例外依存関係を越える部分冗長性除去

大平 怜[†] 平木 敬[†]

実行時の安全を保証するための例外機構は一方で速度低下の原因となるため，部分冗長性除去（Partial Redundancy Elimination; PRE）で上方移動を用いて不要な例外命令を削除することが有効である．しかし我々はプログラムの意味を保つために例外命令どうしの順序関係である例外依存関係を保つ必要がある．したがって，従来の部分冗長性除去では例外命令の上方移動が阻害されることが多い．本研究で我々はプログラムの意味を保存しつつ例外依存関係を越える部分冗長性除去，Sentinel PRE を提案する．Sentinel PRE は例外依存関係を無視して上方移動を行い，その後で高速な解析により例外順序の入れ替わりを検出する．順序が入れ替わった例外命令で例外が起きた場合，プログラムの意味を保つために上方移動する前の状態に脱最適化でコードを戻す．現実のプログラムで例外が起きることは稀であるため，ほとんどの場合は上方移動により最適化された高速なコードが実行される．Sentinel PRE は特別なハードウェアのサポートには依存せず，動的なコード書き換えにより脱最適化を実現する．我々は Sentinel PRE を Java の実行時コンパイラに実装して実験を行い，Java Grande Benchmark 中の heapsort プログラムで 8.4%の性能向上を得た．

Partial Redundancy Elimination beyond Exception Dependency

Rei Odaira[†] and Kei Hiraki[†]

Exception mechanism guarantees runtime robustness, but results in performance degradation. Thus, it is effective to remove redundant excepting instructions by Partial Redundancy Elimination (PRE), which uses hoisting of instructions. However, we must preserve ordering constraints between excepting instructions, which we call exception dependencies, in order to keep the semantics of the program. Therefore, existing PRE algorithms cannot hoist many excepting instructions. In this work, we propose Sentinel PRE, a PRE algorithm which overcomes exception dependencies and at the same time keeps the semantics. Sentinel PRE first hoists excepting instructions without considering exception dependencies, and then detects exception reordering by fast analysis. If exception occurs at a reordered instruction, it deoptimizes the code into the one before hoisting. Since we rarely encounter exception in real programs, the optimized code is executed in almost all cases. Sentinel PRE does not rely on special hardware support, and performs deoptimization by runtime code patching. We implemented Sentinel PRE in a Java just-in-time compiler and conducted experiments. The results show 8.4% performance improvement in “heapsort” program in Java Grande Benchmark.

1. はじめに

近年，Java 環境をはじめとして実行時の安全性を保証するために例外機構を用いる実行環境が増えている．**図 1** に Java における例を示す．(a) ではオブジェクト a のフィールド field1 の値をロードするが，その直前に a のヌルチェックを行う．(b) では配列 a の i 番目の要素をロードするが，その前に a のヌルチェックと添字の範囲チェックを行う．しかし実行速度の観点では例外を起こす可能性のある命令（Potentially Excepting Instruction; PEI），すなわち nullcheck や boundcheck などは速度低下の原因となり，また，すべてのメモリアクセスの安全性を保証するために用いられるので冗長な PEI がプログラム中に頻出する．本研究で我々はコンパイラによる最適化で冗長な PEI を削減することを目的とする．

最適化コンパイラが用いる冗長性除去の中で部分冗長性除去（Partial Redundancy Elimination; PRE）は有効な手法として知られている[17),18),21)]．**図 2** (a) では実行パスが左から来た場合にのみヌルチェック命令 3 とロード命令 4 が冗長となる．この場合，PRE は図 2 (b) に示すようにヌルチェックとロードを 3，4 の位置から 5，6 の位置まで上方へ移動し，元の位置の命令を削除する．このように，PRE は部分冗長な

† 東京大学情報理工学系研究科コンピュータ科学専攻
Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo

(a)	(b)
nullcheck a	nullcheck a
x:=a.field1	t:=arraylength a
	boundcheck t, i
	x:=a[i]

図 1 Java における例外
Fig. 1 Example of exceptions in Java.

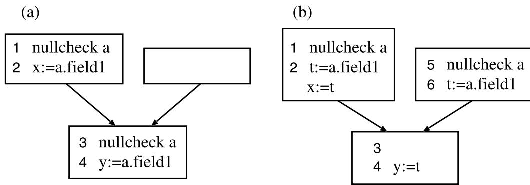

図 2 PRE による最適化の例
Fig. 2 Example of optimization by PRE.

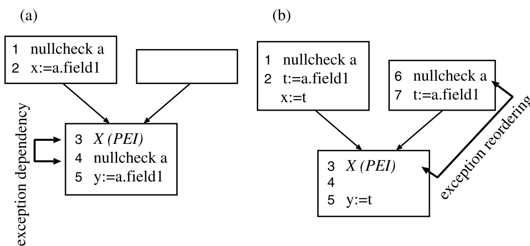

図 3 例外依存と例外順序の入れ替わりの例
Fig. 3 Example of exception dependency and exception reordering.

命令を上方へ移動（hoisting）して実行パス（図 2 の場合は左からのパス）から追い出すことで冗長性除去を実現する．

しかし，例外機構を用いる実行環境では一般に最適化によって実行時に起きる例外が変化してはならないため，PRE が用いる上方移動は制限を受ける．**図 3** (a) では図 2 (a) と異なり，命令の上方移動の範囲内に別の PEI X が存在する．このプログラムにおいて変数 a がヌル，かつ 3 の位置で例外 X が起きると仮定すると，実行が右のパスから来たならば発生する例外は X である．しかし，PRE が図 3 (b) のように最適化したとすると，発生する例外は X ではなく変数 a に関するヌルチェック例外となる．PEI を対象とした従来の PRE [15),16)] は例外の発生順を変えないためにこのような PEI の上方移動を行わない．我々は例外に関するプログラムの意味を保つための依存関係を「例外依存（exception dependency）」と呼ぶ．

このように例外依存を守ることで実行の正しさは保証されるが，そのかわりに除去できない冗長性がプログラム中に多数残る．なぜならば PEI は前述のようにすべてのメモリアクセスに付随するため，それらの PEI によって多くの箇所で上方移動が阻害されるからである．また，Java などではプログラム中のほとんどの PEI をヌルチェックと範囲チェックが占めるが，正常に動作する現実のプログラムでこれらの例外が発生することは稀である．稀にしか起きない例外発生時の実行の正しさを保証するために，例外が起きないほとんどの場合の実行速度が犠牲にされることになる．以上より，例外機構を用いたプログラムを高速化するためには，従来の PRE では除去できない，例外依存を越える冗長性を除去する手法が求められる．

本研究で我々はプログラムの意味を保存しつつ例外依存関係を越える部分冗長性除去，Sentinel PRE を提案する．Sentinel PRE はまず例外依存関係を無視して命令の移動を行い，その後で解析により例外順序の入れ替わりを検出する．順序が入れ替わって移動した PEI で実行時に例外が起きた場合，その場で実際の例外は発生させない．そのかわり，移動した PEI の元の位置（sentinel）に最適化前と同一の PEI を動的に書き込んで実行を続ける．結果として関数は脱最適化により最適化前の状態に戻ったことになるため，実行時の例外順序は維持される．ただし，ほとんどの場合は移動した PEI で例外が起きることはないので，冗長性が除去された最適化後の状態で実行される．

本手法の新規性は以下の点である．

- 特別なハードウェア命令のサポートに依存せず，コンパイラによる解析と実行時コード書き換えの協調によって例外依存関係を越える部分冗長性除去を実現した．
- Java 環境などで広く用いられている実行時コンパイラ上に実装される場合のために，例外順序の入れ替わりを高速に検出するアルゴリズムを開発した．

以降では 2 章で Sentinel PRE の動作の概略を例を用いて説明し，3 章で Sentinel PRE のアルゴリズムを示す．Sentinel PRE の正当性は付録で証明する．4 章でマイクロベンチマークと現実的なマクロベンチマークを用いた実験結果を示し，5 章で関連研究を述べる．最後に 6 章でまとめる．

2. Sentinel PRE の動作例

図 3 の例に Sentinel PRE を適用した場合を**図 4** に示す．まず例外依存を無視して PRE を適用し，PEI 3 と 6 の入れ替わりを解析により検出した結果，生成されるコードを図 4 (a) に示す．移動した PEI 6 の元の位置 4（sentinel）には nop 命令が挿入される．

命令 6 で変数 a がヌルでないならば実行時に起きる

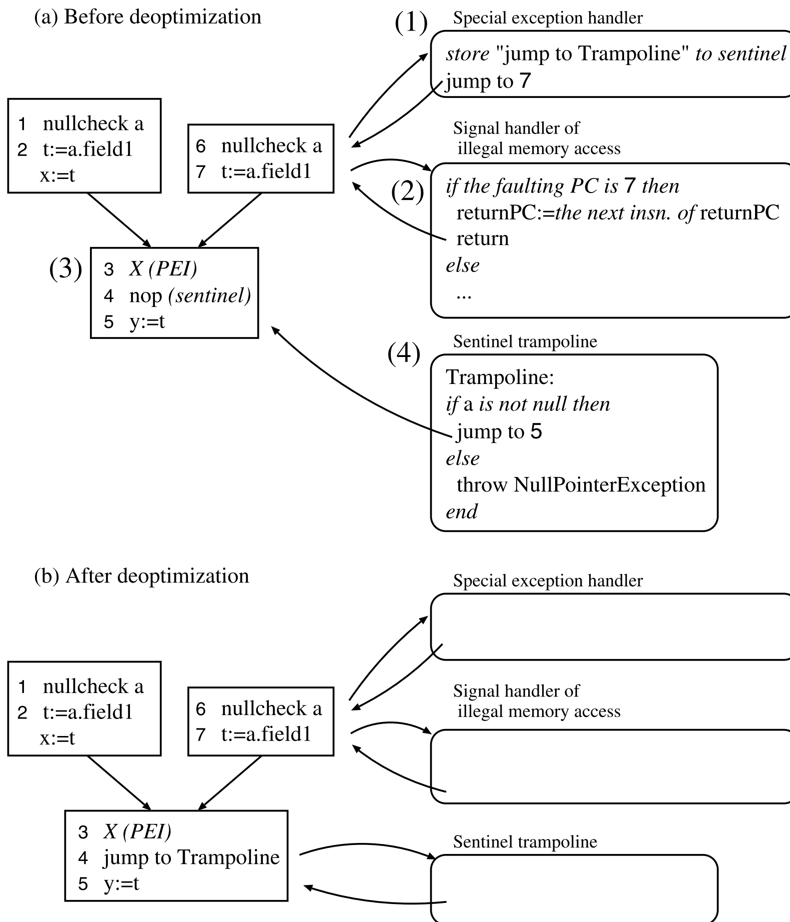

図 4 Sentinel PRE の動作例
Fig. 4 Example behavior of Sentinel PRE.

例外は最適化前と同一であることは明らかである．

命令 6 で変数 a がヌルであった場合を考える．

(1) まず専用の例外ハンドラにジャンプする．ハンドラ内部では移動した PEI の元の位置（sentinel）にジャンプ命令を書き込む．ジャンプ命令の飛び先はその sentinel に対応する “sentinel trampoline” である．その結果の状態を図 4 (b) に示す．ジャンプ命令を書き込んだ後でヌルチェック命令の直後に戻る．

(2) 変数 a の値はヌルのままであるから命令 7 は不正なメモリアドレスからロードしようとする．この不正メモリアクセスは無視することができる．図の例ではシグナルハンドラ内部で返り先を不正メモリアクセスを起こしたロード命令の直後に設定してから返る．

(3) 命令 3 で例外 X が起きるならば実際に X を発生させる．

(4) X が発生しないならば sentinel trampoline へジャンプし，変数 a がヌルであるため実際にヌルチェック例外が発生する．

結果として実行時の正しい例外発生が実現された．

以降のプログラム実行中にこの関数が再び実行されるならば図 4 (b) の状態のコードを実行することになる．移動したヌルチェックに対応する特別な例外ハンドラは実質的に何も状態を変えないコードであるから，図 4 (b) の状態はヌルチェックに関して PRE による最適化を受ける前のコード（図 3 (a)）と意味的に等価である．ロード命令は 7 の位置に移動したままであるが，変数 a がヌルでないならば正しい値をロードし，ヌルならば (2) で述べた仕組みにより無視される．以上より，実行の正しさは保証される．

3. Sentinel PRE のアルゴリズム

前章で述べた脱最適化を用いるためには，上方移動したどの PEI について脱最適化を用いるかを判断しなければならない．なぜならば，上方移動した PEI がす

べて例外順序の入れ替わりを起こすわけではなく，また 3.3.1 項で述べるように，脱最適化を用いると PEI の元の位置（sentinel）に命令スケジューリングとレジスタ割当て上の制約が残るからである．したがって，Sentinel PRE のコンパイル時のアルゴリズムは以下のとおりになる．

（1） 例外依存関係を越える PRE を行う（3.1 節）．
（2） 脱最適化を用いる上方移動 PEI を解析により決定し，同時に脱最適化に必要な情報（対応する sentinel の位置）を集める（3.2 節）．
（3） 脱最適化に対応したコードを生成する（3.3 節）．

以下では我々は Java におけるヌルチェックと範囲チェックを PRE の対象として扱うが，Sentinel PRE は PRE の枠組みで除去できる任意の PEI に適用可能である．

3.1　PRE アルゴリズム

我々は PRE アルゴリズムを PEI の除去に適用する．従来手法[15),16)] では例外依存関係を越えないため，nullcheck a に対してある命令 n が変数 a への代入命令，他の PEI（関数呼び出し含む），もしくはストア命令である場合に nullcheck a が n を越えて上方移動することを許可しない[☆]．boundcheck に関しても同様である．ストア命令で上方移動が阻害されるのは，この PEI に対応する例外ハンドラ中，およびそれ以降の実行中にメモリへ書き込まれた内容を参照する可能性があるからである．しかし，我々の手法では例外依存関係を越える PRE を行うため，n が変数 a への代入命令である場合にのみ n を越える上方移動を許可しない．PRE アルゴリズムの詳細は A.1.2 節に示す．

3.2　解析アルゴリズム

ある実行パスを考えると，例外順序の入れ替わりは

（1） 冗長でないために除去されず移動もしない他の PEI（もしくはストア命令）を越えて上方移動する場合，
（2） 上方移動する他の PEI の移動範囲全体を越えて上方移動する場合，

の 2 つの場合（**図 5** (1)，(2)）に起こる．この 2 つのケースを見つけ出す Sentinel PRE の解析は以下の 2 段階のアルゴリズムである．

（1） 上方移動した PEI の集合を *HoistedExcps* とおく．*HoistedExcps* に対して**図 6** のアルゴリズムを適用する．
（2） ステップ (1) で集めた情報を用いて**図 7** のアルゴリズムを適用する．

☆ 説明を簡単にするために我々は内部に例外ハンドラブロック（Java の場合は try–catch ブロック）を持たない関数を対象とする．したがって，例外が発生した場合は関数の実行は終了し，呼び出し元へ戻る．内部に例外ハンドラブロックがある場合，ブロック境界とレジスタ書き込み命令もまた上方移動を阻害する．

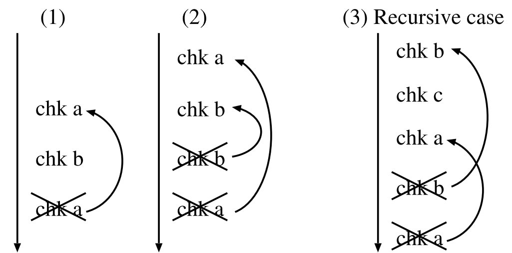

図 5　例外依存関係を越える上方移動
Fig. 5　Hoisting beyond exception dependency.

```
1  SpeculativeExcps := ∅
2  for each h ∈ HoistedExcps do
3    h.Region := ∅
4    h.InnerHoisted := ∅
5    h.Sentinels := ∅
6    W := Succ(h)
7    while W ≠ ∅ do
8      n ∈ W; W := W \ n
9      if n ∈ h.Region then
10       continue
11     else
12       h.Region ∪:= n
13     end
14     if EquivalentInPRE(n, h) then
15       h.Sentinels ∪:= n
16       continue
17     else if n ∈ HoistedExcps then
18       h.InnerHoisted ∪:= n
19     else if n is excp. insn. or memory write then
20       SpeculativeExcps ∪:= h
21     end
22     W ∪:= Succ(n)
23   end
24 end
```

図 6　移動範囲，移動範囲内に移動してくる他の PEI，および sentinel を見つけるアルゴリズム
Fig. 6　Algorithm for identifying hoisting region, other excp. insns. hoisted into the hoisting region and sentinels.

図 6 のアルゴリズムは上方移動した PEI h に対応する sentinel の位置（*h.Sentinels*）を見つけると同時に，移動範囲である *h.Region*，および *h.Region* 内に移動してくる他の PEI（*h.InnerHoisted*）を認識する．アルゴリズムの基本は，ワークリスト W を用いて h から制御フローを forward に sentinel の位置までたどる，というものである．我々が用いる PRE アルゴリズムでは，元々その PEI が存在しなかったパス上にまで PEI を上方移動することはないので，h の位置から制御フローを forward にたどっていけば必ず

```
 1  for each h ∈ HoistedExcps do
 2    if h ∈ SpeculativeExcps then
 3      continue
 4    end
 5    for each e ∈ h.InnerHoisted do
 6      if ∃s ∈ e.Sentinels s.t. s ∈ h.Region then
 7        SpeculativeExcps ∪:= h
 8      end
 9    end
10  end
11  do
12    changed := false
13    for each h ∈ HoistedExcps do
14      if h ∈ SpeculativeExcps then
15        continue
16      end
17      for each i ∈ SpeculativeExcps do
18        if ∃s ∈ i.Sentinels s.t. s ∈ h.Region then
19          SpeculativeExcps ∪:= h
20          changed := true
21        end
22      end
23    end
24  while changed end
```

図 7 例外依存関係を越える上方移動を見つけるアルゴリズム
Fig. 7 Algorithm to find hoisting beyond exception dependency.

sentinel が見つかる．たどった範囲が *Region* であり，1 つの h につき各命令をたかだか 1 回しかたどらないのでこのアルゴリズムは停止する．

14 行目の *EquivalentInPRE* は，命令 n が h の移動前の位置（すなわち sentinel）であるかどうかを直前に行った PRE の情報を用いて判定する関数である．移動前の位置の PEI は PRE によってすでに削除されているため字面で同一か否かを判定基準にはできないが，PRE で冗長性除去を行ったときの情報を残しておいて利用すればよい．Forward にたどる間に上方移動した他の PEI を見つけたら *InnerHoisted* に記録しておく（18 行目）．それ以外の PEI は冗長でないために除去されず移動もしなかった PEI であるから図 5 (1) に相当する．そのような命令を *Region* 内に含むならば例外依存関係を越える上方移動であることを意味するので，*SpeculativeExcps* に h を追加する（20 行目）．

図 5 (2) のタイプの上方移動は図 7 のアルゴリズムの 1–10 行目で判別する．図 5 (2) の例で説明すると，chk a の *InnerHoisted* には chk b が含まれているはずであるから，chk b の sentinel のうち少なくとも 1 つが chk a の *Region* 内にあるならば chk a は例外依存関係を越える上方移動であると見なす．解析アルゴリズムの正当性は A.1.4 節で示す．

以上で例外依存関係を越える上方移動をすべて見つけたことになる．しかし我々は脱最適化を用いるため，例外依存関係を越える上方移動の sentinel もまた上方移動を阻害すると見なさなければならない．図 5 (3) の例では，chk b は chk c に対して例外順序の入れ替わりを起こすが，chk a は chk b と chk c に対して順序の入れ替わりを起こさない．しかし，chk b が脱最適化されると sentinel の位置に戻るため，chk b の sentinel を越える chk a は例外依存関係を越えると見なされる．すなわち，図 5 (1)，(2) のタイプの PEI を基点として，それらの sentinel を上方移動で越える PEI もまた再帰的に *SpeculativeExcps* に追加される．図 7 の 11–24 行目のループがそれにあたる．

SpeculativeExcps の要素は 11–24 行目のループで単調増加し，最大でも *HoistedExcps* と等しくなるまでなので，図 7 のアルゴリズムは停止する．

3.3 脱最適化

前節で述べたアルゴリズムで例外依存を越えると判断された上方移動命令（*SpeculativeExcps*）に関して，我々は上方移動による冗長性除去の効果を維持しつつ，プログラムの実行の正しさを保証しなければならない．我々の基本的な方針は，

- 例外（特にヌルチェック例外や範囲チェック例外）は稀にしか起きない☆ため，その際に実行の正しさを保証する仕組みは遅いものであってかまわない，
- また，一度例外が起きたならばその後に実行されるコードも遅いものであってかまわない，
- そのかわり，一度も例外が起きないときに実行される通常のパス上には可能な限りオーバヘッドが存在しないようにする，

というものである．

そのために我々が用いる手法が 2 章で動作例を示した脱最適化である．そのアルゴリズムは以下のとおりである（**図 8**）．

(1) 例外依存を越えて上方移動したコードを生成する．
(2) 上方移動した PEI で例外が起きない限りはそのコードが実行される．
(3) 上方移動した PEI で例外が起きたならば，特別な例外ハンドラにジャンプして PEI を上方移動する前の位置（sentinel）に戻す．例外ハンドラからは上方移動した PEI の直後に戻る．
(4) 以降は上方移動がキャンセルされたコードが実行される．

☆ プログラム中で明示的に例外を発生させる命令（Java の throw 文など）は部分冗長性除去の対象にはならない．したがって明示的な例外は頻繁に発生しても本研究の有効性は損なわれない．

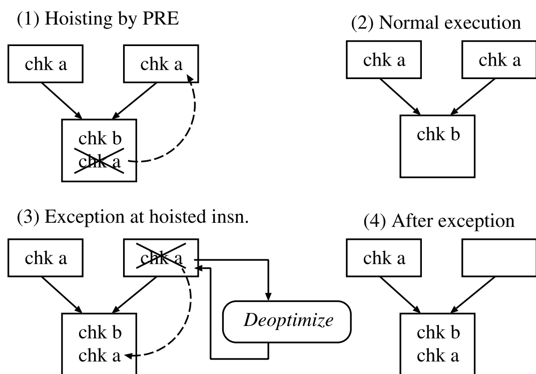

図 8　脱最適化のアルゴリズム
Fig. 8　Algorithm of deoptimization.

3.3.1　実行時コード書き換え

PEI を上方移動する前の状態に戻す手法として

(1)　複数バージョンのコード生成
(2)　実行時再コンパイル
(3)　実行時コード書き換え

の 3 つの選択肢が考えられる．(1) は PEI を上方移動したバージョンと上方移動しないバージョンの 2 種類のコードを生成する手法であるが，コンパイル時間が増加するため実行時コンパイラに実装する場合には向かない．(2) は関数実行中にコードを切り替える on-stack replacement が必要となるため実装が複雑化する欠点がある．

我々は 2 章で示したとおり，実装が最も容易である (3) の実行時コード書き換えを用いる．具体的には関数のコード生成時に，各 sentinel に対応して “sentinel trampoline” と呼ばれるコードを生成する．脱最適化時には sentinel の位置に sentinel trampoline へのジャンプ命令を書き込む．図 4 から該当部分を抜粋して**図 9** に示す．現在の実装では簡単のために sentinel の位置にはジャンプ命令を書き込むための領域として nop 命令を入れておく．nop 命令を用いない場合，上書きされる命令をあらかじめ sentinel trampoline の先頭部分に複製しておけばよい．

実行時コード書き換えを用いる場合，通常の実行パスに以下の制約が残る．

- 命令スケジューリングにおいて，sentinel は元々この場所にあった PEI と同等のスケジューリング制約を持つ．たとえば他の PEI は sentinel を越えて前後に移動できない．なぜならば脱最適化により sentinel で例外が起きるようになる可能性があるからである．
- レジスタ割当てにおいて，sentinel は元々この場所にあった PEI と同一のレジスタを使用すると見なす必要がある．図 9 の場合，変数 a の値は命令 4 の位置でも生きている必要がある．なぜならば sentinel trampoline 中で a の値に対してヌルチェックを行うからである．

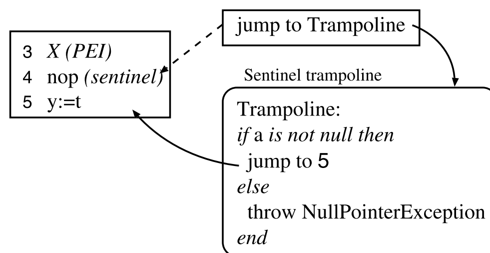

図 9　実行時コード書き換え
Fig. 9　Runtime code patching.

これらの制約は Sentinel PRE を用いるために新たに生じる制約ではなく，Sentinel PRE を用いない場合に sentinel の位置にある PEI が元から持っていた制約である．

3.3.2　特別な例外ハンドラ

実際に実行時コード書き換えを行うのは，例外依存を越えて上方移動した PEI に対応する特別な例外ハンドラである．ハンドラの中では対応する sentinel の位置にジャンプ命令を書き込む．我々は関数のコード生成時にこの特別な例外ハンドラも関数の末尾に追加して生成する．ヌルチェックと範囲チェックは実際には比較命令と条件分岐命令として生成されるため，条件分岐の飛び先を特別な例外ハンドラにする．

マルチスレッド環境では sentinel の位置にジャンプ命令を書き込む操作と，ちょうど sentinel の位置を実行している別のスレッドが競合する可能性がある．ほとんどのアーキテクチャではジャンプ命令 1 つをアトミックに書き込めるため問題はないが，命令長が可変な IA-32 アーキテクチャでは文献 6) で述べられている工夫を用いる必要がある．

また，一度脱最適化を行ったならば特別な例外ハンドラは二度と実行する必要はなく，上方移動した PEI は nop 命令で置き換えればよい．しかしハンドラは 2 回以上実行しても副作用はないため，我々は実装を簡単にするためにそのままにしておく．

3.4　データ依存とロード命令

本章ではこれまで PEI どうしの依存関係のみを扱ってきた．しかし，プログラム中には PEI によって保護されているロード命令の冗長性，およびそのロード命令にデータ依存している PEI の冗長性が存在する．**図 10** に配列要素のロードの部分冗長性を除去する例を示す．元のプログラム図 10 (a) に PRE を 1 回適用

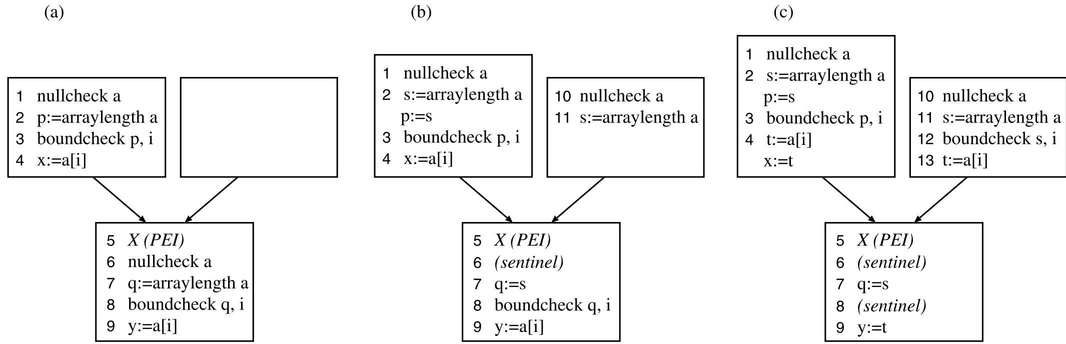

図 10 配列要素のロードの最適化
Fig. 10 Optimization of load from array element.

しただけでは図 10 (b) のようにヌルチェックとそれに保護されているロード命令（arraylength）の部分冗長性が除去されるだけである☆．したがって図 10 (c) のように冗長性を完全に除去するために PRE を繰り返し適用する[15]．

3.4.1 例外依存を越えるロード命令

PEI によって保護されているロード命令に関して問題となるのが例外依存を越えて上方移動した場合である．図 10 (c) の命令 10 でヌルチェック例外が起きると，特別な例外ハンドラで脱最適化を行ってから命令 11 に返ってくる．変数 a はヌルのままであるから，システムによっては不正メモリアクセスとなる．命令 12 で範囲チェック例外が起きた場合も変数 i の値によっては命令 13 が不正メモリアクセスとなる．

ここで我々はこの不正メモリアクセスは無視することができる．たとえば命令 11 と 13 で起きる不正メモリアクセスを無視するとする．命令 10 でヌルチェック例外が起きたと仮定すると変数 s と t は不定な値となる☆☆が，結局は命令 5 か 6 のどちらか一方で必ず例外が発生する．したがって変数 s と t の不定な値が関数外部から参照可能な状態，すなわちストア命令の引数や関数呼び出しの引数，関数の返り値などになることはない．一般化していうと，例外依存を越えて上方移動した PEI h によって保護されているロード命令の不定な結果を参照する可能性がある命令は

(1) 例外依存（h の sentinel を含む）を越えて上方移動した命令

(2) h の sentinel より下にある命令

の 2 種である．ストア命令や関数呼び出しは PRE の対象ではないので (1) のように上方移動することはない．また，遅くとも sentinel の位置までに例外が発生するので (2) の sentinel より下の命令に実行が到達することはない．したがって不正メモリアクセスは無視することができる．

不正メモリアクセスを無視するためには，IA-64 アーキテクチャなどでサポートされている non-faulting ロード命令を用いるか，もしくは対応するシグナルハンドラ（UNIX の場合は SIGSEGV）の中で，無視すべきメモリアクセスの場合はロード命令の直後にすぐに戻るようにすればよい．不正メモリアクセスを無視すべきロード命令を見つけるために，我々は図 6 のアルゴリズムで移動範囲を探索する最中に，上方移動した PEI h が保護するロード命令も同時に探索する．

3.4.2 ヌルチェックとロード命令

ヌルポインタが指す近辺をアクセスするとシグナルが発行されるシステムでは，ヌルチェックをそれが保護する後続のロード命令で代用することができる[15]．ヌルチェックを兼ねるロード命令が例外依存を越えて上方移動したならば，シグナルハンドラ内で sentinel の位置へジャンプ命令の書き込みを行った後でロード命令の直後へ戻ればよい．

4. 実　　験

我々は Sentinel PRE を，我々が開発中の Java 用の実行時コンパイラ RJJ 上に実装した．RJJ は Java バーチャルマシンのフリーな実装である Kaffe-1.0.7[14] から呼び出される形で動作し，部分冗長性除去のアルゴリズムとして我々が提案した Partial Value Number Redundancy Elimination（PVNRE）[22),25] を用いている．ヌルチェックは後続のロード命令で代用せ

☆ PEI とそれに保護されているロード命令は 1 回の PRE で同時に冗長性除去することができる．ただし，ロード命令はメモリエイリアスの影響を受けるので移動範囲は PEI より狭い可能性がある．

☆☆ 命令 12 で範囲チェック例外が起きるかどうかは変数 s と i の値によるが，いずれにせよ結果は同じである．

ずに比較命令と条件分岐命令を用いる．

以後の計測はすべて，8 個の 900 MHz UltraSPARC III プロセッサと 40 GB のメモリを搭載した Sun V880/Solaris 9 上で行った．Kaffe はユーザレベルスレッドライブラリしか持たないため，実行には CPU 1 個のみ用いられる．計測中にゴミ集めを起こさないために，Java バーチャルマシンの起動オプションとして “-ms700m -mx700m” を指定した．実行時間のデータは 5 回実行した中で最も短い実行時間を採用した．

4.1　マイクロベンチマーク

我々は脱最適化が実行速度に与える影響を計測するために，**図 11** に示すマイクロベンチマークを作成した．マイクロベンチマークを用いる理由は，次節で用いる現実的なプログラムのベンチマークではヌルチェック例外と範囲チェック例外が一度も起こらないため脱最適化の影響を調べることができないからである．

マイクロベンチマークの 8 行目のフィールドアクセスにともなうヌルチェックとロード命令はループ不変である．しかし，7 行目にストア命令があるために Sentinel PRE を用いないとループの外へ出せない．我々はまず例外依存を越えてループ不変命令を最適化したコードの実行時間を 18 行目のメソッド呼び出しで計測し，19 行目のメソッド呼び出しで脱最適化を起こし，21 行目のメソッド呼び出しで 2 回目の計測を行う．

結果を**表 1** の上段に示す．脱最適化される前と後では 2 倍以上の差が生じている．脱最適化された後でもロード命令はループの外に出たままであるが，sentinel trampoline へのジャンプ，ヌルチェック，および sentinel の直後へのジャンプがループ内に追加されたことが速度低下の原因である．参考のために Sentinel PRE を用いない場合の結果を下段に示す．Sentinel PRE を用いないとヌルチェックとロード命令がループの中に残ったままであるが，脱最適化された後のコードよりも高速である．なお，Sentinel PRE を用いない場合の 1 回目と 2 回目の差は，1 回目にはメソッド meth が実行時コンパイルされる時間が含まれるためである．実行時コンパイルの時間は Sentinel PRE を用いる場合の 1 回目の実行時間にも含まれている．

```
1  class T {
2  int f1, f2;
3
4  static int meth(T obj1, T obj2, int iter) {
5    int i, ret;
6    for (i = 0, ret = 0; i < iter; i++) {
7      obj2.f2 = iter;
8      ret += obj1.f1;
9    }
10   return ret;
11 }
12
13 public static void main(String args[]) {
14   T obj1, obj2;
15   obj1 = new T(); obj1.f1 =1; obj1.f2 = 0;
16   obj2 = new T(); obj2.f1 =0; obj2.f2 = 0;
17
18   meth(obj1, obj2, 2000000000); // 1 回目
19   try { meth(null, obj2, 1); }
20   catch (NullPointerException ex) {}
21   meth(obj1, obj2, 2000000000); // 2 回目
22 }
23 }
```

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

表 1　マイクロベンチマークの結果
Table 1　Result of the micro benchmark.

	1 回目（秒）	2 回目（秒）
Sentinel PRE 使用	11.138	24.504
Sentinel PRE 不使用	16.707	16.706

4.2　マクロベンチマーク

我々は現実的なマクロベンチマークとして SPEC JVM98 [23] 中の compress と Java Grande Forum Benchmark Suite [13] 中の heapsort，および sor プログラムを用いる．Sentinel PRE を用いる場合，用いない場合，および参考として Kaffe-1.0.7 付属の実行時コンパイラ jit と Sun Hotspot Server VM 1.4.2_04-b05 を用いた場合の実行時間結果を**表 2** に示す．

Sentinel PRE を用いると，用いない場合に比べて heapsort において 8.4%の性能向上を示す．なお，Hotspot Server VM に比べて我々が開発中の RJJ の方が高度な冗長性除去を行っているにもかかわらず compress と sor で性能が劣るのは，単純なレジスタ割当てアルゴリズムを用いているためにコピー命令が多く残ることと，命令スケジューリングを行っていないことが原因である．

heapsort プログラムで最も頻繁に実行されるメソッドである “NumSift” の最内ループ中のコードを抜粋して単純化したものを**図 12** に示す．冗長性除去の前のコードが図 12 (a) である．PVNRE によりヌルチェックと arraylength 命令はループの外に出るが，Sentinel PRE を用いるとさらに範囲チェックとロード命令が上方移動可能である（図 12 (b)）．Sentinel PRE を用いない場合，別の範囲チェックとの例外依存に阻害されて上方移動できない（図 12 (c)）．この範囲チェックとロード命令の部分冗長性を除去できるか否かが，実行時間の差に大きく寄与している．

表 2 マクロベンチマークの実行時間
Table 2 Execution times of the macro benchmarks.

	compress	heapsort	sor
Kaffe-1.0.7/RJJ/Sentinel PRE 使用	17.882 秒	6.843 秒	16.232 秒
Kaffe-1.0.7/RJJ/Sentinel PRE 不使用	18.644 秒	7.419 秒	16.572 秒
性能向上比	4.3%	8.4%	2.1%
Kaffe-1.0.7/jit	47.722 秒	30.426 秒	56.670 秒
Sun Hostspot Server VM 1.4.2	15.227 秒	6.964 秒	11.517 秒

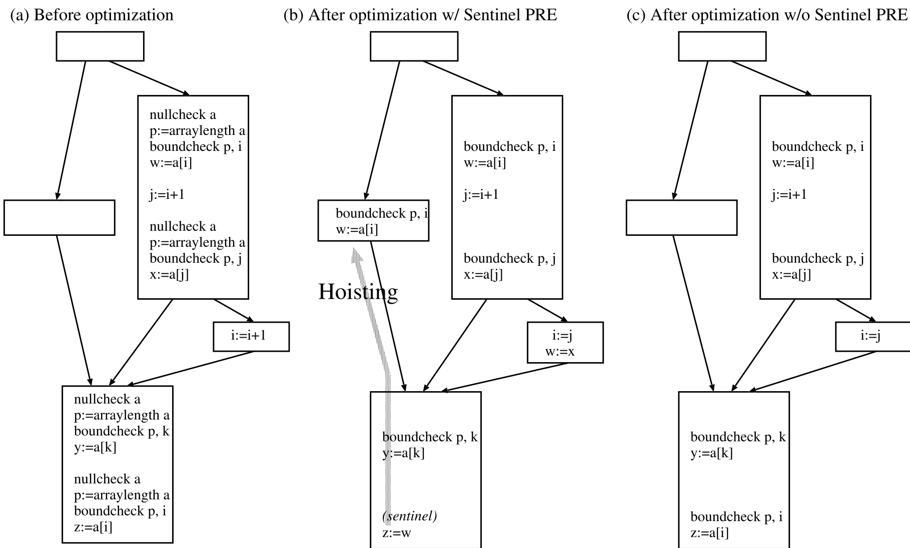

図 12 heapsort プログラム中の NumSift メソッド
Fig. 12 “NumSift” method in “heapsort” program.

表 3 Sentinel PRE の解析アルゴリズムが解析時間全体に占める割合
Table 3 Ratio of the analysis time of Sentinel PRE to that of RJJ.

	compress	heapsort	sor
割合（%）	0.6	0.14	0.15

例外依存関係を越える上方移動を見つける図 6 と図 7 のアルゴリズムが実行時コンパイラの解析時間全体に占める割合を**表 3** に示す．表 3 の結果が示すとおり，Sentinel PRE の解析によりコンパイル時間が大きく増えることはない．

5. 関連研究

5.1 冗長性除去

Sentinel PRE は投機的な最適化の一種ということができる．しかし，既存の PRE の研究で「投機的な PRE」と呼ばれているもの[3),4),9),24)] は**図 13** (a)，(b) に示すように，最適化前にはその計算が存在していなかったパス（右上から右下へのパス）上に命令を投機的に移動することで部分冗長性を除去する手法であ

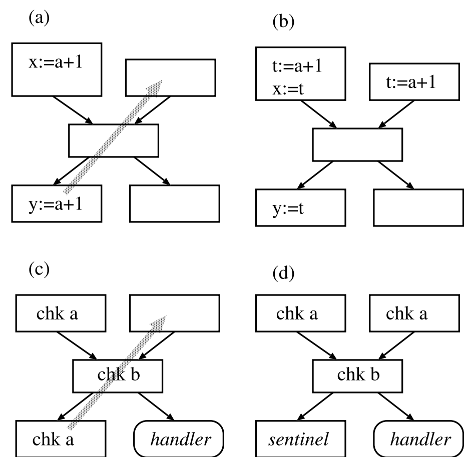

図 13 投機的な PRE
Fig. 13 Speculative PRE.

る．投機的な PRE は必ずしも全体の実行命令数を削減するとは限らないので，実行頻度プロファイルを用いて投機的移動の cost-benefit を見積もることが多い．我々が用いる PRE の枠組みではこのような投機的な

移動は禁じている．

しかし，図 13 (c)，(d) のように PEI を条件分岐命令と見なすと，我々の Sentinel PRE も投機的な PRE の一種と考えられる．ただし，既存の投機的な PRE と Sentinel PRE は以下の点で異なる．

- 既存の投機的な PRE で PEI を移動する場合は，特別なハードウェアのサポートを前提としている．ハードウェアのサポートがない場合，絶対に例外を起こさないことが分かっている命令のみを投機的な移動の対象とする．一方，Sentinel PRE は我々が開発した脱最適化を用いることでハードウェアサポートを前提とせずに PEI を移動することができる．
- 現実のプログラムでは例外が起きる頻度が低いという仮定をおける．したがって Sentinel PRE ではプロファイル情報をとって投機的上方移動の cost-benefit を見積もる必要はない．

PRE の枠組みを用いて PEI の冗長性を除去する研究としては，ヌルチェックを除去する文献 15) や範囲チェックを除去する文献 2)，19) がある．しかしいずれの手法も例外依存を越えない範囲で最適化するか，もしくは例外順序の入れ替わりをまったく考慮せずに最適化する手法である．一方，Sentinel PRE を用いると，例外依存を越えて最適化しつつ例外順序の入れ替わりによるプログラムの意味の変化を防ぐことができる．

範囲チェックの冗長性を除去する手法は PRE を用いるもの以外にも多数提案されている．ループバージョニングを用いる手法[6) は，ループ中の配列アクセスへの添字の範囲がループ実行前に分かる場合に適用される．この手法では添字の範囲が配列の上限下限の範囲内にあるか否かのチェックをループの直前で投機的に行う．チェックが成功したら範囲チェックが除去されたバージョンのループを実行し，チェックが失敗したら範囲チェックを残したままのバージョンのループを実行する．ループバージョニングはループ中の範囲チェックを投機的にループの外に出すことに特化した手法であり，コードを複製するコストがかかる．一方，Sentinel PRE は任意の制御フローグラフに適用可能であり，コード複製は必要ない．両者はループ不変な範囲チェックをループの外に出す場合にのみ適用範囲が重なるが，それ以外は相補的な関係にあるため，バージョニングを行った後のループ，およびバージョニングが適用できなかったループに対して Sentinel PRE を用いればよい．

Gupta の手法[8) とその改良版[16) は *ANTIC* 情報を用いて複数の範囲チェックを例外依存を越えない範囲で上方へまとめ，その後 *AVAIL* 情報を用いて下方の冗長な範囲チェックを除去する．字面で同一の範囲チェックだけでなく条件式が包含関係にある複数の範囲チェックを 1 つのチェックにまとめることができるが，PRE と異なり部分冗長性は除去できない．この手法と Sentinel PRE を組み合わせる場合，Sentinel PRE を適用した後で Guputa の手法を適用する．このとき，例外依存を越えて上方移動した範囲チェックがさらに上方へ移動して他の範囲チェックとまとめられる可能性があるため，対応する sentinel 情報も更新する必要がある．

5.2　命令スケジューリング

コンパイラの最適化フェーズの中で冗長性除去と同様に命令順序の入れ替わりが起こるのは命令スケジューリングである．特に，superblock スケジューリング[10) における general percolation や sentinel スケジューリング[5),20) は例外を起こす可能性のある命令（ロード命令など）を条件分岐を越えて上方へ移動するという点で，Sentinel PRE と目的は異なるが手法は類似している．しかし，general percolation では例外が起きた場合のプログラムの正しさは保証されず，sentinel スケジューリングでは特別なハードウェア命令が必要とされる．

Arnold ら[1) は Java に general percolation を適用したが，ロード命令はそれを保護するヌルチェックを越えて上方移動しないので例外依存を越える手法ではない．Ishizaki ら[11) は Java において sentinel スケジューリングを応用し，例外依存を越えてロード命令を上方移動する手法を提案したが，IA-64 の特別なハードウェア命令のサポートを前提としている．Gupta ら[7) は Java においてハードウェアサポートなしに例外依存を越えるスケジューリングを提案した．しかし彼らの手法は Sentinel PRE のように基本ブロックを越える移動ではなく，基本ブロック内で PEI の順序を入れ替えることを目的としている．また彼らの手法はコード複製を行うためコンパイル時間が増加する．

上にあげた命令スケジューリングに関する研究ではプログラム依存グラフを作成するため，例外順序の入れ替わりを検出するのは容易である．一方，PRE では依存グラフは作成されないため，Sentinel PRE で順序の入れ替わりを検出するには 3.2 節で示した解析アルゴリズムを用いる必要がある．

5.3　実行時コード書き換えと脱最適化

実行時コード書き換えによる脱最適化を用いた研究としては，仮想関数の静的呼び出しへの変換とインラ

イン展開に利用した文献 6)，12) があげられる．これらの研究では，クラスの動的ロードにより最適化の前提が成り立たなくなった場合に関数の呼び出し元を書き換えて仮想関数呼び出しへ戻す，という作業を行う．

これまでのところ，実行時コード書き換えによる脱最適化を PRE に適用した研究は Sentinel PRE 以外にはない．

6. まとめ

本研究で我々はプログラムの意味を保存しつつ例外依存関係を越える部分冗長性除去，Sentinel PRE を提案した．Sentinel PRE は例外依存関係を無視して上方移動を行い，その後で高速な解析により例外順序の入れ替わりを検出する．順序が入れ替わった PEI で例外が起きた場合，プログラムの意味を保つために上方移動する前の状態に脱最適化でコードを戻す．現実のプログラムで例外が起きることは稀であるため，ほとんどの場合は上方移動により最適化された高速なコードが実行される．

我々は Sentinel PRE を Java の実行時コンパイラに実装して実験を行い，Java Grande Benchmark 中の heapsort プログラムで 8.4%の性能向上を得た．このとき，Sentinel PRE の解析にかかる時間は実行時コンパイラの解析時間のうち 1%未満にすぎず，解析時間は問題とならないことを確認した．今後は，より広範囲なベンチマークプログラムで Sentinel PRE の有効性を調べることが我々の研究課題である．

参考文献

1) Arnold, M., Hsiao, M., Kremer, U. and Ryder, B.: Instruction Scheduling in the Presence of Java's Runtime Exceptions, *Proc. International Workshop on Languages and Compilers for Parallel Computing* (*LCPC '99*), pp.18–34 (1999).
2) Bodik, R., Gupta, R. and Sarkar, V.: ABCD: eliminating array bounds checks on demand, *Proc. ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp.321–333, ACM Press (2000).
3) Bodik, R., Gupta, R. and Soffa, M.L.: Complete Removal of Redundant Computations, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.1–14 (1998).
4) Cai, Q. and Xue, J.: Optimal and efficient speculation-based partial redundancy elimination, *Proc. international symposium on Code generation and optimization*, pp.91–102, IEEE Computer Society (2003).
5) ching Ju, R.D., Nomura, K., Mahadevan, U. and Wu, L.-C.: A Unified Compiler Framework for Control and Data Speculation, *Proc. 2000 International Conference on Parallel Architectures and Compilation Techniques*, p.157, IEEE Computer Society (2000).
6) Cierniak, M., Lueh, G.-Y. and Stichnoth, J.M.: Practicing JUDO: Java under dynamic optimizations, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.13–26 (2000).
7) Gupta, M., Choi, J.-D. and Hind, M.: Optimizing Java Programs in the Presence of Exceptions, *Proc. 14th European Conference on Object-Oriented Programming*, pp.422–446, Springer-Verlag (2000).
8) Gupta, R.: Optimizing array bound checks using flow analysis, *ACM Lett. Program. Lang. Syst.*, Vol.2, No.1-4, pp.135–150 (1993).
9) Horspool, R.N. and Ho, H.C.: Partial Redundancy Elimination Driven by a Cost-Benefit Analysis, *Proc. 8th Israeli Conference on Computer-Based Systems and Software Engineering*, p.111, IEEE Computer Society (1997).
10) Hwu, W.-M.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G. and Lavery, D.M.: The superblock: an effective technique for VLIW and superscalar compilation, *J. Supercomput.*, Vol.7, No.1-2, pp.229–248 (1993).
11) Ishizaki, K., Inagaki, T., Komatsu, H. and Nakatani, T.: Eliminating Exception Constraints of Java Programs for IA-64, *The 11th International Conference on Parallel Architectures and Compilation Techniques* (*PACT-2002*), pp.259–268 (2002).
12) Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H. and Nakatani, T.: A study of devirtualization techniques for a Java Just-In-Time compiler, *Proc.15th ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications*, pp.294–310, ACM Press (2000).
13) Java Grande Benchmarking Project: Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/
14) Kaffe.org: Kaffe Open VM. http://www.kaffe.org/
15) Kawahito, M., Komatsu, H. and Nakatani, T.: Effective null pointer check elimination utilizing hardware trap, *SIGPLAN Not.*, Vol.35, No.11, pp.139–149 (2000).

16) Kawahito, M., Komatsu, H. and Nakatani, T.: Eliminating Exception Checks and Partial Redundancies for Java Just-in-time Compilers (2000). IBM Research Report RT0350.
17) Knoop, J., Rüthing, O. and Steffen, B.: Lazy code motion, *ACM SIGPLAN Notices*, Vol.27, No.7, pp.224–234 (1992).
18) Knoop, J., Rüthing, O. and Steffen, B.: Optimal code motion: theory and practice, *ACM Trans. Prog. Lang. Syst.* (*TOPLAS*), Vol.16, No.4, pp.1117–1155 (1994).
19) Kolte, P. and Wolfe, M.: Elimination of redundant array subscript range checks, *ACM SIGPLAN Notices*, Vol.30, No.6, pp.270–278 (1995).
20) Mahlke, S.A., Chen, W.Y., Bringmann, R.A., Hank, R.E., Hwu, W.-M.W., Rau, B.R. and Schlansker, M.S.: Sentinel scheduling: a model for compiler-controlled speculative execution, *ACM Trans. Comput. Syst.*, Vol.11, No.4, pp.376–408 (1993).
21) Morel, E. and Renvoise, C.: Global optimization by suppression of partial redundancies, *Comm. ACM*, Vol.22, No.2, pp.96–103 (1979).
22) Odaira, R. and Hiraki, K.: Partial Value Number Redundancy Elimination, Technical report, Dept. of CS, Univ. of Tokyo (2004). TR04-01.
23) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/
24) 川人基弘，小松秀昭，中谷登志男：Java 言語に対する投機的なメモリアクセスの最適化手法，情報処理学会論文誌，Vol.44, No.3, pp.883–896 (2003).
25) 大平 怜，平木 敬：値番号に基づく部分冗長性除去，情報処理学会論文誌：プログラミング，Vol.45, No.SIG 9 (PRO 22), pp.59–79 (2004).

付 録

A.1 Sentinel PRE の正当性

A.1.1 変形の正当性

以降で述べる Sentinel PRE の正当性を証明するために，我々はまず正当な変形（最適化）とは何か，を定義する．関数への入力環境 in に対して変形前に発生する例外を $X_{before}(in)$，変形後に発生する例外を $X_{after}(in)$ と表記する．ここで入力環境とは関数への引数，メモリの内容，I/O の結果などをすべて含む．入力環境の集合を $Inputs$ とする．

定義 A.1.1 （例外発生に関する変形の正当性）ある変形は例外発生に関して正当な変形である $\overset{def}{\Leftrightarrow}$

$$\forall in \in Inputs\ .\ X_{before}(in) = X_{after}(in)\ .$$

入力環境 in に対して関数内の実行パス p は一意に定まる．実行パス p を与える入力環境の集合を $I(p)$ と表記する．すべてのパスの集合を $Paths$ とすると，$\oplus_{\forall p \in Paths} I(p) = Inputs$ である．したがって，すべてのパス p へのすべての入力環境 $in \in I(p)$ について $X_{before}(in) = X_{after}(in)$ ならば変形の正当性が成り立つ．

A.1.2 PRE の枠組み

我々は Sentinel PRE の基盤となる PRE の枠組みとして Bodik らが提案した手法[3] を用いる．彼らの手法は PRE の枠組みとして広く用いられている Lazy Code Motion[17),18)] とは異なるデータフロー方程式を用いるが，同一の最適化の結果を与える．我々が Bodik らの手法を用いるのは Sentinel PRE の正当性の証明が直感的で容易となるためであり，実装上は LCM を用いることもできる．

図 14 にデータフロー方程式を示す．n，m は命令，x は式を表す．我々は PRE アルゴリズムを PEI の除去に適用するが，従来の手法では例外依存関係を越えないため $COMP$，$TRANSP_{down}$，$TRANSP_{up}$ は以下のようになる：

$COMP(n, \text{nullcheck a}) \Leftrightarrow$
n は nullcheck a である．
$TRANSP_{down}(n, \text{nullcheck a}) \Leftrightarrow$
n は変数 a への代入命令でない
$TRANSP_{up}(n, \text{nullcheck a}) \Leftrightarrow$
n は変数 a への代入命令でなく，
かつ他の PEI（関数呼び出し含む），
ストア命令でない．

boundcheck に関しても同様である．しかし，我々の手法では例外依存関係を越える PRE を行うため，$TRANSP_{up}$ は $TRANSP_{down}$ と同一とする．

出力は $AVAIL^{all}$，$ANTIC^{all}$，$AVAIL^{Msome}$ である．$AVAIL^{all}(n, x)$ は関数の入口から n に至るすべてのパス上で有効な x が存在するならば真，$ANTIC^{all}(n, x)$ は逆に n から関数の出口に至るすべてのパス上で有効な x が存在するならば真である．$AVAIL^{Msome}(n, x)$ は n に至る少なくとも 1 つのパス上で有効な x が存在するならば真であるが，元々 x が存在しないパス上へ投機的に命令を移動しないため，$PREVENTED$ 条件を追加する．詳細は文献 3) で述べられている．

計算結果を用いて $\mathsf{M}ust$，$\mathsf{M}ay$，$\mathsf{N}o$ を以下のように定義する．

$$AVAIL = \mathsf{M}ust \overset{def}{\Leftrightarrow} AVAIL^{all} \wedge AVAIL^{Msome}$$
$$AVAIL = \mathsf{M}ay \overset{def}{\Leftrightarrow} \neg AVAIL^{all} \wedge AVAIL^{Msome}$$

$$AVAIL_{in}^{all}(n,x) = \bigwedge_{\forall m \in Pred(n)} AVAIL_{out}^{all}(m,x)$$
$$AVAIL_{out}^{all}(n,x) = (AVAIL_{in}^{all}(n,x) \wedge TRANSP_{down}(n,x)) \vee COMP(n,x)$$

$$ANTIC_{out}^{all}(m,x) = \bigwedge_{\forall n \in Succ(m)} ANTIC_{in}^{all}(n,x)$$
$$ANTIC_{in}^{all}(n,x) = (ANTIC_{out}^{all}(n,x) \wedge TRANSP_{up}(n,x)) \vee COMP(n,x)$$

$$AVAIL_{in}^{Msome}(n,x) = \bigvee_{\forall m \in Pred(n)} AVAIL_{out}^{Msome}(m,x)$$
$$PREVENTED(n,x) = \neg AVAIL_{in}^{all}(n,x) \wedge \neg ANTIC_{in}^{all}(n,x)$$
$$AVAIL_{out}^{Msome}(n,x) = (AVAIL_{in}^{Msome}(n,x) \wedge TRANSP_{down}(n,x) \wedge \neg PREVENTED(n,x)) \vee COMP(n,x)$$

図 14 PRE のデータフロー方程式
Fig. 14 Equation system for PRE.

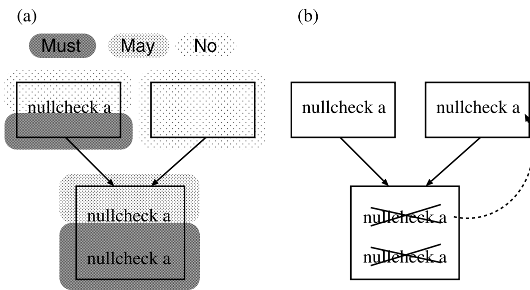

図 15 M*ust*, M*ay*, N*o* 領域
Fig. 15 M*ust*, M*ay*, N*o* region.

$$AVAIL = \mathsf{N}o \overset{\mathrm{def}}{\Leftrightarrow} \neg AVAIL^{all} \wedge \neg AVAIL^{Msome}$$

最終的に以下のような変形を行う．

（1） $AVAIL_{out}(m,x) = \mathsf{N}o \wedge AVAIL_{in}(n,x) = \mathsf{M}ay \wedge ANTIC_{in}^{all}(n,x) \wedge m = Succ(n)$ であるならばエッジ $m \rightarrow n$ に x を新たに挿入する．

（2） $(AVAIL_{in}(n,x) = \mathsf{M}ust \vee AVAIL_{in}(n,x) = \mathsf{M}ay) \wedge COMP(n,x)$ であるならば n における x の計算を削除する．

（3） $AVAIL_{in}(n,x) = \mathsf{N}o \wedge COMP(n,x)$ であるならば命令 n はそのまま．

以上が Bodik らによる PRE の枠組みである．

図 15 (a) に M*ust*，M*ay*，N*o* が成り立つ領域を例示し，図 15 (b) に変形の結果を示す．PRE による変形はいい換えると，M*ay* 領域の下端にある命令を N*o* と M*ay* の境界まで上方移動して M*ay* 領域全体を M*ust* 領域，すなわち完全冗長な領域へ変換する作業である．

A.1.3 Effective PEI

ある実行パスを考えると，その上の PEI は**図 16** の上段に示すとおり M*ust* 命令，M*ust*–M*ay* 命令，N*o*–M*ay* 命令，N*o* 命令の 4 つに分類される．PRE による変形の結果を図 16 の下段に示す．

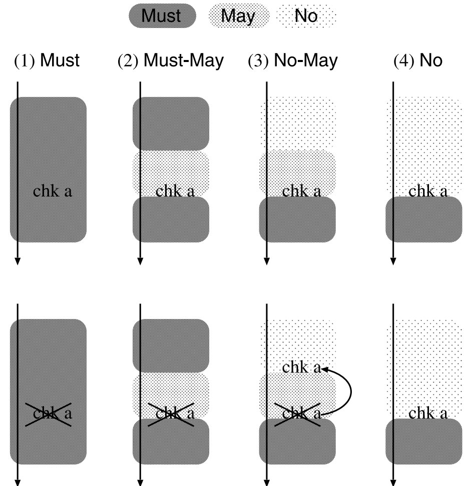

図 16 M*ust* 命令，M*ust*–M*ay* 命令，N*o*–M*ay* 命令，N*o* 命令
Fig. 16 M*ust* insn, M*ust*–M*ay* insn, N*o*–M*ay* insn, and N*o* insn.

（1） M*ust* 命令は削除．

（2） M*ust*–M*ay* 命令は削除．

（3） N*o*–M*ay* 命令は N*o* との境界まで上方移動．

（4） N*o* 命令はそのまま．

補題 A.1.2 任意の実行パス p 上の任意の M*ust* 命令もしくは M*ust*–M*ay* 命令を x とおく．すべての入力環境に対して PRE による変形前にも変形後にも x の上方に同等の PEI が存在する．

補題 A.1.3 任意の実行パス p 上の任意の N*o*–M*ay* 命令もしくは N*o* 命令を x とおく．ある入力環境に対して PRE による変形前に x に実行が到達し，PRE による変形後にも x に実行が到達すると仮定する．変

形前に x で例外が発生しないならば変形後にも x で例外は発生しない．また，変形前に x で例外が発生するならば変形後にも x で例外が発生する．

補題 A.1.2 と A.1.3 はともに PRE アルゴリズムそのものの正しさから証明される．補題 A.1.3 について，PRE により例外順序の入れ替わりが起きるとそもそも x に実行が到達しない可能性があるので，PRE による変形の正当性を直接示唆するものではない．

我々は No–May 命令と No 命令を合わせて effective PEI（ePEI）と呼ぶ．また，ある実行パスから ePEI のみを抽出して並べたものを ePEI 列と呼ぶ．

系 A.1.4 PRE による変形の前後ですべてのパスにおいて ePEI 列が変化しないならば，変形の正当性を満たす．

証明 補題 A.1.2 より Must 命令と Must–May 命令は削除しても例外発生に影響はないため，我々は ePEI についてだけ考えればよい．ePEI 列が変化しないのだから補題 A.1.3 より，変形の正当性が満たされる．□

A.1.4 解析アルゴリズムの正当性

図 6 のアルゴリズムが図 5 (1) のタイプの例外順序の入れ替わりを検出することは自明である．図 7 の 1–10 行目のアルゴリズムが図 5 (2) のタイプの入れ替わりを正しく検出することを以下に示す．

補題 A.1.5 PEI h，e を $h \in \mathit{HoistedExcps}$，$e \in h.\mathit{InnerHoisted}$ とおく．$\forall s \in e.\mathit{Sentinels}$ について $s \notin h.\mathit{Region}$ ならば，すべての実行パスにおいて h と s の順序の入れ替わりはない．

証明 $e \in h.\mathit{InnerHoisted}$ と $s \notin h.\mathit{Region}$ より，e から実行パスを forward にたどっていくと s に到達するよりも先に h の sentinel に到達する．すなわち s は h の移動範囲外から移動範囲内へ上方移動してきたことを意味するので，h と s の順序の入れ替わりはない．□

系 A.1.6 1–10 行目のアルゴリズムは図 5 (2) のタイプの入れ替わりを検出する．

証明 補題 A.1.5 の対偶．□

A.1.5 Sentinel PRE の正当性

最適化前の関数を P_{unopt} とおく．例外依存関係を越える PRE による変形を受けた関数，すなわち $\mathit{HoistedExcps}$ がすべて上方移動した関数を P_{opt}，$\mathit{HoistedExcps}$ のうち $\mathit{SpeculativeExcps}$ が上方移動しない関数を P_{deopt} とおく．

補題 A.1.7 任意の入力環境について P_{unopt} と P_{deopt} は同じ例外を発生させる．

証明 系 A.1.4 より，すべてのパスについて P_{unopt} と P_{deopt} は ePEI 列が同一であることを示せばよい．背理法により証明する．少なくとも 1 つのパスにおいて ePEI 列が異なると仮定する．ePEI は PRE によりパス上から除去，もしくはパス上に追加されることはないので，ePEI 列が変化するとすれば順序の入れ替わりが起きている．No 命令は移動しないので No 命令どうしの順序の入れ替わりはない．したがって順序の入れ替わりは以下の 3 つの場合のみである．

(1) No–May 命令と No 命令の入れ替わり
(2) No–May 命令どうしの入れ替わり
  (a) No–May 命令が，上方移動した別の No–May 命令の移動範囲全体を越えて上方移動
  (b) No–May 命令が上方移動しない別の No–May 命令を越えて上方移動

(1) と (2) (a) の場合，このタイプの No–May 命令は図 6 のアルゴリズムと図 7 の 1–10 行目のアルゴリズムで $\mathit{SpeculativeExcps}$ に入る（系 A.1.6 より）ので上方移動しないはず．よって矛盾．(2) (b) の場合，上方移動しない別の No–May 命令を x とおくと，x は No–May 命令にもかかわらず上方移動しないのだから P_{deopt} の定義より $x \in \mathit{SpeculativeExcps}$ である．しかし，図 7 の 11–24 行目のアルゴリズムにより x を移動範囲内に含む No–May 命令もまた $\mathit{SpeculativeExcps}$ に入るので，上方移動しないはず．よって矛盾．□

補題 A.1.8 P_{deopt} に加えて任意の k 個（$0 \le k \le |\mathit{SpeculativeExcps}|$）の PEI $x \in \mathit{SpeculativeExcps}$ が上方移動した関数 $P_{deopt}(k)$ は，脱最適化を用いると任意の入力環境について P_{deopt} と同じ例外を発生させる．

証明 k に関する数学的帰納法で証明する．$k = 0$ のとき，$P_{deopt}(0) = P_{deopt}$ より明らか．$k = n - 1$（$1 \le n \le |\mathit{SpeculativeExcps}|$）のとき命題は成り立つと仮定する．$k = n$ のとき，$P_{deopt}(n)$ への任意の入力環境について

(1) 上方移動したある $x \in \mathit{SpeculativeExcps}$ で例外が起きるならば，その直後に脱最適化により関数は $P_{deopt}(n-1)$ となるので帰納法の仮定と補題 A.1.3 より，命題は成り立つ．
(2) 上方移動したいかなる $x \in \mathit{SpeculativeExcps}$ でも例外が起きないならば，やはり補題 A.1.3 より，命題は成り立つ．

よって命題は成立する．□

定理 A.1.9 （Sentinel PRE の正当性） Sentinel PRE は例外発生に関して正当な変形である．

証明 $P_{deopt}(|SpeculativeExcps|) = P_{opt}$，補題 A.1.7，および補題 A.1.8 より，脱最適化を用いる P_{opt} は任意の入力環境について P_{unopt} と同じ例外を発生させる．□

(平成 16 年 7 月 5 日受付)
(平成 16 年 9 月 20 日採録)

大平　　怜

2000 年東京大学理学部情報科学科卒業．現在，同大学院情報理工学系研究科コンピュータ科学専攻博士課程在籍．最適化/実行時コンパイラに関する研究に従事．他に仮想マシン，スレッドシステム，オペレーティングシステム，計算機アーキテクチャに興味を持つ．

平木　　敬（正会員）

1976 年東京大学理学部物理学科卒業．1982 年同大学院理学系研究科物理学専攻博士課程修了．理学博士．1982 年通商産業省工業技術院電子技術総合研究所入所．1988 年より 2 年間 IBM 社 T.J. Watson 研究センタ客員研究員．1990 年より東京大学理学部情報科学科（現在，大学院情報理工学系研究科コンピュータ科学専攻）に勤務．現在，超並列アーキテクチャ，超並列超分散計算，並列オペレーティングシステム，ネットワークアーキテクチャ等の高速計算システムの研究に従事．日本ソフトウェア科学会会員．