OpenJIT コンパイラフレームワークにおける実行時特化システム

丸山直也[†] 増原英彦[†††] 小川宏高[†]
丸山冬彦[†] 松岡聡[†,††]

現在我々は OpenJIT コンパイラフレームワーク [16,19] を用いて Java 言語の実行時特化システムを設計、構築している。増原 [15] らによるバイトコード特化は、実行時特化を仮想機械上で行ない、バックエンドとして JIT コンパイラを用いることによって、実行時特化で従来問題であった特化されたコードの質を改善しているが、特化器と JIT コンパイラとの間に直接的なインタフェイスがないため、特化器の実行以外に大きなオーバーヘッドがかかってしまっている。我々は、特化器が生成するコードを JIT コンパイラの中間表現とし、それを直接 JIT コンパイラに渡すことによってこの問題を解決し、さらに OpenJIT のフレームワークを用いることにより、高い可搬性と一般性を備えたプログラムを実行環境に自動的に最適化することを実現する。

Using OpenJIT compiler framework [16,19], we are now designing and developing a runtime specializer for Java. Bytecode specialization (BCS) [15] is a technique to specialize programs w.r.t. runtime values. It generates programs in a bytecode, and then translates the generated bytecode into native code by employing JIT compilers. Thanking to the optimization of JITs, BCS can improve the efficiency of the generated code, compared to previous runtime specialization techniques. Owing to lack of any direct interfaces between specializer and JIT compiler, however, the current BCS costs huge runtime overhead. To solve this problem, we plan to generate specialized code as an intermediate representation of our JIT compiler and pass it to the JIT directly. And furthermore, we plan to specialize more portable and/or general Java programs automatically, by using the compiler framework of OpenJIT.

1. はじめに

1.1 背景

近年, オブジェクト指向プログラミングやデザインパターン [7] などを用いたソフトウェアエンジニアリング、さらにはプログラムをネットワークから動的にダウンロードしての実行など、一般性, 可搬性に富んだソフトウェアの重要性が高まっている. しかし、そのようなプログラムではその性質のために、実行環境に依存するような最適化技術を適用させることはできない。また、デザインパターンを用いたコンポーネントベースのプログラムには、各コンポーネント間のインターフェイスを高度に抽象化することによって動的なプログラムの変化に対応できるようになっているが、同時にその抽象化によるメソッドディスパッチなどのオーバーヘッドが問題になってしまっている。この問題はコンパイル時に得られる情報からは解決できないため、従来の静的コンパイラで用いられている最適化技術では対処できない。

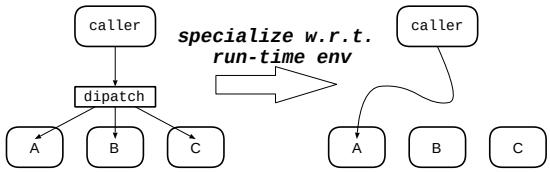

図 1 メソッドディスパッチの除去

プログラム特化とは, あるプログラムから特定のコンテキストに特化したプログラムを自動生成する技術である. 上記の問題を、プログラムの実行時にプログラム特化を行い、実行コンテキストに特化させることによって解決できる (図 1) ため、近年プログラムの最適化に関する研究対象として注目されている [3,4,9,13,15,18,20,22–24]。

実行時プログラム特化とは, プログラム特化の一種であり, プログラムの実行時に得られる情報を用いた特化を実機械語上で実行時に行なう技術である. 一般に、コンパイル時にソースコードを解析し (束縛時解析, BTA [12]), 実行時定数を引数として呼びだされるとその値に特化されたプログラムを生成するプログラム generating extension [12] を作成する。実行時

† 東京工業大学
†† JST
††† 東京大学

には generating extension に実行時定数を渡して特化を行い、もとのプログラムの代わりに用いる。もう一つのプログラム特化である, コンパイル時に得られる情報をもとに特化を行う部分計算 (partial evaluation) [11] に比べて, 実行時情報を用いることができるためより効果の高い特化を行え高速なプログラムを生成することができる. しかし一方で, 特化は実行時に行われるためそのコストはすべて実行時間に対するオーバーヘッドとなる。そのため特化されたプログラムの実行時間と特化に要する実行時間のトータルで高速化されなければ意味がなく, 以下の関係式を常に満たす必要がある.

特化に要する時間+特化されたプログラムの実行時間
< もとのプログラムの実行時間

従って、実行時特化の適用は、特化のコストとそれによる高速化のトレードオフを適切に考慮しなければならない。コストの高いコード生成を行ってしまうと、それによるオーバーヘッドを回収することが困難になってしまい, 逆にコストを抑えることを重視し過ぎると、さらなる高速化の機会を失ってしまう。

従来の実行時特化システムでは、コード生成に関する上記の問題に対して有効な回答を示していない。というのは、その多くがどんなプログラムに対しても一律な最適化を行なうだけであり、adaptive に手段を選択するようなことを行なっていないためである。

例えば, Tempo [3], DyC [9], 'C + VCODE [20] では, 常に局所的な最適化を施すだけである. この場合は高速にコードを生成するが, その質が問題になる. 逆に, 'C + ICODE [20] では常に大域的な最適化を適用する. この場合は, 生成物の質が問題になることはないが, それによる高速化がコード生成のオーバーヘッドを償却できるとは限らない.

また上述のように, 従来の実行時特化ではコンパイル時にソースコードを解析する必要がある. 従って, プログラムの特化はそのプログラムのユーザが明示的に行うことになり, ユーザに対する負担となる. さらに当然, ソースコードが入手可能なプログラムにしか特化できない. つまり, 実機械語プログラムとして配布されるライブラリや, ネットワークからダウンロードされたプログラムの特化は行えない.

以上の問題より, 従来の実行時特化システムはユーザがあらかじめ, プログラムのどの部分にどのようなコード生成の戦略をもって特化を適用すればよいか分かっている場合にのみ, 用いることのできるものであった. 冒頭で述べた従来の静的な処理系が成し得ない最適化をより一般的に行うためには, ユーザ対して暗黙的であり, かつコード生成のトレードオフを解決した実行時特化システムが必要である.

1.2 データとしてのプログラム

特化をユーザに暗黙的に適用するには, 少なくともコンパイル時に何らかの処理が行えることを仮定すべきではない. すなわち, 従来の特化システムではコンパイル時に束縛時解析を行っていたが, これに相当することを実行時に行うようにする必要がある. 従って, 特化システムが実行時にプログラムを解析し, 特化されたプログラムを動的に生成し, それを起動する仕組みが必要である. これは特化システム側から見れば, プログラムをデータとして扱うことを意味する.

従来の実行時特化システムが前述のような問題を抱えている理由は, 特化するプログラムが実機械上で実行されるものであるため, プログラムを実行時にデータとして容易に扱えないことによる. すなわち, (1) 実行時に特化対象プログラムとして得られるものは実機械語プログラム, (2) 特化されたプログラムは実機械語プログラム, でなければならない. 通常, 束縛時解析はコントロールフローグラフなどを構築して行うが実機械語より構成するのは困難である. また, (2) のために, 特化器のバックエンドとして動作する実機械コード生成器が必要になる. 前述のように, 従来の実行時特化システムでは, このコード生成器はコード生成のトレードオフを解決するものでなかった. さらに, 当然その実機械に強く依存したシステムとなってしまう.

1.3 バイトコード特化とその問題点

従来の実行時特化の問題を解決するには, 特化の対象となるプログラム (言語) が, (1) 特化システムが入手可能, (2) 解析が (実機械語のように) 困難でない, (3) 可搬, でなければならない. さらに, コード生成のトレードオフが解決可能であることが望ましい.

一般に, インタプリタ上や仮想機械上で実行されるプログラムは可搬であると言える. 例えば, Lisp や ML, Java bytecode などでは, 機種独立なプログラム形式を用いることによって可搬性を確保している. また, 特化システムをインタプリタや仮想機械に組み込んだ形で実現すれば, プログラムを特化システムが入手可能であることも保証される. もしくは, その言語が何らかのリフレクティブな機能を有し, それによってプログラムを入手できるならば, 独立した (スタンドアローンな) システムとしても実現できる. また, これらの言語は機械語のように複雑ではなく, 解析も比較的容易に行えることが期待できる.

増原らによって提案されているバイトコード特化 [15] は、実機械上ではなく仮想機械上で実行時特化を行う技術である。特化システムは, コンパイル時にバイトコードで記述されたプログラムを束縛時解析し generating extension をバイトコードで生成する. この generating extension は, 実行時に特化されたプログラムをバイトコードで生成する. 特化システムへのインターフェイスに機種独立なバイトコードを用いることによって, システムの可搬性を確保している. さらに, 通常仮想機械ではバイトコードで記述されたプログラムを, JIT コンパイラを用いて動的に実機械語にコンパイルして実行するが, 特化システムによって生

成されたプログラムも JIT コンパイラによって実機械語にコンパイルされてから実機械上で実行される. 従って, 特化システムは特化のみを行い, コードの最適化は JIT コンパイラに委ねることができる.

現在主に Java 言語 [8] を対象とした JIT コンパイラの研究 ([1,2,17] など) が盛んであり, 例えば、コストが高いが効果も大きい最適化をプログラムの 'hot spot' にだけ施す adaptive optimizaion [2] や、インタプリタと高速だがあまり最適化を行わない JIT コンパイラ、低速だが高度に最適化された機械語を生成する JIT コンパイラの 3 つを使いわける mixed mode bytecode execution [1] などがなされている. これらの JIT コンパイラでは, 最適化の段階を設けた JIT コンパイルを行うことによって, 上記のトレードオフを解決できることが示されており、このような技術は実行時特化におけるコード生成においても有効であることが期待できる。実際に JVM (Java Virtual Machine) [14] を仮想機械として選択した [15] では, 従来の実行時特化より効率の良いプログラムが生成できることが確かめられている.

一方, 増原らによる現状のバイトコード特化システムでは, Java プログラムの束縛時解析器が ML で記述されている. ML プログラムを Java プログラムから実行時に呼び出すことは現実的ではないため, 実行時束縛時解析は不可能である. また, 特化に要するコスト, すなわち generating extension の起動と生成されたプログラムの JVM へのロードの実行時間において, 後者が全体の 90%程度占めてしまっている問題がある. これは, 特化されたプログラムをクラスファイルとして生成し, Java の実行環境, すなわち JVM, JIT にロードさせているためである.

またさらに, 従来の実行時特化でその利用の手間が大きな問題であったように, バイトコード特化でも上述のような generating extension の起動, クラスロードなどはプログラマが明示的に行わなければならず, 利用者に対する負担が大きい. また, 明示的に処理を行わなければならないことは, プログラマの感知しないメソッド, すなわちライブラリなどの外部からは隠蔽されたメソッドの特化は起こりえないことを意味する. 特化の機会を最大限に活かせるシステムとするには, 任意のメソッドをプログラマの干渉なしに暗黙的に特化できることが望ましい.

1.4 解決策の考察

従来のバイトコード特化では, 特化システムを JVM, JIT の外側に置くことによって, 任意の JVM, JIT コンパイラ上での動作を可能にしているものであった. しかしその一方, 特化システムと JVM, JIT の間にデータ (クラスファイル) の明示的な受け渡しが必要になってしまっているため, 上述のような問題を引き起こしている.

この問題を解決するためには, まず特化システムを Java 言語の中で実現することが最低条件である. そのうえで, Java 言語のどの部分で実現するかについて, (1) Java アプリケーションとして実現する, (2) JIT に組み込まれたシステムとして実現する, (3) JVM に組み込まれたシステムとして実現する, の 3 つの選択肢が考えられる. 以下, それぞれについて考察する.

Java アプリケーションとして (1) は ML で書かれた既存のバイトコード特化システムを Java 言語に移植し, Java アプリケーションとして利用できるようにするものである. 従来と同様, JVM とのインターフェイスには通常のクラスファイルを用い, JVM, JIT には依存しない. 従来と異なる点は, 束縛時解析器を Java プログラムとすることにより, 実行時束縛時解析が可能になる点である. また, Java 言語は Reflection API として標準でリフレクティブな機能を有し, 未知のクラスが持つメソッドのシグネチャを知ることなどが可能になっている. しかし, この API ではクラスファイル自体を得ることはできないため, 束縛時解析の対象とするクラスについて, ファイル I/O などによってそのクラスファイルを入手する必要がある. Java では一般にクラスパスで指定された位置 (ディレクトリ) に存在するクラスが JVM にロードされるが, そのようなクラスファイルはクラスパスを探索すれば特化システムにもロードできる. しかし, ネットワークからバイト列としてダウンロードされクラスロードされたクラスは, 束縛時解析を行うまでそのバイト列を保持するようにプログラムを変更する必要がある. また, 後述の手法ではクラスロードを省略することでそのオーバーヘッドを失くせることが見込めるが, アプリケーションレベルで実現するこの手法では JVM, JIT とのインターフェイスにクラスファイルを用いざるを得ず, クラスロードは必須である. 従って, オーバーヘッドを失くすことは不可能である.

JIT に組み込む (2) は, バイトコード特化を JIT コンパイラが行う最適化の一つとして実現するものである. 理想的には, 特化のプロセス (束縛時解析, generating extension の起動など) を全て JIT コンパイラが管理し, プログラマからは通常のプログラムの実行が, 特化されたプログラムの実行にプログラマに暗黙的にスイッチされるような実行形態が望ましい. また, (1) では束縛時解析の際に特化システムが独自にクラスファイルをロードする処理が必要であったが, JIT コンパイラでは JVM からその情報を得ることができるため, この方式では必要ない. さらに, JIT コンパイラの内部で行われるため, 特化システムと JIT とのインターフェイスをクラスファイルとする必要はなく, JIT コンパイラ固有の中間表現を用いることができる. そのため, 従来のバイトコード特化シ

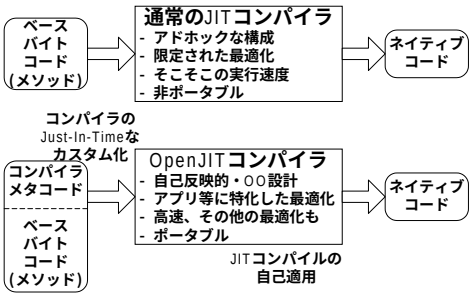

図 2 従来の JIT と OpenJIT の比較の概念図

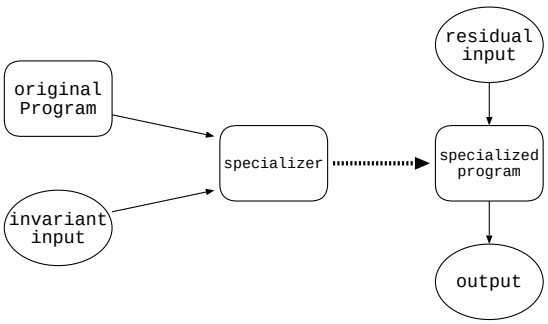

図 3 プログラム特化

ステムで問題であった, クラスロードのオーバーヘッドの削減が期待できる. 一方, JIT コンパイラに組み込むことにより, (1) で保たれていた特化システムの可搬性が損なわれてしまう恐れがあるが, JIT コンパイラ自身が可搬なものであれば問題ない. すなわちこの方式では, 特化システムを容易に組み込め, 可搬な JIT コンパイラが必要である.

JVM に組み込む (3) は, JVM にバイトコード特化の機能を組み込むものである. (2) と同様, クラスファイルを得るのに特別な仕組みは必要なく, JVM の内部で処理を行うためクラスロードのオーバーヘッドの削減も可能ではある. しかしまた, 特化システムを容易に組み込め, 可搬な JVM が必要になることも同様であり, 現実的には機種独立な JVM というのは非常に困難であるため, 従来の問題に対する解決策としてこの方式は適していない.

1.5 目　　的

上記 3 つの手法のなかで, (1) は移植するだけで良いため比較的容易に実装可能と思われる.

しかし, 3 つの解決策のなかで, 最も適しているのは (2) の JIT コンパイラに組み込む手法である. (2) の手法を実現するのに必要な拡張性高く可搬な JIT コンパイラとして, OpenJIT [16, 19] があげられる.

我々のグループで開発されている Java 言語 の JIT コンパイラである OpenJIT は, リフレクションに基づいた Open Compiler 技術をベースとして, アプリケーションや計算環境に特化した言語の機能拡張と最適化が行なえる JIT コンパイラである. また, プログラムの殆どを Java 言語で記述することによって可搬性を確保している. このような性質より, バイトコード特化を組み込むのに適した JIT コンパイラと言える.

そこで本研究では, まず (1) の手法である Java アプリケーションとしてのバイトコード特化システムを実装する. これを用いて, (2) の手法である, バイトコード特化を OpenJIT に組み込んだ場合の予備的な性能評価を行う. またこれは (2) のシステムに容易に再利用できるようにする. さらに, OpenJIT に組み込み込んだシステムの実現に向けた考察を行う.

以降の構成 まず 2 節で従来の実行時特化特化とその問題点を述べ, 3 節で実装したバイトコード特化システウの説明, 性能評価を行う. さらに, 4 節で OpenJIT に組み込んだバイトコード特化に向けた考察を述べ, 5 節で関連研究を紹介する. 最後に 6 節でまとめと今後の課題を述べる.

2. プログラム特化

部分計算に始まったプログラム特化は, 現在はより多くの最適化の機会を持つ実行時特化 (RTS) が中心となっている. この節では, 実行時特化一般とその問題点を提起する.

2.1 実行時特化

プログラム特化とはプログラムの最適化技術の一つである. あるプログラムが繰り返し実行され, その入力の一部が固定されている時, もとのプログラムと同じ結果を返す, その固定入力にカスタマイズされたプログラムを生成する. 固定入力 (以降, 静的な入力, もしくは静的な束縛時を持つと言う) にのみ依存する計算は特化時に行ってしまうので, 生成されたプログラムはもとのプログラムに比べて効率が良くなることが期待できる (図 3).

プログラム特化は, 束縛時解析と特化の 2 つの処理に分けられる. それぞれは以下の仕事をする.

束縛時解析 プログラムを特化時に計算できる静的なパートと, 実行時に計算しなければならない動的なパートにわける.

特化 束縛時解析で得られた情報をもとに, 特化時に静的なパートを計算し, 動的なパートはそれに対応するプログラムを生成する.

特化を行った際, 特化されたプログラムはもとのプログラムと同じ結果を返すものでなければならない. 例えば, x と y を引数にもつプログラム prog を $x = a$ について特化して生成されるプログラムを prog_x とすると, $\mathrm{prog}(a, y) = \mathrm{prog}_x(y)$ を満たさなければならない. 一般に実行時特化 [3,9,13,20] は, ソースコードに

対して 束縛時解析をコンパイル時に行ない, 特化を実行時に行なう. 束縛時解析によって動的とされたパートは実機械語にコンパイルされる. この際, 静的な値を用いる式は “hole” [3] とする. さらに, 実行時に静的なパートを実行し, テンプレートの hole を埋め, それらを適当に組みあわせて特化された実行可能プログラムを生成するプログラム (generating extension) を作成する. この generating extension は束縛時解析したプログラム専用の特化器と見ることができるため, 直感的には特化するプログラムに特化された特化器と言える. 実行時に generating extension に静的な入力を渡すと, その値に特化された実機械語プログラムが生成される. また, プログラムの実行時に静的とする入力がわかれば良いので, 部分計算と比較してより多くの特化の機会を得ることができる.

2.2 従来の実行時特化の問題点

特化の効果を決定づける大きな要因は, 束縛時解析と generating extension と言うことができる. 束縛時解析については, それによって静的と判断された部分のみ特化時に計算できるため, より多くのパートを静的とできる解析が望ましい. これはその手法の違いより, offline か online, monovariant か polyvariant と区別され [12], それぞれ後者の方がより多くのパートを静的とできる可能性がある. また他には, flow sensitive な解析 [10] などが提案されている.

generating extension については, その実行に要するコストと生成するコードの質について, 適切にバランスをとる必要がある. generating extension は特化するプログラムに特化した特化器であるので, 特化自体に要するコストは比較的小さい. しかし, 生成されるコードの質がその後の特化されたプログラムの実行効率に大きく影響するため, コード生成の戦略がドミナントになる. コード生成には上述のように, テンプレートを用意しておき, それを再配置することによるもの ([3] など) の他に, 一度コントロールフローグラフを構成し, 一般的な静的コンパイラで用いられている最適化を施すもの ([20]) がある.

テンプレートベースのコード生成では, テンプレートの機械語列をコンパイル時に用意しておくことで, そのオーバーヘッドを少なく抑えられるが, テンプレートを越えるような大域的なレジスタアロケーションや命令スケジューリングは行えない. そのため, 生成される機械語列の実行効率が問題になってしまう [15].

一方, コントロールフローグラフの構築を行うコード生成では, 大域的な最適化を施すことが可能になり, より高速なプログラムを生成できる. しかしながら, それによるコストはテンプレートベースの特化と比べて大変大きなものとなってしまい, 特化による高速化によって補うことが困難になってしまう.

このようにコード生成はそのコストと生成物の質のトレードオフを考慮しなければならないが, 従来の実行時特化システムではコストを抑えることを重視するか, 生成物の質を重視するかのどちらかであった. つまり, 実行時により多く実行されるメソッドについてはコストの大きいコード生成を行い, そうではないメソッドについては高速なコード生成を行うといった, adaptive に手段を選択するようなことを行っていない. そのため, 従来の実行時特化システムでは, 全体的にさらなる高速化の機会を逃してしまっているか, または逆に特化のコストを補えずに実行効率が下がってしまっている.

3. スタンドアローンバイトコード特化システム

2.2 節で述べたように従来の実行時特化では, そのコストと生成物の質のトレードオフなどが問題になる。本節ではこの問題を解決する要素技術である Java 言語のバイトコード特化システムを Java アプリケーションとして実装した特化システムの概要を説明する。さらにその性能評価を行なう。

3.1 概　　要

現状のバイトコード特化システムでは, メソッドの引数の一部を静的として束縛時解析し, その引数に特化されたメソッドを生成する.

まず初めに, 特化するメソッドとそこから呼び出されるメソッドを束縛時解析する. 束縛時解析の結果, Java 言語のメソッドとして generating extension を生成する. ここまでの処理はコンパイル時に行っておくことができ, その場合の方が束縛時解析, generating extension の生成などの実行時オーバーヘッドを省け効率が良い. また, この際の束縛時解析器への入力, 出力ともにクラスファイルを用いる.

次に, generating extension に実行時定数を渡して, 特化されたメソッドをインスタンスメソッドとして持つクラスを作成する. クラスは byte 型の配列として生成し, 標準のクラスローダを用いて JVM にロードした後, そのインスタンスを生成する. このインスタンスメソッドの起動を通して特化されたメソッドを実行できるようになる.

前述したように, バイトコード特化では, 特化されたプログラムが実際に実行される前に, JIT コンパイラによって最適化された機械語プログラムへ変換されることを仮定している (図 4). この際の JIT コンパイラが adaptive に最適化を適用するものであるならば, 従来の実行時特化で問題であったコード生成のトレードオフを解決できる.

3.2 実装の現状

システムの実装には, クラスファイルの読み込みに OpenJIT 2 のクラスファイル解析ライブラリを用い, 生成に BCEL [5] を用いた。扱えるクラスファイルは JVML の, 1) データ型は int かその配列のみ, 2) メ

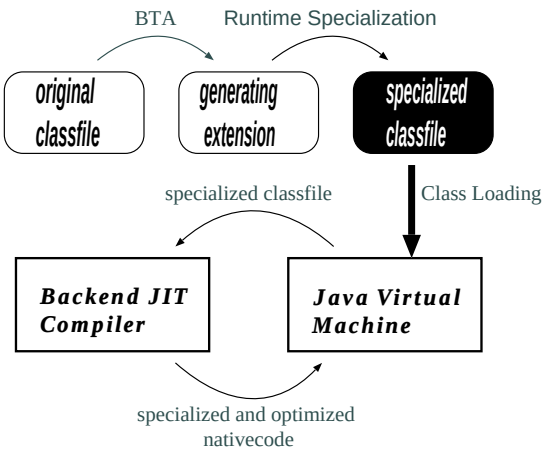

図 4 バイトコード特化の実行図: 白抜の四角は実行時に生成されることを示す

```
static int power(int x, int n) {
    if(n == 0){
        return 1;
    }else{
        return x * power(x, n - 1);
    }
}
```

図 5 メソッド power

ソッドはすべてクラスメソッド, 3) サブルーチン, 例外処理は行えない, ものである. また, 動的な分岐の副作用を打ち消す制約を加えていないため, 動的な分岐を扱うことはできない。また, エイリアス解析も行っていないため, 配列を用いる際にはエイリアス解析を必要とするプログラムの正確な束縛時解析は行えない。

3.3 実 行 例

我々が実装したバイトコード特化システムを用いて, 実際にバイトコード特化を行うには 2 種類の方法がある.

(1) コンパイル時に束縛時解析をし, generating extension を静的に用意しておく. 実行時にその generating extension を呼び出すことによって特化されたメソッドが生成される.

(2) 束縛時解析 を含む全ての処理を実行時に行う.

以下ではそれぞれについて, 前述の `power` を用いて実際の処理の概略を説明する.

3.3.1 束縛時解析をコンパイル時に行っておく方法

まず, コンパイル時にメソッドの引数の束縛時を指定して束縛時解析し, generating extension を作成する。generating extension はクラスファイル`GenExt.class` として生成される。`GenExt.class` は特化されるメソッドの generating extension を持ち, さらにそれらの起動のエントリとなるメソッド `specialize` を持つ。`power` から生成される `GenExt.class` の `javap` の出力を図 6 に示す☆。但し, メソッド `specializePower(int)` が `power` の generating extension である。また, Java 言語は静的に型付けされているので, 特化によって実行時に生成されるメソッドのシグネチャが Java コンパイラから分からなければならない。そのため, generating extension の生成と同時に特化されたメソッドと同じシグネチャのアブストラクトメソッド`residual` を持つアブストラクトクラス`SpecializedPower.class`(図 7) を生成する。

実行時には, 実行時定数を渡して generating extension を実行し, 特化されたメソッドを持つクラスを生成する。このクラスはコンパイル時に作成してある`SpecializedPower` を継承し, 特化されたメソッドが`residual` メソッドを実装する。次に, このクラスをロードし, `residual` メソッドにメソッドの残りの引数 (動的な引数) を渡すと, 通常のメソッド呼び出しと同じ結果が得られる。以上の処理を行うプログラム断片を図 8 に示す。このプログラムの `classGen.getClass` メソッドがクラスロードにあたる。

3.3.2 全ての処理を実行時に行う方法

バイトコード特化を暗黙的に行うことを考えた場合, 束縛時解析がプログラムの実行前に行ってあることを仮定できない。そのため, 束縛時解析と generating extension の作成を含めた全ての処理を実行時にも行えるようにすべきである。しかし, 増原による既存のバイトコード特化システムは ML で実装されているため, Java プログラムの実行時に ML のプログラムである束縛時解析器を呼び出すことは現実的ではなかった。しかし, 本研究では Java 言語を用いて実装しているため, 実行時に特化システムを呼び出すことは, 通常のメソッド呼び出しと同様に行える。Java は静的に型付けされている言語なので, シグネチャが静的に分からない, 実行時に生成されるメソッドを呼びだすことは, 一般的な手法では不可能であるが, Java 言語が有するリフレクションの機能 (`java.lang.reflect` パッケージ) を用いることによって可能になる。付録 A.1 にリフレクションを用いて図 8 に相当することを行うプログラムを示す。

3.4 性 能 評 価

図 9 にあるメソッド dotProduct を, 片方の引数を長さ 100 で 90%が 0 の整数ベクトルとして特化する場合の性能評価を行う. 計測には JNI を用いて `gettimeofday` を呼び出して求めた. テスト環境は PentiumIII 840MHz, 192MB メモリ, Linux 2.2.14, Sun JDK1.2.2 Classic VM, OpenJIT1.1.15 である.

☆ クラス `de.fub.bytecode.*` はクラスファイル操作ライブラリ BCEL [5] のクラスである。

```
public class GenExt extends java.lang.Object {
    private static de.fub.bytecode.generic.InstructionList runtimeIL;
    private static de.fub.bytecode.generic.ConstantPoolGen runtimeCP;
    public GenExt();
    // メソッド power の generating extension
    private static void specializePower(int);
    // generating extension のエントリとなるメソッド
    public java.lang.String specialize
     (int, de.fub.bytecode.generic.ConstantPoolGen,
              de.fub.bytecode.generic.InstructionList);
}
```

図 6 power から生成される generating extension を持つクラス

```
public abstract class SpecializedPower extends java.lang.Object {
    public SpecializedPower();
    // 特化されたメソッドが実装する抽象メソッド
    public abstract int residual(int);
}
```

図 7 特化されたメソッドと同じシグネチャのアブストラクトメソッドを持つクラス

```
static int dotProduct(int[] x, int[] y) {
  int i = 0;
  int result = 0;
  int length = x.length;
  while(i < length){
    result += x[i] * y[i];
    i++;
  }
  return result;
}
```

図 9 メソッド dotProduct

3.4.1 特化されたメソッドの性能

コンパイル時に束縛時解析を行っておき, 実行時に初めに generating extension を実行し, 特化されたメソッドを繰り返し呼んだときにかかる時間と, 通常のメソッド呼び出しにかかる時間を求め比較した (図 10). 特化を行った場合の実行時間には, generating extension の実行, 特化されたクラスファイルのロードなどのオーバーヘッドを含む☆ . グラフの unspecialized が通常の dotProduct の実行時間である。specialized は 1 回のみ行う特化に要した時間と特化されたメソッドの実行時間の和である。この結果によると, 特化のコストを償却するには, およそ 20 万回も実行しなければならない。

expected は特化に要した時間から, generating extension の実行以外を除いた時間を特化されたメソッドと実行時間と足した時間である。すなわち, バイトコード特化を JIT の内部で実現した際に達成できると予測される実行時間である。これによると, JIT の内部で実現することによって, 特化のコストをスタンドアローン特化器よりかなり少い実行回数で償却できることが期待できる。

3.4.2 特化のコスト

同様にして特化に要する時間を計測した。比較のために JDK1.2.2 のインタープリタ, JDK1.3 HotSpot を用いた時間も計測した。結果を図 11 に示す。

図から分かるように, generating extension の実行時間 (specialize メソッドの実行時間) は, 全体の 5%程度でしかない。つまりそれ以外の特化されたメソッドを持つクラスの生成, ロード, インスタンス生成の実行時間が, 特化時におけるドミナントな処理であると言える。

3.4.3 実行時束縛時解析を行なった場合の性能

3.4.1 節と同じ実験を実行時束縛時解析で行った. プログラムは付録 A.1 と同様のプログラムであり, これ

☆ generating extension が生成するクラスファイルは byte 型の配列として生成され, 直接 JVM にロードされる. ファイルとして書き出すことはしていない.

```
// generating extension をメソッドとして持つオブジェクトの生成
GenExt genExt = new GenExt();
ClassGen classGen = new ClassGen("Power");
SpecializedPower specializedPower = null;
try{
    // generating extension の起動
    sig = genExt.specialize(3, classGen.constantPool(),
                            classGen.instructionList());
    // クラスの生成, ロード
    Class specializedClass = classGen.getClass(sig);
    // インスタンス生成
    specializedPower
        = (SpecializedPower)specializedClass.newInstance();
}
catch(Exception exception){
    exception.printStackTrace();
    System.exit(1);
}
// 特化されたメソッドの起動
specializedPower.residual(5);
```

図 8 バイトコード特化を行って 5 の 3 乗を計算するプログラム

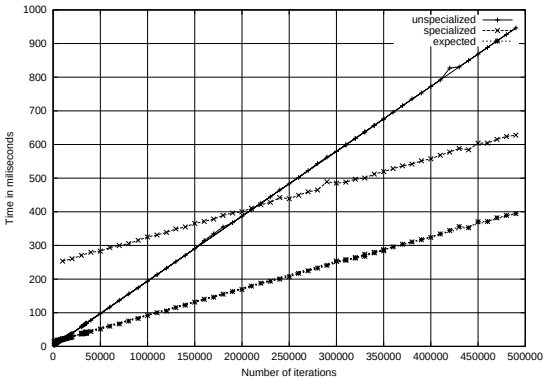

図 10 dotProduct の実行時間: グラフ specialized は特化器の実行と, それ以外のオーバーヘッド含んだ全体の実行時間であり, expected は特化器の実行時間と特化されたコードの実行時間の和で, クラスロードなどのオーバーヘッドをなくせた場合の予測である.

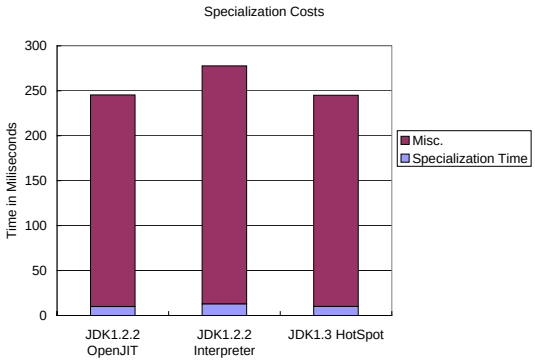

図 11 特化のコスト: Specialization Time が generating extension の実行に要する時間を表し, Misc がその他の実行時間を表す.

の束縛時解析☆, 特化 (クラス生成, ロードを含む), 実行の 3 つに区切って実行時間を測定した. 図 12 に通常の実行時間と特化されたメソッドの実行時間 (束縛時解析, 特化の実行時間は含めていない) を載せ, 表 1 に束縛時解析と特化の実行時間を載せる.

表 1 実行時束縛時解析と特化に要する時間

	実行時間 (ms)
束縛時解析	87.45
特化	259.01

☆ 現状の実装では, 束縛時解析によって生成されるクラスファイルはファイルとして生成される. すなわち, generating extension とは異なり, 配列として生成し直接 JVM に渡すことはしていない. これは将来拡張する予定である.

この方式では, 付録 A.1 のプログラムを見て分かるように, 通常のクラスメソッド呼び出しが, 複雑なオーバーヘッドの多い呼び出しになっている. そのため, 図 12 のように特化されたメソッドが通常のメソッドより遅いものになってしまっている. すなわち, 実行

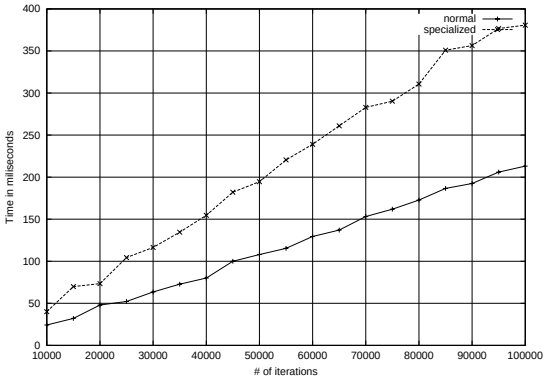

図 12 実行時束縛時解析を行った場合の特化されたメソッドの性能: normal が通常の実行時間で specialized が特化されたメソッドの実行時間である. 特化されたメソッドの実行時間には束縛時解析と特化の実行時間は含めていない.

時束縛時解析では `dotProduct` をどんなに多く実行しようと, リフレクションを用いることによるオーバーヘッドを回収できない. 従って, `dotProduct` よりアグレッシブに特化できるメソッドではリフレクションのオーバーヘッドを償却できる可能性があるが, その可能性は大変小さいことが予想できる. しかし, 実行時束縛時解析自体のコストはそれほど大きくはなく, メソッドの起動にリフレクションを必要としない JIT コンパイラに組み込む手法ではそのコストを十分償却できることが予測できる.

4. OpenJIT コンパイラフレームワークを用いたバイトコード特化

この節では OpenJIT を紹介し, OpenJIT に組み込んだバイトコード特化の実現に向けた考察を述べる.

4.1 OpenJIT 1

OpenJIT 1 [16, 19] は Sun の Classic VM に対する add-on として JIT コンパイラを実現している. 現在対応しているプロセッサは IA32 アーキテクチャと SPARC アーキテクチャである. OpenJIT は従来型のコンパイラ技術とは異なる, 自己反映計算（リフレクション）の理論に基づいた “Open Compiler”（開放型コンパイラ）技術をベースとした JIT コンパイラである. 最も大きな特徴はコンパイラの大部分が Java 言語によって記述されていることで, この特徴はリフレクションを用いてアプリケーションや計算環境に特化した言語の機能拡張や最適化を可能とすることを目指したものである. しかし, これまでの成果で得られたものは主に実行効率と安定性を重視しているために拡張性や再利用性に乏しく, 当初の目標であった, JIT コンパイラの研究基盤としての役割を果たせていない.

4.2 OpenJIT 2

現在我々は, OpenJIT 1 では解決していない JIT コンパイラの構築基盤の提供という課題を解決するために, OpenJIT 2 を開発している. OpenJIT 2 は, JIT コンパイラの各部をオブジェクト指向の技術を用いたアプリケーションフレームワークとして設計および実装することで, 新たな JIT コンパイラの構築や新たな最適化技術の導入, コンパイル時に行う最適化の選択, 異なるアーキテクチャへの対応といった様々な目的に応じた JIT コンパイラの構築基盤として利用可能である.

Java 言語に限らず, あるプログラムに最適な最適化の戦略 (例えば, 「コストがかかるが, 実行効率の良いコードを生成する」など) は, 一般に別のプログラムには通用しない. また, 同じプログラムであったとしても, 計算環境が異なれば異なる戦略をとる必要がある. そのため, アプリケーションに特化した最適化を施す種々の JIT コンパイラを複数同時使用できることが望ましい. しかし, そのような特化した JIT コンパイラの構築コスト, 同時使用によるメモリ使用量, などが問題となるため, 従来の JIT コンパイラでは ad-hoc に小数の JIT コンパイラを組み合わせる程度でしかない [1, 2].

OpenJIT 2 では, OpenJIT 1 とは異なり, JIT コンパイラ向けアプリケーションフレームワークを提供する. これにより, アプリケーションに特化した種々の JIT コンパイラの複数同時使用が実現可能になる. すなわち, フレームワークに沿って実装することにより, 特化した JIT コンパイラの実装の大部分が共有可能であり, その開発コストも抑えられる. また, 共有により, 複数同時使用によるメモリ使用量も抑えられる.

このように実装することは, 従来の monolithic な JIT コンパイラと比較して性能における問題点になることが予想できる. しかし, その初期の予備的な評価では, 最適化コンパイラとしては悪くはない性能を示しており, 今後の開発で従来の JIT コンパイラ並の性能が期待できる.

OpenJIT 2 は現在開発段階であり, JIT コンパイラとして使用できるものに至ってはいないが, その拡張性, 再利用性により, 容易にバイトコード特化を組み込めることが期待できる. 従って, 我々は バイトコード特化を組み込む JIT コンパイラとして OpenJIT 2 を用いることにする.

4.3 バイトコード特化に向けて

3.4.2 節で述べた, 特化器と JIT コンパイラとの間のオーバーヘッドは, 特化器が OpenJIT の中間表現を生成し直接 OpenJIT に渡せば解決できる (図 13). これにより, 従来の実行時特化より効率の良いコードを生成でき, さらに従来のバイトコード特化に比べて, 特化に要するコストを 10 分の 1 程度に抑えられるようになる.

しかしながら, 単純に現状の特化器と OpenJIT を結ぶだけでは, もう 1 つの課題である「プログラマが

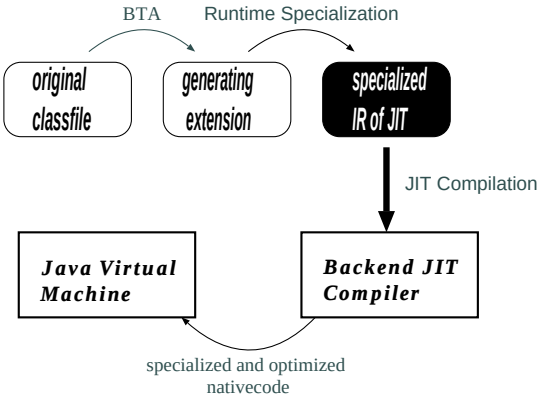

図 13 改良されたバイトコード特化

特化システムを使う手間の軽減」は実現できず, 明示的に特化の処理を行わなければならない. 以下では, JIT コンパイラの最特化の 1 つとしての暗黙的な特化の実現に向けた考察を述べる.

特化の隠蔽 現状では, 3.3 節で述べたように, 特化を行うにはプログラマが明示的に束縛時解析, generating extension の起動, 特化されたメソッドの JVM へのロードを行わければならない. このように手間がかかるようなシステムでは JIT の最適化として実用的に用いることはできず, プログラマ, あるいはそのクラスのユーザに暗黙的に特化が行われるようにしなければならない.

このようにするには, OpenJIT 側で特化対象のメソッドを束縛時解析しておき, そのメソッドの JIT コンパイル時に generating extension の起動と, 特化されたコードの実行にコンパイルすることによって実現できる. 特化のプロセスは JIT コンパイラによって管理されるので, プログラマから隠蔽でき, ソースコードにもバイトコードにも変更を加えなくて済む.

また, すでに生成されている特化されたコードが再利用できるならば, それを用いたほうが特化のオーバーヘッドを無くすことができて良い. そのためには, 何らかの特化されたコードをキャッシュしておくバッファーが必要である. これも JIT コンパイラ側で管理し, プログラマが明示的に行うようにすべきでない (図 14).

さらに, メソッドの引数の束縛時の複数の組み合わせに対してそれぞれ generating extension を用意し, メソッドのコンパイルドコードとして, それらを適切に選択するようにすれば, polyvariant な束縛時解析 [12] も可能である.

特化するメソッドと初期束縛時の決定 バイトコード特化はソースコードを必要としないので, 多数のクラスファイルから構成されるライブラリでも特化の対象とできる. しかしそのような大きなプログラムにおいて, どこをどの程度特化すべきかをユーザが判断するのは非常に困難である. また, 特化をプログラマから隠蔽するためには, 何らかの手段で特化するメソッドとその引数の束縛時をシステム側で決定しなければならない.

```
                    if (existsInCache(power, n)){
                      cachedSpecializedCode(x);
power(x, n)   ⇨     } else {
                      power_gen(n);
                      specializedCode(x);
                    }
```

図 14 JIT コンパイラによる暗黙的な特化

HotSpot [17] のような最近の JIT コンパイラでは, 実行時プロファイル情報をもとにコンパイルするメソッドを決定する. このような手法は特化対象とするメソッドやその引数の束縛時を選択する際にも有効であり, JIT コンパイラのなかに特化器を組み込むことによって, JIT コンパイラのプロファイル情報を共有できることも期待できる.

また, 性能上の見地より, 特化するメソッドとその引数の束縛時がコンパイル時にわかっている場合は, 静的な情報として特化システムに渡せるべきである. 従来の実行時特化では, ソースプログラムに何らかのアノテーションを加える方法をとることが多かったが, そのようにオリジナルのプログラムに変更を加えることは, それらの管理などプログラマに手間がかかり望ましくない.

そこで, specialization class [23] などのように特化対象のプログラムとは別にアノテーションを記述し, 特化システムに渡すようにすればよい.

特化システム側に渡す手段としては, クラスファイルの属性領域に付加する方法が考えられる. ユーザが specialization class を付加するクラスのソースコードを持つ場合は, その機能を持つ特殊な Java 言語コンパイラを利用することで可能になる. クラスファイルのみ有する場合は, クラスファイルにそれを付加するクラスファイル変換ツールを用いることで解決できる. また, JVM の仕様により JVM, もしくは JIT コンパイラが認識しない属性は無視される. 従って, このようにクラスファイルに特殊なアノテーションを加えてしまうことで, クラスファイルがもともと有する可搬性が失われることはない.

5. 関 連 研 究

実行時特化システム 既存の実行時特化システムは, その実行時コード生成のバックエンドによって, 生成されるコードの質が大きく異なる. バックエンドが大域的な命令スケジューリングやレジスタ割り当てなどを行なえば, 生成されるコードの質が高まる

が, 特化にかかるコストがより大きくなる. 一方, バックエンドが局所的にコードを生成するのであれば, そのコストを低くおさえることができるが, 生成されるコードは, レジスタの使用の衝突など, あまり最適化されていないものとなる.

既存の実行時特化システムとしては, Tempo [3], Fabius [13], 'C [20], DyC [9] などがある.

Tempo は部分計算にもとづいた C 言語の実行時特化システムである☆. コンパイル時に, 一般の C コンパイラによって生成されたコードから自動的にテンプレートを作成し, 実行時に特化器がこのテンプレート単位でコードを生成する. しかし, テンプレートを越えるような, 大域的な命令スケジューリングやレジスタ割り当ては行なっていない.

Fabius は, 代入などの副作用を生じる機能を排除した ML のコンパイラで, カリー化された関数を, generating extension の起動と特化されたコードを呼び出すコードにコンパイルする. 関数の引数の順序を用いて多段階の特化を行なっている. 実行時特化器は 1 パスでコードを生成するため, そのコストは低いが最適化は行なわれていない.

'C は C 言語に実行時コード生成のためのコンストラクトを追加した言語である. 実行時特化器は VCODE [6] または ICODE [20] のマクロを生成するようになっており, 特化のコストと生成物の質のトレードオフを考慮して選択できる☆☆. しかし, 実行時コード生成の管理とその正しさの保証がすべてプログラマに委ねられているため, 特化を強力に制御できる一方, 実際に用いるのが困難になってしまっている.

DyC はプログラムの一部にアノテーションを加えることによって, 実行時特化を行なうシステムである. また, プログラマは特化の度合を制御するポリシーを指定することや, polyvariant な特化, 動的な分岐におけるオンデマンドな特化も可能である. しかし, 有効な特化を行なうには, プログラマによるプロファイリングなどによって最適なアノテーションを見つけなければならない. 特化されたコードになされる最適化は, すべて静的に解析された情報によるものだけであり, 局所的なものしかなされない.

以上のように, バックエンドが大域的な最適化を行なうものは少ない. これは, それにともなうオーバーヘッドを上まわる高速化を得ることが困難であったためである. 一方, バックエンドで行われていることは JIT コンパイルとも言える. 現在主に Java 言語の JIT コンパイラについての研究が盛んであるが, そこで用いられている技術を実行時特化システムに用いる, または JIT コンパイラの最適化として, 実行時特化を用いた研究はなされていない.

Java 言語の特化システム Consel らは Tempo を用いて Java 言語の特化システム [22] を開発しており, 特化の指示与えるための形式である specialization class [23] を提案している. これは, それ自体がクラス階層をなしており, 自然な形でインクリメンタルな特化を可能にしている. しかし, プログラムは特殊な変更を施した Harissa [18] と呼ばれる JVM 上で実行しなければならないため, JIT コンパイラを用いた評価はされていない.

☆ 処理をフロントエンドとバックエンドにわけることによって, コンパイル時特化も可能となっている [4].

☆☆ VCODE は局所的にコードを生成し, ICODE は一度中間表現にした後, 大域的な最適化を施す.

6. おわりに

6.1 まとめ

実行時特化は, プログラムの実行時に得られる情報について実行時にプログラムを特化させる技術である. 一般に可搬性と一般性を備えたプログラムは実行効率が問題になるが, そのプログラムの実行時に実行環境に特化させることによって解決することができる. しかし従来の実行時特化システムは, プログラムをデータとして容易に扱えないため, コード生成のトレードオフに対処できていなく, また実際のプログラムへの適用性も低いものであった.

プログラムをデータとして扱える実行時特化として, バイトコード特化がある. バイトコードを特化の対象とすることにより, プログラムをデータとして扱い易くなり, コード生成のトレードオフの解決を JIT コンパイラに委ねることができるようになる. また, 特化システムを JVM, JIT の外側に置くことによって任意の JVM, JIT コンパイラ上での動作を可能にしている. しかしその一方で, 特化システムと JVM, JIT の間にデータ (クラスファイル) の明示的な受け渡しが必要になってしまっているため, クラスロードのオーバーヘッドやその手間などの問題が生じてしまっている.

そこで我々は, バイトコード特化を OpenJIT の最適化の一部として実現することを目指している. JIT の内部で処理を行うことによりクラスロードを省け, 特化に要するコストを 10 分の 1 程度に削減できる. また, JIT コンパイラが自動的に特化を行う機械語プログラムにコンパイルすることにより, プログラマに暗黙的にバイトコード特化を行えるようになる.

本研究では, その前段階として Java 言語によるスタンドアローンなバイトコード特化システムを実装した. これは完全に pure-java で記述し, その一部を OpenJIT が提供するライブラリを用いており, 将来の OpenJIT を用いたバイトコード特化システムに容易に拡張可能である. OpenJIT に組み込んだ場合の予備的な評価として, その性能評価を行った. それによると, 特化のコストを上回る高速化は困難であるが, JIT コンパイラと特化器とのインターフェイスを改善

することによって十分解決できることがわかった.

また, Java 言語で記述することにより, 従来の ML によるシステムでは現実的ではなかった実行時束縛時解析が行えるようになった. しかし, Java 言語は静的に型付けされているため, 実行時に生成されるメソッドを用いるには, コンパイル時にそのメソッドと同じシグネチャである抽象メソッドを作成し, 特化されたメソッドが実装するようにするか, リフレクションを用いなければならない. 実行時束縛時解析を行った場合は, コンパイル時にその抽象メソッドを作成しておくことはできないためリフレクションを用いざるを得ない. しかし, そのオーバーヘッドを償却できず特化されたメソッドがもとのメソッドよりも遅くなってしまっている. だが, JIT コンパイラの内部で実現すれば特化されたメソッドの起動にリフレクションが必要になることはないため, この問題も他の問題と同時に解決可能である. 一方, 実行時束縛時解析自体のコストはそれほど大きくなく, 実行時に束縛時解析を行うことは十分現実的であることがわかった.

6.2 今後の課題

今後の最も重要な課題としては, 実行時特化の有効性を確かめるためにも OpenJIT に組み込んだバイトコード特化を実現することである. また, 現状では実装面での課題がいくつか残されている. 例えばオブジェクトを扱うプログラムの特化は行えない. 藤波による [24] では, C++ 言語において, オブジェクトのメソッドをそのフィールドに対して特化する手法が提案され, 良好な性能を示している. そのような手法はそのまま我々の特化システムでも実現できると思われる. さらに, オブジェクトや配列を扱う際の BTA で必要になるエイリアス解析を行なっていない. C 言語の特化器である DyC は, ユーザが明示的に示すことによって解決している. 同じく C 言語の特化器である Tempo では, エイリアス解析を行なっているため, ユーザによるアノテーションは必要ない. 実行時特化の自動化を達成するには, Tempo のような手段が必要である.

特化の応用としては, 単にメソッドの引数について特化するのではなく, デザインパターンを用いたプログラムの構造に着目した特化を行う specialization patterns [21] がある. これにより, 抽象度の高いプログラムを計算環境に特化して効率良く実行できるようになるため, 応用として大変重要なものである.

付　　録

A.1 実行時束縛時解析を行うバイトコード特化での 5 の 3 乗を計算するプログラム

```
BTAnalyzer analyzer = new BTAnalyzer();
// Power.class のメソッド power を 1 番目の引数を
// 動的, 2 番目を静的として束縛時解析
analyzer.bta("Power.class", "power", "DS");
// generating extension をメソッドとしてもつクラス
// GenExt をロード
Class genExtClass = Class.forName("GenExt");
Object genExt = genExtClass.newInstance();
ClassGen classGen = new ClassGen("Power");
// generating extension に相当するメソッド
// specialize の引数の型
Class[] paraTypes = new Class[]
 {Integer.TYPE,
  Class.forName
    ("de.fub.bytecode.generic.ConstantPoolGen"),
  Class.forName
    ("de.fub.bytecode.generic.InstructionList")};
Method specialize
  = genExtClass.getMethod
      ("specialize", paraTypes);
// メソッド specialize に渡す引数
Object[] args
  = new Object[]{new Integer(3),
                 classGen.constantPool(),
                 classGen.instructionList()};
// generating extension の実行
String sig
  = (String)specialize.invoke(genExt, args);
// 特化されたクラスの作成, ロード
Class specializedClass = classGen.getClass(sig);
Object specializedPower
  = specializedClass.newInstance();
// 特化されたメソッドの引数の型
paraTypes = new Class[]{Integer.TYPE};
Method residual
  = specializedClass.getMethod("residual",
                               paraTypes);
// 特化されたメソッドの引数
args = new Object[]{new Integer(5)};
// 特化されたメソッドの起動
residual.invoke(specializedPower, args)；
```

参　考　文　献

1) Ole Agesen and David Detlefs. Mixed-mode Bytecode Execution. Technical report, Sun Microsystems, 2000.
2) Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapenõ jvm. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.

3) C. Consel and F. Noel. A general approach for run-time specialization and its application to c. In *Proc. 23 rd Annual ACM Symposium on Principles of Programming Languages*, pp. 145–156, January 1996.
4) Chales Consel, Luke Hornof, Francois Noël, Jacques Noyé, and Nicolae Volanschi. A Uniform Approach for Compile-time and Run-time Specialization. Technical report, The French Nationnal Institute for Research in Computer Science and Control, January 1996.
5) Markus Dahm. The Byte Code Engineering Library. http://bcel.sourceforge.net/.
6) D. Engler and V. retargetable. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI ' 96)*, May 1996.
7) Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterms: elements of reusable object-oriented software*. Addiso-Wesley, 1995.
8) James Gosling, Bill Joy, and Guy Steel. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.
9) Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pp. 293–304, Atlanta, Georgia, May 1–4, 1999.
10) Lude Hornof and Jacques Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *Theoritical Computer Science*, Vol. 248, No. 1-2, pp. 3–27, Oct 2000.
11) N.D. Jones, C.K. Gomard, and P.Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.
12) Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, 1993.
13) Peter Lee and Mark Leone. Optimizing ML with run-time code generation. *ACM SIGPLAN Notices*, Vol. 31, No. 5, pp. 137–148, 1996.
14) Timothy Lindoholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition*. The Java Series. Addison-Wesley, 1999.
15) Hidehiko Masuhara and Akinori Yonezawa. Run-time bytecode specialization: A portable approach to generating optimized specialized code. In Olivier Danvy and Andrzej Filinski, editors, *Second Symposium on Programs as Data Objects (PADO II)*, LNCS, Aarhus, Denmark, May 2001. to appear.
16) S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, K. Hotta, and H. Takagi. OpenJIT: A Reflective Java JIT Compiler. In *Proceedings of OOPSLA '98, Workshop on Reflective Programming in C++ and Java*, 1998.
17) Sun Microsystems. Java HotSpot Technology. White paper available at http://java.sun.com/products/hotspot/whitepaper.html.
18) Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *Proceedings of the third Conference on Object-Oriented Technologies and Systems (COOTS)*, pp. 1–20, 1997.
19) H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and F. Kimura. OpenJIT : An Open-Ended, Reflective JIT Compiler Framework for Java. In *Proceedings of European Conference on Object-Oritented Programming (ECOOP'00)*, pp. 262–387, June 2000.
20) Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 2, pp. 324–369, March 1999.
21) U. P. Schultz, J. Lawall, and C. Consel. Specialization patterns. Research Report 3853, INRIA, January 2000.
22) U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, pp. 367–390, June 1999.
23) Eugen N. Volanschi, Charles Consel, and Crispin Cowan. Declarative specialization of object-oriented programs. *ACM SIGPLAN Notices*, Vol. 32, No. 10, pp. 286–300, 1997.
24) 藤波順久. オブジェクト指向言語を利用した自動的かつ効率的な実行時コード生成. コンピュータソフトウェア, Vol. 15(5), pp. 25–37, 1998.