

CESP におけるレジスタ割り当て

久川 敏典

佐藤 良治

1992 年 4 月 28 日

概要

第5世代コンピュータプロジェクトの核言語である ESP¹ は PSI という専用のマシンで実装された。CESP² は ESP を一般の計算機上で動作するようにしたものである。

CESP は最初暫定版として WAM³ コードのエミュレータとして実装されたが、処理時間を速くする為に処理を行なう部分を機械語にコンパイルして直接実行することにした。

CESP の機械語版を作成するにあたり実行時間数で使用した大域的なレジスタ割り当てについて述べる。このオブティマイズによって処理時間を 20% 程度速くすることができた。

%程度速くする

この機械語版は木多直性
を考慮したMCQ方式を用い
ている。

CESP (Common ESP) is prolog based object-oriented language. CESP uses machine description of the target machine to increase portability. Specifically we use global register allocation in run-time-routine. -

佐藤 旧所属 (株) AI 言語研究所
新所属 (株) 日本DEC
久川 旧所属 J I P D E C
新所属 (株) 日本科学技術研修所

CESP におけるレジスタ割り当て

久川 敏典

佐藤 良治

1992 年 4 月 28 日

概要

第5世代コンピュータプロジェクトの核言語である ESP¹ は PSI という専用のマシンで実装された。CESP² は ESP を一般の計算機上で動作するようにしたものである。

CESP は最初暫定版として WAM³ コードのエミュレータとして実装されたが、処理時間を速くするために処理を行なう部分を機械語にコンパイルして直接実行することにした。

CESP の機械語版を作成するにあたり実行時間数で使った大域的なレジスタ割り当てについて述べる。この最適化によって処理時間を 20% 程度速くすることができた。

1 基本事項

1.1 コンパイラの構成

CESP のコードを動作させる為に 2 つの方法がある。

1. CESP のプログラムを WAM コードに変換し、WAM コードのエミュレータで動かす。(エミュレータ版)
2. CESP のプログラムを対象となるマシンの機械語にまで変換して直接実行する。(機械語版)

以下 CESP は機械語版の方を指すことにする。
CESP のコンパイラの構造は次のようになっている。

1. CESP のソースを WAM に変換するフロントエンド。
→ CESP のコードそれ自体で書かれている。
2. WAM コードから LIR コードに変換するミドルエンド。
→ C 言語で書かれています。機種による依存性がない。
3. LIR コードからターゲットとなる機械語に変換するバックエンド
→ C 言語で書かれています。機種による依存性があるが、その部分は後で述べる MD ファイルに吸収させている。

1.2 LIR コード

機械語の生成は内部表現とアセンブラとのマッチング処理により行なわれる。そのマッチング処理は移植担当者が記述する MD ファイルを元にして自動生成される。WAM コードを直接、機械語に変換する方法をとるとは移植担当者に多大な負担をかけてしまう。

これは WAM コードそれ自体が機械語で置き代えるには難しい処理を行なっている為である。

そこで、WAM よりも低レベルで機械語と同レベルの中間言語 LIR を作った。WAM から LIR に変換することによってパターンマッチの数を減らせて、移植担当者の負担を減らすことができた。

コンパイラの側から見ても、LIR はストリーム形式の内部表現になっていて木構造に比べて楽にマッチング処理を行える。

1.3 MD ファイル

MD ファイルはバックエンド部の移植の時に必要な機種依存部の記述ファイルである。これを CGG⁴ に通すことでバックエンド部の C のソースコードが得られる。

MD ファイルの中には、

- レジスタ構成、
- LIR コードとマシン語の対応、

¹Extended Self-contained Prolog

²Common ESP

³Warren Abstract Machine

⁴Code Generator Generator

%程度速くする

この機械語版は木構造性を考慮したMCQ方式を用いている。

- アセンブラの表記
- オブジェクト生成、オブジェクトファイルのリンクを行なう関数
- etc

に関する記述を含んでいる。

レジスタの情報にはレジスタ名、静的に割り当てる変数名や、レジスタのクラス、使用可能なレジスタの情報がある。

1.4 実行時間関数

CESP は WAM コードを経て LIR、LIR を経てマシン語に変換するが、ステップを経るごとにコードの量が増える。コードの増大を抑える為に実行時間関数を導入した。

CESP の本体も CESP 自身で書かれている為、ここを小さくしないと最終的にできる CESP も大きくなってしまふ。

実行時間関数は、LIR コードまで用意されていてインストール時に GEN_RTR という実行時間関数を自動生成するコマンドを使って作られる。

2 バックエンド

2.1 バックエンドの動作

バックエンドは

- MD ファイルから作られた LIR コードのマッチング部分
- 変数に対応するレジスタを管理するレジスタマネージャーの部分
- 関数呼び出し処理の部分
- コードを出力する部分
- etc

に分けられる。関数呼び出し処理はマシン、OS により異なるので移植担当者にコーディングしてもらわなければならない。

2.2 レジスタ割り当て以外のオプティマイズ

レジスタ割り当て以外のオプティマイズとして以下のものがある。

- アセンブラ特殊命令の使用
- グローバルレジスタ
- 実行時間関数のパラメタのレジスタ渡し
- レジスタの固定割り付け
- スケジューリング

3 レジスタ割り当て

レジスタ割り当てには全体で使われている基本ブロック単位のもの、実行時間関数でのみ使われている大域的な方法がある。

3.1 基本ブロックについて

基本ブロックはプログラムの制御が直線的に実行される部分である。つまり、分岐やラベル定義が無いステートメントの集合をいう。

3.2 基本ブロック内のレジスタ割り当て

変数のレジスタ割り当ては基本ブロック単位で行なっている。勿論、大域的なレジスタ割当を行なったものよりはコードの質が落ちるが、何よりもプログラムの制御の解析を行なう必要がない。プログラムの解析を行なう場合、パスが一つ増えてしまいコンパイル時間が遅くなる。CESP はあくまでもインタープリタであり、コンパイルする時間もユーザーから見て待つ時間の対象になる為、遅くするわけにはいかない。

CESP ではアセンブラの出力部分は MD⁵ファイルに記述されている。

```
<MD ファイル例>
stm_offset_ld
    Oprd~OnReg(G)?:{ForceReg(G,LD)}
    const
    Oprd~OnReg(G)?:{ForceReg(G,NEW)}
    { Emit("mov [$2.reg+$3],$4.reg"); }
```

この MD ファイルの例では

```
T_1 := reg_topH[ 4 ]
```

という LIR コードを

```
mov    %hi(_reg_topH), %g1
ld      [%g1+%lo(_reg_topH)], %g1
mov     [%g1 + 4], %g2
```

というアセンブラに変換する。

OnReg() はオペランドがレジスタに割り当てられているかどうかをチェックするマクロで、ここでは、レジスタに割り当てられていなければ何もしないが、そうでなければメモリ上からレジスタにロード処理を行なう。ロード処理を行なうのが ForceReg() マクロである。アセンブラ上の最初の 2 行が対応している。

このようにして変数に割り振られたレジスタは基本ブロックの最後まで変数の情報を保持しようと努める。レジスタマネージャはなるべく、変数が割り当てられていないレジスタを作らないようにする。

基本ブロックの境界にきたら、レジスタの内容がそこにくるまでに代入を受けていた場合のみレジスタ内容のメモリへのセーブを行なう。

基本ブロックの境界に来るまでに使用可能なレジスタが無くなった場合、レジスタに割り振られた情報で一番プライオリティが低いものを解除する。この解除する情報が代入を受けていたのなら、レジスタ内容のメモリ転記を行なったあと解除する。

また、レジスタがどの変数に割り当たっているかという情報は条件付き goto 文以外ではクリアします。条件付き goto 文では次のブロックでも同じ情報を使用しても構わないため情報を残しておきます。

3.3 大域レジスタ割り当ての必要性

勿論、基本ブロック単位のレジスタ割り当てには限界がある。基本ブロックの境界が現れた時、レジスタに載っている変数が多ければ多いほどメモリ転記の命令が増えることになる。

プログラムの流れを解析するだけでも、このメモリの転記量を抑えることができる。それでもメモリ転記が生じるような変数はレジスタに割り当てることによってメモリ転記を抑えられる。

大域的なレジスタ割り当ては実行時間数を作成する時のみ行なった。

クラスをコンパイルする時点でプログラムの解析を行なうのは前述したように時間がかかるせいもあるが、何より間接ジャンプが多い為解析をしにくいからである。

3.4 プログラムの流れの解析

変数が定義されてから参照がなくなるまでの間を Live-Range と言い、変数の寿命を表わす。Live-Range は変数一つにつき複数存在する。

```
例 1      foo(p, q)
           int p, q;
           {
               int r;
               r = p % q;                <s1>
               while (r != 0) {           <s2>
                   p = q;                 <s3>
                   q = r;                 <s4>
                   r = p % q;             <s5>
               }
               return p;                 <s6>
           }
```

この場合、変数 *r* の Live-Range は *s1* または *s5* から始まって *s2, s3* と続き *s4* で終わります。

変数の Live-Range はプログラムの流れに沿って存在するので、まずプログラムの流れを解析しなければならぬ。

LIR は制御構造として

⁵Machine Description

goto Lab	無条件ジャンプ
if cond goto Lab.	条件ジャンプ
jump *reg	間接ジャンプ
call func	関数呼び出し

ここで、間接ジャンプ、関数呼び出しは制御が実行時関数の外へ制御が移るので無視する。

LIR のストリーム内には基本ブロックのマーキング処理が施されていて、それに着目してブロック単位に制御の流れを判断する。この時、基本ブロックごとの変数の参照／定義も調べる。

3.5 基本ブロック単位の Live-Range 分割

実行時関数上でインプリメントする時に、Live-Range を基本ブロック単位で考えた。

Live-Range は変数の定義から始まって、参照がある限り続く。基本ブロック単位の Live-Range であってもこのことは変わらない。

基本ブロック単位の Live-Range の定義、参照は、そのブロック内で変数が最初に現れた時の状況で決定する。そのブロックで最初に出現した時に参照であった場合、以降同じブロック内で何回定義があろうとも、そのブロックでは参照として考えられる。

例 2 前回の例で考える。

まず各ステートメントは次のブロックに分割される。

```

b1 = {s1}
b2 = {s2}
b3 = {s3,s4,s5}
b4 = {s6}

```

この基本ブロックに対して考えると、変数 r の Live-Range は $b1$ から始まり $b2, b3, b2$ と続く。

基本ブロック単位の Live-Range は、普通のステートメント単位のそれと比較して次のような特徴を持つ。

1. ステートメント単位の Live-Range と比べ目が荒くなる。
本来干渉することがない変数が干渉することがある。
コンパイラ時間が短くなる。
2. 同じ基本ブロック内で最初に参照、次に定義というパターンが生じた合、本来はここで分割されるべき Live-Range が一つの Live-Range になる。
3. 本来の Live-Range が基本ブロック内で完結した場合、それを認識することができない。ここでは、基本ブロック単位のレジスタマネージャーに任せた。

Live-Range は各基本ブロックの変数の入力と出力を求めることによって決まる。
入力／出力と定義／参照の関係は次の通り。これをデータフロー方程式という。

$$in[B] = ref[B] \cup (out[B] - def[B])$$

$$out[B] = \bigcup_{S \in Succ[B]} in[S]$$

基本ブロック内で生きている変数の存在を次のように求められる。

$$live[B] = \bigcup_{S \in Succ[B]} ((ref[S] \cup live[S]) - def[S])$$

live[B] → 現在のブロックの Live-Range
live[S] → 一つ後のブロックの Live-Range
ref[S] → 一つ後のアクセススタート
def[S] → 一つ後のデファインスタート

ただし、ここで $ref[S] \cap def[S] = \phi$ 。

この式は「次のブロックで Live-Range が存在する変数のうち、次のブロックで Live-Range が始まらないものの和集合」という意味を持つ。

Live-Range は次のようにして求められる。

アルゴリズム

```

while1 live[] が安定していない do
  foreach B ∈ 各基本ブロック do
    foreach P ∈ B の先行基本ブロック do
      live[P] =  $\bigcup ((ref[B] \cup live[B]) - def[B])$ 
    do
  od
od

```

これで変数単位の Live-Range が求まる。この、Live-Range は繋がっているとは限らないので、繋がっていない Live-Range を複数に分割する。以下、分割した Live-Range に対してレジスタを割り当てる。

基本ブロック単位にした理由は、Live-Range による変数のレジスタ割当てで全て処理してしまうと、レジスタ割り当てに漏れた変数の処理の効率が悪くなってしまうと考えた為である。レジスタマネージャーはレジスタの個数や使用規約が異なる CPU を同じアルゴリズムでレジスタ割り当てを行なわなければならない。

レジスタ数がいくらか多くとも全ての変数に割り当たる分だけあるとは考えられない。どうしても、レジスタが割り当てられない変数がでてくるはずである。

もし、使用できるだけのレジスタを変数に割り当てた場合、割り当てられ損なった変数へのアクセスがロード / ストアの繰り返しになり効率が悪化することが考えられる。

このような状態を緩和する為にレジスタに割り当てられ損ねた変数は前述の基本ブロック単位で行なうレジスタマネージャーに任せることにした。

4 Live-Range を使用した最適化 (レジスタ割り当て以外)

Live-Range が算出することでレジスタ割り当て以外にも以下のような最適化を行なうことができる。

基本ブロック単位のレジスタマネージャーは、レジスタに割り当てられた変数でない限り、その基本ブロック内で代入が生じた変数を基本ブロックの最後でレジスタの内容をメモリにセーブする処理を行なう。しかし、Live-Range が分かっている変数は、変数が継続のブロック内使用されているかどうか分かる為、セーブしなくても良い変数を見分けられる。

基本ブロックの最後でセーブ処理を行なう時に、次の基本ブロックの Live-Range の状況さえ分かればセーブすべき変数かどうかを判断できる。

以下式で示されるのがセーブ処理が必要な変数の集合である。

$$\bigcup_{S \in Succ[B]} lr[S]$$

if 文等でできる基本ブロックの境界では、ある条件の元で変数内容のセーブ処理を次のブロックに持ち越すことができる。その条件というのは

- if 文のように分岐先が複数で、ブロックの一つが現在のコードの直後にあること。
- 直後のコードには Live-Range が繋がっているが、他のジャンプ先のコードには Live-Range が繋がっていないこと

の二つである。

下の例でいうと、最適化を行なわない場合 `if1` の部分で `T_3` のセーブ処理が必要になる。

しかし、Live-Range が既に分かっている `L_xxxx` のラベルで始まる基本ブロックで `T_3` が使用されていないのならば `if1` でセーブする必要がなくなる。

次の `if2` の箇所でも同じことがいえる。

```

T_3 := T_1 & 3
    <1>
if T_3 == 0 goto L_xxxx
    <2>
if T_3 != 1 goto L_yyyy

```

4.1 Live-Range 単位のレジスタ割り当て

レジスタマネージャーで分かっていることは使用できるレジスタの個数と、レジスタのクラスだけである。これだけの情報を元にしてレジスタを変数に割り当てる。

本来ならば最低限のレジスタを残しておいて、後のレジスタを全て割り当て用に残すという方法をとるという方法考えられますが、この場合、最低限のレジスタ数がわからない。その為、この方法は難しい。

そこで、前述の基本ブロック単位のレジスタ割当てと併用することにした。

処理の流れは次のようになる。

1. 基本ブロック単位で Live-Range を解析する。(前述)
2. 仮想的にレジスタマネージャーを動かして基本ブロック単位で使用しているレジスタを調べる。

3. レジスタの使用可能な範囲と Live-Range の範囲を比較して、割り当て可能な Live-Range に対してレジスタ割当を行なう。

CESP のバックエンドを MD ファイルマッピングルールから作るのと同じようにレジスタの使用状況を調べるプログラムも自動生成する。

変数にレジスタを割り当てる時に使用可能レジスタの情報から、その Live-Range 全体で使用可能なレジスタを見つけ出す。この時、使用可能なレジスタが制限を受ける為カラーリングの手法は使えなくなる。(変数の Live-Range とレジスタの使用可能な範囲を一緒にカラーリングの手法でやれば旨くいくとおもったのですが) ここで言う、使用可能なレジスタの判断とは、Live-Range が存在する各基本ブロックで使用可能なレジスタの集合を言う。

Live-Range で使用可能なレジスタが分かったら、Live-Range のプライオリティが高い順にレジスタを割り当てる。ここで、レジスタを Live-Range に干渉する Live-Range の使用可能なレジスタの候補から削除する。これを使用可能レジスタが尽きるか、割り当てる変数が尽きるまで行なう。

普通、Live-Range を利用したレジスタ割り当てではグラフの彩色問題を応用したカラーリング処理を行ないますが、ここでは行なっていない。カラーリング処理を行なわなかったのは、レジスタの使用可能な範囲が限定されている為です。

4.2 実行時間関数引数の特殊処理

実行時間関数は引数の受渡し方法をレジスタ渡しで行なっている。引数の受渡しに使用されるレジスタはバックエンドを生成した時点で決められる。パラメータは関数内部でも頻繁にアクセスする為、レジスタに割り当てるのが望ましい。

引数にレジスタを割り当てる場合、引数の受渡しに使用したレジスタと同じレジスタを割り当てるのが一番望ましい。レジスタ割当をしても違うレジスタに割当たってしまうならば、レジスタ間の代入処理にコストがかかってしまう。

引数も一応ローカルな変数と考えられるので、前述のローカルな変数に対するレジスタ割当を行なう。ただし、レジスタを割り当てる時に、なるべく引数の受渡しに使用しているレジスタを使用するようにする。

もし、引数の受渡し用のレジスタが既に他の Live-Range で使用しているのであり、その Live-Range を他のレジスタに割り当てることが可能であれば、そのレジスタを他のレジスタに再割当をおこなう。そこで、空いたレジスタに引数のレジスタを割り当てる。

4.3 グローバル変数のレジスタ割り当て

グローバルに定義された変数の Live-Range は実行時ルーチンに収まらないことがある。Live-Range は反復解法によって後退方向に求められる。そのため、グローバル変数の Live-Range は定義している部分が見つからず実行時関数の先頭に至ってしまうことがある。また、グローバル変数の Live-Range は最後の参照によって終るものではない。関数を出た後でも使用される可能性がある。

このような時、注意しなければならない点が二つある。

- 一つは Live-Range の先頭が関数の先頭を越していく場合。これは Live-Range の先頭が参照で始まったことを意味する。この時、Live-Range を先頭から手繰っていき最初に参照する箇所までの Live-Range を刈り込む。レジスタの割り当てを行なうことが決まったならば、参照で始まる Live-Range の先頭で割り当てられたレジスタへロード処理を行なう。
- もう一つは Live-Range の中で代入が生じた場合。普通の Live-Range では Live-Range が終了することで変数は使用されないということが保証されるが、グローバルな変数では関数外で参照される可能性がある為に保証されない。この場合、レジスタの内容を Live-Range の最後でメモリに転記しなければならない。

5 評価

以下のベンチマークはプログラムコンテキストのプログラムで測定をした。

5.1 Sun 4 RISC の場合

Sparc Station/SunOS4.1 で測定を行なった。

Sun4 に移植された CESP のレジスタ構成は以下の通り。

1 関数内で使用可能なレジスタ総数	2 6
静的に変数に割り当てられているレジスタ数	1 3
割り当て可能なレジスタ数	1 3

測定結果は以下の通り。LIPS 値が約 2 割増し以上になりました。

	オブティマイズ		
	なし (A)	あり (B)	B/A
Rev-30	89956	109430	1.22
Sort-50	32581	39143	1.20
Cons-1000	8727	10055	1.15
Lisp-Tarai3	14359	18902	1.32
Lisp-Fib-10	17155	22290	1.30
Lisp-Rev-30	14527	19090	1.31
8-Queen-1	96959	125802	1.30
Fib-Fact	11338	15364	1.36
Append-100	94886	115774	1.22
Append-100Back	8717	10386	1.19

5.2 Sun3 CISCの場合

Sun3/260 SunOS4.1 で測定を行なった。

Sun4 に移植された CESP のレジスタ構成は以下の通り。

1 関数内で使用可能なレジスタ総数	14
静的に変数に割り当てられているレジスタ数	10
割り当て可能なレジスタ数	4

測定結果は以下の通り。

Append-100 を除いて 1 割増しにもなっていません。レジスタ数の違いが結果に表れています。

	オブティマイズ		
	なし (A)	あり (B)	B/A
Rev-30	45160	48784	1.08
Sort-50	16334	17911	1.10
Cons-1000	4645	4908	1.06
Lisp-Tarai3	7685	7700	1.00
Lisp-Fib-10	9070	9159	1.01
Lisp-Rev-30	7712	7780	1.01
8-Queen-1	51857	53896	1.04
Fib-Fact	6563	6682	1.02
Append-100	50831	60597	1.20
Append-100Back	4781	4904	1.03

6 現在の課題

レジスタに関する情報の不足により、レジスタの割り当てが最適とは言い難くなることがあった。レジスタクラスに依存したコード生成を行わなければならない場合、~~使用される~~レジスタクラスのレジスタに割り当てると、~~ことにより~~、レジスタ間の代入命令が頻繁に発生することがあった。レジスタクラスの考慮を行なった割り当てが CISC 系の CPU では必要になるであろう。

今後

7 おわりに

CESP 上のレジスタマネージャーでインプリメントされた各種の最適化技法を述べた。AI 言語研究所では CESP を Sun3/Sun4 の 2 機種でサポートしている。この 2 機種の実行時間関数がここで述べた一つのレジスタマネージャーによってできていて、双方共に変数のレジスタ割り当てによってメモリ転搬の量が少ないコードがでている。

この論文がこれからのコンパイラの作成の手助けにでもなれば幸いである。

参考文献

- [1] A.V. エイホ、R. セシィ、J.D. ウルマン：コンパイラ（下）：p.470：サイエンス社：1990
- [2] 松岡 恭正、河内 浩明、茂木 強：コンパイラにおける最適化の技法：インターフェース：Vol.15：No.3：pp.196-210(1989)

- [3] G.J.Chaitin : REGISTER ALLOCATION & SPILLING VIA GRAPH COLORING:ACM::: pp.98-105:(1982-06)
- [4] Peter A.Steenkiste,John L.Hennessy: A Simple Interprocedural Register Allocation and Its Effectiveness for LISP: ACM:Vol.11:No.1:pp.1-32:(1989-01)
- [5] James R.Larus, Paul N.Hilfinger: Register Allocation in the SPUR Lisp Compiler: ACM ::: pp.255-263:(1986-06)
- [6] Frederick Chow, John Hennessy: Register Allocation by Priority-based Coloring:ACM:::pp.98-108