Java2Cトランスレータにおける例外処理の実現

千　葉　雄　司†

Java2C トランスレータの開発にあたっては，Java にあって C 言語にない機能の実現方法が問題になる．それらの問題の 1 つに例外処理がある．例外処理は Java のほかに C++なども提供する機能であり，プログラム実行中に例外的事象が発生した箇所から，それに対処するルーチン（ハンドラ）まで大域的にジャンプ可能にする．例外処理の実現方法には，表引き法，setjmp 法，2 返戻値法の 3 種類があるが，Java2C トランスレータで利用可能なものは後 2 者である．本論文では，setjmp 法と 2 返戻値法について，Java2C トランスレータで実現するにあたり問題となる点を指摘する．また，SPECjvm98 を用いた性能評価からこれらの方法を比較する．評価の結果，全ベンチマーク項目で 2 返戻値法が setjmp 法より実行を高速化し，その差が平均で 1.53%に達することが分かった．

Implementations of Exception Handling in a Java2C Translator

Yuji Chiba†

One of the problems in the development of Java2C translator is the way to implement features Java provides but C language does not. One of the features is exception handling, which not only Java but also C++ provides. Exception handling is a feature for jumping globally from the point where an exception is thrown to the code which handle the exception, and it has been implemented by following three methods: table method, setjmp method and two return values method. Java2C translator can exploit the latter two methods. This paper shows problems in implementing exception handling for Java by each of the two methods, and compare the performance of the methods using evaluation of the methods by SPECjvm98. The result showed that two return values method performs better than setjmp method in every benchmark item by 1.53% in average.

1. はじめに

Java™☆向け静的コンパイラの実現方法の 1 つに，C 言語を中間語として使う方法がある．この方法では，コンパイラを Java2C トランスレータと C コンパイラから合成し，次の手順でプログラムをコンパイルする．まず，Java2C トランスレータで，Java のソースコードあるいはバイトコードで記述したプログラムを，C 言語で記述したプログラムに変換し，次に，C コンパイラで処理して機械語に変換する．C 言語を中間語として使う方法には，コンパイラの移植性を高め，また，機械依存の最適化を既存の C コンパイラに実行させることで，コンパイラの開発にかかる手間を小さくできるなどの利点がある．Java2C トランスレータを利用した Java 向け静的コンパイラには，研究用から商用まで，これまでにいくつかの開発例がある[3),7),8)]．

Java2C トランスレータの開発にあたっては，Java にあって C 言語にない機能の実現方法が問題になるが，それらの機能の 1 つに例外処理がある．例外処理は，プログラム実行中に例外的事象が発生した箇所から，それ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．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．

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 1** のプログラムを例に具体的に述べる．`try` は，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 1 のプログラムではメソッド `boo()` の中に `try` の記述があるが，`try` に続く中括弧で囲んだ範囲が `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

† 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．

```
class Example{
  void boo(){
    try{
      this.foo();
    }
    catch(Exception e){
      // ハンドラ
    }
  }

  private synchronized void foo()
    throws Exception{
    while(true){
      this.woo();
    }
  }

  private void woo() throws Exception{
    throw (new Exception());
  }
}
```

図 1　例外処理の例
Fig. 1　An example of exception handling.

処理する例外のクラスを指定できる．たとえば boo() 内の catch 節では，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，例外を捕捉して catch 節内部（「ハンドラ」とコメントを打ってある箇所）を実行する．そうでない場合には，boo() 内の catch では例外を捕捉せず，対応する catch が現れるまで boo() の呼出元を再帰的にたどる．

throw は例外を投擲し，対応するハンドラまでジャンプする役割を果たす．図 1 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() の内部で Exception クラスの例外を生成して throw で投擲する．この結果，投擲した例外を捕捉するメソッド boo() 内のハンドラにジャンプすることになる．

例外処理の機能は Java のほかに 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．具体的には，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っておく．そして，例外が発生したら，実行中のメソッドに対応する表を引いて例外に対応するハンドラを探し，ジャンプする．実行中のメソッドが対応するハンドラを持たない場合には，見つかるまで実行時スタックを再帰的にたどって表引きを繰り返す．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[9]，Java のインタプリタ[6]，JIT（Just In Time）コンパイラなどが採用している[5]．

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．C++コンパイラのうち，C++のソースコードから C コードを経由して機械コードを生成するものが採用している[1]．

2 返戻値法　メソッドの返戻値を，通常の返戻値と例外の 2 個にする方法．メソッド呼び出しからの返戻点の直後に検査コードを挿入し，2 個目の返戻値を検査して例外が発生しているか調べる．例外が発生しているならば，ハンドラにジャンプする．ハンドラがないならば，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．CACAO という JIT コンパイラが採用していた[4]．

これらの実現方法のうち，最もオーバヘッドが小さい表引き法は，Java2C トランスレータでは実現できない．なぜなら，C 言語では 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できないからである．例外処理を持たない C 言語の代わりに，C++を中間語として使えば例外処理を簡単に実現でき，なおかつ C++コンパイラが表引き法で例外処理を実施している場合には，表引き法を利用可能になるのではないか，という発想もある．しかし，C++の例外処理を使って Java の例外処理を実現する方法には，次に示す理由から，少なくとも移植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分かるように，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 JVM 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 Virtual Machine（Java 仮想機械，JVM）を接続する場合に障害となりうる．

Java 向け静的コンパイラはいろいろな方法で実現できるが，既存の JVM を利用すると少ないコ

ストで開発できる．すなわち，Java 向けコンパイラの開発においては，コンパイラ本体だけでなく，スレッドやごみ集めといったサービスを提供するをライブラリ（実行環境）もあわせて開発する必要があるが，ライブラリの開発コストは小さくない．ライブラリの部分について既存の JVM に依存すれば，新規開発対象をコンパイラ本体に限定し，開発に要するコストを抑制できる．

ただし，既存のライブラリを利用する開発方法では，スレッドの実現が既存のライブラリに依存するが，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，既存のライブラリ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わたり，書き換えにかかる手間は小さくない．

これらの理由が原因であるか否かは定かでないが，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筆者の調査した限りでは存在しない．

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．本論文の目的は，この選択に必要な資料を提供することにある．すなわち，setjmp 法と 2 返戻値法を Java 向けに実現する際の問題点を示し，また，それぞれが実行速度に与える影響を評価，比較する．次章では Java2C トランスレータにおける setjmp 法の実現について述べ，3 章で 2 返戻値法の実現を示す．4 章ではそれぞれの実現を評価し，5 章で関連研究を示す．6 章は結論である．

2. setjmp 法

setjmp 法[1] は Cameron らが C++2C トランスレータで例外処理を実現する方法として提案したものだが，Java の例外処理もほぼ同様に実現できる．図 1 のコードを setjmp 法で C 言語に変換した結果を**図 2** に示す．図 2 の関数 `boo()` の内容から，setjmp 法による例外処理の実現について詳述する．setjmp 法では `try` ブロックに突入する際に `setjmp()` を実行し，例外が発生したら `longjmp()` で `setjmp()` を実施した時点に戻り，そこからハンドラにジャンプする．関数 `boo()` のコードでは `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☆を更新したうえで `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．

なお，setjmp 法では一部の自動変数を `volatile` 宣言する必要がある．ANSI C では `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．したがって `longjmp()` からの復帰後に `catch` 節内などから該当する自動変数を参照する場合には，その自動変数を `volatile` 宣言する．`volatile` 宣言は `longjmp()` からの復帰後にも自動変数が正しい値を持つことを保証する一方で，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．

`try` ブロックの入口にあるマクロ `enterTry()` では，最上位の `try` ブロックの更新と `setjmp()` のほかに，モニタ用のスタックポインタの値を記録している．この処理は Java の同期メソッドを正しく処理するために必要になる．同期メソッドとは排他制御を行うメソッドであり，具体的にはメソッドの出入り口でモニタの確保と解放を行う．Java のソースプログラムで `synchronized` と宣言したメソッドが同期メソッドになる．図 1 ではメソッド `foo()` を `synchronized` と宣言しており，これに対応する図 2 の C 言語のコードには，その出入り口にモニタを確保，解放する関数呼び出し `MonitorEnter()`，`MonitorExit()` がある．

setjmp 法で注意すべき点は，関数 `boo()` から関数 `foo()` を経て呼び出した関数 `woo()` で例外を投擲し，`longjmp()` で関数 `boo()` 中の `catch` 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とである．同期メソッドの問題は Java 固有のものだが，似た問題が C++の例外処理における局所オブジェクトのデストラクトについて発生しており，同じ方法で解決できる．関数 `boo()` が受け取る第 1 引数 `ee` を，実行中のスレッドに固有の資源を収める構造体へのポインタとする．問題を解決するには，まず，`ee` が参照する構造体の中に確保したモニタを記録するスタックを設ける．また，モニタの確保，解放に際してマクロ `pushMonitor()`，`popMonitor()` を実施し，

☆ 未脱出の try ブロックのうち，最後に入ったもの．

```
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{
      /* 再投擲 */
      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->exception)){
      exception = ee->exception;
      ee->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 2 setjmp 法による変換結果
Fig. 2 Program converted by setjmp method.

確保したモニタを ee 中のスタックに記録，消去する．そして，try ブロックへの突入に際してはその前処理マクロ enterTry() でスタックポインタの値を記録し，例外を投擲して longjmp() で catch まで大域ジャンプしたときには，記録したスタックポインタの値より上にあるモニタを解除する．関数 boo() で longjmp() から復帰した直後に ExitJumpedOverMonitors() を呼び出すのは，飛び抜けた同期メソッドのモニタを解放するためである．

例外処理の実現において Java 固有に考慮すべき問題には，同期メソッドのほかに，非同期例外の捕捉がある．Java では java.lang.Thread.stop() というメソッドを使って，あるスレッドが別のスレッドに非同期に例外を発生させることができる．非同期例外を捕捉する方法には様々な実現がありうるが，移植性が高いのはポーリングによる方法である．まず，ee の中に発生した例外を収めるフィールド exception を用意する．次に，フィールド exception を参照して例外を投擲するコードをプログラム中の複数の箇所に挿入する．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 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

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擲する．

非同期例外の発生をポーリングするコードの挿入箇所としては，ループ内の必ず通過する制御パスなどが考えられる．なぜなら，ループの中にポーリングのためのコードがないと，無限ループしたとき永遠に非同期例外を検出できなくなりうるからである．図 2 の関数 foo() では while ループの必ず通過する制御パスにポーリングのためのコードを埋め込んでいる．ここで注意すべき点はポーリングのためのメモリ参照を volatile と指定していることである．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見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．

3. 2 返戻値法

図 1 のコードを 2 返戻値法で C 言語に変換した結果を**図 3** に示す．図 3 のコードでは，通常の返戻値を return 文で返戻し，2 つめの返戻値（例外）を ee 中のフィールド exception を介して返戻する．

図 3 のプログラムの内容から，2 返戻値法による例外処理の実現について詳述する．2 返戻値法は投擲した例外を順次返戻していくことで，例外処理による大域ジャンプを実現する．具体的には，exception フィールドに投擲する例外へのポインタを書き込んだうえで関数呼び出しから返戻することで例外の投擲を実施する．関数呼び出しの直後には例外が発生したか検査するコードを配置し，exception フィールドが空でない場合にはハンドラにジャンプするか，関数から返戻するなど適切な動作をするようにしておく．図 3 のプログラムでは，関数 boo() 中の関数呼び出し foo(ee, this); の直後にある検査コードは，例外発生時にハンドラにジャンプする．関数 foo() 中の関数呼び出し woo(ee, this); の直後にある検査コードは，例外発生時にモニタを解放したうえで関数呼び出しから返戻する．

2 返戻値法の利点は 2 つある．まず第 1 に，try ブロックへの出入りにあたり，setjmp 法では必要になる前処理や後処理（マクロ tryEnter()，tryExit()）をいっさい必要としない．この原因の 1 つは，図 3 の関数 foo() で実行しているように，関数呼び出しから順次返戻する間に同期ブロックのモニタを解除できることにある．そして第 2 に，関数からの返戻時における exception フィールドの検査が非同期例外のポーリングを兼ねる．このことにより，非同期例外の検出だけを目的としたポーリング用のコードの挿入箇所を少なくできる．たとえば，図 3 の関数 foo() 中にある検査のコードはループの本体を実行するたびに必ず 1 回実行されるから，このループに非同期例外をポーリングするコードを追加する必要はない．

一方，2 返戻値法の問題点は，関数呼び出しから返戻するたびに exception フィールドを検査するオーバヘッドである．ただし，例外を返戻しない関数呼び出しについては検査のコードを省略できる．たとえば図 3 の関数 woo() の中にある関数呼び出し ExceptionConstructor(ee, exception); につ

```
void boo(ExecEnv *ee, JObject *this){
  JObject *exception;
  foo(ee, this);
  if (exception = ee->exception){
    if (IsInstanceOf(exception,
                     Exception)){
      ee->exception = NULL;
      /* ハンドラ */
    }
  }
}

void foo(ExecEnv *ee, JObject *this){
  MonitorEnter(this);
  while(true){
    woo(ee, this);
    /* 同期，非同期例外の検出 */
    if (ee->exception){
      goto EXIT;
    }
  }
EXIT:
  MonitorExit(this);
}

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

void throw(ExecEnv *ee,
           JObject *exception){
  LOCK(&(ee->exception));
  if (ee->exception != NULL){
    ee->exception = exception;
  }
  UNLOCK(&(ee->exception));
}
```

図 3 2 返戻値法による変換結果
Fig. 3 Program converted by two return values method.

いて，この関数が例外を返戻しないなら，直後にある検査のコードを除去できる．また，インライン展開によっても検査のコードを除去可能であり，これらの最適化によってオーバヘッドを軽減できる．

4. 評　価

本章ではベンチマークの実行結果を用いて，まず，setjmp 法に固有な最適化の効果を示す☆．次に，setjmp 法と 2 返戻値法について，どちらの方がオーバヘッドが小さいか比較する．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．

本論文では実行速度の評価に，次に示す共通の環境を利用した．評価対象は SPECjvm98 [2] とした．これは 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．SPECjvm98 の問題サイズは 100 とし，_201_compress と _213_javac の 2 項目についてはヒープ不足回避のためヒープ初期サイズ，上限サイズとも 128 Mbyte に指定した．実験機械には HITACHI3500/540MP（CPU：PA7200 100 MHz, メモリ：256 MByte, OS：HI-UX/WE2），Java 実行環境には JeanPaul [10] を用いた．各例外処理方式は，JeanPaul の Java2C トランスレータ上に実現した．静的コンパイル対象のクラスは，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求めた，ベンチマークの終了までにロードした全クラスを指定した．

4.1 setjmp 法に固有な最適化の評価

setjmp 法版の JeanPaul には，setjmp 法に固有な 4 種類の最適化を実装した．それぞれの最適化がプログラムの実行速度に及ぼす影響を評価する．

4.1.1 volatile 宣言の最小化

2 章で指摘したように，setjmp 法では一部の自動変数を volatile 宣言する必要がある☆☆☆．volatile 宣言する自動変数の選定方法として，簡易的には try ブロックを持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．そこで，volatile 宣言によるオーバヘッドを抑止する最適化として，volatile 宣言する自動変数を必要最小限にする機能を実現した．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．

この最適化がベンチマークプログラムの volatile 宣言する自動変数の数に与える影響を**表 1** に，実行速度に与える影響を**図 4** に示す．表 1，図 4 において，最適化なしとは try ブロックを持つメソッド内の全自動変数を volatile 宣言する場合を表す．表 1 から，最適化によって volatile 宣言する自動変数の

表 1 volatile 宣言の最小化による volatile 宣言数の変化
Table 1 Effect of minimizing volatile declarations on number of volatile declarations.

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	1,0,5312	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

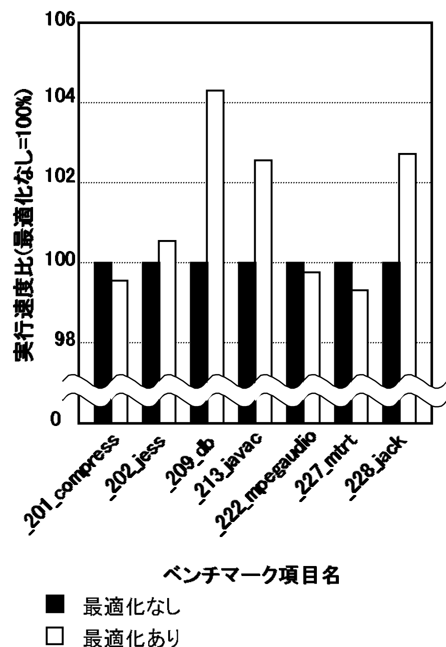

図 4 volatile 宣言の最小化による実行速度の改善
Fig. 4 Effect of minimizing volatile declarations on performance.

☆ 2 返戻値法向けの最適化としては，例外発生検査の除去がある．この最適化はインライン展開の際にあわせて実施する場合がほとんどであり，その効果を単離することが困難なので，本論文では 2 返戻値法向け最適化の評価については省略する．

☆☆ どのクラスをロードしたかログを出力するオプション．

☆☆☆ 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．

表 2 enterTry() の実行回数と最適化の影響
Table 2 Effect of optimizations against the execution count of enterTry().

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	253.813	1,959	1,882(3.93%)	1,644	882(46.35%)	11	0
_202_jess	401.160	1,240,894	30,599(97.53%)	707,879	705,674(0.31%)	1,835	0
_209_db	736.983	61,438,533	32,154(99.95%)	285,355	283,097(0.79%)	428	0
_213_javac	315.741	1,609,073	952,535(40.80%)	2,628,947	2,192,371(16.61%)	9,960	21,373
_222_mpegaudio	280.633	2,146	2,136(0.47%)	18,532	17,794(3.98%)	71	0
_227_mtrt	402.990	4,909	4,772(2.79%)	349,625	189,835(45.70%)	483	0
_228_jack	312.642	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

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分かる．また，図 4 から，この最適化によって _209_db と _213_javac，_228_jack の実行が，それぞれ 4.30%，2.55%，2.47%高速化することが分かる．

4.1.2 冗長な enterTry()，exitTry() の除去

例外を投擲する際に longjmp() でジャンプするのは，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() の代わりに単なる 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，発生しうる例外の投擲をすべて goto 文によって実施する場合である．そういった try ブロックについては，図 2 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および後始末を目的とするものなので，longjmp() が発生しないならば，これらのマクロは不要だからである．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ほかに，自動変数を volatile 宣言する必要が生じない利点もある．

マクロ enterTry()，exitTry() を必要としない try ブロックを検出し，これらのマクロと不要な volatile 宣言を除去する最適化を実現した．この最適化が enterTry() の実行回数に与える影響を評価した結果を**表 2** に，ベンチマークの実行速度に与える影響を評価した結果を**図 5** に示す．表 2 および図 5 から，_209_db について，この最適化が try ブロック入口での enterTry() の実行回数を 99.95%削減し，実行速度を 9.14%改善することが分かる．

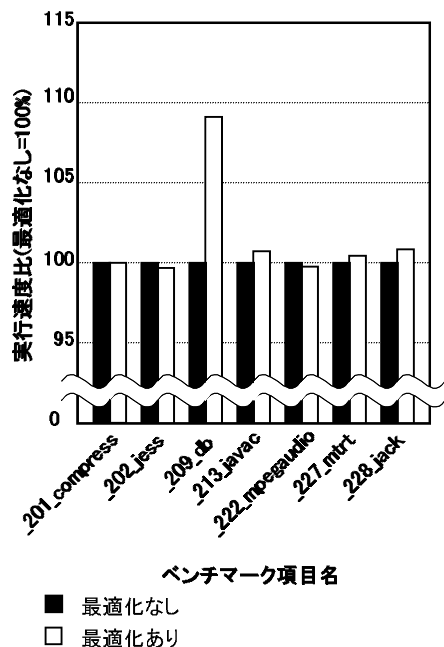

図 5 冗長な enterTry()，exitTry() の除去による実行速度の改善
Fig. 5 Effect of removing redundant enterTry() and exitTry().

4.1.3 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 JIT コンパイラ，それに 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，インタプリタあるいは JIT コンパイラが生成したコードから 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，その逆方向の呼び出しを行う場合には，例外処理方式の変換が必要になる．なぜなら，JeanPaul のインタプリタおよび JIT コンパイラが生成するコードにおける例外処理方式が setjmp 法とは異なるためである．

```
boolean invokeWithConversion(···){
  :
  exception = enterTry(ee, &buf);
  if ((int)exception == 0){
    setjmp 法で例外を処理するコードを呼ぶ;
  }else{
    ExitJumpedOverMonitors(ee, &buf);
    ee->exception = exception;
  }
  exitTry(ee);
  :
}
```

図 6 例外処理方式の変換
Fig. 6 Conversion of exception handling method.

JeanPaul のインタプリタおよび JIT コンパイラが生成したコードは，次の方法で例外処理を実現する．まず，変数 ee が参照するスレッド固有の資源を収める構造体の 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．この例外の伝播方法（exception フィールドを介した伝播）は図 3 の 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にならない．

図 6 に，インタプリタあるいは JIT コンパイラが生成したコードから 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 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 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．そこで setjmp 版 JeanPaul では，例外を発生しえない関数を呼ぶ場合には例外処理方式の変換に関する処理を省略する機能を設けた．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．この最適化が例外処理方式を変換をどれだけ省略にするか評価した結果を表 2 に示す．表 2 から，この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() の実行回数を 0.31〜46.35%削減できることが分かる．しかし，そもそもインタプリタあるいは JIT が生成したコードから静的コンパイル済みコードを呼び出す回数☆自体が少ないため，実行速度に与える影響は小さい（**図 7**）．

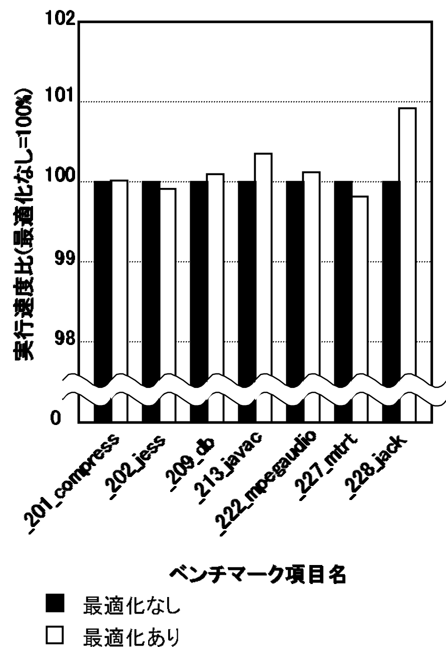

図 7 冗長な例外処理方式の変換の除去による実行速度の改善
Fig. 7 Effect of removing redundant conversion of exception handling method.

4.1.4 _setjmp() の利用

setjmp()，longjmp() は実装によってはシグナルマスクの退避と復帰を行うが☆☆，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．

setjmp()，longjmp() を使う場合と _setjmp()，_longjmp() を使う場合の実行速度の比較を**図 8** に示す．図 8 から，_setjmp() を使うことで，_213_javac と _228_jack の実行速度を改善できることが分かるが，

☆ 表 2 の，例外処理変化部で最適化なしの場合に enterTry() を実行する回数．

☆☆ setjmp() の実装（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．

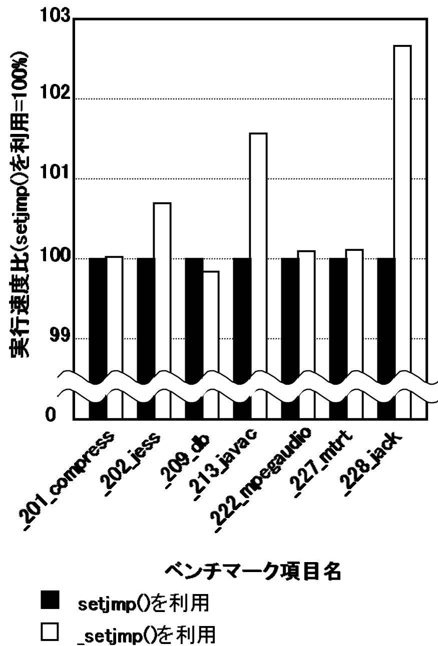

図 8 _setjmp() の利用による実行速度の改善
Fig. 8 Effect of using _setjmp() on performance.

これらは tryEnter() マクロ，すなわち setjmp() の実行頻度が高いベンチマーク項目である（表 2）.

4.2 setjmp 法と 2 返戻値法の比較

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た．実行速度の比較を**図 9** に示す．図 9 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ものとした．**表 3** に，例外発生検査の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ほかに，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．2 返戻値法ではさらに関数呼び出しからの返戻時に例外発生検査を実行するが，setjmp 法と比較して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を適用して関数呼び出しとともに例外発生検査を除去していることにある．インライン展開を適用しない関数呼び出しの後には例外発生検査を挿入するが，

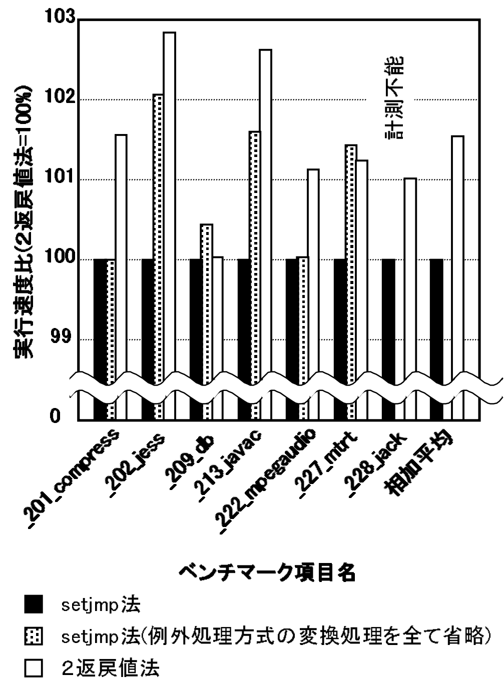

図 9 setjmp 法と 2 返戻値法の比較
Fig. 9 Comparison of setjmp method and two return values method.

表 3 例外発生検査の実行回数
Table 3 Execution counts of test for thrown exceptions.

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

これが非同期例外のポーリングを兼ねるために☆，結果として非同期例外のポーリング専用に例外発生検査を挿入する箇所が減り，例外発生検査の総実行回数が増加しない場合もある．

図 9 から，すべ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 2.84%，相加平均で 1.53%に達することが分かる．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

☆ setjmp 法で例外処理方式を変換する際の例外発生検査も非同期例外のポーリングを兼ねる．

変換は，JeanPaul においてインタプリタや JIT コンパイラが生成するコードが 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，インタプリタや JIT コンパイラの実現によっては不要になりうる．そこで，図 9 には setjmp 法と 2 返戻値法による実行速度のほかに，例外処理方式を変換する処理を強制的にすべて省略して setjmp 法でベンチマークを実行した場合の実行速度を示した．この場合の例外発生検査の実行回数は表 3 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計測できた．なお，2 返戻値法では例外処理方式がインタプリタあるいは JIT コンパイラが生成するコードと似ているため，例外処理方式の変換は不要である．

図 9 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分かる．

4.3　2 返戻値法によ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分かった．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，関数呼び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すべて排除した場合との実行速度の比較で行った．例外発生検査のコードを排除すると，実行中に例外を投擲するベンチマーク項目（`_213_javac` と `_228_jack`）は正常に動作しないが，これらの項目の評価は省略する．**図 10** 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` だけであることが分かる．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，より効率が良い（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 10 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行った．

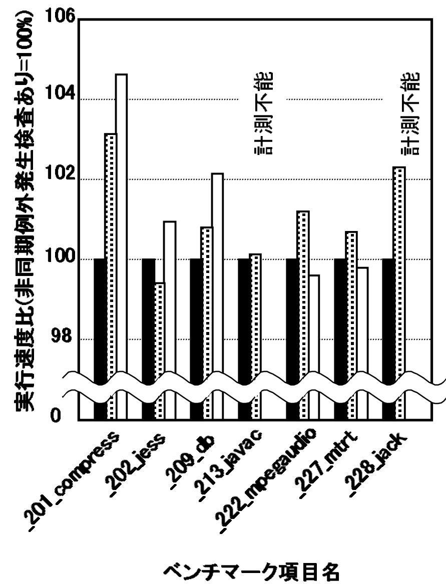

図 10　例外検出用ポーリングのオーバヘッド
Fig. 10　Overhead of polling for exceptions.

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

図 11　マイクロベンチマーク
Fig. 11　A micro benchmark.

表 4　マイクロベンチマーク実行結果
Table 4　Result of the micro benchmark.

例外検出コードの挿入	実行時間（ms）
あり	14.855
なし	10.862
実行速度比	136.76%

※配列長は 1000 とした．

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 11**）では大きく，36.76%に達した（**表 4**）．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 3.15%にとどまり，それほど大きくはならない（図 10）．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．

5. 関連研究

例外処理の実現は Ada や C++の時代から精力的に研究されており，本論文で取り上げた表引き法[5),9)]，setjmp 法[1)]，2 返戻値法[4)] はいずれも過去に考案された技法である．Java2C トランスレータについては，Proebsting らが setjmp 法による例外処理の実現を示しているが[8)]，2 返戻値法との比較はしていない．

Java における例外処理の性能評価については，Krall らが 2 返戻値法と表引き法の比較を試みている[5)]．しかし，本論文のように Java2C トランスレータで実現可能な setjmp 法と 2 返戻値法の比較を試みたものはこれまでにない．また，本論文では setjmp 法に関する最適化や，非同期例外を検出するためのポーリングが実行速度に及ぼす影響なども評価したが，筆者の知る限り，このような評価もこれまでにない．

6. 結　　論

Java2C トランスレータで採用できる例外処理の実現方式である setjmp 法と 2 返戻値法を取り上げ，それぞれを Java 向けに実現する際の問題点を指摘し，ベンチマークの実行速度から評価，比較した．評価の結果，全ベンチマーク項目で 2 返戻値法が setjmp 法より実行を高速化し，その差が平均で 1.53%に達することが分かった．また，2 返戻値法において非同期例外を検出するためのポーリングが，実用的なベンチマークで最大 3.15% 実行速度を低下させることが分かった．

参考文献

1) Cameron, D., Faust, P., Lenkov, D. and Mehta, M.: A Portable Implementation of C++ Exception Handling, *USENIX C++ Technical Conference*, pp.225–243 (1992).
2) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks (1998). http://www.spec.org/osg/jvm98/
3) Howard, R.: Developing and Deploying Server-hosted Applications with Java (1997). http://www.twr.com/java/white-paper.html
4) 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).
5) 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).
6) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, Addison Wesley, Reading, Mass. (1996).
7) 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).
8) 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).
9) Schilling, J.L.: Optimizing Away C++ Exception Handling, *ACM SIGPLAN Notices*, Vol.33, No.8, pp.40–47 (1998).
10) 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 3 月 12 日受付)
(平成 13 年 6 月 21 日採録)

千葉　雄司（正会員）

1972 年生．1997 年慶応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．