Javaプログラムにおいて例外による順序制約を投機的命令移動を用いて除去する方法

石崎一明[†] 稲垣達氏[†] 小松秀昭[†]

Java の型安全性はプログラムの健全さと信頼性を増加させるが，その性質を保証するために実行時例外検査が導入される．これらの検査は，プログラマが意図しない異常なプログラム終了を起こさないことを保証している．だが，これらの検査はハードウェアに関する例外を発生する命令との間に順序制約をもたらし，命令の並べ替えをともなう最適化の効率を低下させる．本論文では，Java のような型安全な言語において，有向グラフを用いて効率的に投機的命令移動を適用するための枠組みを述べる．例外検査の命令とハードウェアに関する例外を発生する命令の間に存在する順序制約を除去するために，この枠組みを用いて後者の命令に投機的命令移動を適用する．従来，分岐命令の制御フローによって表現されていた順序制約は，この枠組みでは命令間の辺として単純に表現される．この表現を用いて，投機的命令移動によるクリティカルパスの短縮を正確に見積もり，不必要な移動を抑制する．我々は，この方式を IA-64 用の Java Just-In-Time コンパイラに実装し，Itanium プロセッサ上において性能向上を得た．

Eliminating Exception Constraints in Java Programs Using a Speculation Technique

Kazuaki Ishizaki,[†] Tatsushi Inagaki[†] and Hideaki Komatsu[†]

While the type-safety of Java language is intended to increase the robustness and reliability of its programs, it requires frequent runtime checks to eliminate error-prone situations. Java exception checks are designed to ensure that any faulting instruction causing a hardware exception does not terminate the program abnormally. These checks, however, impose some contraints upon the execution order between an instruction potentially raising a Java exception and a faulting instruction causing a hardware exception. This prevents the effectiveness of instruction reordering optimizations. This paper describes a new framework to perform speculation for a ntype-safe language effectively using a direct acyclic graph. Using this framework, we apply a well-known speculation technique to a faulting load instruction to eliminate such constraints. We use an edge to represent exception constraints, and can accurately estimate a potential reduction of the critical path length and suppress unprofitable speculation using each edge. We have implemented the technique in our Java Just-In-Time compiler for IA-64, and measured the performance improvement on an Itanium processor.

1. はじめに

型安全性は，不正なメモリアクセスを起こさない，プログラマが意図しない異常なプログラム終了を起こさない，という性質を保証する．この性質は，セキュリティの面からも近年急速に重要性が増している．

Java では，型安全性が保証され，すべての例外はプログラムによって捕捉可能である．したがって，不正なアドレスをともなったメモリアクセス命令，アクセス例外，などハードウェア例外によってプログラムを異常に終了させる可能性がある命令（以下，**H-PEI** ⟨hardware-initiated potentially exception instruction⟩ と呼ぶ）の前に，ソフトウェアによる実行時例外検査によって，プログラムが異常終了しないことを保証する命令（以下，**S-PEI** ⟨software-initiated potentially exception instruction⟩ と呼ぶ）を挿入して，ハードウェア例外が発生する状況でもプログラムをソフトウェア的に安全に停止できなければならない．そのため，S-PEI の後に H-PEI を実行する，という命令間依存（以下，この命令間依存を**例外依存**と呼ぶ）が発生する．例外依存は，命令移動をともなう最適化の妨げとなる．

本論文では，Java のような型安全な言語において，

† 日本 IBM 株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan Ltd.

投機的移動前	**投機的移動後**

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

図 1 投機的実行の例
Fig. 1 An example of speculative execution.

投機的命令移動を用いて H-PEI と S-PEI の間の例外依存を効率良く除去する方法を提案する．この結果，S-PEI を越えた H-PEI の命令移動が可能になる．投機的命令移動[1)]は，依存を無視して実行の有無が決定する以前の位置へ命令を移動する．その結果，命令の実行の有無が決定する前に先行的に実行する，投機的実行が行われる．**図 1** に，その例を示す．投機的実行の際，実行されることのない不正な値をともなって命令が実行された結果（図 1 では p=0 のときの*p の参照），例外が発生する可能性がある．この例外は投機的実行されなければ発生しないものである．したがって，システムによって提供される**例外発生を抑制した投機的命令**を用いて投機的実行する．また，投機的実行の成否を調べて（図 1 では check 命令），失敗した場合には**復旧コード**（図 1 では Recovery 以下）を実行して，正しい値を再生成する仕組みも必要である．

投機的実行を用いて，例外依存を除去する手法が提案されている．S-PEI を比較命令と分岐命令に分解し，例外依存を S-PEI と H-PEI 間の制御依存で表し，スーパーブロックスケジューリング[2)] を用いて制御依存を越えて投機的命令移動を適用し，例外依存を除去する[3)]．この方法では移動前後の各ブロックの実行時間を見積もることはできるが，全体の実行時間を見積もることは簡単ではない．したがって，投機的命令移動による例外依存除去の効果が見積もりにくい．また，分岐命令が数多く生成されブロック数が増えるので最適化の際に効率が低下する問題が生じる．

我々の手法では，S-PEI を従来のように比較命令と分岐命令に分解することなく，1 つの命令として扱う．中間表現として，静的単一代入形式（以下，SSA 形式と呼ぶ）による有向非循環グラフ（以下，DAG と呼ぶ）を用いる．このとき，S-PEI と関連する H-PEI 間の例外依存を，DAG 上でブロック内の S-PEI と H-PEI 間の辺によって表現する．この辺を，**例外依存辺**と呼ぶ．この枠組みの上で，例外依存辺を切断し，H-PEI に対して投機的命令移動を適用することによって，例外依存による制約を除去できる．このように制御依存を越えた投機的命令移動と例外依存を越えた投機的命令移動を区別する我々の手法によって，以下の利点が得られる．

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

さらに，本論文では投機的命令移動を行う際に考慮すべき，投機的移動を行う命令列の選択，復旧コードの生成，についても示す．この命令列の選択方法では，復旧コードが正しく実行可能で，SSA 形式から通常形式に変換する際に余分なレジスタ間移動命令が生成されない．また復旧コードの生成方法は，メインコードのレジスタ割付けやコードスケジュールに与える影響を最小限にする．

我々は，本方式を Java Just-In-Time コンパイラに実装し，いくつかのプログラムを IA-64 [4)] プロセッサ上で実行して得られた結果を示す．小規模なプログラムでは最大 31%の性能向上を，SPECjvm98 ベンチマークでは最大 11%の性能向上を得た．

以下，2 章で関連研究を述べ，3 章で S-PEI と H-PEI 間の制約除去手法について述べる．4 章で投機的命令移動を適用する際のコンパイル手法について述べる．5 章では，評価実験によって本手法の効果を示す．6 章でまとめを述べる．

2. 関 連 研 究

Ebcioglu [5)] らは，アウト・オブ・オーダのバイナリ変換手法を提案している．この方式では，H-PEI であるロード命令を例外発生を抑制したロード命令に置き換えて制御依存を越えて投機的移動するとともに，ロード先のレジスタの名前を付け替える．さらに，元のロード命令があった位置に，値を元のレジスタに戻す移動命令が挿入される．この方式は，変換時間が短くバイナリ変換には適している．しかし，投機的移動可能な命令がロード命令だけであり，余計なレジスタ

a) サンプルプログラム

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

b) バイトコード

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

c) 中間表現

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

図 2 サンプルプログラムと中間表現
Fig. 2 A sample program and intermediate representations.

間移動命令が挿入される，という欠点がある．

Le [6] は，例外発生を抑制した命令を使わずに投機的命令移動を行う方法を提案している．この方法では，通常のコード内に安全点が生成されるため，例外が発生しない場合のクリティカルパスをのばすことがある．

Ju [7] らは，我々の手法と同様に SSA 形式による DAG を用いて，制御依存とデータ依存を越えて投機的命令移動を行う枠組みを提案している．彼らの手法は，C 言語を対象としていて，型安全な言語で発生する例外を扱っていない．また，実験結果は IA-64 のシミュレータ上で得られたものである．

Arnold [3] らは，Java の例外を VLIW 計算機で扱う手法を提案している．この手法では，S-PEI を比較命令と分岐命令に分解し，スーパーブロックスケジューリングと一般的なパーコレーションを適用している．これらは大域的な最適化であり，コンパイル時間を多く消費する．この実験結果もシミュレータ上で得られたものである．

Gupta [8] らは，S-PEI を例外検査命令と例外を投げる命令に分解し，例外検査命令間の制約を除去する手法を提案している．本手法は S-PEI と H-PEI 間の制約を除去するので補完的に適用可能である．

3. S-PEI と H-PEI 間の例外制約除去

本章では，我々が提案する S-PEI と H-PEI 間の例外制約除去のアルゴリズムを示す．まず，DAG の生成方法について述べる．次に，DAG 上での例外依存の制約除去のアルゴリズムについて述べる．

3.1 DAG の生成方法

本節では，Java バイトコードから例外依存を含む DAG の生成方法を示す．

Java バイトコードは，それが占める容量を小さくするために，1 命令で複数処理を行うものがある．たとえば，配列から整数要素を読む `iaload` バイトコード命令は，以下の処理を行う．

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

1 命令が複数処理を行う形式のままコンパイラの中間表現を生成すると，冗長な処理の削除が効果的にできない，スケジューリングの見積りが正確にできない，などの問題が生じる．したがって，1 バイトコード命令の複数処理を中間表現においてそれぞれ分解して表現する．

図 2 a) の Java プログラムは，図 2 b) のバイトコードで表現される．このバイトコードを，1 命令 1 処理の 4 つ組による中間表現に直すと図 2 c) となる．**太字の命令**は S-PEI，**斜体の命令**は H-PEI を示す．ここで，`nullcheck` 命令はオペランドの値が null であれば `NullPointerException` を投げる命令，`boundcheck` 命令はオペランドを比較した結果が成立しなければ `ArrayOutOfBoundsException` を投げる命令，`ld4` 命令は 4 byte の整数値をメモリから読む命令，である．⟨⟩ で囲まれた命令番号は，S-PEI が依存する H-PEI を示す．

この中間表現に対して，冗長な例外検査の命令を除去する．`newarray` 命令は処理が正常終了すれば必ず null でない配列オブジェクトが返されるので，`nullcheck` 命令は冗長であり除去できる．次に，この中間表現を SSA 形式に変換する．

最後に，この表現が持つ依存関係を DAG で表現したのが**図 3** a) である．辺が持つ時間は，命令の実行時間を示す．`ld4` 命令は 3 サイクルかかり，その他の命令は 1 サイクルかかる，と仮定している．`newarray` 命令と `boundcheck` 命令の間に，例外発生の順序を守るために順序依存を表す辺を張る．この辺は，例外発生順序を保つために張られるので，実行時間は 0 とする．さらに，`boundcheck` 命令と `ld4` 命令の間に，

a) 例外依存除去前の DAG と中間表現

```
N1: newarray   t1 = n
N3: ld4        t2 = 0[t1]
N4: boundcheck i < t2
N5: add        t3 = t1, 16
N6: shiftl     t4 = i, 2
N7: ld4        t5 = t4[t3]
N8: add        t6 = t5, 1
N9: ret        t6
```

b) 例外依存除去後の DAG と中間表現

```
N1: newarray   t1 = n
N3: ld4        t2 = 0[t1]
N4: boundcheck i < t2
N5: add        t3 = t1, 16
N6: shiftl     t4 = i, 2
N7: ld4(spec)  t5 = t4[t3]
N8: add        t6 = t5, 1
N10:check      t6, Recovery, t3,t4,t6
N9: ret
```

c) リストスケジューリング後の中間表現

```
N1: newarray   t1 = n
N6: shiftl     t4 = i, 2
N5: add        t3 = t1, 16
N7: ld4(spec)  t5 = t4[t3]
N3: ld4        t2 = 0[t1]
N8: add        t6 = t5, 1
N4: boundcheck i < t2
N10:check      t6, Recovery, t3,t4,t6
N9: ret
```

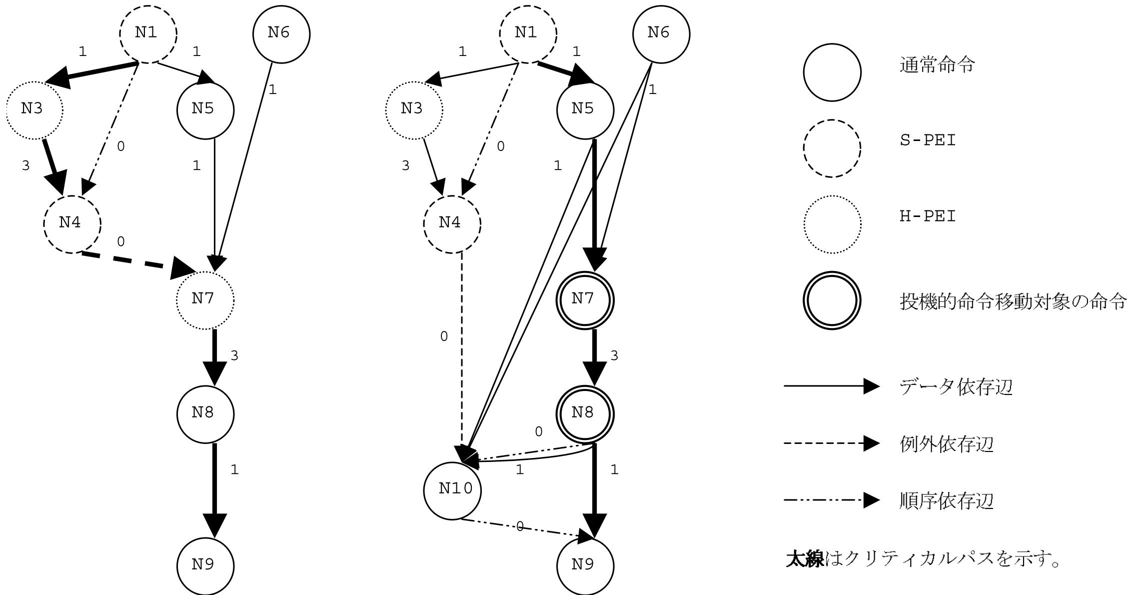

図 3 例外依存除去の適用例
Fig. 3 An example of eliminating exception constraints.

S-PEI と H-PEI 間の例外依存を表す辺を張る．この辺は，命令実行順序を保つために張られるので，実行時間は 0 とする．

従来，S-PEI を比較命令と分岐命令に分解して，例外依存を制御依存によって表現する方式が提案されている[3)]．我々の方式では，図 3 のように例外依存によってブロックが分割されることがないので，コンパイラの内部表現が使用する記憶領域を節約できる，ブロック内の命令数の増加によって投機的命令移動を含む様々な最適化を効果的に適用可能である，という利点がある．

3.2 例外依存の制約除去

本節では，例外依存の辺を投機的命令移動を用いて除去する方法を示す．S-PEI と H-PEI 間の例外依存辺の除去は以下の手順で行われる．

(1) **投機的命令移動の判断**： H-PEI の節点へ入力される例外依存辺と真データ依存辺で制約される 2 つの最早実行開始時間を比べる．もし，例外依存辺による最早実行開始時間のほうが遅いならば，投機的命令移動を行うために H-PEI を例外発生を抑制した投機的命令で置き換える．2 つの時間の差が，例外依存除去によって得られる時間短縮である．

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

(2) **例外依存辺の連結**： 投機的命令の実行が成功したかどうかを調べる `check` 命令（`N10`）を，DAG に挿入する．`check` 命令は，実行の成否を調べる値，復旧コードの分岐先，復旧コードで定義・参照される値，をオペランドとして持つ．

次に，S-PEI から投機的命令への例外依存を除去する☆．除去した例外依存を S-PEI から `check` 命令

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

へ張る．これによって，check 命令の実行結果によって実行されるかもしれない復旧コード内の H-PEI が，S-PEI が実行された後に安全に実行されることを保証する．

図 3 a) では，S-PEI の N4 から H-PEI の N7 への例外依存辺を除去し，check 命令の N10 へ張り替える．

(3)　**投機的移動を行う命令範囲の決定**： H-PEI を出発点とし，真データ依存をたどった結果の命令列を投機的命令移動を行う範囲とする．たどる間にメモリロードや整数の割り算命令などの H-PEI が現れたら，例外発生を抑制した投機的命令で置き換える．ただし，制御命令やメモリストア命令などの副作用を持つ命令がが現れたら，それ以降の依存をたどることを中止する．この方法については，4 章で詳しく述べる．

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

図 3 a) では，ロード命令である N7 と，単純な演算である N8 を投機的移動の対象とする．制御を変更する ret 命令は対象としない．

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

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

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

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

図 4　復旧コードの live-in，live-out 集合を求めるアルゴリズム
Fig. 4　An algorithm for calculating live-in and live-out sets of a recovery code.

図 3 a) では，N7 と N8 に対して live-in 集合 t3，t4 と live-out 集合 t6 が求められる．t3，t4 を定義する N5，N6 から N10 へ真データ依存辺を張る．

(5)　**メインコードと復旧コード間の順序依存辺の連結**： 復旧コードはすべての投機的移動を行う命令の後になければならない．また，復旧コードで再定義された live-out 集合の変数は投機的実行を行わない命令列で使用されるので，投機的実行を行わない命令列の前になければならない．したがって，投機的命令移動を行う命令列が check 命令の前に，投機的命令移動を行わない命令列が check 命令の後に，正しく配置されるように順序依存辺を張る．また，投機的移動を行う命令を推移的にたどり末端の命令が定義するオペランドを，check 命令が投機実行の成否を調べるオペランドとして，真データ依存辺を張る．

図 3 では，投機的移動を行う命令群の末端の N8 から check 命令の N10 へ順序依存辺を張り，N10 から投機的移動を行う命令群の次の命令 N9 へ順序依存辺を張る．また，N8 から N10 へ，投機実行の成否を調べるオペランド t6 のための真データ依存辺を張る．結果として，図 3 b) が得られる．

以上の処理が行われた後，リストスケジューリングを適用する．図 3 では，クリティカルパス長が例外依存除去前に 8 サイクルであったのが，例外依存除去後は 6 サイクルになっている．一般に配列の 1 要素

を参照する場合，NullPointerException と Array-OutOfBoundsException を調べるための 2 つの S-PEI と，配列長と配列要素を読み込む 2 つの H-PEI であるロード命令が生成されるので，本手法によってより大きな性能改善が期待される．

4. 投機的命令移動を行う際のコンパイル方法

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

4.1 投機的移動命令選択

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

- **副作用を持つ命令を選択しない**： 投機的移動を行った結果，例外発生を抑制した命令の実行結果が失敗する可能性がある．その結果を分岐命令やメモリストア命令などのように再実行が難しい副作用を持つ命令で利用することは，投機的実行が失敗したときに正しい再実行を保証できない．したがって，副作用を持つ命令は投機的移動を行わない．
- **循環グラフを生成する命令は選択しない**： もし循環グラフが生成されたならば，DAG 上でのスケジューリングが不可能になり，正確な実行時間の見積りや，正しい命令配置ができなくなる．Java プログラムの中間表現は，例外依存や順序依存を多く持つので，この条件について注意しなければならない．

図 5 に，循環グラフを生成する例を示す．この例において，N3 は副作用を持つ命令であると仮定し，N2 と N4 を投機的命令移動の対象とする．N1 から N2 への例外依存辺を除去して，N1 から新たに挿入した check 命令の節点 N6 に張り替える．N6 は，N4 と N5 の間に配置されなければならないので，N4 から N6 へ，N6 から N5 へ順序依存辺を張る．さらに，check 命令を実行した結果，復旧コードを実行すると N2 と N4 が再実行されるので，これらの計算結果を利用する N3 と N5 へ真データ依存辺を張る．また，復旧コードでは N3 の結果を利用するので，N3 から N6 へ真データ依存辺を張る．この結果，N3→N4→N6 で循環グラフが生成され，正しいコードが生成できなくなる．

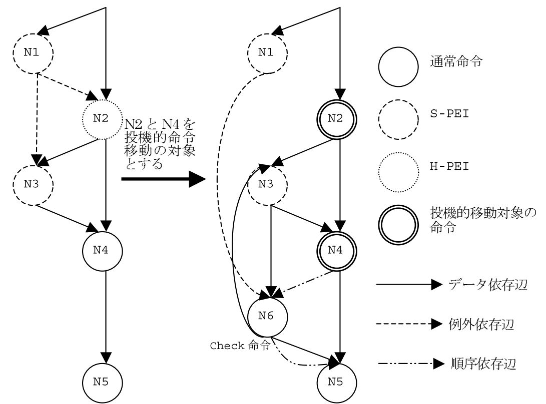

図 5 循環グラフの生成例
Fig. 5 An example of generating a cyclic graph.

このような場合，コンパイラは N2 のみを投機的移動対象命令とする．

- **変数の生存区間を資源以上に増やさない**： 投機的移動を行った結果，命令の配置が変わり変数の生存区間が変化することがある．その結果，同時に生存する変数の数が，CPU が持つレジスタ数を超えることがある．もし超えた場合は，物理レジスタ割付けの段階でスピルコードが生成される．DAG の節点を逆帰りがけ順に並べたとき，各節点をよぎる真データ依存辺の数（たとえば，図 3 b) の N7 の直後ならば 3☆）を数えると同時に生存する変数の数が分かる．スピルコードが生成されないようにするためは，この数がレジスタ数を超える移動になる命令は選択しない．
- **Web を構成する変数を複数参照しない**： Web を以下のどちらかの条件を満たす命令間を真データ依存で結んだ命令の集合と定義する[9]．

(1) 任意の命令で定義された変数が SSA 形式の ϕ 命令で使用されているときに，定義された変数を使用する命令．**図 6** では，S1→S2，S5，S7 にあたる．
(2) ϕ 命令で定義された変数を使用する命令．図 6 では，S7→S8 にあたる．

これらを満たす命令の集合は，直感的には ϕ 命令で結ばれた変数のリンクである．Web を構成する変数は通常形式への変換時に同一変数が割り振られる．

☆ N5→N10，N6→N10，N7→N8，である．

SSA 形式による投機的命令移動前

```
S1: add w1 = v1, 1
S2: if (w1 == 0) goto S7
S3: ld4 y1 = [z0]
S5: add w2 = y1, w1
S7: w3 = Φ(w1, w2)
S8: w4 = w3 + 1
```

SSA 形式による投機的命令移動後

```
S1: add w1 = v1, 1
S2: if (w1 == 0) goto S7
S3: ld4 (spec) y1 = [z0]
S5: add w2 = y1, w1
S6: chk.s w2, Recovery, z0, w1, w2
S7: w3 = Φ(w1, w2)
S8: w4 = w3 + 1
..
Recovery:
     ld y1 = (z0)
     add w2 = y1, w1
     goto S7
```

通常形式への変換後

```
S1: add w = v, 1
S2: if (w == 0) goto S7
S3: ld4 (spec) y = [z]
S4: mov t = w
S5: add w = y, w
S6: chk.s w, Recovery, z, wt, w
S8: w = w + 1
..
Recovery:
     ld y1 = (z)
     add w = y, wt
     goto S7
```

図 6 Web 上の変数が干渉する例

Fig. 6 An example of interferencing variables on a Web.

ϕ 命令は通常形式に変換すると代入命令にはならないので，Web を構成する変数の一部だけを名前書き換えを行うとレジスタ移動命令が生成される．

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

Web 上で変数の生存区間が干渉する例を図 6 に示す．この例において，`S1`，`S2`，`S5`，`S7`，`S8` は変数 `w` に関して Web を形成している☆．`S3` と `S5` に対して投機的命令移動を行うと，復旧コードで参照される `w1` は `S6` まで値を保持しなければならない．通常形式に変換する際に `w1`，`w2`，`w3`，`w4` は同一変数が割り当てられる．したがって，`S6` で生存している `w1` と `w2` が干渉するので，レジスタ移動命令 `S4` が生成される．したがって，この場合 `S3` のみを投機的命令移動の対象とする．

4.2 復旧コード生成

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

我々の方式では，3.2 節で述べたように，復旧コードがメインコードの最適化に与える影響を除外するために，復旧コードは中間表現上で陽には表現されない．投機的移動を行う命令に DAG 上で印を付け，コード生成時に印が付けられた命令を復旧コードとして複製する．複製の際に，例外発生を抑制した投機的命令は，元の H-PEI に置き換える．コード複製を行うので，メインコードと復旧コードのレジスタ使用は同一になる．さらに，リソース情報なども複製することで，復旧コード内でも同時実行可能な命令群を構成することが可能になり，コードの大きさも小さくできる．復旧コードの生成は以下の 4 段階からなる．

(1) **プロローグとエピローグの生成**： 復旧コードの中で定義されるレジスタは，live-in と live-out 集合のレジスタを除いて，復旧コードのプロローグでメモリに待避されて，エピローグでメモリから復元されるようにする．この結果，復旧コードの中で閉じるレジスタ定義参照は，メインコードのレジスタの生存区間に影響を与えない．復旧コードがメインコードに与えるレジスタの影響は，投機的移動を行った命令の中で使用されるレジスタが，復旧コードまで生存区間が伸びることだけである．復旧コードで定義されたレジスタのうち live-out 集合に含まれるものはメインコードで使われるので，復旧コード実行前の値を保存してはならない．また，live-in 集合のうち復旧コードで参照が終了するものは保存する必要がなく，メインコードでも参照されているものは復旧コードでは定義されないのでやはり保存する必要がない．つまり，復旧コードで定義されたレジスタから live-in・live-out 集合を除いたものを，復旧コードの前後で保存する．よって，投機的移動を行う命令群に対応する復旧コードで待避・復元されるレジスタの集合は，**図 7** によって求めら

☆ S1→S2，S1→S5，S1→S7，S5→S7，である．

れる．

図 8 の例では，投機的移動を行う命令群は 2 つある．1 つは I1，I2，I3 で，もう 1 つは I2，I5，I6 である．後者のための復旧コード Recovery2 において，live-in レジスタ集合は {r11}，live-out レジスタ集合は {r12，r23}，復旧コードで定義されるレジスタ集合は {r12，r22，r23} である．よって，待避・復元されるレジスタ集合は {r22} となり，I17，I21 が生成される．

(2) **復旧コードの複製**： 投機的移動を行う命令列を，エピローグとプロローグの間に複製する．命令を複製する際に，例外発生を抑制した投機的命令を元の H-PEI に置き換える．これによって，投機的実行による失敗した実行結果は，復旧コードで正しい実行結果で上書きされるので，正しい実行を保証できる．

図 8 の例では，I1，I2，I3 から I11，I12，I13 が生成され，I2，I5，I6 から I18，I19，I20 が生成される．

(3) **同時実行可能な命令群の複製**： IA-64 のような VLIW 計算機では数命令が同時実行可能であり，この同時実行可能な命令群を構成することによってプログラムから並列性を抽出する．一般に，分岐命令は同時実行可能な命令群の先頭命令へ分岐する．たとえば，復旧コードからメインコードへ戻る場合，check 命令の次の命令群へ戻る．

check 命令が同時実行可能な命令群の最後尾にある場合は，check 命令から復帰コードへ分岐して，メインコードへ次の命令群へ分岐命令で復帰しても正しく実行が再開できる．しかし，check 命令が同時実行可能な命令群の途中にある場合，そこから復旧コードへ分岐して，メインコードへの次の命令群へ復帰すると，check 命令と同一命令群にある後続命令が実行されない．よって，実行が正しく再開できない．この問題は，check 命令を含む命令群の後続命令を，対応する復旧コードの末尾に複製することで解決できる．

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

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

sp = kl ∩ ‾(li ∪ lo)‾
```

図 7 復旧コードで待避・復元されるレジスタの集合を求めるアルゴリズム

Fig. 7 An algorithm for calculating spill and fill sets of a recovery code.

投機的移動前	**投機的移動後**
I1: ld4 r2 = [r1]	I1: ld4(spec) r2 = [r1] //同時実行群 0
I2: ld4 r12 = [r11]	I2: ld4(spec) r12 = [r11] // 〃
I3: add r21 = r2, r12	I3: add r21 = r2, r12 // 〃
	I4: check r21, Recovery1 //同時実行群 1
I5: add r22 = r12, 1	I5: add r22 = r12, 1 // 〃
I6: shiftr r23 = r22,2	I6: shiftr r23 = r22, 2 // 〃
	I7: check r23, Recovery2 //同時実行群 2
I8: st [r21] = r23	I8: st [r21] = r23 // 〃
I9: mov ... = r12	I9: mov ... = r12 // 〃
I10: mov ... = r2	I10: mov ... = r2 //同時実行群 3

```
Recovery1:
I11: ld r2 = [r1]            //I1 の複製
I12: ld r12 = [r11]          //I2 の複製
I13: add r21 = r2, r12       //I3 の複製
I14: add r22 = r12, 1        //I5 の複製
I15: shiftr r23 = r22, 2     //I6 の複製
I16: goto I7

Recovery2:
I17: spill r22
I18: ld r12 = [r11]          //I2 の複製
I19: add r22 = r12, 1        //I5 の複製
I20: shiftr r23 = r22, 2     //I6 の複製
I21: fill r22
I22: st [r21] = r23          //I8 の複製
I23: mov ... = r12           //I9 の複製
I24: goto I10
```

図 8 復旧コードの生成

Fig. 8 Recovery code generation.

表 1 例外依存辺を持つ DAG と持たない DAG 表現の比較
Table 1 Comparison of representations between DAGs with and without exception edges.

	例外依存を制御依存で明示的に表した DAG	例外依存辺を持つ DAG
DAG のブロックの総数	37,327	142,679
1 つのブロック内に含まれる平均命令数	1.23	4.71
DAG のブロック間の辺の総数	262,791	54,669
1 つのブロック内に含まれる依存辺の総数	80,190	145,952

図 8 の例では，I4，I5，I6が同時実行可能な命令群であるとする．このとき，I4 から分岐した Recovery1 は I5 へ戻りたいが，命令群の途中であるため不可能である．したがって，同一命令群の後続命令 I5，I6 を Recovery1 の末尾に I14，I15 として複製する．Recovery2 に関する I22，I23 も同様である．

(4) **メインコードへ戻るジャンプ命令の生成**： 復旧コードの最後尾に，メインコードにある check 命令の次の命令群の先頭命令への分岐命令を生成する．図 8 の Recovery1 の例では，I16 によって次の命令群の先頭命令 I7 へ戻る．

5. 実験結果

本章では，本論文で提案した方式を実験によって評価した結果を示す．評価項目は，以下の 3 つである．

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

実験のために，IBM Developers Kit for Windows on Itanium，Java Technology Edition，Version 1.3.1 の Just-In-Time コンパイラ[10] に，本論文で述べた例外依存の除去方式を実装した．すべての測定は，IBM 社の IntelliStation Z Pro モデル 6894（Itanium 800 MHz × 2，メモリ 2 GB）で Windows を使用して行った．

5.1 中間表現の記憶域効率

本節では，例外依存辺を持つ DAG による中間表現の効率を評価する．

例外依存辺を持つ DAG と，例外依存を制御依存で明示的に表現した DAG における☆，ブロックや辺の統計を**表 1** に示す．評価には，SPECjvm98[11] に含まれる 7 つのプログラム（compress，jess，db，javac，mpegaudio，mtrt，jack）を size=100 で実行したものを使用した．DAG による中間表現では，冗長な例外検査の除去は十分行われている．

表の 2，3 列目は，例外依存辺を持つ DAG 表現により，DAG が持つブロックの数が約 1/4 になり，ブロックに含まれる命令数が約 4 倍になったことを示している．これにより，ブロック内最適化の適用機会が増える可能性がある．また，例外依存制約除去に関する投機的命令移動を，ブロック内の例外依存辺の除去によって適用できる．したがって，最適化にかかる時間を短縮可能である．これは，コンパイル時間を多くは費やせない実行時コンパイラにとっては特に重要である．

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

☆ 実際には，辺や節点数を表現に応じて増減して見積もった．

表 2 小規模プログラムの実行性能向上
Table 2 Runtime performance improvements for the micro benchmarks.

	全点対間最短距離問題	行列積
選択的に例外依存を除去	1.02	1.31
すべての例外依存を除去	1.00	1.28

5.2 小規模プログラムにおける性能向上

本節では，2 つの小規模なプログラムを実行した結果を示す．プログラムは，全点対間最短距離問題，行列積，を用いた．すべて二次元配列を参照するプログラムである．この理由は，Java では多次元配列を参照する際，Java 仮想マシンの配列の実装による制約から，2 次元目以降の配列参照に関する S-PEI を最適化によって除去することが非常に難しい．したがって例外依存が残り，本手法を適用する機会が多いことが期待されるからである．これらのプログラムでは，カーネルでは NullPointerException，ArrayOutOfBoundsException は発生しない．

表 2 に，例外依存の除去を行わない場合に対する，例外依存除去を適用した場合の性能向上を示す．例外依存の除去を選択的に行った場合，2%〜31%の性能向上が得られた．カーネルコードでは S-PEI による例外検査が多いので，すべてのプログラムで例外依存の除去の効果が確認できた．

また，すべての例外依存を除去した場合，選択的に

行った場合よりも性能向上が低い．これは以下の理由による，と考えられる．現在の Itanium プロセッサの実装では，例外発生を抑制した投機的ロード命令は TLB ミスを発生するとロードが失敗したとする[12)]．したがって，正しいアドレスが与えられても TLB ミスが起きるとロード失敗となり，復旧コードが実行される割合が増加する．したがって，本方式の実行時間短縮の正確な見積りによる例外依存除去の適用判断が有効であると思われる．

表 3 に，例外依存の除去を行わない場合に対する，例外依存除去を適用した場合のカーネルのコード増分を示す．コードの増分は 32〜64%で，最も増分が多いのは，性能向上が最も大きい行列積の場合である．コード増分の主な理由は復旧コードの生成によるもので，メインコードのクリティカルパスに影響は与えていない．したがって，性能向上が大きい行列積でコード増分が大きいということは，投機的移動が行われた命令数が多い，ということを示している．

5.3 SPECjvm98 ベンチマークにおける性能向上

本節では，SPECjvm98 ベンチマークを `size=100` で実行した結果を示す．これらのプログラムでも，ベンチマーク中は `NullPointerException`，`ArrayOutOfBoundsException` は発生しない．

表 4 に，例外依存の除去を行わない場合に対する，例外依存除去を適用した場合の性能向上を示す．例外依存の除去を選択的に行った場合，0〜11%の性能向上が得られている．特に，`compress`，`mpegaudio`，`mtrt` という配列参照が多い，つまり S-PEI が多いベンチマークで性能が向上している．

特に `mpegaudio` は，カーネルループで 2 次元配列の参照が頻繁に行われるので，性能向上が大きい．**図 9** に，`mpegaudio` のカーネルループの一部を示す．配列 `C` の 1 次元目はループ内で変化する値である．2 次元目は，その値が指す 1 次元配列から読み込まれる．したがって，冗長な例外検査を除去する最適化を用いても，2 次元目を参照する際の S-PEI を除去することができない．したがって，`mpegaudio` では本手法の効果が大きい．実際，カーネルループのクリティカルパスの長さは 31 クロックから 25 クロックに減少している．

表 5 に，例外依存の除去を行わない場合に対する，例外依存除去を適用した場合のカーネルのコード増分を示す．コードの増分は 1〜11%である．プログラム全体から見ると，`mpegaudio` の場合を除くとコード増分はそれほど多くない．最も増分が多いのは，性能向上が最も大きい `mpegaudio` の場合である．これも，投機的移動が行われた命令数が増加し，結果として性能向上が大きい，ということを示している．

6. ま と め

本論文では，以下の成果を示した．

- 型安全な言語において例外依存を越えた投機的命令移動を効率良く行う枠組みの提案．
- 投機的移動を行う命令の範囲の決定方法と補償コードを生成する方法の提案．
- 提案手法を Java コンパイラに実装し，例外依存除去の効果を IA-64 プロセッサ上で確認．

本論文で提案した手法によって，プロセッサが持つ投機的実行機能を適切に利用する機会を増やし，実際にベンチマークで最大 11%の性能向上を確認できた．

表 3 小規模プログラムのコード増分
Table 3 Static code size expansion for the micro benchmarks.

	全点対間最短距離問題	行列積
カーネルコード増分	1.32	1.64

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

図 9 `mpegaudio` のカーネルループの一部
Fig. 9 A part of a kernel loop of `mpegaudio`.

表 4 SPECjvm98 の実行性能向上
Table 4 Runtime performance improvements for SPECjvm98 benchmarks.

	compress	jess	db	javac	mpegaudio	mtrt	jack
性能向上比	1.04	1.00	1.00	1.00	1.11	1.03	1.00

表 5 SPECjvm98 におけるコード増分
Table 5 Static code size expansion for SPECjvm98 benchmarks.

	compress	jess	db	javac	mpegaudio	mtrt	jack
コード増分	1.03	1.01	1.02	1.01	1.11	1.02	1.02

謝辞 本研究を進めるにあたり，貴重なご意見をいただいた日本 IBM 東京基礎研究所のネットワーク・コンピューティング・プラットフォーム・グループの皆様に深く感謝いたします．

参 考 文 献

1) Mahlke, S.A., Chen, W.Y., Bringmann, R.A., Hank, R.E., Hwu, W.W., Rau, B.R. and Schlansker, M.S.: A model for compiler-controlled speculative execution, *ACM Trans. Comput. Syst.*, Vol.11, No.4, pp.376–408 (1993).
2) Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, G.E.T., Haab, J.G.H. and Lavery, D.M.: The superblock: An effective technique for VLIW and Superscaler Compilation, *Journal of Supercomputing*, Vol.7, No.1, pp.229–248 (1993).
3) Arnold, M., Hsiao, M.S., Kremer, U. and Ryder, B.: Exploring the interaction between Java's implicitly thrown exceptions and instruction scheduling, *International Journal of Parallel Programming*, Vol.29, No.2, pp.111–137 (2001).
4) Intel Corp.: *IA-64 Application Developer's Architecture Guide*. available at http://developers.intel.com/design/ia64/downloads/adag.htm
5) Ebcioglu, K. and Altman, E.R.: DAISY: Dynamic compilation for 100% architectural compatibility, *Proc. International Symposium on Computer Architecture*, pp.26–37 (1997).
6) Le, B.C.: An Out-of-Order Execution Technique for Runtime Binary Translators, *Proc. International Conference on Architectural Support for Programming Language and Operating Systems*, pp.151–158 (1998).
7) Ju, R.D., Nomura, K., Mahadevan, U. and We, L.: A Unified Compiler Framework for Control and Data Speculation, *Proc. Conference on Parallel Architectures and Compilation Techniques*, pp.157–168 (2000).
8) Gupta, M., Choi, J.D. and Hind, M.: Optimizing Java Programs in the Presence of Exceptions, *Proc. 14th European Conference on Object-Oriented Programming — ECOOP '00*, pp.422–446 (2000).
9) Sreedhar, V.C., Ju, R.D., Gillies, D.M. and Santhanam, V.: Translating Out of Static Single Assignment Form, *Static Analysis Symposium*, LNCS 1694, pp.194–210 (1999).
10) 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).
11) The Standard Performance Evaluation Corp.: SPECjvm98 Benchmarks. available at http://www.spec.org/osg/jvm98/
12) Intel Corp.: *Itanium Processor Microarchitecture Reference*. available at http://developers.intel.com/design/ia64/downloads/245474.htm

(平成 14 年 2 月 18 日受付)
(平成 14 年 5 月 16 日採録)

石崎　一明（正会員）
1992 年早稲田大学大学院理工学研究科修士課程修了．同年日本 IBM（株）入社．以来，東京基礎研究所において，並列処理，最適化コンパイラに関する研究に従事．

稲垣　達氏
1970 年生．1995 年東京大学大学院理学系研究科情報科学専攻修士課程修了．1998 年東京大学大学院理学系研究科情報科学専攻博士課程退学．同年日本 IBM（株）入社．現在日本 IBM 東京基礎研究所先任研究員．JAVA Just-In-Time コンパイラの開発に従事．

小松　秀昭（正会員）
1960 年生．1985 年早稲田大学理工学研究科電気工学専攻修了．同年日本 IBM（株）東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報科学）．