Vol. 23 No. 3 情報処理学会論文誌 May 1982

拡張データ依存関係グラフを用いたマイクロプログラムの大局的並列化法†

迫田 行介††

水平型マイクロプログラムをマイクロ操作系列から自動的に構成する実用的な大局的並列化アルゴリズムを提案する.

従来の並列化アルゴリズムの多くは,マイクロ操作の移動が基本ブロック内に限定されていたり,マイクロ操作系列が構造化されていることが前提になっているもので,並列化の効果や実用性に問題があった.

これに対して本アルゴリズムでは,いくつかの基本ブロックを連結した拡張ブロックごとに並列化を行うので,任意の構造のマイクロ操作系列に対して従来の2倍以上の並列化効果が得られ,実行効率やメモリ効率の改善が図れる.

拡張ブロックとは,制御フローの合流点から,途中の条件分岐では任意の分岐先を選びながら次の合流点までたどる任意のパスである.拡張ブロック内のデータ依存関係を保存するために,条件分岐から拡張ブロック外への分岐先でのライブ変数をデータフロー解析して求め,その条件分岐の最後の仮想サブマシンサイクルでその変数が参照されているように扱う.こうすることにより拡張ブロックは従来の基本ブロックと同様の扱いで並列化ができるようになる.

拡張ブロックの決め方は一通りではないがヒューリステックな方法で一意的に決定しても拡張ブロックを用いた大局的並列化の効果はほとんど変らない.

1. まえがき

水平型のマイクロプログラムでは,いくつものマイクロ操作(以後 μOP と略す)を同一マシンサイクル内で実行することができる.

μOP 系列から同時に実行できる μOP 群を抽出する方法(これを並列化と呼ぶ)が研究されてきた[1),2)].その多くは途中に分岐 μOP を含まない μOP 系列を対象とした局所的並列化と呼ばれるものである.この方法では,実際のマイクロプログラムの基本ブロック(以後 BB と略す)ごとに並列化することになるが,BB の中の μOP の数は比較的少なく[6)],並列化の効果も小さい.実用面からは,BB の枠を越えて μOP を抽出できる大局的並列化が必要である.

大局的並列化を論じているものに Desgupta[3)], Tokoro[4)], Wood[5)], Poe[6)] がある.Dasgupta[3)] はループのないマイクロプログラムに関して並列化できる条件を示している.Wood[5)] は構造化プログラミングされている場合を論じ,Poe[6)] は if-then, do-while 構造が抽出されることを前提としており,いずれも性能や容量の制限のきつい実際のマイクロプログラムへの適用はむずかしい.Tokoro[4)] は,BB 間で μOP が移動できるかと逐一調べる方法を提案しているが,移動は BB 間の境界に限定されている.

本論文では,1つ以上の BB が連続した拡張ブロック(以後 EB と略す)を導入し,EB ごとにデータ依存関係グラフを求めることにより,大局的並列化を行う実用的なアルゴリズムを提案する.

μOP 系列の制御の流れの合流点から次の合流点の前までの任意のパス(途中に合流点を含まないが分岐点を含むことはできる)を EB と定義する.途中の分岐点では任意の分岐先を1つ選ぶ.分岐点の前後の μOP が並列化できるためには,選ばなかった分岐先でのデータの参照状況が関係する.これは μOP 系列全体についてデータフロー解析することで容易に求められる.この条件が求まれば EB に関してデータ依存関係グラフが作成でき,たとえば Yau[9)] のような方法で並列化ができる.

拡張ブロックの決め方は一通りではない.すべての組合わせを計算するのでは計算量が多すぎる.ヒューリステックな方法で一意的に決定する実用的な方法でも拡張ブロックを用いた大局的並列化の効果はほとんど変らない.

実用的な大局的並列化アルゴリズムの実現に伴い,マイクロプログラミングシステムの実用化,高級言語によるマイクロプログラムの記述とそのコンパイラの実用化が期待できる.

† Global Microcode Packing using Extended Data Dependency Graph by KOUSUKE SAKODA (Systems Development Laboratory, Hitachi Ltd.).

†† (株)日立製作所システム開発研究所

2. 大局的データ依存関係の解析

2.1 大局的並列化

μOP m_i の系列

$$p=m_1, m_2, \cdots, m_n$$

から，その機能を変えることなく，同時に実行可能な μOP の集合（これをマイクロ命令と呼び，以後 μI と略す）M_I の系列

$$P=M_1, M_2, \cdots, M_N$$

ただし，$M_I=(m_{i_1}, m_{i_2}, \cdots, m_{i_I})$

に変換することを並列化と定義する．

並列化条件として，次の2つの条件が満たされている必要がある．

（1） 同一 μI 内の任意の2つの μOP 間にリソースの競合はない．

（2） 任意の2つの μOP 間のデータ依存関係は，並列化の前後で保存されている．

（1）の条件は，同一 μI 内のすべての μOP が同一マシンサイクルで実行できるための競合回避条件である．この条件は，2つの μOP が同一ハードウェアリソース（レジスタ，演算器，バスなど）を使用しているかどうか，同一マイクロフィールドを異なる目的で使用したり，競合しているフィールドを使用しているかどうかをチェックすればよい．チェックする関数 RC を次のように定義する

$$\mathrm{RC}(i, j)=\begin{cases}0 & m_i \text{ と } m_j \text{ はリソースを競合しない．}\\ 1 & m_i \text{ と } m_j \text{ はリソースを競合する．}\end{cases}$$

（2）の条件は，並列化によって μOP の系列 p と μI の系列 P の機能が変らないためのデータ依存条件である．この条件は，記憶型リソース（レジスタ，メモリ，フリップフロップなど．以後変数と呼ぶ）にデータ値を定義したり，そのデータ値を参照している μOP 間の実行順序関係を求め，それが保存されていることをチェックする必要がある．この条件を基本ブロック（以後 BB と略す）単位に求めるのが局所的並列化，BB 間にまたがって求めるのが大局的並列化である．

2.2 拡張ブロック

μOP 系列 p の部分集合で，その先頭以外に合流点や p の入口点を含まない μOP の系列を拡張ブロック（以後 EB と略す）と定義する．途中に条件分岐 μOP がある場合には，どれか1つの分岐先を選ぶものとする．

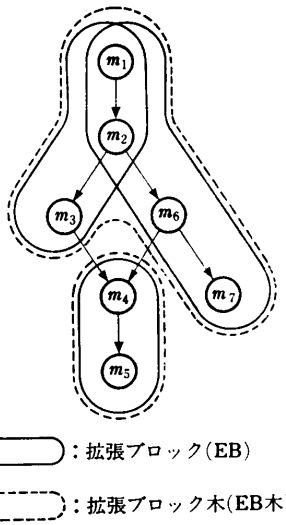

図 1 拡張ブロックと拡張ブロック木の例
Fig. 1 Example of extended block and extended block tree.

他の分岐先を選んだとき，それは別の1つの EB と考える．このとき両 EB は共通部分を持つ．μOP を節，制御の流れを枝とすると，共通部分を持つ EB の和集合は木構造を構成する．これを拡張ブロック木（以後 EB 木と略す）と呼ぶことにする．EB 木の根は p の入口点か合流点であり，各葉は p の出口点か合流点の直前の節である．

EB が，その途中に条件分岐を含むことがあるのでその大きさは BB の大きさに等しいか，より大きい．

図1に EB の例と EB 木の例を示す．いま，m_1, m_2, m_3 を EB としたとき，条件分岐 μOP m_2 にとって m_3 は EB 内の分岐先であり，m_6 は EB 外の分岐先となる．

2.3 データ依存性の保存

EB 内のデータ依存関係を求めるのに，まず EB 内の条件分岐 μOP の分岐先以後で参照される可能性のある変数の集合を求める必要がある．

図2で，いま，m_1〜m_4 の μOP が1つの EB に含まれているものとする．m_2 の分岐先 m_5 あるいはそれ以後で変数 v が参照されていると，m_2 の後の m_3 は m_1 以後 m_2 までの間に移動できるが，m_4 を（m_3 より前はもちろん）m_2 より前に移動することはできない．また，m_1 を m_2 より後に移動することもできない．

このような条件を保存したまま，EB をあたかも BB のように扱う，つまり条件分岐を意識しなくてもすむようにするには次のようにすればよい．実際の最後の

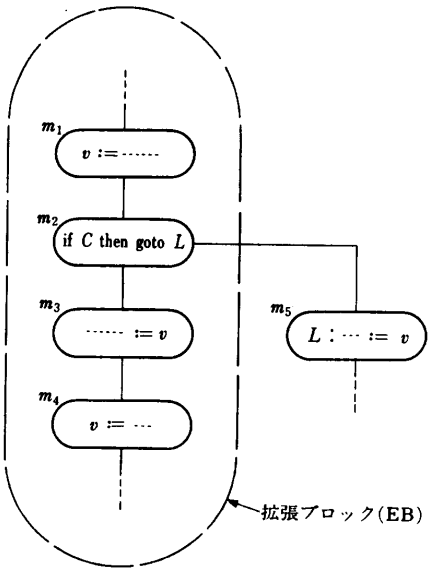

図 2 条件分岐前後のデータ依存関係
Fig. 2 Data dependencies around conditional branch.

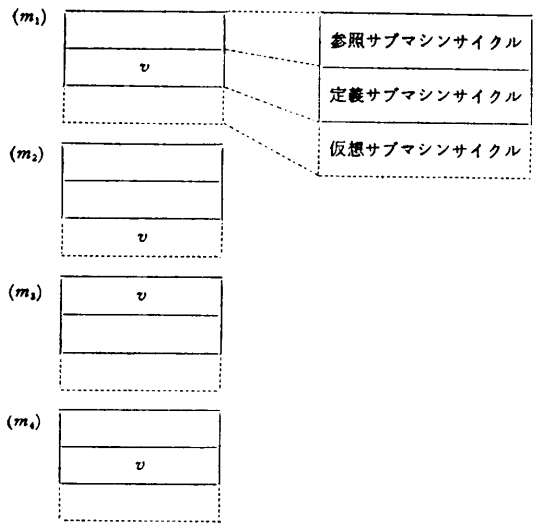

図 3 仮想サブマシンサイクル
Fig. 3 Hypothetical sub-machine cycle.

サブマシンサイクルの次に，もう1つ仮想的なサブマシンサイクルを想定し，EB 外の分岐先で参照される可能性のあるすべての変数が，当条件分岐 μOP の仮想サブマシンサイクルで参照されていると考えればよい．**図3**は仮想サブマシンサイクルを説明する例で，参照と定義のサブマシンサイクルの次に仮想サブマシンサイクルを考える．図2に対応させた変数が記入してある．

2.4 データフロー解析

k 番目の基本ブロック BB_k の入口点での値が，それ以後で参照されている，あるいは参照される可能性のある変数の集合をライブ変数と呼び，ビットベクトル $live_k(v)$ で表わすことにする．すなわち，

$$live_k(v) = \begin{cases} 0 & \text{変数 } v \text{ は } BB_k \text{ のライブ変数でない,} \\ 1 & \text{変数 } v \text{ は } BB_k \text{ のライブ変数である,} \end{cases}$$

とする．

ライブ変数は次のようにして求める．まず BB_k 内の j 番目の μOP, m_i で参照している変数の集合をビットベクトル $ref_i(v)$ で表わす．すなわち

$$ref_i(v) = \begin{cases} 0 & m_i \text{ で変数 } v \text{ を参照していない,} \\ 1 & m_i \text{ で変数 } v \text{ を参照している.} \end{cases}$$

同様に，m_i で定義している変数をビットベクトル $def_i(v)$ で表わす．

$$def_i(v) = \begin{cases} 0 & m_i \text{ で変数 } v \text{ を定義していない.} \\ 1 & m_i \text{ で変数 } v \text{ を定義している.} \end{cases}$$

このとき，BB_k 内で参照している変数と定義している変数をそれぞれビットベクトル $bref_i(v)$ と $bdef_i(v)$ で表わすことにする．BB_k 内の任意の μOP で変数 v を定義していれば，その BB_k でその変数を定義しているものとし，BB_k 内の任意の μOP で変数 v を参照しており，その参照している μOP より前に変数 v を定義している μOP がなければ，その BB_k でその変数を参照しているものとする．

$bref_i(v)$ と $bdef_i(v)$ および制御フロー情報から，ライブ変数 $live_k(v)$ を求めるアルゴリズムは Hecht[7] に示されている．アルゴリズムを付録に示す．

2.5 強・弱順序関係

μOP m_i が，ある変数 v を定義し，その定義内容を後の μOP m_j で参照している場合，m_i と m_j は必ずこの順序で実行されなければならない．また m_j より後の m_k で変数 v を再定義している場合，m_j と m_k はこの順序で実行される必要がある*．

多相制御している場合，変数によっては参照のサブマシンサイクルと定義のサブマシンサイクルが競合しないことがある．両者が競合せず，参照サブマシンサイクルの方が早いとき

$$rtime(v) < dtime(v)$$

と表わし，その逆のとき

$$dtime(v) < rtime(v)$$

と表わすことにする．両者が競合するときは，そのど

* 変数の定義の順序を入れ換えると，一般に割付け時にデッドロックを生じる可能性がある[9]．生じさせないための条件は複雑で，それを適用しても実用上の効果は少ない．

ちらでもないとする．このような関係にある変数を参照・定義している μOP 対は同時に実行できる可能性がある．この性質を考慮して，同一 EB 内の 2 つの μOP m_i と m_j に関して，次の 2 種類の順序関係を定義する．

（1） 強順序関係

m_i を実行するマシンサイクルより後のマシンサイクルで m_j が実行されなければならない順序関係で，(m_i, m_j) と記すことにする．

（2） 弱順序関係

m_i を実行するマシンサイクルと同一マシンサイクルか，それ以後のマシンサイクルで m_j が実行されなければならない順序関係で，$〔m_i, m_j〕$ と記すことにする．

1 つの EB 内の順序関係を具体的に求める方法を示す．ここで m_i, m_k, m_j に関して $i<k<j$ であると仮定する．

(イ) $\mathrm{def}_i(v)=1$ かつ $\mathrm{ref}_j(v)=1$ かつ任意の k について $\mathrm{def}_k(v)=0$ のとき，$\mathrm{dtime}(v)<\mathrm{rtime}(v)$ ならば $〔m_i, m_j〕$，そうでなければ (m_i, m_j)，

(ロ) $\mathrm{ref}_i(v)=1$ かつ $\mathrm{def}_j(v)=1$ かつ，任意の k について $\mathrm{def}_k(v)=0$ のとき，$\mathrm{rtime}(v)<\mathrm{dtime}(v)$ ならば $〔m_i, m_j〕$，そうでなければ (m_i, m_j)，

(ハ) $\mathrm{def}_i(v)=1$ かつ m_j が条件分岐 μOP かつ任意の k について $\mathrm{def}_k(v)=0$ のとき，m_j から EB 外への分岐先である BB のうちの少なくとも 1 つ BB_i において $\mathrm{live}_i(v)=1$ ならば，$〔m_i, m_j〕$，

という順序関係があるものとし，これ以外の場合には順序関係がないものとする．

2.6 拡張データ依存関係グラフ

拡張データ依存関係グラフ（以後 EG と略す）とは EB 内のすべての μOP の順序関係を表わした有効グラフである．$\mu\mathrm{OP}m_i$ に対して節点 n_i が対応し，m_i と m_j の間に順序関係に対して節点 n_i から節点 n_j への有向枝 a_{ij} が対応する（**図 4**）．

EB 内の制御フローが acyclic であることから，EG も acyclic graph になる．

有向枝には順序関係の種別に対応して強順序枝と弱順序枝がある．2 つの隣接する節点間に 2 つ以上の有向枝があるときは，1 つだけ残して他を取り除くことができる．このとき強・弱両方の順序枝があれば，強順序枝を残す．

このようにしてできた EG は，EB 内の μOP の実行順序を規定する．ある $\mu\mathrm{OP}m_i$ に対応する節点 n_i

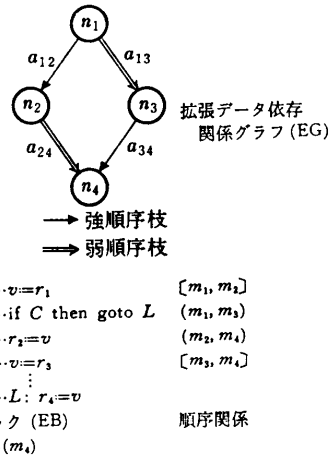

図 4 拡張ブロックの順序関係と拡張データ依存関係グラフ

Fig. 4 Extended block and it's extended data dependency graph.

への入力枝の数 $\mathrm{in}\sharp(n_i)$ が正のとき，その入力枝の始点であるすべての節点に対応する μOP が実行されなければ m_i を実行することができないことを示している．

ある時点で，入力枝の数がゼロであるすべての節点（この集合を Data Available Set という．以後 DAS と略す）に対応する μOP は，リソースの競合がなければ同時に実行可能であることを示している．

μOP が実行されると，その μOP に対応する節点と，その節点を始点とするすべての有向枝を取り除く．このとき入力枝数がゼロになった節点は DAS に加えることができる．（実際には後述のごとく，DAS=$R \cup D$, $R \cap D=\phi$ なる 2 つの部分集合について考える．）

3. 大局的並列化アルゴリズム

3.1 並列化基本アルゴリズム

μOP 系列 $p=m_1, m_2, \cdots, m_n$ から，μI 系列 $P=M_1, M_2, \cdots, M_N$ を求めるアルゴリズムを示す．具体的には各 μI, M_I に割り付ける μOP, m_i の集合を求める．目的は μI のステップ数 N を最小にすることであるが，実用性からは計算時間が少ないことも重要である．

まず，p をいくつかの EB に分割する．この EB 群は互いに共通部を持たず，p のすべてをつくしているものとする．EB 群の決め方は一通りではなく，この決め方によって最終的なステップ数 N は異なる．これについては **3.3** に述べることにし，ここでは一通りの

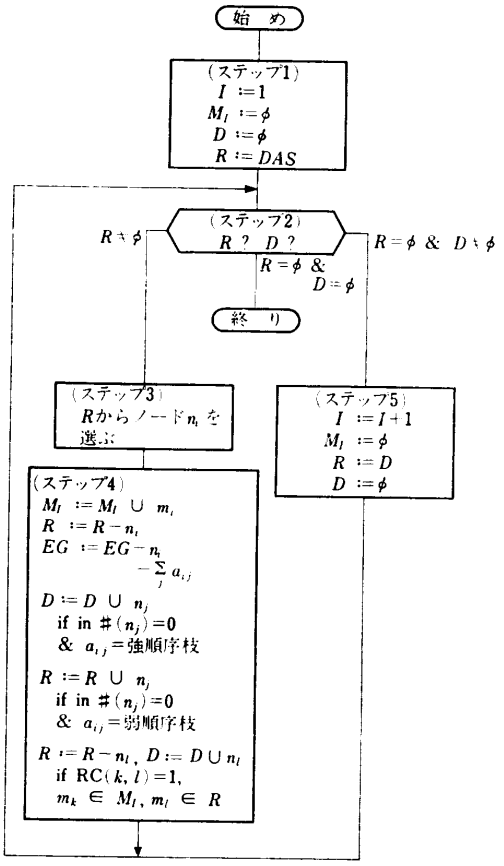

図 5 並列化基本アルゴリズム
Fig. 5 Flow of basic algorithm.

EB 群が決まったときを考える.

前章の解析法を用いて1つの EB について1つの EG を求める. 1つの EG から μI 系列を求める並列化基本アルゴリズムを**図 5** に示す. これは Yau[9] や Wood[10] のアルゴリズムを改良したものである.

(ステップ 1) μI 系列カウンタ I を1にし, M_I を ϕ(空)にする. 節点の集合 D を ϕ にし, EG の DAS を節点の集合 R に代入する.

(ステップ 2) $R=\phi$ でかつ $D=\phi$ のとき終了する. $R=\phi$ でかつ $D\neq\phi$ のときはステップ5へ進む. $R\neq\phi$ のときはステップ3へ進む.

(ステップ 3) R の中から1つ任意の節点 n_i を選び出す*. ステップ4へ進む.

(ステップ 4) ステップ3で選んだ節点 n_i に対応する μOP, m_i を M_I に割り付ける. R から n_i を取り除く. EG から n_i と, n_i を始点とするすべての有効枝 a_{ij} を取り除く.

in#$(n_j)=0$ で, a_{ij} が強順序枝なら n_j を D に加え, 同じく a_{ij} が弱順序枝なら n_j を R に加える.

M_I の中の任意の μOP, m_k と, R内の任意の節点 n_l に対応する μOP, m_l の間にリソース競合がある. つまり

RC $(k, l)=1$

であるならば, n_l を R から D に移す. ステップ2へ進む.

(ステップ 5) I を1つ増加させ, M_I を ϕ にする. D内の節点をすべて R に移し, Dを ϕ にする. ステップ2へ進む.

RとDはどちらも DAS に含まれる節点の集合であるが, Rは現在割付中の μI にさらに割り付けられる可能性のある μOP に対応する節点の集合を表わしている. ステップ5で新しい μI への割り付を開始するときDをRへ移すのは, どの μOP も最初に割り付けることができるからである.

3.2 節点の選択法

基本並列化アルゴリズムのステップ3で, Rの中からどの節点を選ぶかによって結果は異なる. ここでは節点の選択法について述べる.

(1) 最適節点選択法

Rからどの節点を選択すれば最小ステップになるかを解析的に求めることはできない. 1つの方法は全数チェック法で, Rの中のすべての節点を順次選択していき, 理論的最小ステップになるか (Yau[9]), すべての試みのうち最小のものを採る方法である.

この方法を実現するには, 選択の経歴を示すものとバックトラッキングのためのプッシュダウンスタックが必要になる. その処理手続きの一例を**図 6** に示す. 基本並列化アルゴリズムとの相異点を次に示す.

(イ) ステップ1に以下を追加する. 最大ステップ数を記憶する変数Nを0にし, プッシュダウンスタックを ϕ にし, 選択節点集合SにRを代入する.

(ロ) ステップ3を次のように変更する. Sから任意の節点 n_i を選び出し, Sから n_i を取り除く. Sが ϕ でないとき, プッシュダウンスタックに S, D, R, EG, P, I をプッシュする.

(ハ) ステップ4の最後に以下を追加する. SにRを代入する.

(ニ) ステップ2で, RもDも ϕ のときの手続きを次のように変更する. $I>N$ のとき N に I を代入し, P を P' (これが最後に求めるものとなる) に退

* EB の先頭 μOP が合流点のときは最優先に選び, EB 最後の μOP が分岐 μOP のときは最後に選ぶ.

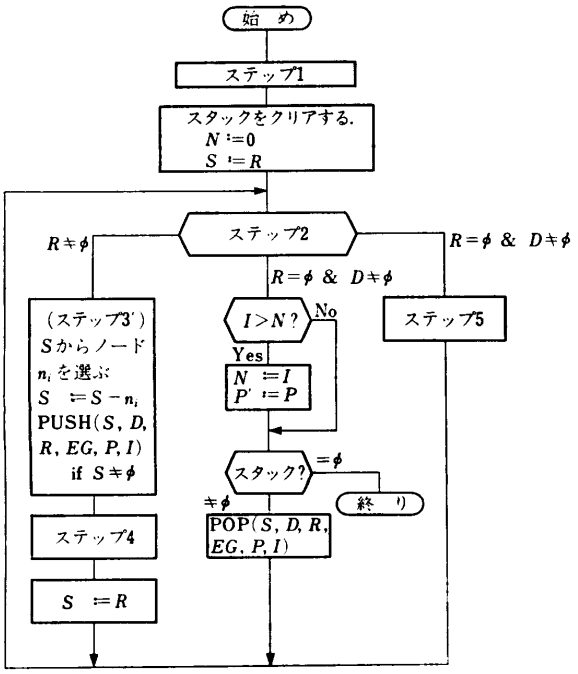

図 6 最適ノード選択アルゴリズム
Fig. 6 Optimal node selection algorithm.

避し，プッシュダウンスタックから S, D, R, EG, P, I をポップし，ステップ2に進む．プッシュダウンスタックが ϕ でポップできないときは終了する．

本方式では，EG の節点数を n としたときの計算量は $O(n!)$ となる．

(2) 重み付け節点選択法

節点の選択をヒューリスティックに行い，手続きを簡単化するもので，Yau[9], Wood[10], Poe[6] などと同様に EG の各節点に重み付けを行い，基本並列化アルゴリズムのステップ3で節点を選択するとき，最も重みの大きいものから選ぶようにする．

EG 内の節点 n_i の重み w_i は次のようにして求められる．

(イ) EG の葉の節点 n_i の重み w_i を1とする．

(ロ) 有向枝 a_{ij} の始点となっている節点 n_i の重み w_i を w_j+1 とする．ただし n_i を始点とする有向枝が2つ以上あるときは，その終点節点の重みが最大のものを選ぶ．

本方式では，$O(n^2)$ の計算となる．しかし，実際のマイクロプログラムの並列化では，DAS の要素数が限りなく増加することがないことが経験からわかっているので，$O(n)$ で計算できると考えられる．

3.3 拡張ブロックの選択

μOP 系列内の EB を決定していくとき，**2.2** に示したように，条件分岐 μOP の分岐先の選び方に自由度があり，それによって並列化したときの μI 系列のステップ数が異なる可能性がある．

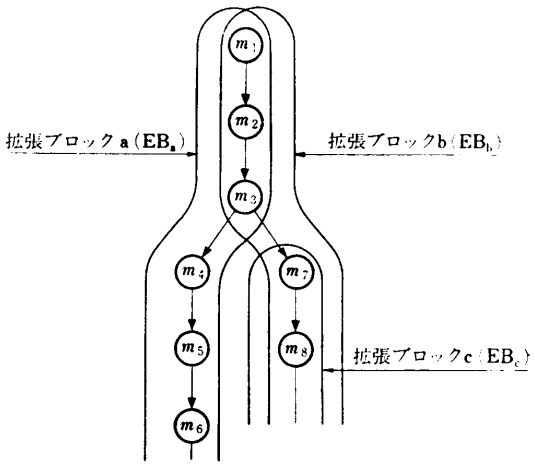

図 7 拡張ブロックの選択
Fig. 7 Selection of extended blocks.

たとえば，**図7**で EB_a と EB_b があるとき，次のような選択法が考えられる．

(1) 最適拡張ブロック選択法

すべての組合わせのうち，最適な場合を求める．つまり，EB_a で並列化し，次に EB_b で並列化した場合と，その逆順で並列化した場合のステップ数を比較して最小の方を選ぶ．

(2) 優先順拡張ブロック選択法

分岐方向に，分岐確率などにより優先順位をつけ，その順に並列化を行う．図7で左方向への分岐確率が高いとすれば，まず EB_a で並列化し，次いで EB_b で並列化する．

(3) 排他的拡張ブロック選択法

分岐方向に，分岐確率などにより優先順位をつけ，その最優先分岐方向のみ EB を形成する．つまり，1つの EB 木を，互いに排他的な EB に分割してしまう．図7では EB_a と EB_c に分割し，それぞれを並列化する．

並列化効果の大きいのは，(1), (2), (3)の順であるが，計算量はその逆順になる．1つの μOP 系列 p 内の EB 木の数は分岐先ラベル（p の入口点を含む）の数に等しく，EB 木の中の EB の数はその木の葉の数 L に等しい．EB 木の中に $r(r>1)$ 方向分岐 μOP が S_r 個含まれていれば，

$$L=1+1\sum_r(r-1)\times S_r$$

と表わすこともできる．(1)の場合，EB の選択順序

表 1 拡張ブロック内の基本ブロック数とマイクロ操作数
Table 1 Number of basic blocks and micro operations in an extended block.

	基本ブロックの数	マイクロ操作の数
基本ブロック	1	4.2
拡張ブロック	2.1 (最大 8)	8.8
排他的拡張ブロック	1.8	8.7

は $L!$ 通りあり，各々に L 個の EB があるから，$L \times L!$ 個の EB 解析と並列化が必要となる．(2)の場合，EB の順序は一通りであるから L 個の EB の解析と並列化が必要となる．(3)の場合も(2)の場合と同様であるが，(2)の場合には1つの並列化が終らないと次の EB の解析ができないのに対して(3)の場合はすべての EB の解析が並列化以前にできるので，手続きがいっそう簡単になる．

4. 適用評価

(1) 拡張ブロックサイズ

並列化の効果を上げるためには EB サイズが大きい方がいい．制御用ミニコンピュータのシステムプログラム用ファームウェアの 15 個のモジュールを調べたところ，1つの EB 中の BB の数は，平均で 2.1 個，規模の大きいモジュールでは，2.3 個から 3.0 個，最大で 8 個であった．μOP の数では，BB 内には平均 4.2 個であるのに対して EB 内には平均で 8.8 個であった．排他的拡張ブロック選択法を用いた場合でも1つの EB 中の BB の数は 1.8 個，EB あたりの μOP 数は 8.7 個で，十分効果があることが確認された (**表 1**)．

(2) 大局的並列化の効果

同じくシステムプログラムの中のタスクディスパッチャ用のファームウェア (130 個の μOP からなる) を並列化したところ，次のようになった．

人手による並列化 (マシンの特性に強く依存した並列化を除く) で 89 μI になった．これを本アルゴリズム(排他的拡張ブロック選択法，重み付け節点選択法)を用いると 95 μI になり，人手の 6.7% 増となった．BB 単位に並列化すると 102 μI で，人手の 14.6% 増であった(**表 2**)．

(3) マイクロプログラミングシステムへの適用

本アルゴリズムを汎用マイクロプログラミングシステムの並列化フェーズに適用した．

各 μOP の変数参照・定義情報 $\mathrm{ref}_i(v)$ と $\mathrm{def}_i(v)$ と μOP 系列の制御フロー情報を入力とし，各 μI 内の μOP 集合の情報を出力する並列化処理プログラムは，FORTRAN で約 2,000 ステップであった．

表 2 拡張ブロックによる並列化効果
Table 2 Effect of extended blocks.

	マイクロ命令数	増分比 (%)
マイクロ操作数	(130)	—
人手による並列化	89	0
排他的拡張ブロック選択法による並列化	95	+6.7
基本ブロック内並列化	102	+14.6

5. あとがき

マイクロプログラムを，基本ブロック間にまたがって自動並列化する実用的な大局的並列化アルゴリズムを提案した．

一般に基本ブロックのサイズは小さく，局所的な並列化では効果が小さい．これまで提案されている大局的な並列化は，マイクロプログラムの構造化を前提としているものが多いが，現実のファームウェア (特にシステムプログラム用のファームウェア) では，性能および容量上の制約から構造化できる状態にない．

本アルゴリズムでは，合流点間のパスからなる拡張ブロックを導入し，拡張ブロック単位に並列化する方法で，並列化効果を従来の2倍以上にできた．

謝辞 本研究に際しご指導いただいた日立製作所システム開発研究所三森定道博士，助言いただいた同所岩本哲夫，同日立研究所平岡良成の両氏，ならびに並列化プログラムの開発に尽力いただいた同大みか工場中西宏明，日立プロセスエンジニアリング太田孝徳の両氏ほか多数の関係者の方々に深く感謝いたします．

参考文献

1) Agerwala, T.: Microprogram Optimization: A Survey, IEEE Trans. Computers, Vol. C-25, No. 10, pp. 962-973 (1976).
2) Landskov, D. et al.: Local Microcode Compaction Techniques, ACM Computing Surveys, Vol. 12, No. 3, pp. 261-294 (1980).
3) Dasgupta, S.: Parallelism In Loop-Free Microprograms, Proceedings of IFIP Congress Information Processing '77, pp. 745-750 (1977).
4) Tokoro, M. et al.: A Technique of Global Optimization of Microprograms, Proceedings of the 11th Annual Workshop on Microprogramming, pp. 41-50 (1978).
5) Wood, G.: Global Optimization of Microprograms Through Modular Control Constructs,

Proceedings of the 12th Annual Workshop on Microprogramming, pp. 1-6 (1979).
6) Poe, M.: Heuristics for the Global Optimization of Microprograms, Proceedings of the 13th Annual Workshop on Microprogramming, pp. 13-22 (1980).
7) Hecht, M. S.: Flow Analysis of Computer Programs, p. 232, Elsevier North-Holland, N. Y., N. Y. (1977).
8) 武末：マイクロプログラムの並列分解法，電信学会論文誌，Vol. J 60-D, No. 4, pp. 243-250 (1977).
9) Yau, S. S., Schowe, A. C. and Tsuchiya, M.: On Storage Optimization of Horizontal Microprograms, Proceedings of the 7th Annual Workshop on Microprogramming, pp. 98-106 (1974).
10) Wood, G.: On the Packing of Microoperations into Micro-instruction Words, Proceedings of the 11th Annual Workshop on Microprogramming, pp. 51-55 (1978).

(昭和56年10月7日受付)
(昭和56年11月18日採録)

付　録　ライブ変数の求め方

変数は本文に定義したものを用いる.

```
procedure live-variables
  set W /*worklist*/
  bit vector B of length r
  /*Step 1a: Initialize live.*/
  live_n:=1^r /*b^r denotes a bit vector of r b's*/
  for i:=1 to n-1 by 1 do live_i:=0^r endfor
  /*Step 1b: Initialize W. */
  B:=¬bdef_n∨bref_n
  for each j∈pred (n) do
      W:=W∪{(j, B)} endfor
  for i:=1 to n-1 by 1 do
      for each j∈pred (i) do
          W:=W∪{(j, bref_i)} endfor
  endfor
  /*Step 2: Process W.*/
  while W≠φ do
      Select and delete any pair (i, B) from W.
      if B∧¬live_i≠0^r then
          live_i:=live_i∨B
          B:=(live_i∧¬bdef_i)∨bref_i
          for each_j∈pred(i) do
              W:=W∪{(j, B)} endfor
      endif
  endwhile
return
```

ここで

¬はビットベクトルの not
∧はビットベクトルの and
∨はビットベクトルの or

を表わす.