Vol. 47 No. SIG 2(PRO 28) 情報処理学会論文誌：プログラミング Feb. 2006

ループ内条件分岐排除に関する新方式の提案と評価

塩足拓也[†] 畑邉誠和[††]
木山真人[†††] 梅野英典[†††]

コンパイラにおける最適化は，計算機の性能を十分に発揮するためのコードを生成することである．この最適化に関する研究の中で，コード中で最も実行頻度が高いとされているループ文の最適化はプログラムの実行速度向上に効果的である．本論文では，ループ最適化の中で特に，ループ内で命令の流れを妨げるような条件分岐を取り除く最適化を取り扱う．ループ内の条件分岐を排除する従来方式としては，Loop Unswitching，Loop Unrolling，Loop Peeling，Index Set Splitting がある．これらの方式は，if 文の条件判定式がループ不変な変数と定数との比較の場合や，単一のループインデックス変数と定数との比較の場合にしか適用できないため，適用範囲が狭いという問題点がある．そこで，本論文では，複数のループインデックス変数を含む条件判定式に対しても適用できる新しいアルゴリズムを提案する．これにより，従来方式では適用できなかった条件分岐を排除できるようになる．また，提案方式について，実行時間に関して評価を行った．その結果，提案方式の有効性を確認できた．

New Method of Conditional Branch Elimination in Loop Statements and Its Evaluation

Takuya Shiotari,[†] Norichika Hatabe,[††] Masato Kiyama[†††]
and Hidenori Umeno[†††]

We focus on the loop optimization methods to reduce the execution times of the programs. Especially we consider the optimization methods for eliminating conditional branches in the program loops. Because those branches cause long execution times by running in every loop iteration. Conventionally, the Index Set Splitting (ISS) method is limited, that is, though splitting loop statements, it can be applied to the branch conditions where “the only one loop index variable is compared with a constant”. To enhance its applicability, we propose the new method that extends the conventional ISS and makes it possible to apply it to the branch conditions where “multiple loop index variables are compared with a constant” by using a newly proposed diagram to decide the execution blocks. Evaluating its execution time, we conclude that our extended ISS method reduces the total execution times of the programs by more than about 30%.

1. まえがき

コンパイラにおける最適化は，計算機の性能を十分に発揮するためのコードを生成することである．この最適化に関する研究の中で，コード中で最も実行頻度が高いとされているループ文の最適化はプログラムの実行速度向上に効果的である．本論文では，ループ最適化の中でも特に，ループ内で命令の流れを妨げるような条件分岐を取り除く最適化を取り扱う．

ループ内の条件分岐を取り除く従来方式として，Loop Unswitching[1),2)]，Loop Unrolling[3)]，Loop Peeling，Index Set Splitting[4)] がある．これらの従来方式は，if 文の条件判定式がループ不変な変数と定数との比較の場合や，単一のループインデックス変数と定数との比較の場合にしか適用できない．

しかし，実際のプログラムにおける条件判定式は複数のループインデックス変数を含んだ複雑な式となる場合がある．したがって，従来方式はその適用範囲が狭いという問題がある．

そこで，本論文では，複数のループインデックス変数を含む条件判定式に対しても適用できる新しいアル

† 熊本大学大学院自然科学研究科
Graduate School of Science and Technology, Kumamoto University
†† 日立製作所ソフトウェア事業部
Software Division, Hitachi, Ltd.
††† 熊本大学工学部数理情報システム工学科
Department of Computer Science, Faculty of Engineering, Kumamoto University

ゴリズムを提案する．

また，評価として，ループ内の条件分岐を排除しなかった場合と提案方式を用いて条件分岐を排除した場合の実行時間の比較を行った．その結果，提案方式の有効性を確認できた．

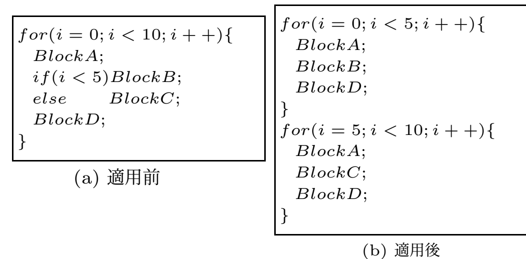
本論文の構成は，2 章でループ内の条件分岐を排除するための従来方式の説明とその問題点を示す．そして，3 章で提案方式について，アルゴリズムと実行例を示し，4 章で提案方式の評価，5 章で結論を述べる．

2. ループ内条件分岐を排除する従来方式と問題点

本章では，ループ内の条件分岐を取り除く 4 つの従来方式とその問題点について述べる．

2.1 従来方式

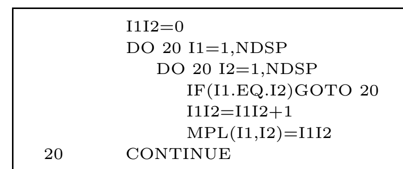
- **Loop Unswitching**
  Loop Unswitching は，ループ不変な変数や定数だけを if 文の条件判定式に含む場合に適用できる．具体的には，条件文をループの外に追い出し，条件を満たす場合と，満たさない場合のループブロックを複製し，内部の処理もそれぞれのブロックに適した処理に置き換える．
- **Loop Unrolling**
  Loop Unrolling は，ループボディを展開し，ループの繰返し回数を減らすことによって，ループのオーバヘッドを軽減させるループ最適化方式である[5]．このループを展開する方式をループ内の条件分岐を排除するために利用できる．それは，条件判定式に剰余演算を含み，その被除数がループインデックス変数で除数がループ不変な変数や定数である場合である．このとき，アンロール段数を除数もしくは除数の約数として展開することにより，条件分岐を排除できる．
- **Loop Peeling**
  Loop Peeling は，ループにおけるある特定の 1 回の繰返しをループから切り離すことによって，依存関係を取り除いたり，条件分岐を排除したりできる．
- **Index Set Splitting**
  Index Set Splitting は，ループを分割することによって，ループ内の依存関係を排除するループ最適化方式である[6)～9]．この Index Set Splitting による条件分岐の排除は，条件判定式にループインデックス変数を含む場合に，その判定式を制約としてループ文に埋め込み，ループブロック自体を分割することによって行う．たとえば，**図 1** (a)




```
for(i = 0; i < 10; i++){
  BlockA;
  if(i < 5)BlockB;
  else     BlockC;
  BlockD;
}
```

```
for(i = 0; i < 5; i++){
  BlockA;
  BlockB;
  BlockD;
}
for(i = 5; i < 10; i++){
  BlockA;
  BlockC;
  BlockD;
}
```

図 1　Index Set Splitting による条件分岐の排除（従来方式）
Fig. 1　Example of Index Set Splitting method to eliminate conditional branch (conventional method).



```
      I1I2=0
      DO 20 I1=1,NDSP
        DO 20 I2=1,NDSP
          IF(I1.EQ.I2)GOTO 20
          I1I2=I1I2+1
          MPL(I1,I2)=I1I2
20    CONTINUE
```

図 2　適用できないループ
Fig. 2　Unapplicable loop.

のように，ループ内の条件判定式がループインデックス変数と定数との比較である場合を考える．ここで，図 1 (b) のように，条件判定式をループ文の判定部分に組み込み，条件を満たすループブロック（処理 Block A，B，D を含む）と，満たさないループブロック（処理 Block A，C，D を含む）に分割する．これによって，if 文の条件分岐がなくなり，条件分岐の処理回数を軽減できる．

2.2 従来方式の問題点

従来方式には，ループ内条件分岐を排除するために，条件判定式に関して以下の制約がある．

- Loop Unswitching：ループ不変な変数や定数だけを含む場合しか適用できない．
- Loop Unrolling：剰余演算を含む場合しか適用できない．
- Loop Peeling：ループインデックス変数と定数との比較であり，かつ，“==”，“!=” といった，ある特定の場合しか適用できない．
- Index Set Splitting：単一のループインデックス変数と定数との比較の場合しか適用できない．

つまり，従来方式では，ループインデックス変数が複数含まれる場合に適用できない．

図 2 は SPECfp 95 の 103.SU2 から抜粋したコードである．図 2 に示すような複数のループインデックス変数を条件判定式に含む場合に従来方式は適用できない．そこで，条件判定式に複数のループインデックス変数を含む場合に適用できる新たな方式を提案する．

3. ループ内条件分岐排除の新方式の提案

提案方式の概要について述べる．提案方式は，ループ内に存在する if 文の条件判定式が複数のループインデックス変数を含む式と定数との比較であった場合，内側のループからループ分割を行うことにより，条件分岐を排除する方式である．提案方式と従来方式との相違点は，条件判定式に複数のループインデックス変数を含む場合に適用できる点である．

また，提案方式はループ分割を行う際，ループの実行順序を考慮して分割を行うため，ループ繰越し依存を考える必要がない．

ここで，提案方式と Index Set Splitting を比較する．一般に，Index Set Splitting ではループ内に条件式を複数包含することを許す．一方，提案方式では，ループ内に単一の if 文か，それに else 文が続く場合にしか適用できない．したがって，提案方式は単一の if-else 文に対して，Index Set Splitting を拡張したものといえる．

【適用範囲】

ここで，提案方式の適用範囲を示す．提案方式が適用可能なループを**図 3** に示す．提案方式は図 3 のようにループネストが 2 以上の場合で，そのループの最も内側に if 文を含んでいる場合に適用できる．ただし，提案方式は単一の if 文か，それに else 文が続く場合にのみ適用できる．また，ループインデックス変数 $i_t (t = 1, ..., k)$ を含む式 $f(i_1, ..., i_k)$ は以下のようにインデックス変数の一次結合の場合に対してのみ適用できる．

$$f(i_1, i_2, ..., i_k) = \sum_{t=1}^{k} a_t \cdot i_t$$

ここで，a_1，a_2，$\cdots$，a_k は定数，あるいは，ループ不変な変数とする．

```
for(i1 = l1; i1 <= u1; i1+ = stride1){
  for(i2 = l2; i2 <= u2; i2+ = stride2){
    ・ ・ ・ ・ ・
      for(ik = lk; ik <= ik; ik+ = stridek){
        if( f(i1, i2, ..., ik)   { 比較演算子 }   N )  BlockA;
        else                                         BlockB;
      }
    ・ ・ ・ ・ ・
  }
}
```

・l_t，u_t … ループ i_t の下限値，上限値
・$i_t = l_t$ … 初期化式
・$i_t <= u_t$ … 継続条件式
・$stride_t$ … ループ i_t のストライド
・$f(i_1, ..., i_k)$ … ループインデックス変数を含む式
・比較演算子 … $\{<, <=, >, >=, != , ==\}$
・N … ループ内で不変な変数，あるいは定数
・$f(i_1, ..., i_k)\{$ 比較演算子 $\}N$ … 条件判定式

図 3 提案方式が適用可能なループ
Fig. 3 General loop form.

次に，提案方式のアルゴリズムに必要な情報について述べ，アルゴリズムと実行例を示す．

3.1 提案方式のアルゴリズムに必要な情報

本節では，提案方式のアルゴリズムに必要な情報に関して議論する．

図 4 (a) は，if 文の条件分岐を含むループ文のサンプルコードである．また，図 4 (b) は図 4 (a) を模式的に表した図である．図 4 (b) は条件判定式（$i + j < 14$）を基にループインデックス変数 (i, j) の組合せによって $BlockA$ と $BlockB$ のどちらが実行されるかを表している．ここで，$BlockA$ は if 文の条件分岐において then 部の実行を，$BlockB$ は else 部の実行を意味する．このループは，インデックス変数の組合せ (i, j) が (0,0)，(0,1)，$\cdots$，(15,9)，(15,10) と変化し，$BlockA$，$BlockB$ のいずれかを実行する．この実行順序を変化させることなくループ分割を行うために，次の 2 つの要素について議論する．

- 単調増加と単調減少による違い
  ループ分割の方法は単調増加と単調減少の違いによって異なる．**図 5** はループインデックス変数 j において $a_j \cdot j$ が (a) 単調増加である場合（$a_j = 1$）と (b) 単調減少である場合（$a_j = -1$）の例を示している．図 5 (a) では $BlockA$ が先行して実行されるのに対し，図 5 (b) では $BlockB$ が先行して実行される．つまり，単調増加か単調減少かによって $BlockA$ と $BlockB$ のどちらが先に実行されるかが異なる．そのため，各ループインデックス変数 $i_t (t = 1, ..., k)$ において $a_t \cdot i_t$ が単調増加か，単調減少かという情報を取得する必要がある．

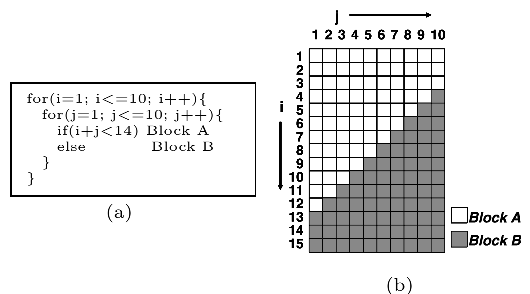


図 4 サンプルコード
Fig. 4 Sample code.

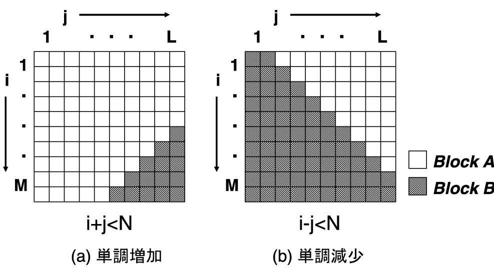


図 5　単調増加と単調減少による違い
Fig. 5　Difference by monotonic increase and decreasing.

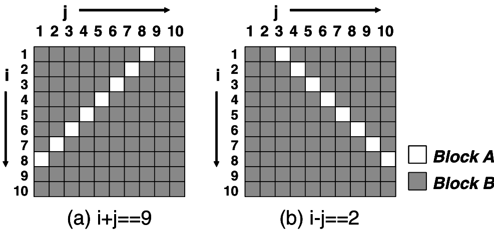


図 6　“==” の場合
Fig. 6　Case of “==”.

ただし，比較演算子が “==”，“! =” の場合は，$a_t \cdot i_t$ が単調増加か，単調減少に関係なくそれぞれ実行順序は同じになる．**図 6** はループインデックス変数 j において $a_j \cdot j$ が (a) 単調増加である場合（$a_j = 1$）と (b) 単調減少である場合（$a_j = -1$）の例を示している．また，図 6 の比較演算子は (a) も (b) も “==” としている．図 6 より，(a) も (b) も j の増加によって，$BlockB$，$BlockA$，$BlockB$ の順に実行されているのが分かる．したがって，比較演算子が “==”，“! =” の場合に限り，単調増加か，単調減少かに関係なく分割できる．

- 比較演算子による違い
  ループ分割の方法は比較演算子の違いによって異なる．**図 7** は比較演算子が (a) “<” の場合と (b) “≥” の場合の例を示している．ただし，図 7 (a) と (b) の定数 N は等しいと仮定している．比較演算子が “<” の場合，(i, j) で $BlockA$ が実行されるとすると，比較演算子が “≥” では $BlockB$ が実行される．これは，比較演算子が $BlockA$ と $BlockB$ の実行順序に関係していることを意味している．このことから，比較演算子の情報を取得する必要がある．

以上，2 つの情報を組み合わせることによってループ分割の方法が決定する．以降，単調増加・単調減少と比較演算子の組合せによって決定するループ分割をループ分割パターンと呼ぶ．このループ分割パターンは，単調増加・単調減少と比較演算子 $less$(“<”，“≤”)，$greater$(“>”，“≥”) の組合せで 4 パターン存在する．また，eq(“==”) と neq(“! =”) は単調増加・単調減少に関係なくループ分割パターンが決定するので，2 パターン存在する．したがって，ループ分割パターンは全部で 6 パターン存在する．

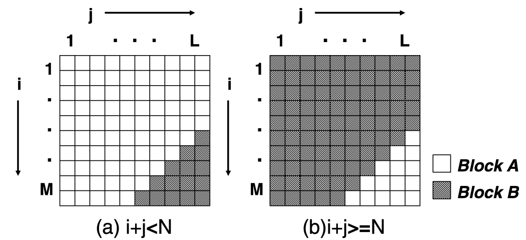


図 7　比較演算子による違い
Fig. 7　Difference by comparison operator.

ここで，アルゴリズムを単純化するために，ループ分割パターンを決定するために使用する比較演算子を “<” と “==” の 2 つにする．そのためには，次のように条件判定式と then 部，else 部の実行を変換する必要がある．

- “≤” の場合
  $f <= N \longrightarrow f < N + 1$
- “>” の場合
  $f > N \longrightarrow f < N + 1$
  then 部で $BlockB$，else 部で $BlockA$ を実行
- “≥” の場合
  $f >= N \longrightarrow f < N$
  then 部で $BlockB$，else 部で $BlockA$ を実行
- “! =” の場合
  $f! = N \longrightarrow f == N$
  then 部で $BlockB$，else 部で $BlockA$ を実行

次に，ループ分割を行うために用いる境界線について述べる．境界線とは $BlockA$ のループブロックと $BlockB$ のループブロックを生成した場合，どちらがどの範囲まで実行するかを決定するための境目を表す線である．この境界線は制約式を用いることで表現できる．制約式とはコード中で境界線の役割を果たす条件式である．制約式は if 文の条件判定式 f を用いることで生成できる．

図 8 は図 4 に境界線を引いた形である．図 8 (a) は $BlockA$ を実行する部分と $BlockB$ を実行する部分の 2 つに分かれるように境界線が引かれている．この境界線でループ文を分割すると**図 9** となる．図 9 からも分かるように，最も内側のループを分割するための制約式は if 文の条件判定式をそのまま使用できる．このように分割することで，$BlockB$ を実行するルー

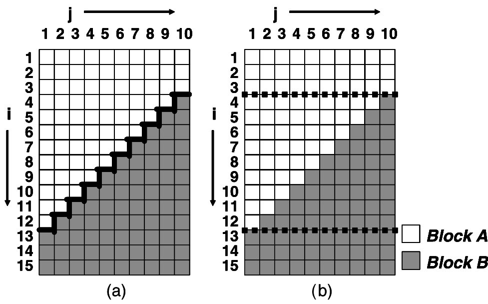


図 8 ループ分割の境界線
Fig. 8 Borderline for loop splitting.

```
for(i=1; i<=15; i++){
    for(j=1; j<=10 && i+j<14; j++){
        Block A
    }
    for( ; j<=10; j++){
        Block B
    }
}
```

図 9 最も内側のループの分割
Fig. 9 Innermost loop splitting.

プでは条件判定を行わずに実行できる．ここでの注意点は，*BlockB* を実行するループが先行する場合，制約式の比較演算子を逆向き（“<” ならば “≥”）にする点である．なぜなら，*BlockB* を実行するループは条件判定式が不成立の場合に実行されるためである．そのため，比較演算子を逆向きにした制約式を用いることで *BlockB* を実行するループを先行して実行できる．

次に，図 8 (b) は *BlockA* のみの部分，*BlockA* と *BlockB* が混在する部分，*BlockB* のみの部分の 3 つに分かれるように境界線が引かれている．この境界線でループ文を分割すると**図 10** となる．これにより，図 9 では *BlockA* を実行する場合に必ず条件判定式が実行されたが，その条件判定の回数を減少させることができる．ここで，制約式 $i+10<14$ は if 文の条件判定式の j に j の上限値 10 を代入した式である．また，$i+1<14$ は j に下限値 1 を代入した式である．したがって，制約式を生成するためには各ループインデックス変数の上限値と下限値が必要となる．

以上より，提案方式によるアルゴリズムは単調増加・単調減少と比較演算子の情報を基にループ分割パターンを決定し，上限値と下限値によって制約式を生成することによりループ分割が可能となる．

3.2 アルゴリズム

提案方式のアルゴリズムについて述べる．アルゴリズムは **1.**〜**3.** のステップからなる．**2.** と **3.** の詳細なループ分割のアルゴリズムをそれぞれ**図 11** と**図 12**，

```
for(i=1; i<=15 && i+10<14; i++){
    for(j=1; j<=10; j++){
        Block A;
    }
}
for( ; i<=15 && i+1<14; i++){
    for(j=1; j<=10 && i+j<14; j++){
        Block A
    }
    for( ; j<=10; j++){
        Block B
    }
}
for( ; i<=15; i++){
    for(j=1; j<=10; j++){
        Block B
    }
}
```

図 10 最も内側以外のループの分割
Fig. 10 Other most inner loop splitting.

図 13 に示す．

1. 処理対象ループの検索
   i) ループの検索
   ii) 条件分岐（if 文）の検索
   単一の if 文の場合か，単一の if 文に else 文が続く場合に適用できる．
   iii) 条件判定式が複数のループインデックス変数を含む式と定数との比較であるか
   iiii) f がループインデックス変数の一次結合であるか

$$f(i_1, i_2, ..., i_k) = \sum_{t=1}^{k} a_t \cdot i_t$$

   ただし，$a_t(t=1,...,k)$ は定数，あるいは，ループ不変な変数とする．

2. ループ分割に必要な情報を取得
   i) 比較演算子の情報取得
   比較演算子の種類によって，次のように *operator* に格納する．
   - 「“<”，“<=”，“>”，“>=”」の場合
     operator = “*less*”
     – 「“<=”，“>”」の場合
       N = N + 1
     – 「“>”，“>=”」の場合
       then 部 → *BlockB*
       else 部 → *BlockA*
   - 「“!=”，“==”」の場合
     operator = “*greater*”
     – 「“!=”」の場合
       then 部 ← *BlockB*
       else 部 ← *BlockA*

   ii) 各ループインデックス変数に関する情報取得

```
struct info{  char *index_var;
              char *lower_upper;
              int max, min; }  split_info[k];
char *operator;

/* 2. ループ分割に必要な情報を取得 */
/* i) 比較演算子の情報取得 */
if ( 比較演算子が"<"または"<="または">"または">=" ) {
   operator = "less";
   if ( 比較演算子が"<="または">" ) N = N + 1;
   if (比較演算子が">"または">=" ) {
      then 部 ← BlockB;
      else 部 ← BlockA;
   }
}
else if ( 比較演算子が"!=", "==" ) {
   operator = "eq";
   if (比較演算子が"!=") {
      then 部 ← BlockB;
      else 部 ← BlockA;
   }
}
/* ii) 各ループインデックス変数に関する情報取得 */
t = 1;
while(t <= k){
   split_info[t].index_var = "i_t";
   if ( a_t · i_t が単調増加 ){
         split_info[t].lower_upper = "U";
         split_info[t].max = u_t;
         split_info[t].min = l_t;
   }
   else if ( a_t · i_t が単調減少 ){
         split_info[t].lower_upper = "L";
         split_info[t].max = l_t;
         split_info[t].min = u_t;
   }
   t++;
}

/* 3. ループ分割 */
/* i) 最も内側のループ分割 */
SPLIT1(f(i_1, ..., i_k), operator, split_info[k].lower_upper);
/* ii) 最も内側以外のループ分割 */
t = k − 1;
while(t >= 1){
   if ( split[t].lower_upper == "U" ){
         g_1(i_1, ..., i_t) = f(i_1, ..., i_t, split_info[t + 1].max, ..., split_info[k].max);
         g_2(i_1, ..., i_t) = f(i_1, ..., i_t, split_info[t + 1].min, ..., split_info[k].min);
   }
   else if ( split[t].lower_upper == "L" ){
         g_1(i_1, ..., i_t) = f(i_1, ..., i_t, split_info[t + 1].min, ..., split_info[k].min);
         g_2(i_1, ..., i_t) = f(i_1, ..., i_t, split_info[t + 1].max, ..., split_info[k].max);
   }
   SPLIT2(g_1(i_1, ..., i_t), g_2(i_1, ..., i_t), operator, split_info[t].lower_upper);
   t−−;
}
```

図 11　アルゴリズム（**2.**，**3.**）
Fig. 11　Algorithm (**2.**, **3.**).

各ループインデックス変数 i_t に関して，$a_t \cdot i_t$ が単調増加か，単調減少かの情報を取得する．さらに，i_t の上限値 u_t と下限値 l_t を以下のように取得する．

- $a_t \cdot i_t$ が単調増加の場合
  - $lower_upper_t = U$
  - $max_t = u_t$，$min_t = l_t$
- $a_t \cdot i_t$ が単調減少の場合
  - $lower_upper_t = L$
  - $max_t = l_t$，$min_t = u_t$

3．ループ分割

2. で得られた情報（*operator*，$lower_upper_t$）からループ分割パターンを決定し，制約式を生成することにより分割を行う．

i) 最も内側のループ分割

(*operator*，$lower_upper_k$) より，ループ分割パターンを決定する（図 12）．制約式は条件判定式をそのまま使用する．ただし，ループ分割パター

```
/* i) 最内ループの分割 */
void SPLIT1 ( f, char *operator, char *lower_upper )
{
   if ( operator == "less" ) {
         if ( lower_upper == "U" )      then 部・else 部に分割;
         else if( lower_upper == "L" ) else 部・then 部に分割;
   }
   else if ( operator == "eq" ) {
         else 部・then 部・else 部に分割;
   }
}
```

図 12 $SPLIT1$（最内ループの分割）
Fig. 12 SPLIT1 (Innermost loop splitting).

```
/* ii) 最内ループ以外のループの分割 */
void SPLIT2 ( g, char *operator, char *lower_upper )
{
   if ( operator == "less" ) {
         if ( lower_upper == "U" )      then 部・then-else 部・else 部に分割;
         else if( lower_upper == "L" ) else 部・then-else 部・then 部に分割;
   }
   else if ( operator == "eq" ) {
         else 部・then-else 部・else 部に分割;
   }
}
```

図 13 $SPLIT2$（最内ループ以外のループの分割）
Fig. 13 SPLIT2 (Other most inner loop splitting).

ンが「else 部・then 部」の場合，比較演算子の向きを逆にする．また，比較演算子が「“==”」の場合は，3 つに分割されるため，2 つの制約式が必要となり，制約式$_1$ には “!=”，制約式$_2$ には “==” を用いる．

ii) 最も内側以外のループ分割

($operator$，$lower_upper_t$) より，ループ分割パターンを決定する（図 13）．制約式 g_1\{ 比較演算子 \}N，g_2\{ 比較演算子 \}N の g_1 と g_2 は次のようにして生成する．

- $lower_upper_t = U$ の場合

$$\begin{cases} g_1(i_1,...,i_t) = f(i_1,...,i_t, max_{t+1},...,max_k) \\ g_2(i_1,...,i_t) = f(i_1,...,i_t, min_{t+1},...,min_k) \end{cases}$$

- $lower_upper_t = L$ の場合

$$\begin{cases} g_1(i_1,...,i_t) = f(i_1,...,i_t, min_{t+1},...,min_k) \\ g_2(i_1,...,i_t) = f(i_1,...,i_t, max_{t+1},...,max_k) \end{cases}$$

また，**i)** と同様に，ループ分割パターンが「else 部・then 部」の場合，2 つの制約式の比較演算子の向きを逆にする．ただし，比較演算子が「“==”」の場合はその比較演算子を次のように変換する．

- $lower_upper_t = U$ の場合

$$\begin{cases} 制約式_1の比較演算子 \cdots “<” \\ 制約式_2の比較演算子 \cdots “<=” \end{cases}$$

- $lower_upper_t = L$ の場合

$$\begin{cases} 制約式_1の比較演算子 \cdots “>” \\ 制約式_2の比較演算子 \cdots “>=” \end{cases}$$

3.3 実 行 例

提案方式を適用した場合の実行例を示す．ここではループネストが 2 の場合と 3 の場合を示すが，同様の方式でそれ以上のネストの場合でも適用できる．また，ループネスト 2 に関しては単調増加と単調減少の場合に分けて考える．

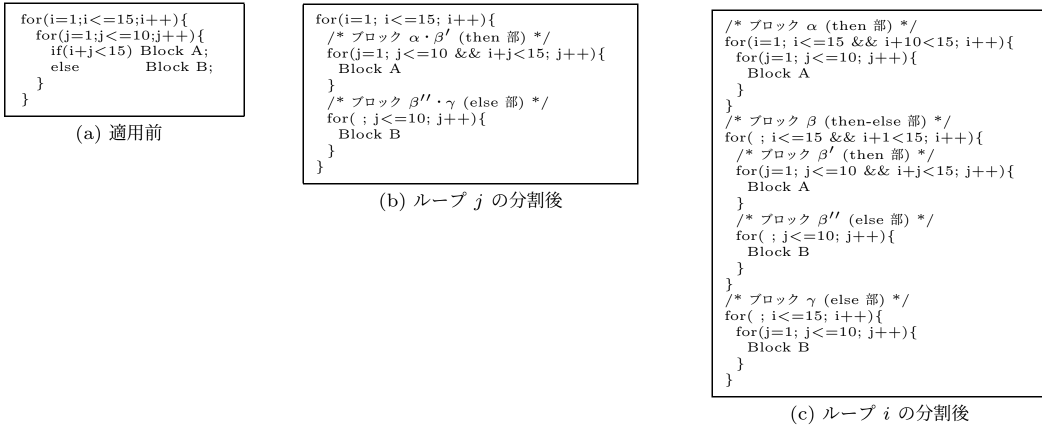
3.3.1 ループネスト 2 の場合

- 単調増加の場合

図 14 (a) のコードを例として考える．図 14 (a) は i，j に関して $a_i \cdot i(a_i = 1)$，$a_j \cdot j(a_j = 1)$ が単調増加となるコードである．

まず，ループ分割に必要な情報を取得する．比較演算子 “<” を記録する（$operator$ = “$less$”）．次に，ループインデックス変数 i の増加によって $a_i \cdot i$ は単調増加であり，上限値は 15，下限値は 1 である（i: $lower_upper_i$=“U”，max_i=15，min_i=1）．同様に，ループインデックス変数 j の増加によって $a_j \cdot j$ は単調増加であり，上限値は 10，下限値は 1 である（j: $lower_upper_j$=“U”，max_j=10，min_j=1）．

得られた情報を基に内側のループから分割していく．最も内側のループ j は，($operator$，$lower_upper_j$) から分割パターンは「then 部・



```
for(i=1;i<=15;i++){
   for(j=1;j<=10;j++){
      if(i+j<15) Block A;
      else       Block B;
   }
}
```

(a) 適用前

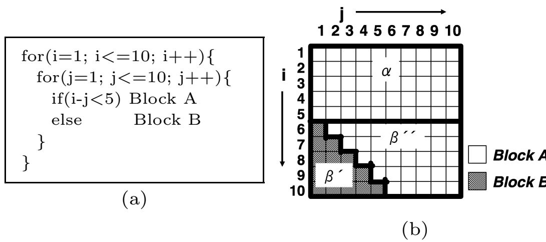
```
for(i=1; i<=15; i++){
  /* ブロック α・β' (then 部) */
  for(j=1; j<=10 && i+j<15; j++){
    Block A
  }
  /* ブロック β''・γ (else 部) */
  for( ; j<=10; j++){
    Block B
  }
}
```

(b) ループ j の分割後

```
/* ブロック α (then 部) */
for(i=1; i<=15 && i+10<15; i++){
  for(j=1; j<=10; j++){
    Block A
  }
}
/* ブロック β (then-else 部) */
for( ; i<=15 && i+1<15; i++){
  /* ブロック β' (then 部) */
  for(j=1; j<=10 && i+j<15; j++){
    Block A
  }
  /* ブロック β'' (else 部) */
  for( ; j<=10; j++){
    Block B
  }
}
/* ブロック γ (else 部) */
for( ; i<=15; i++){
  for(j=1; j<=10; j++){
    Block B
  }
}
```

(c) ループ i の分割後

図 14 ループネスト 2—加算の場合
Fig. 14 Double nested loop — case of add.



```
for(i=1; i<=10; i++){
  for(j=1; j<=10; j++){
    if(i-j<5) Block A
    else      Block B
  }
}
```


(a) (b)

図 15 ループネスト 2 — $i - j$
Fig. 15 Double nested loop — $i - j$.

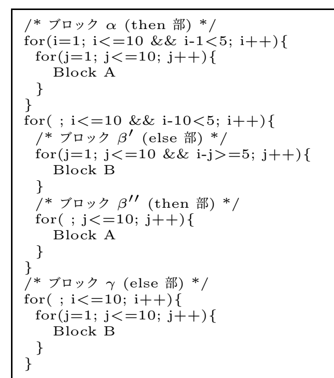
else 部」となる．したがって，制約式は条件判定式 $i + j < 15$ をそのまま使用して，図 14 (b) となる．

次に，外側のループ i の分割は，(*operator*, $lower_upper_i$) から分割パターンは「then 部・then-else 部・else 部」となる．また，$lower_upper_i = U$ より，制約式 $i{+}10 < 15$ と $i{+}1 < 15$ を生成する．よって，2 つの制約式を使用して，図 14 (c) となる．

- 単調減少の場合

図 15 のコードを例として考える．図 15 は i に関して $a_i \cdot i (a_i = 1)$ が単調増加となり，j に関して $a_j \cdot j (a_j = -1)$ が単調減少となるコードである．

まず，ループ分割に必要な情報を取得する．比較演算子 “<” を記録する（*operator* = “*less*”）．次に，ループインデックス変数 i の増加によって $a_i \cdot i$ は単調増加であり，上限値は 10，下限値は 1 である（i: $lower_upper_i$=“U”，max_i=10，min_i=1）．同様に，ループインデックス変数 j の増加によって $a_j \cdot j$ は単調減少であり，上限値は 10，下限値は 1 である（j: $lower_upper_j$=“L”，max_j=1，min_j=10）．



```
/* ブロック α (then 部) */
for(i=1; i<=10 && i-1<5; i++){
  for(j=1; j<=10; j++){
    Block A
  }
}
for( ; i<=10 && i-10<5; i++){
  /* ブロック β' (else 部) */
  for(j=1; j<=10 && i-j>=5; j++){
    Block B
  }
  /* ブロック β'' (then 部) */
  for( ; j<=10; j++){
    Block A
  }
}
/* ブロック γ (else 部) */
for( ; i<=10; i++){
  for(j=1; j<=10; j++){
    Block B
  }
}
```

図 16 ループネスト 2—減算の場合
Fig. 16 Double nested loop — case of sub.

得られた情報を基に分割していく．最も内側のループ j は，(*operator*, $lower_upper_j$) から，分割パターンは「else 部・then 部」となる．制約式は条件判定式の比較演算子 “<” を “>=” にし，$i - j >= 5$ を使用する．また，外側のループ i の分割は，(*operator*, $lower_upper_i$) から分割パターンは「then 部・then-else 部・else 部」となる．よって，制約式 $i - 1 < 5$ と $i - 10 < 5$ を使用して，**図 16** となる．

3.3.2 ループネスト 3 の場合

次に，ループネストが 3 の場合について考える．ここでは，**図 17** のようなネスト 3 で，条件判定式に加算と減算を含むコードを例に考える．

まず，ループ分割に必要な情報を取得する．比較演算子 “<” を記録する（*operator* = “*less*”）．次に，ループインデックス変数 i の増加によって $a_i \cdot i (a_i = 1)$ は単調増加であり，上限値は 10，下限値は 1 であ

る（i: $lower_upper_i$=“U”，max_i=10，min_i=1）．同様に，ループインデックス変数 j の増加によって $a_j \cdot j(a_j = 1)$ は単調増加であり，上限値は 10，下限値は 1 である（j: $lower_upper_j$=“U”，max_j=10，min_j=1）．ループインデックス変数 k の増加によって $a_k \cdot k(a_k = -1)$ は単調減少であり，上限値は 10，下限値は 1 である（k: $lower_upper_k$=“L”，max_k=1，min_k=10）．

得られた情報を基に分割していく．最も内側のループ k は,（$operator$，$lower_upper_k$）から，分割パターンは「else 部・then 部」となる．制約式は条件判定式の比較演算子 “<” を “>=” にし，$i+j-k >= 12$ を使用する．ループ j の分割は,（$operator$，$lower_upper_j$）から分割パターンは「then 部・then-else 部・else 部」となる．制約式 $i+j-1 < 12$ と $i+j-10 < 12$ を使用して分割する．最後に，ループ i の分割は,（$operator$，$lower_upper_i$）から分割パターンは「then 部・then-else 部・else 部」となる．制約式 $i+10-1 < 12$ と $i+1-10 < 12$ を使用して**図 18** となる．

```
for(i=1; i<=10; i++){
  for(j=1; j<=10; j++){
    for(k=1; k<=10; k++){
      if(i+j-k<12) Block A
      else         Block B
    }
  }
}
```

図 17 ループネスト 3（適用前）
Fig. 17 Triple loop nest (before).

```
for(i=1; i<=10 && i+10-1<12; i++){
  for(j=1; j<=10; j++){
    for(k=1; k<=10; k++){
      Block A /* α (then 部) */
    }
  }
}
for( ; i<=10 && i+1-10<12; i++){
  for(j=1; j<=10 && i+j-1<12; j++){
    for(k=1; k<=10; k++){
      Block A /* β (then 部) */
    }
  }
  for( ; j<=10 && i+j-10<12; j++){
    for(k=1; i+j-k>=12; k++){
      Block B /* γ (else 部) */
    }
    for( ; k<=10; k++){
      Block A /* δ (then 部) */
    }
  }
  for( ; j<=10; j++){
    for(k=1; k<=10; k++){
      Block B /* ϵ (else 部) */
    }
  }
}
for(; i<=10; i++){
  for(j=1; j<=10; j++){
    for(k=1; k<=10; k++){
      Block B /* ζ (else 部) */
    }
  }
}
```

図 18 ループネスト 3（適用後）
Fig. 18 Triple nested loop (after).

4. 評　　価

実行時間について，提案方式の評価を行う．評価環境を**表 1** に示す．

提案方式は，条件判定式の実行回数を減少させることによって，実行時間を短縮している．そのため，条件判定式に使用される演算子の種類，then 部と else 部の実行割合，ループボディの負荷によって，実行時間の短縮率は変化する．また，提案方式が既存の最適化と組み合わせても効果があることを示すために，ループ最適化で特に有効な Loop Unrolling を組み合わせて評価する必要がある．さらに，提案方式は複数のループインデックス変数を含むことができるので，ループネストが 2 以上の場合についても有効であるかを検証する．そこで，実行時間の評価は次の 5 つについて行う．

- **演算の種類**
  条件判定式で使用される演算の種類による有効性について評価する．
- **then 部と else 部の実行割合**
  then 部と else 部の実行割合の変化による実行時間の違いについて評価する．
- **ループボディの負荷**
  ループボディの負荷を増加させたときの提案方式の実行時間短縮率について評価する．
- **Loop Unrolling との併用**
  提案方式と Loop Unrolling を併用した場合の効果について評価する．
- **ループネストの深さ**
  ループネストを 1 から 5 まで変化させたときの提案方式の実行時間短縮率について評価する．

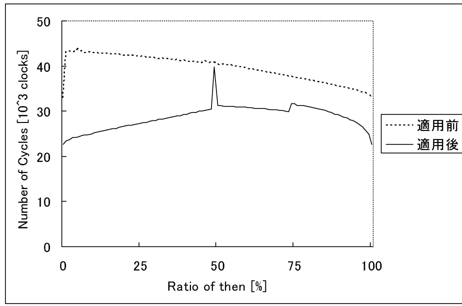
評価として用いるコードを**図 19** に示す．このコー

表 1 評価環境
Table 1 Evaluation environment.

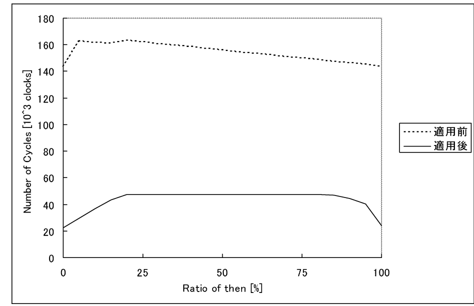
OS	Fedora Core 3 kernel 2.6.11
CPU	Pentium 4 2.66 GHz
memory	256 MB
compiler	gcc-3.4.3 (-O3 -march=pentium4)

```
for(i=1; i<=100; i++){
  for(j=1; j<=100; j++){
    if(f(i, j) < N) sum1+=j;
    else            sum2+=j;
  }
}
```

図 19 評価するコード
Fig. 19 Evaluated code.



(a) $f(i,j) = i + j$



(b) $f(i,j) = i + j/3$

図 20 実行時間の比較（ループネスト 2）
Fig. 20 Comparing execution time (double nested loop).

ドはループネストが 2 でループ内に if 文の条件分岐を含んでいる．また，if 文の条件判定式は $f(i,j) < N$ である．ループ内では，単純な加算を行っている．

条件判定式の算術式 $f(i,j)$ を加算のみの式（$i+j$），除算を含む式（$i+j/3$）の 2 つの場合に分けて測定する．また，2 つの条件判定式に対して N の値を変化させて，then 部と else 部の実行割合を変えることにより測定する．

4.1 演算の種類

はじめに条件判定式で使用される演算の種類によって実行時間の短縮率にどのような影響を与えるかについて述べる．**図 20** のグラフは横軸が then 部の実行割合，縦軸がクロックサイクル数を表している．また，2 つのグラフは算術式が (a) 加算のみの式，(b) 除算を含む式の場合を示している．

表 2 は実行時間の短縮率の最大，最小を表している．表 2 より 2 つの算術式において，提案方式は適用前に比べ実行時間が短縮されていることが分かる．また，表 2 より算術式が加算のみの式と比べ，除算を含む式の方が短縮率が高いことに注目する．これは提案方式が条件判定式の実行回数を減少させる性質を持ち，負荷の高い演算の回数が減少するほど実行時間に大きく影響する．つまり，提案方式は演算の負荷が大きいほど，短縮率が高くなるといえる．

4.2 then 部と else 部の実行割合

次に，then 部と else 部の実行割合による実行時間の変化について述べる．図 20 より，then 部と else 部の実行割合が変化することによって実行時間に影響を与えていることが分かる．特に，then 部と else 部の実行割合が同程度となる場合，提案方式の実行時間短縮率は最小となっている．しかし，then 部と else 部の割合が同程度となる場合でも，提案方式は適用前と比べ実行時間が短縮されている．このことから，then 部と else 部の実行割合により，実行時間は変化するが，つねに実行時間を短縮できるといえる．

表 2 実行時間短縮率
Table 2 Execution time shortening rate.

	加算	除算
最大（%）	46.4	85.8
最小（%）	2.57	67.9

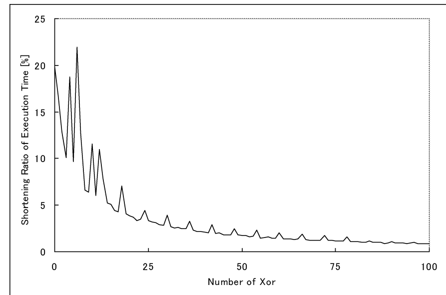


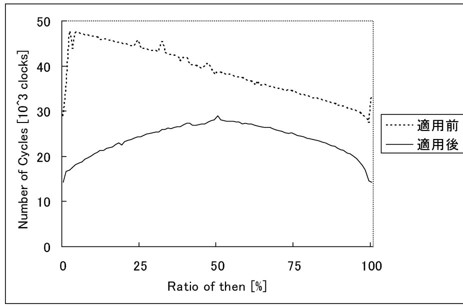
図 21 ループボディの負荷
Fig. 21 Load of loop body.

4.3 ループボディの負荷

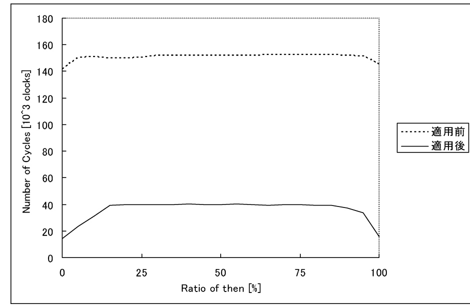
図 21 は図 19 の then 部と else 部のコードそれぞれに xor 命令を 0 個から 100 個まで加えた場合の実行時間の短縮率を示している．図 21 より，ループボディの負荷が大きくなるにつれて短縮率が減少していることが分かる．これは，条件判定式の負荷がループボディの負荷によって相対的に影響が小さくなるためである．しかし，ループボディの負荷が大きくなっても，提案方式の実行時間が適用前の実行時間を上回ることはない．よって，ループボディの負荷が増加すると短縮率は減少するが，実行時間は必ず短縮できる．

4.4 Loop Unrolling との併用

提案方式と Loop Unrolling との併用について述べる．**図 22** はループネスト 2 で条件判定式の算術式が (a) $i + j$ と (b) $i + j/3$ の 2 つの場合に Loop Unrolling を適用した結果を示している．図 22 の適用前は提案方式を適用前のプログラムに Loop Unrolling



(a) $f(i,j) = i + j$



(b) $f(i,j) = i + j/3$

図 22 Loop Unrolling との併用
Fig. 22 Using Loop Unrolling.

表 3 実行時間短縮率（Loop Unrolling）
Table 3 Execution time shortening rate (Loop Unrolling).

	加算	除算
最大 (%)	64.7	90.0
最小 (%)	25.1	73.0

を施した結果である．また，図 22 の適用後は提案方式を適用後のプログラムに Loop Unrolling を施した結果である．図 22 より，図 20 (a) と同様に，提案方式は適用前と比較すると実行時間が短縮されているのが分かる．また，**表 3** は提案方式を適用前と適用後のそれぞれのループに Loop Unrolling を施した場合の加算，除算の実行時間短縮率の最大，最小を表している．表 3 と表 2 を比較すると，Loop Unrolling を併用した場合でも短縮率は同等の結果であることが分かる．したがって，提案方式は Loop Unrolling と組み合わせても有効である．

4.5 ループネストの深さ

本節では，提案方式の性質を述べてから，ループネストの深さについて評価する．提案方式はループ内の条件分岐を排除し，その条件判定式 $f < N$ の実行回数を減らすことによって，実行時間短縮を図っている．したがって，提案方式は条件判定式 $f < N$ の実行回数が実行時間に影響する．

そのため，ループネストが深くなると条件判定式の実行回数がどのようになるかを測定した．測定には**図 23** のコードを用いた．また，条件判定式は $f(i_1, i_2, ..., i_k) = i_1 - i_2 - \cdots - i_k (1 \leq k \leq 5)$ で，N は then 部と else 部の割合が同程度となる部分で測定した．その結果**図 24** のようになった．図 24 はループネストを変化させたときの条件判定式 $f < N$ の実行回数の減少率を示している．図 24 より，ループネストが深くなると，条件判定式の実行回数の減少率が上がっていることが確認できる．したがって，提案方式はループネストの深さに関係なく効果が得られると

```
for(i1 = 1; i1 <= 100; i1++){
  for(i2 = 1; i2 <= 100; i2++){
     .  .  .  .  .
     for(ik = 1; ik <= 100; ik++){
       if( f(i1, i2, ..., ik) < N )  sum1+ = j;
       else                          sum2+ = j;
     }
     .  .  .  .  .
  }
}
```

図 23 ループネストの深さを評価するためのコード
Fig. 23 Evaluated code for depth of loop nest.

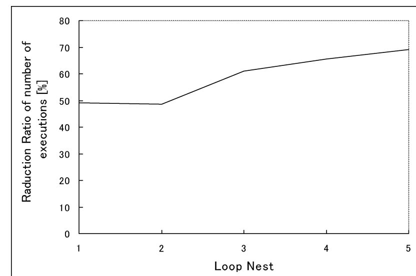


図 24 $f < N$ の実行回数の減少率
Fig. 24 Reduction ratio of number of $f < N$ executions.

考えられる．

そこで，ループネストに関係なく効果が得られることを示すために，ループネストを 1 から 5 まで変化させたときの提案方式の性能評価を行った．評価には図 23 のコードを用いた．条件判定式は $f(i_1, i_2, ..., i_k) = i_1 - i_2 - \cdots - i_k (1 \leq k \leq 5)$ とする．また，性能評価として then 部と else 部の割合が同程度となる部分で比較する．

結果を**図 25** に示す．図 25 より，ループネストが 2 以上で短縮率が一定となっていることが分かる．よって，提案方式はループネストの深さに関係なく一定の効果が得られるといえる．

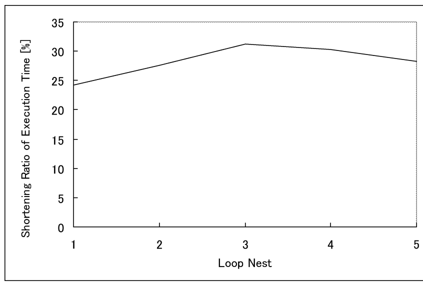


図 25 ループネストの深さ
Fig. 25 Depth of loop nest.

5. む す び

本論文では，最適化方式の 1 つであるループ内条件分岐排除方式について調査・検討を行った．そして，従来方式では適用できなかったループ内の条件分岐を排除できる新方式を提案した．この新方式は，ループ内に存在する if 条件分岐が，複数のループインデックス変数を含む算術式と定数との比較である場合に，内側のループからループ分割を行うことにより，条件分岐を排除するという方式である．本論文では，その提案方式を実現するアルゴリズムを示し，実行時間に関して評価を行った．評価結果は演算の種類や then 部の実行割合，ループボディの負荷によって変化するが実行時間を短縮できた．また，提案方式はループネストの深さに関係なく一定の性能を保つことができる．さらに，Loop Unrolling を併用しても有効であることが分かった．以上より，提案方式は有効であると考えられる．

今後は，分割後にできる冗長なループを削除するアルゴリズムについて検討する．また，適用率をさらに向上させるために本手法の拡張が必要となる．

謝辞 この研究は柏森情報科学振興財団の助成を受けて遂行された．また，論文に対する修正意見をいただいた査読者の方に心より感謝します．

参 考 文 献

1) Bacon, D.F., Graham, S.L. and Sharp, O.J.: Compiler Transformations for High-Performance Computing, *ACM Computing Surveys*, Vol.26, No.4, pp.345–420 (1994).
2) 中田育男：コンパイラの構成と最適化，朝倉書店 (1999).
3) 多田井靖子：ループ内条件分岐削除最適化方式，特許公開 2001-142716.
4) Tal, A.: Generalized Index Set Splitting, *Workshop on Compiler-Driven Performance* (2003).
5) Sparker, V.: Optimezed unrolling of nested loops, *Proc. 14th International Conference on Supercomputing*, pp.153–166 (2000).
6) Griebl, M., Feautrier, P. and Lengauer, C.: Index Set Splitting, *PACT'99* (1999).
7) Allen, R. and Kennedy, K.: *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Pub. (2001).
8) Wolfe, M.: *High Performance Compilers for Parallel Computing*, Addison-Wesley (1995).
9) Jimenez, M., Llaberia, J.M. and Fernandez, A.: Register tiling in nonrectangular iteration spaces, *ACM Trans. Prog. Lang. Syst.* (*TOPLAS*), Vol.24, No.4, pp.409–453 (2002).

(平成 17 年 7 月 5 日受付)
(平成 17 年 11 月 9 日採録)

塩足 拓也（学生会員）
昭和 57 年生．平成 17 年熊本大学工学部数理情報システム工学科卒業．同年熊本大学大学院自然科学研究科数理科学・情報システム専攻入学，現在に至る．

畑邉 誠和
昭和 54 年生．平成 16 年熊本大学大学院自然科学研究科数理科学・情報システム専攻修了．同年日立製作所ソフトウェア事業部入社．

木山 真人（正会員）
昭和 51 年生．平成 11 年広島市立大学情報科学部情報工学科卒業．平成 15 年広島市立大学大学院情報科学研究科情報科学専攻博士後期課程修了．同年熊本大学工学部数理情報システム工学科助手．現在に至る．博士（情報工学）．プログラミング言語，言語処理系，オブジェクト指向言語の高速化手法に興味を持つ．

梅野　英典（正会員）

昭和 22 年生．昭和 45 年 3 月九州大学理学部数学科卒業．同年 4 月株式会社日立製作所中央研究所入所．昭和 51 年 8 月同システム開発研究所．平成 5 年 2 月同汎用コンピュータ事業部．平成 8 年 9 月同オフィスシステム事業部．平成 8 年 9 月博士（理学：東京大学）．平成 10 年 10 月熊本大学工学部数理情報システム工学科教授．研究分野：以下の性能・機能・信頼性向上方式：オペレーティングシステム，仮想計算機，データベース管理システム，SMP クラスタ，最適化コンパイラ，Web サーバ，システムソフトウェアから見た計算機アーキテクチャ．ACM，IEEE Computer Society 各会員．