

9 最適化

- ♣ 最適化とは何か
- ♣ コンパイラはどんなことをしてくれるのか

9.1 コード最適化とは

最適化 (optimization) とは, 生成するコードを **改良** すること

(注: 本当に **最適** にするわけではない)

1. 実行時の **速度**
2. 実行時の **使用記憶量**
3. **コード** のサイズ

分類 (計算機の情報を使うレベル)

1. 機械 **独立**: 種々の計算機に共通の最適化 (**中間言語** を対象に行われる)
2. 機械 **依存**: 命令セットや計算機の実装まで考慮した最適化

分類 (使う情報のレベル)

1. **局所** 的: プログラムの一部だけを見て, その部分だけを最適化する
2. **大局** 的: プログラム全体を見て, データの流れなどを分析し, 最適化する

9.2 コード最適化の手法

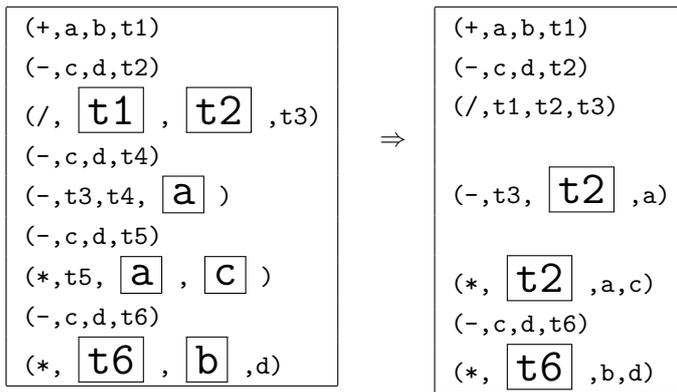
1. 定数の畳み込み (constant folding)

<pre>int a = 3*2; ... double b = a+16.5; unsigned int x = 65530+10;</pre>	⇒	<pre>int a = 6; ... double b = 22.5; unsigned int x = 4 ; /* int が 16bit の場合 */</pre>
---------------------------------------------------------------------------	---	----------------------------------------------------------------------------------------------

2. **共通部分式削除** (common subexpression elimination)

<pre>a = (a+b)/(c-d)-(c-d); c = (c-d)*a; d = (c-d)*b;</pre>	⇒	<pre>r = c-d; a = (a+b)/ R - R ; c = R *a; d = (c-d) *b;</pre>
-------------------------------------------------------------	---	--------------------------------------------------------------------------------------------

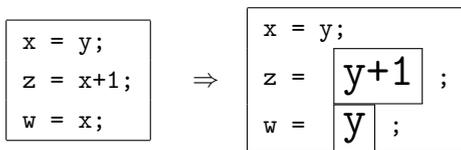
四つ 組ではこの最適化が行ないやすい



共通部分式の抽出が難しいと思われる場合は、括弧でくくってやるとよいことがある

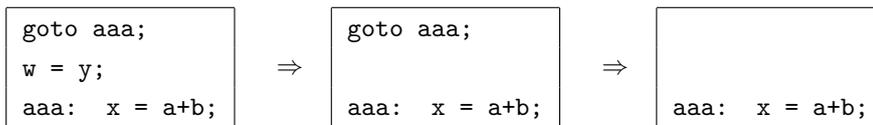
$$a = b/(c+d) - a*(b+c+d); \Rightarrow a = b/(c+d) - a*(b+(c+d));$$

3. 複写伝播 (copy propagation)

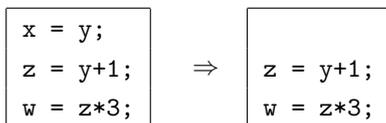


4. 不要コードの除去 (dead code elimination)

実行されない命令を除去

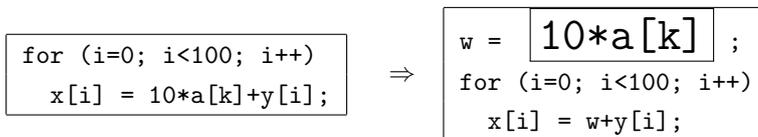


以後参照されない変数への代入を除去

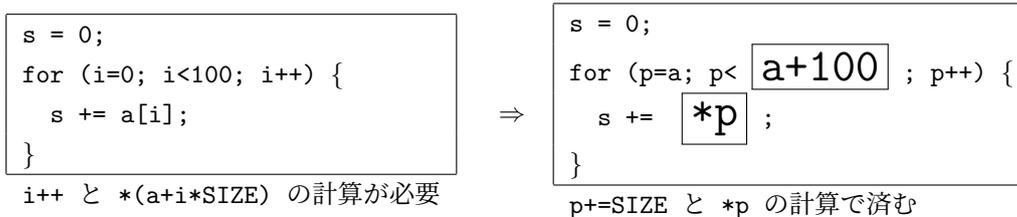


5. コードの移動 (code motion)

特にループの外への移動が有効



6. ループ制御変数の変換



7. ループ展開 (loop unrolling)

```

for (i=0; i<100; i++) {
    a[i] = x[i]+y[i];
}
⇒
for (i=0; i<100; i+=2) {
    a[i] = x[i]+y[i];
    a[i+1] = x[i+1]+y[i+1];
}

```

複数の演算器を同時に使える計算機に有効

8. **インライン** 展開 (**inline expansion**)

```

void push(int d)
{sp++; s[sp]=d;}

int pop()
{int d=s[sp]; sp--; return d;}

int main(void)
{
    ...
    a = b * c;
    push(a);
    push(b);
    d = e + pop();
    ...
}
⇒
int main(void)
{
    ...
    a = b * c;
    sp++; s[sp]=a;
    sp++; s[sp]=b;
    w = s[sp]; sp--;
    d = e + w;
    ...
}

```

9. 演算子の強さの軽減

R1 に 2 をかける (乗算命令)

⇒ R1 に R1 を足す (加算命令)

⇒ R1 を 1 ビット左シフトする (シフト命令)

R1 に 10 をかける (乗算命令)

⇒ R1 を **1 ビット左シフト** したものと **3 ビット左シフト** したものを足す (加算命令)

除算

⇒ **逆数** の乗算

10. 変数のレジスタへの割り付け

変数の生存期間, 使用される可能性を解析し, これをもとに決定

11. 命令順序の入れ換え

特に命令パイプライン方式のプロセッサでは有効

```

lw    $1,24($9) ; $1=Mem[$9+24]
add   $3,$1,$2  ; $3=$1+$2
add   $6,$4,$5  ; $6=$4+$5
⇒
lw    $1,24($9)
add   $6,$4,$5
add   $3,$1,$2

```

☆ 命令パイプライン方式による実行

(a) 命令間のデータ依存がない場合

1: add \$10,\$1,\$2	IF	ID \$1,\$2	EX +	MM	WB \$10			
2: sub \$11,\$3,\$4		IF	ID \$3,\$4	EX -	MM	WB \$11		
3: add \$12,\$5,\$6			IF	ID \$5,\$6	EX +	MM	WB \$12	
4: sub \$13,\$7,\$8				IF	ID \$7,\$8	EX -	MM	WB \$13

(b) 命令間に依存がある場合

1: add \$3,\$1,\$2	IF	ID \$1,\$2	EX +	MM	WB \$3					
2: sub \$11,\$3,\$4		IF	■ stall	■ stall	ID \$3,\$4	EX -	MM	WB \$11		
3: add \$12,\$5,\$6					IF	ID \$5,\$6	EX +	MM	WB \$12	
4: sub \$13,\$7,\$8						IF	ID \$7,\$8	EX -	MM	WB \$13

(c) 命令の入れ換えを行った場合

1: add \$3,\$1,\$2	IF	ID \$1,\$2	EX +	MM	WB \$3			
3: add \$12,\$5,\$6		IF	ID \$5,\$6	EX +	MM	WB \$12		
4: sub \$13,\$7,\$8			IF	ID \$7,\$8	EX -	MM	WB \$13	
2: sub \$11,\$3,\$4				IF	ID \$3,\$4	EX -	MM	WB \$11

☆ 【注】「バイパス回路」を持つプロセッサでは, (b) の依存関係は stall 無しで実行可能.

1: add \$3,\$1,\$2	IF	ID \$1,\$2	EX +	MM	WB \$3			
2: sub \$11,\$3,\$4		IF	ID ↓,\$4	EX -	MM	WB \$11		
3: add \$12,\$5,\$6			IF	ID \$5,\$6	EX +	MM	WB \$12	
4: sub \$13,\$7,\$8				IF	ID \$7,\$8	EX -	MM	WB \$13

☆ 「バイパス回路」を用いても, ロード命令 (lw) 直後のレジスタ参照には 1 サイクルの stall が生じるので, 命令入れ換えが有効

1: lw \$1,24(\$8)	IF	ID \$8	EX +	MM Mem	WB \$1			
2: add \$3,\$1,\$2		IF	■ stall	ID ↓,\$2	EX +	MM	WB \$3	
3: add \$6,\$4,\$5				IF	ID \$4,\$5	EX +	MM	WB \$6



Nagisa ISHIURA