マルチメディアSIMD命令活用のためのデータサイズ推論

鈴木 貢[†1] 藤波順久[†2] 福岡岳穂[†3]
渡邊坦[†1] 中田育男[†4]

最近のほとんどのプロセッサの命令セットには，信号処理や画像処理の高速化を狙った（マルチメディア処理向け）SIMD 命令が付加されている．これらを活用するためには，プログラムの実行頻度が高い部分で SIMD 命令を活用するようにアセンブリ言語や intrinsic ルーチン等で書き下しているのが現状であるが，可搬性や保守性等の観点からはコンパイラによる自動生成が望ましい．C 言語等で記述されたプログラムから SIMD 命令で処理可能な部分を発見し，SIMD 命令を適用していく際に障害となるものの 1 つに，汎整数拡張（integral promotion）という規約がある．コンパイラがこれを素直に遵守してコードを生成しようとすると，アセンブラ言語等による書き下しの場合に比べて，並列実行性が低下したり，あるいは SIMD 命令の適用自体を諦めなければならないことになったりする場合が多い．本論文では，汎整数拡張を遵守する場合と同じ計算結果を保証しながら，並列実行性の向上や SIMD 命令の適用を可能にする解析方式として，データサイズ推論を利用する方式を提案する．

Data Size Inference for Multimedia SIMD Instructions

Mitsugu Suzuki,[†1] Nobuhisa Fujinami,[†2] Takeaki Fukuoka,[†3]
Tan Watanabe[†1] and Ikuo Nakata[†4]

Most of recent processors have been equipped with multimedia SIMD instruction set which is intended to accelerate speeds of media processing programs. Although programmers are using assembly languages or intrinsic routines on coding hot spots of the programs to exploit the SIMD instructions, compilers should generate SIMD codes automatically for portability and maintainability of the programs. “Integral promotion” rule in a high level language often becomes an obstacle for discovering code fragments which can be translated into appropriate SIMD instructions. If compilers obey the rule strictly, they cannot translate the fragments into the SIMD instructions as the programmers does in assembly languages, etc. In this paper, we present a code analysis method based on “data size inference”. It increases translatability and parallelism in using the SIMD instructions, while it guarantees the same result as the integral promotion rule is strictly obeyed.

1. はじめに

1996 年に汎用プロセッサである IA32 命令セットアーキテクチャへ拡張の形で MMX と呼ばれるマルチメディア処理向け SIMD（Single Instruction Multiple Data stream）命令セットが発表された頃から，プロセッサのほとんどが同様な機能を備えるようになった．本論文では，この種の命令セットを「SIMD 命令セット」と呼ぶことにする．これらの命令セットは，2 から 16 並列程度の SIMD 型処理命令の体裁をとっている[1)]．既存の汎用プロセッサの基本命令セットに SIMD 命令セットを付加することによって，マルチメディアアプリケーションの高速化を図るというアプローチは，今後も一般的な設計手法として定着していくものと思われる．

その結果，音声圧縮やグラフィック処理プログラムの多くはもちろんのこと，記号処理プログラム[2)]でさえも，それらの命令を活用してプログラミングされるようになった．ところが，まだコンパイラによる SIMD 命令セット活用の支援は，文献 3)〜5) の例のように，本論文で提案する解析を用いる必要がない単純

†1 電気通信大学情報工学科
Department of Computer Science, The University of Electro-Communications
†2 ソニー株式会社
Sony Corporation
†3 株式会社管理工学研究所
Kanrikogaku Kenkyusho, Ltd
†4 法政大学情報科学部コンピュータ科学科
Faculty of Computer and Information Sciences, Hosei University

な加減算のような場合や，解析を必要とする場合でもコンパイラ内にあらかじめ用意された柔軟性のないテンプレートへのマッチングによるといった限定的なものにとどまっている．現状ではSIMD命令セットを活用するには，アセンブリ言語やintrinsicルーチン☆を用いたプログラミングが必須である．コードの移植性や保守性等を考えると，コンパイラによるSIMD命令の活用は，言語処理系の研究における重要なテーマの1つといえる．

これらSIMD命令セットに共通する特性として，2章で述べるように演算データのサイズをなるべく小さく設定するほど処理の並列度が向上するという事項がある．ところがC言語等では，汎整数拡張（integral promotion）と呼ばれる自動的なデータサイズの昇格が規定されている．プログラマはこの規約を念頭においてプログラムを作成する一方で，画像処理ならR，G，Bにそれぞれ8ビットずつ，音声処理なら16ビットといった具合に処理対象に応じて最適な処理データサイズを選択する．ところが一般的には，特別な仕様を言語処理系に導入するか，本論文で提案する方式を組み込まない限り処理系はこの規約を素直に遵守しなければならない．すると，2章で説明するように，処理の並列度の低下や，命令へのマッチング不適合が生じ，SIMD命令を最大限に活用できなくなる．

本論文では，式のとりうる値と演算結果の有効なビットの集合を推定することによって，処理に最適なデータサイズを解析し，汎整数拡張で昇格するintegral typeよりも狭い演算サイズでも汎整数拡張を行う場合と同じ結果を得る手法を提案する．

以下，C言語のソースプログラムと，それが中間言語に翻訳された形のものを使って議論を行う．中間言語の形式には，「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」[6]（以下COINSと略）で開発された低水準中間表現LIR（Low level Intermediate Representation）[7]を拡張した形式を用いる．LIRを構成するL式（Lノード）の意味を本論文の説明に必要な部分だけ抜粋したものを**表1**に示す．本論文では整数型のみを取り扱う．LIRの整数型は固定ビット数で，型として符号つき/符号なしの区別はなく，その区別は演算子で行う．そしてLノードのそれぞれの出力には型が付けられている．また，符号つき整数では2の補数表示を仮定している．以降の議論では，主にL式の図式表現を用いる．

☆ 高級言語で使える，機械語命令を使った組み込みルーチン．

表1 各Lノードの意味
Table 1 Semantics for each L-node.

Lノード	ノードの意味，出力する値
$(INTCONST : t\ \ x)$	x 整数定数
$(FRAME : t\ \ s)$	ラベル s で示されるフレーム中のデータメモリのアドレス
$(NEG : t\ \ x)$	$-x$ 符号反転
$(ADD : t\ \ x\ \ y)$	$x + y$ 加算
$(SUB : t\ \ x\ \ y)$	$x - y$ 減算
$(MUL : t\ \ x\ \ y)$	$x \times y$ 乗算
$(CONVSX : t\ \ x)$	符号拡張
$(CONVZX : t\ \ x)$	ゼロ拡張
$(CONVIT : t\ \ x)$	精度の低い整数への降格
$(BAND : t\ \ x\ \ y)$	x と y のビットごとの論理積
$(BOR : t\ \ x\ \ y)$	x と y のビットごとの論理和
$(BXOR : t\ \ x\ \ y)$	x と y のビットごとの排他的論理和
$(BNOT : t\ \ x)$	x のビットごとの論理否定
$(LSH : t\ \ x\ \ y)$	x を y ビット左シフト
$(RSHS : t\ \ x\ \ y)$	x を y ビット符号つき右シフト
$(RSHU : t\ \ x\ \ y)$	x を y ビット符号なし右シフト
$(TSTEQ : t\ \ x\ \ y)$	$x = y$ なら真（-1）を返す．そうでなければ，0を返す
$(TSTLTS : t\ \ x\ \ y)$	$x < y$（符号つき）
$(TSTLTU : t\ \ x\ \ y)$	$x < y$（符号なし）
$(MEM : t\ \ x)$	データメモリのアドレス x にある型tのオブジェクトが格納している値
$(IF : t\ \ c\ \ x\ \ y)$	c を評価した結果が真ならば x，偽ならば y の値を返す（c に接続できるのは比較演算子のみ）．

“: t” は型 t の結果を返すことを意味する．
($t \in \{I128, I64, I32, I16, I8\}$)

2. SIMD命令セットと汎整数拡張

本論文で対象としているSIMD命令セットは，**図1**のようにnビット（n = 64，128等）の長いレジスタを，演算データのサイズmビット（m = 8，16，32）のサブレジスタに分割して，mビット幅の同種の演算n/m回分を並列に行うように設計されている．結果的に，演算データのサイズを小さくまとめることができれば，並列度を高めて実行効率を上げることができる．

通常のプロセッサでは条件分岐になる文脈は，if変換[8]を施して，マスクを生成する比較命令とマスク演算を組み合わせた選択演算に翻訳する．そして，小規模ながらSIMD型処理命令の体裁をとっているので各種ループ変換[9)~11)]のようなSIMD計算機やベクタプロセッサ向けの最適化・並列化技術を応用できる．以下の議論では，SIMD命令向けに中間言語に対してこれらの変換が施された形のものを取り扱い，同型の操作が複数並んだ様子をそのままではなく，1つを取り出して図示することにする．

汎整数拡張とは，処理系が規定しているintegral type（COINSの規定値では32ビット整数，アーキ

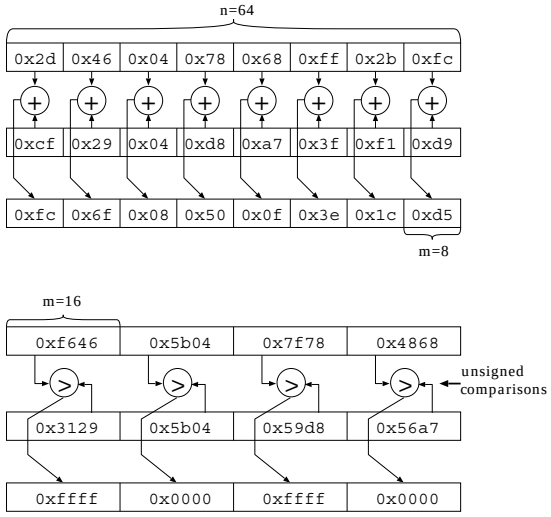

図 1 SIMD 命令の例（加算と比較）
Fig. 1 Two examples for a multi-media instruction set (addition and comparison).

テクチャによっては 64 ビットの場合もある）より小さいサイズの整数型変数を参照する際に，符号拡張やゼロ拡張を施して integral type のサイズにし，それを演算で用いることである．そして，演算結果を小さいサイズの整数型変数に書き出す場合は，結果の上位ビットの値は捨てられ，下位ビットから書き出し先のサイズの分だけ切り出して書き出す．汎整数拡張を行えば，integral type より小さいサイズのデータを演算に用いて，結果を小さいサイズの変数に書き戻す場合でも，多くの場合に演算の途中結果がラップアラウンドせずに済む．

しかし，汎整数拡張を SIMD 命令セットのコード生成で素直に実施すると，実際には SIMD 命令で処理可能なコードであるのに，コンパイラは（integral type のデータサイズを処理する SIMD 命令がなくて）適用不能と判断するか，m = 32 として処理するので変数の参照時につねにサイズ変換命令を挿入することになり，サイズ変換のオーバヘッドや並列実行性能の低下を招く．また，レジスタプレッシャも高まり，レジスタのあふれが起きやすくなる．

ただし，文献 3)〜5) の例のように，式を構成する演算の中に（たとえば右シフトのように）上位ビットの値が結果の下位ビットに影響を与える演算を含まない場合は，代入先のデータサイズと演算のデータサイズを等しくすることができる．また，コンパイラに内蔵された，汎整数拡張を行う場合との互換性が検討済みであるパターンに演算がマッチした場合も，提案する手法を用いる場合と同様にコードを生成できる．しかし，一般にこの種の方法は柔軟性に欠ける．

```
#define AVE(x,y) (((x)>>1)+((y)>>1)+\
                  (((x)|(y))&1)) ‥ (a)
#define AVE(x,y) (((x)+(y)+1)>>1) ‥ (b)
....
short *a, *b, *c;
for (...;i+=8) {
  a[0]=AVE(b[0],c[0]);a[1]=AVE(b[1],c[1]);
  ...
  a[6]=AVE(b[6],c[6]);a[7]=AVE(b[7],c[7]);
  a+=8; b+=8; c+=8;
}
```

図 2 整数配列間の平均値を求める
Fig. 2 Averaging two integer arrays.

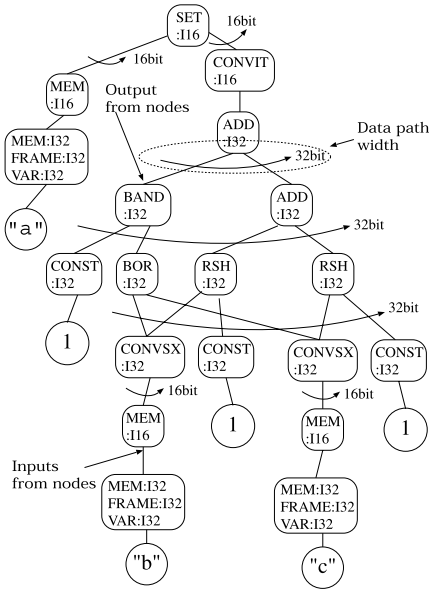

図 3 図 2 の (a) の場合の L 式
Fig. 3 Intermediate representation for case (a) of Fig. 2.

たとえば**図 2** で `AVE` の定義を SIMD 命令向きの (a) とする場合の代入文 `*a=AVE(*b, *c)` に対しては**図 3** のような L 式が生成される．このコードは SIMD 命令向きに処理のすべてが 16 ビットに収まるように工夫されているが，上記のような演算を含むので通常は汎整数拡張が必要となり，32 ビットで処理するコードが生成される．

このように，SIMD 命令を活用して効率的なコードを得ようとしても，汎整数拡張がプログラマの意図の反映を阻害しているといえる．本論文では，integral type よりも狭いデータサイズで演算を行っても，汎整数拡張を素直に行う場合と同じ結果を得るための解析である「データサイズ推論」についての議論を行う．

3. データサイズ推論

データサイズ推論は代入文単位で行い，「上向き解

表 3 上向き解析の推論規則
Table 3 Inference rules for bottom-up analysis.

L ノード	上向き解析の推論規則
$(NEG : t\ \ x)$	$lo = -x.up,\ \ up = -x.lo$
$(ADD : t\ \ x\ \ y)$	x の上下界の差と y の上下界の差が環のサイズより大きければ全域にわたる．そうでなければ，$lo = x.lo + y.lo$, $up = x.up + y.up$.
$(SUB : t\ \ x\ \ y)$	x の上下界の差と y の上下界の差が環のサイズより大きければ全域にわたる．そうでなければ，$lo = x.lo - y.up$, $up = x.up - y.lo$.
$(MUL : t\ \ x\ \ y)$	符号付き整数として扱い，x と y を正と負の領域に分割し，4 通りの場合の値域の合併を求める．
$(CONVSX : t\ \ x)$	符号付きとして $x.lo > x.$up なら，元のビット数の環の全域にわたるとして扱う．そうでなければ，上下界を符号拡張して新しい上下界とする．
$(CONVZX : t\ \ x)$	$x.lo > x.up$ のときは，元のビット数の環の全域にわたるとして扱う．そうでなければ，上下界をゼロ拡張して新しい上下界とする．
$(CONVIT : t\ \ x)$	上下界の差が縮退先のビット数の環のサイズより大きければ，全域にわたるとして扱う．
$(BAND : t\ \ x\ \ y)$, $(BXOR : t\ \ x\ \ y)$, $(BOR : t\ \ x\ \ y)$	x, y の値域が 0 をまたいでいる（$lo > up$）ときは，それぞれで値域を $0..up$ と $lo..2^{size} - 1$ に分割し，文献 16) 4 章 3 節の方法で値域を求め，それらを合併する．
$(BNOT : t\ \ x)$	$lo = \overline{x.up},\ \ up = \overline{x.lo}$（ビットごとの反転）
$(LSH : t\ \ x\ \ y)$, $(RSHS : t\ \ x\ \ y)$, $(RSHU : t\ \ x\ \ y)$	x を y の値域のそれぞれの値でシフトしたときの値域の合併を求める．
$(TSTEQ : t\ \ x\ \ y)$	x と y が定数で同じ値なら $lo = up = 2^{size} - 1$．x と y の値域が疎なら $lo = up = 0$．いずれでもなければ $lo = 2^{size} - 1,\ \ up = 0$.
$(TSTLTS : t\ \ x\ \ y)$, $(TSTLTU : t\ \ x\ \ y)$	それぞれ符号付きとなしで比較して，x の値域が必ず y より小さければ $lo = up = 2^{size} - 1$．x の値域が必ず y 以上なら $lo = up = 0$．いずれでもなければ $lo = 2^{size} - 1,\ \ up = 0$.
$(IF : t\ \ c\ \ x\ \ y)$	図 5 の構造マッチングに外れた場合，c が必ず真（0 以外）なら x の値域，必ず偽（0）なら y の値域．いずれでもなければ，x の値域と y の値域の合併．

表 2 L ノードのデータ構造の拡張
Table 2 Extended data fields for a L-node.

フィールド	意味
`size`	ノードの型（I32 等）に対応するビット幅
`up`	ノードがとりうる値の上界のビットパターン
`lo`	ノードがとりうる値の下界のビットパターン
`lv`	ノードの出力の有効ビット集合

析」と「下向き解析」の 2 つの解析をこの順番で行う．解析のために L ノードを表すデータ構造を拡張し，**表 2** に示すフィールドを付加する．

3.1 上向き解析

まず，右辺値の式を構成している木を深さ優先でたどってゆきながら，各ノードでの演算結果の値域（上界と下界）を求める解析を行う．

ノードのとりうる値のビットパターンを $2^{\texttt{size}}$ を法とする 2 進数と見なし，その値域を**図 4** に示すように `lo` から始まり `up` に終わる連続した値と定義する．図では `lo` が黒丸，`up` が矢印の先に対応し，$2^{\texttt{size}}$ の剰余系の上を時計回りで値が大きくなっていくように表現している．図の例のように，必ずしも `up` $\geq$ `lo`（符号なし比較）の関係ではなく，逆の場合もある．図では，サイズ拡張処理に対する値域の設定や，値域の合併の例も示されている．

値域を $2^{\texttt{size}}$ の剰余系として表現することによって，

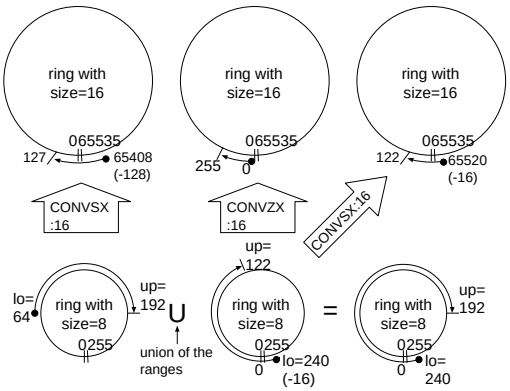

図 4 ノードがとりうる値の表現
Fig. 4 Value range representation for a node.

キャストを施された式の値域の追跡をラップアラウンドやシフトのはみ出しも含めて正確に行うことができる．

演算子によって，可換性や上位ビットと下位ビットの影響関係等が異なり，またノードの各入力が定数かどうかによっても推論される値域が異なる．演算子ごとに定めた解析方法を「上向き解析の推論規則」といい，**表 3** に示す．

さらに IF ノードについては，**図 5** の破線で囲った部分の構造に対するマッチングを行い飽和処理を認識

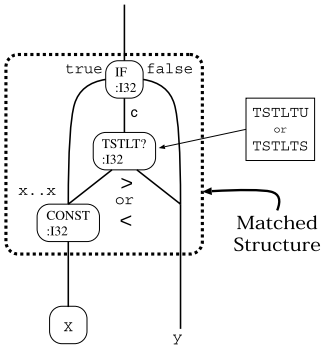

図 5 IF ノードの構造マッチング
Fig. 5 Structural matching of the IF nodes.

表 4 飽和処理の推論規則（符号なしの場合）
Table 4 Inference rules for Saturation operations.

★ $lo \leq_u up$ の場合

比較の方向	$x <_u lo$	$lo \leq_u x \leq_u up$	$up <_u x$
$>$	$lo..up$	$x..up$	x
$<$	x	$lo..x$	$lo..up$

★ $lo >_u up$（値域が 0 をまたぐ）の場合

比較の方向	値域
$>$	$x..2^{size}-1$
$<$	$0..x$

$<_u$, $\leq_u$：符号なし比較
$lo \equiv y.lo$, $up \equiv y.up$
推論した結果として値域が 2 つに分かれる場合は，2 つを覆う最小の値域をノードの値域としてある（ex. $up < x$ のとき $0..up \cup x$ を $0..x$ とする）.

し，**表 4** の推論規則に従って IF ノードの値域を推定する．表では符号なし比較（TSTLTU）の場合だけを示しているが，符号付き比較（TSTLTS）の場合は，表中の比較演算子「$<_u$」「$\leq_u$」等を符号付きのものに置き換えて適用する．符号付きの場合 $lo >_s up$ は値域が 2^{size-1} をまたいでいることを意味する．

変数を参照している MEM ノードでは `size` = 変数のサイズ，`lo` = 0，`up` = $2^{\texttt{size}} - 1$，つまり環の全域とする．この仮定の改良案については，7 章で触れる．

図 3 の L 式に対して，上向きの解析を行った結果を**図 6** に示す．

3.2 下向き解析

次に，代入先のデータサイズで決まる左辺値の有効ビットの集合を元にして，木を下向きにたどってゆきながら，上向き解析で得た値域が張る有効ビット集合（値域の値のすべてを区別して表現できる最小限のビット数だけ 1 を最下位ビットから並べた集合）と，上のノードから与えられた有効ビット集合をつき合わせていく「下向き解析」を行う．値域とそれが張る有効ビット集合の関係を**表 5** に示す．

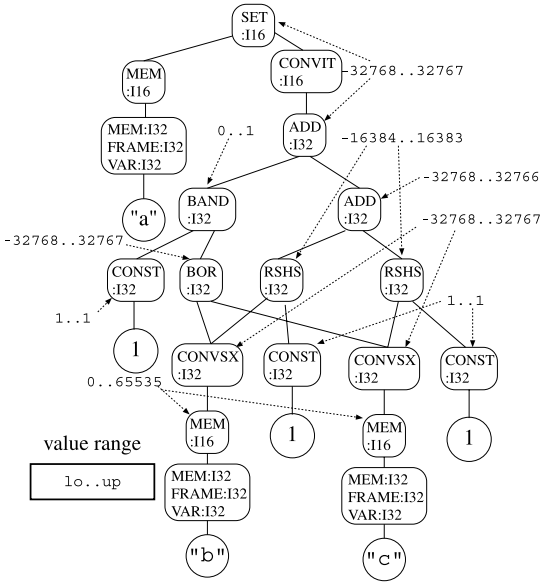

図 6 図 3 に対する上向き解析の結果
Fig. 6 The result of bottom-up analysis for Fig. 3.

表 5 値域が張る有効ビット集合
Table 5 Meaningful bit set for value ranges.

値域	有効ビット集合
$0 \leq lo \leq up < 2^{size-1}$	$cover(up)$
$0 \leq up < lo < 2^{size-1}$	$2^{size}-1$（全域）
$2^{size-1} \leq lo \leq up \leq 2^{size}-1$	$cover(\overline{lo})$
$2^{size-1} \leq up < lo \leq 2^{size}-1$	$2^{size}-1$（全域）
$2^{size-1} \leq lo \leq 2^{size}-1$ かつ $0 \leq up < 2^{size-1}$	$y = cover(\overline{lo}) \mid cover(up)$ として $y \mid y << 1$
$2^{size-1} \leq up \leq 2^{size}-1$ かつ $0 \leq lo < 2^{size-1}$	$2^{size}-1$（全域）

$cover(x)$：x を最上位ビットからみて最初の 1 のビットから下をすべて 1 にした値（ex. $cover(\text{0x0000ff00}) = \text{0x0000ffff}$）
$\overline{x}$：x のビットごとの反転
その他演算子は C 言語の記法に従う.

上のノードから与えられる有効ビット集合」とは，ノードの出力の中で上のノードが必要としているビットの集合，言い換えれば上のノードの演算結果に影響を及ぼす可能性のあるビットの集合を意味する．下のノードに与える有効ビット集合は，**表 6** の推論規則を用いて決定する．たとえば，図 3 の SET ノードでは，`0x0000ffff` が左辺値の有効ビット集合となる．しかし，有効ビット集合は，必ずしも最下位ビットから始まるわけではない．たとえば右シフト演算子のノードが下のノードに有効ビット集合を渡す場合，集合はシフト量の分だけ左に移動したものになる．このように，

表 6 下向き解析の推論規則
Table 6 Inference rules for top-down analysis.

L ノード	下向き解析の推論規則
$(NEG : t\ \ x)$	$cover(w)$ を x に伝える．
$(ADD\ :\ t\ x\ y)$, $(SUB : t\ \ x\ \ y)$	x も y も定数でなければ $cover(w)$ を x と y に伝える．どちらかが定数であるときは**注 1** のとおり．
$(MUL : t\ \ x\ \ y)$	x, y のどちらも定数でないときは $cover(w)$ を x と y に伝える．x, y の片方（たとえば x）が 2 のべき乗の定数なら w を x で割った値を y に伝える．そうでなければ**注 2** のとおり．
$(CONVSX : t\ \ x)$	x の最上位（符号）ビットの位置を i とする．w の i より上のビットに 1 がなければ，w の i より下のビット．そうでなければ，w の i より下のビットの i ビットを 1 にした値を x に伝える．
$(CONVZX : t\ \ x)$	x の最上位ビットの位置を i として，w の i より下のビットを x に伝える．
$(BAND : t\ \ x\ \ y)$	x, y の片方（たとえば x）が定数であるときは $w \wedge x$ を y に，そうでなければ w を x と y に伝える．
$(BXOR : t\ \ x\ \ y)$	w を x と y に伝える．
$(BOR : t\ \ x\ \ y)$	x, y の片方（たとえば x）が定数であるときは $w \wedge \overline{x}$ を y に，そうでなければ w を x と y に伝える．
$(BNOT : t\ \ x)$	w を x に伝える．
$(LSH : t\ \ x\ \ y)$	y が定数のときは w を y ビット右にシフトした値を，そうでなければ $cover(w)$ を x に，全ビットを y に伝える．
$(RSHS : t\ \ x\ \ y)$	y が定数のときは w を y ビット左に符号残しシフトした値を，そうでなければ $over(w)$ を x に，全ビットを y に伝える．
$(RSHU : t\ \ x\ \ y)$	y が定数なら w を y ビット左にシフトした値を，そうでなければ $over(w)$ を x に伝える．
$(TSTcc : t\ \ x\ \ y)$	全ビットを x と y に伝える（$cc \in EQ, LTS, LTU$）．
$(IF : t\ \ c\ \ x\ \ y)$	w を x と y に，全ビットを c に伝える．

w：上からきた有効ビット集合（図 8 の下向き解析のパラメータ）
他の単項演算子は w をそのまま x に伝える．
$over(w)$：w の最下位ビットからみて最初の 1 にビットより上をすべて 1 にした値（ex. $over$(0x0000ff00) = 0xffffff00）
その他の記号は表 5 に同じ．

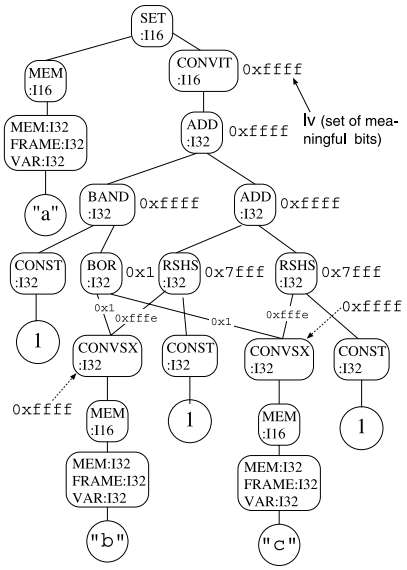

図 7 図 6 に対する下向き解析の結果
Fig. 7 The result of top-down analysis for Fig. 6.

上から必要とされているビットは「最下位から何ビット」という形でなく，ビット集合の形で伝える．

図 6 で AVE の定義を (a) とするの場合に対応する L 式に対して，下向きの解析を行った結果を**図 7** に示す．

3.3 アルゴリズム

データサイズ推論の概略を**図 8** に示す．図 3 の

```
代入式のサイズ推論 (top);
 top: SET ノード;
{
  値域 = 上向き解析 (右辺値のノード);
  下向き解析 (左辺値の有効ビット集合，右辺値のノード);
  top.lv = 左辺値の有効ビット集合;
}

上向き解析 (n): 値域;
 n: ノード;
{
  n. 値域 =
    上向き解析の推論規則 (
      上向き解析 (n. 入力 1),
      上向き解析 (n. 入力 2),
      ....)
  return n. 値域;
}

下向き解析 (w, n)
 w: 上からの有効ビット集合;
 n: ノード;
{
  foreach (i in n. 入力) {
    下向き解析 (
      下向き解析の推論規則を使って他の入力の値域と w から求めた有効ビット集合,
      n. 入力 i)
  }
  /* n.lv の初期値は 0 */
  n.lv = n.lv | (w & n の値域が張る有効ビット);
}
```

図 8 データサイズ推論のアルゴリズム
Fig. 8 Data-size inference algorithm.

CONVSX ノードのように，1 つのノードの出力が複数のノードの入力として共有される場合に，パスによって渡される有効ビットの集合の値が異なる場合がある．出力を共有するために，ノードの最終的な有効ビットの集合の値を，それらの合併集合にする（図 8 の最後の文の処理）．

3.4 推論結果とコード生成

SIMD 命令コード生成の詳細には触れないが，推論結果をどう用いるかを簡単に述べる．

推論の結果として，最適処理データサイズが SIMD 命令の処理データサイズで正規化しても広くなったり狭くなったりする場合がある．この場合コード生成には，(1) サイズが異なる箇所ごとにサイズ拡張/縮退命令を挿入するする方針と，(2) 最初から最大のサイズにサイズ拡張を行い，最後に代入先のサイズに合わせてサイズ変換を行う方針があげられる（あるいは可能なサイズのすべての組合せを試して，実行時間が最短になるものを探すことも考えられる）．いずれの方針をとるにせよ，式を構成する L ノードの lv の値によって，ノード間のデータ幅が決定される．

通常のコード生成では汎整数拡張後のデータの演算のみを考えればよいので，演算子の組合せと（シフト等の）量に注目してマッチングによって命令を生成する．一方データサイズ推論の結果を用いるコード生成では，それに加えてノードの入出力のデータサイズと演算命令の処理データサイズとのマッチングも必要になる．そこで，同じ演算機能を持つテンプレートに対して，処理データサイズを表現する有効ビット集合のフィールドを付加する．そして，ターゲットの命令の処理データ幅のバリエーションに応じて，同じ演算機能に対していくつかのテンプレートを用意する．

L 式とテンプレートが演算機能でマッチングすると，処理データサイズが小さいものから（あるいは別の処理効率の基準で付けられた優先順位で）順にノードの入出力の有効ビット集合のマッチングを行っていく．有効ビット集合のマッチングの成功は，ノード間の有効ビット集合がテンプレートの有効ビット集合で覆われていること，つまり前者の有効ビット 1 であって後者の有効ビットが 0 であるものがないことである．ただし，ビットごとの演算（BAND，BOR，BNOT，BXOR）については処理データ幅のマッチングは不要である．

図 2 で AVE の定義を (b) とする場合の L 式にデータサイズ推論を行った結果に対して，テンプレートをマッチングする様子を**図 9** に示す．具体的な命令のニモニックは，IA32/MMX 命令セットの標記に従っている．

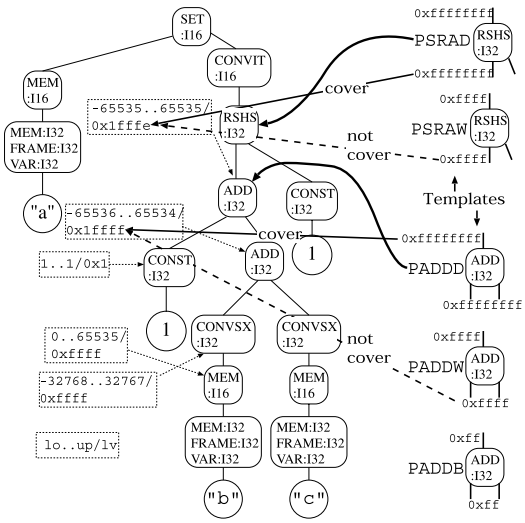

図 9 図 2 の (b) の場合の推論結果への有効ビット集合マッチング
Fig. 9 Meaningful bit set matching for an inferred result of Fig. 2 (b).

表 7 データサイズ推論の実装
Table 7 Implementation of the data-size inference.

モジュール名	行数	機能
IntBound	606	L ノードの値域を表す．各 L ノードに対応したメソッドを提供する．
IntLive	446	L ノードの出力として有効なビットを表す．各 L ノードに対応したメソッドを提供する．
IntConst	525	上記 2 つの計算で使われる基本データ型と演算を提供する．

4. 実　　装

上向きの解析の推論規則と下向きの解析の推論規則を，それぞれ値域と有効ビットの集合のクラスとメソッドとして，COINS の記述言語である Java で実装した．その内訳を**表 7** に示す．

5. データサイズ推論の効果

簡単なプログラムから COINS コンパイラで生成した L 式を元にして，人手で IA32 の MMX や SSE2 命令セットのコードを生成を行い，実行時間を計測した．使用した機材は Celeron 2.0 GHz，SiS-655 チップセット，512 MB Dual-channel DDR-SDRAM メモリ，OS が Windows2000 の構成のもの（以下 P4 と略）と，Pentium-M 1.4 GHz，512 MB DDR-SDRAM メモリ，OS が WindowsXP の構成のもの（以下 PM と略）で，言語処理系は cygwin の C コンパイラ（gcc 3.3.1）やアセンブラを使用した．

```
        movq ones,%mm0      # 0x0001000100010001
loop:
        movq (%edi),%mm1    # %edi = *b
        movq (%esi),%mm2    # %esi = *c
        movq %mm2,%mm3
        por  %mm1,%mm3
        pand %mm0,%mm3
        psraw $1,%mm1
        psraw $1,%mm2
        paddw %mm1,%mm3
        paddw %mm2,%mm3
        movq %mm3,(%eax)    # %eax = *a
```

図 10 図 7 に対して生成されるコード
Fig. 10 Generated code for Fig. 7.

```
        movq ones,%mm0        # 0x0000000100000001
loop:
        punpcklwd (%edi),%mm1 # lo half
        punpcklwd (%esi),%mm2 # CONVSX in LIR

        psrad $16,%mm1
        psrad $16,%mm2
        paddd %mm1,%mm2
        paddd %mm0,%mm2
        psrad $1,%mm2         # lo result
        punpckhwd (%edi),%mm4 # higher half
        punpckhwd (%esi),%mm3 # CONVSX in LIR
        psrad $16,%mm4
        psrad $16,%mm3
        paddd %mm4,%mm3
        paddd %mm0,%mm3
        psrad $1,%mm3         # higher result
        packssdw %mm3,%mm2    # saturated packing
        movq %mm2,(%eax)      # write out the result
```

図 11 図 9 に対して生成されるコード
Fig. 11 Generated code for Fig. 9.

5.1 平均値を求めるプログラム

図 2 の (a) の場合に対応する図 7 の解析結果から生成したコードを**図 10** に，図 2 の (b) の場合に対応する図 9 の解析結果から生成したコードを**図 11** に示す．

まず，上向き解析の結果によるコード生成の改善例を説明する．図 9 の解析結果によると 32 ビットから 16 ビットへの縮退が必要となるが，図 11 では符号付き飽和（符号付き整数の −32768 以下の値を −32768 に，32767 以上の値を 32767 にする）縮退命令 packssdw を用いている．ここでは飽和なし縮退を使うべきだが，命令セットに用意されていない．しかし上向き解析の結果，飽和の影響を受けない値域であることが分かるので，packssdw が飽和なしと同じ動作をすることが保証される．このように，解析結果を用いて命令セットの不足を補うことも可能である．

次に，処理時間で比較を行う．図 10 や図 11 のコード，`AVE` の定義を (a) として素直に汎整数拡張して生成したコード，それに gcc でコンパイルしたコードの処理時間を，**表 8** に示す．

表 8 配列間の平均を求めるプログラムの処理時間
Table 8 Processing time for averaging two arrays.

機種/ AVE		MMX		SSE2	
	gcc	DSI	IP	DSI	IP
P4/(a)	4.37	3.67	4.27	3.68	3.94
P4/(b)	3.80	—	3.90	—	3.82
PM/(a)	4.53	1.11	3.09	1.49	2.98
PM/(b)	1.98	—	2.10	—	2.43

10 億回繰り返したときの要素あたりの平均処理時間（時間の単位は nS）
AVE：図 2 の AVE の定義
gcc：gcc の最適化-O6 で生成したコード
MMX：MMX 命令セットで 64 ビット処理
SSE2：SSE2 命令セットで 128 ビット処理
DSI：データサイズ推論の結果から最適なオペランドサイズを選択したコード
IP：素直に汎整数拡張を実施したコード

表の各欄の (a) と (b) を比較すると，通常命令による処理の場合や，SIMD 命令を使っても汎整数拡張を行う場合は AVE の定義を (b) とした通常の平均の求め方の方が実行効率が高い．しかし，データサイズ推論を行って不要な汎整数拡張を除去可能ならば，AVE の定義を SIMD 命令向きの (a) とした SIMD 命令向きの求め方が実行効率が最も高い．そして，(b) 行の gcc 欄や IP 欄に対する改善比は，SIMD 命令の実行ユニットが改良されている PM の方が高い．このように SIMD 向きのコーディングから最大の実行効率を得るには，提案する方式で最適な処理データサイズを調べることが不可欠であると思われる．

表の (a) 行の DSI 欄と IP 欄を比較すると，汎整数拡張や処理の並列度の半減によるのオーバヘッドは P4 の場合で 6 から 16%であるが，改良型の PM の場合は 100 から 178%にもなっている．これから，機種によっては不要な汎整数拡張が SIMD 命令を使ったコードの実行効率を非常に悪化させることがあると分かる．

5.2 動画圧縮プログラムからの例題

MPEG-4 Video CODEC の 1 つの実装である XviD（1.0.0 版）中の関数 `interpolate8x8_halfpel_hv()` の実行時間を計測した．この関数は，8×8 の画素集合を受け取り，2×2 ドットの輝度の平均を各点について求めて，同じサイズの画素集合に書き出す．平均は，画素集合の要素の 4 つの符号なし 8 ビットの数と丸めをコントロールする定数の和を求めて，その値を右に 2 ビットシフトして求める．データサイズ推論

表 9 XviD の補間関数の実行時間
Table 9 Processing time for a interpolating function of XviD.

		MMX	
機種	gcc	DSI	IP
P4	342	158	372
PM	344	153	355

1,000 万回繰り返したときの 8 × 8 の画素集合あたりの平均処理時間（単位は nS）
各項目の意味は表 8 に同じ.

を用いると 16 ビットに拡張して演算を行えばよいことが分かるが，通常は汎整数拡張のために 32 ビットに拡張してから演算するコードになる．この実行結果を**表 9** に示す．ここでは画素集合のサイズの都合で，SSE2 命令を使った 128 ビット処理の実験は行っていない．

DSI 欄と IP 欄を比較すると汎整数拡張と並列度の半減のためのオーバヘッドが 130%以上になっている．不要な汎整数拡張を行うと，最適な演算データサイズ（16 ビット）での処理の場合に比べて 2 倍以上の時間がかかり，しかも gcc 欄との比較からスカラ処理よりも遅くなり，SIMD 並列化した意味を失っている．

6. 関連研究と文献

Larsen ら[4] の論文では，プログラムからの SIMD 命令向けの並列性抽出について議論しているが，最適処理データサイズの解析については言及していない．Sreraman らの論文[12] では，さらに Intel MMX 命令セットにおける処理データサイズについて議論しているが，最適処理データサイズの解析については言及していない．Leupers らの論文[3),13] では，演算パターンと SIMD 命令とのマッチングについて論じているが，本論文で議論している演算の最適サイズの解析や選択には言及していない．Bik らの論文[5] で紹介されているコンパイル例の中には，演算に右シフトを含むような最適サイズの解析を必要とする例が紹介されているが，同じサイズのデータどうしの限られた演算の場合に限られている．Fisher らの論文[14] の中で，演算の最適サイズの調査が必要であることは指摘しているが，その具体的な解決法については言及していない．Stephenson らの論文[15] では，本論文の解析法と同様の戦略で，ハードウェア設計における演算器の最適サイズの解析を提案し，SIMD 命令のコード生成に応用可能であるとしているが，解析の推論規則の詳細には言及しておらず，また値域の表現方式が本論文で提案している方式で size が ∞ である場合だけであるので，キャストがかけられた式の値域を正確に追跡することができない．また，SIMD 命令のコード生成への具体的な応用方法についても言及していない．

表 3 や表 6 を求めるために文献 16) の 2，3，4 章を参考にした．

7. ま と め

言語拡張を行わないで，効率の良いマルチメディア向け SIMD 命令を生成するためのデータサイズ推論の方式を提案し，試験的なコード生成を行い，方式の有効性を確認した．

本論文では代入式単位での解析に焦点を絞り，コンパイラが式をまとめあげることを仮定した．しかし文献 15) で提案されているように，フロー解析や，ループ解析を応用して，変数の代入間をまたがる値域の推論を行えば，3.1 節 で参照された変数の値域はサイズで決まる環の全域としたのに比べて，精度の高い解析が可能となる．

しかし，処理系が未完成なため，提案方式の評価が少数の例題と人手によるコード生成での比較にとどまっている．上記の解析との連携や，コード生成も含めた提案方式の処理系への組み込み，そして多くの実際の例題による評価は今後の課題である．

謝辞 数々の有益なご指摘をいただいた本論文の査読者に心から感謝する．なお，本研究は文部科学省科学技術振興調整費「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」の補助を受けている．

参 考 文 献

1) 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).
2) 田中哲朗：マルチメディア命令セットを使った文字列処理，情報処理学会研究会資料 97-PRO-14, pp.127–132 (1997).
3) Leupers, R. and Bashford, S.: Graph-Based Code Selection Techniques for Embedded Processors, *ACM Trans. Design Automation of Electronic Systems*, Vol.5, No.4, pp.794–814 (2000).
4) Larsen, S. and Amarasinghe, S.: Exploiting Superword Level Parallelism, *ACM SIG-PLAN Notices*, Vol.35, No.5, pp.145–156 (2000).
5) Bik, A.J.C., Girkar, M., Grey, P. and Tian, X.: Automatic Intra-Register Vectorization for the Intel®Architecture, *International Journal of Parallel Programming*, Vol.30, No.2, pp.65–98 (2002).
6) 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.coins-project.org

7) COINS プロジェクト LIR 仕様書 (2002). http://www.coins-project.org/spec/lir.pdf
8) Allen, J., Kennedy, K., Porterfield, C. and Warren, J.: Conversion of control dependence to data dependence, *Proc. SIGPLAN Conference on Program Language Design and Implementation* (*PLDI*), pp.177–189 (1983).
9) Allen, R. and Johnson, S.: Compiling C for Vectorization, Parallelization, and Inline Expansion, *Proc. SIGPLAN Conference on Program Language Design and Implementarion* (*PLDI*), pp.241–249 (1988).
10) Bacon, D., Graham, S. and Sharp, O.: Compiler Transformations for High-Performance Computing, *ACM Computing Serveys*, Vol.26, No.4, pp.345–420 (1994).
11) Franke, B. and O'Boyle, M.: An Empirical Evaluation of High Level Transformations for Embedded Processors, *Proc.International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp.59–66 (2001).
12) Sreraman, N. and Govindarajan, R.: A Vectorizing Compiler for Multimedia Extensions, *International Journal of Parallel Programming*, Vol.28, No.4, pp.363–400 (2000).
13) Leupers, R.: Code Selection for Media Processors with SIMD Instructions, *Design, Automation and Test in Europe* (*DATE'00*), pp.4–8 (2000).
14) Fisher, R. and Dietz, H.: Compiling for SIMD Within a Register, *LCPC'98*, LNCS 1656, pp.290–304 (1998).
15) Stephenson, M., Babb, J. and Amarasinghe, S.: Bitwidth Analysis with Application to Silicon Compilation, *Proc. SIGPLAN Conference on Program Language Design and Implementation* (*PLDI*), pp.108–120 (2000).
16) Warren Jr., H.: *Hacker's Delight*, Addison-Wesley (2003).

付録　表 6 への注

☆注 1

加算では，定数（たとえば x）の最下位からみて i ビットの 0 が連続しているとき，加算結果の最下位から i ビットの並びには y の最下位から i ビットの並びの値がそのまま反映する．w の最上位から見て最初の 1 のビットの位置を j とすると，加算結果の i ビット目から j ビット目の間の値は，y の i ビット目から j ビット目の間のビットの値で値で決まる．したがって，y に伝える有効ビット集合は，w の 0 ビット目から i − 1 ビット目の値と，i から j ビット目を 1 にした値の合併となる．

例：

```
01000010 w(上から来た有効ビット集合)
00101000 x(定数)
01111010 y に下げるべき有効ビット集合
```

減算では，x が定数なら x のビットごとの反転を，y が定数なら $-y$ を用いて加算の推論規則を用いる．

☆注 2

定数（たとえば x）の最下位からみて i ビットの 0 が連続していて（最下位からみてビット i ではじめて 1），次の 1 が i+j ビット目だとする．すると演算結果のビット 0 からビット i − 1 はつねに 0 になり，ビット i からビット i + j − 1 までは y の最下位のビット 0 からビット j − 1 までの値がそのまま出力される．ビット i + j + k (k ≥ 0) では，y のビット 0 から k までとビット j から j + k までの影響を受ける．よって y に伝える有効ビット集合は，K を w の最上位からみて最初の 1 の位置とすれば，ビット j から j + K までを 1 にした値と，ビット 0 から K までを 1 にした値と，w を右に i ビットシフトした値の合併になる．

例：

```
   hgfedcba
×  01010010
------------
    fedcba
   dcba
   ba
------------
   ****cba0
    |||
    ||a,d の影響を受ける
    |a,b,d,e の影響を受ける
    a,b,c,d,e,f の影響を受ける（3 番目の 1 の影響は 2 番目の 1 の影響に含まれる）
```

(平成 15 年 7 月 14 日受付)
(平成 16 年 2 月 18 日採録)

鈴木　貢（正会員）

電気通信大学情報工学科助手．記憶管理アルゴリズム，並列/分散アルゴリズム，言語処理系等に興味を持つ．平成 14 年度本学会論文賞受賞．ACM，電子情報通信学会，日本ソフトウェア科学会各会員．

藤波 順久

東京大学大学院理学系研究科修了．（株）ソニーコンピュータサイエンス研究所を経て，ソニー（株）社員．言語処理系，プログラミング環境，アセンブリ言語等に興味を持つ．日本ソフトウェア科学会会員．

福岡 岳穂（正会員）

電気通信大学大学院卒業．株式会社管理工学研究所．コンパイラ，並列/分散アルゴリズム，計算機アーキテクチャ等に興味を持つ．日本ソフトウェア科学会会員．

渡邊 坦（正会員）

電気通信大学情報工学科教授．言語処理系，プログラミング言語等に興味を持つ．ACM，IEEE Software，日本ソフトウェア科学会各会員．

中田 育男（正会員）

法政大学情報科学部コンピュータ科学科教授．プログラム言語，言語処理系等に興味を持つ．情報処理学会名誉会員，ACM，IEEE，電子情報通信学会，日本ソフトウェア科学会各会員．