漸化式のスーパースカラ向け高速化

海永正博† 久島伊知郎†

スーパースカラプロセッサ向けの漸化式またはリカレンス高速化方式を提案する．高速化の具体例はリバモアループの11番とする．マルチプロセッサ向けのリカレンス高速化方式として巡回縮約が知られている．巡回縮約は演算数をNから$N\log_2 N$に増大するが，その見返りに並列度を1からNに増大させる．したがって超並列機などかなりの演算器数が期待できる状況では高速化が期待できる．しかし巡回縮約はスーパースカラ向け（シングルプロセッサ向け）の高速化方式にはならない．並列性増大の利点よりも演算数増大の欠点のほうが大きく効いてくるからである．本稿で提案の方式は巡回縮約にヒントを得ている．本稿の提案では巡回縮約の余分な演算数の増大を抑えるとともに，スーパースカラで活用できる程度の並列性を引き出した．これによりスーパースカラプロセッサにおいてリバモアループの11番を2倍以上高速化できた，というのが本稿の主要な主張点である．

A Fast Execution of Recurrences on Superscalar Processors

MASAHIRO KAINAGA† and ICHIRO KYUSIMA†

We propose a fast algorithm to solve recurrences in superscalar processors. Cyclic reduction is known as a fast recurrence solver for multi-processor systems. Though cyclic reduction increases the number of operations from N to $N\log_2 N$, it increases the degree of parallelism from 1 to N. In a system with a large number of processors like an MPP, a reasonable speedup is expected by cyclic reduction. However, cyclic reduction is not suitable for a superscalar processor (single processor system), because the increase in the number of operations is more significant than the increase in the degree of parallelism. Our method prevents the increase of extra operations and exploits suitable parallelism for a superscalar processor. We can get a speedup of more than 2 for livermore loop 11 by this method.

1. はじめに

演算パイプラインプロセッサの並列性を阻害する要因としてリカレンス[1]がある．リカレンスは日本語では漸化式と訳される．数列 $\{a_i\}$ の a_{i+1} の値を求める際に以前の a_i などの値を使用するのがリカレンスである．値の定義/使用（フロー依存）の長い系列をリカレンスが引き起こす．そしてこの長い系列を素朴には逐次的に実行しなくてはならない．この逐次性が並列化を阻害し，したがって高速化の妨げになる．

本稿では，スーパースカラプロセッサ[3]向けのリカレンス高速化方式を提案する．高速化の具体例はリバモアループの11番とする．このループはリカレンスの典型例と判断できるからである．

マルチプロセッサ向けのリカレンス高速化方式として巡回縮約[1]が知られている．そしてこの方式を具体的に適用した例も報告[2]されている．巡回縮約は演算数をNから $N\log_2 N$ に増大するが，その見返りに並列度を1からNに増大させる．したがって超並列機などかなりの演算器数が期待できる状況では高速化が期待できる．しかし巡回縮約はスーパースカラ向け（シングルプロセッサ向け）の高速化方式にはならない．並列性増大の利点よりも演算数増大の欠点のほうが大きく効いてくるからである．この点は後で触れる．

本稿で提案の方式は巡回縮約にヒントを得ている．本稿の提案では巡回縮約の余分な演算数の増大を抑えるとともに，スーパースカラで活用できる程度の並列性を引き出した．これによりスーパースカラプロセッサにおいてリバモアループの11番を2倍以上高速化できた，というのが本稿の主要な主張点である．

2. リカレンス

2.1 リカレンス，リバモアループ11

リカレンスとは，ある本では漸化式と訳されている．ある項 a_{i+1} が前の項 a_i（厳密には以前の項）で計算される式（数列）のことである．以下の式はすべ

† 日立製作所システム開発研究所
Systems Development Laboratory, Hitachi, Ltd.

```
for(i=1;i<1000;i++)
    a[i]=a[i-1]+d[i-1];
```

図 1 リバモアループ 11
Fig. 1 Livermore loop 11.

てリカレンスの例となる．

$$a_{i+1}=a_i+d_i \qquad (1)$$

$$a_{i+1}=c_i*a_i+d_i \qquad (2)$$

式(1)は(逐次)部分和と呼ばれている．数列 $\{d_i\}$ の部分和を逐次的に計算しているからである．式(2)には，線形1次リカレンスという名前がついている．線形は式の形を，1次は以前の項が一つだけ，ということである．

リバモアループのループ11は**図1**のようなループである．本来は FORTRAN で記述されているが，ここではC言語で記述してある．この図の a[i] を a_i，d[i] を d_i に対応させれば，ループ11は式(1)の部分和に対応したループとなる．

2.2 カスケード和，巡回縮約

加算の完了に4クロックかかるとすれば，リバモアループ11は素朴には 4000(=4*1000) クロック程度掛かってしまう．しかし理想的な超並列機（この意味はすぐに説明される）であれば，カスケード和のテクニックで4000を40クロック程度に低減できる．

ここでのカスケード和とは，トーナメント方式の総和演算を少し拡張したものを言う．これをプログラムで表現すると**図2**となる．(a)がトーナメント和で(b)がカスケード和である．この図2で1000が1024に調整されているがその点は無視すること，またカスケード和では負の添え字が出現するがそこには0がある想定である．二つのプログラムの違いは初期値，増分値だけである点に注目のこと．

```
for(i=1;i<1024;i+=2)
  d1[i]=d[i]+d[i-1];
for(i=3;i<1024;i+=4)
  d2[i]=d1[i]+d1[i-2];
....
for(i=511;i<1024;i+=512)
  d9[i]=d8[i]+d8[i-256];
for(i=1023;i<1024;i+=1024)
  a[i]=d9[i]+d9[i-512];
```

(a) トーナメント和 / (a) Tournament sum

```
for(i=1;i<1024;i++)
  d1[i]=d[i]+d[i-1];
for(i=1;i<1024;i++)
  d2[i]=d1[i]+d1[i-2];
....
for(i=1;i<1024;i++)
  d9[i]=d8[i]+d8[i-256];
for(i=1;i<1024;i++)
  a[i]=d9[i]+d9[i-512];
```

(b) カスケード和 / (b) Cascade sum

図 2 超並列向けプログラム
Fig. 2 Programs for MPP.

二つのプログラムでは，個別のループが終わるごとに新たな配列に計算の途中結果が格納されていく．最初，配列 d に $\{d_0, d_1, d_2, \cdots, d_{1023}\}$ が格納されていたとして，**図3**のようになる．なおこの図において，d_0〜d_3 は $d_0+d_1+d_2+d_3$ の省略記法である．トーナメント和では10個のループ処理が完了すると総和 $\{d_0$〜$d_{1023}\}$ が求まり，カスケード和では10個のループ処理が完了すると個別の部分和 $\{d_0, d_0+d_1, d_0$〜$d_2, \cdots, d_0$〜$d_{1023}\}$ が求まることになる．

さて，図2の個別の for ループを吟味する．すると個別の for ループにおいて個別のイタレーションは一斉に実行してよい，または依存解析[4]の言葉ではループ運搬依存がない，ということがわかる．となるとプロセッサ要素数1024の理想的超並列機であれば，具体的には個別の加算の完了が4クロックという想定で，さらに転送オーバーヘッドが0という想定の理想的超並列機であれば，全体処理（総和，部分和）を都合 40(=4*10) クロックで完了できることになる．

理想的な超並列機であれば1000要素の部分和計算を40クロック程度で実行できる点を説明したが，以上の論理展開を吟味すれば，スーパースカラプロセッサでも4000クロックでなく2000クロック以下で実行できそうな気がする．この点を次の章で確かめることにする．

カスケード和では演算数が10倍（一般には $N\log_2 N$）に増大するが，一方で演算数増大を抑え，一方である程度の並列性を引き出せれば，スーパースカラ向きの高速化方式になるであろう，というのが基本の発想である．

3. スーパースカラと部分和計算

3.1 逐次性の回避

部分和の素朴な計算方法を再度示しておく．

$$a_{i+1}=a_i+d_i \qquad (3)$$

これを，文字通り（正確には式通り）に計算すれば，確定した a_i により a_i+d_i を計算し a_{i+1} を確定するわけで，完全に逐次的な計算となる．

しかし最初の式を1度展開すれば以下のようになる．

	(a) トーナメント和 / (a) Tournament sum	(b) カスケード和 / (b) Cascade sum
d	$\{d_0, d_1, \ldots, d_{1023}\}$	$\{d_0, d_1, \ldots, d_{1023}\}$
d1	$\{d_0+d_1, d_2+d_3, \ldots, d_{1022}+d_{1023}\}$	$\{d_0, d_0+d_1, d_1+d_2, d_2+d_3, \ldots, d_{1022}+d_{1023}\}$
d2	$\{d_0$〜d_3, d_4〜$d_7, \ldots, d_{1020}$〜$d_{1023}\}$	$\{d_0, d_0+d_1, d_0$〜d_2, d_0〜$d_3, \ldots, d_{1020}$〜$d_{1023}\}$
	...	...
d9	$\{d_0$〜d_{511}, d_{512}〜$d_{1023}\}$	$\{d_0, d_0+d_1, d_0$〜$d_2, \ldots, d_0$〜d_{511}, d_1〜$d_{512}, \ldots\}$
a	$\{d_0$〜$d_{1023}\}$	$\{d_0, d_0+d_1, d_0$〜$d_2, \ldots, d_0$〜$d_{1023}\}$

図 3 配列の内容
Fig. 3 Contents of arrays.

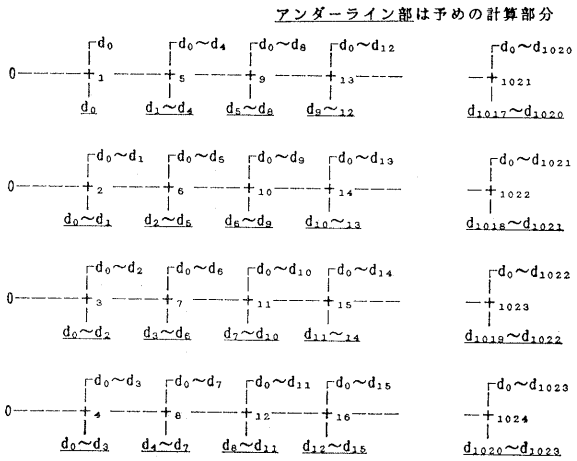


図 4 巡回縮約2回適用後の逐次和
Fig. 4 Sequential sum after two doublings.

$$a_{2i+1}=a_{2i-1}+\underline{d_{2i-1}+d_{2i}} \quad \text{(奇数系列)}$$
$$a_{2i+2}=a_{2i}+\underline{d_{2i}+d_{2i+1}} \quad \text{(偶数系列)}$$

これにより，リカレンスの系列が一つから二つになり，個別の系列の長さが半分になったことになる．なお，$\{\underline{d_{2i-1}+d_{2i}}\}$ や $\{\underline{d_{2i}+d_{2i+1}}\}$ はあらかじめ計算（あらかじめの計算部分をアンダーラインで示す．以下同様）しておく必要があるが，この計算にはリカレンス（や長いフロー依存の系列）は発生しないので，かなり高速に実行できる．

以上，一つの系列から二つの系列を生成する手続きを示したが，再度同じ手続きを踏めば二つの系列から四つの系列を生成できる．このように再帰的に系列の数を倍にする（系列の長さは半分になる）手続きは巡回縮約（cyclic reduction）と呼ばれている．実は，式(3)に巡回縮約を可能な限り適用していった結果は前節のカスケード和そのものである．

さて我々は，巡回縮約を2回程度適用した部分に興味がある．というのは四つの程度の（フロー依存の意味で無関係な）独立した部分和の系列が生成されているからである．これを**図4**に示す．この図の見方は以下のようである．例えば $+_1$ について，$\underline{d_0}$ と0を加算し d_0 を求めそれを $+_5$ に渡す．$+_5$ について，$\underline{d_1\sim d_4}$ と $+_1$ からの d_0 を加算し $d_0\sim d_4$ を求めそれを $+_9$ に渡す．

今，$\{\underline{d_0}, \underline{d_1\sim d_4}, \underline{d_5\sim d_8}, \underline{d_9\sim d_{12}}\cdots\}$ などのあらかじめの計算（の手間）は無視しておく．すると＋記号の右下に付加した番号通りに加算を実行していけば部分和を生成できる．加算の完了が4クロックという想定であれば，フロー依存に伴う遅れは発生しない．すなわち，個別の加算を1クロックごとに実行できることになり，あらかじめの計算が無視できるのであれば，全体処理を1000クロック程度で実行できることになる．これは素朴には4000クロック程度はかかるという想定のものを1000クロックにするわけでかなりの高速化といえる．

ただし，$\{\underline{d_0}, \underline{d_1\sim d_4}, \underline{d_5\sim d_8}, \underline{d_9\sim d_{12}}\cdots\}$ などのあらかじめの（余分な）計算を無視することは本当は許されない．というのはあらかじめ（余分に）計算されておくべき項目は約1000個あり，個別の項目の計算に平均3個の加算が必要であり，結局のところ計4000クロックかかってしまうからである．

そこで，新たな計算方式を提案する．この方式の特徴は以下のように要約できる（ここでNの値は3-4がよいと思われる）．

（1）素朴な計算方法に較べてフロー依存の長さが $1/N$．というか，リカレンスの系列を1からNにする．
（2）余分な加算回数を本来の加算回数の $(N-1)/N$に．
（3）フロー依存に伴う遅れは発生しない．
（4）全クロック数は素朴な計算法の半分以下．ただし，演算の完了は4クロックの想定（以下同様）．

新しい計算方法を**図5**に示す．この図の見方は図4の見方とほぼ同じである．例えば $+_4$ について，$\underline{d_3}\sim\underline{d_5}$ と $+_1$ からの $d_0\sim d_2$ を加算し $d_0\sim d_5$ を求めそれを $+_7$ に渡す．

まず，$\{\underline{d_0}, \underline{d_0\sim d_1}, \underline{d_0\sim d_2}, \underline{d_3}, \underline{d_3\sim d_4}\cdots\}$ があらかじ

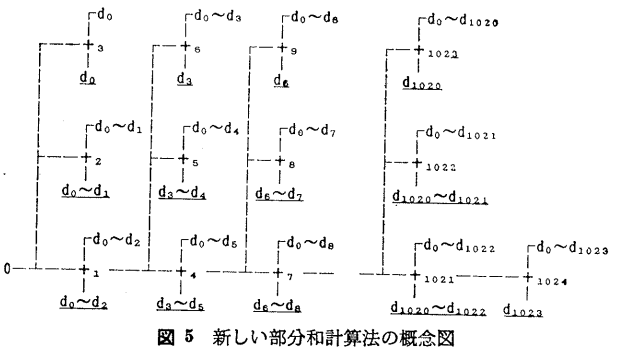


図 5 新しい部分和計算法の概念図
Fig. 5 New method for sequential sum.

め計算されているとする．すると，＋記号の右下に付加した番号通りに加算を実行していけば部分和が成生できる．加算の完了が4クロックという想定であれば，フロー依存に伴う待ちは3回に1回（$3n+1$の部分）発生する．すなわち，個別の加算を4/3クロックごとに実行できることになり，約1333クロックとなる．一方あらかじめの加算は約666個でフロー依存の意味で独立な演算木が333個であり，フロー依存に伴う待ちの回避が可能で明らかに666クロックで実行できる．となると全体の計算を約2000（＝1333＋666）クロックで実行できることになる．

そして実は，さらに速く計算できる．というのは部分和計算の部分とあらかじめの計算部分をうまくシャッフルすれば部分和計算のフロー依存に伴う待ちが解消されるからである．この結果1666クロックで実行されるはずとなる．このための具体的な方法を以下に提案する．

4. 部分和の高速計算関数

以下，部分和計算を関数の形で表現しよう（記述言語はC）．一度に最終版を提示するのでなく，以下の順に段階的に提示する．

（1）ループ展開版
（2）ソフトウェアパイプライニング版
（3）機械語イメージ版

4.1 ループ展開版

まず，ソフトウェアパイプライニングを導入する前の，ループ展開版のコードを示す（**図6**）．図6のラベルrest以降は端数要素（12で割った余り）を処理するためのもので，本質にはかかわらない．なお，入力 $\{d_1, d_2, \cdots\}$ が d[　] に格納されており，出力 {d1, d1＋d2, d1～d3, …} が a[　] に格納されるものとする．

図5に従い3要素ずつを処理する素朴なループであればループイタレーション1回当たり本来の加算3回，余分な加算2回の計5回の加算となる．図6はその4倍展開版であり，加算の回数はループイタレーション1回当たり20回となる．4倍の展開によりループ本体にかなりの

```
void p_sum(n,d,a) register n;float d[],a[];
{    register i;
     register float t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,ta,tb;
     register float sum1=0.0,sum2;
     for(i=0;(n-=12)>=0;i+=12){
          t0=d[i+ 0];          t1=t0+d[i+ 1];  t2=t1+d[i+ 2];    /* 1 */
          t3=d[i+ 3];          t4=t3+d[i+ 4];  t5=t4+d[i+ 5];    /* 1 */
          t6=d[i+ 6];          t7=t6+d[i+ 7];  t8=t7+d[i+ 8];    /* 1 */
          t9=d[i+ 9];          ta=ta+d[i+10];  tb=ta+d[i+11];    /* 1 */
          a[i+ 2]=sum2=t2+sum1;a[i+ 1]=t1+sum1;a[i+ 0]=t0+sum1;  /* 2 */
          a[i+ 5]=sum1=t5+sum2;a[i+ 4]=t4+sum2;a[i+ 3]=t3+sum2;  /* 2 */
          a[i+ 8]=sum2=t8+sum1;a[i+ 7]=t7+sum1;a[i+ 6]=t6+sum1;  /* 2 */
          a[i+11]=sum1=tb+sum2;a[i+10]=ta+sum2;a[i+ 9]=t9+sum2;  /* 2 */
     }
rest:n+=12;
     for(  ;--n>0;i++){
          a[i]=sum1=sum1+d[i];
     }
}
```

図6 ループ展開版
Fig. 6 Loop unrolling version.

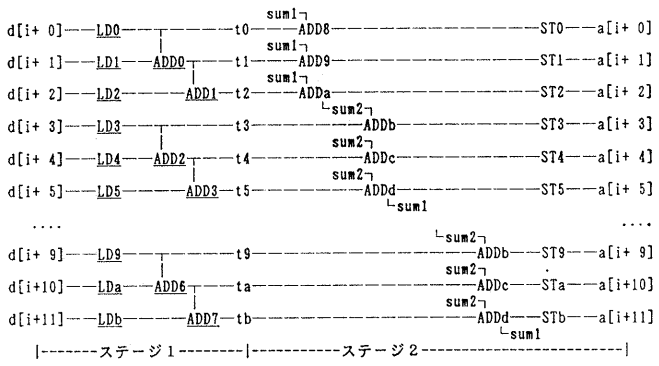


図7 演算構造
Fig. 7 Computation structure.

```
void p_sum(n,d,a) register n;float d[],a[];
{    register i=0;
     register float t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,ta,tb;
     register float sum1=0.0,sum2;
     if((n-=12)<0) goto rest;
     t0=d[i+ 0];t1=t0+d[i+ 1];t2=t1+d[i+ 2];          /* a */   /* 1 */
     t3=d[i+ 3];t4=t3+d[i+ 4];t5=t4+d[i+ 5];                    /* 1 */
     t6=d[i+ 6];t7=t6+d[i+ 7];t8=t7+d[i+ 8];                    /* 1 */
     t9=d[i+ 9];ta=t9+d[i+10];tb=ta+d[i+11];                    /* 1 */
     for(i=12;(n-=12)>=0;i+=12){
          /* b */
          a[i-10]=sum2=t2+sum1;a[i-11]=t1+sum1;a[i-12]=t0+sum1;  /* 2 */
          a[i- 7]=sum1=t5+sum2;a[i- 8]=t4+sum2;a[i- 9]=t3+sum2;  /* 2 */
          a[i- 4]=sum2=t8+sum1;a[i- 5]=t7+sum1;a[i- 6]=t6+sum1;  /* 2 */
          a[i- 1]=sum1=tb+sum2;a[i- 2]=ta+sum2;a[i- 3]=t9+sum2;  /* 2 */
          t0=d[i+ 0];t1=t0+d[i+ 1];t2=t1+d[i+ 2];                /* 1 */
          t3=d[i+ 3];t4=t3+d[i+ 4];t5=t4+d[i+ 5];                /* 1 */
          t6=d[i+ 6];t7=t6+d[i+ 7];t8=t7+d[i+ 8];                /* 1 */
          t9=d[i+ 9];ta=t9+d[i+10];tb=ta+d[i+11];                /* 1 */
     }
     a[i-10]=sum2=t2+sum1;a[i-11]=t1+sum1;a[i-12]=t0+sum1;       /* 2 */
     a[i- 7]=sum1=t5+sum2;a[i- 8]=t4+sum2;a[i- 9]=t3+sum2;       /* 2 */
     a[i- 4]=sum2=t8+sum1;a[i- 5]=t7+sum1;a[i- 6]=t6+sum1;       /* 2 */
     a[i- 1]=sum1=tb+sum2;a[i- 2]=ta+sum2;a[i- 3]=t9+sum2;       /* 2 */
                                                        /* c */
rest:n+=12;
     for(  ;--n>0;i++){
          a[i]=sum1=sum1+d[i];
     }
}
```

図8 ソフトウェアパイプライニング版
Fig. 8 Software pipelining version.

並列性が導入されるが，フロー依存に伴う待ちが完全に回避されるわけではないので，スーパースカラプロセッサであってもこのループイタレーションを1回当たり20クロック，したがって全体処理を1666クロックで実行することはできない．

なお，このプログラムの演算構造は**図7**の通りである．あらかじめの計算部はステージ1に対応する．

4.2 ソフトウェアパイプライニング版

図7の演算構造に対し，ソフトウェアパイプライニングを適用したコードを**図8**に示す．なお，ここでのソフトウェアパイプライニングとはステージ1の初回をループの外に（前に）追い出し，前回のステージ2と今回のステージ1でループ本体を構成し，ステージ2の最終回をループの外に（後ろに）に追い出すループ再構成を意味する．ループ本体の演算構造図は図10のようになる．この図で，演算チェイニングのための中継レジスタ（t0, t1, …）への参照がフロー依存（定義後の使用）の関係でなく，逆依存[5]（使用後の再定義）の関係になっている点に注目のこと．

図8のループ本体部について補足説明をしておく．

（a）まず，部分和計算部のステージ2（前回のイタレーションの後半部）を先に実行する．これにより演算チェイニングのための中継レジスタ（t0, t1, …）の使用を完了させる．

（b）次に，あらかじめの計算部のステージ1（今回のイタレーションの前半部）を実行する．実行の結果は演算チェイニングのための中継レジスタに定義される．

（c）逆依存の制約に反しない（中継レジスタへの定義が使用を追い越さない）範囲で，命令の並べ替えが可能である．これにより2並列のスーパースカラでステージ2の命令一つとステージ1の命令一つを1クロックごとに実行可能となる．

（d）素朴には4段のパイプライニングかとも思えるが，ステージ1がロード，加算の繰り返し，ステージ2が加算，ストアの繰り返しであり，2並列を狙うのであれば，2段のパイプライニングで良い．

（e）キャッシュへの事前フェッチ[6]を考えれば，地点（a）でのn−=12をn−=13に，地点（c）でのn+=12をn+=13にするほうが良い．この場合，地点（b）でx[i+12], y[i+12]を事前ロードしても問題は全く発生しない．

4.3 機械語イメージ版

図8のプログラムは作業レジスタと演算の実行順序を陽には指定していなかった．さて次に，作業レジスタと演算の実行順序を陽に指定したソースコードを**図**

図9 演算順序と論理レジスタの生存区間
Fig. 9 Computation order and live ranges of logical registers.

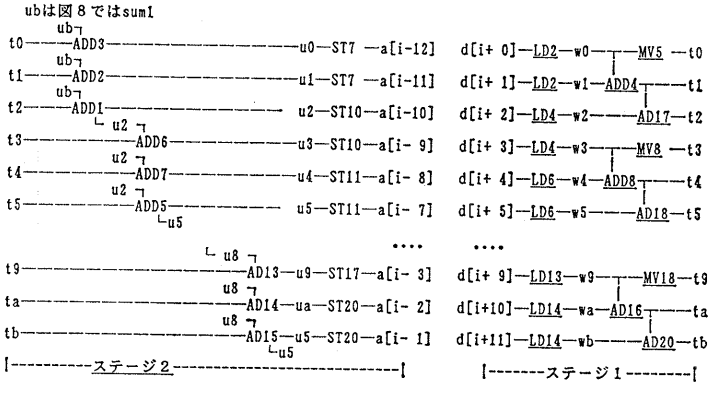


図10 演算構造
Fig. 10 Computation structure.

表 1 物理レジスタの割当て
Table 1 Physical register assignment.

物理	論理	物理	論 理	物理	論理	物理	論理
FR 00		FR 08	u4 ua	FR 16	w8	FR 24	t4
FR 01		FR 09	u5 ub suml	FR 17	w9	FR 25	t5
FR 02		FR 10	w0 w6 wa	FR 18		FR 26	t6
FR 03		FR 11	w1 w7 wb	FR 19		FR 27	t7
FR 04	u0 u8	FR 12	w2	FR 20	t0	FR 28	t8
FR 05	u1 u9	FR 13	w3	FR 21	t1	FR 29	t9
FR 06	u2 u6	FR 14	w4	FR 22	t2	FR 30	ta
FR 07	u3 u7	FR 15	w5	FR 23	t3	FR 31	tb

9に示すことにする（ループ本体部のみ）．図9の個別の行に与えた番号はクロックに対応する．

また演算構造図に陽に作業レジスタ（論理レジスタ[7] w0, w1, u0, u1など）を指定したものが**図10**である．

図9について少し説明しておく．

（1）本来のループ，加算は1000個であるが，ロード，ストアもおのおの1000個であり，全体を1666クロックで実行するには一度に2要素のLD/STが必然となる．// はこのためのロケーションの連結を示す．

なお，floatデータ二つをメモリとレジスタの間で一括転送する命令は各種のアーキテクチャで提供されている[8]．

（2）配列要素の参照は a[i] という形で表現してあるが，実際には *a++ の形を想定している．そしてベースレジスタやインデックスレジスタ更新のための余分な命令が不要な命令体系を想定している[8]．

したがって，ロード/ストア/加算以外に必要となる命令は条件分岐一つ程度の想定となる．

（3）物理レジスタ割当ては例えば**表1**の通り．

このレジスタ割当てでよい点を示すために，個別論理レジスタの生存区間[7]を図9の右側に示している．

この図において，網掛け部はイタレーションをまたぐ生存区間である．

（4）図9を見れば，作業レジスタ（w0, w1, …, u0, u1, …）とチェイニング（パイプライニング）のための中継レジスタ（t0, t1, …）の違いが明確になる．チェイニングのための中継レジスタは生存区間が当然イタレーションをまたぐことになる．

一方，作業用のレジスタはイタレーションをまたがない（ubは別）．

（5）イタレーションをまたぐすべての論理レジスタはイタレーションの開始時点ですべてに生きているので，それら二つの論理レジスタは物理レジスタを共有できない．一方，生存区間の短い作業レジスタは他の作業レジスタや中継レジスタと物理レジスタを共有できる．

（6）ステージ2では，中継レジスタの早期の使用完了が好ましい．一方ステージ1では，中継レジスタへの遅い定義が好ましい．

図9を見れば，図8のループ本体部が20クロック程度で実行できる点が納得できよう．となると1000要素の部分和計算が $1666+\alpha$ で計算できると期待できる．

5. 演算当たりのクロック数

リバモアループ1と11の性能を**表2**に示す．数値は演算当たりのクロック数（clocks/flop）であり，周波数の違いは現れてこない．ループ1のスーパースカラプロセッサの値は，加算器と乗算器を並列動作可能とした場合の単なる推定値である．また，表2中の二つのベクタプロセッサは加算器と乗算器をともに四つ以上持っている．ベクタ1はリカレンス高速化を図っている．ベクタ2はそうでない．

ループ1は最も高速化できるループである．二つのベクタプロセッサはかなり高速といえる．八つ以上の演算器をフルに活用できているからである．

一方ループ11は，本稿で着目しているループであり部分和を計算する．スカラプロセッサに本稿で提案の方式を採用した値（1.7）が最高で，ベクタ1の値よりもよい．ベクタ1はリカレンス対策を施しているのでかなり高速化（2.59）されているが，ベクタ2はそうでないのでかなり遅い（6.87）といえる．

ベクタプロセッサの加算完了クロックは4以上と推定され，一方ここで想定したスカラプロセッサの加算

表 2 リバモアループ性能 (clocks/flop)
Table 2 Performance of livermore loops.

	ベクタプロセッサ		スーパースカラプロセッサ	
ループ	ベクタ1	ベクタ2	従 来	提 案
ループ1	0.25	0.22	0.5-1.0	0.5-1.0
ループ11	2.59	6.87	4.0	1.7

完了クロックは4であり，その意味でベクタプロセッサとの単純な比較はできない．しかし，ベクタプロセッサではリカレンスがかなり遅くなる点と，本稿で提案の方式がかなり有効である点が納得できよう．

なお，本稿の提案方式と同様な考えで式(2)のリカレンスも2倍弱の高速化が可能である．

6. おわりに

演算パイプラインプロセッサの並列性を阻害する容因としてリカレンスがある．リカレンスは素朴には逐次的な実行を強要する．この逐次性が並列化を阻害し，したがって高速化の妨げになる．

本稿では，スーパースカラプロセッサ向けのリカレンス高速化方式を提案した．高速化の具体例はリバモアループの11番とした．

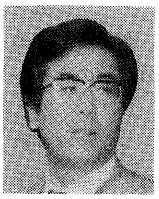
マルチプロセッサ向けのリカレンス高速化方式として巡回縮約が知られている．しかし巡回縮約はスーパースカラ向け（シングルプロセッサ向け）の高速化方式にはならない．並列性増大の利点よりも演算数増大の欠点のほうが大きく効いてくるからである．

本稿で提案の方式は巡回縮約にヒントを得ている．本稿の提案では巡回縮約の余分な演算数の増大を抑えるとともに，スーパースカラで活用できる程度の並列性を引き出した．これによりスーパースカラプロセッサにおいてリバモアループの11番を2倍以上高速化できた．

参考文献

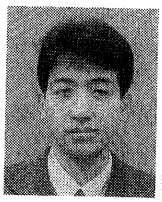
1) Hockney, R. W. et al.: *Parallel Computer 2*, Adam Hilger (1988).
2) Tanaka, Y. et al.: Compiling Techniques for First-order Liner Recurrence on a Vector Computer, *IEEE Supercomputing '88*, pp. 174-181 (1988).
3) Johnson, M.: *Superscalar Microprocessor Design*, Prentice Hall (1991).
4) Banerjee, U.: *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers (1988).
5) Padua, D. A. et al.: Advanced Compiler Optimizations for Supercomputers, *C. ACM*, Vol. 29, No. 12, pp. 1184-1202 (Dec. 1986).
6) Callahan, D.: Software Prefetching, *Proc. ASPLOS-IV*, pp. 40-52 (1991).
7) Aho, A. V. et al.: *Compilers Principles, Techniques and Tools*, Addison-Wesley (1986).
8) Precision Architecture and Instruction Reference Manual, Part No. 09740-90214 (Jun. 1987).
9) スーパーコンピューター製品・技術・応用，日経 BP (1989).

(平成5年4月2日受付)
(平成5年7月8日採録)



海永 正博（正会員）

1970年東京工業大学理工学部数学科卒業．1972年同学科修士課程修了．1973年(株)日立製作所中央研究所入社．現在同社システム開発研究所．研究テーマ：最適化コンパイラ，超並列化コンパイラ．訳書「Cプログラミング」．日本ソフトウェア科学会会員．



久島伊知郎（正会員）

1986年東京大学理学部情報科学科卒業．同年(株)日立製作所に入社．以来同社システム開発研究所．研究テーマ：プログラミング言語，コンパイラ．