C 言語への変換による Java アプリケーションの高速化に関する研究

平成十五年度

千葉 雄司

概要

オブジェクト指向プログラミング言語 Java は生産性の高さなど様々な利点をもち，産業界に普及しつつある．一方で Java で開発したソフトウェアには実行速度が遅いという欠点があり，この欠点の克服を目的として多くの研究がなされてきた．実行を高速化する手段は大別して 2 種類あり，1 つは Java に特化したハードウェアを使う方法で，もう 1 つはソフトウェアを使う方法，すなわちコンパイラで最適化する方法だが，どちらにも共通する問題にコストがある．組込み機器のようにコストへの要求が厳しい環境では，Java に特化したハードウェアの追加は出荷台数に比例したコストがかかるため容易でなく，かといってコンパイラの開発も簡単ではない．組込み機器の分野では多様な OS やプロセッサを利用するが，その個々に対応するコンパイラを逐一開発すると膨大なコストがかかる．そこで，組込み機器向けコンパイラの開発にあたっては，効率的な開発手法が必要になる．

安価な Java 向けコンパイラの開発方法に，Java から C 言語へのトランスレータを使う方法がある．この方法では，コンパイラを Java から C 言語へのトランスレータと既存の C コンパイラから構成し，既存の C コンパイラに古典的最適化や機械コード生成を任せることで開発コストを軽減する．本論文では Java から C 言語へのトランスレータの実現方法について述べる．まず，動的ロードの実現方法を提案し，次に，動的ロードに関連した最適化技術であるクラス初期化検査の除去と，仮想メソッド呼出しの高速化を提案する．最後に，Java から C 言語へのトランスレータで例外処理を実現する方法を検討する．

最初にとりあげる動的ロードは，Java アプリケーションの構成要素であるクラスファイルを，実行時に読み込む機能である．クラスファイルは，Java で記述したソースコードをコンパイルした結果得られるもので，ソースコード上にあるクラスの内容を中間形式に変換した形で格納する．実行時には，Java の実行時システムがクラスファイルを動的ロードして，その内部に格納してあるメソッドを実行する．クラスファイル中のメソッドはバイトコードと呼ぶ機種非依存の中間形式で表現してあり，汎用プロセッサでは直接実行できない．インタプリタによる解釈実行は可能だがオーバヘッドが大きい．このオーバヘッドを軽減する手段の 1 つが，バイトコードコンパイラと呼ぶコンパイラでバイトコードを機械コードに変換し，プロセッサで直接実行可能にする方法である．

バイトコードコンパイラには動的なものと静的なものがある．動的バイトコードコンパイラはプログラムの実行時にバイトコードを機械コードに変換し，静的バイトコードコンパイラは実行を開始するより前にあらかじめ変換しておく．静的バイトコードコンパイラ

が生成した機械コードは実行時に使用できるとは限らない．なぜなら静的コンパイル後にクラスファイルを更新すると，更新前のクラスファイルから生成した機械コードが無効になるからである．Java の言語仕様は動的ロードしたクラスファイルの内容に従ってプログラムを実行するよう規定しており，動的ロードしたクラスファイルが更新後のものであれば，更新前のクラスファイルから生成した機械コードは無効であり使用できない．

Java から C 言語へのトランスレータと既存の C コンパイラからなるコンパイラは，静的バイトコードコンパイラの一種であり，したがって Java から C 言語へのトランスレータを使って生成した機械コードも実行時に使用できるとは限らない．しかし，従来の Java から C 言語へのトランスレータの実現はクラスファイルの更新に応じて機械コードを無効にする機能を備えておらず，一部の Java アプリケーションを正常に実行できないという問題を抱えていた．

この問題の解決を目的として，本論文では，まず，機械コードの有効性を実行時に確認し，確認がとれた場合に限って機械コードをリンクし，プログラムの実行に使うシステムを提案する．次に，動的ロードのオーバヘッドを軽減する最適化を 2 つ提案する．1 つ目の最適化は，クラスを動的ロードするタイミングを決定するための検査であるクラス初期化検査を除去し，2 つ目の最適化は，仮想メソッド呼出しの高速化において動的ロードに配慮し，機械コードの有効性を実行時に確認するまでにかかるオーバヘッドを軽減する．

また，本論文では，例外処理の実現方法を検討する．C 言語へのトランスレータで例外処理を実現する方法として，これまでに setjmp 法と 2 返戻値法が提案されているが，どちらが優れているか定量的に評価した研究がない．そこで，本論文で評価をおこない，今後のトランスレータの実現において例外処理の実現方法を選択する際の指針を示す．さらに，setjmp 法と 2 返戻値法向けの最適化を提案し，評価する．

実用的な Java アプリケーションを集めたベンチマークである SPECjvm98 を用いて評価した結果，本論文で提案するクラス初期化検査の除去により，平均 45% 実行を高速化できることが判った．また，仮想メソッド呼出しの高速化により，最大 9.8%実行速度を改善できることが判った．さらに，例外処理の実現について，2 返戻値法による例外処理のオーバヘッドを，最適化の適用により平均 1.4% 程度にできることが判った．

Abstract

Java is an object-oriented programming language with many advantages such as high productivity, and has been used in many fields. But the performance of Java applications is not necessarily good, and many studies to improve the performance have been carried out. One way to improve performance is to use an optimizing compiler, but its development has a high cost. Cost-effectiveness is important in the development of compilers for Java for embedded machines, because embedded machines use many kinds of operating systems and processors, which we call platforms, and a compiler needs to be developed for each platform.

One cost-effective way to develop compilers for Java is based on developing a translator from Java to C language, which we call Java2C translator, and then applying existing C compilers with many kinds of traditional optimizations. This paper proposes two implementation techniques for Java2C translator. First, we propose a method to implement dynamic loading and two optimizing techniques to reduce overheads for the dynamic loading. Second, we propose an implementation of exception handling.

Dynamic loading is a feature that loads class files at runtime. A class file is an element of Java application and contains contents of a class in Java source code in a platform-independent manner. A runtime system of Java dynamically loads class files and executes methods in them. The body of a method in a class file is represented in bytecode, which is platform-independent. For the execution of bytecode, the runtime system can use an interpreter but it results in poor performance. A bytecode compiler improves performance by compiling bytecode into machine-dependent code, that the processor can execute directly. A Java2C translator is one type of bytecode compiler.

Problems in the implementation of dynamic loading by a Java2C translator lies in the fact that dynamic loading must consider invalidated machine code which had been generated using the Java2C translator. Because a Java2C translator translates bytecode in class files before execution of the Java application, the generated machine code becomes invalid if class files referred in the translation are updated after the translation. At runtime, the runtime system must confirm validity of the machine code before using them.

This paper shows an implementation of a runtime system that checks (or confirms) the validity of the machine code before using them, and a Java2C translator that generates information for the validity confirmation. This paper also presents two optimizations related to the validity confirmation. One optimization implements virtual calls in a manner that reduces overhead for the validity confirmation, and the other optimization removes class initialization tests using the validity confirmation. Class initialization test is code to initialize a class if the class is not initialized, and is inserted in many points of Java application. The result of applying SPECjvm98 benchmarks showed that removal of class initialization tests improve performance by 45% on average, and optimization for virtual calls improve performance up to 9.8%.

This paper also shows an implementation of exception handling in a Java2C translator. There are two implementation methods for exception handling for a Java2C translator: one is setjmp method and the other is two-return-values method. This paper presents optimizations for each method and compares them quantitatively using SPECjvm98 benchmarks. The result of applying SPECjvm98 benchmarks showed two-return-values method performs better and its overhead to implement exception handling is as small as 1.4%.

目 次

第1章 はじめに 1

1.1 Java の特徴 1

1.2 Java の実行モデル 2

1.3 Java アプリケーションの実行高速化手段 4

1.3.1 これまでに開発された高速化手段 4

1.3.2 Java2C トランスレータの位置付け 5

1.4 Java2C トランスレータを実現する上での問題点 6

1.5 本論文の目的 7

1.6 本論文の構成 7

第2章 関連研究 8

2.1 Java2C トランスレータ 8

2.2 動的ロード 8

2.2.1 動的ロードが静的コンパイラにもたらす問題 8

2.2.2 過去の Java2C トランスレータにおける動的ロードの実現 9

2.3 クラス初期化検査 9

2.3.1 クラス初期化検査の定義 9

2.3.2 クラス初期化検査のオーバヘッド削減 11

2.4 仮想メソッド呼出し 12

2.4.1 仮想メソッド 12

2.4.2 ディスパッチ表法 14

2.4.3 I-call if 変換 16

2.4.4 提案手法 17

2.5 例外処理 17

2.5.1 Java のソースコード上における例外処理の記述 18

2.5.2 既存の例外処理の実現方法 19

2.5.3 Java2C トンランスレータで利用可能な例外処理の実現方法 21

2.6 null 検査 21

2.6.1 明示的な null 検査 21

2.6.2 暗黙の null 検査 22

2.6.3 null 検査の実現に関連した Java2C トランスレータの問題 22
2.7 スタックトレース 23
2.7.1 リターンアドレスを使ったスタックトレースの算出方法 23
2.7.2 Java2C トランスレータにおけるスタックトレースの算出方法 . . . 23
2.8 スタック溢れ検査 23
2.9 モニタ 24
2.10 ごみ集め 24
2.10.1 ごみ集めとは 25
2.10.2 Java2C トランスレータにおけるごみ集めの実現 25
2.11 浮動小数点演算の精度保証 25

第 3 章 動的ロードの実現 **27**

3.1 JeanPaul の構成 27
3.1.1 Java アプリケーションのコンパイル 27
3.1.2 Java アプリケーションの実行 29
3.2 JeanPaul における動的ロードの実現 32
3.2.1 クラスを動的ロードするタイミング 32
3.2.2 仮定クラスの集合を計算する手順 34
3.3 Java2C トランスレータが適用する最適化 36
3.3.1 フィールドおよびメソッドの静的解決 36
3.3.2 冗長なキャストの除去 41
3.3.3 仮想メソッド呼出しの高速化 43
3.3.4 インライン展開 44
3.3.5 スタックトレース管理コードのオーバヘッド軽減 44
3.3.6 冗長な null 検査の除去 51
3.3.7 ループのピーリング 52
3.3.8 冗長な配列添字検査の除去 52

第 4 章 クラス初期化検査の除去が実行速度に与える影響 **55**

4.1 評価環境 55
4.2 実行速度への影響 56
4.3 リンクの遅延から生じる影響 57

第 5 章 I-call if 変換の実現 **59**

5.1 I-call if 変換が仮定クラスに与える影響 59
5.1.1 I-call if 変換を適用したコードに関する注意事項 59
5.1.2 追加する仮定クラスの違い 60

5.2 4種類のI-call if変換の使い分ける方法 62
5.2.1 クラスチェック vs. メソッドチェック 62
5.2.2 限定的 vs. 非限定的 66
5.3 メソッドチェック変換とCコンパイラの問題 68

第6章 例外処理の実現 **70**
6.1 setjmp法 70
6.1.1 setjmp法による例外処理の実現 70
6.1.2 setjmp法向けの最適化 74
6.2 2返戻値法 77
6.2.1 2返戻値法による例外処理の実現 77
6.2.2 2返戻値法向け最適化 80
6.3 評価 83
6.3.1 setjmp法向け最適化の評価 83
6.3.2 2返戻値法向け最適化の評価 86
6.3.3 setjmp法と2返戻値法の比較 86
6.3.4 2返戻値法における例外処理のオーバヘッド 89

第7章 結論 **91**
7.1 まとめ 91
7.1.1 動的ロードの実現 91
7.1.2 例外処理の実現 92
7.2 今後の課題と展望 93
7.2.1 動的ロード 93
7.2.2 例外処理の実現 93

謝辞 **94**

参考文献 **95**

研究業績 **103**

図 目 次

1.1 Java の実行モデル 2

2.1 クラス初期化検査の除去 10
2.2 仮想メソッド呼出しとその実現 13
2.3 クラスとメソッド表 13
2.4 ディスパッチ表 15
2.5 例外処理の例 18

3.1 Java2C トランスレータつき Java VM “JeanPaul” の構成 28
3.2 クラスを表すデータ構造 28
3.3 クラスの動的ロードおよび初期化処理 30
3.4 インスタンスフィールド参照 37
3.5 クラスフィールド参照の高速化 39
3.6 メソッドの直接呼出し 40
3.7 冗長なキャスト 42
3.8 スタックトレース管理コード 46
3.9 補償コード領域への移動 48
3.10 インライン展開を施したメソッドでのスタックトレースの実現 50
3.11 ループピーリングと冗長な null 検査および配列添字検査除去 53

4.1 クラス初期化検査の除去が実行速度に与える影響 57
4.2 リンクの遅延が実行速度に与える影響 58

5.1 クラスおよびメソッドチェック変換のコード 62
5.2 実行速度によるクラスおよびメソッドチェック変換の比較 64
5.3 限定的 I-call if 変換と非限定的 I-call if 変換の比較 67
5.4 メソッドチェック変換の非効率的な実現 69

6.1 setjmp 法による変換結果 71
6.2 例外処理方式の変換 76
6.3 2 返戻値法による変換結果 78
6.4 スレッド固有の資源を収める構造体型の定義 79

6.5 下方移動による例外発生検査の集約 82
6.6 setjmp 法向け最適化が実行速度に与える影響 84
6.7 例外発生検査の下方移動が実行速度に与える影響 87
6.8 setjmp 法と 2 返戻値法の比較 87
6.9 例外発生検査のオーバヘッド 89
6.10 マイクロベンチマーク 90

表目次

3.1 JeanPaul の Java2C トランスレータが実施する最適化一覧 36

4.1 SPECjvm98 を構成するベンチマーク 56

5.1 I-call if 変換の分類 60
5.2 静的コンパイル済みコードのみによる実行時間 65
5.3 仮想メソッド呼出しにおける誤分岐回数の分布 65
5.4 final 宣言した仮想メソッドの呼出し箇所数 68

6.1 構造体型 ExecEnv がもつモニタスタックに関連したメンバ 73
6.2 `volatile` 宣言の最小化による `volatile` 宣言数の変化 85
6.3 `enterTry()` の実行回数と最適化の影響 85
6.4 例外発生検査の実行回数 88
6.5 マイクロベンチマーク実行結果 90

第1章　はじめに

本論文ではオブジェクト指向プログラミング言語Java^{TM}[1]で開発したアプリケーションの実行速度を，Java から C 言語へのトランスレータを使って高速化する方法について述べる．本論文では Java から C 言語へのトランスレータを **Java2C トランスレータ**と記述する．本章では論文の冒頭にあたり，まず 1.1 節で Java の特徴を示し，次に 1.2 節で Java の実行モデルを示す．続く 1.3 節では Java アプリケーションの実行を高速化する手段について概観し，数ある高速化手段の中における Java2C トランスレータの位置付けを確認する．1.4 節では Java2C トランスレータを実現する上での問題点を示し，1.5 節で本論文で解決する問題を明確にする．最後に 1.6 節で本論文の構成を示す．

1.1　Java の特徴

Java [Gosling96] は，プラットフォーム非依存性や豊富なライブラリなど様々な利点を備え，サーバから組込み機器まで，産業界の幅広い分野で利用が進んでいるオブジェクト指向プログラミング言語である．ここでプラットフォームとは OS （Operating System）およびプロセッサを意味し，プラットフォーム非依存とはどのプラットフォームでも動作することを意味する．

プラットフォーム非依存性は大きな利点である．たとえば，多種多様な機械を接続したネットワーク上にクライアントサーバシステムを構築する場合について考える．利便性を考えると，ネットワークに接続したあらゆる機械をクライアントとして利用可能にするのが望ましいが，クライアントとしての動作に必要なソフトウェアを全プラットフォーム向けに開発し，維持していくには手間がかかる．ここで，プラットフォーム非依存の Java を使えば，1 つのプラットフォーム向けに開発したソフトウェアを全てのプラットフォームで利用できるので，ソフトウェアの開発および維持管理にかかるコストを削減できる．また，プラットフォーム非依存性には，家庭電化製品など，組込み機器の開発期間を短縮する効果もある．なぜなら，プラットフォーム非依存なソフトウェアの開発はハードウェアの完成を待たずに始めることができるからである．

[1]Java は米国 Sun MicroSystems, Inc. の商標である．

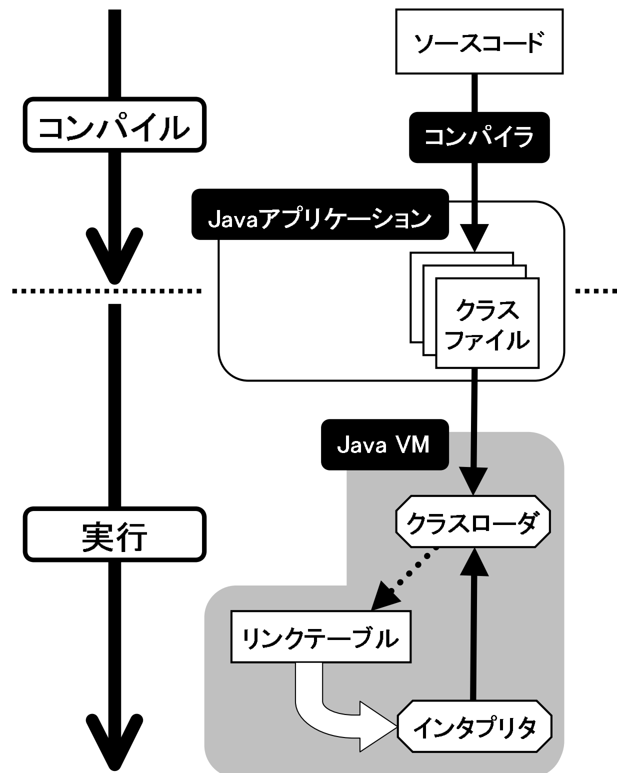

図 1.1: Java の実行モデル

1.2 Java の実行モデル

Java ではプラットフォーム非依存性を実現するために仮想機械を使う．Java における仮想機械の役割を明らかにするために，図 1.1 に Java の実行モデル，すなわち Java で記述したソースコードをコンパイル，実行する手順を示す．Java ではソースコードをコンパイルすると，ソースコード上にある個々のクラスの内容（フィールドやメソッド）は，コンパイラによってプラットフォーム非依存な形式に変換されて，**クラスファイル**と呼ばれるファイルの中に記録される．たとえば，メソッドの内容は，プラットフォーム非依存の**バイトコード**という中間語に変換された形で記録される [Lindholm96].

Java アプリケーションの実行時にはクラスファイル内に記録したメソッドが実行されるが，クラスファイル内におけるメソッドの表現形式はバイトコードであり，機械コードではないので汎用プロセッサでは直接実行できない．そこで，Java アプリケーションの実行にあたっては，バイトコードを実行できる仮想機械を起動して，仮想機械に実行させる．ここで利用する Java 向け仮想機械を **Java 仮想機械**（Java Virtual Machine, Java VM）と呼ぶ．図 1.1 の Java VM はインタプリタを搭載しており，インタプリタを使ってバイトコードを解釈実行する．なお，1.3 節で述べるように，インタプリタとは異なる手段でバイトコードを実行する Java VM もある．

図 1.1 の Java VM 内にある，**クラスローダ**や**リンクテーブル**の役割を示すために，Java VM の起動からインタプリタが `main()` の実行を開始するまでの起動手順について述べる．

1. 多くのプラットフォームでは，Java アプリケーションの実行を開始するために，ユーザはコンマドラインに次のようにコマンドを入力する．ここで，**java** は Java VM を起動するコマンドである．

   ```
   % java Application
   ```

2. この入力によって起動した Java VM は引数で指定した名前（ここでは **Application**）のクラスに対応するクラスファイルを読み込む．クラスファイルを読み込む役割を果す Java VM 内のプログラム片をクラスローダと呼ぶ．クラスローダは読み込んだクラスの親クラスも，まだ読み込んでいなければ順次再帰的に読み込む．そして，読み込んだクラスファイル内に格納してあるクラスを表すデータ構造を作成し，リンクテーブルに登録する．登録したデータ構造はリンクテーブルの一部となる．クラスを表すデータ構造の中には，クラスが定義するフィールドおよびメソッドを表すデータ構造を含む．リンクテーブルはクラス，メソッド，フィールドの名前と，その内部表現であるデータ構造や機械コードを対応付ける辞書の役割を果す．
3. 引数で指定したクラスと，その全親クラスを読み込み，リンクテーブルに登録したら，次に，読み込んだクラスを親クラスから子クラスに向って順次，初期化する．ここで**クラスの初期化**とはクラスファイル内にある "< **clinit** >" という名称のメソッドを実行することを意味する．< **clinit** > 内には，次のコードが入っている．

   - クラスフィールドに初期値を与えるコード
   - ソースコード上でクラス初期化時に実行するように記述しておいたコード

4. クラスの初期化が終ったら，引数で指定したクラスの **main()** メソッドの実行を開始するように，Java VM がインタプリタに指示する．

   指示を受けたインタプリタはリンクテーブルを参照して実行すべき **main()** という名前のメソッドを求め，そのバイトコードを解釈実行する．インタプリタがメソッドを実行している過程で別のメソッドを呼び出すことになった場合にも同様に，リンクテーブルを参照して実行すべきメソッドの所在を求める．

   ここで，他クラスのメソッドを呼び出す場合には，そのクラスがまだ読み込まれていないことから，リンクテーブルに呼出し対象のメソッドが登録されていない場合がある．そのような場合には，インタプリタがクラスローダに指示して呼出し対象のメソッドを定義するクラスを読み込ませ，初期化した上で読み込んだクラス内のメソッドを呼び出す．このように Java ではアプリケーションを構成するクラスをプログラムの実行中に必要に応じて順次読み込むが，実行時にクラスを読み込む機能を**動的ロード**と呼ぶ．

1.3 Java アプリケーションの実行高速化手段

1.2 節で述べたように，Java ではバイトコードで表現されたプログラムを実行する．バイトコードの実行手段として，インタプリタを使うことは可能だが，実行速度が遅い．実行速度の改善は，負荷が集中するサーバや演算能力に劣る組込み機器にとって切実な要求であり，これまでに Java アプリケーションの実行速度を改善する様々な試みがあった．本節ではまず，これまでに提案された高速化方法を概観する．次に，本論文で述べる Java2C トランスレータが，どのような用途に適した高速化方法か示す．

1.3.1 これまでに開発された高速化手段

実行速度を改善する方法は大きくわけて 2 つあり，1 つはハードウェアを使う方法で，もう 1 つはソフトウェア的な工夫によるものである．ハードウェアを使う方法は組込み機器向けプロセッサが採用しており，バイトコードを直接実行する命令を備えるプロセッサも存在する [Berekovic97, Ton97, Chang98, Ton00, Ton01, Kim00, Kim01, Aoki01, O'Connor97, McGhan98, ElKharashi00-1, ElKharashi00-2, Vijaykrishnan98, 木村 02]．Java に特化したハードウェアを持たない汎用的なプロセッサを使うプラットフォームでは，ソフトウェア的な工夫で実行速度の改善を図る．具体的には，インタプリタをチューニングするか [緒方 02]，コンパイラを使う [Suganuma00, Paleczny99, Howard97, Muller97, Proebsting97, Hsieb97, Antoniu01]．

コンパイラはバイトコードを機械コードに変換し，プロセッサで直接実行可能にすることで実行速度を改善する．また，変換の過程で最適化を施すことで，更なる高速化を図る．現在市場にある Java VM のうち，PC（Personal Computer）やサーバなど一定以上の計算機資源（演算能力と記憶容量）をもつプラットフォーム向けのものでは主に**動的コンパイラ**という種類のコンパイラを使う [Suganuma00, Paleczny99]．

コンパイラには，動的コンパイラと**静的コンパイラ**の 2 種類がある．動的コンパイラと静的コンパイラの違いは，プログラムをコンパイルするタイミングにある．静的コンパイラでは，プログラムを実行開始以前にあらかじめコンパイルし，動的コンパイラでは，プログラムの実行中にコンパイルする．Java VM に附属の動的コンパイラは，クラスローダが動的ロードしたクラスファイル内のバイトコードを機械コードに変換する．動的コンパイラには，実行履歴など，実行時に得られる情報を利用しつつコンパイルできるという利点があるが，一方でプログラムの実行と並行してコンパイル作業をおこなうため，相応の計算機資源を必要とするという欠点もある．昨今の PC は高速で動的コンパイラの駆動に十分な計算機資源を持つが，計算機資源の乏しい組込み機器では，プロセッサの演算能力の不足や，コンパイラを記憶するだけの記憶容量がないといった事情から動的コンパイラを利用できない場合も多い．市場に出回っている組込み機器向け Java VM の中には動的コンパイラを持たず，インタプリタのみでプログラムを実行するものも少なくない．

組込み機器向け Java VM がコンパイラを持たない理由は計算機資源の不足だけではない．コンパイラの開発費用も問題である．本論文執筆時点において，組込み機器向け Java VM には，全体として，ある程度の大きさの市場がある．しかし，組込み機器の分野では多彩なプロセッサや OS を利用しており，Java VM はそれぞれのプラットフォームごとに必要になるが，個々のプラットフォームあたりの市場は，必ずしも大きくない．このような環境下では，特定のプラットフォーム向けに膨大な費用を投資してコンパイラを開発しても，投資を回収することは難しい．したがって，組込み機器用の Java 向けコンパイラを開発する場合には，たとえばコンパイラの設計にあたって，複数のプラットフォームをサポートできるよう可搬性に配慮するといった工夫が必要になる [川本 02]．複数のプラットフォームをサポートすれば，それだけ市場は大きくなり，投資を回収できる可能性が増す．

1.3.2 Java2C トランスレータの位置付け

本論文で述べる Java2C トランスレータは，計算機資源やコンパイラ開発費用の問題といった組込み機器固有の問題を考慮した場合に有効な Java 向け静的コンパイラの開発手段である．この静的コンパイラでコンパイルをおこなうには，まず，Java2C トランスレータで Java のプログラム（ソースコードあるいはバイトコード）を C ソースコードに変換し，次に C コンパイラを適用して機械コードを得る．

コンパイラを Java2C トランスレータと既存の C コンパイラから構成することの目的は，コンパイラの可搬性を高め，コンパイラの開発工数（作業量）を軽減することにある．この構成方法では，単一の Java2C トランスレータを開発するだけで全てのプラットフォーム向けの Java 向けコンパイラが完成する．なぜなら，Java2C トランスレータが出力する C ソースコードを機種非依存にすれば，あとはほとんどのプラットフォームが提供する既存の C コンパイラを適用することにより，各プラットフォーム向けの機械コードが得られるからである．また，既存の C コンパイラが持つ最適化機能を利用することで，Java2C トランスレータ向けに新規開発する最適化機能の数を削減できる．トランスレート先を C 言語以外のプログラミング言語にしない理由は，C 言語が最も多くのプラットフォームでサポートされているからである．

Java2C トランスレータを動的コンパイラとして使うことは難しい．なぜなら Java2C トランスレータと組合せて使う既存の C コンパイラが動的コンパイラとしての利用を前提とした設計になっておらず，たとえばコンパイルに膨大な資源を使ったり，コンパイルに長い時間がかかったりするといった問題をおこしうるからである．

したがって，Java2C トランスレータを使って開発できるコンパイラは静的コンパイラになる．静的コンパイラは次の理由から組込み機器向けのコンパイル手段として適当だといえる．

- 静的コンパイラでは，実行時にコンパイルすることから生じる記憶容量的または演算能力的な負荷を回避できる．つまり，プログラムを実行開始以前にあらかじめコンパイルしておくので，コンパイラのプログラムを組込み機器の希少な記憶領域に格納する必要がない．また，組込み機器の貧弱なプロセッサでコンパイルをおこなう必要もなく，PC などの高速なプロセッサを使い，しかも時間をかけて十分に最適化を施しつつコンパイルできる．動的コンパイラではコンパイルに時間をかけると，その間，プログラムの実行が停止して応答性能が劣化するといった障害がおきることから，長時間を要する最適化処理を適用しにくい．

Java2C トラスレータを利用するコンパイラの開発方法は，開発工数が少ないことから，Java アプリケーションの挙動を調査研究するためのコンパイラを開発する場合にも適している．我々は Java アプリケーションの挙動に関する調査研究および，組込み機器を始めとする多くのプラットフォームに安価な Java 向けコンパイラを提供することを目的として，Java2C トランスレータを開発，製品化した．

1.4 Java2C トランスレータを実現する上での問題点

Java2C トランスレータを開発する上での問題は，Java と C 言語の違いから生じる．Java 固有の機能（Java にあって C 言語にない機能）の変換にあたっては，その機能を C 言語で実現する方法が問題になる．Java 固有の機能を次に列挙する．

1. 動的ロード
2. クラス初期化検査
3. 仮想メソッド呼出し
4. 例外処理
5. null 検査
6. スタックトレース
7. スタック溢れ検査
8. モニタ
9. ごみ集め
10. 浮動小数点演算の精度保証

列挙した機能のうち，1 の動的ロードについては 1.2 節で述べた．2 以降の機能については次章で述べる．列挙した機能に関係しない文は簡単にトランスレートできる．たとえば Java のソースコードにおける文 `short x = 1;` は，C 言語でも `short x = 1;` になる．

1.5 本論文の目的

本論文の目的は，1.4 節で列挙した 1 〜 4 の機能を Java2C トランスレータで実現する方法を提案することにある．5 〜 10 の機能を実現する方法についても述べるが，これらの実現方法については過去に提案があり，我々が開発した Java2C トラスンレータにおける 5 〜 10 の機能の実現も過去の研究で提案された方法に準じる．

1 の動的ロードは，過去の Java2C トランスレータ [Howard97, Muller97, Proebsting97, Hsieb97, Antoniu01] が実現できなかった機能である．このため過去の Java2C トランスレータには，一部の Java アプリケーションを正常に実行できないという問題があった．本論文では Java2C トランスレータにおいて動的ロードを実現する方法を提案する．動的ロードなど言語仕様が規定する機能をきちんと実現し，言語仕様に準拠することは，ユーザが安心して利用できる商用コンパイラを開発する上での必須事項である．

本論文では，動的ロードの実現方法を提案するだけでなく，提案した動的ロードの実現と連携して，2 のクラス初期化検査と 3 の仮想メソッド呼出しを最適化する方法を示す．これらの最適化には無視できない効果があり，動的ロードとあわせて実現する価値がある．実用的 Java アプリケーションを構成要素とするベンチマークプログラム SPECjvm98 [SPEC98] を使って評価した結果によると，クラス初期化検査の最適化に平均 45%，仮想メソッド呼出しの最適化に最大 9.8%，実行を高速化する効果があった．

また本論文では 4 の例外処理について，Java2C トランスレータで利用可能な例外処理の実現方法である setjmp 法と 2 返戻値法の優劣を SPECjvm98 を使って定量的に比較する．このような比較は過去に存在せず，比較結果は今後，例外処理の機能を提供するプログラミング言語から C 言語へのトランスレータを開発する場合に活用できる．評価の結果，2 返戻値法による実現の方がベンチマークの実行を 1.5% 高速化できることが判った．

1.6 本論文の構成

本論文の構成について述べる．まず，2 章で 1 〜 10 の機能の実現方法に関する過去の研究を概観する．次に，3 章において我々が開発した Java2C トランスレータを含む Java VM である JeanPaul の全体像を示し，Java2C トランスレータにおける動的ロードの実現と，クラス初期化検査向けの最適化について述べる．また，JeanPaul の Java2C トランスレータを使った Java アプリケーションのコンパイル，実行方法を示し，Java2C トランスレータに実現した最適化を概観する．続く 4 章においてクラス初期化検査向け最適化の効果を評価し，5 章で仮想メソッド呼出しの実現方法について述べる．6 章では例外処理の実現方法について検討する．7 章は結論である．

第2章　関連研究

Javaから C 言語へ変換する上での問題は動的ロードやごみ集めなど，Java 固有の機能を C 言語で実現する方法にあるが，いくつかの問題については過去の研究が既に解決策を示している．そこで，本章では過去の研究について概観する．まず過去の Java2C トランスレータの実現を示し，続いて 1.4 節にあげた 10 個の Java 固有の機能について順次，過去の研究によって提案された実現方法を示す.

2.1　Java2Cトランスレータ

C 言語を中間語とするコンパイル方式は古くから存在し，たとえば FORTRAN や Lisp, C++，Smalltalk といった様々なプログラミング言語から C 言語へのトランスレータが過去に発表されている [Levy95, Yuasa90, Cameron92, Ingalls00]．Java から C 言語へのトランスレータも研究用から商用まで，これまでいくつか発表があった [Muller97, Proebsting97, Howard97, Hsieb97, Antoniu01]．過去に発表された Java2C トランスレータには共通の問題がある．それは，動的ロードを実現していないことである．

2.2　動的ロード

動的ロードとは，1.2 節で指摘したように，実行時にクラスを読み込む機能である．本節ではまず，動的ロードが静的コンパイラにもたらす問題を示す．次に，過去の Java2C トランスレータにおける動的ロードの実現が不完全であったことを指摘する．

2.2.1　動的ロードが静的コンパイラにもたらす問題

Java の言語仕様は，動的ロードしたクラスファイルの内容にしたがってプログラムを実行するよう規定しているが，この規定は静的コンパイラにとって実現上の問題点となる．なぜなら，この規定は静的コンパイル済みコードを無効にすることがあるからである．すなわち，静的コンパイル作業後にクラスファイルが更新されると，静的コンパイル済みコードが無効になる．クラスファイルが更新された場合，実行時に動的ロードされるのは更新後のクラスファイルである．このとき，更新前の古いクラスファイルを静的コンパイルして得た機械コードは無効になり，プログラムの実行に使えなくなる．

静的コンパイラを利用する Java VM で動的ロードを実現するためには，実行時に静的

コンパイル済みコードが有効か確認してから Java VM にリンクし，プログラムの実行に利用すればよい．静的コンパイル済みコードが有効である条件は，静的コンパイル時に参照した全クラスファイルが，動的ロードしたものと同一であることである．本論文の 3 章では，実行時に静的コンパイル済みコード の有効性を確認してからリンクする機能の実現方法を提案する．

2.2.2　過去の Java2C トランスレータにおける動的ロード の実現

Java2C トランスレータを含め，過去の Java 向け静的コンパイラに関する研究で，動的ロード の実現方法について言及したものはない．過去の Java 向け静的コンパイラの実現はいずれも言語仕様に準拠せず，静的コンパイル済みコード を無条件に利用可能であるとみなして Java VM に静的リンクしていたため，一部の Java アプリケーションを正常に実行できなかった．ただし，静的コンパイルしなかったクラスを動的ロード 可能にする Java VM は存在した．この Java VM では，クラスが静的コンパイル済みか実行時に調べ，静的コンパイル済みではないならば動的ロードして，インタプリタ [Muller97] あるいは動的コンパイラ [Howard97] を使って実行する．

2.3　クラス初期化検査

本節ではまず，クラス初期化検査が何か明らかにし，次に，クラス初期化検査のオーバヘッド削減を目的として，過去に提案された技法を紹介する．

2.3.1　クラス初期化検査の定義

クラス初期化検査とは，クラスが初期化済みかど うかを検査し，初期化済みでなければ初期化する動作である．クラス初期化検査の役割は，実行時にクラスを初期化するタイミングを調節することにある．1.2 節で述べたように，Java では実行時にクラスファイルをオンデマンドに動的ロードして初期化する．Java の言語仕様はクラスファイルを動的ロード するタイミングは規定していないが，初期化するタイミングは規定している．具体的には，クラスを初めて参照した際に初期化する必要がある．クラスを初めて参照し うる動作は次の 3 つである．

1. クラスフィールド参照
2. クラスインスタンス生成
3. クラスメソッド呼出し

言語仕様が規定する通りのタイミングでクラスを初期化する方法の 1 つは，クラスを初めて参照し うる動作の前にクラス初期化検査を挿入することだが，この方法ではクラス

```
1:  i = C.field;
```

(a) Java ソースコード

```
1:  if (!IsInitialized(C)){
2:    Initialize(C);
3:  }
4:  i = D_C.static_fields[FIELD_OFFSET].value;
```

(b) 基本的な実現

```
1:  i = D_C.static_fields[FIELD_OFFSET].value;
```

(c) 最適化した実現

```
 1:  REFERENCE:
 2:  goto INIT
 3:      ⋮
 4:  INIT:
 5:  if (!IsInitialized(C)){
 6:    Initialize(C);
 7:  }
 8:  // overwrite "goto INIT" in line 11
 9:  *REFERENCE = "i = D_C.static_fields[FIELD_OFFSET].value";
10:  goto REFERENCE;
```

(d) コードの上書きによる最適化

図 2.1: クラス初期化検査の除去

初期化検査から大きなオーバヘッドが生じる．たとえば，クラスを初めて参照しうる動作の 1 つであるクラスフィールド参照について考える．図 2.1(a) の Java で記述したクラスフィールド参照を実施するソースコードは，単純にはクラス初期化検査を使った図 2.1(b) の C ソースコードに相当する機械コードとして実現できる．図 2.1(b) の 1 〜 3 行目がクラス初期化検査のコードである．図 2.1(b) の 4 行目にある変数 D_C はクラス C を表すデータ構造とする．また，D_C のメンバ `static_fields` はクラス C が定義するクラス変数群を格納する配列とし，その `FILED_OFFSET` 番目の要素が，クラスフィールド `field` の値 `value` を格納するデータ構造であるとする．

図 2.1(b) の実現では，クラスフィールド参照を実行するたびにクラス初期化検査を実行し，大きなオーバヘッドが生じる．図 2.1(b) の 4 行目にあるように，クラスフィールド参照自体は単なる大域変数参照なので，クラスフィールド参照より大きなオーバヘッドがクラス初期化検査から発生することが判る．

2.3.2 クラス初期化検査のオーバヘッド削減

クラス初期化検査は検査対象のクラスの初期化が済めば冗長になる．そこで，実行の高速化を目的として，冗長になったクラス初期化検査を除去する最適化が提案されている．ここでは，インタプリタおよび動的コンパイラにおいて冗長になったクラス初期化検査を除去する方法を示す.

インタプリタにおける高速化

インタプリタでは実行時にバイトコードを書き換えることでクラス初期化検査を実行時に除去する [Lindholm96]．たとえば，クラス初期化検査を必要とするクラスフィールド参照のバイトコードを，1 回実行してクラス初期化検査を実施したら，バイトコードを書き換え，次からクラス初期化検査を省略してクラスフィールド参照をおこなうバイトコードを実行する．

動的コンパイラにおける高速化

動的コンパイラでは，クラス初期化検査から生じるオーバヘッドを軽減するために，クラス初期化検査における検査対象のクラスが初期化済みかコンパイル時に調べ，その結果に応じて次のコードを出力する [Suganuma00].

初期化済みの場合: クラス初期化検査を省略したコード（図 2.1(c)）

初期化済みではない場合: 1 回目の実行において，コードを上書きし，2 回目からはクラス初期化検査を実行しなくなるコード（図 2.1(d)）

図 2.1(d) のコードでは，初めて実行するときに 2 行目から 5 行目のクラス初期化検査にジャンプして，必要であればクラスの初期化を実施し，9 行目でジャンプ元の 2 行目のコードをクラス初期化検査なしのクラスフィールド参照のコードに書き変えた上で，2 行目にジャンプする．こうすると， 2 回目以降の実行ではクラス初期化検査なしでクラスフィールドを参照可能になる．

過去の Java2C トランスレータにおける高速化

Java2C トランスレータでは，図 2.1(d) に示すようなコードの上書きをおこなうコードを出力することはできない．なぜならコードを上書きするには，上書きするコードのアドレスを知る必要があるが，C 言語ではコンパイル済みコードがどのアドレスにあるか知ることができないからである．図 2.1(d) のコードは 9 行目でラベル `REFERENCE` のアドレスを参照しているが，ラベルのアドレス参照は gcc など一部の C コンパイラがサポートしているものの，ANSI など一般的な C 言語の規格が定める機能ではないため，現在市場に出

回っている C コンパイラの多くがサポートしているとは言い難い．また，組込み機器において ROM にコードを配置する場合にはコードを上書きすること自体ができない．1.3 節で指摘したように，Java2C トランスレータの利点は可搬性にあるから，可搬性を代償にしてまで，たとえば gcc 固有の機能を使うことは得策でない．

そこで，Java2C トランスレータではコードの上書きとは違う方法でクラス初期化検査のオーバヘッドを軽減する．過去の Java2C トランスレータではクラス初期化検査を除去するために，検査対象のクラスがクラス初期化関数 < `clinit` > を持つか静的コンパイル時に調べ，持たなければ初期化不要とみなしてクラス初期化検査を除去していた [Proebsting97]．この方法には次に示す 2 つの欠点がある

1. < `clinit` > を持つクラスのクラス初期化検査を除去できない．
2. クラスファイルの更新に対応できない．静的コンパイル時に参照したクラスファイル内の < `clinit` > がなくても，静的コンパイル後にクラスファイルを更新した場合，更新後のクラスファイル内には < `clinit` > がある可能性がある．

本論文の 3 章では，これらの欠点をもたない，全てのクラス初期化検査を除去し，かつ動的ロードを実現できるクラス初期化検査の除去方法を提案する．

2.4 仮想メソッド呼出し

本節ではまず，仮想メソッドとは何かを示し，次に過去に提案された仮想メソッド呼出しの高速化方法のうち，Java で利用可能なディスパッチ表法と I-call if 変換を紹介する．最後に，過去に提案された高速化方法と本論文の 5 章で提案する方法の違いを示す．

2.4.1 仮想メソッド

仮想メソッドとは子クラスにおいて再定義可能なメソッドを意味する．C++ では仮想メソッドを仮想関数と呼ぶ．Java のソースコードにおける仮想メソッド呼出しの例を図 2.2(a) に示す．Java では仮想メソッド呼出しを `obj.`m`()` という形式で記述し，これは変数 `obj` が参照するインスタンスに，名前が m のメソッドを実行するよう指示することを意味する．ここで，メソッドを実行するよう指示を受けるインスタンスをレシーバと呼ぶ．

仮想メソッド呼出しが手続き型言語における手続き呼出しと異なる点は，呼び出すメソッドがレシーバのクラスに対応して変化することである．たとえば図 2.4 左のソースコードではクラス `C` において，クラス `B` が定義するメソッド m`()` を再定義しているが，このとき仮想メソッド呼出し `obj.`m`()` によって呼び出されるメソッドは，レシーバ `obj` のクラスが `B` ならば `B` が定義する m`()` になり，`C` ならば `C` が定義する m`()` になる．

仮想メソッド呼出しの基本的な実現を，図 2.2(b) の C ソースコードに示す．図 2.2(b) のコードは次の手順で仮想メソッド呼出しをおこなう．

```
1:   obj.m();
```
(a) Java ソースコード

```
1:   code = LookUpMethod(obj->class, "m");
2:   code(obj);
```
(b) 基本的な実現

```
1:   code = obj->class.dispatch_table[O_m];
2:   code(obj);
```
(c) ディスパッチ表法

```
 1:   class = obj->class;
 2:   if (class == C_1){
 3:     // クラスが C_1 なら C_1 の m() を直接呼出し
 4:     C_1 :: m(obj);
 5:   } else if (class == C_2){
 6:     // クラスが C_2 なら C_2 の m() を直接呼出し
 7:     C_2 :: m(obj);
 8:   }else{
 9:     // どちらでもなければ間接呼出し
10:     code = obj->class.dispatch_table[O_m];
11:     code(obj);
12:   }
```
(d) I-call if 変換

図 2.2: 仮想メソッド呼出しとその実現

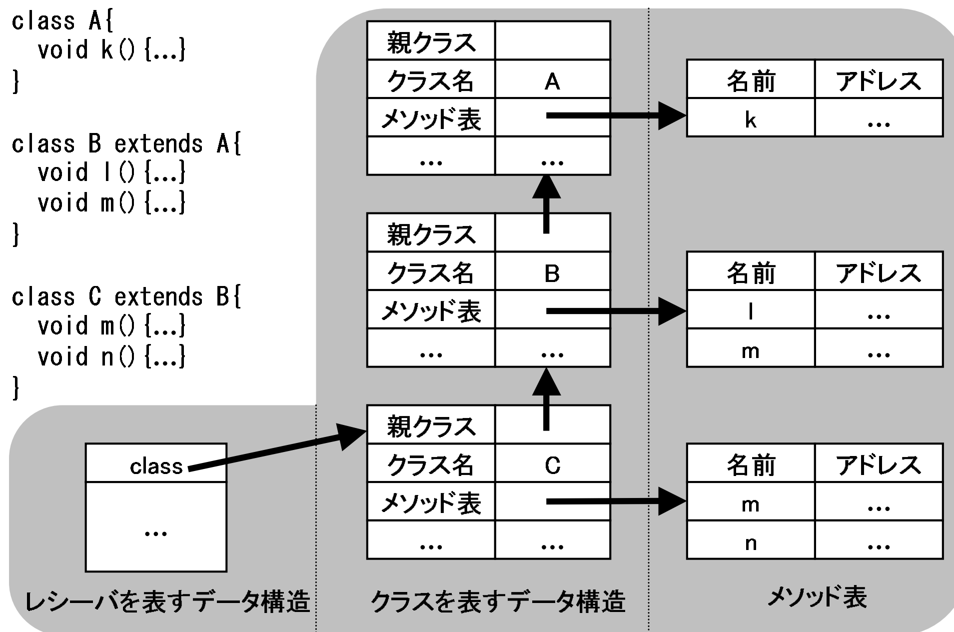

図 2.3: クラスとメソッド表

1. レシーバのクラスと，呼出し対象のメソッド名を鍵として，呼出し対象のメソッドのアドレスを検索する．
   - 図 2.3 に示すように，レシーバを表すデータ構造には，レシーバのクラスを参照するメンバ `class` があるものとする．また，クラスを表すデータ構造には，親クラスへのリンクと，メソッド表へのリンクがあるものとする．ここでメソッド表とは，クラスが定義するメソッドの名前とアドレスを登録する表であるとする．図 2.3 に示した，クラスを表すデータ構造とメソッド表は，図 2.3 左上のソースコードから生成したものである．
   - 図 2.2(b) の 1 行目で呼び出す関数 `LookUpMethod()` はレシーバのクラス $class$ と，呼び出し対象のメソッド名 $name$ を引数として受け取り，呼出し対象のメソッドのアドレスを求めて返戻する．具体的には，クラス $class$ から，その親クラスに向ってクラスを順次たどり，クラスのメソッド表に名前が $name$ のメソッドが登録してあるか調べ，あったら，そのアドレスを返戻する．
2. 求めたアドレスに間接ジャンプする．

図 2.2(b) のコードを仮想メソッド呼出しの実現として使うことは，可能ではあるが適切ではない．なぜなら，関数 `LookUpMethod()` の実行手順から想像できるように，図 2.2(b) の実現はオーバヘッドが大きいからである．オブジェクト指向プログラミングでは，仮想メソッド呼出しを頻繁に実行するため，仮想メソッド呼出しを高速に実現する必要がある．

仮想メソッド呼出しの高速化は，オブジェクト指向プログラミング言語の実現の分野で古くから研究が進んでおり，これまでに数多くの高速化技法が提案されている [小野寺 97]．Java に適用できる高速化技法には，ディスパッチ表法と I-call if 変換がある．

2.4.2 ディスパッチ表法

ディスパッチ表法 [小野寺 97] は，仮想メソッドの検索を高速化する技法である．ディスパッチ表は，クラスごとに作成するデータ構造であり，対応するクラスのインスタンスが実行しうる全仮想メソッドのアドレスを格納する．ディスパッチ表を作成する手順を次に示す．

1. クラス C のディスパッチ表 T_C を作成するにあたり，C が親クラス C_S をもつなら，C_S のディスパッチ表 T_{C_S} を先に作成する．
2. T_C を格納する空のメモリ領域を確保する．
3. C が親クラス C_S を持つならば，T_{C_S} の中身を T_C にコピーする．
4. C が定義する個々の仮想メソッド n() のアドレスを順次 T_C に登録する．登録の手順を次に示す．

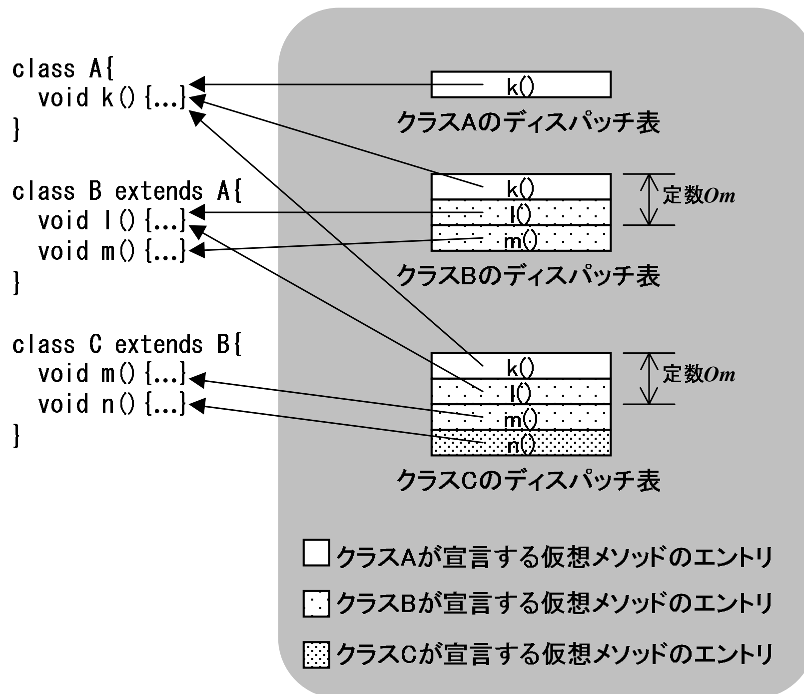

図 2.4: ディスパッチ表

- $n()$ が親クラスにおける仮想メソッドの定義を再定義するものか調べる．
  - 再定義するならば，ディスパッチ表に既に存在する $n()$ のエントリを求める．求めたエントリには親クラスが定義する $n()$ のアドレスが登録してあるが，これを C が再定義する $n()$ のアドレスに書き換える．
  - 再定義しないならば，つまり C が仮想メソッド $n()$ を宣言するならば，ディスパッチ表 T_C の末尾に仮想メソッド $n()$ のアドレスを格納するエントリを追加し，そこに C が定義する $n()$ のアドレスを登録する．

図 2.4 左に示したソースコードにおけるクラス A, B, C に対応するディスパッチ表を構築した結果を図 2.4 右に示す．図 2.4 左のソースコードでは，クラス C がクラス B を継承し，クラス B がクラス A を継承している．クラス C のディスパッチ表をみると，親クラス B のディスパッチ表にクラス C が宣言する仮想メソッド $n()$ を追加し，クラス C が再定義するメソッド $m()$ のアドレスを書き換えた形になっていることが判る．

さて，図 2.2(a) の仮想メソッド呼出し `obj`.$m()$ をディスパッチ表法で実現すると，図 2.2(c) のコードになる．図 2.2(c) のコードにおける定数 O_m は，ディスパッチ表法における，仮想メソッド $m()$ のアドレスが登録してあるエントリの位置を表す．図 2.4 にあるようにクラス B のディスパッチ表では O_m 番目のエントリにクラス B が定義する仮想メソッド $m()$ のアドレスが登録してあり，クラス C のディスパッチ表ではクラス C が定義する仮想メソッド $m()$ のアドレスが登録してある．したがって図 2.2(c) のコードは，`obj` が参照するインスタンスのクラスが B であるときはクラス B が定義する仮想メソッド $m()$

を，C であるときはクラス C が定義する仮想メソッド m() を呼び出すことができる．

図 2.2(c) と図 2.2(b) のコードを比較すると，ディスパッチ表を使う図 2.2(c) のコードの方が呼出し先のメソッドを高速に検索できることが判る．ディスパッチ表法においてメソッドの検索にかかるコストは，レシーバ `obj` からディスパッチ表を取得するためのメモリ参照と，ディスパッチ表からメソッドのアドレスを取得するためのメモリ参照のみであり，クラスを再帰的にたどってメソッドを検索する関数 **`LookUpMethod()`** が必要とするコストより小さい．

2.4.3 I-call if 変換

I-call if 変換 [Calder94] は，仮想メソッド呼出しを，呼出し対象のメソッドを検索するための if 文と，検索したメソッドを直接呼び出す文に変換する最適化である．仮想メソッド呼出し `obj`.m() を I-call if 変換を使って実現した例を図 2.2(d) に示す．図 2.2(d) のコード中にあるクラス C_1，C_2 はレシーバ `obj` がとりうるクラスであり，コンパイル時にクラス階層解析 [Dean95] やクラスフロー解析 [Pande94] によって求める．

I-call if 変換の利点は仮想メソッド呼出しにインライン展開を適用して高速化できることにある．I-call if 変換を適用すると，仮想メソッド呼出しが直接呼出しになり，インライン展開を適用可能になる．間接呼出しを使うディスパッチ表法にはインライン展開を適用できない．

I-call if 変換には幾つかのバリエーションがある．図 2.2(d) のコードでは，レシーバ `obj` のクラスを鍵として呼出し対象のメソッドを検索しているが，クラスの代わりにメソッドを検索鍵とすることもできる [Detlefs99]．本論文では検索鍵にクラスを使う I-call if 変換を**クラスチェック変換**と呼び，メソッドを使うものを**メソッドチェック変換**と呼ぶことにする．さらに，I-call if 変換という用語はクラスチェック変換とメソッドチェック変換を総称するものとする．また，図 2.2 のコードではレシーバ `obj` のクラスが C_1 でも C_2 でもないとき間接呼出しで仮想メソッド呼出しを実行するが，静的解析によって `obj` がとりうるクラスが必ず C_1 か C_2 のいずれかであると特定できる場合には間接呼出しを省略し，if 文を 1 段節約できる．間接呼出しを省略した I-call if 変換を**限定的**であるということにする．

I-call if 変換を応用した研究としては，I-call if 変換を適用した後のコードにおいて，レシーバの候補クラスが明確になることを利用する最適化がある [Hölzle94]．たとえば I-call if 変換を適用して生成した図 2.2(d) のコードでは，その 4 行目において，レシーバ `obj` のクラスを C_1 に限定できる．4 行目のコードは直接呼出しで，このままでは余り最適化できない．しかし，4 行目をインライン展開すれば最適化の余地が生まれ，その際に `obj` のクラスを C_1 に限定できることを使って最適化を実施できる場合がある．この研究のクラスの動的ロードが存在する状況下における応用については Sreedhar らの研究がある [Sreedhar00]．

なお，Java 向け動的コンパイラの中には，動的コンパイル後に，呼出し対象の仮想メ

ソッドを再定義するクラスを読み込まないと仮定してI-call if変換を適用することで，I-call if変換の分岐段数を削減し，実行の高速化を図るものがある．この高速化を動的I-call if変換と呼ぶことにする．動的I-call if 変換を適用した結果，SPECjvm98 の 1 項目である_227_mtrt の実行速度が 2.8 倍になったという報告がある [Ishizaki00, 石崎 02]．動的I-call if変換について図2.2(d)のコードを使って詳述する．

図2.2 (d) のコードには，10 〜 11 行目にディスパッチ表経由の間接呼出しがある．間接呼出しがある理由は，動的コンパイル後に $m()$ を再定義するクラスを動的ロードする可能性があるからである．動的コンパイル時点でメソッド $m()$ を定義するクラスが C_1 と C_2 の 2 つだとすれば，$m()$ を再定義するクラスを新たに動的ロードしない限り，10 〜 11 行目の間接呼出しを実行することはありえない．動的I-call if変換では，図2.2(d) の 10 〜 11 行目にある間接呼出しを除去して分岐段数を 1 段削減する．動的I-call if変換を適用して生成したコードは，動的コンパイル後に呼出し対象の仮想メソッドを再定義するクラスを動的ロードし，メソッドの候補数が増えると無効になる．無効になったコードを実行し続けると，誤ったメソッドを呼び出すなど問題がおきる．そこで，動的I-call if変換を適用するJava VMは，高速化の適用と同時に，コードが無効になった場合に対処するための対策も用意する．対策の実現方法は 2 種類あり，1 つは無効になったコードを上書きして有効な状態に戻す方法であり [Cierniak00, Ishizaki00, 石崎 02]，もう 1 つは脱最適化 [Hölzle92] によって無効になったコンパイル済みコードによるメソッドの実行を，途中からインタプリタなど有効な実行手段に引き継ぐ方法である．

2.4.4 提案手法

本論文の 5 章では，I-call if変換のバリエーションのうち，メソッドチェック変換に動的ロードのオーバヘッドを軽減する効果があることを指摘する．SPECjvm98を使って評価した結果，メソッドチェック変換を使うと，クラスチェック変換を使うより最大で 9.8%実行を高速化できることが判った．I-call if変換自身は過去に提案された技法だが，I-call if変換と動的ロードの関連性を指摘した研究は過去に存在しない．

2.5 例外処理

例外処理とは，プログラム実行中に例外的事象が発生した箇所から，ハンドラと呼ぶ，発生した例外に対処するルーチンまで大域的にジャンプする機能である．大域的ジャンプは，その準備や実行に大きなコストが必要になりうるので，実現方法を慎重に検討する必要がある．本節ではまず，Javaのソースコード上における例外処理の記述方法について述べ，次に既存の例外処理の実現方法を示す．最後に，本論文の 6 章における提案事項と過去の例外処理に関する研究を比較する．

```
 1:  class Example{
 2:    void boo(){
 3:      try{
 4:        this.foo();
 5:      }
 6:      catch(Exception e){
 7:        // ハンドラ
 8:      }
 9:    }
10:
11:    private synchronized void foo()
12:      throws Exception{
13:      while(true){
14:        this.woo();
15:      }
16:    }
17:
18:    private void woo() throws Exception{
19:      throw (new Exception());
20:    }
21:  }
```

図 2.5: 例外処理の例

2.5.1 Java のソースコード上における例外処理の記述

Java における例外処理の利用方法について，図 2.5 に示す Java のソースコードを例として具体的に述べる．例外処理を利用するには，まず，キーワード **try** を使って，ハンドラが管轄するプログラムの領域を定める．この領域を **try** ブロックと呼ぶ．図 2.5 のプログラムでは，メソッド **boo()** の中に **try** の記述があるが（3 行目），**try** に続く中括弧で囲んだ範囲（3 〜 5 行目）が **try** ブロックである．**try** ブロック内部を実行する過程で例外が発生した場合，**try** ブロックに続く **catch** 節に制御が移る．**catch** 節の冒頭では，**catch** 節内部のハンドラで処理する例外のクラスを指定できる．たとえば **boo()** 内の **catch** 節（6 〜 8 行目）では，冒頭の 6 行目で **catch(Exception e)** と記述しているため，発生した例外が **Exception** クラスあるいはそのサブクラスのインスタンスである場合に限って例外を捕捉し，**catch** 節内部（「ハンドラ」とコメントを書いてある箇所）を実行する．そうではない場合には，**boo()** 内の **catch** 節では例外を捕捉せず，対応する **catch** 節が現れるまで **boo()** の呼出し元を再帰的にたどる．

try ブロックを準備したら，次に，キーワード **throw** を使って，例外を投げ，対応するハンドラまでジャンプするコードを書く．図 2.5 のプログラムでは，メソッド **boo()** が **foo()** を呼び，さらに **foo()** が **woo()** を呼ぶ．そして，**woo()** 内部の 19 行目で **Exception** クラスの例外を生成して **throw** で投げる．この結果，投げた例外を捕捉するハンドラ（メソッド **boo()** 内の **catch** 節）にジャンプすることになる．

2.5.2 既存の例外処理の実現方法

例外処理の機能は Java の他に C++なども提供する．既存の Java や C++の処理系における例外処理の実現は，次に示す 3 種類に分類できる．

表引き法 例外が発生した点でのプログラムカウンタの値から，対応するハンドラを表引きで探してジャンプする方法．表引き法で例外処理を実現する手順を次に示す．

1. コンパイル時に，個々のメソッドについて，`try` ブロックの有効範囲（プログラムカウンタの上限と下限）と，それに対応するハンドラのアドレスを収めた表を作成する．本論文ではこの表を**例外分岐表**と呼ぶ．
2. 実行時に例外が発生したら，次の手順でハンドラまでジャンプする．
   (a) 例外発生時点で実行していたコードのアドレス（プログラムカウンタの値）を求める．
   (b) 求めたアドレスを検索鍵として例外分岐表を引き，例外発生時に実行すべきハンドラのアドレスを求める．
      - ハンドラのアドレスが求められたら，該アドレスにジャンプする．
      - ハンドラのアドレスが求められなかった場合，まず，実行時スタックの一番上にあるスタックフレームから返戻先のアドレスを取得し，スタックフレームを破棄する．次に，返戻先のアドレスに (b) の操作を適用する．

表引き法で例外処理を実現している言語処理系の例としては，C++コンパイラのうち，C++のソースコードから直接機械コードを生成するものや [Schilling98, Dinechin00]，Java のインタプリタ [Lindholm96]，動的コンパイラがある [Krall98, Suganuma00, Paleczny99].

setjmp 法 `try` ブロックの入り口で `setjmp()` し，例外が発生したら `longjmp()` でハンドラにジャンプする方法．setjmp 法で例外処理を実現する言語処理系の例としては，C++コンパイラのうち，C++のソースコードから C ソースコードを経由して機械コードを生成するものがある [Cameron92].

2 返戻値法 機械コード中にハンドラへの分岐命令を明示的に挿入し，分岐命令を使ってハンドラまでジャンプする方法．メソッド内にハンドラがない場合にはハンドラに分岐できないが，その場合にはメソッドから返戻し，返戻先に挿入した分岐命令を使ってハンドラまでジャンプする．返戻先で例外が発生したか否か判定するために，メソッドの返戻値に例外を追加する．この結果，返戻値が通常の返戻値と例外の 2 つになることから，2 返戻値法と呼ばれる．2 返戻値法で例外処理を実現する言語処理系の例としては，CACAO という Java 向け動的コンパイラなどがある [Krall97, 八杉 01].

これらの実現方法のうち，最も実行速度に与えるオーバヘッドが小さいのは表引き法である．なぜなら，表引き法では他の実現方法と異なり，例外処理の実現にコードの挿入を必

要としないからである．たとえば setjmp 法で例外処理を実現するには try ブロックの入口に `setjmp()` を挿入する必要があり，また 2 返戻値法ではメソッド呼出しからの返戻先に分岐命令を挿入する必要がある．これら例外処理を実現するために挿入したコードは実行オーバヘッドの発生源となるが，表引き法ではコードを挿入しないで例外処理を実現できるので，他の方法よりオーバヘッドを小さくできる．

しかし，Java2C トランスレータによる例外処理の実現では，表引き法を利用することはできない．なぜなら，表引き法を利用するには `try` ブロックの上限や下限といったプログラム中のアドレスを取得する必要があるが，C 言語ではプログラム中のアドレスを取得することができないからである．C コンパイラの中には製品独自の機能拡張としてプログラム中のアドレスを取得する機能を提供するものもあるが，特定の C コンパイラに依存すると可搬性に問題が出る．

C++を中間語として使うトランスレータでは表引き法を使って例外処理を実現できる場合もある．C++は例外処理の機能を提供する．したがって，もし C++の例外処理を使って Java の例外処理を実現できるならば，Java2C トランスレータではなく Java2C++ トランスレータを作成すれば Java の例外処理を簡単に実現できる．さらに，C++ コンパイラが例外処理を表引き法で実現しているならば，Java2C++トランスレータにおける例外処理の実現も表引き法になる．しかし，C++の例外処理で Java の例外処理を実現する方法には，次に示す 2 つの理由から，少なくとも可搬性の点に問題がある．

スレッドのサポート C++コンパイラの中にはスレッドをサポートしないものがある．そのような C++コンパイラにおける例外処理の実現では，マルチスレッドの Java アプリケーションで正しく例外を処理できない．これは，例外処理の実現が，実行時にスレッドの資源であるスタックを再帰的にたどることから判るように，スレッドの実現に深くかかわっているためである．

既存の Java VM との接続 C++コンパイラにはスレッドをサポートするものもあるが，その例外処理の実現はサポートするスレッドに依存する．このことは，C++コンパイラが生成する機械コードと，既存の Java VM をリンクする場合に障害となりうる．

Java2C トランスレータが生成したコードは実行時に Java VM とリンクして初めて動作する．つまり，Java2C トランスレータの開発にあたっては，Java2C トランスレータだけでなく，生成したコードを接続する Java VM もあわせて開発する必要があるが，Java VM はごみ集めやスレッドなど数々の機能を提供するため，開発に膨大なコストがかかる．

Java2C トランスレータによる高速化機能をもつ Java VM を低コストに開発する手段として，既存の Java VM を使う方法がある．この方法では，新規開発対象を Java2C トランスレータ本体に限定し，開発に要するコストを抑制できる．

ただし，既存の Java VM を利用する開発方法では，スレッドの実現が既存の Java VM に依存するが，それが C++コンパイラがサポートするスレッドと一致するとは限らない．一致しない場合，C++の例外処理は正常に動作しない．この問題の回避策として，既存の Java 仮想機械のスレッドに関連する部分を改変し，スレッドの実現を C++コンパイラがサポートするものに合わせる方法もある．しかし，スレッドに関連する部分はごみ集めなど多岐に渡り，書換えにかかる手間は小さくない．

これらの理由が原因か否かは定かではないが，既存の Java 向け静的コンパイラで，C++を中間語として利用するものは存在しない．

2.5.3 Java2C トンランスレータで利用可能な例外処理の実現方法

C++の例外処理を利用する方法に問題がある以上，可搬性の高い Java の例外処理の実現方法は，setjmp 法か 2 返戻値法のいずれかになり，Java2C トランスレータの設計にあたっては，どちらを採用するか選択する必要がある．選択にあたっては，どちらが優れているかを比較したデータがあると有益だが，過去の研究に setjmp 法と 2 返戻値法を比較したものは存在しない．そこで本論文の 6 章ではこの比較を実施し，また setjmp 法や 2 返戻値法向けの最適化技法を提案，評価する．なお，setjmp 法と表引き法の比較については Krall らの研究がある [Krall98].

2.6 null 検査

null 検査とは，参照型の変数が指示するインスタンスを参照する前に，変数の内容が null か調べ，null ならば実行時例外 `NullPointerException` を投げる動作のことである．Java ではインスタンスフィールド参照およびインスタンスメソッド呼出しの前に null 検査を実施する．null 検査は 26%のバイトコードが実施するほど実行頻度が高い動作なので [千葉 02]，効率的に実現する必要がある．本節ではまず，null 検査の実現のうち，Java2C トランスレータで利用できる明示的な null 検査を紹介し，次に現在市場に出回っている Java 向け動的コンパイラが利用している暗黙の null 検査を紹介する．最後に，null 検査に関連した Java2C トランスレータの問題点を指摘する．

2.6.1 明示的な null 検査

null 検査は単純には分岐命令を使って実現できる．分岐命令による実現を明示的な null 検査と呼ぶ．明示的な null 検査が含む分岐命令は，昨今のプロセッサにとって実行負荷が小さくない [Hennesy90]．そこで null 検査向けの最適化技術として，フロー解析を使って冗長な null 検査を除去する技法が提案されている [Ishizaki99].

2.6.2　暗黙の null 検査

フロー解析を使った最適化では全ての null 検査を除去することはできない．そこで除去できずに残った null 検査を分岐命令なしで効率的に実現する方法として暗黙の null 検査という技法が提案されている [Alpern00, Suganuma00, Kawahito00, 川人 01]．暗黙の null 検査では，たとえばインスタンスフィールド参照を，単にメモリ参照をおこなう機械コードで実現する．こうして出力した機械コードは実行時に参照が null であるとき，null の値が 0 であるならば，0 番地を含むページを参照する．0 番地を含むページにはデバッグ支援などを目的として OS が読み書き禁止の保護属性を与えていることがあり [Richter96]，このとき 0 番地を含むページを参照するとシグナルが発生する．暗黙の null 検査ではこのシグナルを捕捉して例外の発生を検知し，例外 `NullPointerException` を生成する．そして，シグナルを発生した機械コードのアドレスから例外を発生したソースコード上の点を求め，表引き法で対応するハンドラを求めて生成した例外を引き渡す．

現在市場に出回っている Java 向け動的コンパイラでは暗黙の null 検査を使って null 検査を実現することが多いが，Java2C トランスレータでは暗黙の null 検査を利用できない．これは C 言語においてプログラム中のアドレスを取得することができないので，シグナルを発生した機械コードのアドレスから，ソースコード上の例外発生地点を算出できないためである．

2.6.3　null 検査の実現に関連した Java2C トランスレータの問題

Java2C トランスレータには，暗黙の null 検査を利用できないことの他にも，null 検査に関連した問題点が 1 つある．それは，分岐確率に関する情報を活用できないことである．

null 検査など，例外が発生したか検査する分岐は，実行時に高確率で例外が発生していない方に分岐する．この分岐確率の偏りを利用する最適化 [古関 01] の中に，機械コード生成時に実施すべき次の技法があるが，Java2C トランスレータではこれらの最適化を実施できない．また，C 言語では分岐確率を表現できないので，Java2C トランスレータから C コンパイラに分岐確率に関する情報を伝達することもできない．

1. 分岐の機械コード中にある分岐方向の予測値を格納するビット [IBM94, Triebel00] を適切に設定する最適化
2. 滅多に実行しないコード（たとえば null 検査の例外を生成して投げる部分のコード）と，それ以外のコードを分離して，別々に配置することで，雑然と配置する場合より命令キャッシュのヒット率を改善する最適化

2.7 スタックトレース

Java の標準ライブラリクラス `java.lang.SecurityManager` は実行コンテキスト，すなわち実行時スタックにどんなメソッドのスタックフレームが積んであるかという情報を配列に収めて返戻するメソッド `getClassContext()` を提供する．本論文ではこのメソッドが返戻する情報をスタックトレースと呼ぶことにする．本節ではまず，バイトコードを直接機械コードに変換するコンパイラが一般的に利用しているリターンアドレスを使ったスタックトレースの算出方法を示し，次に過去の Java2C トランスレータにおけるスタックトレースの算出方法について述べる．

2.7.1 リターンアドレスを使ったスタックトレースの算出方法

リターンアドレスからスタックトレースを算出する方法では，実行時スタックに積んであるスタックフレームを走査して，スタックフレーム中に保存してあるリターンアドレスに対応するメソッドを順次書き出すことでスタックトレースを算出する [Cierniak00, Alpern00]．この算出方法は，スタックトレースを算出可能にするために準備を必要としない点で効率的である．なぜなら，この算出方法で利用する唯一の情報であるリターンアドレスは，スタックトレースを求めるか否かによらずスタックフレーム上に存在する情報だからである．

2.7.2 Java2C トランスレータにおけるスタックトレースの算出方法

Java2C トランスレータでは，リターンアドレスを使ったスタックトレースの算出方法を利用できない．なぜなら，C 言語ではプログラム中のアドレスを取得できないので，リターンアドレスが C ソースコード上のどこを指示しているか算出できないからである．過去の Java2C トランスレータは C 言語で表現するメソッドの出入口に明示的にどのメソッドを実行中か記録するコードを挿入することでスタックトレースを計算可能にしていたが [Proebsting97]，コードを挿入すると実行速度にもコードサイズにも悪影響がでる．

我々が実現した Java2C トランスレータでも過去の実現と同様にスタックトレースを記録するコードを挿入することでスタックトレースを計算可能にしたが，同時にこのコードが冗長か判定して冗長ならば除去する最適化を実現してオーバヘッドの軽減を図った．この最適化については 3 章で述べる．

2.8 スタック溢れ検査

Java では実行時スタックが溢れたら実行時エラー `StackOverflowError` を投げる必要がある．実行時スタックが溢れたか検査する方法には，明示的な機械コードによって上限を超えたか比較する方法と，シグナルを利用する方法がある [Suganuma00, Alpern00]．ただし Java2C トランスレータではこれらの方法を直接的には利用できない．なぜなら，こ

れらの方法を利用するにはスタックポインタを参照する必要があるが，C 言語にはスタックポインタを参照する手段がないからである．そこで Java2C トランスレータではアセンブラで実現した検査を実行する関数を呼ぶことでスタック溢れ検査を実現するが，関数呼出しにはオーバヘッドがかかる．

2.9 モニタ

モニタはスレッド間の排他制御を実現する機能である．ソースコード上でモニタが保護すると宣言した資源を参照するスレッドはまず排他制御権を確保する必要がある．Java アプリケーションは排他制御権の確保，解放を頻繁に実施するので，確保や解放から生じるオーバヘッドを軽減する対策が必要になる．過去の研究が提案した対策は大まかに次の 2 つに分類できる．

1. Escape 解析を利用して排他制御権の確保，解放が冗長か判定し，冗長であれば省略する最適化 [Park90, Park91, Park92, Blanchet98, Choi99, Blanchet99, Whaley99]
2. 排他制御権を確保，解放する動作を高速化する技法 [Bacon98, Agesen99, Onodera99, Gagnon01, Dice01]

2 の研究の結果，排他制御権の確保，解放を数命令で実現可能になった．しかし，Java2C トランスレータではそれほど効率良く排他制御権の確保を実行できない．なぜなら 2 の研究が示した排他制御権を確保する命令列の中には atomic 演算が入っているが，C 言語では atomic 演算をおこなう機械コードを出力するよう記述できないからである．そこで Java2C トランスレータでは排他制御権を確保するコードを，アセンブラで記述した排他制御関数を呼ぶコードとして実現するが，これでは関数呼出しの分オーバヘッドが増える．バイトコードを直接機械コードに変換する Java 向けコンパイラでは，排他制御の命令列をインライン展開して関数呼出しのオーバヘッドを軽減するのが一般的である．

河内谷らは atomic 演算を使わないモニタの実現を提案しているが [Kawachiya02]，河内谷らの実現では排他制御の確保を特定の命令列で実現する必要がある．Java2C トランスレータでは特定の命令列を出力することは困難であり，したがって Java2C トランスレータでは河内谷らの実現も関数呼出しなしでは利用できない．

2.10 ごみ集め

本節ではまず，ごみ集めとは何かを明らかにし，次に Java2C トランスレータでごみ集めを実現する上での問題点を示し，その解決策となる保守的ごみ集めを紹介する．

2.10.1 ごみ集めとは

ごみ集め [Jones96] は Java が提供する機能の 1 つで，メモリ管理をおこなう．具体的には，ヒープ領域中にある，もう使わなくなったデータ構造を次の手順で検出し，検出したデータ構造が占有している領域を解放して空領域として再利用可能にする．

1. 実行時スタックや大域変数，レジスタに入っているポインタからデータ構造を再帰的にたどって到達可能なデータ構造の集合 M を求める．
2. ヒープ中に存在するデータ構造のうち，集合 M に属さないものを，もう使わないとみなす．

2.10.2 Java2C トランスレータにおけるごみ集めの実現

Java2C トランスレータでごみ集めを実現する上での問題点は，ポインタの所在に関する情報を生成できないことである．ごみ集めの実施にあたっては，まず最初に実行時スタックやレジスタの中にあるポインタを走査する．この走査をおこなうには実行時スタックのどこにポインタがあり，どのレジスタにポインタが入っているという情報があると便利である．この情報の生成は，コンパイラの仕事である．なぜなら，どのポインタをどのレジスタに収めるか決定するのはコンパイラだからである．

Java2C トランスレータはコンパイラではないのでポインタの所在に関する情報を生成できず，機械コードの生成を担当する C コンパイラも，C 言語がごみ集めの機能を提供しないことから，ポインタの所在に関する情報を生成しない．したがって Java2C トランスレータを利用する Java VM では，ポインタの所在に関する情報が欠落していても動作するごみ集めを実現する必要がある．

ポインタの所在に関する情報がない場合向けのごみ集めとして，保守的ごみ集めという技法が提案されている [Bohem88, 小野寺 94]．保守的ごみ集めでは実行時スタックなどに格納してある値を一通り走査して，その値がポインタである可能性があったらポインタとみなす．こうすると実際にはポインタでない値までポインタと見誤り，その値が指示するデータ構造を利用中と誤認する可能性があるが，過去の研究から誤認による影響は小さいことが判っている．我々が開発した Java2C トランスレータは，保守的ごみ集めによってごみ集めを実現する．

2.11 浮動小数点演算の精度保証

Java の言語仕様は浮動小数点演算の演算精度を規定している．この規定通りに演算を実施するには，Java の言語仕様が定める精度で浮動小数点演算をおこなうライブラリ関数を用意し，浮動小数点演算をおこなうバイトコードを，用意したライブラリ関数を呼び出すコードに変換すればよい．しかし，この実現方法では浮動小数点演算のたびに関数呼出し

をおこなうのでオーバヘッドが大きい．そこで Java の言語仕様による規定と同じ演算精度の浮動小数点演算命令をもつプロセッサ向けコンパイラでは，Java の浮動小数点演算をただの浮動小数点演算命令で実現する．

Java2C トランスレータは機械コード生成をおこなわないが，Java の浮動小数点演算を C 言語におけるただの浮動小数点演算子に変換することで C コンパイラに浮動小数点命令を出力させることができる．ただし PowerPC[IBM94] のように，加算の演算精度も乗算の演算精度も Java の言語仕様通りだが，積和命令 fmadd を使うと精度が狂うといったプロセッサもあり，そういったプロセッサ向けの機械コードを生成する際には C コンパイラにオプションで精度を狂わせる命令を出力しないよう指示する必要がある．

第3章　動的ロードの実現

本章の目的はJava2Cトランスレータにおける動的ロードの実現方法を提案することにある．本章ではまず3.1節で我々が開発したJava2Cトランスレータつき Java VMであるJeanPaulの全体像を示し，次に3.2節でJava2Cトランスレータにおける動的ロードの実現方法を提案する．最後に3.3節でJava2Cトランスレータに実現した最適化の一覧を示し，最適化と動的ロードに相関関係があることを示す.

3.1　JeanPaulの構成

我々が開発したJava2Cトランスレータつき Java VMであるJeanPaulの全体像を図3.1に示す．JeanPaulは1.2節で示した図1.1の Java VMに次の2つの機能を追加したものである.

1. 静的コンパイラ
2. 静的コンパイル済みコードをJava VMにリンクする機能

追加した機能の役割を示すために，JeanPaulでJavaアプリケーションを静的コンパイルし，実行する手順について述べる．この手順の中には，静的コンパイル済みコードを利用するJava VMで動的ロードを実現するために必要な動作を含む．動的ロードを実現するために必要な動作とは，静的コンパイル時に参照したクラスファイルと実行時に動的ロードしたクラスファイルが同一か確認する動作を指す．この動作が必要な理由は2.2節で指摘した.

3.1.1　Javaアプリケーションのコンパイル

JeanPaulの静的コンパイラはJava2Cトランスレータと既存のCコンパイラからなる．静的コンパイラはコンパイル対象のJavaアプリケーションを構成するクラスファイル一式を入力として受け取ってC言語経由で機械コードに変換し，生成した機械コードを共有ライブラリに格納する.

- Java2Cトランスレータは Javaアプリケーションを構成するクラスファイル群を一括してトランスレートする．なぜならJava2Cトランスレータにおいてクラス間に跨がる情報を使う最適化を適用するために，クラスファイル一式が必要だからであ

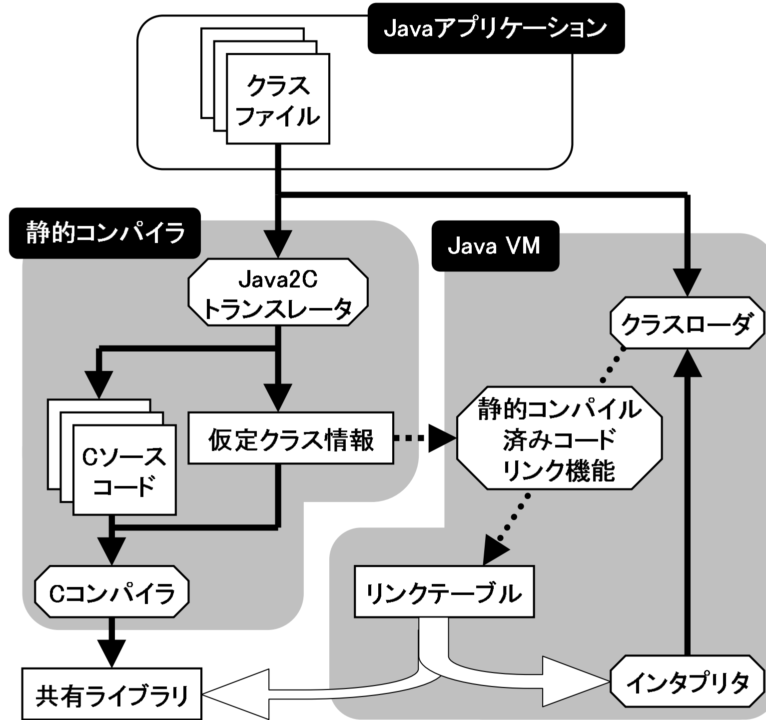

図 3.1: Java2C トランスレータつき Java VM “JeanPaul” の構成

```
struct Class{
  // 固定長要素
  char *class_name;                       // クラス名
  Class *super;                           // 親クラス
                 ⋮
  // 可変長要素
  FunctionAddress dispatch_table[];      // ディスパッチ表
  InstanceField instance_fields[];       // インスタンスフィールド
  StaticField static_fields[];           // クラスフィールド
  InstanceMethod instance_methods[];     // インスタンスメソッド
  StaticMethod static_methods[];         // クラスメソッド
                 ⋮
}
```

図 3.2: クラスを表すデータ構造

る．本論文ではクラス間に跨がる情報を使う最適化を**クラス間最適化**と呼び，単一のクラスのみ参照しておこなう最適化を**クラス内最適化**と呼ぶ．Java2C トランスレータが適用する最適化については 3.3 節で述べる．

- Java2C トランスレータが出力する要素は次の 2 つである．C コンパイラはこれらの要素をコンパイルして共有ライブラリに格納する．
  1. 入力として受け取った個々のクラスファイルに対応する C ソースコード．その構成要素は，クラスが定義する個々のメソッドに対応する関数定義と，クラスを表すデータ構造である．

     ここでクラスを表すデータ構造とは，1.2 節で述べた，クラスローダが動的ロードしたクラスファイルから作成するデータ構造に相当する．クラスを表すデータ構造を図 3.2 に示す．
  2. 動的ロードの実現に必要なデータ構造．本論文ではこのデータ構造を**仮定クラス情報**と呼ぶ．仮定クラス情報は C 言語で記述したデータ構造で，次の 2 つの情報を含む．
     (a) 静的コンパイルした個々のクラスの名前と，クラスファイルの最終更新時刻．
     (b) 個々のメソッドをトランスレートする過程で参照した全クラスを収める集合．

本論文ではメソッドをトランスレートする過程で参照したクラスを**仮定クラス**と呼ぶ．全仮定クラスを収める集合の計算方法については 3.2 節で述べる．仮定クラスはメソッドを定義するクラスだけとは限らない．トランスレート時にクラス間最適化を適用するには複数のクラスを参照する必要がある．

3.1.2 Java アプリケーションの実行

アプリケーションの実行は基本的に 1.2 節で述べた通りの手順で実施する．実行の過程で Java VM 内のクラスローダはオンデマンドにクラスを動的ロードして，初期化する．JeanPaul の JavaVM に固有な動作は，クラスローダがクラスを動的ロードして，初期化する処理に追加した次の 2 つの動作のみである．

1. 動的ロード後に，静的コンパイル時に生成したクラスを表すデータ構造が利用可能か調べ，可能ならば Java VM にリンクして，プログラムの実行に利用する．
2. クラス初期化後に，利用可能になった静的コンパイル済みコードがあるかどうか調べ，あれば Java VM にリンクして，プログラムの実行に利用する．

JeanPaul のクラスローダにおける，クラスの動的ロードおよび初期化処理を図 3.3 に示す．追加した 2 つの動作について順次詳述する．

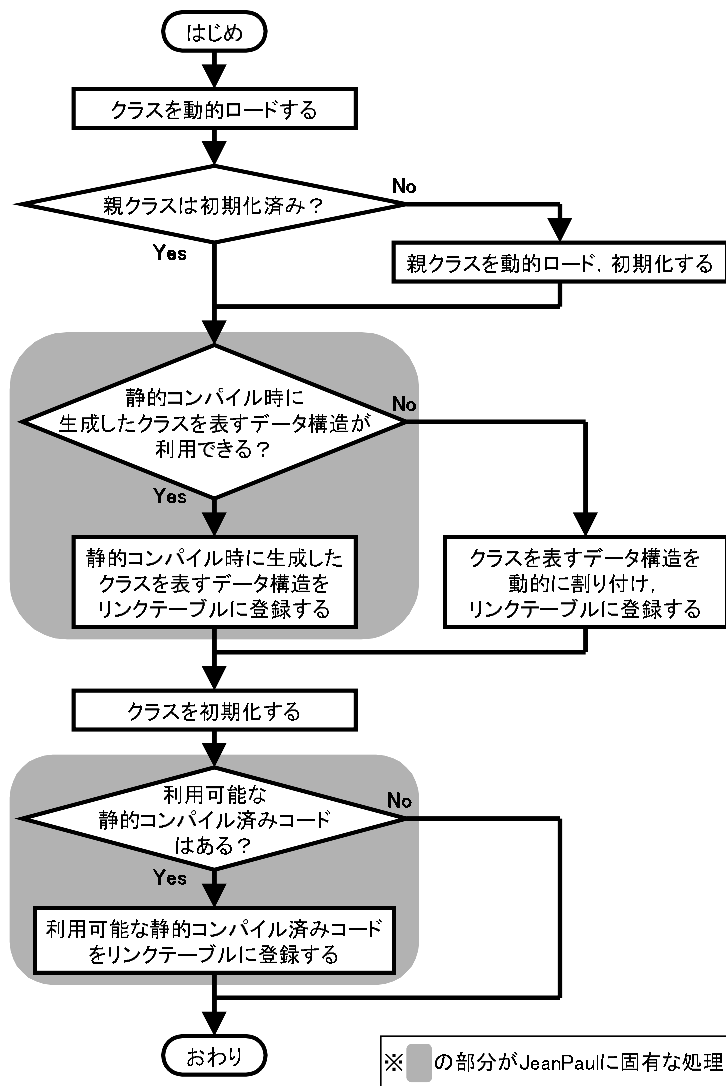

図 3.3: クラスの動的ロードおよび初期化処理

クラスを表すデータ構造のリンク

動的ロード後に，静的コンパイル時に生成したクラスを表すデータ構造を Java VM にリンクする手順を次に示す．

1. クラスローダは，クラスを動的ロードしたあとで共有ライブラリを検索し，動的ロードしたクラスを表すデータ構造が共有ライブラリ内にあるかどうか調べる．
   - クラスを表すデータ構造が共有ライブラリ内にある条件は，クラスが静的コンパイル対象に含まれていたことである．
2. あったならば，そのデータ構造が利用可能か調べる．利用可能になる条件は，データ構造が表現するクラスとその全親クラスについて，静的コンパイル時に参照したクラスファイルと動的ロードしたクラスファイルが一致することである．親クラスも一致する必要があるのは，図 3.2 に示したようにクラスを表すデータ構造の内部にディスパッチ表があり，ディスパッチ表の構造が親クラスに依存するからである．
   クラスファイルが一致するか否かは，クラスファイルの最終更新日時を比較すれば判る．静的コンパイル時に参照したクラスファイルの最終更新日時は仮定クラス情報に記録してある．
3. 利用可能ならば共有ライブラリ内のデータ構造をリンクテーブルに登録し，利用可能でなければ動的ロードしたクラスファイルの内容を参照してクラスを表すデータ構造を作成し，リンクテーブルに登録する．

静的コンパイル済みコードのリンク

クラスの初期化後に，静的コンパイル済みコードを Java VM にリンクする手順を次に示す．

1. クラスローダは初期化が終ったクラスの名前 C と，クラスファイルの最終更新時刻 t を静的コンパイル済みコードリンク機能に通知して，利用可能な静的コンパイル済みコードがあったならばリンクするよう指示する．
2. 静的コンパイル済みコードリンク機能は，受け取った名前 C のクラスが静的コンパイル済みか否か調べる．静的コンパイル済みクラスの名前は仮定クラス情報に記録してある．静的コンパイル済みであれば 3 に進み，そうでなければリンク対象が存在しないのでリンク処理を終了する．
3. 動的ロードしたクラスファイルと，静的コンパイル時に参照したクラスファイルが同一か最終更新時刻を比較して調べる．
   (a) 一致しなかった場合，つまりクラスファイルを静的コンパイル後に更新した場合，なにもせずにリンク処理を終了する．この結果，更新前のクラスファイルから生成

した静的コンパイル済みコードはリンクされず，無効になる．

(b) 一致した場合，クラス C について，クラスファイルの同一性が確認できた結果として利用可能になった静的コンパイル済みコードがあるかどうか調べ，あったら Java VM にリンクする．

静的コンパイル済みコードのリンクは静的コンパイル済みコードのアドレスを図 3.1 のリンクテーブルに登録すること実現できる．リンクテーブル上のメソッドを表すエントリには，初期状態ではインタプリタを呼び出すスタブ関数のアドレスが登録してある．これは静的コンパイル済みコードをリンクするまでの間，プログラムの実行をインタプリタにおこなわせるためである．

リンクはメソッド単位でおこなう．メソッドの静的コンパイル済みコードが実行時に利用可能になる条件は，メソッドの全仮定クラスについて，静的コンパイル時に参照したクラスファイルと動的ロードしたものが同一であるとの確認が済むことである．

個々のメソッドの全仮定クラスを収める集合は仮定クラス情報に記録してある．静的コンパイル済みコードリンク機能は，利用可能な静的コンパイル済みコードがあるかどうか調べるために，仮定クラス情報とクラスファイルの同一性確認が済んだクラスの集合を，照しあわせる．

本論文ではプログラムの実行を開始してから静的コンパイル済みコードのリンクが完了するまでの期間を**リンクの遅延**と表現する．リンクの遅延は短い方がプログラムの実行を高速化できる．なぜならリンクの遅延を短くすると，実行オーバヘッドが大きいインタプリタを使ってプログラムを実行する期間が短くなるからである．

3.2 JeanPaul における動的ロードの実現

JeanPaul は動的ロードを実現するために，実行時に静的コンパイル済みコードを使う前に，その有効性を確認する．有効性の確認は，3.1 節で述べたように，全仮定クラスについて，静的コンパイル時に参照したクラスファイルと実行時に動的ロードしたクラスファイルが同一か検査することでおこなう．

本節では JeanPaul における動的ロードの実現の詳細を示す．まず，JeanPaul においてクラスを動的ロードするタイミングを示す．次に，メソッドの全仮定クラスを収める集合を計算する手順を示す．

3.2.1 クラスを動的ロードするタイミング

Java の言語仕様はクラスを動的ロードするタイミングを規定していない．したがって，Java VM の実現にあたっては，このタイミングを定める必要がある．JeanPaul のように

静的コンパイル済みコードを利用する Java VM では，このタイミングが実行速度に影響する．なぜなら，このタイミングを早くすると，リンクの遅延が短くなるからである．ここではまず，このタイミングを操作してリンクの遅延を最小限にする方法を紹介し，次に JeanPaul においてクラスを動的ロードするタイミングを示す．最後に，JeanPaul ではクラスを動的ロードするタイミングがリンクの遅延に影響しないことと，その理由を示す．

リンクの遅延を最小限にする方法

リンクの遅延を最小限にするには，静的コンパイル時に参照した全クラスを Java VM の起動時に動的ロードすればよい．そうすれば静的コンパイル済みコードをリンクする作業を Java VM の起動時に一括して実施できる．

JeanPaul における動的ロードのタイミング

JeanPaul の Java VM では，静的コンパイル時に参照したクラスを特別扱いせず，他のクラスと同様にクラスを初期化する時点で動的ロードする．JeanPaul でリンクの遅延を最小限にする方法を採用しなかった理由は，リンクの遅延を最小限にする方法に次に示す 2 つの問題があったからである．

1. アプリケーションの起動時にクラスファイルを参照するためのファイルアクセスが集中し，起動に時間がかかる．
2. 実行する Java アプリケーションによっては挙動が不自然になる．挙動が不自然になるのは，実行時に生成したクラスファイルを動的ロードするアプリケーションである．このようなアプリケーションの作者は実行時に生成したクラスファイルを動的ロードすると考えるのが一般的だが，アプリケーションの起動時にあらかじめ読み込んでおく方法では，前にアプリケーションを実行したとき生成した古いクラスファイルを読み込んでしまう．

なお，JeanPaul ではクラスを初期化する時点で動的ロードするが，これより早いタイミングで動的ロードをおこなったとしても，JeanPaul の Java VM が静的コンパイル済みコードをリンクするタイミングは変らない．なぜなら 3.1 節で述べたように，JeanPaul の Java VM がクラスファイルの同一性を確認して静的コンパイル済みコードをリンクするのは，クラスを動的ロードした直後ではなく，クラスの初期化完了後だからである．

クラス初期化検査の除去

JeanPaul が静的コンパイル済みコードをリンクする作業をクラスの初期化完了後におこなう目的は，Java2C トランスレータにおいてクラス初期化検査を除去するためである．

たとえば図 2.1(a) に示す Java のソースコード上におけるクラスフィールド参照にクラ

ス初期化検査の除去を適用し，C 言語のソースコードに変換した結果を，図 2.1(c) に示す．図 2.1(c) のコードはクラス初期化検査をもたない．したがって実行時にクラスフィールドを定義するクラス C を初期化するより前に図 2.1(c) のコードを実行すると，クラス C を初期化するタイミングがおかしくなり，プログラムを正常に実行できなくなる．JeanPaul ではこの問題を次の手順で解決する．

1. 実行時に Java VM は，3.1 節で述べたように，クラスの初期化完了後にクラスファイルの同一性を確認する．したがって，メソッド $n()$ の仮定クラスの集合を S_n とするとき，メソッド $n()$ の静的コンパイル済みコードを Java VM にリンクするタイミングは，S_n に属する全クラスについて初期化とクラスファイルの同一性確認が済んだ後になる．
2. Java2C トランスレータはメソッド $n()$ の中にあるクラス C の初期化検査を除去したら，集合 S_n に C を追加する．これにより，メソッド $n()$ の静的コンパイル済みコードをリンクするタイミングが実行時にクラス C を初期化した後になることを保証する．

ここで提案したクラス初期化検査除去の実現は，2.3 節で示した過去の技法とは異なり，コードを上書きしないため Java2C トランスレータで利用できる．ただし，ここで提案した技法を使うとリンクの遅延が長くなる．つまり，実行時に検査対象のクラスの初期化が済むまで静的コンパイル済みコードをリンクできなくなり，その間の実行をインタプリタでおこなうことから，実行速度が低下しうる．ここで提案したクラス初期化検査除去の実現と，その副作用であるリンクの遅延の延長が実行速度に与える影響は，4 章で評価する．

3.2.2 仮定クラスの集合を計算する手順

3.1 節で述べたように，JeanPaul の Java2C トランスレータは動的ロードを実現するために，トランスレート対象の個々のメソッドについて，メソッドの全仮定クラスを収める集合を計算する．計算の手順を次に示す．

1. Java2C トランスレータは，トランスレート作業の開始にあたり，トランスレート対象の個々のメソッド $n()$ に対して集合 S_n と配列 N_n を用意する．

   - 集合 S_n はメソッド $n()$ の全仮定クラスを収める．集合 S_n の初期値は，メソッド $n()$ を定義するクラスが C であるとき，C だけを要素にもつ集合になる．これはメソッド $n()$ のトランスレート開始時点では，メソッド $n()$ の定義を含むクラス C だけを参照することを意味する．
   - 配列 N_n には，メソッド $n()$ の静的コンパイル済みコードから直接呼出しするメソッドを格納する．

   配列 N_n が集合でなく配列である理由は，インライン展開の実現を簡単にするため

である．インライン展開を適用すると直接呼出しが消失するので，その結果を N_n に反映する必要がある．反映する処理を，N_n から直接呼出し先のメソッドを 1 つ除去するだけの簡単な処理で実現するには，N_n を配列にして，$n()$ の中にある直接呼出し箇所と同じ数だけ，直接呼出し先のメソッドを記録しておく必要がある．

2. 全てのメソッドを C 言語のソースコードに変換する．個々のメソッド $n()$ を最適化する過程で別のクラスを参照した場合には集合 S_n に要素を追加し，直接呼出しを生成した場合には配列 N_n に要素を追加する．**Java2C** トランスレータが実施する最適化一覧を表 3.1 に示す．表 3.1 に示した最適化のうち，※印をつけたものが集合 S_n や配列 N_n に要素を追加する．表 3.1 に示した最適化の詳細と，最適化が集合 S_n や配列 N_n に与える影響については 3.3 節で述べる．ただし，クラス初期化検査の除去については 3.2 節で述べた．

   表 3.1 では最適化を，クラス間最適化とクラス内最適化に分類している．クラス間最適化は原則として集合 S_n や配列 N_n に要素を追加するので※印がついている．中には※印がついていないものもある．それは先行するクラス間最適化で参照したクラスのみを参照するクラス最適化である．

3. 全てのメソッドを C 言語のソースコードに変換しおわったら，個々のメソッド $n()$ について，メソッド $n()$ から直接呼出し経由で到達しうるメソッドの集合 M_n を計算する．計算の手順を次に示す．

   (a) 集合 M_n の初期値を空集合 ϕ とする．

   (b) 集合 M_n に配列 N_n が含む全メソッドを追加する．

   (c) 集合 M_n が含む個々のメソッド $m()$ について，配列 N_m が含む全メソッドを集合 M_n に追加する．

      - メソッドの追加にあたり，追加するメソッド $l()$ を集合 M_n が含んでいなかったら，メソッド $l()$ を集合 M_n に追加してから，配列 N_l が含む全メソッドも集合 M_n に追加する．

   集合 M_n が計算できたら，集合 M_n が含む個々のメソッド $m()$ について，集合 S_m が含む全クラスを集合 S_n に追加する．これでメソッド $n()$ の全仮定クラスから成る集合 S_n の計算は終了である．

直接呼出し経由で到達しうるメソッド $m()$ の仮定クラスを収める集合 S_m が含む全クラスを S_n に追加する目的は，まだ使えない静的コンパイル済みコードを直接呼出し経由で呼び出すことがないようにするためである．直接呼出し先 $m()$ の静的コンパイル済みコードが，クラスファイルの同一性確認が完了しておらず，まだ使えない状況にあるにもかかわらず，直接呼出元 $n()$ の静的コンパイル済みコードをリンクしてしまうと，直接呼出元 $n()$ の静的コンパイル済みコード経由でまだ使ってはいけない直接呼出し先 $m()$ の静的コンパイル済みコードを呼び出してしまい，プログラムが正常に実行できなくなる．

表 3.1: JeanPaul の Java2C トランスレータが実施する最適化一覧

種別	名称	詳述する節
クラス間最適化	フィールドおよびメソッドの静的解決 (※)	3.3.1
	冗長なキャストの除去 (※)	3.3.2
	仮想メソッド呼出しの高速化 (※)	3.3.3
	インライン展開 (※)	3.3.4
	例外発生検査の最適化 (集約，冗長除去)	6.2.2
	スタックトレース管理コードのオーバヘッド軽減	3.3.5
	冗長な null 検査の除去	3.3.6
クラス内最適化	クラス初期化検査の除去 (※)	3.2
	ループのピーリング	3.3.7
	冗長な配列添字検査の除去	3.3.8
	末尾再帰の除去	
	コピー伝播	
	定数伝播	

※印のついた最適化はリンクの遅延に影響を与える．

この問題を防ぐためには，直接呼出し元 $n()$ の静的コンパイル済みコードを実行時にリンクするタイミング t_n を，直接呼出し先 $m()$ の静的コンパイル済みコードをリンクするタイミング t_m と同時，あるいはそれより後にすればよい．ここで t_n が t_m と同時，あるいはそれより後であることを，$t_n \geq t_m$ と表現することにする．集合 S_m が含む全クラスを S_n に追加すれば，関係 $t_n \geq t_m$ が成立する．なぜなら t_n は，実行時に S_n が含む全クラスの初期化が完了した時点なので，$S_n \supseteq S_m$ であれば $t_n \geq t_m$ といえるからである．

3.3 Java2Cトランスレータが適用する最適化

本節では表 3.1 に示した Java2C トランスレータが適用する最適化について順次述べる．また，個々の最適化が 3.2 節で述べた集合 S_n や配列 N_n に与える影響を示す．ただし，表 3.1 に示した最適化のうち，クラス初期化検査の除去については 3.2 節で，例外発生検査の最適化については 6.2.2 節で述べているので，本節では述べない．また，末尾再帰の除去と定数伝播，コピー伝播も参考文献 [Aho86] に記載してある古典的最適化であり，特に Java2C トランスレータ固有のものではないので記述の対象から除く．

3.3.1 フィールドおよびメソッドの静的解決

フィールドおよびメソッドの静的解決とは，フィールドやメソッドのアドレスに関係する情報をコンパイル時に算出する最適化の総称であり，次に示す 3 種類の最適化から成る．

1. インスタンスフィールド参照の高速化
2. クラスフィールド参照の高速化
3. メソッドの直接呼出し

```
1:  int val = this.field;
```

(a) Java ソースコード

```
1:  offset = resolveInstanceField(ee,
2:                     ClassName,
3:                     FieldName,
4:                     FieldDescriptor);
5:  if (exceptionOccurred(ee))
6:     goto HANDLER;
7:  val = this[offset];
```

(b) C ソースコード（最適化なし）

```
1:  val = this[CONSTANT_FIELD_OFFSET];
```

(c) C ソースコード（最適化あり）

図 3.4: インスタンスフィールド参照

インスタンスフィールド参照の高速化

インスタンス参照の高速化ではインスタンスフィールドのオフセットを静的コンパイル時に計算することでインスタンスフィールド参照を高速化する．インスタンスフィールドのオフセットとは，インスタンスフィールドのインスタンスを表すデータ構造中における位置，つまりインスタンスフィールドがインスタンスを表すデータ構造中の先頭から何バイト目に存在するかを示す数で，インスタンスフィールドを参照する際に利用する．

図 3.4(a) に示したインスタンスフィールド参照をおこなう Java ソースコードについて，インスタンスフィールド参照の高速化を適用せずに C ソースコードへトランスレートした場合の結果を図 3.4(b) に，適用した場合の結果を図 3.4(c) に示す．図 3.4(c) の 1 行目にある定数 `CONSTANT_FIELD_OFFSET` が静的コンパイル時に計算したインスタンスフィールドのオフセットである．

Java2C トランスレータはインスタンスフィールド参照の高速化を適用するにあたり，適用先のインスタンスフィールド参照を含むメソッド $n()$ の仮定クラスを収める集合 S_n に次の操作 3.1 を施す．

$$S_n \leftarrow S_n \cup P(C) \tag{3.1}$$

操作 3.1 において，$P(C)$ はインスタンスフィールドを定義するクラス C とその全親クラスからなる集合を表す．したがって操作 3.1 は，S_n にクラス C とその全親クラスを追加する．

操作 3.1 を適用する理由は，インスタンスフィールドのオフセットを計算する際に，インスタンスフィールドを定義するクラスとその全親クラスを参照するからである．インス

タンスフィールドのオフセットはクラス階層の上位にあるクラスが定義するインスタンスフィールドから順に符番して定める．

インスタンスフィールド参照の高速化を適用せず，図3.4(b)のコードを生成する場合には，仮定クラスを追加する必要はない．図3.4 (b)のコードは実行時にインスタンスフィールドのオフセットを検索する関数`resolveInstanceField()`を呼び出すので，静的コンパイル後にクラスファイルの更新によってインスタンスフィールドのオフセットが変化しても正常に動作する．更新によってインスタンスフィールドが消失し，関数`resolveInstanceField()`が実行時例外を返戻した場合についても，図3.4(b)のコードは5行目，6行目を経由して例外ハンドラに制御を移すことができる．

一方，図3.4(c)のコードは，インスタンスフィールド参照を単なるメモリのオフセット参照で実現するので図3.4(b)のコードより動作は高速だが，静的コンパイル後のクラスファイルの更新によるオフセットの変更やインスタンスフィールドの消失といった事態に対処する機能をもたない．ただし，この機能不足が原因で実行時に問題が発生することはない．なぜならJava2Cトランスレータが操作3.1を適用した結果，クラスファイルの更新が発生した場合には図3.4(c)のコードが無効になるからである．

クラスフィールド参照の高速化

クラスフィールド参照の高速化では，クラスフィールドのアドレスを静的コンパイル時に計算してしまうことで高速化を図る．図3.5(a)のクラスフィールド参照をおこなうJavaソースコードについて，クラスフィールド参照の高速化を適用せずにCソースコードへトランスレートした場合の結果を図3.5(b)に，適用した場合の結果を図3.5(c)に示す．図3.5(c)のコードは，クラスフィールドが所属するクラス C を表すデータ構造 D_C の中にあるクラスフィールドを直接参照する．データ構造 D_C はJava2Cトランスレータが静的に割付ける．図3.5(c)の1行目にある定数`FIELD_OFFSET`は，参照するクラスフィールドが D_C 中のクラスフィールド格納領域`static_fields`のどこに格納してあるかを示すオフセットである．

Java2Cトランスレータはクラスフィールド参照の高速化にあたり，高速化対象のクラスタンスフィールド参照を含むメソッド $n()$ の仮定クラスの集合 S_n に次の操作3.2を施す．

$$S_n \leftarrow S_n \cup P(C) \tag{3.2}$$

操作3.2を適用する理由は，実行時にデータ構造 D_C が利用可能な場合に限って，クラスフィールド参照の高速化を適用したコード，つまり D_C 内にあるクラスフィールドを参照するコードを使いたいからである．クラスを表すデータ構造 D_C はJava2Cトランスレータが生成したもので，3.1節で述べたように，実行時に D_C を利用するためにはクラス C とその全親クラスについて，静的コンパイル時に参照したクラスファイルと動的ロードし

```
1:  int val = C.field;
```

(a) Java ソースコード

```
1:  address = resolveStaticField(ee,
2:                     ClassName,
3:                     FieldName,
4:                     FieldDescriptor);
5:  if (exceptionOccurred(ee))
6:    goto HANDLER;
7:  val = *address;
```

(b) C ソースコード（最適化なし）

```
1:  val = D_C.static_fields[FIELD_OFFSET].value;
```

(c) C ソースコード（最適化あり）

図 3.5: クラスフィールド参照の高速化

たクラスファイルが一致するか事前に確認する必要がある．操作 3.2 を適用すれば，この確認ができた場合に限ってクラスフィールド参照を高速化したコードが Java VM にリンクされる．確認ができていない場合にこのコードを使うと，無効なメモリ領域を参照してプログラムの実行がおかしくなる．

クラスフィールド参照の高速化を適用せず，図 3.5(b) のコードを生成する場合には，仮定クラスを追加する必要はない．図 3.4(b) のコードは実行時にクラスフィールドのアドレスを検索する関数 `resolveStaticField()` を呼び出す．この関数は，クラスフィールドを定義するクラスにクラス初期化検査を適用し，必要に応じてクラスを動的ロード，初期化した上でクラスを表すデータ構造内にあるクラスフィールドのアドレスを返戻するものとする．図 3.5(b) のコードによるクラスフィールド参照の実現には次の利点がある．

- 関数 `resolveStaticField()` がクラス初期化検査を適用するので，クラスが未初期化の状態で実行しても正常に動作する．
- 静的コンパイル後にクラスファイルの更新によってクラスフィールドのオフセットが変化しても正常に動作する．更新によってクラスフィールドが消失し，関数 `resolveStaticField()` が実行時例外を返戻した場合についても，図 3.5(b) の 5 行目，6 行目を経由して例外ハンドラに制御を移すことができる．

一方，図 3.5(c) のコードは，クラスフィールド参照を単なるメモリ参照で実現するので図 3.5(b) のコードより動作は高速だが，クラス初期化検査の除去を適用してあるため，クラスが未初期化の状態で使うとクラスを初期化するタイミングが狂って問題をおこす．また，静的コンパイル後のクラスファイルの更新によるオフセットの変更やクラスフィール

```
1:  int val = Class.method();
```
(a) Java ソースコード

```
1:  address = resolveStaticMethod(ee,
2:                      ClassName,
3:                      MethodName,
4:                      MethodDescriptor);
5:  if (exceptionOccurred(ee))
6:    goto HANDLER;
7:  val = address(ee);
```
(b) C ソースコード（最適化なし）

```
1:  val = methodTranlatedToC(ee);
```
(c) C ソースコード（最適化あり）

図 3.6: メソッドの直接呼出し

ドの消失といった事態に対処する機能をもたない．ただし，これらの機能不足が原因で実行時に問題が発生することはない．なぜなら Java2C トランスレータが操作 3.2 を適用した結果，クラスファイルの更新が発生した場合には図 3.4(c) のコードが無効になり，かつクラスフィールド参照を高速化したコードを実行時にリンクするタイミングがクラス初期化後になるからである．

メソッドの直接呼出し

メソッドの直接呼出しとは，メソッド呼出しを，C 言語にトランスレートしたメソッドへの直接呼出しとして実現することを意味する．メソッドの直接呼出しの適用対象となるメソッド呼出しは，次に示す 2 つの条件を満たす．

1. トランスレート時に呼出し先を唯一に定めることができる．
2. 呼出し先のメソッドが C 言語へのトランスレート対象になっている．

コンパイル時に呼出し先を唯一に特定できない仮想メソッド呼出しには，メソッドの直接呼出しを適用できない．仮想メソッド呼出しの高速化については後述する．

図 3.6(a) に示した Java のソースコードにおけるクラスメソッド呼出しについて，トランスレートにあたってメソッドへの直接呼出しを適用した場合に得られるコードを図 3.6(c) に，適用しなかった場合に得られるコードを図 3.6(b) に示す．

Java2C トランスレータはメソッドの直接呼出しの適用にあたり，適用先のメソッド，つまり呼出し元のメソッド $n()$ の配列 N_n に次の操作 3.3 を適用する．ここで$+$は配列に要素を追加する演算子とし，$method()$ は直接呼出し先のメソッドとする．

$$N_n \leftarrow N_n + method() \tag{3.3}$$

メソッドの直接呼出しを適用しなかった場合には直接呼出しが現れないので，操作 3.3 は不要である．メソッドの直接呼出しを適用しなかった場合に得られる図 3.6(b) のコードは次の手順でメソッド呼出しを実現する．

1. 呼出し対象のメソッドのクラス，名前，型を受け取ってメソッドのアドレスを検索する関数を呼び出す．
   - 図 3.6(b) の 1 行目で呼び出す関数 **`resolveStaticMethod()`** は，まず呼出し対象のメソッドを定義するクラスにクラス初期化検査を適用し，次にメソッドのアドレスを検索して返戻するものとする．
2. 検索して得たアドレスにあるコードを呼び出すことでメソッド呼出しを実行する．

図 3.5(b) のコードによるメソッド呼出しの実現には次の利点がある．

- 関数 **`resolveStaticMethod()`** がクラス初期化検査を適用するので，クラスが未初期化の状態で実行しても正常に動作する．
- 静的コンパイル後にクラスファイルの更新によってクラスメソッドが消失し，関数 **`resolveStaticMethod()`** が実行時例外を返戻した場合についても，図 3.6(b) の 5 行目，6 行目を経由して例外ハンドラに制御を移すことができる．

一方，図 3.6(c) のコードは，メソッド呼出しを単なる C の関数への直接呼出しで実現するので図 3.5(b) のコードより動作は高速だが，クラス初期化検査の除去を適用してあるため，クラスが未初期化の状態で使うとクラスを初期化するタイミングが狂って問題をおこす．また，静的コンパイル後のクラスファイルの更新によるクラスメソッドの内容変更やクラスメソッドの消失といった事態に対処する機能をもたない．ただし，これらの機能不足が原因で実行時に問題が発生することはない．なぜなら Java2C トランスレータが操作 3.3 を適用した結果，クラスファイルの更新が発生した場合には図 3.4(c) のコードが無効になり，かつメソッドの直接呼出しを適用したコードを実行時にリンクするタイミングが，呼出し先のメソッドの静的コンパイル済みコードが利用可能と確認できた後になるからである．

3.3.2　冗長なキャストの除去

冗長なキャストの除去では，例外を発生しないキャストを除去する．Java ソースコード上のキャストは，参照をキャストできるか検査し，キャストできなければ例外を投げるが，中には決して例外を投げないものもある．たとえば図 3.7 のソースコードでは 7，9，10 行

```
 1:  // Java ソースコード
 2:  class B extends A { ... }
 3:  class C extends B { ... }
 4:         ⋮
 5:  Object obj = tmp;
 6:  if (...)
 7:    ((B)obj).m1();
 8:  else
 9:    ((C)obj).m2();
10:  val = ((A)obj).field;
```

図 3.7: 冗長なキャスト

目にキャストがあるが，10 行目のキャストは例外を投げない．なぜなら先行する 7 行目や 9 行目でキャスト可能かすでに検査しているからである．したがって 10 行目のキャストは冗長であり除去できる．キャストが例外を投げうるか否か判定する手順を次に示す．

1. キャスト対象の参照の宣言クラスをフロー解析により取得し，集合 F に収める．集合 F を計算する手順を次に示す．
   (a) 集合 F の初期値を空集合 ϕ とする．
   (b) 検査対象のキャストから，キャスト対象の参照の全定義点に向ってあらゆるパスを経由して制御フローを遡り，遡る経路上に存在する文を順次検査して，キャスト対象の参照が指示するインスタンスのクラスに関する情報を与える演算を含むか調べる．この情報を与える演算とは，キャスト対象の参照に対する次の演算である．
      - 任意のクラス C へのキャストと，キャストの結果，例外が発生しなかった場合にのみ検査対象のキャストに到達する分岐
      - 任意のクラス C への instanceof 演算と，その結果が真であった場合にのみ検査対象のキャストに到達する分岐

      これらの演算を通過すると，参照が指示するインスタンスがクラス C のインスタンスであることが判る．そこで集合 F にクラス C を加える．また，情報が得られたことから，発見した演算を越えて定義点に遡ることをやめる．定義点に遡る別の経路がある場合には，別の経路から遡る．
   (c) 遡行の結果，定義点に到達した場合には，定義点から，参照の宣言クラスを求めて集合 F に加える．定義点がフィールド参照であればフィールドの宣言クラスを，メソッド呼出しであれば返戻値のクラスを集合 F に加える．
2. 集合 F の計算が完了したら，集合 F に属する全クラスがキャスト対象のクラス自身あるいはその子クラスか検査する．検査の結果が真であれば，キャストは例外を投げないと判断できる．

この手順に従って図 3.7 の 10 行目にあるかキャストが冗長か判定する過程を示す．まず集合 F を用意し，その値を ϕ とする．次にキャスト対象の参照 `obj` の定義点は 5 行目にあり，5 行目には 7 行目あるいは 9 行目経由で遡ることができるので，まず 7 行目経由で遡ることを試みる．すると 7 行目にはクラス `B` へのキャストがあり，キャストで例外が発生しなかった場合に限り 10 行目に到達するので集合 F に `B` を加え，7 行目経由で遡ることをやめる．次に 9 行目経由で遡ることを試みると，9 行目にはクラス `C` へのキャストがあり，キャストで例外が発生しなかった場合に限り 10 行目に到達する．そこで集合 F に `C` を加え，9 行目経由で遡ることをやめる．これで他に遡るパスがなくなったので，集合 F の計算は終りである．この時点で集合 F にはクラス `B`，`C` が入っているが，これらはいずれも 10 行目におけるキャスト対象のクラス `A` を継承する．従って 10 行目のキャストは冗長だと判断して除去できる．

なお，冗長なキャストの除去を適用すると，静的コンパイル後にクラスファイルを更新し，クラスの継承関係を変更した場合に問題がおきる．たとえばクラス `B` の親クラスをクラス `A` 以外にすると，10 行目のキャストは冗長でなくなるから，このときキャストを除去した静的コンパイル済みコードを使うとプログラムの実行がおかしくなる．

そこで Java2C トランスレータは冗長なキャストの除去を適用した場合，適用したメソッドの仮定クラスを表す集合 S_n に集合 F に属する全クラスと，その全親クラスを加え，冗長なキャストを除去したコードを実行時にリンクするより前に静的コンパイル時に参照したクラスの継承関係に変更がないことを確認させる．

3.3.3 仮想メソッド呼出しの高速化

JeanPaul では 2.4 節で示したディスパッチ表法と I-call if 変換を使って仮想メソッド呼出しを高速化する．I-call if 変換にはクラスチェック変換やメソッドチェック変換などいくつかのバリエーションがあるが，バリエーションによって集合 S_n，配列 N_n に与える影響が違うなど Java2C トランスレータで実現するにあたって考察すべき点が多い．そこで I-call if 変換については別途 5 章で考察する．ここでは，ディスパッチ表法が仮定クラスに与える影響について述べる．

ディスパッチ表法が仮定クラスに与える影響

Java2C トランスレータはメソッド n`()` 内にある仮想メソッド呼出し `obj.`m`()` をディスパッチ表法によって実現するとき，メソッド n`()` の仮定クラスを表す集合 S_n に次の操作を施す．ここでクラス C はメソッド m`()` を宣言するクラスとする．

$$S_n \leftarrow S_n \cup P(C) \tag{3.4}$$

操作 3.4 を適用する理由は，ディスパッチ表法でメソッド m`()` を呼び出すコードを生成

するには，まず，クラス C のディスパッチ表を作成して $m()$ のアドレスがディスパッチ表の何番目に登録してあるか計算する必要があるが，クラス C のディスパッチ表を作成するにはクラス C とその全親クラスを参照する必要があるからである．ディスパッチ表を作成する手順については 2.4 節で述べた．

3.3.4 インライン展開

インライン展開とは呼出し先のメソッド本体を呼出し元に展開する最適化である．静的コンパイラが生成するコードを直接呼出しするメソッド呼出しにはインライン展開を適用できる．

Java2C トランスレータはインライン展開の適用にあたり，メソッド `n()` の内部に適用対象のメソッド呼出し $m()$ があるとするとき，メソッド `n()` の仮定クラスを表す集合 S_n，配列 N_n に次の操作を施す．

- 配列 N_n 中のメソッド $m()$ を 1 つ捨てる．
- $S_n \leftarrow S_n \cup S_m$
- 配列 N_n に配列 N_m 中の要素を全て追加する．

この操作を施す理由は，インライン展開を適用すると，メソッド `n()` の中からメソッド `m()` への直接呼出しが 1 つ減り，代りにメソッド `m()` の本体がメソッド `n()` の中に展開されるからである．メソッド `m()` の本体内には別のクラスを参照して最適化したコードや別のメソッドの静的コンパイル済みコードへの直接呼出しが入っている場合がある．したがってメソッド `m()` の本体をメソッド `n()` の中に展開するならば，メソッド $m()$ の仮定クラスを表す集合 S_m と配列 N_m を，それぞれ集合 S_n と配列 N_n に加える必要がある．

3.3.5 スタックトレース管理コードのオーバヘッド軽減

ここでは JeanPaul におけるスタックトレースの実現方法を示し，次にスタックトレースの実現にともなうオーバヘッドを軽減する次の 3 種類の最適化について述べる．

1. 冗長なスタックトレース管理コードの除去
2. スタックトレース管理コードの補償コード領域への移動
3. スタックトレース管理コードの高速化

JeanPaul におけるスタックトレースの実現

JeanPaul では，実行時にメソッド `java.lang.SecurityManager.getClassContext()` を呼出した際にスタックトレースを計算可能にするために，各スレッドがスタックトレー

スを表すリスト形式のデータ構造を維持管理する．このリストの先頭には，メソッドの実行開始時点で，どのメソッドを実行しているかを示す情報を収めたセルをつなぎ，メソッドから返戻する時点で先頭のセルを破棄する．スタックトレースを計算する際には，このリストにつないであるセルを先頭から順次たどる．

スタックトレースを維持管理するためにセルをつなぎ，破棄するコードを**スタックトレース管理コード**と呼ぶことにする．JeanPaul の Java2C トランスレータはメソッドの出入口にスタックトレース管理コードを挿入する．スタックトレース管理コードを図 3.8 に示す．図 3.8 の 1 〜 4 行目で定義する型 `singleFrame` は実行中のメソッドを示すセルを表す．5 〜 9 行目で定義するマクロ `PushSingleFrame()` は `singleFrame` 型のセルをリストにつなぐ．Java2C トランスレータはマクロ `PushSingleFrame()` をメソッドの入口に挿入する．マクロ `PushSingleFrame()` の定義中に出てくる変数 `ee` はスレッド固有の資源を収める構造体を指示する．変数 `ee` が指示する構造体のメンバ `jp_current_frame` はスタックトレースをあらわすリストの先頭にあるセルを指示する．図 3.8 の 10 〜 12 行目で定義するマクロ `PopSingleFrame()` はリストの先頭にあるセルを破棄する．Java2C トランスレータはマクロ `PopSingleFrame()` をメソッドの出口に挿入する．13 行目以降のコードについてはスタックトレース管理コードの高速化のところで述べる．

スタックトレース管理コードをメソッドの出入口に挿入すると，実行速度的にもコードサイズ的にも大きなオーバヘッドが生じる．このオーバヘッドを軽減する最適化について順次述べる．

冗長なスタックトレース管理コードの除去

スタックトレース情報は，スタックトレースを計算するメソッド（`getClassContext()` など）の直接あるいは間接的な呼出し元にならないメソッドでは管理する必要がない．そこで冗長なスタックトレース管理コードの除去では，メソッドがスタックトレースを計算するメソッドの呼出し元になるかメソッド間に跨がって解析し，ならないならスタックトレース情報の管理コードを削除する．

スタックトレース管理コードの補償コード領域への移動

補償コードとは，実行頻度は低いが万一に備えて挿入するコードを意味する．スタックトレース管理コードの補償コード領域への移動では，スタックトレース管理コードを補償コードが存在する領域に移動できるか調べ，可能ならば移動することによってスタックトレース管理コードの実行頻度を引き下げ，実行を高速化する．

補償コードの例として，図 3.9(a) の Java ソースコードの 5 〜 7 行目にあるメソッド `val()` を C ソースコードにトランスレートした結果を図 3.9(b) に示す．トランスレートにあたっては，図 3.9(a) の 6 行目にある仮想メソッド呼出し `this.val0()` にクラスチェック

```
 1:  struct singleFrame{
 2:    void *previous_frame;
 3:    unsigned long method_id;
 4:  };
 5:  #define PushSingleFrame(ee, frame, mid) \\
 6:    (frame).previous_frame = \\
 7:      (ee)->jp_current_frame; \\
 8:    (frame).method_id = (mid); \\
 9:    (ee)->jp_current_frame = &(frame);
10:  #define PopSingleFrame(ee, frame) \\
11:    (ee)->jp_current_frame = \\
12:      (frame).previous_frame;
13:  #define DeclareMultiFrame(frame, depth) \\
14:    struct { \\
15:      void *previous_frame; \\
16:      unsigned long sp; \\
17:      unsigned long method_ids[(depth)]; \\
18:    } (frame)
19:  #define LinkMultiFrame(ee, frame) \\
20:    (frame).previous_frame = \\
21:      (ee)->jp_current_frame; \\
22:    (ee)->jp_current_frame = &(frame);
23:  #define UnlinkMultiFrame(ee, frame) \\
24:    (ee)->jp_current_frame = \\
25:      (frame).previous_frame;
26:  #define PushMultiFrame(frame, depth) \\
27:    (frame).sp = ((((depth) + 1) << 1) | 1);
28:  #define SwapMultiFrame(frame, mid, depth) \\
29:    frame.method_ids[(depth)] = (mid);
30:  #define PopMultiFrame(frame, depth) \\
31:    (frame).sp = (((depth) << 1) | 1);
```

図 3.8: スタックトレース管理コード

変換を適用した．図 3.9(b) の 7 〜 12 行目がクラスチェック変換後の仮想メソッド呼出しだが，ここで 11 行目の間接呼出しは，静的コンパイル時に予測したメソッド `C::val0` とは違うメソッドを呼ぶ際に使う．静的コンパイル時の予測が外れる可能性は低く，11 行目の間接呼出しを使う機会は少ない．したがって 11 行目の間接呼出しは補償コードである．

さて，図 3.9(b) のコードでは 6 行目と 13 行目でスタックトレース情報の積下しを実施しているが，これは 11 行目の間接呼出しの呼出し先が不定で，その呼出し先からスタックトレースを計算するメソッドを呼ぶ可能性があるためである．7 行目の if 文が 8 行目に分岐する確率が高く，その場合スタックトレース情報の積下しが無駄になることを考慮すると，積下しのコードは 11 行目の間接呼出しの前後に移動した方がよい．

スタックトレース管理コードの補償コード領域への移動を適用して **`PushSingleFrame()`** と **`PopSingleFrame()`** を補償コードの前後に移動した結果を図 3.9(c) に示す．図 3.9(c) のコードは静的コンパイル時の予測が外れない限りスタックトレース管理コードを実行しない．スタックトレース管理コードの補償コード領域への移動では，**`PushSingleFrame()`** と **`PopSingleFrame()`** が囲う領域内に補償コードがあり，補償コードの内部からのみスタックトレースを計算するメソッドを呼出しうる場合に，スタックトレース管理コードを補償コード領域に移動する．

スタックトレース管理コードの高速化

スタックトレース管理コードの高速化では，インライン展開したメソッドに関するスタックトレースの記録に，図 3.8 の 1 〜 12 行目に示したコードより高速な図 3.8 の 13 行目以降のコードを使う．

具体的には，インライン展開を適用したメソッドでは図 3.8 の 13 〜 18 行目にあるマクロ **`DeclareMultiFrame()`** を使ってトレース情報を収めるセルを確保する．セルの内部にはメソッド内部でのスタックトレース情報を保持する配列 **`method_ids`** と，メソッド内での呼出し深さを表すメンバ **`sp`** がある．実行時には，メソッドの入口で 19 〜 22 行目にあるマクロ **`LinkMultiFrame()`** を使ってセルをスタックトレース情報リストにリンクする．そして，メソッド内でインライン展開した部分への突入時にマクロ **`PushMultiFrame()`** と **`SwapMultiFrame()`**（26 〜 29 行目）を使ってメソッド内でのトレース情報を更新し，脱出時に 30 〜 31 行目のマクロ **`PopMultiFrame()`** で元に戻す．マクロ **`PushMultiFrame()`**，**`SwapMultiFrame()`**，**`PopMultiFrame()`** は，リストへのリンクやアンリンクをおこなわないため，**`PushSingleFrame()`** や **`PopSingleFrame()`** より実行コストが小さい．

また，マクロ **`PushMultiFrame()`**，**`SwapMultiFrame()`**，**`PopMultiFrame()`** を対象とした次に示す 2 種類の最適化を実現した．

1. **マクロのループ外移動** ループ不変な情報を記録するマクロをループ外に移動する．

2. **冗長なマクロの除去** **`sp`** を設定するマクロ **`PushMultiFrame()`**，**`PopMultiFrame()`** が

```
1:  class C{
2:    int val0(){
3:      return (this.value);
4:    }
5:    int val(){
6:      return (this.val0());
7:    }
8:  }
```

(a) Java ソースコード

```
1:  int C_val(ExecEnv *ee, Object *this){
2:    struct singleFrame frame;
3:    int result;
4:    int (*code)(ExecEnv *, Object *)
5:      = this->class.dispatch_table[C_val0_offset];
6:    PushSingleFrame(ee, frame, C_val_id);
7:    if (code == C_val0){
8:      result = this->value;
9:    }else{
10:     /* 補償コード領域 */
11:     result = code(ee, this);
12:   }
13:   PopSingleFrame(ee, frame);
14:   return (result);
15: }
```

(b) C ソースコード (最適化なし)

```
1:  int C_val(ExecEnv *ee, Object *this){
2:    struct singleFrame frame;
3:    int result;
4:    int (*code)(ExecEnv *, Object *)
5:      = this->class.dispatch_table[C_val0_offset];
6:    if (code == C_val0){
7:      result = this->value;
8:    }else{
9:      /* 補償コード領域 */
10:     PushSingleFrame(ee, frame, C_val_id);
11:     result = code(ee, this);
12:     PopSingleFrame(ee, frame);
13:   }
14:   return (result);
15: }
```

(c) C ソースコード (最適化あり)

図 3.9: 補償コード領域への移動

冗長か判定し，冗長であれば除去する．マクロが冗長と判断できる条件は，マクロから制御フローを順方向にたどるとき，全てのパスにおいてスタックトレースを計算しうるメソッド呼出しに遭遇するより前に，他の **PushMultiFrame()** あるいは **PopMultiFrame()** に遭遇するか，あるいはメソッドの出口に到達することである．

この条件を満たすマクロでは，**sp** を設定しても，スタックトレース計算ルーチンが設定した sp を参照するより前に，別のマクロが **sp** を上書きするか，あるいはメソッドの出口に到達して，マクロ **UnlinkMultiFrame()** が **sp** を保持するセルを捨ててしまう．したがって，この条件を満たすマクロを冗長と判断して除去できる．

なお，図 3.8 において，**Push** の動作を **PushMultiFrame()** と **SwapMultiFrame()** に分けた理由は，冗長なマクロの除去では **PushMultiFrame()** のみ除去可能であり，**SwapMultiFrame()** を除去できないからである．

これらの最適化について図 3.10 を使って詳述する．図 3.10 (a) の Java ソースコード中にあるメソッド **m1()** を C ソースコードにトランスレートした結果を図 3.10(b) および (c) に示す．トランスレートにあたっては図 3.10(a) の 4 行目にあるメソッド呼出しにインライン展開を適用し，スタックトレース管理コードの高速化を適用した．図 3.10(c) のコードには，さらに，冗長なマクロの除去とマクロのループ外移動を適用した．

図 3.10(b) のコードと図 3.10(c) のコードを比較すると，最適化の結果として図 3.10(b) のコードの 5，15，18，22 行目にあったマクロが消失し，10，11 行目にあったマクロがループ外に移動していることが判る．

図 3.10(b) の 5 行目にある **PushMultiFrame()** が消失するのは，後続する 10 行目の **PushMultiFrame()** あるいは 22 行目の **PopMultiFrame()** の内部で **sp** を更新するまでの間にスタックトレースを計算しうるメソッド呼出しが存在せず，5 行目で **PushMultiFrame()** を実行して **sp** に値を与えることに意味がないからである．5 行目に後続する 7 行目にある関数呼出し **AllocIntArray()** では例外 **OutOfMemoryError** を生成することがあり，例外の生成にあたってはスタックトレースの取得がおきるが，**JeanPaul** では例外発生時に取得するスタックトレースに限っては正確な情報を与えなくてもよいという方針をとった．これは例外発生時に取得するスタックトレースが，開発者にデバッグ用のメッセージを与える用途にのみ使われることを考えると[1]，組込み機器向け Java VM である JeanPaul ではデバッグ用のメッセージよりも，最適化の機会を増やして，たとえば 5 行目のマクロを除去してコードサイズを削減する方が重要だと判断したためである．

図 3.10(b) の 15 行目にある **PopMultiFrame()** が消失するのは，後続する 22 行目で **sp** を更新するまでの間にスタックトレースを計算しうる関数呼出しが存在しないためで，同様の理由から 18 行目のマクロも消失する．22 行目の **PopMultiFrame()** が消失するのは直後の 23 行目で **frame** を捨てるので，22 行目で **sp** を更新する意味がないからである．

[1] 例外オブジェクトが出力するスタックトレースのメッセージは Java 実行系に依存して変わるので，それがプログラムの実行に影響を与えるとは考えにくい．

```
 1:  int[] m1(){
 2:    int result[] = new int[10];
 3:    for(int i=0; i<10; i++)
 4:      result[i] = this.m2();
 5:    return (result);
 6:  }
 7:  final int m2(){return (this.m3());}
 8:  final int m3(){ ··· }
```

(a) Java ソースコード

```
 1:  Object *m1(ExecEnv *ee, Object *this){
 2:    DeclareMultiFrame(frame, 2);
 3:    Object *result; int i, itemp;
 4:    LinkMultiFrame(ee, frame);
 5:    PushMultiFrame(frame, 0);
 6:    SwapMultiFrame(frame, 0, m1_id);
 7:    result = AllocIntArray(ee, 10);
 8:    if (exceptionOccurred(ee)) goto RETURN;
 9:    for(i=0; i<10; i++){
10:      PushMultiFrame(frame, 1);
11:      SwapMultiFrame(frame, 1, m2_id);
12:      { /* m2 の本体 */
13:        itemp = m3(ee, this);
14:        if (exceptionOccurred(ee)){
15:          PopMultiFrame(frame, 1);
16:          goto RETURN;
17:      }}
18:      PopMultiFrame(frame, 1);
19:      result[i] = itemp;
20:    }
21:  RETURN:
22:    PopMultiFrame(frame, 0);
23:    UnlinkMultiFrame(ee, frame);
24:    return (result);
25:  }
```

(b) C ソースコード（最適化なし）

```
 1:  void m1(ExecEnv *ee, Object *this){
 2:    DeclareMultiFrame(2);
 3:    Object *result; int i, itemp;
 4:    LinkMultiFrame(ee, frame);
 5:    SwapMultiFrame(frame, 0, m1_id);
 6:    result = AllocIntArray(ee, 10);
 7:    if (exceptionOccurred(ee)) goto RETURN;
 8:    PushMultiFrame(frame, 1);
 9:    SwapMultiFrame(frame, 1, m2_id);
10:    for(i=0; i<10; i++){
11:      itemp = m3(ee, this); /* m2 の本体 */
12:      if (exceptionOccurred(ee)) goto RETURN;
13:      result[i] = itemp;
14:    }
15:  RETURN:
16:    UnlinkMultiFrame(ee, frame);
17:    return (result);
18:  }
```

(c) C ソースコード（最適化あり）

図 3.10: インライン展開を施したメソッドでのスタックトレースの実現

図 3.10(b) の 10 行目と 11 行目のマクロをループ外に移動できるのはループ不変であるためである．不変式のループ外移動は大概の C コンパイラに実現してある最適化である．Java2C トランスレータの開発にあたっては，開発コストを少なくする意味からも，トレンスレータの維持管理にかかるコストを少なくする意味からも，C コンパイラで実施する最適化を重ねて開発するのは忌避すべきだが，JeanPaul ではスタックトレース向けにあえて不変式のループ外移動を開発した．これは C コンパイラの最適化ではスタックトレースに関するコードに最適化を施せない場合が多くあるためである．たとえば図 3.10(b) の 10 行目と 11 行目にあるマクロをループ外に移動する最適化は，C コンパイラによる不変式のループ外移動では実行できない．なぜなら直後の 13 行目にある関数呼出しで `frame` へのポインタを収めた構造体を参照する変数 `ee` を引数に与えており，C コンパイラはこの関数呼出しの結果として `frame` の内容が変わりうると推定して 10 行目と 11 行目のマクロを不変式とみなさないからである．同様の理由から C コンパイラのデータフロー最適化では 5 行目の `PushMultiFrame()` を除去できない．

なお，スタックトレースを保持するリストに，インライン展開ありの場合向けのセルと，そうではない場合向けセルを接続すると，スタックトレース情報の計算にあたって，セルがどちらの種類か判別する方法が必要になる．図 3.8 のマクロでは，2 番目のフィールドの第 0 ビットを使ってセルの種類を判別可能にしている．インライン展開を適用したメソッド向けのセルでは 2 番目のフィールドに `sp` が入っており，マクロ `PushMultiFrame()` と `PopMultiFrame()` で，その下 1 ビットを必ず 1 にしている．一方で `singleFrame` では，2 番目のフィールドに実行中のメソッドを表す固有の番号 `method_id` が入っているが，Java2C トランスレータはこの番号を付けるにあたって番号の下 1 ビットを 0 にする．この結果，実行時には 2 番目のフィールドの下 1 ビットを参照すればフレームの種別を判別できる．なお，マクロ `PushMultiFrame()`，`PopMultiFrame()` の中にあるシフトや論理和が実行オーバヘッドの元となることはない．なぜなら Java2C トランスレータがこれらのマクロにあたえる引数 `depth` の値は定数なので，シフトや論理和が C コンパイル時に静式評価の対象となるからである．

3.3.6 冗長な null 検査の除去

冗長な null 検査の除去では，null 検査が例外を投げうるか検査し，例外を投げえないならば，null 検査を冗長とみなして除去する．2 章で述べたように，Java2C トランスレータでは暗黙の null 検査を利用できない．そこで JeanPaul の Java2C トランレータは null 検査を明示的な分岐によって実現するが，明示的な分岐は実行速度的な意味からも，コードサイズ的な意味からもオーバヘッドの元となる．そこで JeanPaul の Java2C トランスレータではメソッド間に跨がるフロー解析を使って冗長な null 検査を検出して除去する．

フロー解析を使った冗長な null 検査の除去について，図 3.11(a) に示した Java ソース

コードを使って述べる．図3.11(a)のコードは配列`array`に収めてある数値の合計を求めるプログラムだが，最適化を適用しない場合，1行目で配列の長さを参照するところと，2行目で配列要素を参照するところでnull検査を実施する．最適化を適用せずに図3.11(a)に示したJavaソースコードをCソースコードに変換した結果を図3.11(b)に示す．図3.11(b)の3行目にあるnull検査は配列の長さを参照するところから生じたものであり，7行目のnull検査は配列要素の参照から生じたものである．ところで7行目のnull検査はフロー解析をおこなうと冗長であることを証明できる．なぜなら7行目のnull検査に到達するには3行目のnull検査を経由せねばならず，3行目のnull検査で`array`がnullではない場合に限って7行目方面に制御が移るからである．したがって7行目のnull検査は除去できる．

3.3.7 ループのピーリング

ループのピーリングはループ本体を1回分展開する．ループのピーリング自体にプログラムの実行を高速化する効果はないが，冗長なnull検査の除去や共通式除去など別の最適化を適用する機会を増やす効果がある．

たとえば図3.11(b)のプログラムでは，フロー解析による冗長なnull検査の除去によって7行目のnull検査を除去しても，まだ3行目にnull検査が残っている．3行目はループの内部にあたり，null検査を残すと実行速度に大きな悪影響を与える．このような場合にはループをピーリングしてnull検査をループ外に追い出すことができる．

図3.11(a)のプログラムにループのピーリングを適用した上で冗長なnull検査の除去を適用した結果を図3.11(c)に示す．図3.11(c)の2〜8行目がピーリングの結果生じたコードであり，ループの1回目の実行をおこなう．2回目以降の実行は9〜14行目がをおこなう．2〜8行目のコードと，9〜14行目を比較すると，3行目にあるnull検査が，ループ本体にあたる9〜14行目ではなくなっていることが判る．これはループの1回目の実行でnull検査を実施した結果，2回目以降の実行ではnull検査が不要なった結果であり，このことからループのピーリングがnull検査をループ外に追い出したことが判る．

3.3.8 冗長な配列添字検査の除去

冗長な配列添字検査の除去では，配列参照時の添字検査が冗長か，添字の値域を調査して判定し，冗長ならば添字検査を除去する．添字検査とは，配列参照時に添字を検査して，添字が配列の長さの範囲内にあるか調べ，範囲外ならば実行時例外`ArrayIndexOutOfBounds`を投げる動作を意味する．Javaの言語仕様は不正なメモリ参照の防止を目的として，配列の範囲外を参照したときに実行時例外を投げるよう規定している．

図3.11(a)のJavaソースコードの2行目に配列参照がある．この配列参照に相当するCソースコードが，図3.11(b)の7〜12行目にあるが，そのうち9〜11行目が添字検査をおこなう．ただしこの添字検査は冗長である．なぜなら図3.11(a)のコードを見ると添字

```
 1:   for(int i=0; i<array.length; i++)
 2:     sum += array[i];
```

(a) Java ソースコード

```
 1:   int i=0;
 2:  LOOP:
 3:   if (array == NULL)
 4:     goto THROW_NullPointerException;
 5:   if (!(i<array->length))
 6:     goto EXIT;
 7:   if (array == NULL)
 8:     goto THROW_NullPointerException;
 9:   unsigned int len = array->length;
10:   if ((unsigned int)i >= len)
11:     goto THROW_ArrayIndexOutOfBounds;
12:   sum += array[i];
13:   i = i + 1;
14:   goto LOOP;
15:  EXIT:
```

(b) C ソースコード（最適化なし）

```
 1:   int i=0;
 2:  PEEL:
 3:   if (array == NULL)
 4:     goto THROW_NullPointerException;
 5:   if (!(i<array->length))
 6:     goto EXIT;
 7:   sum += array[i];
 8:   i = i + 1;
 9:  LOOP:
10:   if (!(i<array->length))
11:     goto EXIT;
12:   sum += array[i];
13:   i = i + 1;
14:   goto LOOP;
15:  EXIT:
```

(c) C ソースコード（最適化あり）

図 3.11: ループピーリングと冗長な null 検査および配列添字検査除去

i の値域が $0 \leq \mathtt{i} < \mathtt{array.length}$ であり，2 行目の配列参照において例外が発生しえないことが判るからである．冗長な配列添字検査の除去ではプログラムを解析して配列参照において例外が発生しうるか調べ，2 行目の配列参照については例外が発生しえないので配列添字検査が冗長であると判断して除去する．この結果，図 3.11(b) の最適化なしのコードでは 9 〜 11 行目にあった配列添字検査が，図 3.11(c) の最適化済みコードではなくなっていることが判る．

第4章　クラス初期化検査の除去が実行速度に与える影響

3.2 節で述べたように，JeanPaul の Java2C トランスレータではクラス初期化検査にともなうオーバヘッドを軽減するため，クラス初期化検査を除去し，代りに，実行時にクラスの初期化が済むまで静的コンパイル済みコードのリンクを遅らせる．

本章では，一般的なアプリケーションプログラムの実行結果から，クラス初期化検査を除去する最適化が実行速度に与える影響を評価する．まず 4.1 節で評価環境を示し，次に 4.2 節でクラス初期化検査を除去することで実行速度をどれだけ改善できるか評価する．最後に，4.3 節でクラス初期化検査の除去にともなってリンクの遅延を延長したことが実行速度に与える影響を評価する．評価の結果，クラス初期化検査を除去することでアプリケーションの実行を 45%高速化できることが判った．また，リンクの遅延が長くなることによる実行速度の低下は 1%未満であることが判った．

4.1　評価環境

本論文では評価に次の共通の環境を利用する．

評価対象 SPECjvm98 [SPEC98] とする．SPECjvm98 は javac など中〜小規模の実用的アプリケーション 7 本を構成要素とするベンチマークである．SPECjvm98 を構成するアプリケーションの概要を表 4.1 に示す．

問題サイズ SPECjvm98 の問題サイズは 100 とする．SPECjvm98 の問題サイズとは，ベンチマークプログラムがおこなう計算量を示す指標である．ベンチマークの実行時に実行時オプションとして指定でき，1，10，100 のいずれかの値をとる．値が大きいほど計算量が大きくなる．デフォルト値は 100 である．

ヒープサイズ SPECjvm98 を構成する 7 項目のうち，`_201_compress` と `_213_javac` の 2 項目ではヒープ不足回避のため初期サイズ，上限サイズとも 128Mbyte に指定する．残りの 5 項目はデフォルトの初期サイズ 1Mbyte，上限サイズ 16Mbyte とする．

実行機械 日立製作所製ワークステーション HITACHI3500/540MP で実行をおこなう．このワークステーションのプロセッサは PA7200 100MHz であり，主記憶 256Mbyte を搭載する．また，OS は HI-UX/WE2 を使う．

表 4.1: SPECjvm98 を構成するベンチマーク

ベンチマーク項目	内容
`_201_compress`	データファイルの圧縮伸張
`_202_jess`	エキスパートシステムによる推論
`_209_db`	データベースの操作
`_213_javac`	Java ソースコードからバイトコードへのコンパイル
`_222_mpegaudio`	音楽ファイルの圧縮伸張
`_227_mtrt`	マルチスレッドによるレイトレース
`_228_jack`	構文解析系の生成

Java VM JeanPaul を用いる．JeanPaul の静的コンパイラを構成する C コンパイラとしては日立製作所製の最適化 C コンパイラを使う．

静的コンパイル対象のクラス JeanPaul の静的コンパイラにコンパイルするよう指示するクラスは，SPECjvm98 を実行する過程で Java VM が動的ロードする全クラスとする．

最適化 Java2C トランスレータは論文中で実行しないと明記した最適化を除き，表 3.1 に示す全最適化を適用する．C コンパイラの最適化オプションは`-O` とする．

4.2 実行速度への影響

図 4.1 に JeanPaul においてクラス初期化検査の除去が実行速度に与える影響を示す．図 4.1 では，クラス初期化検査を除去した場合としなかった場合，およびインタプリタのみでアプリケーションを実行した場合の 3 つの場合の実行速度を比較する．クラス初期化検査を除去しないと図 2.1(b) に示したクラス初期化検査が静的コンパイル済みコード内に残る．図 4.1 の縦軸はクラス初期化検査を除去した場合の実行速度を 100 とした場合の相対速度を表し，横軸は測定対象の SPECjvm98 を構成する個々のベンチマーク項目名を表す．

図 4.1 において，クラス初期化検査を除去した場合としなかった場合を比較すると，除去しなかった場合の実行速度が，相乗平均で除去した場合の 69%にとどまることが判る．これは，クラス初期化検査の除去によって相乗平均で実行速度が 45%速くなることを意味する．また，全てのベンチマーク項目で，クラス初期化検査の除去によって実行速度が向上していることが判る．

図 4.1 において，クラス初期化検査を除去した場合とインタプリタで実行した場合を比較すると，インタプリタの実行速度が明確に (約 2 〜 18 倍，平均で 3.5 倍) 劣っていることが判る．このことは，クラス初期化検査を除去した代りにリンクの遅延，すなわちインタプリタで実行する期間が長くなると，クラス初期化検査を除去しない代りにリンクの遅

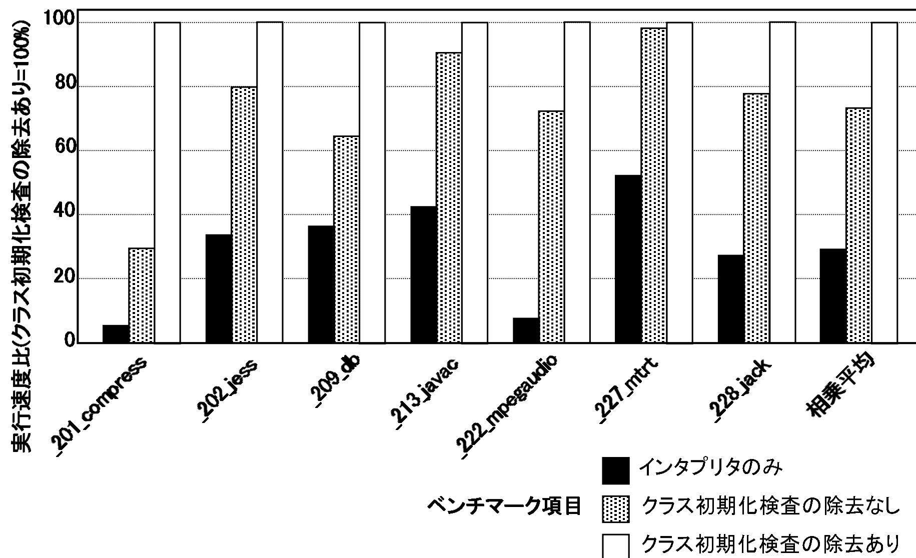

図 4.1: クラス初期化検査の除去が実行速度に与える影響

延も長くしない場合よりもむしろ実行速度が遅くなりうることを意味する．しかし，図 4.1 でクラス初期化検査を除去した場合としなかった場合を比較した限りでは，クラス初期化検査を除去する方が実行を高速化できるといえる．

4.3 リンクの遅延から生じる影響

図 4.2 に，Java2C トランスレータにおいて最適化を適用する代りにリンクの遅延を延長することが実行速度に与える影響を調査した結果を示す．調査はベンチマークを 8 回連続実行し，1 回目の実行速度と 2 回目以降の実行速度を比較することでおこなった．2 回目以降の実行速度とは，2 回目から 8 回目まで 7 回の実行速度のうち，最も高速だった回と最も低速だった回を除いた 5 回の実行速度の相加平均を意味する．図 4.1 の実行速度は 2 回目以降の実行速度である．ベンチマークは利用する全クラスを 1 回目の実行中に初期化するので，リンクの遅延が影響するのは 1 回目の実行のみであり，2 回目以降では静的コンパイル済みコードのみでベンチマークを実行する．したがって，1 回目の実行速度と 2 回目以降の実行速度を比較することにより，リンクの遅延が実行速度に与える影響を推察できる．図 4.2 に，クラス初期化検査を除去した場合としなかった場合のそれぞれについて，1 回目と 2 回目以降の実行速度の比較を示す．図 4.2 の縦軸は 2 回目以降の実行速度を 100 とした場合の相対実行速度を表し，横軸は測定対象の SPECjvm98 を構成する個々のベンチマーク項目名を表す．

図 4.2 から，1 回目と，2 回目以降の実行速度に余り差がないことが判る．たとえばリンクの遅延が大きく影響するクラス初期化検査を除去した場合についてみると，1 回目の実行速度が 2 回目以降より遅くなる度合は相乗平均で 0.8%であり，1%に満たない．差が最

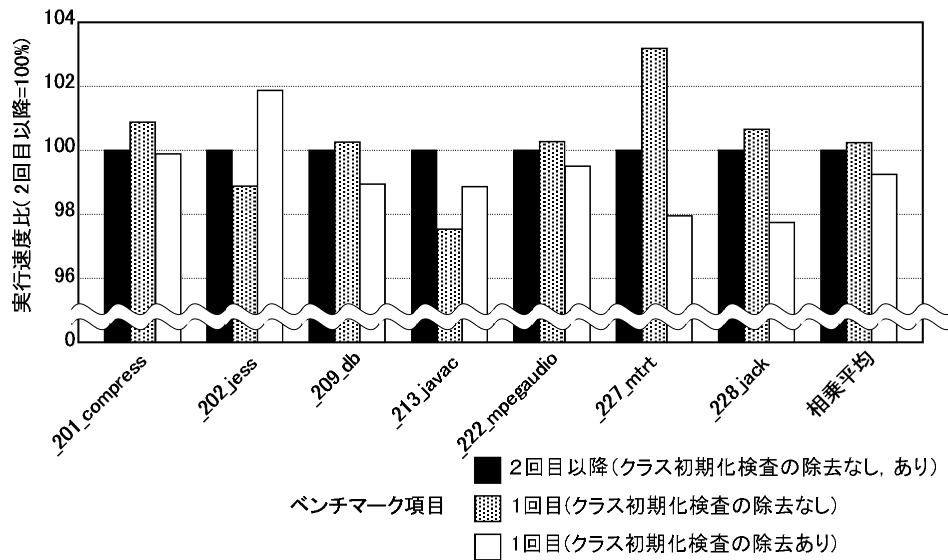

図 4.2: リンクの遅延が実行速度に与える影響

大の_228_jack でも 2%余りである．1 回目の実行が遅くなる原因がリンクの遅延以外にもメモリの初期化などいくつかあることを考慮すると，Java2C トランスレータにおいて最適化を適用する代りにリンクの遅延が長くなることが実行速度に与える影響は小さいといえる．

第5章　I-call if変換の実現

I-call if 変換は，検索鍵にクラスとメソッドのどちらを使うか，また，限定的か否かという点から 4 種類に分類できる．本章では，Java2C トランスレータで 4 種類の I-call if 変換を使い分ける方法を検討する．SPECjvm98 による評価結果から，Java2C トランスレータでは検索鍵にメソッドを使い，可能であれば限定的変換を施すべきであることが判った．

本章の構成を示す．まず 5.1 節で個々の I-call if 変換が仮定クラスに与える影響を示し，非限定的メソッドチェック変換から生じる動的ロードに関係したオーバヘッドが，他の I-call if 変換から生じるオーバヘッドより小さいことを指摘する．次に 5.2 節で 4 種類の I-call if 変換が実行速度に与える影響を評価し，I-call if 変換を使い分ける方法を検討する．最後に 5.3 節で Java2C トランスレータと組合せて利用する C コンパイラの実現によってはメソッドチェック変換を使うと大きなオーバヘッドが発生しうることを示す．

5.1　I-call if変換が仮定クラスに与える影響

メソッド呼出し `obj.`m`()` に 4 種類の I-call if 変換をそれぞれ適用した結果，Java2C トランスレータが生成する C ソースコードと，呼出し元のメソッド n`()` の仮定クラスを表す集合 S_n と配列 N_n に追加する要素を表 5.1 に示す．本節ではまず，表 5.1 中の C ソースコードに現れる記号 D_{C_k} などの意味を明らかにし，次に 4 種類の I-call if 変換が追加する仮定クラスの違いについて述べる．

5.1.1　I-call if 変換を適用したコードに関する注意事項

表 5.1 では，`obj` の候補クラスを $\{C_1, \cdots, C_k\}$ とし，クラスチェック変換の比較候補とした．メソッドチェック変換の比較候補 $\{C'_1 :: m, \cdots, C'_l :: m\}$ は，メソッド $C_1 :: m, \cdots, C_k :: m$ から重複要素を除いたものである．表 5.1 中の C ソースコードについて詳述する．

- $C :: m$ はクラス C に対応するメソッド m`()` の静的コンパイル済みコードを意味する．したがって，直接呼出し $C :: m$`()` は Java2C トランスレータが生成したコードを直接呼び出す．
- メソッドチェック変換のコード中にある式 `code == &(`$C :: m$`)` の値は，`code` が C に対応するメソッド m`()` の静的コンパイル済みコードのアドレスを参照する場合に真になる．

表 5.1: I-call if 変換の分類

名称	非限定的クラスチェック変換	限定的クラスチェック変換

最適化を適用した
C ソースコード
	klass = obj->class;
	if (klass == &D_{C_1}){
	  $C_1 :: m$(obj);
	}else if (klass == &D_{C_1}){
	⋮
	}else if (klass == &D_{C_k}){
	  $C_k :: m$(obj)
	}else{
	  code=obj->class.dispatch_table[O_m];
	  code(obj);
	}
		klass = obj->class;
		if (klass == &D_{C_1}){
		  $C_1 :: m$(obj);
		}else if (klass == &D_{C_2}){
		⋮
		}else{
		  $C_k :: m$(obj);
		}

呼出し元 n() への 仮定クラスの追加	$S_n \leftarrow S_n \cup P(C_0)$ $N_n \leftarrow N_n + [C_1 :: m, C_2 :: m, \cdots C_k :: m]$	$S_n \leftarrow S_n \cup P(C_1) \cup \cdots \cup P(C_k)$ $N_n \leftarrow N_n + [C_1 :: m, C_2 :: m, \cdots C_k :: m]$
名称	非限定的メソッドチェック変換	限定的メソッドチェック変換

最適化を適用した
C ソースコード
	code=obj->class.dispatch_table[O_m];
	if (code == &($C'_1 :: m$)){
	  $C'_1 :: m$(obj);
	}else if (code == &($C'_2 :: m$)){
	⋮
	}else if (code == &($C'_l :: m$)){
	  $C'_l :: m$(obj);
	}else{
	  code(obj);
	}
		code=obj->class.dispatch_table[O_m];
		if (code == &($C'_1 :: m$)){
		  $C'_1 :: m$(obj);
		}else if (code == &($C'_2 :: m$)){
		⋮
		}else{
		  $C'_l :: m$(obj);
		}

呼出し元 n() への 仮定クラスの追加	$S_n \leftarrow S_n \cup P(C_0)$	$S_n \leftarrow S_n \cup P(C_0) \cup \cdots \cup P(C_k)$ $N_n \leftarrow N_n + [C'_1 :: m, C'_2 :: m, \cdots C'_l :: m]$

※メソッド m() はクラス C_0 が宣言するものとする.

- クラスチェック変換のコード中にある式 **klass == &**D_{C_k} は，レシーバ **obj** のクラスを表すデータ構造のアドレス **klass** と，Java2C トランスレータが生成したクラス C_k を表すデータ構造 D_{C_k} のアドレスが一致するか比較する．この式の値が真となるのは次の 2 つの条件を満たす場合である．

  1. レシーバ **obj** のクラスが C_k である．
  2. クラス C_k とその全親クラスについて，静的コンパイル時に参照したクラファイルと動的ロードしたクラスファイルが一致する．

  条件 2 は Java2C トランスレータが生成したデータ構造 D_{C_k} を実行時に利用するために必要な条件である．

5.1.2 追加する仮定クラスの違い

表 5.1 から，I-call if 変換の種類によって追加する仮定クラスが異なることが判る．ここでは，I-call if 変換の種類によって追加する仮定クラスが違う理由を示す.

追加する仮定クラスの違いは Java2C トランスレータにとって大きな意味をもつ．なぜなら仮定クラスの追加が実行時オーバヘッドに影響を与えるからである．一般に仮定クラスを追加するほど，リンクの遅延，つまり実行時にインタプリタを使う期間が長くなり，実行速度が低下する．したがって仮定クラスを追加する最適化の実現にあたり，Java2C

トランスレータの設計者は仮定クラスの追加による実行時オーバヘッドの増加と，最適化によるオーバヘッドの軽減の双方に配慮する必要がある．

4 種類の I-call if 変換が追加する仮定クラスを比較すると，最も追加要素が少ないのは非限定的メソッドチェック変換であることが判る．非限定的メソッドチェック変換を適用した場合に追加する要素は，非限定的メソッドチェック変換の C ソースコードの 1 行目でディスパッチ表を検索するために必要な $P(C_0)$ のみである．非限定的メソッドチェック変換は，直接呼出しを生成するにもかかわらず，直接呼出し対象のメソッドを配列 N_n に追加しない．その理由を次に示す．

- 3.2 節で示したように，直接呼出し対象のメソッドを配列 N_n に追加する目的は，直接呼出しが実行時に利用可能か確認できていない状態の静的コンパイル済みコードを呼び出すことを防ぐことにある．JeanPaul の Java VM は，配列 N_n 中にあるメソッドの静的リンク済みコードを，メソッド n() の静的リンク済みコードより先にリンクする．

  一方で，非限定的メソッドチェック変換のコードは，確かに直接呼出しを含んではいるが，リンクが済んだあとに限って静的コンパイル済みコードを直接呼出しする．このため，非限定的メソッドチェック変換を適用する場合には，配列 N_n への要素の追加が必要にならない．

  非限定的メソッドチェック変換のコードがリンクが済んだあとに限って静的コンパイル済みコードを直接呼出しするのは，式 `code == &(`$C :: m$`)` の値が真である場合に限り $C :: m$ を直接呼出しするからである．`code` にはディスパッチ表から取得した関数のアドレスが入っているから，式 `code == &(`$C :: m$`)` の値が真になるのは少なくとも Java VM が $C :: m$ を利用可能とみなし，リンクテーブルの一部であるディスパッチ表に $C :: m$ を登録した後になる．

一方，非限定的クラスチェック変換はディスパッチ表に登録してあるメソッドのアドレスを比較する訳ではないから直接呼出し対象のメソッドを配列 N_n に追加する必要がある．

限定的メソッドチェック変換ではディスパッチ表に登録してあるメソッドのアドレスを比較するが，最後の 1 候補について比較を省略するので，直接呼出し対象のメソッドを配列 N_n に追加する必要がある．さもないと，間違ったタイミングで限定的メソッドチェック変換を適用したコードをリンクして，プログラムの実行がおかしくなる．たとえばレシーバ `obj` のクラスが C'_2 であり，静的コンパイル済みコード $C'_2 :: m$ がまだ利用可能でない状況で表 5.1 の限定的メソッドチェック変換のコードを実行すると，$C'_l :: m$ を呼び出してしまうが，これはおかしい．この問題を回避するには，直接呼出し対象のメソッドより後でメソッドチェック変換を適用したコードをリンクすればよい．このリンクの順序関係を実現するために，直接呼出し対象のメソッドを配列 N_n に追加するのである．

```
// クラスチェック変換
LDW         4(0,%r25),%r29              ;%r29 ← obj->class
LDW         T'D_{C1}(0,%r19),%r31       ;%r31 ← &D_{C1}
COMBF,=,n   %r29,%r31,NEXT_CANDIDATE    ;if (%r29 != %r31) goto NEXT_CANDIDATE
```

```
// メソッドチェック変換
LDW         4(0,%r25),%r22              ;%r22 ← obj->class.dispatch_table
LDW         -52(0,%r22),%r1             ;%r1 ← dispatch_table[O_m]
LDW         T'$sdp(0,%r19),%r31         ;%r31 ← シンボル表のアドレス
LDW         C'_1 :: m(0,%r31),%r20      ;%r1 ← &(C'_1 :: m)
COMBF,=,n   %r1,%r20,NEXT_CANDIDATE     ;if (%r1 != %r20) goto NEXT_CANDIDATE
```

図 5.1: クラスおよびメソッドチェック変換のコード

限定的 I-call if 変換では，さらに，集合 S_n に呼出し対象のインスタンス `obj` の候補クラス $\{C_1, C_2, \cdots, C_k\}$ とそれらの全親クラスを追加している．これはクラスファイルの更新に対応するためである．この追加を非限定的 I-call if 変換で実施しない理由は，非限定的 I-call if 変換を適用したコードでは，一部の候補クラスのクラスファイルを更新した場合でも，末尾にある間接呼出し経由で正しいメソッドを呼び出せるからである．たとえばレシーバ `obj` のクラスが C_1 であり，C_1 のクラスファイルが静的コンパイル後に更新された状況について考える．この状況で表 5.1 の非限定的クラスチェック変換のコードを実行すると，$C1$ のクラスファイルを更新した結果，データ構造 D_{C_1} は無効になっているので，2 行目の式 `klass == &`D_{C_1} の計算結果は偽となる．したがって 3 行目の直接呼出しを実行することはなく，最終的には末尾の間接呼出しでメソッド呼出しを実施する．間接呼出しは呼出し対象のメソッド m`()` を宣言するクラス C_0 とその全親クラスを変更しない限り正常に動作する．一方，同じ状況で表 5.1 の限定的クラスチェック変換のコードを実行すると，末尾の直接呼出し C_k`::`m`()` を実行して問題をおこす．

5.2 4 種類の I-call if 変換の使い分ける方法

本節では 4 種類の I-call if 変換を使い分ける方法について，個々の I-call if 変換が実行速度に与える影響を評価した結果から考察する．まず，クラスチェック変換とメソッドチェック変換を比較し，次に限定的 I-call if 変換と非限定的 I-call if 変換を比較する．

5.2.1 クラスチェック vs. メソッドチェック

ここではまず，クラスチェック変換とメソッドチェック変換が実行速度に与える影響の定性的な違いについて述べる．次にベンチマークによる定量的な比較結果を示し，比較結果について考察する．最後に，考察の結果から，クラスチェック変換とメソッドチェック変換の使い分け方を提案する．

実行速度に与える影響の定性的な違い

メソッドの候補数 l とクラスの候補数 k については，一般に $l \leq k$ が成り立つ．すなわち，メソッドチェック変換の方がクラスチェック変換より分岐回数が少なくなりうる．なぜなら，メソッドチェック変換では，メソッドの定義を共有する複数のクラスを1回の比較で特定できるためである．

分岐回数が少なくなりうるならば，全ての仮想メソッド呼出しをメソッドチェック変換で最適化すれば良いかというと，必ずしもそうではない [Detlefs99]．なぜなら次の例に示すように，メソッドチェック変換の方が遅くなるケースがあるからである．

- レシーバのクラスがほぼ唯一な仮想メソッド呼出しについて考える．この場合，クラスチェック変換を適用してもメソッドチェック変換を適用しても，実行時には最初の分岐で比較結果が真になり，直接呼出しする経路を辿るケースがほとんどになる．最初の分岐に至るまでの機械コードを図 5.1 に示す．図 5.1 の機械コードを見ると，メソッドチェック変換の方がロード命令を 2 回多く実行するため，遅くなることが判る．

Java2C トランスレータでは，図 5.1 に示した機械コードの差異に加え，最適時に追加する仮定クラスの違いも実行速度に影響する．5.1 節で述べたように，メソッドチェック変換の方が追加する仮定クラスが少なく，リンクの遅延から受ける影響が小さい．

ベンチマークによる比較

図 5.2 に，Java2C トランスレータで次に示す 3 つの方針によって仮想メソッド呼び出しを最適化した場合の実行速度の比較を示す．

1. クラスチェック変換のみ
2. クラスチェック変換とメソッドチェック変換の使分け
3. メソッドチェック変換のみ

クラスチェック変換とメソッドチェック変換の使分けについては，メソッドとクラスの候補数が同一の場合にはクラスチェック変換を，異なる場合にはメソッドチェック変換を用いた．if 文におけるクラスやメソッドの比較順序および比較候補数は，比較順序の違いによる速度差が生じることを防ぐために同一とした．たとえばクラスチェック変換で C_A，C_B の順で比較する場合，メソッドチェック変換では $C_A :: m$，$C_B :: m$ の順で比較した．比較順序は候補メソッドを定義するクラスを読み込んだ順序の通りとした．I-call if 変換は候補メソッド数を 3 以下に限定できる仮想メソッド呼出しに適用した．クラスチェック変換では候補メソッドに対応する候補クラスから 1 つを選んで比較対象とした．候補クラスの選択基準は，メソッドを定義するクラスが抽象クラスでなければ定義するクラスとし，抽

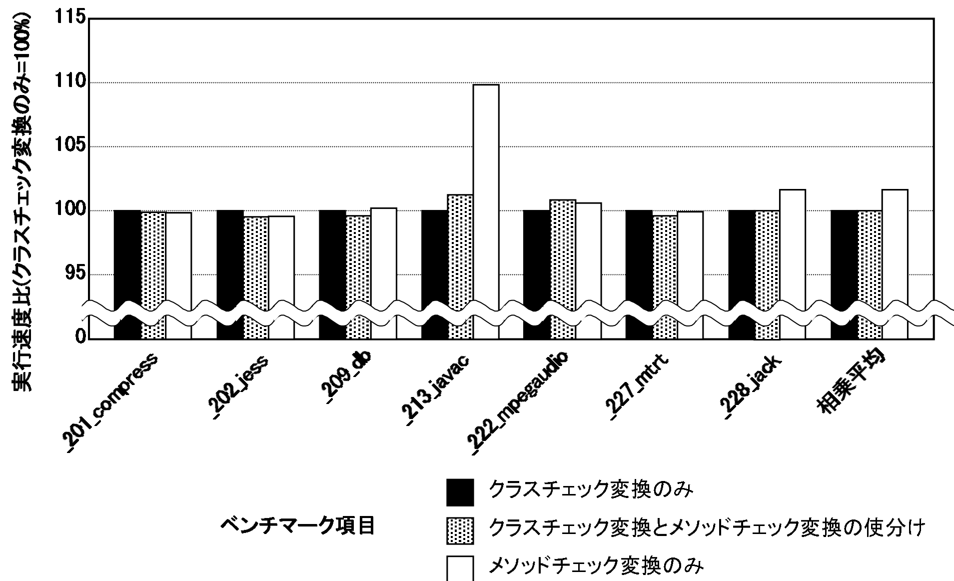

図 5.2: 実行速度によるクラスおよびメソッドチェック変換の比較

象クラスならば候補クラスのうち一番先に読み込んだものとした．限定的 I-call if 変換と非限定的 I-call if 変換の使分けについては，可能な場合には全て限定的 I-call if 変換を施した．

図 5.2 の縦軸はクラスチェック変換のみで最適化したコードの実行速度を 100 とした場合の相対的な実行速度を表し，横軸は SPECjvm98 を構成するベンチマーク項目を表す．図 5.2 から，_213_javac と _228_jack の 2 項目で実行速度に差がでていることがわかる．まず，これらの 2 項目で差がでた原因を考察し，次に他の項目で差が出なかった理由を考察する．

_213_javac と _228_jack で差がでた理由　_213_javac と _228_jack では共に，メソッドチェック変換のみで最適化すると実行速度が最も速くなる．最も遅いクラスチェック変換のみで最適化した場合と比較して，_213_javac では 9.8%，_228_jack では 1.6%高速になっている．メソッドチェック変換のみによる最適化では，クラスチェック変換とメソッドチェック変換を使い分ける最適化よりロード命令を多用する．それにもかかわらずメソッドチェック変換のみの方が実行速度が速くなる原因は，追加する仮定クラスの違いにあると考える．すなわち，非限定的クラスチェック変換と非限定的メソッドチェック変換では，非限定的メソッドチェック変換の方が追加する仮定クラス数が少なく，より早い時期に静的コンパイル済みコードをリンクしてプログラムの実行を高速化できる．

リンク時期の違いを除去して静的コンパイル済みコードの実行速度を比較するため，全メソッドについて静的コンパイル済みコードをあらかじめリンクしてから _213_javac と _228_jack を実行した結果を表 5.2 に示す．図 5.2 の実行速度はベンチマークを 1 回実行した場合の結果であり，リンク時期の影響を受けるが，4 章で述べたようにベンチマー

表 5.2: 静的コンパイル済みコードのみによる実行時間

ベンチマーク項目	最適化方針	実行回次	
		1 回目	2 回目以降
_213_javac	C	334.983	292.167
	H	330.973	292.443
	M	304.990	292.664
_228_jack	C	317.040	309.858
	H	316.932	310.213
	M	312.038	309.875

※数値は実行時間，単位は秒
※最適化方針の略号 C，H，M は次の最適化方針を表す.
C:クラスチェック変換のみ
H:クラスチェック変換とメソッドチェック変換の使分け
M:メソッドチェック変換のみ

表 5.3: 仮想メソッド呼出しにおける誤分岐回数の分布

ベンチマーク項目名	最適化方針	実行時間（秒）	誤分岐回数			
			0	1	2 以上	合計
_201_compress	C	247.250	1,620(70.6%)	571(24.9%)	102(4.4%)	2,293
	H	247.730	1,730(75.4%)	461(20.1%)	102(4.4%)	2,293
	M	247.638	1,730(75.4%)	461(20.1%)	102(4.4%)	2,293
_202_jess	C	392.862	20,123,738(74.8%)	6,771,720(25.2%)	294(0.0%)	26,895,752
	H	395.106	26,892,016(100.0%)	3,442(0.0%)	294(0.0%)	26,895,752
	M	394.909	26,892,294(100.0%)	3,452(0.0%)	294(0.0%)	26,896,040
_209_db	C	734.172	14,946,416(100.0%)	3,786(0.0%)	594(0.0%)	14,950,796
	H	737.286	14,947,417(100.0%)	2,785(0.0%)	594(0.0%)	14,950,796
	M	732.726	14,947,418(100.0%)	2,786(0.0%)	594(0.0%)	14,950,798
_213_javac	C	334.983	14,834,421(66.5%)	6,211,718(27.8%)	1,268,556(5.7%)	22,314,695
	H	330.973	17,797,120(77.2%)	5,223,730(22.7%)	39,553(0.2%)	23,060,403
	M	304.990	21,225,152(79.3%)	5,486,978(20.5%)	42,511(0.2%)	26,754,641
_222_mpegaudio	C	277.168	440,265(12.4%)	3,101,643(87.0%)	21,431(0.6%)	3,563,339
	H	274.814	3,493,138(98.0%)	70,196(2.0%)	16(0.0%)	3,563,350
	M	275.558	3,493,139(98.0%)	70,196(2.0%)	17(0.0%)	3,563,352
_227_mtrt	C	402.205	168,541,406(97.5%)	2,875,063(1.7%)	1,390,708(0.8%)	172,807,177
	H	404.065	171,642,879(99.3%)	1,164,040(0.7%)	258(0.0%)	172,807,177
	M	402.533	171,642,886(99.3%)	1,164,042(0.7%)	258(0.0%)	172,807,186
_228_jack	C	317.040	10,389,602(94.0%)	653,960(5.9%)	8,840(0.1%)	11,052,402
	H	316.932	10,548,147(95.4%)	495,415(4.5%)	8,840(0.1%)	11,052,402
	M	312.038	10,665,710(94.8%)	572,026(5.1%)	8,840(0.1%)	11,246,576

※最適化方針の略号 C，H，M の意味は表 5.2 と共通.

クを複数回連続して実行すると，1 回目で必要なクラスを全て初期化，リンクしてしまうため，2 回目以降は静的コンパイルコードのみでベンチマークを実行できる.

表 5.2 から，2 回目以降の実行，つまり静的コンパイル済みコードのみで実行した場合の速度はほぼ互角といえる．クラスチェック変換を使う方が，ロード命令の実行回数が少ない分だけ速くなる傾向は特に観測できなかった．これらのことから，図 5.2 の実行速度の差は静的コンパイル済みコードのリンク時期の違いから生じたと考える.

_213_javac と _228_jack 以外で差がでない理由　_213_javac と _228_jack 以外の項目で実行速度に差が出ない原因を検討するための資料として，表 5.3 に分岐を必要とする仮想メソッド呼出しの実行回数を，実行時に誤分岐した回数ごとに分類した結果を示す．ここ

で分岐を必要とする仮想メソッド呼出しとは，I-call if 変換により 1 段以上の if 文を含む呼出し文に変換したものを示す．final メソッド呼出しなど分岐を必要としないものについては，クラスチェック変換とメソッドチェック変換で変換結果が同一なので検討対象から除外する．また，誤分岐回数とは，I-call if 変換後の仮想メソッド呼出しのコードを実行する過程で通過する if 文において，呼出し対象のオブジェクトのクラスあるいはメソッドを比較した結果が偽となる回数を表す．

表 5.3 について，分岐を必要とする仮想メソッド呼出しの実行回数が最も多く，したがって I-call if 変換の影響を最も受けやすい **_227_mtrt** に注目して考える．**_227_mtrt** では，各最適化方針間で誤分岐の比率に大きな差がない．また，実行回数の合計欄の値がほぼ同一なことから，静的コンパイル済みコードのリンク時期にも差がないといえる．これらが **_227_mtrt** で実行速度に差が生じないことの原因だと考える．なお，静的コンパイル済みコードのリンク時期が同一ならば，クラスチェック変換の方がロード命令の実行回数が少ない分だけ実行速度が速くなるはずだが，その傾向は観測できなかった．

誤分岐が実行速度に与える影響について，**_227_mtrt** についておこなった実験から考える．表 5.3 から，**_227_mtrt** をクラスチェック変換のみで最適化すると，1 度も誤分岐しない確率が 97.5%になることがわかる．実験のため，クラスチェック変換における比較候補クラスの選択アルゴリズムを変更し，これを 66.4% として差分の 31.1%（67,633,955 回）を 1 回誤分岐した後に間接呼出しで実行するようにしたところ，実行時間が 9.02 秒余計にかかった．表 5.3 から，**_202_jess** をクラスチェック変換のみで最適化すると，1 回誤分岐する仮想メソッド呼出しが増えることがわかるが，増加数は 6,768,300 回程度であり，実行時間への影響は $9.02 \times \frac{6,768,300}{67,633,955} \simeq 0.90$ 秒程度と見積もることができる．0.90 秒は **_202_jess** の全実行所要時間の 0.23%と小さな差に過ぎない．また，表 5.3 から静的コンパイル済みコードのリンク時期にも差がないことがわかり，これらの結果として **_202_jess** では各最適化方針間で実行速度に差が出なかったと考える．**_201_compress** など残りの項目については，分岐を必要とする仮想メソッド呼出しの実行回数自体が少ないために，各最適化方針間で実行速度に差が出なかったと考える．

クラスチェック変換とメソッドチェック変換の使い分け方

結論として，クラスチェック変換とメソッドチェック変換の比較では，メソッドチェック変換を用いる方が静的コンパイル済みコードのリンク時期を早めることでプログラムの実行速度を高速化でき，それ以外の点では大きな差が出ないといえる．

5.2.2 限定的 vs. 非限定的

限定的 I-call if 変換と非限定的 I-call if 変換に関しては，コードの実行速度については，比較段数が少ない限定的 I-call if 変換が勝る．反面，リンクの遅延では，追加する仮想ク

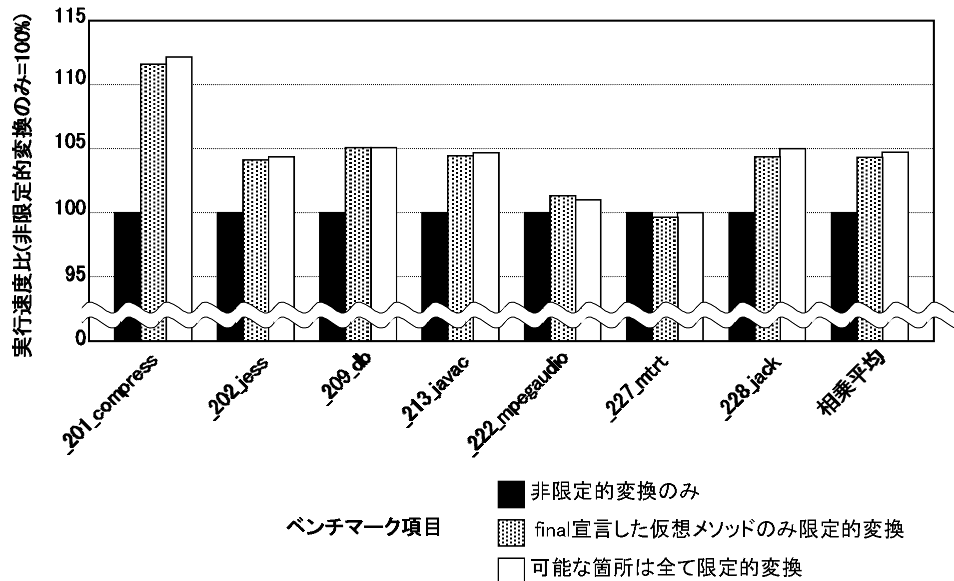

図 5.3: 限定的I-call if 変換と非限定的I-call if 変換の比較

ラス数が少ない非限定的I-call if変換の方が勝る．これらの要素が実行速度に与える影響を評価するため，次に示す3つの方針で最適化した場合の実行速度を測定した．

1. 非限定的I-call if 変換のみを適用する．
2. final 宣言した仮想メソッドの呼出しについてのみ限定的I-call if 変換を適用する．
3. （クラスフロー解析などによって）限定的I-call if 変換を適用可能と判った個所には全て限定的I-call if 変換を適用する．

最適化方針3は限定的I-call if 変換を最も多く適用するため，コードの実行速度は速くなるが追加する仮定クラスも多くなる．なお，メソッドチェック変換とクラスチェック変換の使分けについては，全ての最適化方針でメソッドチェック変換のみを適用した．

測定結果を図5.3に示す．図5.3から，概ねfinal 宣言した仮想メソッドには限定的I-call if 変換を適用する方が実行速度が速くなる傾向があることがわかる．final 宣言した仮想メソッドに限定的I-call if 変換を適用すると，全ての仮想メソッド呼出しに非限定的I-call if変換を適用する場合に比べ，`_201_compress` で12%，他に4つの項目で4～5% 実行が速くなる．final 宣言した仮想メソッドに限定的I-call if変換を適用すると実行速度を比較的大きく改善できる理由を次に示す．

- final 宣言した仮想メソッドでは，限定的I-call if変換を施すと単なる直接呼出しになり，if 文が1段もなくなる．このため，たとえば非限定的I-call if変換で3段のif文を限定化して2段にする場合より，コードの実行速度が大きく改善する．
- final 宣言した仮想メソッドは呼出し個所数が多く，仮想メソッド呼出箇所数全体の38～68% を占める（表5.4）．

表 5.4: final 宣言した仮想メソッドの呼出し箇所数

ベンチマーク項目名	総呼出し箇所数	final メソッド呼出し箇所数
_201_compress	1,157	711(61%)
_202_jess	3,317	2,128(64%)
_209_db	1,341	780(58%)
_213_javac	5,620	3,203(57%)
_222_mpegaudio	1,615	1,106(68%)
_227_mtrt	2,129	806(38%)
_228_jack	2,268	1,242(55%)

final 宣言した仮想メソッド呼出しにのみ限定的 I-call if 変換を適用する場合と，可能な限り限定的 I-call if 変換を適用する場合の実行速度差は微妙だが，わずかに後者が勝る．したがって，限定的 I-call if 変換と非限定的 I-call if 変換については，可能な限り限定的 I-call if 変換を施す方が実行速度が向上し，リンクの遅延がもたらす影響は小さいといえる．

5.3 メソッドチェック変換と C コンパイラの問題

本節では C コンパイラの実現によってはメソッドチェック変換が効率的に動作しない場合があることを指摘する．

SPECjvm98 による評価結果から Java2C トランスレータではメソッドチェック変換と限定的 I-call if 変換を積極的に適用すると実行速度が速くなる傾向があることが判ったが，C コンパイラの実現によってはメソッドチェック変換を使うと実行速度が遅くなる場合がある．

本論文で評価に利用した C コンパイラは I-call if 変換を適用した結果として得られる表 5.1 の C ソースコードを図 5.1 の機械コードに変換する．しかし，C コンパイラには色々な製品があり，同じ OS（HI-UX/WE2）向けの C コンパイラでも，表 5.1 のメソッドチェック変換の C ソースコードを図 5.4 のソースコードに変換するものもある．なお，このコンパイラは，クラスチェック変換の C ソースコードは図 5.1 と同一のコードに変換する．図 5.4 のコードは，図 5.1 のコードが 5 命令で実行する処理に最低 12 命令を必要とし，明らかに実行速度が劣る．したがって，図 5.4 のコードを出力する C コンパイラを利用する場合，メソッドチェック変換を使うと実行速度が遅くなる．図 5.4 と図 5.1 のコードが異なる原因は，関数アドレスの取得方法の違いにある．図 5.4 は古い OS における共有ライブラリの実現に対応するコードだが，共有ライブラリの実現を改善した結果，図 5.1 のコードで効率的に関数アドレスを取得可能になった．共有ライブラリの実現が洗練されている OS とコンパイラでは多くの場合，図 5.1 に準じたコードを得られ，メソッドチェック変換が有効に働く．しかし，古い OS やコンパイラを使う場合には注意が必要である．

Java2C トランスレータは可搬性の高い Java 向け静的コンパイラを実現する手段の 1 つ

```
      LDWS        4(0,%r26),%r25            ;%r25 ← obj->class.dispatch_table
      LDWS        -52(0,%r25),%r26          ;%r26 ← dispatch_table[O_m]
      LDIL        LP'C'_1 :: m,%r24         ;%r24 ← &(C'_1 :: m) の上位 bit
      LDO         RP'C'_1 :: m(%r24),%r31   ;%r31 ← &(C'_1 :: m)
      EXTRU,=     %r31,31,1,%r23            ;if (!(%r23 ← %r31 & 0x80000000)) 次命令を無視
      LDW         -4(0,%r27),%r23           ;（%r31 が関数シンボルなら）%r23 ← オフセット
      COPY        %r26,%r21                 ;%r21 ← %r26
      ADD         %r31,%r23,%r23            ;%r23 ← %r31 + %r23
      BB,>=,N     %r26,30,L1                ;if (%r26 が共有ライブラリ上にない) goto L1
      BL          $$sh_func_adrs,%r31       ; 関数呼出し： %r29 ← 補正済みアドレス
      NOP
      COPY        %r29,%r21                 ;%r21 ← %r29
L1
      COPY        %r23,%r29                 ;%r29 ← %r23
      BB,>=,N     %r23,30,L2                ;if (%r23 が共有ライブラリ上にない) goto L2
      COPY        %r23,%r26                 ; 引数のセット： %r26 ← %r23
      BL          $$sh_func_adrs,%r31       ; 関数呼出し： %r29 ← 補正済みアドレス
      NOP
L2
      COMBF,=,N   %r21,%r29,NEXT_CANDIDATE  ;if (%r21 != %r29) goto NEXT_CANDIDATE
```

図 5.4: メソッドチェック変換の非効率的な実現

だが，移植にあたっては，この問題が示すように，組み合わせる C コンパイラの性質を考慮して，クラスチェック変換とメソッドチェック変換のどちらを適用するか決めると，より実行速度を改善できる．

第6章　例外処理の実現

本章では，Java2C トランスレータで利用可能な例外処理の実現方法である setjmp 法と 2 返戻値法について，どちらが優れた実現方法か SPECjvm98 の実行結果から定量的に比較する．また，それぞれの実現方法向けの最適化を提案，評価する．評価の結果，2 返戻値法で例外処理を実現した方が setjmp 法で実現するよりもプログラムの実行を平均 1.5%高速にできることが判った．本章ではまず，6.1 節で setjmp 法による例外処理の実現について詳述し，setjmp 法向け最適化を提案する．次に 6.2 節で 2 返戻値法による例外処理の実現について詳述し，2 返戻値法向け最適化を提案する．最後に 6.3 節で評価をおこなう．すなわち，提案した最適化の効果を検証し，setjmp 法と 2 返戻値法を比較する．

6.1　setjmp 法

本節ではまず，setjmp 法で Java の例外処理を実現する方法を示し，次に setjmp 法向けの最適化技法を提案する．

6.1.1　setjmp 法による例外処理の実現

setjmp 法を使って Java の例外処理を実現するには，次の処理をおこなう．個々の処理について順次述べる．

- `try`，`catch`，`throw` の `setjmp()`，`longjmp()` への置換
- 自動変数の volatile 化
- モニタ解放コード の挿入
- 非同期例外検出コード の挿入

`try`，`catch`，`throw` の `setjmp()`，`longjmp()` への置換

`throw` を `longjmp()` に変換し，`try` を次のコードに変換する．

- `setjmp()` を実行し，`longjmp()` からの復帰後に `catch` 節にジャンプするコード

```
struct try_buf{
  jmp_buf env;
  int monitor_stack_pointer;
  struct try_buf *next;
};

#define enterTry(ee, buf) ( \
  (buf)->monitor_stack_pointer = \
    (ee)->monitor_stack_pointer, \
  (buf)->next = (ee)->try_clause, \
  (ee)->try_clause = (buf), \
  (JObject*)(setjmp(buf->env)))

#define exitTry(ee, buf) \
  (ee)->try_clause = (buf)->next

void boo(ExecEnv *ee, JObject *this){
  JObject *exception;
  struct try_buf buf;
  /* try 前処理 */
  exception = enterTry(ee, &buf);
  if ((int)exception == 0){
     /* try ブロック */
    foo(ee, this);
    /* try ブロック脱出 */
    exitTry(ee, &buf);
  }else{
    /* longjmp で飛び越した間の同期解除 */
    ExitJumpedOverMonitors(ee, &buf);
    /* try ブロック脱出 */
    exitTry(ee, &buf);
    /* catch */
    if (IsInstanceOf(exception, Exception)){
      /* ハンドラ */
    }else{
      /* 再び投げる */
      longjmp(ee->try_clause, exception);
    }
  }
}

#define pushMonitor(ee, obj) { \
  int sp = ee->monitor_stack_pointer; \
  if (sp >= ee->mmonitor_stack_size){ \
    /* スタック拡張処理 */ \
  } \
  (ee)->monitor_stack[sp] = obj; \
  (ee)->monitor_stack_pointer = sp+1; \
}

#define popMonitor(ee) \
  (ee)->monitor_stack_pointer--;

void foo(ExecEnv *ee, JObject *this){
  JObject *exception;
  MonitorEnter(ee, this);
  pushMonitor(ee, this);
  while(true){
    woo(ee, this);
    /* 非同期例外の検出 */
    if (*((volatile int *)&ee->async_exception)){
      exception = ee->async_exception;
      ee->async_exception = NULL;
      longjmp(ee->try_clause, exception);
    }
  }
  MonitorExit(ee, this);
  popMonitor(ee);
}

void woo(ExecEnv *ee, JObject *this){
  JObject *exception =
    NewObject(ee, Exception);
  ExceptionConstructor(ee, exception);
  longjmp(ee->try_clause, exception);
}

void ExitJumpedOverMonitors(
  ExecEnv *ee, struct try_buf *buf) {
  int top = ee->monitor_stack_pointer;
  int bot = buf->monitor_stack_pointer;
  while(top-- > bot)
    MonitorExit(ee->monitor_stack[top]);
  ee->monitor_stack_pointer = bot;
}
```

図 6.1: setjmp 法による変換結果

この変換の具体例として，図 2.5 の Java ソースコードを setjmp 法で C ソースコードに変換した結果を図 6.1 に示す．図 6.1 の関数 `boo()` では，`try` ブロックの入口に敷設してあるマクロ `enterTry()` の中で最上位の `try` ブロック，つまり最後に入った未脱出の try ブロックを更新した上で `setjmp()` を実施し，関数 `woo()` のコード中で `throw` を最上位の `try` ブロックの `catch` 節への `longjmp()` で実現している．最上位の `try` ブロックは，`try` ブロックからの出口に敷設してあるマクロ `exitTry()` でも更新する．

自動変数の volatile 化

次の条件を満たす自動変数を `volatile` 宣言する．

1. `try` ブロック内に定義点 δ がある．
2. 定義点 δ に対する使用点が `catch` 節以降にある．

この処理が必要になる理由は，ANSI C が `setjmp()` の実施時点から `longjmp()` を実施するまでの間に変更した非 `volatile` 自動変数について，`longjmp()` からの復帰後に正しい内容を保持することを保証しないからである．これは，`try` ブロック内，つまり `setjmp()` を実施した後で定義した値を非 `volatile` 自動変数に格納した場合，同変数を `catch` 節以降，つまり `longjmp()` 後に参照すると，変数に不正な値が入っていて問題が発生する可能性があることを意味する．この問題は自動変数を `volatile` 宣言すれば解決できるが，`volatile` 宣言は最適化を抑止して実行速度に悪影響を与えることもある．

モニタ解放コードの挿入

Java ではモニタを同期メソッドにおいて利用する．同期メソッドとは排他制御の機能を有するメソッドで，メソッドの出入口でモニタの確保と解放を実行する点が他のメソッドとは異なる．同期メソッドを作成するには，メソッドの宣言に `synchronized` というキーワードを付加する．図 2.5 ではメソッド `foo()` を `synchornized` と宣言しており，それに対応する図 6.1 の C 言語のコードには，その出入り口にモニタを確保，解放する関数呼出し `MonitorEnter()`，`MonitorExit()` がある．

同期メソッドについて setjmp 法で注意すべきことは，`longjmp()` で `catch` 節まで戻る際に，モニタを解放せずに同期メソッドを飛び抜けてしまうことである．たとえば図 6.1 で関数 `boo()` から関数 `foo()` を経て呼び出した関数 `woo()` で例外を投げ，`longjmp()` で関数 `boo()` 中の `catch` 節まで大域ジャンプするとき，関数 `foo()` のモニタを解放せずに（`MonitorExit()` を実行せずに）同期メソッドを抜けてしまう．この問題を解決するために，次の処理をおこなう．

表 6.1: 構造体型 ExecEnv がもつモニタスタックに関連したメンバ

メンバ名	役割
monitor_stack	モニタスタックへの参照
monitor_stack_pointer	モニタスタックのスタックポインタ
monitor_stack_size	モニタスタックのサイズ

1. スレッドが確保したモニタを登録するスタックを用意する．このスタックを**モニタスタック**と呼ぶことにする．

   図 6.1 では，スレッド固有の資源を収める **ExecEnv** 型の構造体を変数 **ee** が指示しており，**ExecEnv** 型が表 6.1 のメンバをもつと想定している．
2. 同期メソッドの出入口で確保，解放したモニタを，モニタスタックに登録，消去する動作を挿入する．

   図 6.1 ではマクロ **pushMonitor()**，**popMonitor()** がモニタスタックへの登録，消去をおこなう．これらのマクロは関数 **foo()** 中にあるモニタを確保，解放する関数呼出し **MonitorEnter()**，**MonitorExit()** の直後に挿入してある．
3. **try** ブロックへの入口に，**try** ブロックに入る時点におけるモニタスタックのスタックポインタを記録するコードを挿入する．

   図 6.1 では **try** ブロックの入口に配置するマクロ **enterTry()** の中にモニタスタックのスタックポインタを記録する処理がある．
4. **throw** を表す **longjmp()** を実行後に，飛び抜けた同期メソッドのモニタを解放するコードを挿入する．飛び抜けた同期メソッドのモニタとは，モニタスタックに記録してあるモニタのうち，**try** ブロックの入口で記録したスタックポインタより上位に位置するものである．

   飛び抜けた同期メソッドのモニタを解放するコードの挿入先は，**try** を表す **setjmp()** に後続する分岐で **setjmp()** の返戻値から例外の発生を検知した後，**catch** 節より前である．

   図 6.1 では，関数 **boo()** の中で **try** ブロックの入口にあるマクロ **enterTry()** に後続する if 文で例外発生の有無を検知し，例外が発生した場合に **else** 節にジャンプして，**else** 節の冒頭でモニタを解放する関数 **ExitJumpedOverMonitors()** を呼び出す．

非同期例外検出コードの挿入

非同期例外とは，別のスレッドからの指示によって投げる例外を意味する．Java では，**java.lang.Thread.stop()** というメソッドを使い，あるスレッドが別のスレッドに非同期例外を投げるように命じることができる．非同期例外を投げるよう命じられたスレッドは，どんな処理を実施中であったかによらず，一定期間以内に非同期例外を投げ，例外処

理をおこなう必要がある．setjmp 法では非同期例外を投げよとの指示を検出し，非同期例外を投げることを可能にするために，次の処理をおこなう．

- 非同期例外が発生したか否かを示すフラグを用意し，このフラグをポーリングして非同期例外を検出し，投げるコードをループ内の必ず実行するパスに挿入する．

ポーリングをおこなうコードの挿入先をループ内の必ず実行するパスにする理由は，非同期例外を一定期間以内に検出可能にするためである．ポーリングを含まない無限ループが存在すると，永遠に非同期例外を検出できなくなる．なお，ループを生成する最適化である末尾再帰の除去は，C コンパイラでなく Java2C トランスレータで実施する必要がある．なぜなら C コンパイラは生成したループに非同期例外が発生したか否かポーリングするコードを挿入しないからである．

図 6.1 のコードでは，変数 `ee` が指示する `ExecEnv` 型の構造体内に非同期に発生した例外を収めるフィールド `async_exception` を用意し，このフィールドをポーリングして非同期例外が発生していたら投げるコードを挿入している．ポーリングのコードは関数 `foo()` の `while` ループ内にある．非同期例外を発生する場合には，発生先のスレッドの `ee` の `async_exception` フィールドに例外へのポインタを書き込む．こうすると，挿入したコードが `async_exception` フィールドをポーリングし，書き込んでおいた非同期例外を発見して投げる．

図 6.1 のコードではポーリングのためのメモリ参照を `volatile` と指定している．これは C コンパイラがポーリングのためのメモリ参照をループ不変とみなしてループ外に排出することを防ぐための処置だが，ループ内に `volatile` 参照をおくと，コードの移動に制限がかかり，ループ不変式移動などの最適化を適用できなくなる問題が発生する．

6.1.2 setjmp 法向けの最適化

setjmp 法では `try` ブロックへの出入口でマクロ `enterTry()`，`exitTry()` を実行するなどの理由から例外処理を実現するためにオーバヘッドが発生する．このような setjmp 法固有のオーバヘッドは，Java2C トランスレータで最適化を施すことにより，ある程度軽減できる．Java2C トランスレータが実施しうる次の 4 種類の最適化を提案する．

1. `volatile` 宣言の最小化
2. 冗長な `enterTry()`, `exitTry()` の除去
3. `_setjmp()` の利用
4. 冗長な例外処理方式の変換の除去

volatile 宣言の最小化

setjmp 法では一部の自動変数を volatile 宣言する必要がある[1]．volatile 宣言する自動変数の選定方法として，簡易的には try ブロックをもつメソッド内の全自動変数を volatile 宣言する方法があるが，これではオーバヘッドが大きい．volatile 宣言によるオーバヘッドを抑止するには，Java2C トランスレータに volatile 宣言する自動変数を必要最小限にする最適化機能を実現すればよい．具体的には，try ブロック内に定義点があり，なおかつ longjmp() によって try ブロックから抜けたあとに，その定義点に対応する使用点がある自動変数に限り volatile 宣言する．なぜなら，longjmp() が値を破壊しうる自動変数は try ブロック内で定義したもののみであり，それらのうち値の保護を必要とするのは，longjmp() 後にその値への参照が生じうる場合のみだからである．

冗長な enterTry(), exitTry() の除去

setjmp 法で例外を投げる際に longjmp() が必要になるのは，関数内に存在しないハンドラへ大域的にジャンプする場合である．ハンドラが関数内に存在する場合には，オーバヘッドの大きい longjmp() の代りに単なる goto 文を使ってハンドラへのジャンプを実現できる．

try ブロックによっては局所的に発生する例外のみを捕捉するものもある．すなわち，try ブロック内から例外を投げる動作を全て goto 文によって実現できる場合である．そういった try ブロックについては，図 6.1 の setjmp 法のコードで try ブロックの出入口に敷設しているマクロ enterTry()，exitTry() を省略できる．なぜなら，これらのマクロは longjmp() によって大域的にハンドラに復帰するための準備及び後始末を目的とするもので，longjmp() が発生しないならば不要だからである．なお，longjmp() が発生しない try ブロックには，出入口のマクロを省略できることの他に，自動変数を volatile 宣言する必要が生じない利点もある．

_setjmp() の利用

setjmp()，longjmp() は実現によってはシグナルマスクの退避と復帰をおこなうが，この動作は Java の例外処理の実現には不要である．評価環境の HI_UX/WE2 を含め，多くの OS はシグナルマスクの退避と復帰を省略した setjmp()，longjmp() に相当するライブラリ関数を _setjmp()，_longjmp() といった名称で提供し，これらを setjmp()，longjmp() の代用とすることで高速化を図ることができる．

なお，setjmp() の実現（意味）は OS や C コンパイラに大きく依存する．C コンパイラの中には setjmp() を含む関数について最適化を抑止するものもあるが，そのような C コンパイラとの組合せでは setjmp 法のオーバヘッドはより大きくなる．なお，評価に用

[1] あるいは volatile 宣言以外で，longjmp() からの復帰後に自動変数の値を修復する措置が必要になる．

```
 1:  boolean invokeWithConversion(···){
        ⋮
 2:
 3:    exception = enterTry(ee, &buf);
 4:    if ((int)exception == 0){
 5:      setjmp 法で例外を処理するコードを呼ぶ;
 6:    }else{
 7:      ExitJumpedOverMonitors(ee, &buf);
 8:      ee->exception = exception;
 9:    }
10:    exitTry(ee);
        ⋮
11:
12:  }
```

図 6.2: 例外処理方式の変換

いた HI-UX/WE2 向け最適化 C コンパイラでは，`setjmp()` を含む関数について最適化を抑止することはない．

冗長な例外処理方式の変換の除去

実験環境に用いた JeanPaul はインタプリタと Java2C トランスレータで生成する静的コンパイル済みコードを用いてプログラムを実行するが，インタプリタから setjmp 法で例外を処理する静的コンパイル済みコードを呼び出す場合と，逆に setjmp 法で例外を処理する静的コンパイル済みコードからインタプリタを呼び出す場合には，例外処理方式の変換が必要になる．なぜなら，JeanPaul のインタプリタにおける例外処理方式が setjmp 法とは異なるためである．JeanPaul のインタプリタは，次の手順で例外処理を実現する．

1. 発生した例外への参照を保持するフィールド `exception` を，変数 `ee` が指示する構造体の中に用意する．変数 `ee` が指示する構造体はスレッド固有の資源を収める．
2. 例外を投げる際には，`exception` フィールドを介して発生した例外を伝播し，表引き法でハンドラのアドレスを決定してジャンプする．

なお，この `exception` フィールドを介した例外の伝播方法は次節で述べる 2 返戻値法における伝播方法と同一であり，したがって 2 返戻値法で例外を処理する静的コンパイル済みコードを呼び出す際には例外処理方式の変換は必要ない．

図 6.2 に，インタプリタから setjmp 法で例外を処理する静的コンパイル済みコードを呼び出す場合に，例外処理方式を変換するコードを示す．図 6.2 のコードは次の手順で例外処理方式を変換する．まず，`setjmp()` を含むマクロ `enterTry()` を実施し，次に setjmp 法で例外を処理する静的コンパイル済みコードを呼び出す．そして，例外が発生した場合には，例外を捕捉して変数 `ee` が参照する構造体の `exception` フィールドに収めることで例外処理方式を変換する．

図6.2の変換処理は，例外の発生の有無にかかわらず`setjmp()`を含むマクロ`enterTry()`を実行するためオーバヘッドが大きい．このオーバヘッドを抑止する手段として，例外を発生しえない関数を呼ぶ場合に例外処理方式の変換に関する処理を省略する最適化が考えられる．この最適化は，たとえばインタプリタからset関数やget関数のように例外を発生しえない小さな関数を頻繁に呼び出す場合に有用になる．

6.2 2返戻値法

本節ではまず，2返戻値法による例外処理の実現について詳述する．次に，2返戻値法向けの最適化技法を提案する．

6.2.1 2返戻値法による例外処理の実現

2返戻値法で例外処理を実現するには，コンパイル時に次の操作を実施する．

- メソッドの返戻値を2つにする．一方はメソッドがreturn文を使って返戻する通常の返戻値であり，もう一方は例外が発生したか否かを表すフラグである．本論文ではこのフラグを**例外発生フラグ**と呼ぶ．
- 全メソッド呼出しの直後に**例外発生検査**を挿入する．例外発生検査とは，例外発生フラグを参照し，例外が発生していた場合に分岐するコードを意味する．例外発生検査の分岐先は，直前のメソッド呼出しがソースコード上で`try`ブロック内にある場合には，最内の`try`ブロックに後続する`catch`節となり，`try`ブロック内にない場合には，メソッドからの出口処理となる．ここでメソッドからの出口処理とは，スタックフレームの解放などメソッド実行中に利用した資源を解放してから呼出し元に戻る処理を意味する．
- 非同期例外を検出するための例外発生検査をループ内の必ず通過するパスに挿入する．
- `throw`を，例外発生フラグを立ててから`catch`節あるいはメソッドの出口処理にジャンプするコードとして実現する．

これらの操作を適用すると，実行時には，例外を`throw`すると例外発生検査経由でハンドラまで到達することが可能になる．

2返戻値法による例外処理の実現について，図6.3を使って詳述する．図6.3のCソースコードは，図2.5のJavaソースコードを，2返戻値法で例外処理を実現するJava2Cトランスレータで変換した結果である．図6.3のコードでは，通常の返戻値を`return`文で返戻し，2つ目の返戻値（例外）を`ee`中のフィールド`exception`を介して返戻する．変数`ee`が指示するスレッド固有の資源を収める構造体の定義のうち，例外処理に関係する部分を図6.4に示す．

```
 1:  void boo(ExecEnv *ee, JObject *this){
 2:    foo(ee, this);
 3:    if (exceptionOccurred(ee)){
 4:      JObject *exception = getException(ee);
 5:      if (IsInstanceOf(exception, Exception)){
 6:        clearException(ee);
 7:        /* ハンドラ */
 8:      }
 9:    }
10:  }
11:
12:  void foo(ExecEnv *ee, JObject *this){
13:    MonitorEnter(this);
14:    while(true){
15:      woo(ee, this);
16:      /* 同期，非同期例外の検出 */
17:      if (exceptionOccurred(ee)){
18:        goto EXIT;
19:      }
20:    }
21:  EXIT:
22:    MonitorExit(this);
23:  }
24:
25:  void woo(ExecEnv *ee, JObject *this){
26:    JObject *exception = NewObject(ee, Exception);
27:    if (exceptionOccurred(ee)) return;
28:    ExceptionConstructor(ee, exception);
29:    if (exceptionKind(ee)) return;
30:    throw(ee, exception);
31:  }
```

図 6.3: 2 返戻値法による変換結果

図 6.4 の 46 行目にあるマクロ **throw** の定義から判るように，2 返戻値法においては，例外を投げる動作を，**ee** 中のフィールド **exception** に発生した例外への参照を収め，また，フィールド **exceptionKind** に例外が発生したことを表す値を収めることで実現する．図 6.3 の 30 行目，メソッド **woo()** の内部にマクロ **throw** を実行している箇所がある．メソッド **woo()** は例外を捕捉する **catch** 節を持たないので，ここでは **throw** を実施した直後にメソッドの末尾で出口処理をおこなって呼出し元に返戻する．呼出し元にあたるメソッド **foo()** では，15 行目のメソッド呼出し **woo()** から返戻した直後の 17 行目でマクロ **exceptionOccurred()** を使って例外発生検査を実施し，例外が発生している場合には，メソッド **foo()** も **catch** 節を持たないので 21 行目の出口処理にジャンプする．出口処理では，同期メソッド **foo()** の入口で確保したモニタを解放してから返戻する．このように，2 返戻値法ではメソッドから返戻する時に随時モニタを解放できるので，**setjmp** 法のように飛び越えたモニタをあとからまとめて解放する処理を必要としない．

メソッド **foo()** の呼出し元にあたるメソッド **boo()** では，2 行目のメソッド呼出し **foo()** から返戻した直後の 3 行目で例外発生検査を実施し，例外が発生している場合には 4 〜 8 行目にある **catch** 節に制御を移して例外を処理する．1 〜 10 行目にあるメソッド **boo()**

```
 1:  /*
 2:  ** スレッド固有の資源を収める構造体型の定義
 3:  */
 4:  struct execenv {
 5:    ...
 6:    /*
 7:    ** 例外関係のフィールド群の宣言
 8:    ** 同期例外を投げた際の排他制御コードを
 9:    ** 不要にするため同期例外用と非同期例外
10:    ** 用に別個のフィールドを提供する.
11:    ** exceptionKind:
12:    ** 例外発生フラグ
13:    ** exception:
14:    ** 同期例外を格納するフィールド
15:    ** async_exception:
16:    ** 非同期例外を格納するフィールド
17:    */
18:    union{
19:      struct{
20:        volatile char sync;
21:        volatile char async;
22:      } detail;
23:      volatile short summary;
24:    } exceptionKind;
25:    JHandle *exception;
26:    JHandle *async_exception;
27:    ...
28:  };
29:
30:  typedef struct execenv ExecEnv;
31:
32:  /*
33:  ** 例外発生フラグに収める値の定義
34:  ** ExecEnv の exceptionKind フィールドに
35:  ** 次のいずれかの値が入る.
36:  **/
37:  #define EXCKIND_NONE  0
38:  #define EXCKIND_THROW 1
39:
40:  /*
41:  ** 例外を投げたり, 捕捉する動作や, 例外発生検査を実現するマクロの定義
42:  */
43:  #define exceptionOccurred(ee) \
44:      ((ee)->exceptionKind.summary != 0)
45:
46:  #define throw(ee, exception) \
47:      ee->exception = exception; \
48:      ee->exceptionKind.detail.sync = EXCKIND_THROW;
```

図 6.4: スレッド固有の資源を収める構造体型の定義

のコードから判るように，2 返戻値法では `try` ブロックへの出入りにあたって何らかの処理を必要としない．これは出入口でマクロ `enterTry()`，`exitTry()` を実行する必要がある setjmp 法より 2 返戻値法が優れている点である．反面，2 返戻値法ではメソッド呼出しの直後に挿入する例外発生検査が固有のオーバヘッド発生源となる．

6.2.2　2 返戻値法向け最適化

2 返戻値法のオーバヘッドは例外発生検査から発生する．例外発生検査から発生するオーバヘッドを軽減する方法に次の 4 つがある．

1. インライン展開
2. 下方移動による冗長な例外発生検査の除去
3. 下方移動による例外発生検査の集約
4. 手続き間解析による冗長な例外発生検査の除去

1 のインライン展開は，適用するとメソッド呼出しそのものが無くなるので，直後の例外発生検査も除去できる．ただし，インライン展開にはコードサイズを増やす欠点がある．そこで本論文では 2 〜 4 の最適化技法を提案する．

下方移動による冗長な例外発生検査の除去

下方移動による冗長な例外発生検査の除去では，例外発生検査を移動することを通じて，例外発生検査が冗長か判定し，冗長であれば除去する．下方移動による冗長な例外発生検査の除去の実行手順を次に示す．

1. 個々の例外発生検査について，例外非発生時の分岐側に存在する文を，例外発生検査より前に移動できるか判定する．移動可能な条件は，文が発生する副作用が，例外発生時の分岐側で実行する変数参照やメモリ参照に影響しないことである．
2. 移動可能であれば文を例外発生検査の直前に移動し，次の文が移動可能か判定する．この手順を移動不能な文が現れるまで繰り返す．

   移動前後のコードをソースコードで表現すると，移動後に例外発生検査がソースコードの下方に動いている．そこでこの移動を下方移動と呼ぶことにする．
3. 移動が終了したら，例外発生検査が次の条件を満たすか調べ，満たしていたら冗長とみなして除去する．

   - 例外発生時の分岐先と非発生時の分岐先が同一になった場合．
   - 例外非発生時の分岐先に別の例外発生検査が存在し，2 つの例外発生検査の例外発生時の分岐先が同一である場合．たとえば，下方移動先に非同期例外の検出を目的とした例外発生検査が存在する場合など．

例外発生検査を下方移動する過程で移動する文は投機実行の対象となり，例外発生時には無駄に実行することになる．しかし，例外の発生は稀であり，文の移動が実行速度に悪影響を与える可能性は少ない．例外発生検査の下方移動には，むしろメソッド呼出しの返戻から例外発生検査までの間にコードスケジューリングの余地を与え，実行速度を改善する効果があり，冗長な例外発生検査の除去を適用できない場合でも実施する価値がある．本論文で提案する例外発生検査の下方移動は，既存の技術である分岐の下方移動と次の点で異なる．

- 2 返戻値法固有の特例を使ってより多くの例外発生検査を下方移動できる．2 返戻値法固有の特例とは，例外発生時に **catch** を経由せずメソッドから返戻する例外発生検査を，**return** 文が返戻する通常の返戻値を格納する一時変数の定義点を越えて下方移動できることである．なぜなら，**catch** を経由せずに例外が発生した状態で返戻する場合，通常の返戻値は無効になるので，どのような値をとっても構わないからである．

- 通常の分岐の下方移動では，無駄な投機実行を避けるために，プロファイルなどを使って分岐確率を調査し，分岐の移動先を，確率が高い方の分岐先にある文の下に定める．しかし，例外発生検査では非例外発生側に分岐する確率が高いと判っていることから，分岐確率の調査が不要であり，下方移動を容易に実施できる．

図 6.3 の 14 〜 20 行目にある **while** ループ中には例外発生検査が 1 箇所しかないが，これは 2 つの例外発生検査（15 行目のメソッド呼出し **woo()** が返戻する例外を検出するための例外発生検査と，非同期例外を検出するためにループ中に挿入する例外発生検査）の一方を冗長とみなして除去した結果である．なお，例外発生検査を実施するとき使うマクロ **exceptionOccurred()** では同期例外も非同期例外も検出できる．その理由を次に示す．

- 図 6.4 の 18 〜 24 行目から，例外の発生状況を示すフラグ **exceptionKind** が次の 2 つの要素からなることが判る．
    1. 同期例外の発生を示すフラグ **sync**
    2. 非同期例外の発生を示すフラグ **async**

- 図 6.4 の 43 〜 44 行目の定義にあるように，マクロ **exceptionOccurred()** は，フラグ **sync** と **async** を一括してフラグ **summary** として参照する．したがって，マクロ **exceptionOccurred()** は同期例外も非同期例外も検出できる．

下方移動による例外発生検査の集約

下方移動による例外発生検査の集約では，例外発生検査の下方移動を通じて，複数の例外発生検査を集約し，コードサイズを削減する．下方移動による例外発生検査の集約の実施手順を次に示す．

```
 1:  int alpha(bool condition){
 2:    int result;
 3:    if (condition)
 4:      ClassA.beta();
 5:    else
 6:      ClassA.gamma();
 7:    result += ClassA.delta();
 8:    return (result);
 9:  }
```

(a) Java ソースコード

```
 1:  int alpha(ExecEnv *ee, bool condition){
 2:    int result, tmp;
 3:    if (condition){
 4:      ClassA_boo(ee, this);
 5:      if (exceptionOccurred(ee))
 6:        goto EXIT;
 7:    } else {
 8:      ClassA_foo(ee, this);
 9:      if (exceptionOccurred(ee))
10:        goto EXIT;
11:    }
12:    tmp = ClassA_delta();
13:    if (exceptionOccurred(ee))
14:      goto EXIT;
15:    }
16:    result = tmp;
17:  EXIT:
18:    return (result);
19:  }
```

(b) C ソースコード (最適化なし)

```
 1:  int alpha(ExecEnv *ee, bool condition){
 2:    int result, tmp;
 3:    if (condition){
 4:      ClassA_boo(ee, this);
 5:    } else {
 6:      ClassA_foo(ee, this);
 7:    }
 8:      if (exceptionOccurred(ee))
 9:        goto EXIT;
10:    tmp = ClassA_delta();
11:    result = tmp;
12:  EXIT:
13:    return (result);
14:  }
```

(c) C ソースコード (最適化あり)

図 6.5: 下方移動による例外発生検査の集約

1. 下方移動による冗長な例外発生検査の除去と同じ手順で，個々の例外発生検査を下方移動する．
2. 例外発生検査 t の例外非発生時の分岐先を E_n，例外非発生時の分岐先を E_e と書くとき，次の条件を満たす例外発生検査のペア $(E1, E2)$ は集約できる．

$$(E1_n = E2_n) \wedge (E1_e = E2_e)$$

この条件を満たす例外発生検査のペアがあるか調べ，あったら集約する．

図 6.5(a) の Java ソースコードを C ソースコードに変換するにあたって，下方移動による例外発生検査の集約を適用しなかった結果を図 6.5(b) に，適用した結果を図 6.5(c) に示す．図 6.5(b) と図 6.5(c) を比較すると，図 6.5(b) の 5 行目と 9 行目にある例外発生検査が，図 6.5(c) では 8 行目の一箇所に集約されていることが判る．

なお，この最適化には実行速度を改善する効果は余りない．なぜなら，この最適化を適用しても実行時に例外発生検査を実行する回数は変化しないからである．

下方移動による例外発生検査の集約は，複数のコード中に存在する共通部分を一箇所に集約する最適化の一種である．C コンパイラの多くはこの最適化を提供し，例外発生検査を下方集約できるものもある．しかし，C コンパイラでは例外発生検査を下方移動にあたって，本論文で指摘した 2 返戻値法固有の特例を利用できない．

手続き間解析による冗長な例外発生検査の除去

手続き間解析による冗長な例外発生検査の除去では，個々のメソッド呼出しについて，呼出し先のメソッドが例外を投げる可能性があるか解析し，ないならばメソッド呼出しの直後に例外発生検査を挿入しない．

ただし，この最適化が役に立つ場合は少ない．なぜなら例外を投げる可能性がないメソッドは多くの場合，set メソッドや get メソッドのように小さなメソッドでインライン展開の対象となり，インライン展開が例外発生検査を除去してしまうからである．

6.3 評価

本節ではベンチマークの実行結果から，まず，setjmp 法向け最適化と 2 返戻値法向け最適化が実行速度に与える影響を評価し，次に，setjmp 法と 2 返戻値法を比較する．また，例外処理の実現にどれだけのオーバヘッドがかかるか評価する．

6.3.1 setjmp 法向け最適化の評価

setjmp 法向け最適化が実行速度に与える影響を評価した結果を図 6.6 に示す．評価は全ての最適化を適用した場合の実行速度を基準として，個々の最適化を適用しなかった場合

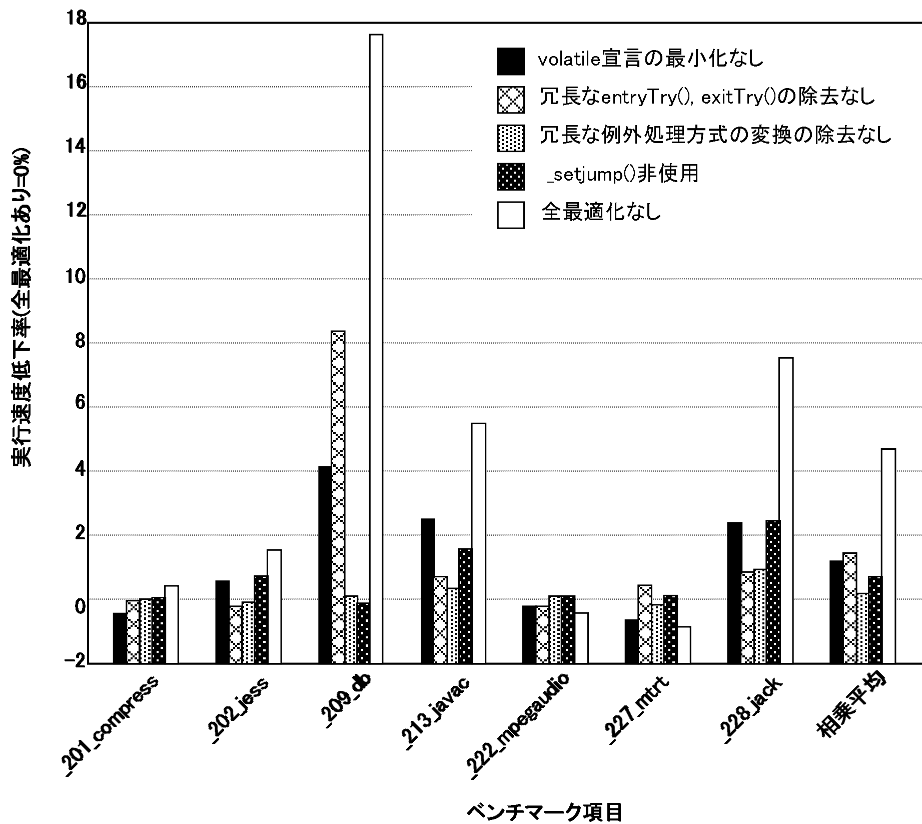

図 6.6: setjmp 法向け最適化が実行速度に与える影響

にどれだけ実行速度が低下するか測定することでおこなった．低下率が大きいほど有効な最適化であるといえる．図 6.6 の縦軸は最適化を適用しなかった場合の実行速度の低下率を表し，横軸はベンチマーク項目名を表す．個々の最適化の効果について考察する．

`volatile` 宣言の最小化

図 6.6 から，`volatile` 宣言の最小化が実行速度に相乗平均で 1.2%，項目別では **_209_db** と **_213_javac**，**_228_jack** にそれぞれ 4.1%，2.5%，2.4%の影響を与えることが判る．この値は単一の最適化から得られる値として決して小さくなく，したがって `volatile` 宣言の最小化は実現する価値のある最適化であるといえる．なお，`volatile` 宣言の最小化によって `volatile` 宣言する自動変数の数に与える影響を評価した結果を表 6.2 に示す．表 6.2，図 6.6 において，最適化なしとは `try` ブロックをもつメソッド内の全自動変数を `volatile` 宣言する場合を表す．表 6.2 から，最適化によって `volatile` 宣言する自動変数の数が最適化なしの場合と比較して $\frac{1}{10}$ 以下に減少することが判る．

表 6.2: volatile 宣言の最小化による volatile 宣言数の変化

ベンチマーク項目名	自動変数宣言総数	try ブロック総数	volatile 宣言数	
			最適化	
			なし	あり
_201_compress	5,532	97	1,012	70
_202_jess	10,100	121	1,276	86
_209_db	6,095	106	1,193	84
_213_javac	13,873	180	2,094	150
_222_mpegaudio	9,088	97	1,155	86
_227_mtrt	6,524	105	1,235	74
_228_jack	9,347	168	1,817	96

表 6.3: enterTry() の実行回数と最適化の影響

ベンチマーク項目名	実行時間 (sec)	enterTry()					throw 回数
		実行箇所				平均実行頻度	
		try ブロック入口		例外処理方式変換部			
		最適化なし	最適化あり（削減率）	最適化なし	最適化あり（削減率）		
_201_compress	253.8	1,959	1,882(3.93%)	1,644	882(46.35%)	11	0
_202_jess	401.2	1,240,894	30,599(97.53%)	707,879	705,674(0.31%)	1,835	0
_209_db	737.0	61,438,533	32,154(99.95%)	28,5355	283,097(0.79%)	428	0
_213_javac	315.7	1,609,073	952,535(40.80%)	2,628,947	2,192,371(16.61%)	9,960	21,373
_222_mpegaudio	280.6	2,146	2,136(0.47%)	18,532	17,794(3.98%)	71	0
_227_mtrt	403.0	4,909	4,772(2.79%)	349,625	189,835(45.70%)	483	0
_228_jack	312.6	7,385,349	4,371,780(40.80%)	634,660	499,353(21.32%)	15,581	241,876

※削減率は，最適化によって減少した enterTry() の実行回数の比率を表す.
※実行時間は，例外処理を setjmp 法で実施（全 setjmp 法向け最適化を適用）した場合のベンチマークの実行時間を表す.
※平均実行頻度は最適化時の enterTry() マクロの秒間あたりの実行回数で、総実行回数を実行時間で除して求めた値である.

冗長な enterTry(), exitTry() の除去

図 6.6 から，冗長な enterTry(), exitTry() の除去が _209_db に特に有効な最適化であり，最適化を適用しないと実行速度が 8.4%遅くなってしまうことが判る．最適化が enterTry() の実行回数に与える影響を評価した結果を表 6.3 に示す．表 6.3 から，_209_db について，最適化によって enterTry() の実行回数を 99.95%削減していることが判り，実行速度の改善が偶然によるものではないと判断できる．

冗長な例外処理方式の変換の除去

図 6.6 から，冗長な例外処理方式の変換を除去する最適化に実行速度を改善する効果は余りないといえる．最適化が例外処理方式の変換をどれだけ除去するか評価した結果を表 6.3 に示す．表 6.3 から，最適化によって例外処理方式の変換回数，すなわち例外処理方式変換部における enterTry() の実行回数が 0.31%〜46.35%減少することが判る．46.35%という数字は小さくないが，それにもかかわらず実行速度に影響がでないのは，そもそもイ

ンタプリタから静的コンパイル済みコードを呼び出す回数（表 6.3 の，例外処理変換部で最適化なしの場合に **enterTry()** を実行する回数）自体が少ないためだと考える．

_setjmp() の利用

図 6.6 から，**_setjmp()**，**_longjmp()** の代わりに **setjmp()**，**longjmp()** を使うと相乗平均で 0.70%実行速度が低下することが判る．特に **_213_javac** と **_228_jack** の 2 項目については，それぞれ 1.5%，2.6%と大きな影響が出ているが，表 6.3 から，これらのベンチマーク項目では **tryEnter()** マクロ，すなわち **setjmp()** の実行頻度が高いことが判り，測定結果が偶然ではないといえる．相乗平均で 0.70%という数値は単一の最適化から生じる影響としては必ずしも小さくなく，OS が **_setjmp()** に相当する機能を提供する場合には，**setjmp()** に替えて **_setjmp()** を利用することが望ましいと考える．

6.3.2　2 返戻値法向け最適化の評価

ここでは本論文で提案した 2 返戻値法向け最適化である，下方移動による例外発生検査の除去あるいは集約が実行速度に与える影響を評価する．2 返戻値法向け最適化には，他にインライン展開と手続間解析による冗長な例外発生検査の除去があるが，これらは評価対象から除外する．前者を除外する理由は，インライン展開そのものが実行速度にあたえる影響から例外発生検査の除去のみが実行速度に与える影響を単離することが困難であるためである．後者を除外する理由は，効果がなかったからである．

下方移動による例外発生検査の除去あるいは集約

例外発生検査の下方移動による除去あるいは集約が実行速度に与える影響を評価した結果を図 6.7 に示す．評価は最適化を実施した場合を基準として最適化を適用しなかった場合に実行速度がどれだけ低下するか測定することでおこなった．図 6.7 の縦軸は実行速度の低下率を表し，横軸は測定対象のベンチマーク項目名を表す．図 6.7 から，最適化を適用しないと実行速度が相乗平均で 0.96%低下することが判り，下方移動による最適化がある程度有用であるといえる．

6.3.3　setjmp 法と 2 返戻値法の比較

図 6.8 に，setjmp 法と 2 返戻値法のそれぞれの方法で例外処理を実現するコードでベンチマークを実行し，実行速度を測定した結果を示す．図 6.8 の縦軸は setjmp 法（全 setjmp 法向け最適化あり）による実行速度を 100%とした場合の相対速度を表し，横軸は測定対象のベンチマーク項目名を表す．なお，setjmp 法，2 返戻値法ともに非同期例外は，例外発生検査で **exception** フィールドをポーリングして検出する．表 6.4 に，例外発生検査の

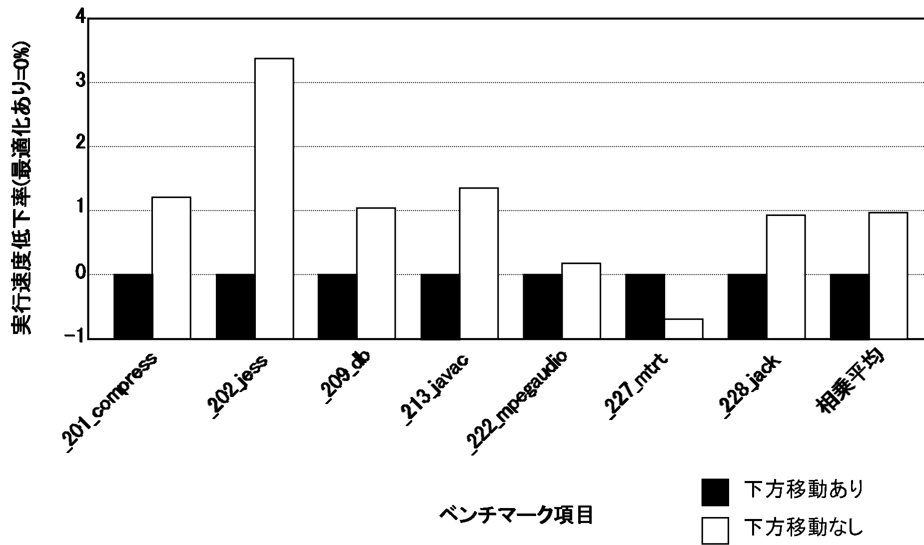

図 6.7: 例外発生検査の下方移動が実行速度に与える影響

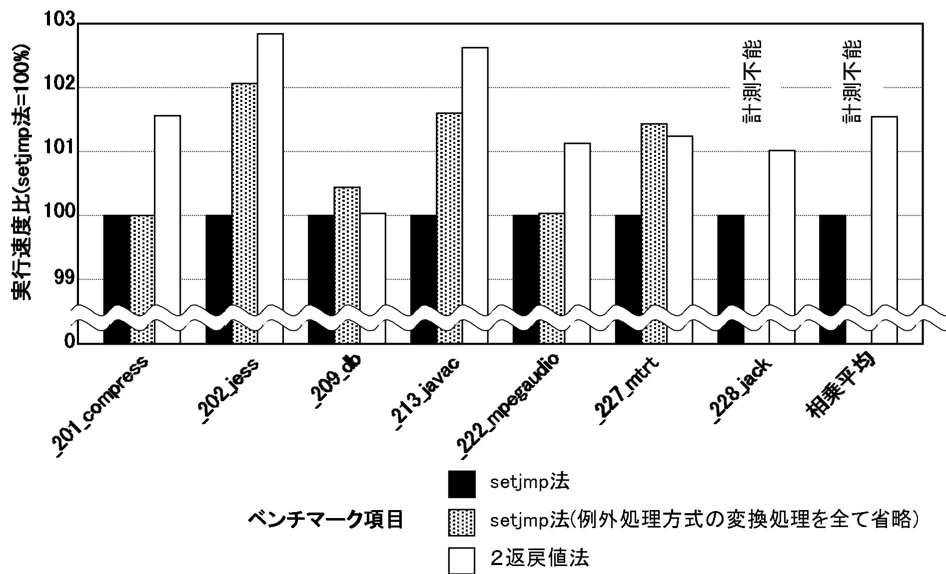

図 6.8: setjmp 法と 2 返戻値法の比較

表 6.4: 例外発生検査の実行回数

ベンチマーク項目名	実行回数（$\times 10^3$ 回）		
	2 返戻値法	setjmp 法	非同期例外用ポーリング
_201_compress	244,302	233,201	233,198
_202_jess	77,900	44,433	36,513
_209_db	141,947	82,195	79,127
_213_javac	97,765	54,692	48,988
_222_mpegaudio	157,782	137,752	137,751
_227_mtrt	107,039	12,218	8,787
_228_jack	158,741	145,138	139,307

実行回数の比較を示す．setjmp 法では，非同期例外をポーリングする他に，静的コンパイル済みコードからインタプリタを呼ぶ際などに例外処理方式を変換する過程で例外発生検査を実行する．このため，setjmp 法における例外発生検査の実行回数は非同期例外のポーリングに必要な回数より多くなる．なお，例外処理方式を変換する際の例外発生検査は非同期例外のポーリングを兼ねる．

2 返戻値法では関数呼出しからの返戻時にも例外発生検査を実行するが，setjmp 法と比較して例外発生検査の実行回数が大幅に増える項目は少ない．この原因は，非同期例外のポーリング回数が多いために増加が目立たないことや，Java2C トランスレータでインライン展開や下方移動を適用して例外発生検査を除去していることにある．

図 6.8 から，全てのベンチマーク項目で 2 返戻値法を採用した方が実行を高速化でき，その差が最大で 2.8%，相乗平均で 1.5%に達することが判る．setjmp 法の方が遅くなる原因の 1 つとして，例外処理方式の変換のコストが考えられる．例外処理方式の変換は，JeanPaul のインタプリタが setjmp 法以外の方法で例外を処理するために必要になる処理であり，インタプリタの実現によっては不要になりうる．そこで，図 6.8 には setjmp 法と 2 返戻値法による実行速度の他に，例外処理方式を変換する処理を強制的に全て省略して setjmp 法でベンチマークを実行した場合の実行速度も併せて示した．この場合の例外発生検査の実行回数は表 6.4 の非同期例外用ポーリングの実行回数と同一である．**_228_jack** は実行時に例外処理方式の変換を必要とするため，変換処理を省略すると正常に動作しないが，他の項目については実行速度を測定できた．なお，2 返戻値法では例外処理方式がインタプリタと似ているため，例外処理方式の変換は不要である．

図 6.8 から，例外処理方式の変換にかかるコストを除いても，setjmp 法による実行速度が 2 返戻値法と同等以下であることが判る．

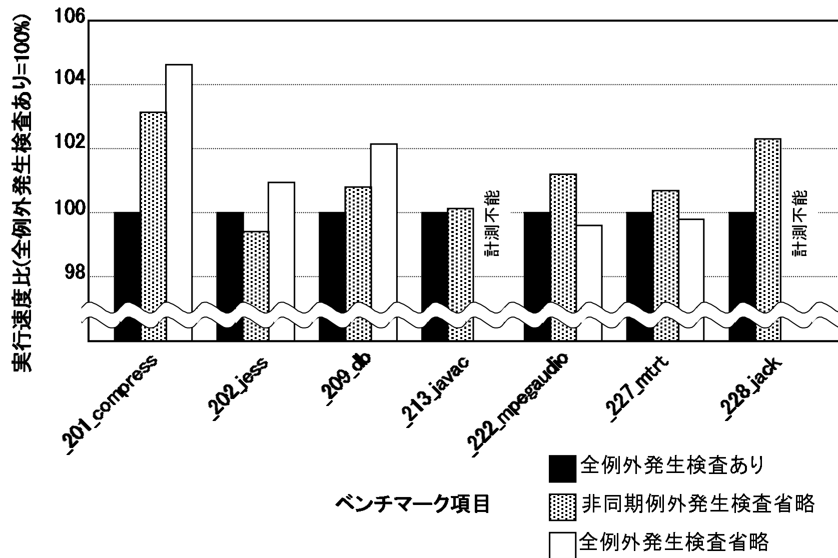

図 6.9: 例外発生検査のオーバヘッド

6.3.4 2 返戻値法における例外処理のオーバヘッド

これまでの評価結果から，可搬的な例外処理の実現方式としては 2 返戻値法の方が優れていることが判った．ここでは 2 返戻値法について，もう一歩踏み込み，2 返戻値法による例外処理の実現にどれだけオーバヘッドがかかるか評価する．評価は，2 返戻値法のオーバヘッドの発生原因である，関数呼出しの直後などに挿入する例外発生検査のコードを強制的に全て排除した場合との実行速度の比較でおこなった．例外発生検査のコードを排除すると，実行中に例外を投げるベンチマーク項目である **_213_javac** と **_228_jack** は正常に動作しないので，これらの項目を評価対象から除外する．図 6.9 の評価結果から，例外発生検査のコードの省略により 1% 以上実行が高速になるベンチマーク項目は，**_201_compress** と **_209_db** だけであることが判る．このことから，例外処理の実現方式として，2 返戻値法の代わりに，例外発生検査のコードを必要とせず，より効率がよい（代わりに可搬性が低く Java2C トランスレータでは採用できない）といわれる表引き法を採用できたとしても，大きな実行速度の改善は望めないと推測する．

図 6.9 には，2 返戻値法による例外処理において非同期例外検出用のポーリングがどの程度のオーバヘッドをもたらすか評価した結果をあわせて示した．非同期例外をポーリングするコードは，ループ本体を実行する過程で必ず 1 回はポーリングするように必要とあらば挿入するが，評価はこの非同期例外の検出のみを目的とする例外発生検査のコードを強制的に排除した場合の実行速度との比較によりおこなった．

非同期例外検出用ポーリングのオーバヘッドは，ループ本体が小さいマイクロベンチマーク（図 6.10）では大きく，36.8%に達した（表 6.5）．しかし，SPECjvm98 のように実用的なサイズのベンチマークでは最大でも 3.2%にとどまり，それほど大きくはならない（図 6.9）．このことから，ループの最適化が実行速度を大きく左右する科学技術計算系

```
int sum(int[] array){
  int result = 0;
  for(int i=0; i<array.length; i++)
    result += array[i];
  return (result);
}
```

図 6.10: マイクロベンチマーク

表 6.5: マイクロベンチマーク実行結果

非同期例外向け例外発生検査の挿入	実行時間（ms）
あり	14.855
なし	10.862
実行速度比	136.8%

※配列長は 1,000 とした.

のアプリケーションでなく，SPECjvm98 のような整数系のアプリケーションを主に実行する場合には，非同期例外をポーリングで検出しても実行速度が大きく低下することはないと考える.

第7章　結論

本章では，まず 7.1 節において本論文のまとめを示し，次に 7.2 節で本論文で述べた動的ロード と例外処理の実現に関する今後の課題と展望について述べる．

7.1　まとめ

本論文では，Java2C トランスレータを実現する上で問題となる点に動的ロード と例外処理の実現があることを指摘し，その解決策を提案した．

7.1.1　動的ロード の実現

動的ロード を実現し，言語仕様に準拠することは，Java2C トランスレータの開発における必須事項である．言語仕様に準拠しない言語処理系は，ソフトウェアを開発および保守する現場に混乱をもたらす．本論文では，Java2C トランスレータにおいて動的ロード を実現する方法を提案した．さらに，提案した動的ロード の実現と連携した最適化技法として，クラス初期化検査の除去と I-call if 変換を提案した．

クラス初期化検査の除去

クラス初期化検査の除去は重要な最適化であり，SPECjvm98 を使って評価した結果，プログラムの実行を相乗平均で 45%高速化する効果があることが判った．Java 向けインタプリタや動的コンパイラでは，実行コード を動的に書き換えるなどの方法でクラス初期化検査を除去するが，この方法は Java2C トランスレータでは利用できない．本論文では Java2C トランスレータでクラス初期化検査を除去する方法として，メソッド を静的コンパイルした際に参照した全クラスについて，実行時に初期化とリンクが済むまで，静的コンパイル済みコード のリンクを遅延する方法を提案した．本論文で提案した方法の問題点は，静的コンパイル済みコード のリンクが済むまでの間，インタプリタでプログラムを実行することから，実行速度に悪影響がでる可能性があることである．この影響について評価した結果，実行速度への影響は相乗平均で 0.8%と大きくないことが判った．

I-call if 変換

I-call if 変換について，I-call if 変換を適用したコードにおける呼出し先メソッドの検索方法が，動的ロードのオーバヘッドに影響することを指摘した．I-call if 変換には，メソッドの検索方法が異なる，クラスチェック変換とメソッドチェック変換の 2 種類のバリエーションがあるが，メソッドチェック変換に動的ロードのオーバヘッドを軽減する効果があることを示した．SPECjvm98 による評価の結果，メソッドチェック変換を使うと，クラスチェック変換を使うより最大 9.8%実行を高速化できることが判った．

7.1.2 例外処理の実現

例外処理の実現に関しては，Java2C トランスレータ向けの実現方法である setjmp 法と 2 返戻値法を比較した．また，個々の実現方法向けの最適化技法を提案した．

setjmp 法向け最適化

setjmp 法向けに次の最適化技法を提案した．

1. `volatile` 宣言の最小化
2. 冗長な `enterTry()`, `exitTry()` の除去
3. `_setjmp()` の利用
4. 冗長な例外処理方式の変換の除去

SPECjvm98 を使って評価した結果，これらの最適化を全て適用すると，実行速度を相乗平均で 4.7%高速化できることが判った．

2 返戻値法向け最適化

2 返戻値法向けに次の最適化技法を提案した．

1. 下方移動による冗長な例外発生検査の除去
2. 下方移動による例外発生検査の集約

SPECjvm98 を使って評価した結果，これらの最適化を全て適用すると，実行速度を相乗平均で 0.96%高速化できることが判った．

setjmp 法と 2 返戻値法の比較

setjmp 法と 2 返戻値法のどちらで例外処理を実現した方が実行を高速化できるか比較をおこなった．比較は，本論文で提案した setjmp 法あるいは 2 返戻値法向けの全最適化を適用した状態で実施した．SPECjvm98 を使って比較した結果，2 返戻値法で例外処理

を実現した方が実行速度を相乗平均で 1.5%高速化できることが判った．実現の単純さという観点からも，2 返戻値法の方が優れていると判断できた．

また，例外処理を実現しない場合と 2 返戻値法で例外処理を実現する場合を比較し，2 返戻値法による例外処理の実現にかかるオーバヘッドを測定した結果，1.4%程度であることが判った．この結果から，表引き法など，2 返戻値法以外の方法で例外処理を実現しても，2 返戻値法を使う場合より実行速度を大幅には改善できないことが判った．

7.2 今後の課題と展望

7.2.1 動的ロード

本論文で提案した動的ロードの実現に残る課題として，仮定クラス情報の圧縮がある．仮定クラス情報は少なからざる大きさになるので，その圧縮方法を検討することは組込み機器の小さな記憶領域にかける負担を軽減する上で大きな意味を持つ．

本論文で提案した動的ロードおよび，クラス初期化検査の除去と I-call if 変換の実現方法は，Java の言語仕様に特化したものであり，現在のところ Java2C トランスレータでしか利用することができない．ただし，今後，新たに現れるプログラミング言語 L において，Java の言語仕様と同じ手順でクラスを動的ロード，初期化するように規定されるならば，本論文で提案した技術を用いて L から C 言語へのトランスレータを実現できる．

7.2.2 例外処理の実現

本論文で示した setjmp 法と 2 返戻値法の比較に残る課題は，比較結果に普遍性があるか判らないことである．その原因は，比較の根拠となるデータである実行速度の測定を，単一プラットフォームだけでおこなったことにある．今後複数のプラットフォームで測定をおこない，比較結果の普遍性を検証したい．また，本論文では実行速度を基準として比較をおこなったが，コードサイズなど別の観点から比較する必要もあると考える．

本論文で示した setjmp 法と 2 返戻値法の比較結果は，今後，例外処理の機能を提供するプログラミング言語から C 言語へのトランスレータを開発する際の参考資料となる．C 言語へのトランスレータは過去に数多く開発されてきたが，今後も開発され続けると考える．C 言語へのトランスレータが過去に開発されてきた理由は，ある程度の性能をもつコンパイラを小さな工数で開発できるからである．このことは，新たなプログラミング言語を提案し，その普及を図る過程において大きな意味をもつ．実際，JeanPaul を含め Java2C トランスレータの多くは，Java の黎明期に開発された．Java がインターネットと共に登場したように，今後も社会の変化と共に新たなプログラミング言語が登場し，その際に C 言語へのトランスレータが必要になりうる．本論文で示した例外処理の実現に関する定量的な評価結果は，今後の C 言語へのトランスレータの開発に役立つと考える．

謝辞

本論文の執筆にあたり数々の御意見を頂いた慶應義塾大学大学院理工学研究科の原田賢一教授に感謝します．

また，本論文に様々なご意見をお寄せ頂いた慶應義塾大学大学院理工学研究科の大野義夫教授，櫻井彰人教授，高田眞吾講師，中央大学理工学部の土居範久教授に感謝します．

さらに，本研究の機会を与えて下さった日立製作所ソフトウェア事業部の菊池純男氏，数々の論文を査読して頂いた日立製作所システム開発研究所の西山博康氏に感謝します．

最後に，修士号を取得した 7 年前から変わらず数々の支援をしてくれた両親に，祖母に，今では妻となった静香に，新たに生まれた息子である秋宣に，16 歳になった犬のダームに，21 歳になった鮒に感謝します．

参考文献

[Agesen99] Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y. S. and White, D.: An Efficient Meta-Lock for Implementing Ubiquitous Synchronization, *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 1999)*, ACM, pp. 207–222 (1999).

[Aho86] Aho, A. V., Sethi, R. and Ullman , J. D.: *Compilers: Principles, Tools and Techniques*, Addison-Wesley Publishing Company, Reading, Massachusetts (1986).

[Alpern00] Alpern, B., Attanasio, C. R., Barton, J. J., Bruke, M. G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Flynn-Hummel, S., Lieber, D., Litvionv, V., Mergen, M. F., Ngo, T., Russel, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H. and Whaley, J.: The Jalapeño virtual machine, *IBM Systems Journal*, Vol. 39, No. 1, pp. 211–238 (2000).

[Antoniu01] Antoniu, G., Bauge, L., Hatcher, P., MacBeth, M., McGuigan, K. and Namyst, R.: The Hyperion System: Compiling Multithreaded Java bytecode for Distributed Execution, *Parallel Computing*, Vol. 27, No. 10, pp. 1279–1297 (2001).

[Aoki01] Aoki, T. and Eto, T.: On the Software Virtual Machine for the Real Hardware Stack Machine, *Proceedings of Java Virtual Machine Research and Technology Symposium*, pp. 221–232 (2001).

[Bacon98] Bacon, D. F., Konuru, R., Murthy, C. and Serrano, M.: Thin Locks: Featherweight Synchronization for Java, *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1998)*, ACM, pp. 258–268 (1998).

[Berekovic97] Berekovic, M., Kloos, H. and Pirsch, P.: Hardware Realization of a Java Virtual Machine for High Performance Multimedia Applications, *Proceedings of IEEE Workshop on Signal Processing Systems*, pp. 479–488 (1997).

[Blanchet98] Blanchet, B.: Escape Analysis: Correctness Proof, Implementation and Experimental Results, *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1998)*, ACM, pp. 25–37 (1998).

[Blanchet99] Blanchet, B.: Escape Analysis for Object Oriented Languages. Application to Java(TM), *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 1999)*, ACM, pp. 20–34 (1999).

[Bohem88] Bohem, H.-J. and Weiser, M.: Garbage Collection in an Uncooperative Environment, *Software Practice and Experience*, Vol. 18, No. 9, pp. 807–820 (1988).

[Calder94] Calder, B. and Grunwald, D.: Reducing Indirect Function Call Overhead In C++ Programs, *Proceedings of the 1996 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1994)*, ACM, pp. 397–408 (1994).

[Cameron92] Cameron, D., Faust, P., Lenkov, D. and Mehta, M.: A Portable Implementation of C++ Exception Handling, *USENIX C++ Technical Conference*, pp. 225–243 (1992).

[Chang98] Chang, L.-C., Ton, L.-R., Kao, M.-F. and Chung, C.-P.: Stack Operation Folding in Java Processors, *IEE Proceedings on Computer and Digital Techniques*, Vol. 145, No. 5, pp. 330–340 (1998).

[Choi99] Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C. and Midkiff, S.: Escape Analysis for Java, *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 1999)*, ACM, pp. 1–19 (1999).

[Cierniak00] Cierniak, M., Lueh, G.-Y. and Stichnoth, J. M.: Practicing JUDO: Java Under Dynamic Optimizations, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000)*, ACM, pp. 13–26 (2000).

[Dinechin00] de Dinechin, C.: C++ Exception Handling, *IEEE Concurrency*, Vol. 8, No. 4, pp. 72–79 (2000).

[Dean95] Dean, J., Grove, D. and Chambers, C.: Optimization of Object-Oriented Programs Using Static Hierarchy Analysis, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP 1995)*, pp. 77–101 (1995).

[Detlefs99] Detlefs, D. and Angesen, O.: Inlining of Virtual Methods, *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP 1999)*, pp. 259–278 (1999).

[Dice01] Dice, D.: Implementing Fast Java Monitors with Relaxed-Locks, *Proceedings of Java Virtual Machine Research and Technology Symposium*, pp. 79–90 (2001).

[ElKharashi00-1] El-Kharashi, M. W., ElGuibaly, F. and Li, K. F.: A Quantitative Study for Java Microprocessor Architectural Requirements. Part I: Instruction set design, *Microprocessors and Microsystems*, Vol. 24, No. 5, pp. 225–236 (2000).

[ElKharashi00-2] El-Kharashi, M. W., ElGuibaly, F. and Li, K. F.: A Quantitative Study for Java Microprocessor Architectural Requirements. Part II: High-Level Language Support, *Microprocessors and Microsystems*, Vol. 24, No. 5, pp. 237–250 (2000).

[Gagnon01] Gagnon, E. M. and Hendren, L. J.: SableVM: A Research Framework for the Efficient Execution of Java Bytecode, *Proceedings of Java Virtual Machine Research and Technology Symposium*, pp. 27–40 (2001).

[Gosling96] Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley Publishing Company, Reading, Massachusetts (1996).

[Hennesy90] Hennessy, J. L. and Patterson, D. A.: *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Francisco, California (1990).

[Hölzle92] Hölzle, U., Chambers, C. and Ungar, D.: Debugging Optimized Code with Dynamic Deoptimization, *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1992)*, ACM, pp. 32–43 (1992).

[Hölzle94] Hölzle, U. and Ungar, D.: Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback, *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1994)*, ACM, pp. 326–336 (1994).

[Howard97] Howard, R.: Developing and Deploying Server-hosted Applications with Java, *http://www.twr.com/java/white-paper.html* (1997).

[Hsieb97] Hsieb, C.-H., Conte, M., Johnson, T., Gyllenbaal, J. and mei Hwu, W.: OPTIMIZING NET Compilers for Improved Java Performance, *IEEE COMPUTER*, Vol. 30, No. 6, pp. 67–75 (2000).

[Ingalls00] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S. and Kay, A.: Back to the Future The Story of Squeak, A Practical Smalltalk Written in Itself, *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 1997)* , pp. 318–326 (1997).

[IBM94] International Business Machines: *The PowerPC Architecture: A Specification of RISC Processors 2nd Edition*, Morgan Kaufmann Publishers, Inc., San Francisco, California (1994).

[Jones96] Jones, R. and Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley and Sons Inc., Indianapolis, Indiana (1996).

[Ishizaki00] Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H. and Nakatani, T.: A Study of Devirtualization Techniques for a Java Just-In-Time Compiler, *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2000)*, pp. 294–310 (2000).

[Ishizaki99] Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T. and Nakatani, H. K. T.: Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler, *Proceedings of the ACM 1999 Conference on Java Grande*, ACM, pp. 119–128 (1999).

[Kawachiya02] Kawachiya, K., Koseki, A. and Onodera, T.: Lock Reservation: Java Locks can mostly do without atomic operations, *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2002)*, ACM, pp. 130–141 (2002).

[Kawahito00] Kawahito, M., Komatsu, H. and Nakatani, T.: Effective Null Pointer Check Elimination Utilizing Hardware Trap, *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pp. 139–149 (2000).

[Kim00] Kim, A. and Chang, M.: An Advanced Instruction Folding Mechanism for a Stackless Java Porcessor, *Proceedings of the International Conference on Computer Design*, pp. 565–566 (2000).

[Kim01] Kim, A. and Chang, M.: Advanced POC Model-based Java Instruction Folding Mechanism, *Proceedings of the 26th Euromicro Conference*, Vol. 1, pp. 332–338 (2000).

[Krall97] Krall, A. and Grafl, R.: CACAO — A 64-bit Just-In-Time Compiler, *Concurrency: Practice and Experience*, Vol. 9, No. 11, pp. 1017–1030 (1997).

[Krall98] Krall, A. and Probst, M.: Monitors and Exceptions: how to implement Java efficiently, *Concurrency: Practice and Experience*, Vol. 10, No. 11–13, pp. 837–850 (1998).

[Levy95] Levy, G. F.: Improving the Output of the FORTRAN to C Translator, f2c, *Software Practice and Experience*, Vol. 25, No. 2, pp. 217–227 (1995).

[Lindholm96] Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, Addison-Wesley Publishing Company, Reading, Massachusetts (1996).

[McGhan98] McGhan, H. and O'Connor, M.: PicoJava: A Direct Execution Engine for Java Bytecode, *IEEE COMPUTER*, Vol. 31, No. 10, pp. 22–30 (1998).

[Muller97] Muller, G., Moura, B., Bellard, F. and Consel, C.: Harrisa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code, *USENIX Conference on Object-Oriented Technologies and Systems*, USENIX, pp. 1–20 (1997).

[O'Connor97] O'Connor, J. M. and Tremblay, M.: picoJava-I: The Java Virtual Machine in Hardware, *IEEE Micro*, Vol. 17, No. 2, pp. 45–53 (1997).

[Onodera99] Onodera, T. and Kawachiya, K.: A Study of Locking Objects with Bimodal Fields, *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 1999)*, pp. 223–237 (1999).

[Paleczny99] Paleczny, M., Vick, C. and Click, C.: The Java HotSpot Server Compiler, *Proceedings of Java Virtual Machine Research and Technology Symposium*, pp. 1–12 (2001).

[Pande94] Pande, H. D. and Ryder, B. G.: Static Type Determination for C++, *USENIX Sixth C++ Technical Conference*, USENIX, pp. 85–97 (1994).

[Park90] Park, Y. G. and Goldberg, B.: Higher Order Escape Analysis: Optimizing Stack Allocation in Functional Program Implementations, *Proceedings of 3rd European Symposium on Programming (Lecture Notes in Computer Science 432)*, pp. 152–160 (1990).

[Park91] Park, Y. G. and Goldberg, B.: Higher Order Escape Analysis: Optimizing Reference Counting based on the Lifetime of References, *ACM SIGPLAN Notices*, Vol. 26, No. 9, pp. 178–189 (1991).

[Park92] Park, Y. G. and Goldberg, B.: Escape Analysis on Lists, *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1992)*, ACM, pp. 116–127 (1992).

[Proebsting97] Proebsting, T., Townsend, G., Bridges, P., Hartman, J., Newsham, T. and Watterson, S.: Toba: Java For Applications A Way Ahead of Time (WAT) Compiler, *USENIX Conference on Object-Oriented Technologies and Systems*, USENIX, pp. 41–53 (1997).

[Richter96] Richter, J.: *Advanced Windows 3rd Edition*, Microsoft Press, Grove City, Ohio (1996).

[Schilling98] Schilling, J. L.: Optimizing Away C++ Exception Handling, *ACM SIGPLAN Notices*, Vol. 33, No. 8, pp. 40–47 (1998).

[Sreedhar00] Sreedhar, V. C., Bruke, M. and Choi, J.-D.: A Framework for Optimization in the Presence of Dynamic Class Loading, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000)*, ACM, pp. 196–207 (2000).

[SPEC98] Standard Performance Evaluation Corporation: SPECjvm98 benchmarks, *http://www.spec.org/osg/jvm98/* (1998).

[Suganuma00] Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H. and Nakatani, T.: Overview of the IBM Java Just In Time Compiler, *IBM Systems Journal*, Vol. 39, No. 1, pp. 175–193 (2000).

[Ton00] Ton, L.-R., Chang, L.-C. and Chung, C.-P.: Exploiting Java Bytecode Parallelism by Enhanced POC Floding Model, *Proceedings of Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference (Lecture Note in Computer Science 1900)*, pp. 994–997 (2000).

[Ton01] Ton, L.-R., Chang, L.-C. and Chung, C.-P.: Optimal Folding for Java Processors, *Proceedings of Java for High-Performance Computing*, pp. 61–72 (2001).

[Ton97] Ton, L.-R., Chang, L.-C., Kao, M.-F., Tseng, H.-M., Shang, S.-S., Ma, R.-L., Wand, D.-C. and Chung, C.-P.: Instruction Folding in Java Processor, *Proceedings of 1998 International Conference on Parallel and Distributed Systems*, pp. 138–143 (1997).

[Triebel00] Triebel, W. A.: *Itenium Architecture for Software Developers*, Intel Press, Santa Clara, California (2000).

[Vijaykrishnan98] Vijaykrishnan, N., Ranganathan, N. and Gadekarla, R.: Object-Oriented Architectural Support for a Java Processor, *Proceedings of the 12th European Conference on Object-Oriented Programming (Lecture Note in Computer Science 1445)*, pp. 330–355 (1998).

[Whaley99] Whaley, J. and Rinard, M. C.: Compositional Pointer and Escape Analysis for Java Programs, *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 1999)*, ACM, pp. 187–206 (1999).

[Yuasa90] Yuasa, T.: Design and Implementation of Kyoto Common Lisp, *Journal of Infomation Processing*, Vol. 13, No. 3, pp. 284–295 (1990).

[石崎 02] 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動的メソッド呼出しの直接 devirtualization, 情報処理学会論文誌, Vol. 43, No. 1, pp. 124–136 (2002).

[緒方 02] 緒方一則, 小松秀昭, 中谷登志男: PowerPC プロセッサの特性を考慮した高速 Java バイトコードインタープリタの構成法, 情報処理学会論文誌プログラミング, Vol. 43, No. SIG8(PRO15), pp. 1–10 (2002).

[小野寺 94] 小野寺民也: 保守的ごみ集め, 情報処理, Vol. 35, No. 11, pp. 1020–1026 (1994).

[小野寺 97] 小野寺民也: オブジェクト指向言語におけるメッセージ送信の高速化技法, 情報処理, Vol. 38, No. 4, pp. 301–310 (1997).

[川人 01] 川人基弘, 小松秀昭, 中谷登志男: Java 言語に対する効果的な Null チェックの最適化手法, 情報処理学会論文誌プログラミング, Vol. 42, No. SIG2(PRO9), pp. 81–96 (2001).

[川本 02] 川本琢二, 春名修介, 金丸智一: 家電向け Java JIT コンパイラの構成方法とその評価, 情報処理学会論文誌プログラミング, Vol. 43, No. SIG8(PRO15), pp. 37–45 (2002).

[木村 02] 木村篤彦, 中島康彦, 宮田佳昭, 中川伸二, 北村俊明, 五島正裕, 森眞一郎, 富田眞治: 低電力 Java プロセッサのための投機的クロック制御, 情報処理学会論文誌, Vol. 43, No. 6, pp. 1956–1967 (2002).

[古関 01] 古関聰, 小松秀昭: 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, Vol. 42, No. SIG2(PRO9), pp. 26–36 (2001).

[千葉 02] 千葉雄司: 組込み機器向け Java2C トランスレータにおける 2 返戻値法を使った例外処理の実現, 情報処理学会論文誌プログラミング, Vol. 43, No. SIG1(PRO13), pp. 85–96 (2002).

[八杉 01] 八杉昌宏, 馬谷誠二, 鎌田十三朗, 田畑悠介, 伊藤智一, 小宮常康, 湯淺太一: オブジェクト指向並列言語 OPA のためのコード生成手法, 情報処理学会論文誌プログラミング, Vol. 42, No. SIG2(PRO12), pp. 1–13 (2001).

研究業績

投稿論文

1. 千葉 雄司: Javaにおけるごみ集めの基本処理の高速化, 情報処理学会論文誌, Vol. 39, No. SIG1(PRO1), pp. 43-49 (1999).
2. 千葉 雄司: Javaにおける静的コンパイル済みコードのリンク方法, 情報処理学会論文誌, Vol. 42, No. SIG2(PRO9), pp. 37-47 (2001).
3. 千葉 雄司: Java向け静的コンパイラによるメソッド呼出しの高速化, 情報処理学会論文誌, Vol. 42, No. SIG7(PRO11), pp. 46-56 (2001).
4. 千葉 雄司: Java2Cトランスレータにおける例外処理の実現, 情報処理学会論文誌, Vol. 42, No. SIG11(PRO12), pp. 14-24 (2001).
5. 千葉 雄司: 組込み機器向けJava2Cトランスレータにおける2返戻値法を使った例外処理の実現, 情報処理学会論文誌, Vol. 43, No. SIG1(PRO13), pp.85-96 (2002).
6. 千葉 雄司: Java2Cトランスレータにおける可搬性のオーバヘッド, 情報処理学会論文誌, Vol. 43, No. SIG8(PRO15), pp. 23-36 (2002).

海外発表

1. Yuji Chiba: Devirtualization techniques in a static compiler for Java, in Proceedings of the third workshop on Java for High Performance Computing, pp.49-59 (2001).
2. Yuji Chiba: Translating Java to C without inserting class initialization tests, in Proceedings of Java Workshop on Parallel and Distributed Computing, in CD-ROM (2002).
3. Yuji Chiba: Optimizations for exception handling by two-return-values method, in Proceedings of International Symposium on Future Software Technology, pp.113-118 (2001).

その他発表

1. 千葉 雄司, 土居 範久: Standard ML of New Jerseyにおける制御スタックの実装, 日本ソフトウェア科学会第12回大会論文集, pp. 269-272 (1995).

2. 千葉 雄司, 土居 範久: コピー式ごみ集めの応答性の向上のためのページング運用法, 日本ソフトウェア科学会第 13 回大会論文集, pp. 157-160 (1996).
3. 千葉 雄司: Java 向け静的コンパイラによるクラス間最適化, 情報処理学会論文誌, Vol. 40, No. SIG7 (PRO4), pp.92 (1999).
4. 千葉 雄司: Java 向け静的コンパイル方式, 第 41 回プログラミングシンポジウム, pp.115-122 (2000).
5. 千葉 雄司: Java2C トランスレータにおける例外処理の実現, 第 42 回プログラミングシンポジウム, pp.135-142 (2001).
6. 千葉 雄司: 1.8 人年による Java2C トランスレータの開発, 第 6 回プログラミングおよび応用のシステムに関するワークショップ, http://spa.jssst.or.jp/2003/program (2003).