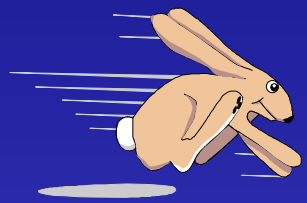
# Java Just-In-Time コンパイラにおける最適化手法



石崎　一明

日本アイ・ビー・エム（株）
東京基礎研究所

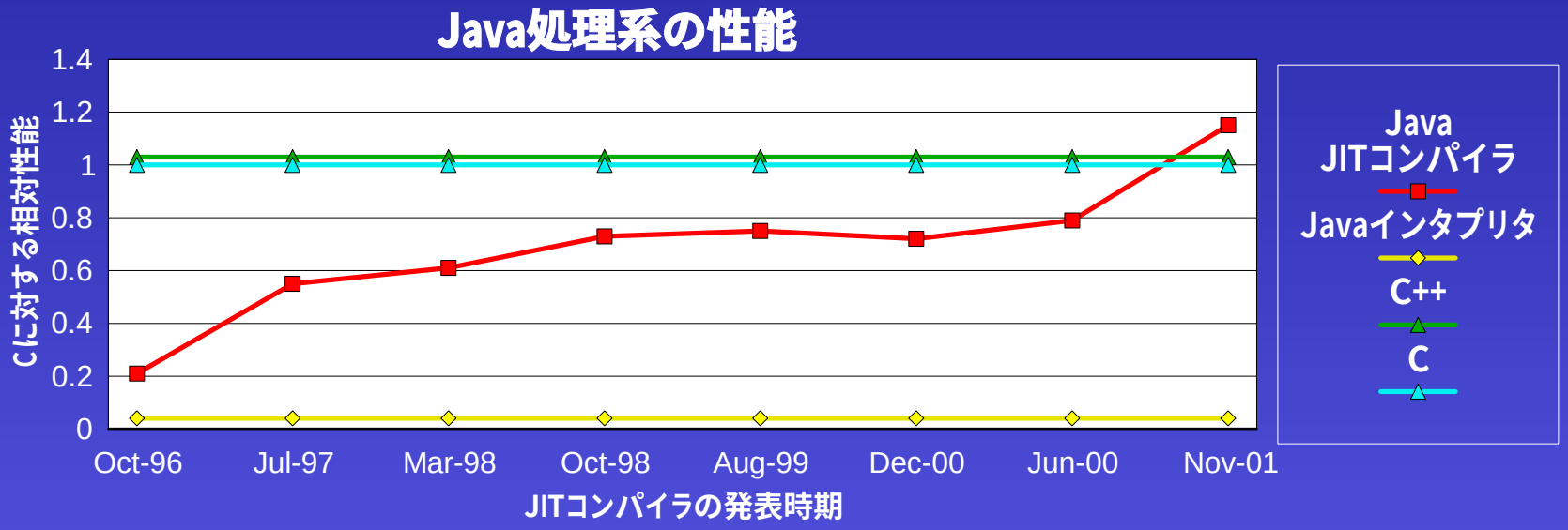
# Java言語の利点

- プログラマから見た利点
  - 仮想機械を用いた実行によるプログラムのプラットフォーム独立性。
  - オブジェクト指向と多態性を持つ動的束縛によるプログラミングの柔軟性。
  - プログラムの安全検査の実行と実行時型検査による型安全性がもたらすプログラムの安全性。

# Java言語の問題点

- 実行性能上の問題点
  - インタプリタである仮想機械上での実行による性能低下。
  - 多態性を持つ動的束縛の導入による、メソッド間に渡るコンパイラ最適化適用の困難さ。
  - 実行時型検査の導入による、検査命令実行の実行時オーバヘッド。

- C言語やC++言語と同等の性能が得られて初めて、Java言語は実用的な言語となる。




IBM

# 研究目的・成果・貢献

- 目的 - Java言語が持つ特徴に起因するオーバヘッドを削減する。

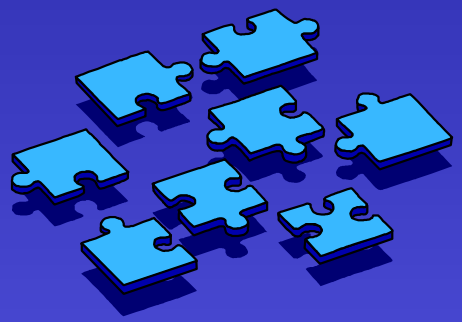
- 成果 - オーバヘッド削減のための
  コンパイラによる最適化の手法、実装、評価を示す。
  - 動的クラスロードを伴う仮想メソッド呼出の高速化。
  - 実行時のオブジェクトの型検査の高速化。
  - 配列要素やインスタンス変数の参照に伴う例外検査の高速化。
  - 安全な参照を保証するための命令間順序制約の緩和による高速化。

- 貢献
  - 上記の成果は、IBM Developers Kit, Java Technology Editionに実装されている。
  - 上記の成果の一部は、他のJava処理系にも実装されている。

IBM

# 仮想メソッド呼び出しの高速化

- 動的クラスロードを伴う仮想メソッド呼び出しの高速化。
- 実行時のオブジェクトの型検査の高速化。
- 配列要素やインスタンス変数の参照に伴う例外検査の高速化。
- 安全な参照を保証するための命令間順序制約の緩和による高速化。

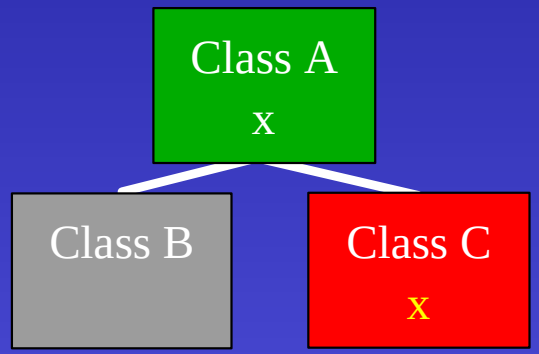


# 性能上の問題点

- 仮想メソッド呼び出しが多態性を持つ。
  - 呼び出し先を一意に特定することが難しくなる。
  - メソッドのインライン展開を行うことが難しい。
  - コンパイラの最適化の範囲が小さくなる。
- Java言語では、実行時にクラスの動的ロードが行われる。
  - コンパイル時には存在しなかったメソッドがコード生成後に現れる。

仮想メソッド呼び出しの例

```
Class A {
  x() { i = 1; }
  foo(A o) {
    o.x();          // a.x()またはc.x()を呼び出す
  }
}
Class B extends A {
}
Class C extends A {
  x() { i = -1; }
}
```

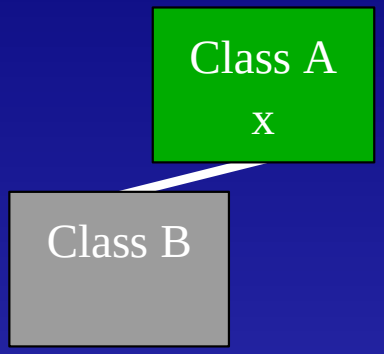


# 解決方法

- ***コード書換え*による直接デバーチャル化**
  - クラス階層解析の結果、呼び出し先を一意に特定可能ならば、
    デバーチャル化されたコードと仮想メソッド呼出のコードを生成し、
    前者を実行する。
    - メソッド呼出時にオーバヘッドがない
    - 合流点によるコンパイラ最適化への影響も最小限にする方法も提案する。
  - 動的クラスロードによってメソッドがオーバライドされた時、
    コードを書換えてデバーチャル化されたコードを無効化し、
    仮想メソッド呼出を実行する。
    - 実行中のメソッドのコンテキスト書き換えが必要な従来方式より、
      実装が容易である。

# 実行例

- メソッドxがオーバライドされていない時




```
Class A {
  x() { i = 1; }
  foo(A o) { o.x(); }
}
Class B extends A {
}
```

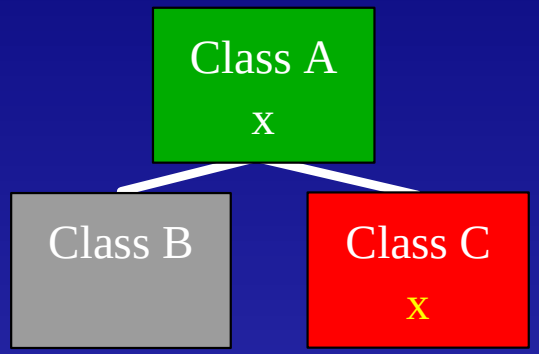
```
  // inlined code of A.x()
  store offset_i(r0), 1
after_inline:
  ret
  ...
dynamic_call:
  load      r1, offset_class_in_object(r0)
  load      r2, offset_method_in_class(r1)
  load      r3, offset_code_in_method(r2)
  call      r3
  jmp       after_inline
```

代替パス

# 実行例（cont'd)

- メソッドxがオーバライドされた時

```
Class A {
  x() { i = 1; }
  foo(A o) { o.x(); }
}
Class B extends A {
}
Class C extends A {
  x() { i = -1; }
}
```




```
  // inlined code of A.x()
  store offset_i(r0), 1
after_inline:
  ret
  ...
dynamic_call:
  load      r1, offset_class_in_object(r0)
  load      r2, offset_method_in_class(r1)
  load      r3, offset_code_in_method(r2)
  call      r3
  jmp       after_inline
```

```
  jmp       dynamic_call
after_inline:
  ret
  ...
dynamic_call:
  load      r1, offset_class_in_object(r0)
  load      r2, offset_method_in_class(r1)
  load      r3, offset_code_in_method(r2)
  call      r3
  jmp       after_inline
```

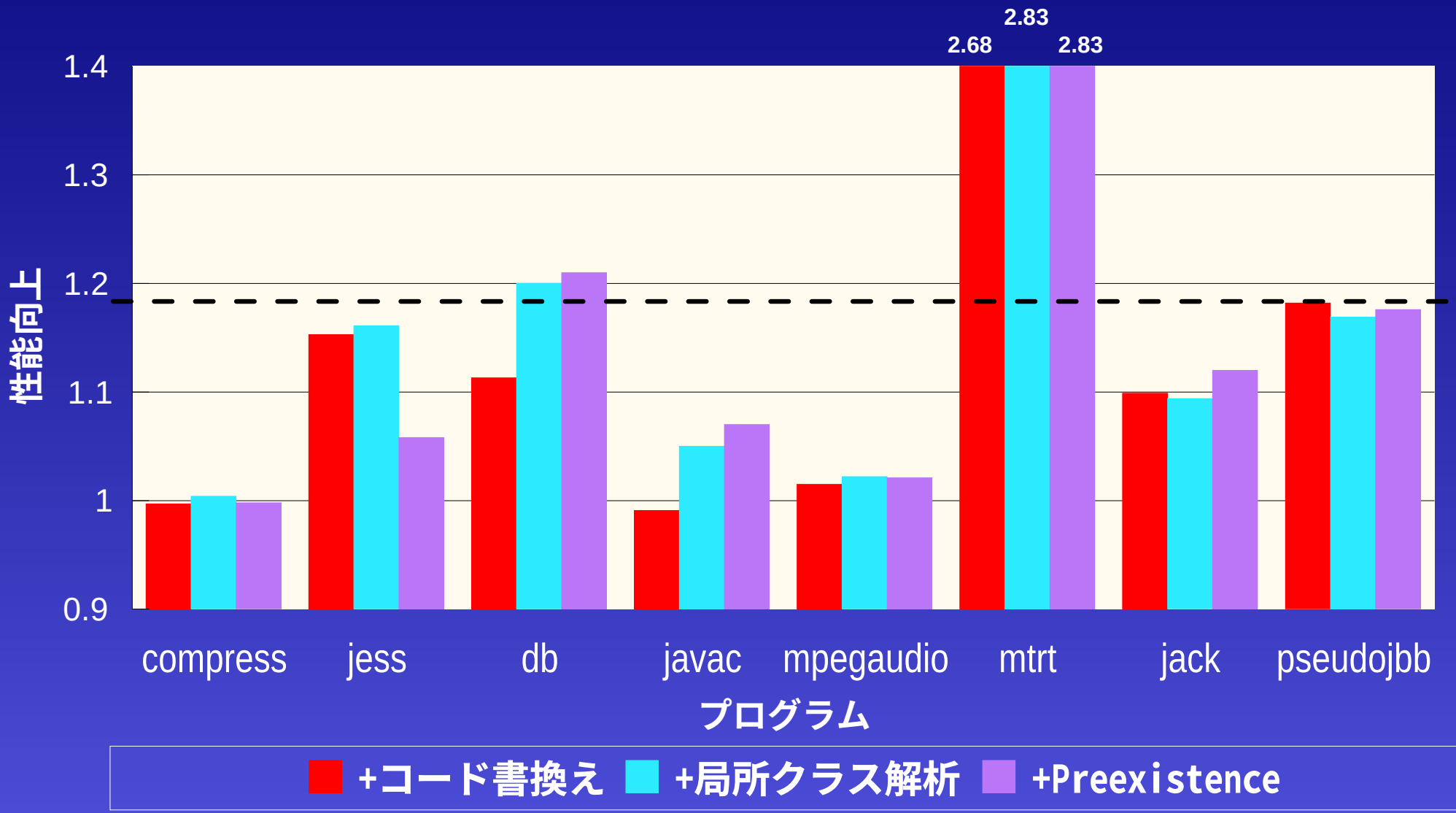
代替パス

メソッドのオーバライドが起きたとき、
生成されたコードを書き換えて、
デバーチャル化されたコードを無効化する

IBM

# 性能評価

- デバーチャル化無しを基準として平均で18%の性能向上



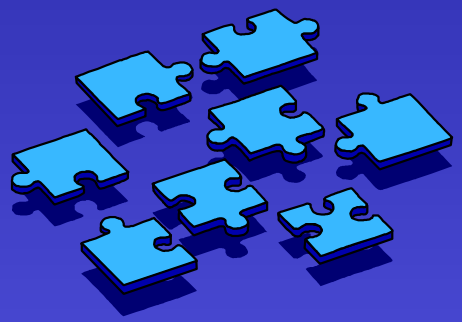

IBM

# メソッド呼び出し高速化のまとめ

- コード書換えによる直接デバーチャル化の提案。
  - 命令実行時のオーバヘッドが無い。
  - 実行中のメソッドのコンテキスト書き換えを必要とせず、実装が容易。
  - 適用有効範囲が広い。
- 簡単な実装にもかかわらず、かなりの性能改善をもたらす。
  - 他の処理系でも採用（Intel JUDO、IBM Jikes RVM on IA32）。
- 代替パスによって制御フロー上に合流点が生成されても、さらなる最適化を可能にする。
  - 局所クラス解析やpreexistenceによる、代替パスを必要としない直接デバーチャル化の適用。

# オブジェクトの型検査の高速化

- 動的クラスロードを伴う仮想メソッド呼び出しの高速化。
- **実行時のオブジェクトの型検査の高速化。**
- 配列要素やインスタンス変数の参照に伴う例外検査の高速化。
- 安全な参照を保証するための命令間順序制約の緩和による高速化。



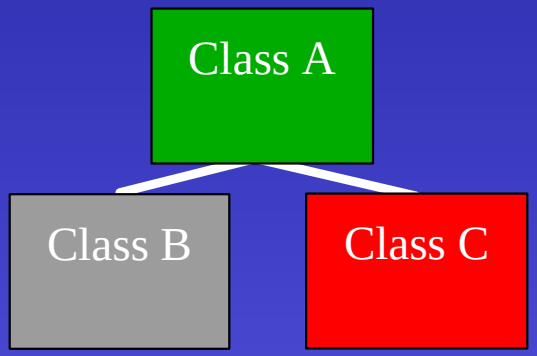
# 性能上の問題点

- 型安全性の保証のために、オブジェクトのキャストの際に実行時の型検査が必要。
  - C言語やC++言語には存在しないオーバヘッド。
- クラス階層を表で表現し、型検査を表引きで行う従来手法。
  - クラス構成にかかわらず、固定時間で実行可能な**利点**。
  - 表引きを行う命令列の実行時の**オーバヘッド**。
  - 各クラス毎に根クラスまでのクラス構成を持つ場所の**オーバヘッド**。

## 型検査の例

```
Type LHS = (Type)RHS;
```

| LHS | | RHS | |
|---|---|---|---|
| Class A | <- | Class B | 成功 |
| Class B | <- | Class A | 失敗 |
| Class C | <- | Class B | 失敗 |




IBM

# 解決方法

- 型検査のうち、頻繁に実行されるパスをコード内に展開する。
  - 本方式ではキャスト前後のクラスが等しい場合3命令、
    最後に型検査が成功したクラスと等しい場合が7命令で処理可能。
  - 従来方式では、常に7命令で処理可能。

**本方式**

```
  load   r1, TIBOffset(RHS)
  cmpi   r1, LHSTIBaddress
  b      eq, OK        // キャスト前後の型が同じか
  load   r1, ClassOffset(RHS)
  load   r1, LastSuccOffset(r1)
  cmpi   r1, LHSClassaddress
  b      ne, MayNG     // 最後に型検査が成功した
OK:                    // クラスと等しいか
  ...

MayNG:
  call   TICRuntime    // 実行時ルーチンで
  jmp    OK            // 型検査を行う
```

**必要な場所：** クラスに2ワード
型検査に成功したクラスと
失敗したクラス

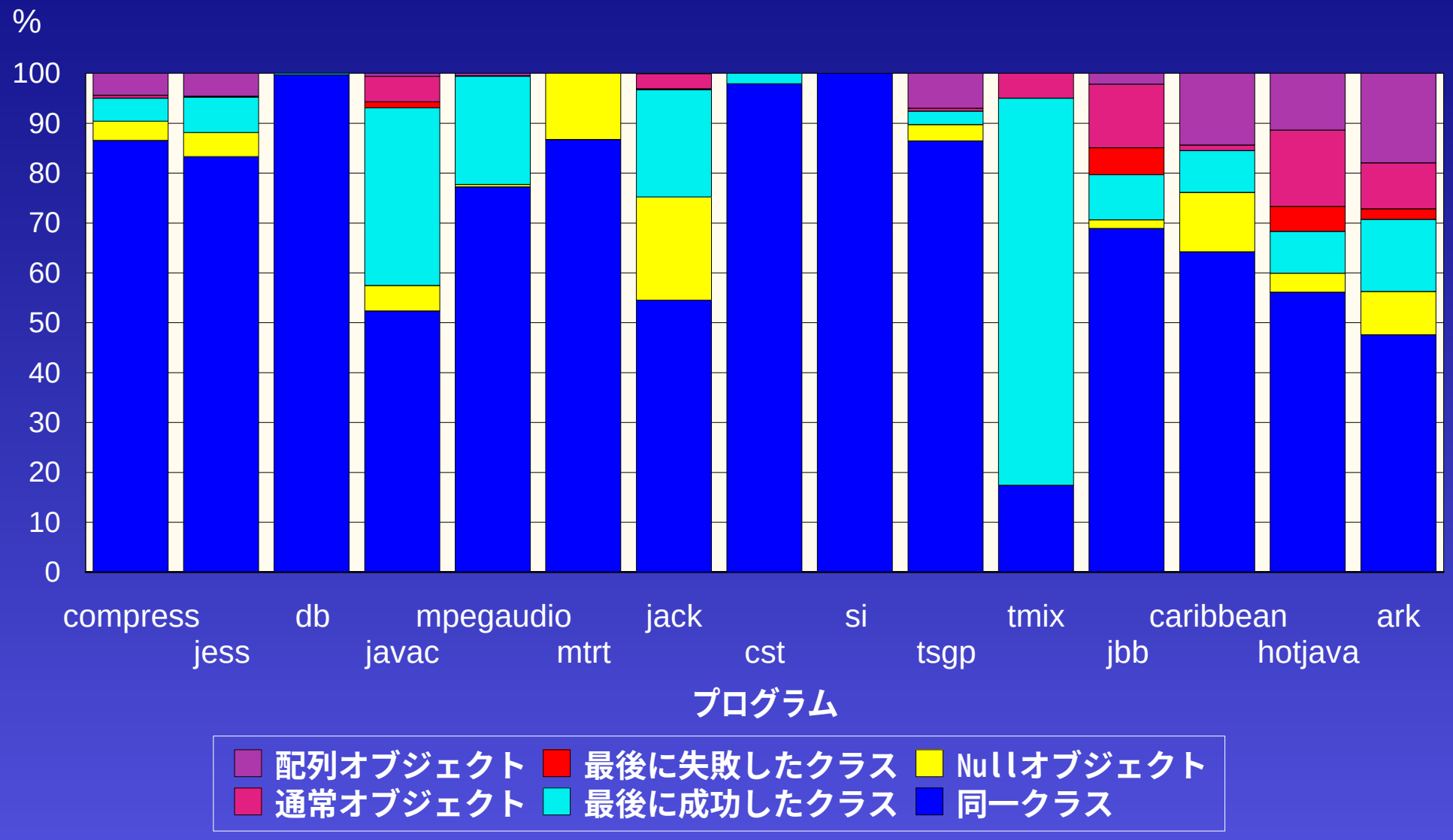
**従来方式**

```
  load   r1, TableOffsets(RHS)
  load   r2, RHSDepthOffset(r1)
  cmpi   r2, LHSDepth
  b      lt, NG        // 表の大きさの方が小さいか
  load   r1, LHSDepth(RHS)
  cmpi   r1, LHSID
  b      ne, NG        // キャスト前後のクラスの
                       // IDが等しいか
```

**必要な場所：** クラスにd+1ワード
自クラスから根クラスまでを
構成するクラスIDの表と大きさ

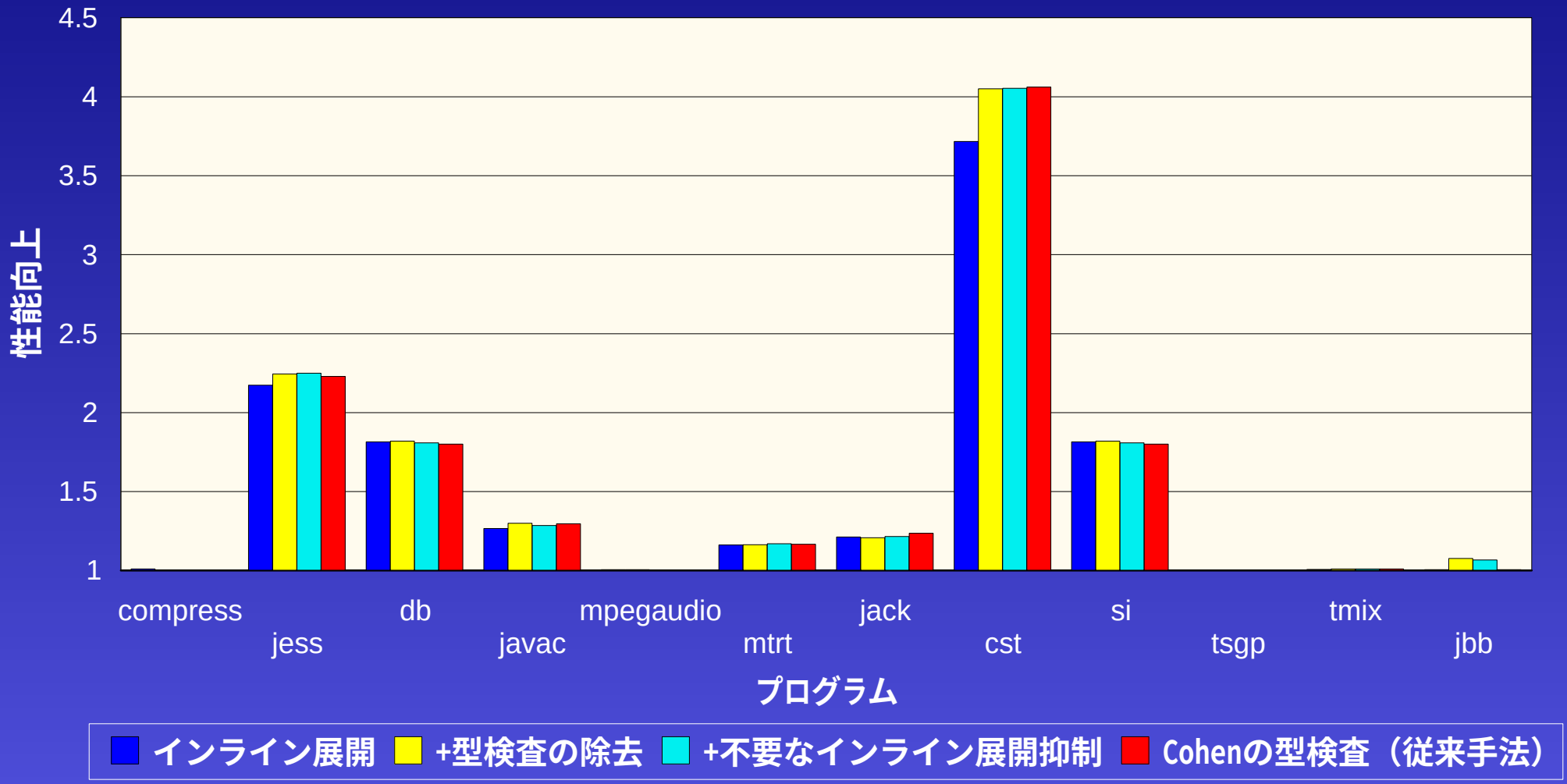
# プログラムの挙動

- 型検査が行われるオブジェクトの種類
  - 同一クラス、Nullオブジェクト、最後に成功したクラスで91%を占める



# 性能評価

- 本方式で36%の性能向上、従来手法で35%の性能向上。
  - 従来手法の厳密な実装、表スペースのオーバヘッドを考慮すると本方式が勝る。



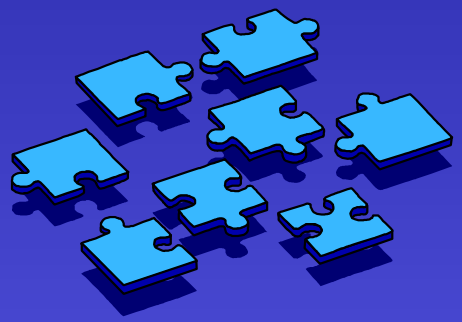

IBM

# 型検査の高速化のまとめ

- 型安全性の保証のために、実行時のオブジェクトのキャスト時に型検査が必要。
  - C言語やC++言語には存在しないオーバヘッドであり、削減が必要。
- 型検査処理において、プログラムの挙動から得られた頻繁に行われるパスをインライン展開する手法を提案。
  - 従来の表引き手法と同等以上の性能向上を実現。
  - 従来の表引き手法より場所のオーバヘッドが少ない。

# 例外検査の高速化

- 動的クラスロードを伴う仮想メソッド呼び出しの高速化。
- 実行時のオブジェクトの型検査の高速化。
- 配列要素やインスタンス変数の参照に伴う例外検査の高速化。
- 安全な参照を保証するための命令間順序制約の緩和による高速化。



# 性能上の問題点

- 安全なメモリ参照を実現するために、オブジェクトへのアクセス時にアドレスの例外検査が行われる。
  - C言語やC++言語には存在しないオーバヘッド。
- 例外検査のオーバヘッド削除を行う従来手法。
  - コンパイル時の最適化で例外検査を除去。
    - 全ての例外検査が除去できるわけではない。
  - OSのメモリ保護機能を利用して、例外検査のコードを生成しない。
    - コンパイラの最適化支援のために、0ページにメモリ保護機能を提供しないOSがある。

# 解決方法

- 除去不可能な例外検査命令を、高速に実行。
  - プロセッサが提供する高速な条件分岐命令の利用。
    - 条件が成立したときに、システムトラップベクタへ分岐するtrap命令。
  - 例外が発生しないときに実行される命令を最小限にする。
    - 例外の種類を例外検査命令へ埋め込み、例外ハンドラでデコードする。

```
j = a[i];
```

**本方式によるネイティブコード**

```
trapi  EQ, r5, 0    // Nullポインタ検査
trap   LLE, r6, r4  // 配列インデックス検査
slwi   r4, r4, 2
lwzx   r3, r4(r5)   // 配列要素へのアクセス
```

**従来方式によるネイティブコード**

```
li     r0, 1        // Nullポインタ検査flag
trapi  EQ, r5, 0    // Nullポインタ検査
li     r0, 2        // 配列インデックス検査flag
trap   LLE, r6, r4  // 配列インデックス検査
slwi   r4, r4, 2
lwzx   r3, r4(r5)   // 配列要素へのアクセス
```

**システムトラップベクタ**

```
iar =  trap命令の条件が成立したアドレス
if (*iar == 'trapi EQ, rx, 0') NullPointer例外の発生
if (*iar == 'trap LLE, rx,ry') ArrayOutOfBounds例外の発生
```

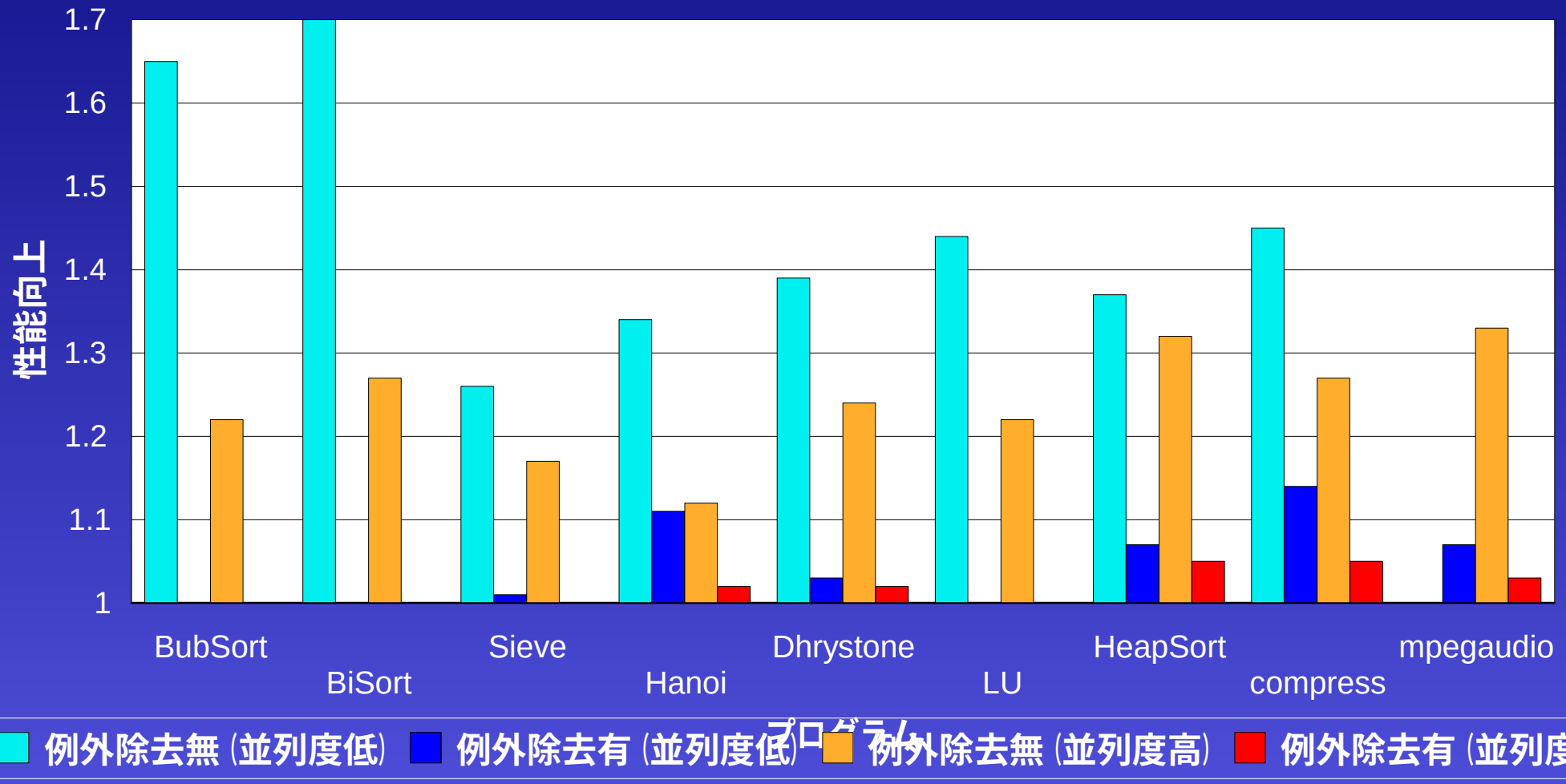
**システムトラップベクタ**

```
if (r0 == 1) NullPointer例外の発生
if (r0 == 2) ArrayOutOfBounds例外の発生
```

IBM

# 性能評価

- 配列要素を参照する、例外検査が多いプログラムを実行。
  - 並列度の低いプロセッサでは、最適化による例外除去後も最高14%の性能向上
  - 並列度の高いプロセッサでも、最適化による例外除去後も最高6%の性能向上



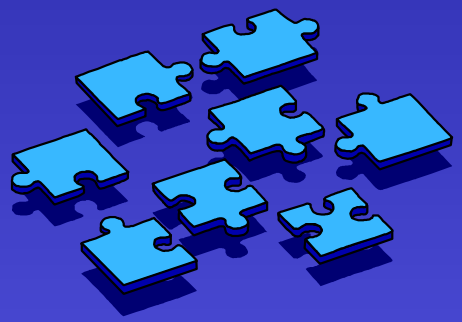

IBM

# 例外検査の高速化のまとめ

- 安全なメモリ参照を実現するために、オブジェクトへのアクセス時にアドレスの例外検査が行われる。
  - C言語やC++言語には存在しないオーバヘッドであり、削減が必要。
- 例外検査命令の実行時の高速化を実現。
  - プロセッサが提供する高速な条件分岐命令の利用。
  - 例外の種類を検査命令に埋め込むことで、
    例外が発生しない場合に実行される命令を最小限にする。
- 並列度が異なるプロセッサ上で効果を確認。
  - 例外検査除去の最適化を適用後も、最高6～14%の性能向上が得られる。

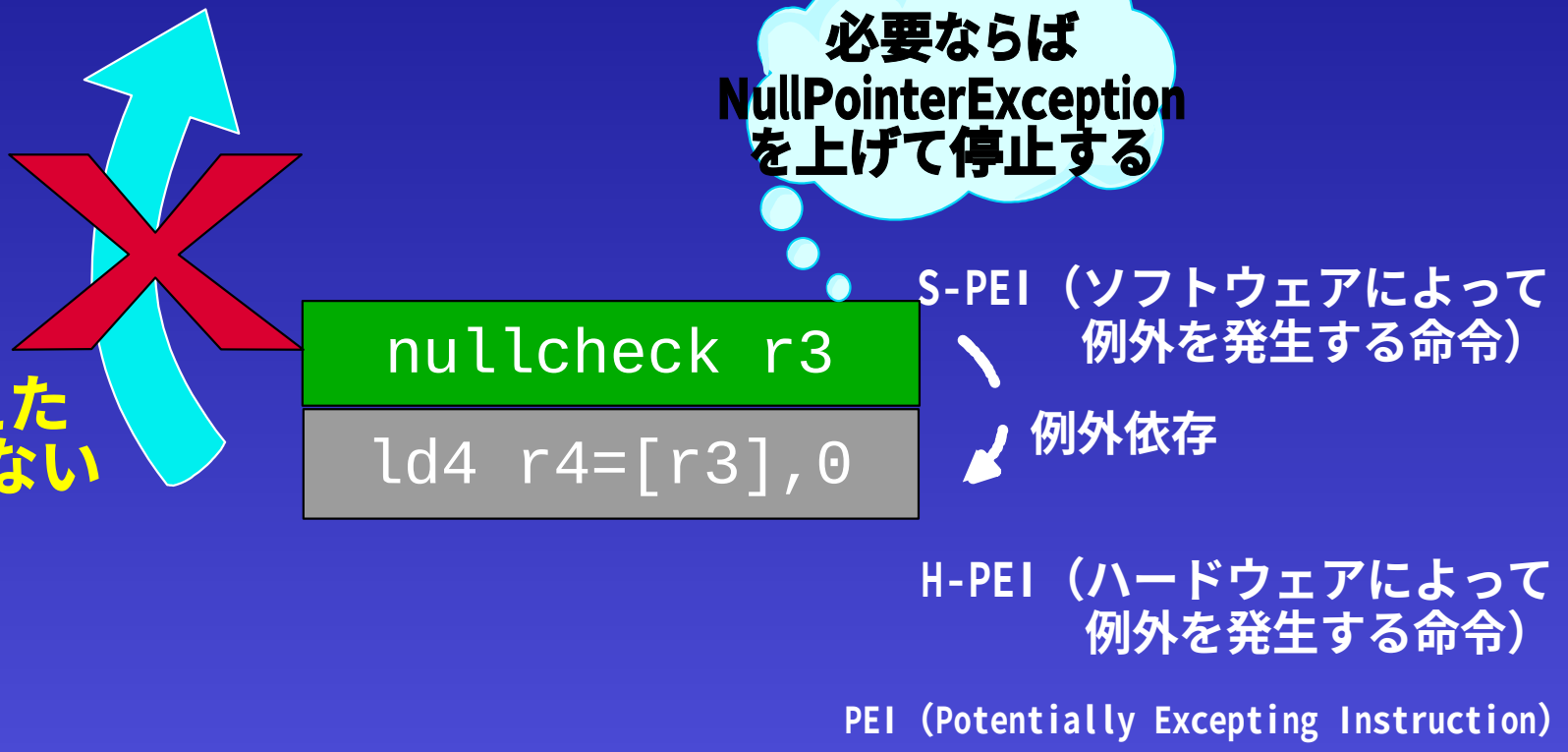
# 安全な参照の高速化

- 動的クラスロードを伴う仮想メソッド呼び出しの高速化。
- 実行時のオブジェクトの型検査の高速化。
- 配列要素やインスタンス変数の参照に伴う例外検査の高速化。
- 安全な参照を保証するための命令間順序制約の緩和による高速化。



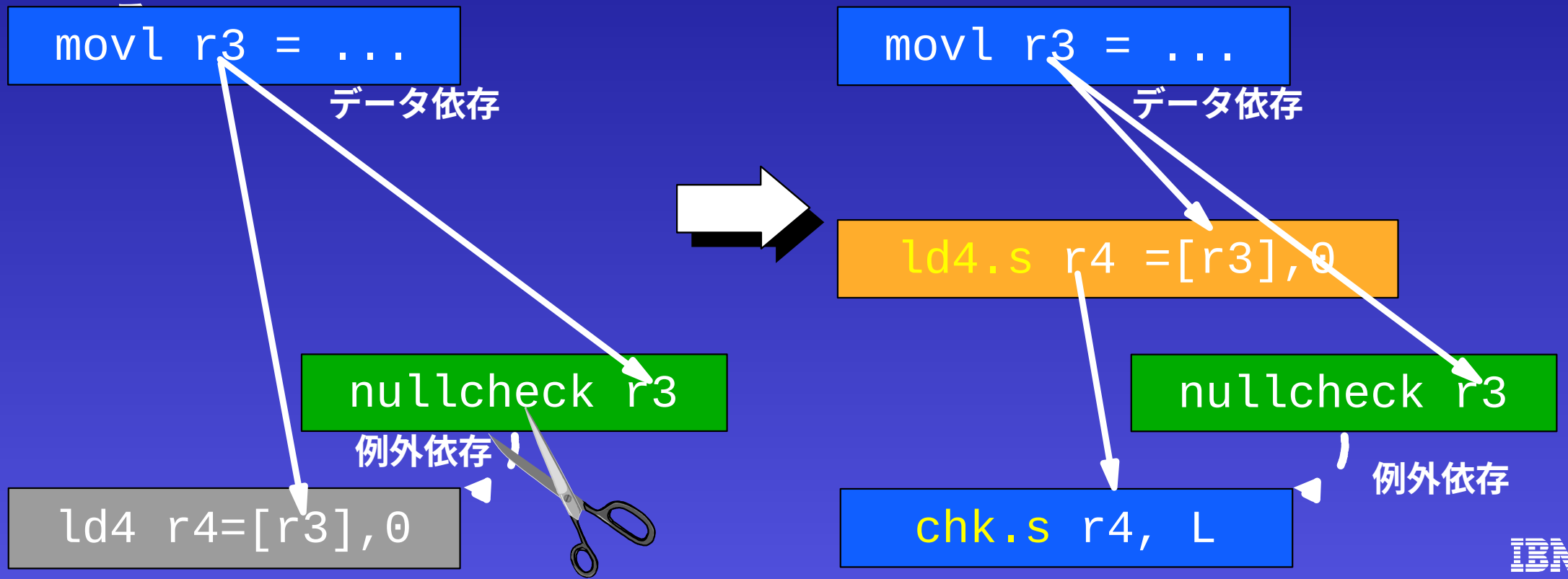
# 性能上の問題点

- 型安全な言語を実現するためのアドレス例外検査。
  - 不正なメモリアクセスを起こさない。
  - プログラマが回復できない異常なプログラム終了を起こさない。
    - S-PEIとH-PEI間の例外依存（nullcheckとld4）が命令移動の妨げとなる。



# 解決方法

- 例外依存辺の除去による、H-PEIの投機的命令移動。
- 例外依存辺を導入したDAGフレームワークの利用。
  - 例外依存を辺で表現することで、基本ブロックの粒度を大きく保つ。
- 投機命令移動の適用判断を正確に行う。
  - 投機命令移動による利得があると判断した場合に例外依存辺を除去す



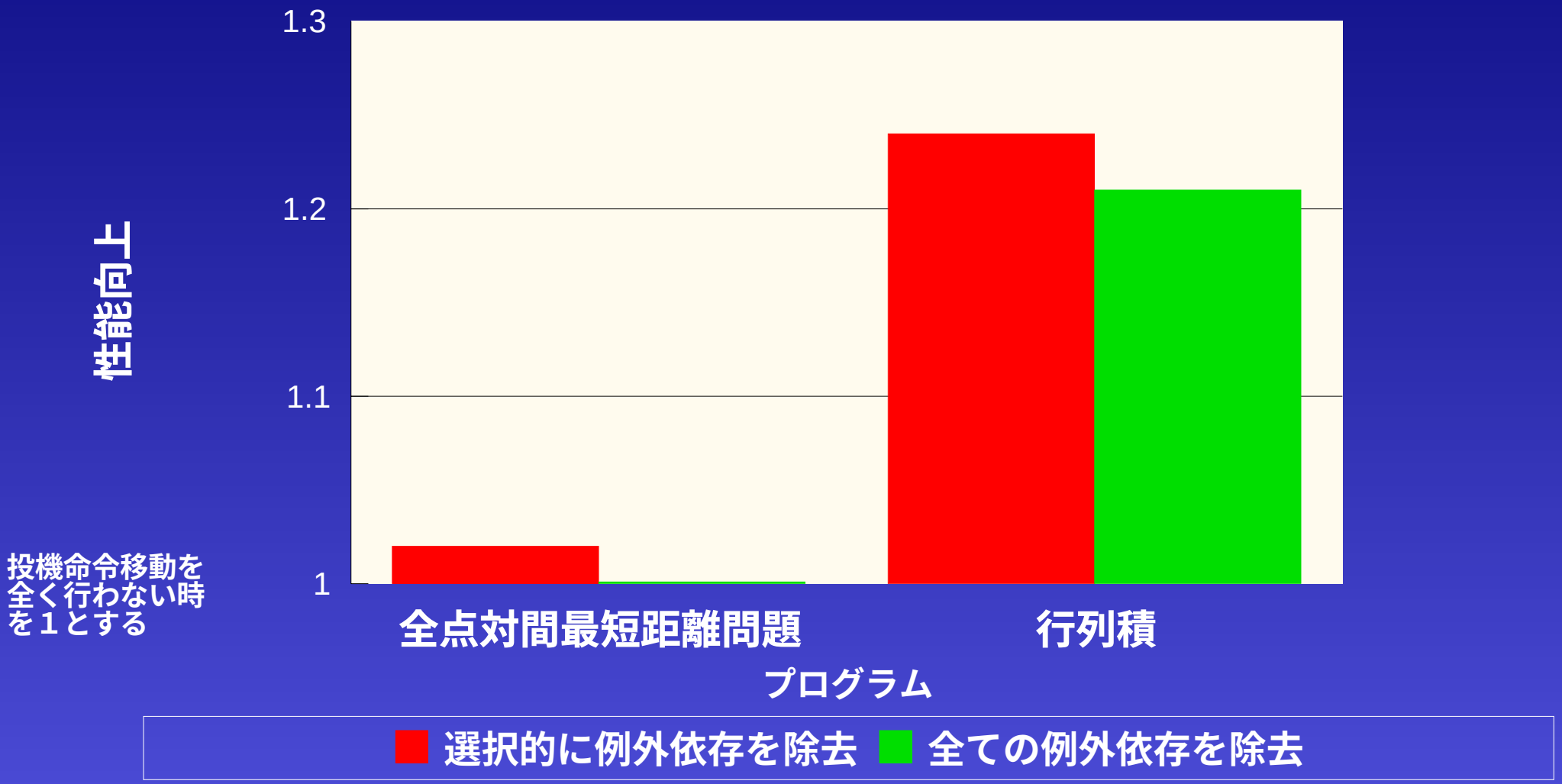

IBM

# コンパイル時の課題

- 効果的な投機移動命令の選択。
  - 副作用を持つ命令を除外。
  - 変数の生存区間を増大させる命令を除外。
  - 循環グラフを生成する命令を除外。
  - 復旧コードと変数の生存区間を干渉させる命令を除外。
- 復旧コードの生成。
  - メインコードへのレジスタ干渉を最小化。
  - 同時実行可能な命令群生成時の制約除去。

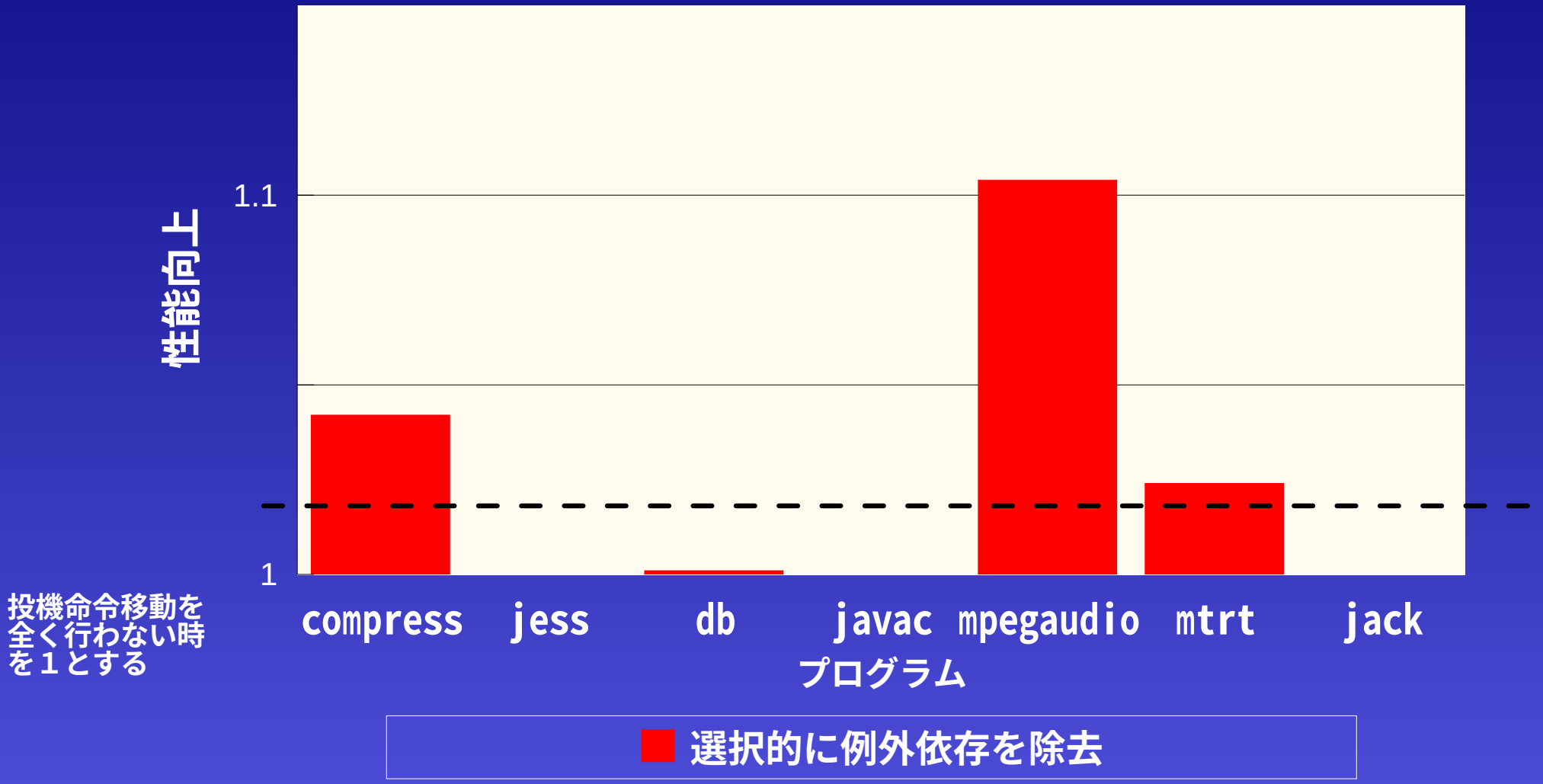
# 性能評価

- 投機的命令移動が可能なIA-64で小規模なベンチマークを実行。
  - 選択的に例外依存を除去した方が効果が大きい。



# 性能評価(cont'd)

- IA-64でSPECjvm98を実行。
  - 配列参照が多いベンチマークで性能向上が大きい（2〜11%）。



# 安全な参照の高速化のまとめ

- 安全な言語における最適化上の問題を示し、解決方法を示した。
  - 例外依存による命令間の制約を除去する、DAG上での選択的な投機命令移動方法。
  - 投機命令移動を行う場合のコンパイル時の課題。
    - 投機移動を行う命令列の選択。
    - 復旧コードの生成方法。
- 本手法を実装し、従来のシミュレータによる実験ではなく、実際の処理系において効果があることを示した。
  - 配列参照の多いプログラムでは2〜11%の性能向上。

# 論文のまとめ

- Java言語が持つ特徴に起因するオーバヘッドを削減するコンパイラ最適化の手法、実装、評価を示した。
  - コード書き換えを用いた直接デバーチャル化による、動的クラスロードを伴う仮想メソッド呼出の高速化。
  - インライン展開を用いた、オブジェクトの型検査の高速化。
  - 実行命令の最小化による、配列要素やインスタンス変数の参照に伴う例外検査の高速化。
  - 投機命令移動を用いた、安全な参照を保証するための命令間順序制約の緩和による高速化。

- 本論文で提案された最適化手法は、IBM社の製品であるJava Just-In-Timeコンパイラにおいて実際に使用されている。
  - コード書き換えを用いた直接デバーチャル化は、他のJava処理系でも使用されている。
    - IBM社　Jikes RVM
    - Intel社 JUDO

IBM