インタバル分析を用いた広域最適化手法†

丸山勝己††

電子交換機のプログラムは同一のものが数百局で永年にわたって使用されるため，効率に対する要求が非常に厳しい．特に扱うデータが膨大なビット単位データであること，動的アドレスデータが多いこと，ベーシックブロックが小さくて狭域最適化コンパイラでは効果が出ないこと等から高度の広域最適化が必要である．本稿では，CCITT 勧告の交換機用言語 CHILL のコンパイラの実用化で実現した最適化手法を紹介する．この最適化はインタバル分析とトリーサブグラフを利用したもので簡単系統的な処理で高い有効性を実証した．

1. まえがき

計算機制御による電話交換機のプログラム（以下交換プログラムという）記述用の NTT 試作高水準言語 DPL[6),7)] 及び同目的の CCITT 勧告言語 CHILL[4),5)] の両最適化コンパイラの研究実用化で開発した最適化手法の基本を紹介する．交換プログラムは同一のものが数百局で永年にわたって使用されるため，効率に対する要求は特に厳しく高度な最適化コンパイラが必要とされる．ところが交換プログラムは以下の点に最適化の難しさがある．

（1） 条件分岐が多いためベーシックブロックが非常に小さく（平均5機械語ステップ位），これを最適化適用範囲とする狭域最適化だけでは効果が出にくい．

（2） 1語内に複数フィールドを詰めこんだ膨大な量のビット詰めデータを対象とするので，データアクセスが複雑であるのみならず，どの値がレジスタに乗っているかの管理がワード／バイト単位データの処理に比して格段に複雑・膨大である．

（3） ポインテッド変数・配列変数のようにオペランドアドレスが動的に計算される変数が多く使われ，これらの最適化は値の変化まで解析しないと行えない．

マルチレジスタマシンの場合は最適化処理のうちで最も影響力のあるのはレジスタ割付である[2),3)]．我々の分析では，一般の応用プログラムの場合は，プロシージャごとに最頻度変数を固定的にレジスタにのせた上で狭域最適化を強化することにより，オーバヘッド 20% 以下の機械語を生成できるが，交換プログラムを同一アルゴリズムで人手でコンパイルしてみたところ，オーバヘッドは80% を越すことが実証された．この対処として本稿に基づく広域最適化コンパイラを実用化し[8),9)]，本手法の簡単さと有効性を確認した．

2. フロー解析の基礎概念

本広域最適化手法は**インタバル**分割による制御フロー解析をベースにしているので，まずその基礎概念をまとめておく（文献1）参照）．

（1） **ベーシックブロック (BB)**：分岐を含まない1入口1出口の直線プログラム部分

（2） **フローグラフ (FG)**：BB をノード，BB 間の制御の流れをアークで表現した有向グラフ

（3） **インタバル (IVL)**：フローグラフのノード h が与えられたとき，h を唯一の入口ノードとし，その中のループは必ず h に戻るような最大のサブグラフを，h を**インタバルヘッド**とするインタバル I(h) と呼ぶ．これは次のように求まる．{ } は集合を表す．

① I(h)≔{h} からスタートする．

② ノード n の直前のノードがすべて I(h) に含まれているならば，その n を I(h) に加える．

③ ②を収束するまで繰り返す．

インタバルは，インタバルヘッドが唯一の入口であるが出口は複数有りうる．また定義より**ループ**は必ずインタバルヘッドに戻る．

（4） **インタバル分割**：フローグラフは次の手順によりユニークにいくつかのインタバルに分割される．

① H≔{e} ここに e はフローグラフの入口ノード

② h∈H に対して I(h) を作り，かつ I(h) の直後のノードでHに含まれないものがあればHに加える．

③ ②をHの全要素に対して終るまで繰り返す．

（5） ***n* 次インタバルグラフ (*n* 次 IVL)**：プログラムに対応するフローグラフをインタバル分割してできた各インタバルをノード，その間の制御の流れをアークで表現してできたグラフを1次インタバルグラ

† Global Optimization with Interval Analysis by KATSUMI MARUYAMA (Electrical Communication Laboratories, NTT)

†† NTT 電気通信研究所

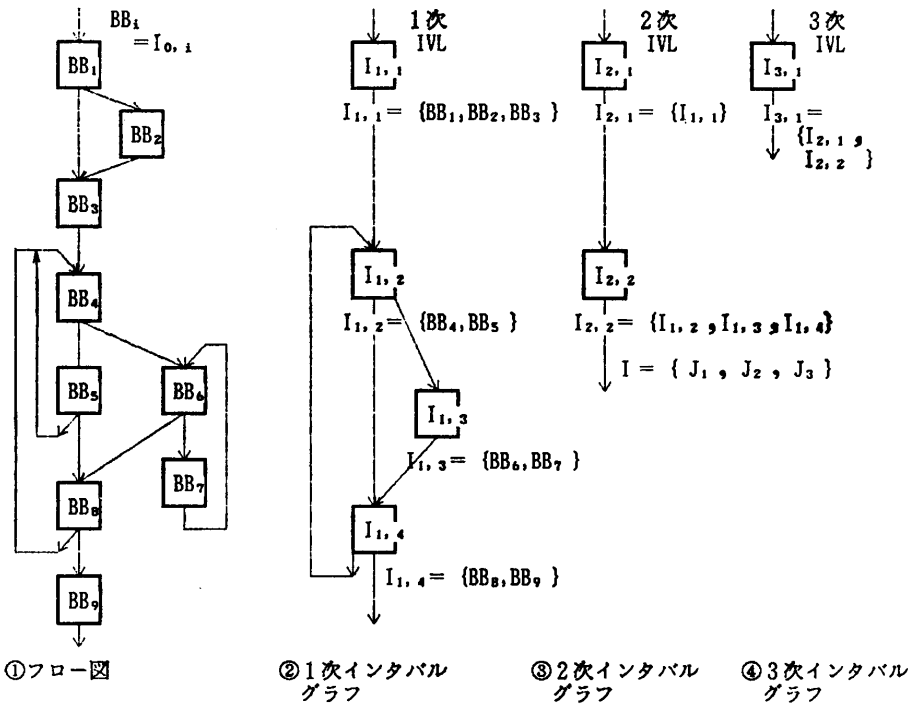


図 1 インタバルによる階層的分割
Fig. 1 Hierarchical interval partition.

フと呼ぶ．同様な処理を繰り返すと2次インタバルグラフが求まり，これを繰り返すと n 次インタバルグラフで1ノードに収束する（**図1参照**）．このようにフローグラフに対しインタバル分割を繰り返すことにより，フローグラフは階層的かつユニークに分割される．

（6） **インタバル要素順**：インタバルヘッドに戻るアーク（ループ）を切って考える．するとインタバルの各要素は制御の流れに対して**半順序関係**となり，以下のように並べられる．これをインタバル要素順という．

$I=\{J_1, J_2, \cdots, J_n\}$

ここに J_1 はインタバルヘッドであり，$i<j$ ならば J_i は J_j よりも先に実行されるか，もしくは J_i から J_j に行くパスが無い．

（7） **トリーサブグラフ（TSG）**：フローグラフを複数入力ノードの入口で分割してできるサブグラフ．TSG を**図2**の順で歩くことを**トリーウォーク順**という．

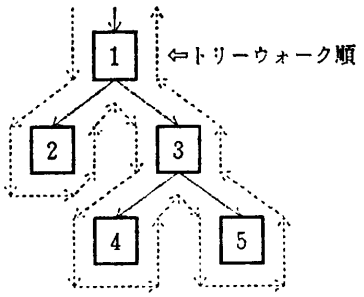


図 2 トリーサブグラフとトリーウォーク順
Fig. 2 Tree subgraph and tree walk.

3. 本コンパイラの構成

本コンパイラの構成を**図3**に示す．語彙分析・構文分析の出力は，制御フロー解析に入りベーシックブロック，インタバル，トリーサブグラフが得られる．次いでデータフロー解析により各変数のアクティビティが分析される．制御・データフロー解析の結果に基づいて広域レジスタ割付を行い，さらにこれら全部の結果を用いてトリーサブグラフごとに機械語に変換する．複雑な処理とコンパイル時間を許せば幾らも高級な広域最適化は可能であるが，インタバルグラフとトリーサブグラフに基づいた本手法は比較的簡単な処理で実効上十分な最適化を達成できることに特色がある．

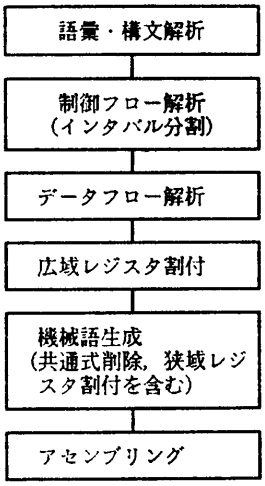


図 3 コンパイラの構成
Fig. 3 Compiler structure.

4. データフロー解析

後述の広域レジスタ割付やレジスタ管理をするためには，フローグラフ上の各点で各変数の値が**アクティブ**（その値が後で使われる）かどうかを知ることが必須である．この解析をデータフロー解析と呼ぶ．データフロー解析は，ループが無ければフローを逆にたどることで簡単に求まるが，ループの対処が問題でいろいろの工夫がとられる．ここにインタバル分割の性質『プログラムが階層的に分割され，かつループはインタバルヘッドに戻る』を活用すると，以下のように2パス処理で系統的に解析することができる．またインタバルは，後述のように広域レジスタ割付け等の最適化の単位としても使われるので二重に有効である．

①1パス目：1次から最高次インタバルグラフの昇順に各インタバルごとの各変数の値の読出し（以下**参照**という），値の書込み（以下**代入**という）を解析する．

②2パス目：最高次から1次インタバルグラフの降順に処理して各インタバルの入口・出口での各変数のアクティビティを解析する．

ここで次の記法を用いる．

① ある次数のインタバルIの要素(つまり1次数若番のインタバル) を $J_1, J_2, \cdots, J_n$ とする．特にIが1次インタバルの場合は，J_i はベーシックブロック BB となる．インタバル J_i は複数出口を持ちうるので，k 番目の出口を指定する時は上添字 J_i^k で示す．

② **use** (J_i)：インタバル J_i で該当変数の値が（代入が有る場合はその前に）参照される場合をT（真），それ以外の場合をF（偽）で表す．

③ **set** (J_i^k)：インタバル J_i の k 番目の出口に至るパスで，該当変数に必ず代入が有る時T，それ以外をFで表す．

解析結果は以下のように表す．

④ **aen** (J_i)：インタバル J_i の入口 entry で該当変数がアクティブの時T，そうでない時F．

⑤ **aex** (J_i^k)：インタバル J_i の k 番目の出口 exit で該当変数がアクティブの時T．

4.1 パス1：各インタバルの use/set 情報の解析

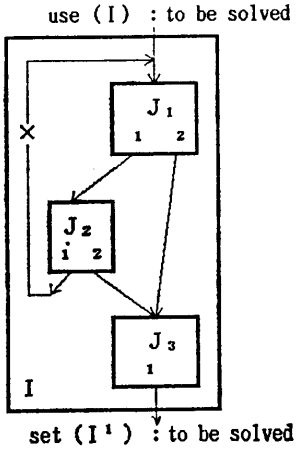
ベーシックブロックの use/set はプログラムから直ちに求まる．また，インタバル $I=\{J_1, J_2, \cdots\}$ の use/set は $J_1, J_2, \cdots$ の use/set を用いて以下のように計算できる．この計算の鍵は**ループを無視する**ということである．したがって1次から n 次インタバルグラフまで昇順にこの計算を繰り返すことにより，全インタバルの use/set が計算できる（図4参照）．ここに $\sum$ は論理和，$\prod$ は論理積を表す．実際の処理は，各変数を座標とするビット列演算の繰り返し（J_i を逆インタバル順にたどると再帰計算は不要）により複数の変数を一度に高速に計算できる．

$$\mathrm{use}(I) = \mathrm{usef}(J_1)$$

ここに J_1 はIのインタバルヘッドで，かつ

$$\mathrm{usef}(J_i) = \mathrm{use}(J_i) \ \mathbf{or} \ \sum_{e:\ J_i \text{ の全出口}} \{\mathbf{not}\ \mathrm{set}(J_i^e)\ \mathbf{and} \sum_{J_s:\ J_i \text{ の } e \text{ 番出口の直後の全インタバル．ただし } J_1 \text{ を除く}} \mathrm{usef}(J_s)\}$$

$$\mathrm{set}(I^e) = \mathrm{setf}(J_i^e)$$



{J1, J2, J3} の use/set 情報から
I の use/set 情報を求める．
ループを切って考える．

use(I) = use (J1) **or**
not set (J1^1) **and** use (J2) **or**
not set (J1^1) **and not** set (J2^1)
and use (J3) **or**
not set (J1^1) **and** use (J3)

set (I^1) = set (J3^1) **or**
set (J2^1) **and** (set (J1^1) **or** set (J3^2))

図 4 Use/Set の解析の例
Fig. 4 Use/Set analysis example.

ここに出口 I^e は出口 J_i^k に対応し，かつ

$$setf(J_i^k) = set(J_i^k) \ \textbf{or} \ \prod_{J_m^n} setf(J_m^n)$$

J_m^n：J_i の直前の全インタバル出口．ただしループを除く

4.2 パス 2：変数のアクティビティの計算

階層的に分割された各インタバルの use/set を用いて以下のように最高次から1次インタバルグラフに向けて降順に計算することにより，各インタバルの入口と出口のアクティビティ aen/aex が求まる（**図5**参照）．0次インタバル (BB) の出入り口のアクティビティが求まれば，BB 内のすべての点でのアクティビティが直ちに求まる．

（1） 最高次インタバル

ローカル変数は出口で必ず nonactive であるので，**aex (各出口)=F** である．また入口の aen は，use と一致するので **aen (I)** = **use (I)** である．

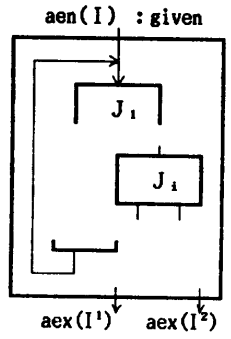
（2） m 次 IVL 情報から $m-1$ 次 IVL 情報の計算

m 次インタバル I の aen/aex 情報及びパス1で求められた use/set 情報から，その要素インタバル $\{J_1, J_2, \cdots\}$ の aen/aex が，以下のように計算できる．この計算も各変数を座標とするビット列演算の繰り返し（再帰でなく）で実現できるので高速である．

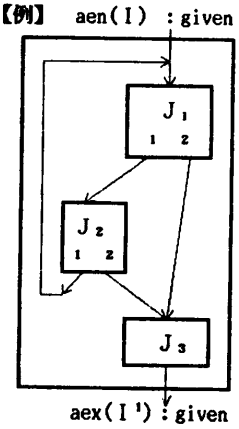
$$aen(J_1) = aen(I)$$

$$aex(J_i^k) = \sum_{J_s:\ J_i\ の\ k\ 番出口の直後の全インタバル} aen(J_s)$$

$$aen(J_i) = use(J_i) \ \textbf{or} \ \sum_{k:\ J_i\ の全出口} \{\textbf{not}\ set(J_i^k)\ \textbf{and}\ aex(J_i^k)\}$$



I の情報：aen (I), aex (I^e)
が与えられて，
要素 $\{J_1, J_2, \cdots\}$ の aen, aex 情報を求める．

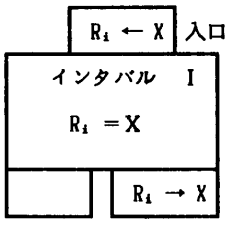


$aen(J_3) =$
use (J_3) **or**
not set (J_3^1) **and** aex (J_3^1)
$aex(J_3^1) = aex(I^1)$ ——右辺は given

$aen(J_2) =$ use (J_2) **or**
not set (J_2^1) **and** aex (J_2^1) **or**
not set (J_2^2) **and** aex (J_2^2)

$aex(J_2^1) = aen(J_3)$
$aen(J_3) = aen(I)$ ——右辺は given
$aex(J_2^2) = aen(J_3)$
…以下同様…

図5 変数のアクティビティ解析とその例
Fig. 5 Activity analysis and example.



入口ポート： $R_i \leftarrow X$ **は変数X をレジスタ** R_i **にロードすることを意味する。**

$R_i = X$ **は変数X がレジスタ** R_i **に固定されている事を意味する。**

出口ポート： $R_i \rightarrow X$ **はレジスタ** R_i **の内容を変数X のメモリにストアすることを意味する。**

図6 入口/出口ポート
Fig. 6 Entry/Exit ports and notation.

5. インタバル単位の広域レジスタ割付

広域レジスタ割付の**鍵**は，各変数のデータフロー情報とアクセス回数をもとに『**どの変数をどの範囲でどのレジスタ**に乗せるか（**レジスタ固定**と呼ぶ）』の決定にある．本手法の発想は，インタバル単位にアクセス頻度の高い変数をレジスタに固定することである．これは以下の点から妥当である．

①インタバル分割によりプログラムが階層的に分割されるので，系統的処理に向く．

②ループは必ずインタバルヘッドに戻るのでループの最適化も同時に行える．

③インタバル分割は経験的にみてプログラムの動作をよく反映した階層的分割となっている．

広域レジスタ割付の処理を以下に説明する．ここに**図6の入口／出口ポート**とは，変数をレジスタ固定するためにインタバルの入口・出口にロード／ストア命令を置くための場所である．

（1） **広域レジスタ割付対象変数の選択**

各インタバルにおいて，各変数がレジスタ固定された場合とメモリ上にある場合との機械語ステップ数の差を参照・代入回数，ループ内の重み付け，レジスタ固定に要するロード・ストアの個数等から計算する．これは変数，インタバル，レジスタの関数であり**レジ**

スタ利得 merit (var, ivl, reg) と呼ぶ.

（2） **広域レジスタ割付（図 7）**

（1）の結果を用いて，全体として最も利得の大きくなる，つまり『$\sum_{var}\sum_{ivl}$ merit (var, ivl, reg)』を最大化する［インタバル，変数，広域割付用レジスタ］の組み合わせを図7の簡略手法で求める．すなわち，まず最高次インタバル上で最大利得となる組み合わせを求める．次いで $n-1$ 次の各インタバル上でこれよりも利得の大きくなる組み合わせが有ればそれに置き換える．これを1次インタバルまで繰り返す．なお，固定されたレジスタでも，該当変数がアクティブで無い時は，ワーク用として使ってよい．

（3） **入口ポートのロード命令削除（図 8）**

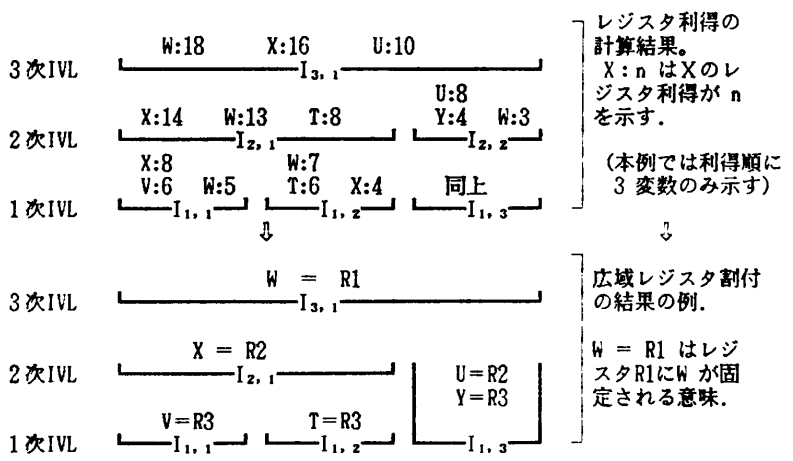
入口ポートのロード命令は，その変数が直前の全インタバルでレジスタ固定されている場合には削除する．この処理はインタバル要素順に従って実行することにより，後戻りなく行える．

（4） **出口ポートのストア命令削除**（図 8）

前記のロード命令削除の結果出口ポートに置いたストアを受ける全ロード命令が無くなった場合，このストア命令を削除する．ここにストアを受けるものは入口ポートに限定されないことに注意.

（5） **ループの最適化（図 9）**

多重ループに対しロード・ストア命令をできるだけ外側のループに移すため，入口ポートのロード命令を親インタバルの入口ポートに移す（必要ならばループの戻りノードの出口ポートにロード命令を出す）．また出口ポートのストア命令も親インタバルの出口ポートに移す．ループはインタバルヘッドに戻る性質のために必ずインタバルに対応するので，本手法により多重ループは内側から外側に向かって最適化される．



広域レジスタ固定数を 3個とする．最初 3次IVL でW,X,U を選ぶ．次に 2次IVL の $I_{2,1}$では UをT に，$I_{2,2}$では XをV に換えた方が得なので差し替える．最後に 1次IVL の $I_{1,1}$では TをV に換えた方が得なので差し替える．

図 7 インタバル単位の広域レジスタ割付例
Fig. 7 Interval based global register allocation.

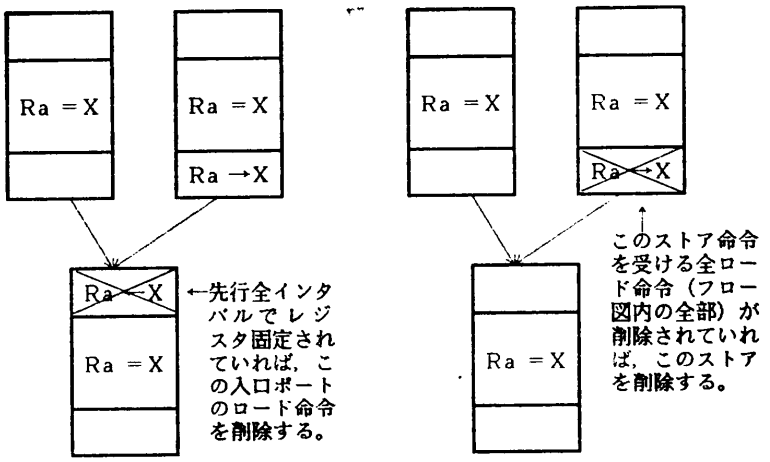


図 8 ロード命令の削除とストア命令の削除
Fig. 8 Load elimination and store elimination.

6. トリーサブグラフ TSG ベースの処理

交換プログラムはベーシックブロック BB が小さいので共通式削除や狭域最適化を BB 内で行っても効果が小さい．そこで共通式削除と機械語生成を TSG ごとにトリーウォーク順で行うことにより，簡単な処理で狭域最適化の範囲も BB から TSG に拡大した.

6.1 トリーサブグラフ TSG 内での共通式削除

交換プログラムでは実行時にアドレスが計算されるオペランドが頻繁に使われ，その最適化は重要である.

【例】 X := P -> Q(I); (1)

:

Y := P -> Q(I); (2)

この例で(1)の右辺値が(2)で使えるのは，P, I 及び P -> Q(I) の値が更新されなかった場合のみである．

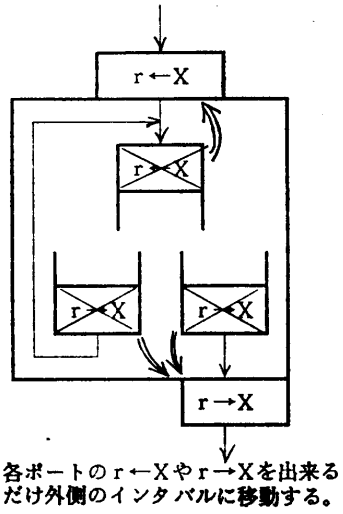


図 9 ループの最適化
Fig. 9 Loop optimization.

このような動的オペランドの最適化には共通式削除の技法が必要になる．完全な広域共通式削除は計算機時間がかかるが，TSG 内に限定することにより，BB 内とほぼ同様な簡単な処理で実現できる（図 10 参照）．

（1） TSG 内の各 BB をトリーウォーク順に歩いて中間言語を作成していく．

（2） 作成された中間言語のうち共通式削除の対象となりうるもの，及び代入命令のコピーを共通式スタックに記録しておく．

（3） 中間言語作成ごとに共通式スタックをそのオペランドに代入が無い範囲でサーチし，一致するものが見つかったらそれを使い今作成した中間言語は捨てる．

（4） 一つの枝を処理し終えて次の枝に移る時には，共通式スタックから前の枝の分を削除する．

6.2 機械語生成とレジスタ割付

機械語生成時のレジスタ要求には以下がある．

（1） 値参照用レジスタ要求（例 LOAD R 1←X）

（2） ワーク／値代入用レジスタ要求（例 STORE R 2→Y）

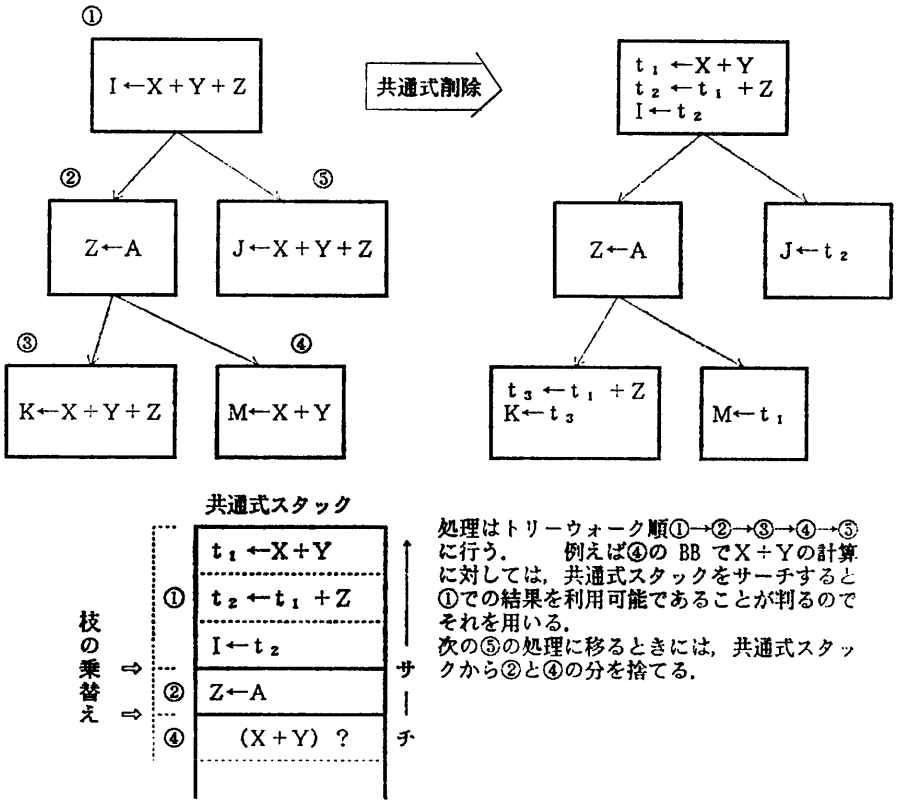


図 10 トリーサブグラフ内での共通式削除
Fig. 10 Tree subgraph common expression elimination.

変数の値とレジスタ及びメモリの関係を管理するために以下の**変数状態**を用いる．老番ほどレジスタの解放に必要なステップ数が多いので高優先度と呼ぶ．

（1） メモリのみ：変数の値はメモリ上にのみ存在．

（2） 全同値：変数の値がレジスタに乗っており，かつメモリにストア済である．

（3） 半同値：変数の値がレジスタに乗っているが，メモリには未ストアである．

（4） 全固定：変数が広域レジスタ割付でレジスタ固定済で，かつメモリにストアされている．

（5） 半固定：変数が広域レジスタ割付でレジスタ固定済であるが，メモリには未ストアである．

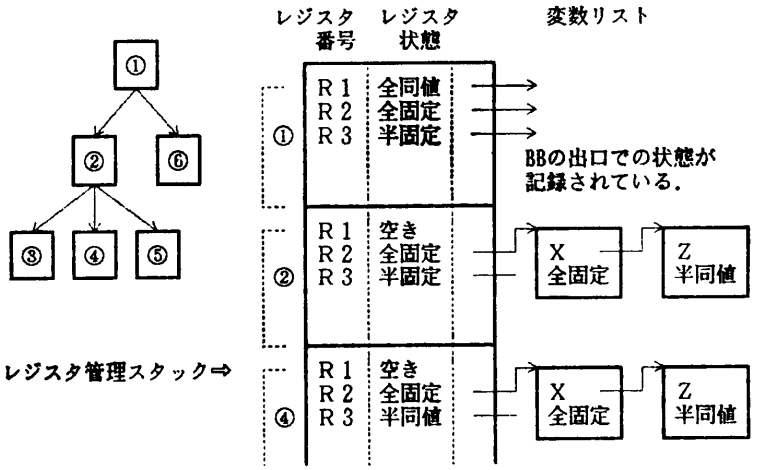


図 11 トリーベースの狭域レジスタ割付
Fig. 11 Tree based local register allocation.

また，**レジスタ状態**は，変数の値が乗っていればその中で最優先の変数状態，さもなければ空きと定義する．例えば変数 X が R1 に全固定，変数 Y がメモリのみの場合に，“$X := Y$;” の代入処理で変数 Y の値をメモリから R1 にロード (LOAD R1←Y) すると，X が半固定，Y が全同値，R1 が半固定となる．

この［レジスタ，変数，状態］の関係を管理するために，**図 11** に示す**レジスタ管理スタック**を用いる．TSG 内でレジスタ引き継ぎを実現するため，トリーウォーク順で機械語を生成しながら各 BB の出口でレジスタ管理スタックに記録し，枝を移るごとに前の枝の分を捨てていく．

機械語生成でのレジスタ割付は，データフロー解析結果（変数のアクティブ性）を参照し，レジスタ管理スタックを更新しながら以下の手順で行う．

（1） **参照用レジスタ要求**：レジスタ管理スタックをサーチして対象変数の乗っているレジスタを捜し，見つかればそれを用い，さもなければ下記(2)によりレジスタを確保してそこにロードする．

（2） **代入用／ワーク用レジスタ要求**：

①空きレジスタが有ればそれを確保する．無ければ：②最低優先度状態のレジスタを確保する．同一優先度間では次の参照までの距離の長い方から確保する．ここに距離とは，TSG 上で値の代入から参照までの間の命令数であり，参照が複数個有る場合は並列抵抗値の計算の類推で調和平均をとって $1/(1/L_1+\cdots+1/L_n)$ と計算する．また半同値・半固定の両状態の場合は，値をメモリに返す必要があるのでストア命令を出した上でレジスタを解放する．

（3） 乗っていた全変数がアクティブで無くなった時には，そのレジスタを空き状態にする．

（4） 値代入に対しては，その変数が TSG 出口でアクティブでかつ最後の代入の場合のみストア命令を出し，それ以外はレジスタへ乗せたままとする．

このようにトリーサブグラフのトリーウォーク処理は，少ないメモリと簡単な処理で効果が大きい．

7. 結　　言

本方式に基づいて実用化した NTT の交換機用プロセッサ用 CHILL コンパイラの出力コードから熟練者がさらに消去できた冗長コードは十数%以下で，ほぼアセンブラ並の効率を達成した．本コンパイラは既に多数の商用交換プログラム作成に使用されている．広域最適化は理論的には非常に高度な処理も可能であるが，効果の度合に比べて処理の複雑度と計算機時間が拡大する傾向にあるのに対し，インタバルの階層的分割とトリーサブグラフを利用した本手法は比較的簡単かつ系統的な処理で効果的な広域最適化を実現できた．本コンパイラは，ビット詰め構造体データを膨大に（数百～）持つ交換プログラムを少ない主メモリ上でコンパイルできるようにするために，最適化処理関連データ構造を工夫しさらにこれを二次記憶装置に置

いたことで複雑化したが，最適化処理そのものは単純である．また，コンパイル時間も二次記憶装置の使用による増加が主で，最適化処理そのものの計算時間は小さい．

謝辞 本方式の検討に対し多くの上司及び同僚の方方の御指導・御協力をいただいた．ここに深謝いたします．

参考文献

1) Allen, F. E.: Control Flow Analysis, *Sigplan Notice, Proceedings of a Symposium on Compiler Optimization*, pp. 1-19 (1970).
2) Lowly, E. S. et al.: Object Code Optimization, *CACM*, Vol. 12, No. 1, pp. 13-22 (1969).
3) Betty, J. C.: Register Assignment for Generation of Highly Optimized Object Code, *IBM J.*, Vol. 18, No. 1, pp. 20-39 (1974).
4) CCITT: Z. 200 CHILL: A Recommendation for a CCITT High Level Programming Language.
5) 丸山ほか：交換プログラム用仕様記述法と高水準言語，情報処理，Vol. 21, No. 3, p. 233 (1979).
6) Kakuma, O. and Maruyama, K.: DPL—A High Level Programming Language for Electronic Switching Systems, ISS 76 (1976).
7) 工藤，丸山：交換プログラム作成用言語 DPL，研究実用化報告，Vol. 24, No. 11, pp. 2457-2473 (1975).
8) 丸山ほか：交換プログラム作成用言語 CHILL の実用化，研究実用化報告，Vol. 31, No. 3, pp. 631-647 (1982).
9) 丸山ほか：交換プログラム記述用並列処理言語の実現，情報処理，Vol. 26, No. 3, pp. 407-413 (1985).

(昭和59年4月11日受付)
(昭和61年5月15日採録)

丸山 勝己（正会員）

1944年生．1968年東京大学工学部電子工学科卒業．1970年同大学院修士課程修了．同年日本電信電話公社入社．現在NTT通信網第一研究所勤務．電子交換機実時間増設方式，電子交換機用高水準言語とコンパイラ，交換プログラム構造などの研究実用化に従事．また，1973年～1980年まで CCITT（国際電信電話諮問委員会）高水準言語設計チームの一員として，CHILL の設計に従事．1985年より CCITT 第X研究委員会の副議長として仕様記述言語の検討を担当．著書「交換用プログラミング言語 CHILL」（電気通信協会）．電子通信学会会員．