times of profilers (sec.)

| 実現方式 | user + sys |
|---|---|
| 元のプログラム | $1.82 \times 10^0$ |
| A：1 時刻フィールド × $m$ 回 | $2.07 \times 10^5$ |
| B：$m$ 時刻フィールド共有 × 1 回 | $3.46 \times 10^2$ |

フレクション機能を用いて，引数の参照（ハッシュ値）と更新時刻を調べ，記録する．

また，すべてのオブジェクトに更新時刻の配列を持たせるために AspectJ の型間定義（inter-type declaration）を用いた．ただし，AspectJ の制約から標準ライブラリのクラスには追加していない．

## 5. 評　　価

実際に作成したプロファイラの実行効率と有用性を調べるための実験を行った．以下の実験は，AspectJ 1.0.6 のコンパイラ（クラス数 590 個，64,602 行からなる Java プログラム）が付属の例題プログラムである Point.java☆1（75 行）をコンパイルする時間についての計測である．また，このコンパイルでは 1,513 種類のメソッドが呼び出された．

### 5.1 プロファイラの実行効率

3.3.4 項で提案したメソッドごと参照・時刻同値性の 2 種類の実現方式について，プロファイラの実行速度を比較した．

実現方式は，（A）各オブジェクトに時刻フィールドを持たせ，メソッドの個数回のプロファイルの実行を行うもの，（B）各オブジェクトにフィールド参照集合が異なるメソッドの数だけ時刻フィールド配列を持たせ 1 回実行するものである．（A）は単純な方式であるため，実現の労力や 1 回あたりの実行速度に関しては（B）より有利であると予想できる．このトレードオフを明らかにするために（A），（B）両方の実行時間を測り比較した．

実行は 1.9 GHz の Pentium4 と 512 MB のメモリを搭載した Vine Linux 2.6 上の Java2 SDK 1.4.1 および AspectJ 1.1.1 で行った．計測には Linux の time コマンドを用い，ユーザ時間とシステム時間の和を実行時間とした．

**表 3** に示された結果のように，方式 B は A より大幅に速い．B はプロファイルしないプログラム実行の約 200 倍の時間でプロファイルが終了しており，A のように元のプログラムと比べて 5 桁以上の時間がかかるものと比べて明らかに速くなっている．

### 5.2 プロファイル結果の有用性

メモ化によるプログラムの最適化を行う際に，同値性プロファイラによって得られる情報がどの程度有益であったかについて調べた．

まず，プログラムの各メソッドの実行時間を Java VM の `-Xrunhprof` オプション☆2により計測した．そこで得た実行時間の長い順に 56 個のメソッドを検討し，メモ化が適用できた☆3ものについてそれぞれ単独でメモ化した場合の全体の実行時間の変化を調べた．

**表 4** に実際に全体の実行時間を短縮したメソッドを示す．表の各行は左から順に，メソッド名，メソッドの行数，変換前のメソッドが全体の実行時間に占めた割合，総呼び出し回数，メソッドごと参照・時刻同値による引数の同値率（同値率 A），参照・時刻同値による同値率（同値率 B），高速化の程度である．ここで，あるメソッドの引数の同値率とは，同値な引数の組のうち呼び出し回数の多かった上位 3 組の回数の合計をそのメソッドが呼び出された回数で割った値としている．また，高速化の程度とは，そのメソッドだけメモ化した場合のプログラム全体の実行時間を，メモ化前の実行時間で割った値である．全体の実行時間は，対象プログラムの `main` メソッドの本体 1,000 回の繰り返しに要した時間を `System.currentTimeMillis` メソッドで計り平均した．高速化によって短縮された率が各メソッドの相対実行時間よりも大きい理由は，（1）前者は Java VM の起動から終了までの時間を計っているのに対し，後者は `main` メソッドの部分のみを計っていることと，（2）後者の時間にはそのメソッドから呼び出されたメソッドの実行時間が含まれていない点があげられる．実行は 3.06 GHz の Pentium Xeon を 2 つと 6 GB のメモリを搭載した Redhat Linux Advanced Server 2.1 上の Java2 SDK 1.4.2 で行った☆4．

まず，同値率 A と B を比較してみると，表にあげた 5 つのメソッドのうち 3 つで同値率 B の値よりも A の値の方が大きくなっている．つまり，メソッド参照・時刻同値を用いることで，これまで同値でないと判定されていた値が同値であると判定できるようになる場合があることが確認できる．また `ConstantPool.addUtf8` メソッドのように，同値率が大きく変化するメソッドがあることも確認できる．以降は，メソッドごと参照・

☆1 Point.java は Java プログラムであるので，AspectJ コンパイラの中の Java プログラムをコンパイルする部分のみプロファイルしたことになる．

☆2 このオプションには samples 方式と times 方式があるが，前者を用いた．

☆3 複雑なメソッドについてはメモ化ができないと判断した．

☆4 この実験は 5.1 節とは異なる環境で行ったが，両者の実行時間はほぼ同じ傾向であった．

**表 4** 最適化されたメソッドのプロファイル結果（point）
Table 4 Optimized methods and their profile results (point).

| メソッド名 | 行数 | 実行時間 (%) | 総呼出回数 | 同値率 A (%) | 同値率 B (%) | 高速化 (%) |
|---|---|---|---|---|---|---|
| ASTObject.getLexicalType() | 9 | 0.09 | 410 | 23.9 | 23.9 | 89.9 |
| ASTObject.getBytecodeTypeDec() | 9 | 0.09 | 275 | 34.2 | 26.2 | 88.8 |
| JarClassManager.makeSubPackageManager(String) | 12 | 0.31 | 121 | 40.5 | 40.5 | 96.9 |
| ASTConnection.makeTypeD(String) | 3 | 0.77 | 184 | 100 | 74.5 | 98.9 |
| ConstantPool.addUtf8(String) | 3 | 0.09 | 124 | 98.4 | 47.6 | 97.3 |

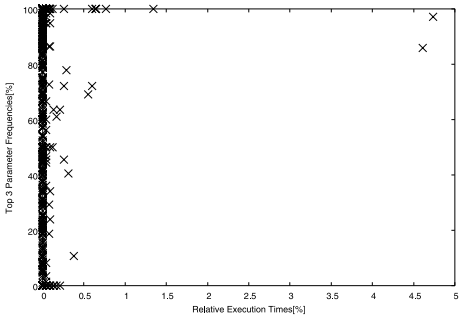


**図 6** メソッドの相対実行時間と同値率
Fig. 6 Relative execution times and equality ratio of methods.

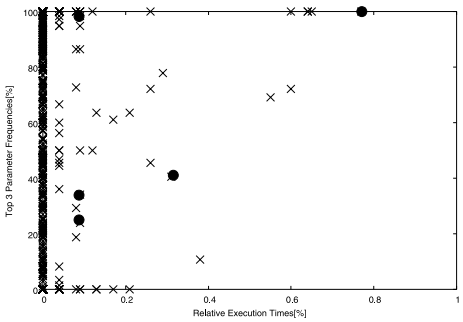


**図 7** メソッドの相対実行時間と同値率（一部）
Fig. 7 Relative execution times and equality ratio of methods (excerpt).

時刻同値（同値率 A）に注目して議論する．

次に高速化に貢献したメソッドが，どのような実行時間と引数の同値率を持っていたかを調べた．**図 6** のグラフは，プログラム中の各メソッドの実行時間率と引数の同値率の関係を示している．また，**図 7** は図 6 の一部を拡大したものである．横軸は各メソッドの実行時間の全体に占める割合，縦軸は各メソッドの引数の同値率であり，各十字は 1 つのメソッドに対応している．図 7 の黒丸は表 4 に示されたメソッドに対応している．これから分かるように，高速化に貢献したメソッド自体は多くないものの，すべて同値率が 20%以上になっていることが分かる．また，同値率が 0 に近く，かつ，実行時間の長いメソッドが相当数あるため，同値性プロファイラによってこれらのメソッドを除外することができると予想できる．

これらのデータを元に，人間が対象プログラムのメソッド定義を順に検討する際の同値率の有用性を推定する．まず，人間が高速化を適用する際に，各メソッドの実行時間の多い順に検討したとする．この場合，表 4 にあるすべてのメソッドを検討するためには，30 個のメソッドを検討する必要がある．次に，同値率 20%以上のメソッドだけを，メソッドの実行時間の多い順に検討したとする．この場合，23 個のメソッドを検討すればよい．結果として，同値率を用いることで検討するメソッドの数を 20%以上減らすことができたといえる．また，メソッドの行数に注目してみても，30 個のメソッドの合計行数 383 行のうち，23 個のメソッドの合計行数 295 行を検討すればよい．結果として，同値率を用いることで検討する行数を 20%以上減らすことができたといえる．この推定から一般的な有用性を結論することはできないが，検討対象を減らす以外の有用性としては，引数の同値性をあらかじめ知ることによってメモ化の労力が減ることを，著者が各メソッドを変換する際に感じられた．

そして，変換したメソッドはそのままに，AspectJ のコンパイル対象を Java2 SDK に付属のプログラム TableExample3（596 行）にした結果を**表 5** に示す．表中の - は`-Xrunhprof` による時間プロファイラで検出されなかったことを示している．実行時間の割合が減少し同値率が 20%を切ったメソッドに関しては変換後の方が遅くなったが，その他のメソッドについては数%ではあるものの高速化した．また，ここでもメソッドごと参照・時刻同値の効果を確かめることができる．

## 6. 関 連 研 究

プロファイル情報を用いて最適化対象を決定するシステムは少なくない（[4] など）．特に just-in-time コンパイラや HotSpot コンパイラでは，コンパイル対象やインライン化を適用する手掛かりを集めるものが多くある．これらのプロファイラはプログラム実行中に情報を収集するため，メソッドの実行回数や時間

**表 5** 最適化されたメソッドのプロファイル結果（TableExample 3）
Table 5 Optimized methods and their profile results (TableExample 3).

| メソッド名 | 行数 | 実行時間 (%) | 総呼出回数 | 同値率 A (%) | 同値率 B (%) | 高速化 (%) |
|---|---|---|---|---|---|---|
| ASTObject.getLexicalType() | 9 | - | 6883 | 17.3 | 16.6 | 101.4 |
| ASTObject.getBytecodeTypeDec() | 9 | - | 2695 | 14.5 | 10.4 | 101.8 |
| JarClassManager.makeSubPackageManager(String) | 12 | 0.05 | 253 | 22.9 | 22.9 | 95.5 |
| ASTConnection.makeTypeD(String) | 3 | 0.51 | 3640 | 71.7 | 64.4 | 97.0 |
| ConstantPool.addUtf8(String) | 3 | 0.08 | 1395 | 82.7 | 13.8 | 95.9 |

のような簡単な情報を集めるだけのものがほとんどである．値に関するプロファイルを行うものとしては，Shaham らのヒープ使用状況解析するプロファイラ[9]がある．このシステムは，改造した Java VM を用いて各オブジェクトに時刻を持たせ，ガーベッジ・コレクタが回収できないけれども使われることのないオブジェクトを見つけることができる．

Calpa[8] は，実行時特化システム DyC[3] のためのプロファイラである．これは実行頻度や変数の値を調べ，自動的に高速化の対象を決定するが，オブジェクト指向プログラムを対象としていない．

プロファイル情報を用いずに高速化の対象を発見する方法として，Schultz らはデザインパターンを用いて書かれたプログラムに部分計算を適用する方法を提案した[10]．この方法は，実験的に書かれたプログラムについては有用性が示されているが，大規模なプログラムに対してどのように適用できるかは明らかではない．

## 7. まとめと今後の課題

本発表では，オブジェクト指向プログラムをメモ化によって高速化するような場合に有用な情報を提示する同値性プロファイラを提案した．このプロファイラは，プログラムの試し実行中に現れたメソッド呼び出しの引数を記録し，同じ引数で繰返し実行されたメソッドを発見する．同じ引数を精度良くかつ効率的に判定するため，メソッドごと参照・時刻同値を提案した．AspectJ のコンパイラの実行をプロファイルする実験により，元のプログラムに比べて 200 倍程度の時間でプロファイルが完了することと，メモ化による高速化に貢献するメソッドをより容易に発見できることを確かめた．具体的には，人手によるメモ化を適用する際に，同値性プロファイラを利用することで，適用を検討するメソッドの数を 20%以上減らせるという見積りを今回の実験では得られた．

今後はこのプロファイラをより多くのプログラムに適用し，より的確に高速化に貢献するメソッドを提示する手法を検討してゆく．特に現在はメソッド内部での値の使い方を人間が判断しているが，自動的に判定することは重要だと考えている．

## 参 考 文 献


1) Consel, C., Lawall, J., Marlet, R., Muller, G., Noye, J., Thibault, S. and Volanschi, N.: Tempo: Specializing systems applications and beyond, *ACM Computing Surveys, Symposium on Partial Evaluation*, Vol.30, No.3 (1998).
2) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns*, Addison-Wesley (1995).
3) Grant, B., Mock, M., Philipose, M., Chambers, C. and Eggers, S.J.: DyC: An Expressive Annotation-Directed Dynamic Compiler for C, Technical Report UW-CSE-97-03-03 (1998).
4) Grove, D., Dean, J., Garrett, C. and Chambers, C.: Profile-Guided Receiver Class Prediction, *ACM SIGPLAN '95 Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA'95*), pp.108–123 (1995).
5) Jones, N.D., Gomard, C.K. and Sestoft, P.: *Partial Evaluation and Automatic Program Generation*, Prentice-Hall International Series in Computer Science (1993).
6) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *Proc. European Conference on Object-Oriented Programming* (*ECOOP'01*), pp.327–353 (2001).
7) Knoblock, T.B. and Ruf, E.: Data Specialization, *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation* (*PLDI'96*), pp.215–225 (1996).
8) Mock, M., Chambers, C. and Eggers, S.J.: Calpa: A Tool for Automating Selective Dynamic Compilation, *Proc. 33rd Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO-33*), pp.291–302 (2000).
9) Shaham, R., Kolodner, E. and Sagiv, M.: Heap profiling for space-efficient Java, *ACM SIGPLAN '01 Conference on Program-*

*ming Language Design and Implementation* (*PLDI'01*), pp.104–113 (2001).
10) Schultz, U.P., Lawall, J.L. and Consel, C.: Specialization Patterns, *Proc. 15th IEEE International Conference on Automated Software Engineering* (*ASE*), pp.197–206 (2000).

(平成 16 年 2 月 23 日受付)
(平成 16 年 10 月 4 日採録)

**神尾 貴博**

1980 年生．2002 年早稲田大学理工学部情報学科卒業．2004 年東京大学大学院総合文化研究科広域科学専攻修士課程修了．現在みずほ情報総研（株）に勤務．

**増原 英彦**（正会員）

1970 年生．1992 年東京大学理学部情報科学科卒業．1994 年同大学大学院理学系研究科情報科学専攻修士課程修了．1995 年同専攻博士課程中退．東京大学大学院総合文化研究科助手・講師を経て 2002 年より助教授．2001 年から 2002 年までカナダ・ブリティッシュコロンビア大学 Visiting Assistant Professor．博士（理学）．アスペクト指向プログラミング・自己反映計算・部分計算等に興味を持つ．1996 年情報処理学会論文賞受賞．ACM，日本ソフトウェア科学会各会員．