Java Just-In-Time コンパイラにおける最適化手法

石崎　一明

本論文では、Java Just-In-Time（JIT）コンパイラにおける最適化手法、実装、評価について示す。特に、Java言語が持つプログラムの再利用性と安全性の高さという特徴をもたらす型安全性と多態性を持つ動的束縛の実装に伴う、性能に関するオーバヘッドを削減する最適化手法を示す。この結果、Java言語が従来のC言語やC++言語と比べて同等の実用的な処理速度を得ることが出来る。本論文の成果はIBM社IBM Developers Kit. Java Technology Editionで使用され、Java言語が発表されて以来性能向上を遂げてきた要因の１つとなっている。また、仮想メソッド呼び出しのオーバヘッドを削減するための最適化手法は、他のJava JITコンパイラでも使用されている。

Java言語は1995年に発表されたオブジェクト指向プログラミング言語で、仮想機械によるプログラムのプラットフォーム独立性、オブジェクト指向と多態性を持つ動的束縛によるプログラミングの柔軟性、仮想機械による安全検査の実行と型検査による型安全性がもたらすプログラムの安全性、という特徴を持つ。これらの特徴は、Java プログラムの再利用性と実行時の安全性をもたらす。再利用性と安全性の高さによって、ユーザやプログラマの利便性は大きく向上し、Java言語は広く受け入れられ普及した。一方、これらの特徴を実現している仮想機械、動的束縛による多態性、型安全性といった方式は、その実装に伴うオーバヘッドによって性能的な問題を生じさせる。

仮想機械は基本的にインタプリタであるため、特定のプラットフォームのネイティブコードに変換して実行するコンパイル型の言語と比較して実行速度が劣る。現在では仮想機械の実装の際 JIT コンパイラと呼ばれる実行時コンパイラを採用し、この問題を解決している。しかし単純なJITコンパイラでは、Java言語の特徴をもたらす型安全性や多態性によるオーバヘッドを解消することは出来ない。従って、プログラムの安全性と再利用性という特徴を持ちながら、C言語やC++言語と同等の性能を得るためには、これらの特徴の実装に伴う性能のオーバヘッドをコンパイラの最適化によって削減することが重要である。

クラス階層における多態性を実現する動的束縛の導入によって、仮想メソッド呼び出しにおいて呼び出し先が複数になる可能性が生じるので、コンパイラによるメソッド間に渡る解析を困難にする。さらに、実行時にクラスの新規ロードを許す動的クラスローディングの導入により、実行時にクラスの構成が変化する可能性が生じるので、メソッド間に渡る静的な最適化を適用すると以前の解析結果に基づいたコードを実行できなくなる場合がある。これらの原因によって、従来の言語で行われていたコンパイル時の単純なメソッド呼び出しのインライン展開が適用できなくなり、コンパイル時の最適化の範囲が狭められるので、生成されたネイティブコードの質が従来の言語より低下する可能性がある。さらに、仮想メソッド呼び出しにおける実行時の呼び出し先検索が実行時オーバヘッドとなる。

プログラムの安全性を保証するために型安全な処理系を効率よく実装する場合、静的な型検査だけでなく、実行時の型検査や例外検査が必要となる。これらのオブジェクトのキャスト時の型検査やメモリアクセス時の例外検査が実行時オーバヘッドとなる。さらに、Java 言語は不正なアドレスを持たない安全なメモリアクセスを保証しているので、例外検査命令とメモリアクセス命令の間に順序制約をもたらす。この結果、命令移動を伴う最適化の適用が制限されるので、生成されたコードの質が従来の言語より低下する可能性がある。

従って、多態性を持つオブジェクト指向言語の実装におけるオーバヘッドは以下の 1.であり、型安全な言語の実装におけるオーバヘッドは以下の2.、3.、4.である。

1. 動的クラスローディングを行う言語における仮想メソッド呼び出し
2. オブジェクトの型検査
3. 配列要素やインスタンス変数の参照に伴う例外検査
4. 安全な参照を保証するための命令間順序制約

以下では、上記の４つのオーバヘッドを削減する手法を示すとともに、７章からなる本論文の構成を示す。

第１章では、本論文の研究目的とその成果の概要を示す。

第２章では、研究背景として、Java 言語の特徴とその実装上の問題点を示す。さらに、コンパイラを用いた代表的なJava言語の処理系、そして本論文で提案する手法を実装したコンパイラプラットフォームである、IBM Java Just-In-Time コンパイラについて述べる。

第３章では、仮想メソッド呼び出しの高速化手法について述べる。プログラミングにおける柔軟性を提供する多態性を持つ仮想メソッド呼び出しでは、あるメソッド呼び出

し元において複数の呼び出し先を持つ可能性がある。この場合、プログラム全体に渡る解析を行いコンパイラの最適化の範囲を拡大するためには、仮想メソッド呼び出しに対して呼び出し先を一意に決定できるかどうか調べる。もし一意に決定可能ならば、直接デバーチャル化を適用してコンパイラの最適化の範囲を拡大する。このためには、プログラム全体のクラス階層の構成を調べるクラス階層解析を行わなければならない。動的クラスローディングを行う Java 言語では、実行中にクラス階層が変化して呼び出し先のメソッドがオーバライドされた場合、直接デバーチャル化されたコードを無効化しなければならない。この無効化を行うために、実行中のメソッドを再コンパイルする脱最適化という手法が提案されているが、複雑な実装を必要とする。本章では、動的クラスローディングを行う Java 言語において、実装が容易な仮想メソッド呼び出しの直接デバーチャル化手法を提案する。本手法では、仮想メソッド呼び出しに対して直接デバーチャル化されたコードとメソッドがオーバライドされた場合に実行する仮想メソッド呼び出し、の２種類のコードをコンパイル時に生成する。最初は前者を実行し、メソッドのオーバライドが起きたときにコードを書き換えて後者を実行する。本手法では、コード書き換えによって直接デバーチャル化されたコードを無効化するので、脱最適化において要求される実行中のコードの再コンパイルを行うための複雑な実装が不要である。一方、再コンパイルを不要にするためにコンパイル時に２種類のコードを用意するので、制御フロー上に合流点が生成される。一般に制御フローの合流点はコンパイラの最適化を妨げるが、合流点が存在しても十分な最適化を可能にする手法を提案する。

第４章では、オブジェクトの型検査の高速化手法について述べる。オブジェクトの型検査は、言語の型安全性を保証するための重要な機能の１つである。型検査は、代入文の右辺のオブジェクトのクラス型が、左辺のクラス型に正しく変換可能であるかどうかを調べることである。右辺のオブジェクトのクラス型が、左辺のクラス型の下位クラスに属するならば、そのオブジェクトはクラスの型変換可能である。静的な型付けを持つ Java 言語であっても、コンパイル時に全ての型検査が実行可能、なわけではない。メソッド間で受け渡される参照型のオブジェクトに対して例外検査を行う場合など、実行時型検査のほうが効率よい場合がある。本章では、型検査のうち頻繁に実行される検査部分をインライン展開することによって型検査を高速化する方法を提案する。この方法は、従来手法と比較して実装が簡単であり実行時のメモリ消費量が少ない上に、従来手法と同等の性能向上が得られることを実験によって示す。

第５章では、例外検査の高速化手法について述べる。Java 言語の言語仕様では、プログラムの安全性を保証するために、インスタンス変数のアクセス、配列要素のアクセス、メソッド呼び出しのオブジェクトの参照、等の際に例外検査を行うことが規定されている。例外検査は、プログラムの安全実行を保証するために重要な機能であり、例外検査によって不正なメモリへのアクセスは Java プログラムでは発生しない。しかし、プログラムの安全性が保証されていない C 言語や C++言語等に比べて、例外検査は実行時オーバヘッドとなり、実行速度を低下させる。従来、データフロー解析を用いてコンパイル時の冗長な例外検査の削減手法が提案されている。これらの手法によって例外検査を減少させることはできるが、ゼロにすることはできず、依然実行時に例外検査は残ってしまう。このため、０ページを読み出し不可にするオペレーティングシステム（OS）のメモリ保護機能を用いて例外検査を高速化する方法が提案されている。しかし、コンパイラの最適化支援のために０ページを読み出し可能にしている OS も存在する。この場合、メモリ保護機能を用いた例外検査の高速化手法を利用できない。本章では、発生する例外の種類を例外検査命令に埋め込むことによって、例外が発生しない通常実行される部分に生成されるコードを最小限に押さえ、例外検査を高速化する方法を提案する。また、OS のメモリ保護機能を用いた高速化手法を利用できない場合に、ハードウェアに用意された高速な条件分岐命令を例外検査のために使用する際、本手法を用いて例外検査をさらに高速化することができる。

第６章では、安全な参照を保証した命令間順序制約緩和手法について述べる。Java 言語の型安全性の保証はプログラムの安全性と信頼性を増加させるが、その保証のために実行時に行われる例外検査が導入されている。例えば、インスタンス変数のアクセス、配列要素のアクセス、メソッド呼び出しのオブジェクトの参照、等のメモリアクセス時におけるアドレス検査である。これらの検査は、ハードウェアに関する例外によってプログラマが意図しない異常なプログラム終了を起こさないことを保証している。しかし、これらの検査はハードウェアに関する例外を発生する命令との間に順序制約をもたらし、命令の並べ替えを伴うコンパイラの最適化の効率を低下させる。本章では、型安全性を保証する Java 言語において、非循環有向グラフ表現を用いて効率的に投機的命令移動を適用する枠組みを提案する。この枠組みを用いてハードウェアに関する例外を発生する命令に投機的命令移動を適用して、例外検査の命令とハードウェアに関する例外を発生する命令の間に存在する順序制約を除去する。従来、分岐命令の制御フローで表現されていた順序制約は、この枠組みでは命令間の辺として単純に表現される。この表現を用いて、投機的命令移動によるクリティカルパスの短縮を正確に見積り、不必要な投機的命令移動を抑制する手法を提案する。

第７章では、本論文における研究成果をまとめる。