Java Just-In-Time コンパイラにおける最適化手法

Optimization Techniques in a Java Just-In-Time Compiler

2002 年 12 月

石崎　一明

Kazuaki Ishizaki

目次

図目次

表目次

第1章 序論

Java 言語[1]は 1995 年に Sun Microsystems 社が発表したオブジェクト指向プログラミング言語である。Java 言語は、

- 仮想機械（Virtual Machine）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
- 仮想機械によるプログラムの安全検査（Verification）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

という特徴をあわせ持つ。これらの特徴は、ライブラリの使用によるプログラムの再利用性・複数プラットフォーム間でのプログラムの再利用性と、Java プログラムの実行時の安全性をもたらす。Java プログラムの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、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機械語（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すると実行速度が劣る。この性能面の問題を解決するために、現在では仮想機械の実装において Just-In-Time（JIT）コンパイラ[2]と呼ばれる動的コンパイラが採用されることが多い。Java 言語が登場した初期には単純な JIT コンパイラであっても、仮想機械上での実行オーバヘッドを減らすことが可能であり、かなりの性能向上が得られた。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性能に関するオーバヘッドは依然解消されず、これまで主流であった C 言語や C++言語等と同等の性能を得ることはできない。Java 言語の持つ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、コンパイラの最適化によって Java 言語の特徴の実装によって引き起こされる性能に関するオーバヘッドを削減することが重要である。従来言語と同等の性能が得られることによって、Java 言語は初めて実用的な言語となり研究・開発にと広く使用される。

前述に加え、Java 言語の処理系を実装する上でのオーバヘッドを以下に挙げる。Java 言語ではメモリ管理機構が自動化されているためユーザが明示的にメモリ解放を行うのではなく、自動ごみ集め（Garbage Collection、GC）が実行時に行われ不要なメモリを回収する。これにかかる時間が実行時オーバヘッドとなる。また、Java 言語ではマルチスレッド機能を持つため、スレッド間の同期機構が言語によって提供されている。この同期にかかる時間が、やはり実行時オーバヘッドとなる。これらのオーバヘッドも Java 言語の登場時には深刻な問題であった。現在では、高速な GC アルゴリズム[3]や、高速な同期システム[4]、の登場によってこれらの問題は解消されている。これらの結果、現在ではコンパイルされたコードの実行時間が性能に大き

な影響を与える。

本論文では、Java Just-In-Time コンパイラにおける最適化手法、その実装、評価を示す。特に、Java 言語が持つプログラムの安全性と再利用性の高さという特徴をもたらす型安全性と動的束縛の実装によって引き起こされる性能に関するオーバヘッドを削減するための最適化手法を示す。この結果、Java 言語が従来の言語と比べて同等な実用的な処理速度を得ることが出来る。実際に、OS のタスク切り替えのシミュレーションを行う Richards ベンチマーク[5]を用いて C 言語、C++言語、Java 言語の間で性能を比較した結果を図 1.1に示す。この結果から、現在では Java 言語を用いても C 言語、C++言語と同等以上の性能が得られることが分かる。

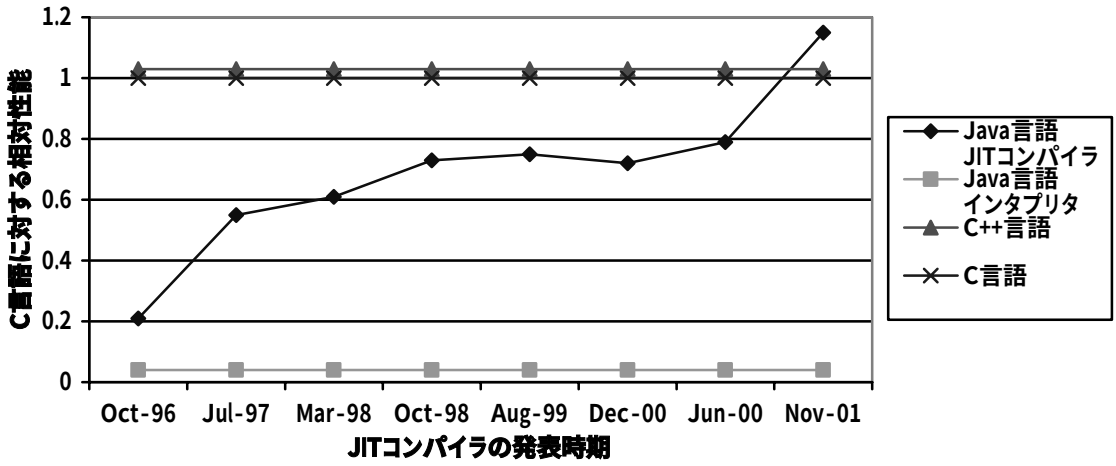


図 1.1: 異なる言語間における性能比較

（Richards benchmark using IBM DK Java Technology Edition for AIX）

本論文の成果は IBM 社 IBM Developers Kit, Java Technology Edition で実際に使用され、Java 言語が発表されて以来、図 1.2に示される約 10 倍の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 Just-In-Time コンパイラでも使用されている。

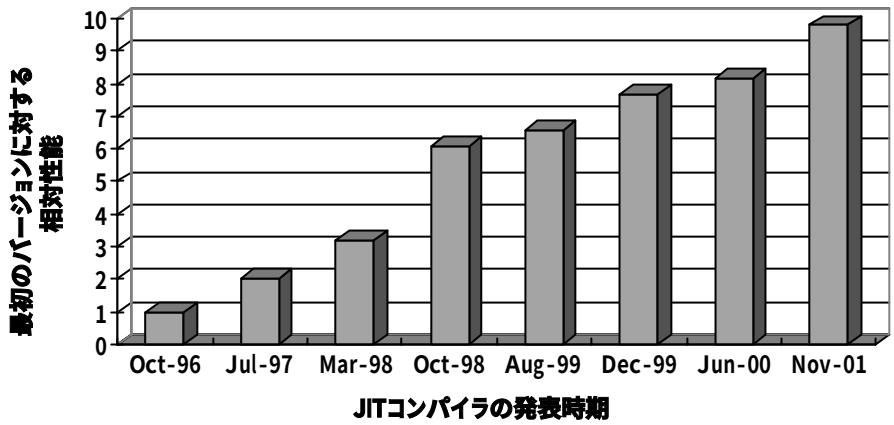


図 1.2: Java JIT コンパイラの性能向上

(SPECjvm98[6] using IBM DK Java Technology Edition for AIX)

1.1 研究目的と成果

本論文の目的は、Java Just-In-Time コンパイラにおいて、Java 言語が持つ特徴に起因するオーバヘッドを削減する最適化手法、実装、評価について示すことである。Java 言語では、プログラムの柔軟性を提供するために、動的クラスローディング、遅延解決、動的束縛を実現する仮想メソッド呼び出しによる多態性、を提供している。また、実行されるプログラムの安全性を保証するために、実行時の型検査や例外検査が導入されている。これらの機能は、ユーザやプログラマの使い勝手を大きく向上させる。一方、これらの機能の実装に伴うオーバヘッドがプログラムの性能に大きな影響を与える。

クラス階層におけるメソッド呼び出しに関する多態性を持つ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、コンパイル時に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プログラム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 C 言語や C++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いてメソッド呼び出し時に行われる呼び出し先の検索が実行時オーバヘッドとなる。また、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は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 言語で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。

Java プログラムの実行性能を向上させる高速な Java 言語処理系を実現するためには、Java 仮想機械に JIT コンパイラを導入し、実行時に Java バイトコードを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手法が一般的である。この場合、JIT コンパイラが生成するコードの質が実行性能に関する大きな鍵を握る。従って、ネイティブコードで発生する Java 言語の特徴的な機能の実装に伴うオーバヘッドをコンパイラにおける最適化によって低減することが重要である。Java 言語の実装におけるオーバヘッドを図 1.3に分類した。本論文ではネイティブコードにおけるオーバヘッドの削減を目的とするので、Java JIT コンパイラにおける最適化について示す。ネイティブコードにおけるオーバヘッドは、多態性を実装するオブジェクト指向言語が持つものと、型安全な言語が持つものに分類できる。

オブジェクト指向言語における多態性は、一般には動的束縛を用いた仮想メソッド呼び出しによって実現される。動的束縛を用いた仮想メソッド呼び出しは、複数の呼び出し先をとる可能性があるためコンパイル時のメソッド間最適化を妨げる。また、動的クラスロードの導入に

よって実行時にプログラム構成が変更されるため、コンパイル時の解析結果が正しくなくなることがある。また、仮想メソッド呼び出し時にメソッド検索が必要であり、これが実行時オーバヘッドとなる。さらに、型安全な言語において、効率的にプログラムの型安全性を保証するためには、実行時の型検査が必須である。このため、オブジェクトのキャスト時の型検査や、オブジェクトやインスタンス変数へのアクセス時のアドレス検査が必要になる。これらの検査が実行時オーバヘッドとなる。また、安全なメモリ参照を保証する必要があるので、コンパイラによる命令移動が制限されることがありコンパイル時の最適化を妨げる。

従って、多態性が導入された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

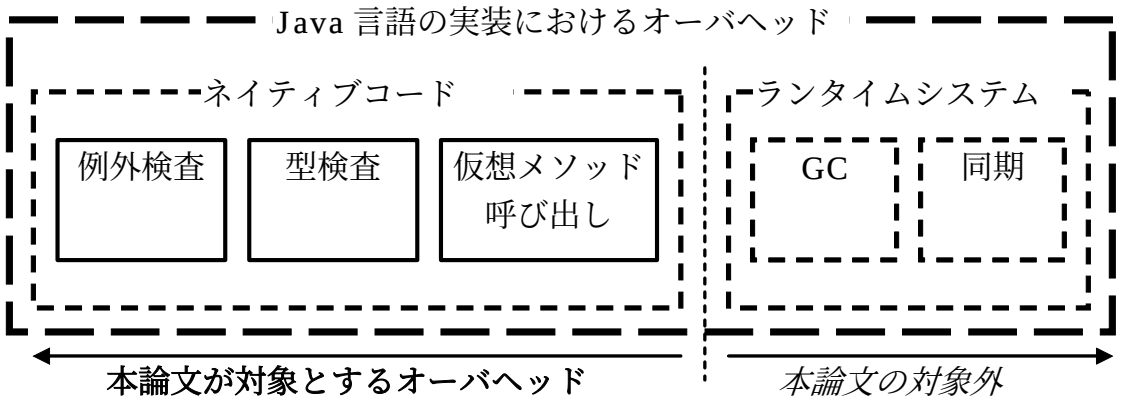


図 1.3: Java 言語の実装におけるオーバヘッドと本論文の対象範囲

1.1.1 仮想メソッド呼び出しの高速化

Java 言語では、オブジェクト指向の特徴の１つである多態性を仮想メソッド呼び出しによって実現することで、プログラミングにおける柔軟性を提供している。多態性を持つ仮想メソッド呼び出しでは、ある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。Java 言語に代表される動的クラスローディングを行う言語では、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、この手法は複雑な実装を必要とする。

本論文では、Java 等の動的クラスローディングを行う言語において、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。

1.1.2 オブジェクトの型検査の高速化

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。Java 言語に代表される静的な型付けを持つ言語であっても、コンパイル時に全ての型検査が実行可能、というわけではない。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、実行時にオブジェクトの実クラスが決まった際に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。

本論文では、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可能する方法を提案する。この方法は、従来提案されている方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、従来の方法と同等の性能向上が得られることを実験結果から示す。

1.1.3 例外検査の高速化

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。この検査によって Java プログラムでは許されないメモリへのアクセスは発生しない。これらの検査は、プログラムの安全な実行を保証するために重要な機能である。しかし、C 言語や C++言語等のプログラムの安全性が保証されていない言語に比べて、これらの検査は実行時オーバヘッドとなる要因であり、実行速度を減少させる。従来、データフロー解析を用いてコンパイル時に冗長な例外検査を削減する方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（Operating System、OS）のメモリ保護機能を用いて高速に処理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用いることができない。

本論文では、発生する例外の種類を例外検査の命令に埋め込むことによって、例外の種類を示す命令を生成することなく、例外が発生せずに通常に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、この方法を用いて例外検査をさらに高速化することができる。

1.1.4 安全な参照を保証する命令間順序制約の緩和

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。

本論文では、Java 言語に代表される型安全な言語において、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によって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。

1.2 提案最適化の効果

本論文で提案した最適化手法を、全て適用した場合と全て適用しなかった場合の性能を SPECjvm98 ベンチマークを用いて、PowerPC、IA-32、IA-64、の３つのプラットフォームで比較した。その結果を、図 1.4に示す。PowerPC では、1.1.1節、1.1.2節、1.1.3節で述べた最適化手法を実装して比較した。IA-32 では、1.1.1節、1.1.2節で述べた最適化手法を実装して比較した。IA-64 では、1.1.1節、1.1.2節、1.1.4節で述べた最適化手法を実装して比較した。

PowerPC では、mtrt の 3.0 倍を最高として、相乗平均で 1.60 倍の性能向上が得られた。IA-32 では、mtrt の 2.2 倍を最高として、相乗平均で 1.26 倍の性能向上が得られた。IA-64 では、mtrt の 2.2 倍を最高として、相乗平均で 1.33 倍の性能向上が得られた。どのプラットフォームでも、mtrt の性能向上が大きいが、1.1.1節で述べた仮想メソッド呼び出しの高速化による効果が大きい。それぞれの最適化の性能向上への寄与の度合いは、この後の各章の実験結果で示す。

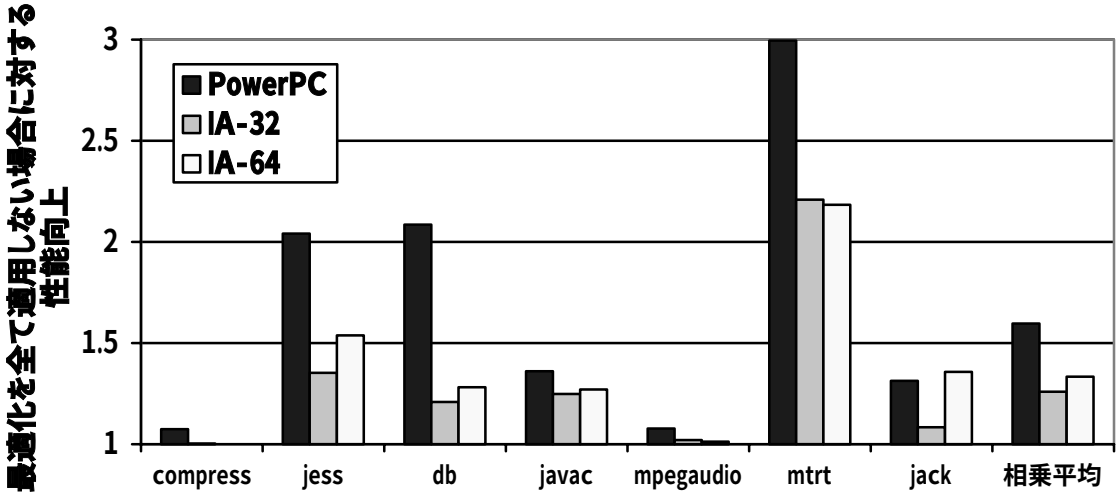


図 1.4: 提案最適化を全て適用した場合の効果

1.3 本論文の構成

本章では、本論文の研究目的と成果、その効果について述べた。次に、第 2 章で研究背景の説明として、Java 言語とコンパイラを備えた代表的な処理系の実装について述べる。以降、第 3 章では、仮想メソッド呼び出しのオーバヘッド削減手法について述べる。第 4 章では、オブジェクトの型検査のオーバヘッド削減手法について述べる。第 5 章では、例外検査のオーバヘッド削減手法について述べる。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。最後に、第 7 章で、本論文の研究成果をまとめる。

第2章 背景

本章では、本論文が対象とする Java 言語と、コンパイラを用いた代表的な Java 言語の処理系の実装について述べる。

2.1 Java 言語

Java 言語[1]は 1995 年に Sun Microsystems 社が発表したプログラミング言語であり、言語仕様[7]、仮想マシン仕様[8]、標準 API 仕様[9]、から規定されている。Java 言語は、仮想機械（Virtual Machine）によるプログラムのプラットフォーム独立性、型安全性（Type Safety）と仮想機械によるプログラムの安全検査（Verification）の実行によるプログラムの安全性の保証、オブジェクト指向と遅延解決によるプログラミングの柔軟性、といった特徴を持つ。これらの特徴によって、Java 言語は急速に広く受け入れられた。以下では、それぞれの特徴について述べる。

2.1.1 プラットフォーム独立性

コンピュータの存在が一般的となった現在では、携帯電話や Personal Digital Assistants（PDA）に代表される小型機器から、家電製品、パーソナルコンピュータ、Web サーバ、広域環境での分散計算（GRID コンピューティング）まで、異なるアーキテクチャが混在するコンピュータ環境は当然のものとなっている。この環境において、プラットフォーム独立性の高いプログラムを記述することができる Java 言語は広く受け入れられている。

Java 言語は、その実行環境としてプラットフォームに依存しない共通の Java 仮想機械（Java Virtual Machine、Java VM）を仮定している。Java 言語は、各プラットフォームにおいてこの仮想機械の実装を前提とすることによって、実行プラットフォームの違いを吸収している。Java 言語で書かれたプログラムは Java 仮想機械用の実行形式であるバイトコードにコンパイルされた後、仮想機械上で実行されるため、いかなる環境下においても再コンパイル無しにプログラムが実行可能である。従って、Java 言語ではソースレベルまたはバイトコードレベルでの移植性の問題を意識する必要はない。さらに、１つのソースコードで様々なプラットフォームで実行可能なアプリケーションを開発することが可能である。

現在の CPU の多くはレジスタマシンアーキテクチャを採用している。レジスタマシンアーキテクチャでは、高速アクセス可能な小容量の記憶装置であるレジスタを CPU 内に持ち、頻繁に使われる演算結果の保持などに利用して高速な処理を実現している。このレジスタの個数やサイズなどは各 CPU によって異なる。Java 言語を実行するためには、各プラットフォーム上に Java 仮想機械を構築する必要がある。ここで、Java 仮想機械のアーキテクチャとしてレジスタマシンアーキテクチャを採用すると、各プラットフォームにおけるレジスタの個数やサイズの違いが大きな問題となり移植性が低くなる。この問題を回避するために、Java 仮想機械ではス

タックマシンアーキテクチャを採用した。スタックマシンアーキテクチャでは計算を行う過程で、オペランドとして FIFO キューだけを仮定し、レジスタの数量や存在を仮定しない。そのため、レジスタが存在しない、またはレジスタの個数が少ないプラットフォームにおいても Java 仮想機械を実装することが可能である。さらに、レジスタの個数やサイズが異なるプラットフォーム間でも、Java 仮想機械用の同一のプログラムを実行することが可能である。

2.1.2 プログラムの安全性

近年、プログラムのセキュリティに関する問題が、しばしば話題になる。特に、Nimda、CodeRed 等に代表されるバッファオーバフローによるプログラムへの攻撃[10]は、代表的な攻撃方法の１つである。これらの攻撃を許すのは、攻撃対象となるプログラムに誤り（バグ）があることが大きな原因である。さらに、実装に用いられた言語が正しくないメモリ領域の読み書きを許すこと[11]も原因の１つである。後者の原因は、実装に用いられた言語が型安全性を保証できるならば、型安全なプログラムのみを実行することによって解決可能である。

型安全性とは、与えられたプログラムを実行して得られた出力データがある制約条件を常に満たす、ことをいう。型検査によって、プログラム中で実行される演算が適正な型を持つデータに適用されるかどうかを調べられる。適正でない型を持つデータに演算が適用されることを型誤りと呼び、型誤りが発生しないプログラムを型安全なプログラムと呼ぶ。型安全なプログラムは、予期しないメモリ領域を読み書きしないという意味においても安全であることが保証される。

Java 言語では、静的な型付けの導入と Java 仮想機械におけるスタックマシンアーキテクチャの導入によって、固定個数のスタックオペランドと局所変数から構成される非常に小さいワーキングセットの型を、コンパイル時に容易に決定可能である。従って、静的な型検査を低い解析コストで可能にしている。Java 仮想機械では、安全検査器（Verifier）によって、実行直前に静的な型検査が行われ、プログラムの正しさが確かめられる。また、Java 言語等の静的に型付けられた言語においても、実行時に決定されるオブジェクトの型に関する検査を静的な型検査では行うことは困難であるので、動的な型検査も必要となる。Java 言語ではキャスト演算子や一部の代入文で動的な型検査を行い、またメモリアクセス時にアドレスの正しさを確かめる例外検査を行う。これらの検査で誤りが検出されると例外を発生し、安全に実行を中断する。これらの結果、Java 言語は型安全性を保証する。従って、バッファオーバフローによるプログラムの攻撃に対して、Java 言語のプログラムは安全であることが保証される。

Java 言語のプログラムでは、メモリが必要になるとプログラマは new 文によってオブジェクトを確保し、不要になったメモリの開放は仮想機械が備える GC が自動的に行う。GC を用いて言語処理系がメモリ管理を自動化することによって、プログラマはメモリ管理に労力を割く必要がなくなる。また、確保したメモリの解放し忘れやメモリの解放間違いによるメモリのリークが無くなるため、プログラムの信頼性と安全性が増す。

これらの型安全性の保証と自動メモリ管理の導入によって、Java 言語で書かれたプログラムは非常に安全に実行可能である。

2.1.3 プログラムの柔軟性

Java 言語はオブジェクト指向言語である。オブジェクト指向を利用することにより問題の記述性やコードの再利用性が向上することは、これまでの研究によって示されている。オブジェクト指向はその特徴を実現するために多くの機構を提供するが、Java 言語では単純な設計を目指してオブジェクト指向の根幹をなす以下の機構を備える。その単純な設計のため、プログラマは容易にオブジェクト指向の恩恵を受けることができる。

- **継承（Inheritance）** 新しいオブジェクトを作成する場合に、あるオブジェクトを拡張することによって、特化したオブジェクトを作りたい場合がある。または、複数のオブジェクトの共通点を抜き出し汎化したオブジェクトを作成し、そこから各オブジェクトを作る場合もある。これらの要求に答えるのが継承の概念である。オブジェクトを作成する場合は、継承元のオブジェクトの性質はそのままに、新たに差分のみを記述すればよい。従って、コードの再利用性が向上する。
- **カプセル化（Encapsulation）** オブジェクト指向では、データはオブジェクトの中に隠蔽される。つまり、データに対する操作はそのオブジェクトに対する決められた手続きであるメソッド呼び出しのみを許す。このために、オブジェクトが持つデータ構造に変更が加えられても、そのデータへのインタフェース（メソッド）の形式が変わらなければ、他のオブジェクトからはその変更が見えない。従って、そのオブジェクトへアクセスする場合も、オブジェクト内部のデータ構造やメソッドのロジックについて考慮する必要はなく、オブジェクトの独立性が高まる。この結果、オブジェクト内部の仕様変更が外部に影響しなくなり、ソフトウェアの保守性や開発効率が高まり、コードの再利用性が向上する。
- **多態性（Polymorphism）** 多態性とは、1 つ以上の型を取ることの出来るプログラムエンティティを定義する能力である。オブジェクトに対するインタフェース（メソッド）の実装は、その時に取るオブジェクトの型によって意味が異なる。多態性を実現するためには、動的束縛（dynamic binding）と呼ばれる、実行時に型を検査して適したメソッドを探索する動的な関連づけが必要となる。

これ以外の、例えば多重継承（multiple inheritance）や演算子のオーバーロード等は、複雑さやプログラムの理解の妨げのため実装されていない。

さらに、Java 言語はその登場時から豊富な標準 API を持ち、プログラムの部品化や再利用を強く意識していた、といえる。プログラムの部品化を強く押し進めるために、Java 言語では実行時にクラスが要求されたときにクラスをロードする動的クラスローディング（dynamic classloading）、実行時に参照が発生した時点で初めてシンボリックな参照を解決する遅延解決

（lazy resolution）を行う。

C++言語[12]では、実行前のリンク時に静的なクラスローディング（static classloading）とスーパークラスへのシンボリックな参照を解決する静的解決（static resolution）を行う。静的クラスローディングと静的解決では、部品として提供されるクラス、もしくはそのスーパークラスの構造（具体的にはメソッドやフィールドの数など）が修正された場合でも、そのクラスを参照する全てのサブクラスを再コンパイルしなければならない。これは「脆弱なスーパークラス」の問題と呼ばれていて、言語の仕様と代表的な実装に由来するものであり、解決することは非常に困難である。

一方、Java 言語ではスーパークラスの構造が修正された場合でも、実行前に修正されたクラスだけを再コンパイルすればよい。なぜなら、変更されたメソッドやフィールドが実行時に実際に参照される時に初めて動的クラスローディングが行われ、メソッドやフィールドのシンボリックな参照を解決する遅延解決を実行時に行うからである。この方法によって、Java 言語では「脆弱なスーパークラス」の問題を解決している。

動的クラスローディングは、プログラミングの柔軟性を高める以外にも利用される。例えば、シンボリックな参照の解決時に１回だけ型検査を行うことで、少ない実行時オーバヘッドで型検査を行う。また、クラスをロードしてくるリモートの場所を指定する、クラスに適切なセキュリティ属性を割り当てる、等の独自のクラスローダを定義可能である。このクラスローダを用いて、それぞれのクラスにプログラム中で個別の名前空間を提供することも可能である[13]。一方、コンパイラの最適化の面から見ると、実行時に新たなクラスがロードされる可能性を常に考える必要があるため、プログラム全体に対する最適化が阻害される可能性がある。

これらの機能によって、Java 言語ではプログラムの部品化や再利用性の高さを利用した、柔軟なプログラミングが可能である。

2.2 Java 仮想機械

Java 言語が持つプログラムのプラットフォーム独立性は、2.1節で述べたバイトコードの「仮想機械によるプログラムの実行」という計算モデルを用いていることに由来する。

仮想機械ではバイトコードを解釈実行することにより、マルチプラットフォームな実行環境を可能にしている。仮想機械は、実際のマシンアーキテクチャとは異なり、ソフトウェアで実装された実行環境である。このため、マルチプラットフォームな実行環境の実現という利点と引き替えに、実行速度の低下という欠点を持つ。仮想機械は基本的にインタプリタであるため、以下の手順をバイトコード中の１つ１つのオペコードに対してソフトウェアで行わなければならない。

1. オペコードを読み出す。
2. そのオペコードの処理内容を実装したサブルーチンを呼び出す。
3. 実行する。

従って、各プラットフォーム依存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計算モデルを用いた言語と比較すると実行速度が劣る。

この性能面の問題を解決するために、現在では Java 仮想機械の実装において JIT コンパイラ[2]が採用されることが多い。JIT コンパイラは、一般的には動的コンパイラと呼ばれ、高速実行が必要になった段階でプログラムをコンパイルしてネイティブコードを生成し、実行環境に依存したコードを実行する。Lisp や Smalltalk の時代から研究されてきた技術であるが、Java 言語の実装において初めて実用的な技術として広く用いられた。

現在の典型的な Java の実行環境、特に Java 仮想機械の実装を図 2.1 に示す。Java 仮想機械における主なコンポーネントは次に示す役割を行う。

- **Bytecode Verifier** バイトコードの正当性を確認する安全検査器。
- **Bytecode Interpreter** バイトコードで書かれたメソッドを、逐次解釈・実行するインタプリタ。
- **Memory System** GC を含むヒープ管理を行うメモリシステム。
- **Just-In-Time Compiler** バイトコードで書かれたメソッドをコンパイルして、ネイティブコードを生成する動的コンパイラ。

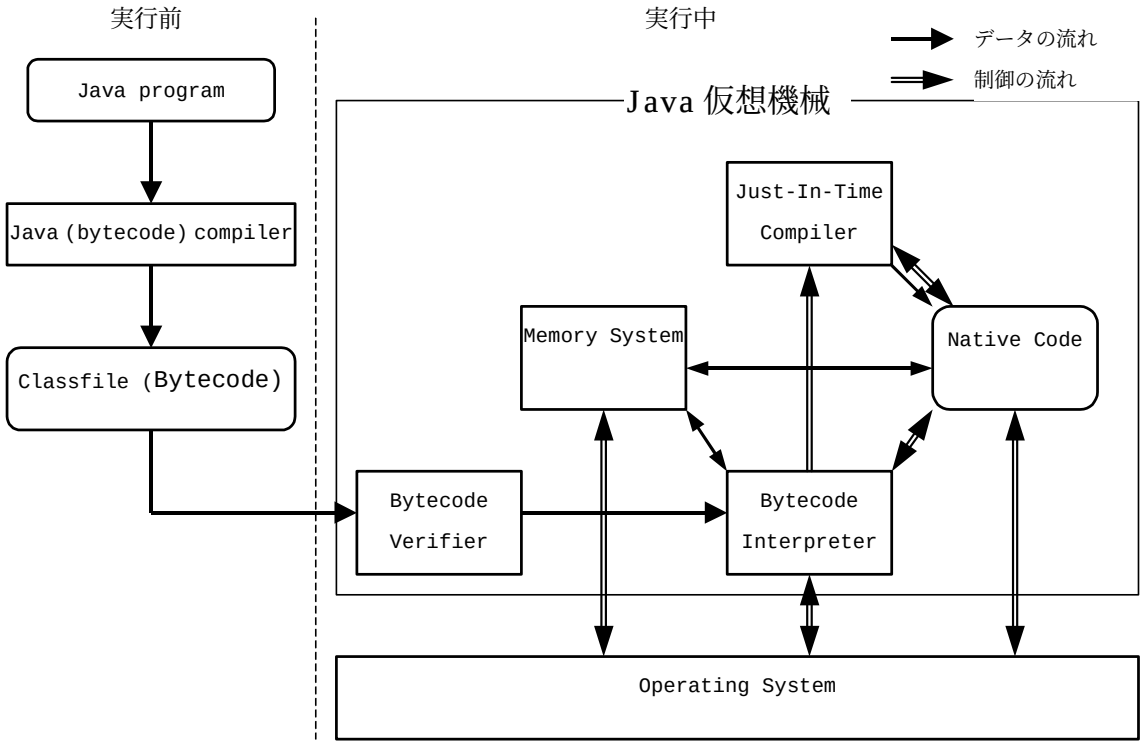


図 2.1: 典型的な Java 言語の処理系の構造

Java 言語で記述されたプログラムは実行前に Java バイトコードコンパイラによって Java クラスファイルに変換される。Java 仮想機械はこの Java クラスファイルを読み込んでプログラムを実行する。Java 仮想機械は、メソッドの実行前に読み込んだ Java クラスファイルの正当性を必要に応じて安全検査器によって確認する。その後、各メソッドの実行を開始する。

各メソッドの実行開始時には、メソッドはインタプリタによって実行される。与えられた判

断基準（一般には頻繁にメソッドが実行された時）に基づいて、各メソッドに対して JIT コンパイラが起動され、メソッドのコンパイルを行う。コンパイルが終了すると、コンパイラの生成したネイティブコードを使ってメソッドが実行される。この結果、プログラムの実行性能が向上する。

2.3 コンパイラを持つ Java 言語処理系

Java 言語の処理系を利用する際には、その実行速度の観点からなんらかの方式でネイティブコードへ変換するコンパイラを持つ処理系を利用するのが、現在の標準であると考えられる。Java バイトコードのクラスファイルをネイティブコードに変換して実行する場合、ネイティブコードへのコンパイル方式は、プログラムの実行前にネイティブコードにコンパイルする静的コンパイル方式と、プログラムの実行中にネイティブコードにコンパイルする動的コンパイル方式、の２つが考えられる。もちろん、この両者を組み合わせることも可能である。

本節では、静的コンパイル方式と動的コンパイル方式についてその得失を述べ、それぞれの代表的な Java 言語の処理系を示す。

2.3.1 静的コンパイル方式

プログラムの実行前に、クラスファイルをネイティブコードにコンパイルするのが静的コンパイル方式である。この方式では、コンパイル時間がプログラムの実行時間に含まれないので、大量のコンパイル時間を消費する高度な最適化を適用することができる。また、実行時にコンパイラが起動されないので、実行時に消費する作業メモリ量が少ない、プログラムの立ち上がり時間が短い、等の利点がある。

さらに、動的クラスローディングが発生しない閉じた環境（closed world）を仮定して、プログラム全体に対するクラス階層解析[14][15]や脱出解析[16]などの最適化を適用すると、動的クラスローディングを行うと仮定した環境下より、適用機会が増える。なぜならば、コンパイル時に存在しないメソッドは実行中に現れることが決してなく、コンパイル時にオーバライドされていないメソッドは永久にオーバライドされることが無い。従って、コンパイル時において、静的束縛やスレッドから脱出しないオブジェクトの検出の機会が増える。しかし、この仮定は動的クラスローディングを前提とする Java 言語の仕様に反する。

静的なコンパイル方式は、静的クラスローディングを行う C++言語では標準的なコンパイル方式である。Java 言語では、動的クラスローディングが前提であるので、単純に静的なコンパイル方式を用いると Java 言語の動的クラスローディングや遅延解決といった機能に制限が加わる。この結果、Java 言語の一部の機能を利用できないという問題を生じる。一部の処理系ではこの欠点を回避するために、動的クラスローディングされるクラスファイルをあらかじめコンパイルしておく、Java バイトコードのみを持つクラスファイルとコンパイルされたネイティブコードが混在した状況下での実行を可能にする、実行時に静的コンパイラを起動する、等の方

法を用いて問題を解決している。また、静的にコンパイルされたネイティブコードを配布するため、プログラム実行のために配布されたバイナリファイルがプラットフォーム依存性を持つ、という欠点が存在する。

静的コンパイル方式を用いる処理系として、以下のものが挙げられる。

- **SUIF Java**　SUIF Java [17]は、SUIF [18]にオブジェクト指向プログラミング言語特有の中間表現を追加した OSUIF[19]と、Java 言語用のコンパイラを用いる際に必要となる OSUIF の Java 言語用フロントエンドと、OSUIF を用いてコンパイルして得られたネイティブコードを実行するために必要なランタイムシステムから構成される。
- **JOVE**　JOVE は、中間言語として静的単一代入則（Static Single Assignment、SSA）[20] 形式を用いたコンパイラである。静的クラスローディングを仮定して、プログラム全体の最適化、またオブジェクト指向言語向けの最適化を行う。ランタイムシステムにおいては、世代 GC を用いている。
- **Vortex**　Vortex [21]はオブジェクト指向プログラミング言語向けの最適化コンパイラである。Vortex は C++言語、Modula-3 言語、Java 言語、純粋なオブジェクト指向言語である SmallTalk 言語または Cecil 言語、を Vortex Register Transfer Language（RTL）に変換し、最適化を行い、SPARC のアセンブラか C++言語のソースコードを生成する。

  Vortex は、局所クラス解析[22]、クラス階層解析[23]、プロファイルを用いたレシーバクラス予測等のオブジェクト指向言語向けの最適化を行う。また、共通部分式削除や不要コードの除去等、従来から知られている最適化も行う。
- **Caffeine**　Caffeine[24]は Impact[25]上で実装された、Java バイトコードからネイティブコードへのコンパイラである。Java バイトコードが対象とするスタックマシンアーキテクチャから、ネイティブコードが対象とするレジスタマシンアーキテクチャへの変換方式に特徴を持つ。
- **HPCJ**　High Performance Compiler for Java[26]は Java バイトコードを直接ネイティブコードに変換するコンパイラである。ループ内で例外が発生しない安全領域を生成する[105]、等例外除去に関する最適化を特に行っている。

  実行中に動的クラスローディングが行われるクラスは、あらかじめ HPCJ でコンパイルされたネイティブコードが用意されている必要がある。
- **TOWER J**　Tower J[27]は、Java バイトコードを直接ネイティブコードに変換するするコンパイラである。実行中に動的クラスローディングが行われるクラスは、あらかじめ TOWER J コンパイラでコンパイルされてネイティブコードが用意されている必要がある。
- **JeanPaul**　JeanPaul[28]は、静的コンパイル方式と動的コンパイル方式を組み合わせた Java の実行環境である。静的コンパイラがコンパイル時に参照したクラスが、実

行時に動的にクラスローディングされたものと同一であることを確認後、静的コンパイルによって生成されたネイティブコードを実行する。参照したクラスが更新されていた場合には、動的コンパイラによってコンパイルしたネイティブコードを実行する。

- **GCC**　GNU Compiler Collection（GCC）[29]は C 言語、C++言語、Objective-C 言語、Fortran 言語および Java 言語に対応するコンパイラであり、非常に多くのアーキテクチャ・システムに対応していることが特徴である。
  多くのプラットフォームにおいては、静的にコンパイルされてネイティブコードが用意されたクラスのみが実行可能である。一部のプラットフォームでは静的にコンパイルされたクラスと、Java バイトコードで存在するクラスファイルを混在させて、インタプリタ上で実行可能である。
- **JET**　JET[30]は、SSA を中間表現として用いた、メソッド呼び出しのインライン展開、共通部分式削除、不要コード除去、彩色グラフを用いたレジスタ割り付け、命令スケジューリング等、数多くの従来から知られている最適化を行うコンパイラである。さらに、クラス解析を用いて仮想メソッド呼び出しやインタフェースメソッド呼び出しを直接メソッド呼び出しに変換する。また、例外除去の最適化[31]や、脱出解析[16]によるオブジェクトのスタックへの割り付け等、Java 向けの最適化も多く実装されている。
  あらかじめネイティブコードが用意されていないクラスが実行時に動的にローディングされた場合、JET コンパイラを実行時に起動してネイティブコードを生成して実行を続ける mixed compilation mode を提供している。
- **Turbo Chai**　Turbo Chai[32]は、Java バイトコードから C 言語のソースコードを生成するコンパイラと、仮想機械を含む実行環境からなる。Turbo Chai は、組み込み環境用に開発された実行環境である。提供される実行環境では、コンパイルされたネイティブコードと Java バイトコードで存在するクラスファイルが混在した実行が可能である。
- **Harissa**　Harissa[33]は、Java バイトコードから C 言語のソースコードを生成するコンパイラである。クラス階層解析を行い、仮想メソッド呼び出しを直接メソッド呼び出しに変換する。型検査の除去やメソッド呼び出しのインライン展開も行う。さらに、静的にコンパイルされたクラスと、Java バイトコードで存在するクラスファイルを混在させて実行可能である。

2.3.2 動的コンパイル方式

プログラムの実行中に、クラスファイルをネイティブコードにコンパイルするのが動的コンパイル方式である。この方式では、実行前にコンパイルされたネイティブコードを用意する必

要がないので、バイトコードの実行に関して完全にプラットフォーム独立性を保つことができる。Java 言語では動的クラスローディングが前提であり、さらに実行時に遅延解決を行う。従って、実行時にクラスをネイティブコードにコンパイルする際にシンボルの解決を行い、ネイティブコードを実行する、という動的コンパイル方式は、Java 言語の実行環境に合致した方式であると考えられる。

動的コンパイル方式では、コンパイル時間がプログラムの実行時間に含まれるので、大量のコンパイル時間を消費する高度な最適化を適用しにくい。また、実行時に JIT コンパイラが起動されるので、実行時に消費するメモリ量が多い、プログラムの立ち上がり時間が長い、等の欠点がある。これらの欠点を解決するために、インタプリタと、短いコンパイル時間で高度な最適化は行わない JIT コンパイラと、長いコンパイル時間で高度な最適化を行う JIT コンパイラ、を適当に使い分ける adaptive compilation と呼ばれる実行方式が提案されている[34][35]。この方式によって消費メモリ量と実行時間の問題を解決し、性能向上も実現している。さらに、メソッドの起動回数、メソッド呼び出しのコールチェイン、メソッドの呼び出し元毎のレシーバ情報等、実行時のプロファイル情報を記録し、その情報を利用して頻繁に実行される部分に高度な最適化[36][37]を適用することもできる。この実行時の情報を用いた最適化は静的コンパイル方式では困難であり、静的コンパイル方式では得られない高い性能を実現可能であると考えられる。

動的コンパイル方式を用いる代表的な処理系として、以下のものが挙げられる。

- **Java Classic VM**　Sun の Classic VM[38]は、一般に公開された最初の Java VM である。ハンドルを持つオブジェクトを用いたメモリシステムや、保守的な GC が実装されている。実行される全てのメソッドが JIT コンパイラによってコンパイルされる。
- **Java HotSpot Virtual Machine**　Sun の HotSpot Virtual Machine[39] は、ハンドルの無いオブジェクトを用いた型厳密（type accurate）な GC[40]を行うメモリシステム、スレッド管理、高速な同期方式、等の特徴を持つ Java VM である。
  インタプリタを用いてプログラム内で頻繁に実行される”hot-spots”部分を検出し、その部分を含むメソッドに対して HotSpot コンパイラを起動し高度な最適化を適用してコンパイルする。”hot-spots”に関して収集される情報には、仮想メソッド呼び出し時の呼び出し元と呼び出し側の関係も含まれる。サーバ向けの HotSpot コンパイラでは、SSA 形式を用いた最適化として、共通部分式削除、不要コード除去、彩色グラフを用いた大域レジスタ割り付け、等を行う。さらに Java 用の最適化として冗長な例外検査の除去、仮想メソッド呼び出しのインライン展開も行う。また、動的クラスローディングによってコード最適化時の前提が成立しなくなった場合、そのネイティブコードの実行中に最適化されていないコードへ実行を移す、脱最適化[41]を実装している。
- **IBM Developer kit, Java Technology Edition**　IBM の Java VM は、Sun の Classic VM

を元に、ハンドルの無いオブジェクトを用いた保守的な GC を含むメモリシステム、同期方式[44]、JIT コンパイラ[43]、等に大幅な改良が行われた Java VM である。

JIT コンパイラでは、定数伝搬、不要コード除去、共通部分式削除、等様々な最適化を行っている。また、SSA 形式を用いた最適化も行っている。さらに、Java 用の最適化として、インスタンス変数アクセスに関する共通部分式の除去[45]、例外処理の高速化[46]、型検査の高速化[47]、仮想メソッド呼び出しの最適化[48]、等を行う。メソッドの呼び出し回数とループの実行回数を元にして"hot-spots"部分を検出し、その部分を JIT コンパイラでコンパイルしている。プラットフォームによっては、タイマーサンプリングによってネイティブコードから"hot-spots"部分を検出し、実行時に得られた情報を用いて再コンパイルを行い、より最適化されたコードを生成する機能も実装されている[35]。

本論文で対象とするこの処理系については、2.4節で詳細を述べる。

- **Jikes RVM** Jikes RVM は、Java 言語で記述された研究用の Java VM[49]で、全てのメソッドがコンパイルされる、様々なバリエーションの型厳密な GC が実装されている、コンパイラによって生成される安全点による quasi pre-emptive なスレッド切替えを行う、という特徴を持つ[50]。quick、optimizing の２つの JIT コンパイラとその最適化レベルを使い分けることにより、コンパイル時のオーバヘッドを減少しながら性能を向上させている。これらのコンパイラの選択は、メソッドの実行回数に基づいて決定される[34]。
- **Research VM** Sun Laboratories の Research VM[51]は、型厳密な GC やコード書き換えによる GC 安全点の生成[52]を含むメモリシステム、高速な同期方式[53]、インタプリタ・基本ブロック内の最適化を行う高速な JIT コンパイラ・メソッド呼び出しのインライン展開やメソッド全体に対する最適化を行う JIT コンパイラの３種類を使い分ける mixed-mode システム[54]、 などの特徴を持った Java VM である。
- **JUDO** Intel の JUDO[55]は、型厳密な GC を持つメモリシステムを持ち、全てのメソッドがコンパイルされる、という特徴を持つ Java VM である。

  fast と optimizing の２つのコンパイラを持ち、fast コンパイラでは optimizing コンパイラによる再コンパイル時に用いられる実行時情報を収集するためのコードも生成する。Optimizing コンパイラでは、メソッド呼び出しのインライン展開、拡張基本ブロック内での共通部分式削除、大域的な定数伝搬・不要コード除去・冗長な例外検査除去、等の最適化を行う。また、仮想メソッド呼び出しの直接呼び出しへの置き換えも行う。
- **Fast VM** COMPAQ の Fast VM[56]は、ハンドルを介した参照を行わないオブジェクトを用いて高速なメモリ割り付けや保守的な GC と複写式 GC を併せ持つメモリシステム、高速な同期方式、等の特徴を持つ Java VM である。JIT コンパイラは全ての

メソッドをコンパイルし、冗長な例外検査除去や、仮想メソッド呼び出しの直接呼び出しへの置き換え、プロセッサ特有の命令生成、等の最適化を行う。

- **Kaffe OpenVM**　Kaffe[57]は独自の Java 仮想機械を実装し、数多くの OS とプロセッサに対応している。多くのプラットフォームでは JIT コンパイラも実装されている。単純な JIT コンパイラは、Java バイトコードのそれぞれの命令に対するネイティブコード列のテンプレートを用意し、そのテンプレートを繋げてネイティブコードを生成する。中間表現を用いて最適化を行い、より高速なネイティブコードを生成する JIT コンパイラを持つプラットフォームもある。
- **LaTTe**　LaTTe[58]は Kaffe を元に GC を含むメモリシステム[59]、同期方式、JIT コンパイラなどを改良した Java VM である。コンパイラでは、拡張基本ブロックを単位として共通部分式削除や定数伝搬などの最適化を行う。Java 用の最適化として、メソッド呼び出し履歴に基づいた仮想メソッド呼び出しのインライン展開も行う。さらに、メソッドの起動回数に基づいた adaptive compilation も行う[60]。
- **OpenJIT**　OpenJIT[61]は Sun の Classic VM のプラグインとして動く JIT コンパイラである。OpenJIT は、Java 言語で記述されている。コンパイラは、中間表現として RTL を用い、peephole 最適化やレジスタ割り付けを行う。
- **ShuJIT**　shuJIT[62] は Sun の Classic VM のプラグインとして動く JIT コンパイラである。コンパイラは、Java バイトコードのそれぞれの命令に対するネイティブコード列のテンプレートを用意して、そのテンプレートを繋げてネイティブコードを生成する。スタックオペランドの先頭から最大２段分の値をレジスタに保持する最適化を行っている。また、静的メソッド呼び出しのインライン展開、末尾再帰、peephole 最適化、等も行っている。

2.4 IBM Java JIT Compiler

本論文では、提案する最適化手法を実装するコンパイラプラットフォームとして、IBM Developers Kit, Java Technology Edition に含まれる IBM Java JIT Compiler を用いた。IBM Developers Kit, Java Technology Edition は、高速性、安定性、サポートするアーキテクチャ・ＯＳのプラットフォームの多さのため、その発表時より幅広く用いられている。本節では、IBM Java JIT Compiler の構成と特徴について述べる。IBM Java JIT Compiler は、Java VM の構成要素の１つである JIT コンパイラである。1996 年に最初のバージョンが公開されて以来、数度の更新の度に内部構成も変化している。ここでは、IBM Developers Kit, Java Technology Edition, Version 1.3.1 に含まれる IBM Java JIT Compiler 4.0 について述べる。

JIT コンパイラの全体構成を、図 2.2に示す。Java VM は、MMI（Mixed Mode Interpreter）と呼ばれるインタプリタを持つ。MMI は、実行されたプログラム内のループを検出する機能や分岐命令の実行履歴を記録する機能を持つ。また、コンパイルされたネイティブコードとほぼ同

一のスタックフレームの構造をとることによって、コンパイルされたネイティブコードの実行とバイトコードの実行の遷移オーバヘッドが少なくしている、という特徴を持つ。この MMI が、バイトコードを読み込みながら実行を続け、メソッドの呼び出し回数とループの実行回数を元にして”hot-spots”部分を検出する。そして、”hot-spots”と判断されたメソッドを JIT コンパイラでコンパイルする。

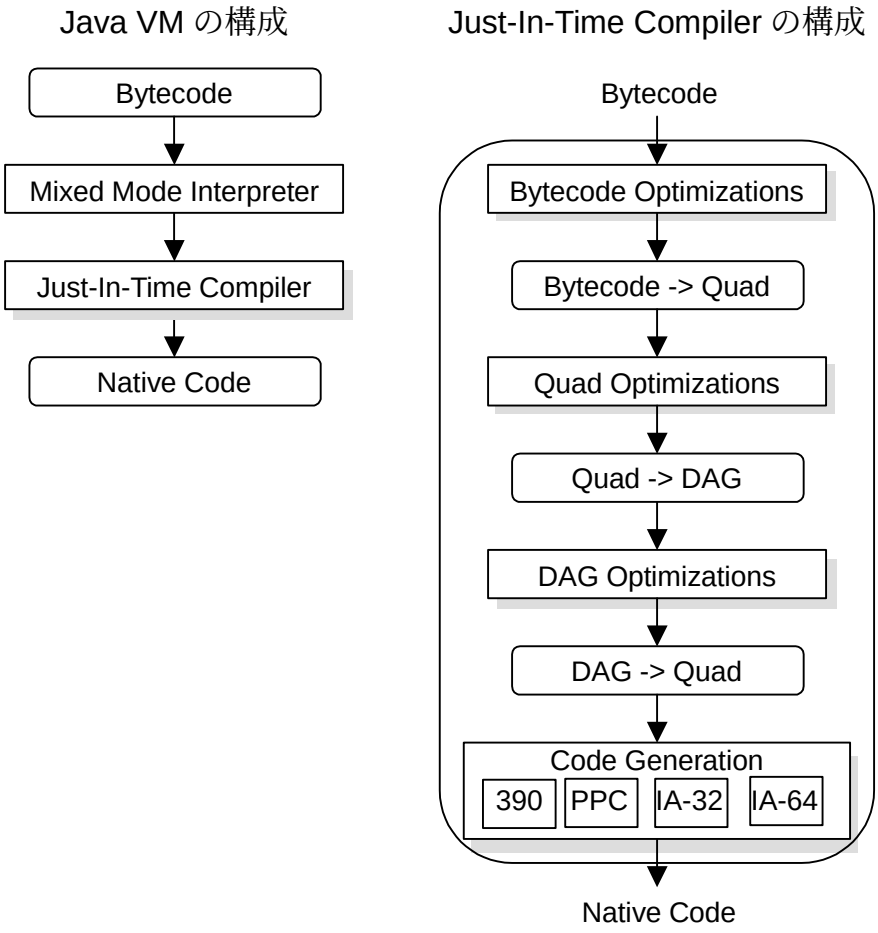


図 2.2: IBM Java JIT コンパイラの全体構成

JIT コンパイラは、Java バイトコードを入力として、内部表現に変換し最適化を行い、ネイティブコードを生成する。IBM Java JIT Compiler 4.0 は、IA-32[63]、IA-64[64]、32bit と 64bit の PowerPC[65]、32bit と 64bit の System 390[66]、と全く性格の異なる４つの CPU アーキテクチャに対応した最適化されたネイティブコードを生成することができる。

コンパイラの内部表現は、拡張バイトコード、４つ組、非循環有向グラフ（Directed Acyclic Graph、DAG）の３種類を持つ。４つ組は、命令コード、命令の２入力オペランド、命令の出力オペランド、をフィールドとして持つ構造体である。DAG は命令を節、命令間のデータ依存関係・制御依存関係・例外依存関係等を辺、として構成されるグラフである。それぞれの内部表現の段階で最適化が行われる。また、最終段階として、各プラットフォームに応じたレジスタ割り付けとネイティブコード生成が行われる。以下では、それぞれの段階の機能を述べる。

拡張バイトコード表現の段階における、最適化の流れを図 2.3に示す。拡張バイトコードは、

Java バイトコードにおいてオペランドやレジスタに対して型付けを行われていない部分を補い、完全に型付けしたものである。これにより、以降の様々な解析を容易にする。

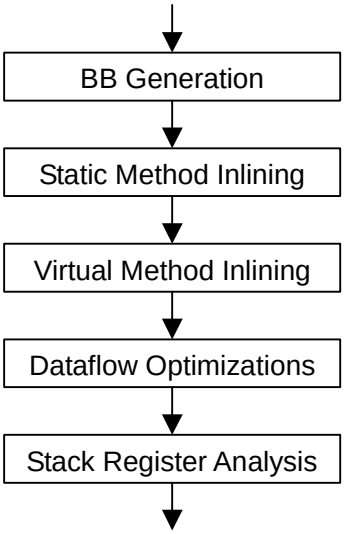


図 2.3: 拡張バイトコード表現での最適化

まず、Java バイトコードを完全に型付けされた拡張バイトコードに変換し、基本ブロック（Basic Block、BB）と呼ばれる、ブロックの入口と出口以外に制御の流れが存在しないブロックに分割する。拡張バイトコードにおける最も重要な最適化は、メソッド呼び出しをその位置で呼び出されるメソッド本体でそのまま置き換える、メソッド呼び出しのインライン展開である。これによって、後に続く最適化の適用範囲をメソッド間に拡大する。この段階でインライン展開を行う理由は、３種類の内部表現の中で拡張バイトコードによる表現が、最もメモリ消費量が少ないからである。そのため、この段階でコード量を最も増加させる最適化であるインライン展開を行う。インライン展開は、静的メソッド呼び出しに対するものと、仮想メソッド呼び出しに対するものの２種類がある。静的メソッド呼び出しは「static か private か final が宣言されているメソッド、またはコンストラクタメソッド」の呼び出しで、仮想メソッド呼び出しは「static も private も final も宣言されずコンストラクタでもないメソッド」の呼び出しである。静的メソッド呼び出しは、呼び出し先のメソッドを容易に一意に確定できるので、インライン展開可能である。仮想メソッド呼び出しは、実行時にクラス階層解析を適用して呼び出し先のメソッドを一意に確定できるか調べ、確定できた場合に仮想メソッド呼び出しの直接デバーチャル化を行ってメソッド呼び出しのインライン展開を行う。また、Java 言語では動的クラスローディングが行われるので、インライン展開を行ったメソッドが実行中のクラス階層においてオーバライドされた時には、直接デバーチャル化を用いてインライン展開したコードを無効化しなければならない。この手法[48]については、第 3 章で詳しく述べる。

スタックマシンモデルの拡張バイトコードからレジスタマシンモデルの４つ組表現へ単純に変換を行うと、内部表現のコード量が増加する。この増加を最小限にするために、後方データフロー解析を用いて実行されない命令や使用されていない定義を行っている命令を削除する不要コード除去や、前方データフロー解析を用いた Java 言語における例外検査のうち冗長な検査

の除去、等の最適化を行い、拡張バイトコードの量を削減する。その後、４つ組表現に変換するために必要なスタックの高さ解析を行い、拡張バイトコードから４つ組表現に変換する。

４つ組表現の段階における、最適化の流れを図 2.4に示す。４つ組表現の段階では、データフローを用いた最適化を主に行い、不要命令除去、命令変換、ループ外への命令移動、が行われる。

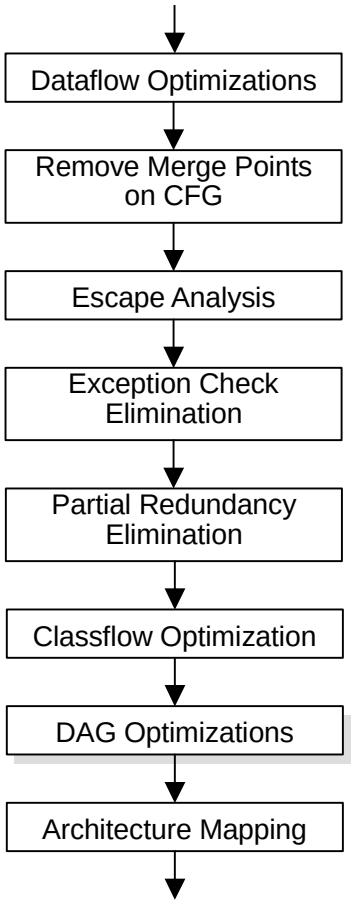


図 2.4: ４つ組表現での最適化

拡張バイトコードからの変換直後は、スタックマシンアーキテクチャを表現するための非効率的なコードになっている。従って、前方データフロー解析を用いて、単純な複写命令を削除する複写伝搬、定数式をコンパイル時に評価して式を定数で置き換える定数の畳み込み、等を適用し中間コードを単純化する。さらに、不要コード除去を適用しコード量を削減する。その後、後方データフロー解析を用いた最適化を効率的に適用するために、制御フロー上の合流点を取り除く最適化[67]を適用する。

次に、スレッド外や大域変数へアクセスすること無く、スレッド内に閉じて使用される配列やオブジェクトを検出する脱出解析[16][92]を行う。スレッド内で閉じたアクセスのみが行われると検出された配列要素やインスタンス変数の単純変数への割り付け、閉じたアクセスのみが行われるオブジェクトのスレッドスタック上への割り付け、閉じたアクセスのみが行われるオブジェクトに関するスレッド間の同期除去、を行う。さらに、Java 言語の正確な例外のセマンティクスを保持したまま、冗長な配列インデックス検査の除去、冗長な `null` ポインタ検査の除

去、プログラムの流れによっては冗長となる部分式を共通化する部分冗長除去（Partial Redundancy Elimination、PRE）[68][69][70]、を適用する[45]。また、２つ以上の式に現れる同一の部分式を１つにまとめる共通部分式除去も行う。さらに、PRE を用いた冗長な符号拡張命令の除去も行う[71]。その後、メソッド内の変数に流れるクラス情報を得るために前向きのデータフロー解析を用いて局所クラス解析を行う。この結果を用いて、仮想メソッド呼び出しの直接デバーチャル化の際に生成される仮想メソッド呼び出しを伴うバックアップパスの削除、冗長な型検査の除去、等の最適化を行う。さらに、メソッドの実行中に不変な仮想メソッド呼び出しのレシーバを検出する preexistence 解析[72]を用いたバックアップパスの削除を行う。その後、DAG 表現に変換し最適化を行い、再度４つ組表現に戻す。その後、IA-32 アーキテクチャにおける２オペランド表現など、対象アーキテクチャに適した４つ組表現への変換を行う。この際に高速な型検査への変形も行う[47]。

DAG 表現の段階における、最適化の流れを図 2.5に示す。DAG 表現の段階では、主にループに関する最適化が行われる。また、リストスケジューリングによる命令移動も行われる。

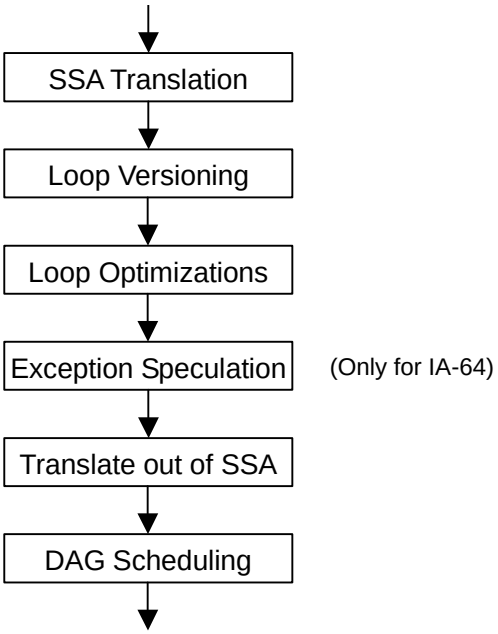


図 2.5: DAG 表現での最適化

まず、一変数に対して一回しか代入を許さない静的単一代入則（Static Single Assignment、SSA）[20]表現を用いた DAG 表現に変換する。その後、例外検査の実行回数を削減したループを作るループバージョニング[43][105]を適用する。さらに、従来知られている様々なループ最適化を行う。その後 IA-64 では、例外依存による命令間順序制約の緩和を行う[106][107]。その後、SSA 表現を通常表現に変換し[73]、最後に DAG 表現を用いたリストスケジューリング[74]を適用する。

ネイティブコード生成部の流れを、図 2.6に示す。ネイティブコード生成部ではコード変形は行わず、まずレジスタ割り付けを行う。レジスタ割り付けのアルゴリズムは、プラットフォーム毎のレジスタ数にあわせて最適なアルゴリズムを選択したために、各プラットフォームで異

なる[75][76]。その後、命令の生成順序を優先して並べる控えめなネイティブコードスケジューリングを適用しながら、ネイティブコードを生成する。

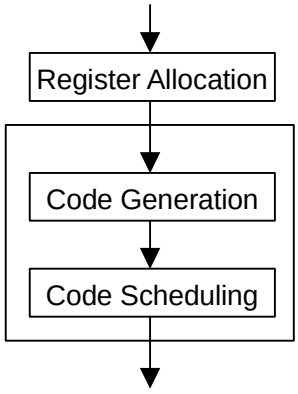
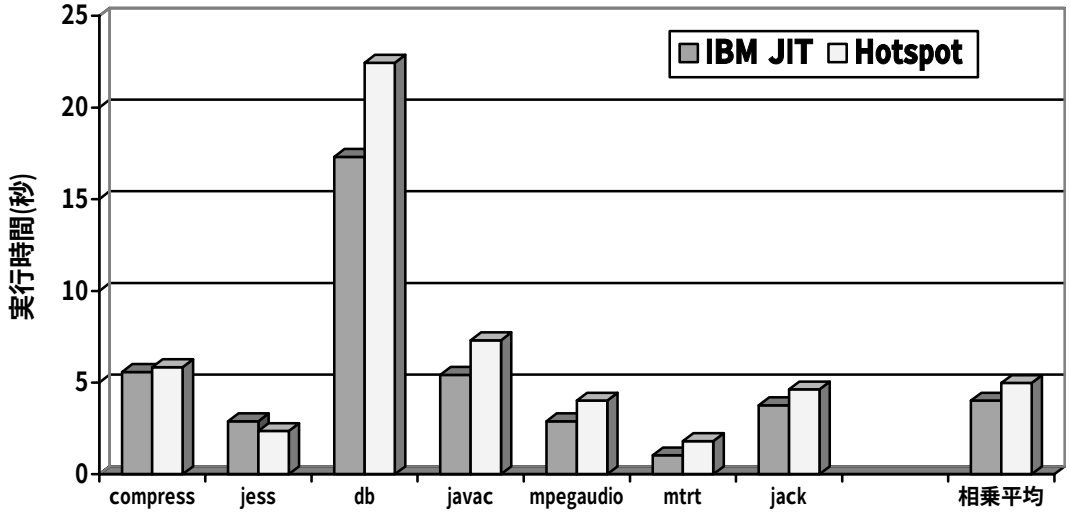


図 2.6: ネイティブコード生成部

最後に、IBM Java JIT コンパイラを含む IBM Developers Kit for Win32, Java 2 Technology Edition, Version 1.3.1 と、Java Hotspot Virtual Machine を含む Java 2 SDK, Standard Edition v1.3.1 の性能を、Win32 上で Xeon 2.2GHz Dual のマシンを使用して SPECjvm98 を用いて性能を比較した結果を図 2.7に示す。IBM Developers Kit は Java Hotspot Virtual Machine と比較して、SPECjvm98 において相乗平均で 2 割以上高い実行性能を示している。



それぞれの実行時オプション
IBM JIT: -noclassgc -ms512m -mx512m -Xquickstart
Hotspot: -noclassgc -server -Xms42m -Xmx42m -XX:NewSize=13m -XX:MaxNewSize=13m

図 2.7: IBM JIT と Hotspot の性能比較

第3章 仮想メソッド呼び出しの高速化

3.1 はじめに

Java 言語においてプログラミングの柔軟性を提供するために、仮想メソッド呼び出しは必要不可欠な機能である。仮想メソッド呼び出しとは、オブジェクト指向の特徴の 1 つである多態性を実現するための機能である。メソッド呼び出しにおける多態性とは、クラス階層全体において 1 つ以上のメソッドの実装をとることができるメソッド呼び出しが可能であることをいう。これは、メソッド呼び出し時にとるレシーバオブジェクトのクラスに属するメソッドを呼ぶことで、呼び出し先のメソッドの動的な変更を実現する。このためには動的束縛と呼ばれる、実行時に型を検査して適したメソッドを探索する動的な関連づけが必要となる。Java 言語では、（static も private も final も宣言されていない）インスタンスメソッドとインタフェースクラスのメソッド呼び出しを、仮想メソッド呼び出し[i)]を用いて実装することでメソッド呼び出しに関する多態性を実現している。以降、クラス階層におけるメソッド呼び出しに関する多態性を、単に多態性と呼ぶ。一方、static か private か final が宣言されているメソッドを呼び出すメソッド呼び出しが、静的メソッド呼び出しである。仮想メソッド呼び出しと、静的メソッド呼び出しのコード例を図 3.1に示す。静的メソッド呼び出しでは呼び出し先のメソッドが一意に決定しているので検索のオーバヘッドはないが、仮想メソッド呼び出しでは実行時に行われるメソッド探索がオーバヘッドになる。さらに、呼び出し先が一意に決定できないので、メソッド呼び出しのインライン展開を容易に適用することが出来ない。仮想メソッド呼び出しにおいて、探索オーバヘッドを削減するためにコンパイル時にメソッド検索を行い、呼び出し先のメソッドを一意化する手法はデバーチャル化（devirtualization）と呼ばれる。これまでに多くのデバーチャル化の手法[22][23][47][72][77][78][79][80][81][82][83]が提案されてきた。デバーチャル化は、その方法に基づいて、間接デバーチャル化、直接デバーチャル化、の２つに分けることができる。

```
r0 = <receiver object>
r1 = load(r0 + #offset-of-class-in-object)
r2 = load(r1 + #offset-of-method-in-class)
r3 = load(r2 + #offset-code-in-method)
call r3
```

a) 仮想メソッド呼び出しのコード例

```
call #native-code-address
```

b) 静的メソッド呼び出しのコード例

図 3.1: 仮想メソッド呼び出しと静的メソッド呼び出しのコード例

[i)] 仮想メソッド呼び出しは、Java バイトコードでは `invokevirtual`、 `invokeinterface` に相当する。静的メソッド呼び出しは、Java バイトコードでは `invokestatic`、 `invokespecial` に相当する。

間接デバーチャル化は、仮想メソッド呼び出しを行う際にレシーバのクラスの型テストを行い、このテストが成功したとき呼び出し先メソッドの検索無しにメソッド呼び出しを実行する[77]。Self[79]等の動的な型付けを持つ言語では、呼び出し先メソッドの検索のオーバヘッドが非常に大きいので、この方法は有効である。しかし、Java 言語等の静的な型付けを持つ言語では、呼び出し先メソッドの検索は図 3.1 a)に示される数個のロード命令と間接分岐で実現されるため、メソッド呼び出しのオーバヘッドが比較的小さい。このため間接デバーチャル化の効果が小さいことが知られている[60]。

直接デバーチャル化は、仮想メソッド呼び出しを行う際にレシーバのクラスの型テストを行わず、呼び出し先メソッドの検索無しにメソッド呼び出しを実行する。つまり、静的メソッドと同じコストでメソッド呼び出しを実行できる。従って、静的な型付けを持つ言語でも大きな効果を得られる。さらにメソッド呼び出しのインライン展開の適用範囲を拡大することができる。この結果、コンパイラの最適化の適用範囲も拡大される。これまで提案された直接デバーチャル化の多くは、プログラム実行前にクラス階層解析[22][78][80][81][82][83]を行うため、C++ 言語や Module-3 言語等の動的クラスローディングを行わない言語に適用されていた。Java 言語等の動的クラスローディングを行う言語では、実行時にクラス階層が変更される。このような言語で直接デバーチャル化を適用するためには、実行時もクラス階層解析を行い、動的クラスローディングによってメソッドのオーバライドが発生して直接デバーチャル化を適用した際のクラス階層の仮定が成立しなくなったときに、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[41]が提案されているが、従来この方法は複雑な実装を必要としている。

本章では、Java 言語等の動的クラスローディングを行う言語において、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[48]を提案する。この方式によって実行時にクラス階層が変化する Java 言語においても容易に直接束縛が可能になる。この結果、仮想メソッド呼び出しに対してメソッド呼び出しのインライン展開を適用し、より広範囲に最適化を適用することが可能になる。本手法は、他の Java Just-In-Time コンパイラ[34][55]でも実装されている。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元の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。最初は直接デバーチャル化されたコードを実行する。実行中に、動的クラスローディングによって仮想メソッド呼び出しが複数のメソッドを呼び出し先として持ったとき、仮想メソッド呼び出しのコードを実行するために直接デバーチャル化されたコードの先頭命令を書き換える。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だけであるので、脱最適化による再コンパイルで要求される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ため、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、制御フロー上に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。また、本手法と局

所クラス解析と preexistence 解析[72]を組み合わせて Java Just-In-Time コンパイラに実装し評価を示す。その結果、デバーチャル化を行わない場合に比べ、仮想メソッドの呼び出し回数に関して SPECjvm98、SPECjbb2000、エージェントベンチマークにおいて 34.0%～98.6%（平均 69.8%）減少できることを、性能に関して SPECjvm98、SPECjbb2000、エージェントベンチマークにおいて相乗平均で PowerPC では 18.3%、IA-32 では 17.9%、IA-64 では 14.7%の性能向上が得られることを示す。

以下、3.2節で関連研究を述べ、3.3節で仮想メソッド呼び出しのデバーチャル化手法について述べる。3.5節では、プログラムを用いて本方式を評価した結果を示す。3.6節で本章のまとめを述べる。

3.2 関連研究

本節では、従来提案されたデバーチャル化手法を分類して示す。仮想メソッド呼び出しのデバーチャル化は、その方法に基づいて、レシーバのクラスの型テストを必要とする間接デバーチャル化と、テストを必要としない直接デバーチャル化、の２つに分けられる。

3.2.1 間接デバーチャル化

間接デバーチャル化は、実行時に仮想メソッド呼び出しのレシーバに関するテストを行って呼び出し先のメソッドを一意化する手法である。間接デバーチャル化では、コンパイル時に仮想メソッド呼び出しにおいて呼び出し頻度の高いメソッドを選び、実行時にテストを行う候補とする。仮想メソッド呼び出しを実行する際に、コンパイル時に選んだメソッドが属するクラスとレシーバのクラスの型テストを行う。このテストが成功した場合、呼び出し先メソッドの検索無しに高速にメソッド呼び出しを実行できる。

クラステスト[77]は、インライン展開されたメソッドのクラスとレシーバのクラスについて型テストを行い、一致した場合にはインライン展開されたコードを実行する。この場合、呼び出し先のメソッドを高速に実行できる。テストが失敗した場合には、呼び出し先メソッドを検索後メソッド呼び出しを行う。メソッドテスト[72]は、インライン展開されたメソッドとレシーバのクラス内のメソッドについて型テストを行い、一致した場合にインライン展開されたコードを実行する。テストが失敗した場合は、呼び出し先メソッドを検索後メソッド呼び出しを行う。

さらに、呼び出し頻度の高いメソッドを実行時に決定する方法としてインラインキャッシュがある。この手法では、実行時に最後の呼び出しが行われた際のレシーバオブジェクトのクラスを呼び出し元で記録し、そのクラスに属するメソッドが持つスタブへ直接飛ぶコードを生成する。スタブでは、レシーバのクラスが記録されたクラスと異なるならば、通常のメソッド探索を行ってメソッド呼び出しを行い、レシーバオブジェクトのクラスを呼び出し元に記録する。この手法を多態的な呼び出しの場合にも適用できる手法も提案されている[88]。

間接デバーチャル化は、キャッシュに基づいた高速化手法であるので、メモリ参照を伴う実

行時テストが新たに必要となる。簡単な実験[89]によると、メソッドのインライン展開を行わない間接デバーチャル化では、インライン展開を行わない直接デバーチャル化の場合の性能を越えることはできない。また、インライン展開を行った間接デバーチャル化でキャッシュのヒット率が 90%程度ないと、インライン展開を行わない直接デバーチャル化の場合の性能を越えることはできない。さらに、直接デバーチャル化でインライン展開を行った場合、間接デバーチャル化ではその性能を越えることはできない。

実際に Java 仮想マシンにインラインキャッシュと多態的なインラインキャッシュを実装した場合、動的な型付けを持つ言語の場合ほどの性能向上は得られないことが知られている[60]。Java 言語等の静的な型付けを持つ言語では、仮想メソッド呼び出しの実装のコストは、キャッシュに基づいた間接デバーチャル化の実装のコストと同等である。従って、インライン展開を伴った間接デバーチャル化は、メソッド間に渡ってコンパイラの最適化の範囲を拡大するために有効である。

3.2.2 直接デバーチャル化

間接デバーチャル化は、実行時に仮想メソッド呼び出しのレシーバに関するテスト無しに呼び出し先のメソッドを一意化する手法である。直接デバーチャル化では、まず仮想メソッド呼び出しのレシーバが取り得るクラス集合において、コンパイル時に呼び出しメソッドを一意に決定可能かどうか、クラス階層解析等の解析手法を用いて調べる。もし一意に決定できるならば、静的メソッド呼び出しに変換して、またはインライン展開したメソッドを、レシーバに関する型テスト無しに実行する。

直接デバーチャル化は、再コンパイルを必要とする方式と、必要としない方式に分けることができる。以下、それぞれの方式を示す。

3.2.2.1 再コンパイルを必要としないデバーチャル化

局所クラス解析[84][85][86][87]は、データフロー解析を用いて仮想メソッド呼び出しのレシーバの型を絞り込む。この結果、レシーバが取るクラスが唯一であることを示せば、仮想メソッド呼び出しを静的メソッド呼び出しに変換できる。さらに、静的メソッド呼び出しのコードのみを生成すればよいので、本章で提案する方式と異なり制御フロー上に合流点を生成しない。この変換は動的クラスローディングを行う言語でも常に成り立ち、再コンパイル不要である。

仮想メソッド呼び出しが取り得るクラス集合内で呼び出しメソッドがオーバライドされていなければ、メソッドを一意に決定できるので、直接デバーチャル化可能である。この判定には、プログラム全体のクラス階層においてどのメソッドが定義されているかを調べる、クラス階層解析[14] が必要である。この解析をプログラムのリンク時に行う手法[15] や、より正確な階層間の解析をコンパイル時に行う手法が提案されている[81][82][83]。静的クラスローディングによってクラス階層がプログラム実行前に決定し実行時に変化しない C++言語等では、コンパイ

ル時に直接デバーチャル化を適用した際の解析結果は不変であり、再コンパイルは不要である。

C++、Dylan、Java 言語では、ユーザの明示的な宣言によって、あるクラスの継承を不可能にする機能を持つ。これによって、あるクラスが実行時にオーバーロードされないことが保証されるので、そのクラスに属するメソッドを直接デバーチャル化可能である。しかし、Java API の中では、この機能が用いられることは少ない。なぜならば、クラスの継承を禁止することで、プログラムの再利用性を低下させるからである。例えば、java.util.Vector クラスの多くのメソッドでは、Java で行われていたクラスの継承を禁止する宣言は、Java 2 では取り去られている。

3.2.2.2 再コンパイルを必要とするデバーチャル化

動的クラスローディングによってプログラム実行中にクラス階層が変化する Java 言語では、コンパイル時に直接デバーチャル化を適用した際のクラス階層内でメソッドのオーバライドがない、という仮定が実行時に無効になることがある。以下に述べる２つの方式では、 デバーチャル化されたコードのみをコンパイル時に生成するので、メソッドがオーバライドされた際に生成されたコードを無効化する再コンパイルが必要である。

脱最適化[41]は、直接デバーチャル化を適用した際のコンパイル時の仮定が、メソッド実行中のクラス階層の変化によって無効になった時、コンパイル時に設定した安全点に到達するまでコードが実行される。この地点で、現在実行中の最適化されたコードのためのスタックフレームを最適化前のコードに対応するスタックフレームに変換する。この後、直接デバーチャル化される前のコードを実行し、多態的な仮想メソッド呼び出しを正しく実行する。この手法で用いられるスタックフレームの変換は、最適化されたコードから最適化される前の状態へ戻すために多くの情報を実行時まで持つことを要求する。さらに、インライン展開された 1 つのメソッドのスタックフレームをインライン展開前の複数メソッドのスタックフレームに対応づけて再構成した後実行を再開する、等実装が非常に複雑である。さらに、マルチスレッドの実行環境下でコードを破棄することも容易ではない。

クラス階層の変更がメソッド実行中に発生し、直接デバーチャル化をコンパイル時に適用した際のクラス階層の仮定が無効になった場合でも、仮想メソッド呼び出しのレシーバに到達するオブジェクトが実行中のメソッドの呼び出し前に生成されていたならば、クラス階層の変化に影響されずに直接デバーチャル化されたコードを実行できる。この性質を preexistence[72]と呼ぶ。Preexistence が成り立つことは、仮想メソッド呼び出しのレシーバがメソッドの実行中に変化しない場合を検出することで示される。この方式では、直接デバーチャル化したコードを無効化するメソッドの再コンパイルは次回のメソッド呼び出し時に行えばよい。従って、脱最適化で要求される複雑な実装を必要としない。

3.3 命令書き換えによる直接デバーチャル化

本節では、動的クラスローディングを行う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[48]について述べる。3.3.1節ではコード書き換えを用いたデバーチャル化の方法を示し、3.3.2節では各プロセッサにおけるコードの書き換え方法を示す。

3.3.3節では、制御フロー上に合流点を生成するが、メソッドのオーバライドが起きない場合のコードの質を、各種の最適化によって制御フロー上に合流点を持たない場合とほぼ同等にできることを示す。3.3.4節では、脱出解析の適用方法について述べる。

3.3.1 コード書き換えを用いた直接デバーチャル化

本節では、動的クラスローディングを行う言語において仮想メソッド呼び出しの直接デバーチャル化を行う方法を示す。以下では、Java 言語の仮想メソッド呼び出しとインタフェースメソッド呼び出しについて述べる。

コンパイル時のクラス階層解析によって仮想メソッド呼び出しの呼び出し先メソッドを一意に決定できると判断したとき、コンパイラはレシーバのクラスの型テスト無しに、メソッドをインライン展開するか、静的メソッド呼び出しを実行するコードを生成する。同時に、呼び出し先メソッドの検索を行う仮想メソッド呼び出しのコードも生成する。このとき、テスト無しに実行できる命令列の先頭アドレスを記録する。

直接デバーチャル化を適用後にインライン展開を行った場合のコード例を、PowerPC の命令セットを用いて、図 3.2に示す。まず、メソッドがオーバライドされていないときは、図 3.2の左の直接デバーチャル化されてインライン展開されたメソッドのコードが実行され、仮想メソッド呼び出しの*斜体のコード*は実行されない。動的クラスローディングによってクラス階層内でメソッドのオーバライドが起きて、仮想メソッド呼び出しが複数のメソッドを呼び出し先として持った場合、コンパイル時に記録されたテスト無しに実行できる命令列の先頭アドレスの命令を、仮想メソッド呼び出しを実行するために分岐命令である b 命令で書き換える。この結果が、図 3.2の右のコードであり、インライン展開されたコードはもはや実行されない。このコード書き換えは実行中の任意の時点で適用可能であるので、仮想メソッド呼び出しのレシーバが取るオブジェクトのクラス階層が実行中の任意の時点で変更されても正しく実行されることを保証する。従って、本方式はコンパイル時にオーバライドされていない全ての仮想メソッド呼び出しに対して適用可能である。

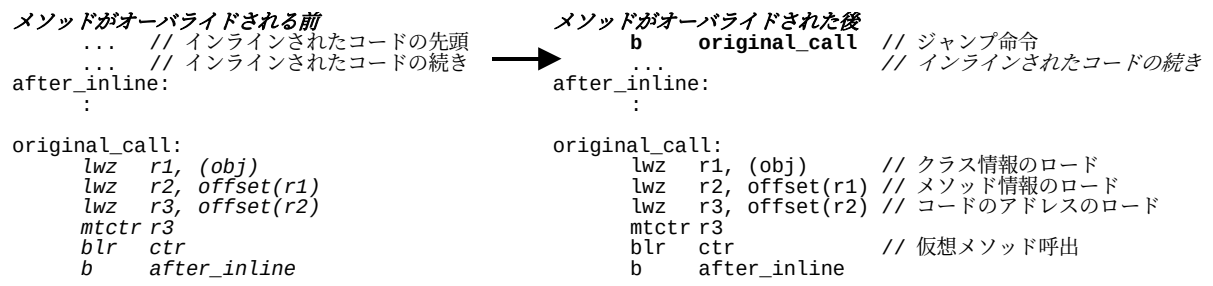


図 3.2: 直接デバーチャル化された仮想メソッド呼び出しのコード

Java 言語では、多重継承を実現するためにインタフェースクラスを用意している。インタフェースメソッド呼び出しも、クラス階層解析を用いて最適化できる。クラス階層解析によって、

インタフェースクラスを実装するクラスが１つだけであることが分かれば、直接デバーチャル化を適用してインタフェースメソッド呼び出しから実装しているクラスについての仮想メソッド呼び出しに変換できる。さらに、インタフェースクラスを実装しているクラスのサブクラスにおいてメソッドがオーバライドされていないならば、変換された仮想メソッド呼び出しを直接デバーチャル化可能である。一般にインタフェースメソッド呼び出しでは、ループを実行して呼び出し先のメソッド検索を行うのでオーバヘッドが大きい。従って、この最適化は非常に有効である。この時、コンパイラは、以下の 3 種類のコードを生成する。

1. 呼び出し先が一意に決定されたメソッドを直接デバーチャル化したコード
2. 仮想メソッド呼び出しのコード
3. インタフェースメソッド呼び出しのコード

直接デバーチャル化を適用後にインライン展開を行った場合のコード例を、図 3.3に示す。まず、インタフェースクラスを実装するクラスが１つで、メソッドがオーバライドされていないときは、図 3.3の左の直接デバーチャル化によってインライン展開されたコードが実行され、*斜体のコード*は実行されない。もし、動的クラスローディングによってインタフェースクラスを実装しているクラス階層内でメソッドのオーバライドが起きた場合、仮想メソッド呼び出しを実行するためにインライン展開されたコードの先頭の命令が b 命令で書き換えられて、インライン展開されたコードは実行されない。また、動的クラスローディングによってインタフェースクラスを実装するクラスが２つ以上になった場合、元のインタフェース呼び出しを実行するためにインライン展開されたコードの先頭命令が b 命令で書き換えられる。この結果、インライン展開されたコードも、仮想メソッド呼び出しのコードも実行されず、図 3.3の右のコードによってインタフェース呼び出しが実行される。

１つのクラスで実装しているとき

```
    ...   // インラインされたコードの先頭
    ...   // インラインされたコードの続き
after_inline:
    :

virtual_call
    lwz   r1, (obj)
    lwz   r2, offset(r1)
    lwz   r3, offset(r2)
    mtctr r3
    blr   ctr
    b     after_inline
interface_call:
    mr    r1, obj
    blr   rt_interface
    b     after_inline
```

２つ以上のクラスで実装しているとき

```
      b     interface_call // ジャンプ命令
      ...                  // インラインされたコードの続き
after_inline:
      :

virtual_call
      lwz   r1, (obj)      // クラス情報のロード
      lwz   r2, offset(r1) // メソッド情報のロード
      lwz   r3, offset(r2) // コードのアドレスのロード
      mtctr r3
      blr   ctr
      b     after_inline
interface_call:
      mr    r1, obj        // レシーバのセット
      blr   rt_interface   // インタフェースメソッド呼出の実行
      b     after_inline
```

図 3.3: 直接デバーチャル化されたインタフェースメソッド呼び出しのコード

本方式で用いられる実行時の命令書き換えの実装を、以下に示す。コンパイル時に、ある仮想メソッド呼び出しが複数の呼び出し先メソッドを持つかどうか調べる際に、もし対象のメソッド呼び出しの静的なメソッド宣言に対しての初めての問い合わせであったならば、キャッシュ構造体を生成しその結果を格納してから問い合わせに対する結果を返す。さらに、このキャッシュ構造体にメソッドがオーバライドされた時に命令を書き換えるネイティブコードのアドレスを記録する。２度目以降の問い合わせならば、キャッシュ構造体の結果を返し、さらにキ

ャッシュ構造体にメソッドがオーバライドされた時に命令を書き換えるアドレスを追加記録する。

新しいクラスが実行時に動的クラスローディングされたときにメソッドがオーバライドされたならば、実行時ルーチンはキャッシュ構造体から直接デバーチャル化されたメソッドの一覧を調べ、一致したものが有ったならば、キャッシュ構造体の結果を複数のメソッドが実装されたとして書き換え、そこに記録されているネイティブコードのアドレスの命令を書き換えて直接デバーチャル化されたコードを無効にする。さらに、ロードされたクラスがインタフェースクラスを実装する時に、インタフェースクラスを実装するクラスの数が２つ以上になった場合、キャッシュ構造体に記録されているネイティブコードのアドレスの命令を書き換えて直接デバーチャル化されたコードを無効にする。

このコード書き換えを用いたデバーチャル化の実装の容易さは、Jikes RVM[34]や JUDO[55]でも本手法が採用されていることからも実証される。

3.3.2 各アーキテクチャにおける命令書き換えの実装方法

本方式では、生成されたコードの一部を無効化するために実行時に命令を書き換える必要がある。Java 言語ではマルチスレッドがサポートされているので、ある命令列が複数のスレッドで同時に実行される可能性がある。従って、安全にコードを無効化するためには、命令の書き換えはアトミックに行われなければならない。以下では、PowerPC、IA-64、IA-32、System 390 と様々なアーキテクチャで、命令をアトミックに書き換える方法を示す。

3.3.2.1 PowerPC

PowerPC アーキテクチャ[65]は、１ワード（32 ビット）の固定長の命令を持つ RISC アーキテクチャである。メモリシステムはワード単位の読み書きがアトミックな操作であることを保証しているので、命令のアトミックな書き換えはワード書き込みで新しい分岐命令を書き込むことで実現できる。

PowerPC アーキテクチャでは、命令キャッシュとデータキャッシュが分離していて、２つのキャッシュ間で自動的に同期が取られることがない。キャッシュ間の同期を取るためには明示的に両方のキャッシュの内容を破棄する命令を実行し、新たにメモリからデータを読み込む必要がある。さらに、書き換え対象の命令がすでにプロセッサ内の命令キューに投入されている可能性もあるので、明示的にプロセッサ内の命令キューの内容も破棄しなければならない[90]。これらの処理は、図 3.4に示す命令列を実行することで行われる。

```
stw   r4, 0(r3) // r3 のアドレスにある命令を、r4 の命令で書き換える。
dcbst 0, r3     // データキャッシュの内容でメモリを更新する。
sync            // メモリの更新を待つ。
icbi  0, r3     // 命令キャッシュのコピーを無効化する。
isync           // この命令が実行された時点のコンテクストでフェッチと実行を行う
                // コンテクスト同期化を実行する。
```

図 3.4: PowerPC アーキテクチャにおける命令書き換えの例

以上の処理を行うことによって、PowerPC アーキテクチャにおいてアトミックに安全に命令を書き換え可能である。

3.3.2.2 IA-64

IA-64 アーキテクチャ[64]は、同時実行可能な３命令がアライメントされた 128 ビット（16 バイト）のバンドルに含まれる VLIW アーキテクチャである。メモリシステムは、アライメントされたメモリへの１命令による読み書きはアトミックな操作であることを保証している[91]。１命令で読み書きできる大きさは、１バイト・２バイト・４バイト・８バイトである。

バンドルの形式を図 3.5に示す。バンドルは 16 バイトアライメントされているので、アトミックに読み書きできるのは、命令スロット０か命令スロット２に限られる。実行時に、命令スロット０か命令スロット２へ新しい分岐命令の書き込みを１命令で行うことで、アトミックに命令を書き換えることができる。ただし、各命令スロットに配置できる命令の種類はテンプレートによって決定される。８バイト（64 ビット）のメモリ書き込みを用いても、命令スロット２とテンプレートを同時に書き換えることは出来ない。従って、命令スロット０の命令を書き換える場合には命令の種類を変更することが出来るが、命令スロット２の命令を書き換える場合には命令の種類を変更することは出来ない。

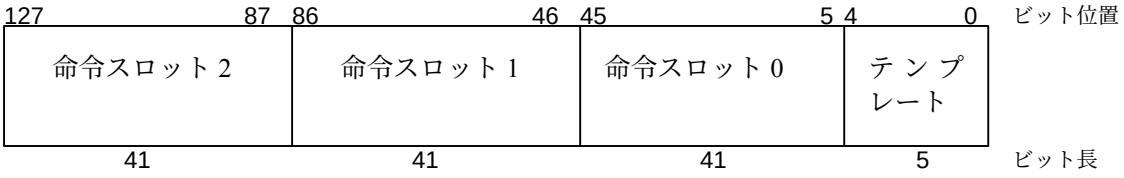


図 3.5: バンドルの形式

IA-64 アーキテクチャでは、命令キャッシュとデータキャッシュが分離していて、２つのキャッシュ間で自動的に同期が取られることがない。キャッシュ間の同期を取るためには明示的に両方のキャッシュの内容を破棄する命令を実行し、新たにメモリからデータを読み込む必要がある。さらに、書き換え対象の命令がすでにプロセッサ内の命令キューに投入されている可能性もあるので、明示的にプロセッサ内の命令キューの内容も破棄しなければならない。これらの処理は、図 3.6に示す命令列を実行することで行われる。

```
st [r32] = r33   // r32 のアドレスにある命令を、r33 の命令で書き換える。
fc r32           // 命令・データキャッシュの内容でメモリを更新する。
sync.i           // メモリの更新を待つ。
srlz.i           // 命令キューを再フェッチする。
```

図 3.6: IA-64 アーキテクチャにおける命令書き換えの例

以上の処理を行うことによって、IA-64 アーキテクチャにおいてアトミックに安全に命令を書き換え可能である。

3.3.2.3 IA-32

IA-32 アーキテクチャ[ii][63]は、可変長の命令を持つ CISC アーキテクチャである。メモリシステムは、同一キャッシュライン内での１命令による読み書きはアトミックな操作であることを保証している[91]。１命令で読み書きできる大きさは、１バイト・２バイト・４バイトである。一方、無条件分岐命令は、２バイト長と５バイト長の命令がある。

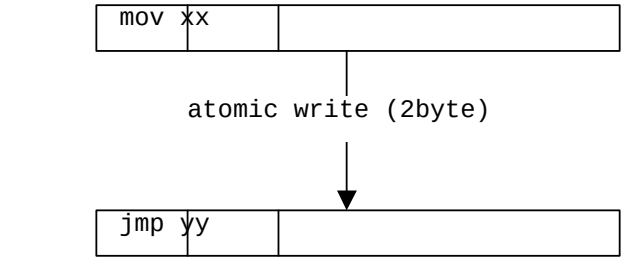
新しく書き込む分岐命令の長さが２バイトである場合、コンパイラは書き換えられる先の命令が２バイト以上であることと、この書き換え先の命令列が同一キャッシュライン内に配置されること、を保証して命令を生成する。この命令に対して、実行時に新しい分岐命令の書き込みを１命令で行うことで、アトミックに命令を書き換えることができる。書き換え処理の流れを、図 3.7 a)に示す。

新しく書き込む分岐命令の長さが５バイトである場合、コンパイラは書き換えられる先の命令が５バイト以上であることと、この書き換え先の命令列が同一キャッシュライン内に配置されることを保証して命令を生成する。実行時には、まず書き換え先の命令列の先頭で自己ループさせるために、先頭２バイトを１命令でアトミックに書き換える。その後、残りの３バイトを新しい分岐命令の一部で非アトミックに書き換える。最後に、先頭２バイトを１命令でアトミックに書き換えることで、５バイトの分岐命令を安全に書き換えることができる。書き換え処理の流れを、図 3.7 b)に示す。

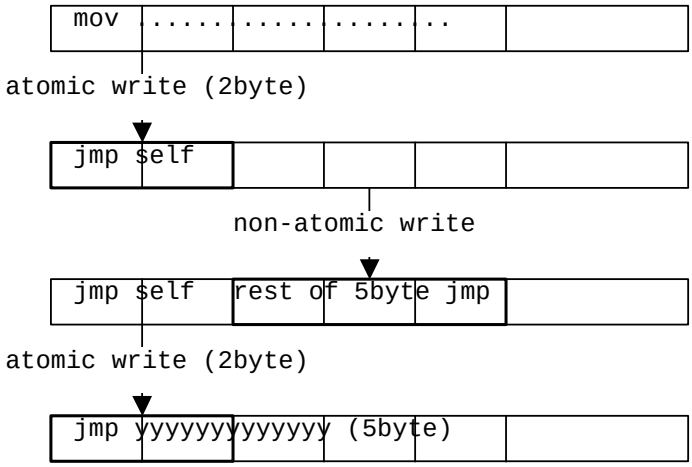
IA-32 アーキテクチャではメモリへの書き込み命令は、データキャッシュと命令キャッシュ間の同期を自動的に取り、必要ならば命令キューの内容も破棄するので、これらの同期に関する考慮は必要ない。

以上の処理を行うことによって、IA-32 アーキテクチャにおいてアトミックに安全に命令を書き換え可能である。

[ii] 本節では、NetBurst マイクロアーキテクチャを対象とする。



a) 2 バイト命令で書き換える例



b) 5 バイト命令で書き換える例

図 3.7: IA-32 アーキテクチャにおける命令書き換えの例

3.3.2.4 System 390

System 390 アーキテクチャ[66]は、可変長の命令を持つ CISC アーキテクチャである。メモリシステムは、キャッシュライン内での１命令による読み書きはアトミックな操作であることを保証している[91]。１命令で読み書きできる大きさは、１バイト・２バイト・４バイトである。一方、無条件分岐命令は、４バイト長の命令である。

コンパイラは、新しく書き込む分岐命令の書き換え先に生成する命令が４バイト以上であることと、この書き換え先の命令列が同一キャッシュライン内に配置されること、を保証して命令を生成する。この命令列に対して、実行時に新しい分岐命令の書き込みを１命令で行うことで、アトミックに命令を書き換えることができる。

また、System 390 アーキテクチャではメモリへの書き込み命令は、データキャッシュと命令キャッシュ間の同期を自動的に取り、必要ならば命令キューの内容も破棄するので、これらの同期に関する考慮は必要ない。

以上の処理を行うことによって、System 390 アーキテクチャにおいてアトミックに安全に命令を書き換え可能である。

3.3.3 制御フローの合流点を持つコードの最適化

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し、あらかじめ用意された仮想メソッド呼び出しのコードを実行する。従って、脱最適化に比べ実装が容易である。なぜなら、実行中の最適化されたスタックフレームの変換を伴うメソッドの再コンパイルという脱最適化の実装に必要な複雑な機構が不要なためである。一方、再コンパイルを不要にするために、直接デバーチャル化されたコードと仮想メソッド呼び出しを含むコード、と２種類以上のコードをコンパイル時に用意するので、制御フロー上に合流点が生成される。一般に制御フロー上の合流点では、データフロー解析を行う場合１つの変数に関する複数情報が合流するため、データフロー解析を用いた最適化が妨げられる可能性が高い。さらに、本手法が生成する仮想メソッド呼び出しのコードを含むパスでは呼び出し先のメソッドが不定であるため副作用に影響される全ての変数についてデータフローの情報を不定とする。この結果、制御フローの合流点においてスカラへの置き換え、コード移動、といった後ろ向きデータフロー解析を用いた最適化が制限される。また、メソッド呼び出しの結果で得られる型情報は（特にテンプレートを持たない Java 言語では）非特定的であることが多いので、合流点で型情報が失われやすい。この結果、局所クラス解析の結果が不定になる。

前述のコードに対して、フロー合流点の除去、例外などの副作用の除去、を行う以下の最適化を適用し、フロー解析における情報を一意にしデータフロー解析の精度を改善する。

1. 型情報に関する制御フローの合流点の除去
2. ループ皮むき（loop peeling）
3. 後方移動による `null` ポインタ検査の除去
4. 実行確率情報を利用した補償コードの生成

この結果、スカラへの置き換え等、他の最適化の適用機会を増やすことができる。以下、図 3.8 a)のプログラムを用いて、制御フローの合流点が存在しても最適化可能な手法を示す。まず、本方式による直接デバーチャル化を S1 に適用する。その結果が図 3.8 b)である。基本ブロック（Basic Block、BB） 2 は、コード書き換え点にあたる仮想的な制御フローの分岐点で、実際にコードは生成されない。BB 3 と BB 4 の合流点で c に関するクラスの型情報が失われている。本方式を S2 に適用後、分岐の偏りを考慮したコードの複製[67]を適用して、BB 3 と BB4 のフローの合流点を BB6 の下へ移動する。この結果、BB3→5→6 は c に関して特定された型情報が到達する。その結果が、図 3.8 c)である。BB5 は、コード書き換え点にあたる仮想的な制御フローの分岐点で、実際にコードは生成されない。

次に、インスタンス変数 `a.x`、 `b.y` の参照に対してスカラへの置き換えを適用したい。そのため `a` と `b` に関する `null` ポインタ検査を除去したいが、Java 言語では例外発生の順序は正確に守られなければならないので、単に `nullcheck a` や `nullcheck b` をループ外に移動する最適化は適用できない。`a`、 `b` はループ内不変変数であるので、ループ皮むきを適用して BB2～8 の複製である BB2’～8’をループ前に生成する。その後、前方到達による `null` ポインタ検

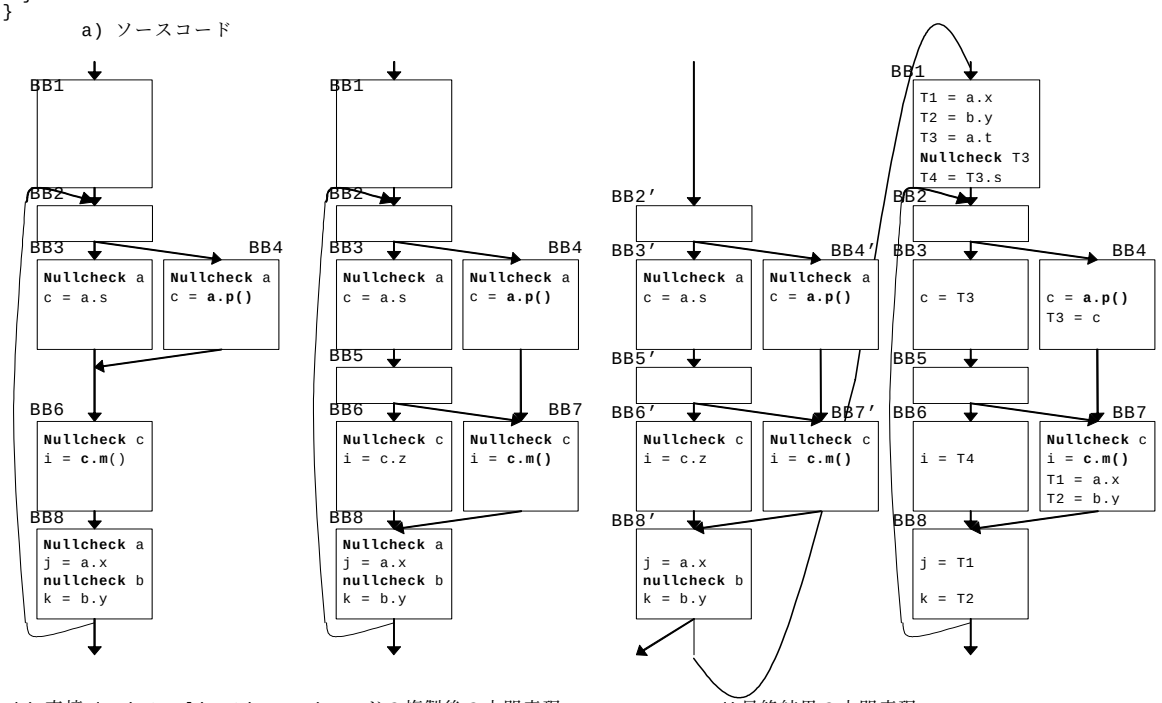
査除去を適用する。これによって、ループ内に存在する a、 b に関する null ポインタ検査を除去できる。この結果、a.x、 b.y の参照に関してループ外にコードを生成可能になるので、これらの参照に対してスカラへの置き換えを適用できる。

最後に、c.z の参照に対してスカラへの置き換えを適用したい。そのため、後方移動による null ポインタ検査除去[45]を用いて nullcheck c をループ外に移動する。さらに、BB 7 はメソッドのオーバライドが起きたときにだけ実行されるため実行確率が低いと考えられる。従って、部分冗長性除去を用いて BB 7 の仮想メソッド呼び出し c.m()の直後に補償コードを生成して、スカラへの置換えを適用する。最終結果が、図 3.8 d)である。

この例では、メソッドのオーバライドが起きないときに実行される BB 2→3→5→6→8 のコードは、制御フロー上に合流点が存在しないコードとほぼ同等である。

```
Class Bar extends Foo {
  int m() { return this.z;}
}
class Foo {
  Bar s;
  int x, y, z;
  Foo p() { return this.s;}
  int m() { return this.z+1;}
  int caller(Foo a, Foo b) {
    Foo c;
    do {
S1:   c = a.p();// BB3 and 4
S2:   i = c.m();// BB6 and 7
S3:   j = a.x;  // BB8
S4:   k = b.y;  // BB8
    } while (cond)
    ...
  }
}
```

a) ソースコード



b) 直接 devirtualization 後の中間表現

c)コードの複製後の中間表現

d)最終結果の中間表現

図 3.8: 最適化の適用例

3.3.4 脱出解析の適用

脱出解析の Java 言語への適用[90]が提案されている。脱出解析では以下の条件が満たされる

ならば、スレッド内で参照されるオブジェクトは、その生存区間がメソッド内部かつスレッド内で閉じているので、捕捉されていると判断する。

1. 変数が単一代入である。
2. 変数の定義が new 式によって行われる。
3. 変数の全ての参照が、仮想メソッド呼び出しのレシーバや引数となる、等による脱出を起こさない。

この判断の結果、捕捉されているオブジェクトの各フィールドの定義・参照に対してスカラへの置換えを適用可能である。この結果、オブジェクトをヒープに確保するオーバヘッドが削減される、変換後のプログラムに対して他の最適化を行う機会が増す、等の利点が得られる。

仮想メソッド呼び出しのレシーバや引数によって変数の参照が起きた場合は、前記の条件 3. によって、その変数から参照されるオブジェクトは捕捉されていない、と通常は判断される。しかし、以下のアルゴリズムで補償コードを生成することにより、仮想メソッド呼び出しから脱出しているオブジェクトも捕捉されている、と扱うことが可能である。

1. new 式があった場所に、オブジェクトをヒープに確保したかを表す論理型変数を定義し、偽の値で初期化する。
2. 仮想メソッド呼び出しの前で、以下の 2 つの処理を行う。
   a) 1.で定義した論理型変数が偽ならば、new 式でヒープにオブジェクトを確保し、論理型変数に真の値を代入する。
   b) スカラに置き換えられたフィールド内の値をヒープ上のオブジェクトのフィールドへ代入する。
3. 仮想メソッド呼び出しの後ろで、ヒープ上のオブジェクト内のフィールドの値をスカラに置き換えられたフィールドへ代入する。
4. return 文、同一メソッド内で捉えられない例外、等メソッドから脱出する直前で、1. で定義した論理型変数が真ならばスカラに置き換えられたフィールド内の値をヒープ上のオブジェクト内のフィールドへ代入する。

通常、プログラム中に元から存在する仮想メソッド呼び出しの実行確率は高いと考えられるので、これらの補償コードを生成することはその実行のオーバヘッドに見合わないと考えられる。3.5.1節の実験結果から、コンパイル時に呼び出し先メソッドのオーバライドが起きていない仮想メソッド呼び出しは、その後メソッドのオーバライドが起きることは非常に少ないことが分かる。本方式が生成する仮想メソッド呼び出しは、動的クラスローディングの結果メソッドがオーバライドされた後のみ実行されるので、この仮想メソッド呼び出しの実行確率は低いと考えられる。従って、補償コードが挿入されても、その実行オーバヘッドは無視できる。補償コードの生成によって、本方式が生成する仮想メソッド呼び出しから脱出していたオブジェクトでも捕捉されているとして扱うことができる。この例を、図 3.9に示す。本方式による直接デバーチャル化によって仮想メソッド呼び出しが生成された場合でも、その特性を利用してこ

の手法を用いて補償コードを生成することで、脱出解析による最適化の機会を十分に得ることができる。

```
class Foo {
  int x, y;
  void m() { this.x++;}
  int caller() {
    Foo f = new Foo();

    f.x = 1;
    f.m();









    f.y = f.x + 2;
    System.out.println(f.y);



  }
}
```

a) ソースコード

```
class Foo {
  int x, y;
  void m() { this.x++; }
  int caller() {
    Foo f;
    Boolean heap_allocated = false;
    int t1 = 1;
    if (!is_code_patched) {
      t1++;
    } else {
      if (!heap_allocated) {
        f = new Foo();
        heap_allocated = true;
      }
      f.x = t1;
      f.y = t2;
      f.m();
      t1 = f.x;
      t2 = f.y;
    }
    t2 = t1 + 2;
    System.out.println(t2);
    if (heap_allocated) {
      f.x = t1;
      f.y = t2;
    }
  }
}
```

b) 脱出解析後のコード

図 3.9: 脱出解析の適用例

3.4 仮想メソッド呼び出しのデバーチャル化

本節では、従来提案されている仮想メソッド呼び出しのデバーチャル化手法について述べた後、コンパイラにおける仮想メソッド呼び出しのデバーチャル化の実装について述べる。

3.4.1節では局所クラス解析について述べ、3.4.2節では preexistence 解析について述べ、3.4.3節ではクラステストとメソッドテストについて述べる。3.4.4節では、コンパイラにおいて仮想メソッド呼び出しのデバーチャル化を実装する際のアルゴリズムを述べる。

3.4.1 局所クラス解析

局所クラス解析[22][85][86][87]、はデータフロー解析において、オブジェクト生成式（例えば new 式）を元に変数から型集合への写像を作り、制御フローに沿って伝搬する。制御フローの合流点では入力の型集合の和を取り出力へ伝搬する。この結果、仮想メソッド呼び出しのレシーバの型がオブジェクト生成式と同一であれば、レシーバが取るクラスは唯一であるので、仮想メソッド呼び出しを静的メソッド呼び出しに変換できる。さらに、変換された静的メソッド呼び出しのコードのみを生成すればよいので、制御フロー上に合流点を生成しない。この変換は動的クラスローディングを行う言語でも常に成り立つ。

局所クラス解析によって、javac や jikes version 1.06[93]でソースコードをバイトコードに変換する際に起きる型情報の喪失から、型情報を回復することもできる。図 3.10の例を用いて説明する。メソッド m()内のメソッド呼び出し a.equal()は、クラス A にあるメソッド equals()を呼び出すことを示している。しかし、javac コンパイラは、メソッド呼び出し

a.equal()のレシーバの静的な型情報として、クラス Object とメソッド equals()をクラスファイルに対して生成する。この javac コンパイラのメソッド呼び出しの生成が引き起こす型情報の喪失によって、メソッド呼び出し a.equals()に対するクラス階層解析の精度が低下する。A.equals()に対してクラス階層解析を適用すれば、このメソッドをオーバライドするメソッドが存在しないことが分かる。しかし、Object.equals()に対してクラス階層解析を適用すると String.equals()が存在するので、オーバライドするメソッドが存在する、と判断する。この結果、ソース上では A.equals()のみが呼び出されることを意図したプログラムに対して、レシーバの型情報が失われることによって仮想メソッド呼び出しの直接デバーチャル化が適用できなくなる。この問題は、局所クラス解析を適用することによって解決される。

実際に、この問題は java.lang.Object クラスで実装されている equals()メソッドや hashCode()メソッドを呼び出す際にメソッド呼び出し元で発生する。

```
class Object { boolean equals(Object o) { ... }; }

class String extends Object {
   boolean equals(Object o) { ... };   // Overrides equals()
}

class A extends Object { ... }         // Does not override equals()

class X {
  void m(A a) {
    a.equals();
  }
}
```

図 3.10: 呼び出し元で型情報が失われる例

hashCode() メソッド、 toString() メソッド、 equals() メソッドは、java.lang.Object クラスで実装されているので、全てのオブジェクトについて呼び出される可能性がある。これらのメソッドは、呼び出しの際に静的な型情報として java.lang.Object クラスを伴うことが多い。java.lang.Integer クラスや java.lang.String クラス等の基本的なクラスでは、hashCode()メソッドが final 宣言を伴って実装されている。これらの実装は非常に単純なので、局所クラス解析によってレシーバがこれらのクラスであることが分かれば、コンパイル時にメソッドをインライン展開可能である。

3.4.2 preexistence 解析

クラス階層の変更がメソッド実行中に発生し、メソッド呼び出しの直接デバーチャル化をコンパイル時に適用した際のクラス階層の仮定が無効になった場合においても、仮想メソッド呼び出しのレシーバに到達するオブジェクトが実行中のメソッドの呼び出し前に生成されていたならば、クラス階層の変化に影響されずに直接デバーチャル化されたコードを実行できる。この性質を preexistence[72]と呼ぶ。この方式では、直接デバーチャル化を無効化するメソッドの再コンパイルは次回のメソッド呼び出し時に行えばよい。従って、脱最適化で要求される複雑な再コンパイルに関する実装を必要としない。

Preexistence が成り立つことは、仮想メソッド呼び出しのレシーバがメソッドの実行中に変化しない場合を検出することによって示される。この検出方式として、メソッドの引数が直接メソッド呼び出しのレシーバに到達した場合を検出する invariant argument analysis を実装した。

3.4.3 クラステストとメソッドテスト

クラステスト[77]は、インライン展開されたメソッドのクラスとレシーバのクラスについて型テストを行い、一致した場合にはインライン展開されたコードを実行する。この場合、呼び出し先のメソッドを高速に実行できる。テストが失敗した場合には、呼び出し先メソッドの検索後メソッド呼び出しを行う。コード例を、図 3.11 a)に示す。メソッドテスト[72]は、インライン展開されたメソッドとレシーバのクラス内のメソッドについて型テストを行い、一致した場合にインライン展開されたコードを実行する。テストが失敗した場合はクラステストと同様である。コード例を、図 3.11 b)に示す。これらの方式では、１つの仮想メソッド呼び出しに対してコンパイル時に２種類以上のコードを生成するので、制御フロー上に合流点が生成される。

レシーバのクラスが、インライン展開されたメソッドのサブクラスであり、そのクラスでメソッドをオーバライドしていない場合、クラステストでは失敗となるが、メソッドテストでは成功となる。従って、メソッドテストの方が型テストの成功率が高い。しかし、メソッドテストはクラステストよりロードが１つ多いため、実行時オーバヘッドが大きい。コンパイラが２つのロードをコード移動や部分式共通化などの最適化対象にすることで、オーバヘッドを低減できる場合もある。

```
r0 = <receiver object>
r1 = load(r0 + #offset-of-class-in-object)

if (r1 == #address-of-proper-class) {
    <inlined code>
} else {
    r2 = load(r1 + #offset-of-method-in-class)
    r3 = load(r2 + #offset-code-in-method)
    call r3
}
```

a) クラステストのコード例

```
r0 = <receiver object>
r1 = load(r0 + #offset-of-class-in-object)
r2 = load(r1 + #offset-of-method-in-class)
if (r2 == #address-of-inlined-method) {
    <inlined code>
} else {
    r3 = load(r2 + #offset-code-in-method)
    call r3

}
```

b) メソッドテストのコード例

図 3.11: クラステストとメソッドテストのコード例[72]

3.4.4 デバーチャル化のアルゴリズム

本節では、コンパイラに仮想メソッド呼び出しのデバーチャル化を実装する場合の、デバーチャル化アルゴリズムを述べる。

3.3.1節で提案したコード書き換えによる直接デバーチャル化は、実装の容易さと、preexistence 解析や局所クラス解析よりも広い適用範囲を特徴とする。しかし、制御フローに合流点を生成することが、各種最適化によるコンパイル時間の増加や、補償コードによるコード量の増加を招く可能性がある。従って、できる限り制御フロー上に合流点を生成しないデバーチャル化方式を適用したい。一方、制御フロー上に合流点を生成しない脱最適化方式は、本方式と同様にコンパイル時にメソッドがオーバライドされていない全ての仮想メソッド呼び出し

に適用可能である。しかし、脱最適化は4.2節の関連研究でも述べたが実装が非常に複雑であるため、手間を考えた場合、候補に挙げにくい。

制御フロー上に合流点を生成しない直接デバーチャル化に、3.2節の関連研究で述べた preexistence 解析や局所クラス解析がある。適用範囲は本方式や脱最適化より狭いが、これらの方式は実装が容易である。preexistence 解析によって要求されるメソッドの再コンパイルは、メソッドの次回の起動時に行えばよい。局所クラス解析はメソッドの再コンパイルの必要が無く、解析は静的型付けを持つ言語では型に関するデータフロー方程式を解くだけでよい。これらの方式は、制御フローの上に合流点がないため、脱最適化と同等のコードを生成できる。もし、仮想メソッド呼び出しに対して preexistence 解析と局所クラス解析が適用できない場合、コード書き換えによるデバーチャル化を適用する。

最後に、コンパイル時に仮想メソッド呼び出しが取るレシーバのクラス階層において、すでにメソッドがオーバライドされていた場合、メソッドテストによる間接デバーチャル化を適用する。この方式では型テストが必要になり、制御フローの合流点が生成される。しかし、3.3.1節で述べた最適化を行うことで、合流点によるオーバヘッドを低減できる。

従って、コンパイラに仮想メソッド呼び出しのデバーチャル化を実装する場合、実装の容易さと最適化能力を考慮して、図 3.12のアルゴリズムに基づいてデバーチャル化を適用するのが適切である。このアルゴリズムでは、コンパイル中のメソッド `m()`内の全ての仮想メソッド呼び出しのレシーバに対して局所クラス解析と preexistence 解析が行われていること、実行中のプログラム `P` の現在のクラス階層全体に対してクラス階層解析が行われていること、を前提としている。

本方式と preexistence による仮想メソッド呼び出しの直接デバーチャル化は、コンパイル時の最適化と実行時ルーチンの協調によって実現される。図 3.13に、実行時のアルゴリズムを示す。

また、図 3.12のアルゴリズムによって、各仮想メソッド呼び出しへ適用される各デバーチャル化の適用範囲の分類を、図 3.14に示す。

```
V = コンパイル中のメソッド m()の仮想メソッド呼び出しの集合。
F = コンパイル中のメソッド m()のインタフェースメソッド呼び出しの集合。
while (V が空でない && F が空でない) do
  f = F から選んだインタフェースメソッド呼び出し。
  F から f を削除する。
  Cf = f に関するインタフェースクラスを実装するクラス集合。
  if (Cf が唯一のクラスを持つ) then
    Cf を仮想メソッド呼び出しで宣言される静的なクラスとして、
    インタフェースメソッド呼び出し f を仮想メソッド呼び出しに変換して V に登録し、
    元のインタフェースメソッド呼び出しを含むコードを生成する。
    /* プログラム P を実行中、f に関するインタフェースクラスが複数のクラスによって
       実装された場合に、f を仮想メソッド呼び出しに変換したコードを無効にするための
       コード書き換え要求を出す必要がある。 */
    プログラム実行中のメソッドオーバライド時に参照される集合 L に以下の tuple を登録
    （key：f に関するインタフェースクラス、
     action：コード書き換え要求、情報：[書き換えアドレス Af、書き換えコード Cf]）。
  else
    通常のインタフェースメソッド呼び出しを生成する。
  endif

  v = V から選んだ仮想メソッド呼び出し。
  V から v を削除する。
  Nv = クラス階層解析の結果に基づく、v における呼び出し先メソッドの集合。
  if (Nv が唯一のメソッドを持つ) then
    T = 局所クラス解析の結果に基づく、仮想メソッド呼び出し v のレシーバに到達する型集合。
    if (T がオブジェクト生成式の型と同一) then
      仮想メソッド呼び出し v に対して、
      制御フローの合流点を生成しない仮想メソッドの直接デバーチャル化を適用する。
    elif (preexistence 解析の結果より、
      仮想メソッド呼び出し v のレシーバがメソッド m()の実行中に不変である) then
      仮想メソッド呼び出し v に対して、
      制御フローの合流点を生成しない仮想メソッド呼び出しの直接デバーチャル化を適用する。
      /* プログラム P を実行中、メソッド n()がオーバライドされた場合に、
         メソッド m()の次の実行時に再コンパイル要求を出す必要がある。*/
      プログラム実行中のメソッドオーバライド時に参照される集合 W に以下の tuple を登録
      （key：呼び出し先のメソッド n()、
       action：再コンパイル要求、情報：コンパイル中のメソッド名 m()）。
    else
      仮想メソッド呼び出し v に対して、
      コード書き換えによる仮想メソッド呼び出しの直接デバーチャル化を適用し、
      直接デバーチャル化されたコードと、元のメソッド呼び出しを含むコードを生成する。
      /* プログラム P を実行中、メソッド n()がオーバライドされた場合に、
         仮想メソッド呼び出し v のデバーチャル化されたコードを無効にするための
         コード書き換え要求を出す必要がある。 */
      プログラム実行中のメソッドオーバライド時に参照される集合 W に以下の tuple を登録
      （key：呼び出し先のメソッド n()、
       action：コード書き換え要求、情報：[書き換えアドレス Amv、書き換えコード Cmv]）。
    endif
  else
    メソッドテストを適用して、間接デバーチャル化されたコードと、
    元のメソッド呼び出しを含むコードの両方を生成する。
  endif
```

図 3.12: 仮想メソッド呼び出しのデバーチャル化のコンパイラアルゴリズム

```
プログラム P を実行する。
while TRUE do
  if (動的クラスローディングが発生) then
    C = ロードされたクラス。
    クラス階層解析を行う。
    if (メソッドオーバライドが発生) then
      n = メソッドオーバライドが発生したメソッド。
      while (コンパイル時に登録した集合 W に、key として n を含む tuple が存在する) do
        T = W から見つけた tuple。
        W から T を除く。
        switch T.action do
        case コード書き換え要求:
          /* メソッド T.m に存在するメソッド n の仮想メソッド呼び出しの
             コード書き換えを必要とする場合 */
          アドレス T.Amv を指定された命令 T.Cmv でアトミックに書き換えを行う。
          break
        case 再コンパイル要求:
          /* メソッド n の仮想メソッド呼び出しを含むメソッド T.m が次に呼び出された際に
             再コンパイルを必要とする場合 */
          メソッド T.m のエントリのネイティブコードを、JIT コンパイラを起動するための
          スタブへの無条件分岐命令でアトミックに書き換えを行う。
          break
        endswitch
      endwhile
    endif

    while (コンパイル時に登録した集合 L に、
           key として C が実装するインタフェースクラスを含む tuple が存在する) do
      T = L から見つけた tuple。
      L から T を除く。
      switch T.action do
      case コード書き換え要求:
        /* メソッド T.f に存在するインタフェースメソッド呼び出しの
           コード書き換えを必要とする場合 */
        アドレス T.Af を指定された命令 T.Cf でアトミックに書き換えを行う。
        break
      endswitch
    endwhile
  endif
endwhile
```

図 3.13: 仮想メソッド呼び出しのデバーチャル化の実行時アルゴリズム

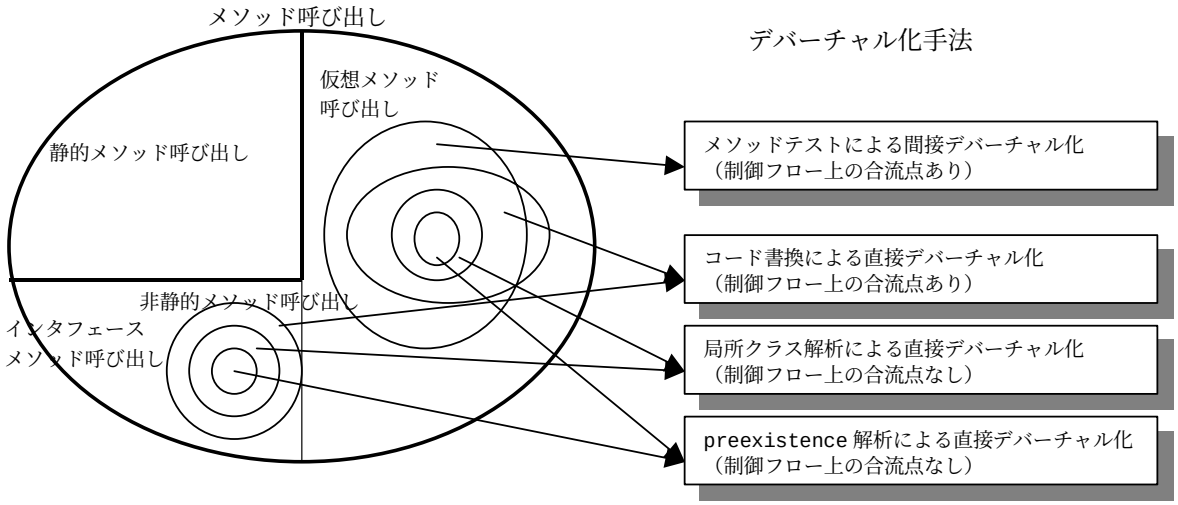


図 3.14: デバーチャル化手法の適用分類

3.5 実験結果

本節では、前節で述べたアルゴリズムに基づいてコンパイラに実装した仮想メソッド呼び出しのデバーチャル化の効果を、複数のプラットフォームにおける実験結果から示す。実験のために、IBM Developers Kit, Java Technology Edition, Version 1.3.1 の Just-In-Time コンパイラに、3.4.4節で述べたデバーチャル化方式を実装した。このコンパイラは、静的メソッド呼び出しのインライン展開、仮想メソッド呼び出しのデバーチャル化とインライン展開、脱出解析、データフロー解析を用いた複写除去・定数伝搬・不要コード除去・共通部分式除去・スカラへの置き換え・冗長な Java 例外検査の除去、等多くの最適化を実装したコンパイラである。本章で述べた最適化は、ループ皮むき、分岐の偏りを考慮したコードの複製によって得られるより正確な型情報に基づいた直接デバーチャル化、脱出解析のための補償コード生成、を除いて全て実装した。

仮想メソッド呼び出しの特性の測定は、PowerPC プラットフォーム上で行った。性能評価は、PowerPC、IA-32、IA-64 と異なる３つのプラットフォームで行った。

PowerPC では、IBM Developers Kit for AIX, Java Technology Edition, Version 1.3.1 の Just-In-Time コンパイラを用いて、IBM 社の RS/6000 44P モデル 170（POWER3-II 400MHz、メモリ 768MB）に OS は AIX 4.3.3 を用いた。IA-32 では、IBM Developers Kit for Windows on IA-32, Java Technology Edition, Version 1.3.1 の Just-In-Time コンパイラを用いて、IBM 社の IntelliStation Mpro モデル 6850（Pentium 4 Xeon 2.2MHz x 2、メモリ 1GB）に OS は Windows 2000 を用いた。IA-64 では、IBM Developers Kit for Windows on Itanium, Java Technology Edition, Version 1.3.1 の Just-In-Time コンパイラを用いて、IBM 社の IntelliStation Z Pro モデル 6894（Itanium 800MHz x 2、メモリ 2GB）に OS は Windows XP Advanced Server を用いた。

プログラムには、SPECjvm98[6]に含まれる７つのプログラム（compress、 jess、 db、 javac、 mpegaudio、 mtrt、 jack）、SPECjbb2000[94]を Darko Stefanovic がトランザクション数固定に変

更した pseudojbb、エージェント構築環境の caribbean[99]を用いた。この処理系には、頻繁に実行されるメソッドだけをコンパイルする選択コンパイル機能が実装されているが、今回の実験では全てのメソッドをコンパイルする設定で行った。

表 3.1: ベンチマークプログラムの説明

ベンチマーク	プログラムの説明
compress	LZW 法を用いた圧縮・展開プログラム。SPECjvm98 の１つ。size=100 で実行。
jess	NASA の CLIPS エキスパートシステム。SPECjvm98 の１つ。size=100 で実行。
db	データ管理プログラム。SPECjvm98 の１つ。size=100 で実行。
javac	JDK 1.0.2 の java class ファイルを生成するコンパイラ。SPECjvm98 の１つ。size=100 で実行。
mpegaudio	MP3 のデータを伸長するプログラム。SPECjvm98 の１つ。size=100 で実行。
mtrt	２つのスレッドが走るレイトレーシングプログラム。SPECjvm98 の１つ。size=100 で実行。
jack	パーサジェネレータ。SPECjvm98 の１つ。size=100 で実行。
pseudojbb	三層モデルを用いてトランザクション処理を行う。SPECjbb2000 を Darko Stefanovic がトランザクション数固定に変更したもの。warehouse=1 で実行。
caribbean	エージェント開発環境。40000 エージェントによるトランザクション処理を実行。

3.5.1 仮想メソッド呼び出しの特性

まず、最適化前のプログラムの静的メソッド呼び出し、仮想メソッド呼び出し、インタフェースメソッド呼び出しの特性を、表 3.2に示す。33.4%～97.2%（平均 77.6%）の仮想メソッド呼び出しが、単一メソッドを呼び出している。mpegaudio を除いて、アプリケーションのクラス内の多くの仮想メソッド呼び出しが単一メソッドを呼び出している。この結果から、多くの仮想メソッド呼び出しに対して直接デバーチャル化を適用する機会があると予想される。compress は静的メソッド呼び出しが非常に多いのでデバーチャル化の効果がないと予想される。また、db ではインタフェースメソッド呼び出しの回数が多いことが分かる。

表 3.2: 静的メソッド呼び出しと仮想メソッド呼び出しの特性

ベンチマーク	Static Call	Virtual Call	Monomorphic Virtual Call %		Interface Call	Monomorphic Interface Call %	
			Lib.	App.		Lib.	App.
compress	225,978,149	12,432	47.1%	24.9%	497	43.7%	55.1%
jess	78,372,934	36,871,584	0.2%	83.8%	706,556	0.0%	0.7%
db	52,995,096	52,529,625	0.1%	97.1%	14,931,590	0.0%	100.0%
javac	58,055,300	48,354,363	5.1%	62.3%	3,379,147	0.0%	99.8%
mpegaudio	98,467,629	9,853,819	0.2%	33.2%	182,271	0.1%	99.9%
mtrt	17,408,325	269,740,775	0.3%	90.6%	453	48.3%	50.3%
jack	25,516,426	25,219,183	20.3%	59.5%	4,155,366	0.0%	55.0%
pseudojbb	129,668,163	185,124,778	14.8%	81.7%	4,401,509	0.02%	100.0%
caribbean	38,500,974	30,215,699	30.4%	32.5%	684,599	23.9%	58.5%
平均			77.6%			81.6%	

Static Call: 静的メソッド呼び出しの総実行回数。
Virtual Call: 仮想メソッド呼び出しの総実行回数。
Monomorphic Virtual Call: 単一メソッドを呼ぶインタフェースメソッド呼び出しが実行された割合。
Interface Call: インタフェースメソッド呼び出しの総実行回数。
Monomorphic Interface Call: 単一メソッドを呼ぶインタフェースメソッド呼び出しが実行された割合。
Lib.: メソッド呼び出しが Java ライブラリクラス内で実行された割合。
App.: メソッド呼び出しがアプリケーションクラス内で実行された割合。

次に、メソッドテスト、コード書き換え、局所クラス解析、preexistence 解析、と４つのデバーチャル化手法を適用したときの非静的メソッド呼び出しの特性の変化を調べた。図 3.15に、仮想メソッド呼び出しにデバーチャル化手法を適用した場合、どのデバーチャル化方式によって仮想メソッド呼び出しの実行回数が減少したかを示す。左側のバーがデバーチャル化手法を全く適用しない場合、右側のバーがメソッドテスト、コード書き換え、局所クラス解析、preexistence 解析の４つのデバーチャル化手法を適用した場合を示す。コード書き換えによるメソッド呼び出しの実行回数の減少は、jess、db、mtrt、jack、pseudojbb、caribbean で大きい。

db では、`spec.benchmarks._209_db.Database` クラスの `shell_sort()`メソッドの中にカーネルループがある。このループ内ではインスタンス変数の参照を行う小さなメソッド、具体的には `java.util.Vector` クラスの `elementAt ()`メソッド、を頻繁に呼び出している。これらのメソッド呼び出しが直接デバーチャル化によってインライン展開されたため、仮想メソッド呼び出しの回数が減少した。

mtrt では、`spec.benchmarks._205_raytrace.Octnode` クラスの `Intersect()`メソッドの中にカーネルループがある。このループ内では、インスタンス変数の参照を行う小さなメソッド、具体的には `spec.benchmarks._205_raytrace.Point` クラスの `GetX()`、`GetY()`、`GetZ()`、メソッドを頻繁に呼び出している。これらのメソッド呼び出しが直接デバーチャル化によってインライン展開されたため、仮想メソッド呼び出しの回数が減少した。

局所クラス解析による実行回数の減少は、db、javac で大きい。db では、`spec.benchmarks._209_db.Database` クラスの `set_index ()`メソッドの内のループで `java.util.Enumeration` クラスの `hasMoreElement ()`、`nextElement ()`メソッド、の２つのインタフェースメソッド呼び出しが存在する。局所クラス解析の結果を用いたクラス階

層解析によってこれらのインタフェースメソッド呼び出しを仮想メソッド呼び出しに変換でき、さらにその仮想メソッド呼び出しを直接デバーチャル化できた。

preexistence 解析は、主に制御フローの合流点を持つ直接デバーチャル化のコードから合流点を取り除く変形に使われるため、仮想メソッド呼び出しの実行回数はほとんど変化しない。

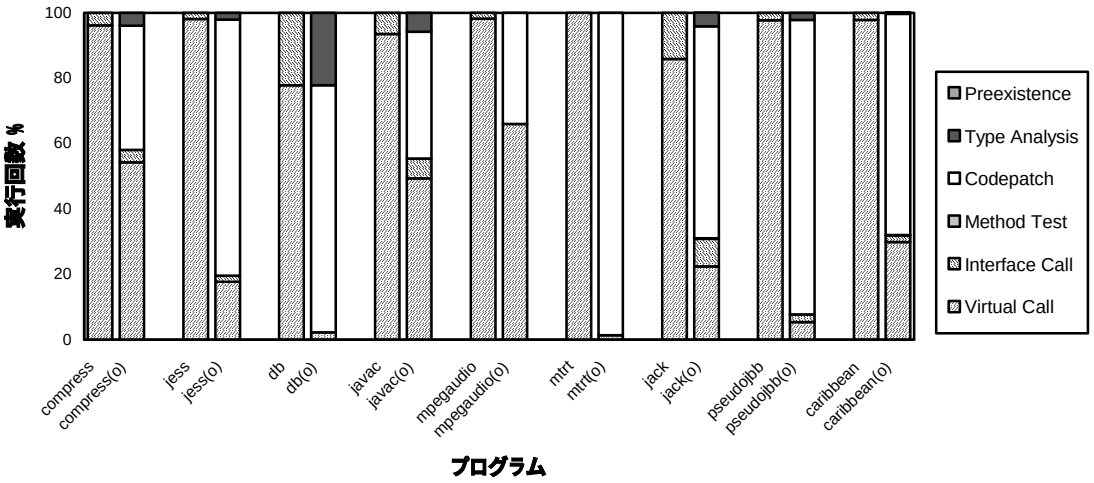


図 3.15: それぞれのデバーチャル化手法によって減少した非静的メソッド呼び出し実行回数
(o)無しはデバーチャル化手法適用前、(o)有りはデバーチャル化手法適用後

次に、メソッドテスト、コード書き換え、局所クラス解析、preexistence 解析、と４つのデバーチャル化手法を順々に適用していったときの、静的メソッド呼び出し、仮想メソッド呼び出しの特性の変化を調べた。表 3.3に、メソッドテスト、コード書き換え、局所クラス解析、preexistence 解析、と４つのデバーチャル化手法の適用によって、仮想メソッド呼び出し回数と直接デバーチャル化されたコードの実行回数の減少の様子を示す。４つの手法全てを適用した場合には、34.0%～98.6%（平均 69.8%）仮想メソッド呼び出しの回数が減少した。

仮想メソッド呼び出しの実行回数の多いベンチマークでは、mtrt での仮想メソッド呼び出し回数の減少が特に大きい。この理由は前述しているが、インスタンス変数の参照を行う小さなメソッドの呼び出しが頻繁に行われ、そのメソッド呼び出しが直接デバーチャル化によってインライン展開されたためである。また、コード書き換えによって直接デバーチャル化された仮想メソッド呼び出しのうち 0.0%～88.9%（平均 29.3%）が、preexistence 解析によって制御フロー上の合流点を生成しないデバーチャル化に変換可能である。

表 3.3: 実行回数の減少率に基づく各デバーチャル化手法の効果

ベンチマーク	N から MCTP での削減率			M から MCTP での削減率	MC から MCTP での削減率	MCT から MCTP での削減率
	仮想メソッド呼び出し	インタフェースメソッド呼び出し	両方	メソッドテストによってテスト無しで実行できるコード	コード書き換えによってテスト無しで実行できるコード	コード書き換えによってテスト無しで実行できるコード
compress	43.5%	0.6%	41.9%	100.0%	18.6%	22.2%
jess	82.0%	0.7%	80.4%	100.0%	6.5%	26.0%
db	97.1%	99.9%	97.7%	100.0%	0%	0%
javac	47.1%	8.8%	44.6%	91.0%	7.0%	22.1%
mpegaudio	32.8%	99.8%	34.0%	100.0%	0%	88.9%
mtrt	98.6%	0.7%	98.6%	100.0%	6.6%	31.2%
jack	74.0%	39.8%	69.1%	100.0%	3.0%	22.8%
pseudojbb	94.4%	0.9%	92.3%	100.0%	2.3%	22.0%
caribbean	69.4%	11.7%	68.1%	100.0%	1.0%	36.3%
平均	71.2%	31.4%	69.8%	98.9%	5.5%	29.3%

N: デバーチャル化適用無し。
M: メソッドテストを適用。
MC: メソッドテスト、コード書き換えを適用。
MCT: メソッドテスト、コード書き換え、局所クラス解析を適用。
MCTP: メソッドテスト、コード書き換え、局所クラス解析、preexistence 解析を適用。

表 3.4に、メソッドテスト、コード書き換え、局所クラス解析、preexistence 解析と４つのデバーチャル化手法を全て適用した場合に、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がどの程度の割合で実行されるか、書き換えられた仮想メソッド呼び出しサイトの数、preexistence 解析によって再コンパイルされたメソッド数、を示す。直接デバーチャル化されたコードが実行される割合は、96.6%〜100%（平均 99.5%）と非常に高い。つまり、最初のメソッド呼び出し時にオーバライドされていない仮想メソッド呼び出しは、後からオーバライドされることが非常に少ない。また、コード書き換えが起きる呼び出しサイト数や再コンパイルされるメソッドの数も、非常に少ない。この傾向はプログラムの特性に大きく依存し、今回の実験の対象が主にベンチマークプログラムであることも一因であると考えられる。

表 3.4: 直接デバーチャル化された仮想メソッド呼び出しサイトの特性

ベンチマーク	テスト無しに実行可能なコードの実行割合	コード書き換えが起きた仮想メソッド呼び出しサイト数	再コンパイルされたメソッド数
compress	96.6%	25	4
jess	100%	28	4
db	100%	25	4
javac	99.9%	28	8
mpegaudio	99.9%	25	4
mtrt	100%	25	4
jack	99.5%	29	4
pseudojbb	100%	13	2
caribbean	100%	77	4
平均	99.5%	24.8	4.3

3.5.2 性能評価

各種デバーチャル化手法の組合せによる性能向上を測定した結果を、PowerPC の場合を図 3.16に、IA-32 の場合を図 3.17に、IA-64 の場合を図 3.18に示す。全ての値は、デバーチャル化を行わなかったときの実行時間に対する性能向上比である。グラフ内のそれぞれのバーは、以

下のデバーチャル化を適用したときの値である。

- M　　メソッドテスト
- MC　　メソッドテスト、コード書き換え
- MCT　　メソッドテスト、コード書き換え、局所クラス解析
- MCTP　　メソッドテスト、コード書き換え、局所クラス解析、preexistence 解析

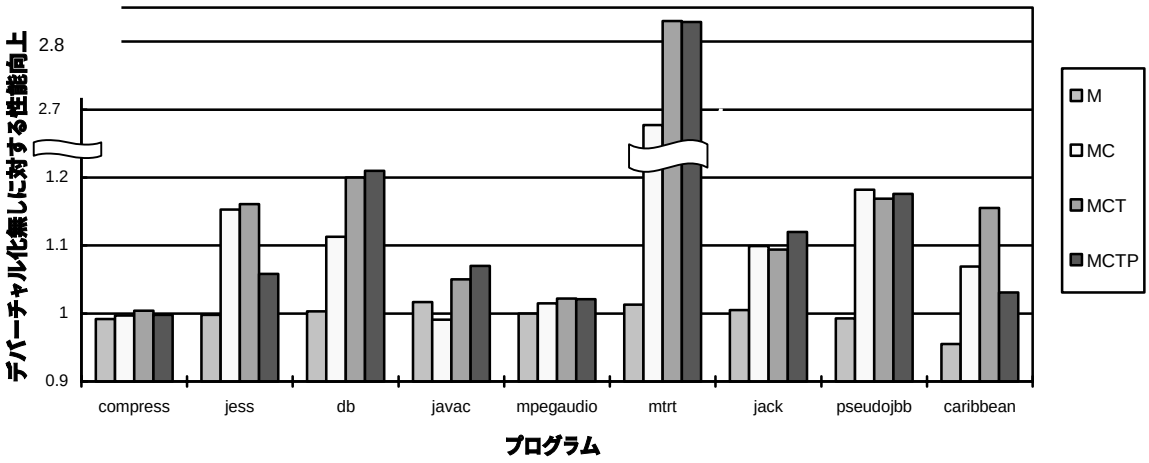


図 3.16: SPECjvm98 と SPECjbb2000 の性能向上割合（PowerPC）

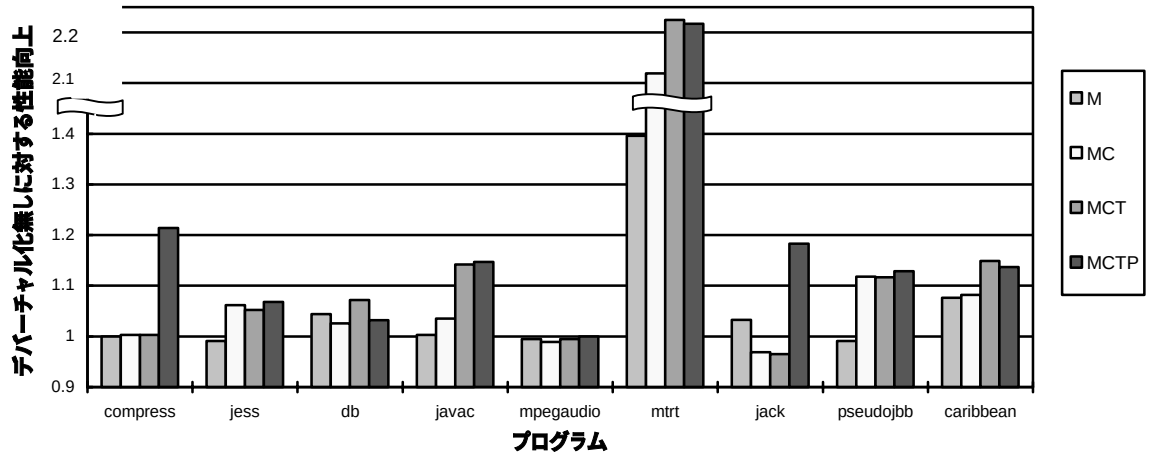


図 3.17: SPECjvm98 と SPECjbb2000 の性能向上割合（IA-32）

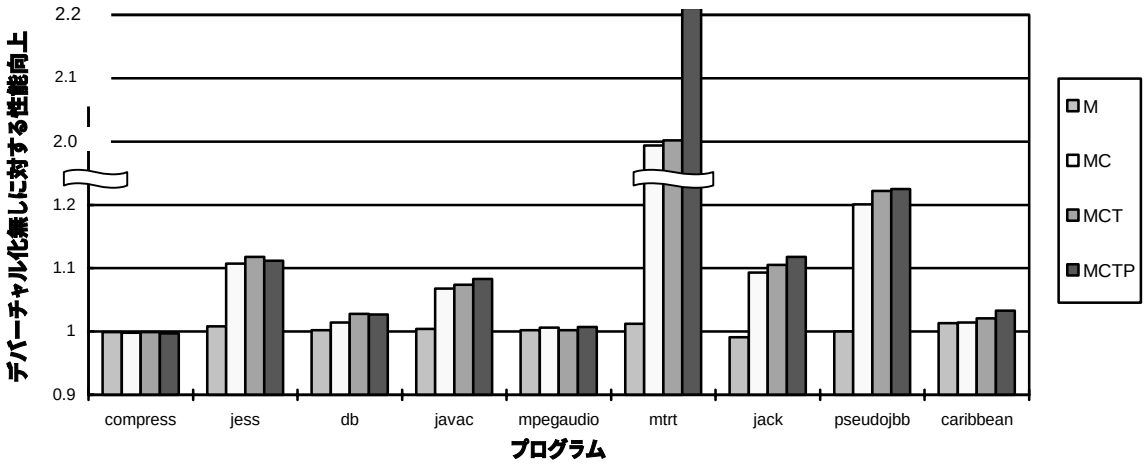


図 3.18: SPECjvm98 と SPECjbb2000 の性能向上割合（IA-64）

間接デバーチャル化によるメソッドテストだけを適用した場合には、ほとんど性能に変化がない。PowerPC の caribbean では、性能が悪化している。一方、IA-32 の caribbean では、性能が向上している。

コード書き換えによる直接デバーチャル化も適用した場合には、相乗平均で PowerPC では 16.9%、IA-32 では 10.8%、IA-64 では 12.7%の性能向上が得られた。特に mtrt での性能改善が大きい。3.5.1節でも述べたが、このプログラムでは、インスタンス変数の参照を行う小さなメソッドの呼び出し頻度が非常に多い。このメソッド呼び出しが直接デバーチャル化によってインライン展開されて、オーバヘッド無しで実行可能になったため、性能が大きく向上した。

さらに、局所クラス解析による直接デバーチャル化も適用した場合には、相乗平均で PowerPC では 20.2%、IA-32 では 13.6%、IA-64 では 13.6%の性能向上が得られた。特に、db での性能改善が大きい。この理由は、3.5.1節で述べたが、局所クラス解析の結果を用いたクラス階層解析によってインタフェースメソッド呼び出しを仮想メソッド呼び出しに変換でき、さらにその仮想メソッド呼び出しを直接デバーチャル化してインライン展開されたコードを実行できたためである。さらに、PowerPC と IA-32 での javac の性能向上も大きい。このプログラムでは、`hashCode()`、`equals()`、`toString()`メソッドを用いて String オブジェクトを操作することが多く、これらのメソッド呼び出しが局所クラス解析の結果を用いてコード書き換えによる直接デバーチャル化が可能になり、インライン展開されたコードをテスト無しで実行できたためである。

最後に、preexistence 解析による直接デバーチャル化も適用し、3.4.4節で述べたデバーチャル化の戦略を全て実行した場合には、相乗平均で PowerPC では 18.3%、IA-32 では 17.9%、IA-64 では 14.7%の性能向上が得られた。IA-64 の mtrt で、さらに性能が改善した理由は以下の通りである。preexistence 解析を用いたデバーチャル化によって、頻繁に呼び出しされる `spec.benchmarks._205_raytrace.OctNode` クラスの `Intersect()`メソッド内にある仮想メソッド呼び出しが、脱出解析においてオブジェクトの脱出点となる仮想メソッド呼び出し

を生成することなく直接デバーチャル化された。その結果、捕捉されたオブジェクトの各フィールドに対してスカラへの置き換えを適用できたためである。

一方、PowerPC の jess と caribbean で性能が落ちている理由は不明である。

3.6 まとめ

本章では、動的クラスローディングを行う言語における、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した。メソッドの呼び出し先を一意に決定可能な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の２種類のコードをコンパイル時に生成し、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書き換えによって後者を実行する手法を述べた。本手法では、コードの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、脱最適化で要求される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。一般に制御フロー上の合流点はコンパイラの最適化を妨げるが、制御フロー上に合流点が存在しても制御フローが存在しない場合と同等のコードに最適化できることを示した。さらに、本手法と他のデバーチャル化手法を合わせて Java Just-In-Time コンパイラに実装し、SPECjvm98、SPECjbb2000、エージェントベンチマークにおいてデバーチャル化を行わない場合に比べ、相乗平均で PowerPC では 15.3%、IA-32 では 20.8%、IA-64 では 17.7%の性能向上が得られた。

第4章 オブジェクトの型検査の高速化

4.1 はじめに

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に重要な機能の１つである。型検査は、右辺のオブジェクトのクラス型を正しく左辺のクラス型に変換（キャスト）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であるならば、そのオブジェクトは左辺の型へ変換可能である。Java 言語等の静的な型付けを持つ言語はコンパイル時にこの型変換を評価できるが、コンパイル時に全ての型検査が行えるわけではない。メソッド間で受け渡される参照型のオブジェクトにおいて、実行時にオブジェクトの実クラスが決まった際に例外検査を行う場合など、実行時に型検査を行うほうが効率よいものがある。

Java 言語はテンプレートを持たず、型システムにおいてパラメタ型を持たない。そのため `java.util.Vector` クラス等のコレクションクラスでは、最も汎用的なオブジェクトの型である `Object` 型をコレクションの要素型としている。そして、コレクション要素のオブジェクトを代入する際に汎用的な `Object` 型に暗黙的に型変換することで、多態的な入れ物を実現している。静的な型付けを持つ言語で多様性を利用したプログラミングを行う際には、実際に使用されるクラス型を考慮する必要がある。従って、コレクションクラスから要素を取り出す場合には、取り出すオブジェクトの実際のクラス型への明示的な型変換が必要となる。この型変換は、Java 言語では `Object` 型からのダウンキャストを行う型検査として行われる。

Java 言語において型検査は、`checkcast` バイトコード命令による明示的な型変換、`instanceof` バイトコード命令による型変換が可能であるかの検査、`aastore` バイトコード命令による配列へオブジェクトを代入する際の型検査、の３種類として行われる。図 4.1に、SPECjvm98 ベンチマークの db のカーネルループで型検査（String）が行われている例を示す。２重ループの内側で型検査が行われるこの例からも、Java 言語処理系の高速化のために、オブジェクトの型検査の高速化は重要であることが分かる。

```
public class Entry
{
  Vector items;
  ...
}

void shell_sort(int fn)
{
  String s1, s2;
  Entry e;
...
  for (gap = n/2; gap > 0; gap/=2)
    for (i = gap; i < n; i++)
      for (j = i-gap; j >=0; j-=gap) {
        s1 = (String)index[j].items.elementAt(fn);     // Object 型から String 型へ変換
        s2 = (String)index[j+gap].items.elementAt(fn); // Object 型から String 型へ変換
....
      }
}
```

図 4.1: ベンチマークプログラム中での型変換の例

単純な型検査の実装では、右辺のオブジェクトが左辺のクラス型の下位クラスに存在するかどうかを調べるため、クラス階層をたどる必要がある。より高速な型検査の実現のために、クラス階層を表にまとめ、型検査時に表を引くことによってクラス階層を辿らず定数時間で型検査を行う手法が従来提案されている[95][96]。また、この方式を Java 言語へ適用した研究[97]もある。これらの手法は、Java 言語等の動的クラスローディングを行う言語では、表の再構成のオーバヘッド等が存在する。本章では、型検査のうち頻繁に実行される部分をインライン展開することによって高速化する方法を提案する。本手法を Java Just-In-Time コンパイラに実装し、SPECjvm98 と SPECjbb2000 において、平均して PowerPC では 36%、IA-64 では 14%の性能向上が得られた。この方法は従来の方法と比較して実装が簡単であるうえ、従来の方法と同等以上の性能向上が得られる。

以下、4.2節では関連研究を述べる。4.3節では、オブジェクトの型検査のコードをインライン展開して高速化する手法について述べる。4.4節では、プログラムを用いて本方式を評価した結果を示す。4.5節では、本章のまとめを述べる。

4.2 関連研究

単純な型検査の実装では、右辺のオブジェクトが左辺のクラス型の下位クラスに存在するかどうかを調べるため、クラス階層をたどるループを実行する。ループを実行する代わりに、クラス階層の関係を表に表して型検査の際に表を引くことによって、定数時間で型検査を行う基本的なアイデアは、Cohen によって提案された[95]。この手法では、それぞれのクラス毎に、クラス階層の根のクラスまでの継承関係を表にしたものを持つ。このため、クラスが増えると表が非常に大きくなる、という欠点がある。この表を、ビット表現を用いて疎な表を密に構成することによって圧縮する方法を Vitek らが提案した[96]。

Alpern らは、Cohen のアルゴリズムを Java JIT コンパイラに実装して、性能評価を行った[97]。この方式は、動的クラスローディングが起きた場合も効率よく処理可能である。しかし、性能

評価の対照に用いた方式が、型検査の高速化の処理をネイティブコードの末尾に飛んでから行う方式であるため、実験において他の処理系より大きな性能向上が得られた可能性がある。

Intel の Java JIT コンパイラでは、本方式と同様に型検査において頻繁に実行される場合をインライン展開している。Intel の方式ではクラス階層を２段上まで遡るコードをインライン展開している[98]。SPECjvm98 に含まれる javac プログラムを実行した場合、両方式において型検査の 92%はインラインコードで処理されている。本方式のほうが、Intel の方式より生成されるコードの大きさが小さい。

4.3 オブジェクトの型検査のインライン展開

本節では、オブジェクトの型検査をインライン展開して高速化する手法について述べる。4.3.1節では型検査のインライン展開の方法と展開されたコードのアルゴリズムについて述べ、4.3.2節では局所クラス解析の結果を用いて、冗長な型検査の生成を抑制するアルゴリズムについて述べる。4.3.3節では、型検査に関して生成されるネイティブコードについて考察する。

4.3.1 型検査のインライン展開

型検査は、右辺のオブジェクトのクラス型が、正しく左辺のクラス型に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であるならば、そのオブジェクトは変換可能である。単純な型検査の実装では、右辺のオブジェクトが左辺のクラス型の下位クラスに存在するかどうかを調べるため、クラス階層をたどる必要がある。しかし、実際には4.4節において示される特定の条件下での型検査が多い。この結果に基づいて、頻繁に行われる検査をネイティブコードにインライン展開して生成することによって、型検査を高速化する。

`checkcast` 命令の型検査をインライン展開した場合の擬似コードを図 4.2に示す。展開されたコードは、以下のアルゴリズムで型検査を行う。

1. １行目では、検査するオブジェクトが null であるかを調べる。もし null であったら、左辺に null を渡す。
2. ２行目では、検査するオブジェクトが配列オブジェクトであるかを調べる。もし、配列オブジェクトであったら、用意された実行時ルーチンで型検査を行う。この特別処理を行う理由は以下の通りである。

   IBM Java VM のオブジェクトの形を図 4.3に示す。通常のオブジェクトは、クラス毎に用意されるテーブル情報へのポインタを持つ。配列オブジェクトは `java.lang.Object` 型の派生クラスとして特別に扱われ、テーブル情報へのポインタが存在した位置に配列長が存在する。さらに型検査のために、通常オブジェクトより複雑な処理が必要となる。多くのプログラムでは、配列オブジェクトが型検査されることは少ないので、C 言語で書かれた実行時ルーチンで処理される。

3. ３行目では、検査するオブジェクトの型が、キャスト先の型と等しいか調べる。もし、等しいならば、左辺に検査済みのオブジェクトを渡す。
4. ４行目では、オブジェクトが型検査を行った時に最後に成功したキャスト型と、キャスト先の型が等しいかを調べる。もし、等しいならば、左辺に検査済みのオブジェクトを渡す。
5. ５行目では、オブジェクトが型検査を行った時に最後に失敗したキャスト型と、キャスト先の型が等しいかを調べる。もし、等しいならば、例外を発生する。
6. ６行目では、その他の場合を扱うために、C 言語で書かれた実行時ルーチンで型検査を行う。ここでは、必要があればクラス階層を辿って型検査を行う。この結果、キャストが成功したら、検査するオブジェクトにキャスト先の型を最後に成功したキャスト型として保存する。さらに、左辺に検査済みのオブジェクトを渡す。
7. ７行目では、キャストが失敗した状態であるので、検査するオブジェクトにキャスト先の型を最後に失敗したキャスト型として保存する。さらに、例外を発生する。

`instanceof` 命令の型検査をインライン展開した場合には前記において、「左辺に検査済みのオブジェクトを渡す」が「１を返す」に、「例外を発生する」が「０を返す」、となる。インライン展開を行われたそれぞれのテストは、２～３命令で高速に実行可能である。

Java のプログラム

```
Type to = (Type)from;
```

ネイティブコードの擬似コード

```
1: if (from == NULL) {to = NULL;}
2: else if (is_array_object(from)) {call expensive_test in C}
3: else if (from.type == Type) {to = from;}
4: else if (from.type.lastsucc == Type) {to = from;}
5: else if (from.type.lastfail == Type) {throw exception}
6: else if (call expensive_test in C) {to = from; from.type.lastsucc = Type;}
7: else { from.type.lastfail = Type; throw exception}
```

図 4.2: 型検査のインライン展開の擬似コード

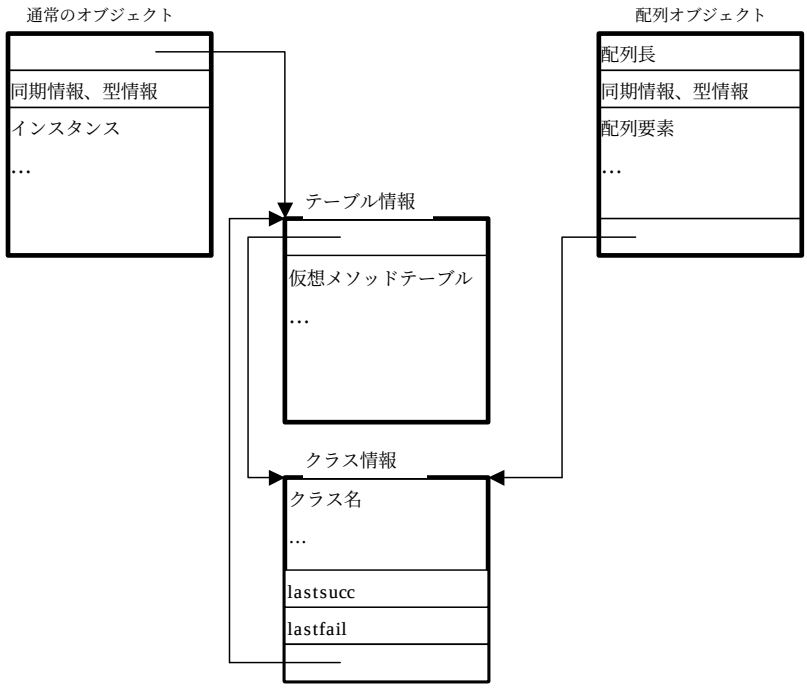


図 4.3: オブジェクトの形

4.3.2 冗長な型検査の抑制

本節では、局所クラス解析の情報を用いて、冗長な型検査を抑制する方法について述べる。4.3.2.1節では、型検査の除去について述べる。4.3.2.2節では、型検査をインライン展開する際に不必要な検査の生成を抑制する方法について述べる。

4.3.2.1 型検査の除去

コンパイル時の最適化として、局所クラス解析と定数伝搬を適用することによって、コンパイル時に型検査の結果が確定できる場合ができる。この時、実行時に型検査を行う必要がなくなるので、型検査を除去することができる。この型検査の除去は、図 4.4に示すアルゴリズムによって行うことができる。このアルゴリズムでは、コンパイル中のメソッド `m()`内の全ての型検査に対して局所クラス解析と定数伝搬が行われていることを前提としている。

```
T = コンパイル中のメソッド m()の型検査の集合
while T が空でない do
  t = T から選んだ型検査。
  T から t を削除する。
  /* t は (Ct) O = t Ot という代入文の形式を仮定する。
     Ot は型検査されるオブジェクト、Ct は変換先のクラス、O は変換後のオブジェクト。 */
  V0 = 定数伝搬の結果に基づく、Ot が取る定数値。
  if (V0 == null) then
    switch t do
    case checkcast 命令:
      t を、左辺値へ null を代入する代入文に置き換える。
      break
    case instanceof 命令:
      t を、左辺値へ 0 を代入する代入文に置き換える。
      break
    case aastore 命令:
      t の代入文の右辺値を null に置き換える。
      break
    endswitch
  else
    C0 = 局所クラス解析の結果に基づく、Ot が取るクラス集合。
    if (C0 は型検査 C の左辺のクラス Ct の下位クラスに含まれる) then
      t を除去。        /* 型検査 t は必ず成功する */
    elif ((Ot を Ct への変換を行う instanceof 命令が、t の前に存在する) &&
          (instanceof 命令の結果を評価した if 文の then 節内で、t が支配されている) &&
          (instanceof 命令実行後、Ot が t までに変更されていない)) then
      t を除去。        /* 型検査 t は必ず成功する */
    else
      型検査 t をインライン展開する。/* 4.3.2.2 節で置き換えられる */
    endif
  endif
endwhile
```

図 4.4: 型検査除去のアルゴリズム

4.3.2.2 インライン展開する検査の削減

前節のアルゴリズムを適用し型検査が削除できない場合でも、局所クラス解析などの情報からインライン展開する検査のコードを削減することができる。この検査コードの削減は、図 4.4 のアルゴリズムの「型検査 t をインライン展開する」を、図 4.5のアルゴリズムで置き換えることで行うことができる。

C_0 = 局所クラス解析の結果に基づく、O_t が取るクラス集合。
if ((変換先のクラス C_t が final 宣言されている) && (C_0 == C_t)) **then**
  /* C_t の下位クラスは存在しない ***/**
  4.3.1 節の 1.〜3.のみのコードを生成する。
elif (C_0 は null を含まない) **then**
  /* C_0 には null は到達しない***/**
  4.3.1 節の 2.〜7.のみのコードを生成する。
elif (C_0 は、配列のシグネチャを持つクラスや配列オブジェクトを取る可能性のあるクラス（java/lang/Object、java/io/Serializable、java/io/Serializable）を含まない)
then
  4.3.1 節の 1.、3.〜7.のみのコードを生成する。
else
  4.3.1 節の 1〜7.のコードを生成する。
endif

図 4.5: 検査コードの削減アルゴリズム

4.3.3 実際のコード例

本節では、型検査において実際に生成されるネイティブコードを、本方式と他の方式で比較する。図 4.6に、本方式と Jikes RVM に実装された方式において、通常オブジェクトへの型検査に対して生成されるネイティブコードを PowerPC の命令セットを用いて示す。

本方式では、最短であるキャスト先のクラスと検査されるオブジェクトのクラスが等しい場合は、メモリアクセスが１回で済み、３命令で終了する。この比較が成り立たなかった場合、さらに２回メモリアクセスを必要とする。さらに、これらのインライン展開されたコードの検査が失敗した場合、C 言語で書かれた実行時ルーチンが呼ばれるので、長い実行時間がかかる。

Jikes RVM の方式では、階層の深さが表の大きさ以下であるか調べる、階層の深さから表を実際に引いて対応するクラスを求める、ための２つの必要な値を得るためにメモリアクセスを３回行う。本方式のキャスト前後のクラスが等しい場合に比べ、オーバヘッドが大きい。しかし、常に固定数の命令実行によって型検査が済むので、最悪の場合のオーバヘッドは本方式より小さい。一方、この方式ではクラス階層を表として表すので、場所のオーバヘッドが存在する。

この２つの方式の優劣は、実際のアプリケーションにおいて、本方式でインライン展開された部分でどれだけ型変換が行われるかにかかっている。その評価を、次節で示す。

a) 本方式の型検査テストのコード（null と配列オブジェクトを取らない場合）

```
   lwz   r1,TIBOffset(RHS)       // 検査される RHS オブジェクトから型情報を取り出す
   cmpi  r1,LHSTIBAddress        // キャスト先の型情報と比較する
   b     eq,OK                   // 比較結果が同じであれば OK
   lwz   r1,ClassOffset(r1)      // 型情報からクラス情報を取り出す
   lwz   r1,LastsuccOffset(r1)   // 最後に型検査に成功したクラス情報を取り出す
   cmpi  r1,LHSClassAddress      // キャスト先のクラス情報と比較する
   b     ne,callTICRuntime       // 比較結果が異なれば runtime を呼ぶ
OK:
   ...

callTICruntime:
   blr   TICRuntime              // 実行時ルーチンで型検査を行う
   b     OK
```

b) Jikes RVM の通常オブジェクトの型検査テストのコード

```
   lwz   r1,TableOffsets(r1)     // クラス階層の表のアドレスを取り出す
   lwz   r2,LengthOffset(r1)     // クラス階層の表の長さを取り出す
   cmpi  r2,LHSDepth             // クラス階層の表の長さとキャスト先クラスの階層の深さを比較
   b     ge,ThrowException       // 表の長さが階層の深さより大きいならば、例外を発生する
   lwz   r1,LHSDepth*2(r1)       // クラス階層表から対応する ID を取り出す
   cmpi  r1,LHSid                // キャスト先のクラスの ID と比べる
   b     ne,ThrowException       // 異なるならば、例外を発生する
```

図 4.6: 型検査のネイティブコード

4.4 実験結果

本節では、型検査をインライン展開した場合の性能向上に対する効果について述べる。実験のために、IBM Developers Kit, Java Technology Edition, Version 1.3.1 の Just-In-Time コンパイラ[43]に、本方式を実装した。型検査の統計に関する実験は、PowerPC プラットフォーム上で行った。性能に関する実験は、PowerPC、IA-64 と異なる２つのプラットフォームで行った。

PowerPC では、IBM Developers Kit for AIX, Java Technology Edition, Version 1.3.1 の Just-In-Time コンパイラを用いて、IBM 社の RS/6000 44P モデル 170（POWER3-II 400MHz、メモリ 768MB）に AIX 4.3.3 を使用して行った。IA-64 では、IBM Developers Kit for Windows on Itanium, Java Technology Edition, Version 1.3.1 の Just-In-Time コンパイラを用いて、IBM 社の IntelliStation Z Pro モデル 6894（Itanium 800MHz x 2、メモリ 2GB）に Windows XP Advanced Server を使用して行った。

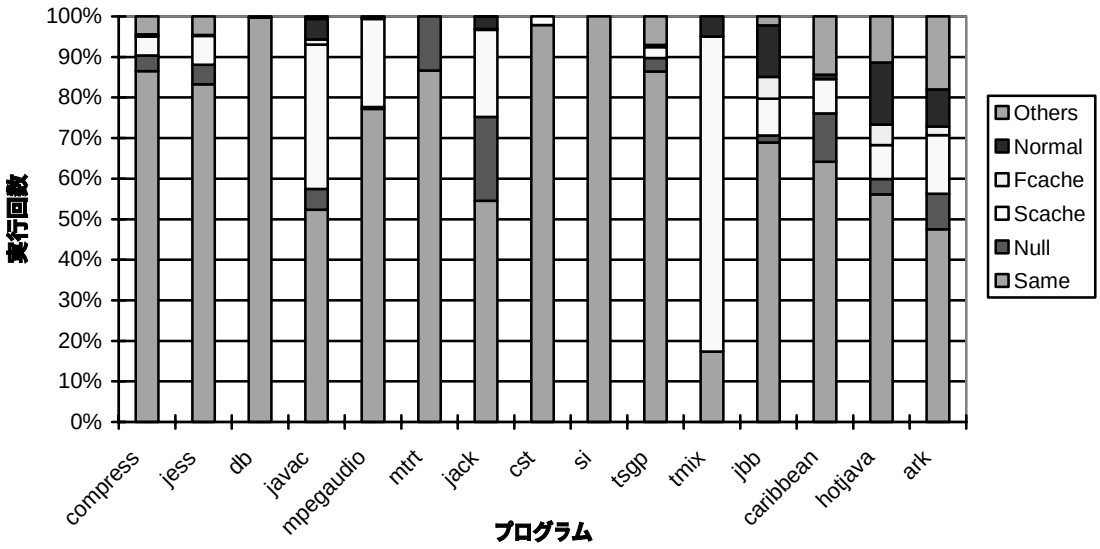
対象とするプログラムには、SPECjvm98[6]に含まれる７つのプログラム（compress、jess、db、javac、mpegaudio、mtrt、jack）と、その候補になった４つのプログラム（cst、si、tsgp、tmix）、SPECjbb2000[94]、エージェント構築環境の caribbean[99]、Web ブラウザの HotJava[100]、ワードプロセッサの一太郎 ark[101]、を用いた。この処理系には、頻繁に実行されるメソッドだけをコンパイルする選択コンパイル機能が実装されているが、今回の実験では全てのメソッドをコンパイルする設定で行った。

表 4.1に、それぞれのプログラムの説明と、最適化を適用しない場合の実行時の型検査実行回数を示した。これより、compress、mpeguaudio、tsgp、caribbean の型検査の実行回数が比較的少ないことが分かる。

表 4.1: ベンチマークプログラムの説明と、最適化を行わない場合の実行時の型検査実行回数

ベンチマーク	型検査回数	プログラムの説明
compress	4247	LZW 法を用いた圧縮・展開プログラム。SPECjvm98 の１つ。size=100 で実行。
jess	29171502	NASA の CLIPS エキスパートシステム。SPECjvm98 の１つ。size=100 で実行。
db	51077153	データ管理プログラム。SPECjvm98 の１つ。size=100 で実行。
javac	11954959	JDK 1.0.2 の java class ファイルを生成するコンパイラ。SPECjvm98 の１つ。size=100 で実行。
mpegaudio	54980	MP3 のデータを伸長するプログラム。SPECjvm98 の１つ。size=100 で実行。
mtrt	2238231	２つのスレッドが走るレイトレーシングプログラム。SPECjvm98 の１つ。size=100 で実行。
jack	6420397	パーサジェネレータ。SPECjvm98 の１つ。size=100 で実行。
cst	50722340	標準ライブラリが用意するデータ構造を操作する。SPECjvm98 の候補だった。size=100 で実行。
si	51077153	最小限の制御構造を持つインタプリタ。SPECjvm98 の候補だった。size=100 で実行。
tsgp	4330	遺伝プログラムミングを用いて問題を解く。SPECjvm98 の候補だった。size=100 で実行。
tmix	3935181	いくつかの単純なプログラムをマルチスレッドで実行する。SPECjvm98 の候補だった。size=100 で実行。
jbb	3829257	三層モデルを用いてトランザクション処理を行う。SPECjbb2000。Warehouse=1 で実行。
caribbean	16581	エージェント開発環境。Ping-pong を行うプログラム例を実行。
hotjava	281976	Web ブラウザを起動し、ページを閲覧し終了する。
ark	311353	ワードプロセッサである一太郎 ark を、起動終了する。

図 4.7に、最適化を適用しない場合、実行時の型検査が行われた際のオブジェクトの種類の分布を示す。Same、Null、Scache の場合がインライン展開されたコードで処理される。他の場合は、C 言語で書かれた実行時ルーチンで処理される。全体で、Same、Null、Scache を合わせた場合が 91%を占め、インライン展開されたコードで処理される。従って、本方式でも型変換が高速化されることが期待される。



Same: 検査したオブジェクトのクラス型と、左辺の型が等しい。
Null: 検査したオブジェクトが null である。
Scache: 検査したオブジェクトのクラス型が、最後に型検査が成功した場合の左辺に等しい。
Fcache: 検査したオブジェクトのクラス型が、最後に型検査が失敗した場合の左辺に等しい。
Normal: クラス階層を辿って型検査を行った場合。
Others: 配列型やインタフェースのクラス型など、その他の型検査の場合。

図 4.7: 実行時型検査を行うオブジェクトの種類

本方式を適用した際の性能向上を、PowerPC の場合を図 4.8に、IA-64 の場合を図 4.9に示す。ここでは、対話操作を必要とする caribbean、hotjava、一太郎 ark は実行速度を測ることが難しいので対象から除いた。インライン展開しない場合を 1 として、インライン展開のみを適用した場合、局所クラス解析によって型検査の除去も適用した場合、局所クラス解析の結果を用いて不要なインライン展開を抑制も適用した場合、の結果を示した。さらに Cohen のアルゴリズム[97]を通常のオブジェクトの型検査に対してのみ適用し、表の長さのテスト無しで実装した場合の結果も合わせて示した。

インライン展開のみを適用した場合、性能向上の相乗平均で PowerPC では 33%、IA-64 では 16%性能が向上する。局所クラス解析による型検査の除去を適用すると、PowerPC では 36%、IA-64 では 16%、とやや性能が改善される。不要なインライン展開の抑制も適用した場合、PowerPC では 36%、IA-64 では 14%、と適用しない場合とほぼ同程度の性能向上である。Cohen のアルゴリズムを適用した場合、PowerPC では 35%、IA-64 では 17%の性能向上となり、本章で提案したインライン展開を適用した場合と同程度である。実際の Cohen のアルゴリズムの実装においては、表の長さに関するテストが常に行われるので、今回の実験より速度が低下すると予想される。

型検査の高速化は、PowerPC では、jess、db、javac、mtrt、jack、cst、si において効果がみられた。IA-64 では、jess、db、javac、mtrt、jack、cst、si に加え tmix において効果が見られた。PowerPC と IA-64 を比べると、PowerPC の方が性能向上が大きい。これは、コンパイルされたネイティブコードから C 言語で書かれた型検査のための実行時ルーチンを呼ぶオーバヘッドが

PowerPC の方が大きいため、と考えられる。PowerPC では C の関数で使われる不揮発レジスタを明示的に待避・復帰する必要があるが、IA-64 ではレジスタスタックエンジンが自動的にレジスタを待避・復帰するため、通常は IA-64 の方式がオーバヘッドが少ない。

これらの実験結果より、Cohen のアルゴリズムを適用した場合でも、PowerPC と IA-64 において本方式と同等程度またはそれ以下の性能しか得ることができないことが分かる。

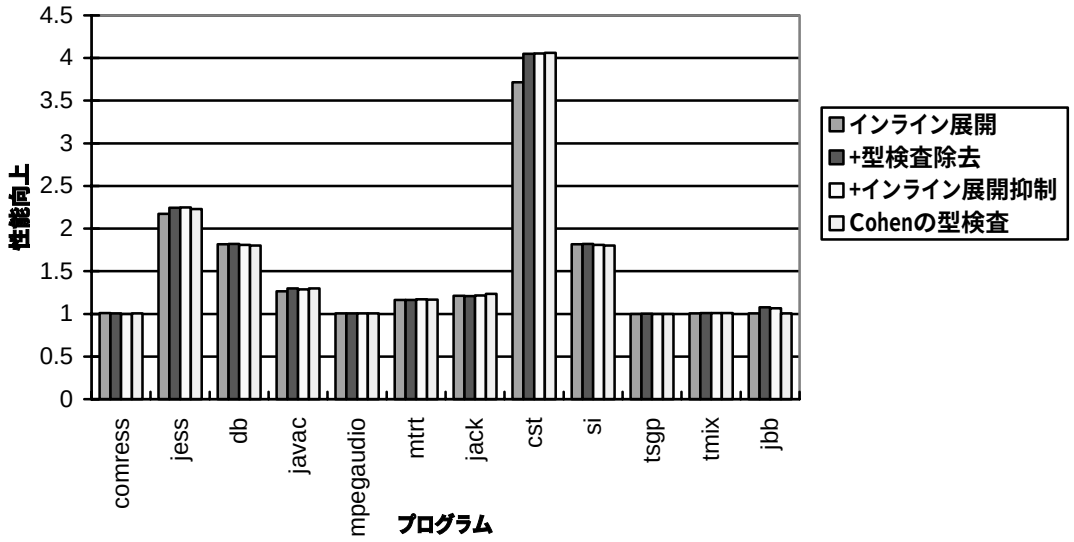


図 4.8: 型検査の高速化による性能向上（PowerPC）

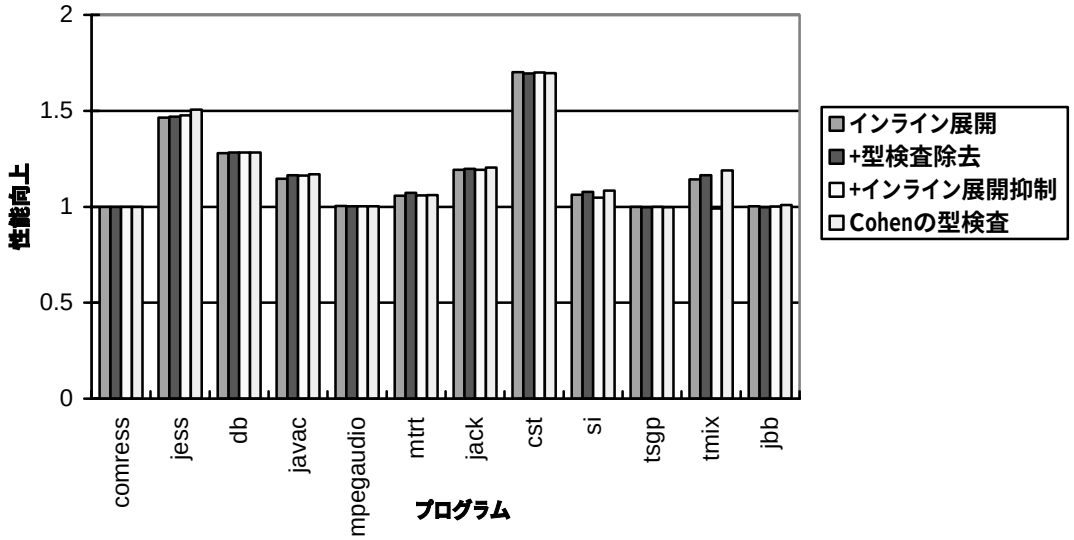


図 4.9: 型検査の高速化による性能向上（IA-64）

4.5 まとめ

本章では、Java 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、オブジェクトの型検査の高速化手法を提案した。本手法では、アプリケーションの挙動から得られた型検査の処理のうち頻繁に実行される部分を、ネイティブコードにインライン展開する。

この手法によって、SPECjvm98、SPECjbb2000 等のプログラムにおいて、型検査の高速化手法を適用しない場合に比べて、平均して PowerPC では 36%、IA-64 では 14%の性能向上が得られた。また、この方法は従来の方法と比較して実装が簡単であるが、従来の方法と同等またはそれ以上の性能向上が得られることを実験結果より示した。

第5章 例外検査の高速化

5.1 はじめに

Java 言語では、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、の際等に例外検査を行うことが言語仕様において規定されている。これらの例外検査において、null を伴うアクセスであれば `NullPointerException` を発生し、配列のインデックスが不正な値であれば `ArrayOutOfBoundsException` を発生する。これらの検査によって許されないメモリへのアクセスは発生しないので、プログラムの安全性を保証する Java 言語においてプログラムを安全に実行するために例外検査は重要な機能である。しかし、C 言語や C++言語等のプログラムの安全性が保証されていない言語に比べて、例外検査が実行時オーバヘッドとなり実行速度を減少させる要因となる。

従来、データフロー解析を用いてコンパイル時に冗長な例外検査を削減する最適化が提案されている[31][45][102]。これらの最適化によって例外検査を減らすことはできるが、多くの場合ゼロにすることはできず、実行時に例外検査は残ってしまう。さらに、実行時にオブジェクトの値が０であるかどうかの検査を高速に処理するために、０番地から始まる最初のページ（一般に 4KB 程）（以降、０ページと呼ぶ）を読み出し不可にしているオペレーティングシステム（Operating System、OS）のメモリ保護機能を用いる方法が提案されている[49][58]。しかし、投機的なロードを行う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[103]。この OS を用いた環境では、OS のメモリ保護機能を用いた高速化手法を用いることができない。

本章では、発生する例外の種類を例外検査のための命令列に埋め込むことによって、例外が発生せずに通常に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、この方法を用いて、メモリ保護機能を用いた高速化手法を使用できない場合でも、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に利用し、例外検査を高速化する。さらに、本章で提案した手法を Java Just-In-Time コンパイラに実装し評価を示す。実験の結果、コンパイル時に冗長な例外検査除去の最適化を適用しない場合には、整数命令を同時実行できないプロセッサでは最高 70%の性能向上が、整数命令を同時実行可能なプロセッサでは最高 33%の性能向上が得られた。また、コンパイル時に全ての冗長な例外検査除去の最適化を適用した場合でも、整数命令を同時実行可能なプロセッサでは最高 6%の性能向上が、整数命令を同時実行できないプロセッサでは最高 14%の性能向上が得られた。

以下、5.2節では関連研究を述べる。5.3節では、例外の種類を命令に埋め込むことによって例外検査を高速化する手法について述べる。5.4節では、プログラムを用いて本方式を評価した結

果を示す。5.5節では、本章のまとめを述べる。

5.2 関連研究

本節では、コンパイル時の最適化によって冗長な例外検査を除去する方法と、実行時に OS のメモリ保護機能を用いて例外検査のオーバヘッドを削減する方法について示す。

5.2.1 コンパイル時に冗長な例外検査を除去する方法

あるオペランドに対して一度 null 値をとるかどうかの例外検査が行われた後、その後そのオペランドに対して代入が行われることなく、もう一度同一オペランドに対して例外検査が行われる場合、後者の例外検査は冗長であるので除去することができる。この状況は、前向きのデータフロー方程式を解くことによって検出可能である[31]。さらに、部分冗長除去を用いて、null 値をとるかどうかの例外検査を後方移動しループ外や実行頻度の低いパスに置くことによって、例外検査の実行回数を削減する方法も提案されている[45]。

また、ループを複製し片方のループでは配列のインデックスに関する例外検査を削減したコードを生成し、実行時の値に基づいてどちらのループを実行するか決定する、ループバージョニングも提案されている[43][105]。この方式では、ループ内で使用される配列の範囲とループ上限値の関係式を作り、実行時にその関係式を評価し、配列の使用される範囲が配列長内に収まることが分かればループ内で例外は発生しないことが分かるので、例外検査を削減したループを実行する。例外が発生する可能性がある場合は、元のループを実行する。生成される疑似コードを、図 5.1に示す。

```
if ((array != NULL) && (0 <= start) && (end <= array.length)) {
    /* safe loop (eliminate all array index exception checks for array[] */
    for (i = start; i < end; i++) {
        array[i] = array[i] + 1;
    }
} else {
    /* unsafe loop (original loop with array index exception checks */
}
```

図 5.1: ループバージョニングの疑似コード[43]

5.2.2 例外検査の実行時オーバヘッドを削減する方法

従来提案されているコンパイラの最適化によって冗長な例外検査を削減する方法を5.2.1節で述べたが、全ての例外検査を除去できるわけではない。従って、例外検査の実行時オーバヘッド削減も重要である。

オペランドが null 値をとるかどうかの例外検査を OS のメモリ保護機能を利用し、検査のためのコード生成を省略することによってネイティブコードを高速化する手法が提案されている[49][58]。Java プログラムにおいて null を表す 0 番地がアクセスされた場合、0 ページへの読み書きを禁止する OS のメモリ保護機能によって、例外処理ハンドラへ制御が移る。例外処理ハンドラでは、不正なアクセスが発生した命令の番地を調べ、その番地が JIT コンパイラによってコ

ンパイルされたネイティブコード内であったら、NullPointerException を発生させる。この方法によって、ネイティブコード内で例外検査のために明示的な比較命令と分岐命令が実行されることがなくなる。さらに、オブジェクトへのアクセスの多くは例外を発生しないため、実行時間が短縮される。この手法は、一般には配列のインデックス検査に対しては適用できない。この理由は以下の通りである。OS のメモリ保護機能は 0 ページと、OS がメモリを確保していないページに対して設定されている。不定な範囲をアクセスする可能性がある配列のインデックスによって得られたメモリアドレスが正しいインデックスの範囲を逸脱しても、別の配列のために確保されたメモリを指し不正に読み書きができて、メモリ保護機能が働かずに実行を続けてしまう可能性があるためである。

5.3 例外検査命令列への例外種類の埋込み

本節では、例外検査において例外が発生せずに通常に実行される部分に生成されるコードを最小限にすることによって、例外検査を高速化する方法を示す。また、この方法を用いて、メモリ保護機能を用いた高速化手法を使用できない場合でも、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利用し、例外検査を高速化する手法も示す。

一般に複数の種類の例外検査を行うコードは、例外検査と例外の種類毎の飛び先がコンパイラによって作られ、ランタイム共通の例外処理ハンドラを持ち、図 5.2のコードが生成される。共通例外ハンドラでは、レジスタにセットされた例外の種類に従って、必要な処理を行う。

```
    例外条件 1 のチェック
    条件 1 が成り立ったら、ERROR_JMP1 へ飛ぶ
    ...

    例外条件 2 のチェック
    条件 2 が成り立ったら、ERROR_JMP2 へ飛ぶ
    ...

    例外条件 1 のチェック
    条件 1 が成り立ったら、ERROR_JMP1 へ飛ぶ
    ...


ERROR_JMP1:
    例外種類 1 をレジスタへセット
    共通例外ハンドラへ飛ぶ

ERROR_JMP2:
    例外種類 2 をレジスタへセット
    共通例外ハンドラへ飛ぶ

共通例外ハンドラ:
    レジスタの値に従って例外処理を行う。
```

図 5.2: 一般的な例外検査のコード

また、PowerPC アーキテクチャや SPARC アーキテクチャでは、指定された条件が成立しない場合を高速に実行し、条件が成立した場合プロセッサが持つシステムトラップベクタへ分岐した後 OS が用意するハンドラへ制御を移す命令が用意されている。PowerPC では tw 命令、SPARC では tcxx 命令である。例えば、r31 が 0 であるかどうかは、それぞれ以下の命令列で高

速に検査可能である。

```
PowerPC:                          SPARC:
    twi   r31, 0                        cmp   %g31, %g0
                                        teq   16
```

これらの命令を用いた場合、検査する例外の種類が異なっても、OS が用意する同一のハンドラへ制御が移るので、分岐前に例外の種類を指定する必要がある。従って、図 5.3のコードが生成される。共通例外ハンドラでは、レジスタにセットされた例外の種類に従って、必要な処理を行う。

```
        例外種類１をレジスタにセット
        例外条件１のチェック
        条件１が成り立ったら、共通例外ハンドラへ飛ぶ
        ...

        例外種類２をレジスタにセット
        例外条件２のチェック
        条件２が成り立ったら、共通例外ハンドラへ飛ぶ
        ...

        例外種類１をレジスタにセット
        例外条件１のチェック
        条件１が成り立ったら、共通例外ハンドラへ飛ぶ
        ...

    共通例外ハンドラ:
        レジスタの値に従って例外処理を行う。
```

図 5.3: プロセッサが持つ特別な命令を用いた例外検査のコード

図 5.2の場合、例外の種類毎に飛び先がまとめられ例外の種類を表す値をレジスタにセットしてから、例外共通ハンドラへ飛んでいる。もし、通常実行のパスにオーバヘッドにおいてオーバヘッド無しに例外の種類を設定して例外ハンドラへ飛ぶことができるならば、例外の種類毎の飛び先を無くすことができ、コード量を小さくすることができる。

図 5.3の場合は、通常の実行のパスで例外の種類を表す値をレジスタにセットしてから、例外条件の比較と分岐を行っている。上記と同様に、もし通常実行のパスにおいてオーバヘッド無しに例外の種類を設定して例外ハンドラへ飛ぶことができるならば、通常の実行パスの命令量が減るので、実行性能の向上が期待できる。

以下の手法によって、これらの要求は実現可能である。例外の種類を表す値をレジスタにセットすること無く例外ハンドラで例外の種類を判断するために、例外ハンドラへ分岐する命令とその検査を行う命令が例外の種類に対応して一意なビット列を含むことを保証して、コンパイラが命令列を生成する。実行時に例外ハンドラへ制御が移った場合、例外ハンドラに分岐してきた命令のアドレスに存在する分岐命令とそれに対応する比較命令を読み込んでデコードし、命令のビット列から発生した例外の種類を決定する。

本手法を PowerPC の命令列に対して適用する例を図 5.4に示す。PowerPC アーキテクチャでは、比較とハンドラへの分岐を同時に行う `trap` 命令が用意されている。この命令は分岐が不成立の時は１サイクルで実行可能である。Java 言語において例外検査は頻繁に行われるが実際に例外が発生することは少ないので、例外検査に用いるには適当な命令である。ソースプログラ

ムに対して生成されるネイティブコードの最初の命令は、r5 が null であるかどうかの例外検査を行うための trap 命令である。５番目の命令は割り算の除数である r7 が０であるかどうかの例外検査を行うための trap 命令である。どちらもレジスタの値が０であるかどうかを調べるための trap 命令であるが、それぞれの例外の種類に対応して一意なビット列を含むことを保証するため、r = 0 と $r <_{logical} 1$ という「r が０であるかを調べる」同じ条件を表す、異なる２つの命令を用いている。２番目の命令は、配列インデックスの例外検査を行うための trap 命令である。

実行時に、tw/twi 命令に記述された条件が成立した場合、ハンドラに渡される実行時コンテキストから例外が発生したアドレスを指す IAR のアドレス[iii]の内容を調べて、発生した例外の種類を検出する。例えば、r7=0 でプログラムが実行されたとき、「twi llt, r7, 1」が条件を満たし、例外ハンドラへ実行を移す。例外ハンドラの中で、例外が発生したアドレス iar を取得し、iar にある命令を調べる。この場合、IS_TRAPI_LLT(iar)が成り立つので、検出された例外の種類に従って ARITHMETIC_EXCEPTION()の処理を行う。

実際 AIX では、コンパイラの最適化による投機的ロードを許すために０ページを参照可能に設定してある[103]ので、明示的に null であるかどうかの例外検査を行う必要がある。従って、この例で示した高速な比較分岐命令である trap 命令を使用した場合の高速化手法は有用である。

ソースコード

```
int foo(int[] a, int i, int j) {
     return r = a[i] / j;
}
```

生成されたネイティブコード

```
; r4 : 配列アクセスインデックス(i)  r5: 配列の基底アドレス(a[])
; r6 : 配列の大きさ(a.length)     r7: 除数(j)

twi[1]   EQ, r5, 0                      // null ポインタ検査
tw[1]    LLE, r6, r4                    // 配列インデックス検査
mulli  r4, r4, 2
lwzx   r3, r4(r5)                       // 配列要素へのアクセス
twi[1]   LLT, r7, 1                     // 除数の０検査
divi   r3, r3, r7
```

[1]tw/twi 命令は、オペランドに記述された条件が成立した場合、OS で指定されたハンドラへ分岐する、比較と分岐を同時に行う命令。

例外ハンドラ

```
void TrapHandler(struct context *cp)
{
     int *iar = cp->IAR;                // 例外が発生したアドレスの取得
     if IS_TRAPI_EQ(iar) {              // 命令は‘twi EQ’であるか ?
          process_NULLPOINTER_EXCEPTION()
     } else if IS_TRAP_LT(iar) {        // 命令は‘tw LT’であるか ?
          process_ARRAYOUTOFINDEX_EXCEPTION()
     } else if IS_TRAPI_LLT(iar) {      // 命令は‘twi LLT’であるか ?
          process_ARITHMETIC_EXCEPTION()
     }
}
```

図 5.4: 高速な例外検査の例

[iii] tw/twi 命令では、１命令で比較と分岐を行っているので IAR のアドレスだけを調べればよいが、比較と分岐を２命令で行う場合、分岐命令に対応する比較命令を見つける必要がある。

5.4 実験結果

本節では、ハードウェアに用意された高速な条件分岐命令に例外の種類を埋め込んだ場合の性能向上の効果について述べる。実験のために、IBM Developers Kit, Java Technology Edition, Version 1.3.1 の Just-In-Time コンパイラ[43]に、本方式を実装した。本実験は、０ページが読み込み可である PowerPC プラットフォームの AIX 環境で行った。

コンパイラは、IBM Developers Kit for AIX, Java Technology Edition, Version 1.3.1 の Just-In-Time コンパイラを用いた。対象とするプログラムには、Symantec Benchmark のうち５つのプログラム、Java Grande Benchmark[104]のうち２つのプログラム、SPECjvm98[6]のうち３つのプログラムを用いた。表 5.1に、それぞれのプログラムの説明を示す。これらのプログラムは、よく用いられるベンチマークの中でも例外検査の多いプログラムである。この処理系には、頻繁に実行されるメソッドだけをコンパイルする選択コンパイル機能が実装されているが、今回の実験では全てのメソッドをコンパイルする設定で行った。

表 5.1: ベンチマークプログラムの説明

ベンチマーク	プログラムの説明
BubSort	バブルソートを実行する。Symantec Benchmark に含まれている。
BiSort	双方向バブルソートを実行する。Symantec Benchmark に含まれている。
Sieve	素数を求める。Symantec Benchmark に含まれている。
Hanoi	ハノイの塔問題を解く。Symantec Benchmark に含まれている。
Dhrystone	Dhrystone ベンチマークを実行する。Symantec Benchmark の１つ。
LU	LU 分解を実行する。Java Grande Benchmark の１つ。SizeB を実行。
HeapSort	ヒープソートを実行する。Java Grande Benchmark の１つ。SizeB を実行。
compress	LZW 法を用いた圧縮・展開プログラム。SPECjvm98 の１つ。Size=100 で実行。
mpegaudio	MP3 のデータを伸長するプログラム。SPECjvm98 の１つ。Size=100 で実行。
mtrt	２つのスレッドが走るレイトレーシングプログラム。SPECjvm98 の１つ。Size=100 で実行。

また、プロセッサの並列度が与える影響を調べるため、PowerPC アーキテクチャで、RS64II、PowerPC 604e、POWER3、と異なる３つのプロセッサで実験を行った。それぞれのプロセッサの構成を表 5.2に示す。全てのプロセッサにおいて `trap` 命令は１サイクルで実行可能である。

表 5.2: プロセッサの構成

プロセッサ名	総ユニット数	整数ユニット数		trap 命令を実行するユニット	整数命令と trap 命令の並列実行
		単純	複雑		
RS64II	4	1	1	単純整数ユニット	一部可
PowerPC604e	6	2	1	単純整数ユニット	可
POWER3	8	2	1	単純整数ユニット	可

さらに、実行速度は例外検査の命令の有無に大きく左右されるため、冗長な例外検査除去のために適用する最適化を、以下の５通りに変化させた。

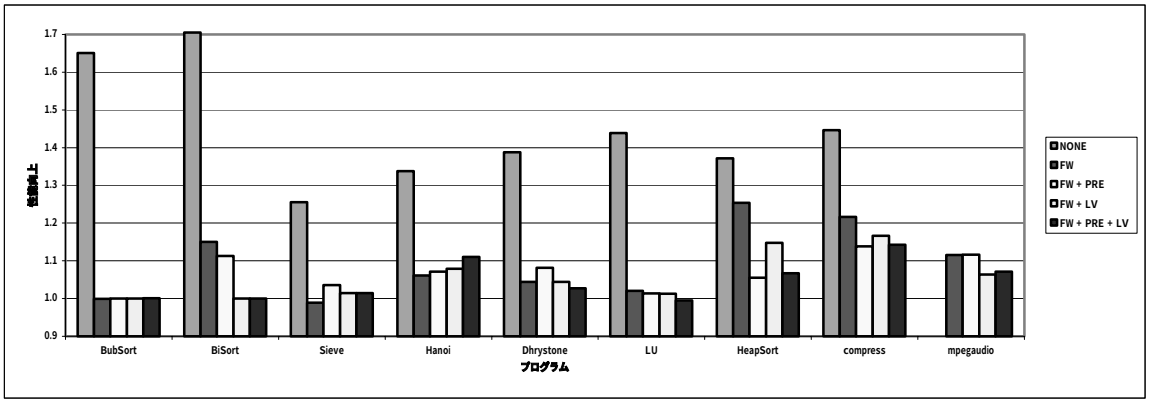
- 例外検査除去の最適化適用無し
- 前方データフロー解析[31]を用いた例外検査除去
- 前方データフロー解析と部分冗長除去[45]を用いた例外検査除去
- 前方データフロー解析とループバージョニング[43]を用いた例外検査除去
- 前方データフロー解析と部分冗長除去とループバージョニングを用いた例外検査除去

本方式を適用して、プロセッサ毎に最適化の適用を変化させた場合の性能向上を図 5.5に示す。例外の種類を明示的にレジスタに代入してハードウェアに用意された高速な条件分岐命令（`trap` 命令）を使用して例外検査を行った場合を１としている。a)が RS64II における結果、b)が PowerPC 604e における結果、c)が POWER3 における結果、である。

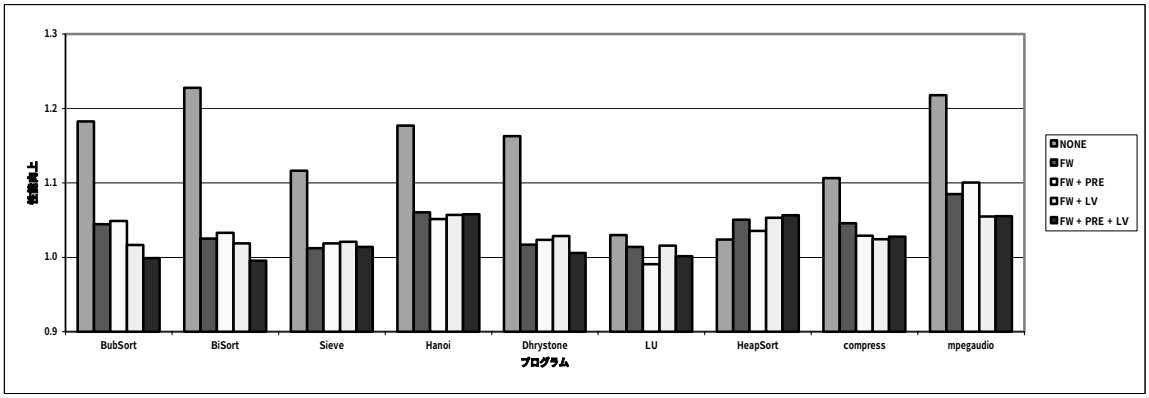
これらの結果から、例外検査除去の最適化を適用しない場合の効果が非常に大きいことが分かる。RS64II で 18〜70%、PowerPC 604e で 2〜23%、一番並列度の高い POWER3 でも 6〜33%の性能向上が得られている。これは、Java 言語では冗長な例外検査除去を行わない場合には非常に例外検査が多いので、実行される総命令数が増加することが原因と思われる。

従来提案されている前方データフロー解析を用いた冗長な例外検査除去だけを適用した場合でも、RS64II では-1%〜25%の性能向上が得られている。さらに、PowerPC 604e では 1%〜8%の性能向上が、POWER3 では 2%〜11%の性能向上が得られる。これは、前方データフロー解析を用いた冗長な例外検査除去では、十分に例外検査を除去出来なかったためと思われる。部分冗長除去やループバージョニングを用いた例外検査除去は命令移動を伴うので、前方データフロー解析を用いた命令除去より、コンパイル時間がかかる最適化である。従って、adaptive compilation を適用するときに要求される短いコンパイル時間でコンパイルを行う軽い JIT コンパイラでは、前方データフロー解析を用いた例外検査除去しか適用できないことも考えられる。この場合、本方式はより有効であると考えられる。

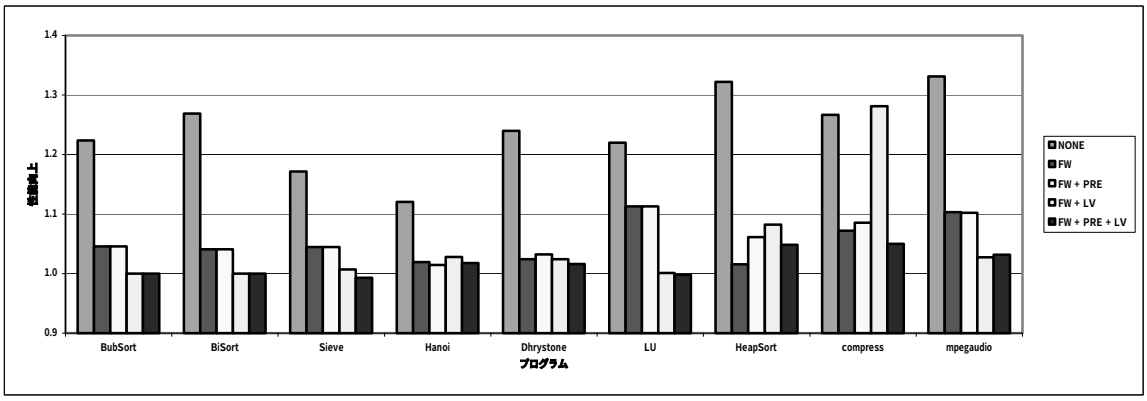
また、プロセッサの並列度が低く整数命令を同時実行できない RS64II では、本方式によってレジスタへの代入命令を生成しないことによる性能向上がより大きい。全ての例外検査除去の最適化を適用した場合でも、SPECjvm98 の compress や mpegaudio においては、整数命令を同時実行できる PowerPC 604e と POWER3 では 3%〜6%の性能向上だが、整数命令を同時実行できない RS64II では 7%〜14%と他のプロセッサに比べて高い性能向上が得られている。実行したプログラム全体では、PowerPC 604e と POWER3 では-1%〜6%の性能向上だが、RS64II では 0%〜14%の性能向上が得られている。



a) RS64II における性能向上



b) PowerPC 604e における性能向上



c) POWER3 における性能向上

NONE:　例外検査除去の最適化適用無し（a）の mpegaudio はデータ取得不能）。
FW:　前方データフロー解析を用いた例外検査除去。
FW + PRE:　前方データフロー解析と部分冗長除去を用いた例外検査除去。
FW + LV:　前方データフロー解析をループバージョニングを用いた例外検査除去。
FW + PRE + LV:　前方データフロー解析部分冗長除去とループバージョニングを用いた例外検査除去。

図 5.5: 例外検査の高速化による性能向上

表 5.2の RS64II の欄から、RS64II では単純整数命令を同時実行できないことが分かる。従って、`trap` 命令を実行すると例外と無関係の整数加減算等の多くの単純命令の実行が遅らされる。一方、整数命令を同時実行できる PowerPC 604e や POWER3 では、`trap` 命令と例外と無関係の整数命令は並列実行可能である。従って、例外に関係の無い命令の実行が遅らされる可能性の高い RS64II において、本方式による効果が高い。

5.5 まとめ

本章では、Java 言語のプログラムの安全性を保証するための重要な機能の１つである、例外検査の高速化手法を示した。本手法では、発生する例外の種類を命令に埋め込むことによって、例外が発生せずに通常に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示した。また、OS のメモリ保護機能を用いた高速化手法を使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利用する際に、本手法を用いて例外検査をさらに高速化した。

冗長な例外検査除去の最適化を適用しない場合には、整数命令を同時実行できないプロセッサでは最高 70%の性能向上が、整数命令を同時実行可能なプロセッサでは最高 33%の性能向上が得られた。また、全ての冗長な例外検査除去の最適化を適用した場合でも、整数命令を同時実行可能なプロセッサでは最高 6%の性能向上が、整数命令を同時実行できないプロセッサでは最高 14%の性能向上が得られた。

第6章 安全な参照を保証した命令間順序制約緩和

6.1 はじめに

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、その性質を保証するために実行時に行われる数々の検査が導入される。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、の際に行われるアドレスの例外検査がその１つである。これらの検査は、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。だが、例外検査を行うこれらの命令は、不正なアドレスを伴う可能性のあるメモリアクセス命令等の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引き起こし、命令の並べ替えを伴う最適化の効率を低下させる。本章では、Java 言語等の型安全な言語において、DAG 表現を用いてハードウェアに関する例外を発生させる命令に対して効率的に投機命令移動を適用することによって、順序制約を解消する方法[106][107]を述べる。

型安全性が成り立つならば、不正なメモリアクセスを起こさない、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、という性質を保証する。この性質は、セキュリティの面からも近年急速に重要性が増している。Java 言語では型安全性が保証され、正しくないアドレスのメモリをアクセスする等の、安全でない処理に伴って発生する全ての例外（例えば、`NullPointerException` や `ArrayOutOfBoundsException`）はプログラムによって捕捉可能である。例外を発生する可能性がある命令は、potentially excepting instruction (PEI) [108]と呼ばれる。従って、ハードウェアが発生する例外と、Java 言語によって定義されるソフトウェアが発生する例外の２種類がある。不正なアドレスを伴ったメモリアクセス命令、アクセス例外などハードウェア例外によってプログラムを異常に終了させる可能性がある命令（以下、*H-PEI* <hardware-initiated potentially exception instruction>と呼ぶ）の前に、ソフトウェアによる実行時例外検査によってプログラムが異常終了しないことを保証する命令（以下、*S-PEI* <software-initiated potentially exception instruction>と呼ぶ）を挿入して、ハードウェア例外が発生する状況でもプログラムをソフトウェア的に安全に停止できなければならない。そのため、S-PEI の後に H-PEI を実行する、という命令間依存（以下、この命令間依存を例外依存と呼ぶ）が発生する。例外依存は、命令移動を伴う最適化の妨げとなる。

本章では、Java 言語等の型安全な言語において、投機的命令移動を用いて H-PEI と S-PEI の間の例外依存を効率よく除去する方法を提案する。この結果、S-PEI を越えた H-PEI の命令移動が可能になる。投機的命令移動[109]は、依存を無視して実行の有無が決定する以前の位置へ命令を移動する手法である。その結果、移動された命令が実行の有無が決定する前に先行的に実

行される、投機的実行が行われる。投機的命令移動については、２種類の手法がよく知られている。１つは、分岐命令と後続命令の間の制御依存を越えて命令を移動する、制御依存を越えた投機的命令移動 [108][109][110][111][112][113][114][115]である。もう一つは、メモリストア命令と後続のメモリロード命令のアドレスが異なると仮定して命令間のデータ依存を越えて移動する、データ依存を越えた投機的命令移動 [116][111][114][115]である。図 6.1に、制御依存を越えた投機的実行の例を示す。投機的実行の際、実行されることのない不正な値を伴って命令が実行された結果（図 6.1では `p=0` の時の`*p` の参照）、例外が発生する可能性がある。この例外は投機的実行が行われなければ発生しなかったものであり、`*p` の参照の時点でプログラマに例外の発生を伝えてはならない。このために、システムによって提供される例外発生を抑制した投機的命令を用いて投機的実行を行う。さらに、投機的実行の成否を調べる命令によって（図 6.1では `check` 命令）、失敗した場合には復旧コード（図 6.1では `Recovery` 以下）を実行して、正しい値を再生成する仕組みも必要である。

投機的移動前

```
if (p != 0) {
  s = *p;
  t = s + 1;
}
...
```

投機的移動後

```
s = *p; // 投機的命令による参照
t = s + 1;
if (p != 0) {
  check t, Recover
}
...
Recover:
s = *p;
t = s + 1;
```

図 6.1: 制御依存を越えた投機的実行の例

投機的実行を用いて例外依存を除去する手法として、S-PEI を比較命令と分岐命令に分解し、例外依存を S-PEI と H-PEI 間の制御依存で表す方法が提案されている。この変形されたコードに対して、スーパーブロックスケジューリング[111]を用いて制御依存を越えて投機的命令移動を適用し、S-PEI と H-PEI 間の例外依存を除去する[117]。この方法では移動前後の各ブロックの実行時間を見積もることはできるが、ブロック間全体の実行時間を見積もることは簡単ではない。従って、投機的命令移動による例外依存除去の効果が正確に見積もりにくい。また、分岐命令が数多く生成されブロック数が増えるので、ブロック内の平均命令数が減少し、ブロック内最適化の効率が低下する問題が生じる。

本手法では、従来手法における S-PEI から比較命令と分岐命令への分解を適用することなく、中間表現において S-PEI を１つの命令として扱う。中間表現として、SSA 形式による DAG 表現を用いる。ここで DAG は命令を節、命令間のデータ依存関係・制御依存関係・例外依存関係等を辺、として構成されるグラフである。このとき、S-PEI と関連する H-PEI 間の例外依存を、DAG 表現上でブロック内の S-PEI と H-PEI 間の辺によって表現する。この辺を、例外依存辺と呼ぶ。この枠組みの上で、H-PEI に対して投機的命令移動を適用し、例外依存辺を切断することによって、例外依存による制約を除去する。この制御依存を越えた投機的命令移動と例外依存を越えた投機的命令移動を区別する本手法によって、以下の利点が得られる。

- 大域最適化であるスーパーブロックスケジューリングを適用することなく、例外に関する投機的命令移動ができる。
- 例外依存辺を用いて投機移動適用後の実行時間の短縮を正確に見積もり可能である。
- S-PEI が実行される際に例外が発生することは少ないので、S-PEI の後続命令はほぼ実行されると見なして投機的移動を適用できる。
- 例外依存によってブロックが分割されることがないので、コンパイラの内部表現が使用する記憶領域を節約できる。
- ブロック内の命令数の増加によって、投機的命令移動を含む様々な最適化を効果的に適用できる。

また、本章では投機的命令移動を行う際に考慮すべき、投機的移動を行う命令列の選択、復旧コードの生成、についても示す。この命令列の選択方法は、復旧コードが正しく実行可能で、SSA 形式から通常形式に変換する際に余分なレジスタ間移動命令を生成しない。また復旧コードの生成方法は、メインコードのレジスタ割り付けやコードスケジューリングに与える影響を最小限にする。

さらに、本方式を Java Just-In-Time コンパイラに実装し、いくつかのプログラムを IA-64[64]プロセッサ上で実行して得られた結果を示す。小規模なプログラムでは最大 31%の性能向上を、Java Grande Benchmark では最大 10%（平均 1.7%）の性能向上を、SPECjvm98 では最大 12%（平均 2.3%）の性能向上が得られた。

以下、6.2節で関連研究を述べ、6.3節で S-PEI と H-PEI 間の制約除去手法について述べる。6.4節で投機的命令移動を適用する際のコンパイル手法について述べる。6.5節では、評価実験によって本手法の効果を示す。6.6節で本章のまとめを述べる。

6.2 関連研究

Ebcioglu ら[118]は、アウト・オブ・オーダのバイナリ変換手法を提案している。この方式では、H-PEI であるロード命令を例外発生を抑制したロード命令に置き換えて制御依存を越えて投機的移動するとともに、ロード先のレジスタの名前を付け替える。さらに、元のロード命令があった位置に、値を元のレジスタに戻す移動命令が挿入される。この方式は、変換時間が短くバイナリ変換には適している。しかし、投機的移動可能な命令がロード命令だけであり、余計なレジスタ間移動命令が挿入される、という欠点がある。

Le[109]は、例外発生を抑制した命令を使わずに投機的命令移動を行う方法を提案している。この方法では、通常のコード内に安全点が生成されるため、例外が発生しない場合のクリティカルパスをのばすことがある。

Ju ら[114]は、本手法と同様に SSA 形式による DAG 表現を用いて、制御依存とデータ依存を越えて投機的命令移動を行う枠組みを提案している。彼らの手法は、C 言語を対象として、型安全な言語で発生する例外を扱っていない。また、実験結果は IA-64 のシミュレータ上で得られた

ものである。

Arnold ら[117]は、Java の例外を VLIW 計算機で扱う手法を提案している。この手法では、S-PEI を比較命令と分岐命令に分解し、スーパーブロックスケジューリングと一般的なパーコレーションを適用している。これらは大域的な最適化であり、コンパイル時間を多く消費する。この実験結果もシミュレータ上で得られたものである。

Gupta ら[119]は、S-PEI を例外検査命令と例外を発生する命令に分解し、例外検査命令間の制約を除去する手法を提案している。本手法は S-PEI と H-PEI 間の制約を除去するので補完的に適用可能である。

6.3 S-PEI と H-PEI 間の例外制約除去

本節では、S-PEI と H-PEI 間の例外制約除去のアルゴリズムを示す。まず、DAG の生成方法について述べる。次に、DAG 表現上での例外依存の制約除去のアルゴリズムについて述べる。最後に、本アルゴリズムを例題プログラムに適用した例を示す。

6.3.1 DAG 上での例外発生命令の表現

本節では、Java バイトコードから例外依存を含む DAG の生成方法を示す。Java バイトコードは、それが占める記憶容量を小さくするために、１命令で複数の処理を行うものがある。例えば、配列から整数要素を読む `iaload` バイトコード命令は、以下の複数の処理を行う。

1. 配列オブジェクトが null ならば `NullPointerException` を発生する。
2. 配列オブジェクトから配列長を読む。
3. 添字が配列範囲外ならば `ArrayOutOfBoundsException` を発生する。
4. 配列要素のアドレスを計算する。
5. 整数要素を配列から読む。

１命令が複数処理を行う形式のままコンパイラの中間表現を生成すると、冗長な処理の除去が効果的にできない、スケジューリングの見積りが正確にできない、等の問題が生じる。従って、１バイトコード命令における複数処理をコンパイラの中間表現では分解してそれぞれに表現する。

図 6.2 a)の Java プログラムは、図 6.2 b)のバイトコードで表現される。このバイトコードを、１命令１処理の４つ組を用いた中間表現に直すと図 6.2 c)となる。**太字の命令**は S-PEI、*斜体の命令*は H-PEI を示す。ここで、`nullcheck` 命令はオペランドの値が null であれば `NullPointerException` を生成する命令、`boundcheck` 命令はオペランドを比較した結果が成立しなければ `ArrayOutOfBoundsException` を生成する命令、`ld4` 命令は 4 バイトの整数値をメモリから読む命令、である。<>で囲まれた命令番号は、S-PEI が依存する H-PEI を示す。

a) サンプルプログラム

```
int foo(int n, int i) {
   int a[] = new int[n];
   return a[i] + 1;
}
```

b) バイトコード

```
iload 1
newarray int
astore 3
aload 3
iload 2
iaload
iconst 1
iadd
ireturn
```

c) 中間表現

```
N1: newarray   t1 = n
N2: nullcheck  t1
N3: ld4        t2 = 0[t1]  <N2>    // 配列長のロード
N4: boundcheck i < t2
N5: add        t3 = t1, 16         // 配列先頭の計算
N6: shiftl     t4 = i, 2
N7: ld4        t5 = t4[t3] <N2,N4>// 配列要素ロード
N8: add        t5 = t5, 1
N9: ret        t5
```

図 6.2: 例題プログラムと中間表現

この中間表現に対して、冗長な例外検査命令除去の最適化を適用する。newarray 命令は処理が正常終了すれば必ず null でない配列オブジェクトが返されるので、N2 の nullcheck 命令は冗長であり除去可能である。次に、この中間表現を SSA 形式に変換する。この中間表現が持つ依存関係を DAG で表現したものが図 6.3 a)である。辺が持つ時間は、命令の実行時間を示す。ld4 命令は実行に 3 サイクルかかり、その他の命令は実行に 1 サイクルかかる、と仮定している。newarray 命令と boundcheck 命令の間に、例外発生の順序を守るために例外依存を表す辺を張る。この辺は、例外発生順序を保つために張られるので、実行時間は 0 とする。さらに、boundcheck 命令と ld4 命令の間に、S-PEI と H-PEI 間の例外依存を表す辺を張る。この辺は、命令実行順序を保つために張られるので、実行時間は 0 とする。

S-PEI を比較命令と分岐命令に分解して例外依存を制御依存によって表現する方式が、従来提案されている[117]。本方式では、例外依存を DAG 表現上の辺で表現することによって、最適化中にブロックが分割されないことが、図 6.3の例からもわかる。従って、本方式は、コンパイラの内部表現が使用する記憶領域を節約できる、ブロック内の命令数の増加によって投機的命令移動を含む様々なブロック内最適化を効果的に適用可能である、という利点を持つ。

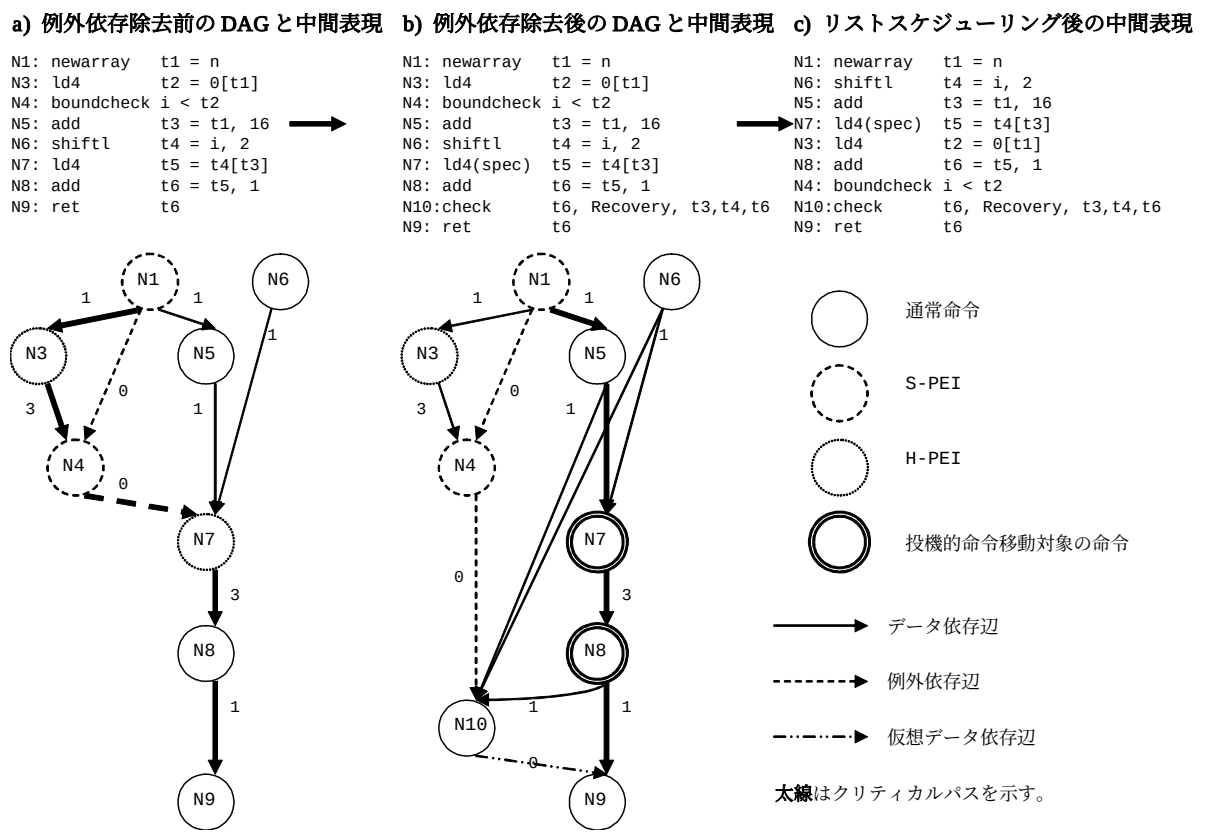


図 6.3: 例外依存除去の適用例

6.3.2 例外依存の制約除去アルゴリズム

本節では、例外依存辺を投機的命令移動によって除去する方法を示す。S-PEI と H-PEI 間の例外依存辺の除去は、以下のアルゴリズムに基づいて行われる。

1. 投機的命令移動の判断：　H-PEI の節点へ入力される例外依存辺と真データ依存辺で制約される 2 つの最早実行開始時間を比べる。もし、例外依存辺による最早実行開始時間のほうが遅いならば、投機的命令移動を行うために H-PEI を例外発生を抑制した投機的命令で置き換える。2 つの最早実行開始時間の差が、例外依存除去によって得られる時間短縮である。
2. 例外依存辺の連結：　投機的命令の実行が成功したかどうかを調べる `check` 命令（`N10`）を、DAG に挿入する。`check` 命令は、実行の成否を調べる値、復旧コードの分岐先、復旧コードで定義・参照される値、をオペランドとして持つ。
   次に、S-PEI から投機的命令への例外依存を除去する[iv]。除去した例外依存を S-PEI から `check` 命令へ張る。これによって、`check` 命令の実行結果に基づいて実行されるかもしれない復旧コード内の H-PEI が、S-PEI が実行された後に安全に実行されるこ

[iv]投機的命令がメモリからのロード命令であり、メモリへのストア命令からの順序依存がある場合、データ依存に関する投機的ロード命令によって順序依存を除去できるが、本章では扱わない。

とを保証する。

3. 投機的移動を行う命令範囲の決定： H-PEI を出発点とし、真データ依存によって連結している命令列を投機的命令移動を行う範囲とする。真データ依存をたどる間にメモリロードや整数の割り算命令などの H-PEI が現れたら、例外発生を抑制した投機的命令で置き換える。ただし、分岐命令やメモリストア命令などの副作用を持ち取り消しが困難な命令がが現れたら、それ以降の真データ依存をたどることを中止する。この方法については、6.4節で詳しく述べる。

   さらに、Java 言語では例外が発生して例外ハンドラが例外を捕捉した場合、例外ハンドラから実行が再開する。投機的実行によって失敗した値の参照を例外ハンドラで許すと、復旧コードを例外ハンドラにも生成する必要がある。例外ハンドラは、複数の S-PEI から制御が移る可能性があるので、それぞれの S-PEI に対応する復旧コードを１つの例外ハンドラに効率よく生成することは難しい。従って、例外ハンドラで参照される変数を定義する命令は、投機的移動の対象としない。

4. 復旧コードのためのデータ依存辺の連結： コンパイラは、復旧コードの直前で生きている値（以下、live-in と呼ぶ）集合、復旧コードの直後で生きている値（以下、live-out と呼ぶ）集合、を check 命令に付加する。復旧コードは、投機的に実行した命令を再実行して正しい値を得る。復旧コードを実行する際に必要な値（live-in に等しい）は、投機的移動を行った命令外で定義された値である。これは、投機的命令移動を行っていない命令か、他の投機的移動を行った命令、で定義されたオペランドである。また、復旧コードを実行した後にメインコードで必要な値（live-out に等しい）は、復旧コードで定義されて投機的移動を行った命令外で参照される値である。これは、投機的命令移動を行っていない命令か、他の投機的移動を行った命令、で参照されるオペランドである。従って、復旧コードに関する live-in・live-out 集合を求めるアルゴリズムは図 6.4で示される。

   本方式では、復旧コードは DAG 表現上で陽に表現しない。なぜなら、復旧コードを DAG 表現上で陽に表現すると、復旧コード内の定義とメインコードの定義の合流点が生成され、これが他の最適化やレジスタ割り付けに悪影響を与える可能性があるためである。また、live-out 集合に関して真データ依存辺を張ると、合流点として SSA 形式のφ命令が生成され他の最適化の妨げになるので、ここでは張らない。従って、live-in 集合に関してのみ、真データ依存辺を張る。

5. メインコードと復旧コード間の順序依存辺の連結： 投機的移動を行う命令を推移的に辿り末端の命令が定義するオペランドを、check 命令が投機実行の成否を調べるオペランドとして、真データ依存辺を張る。この結果、投機的命令移動を行う命令列の後に check 命令が配置される。また、復旧コードで再定義された live-out 集合の変数は投機的実行を行わない命令列で使用されるので、投機的実行を行わない命令列の前

になければならない。従って、投機的命令移動を行わない命令列が check 命令の後に正しく配置されるために、check 命令から復旧コードで定義されたデータを使用する命令へ仮想データ依存辺を張る。

```
sc<入力>     : 同一の投機的移動を行う命令の集合
scOTH<入力> : 別の投機的移動を行う命令の集合
li<出力>     : live-in オペランドの集合
lo<出力>     : live-out オペランドの集合

li = lo = ϕ
for (s ⊂ statements(sc)) {
  for (o ⊂ dst operands(s)) {
    if ((Succ(o) ∩ src operands(sc) == ϕ) ||
        (Succ(o) ∩ src operands(scOTH) ≠ ϕ)) {
      //左辺値が同一の投機的移動が行われる命令群以外で使われる
      lo ∪= o
    }
  }
  for (o ⊂ src operands(s)) {
    if ((Pred(o) ∩ dst operands(sc) == ϕ) ||
        (Pred(o) ∩ dst operands(scOTH) ≠ ϕ) {
      //右辺値が同一の投機的移動が行われる命令群外で定義される
      li ∪= o
    }
  }
}
```

図 6.4: 復旧コードの live-in、live-out 集合を求めるアルゴリズム

以上の処理が行われた後、リストスケジューリングを適用する。

6.3.3 アルゴリズムの適用例

本節では、前節のアルゴリズムを図 6.3のプログラムに適用する例を示す。

まず、プログラム中の H-PEI に対して、投機的命令移動を適用するかどうか判断する。図 6.3 a)では、N3 と N7 が H-PEI である。N3 には入力する例外依存辺が存在しないので、投機的命令移動の対象としない。N7 では、例外依存辺で制約される最早実行開始時間が 4（=1+3+0）、真データ依存辺で制約される最早実行開始時間が 2（=1+1）である。前者の時間の方が遅いので投機的命令移動を行うと判断する。N7 の ld4 命令を、例外発生を抑制した投機的ロード命令 ld4(spec)で置き換える。

次に、投機的命令移動を行う命令に関する例外依存辺の張り替えを行う。図 6.3 a)では、S-PEI の N4 から H-PEI の N7 への例外依存辺を除去し、check 命令の N10 へ張り替える。

次に、投機的移動を行う H-PEI に後続する命令の範囲を決定する。図 6.3 a)では、ロード命令である N7 と、単純な演算である N8 を投機的移動の対象とする。制御を変更する ret 命令は対象としない。

次に、復旧コードのためのデータ依存辺の連結を行う。図 6.3 a)では、N7 と N8 に対して live-in 集合 t3、 t4 と live-out 集合 t6 が求められる。t3、 t4 を定義する N5、 N6 から N10 へ真データ依存辺を張る。

次に、メインコードと復旧コード間の順序依存辺の連結を行う。図 6.3では、N10 から投機的移動を行う命令群の次の命令 N9 へ仮想データ依存辺を張る。また、N8 から N10 へ、投機実行の成否を調べるオペランド t6 のための真データ依存辺を張る。結果として、図 6.3 b)が得られ

る。

最後に、図 6.3 b)に対してリストスケジューリングを適用する。クリティカルパス長が例外依存除去前に８サイクルであったのが、例外依存除去後は６サイクルになっている。一般に配列の１要素を参照する場合、NullPointerException と ArrayOutOfBoundsException を調べるための２つの S-PEI と、配列長と配列要素を読む２つの H-PEI であるロード命令が生成されるので、本手法によってより大きな性能向上が期待される。

6.4 投機的命令移動を行う際のコンパイル方法

本章では、コンパイラが投機的命令移動を行う際に考慮すべき２つの点を示し、解決方法を述べる。１つは、復旧コードが正しく実行可能で、SSA 形式を通常形式に戻す際に余計なレジスタ間移動命令が発生しないことを保証して投機的移動を行う命令列を選ぶことである。もう１つは、復旧コードによるメインコードのレジスタ割り付けと命令配置への影響を最小限にすることである。

6.4.1 投機的移動命令選択

本節では、復旧コードを正しく実行可能で、SSA 形式を通常形式に戻す際に余計なレジスタ間移動命令が発生しない投機的移動命令列の選択手法を述べる。投機的命令移動によって余分なレジスタ移動命令が生成されると、実行ユニットも実行時間も消費される。これにより投機的命令移動による利得が打ち消される可能性がある。従って、余分な命令の生成は抑制しなければならない。この要請を満たすために、H-PEI から始まる真データ依存辺によって繋がれた命令列を辿りながら、以下の条件を満たす命令群を投機的移動を行う命令とする。

- 副作用を持つ命令を選択しない：　投機的移動を適用した結果、例外発生を抑制した命令の実行結果が失敗する可能性がある。その結果を call 命令やメモリストア命令等の取り消しや再実行が難しい副作用を持つ命令で利用することは、投機的実行が失敗した時に正しい再実行を保証できない。従って、副作用を持つ命令は投機的移動を行わない。
- 循環グラフを生成する命令は選択しない：　もし DAG 上に循環グラフが生成されたならば、DAG 上でのリストスケジューリングが不可能になり、正確な実行時間の見積もりや、正しい命令配置が不可能になる。Java プログラムの中間表現は、他の言語の中間表現より例外依存や順序依存を多く持つので、この条件について特に注意しなければならない。

  図 6.5に、循環グラフを生成してしまう例を示す。この例において、N3 は副作用を持つ命令であると仮定し、N2 と N4 を投機的命令移動の対象とする。N1 から N2 への例外依存辺を除去して、N1 から新たに挿入した check 命令の節点 N6 に張り替える。また、投機実行の成否を調べるための真データ依存辺を N4 から N6 へ張る。さ

らに、check 命令を実行した結果、復旧コードを実行すると N2 と N4 が再実行されるので、これらの計算結果を利用する N3 と N5 へ仮想データ依存辺を張る。また、復旧コードでは N3 の結果を利用するので、N3 から N6 へ真データ依存辺を張る。この結果、N3→N4→N6 で循環グラフが生成され、正しいコードが生成できなくなる。この場合、コンパイラは N2 のみを投機的移動の対象命令としなければならない。

- 変数の生存区間を資源以上に増やさない：　投機的移動を行った結果命令の配置が変わり、変数の生存区間が変化することがある。その結果、同時に生存する変数の数が、CPU が持つレジスタ数を超えることがある。もし超える場合は、物理レジスタ割り付けの段階でスピルコードが生成される。DAG の節点を逆帰りがけ順に並べた時、各節点をよぎる真データ依存辺の数（例えば、図 6.3 b）の N7 の直後ならば 3[v]）を数えることによって、同時に生存する変数の数が分かる。スピルコードが生成されないことを保証するために、この数がレジスタ割り付けで利用可能なレジスタ数を超える移動になる命令は選択しない。
- web を構成する変数を複数参照しない：　web を以下のどちらかの条件を満たす命令を真データ依存で結んだ命令の集合と定義する[73]。1) 任意の命令で定義された変数が SSA 形式のϕ命令で使用されている時に、その変数を使用する命令。2) ϕ命令で定義された変数を使用する命令。直感的にはϕ命令で結ばれた変数のリンクであり、web を構成する変数は通常形式への変換時に同一変数が割り振られる。ϕ命令は通常形式に変換すると代入命令にはならないので、変数の名前書き換えを行うとレジスタ移動命令が生成される。

  投機的移動を行った命令で参照される変数は復旧コードでも参照されるため、復旧コードの live-in・live-out 集合を check 命令まで生存させて、値が保存する必要がある。web を構成する変数の（SSA 形式における）別世代が投機的移動を行う命令群で参照された場合、web を構成する変数は通常形式への変換時に同一変数が割り当てられるので、これらの変数は変換時に check 命令で生存区間の干渉を起こす。干渉が起きた場合、通常形式への変換時にプログラムの正しさを保証するためにレジスタ移動命令が生成される。余分な命令の生成を避けるために、変数の生存区間の干渉を引き起こす命令は、投機的移動の対象としない。

  図 6.6に、web 上で変数の生存区間が干渉する例を示す。この例において、S1、S2、S5、S7 は変数 w に関して web を形成している[vi]。S3 と S5 に対して投機的命令移動を行うと、復旧コードで参照される w1 は S6 まで値を保持しなければならない。通常形式に変換する際に w1、w2、w3 は同一変数が割り当てられる。従って、S6 で生

[v] N5→N10、N6→N10、N7→N8、である。

[vi] S1→S2、S1→S5、S1→S7、S5→S7、である。

存している w1 と w2 が干渉するので、レジスタ移動命令 S4 が生成される。従って、この場合 S3 のみを投機的命令移動の対象とする。

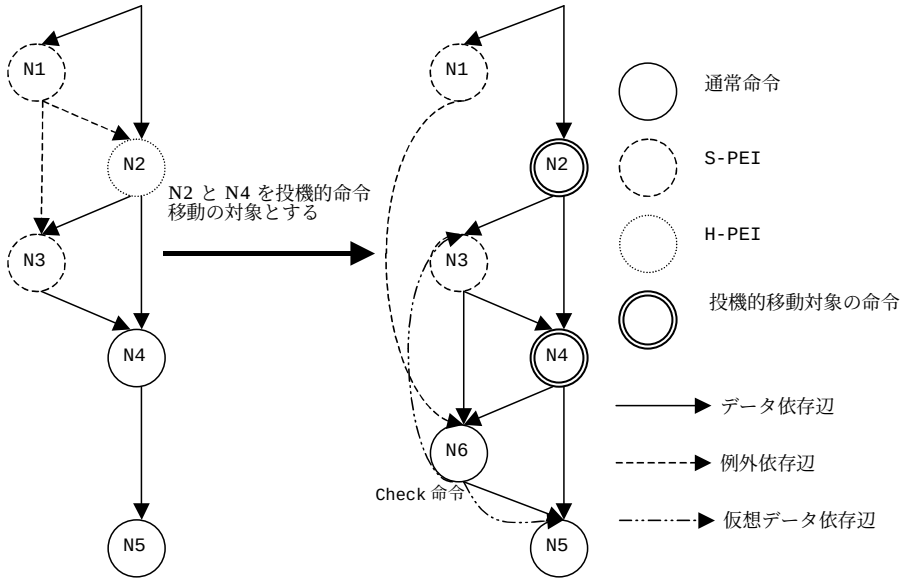


図 6.5: 循環グラフの生成例

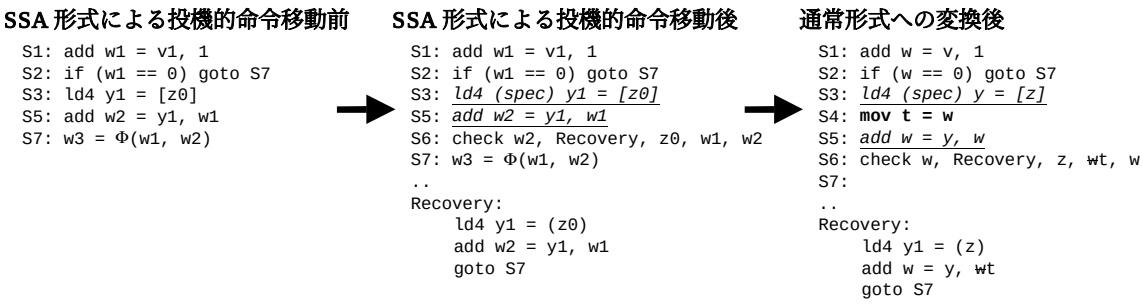


図 6.6: web 上の変数が干渉する例

6.4.2 復旧コード生成

本節では、復旧コードの生成方法について述べる。復旧コードは、投機的移動を行った命令の実行が失敗したときのみ実行される。従って、復旧コードがメインコードに与える影響は最小限にしなければならない。このために、1)中間表現で陽に表現されない復旧コードの生成、2)レジスタの干渉を最小限にする、3)同時実行可能な命令群を生成する際の制約を無くす、という３つの問題を解決した、復旧コードを生成する方法について述べる。

本方式では、6.3.2節でも述べたが、復旧コードがメインコードの最適化に与える影響を除外するために、復旧コードは中間表現上で陽には表現されない。投機的移動を行う命令に DAG 表現上で印を付け、コード生成時に印が付けられた命令を復旧コードとして複製する。複製の際に、例外発生を抑制した投機的命令は元の H-PEI に置き換える。コード複製を行うので、メインコードと復旧コードのレジスタ使用は同一になる。さらに、リソース情報なども複製するこ

とで、復旧コード内でも同時実行可能な命令群を構成することが可能になり、ネイティブコードの大きさも小さくできる。復旧コードの生成アルゴリズムは以下の 4 段階からなる。

1. プロローグとエピローグの生成：　復旧コードの中で定義されるレジスタが、live-in と live-out 集合のレジスタを除いて、復旧コードのプロローグでメモリに待避されて、エピローグでメモリから復元されること保証するコードを、コンパイラが生成する。この結果、復旧コードの中で閉じるレジスタ定義参照は、メインコードのレジスタの生存区間に影響を与えない。復旧コードがメインコードに与えるレジスタ使用の影響は、投機的移動を行った命令の中で使用されるレジスタが、復旧コードまで生存区間が伸びることだけである。

   復旧コードで定義されたレジスタのうち live-out 集合に含まれるものはメインコードで使われるので、復旧コード実行前の値を保存してはならない。また、live-in 集合のうち復旧コードで参照が終了するものは保存する必要が無く、メインコードでも参照されているものは復旧コードでは定義されないのでやはり保存する必要がない。つまり、復旧コードで定義されたレジスタから live-in・live-out 集合を除いたものを、復旧コードの前後で保存する。よって、投機的移動を行う命令群に対応する復旧コードで待避・復元されるレジスタの集合は、図 6.7のアルゴリズムによって求められる。
2. 復旧コードの複製：　投機的移動を行う命令列を、エピローグとプロローグの間に複製する。命令を複製する際に、例外発生を抑制した投機的命令を元の H-PEI に置き換える。これによって、投機的実行による失敗した実行結果は、復旧コードで正しい実行結果で上書きされるので、正しい実行を保証する。
3. 同時実行可能な命令群の複製：　IA-64 等の VLIW 計算機では数命令が同時実行可能であり、この同時実行可能な命令群を構成することによってプログラムから並列性を抽出する。一般に、分岐命令は同時実行可能な命令群の先頭命令へ分岐する。例えば、復旧コードからメインコードへ戻る場合、check 命令の次の命令群へ戻る。

   check 命令が同時実行可能な命令群の最後尾にある場合は、check 命令から復帰コードへ分岐して、メインコードへ次の命令群へ分岐命令で復帰しても実行を正しく再開できる。しかし、check 命令が同時実行可能な命令群の途中にある場合、そこから復旧コードへ分岐して、メインコードへの次の命令群へ復帰すると、check 命令と同一命令群にある後続命令が実行されない。よって、実行を正しく再開できない。この問題は、check 命令を含む命令群における後続命令を、対応する復旧コードの末尾に複製することで解決可能である。
4. メインコードへ戻るジャンプ命令の生成：　復旧コードの最後尾に、メインコードにある check 命令の次の命令群の先頭命令への分岐命令を生成する。

```
sc<入力>     : 同一の投機的移動を行う命令の集合
scOTH<入力> : 別の投機的移動を行う命令の集合
li           : live-in レジスタの集合
lo           : live-out レジスタの集合
kl           : 復旧コードで定義されるレジスタの集合
sp<出力>     : 復旧コードの出入りで待避・復元するレジスタの集合

kl = li = lo = ϕ
for (s ⊂ instuctions(sc)) {
  for (r ⊂ dst registers(s)) {
    kl ∪= r
    if ((Succ(r) ∩ src registers(sc) == ϕ) ||
        (Succ(r) ∩ src registers(scOTH) ≠ ϕ)) { lo ∪= r }
  }
  for (r ⊂ src registers(s)) {
    if ((Pred(r) ∩ dst registers(sc) == ϕ) ||
        (Pred(r) ∩ dst registers(scOTH) ≠ ϕ)) { li ∪= r }
  }
}

sp = kl ∩ (li ∪ lo)
```

図 6.7: 復旧コードで待避・復元されるレジスタの集合を求めるアルゴリズム

上記のアルゴリズムを、図 6.8の例に適用する例を示す。この例では、投機的移動を行う命令群は2つある。1つは I1、I4 で、もう1つは I2、I3、I4 である。

まず、命令群 I2、I3、I4 に対応する復旧コード Recovery2 において、live-in レジスタ集合は{r2、r11}、live-out レジスタ集合は{r12、r21}、復旧コードで定義されるレジスタ集合は{r12、r13、r21}である。よって、待避・復元されるレジスタ集合は{r13}となり、プロローグでは I13 が、エピローグでは I17 が生成される。復旧コード Recovery1 では、待避・復元されるレジスタ集合は空であるので、退避・復元コードは生成されない。

次に、復旧コードを生成する。Recovery1 のために I1、I4 から I9、I10 が生成され、Recovery2 のために I2、I3、I4 から I14、 I15、I16 が生成される。

次に、同時実行可能な命令群内の命令複製を行う。I4、I5、I6 が同時実行可能な命令群である。このとき、I5 から分岐した Recovery1 は I6 へ戻りたいが、命令群の途中であるため不可能である。従って、同一命令群の後続命令 I6 を Recovery1 の末尾に I11 として複製する。Recovery2 は I6 から分岐して同一命令群には後続命令が存在しないので、命令の複製の必要は無い。

最後に、メインコードへ戻るジャンプ命令の複製を行う。Recovery1 の例では、I12 によって次の命令群の先頭命令 I7 へ戻る。Recovery2 の例では、I18 によって次の命令群の先頭命令 I7 へ戻る。

投機的移動前

```
I1:  ld4 r2 = (r1)
I2:  ld4 r12 = (r11)
I3:  add r13 = r12, 1
I4:  add r21 = r2, r13


I7:  mov .., r12
I8:  mov .., r21
```

投機的移動後

```
I1:  ld4(spec) r2 = (r1)   // 同時実行群 0
I2:  ld4(spec) r12 = (r11) //      “
I3:  add r13 = r12, 1      //      “
I4:  add r21 = r2, r13     // 同時実行群 1
I5:  check r2, Recovery1   //      “
I6:  check r12, Recovery2  //      “
I7:  mov .., r12           // 同時実行群 2
I8:  mov .., r21           //      “

Recovery1:
I9:  ld4 r2 = (r1)         // I1 の複製
I10: add r21 = r2, r13     // I4 の複製
I11: check r21, Recovery2  // I6 の複製
I12: goto I7

Recovery2:
I13: spill r13
I14: ld4 r12 = (r11)       // I2 の複製
I15: add r13 = r12, 1      // I3 の複製
I16: add r21 = r2, r13     // I4 の複製
I17: fill r13
I18: goto I7
```

図 6.8: 復旧コードの生成

6.5 実験結果

本節では、本章で提案した方式を実験によって評価した結果を示す。評価項目は、以下の 3 つである。

- 中間表現の記憶域効率。
- 小規模プログラムにおける性能向上。
- 標準的なベンチマークにおける性能向上。

実験のために、IBM Developers Kit for Windows on Itanium, Java Technology Edition, Version 1.3.1 の Just-In-Time コンパイラ[43]に、本章で述べた例外依存の除去方式を実装した。全ての測定は、IBM 社の IntelliStation Z Pro モデル 6894（Itanium 800MHz x 2、メモリ 2GB）に Windows XP Advanced Server を使用して行った。

6.5.1 中間表現の記憶域効率

本節では、例外依存辺を持つ DAG 表現による中間表現の記憶域に関する効率を評価する。表 6.1は、例外依存辺を持つ DAG と、例外依存を制御依存で明示的に表現した DAG 表現におけるブロックや辺の統計を示す[vii]。評価には、SPECjvm98[6]に含まれる 7 つのプログラム（`compress`、 `jess`、 `db`、 `javac`、 `mpegaudio`、 `mtrt`、 `jack`）を `size=100` で実行したものを使用した。DAG 表現に変換する対象の中間表現は、ループバージョニング[43]や部分冗長除去による例外検査の除去[45]により、冗長な例外検査は十分除去されている。

表の 2、 3 列目は、例外依存辺を持つ DAG 表現により、DAG が持つブロックの数が約 1/4

[vii]実際には、辺や節点数を表現に応じて増減して見積もった。

になり、ブロックに含まれる命令数が約４倍になったことを示している。これにより、ブロック内最適化の適用機会が増える可能性がある。また、例外依存制約除去に関する投機的命令移動を、ブロック内の例外依存辺の除去によって適用できる。従って、最適化にかかる時間を短縮可能である。コンパイル時間を多く費やすことが出来ない動的コンパイラにとって、この短縮は特に重要である。

表の４、５列目は、例外依存辺を持つ DAG 表現により、ブロック内の辺の数は増加したが、DAG 全体では辺の数が減少したことを示している。これにより、コンパイル中の中間表現が必要とする記憶領域が削減できることを確認できた。

表 6.1: 例外依存辺を持つ DAG と持たない DAG 表現における記憶域効率の比較

	例外依存辺を持たない DAG	例外依存辺を持つ DAG
DAG のブロックの総数	142679	37327
１つのブロックに含まれる平均命令数	1.23	4.71
DAG のブロック間の辺の総数	262791	54669
ブロックに含まれる依存辺の総数	80190	145952

6.5.2 小規模プログラムにおける性能向上

本節では、２つの小規模なプログラムを実行した結果を示す。用いたプログラムの説明を表6.2に示す。全て２次元配列を参照するプログラムであり、２次元配列を用いるプログラムを取り上げた理由は、Java 言語では多次元配列が定義されておらず、一般的な Java 仮想マシンにおいて多次元配列は配列の配列として実装されているため、２次元目以降の配列参照に関する S-PEI を最適化によって除去することが非常に難しい。従って、例外依存が残り本手法を適用する機会が多いことが期待されるためである。これらのプログラムでは、カーネルでは `NullPointerException`、`ArrayOutOfBoundsException` は発生しない。

表 6.2: 小規模プログラムの説明

ベンチマーク	プログラムの説明
All-pairs Shortest-path	全ての点の間の最短距離を求める。
Matrix Multiply	２つの行列の内積を計算する。

図 6.9に、例外依存の除去と S-PEI を除去する最適化であるループバージョニングを行わなかった場合に対する、例外依存除去とループバージョニングをそれぞれ適用した場合の性能向上を示す。

ループバージョニングを適用しなかった時に、例外依存の除去を行った場合、1%〜24%（平均 12.5%）の性能向上が得られている。ループバージョニングを適用した時に、例外依存の除去を行った場合、5%〜24%（平均 14.5%）の性能向上が得られている。カーネルコードでは S-PEI による例外検査が多いので、全てのプログラムで例外依存の除去の効果があることが示され

た。

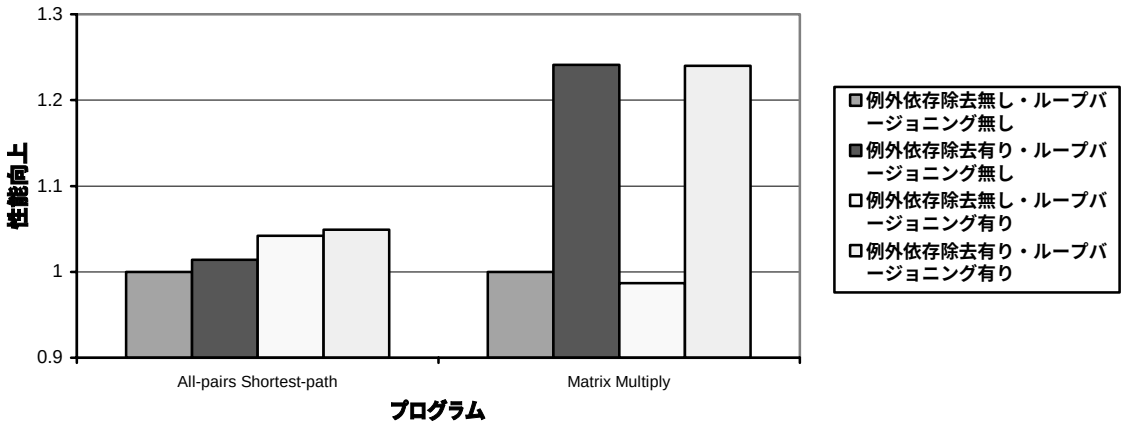


図 6.9: 小規模プログラムの実行性能向上

表 6.3に、ループバージョニングを適用しない際に、例外依存の除去を選択的に行った場合と、全て行った場合の性能向上を示す。全ての例外依存を除去した場合、選択的に行った場合よりも性能向上が小さい。これは以下の理由によるものと考えられる。現在の Itanium プロセッサの実装では、例外発生を抑制した投機的ロード命令は TLB ミスを発生するとロードが失敗したとする[120]。従って、正しいアドレスが与えられても TLB ミスが起きるとロード失敗となり、復旧コードが実行される割合が増加する。従って、本方式の実行時間短縮の正確な見積りによる例外依存除去の適用判断が有効であることが示された。

表 6.3: 小規模プログラムの例外依存除去方法による実行性能向上
（例外依存除去を行わない場合を 1 とする）

	All-pairs Shortest-path	Matrix Multiply
選択的に例外依存を除去	1.02	1.24
全ての例外依存を除去	1.00	1.21

図 6.10に、例外依存の除去とループバージョニングを行わなかった場合に対する、例外依存除去とループバージョニングをそれぞれ適用した場合のカーネルのコード増分を示す。

ループバージョニングを適用しなかった時に例外依存の除去を行った場合、32%～64%のコード増分が起きている。ループバージョニングを適用した時に、例外依存の除去を行った場合、32%～61%のコード増分が起きている。最も増分が多いのは、性能向上が最も大きい行列積の場合である。コード増分の主な理由は復旧コードの生成によるもので、メインコードのクリティカルパスに影響は与えていない。従って、性能向上が大きい行列積でコード増分が大きいということは、投機的移動が行われた命令数が多い、ということを示している。

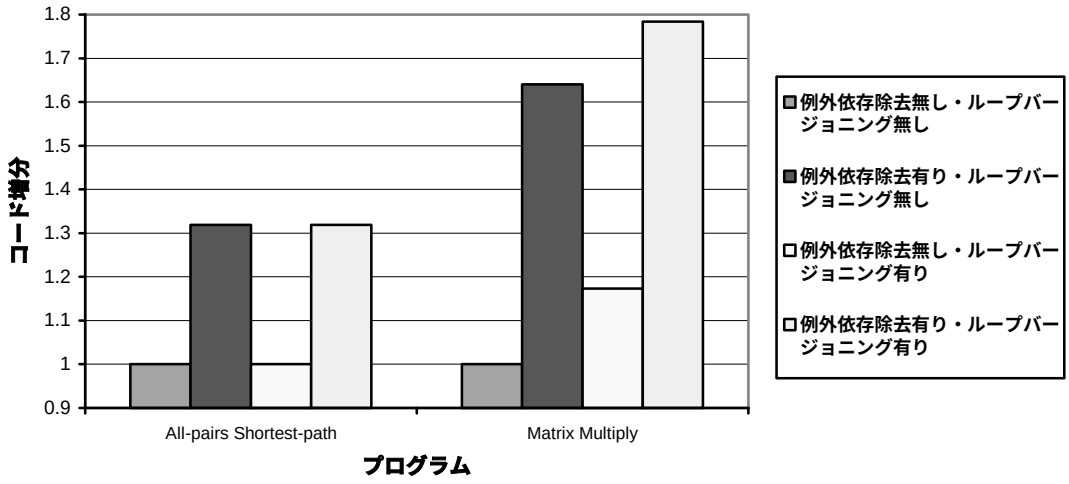


図 6.10: 小規模プログラムのコード増分

6.5.3 Java Grande Benchmark における性能向上

本節では、標準的に用いられるベンチマークの１つである Java Grande Benchmark Suites [104] を実行した結果を示す。Java Grande Benchmark Suite にはいくつかのベンチマークがあるが、ここでは Section II の kernel を SizeA で実行した。それぞれのプログラムの説明を、表 6.4に示す。これらのプログラムでも、ベンチマーク中は `NullPointerException`、`ArrayOutOfBoundsException` は発生しない。

表 6.4: Java Grande Benchmark Suites のプログラムの説明

ベンチマーク	プログラムの説明
Series	最初の N 個のフーリエ係数の計算.
LUFact	LU 分解を用いて N x N 次元の方程式を解く。
HeapSort	N 個の整数をヒープソートで整列する。
Crypt	IDEA で暗号化と復号化を行う。
FFT	倍精度複素数の FFT 計算を行う。
SOR	加速緩和(SOR)法で方程式を解く。
SparseMatmul	２つの１次元粗行列の乗算

図 6.11に、例外依存の除去とループバージョニングを行わなかった場合に対する、例外依存除去とループバージョニングをそれぞれ適用した場合の性能向上を示す。

ループバージョニングを適用しなかった時に例外依存の除去を行った場合、-1%～10%（平均 1.8%）の性能向上が得られている。特に、SOR、SparseMatmul で性能が向上している。

ループバージョニングを適用した時に例外依存の除去を行った場合、-2%～10%（平均

1.7%）の性能向上が得られている。やはり、SOR、SparseMatmul で性能が向上している。この理由は、SOR のカーネルループでは 2 次元配列の要素が頻繁に使用され、SparseMatmul のカーネルループでは間接参照を伴う配列要素が頻繁に参照されるためである。LUFact では、ループバージョニングによって、25%性能が向上している。

Crypt では、ループバージョニングによって少し性能が落ちている。この理由は以下の通りであると考えられる。ループバージョニングでは、ループ実行前に例外検査が削減されたループを実行可能であるかどうか調べるコードが生成される。この判断のためのコードの実行がオーバヘッドとなる。

FFT では、例外依存の除去によって少し性能が落ちている。このプログラムでは、`NullPointerException`、`ArrayOutOfBoundsException` は発生していないので、不正なアドレスを伴った投機的ロード命令は実行されていない。従って、前節で述べたと同様に、TLB ミスによるロード失敗の増加が原因であると考えられる。

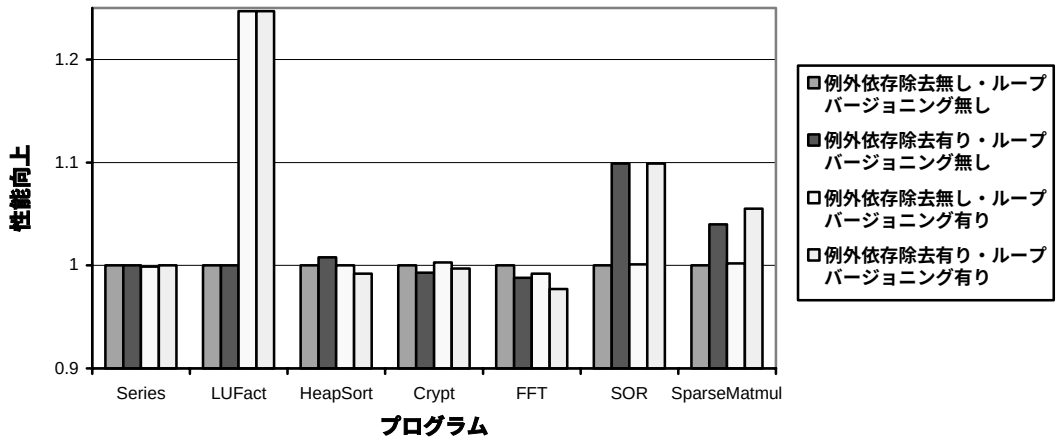


図 6.11: JavaGrande Benchmark Suites の実行性能向上

図 6.12に、例外依存の除去とループバージョニングを行わなかった場合に対する、例外依存除去とループバージョニングを適用した場合のコード増分を示す。コードの増分は 0%～7%である。プログラム全体で見ると、小規模ベンチマークの場合ほどコード増分は大きくない。

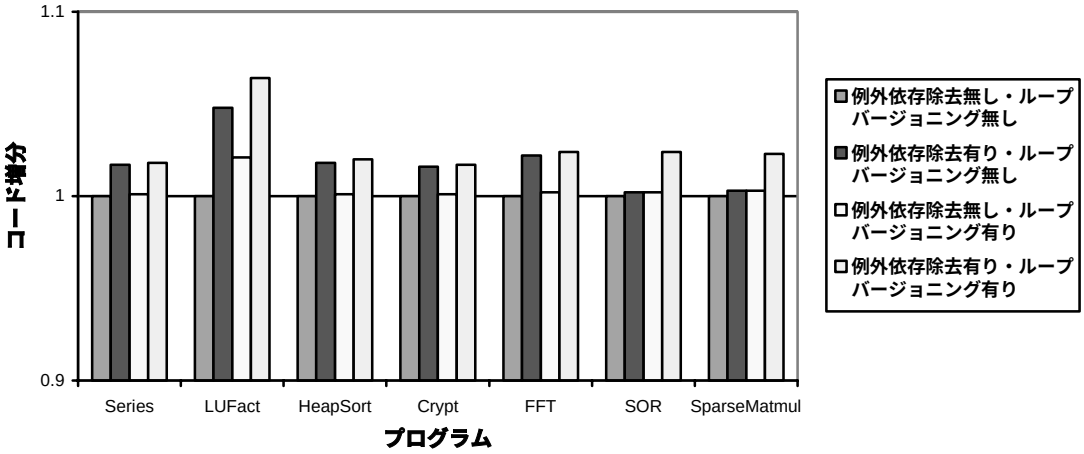


図 6.12: JavaGrande Benchmark Suites のコード増分

6.5.4 SPECjvm98 における性能向上

本節では、標準的に用いられるベンチマークの１つである SPECjvm98 を `size=100` で実行した結果を示す。これらのプログラムでも、ベンチマーク中は `NullPointerException`、`ArrayOutOfBoundsException` は発生しない。

表 6.5: SPECjvm98 のプログラムの説明

ベンチマーク	プログラムの説明
compress	LZW 法を用いた圧縮・展開プログラム。size=100 で実行。
jess	NASA の CLIPS エキスパートシステム。size=100 で実行。
db	データ管理プログラム。size=100 で実行。
javac	JDK 1.0.2 の java class ファイルを生成するコンパイラ。size=100 で実行。
mpegaudio	MP3 のデータを伸長するプログラム。size=100 で実行。
mtrt	２つのスレッドが走るレイトレーシングプログラム。size=100 で実行。
jack	標準ライブラリが用意するデータ構造を操作する。size=100 で実行。

図 6.13に、例外依存の除去とループバージョニングを行わなかった場合に対する、例外依存除去とループバージョニングを適用した場合の性能向上を示す

ループバージョニングを適用しなかった時に例外依存の除去を行った場合、0%～10%（平均2.4%）の性能向上が得られている。特に、`compress`、 `mpegaudio`、 `mtrt` という配列参照が多い、つまり S-PEI が多いベンチマークで性能が向上している。

ループバージョニングを適用した時に例外依存の除去を行った場合、-1%～12%（平均2.3%）の性能向上が得られている。やはり、`compress`、 `mpegaudio`、 `mtrt` という配列参照が多い、つまり S-PEI が多いベンチマークで性能が向上している。特に `mpegaudio` は、カーネルループで２次元配列の参照が頻繁に行われるので、性能向上が大きい。図 6.14に、`mpegaudio` のカーネルループの一部を示す。配列 `c` の１次元目はループ内で変化する値である。

2 次元目は、その値が指す 1 次元配列から読み込まれる。従って、冗長な例外検査を除去する最適化を用いても、配列 C の 2 次元目を参照する際の S-PEI を除去することができない。従って、mpegaudio では本手法の効果が大きい。実際、カーネルループのクリティカルパスの長さは 31 クロックから 25 クロックに減少している。

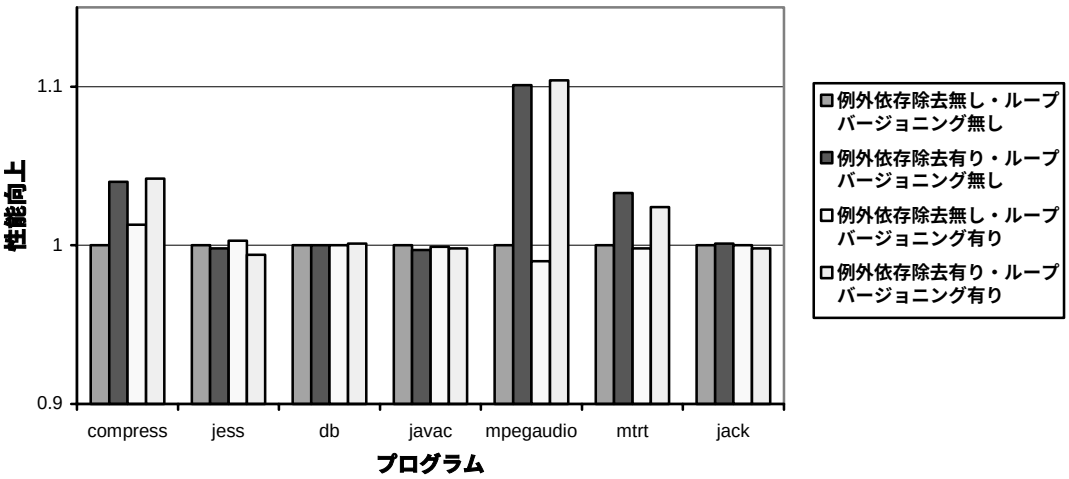


図 6.13: SPECjvm98 の実行性能向上

```
for (int j = 0; j < n; j++) {
    float  B[] = A[i++];
    f += C[j][k+16] * B[0];
}
```

図 6.14: mpegaudio のカーネルループの一部

図 6.15に、例外依存の除去とループバージョニングを行わなかった場合に対する、例外依存除去とループバージョニングを適用した場合のコード増分を示す。

ループバージョニングを適用した場合でも、コードの増分は 0%～10%である。プログラム全体から見ると、mpegaudio の場合を除くとコード増分はそれほど多くない。最も増分が多いのは、性能向上が最も大きい mpegaudio の場合である。これも、投機的移動が行われた命令数が増加し、結果として性能向上が大きい、ということを示している。

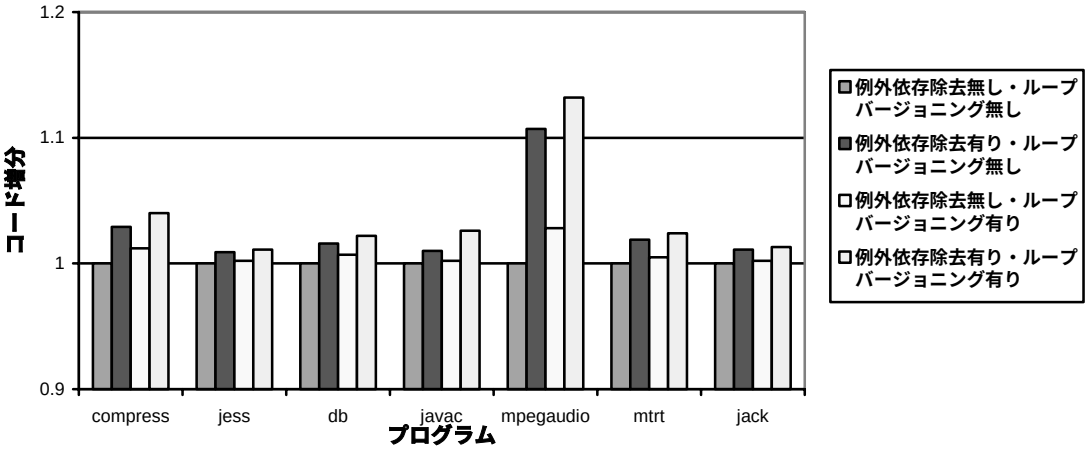


図 6.15: SPECjvm98 のコード増分

6.6 まとめ

本章では、型安全な言語において例外依存を越えた投機的命令移動を効率よく行う枠組みと投機的移動を行う命令の範囲の決定方法と補償コードを生成する方法を提案した。本手法によって、プロセッサが持つ投機的命令実行機能を適切に利用する機会を増やすことが可能である。さらに、本章で提案した手法を IA-64 の Java Just-In-Time コンパイラに実装し、Java Grande Benchmark では最大 10%（平均 1.7%）の性能向上が、SPECjvm98 では最大 12%（平均 2.3%）の性能向上が得られた。

第7章 結論

本論文では、Java 言語の特徴である型安全性と多態性がもたらす、高頻度の型検査と例外検査、実行時に動的クラスローディングによってプログラムの構成が変化する環境における動的束縛の導入によるプログラム全体の解析の困難さ、によって発生する実行時オーバヘッドを削減する最適化手法について述べた。これらのオーバヘッドを削減し Java 言語処理系の実行性能を改善し実用的な性能を得ることは、Java 言語が従来の C++言語などに置き換わってプログラミング言語の主流となるためには不可欠であった。本論文では Java 言語処理系の実行性能を向上させるための、コンパイラにおける最適化手法について議論を行ってきた。

本論文における研究成果を以下にまとめる。

- 動的クラスローディングを行う言語において、コード書き換えによる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し、性能向上が得られることを示した。
- 型安全な言語を実装するために必要なオブジェクトの型検査を、頻繁に実行される部分をネイティブコードにインライン展開する高速化方法を提案し、従来提案されている方法より簡単な実装で同等以上の性能向上が得られることを示した。
- 型安全な言語におけるオブジェクトへアクセスを行う際のアドレスが正しいかどうかの例外検査を、プロセッサが提供する高速な条件分岐命令を有効利用し、例外が発生しない場合に実行される命令を最小限にすることで、高速化する手法を提案し、性能向上が得られることを示した。
- 型安全な言語におけるオブジェクトへの安全な参照を保証したまま、投機的命令移動を用いて命令間の順序制約を緩和し、高速化する手法を提案し、性能向上が得られることを示した。

本論文における研究成果は、IBM 社 IBM Developers Kit, Java Technology Edition で実際に使用されている。また、コード書き換えによる仮想メソッド呼び出しの直接デバーチャル化手法は、他の Java Just-In-Time コンパイラでも使用されている。

Java 言語がプログラミング言語の主流となった現在では、従来のプログラミング言語の主流であった C 言語や C++言語等の静的なコンパイルを行う処理系では困難であった、実行時情報を用いた動的な最適化に関する研究が数多く行われている。これらの研究により、Java 言語処理系は C 言語や C++言語等を越えて、より高速になってゆくであろう。

謝辞

本論文は、私が日本アイ・ビー・エム㈱東京基礎研究所在勤中に行った研究をまとめたものです。

本論文の執筆にあたり、学部・修士時代よりご指導頂き、終始懇切なる御指導・御鞭撻を賜わりました、早稲田大学理工学部情報学科　村岡洋一教授に、深く感謝の意を表します。

本論文の審査をお引き受け頂き、審査を通して貴重な助言を頂きました、早稲田大学理工学部情報学科　上田和紀教授、筧捷彦教授、深澤良彰教授、に深く感謝致します。また、審査の場において助言を頂きました早稲田大学理工学部情報学科　山名早人助教授に感謝いたします。

本論文は、日本アイ・ビー・エム㈱東京基礎研究所で 1995 年に始まった Java Just-In-Time コンパイラプロジェクトで得られた研究成果をまとめたものです。本プロジェクトを運営し、また本研究に終始御協力を賜わり、多くの御指導を賜わりました、日本アイ・ビー・エム㈱東京基礎研究所　中谷登志男氏に深く感謝致します。また、本研究を進める上で、終始熱心に御指導、御協力、御討論を頂き、IBM Java Just-In-Time コンパイラの実装に関わられた、日本アイ・ビー・エム㈱東京基礎研究所　小松秀昭氏、小野寺民也氏、郷田修氏、菅沼俊夫氏、河内谷清久仁氏、小笠原武史氏、川人基弘氏、安江俊明氏、竹内幹雄氏、緒方一則氏、稲垣達氏氏、古関聰氏、今野和浩氏（当時）、百瀬浩之氏（当時）、田端邦夫氏（当時）、に深く感謝致します。また、Java Just-In-Time コンパイラプロジェクトに関わり支えてくださった、日本アイ・ビー・エム㈱東京基礎研究所、日本アイ・ビー・エム㈱大和研究所、IBM T. J. Watson Research Center、IBM United Kingdom Hursley Laboratory、IBM Austin Laboratory、IBM Canada Toronto Laboratory、の多くの方々に感謝致します。

これまで、励まし、御支援、御助言を頂きました多くの友人に、深く感謝致します。また、私の研究活動を続けることを可能にして下さった多くの方々に、感謝致します。

最後に、今日まで私の研究生活を見守って下さった母と、本論文の完成を見ることなく手術後急逝した父に感謝致します。

参考文献

[1] Ken Arnold and James Gosling. Java Programming Language, Addison-Wesley, 1996.

[2] Ronald L. Johnston. The Dynamic Incremental Compiler of APL 3000. In *Proceedings of the APL '79 Conference*. Published as APL Quote Quad 9(4), pp. 82-87, 1979.

[3] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines, In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 269-279, 1998.

[4] David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin Locks: Featherweight Synchronization for Java, *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 258-268, 1998.

[5] Mario Wolczko. Benchmarking Java with Richards and DeltaBlue, available at http://research.sun.com/people/mario/java_benchmarking/.

[6] The Standard Performance Evaluation Corp., SPECjvm98 Benchmarks, available at http://www.spec.org/osg/jvm98/.

[7] James Gosling, Bill Joy, and Guy Steele. The Java Language Specification, Addison-Wesley, 1996.

[8] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification, Addison-Wesley, 1997.

[9] Patrick Chan and Rosanna Lee. The Java Class Libraries: An Annotated Reference, Addison-Wesley, 1996.

[10] Nathan P. Smith. Stack Smashing Vulnerabilities in the UNIX Operating System, available at http://destroy.net/machines/security/.

[11] David A. Wheeler. Secure Programming for Linux and Unix HOWTO, available at http://www.dwheeler.com/secure-programs/.

[12] Bjarne Stroustrup. The C++ Programming Language, Third Edition. Addison-Wesley 1997.

[13] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine, In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications,* pp. 36-44, 1998.

[14] Jeffery Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy, In *Proceedings of the 9th European Conference on Object-Oriented Programming – ECOOP '95*, volume 952 of LNCS, Springer-Verlag, pp. 77-101, 1995.

[15] Mary F. Fernandez. Simple and Effective Link-Time Optimization of Modula-3 Programs, In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 103-115, 1995.

[16] Young Gil Park and Benjamin Goldberg. Escape analysis on lists, In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 116-127, 1992.

[17] Holger M. Kienle. A SUIF Java Compiler. Technical Report TRCS98-18, OOCSB, 1998.

[18] Holger M. Kienle and Urs Holzle. Introduction to the SUIF2.0 Compiler System, Technical Report TRCS97-22, UCSB, 1997.

[19] The OSUIF group. The OSUIF Library, available at http://www.cs.ucsb.edu/~osuif/.

[20] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 451-490, 1991.

[21] Jeffrey Dean, Greg Defouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pp. 83-100, 1996.

[22] Jeffery Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy, In *Proceedings of the 9th European Conference on Object-Oriented Programming – ECOOP '95*, Volume 952 of LNCS, Springer-Verlag, pp. 77-101, 1995.

[23] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction, In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pp. 108-123, 1995.

[24] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen-mei W. Hwu. Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results, In *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 90-99, 1996.

[25] Pouha P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen-mei W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors, In *Proceedings of the 18th Annual International Symposium Computer Architecture*, pp. 266-275, 1991.

[26] IBM Corp. High Performance Compiler for Java, available at http://domino.watson.ibm.com/Comm/bios.nsf/pages/hpcj.html.

[27] Robert Haward. Offline Analysis and Compilation of Java-based Systems, available at http://www.towerj.com/whitepapers/OfflineConceptPaper.pdf.

[28] 千葉. Java 向け静的コンパイラによる仮想メソッド呼び出しの高速化, 情報処理学会論文誌：プログラミング研究会, Vol. 42, No. SIG2(PRO 9), pp. 26-36, 2001.

[29] Free Software Foundation. The GNU Compiler Collection, available at http://gcc.gnu.org/.

[30] EXCELSIOR, LLC. Excelsior JET: the Java(tm) Performance Solution, available at http://www.excelsior-usa.com/pdf/jetwp.pdf.

[31] John Whaley. Dynamic optimizations through the use of automatic runtime specialization. M. Eng., Massachusetts Institute of Technology, 1999.

[32] Hewlett-Packard Company. hp turbo chai, available at http://www.hp.com/products1/embedded/infolibrary/pdfs/turbochai.pdf.

[33] Gilles Muller, Barbara Moura, Fabrice Bellard, Charles Consel. Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code, In *3rd Usenix Conference on Object-Oriented Technologies and Systems (COOTS'97)*, pp. 1-20, 1997.

[34] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeno JVM, In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 47-65, 2000.

[35] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani, A

Dynamic Optimization Framework for a Java Just-In-Time Compiler. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 180-194, 2001.

[36] Urs Holzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pp. 326-336, 1994.

[37] Urs Holzle and David Ungar. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 4, pp. 355-400, 1996.

[38] Sun Microsystems, Inc. Java2 Platform Standard Edition, available at http://java.sun.com/products/jdk/1.2/.

[39] Sun Microsystems, Inc. The Java HotSpot Virtual Machine, available at http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.pdf

[40] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language, In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 273-282, 1992.

[41] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization, In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 32-43, 1992.

[42] IBM Corp. IBM Development kit. Java Technology Edition, available at http://www.ibm.com/developerworks/java/jdk/.

[43] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java Just-In-Time Compiler, *IBM Systems Journals*, Vol. 39, No. 1, pp. 175-193, 2000.

[44] Tamiya Onodera and Kiyokuni Kawachiya. A Study of Locking Objects with Bimodal Fields, In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 223-237, 1999.

[45] Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Effective Null Pointer Check Elimination Utilizing Hardware Trap, In *Proceedings of the Nineth International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS-IX)*, pp. 139-149, 2000.

[46] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. A Study of Exception Handling and Its Dynamic Optimization in Java, In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 83-95, 2001.

[47] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, Toshio Nakatani. Design, Implementation, and Evaluation of Optimizations in a Java(tm) Just-In-Time Compiler, *Concurrency: Practice and Experience*, Vol. 12, No. 6, pp. 457-475, 2000.

[48] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler, In *Proceedings of the ACM*

Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 294-310, 2000.

[49] Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark Mergen, Janice C. Shepherd, Stephen Smith. Implementing Jalapeno in Java, In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 314-324, 1999.

[50] Bowen Alpern, Dick Attanasio, John Barton, Michael Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio Serrano, Janice Shepherd, Stephen Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeno Virtual Machine, *IBM System Journal*, Vol. 39, No. 1, pp. 211-238, 2000.

[51] Derek White and Alex Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories Inc, 1999.

[52] Ole Agesen. GC points in a threaded environment. Technical Report SMLI TR-98-70, Sun Microsystems, Inc., 1998.

[53] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna and Derek White. An efficient meta-lock for implementing ubiquitous synchronization, In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 207-222, 1999.

[54] Ole Agesen and David Detlef. Mixed-mode Bytecode Execution. Technical Report SMLI TR-2000-87, Sun Microsystems, Inc., 2000.

[55] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java™ Under Dynamic Optimizations, In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 13-26, 2000.

[56] Compaq Computer Corp. The Compaq Fast Virtual Machine, available at http://www.compaq.com/java/FastVM.html.

[57] Transvirtual Technologies Inc., Kaffe, http://www.kaffe.org/.

[58] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. LaTTe: A Java VM Just-In-Time Compiler with Fast and Efficient Register Allocation, In *Proceeding of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 128-138, 1999.

[59] Yoo C. Chung, Soo-Mook Moon, Kemal Ebcioglu, and Dan Sahlin. Reducing Sweep Time for a Nearly Empty Heap, In *Proceeding of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 378-389, 2000.

[60] Junpyo Lee, Byung-Sun Yang, Suhyun Kim, SeungIl Lee, Yoo C. Chung, Heungbok Lee, Je Hyung Lee, Soo-Mook Moon, Kemal Ebcioglu, Erik Altman. Reducing Virtual Call Overheads in a Java VM Just-In-Time Compiler, *The 4th Annual Workshop on Interaction between Compilers and Computer Architectures*, pp. 21-33, 2000.

[61] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and

Yasunori Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming – ECOOP '00*, Volume 1850 of LNCS, Springer-Verlag, pp. 362-387, 2000.

[62] 首藤, 根山, 村岡. プログラマに単一マシンビューを提供する分散オブジェクトシステム, 情報処理学会論文誌：プログラミング研究会, Vol. 40, No. SIG 7 (PRO 4), pp. 66-79, 1998.

[63] Intel Architecture Software Develoer's Manual, Order Number 243191-001, Intel Corp., 1997.

[64] Intel Corp. IA-64 Architecture Software Developer's Manual, available at http://developer.intel.com/design/itanium/manuals/index.htm.

[65] Cathey May, Ed Silha, Rick Simpson, and Hank Warren. The PowerPC Architecture, Morgan Kaufmann Publishers, Inc., 1994.

[66] IBM Corp. ESA/390 Principles of Operation, available at http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/DZ9AR006/CCONTENTS.

[67] 古関, 小松. 非常に偏った条件分岐が存在するプログラムのデータフロー最適化, 情報処理学会論文誌：プログラミング研究会, Vol. 42, No. SIG2(PRO 9), pp. 26-36, 2001.

[68] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Lazy code motion, In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 224-234, 1992.

[69] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Optimal code motion, *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 5, pp. 777-802, 1995.

[70] Etienne Morel and Claude Renvoise. Global Optimization by suppression of partial redundancies, *CACM*, Vol. 22, No. 2, pp. 96-103, 1979.

[71] Motoshiro Kawahito, Hideaki Komatsu, Toshio Nakatani. Effective Sign Extension Elimination, In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 187-198, 2002.

[72] David Detlefs and Ole Agesen. Inlining of Virtual Methods, In *Proceedings of the 13th European Conference on Object-Oriented Programming – ECOOP '99*, Volume 1628 of LNCS, Springer-Verlag, pp. 258-278, 1999.

[73] Vugranam C. Sreedhar, Roy Dz-ching Ju, David M. Gillies, and Vatsa Santhanam. Translating Out of Static Single Assignment Form. In *Static Analysis Symposium*, Volume 1694 of LNCS, pp. 194-210, 1999.

[74] Philip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pp. 11-16, 1986.

[75] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 5, pp. 895-913, 1999.

[76] Gregory J. Chaitin. Register allocation and spilling via graph coloring, In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 17, No. 6, pp. 257-265, 1982.

[77] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead In C++ Programs, In *Proceeding of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages, pp. 397-408, 1994.

[78] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs, In *Proceedings of the 10th European Conference on Object-Oriented Programming – ECOOP '96*, volume 1098 of LNCS, Springer-Verlag, pp. 142-166, 1996.

[79] Urs Hölze. Adaptive Optimization for Self: Reconciling High Performance With Exploratory Programming, PhD thesis, Stanford University, 1994.

[80] Mary F. Fernandez. Simple and Effective Link-Time Optimization of Modula-3 Programs, In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 103-115, 1995.

[81] David F. Bacon and Peter F. Sweeny. Fast Static Analysis of C++ Virtual Function Calls, In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications,* pp. 324-341, 1996.

[82] Frank Tip and Jens Palsberg. Scalable Propagation-Based Call Graph Construction Algorithm, In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications,* pp. 281-293, 2000.

[83] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Garnon, and Charles Godin. Practical Virtual Method Call Resolution for Java, In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications,* pp. 264-280, 2000.

[84] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference, In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications,* pp. 146-161, 1991.

[85] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages, In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications,* pp. 91-107, 1995.

[86] Paul R. Carini, Hirini Srinivasan, and Michael Hind. Flow-Sensitive Type Analysis for C++, *IBM Research Report*, RC 20267, 1995.

[87] Etienne M. Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient Inference of Static Types for Java Bytecode, In Static Analysis Symposium, Volume 1824 of LNCS, pp. 199-219, 2000.

[88] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches, In *Proceedings of the 5th European Conference on Object-Oriented Programming – ECOOP '91*, volume 512 of LNCS, Springer-Verlag, pp. 21-38, 1991.

[89] David F. Bacon. Fast and Effective Optimization of Statically Typed Object-Oriented Languages, Ph.D. thesis, University of California at Berkeley, 1997.

[90] Bowen Alpern, Mark Charney, Jong-Deok Choi, Anthony Cocchi, and Derek Lieber. Dynamic Linking on a Shared-Memory Multiprocessor, In *Proceeding of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 177-182, 1999.

[91] Intel Corp. Intel Architecture Software Developer's Manual, order number 243192, 1997.

[92] Jong-Deok Choi, Manish Gupta, Muaricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Es-

cape Analysis for Java, In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pp. 1-19, 1999.

[93] IBM Corp. Jikes, available at http://oss.software.ibm.com/developerworks/opensource/jikes/project/.

[94] The Standard Performance Evaluation Corp., SPECjbb2000 Benchmarks, available at http://www.spec.org/osg/jbb2000/.

[95] Norman H. Cohen. Type-extension type tests can be performed in constant time, *ACM Transactions on Programming Language and Systems*, Vol. 13, No. 4, pp. 626-629, 1991.

[96] Jan Vitek, R. Nigel Horspool, and Andreas Krall. Efficient Type Inclusion Test, In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pp. 142-157, 1997.

[97] Bowen Alpern, Anthony Cocchi, and David Grove. Dynamic type checking in Jalapeno. In *USENIX Java Virtual Machine Research and Technology Symposium*, pp. 41-52, 2001.

[98] Aart Bik, Milind Girker, and Mohammad Haghighat. JIT Compilation of Java for Intel Architecture, *ACM 1999 Java Grande Conference Tutorial*, 1999.

[99] IBM Corp. Caribbean, available at http://www.alphaworks.ibm.com/tech/caribbean.

[100] Sun Microsystems Inc. HotJava Browser, available at http://java.sun.com/products/hotjava/index.1_0.html.

[101] ジャストシステム. 一太郎 ark, available at http://www.justsystem.co.jp/ark/.

[102] Rajiv Gupta. Optimizing array bound checks using flow analysis, *ACM Letters on Programming Languages and Systems*, 2(1-4): pp. 135-150, 1993.

[103] Blainey, R. J. Instruction scheduling in the TOBEY compiler, *IBM Journal of Research and Development*, Vol. 38, No. 5, pp. 577-594, 1994.

[104] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A methodology for benchmarking Java Grande applications. In *Proceedings of the ACM 1999 Conference on Java Grande*, pp. 81-88, 1999.

[105] Jose E. Moreira, Samuel P. Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing, *ACM Transactions on Programming Languages and Systems*, Vol. 22, No. 2, pp. 265-295, 2000.

[106] 石崎, 稲垣, 小松. Java プログラムにおいて例外による順序制約を投機的命令を用いて除去する方法, 情報処理学会論文誌：プログラミング研究会, Vol. 43, No. SIG8(PRO 15), pp. 87-97, 2002.

[107] Kazuaki Ishizaki, Tatsushi Inagaki, Hideaki Komatsu, Toshio Nakatani. Eliminating Exception Constraints of Java Programs for IA-64, In *Proceedings of the Eleventh International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 259-268, 2002.

[108] Scott A. Mahlke, William Y. Chen, Roger A. Bringmann, Richard E. Hank, Wen-Mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, Vol. 11, No. 4, pp. 376-408, 1993.

[109] Bich C. Le. An Out-of-Order Execution Technique for Runtime Binary Translators, In *Proceedings*

of the Eighth International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS-VIII), pp. 151-158, 1998.

[110] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting Beyond Static Scheduling in a Superscalar Processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 344-354, 1990.

[111] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, Vol. 7, No. 1, pp. 229-248, 1993.

[112] David I. August, Brian L. Deitrich, and Scott A. Mahlke. Sentinel Scheduling with Recovery Blocks. Computer Science Technical Report CRHC-95-05, University of Illinois, Urbana, 1995.

[113] Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, Chu-cheow Lim, John Ng, David Sehr. An Advanced Optimizer for the IA-64 Architecture, *IEEE Micro*, Nov.-Dec. 2000, pp. 60-68, 2000.

[114] Roy Dz-ching Ju, Kevin Nomura, Uma Mahadevan, and Le-Chun We. A Unified Compiler Framework for Control and Data Speculation. In *Proceedings of the 2000 Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 157-168, 2000.

[115] Rumi Zahir, Jonathan Ross, Dale Morris, and Drew Hess. OS and Compiler Considerations in the Design of the IA-64 Architecture, In Proceedings of the International Conference on Architectural Support for Programming Language and Operating Systems, pp. 212-221, 2000.

[116] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen-mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Nineth International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS-IX)*, pp. 183-193, 1994.

[117] Matthew Arnold, Michael Hsiao, Ulrich Kremer, and Barbara Ryder. Exploring the interaction between Java's implicitly thrown exceptions and instruction scheduling. *International Journal of Parallel Programming*, Vol. 29, No. 2, pp. 111-137, 2001.

[118] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 26-37, 1997.

[119] Manish Gupta, Jong-Deok Choi, and Michael Hind. Optimizing Java Programs in the Presence of Exceptions, In *Proceedings of the 14th European Conference on Object-Oriented Programming – ECOOP '00*, Volume 1850 of LNCS, Springer-Verlag, pp. 422-446, 2000.

[120] Intel Corp. Intel Itanium™ Processor Reference Manual for Software Optimization, http://developers.intel.com/design/ia64/downloads/245474.htm.

研究業績

●は筆頭著者であるもの。
○は筆頭著者でないもの。

論文誌（査読有）

● 石崎　一明, 稲垣　達氏, 小松　秀昭
Java プログラムにおいて例外による順序制約を投機的命令を用いて除去する方法
情報処理学会論文誌：プログラミング研究会, Vol. 43, No. SIG8(PRO 15), pp. 87-97, 2002.

● 石崎　一明, 安江　俊明, 川人　基弘, 小松　秀昭
コード書換えによる仮想メソッド呼び出しの直接 devirtualization
情報処理学会論文誌, Vol. 43, No. 1, pp. 124-136, 2002.

● Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani
Design, Implementation, and Evaluation of Optimizations in a Java(tm) Just-In-Time Compiler
Concurrency: Practice and Experience, Vol. 12, No. 6, pp. 457-475, 2000.

○ Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani
Overview of the IBM Java Just-in-Time Compiler
IBM Systems Journal, Vol. 39, No. 1, pp. 175-193, 2000.

● Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani
A Loop Transformation Algorithm for Communication Overlapping
International Journal of Parallel Processing, Vol. 28, No. 2, pp. 135-154, 2000.

● 石崎　一明, 小松　秀昭
分散メモリ並列計算機のためのコンパイラによる通信遅延隠蔽アルゴリズム
情報処理学会論文誌, Vol. 38, No. 9, pp. 1849-1858, 1997.

国際会議（査読有）

● Kazuaki Ishizaki, Tatsushi Inagaki, Hideaki Komatsu, Toshio Nakatani
Eliminating Exception Constraints of Java Programs for IA-64
In Proceedings of the Eleventh International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 259-268, 2002.

● Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, Takatani Nakatani
A study of Devirtualization Techniques for a Java(tm) Just-In-Time Compiler
In Proceedings of the ACM Conference object Oriented Programming Systems, Languages, and Ap-

plications, pp. 294-310, 2000.

- Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, Toshio Nakatani
  Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler
  In Proceedings of the ACM 1999 Conference on Java Grande, pp. 119-128, 1999.
- Kazuaki Ishizaki, Hideaki Komatsu, Toshio Nakatani
  An Algorithm for Automatic Detection of Loop Indices for Communication overlapping
  International Symposium on High Performance Computing
  Volume 1336 of Lecture Notes in Computer Science, Springer-Verlag, pp. 217-230, 1997.
- Kazuaki Ishizaki, Hideaki Komatsu
  A Loop parallelization Algorithm for HPF Compilers
  Eighth Annual Workshop on Language and Compilers for Parallel Computing, pp. 12.1-15, 1995
  Volume 1033 of Lecture Notes in Computer Science, Springer-Verlag, pp. 176-190, 1995.

国際会議（査読無）

- Kazuaki Ishizaki
  A study of Devirtualization Techniques for a Java(tm) Just-In-Time Compiler
  Dagstule Seminars 00451: Effective Implementation of Object-Oriented Programming Languages, 2000.

国内会議（査読有）

- 石崎　一明, 小松　秀昭
  分散メモリ並列計算機のためのコンパイラによる通信遅延隠蔽アルゴリズム
  並列処理シンポジウム JSPP 1996, pp. 339-346, 1996.

国内会議（査読無）

- 石崎　一明, 安江　俊明, 川人　基弘, 小松　秀昭
  コード書換えによる仮想メソッド呼び出しの直接 devirtualization
  情報処理学会プログラミング研究会, 2002.
- 石崎　一明, 川人　基弘, 今野　和浩, 安江　俊明, 竹内　幹雄, 小笠原　武史, 菅沼　俊夫, 小野寺　民也, 小松　秀昭
  Java Just-In-Time コンパイラにおける最適化とその評価
  電子情報通信学会技術研究報告, CPSY-99-64, pp. 17-24, 1999.
- 石崎　一明, 小松　秀昭
  ＨＰＦコンパイラにおける並列化手法
  情報処理学会研究報告, HPC-57-18, pp. 103-108, 1995.

○ 小笠原　武史, 石崎　一明, 小松　秀昭
ＨＰＦにおける実行時の通信解析オーバヘッドの削減手法
情報処理学会研究報告, HPC-57-19, pp. 109-114, 1995.

○ 郷田　修, 大澤　暁, 小松　秀昭, 菅沼　俊夫, 小笠原　武史, 石崎　一明, 中谷　登志男
ＨＰＦコンパイラの実装と評価
情報処理学会研究報告, HPC-57-20, pp. 115-120, 1995.

● 石崎　一明, 小松　秀昭
分散並列計算機のための並列性抽出法
電子情報通信学会技術研究報告, CPSY-384, pp. 97-104, 1994.

○ 石井　吉彦, 石崎　一明, 萩本　猛, 山名　早人, 村岡　洋一
並列処理システム-晴-における要素プロセッサ制御手順
情報処理学会第 43 回全国大会, 1991.

○ 山名　早人, 石崎　一明, 安江　俊明, 村岡　洋一
並列処理システム-晴-における条件分岐の先行評価制御方式
情報処理学会研究報告, ARC-89-19, 1991.

● 石崎　一明, 安江　俊明, 山名　早人, 村岡　洋一
先行評価に適した並列計算機のネットワーク構成
情報処理学会研究報告, ARC-89-20, 1991.

● 石崎　一明, 石井　吉彦, 萩本　猛, 山名　早人, 村岡　洋一
並列処理システム-晴-の要素プロセッサ制御方式
情報処理学会第 42 回全国大会, 1991.

● 石崎　一明, 山名　早人, 村岡　洋一
並列処理システム-晴-のパケット管理機構の構成
情報処理学会研究報告, ARC-85-2, pp. 7-14, 1990.

● 石崎　一明、神舘　淳, 山名　早人, 村岡　洋一
科学技術計算用並列処理システム-晴-における関数実行方式
電子情報通信学会研究報告, CPSY-90-2, pp. 7-14, 1990.

研究報告（Research Report）

● Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, Toshio Nakatani
A Study of Devirtualization Techniques for a Java Just-In-Time Compiler
IBM Research Report RT-0352, 2000.

● Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, Toshio Nakatani
Optimizations for Reducing the Runtime Overhead of the Java Language in a Just-In-Time Compiler
IBM Research Report RT-0285, 1998.

その他

学会誌

○ 小松　秀昭, 菅沼　俊夫, 小笠原　武史, 石崎　一明, 郷田　修
ＳＰ２のためのＨＰＦコンパイラにおける最適化技術
情報処理, Vol. 38, No.2, pp. 100-104, 1997.