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

千 葉 雄 司†

メモリ資源などの制約が厳しい組込み機器向けコンパイラの開発にあたっては，コードサイズの削減が重要な課題になる．本論文では，組込み機器向け Java2C トランスレータの開発にあたり，コードサイズを削減するために，例外処理の実現方法に配慮する必要があることを指摘する．そして，2 返戻値法で例外処理を実現する Java2C トランスレータ向けのコードサイズの削減法として，冗長な例外発生検査の除去や下方移動による例外発生検査の集約，例外オブジェクトの生成を `catch` まで遅延する技法を提案する．SPECjvm98 による評価の結果，本論文で提案する技法により，SH4 プラットホームにおけるコードサイズを相加平均で 5.21% 削減できることが分かった．

An Implementation of Exception Handling by Two Return Values Method in a Java2C Translator for Embedded Machines

Yuji Chiba†

Code size is one of the most important problems that compiler for embedded machines must struggle, because embedded machines often have poor memory resource. This paper shows that exception handling is one of the key points for Java2C translators to make the code size compact, and presents techniques that implement exception handling using less code by removing redundant tests for thrown exception, downward integration of the tests, and delaying generation of exception objects. These techniques can be adopted by Java2C translators that implement exception handling by two return values method. The result of SPECjvm98 shows that these techniques improve code size for SH4 platform by 5.21% on average.

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 の利点の 1 つは可搬性の高さにあり，Java で記述したアプリケーションは組込み機器からサーバまで，機種を問わず実行できる．なぜなら，Java ではアプリケーションをバイトコードという機種非依存の中間語に変換して配布するからである．アプリケーションの実行時には，多くのプロセッサではバイトコードを直接実行することはできず，インタプリタを使って解釈実行するか，あるいはコンパイラで機械コードに変換してから直接実行する．

Java 向けコンパイラは，実行時にコンパイルを行う Just In Time（JIT）コンパイラと，実行前にあらかじめコンパイルする静的コンパイラの 2 種類に分類できる．Java アプリケーションはインタプリタ，JIT コンパイラあるいは静的コンパイラのどれを使っても実行できるが，それぞれ消費メモリや起動待ち時間などに利点や欠点を持つため，ユーザあるいは開発者が利用可能なメモリ容量やアプリケーションの性質などを考慮して最適な実行手段を選ぶ必要がある．

PC 上での Java アプリケーションの実行手段としては JIT コンパイラが一般的だが，組込み機器の分野では必ずしもそうでなく，インタプリタを使う場合も多い．なぜなら，組込み機器では PC に比べてメモリなどの資源的制約が厳しく，JIT コンパイラを搭載すること自体に消費するメモリや，実行時にコンパイルするために必要になる時間やメモリを無視できないからである．JIT コンパイラのプログラムサイズや，コンパイル時に消費する資源は，実装する最適化を減らすことで抑えることもできるが，それでは実行速度を改善しにくい．また，組込み機器の分野では多彩なプロセッサや OS を利用するが，それぞれのプラット

† （株）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 実行時例外
Table 1 Runtime exceptions.

名称	略号	暗黙の発生源になるバイトコードの種別
java.lang.NullPointerException	Null	配列参照，配列長取得，インスタンスフィールド参照，インスタンスメソッド呼び出し，例外投擲，モニタ演算
java.lang.ArithmeticException	Arith	整数の除算，剰余
java.lang.ArrayIndexOutOfBoundsException	Index	配列参照
java.lang.ArrayStoreException	Store	参照型配列へのストア
java.lang.ClassCastException	Cast	クラスキャスト検査

表 2 例外を発生しうるバイトコード命令数
Table 2 Bytecode counts which may cause exceptions.

ベンチマーク項目名	バイトコード命令総数	発生しうる例外名（略号）					総計
		Null	Arith	Index	Store	Cast	
_201_compress	10,751	2,475(23.0%)	45(0.42%)	201(1.87%)	24(0.22%)	24(0.22%)	2,544(23.7%)
_202_jess	25,956	7,055(27.2%)	53(0.20%)	442(1.70%)	42(0.16%)	99(0.38%)	7,207(27.8%)
_209_db	12,100	2,786(23.0%)	43(0.36%)	212(1.75%)	19(0.16%)	42(0.35%)	2,871(23.7%)
_213_javac	45,720	12,661(27.7%)	51(0.11%)	499(1.09%)	79(0.17%)	332(0.73%)	13,044(28.4%)
_222_mpegaudio	20,528	5,180(25.2%)	52(0.25%)	1,307(6.37%)	49(0.24%)	31(0.15%)	5,263(25.6%)
_227_mtrt	16,780	4,176(24.9%)	44(0.26%)	469(2.79%)	39(0.23%)	28(0.17%)	4,248(25.3%)
_228_jack	24,861	6,501(26.2%)	57(0.23%)	1,178(4.74%)	11(0.04%)	166(0.67%)	6,724(27.1%)
相加平均		(25.7%)	(0.26%)	(2.90%)	(0.18%)	(0.38%)	(26.0%)

※括弧内の数値は総バイトコード数に対する比率.
※例外名の略号の意味は表 1 に準ずる.

ホーム向けに JIT コンパイラを開発し，維持管理するには多大な人的資源が必要になる．

資源あるいは人的制約から JIT コンパイラの適用が難しいが，インタプリタでは十分な実行速度を得られない場合などに有効な Java アプリケーションの高速化手段に静的コンパイラがある．静的コンパイラはコンパイル作業を実行前に行うため，実行時に行う JIT コンパイラに比べ次の利点を持つ．

- コンパイラを組込み機器に搭載する必要がなく，メモリ資源の圧迫を回避できる．
- コンパイルを組込み機器上で実施する必要がなく，実行時にコンパイルのためのオーバヘッドが生じない．PC など組込み機器より潤沢なプロセッサやメモリ資源を持つ機械でコンパイルを実施でき，なおかつ JIT コンパイラよりコンパイル時間の制約が緩いため，十分に最適化を実施できる．

静的コンパイラには様々な構成法があるが，Java2C トランスレータを使う方法では，コンパイラの開発や維持にかかる人的資源も抑止できる．この方法では，静的コンパイラを Java2C トランスレータと C コンパイラから構成し，まず，Java アプリケーションを Java2C トランスレータで C ソースに変換し，次に C コンパイラで機械コードに変換する．Java2C トランスレータが出力する C ソースを特定のプラットホームに依存しないものにすれば，Java2C トランスレータをプラットホーム間で共有できる．プロセッサに固有な最適化や機械コードの生成は各プラットホームが提供する既存の C コンパイラで実施する．

コードサイズの削減は，メモリなど資源的制約が厳しい組込み機器向け Java2C トランスレータにとって重要な課題となる．Java2C トランスレータで利用できる例外処理の実現方法には setjmp 法と 2 返戻値法があるが[11]，本論文の目的は，2 返戻値法による例外処理の実現においてコードサイズを削減する効果を持つ技法を提案し，その効果を定量的に評価することにある．まず 2 章で，コードサイズの削減にあたり例外処理の実現に配慮する必要があることを指摘する．また，関連研究として，例外処理に関連してコードサイズを削減する効果を持つ既存の技法を示す．次に，3 章で 2 返戻値法による例外処理向けのコードサイズ削減法として，冗長な例外発生検査を除去する技法や，例外オブジェクトの生成を `catch` まで遅延する技法などを提案し，続く 4 章で提案した技法がコードサイズに与える影響を評価する．5 章は結論である．

2. 例外処理がコードサイズに与える影響

本論文で評価に用いる Java 向けベンチマーク SPECjvm98[7] を構成するクラス中のメソッドが，**表 1** に示す実行時例外を暗黙に発生しうるバイトコード☆をどれだけ含むか調査した結果を**表 2** に示す．

☆ 暗黙の発生とは，Java ソース上で `throw` によって明示的に投擲する以外の原因により例外を発生することを表す．たとえば，配列を参照するバイトコードは添字が範囲外であることを表す例外 `java.lang.ArrayIndexOutOfBounds` を暗黙に発生しうる．

SPECjvm98 は javac など中〜小規模の実用的アプリケーションを構成要素とするベンチマークである．表 2 では，調査対象のメソッドを次の手順で定めた．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調査対象とした．

表 2 から，相加平均で 26.0% ，おおむね 4 つに 1 つものバイトコードが表 1 に示す 5 種類の例外のいずれかを発生しうることが分かる．このため，例外に関連する最適化は，Java 向けコンパイラにとって，実行速度とコードサイズの両方の意味で重要な課題となっている．最も多くのバイトコードが発生しうる NullPointerException については，OS が 0 番ページ（アドレス 0H から始まるページ）への参照をトラップする機能を備えている場合には，トラップを使った暗黙の null 検査により，明示的な null 検査のコードを省略できる[10]．暗黙の null 検査はコードサイズを削減するうえでも実行速度を改善するうえでも有効であり，いくつかの Java 向けコンパイラが採用している[1),8),9)]．しかし，組込み機器向け Java2C トランスレータで暗黙の null 検査を利用することは，可搬性を重視する場合，次の理由から困難である．

- 組込み OS の中には，0 番ページへの参照をトラップする機能を持たないものもある．
- 暗黙の null 検査を使うには，一部のコード移動最適化を抑止する必要がある．しかし，C コンパイラにコード移動最適化を抑止すべき部分を指示することは困難である．
- 暗黙の null 検査では，NullPointerException 発生時に生じるトラップを捕捉し，例外発生時のプログラムカウンタの値から対応するハンドラを求めてジャンプする機能が必要になる．しかし，C 言語には例外の発生源となる文のアドレスを求める機能がないため，プログラムカウンタの値とハンドラを対応づける処理の実装が困難である．

冗長な null 検査（や配列添字検査など）は，暗黙の null 検査を利用できなくても，フロー解析を使って除去できる[2),4)]．しかし，暗黙の null 検査とは違い，フロー解析を使って除去できるのは冗長な検査コードのみであり，冗長でないコードは残る．

☆ 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:  val1 = obj1.field;
 2:  val2 = obj2.field;
```

```
 0:  /* C ソース（最適化なし）*/
 1:  if (obj1 == NULL){
 2:    throwNullPointerException();
 3:    goto ハンドラ;
 4:  } else {
 5:    val1 = obj1->field;
 6:  }
 7:  if (obj2 == NULL){
 8:    throwNullPointerException();
 9:    goto ハンドラ;
10:  } else {
11:    val2 = obj2->field;
12:  }
```

```
 0:  /* C ソース（最適化あり）*/
 1:  if (obj1 == NULL){
 2:    goto SHARED_FUNC_CALL;
 3:  } else {
 4:    val1 = obj1->field;
 5:  }
 6:  if (obj2 == NULL){
 7:  SHARED_FUNCTION_CALL:
 8:    throwNullPointerException();
 9:    goto ハンドラ;
10:  } else {
11:    val2 = obj2->field;
12:  }
```

図 1 共通式の下方集約
Fig. 1 Downward common subexpression integration.

残ったコードのサイズを削減する方法として，Krall らは例外オブジェクトを生成して投擲する関数呼び出しを下方集約する技法を示している[6]．たとえば**図 1** 上段の連続した 2 つのインスタンスフィールド参照のコードは，それぞれ NullPointerException を発生しうる．ここでコンパイラが個々のインスタンスフィールド参照ごとに NullPointerException を生成して投擲する関数呼び出しを出力すると図 1 中段のコードになるが，Krall らの技法では中段のコードの 2 行目と 8 行目にある関数呼び出しを下方集約して 1 カ所にまとめ，図 1 下段のコードにする．下段のコードでは関数呼び出しが 8 行目の 1 カ所になり，2 行目（中段のコードでは 3 行目）の goto 文のジャンプ先が 7 行目のラベル SHARED_FUNCTION_CALL に変わっているが，その他の部分は中段のコードと同一である．結果として，下段のコードは中段のコードと比較して関数呼び出し 1 カ所分だけサイズが小さくなる．

3. 例外処理に関するコードサイズの削減

本章では 2 返戻値法で例外処理を実現する Java2C トランスレータ向けに，新たに次に示す 3 つのコード

サイズの削減方法を提案する．

- 冗長な例外発生検査の除去
- 下方移動による例外発生検査の集約
- 例外オブジェクト生成の遅延

3.1 2 返戻値法による例外処理の実現

提案に先立ち，まず，2 返戻値法による例外処理の実現について述べる．2 返戻値法は Krall らが JIT コンパイラの実装にあたって採用した例外処理の実現方法であり[5]，Java2C トランスレータへの適用については参考文献 11) で述べている．

2 返戻値法による例外処理の実現では，まず，メソッドの返戻値を，通常の返戻値と例外の 2 個にする．次に，メソッド呼び出しの直後に例外が発生したか検査するコード（例外発生検査）を挿入し，2 個目の返戻値（例外）を検査する．例外が発生していた場合，ハンドラがあるならばハンドラにジャンプし，ないならば，2 個目の返戻値に例外をセットしたままでメソッドから返戻する．

2 返戻値法で例外処理を実現する Java2C トランスレータによって**図 2** 上段の例外処理を含む Java ソースを C ソースに変換した結果を図 2 下段に示す．図 2 上段の Java ソース中にあるメソッド `sum()` は，引数のリスト `list` につながれたセルが格納する整数値の合計を求める．具体的には，3 行目で突入する `try` ブロック内で無限ループにより，メソッド `getValue1()` を呼び出してセルが格納する値を求め，順々に加算する．セルがなくなると，メソッド `getValue1()` のさらに呼出し先のメソッド `getValue2()` で `NullPointerException` が発生し，無限ループから抜ける．そして，8 行目の `catch` 節で発生した例外を捕捉して読み捨て，最後に，これまでの加算により求めた合計値を返戻する．

`getValue2()` における `NullPointerException` の発生から，呼出し元のさらに呼出し元である `sum()` で例外を捕捉するまでの過程を，図 2 下段の C ソースで実施する手順を次に示す．

(1) `getValue2()` の引数 `cell` が `NULL` である場合，33 行目の分岐から 34 行目に進んで関数 `throwNullPointerException()` を呼び出す．`throwNullPointerException()` では，まず，`NullPointerException` オブジェクトを生成し，生成したオブジェクトへのポインタを，スレッド固有の資源を格納する構造体に収める．図 2 下段のコード中にあるポインタ `ee` はこの構造体を参照する．次に，同構造体中に存在する，例外の発生状況を表す「例外発生フラグ」を立てる．このフラグが関数

```
 0:  /* Java ソース */
 1:  static int sum(List list){
 2:    int result = 0;
 3:    try{
 4:      while(true){
 5:        result += getValue1(list);
 6:        list = list.next;
 7:      }
 8:    } catch(NullPointerException e){}
 9:    return (result);
10:  }
11:  static int getValue1(List cell){
12:    int result = getValue2(cell);
13:    return (result);
14:  }
15:  static int getValue2(List cell){
16:    return (cell.value);
17:  }
```

```
 0:  /* C ソース（最適化前）*/
 1:  int sum(ExecEnv *ee, List *list){
 2:    int result = 0;
 3:    Object *atemp;
 4:    while(true){
 5:      int itemp = getValue1(ee, list);
 6:      if (exceptionOccurred(ee))
 7:        goto CATCH;
 8:      result += itemp;
 9:      if (list == NULL){
10:        throwNullPointerException();
11:        goto CATCH;
12:      }
13:      list = list->next;
14:      /* 非同期例外の検出 */
15:      if (exceptionOccurred(ee))
16:        goto CATCH;
17:    }
18:  CATCH:
19:    atmp = catch(ee, NullPointerException);
20:    return (result);
21:  }
22:  int getValue1(ExecEnv *ee, List *cell){
23:    int result = 0;
24:    int itemp = getValue2(ee, cell);
25:    if (exceptionOccurred(ee))
26:      goto RETURN;
27:    result = itemp;
28:  RETURN:
29:    return (result);
30:  }
31:  int getValue2(ExecEnv *ee, List *cell){
32:    int itemp;
33:    if (cell == NULL)
34:      throwNullPointerException();
35:    else
36:      itemp=cell->value;
37:    return (itemp);
38:  }
```

図 2 2 返戻値法による例外処理の実現
Fig. 2 Implementation of exception handling by two return values method.

getValue2() の「2 個目の返戻値」となる．最後に，throwNullPointerException() から getValue2() に返戻し，さらに，37 行目を経由して getValue1() に返戻する．

(2) getValue1() では，getValue2() から返戻した直後の 25 行目で例外発生検査を実施する．具体的には，2 個目の返戻値，すなわち例外発生フラグの値を検査する．ここでは例外が発生しているので 26 行目に進み，29 行目を経由して sum() に返戻する．

(3) sum() でも getValue1() から返戻した直後の 6 行目で例外発生検査を実施し，例外が発生しているので 7 行目を経由して Java ソースの catch に相当する 19 行目に進み，関数 catch() を呼び出す．catch() では，発生した例外が，指定した捕捉対象クラス NullPointerException のインスタンスであれば捕捉する．具体的には例外発生フラグを下ろし，発生した例外を返戻する．捕捉しない場合には，例外発生フラグには変更を加えずに NULL を返戻する．

3.2 冗長な例外発生検査の除去

図 2 下段の C ソースは，コードサイズの点で改善の余地がある．たとえば 15 行目と 25 行目にある例外発生検査は次に述べる理由から冗長であり，これらを除去することでコードサイズを削減できる．

3.2.1 冗長な非同期例外発生検査の除去

15 行目の例外発生検査は非同期的に発生する例外を検出するために挿入したものである．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．この指示を検出する方法には様々な実装がありうるが，Java2C トランスレータで利用できる可搬性の高い方法としては，例外発生フラグのポーリングがある．この方法では，C コード中の複数の箇所に例外発生検査のコードを挿入する．挿入したコードは例外発生フラグをポーリングし，非同期例外が発生して例外発生フラグが立つと，これを検出して例外処理に制御を移す．

例外発生検査のコードを挿入する箇所としては，Java バイトコードにおいて負方向へのジャンプをしうる命令の手前が考えられる．ループの原因となる負方向へのジャンプの手前に例外発生検査を挿入すれば，ループ本体実行時に必ず 1 度は例外発生検査がかかり，無限ループに陥った場合に，いつまでも非同期例外の発生を検出できなくなる現象を抑止できる．

15 行目の例外発生検査は負方向ジャンプ命令の手前で非同期例外を検出するために存在するが，これは冗長であり除去できる．なぜなら同じループ本体内の 6 行目に例外発生検査が存在し，すでにループ本体実行時に 1 度は例外発生検査を実行することが保証されているからである．一般に非同期例外の検出を目的として挿入した例外発生検査は，例外発生検査を収める基本ブロックから出発して例外発生検査を通過せずに出発元の基本ブロックに戻るパスが存在しなければ冗長であり除去できる．

3.2.2 下方移動による冗長な例外発生検査の除去

例外発生検査は，例外を投擲しうるメソッド呼び出しの直後に挿入するのが基本だが☆，例外非発生時の分岐側に存在する文を越えて下方移動できる場合もある．下方移動が可能になる条件は，移動によって例外発生時の挙動が変わらないことである．原則的には，例外非発生時の分岐側に存在する文を越えて移動することで，その文が生じる副作用により例外発生時の分岐側以下で実行する変数参照やメモリ参照が影響を受けてはならない．この規則に従って順次下方移動した結果，次のいずれかの条件を満した例外発生検査は冗長と見なして除去できる．

(1) 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．

(2) 他の例外発生検査と合併できる場合．下方移動先に非同期例外のポーリング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乗り越え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，この

☆ 例外を投擲しえないメソッド呼び出しの後ろに例外発生検査を挿入する必要はない．4 章で評価に使う JeanPaul[12] の Java2C トランスレータは手続き間解析によってメソッドが例外を投擲しうるか判断し，例外を投擲しえないメソッドについてはメソッド呼び出し後の例外発生検査を除去する機能を持つ．この機能はコードサイズを削減する効果を持つはずだが，役に立つ場合は少ない．なぜなら例外を投擲しえ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るが，インライン展開を適用すれば展開元のメソッドが例外を投擲するか否かによらず例外発生検査が消失するからである．なお，この機能のように例外発生検査を省略する最適化を適用する場合，ループを生成する最適化である末尾再帰の除去は Java2C トランスレータで実施し，ループ内に例外発生検査を挿入する必要がある．なぜなら，末尾再帰の除去を C コンパイラに実施させると例外発生検査を含まない無限ループが生じて問題になりうるからである．末尾再帰のように除去の対象にならない再帰も一種のループだが，これは関数呼びを介したループであるため，返戻時に例外発生検査がかかる．

こと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，むしろメソッド呼び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．なお，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に影響を与える文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，通常の返戻値が無効になり，どのような値をとってもかまわないからである．
- 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分かっていることから，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 2 下段の C ソースの 25 行目の例外発生検査は，下方移動によって除去できる．なぜなら，この例外発生検査は特例により 27 行目の代入文を越えて下方移動できるが，下方移動すると例外発生時の分岐先と非例外発生時の分岐先が同一（ともに 29 行目の `return` 文）になり，冗長と見なせるからである．

3.2.3 下方移動による例外発生検査の集約

例外発生検査の下方移動は，複数の例外発生検査を集約する用途にも有用であり，これによりコードサイズを削減できる．たとえば**図 3** 上段の Java ソースを単純に C ソースに変換すると図 3 中段のコードになるが，中段のコードの 3 行目と 7 行目にある例外発生検査を下方集約して図 3 下段のコードに変換することで，コードサイズを削減できる．

C コンパイラの多くは共通式の下方集約する最適化を提供し，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 2 返戻値法固有の特例を使って例外発生検査を下方移動することはできない．

3.3 例外オブジェクト生成の遅延

図 2 下段の C ソースでは，10 行目と 34 行目の例外発生箇所で，関数 `throwNullPointerException()` を呼び出して例外オブジェクトを生成しているが，関数呼び出しはコードサイズが大きいことが多い．そこ

```
0:  /* Java ソース */
1:  if (condition)
2:    this.boo();
3:  else
4:    this.foo();
```

```
0:  /* C ソース（最適化なし）*/
1:  if (condition){
2:    boo(ee, this);
3:    if (exceptionOccurred(ee))
4:      goto ハンドラ;
5:  } else {
5:    foo(ee, this);
7:    if (exceptionOccurred(ee))
8:      goto ハンドラ;
9:  }
```

```
0:  /* C ソース（最適化あり）*/
1:  if (condition)
2:    boo(ee, this);
3:  else
4:    foo(ee, this);
5:  if (exceptionOccurred(ee))
6:    goto ハンドラ;
```

図 3 例外発生検査の下方集約
Fig. 3 Downward thrown exception test integration.

で，例外発生箇所における関数呼び出しを，発生する実行時例外のクラスに応じた値を例外発生フラグに書き込む動作（メモリへのストア）で代用する技法を提案する．メモリへのストアは，多くのプラットホームにおいて関数呼び出しよりコードサイズが小さいため，この技法によりコードサイズを削減できる．例外発生フラグに値を書き込むだけでは実行できない，`NullPointerException` などの例外オブジェクトを生成する処理については，例外の投擲に続いて必ず実施する動作である `catch`（あるいは `finally`）まで遅延する．Java ソースの `catch` に相当する C ソース上の箇所には図 2 下段のコードの 19 行目のように，発生した例外が捕捉対象か検査するライブラリ関数呼び出し `catch()` がもとからあり，このライブラリ関数 `catch()` の中に例外発生フラグの値に対応する例外オブジェクトの生成処理を追加すれば，例外発生フラグに値を書き込む動作によってコードサイズを削減しつつ例外を発生することと，例外オブジェクトを生成することを両立できる．

例外発生用関数呼び出しの例外発生フラグへの書き込みによる代用は，バイトコードが表 1 に示す例外を暗黙に投擲する箇所に対して適用する．例外にはバイトコードが暗黙に投擲するもののほかに，プログラム上で明示的に生成して投擲するものもあるが，後者には適用しない．なぜなら，後者では任意のクラスの例外オブジェクトを生成しうるが，生成の過程でモニタに依存する動作を実施しうるからである（表 1 に示

す例外オブジェクトの生成中にはモニタに依存する動作は起きない）．2 返戻値法では，例外を投擲してから catch に到達するまで順次メソッド呼び出しから返戻する過程で，同期メソッドから返戻する（同期ブロックから脱出する）場合には確保したモニタを解除する[11]．したがって catch に到達したときには，例外投擲時とは違っていくつかのモニタが解除されていることがあり，例外オブジェクトの生成の過程でモニタに依存した動作がある場合には，生成を遅延すると，プログラムの挙動がおかしくなりうる．なお，最適化の適用対象をバイトコードが暗黙に例外を投擲する箇所に限定しても，最適化がコードサイズに与える影響はほとんど変わらない．なぜならプログラム上で明示的に例外を投擲することは比較的まれであり，それよりもバイトコードが暗黙に投擲する箇所のほうが圧倒的に多いからである．

例外オブジェクトの生成を遅延するにあたっては，例外オブジェクトに正しいスタックトレース情報を与えるために，Java スタックフレームの解放も catch まで遅延する必要がある☆．すなわち，2 返戻値法ではメソッド呼び出しから返戻しながら対応するハンドラを検索するが，catch に到達するまでは，返戻時に Java スタックフレームを解放してはならない．さもないと，スタックトレース情報のうえでは catch したメソッドで例外が発生したことになってしまう．

例外オブジェクトを生成する関数呼び出しを例外発生フラグへの値の書き込みで代用する技法の実装について詳述する．この技法では，例外 NullPointerException を生成する関数呼び出し throwNullPointerException() を**図 4** に示すコードで代用する．

図 4 のコードにおいて変数 ee が参照する構造体のメンバ exceptionKind は例外発生フラグを表す．例外発生フラグなどスレッド固有の資源を収める構造体型 ExecEnv と，関連するマクロの定義を**図 5** に示す．図 4 のコードでは相互排除なしで例外発生フラグに値を書き込んでいるが，これが可能なのは図 5 のコー

☆ 図 2 下段の C ソースでは，Java スタックフレームに関する処理を省略している．コードサイズの縮小が重要である一方で利用時にスタックトレース情報を使う可能性が稀な組込み機器向けの Java2C トランスレータでは，Java スタックフレームに関する処理を省略するのも 1 つの解である．4 章で評価に使う JeanPaul の Java2C トランスレータでも Java スタックフレームに関する処理を一部省略している．もっとも，java.lang.SecurityManager.getClassContext() のようにスタックトレースを取得するクラスライブラリメソッドが存在するので，すべてのスタックフレームに関する処理を省略することはできない．

```
ee->exceptionKind.detail.sync =
        EXCKIND_NullPointerException;
```

図 4 NullPointerException を発生するコード
Fig. 4 Code to generate NullPointerException.

```
 0:  /*
 1:  ** スレッド固有の資源を収める構造体型の定義
 2:  */
 3:  struct execenv {
 4:    ...
 5:    /*
 6:    ** 例外関係のフィールド群の宣言
 7:    ** 同期例外投擲時の排他制御コードを不要
 8:    ** にするため同期例外用と非同期例外用に
 9:    ** 別個のフィールドを提供する．
10:    ** exceptionKind:
11:    ** 例外発生フラグ
12:    ** exception:
13:    ** 同期例外を格納するフィールド
14:    ** async_exception:
15:    ** 非同期例外を格納するフィールド
16:    */
17:    union{
18:      struct{
19:        char sync;
20:        char async;
21:      } detail;
22:      volatile short summary;
23:    } exceptionKind;
24:    JHandle *exception;
25:    JHandle *async_exception;
26:    ...
27:  };
28:
29:  typedef struct execenv ExecEnv;
30:
31:  /*
32:  ** 例外発生フラグに収める値の定義
33:  ** ExecEnv の exceptionKind フィールドに
34:  ** 次のいずれかの値が入る．
35:  **/
36:  #define EXCKIND_NONE  0
37:  #define EXCKIND_THROW 1
38:  #define EXCKIND_NullPointerException  3
39:  #define EXCKIND_ArrayIndexOutOfBounds 4
40:  #define EXCKIND_ArrayStoreException   5
41:  #define EXCKIND_ArithmeticException   6
42:  #define EXCKIND_ClassCastException    7
43:
44:  /*
45:  ** 例外発生検査に使うマクロの定義
46:  */
47:  #define exceptionOccurred(ee) \
48:      ((ee)->exceptionKind.summary != 0)
49:
```

図 5 スレッド固有の資源を収める構造体型の定義
Fig. 5 Definition of execution environment.

表 3 最適化の略号
Table 3 Abbreviations for optimizations.

略号	名称
Async	冗長な非同期例外発生検査の除去
Downward	下方移動による冗長な例外発生検査の除去
Drain	下方移動による例外発生検査の集約
Delay	例外オブジェクト生成の遅延

ドで例外発生フラグを同期例外用のフィールド sync と非同期例外用のフィールド async に分割しているからである．相互排除の省略はコードサイズを削減するうえで役立っている．図 5 ではフィールド sync と async の型を char としているが，この型は実行機械が不可分に参照できるメモリ幅に従って定める．

図 5 で例外発生フラグを 2 つのフィールドに分割したからといって，例外発生検査時に個々のフィールドを別個に検査する必要はない．フィールド sync と async は連続したメモリ領域に存在し，ひとまとめの値 summary として参照できる．例外発生検査において使用するマクロ exceptionOccurred(ee) では，これらのフィールドを 1 回のメモリ参照で，ひとまとめの値 summary として参照する（図 5 の 47 行目）．例外発生フラグを構成する個々のフィールド sync と async には例外が発生していない場合，図 5 の 36 行目で定義している値 EXCKIND_NONE（つまり 0）が入っているので，同期例外も非同期例外も発生していなければ summary の値は 0 になる．したがって，summary の値が 0 か調べれば例外が発生しているか検査できる．

なお，22 行目では summary フィールドを volatile 宣言しているが，これには C コンパイラが非同期例外ポーリング用の例外発生検査を不変式と見なしてループ外に移動することを防ぐ意味がある．

4. 評　　価

本論文で示したコードサイズ削減方法を JeanPaul[12] の Java2C トランスレータ上に実装し，その効果を Java 向けベンチマーク SPECjvm98 を用いて評価した．本章では，実装した最適化を**表 3** に示す略号によって表記する．評価対象プラットホームは SH4（Windows®CE）および x86（Windows 2000）とした．C コンパイラは SH4 では embedded Visual C++®3.0，x86 では 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☆．

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．これは表 2 を求める際に適用したメソッドの選択基準と同一である．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点が消失したメソッドについて C コードを生成しない．

評価結果を**表 4** に示す．表 4 は，本論文で述べた全最適化（表 3 の全最適化と例外発生検査の下方移動）を適用して得た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）を基準として，各最適化の抑止が総コード長に与える影響を表す．なお，表 4 上段の SH4 向けコンパイル結果と下段の x86 向けコンパイル結果を比較すると，おおむね各ベンチマーク項目のコードサイズが同程度であることが分かるが，このことから SH4 と x86 のコード効率が同程度であるとは単純にはいえない．なぜならコンパイルに使用した C コンパイラが SH4 と x86 で同一でなく，その性能差がコードサイズに及ぼす影響が少からずあるからである．

4.1 例外発生検査向け最適化の効果

表 4 の相加平均の項目から，例外発生検査向け最適化 Async，Downward，Drain がそれぞれ SH4 向けのコードサイズを 0.17，0.20，1.61%コードサイズを削減する効果があることが分かる．これらのうち Downward については，Java2C トランスレータで除去できる例外発生検査の個数に比してコードサイズへの影響が小さい傾向がある．最適化の抑止が Java2C トランスレータの出力する C ソース中の例外発生検査数に与える影響を測定した結果（**表 5**）と表 4 を比較すると，相加平均の項目から Drain を抑止すると例外発生検査が 10.05%増加し，このとき SH4 でコードサイズが 1.61%増加することが分かるが，Downward では例外発生検査の個数が 7.67%増加するのに，コードサイズは 0.20% しか増加しない．

Downward がコードサイズに及ぼす影響が小さい原因は，Java2C トランスレータが実施するフロー最適化により，Downward で除去できる例外発生検査の多くが C コンパイラでも除去可能になることにある．たとえば図 2 下段の C ソースの 25 行目にあるメソッド

☆ 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．

表 4 最適化がコードサイズに及ぼす影響
Table 4 Effect of optimizations on code size.

(a) SH4 プラットホーム

ベンチマーク項目名	総バイトコード長	総機械コード長					
		全最適化あり	抑止した最適化（略号）				
			Async	Downward	Drain	Delay	すべて
_201_compress	20,952	106,794 (510%)	106,976 (0.17%)	106,962 (0.16%)	108,168 (1.27%)	108,810 (1.85%)	111,658 (4.36%)
_202_jess	53,795	297,854 (554%)	298,758 (0.30%)	298,788 (0.31%)	302,076 (1.40%)	304,060 (2.04%)	316,480 (5.89%)
_209_db	24,118	124,116 (515%)	124,292 (0.14%)	124,216 (0.08%)	125,950 (1.46%)	126,442 (1.84%)	130,138 (4.63%)
_213_javac	101,606	497,358 (489%)	498,140 (0.16%)	499,150 (0.36%)	511,934 (2.85%)	506,568 (1.82%)	533,884 (6.84%)
_222_mpegaudio	37,621	186,336 (495%)	186,880 (0.29%)	186,720 (0.21%)	187,914 (0.84%)	193,978 (3.94%)	198,648 (6.20%)
_227_mtrt	32,615	193,414 (593%)	193,486 (0.04%)	193,868 (0.23%)	195,610 (1.12%)	196,862 (1.75%)	202,502 (4.49%)
_228_jack	37,621	286,146 (596%)	286,312 (0.06%)	286,202 (0.02%)	292,920 (2.31%)	288,180 (0.71%)	298,356 (4.09%)
相加平均		– (536%)	– (0.17%)	– (0.20%)	– (1.61%)	– (1.99%)	– (5.21%)

(b) x86 プラットホーム

ベンチマーク項目名	総バイトコード長	総機械コード長					
		全最適化あり	抑止した最適化（略号）				
			Async	Downward	Drain	Delay	すべて
_201_compress	20,952	105,664 (504%)	105,936 (0.26%)	106,880 (1.14%)	107,008 (1.26%)	105,136 (-0.50%)	108,688 (2.78%)
_202_jess	53,795	295,376 (549%)	297,536 (0.73%)	299,504 (1.38%)	299,776 (1.47%)	296,144 (0.26%)	311,840 (5.28%)
_209_db	24,118	122,336 (507%)	122,784 (0.36%)	124,112 (1.43%)	124,352 (1.62%)	122,272 (-0.05%)	127,280 (3.88%)
_213_javac	101,606	476,912 (469%)	478,368 (0.30%)	484,336 (1.53%)	492,512 (3.17%)	477,872 (0.20%)	509,600 (6.41%)
_222_mpegaudio	37,621	188,512 (501%)	189,632 (0.59%)	190,336 (0.96%)	190,192 (0.88%)	186,064 (-1.32%)	192,320 (1.98%)
_227_mtrt	32,615	184,464 (566%)	184,992 (0.29%)	187,120 (1.42%)	186,560 (1.12%)	184,832 (0.20%)	192,544 (4.20%)
_228_jack	37,621	245,136 (510%)	245,376 (0.10%)	247,264 (0.86%)	251,488 (2.53%)	244,128 (-0.41%)	255,088 (3.90%)
相加平均		– (515%)	– (0.38%)	– (1.25%)	– (1.72%)	– (-0.23%)	– (4.06%)

※単位はバイト.
※全最適化ありの項目の括弧内の数値は総バイトコード長に対する比率.
※その他の項目の括弧内の数値は全最適化ありの項目に対する比率.

表 5 最適化が C ソース中の例外発生検査数に及ぼす影響
Table 5 Effect of optimizations on counts of test for thrown exceptions in C code.

ベンチマーク項目名	例外発生検査総数（全最適化あり）	最適化の抑止により増加する例外発生検査数			
		Async	Downward	Drain	すべて
_201_compress	1,714	30(1.75%)	137(7.99%)	127(7.41%)	391(22.8%)
_202_jess	5,107	142(2.78%)	400(7.83%)	405(7.93%)	1,327(26.0%)
_209_db	2,086	33(1.58%)	155(7.43%)	176(8.44%)	508(24.4%)
_213_javac	7,049	78(1.11%)	744(10.55%)	1,320(18.73%)	3,116(44.2%)
_222_mpegaudio	2,502	74(2.96%)	200(7.99%)	156(6.24%)	634(25.3%)
_227_mtrt	2,814	16(0.57%)	186(6.61%)	190(6.75%)	656(23.3%)
_228_jack	3,941	27(0.69%)	208(5.28%)	587(14.89%)	1,140(28.9%)
相加平均		(1.63%)	(7.67%)	(10.05%)	(27.9%)

※括弧内の数値は全最適化を適用した場合の例外発生検査総数に対する増加率.

getValue1() 中の例外発生検査は，除去すると通常の返戻値 result に影響するので C コンパイラでは除去できない．除去するには Java2C トランスレータで最適化する必要がある．この例外発生検査は Downward で除去できるが，フロー最適化でも除去できる．Java2C トランスレータにおいて 2 返戻値法固有の特例を適用し，例外が発生した状態で返戻する場合には通常の返戻値 result がどんな値をとってもよいという条件を与えてフロー最適化を実施すると，getValue1() は**図 6** に示す C ソースになるが，図 6 の 2 行目にある例外発生検査は例外発生時の分岐先と非発生時の分岐先が同一であり，C コンパイラでも冗長と判断して除去できる．表 4 で Downward がコードサイズに与える影響が小さくなっているのは，Downward のみ抑止して 2 返戻値法を意識したフロー解析を抑止していないためである．

なお，表 5 において，すべての最適化を抑止した場合の例外発生検査数の増分が個々の最適化を抑止した場合の増分の合計を上回っているが，これは複数の最適化で除去できる例外発生検査が存在するためである．そのような例外発生検査は，単一の最適化を抑止しただけでは他の最適化が除去するため，増分に現れない．同じ理由から，表 4 においても，すべての最適化を抑止する場合のコードサイズの増分が個々の最適化を抑止した場合における増分の合計を上回っている．

```
0:   int getValue1(ExecEnv *ee, List *cell){
1:     int itemp = getValue2(ee, cell);
2:     if (exceptionOccurred(ee))
3:       goto RETURN;
4:   RETURN:
5:     return (itemp);
6:   }
```

図 6 2 返戻値法を意識したフロー最適化を getValue1() に適用した結果
Fig. 6 Optimized code of getValue1() by two return values method aware flow optimization.

4.2 Delay の効果

表 4 から，SH4 では Delay によってコードサイズを相加平均で 1.99% 削減できるのに対し，x86 では逆に 0.23%増えてしまうことが分かる．Delay の効果がプロセッサによって変わるのは，Delay 適用前の例外発生コード（関数呼び出し．たとえば図 2 下段の 10 行目と 34 行目のコード）と適用後の例外発生コード（例外発生フラグへの書き込み．たとえば図 4 のコード）がプロセッサごとに違うためである．Delay 適用前後における例外発生コードのアセンブリ表現の例を**図 7** に示す（最適化の影響で図 7 とは異なるコードになることもある）．図 7 は NullPointerException を発生するコードだが，他の例外も，関数名などの違いを除き，同様のコードで発生できる．

まず，Delay が SH4 に対して有効に働く理由について考察する．図 7 から，Delay 適用後の SH4 のコードサイズが 4〜6 バイトであることが分かる．コードサイズが変化しうるのは，ee がレジスタ上にない場合に，スタックを参照して ee を取得するコードが必要になるからだが，SH4 は組込み機器向けプロセッサとしては多くの汎用レジスタ（16 本）を持ち，ee は多くの場合レジスタ上にある．このとき Delay 適用後のコードサイズは 4 バイトになり，適用前（4〜10

Delay 適用	SH4 コード	SH4 コード長	x86 コード	x86 コード長
なし	; throwNullPointerException() mov @(定数プールオフセット，pc)，r0 jsr r0 （遅延スロット） ⋮ 定数プール： throwNullPointerException のアドレス	 2 2 (2) (4)	; throwNullPointerException() call throwNullPointerException	 5
コード長合計		4〜10		5
あり	; ee の取得（レジスタにあれば不要） mov.l @(EE_SLOTID，sp)，r1 ; 例外のクラスを表すコードの書き込み mov.b #3，r2 mov.l r2，@(SYNC_FIELD_OFFSET，r1)	 (2) 2 2	; ee の取得（レジスタにあれば不要） mov eax，EE_SLOTID[esp] ; 例外のクラスを表すコードの書き込み mov [eax+SYNC_FIELD_OFFSET]，3	 (4) 4
コード長合計		4〜6		4〜8

※コード長の括弧内の数値は不要になりうる命令や削減しうる領域の大きさを表す．

図 7 NullPointerException を発生するアセンブリコードの例
Fig. 7 Exmaples of assembly codes to generate NullPointerException.

バイト）より必ず小さくなる☆1．図 7 の適用前のコードでは，関数呼び出し前後に挿入されることがある，呼び出し元保存レジスタの内容の退避回復コードを計算に含めていないが，退避回復コードが発生した場合には，Delay の効果はより大きくなる．

一方，x86 では適用前のコードサイズが 5 バイトであるのに対し，適用後のコードサイズは 4〜8 バイトである．適用後のコードサイズが変化しうるのは ee がレジスタ上にない場合に，スタックを参照して ee を取得するコードが必要になるからだが，x86 ではレジスタ数が少なく ee がレジスタ上にある確率は必ずしも高くない．この結果，適用後のコードサイズが適用前より大きくなりうる．x86 でも呼び出し元保存レジスタの退避回復コードが発生することはあるが，レジスタ数が少ないため，その影響は SH4 ほど大きくならない．さらに，本評価では Delay を適用前の例外発生コードを，引数が 0 個の関数呼び出しとしたが，引数の数が少ないことも適用前の例外発生コード（関数呼び出し）のサイズを小さくすることに貢献している☆2．これらの理由から x86 では Delay によってコードサイズを削減できなかったと考える．

なお，例外オブジェクトの生成を catch まで遅延する技法は，Delay のほかに，Cierniak らが提案した最適化でも利用している[3]．ただし，Delay が例外オブジェクトを生成するための関数呼び出しをメモリ参照で代替してコードサイズを削減するのに対し，Cierniak らの最適化は catch 直後に破棄される例外オブジェクトの生成を省略して実行を高速化するものであり，最適化の内容は大きく異なる．

4.3 実行速度への影響

本論文で提案した最適化が実行速度に与える影響を評価した結果を**図 8** に示す．実行には，組込み機器向けプロセッサ Am5x86™ ☆3 133 MHz を搭載した PC と，Pentium® ☆4 III 450 MHz を搭載した PC を利用した（**表 6** 参照）．SPECjvm98 の問題サイズは 100 とし，ヒープサイズは初期値と上限値をともに 20 MByte とした．

表 6 評価に使用した PC の構成
Table 6 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

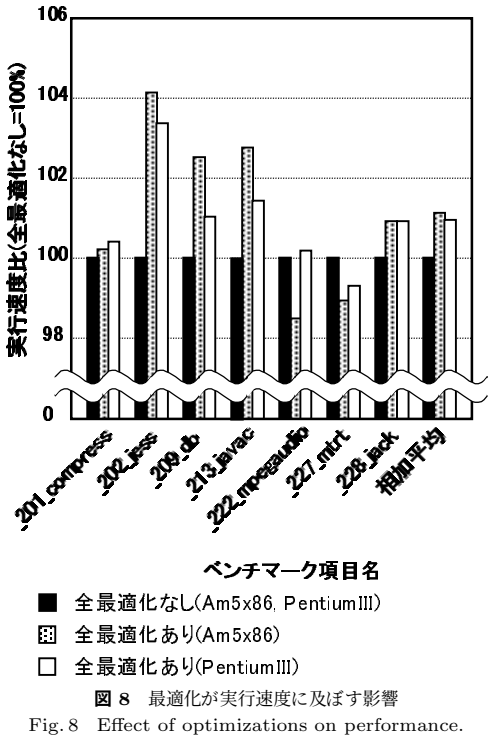

図 8 最適化が実行速度に及ぼす影響
Fig. 8 Effect of optimizations on performance.

図 8 の縦軸は全最適化を実施しない場合の実行速度を基準とする実行速度の比を表す．図 8 から，全最適化の適用により相加平均で Am5x86 では 1.14%，PentiumIII では 0.97%それぞれ実行を高速化できることが分かる．高速化の主な原因は，冗長な例外発生検査の除去により例外発生検査の実行回数が減少したことと，例外発生検査の下方移動によりメソッド呼び出しの返戻点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が生じたことにあると考える．

☆1 SH4 における Delay 適用前の例外発生コードのサイズは 4 バイトより大きい．なぜなら，適用前のコードサイズが変化しうるのは，遅延スロットが有効な命令で埋まらない場合があることと，関数のアドレスを収める定数プールのスロットを複数の関数呼び出しで共有することがあるためだが（SH4 では関数のアドレスや定数の値を収める定数プールをオブジェクトファイルの .text セクション内に配置するので，定数プールをコードサイズの計算に含める），後者の影響は，より多くの関数呼び出しでスロットを共有するほど小さくなるものの，0 にはならないからである．

☆2 例外を発生するには ee を参照する必要があるが，ee を引数として渡す必要は必ずしもない．なぜなら ee はシステムコールでスレッド番号を取得し，それを手がかりに逆算するといった手段でも求めうるからである．

☆3 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．

☆4 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．

5. 結　　論

コードサイズの削減は組込み機器向けコンパイラにとって重要な課題である．本論文では 2 返戻値法で例外処理を実現する Java2C トランスレータ向けのコードサイズ削減方法として，冗長な例外発生検査の除去や，下方移動による例外発生検査の集約，例外オブジェクトの生成を `catch` まで遅延する技法を提案した．また，相互排除なしで同期例外を投擲する方法や，同期例外と非同期例外を 1 回のメモリ参照で検出する方法を示した．SPECjvm98 による評価の結果，提案した技法により SH4 プラットホームにおけるコードサイズを相加平均で 5.21%削減できることが分かった．

参 考 文 献

1) 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).
2) Budimlic, Z. and Kenedy, K.: Optimizing Java — Theory and Practice, *ASC 1997 Workshop on Java for Science and Engineering Computation* (1997).
3) Cierniak, M., Lueh, G.-Y. and Stichnoth, J.M.: Practicing JUDO: Java under Dynamic Optimizations, *PLDI 2000*, pp.13–26 (2000).
4) Kolte, P. and Wolfe, M.: Elimination of Redundant Array Subscript Range Checks, *PLDI 95*, pp.270–278 (1994).
5) 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).
6) 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).
7) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks (1998). http://www.spec.org/osg/jvm98/
8) 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).
9) Yang, B.-S., Moon, S.-M., Park, S., Lee, J., Lee, S., Park, J., Chung, Y.C., Kim, S., Ebcioglu, K. and Altmannn, 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).
10) 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).
11) 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).
12) 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).

(平成 13 年 5 月 31 日受付)
(平成 13 年 7 月 31 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．