計算機アーキテクチャ 91－8
（1991. 11. 22）

擬似ベクトル処理向きメモリアーキテクチャの一提案

位守弘充　伊藤元久 中村宏　中澤喜三郎
筑波大学電子情報工学系

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、局所性がないためにキャッシュメモリは有効に働かない。そのため通常のRISC系を含めたスカラプロセッサでは、主記憶との間のデータ供給／格納が頻繁に発生し、実効性能が大きく低下することが多い。これを防ぐためには主記憶をパイプライン化して基本的な主記憶スループットを強化することが必要であり、すでに各種の方式が提案されている。本稿ではその上にプロセッサ側の新方式として多重キャッシュメモリ構成を導入する方法と、浮動小数点レジスタのウィンドウ化という２つの手法を提案する。この２手法のいずれによってもパイプライン化した主記憶をプリフェッチ命令によって有効に利用することができるようになる。

この手法を用いたスカラプロセッサによる擬似ベクトル処理について述べ、いくつかのベンチマークを用いて定量的な評価を行なった。その結果これらの手法は有効であり、キャッシュにデータが格納されている場合の性能と遜色のない性能が得られることを示す。

Pseudo Vector Processor Based on New Memory related Architecture

IMORI,Hiromitsu　ITO,Motohisa　NAKAMURA,Hiroshi　NAKAZAWA,Kisaburo
Institute of Information Sciences and Electronics, University of Tsukuba
1-1-1,Tennodai,Tsukuba,Ibaraki,305,Japan

In engineering/scientific applications, cache memory does not work effectively because non-local and large sizes of data are required. Therefore, in usual scalar processors including superscalar RISC , frequent data transfers between main memory and registers degrade the performance. To avoid this degradation, pipelined memory to maintain sufficient memory through-put , and prefetch feature are substantially required as usually mentioned. In addition to these methods, two independent approaches of processor are proposed. One is *multiple cache organization* and the other is *floating-point register windows* .

Based on the proposed architecture, high-speed pseudo vector processing is realized in just a scalar processor. We have applied some benchmarks to the proposed architecture and evaluated its performance. The performance evaluation shows that the proposed architecture drastically reduces the penalty of memory access and achieves almost the same performance as the case of all in cache.

1 はじめに

科学技術計算の分野には行列計算、連立1次方程式、微分方程式、積分、有限要素法などがある。この領域では配列データを扱うことが多く、特徴として以下に述べるようなものがある。

- データ領域はかなり大きく、局所性がない。
- 構造は比較的規則的であり、メモリアクセスパターンが決っていることが多い。
- ループを構成していることが多く、同じ処理を規則的に実行するのでスケジューリングすることでデータハザードを解消しやすい。

このような特徴を持つ問題を処理する場合、通常のスカラプロセッサのようにキャシュメモリを備えていても、データ空間が大きい、データの時間的局所性がない等の理由によりキャッシュは有効に働かない事が多い。そのためメインメモリとの間のデータ供給／格納が頻繁に発生し、メモリスループットがネックとなり実効性能が大きく低下することが多い。

このようなアプリケーションを高速に処理するためにはベクトルプロセッサがある。しかしベクトルプロセッサでは、多くのベクトル演算パイプライン、多くのベクトルレジスタ、多くのロード／ストアパイプラインを備えることで並列性を利用して高速化を図るため、ハードウェアの物量が増えコストが高いなどの問題点がある。従って通常の汎用スカラプロセッサによるベクトル処理はスカラ命令によるループの繰り返しにより処理されている。

スカラプロセッサにおいても最近ではスーパースカラ方式などにより複数の演算器などをパイプライン的に並行動作することで高速化を狙ったものが現れて来ている［2］。

そこで本稿では、通常のスーパースカラプロセッサに対してある種の機能追加により、スカラプロセッサでありながらベクトルプロセッサと等価な処理を行う擬似ベクトルプロセッサのアーキテクチャの方式として、多重キャッシュ構成、浮動小数点レジスタのレジスタウィンドウ構成という2つの独立な方式を提案する。ここではいくつかの浮動小数点演算のベンチマーク問題を取りあげキャシュミスによる実効性能の低下を評価する。次にこの低下を防ぐために本稿で提案する2つの手法を適用した時の性能を評価し、有効性を提示する。

2 評価環境

実際に評価をするにあたり理解を容易にするため具体例を提示し説明する。ここでは以下に示す評価のための主な要因についての仮定を行う。

2.1 マシン構成

2.1.1 アーキテクチャ

評価の際にアーキテクチャとして必要な要素を下記のように仮定する。

- RISCアーキテクチャの一例としてDLXアーキテクチャ［1］を用いる。これは汎用レジスタを32個（32bit）、浮動小数点レジスタを32個（64bit）持つロード／ストア・アーキテクチャである。

2.1.2 制御方式

以下のようなスーパースカラパイプライン方式を仮定する［2］。

- クロック周波数（f）は50MHz、100MHzの場合の2種類を想定する。
- 1サイクルで最大3命令同時実行可能。
  （分岐命令）＋（浮動小数点演算命令）
  ＋（固定小数点演算orロードorストア命令）
- 演算の実行にかかるサイクル数
  固定小数点演算 → 1サイクル
  浮動小数点演算 → 4サイクル
- ステージ構成
  5つのステージで構成されているものとする（フェッチ、デコード、実行、メモリアクセス、データ書き込み）。浮動小数点演算の場合は実行ステージが4サイクルになる。

2.2 ベンチマークプログラム

ベンチマークとして、Livermore Loopからkernel #3、kernel #5の2つとLinpackの中心的演算であるDAXPY（Double - precision A*X Plus Y）を取り上げる。これらにループアンローリングの技法を用いてハンドコーディングを行い、仮定したスーパースカラプロセッサ向きの最適化コードを生成する。また浮動小数点演算の各データサイズは全て64ビット（8バイト）とする。

各ベンチマークについての詳細を下記に記す。

（1）Livermore Loop kernel #3（内積）

```
DO  10  k = 1,n
      Q = Q + Z(k) * X(k)
10    CONTINUE
```

本稿では以降L.#3と略記する。

（2）Livermore Loop kernel #5（recurrence）

```
DO  10  k = 2,n
      X(k) = Z(k) * {Y(k) - X(k-1)}
10    CONTINUE
```

本稿では以降L.#5と略記する。

（3）DAXPY

```
DO  10  k = 1,n
      Y(k) = A*X(k) +Y(k)
10    CONTINUE
```

本稿では以降DAXと略記する。

2.3キャッシュメモリ

キャッシュメモリに関して以下のように仮定する。

- データ領域が広くキャッシュには納まりきれないような問題を想定するので、キャッシュの容量は具体的に仮定を置かない。
- キャッシュに格納されたデータは1サイクルで8バイトのデータがread又はwriteができ、演算系へのデータ供給ができるものとする。
- キャッシュミスが起こったときのキャッシュへのブロック転送動作（リプレースメント動作）は、主記憶へのアクセス、ECC処理、データ転送、キャッシュへのデータの書き込みという一連の動作から成り立つ。これらの動作のミスペナルティをまとめて変数latencyとして与えることにする。主記憶にはDRAMなどの使用を考慮し、上記のような各動作による遅れを含め主記憶にプロセッサからアクセス要求を出しデータが得られるまでを合計200nsecかかるとする。各周波数の場合、latencyは以下のようになる。

  周波数(f)＝50MHz : latency＝10サイクル
  (f)＝100MHz : latency＝20サイクル
- ブロックサイズを16バイトと仮定する。各データサイズは8バイトであるので、一回のブロック転送で2つの連続したデータがキャッシュに格納される。
- 1ループ当りのキャッシュミス回数を示す(miss/loop)は、対象とするプログラムの最適化コードと上記のキャッシュのブロックサイズにより決定できる。今回は1で述べた配列データの特徴より、各データは連続して格納されているとする。1ブロック当り2つのデータが入っているのでミス回数はループ当り必要とされるデータ数の半分となる。3以降の評価において、予めデータがキャッシュに格納されていない場合を示すcold cacheの場合でも、キャッシュのブロック転送の効果で全てのオペランドアクセスがキャッシュミスとなるのでなく、オペランドアクセスの半分はキャッシュヒットすることになると仮定した。もしデータが非連続に格納されているとすれば、ミス回数はより大きくなる。

3 現状の性能評価と問題点

2で想定した仮定に基づき性能評価を行う。

3.1キャッシュミスによる性能低下

データが全てキャッシュに格納されている場合、すなわちcache all hitの時の性能評価式をMFLOPS（Million FLoatingpoint Operation Per Second）で表すと、

$$\frac{(\mathrm{FLOP/loop}) \times n}{\mathrm{const} + \mathrm{T} \times n} \times \mathrm{f}$$

const :loop外実行サイクル数　f :周波数［MHz］
FLOP :浮動小数点演算数　n :loop繰り返し回数
T : loop内実行サイクル数

となる。またnが大きい時constは無視できるので、近似的に性能評価式は

$$\frac{\mathrm{FLOP/loop}}{\mathrm{T}} \times \mathrm{f} \cdots (1)$$

とでき、以降この近似式を用いる。

次にデータが予めキャッシュに格納されていない場合すなわちcold cacheの時は、主記憶からのデータ供給のオーバーヘッドをループ内実行サイクル数と分割し概念的に示すと図1のようになる。

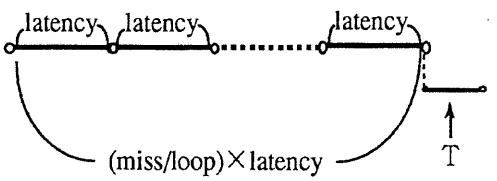

図1 cold cacheの時のメモリオーバーヘッド

この時性能評価式は,

$$\frac{\mathrm{FLOP/loop}}{\mathrm{T}+(\mathrm{miss/loop})\times\mathrm{latency}} \times \mathrm{f} \cdots (2)$$

と近似できる。以上の評価式（1）、（2）による結果を表1に示す。

表1によると、データ領域が大きくキャッシュにデータが入り切らない、また再利用がないような最悪の場合には11～37％に性能が低下している。これから完全にメモリスループットがネックとなりプロセッサのデータ需要にデータ供給がついていってないことがわかる。

表1　評価式1,2から算出した性能結果

	f	all hit	[0]
L.#3	50	40.0(1)	8.0(0.20)
	100	80.0(1)	8.9(0.11)
L.#5	50	46.2(1)	16.9(0.37)
	100	92.3(1)	20.7(0.22)
DAX	50	31.9(1)	7.6(0.24)
	100	63.8(1)	8.6(0.13)

単位：MFLOPS
all hit :データが全てキャッシュに存在
[0]:データ供給がパイプライン化されてない
()はall hit を１とした時の性能比

3.2 キャッシュ・主記憶間のパイプライン化

3.1の結果より、いかに高速にデータを主記憶から供給できるかということが、高い実効性能を保持するには重要であることがわかった。

主記憶のデータ供給能力にはスループットとレーテンシーという２つの側面がある。そこでまずスループットについて検討する。3.1の場合では、主記憶からのデータ供給は逐次的に行われていた。それを図２のようにパイプライン化することでデータの供給能力（スループット）の向上が期待できる。

主記憶をこのように擬似的にパイプライン化することは、主記憶を独立に動作するバンクに分割しアドレス付けをインターリービングすることで通常実現されている。

このときのデータ供給ピッチをPitchと定義する。Pitchは主記憶へのアクセス要求を出すことができる、すなわち主記憶からプロセッサへデータを供給する間隔（メモリバンクの衝突はないものと仮定している）であり、メモリのバンク構成、バス、プロセッサの構成方法等により定まる。ここではデータのバス幅は８バイト、pitchは１サイクルと仮定する。それはキャッシュミス時のデータのブロック転送において１６バイトのうち８バイトはキャッシュへの書き込みとレジスタへのロードを同時に行ない（これはTに含まれる）、残りの８バイトのキャッシュへの書き込みがpitchであると想定したからである。これにより2.3で仮定したパラメータlatencyをlatency = Moh + Pitch（Moh：メモリオーバーヘッド）のように分解する。

このようなパイプライン化された主記憶でかつバンクの衝突などを考慮しない場合、性能評価式は以下のようになる。

$$\frac{\text{FLOP/loop}}{\text{T+ (miss/loop)} \times \text{Pitch} + \text{Moh}} \times \text{f} \cdots (3)$$

これを用いた性能結果は表２となる。

表2　　評価式３から算出した性能結果

	f	[0]	[1]
L.#3	50	8.0(1)	20.0(2.5)
	100	8.9(1)	30.0(3.0)
L.#5	50	16.9(1)	34.3(2.0)
	100	20.7(1)	60.0(3.0)
DAX	50	7.6(1)	21.1(2.8)
	100	8.6(1)	37.0(4.3)

単位：MFLOPS
[0]：データ供給がパイプライン化されてない
[1]：データ供給をパイプライン化した時
()は[0]を１とした時の性能比

表２よりデータ供給がパイプライン化されてない場合（cold cacheの場合）と比較して2.0～4.3倍性能が向上した。この結果から主記憶をパイプライン化し、スループットを向上させることは有効な手法であるといえる。

ここに述べたメモリパイプライン化の対策は、命令セットアーキテクチャとは無関係な構成方式上の手法であり、既存のプロセッサにも導入されている［３、４］。

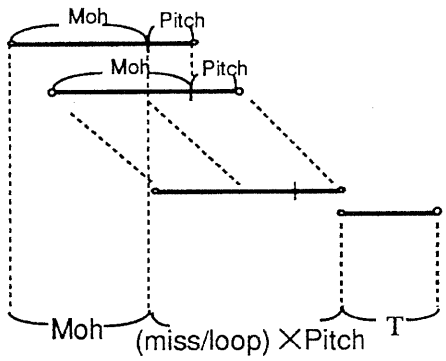

図２主記憶からのデータ供給のパイプライン化

４アーキテクチャの拡張を伴う対策

主記憶のパイプライン化によりスループットの問題は一応解決されるが、もう一つの側面であるレーテンシー（Moh）の問題は何も解決されてない。これこそ本稿が取り上げる主要テーマであり、次にこの問題を検討する。

4.1 プリフェッチ命令の導入

1でも述べたように配列型データでは、メモリのアクセスパターンは規則的であることが多く、データのプリフェッチが一般には可能である。そこでプリフェッチ命令を導入し、必要とされるデータをあらかじめ先発して主記憶へのデータ要求を出すことでレーテンシー（Moh）を少しでも隠すことが可能な筈である。

図3のようにi番目のループ実行時にi+1のデータをプリフェッチするプリフェッチ命令を発行する。Tpはプリフェッチ命令の追加による実行サイクルの増加分を表している。スーパースカラ方式においてプリフェッチ命令は他の演算命令と同時に並行して命令発行ができれば、Tpは必ずしもプリフェッチ命令数だけ必要ではないが、ここではプリフェッチ命令は1サイクル／命令を要するものとして、Tpとしてはプリフェッチ命令の数と等しいサイクルを仮定している。こうすることによって演算実行(T+Tp)と主記憶からのデータ供給（Moh）とが並行に動作する部分があり、その分だけレーテンシーを隠すことが可能である。

この隠せる割合をパラメーターkで表すことができる。このkは実行する各ベンチマークの1ループ当りの演算時間（T+Tp）とMohの大小関係で決定でき、従ってコードスケジューリングによる最適化といくつ先のデータをプリフェッチするか、すなわち多重プリフェッチを行なうか等の要素によってkを0にすることも可能である。

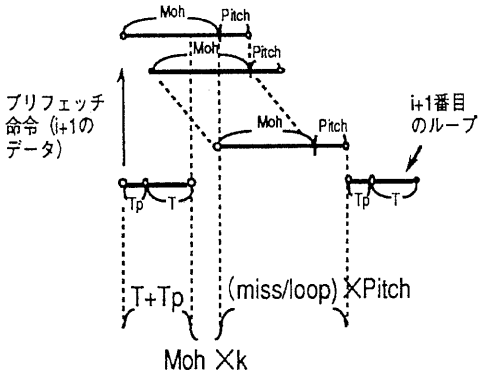

図3 プリフェッチ命令を導入した場合のメモリからのデータ供給のオーバーヘッド

この時の性能評価式は以下のようになる。

$$\frac{\text{FLOP/loop}}{\text{T+Tp+(miss/loop)} \times \text{Pitch+Moh}\times k} \times f \cdots (4)$$

Tp：プリフェッチ命令実行サイクル数
k：0≦k≦1

性能結果は表3のようになる。

表3よりcold cacheの場合の性能（0の場合）と比較して、1.8～5.0倍性能が向上した。

プリフェッチ命令を導入すれば実行サイクル数は増加するが、多重プリフェッチを行うことでパラメータkを0にすることができ、レーテンシーを隠すことができることから有効な手法であるといえる。しかし命令セットアーキテクチャの面からいうと、一般にプリフェッチ命令を持つものは少ない。この命令の導入はアーキテクチャの拡張を要することになる。

表3 評価式4から算出した性能結果

	f	0	2 (0≦k≦1)
L.#3	50	8.0(1)	16.2(2.0) ~ 22.2(2.8)
	100	8.9(1)	26.1(2.9) ~ 44.4(5.0)
L.#5	50	16.9(1)	30.4(1.8) ~ 34.3(2.0)
	100	20.7(1)	53.9(2.6) ~ 68.6(3.3)
DAX	50	7.6(1)	17.4(2.3) ~ 19.2(2.6)
	100	8.6(1)	31.3(3.6) ~ 39.0(4.5)

単位：MFLOPS
0：データ供給がパイプライン化されてない
2：データ供給のパイプライン化とプリフェッチ命令
()は0を1とした時の性能比

4.2 多重キャッシュ

4.1の性能評価式（4）よりプリフェッチ命令を導入することでレーテンシー（Moh）を解消できることを述べた。しかしまだデータ供給ピッチのオーバーヘッドは存在する。これを解消することができればさらに性能の向上が期待できる。

この供給ピッチは3.2に記したように16バイトのブロック転送のうち、後ろの8バイトをキャッシュに書き込むことに起因するものである。これを解消するには、プリフェッチ命令によるキャッシュへの書き込みと、ループ内のロード又はストア命令が同時に可能であればよい。すなわち1サイクルでメモリへのアクセスが2回以上できれば、データ供給ピッチのオーバーヘッドを解消することが可能である。

現状ではキャッシュを1次、2次キャッシュとして多階層の構成としたり、命令キャッシュとデータキャッシュを分ける分割キャッシュの方式を採用することはよくある。

本稿ではこれらと異なり同階層のデータキャッシュを複数の独立なキャッシュとすることで1サイクルで2回以上のメモリアクセスを可能とする。

4.2.1 多重キャッシュによる擬似ベクトル処理

例えばキャッシュA、キャッシュBという2つのキャッシュを独立に持つとする（これを2重キャッシュと呼ぶことにする）。図4のようにi番目のループのプリフェッチ命令（i+1番目のデータをプリフェッチする）はキャッシュAにデータの格納を行い、ループ内のロード、ストア命令はキャッシュBに格納されているi番目のデータを利用して実行を行う。すなわち2つのキャッシュをプリフェチ用キャッシュと演算用キャッシュに分けループごとに切り換えることで、両方のキャッシュを独立に同時に動かすことにより等価的に1サイクルに2回のキャッシュメモリアクセスが可能となる。このように2重キャッシュによって主記憶からキャッシュへのデータ供給と演算の実行を並行に処理することにより、ベクトルプロセッサにおけるベクトル処理と擬似的に等価な処理が可能となることが期待できる。勿論2組のデータキャッシュの一貫性の問題に対する対策は十分されていると仮定している。

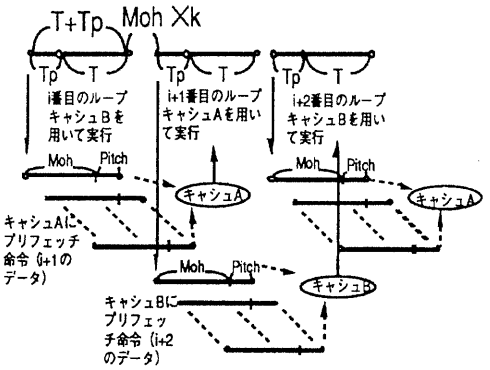

図4　多重キャッシュによる擬似ベクトル処理

4.2.2 評価式

上記のような2重キャッシュ構成とした場合、性能評価式は以下のようになり、その結果を表4に示す。

$$\frac{\text{FLOP/loop}}{\text{Max}(T+Tp\ ,\ (\text{miss/loop})\times Pitch)+Moh\times k}\times f \quad \cdots\cdots(5)$$

k: $0 \leq k \leq 1$ （4）式におけるkと同様の係数

表4よりcold cache（[0]）の場合と比較して2.0～6.4倍の性能向上が見られた。

このようにキャッシュが有効に働かないような科学技術計算においてもパイプライン化された主記憶と2重キャッシュ構成で、主記憶からのデータ供給をほぼ演算実行と独立に処理し、キャッシュミスによるメモリアクセスのペナルティを吸収して擬似ベクトル処理を行なうことができる。この場合cache all hitの場合の70～90%の性能が得られ、メモリスループットによる実効性能の低下を解消することができる。

表4　評価式5から算出した性能結果

	f	[0]	[3] (0 ≦k≦1)	
L.#3	50	8.0(1)	19.4(2.4)~	28.6(3.6)
	100	8.9(1)	30.0(3.4)~	57.1(6.4)
L.#5	50	16.9(1)	34.3(2.0)~	39.3(2.3)
	100	20.7(1)	60.0(2.9)~	78.7(3.8)
DAX	50	7.6(1)	21.1(2.8)~	24.2(3.2)
	100	8.6(1)	37.0(4.3)~	48.4(5.6)

単位：MFLOPS
[0]:データ供給がパイプライン化されない
[3]: データ供給のパイプライン化
プリフェッチ命令、2重キャッシュ構成
()は[0]を1とした時の性能比

5浮動小数点レジスタのウィンドウ化

5.1 浮動小数点レジスタウィンドウによる擬似ベクトル処理

5.1.1 多重キャッシュ構成におけるキャッシュの本質的な役割

1でも述べたように本稿で対象としている科学技術計算では一度キャッシュに格納したデータの再利用性が少ない。4.2.1で提案した多重キャッシュ構成もキャッシュの再利用性を主眼に置いたものでなく、多重キャッシュは主記憶から供給されるデータの転送バッファの役割を果しているにすぎない。よって独立な多重キャッシュとするのでなくprefetch data buffer register(PDBR)のようなバッファを適当数設けることでも多重キャッシュと等価なことが期待できる筈である。

5.1.2 浮動小数点レジスタのウィンドウ化

PDBRを用いても、演算時には浮動小数点レジスタへのデータの供給（要するに通常のロード命令）が必要であるが、ベクトルプロセッサにおけるベクトルレジスタのようにPDBRのデータを直接演算に利用しても何も問題は生じ

ない筈である。

そこで浮動小数点レジスタをウィンドウ化（FRウィンドウと呼ぶ）して図５のような構成をとることにする。そして図６のようにデータのプリフェッチをキャッシュを介さずに現在用いてないウィンドウ（裏ウィンドウ）のレジスタに直接行なうようなプリロード命令を追加し、i番目のループはウィンドウ１のデータを利用して実行を行ない、ウィンドウ２にプリロード命令によるデータの供給を行なう。次のi+1番目のループはウィンドウを切り換えてウィンドウ２のデータを利用して実行を行ない、ウィンドウ３にプリロード命令によるデータの供給を行なう。このようにループごとにウィンドウを切り換えて使用していないウィンドウのプリフェッチ部のレジスタにデータのプリフェッチを行なう。このプリロード命令の動作は通常のロード命令と同様な扱いとすることができる。そして主記憶のパイプライン化により十分なスループットでレジスタへのデータ供給ができれば、これを用いてスカラ命令でも連続的に演算をする可能性が出てくる。すなわち多重キャッシュと同様に擬似ベクトル処理が可能となる。

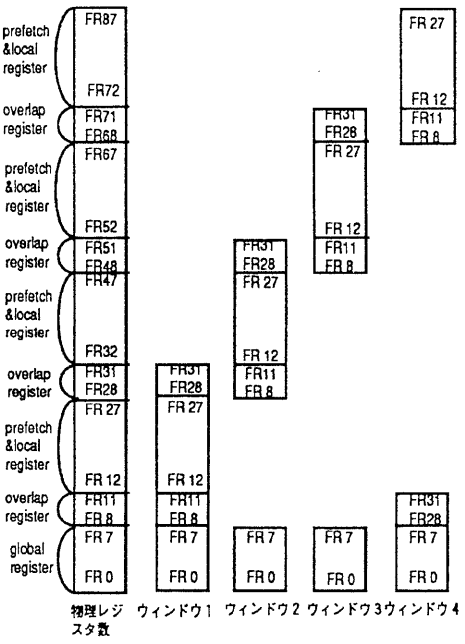

図５　浮動小数点レジスタウィンドウ構成

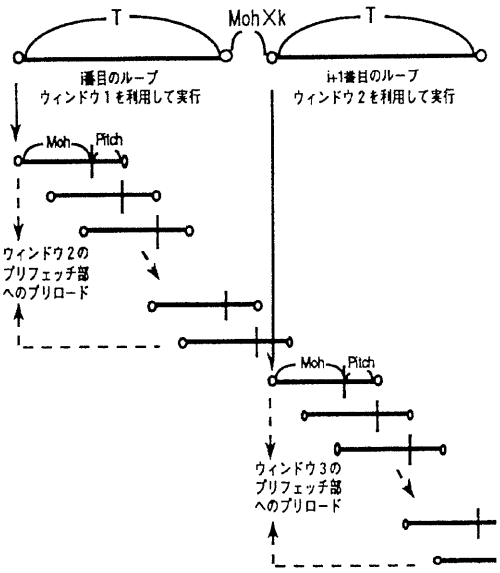

図６　浮動小数点レジスタウィンドウによる擬似ベクトル処理

5.1.3 多重キャッシュとの違い

多重キャッシュ構成とFRウィンドウとの違いとしては以下のことが挙げられる。

- ２重キャッシュ構成に比べコストが安い
- 実装上追加する論理が少ない
- 数種の命令（指定したレジスタへのプリロード命令、ポストストア命令、ウィンドウ切り換え命令等）の追加でよい
- 命令フォーマット中のレジスタフィールド長を変える必要がない

以上よりFRウィンドウはわずか数種の命令を追加することで実装が可能である。この点で多重キャッシュに比べ現実的な手法であるといえる。

5.2 評価

5.2.1 評価環境

今回はアーキテクチャとしてHewlett-Packard社のPA-RISC1.1アーキテクチャ［５］を用いて評価を行なった。ベンチマークは先と同様にL.#3、L.#5、DAX、及びLivermore loop kernel#7（L.#7と略記）を取り上げ、クロック周波数５０MHz（ピーク性能１００MFLOPS、メモリオーバーヘッド１０サイクル）と、１００MHz（ピーク性能２００MFLOPS、メモリオーバーヘッド２０サイクル）の２種類を仮定した。

ウィンドウ構成は図５のような構成を仮定した。

5.2.2 評価式

5.1.2で述べたような浮動小数点レジスタのウィンドウ化を行なった時の評価式は以下のようになる。

$$\frac{\text{FLOP/loop}}{\text{Max(T, (load/store inst)}\times\text{Pitch)+Moh}\times k}\times f \quad \cdots\cdots(6)$$

load/store inst : ロード／ストア命令発行数
k : $0 \leqq k \leqq 1$（4）式と同様な意味を持つ係数

4.2.2の評価式（5）との違いとしては、データのプリフェッチをキャッシュを使わずにレジスタに格納するので、キャッシュミスの回数であるmiss/loopがロード／ストア命令発行数となることと、ループ内のロード命令をプリロード命令に変更することで（5）式のTpというプリフェッチ命令の追加による実行サイクル数の増加がないことが挙げられる。表5に(6)式による結果を示す。（ただし表5における cold cache（0）、all hitのデータは5.2.1での仮定を基にしており、2.1での仮定を基にした表1とは異なるものである。）

表5よりk＝0の場合、cold cache（0）の場合の3.0～11倍の性能向上が見られ、cache all hit の場合の性能と全く遜色がない。

このように浮動小数点レジスタをウィンドウ化することで主記憶からのデータ供給と演算実行を高度に並行処理し、擬似ベクトル処理を行なうことは有効な手法であるといえる。

表5　評価式6から算出した性能結果

	f	0	4 (k=0)	all hit
L.#3	50	8.3(1)	50.0(6.0)	50.0
	100	9.1(1)	100.0(11)	100.0
L.#5	50	20.0(1)	60.0(3.0)	60.0
	100	24.0(1)	120.0(5.0)	120.0
DAX	50	7.7(1)	33.4(4.3)	33.4
	100	8.7(1)	66.7(7.7)	66.7
L.#7	50	23.5(1)	83.0(3.5)	89.0
	100	27.1(1)	166.0(6.1)	178.0

単位：MFLOPS
0：データ供給がパイプライン化されてない
4：データ供給のパイプライン化
　プリフェッチ命令、FRウィンドウ構成
all hit：データが全てキャッシュに存在する
()は0を1とした時の性能比
　0とall hitのデータはアーキテクチャと制御方式の違いにより表1の結果と異なる

5.3 浮動小数点レジスタのウィンドウ化の意義

このレジスタウィンドウの方式は等価的にはプリフェッチ用に多くのレジスタが利用できるアーキテクチャ、すなわち浮動小数点レジスタが十分ある（80以上）ようなプロセッサにおいて、ソフトウェアパイプライン的手法でレジスタを用いるのと同じ処理であるといえる。しかし一般に直接利用可能なレジスタの数は命令形式のレジスタフィールドの長さで制限され、あまり多くは用意できないという問題がある。浮動小数点レジスタのウィンドウ化はこのような制約の下でも実効的にレジスタ数を上位互換性に拡張できる所に意義がある。

6まとめ

キャッシュメモリが有効に働かないような科学技術計算分野のアプリケーションにおいて実効性能の低下を防ぐために、主記憶からのデータ供給のパイプライン化、プリフェッチ命令という手法を用いて多重キャッシュ構成、浮動小数点レジスタのウィンドウ化よりスカラプロセッサで擬似的にベクトル処理を行なう擬似ベクトルプロセッサアーキテクチャを提案した。

上記の手法を組み合わせることで、予めデータがキャッシュメモリに入ってないcold cacheの場合でもcache all hitの場合と遜色のない性能が得られることがわかった。

謝辞

本稿の内容に関して有益な御意見を頂戴した中田育男教授（筑波大学）、後藤英一教授（神奈川大学）、小柳義夫教授（東京大学）及び日立製作所の方々に感謝の意を表します。

参考文献

[1] J.L.Henessy & D.A.Patterson , "COMPUTER ARCHITECTURE A QUANTITATIVE APPROACH ", MORGAN KAUFMANN PUBLISHERS 1990

[2] R.R.Oehller,R.D.Groves, "IBM RISC System/6000 processor architecture", IBM Jounal of Reserch and Development , 1990

[3] M.Nakajima et al.,"OHMEGA :A VLSI Superscalar Processor Architecture for Numerical Applications", Proc. 18th Int'l Symp. Computer Architecture , pp160-168,1991

[4] Intel Corporation,"i860 64-Bit Microprocessor Programmer's Reference Manual", ISBN 1-55523-086-6, 1989

[5] Hewllet-Packard Company, "PA-RISC 1.1 Architecture and Instruction Set Reference Manual", Manual Part Number 09740-90039, 1990