大域的なコード移動を使った複数式の実行コスト削減法

川人基弘[†] 小松秀昭[†] 中谷登志男[†]

コンパイラ上で式を最適化する方法は，大きく分けると冗長な式の実行回数を減らす方法と式の演算強度を軽減させる方法の 2 種類がある．本稿では，実行回数を減らしつつ，アーキテクチャの性質を利用して複数の式をより演算強度が軽い式へ合成させる新しいアルゴリズムを提案する．本稿のアルゴリズムは，依存関係を持つ式だけではなく，依存関係がない複数の式も合成できる点が大きな特徴である．さらにこのアルゴリズムは，多くの言語やアーキテクチャに対して適用可能である．我々は IA-64 上の IBM の Java Just-in-Time（JIT）コンパイラに対して，このアルゴリズムを使った 2 種類の最適化の実装を行い，SPECjvm98 のベンチマークを用いて評価を行った．

A Method for Reducing Execution Cost of Multiple Expressions by Using Global Code Motion

Motohiro Kawahito,[†] Hideaki Komatsu[†] and Toshio Nakatani[†]

In the past, there have been two methods to optimize expressions in compilers. One is to reduce the number of execution count of an expression. The other is to transform expensive expressions to less expensive ones. In this paper, we propose a new algorithm of integrating some expressions by utilizing architecture characteristics with reducing the number of execution count of each expression. This algorithm can optimize both expressions with a dependency and expressions without a dependency. Moreover, this algorithm can be applied to many languages and architectures. We implemented two optimizations using this algorithm in the IBM Java Just-in-Time (JIT) compiler on IA-64, and evaluated them by measuring the SPECjvm98 benchmark suite.

1. はじめに

コンパイラ上で式を最適化する方法は，大きく分けると 2 つのカテゴリに分類される．1 つは partial redundancy elimination（PRE）に代表されるような，式の実行回数を減らす方法．もう 1 つは，式の演算強度（実行コスト）を軽減させる方法である．

演算強度軽減（strength reduction）は演算強度が重い式をより演算強度が軽い式へと変形するコンパイラ上の最適化手法の 1 つである[19]．従来の式の演算強度を軽減させる方法としては，単純に 1 つの式をより軽い式にマッピングする方法と，複数の式についてより演算強度が軽い式へ変形させる方法がある．

従来，複数の式についての演算強度軽減は「依存関係☆がある式」については多くの研究がなされてきた．特に，ループの誘導変数に関連する掛け算式の演算強度軽減についての研究は多くのバリエーションがある．一方，「依存関係を持たない複数の式」についても合成することにより，さらに演算強度軽減の機会を増やすことができる．たとえば，IA-64 等は浮動小数点に関してペア・ロード命令を持っており，これを利用すると隣接する 2 つの領域から一度に 2 つのレジスタにロードを行うことができる[11]．従来，依存関係を持たない複数の式について合成を行うアルゴリズムは，ループの誘導変数（induction variable）の性質に特化したものしかなかった[1),5),18),20)]．しかし，プログラム内にはループの誘導変数とは無関係な式は数多く存在し，これらの式の演算強度を軽減することはパフォーマンスの向上に貢献する．

本稿では，プログラム内の一般的な式について実行回数を減らしつつ，アーキテクチャの性質を利用して複数の式をより演算強度が軽い式へ合成させる新しいアルゴリズムを提案する．本稿のアルゴリズムは，依存関係を持つ式だけではなく，依存関係がない複数の式も合成できる点が大きな特徴である．さらには，一

† 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.

☆ 本稿では，true（flow）dependence を指す

部のパス上で合成可能な複数の式をコード移動により演算強度軽減することも可能である．

我々は，コード移動の基本アルゴリズムとして，PRE の一手法である Knoop らの Lazy Code Motion（LCM）[16]を用いる．このアルゴリズムに「合成することにより効果がある式の集合」という概念を組み込み，**合成しようとする式ができるだけまとまるようにコード移動を行う．そして，まとまった式について合成を行う．さらに我々の手法は，LCM と同様に bit vector を使ったアルゴリズムのため，高速に問題を解くことができる．

我々は IA-64 上の IBM の Java Just-in-Time（JIT）コンパイラに対して，このアルゴリズムを使った 2 種類の最適化の実装を行い，SPECjvm98 のベンチマークを用いて評価を行った．その結果，わずかなコンパイル時間の増加でパフォーマンスの向上を達成することができた．

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

Partial redundancy elimination（PRE）は実行方向とは逆向きに，計算式を大域的にベーシックブロックの境界を越えて移動させる（global code motion）ことによって，部分的な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[8),10),16),22)]．これらの技術は，それぞれの式を元の場所より実行頻度がより低い場所に移動させることを目的としている．

演算強度軽減の最も簡単な方法としては，1 つの式をより軽い式にマッピングする方法がある．たとえば，掛け算をシフトと足し算の組合せで表現できるようにする方法がある[2),3)]．これは “i × 5” という式を “(i << 2) + i” のように変形する．また，分母が定数の割り算を掛け算に置き換える方法も提案されている[7)]．

もっと強力な方法としては，複数の式について演算強度軽減を行う方法がある．**表 1** は従来の複数式に対する手法と我々の手法の比較を表にしたものである．従来，複数の式についての演算強度軽減は「依存関係がある式」については多くの研究がなされてきた．依存関係を持つ式を対象にした最適化の中でも，特にループの誘導変数に関連する掛け算式の演算強度軽減についての研究は古くからさかんに行われている[4),9),14),15),19)]．この中には，ループの最適化の一環としてループの誘導変数に特化した最適化を行うもの[4),19)]（表 1 (1)）や PRE と演算強度軽減を組み合わせた手法がある．

表 1 従来の複数式に対する手法との比較
Table 1 Comparisons to previous approaches optimizing some expressions.

	大域的なコード移動を行う	一部のパスに沿って式を合成可能	依存関係がある式を対象	依存関係がない式を対象
(1)	○	×	○	×
(2)	○	○	○	×
(3)	×	×	×	○
(4)	**○**	**○**	**○**	**○**

(1) ループの誘導変数に関連する掛け算を足し算に変換.
(2) 従来の部分冗長性除去（PRE）と演算強度軽減の組合せ.
(3) ループ内の配列アクセスについて，ループのアンローリングを行い，iteration 間のメモリアクセスを合成.
(4) 我々の手法.

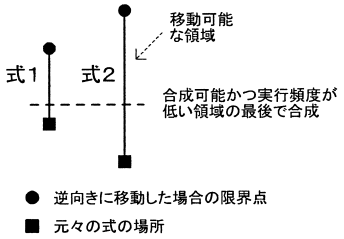


図 1 依存関係のない複数式の演算強度軽減
Fig. 1 Strength reduction for non-dependence expressions.

従来の PRE と演算強度軽減を組み合わせた手法（表 1 (2)）は，すべて「依存関係がある式」を対象にした最適化だった[9),14),15)]．これらの手法は誘導変数に関連する掛け算式 1 つに注目し，この式を後方に移動させループ内の誘導変数の更新状況を解析することにより，掛け算式を足し算式へと変形する．これらの手法は，誘導変数に関連する掛け算式がループ内の誘導変数の更新を行う足し算式を越えて移動しないことを積極的に利用して，変形を行っている．しかし，これらの手法を使って依存関係を持たない複数の式を最適化しようとすると，合成しようとする個々の式が他の式を越えて移動してしまうため，うまく 1 カ所にまとまらないという問題がある．

我々の手法は**図 1** のように，合成可能かつ実行頻度が低い領域の最後の場所に最適化対象の式がまとまるように移動し合成を行うことにより，この問題を解決している．そのため，依存関係がある式だけではなく，依存関係がない複数の式を合成できる点がこれらの手法と大きく異なる．

「依存関係がない複数の式」について合成を行うアルゴリズムは，ループ内の配列アクセスについて，ルー

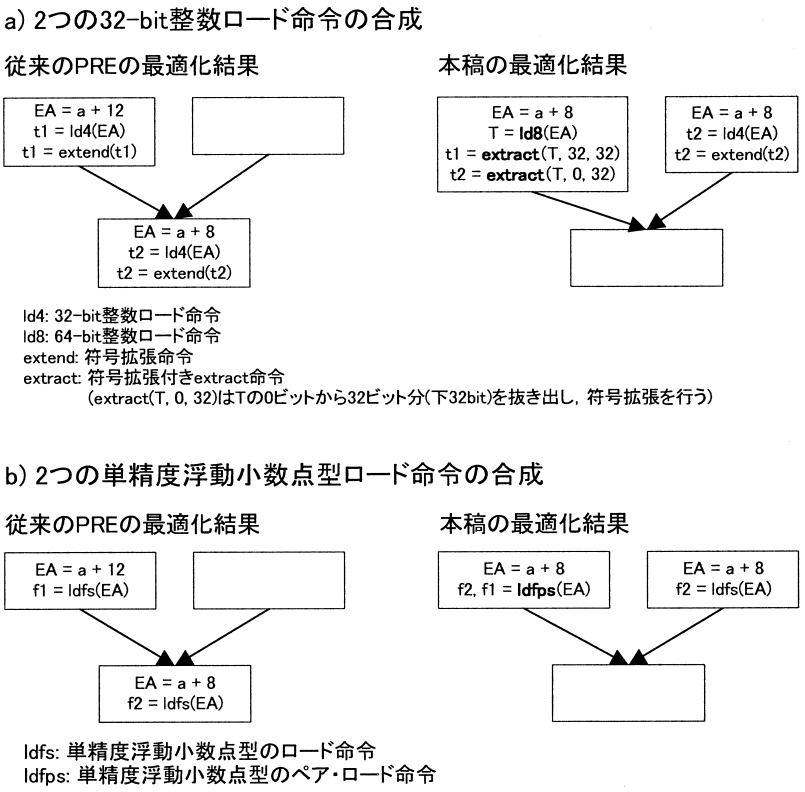


図 2 IA-64 向けに本稿の最適化を適用した例
Fig. 2 Two examples of our optimization for IA-64.

プのアンローリングを行い，iteration 間のメモリアクセスを合成する方法がある[1),5),18),20)]．しかし，これらの方法は大域的なコード移動（global code motion）を行っていないため，ブロック内で閉じた最適化だった（表 1 (3)）．そのため，これらの手法は適用できる対象が限定されていた．

我々の手法（表 1 (4)）は，表 1 の 4 つの特徴をすべて持っている．そのため，依存関係がある式・ない式どちらも合成可能であり，さらには 3 章で述べる**図 2** の例のように，一部のパスで合成可能な式を扱うこともできる．また，我々の手法はメモリアクセスの最適化だけにとどまらず，3.3 節に示すように，式の合成を行う多くの最適化に対して適用可能である．

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．(a) は 2 つの 32-bit 整数型ロード命令を組み合わせる例を示している．IA-64 では，隣接する 2 つの領域に対する 32-bit の整数型ロード命令を合成して，64-bit ロード命令とそれぞれの 32-bit データを符号拡張つき extract 命令で抜き出す処理に変形することができる．この結果，(a) の左のパスでは 4〜6 命令必要だった処理（符号拡張命令は後続の最適化[13)]により除去される可能性がある）を 4 命令に減らすことができる．世の中のシステムやアプリケーションの多くはまだ 32-bit アーキテクチャを前提として設計されているため，32-bit のデータ型はプログラム内に多く現れる．たとえば Java の言語仕様[6)]では，最も多く使われる int 型を 32-bit と規定している．そのため，(a) のような最適化は 64-bit アーキテクチャでは効果がある．

(b) は 2 つの浮動小数点型のロード命令を組み合わせる例を示している．IA-64 は浮動小数点に関してペア・ロード命令を持っており，これを利用すると隣接する 2 つの領域から一度に 2 つのレジスタにロードを行うことができる．これを利用して本稿の最適化を利用すると，(b) の左のパスでは 4 命令必要だった処理を 2 命令に減らすことができる．なお，ペア・ロード命令を使う際には，奇数番号と偶数番号のレジスタの組が左辺にくる必要があるが[11)]，これは制約を考慮したレジスタ割付法[17)]を適用することにより達成できる．

さらに，このような式の合成を最適化の視野に入れ

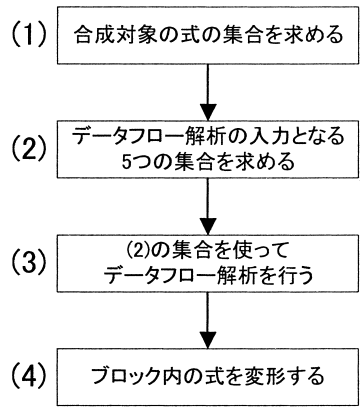


図 3 我々のアルゴリズムの流れ
Fig. 3 Flow diagram of our algorithm.

ると，部分的な合成可能性が新たに発生する．たとえば図 2 の (a)，(b) はともに，左側のパスでは合成の機会があるが，右側のパスでは合成の機会がない．そこで，我々は PRE の技術を応用して組み合わせることにより，一部のパスで合成可能な複数式の演算強度軽減を達成している．

なお，(a)，(b) の最適化を行う際にはアラインメント（alignment）に注意しなければならない．(a)，(b) の例はともに 64-bit データをロードする処理となるため，最適化対象となる 2 つのロード命令の対象アドレスはともに 64-bit でアラインされた領域に入っている必要がある．

以下，3.1 節ではアルゴリズムについて述べ，3.2 節では図 2 の例を使った具体例を示す．3.3 節ではほかの最適化への応用例を示す．

3.1 アルゴリズム

この節では，Knoop らの Lazy Code Motion（LCM）アルゴリズム[16]に，「合成することにより効果がある式の集合」という概念を組み込むための枠組みについて述べる．**図 3** に我々のアルゴリズムの流れを示す．以下では，この 4 つのステップについて，順に説明していく．

まず**最初のステップ**では，プログラム内の「合成することにより効果がある式の集合」GROUP を求める．我々は，次の 3 つの条件すべてを満たすものをこのような集合として扱っている．

- 複数の式を合成したコード片 C を定義できる．
- C からそれぞれの式の結果を作成することができる．なお，C からそれぞれの式の結果を作成するためのコードは必ずしも必要ではない．
- この変形によって，最終的な結果が良くなる可能性がある．

```
for (each n ∈ すべてのベーシックブロック){
  N-COMP_G(n) = N-COMP(n)
  for (each e ∈ N-COMP(n)){
    g = e に対応するグループ
    N-COMP_G(n) ∪= g に含まれるすべての式
  }
  X-COMP_G(n) = X-COMP(n)
  for (each e ∈ X-COMP(n)){
    g = e に対応するグループ
    X-COMP_G(n) ∪= g に含まれるすべての式
  }
}
```

図 4 $\text{N-COMP}_\text{G}(n)$，$\text{X-COMP}_\text{G}(n)$ の集合を求めるアルゴリズム
Fig. 4 Algorithm for computing the sets $\text{N-COMP}_\text{G}(n)$ and $\text{X-COMP}_\text{G}(n)$.

次に各式に対応する集合 GROUP の情報をセットする．集合 GROUP は各式を bit に割り当てた bit vector で表現できる．たとえば，5 つ式があり，bit0 と bit1 に対応する式が同じグループ，bit2 と bit4 に対応する式が同じグループとする．この場合，bit0 と bit1 に対応する式に対応するグループは {11000} と表現され，bit2 と bit4 に対応する式に対応するグループは {00101}，bit3 に対応する式に対応するグループは {00000}（空集合）と表現される．

LCM アルゴリズム[16]は，TRANSP，N-COMP，X-COMP という 3 つの集合を入力として，N-INSERT，X-INSERT という集合を最終的には求める．これら 5 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

N-INSERT(n) ブロック n の先頭に挿入する式の集合

X-INSERT(n) ブロック n の最後に挿入する式の集合

2 番目のステップでは，入力となる 3 つの集合をまず求め，次に N-COMP(n)，X-COMP(n) に集合 GROUP の要素を加えた $\text{N-COMP}_\text{G}(n)$，$\text{X-COMP}_\text{G}(n)$ を**図 4** のアルゴリズムで作成する．

3 番目のステップでは，2 番目のステップで求めた TRANSP，N-COMP，X-COMP，N-COMP_G，X-COMP_G の 5 つの集合を入力として，データフロー解析を行い，式の挿入点および挿入する式が有効な範囲を求める．

(1) TRANSP(n), N-COMP(n), X-COMP(n) を入力にして Busy Code Motion アルゴリズム[16]を適用し，N-EARLIEST(n) と X-EARLIEST(n) を求める．

(2) Delayability Analysis:

$$\text{N–DELAYED(n)} = \text{N–EARLIEST(n)} \cup \bigcap_{\text{m}\in\text{Pred(n)}} (\overline{\mathbf{X\text{–}COMP_G(m)}} \cap \text{X–DELAYED(m)})$$

$$\text{X–DELAYED(n)} = \text{X–EARLIEST(n)} \cup \text{N–DELAYED(n)} \cap \overline{\mathbf{N\text{–}COMP_G(n)}}$$

(3) Computation of Latestness:

$$\text{N–LATEST(n)} = \text{N–DELAYED(n)} \cap \mathbf{N\text{–}COMP_G(n)}$$

$$\text{X–LATEST(n)} = \text{X–DELAYED(n)} \cap (\mathbf{X\text{–}COMP_G(n)} \cup \bigcup_{\text{m}\in\text{Succ(n)}} \overline{\text{N–DELAYED(m)}})$$

(4) Isolation Analysis:

$$\text{N–ISOLATED(n)} = \text{X–EARLIEST(n)} \cup \text{X–ISOLATED(n)}$$

$$\text{X–ISOLATED(n)} = \bigcap_{\text{m}\in\text{Succ(n)}} (\text{N–EARLIEST(m)} \cup \overline{\mathbf{N\text{–}COMP_G(m)}} \cap \text{N–ISOLATED(m)})$$

(5) Insertion Point:

$$\text{X–INSERT(n)} = (\text{N–LATEST(n)} \cap \overline{\text{N–ISOLATED(n)}}) \cup (\mathbf{X\text{–}LATEST(n)} \cap \overline{\mathbf{X\text{–}ISOLATED(n)}} \cap \mathbf{TRANSP(n)})$$

(6) Reachability Analysis:

$$\text{N–REACH(n)} = \bigcap_{\text{m}\in\text{Pred(n)}} \text{X–REACH(m)}$$

$$\text{X–REACH(n)} = \text{X–INSERT(n)} \cup (\text{N–REACH(n)} \cap \text{TRANSP(n)})$$

図 5 LCM アルゴリズムの変更点（Bold 体が変更または追加した点）
Fig. 5 Our modifications of the LCM algorithm (Bold font denotes our modification or addition).

LCM アルゴリズムは，各式を実行とは逆向きにできるだけ遠くへ移動させる Busy Code Motion (BCM)[16]パート，次にレジスタプレッシャや不必要な移動を減らすために実行方向へ移動させる Lazy パートの 2 つに分かれる．BCM パートについてはそのまま適用しているため，各式の実行回数に関する最適化性能は BCM と同等である．我々の変更点は，式を実行方向へ移動させる Lazy パートで，グループ内に含まれる式のどれかが現れた点で移動が止まるようにした点である．これによって，グループ内の式ができるだけまとまるような移動を行うことができる．

図 5 に LCM アルゴリズムに対する詳細な変更点を示す．**Bold 体**で書かれている部分が変更もしくは追加した部分である．図 5 (1) の BCM によって，N-EARLIEST，X-EARLIEST の 2 つの集合が求まる．これらの集合は，式を実行方向とは逆向きにできるだけ遠くに移動させた場合の式の挿入点を意味する．

図 5 (2)〜(4) 内の処理で LCM では元々N-COMP(n) と X-COMP(n) を使っていた部分を N-COMP_G(n) と X-COMP_G(n) に変えたことにより，グループ内に含まれる式のどれかが現れた点で移動が止まるようになる．まず，(2) では N-EARLIEST，X-EARLIEST に含まれる式を実行方向に移動させ，実行頻度を増やさない領域かつグループ内の式が最初に現れるまでの領域 DELAYED を求める．N-DELAYED(n)，X-DELAYED(n) は，ブロック n における，領域 DELAYED に含まれる式の集合を意味する．

次に，(3) では領域 DELAYED の中で一番最後の場所である LATEST を求める．N-LATEST(n)，X-LATEST(n) は，ブロック n における，LATEST に含まれる式の集合を意味する．LATEST に含まれる式が移動先の候補となる．

しかし，単に LATEST の場所に移動を行うと，実行回数（コスト）が変わらないにもかかわらず，元の式の位置よりも前に移動する場合がある．このような移動は，レジスタプレッシャを高めてしまうため，実行回数が変わらない場合には移動を行わないほうが望ましい．そのため，LCM は元々式の冗長性を解析している．たとえば，あるパス上に同じ式が 2 つ以上あれば，下にある式は上の式の場所まで冗長性があることになる．(4) では，このような式の冗長性が**ない**領域 ISOLATED を求める．N-ISOLATED(n)，X-ISOLATED(n) は，

ブロック n における，領域 ISOLATED に含まれる式の集合を意味している．

元々の LCM アルゴリズムは，まずブロック内の冗長性を取り除き，この状態を出発点として，次にデータフロー解析を行ってブロック間の冗長性を解消する．すなわち，2 回コードの変形を行う必要がある．個々の式を独立に最適化する場合には，この方法でもかまわないが，複数の式を関連付けて最適化する場合には，コードの変形を最後にまとめて行うやり方が効率が良い．そのための処理として図 5 (5) を変更し，(6) で「ブロック n に X-INSERT(n) 内の式を挿入したことによって，これ以降で冗長であると判断できる領域 REACH」を求める．これによって，X-INSERT(n) 内の式の結果を挿入場所でテンポラリ変数にキャッシュしておけば，領域 REACH 内の式はそのテンポラリ変数で置き換えることができる．N-REACH(n)，X-REACH(n) は，ブロック n における，領域 REACH に含まれる式の集合を意味している．

4 番目のステップでは，3 番目のステップで求まった X-INSERT(n) と N-REACH(n) を用いて，各ブロック内の変形を行う．**図 6** にブロック n 内を変形するアルゴリズムを示す．このアルゴリズムは大きく分けると 2 つの部分に分かれる．最初の部分ではブロック n 内の命令を実行方向にスキャンし，集合 N-REACH(n) を元にブロック n 内のローカルな最適化を行う．次の部分はブロック n 内のスキャン後に，集合 X-INSERT(n) に含まれる式の挿入を行う．なお，図 6 で集合 inner は，テンポラリ変数 T[ ] に置き換えることができる式（言い換えると冗長な式）の集合を意味している．またアルゴリズムの説明では，この配列を使う際に “T[それぞれの式]” のようにインデックスを式で表している．実際の実装では，各式に対応するビットの場所等を配列のインデックスとして使えばこれは実現可能である．また，テンポラリ変数の配列 T[ ] の要素は，図 6 のアルゴリズムを実行する前に変数番号で初期化されているものと仮定している．

3.2 具 体 例

図 2 の (a)，(b) の例に最初のステップの条件をあてはめると，**図 7** に示すようになる．なお，図 7 の「(a) の例：」内の「C から ld4(...) の結果を作成するためのコード片」でともに，符号拡張命令（extend）を生成するようにしている．直前の extract 命令の結果は必ず符号拡張された状態になるため，これらの符号拡張命令は明らかに冗長である．冗長な命令をわざわざ生成している理由は，後続する符号拡張命令を効果的に最適化するためである．詳しくは**図 9** で説明する．

```
inner = N-REACH(n);
for (each I ∈ ブロック n 内の先頭から最後までの命令){
  R = I の右辺式;
  for (each e ∈ 最適化対象となっているすべての式)
    if (R は e の移動を妨げる) inner -= e;
  if (R は最適化対象) {
    if (R ∈ inner){
      if (ブロック内の I 以前に R と同一の式が存在する &&
          R の結果は変数 T[R] に代入されていない)
        その場所に R を変数 T[R] に代入するコードを生成;
      R をテンポラリ変数 T[R] で置き換える;
    } else {
      g = R に対応する GROUP;
      if (ブロック内の I 以前に R 以外の g に
                含まれる式が存在する &&
          R はその場所に移動可能){
        その場所に「g に含まれる式を合成した式 C」を挿入;
        C の後に「C の結果から g に含まれる式の結果をテンポ
            ラリ変数 T[それぞれの式] に代入するコード」を生成;
        inner ∪= g に含まれる式;
        R をテンポラリ変数 T[R] で置き換える;
      }
    }
    inner ∪= R;
  }
  for (each e ∈ 最適化対象となっているすべての式)
    if (I の変数定義は e の移動を妨げる) inner -= e;
}

// X-INSERT(n) に含まれる式を挿入
insert = X-INSERT(n);
for (each e ∈ insert){
  e_g = e に対応する GROUP ∩ X-INSERT(n);
  if (e_g に含まれる複数の式どうしを合成可能 &&
      e_g 内の式で inner に含まれていないものがある){
    P = ブロック n の最後;
    if (ブロック内に e_g に含まれる式が存在する) P = その場所;
    P の場所に「e_g を合成した式 C」を挿入;
    C の後に「C の結果から e_g に含まれる式の結果をテンポ
          ラリ変数 T[それぞれの式] に代入するコード」を生成;
    inner ∪= e_g に含まれる式;
    insert -= e_g に含まれる式;
  } else if (e ∉ inner)
    ブロック n の最後に e をテンポラリ変数
          T[e] に代入するコードを生成;
  else if (ブロック内に元々e と同一の式が存在する)
    その場所に e をテンポラリ変数 T[e] に代入するコードを生成;
}
```

図 6 ブロック n 内の変形アルゴリズム
Fig. 6 Our code transformation algorithm in block *n*.

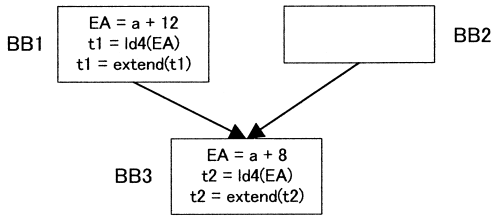
図 2 (a) の例に 2 番目・3 番目のステップを適用した状態を**図 8** に示す．図 2 (b) については，図 8 内の ld4 を ldfs と読み替えれば同様の結果となる．

2 番目のステップを適用すると集合 TRANSP，N-COMP，X-COMP，N-COMP$_G$，X-COMP$_G$は図 8 の STEP2 で示すように求まる．なお，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

(a) の例：
- 式 ld4(a+8) と ld4(a+12) を合成したコード片 C: { EA=a+8; T=ld8(EA); }
- C から ld4(a+12) の結果を作成するためのコード片: { T1=extract(T, 32, 32); T1=extend(T1); }
- C から ld4(a+8) の結果を作成するためのコード片: { T2=extract(T, 0, 32); T2=extend(T2); }
- この変形によって，後続する符号拡張命令（extend）を除去できる可能性がある．

(b) の例：
- ldfs(a+8) と ldfs(a+12) を合成したコード片 C: { EA=a+8; T2,T1=ldfps(EA); }
- C から ldfs(a+8) の結果を作成するためのコード片: { φ }
- C から ldfs(a+12) の結果を作成するためのコード片: { φ }
- この変形によって，4 命令が 2 命令に減る．

図 7 図 2 (a), (b) の例に対応する最初のステップの条件
Fig. 7 Conditions of the first step for Fig. 2 (a), (b).



bit0: ld4(a + 8) 対応するグループ {11}
bit1: ld4(a + 12) 対応するグループ {11}

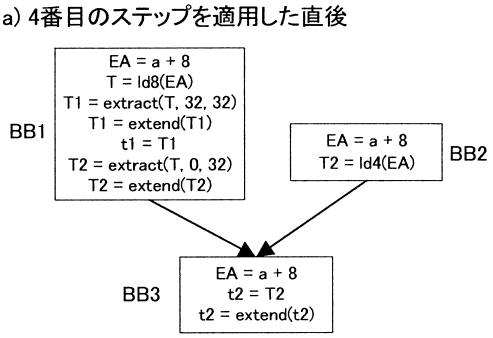
STEP2	TRANSP	N-COMP	X-COMP	N-COMP$_G$	X-COMP$_G$
BB1:	00	01	01	11	11
BB2:	00	00	00	00	00
BB3:	00	10	10	11	11

STEP3	X-INSERT	N-REACH
BB1:	11	00
BB2:	10	00
BB3:	00	10

図 8 図 2 (a) に 2・3 番目のステップを適用した状態
Fig. 8 Applying both the second and third steps to Fig. 2 (a).

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．文献 22) には，ロード命令について入力となる 3 つの集合（TRANSP，N-COMP，X-COMP）の求め方が記述されている．詳細については，文献 22) を参照していただきたい．

次にこれら 5 つの集合を元に 3 番目のステップとして図 5 のアルゴリズムを適用すると，図 8 の STEP3 で示すように X-INSERT，N-REACH の集合が求まる．なお，図 5 内のアルゴリズムで N-COMP$_G$，X-COMP$_G$ を使わずに，仮に従来のように N-COMP，X-COMP を使うと，X-INSERT，N-REACH の計算結果はすべてのブロックで {00} となる．



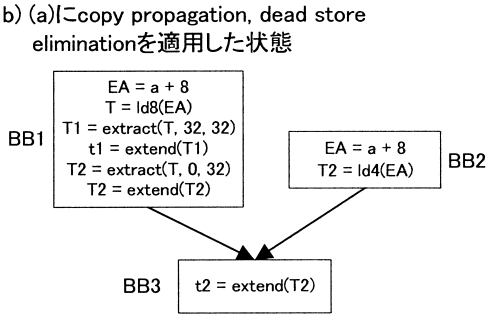


図 9 図 8 に 4 番目のステップを適用した状態
Fig. 9 Applying the fourth step to Fig. 8.

図 8 の結果に 4 番目のステップを適用した直後の状態を，図 9 (a) に示す．図 6 のアルゴリズムを適用する際には，図 7 で示したコード片を用いる．図 9 (a) が本稿の最適化が終わった直後の出力コードとなる．

次に，図 9 (a) に対して copy propagation，dead store elimination を適用した状態を (b) に示す．図 7 の説明で，後続する符号拡張命令を効果的に最適化するために冗長な符号拡張命令をわざわざ生成していると述べた．この例では，BB1 に冗長な符号拡張命令を生成したおかげで，BB3 にある “extend(T2)” が部分的に冗長となっている．そこで，図 9 (b) にもう一度 PRE を適用すると，BB3 内にある extend(T2) が BB2 に移動する．さらに，符号拡張命令の除去[13)]，copy propagation，dead store elimination 等の従来の最適化を適用すると，最終的には図 2 (a) の最適化結果のように変形される．図 2 (b) についても，図 7 で示したコード片を用いれば，(a) の例と同様に最適化が行える．

我々の手法は PRE アルゴリズムを基本としているため，演算強度軽減と同時に実行回数を減らすことも可能である．**図 10** にループ外への式の移動と図 2 (b)

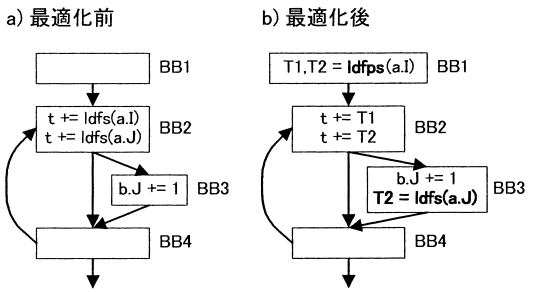


図 10 ループ外への式の移動と演算強度軽減
Fig. 10 Moving expressions out of a loop and strength reduction.

の合成を同時に行う例を示す．この例で，2 つのメモリ領域 a.I と a.J が隣接し，a.J は BB3 内の b.J とエイリアスされている可能性があると仮定している．この場合，a.I はループ不変だが，a.J は BB3 内に b.J に対して書き込みがあるため，一般的にはループバージョニング等の処理を行わなければ，ループ不変と判断できない．しかしこのような場合でも，我々の手法は (b) に示すように BB3 に a.J に対する補償コードを生成することで，BB2 内の 2 つのメモリロードをループ外の BB1 に移動しつつ合成を行うことができる．

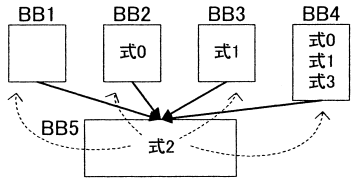
3.3 その他の最適化への応用

前節では，IA-64 アーキテクチャを例にとり，本稿のアルゴリズムを使った最適化例を 2 つ述べた．この節では，その他の最適化への応用について述べる．

図 2 (a) の例を応用すると，32-bit アーキテクチャに対しても 8-bit や 16-bit の複数のロード命令を合成し，32-bit のロード命令と，ロードした結果から 8-bit や 16-bit 単位で抜き出すコードを生成するような最適化を行うことができる．たとえば，IBM のネットワークプロセッサ PowerNP は 32-bit レジスタを 8-bit や 16-bit 単位でアクセスすることが可能である．この性質を利用すれば，32-bit でロードした結果から 8-bit や 16-bit 単位で抜き出すコードは，copy propagation を行うことにより，最終的には多くの場合明示的なコピー命令を生成せずに行うことができる．

図 2 (b) の例を応用すると，S390 や PowerPC の Load Multiple 命令を利用することによって，入力コード内に含まれる複数の整数ロード命令を合成することができる．

以下の項では，3.1 節で述べたアルゴリズムに，追加処理を行うことによって達成できるいくつかの最適化について述べる．3.3.1 項では 1 つの式が複数のグループに含まれる場合の最適化について述べる．



式0,1はグループ化可能（GROUP1）
式1,2はグループ化可能（GROUP2）
式2,3はグループ化可能（GROUP3）

(a) 必ず1つのグループに属するように設定

	対応するグループ
式0: {1100}	(GROUP1)
式1: {1100}	(GROUP1)
式2: {0011}	(GROUP3)
式3: {0011}	(GROUP3)

(b) 複数のグループを設定

	対応するグループ
式0: {1100}	(GROUP1)
式1: {1100}, {0110}	(GROUP1,2)
式2: {0110}, {0011}	(GROUP2,3)
式3: {0011}	(GROUP3)

図 11 1 つの式が複数のグループに含まれる場合の最適化
Fig. 11 Example where one expression is included by some groups.

3.3.2 項では依存関係を持つ式の最適化について述べる．3.3.3 項ではループ変形との組合せによる最適化について述べる．

3.3.1 1 つの式が複数のグループに含まれる場合

この項では，1 つの式が複数のグループに含まれる場合の対処方法について述べる．図 2 の 2 つの例は 64-bit のアラインメントの制約があるため，1 つの式が複数のグループに含まれることはなかった．しかし，一般的には 1 つの式が複数のグループに含まれる場合がある．

たとえば，**図 11** の例で式 0，1，式 1，2，式 2，3 がグループ化可能であるとする．最も簡単な解決方法は，図 11 (a) のように 1 つの式は必ず 1 つのグループだけ対応する（この例では，GROUP2 を除外する）ように設定する方法が考えられる．このようにすれば，図 11 の BB5 内の式 2 は BB1〜4 に移動し，BB4 内で式 0，1 と式 2，3 について合成が行われる．しかし，この方法は式 1，2 をグループ化する機会を逃してしまう結果となり，最適化の効果が少なくなる．この例では，BB3，BB4 内の式 1 と式 2 を合成する機会が失われる．

そこで，図 11 (b) のように 1 つの式に対して複数のグループを対応付けられるようにする．このように設定した場合にも，(a) と同様に BB5 内の式 2 は BB1〜

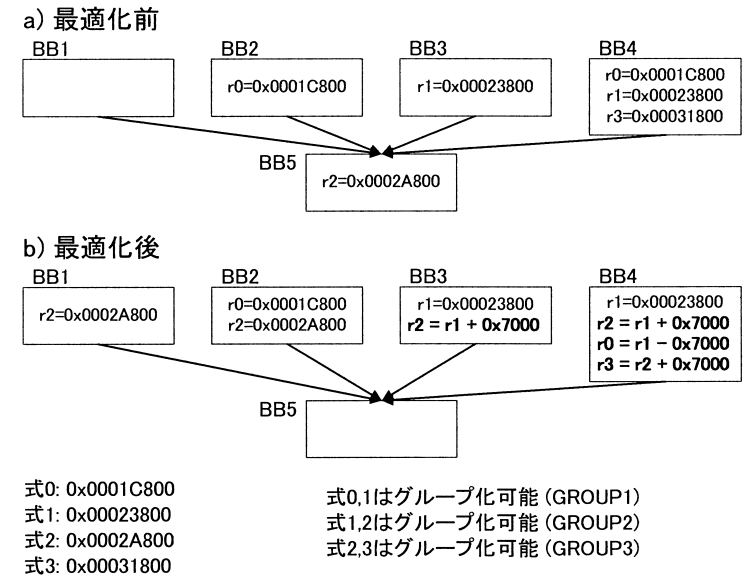


図 12 PowerPC 向けの定数ロードの最適化
Fig. 12 Constant load optimization for PowerPC.

4 に移動する．この場合の問題点は，重複している式を含む複数のグループが 1 つのブロック内で同時に変形対象となったときに，重複している式をどのように扱うかという点である．図 11 の例では BB4 内に変形対象となっているグループが 3 つ（GROUP1, GROUP2, GROUP3）あり，重複している式 1 と式 2 をどうするかという問題となる．

この解決方法としては，重複している式を含むグループに効果が大きい順に優先順位をつけ，優先度の高いグループから変形を行い，優先度の低いグループについては，前の変形結果を考慮して変形を行う・行わないを決定するという方法が考えられる．たとえば，図 11 の例で GROUP2，GROUP1，GROUP3 の順で優先度が高いと仮定する．その場合には，GROUP2 の変形をまず行い，GROUP1 については重複していない式 0 について GROUP2 の変形結果も考慮して合成可能であれば合成し，合成できなければ式 0 のまま残す．その次に GROUP3 の変形を試み，GROUP2 の変形結果後の式 2 の状態と式 3 を考慮し合成可能であれば合成し，合成できなければ式 3 のまま残すということになる．

図 12 は図 11 に対応する具体的な例を示している．この例は，PowerPC 向けの 32-bit 定数ロードの最適化を示している．一部の値を除いて，PowerPC 上では 32-bit の定数ロードは，上下 16-bit ずつ値をセットするため 2 命令必要となる．しかし，演算命令を利用することで，これを 1 命令に減らすことが可能となる．

図 12 の例は BB4 内を GROUP2，GROUP1，GROUP3 の順で変形した場合の結果を示している．この場合には，GROUP2 内の式 1，2 の合成をまず行う．次に，GROUP1 の重複していない式 0 について考えてみると，式 1 の結果から再計算可能なため，式 0 の変形を行う．次に，GROUP3 の重複していない式 3 について考えてみると，変形後の式 2 の結果からも再計算可能なため，式 3 の変形を行う．もし仮に，式 0 や式 3 が合成できなければ，変形せずにそのまま残すことになる．

3.3.2 依存関係を持つ式の最適化

本稿は，互いに依存関係のない式の合成が可能なことを特徴としているが，依存関係を持つ 2 つの式を最適化することも可能である．**図 13** は PowerPC 向けに依存関係を持つ式を最適化した例を示している．左ローテート・マスク命令を使うことにより，従来 (a) のように左のパスでは 2 命令必要だったものが，t1 が dead store となっていれば，(b) のように左のパスでは 1 命令に減らすことができる．

依存関係を持つ 2 つの式を合成する場合には，式の順序を考慮する必要がある．ここで，合成する 2 つの式のうち，先行する必要がある命令を first，後続する必要のある命令を second と呼ぶことにする．図 13 の例では，first は「t1 = a >> 16」，second は「t2 =

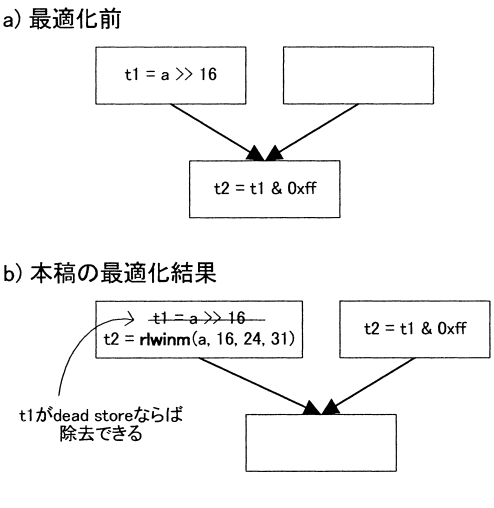


図 13 依存関係を持つ式への適用例
Fig. 13 Example for expressions that have dependency.

t1 & 0xff」となる．式の順序を考慮しつつ合成を行うためには，second→first という順番のときに間違って最適化することを防ぐ必要がある．そのためには，first の直前に，first の移動を妨げる命令があると仮定して，本稿で述べた最適化を行うことにより，この問題は回避できる．

3.3.3 ループ変形との組合せによる最適化

ループアンローリング・ループバージョニングといったループの変形手法と本稿を組み合わせて利用することで，ループ内の配列アクセスについて，iteration 間のメモリアクセスを合成することが可能となる．

文献 5) には，iteration 間のメモリアクセスを合成するためのループの変形方法が書かれてある．この変形は**図 14** のように，まず最初に 1 つのループをアラインメントとエイリアスチェックの成否によって，2 つのバージョンに分岐するように変形する．次に，アラインメント等のチェックが成功するバージョンでは，ループアンローリングを行う．

このように変形を行っておけば，アンロールしたループボディについて，ループ内の連続した配列アクセスをグループ化し本稿の処理を行えば，従来の iteration 間のメモリアクセスを合成する最適化と同様の結果を得ることができる．

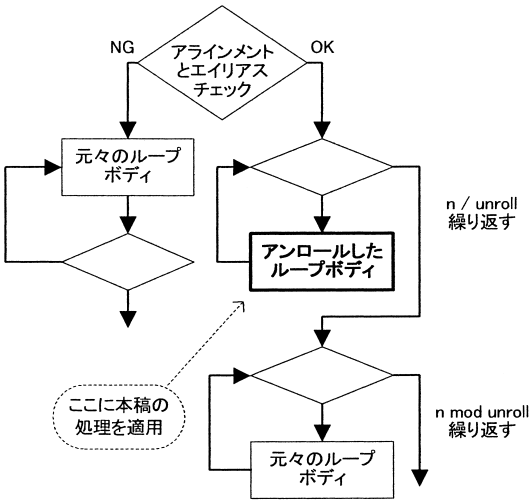


図 14 ループ変形との組合せ
Fig. 14 A combination of loop transformations.

4. 実験結果

我々は SPECjvm98 [21] 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．すべての測定結果は，IA-64 向けの IBM Java JIT Compiler に本稿で述べた図 2 の 2 つの最適化を実装し，IBM IntelliStation Z Pro model 689412X（Itanium 800 MHz 2 個，2 GB RAM）上で測定した．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．SPECjvm98 の測定方法は問題の大きさを 100 にして実行させた．

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．なお，我々の JIT Compiler ではコードサイズやコンパイル時間を大きく増やす図 14 で述べたループの変形は行っていない．そのため，従来の iteration 間のメモリアクセスを合成するような最適化はどちらの版でも行っていない．

Baseline コード移動を行わずに，図 2 の 2 つの最適化を行う版．通常の PRE を使った最適化[10),16)]，ヌルチェックの最適化[12)]，投機的なメモリアクセスの最適化[22)]，符号拡張命令の除去処理[13)]等の別の最適化は有効にしている．

Our approach Baseline に加えて，コード移動を行って図 2 の 2 つの最適化を行う版．

図 15 に Baseline と比較して，SPECjvm98 上で達成できた速度向上率を示す．我々の手法は幾何平均で 1.77%程度，SPECjvm98 の速度向上を達成することができた．また，compress と mpegaudio の 2 つのベ

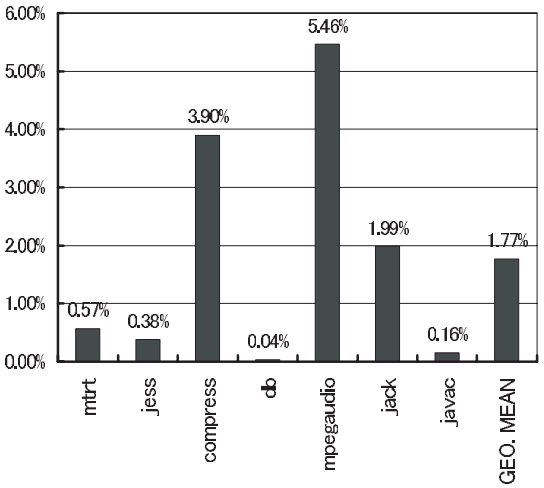


図 15 SPECjvm98 のパフォーマンス向上率
Fig. 15 Performance improvement for SPECjvm98.

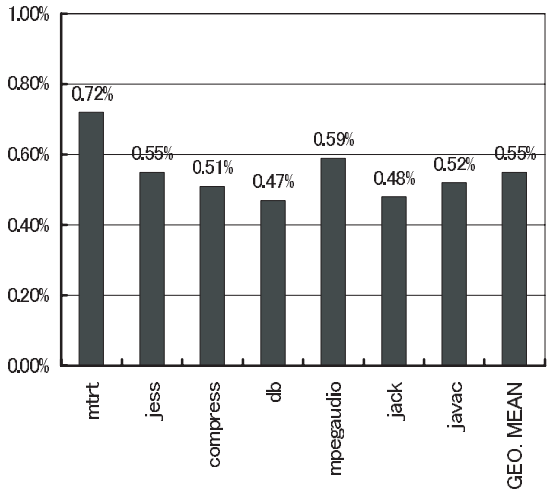


図 16 我々の手法に対するコンパイル時間の比率
Fig. 16 Ratio of JIT compilation time of our approach.

ンチマークで大きな効果が見られた．これらのベンチマークを調べた結果，compress ベンチマークについては図 2 (a) の整数ロード命令の合成，mpegaudio については図 2 (b) の浮動小数点ロード命令の合成が効果があることが分かった．

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，IA-64 上で利用可能なトレースツールを用いて，JIT のコンパイル時間の内訳を測定した．SPECjvm98 について測定した，我々の手法に対するコンパイル時間の比率を**図 16** に示す．本稿で述べた我々の手法は，JIT のコンパイル時間のうち，幾何平均で 0.55%程度占めていることが分かった．我々の手法に対するコンパイル時間の比率は 0.47%〜0.72%とベンチマークにかかわらず，ばらつきが少なかった．

5. 終わりに

本稿では，実行回数を減らしつつ，複数の式をより演算強度が軽い式へ合成させる新しいアルゴリズムについて述べた．我々の手法は，PRE の 1 手法である Lazy Code Motion アルゴリズム[16]に，「合成することにより効果がある式の集合」という概念を組み込み，合成しようとする式ができるだけまとまるようにコード移動を行い，まとまった式について合成を行う．これによって，依存関係を持つ式だけではなく，依存関係がない複数の式も合成できるようになった．また，本アルゴリズムを使って式の合成を行う具体例として，2 つの最適化を実装し評価を行った．この結果，わずかなコンパイル時間の増加で大きくパフォーマンスを向上させることができた．本手法は，多くのアーキテクチャや言語に対して，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) Alexander, M., Bailey, M., Childers, B., Davidson, J.W. and Jinturkar, S.: Memory Bandwidth Optimizations, *Proc. 26th Annual Hawaii International Conference on System Sciences*, pp.466–475 (1993).
2) Bernstein, R.: Multiplication by integer constants, *Software—Practice and Experience*, Vol.16, No.7, pp.641–652 (1986).
3) Briggs, P. and Harvey, T.: *Multiplication by integer constants* (1994).
4) Cooper, K.D., Simpson, L.T. and Vick, C.A.: Operator strength reduction, *TOPLAS'01*, Vol.23, No.5, pp.603–625 (2001).
5) Davidson, J.W. and Jinturkar, S.: Memory Access Coalescing: A technique for Eliminating Redundant memory Accesses, *PLDI'94*, pp.186–195 (1994).
6) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley Publishing Co., Reading (1996).
7) Granlund, T. and Montgomery, P.L.: Division by invariant integers using multiplication, *PLDI'94*, pp.61–72 (1994).

8) Gupta, R., Berson, D. and Fang, J.: Path Profile Guided Partial Redundancy Elimination Using Speculation, *IEEE Conference on Computer Languages* (1998).
9) Hailperin, M.: Cost-optimal code motion, *TOPLAS'98*, Vol.20, No.6, pp.1297–1322 (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) Intel Corp.: Itanium Architecture—Manuals. http://www.intel.com/design/itanium/manuals.htm
12) Kawahito, M., Komatsu, H. and Nakatani, T.: Effective Null Pointer Check Elimination Utilizing Hardware Trap, *ASPLOS'00*, Cambridge, MA, pp.139–149 (2000).
13) Kawahito, M., Komatsu, H. and Nakatani, T.: Effective Sign Extension Elimination, *PLDI'02*, Berlin, Germany, pp.187–198 (2002).
14) Kennedy, R., Chow, F.C., Dahl, P., Liu, S.-M., Lo, R. and Streich, M.: Strength Reduction via SSAPRE, *Computational Complexity*, pp.144–158 (1998).
15) Knoop, J., Rüthing, O. and Steffen, B.: Lazy Strength Reduction, *International Journal of Programming Languages*, Vol.1, No.1, pp.71–91, Chapman & Hall, London (UK) (1993).
16) Knoop, J., Rüthing, O. and Steffen, B.: Optimal code motion: Theory and practice, *TOPLAS'94*, Vol.16, No.4, pp.1117–1155 (1994).
17) Koseki, A., Komatsu, H. and Nakatani, T.: Preference-directed graph coloring, *PLDI'02*, pp.33–44 (2002).
18) Larsen, S. and Amarasinghe, S.: Exploiting Superword Level Parallelism with Multimedia Instruction Sets, *PLDI'00*, pp.145–156 (2000).
19) Muchnick, S.S.: *Advanced compiler design and implementation*, Morgan Kaufmann Publishers, Inc. (1997).
20) Shin, J., Chame, J. and Hall, M.: Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures, *PACT'02*, pp.45–55 (2002).
21) Standard Performance Evaluation Corp.: SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/
22) 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).

(平成 15 年 2 月 18 日受付)
(平成 15 年 7 月 9 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．