第４章 ジェットパイプライン・アーキテクチャの性能評価

§4.1 緒言

本章では、第３章で提案したジェットパイプライン・アーキテクチャの性能評価を行う。新たに提案した計算機アーキテクチャの性能を評価するためには、実際のハードウエアを作成するのは困難であり、またその困難なハードウエア製作作業以前にあらかじめその性能を評価して欠点を洗い出しておく必要もあるため、既存の計算機上にソフトウエアによりシミュレータを構築して性能評価を行うことが一般的である[Hironaka 94][Mitchell 88]。また､シミュレータ上で実行されるベンチマークプログラムも、各種のものが存在する[Weicker 90]ので、それらのベンチマークプログラムの性質をよく把握し、目的にあったベンチマークを選択し実行する必要がある。ただし、実在の計算機の評価を目的としたベンチマークの実行時とは異なり、今回のような新規のアーキテクチャの評価を行う場合、コンパイラ等のソフトウエア環境が充実していないために、複雑なベンチマークプログラムの実行が不可能なこともある。

本章ではまず、ジェットパイプライン・アーキテクチャのシミュレーションのためのモデルとそのパラメータを示し､実際に作成したシミュレータについて述べる。さらにシミュレーションで使用したリバモアループとスタンフォード(Stanford)ベンチマークの２種類のベンチマークプログラムについてその概要を示し、ベンチマークプログラムを実行する際のソフトウエア環境についても述べる。最後に、シミュレーション結果を示し、それに基づいて検討を行う。

§4.2 シミュレーションモデルおよびパラメータ

ジェットパイプライン・アーキテクチャの性能評価のためにシミュレータを構築した。シミュレーションを行う機能レベルには、たとえばゲートレベルまで追跡するものから、レジスタトランスファレベルで評価するもの、あるいは通常の計算機上での実行トレースとターゲットアーキテクチャの実行パラメータを併用して評価を行うもの[Mitchell 88]まで、幅広く存在しているが、ここでは§3.4で述べた機能ブロックレベルを想定し,命令パイプラインを4本持つシステムについてシミュレー

タを構築した。ただし、今回構築したシミュレータでは、ジェットパイプライン・アーキテクチャの特徴的な演算部の構成とその制御方法によって得られる性能の評価を目標としているため、以下のような仮定を行っている。

(1)レジスタは§3.4において述べたように、オーバラップしたバンク分けレジスタ方式を仮定してある。レジスタアクセスポートと結合ネットワークの制限を考慮し、レジスタファイルへの同時並列アクセスは同一バンクに対して最大2までに制限した。

(2)メモリは1クロックでアクセス可能とし、さらに並列アクセスも制限を受けず行えるものとする。

(3)演算パイプラインは命令パイプラインよりも各ステージあたりの処理が限定されているため、命令パイプラインの2倍のクロックで動作するとし、加減算、乗算、除算をそれぞれ3,5,7演算パイプラインクロックの遅延で処理するとした。これらの演算遅延クロック数は、ある命令レベル並列処理システムのシミュレーションによる評価例[Hironaka 94]においても類似した値が使用されており、妥当であると考えられる。

§4.3　ジェットパイプラインシミュレータ

本研究において性能評価のために構築されたジェットパイプラインシミュレータはC言語により記述され、UNIXオペレーティングシステムが稼働しているワークステーション上で実行可能である。

シミュレータはジェットパイプラインの機械語コードをロードしてそれを実行するが、ジェットパイプラインアセンブラプログラムを機械語オブジェクトコードに変換するアセンブラを開発した。アセンブラは2パスで、1パス目にジャンプ命令のターゲットアドレスやデータのアドレスを示すラベルの値を決定し、それに基づいて2パス目で実際のオブジェクトコードを出力する。ただし現段階ではアブソリュートアセンブラであり、リンクローダに対応した機能は持たない。

機械語プログラムは高級言語と比較して非常に低レベルであり、かつジェットパイプライン・アーキテクチャはRISCアーキテクチャベースのため、人間が機械語を用いてプログラミングするとバグが生じがちである。現在ジェットパイプライン・アーキテクチャ用のコンパイラシステムを開発中である。このようなことから、シ

ミュレータにはデバッグのための機能として、各シミュレーションサイクルごとにステップ実行する機能や、プログラム中にブレークポイントを設定してそこで一旦停止させる機能がある。

シミュレータはUNIXオペレーティングシステム上のcursesライブラリを用いたキャラクタベースのユーザインタフェースを備えており、キー操作を用いて上述したデバッグ機能を使用したり、画面上にシミュレータ内部の命令パイプラインおよび各ユニットの内部状態を表示させながら機械語プログラムを実行することができる。また、スカラレジスタ、ベクトルレジスタ、プログラムおよびデータメモリの内容も表示可能で、さらにこの３つのうちどれか一つを表示させながらの実行も可能である。実行中の画面表示の例を図4-1に示す。

また、多くのプログラムを用いて性能評価を行う際に、内部状態をトレースしながら実行したのではその実行に時間がかかるため、状態表示をオフすることが可能である。また、デバッグ済みのプログラムを自動実行し結果（実行サイクル数）のみを表示するバッチモードも処理の自動化のために導入されている。

§4.4　ベンチマークプログラム

シミュレータを構築したのち、実際にその上でベンチマークプログラムを実行し、本システムの性能評価を行った。

性能評価を行うためのベンチマークプログラムとして、スーパーコンピュータの性能評価に用いられるリバモアループ[McMahon 88]と、汎用計算機向けのベンチマークプログラムであるスタンフォードベンチマーク[Weicker 90]を採用した。リバモアループは米国のローレンス・リバモア研究所が計算機の数値計算に対する性能評価を行うために、代表的な実用プログラムから核となるループを抜き出して作成したもので、高度にベクトル化可能なものからベクトル化が困難なものまで各種のループから構成されている。リバモアループはベクトルアーキテクチャのベンチマーク[Karaki 84][Imori 93]ばかりでなく、命令レベル並列処理計算機の評価にも使用されている例がある[Kainaga 93][Hironaka 94]。また、スタンフォードベンチマークは、スタンフォード大にてRISC CPUの開発を行った際に、CISCとRISCの比較に用いられた[Weicker 90]もので、８つの整数プログラム(Permutations, Towers of Hanoi, Eight Queens, Integer Matrix Multiplication, Puzzle, Quicksort, Bubble Sort, Tree

```
COMMAND: status
TRACE ON, STEP OFF, wait time 100 msec.


cycle       61 : phase 0 : PC = 0x00000030(      48): write_collision 0
Flags: ERR HLT SF0 ZF0 SF1 ZF1 SF2 ZF2 SF3 ZF3
        0   0   0   0   0   0   0   0   0   0

Scalar Registers:
SR000:               0(       0.0000000)  SR001:              48(       0.0000000)
SR002:               0(       0.0000000)  SR003:               0(       0.0000000)
SR004:             256(       0.0000000)  SR005:             512(       0.0000000)
SR006:             768(       0.0000000)  SR007:               0(       0.0000000)
SR008:               0(       0.0000000)  SR009:               0(       0.0000000)
SR010:               0(       0.0000000)  SR011:               0(       0.0000000)
SR012:               0(       0.0000000)  SR013:               0(       0.0000000)
SR014:               0(       0.0000000)  SR015:               0(       0.0000000)
SR016:               0(       0.0000000)  SR017:      1073741824(       2.0000000)
SR018:      1077936128(       3.0000000)  SR019:      1082130432(       4.0000000)
VMR1: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
      11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111

ip0: IF = Addr.ALU0:in1=0x30, in2=0x4, output=0x34
ip0: ID = decoded:addr=0x2C:FLOATCALC-MLT src1=Reg[23]src2=Reg[19]dstn=Reg[25]
ip0: ALU = set 791 to alu out latch
ip0: MEM = no operation
ip0: WB = no operation
ip0: FWB = no operation

ip1: IF = Addr.ALU1:in1=0x30, in2=0x4, output=0x34
ip1: ID = decoded:addr=0x2D:FLOATCALC-MLT src1=Reg[53]src2=Reg[49]dstn=Reg[55]
ip1: ALU = set 792 to alu out latch
ip1: MEM = no operation
ip1: WB = no operation
ip1: FWB = no operation

ip2: IF = Addr.ALU2:in1=0x30, in2=0x4, output=0x34
ip2: ID = decoded:addr=0x2E:FLOATCALC-MLT src1=Reg[83]src2=Reg[79]dstn=Reg[85]
ip2: ALU = set 793 to alu out latch
ip2: MEM = no operation
ip2: WB = no operation
ip2: FWB = no operation

ip3: IF = Addr.ALU3:in1=0x30, in2=0x4, output=0x34
ip3: ID = decoded:addr=0x2F:FLOATCALC-MLT src1=Reg[113]src2=Reg[109]dstn=Reg[115]
ip3: ALU = set 794 to alu out latch
ip3: MEM = no operation
ip3: WB = no operation
ip3: FWB = no operation

fasp0: (no operation)=   0.00   0.00   0.00 ->  0.00
fasp1: (no operation)=   0.00   0.00   0.00 ->  0.00
fmp0: (no operation)=   0.00   0.00  27.00  24.00   0.00 ->  0.00
fmp1: (no operation)=   0.00   0.00  33.00  30.00   0.00 ->  0.00
fdp0: (no operation)=   0.00   0.00   0.00   0.00   0.00 ->  0.00
fdp1: (no operation)=   0.00   0.00   0.00   0.00   0.00 ->  0.00
```

図4-1　ジェットパイプライン・シミュレータの動作画面

Sort)および２つの浮動小数点プログラム(Floating-point Matrix Multiplication, Fast Fourier Transformation)を含んでいる。リバモアループが単純なループのみからなるプログラムであるのに対し、スタンフォードベンチマークは規模は小さいながらも一般的なプログラムにより近いものとなっているため、一般的な応用プログラムの例として採用した。また、スタンフォードベンチマークには関数呼び出しや再帰も含まれている。

今回はリバモアループの中から、ベクトル化及び並列化が容易なものからベクトル化が不可能でループの並列化も困難なものまで、５つのループを選び、実行に要するシミュレーションクロックサイクル数を計測した。また、スタンフォードベンチマークからもいくつか選び、シミュレータ上で実行してクロックサイクル数を計測した。

スカラ命令のみを使用する場合、Fortranで記述されているリバモアループのソースプログラムは、C言語に書き直した後に、またC言語で記述されているスタンフォードベンチマークは結果表示の部分などを取り除いた後に、それぞれGNU Cコンパイラにより SPARC命令セット[SPARC 92]にコンパイルされる。この機械語コードをジェットパイプラインアセンブリコードに変換した後、ディスパッチスタック法およびソフトウエアパイプライニング法を用いて、並列化を行う。また、比較の対象として、並列化を行わないコードも用意する。並列化を行った後に、手作業にてロードや浮動小数点演算命令後の遅延スロットに関するスケジューリングのチューニングを行った後、アセンブラによりオブジェクトコードに変換される。このオブジェクトコードがシミュレータ上で実行されることになる。シミュレーション実行までのプロセスを図4-2に、実行したリバモアループベンチマークプログラムを図4-3に、スタンフォードベンチマークプログラムの一つ(N_Queens)を図4-4にそれぞれ示す。なお、スタンフォードベンチマークにはメモリの動的割り当てを行うmallocライブラリ関数を使用しているものがあるが、ジェットパイプライン・アーキテクチャシミュレータ上ではライブラリ関数等は存在しないため、それと同等の関数を作成しソースプログラム中に挿入してコンパイル、実行した。

一方、スカラ命令とベクトル命令を併用して実行する場合には、ベクトル化を行わなければならないため上記のような方法を採ることはできず、ソースプログラムから直接ハンドコンパイルによりベクトル命令を用いたアセンブリコードに変換

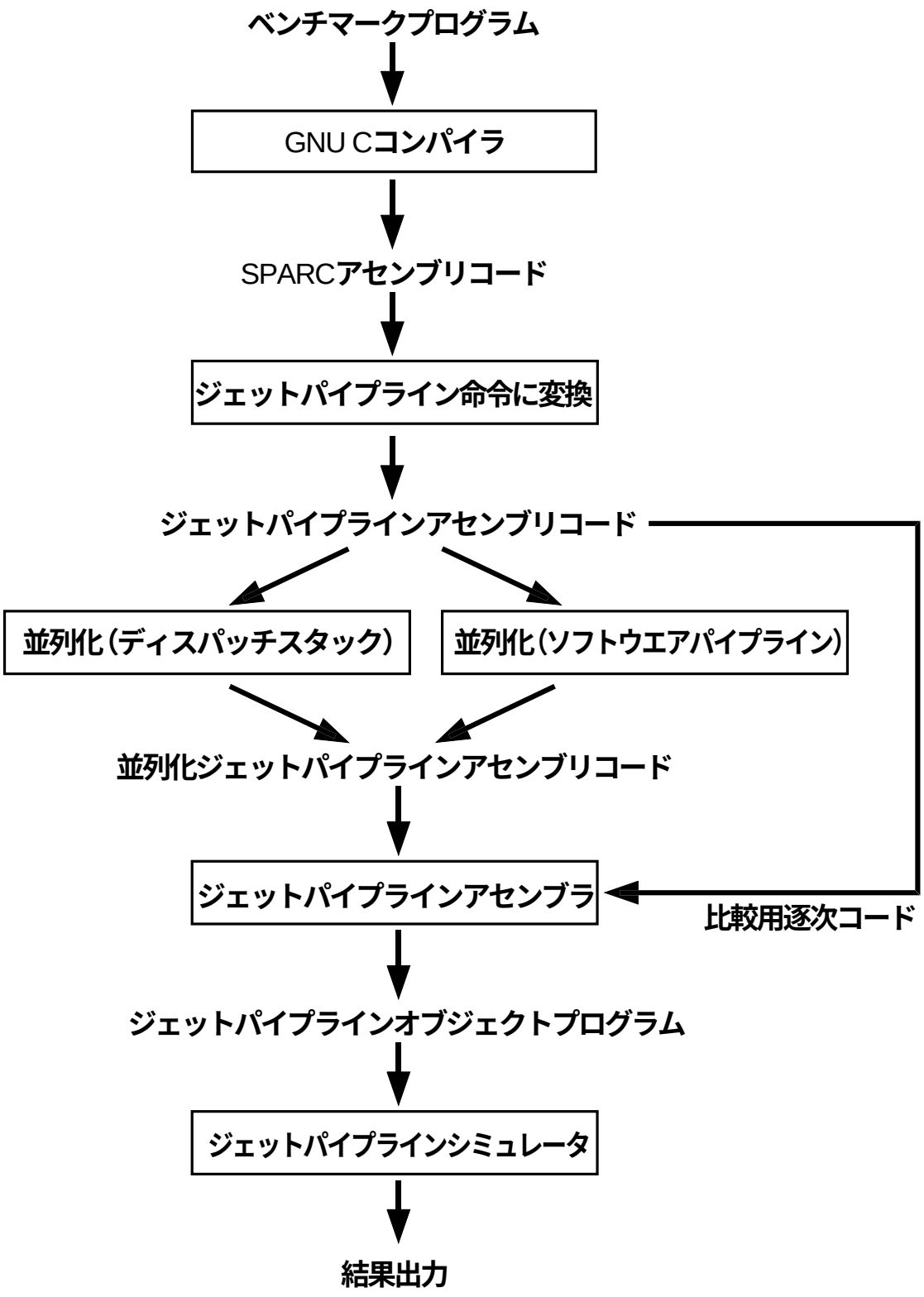

図4-2　ベンチマークプログラムの実行過程

・KERNEL 1**ソースプログラム：流体**
（ベクトル化、並列化ともに容易）

```
      DO 1 K = 1,N
    1    X(K)= Q + Y(K)*(R*ZX(K+10) + T*ZX(K+11))
```

・KERNEL 3**ソースプログラム：ベクトルの内積**
(合計以外はベクトル化可能、並列化も可能)

```
           Q = 0.000D0
      DO 3 K = 1,N
    3      Q = Q + Z(K)*X(K)
```

ベクトルの内積

・KERNEL 5**ソースプログラム：三角化消去**
(ベクトル化不可能、並列化も困難)

```
      DO 5 I = 2,N
    5   X(I) = Z(I)*(Y(I) - X(I-1))
```

ループの繰り返し間に依存関係あり

・KERNEL 7**ソースプログラム：状態方程式**
（ベクトル化、並列化ともに容易）

```
      DO 7 k= 1,N
        X(k)=     U(k  ) + R*( Z(k  ) + R*Y(k  )) +
     .         T*( U(k+3) + R*( U(k+2) + R*U(k+1)) +
     .         T*( U(k+6) + R*( U(k+5) + R*U(k+4))))
    7 CONTINUE
```

・KERNEL 9**ソースプログラム：整数予測**
（ベクトル化、並列化ともに容易）

```
      DO 9  i = 1,n
      PX( 1,i)= DM28*PX(13,i) + DM27*PX(12,i) + DM26*PX(11,i) +
     .          DM25*PX(10,i) + DM24*PX( 9,i) + DM23*PX( 8,i) +
     .          DM22*PX( 7,i) +  C0*(PX( 5,i) +      PX( 6,i))+
     .          PX( 3,i)
    9 CONTINUE
```

図4-3　**リバモアループベンチマークプログラム**

```
#include <stdio.h>

#define false 0
#define true  1

Try(i, q, a, b, c, x)
int  i,*q,a[],b[],c[],x[];
{
      int j;

      j = 0;
      *q = false;
      while ( (! *q) && (j != 8) )
      {
            j = j + 1;
            *q = false;
            if ( b[j] && a[i+j] && c[i-j+7] )
            {
                  x[i] = j;
                  b[j] = false;
                  a[i+j] = false;
                  c[i-j+7] = false;
                  if ( i < 8 )
                  {
                        Try(i+1,q,a,b,c,x);
                        if ( ! *q )
                        {
                              b[j] = true;
                              a[i+j] = true;
                              c[i-j+7] = true;
                        }
                  }
                  else
                        *q = true;
            }
      }
}

Doit ()
{
      int i,q;
      int a[9], b[17], c[15], x[9];

      i = 0 - 7;
      while ( i <= 16 )
      {
            if ( (i >= 1) && (i <= 8) )
                  a[i] = true;
            if ( i >= 2 )
                  b[i] = true;
            if ( i <= 7 )
                  c[i+7] = true;
            i = i + 1;
      }

      Try(1, &q, b, a, c, x);
      /* Don't use printf in the simulation on Jetpipeline.
      if ( ! q )
            printf (" Error in Queens.¥n");
      */
}

Queens ()
{
    int i;

    for ( i = 1; i <= 50; i++ )
            Doit();
}
```

図4-4　**スタンフォードベンチマークプログラムの例**(N_Queen)

し、実行を行った。この際、そのままではイタレーション間に依存関係が存在するためにベクトル化できないループにおいては、できるだけベクトル命令を使用できるようにソースプログラムの変形を行った。その例を図4-5に示す。いずれの場合も、ループ中で依存関係がない乗算の部分を分離して実行し、その結果を一時変数に格納した後にスカラ命令を用いて依存関係のある部分を処理するように変形した。

本アーキテクチャでは通常の命令語を複数個並べて各命令パイプラインに割り付ける方式をとっているため、VLIW方式とは異なり簡単にアセンブリ言語レベルで並列化プログラムを記述できる利点がある。具体的には、４の倍数であるワードアドレスからの４命令が同時にフェッチされ、並列実行されるので、アセンブラ上で同時に実行したい命令を４の倍数のアドレスから４つ並べて書くだけでよい。また、アセンブラ側でもこの記述法を支援するために、プログラムコードの部分ではジャンプ先を示すラベルが４の倍数でないアドレスにある場合にエラーを生じるようになっている。

§4.5　シミュレーション結果と検討

4.5.1　数値演算処理に対するスカラ命令による並列処理の評価

まずはじめに、ジェットパイプライン・アーキテクチャが持つ複数の命令パイプラインによる性能向上を評価するために、ベクトル命令を用いずにスカラ命令のみを使用し、各種の方法で並列化を行いシミュレータ上で実行した。なお、主に数値計算における性能を評価することを目的とし、ベンチマークプログラムにはリバモアループを使用した。図4-6に、シミュレーション結果を示す。グラフは実行サイクル数により計測された実行時間を表している。また、各グラフに付加されている括弧の中の数値は、並列化を行わず一つの命令パイプラインのみを使用して実行した場合の実行時間を１とし、それに対するジェットパイプラインの実行時間の比率で表したスピードアップである。並列化手法には、ディスパッチスタック法とソフトウエアパイプライン法の両方を使用し、比較を行った。

KERNEL 1は流体、KERNEL 7は状態方程式、およびKERNEL 9は整数予測の計算をそれぞれ行うループプログラムである。これらのプログラムは繰り返し間に依存関係を持たないため、並列化が容易で大きな速度向上が達成された。とくに、ソフ

```
for (k = 1; k <= n; k++)
        q = q + z[k] * x[k];
```
← 依存関係あり

↓

```
for (k = 1; k <= n; k++)
        p[k] = z[k] * x[k];
```
← こちらのみベクトル化
```
for(k=1; k<=n; k++)
        q = q + p[k];
```

(a) KERNEL 3の変形

```
for (i = 1 ; i < n ; i++)
x[i] = z[i] * (y[i] - x[i-1]);
```
← 依存関係あり

↓

```
for (i = 1 ; i < n ; i++)
    p[i] = z[i] * y[i];
```
← こちらのみベクトル化
```
for(i = 1 ; i < n ; i++)
    x[i] = p[i] - z[i] * x[i-1];
```

(b) KERNEL 5の変形

図4-5　**リバモアループのベクトル化のための変形**

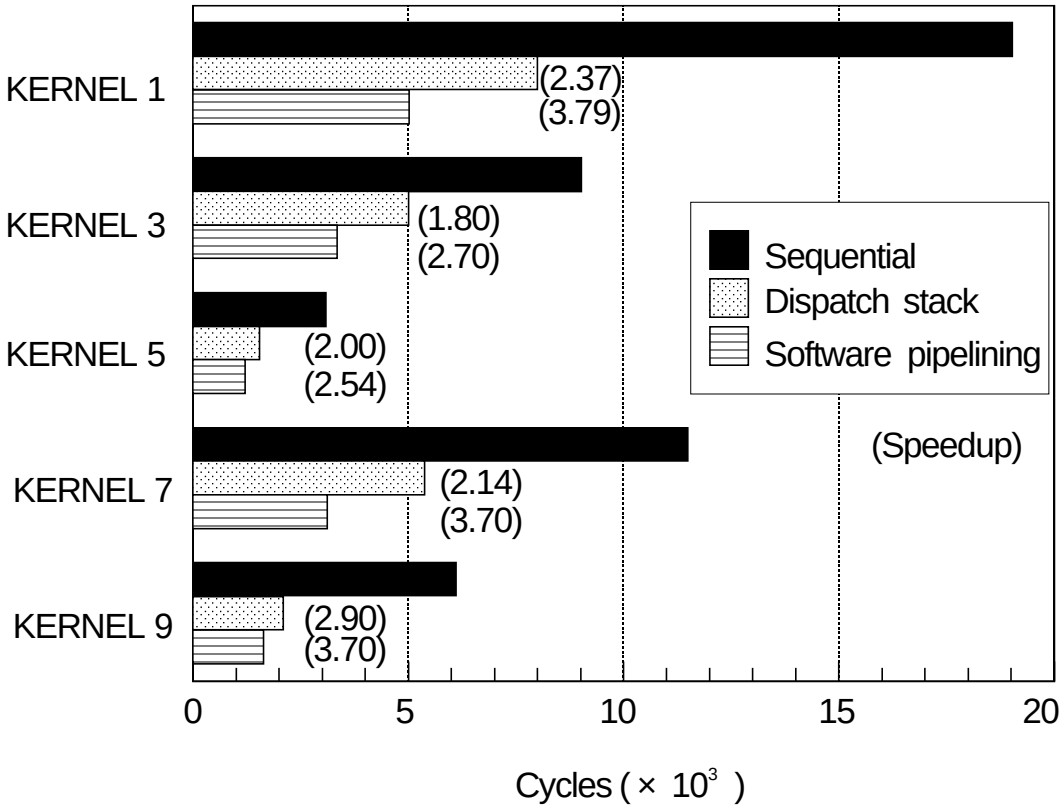

図4-6　**リバモアループによるシミュレーション結果**
(スカラ命令のみを使用)

トウエアパイプラインを用いた場合には、スピードアップはIP数に近い3.70 ~ 3.79という結果を得ることができた。

KERNEL 3はベクトルの内積、KERNEL 5は三角化消去の計算をそれぞれ行うループプログラムである。これらのループには繰り返し間に依存関係が存在し、容易には並列化およびベクトル化を行うことができない。しかしながら、ジェットパイプライン・アーキテクチャにおいてソフトウエアパイプライン法を適用した場合、KERNEL 3では2.7倍、KERNEL 5では2.54倍のスピードアップを達成できた。

これらの例において、ソフトウエアパイプライン手法はかなり有効であることが示された。ディスパッチスタック法はソフトウエアパイプライン手法ほどの速度向上は得られなかったものの、ループ以外の部分にも適用可能であるという利点がある。また、ループにアンローリングを行い、並列性を増した後にディスパッチスタック法を適用して並列化を行うと性能が向上することが考えられる。実際の応用においては両方が使用されることになるであろう。

4.5.2 汎用処理に対するスカラ命令による並列処理の評価

つぎに、数値計算に限らない幅広い応用分野に対する性能を評価するために、スタンフォードベンチマークプログラムを用いて性能評価を行った。スタンフォードベンチマークプログラムは、リバモアループのような単純なループのみから構成されているわけではないので、並列化手法にはループ以外でも適用可能なディスパッチスタックを使用し、逐次型プログラム（実際にはIP0にのみ命令を入れて実行）と比較し評価を行った。図4-7～図4-13にシミュレーション結果を示す。なお、グラフ中のX軸は各ベンチマークの問題の大きさ、たとえばN_QueensプログラムであればQueenの数、Sortを実行するプログラムの場合にはソートするデータ数などを意味している。また、折れ線グラフが実行時間(サイクル数,左のY軸)、棒グラフがプログラムのコードサイズ(キロバイト,右のY軸)をそれぞれ表している。ここで、逐次型のプログラムサイズはIP0以外に割り当てられているNOPの数を含んでおり、実質上はその1/4である。従って、並列化後のコードサイズと逐次型コードサイズの差が、並列化の結果逐次型でNOPであった部分に移動できなかったコードの割合を示していると考えられる。

各ベンチマークプログラムにおける速度向上比は、もっとも良い結果を示したハ

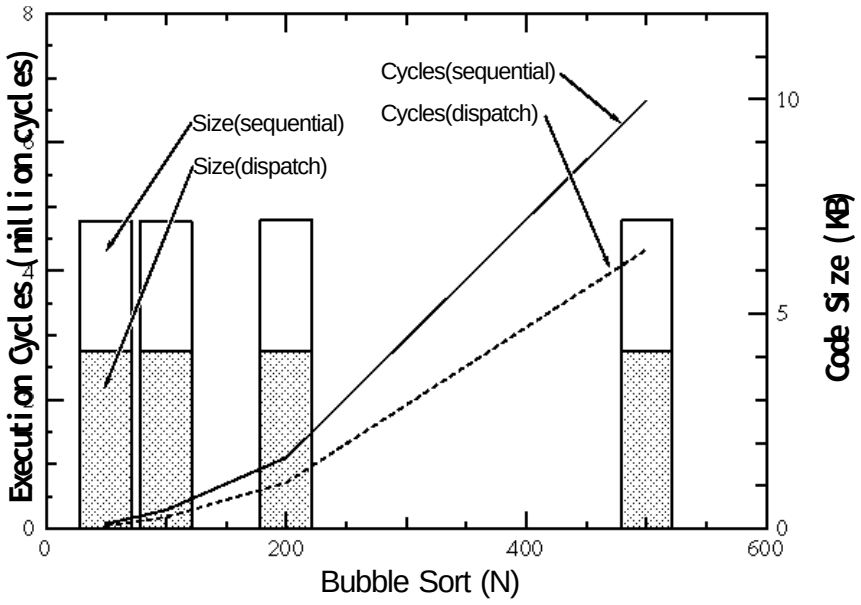

図4-7　**スタンフォードベンチマークによるシミュレーション結果** (Bubble Sort)

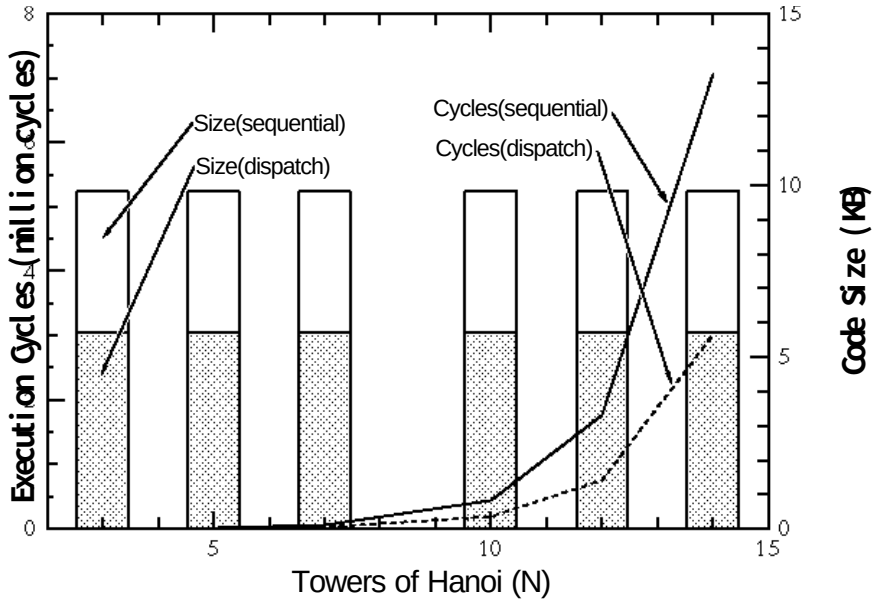

図4-8　**スタンフォードベンチマークによるシミュレーション結果**
(Towers of Hanoi)

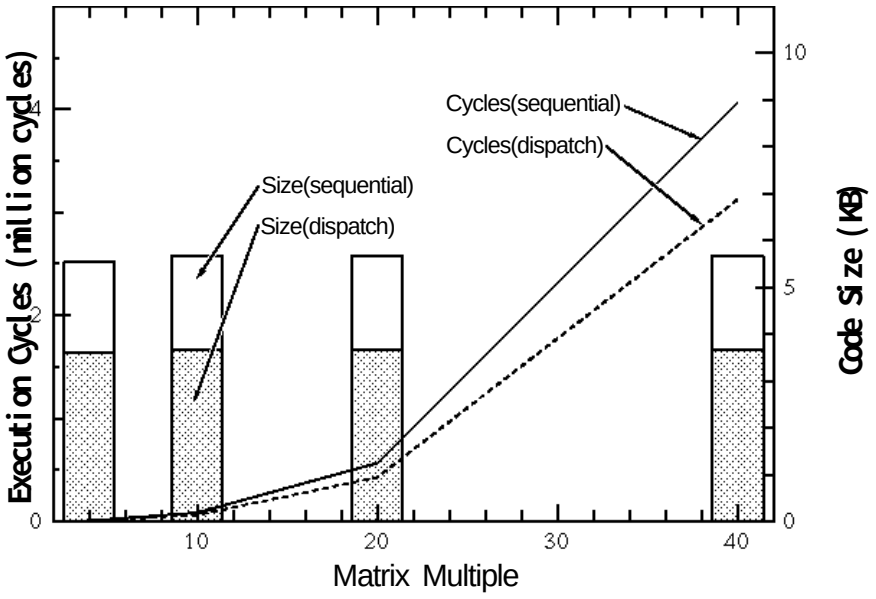

図4-9　**スタンフォードベンチマークによるシミュレーション結果**
(Matrix Multiple)

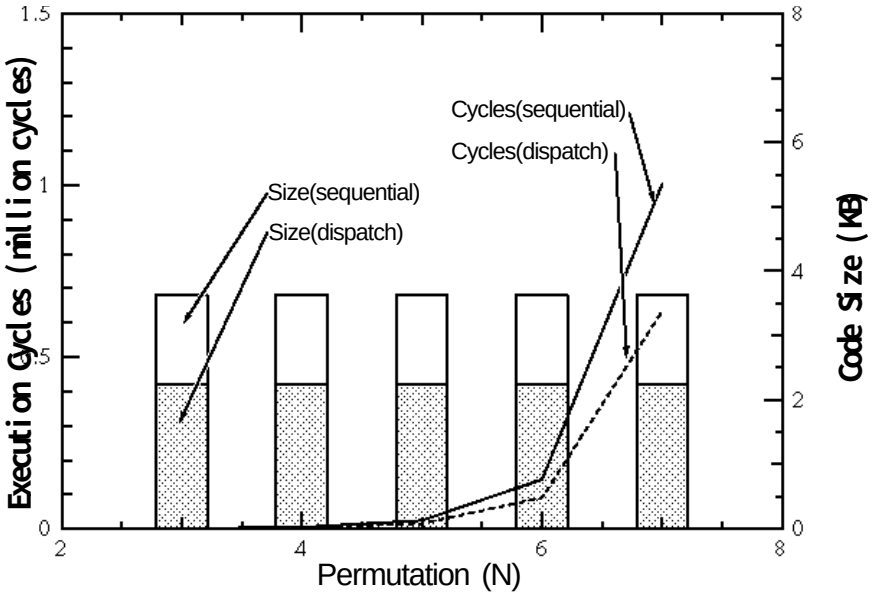

図4-10　**スタンフォードベンチマークによるシミュレーション結果** (Permutation)

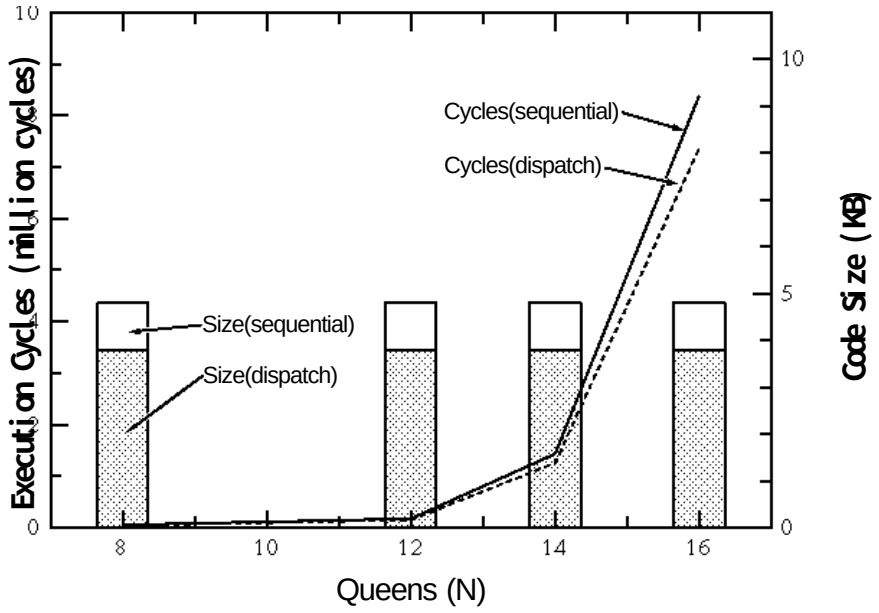

図4-11　**スタンフォードベンチマークによるシミュレーション結果** (N_Queens)

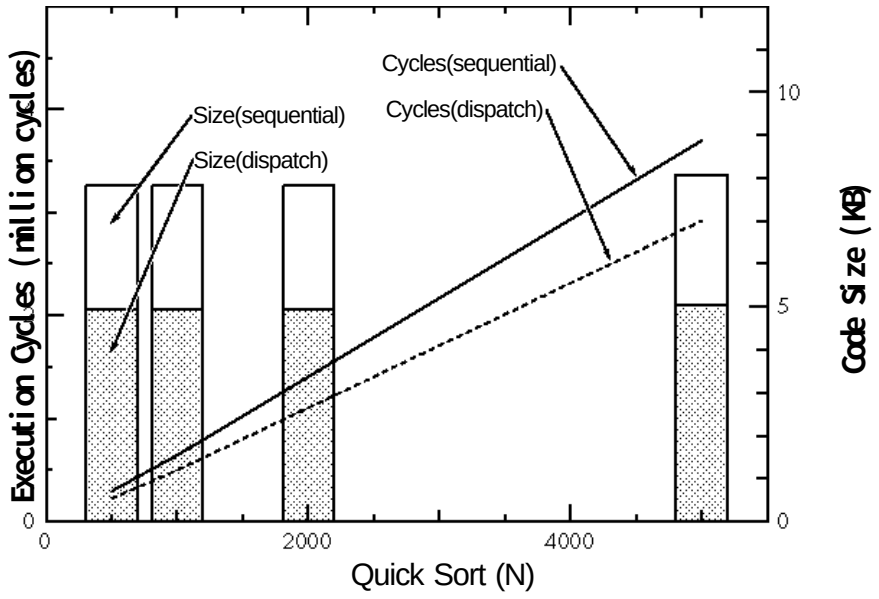

図4-12　**スタンフォードベンチマークによるシミュレーション結果**
(Quick Sort)

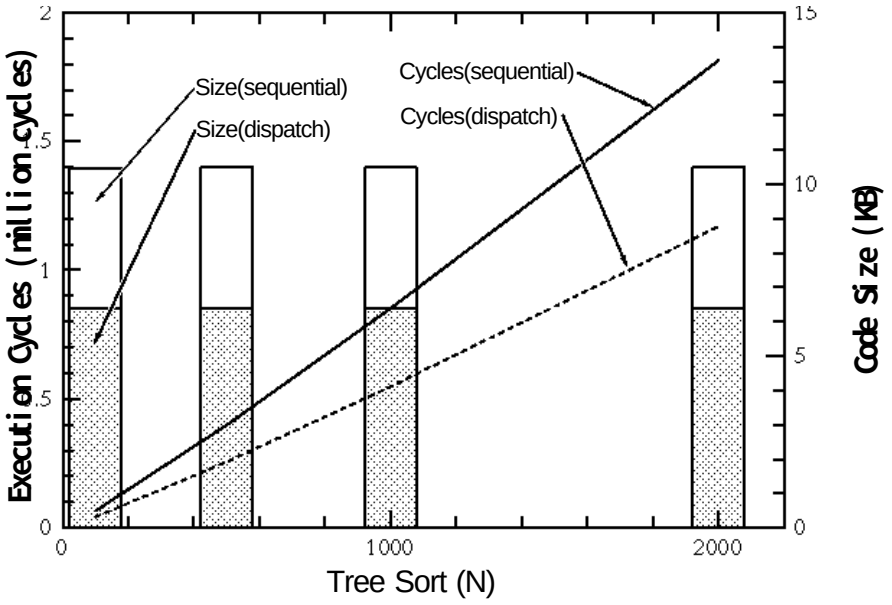

図4-13　**スタンフォードベンチマークによるシミュレーション結果** (Tree Sort)

ノイの塔の場合で2.3倍程度、最悪値を示したN Queensで1.2倍程度、残りは1.3〜1.6倍といった結果であった。リバモアループにおける依存関係を持つループについてのディスパッチスタックを用いた場合と同程度か、若干低い速度向上が得られた。関数呼び出し等を含み、基本ブロック長も短いプログラムが多いことを考慮すると、妥当な結果であるといえる。

さらに性能を向上させるためには、VLIWアーキテクチャで実際に使用されているような、基本ブロックを超えた最適化が必要であると思われる。

4.5.3 数値演算処理に対するベクトル性能の評価

最後に、リバモアループを用いて数値演算処理に対するベクトル処理の性能評価を行った。前述したとおり、ソースプログラムからベクトル命令を使用した機械語コードへの変換は人手によって行っている。また、図4-5で示したようなソースプログラムの変形を用いてできるだけベクトル化可能な部分が多くなるようにしている。ここでは4.5.1節で使用したKERNEL番号と同じものをベンチマークプログラムとして採用した。図4−14〜図4−18にそれぞれのKERNELに対するベクトル命令およびスカラ命令のスケジューリングの概略を示す。なお、図中のベクトル命令については、その種類(VIがベクトル整数演算命令(ブロードキャストを含む), VAがベクトル浮動小数点加減算, VMがベクトル浮動小数点乗算, VL/VSがそれぞれベクトルロード・ストア)を併記している。

図4-19に、ベクトル命令を用いた場合の実行時間と4.4.1節で評価したスカラ命令のみを用い並列化した場合との実行時間を比較したグラフを示す。ベクトル化できる部分が多いKERNEL 1,7,9は他の手法とほぼ同じかわずかに良い結果が得られたが、完全にはベクトル化できないKERNEL 3,5については、ソースプログラムの変形によりベクトル化可能な部分を引き出したにも関わらず、他の並列化手法よりも性能が低く逐次型プログラムと同程度の速度向上しか得られないものもあった。

これらのことから、ベクトル化率を高くできるプログラムについてはベクトル処理を用いることによるメリットがあるものの、ベクトル化率をあまり高くできないプログラムではジェットパイプライン・アーキテクチャの複数の命令パイプラインを使用したスカラ命令による並列化の方が効果があるということが判明した。

なお、今回はベクトル命令について完全に最適化を行ったわけではなく、またチ

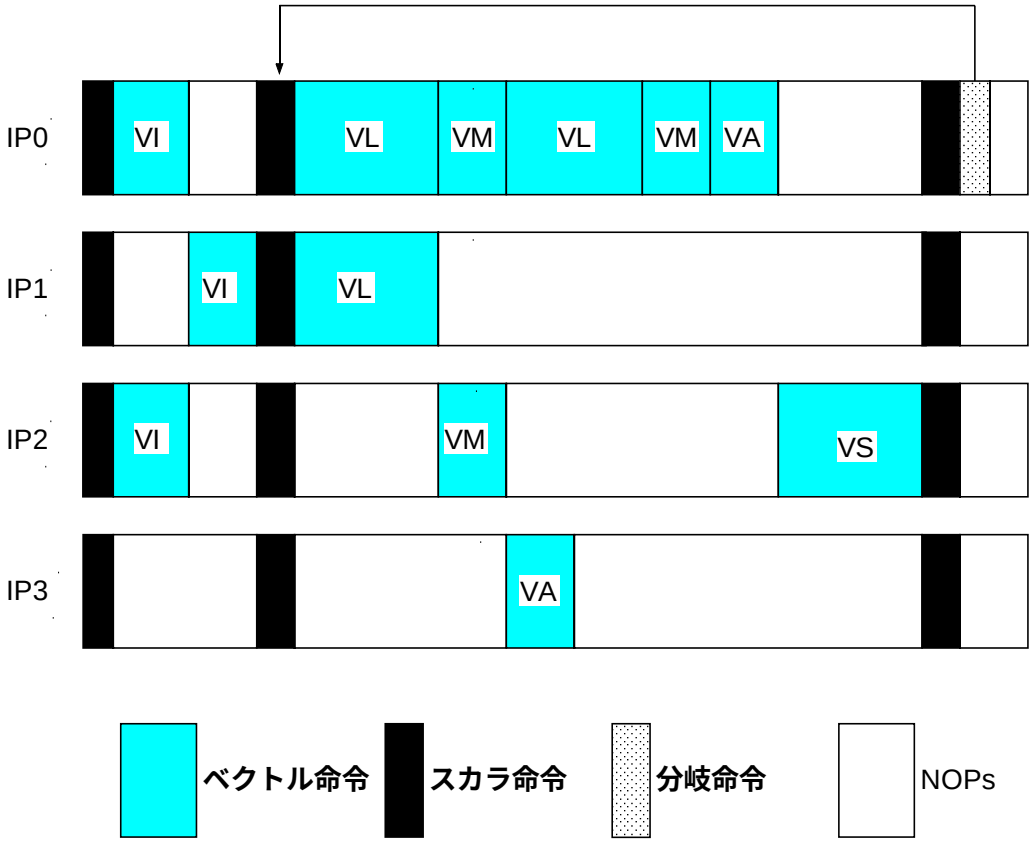

図4-14　KERNEL 1**の命令スケジューリング**

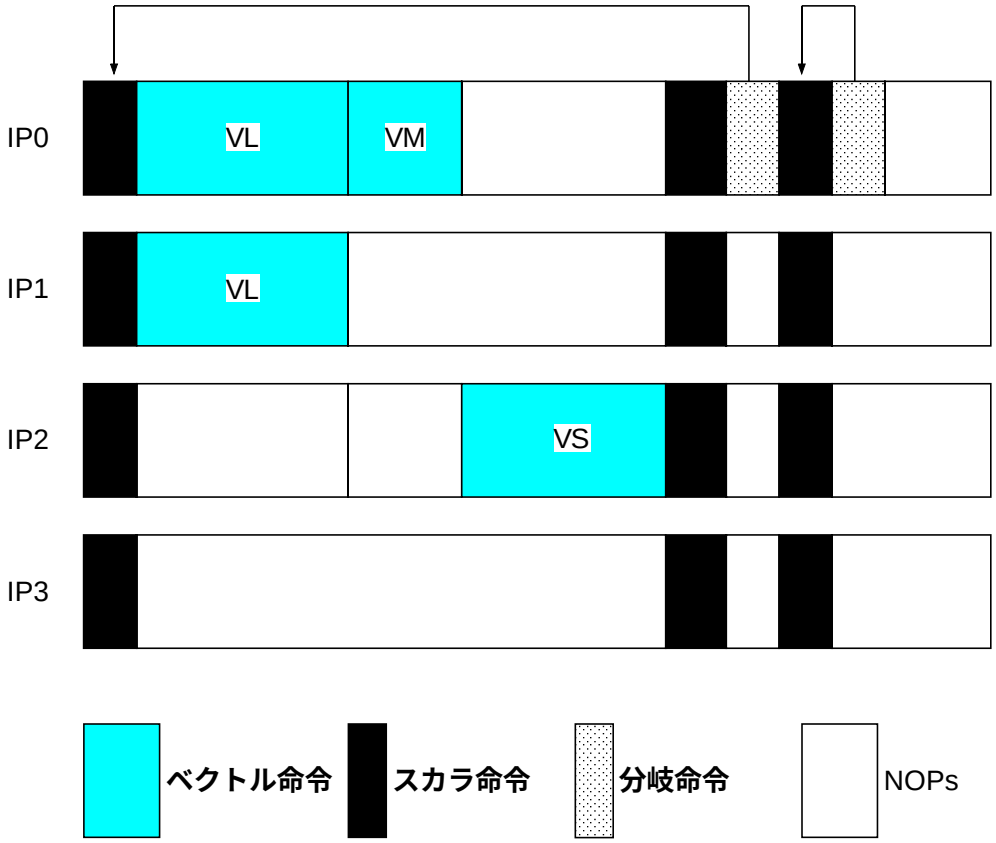

図4-15　KERNEL 3の命令スケジューリング

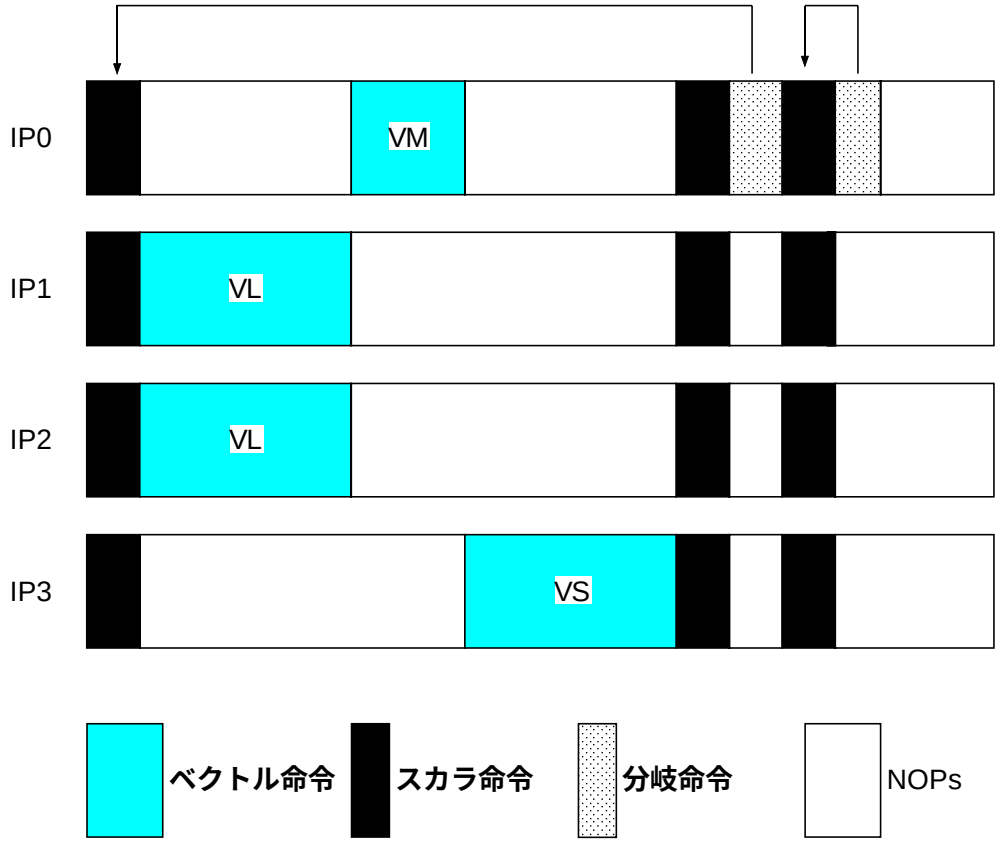

図4-16　KERNEL 5**の命令スケジューリング**

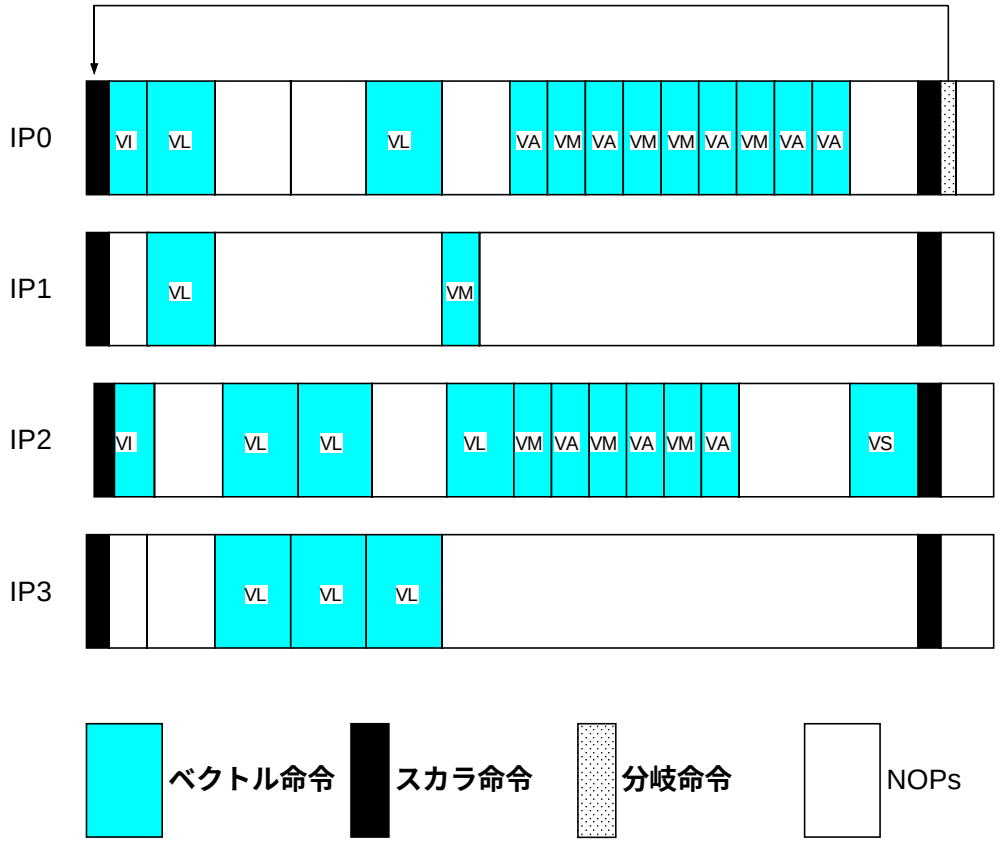

図4-17　KERNEL 7**の命令スケジューリング**

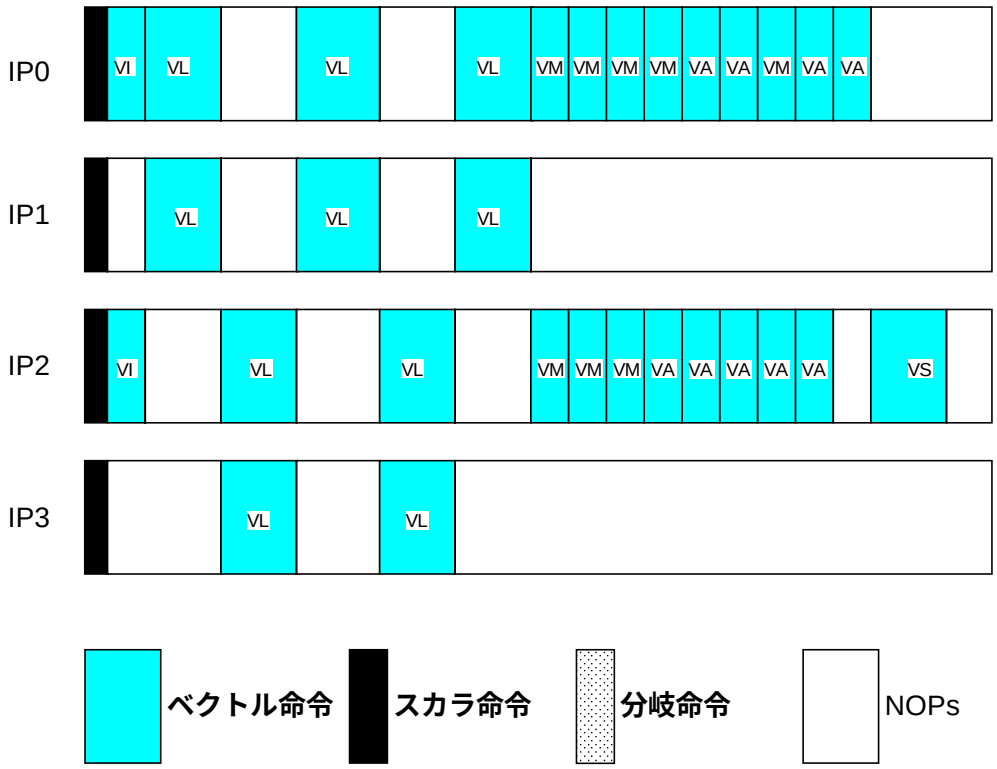

図4-18　KERNEL 9**の命令スケジューリング**

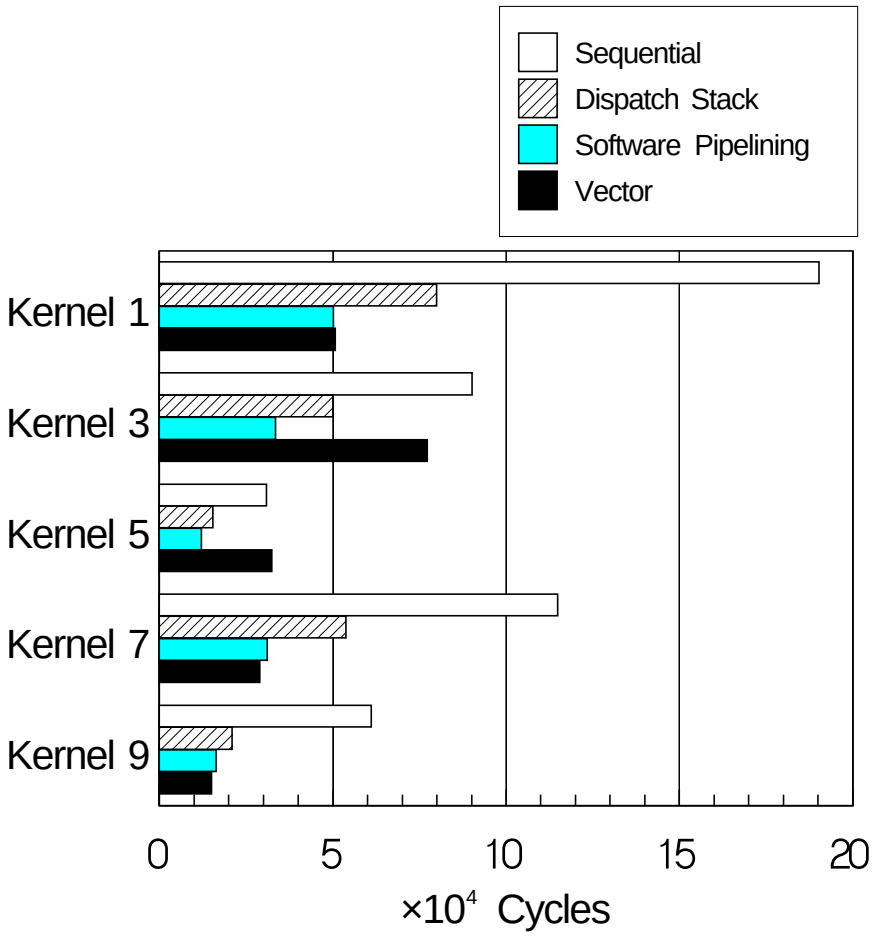

図4-19　ベクトル処理とスカラ並列処理の比較（リバモアループ）

ェイニングも行っていないので、チェイニング等を用いて最大限に最適化を行った場合ベクトル性能は今回の結果より改善される可能性もある。

§4.5　結言

本章では、前節で提案したジェットパイプライン・アーキテクチャを、ソフトウエアシミュレーションによりその性能評価を行った。

まず、スカラ命令のみを用いた並列化に対する性能評価を行った。複数の命令パイプラインを用いた並列化により、各種のプログラムに対して高速化できることがわかった。特に、ループ間に依存関係があり、簡単には並列化できないプログラムにおいても、ソフトウエアパイプライン手法を適用することにより高速化可能な場合があることがわかった。また、スタンフォードベンチマークのような、単純なループのみから構成されてはいない一般の応用プログラムについても、ディスパッチスタック法を適用することによりリバモアループにおける依存関係のあるプログラムに対してディスパッチスタック法を適用した場合と同程度の速度向上が得られることもわかった。ただし、基本ブロック長が短いプログラムに関しては余りよい結果が得られなかったため、さらなる性能向上のためにはVLIWのような基本ブロックを超えた最適化が必要であると思われる。

次に、数値演算処理におけるベクトル処理性能についても評価した。その結果、ベクトル化率を高くできるプログラムに関してはメリットがあるものの、ベクトル化率の低いプログラムではスカラ命令による並列化よりも劣ることがわかった。従って、プログラムの性質に応じて適当な並列化手法を選択し適用することが、ジェットパイプライン・アーキテクチャ上で高性能を発揮させるためには必要であるといえる。

今回のシミュレーションはメモリアクセス遅延について考慮していなかった。性能評価に使用したベンチマークのようにその命令およびデータがキャッシュにすべて含まれてしまうような小さなループプログラムの場合には大きな問題にはならないかもしれないが、実際の応用プログラムの場合には影響があると考えられる。したがって、メモリシステムも考慮したシミュレーションが今後の課題である。