

SIMD最適化向けソースコードレベル でのコード変形

東京工業大学
大学院 情報理工学研究科
数理・計算科学専攻

蒲野 茂幸
(07M37080)

平成20年度 修士論文

指導教官 佐々 政孝 教授

平成21年1月30日

概要

SIMD (Single Instruction Multiple Data stream) 命令では、1 命令で同一演算、同一型の複数の命令を並列に実行でき、SIMD 命令をうまく使うことによってプログラムの実行時間を大きく改善することができる。しかし、コンパイラによって SIMD 命令を生成する技術は確立されておらず、効果は限定的である。また、プログラムによっては通常のスカラ最適化よりも実行時間が増加してしまうこともある。その原因として、SIMD 化を期待してループ展開や if 変換などのコード変形をしたが SIMD 化ができなかった場合に、その変形されたプログラムをスカラ実行することによって効率が落ちてしまう場合などがあげられる。そこで、本研究ではループ単位での SIMD 並列化を行うか否かを判定することを目標にした。また、使用するコンパイラとして、ソースコードレベルとアセンブリ言語レベルで解析を行うものを仮定する。スカラでの最適化だけ行うのであれば、アセンブリ言語レベルでの解析だけで十分であるが、並列化を行うことを考えるとソースコードレベルでの解析は必要である。このソースコードレベルで SIMD 化を行うにあたって必要とされる手法として、ループ展開やポインタ配列の配列変換などがある。ここで、ループ単位での SIMD 化を行うか否かの判定をアセンブリ言語レベルで行ったとすると、変形されたコードをソースコードレベルでの変形の前まで戻すことは難しい。そこで、ソースコードレベルで SIMD 化の判定を行い SIMD 化できないと判断したプログラムに対しては、変形前の状態へと戻し、通常のスカラ最適化を行うことにする。

現在自動で SIMD 化が行えるコンパイラとして GCC や Intel のコンパイラ (icc) があり、Cell プロセッサなどでは SIMD 化が行えるようなコンパイラの開発がおこなわれている。しかし、これらのコンパイラの SIMD 化技術が書かれている論文では SIMD 化を行う変形手法は書かれているが、SIMD 化判定についての記述がない。実際、本論文の実験でも icc で SIMD 化オプションを付けてコンパイルを行った結果、実行時間が増加してしまった例が見られた。

本研究で設計したソースコードレベルでのコード変換器を COINS コンパイラインフラストラクチャ上で実装を行い、マルチメディア向けのベンチマークを使用して評価を行った。提案手法でコード変換を行い SIMD 化可能と判断されたループに対して、期待される命令レベルの変換が行われた場合、50%以上の実行時間の減少が見られた。また、SIMD 化ができなかった場合の実行時間が増加してしまうということがなくなった。これらの実験結果により、ソースコードレベルでの SIMD 化判定の有効性を示した。

目次

第 1 章	はじめに	6
1.1	背景	6
1.2	本研究の概要	7
1.3	構成	7
第 2 章	準備	9
2.1	基本ブロック	9
2.2	制御フローグラフ	10
2.3	支配関係	11
2.4	支配木	12
2.5	支配境界	13
2.6	制御依存	15
2.7	ループ	16
2.7.1	ループ制御変数	16
2.8	抽象構文木	16
第 3 章	静的単一代入 (SSA) 形式	19
3.1	SSA 形式	19
3.2	SSA 変換	19
3.3	SSA 形式に基づく最適化	21
3.3.1	コピー伝播	21
3.3.2	SSA 形式を用いたコピー伝播	21
3.4	SSA 逆変換	22
3.5	SSA 形式の性質	23
第 4 章	SIMD 化向けコード変形	24
4.1	ソースコードレベルでの変形	24
4.2	アセンブリ言語レベルでの変形	24
第 5 章	ソースコードレベルでのコード変形器の設計	26
5.1	ポインタ配列の配列変換	26
5.2	if 変換	33
5.2.1	パターンマッチング	35
5.2.2	select 文向きの変換	36
5.3	スカラ拡張	44

5.4	ループ展開	46
5.5	Super Level Parallelism	46
5.6	SIMD 化判定	51
第 6 章	並列化コンパイラ向け共通インフラストラクチャCOINS	54
6.1	概要	54
6.2	構成	54
6.3	COINS の特徴	55
6.4	高水準中間言語 HIR について	56
6.4.1	概要	56
6.4.2	具体表現	56
第 7 章	実験と考察	58
7.1	実装	58
7.2	実験	58
7.3	考察	60
7.4	今後の課題	62
7.4.1	SIMD 化判定方法の検討	62
7.4.2	SIMD 化できる文を増やす	62
7.4.3	命令レベルでのコード変形器の設計	62
第 8 章	関連研究	64
8.1	Bik らの研究	64
8.2	Sreraman らの研究	64
8.3	鈴木らの研究	64
第 9 章	まとめ	65

目次

1.1	SIMD の例	7
2.1	基本ブロック	9
2.2	制御フローグラフ (CFG)	10
2.3	支配木	13
2.4	X の支配境界は Y であり Z の支配境界も Y の例	14
2.5	Z の支配境界は Y であり X の支配境界は Y ではない例	14
2.6	制御依存	15
2.7	式 2.1 に対する解析木	17
2.8	式 2.1 に対する抽象構文木	18
5.1	ソースコードレベルでのコード変形器	26
5.2	配列に変換するポインタプログラム例	28
5.3	ポインタ変数 p_a_0 を初期化する	32
5.4	絶対値を求めるプログラム	33
5.5	条件分岐の例	34
5.6	if 変換のフローチャート	35
5.7	Select 文の例	36
5.8	if 変換を行うプログラム例	37
5.9	図 5.8 の制御フローグラフ	37
5.10	phi-list を作成	39
5.11	GP を求めるアルゴリズム	40
5.12	図 5.11 に対する基本ブロックの関係	41
5.13	phi-list から phi 文を生成するアルゴリズム	42
5.14	制御フローグラフと対応する GP	43
5.15	if 変換されたプログラム	44
5.16	SIMD レジスタとデータサイズ	46
5.17	変形前との実行時間の比較	53
6.1	COINS 構成図	55
6.2	プログラム 6.1 の HIR	57
7.1	実行時間の比較	61

表 目 次

2.1	簡単な文法の例	16
5.1	CR の正規化ルール	30
5.2	CR^{-1} 変換のルール	31
7.1	実験方法	59
7.2	Let's note CF-W5 の仕様	60

第1章 はじめに

1.1 背景

SIMD (Single Instruction Multiple Data stream) 命令では、1 命令で同一演算、同一型の複数の命令を並列に実行できる。図 1.1 の例だと足し算のスカラ命令 4 つを、複数のスカラデータを保存できる SIMD レジスタを使用して SIMD 命令 1 命令で実行している。また、同一命令を並列実行する SIMD 命令は、大量の同一演算の命令を処理する必要があるマルチメディア処理に強い。よって、ゲームや物理現象のシミュレーションなどで高度な処理が求められている中、有効利用することが重要である。

プログラムを SIMD 化する方法として現在とられているのが、プログラマによるアセンブリ言語や SIMD 命令と 1 対 1 に対応する関数 (intrinsic ルーチン) を使って SIMD 並列化を行うことである。しかし、プログラマによる SIMD 命令の生成は特別な知識が必要になったり、プログラムがアーキテクチャ依存になってしまったりという問題がある。よって、通常のソースコードに対してコンパイラによって SIMD 命令を自動生成する技術を確立することが重要になってきている。

コンパイラで SIMD 命令を生成する方法としてパターンマッチングを行う方法がある。しかし、パターンマッチングだけだと記述したパターンと意味は同じでもわずかに書き方が違うプログラムに対して SIMD 命令を生成することができない。たとえば、配列のポインタ表現などがこれにあたる。そこで、1 つのパターンで多くの記述に対応できるようにするためにプログラム変形を行う必要がある。

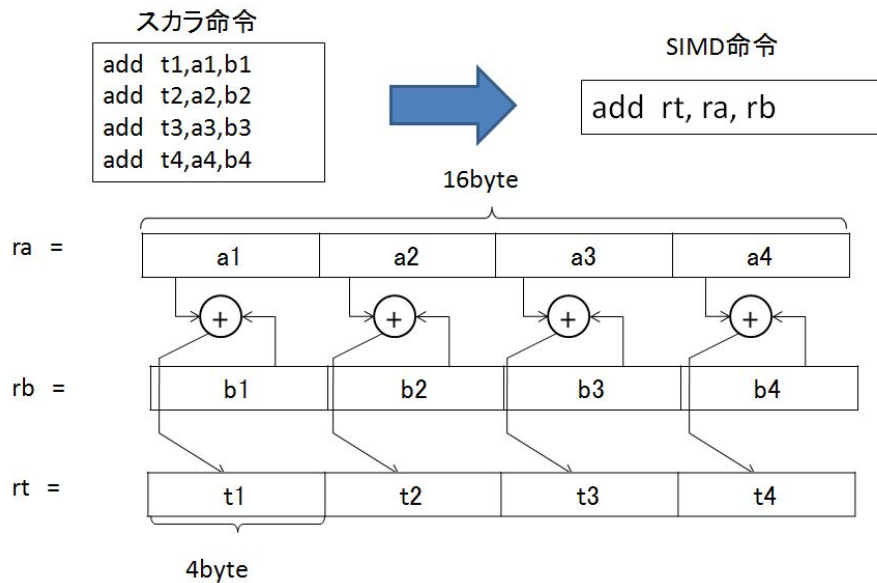


図 1.1: SIMD の例

1.2 本研究の概要

本研究で使用するコンパイラとして、ソースコードレベルとアセンブリ言語レベルで解析を行うものを仮定する。ソースコードレベルではループや配列の解析などを容易にでき、アセンブリ言語レベルでは命令スケジューリングやレジスタへの操作などソースコードレベルより細かい操作ができる。本研究では、SIMD 命令を生成するには両方のレベルでの変形が必要だと考え、ソースコードレベルでの特徴を生かした変形方法を選択し、ソースコードレベルでのコード変形器の設計を行う。また、SIMD 化できなかった場合の実行時間の増加を抑えるために、ループ単位での SIMD 化判定を行うことを目的とする。これによって、通常のスカラー最適化を行った場合より SIMD 化を行った方が常に効率のよいコードを生成することができる。

1.3 構成

本論文の構成を以下に示す。

- 2 章: 準備 (本論文で用いる用語の説明)
- 3 章: 静的単一代入 (SSA) 形式の説明
- 4 章: SIMD 化向けコード変形
- 5 章: ソースコードレベルでのコード変形器の設計
- 6 章: 並列化コンパイラ向け共通インフラストラクチャ COINS の説明

- 7章: 実験と考察
- 8章: 関連研究
- 9章: まとめ

第2章 準備

本論文の説明にあたって予備知識が必要となる。本章ではフローグラフ等の予備知識の説明を行う。

2.1 基本ブロック

基本ブロック (Basic Block) あるいはブロックとはプログラムの一部分であり、そのブロックの実行を開始する文が、そのブロックの先頭の文だけであり、末尾が無条件飛び越し、あるいは条件つき飛び越しであるものである。基本ブロック途中への飛び込みや、基本ブロック途中からの飛び出しはない。基本ブロックと、そのブロック間の関係を示した制御フローグラフ (後述) は、プログラムを解析する基本的なグラフとして現在広く用いられている [1, 26, 2, 17]。図 2.1 の (a) のような C プログラムを基本ブロックに分割したものが (b) である。図中の矩形は基本ブロックを表しており、矩形の左上の L1 等は基本ブロックにつけた便宜的なラベルである。

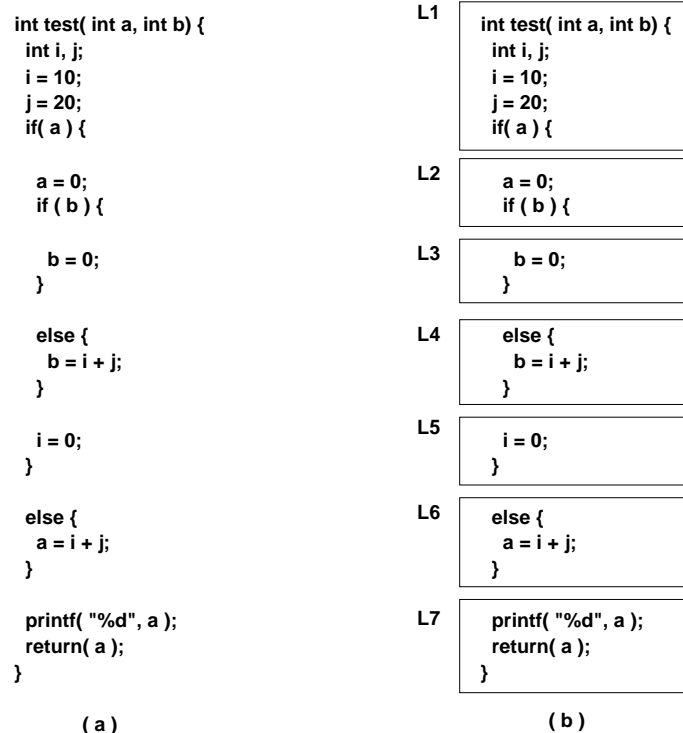


図 2.1: 基本ブロック

2.2 制御フローグラフ

制御フローグラフ (Control Flow Graph : CFG) とは, プログラムの入り口から出口までの基本ブロックを, 制御の流れに沿って図にしたグラフである [1, 26, 2]. CFG の節点 (ノード) は基本ブロックであり, ノード間是有向辺 (エッジ) で結ばれる. コンパイラでの最適化の多くは CFG の情報を用いている.

図 2.2 に, 図 2.1 に対応する CFG を示す. 図中の矩形は基本ブロックであり矢印はエッジを表している. また, 矩形の中にある L1 等は基本ブロックに対して付けた便宜的なラベルである. 各ノードのラベルは, 図 2.1 のそれと対応している. なお, 定式化によっては, これ以外に入り口ノードと出口ノードを別途付加することもあるが, ここでは立ち入らない.

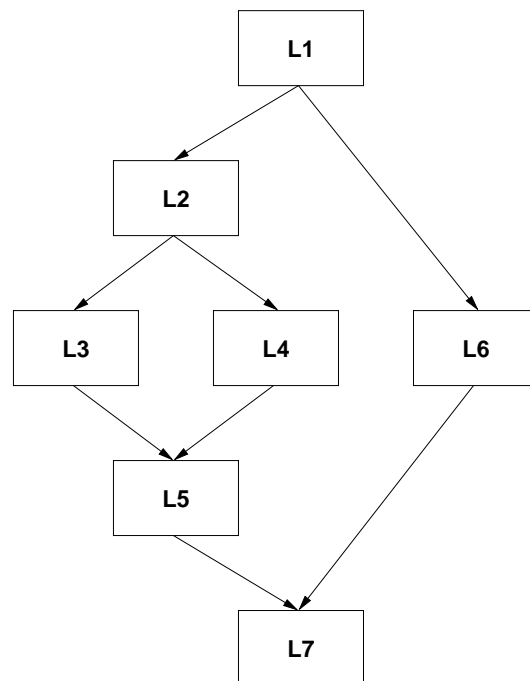


図 2.2: 制御フローグラフ (CFG)

基本ブロック X と Y が存在し, X から Y への有向辺がある場合, X は Y の先行ノード (predecessor) といい, Y は X の後続ノード (successor) と言う. 以下では X の predecessor を $pred(X)$ とし, X の successor を $succ(X)$ とする. たとえば図 2.2 の各ノードの $pred()$ と $succ()$ は以下ようになる.

- L1 : $pred(L1) = \{\}, succ(L1) = \{L2, L6\}$
- L2 : $pred(L2) = \{L1\}, succ(L2) = \{L3, L4\}$
- L3 : $pred(L3) = \{L2\}, succ(L3) = \{L5\}$
- L4 : $pred(L4) = \{L2\}, succ(L4) = \{L5\}$

- $L5 : pred(L5) = \{L3, L4\}, succ(L5) = \{L7\}$
- $L6 : pred(L6) = \{L1\}, succ(L6) = \{L7\}$
- $L7 : pred(L7) = \{L5, L6\}, succ(L7) = \{\}$

2.3 支配関係

CFG 上のふたつのノード X, Y について, CFG の入り口から Y に達するどの路も必ず X を通る場合に, X は Y を支配 (dominate) するという [1, 26, 2]. また, X は Y の支配ノード (dominator) であるという. 以下では X の支配ノードの集合を $dom(X)$ と書く. 支配関係は反射的 (reflexive) である. すなわち, ノード X は自分自身を支配する. また, 支配関係は推移的 (transitive) である. すなわち, ノード X がノード Y を支配し, ノード Y がノード Z を支配するのであれば, ノード X はノード Z を支配する [26]. 例えば図 2.2 の CFG では以下のような支配関係がある.

$$\begin{aligned}
 dom(L1) &= \{L1\} \\
 dom(L2) &= \{L1, L2\} \\
 dom(L3) &= \{L1, L2, L3\} \\
 dom(L4) &= \{L1, L2, L4\} \\
 dom(L5) &= \{L1, L2, L5\} \\
 dom(L6) &= \{L1, L6\} \\
 dom(L7) &= \{L1, L7\}
 \end{aligned}$$

ノード X がノード Y を支配し, $X \neq Y$ である場合, X は Y を厳密に支配 (strictly dominate) するという [1, 26, 2]. 例えば図 2.2 の $L1$ は $L3$ を厳密に支配している. 以下, X を厳密に支配するノードの集合を $sdom(X)$ と書く.

ノード X がノード Y を厳密に支配し, X から Y への路に, X 以外に Y を厳密に支配するノードがないとき, X は Y を直接支配 (immediately dominate) するという [1, 26, 2]. 以下では X が Y を直接支配するときに $X = idom(Y)$ と書く. 例えば図 2.2 の CFG では以下のような直接支配の関係がある.

$$\begin{aligned}
\text{idom}(L1) &= \{\} \\
\text{idom}(L2) &= \{L1\} \\
\text{idom}(L3) &= \{L2\} \\
\text{idom}(L4) &= \{L2\} \\
\text{idom}(L5) &= \{L2\} \\
\text{idom}(L6) &= \{L1\} \\
\text{idom}(L7) &= \{L1\}
\end{aligned}$$

2つのノード X, Y について, Y からプログラムの出口に達するどの路も必ず X を通るとき X は Y を後支配 (postdominate) するという [1, 26, 2].

$$\begin{aligned}
\text{pdom}(L1) &= \{L1, L7\} \\
\text{pdom}(L2) &= \{L2, L5, L7\} \\
\text{pdom}(L3) &= \{L3, L5, L7\} \\
\text{pdom}(L4) &= \{L4, L5, L7\} \\
\text{pdom}(L5) &= \{L5, L7\} \\
\text{pdom}(L6) &= \{L6, L7\} \\
\text{pdom}(L7) &= \{L7\}
\end{aligned}$$

また, X が Y を後支配し, X = Y であるとき, X は Y を厳密に後支配 (strictly postdominate) するといいい, X が Y を厳密に後支配し, Y から X への路にそれ以外に Y を厳密に後支配するノードがないとき, X は Y を直接後支配 (immediately postdominate) するという. X は Y を直接後支配するときに $X = \text{ipdom}(Y)$ と書くと, 図 2.2 の CFG では, 以下のようなになる.

$$\begin{aligned}
\text{ipdom}(L1) &= \{L7\} \\
\text{ipdom}(L2) &= \{L5\} \\
\text{ipdom}(L3) &= \{L5\} \\
\text{ipdom}(L4) &= \{L5\} \\
\text{ipdom}(L5) &= \{L7\} \\
\text{ipdom}(L6) &= \{L7\} \\
\text{ipdom}(L7) &= \{\}
\end{aligned}$$

2.4 支配木

支配木 (dominator tree) とは, CFG 上のあるノード X と, X を直接支配するノード Y を有向辺でむすんだグラフである. 得られたグラフは木構造となる. な

ぜなら, CFG のあるノード Z を直接支配するノードはひとつだからである. 支配木の根は CFG の入口ノードである. 以下, 支配木のノード X の子ノード Y を $Y \in \text{domChild}(X)$ と書く. 図 2.2 の CFG に対応した支配木を図 2.3 に示す.

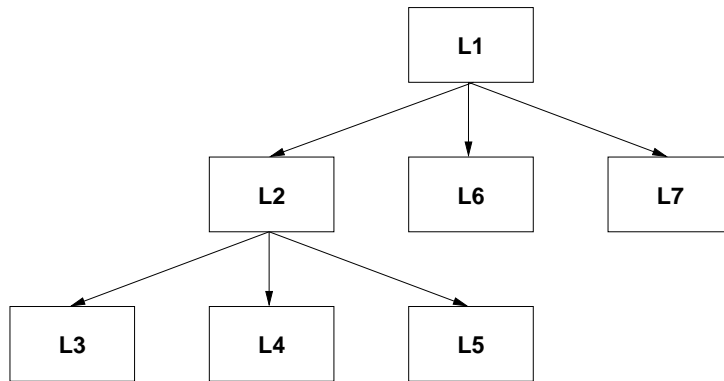


図 2.3: 支配木

2.5 支配境界

支配境界とは, CFG 上で, あるノード X からエッジを辿ってゆき, はじめて X の支配から外れたノード Y の集合である. ノード X の支配境界 $DF(X)$ は以下のように定義される [26].

$$DF(X) = \{Y | U \in \text{pred}(Y) \text{ が存在し, } X \text{ は } U \text{ を支配し, } X \text{ は } Y \text{ を厳密に支配はしない}\}$$

この定義の X と U は同じノードの場合もある. X と U が同じノードと仮定すると, $Y \in \text{succ}(X)$ であり, かつ X は Y に対して厳密な支配をしないものである. また $X \neq U$ である場合には, 定義から $X \in \text{dom}(U)$ である. ここで X が厳密な支配をするノード Z があると仮定して $DF(X)$ と $DF(Z)$ について考えてみる. $Y \in DF(X)$ であり, かつ X から Y に至る際にかならず U を通る場合は図 2.4 のような関係になる. この場合は $Y \in DF(Z)$ であることは明らかである.

逆に, $Y \in DF(Z)$ であるが $Y \notin DF(X)$ の場合の各々のノードの関係は図 2.5 のようになる. $Y \in DF(Z)$ であるから, 入口ノードから Z を通らずに Y に到達することはできる. しかし $Y \notin DF(X)$ であるため, 入口ノードから Y に到達する際には必ず X を通ることになる. 別の言葉で言えば, $X \in \text{dom}(Y)$ となり, かつ $X \in \text{sdom}(Y)$ となる.

以上のことから, X の支配境界は以下の Y の集合といえる.

- $Y \in \text{succ}(X)$ であり, $X \neq \text{idom}(Y)$
- $Z \in \text{dom}(X)$ であるときに $Y \in DF(Z)$ であり, $X \neq \text{idom}(Y)$

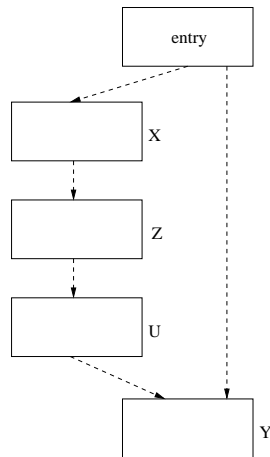


図 2.4: X の支配境界は Y であり Z の支配境界も Y の例

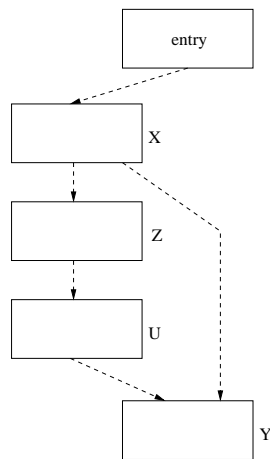


図 2.5: Z の支配境界は Y であり X の支配境界は Y ではない例

例えば図 2.2 の CFG の各ノードに対する支配境界は以下のようなになる。

$$\begin{aligned}
 DF(L1) &= \{\} \\
 DF(L2) &= \{L7\} \\
 DF(L3) &= \{L5\} \\
 DF(L4) &= \{L5\} \\
 DF(L5) &= \{L7\} \\
 DF(L6) &= \{L7\} \\
 DF(L7) &= \{\}
 \end{aligned}$$

2.6 制御依存

制御依存とは、ノード X からのある辺を辿ると、その後必ず Y を通るが、別の辺をとれば Y は通らないことを表す。次の 2 つが成り立つとき Y は X に制御依存すると定義される。

- X から Y への空でない路があり、 Y はその路の X より後のすべてのノードを後支配する
- Y は X を厳密に後支配しない

この関係を図で表すと、図 2.6 のようになる。

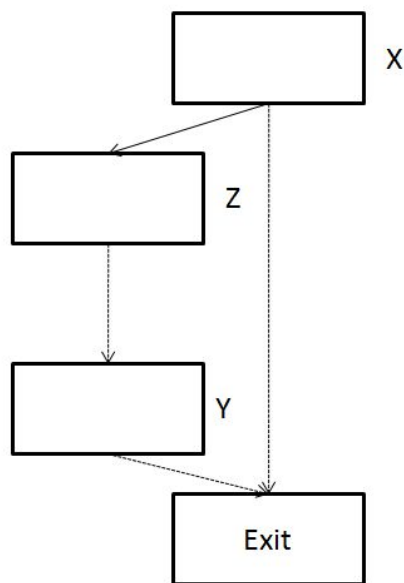


図 2.6: 制御依存

Y が制御依存するノード X から Y 方向へのエッジの集合を返す関数を $CD(Y)$ と書く。例えば図 2.2 の CFG の各ノードに対する制御依存は以下ようになる。

$$\begin{aligned} CD(L1) &= \{\} \\ CD(L2) &= \{\{L1 \rightarrow L2\}\} \\ CD(L3) &= \{\{L2 \rightarrow L3\}\} \\ CD(L4) &= \{\{L2 \rightarrow L4\}\} \\ CD(L5) &= \{\{L1 \rightarrow L2\}\} \\ CD(L6) &= \{\{L1 \rightarrow L6\}\} \\ CD(L7) &= \{\} \end{aligned}$$

2.7 ループ

2.7.1 ループ制御変数

帰納変数とは、ループの中で定数刻みで値が変わる変数で下記のように定義することができる。 $j = c1 * i + c2$ ここで、 i が帰納変数、 $c1, c2$ はループ不変式である [1, 26, 2, 17].

ループ内での代入が 1 箇所、 $i = i + c$ または $i = i - c$ の形をとるような変数のことをループのループ制御変数 (loop control variable)、またはループ変数と呼ぶ。

例えば、プログラム 2.1 の変数 i は、初期値が 1 で、ループ内で 1 ずつ値が変化していき、ループの繰返し条件式 $i < 100$ に含まれている。よって変数 i は、このループのループ制御変数である。

プログラム 2.1 (ループの例)

```
1: for ( $i = 1; i < 100; i = i + 1$ ) {  
2:    $a[i] = 1;$   
3: }
```

2.8 抽象構文木

文が与えられたとき、文法によってその文がどのように導かれるか、つまり生成規則をどのように適用すればその文が得られるかを、木の形で示すと都合がよい。このような木を解析木 (parse tree) または導出木 (derivation tree) という [17].

例として、式 2.1 に対する、表 2.1 の文法から導かれる解析木を図 2.7 に示す。

$$i_1 := (i_2 + i_3) * i_4 / \text{num} \quad (2.1)$$

解析木によって、演算とそのオペランド (被演算子) がはっきり示される。例えば、図 2.7 では、 i_2 と i_3 を加えたものに i_4 をかけ、それを num で割ったものを i_1 へ代入することが読み取れる。

```
 $S \rightarrow i := E$   
 $E \rightarrow E * E$   
 $E \rightarrow E / E$   
 $E \rightarrow E + E$   
 $E \rightarrow E - E$   
 $E \rightarrow (E)$   
 $E \rightarrow i$   
 $E \rightarrow \text{num}$ 
```

表 2.1: 簡単な文法の例

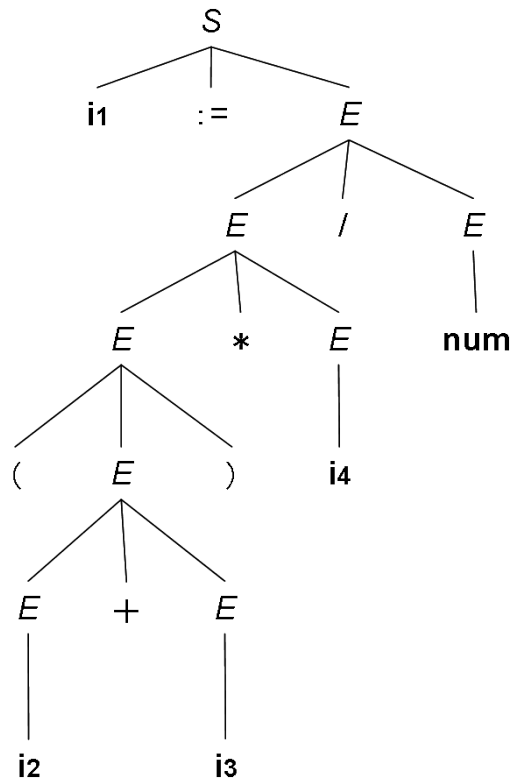


図 2.7: 式 2.1 に対する解析木

解析木は構文を正確に表しているが、演算の意味を示すには内部の節がやや冗長である。解析木のかわりに、解析木のうち演算に本質的なものだけを取り出した木を用いることがある。これは、節として演算子、節の子供として節の演算子のオペランドがくるようにした木であり、抽象構文木 (abstract syntax tree) あるいは単に構文木 (syntax tree) と呼ばれる。抽象構文木では、葉をつなげても、一般に文にはならない。式 2.1 に対する抽象構文木を図 2.8 に示す。抽象構文木には非終端記号 (表 2.1 の S と E) は現れない。演算に本質的でない構造、例えば子供が一人だけの枝や、 $()$ をまとめるためだけの構造は取り除かれる。その結果、一般に抽象構文木のほうが解析木より簡潔である。

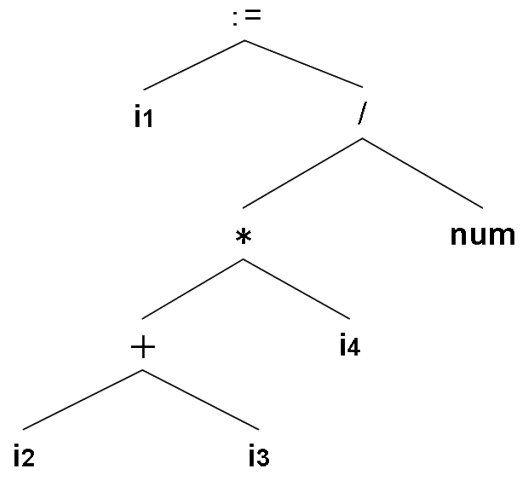


図 2.8: 式 2.1 に対する抽象構文木

第3章 静的単一代入 (SSA) 形式

本章では, 静的単一代入 (static single assignment, SSA) 形式について説明を行う.

3.1 SSA 形式

静的単一代入形式とは, 変数の定義がプログラムの字面上で唯一となるようにしたプログラムの表現形式である [26, 2, 10]. 静的とは, プログラムの字面上で, という意味である. 変数の定義が唯一になるように変数の名前替えを行うが, これは普通変数に添え字をつけてあらわす. この結果, SSA 形式では, プログラムあるいは中間表現の上で, 各変数の定義が 1 箇所だけになる. 以下に, SSA 形式の例をあげる.

プログラム 3.1 (SSA 形式の簡単な例)

```
1:  $a_1 = x_0 + y_0;$   
2:  $a_2 = a_1 + 3;$   
3:  $b_1 = x_0 + y_0;$ 
```

3.2 SSA 変換

SSA 形式に対して, 元のプログラムの形式を通常形式 (normal form) と呼ぶ. 通常形式から SSA 形式に変換することを SSA 変換 (SSA translation) という. 次のような通常形式のプログラムがあったとする.

プログラム 3.2 (通常形式の例)

```
1:  $a = x + y;$   
2:  $a = a + 3;$   
3:  $b = x + y;$ 
```

これを SSA 形式に変換すると次のようになる.

プログラム 3.3 (プログラム 3.2 の SSA 形式)

```
1:  $a_1 = x_0 + y_0;$   
2:  $a_2 = a_1 + 3;$   
3:  $b_1 = x_0 + y_0;$ 
```

たとえば a について見ると、代入があるごとに変数の新しい version を作り、 a_1, a_2 のように添え字を付けて区別する。

以下は制御の合流がある例である。次のような通常形式のプログラムがあったとする。

プログラム 3.4 (制御の合流がある例)

```
1:  $x = \dots$ ;  
2: if ( $x > 0$ ) {  
3:      $a = x$ ;  
4: } else {  
5:      $a = -x$ ;  
6: }  
7:  $print(a)$ ;
```

これを SSA 形式に変換すると次のようになる。

プログラム 3.5 (プログラム 3.4 の SSA 形式)

```
1:  $x_1 = \dots$ ;  
2: if ( $x_1 > 0$ ) {  
3:      $a_1 = x_1$ ;  
4: } else {  
5:      $a_2 = -x_1$ ;  
6: }  
7:  $a_3 = \phi(a_1, a_2)$ ;  
8:  $print(a_3)$ ;
```

文 7 が属するブロックでは、 a_1 と a_2 の値が合流するので、それ以降に a が使われていると、 a_1 か a_2 が決められなくなる。そこで、SSA 形式では ϕ 関数 (ϕ function) と呼ばれる仮想的な関数を導入する。文 7 「 $a_3 = \phi(a_1, a_2)$ 」により、これ以後使われる a は a_3 であることをあらわす。また、この ϕ 関数 $\phi(a_1, a_2)$ は、制御が文 3 「 $a_1 = x_1$ 」が属するブロックから来た時には a_1 の値を返し、文 5 「 $a_2 = -x_1$ 」が属するブロックから来た時には a_2 の値を返す関数をあらわす。

SSA 変換についての効率的なアルゴリズムは、代表的なものに

- Cytron らの方法 [10]
- Brandis らの方法 [4]
- Sreedhar らの方法 [18]
- Aycock らの方法 [3]

が知られている。

3.3 SSA 形式に基づく最適化

SSA 形式では、変数の使用に対する定義が1つだけなので、SSA 形式を用いると、コンパイラにおけるデータフロー解析やさまざまな最適化が見通しよく行える。SSA 形式を用いることで、最適化が容易になる例としてコピー伝播について説明する。

3.3.1 コピー伝播

次のようなプログラムを考える。

プログラム 3.6 (通常形式)

```
1:  $x = y$ ;  
2:  $u = 2 + x$ ;  
3:  $v = x + 1$ ;
```

$x = y$ のような複写をする代入文をコピー文 (copy statement) と呼ぶ。この時 x が再定義されるまでは、 x と y は同じ値を持つ。よってこのコピー文の後方に現れて再定義されるまで、あるいは x の別の定義が合流するまでの x は y に置き換えることが出来る。これをコピー伝播 (copy propagation) という。つまり、コピー文の後方に現れる x を y に置き換えて下のように変換できる。

プログラム 3.7 (コピー伝播)

```
1:  $x = y$ ;  
2:  $u = 2 + y$ ;  
3:  $v = y + 1$ ;
```

すると一番上のコピー文は不要なので削除することが出来る。

3.3.2 SSA 形式を用いたコピー伝播

通常、コピー伝播をするときは、コピー文に現れる変数の持つ値がどこで定義されたものかという情報を考える必要がある。

しかし、SSA 形式で表されたプログラムに現れる変数の定義は一箇所だけなので、定義される場所を考える必要がない。例えば、プログラム中に「 $a = b$ 」が現れるとする。このとき、「 $a = b$ 」を削除し、プログラム中に現れる a を b で置き換えることでプログラムの意味を換えずに「 $a = b$ 」を取り除くことが出来る。

3.4 SSA 逆変換

SSA 形式を通常形式に戻すことを SSA 逆変換 (SSA reverse translation) という。 ϕ 関数をそのまま実行できるアーキテクチャはないので、目的コードを生成する前に ϕ 関数の除去を行う必要がある。次に ϕ 関数がある場合の SSA 逆変換の例を示す。以下のような SSA 形式のプログラムがあったとする。

プログラム 3.8 (SSA 形式)

```
1:  $x_1 = \dots$ ;  
2: if ( $x_1 > 0$ ) {  
3:      $a_1 = x_1$ ;  
4: } else {  
5:      $a_2 = -x_1$ ;  
6: }  
7:  $a_3 = \phi(a_1, a_2)$ ;  
8: print( $a_3$ );
```

これを SSA 逆変換すると次のようになる。

プログラム 3.9 (プログラム 3.8 の SSA 逆変換の結果)

```
1:  $x_1 = \dots$ ;  
2: if ( $x_1 > 0$ ) {  
3:      $a_1 = x_1$ ;  
4:      $a_3 = a_1$ ;  
5: } else {  
6:      $a_2 = -x_1$ ;  
7:      $a_3 = a_2$ ;  
8: }  
9: print( $a_3$ );
```

前述したとおり、プログラム 3.8 の文 7 「 $a_3 = \phi(a_1, a_2)$ 」は、どちらの基本ブロックからこの合流点に来たかによって a_3 の値を決めるものであった。そこで、プログラム 3.9 では、通常形式でそれを復元するために、合流点に入ってくるそれぞれの基本ブロックの最後に文 4 「 $a_3 = a_1$ 」と文 7 「 $a_3 = a_2$ 」のコピー文を置き、 ϕ 関数を消去する。

ただし、このままでは無駄なコピー文があるので、合併 (coalescing) という手法を使って、それらを取り除くことが多い [5, 7, 12, 14]。プログラム 3.9 で a_1, a_2, a_3 を合併すると、元の通常形式であるプログラム 3.4 と同じものが得られる。

しかし SSA 逆変換では、上記の素朴な方法がうまくいかない例が知られている [6]。それらの問題点を解決した SSA 逆変換についての効率的なアルゴリズムは、代表的なものに、

- Briggs らの方法 [6]
- Sreedhar らの方法 [19]
- Rastello らの方法 [15]

が知られている。

3.5 SSA 形式の性質

性質 3.1 (SSA 形式) 元プログラムのすべての変数 v に対して, 次の 3 条件を満たすとき, そのプログラムは SSA 形式になっているという。

- もしブロック Z がパス $X \rightarrow Z$ と $Y \rightarrow Z$ (ブロック X とブロック Y は v の定義を含んでいる) の最初の結合ブロックならば, v に対する ϕ 関数が Z の先頭に挿入されている。
- v に対する新しい名前 v_i はプログラムテキスト上で唯一の定義を持つ。
- 任意の制御フローグラフのパスにおいて, v に対する新しい名前 v_i の使用と, 元プログラムでそれに対応する v の使用を考える。このとき v と v_i は同じ値を持つ。

第4章 SIMD化向けコード変形

SIMD化向けのコード変形には、ソースコードレベルとアセンブリ言語レベルの2つの段階が考えられる。本研究では、この2つの段階でのコード変形の操作を明確に分ける。

4.1 ソースコードレベルでの変形

ソースコードレベルでの変形では、ソースコードレベルで表現可能な変形を文単位まで行うことにする。

ソースコードレベルで行う変形の特徴を下記に示す。

- if文やforループなどの制御構造が容易に認識できる
アセンブリ言語レベルだとこれらの制御構造は分岐命令を組み合わせることによって実現される
- 配列構造が容易に認識できる
アセンブリ言語レベルだと配列構造がそのアドレス計算を含むいくつかの命令列に分解されてしまう

本研究では、以上のようなソースコードレベルの特徴を踏まえて、SIMD命令生成のための変形手法の中からソースコードレベルに適したものを選び実装を行った。さらに、アセンブリ言語レベルでの変形とセットで行うことにより、SIMD命令とのパターンマッチングを目指す。本研究の対象はこのレベルでの変形として、5章で詳しく説明を行う。

4.2 アセンブリ言語レベルでの変形

アセンブリ言語レベルでの変形では、命令単位での変形を行う。ソースコードレベルとの違いとして、レジスタやアーキテクチャ特有の命令などを意識した変形が行える。また、演算レベルでのより細かい単位で命令スケジューリングを行うことができる。ここでは、このレベルで行う研究について示す。

SIMD命令では、16byteなどのSIMD命令専用のレジスタを持っていて、メモリ上で連続している配列参照などのデータをSIMDのロード命令を使うことによって、1命令でSIMDレジスタに代入することができる。しかし、このロード命令を使うにはデータの先頭アドレスが16byte境界でなければならないなどの制約がある。こ

れに対し shift 命令や shuffle 命令などを使用してデータを整えるアルゴリズム [21] が開発されている。

また，SIMD レジスタ上のデータ位置を考慮して，SIMD 命令数を減らす研究 [22] もおこなわれている。このように，SIMD レジスタや SIMD 命令を意識した変形というのは命令レベルで行うべきである。

第5章 ソースコードレベルでのコード変形器の設計

5章では、提案するソースコードレベルでのコード変換器の設計について述べる。図 5.1 には、コード変形器全体のフローチャートを示した。また、これらの手法はすべてループ単位で行うことにする。

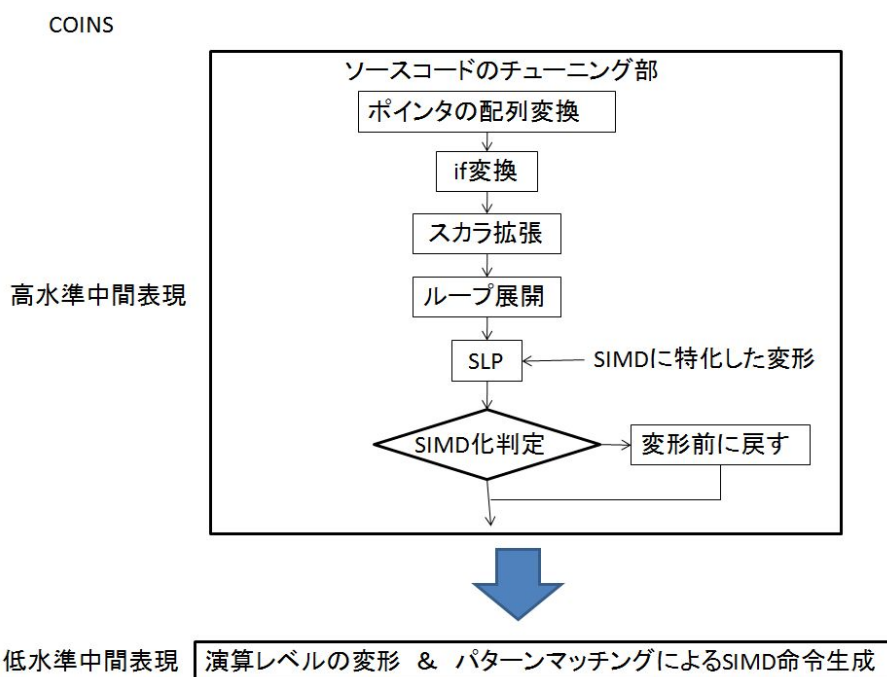


図 5.1: ソースコードレベルでのコード変形器

5.1 ポインタ配列の配列変換

C言語では、配列がポインタ表現として書かれていることが多い。しかし、ポインタ配列をそのまま使用するとコンパイラによる解析を難しくする。これは、SIMD化を行う場合でも問題となる。そこで、Symbolic Analysisの手法を使用して、ポインタ配列を標準形式に変形して、配列表現に逆変換する手法 [24] が提案されている。本研究では、この変形のアルゴリズムを図 5.1 のようにコード変形の最初に行うことによってそれ以降の変形のアルゴリズムでポインタ配列を考慮する必要がなくなるようにした。

次に、ポインタ表現の配列を配列に変換するアルゴリズムについて説明する。ポインタ演算の解析は、帰納変数を解析することと同等とみなすことができる。帰納変数とは、ループの中で定数刻みで値が変わる変数で下記のように定義することができる。

$$j = c1 * i + c2$$

ここで、 i が帰納変数、 $c1, c2$ はループ不変式である。Engelen らはこの帰納変数を解析するために CR (Chains of Recurrences) 形式を導入した。CR 形式というのは Zima によって開発され、後に Bachamann, Zima, Wang によって改良された。CR 形式を説明するのに下記のようなプログラムのポインタ $F(i)$ について考える。

```

F[0] :=  $\phi_0$ 
cr1 :=  $\phi_1$ 
      :
crk :=  $f_k$ 
for i = 1 to n do
    F[i] := F[i - 1]  $\odot_1$  cr1
    cr1 := cr1  $\odot_2$  cr2
          :
    crk-1 := crk-1  $\odot_k$  crk
end for

```

ここで、 $cr_j, j = 1, \dots, k$ は、帰納変数であり、 $\odot \in \{+, *\}$ である。ポインタ $F(i)$ を解析しようとしたとき、 $F(i)$ は帰納変数である cr_1 と依存関係にある。また、 cr_1 は cr_2 と依存関係がある。ここで、CR 表現を使うことによって、ループ中のポインタ $F(i)$ をそれと依存関係がある帰納変数と合わせて 1 つの CR 表現で簡単に表すことができる。ポインタ $F(i)$ に対する CR 形式は下記のように書く。

$$F[i] = \{\phi_0, \odot_1, \{\phi_1, \odot_2, \dots, \{\phi_{k-1}, \odot_k, f_k\}_i\}_i\}_i$$

これは、カッコを省略して、

$$F[i] = \{\phi_0, \odot_1, \phi_1, \odot_2, \dots, \phi_{k-1}, \odot_k, f_k\}_i$$

のように書くことができる。

この CR 表現を使って、
ポインタ配列 \Rightarrow CR 形式 \Rightarrow 配列
の手順で変形を行う。

次に、CR 表現を使って配列を表し方について説明する。ポインタアクセスを CR 形式で表すと、

$$\{\phi_0, +, \dots, f_k\}_i$$

となり、 ϕ_0 はポインタ表現となり、 $\phi_j (j=1, \dots, k-1)$ と f_k は、整数型の表現となる。つまり、配列 $a[n]$ で言うと、これはポインタ表現の $*(a+n)$ と同じ意味を表わし、 ϕ_0 が a であり、 $\phi_j (j=1, \dots, k-1)$ と f_k が n にあたる。

次に、プログラムを使ってアルゴリズムの説明を行う。図 5.2 に配列変換アルゴリズムによって変形を行うプログラムの例を示す。このプログラムは、行列の掛け算のプログラムをポインタ配列で行っている。

```

1: int *p_a = &A[0] ;
2: int *p_b = &B[0] ;
3: int *p_c = &C[0] ;
4:
5: for (k = 0 ; k < Z ; k++)
6: {
7:     p_a = &A[0] ;/* point to the beginning of array A */
8:
9:     for (i = 0 ; i < X; i++)
10:    {
11:    p_b = &B[k*Y] ;          /* take next column */
12:
13:    *p_c = 0 ; //次に計算するcの値を初期化
14:
15:        for (f = 0 ; f < Y; f++){ /* do multiply */
16:        *p_c = *p_c + *p_a++ * *p_b++ ;
17:        }
18:
19:    *p_c++ ; //次に計算するcにアドレスを進める
20:
21:    }
22: }

```

図 5.2: 配列に変換するポインタプログラム例

i プログラムの変形

最初に，CR 変形が行いやすいように，プログラム変形を行う．この変形では，変数の更新を伝播させて，単一代入の形式にする．たとえば，プログラム中に

$$p = p + j;$$

$$p = p + 1;$$

のような文があった場合，

$$p = (p + j) + 1;$$

のような文に変形する．また，このような帰納変数をループの最後に集める．

図 5.2 のプログラム例では，16 行目の文

$$*p_c = *p_c + *p_a++ * *p_b++ ;$$

に対して， p_a , p_b は帰納変数なので，別の文としてループの最後に置くと，

```

15: for (f = 0 ; f < Y; f++){ /* do multiply */
16:     *p_c = *p_c + *p_a * *p_b;
17:     p_b = p_b + 4*1;
18:     p_a = p_a + 4*1;          }

```

のようになる．ここで， $p_b + 1$ ではなく， $p_b + 4 * 1$ となっているのは，バイトアドレッシングであるからである．

ii 帰納変数のCR変形

帰納変数のCR変形は，最内部のループから順番に行う．下記のプログラム中の3行目，4行目には，それぞれ $V = V + C$ の形をした帰納変数があるので，これをCR形式で置き換えると，

```

1: for (f = 0 ; f < Y; f++){ /* do multiply */
2:     *p_c = *p_c + *p_a * *p_b;
3:     p_b = p_b + 4*1;
4:     p_a = p_a + 4*1;          }

```

が

```

1: for (f = 0 ; f < Y; f++){ /* do multiply */
2:     *p_c = *p_c + *p_a * *p_b;
3:     p_b = {p_b0, +, 1}_f;
4:     p_a = {p_a0, +, 1}_f;
5: }

```

のようになる． $\{p_b0, +, 1\}_f$ は，ループ変数 f のループ中で p_b がループを1回まわるごとに1つつ増えていることを表している．また， p_a0, p_b0 は，ループ変数が f のループの先頭での p_a, p_b の値である．

iii CRの置き換え

ここで，CRに置き換えられた文の左辺が使用されている部分を右辺のCRで置き換える． p_c はこのループでの帰納変数の定義がないので， $\{p_c0, +, 0\}_f$ で置き換えると，

```

1: for (f = 0 ; f < Y; f++){ /* do multiply */
2:     *p_c = *p_c + *p_a * *p_b;
3:     p_b = {p_b0, +, 1}_f;
4:     p_a = {p_a0, +, 1}_f;
5: }

```

が

```
1: for (f = 0 ; f < Y; f++){ /* do multiply */
2:     *{p_c0, +, 0}_f = (*{p_c0, +, 0}_f) + (*{p_a0, +, 1}_f) * (*{p_b0, +, 1}_f)
3:     p_b = {p_b0, +, 1}_f;
4:     p_a = {p_a0, +, 1}_f;
5: }
```

のようになる .

次に, CR 変形されたプログラムの正規化を行う . このルールの一部を表 5.1 に示す . これは, 0 加算や, 1 の乗算などの必要のない計算を取り除く . また, CR に定数を足した場合や, CR どうしの演算などのルールもある . 行列の掛け算の例では, 適用できるルールはなかった .

表 5.1: CR の正規化ルール

	変換前		変換後	条件
1	$\{\phi_0, +, 0\}_i$	\Rightarrow	ϕ_0	
2	$\{\phi_0, *, 1\}_i$	\Rightarrow	ϕ_0	
3	$\{0, *, f_1\}_i$	\Rightarrow	0	
4	$-\{\phi_0, +, f_1\}_i$	\Rightarrow	$\{-\phi_0, +, -f_1\}_i$	

また, ループの正規化も行う .

```
for(f=a; f<=b; f=f+s)
```

のようなループを,

```
for(f=0; f<=(b-a)/s; f++)
```

に変形する . さらに, ここでは帰納変数の移動を行う . 帰納変数 p_b, p_a がこのループの出口で生存していたら, 1 つ外側のループへ出し, 生存していなかったら取り除く . たとえば, p_a の場合, ループの出口で生存しているので, 1 つ外側のループへ出すことができる . ここで, 表 5.2 にあるような CR 形式から元の形に戻す CR^{-1} 変形を行う . 表 5.2 のルール 1 を適用すると,

```
p_a = {p_a0, +, 1}_f;
```

が

```
p_a = p_a0 + {0, +, 1}_f;
```

となり, ルール 9 を適用すると,

```
p_a = p_a0 + 1 * f;
```

となる . ここで, ループ変数 f が出てきたので, これを $(b-a+s)/s$ で置き換える . この例では, $(b-a+s)/s$ は Y となるので,

```
p_a = p_a0 + Y;
```

となり, これをループの外へ出す .

この後，2，3の手順をループの内側から外側に向かって実行する．

表 5.2: CR^{-1} 変換のルール

	変換前	変換後	条件
1	$\{\phi_0, +, f_1\}_i$	$\Rightarrow \phi_0 + \{0, +, f_1\}_i$	$\phi_0 \neq 0$
2	$\{\phi_0, *, f_1\}_i$	$\Rightarrow \phi_0 * \{1, *, f_1\}_i$	$\phi_0 \neq 0$
9	$\{0, +, f_1\}_i$	$\Rightarrow i * f_1$	i は f_1 の中には存在しない
10	$\{0, +, i\}_i$	$\Rightarrow (i^2 - i)/2$	

iv ポインタ変数を初期化する

次に，9行目のCR式にある p_{-a_0} , p_{-b_0} , p_{-c_0} を実際の値で置き換える．この操作も内側のループから順番に行う．図5.3を使って， p_{-a_0} を初期化するアルゴリズムの説明をする． p_{-a_0} の初期化とは， p_{-a_0} はループ変数 f のループ中の値なので，8行目のループの入り口に入ってくる時の値のことをいう．この初期化の操作は，図5.3の番号順に行う．まず，操作1として，3行目からプログラムの実行の逆を辿って， p_{-a} の定義を探す． p_{-a} は3行目で初期値 A が与えられているのがわかる．しかし，この値は，4行目のループ変数 i のループの入り口の値である． p_{-a_0} を初期化するには，ループ変数 f のループの入り口の値を求める必要がある．そこで，14行目の操作2でループ変数 i のループの帰納変数 p_{-a} の初期値を A で置き換える．次に，表5.2のルールに従って CR^{-1} 変換を行う． CR^{-1} 変換は，CR形式にある文を配列の先頭アドレス+インデックスに書き換えることができる．ここで，14行目の操作3にあるように CR^{-1} 変換を行う．まず，

$$\{A, +, Y\}_f$$

に表5.2のルール1を適用すると，

$$A + \{0, +, Y\}_f$$

のようになる．次に，ルール9を適用すると，

$$A + i * Y$$

となる．この値が p_{-a_0} になるので，9行目の操作4で p_{-a_0} を $A + i * Y$ に置き換える．同様にして， p_{-b_0} , p_{-c_0} も初期化する．

```

1: for (k = 0 ; k < Z ; k++)
2: {
3:   p_a = A + 4*0; ← 1 ループiにおけるp_aの初期値発見
4:   for (i = 0 ; i < X ; i++)
5:   {
6:     p_b = B + (4 * (k * Y));
7:     *{p_c0,+ ,1}_i = 0
8:     for (f = 0 ; f < Y ; f++){
9:       *{p_c0,+ ,0}_f = (*{p_c0,+ ,0}_f) + (*{p_a0,+ ,1}_f) × (*{p_b0,+ ,1}_f)
10:      {A+i × Y,+ ,1}_f
11:     }
12:   }
13:
14:   p_a = {p_a0,+ ,Y}_i → 2 p_a = {A,+ ,Y}_i → 3 p_a = A+i × Y
15:                                     CR-1
16: }
17: p_c = {p_c0,+ ,X}_k
18: }

```

図 5.3: ポインタ変数 p_a_0 を初期化する

また，ポインタ変数を初期化することによって必要なくなったポインタ代入文は除去することができる．しかし，下記にある代入文は除去できない．

- $p = \text{malloc}(n)$
- $p = \&a[b[i]]$
- $p = f(n)$

v 配列への変形

最後に表 5.2 にある CR^{-1} を適用して配列への変換を行う．図 5.3 の 9 行目の文

$$*\{C + k * X + i, +, 0\}_f = (*\{C + k * X + i, +, 0\}_f) + (*\{A + i * Y, +, 1\}_f) * (*\{B + k * Y, +, 1\}_f)$$

に表 5.2 のルール 1 を適用すると，

$$*(C + k * X + i + \{0, +, 0\}_f) = (*(C + k * X + i + \{0, +, 0\}_f)) + (*(A + i * Y + \{0, +, 1\}_f)) * (*(B + k * Y + \{0, +, 1\}_f))$$

のようになる．さらに，ルール 9 を適用すると，

$$*(C + k * X + i) = *(C + k * X + i) + *(A + i * Y + f) * (*(B + k * Y + f))$$

のようになる．これは，配列の先頭アドレス+インデックスの形をしているので，

$$C[k * X + i] = C[k * X + i] + A[i * Y + f] * B[k * Y + f]$$

のように変形できる．

このアルゴリズムには 3 つの制約がある．

- ポインタの別名解析を行わない
- break 文のあるループに対しては行わない

- 関数間の解析は実行されない

これらの制約に当てはまるループは配列への変換がされないので，SIMD化は行われない．

5.2 if変換

SIMD 並列化を行うにあたって if 文の扱い方が問題になってくる．ループ展開を行った同一の if 文に対しても，要素ごとの分岐方向は異なるため，1つの方向で計算をしてしまうと間違っただ値が含まれていることがある．例えば，図 5.4 のような絶対値を求めるプログラムがある．

```
1: for (i = 0; i < SIZE; i++) {  
2:     if (in[i] > 0) {  
3:         out[i] = in[i];  
4:     else  
5:         out[i] = -in[i];  
6: }
```

図 5.4: 絶対値を求めるプログラム

これを SIMD 化するためにループ 4 回分を展開すると，

```

1: for (i = 0; i < SIZE; i=i+4) {
2:     if (in[i] > 0)
3:         out[i] = in[i];
4:     else
5:         out[i] = -in[i];
6:
7:     if (in[i+1] > 0)
8:         out[i+1] = in[i+1];
9:     else
10:        out[i+1] = -in[i+1];
11:
12:    if (in[i+2] > 0)
13:        out[i+2] = in[i+2];
14:    else
15:        out[i+2] = -in[i+2];
16:
17:    if (in[i+3] > 0)
18:        out[i+3] = in[i+3];
19:    else
20:        out[i+3] = -in[i+3];
21: }

```

のようになる。このプログラムに対してif文のthenパート、elseパートをそれぞれSIMD命令で並列に計算すると、図5.5のようにそれぞれのレジスタで2個ずつの誤ったデータが含まれてしまうので、if文をSIMD化して正しいデータを得るためには工夫が必要である。

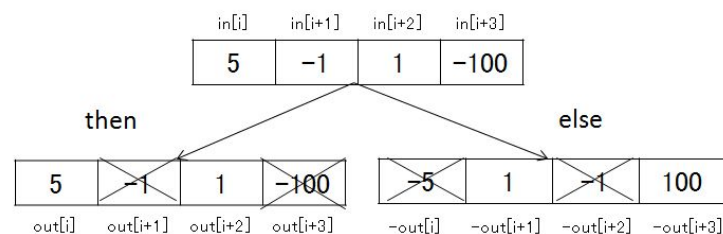


図 5.5: 条件分岐の例

そこで本研究では、図5.6のようなフローチャート図に従ってif変換を行う。最初にif変換を行うかどうかの判定を行う。これは、if文内部の文の数が多かった場合、if変換を行ってSIMD化してもプログラムが高速化されない可能性があるからである。なぜなら、if変換というのは条件が成り立っていてもいなくてもif文内部の計算を実行する変形であるからである。例えば、thenパートに100個の文がある場合

を考える．通常のプログラムでは条件が false の場合，100 文は実行しないが，if 変換を行うと常に 100 文を実行することになる．そこで，if 文内部の文の数がしきい値より小さい場合のみ if 変換を行うことにする．

if 変換には，パターンマッチングと select 文向きの変換という 2 つの手法をとっている．これは，ある特定の形にマッチングすると select 文向きの変換よりプログラムが高速に実行できるためである．次に，この 2 つの手法について詳しく説明する．

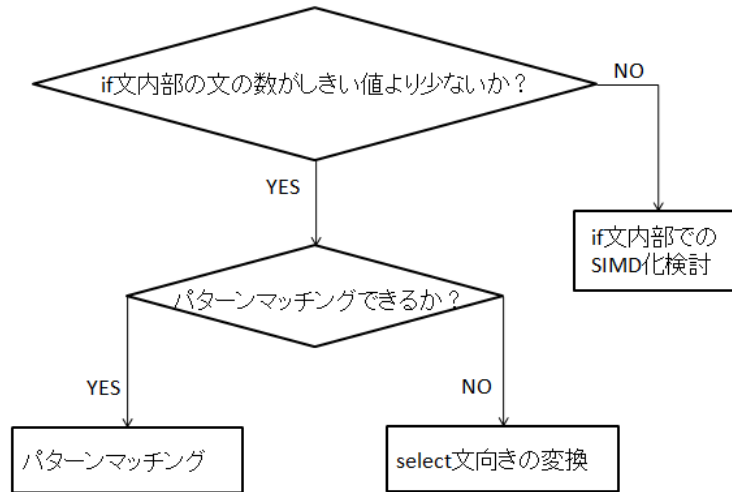


図 5.6: if 変換のフローチャート

5.2.1 パターンマッチング

パターンマッチングで行う方法として現在は 2 つの方法を実装している．どちらの手法も SIMD の比較命令の演算結果である真/偽が-1/0 となっていることに着目したものである．

1. SIMD の比較演算が生成するマスク値を-1 と 0 の数値として使用する場合



2. SIMD の比較演算が生成する-1 と 0 をマスク値として使用する場合



また、この変形が行えるのは、true が-1になる時 (SIMD の比較演算) だけなので下記のようなマスクを使って明示的に与えることにする。

```

if (cond)
{
    mask_0 = (int )-1;
}
else
{
    mask_0 = (int )0;
}

```

上記のようにマスク値を与え、

```
a += -mask_0
```

のようにマスク値を使って論理式をあらわすことにする。上記のマスクを生成した if 文は simd 命令を生成するときに条件式とマスク文のセットで SIMD 命令とのマッチングを行えばよい。

5.2.2 select 文向きの変換

SIMD プロセッサには select 命令というデータ選択命令を備えているプロセッサがある。図 5.4 の絶対値を求めるプログラムに対して select 命令を使用して SIMD 化したものが図 5.7 のようになる。then パート、else パート、条件式を 4 回分計算したものをそれぞれ SIMD レジスタ va, vb, cond に保存をする。then パート、else パートを計算したレジスタには、図 5.5 で示したように、誤ったデータが含まれているので、select 命令を使用して正しいデータを選択する必要がある。select 文を使用すると、cond のそれぞれのデータでの真偽によって、va,vb からデータを取り出して、if 文を実行した最終的な結果が得られる。

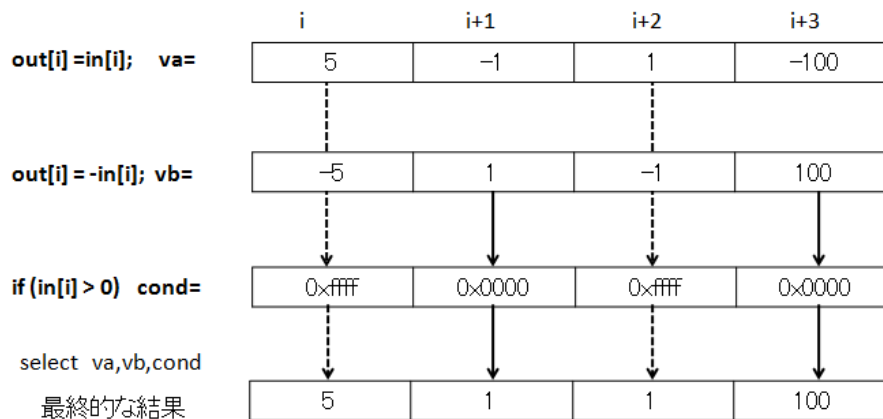


図 5.7: Select 文の例

この節で説明する変形方法は、この `select` 文をコンパイラによって自動的に生成するためのものである。また、`select` 文がないプロセッサに対しては `select` 文と同値の論理式を生成する。

IA64 の命令セット用にネストしている `if` 文にも適用可能なアルゴリズム [25] が提案されている。これは、SSA 変換のアルゴリズムを応用したものである。本研究では、この手法をソースコードレベルで SIMD 命令生成のために行うことにする。

ここでは、図 5.8 のようなプログラムを例にアルゴリズムの解説を行う。

```
1: if(a[i] < x){
2:     a[i] = x;
3:     if(a[i] < z)
4:         a[i] = z;
5: }
```

図 5.8: `if` 変換を行うプログラム例

`if` 変換を行うには制御フローグラフ上での解析が必要である。ここで、図 5.8 に対する制御フローグラフは図 5.9 のようになる。

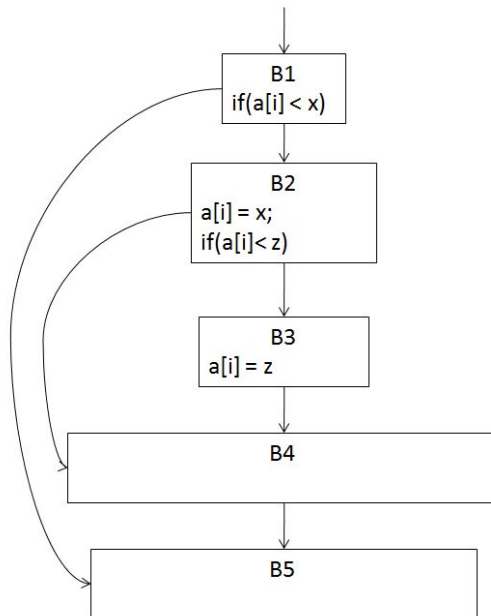


図 5.9: 図 5.8 の制御フローグラフ

図 5.9 のノードの `pred()` と `succ()` は以下ようになる。

```
pred(B2) = {B1}, succ(B2) = {B3, B4}
pred(B3) = {B2}, succ(B3) = {B4}
pred(B4) = {B2, B3}, succ(B4) = {B5}
```

次に、if 変換のアルゴリズムについて説明する。

i ループ中の if 文を探索する

1 回の if 変換では if 文が見つかった所から、if 文のある基本ブロックを直接後支配する基本ブロックまでを範囲とする。これによって、上記の例では、if 文がネストしているが、最外部の if 文全体を 1 つの if 変換を行う単位として行う。また、if 文中に break 文がある場合は、if 変換は行わないが、continue 文があった場合は、if 変換の対象とする。その場合は、continue 文がある if 文の入口ブロックからそのブロックを直接後支配するループの最後のブロックまでを 1 つの if 変換の単位として処理する。

ii 支配境界 (Dominance frontier) を求める

図 5.9 の制御フローグラフの各ノードに対する支配境界は以下ようになる。

$$DF(B2) = \{B5\}$$

$$DF(B3) = \{B4\}$$

$$DF(B4) = \{B5\}$$

iii 見つけた if 文に対して支配境界に phi 命令を挿入する

ここでの phi 命令は、SSA 形式の ϕ 関数と似ているが、phi 命令は、 $\text{phi}(\text{cond}), v1, v2$ という形をとっており、cond によって、true の場合 v1、false の場合 v2 を選ぶ。また、phi 命令を並列化できるものを複数個集めたものが SIMD の select 命令となる。

iv if 文内部のみ SSA 形式に変形する

図 5.9 の制御フローグラフを SSA 形式にすると図 5.10 のようになる。なお、SSA 形式に変換するのは、if 変換を行う if 文内部だけなので、if 変換の出口ブロックである B5 では a[i] を一時変数で置き換えていない。ここで SSA 形式にする目的は、文と文の依存関係をなくすためである。これによって、if 変換を行って if 文を 1 つの基本ブロックにしたとき、それぞれの変数の定義使用関係が保たれる。

また、phi 文の右边を設定するために使う phi-list を作成する。phi-list の求め方は、 ϕ 関数の求め方と同じである。phi 文を定義したブロックから phi-list には ϕ 関数との違いとして、変数名と一緒にその変数が定義された基本ブロックの情報も保存する。基本ブロック情報は、phi 文の条件式を求めるために使用する。

図 5.10 の制御フローグラフを見ると、B4 の phi-list は B3 の支配境界となり追加されたものである。この B4 の phi-list には、B2, B3 のように変数とペアでその変数が定義されている基本ブロックの情報も保存されている。また、phi-list の要素は基本ブロックの番号順になっている。

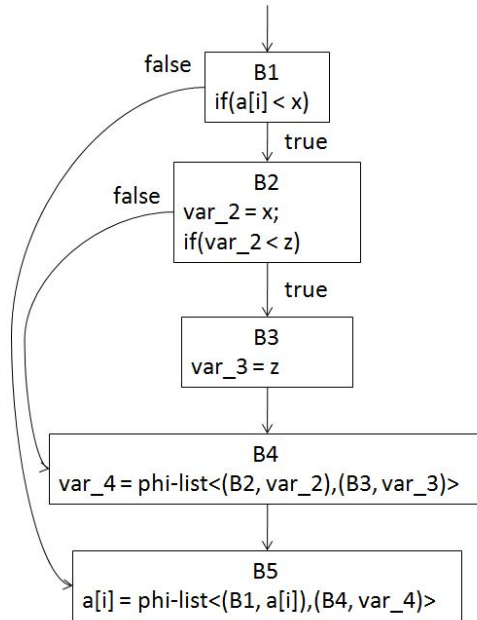


図 5.10: phi-list を作成

v 制御依存 (control dependent) を求める

Y が制御依存するノード X から Y 方向へのエッジの集合を返す関数を $CD(Y)$ と書く。図 5.10 の制御フローグラフの各ノードに対する CD は以下のようになる。

$$\begin{aligned}
 CD(B1) &= \{\} \\
 CD(B2) &= \{\{B1 \quad B2\}\} \\
 CD(B3) &= \{\{B2 \quad B3\}\} \\
 CD(B4) &= \{\{B1 \quad B2\}\} \\
 CD(B5) &= \{\}
 \end{aligned}$$

vi それぞれの基本ブロックに対して GP (Guarding Predicate) を求め、select 文の右辺の値を決める

GP とは、それぞれの基本ブロックが実行される時 true、されないとき false になるように定めた条件式である。

GP を求めるアルゴリズムを図 5.11 に示す。ループ中のすべての基本ブロックに対して、制御依存の数で場合分けをしているのがわかる。0 の場合というのは、if 変換の入口ブロックと出口ブロックに当たる。これらのブロックは必ず通るので、true になる。6 行目にある $PE(edge\{X \quad Z\})$ は、X の条件式で Z 方向を進むエッジを選んだとき、true となる条件式を返す。図 5.10 の制御フローグラフに対する PE は以下のようになる。

$$\begin{aligned}
PE(\{B1 \ B2\}) &= a[i] < b \\
PE(\{B1 \ B5\}) &= !(a[i] < b) \\
PE(\{B2 \ B3\}) &= var_2 < z \\
PE(\{B2 \ B4\}) &= !(var_2 < z)
\end{aligned}$$

1の場合に、このPEを使用してGPを設定する。5行目にあるYが制御依存しているエッジとの関係は図5.11のようになる。図5.11の6行目で、関数PEに5行目のエッジを引数にとって、論理式Pを求める。7行目で、Xが常に通るブロックであるときは、GP(Y)がPになる。常に通るブロックでないときは、PとGP(X)を&した論理式がGP(Y)となる。たとえば、図5.13のB3をみると、p3をPE({B2 B3}) & p2により求めているのがわかる。また、1以上ある場合が書かれていないが、switch文を対象にすると考慮する必要があるが本研究ではif文のみ考えることにする。

```

1: for each ループ do
2:   for each ブロック Y(トポロジカルソートされた順序) do
3:     if |CD(Y)| = 0 then GP(Y) を true にする
4:     if |CD(Y)| = 1 then
5:       E{X Z}   CD(Y)
6:       P := PE(E)
7:       if GP(X) = true then
8:         GP(Y) := P
9:       else
10:        GP(Y) := P & GP(X)
11:      end if
12:      //if |CD(X)| > 1 は、switch 文のみ
13:    end for
14: end for

```

図 5.11: GP を求めるアルゴリズム

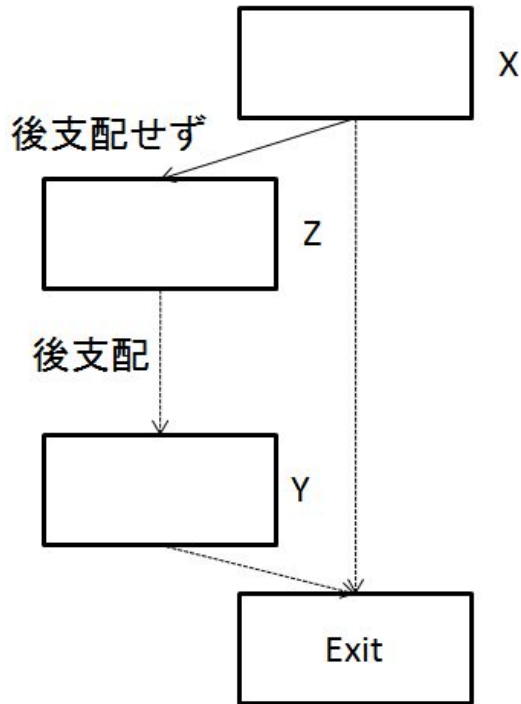


図 5.12: 図 5.11 に対する基本ブロックの関係

GP は phi 文の条件式として使用される．GP を用いて phi-list から phi 文を生成するアルゴリズムを図 5.13 に示す．通常の if 文は phi-list の要素が 2 個であるので，アルゴリズムの if 文では 9,13 行目が実行される．まず，2 行目から 5 行目で $\{block1, var1\}$ に phi-list の 2 番目の要素を代入し， $\{block2, var2\}$ に phi-list の 1 番目の要素が代入される．ブロック 1 の GP を求めて，phi-list のある代入文の右辺を phi (exp) var1, var2 で置き換える．これで，exp が true のとき var1 が実行される phi 文が作成される．また，phi-list の要素が 3 個以上ある場合 (continue 文がある場合) は，一時変数を使用する必要がある．たとえば，

$a[i] = phi-list < (B2, var_2), (B3, var_3), (B8, var_8) >$

のような phi-list があったとき，

$temp = phi(GP(B8)) var_8, var_3$

$a[i] = phi(GP(B8)|GP(B3)) temp, var_2$

のように 2 つの phi 文に置き換えることができる．アルゴリズムでは，15 から 20 行目でこのような変形が実行される．

```

1: for phi-list が空ではない
2:   phi-list の最後の要素を{block1, var1}へ代入し,
3:   phi-list からその要素を削除する
4:   phi-list の最後の要素を{block2, var2}へ代入し,
5:   phi-list からその要素を削除する
6:   if block1 = null(17 行目で追加された要素) then
7:     exp := tempExp
8:   else
9:     exp := GP(block1)
10:  end if
11:
12:  if phi-list が空
13:    phi-list を phi (exp)var1, var2 で置き換える
14:  else
15:    新しい変数を作成する (temp)
16:    phi 文の前に
17:    temp := phi (exp)var1, var2;
18:    を追加する .
19:    phi-list の最後に{null, temp}を追加する
20:    tempExp := exp | GP(block2)
21:  end if
22:end for

```

図 5.13: phi-list から phi 文を生成するアルゴリズム

図 5.14 にはそれぞれの基本ブロックに対応する GP が書かれている . また , その GP を利用して図 5.10 の phi-list から phi 文への変換を行っている .

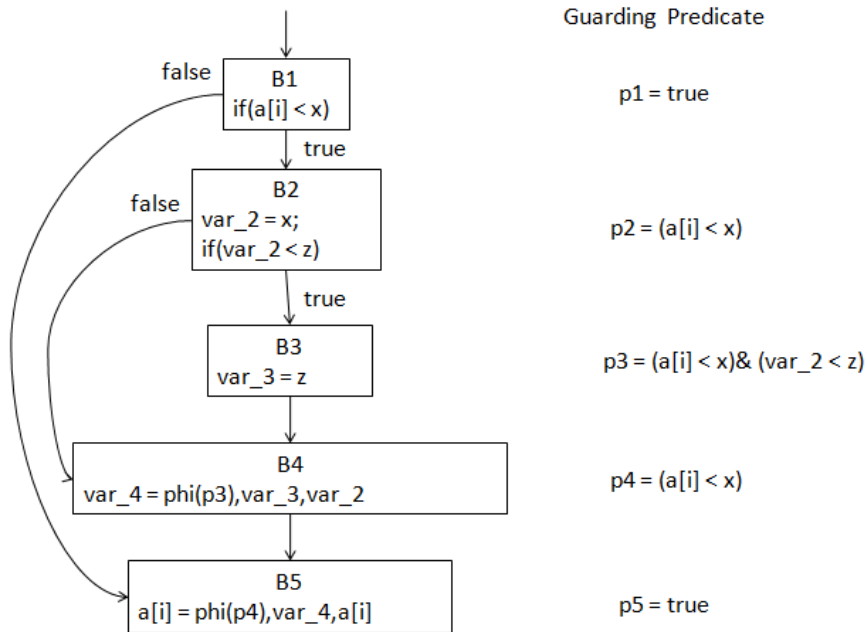


図 5.14: 制御フローグラフと対応する GP

vii マスク文を追加して if 文を取り除く

select 文はソースコードレベルで直接表現できないので、代わりに論理式を使って表現し、命令レベルの処理でこの形を認識して、select 文を生成してくれることを期待することにする。phi (cond), v1, v2 の代わりになる論理式は、 $(v1 \& cond) | (v2 \& \sim cond)$ のように書く必要がある。しかし、前節のパターンマッチングで示したように、この変形が行えるのは、true が-1 になる時 (SIMD の比較演算) だけなのでマスクを使って明示的に与る必要がある。マスクを使って書くと、

```

if (cond)
{
    mask_0 = (int )-1;
}
else
{
    mask_0 = (int )0;
}
(v1 & mask_0) | (v2 & ~mask_0)

```

のようになり、命令生成では、マスク生成の if 文も含めて select 文とマッチングすることを期待する。また、select 文がない SIMD 命令セットでは上記のような論理式に対応する SIMD 命令を生成すればよい。最終的に出力されるプロ

グラムは、図 5.15 のようになる。図 5.14 に対して、トポロジカルソートをした基本ブロックの順序 (ブロック番号順) で代入文を出力した結果である。

```
1: //B2
2: var_2 = x;
3: //B3
4: var_3 = z;
5: //B4
6: if ((a[i] < x) & (var_2 < z))
7: {
8:     mask_0 = (int )-1;
9: }
10: else
11: {
12:     mask_0 = (int )0;
13: }
14: var_4 = (var_3 & mask_0) | (var_2 & (~mask_0));
15: //B5
16: if (a[i] < x)
17: {
18:     mask_1 = (int )-1;
19: }
20: else
21: {
22:     mask_1 = (int )0;
23: }
24: a[i] = (var_4 & mask_1) | (a[i] & (~mask_1));
```

図 5.15: if 変換されたプログラム

5.3 スカラ拡張

スカラ拡張とは、変数の依存関係をなくすために配列を導入することである。たとえば、

```
1: for(i=0; i < N; i++){
2:     T = A[i] + B[i];
3:     C[i] = T + 1/T;
4: }
```

を

```

1: //Mは展開数
2: for(i=0; i < N; i++){
3:     Tx[i%M] = A[i] + B[i];
4:     C[i] = Tx[i%M] + 1/Tx[i%M];
5: }
6: T = Tx[M-1];

```

のように変換する．Mは展開数となっており，次の5.4節で説明するループ展開で与える展開数のことである．ここでは，変数Tが1つのループ内で定義使用関係があり，ループをまたいだ依存関係がないので，配列Txの導入を行った．これによって，例えばループ2回分を展開した場合，

```

1: for(i=0; i < N-1; i+=2){
2:     Tx[i%2] = A[i] + B[i];
3:     C[i] = Tx[i%2] + 1/Tx[i%2];
4:
5:     Tx[(i+1)%2] = A[i+1] + B[i+1];
6:     C[i+1] = Tx[(i+1)%2] + 1/Tx[(i+1)%2];
7: }
8: for(; i < N; i++){
9:     T = A[i] + B[i];
10:    C[i] = T + 1/T;
11: }
12: T = Tx[1]

```

のようになる．また，配列Txのインデックス $(i+x)\%2$ ($0 \leq x < 2$) は，iが $2t$ ($0 \leq t < N-1$) となり，インデックスが $(2t+x)\%2$ となるので，xになる．したがって，

```

1: for(i=0; i < N-1; i+=2){
2:     Tx[0] = A[i] + B[i];
3:     C[i] = Tx[0] + 1/Tx[0];
4:
5:     Tx[1] = A[i+1] + B[i+1];
6:     C[i+1] = Tx[1] + 1/Tx[1];
7: }
8: for(; i < N; i++){
9:     T = A[i] + B[i];
10:    C[i] = T + 1/T;
11: }
12: T = Tx[1]

```

のようになり，Tx[0] と Tx[1] の依存関係がなくなるので，並列に実行できる．

5.4 ループ展開

ループ展開では，ループがネストしている場合，最内部のループのみを展開の対象とする．SIMD レジスタサイズとプログラムのデータサイズを考えて展開数を変える必要がある．SIMD のレジスタサイズを n byte，演算のデータサイズ m byte であった場合ループの展開数は n/m (と余り) となる．図 5.16 の例だと，16byte の SIMD レジスタを使用し，4byte のデータを扱っているので，展開数は 4 となる．また，ループ中に異なるデータサイズの計算がある場合の m の値は，それぞれの SIMD 化できる文の最小のサイズを指定する．

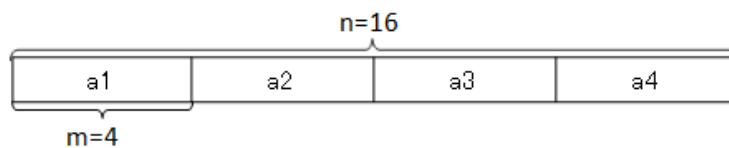


図 5.16: SIMD レジスタとデータサイズ

5.5 Super Level Parallelism

SLP 解析 [27] では，同一の演算を同一の型で行う文を見つけて SIMD 命令のスケジューリングを行う．SLP 解析の手順を以下のような例を使って説明する．

```
b1 = a[i+0]
c1 = 5
d1 = b1 + c1

b2 = a[i+1]
c2 = 6
d2 = b2 + c2

b3 = a[i+2]
c3 = 7
d3 = b3 + c3
```

1. 隣接メモリ参照を検知する

プログラム中で配列参照を行っているところを探して，メモリ上で隣接しているものを組にする．

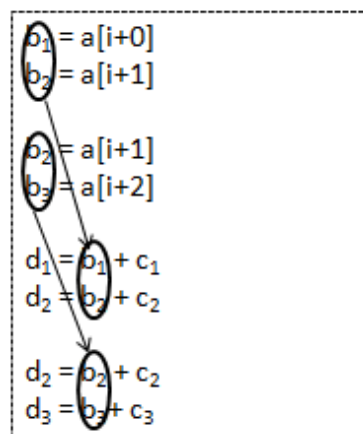
```

b1 = a[i+0]
b2 = a[i+1]

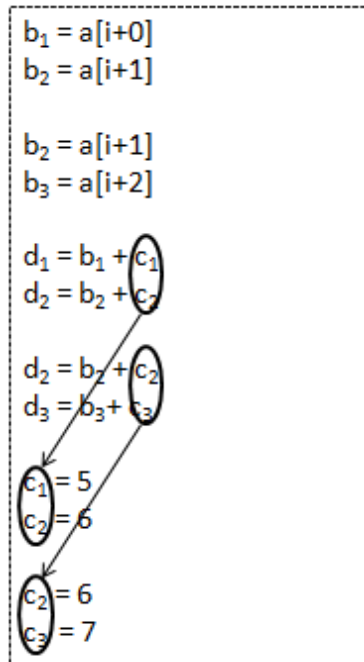
b2 = a[i+1]
b3 = a[i+2]

```

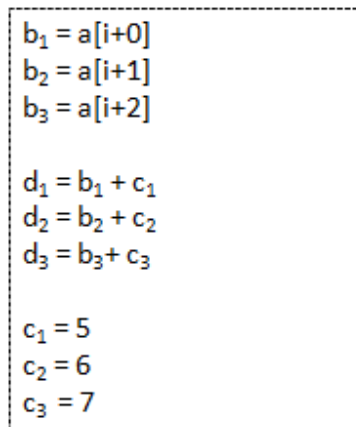
2. 組にした命令と def-use 関係にある組を探す
 組にした命令で定義されている変数が，使用されている命令を新たに組みとして追加する．



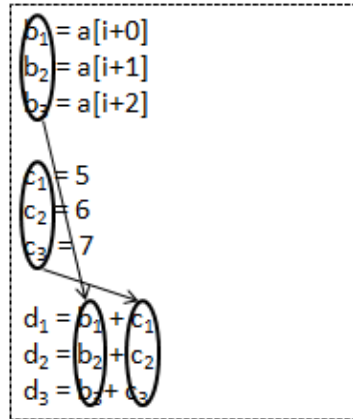
3. 組にした命令と use-def 関係にある組を探す
 組にした命令で使用されている変数が，定義されている命令を新たに組みとして追加する．



4. それぞれの組を結合して1つのSIMD命令で実行できる命令列を作る
 例では1つ目の組と2つ目の組で、 $b_2 = a[i+1]$ が重なっているので、2つのブロックを結合する



5. 元のプログラムと比較して、命令列間の依存関係を調べる
 4での作業時の命令順序では、 (c_1, c_2, c_3) のdef-use関係が逆になっているので2番目の命令列と3番目の命令列を入れ替える。



本研究での SLP 解析は基本的には Larsen のアルゴリズム [27] を使用するが、アルゴリズムの対象が 3 番地コードとなっていて、コード生成までを行っている。しかし、本研究の SLP 解析の目的は、SIMD 化可能かどうかの判定を行うことにある。よって、文全体が同等かどうかの判定までを行うことにする。

SIMD 命令に特化した変形

この段階でソースコードレベルで記述できる SIMD 命令に特化した変形 [29] ができる場合は行っている。この変形について、ここでは下記のプログラム例を使って説明する。

```
unsigned char a[M], b[M];
int c = 0, i = 0;
for(; i < M; i++)
    if(a[i] > b[i]) c++;
```

まず、このプログラムに対して本手法を適用すると、

```

static unsigned int t[8];
int lc = 0;
for(c = 0,i=0; i < M-7; i+=8){
    t[0] += -(a[i+0] > b[i+0]);
    t[1] += -(a[i+1] > b[i+1]);

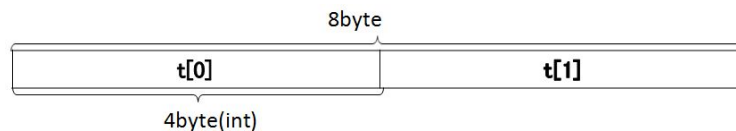
    t[2] += -(a[i+2] > b[i+2]);
    t[3] += -(a[i+3] > b[i+3]);

    t[4] += -(a[i+4] > b[i+4]);
    t[5] += -(a[i+5] > b[i+5]);

    t[6] += -(a[i+6] > b[i+6]);
    t[7] += -(a[i+7] > b[i+7]);
}
c += t[0] + t[1] +...+t[7];
for(;i < M; i++)
    if(a[i] > b[i]) c++;

```

のようになる。ここで、本来ならマスクの生成も行うがここでは省略する。このプログラムは、if文があるが5.2.1節で紹介したマッチングのテンプレートにマッチする。また、配列 t はスカラ拡張により導入されたものである。ループ展開数は、ここでは SIMD レジスタサイズを 8byte で、ループ中で使用されている最も小さなデータは配列 a, b の 1 byte なので、ループ 8 回分の展開を行う。しかし、この展開したプログラムを SIMD 化する場合、配列 t のサイズが 4byte であるので、配列 a と b の比較演算の結果 (1byte) のキャスト (char から int) を行い配列 t との加算を行う。このように行くと SIMD レジスタを



のように使用しなければならず、2命令しか並列に実行できない。しかし、このプログラムは工夫することにより、8命令を1命令で実行することができる。その工夫を行ったプログラムは下記のようなになる。

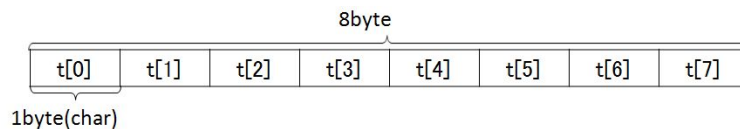
```

static unsigned char t[8];
int lc = 0;
for(c = 0,i=0; i < M-7; i+=8){
    t[0] += -(a[i+0] > b[i+0]);
    t[1] += -(a[i+1] > b[i+1]);
    t[2] += -(a[i+2] > b[i+2]);
    t[3] += -(a[i+3] > b[i+3]);
    t[4] += -(a[i+4] > b[i+4]);
    t[5] += -(a[i+5] > b[i+5]);
    t[6] += -(a[i+6] > b[i+6]);
    t[7] += -(a[i+7] > b[i+7]);

    if(lc == 255){
        c += t[0] + t[1] ... + t[7];
        t[0:7] = 0; lc = 0;
    } else lc++;
}
c += t[0] + t[1]+... + t[7];
for(;i < M; i++)
    if(a[i] > b[i]) c++;

```

この変形では配列 t の型を `1byte` とする．こうすることによって， a , b の比較演算の結果をそのまま使って配列 t と加算することができ，下記のような SIMD レジスタを使い 8 命令を並列に実行することができる．



しかし，このままでは寄せ集めた結果を保存する変数 c が 4byte なので，1byte である配列 t に保存したデータがオーバーフローしてしまう可能性がある．そこで， lc (ループカウンタ) を導入することによって，ループを 255 周することに配列 t の寄せ集めを行っている．このような手法は，ソースコードレベルで記述することができるので，この段階で行うことにする．

5.6 SIMD 化判定

現在 SIMD 化が行えるコンパイラで SIMD 化が難しいプログラムをコンパイルすると，通常のスカラでの最適化よりも実行時間が遅くなってしまうことがある．図 5.17 に動画圧縮プログラムである `quant5`[28] に対して変形していないプログラムと

SIMD 化向けの変形を行ったプログラムに対して gcc -O3 でコンパイルした結果を示す。このグラフは、原型を 1 とした実行時間の相対値である。グラフより SIMD 化向けの変形が SIMD 化できなかったとき、スカラでの最適化よりも遅くなってしまうことがあることがわかる。この実行時間の増加の主な理由は if 変換によるものである。quant5 にはネストしている if 文があり、その if 文を本研究の if 変換を行うことによるオーバーヘッドが表れている。if 変換の説明でも述べたが if 変換は条件が成立していてもいなくても文が実行されてしまう。たとえば、

```
if(a>0){
  x[i]= exp1
  y[i]= exp2
  z[i]= exp3
}
```

のようなプログラムでは $a \leq 0$ のとき文は実行されないが、if 変換を行うと、

```
var1 = exp1
var2 = exp2
var3 = exp3
x[i] = phi(a>0) var1, x[i]
y[i] = phi(a>0) var2, y[i]
z[i] = phi(a>0) var3, z[i]
```

となり常に実行されてしまう。また、図 5.17 ではこの if 変換を行ったプログラムをループ展開することによってさらに実行時間が増加してしまっている。しかし、if 文がないループに対してはループ展開により常に実行時間が増加するとは言えない。逆に、ループ展開を行ってループ制御命令の実行回数が減ることによって実行時間が改善されることもある。しかし、本研究ではループ展開によりループ中の命令数が増え、その分必要とするレジスタの数も増えてしまうという問題を考え、実行時間の改善する可能性があるとしても SIMD 化不可能だったら変換前のコードに戻すことにする。

本研究では、SLP 解析の命令スケジューリングを行った情報によってループ全体の SIMD 化判定を行う。本研究の SIMD 化判定で SIMD 化できないと判断される文は、

- ポインタ配列を配列変換できない
- break 文や return 文により if 変換ができない文
- メモリアドレスが隣接していない
- 関数呼び出し
- while 文
- switch 文

である。現在の判定方法では、SLP 解析で SIMD 命令が 1 命令でも生成できると判断されたら、変形されたコードの出力を行っている。1 命令も SIMD 化できなかった場合、変換前のコードと置き換える。このような判定を行うことによって、プロ

グラム中のループがすべてSIMD化できなかった場合でも、スカラ最適化をした時と同じ実行時間になり、実行時間が増加してしまうことがなくなる。

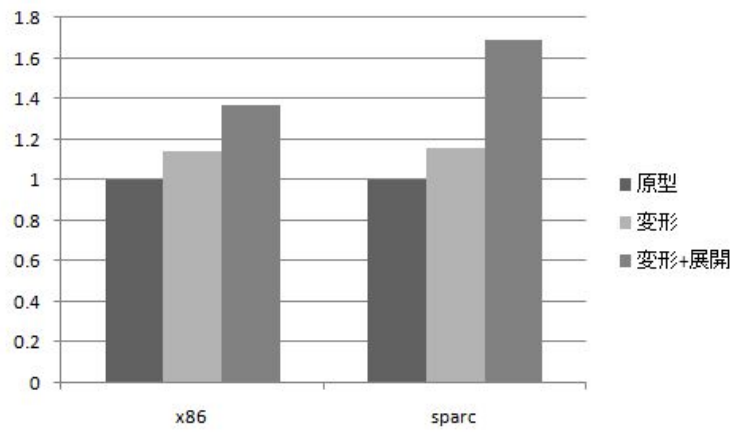


図 5.17: 変形前との実行時間の比較

第6章 並列化コンパイラ向け共通インフラストラクチャCOINS

本章では、本手法の適用を行う際に利用した並列化コンパイラ向け共通インフラストラクチャ (COmpiler INfraStructure, COINS) について説明を行う。

6.1 概要

COINS は、コンパイラ研究の基盤となる共通のコンパイラインフラストラクチャの作成をテーマに 2000 年度より研究が進められているプロジェクトである [8]。現在他機関でも、多くの最適化コンパイラの研究、開発が行われているが [11, 13, 20]、本研究室の佐々政孝教授が COINS の研究プロジェクトに携わっているため、本研究室では COINS をインフラとして利用した研究が多く行われており、データの蓄積が行いやすいため、本手法の実装を行うインフラに COINS を選択した。

6.2 構成

一般にコンパイラは、フロントエンド (front end) とバックエンド (back end) から構成される。フロントエンドは、原始プログラム (source program) を中間コード (intermediate code) と呼ばれる内部形式に変換する。バックエンドは、中間コードを計算機の機械コード (machine code) に変換する。フロントエンドはさらに、字句解析器 (lexical analyzer)、構文解析器 (syntax analyzer)、意味解析器 (semantic analyzer) に分けられる。バックエンドは、最適化器 (optimizer) とコード生成器 (code generator) に分けられる。これら各部分は、コンパイラのフェーズと呼ばれる。

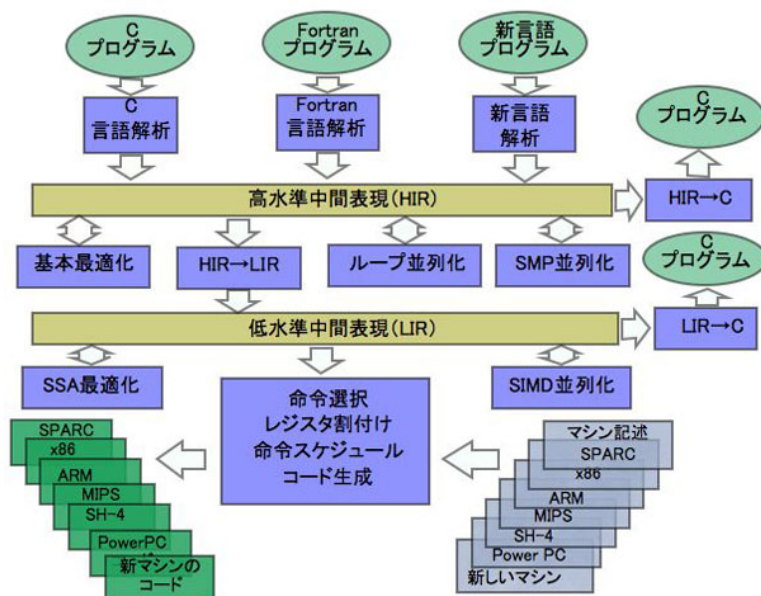


図 6.1: COINS 構成図

COINS では、複数の入力言語、複数の対象機種に対応する 2 つの中間コードがある (図 6.1)。入力言語の論理構造に近いレベルの中間コードを、高水準中間表現 (high-level intermediate representation, HIR)¹ と呼び、機械語に近いレベルの中間コードを、低水準中間表現 (low-level intermediate representation, LIR)² と呼ぶ。

6.3 COINS の特徴

COINS の特徴として 2 つの中間表現 (HIR, LIR) がある。ここで、HIR はソースコードレベルの表現と対応し、LIR は命令レベルでの表現と対応する。本研究の実装で使用する HIR は、おもにコンパイラによる並列化を行うための中間表現で、下記のような特徴がある。

- if 文や for ループなどの制御構造が容易に認識できる
LIR だとこれらの制御構造は分岐命令を組み合わせることによって実現される
- 配列構造が容易に認識できる
LIR だと配列構造がそのアドレス計算を含むいくつもの命令列に分解されてしまう

本研究では、以上のような特徴から並列化のための変形を容易に行うことができるソースコードレベルの中間表現のある COINS を利用した。

¹以後 HIR と呼ぶ

²以後 LIR と呼ぶ

6.4 高水準中間言語 HIR について

本節では、本研究で使用する中間表現である HIR について説明する [9].

6.4.1 概要

HIR は一般的な手続き型言語において共通的な概念を抽出し、特定の言語に依存しないように表現された抽象構文木である。HIR 自身は Java 言語で実装される。

6.4.2 具体表現

HIR をテキストで表すとき、節は

(オペレータ 型 第1子 第2子 … 第n子)

葉は

< 種別 記号 型 >

と表す。HIR ではコンパイル単位全体を1つの木として表現する。その中には一般に、複数の副プログラム定義が含まれる。例えば次のプログラムがあったとする。

プログラム 6.1 (入力プログラム)

```
int fact(int p) {  
    if (p <= 1)  
        return 1;  
    else  
        return p * fact(p - 1);  
}
```

このプログラムから生成される HIR は図 6.2 のようになる。

```

(prog
  <null 0 void>
  <nullNode>
  (subpDef void
    <subp <SUBP < int > false false int> fact>
  <null void>
    (labeldSt void
      (list <labelDef _lab1>)
      <block void
        (if void
          (cmpLe bool <var int p> <const int 1>)
          (labeldSt int
            (list <labelDef _lab3>)
            (return int <const int 1>))
          (labeledSt int
            (list <labelDef _lab4>)
            (return int
              (mult int
                <var int p>
                (call int
                  (addr <PTR <SUBP < int > false false int>>
                    <subp <SUBP < int > false false int> fact>)
                  (list
                    (sub int <var int p> <const int 1>))))))
            (labeledSt void
              (list <labelDef _lab5>)
              <null void>))))))
  )

```

図 6.2: プログラム 6.1 の HIR

第7章 実験と考察

7.1 実装

実装は，COINS 上の高水準中間表現 HIR 上におこなった．ループ展開はすでに COINS での実装が行われている．そこで，本実装では SIMD 化向けの展開数を指定する部分のみ実装を行った．また，COINS を使うことにより HIR の制御フローグラフや変換を行うメソッドが使用できたので容易に実装できた．本研究で実装したのはソースコードレベルの変形だけなので実際に SIMD 命令の生成を行うには，命令レベルでの変形やコード生成を行う機能が必要である．COINS 上には命令レベルでの SIMD 命令生成器が実装されているが，これはプロトタイプ実装ということもあり SIMD 命令への変換性能があまり高くない．よって，高水準中間表現上で提案手法の実装を行い，低水準中間表現は通さずに変形した中間表現を C 言語へ逆変換を行ったものを用いた．

7.2 実験

テストプログラムは，論文中にある条件に合う件数を数える例題 [29] と SIMD ベンチマーク [30] からそれぞれ `count`，`quant5` を使用した．これらは，ループ中に `if` 文があり SIMD 化をするには `if` 変換を要するものである．また，マルチメディア向けベンチマークプログラムである `DSPstone`[28] から `matrix`，`convolution` を使用した．これらは，ポインタ配列を用いているので配列変換の効果を検証できる．これらの4つのプログラムは，アセンブリ言語レベルでのアライメントを整える処理が必要でないものを選んだ．

実験方法は表 7.1 に示す．実験では，原型のプログラム，提案手法で変形したプログラムそれぞれに対して，`icc` (Var. 11.0) のスカラ最適化 (`-O3`)，SIMD 化 (`-axN -O3`) を適用した．もう1つは，変形したプログラムに対して，本研究で SIMD 化可能と判断された部分に対して手で組み込み関数 [33] を使い SIMD 化を行ったものである．たとえば，`DSPstone`[28] の `matrix` に対して，ポインタ配列の配列変換を行ったプログラムの一部は，

```
C[k*Z+i] = C[k*Z+i] + A[i*X+f] * B[k*Z+f];
C[k*Z+i] = C[k*Z+i] + A[i*X+f+1] * B[k*Z+f+1];
C[k*Z+i] = C[k*Z+i] + A[i*X+f+2] * B[k*Z+f+2];
C[k*Z+i] = C[k*Z+i] + A[i*X+f+3] * B[k*Z+f+3];
```

のようになる．これに組み込み関数を適用すると，

```

Areg = *(__m128*)&A[i*X+f];
Breg = *(__m128*)&B[k*Z+f];
Creg = _mm_add_ps(Creg, _mm_mul_ps(Areg, Breg));

```

のようになる。_mm_add_ps(SIMDの足し算), _mm_mul_ps(SIMDの掛け算)が組み込み関数であり, Areg, Breg, CregがSIMDレジスタ(16byte)である。また, それぞれの実験に対する目的は下記のようなになる。

表 7.1: 実験方法

実験	ソースコードレベル	コード生成
1	原型	icc -O3
2	原型	icc -axN -O3 (SIMD)
3	提案手法で変形	icc -axN -O3 (SIMD)
4	提案手法で変形	組み込み関数

実験 1 基準

スカラ最適化を行ったプログラムを基準にして, SIMD化により基準よりも実行時間が改善されているプログラムに対して, SIMD化の効果があったと判定できる。

実験 2 icc の SIMD 命令生成能力

現在使用されている SIMD 命令を生成できるコンパイラである icc で, ポインタ配列や if 文へどれだけ対応できるかを考察する。

実験 3 変形が icc の SIMD 化を助けるかどうか

実験 3, 4 は本研究でのコード変形器を使用した。本研究で作成したのはソースコードレベルでのコード変形器なので, 変形された C 言語のコードを SIMD 命令を生成できるコンパイラの入力として与えることができる。このような使い方を想定したとき, 実験 2 と比べてコード変形がどの程度実行時間に影響を与えるかを考察する。

実験 4 ソースコードレベルでの期待通りにコード生成が行われた場合

本研究ではアセンブリ言語レベルでのコード変形器とコード生成器の実装は行っていない。そこで, テストプログラムとしてアセンブリ言語レベルでの変形が必要でないプログラムを選び, コード生成では組み込み関数を使用した。この実験では, 実験 1 と比べて SIMD 化を行うとどれだけ実行時間の改善されるかを考察する。

実験環境は, 表 7.2 に示す。

表 7.2: Let's note CF-W5 の仕様

機種	Let's note CF-W5
プロセッサ種別	Intel Core Duo 1.06GHz
プロセッサ数	2
1次キャッシュ	2 × 32KB
2次キャッシュ	2MB
メモリ容量	512MB
オペレーティング環境	Ubuntu-8.04

7.3 考察

提案手法で変形したときと提案手法を適用していない原型での実行時間の比較を図 7.1 に示す。このグラフは、原形でスカラ最適化をした場合を 1 としている。これらの実験に対する考察を下記に示す。

実験 2 icc の SIMD 命令生成能力

icc では簡単な if 文（ネストしてない）に対して if 変換を行い、SIMD 命令を自動生成することができる。しかし、ネストしている if 文とポインタ配列があるプログラムに対しては、SIMD 命令を生成することができなかった。

実験 3 変形が icc の SIMD 化を助けるかどうか

実験 2 と比べて、実行時間は改善しなかった。quant5 ではスカラ最適化と比べて、1.7 倍の実行時間となってしまった。これは、本研究では SIMD 化できないと判定されたループがスカラ最適化されると期待している。また、SIMD 化できると判定されたループに対して SIMD 命令が生成されると期待している。しかし、icc がこれらの期待通りのコード生成ができていないことが原因だと考えられる。したがって、icc 向けにソースコードレベルでの変形器を使用する場合は、icc で SIMD 化できるプログラムの形を実験で調べることにより、コード変形器の生成するコードの形をチューニングする必要がある。

実験 4 ソースコードレベルでの期待通りにコード生成が行われた場合

この実験は、ソースコードレベルでコード変形を行い、SIMD 化可能と判断されたループに対して、ソースコードレベルで期待されるコード生成を命令レベルで行ったと仮定した場合である。実験結果より、50% 以上の実行時間の減少が見られた。これからわかるように、提案手法を使って変形したのに対して命令レベルで期待されるコード生成を行うと SIMD 化による効果が期待できる。

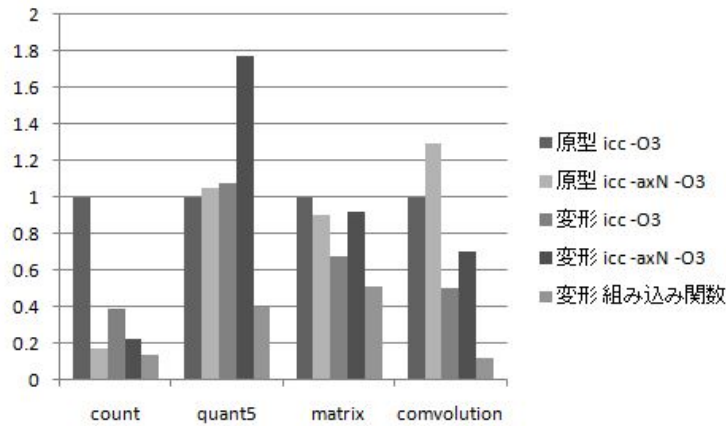


図 7.1: 実行時間の比較

また，DSPstone のすべてのプログラムの for ループに対して，SIMD 化ができるかどうかの判定を行い下記の 3 つに分類する静的な実験を行った．それぞれのパターンとプログラム中の出現率を下記に示す．

パターン 1 SIMD 化可能と判断され，命令レベルでの処理が必要ない 出現率 40%
アライメントを整える必要がない

パターン 2 SIMD 化可能と判断されるが，命令レベルでアライメントを整える必要がある 出現率 13%

SIMD 命令にはアドレスが 16byte 境界でないとロードできないなどの制約がある．下記の例だと， $a[0]$ ， $b[0]$ ， $c[0]$ が 16byte 境界で配列の要素のサイズが 4byte だとすると，それぞれ 8, 4, 12byte ずれているのでアライメントを整える必要がある．

```
例：for(i=0; i<100; i++){
    a[i+2] = b[i+1] + c[i+3];
}
```

パターン 3 SIMD 化不可能と判断される 出現率 47%

SLP のスケジューリングではメモリ上で隣接しているデータにのみ SIMD 化可能だとした．SIMD 化不可能と判断されるのは，下記のようなプログラムである．配列の要素の 1 つ置きに同一の演算子の計算をしている．

```
例：for (i = 0; i < N*2; i+=2)
{
    d[i] = a[i]+b[i];
    d[i+1] = a[i+1]*b[i+1];
}
```

図 7.1 では上記のパターン 1 のループに対して，SIMD 化による実行時間の改善を見た．これから，それぞれのパターンのループに対して SIMD 化によりどれだけのオーバーヘッドがあり，SIMD 化による実行時間の改善が望めるかの考察を行う．

まず，SIMD 並列化を行うことによるオーバーヘッドとして，SIMD 命令を使用するには SIMD レジスタへロードしなければいけないことと，SIMD レジスタのデータは通常のレジスタへ置き換えなければスカラ演算ができないことがあげられる．1 はメモリ上に連続しているデータを 1 命令でロードできるパターンなので，このオーバーヘッドが少なくて済み，実行時間の改善が期待できる．また，DSPstone ではプログラム中に 40%出現しているので，マルチメディア系のプログラムに対して十分実行時間の改善が見込めることがわかる．2 は 1 でのオーバーヘッドに加えて，アライメントを整えるオーバーヘッドがある．本研究では，アライメントを整えるアルゴリズム [21] はアセンブリ言語レベルで行う手法として分類した．この手法を使うことにより，パターン 1 よりも実行時間は遅くなるが SIMD 化可能になる．1,2 のパターンが本研究で SIMD 化可能と判断されるパターンで DSPstone では全体の半分を占めた．3 は本研究では SIMD 化不可能と判断されているものである．SIMD 化できないと判断される文は，5 章の SIMD 化判定で述べた．そこで述べた中にあるメモリ上で同一演算の命令が連続していない場合は，データの並べ替えを行ってから SIMD 化するアルゴリズム [23] により SIMD 化可能になる可能性がある．このように，パターン 3 に分類されたものにも SIMD 化可能なループも存在するが，SIMD 化効果を予測することが難しく新たな SIMD 化判定方法の開発が必要なため，本研究では SIMD 化不可能と判定することにした．

7.4 今後の課題

7.4.1 SIMD 化判定方法の検討

現在の SIMD 化判定は，1 命令でも SIMD 化が可能ならループ全体の SIMD 化を行っている．しかし，SIMD 化には考察で述べたようなさまざまなオーバーヘッドがある．このようなコストも考え判定方法を確立していく必要がある．

7.4.2 SIMD 化できる文を増やす

本研究で SIMD 化できないと判断される文の中には，switch 文や while 文などがあるがこれらも SIMD 化できるように，アルゴリズムを拡張することが考えられる．これによって，SIMD 化不可能と判断されていたループの一部が SIMD 化可能となり実行時間の改善が期待できる．

7.4.3 命令レベルでのコード変形器の設計

4 章で述べたように，本研究では SIMD 化変形手法をソースコードレベルと命令レベルに分類した．そこで，アセンブリ言語レベルでのコード変形器を作成し，ソ-

スコードレベルでのコード変形器と共に使用することにより、大きな効果を得ることができる。

第8章 関連研究

現在自動で SIMD 命令を生成するようなコンパイラがいくつか設計されている。本手法に関連の深いいくつかの手法と本手法の比較を述べる。

8.1 Bikらの研究

Intel では、ループ展開から Intel のアーキテクチャに特化した SIMD 命令の生成も行えるようになっている [31]。しかし、SIMD 命令の生成に失敗した場合の対処については書かれていない。実際、本論文の実験でも `icc` で SIMD 化オプションを付けてコンパイルを行った結果、実行時間が増加してしまった例が見られた。

8.2 Sreramanらの研究

ソースコードレベルでループ展開や変数の依存関係などの解析を行い、C+SIMD 化されたインラインアセンブラで出力するコンパイラの研究もあった [32]。しかし、ソースコードレベルだけの処理で効率のよいコード生成するのは限界がある。なぜなら、4.2 節で示したようなアセンブリ言語レベルの手法を行うことが困難になるからである。

8.3 鈴木らの研究

7 章で述べたように、本手法を実装した COINS 上にもアセンブリ言語レベルで SIMD 命令の生成を行っている [29]。SIMD 命令と低水準中間表現で SIMD 命令と対応する動作テンプレートとの照合を行い、SIMD 命令を作成している。アセンブリ言語レベルでの変形なので、本研究で行ったループ展開や配列変換は行われぬ。よって、本研究で行ったようなソースコードレベルでの変形手法を行ったコードに対して、SIMD 命令が生成されることが期待できる。しかし、現在の実装 (COINS1.4.2.2) では、ソースコードレベルで変形を行った一部のプログラムしか変形ができていない。また、SIMD 特有の命令である `shuffle` 命令などに対応していない。

第9章 まとめ

本研究では，SIMD 命令を生成するためのコード変形方法の中から，ソースコードレベルに適した手法を選び，コード変形器の設計を行った．また，コード変形を行うことによる実行時間の増加を防ぐために，SIMD 化可能かどうかの判定を行った．そして，COINS 上で実装を行い，マルチメディア向けのベンチマークを使用して評価を行った．本研究でのコード変形を行い，期待されるコード生成が行われると既存のコンパイラ以上の SIMD 命令が生成できることがわかり，SIMD 命令が生成できなかった場合に実行時間が増加することがなくなった．

謝辞

本研究を進めるにあたり多大なるご指導ご鞭撻を頂いた, 東京工業大学 数理・計算科学専攻教授の佐々政孝先生に深く感謝の意を表します.

また, 佐々研究室の皆様にはさまざまな面で助力を頂きました. あらためまして, ここに深くお礼申し上げます.

参考文献

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools 2nd ed.* Addison Wesley, 2006.
- [2] Andrew W. Appel. *Modern Compiler Implementation in Java second edition.* Cambridge University Press, 2002.
- [3] Aycock, J. and Horspool, R. N. Simple Generation of Static Single Assignment Form. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction* , pp. 110-124, London, UK, 2000. Springer-Verlag.
- [4] Brandis, M. M. and Mössenböck, H. Single-pass generation of static single assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, Vol.16, No.6, pp. 1684-1698, 1994.
- [5] Briggs, P., Cooper, K. D., and Torczon, L. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, Vol. 16, No. 3, pp. 428-455, 1994.
- [6] Briggs, P., Cooper, K. D., Harvey, T. J., and Simpson, L. T. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, Vol. 28, No. 8, pp. 859-881, 1998.
- [7] Chaitin, G. J. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pp. 98-101, New York, NY, USA, 1982. ACM Press.
- [8] COINS Project. COINS home page. <http://www.coins-project.org/> .
- [9] COINS Project. 高水準中間表現の概要, 2004. <http://www.coins-project.org/050303/base/HirOutline.pdf>.
- [10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, Vol. 13, No. 4, pp. 41-486, October 1991.
- [11] GNU-Project. GCC homepage. <http://gcc.gnu.org/>.

- [12] George, L. and Appel, A. W. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, Vol. 18, No. 3, pp. 300-324, 1996.
- [13] INTEL. Intel Compilers homepage. <http://www.intel.com/cd/software/products/asmo-na/eng/compilers/index.htm>.
- [14] Park, J. and Moon, S. M. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, Vol 26, No. 4, pp. 735-765, 2004.
- [15] F. Rastello, F. de Ferrière, and C. Guillon. Optimizing Translation Out of SSA Using Renaming Constraints. In *IEEE CGO '04*, pp. 265-276, March 2004.
- [16] 佐々研究室. 静的単一代入形式最適化システム外部仕様書, 2007. <http://www.is.titech.ac.jp/~sassa/coins-www-ssa/japanese/ssa-external-japanese.pdf>.
- [17] 佐々政孝. プログラミング言語処理系. 岩波書店, 1989.
- [18] Sreedhar, V. C. and Gao, G. R. A Linear Time Algorithm for Placing ϕ -nodes. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 62-73, San Francisco, California, 1995.
- [19] Sreedhar, V. C., Ju, R. D.-C., Gillies, D. M., and Santhanam, V. Translating Out of Static Single Assignment Form. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis, Lecture Notes in Computer Science*, Vol. 1694, pp. 194-210, London, UK, 1999. Springer-Verlag.
- [20] The Machine SUIF Group. Machine SUIF homepage. <http://www.eecs.harvard.edu/hube/research/machsuiif.html>.
- [21] Peng Wu , Alexandre E. Eichenberger , Amy Wang, Efficient SIMD Code Generation for Runtime Alignment and Length Conversion, Proceedings of the international symposium on Code generation and optimization, pp.153-164, March 20-23, 2005.
- [22] 三好健文・杉野暢彦：“レジスタスロットを考慮したSIMD向け細粒度自動並列化コンパイラ”，情報処理学会論文誌コンピューティングシステム, Vol.1, No.2, pp.240-249, 2008年8月.
- [23] Gang Ren, Peng Wu, David Padua, Optimizing data permutations for SIMD devices, Proceedings of the Conference on Programming Language Design and Implementation, pp.118-138, June 2006.
- [24] R.A., van Engelen and K. A., Gallivan : An Efficient Algorithm for Pointer-to-Array Access Conversion for Compiling and Optimizing DSP Applications, Proc.

of Innovative Architecture for Future Generation High-Performance Processors and Systems, pp.80-89, 2001.

- [25] Weihaw Chuang , Brad Calder , Jeanne Ferrante, Phi-Predication for lightweight if-conversion, Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, pp.179-190, March 23-26, 2003, San Francisco, California.
- [26] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.
- [27] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In Conference on Programming Language Design and Implementation, pp. 145-156, June 2000.
- [28] V. Zivojnovic , J. Martinez , C. Schlager and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. Proc. of ICSPAT'94 - Dallas, pp. 715-720, Oct. 1994.
- [29] 鈴木 貢, 藤波 順久: "COINS における SIMD 並列化 ", コンピュータソフトウェア, Vol.25, No.1, pp.65-81, 2008 年 1 月.
- [30] 鈴木貢, 小川大介, 室田 樹, 渡邊坦: "SIMD ベンチマークの設計と実装 ", 情報処理学会論文誌: コンピューティングシステム, Vol.46, No.SIG 16 (ACS 12), pp.95-107, 2005 年 12 月.
- [31] Aart J. C. Bik , Milind Girkar , Paul M. Grey , Xinmin Tian, Automatic Intra-Register Vectorization for the Intel Architecture, International Journal of Parallel Programming, Vol.30 No.2, pp.65-98, April 2002.
- [32] N. Sreeramam , R. Govindarajan, A Vectorizing Compiler for Multimedia Extensions, International Journal of Parallel Programming, Vol.28 No.4, pp.363-400, August 2000.
- [33] インテル株式会社:Linux* 版インテル (R) C++ コンパイラ ユーザーズ・ガイド, 2004.