

仮想記憶システムにおけるプログラム動作 モデルの一提案†

フィン・トン・ハン^{††} 益 田 隆 司^{†††}

仮想記憶システムでは、プログラムの動作特性がシステムの性能に大きな影響を与えることが知られている。プログラムの動作特性の中で特に重要な性質は、局所参照に関する性質である。従来、オペレーティング・システムの主記憶管理にしても、あるいは、主記憶管理方式の解析に不可欠なプログラム動作のモデル化にしても、プログラムの局所参照特性の把握は、すべて、実行時のアドレス参照列に基づいていた。しかしながら、局所参照に関する重要な情報は、アドレス参照列の中だけでなく、原始プログラム内にも含まれているはずであり、本論文の狙いはこの点にある。本論文では、プログラム動作のモデル化のために、アドレス参照列だけでなく、原始プログラムの構造から得られる情報をあわせて利用する方法を提案する。プログラム実行時の局所参照の性質は、原始プログラム内の繰り返し構造によるものと考えられる。そこで、局所参照特性、すなわち、プログラム実行中に生ずるフェイズとその移動のモデル化を、原始プログラム内の繰り返し構造に対応づけて行う。原始プログラムが高級言語で記述されている場合には、繰り返し構造はコンパイラが検出することができる。本論文の提案は、プログラムの局所参照の性質を従来と異なった視点から定義したものであり、プログラム動作のモデル化に有効であるだけでなく、オペレーティング・システムの主記憶管理にも利用できる可能性を含んでいる。

1. ま え が き

仮想記憶システムのもとで動作するプログラムは、それが使用する仮想アドレス空間の全体を一様に参照するのではなく、参照するのはその一部であり、かつ、参照部分が時間と共に移動するという、いわゆる、局所参照の性質を有していることが知られている。そして、システムの性能は、このプログラム動作の局所参照の程度によって大きな影響を受ける。

仮想記憶システムの性能に影響を与えるほかの1つの要因は、主記憶の管理方式^{1)~3)}であり、ワーキング・セット法、LRU (Least Recently Used) 法をはじめ、これまでに数多くの方法が、提案、実現されている。これら主記憶管理方式の評価を行うには、プログラム動作のモデルを作成することが不可欠であり、上記の局所参照の性質もそのモデルに反映されねばならない。

このような背景から、これまでにいくつものプログラム動作モデルが提案されているが^{8), 12)}、その多くは局所参照の性質、すなわち、プログラムの実行に伴って生ずるフェイズと各フェイズの中で利用されるペー

ジの集合を適切に表現できるものではない。たとえば、しばしば利用される LRU スタック・モデルでは、各時刻での大きさ i の局所参照集合を最近に参照された i 個のページとして定義する。 n 個のページからなるプログラムが存在する場合には、各時刻で n 種類の局所参照集合の候補があるが、いずれがその時点での真の局所参照集合であるかについての情報は提供していない。また、Denning は、局所参照集合とその中にプログラムが留まる時間の対の系列によるモデルを提案したが¹¹⁾、実行時のアドレス参照列が与えられたとき、それをどのようにしてそのような系列に分割するかの手続きを与えていない。

これに対して、Madison 等が提案した BLI (Bounded Locality Interval) モデル⁹⁾は、アドレス参照列を、フェイズとその継続時間に分割するという点ではかのモデルと比較してすぐれたモデルであるが、反面、2. に述べるような欠点を含んでいる。

これまでに提案されているプログラムの局所参照の概念、および、モデルは、プログラム実行中に生成されるアドレス参照列に含まれる情報にのみ基づいている。これは、オペレーティング・システムの主記憶管理方式が、プログラム実行時のアドレス参照特性に関する情報を動的に収集し、それに基づいて、各プログラムの局所参照特性を推定することを試みていることによるものと考えられる。Denning が提案し、現実のシステムでも広く採用されているワーキング・セッ

† A Study of Program Behavior Modeling in Virtual Memory Systems by TONG-HAING FIN (Department of System Science, Tokyo Institute of Technology) and TAKASHI MASUDA (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 東京工業大学総合理工学研究科システム科学専攻

††† 筑波大学電子情報工学系

トは、局所参照集合の推定量の典型的なものである。しかしながら、プログラム動作のモデル化にせよ、主記憶管理方式にせよ、より基本に立ちかえて考えてみると、プログラムの局所参照に関する重要な情報は、アドレス参照列だけでなく、原始プログラムの中にも含まれているはずである。

本論文の目的は、プログラムの局所参照、あるいは、フェイズを構成する要素を明確にし、それに基づいて、現実のプログラム動作を反映した局所参照、あるいは、フェイズの移動モデルを作成するところにある。特に、プログラムの局所参照特性を検出するために、実行時のアドレス参照列だけでなく、原始プログラムの中に含まれる情報を利用する方法を提案する。この方法はまた、新しい局所参照の概念を提案するものであり、主記憶管理の方法に対しても大きな影響を与える可能性を含んでいる。

2.で、これまでのアドレス参照列に基づいたプログラムの動作モデル、特に、BLIモデルの特徴、問題点を述べ、3.で、本論文で提案する原始プログラム内の情報を利用した局所参照モデルについて述べる。

2. アドレス参照列に基づいた局所参照モデル

2.1 プログラムの局所参照特性

プログラムの局所参照の性質は以下のようにまとめることができる。

(i) プログラムが参照するページの集合は、プログラム実行中の大半の時間、プログラム全体の一部分のみである。

(ii) ある特定のページに対する参照の密度は、時間の経過と共に徐々に変化する。

(iii) プログラムの実行に伴って生ずるアドレス参照列の中から2つの部分参照列を取り出してみると、その部分参照列が相互に近くにあるときには、それらの中で参照しているページには一致するものが多いが、その距離が離れると、参照するページの集合は異なったものになってくる可能性が強い。

通常のプログラムにはこのような局所参照の性質が存在するため、その実行に伴って、利用する局所参照集合が移動する傾向がある。このようなプログラムの性質から、Denningは、プログラム実行時の動作を以下のようにモデル化することを提案している¹⁾。

$(L_1, \tau_1), (L_2, \tau_2), \dots, (L_i, \tau_i), \dots$

ここで、 L_i は、プログラム実行時における i 番目の

局所参照集合であり、 τ_i はそのライフタイム、すなわち、 L_i が継続する時間長を表わす。このモデルを実際のプログラム動作のモデル化に利用するには、各 (L_i, τ_i) を定めなければならないが、それらを定める手順については述べていない。

プログラムの実行に伴い、局所参照集合がはっきりしており、かつ、それぞれがかなり長時間使用されるようなプログラムを局所参照特性の良いプログラムと呼ぶ。上記のモデルでいえば、プログラム全体の大きさに比較して個々の L_i の大きさが小さく、 τ_i が長い場合に相当する。局所参照特性の良いプログラムは、仮想記憶システムの上で効率よく動作する。プログラムの再構成により、局所参照特性を改良する方法が提案されている^{4), 6), 7)}。

2.2 BLI モデル⁵⁾

Madison 等は、現実のプログラムの動作特性から、プログラムの実行の各時点での局所参照集合とそのライフタイムを求める BLI モデルを提案した。プログラムの動作特性として、アドレス参照列を与えたとき、そのプログラム動作をモデル化する方法はいくつも提案されているが、それらの中で、BLI モデルは、アドレス参照列から、各時点での局所参照集合とそのライフタイムを求める唯一のモデルである。これを用いれば、2.1 で述べた Denning のモデルで (L_i, τ_i) の系列を求めることも可能である。

BLI モデルは、LRU スタック・モデルを、特に、各時点でのアクティブな局所参照集合の検出という点に焦点をあわせて改良したものである。LRU スタック・モデルでは、 n ページからなるプログラムが存在すると、各時刻で LRU スタックの各深さまでがそれぞれ1つの局所参照集合であり、常に n 個の局所参照集合が存在していると考えることができる。しかしながら、その内のいずれがアクティブな局所参照集合であるかについては何ら特定していない。これに対して、BLI モデルでは、局所参照集合の候補は、LRU スタックの各深さまで各時点で n 種類存在するが、それらの内、アクティブな局所参照集合を以下のようにして決定する。

時刻 t において LRU スタックの上位 i 個の中に含まれる要素に注目し、これらの要素がスタックの上位 i 個の位置を占めた時刻 t_{i1} から、時刻 t までに再びすべての要素が参照されていたとき、時刻 t_{i1} に、LRU スタックの上位 i 個の要素から成るアクティブ・セット A_i が生成されたと呼ぶ。そして、 A_i

がアクティビティ・セットである状態が時刻 t_{i2} まで続いたとすると、BLI を (A_i, τ_i) として定義する。

ここで、 $\tau_i = t_{i2} - t_{i1}$ であり、 τ_i は、アクティビティ・セット A_i のライフタイムを表わしている。このように BLI を定義すると、アドレス参照列から、BLI を図 1 の例に示すように階層的に作成することができる。図 1 を見るとレベル 1 からレベル 3 までの BLI が存在している。特にレベル 1 の BLI は、階層構造の BLI の最上位に位置し、アドレス参照列を最も長い部分列に分割する点からも重要な意味を有する BLI である。

Madison 等は、ALGOL 60 のプログラム例で、データの参照列に基づいて BLI を求め、その結果から、時間的に長いレベル 1 の隣りあった BLI の中で参照されるセグメント (Madison 等の実験では、B 5500 システムを利用しており、LRU スタックの各要素は論理的な情報の単位であるセグメントから成っている) には、共通のものが少ないことを確かめている。

Madison 等の結果は、データ・セグメント、特に、配列を対象にして得られたものであるが、同様の解析を命令参照列に対して行った場合の結果は、データの場合の結果とかなり異なる可能性がある。

BLI モデルは、アドレス参照列からフェイズとそのライフタイムを検出するためのすぐれたモデルであるが、以下のような問題点も含んでいる。

(i) アドレス参照列から BLI モデルを用いて、フェイズとその遷移を作成した場合、結果が直感と一致しない場合がある。1つの例は、ある BLI 内のすべての要素が、BLI の最初の方で参照され、その後はその中の一部の要素のみが参照され続けたような場合である。たとえば、図 1 で、時刻 30 以降、セグメント D のみが参照され続けたとしても、セグメント A, B, C もまた、レベル 1 の BLI に留まることになる。

(ii) プログラムがあるセグメントの集合を繰り返し参照して定常的に動作しているとき、その途中で 1 つでもその集合に属さないセグメントを参照すると、その時点で BLI は終了する。しかしながら、このような場合には、プログラム実行中のフェイズは移動しないとした方がよいと考えられる場合が多い。

(iii) プログラムの局所参照の性質は、原始プログラム内の繰り返しの構造によるものと考えられるが、

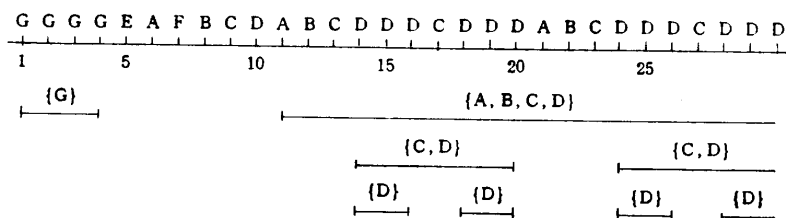


図 1 BLI の階層構造

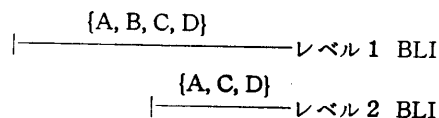
Fig. 1 Hierarchy of Bounded Locality Intervals.

BLI と原始プログラム内のループ構造とは必ずしも対応していない。下記のプログラム例を考えてみる。

```
for i:=1 to 100 do
begin
  A[i]:=B[i]*C[i];
  D[i]:=B[i]*2
end;
```

```
for i:=1 to 100 do
  C[i]:=A[i]-D[i];
```

このプログラムを実行したとき生成される BLI は下記のようなになる。



原始プログラム内で、{A, C, D} からなる BLI に対応したプログラムが入れ子になっているわけではない。原始プログラムの構造からすると、{A, B, C, D}, {A, C, D} を別々のフェイズとして考えるのが自然である。

3. 原始プログラムの構造に基づいた局所参照モデル

これまで仮想記憶システムの主記憶管理の方法が数多く提案されているが、それらの方法に共通していることは、オペレーティング・システムが主記憶管理のために利用する情報は、プログラム実行時の動作から抽出できる情報にのみ限っていたことである。アドレス参照に関する LRU 的な性質、あるいは、局所参照集合の推定量としてのワーキング・セットの有効性など、すべて、プログラム実行時のアドレス参照特性に基づいたものである。

これに対応して、前節で論じた BLI モデルを含めて、これまでに提案されているプログラム動作モデルもまた、実行時のアドレス参照列に基づいている。しかしながら、我々の興味の対象は、プログラムの動作特性自身であり、そのアドレス参照列ではない。アドレス参照列が作り出される過程は、原始プログラムの

中に含まれているはずであり、アドレス参照列において重要と考えられる情報は、原始プログラムの構造からも推察できる可能性がある。

我々が提案する方法の基本はこの点にある。特に、本論文では、プログラム動作のモデル化に焦点をあわせ、そのモデル化のために、アドレス参照列だけを利用するのではなく、それに加えて、原始プログラムの中に含まれる情報を利用することを試みる。

3.1 フェイズの階層表現

図2は、Gaussの消去法をPASCAL言語で書いた例である¹³⁾。プログラムの左側に各ループの範囲を示すための括弧をつけてある。図3は、図2のプログラム内の各ループによって作り出される命令（ここでは文番号）、および、配列の参照の部分列の階層構造を示したものである。これら各々の部分列は、その長さが十分に長ければ、局所参照によるフェイズの概念を満足するものと考えられる。しかしながら、「十分に長い」、あるいは、「あるフェイズがほかのフェイズよりも重要である」とかは、プログラム実行の環境によって定まることである。純粋にプログラム動作をモデル化するには、プログラムのフェイズを階層的に表現しておくのがよい。それによって、特定のシステム向きでない一般的なモデル化が可能である。

2.2 で述べた BLI モデルは、階層モデルの例である。

図3(a)の階層表現では、各部分列は原始プログラム内の1つのループ構造に対応している。データの参照に関して、基本的には各ループに対応して1つの部分列ができるが、ループが入子構造になっていると、外側と内側のループで同一のデータを参照しているときには、その部分列は重なりあう。図3(b)はそのような場合の例を含んでいる。 $L_{1.1}$, $L_{1.2}$ によって参照される配列は同一であり、上に述べた部分列は重なることになる。モデルの作成に際しては、この点を考慮しておく必要がある。

3.2 原始プログラム内の同時利用領域の概念

実行時に作り出される命令あるいはデータの参照列におけるフェイズと原始プログラムの構造との関係について考察する。原始プログラム内には、実行時の命令またはデータの参照列における各フェイズに対応した領域があるはずである。そして、原始プログラムは、その実行順序に関して互いに作用を及ぼしあう複

```

var i, j, k: 1.. n;
    p, t : real;
    A : array [1.. n, 1.. n] of real;
    B, X: array [1.. n] of real;

begin
  for k:=1 to n do
    begin
      p:=1.0/A[k,k];
      L1.1 { for j:=k+1 to n do
        A [k,j]:=p * A[k,j];
        B[k]:=p * B[k];
      L1.2 { for i:=k+1 to n do
        begin
          L1.2.1 { for j:=k+1 to n do
            A[i,j]:=A[i,j]-A[i,k] * A[k,j];
            B[i]:=B[i]-A[i,k] * B[k];
          end;
        end;
      end;
    end;
  k:=n;
  L2 { repeat
    t:=B[k];
    L2.1 { for j:=k+1 to n do
      t:=t-A[k,j] * X[j];
      X[k]:=t; k:=k-1;
    until k=0;
  end.

```

図2 プログラム例

Fig. 2 Sample Program.

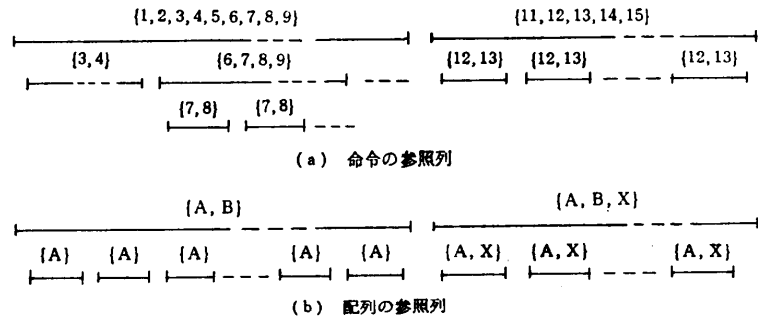


図3 命令および配列の参照列

Fig. 3 Instruction and array reference strings.

数個のこのような領域から成っていると考えることができる。プログラムの実行によって利用される領域の種類は、入力データに応じて異なる可能性がある。また、ある領域が一度参照されると、かなりの時間その領域が継続して参照される可能性がある。原始プログラム内のこのような領域を同時利用領域と呼ぶことにする。図2の例で同時利用領域を考えてみる。図3(a)の命令参照に関する各部分列をそれぞれ1つのフェイズと考えてよいとすると、原始プログラムの各ループ内に存在する命令およびそれによって参照されるデータの集合が同時利用領域を形成する。図4(a)はこのようにして作られた同時利用領域の間の関係を示したものである。図3(b)の配列の参照列において、その各部分列を1つのフェイズと考えることがで

ければ、配列の参照を基礎にした同時利用領域は、図4(b)に示すようになる。 L_1 , $L_{1.2}$ の2つのループ内で参照する配列は同一であるので、 L_1 と $L_{1.2}$ の中は1つの同時利用領域と考えるのが自然である。このように、データの参照を基礎にして同時利用領域を作成した場合には、必ずしも、原始プログラム内の1つのループが1つの同時利用領域に対応するとは限らない。

原始プログラム内での同時利用領域は、プログラムを実行したときの命令あるいはデータのアドレス参照列と原始プログラムの構造から局所参照集合をどのように定義するかによって定まる。逆に、原始プログラム内の同時利用領域の構造と実行時の参照列が定まれば、その参照列に含まれる局所参照集合とそのライフタイムを正確に求めることができる。

そこで、次に、原始プログラムの構造内で、同時利用領域をどのように定義すべきかについて考察する。

3.3 同時利用領域モデル

プログラム内部の命令の実行順序を表現するために、高水準制御フローグラフ技法^{9),10)}に似た方法を利用することにする。通常のフローグラフでは、プログラムの構文的な構造は表現されず、制御フローに関する構造としては、分岐、条件付分岐のみがある。これに対して、制御フローグラフ技法では、プログラムの構文構造を直接に取り扱い、begin-end, do, repeat, for などの高水準制御構造をすべて利用する。本論文の目的に関連する制御構造は、局所参照に関連して、命令あるいはデータの集合を繰り返し参照するようなものである。

プログラムの実行の流れを表現するために、以下のようなブロックを利用する。

(i) 順次ブロック(図5(a)): 逐次実行される文またはブロックから成るブロック

(ii) 選択ブロック(図5(b)): if-then-else に相当する文を含むブロック

(iii) 繰り返しブロック(図5(c)): 2回以上繰り返し実行される可能性がある文またはブロックから成るブロック。このブロックは, do, while, repeat などのループを構成する文、あるいは、再帰手続きなどによって生成される。

図6に、単純なプログラムとそれを高水準制御フローグラフで表現した例を示す。現実のプログラムでは、その制御構造はかなり複雑になると考えられる。

局所参照の性質が生ずるためには、プログラムのあ

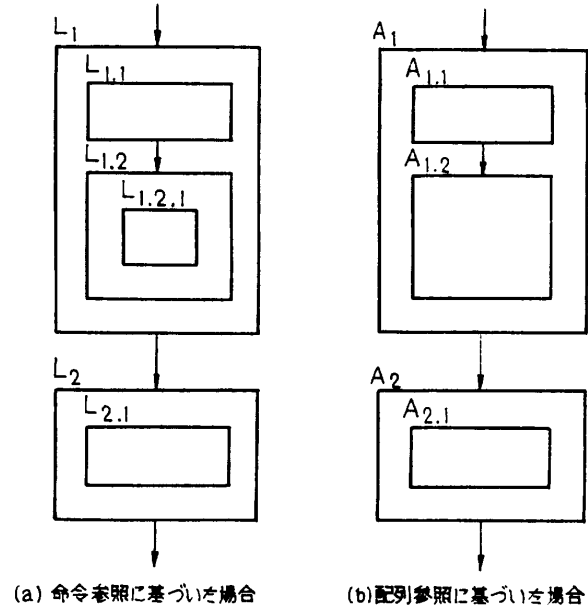


図4 原始プログラム内の同時利用領域の構造

Fig. 4 Locality areas in source program.

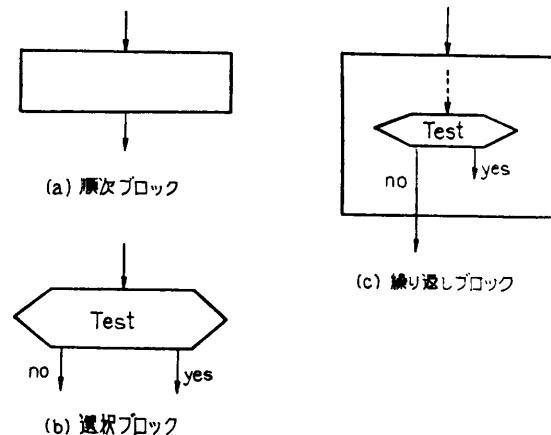


図5 フローグラフを表現するための基本ブロック

Fig. 5 Blocks which are used to represent flowgraph.

る部分が繰り返し実行されなければならない。繰り返しのないプログラム部分の実行は短時間で終了するはずであるので、そのような部分は、局所参照の点からは考慮する必要がない。また、大きなデータ領域を占める配列についても、それらは一般に、ループの中でアクセスされることが多い。これらの理由から、上記の繰り返しブロックを用いて、3.2で述べた同時利用領域を以下のようなステップで求める。

〈ステップ1〉: 前述の3種類のブロックを用いて、原始プログラムから高水準制御フローグラフを作成する。グラフの各ノードに文番号とそこで参照される

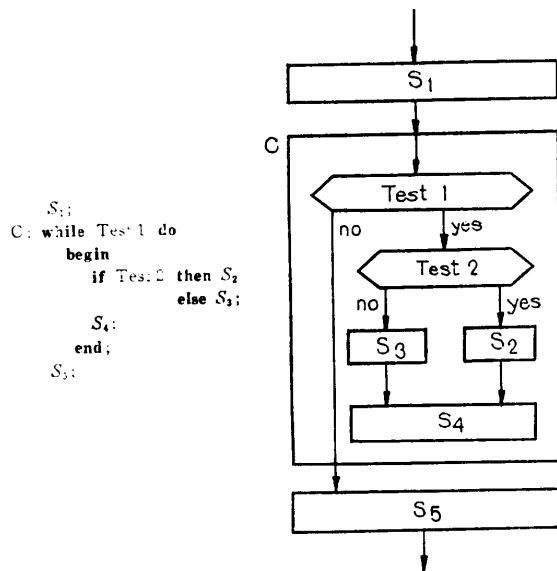


図 6 高水準制御フローグラフの例

Fig. 6 Sample program and its high-level flowgraph.

データを対応づける。このグラフを基本フローグラフと呼ぶ。

〈ステップ2〉: 基本フローグラフの各ノードにある文番号, データをセグメント番号に変換する。ここで, セグメントとは, 原始プログラム内の文またはデータのある論理的なまとまり, または, ページのような物理的な情報の単位に対して割り当てられたものである。このようにして, セグメントに関する順次ブロック, 選択ブロック, 繰り返しブロックから成るセグメント・フローグラフを作成する。

〈ステップ3〉: セグメントに関する繰り返しブロック C 内に存在するセグメントの集合を Sc で表わす。繰り返しブロック C_2 が C_1 に入れ子になっていて, かつ, $Sc_1 = Sc_2$ ならば, C_2 をグラフから消去する。また, C_2 が C_1 の後にきて, かつ, $Sc_1 = Sc_2$ のときには, C_1 と C_2 をあわせて, 拡張された繰り返しブロック $C = C_1 \cup C_2$ を作成する。

以上の手続きのあと, セグメント・フローグラフの中に残っている繰り返しブロックが, セグメントに関する同時利用領域の概念を満足するものである。

図2のプログラムを再利用して, 上記の手続きを説明する。図2のプログラムから求めた基本フローグラフを図7に示す。文, データとセグメントの対応づけを下に示すようにすると, セグメント・フローグラフは図8のようになる。

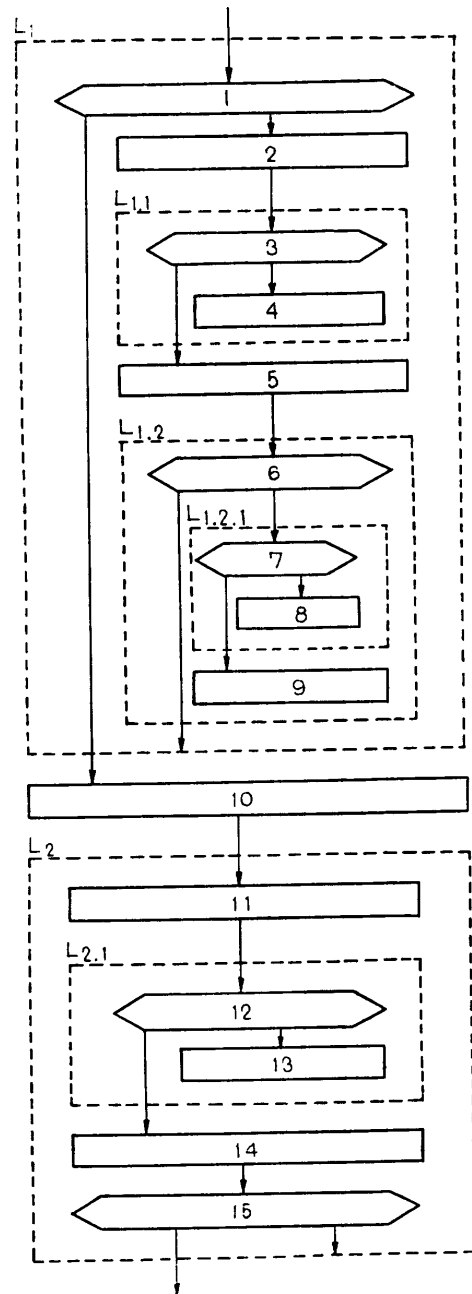


図 7 図2のプログラム例の基本フローグラフ表現
Fig. 7 Basic flowgraph of program in Fig. 2.

セグメント番号	要素
1	文 $S1 \sim S15$
2	変数 i, j, k, n, p, t
3	配列 A
4	配列 B
5	配列 C

図8のセグメント・フローグラフに〈ステップ3〉を

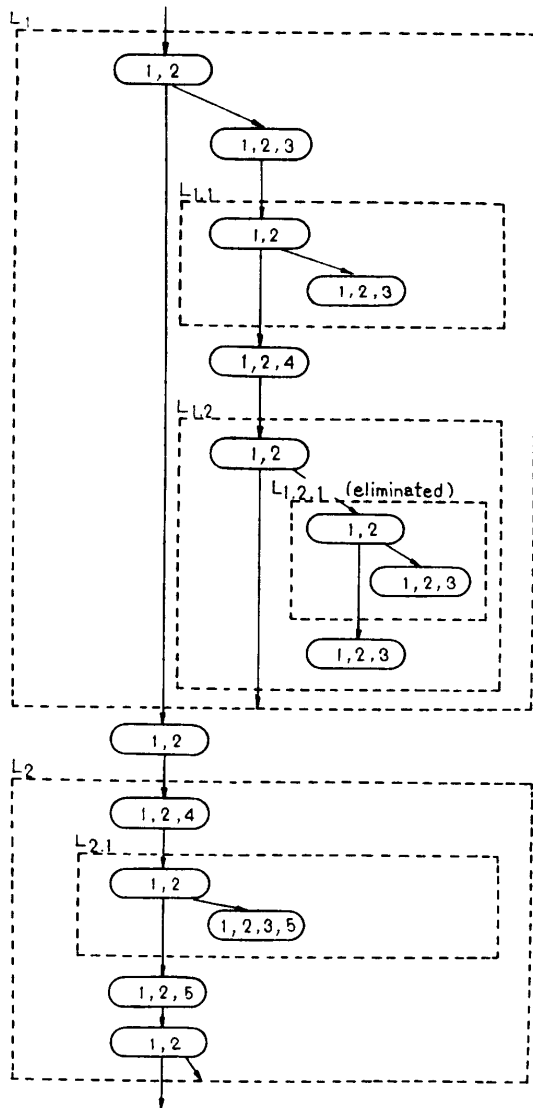


図 8 図 7 のセグメント・フローグラフ
Fig. 8 Segment flowgraph of Fig. 7.

適用すると、繰り返しブロック $L_{1.2.1}$ は消去される。

このようにして求めた繰り返しブロックとセグメント参照列を関連づけることにより、局所参照特性のモデル化が可能になる。具体的には、プログラムの実行に伴って生ずる各フェイズに対して、それを構成するセグメントの集合、および、そのライフタイムを求めることができる。

時刻 t で参照されたセグメントが、繰り返しブロック C に含まれているとき、 C は時刻 t でアクティブであるという。繰り返しブロック C がアクティブであるとき、 C を入れ子にしているすべての繰り返しブロッ

クもまたアクティブになる。

セグメント・フローグラフ内の繰り返しブロックは、セグメント参照列の中のフェイズの開始点と終了点を定義するために利用される。繰り返しブロック C に時刻 t_1 で入り、時刻 t_2 でそこから抜け出るならば、参照列内の部分列 $r_{i_1}, \dots, r_{i_2}$ が 1 つのフェイズとして定義される。その間のプログラム動作は、 (S_c, t_1, t_2) としてモデル化される。ここで、 S_c は、時刻 t_1 から時刻 t_2 までに繰り返しブロック C 内で参照されたセグメントの集合であり、 t_1 は、このフェイズの開始時刻 t_1 、 t_2 は、フェイズの継続時間 $t_2 - t_1$ である。

本論文で提案した繰り返しブロックに基づいた局所参照モデルは、先に述べた BLI モデルの問題点を解決している。ここでは問題点 (ii) についてのみ、簡単に述べる。

プログラムがいくつかのセグメントを参照して定常的にある BLI の中で動作しているとき、その途中で一時的に 1 つでも新しいセグメントが参照されると、その時点で BLI は終了してしまう。これに対して、本論文のモデルでは、その新しいセグメントが同一の繰り返しブロックの中に存在していれば、それが参照された時点でフェイズが終了することはない。図 9 はこの例を示したものである。図 9 (a) は、セグメント・フローグラフを表わし、図 9 (b) は、そこから作り出されたセグメント参照列の例とその参照列に基づいたフェイズの構成を示している。繰り返しブロックに基づいた局所参照モデルの場合には、定義から、フェイズは時刻 2 に始まり、時刻 17 で終了する。時刻 12 に新しいセグメント D が参照されてもフェイズは終了しない。BLI モデルの場合には、時刻 12 でフェイズが終了するという不自然な結果を生ずる。

最後に、本論文で提案した局所参照モデルの特徴をまとめると以下ようになる。

(i) 本モデルは、原始プログラム内の構造、特に、原始プログラム内の繰り返し構造を局所参照特性の把握に利用したモデルである。

(ii) したがって、本モデルから得られる結果は、直観的な局所参照の概念にきわめてよく一致している。また、本モデルは、BLI モデルの持つ問題点の多くを解決している。

(iii) 本モデルから得られる結果は、プログラムの動作特性のみに依存し、アドレス参照列を生成したシステムの構成、パラメータなどには依存しない。

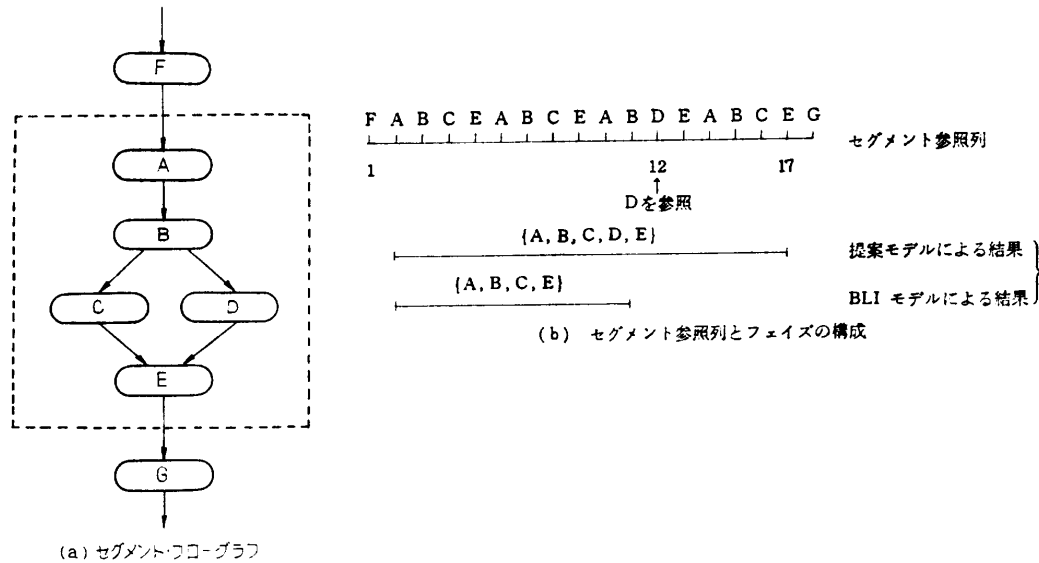


図 9 提案モデルとBLI モデルの比較例

Fig. 9 Comparison of proposed model with BLI model.

4. む す び

本論文では、プログラム動作のモデル化のために、従来のように実行時のアドレス参照列を利用するだけでなく、原始プログラム内の情報をあわせて利用する方法を提案した。提案した方法では、実行時の局所参照特性は、原始プログラム内の繰り返し構造によるという性質を利用した。このような構造は、コンパイラが原始プログラムをコンパイルしたときに検出することが可能であり、現在、本方式の有効性を確かめるために、PASCAL コンパイラを例にして、コンパイラが繰り返しブロックを検出したときに、その入口と出口で特別な目的コードを生成するようにコンパイラに手を加えて実験を行うことを計画中である。

プログラム動作のモデル化だけでなく、提案した方法のほかの1つの可能性は、オペレーティング・システムの主記憶管理に適用することである。ワーキング・セット法、LRU 法など、従来の主記憶管理方式では、主記憶管理を行うために利用する情報は、すべて、過去のアドレス参照特性に関する情報のみである。オペレーティング・システムが、プログラムの局所参照特性を検出するために、原始プログラム内の情報を利用することができれば、その影響は非常に大きいと考えられる。基本的には、本論文で述べた繰り返しブロックに関する情報を、プログラム実行時にオペレーティング・システムに提供し、オペレーティング・システムがそれをプログラムの局所参照特性として利

用することである。このような方法の有効性を検討することが今一つの検討課題である。

最後に、本論文の両著者が協力して研究できるような機会を与えていただいた東京工業大学情報科学科森村英典教授に厚く感謝致します。また、1980 年秋 IBM 天城シンポジウムに來日された折に本方式の有効性について討論していただき、貴重なご意見を賜った IBM ワトソン研究所の L. A. Belady 博士に心から感謝致します。

参 考 文 献

- 1) Belady, L. A.: A Study of Replacement Algorithms for a virtual Storage Computer, IBM Sys. J. Vol. 5, No. 2, pp. 78-101 (1966).
- 2) Denning, P. J.: The Working Set Model for Program Behavior, CACM, Vol. 11, No. 5, pp. 323-333 (1968).
- 3) Denning, P. J.: Working Sets Past and Present, IEEE Trans. Software Engineering, Vol. SE-6, No. 1, pp. 64-84 (1980).
- 4) Ferrari, D.: Improving Locality by Critical Working Sets, CACM, Vol. 17, No. 11, pp. 614-620 (1974).
- 5) Madison, A. W. and Batson, A. P.: Characteristics of Program Localities, CACM, Vol. 19, No. 5, pp. 285-294 (1976).
- 6) 益田, 塩田: 仮想メモリスシステム向けの最適プログラム構成方式と実験, 情報処理, Vol. 15, No. 9, pp. 662-669 (1974).
- 7) Masuda, T.: Methods for the Measurement of Memory Utilization and the Improvement

- of Program Locality, IEEE Trans. Software Engineering, Vol. SE-5, No. 6, pp. 618-631 (1979).
- 8) Masuda, T. and Fin, T.H.: Program Behavior and Its Models, Proc. 14th IBM Computer Science Symposium (Operating Systems Engineering), pp. 88-113 (1980).
- 9) Nassi, I. and Shneiderman, B.: Flowchart Techniques for Structured Programming, ACM SIGPLAN Notices, Vol. 8, No. 8, pp. 12-26 (1973).
- 10) Rosen, B. K.: High-Level Data Flow Analysis, CACM, Vol. 20, No. 10, pp. 712-724 (1977).
- 11) Spirn, J. R. and Denning, P. J.: Experiments with Program Locality, Proc. 1972 FJCC, pp. 611-621 (1972).
- 12) Spirn, J. R.: Program Behavior: Models and Measurements, Elsevier North-Holland (1977).
- 13) Wirth, N.: Systematic Programming, Prentice-Hall, Englewood Cliffs, N. J. (1973).
- (昭和56年1月30日受付)
- (昭和56年4月27日採録)