

スケーラブル・レイテンシ・トレラント・アーキテクチャ

清水 尚彦

東海大学 工学部 通信工学科

〒259-12 神奈川県平塚市北金目 1117

TEL: +81-463-58-1211(ex.4084)

FAX: +81-463-58-8320

email: nshimizu@et.u-tokai.ac.jp

あらまし プロセッサの単体性能の向上に対してメモリレイテンシの改良は追いついておらず従来より性能上の隘路となっていた。近年RISCプロセッサのクロック周波数が1GHzを狙うまでに向上しておりこの傾向はますます強くなっている。従来リネームレジスタや疑似ベクトル処理によってこのギャップを埋める努力がなされて来たが、プロセッサの単体性能が更に上がるとこれらの技術ではメモリレイテンシを隠蔽することは困難であり別のアーキテクチャが必要とされる。本報告ではプロセッサの単体性能を支えるために必要な資源を簡単に整理し、アーキテクチャ上連続性を保ちながら幅広い性能レンジのプロセッサに簡単に適応できる手法を紹介する。

キーワードメモリレイテンシ, ハイパフォーマンス計算, RISC

Scalable Latency Tolerant Architecture

Naohiko Shimizu

School of Engineering, Tokai Univ.,

1117 Kitakaname Hiratsuka-shi Kanagawa 259-12, Japan

TEL: +81-463-58-1211(ex.4084)

FAX: +81-463-58-8320

email: nshimizu@et.u-tokai.ac.jp

Abstract The RISC technology is pushing the clocking rate up to 1GHz order. But comparing to the CPU clocking rate the improvement of memory latency is not so easy. Then for many numerical computations RISC could not offer sufficient performance, due to the heavy stalls for memory latency. One approach to this problem is the register renaming combined with the out of order execution. Another approach is the pseudo vector architecture, which offers software visual registers and asynchronous transfer instructions. They can provide more registers to wait for the memory latency. But more and more latency will be, it requires more and more registers to wait. And it will be too complex for high speed RISC core to provide enough number of registers to cover the memory latency. In this paper I present simple way for this problem.

key words memory latency, high performance computing, RISC

1 はじめに

プロセッサの単体性能の向上に対してメモリレイテンシの改良は追い付いておらず従来より性能上の隘路となっていた。近年RISCプロセッサのクロック周波数が1GHzを狙うまでに向上しておりこの傾向はますます強くなっている。従来リネームレジスタ [1] や疑似ベクトル処理 [2] によってこのギャップを埋める努力がなされて来た。しかしながら、たとえばレイテンシが一定と仮定するとプロセッサの単体性能が更に上がるとレイテンシを隠蔽するために必要とされるレジスタの数は単体性能に比例して増大し、プロセッサの実装を困難にする。そこで、これらのアーキテクチャを使用した場合には単体性能の向上が進んだ際に別のアーキテクチャを必要とし、アーキテクチャの連続性が失われる可能性がある。そこで、耐レイテンシ性を有するスケーラブルなアーキテクチャが要望されている。

耐レイテンシ性を有するスケーラブルなアーキテクチャであれば、デスクトップのワークステーションから計算機センターに設置する大型のマシンまで同一アーキテクチャで構成できる可能性を持つ。超並列コンピュータも同様のゴールを目指していたが、並列プログラムは誰にでも作れるものではなく結果として一般利用者にとって性能上のスケーラビリティはあまり確保されていないのではないだろうか？一方、数値計算系のプロセッサとしてはベクトル系のプロセッサが従来より使用されており優れた実績を残して来た。ところが、ベクトル系のプロセッサはクローズドなアーキテクチャでありプログラムの互換性からみた場合に難点があった。特にデスクトップのワークステーションの性能が向上している昨今、小さな問題はデスクトップで解いて大きな問題を計算機センターに依頼するような環境の住み分けが可能となっているが、アーキテクチャの異なるマシンで最適な性能を出すためにはコンパイラを変えるだけでなくプログラムのチューニングすら必要な場合が多かった。逆にベクトル系プロセッサを単一チップのLSIで構成することも試みられてきたが、デスクトップワークステーションクラスのマシンに普及しているとは言えない状況である。

本報告ではプロセッサの単体性能を支えるために必要な資源を簡単に整理し、アーキテクチャ上連続性を保ちながら幅広い性能レンジのプロセッサに簡単に適応できる手法を紹介する。

2 耐レイテンシ性を有するアーキテクチャのスケーラビリティ

並列コンピュータ等ではプロセッサ台数に対して性能の向上曲線がリニアに近いときスケーラビリティの高いシステム/プログラムであると呼ばれる。

本報告では単体プロセッサの性能にも同様にスケーラビリティを定義する。

定義

耐レイテンシ性を有するスケーラブルなアーキテクチャとは次の項目を満たすものとする。

1. 命令セットアーキテクチャの変更を伴わずにプロセッサ性能の向上の結果としての相対的なレイテンシの増大に対して必要とされるデータを遅れなく供給するハードウェア手段を提供できる。
2. 相対的なレイテンシの増大に対して必要とされるデータを遅れなく供給するハードウェア手段がプロセッサクロック周波数の向上のクリティカルパスとならずに構成できる。
3. 同一アーキテクチャによって様々な性能レンジに最適化されたマシンを構成でき、かつ各性能レンジのマシンが当該レンジの他のアーキテクチャに対して十分競争力を維持できる。

3 例題による要求仕様の検討

行列計算において良く使われる DAXPY というサブルーティンがある。

$$y(i) = y(i) + a * x(i) \quad (1)$$

このサブルーティンは各要素演算において2つのデータを必要とし、一つのデータを生成する。また、加算および乗算が各一回発生する。DAXPY の性能が支配的なベンチマークとしては LINPACK が有名であり、Dongarra がベンチマーク結果を公開している。[3] LINPACK の中でも改変の許されていない 100 元のプログラムはそのマイナーループに DAXPY が入っており、実行時間をほぼ支配している。Dongarra レポートの 2 月号によると LINPACK 100 元の性能上位に位置するマシンはほぼ例外なくベクトル機である。いわゆる RISC プロセッサの DAXPY 性能で最上位に位置するマシンは DEC 500/500 というマシンであるが、まだまだその差は大きい。また、Dongarra レポートでは 100 元という小さな行列を対象とするため RISC プロセッサにとってはデータがキャッシュに十分入りきり実際の問題を解決する場合の性能よりも高い性能が報告される傾向にあり、実性能の面からは更に差が開いている可能性がある。

DAXPY の性能に対して要求されるメモリ転送性能は表 1 に示すように、非常に大きい。

表 1: DAXPY 性能に対して要求されるロード/ストアのメモリ転送性能

DAXPY (MFLOPS)	Load (MB/S)	Store (MB/S)
100	1600	800
200	3200	1600
500	8000	4000
1000	16000	8000
2000	32000	16000

この転送性能を確保するのですら単一の LSI で構成する RISC プロセッサに取っては非常に困難を伴うが、転送性能はバス幅を広げるか、高速バスを使うことによって確保されると仮定する。次に問題になるのはメモリ要求を発してからデータが到着するまでの時間 (メモリレイテンシ) であり、レイテンシが長い場合にも演算を中断することなく実行する方式が求められる。このために一般には明示的もしくは暗黙的 (レジスタリネーミング) 手法によりデータをプールするバッファを用意する。[1][2] メモリレイテンシが一定であり、メモリ競合が起きないものとするとき要求されるデータ転送スループット LT、メモリレイテンシ ML と必要とされるバッファの容量 B には式 (2) のような関係がある。

$$B = LT \times ML \quad (2)$$

この容量を表 2 にまとめるが、たとえばレイテンシ 200nS のメモリスистемで 2GFLOPS の DAXPY 性能を確保する場合に要求されるバッファ容量は約 6KB になり、リネームレジスタや疑似ベクトル処理で行なっているようにこれをレジスタで実現する場合には 800 個ものレジスタが必要とされる。命令パイプラインのクリティカルパスに大量のレジスタを配置するのは高性能なプロセッサにとって負担が大きく明らかに別の方法が必要とされる。

一方、キャッシュプリフェッチを使うことで容量的には問題なくなるが、キャッシュラインの管理はペンディングとなるキャッシュロード要求が多くなると困難になってくる。先程の例では、たとえばキャッ

表 2: DAXPY 性能/メモリレイテンシに対して要求されるデータバッファの容量

DAXPY (MFLOPS)	Load (MB/S)	Latency		
		100nS	200nS	400nS
100	1600	160	320	640
200	3200	320	640	1280
500	8000	800	1600	3200
1000	16000	1600	3200	6400
2000	32000	3200	6400	12800

シュラインを 128B とするとペンディングラインが 50 ラインにも達する。各後続リクエストに対してペンディングラインへのリクエストかどうかを判定するためには連想メモリなどによってペンディングラインの管理が要求される。このように、定常的な計算実行のためのデータバッファの確保という単純な目的のためにはレジスタリネーミングやキャッシュプリフェッチといった汎用的な手法ではハードウェアの負担が多くなり好ましいものではない。

4 スケーラブル・レイテンシ・トレラント・アーキテクチャ:SCALT

前節までの検討を踏まえて、スケーラビリティの高い耐レイテンシ性を持つアーキテクチャを検討する。(ここでは、このアーキテクチャに SCALT というニックネームをつける。) SCALT の満たすべき条件を挙げると、次のようになる。

1. メモリ素子によるバッファの実現が容易なこと。
2. バッファの容量が増大した時にも連想メモリなどの管理手法を必要としないこと。
3. 命令パイプラインのクリティカルパスには負荷の重い論理を必要としないこと。
4. 性能を引き出すソフトウェアが容易に書けること。

これらの要件を満たす SCALT のデータパスの構成を図 1 に示す。この構成図ではデータバッファ領域として SCALT バッファというメモリをキャッシュと並行して配置する。この領域はキャッシュと同一のメモリによって実現し、メモリアドレスによってアクセスを切り替えるのが自然な実現方法であろう。主記憶装置は要求されたデータを送出する前に次のデータの受付を行なうストリーミング動作をする。そこで、プロセッサは主記憶装置からの戻りデータを区別するためのタグをメモリ要求に付加して発行する。タグの大きさはストリーミング動作によってペンディングとなるメモリリクエストを全て格納するだけの小さな整数を表せればよいが、たとえば、8 ビットのデータとすることができる。タグデータが前出のように 8 ビットとすると全部で 256 個のペンディングリクエストを処理できるが、たとえばこの内 8 個を通常のキャッシュアクセスに使用し残りの 248 個で SCALT バッファのエントリ番号を直接指定するようにする。主記憶からの戻りデータはタグの値によって直接 SCALT バッファの対応エントリ番号が与えられるので(もちろん通常のキャッシュへのリクエストはキャッシュ制御回路が格納場所を指定する) SCALT バッファのエントリ数が増大してもデータの格納に際するハードウェア構成の複雑化を招くことはない。SCALT バッファのエントリの大きさはインプリメント依存であるが、主記憶装置やプロセッサの構成を単純化するためにはキャッシュのラインサイズに合わせるのがいいだろう。

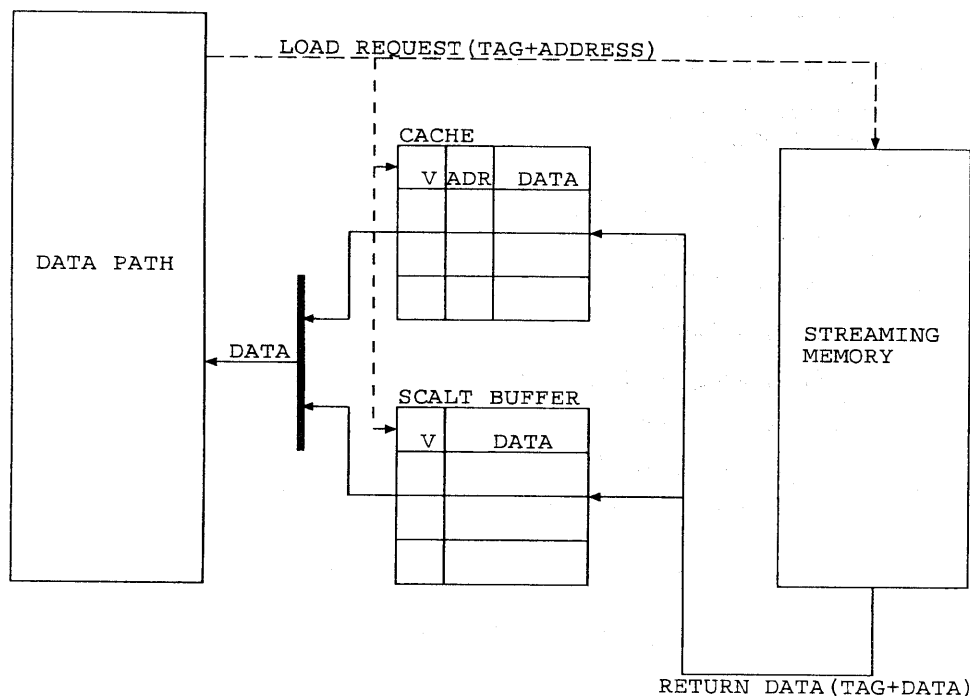


図 1: An example load path configuration with SCALT

次に SCALT バッファへの転送要求 (SCALT フェッチ) について検討する。SCALT フェッチは主記憶のアドレスと転送先の SCALT バッファのエントリ番号を指定して発行する。これは通常のストア命令とハードウェアの処理はほぼ同等である。SCALT フェッチが発行されると、SCALT バッファの対応エントリの有効ビットは解除される。SCALT バッファの有効ビットが既に解除されているエントリへの新たな SCALT フェッチの発行時は先行した SCALT フェッチを上書きするが、ソフトウェアデバッグ時には例外として処理しても構わない。(ただし、有効ビットを命令パイプラインの中でチェックするのはハードウェア的なオーバーヘッドとなるので、命令に非同期な例外処理とするかデバッグ専用のチェック命令を挿入するなどの対策をするべきである。)

SCALT バッファは特別な仮想アドレスに写像し通常のロード命令によってデータを取り出す。有効ビットが解除されているエントリへのロード命令は通常のキャッシュロードと同様にプロセッサをストールさせる。SCALT バッファのマッピングのために TLB のエントリが一つ必要となるが、全体性能に大きな影響を与えることはないと思われる。また、仮想アドレスへ写像することによって SCALT バッファを使うプログラムと使わないプログラムの間の資源保護が簡単になる。

以上のような SCALT の仕組みを使ってソフトウェアがプロセッサの性能を引き出すためには実装依存の幾つかのパラメータをオペレーティングシステム等から入手する必要がある。これには次のようなものがある。

- SCALT バッファエントリ数 (SNE)
- SCALT バッファエントリのサイズ (SSE)
- SCALT バッファの仮想アドレス (SBP)

```

#define SSEd sse/sizeof(double)
double *sbp;
int xb=0, yb=sne/2;
sbp=scaltbp();
for(i=0;i<sne/2;i++)
{
    scaltf(i, &x[i*SSEd]);
    scaltf(i+sne/2, &y[i*SSEd]);
}
for(i=0;i<n;i+=SSEd)
{
    for(j=0;j<SSEd;j++)
    {
        y[i+j]=sbp[yb*(SSEd)+j]+a*sbp[xb*(SSEd)+j];
    }
    scaltf(xb, &x[i+SSEd]);
    scaltf(yb, &y[i+SSEd]);
    xb=(xb+1) % (sne/2);
    yb=((yb+1) % (sne/2))+sne/2;
}

```

図 2: SCALT を使った DAXPY ループのプログラム例

5 プログラム例

図 2 に DAXPY ループを SCALT によって実現するプログラムの例を示す。ここに示すように比較的簡単にレイテンシを隠蔽するプログラムを作成することができる。もちろん、ループが非常に短い場合には SCALT を使うことによってプログラム上のオーバーヘッドが大きくなり性能低下の原因になる。そこで、数値計算ライブラリ等ではループ長によって異なる戦略のプログラムを用意する必要があるだろう。

6 まとめ

プロセッサの単体性能の向上にともない相対的に増大して来たメモリレイテンシを隠蔽するための新しいアーキテクチャ SCALT について報告した。今後、SCALT の実証のためにプロセッサの設計を進めていく予定である。また、本報告では触れなかったがストライドを持つ行列要素の扱いや実アプリケーションにおける SCALT の利用方法についても今後さらに検討を進めて行く予定である。

参考文献

- [1] Bhandarkar, "Alpha Implementations and Architecture", Digital Press, 1996
- [2] Nakazawa, Nakamura, Imori, Kawabe, "Pseudo Vector Processor based on Register-Windowed Superscalar Pipeline", Supercomputing'92, 1992.
- [3] <ftp://netlib.att.com/netlib/benchmark/performance.ps.Z>