Java 言語に対する効果的な Null チェックの最適化手法

川人基弘† 小松秀昭† 中谷登志男†

我々は Java 言語で書かれたプログラムから Null ポインタチェックの最適化を行う新しいアルゴリズムを提案する．この手法は Java に限らず，Null チェックを必要とする言語に対して適用可能である．我々のアルゴリズムは Null チェックを実行とは逆向きに移動させ，冗長な Null チェックを除去する．この最適化により，コード移動を妨げる多くの Null チェックを除去し，他の最適化の機会を増やすことができる．また，この最適化を他の最適化と協調して実行させることにより，お互いの最適化効果を最大限に高めることができる．さらに，Null チェックを実行方向に移動させ，多くの Null チェックをハードウェア trap で代用し，Null チェックの実行コストを最小限にする．我々はこのアルゴリズムを IBM Java Just-in-Time（JIT）compiler に実装して評価を行った．その結果，我々の手法により以前の手法と比較して，jBYTEmark で最大 71%，SPECjvm98 で最大 10%パフォーマンスを改善することができた．

Effective Null Pointer Check Optimization for Java

Motohiro Kawahito,† Hideaki Komatsu† and Toshio Nakatani†

We present a new algorithm for optimizing null pointer checks from programs written in Java. The same approach should work for any languages requiring null checking. Our new algorithm moves null checks backwards and eliminates redundant null checks. This increases the opportunities for other optimizations to be applied by eliminating many null checks that impede code motion. This also greatly improves the effect of optimizations by means of teaming up with other optimizations. In a separate pass, it moves null checks forwards and converts many null checks to hardware traps in order to minimize the execution costs for the remaining null checks. This algorithm has been implemented in the IBM Java Just-in-Time (JIT) compiler. Our experimental results show that our approach improves performance by up to 71% for jBYTEmark and up to 10% for SPECjvm98 over previously known algorithms.

1. はじめに

Java 言語[1]は優れた例外処理機構を持っており，これはエラー処理，プログラムの制御，安全性の向上などに役立つ．しかし，例外を起こしうる命令は制御を変更する命令となるために，プログラムをコンパイルする際に最適化を抑制する．Java 言語で書かれたプログラムは，一般的に例外を起こしうる命令を多く含み，これらの命令はコード移動の際に障害となり，最適化の範囲を非常に狭める．なかでも Null ポインタチェックはインスタンス変数，メソッド呼び出し，配列に対するアクセスについて必要であり，これらの命令は通常の Java のプログラムで非常に多く使われる．

Null チェックの実装方法として，ハードウェアや OS の機能を利用し，Null チェックを行うためのコードを生成しない方法[2)~4)]が一般的である．最近の OS は最初のページ（アドレス 0）に対してメモリアクセスを行うと，アクセスしたアプリケーションに対して例外を通知する機構が備わっているため，明示的な命令を生成せずに，Null ポインタをチェックすることができる．しかし，このような実装を用いても，Null チェックの除去は次の 2 つの点で重要である．1 つめはこのような実装を用いても，Null チェックは最適化を行う際の障害となり，最適化の範囲を非常に狭めるという点である．2 つめはすべての Null チェックについて，ハードウェアのサポートを頼れない場合があるという点である．たとえば，ある OS ではオブジェクトが Null のときの，メモリをアクセスする際のオフセットが，ある値よりも大きいと例外を起こさない．また，AIX はアドレス 0 に対しての read アクセスは例外を起こさない．もう少し複雑な例として，devirtualization[2),5)~7)]を適用した場合があげられる．devirtualization とは，本来実行時に初めて呼び出し先が決定できるようなダイ

† 日本アイ・ビー・エム株式会社東京基礎研究所
Tokyo Research Labolatory, IBM Japan, Ltd

ナミックなメソッド呼び出しを，コンパイル時に呼び出し先を決定できるように変形を行う最適化である．メソッドを特定することにより，メソッドの呼び出しコストを減らし，さらにメソッドインライニングが行えるようになる．しかし，この変形を行うことにより，メソッドテーブルのアクセスが行われない場合があるために，メソッドテーブルのアクセスに対応するNullチェックは，明示的な命令生成を行わなければならない．インラインするメソッドが小さく，Nullチェックに必要な実行コストとインラインしたメソッド内の実行コストが大差ない場合（アクセスメソッドなど）もあるために，このようなNullチェックを除去することは重要である．

従来のNullチェックの最適化の流れは，まず前方データフロー解析（forward data-flow analysis）を使ってオブジェクトがNullではない範囲を求め，Nullチェックを行わなければならない命令について実際にチェックが必要かどうかを記録していた．そしてコード生成を行う時点で実際にNullチェックが必要な命令について，ハードウェアtrapで代用できないものは明示的なチェックを行う命令を生成していた．しかし，従来の方法ではNullチェックの移動を行っていないため，例外を起こす命令がscalar replacement[8]などの最適化を行う際の障害となり，最適化の範囲を非常に狭めるため，最適化の効果を抑制する．ここで，scalar replacementとは，インスタンス変数や配列などのグローバルなエリアに対するアクセスを，副作用を起こす命令を超えない範囲内でメソッド内のローカルなエリア（たとえばローカル変数）に対するアクセスに置き換える最適化を意味する．

我々はこの問題を解決するために，Nullチェックの最適化を2段階で行う．

1段階目はアーキテクチャと独立した最適化である．本来，JavaのNullチェックによる例外は，Nullチェックを行わなければならない命令で発生する．我々はNullチェックの移動を行えるように，中間コード上でNullチェックを行わなければならない命令とNullチェックを分割した．この最適化は，partial redundancy elimination（PRE）[9)~11)]を用いて，Nullチェックを実行とは逆向きに移動することによる副作用が観測できない範囲内で，Nullチェックを移動させる．副作用が生じる例としては，Nullチェックの対象変数への書き込み，他の種類の例外，メモリへの書き込みなどの命令を超えてNullチェックを移動した場合があげられる．このようなNullチェックの最適化によって，他の最適化がより広い範囲に対して適用できるようになる．また，scalar replacementの最適化を行うと，Nullチェックの移動を阻害するNullチェックの対象変数への書き込みがプログラム内から除去されるため，Nullチェックの最適化の機会が増える．そこで，Nullチェックの最適化と他の最適化を繰り返して実行させることにより，お互いの最適化効果を最大限に高めることができる．

Nullチェックの最適化の2段階目はアーキテクチャに依存した最適化である．この最適化は，1段階目の最適化で残ったNullチェックの実行コストを最小限にするために逆向きにPREを使い，Nullチェックを実行方向に移動させ，Nullチェックをハードウェアのtrapで代用する．

我々はこのアルゴリズムをIBMのJava Just-in-Time（JIT）コンパイラに対して実装を行った．我々のJITコンパイラはIntel IA32，PowerPC，S/390をサポートしており，我々はこのアルゴリズムをこれらすべてのアーキテクチャに対して適用している．我々はjBYTEmark v.0.9とSPECjvm98のベンチマークを用いて，Pentium III 600 MHzのマシン（Windows NT 4.0，IBM Developer Kit for Windows(R)，Java Technology Edition，Version 1.2.2）およびPowerPC 604e 332 MHzのマシン（AIX 4.3.1，IDK for AIX，Version 1.2.2）上でアルゴリズムの評価を行った．この結果，以前のアルゴリズムと比べて大きくパフォーマンスが改善した．

1.1 本アルゴリズムの特徴

- partial redundancy eliminationを利用した，より強力なNullチェックの最適化を行う．
- Nullチェックの最適化と他の最適化を繰り返して実行させることにより，お互いの最適化効果を最大限に高める．
- ハードウェアのサポートを存分に利用し，Nullチェックの実行コストを最小限にする．

1.2 実験に基づく評価

我々の実験では，このアルゴリズムは以前のアルゴリズムに比べて，最大jBYTEmarkで71%，SPECjvm98で10%パフォーマンスを向上させることができた．また，JITのコンパイル時間は2.3%程度の増加にとどまった．

本稿の以下の構成は次のようになっている．2章では従来の研究について述べる．3章では我々の手法の概略を述べる．4章では我々のアルゴリズムの詳細について述べる．5章では我々の手法の評価を行う．

2. 従来の研究

2.1 Null チェックの実装

いくつかの JIT コンパイラ，たとえば Jalapeño Dynamic Optimizing Compiler[3]，LaTTe JIT compiler[4]，我々の JIT コンパイラ[2),5)]は，ハードウェアと OS の機能を利用した Null チェックの実装を行っている．Jalapeño のオブジェクトレイアウトは，彼らのターゲットアーキテクチャ（AIX，PowerPC）が最初のページに対しては，メモリ書き込みしかハードウェアで検出できないため，メモリの読み書き両方で検出できるように以下の工夫を施している．Jalapeño はオブジェクトの内容にアクセスする際に，オブジェクトのポインタアドレスから負のオフセットを使ってアクセスする．彼らは AIX 上では最後のページに対するメモリ読み書き両方が，ハードウェアで検出できることを利用している．LaTTe（彼らのターゲットアーキテクチャは SPARC）は Null ポインタのときのすべてのオブジェクトアクセスは，ハードウェアで検出できることを利用している．Jalapeño と LaTTe はともにすべての Null チェックがハードウェア trap で代用できることを仮定している．しかし，devirtualization によるメソッドインライニングなどのコードの変形を行うことにより，このような仮定ができない場合がある．このような場合は，Null チェックを行うためのコードを明示的に生成しなければならない．そのために，実行速度が遅くなることが問題となる．我々の Null チェックの実装方式については 3.3.1 項で詳しく述べる．

2.2 前方データフロー解析を用いた Null チェック除去

以前の JIT コンパイラ，たとえば Jalapeño[12)]や以前の我々の JIT コンパイラ[2),5)]は，前方データフロー解析を使って Null ポインタチェックを除去していた．このアルゴリズムは，データフローに沿ってすでにチェックを行った Null チェックを除去する．しかし，この手法は次の 2 つの問題点があった．

- このアルゴリズムではループ不変の Null チェックを移動できない．たとえば，最初のオブジェクトアクセスがループ内にある場合には，その Null チェックはループ内に残ることになる．このような Null チェックは他の最適化の障害となり，最適化の範囲を非常に狭くする．
- このアルゴリズムはハードウェアサポートの利用を考慮に入れていない．

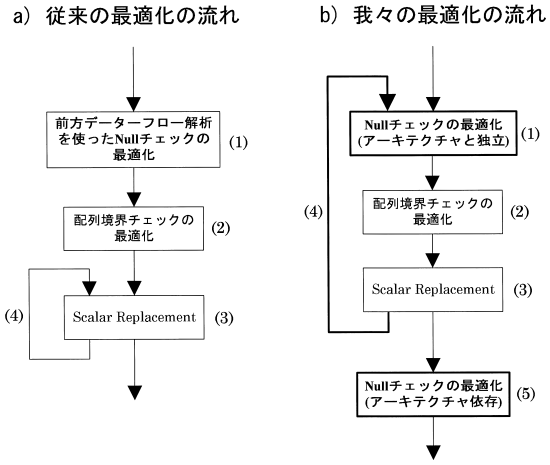


図 1 Null チェックの最適化の流れ図
Fig. 1 High-level flow diagram of null check optimization.

3. 我々の手法の概略

本章では 2 章で述べた問題点を我々がどのように解決しているのかを述べる．3.1 節では Null チェックの最適化の流れを述べる．3.2 節では 2.2 節で述べた最初の問題を解決する「アーキテクチャと独立した最適化」の概略について述べる．3.3 節では 2.1 節と 2.2 節の 2 つめで述べた問題を解決する「アーキテクチャに依存する最適化」の概略について述べる．

3.1 Null チェックの最適化の流れ

図 1 は Null チェックの最適化の流れ図である．我々の Null チェックの最適化 (b) は 2 段階に別れる．1 段階目はアーキテクチャと独立した最適化 (1)，2 段階目はアーキテクチャ依存の最適化 (5) である．アーキテクチャと独立した最適化 (1) では，Null チェックを実行とは逆方向に移動させ，冗長な Null チェックを除去する．この最適化は従来行われていた手法 (a) (1) よりも強力な最適化である．また，この最適化によって配列境界チェックの最適化 (2) や scalar replacement (3) の最適化の機会を増やす．また，(2)，(3) の最適化も Null チェックの最適化 (1) の機会を増やす．そのため，繰返し (4) を行うことによって最適化の効果をさらに高められる．従来の方法では，この繰返し (a) (4) は，(3) から (3) への繰返しとなっていた．我々の手法の特徴はこれを (3) から (1) への繰返しとした点で，これにより最適化するプログラムによっては非常に効果的に作用する．さらに，アーキテクチャ依存の最適化 (5) は Null チェックを実行方向に移動させ，ハードウェア trap で代用することにより，Null チェックの実行コストを最小限にする．

3.2 アーキテクチャと独立した最適化

我々は，冗長な Null チェックの除去や Null チェックをループの外に移動させるために，partial redundancy elimination（PRE）[9)~11)]を利用している．

図 2 は部分的冗長性を持つ Null チェックを最適化した場合の例を示している．図 2 (1) 内の 2 つの Null チェックは従来の前方データフロー解析では，右側のパスが Null チェックを含んでいないため，最適化できなかった．この最適化は，まず Null チェックを実行とは逆方向に移動できる領域を求める．このとき移動の障害となる命令としては，Null チェックの対象変数への書き込み，他の種類の例外，メモリへの書き込みがある．次に Null チェックの挿入点を求める．あるベーシックブロック B の終わりの点が，ある Null チェック C の移動可能な領域内にあり，B の直前のベーシックブロックの終わりの点が領域外のときは，ベーシックブロック B の終わりの点を Null チェック C の挿入点とする．たとえば，図 2 (1) に適用すると図 2 (2) の領域，および挿入点が求まる．次に，挿入点に Null チェックを挿入した場合を仮定し，オブジェクトが Null ではないと判断できる領域を求める．そして，その領域内にある Null チェックおよび挿入点の除去を行う．図 2 (2) に適用すると図 2 (3) の領域が求まり，左側のパス内にある挿入点，およびパスの合流点にある Null チェックが除去される．最後に残った挿入点に Null チェックを挿入する．そして，図 2 (4) の最適化結果が求まる．図 2 (1) と図 2 (4) を比べると，図 2 (4) のほうが左側のパスについて Null チェックの実行回数が 1 回減っていることが分かる．

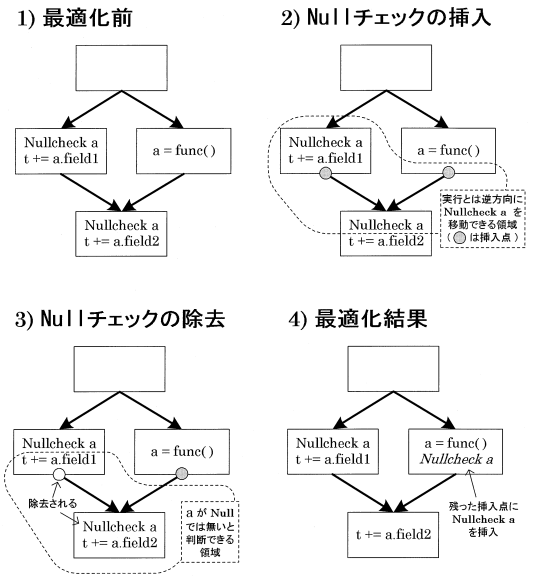
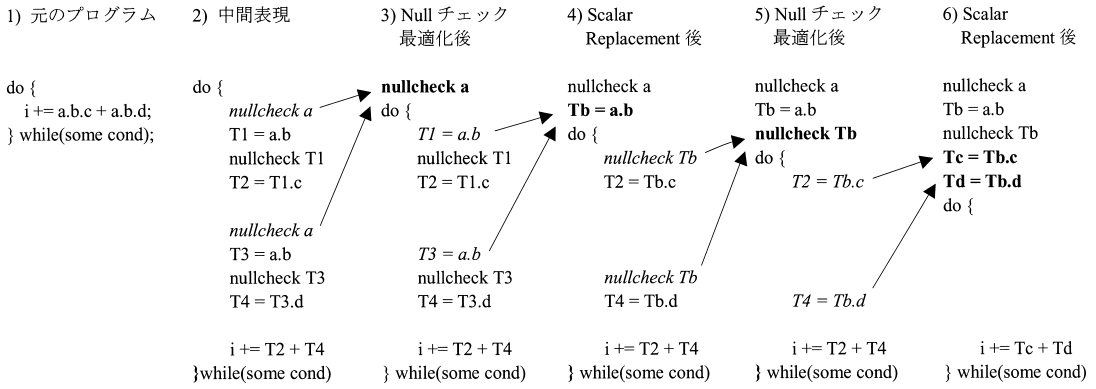


図 2 部分的冗長性を持つ Null チェックの最適化
Fig. 2 Optimization of partially-redundant null check.

図 3 は図 1 (b) (4) の繰返しによって，アーキテクチャと独立した Null チェックの最適化と scalar replacement がお互いに助け合う例である．図 3 (2) 内の最初の nullcheck a はループの外側に Null チェックを含んでいないため，前方データフロー解析のような以前の手法では除去できない．ところが我々の手法では図 3 (3) のように nullcheck a をループの外へ移動させる．Null チェックはそれに関連したメモリアクセスの命令移動にとって障害となるため，図 3 (4) の結果は Null チェックの最適化結果 (3) を基にしなければ得られない．また，Null チェックの対象となる変数への書き込みは，その Null チェックのコード移動にとって障害となる．そのため，図 3 (3) に対して Null チェックの最適化を行っても，T1 や T3 に対する Null



(仮定：a 及び i はローカル変数)

図 3 アーキテクチャと独立した Null チェックの最適化と scalar replacement
Fig. 3 The architecture independent null check optimization and scalar replacement.

チェックを移動する際に，直前のそれぞれの変数への書き込みが移動の障害となるため，ループ内の Null チェックは最適化されない．よって，図 3 (5) の結果は，図 3 (4) の scalar replacement の最適化結果を基にしなければ得られない．このように，図 1 (b)(4) の繰返しによる最適化はループ不変の命令移動に対して特に効果が大きい．

3.3 アーキテクチャ依存の最適化

3.3.1 暗黙的な Null チェックと明示的な Null チェック

本項では我々の Null チェックの実装方式について説明する．我々は次の 2 つの実装方式を使っている．

- **Implicit Nullcheck**：ハードウェア trap に頼ることにより，コード生成が不要な Null チェック命令．
- **Explicit Nullcheck**：コード生成が必要な Null チェック．Implicit Nullcheck で実装できないときには，この Explicit Nullcheck を生成する．

Null チェックを実装する際には，極力 Implicit Nullcheck を使う．しかし，Java の言語仕様を維持するために，Explicit Nullcheck を生成しなければならない場合がある．たとえば，Null チェックを必要とする命令がハードウェア trap を起こさないとき，その Null チェックは Explicit Nullcheck でなくてはならない．

図 4 は Explicit Nullcheck を生成しなければならない例である．図 4 (1) の b.BigOffset のように，オブジェクトが Null のときのメモリアクセスがハードウェア trap を起こす範囲内にない場合には，その Null チェックは Explicit Nullcheck として生成しなければならない．幸運にも Java 言語ではこのようなケースはまれである．Java 言語では，すべての配列アクセスは配列境界チェックのために，配列の長さが先に必要となる．オブジェクトレイアウトによるが，配列の長さはオブジェクトからのオフセット 0 のところに入っている．そのため，配列アクセスは問題にはならない．インスタンス変数のアクセスについても，多くの場合は trap の範囲内にインスタンス変数に対するメモリアクセスは収まる．オブジェクトのポインタからのインスタンス変数のオフセットが最大となるケースを考えた場合，オフセットは約 512 KB（Java Virtual Machine Specification [13] によると $65534 * 8 = 524272$）となるので，システムによっては trap の範囲内にオブジェクトからのオフセットが収まらない場合がある．また，一般のオブジェクトのポインタは通常 0 番地付近にとられることはないが，その可能性がある場合は，アロケートされうる範囲をハードウェア trap を起こさない範囲として扱う．我々の Intel IA32 上の JIT コンパイラは Null チェックについて，Explicit Nullcheck と Implicit Nullcheck の両方の実装を行っている．

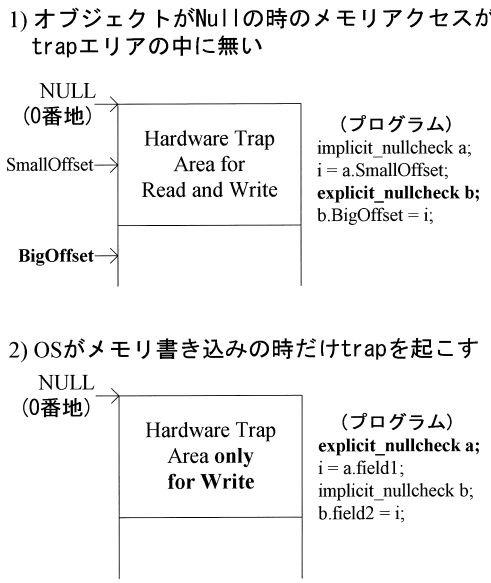


図 4 Explicit Nullcheck が必要な例
Fig. 4 Examples of explicit nullcheck required.

図 4 (2) は OS がメモリ書き込みだけについて，trap を起こす場合である．この場合には，メモリ読み込みについては Explicit Nullcheck を生成しなければならない．しかし，このような OS に対してはコンパイラがメモリ読み込みに対して speculation を適用できるという利点がある．もし，Null ポインタのときのメモリ読み込みがハードウェア trap を起こさないことが保証されているならば，そのメモリ読み込みは投機的にそれに関する Null チェックを超えて移動することができる．これは，メモリ読み込みを Null チェックを超えて移動させても，外部から超えたことによる影響が観測できないためである．speculation を行うことにより，メモリ読み込みを Null チェックを超えて，ループ不変の命令としてループの外に出すこともできる．

図 5 はそのような例である．図 5 (2) 内の「bound check index, len」は配列の境界チェックを行う中間コードで，$0 \leq \text{index} < \text{len}$ の条件を満たさないときに例外を起こす命令を意味する．また，arraylength b は配列 b の長さを返す命令を意味する．例外が起きる命令をメモリへの書き込み命令を超えて移動することによって，例外が起きた場合に副作用が起きる可能性がある．そのため，例外を起こす命令の移動の際に，メモリへの書き込み命令を超えることはできな

1) 元のプログラム　2) 中間表現　3) 最適化結果

```
do {                                    explicit_nullcheck a
   total += b[a.I++];       do {        Ti = a.I
} while(some cond);            nullcheck a      Tbl = arraylength b
                               T1 = a.I         do {
                               T2 = T1 + 1         T1 = Ti
                               nullcheck a         Ti = Ti + 1
                               a.I = T2 /* 例外命令   a.I = Ti
                               の移動に対する barrier */
                               nullcheck b         explicit_nullcheck b
                               T3 = arraylength b
                               boundcheck T1, T3   boundcheck T1, Tbl
                               T4 = b[T1]          T4 = b[T1]
                               total += T4         total += T4
                            } while(some cond)  } while(some cond)
```

図 5　speculation の例
Fig. 5　An example of speculation.

い．よって，図 5 (2) 内の nullcheck b は，メモリ書き込み $a.I = T2$ を超えて移動できない．次に scalar replacement による arraylength b の移動について考えると，b が Null のときに arraylength b が trap を起こすならば，nullcheck b を超えて移動することはできない．しかし，OS がメモリ読み込みに対して trap を起こさなければ，arraylength b は nullcheck b を超えて移動させることができる．この例では図 5 (3) に示すように arraylength b をループの外に移動することができる．我々の AIX 上の JIT コンパイラは Explicit Nullcheck の実装に conditional trap 命令（指定した条件を満たすと trap を起こす）を使っている．また，scalar replacement の際にメモリ読み込みに対して speculation を適用し，Null チェックを超えて移動可能として扱っている．conditional trap 命令は trap を起こさないときは 1 サイクルで実行可能であり，実行コストが小さいため，speculation を適用したほうが効果が大きい場合も多い．

もう 1 つ別な例をあげると，devirtualization によるメソッドインライニングを適用して，バーチャルメソッドをインラインするときには，**図 6** (2) に示すように Explicit Nullcheck を生成する必要がある．仮に，図 6 (2) 内の explicit_nullcheck を implicit_nullcheck で置き換えるとすると，obj が null ポインタかつ a が負の値だった場合には，obj の内容がアクセスされないため，例外が発生せずに実行が継続してしまう．これは Java の言語仕様に違反する．メソッドインライニングにともなう Explicit Nullcheck は普通の Java プログラム内に頻繁に現れるため，Explicit Nullcheck に対する最適化は効果が大きい．obj が Null のときに，obj.field のアクセスがハードウェア trap を起こす場合は，我々のアルゴリズムを適用すると図 6 (3)

1) インライン前　2) インライン後　3) 最適化結果

```
result = obj.func(a);      explicit_nullcheck obj;   if (a < 0){
                           if (a < 0){                  result = a;
int func(int s1) {            result = a;               explicit_nullcheck obj;
   if (s1 < 0){            } else {                  } else {
      return s1;              result = obj.field;        implicit_nullcheck obj;
   } else {                }                             result = obj.field;
      return this.field;                                 }
   }
}
```

(*Italic* は実際に例外が起きる場所を示す)

図 6　メソッドインライニングにともなう Explicit Nullcheck
Fig. 6　Explicit Nullcheck with method inlining.

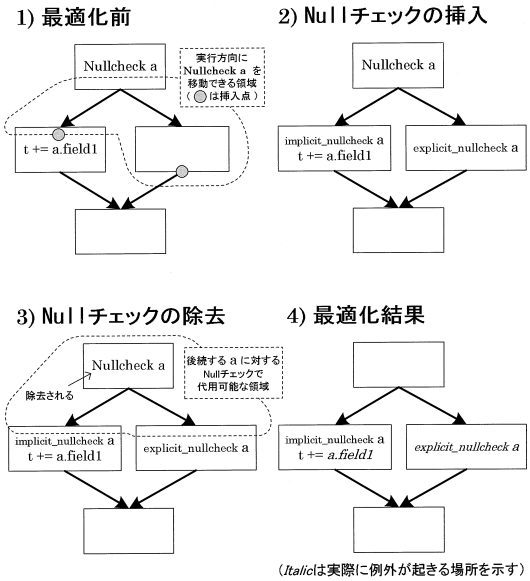


図 7　アーキテクチャ依存の最適化
Fig. 7　Architecture dependent optimization.

に示すように，Null チェックの実行コストを最小限にすることができる．

3.3.2　アーキテクチャ依存の最適化の概要

図 7 を使ってこの最適化の説明を行う．図 7 (1) 内の Null チェックは，Implicit Nullcheck として実装すると，右側のパスではオブジェクト a にアクセスされないために，このままでは Explicit Nullcheck として実装を行わなければならない．我々はハードウェア trap を利用し，Null チェックの実行コストを最小限にするために，**逆向きに** PRE を利用している．この最適化は最初に，メソッド内の Null チェックに対して実行方向に移動できる領域を求める．このとき移動の障害となる命令としては，Null チェックの対象変数の書き込み，他の種類の例外，メモリへの書き込み，そしてオブジェクトが Null のときに Null チェックの代用となるハードウェア trap を起こすメモリアクセスがある．

次に Null チェックの挿入点を求める．あるベーシックブロック B の最初の点が移動可能な領域内にあり，B の直後のベーシックブロックの最初の点が領域外のときは，ベーシックブロック B 内の領域の境界点を挿入点とする．例では図 7 (1) の領域，および挿入点が求まる．次に，Null チェックの挿入を行うが，挿入する Null チェックは挿入点の次の命令によって 2 種類ある．挿入点の次の命令が，オブジェクトが Null のときに Null チェックの代用となるハードウェア trap を起こすメモリアクセスのときは，Implicit Nullcheck を挿入し，それ以外のときは Explicit Nullcheck を挿入する．例では a が Null のときに a.field1 が trap を起こすとすると，図 7 (2) のように挿入される．Implicit Nullcheck の次の命令は実際に例外が起きる場所なので，この命令をコンパイラ内で例外発生点として記録しなければならない．これはマシン命令レベルの最適化（たとえばコードスケジューリング）が例外発生点を超えて，間違ったコード移動を行うのを防ぐためである．次に後続する Null チェックによって，Null チェックを代用できる領域を求め，その領域内の Null チェックを除去する．例では図 7 (3) のように領域が求まり，1 番上の Nullcheck a が除去される．最終的には図 7 (4) となり，図 7 (1) と比べると，左側のパスについて実行コストが減っている．

他の例として，図 3 (6) に対してこの最適化を適用した場合には，3.3.1 項の Implicit Nullcheck の条件を満たしていれば，すべての Null チェックは Implicit Nullcheck に置き換えられる．

4. 我々のアルゴリズム

この章では Null チェックの最適化のアルゴリズムについて述べる．4.1 節ではアーキテクチャと独立した最適化のための 2 つの変形方法について述べる．4.2 節ではアーキテクチャ依存の最適化のための 2 つの変形方法について述べる．なお，以下のアルゴリズム内で「他の種類の例外を起こす命令」とは，NullPointerException 以外の例外を起こす可能性があるすべての命令を示す．また，「メモリ書き込みを行う命令」とは，グローバルなエリアへのメモリ書き込みを行う可能性があるすべての命令を示す．また，Gen(n)，Kill(n)，In(n)，Out(n)，InnerBB の各集合については各項で再定義を行っている．

4.1 アーキテクチャと独立した最適化

4.1.1 項では Null チェックの挿入点を求め，4.1.2 項では Null チェックの除去および挿入の処理を行う．

4.1.1 Null チェックの挿入点を求めるためのアルゴリズム

ここでの目的は，Null チェックが実行とは逆方向に移動できる範囲の最初の点を求めることである．Out(n) はメソッド内に含まれる Null チェックを実行とは逆方向に移動させた場合に，ベーシックブロック n の最後へ移動可能な Null チェックの集合である．この集合は次の後方データフロー解析（backward data-flow analysis）を使うことによって求められる．

$$\mathrm{Out(n)} = \bigcap_{\mathrm{m=Succ(n)}} (((\mathrm{Out(m)} - \mathrm{Kill(m)}) \cup \mathrm{Gen(m)}) - \mathrm{Try(m,n)})$$

ここで Gen，Kill，Try は次のような集合を意味する．

Gen(n)：ベーシックブロック n 内の Null チェックについて，ベーシックブロック n の先頭に移動できる Null チェックの集合．

Kill(n)：ベーシックブロック n を超えて，実行とは逆方向に移動できない Null チェックの集合．ベーシックブロック n 内のすべての命令に対して，次のアルゴリズムによりこの集合 Kill(n) を求める．

```
Kill(n) = φ
for (n 内のすべての命令 I について){
  if (I は他の種類の例外を起こす ||
      I はメモリ書き込みを行う ||
      (n は try 領域内 &&
       I はローカル変数の書き込みを行う)){
    Kill(n) = すべての Null チェック
    break;
  } else if (I は変数への書き込みを行う){
    C = その変数に対する Null チェック
    Kill(n) = Kill(n) ∪ C
  }
}
```

Try(m,n)：ベーシックブロック m からベーシックブロック n のエッジを移動できない Null チェックの集合．ベーシックブロック m とベーシックブロック n が同じ try 領域内にない場合にはすべての Null チェックがこの集合に含まれる．

次に，Null チェックが実行とは逆方向に移動できる範囲の境界が，ベーシックブロック n となる Null チェックの集合 Earliest(n) を次の式を使って求める．

$$\mathrm{Earlist(n)} = \left(\bigcup_{\mathrm{m=Pred(n)}} \overline{\mathrm{Out(m)}} \right) \cap \mathrm{Out(n)}$$

Earliest(n) はベーシックブロック n の最後に挿入する Null チェックの集合となる．この挿入により，部分的

冗長性を持つ Null チェックやループ内の Null チェックが除去できるようになる．次の最適化が Null チェックの挿入点を除去するため，ここの処理ではまだ Null チェックの挿入は行わず，次の最適化の最後で挿入する．

4.1.2 Null チェックの除去および挿入アルゴリズム

ここでの目的は，対象オブジェクトが Null ではないと判断できる Null チェックを除去し，最後に Null チェックの挿入処理を行うことである．In(n)，Out(n) はそれぞれベーシックブロック n の先頭，最後で，対象オブジェクトが Null ではないと判断できる Null チェックの集合である．この集合は次の前方データフロー解析を使うことによって求められる．

$$\mathrm{In(n)} = \bigcap_{\mathrm{m=Pred(n)}} (\ \mathrm{Out(m)} \cup \mathrm{Earliest(m)} \cup \mathrm{Non\_null(m,n)}\ )$$

$$\mathrm{Out(n)} = (\mathrm{In(n)} - \mathrm{Kill(n)}) \cup \mathrm{Gen(n)}$$

ここで Gen，Earliest，Kill，Non_null は次のような集合を意味する．

- Gen(n)：ベーシックブロック n 内の Null チェックやオブジェクトの内容にアクセスする命令や *new* などの命令によって，ベーシックブロック n の最後で Null ではないと判断できる Null チェックの集合．
- Earliest(n)：4.1.1 項で求めた，ベーシックブロック n の最後に挿入する Null チェックの集合．
- Kill(n)：ベーシックブロック n を超えることによって，Null ではないと判断できなくなる Null チェックの集合．n 内の変数への書き込みに対する Null チェックがこの集合に含まれる．
- Non_null(m, n)：元の Java プログラム内に存在する条件分岐や this オブジェクトなどから，ベーシックブロック m からベーシックブロック n のエッジ上で Null ではないと判断できる Null チェックの集合．具体的には，次の Null チェックが，この集合に含まれる．
  - *ifnull*，*ifnonnull*，*instanceof* の結果による *if* などの条件分岐により，m から n のエッジでは Null ではないと判断できる Null チェック
  - バーチャルメソッド内の this オブジェクトに対する Null チェック

まず，次のアルゴリズムを使って，ベーシックブロック内の Null チェックを除去する．ベーシックブロック n 内の各命令について，Null ではないと判断できる Null チェックの集合を In(n) より求め，Null ではないと判断できる Null チェックを除去する．

```
InnerBB = In(n)
for (n 内の初めから終わりまでの命令 I について){
  if (I は Null チェック){
    if (I ∈ InnerBB){
      I を除去
    } else {
      InnerBB = InnerBB ∪ I
    }
  } else if (I はオブジェクトの内容をアクセスする){
    C = I のための Null チェック
    InnerBB = InnerBB ∪ C
  } else if (I は変数への書き込みを行う){
    C = 変数に対する Null チェック
    if (new などにより書き込み元が
        Null ではないと判断できる){
      InnerBB = InnerBB ∪ C
    } else {
      InnerBB = InnerBB - C
    }
  }
}
```

次に，Null チェックを挿入する．Earliest(n) 内の冗長な挿入点を次の式を使って除去し，除去後の Earliest(n) を基にベーシックブロック n の最後に Null チェックを挿入する．

$$\mathrm{Earliest(n)} = \mathrm{Earliest(n)} - \mathrm{Out(n)}$$

4.2 アーキテクチャ依存の最適化

まず，この最適化の入力コード内の Null チェックはすべて Explicit Nullcheck として扱う．4.2.1 項では Explicit Nullcheck または Implicit Nullcheck の挿入処理を行い，4.2.2 項では Explicit Nullcheck の除去処理を行う．以下のアルゴリズム内に書かれている「Null のときに trap を起こす命令」は 3.3.1 項で述べた Implicit Nullcheck の条件を満たす，オブジェクトの内容にアクセスしオブジェクトが Null のときに trap を起こす命令を意味する．この命令は，実装するアーキテクチャによって異なるが，このような命令が存在しないアーキテクチャの場合には，本節の最適化は行わないものとする．

4.2.1 Null チェックの挿入アルゴリズム

ここでの目的は，Null チェックが実行方向に移動できる領域の最後の点を求めることである．In(n) はメソッド内に含まれる Null チェックを実行方向に移動させた場合に，ベーシックブロック n の先頭へ移動可能な Null チェックの集合である．この集合は次の前方

データフロー解析を使うことによって求められる．

$$In(n) = \bigcap_{m=Pred(n)} (((In(m) - Kill(m)) \cup Gen(m)) - Try(m, n))$$

ここで Gen と Kill は次のような集合を意味する．

Gen(n)：ベーシックブロック n 内の Null チェックについて，ベーシックブロック n の最後に移動できる Null チェックの集合．

Kill(n)：ベーシックブロック n を実行方向に超えることができない Null チェックの集合．ベーシックブロック n 内のすべての命令に対して，次のアルゴリズムによりこの集合 Kill(n) を求める．

```
Kill(n) = φ
for (n 内のすべての命令 I について){
  if (I は他の種類の例外を起こす ||
      I はメモリ書き込みを行う ||
      (n は try 領域内 &&
       I はローカル変数の書き込みを行う)){
    Kill(n) = すべての Null チェック
    break;
  } else if (I は変数への書き込みを行う){
    C = 変数に対する Null チェック
    Kill(n) = Kill(n) ∪ C
  } else if (I は Null のときに trap を起こす){
    C = I のための Null チェック
    Kill(n) = Kill(n) ∪ C
  }
}
```

次に，Null チェックが実行方向に移動できる範囲の境界がベーシックブロック n となる Null チェックの集合 Latest(n) を次の式を使って求める．

$$Latest(n) = \left(\bigcup_{m=Succ(n)} \overline{In(m)} \right) \cap In(n)$$

Latest(n) はベーシックブロックの先頭に挿入可能な Null チェックの集合である．ベーシックブロック n 内の挿入処理は，Latest(n) を基に次のアルゴリズムによって行われる．この処理によって，Explicit Nullcheck または Implicit Nullcheck が挿入される．

```
InnerBB = Latest(n)
for (n 内の初めから終わりまでの命令 I について){
  if (I が Null チェック){
    InnerBB = InnerBB ∪ I
  } else {
    if (I は Null のときに trap を起こす){
      C = I のための Null チェック
      if (C ∈ InnerBB) {
        I の直前に Implicit Nullcheck C を挿入
        I を例外発生点としてコンパイラ内で記録
        InnerBB = InnerBB - C
      }
    }
    if (I が他の種類の例外を起こす ||
        I がメモリ書き込みを行う ||
        (n は try 領域内 &&
         I はローカル変数の書き込みを行う)){
      for (each C ∈ InnerBB) {
        I の直前に Explicit Nullcheck C を挿入
      }
      InnerBB = φ
    } else if (I が変数への書き込みを行う){
      C = 変数のための Null チェック
      if (C ∈ InnerBB) {
        I の直前に Explicit Nullcheck C を挿入
        InnerBB = InnerBB - C
      }
    }
  }
}
for (each C ∈ InnerBB) {
  n の最後に Explicit Nullcheck C を挿入
}
```

4.2.2 Explicit Nullcheck の除去アルゴリズム

ここでの目的は，後続する Explicit Nullcheck または Implicit Nullcheck で代用できる Explicit Nullcheck を除去することである．このような性質を持った Nullcheck を以下「代用可能」と呼ぶことにする．Out(n) はメソッド内に存在する Null チェックによって，ベーシックブロック n の最後の点で代用可能と判断できる Null チェックの集合である．この集合は次の後方データフロー解析を使うことによって求められる．

$$Out(n) = \bigcap_{m=Succ(n)} (((Out(m) - Kill(m)) \cup Gen(m)) - Try(m, n))$$

ここで Gen と Kill は次のような集合を意味する．

Gen(n)：ベーシックブロック n 内の Null チェックや Null チェックの代用となる trap を起こす命令によって，ベーシックブロック n の先頭で代用可能と判断できる Null チェックの集合．

Kill(n)：4.2.1 項の Kill(n) と同じ．

次にベーシックブロック n 内の各命令について，次

表 1 jBYTEmark v.0.9 のパフォーマンス（大きい値ほど良い）
Table 1 Performance for jBYTEmark v.0.9 (Larger numbers are better).

(単位：index)	Numeric Sort	String Sort	BitField	FP Emul.	Fourier	Assign.	IDEA encrypt.	Huffman	Neural Net	LU Decomp.
本アルゴリズム	201.96	54.41	258.86	219.64	22.75	207.41	67.46	159.33	200.50	205.90
以前の手法	160.78	49.87	245.25	186.12	22.74	130.10	63.27	156.08	130.82	158.31
最適化無し，Implicit	157.01	49.58	245.13	170.18	22.74	125.31	63.14	151.88	130.42	119.91
最適化無し，Explicit	156.94	49.08	227.85	163.87	22.68	107.87	62.99	134.40	116.81	112.57
HotSpot	207.13	44.73	234.00	206.56	8.06	114.74	25.69	145.24	88.87	106.62

表 2 SPECjvm98 のパフォーマンス（小さい値ほど良い）
Table 2 Performance for SPECjvm98 (Smaller numbers are better).

(単位：秒)	mtrt	jess	compress	db	mpegaudio	jack	javac
本アルゴリズム	6.44	7.67	17.38	24.42	11.32	9.39	14.18
以前の手法	7.05	7.86	17.49	24.70	11.33	9.77	14.30
最適化無し，Implicit	7.09	7.95	17.55	24.71	11.39	9.80	14.33
最適化無し，Explicit	7.38	8.25	18.70	25.33	12.00	10.02	15.17
HotSpot	5.73	6.53	20.13	24.61	14.78	9.25	17.50

本アルゴリズム: 本アルゴリズムを適用し，ハードウェア trap を利用している．
以前の手法: Whaley のアルゴリズムを適用し，ハードウェア trap を利用している．
最適化無し,Implicit: Null チェックの最適化を無効にするが，ハードウェア trap は利用する．
最適化無し,Explicit: Null チェックの最適化を無効にし，ハードウェア trap も利用しない．
HotSpot: HotSpot Server VM 2.0 beta

のアルゴリズムを使って代用可能な Null チェックの集合を Out(n) より求め，n 内の代用可能な Explicit Nullcheck を除去する．

```
InnerBB = Out(n)
for (n 内の終わりから初めまでの命令 I について){
  if (I が Null チェック){
    if (I ∈ InnerBB){
      /* I は必ず Explicit Nullcheck となる */
      I を除去
    } else {
      InnerBB = InnerBB ∪ I
    }
  } else if (I は他の種類の例外を起こす ||
             I はメモリ書き込みを行う ||
             (n は try 領域内 &&
              I はローカル変数の書き込みを行う)){
    InnerBB = φ
  } else if (I は変数への書き込みを行う){
    C = 変数に対する Null チェック
    InnerBB = InnerBB - C
  } else if (I は Null のときに trap を起こす){
    C = その命令に対する Null チェック
    InnerBB = InnerBB - C
  }
}
```

5. 実験結果

我々は jBYTEmark version 0.9 と SPECjvm98[14] の 2 つのベンチマークを測定し，本アルゴリズムの評価を行った．SPECjvm98 の測定方法は test mode（SPEC-compliant mode ではない），count は 100 で行った．5.1 節から 5.3 節までの実験結果は IBM IntelliStation M Pro（Pentium III 600 MHz，メモリ 384 MB），Windows NT 4.0 Service Pack 5，IBM Developer Kit for Windows，Java Technology Edition バージョン 1.2.2 を使って測定した．5.4 節の実験結果は PowerPC 604e 332 MHz，メモリ 128 MB，AIX4.3.1 を使って測定した．

5.1 本アルゴリズムによるパフォーマンスの向上

我々は本アルゴリズム（**表 1**，**表 2** の「本アルゴリズム」）と比較するために，ベースラインとしてすべての Null チェックの最適化を無効にし，すべての Null チェックを Explicit Nullcheck として生成するようにした版を作成した（表 1，表 2 の「最適化なし，Explicit」）．また，以前の手法と比較するために，Whaley のアルゴリズム[12]を実装した（表 1，表 2 の「以前の手法」）．これは我々の以前の JIT[2),5)]で行っていた手法である．Implicit Nullcheck の効果を比較するために，すべての Null チェックの最適化を無効にし，ハードウェア trap を利用するような実装を行った

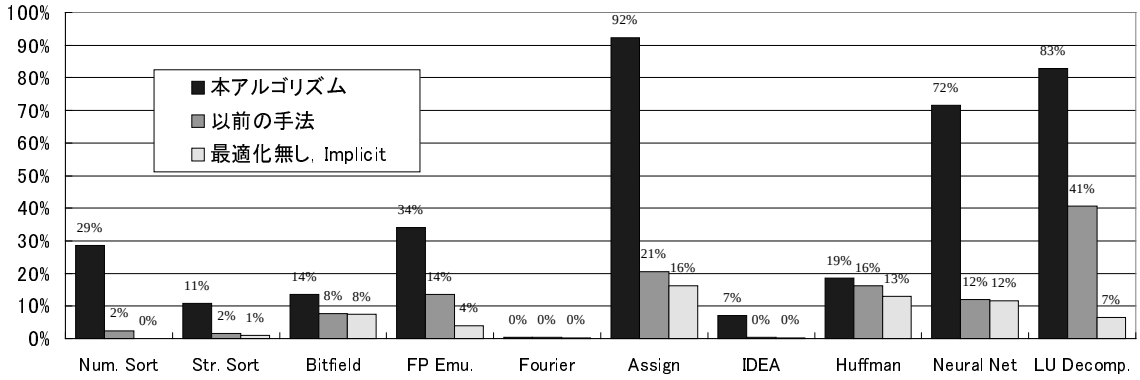


図 8 jBYTEmark バージョン 0.9 のパフォーマンス向上割合
Fig. 8 Improvement for jBYTEmark v.0.9.

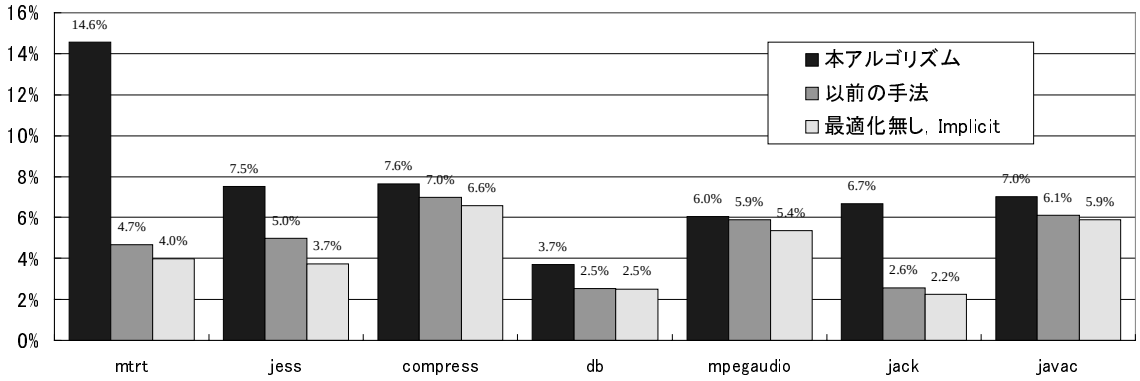


図 9 SPECjvm98 のパフォーマンス向上割合
Fig. 9 Improvement for SPECjvm98.

（表 1，表 2 の「最適化なし，Implicit」）．他の Java 言語処理系と比較するために，HotSpot Server VM 2.0 beta [15] を使って同じベンチマーク，同じ条件で測定を行った（表 1，表 2 の「HotSpot」）．

図 8 は表 1 に対応し，jBYTEmark について我々のベースライン（最適化なし，Explicit）からのパフォーマンス向上の割合を示したものである．調査の結果，アーキテクチャと独立した最適化が，Assignment，Neural Net，LU Decomposition のベンチマークについて非常に効果が大きいことが分かった．これらのベンチマークでは多次元配列を使っており，Null チェック最適化，配列境界チェック最適化，Scalar Replacement の 3 つがそれぞれ助け合い，いくつかのループ不変な配列アクセスがループの外に移動し，大きくパフォーマンスが向上した．

図 9 は表 2 に対応し，SPECjvm98 について我々のベースライン（最適化なし，Explicit）からのパフォーマンス向上の割合を示す．調査の結果，アーキテクチャ依存の最適化が，mtrt のベンチマークについて非常に効果が大きいことが分かった．このベンチマークでは，クラス内のデータをアクセスするための小さなメソッドがあり，それらのメソッドが多くの場所で用いられていた．これらのメソッドをインラインする際に，関連する Explicit Nullcheck が多く生成され，アーキテクチャ依存の最適化によって Implicit Nullcheck に変形され，パフォーマンスが向上した．

5.2 HotSpot と比較したパフォーマンス

HotSpot は Server 版と Client 版の 2 種類のバージョンを提供し，Server 版はコンパイルタイムを犠牲にし，強力な最適化を行っている．Client 版は逆にコンパイルタイムの削減を重要視し，最適化のレベルが低い．我々の JIT は特にバージョンを分けていないため，強力な最適化を行っている HotSpot Server 版を比較対象とし，我々の JIT の性能を評価した．

図 10 は表 1 に対応し，jBYTEmark について本アルゴリズムを適用した我々の JIT と HotSpot Server のパフォーマンスの比較を示す．我々の JIT がほとんどのベンチマークで良い結果を示し，平均すると HotSpot よりも 1.7 倍ほど良い結果が出ている．

図 11 は表 2 に対応し，SPECjvm98 について本

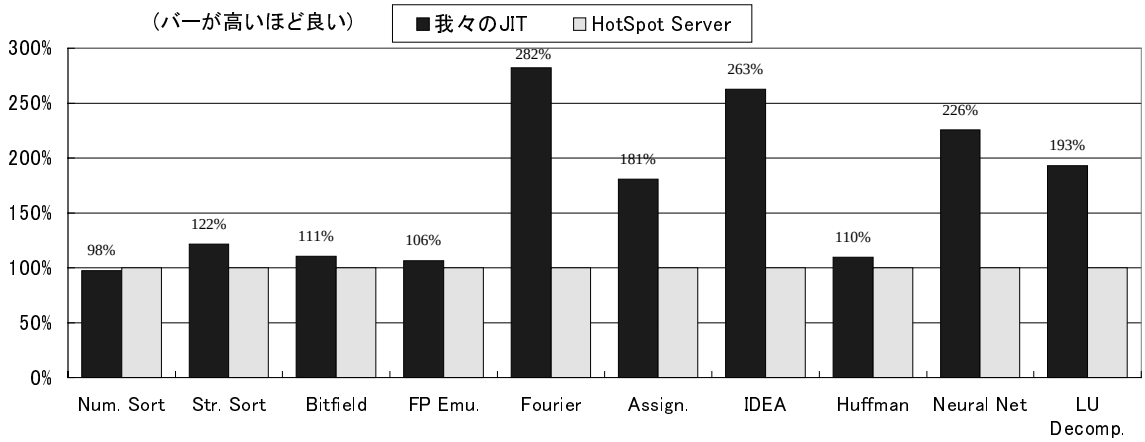


図 10 HotSpot Server 版と比較した jBYTEmark バージョン 0.9 のパフォーマンス（HotSpot = 100%）
Fig. 10 Performance Comparison for jBYTEmark (HotSpot = 100%).

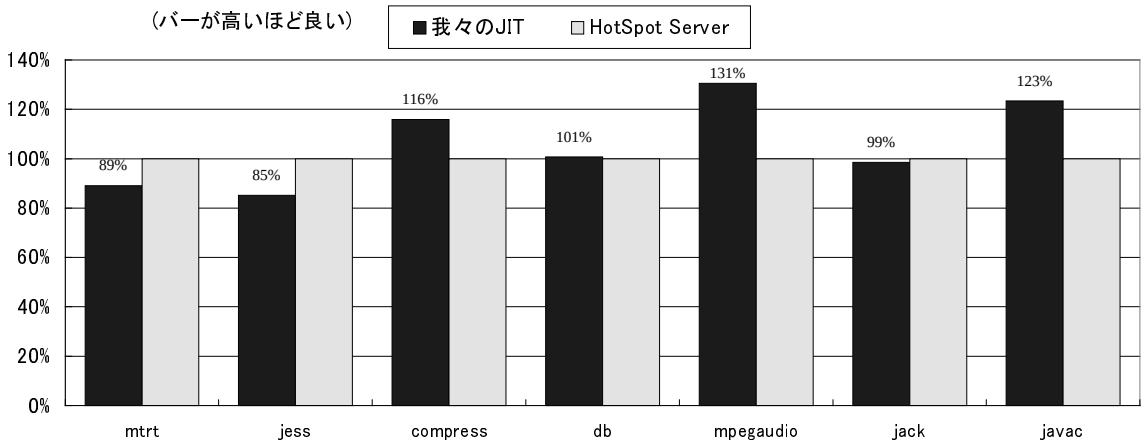


図 11 HotSpot Server 版と比較した SPECjvm98 のパフォーマンス（HotSpot = 100%）
Fig. 11 Performance Comparison for SPECjvm98 (HotSpot = 100%).

表 3 JIT のコンパイル時間
Table 3 JIT compilation time.

（単位：秒）		mtrt	jess	compress	db	mpegaudio	jack	javac
我々の JIT	最初の実行時間	9.47	10.37	17.43	24.62	12.56	11.95	22.33
	最速の実行時間	6.44	7.67	17.38	24.42	11.32	9.39	14.18
	コンパイル時間	3.03	2.70	0.05	0.20	1.24	2.56	8.15
	コンパイル時間の割合	32.00%	26.04%	0.29%	0.81%	9.87%	21.42%	36.50%
HotSpot Server	最初の実行時間	11.50	18.06	20.75	26.80	19.23	21.88	57.38
	最速の実行時間	5.73	6.53	20.13	24.61	14.78	9.25	17.50
	コンパイル時間	5.77	11.53	0.62	2.19	4.45	12.63	39.88
	コンパイル時間の割合	50.17%	63.84%	2.99%	8.17%	23.14%	57.22%	69.50%

アルゴリズムを適用した我々の JIT と HotSpot のパフォーマンスの比較を示す．我々の JIT がわずかに良い結果を示し，平均すると HotSpot よりも 6%ほど良い結果が出ている．

5.3 JIT のコンパイル時間

本節では，以前の手法と比べた我々の手法のコンパイル時間の増加について比較する．jBYTEmark についてはコンパイル時間が非常に短いために，測定することができなかった．そこで，我々は最速の実行時間にはコンパイル時間が含まれないと仮定し，SPECjvm98 の最初の実行時間と最速の実行時間の差が，実質的なコンパイル時間であると仮定した．**表 3** に我々の JIT

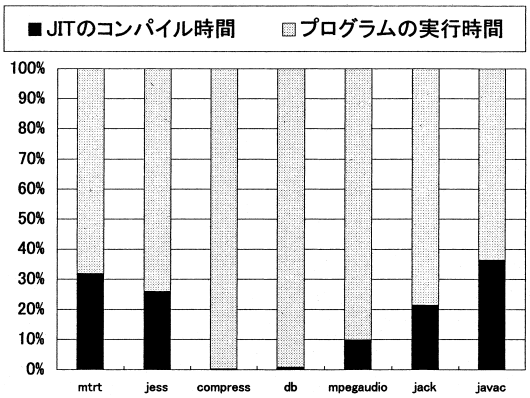


図 12 我々の JIT 上でコンパイル時間の占める割合（100% = 最初の実行時間）
Fig. 12 Percentage of our JIT compilation time (100% = time spent for the first run).

コンパイラと HotSpot Server で測定した最初の実行時間，最速の実行時間，仮定したコンパイル時間，およびそれぞれの最初の実行時間を 100%としたときのコンパイル時間の割合を示す．我々のコンパイル時間に関する仮定が正しいとすると，我々の JIT コンパイラは HotSpot Server に比べて，非常に速いコンパイル処理を行っていることが分かる．

図 12 は表 3 に対応し，最初の実行時間に対する我々の JIT のコンパイル時間の占める割合を示す．javac は一番 JIT のコンパイル時間の占める割合が高いベンチマークであることが分かる．

さらに我々は，AIX 上のトレースツールを用いて JIT のコンパイル時間の内訳を測定し，プラットフォームの違いを考慮に入れてコンパイル時間を計算した．**表 4**，**図 13** にその結果を示す．表 4，図 13 内に書かれている比率は，すべて本アルゴリズムを使った最初の実行時間を 100%として計算している．また，図 13 に関しては 90%以下の部分は省略してある．compress，db，mpegaudio のベンチマークについては，コンパイル時間が短かすぎるために内訳は測定できなかった．「Null チェックの最適化」に費やされた時間は，以前の手法に比べて本アルゴリズムのほうが約 3 倍コンパイル時間がかかっている．また，「その他」にかかる時間も若干増えている．これは，本アルゴリズムによる Null チェックの最適化が，scalar replacement などの他の最適化の機会を増やしているためと考えられる．

表 5 に以前の手法と比べて，本アルゴリズムを適用することによって増えたコンパイル時間を示す．本手法により，平均して約 2.3%程度全体のコンパイル時間が増加することが分かった．

表 4 JIT のコンパイル時間の内訳
Table 4 Breakdown of our JIT compilation times.

		Null チェックの最適化	その他
mtrt	本アルゴリズム	0.07(2.31%)	2.96(97.69%)
	以前の手法	0.02(0.66%)	2.93(96.70%)
jess	本アルゴリズム	0.06(2.22%)	2.64(97.78%)
	以前の手法	0.02(0.74%)	2.62(97.04%)
jack	本アルゴリズム	0.06(2.34%)	2.50(97.66%)
	以前の手法	0.02(0.78%)	2.49(97.27%)
javac	本アルゴリズム	0.17(2.09%)	7.98(97.91%)
	以前の手法	0.06(0.74%)	7.86(96.44%)

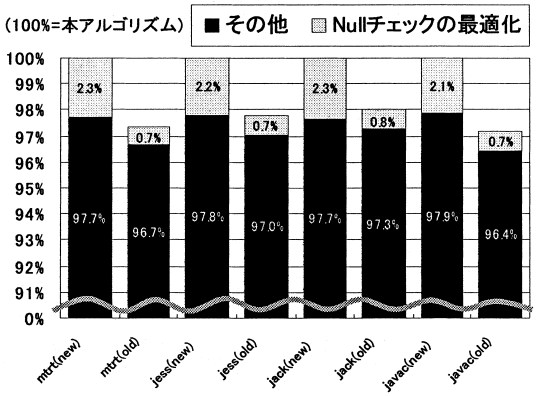


図 13 JIT のコンパイル時間の内訳（new = 本アルゴリズム，old = 以前の手法）
Fig. 13 Breakdown of JIT compilation times (new = our approach, old = previous approach).

表 5 本アルゴリズムによる JIT コンパイル時間の増加
Table 5 Increases in JIT compilation time in our approach.

	コンパイル時間の増加（秒）	コンパイル時間の増加（%）
mtrt	0.07	2.31%
jess	0.06	2.22%
jack	0.05	1.95%
javac	0.23	2.82%

5.4 Speculation と Implicit Nullcheck の効果の比較

3.3.1 項で述べたように，我々の AIX（PowerPC）上の JIT コンパイラでは Null チェックを検出するために，conditional trap 命令を使っている．また，scalar replacement の際に，メモリ読み込みに対しては speculation を使っている．speculation の効果（**表 6**，**表 7** の「Speculation」）を調べるために，scalar replacement 内の speculation を無効にして測定した（表 6，表 7 の「No Speculation」）．また，Implicit Nullcheck

表 6 AIX 上で測定した jBYTEmark v.0.9 のパフォーマンス（大きい値ほど良い）
Table 6 Performance for jBYTEmark v.0.9 on AIX (Larger numbers are better).

(単位：index)	Numeric Sort	String Sort	BitField	FP Emul.	Fourier	Assign.	IDEA encrypt.	Huffman	Neural Net	LU Decomp.
Implicit Nullcheck	183.28	29.91	84.40	86.62	13.25	95.66	45.60	100.74	77.35	92.66
Speculation	186.12	30.01	84.45	87.46	13.26	96.47	45.14	97.35	86.03	92.08
No Speculation	181.09	29.77	83.65	86.16	13.25	94.76	45.14	97.20	75.94	91.66
最適化無し，Explicit	173.92	28.17	83.42	79.89	13.23	81.71	44.68	97.14	73.93	79.98

表 7 AIX 上で測定した SPECjvm98 のパフォーマンス（小さい値ほど良い）
Table 7 Performance for SPECjvm98 on AIX (Smaller numbers are better).

(単位：秒)	mtrt	jess	compress	db	mpegaudio	jack	javac
Implicit Nullcheck	19.94	26.09	43.75	71.86	19.87	44.71	46.90
Speculation	20.34	25.92	43.80	72.08	20.16	44.56	47.14
No Speculation	20.56	26.28	44.21	72.39	20.33	44.66	47.26
最適化無し，Explicit	21.00	26.28	44.25	72.85	20.42	45.36	47.34

Implicit Nullcheck: Intel 上に適用している 4.2 節の「アーキテクチャ依存の最適化」を AIX 上で適用した．
Speculation: Speculation を有効にした我々の現状の JIT コンパイラ．Null チェックには conditional trap 命令を使っている．
No Speculation: 我々の JIT コンパイラから Speculation だけを無効にした．
最適化無し,Explicit: Null チェックの最適化を無効にし，ハードウェア trap も利用しない．

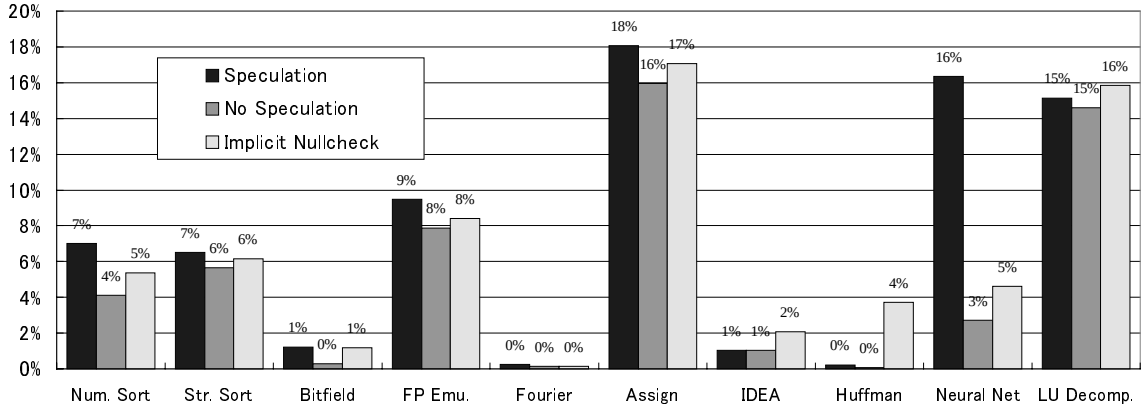


図 14 AIX 上での jBYTEmark バージョン 0.9 のパフォーマンス向上割合
Fig. 14 Improvement for jBYTEmark v.0.9 on AIX.

の効果を調べるために，Intel 上で行っている 4.2 節の「アーキテクチャ依存の最適化」を PowerPC 上で適用し測定した（表 6，表 7 の「Implicit Nullcheck」）．このモジュールは，NullPointerException が正しく検出されないために，Java の言語仕様に違反している．我々は単に実験のためにこのようなモジュールを作成し，測定した．パフォーマンスの向上を見るために，ベースラインとして Null チェックの最適化をすべて無効にして測定した（表 6，表 7 の「最適化なし，Explicit」）．

図 14 は表 6 に対応し，jBYTEmark について我々のベースライン（最適化なし，Explicit）からのパフォーマンス向上の割合を示す．Neural Net のベンチマークについて，Speculation の効果が大きいことが分かる．これは，Speculation によって 4 命令が Null チェックを超えて最内ループから外に移動したためと分かった．Implicit Nullcheck について，Intel 上でのパフォーマンス向上割合（図 8 の「本アルゴリズム」）と比べると，Neural Net だけが極端に効果が少ない．これは，Null チェックの最適化への入力コードが Intel 上と PowerPC 上で異なるためということが分かった．Intel 上では，メソッド java.lang.Math.exp はインラインされているが，PowerPC 上ではインラインされていない．我々の Intel 上の JIT コンパイラは，このメソッドを exponential 命令に変換している．しかし，PowerPC にはそのような命令がないために，我々の

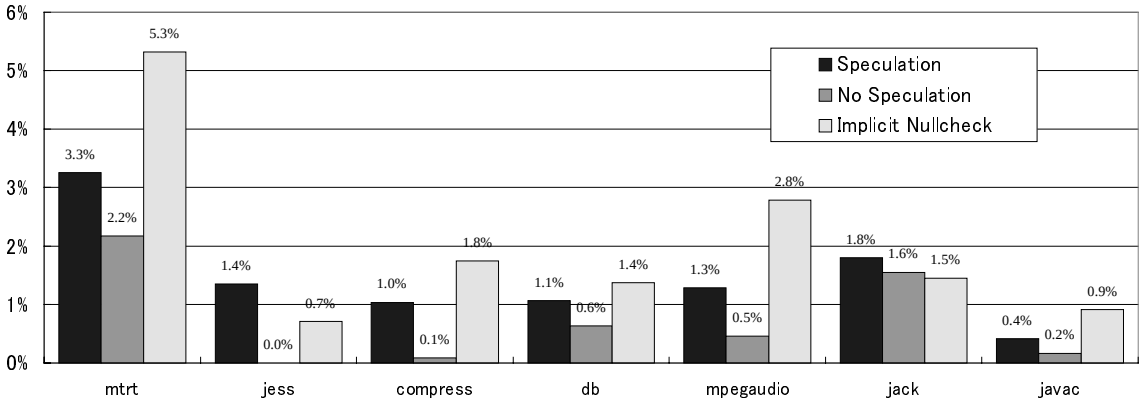


図 15 AIX 上での SPECjvm98 のパフォーマンス向上割合
Fig. 15 Improvement for SPECjvm98 on AIX.

PowerPC 上の JIT コンパイラでは，そのような変換は行っていなかった．そのため，このメソッド呼び出しが scalar replacement を行う際の障害となり，PowerPC 上ではこのベンチマークプログラムが極端に効果が少なかった．

図 15 は表 7 に対応し，SPECjvm98 について我々のベースライン（最適化なし，Explicit）からのパフォーマンス向上の割合を示す．Implicit Nullcheck が mtrt について，特に効果が大きかったことが分かる．これは，Intel 上でも同じ傾向だが，効果は PowerPC 上のほうが少ない．これは，PowerPC 上では Null チェックに conditional trap 命令を使っており，Null チェックの実行コストが Intel 上よりも小さいためと考えられる．

6. 終わりに

本稿では，Null ポインタチェックの除去についての新しいアルゴリズムを述べた．「アーキテクチャと独立した最適化」は Null チェックを実行とは逆方向に移動させ，冗長な Null チェックを除去する．この最適化は，他の最適化と協調して実行させることにより，お互いの最適化の効果を最大限にすることができる．「アーキテクチャ依存の最適化」では Null チェックをハードウェア trap に変換する．この最適化は，Null チェックの実行コストを最小限にする．これらの最適化を適用した結果，従来の手法と比べて大きなパフォーマンスの向上が得られた．我々のアルゴリズムは，Java 言語だけではなく，Null チェックを必要とするほかの言語に対しても適用できる．我々は，本稿で述べた手法の重要性が，将来高まることを期待している．

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 IBM 東京基礎研究所 JIT compiler グループ，IBM トロント研究所 Java JIT Development グループの皆様に深く感謝します．

参 考 文 献

1) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley, Reading, Massachusetts (1996).
2) 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).
3) Alpern, B., Attanasio, C.R., Barton, J.J., Burke, M.G., Cheng, P., Choi, J.D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M.F., Ngo, T., Russel, J.R., Sarkar, V., Serrano, M.J., Shepherd, J.C., Smith, S.E., Sreedhar, V.C., Srinivasan, H. and Whaley, J.: The Jalapeño virtual machine, *IBM Systems Journal*, Vol.39, No.1, pp.211–238 (2000).
4) Yang, B.S., Moon, S.M., Park, S., Lee, J., Lee, S., Park, J., Chung, Y.C., Kim, S., Ebcioglu, K. and Altman, E.: LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation, *Conference on Parallel Architectures and Compilation Techniques* (Oct. 1999).
5) Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H. and Nakatani, T.: Optimizations to reduce overheads of the Java language in a Just-in-Time Java compiler, *Proc. ACM SIGPLAN Java Grande Conference* (June 1999).
6) Dean, J., Grove, D. and Chambers, C.: Optimization of object-oriented programs using

static class hierarchy, *Proc. 9th European Conference on Object-Oriented Programming – ECOOP '95*, pp.77–101 (1995).
7) Aigner, G. and Holzle, U.: Eliminating Virtual Function Calls in C++ Programs, *Proc. 10th European Conference on Object-Oriented Programming – ECOOP '96*, pp.142–166 (1996).
8) Muchnick, S.S.: *Advanced compiler design and implementation*, Morgan Kaufmann, San Francisco, California (1997).
9) Morel, E. and Renvoise, C.: Global optimization by suppression of partial redundancies, *CACM*, Vol.22, No.2, pp.96–103 (1979).
10) Knoop, J., Rüthing, O. and Steffen, B.: Lazy code motion, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.224–234 (June 1992).
11) Knoop, J., Rüthing, O. and Steffen, B.: Optimal code motion: Theory and practice, *ACM Trans. Prog. Lang. Syst.*, Vol.17, No.5, pp.777–802 (1995).
12) Whaley, J.: Dynamic optimization through the use of automatic runtime specialization, Master's Thesis, Massachusetts Institute of Technology (May 1999).
13) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, Addison-Wesley, Reading, Massachusetts (1996).
14) Standard Performance Evaluation Corp.: SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/
15) HotSpot の homepage: http://java.sun.com/products/hotspot/

(平成 12 年 5 月 26 日受付)
(平成 12 年 9 月 8 日採録)

川人 基弘（正会員）
1968 年生．1991 年早稲田大学理工学部電子通信学科卒業．同年日本 IBM（株）に入社．現在，同社東京基礎研究所に所属．コンパイラの研究に従事．

小松 秀昭（正会員）
1960 年生．1985 年早稲田大学大学院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）．

中谷登志男（正会員）
1975 年早稲田大学理工学部数学科卒業．同年，日本 IBM（株）野洲工場入社．1983 年から米国プリンストン大学大学院（コンピュータ・サイエンス学科）．1985 年同大学から M.S.E. および M.A. 1987 年同大学から Ph.D. 同年より，日本 IBM（株）東京基礎研究所に移り，VLIW コンパイラ，HPF コンパイラ，JIT コンパイラ等のプロジェクトを担当．一貫して，プログラムを最適化して高速に実行させるための新しいソフトウェア技術について研究開発している．現在，IBM Distinguished Engineer，ネットワーク・コンピューティング・プラットフォーム担当．