ループを並列実行するクラスタ型アーキテクチャ

渡 辺 憲 一[†] 五 島 正 裕[†] 坂 井 修 一[†]

近年では，音楽や映像などのマルチメディア処理といった，データ並列性のあるプログラムがプロセッサ上で実行されることが多くなっている．これらの処理を高速に行うために，マイクロプロセッサに SIMD（Single Instruction/Multiple Data stream）命令セットを実装し，それを用いてプログラムを記述するのが主流になっている．しかし，アルゴリズムが高度化するに従いループも複雑化し，ベクトル命令の一種である SIMD 命令で対応するのは，限界が近づき始めている．そこで本論文ではデータ並列性のある処理を高速に実行するために，クラスタ型アーキテクチャを用いてループを並列に実行する方法を提案する．

Clustered Architecture for Loop-level Parallelism

KEN-ICHI WATANABE,[†] MASAHIRO GOSHIMA[†] and SHUICHI SAKAI [†]

Recent microprocessors have SIMD instruction sets in order to execute multimedia tasks effectively. SIMD instruction is one of vector operations, so it is useful for data-parallel loops. But loops in multimedia tasks become complex and irregular because algorithms become highly complicated. In this paper, we propose a clustered architecture which runs loops parallely to deal with data-parallel tasks.

1. はじめに

近年スカラ，スーパースカラなどの汎用プロセッサに追加される **SIMD(Single Instruction/Multiple Data Stream)** 命令が，新しいベクトル処理方式として注目されている．SIMD 命令は，当初，汎用プロセッサの持つ浮動小数点演算器/レジスタを流用して，より幅の狭いデータのパック(pack)——典型的には，色情報 (R, G, B, α) や同次座標 (x, y, z, w) ——を処理するために導入された．しかし現在では，複数の単精度，あるいは，倍精度の浮動小数点数をビット幅の長いレジスタに格納して扱えるようになり，汎用のベクトル処理機構として期待されている．近年の高性能プロセッサの多くは，ベクトル処理の高速化のため，SIMD 命令セットを実装している．SONY，東芝，IBM の開発した Cell プロセッサに至っては，SIMD 命令専用のプロセッサ・コアを複数集積している[1]．

SIMD 命令の利点は，その名のとおり，“Single Instruction/Multiple Data”というヒューリスティクスから生まれる．SIMD 命令は 1 命令で複数のデータを処理できるため，制御回路の占める割合を小さく抑えることができ，面積効率，電力効率に優れている (**図 1**)．このことはまた，命令スケジューリング・ロジックの簡単化にも著しい効果がある．

一方で，専用演算器やベクトル・プロセッサほどではないとはいえ，SIMD 命令セットの柔軟性は，通常のスーパスカラ・プロセッサのそれと比べると貧弱と言わざるを得ない．すなわち，SIMD 命令セットとその実行機構には，以下のような問題がある：

(1) SIMD 命令を使用するには，基本的には，パックにするデータが揃っている必要がある．そのため，コンパイラによって SIMD 命令を自動生成することは，典型的な場合を除き困難である．また，回転回数の少ないループ，不定なループを効率よく処理することができない．

(2) 演算器の幅を拡張するため，また，行列や複素数の乗算などの有用な演算パターンの実現のためには，それ専用の命令をいちいち追加する必要がある．実際，Intel Pentium プロセッサ[2] では，MMX，SSE，SSE2～SSE3 と，十年で 3 度の拡張を行っており，SSE4 の存在も噂されている．命令の追加は，命令のフィールドを圧迫する．また，命令を追加しても，既存のバイナリは高速化されない．

これらの問題は，端的に言えば，SIMD 演算器の多種

† 東京大学
University of Tokyo

多様な動作パターンの１つ１つを，個々のSIMD命令によってそれぞれ完全に指定するために生じる．これらの問題のため，SIMD命令で効率よく処理できるのは，いわゆるdo_allのようなごく単純なループに限られる．

ところが，SIMD命令の対象たるベクトル処理は，近年高度に複雑化する傾向にある．これは，コーデック(CODEC)などに顕著に見られるように，ユーザや市場の要求に応えるため，処理アルゴリズム自体が高度化しているからである．すべてのデータに一律に同じ処理を行うような単純なアルゴリズムでは，もはや所望の性能を得ることはできない．

高度なアルゴリズムでは，データ構造自体が不規則なものになったり，個々のデータの特性に合わせて適応的に処理を選択したり，実質上効果のない処理を省略したりするようになる．その結果，プログラムのループ構造は不規則なものとなる．具体的には，ループの回転回数が極端に少なくなったり，イタレーション中にif-then-else構造が含まれるようになったり，ループからの脱出が現れたりするようになる．

前述したように，現在のSIMD命令では，このような不規則ループを効率よく処理することができない．不規則なループの場合には，コンパイラによる対応が困難であることはもちろん，たとえ人手でアセンブリ言語を用いてプログラミングを行ったとしても高い効率は望めない．

こうした背景を踏まえ，本研究ではループを動的に複製して並列に実行することで，高速化を行うアーキテクチャを提案する．命令セットがスカラのまま並列実行することで以下のようなメリットが得られると考えられる：

- 命令セットはスカラのままであるので，バイナリ互換性を保つことができる．
- プログラムのセマンティクスもスカラのままであるので，SIMD命令単位ではなく，命令に含まれる個々の処理ごとに投機を行うことが可能になり，回転回数の不定なループへの対応が容易になる．
- 異なる種類の命令を同時に実行することが可能になる．通常のSIMD命令はループ・アンローリングにのみ対応するものであるが，異種操作を単一の命令に合成することによってソフトウェア・パイプライニングにも対応可能となり，いわゆるdo_across型のループも処理できるようになる．

以下本稿では２でクラスタ型アーキテクチャ上でループを並列に実行する方法を提案し，３で行った評価と，その結果を述べる．最後に４でまとめと今後の方針について述べる．

2. 提案手法

2.1 概　　要

ループを高速に実行するために，例えば単純にフェッチ幅を４倍にし，命令ウィンドウに入れて，依存関係が無いものだけを実行するといったことはハードウェアの制約から不可能である．

まず，１サイクルに命令キャッシュからフェッチ可能な命令数は通常最大で４命令程度である．これは以下のような原因によるものである：

- キャッシュラインの先頭からフェッチするとは限らない．
- 分岐命令が存在すると，フェッチする命令のアドレスが不連続になる．
- １サイクルで予測可能な分岐命令の数が限られている．

またより多くの命令を同時に実行するためには命令発行幅を広くする必要がある．しかし，レジスタ・リネーミングや命令のスケジューリング，演算結果のフォワーディングといったロジックの回路は，発行幅の２乗〜３乗に従って複雑化する．すると，配線遅延が大きくなるため，各ロジックのレイテンシが増大してしまい，動作周波数低減の要因になる．

提案手法では上に挙げた問題を解決するために，非対称であるクラスタ型アーキテクチャに，ループを高速に実行するためのデータパスを追加するという構成をとる．このハードウェア上で，ループ内の命令を複製して各クラスタに割り当てることでループを並列に実行する．以下では従来のクラスタ型アーキテクチャについて述べた後，提案手法について述べる．

2.2 クラスタ型アーキテクチャ

従来のクラスタ型アーキテクチャ[4]では，プロセッサを広い発行幅の命令ウィンドウや多数の演算器を持つ単一の実行部で構成するのではなく，小さい発行幅の命令ウィンドウとや小数の演算器をもつクラスタを複数集めて構成する．クラスタ型アーキテクチャのメリットは，実行部を分割したため，全体としては広い発行幅を持ちながら，回路の複雑さの増加を抑えられることにある．

一方で，命令のリネームを行った後で，どのクラスタでその命令が実行されるかを決定するステアリングというステージが必要となる．このステアリングにおいて依存元の命令と依存先の命令が別々のクラスタに割り当てられた場合は演算結果のフォワーディングが行えず，スループットが低下することになる．

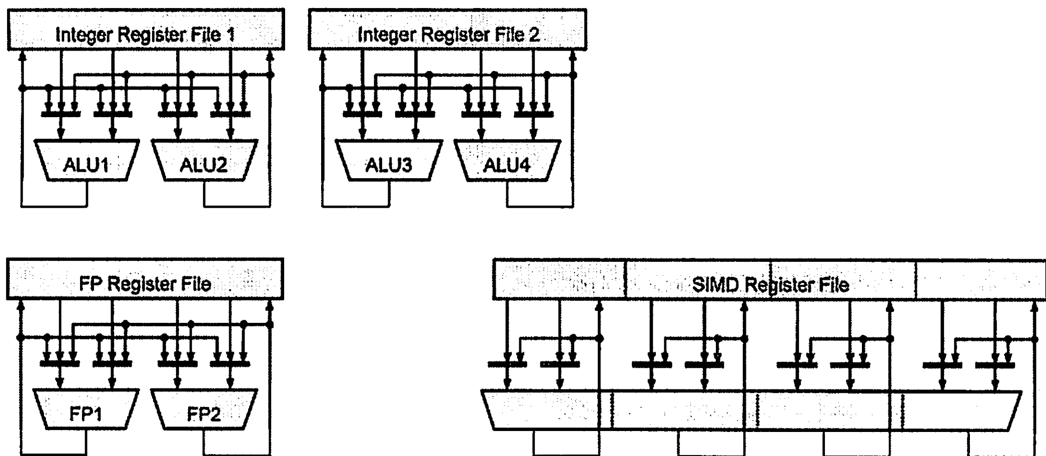

図1　SIMD 命令セットを実装したプロセッサ

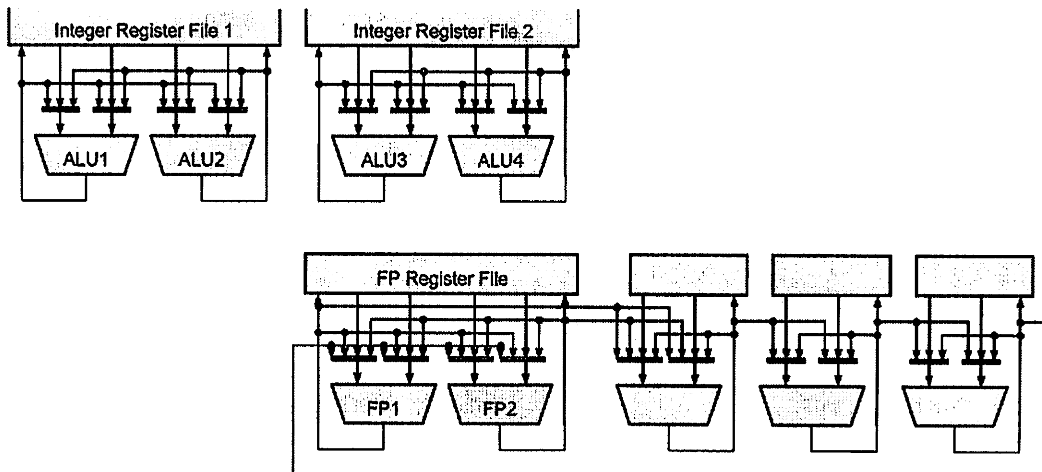

図2　提案手法

2.3 提案手法

提案手法では，クラスタ型アーキテクチャを用いるのに加えて，広い発行幅や多数の演算器を持つ**親クラスタ**1つと，発行幅も小さく演算器の数も少ない複数の**子クラスタ**という非対称な構成をとる．そして，一方向のフォワーディング用の回路を新たにクラスタとクラスタの間に追加する (図 2)．

このアーキテクチャ上で，ループ内の命令を複製し各クラスタに1イタレーションずつ割り当てることで，ループを並列に実行する．

非対称な構成

通常のクラスタ型アーキテクチャでは演算器などの資源を均等に各クラスタに分配するが，提案手法では親クラスタに発行幅や演算器を多く，子クラスタには少なく割り当てる．ループ以外の部分は親クラスタのみで，ループは並列に親クラスタと子クラスタで実行することで，マルチコア・プロセッサを用いて並列処理を行うよりも少ないハードウェア量で並列実行を行うことができる．子クラスタの演算器の構成は，データ並列性のあるプログラムが主に浮動小数点演算を行うことを考慮すると，浮動小数点の演算器の数は親クラスタと同数程度必要であるが，整数の加算器・乗算器などの演算器の数は少なくてよいと考えられる．

フォワーディング用の回路の追加

上で述べたように，クラスタ型アーキテクチャでは依存元の命令と依存先の命令が別々のクラスタに割り

当てられた場合は演算結果のフォワーディングを行うことができない．提案手法においては，ループの各イタレーションをそれぞれのクラスタに割り当てるため，ループの繰越依存があるとフォワーディングが行えず，スループットが低下することになる．例えばベクトルの各要素の和を求めるときは，前のイタレーションの命令の結果を使って演算を行う．これらの演算はクリティカル・パス上に乗っていることが多いため，可能な限り間隔を空けずに連続して実行する必要がある．

ここで，多くの場合においてあるイタレーションの命令の演算結果は，その次のイタレーションで使われることに着目する．図2で追加されたフォワーディング用の回路は，あるイタレーションを実行するクラスタの演算器から，その次のイタレーションを実行するクラスタの演算器に結果を送るために使われる．これにより，ループのイタレーションをまたいで依存関係がある場合にかかる通信遅延を低減することができる．

このフォワーディング用の回路は，隣のクラスタに対して一方向だけ追加されるので，クラスタ数nに比例して増加するだけである．そのため，回路の複雑さを増加させることなく追加することが可能である．

3. 評　　価

評価環境

本研究室で作成したスーパースカラ・プロセッサのシミュレータであるKototoiに提案手法を実装して評価を行った．Kototoiはcycle-accurateに単一のプログラムの実行をシミュレートすることができ，Alphaの命令セットに対応している．

構　　成

Alpha 21264の命令セットを持つプロセッサのシミュレーションを行った．評価の対象にしたモデルは通常のスーパースカラ・プロセッサと，親クラスタを1つと子クラスタを3つ持つ提案手法のプロセッサである．

親クラスタは通常のプロセッサと同じ演算器・命令ウィンドウの構成とした．一方で子クラスタは2.3でも述べたように，浮動小数点の演算器は親クラスタと同等に，整数系の演算器は親クラスタの半分程度で構成した．

ループの検出

今回はループの検出をアーキテクチャ上で動的に行った．ループは，2周回ることで動的に検出できる[3]．図3のようなループを例として考える．このループはadds命令から始まり，bne命令で終わる．

まず，ループの1周目の段階ではadds命令がループの先頭であることを知ることはできない．1周目の終わりにbne命令が実行され，後方への分岐が成立した場合，次の命令，addsをループの先頭，bneをループの最後の候補として記録する．

そして2周目にbne命令がフェッチされた時，adds命令からbne命令がループであることが判別できる．

ループを検出できた場合，命令を複製して，各クラスタにディスパッチすることで，クラスタの数だけのイタレーションに相当する命令をバックエンドに供給することができる．必然的にループ内のすべての分岐命令は常に不成立と予測し，ループ末尾の分岐のみ常に成立と予測することになる．

予備評価

単純なベクトル和を求めるループをプログラムを用いて実行速度の向上を確認した．評価に用いたプログラムは図4と図5の2つである．

評価の結果，IPCは表2のようになった．プログラム(a)は本来すべてのイタレーションを並列に実行できるため，提案手法でのIPCは通常のプロセッサの4倍になることが期待されたが，実際には2.3倍という

	通常のプロセッサ	提案手法	
		親クラスタ	子クラスタ
フェッチ幅	4 命令		4
命令ウィンドウ	128 エントリ	128	128
発行幅	4 命令	4	2
iALU	4	4	2
iMULT,iDIV	1,1	1,1	1,1
fpALU	2	2	2
fpMULT fpDIV fpELEM	2	2	2

表1 構成

```
       :
   lds $f11,0($1)
   lds $f10,0($2)
   mov $2,$4
$L6: #ループの先頭
   adds $f10,$f11,$f10
   addq $3,1,$1
   addl $1,$31,$2
   mov $2,$3
   s4addq $2,$4,$2
   cmple $3,9,$1
   sts $f10,0($2)
   bne $1,$L6 #末尾
       :
```

図3 ループの例

```
//ソースコード
double a[SIZE];
double b[SIZE];
double c[SIZE];

for(i = 0; i < SIZE; ++i) {
  c[i] = a[i] + b[i];
}

#アセンブリ
$L9:
ldt $f13,0($2)
bis $31,$31,$31
addl $3,1,$3  #ループ繰越依存のある命令
ldt $f14,8192($2)
cmple $3,$4,$5
addt $f13,$f14,$f1
stt $f1,16384($2)
cpys $f31,$f31,$f31
lda $2,8($2)
bne $5,$L9
```

図4 単純なベクトル和

```
//ソースコード
double a[SIZE];
double b[SIZE];
double c[SIZE];
double d[SIZE];
double e[SIZE];

for(int i = 0; i < SIZE; ++i) {
  c[i] = ALPHA * a[i] * b[i]
       + BETA * c[i] * d[i];
}

#アセンブリ
s8addq $2,0,$1
addq $1,$30,$1
ldt $f11,0($1)
mult $f14,$f11,$f11
ldt $f10,4096($1)
mult $f11,$f10,$f11
ldt $f10,8192($1)
mult $f13,$f10,$f10
ldt $f12,12288($1)
mult $f10,$f12,$f10
addt $f11,$f10,$f11
ldt $f10,16384($1)
addt $f11,$f10,$f11
stt $f11,8192($1)
addl $2,1,$2  #ループ繰越依存のある命令
cmple $2,$3,$1
bne $1,$L9
```

図5 命令数の多いループ

結果であった．これは提案手法では4イタレーション分，ループの誘導変数 i をインクリメントするのに4サイクルかかってしまうが，この処理にかかったサイクル数が，通常のプロセッサでループを1イタレーション分実行するのにかかるサイクル数よりも長かったため，完全に4倍の速度では実行できなかったためである．

一方で，ループ内の命令数を増やしたプログラム(b)では，通常のプロセッサで1イタレーションを実行するサイクル数の間に，提案手法で4イタレーション分のループの繰越依存を処理できたため，完全に並列処理を行うことができたため，IPC は4倍となっている．

	通常のプロセッサ	提案手法
(a)	3.01	6.97
(b)	2.59	10.3

表2 IPC

4. おわりに

本稿では，近年ますます重要になっているベクトル処理が高度化・複雑化している問題に対し，ループを並列実行するためのクラスタ型アーキテクチャを提案した．

予備評価の結果，スカラな命令セットで記述されたループを動的に並列実行することで，SIMD 命令セットが用いられていないプログラムの高速化が可能であることを示した．

しかし一方でループの繰越依存がある命令を実行するためにかかるサイクル数が大きくなると，並列実行を行ってもパフォーマンスが上昇しないという問題も発生した．

今後は以下のような方向で研究を進めようと考えている：

- 並列実行に適した命令セット
  現在の命令セットでは，本来並列に実行できるループにも，誘導変数のようなループの繰越依存が生じている．並列に実行可能ならば依存関係を生まないような命令セットが必要なのではないかと思われる．
- 親クラスタ・子クラスタへの資源の分配の検討
  本論文では，系統的に最適な子クラスタの数や演算器の割り当てなどを調べはしなかった．コア全体としてのハードウェア量の検討を行うためにも，最適な構成を検討する必要があると考える．

謝　辞

本論文の研究は，一部21世紀 COE「情報技術戦略コア」，および科学技術信仰機構 CREST「ディペンダブル情報基盤」による．

参考文献

1) Cell Broadband Engine, http://cell.scei.co.jp/j_download.html.
2) IA-32 Intel Architecture Software Developer's Manual, http://developer.intel.com/design/pentium4/documentation.htm.
3) 中島康彦, 津邑公暁, 五島正裕, 森眞一郎, 富田眞治: 「動的命令解析に基づく多重再利用および並列事前実行」情報処理学会 ACS 論文誌, Vol. 44, pp.1-16, No. SIG 10(ACS 2) (7 月)
4) 入江英嗣:「クラスタ型プロセッサ」情報処理学会誌, Vol. 46, pp.1111-1117, No. 10 (2005 年 10 月)