Java 言語に対する投機的なメモリアクセスの最適化手法

川人基弘† 小松秀昭† 中谷登志男†

本稿では，partial redundancy elimination（PRE）の手法を使い，Java 言語に対して投機的に，コントロールフローの制御を超えてメモリアクセスの最適化を行う新しいアルゴリズムを提案する．我々の手法は最初に，メモリ移動を妨げる命令をできるだけ減らすために，エイリアス解析を行う．この解析の結果，各オブジェクト変数がエイリアスされない領域が求まる．次に，この領域を使ってすべてのメモリ移動を妨げる可能性のある命令を見つける．最後に，メモリの移動を次の 3 つのステップで行う．最初のステップでは，投機的ではない PRE のアルゴリズムを用いて，ロード命令と後続する演算命令等を，実行とは逆向きに移動させる．第 2 のステップでは，投機的な PRE のアルゴリズムを用いて，一部のロード命令と後続する演算命令等を，条件分岐の制御を超えて，実行とは逆向きに移動させる．第 3 のステップでは，ロード命令の最適化結果を利用することにより，ストア命令を実行方向にコントロールフローの合流点を超えて，投機的に移動させる．我々の手法は，投機的ロード命令等の特殊な命令を利用しないため，すべてのアーキテクチャに対して適用可能である．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．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．

A Method for Speculative Memory Access Optimizations for Java

Motohiro Kawahito,† Hideaki Komatsu† and Toshio Nakatani†

This paper presents a new algorithm for optimizing memory accesses using control speculation for Java by applying partial redundancy elimination (PRE) techniques. First, for reducing as many barriers as possible for enhancing code motions, we perform alias analysis to identify all the regions in which each object reference is not aliased. Secondly, using the regions computed by the alias analysis, we find all the possible barriers including those unidentified by the previous work. Finally, we perform code motions in three steps. For the first step, we apply a non-speculative PRE algorithm to move load instructions and their following instructions in the backward direction of the control flow graph. For the second step, we apply a speculative PRE algorithm to move some of them aggressively before the conditional branches. For the third step, we apply our modified version of a non-speculative PRE algorithm to move store instructions in the forward direction of the control flow graph and move some of them even after the merge points. Our approach does not require any special instructions, such as speculative load, and thus it can apply to all architectures. The same approach should work for any type-safety languages. We implemented the algorithm in the IBM Java Just-in-Time (JIT) compiler. Our experimental results show that our approach significantly improves performance over previously known algorithms.

1. はじめに

一般的に Java のようなオブジェクト指向言語は，データをオブジェクト内に閉じ込められるという利点があるが，その反面オブジェクト内のメモリアクセスが多くなるという傾向がある．これらの多くのメモリアクセスはパフォーマンスの低下を引き起こす．

この問題を解決するために，Fink [6] らによって Java 言語に対してインスタンス変数や配列等のグローバルなエリアに対するアクセスを，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（以下，scalar replacement と呼ぶ）が提案されている．この方法でも効果があるが，彼らの方法はメモリアクセス命令の移動を行っていないため，まだまだ改良の余地がある．

一般的に，メモリアクセス命令の移動の有無にかかわらず，scalar replacement を正しく適用するためには，最適化の障害となりうる命令を正しく設定しなければならない．さらに，2 つのオブジェクト変数のエイリアスを解析することは，パフォーマンスを向上させるために重要である．さらには，JIT コンパイラ向けには解析時間を短くすることも重要である．

本稿では，partial redundancy elimination（PRE）

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

の手法を使い，Java 言語に対して投機的に，コントロールフローの制御を超えてメモリアクセスの最適化を行う新しいアルゴリズムを提案する．我々の手法は最初に，メモリ移動を妨げる命令をできるだけ減らすために，エイリアス解析を行う．この解析の結果，オブジェクト変数がエイリアスされない領域が求まる．次に，この領域を使ってすべてのメモリの移動を妨げる可能性のある命令を見つける．本稿では，Java 言語に対する従来のメモリの最適化手法[6]では述べられていなかった synchronization や例外を起こしうる命令を適切に扱い，正しくメモリアクセスの移動を行う．最後に，我々はメモリの移動を次の 3 つのステップで行う．最初のステップでは，投機的ではない PRE のアルゴリズム[13]を用いて，ロード命令と後続する演算命令等を，実行とは逆向きに移動させる．第 2 のステップでは，投機的な PRE のアルゴリズム[10]を用いて，一部のロード命令と後続する演算命令等を，条件分岐の制御を超えて，実行とは逆向きに移動させる．第 3 のステップでは，ロード命令の最適化結果とブロックの実行頻度情報を利用することにより，ストア命令を実行方向にコントロールフローの合流点を超えて，投機的に移動させる．このステップでは，投機的ではない PRE のアルゴリズム[13]をストア命令の最適化向けに変更し適用している．

我々の手法の新規性は次の点である．

- Java が持っている，型安全性（type-safety）を保障する例外処理を，メモリアクセスを移動する際の防御壁として利用することにより，メモリアクセスを投機的にコントロールフローの制御を超えて移動させる．我々は，より広い範囲でメモリアクセスの移動を行うために，これらの例外処理を明示的な命令として分離し最適化を行う．型安全性を保障するための例外処理は実行時にコストがかかるため，型安全性を保障しない言語と比べて速度を落とす原因となっていた．本稿はこれらの例外処理を積極的に利用することにより，すべてのアーキテクチャ上で適用可能な，メモリアクセスを投機的にコントロールフローの制御を超えて移動させる方法を提案する．
- JIT コンパイラ向けに，次の 3 つの独自技術を考案し利用することで，コンパイル時間を抑えつつ効果の高い最適化を行う．
  - エイリアス解析によって，メソッド間解析を行わずに，メソッド呼び出しを超えてメモリアクセスの移動を行うことができる．
  - 実行頻度の高いロード命令に限定して，コンパイル時間がかかる投機的な PRE のアルゴリズムを適用している．
  - ストア命令を投機的に移動する手法として，bit vector を利用可能なアルゴリズムを適用している．

```
(a) C のサンプルプログラム   (b) 間違った変形
(1)                          (1) T = a->f1;
(2) do {                     (2) do {
(3)   if (...){              (3)   if (...){
(4)     b += a->f1;          (4)     b += T;
(5)   }                      (5)   }
(6) } while(...);            (6) } while(...);
```

図 1 投機的なロード命令の移動が正しく動作しない例
Fig. 1 Example where speculative scalar replacement produces an incorrect result.

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い，jBYTEmark と SPECjvm98 のベンチマークを用いて評価を行った．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．

本稿の以下の構成は次のようになっている．2 章では従来の関連研究について述べる．3 章では我々の手法について述べる．4 章では我々の手法の評価を行う．

2. 関連研究

Scalar replacement は，最初に Fortran 言語向けに開発され[2),3)]，次に C 言語向けに拡張が行われた[4),14)]．Java 言語向けには，最近 Fink らによって，ポインタのエイリアスを考慮した scalar replacement[6)]が提案された．しかし，Fink らの方法は命令の移動を行わないため，メモリアクセスがループの外に出ない場合があり，まだまだ改良の余地がある．我々は，null check の最適化の論文中[12)]で投機的ではない scalar replacement を簡単に紹介した．しかし，その詳細なアルゴリズムや投機的な scalar replacement については，その論文中でまったく述べられていない．

一般的な言語に対して投機的にすべてのロード命令を条件分岐の前に移動させるのは難しい問題である．たとえば，**図 1** (a) の C プログラムを例にとると，(b) のように (4) のロード命令 “a–>f1” を (3) の if 文を超えて (1) に移動させることは間違った最適化結果となる．これは，変数 a が不正なアドレスを指していたときに，プログラムが異常終了する場合があるためである．C 言語向けの投機的なメモリアクセスの最適化手法[14)]は，このようなポインタ参照のメモリアクセスを最適化対象から除外している．一般的には，不正なアドレスをアクセスする可能性がある命令は，アーキテ

クチャが特殊な命令を持っていなければ，コントロールフローの制御を超えて移動することはできない．たとえば，IA64 アーキテクチャは投機的なロード命令（ld.s）とその成否を調べる命令（chk.s）が用意されており，これらを利用すると図 1 の例に対しても，投機的にコントロールフローの制御を超えるような最適化が行える[20]．しかし，このような専用命令を持っているアーキテクチャは少なく，多くのアーキテクチャ上では同様の方法を使うことはできない．

その反面，Java 言語ではメモリアクセスに対して例外処理を行うことが決められているため，言語仕様上メモリアクセスについて安全性が保障されている．一般的に，この安全性のことは「型安全性」と呼ばれている．たとえば，null ポインタに対するメモリアクセスは，NullPointerException という例外を起こす．我々はこれらの例外処理を別命令に分離して，個々に最適化を行っている．これらの例外を起こす命令を，メモリを移動する際の防御壁として利用することで，安全性を保ちながらコントロールフローの制御を超えて，投機的にメモリアクセスの最適化を行うことができる．

しかし，Java でも投機的にロード命令を移動する際に，特別な処理をしなければならない場合がある．たとえば，null チェックの最適化[12]を行った場合，if (a != null) のような条件分岐があった場合，then 節内の a の null チェックはすべて除去される．この状態でロード命令を投機的に移動すると，防御壁として使っていた null チェックが除去されているために，間違った最適化結果となる場合がある．そのため，我々はこのような if 文のエッジ上にダミーの null チェックを挿入し，これを防御壁として利用する．これについての詳細は 3.1.3 項で説明する．

ストア命令を実行方向にコントロールフローの合流点を超えて，投機的に移動させる方法は，C 言語向けの投機的なメモリアクセスの最適化手法[14]内で提案されている．この方法は各メモリアクセスについて，Node profile の結果を元に Cost-Benefit 解析を行い，挿入する場所を決める．Java でも，本稿の 3.1.4 項で述べるストア命令の移動を妨げるような命令を適切に扱えば，この手法は適用可能である．しかし Java では C とは違い，例外を起こしうる命令を，ストア命令の移動を妨げる命令として扱わなければならない．Java では例外を起こしうる命令は非常に多くプログラム内に現れるため，ストア命令はロード命令に比べると最適化される機会が少ない．そのため，我々はストア命令の最適化では，コンパイル時間がかかる Cost-Benefit 解析を使わず，精度は落ちるが軽い最適化方法を使って投機的にストア命令を移動している．

Java 向けのエイリアス解析は Fink[6]らによって提案されている．この方法は，オブジェクト変数を定義している命令を調べることによって，2 つのオブジェクト変数がエイリアスされているか否かを判別している．しかし，彼らの解析方法はメソッド呼び出しに関しては，完全にメソッド間解析（interprocedural analysis）の結果に頼っていた．しかし，Java では呼ばれるメソッド内に virtual method call が含まれると，メソッド間解析は通常失敗する．virtual method は Java のプログラム内で頻繁に現れるため，メソッド間解析の効果は限定されている．それに対して我々の方法は，各オブジェクト変数が new されてからエイリアスされるまでの領域を見つけることにより，メソッド間解析を行わずにメソッド呼び出しを超えてメモリアクセスを移動することができる．これにより，最適化の効果が上がるだけではなく，メソッド間解析を行う必要があるメソッドを絞り込むことができるため，コンパイル時間を削減することができる．

Partial Redundancy Elimination（PRE）は実行方向とは逆向きに計算式を移動させることによって，部分的冗長性の除去およびループの外への移動を行う最適化技術である．PRE に関する研究は主に投機的ではないアルゴリズムと投機的なアルゴリズムの 2 種類に分類される．

我々は投機的ではない PRE アルゴリズムとして Lazy Code Motion（LCM）[13]を用いている．このアルゴリズムは，レジスタ・プレッシャの増加を抑えるために，実行方向とは逆向きに式を移動させたうえで，実行回数を増やさない範囲内で実行方向に再移動させ，最終的な挿入場所を決定する．これは，投機的な PRE アルゴリズムよりも効果は落ちるが，bit vector を使って問題を解くことができるため，コンパイル時間が短く JIT コンパイラ上でも積極的に適用できる．

投機的な PRE アルゴリズムとしては，Path profile を使うもの[9]，Edge profile を使うもの[10]，Node profile を使うもの[14]の 3 つが知られている．これらはすべて，各種 profile の情報を用い，Cost と Benefit の解析を行い，最終的な挿入場所を決定する．Path profile を使ったアルゴリズムは Edge profile を使うものよりも効果があるが，コンパイル時間が大きい．そのため，このアルゴリズムは JIT コンパイラで使うには実用的ではない．Edge profile を使ったアルゴリズムは Node profile を使うものよりも効果があり，コンパイル時間もほぼ同等である．そのため，我々は Edge profile を使ったアルゴリズム[10]を用いている．

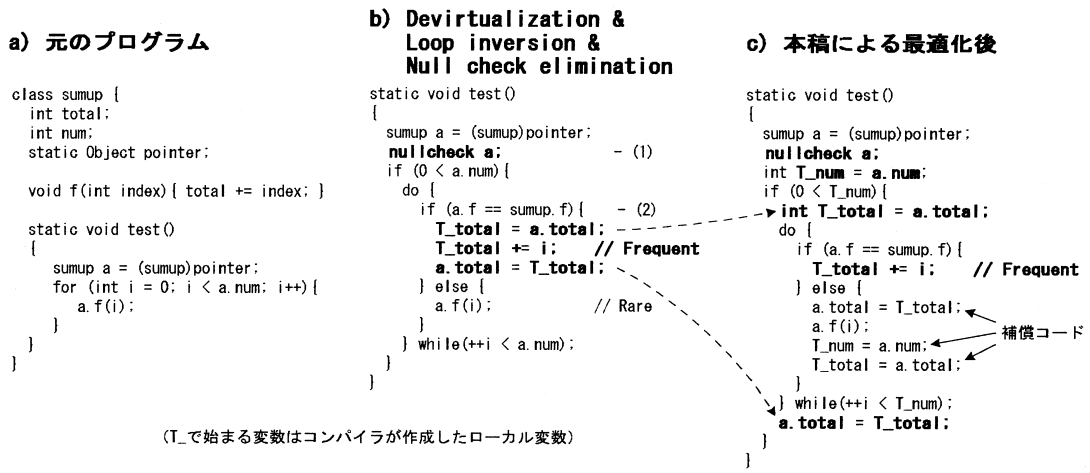

図 2 本稿の最適化を適用した例
Fig. 2 Example of our optimization.

投機的なアルゴリズムは，効果は大きいが個々の式について Cost と Benefit を求めなければならない．これらの計算は，bit vector を使って解くことができないため，複数の式を最適化する場合，コンパイル時間が長いという欠点がある．我々の手法は，投機的ではないアルゴリズムと投機的なアルゴリズムを組み合わせることで，全体のコンパイル時間を抑えつつ効果の高い最適化を行える．

Java のメモリアクセスに対してこれらの PRE アルゴリズムを適用するには，言語仕様（命令の順序制約）に違反しないように注意しなければならない．本稿では，Java 上のメモリアクセスの移動を妨げる命令を正しく設定することにより，この問題を解決している．

3. 我々の手法

図 2 の例を使って我々の最適化を説明する．test 内のメソッド呼び出し a.f(i) はオーバライド（override）されている可能性があるため，単純にクラス sumup 内の f( ) をインラインすることができない．そこで，(b) のように devirtualization [1),5),11),18)] を行い，(2) の if 文でガードして sumup.f() のメソッドをインラインする．ここではこのように変形することにより，インラインしたパスが頻繁に実行されると仮定する．なお，(b) では Loop inversion [15)]（while ループを do-while ループにする変形）および例外処理の最適化 [12)] も適用している．型安全性を保障しない言語では，変数 a が不正なアドレスを指す可能性があるため，インラインされたパス内のロード・ストア命令を (2) の if 文を超えて移動させることはできない．しかし，Java では (b)(1) で null かどうかを検査し，null ならば例外を起こす命令（以下 nullcheck と呼ぶ）を実行するために，(1) 以降の領域では a に対して投機的なメモリアクセスの最適化を行うことができる．なお，nullcheck という命令は，元の Java のバイトコードには直接含まれてはいない．我々は，効果的にプログラムを最適化するために，中間コードレベルで明示的な命令として生成している．(b) に対して最適化を行うと，(c) のようにインラインされたパス内のロード・ストア命令を (2) の if 文を超えて，ループの外に移動させることができる．このとき，インラインされなかったパス内のメソッド呼び出しでは，この中でメモリ上の値が参照または変更される可能性があるので補償コードが必要となる．ロード命令の補償コードは，メモリアクセスの移動を妨げる命令を適切に設定し，投機的な PRE のアルゴリズムを使うことにより，挿入すべき場所を求めている．ストア命令の補償コードは，挿入するロード命令をストア命令と仮定し，メモリアクセスの移動を妨げる命令を適切に設定し，LCM アルゴリズムを逆向きに使うことにより，挿入すべき場所を求めている．

以下，3.1 節ではメモリアクセスの最適化について述べ，3.2 節ではこれに関連する最適化全体の流れを述べる．

3.1 メモリアクセスの最適化

この最適化では，算術演算およびメモリアクセスを一度に同じ枠組みで最適化する．我々はすべての式を {opcode, src1, src2, misc} の 4 つ組で表現し，これを各ビットに割り振ってデータフロー解析を行っている．通常の演算命令は最初の 3 つを使えば表現できる．

表 1 メモリアクセスに対する，4 つ組の各フィールドの意味
Table 1 Meaning of each field of quadruple for memory access.

	opcode	src1	src2	misc
static 変数	StaticAccess	定数アドレス	0（使用せず）	シグニチャと名前
field 変数	FieldAccess	オブジェクト変数	定数オフセット	シグニチャと名前
配列	ArrayAccess	オブジェクト変数	オフセット（定数または変数）	型(int, byte, ...)

```
 1: ScalarReplacement() {
 2:   dummy exception check の挿入 ();
 3:   NoAliasPos = エイリアス解析 ();
 4:   /* ロード命令の最適化 */
 5:   NoAliasPos を使って TRANSP_L, N–COMP_L, X–COMP_L を求める.
 6:   X–INSERT_L = LazyCodeMotion(TRANSP_L, N–COMP_L, X–COMP_L);
 7:   HIGHLEVEL-TGT = { 実行頻度の高いロード命令・演算命令 };
 8:   if (HIGHLEVEL-TGT != φ)
 9:     X–INSERT_L ∪= CostBenefitPRE(TRANSP_L, N–COMP_L, X–COMP_L, HIGHLEVEL-TGT);
10:   N–REACH_L = ComputeReachForward(TRANSP_L, X–INSERT_L);
11:   /* ストア命令の最適化 */
12:   NoAliasPos を使って TRANSP_S, N–COMP_S, X–COMP_S を求める.
13:   N–INSERT_S = StoreOptimization(TRANSP_S, N–COMP_S, X–COMP_S, X–INSERT_L);
14:   X–REACH_S = ComputeReachBackward(TRANSP_S, N–INSERT_S);
15:   /* コードの変形 */
16:   CodeTransformation(X–INSERT_L, N–REACH_L, N–INSERT_S, X–REACH_S);
17: }
```

図 4 メモリ最適化のアルゴリズム
Fig. 4 Algorithm of our scalar replacement.

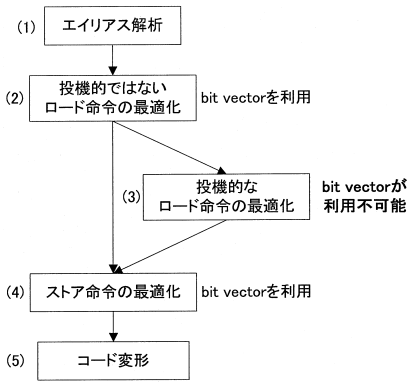

図 3 メモリアクセスの最適化の流れ
Fig. 3 Flow diagram of our scalar replacement algorithm.

ロード命令およびストア命令は，各フィールドに **表 1** に示すものがそれぞれセットされる．見れば分かるように，この 4 つ組表現上ではロード・ストアの区別はしていない．

図 3 に我々のメモリアクセスの最適化の流れを示す．まず最初にオブジェクト変数のエイリアス解析 (1) を行う．我々のロード命令の最適化は，投機的ではない最適化 (2) を基本とし，効果的であると思われるロード命令を限定して投機的な最適化 (3) を行う．

このようなデザインにしているのはコンパイルタイムの増加をできるだけ抑えるためである．(2) の最適化では，LCM[13] を用いている．これは，bit vector を使って問題を解くことができるため，コンパイル時間が短く JIT コンパイラ上でも積極的に適用できる．投機的な最適化 (3) は Cost-Benefit 解析に基づく PRE[10] を用いている．このアルゴリズムは問題を解くためにエッジの頻度情報を元に計算を行う．これは bit vector を使って表現できないため，個々のロード命令に対して毎回解析を行わなければならない．そのため，複数の式を最適化する場合，このアルゴリズムは LCM に比べて非常に遅い．たとえば，64-bit アーキテクチャで 64 個の式を最適化することを考えてみると，1 つの dataflow 解析に必要な時間は，bit vector が使えないアルゴリズムは使えるアルゴリズムより 64 倍以上必要となる．そこで，我々は実行頻度が高いロード命令や演算命令に限定して投機的な最適化 (3) を適用している．

ストア命令の最適化 (4) については，2 章で述べたように，ストア命令の移動を妨げる命令が Java では非常に多いために，コンパイル時間のかかる Cost-Benefit 解析を使わず，LCM アルゴリズムを変更して適用することによって，投機的にストア命令を移動している．

以上の説明を擬似コードで書いたアルゴリズムを **図 4** に示す．以下の節では，図 4 の各行の記号や処理内容について順次説明していく．

Java 言語では，マルチスレッドの環境を仮定しており，それをサポートするために volatile 変数と synchronization という 2 つの仕組みを提供している．Java の言語仕様[8] によると，volatile 変数をロード・

```
IsAliasPos(q, pos) {
   if (q.src1が定数) return TRUE;
   return !(変数q.src1 ∈ NoAliasPos(pos));
}
```

図5 関数 IsAliasPos()
Fig. 5 Function IsAliasPos().

```
MayAlias(q, r, pos) {
 if (q.opcode == r.opcode && q.misc == r.misc){
   if (q.src1 == r.src1 && q.src2 == r.src2)
     return FALSE; /* 完全に同じ4つ組 */
   if ((q.opcode == FieldAccess ||
        q.opcode == ArrayAccess) &&
       ((q.src1 == r.src1 ||
         (IsAliasPos(q, pos) && IsAliasPos(r, pos))) &&
        (q.src2 == r.src2 ||
         q.src2は変数 || r.src2は変数)))
     return TRUE;
 }
 return FALSE;
}
```

図6 関数 MayAlias()
Fig. 6 Function MayAlias().

ストアするたびに，volatile 変数の shared memory とプログラム内のローカル変数を一致させなければならないと規定されている．そこで，我々は volatile 変数と synchronization を scalar replacement の対象から除くとともにメモリアクセスを移動させる際のバリアとして扱っている．さらに，あるメモリ上の変数が resolve されていないときには，この変数が volatile かどうかを判別することができない．そのため，我々は resolve されていない変数も volatile 変数として扱っている．

3.1.1 オブジェクト変数のエイリアス解析

この解析の目的は，scalar replacement を行う際のバリアとなる命令をできるだけ減らすことである．この項では，我々は次の 2 つの関数を定義する．

IsAliasPos(q, pos) メモリアクセス q は pos の場所でエイリアスされている可能性がある．

MayAlias(q, r, pos) 2 つのメモリアクセス q，r は pos の場所で互いにエイリアスされている可能性がある．

まず最初にベーシックブロック n の先頭で，けっしてエイリアスされることがないオブジェクト変数の集合 NoAlias(n) を，前進データフロー解析（forward dataflow analysis）を使って，次のように求める．

$$\mathrm{NoAlias(n)} = \prod_{\mathrm{m}\in\mathrm{Pred(n)}} \begin{pmatrix} (\mathrm{NoAlias(m)} - \mathrm{Kill(m)}) \\ +\mathrm{Gen(m)} \end{pmatrix}$$

ここで，Gen，Kill は次のような集合を意味する．

```
a = new ...; - (1)
for ( . . . ){
  t += a.I;  - (2) // a.Iをループの外に移動可能
  f(t);      - (3)
}
b.obj = a;   - (4) // これ以降aはaliasされる可能性がある
for ( . . . ){
  t += a.I;  - (5) // a.Iをループの外に移動不可能
  f(t);      - (6)
}
```

図7 エイリアス解析が効果的な例
Fig. 7 Example where our alias analysis is effective.

Gen(n) ブロック n 内で new 等によって作成され，ブロック n の終わりでまだエイリアスされていないと判断できる変数の集合．

Kill(n) ブロック n 内でエイリアスされていないと判断できなくなる変数の集合．具体的には次のような変数が対象となる．

- メソッドの引数になっている変数．
- メモリに書き込まれた変数．
- コピー元の変数．
- 代入が行われた変数．

NoAlias(n) を使って各ブロック内を Kill となる命令を考慮して 1 命令ずつ検査していくことにより，各命令 pos の場所でけっしてエイリアスされることがないオブジェクト変数の集合 NoAliasPos(pos) を求めることができる（図 4，3 行目）．最後に，IsAliasPos，MayAlias をそれぞれ**図 5**，**図 6** に示すように定義する．

2 章で述べたように，我々のエイリアス解析はメソッド呼び出しに対して効果的である．**図 7** にそのような例を示す．この例で変数 a について (1) から (4) までが，けっしてエイリアスされることがない領域となる．そのため，最初のループでは，(3) の f(t) がどんなメソッドであっても，オブジェクト a 内のメモリは変更されることがないため，a.I をループの外に移動させることができる．しかし，2 つめのループでは，(6) の f(t) 内で (4) の b.obj を経由して，オブジェクト a 内のメモリを変更することが可能となる．そのため，任意の f(t) について，a.I をループの外に移動させることはできない．

3.1.2 投機的ではないロード命令の最適化

我々はロード命令に対するコード移動のアルゴリズムとして，Lazy Code Motion（LCM）[13]を使っている．このアルゴリズムは，TRANSP，N-COMP，X-COMP という 3 つの集合を入力として，X-INSERT という集合を最終的には求める．これら 4 つの集合はそれぞれ次のような意味である（なお，**N-**は entry，

X-は exit を意味している）.

TRANSP(n) ブロック n 内を通過することのできる，メソッド内の式の集合.

N-COMP(n) ブロック n の先頭に移動できる，ブロック n 内の式の集合.

X-COMP(n) ブロック n の最後に移動できる，ブロック n 内の式の集合.

X-INSERT(n) ブロック n の最後に挿入する式の集合.

本稿では，ロード命令に関するこれら 4 つの集合をそれぞれ $\mathrm{TRANSP_L}$，$\mathrm{N–COMP_L}$，$\mathrm{X–COMP_L}$，$\mathrm{X–INSERT_L}$と表す．LCM アルゴリズムは計算式を最適化対象としたアルゴリズムのため，計算式に含まれる変数に対する書き込みのみを，式の移動を妨げるような命令として扱っている．しかし，Java でメモリアクセスを移動させる際には，言語仕様に違反しないように，移動を妨げる命令を正しく定義する必要がある．表 1 で示した {opcode, src1, src2, misc} の 4 つ組に対応する，あるロード命令 q について，ブロック n 内の移動を妨げる命令としては次の命令があげられる．

ロード命令 q について移動を妨げる命令

- q.src1 や q.src2 の変数に対する書き込み.
- q.src1 の変数に対する nullcheck．ただし，オブジェクト変数が null の場合に，ロード命令 q が副作用を起こさないと分かる場合は除く.
- 配列のロード命令に関しては，q.src1，q.src2 に対する配列の境界チェック.
- 同期をとる命令（monitorenter，monitorexit）B について IsAliasPos(q，B の場所) が成り立つ場合.
- volatile 変数や unresolved 変数についてのロードやストア命令 B について，IsAliasPos(q，B の場所) が成り立つ場合.
- ストア命令 B☆について，MayAlias(q，B，B の場所) が成り立つ場合.
- メソッド呼び出し B について，次のどちらの条件も満たす場合.
  - IsAliasPos(q，B の場所) が成り立つ.
  - q が B 内で変更されないことがメソッド間解析（interprocedural analysis）や field analysis [7] により判別できない.

☆ 厳密にいえば，memory coherence の仮定を持つ現在の Java memory model（JMM）の仕様からすると，「ロード命令 B」もバリアとしてここに加える必要がある[16]．しかし，この仕様は Java Specification Request（http://jcp.org/jsr/detail/133.jsp）で今修正される方向で話が進んでいるため，ここには加えなかった.

a) Null チェック除去に関する問題

```
if (a != null){          --- (1)
   nullcheck a;          --- (2)
   t = a.f1;             --- (3)
}
```

b) 配列境界チェック除去に関する問題

```
nullcheck a;                              --- (1)
boundcheck 0, a[];                        --- (2)
a[0] = 0;                                 --- (3)
if (0 <= i && i < a.arraylength){         --- (4)
   nullcheck a;                           --- (5)
   boundcheck i, a[];                     --- (6)
   t = a[i];                              --- (7)
}
```

図 8 コントロールフローの条件によって除去される例外チェック
Fig. 8 Exception checks eliminated because an equivalent test statement precedes them.

$\mathrm{TRANSP_L(n)}$ は，各ロード命令 q についてブロック n 内に移動を妨げる命令がまったくなければ，q を集合に追加するという処理をすることにより作成できる．$\mathrm{N–COMP_L(n)}$，$\mathrm{X–COMP_L(n)}$ は，ブロック n 内に含まれるロード命令 q について，n 内に前述の移動を妨げる命令が q の場所より前・後になければ，q を $\mathrm{N–COMP_L(n)}$，$\mathrm{X–COMP_L(n)}$ にそれぞれ追加するという処理をすることにより作成できる（図 4，5 行目）.

$\mathrm{TRANSP_L}$，$\mathrm{N–COMP_L}$，$\mathrm{X–COMP_L}$の 3 つの集合が正しく求まれば，LCM アルゴリズム[13]を通常の計算式の移動と同様に適用できる．このアルゴリズムを解くと，各ブロック n の最後の点に挿入を行うロード命令の集合 $\mathrm{X–INSERT_L}$が求まる（図 4，6 行目）.

3.1.3 投機的なロード命令の最適化

型安全性を持つ Java においても，投機的なロード命令の最適化を行うには特別な処理が必要となる．これは，例外除去処理がコントロールフローの条件を利用して，例外チェック命令を除去する場合があるためである．**図 8** に投機的に最適化を行う際に問題が起きる場合を示す．(a) は (1) の if 文によって (2) が除去され，(b) は (4) の if 文によって (6) の配列境界チェックが除去される．このように除去された状態で，(a)(3) や (b)(7) のロード命令を (a)(1) や (b)(4) の if 文を超えて移動すると型安全性が保障されなくなってしまう.

そこで，例外除去処理を行う際に，コントロールフローの条件によってメモリアクセスに関連する例外チェックが除去される可能性があるコントロールフローのエッジを特定する．我々の実装では，nullcheck が除去される可能性があるエッジは ifnull，ifnonnull，instanceof-if のバイトコードによってオブジェクト変

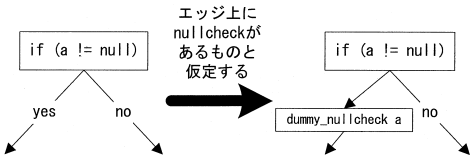

図 9 ifnonnull に対する処理例
Fig. 9 Example of handling an *ifnonnull*.

数が null ではないと判断できるエッジとなる．また，配列の境界チェック（以下 boundcheck と呼ぶ）に関しては，arraylength との比較命令によって boundcheck が除去される可能性があるエッジが対象となる．

次に，メモリアクセスを投機的にコントロールフローの制御を超えて移動させる最適化では，例外除去処理で特定したエッジ上に除去対象となる例外があるものと仮定し，最適化を行う．**図 9** に ifnonnull に対する処理例を示す．図 9 内の条件分岐が成り立つパス内では，a に対する null チェックが除去される可能性がある．そこで，a が null ではないと分かるエッジ上に，a に対して dummy の null チェックを挿入する（図 4，2 行目）．dummy の命令は，基本的には NOP（no-operation）命令として扱うが，投機的にメモリアクセスをコントロールフローの制御を超えて移動させるような最適化は，dummy ではない命令と見なして処理を行う．

例外除去処理が特定したエッジ上に例外チェック命令があると仮定することにより，ロード命令も通常の計算式と同様に，Cost-Benefit 解析に基づく PRE [10] を用いることで最適化が行える（図 4，9 行目）．

投機的なロード命令の最適化は図 2 のような devirtualization の例にとどまらず，ループ不変のメモリアクセスのコード移動にも効果的である．**図 10** は java/util/StringTokenizer.scanToken の一部分に対して最適化を行った場合を示している．元のプログラム (a) に対して，charAt のインラインおよび loop inversion を適用すると，(b) のように変形される．(b) に対して投機的な最適化をやめた場合には，(c) のようにループ内は this.str を置き換えただけで終わってしまう．これに対して投機的な最適化を行った場合には，(d) のように (1) の if 文を超えて新たに 4 つのメモリアクセスがループの外に移動できる．興味深いことに，(d)(2) の this.delimiters はループ不変のメモリアクセスだが，Cost-Benefit 解析により，ループ外に移動しても効果はないと判断され，ループ内にとどまった．なお，我々は小さなメソッドに対して，メソッド間解析を行っている．これにより，(d)(2) の indexOf メソッド内では，メモリへの書き込みがまったくないことを判別したため，この例では補償コードは生成されなかった．

投機的ではないロード命令の最適化と投機的なロード命令の最適化のアルゴリズムの出力は，ともにブロック n の最後に挿入するロード命令の集合である．そこで，図 4 の 6，9 行目に示すように，この 2 つの集合の和集合を X–INSERT$_{\mathrm{L}}$(n) とする．ロード命令の最適化の最後の処理として，各ベーシックブロックの先頭，最後に到達可能なロード命令の集合 N–REACH$_{\mathrm{L}}$(n)，X–REACH$_{\mathrm{L}}$(n) を求め，N–REACH$_{\mathrm{L}}$を使って各ベーシックブロック内の最適化を行う．図 4 の 10 行目 ComputeReachForward は次の前進データフロー方程式を解く処理を意味する．

$$\mathrm{N\text{–}REACH_L}(n) = \prod_{m \in \mathrm{Pred}(n)} \mathrm{X\text{–}REACH_L}(m)$$

$$\mathrm{X\text{–}REACH_L}(n) = \mathrm{X\text{–}INSERT_L}(n) + \mathrm{N\text{–}REACH_L}(n) * \mathrm{TRANSP_L}(n)$$

図 4 の 16 行目 CodeTransformation の処理で集合 N–REACH$_{\mathrm{L}}$および X–INSERT$_{\mathrm{L}}$を使って各ブロック内の冗長な式の除去と式の挿入を行う．

3.1.4 ストア命令の最適化

我々はストア命令の最適化のために Lazy Code Motion アルゴリズム [13] を逆向きに適用している．さらに，ロード命令の最適化と node profile の結果を利用することにより，より効果的にストア命令の最適化を行っている．

図 11 は図 2 (b) のプログラムを図で表したものである．ロード命令の最適化を行う前（図 11 (a)）では，(3) に含まれるストア命令 “a.total = T_total;” は (6) に移動することができない．一方，ロード命令の最適化を行った後（図 11 (b)）では，(4) の灰色の三角で示した場所にストア命令の補償コードを挿入することで，(3) のストア命令を (6) に移動することができるようになる．この変形は，ロード命令 “T_total = a.total;” が (1) と (4) の灰色の丸で示した場所に挿入されるために，正しく行うことができる．ただし，このような変形を行うと場合によっては遅くなる場合がある．たとえば，ブロック (2) からのエッジの頻度が逆の場合には，このような変形はストア命令の実行回数を増やしてしまう．この問題を解く方法としては，2 章で述べたように，Cost と Benefit を計算する方法 [14] が考えられる．しかし，我々はコンパイル時間と効果を考慮して，精度は落ちるが軽い最適化方法を使っている．

ストア命令に対する最適化も 3.1.2 項のロード命令の最適化と同様に，TRANSP，N-COMP，X-COMP という 3 つの集合を入力として計算する．出力に関し

a) 元のプログラム

```
private int scanToken(int startPos) {
  int position = startPos;
  while (position < this.maxPosition) {
    char c = this.str.charAt(position);
    if ((c <= this.maxDelimChar) &&
        (this.delimiters.indexOf(c) >= 0))
      break;
    position++;
  }
```

b) charAt をインラインし，Loop inversion を適用する

```
private int scanToken(int startPos) {
  int position = startPos;
  if (position < this.maxPosition) {
    do {
      char c;
      nullcheck this.str;  // メソッド charAt のための nullcheck
      if ((position < 0) || (position >= this.str.count)) {
        throw new StringIndexOutOfBoundsException(position);
      } else {
        c = this.str.value[position + this.str.offset]
        if ((c <= this.maxDelimChar) &&
            (this.delimiters.indexOf(c) >= 0))
          break;
        position++;
      }
    } while (position < this.maxPosition);
  }
```

c) 投機的に最適化しない場合

```
private int scanToken(int startPos) {
  int position = startPos;
  T_maxPosition = this.maxPosition;
  if (position < T_maxPosition) {
    T_str = this.str;
    nullcheck T_str;
    do {
      char c;
      if ((position < 0) || (position >= T_str.count)) {    --- (1)
        throw new StringIndexOutOfBoundsException(position);
      } else {
        c = T_str.value[position + T_str.offset]
        if ((c <= this.maxDelimChar) &&
            (this.delimiters.indexOf(c) >= 0))
          break;
        position++;
      }
    } while (position < T_maxPosition);
  }
```

d) 投機的に最適化した場合

```
private int scanToken(int startPos) {
  int position = startPos;
  T_maxPosition = this.maxPosition;
  if (position < T_maxPosition) {
    T_str = this.str;
    nullcheck T_str;
    T_value = T_str.value;
    T_offset = T_str.offset;
    T_count = T_str.count;
    T_maxDelimChar = this.maxDelimChar;
    do {
      char c;
      if ((position < 0) || (position >= T_count)) {        --- (1)
        throw new StringIndexOutOfBoundsException(position);
      } else {
        c = T_value[position + T_offset]
        if ((c <= T_maxDelimChar) &&
            (this.delimiters.indexOf(c) >= 0))              --- (2)
          break;
        position++;
      }
    } while (position < T_maxPosition);
  }
```

図 10 ループ不変のメモリアクセスに対する scalar replacement
Fig. 10 Scalar replacement for loop invariant memory access.

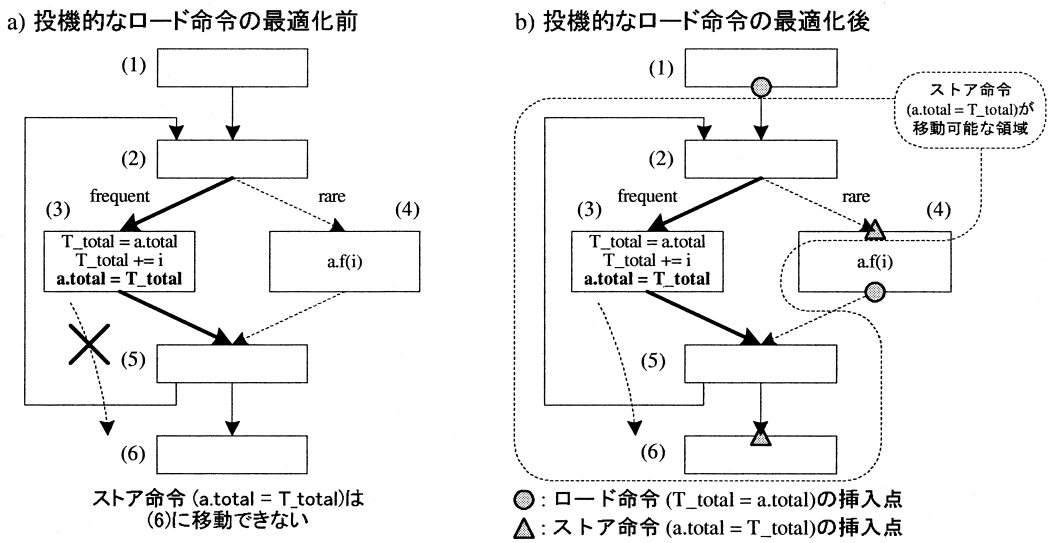

図 11 投機的なストア命令の最適化例
Fig. 11 Example of our speculative store optimization.

1. Safety Analysis: (Forward Dataflow Analysis)

$$\mathrm{N\text{–}DSAFE_S(n)} = \prod_{\mathrm{m\in Pred(n)}} \mathrm{X\text{–}DSAFE_S(m)}$$

$$\mathrm{X\text{–}DSAFE_S(n)} = \begin{cases} \mathrm{X\text{–}COMP_S(n)} + (\mathrm{N\text{–}DSAFE_S(n)} * \mathrm{TRANSP_S(n)}) & (n : frequent) \\ \mathrm{X\text{–}COMP_S(n)} + (\mathrm{N\text{–}DSAFE_S(n)} * \mathrm{TRANSP_S(n)}) + \mathbf{X\text{–}INSERT_L(n)} & (otherwise) \end{cases}$$

2. Computation of Latestness: (No dataflow analysis)

$$\mathrm{N\text{–}LATEST_S(n)} = \mathrm{N\text{–}DSAFE_S(n)} * \overline{\mathrm{TRANSP_S(n)}}$$

$$\mathrm{X\text{–}LATEST_S(n)} = \mathrm{X\text{–}DSAFE_S(n)} * \sum_{\mathrm{m\in Succ(n)}} \overline{\mathrm{N\text{–}DSAFE_S(m)}}$$

3. Anticipatable Analysis: (Backward Dataflow Analysis)

$$\mathrm{N\text{–}ANTIC_S(n)} = \mathrm{N\text{–}LATEST_S(n)} + \mathrm{X\text{–}ANTIC_S(n)} * \overline{\mathrm{X\text{–}COMP_S(n)}}$$

$$\mathrm{X\text{–}ANTIC_S(n)} = \mathrm{X\text{–}LATEST_S(n)} + \prod_{\mathrm{m\in Succ(n)}} \mathrm{N\text{–}ANTIC_S(m)} * \overline{\mathrm{N\text{–}COMP_S(m)}}$$

4. Computation of Earliestness: (No dataflow analysis)

$$\mathrm{N\text{–}EARLIEST_S(n)} = \mathrm{N\text{–}ANTIC_S(n)} * (\mathrm{N\text{–}COMP_S(n)} + \overline{\prod_{\mathrm{m\in Pred(n)}} \mathrm{X\text{–}ANTIC_S(m)}})$$

$$\mathrm{X\text{–}EARLIEST_S(n)} = \mathrm{X\text{–}ANTIC_S(n)} * \mathrm{X\text{–}COMP_S(n)}$$

5. Isolation Analysis: (Forward Dataflow Analysis)

$$\mathrm{N\text{–}ISOLATED_S(n)} = \prod_{\mathrm{m\in Pred(n)}} \mathrm{X\text{–}LATEST_S(m)} + (\overline{\mathrm{X\text{–}COMP_S(m)}} * \mathrm{X\text{–}ISOLATED_S(m)})$$

$$\mathrm{X\text{–}ISOLATED_S(n)} = \mathrm{N\text{–}ISOLATED_S(n)} + \mathrm{N\text{–}LATEST_S(n)}$$

6. Insertion Points of Store Optimization: (No dataflow analysis)

$$\mathrm{N\text{–}INSERT_S(n)} = (\mathrm{N\text{–}EARLIEST_S(n)} * \overline{\mathrm{N\text{–}ISOLATED_S(n)}}) + (\mathrm{X\text{–}EARLIEST_S(n)} * \overline{\mathrm{X\text{–}ISOLATED_S(n)}} * \mathrm{TRANSP_S(n)})$$

7. Reachability Analysis: (Backward dataflow analysis)

$$\mathrm{N\text{–}REACH_S(n)} = \mathrm{N\text{–}INSERT_S(n)} + (\mathrm{X\text{–}REACH_S(n)} * \mathrm{TRANSP_S(n)})$$

$$\mathrm{X\text{–}REACH_S(n)} = \prod_{\mathrm{m\in Succ(n)}} \mathrm{N\text{–}REACH_S(m)}$$

図 12 ストア命令の最適化アルゴリズム
Fig. 12 Algorithm of our store optimization.

ては，Lazy Code Motion を逆向きに適用するために，ブロック n の**最初**に挿入する式の集合 N-INSERT(n) が結果となる．

本稿では，ストア命令に関するこれら 4 つの集合をそれぞれ $\mathrm{TRANSP_S}$，$\mathrm{N\text{–}COMP_S}$，$\mathrm{X\text{–}COMP_S}$，$\mathrm{N\text{–}INSERT_S}$ と表す．入力の 3 つの集合を求めるためには，各ストア命令についてブロック n の通過を妨げるかどうかを正しく判別する必要がある．表 1 で示した {opcode, src1, src2, misc} の 4 つ組に対応する，あるストア命令 q について，ブロック n 内の移動を妨げる命令としては次のようなものがあげられる．

ストア命令 q について移動を妨げる命令・条件

- q.src1 や q.src2 の変数に対する書き込み．
- 任意の例外を起こしうる命令 B について，（B は try ブロック内）または（IsAliasPos(q，B の場所)）のどちらかが成り立つ場合．
- 同期をとる命令（monitorenter，monitorexit）B について IsAliasPos(q，B の場所) が成り立つ場合．
- volatile 変数や unresolved 変数についてのロードやストア命令 B について，IsAliasPos(q，B の場所) が成り立つ場合．
- ロードまたはストア命令 B について，MayAlias(q，B，B の場所) が成り立つ場合．
- メソッド呼び出し B について，次のどちらの条件も満たす場合．

– IsAliasPos(q，B の場所) が成り立つ．
– q が B 内でアクセスされないことがメソッド間解析や field analysis [7] により判別できない．

- $q \in X\text{–}INSERT_L(n)$ が成り立つ場合には，n の最後の点．

$TRANSP_S(n)$ は，各ストア命令 q についてブロック n 内に移動を妨げる命令がまったくなければ，q を集合に追加するという処理をすることにより作成できる．$N\text{–}COMP_S(n)$，$X\text{–}COMP_S(n)$ は，ブロック n 内に含まれるストア命令 q について，n 内に前述の移動を妨げる命令が q の場所より前・後になければ，q を $N\text{–}COMP_S(n)$，$X\text{–}COMP_S(n)$ にそれぞれ追加するという処理をすることにより作成できる（図 4，12 行目）．

これら 3 つの集合を出発点として，**図 12** の方程式を解く．これらの方程式はストア命令を実行方向に移動させるため，LCM アルゴリズム [13] の各解析の方向を逆向きに変え，さらにロード命令の最適化結果を利用する処理を追加している．この中で，ロード命令の最適化結果を利用している部分は，$X\text{–}DSAFE_S(n)$ を求めている部分である．n が頻繁に実行されないときに限り，ロードの最適化の処理で求めた $X\text{–}INSERT_L(n)$ を加えることによって，先ほど述べた実行回数が増加する問題を解決している．なお，ストア命令の投機的な移動をやめた版を作成するには，つねにすべての n が頻繁に実行すると仮定することにより作成できる（すなわち，$X\text{–}INSERT_L(n)$ を加えない）．

図 12 の方程式を解いた結果，各ブロック n の最初の点に挿入を行うストア命令の集合 $N\text{–}INSERT_S$ が求まる（図 4，13 行目）．図 4 の 14 行目 ComputeReachBackward は図 12 の 7 に示した後進データフロー解析を解く処理を意味する．これを解くことにより，各ベーシックブロックの最後に到達可能なストア命令の集合 $X\text{–}REACH_S(n)$ が求まる．

最後に，図 4 の 16 行目の処理で集合 $X\text{–}REACH_S$ および $N\text{–}INSERT_S$ を使って各ブロック内の冗長なストア命令の除去とストア命令の挿入を行う．

3.2 最適化全体の流れ

図 13 にメモリの最適化に関連する最適化全体の流れを示す．まず最初に (1) でメモリアクセスの最適化に影響する暗黙的な処理を分離して，明示的な命令として表現する．暗黙的な処理としては**例外処理**と**クラスの初期化チェック**があげられる．これらの処理はともにメモリアクセス命令の移動を阻害するため，明示的な命令として表現し，個々に最適化を適用する．

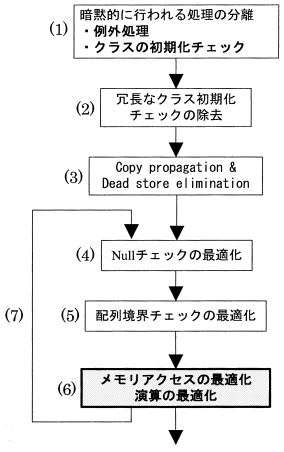

図 13 我々の最適化全体の流れ
Fig. 13 High-level flow diagram of our optimizations.

Java ではメモリアクセスに関連する例外処理として null チェックと配列の境界チェックがある．クラスの初期化チェックは対象のクラスが初期化されていなければ，クラスの初期化メソッド（⟨clinit⟩）を呼ぶ．そのため，メモリアクセスの最適化では基本的に，クラスの初期化チェックをメソッド呼び出しと同等に扱わなければならない．これは，最適化の効果を抑制する．そこで，メモリアクセスの最適化の効果を高めるために，前進データフロー解析を使って，冗長な「クラスの初期化チェック」を除去する (2)．

次に，copy propagation と dead store elimination (3) を行い，コピー命令を極力減らす．次に分離した nullcheck と boundcheck に対して PRE を使った最適化 (4)，(5) を行う [12),18)]．なお，nullcheck の最適化の論文 12) には，例外を起こしうる命令の移動を妨げる命令や条件が書かれている（たとえば，異なる try-region をまたぐような移動を行わない等）．我々は，boundcheck に関しても nullcheck の最適化と同じような方法で最適化を行っている．これらの処理によって，メモリアクセスに関連する例外を起こす命令が，最小限必要な位置に置かれる．

次に，本稿で説明したメモリアクセスの最適化 (6) を行う．(2)，(4)，(5) によってメモリアクセスの移動を阻害する命令が最小限となっているために，広い範囲で最適化が可能となる．また，(6) の最適化によって (4)，(5) の最適化の機会を増やす．そのため，繰返し (7) を行うことによって最適化の効果をさらに高められる [12)]．

4. 実験結果

我々は jBYTEmark と SPECjvm98 [17] の 2 つのベ

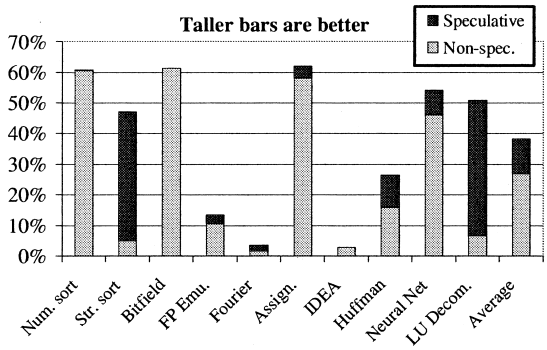

図 14 jBYTEmark 上で除去できたメモリアクセスの実行回数の割合
Fig. 14 Dynamic counts of memory operations eliminated for jBYTEmark.

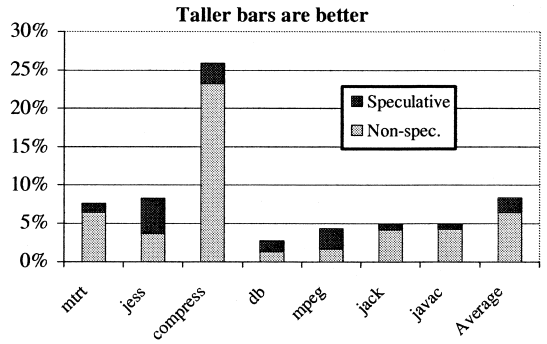

図 15 SPECjvm98 上で除去できたメモリアクセスの実行回数の割合
Fig. 15 Dynamic counts of memory operations eliminated for SPECjvm98.

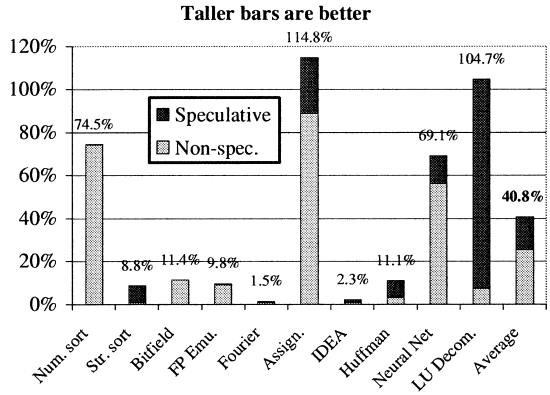

図 16 jBYTEmark に対する速度向上率
Fig. 16 Performance improvement for jBYTEmark.

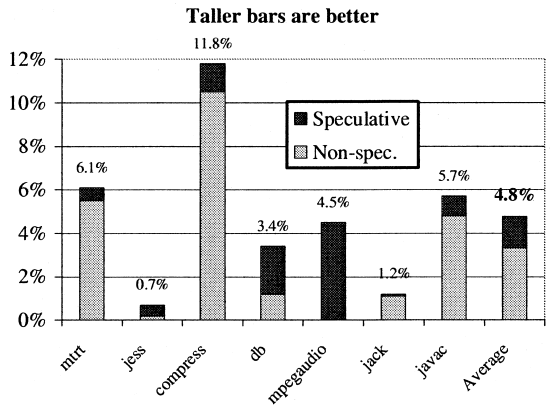

図 17 SPECjvm98 に対する速度向上率
Fig. 17 Performance improvement for SPECjvm98.

ンチマークを測定し，本アルゴリズムの評価を行った．すべての測定結果を同じ環境で行うために，コマンドラインからベンチマークを個別に実行させた．jBYTEmark の測定方法は問題の大きさをそれぞれ固定して実行させた．SPECjvm98 の測定方法は問題の大きさを 100 にして実行させた．すべての測定結果は，POWER3 400 MHz，メモリ 768 MB，AIX4.3.3，IBM Developer Kit for AIX，Java Technology Edition に変更を加えて測定した．

我々の JIT コンパイラではインタープリタとダイナミック・コンパイラを組み合わせて使っている[19]．まず，インタープリタでメソッドを実行し，頻繁に実行されると判断するとダイナミック・コンパイラに遷移してコンパイル・実行を行う．この組合せは，コンパイル時間を減らすだけではなく，resolve されていない変数やクラスの初期化チェックを減らすことにも貢献している．さらに，我々はインタープリタ上で条件分岐に関して統計をとっている．この条件分岐に関する統計を基に，各エッジの分岐確立を求めて，投機的なロード命令の最適化を行っている．

4.1 本アルゴリズムによる効果

本アルゴリズムの比較を行うために，次の版を作成し測定を行った．

Baseline コード移動をやめて scalar replacement を行う版．従来の Fink[6] らの方法を想定しており，コード移動をやめる以外の処理は有効にしている．

Non-spec. 投機的な scalar replacement（図 3 (3)）をやめた版．

Speculative 本稿のすべての処理を有効にした版．

図 14，**図 15** に Baseline の実行回数に対して，jBYTEmark と SPECjvm98 上で除去できた割合を示す．これは，コンパイルされたコード内にメモリアクセスを行う命令を生成する際に，回数を数えるコードも同時に生成させて得たものである．この結果，我々のアルゴリズムは Baseline で実行されたメモリアク

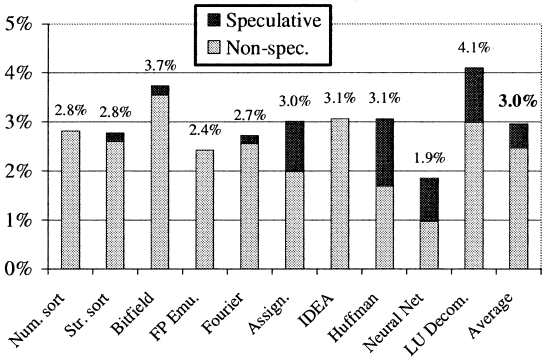

図 18 jBYTEmark に対するコンパイル時間の増加
Fig. 18 JIT compilation time increases for jBYTEmark.

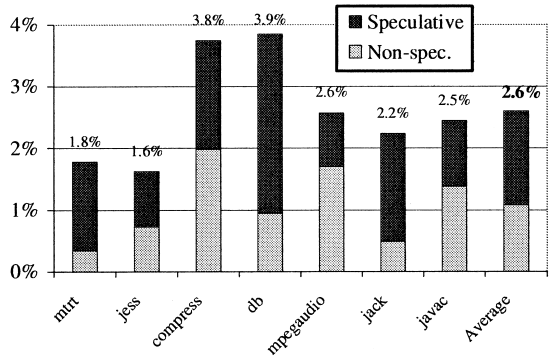

図 19 SPECjvm98 に対するコンパイル時間の増加
Fig. 19 JIT compilation time increases for SPECjvm98.

セスの実行回数のうち，jBYTEmark では平均 38%，SPECjvm98 では平均 8%減らすことができた．

図 16，**図 17** に Baseline と比較して，それぞれ jBYTEmark と SPECjvm98 上で達成できた速度向上率を示す．我々のアルゴリズムによって jBYTEmark では平均 41%，SPECjvm98 では平均 5%速度を向上させることができた．投機的ではない scalar replacement（Non-spec.）はほとんどのベンチマークで効果が認められた．投機的な scalar replacement（Speculative）は LU Decomposition で特に効果が認められた．このベンチマークを調べた結果，次のようなことが分かった．まず，多次元配列のアドレスを持つインスタンス変数のアクセスが，投機的な最適化によって条件分岐を超えてループ外に移動した．これによって，この多次元配列に対するアクセスのいくつかがループ不変となり，null チェック，配列境界チェック，scalar replacement 等の最適化が効果的に適用できるようになり，速度が大きく向上した．

4.2 JIT のコンパイル時間

本節では，我々の手法がどの程度，JIT のコンパイル時間に影響を与えているかについて述べる．我々は，AIX 上で利用可能なトレースツールを用いて，JIT のコンパイル時間を測定した．**図 18**，**図 19** に jBYTEmark と SPECjvm98 上で増加したコンパイル時間の割合を示す．我々の手法は jBYTEmark では平均 3.0%，SPECjvm98 では平均 2.6%コンパイル時間を増加させることが分かった．

5. 終わりに

本稿では，Java 言語に対して投機的に，コントロールフローの制御を超えてメモリアクセスの最適化を行うアルゴリズムについて述べた．ロード命令を投機的に移動する際には，例外除去処理が指定したコントロールフロー上のエッジに例外処理命令があると仮定して最適化を行う．また，オブジェクト変数のエイリアス解析により，メソッド呼び出し等の命令に対しても一部のメモリアクセス命令の移動は妨げられなくなった．ストア命令の最適化では，ロード命令の最適化の結果を利用して実行することにより，より多くのストア命令をループの外に移動させることが可能となった．この手法は Java 言語に限らず，メモリアクセスについて例外処理により言語仕様上安全性が保障されている言語に対して適用可能である．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 グループの皆様に深く感謝します．

参考文献

1) Aigner, G. and Holzle, U.: Eliminating Virtual Function Calls in C++ Programs, *10th European Conference on Object-Oriented Programming — ECOOP '96*, pp.142–166 (1996).
2) Bodik, R. and Gupta, R.: Array Data Flow Analysis for Load-Store Optimizations in Superscalar Architectures, *Languages and Compilers for Parallel Computing*, pp.1–15 (1995).
3) Carr, S. and Kennedy, K.: Scalar Replacement in the Presence of Conditional Control Flow, *Software — Practice and Experience*, Vol.24, No.1, pp.51–77 (1994).
4) Cooper, K.D. and Lu, J.: Register promotion in C programs, *Conference on Programming Language Design and Implementation*, pp.308–319 (1997).
5) Dean, J., Grove, D. and Chambers, C.: Optimization of object-oriented programs using static class hierarchy, *9th European Conference on Object-Oriented Programming — ECOOP*

'95, pp.77–101 (1995).
6) Fink, S.J., Knobe, K. and Sarkar, V.: Unified Analysis of Array and Object References in Strongly Typed Languages, *Static Analysis Symposium*, pp.155–174 (2000).
7) Ghemawat, S., Randall, K.H. and Scales, D.J.: Field analysis: Getting useful and low-cost interprocedural information, *Conference on Programming language design and implementation*, pp.334–344, ACM Press (2000).
8) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley Publishing Co., Reading (1996).
9) Gupta, R., Berson, D. and Fang, J.: Path Profile Guided Partial Redundancy Elimination Using Speculation, *IEEE Conference on Computer Languages* (1998).
10) Horspool, R. and Ho, H.: Partial redundancy elimination based on a cost-benefit analysis, *8th Israeli Conference on Computer Systems and Software Engineering*, Herzliya, Israel, pp.111–118, IEEE Computer Society (1997).
11) 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, *ACM SIGPLAN Java Grande Conference* (1999).
12) Kawahito, M., Komatsu, H. and Nakatani, T.: Effective Null Pointer Check Elimination Utilizing Hardware Trap, *9th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), Cambridge, MA, pp.139–149 (2000).
13) Knoop, J., Rüthing, O. and Steffen, B.: Optimal code motion: Theory and practice, *ACM Trans. Progr. Lang. Syst.*, Vol.17, No.5, pp.777–802 (1995).
14) Lo, R., Chow, F., Kennedy, R., Liu, S. M. and Tu, P.: Register promotion by Sparse Partial Redundancy Elimination of Loads and Stores, *Conference on Programming Language Design and Implementation*, pp.26–37 (1998).
15) Muchnick, S.S.: *Advanced compiler design and implementation*, Morgan Kaufmann Publishers, Inc. (1997).
16) Pugh, W.: Fixing the Java Memory Model, *Java Grande Conference*, pp.89–98 (1999).
17) Standard Performance Evaluation Corp.: SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/
18) 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).
19) Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H. and Nakatani, T.: A Dynamic Optimization Framework for a Java Just-In-Time Compiler, *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2001).
20) Zahir, R., Ross, J., Morris, D. and Hess, D.: OS and Compiler Considerations in the Design of the IA-64 Architecture, *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, pp.212–221 (2000).

(平成 14 年 8 月 21 日受付)
(平成 15 年 1 月 7 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 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 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，日本アイ・ビー・エム（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．