プログラムの命令列表現の自由度に関する研究

Redundancy in instruction sequences of computer programs

豊橋技術科学大学 大学院 工学研究科
知識情報工学専攻 市川研究室

013740 八反田一宏

目次

1　はじめに

コンピュータプログラムは，目的とする機能を実現するための命令列等の集合である．プログラムが機能 f を実現するとき，構成する命令列表現は一意に決定しないため，製作者が自由に命令を選び利用することができる．同一の機能 f を実行する命令列 I と I' が存在するとき，これらを「機能等価な命令列」と呼ぶ．さらに，１つのプログラムに対する機能等価な命令列の数を「命令列表現の自由度」と呼ぶ．命令列表現の自由度を利用して情報を表現することができるため，電子透かしや情報隠蔽等に応用されている[1)]．

本研究の目的は，命令列中の命令とデータの位置を変更することによって得られる自由度を定量的に測定することである．しかし，機能等価な命令列を得るための方法は多数存在するため，全ての自由度について測定を行うことは困難である．そのため今回は，

- グローバル変数の配置順序
- ローカル変数の配置順序
- 基本ブロックの配置順序
- 基本ブロック内の命令の順序変更

の 4 種類に関して，自由度を測定する．

配置順序の変更を行うことで，性能やオブジェクトファイルサイズが変化する可能性がある．そのため，ベンチマークプログラムとフリーソフトウェアを用い，オブジェクトファイルサイズの変化を測定した．さらに，ベンチマークプログラムの実行ファイルを生成し，性能値の変化を測定した．

2　関連研究

命令列表現の自由度が存在することは，機能等価なプログラムを生成できることを意味する．機能等価な命令列は，情報を表現する電子透かしや電子署名，情報隠蔽などの手法以外にも利用されている．

本章では，プログラムの自由度を用いた応用であるプログラムの難読化や電子透かし，電子署名に関する先行研究について説明する．

2.1　プログラムの難読化

プログラムが実現する機能は，逆アセンブルや逆コンパイルなどの手法によって解析される可能性がある．機能自体が秘密なプログラムでは，第三者に実装機能が解析されると作者に対し不利益が生じる．リバースエンジニアリングを抑止するためには，解析に必要な人的・時間的コストを上昇させる必要がある．そのため，機能の解析が困難になるよう命令列を機能等価に変形する．

門田らは，ループの構造を変形することで，プログラムの構造を難読化する手法を提案している[11]．制御構造を有向グラフに変換し,「ループの移動」「ループの拡大」「合流点の併合」「合流点の分割」といった制御構造の変換規則に従い，制御構造を等価変換する．プログラムを難読化することは簡単だが，難読化されたプログラムを元に戻すことは難しい．手法の評価実験として，大学院生に対してプログラムの機能解析テストを行っている．

2.2　プログラムへの電子透かし・電子署名

プログラムへの電子透かし・電子署名手法は複数提案されている．電子透かし法は「静的な手法」と「動的な手法」の 2 種類に分けられる．静的な手法は，プログラム自体の構造を変形することで情報を構成するのに対し，動的な手法はプログラムの実行中にメモリ上に情報を構成するための命令列を挿入する．動的な手法は静的な手法よりも情報自体を読み取られにくいという特性がある．

はじめに，静的な手法について説明する．

中村らは，等価な演算命令に変換することで，情報を埋め込む手法を提案している[9]．冗長な命令の挿入位置を情報として保持することで，電子透かしやステガノグラフィ，認証へ応用できる可能性がある．結託攻撃を除けば，冗長な命令の位置を特定するのは困難であるため，秘匿性が高くなるとしている．

Davidson らは，プログラム中の基本ブロックの順序を変更することにより情報を埋め込むことができる手法を考案している[2]．しかし，埋め込み後のプロ

グラム単体から情報を抽出することはできない上，手法を適用することによって生じるオーバーヘッド等については測定されていない．筆者の卒業研究では，MIPSの命令セットを用いた電子透かし法として実装した[4]．その結果，プログラムコードサイズの約0.2%の情報密度で，情報の埋め込み・抽出ができた．

門田らは，Javaバイトコードに対する電子透かし法を提案している[12]．バイトコード中に実行されない関数を作成し，その中に含まれる命令の種類を情報として構成している．Javaインタプリタは，バイトコードを厳密に評価するため，無効な命令コードを挿入することができない．そのため，命令と情報の対応を示すテーブルを定義し，埋め込みデータに対して適切な命令を配置し情報構成する．門田は提案手法を実装し，電子透かしツールをフリーソフトとして公開している[10]．

Venkatesanらは，基本ブロックとコントロールフローをグラフに置き換え，透かしのグラフを挿入することで電子透かしを構成する手法を提案している[7]．透かし情報をグラフとして定義し，オリジナルプログラムのコントロールフローグラフと結合する．これにより，追加した基本ブロックとコントロールフローの特定が難しくなるため透かしの抽出が困難になる．そのためVenkatesanらの手法は，静的な電子透かし法の中では最も強固な手法であると言われている[1]．

次に, 動的な手法について説明する．

Collbergらは，実行時に動的なメモリ空間に透かし情報を構築する手法を提案している[1]．実行時に特定の入力を与えると，挿入された透かし構築命令列が動的なメモリ空間に透かしを示すグラフ構造を構築する．透かし抽出命令列によりグラフ構造を読み取ることで，埋め込まれたデータを読み込むことができる．Collbergらの手法では，実行時にはじめて透かし情報が構築される．プログラム自体を解析するだけでは，埋め込まれた情報を解読することが難しいため，秘匿性が高まる．

本章では，命令列表現の自由度を用いた，電子透かし，難読化の手法を挙げた．手法の有用性は，埋め込み可能な情報量や実行時オーバーヘッドを定量的に測定しなければ評価することはできない．

3　順序変更可能な要素と手法

本章では，プログラム中に存在する配置順序変更可能な要素について検討する．命令列表現の自由度とは，生成可能な機能等価な命令列の組み合わせ数である．配置順序変更可能な要素が n 個存在するとき，$n!$ 通りの自由度が存在する．

本研究における検討は，Intel x86 命令セット (ELF ファイル形式) を前提として行う．

開発環境として GCC 2.95.3 と binutils 2.13 を使用し，オブジェクトファイル，実行ファイルを生成した．

3.1 変数の配置順序変更

変数とは，プログラムの実行中に必要な情報を格納しておくための領域である．変数は，実行ファイル中ではレジスタかメモリ上に配置され，命令によって適切な値が格納される．

3つの変数 a, b, c が存在するとき，メモリ上の配置順序を (a, b, c) から (b, c, a) に変更することを考える．命令に与える参照アドレスを適切に変更すれば実行結果に影響を与えないため，機能等価なプログラムを生成することができる．

C 言語における変数には，グローバル変数とローカル変数が存在する．さらにグローバル変数は，実行中に配置されるメモリ領域の違いにより，外部変数 (関数の外で宣言される変数)，static 変数 (static 識別子を指定して定義された変数)，初期化済み変数 (外部変数，static 変数の中で，初期値が与えられて宣言されている変数) に分類できる．各変数の種類とアドレス決定のタイミングと変数が格納される領域を**表 1** に示す．ELF ファイル形式では，.data セクションは初期化済みデータが格納されたメモリイメージ，.bss セクションは未初期化データが格納されたメモリイメージである．これらの変数は，実行時の配置領域が異なるため，それぞれ別の配置要素として扱うことができる．グローバル変数の配置自由度は，外部変数，static 変数，初期化済み変数の配置自由度の積である．

表 1　変数の種類と配置領域

変数の種類	アドレスの決定	変数の配置領域
外部変数	リンク時	.bss セクション
static 変数	アセンブル時 (オフセットの決定)	.bss セクション
初期化済み変数	アセンブル時 (オフセットの決定)	.data セクション
ローカル変数	コンパイル時	スタックフレーム (ebp レジスタ相対アドレスでアクセス)

3.1.1 外部変数

C 言語における外部変数とは，関数の外部で宣言されている変数のことである．extern 宣言によって，実体が宣言されていないソースファイルからも変数を参照することができる．外部変数の実行時アドレスは，リンクする全てのオブジェクトファイルで定義された情報が揃わなければ決定することができない．そのため，リンカがリンク時にアドレスを決定する．

本研究では，binutils に含まれるリンカ (ld) の外部変数アドレス決定部分を改造し，外部変数を任意の順序で配置できるようにした．外部変数を含む C 言語プログラムのソースファイルと，配置順序変更に伴う変数のアドレス変化の例を**図 1** に示す．上段は外部変数を含む C のソースプログラムであり，下段は nm を用いて実行形式ファイルのシンボル情報を表示したものである．左側は順序変更前のもの，右側は順序変更後のものである．順序変更前は (b, c, a) の順序で登録されているものが，順序変更後は (a, b, c) の順序で登録されている．

アセンブリファイル中において外部変数は，.comm 命令を用いて定義される．後述する static 変数は .comm 命令と .local 命令を用いて定義される．重複を防ぐため，外部変数の数は (.comm 命令の数 - .local 命令の数) で数えあげる．

```
int a, b, c;

int main(int argc, char *argv[])
{(略)
```

```
$ nm --numeric-sort global  | grep ' B '
08049564 B b
08049568 B c
0804956c B a
```

```
$ nm --numeric-sort global2 | grep ' B '
08049564 B a
08049568 B b
0804956c B c
```

図 1　グローバル変数の順序変更に伴う，変数のアドレスの変化

3.1.2　static 変数

C 言語における static 変数とは，変数の定義時に static 識別子が指定されているものである．static 変数は，関数の外部で宣言するとソースファイル全体で参照することができるが，関数の内部で宣言すると関数の内部でのみ参照可能な変数となる．前者を「外的な static 変数」，後者を「内的な static 変数」と呼ぶ．static 変数は各ソースファイルごとに独立して定義できるため，別ファイル間で同一名の変数を定義できる．さらに内的な static 変数は，同一ファイル内の関数間でも同一名の変数を定義することができる．

実行ファイル内では，外的，内的な static 変数は同一のものとして扱われる．内的な static 変数は変数名の衝突を避けるため，変数名の末尾に .(ピリオド) と通し番号が付加される．

static 変数のアドレスは，2 段階の処理を行い決定する．はじめに，アセンブラが各変数ごとに.bss セクション先頭からのオフセット値を決定する．このオフセット値が変数の配置順序となる．次にリンカは，各オブジェクトファイルが持つ .bss セクションの先頭アドレスを決定する．先頭アドレスに各変数のオフセット値を加算することで，実行時のアドレスを決定する．

アセンブリファイルの書き換え例を**図 2** に示す．左がオリジナル順序のアセンブリファイル，右が逆順に変更したアセンブリファイルの一部である．static 変数の配置順序変更は，アセンブリファイル中に記述された変数定義命令の順序を変更することで実現できる．そのため，アセンブリファイルを読み込み，任意の順序に変数を並べ変えたアセンブリファイルを出力するプログラム (Perl スクリプト) を作成した．

static 変数は，.local 命令と .comm 命令によって定義される．.comm 命令の第 1 引数はシンボル名，第 2 引数はサイズ，第 3 引数はアライメントの指定サイズである．.comm 命令は外部変数の定義に使用されるが，.local 命令を前置することによりローカルシンボルとして定義できる．そのため，.local 命令と.comm 命令を 1 組として任意の順序に変更することで，static 変数の配置順序を変更することができる．

static 変数の数は，.local 命令と .comm 命令の組数によって数えることができる．

```
.local   aa.0
.comm aa.0,320000,32
.local   a.1
.comm a.1,321600,32
.local   b.2
.comm b.2,1600,32
.local   x.3
.comm x.3,1600,32
.local   ipvt.4
.comm ipvt.4,800,32
.local   n.5
.comm n.5,4,4
```

```
.local   n.5
.comm n.5,4,4
.local   ipvt.4
.comm ipvt.4,800,32
.local   x.3
.comm x.3,1600,32
.local   b.2
.comm b.2,1600,32
.local   a.1
.comm a.1,321600,32
.local   aa.0
.comm aa.0,320000,32
```

図 2　static 変数順変更前後のアセンブリファイルの一部

3.1.3 初期化済み変数

初期化済み変数とは，宣言時に初期値が設定された外部変数と static 変数のことである．実行ファイル中では，未初期化データが配置される.bss セクションではなく.data セクションに配置される．そのため，外部変数や static 変数とは独立した要素として配置順序を変更することができる．

初期化済み変数の配置順序は static 変数と同様に，2 段階の処理を行い決定する．はじめに，アセンブラが各変数ごとに.data セクション先頭からのオフセット値を決定する．このオフセット値が変数の配置順序となる．次にリンカは，各オブジェクトファイルが持つ .data セクションの先頭アドレスを決定する．先頭アドレスに各変数のオフセット値を加算することで，実行時のアドレスが決定する．

初期化済み変数は.type 命令によって定義される．初期化済み変数は初期値の定義が行われているため，データ指定命令もあわせて扱う必要がある．そのため，変数名の記述されたラベルを検索し，定義されたデータ指定命令を読み込む．値の定義は，**表 2** に示す擬似命令によって行われる．最後に，変数の定義とデータを示す命令を任意の順序で出力することで，変数の配置順序を変更することができる．

表 2　値を定義する擬似命令

long	ascii	asciz	byte	double
float	int	hword	octa	quad
short	single	sleb128	uleb128	word
zero	string			

図 3 に，アセンブリファイルの書き換え例を示す．左が配置順序変更前，右が配置順序変更後である．

初期化済み変数の数は，第 2 引数に@objectが指定された.type 命令の数によって数え上げる．

```
.data
   .align 4
   .type  a1,@object
   .size  a1,4
a1:
   .long 0
   .align 4
   .type  a2,@object
   .size  a2,4
a2:
   .long 1
   .align 4
   .type  a3,@object
   .size  a3,4
a3:
   .long 2
   .align 4
   .type  a4,@object
   .size  a4,4
a4:
   .long 3
   .align 4
   .type  a,@object
a:
   .long 0
.byte 10
   .zero 3
   .long 1
.byte 20
   .zero 3
   .size  a,16
```

```
.data
   .align 4
   .type  a1,@object
   .size  a1,4
a:
   .long 0
.byte 10
   .zero 3
   .long 1
.byte 20
   .zero 3
   .size  a,16
a4:
   .long 3
   .align 4
   .type  a,@object
a3:
   .long 2
   .align 4
   .type  a4,@object
   .size  a4,4
a2:
   .long 1
   .align 4
   .type  a3,@object
   .size  a3,4
a1:
   .long 0
   .align 4
   .type  a2,@object
   .size  a2,4
```

図 3　初期化済みスタティック変数順変更前後のアセンブリファイル

3.1.4　ローカル変数

C 言語におけるローカル変数とは，関数の内部で定義されている変数のことである．ローカル変数は関数の内部でのみアクセス可能であり，関数から制御が離れると値は保持されない．

GCC では，ローカル変数はスタック上の領域か擬似レジスタとして登録される．擬似レジスタとは，GCC 内部で定義される仮想的なレジスタである．GCC は，レジスタが無限にあるものと仮定して RTL (Register Transtfer Language) を生成している．

GCC では，最適化をかけない場合全てのローカル変数をスタック上に配置する．しかし，最適化をかけた場合は可能な限り擬似レジスタに割り当てる．変数

のサイズの制約から擬似レジスタに割り当てられない変数 (配列等) は，スタック上に配置される．その後 GCC は，可能な限り擬似レジスタを実レジスタに割り当てるように命令コードを生成する．実レジスタに空きがない場合はスタック上に配置する．

Intel x86 プロセッサ上では，スタック上に配置されたローカル変数を EBP レジスタとオフセット値を用いて参照する．EBP レジスタの値はプログラム実行時に決定するため，コンパイル時では不明である．オフセット値はコンパイラで決定し，スタック上の変数の配置順序を決定する．

GCC は構文解析時に変数の登録を行い，変数をスタック上か擬似レジスタに割り当てる．今回は配置順序の変更対象を，あらかじめスタック上に配置され，かつ関数の先頭部分で定義された変数のみに限定する．擬似レジスタは扱わない．変数宣言終了後の文を処理する前に，スタック上のローカル変数のオフセット値を任意に変更できるように GCC を改造する．

ローカル変数を含む C 言語プログラムと，ローカル変数順序の変更による命令の変化を**図 4** に示す．上段がサンプルとなる C 言語プログラムであり，下段左がオリジナルの順序の場合のアセンブリソース，下段右が配置順序の変更を行った後のアセンブリソースである．下段左では，4 行目で 0 を (ebp-8) のアドレスに代入しているが，下段右では (ebp-24) のアドレスへ代入されている．

```
int main(int argc, char *argv[])
{
        int a, b = 0, c = 1, d, e, f;
        a = b + c;
        return a;
}
```

```
(略)
  pushl %ebp
  movl %esp,%ebp
  subl $40,%esp
  movl $0,-8(%ebp)
  movl $1,-12(%ebp)
  movl -8(%ebp),%eax
  movl -12(%ebp),%edx
(略)
```

```
(略)
  pushl %ebp
  movl %esp,%ebp
  subl $40,%esp
  movl $0,-24(%ebp)
  movl $1,-20(%ebp)
  movl -24(%ebp),%eax
  movl -20(%ebp),%edx
(略)
```

図 4　ローカル変数の順序変更に伴う，変数の相対アドレスの変化

3.2 命令順序の変更

3.2.1 基本ブロックの配置順序変更

命令列は基本ブロックに分割することができる．基本ブロックとは，先頭に制御が与えられたとき，途中で分岐することなく終端で制御が離れる命令列のことである[8]．実行順序を適切に保持すれば，基本ブロックの配置順序を変更しても機能は等価となる (**図 5**)．本研究では，アセンブリファイルに記述された命令列を基本ブロックへ分割し，任意の順序に並び換えるためのフィルタプログラム (Perl スクリプト) を作成した．

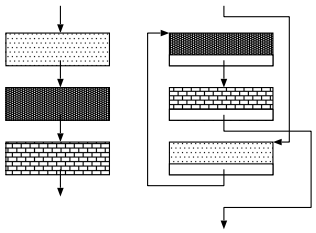


図 5　基本ブロックの順序変更と jmp 命令の挿入

フィルタプログラムは，コンパイラが出力したアセンブリファイルを読み込み，以下の処理を行う．

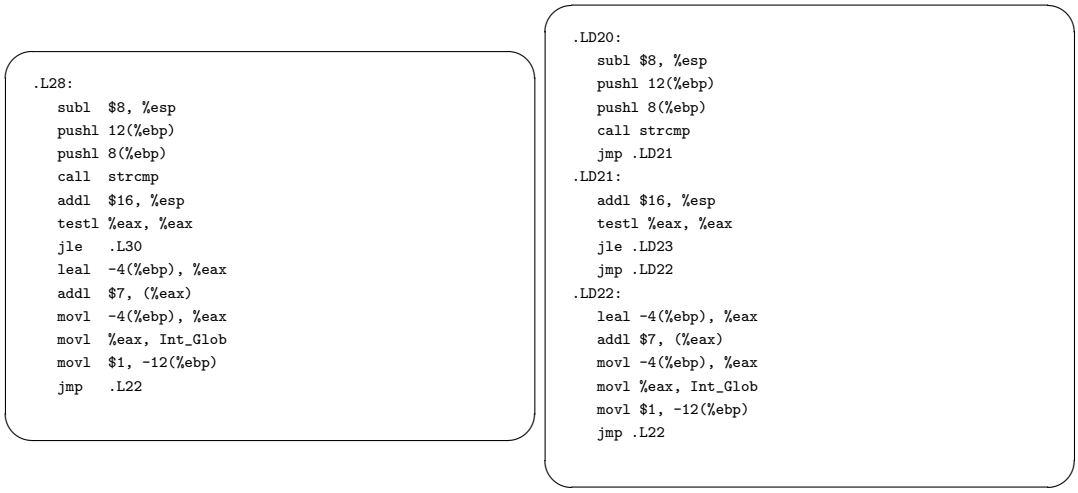
- 分岐命令 (**表 3**) の直後にラベルを挿入
- 連続したラベルの削除
- 次の基本ブロックへの無条件分岐命令の挿入 (実行順序保持のため)

分岐命令の直後の命令が無条件分岐命令の場合は，無条件分岐命令の直後にラベルを挿入し，次基本ブロックへの無条件分岐命令を挿入しない．

表 3　基本ブロックを分割する分岐命令

jmp	je	jz	jne	jnz	ja	jnbe	jae
jnb	jb	jnae	jbe	jna	jg	jnle	jge
jnl	jl	jnge	jle	jng	jc	jnc	jo
jno	js	jns	jop	jnp	jpe	jp	jcxz
jecxz	loop	loopz	loope	call	ret	iret	int
into	bound	enter					

図 6 に，アセンブリファイルを基本ブロックに分割した例を示す．左がオリジナルのアセンブリファイル，右が分割後のアセンブリファイルである．call 命令の直後に.LD21 ラベルとそのラベルへの無条件分岐命令が挿入され，jl 命令の直後にも同様に挿入されている．



```
.L28:
   subl  $8, %esp
   pushl 12(%ebp)
   pushl 8(%ebp)
   call  strcmp
   addl  $16, %esp
   testl %eax, %eax
   jle   .L30
   leal  -4(%ebp), %eax
   addl  $7, (%eax)
   movl  -4(%ebp), %eax
   movl  %eax, Int_Glob
   movl  $1, -12(%ebp)
   jmp   .L22
```

```
.LD20:
   subl $8, %esp
   pushl 12(%ebp)
   pushl 8(%ebp)
   call strcmp
   jmp .LD21
.LD21:
   addl $16, %esp
   testl %eax, %eax
   jle .LD23
   jmp .LD22
.LD22:
   leal -4(%ebp), %eax
   addl $7, (%eax)
   movl -4(%ebp), %eax
   movl %eax, Int_Glob
   movl $1, -12(%ebp)
   jmp .L22
```

図 6　基本ブロックの分割例

以下の基本ブロックは配置順序を変更できない.

- 制御命令のみで構成された基本ブロック

  制御命令は実命令が生成されないため，基本ブロックとして扱うことができない.

- 関数の先頭となる基本ブロック

  図 7 に示す例では，オリジナル (左側) から.LD24 を.Lfe5 の前に移動させる場合 (中央) は移動可能だが，Func_3 を.Lfe5 の前に移動させる (右側) と，Func_3 の定義前に存在することになりエラーとなる.

```
(オリジナル)
.L22:
   movl -12(%ebp), %eax
   leave
   ret
.Lfe5:
   .size Func_2,.Lfe5-Func_2
   .globl Func_3
   .type Func_3,@function
Func_3:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl 8(%ebp), %eax
   movl %eax, -4(%ebp)
   cmpl $2, -4(%ebp)
   jne .LD25
   jmp .LD24
.LD24:
   movl $1, -8(%ebp)
   jmp .L32
```

```
(.LD24 が.Lfe5 の前に移動 [エラーはなし])
.L22:
   movl -12(%ebp), %eax
   leave
   ret
.LD24:
   movl $1, -8(%ebp)
   jmp .L32
.Lfe5:
   .size Func_2,.Lfe5-Func_2
   .globl Func_3
   .type Func_3,@function
Func_3:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl 8(%ebp), %eax
   movl %eax, -4(%ebp)
   cmpl $2, -4(%ebp)
   jne .LD25
   jmp .LD24
```

```
(Func_3 が.Lfe5 の前に移動 [エラー])
.L22:
   movl -12(%ebp), %eax
   leave
   ret
Func_3:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl 8(%ebp), %eax
   movl %eax, -4(%ebp)
   cmpl $2, -4(%ebp)
   jne .LD25
   jmp .LD24
.Lfe5:
   .size Func_2,.Lfe5-Func_2
   .globl Func_3
   .type Func_3,@function
.LD24:
   movl $1, -8(%ebp)
   jmp .L32
```

図 7　アセンブル時のエラー

3.2.2　依存しない命令間の配置順序変更

基本ブロック内の命令も，命令間に依存関係がなければ配置順序を変更することができる．**図 8** に示す命令列があるとき，(ax := var1) と (dx := var2) の実行順序を変更しても機能等価となる.

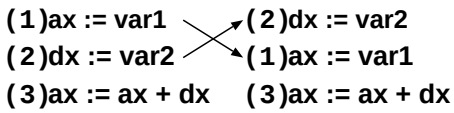


図 8　命令の順序

命令列 A の命令数を $|A|$ とするとき，命令間に依存関係がない場合の組み合わせ数は $|A|!$ である．しかし，依存関係を持つ命令列の場合の配置自由度は数式では表せないため，探索が必要となる．

以下のアルゴリズムに従い，依存関係を持つ組み合わせ数を数え上げる．L は探索ポインタが指すツリーノードの深さ (ルートノードは $L = 1$)，$A[i]$ は命令列 A の i 番目の要素を表すとする．

(1) $A[1]$ の要素をルートノードに設定する．

(2) 探索ポインタをルートノードに設定する．

(3) 探索ポインタが指すノードの命令列に，$A[L + 1]$ を付加して得られる組み合わせを列挙し，命令間の依存関係がなく機能等価な命令列のみを子ノードに接続する．

(4) $|A| = L$ ならばカウンタをインクリメントし，探索ポインタを親ノードへ移動する．

(5) $L = 1$ かつ未探索の子ノードがなければ終了する

(6) 探索ポインタを未探索の子ノードへ移動し (3) へ戻る．

上記アルゴリズムによる探索例を示す．命令列 A(a, b, c) の命令 a と命令 c に依存関係がある (命令 a よりも前に命令 c を配置できない) として探索を行う．探索ツリーを**図 9** に示す．

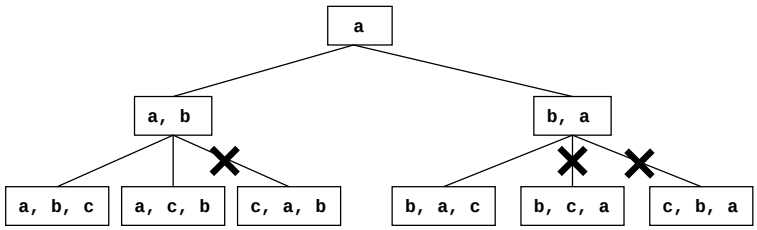


図 9　依存関係のある命令列の探索ツリー

はじめに，ツリールートには [a] を設定し (1)，探索ポインタをルートノードに設定する (2)．[a] に命令 b を付加するとき，2 命令間には依存関係はないので [a, b][b, a] の組み合わせを得ることができる．そのため，ルートノードから 2 つのノードを接続し (3)，探索リストを未探索のノード [a, b] に移動する (6)．ノード [a, b] に 命令 c を付加する場合，命令 a と命令 c 間には依存関係がある

ため，[c, a, b] は枝刈りされる．ノード [a, b, c][a, c, b] は $|A| = L$ であるので，カウンタをインクリメントし親ノードである [a, b] に戻る (4)．ノード [b, a] においては，[b, c, a][c, b, a] は機能等価な命令列とならないために，枝刈りされる．最後に，[b, a] の探索を終了し親ノード [a] に戻り探索を終了する (5)．

命令間の依存関係の判定方法について説明する．命令列では，依存関係を持つ要素は複数存在している．命令 x がもつ参照要素と格納要素の集合を $Ref[x], Sto[x]$ とするとき，以下の式条件を満たせば依存関係がないことを判定できる．

$$((Ref[x] \cap Sto[y]) \cup (Sto[x] \cap Ref[y]) \cup (Sto[x] \cap Sto[y])) = \phi$$

本研究では，アセンブリファイルから命令間の依存関係を解析し，機能等価な組み合わせに順序変更するためのフィルタプログラムを作成した．

Intel x86 命令セットでは，暗黙的に要素を参照・格納する命令が存在する．そのため，あらかじめ命令が暗黙的に扱う要素のリストを定義し，明示的に示された要素との和集合をとり，各命令の参照・格納要素とする．

実際の命令列では，複数の依存関係を持つ要素が存在する．本研究では，次の依存要素を対象として依存関係を解析する．ジャンプ命令については移動の対象としない．

- 引数
- 汎用レジスタ (eax, ebx, ecx, edx, esi, edi, ebp, esp)
- 命令ポインタレジスタ (eip)
- セグメントレジスタ (cs, ds, es ,fs, gs)
- EFLAGS レジスタの各フラグ
- FPU レジスタ (ST(0)-ST(7))
- x87 FPU ステータスレジスタの各フラグ
- メモリアクセス

4 実験

本研究では，測定対象として C 言語のベンチマークプログラムとフリーソフトウェアを使用した．それぞれのプログラムの情報を**表 4** に示す．フリーソフトウェアに与えたコンパイルオプションを**表 5** に示す．

各プログラムについて，4 種類の要素の配置順序変更による配置自由度と，手法適用によるオブジェクトファイルサイズの変化を測定する．さらに，ベンチマークプログラムの実行ファイルを生成し，要素の配置順序変更前後の性能値の変化を測定する．測定は，**表 6** に示す環境で行う．

n 個の要素が存在するときの配置自由度は $n! = O(n^n)$ であり，n の増加に伴い計算機での処理が困難になる．本研究では，PPS (Partial Permutation Scheme)[5] を使用する．要素を 6 個ごとのチャンクに分割し，各チャンクが $6! = 720$ 通りの配置自由度を持つものとして計算する．今回は，端数となる要素については扱わない．

表 4 プログラムの情報

Program Name	Note (Compile Option)	#Func.	#Line	Object file size [byte]
dhry_1.c	Dhrystone Pack. 1 (`-DHZ=100 -DTIME`)	6	385	7464
dhry_1.c	Dhrystone Pack. 1 (`-DHZ=100 -DTIME -O2`)	6	385	7064
dhry_2.c	Dhrystone Pack. 2	6	192	1936
dhry_2.c	Dhrystone Pack. 2 (`-O2`)	6	192	1600
linpackc.c	LINPACK Benchmark (`-DDP -DUNROLL`)	12	907	15856
linpackc.c	LINPACK Benchmark (`-DDP -DUNROLL -O2`)	12	907	11848
whetstone.c	Whetstone Benchmark	4	433	5984
whetstone.c	Whetstone Benchmark (`-O2`)	4	433	4096
ed/main.c	editor (-O)	26	1684	38704
ed/regex.c	edirot (-O)	26	5171	28428
bzip2.c	bzip2 archiver (-O2)	43	2103	31936
gzip.c	gzip archiver (-O)	23	1744	28132
gcc.c	compiler (-O2)	50	5840	77448
ldmain.c	linker (-O2)	20	1376	20512
ldlang.c	linker (-O2)	126	5525	48128

表 5 フリーソフトウェアのデフォルトコンパイルオプション

ed/main.c	`-DHAVE_CONFIG_H`
ed/regex.c	`-DHAVE_CONFIG_H`
bzip2.c	`-Wall -Winline -fomit-frame-pointer -fno-strength-reduce \` `-D_FILE_OFFSET_BITS=64`
gzip.c	`-DASMV -DSTDC_HEADERS=1 -DHAVE_UNISTD_H=1 -DDIRENT=1`
gcc.c	`-I. -I.. -I./config -I../include -DDEFAULT_TARGET_VERSION=\"2.95.3\" \` `-DDEFAULT_TARGET_MACHINE=\"i686-pc-linux-gnu\" -DHAVE_CONFIG_H -DIN_GCC`

ldmain.c	`-DHAVE_CONFIG_H -I. -D_GNU_SOURCE -I../bfd -I./../bfd \`
	`-I./../include -I./../intl -I../intl \`
	`-DLOCALEDIR="\"/usr/local/share/locale\""  -W -Wall -Wstrict-prototypes \`
	`-Wmissing-prototypes -DDEFAULT_EMULATION='"elf_i386"' \`
	`-DSCRIPTDIR='"/usr/local/i686-pc-linux-gnu/lib"' \`
	`-DTARGET='"i686-pc-linux-gnu"'`

ldlang.c	`-DHAVE_CONFIG_H -I. -D_GNU_SOURCE -I../bfd -I./../bfd -I./../include \`
	`-I./../intl -I../intl -DLOCALEDIR="\"/usr/local/share/locale\"" \`
	`-W -Wall -Wstrict-prototypes -Wmissing-prototypes \`
	`-DDEFAULT_EMULATION='"elf_i386"' \`
	`-DSCRIPTDIR='"/usr/local/i686-pc-linux-gnu/lib"' \`
	`-DTARGET='"i686-pc-linux-gnu"'`

表 6 測定環境

CPU	Intel Xeon 2.80 GHz
Mem	1 GB
OS	Redhat Linux 9
Compiler	gcc 2.95.3
	binutils 2.13

5 実験結果

5.1 機能等価なプログラムの配置自由度の測定

5.1.1 グローバル変数の配置自由度

グローバル変数は，実行時に配置されるメモリ領域の違いにより，外部変数，static 変数，初期化済み変数の 3 種類に分類できる．これらの変数は独立して配置順序変更を行えるため，各種変数の組み合わせ数を n_1, n_2, n_3 とすると，$720^{(\lfloor n_1/6 \rfloor + \lfloor n_2/6 \rfloor + \lfloor n_3/6 \rfloor)}$ 通りの配置自由度が存在する (PPS[5] $k = 6$)．

各プログラム中に存在するグローバル変数の数と配置自由度の組み合わせ数を**表 7** に示す．ベンチマークプログラムにおける最大配置自由度は 5.18×10^5 通り (dhry_1.c と linpackc.c)，フリーソフトにおける最大配置自由度は 5.22×10^{45} 通り (gcc.c) である．

表 7 グローバル変数の数と組み合わせ数

Program Name	Compile Option	#Global Var.	#Static Var.	#Initialized Var.	#Combination
dhry_1.c	`-DHZ=100 -DTIME`	13	0	1	5.18e+05
dhry_1.c	`-DHZ=100 -DTIME -O2`	13	0	1	5.18e+05
dhry_2.c		0	0	0	1.00e+00
dhry_2.c	`-O2`	0	0	0	1.00e+00
linpackc.c	`-DDP -DUNROLL`	0	13	0	5.18e+05
linpackc.c	`-DDP -DUNROLL -O2`	0	13	0	5.18e+05
whetstone.c		7	0	0	7.20e+02
whetstone.c	`-O2`	7	0	0	7.20e+02
ed/main.c	default option (-O)	24	1	27	7.22e+22
ed/regex.c	default option (-O)	0	1	4	1.00e+00
bzip2.c	default option (-O2)	22	1	2	3.73e+08
gzip.c	default option (-O)	25	0	29	7.22e+22
gcc.c	default option (-O2)	2	56	45	5.22e+45
ldmain.c	default option (-O2)	10	4	5	7.20e+02
ldlang.c	default option (-O2)	7	20	11	1.93e+14

5.1.2 ローカル変数の配置自由度

各関数ごとに定義されたローカル変数は，それぞれ独立して配置順序を変更できる．m 個の関数で定義されているローカル変数の数をそれぞれ $n_1, n_2 ... n_m$ 個とすると，プログラム全体で $720^{(\lfloor n_1/6 \rfloor + \lfloor n_2/6 \rfloor + ... + \lfloor n_m/6 \rfloor)}$ 通りの配置自由度が存在する．

今回は GCC を改造し，コンパイル時にスタックに配置されるローカル変数の数を数え上げる．

各プログラムのローカル変数の数と配置順序の組み合わせ数を**表 8** に示す．最適化したプログラムでは，最適化をかけないプログラムよりもローカル変数の数が少ない．GCC の最適化コンパイルでは，ローカル変数は擬似レジスタとして登録されることが原因である．ローカル変数による自由度は消滅したのではなく，レジスタの自由度に編入される．レジスタの配置自由度についての検討は今後の課題である．

表 8 ローカル変数の数と組み合わせ数

Program Name	Compile Option	#Func	Ave. Local Var. [var./func.]	#Combination
dhry_1.c	-DHZ=100 -DTIME	6	2.17	7.20e+02
	-DHZ=100 -DTIME -O2	6	0.33	1.00e+00
dhry_2.c		6	1.33	1.00e+00
	-O2	6	0.00	1.00e+00
linpackc.c	-DDP -DUNROLL	12	4.33	2.69e+11
	-DDP -DUNROLL -O2	12	0.00	1.00e+00
whetstone.c		4	7.00	2.69e+11
	-O2	4	0.00	1.00e+00
ed/main.c	default option (-O)	26	0.04	1.00e+00
ed/regex.c	default option (-O)	26	0.15	1.00e+00
bzip2.c	default option (-O2)	43	0.28	1.00e+00
gzip.c	default option (-O)	23	0.17	1.00e+00
gcc.c	default option (-O2)	50	0.04	1.00e+00
ldmain.c	default option (-O2)	20	0.10	1.00e+00
ldlang.c	default option (-O2)	126	0.01	1.00e+00

5.1.3 基本ブロックの配置順序の配置自由度

命令列は基本ブロックに分割することができる．適切なジャンプ命令を挿入して実行順序を保持すれば，配置順序を任意に変更することができる．本研究の条件では，基本ブロックの数を n，関数の数を f とするとき，$720^{\lfloor (n-f)/6 \rfloor}$ 通りの配置自由度が存在する．

関数と基本ブロックの数，基本ブロックの配置順序によって得られる組み合わせ数を**表 9** に示す．プログラムがもつ基本ブロック数の増加に従い，自由度も増加する．

表 9 基本ブロックの数と配置順序の組み合わせ数

Program Name	Compile Option	#Func	#Basic Block	#Combination
dhry_1.c	`-DHZ=100 -DTIME`	6	128	1.40e+57
dhry_1.c	`-DHZ=100 -DTIME -O2`	6	112	3.76e+48
dhry_2.c		6	46	1.39e+17
dhry_2.c	`-O2`	6	37	1.93e+14
linpackc.c	`-DDP -DUNROLL`	12	344	1.42e+157
linpackc.c	`-DDP -DUNROLL -O2`	12	291	2.74e+131
whetstone.c		4	101	5.22e+45
whetstone.c	`-O2`	4	77	1.94e+34
ed/main.c	default option (-O)	26	682	2.81e+311
ed/regex.c	default option (-O)	26	735	1.46e+337
bzip2.c	default option (-O2)	43	414	1.98e+174
gzip.c	default option (-O)	23	307	1.97e+134
gcc.c	default option (-O2)	50	1346	1.53e+617
ldmain.c	default option (-O2)	20	207	3.78e+88
ldlang.c	default option (-O2)	126	853	5.46e+345

5.1.4 命令順序の配置自由度

基本ブロック内の命令について，命令間に依存関係がなければ配置順序を変更することができる．依存関係を考慮した探索を行うことで，命令の配置自由度を数え上げることができる．

6 個の命令を 1 つのチャンクとすると，各チャンクごとの最大配置自由度は $6! = 720$ 通りである．しかし，依存関係により機能等価とならない命令列があるため，配置自由度が 720 通りよりも少なくなる可能性がある．

基本ブロック内の命令順序の入れ替えによる自由度を**表 10** に示す．

表 10 順序変更可能な命令の組み合わせ数

Program Name	Compile Option	#Func	#Basic Block	#Combination
dhry_1.c	`-DHZ=100 -DTIME`	6	128	1.47e+37
dhry_1.c	`-DHZ=100 -DTIME -O2`	6	112	4.23e+32
dhry_2.c		6	46	2.15e+10
dhry_2.c	`-O2`	6	37	8.06e+03
linpackc.c	`-DDP -DUNROLL`	12	344	6.81e+164
linpackc.c	`-DDP -DUNROLL -O2`	12	291	1.11e+128
whetstone.c		4	101	1.13e+43
whetstone.c	`-O2`	4	77	7.56e+31
ed/main.c	default option (-O)	26	682	2.10e+77
ed/regex.c	default option (-O)	26	735	4.60e+286
bzip2.c	default option (-O2)	43	414	5.17e+122
gzip.c	default option (-O)	23	307	3.82e+114
gcc.c	default option (-O2)	50	1346	7.18e+602
ldmain.c	default option (-O2)	20	207	1.61e+88
ldlang.c	default option (-O2)	126	853	2.39e+255

5.1.5 命令列表現の自由度の比較

各手法によるプログラムの配置自由度を**表 11** に示す．配置順序変更可能な基本ブロックや命令の配置自由度は変数の配置自由度よりも多いため，組み合わせ数に圧倒的な差が出ている．定義される変数の数が多くなることで配置自由度の増加が期待できる．しかし，測定対象の中では規模の大きなフリーソフトウェアにおいても，命令の配置自由度に比べ変数の配置自由度は少ない．

表 11 配置自由度の比較

Program Name	Program Name	Glboal Var.	Local Var.	Basic Block	Instruction
dhry_1.c	`-DHZ=100 -DTIME`	5.18e+05	7.20e+02	1.40e+57	1.47e+37
dhry_1.c	`-DHZ=100 -DTIME -O2`	5.18e+05	1.00e+00	3.76e+48	4.23e+32
dhry_2.c		1.00e+00	1.00e+00	1.39e+17	2.15e+10
dhry_2.c	`-O2`	1.00e+00	1.00e+00	1.93e+14	8.06e+03
linpackc.c	`-DDP -DUNROLL`	5.18e+05	2.69e+11	1.42e+157	6.81e+164
linpackc.c	`-DDP -DUNROLL -O2`	5.18e+05	1.00e+00	2.74e+131	1.11e+128
whetstone.c		7.20e+02	2.69e+11	5.22e+45	1.13e+43
whetstone.c	`-O2`	7.20e+02	1.00e+00	1.94e+34	7.56e+31
ed/main.c	default option (-O)	7.22e+22	1.00e+00	2.81e+311	2.10e+77
ed/regex.c	default option (-O)	1.00e+00	1.00e+00	1.46e+337	4.60e+286
bzip2.c	default option (-O2)	3.73e+08	1.00e+00	1.98e+174	5.17e+122
gzip.c	default option (-O)	7.22e+22	1.00e+00	1.97e+134	3.82e+114
gcc.c	default option (-O2)	5.22e+45	1.00e+00	1.53e+617	7.18e+602
ldmain.c	default option (-O2)	7.20e+02	1.00e+00	3.78e+88	1.61e+88
ldlang.c	default option (-O2)	1.93e+14	1.00e+00	5.46e+345	2.39e+255

5.2 情報密度

各要素の配置自由度を情報量に換算し，埋め込み可能な情報量とオブジェクトファイルサイズとの関係を調べた．オブジェクトファイルサイズと情報量，最小二乗法による近似直線をグラフに示す．グローバル変数によるものを **図 10**，ローカル変数によるものを **図 11**，基本ブロックによるものを **図 12**，基本ブロック内の命令の配置順序によるものを**図 13** にそれぞれ示す．

ローカル変数の配置順序によるもの以外は，全て相関関係が認められた．最小二乗法で近似した結果，グローバル変数の配置自由度による情報密度は 0.02%，基本ブロックの配置順序と基本ブロック内の命令による情報密度はともに 0.3%であった．

多くのローカル変数が擬似レジスタとして登録されたために，配置自由度が消えてしまった。本研究では，擬似レジスタとして割り当てられた変数は対象としていないため配置自由度を取り出すことができなかった．擬似レジスタの配置自由度の検討は，今後の課題である．

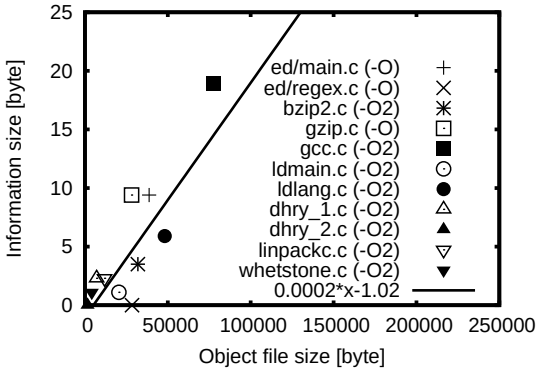


図 10　グローバル変数のの配置順序による情報密度

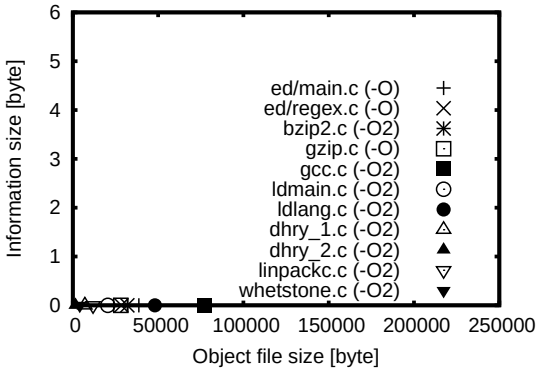


図 11　ローカル変数の配置順序による情報密度

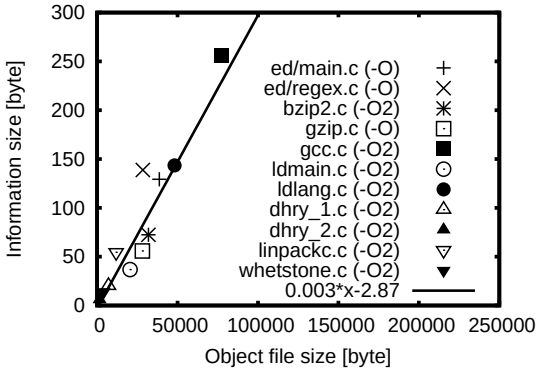


図 12　基本ブロックの配置順序による情報密度

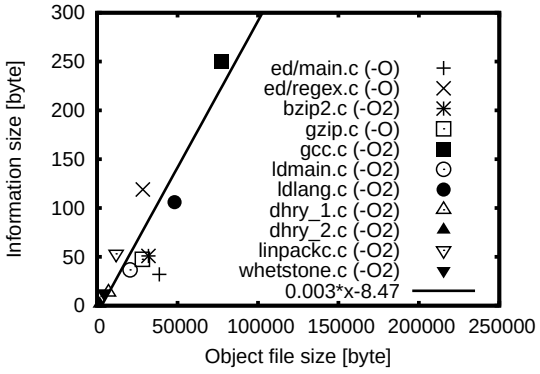


図 13　命令の配置順序による情報密度

5.3 実行性能の測定

要素の配置順序を変更することで，プログラムの性能に影響が現れる可能性がある．そこで，3種類のベンチマークプログラム (dhrystone, linpack, whetstone) を用い，オリジナルプログラムを100回実行した最大・最小・平均性能値と，要素をランダムな配置順序に並べ換えたプログラム100個の最大・最小・平均性能値を比較する．

グローバル変数，ローカル変数，基本ブロック，順序変更可能な命令列の配置順序を変更したプログラムの性能値を，それぞれ**表12**，**表13**，**表14**，**表15**に示す．

表12 グローバル変数の順序変更による実行性能の変化

Program Name	Optimize	#Loop	Original (100 trials)			Changed (100 random order)		
			max.	ave.	min.	max.	ave.	min.
dhry [Dhrystones/s]	-DHZ=100 -DTIME	1.00e+09	2.88e+06	2.85e+06	2.66e+06	3.01e+06	2.89e+06	2.67e+06
	-DHZ=100 -DTIME -O2	1.00e+09	4.18e+06	4.13e+06	3.66e+06	4.15e+06	4.06e+06	3.80e+06
linpackc [Kflops]	-DDP -DUNROLL	NTIMES=10000	2.38e+05	2.36e+05	2.33e+05	2.38e+05	2.36e+05	2.34e+05
	-DDP -DUNROLL -O2	NTIMES=10000	5.16e+05	5.07e+05	4.95e+05	5.16e+05	5.07e+05	4.95e+05
whetstone [MIPS]		1.00e+06	1.18e+02	1.17e+02	1.16e+02	1.18e+02	1.17e+02	1.17e+02
	-O2	1.00e+06	2.58e+02	2.57e+02	2.49e+02	2.58e+02	2.57e+02	2.55e+02

表13 ローカル変数の順序変更による実行性能の変化

Program Name	Option	#Loop	Original (100 trials)			Changed (100 random order)		
			max.	ave.	min.	max.	ave.	min.
dhry [Dhrystones/s]	-DHZ=100 -DTIME	1.00e+09	2.88e+06	2.85e+06	2.66e+06	2.95e+06	2.91e+06	2.84e+06
	-DHZ=100 -DTIME -O2	1.00e+09	4.18e+06	4.13e+06	3.66e+06	4.13e+06	4.07e+06	3.62e+06
linpackc [Kflops]	-DDP -DUNROLL	NTIMES=10000	2.38e+05	2.36e+05	2.33e+05	2.38e+05	2.36e+05	2.33e+05
	-DDP -DUNROLL -O2	NTIMES=10000	5.16e+05	5.07e+05	4.95e+05	5.10e+05	5.04e+05	4.97e+05
whetstone [MIPS]		1.00e+06	1.18e+02	1.17e+02	1.16e+02	1.18e+02	1.17e+02	1.13e+02
	-O2	1.00e+06	2.58e+02	2.57e+02	2.49e+02	2.58e+02	2.57e+02	2.56e+02

表14 基本ブロックの順序変更による性能値変化

Program Name	Option	#Loop	Original (100 trials)			Changed (100 random order)		
			max.	ave.	min.	max.	ave.	min.
dhry [Dhrystones/s]	-DHZ=100 -DTIME	1.00e+09	2.88e+06	2.85e+06	2.66e+06	2.89e+06	2.76e+06	2.49e+06
	-DHZ=100 -DTIME -O2	1.00e+09	4.18e+06	4.13e+06	3.66e+06	4.03e+06	3.88e+06	3.60e+06
linpackc [Kflops]	-DDP -DUNROLL	NTIMES=10000	2.38e+05	2.36e+05	2.33e+05	2.37e+05	2.34e+05	2.27e+05
	-DDP -DUNROLL -O2	NTIMES=10000	5.16e+05	5.07e+05	4.95e+05	5.19e+05	5.00e+05	4.85e+05
whetstone [MIPS]		1.00e+06	1.18e+02	1.17e+02	1.16e+02	1.18e+02	1.17e+02	1.17e+02
	-O2	1.00e+06	2.58e+02	2.57e+02	2.49e+02	2.58e+02	2.57e+02	2.56e+02

表15 命令の順序変更による性能値変化

Program Name	Option	#Loop	Original (100 trials)			Changed (100 random order)		
			max.	ave.	min.	max.	ave.	min.
dhry [Dhrystones/s]	-DHZ=100 -DTIME	1.00e+09	2.88e+06	2.85e+06	2.66e+06	3.09e+06	2.92e+06	2.82e+06
	-DHZ=100 -DTIME -O2	1.00e+09	4.18e+06	4.13e+06	3.66e+06	4.17e+06	4.00e+06	3.61e+06
linpackc [Kflops]	-DDP -DUNROLL	NTIMES=10000	2.38e+05	2.36e+05	2.33e+05	2.42e+05	2.36e+05	2.22e+05
	-DDP -DUNROLL -O2	NTIMES=10000	5.16e+05	5.07e+05	4.95e+05	5.13e+05	5.03e+05	4.92e+05
whetstone [MIPS]		1.00e+06	1.18e+02	1.17e+02	1.16e+02	1.18e+02	1.18e+02	1.17e+02
	-O2	1.00e+06	2.58e+02	2.57e+02	2.49e+02	2.60e+02	2.57e+02	2.52e+02

ベンチマークプログラムの平均性能値の割合を**表 16** に示す．基本ブロックの順序変更による性能値は 0.0-6.1%であり，他の手法よりも大きな性能低下が見られる．これは，基本ブロックの実行順序を保持するために挿入したジャンプ命令によるオーバーヘッドであると考える．

表 16　ベンチマークプログラムの性能変化の比較

Program	Compile Option	Global	Local	Basic Block	Inst.
dhry	-DHZ=100 -DTIME	1.014	1.021	0.968	1.025
dhry	-DHZ=100 -DTIME -O2	0.983	0.985	0.939	0.969
linpackc	-DDP -DUNROLL	1.000	1.000	0.992	1.000
linpackc	-DDP -DUNROLL -O2	1.000	0.994	0.986	0.992
whetstone		1.000	1.000	1.000	1.009
whetstone	-O2	1.000	1.000	1.000	1.000

5.4　プログラムサイズの測定

各手法を適用したプログラムのオブジェクトファイルサイズを測定した．さらにベンチマークプログラムの実行ファイルを生成し，.text，.data，.bss セクションのサイズを測定した．.text セクションは実行する命令列が格納されたメモリイメージ，.data セクションは初期値が指定されたデータのメモリイメージ，.bss セクションは未初期化データのメモリイメージである．ただし，.bss セクションは実行時にメモリ上に配置されるセクションであるため，実行ファイル中に実体は存在しない．

グローバル変数の配置順序変更を行った際のサイズの変化について，オブジェクトファイルサイズの変化を**表 17** に，.text，.data，.bss セクションのサイズ変化をそれぞれ**表 18**，**表 19**，**表 20** に示す．適用後の実行ファイル中の.bss セクションサイズに変化がみられた．これは，変数が配置される際に行われるアライメント処理によって，使用されないメモリ領域が発生するからである．.bss セクションは実行ファイル中では実体のないセクションであるため，オブジェクトファイルサイズには影響しない．

表 17 グローバル変数の順序変更による オブジェクトファイルのサイズ変化

Program Name	Compile Option	Original	Order changed object file size		
			max.	ave.	min.
dhry_1.c	-DHZ=100 -DTIME	7464	7464	7464.0	7464
dhry_1.c	-DHZ=100 -DTIME -O2	7064	7064	7064.0	7064
dhry_2.c		1936	1936	1936.0	1936
dhry_2.c	-O2	1600	1600	1600.0	1600
linpackc.c	-DDP -DUNROLL	15856	15856	15856.0	15856
linpackc.c	-DDP -DUNROLL -O2	11848	11848	11848.0	11848
whetstone.c		5984	5984	5984.0	5984
whetstone.c	-O2	4096	4096	4096.0	4096
ed/main.c	default option (-O)	38704	38704	38704.0	38704
ed/regex.c	default option (-O)	28428	28428	28428.0	28428
bzip2.c	default option (-O2)	31936	31936	31936.0	31936
gzip.c	default option (-O)	28132	28132	28132.0	28132
gcc.c	default option (-O2)	77448	77448	77448.0	77448
ldmain.c	default option (-O2)	20512	20544	20512.6	20512
ldlang.c	default option (-O2)	48128	48128	48128.0	48128

表 18 グローバル変数の順序変更による .text セクションのサイズ変化

Program Name	Compile Option	Original	Order changed text size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	6139	6139	6139.0	6139
	-DHZ=100 -DTIME -O2	5510	5510	5510.0	5510
linpackc	-DDP -DUNROLL	12147	12147	12147.0	12147
	-DDP -DUNROLL -O2	8287	8287	8287.0	8287
whetstone		4921	4921	4921.0	4921
	-O2	3608	3608	3608.0	3608

表 19 グローバル変数の順序変更による .data セクションのサイズ変化

Program Name	Compile Option	Original	Order changed data size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	288	288	288.0	288
	-DHZ=100 -DTIME -O2	284	284	284.0	284
linpackc	-DDP -DUNROLL	284	284	284.0	284
	-DDP -DUNROLL -O2	280	280	280.0	280
whetstone		312	312	312.0	312
	-O2	304	304	304.0	304

表 20 グローバル変数の順序変更による .bss セクションのサイズ変化

Program Name	Compile Option	Original	Order changed bss size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	10316	10332	10307.1	10284
	-DHZ=100 -DTIME -O2	10316	10332	10308.2	10284
linpackc	-DDP -DUNROLL	646312	646344	646344.0	646344
	-DDP -DUNROLL -O2	646312	646344	646344.0	646344
whetstone		116	140	122.7	108
	-O2	116	140	123.7	108

ローカル変数の配置順序変更を行った際のサイズの変化について，オブジェクトファイルサイズの変化を**表 21** に，.text，.data，.bss セクションのサイズ変化をそれぞれ**表 22**，**表 23**，**表 24** に示す．オブジェクトファイルサイズとセクションサイズともに変化はなかった．

表 21　ローカル変数の順序変更による オブジェクトファイルのサイズ変化

Program Name	Compile Option	Original	Order changed object file size		
			max.	ave.	min.
dhry_1.c	-DHZ=100 -DTIME	7464	7464	7464.0	7464
	-DHZ=100 -DTIME -O2	7064	7064	7064.0	7064
dhry_2.c		1936	1936	1936.0	1936
	-O2	1600	1600	1600.0	1600
linpackc.c	-DDP -DUNROLL	15856	15856	15856.0	15856
	-DDP -DUNROLL -O2	11848	11848	11848.0	11848
whetstone.c		5984	5984	5984.0	5984
	-O2	4096	4096	4096.0	4096
ed/main.c	default option (-O)	38704	38704	38704.0	38704
ed/regex.c	default option (-O)	28428	28428	28428.0	28428
bzip2.c	default option (-O2)	31936	31936	31936.0	31936
gzip.c	default option (-O)	28132	28132	28132.0	28132
gcc.c	default option (-O2)	77448	77448	77448.0	77448
ldmain.c	default option (-O2)	20512	20512	20512.0	20512
ldlang.c	default option (-O2)	48128	48128	48128.0	48128

表 22　ローカル変数の順序変更による .text セクションのサイズ変化

Program Name	Compile Option	Original	Order changed text size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	6139	6139	6139.0	6139
	-DHZ=100 -DTIME -O2	5510	5510	5510.0	5510
linpackc	-DDP -DUNROLL	12147	12147	12147.0	12147
	-DDP -DUNROLL -O2	8287	8287	8287.0	8287
whetstone		4921	4921	4921.0	4921
	-O2	3608	3608	3608.0	3608

表 23　ローカル変数の順序変更による .data セクションのサイズ変化

Program Name	Compile Option	Original	Order changed data size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	288	288	288.0	288
	-DHZ=100 -DTIME -O2	284	284	284.0	284
linpackc	-DDP -DUNROLL	284	284	284.0	284
	-DDP -DUNROLL -O2	280	280	280.0	280
whetstone		312	312	312.0	312
	-O2	304	304	304.0	304

表 24　ローカル変数の順序変更による .bss セクションのサイズ変化

Program Name	Compile Option	Original	Order changed bss size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	10316	10316	10316.0	10316
	-DHZ=100 -DTIME -O2	10316	10316	10316.0	10316
linpackc	-DDP -DUNROLL	646312	646312	646312.0	646312
	-DDP -DUNROLL -O2	646312	646312	646312.0	646312
whetstone		116	116	116.0	116
	-O2	116	116	116.0	116

基本ブロックの配置順序変更を行った際のサイズの変化について，オブジェクトファイルサイズの変化を**表 25** に，.text，.data，.bss セクションのサイズ変化をそれぞれ**表 26**，**表 27**，**表 28** に示す．.text セクションサイズの変化は実行順序保持のために挿入したジャンプ命令によるものである．

表 25　基本ブロックの順序変更による オブジェクトファイルのサイズ変化

Program Name	Compile Option	Original	Order changed object file size		
			max.	ave.	min.
dhry_1.c	-DHZ=100 -DTIME	7464	7496	7472.3	7432
	-DHZ=100 -DTIME -O2	7064	7068	7044.3	7004
dhry_2.c		1936	2064	2035.0	1992
	-O2	1600	1692	1663.8	1640
linpackc.c	-DDP -DUNROLL	15856	16524	16426.1	16332
	-DDP -DUNROLL -O2	11848	12480	12423.0	12352
whetstone.c		5984	6200	6137.6	6104
	-O2	4096	4348	4297.4	4252
ed/main.c	default option (-O)	38704	39824	39747.5	39664
ed/regex.c	default option (-O)	28428	29644	29538.4	29420
bzip2.c	default option (-O2)	31936	32224	32138.6	32064
gzip.c	default option (-O)	28132	28452	28379.7	28324
gcc.c	default option (-O2)	77448	79176	79033.6	78920
ldmain.c	default option (-O2)	20512	20864	20832.6	20800
ldlang.c	default option (-O2)	48128	49472	49380.5	49280

表 26　基本ブロックの順序変更による .text セクションのサイズ変化

Program Name	Compile Option	Original	Order changed text size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	6139	6243	6202.2	6115
	-DHZ=100 -DTIME -O2	5510	5534	5492.6	5454
linpackc	-DDP -DUNROLL	12147	12787	12681.2	12595
	-DDP -DUNROLL -O2	8287	8911	8834.5	8751
whetstone		4921	5113	5057.8	5017
	-O2	3608	3816	3772.5	3720

表 27　基本ブロックの順序変更による .data セクションのサイズ変化

Program Name	Compile Option	Original	Order changed data size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	288	288	288.0	288
	-DHZ=100 -DTIME -O2	284	284	284.0	284
linpackc	-DDP -DUNROLL	284	284	284.0	284
	-DDP -DUNROLL -O2	280	280	280.0	280
whetstone		312	312	312.0	312
	-O2	304	304	304.0	304

表 28　基本ブロックの順序変更による .bss セクションのサイズ変化

Program Name	Compile Option	Original	Order changed bss size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	10316	10316	10316.0	10316
	-DHZ=100 -DTIME -O2	10316	10316	10316.0	10316
linpackc	-DDP -DUNROLL	646312	646312	646312.0	646312
	-DDP -DUNROLL -O2	646312	646312	646312.0	646312
whetstone		116	116	116.0	116
	-O2	116	116	116.0	116

基本ブロック内の命令列で，依存しない命令の配置順序変更を行った際のサイズの変化について，オブジェクトファイルサイズの変化を**表 29**に，.text，.data，.bss セクションのサイズ変化をそれぞれ**表 30**，**表 31**，**表 32** に示す．オブジェクトファイルサイズとセクションサイズともに，変化はなかった．

表 29　命令の順序変更による オブジェクトファイルのサイズ変化

Program Name	Compile Option	Original	Order changed object file size		
			max.	ave.	min.
dhry_1.c	-DHZ=100 -DTIME	7464	7464	7464.0	7464
	-DHZ=100 -DTIME -O2	7064	7064	7064.0	7064
dhry_2.c		1936	1936	1936.0	1936
	-O2	1600	1600	1600.0	1600
linpackc.c	-DDP -DUNROLL	15856	15856	15856.0	15856
	-DDP -DUNROLL -O2	11848	11848	11848.0	11848
whetstone.c		5984	5984	5984.0	5984
	-O2	4096	4096	4096.0	4096
ed/main.c	default option (-O)	38704	38704	38704.0	38704
ed/regex.c	default option (-O)	28428	28428	28428.0	28428
bzip2.c	default option (-O2)	31936	31936	31936.0	31936
gzip.c	default option (-O)	28132	28132	28132.0	28132
gcc.c	default option (-O2)	77448	77448	77448.0	77448
ldmain.c	default option (-O2)	20512	20512	20512.0	20512
ldlang.c	default option (-O2)	48128	48128	48128.0	48128

表 30　命令の順序変更による .text セクションのサイズ変化

Program Name	Compile Option	Original	Order changed text size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	6139	6139	6139.0	6139
	-DHZ=100 -DTIME -O2	5510	5510	5510.0	5510
linpackc	-DDP -DUNROLL	12147	12147	12147.0	12147
	-DDP -DUNROLL -O2	8287	8287	8287.0	8287
whetstone		4921	4921	4921.0	4921
	-O2	3608	3608	3608.0	3608

表 31　命令の順序変更による .data セクションのサイズ変化

Program Name	Compile Option	Original	Order changed data size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	288	288	288.0	288
	-DHZ=100 -DTIME -O2	284	284	284.0	284
linpackc	-DDP -DUNROLL	284	284	284.0	284
	-DDP -DUNROLL -O2	280	280	280.0	280
whetstone		312	312	312.0	312
	-O2	304	304	304.0	304

表 32　命令の順序変更による .bss セクションのサイズ変化

Program Name	Compile Option	Original	Order changed bss size		
			max.	ave.	min.
dhry	-DHZ=100 -DTIME	10316	10316	10316.0	10316
	-DHZ=100 -DTIME -O2	10316	10316	10316.0	10316
linpackc	-DDP -DUNROLL	646312	646312	646312.0	646312
	-DDP -DUNROLL -O2	646312	646312	646312.0	646312
whetstone		116	116	116.0	116
	-O2	116	116	116.0	116

セクションサイズに変化があった手法について検討する．グローバル変数と基本ブロックの配置順序変更によるセクションサイズとオリジナルプログラムのセクションサイズの割合を**表 33** に示す．グローバル変数のの配置順序変更では，.bss セクションのサイズが 99.9-106.6%となっていた．これは，アライメント処理によってメモリ上の変数間に使用されない領域が生じるためである．基本ブロックの配置順序変更では，.text セクションのサイズが，オリジナルの 99.7-106.6 %となっていた．これは，実行順序保持のためのジャンプ命令が挿入されているためである．

表 33　順序変更後のプログラムセクション平均サイズ割合 (グローバル変数と基本ブロック順序のみ抜粋)

Program	Compile Option	Global			Basic Block		
		.text	.data	.bss	.text	.data	.bss
dhry		1.000	1.000	0.999	1.010	1.000	1.000
dhry	-O2	1.000	1.000	0.999	0.997	1.000	1.000
linpackc		1.000	1.000	1.000	1.044	1.000	1.000
linpackc	-O2	1.000	1.000	1.000	1.066	1.000	1.000
whetstone		1.000	1.000	1.058	1.028	1.000	1.000
whetstone	-O2	1.000	1.000	1.066	1.046	1.000	1.000

オリジナルのオブジェクトファイルサイズと，各手法適用による平均オブジェクトファイルサイズの比を**表 34** に示す．グローバル変数の順序変更では，.bss セクションのサイズに変化があったが，オブジェクトファイルサイズには影響はない．これは，.bss セクションが実行時にメモリ上に展開されるセクションで，実行形式ファイルやオブジェクトファイル中には存在していないからである．

基本ブロックの順序変更によるオブジェクトファイルサイズの変化は 99.7-105.1%であった．これは，.text セクションの変化による影響であると考える．

表 34 オブジェクトファイルサイズの平均サイズ割合

Program	Compile Option	Global	Local	Basic Block	Inst.
dhry_1.c	`-DHZ=100 -DTIME`	1.000	1.000	1.001	1.000
dhry_1.c	`-DHZ=100 -DTIME -O2`	1.000	1.000	0.997	1.000
dhry_2.c		1.000	1.000	1.051	1.000
dhry_2.c	`-O2`	1.000	1.000	1.040	1.000
linpackc.c	`-DDP -DUNROLL`	1.000	1.000	1.036	1.000
linpackc.c	`-DDP -DUNROLL -O2`	1.000	1.000	1.049	1.000
whetstone.c		1.000	1.000	1.026	1.000
whetstone.c	`-O2`	1.000	1.000	1.049	1.000

6　おわりに

本研究ではプログラムの命令列が持つ順序変更可能な要素の検討と，プログラムが持つ配置自由度を測定した．さらに，配置自由度の変更に伴い発生する，プログラムサイズや性能のオーバーヘッドを定量的に測定した．

命令に起因する配置自由度 (基本ブロックや基本ブロック内の命令順序) のほうが，変数の順序変更による配置自由度よりも多くの自由度を持つことがわかった．これは，プログラム中に存在する変数の数よりも命令の方が多く，かつ順序変更可能であることが理由である．

本研究で検討した要素は，それぞれ独立して配置順序を変更することができる．さらに，レジスタの配置自由度など，本研究で対象としなかった自由度も存在する．1 手法だけでは最大でも 0.3%の情報密度であったが，手法を併用することにより情報密度の向上が期待できる．

今後の課題として，これらの要素の配置自由度を使用した電子透かしへの応用の検討が必要である．

付録

A.1 外部変数の順序変更

外部変数はグローバル変数の 1 種であり，プログラム全体から参照することができる．外部変数の実行時アドレスは，リンクする全てのオブジェクトファイルで定義された変数情報が揃わなければ決定することができない．そのため，リンカが持つ外部変数アドレス決定部を改造することで配置順序変更を実現する．

本研究では，GNU binutils 2.13 に含まれる ld を使用した．

A.1.1 追加機能

外部変数の配置順序を変更するため，ld (リンカ) に以下の機能を追加する．

- `global-order` オプションを追加し，変数の配置順序に対応した順列番号を指定する
- 外部変数の順序を `global-order` によって指定された配置順序に並べ換える

A.1.1.1 パラメータの追加

`global-order` オプションは，外部変数の配置順序を任意に変更するためのものである．`global-order` を = で区切り，右辺に順列番号を 16 進数で指定する．順列番号とは，辞書式順列の登録番号である．

`global-order` オプションを指定した時の実行例を**図 14** 示す．gcc の-Wl オプションは，リンカに対して与えるパラメータを指定するものである．

本研究では，変数を 6 つごとのチャンクに分け，各チャンクごとに配置順序を変更する．そのため 6! = 720 通りの配置順序を 16 進数 3 桁に分割して指定する．図 14 の例では引数として 0x0af = 175 を指定しているので，辞書式順序による順列は (1, 3, 2, 0, 5, 4) となる．

```
$ gcc -Wl,--global-order=0af source.c
```

図 14 global-order オプションの使用例

オプションの追加するため，lexsup.c を修正する．各オプションには識別番号が定義されているため，global-order 用の識別番号を新たに定義する．global-order 用の識別番号 OPTION_GLOBALORDER の定義部分を**図 15** に示す．OPTION_NOSTDLIB は「リンク時にシステム標準のライブラリを使用しない」オプションを示す識別子であり，識別番号の定義において最大のものである．(識別番号の最大値+1) の値は定義されていない番号であるため，新規オプションのための識別番号として使用することができる．

```
#define OPTION_GLOBALORDER    (OPTION_NOSTDLIB + 1)
```

図 15　global-order 用の識別番号の定義

オプションの定義情報は，ld_options (ld_option 構造体配列変数) の宣言時に設定する．ld_options は ld_option 構造体の定義に従い，オプションの名前，引数の有無，ヘルプでの出力文字列，オプション識別番号を指定する．**図 16** に，global-order オプションの定義情報を示す．global-order は引数を必要とするので，required_argument を指定する．

```
static struct ld_option[] = {
(略)
  { {"global-order", required_argument, NULL, OPTION_GLOBALORDER},
 '\0', N_("=ORDER_VALUE"), N_("Set Global Order from ORDER_VALUE"),
 EXACTLY_TWO_DASHES}
};
```

図 16　ld_options 構造体変数への追加

リンカプログラム中では，global-orderによって指定された順列番号を参照する必要がある．そこで，引数の処理を行う関数 parse_args を改造し，global-orderので指定した順列番号を示す文字列へのポインタを変数 global_order_string に格納する．追加した部分を**図 17** に示す．optc は ld_option 構造体で指定した識別番号，optarg は global-order で指定された順列番号を示す char 型ポインタである．この global_order_string は，次節で使用する．

```
      switch (optc)
   {
(略)
   case OPTION_GLOBALORDER:
          /* Set Global Order DEFSYM */
          global_order_string = optarg;
          break;
(略)
   }
```

図 17　指定されたパラメータの処理 (parse_args 内)

A.1.1.2　外部変数の登録とアドレス決定

オブジェクトファイルで定義された外部変数は，リンカによって実行形式ファイルに結合される．ELF ファイル形式では，外部変数として定義されたシンボルを「グローバルコモンシンボル」と呼ぶ．リンク処理時に必要な全てのシンボルはハッシュ表に登録されている．グローバルコモンシンボルの登録は，ldland.c の ld_common 関数内で行われる．ld_common 関数内の処理を以下に示す．

(1)　ハッシュ表をたどり，グローバルコモンシンボルを検索する

(2)　グローバルコモンシンボルを発見したら，シンボル表に登録する

リンカに sort-common オプションを与えた場合，シンボルの登録順序はシンボルのサイズに依存したものとなる．そのため今回は，sort-common と global-order は両立しないものとする．

外部変数の実行時アドレスは，シンボル表への登録時に決定する．オリジナルの ld では，ハッシュ表からグローバルコモンシンボルを発見後すぐにシンボル表へ登録しているため，配置順序はハッシュ表の登録順序に依存する．

そのため，1 回目の探索においてシンボル表に登録すべきエントリを記録しておく．2 回目の探索で，記録したエントリを任意の順序でシンボル表に登録すれば，ハッシュ表の登録順序に依存することなく外部変数の配置順序を変更することができる．

以上を踏まえ，ld_common 関数の処理を次のように変更する．

(1)　ハッシュ表をたどり，グローバルコモンシンボルを検索する
(2)　グローバルコモンシンボル発見したら，エントリリストに登録する
(3)　エントリリストを任意の順序に変更する
(4)　変更した順序に従い，シンボル表に登録する

グローバルコモンシンボルの検索部分を**図 18** に示す．bfd_link_hash_traverse は，ハッシュテーブルのエントリを探索するための関数である．第 1 引数に指定したハッシュテーブルを探索し，各エントリを第 2 引数で示すポインタのエントリ処理関数に渡す．第 3 引数は，エントリを処理する際に必要な関数への情報であり，任意に指定できる．

```
/* グローバル変数となるシンボルを，ハッシュ表から取得する */
memset(&ginfo, 0, sizeof(struct gval_info));
memset(&new_ginfo, 0, sizeof(struct gval_info));
bfd_link_hash_traverse (
         link_info.hash,
         get_global_variables,
         (PTR) &ginfo);
bfd_link_hash_traverse (
         link_info.hash,
         get_global_variables,
         (PTR) &new_ginfo);
```

図 18　グローバルコモンシンボルの検索部分 (ld_common 内)

エントリ処理関数として，get_global_variables を作成した．これは，ハッシュエントリがグローバルコモンシンボルであれば，gval_info 構造体 (**図 19**) に情報を格納する関数である．bfd_link_hash_traverse において，エントリ処理用の関数を get_global_variables と指定すれば，ハッシュ表に登録されたグローバルコモンシンボルのエントリを取得することができる．

図 20 に関数 get_global_variables のソースファイルを示す.

```
/*
 * グローバル変数情報を格納するための構造体
 */
struct gval_info
{
   unsigned long              gval_count;
   struct bfd_link_hash_entry **hash;
};
```

図 19　gval_info 構造体の定義

```
/*
 *  グローバル変数となるシンボルがあれば，ハッシュエントリへの
 *  ポインタを取得し，struct gval_info 内のポインタに追加する
 */
static boolean
get_global_variables (h, info)
   struct bfd_link_hash_entry *h;
   PTR info;
{
  struct gval_info *pInfo=(struct gval_info*)info;
  int iCount;

  if (h->type != bfd_link_hash_common)
    return true;

  iCount =pInfo->gval_count+1;
  pInfo->hash =
     (struct bfd_link_hash_entry**)
        realloc(pInfo->hash,
   sizeof(struct bfd_link_hash_entry)*iCount);
  pInfo->hash[iCount-1] = h;
  pInfo->gval_count =iCount;

  return  true;
}
```

図 20　get_global_variable

以上の処理で取得したグローバルコモンシンボルエントリを，任意の順序に並べ換える．global-order で指定した順列番号に従いエントリを並べ替えるプログラムを**図 21** に示す．GetOrder は，順列番号に並べ換えた情報を返す関数である．

```
/* 文字数値へ変換 */
i = (ginfo.gval_count / 6 + 1) * 2;
pValueString = (char*) malloc(i * sizeof(char) + 1);
memset(pValueString, '0', sizeof(char) * i);
if(global_order_string != NULL){
   len = strlen(global_order_string);
  if(i < len){
    memcpy(pValueString, global_order_string, i);
  }else{
    memcpy(&pValueString[i-len],
           global_order_string,
           strlen(global_order_string));
  }
}
pValueString[i] = '\0';

/* global_order_string から順序を得る */
chank = ginfo.gval_count / 6;
mini  = ginfo.gval_count % 6;

memset(szValue, 0, sizeof(char) * 5);
for(c = 0;c < (unsigned int)chank+1;c++) {
  iOffset = strlen(pValueString) - (c+1) * 2;
  memcpy(szValue, &pValueString[iOffset], 2);
  szValue[4] = '\0';
  sscanf(szValue, "%x", &iValue);
  if(iOffset == 0){ iOrderNum = mini; }
  else            { iOrderNum = 6; }

  /* 数値から順列を得る */
  GetOrder(&pOrder, iOrderNum, iValue);

  /* ハッシュエントリの順序を任意に変更する */
  for(i = 0;i < (unsigned int)iOrderNum;i++){
    new_ginfo.hash[pOrder[i]+c*6] = ginfo.hash[i+c*6];
  }
  free(pOrder);
}
```

図 21　グローバルコモンシンボルの並べ換え (ld_common 内)

グローバルコモンシンボルを並べ替えた順序にシンボルテーブルに登録する処理を行う部分を**図 22** に示す．lang_one_common 関数は，シンボルテーブルにグローバルコモンエントリを登録する関数である．第 1 引数にエントリへのポインタ，第 2 引数に必要な情報を指定する．これにより，グローバルコモンシンボルがシンボルテーブルに登録され，プログラム実行時に配置するアドレスが決定する．

機種によっては，変数の開始アドレスが切り上げられて設定されることがある．これをアライメントといい，切り上げられた領域は未使用領域となる．これによって，データセクション自体のサイズが変化することがある．

```
/*
 * 任意順序に並べ換えた順に変数位置の割り当てを行う
 */
for(i = 0;i < ginfo.gval_count;i++)
{
  lang_one_common(new_ginfo.hash[i], (PTR)NULL);
}
```

図 22　シンボルテーブルへの登録 (ld_common 内)

A.2　ローカル変数の順序変更

C 言語におけるローカル変数とは，関数の内部で定義されている変数のことである．ローカル変数は関数の内部でのみアクセス可能であり，関数から制御が離れると値は保持されない．Intel x86 命令セットの GCC は，ローカル変数をスタック上かレジスタに配置する．

スタック上にローカル変数を配置する場合，実行時アドレスは一定にならない．Intel x86 命令セットでは，EBP レジスタに格納されたアドレス値と，各変数に定義されたオフセット値の和で参照する．GCC がローカル変数をあらかじめスタック上に配置する場合は，構文解析時に各変数に対しオフセット値を決定する．本研究では GCC を改造し，各変数で決定されたオフセット値を変更できるようにする．そのため，以下の機能を実装した．

- gcc (ドライバプログラム) に，`--local-order` オプション (省略形 -fL) を追加
- cc1 (C コンパイラ) に，-fL オプションを追加
- cc1 で，ローカル変数のオフセット値の変更機能

A.2.1　機能追加

A.2.1.1　gcc のパラメータ追加

gcc は，コンパイラやアセンブラ，リンカなどへ適切な引数を渡すためのドライバプログラムであり，C コンパイラは cc1 である．

C コンパイラ (cc1) に-fL オプションを追加するためには，ドライバプログラム (gcc) にも-fL オプションを定義する必要がある．これは，gcc が「コンパイラに対する無効なオプション」としてエラーを返すことを防ぐためである．

`--local-order` オプションは，gcc 内部では省略形である-fL オプションに置き換えられる．gcc が受け取ることのできるパラメータは，option_map 構造体の配列変数で定義する．option_map の追加項目を**図 23** に示す．

```
struct option_map option_map[] =
{
(略)
   {"--local-order", "-fL", 0},
(略)
};
```

図 23　option_map の追加項目 (gcc.c)

A.2.1.2 cc1 のパラメータ追加

gcc は適切な言語のコンパイラやアセンブラ，リンカ等を起動するためのドライバプログラムである．C コンパイラを改造するためには，実体である cc1 の改造を行わなければならない．

cc1 のパラメータ処理は，toplev.c 内の main 関数で行っている．今回は，-fL オプションを f オプション (-fcall-used, -ffixed 等) の 1 つとして実装する．

-fL オプションは，ローカル変数の順序変更を行うためのランダムシードを指定できるようにする．ランダム初期化関数 srand を呼び出し，ランダムシードを設定する．

toplev.c への追加部分を **図 24** に示す．変数 p は 与えられたオプションで -f の次の文字を指すポインタである．-fL が指定された場合は 'L' を指す．local_variable_test は，-fL が指定されたことを示すフラグで，function.c で宣言している．ランダムシードが指定されない場合は，time 関数を用いて設定する．

```
else if (*p == 'L')
{
  p++;
  if(*p == '\0')
  {
    srand(time(NULL));
  }else
  {
    srand(atol(p));
  }
  local_variable_order = 1;
}
```

図 24 toplev.c の改造部分

A.2.1.3 ローカル変数の配置順序変更

ローカル変数の登録は構文解析時に行われる．本研究では，関数先頭での変数定義後の文を処理する前に，登録された変数の配置順序を変更する．今回は，level=1 (関数の先頭で定義された変数) のみを扱うこととする．

C コンパイラの構文解析は c-parse.y (c-parse.c) で行われている．yyn = 166 の時に start_decl 関数が呼び出され，変数を擬似レジスタかスタック上に配置する．

命令文を解析する yyn = 86 が初めて呼び出されたとき，スタック上に配置された変数の配置順序を変更する．

A.2.1.4 ローカル変数順序変更関数の実装

スタック上に配置されたローカル変数の情報は，temp_slot 構造体のリストに登録されている．変数 temp_slots はリストの先頭要素を指し，メンバ変数 next は次の要素を指す．temp_slot 構造体リストをたどる，宣言されたローカル変数の情報を得ることができる．

ローカル変数が宣言されたときに関数 temp_stack_assign が呼ばれ，ローカル変数の適切な配置場所 (スタック上かレジスタ) を決定する．スタック上に作成したローカル変数の RTX(Register Transfer Language) は，temp_slot 構造体の slot メンバ変数に格納する．

ローカル変数のオフセット値を-4 から-12 に変更した場合の RTX の例を**図 25** に示す．

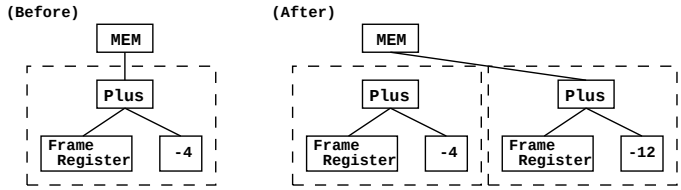


図 25　ローカル変数を示す RTX の変更

ここでは，plus ノードに接続されたオフセット値ノードの値を変更するのではなく，新たに plus ノードを作成して接続を変更している．これは，オフセット値ノードが別の項目からも参照されているため，値を変更することにより無関係な項目にまで影響を与えてしまうことを防ぐためである．

図 26, **図 27** に，実装した stack_order_change のソースファイルを示す．stack_slot_list は以下の処理を行う．

(1)　任意の順序を生成する．
(2)　temp_slots のリストをたどり，変数の一覧を配列に格納する．
(3)　各変数に対して，順序に従い次の処理を行う
　(a)　フレームオフセット値を再計算する．
　(b)　変数の RTX のオフセット部分を取替える．
　(c)　stack_slot_list に，変数を示す RTX を接続する．

```
stack_order_change(void){
  int i;
  struct temp_slot *p, *p2;
  int    slot_size;

  /* Stack Order Flag is OFF */
  if(local_variable_order == 0){
     return;
  }
  temp_slots_order++;
  if(temp_slots_order == 2 && temp_slot_level == 1) {
   printf("[%20s] (%d, %d)\n",
      current_function_name, temp_slots_order, temp_slot_level);
  }

  /* Slot count */
  for(p = temp_slots, slot_size = 0;p != NULL;p = p->next, slot_size++)
     ;
  if(slot_size <= 0) return;

  if(temp_slots_order == 2 && temp_slot_level == 1) {
   printf(" local variables in stack = %d, level=%d\n",
      slot_size, temp_slot_level);
  }

  /* Swap Offset */
  if(temp_slots != NULL && temp_slots->next != NULL) {
    int j, local_frame_offset, align;
    rtx *ppTempSlots, tmp;
    struct temp_slot **ppSlots, *tmp_slots;
    int   *pOrder, *pSemiOrder;

    /* Get order */
    pOrder     = (int*) malloc(sizeof(int) * slot_size);
    for(i = 0;i < slot_size;i++) {
      pOrder[i] = slot_size - 1 - i;
    }
    for(i = 0;
        (temp_slot_level == 1 && temp_slots_order == 2) &&
        i <= slot_size-PPS;
        i+=PPS) {
      /* Semi Order */
      j = rand() % 720;
      GetOrder(&pSemiOrder, PPS, j);
```

図 26　関数 stack_order_change の実装 (1)

```
      for(j = 0;j < 6;j++) {
        pSemiOrder[j] = slot_size - 1 - (i+pSemiOrder[j]);
      }
      for(j = 0;j < 6;j++) {
        pOrder[i+j] = pSemiOrder[j];
      }
      free(pSemiOrder);
      pSemiOrder = NULL;
    }
    /* Copy slot rtx */
    ppTempSlots = (rtx*)malloc(sizeof(rtx) * slot_size);
    ppSlots = (struct temp_slot**)
        malloc(sizeof(struct temp_slot) * slot_size);

    /* Store temp_slots to array */
    for(i = 0, p = temp_slots;p != NULL;i++, p = p->next) {
      ppTempSlots[i] = XEXP(p->slot, 0);
      ppSlots[i] = p;
    }

    local_frame_offset = 0;
    for(i = 0;i < slot_size;i++) {
      j = pOrder[i];
      local_frame_offset -= ppSlots[j]->size;
#ifdef FRAME_GROWS_DOWNWARD
      local_frame_offset = FLOOR_ROUND (local_frame_offset, 4);
#else
      local_frame_offset = CEIL_ROUND (local_frame_offset, 4);
#endif
      ppSlots[j]->slot->fld[0].rtx =
         plus_constant(
            ppSlots[j]->slot->fld[0].rtx->fld[0].rtx, local_frame_offset);
      ppSlots[j]->base_offset = local_frame_offset;
      stack_slot_list = gen_rtx_EXPR_LIST(
            VOIDmode, ppSlots[j]->slot, stack_slot_list);
     }

   /* Sort List Structure */
   temp_slots = NULL;
   for(i = 0;i < slot_size;i++){
      ppSlots[pOrder[i]]->next = temp_slots;
      temp_slots = ppSlots[pOrder[i]];
   }
   free(pOrder);
   free(ppSlots);
   free(ppTempSlots);
  }
}
```

図 27　関数 stack_order_change の実装 (2)

A.3 gas の文法

GNU as は，アセンブリファイルの文法に AT&T 記法を採用している．そのため，gcc は x86 の仕様書[6] の記述 (Intel 記法) とは異なるアセンブリファイルを出力する．本章では，Intel 記法と AT&T 記法の相違点について解説する[3]．

A.3.1 命令名

一部の命令の名前が変更されている．**表 35** に異なる命令名の対応を示す．

さらに，扱うデータ型に従い命令の末尾に文字を付加する．**表 36** に付加する型に対応する文字を示す．例えば，32bit 整数演算を行う add 命令は addl と記述する．

表 35 Intel 記法と AT&T 記法の命令の違い

Intel 記法	AT&T 記法
cbw	cbtw
cwde	cwtl
cwd	cwtd
cdq	cltd

表 36 扱う型により命令に付加する文字

型	付加文字	型	付加文字
8 bit 整数	b	32 bit 浮動小数	s
16 bit 整数	w	64 bit 浮動小数	l
32 bit 整数	l	80 bit 浮動小数	t

A.3.2 要素記述

AT&T 記法では，即値とレジスタ名の先頭に記号を付加する．**表 37** に，即値とレジスタをスタックに push するときの例を示す．即値の先頭には$を，レジスタ名の先頭には%を付加する．

表 37 Intel 記法と AT&T 記法の要素の先頭に付加する記号

要素	Intel 記法	AT&T 記法
即値	push 4	push $4
レジスタ	push eax	push %eax

A.3.3 ニモニックの記述順序

Intel 記法では，第 1 ニモニックに転送先要素を指定する．しかし，AT&T 記法では最終ニモニックに転送先要素を指定する．add 命令と imul 命令の記述の違いを**表 38** に示す．Intel 記法では ‘dest, source’ となっているが，AT&T 記法では，‘source, dest’ となっている．

表 38　Intel 記法と AT&T 記法のニモニック順序の違い

Intel 記法	AT&T 記法
add eax, 4	addl $4, %eax
imul eax, edx, 19	imull %edx, $19, %eax

A.3.4 メモリ参照

以下に，メモリ参照の構文を示す．

```
Intel : section:[base + index*scale + disp]
AT&T  : section:disp(base, index, scale)
```

Intel 記法では [ ] の内部で演算を行う必要があったが，AT&T 記法では引数に適切な値を指定すればよい．引数は適宜省略することができる．

メモリ参照記述の例を**表 39** に示す．

表 39　メモリ参照方法の違い

Intel 記法	AT&T 記法
[ebp - 4]	-4(%ebp)
[foo + eax*4]	foo(,%eax,4)
[foo]	foo(,1)
gs:foo	%gs:foo

A.3.5 依存要素

Intel x86 命令セットにおける命令間の依存要素は複数存在する．本節では，本研究で検討する依存する要素について説明する．参照・格納要素は引数とし

て与えられるが，命令によっては暗黙的に参照・格納が行われる場合がある．

A.3.5.1 引数

加算命令の「足す数」と「足される数」のように，引数を必要とする命令がある．各命令は，要素を参照する引数，格納する引数，またはその両方の引数を持つ．

A.3.5.2 汎用レジスタ

Intel x86 プロセッサでは 8 つの汎用レジスタがあり，各種演算に使用される．ただし，ESP レジスタのみはスタックポインタを保持するために使用されるため，ユーザが自由に使用することはできない．

A.3.5.3 EFLAGS レジスタ

EFLAGS レジスタは，制御フラグやシステムフラグが格納されるレジスタである．各ビットごとに意味を持ち，演算結果に従いビットの状態を変更する．

A.3.5.4 FPU レジスタ

FPU レジスタは，浮動小数命令の演算結果を格納するためのものである．アセンブリファイル中では ST で表現され，ST(0) から ST(7) までの 8 つをもつ．アセンブリ中で ST とのみ記述された場合は，FPU ステータスレジスタ中の TOP で示されたレジスタ番号となる．TOP の値は静的解析では値を特定できないため，ST と記述されている場合は全ての FPU レジスタとして扱う．

A.3.5.5 FPU ステータスレジスタ

FPU ステータスレジスタは，現在の FPU の状態を示す．例外やオーバーヘッドフラグ，条件コード，スタックトップ等の状態が定義されている．

A.3.5.6 メモリアクセス

メモリへのアクセスにはレジスタの値を使用することができる．静的解析では，実行時のレジスタの値の値を解析することは困難である．

図 28 に示す例では，2，3 番目の命令は記述は違うもののレジスタの値により同じ領域を指していることが分かる．そのためメモリへのアクセスを依存要素として扱わなければならない．

```
movl $-40, $edx
movl $30, -40(%ebp)
addl $10, %edx(%ebp)
```

図 28 メモリアクセス時に同じアドレスを参照する例

参　考　文　献

1) Collberg, C. S. and Thomborson, C.: Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection, *IEEE Transactions on Software Engineering*, Vol. 28, No. 8, pp. 735–746 (2002).

2) Davidson, R. I. and Myhrvold, N.: Method and system for generating and auditing a signature for a computer program, *USPatent*, No. 5,559,884 (1996).

3) Elsner, D., Fenlason, J. and friends: *Using as : The GNU Assembler*, `http://www.gnu.org/software/binutils/manual/gas-2.9.1/html_mono/as.html` (1994).

4) Hattanda, K. and Ichikawa, S.: The Evaluation of Davidson's Digital Signature Scheme, *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E87-A, No. 1, pp. 224–225 (2004).

5) Ichikawa, S., Chiyama, H. and Akabane, K.: Redundancy in 3D Polygon Models and Its Application to Digital Signature, *Journal of WSCG*, Vol. 10, No. 1, pp. 225–232 (2002).

6) Intel: IA-32 インテル アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル中巻 (2003).

7) Venkatesan, R., Vazirani, V. and Sinha, S.: A Graph Theoretic Approach to Software Watermarking, *Lecture Notes in Computer Science*, Vol. 2137, pp. 157–168 (2001).

8) A. V. エイホ, R. セシィ, J. D. ウルマン: コンパイラ II : 原理・技法・ツール, サイエンス社 (1990).

9) 中村直己, 西垣正勝, 曽我正和, 田窪昭夫: プログラムの冗長化に関する検討, 電子情報通信学会技術研究報告, Vol. 100, No. 213, pp. 41–48 (2000).

10) 門田暁人: Java watermarking tools, *http://se.aist-nara.ac.jp/jmark/*.

11) 門田暁人, 高田義広, 鳥居宏次: ループを含むプログラムを難読化する方法の提案, 電子情報通信学会論文誌, Vol. J80-D-I, No. 7, pp. 644–652 (1997).

12) 門田暁人, 飯田元, 松本健一, 鳥居宏次: Java クラスファイルに対する電子透かし法, 情報処理学会論文誌, Vol. 41, No. 11, pp. 3001–3009 (2000).