スラック予測を用いた クラスタ型スーパースカラ・プロセッサ向け命令ステアリング

福山智久[†] 嶋田創[†] 三輪忍[†]
五島正裕[††] 中島康彦[†]
森眞一郎[†] 富田眞治[†]

我々は，命令のスラック (slack) に基づくクリティカリティ予測を提案している．ある命令の実行を s サイクル遅らせてもプログラムの実行時間が増大しないとき，s の最大値をその命令のスラックという．前回の実行時のスラックを予測表に登録しておくことによって，それを今回の予測値とすることができる．本稿では，スラック予測をクラスタ型スーパースカラ・プロセッサのステアリングに応用する方法を提案する．各命令の実行後に得られるスラックの値によって，その命令が次回実行時に使用するクラスタを決定する．シミュレーションによる評価の結果，発行幅が 4 のプロセッサを 2 つのクラスタに分割した場合，クラスタ化されていないプロセッサに比べ約 10 % IPC が低下することが分かった．

Instruction Steering for Clustered Superscalar Processor with Slack Prediction

Tomohisa Fukuyama,[†] Hajime Shimada,[†] Miwa Shinobu,[†] Masahiro Goshima,[††]
Yasuhiko Nakashima,[†] Shin-ichiro Mori[†] and Shinji Tomita[†]

We proposed an instruction criticality prediction technique based on prediction of instruction slacks. When the execution time of a program doesn't become longer even if an instruction of the program is delayed by s cycles, the maximum of s is referred to as the slack of the instruction. The slack value is stored to the prediction table to be a predicted value for the next time. This paper describes instruction steering of clustered processor with slack prediction. The cluster that a instruction will use at the next time is decided by the slack value given after the execution of the instruction. Evaluation result shows IPC is reduced 10 % in comparison with non-clustered processor.

1. はじめに

命令の**クリティカリティ** (criticality)，すなわち，命令がどれほどクリティカルかを知ることは，スーパースカラ・プロセッサの高性能化と省電力化の両方に効果がある．例えば，命令をスケジューリングするときには，よりクリティカルな命令を優先的に発行した方がよい．また，クリティカルでない命令のみを低速 / 低消費電力の演算器で実行することで，性能を大きく低下させずに省電力化を図ることができる[1)~5)]．

さて，従来このような研究の多くは，プログラムのクリティカル・パスに基づいて行われてきた．しかし，この方法には，以下のような問題点がある：

(1) **論理的**　演算器の数やキャッシュ・ミスなど，実行しているプロセッサの物理的な制約が反映されていない．

(2) **二値的**　最もクリティカルな命令を教えるのみで，それ以外の命令がどの程度クリティカルで**ない**のか判定できない．

(3) **クリティカル・パスの判定が困難**　実行中のプログラムのクリティカル・パスを判定することはそれほど容易ではない．

一方，我々はクリティカル・パスではなく命令の**スラック** (slack)[6)] によって，命令のクリティカリティを測ることを提案した[7),8)]．ある命令の実行を s サイクル遅らせてもプログラムの実行時間が増大しないような s の最大値をその命令のスラックという．したがって，クリティカルな命令のスラックは 0 サイクルである．

スラックはデータの定義時刻 とそのデータの使用時刻の差で求められ，前回実行時のスラックを予測表に登録しておくことによって，それを今回の予測値とすることができる．

図 1 に，命令が実行される様子を表わすタイム・チャートを示す．図中，“I” が命令の実行を表し，“I” の長さはその命令の実行レイテンシを表す．上下の “I” の間にある横線は，フロー依存関係を表す．

同図では，命令 I_1 が定義した結果を最初に使用す

† 京都大学 Kyoto University
†† 東京大学 University of Tokyo

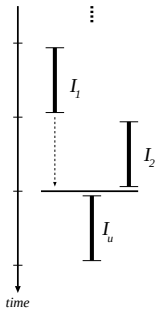

図 1 タイム・チャート

る命令は I_u となっている．すると I_1 のスラック s は，原則的には，定義命令 I_d による定義時刻 t_d と，使用命令 I_u による使用時刻 t_u の差，つまり：

$$s = t_u - t_d - 1 \quad (1)$$

によって得られる．この式に従えば，図 1 の I_1 のスラックは 1 サイクル，I_2 のスラックは 0 サイクルとなる．なお，以下ではスラックの単位を省略し，「命令 I_1 のスラックは 1」のように言うことにする．

スラック予測器を用いることで，前述した問題点は以下のように解決されると期待できる：

(1) **実効的**　履歴に基づいてスラックを予測するので，物理的，実効的なクリティカリティが反映される．

(2) **多値的**　クリティカリティの大 / 小は，スラックの小 / 大によって多値的に表現される．よって，例えばスラックの値がそれぞれ 0，1，10 である 3 つの命令があった場合，スラックが小さい 2 つの命令を優先して実行することができる．

(3) **スラックの判定は容易**　クリティカル・パスとは異なり，スラックは (1) で容易に求めることができる．

本稿では，スラック予測を用いた高速化へのアプローチとして，クラスタ型スーパースカラ・プロセッサ[9] における命令ステアリングに，スラック予測器による予測結果を用いる方法について述べる．

クラスタ型スーパースカラ・プロセッサでは，実行ユニットをクラスタと呼ばれる複数のグループに分割し，クラスタ間のオペランド・バイパスを省略する．このため，クラスタ内のバイパスは約 1 /（クラスタ数）に短縮される．よって，高速なバイパシングが可能となり，プロセッサの高クロック化に繋がる．

一方，IPC（Instruction Per Cycle）は低下傾向にある．依存関係にある命令を別クラスタで実行した場合，オペランド・バイパスが利用できない．そのため，同一クラスタで実行した場合に比べ，クラスタ間データ転送に数サイクル余分に遅延が生じ，依存命令の Wakeup が遅れてしまう．また，個々のクラスタはクラスタ化されていないプロセッサに比べスループットが小さいため，特定のクラスタに命令が集中することにより Wakeup 済み命令の Select が遅れてしまう．よって，依存関係にある命令を同一クラスタにステアリングし，なおかつ各クラスタの負荷が均一になるようなバランス良いステアリング方式が求められる．

既存のステアリング方式の多くは，レジスタ・リネーミングの際に依存する先行命令がどのクラスタで実行されるかを知り，その後にそれぞれのステアリング・アルゴリズムに従いクラスタを決定する．そのため，フロントエンドが複雑になりがちである．

そこで，我々は各命令の実行後に得られるスラックの値によって，その命令が次回実行時に使用する命令を決定するステアリング方式を提案する．決定したクラスタ番号はスラックと同じ予測表に記憶し，次回の使用クラスタ番号とする．これにより，フロントエンドに複雑なロジックがなくても依存関係を考慮することができ，高精度な命令ステアリングが期待できる．

以下，2 章でスラック予測器について，3 章で用いたクラスタ型スーパースカラ・プロセッサの構成と基本的な命令ステアリングについて，4 章 でスラックを用いた命令ステアリングについて述べた後，5 章で評価を行う．

2. スラック予測器

本章では，スラック予測器の基本的な実装について述べる．以下，まず 2.1 節でスラック予測器のデータ構造についてまとめた後，2.2 節，2.3 節で予測器に対する登録，参照といった操作とアクセス・タイミングについて説明する．

2.1 スラック予測器の構成

スラック予測器は，主に**スラック表**と**定義表**の 2 種の表からなる．

スラック表

スラック表は，命令の過去のスラックを記録する予測表本体であり，値予測における VHT（Value History Table）に相当する．なお，本稿で述べる命令ステアリングでは，スラック表にクラスタ番号を記録するためのフィールドを付加してある．詳細は 4 章 で述べる．

定　義　表

定義表は，各データに対し，以下を記録する：

(1) **定義時刻**　そのデータが定義された時刻

(2) **定義命令**　そのデータを定義した命令

(3) **クラスタ番号**　使用したクラスタ番号

定義表は，スラック表に記録するスラック自体を計算するために用いられる．論理的には，レジスタ・ファイルやメモリ上の各データに対して，定義時刻，定義命令を記録するフィールドを付加したものと考えてよい．データが使用されるとき，データと同時に定義表に記録された定義時刻を読み出せば，定義命令のスラックを計算することができる．

ただし，システム中のすべてのデータに対して定義時刻，定義命令，クラスタ番号を記録することは非現実的である．予測精度とハードウェア・コストのバランス

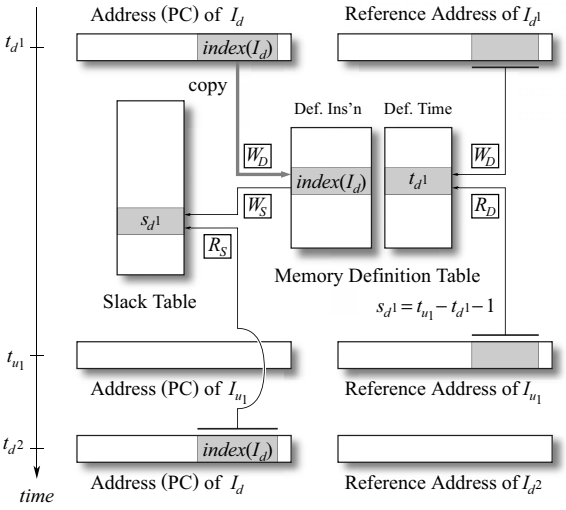

図 2　予測器に対する操作（ストア命令の場合）

をとるためには，アクセス頻度の高いロケーションに対してエントリを提供することが肝要である．

アクセス頻度を考慮して，定義表は，データの格納場所がレジスタかメモリかによって 2 つに分け，それぞれ以下のように実装する：

(1) **レジスタ定義表**　物理レジスタ番号をインデクスとする RAM によって構成する．

(2) **メモリ定義表**　ロード / ストア命令の参照アドレスをインデクスとするキャッシュとして構成する．

2.2　スラック予測器の動作

以下では，命令 I_d の n 回目 ($n \in \mathbf{N}$) の実行を，肩付き数字を用いて，$\boldsymbol{I_{d^n}}$ のように表すことにする．また，I_{d^n} が定義したデータを最初に使用する命令の実行を $\boldsymbol{I_{u_n}}$ と表す．なお，I_{d^i} と I_{d^j} $(i, j \in \mathbf{N},\ i \neq j)$ は同じ命令であるが，I_{u_i} と I_{u_j} は一般に異なる．

図 2 では，ストア命令 I_{d^1} とロード命令 I_{u_1} が，それぞれ，時刻 $\boldsymbol{t_{d^1}}$ および $\boldsymbol{t_{u_1}}$ に実行されている．スラック表への登録，および，同スラック表への参照，すなわち，予測は，以下の様に行われる：

(1) **登録**　登録は，以下のように，(a) I_{d^1} がデータを定義するときと，(b) I_{u_1} がそのデータを使用するとき の 2 つのフェーズからなる：

(a) **定義**　I_{d^1} がデータを定義するとき，以下の操作が行われる：

$\boldsymbol{W_D}$　現時刻 t_{d^1} と I_{d^1} 自身（のアドレス）を定義表に書き込む．

(b) **使用**　I_{u_1} がそのデータを使用するとき，以下の 2 つの操作が連続して行われる：

$\boldsymbol{R_D}$　定義表を読み出して，定義時刻 t_{d^1} と定義命令 I_{d^1}（のアドレス）を得る．

$\boldsymbol{W_S}$　定義時刻 t_{d^1} と現時刻 t_{u_1} から，I_{d^1} のスラック s_{d^1} が，$s_{d^1} = t_{u_1} - t_{d^1} - 1$ と求まる．スラック表の定義命令 I_{d^1} のエントリに，求めた s_{d^1} を書き込む．

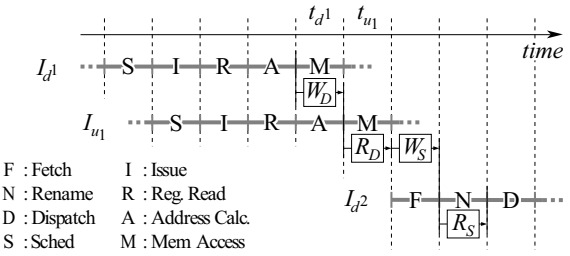

図 3　予測器に対する操作のタイミング（ストア命令の場合）

(2) **予測**　命令 I_d が再びフェッチされると，以下の操作が行われる：

$\boldsymbol{R_S}$　I_d のアドレスをインデクスとしてスラック表を直接読み出すことで，前回のスラック s_{d^1} が得られる．

ストア命令以外の場合には，基本的には，メモリ定義表をレジスタ定義表に，参照アドレスを物理レジスタ番号に，それぞれ読み替えればよい．

なお，スラックの計算を行うのは，そのデータが最初に使用されるときのみである．よって，R_D において，読み出された定義表のエントリを無効化し，同じエントリが繰り返し読み出されないようにする．これは，メモリ定義表のエントリの有効利用にも効果がある．

2.3　スラック予測器へのアクセス・タイミング

図 3 に，前述したストア命令の場合の予測器へのアクセスのタイミングを示す．図中の W_D〜R_S は，前述した操作と対応している．スラック予測器へのアクセス・タイミングは，以下のとおりである：

(1) **登録**　ストア命令の場合には，定義側，使用側共に，アドレス計算（図 3 中，A）の次のステージからアクセスを開始できる．一方，ストア以外の命令の場合，表へのインデクスとして用いる物理レジスタ番号が既に分かっているため，1 サイクル早く，発行の次のステージからアクセスを開始できる．

(2) **予測**　分岐予測器，値予測器などと同様，命令のアドレスをもってアクセスすればよいので，命令フェッチと同時にアクセスを開始することができる；ただし，得られたスラックは専ら命令スケジューリングに使用されるため，ディスパッチ（図 3, D）に間に合うように読み出せばよい．

3.　クラスタ型 スーパースカラ・プロセッサ

本章では，まず 3.1 節で対象とするクラスタ型スーパースカラ・プロセッサの構造について述べ，3.2 節で基本的な命令ステアリングについて述べる．

3.1　プロセッサの構成

図 4 に今回用いたクラスタ型スーパースカラ・プロ

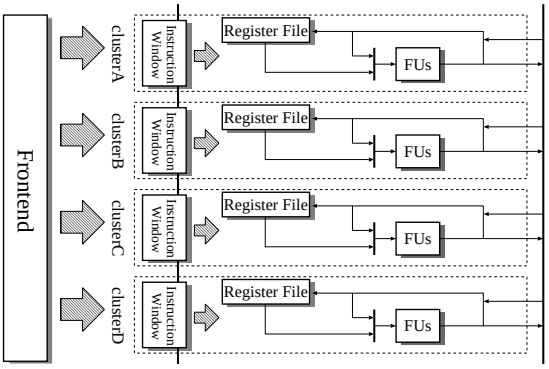

図 4 クラスタ型スーパースカラ・プロセッサ

セッサの構造を示す．各クラスタは少数の実行ユニットから成っており，クラスタ化されていないプロセッサに比べオペランド・バイパスが短いため高速に動作させることができる．一方，クラスタ同士はバスで結合されており，クラスタ間のデータの通信はこのバスを通じて行われる．しかし，この通信はクラスタ内で行われるオペランド・バイパシングに比べ，数サイクル余分に時間が掛かってしまう．

命令ウィンドウ

図 4 に示すように，命令ウィンドウも各クラスタに分散させている．これにより，分散させない場合に比べ，命令ウィンドウ自体のサイズ，各実行ユニットへの発行幅が共に小さくなり，高速に動作させることができる．フロントエンドでフェッチ，デコード，リネームなどの処理をされた命令はいずれかのクラスタの命令ウィンドウにディスパッチされ，実行条件が揃えばクラスタ内の実行ユニットに発行される．

レジスタ・ファイル

今回対象としたクラスタ型スーパースカラ・プロセッサでは，命令ステアリングに焦点を絞るため，比較的実装が容易な複製レジスタ方式を採用した．各クラスタのレジスタは同じ内容を持ち，あるクラスタでレジスタが更新されるときにはその内容を他の全てのクラスタにブロードキャストする．

3.2 命令ステアリング

冒頭でも述べたように，クラスタ型スーパースカラ・プロセッサでは命令ステアリングの質がその性能を左右する．本節では，基本的な 2 つのステアリング方式について説明する．

Round-Robin

連続した N 個の命令を同じクラスタにステアリングする．X 番目にフェッチされた命令は，クラスタ数を C とすると [X / N mod C] 番のクラスタへステアリングされる．特定のクラスタに負荷が集中しにくい一方で，依存関係にある命令が別々のクラスタにステアリングされやすくなる．

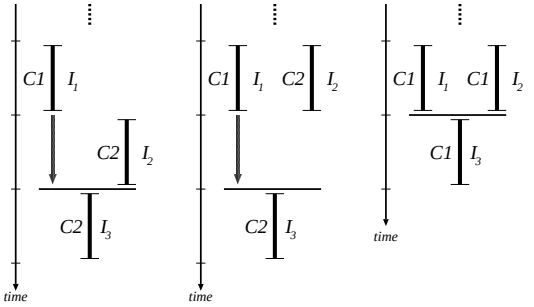

図 5 スラックの利用・計算

Dependence Base

命令は依存関係にある先行命令と同じクラスタにステアリングされる．この方式では，クラスタ間通信による遅延の影響を受けにくい一方，特定のクラスタに命令が集中しやすくなる．また，ステアリング時に命令のクリティカリティ情報を用い，クリティカルでないと判定された命令を負荷の少ないクラスタにステアリングするという方式も提案されている[10),11)]．

しかし，これらの方式は，リネーム・ロジックによる依存関係の解析後に使用するクラスタを決定するため，1 ステージ程度のステアリング・ステージを追加する必要がある．

4. スラック予測を用いた命令ステアリング

本章では，スラック予測をクラスタ型スーパースカラ・プロセッサの命令ステアリングに応用する方法について述べる．以下，まず 4.1 節，4.2 節でスラックの利用方法と計算方法についてそれぞれ述べた後，4.3 節で提案するステアリング方式として Consumer Base と Producer Base のステアリングについて述べる．

4.1 スラックの利用

図 5 はクラスタ型スーパースカラ・プロセッサにおける命令の実行の様子を表すタイム・チャートである．図中の “I” の左側の添字 $C1$，$C2$ は命令を実行したクラスタ番号を表しており，矢印はクラスタ間通信による遅延（この図では 1 サイクル）を表している．つまり，図 5・左でクラスタ $C1$ で実行される I_1 とクラスタ $C2$ で実行される I_3 との間には通信遅延として 1 サイクル必要となり，両方ともクラスタ $C2$ で実行される I_2 と I_3 の間には通信遅延は生じない．

1 章でも述べたように，スラックはサイクルを単位とし多値的に表現される．よって，クラスタ間通信遅延が x サイクルの場合，スラックが x 以上の命令を後続命令と別クラスタで実行しても後続命令の実行開始は遅れない．例えば，図 5・左のようにスラックが 1 の命令 I_1 と後続命令 I_3 を別クラスタで実行した場合，クラスタ間通信遅延により I_1 の実行結果を使用できる時刻が 1 サイクル遅れるが，I_3 の実行開始は遅れておらず，

プログラム全体としての実行時間は増大しない．一方，図 5・中ではクラスタ間通信遅延により I_3 の実行が 1 サイクル遅れている．このような場合，図 5・右のように I_1，I_2，I_3 の 3 命令を同じクラスタで実行することにより，1 サイクル早く I_3 を実行できる．

また，スラック予測はプロセッサの物理的な制約も考慮している．つまり，特定クラスタへ命令が集中し Select が遅れた場合，次回実行時にはそれらの命令のスラックは大きく予測される．よって，スラックが大きな命令を負荷の少ないクラスタへステアリングする事により特定クラスタへの負荷の集中も防ぐことができると期待できる．

4.2 スラックの計算

クラスタ型スーパースカラ・プロセッサにおいてスラックの計算をする際には注意が必要である．

データを定義する命令と使用する命令を別クラスタで実行する場合，スラックの計算で用いる定義時刻は定義命令が実行されたクラスタでデータを定義した時刻であり，別クラスタで使用可能になった時刻ではない．つまり，図 5・左のような場合，1 章の式 (1) に基づいてスラックの計算をすると，I_1 のスラックは 1，I_2 のスラックは 0 となる．その結果，次回実行時にも I_1 のみ別クラスタで実行しても，通信遅延により I_3 の実行開始は遅れないと予測される．

一方，図 5・中のような場合，前述したように 3 つの命令を同一クラスタにステアリングすることにより，I_3 を 1 サイクル早く実行できる．ところが，式 (1) に基づき計算すると，I_1，I_2 の両方のスラックが 1 となり，次回実行時には 3 つの命令が別々のクラスタで実行されてしまう可能性がある．

そこで，定義表にデータを定義した命令が使用したクラスタを記憶しておき，クリティカルになっている命令が全て使用命令とは別のクラスタで実行されていれば，クラスタ間通信による遅延の分だけスラックを小さくする[12]．つまり，図 5・中のように，クリティカルな命令が全て別のクラスタで実行された場合（図 5・左では，I_3 と同じクラスタで実行された I_2 がクリティカルになっている）には，式 (1) により得られるスラックから遅延分の 1 を引く．結果，図の I_1，I_2 の両方のスラックは 0 となり，次回実行時には図 5・右のように全ての命令を同じクラスタで実行できるようになる．

4.3 スラック予測を用いた命令ステアリング

本稿で提案するステアリング方式では，各命令の実行後に得られるスラックの値によって，その命令が次回実行時に使用するクラスタを決定し，そのクラスタ番号をスラック表に登録する．そして，その命令が再度実行される時には，スラック表を読み出すことにより今回使用するクラスタ番号を得る．命令が初めて実行される時やスラック表のエントリがリプレースされるなどしてアクセス・ミスした場合には，負荷が最小のクラスタにステアリングする．

Consumer Base

スラックの計算の結果，データを定義した命令とデータを使用した命令（**Consumer**）を同じクラスタで実行した方が良いと判断した場合，次回実行時には使用命令が今回使用したクラスタで両方の命令を実行するようにする．同じクラスタで実行する必要がないと判断した場合には，定義命令は今回使用したクラスタを次回も使用する．

2.2 節で述べたように，定義命令のスラックの計算は使用命令のレジスタ読み出しまたはメモリ・アクセスの時に行われる．スラックの計算の結果，得た値がクラスタ間通信遅延より小さければ，定義命令と使用命令を同じクラスタで実行した方がよいと判断し，使用命令が実行中であるクラスタ番号をスラック表の定義命令のエントリに登録する．逆に，スラックの値がクラスタ間通信遅延以上であれば，定義表に登録してある定義命令が前回使用したクラスタ番号をスラック表に登録する．

この方式では，スラックがクラスタ間通信遅延よりも小さい定義命令が複数あれば，次回実行時には全て同じクラスタで実行することが可能となる．一方，使用命令が次回実行時に使用するクラスタは，その後続の命令により決定され，今回と同じクラスタで実行されるとは限らない．そのため，スラックの値がクラスタ間通信遅延よりも小さい一連の命令列を全て同じクラスタで実行できるようになるのは，数回の実行後となる．

Producer Base

Consumer Base とは逆に，定義命令（**Producer**）と使用命令を同じクラスタで実行した方がよいと判断した場合には，次回実行時には定義命令が今回使用したクラスタで使用命令を実行する．

スラックの計算の結果，スラックがクラスタ間通信遅延より小さければ，定義命令が今回使用した，もしくは，定義命令の先行命令によって定義命令が次回実行するクラスタが決められていればそのクラスタ番号を，定義表の使用命令のエントリに使用命令が今回使用したクラスタ番号とは別に登録する．定義命令については，次回使用するクラスタが登録されていればそのクラスタ番号を，されていなければ今回使用したクラスタ番号をスラック表に登録する．このように，Producer Base では今回使用したクラスタ番号だけでなく次回使用するクラスタ番号も定義表に登録する必要がある．

この方式では，一度の実行で連続する命令列を全て同じクラスタで実行することができるが，定義側の命令にスラックの値がクラスタ間通信遅延より小さいものが複数あった場合には，そのうちの 1 つしか使用するクラスタ番号を登録できず，他の命令は次回も同じクラスタで実行されてしまう．

図 6，**図 7** に Consumer Base，Producer Base を用いた場合の使用クラスタの遷移を示す．図 6 は Consumer Base が有利になる場合であり，図 7 は Producer Base

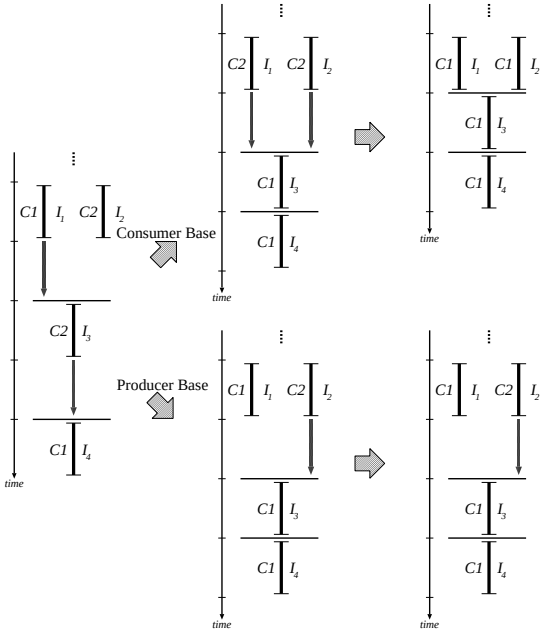

図 6 Consumer Base が有利な場合における使用クラスタの遷移

が有利になる場合である．

図 6･左では，I_1 と I_3，I_3 と I_4 がそれぞれ別クラスタで実行されているのでクラスタ間通信による遅延が生じる．その結果，全ての命令を同じクラスタで実行したときに比べ，合計で 2 サイクル I_4 の実行が遅れる．

Consumer Base を用いた場合，I_3 の実行時に I_1，I_2 のスラックが 0 と計算されるので，この 2 つの命令が次に実行される時には，I_3 が今回使用したクラスタ $C2$ で実行される．一方，I_4 の実行時も同様に I_3 のスラックが 0 と計算されるので，I_3 が次回実行されるクラスタは I_4 が今回使用した $C1$ となる．その結果，これらの 4 つの命令が次に実行されるときには，図 6・中上のように，I_1 と I_2 がクラスタ $C2$ で，I_3 と I_4 がクラスタ $C1$ で実行され，I_4 の実行タイミングは前回と比べ，1 サイクル早くなる．同様に，さらに次の実行時には，図 6･右上のように，4 つの命令が全て同じクラスタで実行されクラスタ間通信による遅延は発生しなくなる．

一方，Producer Base を用いた場合，図 6･ 中下では，I_1，I_2 が使用するクラスタは前回使用したクラスタと同じであり，I_3，I_4 が使用するクラスタは I_1 と同じクラスタ $C1$ である．その結果，Consumer Base の場合と同様，I_4 の実行タイミングは 1 サイクル早くなる．しかし，さらに次の実行時では，図 6･右下のように，4 つの命令の使用クラスタは変わらないので，I_4 の実行タイミングは Consumer Base の場合に比べ，毎回 1 サイクル遅れることになる．

図 6･左のような場合では Consumer Base が有利であったが，逆に，図 7･左のような場合では Producer Base が有利となる．つまり，Producer Base の場合，

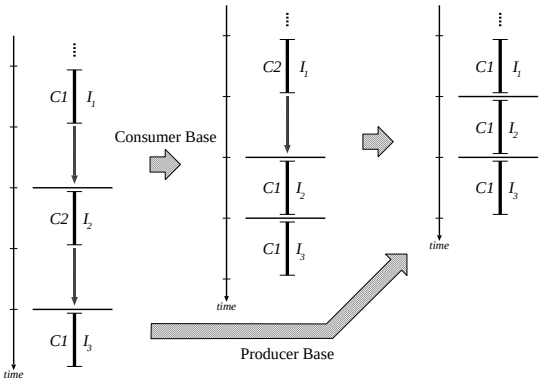

図 7 Producer Base が有利な場合における使用クラスタの遷移

図 7･左の次の実行時には，右の図のように，I_1，I_2，I_3 の 3 つの命令を全て同じクラスタで実行し I_3 の実行タイミングが 2 サイクル早くなる．一方，Consumer Base の場合では，図 7･中を経由するので，図 7･右のように実行できるのに，Producer Base よりも 1 回多く実行しなければならない．

このように，命令の実行後に次回実行時の使用クラスタを決定することにより，フロントエンドでの依存関係の解析を待たずに依存関係を考慮した命令ステアリングが可能となる．また，2.3 節でも述べたように，スラック表へのアクセスは命令のアドレスを用い，命令フェッチと同時に開始できる．よって，この方式では，3.2 節で述べた Dependence Base のようにディスパッチの前にステアリング･ステージを必要としない．このことは，分岐予測ミス･ペナルティの軽減にも効果がある．

5. 評　　価

シミュレーションにより，4 章で提案したステアリング方式の評価を行う．

5.1 評 価 方 法

SimpleScalar ツールセット (ver. 3.0) の sim-outorder シミュレータに対して，スラック予測器とクラスタ型スーパースカラ･プロセッサを実装し，SPEC ベンチマークを用いて本稿で述べた命令ステアリングの効果を測定した．測定には SPEC CINT2000 の 8 本のプログラムを使用した．最初の 10 億命令をスキップし，続く 1 億命令を実行した．

プロセッサのモデル

プロセッサの主なパラメタを**表 1** に示す．評価で用いたプロセッサはクラスタを 2 つ持ち，各クラスタの命令ウィンドウのサイズは 32 エントリとした．各クラスタは全種類の命令を実行できる，いわゆる万能実行ユニットを 2 つずつ持っている．つまり，各クラスタ毎の

発行幅は 2 命令となる．クラスタ間の通信遅延は 2 サイクルとした．

また，評価の対象としたモデルは，分離 (separate) ロード / ストア方式を採用している．すなわちロード / ストア命令は，ディスパッチ時に，アドレス計算を行う命令と，実際にメモリ（キャッシュ）アクセスを行う命令に分離され，個別にスケジューリングされる．評価では，分離されたそれぞれに対してスラックを予測し，スケジューリングを行っている．したがって，最初からアドレス計算命令とメモリ・アクセス命令の 2 つの命令があったものと考えてよい．

テーブルのパラメタ

表 2 に，テーブルのパラメタを示す．スラック表，および，メモリ定義表の容量は，それぞれ，1次命令，および，1 次データ・キャッシュと同じ範囲をカバーできるようにした; すなわち，それぞれ 8Kエントリである．ただし連想度は，1次命令，および，1次データ・キャッシュがそれぞれ 2 であるのに対して 4 とした．このように，やや大きな容量 / 連想度としたのは，ミスによる影響を評価結果から除外するためである．メモリのレイテンシ（18 サイクル）は，メモリ・インタフェイスを集積する AMD Athlon プロセッサのものを参考にした．

ステアリング方式

測定に用いたステアリング方式は次の 5 つ：

RR　3.2 節 で述べた Round-Robin 型のステアリング方式．各クラスタに均等にステアリングできる一方で，依存関係にある命令を別々のクラスタにステアリングしてしまう可能性が高い．

DB　3.2 節 で述べた Dependence Base のステアリング方式．依存関係にある命令を同じクラスタにステアリングできるが，特定のクラスタに命令が集中してしまう．

WS　Dependence Base にクリティカリティ予測としてスラック予測を用い（With Slack），スラックの値がクラスタ間通信遅延以上の命令を負荷の小さなクラスタへステアリングする方式．DB 方式に比べ，特定のクラスタに命令が集中しにくい．

CB　4.3 節 で述べた Consumer Base のステアリング方式．次回実行時に，Consumer 側の命令が使用したクラスタで Producer 側の命令を実行する．

PB　4.3 節 で述べた Producer Base のステアリング方式．次回実行時に，Producer 側の命令が使用したクラスタで Consumer 側の命令を実行する．

表 1　プロセッサの各パラメタ

パラメタ	サイズ
フェッチ幅	4
コミット幅	4
クラスタ数	2
クラスタ毎のパラメタ	
クラスタ間通信遅延	2 サイクル
命令ウィンドウ	各 32
発行幅	各 2
分岐予測器	
予測方式	gshare
PHT	4K エントリ
グローバル分岐履歴長	12
ミス・ペナルティ	6 サイクル
リターン・アドレス・スタック	8 エントリ

表 2　各表，キャッシュ，メモリのパラメタ

		容量	ライン サイズ	連想度	レイテンシ (cycles)
スラック表		8K命令	—	4	1
レジスタ定義表		64命令	—	64	1
メモリ定義表		8K命令	—	4	1
1 次	命令	8K命令*	8命令*	2	1
	データ	8Kワード	8ワード	2	1
2 次		1MB	64B	2	6
メモリ		—	—	—	18†

*: SimpleScalar ツールセットでは 8B/命令.

†: 最初のワード．後続ワードには 2 サイクル / ワードが必要.

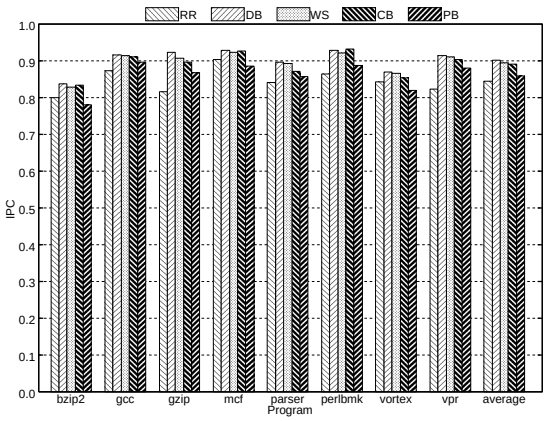

図 8　各ステアリング方式における IPC の比

これらの 5 つの方式をクラスタ化されていないプロセッサと IPC を比較することにより評価を行う．クラスタ化されていないプロセッサの命令ウィンドウの大きさ，発行幅はクラスタ化されたプロセッサのそれの合計，つまりそれぞれ 32 エントリ，4 命令となる．比較は IPC のみにより行い，動作クロックなどは考慮しない．

なお，DB と WS についてはステアリング・ステージがフロントエンドに追加されるとし，分岐予測のミス・ペナルティを他の方式よりも 1 大きい 7 サイクルとし，ステアリングに用いる負荷情報は命令ウィンドウ内の命令数とした．

また，総発行幅が 4 でクラスタ数を 4 とした場合と，実効幅，総発行幅が 8 でクラスタ数を 2，4，8 とした場合についても測定を行った．IPC の基準としたクラスタ化されていないプロセッサの発行幅は，クラスタ型プロセッサの総発行幅と同じであるが，命令ウィンドウのサイズは 64 エントリで統一した．

5.2 評価結果

図 8，**図 9** に結果を示す．2 つの図には，それぞれ

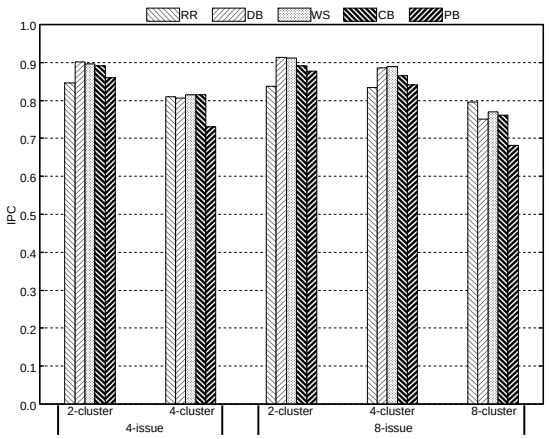

図 9 各プロセッサ構成における IPC の比

の場合とクラスタ化されていないプロセッサとの IPC の比を表している．図 8 には 9 組のバーがあり，左の 8 組は SPEC CINT2000 の 8 つのベンチマーク・プログラムに対応しており，最右 1 組はそれらの平均である．一方，図 9 には 5 組のバーがある．それぞれ左から，総発行幅が 4 でクラスタ数が 2，4 の場合，総発行幅が 8 でクラスタ数が 2，4，8 の場合の 8 つのプログラムの平均である．すなわち，図 8 の最右と図 9 の最左は同じグラフを表している．2 つの図の各組にはバーが 5 本ずつあり，前節で述べた 5 つのステアリング方式に対応している．

図 8 では，いずれのプログラムでも同様の傾向が見られる．IPC の低下は，平均して RR で約 15 %，DB，WS，CB でそれぞれ約 10 %，PB で約 14 %であった．

また，図 9 をみると，CB はいずれの場合においても，DB，WS と同程度の IPC であることから，依存関係を考慮した命令ステアリングができていると言える．しかし，各クラスタの発行幅が小さくなると，RR との差が小さくなっている．特に，総発行幅が 8 でクラスタ数が 8 の場合では，RR よりも 3 % 程度大きな IPC 低下が見られる．このことは，CB はクラスタの負荷分散が十分にできていないことを示している．

一方，PB はいずれの場合においても，CB より大きな IPC 低下が見られる．特に，各クラスタの発行幅が小さい場合に，大きく IPC が低下している．

6. おわりに

本稿では，スラック予測をクラスタ型スーパースカラ・プロセッサにおける命令ステアリングに応用する方法について述べた．

評価の結果，スラック予測を用いることでフロントエンドに複雑なステアリング・ロジックを用いずに，用いた場合と同程度の IPC を保つことができることが分かった．しかし，IPC の向上はなく，個々のクラスタの発行幅が小さいときには大きな IPC 低下が見られた．その原因として，本稿で提案した方式では柔軟なステアリングができない点が挙げられる．4.3 節でも述べたように，提案方式では命令の実行後に次回使用するクラスタを決定する．つまり，その命令が実際にステアリングする際，クラスタの負荷情報が考慮されることはない．よって，ステアリングされたクラスタの負荷が大きい場合，その命令の実行は遅れ IPC の低下の原因となる．

今後は，ステアリング精度の向上，分散レジスタ方式への応用，ハードウェアへの実装などを検討している．

参 考 文 献

1) Fields, B. and Blodik, S. R. R.: Focusing Processor Policies via Critical-Path Prediction, *28th Int'l Symp. on Computer Architecture (ISCA-28)*, pp. – (2001).
2) Fields, B., Bodik, R. and Hill, M. D.: Slack: Maximizing Performance under Technological Constraints, *29th. Int'l Symp. on Computer Architecture (ISCA-29)* (2002).
3) Tune, E., Liang, D., Tullsen, D. M. and Calder, B.: Dynamic Prediction of Critical Path Instructions, *7th Int'l Symp. on High Performance Computer Architecture (HPCA7)* (2001).
4) Grunwald, D.: Micro-architecture Design and Control Speculation for Energy Reduction, *Power Aware Computing*, Kluwer, ISBN 0-306-46786-0, chapter 4 (2002).
5) 千代延昭宏ほか: プログラム実行時における命令の重要度決定に関する検討, *SWoPP*, pp. 1–6 (2003).
6) Casmira, J. and Grunwald, D.: Dynamic Instruction Scheduling Slack, *Kool Chips Workshop (in conjunction with MICRO-33)* (2000).
7) 劉小路ほか: クリティカリティ予測のためのスラック予測, *SACSIS* (2004).
8) 福田匡則ほか: グローバル分岐履歴を用いたスラック予測器, *SWoPP* (2004).
9) Palacharla, S., Jouppi, N. P. and Smith, J. E.: Complexity-Effective Superscalar Processors, *Proc. 24th Int'l Symp. on Computer Architecture (ISCA24)* (1997).
10) 小林良太郎ほか: データフロー・グラフの最長パスに着目したクラスタ化スーパースカラ・プロセッサにおける命令発行機構, *JSPP* (2001).
11) 服部直也ほか: クリティカルパス情報を用いた分散命令発行型マイクロプロセッサ向けステアリング方式, 情報処理学会 ACS 論文誌, Vol. 45, No. SIG 6(ACS 6), pp. 12–22 (2004).
12) 福山智久ほか: スラック予測を用いた省電力アーキテクチャ向け命令スケジューリング, *SACSIS* (2005).