解　説

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

Optimizing Message Sends in Object-Oriented Languages by Tamiya ONODERA (IBM Japan, Tokyo Research Laboratory).

小野寺　民　也[1]
1 日本アイ・ビー・エム(株)東京基礎研究所

1. はじめに

メッセージ送信はオブジェクト指向言語における基本的な計算パラダイムであり，手続き型言語における関数呼出しに相当する．しかし，関数呼出しとは異なり，実行時に起動されるメソッドはレシーバのクラスによって変化し，一般にコンパイル時に決定することができない．そのため，レシーバのクラスとセレクタ（メッセージ名）の対から実行すべきメソッドを探し出す必要が生じる．これがいわゆる**メソッド探索**と呼ばれるものである．

メソッド探索の基本的な実装を説明すると，まず，探索のためのデータ構造として，1つのクラスに対してそのクラスに関する情報を表すオブジェクトをつくり，スーパクラス鎖と**メソッド辞書**をもたせる．メソッド辞書には，そのクラスで定義されているメソッドを，セレクタをキーとして格納しておく．実際のメソッド探索では，セレクタをキーにレシーバクラスのメソッド辞書を検索し，対応するメソッドがみつからなければスーパクラス鎖にしたがってこれを繰り返す．

このように，メッセージ送信は，メソッド探索のため関数呼出しに比べかなり重いものとなる．しかも，オブジェクト指向言語でのプログラミングでは，より小さなメソッドが多数作成される傾向があるため，実行頻度もより高い．

メッセージ送信の重量と頻度を考えると，揺籃期の処理系ですら簡単な高速化を施していたのも当然で，以来現在に至るまで，メッセージ送信の高速化は，オブジェクト指向言語の実装における中心的な課題の1つとして精力的に研究されてきた．この間，多数の技法が提案されているが，本稿で代表的なものを概観してみたい．

ところで，オブジェクト指向言語といってもさまざまで，メッセージ送信の実装も一様ではない．したがって，まず第**2**章で，メッセージ送信の実装への影響という観点から，代表的な言語を分類整理する．続いて，メッセージ送信の高速化技法を3つに大別して説明する．すなわち，**キャッシュ法**（caching），**ディスパッチ表法**（table dispatching），**直接束縛法**（direct binding）で，それぞれ第**3**，**4**，**5**章で取り扱う．第**6**章では，これらの高速化技法を，最も普及していたオブジェクト指向言語であるC++の視点からみてみる．最後に第**7**章で結論する．

2. オブジェクト指向言語の類別

オブジェクト指向言語の特徴の中で，メッセージ送信の実装にとって重要なのは，純正度，型，実行系の3つである．

純正のオブジェクト指向言語では，計算対象はすべてオブジェクトで，計算ないし演算はすべてメッセージ送信により実行される．SelfやCecilは純正の言語で，整数や浮動小数点もオブジェクトであり四則演算や制御構造（if文やwhile文など）もメッセージ送信式になるが，C++やJavaではそうではない．また，Javaには通常の関数は存在しないがC++では許される．純正度が高ければ，メッセージ送信の頻度はより高くなり，最適化の要求もより強くなる．

型の観点からは，型宣言のある**静的型言語**（statically typed languages）と型宣言のない**動的型言語**（dynamically typed languages）に分類できる．C++やJavaは前者に，SelfやSmalltalkは後者に属する．一般に，型情報は最適化にきわめて有用である．静的型言語ではこれを簡単に入手することができるため，高速化は相

対的に容易であるといえる．

言語の実行系に視点を移してみると，C＋＋のように実行系が実マシンであるものと，Smalltalk のように**仮想マシン**（virtual machine）であるものがある．仮想マシン方式の言語では，ソースコードは，**バイトコード**という仮想マシン命令にコンパイルされ，これを仮想マシンが実行する．仮想マシンは実マシン上の解釈系として実現されることも多いが，先進的なシステムでは，仮想マシンは，仮想マシン命令から実マシン命令へのコンパイラを装備しており，メソッドを初めて実行するときに実マシン命令に変換する．これは**動的コンパイル**と呼ばれる重要な技法である．また，これに対比して，実行前にすべて終了していなければならないコンパイルを**静的コンパイル**という．

表-1 に，代表的なオブジェクト指向言語の分類を示す．実行系については現在の典型的な実装に基づいて判断しているだけで，必ずしも固定的なものではない．もちろん言語と実行系にかなりの相性はあるが，C＋＋を仮想マシンで実装することもできるだろうし，Smalltalk でも静的コンパイルを採用することができるであろう．

最後に，用語について述べておくと，言語によって相違がある場合は本稿では原則として Smalltalk のそれを用いる．**メッセージ送信**は，C＋＋では「仮想関数呼出し」と呼ばれ，Java では「static でも final でもないメソッドの起動」に相当する．**メソッド**は，C＋＋では「仮想関数」で Java では「static でも final でもないメソッド」のことになる．**セレクタ**は，Java では「メソッドシグネチャ」と呼ばれC＋＋では「エンコードされた関数名」のことになる．

表-1　代表的なオブジェクト指向言語の分類

言　語	純正度	型	実行系
C＋＋	低い	静的	実マシン
Modula-3	低い	静的	実マシン
Java	やや低い	静的	仮想マシン
Smalltalk	やや高い	動的	仮想マシン
Cecil	高い	両方	実マシン
Self	高い	動的	仮想マシン

3．キャッシュ法

一般にキャッシュは高速化の常套手段の1つだが，メソッド探索の高速化法としては主として動的型言語で使われる．キャッシュの配置の点でいくつかのバリエーションがある．

3.1　メソッドキャッシュ

探索結果を格納するためにシステムに1つ設けられるキャッシュを，**メソッドキャッシュ**という[14]．実体はハッシュ表で，クラスとセレクタとメソッドの3つ組が積まれ，ハッシュ値は，クラスとセレクタの対から，それぞれのアドレスで論理演算をするなどして計算される．

実際のメソッド探索ルーチン lookup はどうなるかというと，まず，与えられたクラス C とセレクタ S の対からハッシュ値を計算し，これをインデックスとしてキャッシュを覗く．積まれているクラスとセレクタがそれぞれ C と S に等しければメソッド部が探していたものである．そうでなければ，第**1**章で述べた通常の探索を行い，結果を返す前にハッシュ表を更新する．

メソッドキャッシュを用いることにより，メソッド探索のコストは，ヒット時で，ハッシュ表へのアクセス1回のそれとなる．Berkley Smalltalk での測定結果によると，エントリ数が512のメソッドキャッシュでヒット率は90%を超えている[5]．

3.2　インラインキャッシュ

メッセージ送信ごとに存在する長さ1のキャッシュは，**インラインキャッシュ**（inline cache）と呼ばれ，「1つのメッセージ送信に注目すると，概ね前回と同じメソッドが呼ばれる」という実行特性を利用したものである[9]．主に実行系が仮想マシンで動的コンパイルを行うものに用いられる．

具体的に述べると，メッセージ送信（に相当するバイトコード）は，前述の探索ルーチン lookup への call 命令（マシンコード）へとコンパイルされる．ただしその際1ワードの空コードも生成され（**図-1**上），これが call 命令のアドレス部とともにキャッシュ領域を形成する．最初の実行では lookup ルーチンが呼び出され，そこでメソッドが探索され，キャッシュが書き換えられ，メソッドが起動される（**図-1**下）．

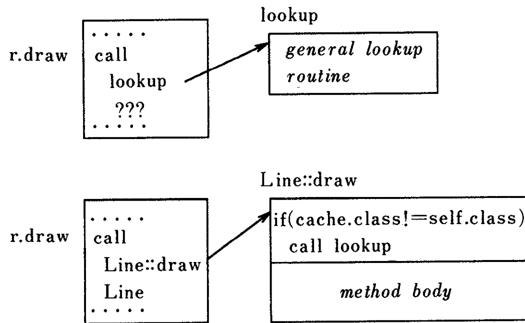


図-1　インラインキャッシュ

キャッシュに書き込まれるのはメソッドとレシーバクラスであるが，前者については正確には図-1に示すようなメソッドのプロローグのアドレスである．インラインキャッシュを実装したシステムでは，プロローグはメソッドごとに自動的に生成され，キャッシュがヒットしているか，つまり，現在のレシーバのクラスが前回と同じであるか否かが調べられる．

インラインキャッシュを用いることにより，メソッド探索のコストは，キャッシュヒット時で，1つの**型検査**（type test）のコストとなる．Smalltalk-80での測定結果によると，インラインキャッシュのヒット率は95%以上となっている[9]．

3.3 PIC

インラインキャッシュを動的に伸長可能にしたものが，**PIC**（Polymorphic Inline Cache）である[16]．メッセージ送信はlookupルーチンへのcall命令にコンパイルされ，初回の実行で「キャッシュ領域」が生成され，続く実行で新たなレシーバクラスに出会うたびに「キャッシュ領域」は伸長する（**図-2**）．この領域は**PICスタブ**と呼ばれ，本来の意味でのキャッシュ領域と型検査ルーチンが一体となったものである．

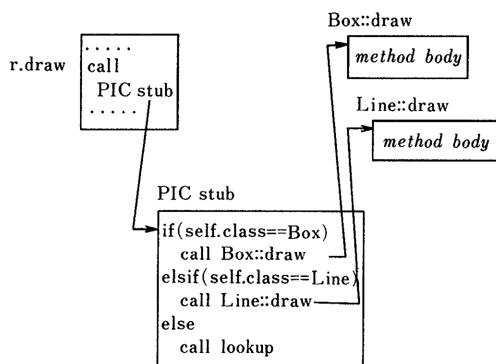


図-2　複数の型検査のあるPICスタブ

レシーバクラスが交互に変わるような場合，インラインキャッシュではキャッシュミスが大きくなり性能が低下してしまうが，PICではそうはならない．また，呼出先が小さなメソッドならば，PICスタブ中へインライン展開することもできる．

第5章でみるように，直接束縛法による高速化技法の1つでレシーバクラス予測と呼ばれるものでは，PICが巧妙に利用されている．

4. ディスパッチ表法

ディスパッチ表法の基本的なアイデアは，「クラスとセレクタの組合せに対してあらかじめメソッド探索を行っておき，結果をディスパッチ表と呼ばれる配列に格納しておく」というものである．メソッド探索のコストは配列の参照のそれに減らすことができる．

ディスパッチ表法では，通常クラスごとに1つのディスパッチ表が作られ，メッセージ送信は次の疑似Cコードと同様なものにコンパイルされる．

```
(*receiver->
    dispatchTable[CODE (selector)])
        (receiver, arg1, arg2, …);
```

ここで，CODE関数はセレクタに番号（コードともいう）を付す関数で，ディスパッチ表法の実装において鍵となるものである．

C++やJavaのような静的型言語では，簡単なCODE関数でメモリ効率よくディスパッチ表を組立てることができるため，この技法が広く使用されている．動的型言語では，単純な方法ではメモリ効率が悪くなるためディスパッチ表法は使用されてこなかったが，実用に向けていくつかの改善策が提案されている．

4.1 静的型言語への適用

静的型言語では，継承パスにしたがって上位クラスのセレクタから順に付番してゆけばよい．継承パスのそれぞれは独立に付番してよい．たとえば，**図-3**の階層には3つの継承パスがあり，それぞれのパスでセレクタは**図-4**のように付番される．

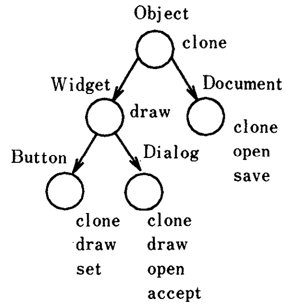


図-3　クラス階層の例

path to Button		*path to Dialog*		*path to Document*	
clone	0	clone	0	clone	0
draw	1	draw	1	open	1
set	2	open	2	save	2
		accept	3		

図-4　静的型言語でのセレクタ付番例

セレクタ open には，Dialog へのパスでは2番が，Document へのパスでは1番が振られている．メッセージ open の送信式をコンパイルする際にいずれの番号を用いればよいか困りそうだが，コンパイラはレシーバの静的型を知っているため正しく判断することができる．

4.2　動的型言語への適用

前節で述べた付番は，動的型言語ではうまくいかない．動的型言語のコンパイラは，open の送信式をコンパイルするとき，どの継承パスの付番を用いればよいか決定できないからである．動的型言語の CODE 関数は，同じセレクタには同じ番号を付すものでなくてはならない．

最も単純には，すべてのセレクタを一意に付番すればよいのだろうが，セレクタ数の大きさのディスパッチ表がクラス数だけつくられることになる．今日の商用の Smalltalk システムには，1,000 を超えるクラスと 10,000 を超えるセレクタがあり，メモリ使用量は極端に大きくなり実用にならない．

ところで，大きなシステムで一意付番によりディスパッチ表を構成した場合，表のエントリはほとんど使用されていない．1つのクラスが理解できるセレクタの集合は，システム全体のごく一部であるからである．

もっと濃密に表を構成することができれば，メモリ使用量を減らすことができる．実際，そのよ

codes	*selectors*
0	clone
1	draw save
2	set　open
3	accept

図-5　セレクタ彩色の例

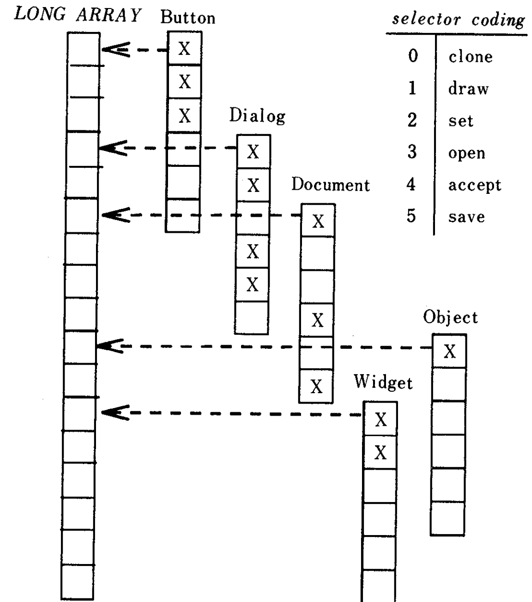


図-6　行転位圧縮の例

うな技法として，**セレクタ彩色法**（selector coloring）[13]と**行転位圧縮法**（row displacement compression）[10]が知られている．

セレクタ彩色法では，ある2つのセレクタを同時に理解できるクラスが存在しなければ，これらのセレクタに同じ番号をふる．たとえば，図-3 のクラス階層は，**図-5** のように付番する．実際のアルゴリズムは，その名が示唆するようにグラフの彩色問題に還元されるが，クラス階層の規則性をうまく利用して最適に近い彩色を効率よく求めることができる．

一方，行転位圧縮法では，（たくさんの）疎なディスパッチ表は1つの長い配列に互いに重ならないように詰め込まれる．たとえば，図-3 の階層で一意付番により構成される5枚のディスパッチ表は，**図-6** のように圧縮される．このような圧縮は，一般には NP 完全問題になるが，ディスパッチ表の場合には，やはりクラス階層の性質を利用したうまいヒューリスティクスがある．

ここで，2つの圧縮法の効果を，IBM社のSmalltalkシステムであるVisualAge 2.0でみてみよう．単純な一意付番では表の充塡率は1.9％以下であるが，彩色法や行転位圧縮法ではそれぞれ43％と55％になる[11]．

ところで，行転位法には面白い変形がある．クラスごとではなくセレクタごとにディスパッチ表をつくり行転位圧縮するというもので，充塡率を大幅に上げることができる[11]．その理由は，行転位圧縮のアルゴリズムが「少数の長い配列」よりは「多数の短い配列」を好むからである．実際，この変形をVisualAge 2.0に適用した場合，充塡率は99％を超える．

しかし，ディスパッチ表のメモリ使用量を考えてみると，充塡率100％の場合ですら，VisualAge 2.0で7.7Mバイトに達し仮想イメージを倍以上膨張させてしまう．したがって，これらの圧縮法の実用性には，まだ疑問が残る．

まったく異なるアプローチとして，キャッシュ法とディスパッチ表法を複合した**CISCO法**(Caches Indexed by Selector Codes)[21]というものもある．簡単にいえば，「送信されるセレクタ」と「受信するクラス」に関する局所性を利用して，少数の小さなディスパッチ表をキャッシュのように使い回すというもので，メモリ使用量は小さな定数に抑えられる．

最後に，時間効率について付言すると，彩色法でも行転位圧縮法でもCISCO法でも，**セレクタミスマッチ**（selector mismatch）という現象が発生し，これに対処するため，少しの余分な処理が必要になる．紙幅が限られているため，ここでは詳しく解説できないが，たとえば文献21）を参照されたい．

5．直接束縛法

なんらかの方法でレシーバの型を絞り込んで，コンパイル時にメソッド探索を行ってしまおうというのが**直接束縛法**で，メッセージ送信はメソッドを直接呼び出すコードにコンパイルされる．

1つの型に絞り込むことができれば，メソッド探索のコストは完全になくなる．少数の型ならば，生成されるコードはPICスタブと似た形になり，メソッド探索のコストは，1回ないし数回の型検査のそれになる．

直接束縛法の強みは，メッセージ送信がメソッドの直接呼出しになるだけでなく，インライン展開が可能となることである．つまり，メッセージ送信の頻度の問題にも対処することができるのである．さらにもっと重要なことは，実際にインライン展開することにより，共通部分式除去や定数畳込みをはじめとする多彩な最適化の機会が生ずることである．

このように，直接束縛法はキャッシュ法やディスパッチ表法に比べ段違いの潜在能力をもっているため，動的型言語の性能を静的型言語のそれに近づける切札として，Selfプロジェクトの研究者を中心に精力的に追求されてきた．

さて，直接束縛法を行うには，型情報を獲得しなくてはならない．代表的な方法としては，静的解析によるもの，実行履歴に基づくもの，メソッドのカスタム化によるものがある．

5.1 クラス解析

メソッド中の変数や式の取り得る値のクラスの集合，すなわち**型集合**を，静的に計算することを**クラス解析**という．メッセージ送信の高速化の見地からは，レシーバの解析結果が重要となる．

解析には局所的なものと広域的なものがあるが，後者は研究が進行しつつある分野で本格的な展開はこれからであろう．ここでは，局所的なクラス解析と，それを補完する階層解析と呼ばれるものについて説明する．

5.1.1 局所クラス解析

処理中のメソッドの（ソース）コードだけをみる局所的なクラス解析は，通常のデータフロー解析でよく，簡単に実装することができる．リテラルやオブジェクト生成式（C++やJavaではnew式）などを情報源に，変数から型集合への写像をつくり，制御フローグラフにそって伝搬してやればよい．

制御フローの合流点では，それぞれの変数に対して，入力フローでの型集合の合併がとられ，出力フローでの型集合となる．ここで1つ問題になるのは，上流では特定的だった型情報がしばしば合流点で失われ，下流のメッセージ送信を最適化する機会もまた失われてしまうということである（図-7左）．これを解決しようというのが**メッセージ分裂**（message splitting）[3]という技法で，合流点からメッセージ送信までのコードを押し上

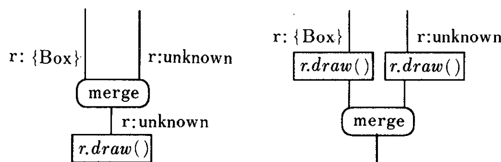


図-7　メッセージ分裂の例

げ，入力フローのそれぞれにコピーする (**図-7**右).

次に，局所解析の限界について議論したいのだが，まず記法を導入しておこう．クラス X のみからなる型集合を $Class(X)$，「クラス X とそのサブクラスすべて」からなる型集合を $Cone(X)$ と表すことにする．また，後者を錐集合と呼ぶことにする．

静的型言語の型宣言において変数の型がクラス X であるという場合，その型集合は $Class(X)$ ではなく $Cone(X)$ になる．また，動的型言語といえどもメソッドのレシーバの型についてはコンパイル時に判明しており（メソッドの定義クラス），やはり錐集合になる．

局所クラス解析の限界は，レシーバの型集合を錐集合（または錐集合から集合演算で構成されるもの）に絞り込んでも，メッセージ送信を直接束縛することができないことである．静的型言語での豊富な型宣言を活かすことができず，また，メソッドのレシーバへはメッセージが高頻度で送信されるにもかかわらず，動的型言語でも静的型言語でも最適化することができない．

この限界を打破するには，クラス階層を解析する必要があり，次項の話題となる．また，**5.3** 節で説明するカスタム化は，この限界を回避しようというものである．

5.1.2　階層解析

メッセージ m の送信式において，レシーバの型集合が $Cone(X)$ であることがわかっているとする．このとき，X のサブクラスで m を再定義しているものがいなければ，この送信式は，クラス X およびセレクタ m でメソッド探索を行った結果に束縛することができる．

この判定は局所解析の範囲を越えているが，プログラム全体のクラス階層を調べればわかり，**階層解析** (hierarchy analysis)[8]という．広域解析の一種であるが，「どんなクラスがありどんなメソッドを定義しているか」だけに興味があり，メソッドの本体は関係しない．

具体的にいうと，m という名前のメソッドでプログラム中にあるものそれぞれについて，定義しているクラスと継承しているクラスからなる**適用集合** (applies-to set) を計算する．そして，レシーバの型集合との交わりが空でないものが1つだけあれば，そのメソッドに束縛することができる．

図-3の例でいえば，メッセージ clone の送信式を最適化したい場合は，clone をセレクタにもつすべてのメソッド，すなわち，Object :: clone, Button :: clone, Dialog :: clone, Document :: clone の適用集合を求めることになる．ここで，Object :: clone についてだけ値を示すと，{Object, Widget} となる．

判定はコンパイル中に頻繁におこるため，高速に行わなくてはならない．Cecil で講じられている方法は，型集合や適用集合はビットベクタで表現し，適用集合はオンディマンドで計算し，さらに判定結果はキャッシュに積んでおくというものである[8].

5.2　レシーバクラス予測

コードの最適化の定石の1つに「よくある場合を特別に処理する」というものがあるが，これをメッセージ送信の最適化に応用したのが**レシーバクラス予測** (receiver class prediction)[15),18)] である．たとえば，メッセージ送信式 r. draw( )をコンパイルするときに，「レシーバのクラスはほとんどの場合 Box である」と予測できるならば，次のようにコンパイルすることができる．

```
if (r. class==Box)
  Box : : draw( );
else //general case
  r. draw( );
```

高頻度のレシーバクラスに対してはメソッド探索は1つの型検査ですんだことになり，メソッド探索のコストは平均的に下がる．そして，インライン展開の可能性が開ける．

予測の根拠となるのは**実行履歴**（プロファイル情報）である．具体的には，メッセージ送信点ごとの「レシーバのクラス分布」で，これを基に最適化を行う．

ところで，このような最適化が可能ないし有効であるためには，メッセージ送信において1つないし少数のレシーバクラスが支配的である必要が

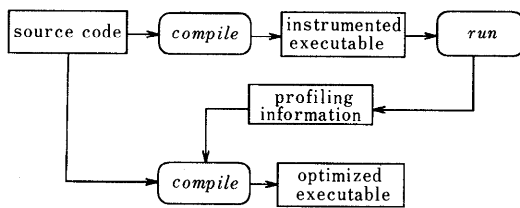


図-8　静的コンパイル系でのレシーバクラス予測

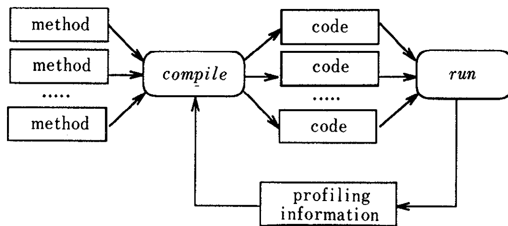


図-9　動的コンパイル系でのレシーバクラス予測

ある．Cecil と C＋＋における計測によると，それぞれの送信点で最も支配的なレシーバクラスに対して送られたメッセージの（動的）割合は，それぞれ 72％と 71％であったという[15]．したがって，レシーバクラス予測による最適化の適用機会は十分多いといえる．

次に，実装について少し詳しく説明する．履歴の収集と返送（フィードバック）が鍵になるが，これらをどのように構成するかは，静的コンパイル系と動的コンパイル系ではかなり違ったものになる．

5.2.1　静的コンパイル系での実装

静的コンパイル系では，図-8 のように 3 段階の構成になる．第 1 のコンパイルでは履歴出力のコードがついた実行ファイルが生成される．次にこれを走らせて履歴ファイルを得る．第 2 のコンパイルでは，この履歴を基にレシーバクラスの予測が行われ，最適化された実行ファイルが出力される．

2 度のコンパイルと 1 度の実行が最適化のために必要で，処理のオーバヘッドは大きいといえる．したがって，プログラム開発時も受信クラス予測による高速化を享受したい場合は，履歴は間欠的に（たとえば週 1 回程度）取得し，普段のコンパイルではこれを代替として繰り返し利用することが考えられる．

このような履歴の代替利用が成立するためには，異なるバージョンの履歴がかなり“似ている”必要がある．Cecil での実験では，類似性はかなり高いことが観測されている[15]．

5.2.2　動的コンパイル系での実装

動的コンパイル系では，図-9 のような構成になる[18]．特徴的なのは，再コンパイルはメソッド単位に行われ，まだ最適化すべきメソッドが残っているかぎり繰り返される点である．いわゆる**適応コンパイル**（adaptive compilation）で，システムは部分的に徐々に最適化されてゆく．

Self における実装[18]を基にもう少し具体的に説明する．Self では，巧妙にもレシーバのクラス分布を PIC から獲得している．つまり履歴の取得に余分なオーバヘッドが時間的にも空間的にもかからない．PIC には頻度情報はないが，レシーバクラス予測にはしばしば十分である．

コンパイルの起動時機については，メソッドには実行カウンタがついており，最適化されていないメソッドのカウンタがある閾値を超えたら，コンパイラが起動されるようになっている．

しかし，コンパイルの対象の決定は単純ではない．起動の契機となったメソッドだけを最適化コンパイルすればよいかというと，必ずしもそうではない．たとえば，次の C＋＋プログラムで考えてみると，カウンタが先に溢れるのは find であろうが，再コンパイルすべきは lookup であろう．

```
Symbol * lookup (char * name) {
  ...
  for (int i=0; i<SIZE; i++)
    if (symtab[i]&& (s=symtab[i]->find
      (name)))
      ...
}
Symbol * SymbolList::find (char * name)
  {...}
```

したがって，スタックを遡り，呼び出したメソッドを調べる必要がある．Self では，未最適化メソッドへの呼出しを「多く」含んでいるものをみつけ，そこまでのメソッドをすべて再コンパイルの対象としている．

5.3　カスタム化とスペシャル化

第 **5.1** 節で述べたように，メソッドのレシーバの型集合は錐集合になり，メソッドの定義は下位クラスすべてにわたって通用する．つまりコー

ドの共有が促進されているわけだが，反面，レシーバの型情報が薄められているため，コードの最適化は抑制される．

この問題に対処しようというのが**カスタム化**(customization)[4]で，特定のクラス専用のメソッドを自動生成しようというものである．具体的には，「クラス X で定義されたメソッド M」と「*Clone*(X) の要素のクラス Y」に対して，M のレシーバの型集合を *Class*(Y) に制限したものを自動生成するのである．これによりレシーバへのメッセージ送信をコンパイル時に束縛することが可能となる．

この自動生成についてもう少し詳しく説明すると，Satherでは，サブクラスの（静的）コンパイル時に上位クラスから継承したメソッドをすべてカスタム化する[20]．Selfではメッセージ送信の実行の際にカスタム化が起こり，メソッド探索でみつかったメソッドが，レシーバのクラスに対して（まだカスタム化されていなければ）カスタム化される[4]．

ところで，カスタム化をいくつかの点で改善した**選択的スペシャル化**(selective specialization)[7]というものがある．スペシャル化では，レシーバの型のみならず，引数の型も制限の対象になり，引数へのメッセージ送信もコンパイル時に束縛されるようになる．また，カスタム化でもすでにコード量の増大が問題視されているが，制限対象の増えたスペシャル化ではこれは一層深刻な問題となる．そのため，gprof風の実行履歴に基づいて高速化への寄与が大であるものを選択的にスペシャル化するようになっている．

5.4 インライン展開の難しさ

直接束縛法の大きな潜在能力はインライン展開に存するが，その実装にはいくつかの課題があり容易ではない．

第1の課題は，「何をインラインすべきか」という点で，インライン展開はコード量と実行時間のトレードオフである以上，やみくもに展開してゆくわけにはいかない．しかも，RISCマシンなどでは命令キャッシュの大きさの関係で，コード量も増えるし実行も遅くなることもある．

簡単な決定法は「呼出先のソースコードが小さければインライン展開する」というものだが，展開の対費用効果をもう少しまじめに評価する必要があるだろう．たとえば，展開後の最適化の効果まで見積もるため，呼出点のコンテキストで**試しに**インライン展開を行うというものがある[6]．対費用効果が小さければ実際に展開はしない．試し展開を反復することによるコンパイル時間が増えるのを防ぐため，試し展開したならば「呼出点での静的情報」と「対費用効果」をデータベースに登録する．そして，別な送信点で同じ呼出先の展開を判断するときには，このデータベースをまず検索にいくのである．

第2の課題は「デバッグへの対応」である．単純な処理系では開発中はインライン展開そのものを禁止しているが，開発中といえども「高速な実行」と「ソースコード上でのデバッグ」の2つを同時に享受したいであろう．1つの有力な対処法が**脱最適化**(deoptimization)[17]と呼ばれるもので，デバッガ起動時に現在スタックに積まれているフレームを最適化前のコードに対応するように変換するのである．当然ながらこれは1対多の変換になる．

第3の課題は「プログラムの変化への対応」で，呼出先のソースコードが変わった場合インライン展開を**無効化**し呼出元を再コンパイルしなくてはならない．すべてを無効化し再コンパイルするという単純な解法が許されないならば，依存グラフを作成して対処することになる．

以上みたように，インライン展開の本格的な実装は，候補選択や脱最適化や無効化の実装までも含む，相当の労力を要する作業になる．

6. C++仮想関数呼出しの高速化

これまでに説明した技法を，最も広く使われてきたオブジェクト指向言語であるC++の観点からみてみよう．先に述べたように，C++ではディスパッチ表による実装が一般的で，ほどほどの高速化を簡単に達成することができる．

ディスパッチ表法以外では直接束縛法のいくつかが試みられている．まず，階層解析は，C++ではリンク時の最適化という位置づけになり，そのアドホックな変形は，以前より知られていた．しかし，本格的に追求されるようになるのは最近になってからで，レシーバクラス予測と組み合わせてC++の仮想関数呼出しをできるだけ取り除こうという研究が複数みられる[1),22]．また，カス

タム化については，Selfでの研究を受けて，C++でも劇的な効果があることが1990年という早い段階に示されている[19].

ところで，C++の仮想関数呼出しの高速化は，間接分岐命令の高速化というアーキテクチャ上の見地から議論することもできる[2),12)]. 最近のプロセッサでは，いわゆる投機実行により分岐先の命令まで含めた命令レベルの並列実行が可能となっている．このようなプロセッサでは，分岐を正しく予測することができれば，仮想関数呼出しのオーバヘッドはかなり低くなるのではないかと期待されている．

分岐先バッファ (Branch Target Buffer) は，間接分岐命令の分岐予測ハードウェアで，これにはいくつかのバリエーションがあるが，基本的な考えは，分岐命令のアドレスをキーに最新の分岐先を格納しておこうというものである．インラインキャッシュを彷彿させるが，分岐先バッファには，2回続けてミスしたらバッファを書き換えるというものがあり，インラインキャッシュ以上のヒット率がシミュレーションで観測されている[2].

条件分岐の予測には，**予測テーブル** (prediction table) がある．これにもバリエーションがあるが，たとえば，分岐命令のアドレスをインデックスに「分岐がとられたか否かの1ビット」が格納される．プロセッサのなかには，予測テーブルはあるが分岐先バッファのないものもあり，そのようなプロセッサに対しては，レシーバクラス予測を用いた最適化が有効になる．

7. おわりに

メッセージ送信の高速化技法を，キャッシュ法，ディスパッチ表法，直接束縛法の3つに類別して解説した．このなかで，直接束縛法が最も大きな潜在能力をもっているが，キャッシュ法やディスパッチ表法にも実装の単純さという大きな利点がある．紹介した技法のほとんどは，動的型言語で追求されてきたものである．なかんずく，SelfとCecilにおける成果が質量ともに水際立っている．C++ではディスパッチ表による実装が一般的だったが，これらの動的型言語での成果の影響を強く受けながら，直接束縛法に基づく高速化が本格的に追求されつつある．また，プロセッサのアーキテクチャの面からも研究が進みつつある．こうした流れはJavaに引き継がれていくのであろう．

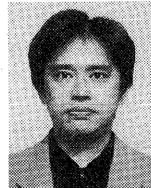
謝辞　東京工業大学の脇田健助手，日本アイ・ビー・エム東京基礎研究所の石崎一明研究員より，本稿に対して貴重なコメントをいただいた．感謝します．

参考文献

1) Aigner, G. and Hölzle, U.: The Direct Cost of Virtual Function Calls in C++, ECOOP '96 Conference Proceedings, pp. 142-166.
2) Calder, B. and Grunwald, D.: Reducing Indirect Function Call Overhead In C++ Programs, Proceeding of the 21 st Symposium on Principles of Programming Languages 1994, pp. 397-408.
3) Chambers, C. and Ungar, D.: Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs, Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, pp. 150-164.
4) Chambers, C. and Unger, D.: Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language, Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, pp. 146-180.
5) Conroy, T. J. and PelegriLiopart, E.: An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations, In Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, pp. 239-247 (1983).
6) Dean, J. and Chambers, C.: Towards Better Inlining Decisions Using Inline Trials, Proceedings of the SIGPLAN '94 Conference on Lisp and Functional Programming.
7) Dean, J., Chambers, C. and Grove, D.: Selective Specialization for Object-Oriented Languages, Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation, pp. 93-102.
8) Dean, J., Grove, D. and Chambers, C.: Optimization of Object-Oriented Programs Using Static Hierarchy Analysis, Proceeding of ECOOP' 95.
9) Deutsch, L. P. and Schiffman, A.: Efficient Implementation of the Smalltalk-80 System, Proceeding of the 11 th Symposium on Principles of Programming Languages 1984, pp. 297-302.
10) Driesen, K.: Selector Table Indexing & Sparse Arrays, OOPSLA '93 Conference Proceedings, pp. 259-270.
11) Driesen, K. and Hölzle, U.: Minimizing Row Displacement Dispatch Tables, OOPSLA '95

Conference Proceedings, pp. 141-155.
12) Driesen, K. and Hölzle, U.: The Direct Cost of Virtual Function Calls in C++ OOPSLA '96 Conference Proceedings To appear.
13) Dixon, R., McKee, T., Schweizer, P. and Vaughan, M.: A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance, OOPSLA '89 Conference Proceedings, pp. 211-214.
14) Goldberg, A. and Robson, D.: Smalltalk-80: The Language and Its Implementation, Addison-Wesley (1983).
15) Grove, D., Dean, J., Garrett, C. and Chambers, C.: Profile-Guided Receiver Class Prediction, OOPSLA '95 Conference Proceedings, pp. 107-122.
16) Hölzle, U., Chambers, C. and Ungay, D.: Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, ECOOP '91 Conference Proceedings, pp. 259-270.
17) Hölzle, U., Chambers, C. and Ungay, D.: Debugging Optimized Code with Dynamic Deoptimization, Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation, pp. 32-43.
18) Hölzle, U. and Ungay D.: Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback, Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation, pp. 146-180.
19) Lea, D.: Customization in C++, Proceeding of 1990 USENIX C++ Conference, pp. 301-314.
20) Lim, C-C. and Stolcke, A.: Sather Language Design and Performance Evaluation, International Computer Science Institute, Berkeley, TR 91-034 (Apr. 1996).
21) Onodera, T. and Nakamura, H.: Optimizing Smalltalk by Selector Code Indexing Can Be Practical, IBM Tokyo Research Laboratory, RT 0145 (Apr. 1996).
22) Porat, S., Bernstein, D., Fedorov, Y., Rodrigue, J. and Yahav, E.: Compiler Optimization of C++Virtual Function Calls, Proceeding of the USENIX 1996 Conference on Object-Oriented Technologies.

(平成8年9月12日受付)



小野寺民也（正会員）
1959年生．1983年東京大学理学部情報科学科卒業．1988年同大学院理学系研究科情報科学専門課程修了．理学博士．同年日本アイ・ビー・エム(株)入社，現在東京基礎研究所勤務．プログラム言語処理系，オブジェクト指向プログラミングなどの研究に従事．本学会第41回全国大会学術奨励賞，平成7年度山下記念研究賞受賞．著書に「A Formal Model of Visualization in Computer Graphics Systems」(共著，Springer-Verlag)がある．日本ソフトウェア科学会，ACM，IEEE，USENIX各会員．