Java2Cトランスレータにおける可搬性のオーバヘッド

千 葉 雄 司 †

Javaでは，セキュリティの確保やソフトウエア生産性の向上を目的として，null検査によって不正なメモリ参照を防いだり，スタックトレース情報を提供したりする．これらの処理はJavaが暗黙に実施するものであり，逆に，暗黙の処理を実施しないプログラムをJavaで書くことはできない．暗黙の処理は実行時間やコードサイズに悪影響を与えうるが，暗黙の処理のオーバヘッドを軽減する作業は，Javaでは，プログラマではなく，コンパイラなど実行環境が実施する．暗黙の処理の中にはプラットホーム依存の技法を使うと効率的に実現できるものがあり，たとえばnull検査はページトラップを使うと明示的なコードなしで実現できるが，ページトラップのようにプラットホーム依存の技法は可搬性を重視するJavaコンパイラでは採用しにくい．本論文の目的はJava2Cトランスレータと Cコンパイラから構成する可搬性を重視したJavaコンパイラにおいて，プラットホーム依存の技法を使わずに暗黙の処理を実現するとどれだけオーバヘッドが生じるか評価することにある．SPECjvm98を使って評価した結果，プラットホーム依存の技法を使えば省略できる暗黙の処理向けのコードサイズが，Java2Cトランスレータが生成するコード全体の40.03%を占めることが分かった．また，それらのコードの実行に，PentiumIII 450 MHzと Am5x86 133 MHzを搭載したPCにおいてそれぞれ実行時間全体の5.75%および10.56%を費やすことが分かった．

Overheads for Portability in a Java2C Translator

Yuji Chiba†

Java implicitly provides features such as null tests to prevent invalid memory access for security, and stack traces to improve software productivity, and it is impossible to write a Java program that is free from the implicit features. In Java, the respoinsibility to remove overheads for the implicit features is not on the programmer but on the Java runtime environment, especially on its compiler. A Java compiler that is composed of a Java2C translator and a C compiler has good portability, but it suffers from more overheads to implement implicit features because a Java2C translator cannot use techniques that depend on platform. Some of the implicit features, such as null tests, can be implemented with little performance overhead by platform dependent techniques such as page traps. This paper shows the overheads Java2C translator suffers from inability to use platform dependent techniques. The result of SPECjvm98 showed that codes for implicit features share 40.03% of code-size a Java2C translator emits and the codes consume respectively 5.75% and 10.56% of the total execution time in PCs with PentiumIII 450 MHz or Am5x86 133 MHz.

1. はじめに

Java™☆は生産性の高さなどから広い分野に普及しつつあるオブジェクト指向プログラミング言語である．Javaには実行速度が遅いという問題があるが，その原因の1つに，null検査などJavaが暗黙に実施する処理がある[5]．null検査とは，変数が指示するオブジェクトを参照する前に，変数の値がnullか検査し，nullならば例外`NullPointerException`を生成して投擲する処理を表す．null検査は不正なメモリ参照を防止してセキュリティを確保したり，エラーを検出してソフトウエア生産性を向上したりするうえで有用だが，実行時オーバヘッドの原因ともなる．Javaではnull検査など暗黙の処理を実施しないプログラムを書くことはできず，暗黙の処理から生じるオーバヘッドを削減する作業は，プログラマではなく，コンパイラなど実行環境が実施する．

PC向けのJavaコンパイラとしては，実行時にコンパイルを行うJust In Time（JIT）コンパイラが一般的だが，組込み機器の分野では実行前にあらかじめコ

† 日立製作所システム開発研究所
Systems Development Laboratory, Hitachi, Ltd.

☆ Javaは米国およびその他の国における米国Sun Microsystems, Inc.の商標です．

表 1 暗黙の処理
Table 1 Implicit features.

処理名	略号
null 検査	Null
スタックオーバフロー検査	OvFlw
スタックトレース情報の提供	Trace
例外処理のサポート	ETest
非同期例外のサポート	Async
OutOfMemoryError のサポート	OutOf

ンパイルしておく静的コンパイラを使う場合もある．これは，PC に比べてプロセッサ資源やメモリ資源が乏しい組込み機器では，実行時にコンパイルを行うことや，メモリ上にコンパイラを記録しておくことが困難だからである．

Java 向け静的コンパイラの構成法の 1 つに，Java から C 言語へのトランスレータ（本論文では Java2C トランスレータと呼ぶ）を使う方法がある．この方法では，静的コンパイラを Java2C トランスレータと既存の C コンパイラで構成する．コンパイルにあたっては，まず，Java2C トランスレータで Java のソースあるいはバイトコードを C ソースに変換し，さらに C コンパイラを適用して機械コードを得る．Java2C トランスレータを使った静的コンパイラの構成法は，可搬性に優れ，特に組込み機器向けコンパイラの開発作業量を軽減するうえで役立つ．組込み機器では様々なプラットホーム（CPU や OS など）を使うため，コンパイラの開発にあたっては，プラットホーム依存のコード生成部の開発作業量が問題になる．Java2C トランスレータを使う方法では，コード生成部として，ほとんどのプラットホームで提供されている既存の C コンパイラを使うため，機種非異存の Java2C トランスレータを開発するだけで，数多くのプラットホーム用の Java 向けコンパイラを開発できる．

Java2C トランスレータは可搬性を重視した Java 向けコンパイラの開発方法だが，反面，プラットホーム依存の機能や C 言語で表現できない技法を利用しないために，コードサイズが大きくなったり，実行時間が長くなったりする不利益を被る．特に，null 検査など Java が暗黙に実施する処理の中には，ページトラップなどプラットホーム依存の技法を使えば効率的に実現できるものがある．**表 1** に，Java2C トランスレータでは使えない技法によって効率的に実現できる暗黙の処理を示す．表 1 に示す処理のうち，非同期例外と OutOfMemoryError のサポートは，例外処理のサポートの一環である．

本論文の目的は，表 1 に示す暗黙の処理を Java2C トランスレータで実現するにあたり，プラットホーム依存の技法や C 言語で表現できない技法を使えないことが，どれだけコードサイズや実行時間に影響を及ぼすか評価することにある．本論文で C 言語とは，いわゆる ANSI C☆を意味する．これは，現在市場にある C コンパイラのほとんどが ANSI C をサポートしており，Java2C トランスレータが出力する C ソースを ANSI C に準拠したものにすれば大概の C コンパイラで機械語に変換できるからである．C コンパイラの中には独自の機能拡張によって C 言語に ANSI C 以上の表現能力を与えるものもある．そういった機能を Java2C トランスレータが出力する C ソースの中で使うと，たとえば暗黙の処理をより効率的に実現できる場合もあるが，反面，Java2C トランスレータが特定の C コンパイラに依存することになり，その分だけ可搬性を失う．本論文では可搬性を重視し，特定の C コンパイラ独自の機能拡張については言及しない．

本論文では，表 1 に示す個々の暗黙の処理について，それぞれの Java2C トランスレータにおける実現を示し，そのコードが Java2C トランスレータが通常出力するコードサイズ全体のどれだけの割合を占め，その実行に実行時間全体のどれだけの割合を費やすかを評価する．ここで「Java2C トランスレータが通常出力するコード」とは，表 1 に示すすべての暗黙の処理をサポートするが，暗黙の処理から生じるオーバヘッドを軽減するために，暗黙の処理を実現するコードのうち，Java2C トランスレータの最適化で冗長と判断できるものを除去したコードを意味する．したがって，たとえば表 1 に示す処理の 1 つである null 検査のコードサイズが Java2C トランスレータが通常出力するコード全体の中で占める割合とは，Java2C トランスレータの最適化では除去できなかった null 検査のコードサイズが全コード中に占める割合を意味する．Java2C トランスレータで実施した暗黙の処理向け最適化については 2 章から 5 章で詳述するが，たとえばフロー解析による冗長な null 検査の削除などがある．

個々の暗黙の処理から生じる影響の評価は，Java2C トランスレータが通常出力するコードと，該当する暗黙の処理を実現するために Java2C トランスレータが挿入したコードをすべて強制削除した場合の比較によって実施する．表 1 に示す暗黙の処理はいずれも，Java2C トランスレータで利用できない技法を使えば明示的なコードなしで実現できる．したがって，たとえば Java2C トランスレータが出力するコードか

☆ 米国規格協会 American National Standards Institute, ANSI のドキュメント ANSIX3.159-1989 による C 言語.

ら null 検査を実現するためのコードを強制削除すると，Java2C トランスレータでは使えない技法で null 検査を実現した場合に近いコードが得られる．そこで，null 検査を強制削除したコードの大きさと，Java2C トランスレータが通常出力するコードの大きさの差分から，Java2C トランスレータが通常出力するコード全体の中で，null 検査のコードがどれだけの割合を占めるかを求める．また，それぞれの実行時間の差分から，null 検査の実行に実行時間全体のどれだけの割合を費しているかを求める．

評価用の Java 実行環境には JeanPaul[17)~20)] を用いた．JeanPaul は ANSI C に準拠した C ソースを出力する Java2C トランスレータと C コンパイラからなる静的 Java コンパイラを持つが，評価のため，Java2C トランスレータに暗黙の処理を強制削除する機能を追加した．追加した機能は，Java2C トランスレータにコンパイルオプションを通じて指示することにより動作する．機能の追加にあたっては，評価結果に暗黙の処理とは無関係の要因が影響することを防ぐため，暗黙の処理に直接関与しない最適化（インライン展開など）の挙動が強制削除を実施するか否かによって影響を受けないよう配慮した．静的 Java コンパイラのバックエンドである C コンパイラは Visual C++®☆ 6.0 とし，コンパイルオプションは/O2（実行速度高速化）とした．

評価対象の Java アプリケーションは SPECjvm98 とした[11)]．SPECjvm98 は `javac` など中〜小規模の実用的 Java アプリケーションからなるベンチマークである．コンパイル対象のメソッドは次の手順で定めた．まず，SPECjvm98 の各ベンチマークを `-verbose` オプション☆☆付きで実行した結果からベンチマークの終了までにロードした全クラスを求め，次に，求めたクラスが定義するメソッドのうち，SPECjvm98 の `main()` メソッドを定義する `SpecApplication` クラス中のメソッドからコールツリーを経由して到達しうるすべてのメソッドをコンパイル対象とした．ただし，JeanPaul の Java2C トランスレータではコンパイル対象に指定したメソッドのうち，インライン展開によってすべての呼び出し元が消失したメソッドについては C ソースを生成しない．

SPECjvm98 の実行にあたっては，問題サイズを 100 とし，ヒープサイズは初期値，最大値ともに 20 MByte とした．ベンチマークは PC（論文中に明記しない限り，**表 2** の PC2）で実行した．

表 2 評価に使用した PC の構成
Table 2 Specifications of PCs for evaluation.

	PC1	PC2
プロセッサ	Am5x86	PentiumⅢ× 2
動作周波数	133 MHz	450 MHz
一次キャッシュ	16 KByte	32 KByte
二次キャッシュ	なし	512 KByte
主記憶	36 MByte	256 MByte
OS	Windows95	Windows2000

本論文の 2 章から 5 章では，表 1 に示す個々の暗黙の処理について，その Java2C トランスレータにおける実現と最適化方法について詳述し，それぞれの処理を実現するためのコードが Java2C トランスレータが通常出力するコードサイズ全体のどれだけの割合を占め，実行時間全体のどれだけの割合を消費するか評価する．6 章では暗黙の処理がコードサイズや実行時間に与える影響について総合的に評価し，7 章で関連研究を示す．8 章は結論である．

2. null 検 査

null 検査は変数が指示するオブジェクトを参照する際に，変数の値が null か検査し，null であれば実行時例外 `NullPointerException` を投擲する．null 検査には，暗黙の null 検査というプラットホームに依存するが効率的な実現方法がある[12),14),15)]．この方法では null 検査のためのコードを明示的には生成しない．そのかわり，0 番ページ（アドレス 0 から始まるページ）に保護をかけておく．すると，変数の値が null であるときにオブジェクト参照を実行しようとすると，ページトラップがおきるので，これを捕捉して `NullPointerException` を生成し，ページトラップを起こした時点でのプログラムカウンタの値から例外を発生した文を算出し，算出した文に対応するハンドラを求めてジャンプする．

可搬性を重視する Java2C トランスレータでは，次に示す 2 つの理由から暗黙の null 検査を使えない．

- C 言語では個々の文を実現する機械コードのアドレスを得ることができない．したがってページトラップを起こした時点でのプログラムカウンから，例外を発生した文を算出できない．
- プラットホームによってはページ保護を使えない．

そこで，JeanPaul の Java2C トランスレータでは null 検査を明示的に実現する．たとえば，**図 1** 上段の Java ソースには 2 行目にインスタンスフィールド参照があるが，インスタンスフィールド参照を実施する前に，変数 object の中身が null か検査する必要がある．図 1

☆ Windows および Visual C++は米国 Microsoft Corporation の米国およびその他の国における登録商標です．
☆☆ どのクラスをロードしたかログを出力するオプション．

```
0:    /* Java ソース */
1:    while(i<100){
2:      i += object.field;
3:    }
4:    /* C ソース（ピーリング前） */
5:    while(i<100){
6:      if (object == NULL){
7:        /* 例外の生成 */
8:        jp_ThrowNullPointerException(ee);
9:        /* ハンドラへジャンプ */
10:       goto HANDLER;
11:     }
12:     i += object->field;
13:   }
14:   /* C ソース（ピーリング後） */
15:   if (i<100){
16:     if (object == NULL){
17:     THROW:
18:       jp_ThrowNullPointerException(ee);
19:       goto HANDLER;
20:     }
21:     else{
22:       i += object->field;
23:       while(i<100){
24:         if (object == NULL){
25:           goto THROW;
26:         }
27:         i += object->field
28:       }
29:     }
30:   }
```

図 1 null 検査を含むコード
Fig. 1 Code with null test.

上段の Java ソースを C ソースに変換すると図 1 中段のコードになるが，その 6〜11 行目までが明示的な null 検査のコードである．

null 検査を明示的に実現すると，コードサイズが増え，また，実行速度が遅くなる．null 検査はバイトコードの 26.0%，おおむね 4 命令に 1 命令から生成されるほど数多く存在し[20]，影響を無視できない．JeanPaul の Java2C トランスレータでは null 検査のオーバヘッドを削減するため，次の最適化を実施する．

冗長な null 検査の除去 変数の値が null になりうるかフロー解析によって調査し，なりえない変数を対象とした null 検査を冗長と見なして除去する[15]．

ループのピーリング フロー解析による冗長な null 検査の除去がかかりやすくなるように，ループの本体を 1 回分展開する[1]．たとえば図 1 中段の C ソースではループ内に null 検査が存在し，実行速度を大きく低下させる．そこで図 1 中段のコードにループのピーリングを適用すると図 1 下段のコードになる．図 1 下段の 24〜26 行目にある null 検査は，16 行目ですでに `object` が null か検査していることから，フロー解析で冗長と判断して除去できる．これにより，ループ内から null 検査を除去して実行速度を改善できる．ただし，ループをピーリングするとコードサイズは大きくなる．コードサイズへの配慮から，JeanPaul の Java2C トランスレータでピーリングの対象とするのは，ピーリングで除去可能な null 検査を含み，かつ，インタプリタ呼び出し（実行に長い時間がかかると予測できる処理）を含まず，内包する基本ブロック数が 20 個以下の最内ループである．

例外発生コードの下方集約 例外を発生するコード（図 1 中段の 7〜10 行目のコード）を，例外発生時に同じハンドラにジャンプする null 検査間で共有し，コードサイズを削減する[8]．たとえば図 1 下段のコード中にある 2 つの null 検査（16〜20 行目と，24〜26 行目）は例外を発生するコードを共有し，`jp_ThrowNullPointerException(ee)` を 1 カ所に集約することでコードサイズを削減する．

最適化で除去できない null 検査のオーバヘッドがどれだけ残っているか評価するために，最適化で冗長な null 検査を削除した場合と，すべての null 検査を強制削除した場合のコードサイズと実行時間の比較を **表 3**，**図 2** それぞれの `Null` の項目に示す．表 3 および図 2 において，「Java2C（トランスレータ）が通常出力するコード」では，どの暗黙の処理についても強制削除はせず，ただ最適化で冗長と判断できた暗黙の処理のコードのみ削除した．また，表 3 においてコードサイズとはオブジェクトファイルの総コード長（オブジェクトファイル中の機械コードを収める部分である `.text` セクションの長さの合計）を意味する．本論文において平均とは相乗平均を意味する．

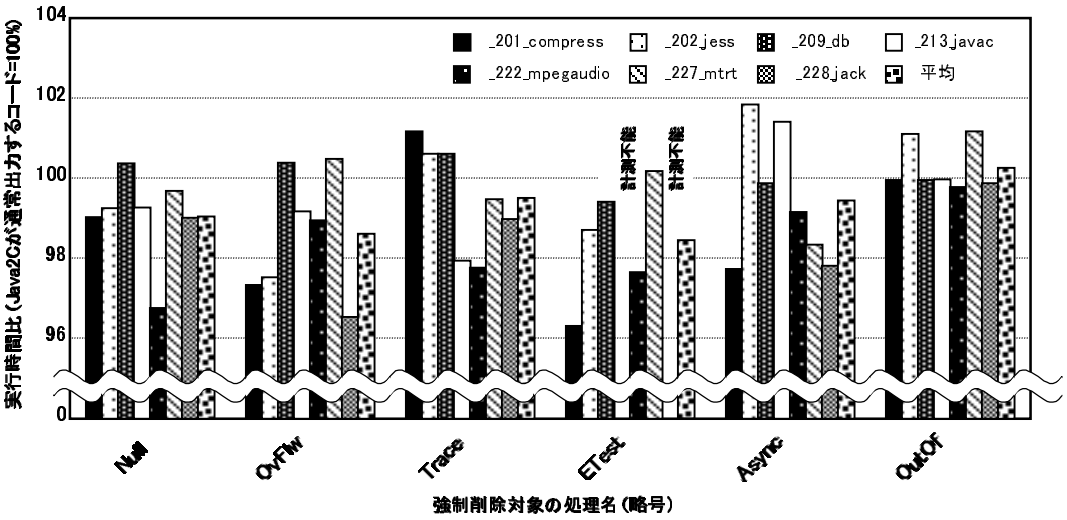
表 3 および図 2 から，Java2C トランスレータが通常出力するコードに比べ，すべての null 検査を強制削除すると，平均でコードサイズが 9.73%小さくなり，実行時間が 0.96%短くなることが分かる．このことは，null 検査を実現するためのコードが平均で Java2C トランスレータが通常出力するコードサイズ全体の 9.73%を占め，実行時間全体の 0.96% を消費することを意味する．コードサイズが実行時間より大きな影響を受けている原因の 1 つは，Java2C トランスレータがコードサイズの増加を代償に実行を高速化する最適化であるループのピーリングを実施していることにある．ループのピーリングを抑止すれば，コードサイズの増加を抑止できるが実行速度は遅くなる．

なお，コードサイズの増加は null 検査を明示的に実現した結果だが，暗黙の null 検査を利用できたと

表 3 暗黙の処理向けコードの強制削除がコードサイズにもたらす影響
Table 3 Effect of compulsory removal of codes for implicit features on code size.

ベンチマーク項目名	Java2C が通常出力するコード	コンパイルオプションの指定により強制削除する処理名（略号）						
		Null	OvFlw	Trace	ETest			全削除
					全部	Async	OutOf	
_201_compress	127,904	116,736 (8.73%)	124,048 (3.01%)	112,128 (12.33%)	108,928 (16.40%)	126,544 (1.06%)	125,904 (1.56%)	76,368 (40.29%)
_202_jess	353,152	307,872 (12.82%)	341,984 (3.16%)	307,920 (12.81%)	297,296 (15.82%)	351,360 (0.51%)	345,968 (2.03%)	199,440 (43.53%)
_209_db	148,192	135,728 (8.41%)	143,920 (2.88%)	128,464 (13.31%)	122,992 (17.00%)	146,544 (1.11%)	145,280 (1.97%)	86,160 (41.86%)
_213_javac	572,128	517,168 (9.61%)	553,040 (3.34%)	498,480 (12.87%)	491,344 (14.12%)	569,360 (0.48%)	563,408 (1.52%)	345,264 (39.65%)
_222_mpegaudio	221,984	197,024 (11.24%)	215,936 (2.72%)	196,512 (11.47%)	193,312 (12.92%)	220,320 (0.75%)	218,736 (1.46%)	138,256 (37.72%)
_227_mtrt	207,328	190,240 (8.24%)	202,144 (2.50%)	185,152 (10.70%)	173,024 (16.55%)	205,840 (0.72%)	204,384 (1.42%)	127,856 (38.33%)
_228_jack	284,272	258,768 (8.97%)	279,856 (1.55%)	252,160 (11.30%)	236,704 (16.73%)	283,008 (0.44%)	279,600 (1.64%)	174,608 (38.58%)
平均		– (9.73%)	– (2.74%)	– (12.12%)	– (15.66%)	– (0.73%)	– (1.66%)	– (40.03%)

※処理名の略号の意味は表 1 に準ずる．
※数値はコードサイズ，単位はバイト．括弧内の数値は Java2C トランスレータが通常出力するコード全体に対して，強制削除で除去したコードサイズ（Java2C トランスレータが通常出力するコードサイズと強制削除適用後のサイズの差分）が占める割合．



※ETestの平均は計測不能項目を除いた平均

図 2 暗黙の処理向けコードの強制削除が実行時間にもたらす影響
Fig. 2 Effect of compulsory removal of codes for implicit features on execution time.

しても，null 検査を強制削除する場合と同じコードサイズや実行速度が得られるわけではない．なぜなら暗黙の null 検査では強制削除する場合に比べフロー最適化など一部の最適化に制限がかかるためで，強制削除する場合より若干，コードサイズが大きく，実行速度が遅くなる[15]．

3. スタックオーバフロー検査

Java 言語仕様ではセキュリティの確保などを目的として，実行時スタックがオーバフローした場合に `StackOverflowError` を生成，投擲するよう定めている．スタックオーバフローの検出方法の 1 つに，ページトラップを使う方法がある．この方法では，スタックの上端にあるページに保護をかける．すると，オーバフロー時には，スレッドが保護したページを参照してページトラップを起こすので，これを捕捉することでオーバフローを検出する．

ページトラップを使う方法では実行速度やコードサ

```
 0:   boolean jpCheckStack(ExecEnv *ee){
 1:     boolean result =
 2:       (&result > ee->stack_upper_limit);
 3:     if (result){
 4:       /* StackOverflowError の発生 */
 5:       jpThrowStackOverflowError(ee);
 6:     }
 7:     return (result);
 8:   }
                          :
                          :
 9:   void boo(ExecEnv *ee){
10:     if (jpCheckStack(ee)){
11:       /* 対応するハンドラまでジャンプ */
12:       goto HANDLER;
13:     }
14:     /* 関数本体 */
                          :
                          :
15:   }
```

図 3 スタックオーバフロー検査
Fig. 3 Stack overflow test.

イズにほとんど悪影響を与えることなくスタックオーバフローを検出できるが，反面，プラットホームにページ保護の機能がある環境でないと使えない．JeanPaul の Java2C トランスレータでは可搬性を重視し，ページトラップを使う方法を採用せず，かわりに個々の関数の冒頭にスタックオーバフローの検出を試みるコード（**図 3** の 10〜13 行目）を挿入する☆．挿入するコードは 10 行目で関数 `jpCheckStack()`（0〜8 行目）を呼び出し，呼び出した関数内の 2 行目でスタック上に確保した変数 `result` のアドレスとスタックの上限 `ee->stack_upper_limit` を比較してスタックオーバフローが発生していないか検査する．

コードを挿入すると実行速度やコードサイズに影響が出る．表 3 と図 2 の `OvFlw` の項目に示した評価結果から，スタックオーバフローを検出するためのコードが，平均で Java2C トランスレータが通常出力するコードサイズ全体の 2.74%を占め，実行時間全体の 1.39%を消費することが分かる．

4. スタックトレース情報の提供

Java では実行時スタックのトレース情報をオブジェクトとして取得するメソッド `getClassContext()` を標準ライブラリクラス `java.lang.SecurityManager` で提供する．このメソッドを実装するため，Java 実行環境はスタックトレース情報を算出する手段を確保する必要がある．

スタックトレースを求める方法の 1 つに，リターンアドレスから呼び出し元の関数名を逆算する方法がある．この方法では，次の手順でスタック上にあるスタックフレームを上から順にたどってスタックトレースを算出する☆☆．

(1) スタックトレースを計算する関数を呼び出すと，関数はまず，受け取ったリターンアドレスから呼び出し元の関数名とスタックフレームサイズを求める．この計算には，リターンアドレスと呼び出し元の関数名およびフレームサイズを対応づける表が必要だが，この表はコンパイル，リンク時に生成しておく．インライン展開を使って最適化したオブジェクトについてスタックトレース情報を算出する場合は，どこにインライン展開を施したかという情報も必要になる．
(2) 求めた関数名を記録する．スタックポインタからスタックフレームサイズを減じて，直下にあるスタックフレームのアドレスを求め，その中に記録してあるリターンアドレスを求める．
(3) リターンアドレスが `NULL`（スタックの底に達した）ならば終了．さもなくば，リターンアドレスから呼び出し元の関数名とスタックフレームサイズを求め，(2) に戻る．

リターンアドレスから呼び出し元の関数名を逆算する方法の利点は，実行速度やコードサイズに悪影響を与えないことである．なぜなら，この方法でスタックトレースを計算するにあたって必要な情報はリターンアドレスだけだが，リターンアドレスは関数呼び出しのために元々必要になるものであり，スタックトレースのために手間をかけて用意するわけではないからである．逆算のための情報を収める表の大きさだけオブジェクトサイズは大きくなるが，コードサイズは増えないから，命令キャッシュを圧迫してキャッシュミス率を増やすことはない．このことはキャッシュが貧弱な組込み機器では大きな意味を持ちうる．

C 言語はスタックトレースを計算する手段を提供しないが，C コンパイラによってはデバッグ情報の提供などを目的として，リターンアドレスからスタックトレースを逆算するために必要な情報をオブジェクトファイル中に出力するものもある．その場合，スタックトレースを計算するライブラリ関数の実装がプラットホーム依存になるものの，リターンアドレスから C の関数のスタックトレースを逆算できる．

☆ 本論文では簡単のため，図 3 以外の C ソースではオーバフロー検査の記述を省略する．

☆☆ 動作の詳細は関数の呼び出し規約などにより若干変わる．

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

図 4 スタックトレース管理マクロ
Fig. 4 Macros for keeping stack trace.

しかしながら，この方法では，Java2C トランスレータが生成する C ソースにおけるスタックトレースを計算できても，変換元の Java ソースにおけるスタックトレースは計算できない．なぜなら，この方法では Java2C トランスレータで適用したインライン展開を考慮できず，その情報が欠落するからである．このため Java2C トランスレータでは，少なくともインライン展開を適用した部分について，スタックトレースを計算する何らかの手段を用意する必要がある．

JeanPaul の実装にあたっては可搬性を重視したため，リターンアドレスからスタックトレースを逆算する方法をいっさい採用せず，スタックトレース情報を記録するコードを明示的に出力することにした．スタックトレース情報に関する型とマクロの定義を**図 4** に示す．スタックトレース情報はリスト構造をとる．Java2C トランスレータは Java のメソッドを C の関数に変換する際に，C の関数の先頭でスタックトレース情報を収めるセル（図 4 の 0〜3 行目で定義する型の構造体）の領域を宣言し，さらにマクロ `PushSingleFrame()`（4〜8 行目）を挿入して実行中のメソッドをセルに記録し，スタックトレース情報リストの先頭に連結する．また，マクロ `PopSingleFrame()`（9〜11 行目）を `return` 文の前に挿入して関数から返戻する際にセルをリストから外す．スタックトレースを計算するには，マクロ `PushSingleFrame()` で変数 `ee` が参照するスレッド固有の資源を収める構造体に記録したリストの先頭セルへのポインタから，順次セルをたどる．

明示的にスタックトレースを管理する方法は実行速度的にもコードサイズ的にもオーバヘッドが大きい．そこで JeanPaul の Java2C トランスレータではスタックトレース情報を管理するコードに次の最適化を適用してオーバヘッドの削減を試みる．

4.1 冗長なコードの除去

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び出し元にならないメソッドでは管理する必要がない．そこでメソッドがスタックトレースを計算するメソッドの呼び出し元になるかメソッド間にまたがって解析し，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．

4.2 補償コード領域への移動

図 5 上段の Java ソースの 5〜7 行目にあるメソッド `val()` の内部に仮想呼び出し `this.val0()` があるが，この仮想呼び出しに I-call if 変換[2),4),19)] を適用しつつ，`val()` を C ソースに変換すると図 5 中段のコードになる．16〜21 行目が I-call if 変換後の仮想呼び出しだが，ここで 20 行目の間接呼び出しは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 `this` が呼び出すときのみ使う．コンパイル時の予測が外れる可能性は高くないため，20 行目の間接呼び出しを使う機会は少ない．このように使用頻度は低いが万一に備えて挿入するコードを補償コードという．

さて，図 5 中段のコードでは 15 行目と 22 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 20 行目の間接呼び出しの呼び出し先が不定で，その呼び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．16 行目の if 文が 17 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 20 行目の間接呼び出しの前後に移動した方がよい．

そこで JeanPaul の Java2C トランスレータには，`PushSingleFrame()` と `PopSingleFrame()` が囲う両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び出しうる

```
 0:  /* Java ソース */
 1:  class C{
 2:    int val0(){
 3:      return (this.value);
 4:    }
 5:    int val(){
 6:      return (this.val0());
 7:    }
 8:  }
 9:  /* C ソース（最適化前） */
10:  int C_val(ExecEnv *ee, Object *this){
11:    struct singleFrame frame;
12:    int result;
13:    int (*code)(ExecEnv *, Object *)
14:      = this->dispatch_table[C_val0_offset];
15:    PushSingleFrame(ee, frame, C_val_id);
16:    if (code == C_val0){
17:      result = this->value;
18:    }else{
19:      /* 補償コード領域 */
20:      result = code(ee, this);
21:    }
22:    PopSingleFrame(ee, frame);
23:    return (result);
24:  }
25:  /* C ソース（最適化後） */
26:  int C_val(ExecEnv *ee, Object *this){
27:    struct singleFrame frame;
28:    int result;
29:    int (*code)(ExecEnv *, Object *)
30:      = this->dispatch_table[C_val0_offset];
31:    if (code == C_val0){
32:      result = this->value;
33:    }else{
34:      /* 補償コード領域 */
35:      PushSingleFrame(ee, frame, C_val_id);
36:      result = code(ee, this);
37:      PopSingleFrame(ee, frame);
38:    }
39:    return (result);
40:  }
```

図 5 補償コード領域への移動
Fig. 5 Immigration into complementary code area.

場合に，PushSingleFrame() と PopSingleFrame() を補償コードの前後に移動する最適化を実装した．この最適化を図 5 中段のコードに適用すると図 5 下段のコードになる．下段のコードではコンパイル時に予測したメソッドを呼ぶ場合にはスタックトレース情報の積下しを省略でき，実行を高速化できる．

4.3 コード自体の高速化

リスト構造をとるスタックトレース情報の積下しにかかるコストは小さくない．そこで，インライン展開したメソッドについては，より高速な方法でスタックトレース情報を管理する．具体的には，まず，インライン展開を適用したメソッドでは図 4 の 12〜17 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，メソッド内での呼び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 18〜21 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()（25〜28 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 29〜30 行目のマクロ PopMultiFrame() で元に戻す．たとえば**図 6** 上段の Java ソースは中にあるメソッド呼び出し this.m2() に Java2C トランスレータでインライン展開を適用しつつ C ソースに変換すると，図 6 中段のコードになる．

トレース情報リストに複数の種類のセルを接続すると，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 4 のマクロでは，2 番目のフィールドの第 0 ビットを使って判別することを可能にしている．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 トランスレータがこの番号を付けるにあたって下ビットを 0 にすることで，2 番目のフィールドの第 0 ビットからフレームの種別を判別可能になる．なお，Java2C トランスレータがマクロ PushMultiFrame()，PopMultiFrame() に与える引数 depth の値は，図 6 中段のコードの 14 行目や 20 行目にあるように定数であり，C コンパイル時にマクロを静式評価して定数ストアに還元できるので，マクロ中のシフトや論理和の計算を実行時に行うことはない．

インライン展開を適用したメソッド用トレースマクロ PushMultiFrame()，SwapMultiFrame()，PopMultiFrame() は，リストへのリンクやアンリンクをする PushSingleFrame() や PopSingleFrame() より実行コストが小さいが，それだけではなく，最適化の適用対象とすることでさらに実行コストを軽減できる．たとえばマクロ PushMultiFrame() は概念的にはスタックの Push だが，実際に実行することはセル内の整数メンバ sp の更新なので，もしこのマクロが 2 つ連続して存在するならば，その前者を省略できる．Jean-Paul の Java2C トランスレータには，そういった冗長なメソッド内トレースマクロを検出して削除するデー

```
 0:  /* Java ソース */
 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(){ ··· }
 9:  /* C ソース（最適化前） */
10:  Object *m1(ExecEnv *ee, Object *this){
11:    DeclareMultiFrame(frame, 2);
12:    Object *result; int i, itemp;
13:    LinkMultiFrame(ee, frame);
14:    PushMultiFrame(frame, 0);
15:    SwapMultiFrame(frame, 0, m1_id);
16:    result = AllocIntArray(ee, 10);
17:    if (exceptionOccurred(ee))
18:      goto RETURN;
19:    for(i=0; i<10; i++){
20:      PushMultiFrame(frame, 1);
21:      SwapMultiFrame(frame, 1, m2_id);
22:      { /* m2 の本体 */
23:        itemp = m3(ee, this);
24:        if (exceptionOccurred(ee)){
25:          PopMultiFrame(frame, 1);
26:          goto RETURN;
27:      }}
28:      PopMultiFrame(frame, 1);
29:      result[i] = itemp;
30:    }
31:  RETURN:
32:    PopMultiFrame(frame, 0);
33:    UnlinkMultiFrame(ee, frame);
34:    return (result);
35:  }
36:  /* C ソース（最適化後） */
37:  void m1(ExecEnv *ee, Object *this){
38:    DeclareMultiFrame(2);
39:    Object *result; int i, itemp;
40:    LinkMultiFrame(ee, frame);
41:    SwapMultiFrame(frame, 0, m1_id);
42:    result = AllocIntArray(ee, 10);
43:    PushMultiFrame(frame, 1);
44:    SwapMultiFrame(frame, 1, m2_id);
45:    for(i=0; i<10; i++){
46:      /* m2 の本体 */
47:      itemp = m3(ee, this);
48:      if (exceptionOccurred(ee))
49:        goto RETURN;
50:      result[i] = itemp;
51:    }
52:  RETURN:
53:    UnlinkMultiFrame(ee, frame);
54:    return (result);
55:  }
```

図 6 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
Fig. 6 Implementation of stack trace in a method that includes an inlined call.

タフロー最適化を実装してある．なお，図 4 で Push の動作を PushMultiFrame() と SwapMultiFrame() に分けたのは，データフロー最適化において，その一方だけが除去可能な場合があるからである．また，JeanPaul の Java2C トランスレータには，可能ならばメソッド内トレースマクロをループ外に移動する最適化も実装してある．これらの最適化を図 6 中段のコードに適用すると図 6 下段のコードになる．

図 6 中段のコードと下段のコードを比較すると，最適化の結果として中段のコードの 14，25，28，32 行目にあったマクロが消失し，20，21 行目にあったマクロがループ外に移動していることが分かる．

14 行目の PushMultiFrame() が消失するのは，後続する 20 行目の PushMultiFrame() あるいは 32 行目の PopMultiFrame() で sp を更新するまでの間にスタックトレース情報を取得しうるメソッド呼び出しが存在しないため，14 行目において PushMultiFrame() を実施して sp に値を与えることに意味がないからである．14 行目に後続する 16 行目には関数呼び出し 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起きるが，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ている．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☆，組込み機器向け Java 実行環境である 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 14 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．

25 行目の PopMultiFrame() が消失するのは後続する 32 行目で sp を更新するまでの間にスタックトレースを取得しうる関数呼び出しが存在しないためで，同様の理由から 28 行目のマクロも消失する．32 行目の PopMultiFrame() が消失するのは直後の 33 行目で frame を捨てるので，32 行目で sp を更新することに意味がないからである．

20 行目と 21 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 コンパイラに実装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

☆ 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．

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とデータフロー最適化を開発した．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 20 行目と 21 行目の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 23 行目にある関数呼び出しで `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び出しの結果として `frame` の内容が変わりうると推定して 20 行目と 21 行目のマクロを不変式と見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 14 行目の `PushMultiFrame()` を削除できない．

最適化で除去しきれないスタックトレース管理コードのオーバヘッドがどの程度あるか，管理コードを強制削除した場合との比較によって評価した結果を，表 3 と図 2 の `Trace` の項目に示す．表 3 および図 2 から，管理コードが，平均で Java2C トランスレータが通常出力するコードサイズ全体の 12.12%を占め，実行時間全体の 0.51%を消費することが分かる．

5. 例外処理のサポート

例外処理とは，ファイルの読み書きに失敗したなど，例外的事象が発生した場合に，その事象に対処するルーチン（ハンドラ）まで大域的にジャンプする機能である．Java では例外処理を `try`，`catch`，`throw` というキーワードを使って記述する．個々のキーワードの意味について，**図 7** 上段の Java ソースを例に具体的に述べる．メソッド `foo()` の 3 行目にある `try` は，後続する中括弧で括った `try` ブロックと呼ぶ領域（3〜7 行目）の実行中に例外が発生した場合，`try` ブロックの直後の 8 行目にある `catch` で例外を捕捉し，処理を試みることを宣言する．8 行目の `catch` は，投擲された例外オブジェクトが `Exception` クラスのインスタンスである場合に捕捉して，その参照を変数 `e` に納め，9 行目の「ハンドラ」とコメントを打ってある部分に進んで例外を処理する．`foo()` を実行すると，まず 3 行目で `try` ブロックに突入し，5 行目でメソッド `woo()` を呼び出す．`woo()` では 13 行目で `Exception` クラスの例外オブジェクトを生成し，投擲（`throw`）する．例外を投擲すると，`woo()` 内部には例外を捕捉する `catch` がないので呼び出し元の `foo()` に戻り，8 行目の `catch` で例外を捕捉する．

```
 0:  /* Java ソース */
 1:  void foo(){
 2:    int result = 0;
 3:    try{
 4:      while(true){
 5:        this.woo();
 6:      }
 7:    }
 8:    catch(Exception e){
 9:      // ハンドラ
10:    }
11:  }
12:  void woo() throws Exception{
13:    throw (new Exception());
14:  }
15:  /* C ソース（最適化前） */
16:  void foo(ExecEnv *ee, Object *this){
17:    Object *atemp;
18:    while(true){
19:      woo(ee, this);
20:      if (exceptionOccurred(ee))
21:        goto CATCH;
22:      /* 非同期例外の検出 */
23:      if (exceptionOccurred(ee))
24:        goto CATCH;
25:    }
26:  CATCH:
27:    atmp = catch(ee, Exception);
28:  }
29:  int woo(ExecEnv *ee, Object *this){
30:    Object *atemp = allocObject(Exception);
31:    ExceptionConstructor(ee, atemp);
32:    exceptionThrow(ee, atemp);
33:  }
```

図 7 2 返戻値法による例外処理の実現
Fig. 7 Exception handling by two-return-values method.

例外処理の実現方法には大別して表引き法[8]，`setjmp` 法[3),17)]，2 返戻値法[7),16),17),20)] の 3 種類がある．表引き法では次の手順で `throw` から `catch` までジャンプする．

(1) 例外を投擲する関数 `throw` を呼び出すと，`throw` はまず，受け取ったリターンアドレスを参照する．
(2) リターンアドレスから呼び出し元のソース上の位置を特定し，対応する `catch` があるか調べる（リターンアドレスと対応する `catch` を関連付ける表をコンパイル，リンク時に用意しておく）．あるならば，`catch` が投擲された例外オブジェクトを捕捉するか検査する．捕捉するならば `catch` に後続するハンドラに制御を移し，ジャンプ完了．そうでなければ，(3) に進んで更なる呼び出し元の `catch` を探索する．
(3) 呼び出し元のスタックフレームのアドレスを求める．これは，リターンアドレスからスタック

フレーム長を逆算し，求めたスタックフレーム長をスタックポインタから減じることで求められる．

(4) 呼び出し元のスタックフレームからリターンアドレスを求める．リターンアドレスが NULL だったら，つまりスタックの底に到達していたら探索完了．この場合，例外を捕捉するハンドラがなかったことになる．さもなくば，(2) に戻って探索を継続する．

表引き法は実行コード中に例外処理を実現するためのコードを挿入する必要がなく，3 種類の実現方法のうちで発生するオーバヘッドが最も小さい．しかし，Java2C 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ない．その理由はいくつかあるが，たとえば，表引き法の実現には関数呼び出しのリターンアドレスから対応するハンドラを特定する動作などが必要になるが，C 言語では関数呼び出しのリターンアドレスを計算できないといった点があげられる．

JeanPaul では Java2C トランスレータで使える setjmp 法と 2 返戻値法のうち，2 返戻値法を使って例外処理を実現する[17]．2 返戻値法では，関数は 2 つの値を返戻する．その一方は，通常の返戻値であり，もう一方は例外発生の有無を表すフラグ（例外発生フラグ）である．そして，個々の関数呼び出しの直後に，例外発生フラグを検査し，例外が発生したならば対応するハンドラにジャンプする文を挿入する（対応するハンドラがない場合には関数から返戻する）．挿入する文を例外発生検査と呼ぶことにする．2 返戻値法では，例外を投擲した時点から順次，関数から返戻しながら対応するハンドラを探すことで例外処理を実現する．

JeanPaul の Java2C トランスレータで図 7 上段の Java ソースを C ソースに変換した結果を，図 7 下段に示す．図 7 下段の 19 行目に関数 woo() の呼び出しがあり，呼び出しを実行して例外が発生すると，その直後の 20〜21 行目にある例外発生検査でマクロ exceptionOccurred() を使って ee が参照するスレッド固有の資源を収める構造体の中にある例外発生フラグを参照して例外の発生を検知し，26 行目の CATCH にジャンプする．

2 返戻値法では例外処理を実現するために，例外発生検査のコードをプログラム中の複数の箇所に挿入するが，その結果として，コードサイズや実行速度に悪影響が出る．JeanPaul では例外発生検査によるオーバヘッドを軽減するために，冗長な例外発生検査を除去する最適化や，複数の例外発生検査を 1 カ所に集約する最適化などを適用する（詳細については参考文献 20) を参照されたい）．最適化後にどの程度のオーバヘッドが残っているか，最適化のみによってオーバヘッドを軽減した場合と，例外発生検査をすべて強制除去した場合を比較して評価した結果を表 3 と図 2 の ETest の項目に示す．

表 4 例外発生検査の箇所数と推定コードサイズ
Table 4 Count and guessed code size of exception tests.

ベンチマーク	例外発生検査	
項目名	箇所数	推定コードサイズ
_201_compress	1,748	19,228(15.03%)
_202_jess	5,174	56,914(16.12%)
_209_db	2,111	23,221(15.67%)
_213_javac	7,193	79,123(13.83%)
_222_mpegaudio	2,529	27,819(12.53%)
_227_mtrt	2,894	31,834(15.35%)
_228_jack	4,008	44,088(15.51%)
平均		- (14.87%)

※括弧内は表 3 の「最適化のみで削除」から例外発生検査のみ削除した場合の推定コード削減率

表 3 から例外発生検査を強制削除するとコードサイズが平均で 15.66%小さくなることが分かるが，余分にコードを挿入することがない表引き法を使って例外処理を実現できたしても，これほどにはコードサイズを小さくできない．なぜなら例外発生検査をすべて強制削除すると，例外発生検査からの分岐によってしか到達できないハンドラの部分まで一緒に削除してしまうからである．もっともハンドラのコードサイズは大きくはない．コード中にある例外発生検査のコードサイズの合計を，例外発生検査の 1 カ所あたりのコードサイズを 11 バイトとして☆，プログラム中の例外発生検査の箇所数と積算して推定した結果を**表 4** に示す．表 4 から，例外発生検査が平均で Java2C トランスレータが通常出力するコード全体の 14.87%を占め，したがって例外発生検査の強制削除にあたって一緒に削除するハンドラのサイズは $15.66 - 14.87 = 0.79\%$ 程度であると推測できる．

図 2 の実行時間の測定結果について，_213_javac と _228_jack の 2 項目が測定不能となっているのは，これらの項目が実行中に例外を投擲するため，例外発生検査を強制削除するとベンチマークが正常に動作しないためである．測定不能の 2 項目を除いて求めた平均から，例外発生検査の実行にかかる時間が，Java2C トランスレータが通常出力するコードによるベンチマーク実行時間全体の 1.57%を占めることが分かる．

☆ ジャンプのオフセット長や ee がレジスタ上にあるか否かで，7, 10, 11, 14 バイトのいずれかになるが，11 バイトであることが多い．

5.1 非同期例外のサポート

JeanPaul の Java2C トランスレータは，Java の非同期例外をサポートするため，関数呼び出しの直後以外にも例外発生検査を挿入する．Java では `java.lang.Thread.stop()` というメソッドを使い，別のスレッドに非同期的に例外を発生するよう指示できる．このメソッドにより発生する例外を非同期例外と呼ぶ．非同期例外を発生するよう指示を受けたスレッドは，短い期間内に指定の例外を投擲して例外処理に制御を移す必要がある．JeanPaul では例外発生検査によって非同期的に発生する例外を検出するが，例外発生検査の挿入箇所が関数呼び出しの直後だけだと，例外発生検査を含まないループができて，最悪の場合，実行時に発生した非同期例外を検出できない無限ループに陥る．そこで，JeanPaul の Java2C トランスレータは関数呼び出しの直後以外にも，負方向への分岐の直前にも例外発生検査を挿入し，ループ本体を実行する際に最低 1 回は例外発生検査を実施することを保証する．

たとえば図 7 下段の 23〜24 行目には，非同期例外を検出するために挿入した例外発生検査がある．もっとも，この例外発生検査は直前の 5〜6 行目に 4 行目にある関数呼び出しの直後に挿入した例外発生検査があるため冗長である．非同期例外を検出するために挿入した例外発生検査は，ループ本体実行時に必ず別の例外発生検査を実行するならば冗長になるが，JeanPaul の Java2C トランスレータには，冗長な非同期例外向け例外発生検査を削除する最適化を実装してある☆．

最適化で削除しきれなかった非同期例外検出用の例外発生検査が生じるオーバヘッドを，すべての非同期例外検出用の例外発生検査を強制削除した場合と比較して評価した結果を表 3 と図 2 の `Async` の項目に示す．なお，`ETest` の項目は `Async` を含めすべての例外発生検査を削除した結果である．表 3 と図 2 から，非同期例外向け例外発生検査のコードが平均で Java2C トランスレータが通常出力するコードサイズ全体の 0.73%を占め，実行時間全体の 0.57%を消費することが分かる．

5.2 `OutOfMemoryError` のサポート

例外発生検査の中には，最適化では除去できないが，アプリケーションの実行条件によっては冗長になるものもある．`OutOfMemoryError` はヒープが不足した場合に発生する例外で，JeanPaul の Java2C トランスレータは `OutOfMemoryError` に備えてオブジェクトを割り付ける関数呼び出しの直後に例外発生検査を挿入する．しかし，ユーザが実行時に十分なヒープを与え，`OutOfMemoryError` が発生しないことを保証する場合には，この例外発生検査は冗長になる．

`OutOfMemoryError` 向け例外発生検査の強制削除によって実行時間やコードサイズが受ける影響を評価した結果を表 3 と図 2 の `OutOf` の項目に示す．表 3 から，`OutOfMemoryError` 向け例外発生検査のコードが，平均で Java2C トランスレータが出力するコードサイズ全体の 1.66%を占めることが分かる．また図 2 から，強制削除を実行しても実行時間にあまり影響がでないことが分かる．これはオブジェクト割付けのコストと比較して，例外発生検査コストが小さいためである．平均では実行時間が 0.25%余計にかかっているが，これは測定誤差の範囲内と考える．

6. 総合評価

2 章から 5 章では，表 1 に示した暗黙の処理について，それぞれ強制削除した場合に生じる影響を個別に評価した．本章では，すべての暗黙の処理を強制削除した場合の影響について考える．表 3 の全排除の項目に，すべての暗黙の処理を強制削除した場合のコードサイズを示す．平均から，すべての暗黙の処理を強制削除するとコードサイズが 40.03%小さくなる，換言すれば Java2C トランスレータが通常出力するコードにおいて，暗黙の処理がコードサイズ全体の 40.03%を占めることが分かる．この値は，2 章から 5 章で個別に強制削除して評価した個々の暗黙の処理がコードサイズ全体の中で占める割合の合計にほぼ一致し（`ETest` を全部削除する場合，`Async` と `OutOf` も削除対象になる），暗黙の処理の間に相互干渉があまり生じていないと推測できる．また，7 つのベンチマーク項目を通じて暗黙の処理がコードサイズ全体の中で占める割合はほぼ一定であり，大まかにどのアプリケーションについても，暗黙の処理をすべて強制削除すれば，すなわちプラットホーム依存の技法で暗黙の処理を実現できればコードサイズが約 40% 小さくなると推測できる．

図 8 に，全暗黙の処理を強制削除した場合に実行時間が受ける影響を評価した結果を示す．ただし，`_213_javac` と `_228_jack` については `ETest` を全部削除すると正常に動作しなくなるので，`Async` と `OutOf` に限って削除した．評価は組込み機器向けプ

☆ 非同期例外向けには，参考文献 20) に示した最適化のほかに，繰返し回数が小さな定数（100）以下だと静的に解析できるループから非同期例外向け例外発生検査を除去する最適化を実施する．

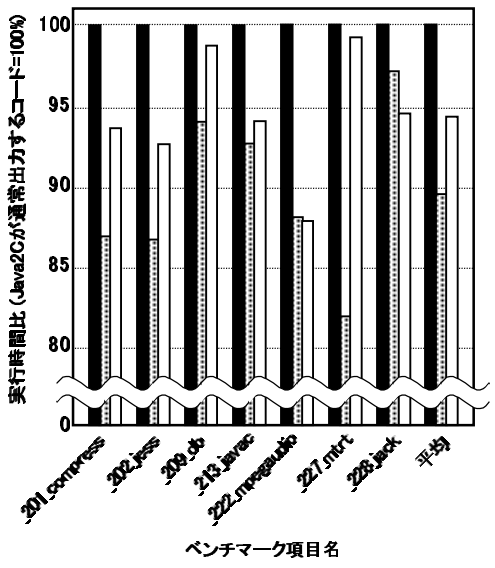


図 8 全暗黙の処理の強制削除が実行時間に与える影響
Fig. 8 Effect of compulsory removal of codes for all implicit features on execution time.

ロセッサ Am5x86™☆133 MHz を搭載した PC と，Pentium® ☆☆Ⅲ 450 MHz を搭載した PC（表 2 の PC1 と PC2）の 2 台を使って行った．図 8 から，暗黙の処理の強制削除した影響は多くの項目で Am5x86 を搭載した PC により顕著に現れ，Am5x86 を搭載した PC では実行時間が平均で 10.56%短くなるのに対し，PentiumⅢ を搭載した PC では 5.75%しか短くならないことが分かる．Am5x86 の方がより大きな影響を受ける原因の 1 つにキャッシュメモリのヒット率が考えられる．Am5x86 は PentiumⅢ に比べてキャッシュメモリの搭載量が少ない．このため，暗黙の処理を強制削除してコードサイズを小さくすることにより，キャッシュメモリのヒット率をより大きく改善して実行を高速化できる．

7. 関連研究

本論文では Java2C トランスレータで暗黙の処理を実現するにあたり，プラットホーム依存の技法や C 言語で表現できない技法を使えないために，どれだけオーバヘッドを被るか，コードサイズと実行速度の両面から評価した．Java2C トランスレータは研究用から商用まで，これまでいくつか開発されているが[6),9),10),13)]，本論文と同様の評価を与えている論文は著者の調査した限りでは存在しない．例外処理の実現に関しては，Krall らが CACAO という JIT コンパイラを使って 2 返戻値法と表引き法の比較を試みているが[8)]，ベンチマークを使った定量的な評価には及んでいない．

8. 結　論

Java2C トランスレータで暗黙の処理を実現するにあたり，プラットホーム依存の技法や C 言語で表現できない技法を使えないために，どれだけオーバヘッドを被るか評価した．評価は暗黙の処理を実現するコードを，Java2C トランスレータでプラットホーム非依存の最適化を適用して冗長なものを除去する場合と，強制的にすべて削除する場合の比較によって行った．強制的にすべて削除したコードは，プラットホーム依存の技法を使えるコンパイラが生成するコードに近い．SPECjvm98 を使って評価した結果，プラットホーム依存の技法を使えば省略できる暗黙の処理向けのコードサイズが，Java2C トランスレータが生成するコード全体の 40.03%を占めることが分かった．また，それらのコードの実行に，PentiumⅢ 450 MHz と Am5x86 133 MHz を搭載した PC においてそれぞれ実行時間全体の 5.75%および 10.56%を費やすことが分かった．

☆ Am5x86 は米国 Advanced Micro Devices Incorporated の米国およびその他の国における商標です．
☆☆ Pentium は米国 Intel Corporation の米国およびその他の国における登録商標です．

参 考 文 献

1) Budimlic, Z. and Kenedy, K.: Optimizing Java—Theory and Practice, *ASC 1997 Workshop on Java for Science and Engineering Computation* (1997).
2) Calder, B. and Grunwald, D.: Reducing Indirect Function Call Overhead In C++ Programs, *POPL 94*, pp.397–408 (1994).
3) Cameron, D., Faust, P., Lenkov, D. and Mehta, M.: A Portable Implementation of C++ Exception Handling, *USENIX C++ Technical Conference*, pp.225–243 (1992).
4) Detlefs, D. and Agesen, O.: Inlining of Virtual Methods, *ECOOP 99*, pp.259–278 (1999).
5) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison Wesley, Reading, Mass. (1996).
6) Howard, R.: Developing and Deploying Server-hosted Applications with Java (1997). http://www.twr.com/java/white-paper.html
7) Krall, A. and Grafl, R.: CACAO — A 64-bit just-in-time compiler, *Concurrency: Prac-*

tice and Experience, Vol.9, No.11, pp.1017–1030 (1997).
8) 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).
9) 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*, pp.1–20 (1997).
10) 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*, pp.41–53 (1997).
11) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks (1998). http://www.spec.org/osg/jvm98/
12) 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).
13) Weiss, M., Férriere, F., Delsart, B., Fabre, C., Hirsch, F., Johnson, E.A., Joloboff, V., Roy, F., Siebert, F. and Spengler, X.: TurboJ, a Java Bytecode-to-Native Compiler, *LCTES98*, pp.119–130 (1997).
14) Yang, B.-S., Moon, S.-M., Park, S., Lee, J., Lee, S., Park, J., Chung, Y.C., Kim, S., Ebcioglu, K. and Altman, E.: LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation, *1999 International Conference on Parallel Architectures and Compilation Techniques* (1999).
15) 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(PRO12), pp.81–96 (2001).
16) 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.SIG11(PRO12), pp.1–13 (2001).
17) 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：プログラミング，Vol.42，No.SIG11(PRO12)，pp.14–24 (2001).
18) 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：プログラミング，Vol.42, No.SIG2(PRO9), pp.37–47 (2001).
19) 千葉雄司：Java 向け静的コンパイラによる仮想メソッド呼出しの高速化，情報処理学会論文誌：プログラミング，Vol.42, No.SIG7(PRO11), pp.46–56 (2001).
20) 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).

(平成 14 年 1 月 17 日受付)
(平成 14 年 6 月 10 日採録)

千葉　雄司（正会員）

1972 年生．1997 年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．同年日立製作所（株）入社．システム開発研究所にてコンパイラの研究開発に従事．2001 年より慶應義塾大学非常勤講師を兼務．ソフトウェア科学会会員．