IA32/SSE2 用マルチメディア指向並列化
実証用プロトタイプの改良とリリース

外部仕様書

2005 年 3 月

株式会社 管理工学研究所

目 次

第 1 章 はじめに **1**

第 2 章 SIMD 並列化処理の概要 **2**

2.1 SIMD 並列化処理の流れ 2

2.1.1 SIMD 並列化固有の処理の実行 2

2.1.2 SIMD 並列化に失敗した場合等の処理 3

2.1.3 実行時のチェックコードの挿入 3

第 3 章 マシン依存のデータ **4**

3.1 BOP 4

3.1.1 BOP パターン 4

3.1.2 BOP の登録 4

3.2 Bone 6

3.2.1 Bone の登録 6

3.2.2 Bone パターン 6

3.2.3 Bone 情報 7

3.2.4 補助パターン 8

3.3 TMD 9

3.3.1 IA32 用 TMD 9

3.3.2 TMD の変更手順 9

第 4 章 SIMD 並列化個々の処理 **13**

4.1 論理演算を用いた IF 変換 13

4.1.1 目的と概要 13

4.1.2 基本ブロックの結合 13

4.1.3 論理演算を用いた IF 変換の制約 14

4.1.4 論理演算の組み立て 16

4.1.5 アルゴリズム 17

4.2 DAG 生成 19

4.2.1 概要 19

4.2.2 アルゴリズム 19

4.3 範囲解析 21

4.3.1 上向きの範囲解析 21

4.3.2 下向きの範囲解析 21

4.3.3 サイズの書き換え 21

4.4 基本命令切り出し 22

4.4.1 「切り出し」の方法 22

4.5 SIMD 命令の組み立て 23

4.5.1 メモリアクセス命令の組み立て 23

4.5.2　一般命令の組み立て 23
4.6　命令列の整列化 25
4.7　制約の適用 26
4.8　SIMD レジスタ割り当て 27
4.8.1　SIMD レジスタ割り当て処理 28
4.9　Cleanup 29

図 目 次

4.1 論理演算を用いた IF 変換と基本ブロックの結合 14
4.2 除去する辺 14
4.3 制御構造の制約 15
4.4 論理演算組み立てアルゴリズム 17
4.5 論理演算を用いた IF 変換アルゴリズム 18
4.6 DAG 生成アルゴリズム 20

表 目 次

4.1 SIMD レジスタの割り当ての例 28

第1章　はじめに

近年のマイクロプロセッサのほとんどは, レジスタ分割式ショートベクタ処理用 SIMD (Single Instruction Multiple Data stream) 型のマルチメディア処理向け拡張命令 (以下 SIMD 命令) を持っている. また, プロセッサメーカやサードパーティからは, SIMD 命令を使ったコードを生成するコンパイラも提供されている.

しかしながら, これらのコンパイラが自動的に SIMD 命令を生成する場面は非常に限定されており, 従来から研究されているベクトル化の技術をそのまま持ってきたに過ぎないと言ってよい. ベクトル化だけでは対応できない SIMD 命令の使い方もいろいろあるため, 多くの場合, SIMD 命令を使うためには, プログラマは, SIMD 命令対応のコンパイラで拡張されたデータ型を明示的に使用し, 命令に対応する関数 (intrinsic) を明示的に呼び出さなければならない.

IA32/SSE2 用マルチメディア指向並列化実証用プロトタイプ (以下本システム) では, [2] で提案されている方式を COINS コンパイラインフラストラクチャ [3, 1] を用いて実装した.

本仕様書では以下の記号を使う. また, 一般的な数学の記号も使用する.

= この記号で結ばれる変数や集合が等しいことを示す. 例えば a=b と書いた場合は a と b が等しい. ただし, C プログラムを記述している場合などは代入を示す場合がある.

← この記号で結ばれる変数や集合に関して, 記号の左側の変数や集合に, 右側の変数や集合を代入することを示す. 例えば a←b と書いた場合は a に b を代入する.

⟶ この記号で結ばれるものの間にエッジが存在することを示す. 例えば a⟶b は, a を始点とし b を終点とするエッジを表す.

以下, 第 2 章で SIMD 並列化の概要を述べ, 第 3 章で SIMD 並列化で用いるターゲットマシン依存のデータについて述べる. そして第 4 章では, SIMD 並列化で行う処理の詳細について述べる.

第2章　SIMD 並列化処理の概要

SIMD 並列化でもちいた手法は，パターンマッチによって基本命令を組み合わせ，SIMD 命令に対応する PARALLEL 式を生成することである．

このとき，組み合わせられる基本命令の種類および組み合わせ方は，プロセッサ毎に異なるものである．そこで，組み合わせにもちいられるパターンは，プロセッサ毎に定義できるようにした．これらのパターンの記述の仕方については，第 3 章で述べる．

本章の以下の項では，SIMD 並列化処理全体の流れを述べる．

2.1　SIMD 並列化処理の流れ

SIMD 並列化処理は，それ固有の処理とその処理が失敗した場合への対応や生成された SIMD コードの制約から派生するコードの挿入をおこなう処理からなる．

SIMD 並列化処理は，SIMD コードを生成する処理をおこなうが，その途中で SIMD コードの生成に失敗した場合，あるいは生成された SIMD コードが十分な効果を挙げることができないと判定された場合，SIMD 並列化の固有の処理を適用する以前のコードを出力する．また，生成された SIMD コードを実行するのが適切か，オリジナルコード (非 SIMD コード) を実行するのが適切かの判断を条件分岐の形で挿入する．

2.1.1　SIMD 並列化固有の処理の実行

本 SIMD 並列化は，基本ブロック内の L 式の列を対象とし，SIMD 命令に対応する PARALLEL 式の列を出力する．その処理は，つぎに挙げる処理 (L 式列から L 式列への変換) を順次適用することによりおこなわれる．

1. If 変換
2. DAG 生成
3. 範囲解析
4. 基本命令の切り出し
5. メモリアクセス命令の組み立て
6. 一般命令の組み立て
7. 命令列の整列化
8. 制約の適用
9. SIMD レジスタ割り当て
10. 無用なコピー文の除去

なお，これらの処理の途中で不具合が発見された場合，例外を発生させる．後述の「SIMD 並列化に失敗した場合等の処理」では，この例外をとらえてその処理をおこなう．

2.1.2 SIMD 並列化に失敗した場合等の処理

SIMD 並列化に失敗した場合，および生成されたコード列を評価した結果 SIMD 並列化の効果が現れないと判断された場合，SIMD 化される以前のコードを出力する．

現時点では，SIMD 並列化の効果があらわれない場合として，つぎの場合を実装しているが，判定条件の多様化と詳細化が必要である:

- SIMD レジスタが単独で現れる場合

  すなわち，メモリから SIMD レジスタにロードされた値が，SIMD 命令で使用されない場合である．

2.1.3 実行時のチェックコードの挿入

実行時に，条件チェックをおこない，条件がみたされていなければ，非 SIMD 化コードを実行し，条件がみたされていれば，SIMD 化コードを実行するように，条件分岐コードを挿入する．

ただし，前述の「SIMD 並列化に失敗した場合等の処理」がおこなわれた場合には，本処理をおこなわない．

現時点では，つぎの 2 つのチェックをおこなっているが，必要に応じて拡張可能である．

1. アラインメントのチェック
2. メモリへのアクセスの重なりのチェック

第3章　マシン依存のデータ

3.1　BOP

DAG 化されたＬ式の列から基本命令として切り出すためのパターンを BOP(Basic Order Pattern) として登録する．

3.1.1　BOP パターン

登録するパターンはマシン共通の基本的な演算やメモリのアクセスパターンのほか，１つの命令として切り出したいマシン依存の特殊な命令のパターンも登録する．パターンは LIR の部分式を (HOLE n <Ltype>) で置き換えたものである．

- 基本演算

```
(ADD I8 (HOLE 1 I8) (HOLE 2 I8))
(BAND I32 (HOLE 1 I32) (HOLE 2 I32))
(TSTGES I16 (HOLE 1 I16) (HOLE 2 I16))
(BNOT I32 (HOLE 1 I32))
(CONVIT I8 (HOLE 1 I32))
```

- メモリのアクセスパターン

```
(MEM I8 (ADD I32 (HOLE 1 I32) (HOLE 2 I32)))
```

- マシン依存の特殊なパターン

```
(BAND I32 (BNOT I32 (HOLE 1 I32)) (HOLE 2 I32))
```

3.1.2　BOP の登録

パターンはマシン依存の特殊な命令も含むので，LirBopList_x86.java の templateList に関数 mkBop のパラメータとして登録する．

```
templateList[n]=mkBop("パターン");
```

パターンの検索は templateList の添字に関し昇順で行われるため，より詳細なパターンを先に登録する．例えば，`~A & B` を `~A(A’`とする) と `A’ & B` の２つの命令として切り出すのでなく，IA32/SSE2 の PANDN(Logical AND NOT) 命令を使い１つの命令とするためには，
パターン`"(BAND I32 (BNOT I32 (HOLE 1 I32)) (HOLE 2 I32))"`
をパターン`"(BAND I32 (HOLE 1 I32) (HOLE 2 I32))"` や`"(BNOT I32 (HOLE 1 I32))"`
よりも先に登録しなくてはいけない．

1. マシン依存の特殊なパターン
2. メモリのアクセスパターン
3. 基本演算

の順に登録する.

3.2 Bone

基本命令の組み合わせにもちいられるパターンを bone という.

3.2.1 Bone の登録

Bone パターンはマシン依存の特殊な命令も含むので，LirBoneList_x86.java の templateList に関数 mkBone のパラメータとして Bone 情報と共に登録する.

```
templateList[n]=mkBone("Bone 情報","パターン");
```

パターンの検索は templateList の添字に関し昇順で行われるため，より詳細なパターンを先に登録する.

3.2.2 Bone パターン

Bone パターンは Bop パターンを SET 式にしたもので，PARALELL 式に含まれる基本パターンである.

- 基本演算

```
(SET I8 (HOLE 0 I8) (ADD I8 (HOLE 1 I8) (HOLE 2 I8)))
(SET I32 (HOLE 0 I32) (BAND I32 (HOLE 1 I32) (HOLE 2 I32)))
(SET I16 (HOLE 0 I16) (TSTGES I16 (HOLE 1 I16) (HOLE 2 I16)))
(SET I32 (HOLE 0 I32) (BNOT I32 (HOLE 1 I32)))
(SET I8 (HOLE 0 I8) (CONVIT I8 (HOLE 1 I32)))
```

- マシン依存の特殊なパターン

```
(SET I32 (HOLE 0 I32) (BAND I32 (BNOT I32 (HOLE 1 I32)) (HOLE 2 I32)))
```

3.2.3 Bone 情報

Bone 情報は，bone パターンに付加された情報であり，基本命令から SIMD 命令を組み立てるとき，一般レジスタを SIMD レジスタに置き換えられるときなどにもちいられる．

Bone 情報は，つぎのような S 式の形式で記述される．

```
<BoneInfo>        ::=  (<Paracnts> <Holenum> <Chng> <Replnum> <UnUsed>
                        <Sharednum> <Nosubsthnum> <Subgroups>)
<Paracnts>        ::=  (<int-list>)
<Holenum>         ::=  <int>
<Chng>            ::=  <t or nil>
<Replnum>         ::=  <int>
<UnUsed>          ::=  nil
<Sharednum>       ::=  (<int-list>)
<Nosubsthnum>     ::=  (<int-list>)
<Subgroups>       ::=  (<int-list-list>)
<BoneBody>        ::=  <LIR>
<t or nil>        ::=  t | nil
<int-list>        ::=  <int> | <int> <int-list>
<int-list-list>   ::=  (<int-list>) | (<int-list>) <int-list-list>
```

Bone 情報の各要素は，つぎのような意味をもつ．

1. <Paracnts> 個数：int list

   PARALLEL 式に含まれる L 式の個数 (複数の場合があり得る) を表す．

2. <Holenum> 出力 HOLE 番号：int

   演算結果が出力される HOLE 番号を表す．

3. <Chng> 交換可能性：t — nil

   HOLE 1 と HOLE 2 が交換可能であることを表す．

4. <Replnum> 書き換え規則番号：int

   適用される書き換え規則の番号を表す．

5. <UnUsed> 未使用

   現在使われていない．

6. <Sharednum> 共通 HOLE 番号：int list

   PARALLEL にまとめられたとき，それを構成する各式に共通なレジスタに対応する HOLE の番号を表す．

7. <Nosubsthnum> 置き換え抑制 HOLE 番号：int list

   レジスタに置き換えることを禁止する．

8. <Subgroups> グループ化：int list list

   同一のレジスタに置き換えることを指示する．

3.2.4 補助パターン

連続メモリ領域へのアクセス命令などは，パターンだけでは記述しきれない．

したがって，本システムでは，基本的なパターンを補助パターンとして用意し，それとは別に用意したプログラムによって，条件をみたすことを判定している．

3.3 TMD

TMD (Target Machine Description) にはターゲットマシンに依存する全ての情報を記述する．これには実レジスタのセット及び特性，アドレッシングモード，関数呼びだし規約，アセンブラ表記，命令記述等が含まれる．

TMD の詳しい記述規則は,「Coins Retargetting Guide 森 公一郎 (Koichiro Mori)」を参照している．

3.3.1 IA32 用 TMD

IA32 用の TMD は，マシン情報と一般命令を記述している x86.tmdpp と SIMD 命令を記述している x86simd.tmdpp に分かれている．

3.3.2 TMD の変更手順

本節では TMD の変更手順について述べる．

```
java.lang.Error: compilation aborted.
at coins.backend.gen.CodeGenerator.instructionSelection(CodeGenerator.java)
```

このようなエラーメッセージが出た場合は，パターンが登録していないためで，コード生成部のトレース情報を出して，No Match のパターンを確認する．それが PARALELL で括られたものでない場合，x86.tmdpp に追加する．それが PARALELL で括られたものであった場合，このパターンを LirBoneList_x86.java または x86simd.tmdpp に追加する．

アセンブルコードが思ったようなコードになっていない場合は，コード生成部のトレース情報を出して，Instruction Selection のパターンを調べ，対象のパターンを見つける．それが PARALELL で括られたものでない場合，このパターンを x86.tmdpp で探し，ある場合はその code 属性 (コード) を変更する．それが PARALELL で括られたものであった場合，このパターンを x86simd.tmdpp で探し，ある場合はその code 属性を変更する．ない場合はパターンと code 属性を追加する．

コード生成部のトレース情報を標準出力に表示するには COINS driver option に

```
-coins:trace=TMD[.N]
```

を追加する．

x86simd.tmdpp の変更例

8bit のシフトが LirBoneList_x86.java には登録されているが，x86simd.tmdpp には登録されていない

- プログラム例

```
void psrasub(unsigned char *a,unsigned char b,unsigned char *c) {
        c[0]=a[0] >> b;
        c[1]=a[1] >> b;
        c[2]=a[2] >> b;
        c[3]=a[3] >> b;
        c[4]=a[4] >> b;
```

```
        c[5]=a[5] >> b;
        c[6]=a[6] >> b;
        c[7]=a[7] >> b;
        return;
}
```

- アセンブルコード

```
pextrw $0,%MM0,%edx
movb %dl,-385(%ebp)
movb -385(%ebp),%al
sarb %cl,%al
movb %al,-385(%ebp)
pextrw $0,%MM0,%edx
movb %dh,-386(%ebp)
movb -386(%ebp),%al
sarb %cl,%al
movb %al,-386(%ebp)
pextrw $1,%MM0,%edx
movb %dl,%bh
sarb %cl,%bh
...
pextrw $3,%MM0,%edx
movb %dh,%dl
sarb %cl,%dl
movl %edx,%edi
movb %bl,%dh
movb %dl,%dl
pinsrw $3,%edx,%MM1
movl %edi,%edx
movb %ah,%dh
movb %ch,%dl
pinsrw $2,%edx,%MM1
movl %edi,%edx
movb %bh,%dh
movb %al,%dl
pinsrw $1,%edx,%MM1
movl %edi,%edx
movb -385(%ebp),%dh
movb -386(%ebp),%dl
pinsrw $0,%edx,%MM1
movl %edi,%edx
```

- Instruction Selection　のパターン

```
Matching:
(PARALLEL
 (SET I8 (SUBREG I8 (REG I64 "%MM1") (INTCONST I32 0))
```

```
        (RSHS I8 (SUBREG I8 (REG I64 "%MM0") (INTCONST I32 0)) (REG I8 "b.2%_1%0")))
(SET I8 (SUBREG I8 (REG I64 "%MM1") (INTCONST I32 1))
        (RSHS I8 (SUBREG I8 (REG I64 "%MM0") (INTCONST I32 1)) (REG I8 "b.2%_1%0")))
(SET I8 (SUBREG I8 (REG I64 "%MM1") (INTCONST I32 2))
        (RSHS I8 (SUBREG I8 (REG I64 "%MM0") (INTCONST I32 2)) (REG I8 "b.2%_1%0")))
(SET I8 (SUBREG I8 (REG I64 "%MM1") (INTCONST I32 3))
        (RSHS I8 (SUBREG I8 (REG I64 "%MM0") (INTCONST I32 3)) (REG I8 "b.2%_1%0")))
(SET I8 (SUBREG I8 (REG I64 "%MM1") (INTCONST I32 4))
        (RSHS I8 (SUBREG I8 (REG I64 "%MM0") (INTCONST I32 4)) (REG I8 "b.2%_1%0")))
(SET I8 (SUBREG I8 (REG I64 "%MM1") (INTCONST I32 5))
        (RSHS I8 (SUBREG I8 (REG I64 "%MM0") (INTCONST I32 5)) (REG I8 "b.2%_1%0")))
(SET I8 (SUBREG I8 (REG I64 "%MM1") (INTCONST I32 6))
        (RSHS I8 (SUBREG I8 (REG I64 "%MM0") (INTCONST I32 6)) (REG I8 "b.2%_1%0")))
(SET I8 (SUBREG I8 (REG I64 "%MM1") (INTCONST I32 7))
        (RSHS I8 (SUBREG I8 (REG I64 "%MM0") (INTCONST I32 7)) (REG I8 "b.2%_1%0"))))
```

- LirBoneList_x86.java には登録あり

```
mkBone("((16 8) 1 nil nil nil nil (2))",
       "(SET I8 (HOLE 0 I8) (RSHS I8 (HOLE 1 I8) (HOLE 2 I8)))");
```

- x86simd.tmdpp に登録する

1. Instruction Selection の PARALLEL で囲まれえたパターンの仮想レジスタを x86.tmdpp の defregset で定義したレジスタ symbol に置き換える

```
(REG I64 "%MM1")    ->   regm
(REG I64 "%MM0")    ->   regm
(REG I8 "b.2%_1%0") ->   regb

(PARALLEL
 (SET I8 (SUBREG I8 regm (INTCONST I32 0))
         (RSHS I8 (SUBREG I8 regm (INTCONST I32 0)) regb))
 (SET I8 (SUBREG I8 regm (INTCONST I32 1))
         (RSHS I8 (SUBREG I8 regm (INTCONST I32 1)) regb))
 (SET I8 (SUBREG I8 regm (INTCONST I32 2))
         (RSHS I8 (SUBREG I8 regm (INTCONST I32 2)) regb))
          ...
 (SET I8 (SUBREG I8 regm (INTCONST I32 7))
         (RSHS I8 (SUBREG I8 regm (INTCONST I32 7)) regb)))
```

2. 同じパターンをまとめ、異なる個所を foreach マクロで表わす
(INTCONST I32 @i) を foreach マクロで表わす

```
(PARALLEL
  (foreach @i (0 1 2 3 4 5 6 7)
    (SET I8 (SUBREG I8 regm (INTCONST I32 @i))
            (RSHS I8 (SUBREG I8 regm (INTCONST I32 @i)) regb))))
```

マシン記述を簡潔にするために、以下のようなマクロを使用出来る.

マクロ表記 展開形

```
(foreach @x (a b)    (foo a) (foo b)
  (foo @x))

(foreach (@x @y) ((a 1) (b 2))    (foo a 1) (foo b 2)
  (foo @x @y))

(foreach @a (abracadabra)    (foo abracadabra) (bar abracadabra))
  (foo @a) (bar @a))
```

3. defrule に code 属性と必要な属性を加える

```
(defrule void
  (PARALLEL
    (foreach @i (0 1 2 3 4 5 6 7)
      (SET I8 (SUBREG I8 regm (INTCONST I32 @i))
              (RSHS I8 (SUBREG I8 regm (INTCONST I32 @i)) regb))))
  (regset ($3 *reg-cx-I8*))
  (code
    (movq2dq $2 "%XMM6")
    (pxor "%XMM7" "%XMM7")
    (punpcklbw "%XMM7""%XMM6")
    (movd $3 "XMM7")
    (psraw "%XMM7" "%XMM6")
    (packuswb "%XMM6" "%XMM6")
    (movdq2q "%XMM6" $1))
  (clobber (REG I128 "%XMM6") (REG I128 "%XMM7"))
  (cost 3))
```

第4章 SIMD 並列化個々の処理

4.1 論理演算を用いた IF 変換

本節では, 論理演算を用いた IF 変換の仕掛けと実装について述べる.

4.1.1 目的と概要

論理演算を用いた IF 変換とは, 代入文が選択的に実行されるだけの単純な if 文を, 論理演算を用いて同じ効果が得られるように変換することである. 例えば以下のプログラムを考える.

```
if( c ){
  a = x + 1    <--- (1)
}
else{
  a = y + 1    <--- (2)
}
print(a)
```

このプログラムを CFG として表現すると if 部分で条件分岐が発生し, then パート (1) と else パート (2) の基本ブロックに分割される. if 変換では上記プログラムを以下のように変形する. ただしこの変形を正しく行うためには, 条件文 c が真の場合に全 bit 1 になり, 偽の場合には全ビット 0 であることが必要である.

```
a = ((c) & (x + 1)) | (!(c) & (y + 1))
print(a)
```

この変形は, then パート (1) と条件文 c の論理積結果と, else パート (2) と条件文 c の論理積結果を論理和したものである. 条件文 c が真の場合 (上述の条件に基づくと c は全 bit 1) には

```
a = ((全 bit 1) & (x + 1)) | (~(全 bit 1) & (y + 1))
```

となり, then パート (1) の値が a に代入される.

4.1.2 基本ブロックの結合

SIMD 並列化はひとつの基本ブロック単位で行われるため, 基本ブロックはなるべくまとめた方がよい. 論理演算を用いた IF 変換を行うと, プログラムの制御構造が変わってしまう. これにより図 4.1 (a) のような制御構造が, 図 4.1 (b) のようになってしまう可能性がある. しかし基本ブロックの結合を行うことにより, 図 4.1 (b) は図 4.1 (c) に変換される.

COINS の実装では, 図 4.2 の左側のようなプログラムから制御フローグラフを生成すると, 図 4.2 の真ん中のような制御フローグラフができる.

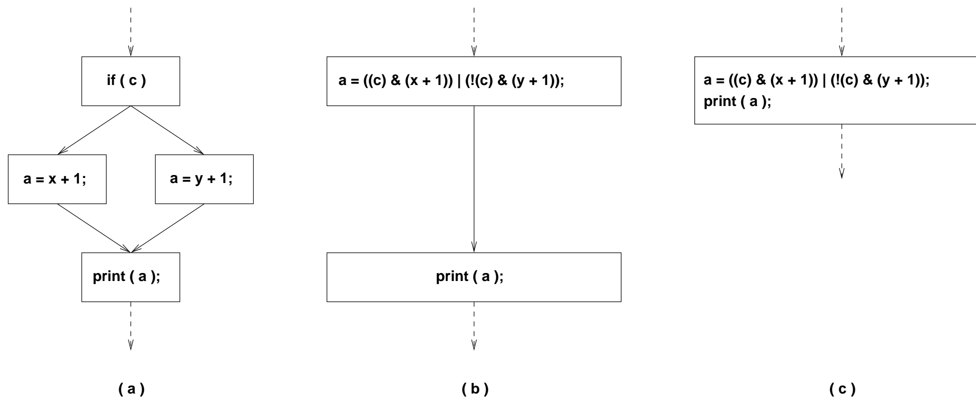

図 4.1: 論理演算を用いた IF 変換と基本ブロックの結合

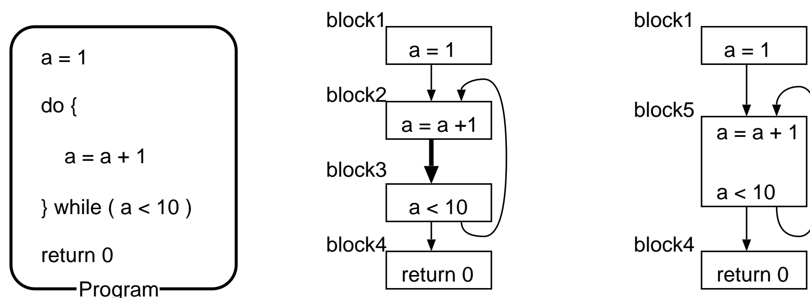

図 4.2: 除去する辺

図 4.2 の真ん中の制御フローグラフにおいて, `block2` と `block3` を結ぶ辺は除去することの可能な辺で, 無駄なジャンプ命令が存在することになる. `block2` のように後続 node をひとつしか持たず, その後続 node である `block3` の先行 node がひとつ (`block2`) だけであるとき, このような辺は除去できる.

そこで本システムでは 図 4.2 の右側の制御フローグラフのように, `block2` と `block3` を合成した新しい node (`block5`) を作り, 無駄なジャンプ命令の除去を行う.

なお現実装では `block5` に相当する基本ブロックは全く新しい基本ブロックを作成するのではなく, `block2` または `block3` で代用する形になっている.

4.1.3 論理演算を用いた IF 変換の制約

論理演算を用いた IF 変換では全ての if 文を扱うわけではなく, ある特定の条件を満たした場合のみ変換を行う. ここでいう条件とは, 以下のものに対する制約である.

1. if 文の制御構造
2. if 文の then パートおよび else パートに含まれる式

以下, 各々の制約の詳細を述べる.

制御構造の制約

一般的に, if 文で作られる制御構造は then パートおよび else パート内部に複数の基本ブロックを持つ. 本システムでは, 簡単のため, 以下に挙げる制約を満たす if 文のみを論理演算を用いた IF 変換の対象とする.

1. then パートおよび else パートはひとつの基本ブロックである

2. then パートおよび else パートは, 分岐元の基本ブロック以外の predecessor を持たない

3. then パートおよび else パートは, 単一の successor を持つ

この制約を満たす制御構造は, 図 4.3 (a) のような単純な構造となる. また 図 4.3 (b) のような制御構造を持つ場合でも, 論理演算を用いた IF 変換を内側の if 文から行うことによって, (b) の 3, 4, 5, 6 からなる制御構造が (a) の 3 に変換され適用可能な形となる.

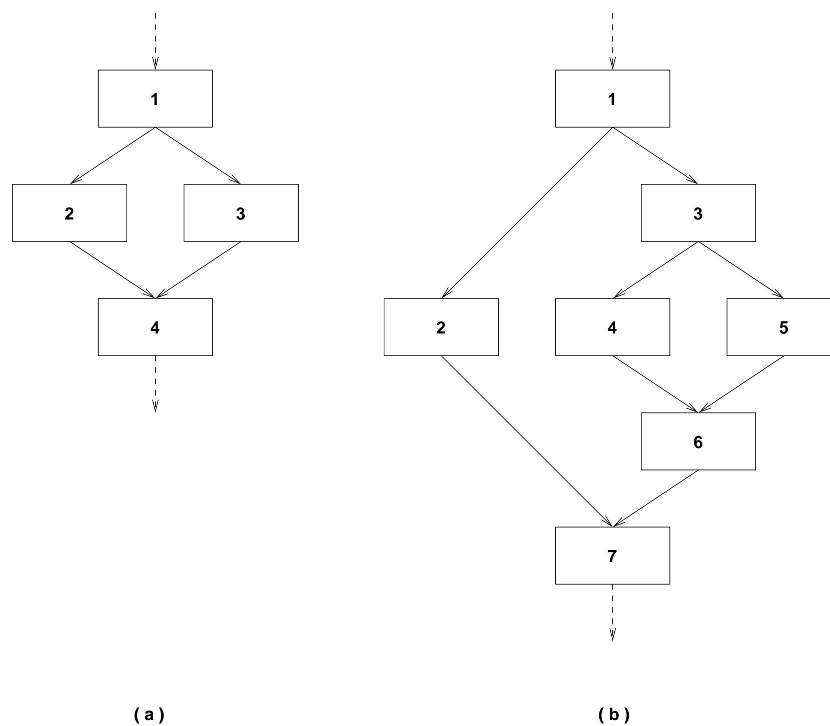

図 4.3: 制御構造の制約

式の制約

論理演算を用いた IF 変換では, then パートおよび else パートに含まれる式に対する制約を以下のように定める. なお, 変換の対象は LIR であるため, 制約は L 式に則した形で記述する.

1. 対象となる L 式は, REG 式に対する SET 式でなければならない

2. 対象となる式の個数は, 別途定められた数をこえてはならない

3. 対象となる L 式の型は整数型でなければならない

4. 対象となる L 式に除算が含まれてはならない

1 により, 論理演算を用いた IF 変換の対象を代入文だけと限定している. これは藤波らが [2] で提案した方針に従ったものである.

論理演算を用いた IF 変換を行うことによって得られる効果は, 同形の論理演算をまとめて SIMD 並列化の対象とすることと, 条件分岐をへらすことが挙げられる. 後者の理由は, 一般的に条件分岐はコストが高いためである. しかし, 多くの式を then パートや else パートに含む if 文を変換することにより, 条件分岐のコストよりも論理演算のコストの方が大きくなってしまうことが考えられる. そこで本システムでは, 論理演算を用いた IF 変換で対象とする式の数を COINS コンパイラオプションによって指定することができる[1]. これが 2 で書かれた制約である.

3 および 4 で書かれた制約は, 精度に関する問題が発生するのを防ぐためのものである.

4.1.4 論理演算の組み立て

本節では, 論理演算を用いた IF 変換でどのように論理演算を作成するかについて述べる. まず例として以下のプログラムを考える.

```
if( c ){
  a = x + 1
}
else{
  b = y + 1
}
print(a,b)
```

この if 文からなる制御構造は上記の制約を全て満たしており, 変換可能である. この構造を LIR で表すと以下のようになる.

```
L0:
  (JUMPC (TSTNE I32 (REG I32 "c") (INTCONST I32 "0"))
         (LABEL I32 "L1")
         (LABEL I32 "L2"))

L1:
  (SET I32 (REG I32 "a")
           (ADD I32 (REG I32 "x") (INTCONST I32 "1")))
  (JUMP (LABEL I32 "L3"))

L2:
  (SET I32 (REG I32 "b")
           (ADD I32 (REG I32 "y") (INTCONST I32 "1")))
  (JUMP (LABEL I32 "L3"))

L3:
  (CALL (STATIC I32 "print") ((REG I32 "a") (REG I32 "b")) ())
```

論理演算の組み立てには図 4.4 に示すアルゴリズムを用いる.

図 4.4 のアルゴリズムを用いて変換することにより, 下記のコードを得ることができる.

[1]デフォルトは 2 となっている

```
cond=条件式
candidateThen={}
candidateElse={}
for each then パートの代入文 node do
  candidateThen に node を加える
  if (else パートに node と同じ左辺を持つ代入文 n がある) then
    candidateElse に n を加える
    n に選択されたしるしをつける
  else
    candidateElse に "val←val" を加える
    (val は node の左辺変数)
  endif
end for
for each else パートの代入文 node2 do
  if (node2 に選択されたしるしが無い) then
    candidateElse に node2 を加える
    candidateThen に "val2←val2" を加える
    (val2 は node2 の左辺変数)
  endif
end for
while (candidateThen または candidateElse が空でない) do
  x←candidateThen からひとつ取り出す
  y←candidateElse からひとつ取り出す
  band1←cond と x の論理積
  band2←!cond と y の論理積
  bor←band1 と band2 の論理和
  "x (または y) の左辺変数 ←bor" という L 式を作る
end while
```

図 4.4: 論理演算組み立てアルゴリズム

```
a = ((c) & (x + 1)) | (!(c) & (a))
b = ((c) & (b) | (!(c) & (y + 1))
print(a,b)
```

4.1.5 アルゴリズム

論理演算を用いた IF 変換のアルゴリズムを図 4.5 に示す.

```
boolean checked←true;
while(checked)
  基本ブロックの結合を行う
  changed←false;
  for each 基本ブロック blk (最内側 if から) do
    if (blk の末尾が if 文でない) then
      continue;
    endif
    if (if 文の構造が対象のものでない) then
      continue;
    end if
    if (if 文の条件分岐後に対象外の演算がある) then
      continue;
    end if
    論理演算の組み立て
    CFG に反映
    changed←true;
  end for
end while
```

図 4.5: 論理演算を用いた IF 変換アルゴリズム

4.2 DAG 生成

本節では DAG (Directed Acyclic Graph) 生成について述べる.

4.2.1 概要

基本ブロックを DAG に変換することは, 局所的な共通部分式を除去するためのよく知られた方法である. SIMD 並列化における DAG 生成の目的は, 特殊な演算子を伴う SIMD 命令を利用するためのパターンマッチングを容易に行えるように, 細かい式を大きな式にまとめることである.

ひとつの基本ブロック内の LIR を DAG (Directed Acyclic Graph) に変形する際に, 本システムではプログラム全体を SSA (Static Single Assignment) 形式に変換する. これにより変数の定義が一意に定まるので, 変換を容易に行うことができる.

例えば以下のプログラム片を考える.

```
...
v = a + 1;                (1)
x = v * a;                (2)
y = v - a;                (3)
z = v / a;                (4)
...
```

(1) 式で定義された変数 “v” は (2), (3), (4) 式で利用されている. DAG を作成するためには, これらの式で利用されている “v” を, “v” を定義する式 “a + 1” で置き換える. これにより上記のプログラム片は以下のように変形される.

```
...
v = a + 1;                (1’)
x = (a + 1) * a;          (2’)
y = (a + 1) - a;          (3’)
z = (a + 1) / a;          (4’)
...
```

ここで v の生存区間が対象とする基本ブロック内部で閉じていれば, (1’) は無用な命令として除去することができる. 本システムではこれを行う.

4.2.2 アルゴリズム

DAG 生成のアルゴリズムを図 4.6 に示す.

```
map={}
for each 基本ブロック blk do
  for each L 式 node do
    if (node が代入文である) then
      if (map に登録されている変数 v が node 右辺にある) then
        tmp←map.get(v)
        node 右辺の v を tmp で置換する
      endif
      if (node 右辺にメモリアクセスがない) then
        if (node 左辺がレジスタ変数である) then
          node 左辺が key として map に登録してあれば除去する
          map.put(node 左辺, node 右辺)
        endif
      else if (node 左辺がレジスタ変数である) then
        node 左辺が key として map に登録してあれば除去する
      endif
    endif
  end for
  blk 内の無用な命令を除去する
end for
```

map.get(v) : v を key として map から値を取り出す
map.put(key,val) : key を key, val を値として map に登録する

図 4.6: DAG 生成アルゴリズム

4.3 範囲解析

本範囲解析では，各命令 (L 式) 毎に上向きの範囲解析と下向きの範囲解析をおこない，その結果をもちいて各命令に含まれるサイズの書き換えをおこなう．

ただし，SET 式以外の L 式にたいしては，範囲解析をおこなわない．

4.3.1 上向きの範囲解析

上向きの範囲解析の処理は，L 式にたいしておこなわれ，解析結果は，Hashtable に IntBound の形式で，保存される．この解析結果は，下向きの範囲解析でつかわれる．

処理は，L 式のノードの構成にしたがってボトムアップに行なわれる．

各ノードでは，子ノードから上がってきた上向き解析の結果えられる IntBound にたいし，演算毎に定義された範囲解析を適用し，親ノードにその結果を返す．

4.3.2 下向きの範囲解析

下向きの範囲解析の処理は，L 式にたいしておこなわれ，解析結果は，Hashtable に IntBound の形式で，保存される．この解析結果は，サイズの書き換え処理でつかわれる．

処理は，L 式のノードの構成にしたがってトップダウンに行なわれる．

各ノードでは，親ノードから降りてきた下向き解析の結果えられる IntLive と上向き解析の結果 IntBound を live ビットに変換したものとの intersection をそのノードの live ビットとする．さらに，子ノードには，上記のようにしてもとまった live ビットに演算毎に定義された範囲解析を適用してわたす．

4.3.3 サイズの書き換え

サイズの書き換えの処理は，各命令 (L 式) 毎に，下向き範囲解析の結果をもとにおこなわれ，サイズを書き換えられた L 式を出力する．

ただし，この書き換え処理は，SET 式にたいしてのみおこなわれる．

処理は，L 式のノードの構成にしたがった,「拡張/縮退」演算を挿入と,「拡張/縮退」演算の除去とでおこなわれる．

4.4 基本命令切り出し

本 SIMD 並列化では，基本命令を組み合わせて SIMD 命令に対応する命令を作成するが,「基本命令の切り出し」処理では，DAG 化された L 式の列を入力とし，各々の DAG に含まれる基本命令列を切り出し，その列を出力とする．

基本命令は，つぎのように分類できる:

1. メモリからのロード命令
2. メモリへのストア命令
3. 一般レジスタへの演算適用と代入
4. 定数のレジスタへの代入
5. マシン依存の特殊なパターンをもつ命令

   マシン依存の特殊なパターンの例としては，IA32/SSE2 の PSADBW などがある．

   PSADBW の記述例

4.4.1 「切り出し」の方法

基本命令に対応するパターン (bop パターン) を用意する．

DAG と bop パターンとのパターンマッチングをトップダウンにおこなう．

あらたなレジスタを生成し，レジスタを destination とし，マッチした L 式を source とする SET 式を作成する．

4.5 SIMD 命令の組み立て

SIMD 命令の組み立ての処理は，DAG から切り出された基本命令列を入力とし，同じパターンをもつ基本命令列を集めて，SIMD 命令に対応する PARALLEL 式に変換して出力する．

ここで，同じパターンをもつということが，連続メモリ領域へのアクセス命令の場合とそれ以外の命令では，処理上異なるので，別々に説明する．

4.5.1 メモリアクセス命令の組み立て

最初にメモリからのロードやメモリへのストアなどの連続メモリ領域へのアクセス命令の組み立てについて述べる．

基本命令の切り出し処理の出力として，つぎのような連続メモリ領域へのアクセス命令の列がえられる．この例では，メモリから 16 ビットずつ読み出して，レジスタにセットする基本命令が並んでいる．

```
(SET I16 (REG I16 "b1") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 0))))
(SET I16 (REG I16 "b2") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 2))))
(SET I16 (REG I16 "b3") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 4))))
(SET I16 (REG I16 "b4") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 6))))
(SET I16 (REG I16 "b5") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 8))))
(SET I16 (REG I16 "b6") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 10))))
(SET I16 (REG I16 "b7") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 12))))
(SET I16 (REG I16 "b8") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 14))))
```

このような基本命令列をあつめて，つぎのような PARALLEL 式を構成するのが，連続メモリ領域へのアクセス命令の組み立ての処理である．

```
(PARALLEL
  (SET I16 (REG I16 "b1") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 0))))
  (SET I16 (REG I16 "b2") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 2))))
  (SET I16 (REG I16 "b3") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 4))))
  (SET I16 (REG I16 "b4") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 6))))
  (SET I16 (REG I16 "b5") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 8))))
  (SET I16 (REG I16 "b6") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 10))))
  (SET I16 (REG I16 "b7") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 12))))
  (SET I16 (REG I16 "b8") (MEM I16 (ADD I32 (REG I32 "b") (INTCONST I32 14)))))
```

4.5.2 一般命令の組み立て

つぎに，一般命令の組み立てについて述べる．
基本命令の切り出し処理の出力として，つぎのような基本命令の列がえられる．

```
(SET I16 (REG I16 "sv$1b19%") (ADD I16 (REG I16 "sv$1b17%") (REG I16 "sv$1b18%")))
    ....
(SET I16 (REG I16 "sv$1b22%") (ADD I16 (REG I16 "sv$1b20%") (REG I16 "sv$1b21%")))
    ....
(SET I16 (REG I16 "sv$1b25%") (ADD I16 (REG I16 "sv$1b23%") (REG I16 "sv$1b24%")))
    ....
(SET I16 (REG I16 "sv$1b28%") (ADD I16 (REG I16 "sv$1b26%") (REG I16 "sv$1b27%")))
    ....
(SET I16 (REG I16 "sv$1b31%") (ADD I16 (REG I16 "sv$1b29%") (REG I16 "sv$1b30%")))
    ....
(SET I16 (REG I16 "sv$1b34%") (ADD I16 (REG I16 "sv$1b32%") (REG I16 "sv$1b33%")))
    ....
(SET I16 (REG I16 "sv$1b37%") (ADD I16 (REG I16 "sv$1b35%") (REG I16 "sv$1b36%")))
    ....
(SET I16 (REG I16 "sv$1b40%") (ADD I16 (REG I16 "sv$1b38%") (REG I16 "sv$1b39%")))
```

ここでは，ADD をオペレータとしてもつもののみに着目するために，その他のものを .... として省略して表示している．この基本命令列から，PADD 命令に対応するつぎの PARALLEL 式を構成することが，一般命令の組み立て処理である．

```
(PARALLEL
  (SET I16 (REG I16 "sv$1b40%") (ADD I16 (REG I16 "sv$1b38%") (REG I16 "sv$1b39%")))
  (SET I16 (REG I16 "sv$1b37%") (ADD I16 (REG I16 "sv$1b35%") (REG I16 "sv$1b36%")))
  (SET I16 (REG I16 "sv$1b34%") (ADD I16 (REG I16 "sv$1b32%") (REG I16 "sv$1b33%")))
  (SET I16 (REG I16 "sv$1b31%") (ADD I16 (REG I16 "sv$1b29%") (REG I16 "sv$1b30%")))
  (SET I16 (REG I16 "sv$1b28%") (ADD I16 (REG I16 "sv$1b26%") (REG I16 "sv$1b27%")))
  (SET I16 (REG I16 "sv$1b25%") (ADD I16 (REG I16 "sv$1b23%") (REG I16 "sv$1b24%")))
  (SET I16 (REG I16 "sv$1b22%") (ADD I16 (REG I16 "sv$1b20%") (REG I16 "sv$1b21%")))
  (SET I16 (REG I16 "sv$1b19%") (ADD I16 (REG I16 "sv$1b17%") (REG I16 "sv$1b18%"))))
```

入力としてあたえられた基本命令列の各々の基本命令 ins にたいし，以下の処理をおこなう．

1. ins とマッチする bone をもとめる
   Bone の検索順序は，bone の templateList の添字に関し昇順である．
   もとまった bone を b と書く．そのような bone がもとまらなければ，ins は，SIMD 命令の構成要素として扱われない．
2. 基本命令列の中から b とパターンがマッチし，マッチ条件をみたすものを集める
   このとき，マッチするものの個数は，b の bone 情報の Paracnts に書かれた個数の最大値である．
   マッチングは，基本命令列で ins のつぎに置かれている L 式以降のものにたいしておこなわれる．
   集められた L 式は，互いにレジスタ間の参照関係をもたないものとする．
   マッチ条件としては，つぎのようなものがある:
   同じ HOLE 番号にマッチしたレジスタ同士が「同値」であること．ここで,「同値」とは,
   同じ HOLE 番号にマッチしたものが同じ演算種別をもつこと．
   - ins と b をマッチさせたとき env
3. ins および集められた L 式を要素とする PARALLEL 式を構成し，ins と置き換える
   基本命令列からは，集められた L 式は，削除される．

4.6　命令列の整列化

SIMD 命令の組み立てでは，命令列の中で離れた位置にあった命令群を削除して 1 個の PARALLEL 式に組み立てて命令列の中に置くため，命令列の順序が崩れている可能性がある．

そこで，命令列の順序の整合性を保証するために，レジスタの定義使用関係をもちいて，命令列の整列化をおこなう．

4.7　制約の適用

L 式からなる命令列を入力とし，各命令 ins にたいし，Bone 情報に記述された内容をもとに，つぎのような制約を適用してえられた命令列で置き換えたものを出力とする．

0 以外の HOLE 番号が，Bone 情報の Holenum に記述されている場合，Holenum で指定された HOLE 番号にマッチしたレジスタを SET 式の destination とする．また，このレジスタの置換えを命令列全体に伝播させる．

このとき，ソース側のレジスタへの代入が発生しないように，デスティネーション側のレジスタにまずソース側のレジスタをコピーし，L 式中のソース側のレジスタをデスティネーション側のレジスタで置き換える．

Bone 情報の Replnum n が記述されている場合，ins を boneList の rewriteList[n] にある L 式の列で置き換える．

4.8 SIMD レジスタ割り当て

SIMD レジスタ割り当て処理は，L 式の命令列を入力とし，それらに含まれる一般レジスタを SIMD レジスタに置き換えた L 式の命令列を出力する．

ただし，SIMD レジスタ割り当て処理の入力は，L 式の列であるが，それらの各々は，SIMD 命令に対応する形式にまとめられているものとする．

たとえば，IA32 の SIMD 命令 padd に対応する L 式は，つぎのような PARALLEL 式に変換されたものが，SIMD レジスタ割り当て処理の入力となる．

```
(PARALLEL
  (SET I16 (REG I16 "sv$1b19%") (ADD I16 (REG I16 "sv$1b17%") (REG I16 "sv$1b18%")))
  (SET I16 (REG I16 "sv$1b22%") (ADD I16 (REG I16 "sv$1b20%") (REG I16 "sv$1b21%")))
  (SET I16 (REG I16 "sv$1b25%") (ADD I16 (REG I16 "sv$1b23%") (REG I16 "sv$1b24%")))
  (SET I16 (REG I16 "sv$1b28%") (ADD I16 (REG I16 "sv$1b26%") (REG I16 "sv$1b27%")))
  (SET I16 (REG I16 "sv$1b31%") (ADD I16 (REG I16 "sv$1b29%") (REG I16 "sv$1b30%")))
  (SET I16 (REG I16 "sv$1b34%") (ADD I16 (REG I16 "sv$1b32%") (REG I16 "sv$1b33%")))
  (SET I16 (REG I16 "sv$1b37%") (ADD I16 (REG I16 "sv$1b35%") (REG I16 "sv$1b36%")))
  (SET I16 (REG I16 "sv$1b40%") (ADD I16 (REG I16 "sv$1b38%") (REG I16 "sv$1b39%"))))
```

この L 式に SIMD レジスタ割り当て処理を適用すると，つぎのような L 式が構成される:

```
(PARALLEL
  (SET I16
    (SUBREG I16 (REG I128 "%XMM1") (INTCONST I32 0))
    (ADD I16
      (SUBREG I16 (REG I128 "%XMM0") (INTCONST I32 0))
      (SUBREG I16 (REG I128 "%XMM2") (INTCONST I32 0))))
  (SET I16
    (SUBREG I16 (REG I128 "%XMM1") (INTCONST I32 1))
    (ADD I16
      (SUBREG I16 (REG I128 "%XMM0") (INTCONST I32 1))
      (SUBREG I16 (REG I128 "%XMM2") (INTCONST I32 1))))
  (SET I16
    (SUBREG I16 (REG I128 "%XMM1") (INTCONST I32 2))
    (ADD I16
      (SUBREG I16 (REG I128 "%XMM0") (INTCONST I32 2))
      (SUBREG I16 (REG I128 "%XMM2") (INTCONST I32 2))))
  (SET I16
    (SUBREG I16 (REG I128 "%XMM1") (INTCONST I32 3))
    (ADD I16
      (SUBREG I16 (REG I128 "%XMM0") (INTCONST I32 3))
      (SUBREG I16 (REG I128 "%XMM2") (INTCONST I32 3))))
  (SET I16
    (SUBREG I16 (REG I128 "%XMM1") (INTCONST I32 4))
    (ADD I16
      (SUBREG I16 (REG I128 "%XMM0") (INTCONST I32 4))
      (SUBREG I16 (REG I128 "%XMM2") (INTCONST I32 4))))
  (SET I16
    (SUBREG I16 (REG I128 "%XMM1") (INTCONST I32 5))
    (ADD I16
      (SUBREG I16 (REG I128 "%XMM0") (INTCONST I32 5))
      (SUBREG I16 (REG I128 "%XMM2") (INTCONST I32 5))))
  (SET I16
    (SUBREG I16 (REG I128 "%XMM1") (INTCONST I32 6))
    (ADD I16
      (SUBREG I16 (REG I128 "%XMM0") (INTCONST I32 6))
      (SUBREG I16 (REG I128 "%XMM2") (INTCONST I32 6))))
  (SET I16
    (SUBREG I16 (REG I128 "%XMM1") (INTCONST I32 7))
    (ADD I16
      (SUBREG I16 (REG I128 "%XMM0") (INTCONST I32 7))
      (SUBREG I16 (REG I128 "%XMM2") (INTCONST I32 7)))))
```

この例では，つぎのような SIMD レジスタ割当がおこなわれている:

一般レジスタ		SIMD レジスタ
(REG I16 "sv$1b17%")		
(REG I16 "sv$1b20%")		
(REG I16 "sv$1b23%")		
(REG I16 "sv$1b26%")	⇒	(REG I128 "%XMM0")
(REG I16 "sv$1b29%")		
(REG I16 "sv$1b32%")		
(REG I16 "sv$1b35%")		
(REG I16 "sv$1b38%")		
(REG I16 "sv$1b18%")		
(REG I16 "sv$1b21%")		
(REG I16 "sv$1b24%")		
(REG I16 "sv$1b27%")	⇒	(REG I128 "%XMM2")
(REG I16 "sv$1b30%")		
(REG I16 "sv$1b33%")		
(REG I16 "sv$1b36%")		
(REG I16 "sv$1b39%")		
(REG I16 "sv$1b19%")		
(REG I16 "sv$1b22%")		
(REG I16 "sv$1b25%")		
(REG I16 "sv$1b28%")	⇒	(REG I128 "%XMM1")
(REG I16 "sv$1b31%")		
(REG I16 "sv$1b34%")		
(REG I16 "sv$1b37%")		
(REG I16 "sv$1b40%")		

表 4.1: SIMD レジスタの割り当ての例

4.8.1 SIMD レジスタ割り当て処理

入力としてあたえられた L 式の命令列の各々にたいし，以下の処理を適用する:
処理の対象である L 式の命令を ins とする．
ins が PARALLEL 式でなければ，SIMD 命令に対応するものでないとみなし，SIMD レジスタ割り当て処理を適用せず，出力命令列にそのまま加える．

1. Bone とのマッチング
   最初に，ins にマッチする bone をもとめる．
2. Bone 情報の Subgroups が空でない場合
3. Bone 情報の Subgroups が空の場合

4.9 Cleanup

CleanupLir の処理は，SIMD 並列化処理の間に発生したつぎのような形の無用なコピー文を除去することを目的としている．
すなわち，

```
y=x
...（間で x の使用がない）
x=y
```

の形で，y は，x=y 以降出現しないという条件を満たす L 式の列があった場合，この組の L 式を削除することである．
この処理は，便宜的なものであり，本来は一般的な最適化の手法を適用して，無用命令の除去をおこなうべきであろう．

参考文献

[1] COINS project. http://www.coins-project.org/.

[2] 藤波順久, 阿部正佳. SIMD 型拡張命令をもっと使った最適化への道のり. 第 43 回プログラミング・シンポジウム報告集, pp. 185–196, 2002.

[3] 文部科学省. 科学技術振興調整費 総合研究「並列化コンパイラ向け共通インフラストラクチャの研究」. http://www.mext.go.jp/a_menu/kagaku/chousei/h12unyob3l.html.