Java 言語向け適応的部分計算の設計と実装

丸山直也[†] 松岡聡[†,††] 小川宏高[†]

部分計算によって固定入力で頻繁に実行されるプログラムを高速化できることが知られている。しかし従来の手法はコード爆発、メモリ使用量の増加などのため現実的な応用は少ない。我々は、プログラム特化の 1 手法であるデータ特化技術と実行時プロファイルによって、上記問題を解決する技術を提案する。データ特化技術をループにのみ適用しコード爆発を回避しつつ、特化されたコードを生成する。また、実行時プロファイルによって頻繁に実行されるパスを判別し、特化をその部分にのみ適用することによって使用メモリ量を抑える。同手法を Java 言語上で予備評価を行い、我々の手法が実際に上記問題を解決し、かつ高速化を実現できることを確認した。

1. はじめに

1.1 背景

近年 Java 言語 [6] が、そのオブジェクト指向的機構、安全性等によりプログラマの注目を集めている。Java 言語は仮想機械と機種独立なプログラム表現形態を用いることにより、実機械非依存性を実現している。しかし同時に、Java プログラムはインタプリタによって解釈実行されるか、機械語に実行時にコンパイルして実行されるため、大きな実行時オーバーヘッドを共なう。そのため、効率的な JIT コンパイルによって Java プログラムの性能を向上させる、多くの研究が為されてきた [13,14]。しかし依然として、Java の性能は C や Fortran 等のそれより低く、特に数値計算などの高い性能が要求される分野には Java は適していないと見なされている。

部分計算 (partial evaluation) とは、一般的なプログラムをその入力の 1 部をある値に固定したプログラムを自動生成する技術である [8]。固定された入力 (以下、静的な入力と呼ぶ) に依存する計算は全て特化によって評価され、特化されたプログラムには残らない。従って、あるプログラムが固定入力で頻繁に実行される場合、その入力を静的な入力として部分計算を適用することによって高速化を達成できる。実際に、Glück らは入力配列を走査するループが主である数値計算プログラムについて、その長さが静的ならば特化によって高速化を達成できることを示している [5]。また、特に、実行時に得られる情報を用いた部分計算は実行時特化 (runtime specialization) と呼ばれ [4, 7]、実行時に分かる値について特化可能なため、コンパイル時部分計算より効果の高い特化を行える。

増原らによるバイトコード特化 (bytecode specialization, BCS) [12] は、上記実行時特化技術を Java プログラムに適用した研究である。BCS は Java バイトコード [11] で表されるメソッドに対して、実行時に特化されたメソッドをバイトコードとして生成する。従って、従来の実行時特化とは違い、実機械、Java virtual machine (JVM) 独立なシステムである。さらに, 通常 JVM ではバイトコードで記述されたプログラムを JIT を用いて動的に実機械語にコンパイルされるが、この際最適化も適用される。特化システムによって生成されたプログラムも JIT によって実機械語にコンパイルされてから実行される. 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 に委ねることができ、効率の良い実行が期待できる。実際に BCS によって、インタプリタなどのプログラムを高速化できることが確かめられている [12]。

1.2 問題

上述した背景にも関わらず、Java 言語において部分計算は広く用いられてはいない。これは以下にあげた理由の為である。

コード爆発 部分計算によって静的な条件からなるループをアンロールすることができる。特に FFT のような数値計算アプリケーションにおいてアンローリングによって高速化を達成できる機会が多く存在する [5,10]。しかしながら、ループ展開はコード爆発を起こす可能性があり、その結果大幅な性能低下を招きかねない。特に Java では 1 つのメソッドのサイズが 64 キロバイト以内と定められており [11]、安易にループ展開を適用することは避ける必要がある。また、コードサイズが増大することによって、JIT に要する時間が増大することも考慮すべきである。さらに、コードサイズがある程度以上になると、JIT コンパイルせずインタプリタ実行になり、大幅な性能低下を引き起こす場合がある☆。

特化と JIT コンパイラの最適化の衝突 Sun の HotSpot JIT compiler [13] や IBM の jitc [14] は、JIT

† 東京工業大学
†† 国立情報学研究所

☆ Linux 上の Sun JDK1.4.0 を使った実験では、10KB のメソッドを 1 万回実行しても JIT コンパイルされなかった。

コンパイルのオーバーヘッドを削減するために、積極的な最適化をインナーループなどの限られた範囲に限定している。従って、特化によるループ展開を JIT コンパイルの前に適用すると、JIT による積極的な最適化が適用されない。

上述の通り、特化によるループ展開は為されるべきではないが、ループ展開せずに繰り返し条件が静的であるループの内部のコードを特化することはできない。これは、各繰り返しごとに生成される式が異なるためである。

Knoblock らによるデータ特化 [9] は上記問題に取り組んだ研究である。データ特化とは、実行時特化の一種であり、特化対象プログラムからローダとリーダを生成する。ローダは、プログラム中の静的な命令からなり、動的な命令で用いられる式 (リフトされる式 [8]) の値をメモリにキャッシュとして保存する。リーダは残りの動的な命令からなり、ローダによるキャッシュを用いて計算を行う。すなわち、実行時にローダが一度実行され、リーダが繰り返し実行される場合、データ特化によって高速化が期待できる。データ特化がプログラム特化と違う点は、データ特化では静的な命令の実行結果をコード中に埋め込まないことである。このため、ループアンローリングをせずにループ中の静的な命令を特化可能である。さらに、ローダ、リーダ共に実行時に与えられる定数に依存しないため、コンパイル時に生成できる。従って、実行時コード生成の必要がなく、実行時特化と比較して軽量な特化を実現できる。

一方、キャッシングにはスペースオーバーヘッドが共なうため、使用可能なメモリを考慮せずにキャッシュすることは性能低下につながる。また、同じデータサイズの式をキャッシュするならば、頻繁に実行されるパスの式をキャッシュした方が効果的である。従って、キャッシュする式の選択は、使用可能なメモリ量、キャッシングの効果を考慮したアルゴリズムが必要である。しかしながら、Knoblock らによる従来のデータ特化では単純な静的なヒューリステックスによって式を選択しており、実行時の振舞いを必ずしも反映しない。従って、より実行時の振舞いを考慮した手法が必要である。

1.3 貢　　献

本論文の最も主要な貢献は、実行時プロファイル情報を用いた Java プログラム向け適応的部分計算 (feedback-directed partial evaluation, 以後 FDPE と呼ぶ) の提案である。FDPE は実行時にメソッドを特化する技術である。FDPE の主な特徴は以下の通りである。

- コンパイル時に Java クラスファイル中のメソッドを解析し、各メソッドに専用な特化器を生成する。実行時に特化器を実行し、特化されたコードをバイトコードとして生成、実行する。
- 従来の実行時特化とデータ特化を組み合わせ特化技術である。ループ外のコードには実行時特化を適用し、ループ内のコードにはデータ特化を適用する。
- 1 つのメソッドに対して 2 段階の特化を行う。最初の段階 (ベースレベル) では、ループ外のコードにのみ実行時特化を適用し、ループ内のコードには特化を適用せず、生成されるコードに残す。次の段階の特化では、ベースレベル特化に加えて、ループ中の静的な式をキャッシュする。FDPE では、実行時プロファイル情報を用いて、頻繁に実行されるメソッドについてのみ、第 2 段階の特化を適用する。これにより、使用可能なメモリを効率的に用いることが可能になる。
- キャッシュ選択は上記プロファイルとヒューリスティックスにより自動的に行われる。
- 完全に実機械、JVM 独立である。

本論文では、上記 FDPE の効率的な実装方法を示す。また、FDPE の予備評価を行い、従来の手法と同程度の高速化かつスペースオーバーヘッドを削減可能なことを示す。

構　　成

FDPE の設計について 2 節で、実装について 3 節で述べる。次に、実験、評価について 4 節で述べ、関連研究を 5 節で紹介する。最後に 6 節でまとめと今後に向けた課題を述べる。

2. Java 言語向け FDPE システムの設計

本節では FDPE システムの設計について述べる。まず、システムが満たすべき要求について述べる。次に、プログラム特化とデータ特化を比較しそれぞれの長所短所について考察する。次に、その考察と要求に対する技術である実行時プロファイル情報を用いた FDPE について述べる。最後にシステムの全体像を示す。

2.1 要　　求

上述した問題に対処し、かつ効果的な特化を実現するため、本システムに対して以下の要求が挙げられる。

- ループ展開を行わない。
- CPU、メモリに対する実行時オーバーヘッドを極力減らす。

ループ展開を行わないのは、コード爆発、JIT コンパイラとの作用を考慮したためである。

また、我々は本システムを JVM 独立なものとすることを目標とする。実行時コード生成やプロファイリング等は、バイトコードレベルで実現するより JVM 内部で実現した方が効率が良いことが予想される。しかし、その場合、本特化システムを用いるために、我々が変更を施した特殊な JVM が必要となる。 我々は、効率より JVM 独立であることを重視する。

2.2 プログラム特化とデータ特化の比較

静的な命令の実行結果の内、動的な命令で用いられる値は特化されたコードから利用可能にしなければならない。この操作は、部分計算ではリフトと呼ばれている [8]。リフトは、通常のプログラム特化では値を特化されたコードに直接埋め込むことで実現されている。一方、データ特化では、メモリにキャッシュとして保存し特化されたコードがそのメモリを参照することで実現されている。直接埋め込むプログラム特化の手法では、ループ中で定義される静的な値はアンローリングなしにはリフトできない。データ特化のキャッシュを用いる手法では、値をコード外に置くためアンローリングは必要ない。

図 1 にプログラム特化とデータ特化をループに適用した場合の比較図を示す。左側のコード断片を S を静的として特化するとする。ここで下線が引かれている式、命令は、S にのみ依存するものである。例えば、関数 F は静的に評価可能である。特化の例として、図では S = 4 の場合について特化したコードを右側に示す。また、F の 返り値は左側下部に示した値だとする。右側上部のコード断片がプログラム特化によって生成されるコードを表し右側下部のコードとキャッシュがデータ特化によって生成されるコードとデータである。この元のコードでは、変数 t がループ中で静的に定義され、かつ動的な命令で使用されている。従って上述したように、t の値は生成されたコードにリフトされる。プログラム特化によるコードは、ループがアンロールされ、コードに t の値が埋め込まれていることを示している。一方、データ特化ではループはそのままコードに残り、変数 t がそのキャッシュである t_cache への参照に置き換えられている。

図中のプログラム特化によって生成されたコードは、明らかにデータ特化のそれより効率が良い。 これはループの繰り返し回数が少なく、コード爆発などの問題を起す恐れがないからである。しかし、1 節で述べたように、一般的にループ展開をバイトコードレベルで適用することは避けるべきである。従って、プログラム中のループはプログラム特化ではなく、データ特化を適用すべきである。

また、データ特化では、静的な分岐命令を除去できない。図 2 は、分岐命令にプログラム特化とデータ特化を比較した図である。図 1 と同様、左側のコード断片を変数 S を静的として特化する。ここで、このコード断片はループ外にあるものとする。 右側のコードが、S、F を左下の値だとした時、プログラム特化、データ特化それぞれで生成されるコード、データである。この際分岐命令は、その条件部分が静的なので、プログラム特化では除去される。しかしデータ特化では、コードを静的に生成するため実行時の値による分岐命令の除去は不可能である。結果的に、特化されたコードは F の計算以外元のコードに等しい。ゆえに、

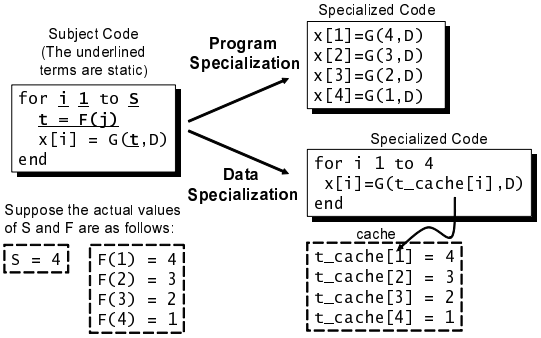

図 1 プログラム特化とデータ特化の比較図: 左側のコードが特化対象のコードを表し、右側のコードがプログラム特化、データ特化それぞれで生成されたコードを表す。ここで、元のコード中の変数 S が静的だとすると、下線が引かれた式が静的となる。さらに、 S の値が 4、F(i) の値がそれぞれ左側下部にある通りだとする。

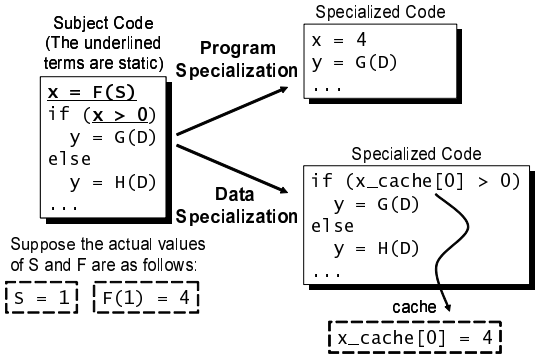

図 2 プログラム特化とデータ特化を分岐に適用した比較図: 左側のコードが特化対象のコードを表し、右側のコードがプログラム特化、データ特化それぞれで生成されたコードを表す。ここで、元のコード中の変数 S が静的だとすると、下線が引かれた式が静的となる。さらに、 S の値が 1、F(1) の値が 4 だとする。また、x のキャッシュはヒープ上にあることを示すため、配列形式で記述した。

データ特化で生成されるコードはプログラム特化のそれより非効率である。

2.3 実行時プロファイル情報を用いた適応的部分計算

上記比較を踏まえて、我々はプログラム特化とデータ特化を組み合わせた特化を行う。まず、ループ展開を避けるため、ループ内のコードにはデータ特化を適用する。しかし、ループ外のコードにはプログラム特化を適用してもコード爆発を起こすことはない☆。従っ

☆ ある関数を静的な引数についてプログラム特化すると、その静的な引数の値に特化されたコードを生成する。従って、実際には、異なる値で特化を何回も行うと毎回新しいコードを生成しコード爆発を起こす危険がある。しかし、そもそも静的な引数は実行時に頻繁に値を変えない引数のことであり、現実的にはコード爆発の可能性はないと言える。一方、データ特化では異なる静的な引数の値について、毎回同じコードを用い、別々のキャッシュ

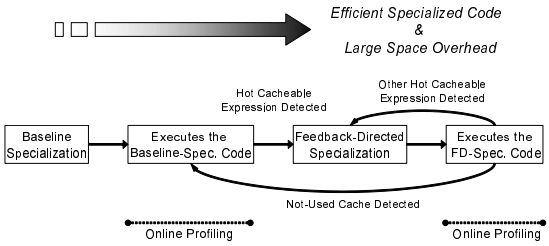

図 3 FDPE における実行時の流れ

て、ループ外のコードにはプログラム特化を適用する。これにより、図 2 のような分岐は最終コードから除去される。

この手法によりコード爆発を阻止できるが、キャッシュのスペースオーバーヘッドを考慮する必要がある。まず、キャッシングは、必要なメモリ量とその時点での利用可能なメモリ量を考慮する必要がある。例えば、JVM 上のプログラム全体がメモリを大量に使用し GC が頻繁に起きているような状況では、キャッシングを行うべきではない。また、ある式をキャッシュしたとしても、そのキャッシュが頻繁に使用されなければ効果は小さく、頻繁に使用される式を優先的にキャッシュした方が効率的にメモリを使うことができる。

我々は、利用可能なメモリを効率的に用いるため、実行時プロファイルを用い、特化を頻繁に実行される部分に段階的に適用する。本手法は、以下の通りである (図 3 参照)。まずベースライン特化として、ループ外のコードのみに通常の部分計算を適用する。これはループ中の静的な式も全て動的としたものであり、キャッシングは一切行わない。この際、この動的とした式を次段階のデータ特化の対象とする。次に、ベースライン特化によって生成されたコードを実行しつつ、キャッシング対象となる式の値を用いる命令の実行回数を計測する。このプロファイル結果を元に頻繁に使用される式を優先的に選択し、データ特化を適用する (キャッシュする式を選択するアルゴリズムの詳細は後述する)。この段階で生成されるコードは、ベースライン特化とデータ特化が適用されたものになる (これを FD 特化と呼ぶ)。また、キャッシュした式はこの時点で頻繁に使用されると判断されたものだが、その後の実行を通して頻繁に使用されるとは限らない。従って、データ特化が適用されたコードもプロファイルをとり、キャッシュの使用頻度を調べる。使用頻度が低く、GC が頻繁に起きている状況や、他の使用頻度の高いキャッシュ対象が存在する場合、キャッシュ用メモリを解法し、ベースラインのコードに戻す。またそうではない場合にも、さらに同一のメソッド中で他のキャッシュ対象が現れた場合は、さらなるキャッシングを行う。

を作る。従って、コード自体が占めるスペースは増大しない。

2.3.1 キャッシングアルゴリズム

本システムは、頻繁に使用される値を優先的にキャッシュするため、キャッシュする値の選択に実行時プロファイル情報を用いる。

2.2 節で述べたように、キャッシングはリフトされる値に対してのみ行えばよい。以下では、これを (キャッシング) 対象値と呼ぶ。対象値をキャッシュした場合、その値を計算するコードはキャッシュへの参照 (ヒープへの参照) に置き換えられる。従って、キャッシングは以下の利益、オーバーヘッドを伴う。

利益 対象値の再計算の省略

オーバーヘッド ヒープからの値の読み込みと、そのスペースオーバーヘッド

すなわち、キャッシングは対象値を再計算するよりキャッシュを参照した方が効率が良い場合のみ、高速化を達成できる。また、プログラム中に複数のキャッシング候補である式が存在するとする。利用可能なメモリが十分ある場合は、それら複数の候補をキャッシュ可能である。しかし、 そうではない場合は、候補から選択的にスペースに収まるようなキャッシングを行う必要がある。この場合、候補それぞれの利益とオーバーヘッドを比較し判断されるべきである。

しかしながら、上記利益とオーバーヘッドを比較するのは難しい。なぜならば、あるバイトコード列の計算にかかる時間は実機械、JVM 依存であり、静的に判別できる値ではない。また、実行時プロファイルを用いても、数バイトコード命令程度の実行時間を実行時に調べることは精度上大変困難である☆。

我々は、静的なヒューリスティックスと動的なプロファイル情報を用いたアルゴリズムを提案する。まずコンパイル時に、Knoblock らによるもとの同様なヒューリスティックスにより、対象値の計算コストを調べる [9]。このコストがメモリ参照より小さいと予測される対象値は、キャッシング対象から除外する (例えば、加算命令など)。実行時には、初めは一切キャッシュなしに計算を開始し、キャッシング対象の値について、その値がどの程度頻繁に計算される値なのかを調べる。これはその値が定義されるベーシックブロックの実行頻度を調べれば判別できる。また同時にキャッシュに必要なメモリスペースを調べる。最後に、これらの情報を元に、最も頻繁に計算されている値を優先的にキャッシュする。 その際、そのキャッシュに必要なメモリスペースとその時点での利用可能なメモリスペースを考慮する。これはキャッシュにどの程度メモリを与えるかを来めるポリシーに依るものであり、例

☆ バイトコード上で、JVM、実機械独立な方法で、バイトコード列の実行時間を調べるには、そのコード列前後に時間を計るコードを挿入すれば可能かもしれない。しかし、JIT コンパイラによって生成される機械語では、バイトコード列での命令順序が Java の仕様に沿って自由にスケジューリングされる。従って、得られた実行時間は意図した命令の実行時間になる保証はない。

えば、常にキャッシュに 50% 以上のメモリを使用させないといったポリシーをプログラマが指定可能にする。

本手法により、自動的に頻繁に使用される値のみをキャッシュすることが可能になり、従来のデータ特化より利用可能なメモリを効率良く用いることができる。

2.4 システム概要

本システムは、コンパイル時の解析、コード生成によって、実行時に特化を実現する。以下では、コンパイル時、実行時それぞれの処理の概要を述べる。

2.4.1 コンパイル時

まず、ユーザが特化するメソッドと、その引数の束縛時 (静的または動的) を指定し、そのメソッド中の式、命令を静的な引数にのみ依存するものとそうでないものに分離する (束縛時解析、BTA [8])。また、上述のキャッシングアルゴリズムを適用し、ループ中のキャッシング候補を探す。次に、動的な命令からなる特化されたメソッドのテンプレートとなるクラスをバイト配列として生成し、ファイルに保存する。さらに、静的な命令から構成され、このテンプレートを用いて特化されたメソッドを生成するメソッド `Specializer` を生成する。

2.4.2 実 行 時

図 4 に本システムの実行時の全体像を示す。ここで、`Specializer` は静的に、`Specialized Code` は動的に自動生成されたプログラムである。処理の流れは以下の通りである。

(1) ユーザコードから `Specialization Controller` にリクエストを出す。この際、静的に解析済みである、特化するメソッド、その引数の束縛時、静的とされた引数の実際の値を指定する。
(2) リクエストを受けた `Specialization Controller` は、指定されたメソッドに対応する `Specializer` を、静的な引数の値を指定し呼び出す。
(3) 呼び出された `Specializer` は、与えられた静的な引数を用いて、静的な命令を実行する。
(4) 生成する特化されたクラスのテンプレートとなるバイト配列をファイルシステムからロードし、実行の結果を反映させる (詳細は次節で述べる)。
(5) `Specializer` はこのバイト配列を `Custom Class Loader` に渡し、JVM にロードする。
(6) ユーザコードが、このロードされたクラスの特化されたメソッドを直接呼び出し、実行する。同時にプロファイル情報を `Specialization Controller` に渡す。

`Specialization Controller` は、システム中で特化を一元的に管理するコンポーネントである。`Specialization Controller` は、特化されたメソッドのプロファイル情報を定期的に読み、頻繁に使用されるキャッシュ対象の値を探す。実際にどの値をキャッシュするかは、上述のキャッシングアルゴリズムに元づいて判断する。

3. 実 装

本節では、2 節で述べたシステムのプロトタイプ実装について説明する。

3.1 束縛時解析

コンパイル時に行う束縛時解析は、Java のクラスファイルを解析し、バイトコードについて各命令の依存関係を調べるものである。Java バイトコードはスタック操作 (`iadd` など) とレジスタ操作 (`iload` など) 両方を含むものである。そのため、直接バイトコード列を束縛時解析するのは容易ではない。我々は、バイトコードをレジスタ操作命令のみからなる中間表現に変換し、それに対して束縛時解析を実現する。クラスファイルの解析、変換には Jikes RVM ☆ の最適化コンパイラ部分を用いる。Jikes RVM は、x86 または PowerPC 上で動作する研究向け Java 仮想機械であり、ブートストラップなどの一部を除いてほぼ Java で記述されている。ここで、我々が用いる部分は全て Java で記述されているため我々のシステムの可搬性を損ねることはない。束縛時解析はこの Jikes RVM の中間表現に対して実現する。また、Jikes RVM は各種最適化器が用意されているため、最適化された中間表現を束縛時解析の入力とすることができる。

3.2 実行時バイトコード生成

CPU に対するオーバーヘッドを減らすために、実行時コード生成のコストを削減する方法を考察する。

増原らによる BCS [12] では、generating extension が生成される各命令を実行時に生成する。この方法では、実装が容易な反面、効率が良くない。

Consel らによる Tempo [4] では、各ベーシックブロックに対応する機械語のテンプレートを用意し、実行時にそのテンプレートを静的な命令の実行結果を元に並び換える。テンプレートは、静的な命令の実行結果を埋める「ホール」を持ち、実行時に適宜ホールを実際の値で置き換える。この方法では、実行時処理はテンプレートを並び換えるだけであり、比較的効率が良いが、テンプレートのコピーのコストや、ここで、テンプレートをメソッド全体ではなく、ベーシックブロック単位で用意する理由は、メソッド中の静的な分岐命令を除去するためである。クラスファイルとして正しいフォーマットにするためのコストがかかる。

データ特化 [9] では、コードは全てコンパイル時に生成される。実行時には、静的な命令の実行結果をキャッシュとしてヒープ上に保持し、静的な命令をヒープへの参照に置き換えたコードを実行する。この方式では、分岐命令はたとえ静的であっても除去されず、最終コードに定数による分岐命令として残る。このためこの方式では、実行時コード生成のオーバーヘッド

☆ http://www-124.ibm.com/developerworks/oss/jikesrvm/index.shtml

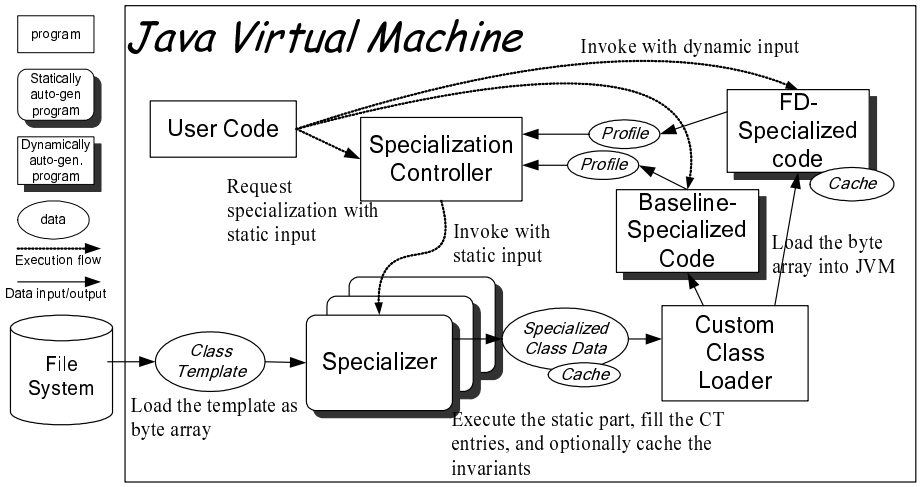

図 4 FDPE システムの実行時の全体像

がかからないが、生成されるコードは通常の部分計算によるものより効率が良くない。

これらの従来の方法を考慮し、我々は JIT コンパイラの存在を仮定した、プログラム特化とデータ特化の中間的な方針を取る。本方針では、JIT コンパイラによってバイトコード上の定数による分岐命令が除去されることを仮定する。まず、データ特化と同様にメソッド全体を単位としてテンプレートコードをコンパイル時に生成する。実行時には、データ特化とは違い、テンプレートコードをそのまま使うのではなく、コード中に静的な命令の結果をコンスタントプールエントリとして埋め込む。定数であることを JVM に示すことによって、その定数による分岐命令が JIT コンパイラによって除去されることを仮定できる。

すなわち、本システムにおける実行時特化は以下の通りである: 1) 静的な命令を実行する、2) 実行時に静的に生成されたクラスファイルをバイト配列としてファイルシステムから読み込む、3) バイト配列中の適切なコンスタントプールエントリを静的な命令の実行結果で書き換える、4) クラスとして JVM にロードする。この方式では、実行時に必要な処理がコンスタントプールの値を書き換えるだけであり、Consel らによる手法とは違い、テンプレートのコピー、並び換えなどの操作が必要ないことである。かつ、従来のデータ特化とは違い、ループ外の静的な値による命令は JIT コンパイラによって最適化されることが期待できる。

しかし、FDPE では、プログラム中のどのコードをキャッシュからの読み込みに置き換えるのかは静的に知りえない。しかし、可能な組み合わせの全てに対して、あらかじめコードを用意しておくことは非効率的である。我々は、通常のコードかキャッシュからのコードを実行するのかを分岐命令で選択するようなコードをテンプレートとして用意する。ここで、分岐命令はコンスタントプールの値で制御され、クラスを JVM にロードする前に適切に書き換える。これにより、クラスファイルテンプレートを 1 つのみ用意するだけで、全ての条件に対応でき、かつ分岐命令は JIT コンパイラにより削除されれば、性能の低下も無い。

3.3 実行時プロファイリング

本システムでは、実行時にループ中のデータ特化の対象となる変数の使用頻度のプロファイルをとる。使用頻度は、メソッド中の各ベーシックブロックの実行回数と、それぞれのベーシックブロック中での使用回数がわかればよい。ここで、ベーシックブロック中での使用回数は静的にわかる情報であり、実行時にはベーシックブロックの実行回数を計測すれば十分である。

JVM 非依存な方法でプロファイルをとるには、バイトコードにプロファイルをとるコードを挿入するか (インストラメンテーション)、JVM に標準で備わる JVMPI を用いる方法が考えられる。しかし、JVMPI ではメソッドの実行開始、終了や、GC の開始、終了程度のイベントしか捉えられず、ベーシックブロックレベルの情報を得ることは不可能である。従って、我々はバイトコードインストラメンテーションによって、プロファイル情報を収集する。

また、プロファイルによるオーバーヘッドを削減するため、サンプリングしたプロファイリングを行う。Arnold らによる、プロファイリングをサンプリングする手法 [3] は、メソッド全体をコピーし、インストラメンテーションをコピーにのみ適用する。さらに、メソッドの先頭にオリジナルのコードかインストラメントされたコードのどちらを実行するかを選択する分岐命令を追加する。Arnold らによって、実際にベーシックブロックの実行頻度プロファイリングを軽量かつ十分に高精度に行えることが確かめられており [2]、我々のシステムでも同様の手法を用いる。

具体的には、ベースライン特化、FD 特化によって生成されるコードの複製を用意し、その複製にのみベー

シックブロックの実行回数を計測するコードを挿入する。さらに、プロファイル用メソッドと元のメソッドのどちらを実行するか選択するメソッド (エントリーメソッドと呼ぶ) を用意する。ユーザコードは、このエントリーメソッドを介して特化されたメソッドを実行する。エントリーメソッドはある回数に 1 度プロファイル用メソッドを呼び出すが、それ以外は単純に特化されたメソッドを呼び出すだけである。

3.4 実行メソッドの切り換え

上述の通り、本システムでは元のメソッドに対して以下のメソッドが生成される。

(1) エントリーメソッド
(2) ベースライン特化メソッドとそのプロファイル用メソッド
(3) FD 特化メソッドとそのプロファイル用メソッド

このうち、エントリーメソッドのみがユーザから実際に呼び出されるメソッドであり、その他はエントリーメソッドから選択的に呼び出される。また、エントリーメソッド以外は実行時に特化器によって生成され、かつ FD 特化メソッドは任意の時点で生成されうる。従って、エントリーメソッドと各特化メソッドはそれぞれ別のクラスのメソッドとして生成する。まず 1 つ目のクラスとして、エントリーメソッドを含むクラスを生成する (以下、これをエントリークラスと呼ぶ)。エントリーメソッド自身は実行時情報に元づいて生成されるわけではないので、エントリークラスは静的に生成可能である。次にそれぞれの特化メソッドを含むクラスを生成する。このクラスは特化メソッドとそのプロファイル用メソッドを含み、実行時に動的にテンプレートより生成される (3.2 節参照)。一方、FD 特化メソッドは任意の時点で生成されるため、エントリーメソッドの呼び出し先を動的に変更する必要がある。我々は、これをいわゆるコマンドパターンとして実現する。すなわち、 各特化メソッドと同一のシグネチャの抽象メソッドを持つ抽象クラスを用意し、 実際の特化メソッドはこれをオーバーライドさせる。ここで、メソッドのシグネチャは実行時情報に依らないため、抽象クラスは静的に生成可能である。エントリークラスは抽象クラス型の変数として、サブクラスのインスタンスを持つ。エントリーメソッドはこのインスタンスについて特化メソッドの呼び出しを行う。またプロファイル用メソッドも同様である。エントリークラスはエントリーメソッド以外に、このインスタンスを切り換えるメソッドを持ち、Specialization Controller からプロファイル情報に元づいて適宜呼び出される。

4. 予備評価

提案するシステムの実装はまだ完了していないため、本論文では予備評価として人手でシミュレートした FDPE によるピーク性能向上を従来の手法による特化と比較する。

従来の特化手法とは異なり FDPE では以下のオーバーヘッドがかかる：1) プロファイルにかかる時間、2) エントリーメソッドを介した間接的な呼び出しによるオーバーヘッド。前者は定期的に発生するオーバーヘッドであり、後者はメソッド呼び出し毎にかかるオーバーヘッドである。従って、FDPE によるピーク性能向上は従来の手法によるものより劣る恐れがある。以下では、FDPE とプログラム特化、データ特化を FFT に適用し、ピーク性能向上を比較する。

4.1 評価方法

特化対象のプログラムとして、FFT を用い☆、サンプル数と変換の方向を固定した特化を行った。変換を行うプログラムは 1 つのクラスメソッドとして実現されており、入力として、変換される倍精度浮動小数の配列を 2 つ (複素数のため)、配列の長さ、変換の方向を示す整数値、の 4 つを持つ。このメソッドの主な計算は 3 重ループからなり、計算コストは $O(n \log n)$ である。

また、この場合、束縛時解析で静的とされる主な計算は、1 回の `Math.sqrt` の呼び出し (ループ外)、n 回の `Math.sin`、`Math.cos` の呼び出し (ループ内)、である (n をサンプル数とする)。さらに、メソッドに含まれる 3 重ループは完全に静的であり、ループ展開可能である☆☆。

本実験における特化は以下のとおり実現した。まず、プログラム特化は、実装中のシステムで実現可能であり、自動生成したメソッドを用いた。このメソッドでは、3 重ループが全て展開されており、上述の数学関数の結果が埋め込まれている。データ特化は、数学関数の結果をキャッシュするコードを人手で記述し、実現した。FDPE も同様に、キャッシュするコードや、エントリーメソッド、プロファイル用メソッドを人手で記述し実現した。ただし、ループ外の `Math.sqrt` の結果は、本来の FDPE ではコード中に埋め込まれる。今回は、その値を定数として直接コードに記述することでシミュレートした。これらのメソッドのピーク性能を計測するため、それぞれ 1 万回実行した後に、100 回の実行時間を計測した。また、プロファイル用メソッドは 100 回に 1 度、すなわちだけ計測中に 1 回実行されるようにした。

実験環境は、CPU Athlon 1.2 GHz、RAM 1 GByte 上の Linux kernel 2.4.18 であり、JVM として、Sun JDK 1.4.1 Server VM を用いた。

4.2 結　　果

図 4.2 に結果を示す。まず、プログラム特化のデー

☆ ソースコードは http://sepwww.stanford.edu/oldsep/hale/FftLab.java からダウンロード可能

☆☆ 前述したように、プログラム特化ではループ展開しないとループ中の命令を特化できない。すなわち n 回の `Math.sin`, `Math.cos` を簡約するためには、ループ展開が必要である。

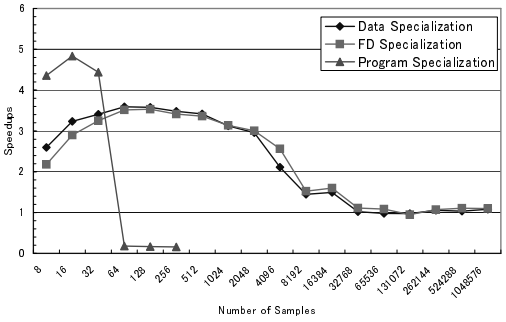

図 5 ピーク性能向上の比較

タがサンプル数 256 までしかない理由は、それ以降では特化されたメソッドのサイズがループ展開のため 64KB 以上になってしまうためである。また、サンプル数が 32 まではその他の特化を凌ぐ高速化を達成できているが、逆に 64 以降では急激な性能低下が発生している。これらは、性能向上はループ展開によるものであり、また性能低下もループ展開によるコードサイズの増大が原因である (JIT コンパイルされない)。一方、他のデータ特化と FDPE ではほぼ等しい性能向上を達成している。これは、プロファイルのオーバーヘッドはほぼ無視できることを意味し、また JIT コンパイラによってメソッド呼び出しのオーバーヘッドも除去されていることを意味する。

4.3 考　　察

前述した通り、FDPE の目標はより少ないメモリオーバーヘッドで従来の手法と同程度またはそれ以上の高速化を達成することである。上記結果より、FDPE におけるオーバーヘッドはピーク性能としては無視できるものであると言える。従って、長時間実行されるプログラムでは、我々の手法でも従来のデータ特化と同程度の高速化が可能である。しかし、FDPE は従来のデータ特化とは異なり、キャッシュする式を自動的に選択可能である。よって、同程度の高速化を達成しつつメモリオーバーヘッドを削減することが期待できる。

5. 関 連 研 究

Consel らによる Tempo [4] は C 言語向け特化システムであり、コンパイル時、実行時の特化が可能である。実行時特化では、通常の C の関数に対して、実行時に特化された関数を機械語で自動生成する。この際、コード生成のオーバーヘッドを削減するために、コンパイル時に機械語のコード断片をテンプレートとして用意する。また、テンプレートは実行時定数から静的な命令を実行することによって得られる値を埋め込むホールを持つ。実行時には、実行時定数で静的な命令を実行し、ホールを埋め、テンプレートを適切に並び換えることによって最終的な特化されたコードを生成する。従って、実行時オーバーヘッドは比較的小さいと言える。しかし、特化されたコードはテンプレートの並び換えに過ぎないため、最適化されてはいない。実際に、従来のコンパイル時特化と同等の性能向上が達成できないことが報告されている [12]。

増原らによるバイトコード特化 (BCS) [12] は、Java バイトコード向け実行時特化技術である。静的にクラスファイル中のメソッドを束縛時解析し、generating extension をバイトコードとして生成する。実行時にこの generating extension を実行時定数により実行し、特化されたコードをバイトコードとして生成する。BCS ではバイトコードのみ解析、生成するため、ソースコードを必要とせず、JVM 可搬である。また、特化されたバイトコードは、JVM が通常備える JIT コンパイラにより機械語にコンパイルされてから実行される。従って、最終的な実行コードは特化かつ最適化された高品質なものとなることが期待できる。しかし、1 節 で述べたように、BCS ではコード爆発や特化と JIT コンパイラによる最適化と衝突などを考慮していない。

Knoblock らによるデータ特化 [9] は、Tempo による実行時特化と同様に、C プログラムを実行時定数について部分計算する。データ特化では特化対象関数から、ローダとリーダと呼ばれる 2 つの関数を生成する。ローダは元の関数の静的な引数のみに依存する命令からなり、リーダは残りの命令からなる。ローダは実行時に実行時定数を与えられて実行され、その結果をメモリ上にキャッシュとして保存する。次に、リーダがローダによるキャッシュを用いて実行され、元の関数と等価な結果を得る。ここでローダは一度だけ実行すれば良く、リーダが繰り返し実行されればキャッシングによる再計算の省略のため高速化が可能である。すなわち、データ特化で生成されるものは、特化されたコードではなく、特化されたデータである。従って、ループ中の静的な命令はアンロールすることなく特化可能であり、コード爆発の危険はない。また、ローダ、リーダ共に静的に生成でき、実行時オーバーヘッドが通常の実行時特化に比較して小さい。しかしコード爆発が起きない限り、リーダの性能は通常の実行時特化の性能より一般的に低い。これは、通常の実行時特化では generating extension の結果が即値として埋め込まれるのに対して、データ特化のリーダではメモリ参照になるからである。また、過度にキャッシュしてしまっては、そのスペースオーバーヘッドが著しく大きくなる可能性があり、逆に性能低下を招きかねない。従って、キャッシングは使用可能なメモリ量を考慮して、選択的に行う必要であり、Knoblock らによる研究では静的なヒューリスティックスによる手法を用いている。しかし、彼等の静的なヒューリスティックスはある特定の実機械に依存したものであり、実行時の

正確なプログラムの挙動を反映しないという問題がある。我々のシステムは、この問題を実行時プロファイル情報を用いることによって解決するものである。

6. おわりに

我々は Java 言語向けの新しい特化技術を提案した。本技術は、従来のプログラム特化とデータ特化を元にしており、実行時プロファイル情報により特化を選択的に適用する。従来のプログラム特化では、ループ展開によるコード爆発や JIT コンパイラとの最適化の関係が問題だったが、本手法ではデータ特化を応用することにより回避する。一方、データ特化では特化コードの性能やスペースオーバーヘッドが問題だったが、本技術ではデータ特化をループに限定することと、実行時プロファイル情報による選択的特化により、これを解決する。我々は本技術の予備評価を行い、従来の手法と同程度の高速化かつスペースオーバーヘッドを削減可能なことを示した。

今後の課題として、3 節 で述べた本技術の実装を終えることが挙げられる。その他の課題として、完成した実装を用いた実験的評価が挙げられる。しかし、現状では Java のオブジェクト指向的機構 (クラス、メソッドディスパッチなど) を十分に考慮していないため、適用可能なアプリケーションが限られてしまう。我々は、Affeldt らによるバイトコード特化のオブジェクト指向への拡張方法 [1] と同様な手法を本システムに取り入れることを検討している。

謝　　辞

本研究について有意義な助言を頂いた東京大学の増原英彦氏、東京工業大学の千葉滋氏に感謝します。

参 考 文 献

1) Reynald Affeldt, Hidehiko Masuhara, Eijiro Sumii, and Akinori Yonezawa. Supporting Objects in Run-time Bytecode Specialization. In *Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*, Aizu, Japan, September 2002.

2) Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online Feedback-Directed Optimization of Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, Seattle, Washintgton, USA, November 2002.

3) Matthew Arnold and Barbara G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation (PLDI '01)*, Snowbird, Utah, United States, June 2001.

4) Charles Consel and F Noel. A General Approach for Run-Time Specialization and Its Application to C. In *POPL96*, pp. 145–156, 1996.

5) Robert Glück, Ryo Nakashige, and Robert Zöchling. Binding-Time Analysis Applied to Mathematical Algorithms. In *System Modelling and Optimization*, pp. 137–146. Chapman & Hall, 1995.

6) James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. The Java Series. Addison-Wesley, 2000.

7) 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*, June 1999.

8) N.D. Jones, C.K. Gomard, and P.Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

9) Todd B. Knoblock and Erik Ruf. Data Specialization. In *Proceedings of the ACM SIGPLAN '96 conference on Programming language design and implementation*, pp. 215–225, Philadelphia, Pennsylvania, United States, May 1996.

10) Julia L. Lawall. Faster Fourier Transforms via Automatic Program Specialization. In *The Lecture notes of the DIKU International Summer School '98 on Partial Evaluation: Practice and Theory*, Copenhagen, Denmark, June 1998.

11) Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. The Java Series. Addison-Wesley, 1999.

12) Hidehiko Masuhara and Akinori Yonezawa. Run-Time Bytecode Specialization. In *Program as Data Objects (PADO-II)*, No. 2053 in LNCS, pp. 138–154. Springer-Verlag, 2001.

13) Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, California, USA, April 2001.

14) T. Suganuma, T. Ogasawara, M. Takeuchi, T.Yasue, M.Kawahito, K.Ishizaki, H.Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal, Java Performance Issue*, Vol. 39, No. 1, Feburuary 2000.