COINS を用いた大域的データサイズ推論の実現

Realization of Glocal Data Size Inference Utilizing COINS

太田 眞敬 †
Masanori OTA

滝本 宗宏 ††
Munehiro TAKIMOTO

† 東京工業大学大学院情報理工学研究科
Dept. of Mathematical and Computing Sciences,
Tokyo Institute of Technology
ota.m.ac@m.titech.ac.jp

†† 東京理科大学理工学部情報科学科
Faculty of Science and Technology,
Tokyo University of Science
mune@cs.is.noda.tus.ac.jp

最近のほとんどのプロセッサの命令セットには，画像処理などの高速化を狙った SIMD 命令が付加されている．そして現状では SIMD 命令の活用は intrinsic 命令の活用やアセンブリ言語を人手で入力して行っている．しかし汎用性や可搬性，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．今回は自動生成の過程で弊害となる C 言語の汎整数拡張を SIMD 向けに拡張したデータサイズ推論をさらに拡張した大域的データサイズ推論を提案する．

1 はじめに

最近のほとんどのプロセッサの命令セットには画像処理や音声処理の高速化を狙った SIMD 命令が付加されている．現在これらを活用するためには，プログラムをアセンブリ言語の SIMD 命令を用いたり，intrinsic 命令を人手で書くことで高速化を図っている．しかし汎用性や保守性等の観点から，コンパイラによる自動生成が望ましい．そのため現在さまざまな方法でコンパイラによる自動生成が模索されているが，このときに SIMD 命令活用の弊害となる仕様が存在する．

汎整数拡張 (integral promotion) と呼ばれるこの仕様の詳細は後述するが，これは C 言語等において計算の安全性を保障するために存在する規約だが，これにより SIMD 命令の活用が非常に難しくなっている．コンパイラがこの規約を遵守し，コードを生成するとアセンブリ言語により書き下した場合に比べて，並列実行度が低下する場合が少なくない．そのため COINS ではそれを回避する方法として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法としてデータサイズ推論が実装されている．本稿はデータサイズ推論 [2] をさらに拡張した大域的データサイズ推論を提案する．

なお COINS[1] とは，コンパイラ研究の基盤となる共通のコンパイラの作成を目的に開発された，並列化コンパイラ向け共通インストラクチャである．

2 SIMD

現在発表されているプロセッサは，音声や画像といったマルチメディア処理の高速化を実現する手段として SIMD 命令を備えるのが一般的になった．これは一般的に SIMD 命令や SIMD 命令セットと呼ばれる．この傾向は Intel 社の IA32(x86) 系や IBM の PowerPC 系といった汎用プロセッサにとどまらず，ARM 系といった組み込み系のプロセッサにまで及んでいる．さらに最近では家庭用ゲーム機の PlayStation2 の Emotion Engine や Playstation3 の Cell Broadband Engine と呼ばれる CPU に対しても SIMD 命令が組み込まれている．

SIMD とは Single Instruction Multiple Data の省略で呼んで字のごとく 1 回の命令で複数のデータを扱うことができる命令形態である．

SIMD 化を行った簡単な例を説明する．

```
int i;
char a[100];
for(i = 0; i < 100; i++) {
  a[i] = i;
}
```

これは for 文を用いた簡単な例である．SIMD 化を行うにあたり，まず繰り返しの展開を行う．今回の例では char 型の計算を 2 つ同時に行えると仮定する．

```
int i;
char a[100];
```

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

上記の展開したソースプログラムに対して SIMD 命令を適用する．

```
int i;
char a[100];
for(i = 0; i < 100; i+=2) {
  {a[i + 0], a[i + 1]} = {i + 0, i + 1};
}
```

「{ }」は 1 つの命令で実行できることを意味する．これは 2 行を単純に 1 行にまとめただけに見えるが，実際には 2 つのデータを 1 つの命令で実行しているのである．そのため最初のプログラムでは 100 回繰り返し計算を行っていたが，SIMD 命令を用いることで 50 回に減らすことができる．このように SIMD 命令を適用することで，命令数を削減することができる．

このとき重要になるのが，並列度である．並列度とは 1 度の計算で何個のデータを扱えるかを表した数値である．たとえば今回の例だと並列度は 2 となる．当然だが，並列度が上がれば上がるほど計算効率はあがるため，実行速度は上昇する．並列度を上昇させる方法は，1 つのデータの値を小さくすることである．SIMD 命令の並列実行は 1 つのレジスタを複数個に分割し，それぞれにデータを入れることで行われる．そのため 1 つのデータが小さくなればなるほど並列度が上がることになる．しかし汎整数拡張のためにデータを小さくするのが困難という現状があるため，以下ではまず汎整数拡張について説明する．

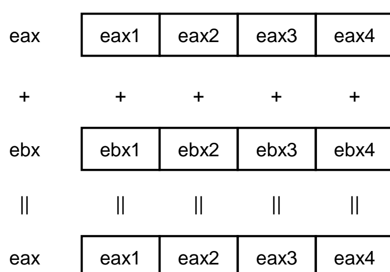

図 1: レジスタを分割し複数データを扱う

3 汎整数拡張

汎整数拡張についての詳細を説明する．そもそも汎整数拡張とは C 言語および C++言語で使用されている仕様である．これは int 型以下の大きさを持つ整数型の変数は，その値が評価される時点で int 型に拡張されるというものである．この変換は明示的には行われず，暗黙のうちに変換が行われる．

例えば int が 4 バイト，short が 2 バイト，char が 1 バイトで表現されている処理系だと仮定する．このとき，$-2147483648 \leq \text{int} \leq 2147483647$，$-32768 \leq \text{short} \leq 32767$，$-128 \leq \text{char} \leq 127$，のようになる．これを踏まえて以下のようなソースコードを考える．

```
short s = 32767;
char  c = 127;

s++;
c++;

printf("s = %d, c = %d", s, c);
```

汎整数拡張が行われない場合，s,c ともにインクリメントを行うと，表現域を超えるためラップアラウンドが起こり「s = -32768，c = -128」と表示される．しかし汎整数拡張が行われる場合には格上げと呼ばれる処理が行われ，s,c ともに一時的に値が int 型と同じ大きさを持つため内部的に s は 32768，c は 128 となる．そして printf で表示される際には格下げと呼ばれる処理が行われ,「s = 32767，c = 127」と表示される．

このように汎整数拡張によって，プログラマが意図していないであろうラップアラウンドを防ぐことができる．しかし内部的に int に変換されるということは，計算を行ううえでは 4 バイトのメモリおよびレジスタが必要ということになる．これはデータの大きさを極力小さくし並列度を上げるという SIMD 命令を活用する上での弊害となる．かといってラップアラウンドは望ましくないため，極力計算の安全性を保障する必要がある．そこで次の章では計算の安全性を保障しつつデータの大きさをより小さくするデータサイズ推論について説明する．

4 データサイズ推論

前述のようにデータサイズ推論は計算の安全性を保障しつつよりより小さい大きさで計算を行うため

の処理である．データサイズ推論 [2] は代入文単位で行われ，

- 上向き解析
- 下向き解析

の 2 種類の解析からなる．また解析は COINS[1] の LIR 表現を用いて行う．

4.1 上向き解析

上向き解析は以下の手順で行われる．

1. 代入文において右辺値を LIR 表現の木構造で表現する
2. 木を深さ優先探索でたどり，上向き推論規則に従い各ノードのとり得る最大値 (up_0) と最小値 (lo_0) を決定する
3. up_0 と lo_0 を符号なし表現とみなし，上界を up, 下界を lo とする (必ずしも $lo \leq up$ とは限らない)
4. lo と up から値域の張る有効ビット集合 (表 1) を求める．

上向き推論規則とは各ノードについてどのような上界および下界が考えられるかを与えた規則の集合である．

有効ビット集合とは値域の値をすべて区別して表現できる最小限のビットに最下位ビットから 1 を並べた集合である．

$cover(x)$ とは x を最上位ビットから見て最初の 1 より下のビットをすべて 1 にした値である．
例えば

```
  ADD
  / \
 x   y
```

のような木構造が存在していた場合に，ADD ノードの up と lo は

- x の範囲もしくは y の範囲が環のサイズより大きければ ADD の範囲は全域にわたる
- そうでなければ，lo =x.lo+y.lo, up =x.up+y.up

と定められる．前者は被演算子 (例では x と y) のどちらかもしくは両方の範囲が全域にわたっている場合，後者では正しく推論されないための処理である．

値域 (符号なし表現)	有効ビット集合
$0 \leq lo \leq up < 2^{size} - 1$	$cover(up)$
$0 \leq up < lo < 2^{size} - 1$	$2^{size} - 1$
$2^{size-1} \leq lo \leq up \leq 2^{size} - 1$	$cover(\bar{lo})$
$2^{size-1} \leq up \leq lo \leq 2^{size} - 1$	$2^{size} - 1$
$2^{size-1} \leq lo \leq 2^{size} - 1$ かつ $0 \leq up < 2^{size-1}$	$y = cover(\bar{lo})$ として $y \| y << 1$
$2^{size-1} \leq up \leq 2^{size} - 1$ かつ $0 \leq lo < 2^{size-1}$	$2^{size} - 1$

表 1: 値域の張る有効ビット集合

このとき MEM ノード即ち変数が出現した際，代入文単位で解析が行われるデータサイズ推論ではその変数がどのように定義されているかかを判断できない．そのためその変数に関して最大値と最小値を決定することはできない．そういった場合はその変数がとり得る範囲全域をカバーするように最大値と最小値を設定する．例えば 2 バイトの short 型の変数が出現した際には下界が 0，上界が 65535 となる．本論文ではこの推論についての改善を 5 章で示す．

4.2 下向き解析

下向き解析は木を下に辿って行きながら，上向き解析で得た自分自身のノードの有効ビット集合 (表 1) と上のノードから与えられる有効ビット集合をつきあわせていくことで解析を行う．この時つきあわせ作業は下向き推論規則によって行われる．下向き推論規則とはノードの種類によって使用するビット数がどのように変化するかを記述した規則の集合である．

5 大域的データサイズ推論

データサイズ推論は代入文単位で行われた．そのため LIR においてメモリ (MEM) やレジスタ (REG) ノードが出現場合にデータサイズを予想することが難しくなる．これは 4.1 でも述べたように代入文単位で 3 階席が行われる場合，変数がどのように定義されているかが判断できないためである．よって上向き解析では MEM ノードや REG ノードが出現した際には，その変数もしくはレジスタの範囲を全域とした．しかしこの方法は効率をさげる可能性がある．これを解消するために我々が提案する方法では代入

文単位でデータサイズ推論を行うのではなく，プログラム全体でデータサイズ推論を行う手法を提案する．大域的データサイズ推論は以下の 2 つの処理に分類できる．

- 登録
  SET ノード (代入演算子) が出現した際に左辺値の変数名と右辺値のデータサイズ推論の結果を組の形で保存
- 取得
  MEM ノード，REG ノードが出現した際に変数名を取り出し，登録されているのであれば登録された値を使用．登録されていないのであれば，変数が示し範囲全域を範囲とする．

5.1 登録

登録作業は SET ノードに対して行う．このときデータサイズ推論では右辺値のみを処理の対象としていたが，ここでは左辺値の変数名と右辺値の結果の組として保存する．しかしその変数名がすでに登録されている可能性が考えられる．これは今回の手法ではプログラム全体を処理の対象としているため，ローカル変数名で同じ識別子が使われている可能性があるためである．以下にその例を示す．

```
void func1(void){
  int hoge;
  ...
}
void func2(void){
  int hoge;
  ...
}
```

このようにすでに同じ変数名で登録されている場合は，

1. すでに登録されている右辺値の結果を取り出す．
2. 現在登録しようとしている結果と 1 の右辺値の結果の合算を考える．
3. 2 の結果を新しい結果として登録しなおす．

このような手順で保守的な近似により登録し直すことにする．

具体的な合算の方法は以下の通りである．データサイズ推論で使われる右辺値の結果は { 最小値，最大値，サイズ } の 3 つの要素からなる．ちなみに前述の

	min	max	size
old	0	255	16
pre	123	456789	32
new	0	456789	32

表 2: 合算処理の例

とおり，これらの要素は非負整数である．以下ではすでに登録されている右辺値の結果の要素を { min_{old} , max_{old}, $size_{old}$} とする．また現在登録しようとした右辺値の結果の要素を { min_{pre}, max_{pre}, $size_{pre}$} とする．さらにそれらを合算した結果を { min_{new} , max_{new}, $size_{new}$} とする．合算処理は

$$\begin{aligned} min_{new} &= min(min_{old}, min_{pre}) \\ max_{new} &= max(max_{old}, max_{pre}) \\ size_{new} &= max(size_{old}, size_{pre}) \end{aligned}$$

となる．$min(a, b)$ は a と b を比較し，小さい方を返す関数，$max(a, b)$ はその逆である．簡単な例を表 2 に示す．

5.2 取得

この作業は前章のデータサイズ推論と基本的に同じアルゴリズムで処理する．代入文単位では MEM や REG ノードが出現した場合，そのノードの範囲は変数がとりうる値の範囲全域であるが，大域的データサイズ推論の場合は変数名が登録されているか否かで処理が変わる．登録されている場合は，それと組になっている右辺値の範囲を範囲として用いる．右辺値が存在しない場合は代入文単位のデータサイズ推論と同じで，全体を範囲としてデータサイズ推論を行う．

6 実験

実験は SIMD 向きに改良を加えた平均値を求めるプログラムと XviD の interpolate 関数を用いて行った．

SIMD 向きの平均値を求めるプログラムとは演算途中でデータサイズが極力大きくならないようにした平均値を求めるプログラムである．今以下のような 2 つの整数値の平均値を求めるマクロを考える

$$AVE1(x, y)\ (((x) + (y) + 1) >> 1)$$

$$AVE2(x, y)\ (((x) >> 1)+((y) >> 1)+(((x)|(y))\&1))$$

$AVE1$，$AVE2$ は同値である．それを証明するためにまず $AVE1$ から説明する．一般的な平均値を求める式は $(x+y)/2$ で表される．しかしこの場合 $x+y$ の値が奇数だと .5 という小数点が出現する．これを整数値で表すことを考える場合四捨五入を行うのが一般的である．それに対して計算機では整数値の計算において小数点以下は切り捨てられる．この 2 つの差異を埋めるために，予め 1 を加えている．さらに 2 で割ることは右シフトを 1 回行うことと同値であるため，シフト演算に置き換えている．
次に $AVE2$ についてである．これは $(x+y+1) >> 1$ を展開しさらに変形させた形である．展開することで $(x >> 1) + (y >> 1)$ の項が出現することは容易にわかる．$(x|y)\&1$ に関しては補足が必要である．この部分が $AVE1$ と同じように小数点部分の繰り上げ処理をおこなっている．しかし $AVE2$ の場合は x，y について先にシフト演算を行っているため，それぞれの値について小数点以下が考慮されていない．それぞれのシフト演算で小数点が発生し繰り上げが必要とされる場合は x と y の値が (奇数，奇数)，(奇数，偶数)，(偶数，奇数) の 3 パターンである．これら 3 パターンのときに繰り上げ即ち 1 を加える作業が必要となる．これは x か y どちらかが奇数，言い換えると 2 進数で表した場合最下位ビットが 1 となっているときである．これを論理演算で表すと $(x|y)\&1$ となる．

$AVE1$ は一般的な平均値を求める方法であるが，この場合 x と y を加算しているため，データが大きくなる可能性がある．それに対して $AVE2$ は x と y それぞれを先に右シフトしているため，データが大きくなる可能性がない．今回の実験では下の $AVE2$ を用いた．また入力は 16 ビットの数値を用いた．

XviD の interpolate 関数とはオープンソースで開発されている映像フォーマット XviD の関数の 1 つである．関数の内容は 8 ドット×8 ドットから光度の平均を求めるものである．入力については 8 ビットの数値である．

7 結果と評価

結果は図 2 のようになった．

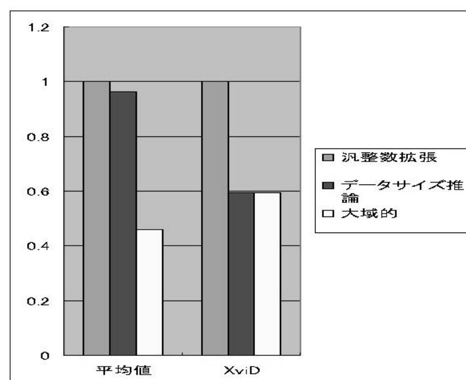

図 2: 実験結果

縦軸は汎整数拡張を行った場合の実行時間を 1 としたときの相対的な値である．

平均値だが，これは変数の値の範囲を絞ることで，データサイズ推論に比べかなりよい結果が出ている事がわかる．それに対して interpolate 関数についてはデータサイズ推論と同じ結果となっている．代入文同士が互いに依存していない場合にはこのような結果になるが，最低限データサイズ推論と同じ速度が保障されることわかる．以上により本手法はデータサイズ推論と同じもしくはそれ以上の効果を発揮できていることが確認できた．
データサイズ推論は変数の範囲を全域として考えて処理を行った．それに対して本手法では範囲を定めることでより効率のよい SIMD 化を目指した．そのときデータサイズの範囲をうまく小さくできなかったとしても，データサイズ推論と同じ結果になることは明らかである．

8 まとめ

C 言語と C++言語には汎整数拡張と呼ばれる規約が存在する．この規約によってラップアラウンドが起こりにくくなるため，プログラムの安全性は高まるが，SIMD 命令を活用したプログラムを考える際には並列度を下げる要因となってしまう．そこで SIMD 向けに汎整数拡張を改良したデータサイズ推論とよばれる技術を導入することで COINS では並列度を高めている．しかしこの技術は代入文単位で行われるため，変数が出現した際の処理が非常に非効率的なものになってしまっている．それを解消するためにプログラム全体でデータサイズ推論を行う

大域的データサイズ推論を導入した．これによりプログラムの種類によってはデータサイズ推論より処理速度が速い目的コードを作成できることが確認できた．ただし今回はデータフロー解析など，変数の使用や定義を考慮していないため，ローカル変数で同じ変数名が出現した際には，合併処理を行った．この部分を改善することでより効率のよい大域的データサイズ推論が可能となる．

謝辞

多忙な中，本稿を精査していただいた東工大 佐々氏に感謝します．

参考文献

[1] COINS Project. Coins project home page. http://www.coins-project.org/

[2] 鈴木 貢，藤波 順久，福岡 岳穂，渡邊 坦，中田 育男 マルチメディア SIMD 命令活用のためのデータサイズ推論　情報処理学会論文誌 プログラミング Vol. 45, No. SIG5 (PRO21) pp. 1-11.