計算機アーキテクチャ　96－17
（1992. 10. 23）

擬似ベクトルプロセッサによるリストベクトル処理とその評価

位守弘充＊　中村宏　朴泰祐　中澤喜三郎

＊筑波大学大学院工学研究科

筑波大学電子情報工学系

〒305 茨城県つくば市天王台1-1-1

あらまし　大規模な科学技術計算分野のアプリケーションにおいて要求されるデータ領域は大きく、時間的局所性が少ないためにキャッシュメモリは有効に働かない。そのためスカラプロセッサでは主記憶とのデータ転送が頻繁に発生し、実効性能が著しく低下することが多い。この解決策として、データキャッシュの代りに浮動小数点レジスタウィンドウを用い、スーパースカラ処理方式の利点を活用することにより、スカラプロセッサでありながら、高速にベクトル処理を行なう擬似ベクトルプロセッサを我々は提案している。本稿では従来より提案している擬似ベクトルプロセッサに新たな機能追加として汎用レジスタのレジスタウィンドウ化を行なうことで、既存のアーキテクチャと上位互換性を保ちながらリストベクトル処理を効率よく扱う手法を提案する。また、実行時でないと依存関係が判明しないため、ベクトル計算機ではベクトル化できないリストベクトル処理をも連想レジスタを用いることで効率よく処理する手法も提案する。これらの手法をベンチマークを用いて評価した結果を合わせて示す。

和文キーワード　擬似ベクトルプロセッサ　リストベクトル処理　スーパースカラ方式　レジスタウィンドウ　レジスタプリロード　パイプラインメモリ

List Vector Processing on the Pseudo Vector Processor and Its Performance Evaluation

Hiromitsu IMORI*　Hiroshi NAKAMURA　Taisuke BOKU　Kisaburo NAKAZAWA

*Doctoral Program of Engineering,University of Tsukuba

Institute of Information Sciences and Electronics,University of Tsukuba

1-1-1,Tennodai,Tsukuba,Ibaraki,305,Japan

Abstract　In engineering/scientific applications, cache memory is not effective because a large amount of data is accessed with little temporal locality.Therefore, performance of ordinary scalar processors degrades seriously in these applications. We have proposed the Pseudo Vector Processor based on Floating-point Register Window which avoids this performance degradation,. Instead of data cache, floating-point register window is adopted in the proposed architecture. The processor can realize high-speed vector processing based on a superscalar pipeline, though it is a scalar processor.

In this paper, we present a enhanced pseudo vector processor architecture for effective list vector processing which also introduce window structure for the general registers. The new architecture still holds upward compatibility with existing scalar architectures. We also describe an extension of the processor for dynamic list vector handling which can not be vectorlized in ordinary vector processors due to dynamic data dependencies. An associative register is introduced in this extension. The evaluation result of the proposed architecture is described.

英文 key words　Pseudo Vector Processor　List Vector Processing　SuperScalar　Register Window　Register Preloading　Pipelined Memory

1. はじめに

近年のスカラプロセッサにおいては、スーパースカラ方式などプログラムの並列性を抽出して高速に処理を行なう処理方式が実用化され、また集積回路技術の進歩によりクロック周波数が年々高くなっているため、その処理能力は著しく向上している。一方、データ供給を行なう主記憶については容量は飛躍的に増加しているものの、アクセスタイムはさほど改善されていない。このため主記憶アクセスレーテンシーがプロセッサに与える影響は相対的に大きくなっている。

この問題に対して、スカラプロセッサではデータの局所性を利用したキャッシュメモリにより対処している。しかしベクトル計算機が扱う科学技術計算分野では、データ領域が非常に大きく、データの時間的局所性が少ない等の理由によって、データキャッシュが有効に働かないことが多い[6]。この時はデータキャッシュにより高性能を達成しているスカラプロセッサでは主記憶アクセスペナルティによる実効性能の低下が著しい。

この問題に対する解決法として、プリフェッチ機能をキャッシュメモリに追加し、必要なデータを予めキャッシュメモリに格納しておくことにより主記憶へのアクセスペナルティーを軽減する手法が提案されている[7,8]。また、小容量のベクトルレジスタを備え、マルチスレッド機能等を追加し、load/store pipelineを強化して演算パイプラインを切れ目なく有効に稼働させようとするマイクロベクトルアーキテクチャも提案されている[9,10]。

前者においては、データに時間的局所性がない場合、プリフェッチされたデータは一度利用された後再利用性がないために主記憶に戻される。そのためキャッシュメモリは主記憶からのデータの転送バッファにすぎなくなる。また非連続アドレスをアクセスする場合、データをキャッシュにプリフェッチする時、不要なデータも同時にプリフェッチする可能性があり、不必要なデータ転送が生じてしまう。これらの理由によりキャッシュメモリに要求されるスループットは厳しくなってしまうという問題点がある。

後者についてはベクトル命令とベクトルレジスタを備えることにより既存のスカラアーキテクチャとの上位互換性が保てないという問題点がある。

これらに対し我々は、スーパースカラ処理方式の利点を活用して、既存のスカラプロセッサと上位互換性を保ちながら、主記憶アクセスペナルティーを軽減し、ベクトル計算を高速に処理する擬似ベクトルプロセッサを提案している[2,3,5]。

本稿では、提案している擬似ベクトルプロセッサをさらに拡張して、リストベクトル処理をも効率よく行なう手法を提案する[4]。このための機能追加として汎用レジスタのレジスタウィンドウ化を新たに行なう。さらに、ベクトル計算機ではベクトル化できないような、動的に依存関係が決定されるリストベクトルに対する処理手法も提案する。

これにより擬似ベクトルプロセッサは上位互換性を保ちながら、大規模科学技術計算でよく用いられるリストベクトル処理を効率よく扱うことができる。

2. 擬似ベクトル処理

ベクトル計算機がスカラプロセッサと異なる特徴としては、

- 主記憶がマルチバンク構成によりパイプライン化されているため、主記憶からのデータ転送をベクトル命令で高速に処理できる。
- 大容量のベクトルレジスタを持ち、ベクトルロード/ストア命令とchaining機能によりこれを用いることで主記憶へのアクセスペナルティーを隠し、一種のプリロード機能の役割を担っている。

擬似ベクトルプロセッサではこれらの機能をスカラプロセッサに、

- 主記憶のパイプライン化
- 浮動小数点レジスタのレジスタウィンドウ化
- レジスタへのプリロード機能(プリロード命令の導入)

という形で追加することで上位互換性を保ちながら、ベクトル計算機と等価な処理を行なっている。

科学技術計算において最も処理能力を必要とするのはループ処理であり、各ループの実行は概念的にはロード/演算/ストアという3つのphaseに分割できる。擬似ベクトルプロセッサでは図1に示すようなレジスタウィンドウ構成を用いて、各phaseをpipeline的に処理すること(図2)で高速化を図っている(phase pipelining)。

例えばi loopの実行は図2のwindow 2のレジスタ空間を用いて実行されるとする。この時現在activeであるウィンドウを示すCFRWP(Current Floating Register Window Pointer)はwindow 2を指しているとする。

この処理において以下の3つのphaseが3つのレジスタウィンドウを用いてスーパースカラ方式により並行して実行される。

- 次のi+1 loopで用いられるデータを次ウィンドウ(window 3)にロードする命令を発行する(preload phase)。
- i loopの実行を現在activeであるウィンドウ(window 2)で処理する(execution phase)。
- i-1 loopで計算されたデータを前ウィンドウ(window 1)からストアする(poststore phase)。

i loop終了時に、CFRWPをwindow 2からwindow 3に切り替えることにより、i+1 loopの実行時には、active windowであるwindow 3に既に必要なデータが、i loopでのpreload phaseにより到着しているので、execution phaseを連続的に実行することが可能となる。

擬似ベクトルプロセッサでは、このようにスーパースカラ処理方式を利用してデータ供給と演算実行を並列に実行することで、演算パイプラインを切れ目なく稼働させることができる。これにより1つのベクトル命令の処理を、複数のスカラ命令で垂直マイクロプログラミング的に実行することで、上位互換性を保ちながらベクトル処理と等価な処理が可能となる。(擬似ベクトル処理[1])

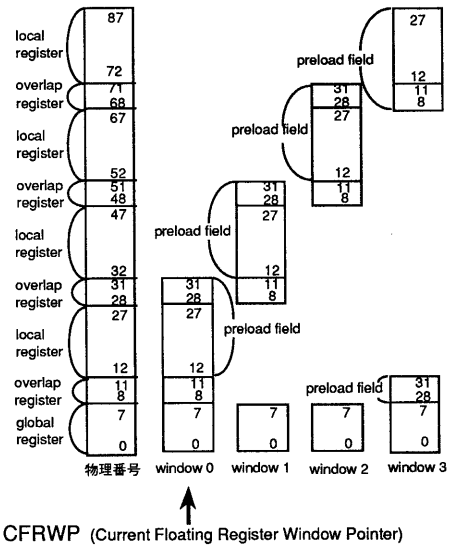


図1 レジスタウィンドウの構成

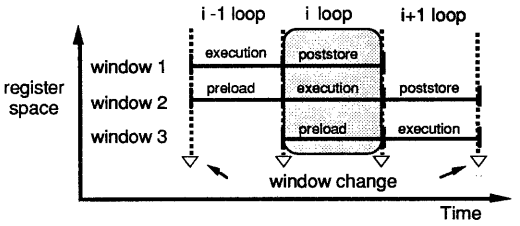


図2 擬似ベクトル処理の原理

3. リストベクトル処理

3.1. リストベクトル

科学技術計算において、例えば非常に大きなsparse matrix問題では通常0でない要素のみを主記憶に格納し、そのアドレスを保持するベクトルを用意している。このようなベクトルをリストベクトルという。

ここでは以下のようなリストベクトル演算を例に挙げ説明する。

・A(L(i)) = A(L(i)) + const

この演算において扱うデータが長大である場合、キャッシュメモリが有効に働かず、通常のスカラプロセッサではL(i)、A(L(i))という2回の主記憶への読みだし要求が生じることになり、性能低下が著しい。

また、この例ではリストベクトルL(i)が式の両辺に存在するため、配列Aに関してL(i)の値によって動的に依存関係が生じる可能性があり、通常のベクトル計算機でもベクトル化できていない。

依存関係の生じる場合の対処については4章で述べることとし、本章ではまず依存関係がない場合の処理について述べる。

3.2. phase pipeliningの拡張

擬似ベクトルプロセッサにおいてリストベクトル処理を効率よく処理するために、2章で述べたphase pipelingの拡張を行なう。

リストベクトル処理のphase pipeliningでは汎用レジスタのレジスタウィンドウ化を新たに行なう。先に例で挙げた演算は、このウィンドウ化された汎用レジスタにリストベクトルL(i)をプリロードすることにより効率よく処理できる。

このウィンドウの構造は、図1に示した浮動小数点レジスタウィンドウと同一のものとし、現在activeであるウィンドウナンバーも一致しているものとすれば、リストベクトル処理は以下のように行なわれる(図3)。

今、i loopの計算に必要なデータが既に浮動小数点レジスタウィンドウ側のwindow j(f-window jとする)に格納されているものとする。スーパースカラ方式によりこのi loopの計算(execution phase)と並列に以下のload/store phaseの実行を行なう。

(1)GRlistload phase
2つ先の汎用レジスタウィンドウ(g-window j+2)にi+2 loopで用いるL(i+2)をロードする。

(2)FRlistload phase
1つ先のg-window j+1へ既にロードされているL(i+1)を用いて、f-window j+1へA(L(i+1))をロードする。

(3)FRliststore phase
1つ前のg-window j-1にあるL(i-1)を用いてA(L(i-1))を1つ前のf-window j-1からストアする。

これら各phaseが図3に示すように異なるレジスタウィンドウ空間を利用しながら1ループを構成し、1ループ終了ごとに汎用/浮動小数点レジスタウィンドウ両方を同時に切り替えることで、図4に示すphase pipeliningを実現できる。

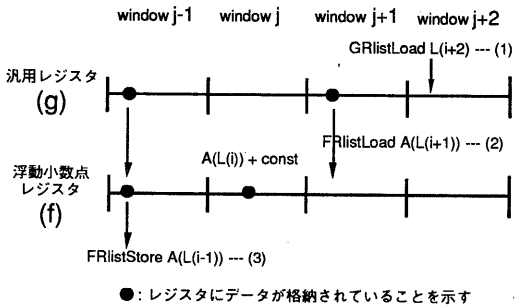


図3 リストベクトル処理の原理

3.3. 命令セットアーキテクチャの拡張

3.2で述べたphase pipeliningの実現のために、従来の擬似ベクトルプロセッサに命令セットアーキテクチャ上、新たに追加する命令を以下に示す。

これらの命令を追加するだけで上位互換性を保ちながら効率よくリストベクトルを処理できる。

・GRlistLoad：2つ先の汎用レジスタウィンドウの指定レジスタに主記憶の指定アドレスよりデータをロードする。キャッシュヒット時にはキャッシュメモリにあるデータを転送するが、キャッシュミス時には、キャッシュメモリ内のデータを更新することなく、主記憶からデータを転送する。
・FRlistLoad：1つ先の汎用レジスタウィンドウの指定レジスタの内容をアドレスとして(間接アドレス)、1つ先の浮動小数点レジスタウィンドウ内の指定レジスタに主記憶からデータをロードする。キャッシュメモリとの関係は上記命令と同様である。
・FRlistStore：1つ前の汎用レジスタウィンドウの指定レジスタの内容をアドレスとして、1つ前の浮動小数点レジスタウィンドウ内の指定レジスタのデータを主記憶へストアする。キャッシュメモリとの関係は上記命令と同様である。
・CFRWPinc：従来からある命令であり、ウィンドウナンバーのインクリメントを行なう。

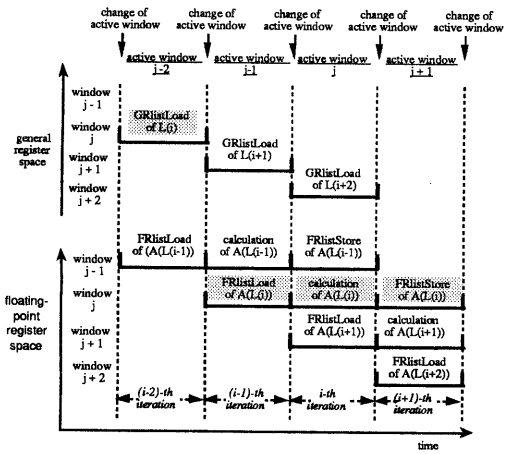


図4 リストベクトル処理のphase pipelining

3.4. オブジェクトコード例

例に挙げたリストベクトル演算のオブジェクトコードは、3.3で述べた命令を用いることにより、図5のようになる。左側はオブジェクトコードを示し、右側は対応する各命令の動作を示している。

```
                <<object code>>              <<operation>>
Loop:  GRlistLoad 8(GR3), GR20      (GR<j,3>) ->GR<j+2,20>
                                    (GR<j,3>) <- (GR<j,3>) + 8
       FRlistLoad (GR20), FR20      (GR<j+1,20>) -> FR<j+1,20>
       FADD FR20, FR5, FR20         FR<j,20> + FR<j,5> -> FR<j,20>
       FRlistStore (GR20), FR20     FR<j-1,20> -> (GR<j-1,20>)
       Branch Loop
       CFRWPinc
```

FR5: const.　GR3: L(i)へのポインタ　j: 現在activeなウィンドウ
GR<j ,3> はglobal register window jの汎用レジスタGR3を示す

図5 A(L(i))=A(L(i))+constのコード例

4.動的に依存関係が決定するリストベクトルへの対処

3章で述べた仕様によって通常のベクトル計算機がサポートする程度のリストベクトル演算は実行可能となる。すなわち、あらかじめデータ依存関係がないことが保障されている場合については、リストベクトル演算は問題なく実行できる。しかしながら現在のベクトル計算機においては、リストベクトルをハードウェア的に一応サポートしているものであっても、先の例のように式の両辺にリストベクトルL(i)が存在することなどにより実行時にしかRAW(Read After Write)conflictが判明できないリストベクトルは、ベクトル化されていない。同様の理由により図6-(a)に示すLFK(Livermore Fortran Kernel loop)14もベクトル化されていない。

このような処理は、3章で示した擬似ベクトルプロセッサにおいても以下の2つの理由により、擬似ベクトル化できない。

(A)ソースプログラムにおいてunrolling手法を用いる場合、オブジェクトコード中の1ループ内のiteration間でRAW conflictが生じる可能性がある。(図7-(a))
(B)phase pipeliningを行なう場合、FRlistload phaseとFRliststore phaseの実行順序が逆転してしまうため、RAW conflictが生じる可能性がある。(図7-(b))

これらの問題への対処として、(A)に対してはソフトウェアにより、(B)に対してはさらなる機能追加により解決する手法をここでは述べる。以降図6に示すLFK 14を例として説明する。

```
      DO 14    k= 1, n
      RH(IR(k)  ) =RH(IR(k)  ) + fw - RX(k)
      RH(IR(k)+1) =RH(IR(k)+1) + RX(k)
14    CONTINUE
```

図6 -(a)

```
      DO 14    k= 1, n
      RH (IR(k  )  ) =RH  (IR(k)  ) + fw - RX(k)
      RHA(IR(k+1)  ) =RHA(IR(k+1)  ) + fw - RX(k+1)
      RHB(IR(k+2)  ) =RHB(IR(k+2)  ) + fw - RX(k+2)
      RH (IR(k  )+1) =RH  (IR(k)+1) + RX(k)
      RHA(IR(k+1)+1) =RHA(IR(k+1)+1) +RX(k+1)
      RHB(IR(k+2)+1) =RHB(IR(k+2)+1) +RX(k+2)
14    CONTINUE

      DO 143   I= 1, m
      RH(I) =RH(I) + RHA(I) + RHB(I)
143   CONTINUE
```

図6 -(b)

図6 LFK14(1D-PIC)

4.1. unrolling手法によるRAW conflictへの対処

図6-(a)で示したソースコードにおいて1ループ内のiteration間で依存関係が生じないようにunrolling回数分の異なる配列を用意する。すなわち図6-(b)のようにソースコードを変換し、演算終了後全ての配列の総和をとるようにする[9]。この結果、unrolling手法を用いることによるRAW conflictは解消できる。

4.2. phase pipelining手法による RAW conflictへの対処

図7-(b)中の枠で囲んだFRlistload phaseで、RAW conflictを調べる必要があるのは、矢印が指す2つのFRliststore phaseのみであり、それ以前のFRliststore phaseとのconflictは生じない。phase pipeliningを有効に実現するために、これらのRAW conflictへの対処として以下のような機能追加を行なう。

4.2.1. FASR(Floating ASsociative Register)

図7-(b)で示すように、順序が逆転して発行されたFRlistLoad命令のオペランドアドレスと、その後に発行されるFRlistStore命令のディスティネーションアドレスとが一致する場合、RAW conflictが生じていることになる。この時、FRlistLoad命令によりプリロードされるべきデータは、時間的に後に発行されるFRlistStore命令でストアされるべきデータである。従って、既に発行されたFRlistLoad命令によるメモリからのデータを破棄し、FRlistStore命令によってストアされるべき新しいデータを代りにforwardingしてくることによりRAW conflictは解決できる。この機能を実現するために図8に示すような構造をもつ連想レジスタを用いる。以降この連想レジスタをFASRと呼ぶこととする。

FASRの各エントリは4つのfieldで構成される。W fieldはウィンドウ番号、R fieldは論理レジスタ番号、op.add fieldはオペランドアドレスを示す。f fieldは制御flagとして、f1とf2を格納する。

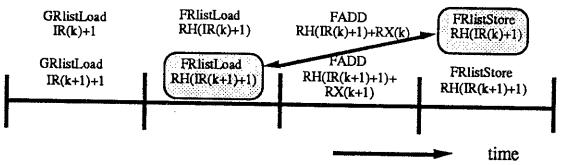


図7-(a) (A)loop unrollingによるRAW conflict

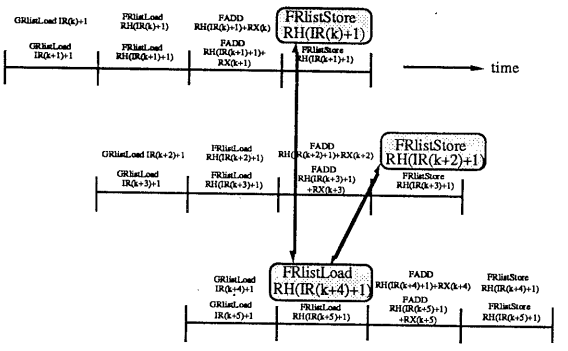


図7-(b) (B)phase pipeliningによるRAW conflict

図7 擬似ベクトルプロセッサにおける RAW conflictの可能性

f1はその対応するエントリの情報が有効であるかを示し、f2はRAW conflict発生時にデータがforwardingされたことを示す。FRlistLoad命令発行時には、f1が0である無効エントリを1つ選択し、そのエントリのW field、R fieldにデスティネーションレジスタの論理レジスタ番号、オペランドアドレスを登録し、op.add fieldにソースオペランドのアドレスを登録する。そしてf1 に1をセットし、f2をリセットする。RAW conflictは、このように登録されたオペランドアドレスに対し、後からFRlistStore命令によるストアが実行される場合に生じる。この場合、登録されているレジスタにストアすべきデータをforwardingし、そのエントリのf2を1にセットする。

簡単に例を述べる。現在activeであるウィンドウをwindow i (CFRWP=i)とする。

図9の時間軸上、T1、T2、T3、T4それぞれの時間において以下の命令が発行されたとする。

T1において　FRlistLoad FR20 (address1)
T2において　FRlistStore FR15 (address2)
T3において　FADD FR30 FR20+FR28
T4において　CFRWP window change

各時間におけるFASRの処理は、

T1:FRlistLoad命令発行時に、FASR内のf1=0である任意のentry(以降X-entryとする)にW ← i+1、R ← 20、op.add ← (address1)を登録し、このエントリが有効になったことを示すf1を1にする。

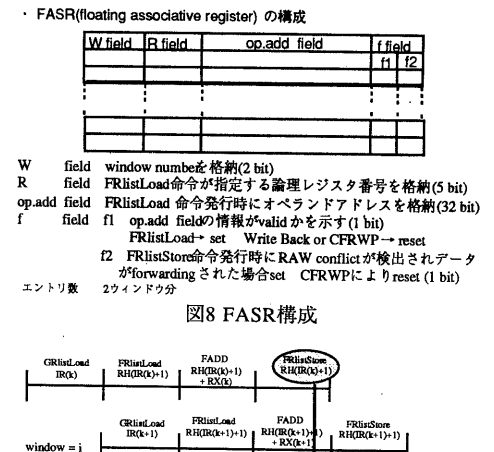


図8 FASR構成

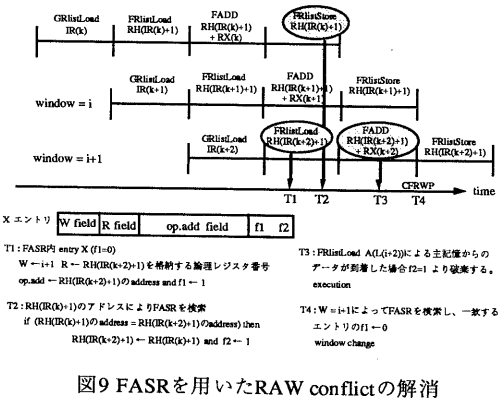


図9 FASRを用いたRAW conflictの解消

T2:FRlistStore命令実行時には、デスティネーションアドレス(address2)とFASR内の有効エントリのop.add.fieldとの一致検索を行なう。もし、(address1)=(address2)であれば、X-entryのop.add fieldとの一致がとれるので、X-entryの指しているレジスタ(window i+1、FR20)に、ストアすべきデータをforwardingし、X-entryのf2 を1をセットする。

T3:FADD命令が実行される。T2とT3の間にFRlistLoad命令により主記憶からデータが到着してもX-entryのf2bitが1のため、メモリからのデータは破棄される。

T4:CFRWPinc命令によってwindowがi+1からi+2に切り替わる時、W = i+1によってFASRを検索し、一致するもの全てに対しf1 ←0とし無効にする。

このような処理を行なうことでRAW conflictを検出し、解消することができる。

・FASRのエントリが有効な期間

FASRの役割は、FRlistLoad命令によりプリロードされるデータに対するRAW conflictを、execution phaseにおいてそのデータがオペランドとして消費される前に解消することである。よって例えば図10に示すように、j-th loopでのFRlistload phaseで発行されるFRListLoad RH(IR(k+1)+1)の情報を格納しているFASR中のエントリが有効である期間は、次の(j+1)-th loopのexecution phase(FADD RH(IR(k+1)+1)+RX(k+1))でオペランドとして消費されるまでの期間だけ有効であればよい。

すなわち1つのエントリは2つのループ中(上の例ではj-th loopと(j+1)-th loopの間)で有効であればよいことになる。

・FASRの総エントリ数

ある1ループ、例えば(j+1)-th loopにおいてはFRlistload phaseによってFASRのエントリに登録が行なわれる。また、execution phaseではj-th loopのFRlistload phaseで登録されたエントリの情報を用いて演算を行なう。すなわち図10中の矢印の重複が示すように、(j+1)-th loopでは2つのFRlistload phase分の情報をFASRは保持しなければならない。FRlistload phaseにおいてFRlistLoad命令が1ウィンドウを構成する全レジスタに対し発行されるとすると、FASRは1ウィンドウ分のエントリ数が必要である。よってFASRの総エントリ数は2ウィンドウのレジスタ数分必要となる。

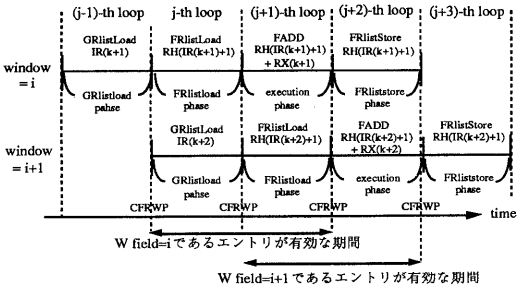


図10 FASRのエントリが有効な期間

4.2.2. CFRWPinc命令への機能追加

CFRWPinc命令は、3.3で述べた仕様の通りウィンドウ切替をおこなうだけの命令であったが、FASRの追加により以下のような機能を命令仕様に追加する必要がある。

```
CFRWPinc:
    ∀Entry(W,R,op.add,f1,f2)
        if (W = current window)
        then      f1 flag ← 0
    CFRWP ← CFRWP + 1;
```

4.2.3. コンパイラによるオブジェクトコードの制限

図9中、時刻T3の演算命令で使用されるオペランドのデータの正しさは絶対に保障されなければならない。そのためオブジェクトコード生成時にある制限を持たせる必要がある。それはスーパースカラ方式により演算命令と並列に発行される可能性があるFRlistStore命令が、T3より以前に全て発行を終了していなければならない、ということである。なぜなら演算命令より後にFRlistStore命令が発行される場合には、FASRによりRAW conflictが検出されたとしても、演算命令は既に正しくないデータを用いて実行を開始しているからである。

このためコンパイラは以下のような制限によりオブジェクトコードを生成しなければならない。

・1ループ内で処理される全てのFRlistStore命令は、演算命令が発行される以前に、その発行を終了しているようにオブジェクトコードはスケジューリングされなければならない。

4.3. 処理コード例

図11に図6-(b)で示したDO 14の1ループのコード例を示す。この例ではload/store pipelineが2本あるものとしている。最も右側が計算部分の命令、他の2列は並列に処理されるロード/ストア命令を示している。このコード例では2回のループアンローリングを施した例であり、このように演算命令が実行される前に全てのFRlistStore命令が発行済みでなければならない。

	load/store		load/store		floating ADD / SUB
1	FRlistStore	RH(IR(k))	FRlistStore	RH(IR(k)+1)	
2	FRlistStore	RHA(IR(k+1))	FRlistStore	RHA(IR(k+1)+1)	
3	FRlistStore	RHB(IR(k+2))	FRlistStore	RHB(IR(k+2)+1)	
4	GRlistLoad	IR(k+9)	GRlistLoad	IR(k+10)	RH(IR(k+3))+fw
5	GRlistLoad	IR(k+11)	FRlistLoad	RH(IR(k+6))	RHA(IR(k+4))+fw
6	FRlistLoad	RH(IR(k+6)+1)	FRlistLoad	RHA(IR(k+7))	RHB(IR(k+5))+fw
7	FRlistLoad	RHA(IR(k+7)+1)	FRlistLoad	RHB(IR(k+8))	RH(IR(k+3)+1)+RX(k+3)
8	FRlistLoad	RHB(IR(k+8)+1)	FRPreLoad	RX(k+6)	RHA(IR(k+4)+1)+RX(k+4)
9	FRPreLoad	RX(k+7)	FRPreLoad	RX(k+8)	RHB(IR(k+5)+1)+RX(k+5)
10					RH(IR(k+3))+fw-RX(k+3)
11	branch				RHA(IR(k+4))+fw-RX(k+4)
12	CFRWP				RHB(IR(k+5))+fw-RX(k+5)

図11 LFK14 コード例

5. 評価

5.1. 評価対象モデル

以下の4つのプロセッサモデルを仮定して評価を行なった。

<basic> PA-RISC1.1 Architecture

<prefetch> 主記憶のパイプライン化とプリフェッチ命令を追加し、データキャッシュへのプリフェッチをコンパイラがサポートしたもの[1]。

<pvp> 従来提案していた擬似ベクトルプロセッサ。主記憶のパイプライン化、浮動小数点レジスタのレジスタウィンドウ化の拡張のみを行なったもの。

<list pvp> pvpに対し3、4章で述べたリストベクトル処理を追加したもの。汎用レジスタのウィンドウ化、FASRを機能追加。

5.2. 評価環境

オブジェクトコードは各プロセッサモデルに対し、最適なコードをハンドコンパイルすることにより生成した。

評価においては以下のことを仮定した。

・命令発行:2命令発行のスーパースカラ方式。ストールが発生すると、後続命令はインタロックされる。同時発行される2命令は、ロード系(FRlistLoad等)、浮動小数点演算、整数演算(CFRWPincを含む)、その他(分岐命令等)のいずれか異なるグループに属する。またload/store pipelineを強化して、ロード系、又は整数演算グループに属する命令のみ、2命令が同時発行できるような3命令発行のスーパースカラ方式も仮定する。

データ依存関係:先行命令が浮動小数点演算なら5MC以上、先行命令がロード命令ならcache hit時に2MC以上、cache miss時ならmemory access latency以上、後続命令の発行を遅らせるとする。特に<prefetch>ではデータ依存関係によるストールが生じないように、コンパイラによってプリフェッチ命令が十分先行されて発行されるものとする。

・データサイズは8byteとする。load/store pipeline1本のメモリスループットは8byte/cycleとし、バンクコンフリクトは生じないものと仮定する。2命令発行のスーパースカラ方式の時は、1本のload/store pipeline、3命令発行のスーパースカラ方式の時は、2本のload/store pipelineを仮定する。

・データキャッシュ:ブロックサイズ 16byteとし、<basic>、<prefetch>ではline conflictは起こらないとする。また、<prefetch>においてはmulti-port cacheを仮定し、間接アドレスを持つ配列(例えばA(L(i)))以外に関してプリフェッチをおこなう。<pvp>、<list pvp>は通常のキャッシュメモリとする。

・命令キャッシュ:必要な命令は命令キャッシュ内に格納されているものとする。

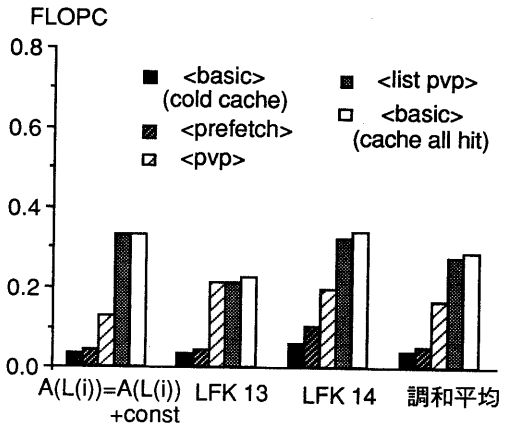
5.3評価結果

リストベクトル処理が効率よく扱えるかを評価するためにベンチマークとして、2章で例として用いたA(L(i))=A(L(i))+const、及びLFK 13、LFK14を取り挙げる。A(L(i))=A(L(i))+constに関しては、配列Aについて依存関係が生じないことが保障されているとして評価を行なう。

キャッシュメモリに入り切らないような大規模なデータ領域を想定するため、ループの繰り返し回数は増加して考える。

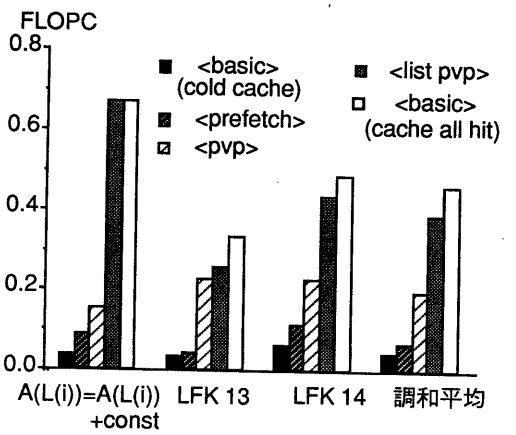
主記憶へのmemory access latencyを20CPUcycleとし、各モデルの性能とその調和平均を図12に示す。またload/store pipelineを強化してload/store pipelineを2本とし、3命令並列に実行可能なスーパースカラ方式を仮定した時の各ベンチマークの評価結果を図13に示す。

評価指標としてFLOPC(floating-point operations per clock cycle)を用いる[9]。



memory access latency = 20 cycle

図12 2命令発行スーパースカラ方式の時の各モデルの性能結果(load/store pipeline 1本)



memory access latency = 20 cycle

図13 3命令発行スーパースカラ方式の時の各モデルの性能結果(load/store pipeline 2本)

まずload/store pipelineが1本の場合の評価であるが、<basic>のモデルにおいてデータが予めcacheに入っていないcold cacheの状態は、cache all hitの性能と比較するとキャッシュミスによるペナルティーのため調和平均で14%と大きく性能が低下してしまう。またコンパイラによってソフトウェア的にキャッシュへのプリフェッチを行なう<prefetch>の性能は、cache all hitと比較して調和平均で18%程度となっている。これは、リストベクトル(LFK14ならIR(k))に関してはプリフェッチの効果が期待できるが、このリストベクトルを用いた配列(LFK14ならRH(IR(k)))についてはIR(k)の値がわからないとプリフェッチできないため、その効果が現れていないからといえる。<pvp>ではリストベクトルに対するプリロードができないこと(汎用レジスタへのプリロードができない)や、依存関係が実行時にならないと判明しないため、浮動小数点レジスタへのプリロード機能を追加しても効果が上がらない。そのため調和平均の58%程度の性能に留まっている。一方、<list pvp>では依存関係の有無によらず汎用/浮動小数点レジスタへのプリロード機能を十分活かすことができ、性能低下は起こっていない。またこの結果より、memory access latencyが20サイクル程度なら、主記憶へのアクセスレーテンシーをほとんど隠すことができていることがわかる。

次にload/store pipelineを2本と強化し、3命令発行のスーパースカラ方式とした場合、メモリスループットが2倍になったことにより、load/store pipelineが1本に場合と比較して<list pvp>において調和平均で40%程度性能が向上している。特にA(L(i))=A(L(i))+constでは2倍の性能が得られている。

各プロセッサモデルの相対的な性能の優劣はほぼload/store pipelineが1本の場合と同じ傾向を示している。

5.4実現可能性

追加するハードウェアとしては浮動小数点/汎用レジスタウィンドウ、論理レジスタ番号から物理レジスタ番号への変換回路、制御論理の拡張等であり、実装上大きな問題はないといえる。しかし連想レジスタであるFASRをchip上に実装する場合、その実装面積と検索時間が問題となる。

これらの問題に関しては詳細な設計を行なう必要があるが、将来的には実装技術の進歩によって実現可能であると考えられる。

また、主記憶のパイプライン化は最もコストがかかることである。しかしながら主記憶から高速にデータを転送するためには、メモリスループットを高くすることは本質的に必要不可欠である。

6.まとめ

擬似ベクトルプロセッサに機能追加として、汎用レジスタのレジスタウィンドウ化を行なうことでリストベクトル演算が効率よく扱えることを示した。また、実行時でないと依存関係が判明しないリストベクトル演算のために連想レジスタとソフトウェアにある制限を設けることで、依存関係の有無によらず擬似ベクトル処理を行なう手法を提案し、ベンチマークを用いた性能評価によりその有効性を確認した。

提案したアーキテクチャは、既存のアーキテクチャと上位互換性を保ちながらリストベクトル演算もその依存関係によらず、ベクトル処理を高速に処理できることを示した。

参考文献

[1] 位守弘充,伊藤元久,中村宏,中澤喜三郎,"擬似ベクトル処理向きメモリアーキテクチャの一提案",情報処理学会研究報告,ARC-91-8,1991

[2] 位守弘充,伊藤元久,中村宏,中澤喜三郎,"浮動小数点レジスタウィンドウを用いた擬似ベクトル処理",情報処理学会第44回全国大会,1992

[3] 中村宏,位守弘充,伊藤元久,中澤喜三郎,"レジスタウィンドウとスーパースカラ方式による擬似ベクトルプロセッサの提案',並列処理シンポジウムJSPP'92論文集,1992

[4] 位守弘充,中村宏,朴泰祐,中澤喜三郎,"擬似ベクトルプロセッサによるリストベクトル処理',情報処理学会第45回全国大会,1992

[5] K.Nakazawa, H.Nakamura, H.Imori and S.Kawabe, "Pseudo Vector Processor Based on Register-Windowed Superscalar Pipeline", to appear in Supercomputing'92

[6] M.L.Simmons and H.J.Wasserman, "Performance Evaluation of an Optimized Scalar with Two Vector Processors", Proc.of Supercomputing'90, pp132-141,1990

[7] N.P.Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a small Fully-Associative Cache and Prefetch Buffers", Proc.of 17th Int'l Symp. on Computer Architecture, pp364-373,1990

[8] A.C.Klaiber, H.M.Levy, "An Architecture for Software-Controlled Data Prefetching", Proc.18th Int'l Symp. on Computer Architecture, pp.43-53,1991

[9] 村上和彰,"マイクロベクトルアーキテクチャの検討",情報処理学会研究報告,ARC-92-6,1992

[10] 橋本隆,岡崎恵三,弘中哲夫,村上和彰,富田真治,"「順風」:MSF型ベクトル・プロセッサ・プロトタイプ",並列処理シンポジウム JSPP'92 論文集,1992