リカレンスをもつループのスーパースカラ向け高速化方式

6E−1

久島　伊知郎　　海永　正博
(株) 日立製作所　システム開発研究所

1．はじめに

演算パイプラインプロセッサの並列処理を阻害する要因としてリカレンスがある。リカレンスは値の定義/使用（フロー依存）の長い系列であり、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するわけである。

本稿では、リカレンスの代表例であるLivermore Loop kernel 11をスーパースカラプロセッサ向けにいかに高速化するかを述べる。スーパースカラプロセッサにおいてこのループを2倍以上高速化できる、というのが本稿の主要な主張点である。

2．リカレンス

リカレンス（漸化式と訳されることもある）とは、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。以下の式はすべてリカレンスの例となる。

$a_{i+1} = a_i + d_i$　　（式1）
$a_{i+1} = a_i + c_i * d_i$　　（式2）
$a_{i+1} = c_i * a_i + d_i$　　（式3）

式1は（逐次）部分和と呼ばれている。数列$\{d_i\}$の部分和を逐次に計算しているからである。式2は部分積和、式3は線形1次リカレンスなどと呼ばれる。

Livermore Loopの kernel 11は以下のループである。

```
for ( k=1 ; k<n ; k++ ) {
  x[k] = x[k-1] + y[k];
}
```

図1 Livermore Loop kernel 11

x[k]をa_{i+1}、y[k]をd_iに対応させれば、kernel 11は式1の部分和に対応したループとなる。なお、kernel 5およびkernel 6もリカレンスである。

3．巡回縮約

部分和（式1）を式どおり計算すれば、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。要素の数をn個とし、1回の加算の完了に4クロックかかるマシンでこれを実行しようとすると、全体で少なくとも4*nクロックかかることになる。

しかし式1を1度展開すれば以下のようになる。

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

これにより、リカレンスの系列が1つから2つになり、また系列の長さが半分になる。個別の系列は独立に計算することができるので、並列性が引き出せる（このよう手続きは巡回縮約(cyclic reduction)と呼ばれる[1]）。再度同じ手続きを踏めば2つの系列から4つの系列を生成できる。4つの系列にしたときの演算構造を概念的に示すと図2のようになる（図で$S_{i..j}=d_i+d_{i+1}+...+d_j$）。

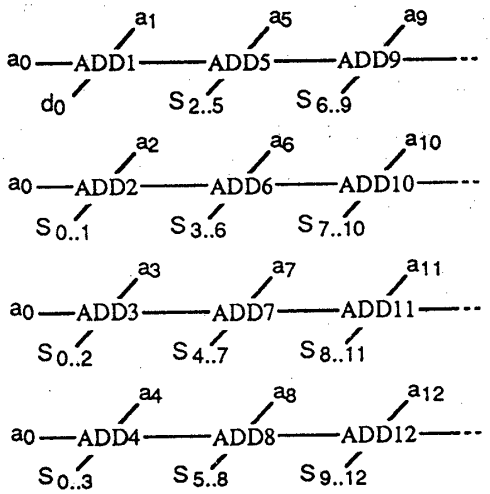

図2　4倍巡回縮約した場合の演算構造

いま図2で、$S_{0..1}$, $S_{0..2}$, ...などの（余分の）計算が既になされているとする。するとそれぞれの系列に属する加算を順に（ADDの番号順に）実行していけば部分和が計算できる。しかも加算の完了に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。即ち、個別の加算を1クロックごとに実行できることになり、余分の計算を無視できるのであれば、全体をnクロックで実行できることになる。

しかし、余分の計算を無視することは本当はでき

Fast Execution of Loops with Recurrences on Superscalar Processors
Ichiro KYUSHIMA, Masahiro KAINAGA
Systems Development Laboratory, Hitachi Ltd.

ない。というのは余分に計算しておく項目は全部でn個あり、それぞれに3個の加算が必要であり、結局のところそれだけで3*nクロックかかってしまうからである。そこで、新しい計算方法を提案する。

4．計算方法

新しい計算方法の演算構造を図で表現すれば以下のようになる。

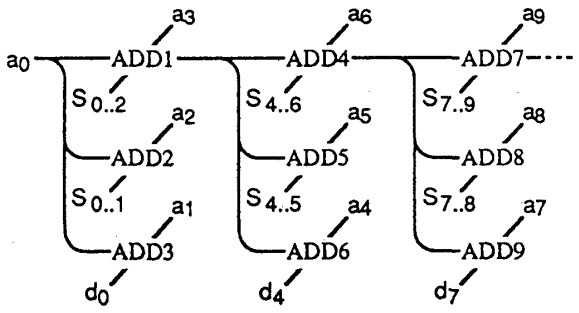

図3　新しい計算方法の演算構造

先ほどと同様にそれぞれの系列に属する加算を順に行っていけば、フロー依存に伴う待ちは3回に1回発生する（加算の完了に4クロックという想定）。すなわち個別の加算を4/3クロック毎に実行できることになり、全体で4/3*nクロックとなる。一方、余分の計算は2/3*n個で、フロー依存の意味で独立な計算がn/3個であり、明らかに2/3*nクロックで実行できる。よって全体の計算を2*n(=4/3*n+2/3*n)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余分の計算の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。すなわち5/3*n(=n+2/3*n)クロックで実行される。

5．部分和の高速計算関数

新しい計算方法をCで記述したものが図4である。これはループアンローリングおよびソフトウェアパイプライニング[2]による最適化を適用した後のプログラムである。ループアンローリングによって1回のループで12要素分（図の4列分）計算するようにし、さらにソフトウェアパイプライニングにより、今回の繰り返しの部分和計算の部分と、次回の繰り返しの余分の計算部分をオーバーラップして実行できるようにしている。

図4のプログラムを、実際のマシン（PA-RISC[3]）を2-scalar化した机上のスーパースカラプロセッサ用にアセンブリコーディングすると、ループ1回あたり約20クロックで実行できる。これは要素1個あたり5/3クロックにあたる。このマシンでは加算の完了に4クロックかかるので、素朴な計算方法に比べて2倍以上の高速化が図られたことになる。

6．おわりに

本最適化をコンパイラで実現する場合には、部分和計算を行うループをパタンマッチで認識し、念入りにアセンブリコーディングされたライブラリ関数を呼び出すことになろう。なお、kernel 5と6についても同様の高速化が適用できる。

参考文献

[1] R.W.Hockney and C.R.Jesshope, *Parallel Computers 2*, Adam Hilger, 1988.

[2] M.Lam, "Software Pipelining: An Effective Scheduling Techniques for VLIW Machines," *ACM SIGPLAN*, 1988.

[3] *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, Hewlett-Packard Co., 1990.

```
sum1=0;
t0=   d[i+0]; t3=   d[i+3]; t6=   d[i+6]; t9 =    d[i+9];
t1=t0+d[i+1]; t4=t3+d[i+4]; t7=t6+d[i+7]; t10= t9+d[i+10];
t2=t1+d[i+2]; t5=t4+d[i+5]; t8=t7+d[i+8]; t11=t10+d[i+11];
for (i=12; (n-=12) >= 0; i+= 12) {
    a[i+2] =sum2= t2+sum1; a[i+1] =t1+sum1;  a[i+0]=t0+sum1;
    a[i+5] =sum1= t5+sum2; a[i+4] =t4+sum2;  a[i+3]=t3+sum2;
    a[i+8] =sum2= t8+sum1; a[i+7] =t7+sum1;  a[i+6]=t6+sum1;
    a[i+11]=sum1=t11+sum2; a[i+10]=t10+sum2; a[i+9]=t9+sum2;
    t0=   d[i+0]; t3=   d[i+3]; t6=   d[i+6]; t9 =   d[i+9];
    t1=t0+d[i+1]; t4=t3+d[i+4]; t7=t6+d[i+7]; t10=t9+d[i+10];
    t2=t1+d[i+2]; t5=t4+d[i+5]; t8=t7+d[i+8]; t11=t10+d[i+11];
}
a[i+2] =sum2= t2+sum1; a[i+1] =t1+sum1;  a[i+0]=t0+sum1;
a[i+5] =sum1= t5+sum2; a[i+4] =t4+sum2;  a[i+3]=t3+sum2;
a[i+8] =sum2= t8+sum1; a[i+7] =t7+sum1;  a[i+6]=t6+sum1;
a[i+11]=sum1=t11+sum2; a[i+10]=t10+sum2; a[i+9]=t9+sum2;
```

図4　部分和計算プログラム