クリティカリティ予測のためのスラック予測

劉　小路[††]　額田匡則[†]　小西将人[†††]
五島正裕[†]　中島康彦[†]
森眞一郎[†]　富田眞治[†]

本稿では，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する．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．したがって，いわゆるクリティカルな命令のスラックは 0 サイクルである．本稿では，履歴に基づく予測表を用いて命令のスラックを予測する手法について述べる．シミュレーションによって予測精度を評価した．予測スラックが 1 以上の命令の実行レイテンシを 1 サイクル増加させた場合には，IPC は 12.8% 悪化するが，68.9% もの命令を遅らせることができる．予測スラックが 2 以上の命令の実行レイテンシを 1 サイクル増加させた場合には，IPC は 2.8% しか悪化せず，26.7% の命令を遅らせることができる．

Slack Prediction for Criticality Prediction

Liu Xiao Lu,[†] Masanori Nukata,[†] Masahito Konishi,[†] Masahiro Goshima,[†]
Yasuhiko Nakashima,[†] Shin-ichiro Mori[†] and Shinji Tomita[†]

We propose 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 refered as the slack of the instruction. Thus the slack of a critical instruction is zero cycles. We evaluated a hitory-based slack predictor. When the execution latency of instructions with predicted slack of more than one cycle is prolonged by one cycle, IPC decreased by 12.8% but 68.9% instructions are prolonged. When the execution latency of instructions with predicted slack of more than two cycle is prolonged by one cycle, IPC decreased by 2.8% but 26.7% instructions are prolonged.

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．小林らは，クラスタ化された演算器を持つプロセッサ[1] に対して，クリティカル・パスの情報を用いたスケジューリング手法を提案している[2]．

クリティカリティの情報は，省電力アーキテクチャにおいても有用である．例えば，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ることなく省電力化を図ることができる[3)~7)]．

近藤らは，主記憶によるキャッシュ・ミスのサービスを契機として，プロセッサに DVS (dynamic voltage scaling) 制御を行うことを提案している[8] が，これも上述の省電力アーキテクチャの延長線上にあると考えてよい．キャッシュ・ミスの処理を待ってプロセッサがストールしているときには，ミスを起こしたロード命令はクリティカルであり，その他の命令はすべてクリティカルでない．この場合にプロセッサを低電圧化することは，クリティカルでない命令の実行に関わるハードウェアを低電圧化することと考えることができる．

さて，従来このような研究の多くは，論文の題目にも顕著に表われているように，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．しかしクリティカル・パスに基づく方法には，以下のような問題点がある：

(1) **論理的**　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．

† 京都大学
Kyoto University
†† 東京大学
University of Tokyo
††† 大阪工業大学
Osaka Institute of Technology

(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．

その上，本来の目的のためには，命令がクリティカル・パス上にあるかどうかを知りたいのではない．本当に必要なのは，その命令の実行の遅れがプロセッサによるプログラムの実行時間をどの程度増大させるか，すなわち，その命令のクリティカリティそのものである．

そこで我々は，クリティカル・パスではなく，命令の**スラック** (slack)[9] によって，命令のクリティカリティを測ることを提案する．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 である．

本稿では，履歴に基づくスラック予測器について述べる．以下，2 章では，上述したクリティカル・パスに基づく方法の問題点についてより詳しく考察するとともに，スラックとクリティカリティの関係について述べる．3 章では，スラック予測器について述べる．4 章では，3 章で述べたスラック予測器の予測精度を評価した結果を示す．

なお本稿では，スラック予測器の予測精度を評価するに留め，スラック予測器を上述した命令スケジューリングや省電力アーキテクチャに応用した場合の結果までは評価しない．スラック予測器にに必要とされる詳細は応用によって異なるため，研究の初期から応用を絞ると，一般性を失う恐れがあるためである．

2. クリティカル・パスとスラック

前章で述べたように，命令のクリティカリティに関する研究の多くはプログラムのクリティカル・パスに基づいている．しかしクリティカル・パスに基づく手法には，(1) 論理的，(2) 二値的，(3) クリティカル・パスの判定が困難 といった問題がある．そこで我々は，クリティカル・パスではなく，各命令のスラックによって命令のクリティカリティを予測することを提案する．以下，2.1 節でクリティカル・パスに基づく方法の問題点をまとめた後，2.2 節でスラックに基づく方法を導入する．

2.1 クリティカル・パスに基づく方法

クリティカリティに関する研究の多くは，以下のように，プログラムのクリティカル・パスに基づいている．

小林らは，クラスタ化されたプロセッサ[1] の命令スケジューリングのため，パス情報テーブルを提案している[2]．このテーブルは，命令ウィンドウ内にある命令に対して近似的なデータ・フロー・グラフを構築し，そのグラフの最長パス，そして，そのパスの先頭にある命令を答えることができる．

Tune らは，適当なヒューリスティクスに基づいて，『クリティカル・パス上の命令らしさ』を推測している[5]．ヒューリスティクスとしては，例えば，「命令ウィンドウ内で，最も古い命令 (QOLD)」とか「命令ウィンドウ内で，その結果が最も多くの命令に使用される命令 (QCONS)」などといったものである．Tune らは，命令ごとのローカルな履歴のみを用いた予測器を評価しているが，千代延らはいわゆるグローバル履歴を用いた 2 レベル予測器を評価している[10]．

しかし，クリティカル・パスに基づく手法には以下のような問題がある：

(1) **論理的**　データ・フロー・グラフに現れるようなプログラムの論理的な制約のみを考慮しており，実行しているプロセッサの物理的な制約を考慮していない．そのため，以下のような食い違いが生じる：

- **ミス**　キャッシュ・ミスを起こすロード命令，分岐予測ミスを起こす条件分岐命令などは，やはり論理的にはクリティカルではなくても，実効的にクリティカルになる可能性が高い．
- **資源制約**　資源制約，特に演算器の個数が考慮されていない．例えば，同じ演算器を使用する命令が多数連続する場合，それらの命令が論理的にはクリティカルでなくても，実効的にクリティカルになり得る．
- **メモリを介した依存**　小林らの方法も Tune らの方法も，メモリを介した依存を考慮できていない．

(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**のか判定でない：

- 2 番目にクリティカルな命令を優遇しなかった結果として，その命令が実質的にクリティカルになることがあり得る．
- ミス・ペナルティの大きい技術には適用できない．例えば DVS 制御では，電圧の制御に時間がかかるため，単にクリティカルでないと予測されただけでなく，非常にクリティカルでないと予測されなければ適用できない．

(3) **クリティカル・パスの判定が困難**　プログラム全体，あるいは，関数などのクリティカル・パスを求めることに対して，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，原理的に不可能と言ってもよい; 実行中のプログラムの場合，データ・フロー・グラフの始点と終点を定めることができないからである．

また，データ・フロー・グラフは一般的なアサイクリック・グラフであり，ハードウェアで取り扱うにはやや複雑過ぎる．そのため，精度とハードウェア・コストのバランスをとることが難しい:

- 小林らのパス情報テーブルは（命令ウィンドウ内の命令に限れば）クリティカル・パスをかなり正確に推定できるものの，複雑なハードウェアを必要とする．
- Tune らの用いているヒューリスティクスは，実装は比較的容易であるが，実際にクリティカル・パス上の命令を正しく判定できない[7]．

2.2 スラック予測に基づく方法

前章で述べたように，上述した，(1) 論理的，(2) 二値的，(3) 判定が困難 といった問題点は，これらの手法がクリティカル・パスに基づいていることに起因する．その上，本当に必要なのは，その命令のクリティカリティそのものであって，その命令がクリティカル・パス上にあるかどうかではない．

そこで我々は，クリティカル・パスではなく，各命令のスラックによって命令のクリティカリティを測ることを提案する．小林らのパス情報テーブルのように，命令ウィンドウ内の命令のスラックを計算することも考えられるが，本稿では，履歴に基づく予測器によって命令のスラックを予測することを試みる．

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:

- **ミス**　ロード命令がキャッシュ・ミスを起こすかどうか，条件分岐命令が分岐予測ミスを起こすかどうかには，履歴依存性があることが既によく知られている．そのような命令に対して，予測器は小さいスラックを出力すると期待できる．
- **資源制約**　同じ演算器を使用する命令が多数連続するような状況にも，少なからず履歴依存性があると推測される．そのような命令に対しても，予測器は小さいスラックを出力すると期待できる．
- **メモリを介した依存**　計算されたアドレスに基づいて，メモリを介した依存を考慮することができる．

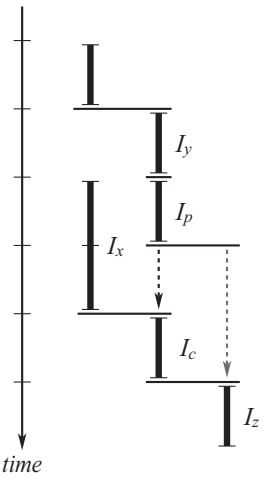

図 1　命令実行のタイム・チャート

(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．そのため，以下のようなことが可能になる:

- 例えば，3 つの命令のスラックがそれぞれ 0, 1，および，10 であった場合，最初の 2 つを優先するとよいだろう．
- 残っている命令のスラックがすべて大きい値，例えば 100 程度以上であった場合には，DVS 制御をかけてもよいだろう．

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

次節では，本稿で扱うスラックについて詳しく説明する．

2.3 近似スラック

図 1 に，命令が実行される様子を表すタイム・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．

図では，命令 I_d が定義した結果を**最初に**参照する命令は I_u である．本稿で提案するスラック予測器は，I_d による定義時刻と I_u による参照時刻の差を I_d のスラックと近似する．したがって図では，I_d のスラックは 1 サイクルとなる．なお，以下ではスラックの単位を省略し，「I_d のスラックは 1」のように言うことにする．

同図は，データ・フロー・グラフではないことに注意されたい．同図では，命令 I_x は，データ依存ではない何らかの理由により，理想的な場合より 1 サイクル遅れて実行されている．命令 I_d のスラックが 1 となるのは，I_x の実行が遅れたことに起因している．データ・フロー・グラフならば，データ依存関係と命令の実行レイテンシだけを考慮するため，I_d のスラックは 0 と定

義される．本稿では，このとき I_x はデータ依存ではない，プロセッサの物理的な制約，例えば，演算器の不足よって遅れたとして，I_d のスラックを 1 とするのである．データ・フロー・グラフ上のスラックは**論理スラック**; 本稿で用いるスラックは**実スラック**と呼んでよいだろう．

なお，上記のスラックの求め方は，前節で述べた一般的なスラックの定義に厳密には従っていない．本稿の求め方によれば，同図の命令 I_y のスラックは 0 となる．前章で述べた,「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」という一般的な定義に厳密に従えば，I_y のスラックも 1 でなければならない☆．しかし，このようなスラックを厳密に求めることは極めて困難であるので，本稿では上記の近似を採用することにする．

3. 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 3.1 節でスラック予測器のデータ構造についてまとめた後，3.2 節で予測器に対する登録，参照といった操作について説明する．3.7 節以降では，実装上の考慮点について考察する．

3.1 スラック予測器の構成

スラック予測器は，主に以下の 2 種の表からなる:

(1) **スラック表**　命令のアドレスをインデクス，スラックを内容とする表.

(2) **定義表**　各データに対し，以下を記録する:

(a) **定義時刻**　そのデータが定義された時刻

(b) **定義命令**　そのデータを定義した命令

スラック表は，命令の過去のスラックを記録する，言わば予測表本体である．一方，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．定義表は，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，定義時刻と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，当然のことながら，システム中のすべてのデータに対して定義時刻と定義命令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．

☆ あるいは，命令 I_y と命令 I_d のスラックは合わせて 1 と考えるのがより正確であろう.

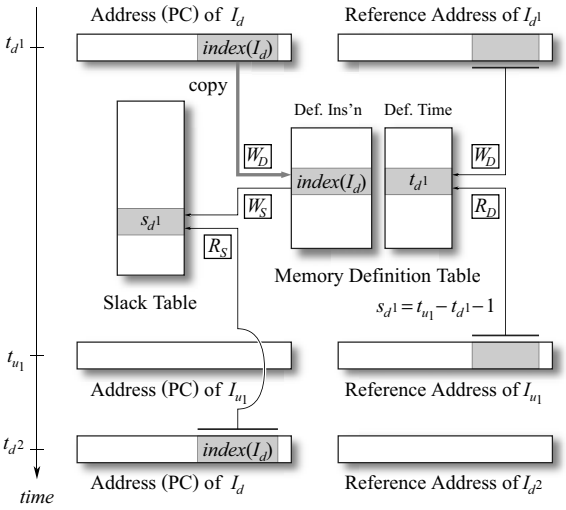

図 2　予測器に対する操作（ストア命令の場合）

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．すなわち，そのエントリ数は物理レジスタ・ファイルに等しく，まさに物理レジスタ・ファイルに定義時刻と定義命令を格納するフィールドを付加したものと考えてよい.

(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.

したがって，メモリ定義表では，ミスが発生することになる．メモリ定義表のミスや，エントリ数と連想度については後で詳しく述べることにして，次節では，これらの表を用いたスラック予測器の動作について述べる．

3.2 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:

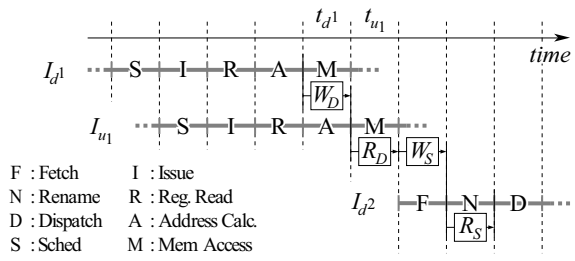

図 3 予測器に対する操作のタイミング（ストア命令の場合）

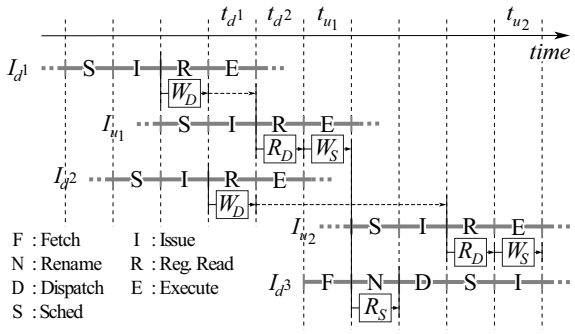

図 4 予測器に対する操作のタイミング（ストア命令以外の場合）

(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：

***W*D** 現時刻 t_{d^1} と I_{d^1} 自身（のアドレスのインデクス部）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：

***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}（のアドレスのインデクス部）を得る．

***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．

(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：

***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．

3.3 スラック予測器へのアクセス・タイミング

図 3 と**図 4** に，前述したストア命令の場合と，それ以外の命令の場合の，予測器へのアクセスのタイミングをそれぞれ示す．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には，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（図 4 中，I）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, 4 中，D）に間に合うように読み出せばよい．

3.4 操作のオーバラップ

命令数の少ないループなどでは，定義（W_D）と使用（R_D，W_S）とからなる一連の登録操作が完了する前に，同じ命令の次の実行が開始されてしまうことがある．その場合，以下の 2 つのオーバラップが生じることになる：

(1) **登録–登録のオーバラップ**　例えば図 4 では，I_{u_1} の W_S より以前に I_{d^2} の W_D が開始されており，I_{d^1}，I_{d^2} の 2 つの実行に対するスラックの登録操作が同時に進行している．

(2) **登録–予測のオーバラップ**　同図 4 では，I_{u_1} によって計算されるスラック s_{d^1} を読み出すことができるのは，I_{d^3} として示したタイミングにフェッチされた場合以降である．I_{d^2} の R_S はそれよりずっと以前に終了している．

以下，それぞれについて述べる．

登録–登録のオーバラップ

2 つの登録がオーバラップするかどうかは，ストア命令とそれ以外の命令の場合で異なるが，いずれの場合であっても，スラック s_{d^1} や s_{d^2} 自体は，以下のようにして正しく計算できる：

ストア以外の命令　レジスタ定義表のエントリは，論理ではなく，物理レジスタに対応している．したがって I_{d^1} と I_{d^2} は，レジスタ・リネーミングによって，レジスタ定義表のそれぞれ異なるエントリを使用することになる．

ストア命令　メモリ投機を行うかどうかによって経過は若干異なるが，同一アドレスに対するロード / ストアは最終的には逐次的に実行される：

行わない場合　同一アドレスに対するロード / ストアは逐次的に実行されるので，図 4 のような状況はそもそも起こり得ない．

行う場合　図 4 のような状況では，メモリ順序違反 (memory order violation) が起きており，これらの命令の実行はキャンセル / 再実行される必要がある．再実行時には，これらの命令は逐次的に実行され，正しいスラックが求められる．

登録–予測のオーバラップ

前述したように，同図 4 において，I_{d^2} の R_S はずっと以前に終了しており，I_{d^2} は s_{d^1} を利用できない．

上述したように，ストア命令の場合には，登録–登録のオーバラップは起こらないが，登録–予測のオーバラップはストア以外の場合と同様に起こり得る．

登録–予測のオーバラップが起こった場合，命令は，更新が間に合った中で最も最近計算されたスラックを読み出すことになる．このようにプログラム・オーダ上で最新のスラックが得られないことは，分岐予測を始めとする他の予測技術でも同様に起こり得ることで，好ましくはないが，致命的なことでもない．

3.5　操作のパイプライン化可能性

図 3，図 4 では，W_D〜R_S の各操作の遅延はそれぞれ 1 サイクル未満であるとしたが，各操作はそれぞれパイプライン化可能である．以下のように，W_D–R_D 間と，R_D–W_S–R_S 間に分けて考えることができる．

W_D–R_D 間

対になる W_D と R_D の間では，W_D で書き込まれた内容が R_D で読み出されなければならない；読み損ねた場合，2 回目（以降）に使用する命令の実行によって，異常に大きいスラックが計算される可能性があるからである．

その上，対になる W_D と R_D は，引き続くサイクルに実行されることも多い．

したがって W_D，R_D をパイプライン化する場合には，適当なバイパス機構を付加する必要がある．前述したように，レジスタ定義表は，物理レジスタ・ファイルの各ワードに対して，定義時刻と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．したがって，物理レジスタ・ファイルに対するオペランド・バイパス機構をメモリ定義表に対して拡張することができる．

ただしレジスタ定義表の場合は，物理レジスタ・ファイルよりずっとタイミング制約が緩い．物理レジスタ・ファイルの場合，書き込みと読み出しが，それぞれ 0.5 サイクル，合わせて 1 サイクルであっても，バイパスが必要になる．それに対してレジスタ定義表の場合には，図 3 に示されているように，合わせて 2 サイクルまではバイパスが不要である．

同様にメモリ定義表に対しても，ストア – ロード命令間のフォワーディング機構を拡張することができる．

R_D–W_S–R_S 間

上記以外に対しては，単純にパイプライン化することも許容できる；ただし，パイプライン化によるレイテンシの増大は，前節で述べた，最新のスラックが得られない期間を増加させることになる．

また，R_D–W_S 間には不可能だが，W_D–R_S 間には W_D–R_D 間と同様のバイパス機構を付加することで，最新のスラックが得られない期間を短縮することができる．

3.6　スラック表，メモリ定義表のミス

メモリ定義表，スラック表は，キャッシュであり，ミスが起こる．メモリ定義表，スラック表共に，W_D，W_S では，ミスが起こっても割り当てられたエントリに上書きするだけであるから，その時点でのミスは性能上影響がない．したがって，W_D，W_S で割り当てられたエントリが，R_D，R_S までにリプレースされてしまった場合のことを考えればよい．

R_D，R_S におけるミスは，以下のようにする：

メモリ定義表　W_D で割り当てられたエントリが R_D までにリプレースされたのだから，定義から参照まで十分時間が経っていると判断し，最大のスラックを予測する．

スラック表　スラックは 0 と予測する．4 章で述べるように，スラックが 0 である命令は全体の半分以上に上り，0 としておいても相応の予測ヒット率が期待できるからである．0 としておけば，スラック表ミスによる性能の低下は生じない．また，5.3 節で述べるような手法も考えられる．

3.7　条件分岐命令

分岐命令は，その実行結果を参照する命令が存在しないため，上述の方法でスラックを計算することはできない．そこで，以下に述べる理由により，分岐予測ミスを起こした分岐命令のスラックは 0，ヒットした分岐命令のスラックは 0 または 1 とする．

分岐予測ミスを起こす分岐命令は，できるだけ早く実行した方がよい．分岐予測ミスを起こす分岐命令より下流にある命令の実行はすべて無駄である．その分岐命令を早く実行すると，この無駄が省かれるとともに，状態回復がそれだか早く開始されるからである．

一方，分岐予測ヒットする分岐命令は，論理的にはクリティカルにはならない．しかし，以下に述べる理由から，できるだけ早く実行した方がよい；未実行のまま (pending) にできる分岐命令の数には，分岐予測に関する資源の量に起因して，他の命令より強い制約がある．例えば，4 章の評価で用いる MIPS R10000 プロセッサの場合，たかだか 4 個の条件分岐命令しか未実

行にできない．資源が不足した場合には，フロントエンドがストールすることになる．

このように分岐命令は，予測ヒット / ミスに関わらず他の命令より早く実行した方がよく，ミスする命令の方がよりクリティカルである．例えば，分岐命令の実行ユニットが 1 つしか空いていない場合には，ヒットするものよりミスするものを先に実行した方がよい．したがって，このことを考慮したい場合には，ミスすると予測される命令のスラックは 0，ヒットすると予測される命令のスラックは 1 とするとよいと考えられる．

4. 評　　価

シミュレーションにより，スラック予測器の評価を行った．なお本稿では，前述したとおり，スラック予測器の予測精度を求めることのみを目的としており，スラック予測器を命令スケジューリングや省電力アーキテクチャに応用した場合の結果までは評価していない．

4.1 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，スラック予測器を実装し，SPEC ベンチマークを用いて予測精度を測定した．**表 1** に示す CINT95 の 8 つのプログラムを実行した．

コンパイラは，gcc (ver. 2.7.2.3) を用いた．最適化オプションは，-O6 -funroll-loops である．

プロセッサのモデル

プロセッサのモデルとしては，MIPS R10000 プロセッサ[11] を用いた．R10000 プロセッサは，整数，ロード / ストア，浮動小数点のそれぞれに，ディスパッチ幅，フェッチ幅 2 命令，深さ 16 命令の命令ウィンドウを持つ．キャッシュ，メモリのパラメタを**表 2** にまとめる．

ただし，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．また，分岐予測には，同ツールセットに用意されている Bimodal 予測器を用いた．

表 1　SPEC CINT95 ベンチマーク・プログラム

プログラム	入力セット	実行命令数
099.go	9 9	132M
124.m88ksim	dcrand.big	120M
126.gcc	genrecog.i	122M
129.compress	10000 q 2131	35M
130.li	train.lsp	183M
132.ijpeg	vigo.ppm -GO	26M
134.perl	primes.in	10M
147.vortex	persons.250	157M

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.

テーブルのパラメタ

表 2 に，テーブルのパラメタをまとめる．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．これらのパラメタに関する考察は，5 章にまとめる．

4.2 予測値と誤差

以下では，まず予測値と誤差の傾向について示す．本節の結果はすべて 134.perl のものであるが，その他のプログラムでもほぼ同様の傾向を示す．

計算されたスラック

まず，**図 5** に，（予測値ではなく）実際に計算されたスラックの度数の分布と累積を示す; すなわち，グラフの横軸はスラックであり，縦軸は割合である．グラフ中，下部の曲線は分布を，上部の曲線は 0 からの累積を表す．

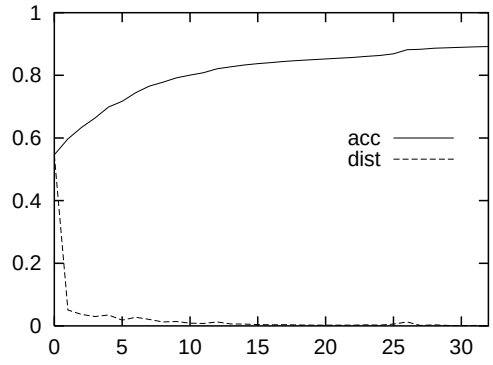

図 5　計算されたスラックの度数の分布（上）と累積（下）

グラフからは，以下のことが読み取れる：

- 半分以上の命令のスラックが 0 である．
- スラックが 1 の命令は 5% ほどあり，以下スラックの増加と共に漸減する．
- スラックがおよそ 30 までで，全命令の 90% 程度をカバーできる．

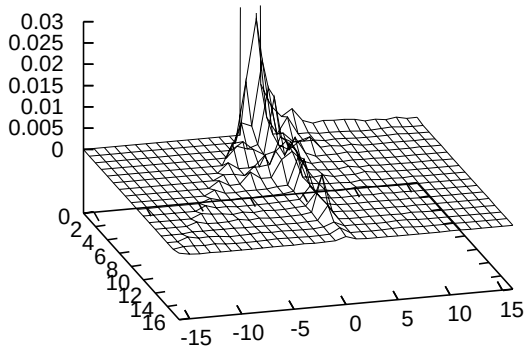

図 6　予測スラックと誤差 の度数分布

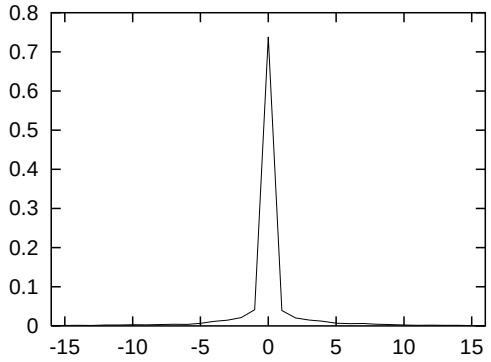

図 7　誤差の度数分布

- 逆に言えば，30 を越える命令が 10% 程度もあり，総量としては無視できない．

予測値と誤差

図 6 に，予測スラックと誤差の度数分布を示す．グラフの x 軸（画面奥行き方向）は，予測スラックを，y 軸（左 / 右方向）は予測値からの誤差を，z 軸はその頻度を表す．

グラフからは，以下のことが読み取れる：

- （グラフでは見切れているが）0 と予測して 0 である場合が極めて多く，全体の半分程度を占める．
- 直線 $y = 0, x = 0, y = -x$ 上に『尾根』が見られる：
  - $\boldsymbol{y = 0}$　予測スラックによらず，誤差が 0 であることが多い．
  - $\boldsymbol{x = 0}$　予測スラックが小さいときに，1〜2 サイクル程度の誤差が生じる場合が比較的多い．
  - $\boldsymbol{y = -x}$　予測スラックによらず実際のスラックが 0 であることが若干多い．

誤　　差

図 7 に，誤差の度数分布を示す．これは，図 6 の度数分布を x 軸方向に累積したものである．

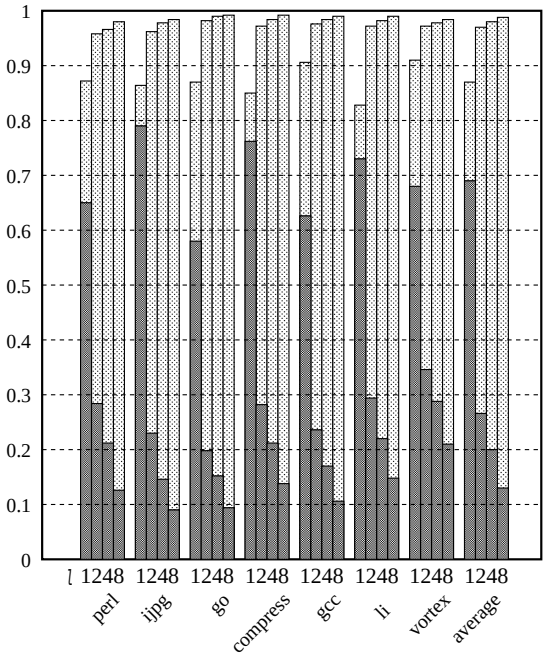

図 8　IPC の比（上）と遅らせることができた命令の割合（下）

グラフからは，以下のことが読み取れる：

- 誤差が 0 である場合が全体の 70% 程度ある．
- 誤差が ±1 である場合が，それぞれ 5% 程度あり，以下絶対値の増加と共に漸減する．

4.3　IPC

予測精度を測るため，予測されたスラックに基づいて命令の実行レイテンシを増加させた場合に，IPC がどの程度低下するかを計測した．具体的には，予測スラックが l 以上である命令の実行レイテンシを 1 サイクル増加させた．同時に，この操作によって全体の何 %の命令の実行レイテンシを増加させることができたかを測った．

図 8 に，結果を示す．グラフには，9 組のバーがある．左の 8 組は SPEC CINT95 の 8 つのプログラムに対応しており，最右の 1 組はそれらの平均である．各組には，上部の薄いバーと下部の濃いバーからなる．薄いバーは，実行レイテンシを増加させないものに対する IPC の比を表し；濃いバーは，実行レイテンシを増加させることができた命令の割合を表す．各組のバーは，左から $l = 1, 2, 4, 8$ の場合のものである．

グラフからは，以下のことが読み取れる：

- 各プログラムでほとんど同じ傾向を示している．
- $l = 1$，すなわち，予測スラックが 1 以上の命令の実行レイテンシを 1 サイクル増加させた場合には，IPC は平均 12.8% 悪化するが，平均 68.9% もの

命令を遅らせることができる．

- $l = 2$，すなわち，予測スラックが 2 以上の命令の実行レイテンシを 1 サイクル増加させた場合には，IPC は平均 2.8% しか悪化せず，平均 26.7% の命令を遅らせることができる．

なお，このように，l の値によって結果が変わることは都合がよい．例えば省電力アーキテクチャならば，$l = 1$ と $l = 2$ を切り替えるなどして，性能と消費電力のバランスをいろいろと変えることができる．

5. 今後の課題

3 章で述べたスラック予測器の構成は，ごく基本的なものであり，以下のような改良が考えられる．

5.1 他の予測器のための技術の応用

値予測を始めとする他の予測で培われた技術をスラック予測器に応用することができる．

グローバル履歴

分岐予測器などの場合と同様に，分岐のグローバル履歴を予測に用いることは予測精度の向上に一定の効果があると考えられる．

タグの省略

値予測などで試みられているように，連想テーブルであるスラック表，メモリ定義表のタグは，省略することができる．タグ，および，タグ・チェックを省略すると，別の命令のデータが読み出されることになるが，プログラムの実行の正しさが失われるわけではない．ヒット率が十分に高ければ，そのようなことが起る確率は十分に低く，それによる性能の低下は許容できる可能性が高い．

5.2 スラック表の連想度

4 章で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，別個にスケジューリングされる．

本稿では，分離されたそれぞれに対してスラックを予測した．そのため，スラック表に登録するときには，通常通り元の命令のアドレスの下位をとって生成したカラム・アドレスをアドレス計算命令に割り当て，そのカラム・アドレスの最上位ビットを反転したものをメモリ・アクセス命令に割り当てた．

このことによって，使用されるカラムに偏りが生じ，スラック表のヒット率が悪化したため，連想度を 4 にすることで対処した．

このことは，カラム・アドレスの生成方法を工夫することで回避できると思われる．

5.3 スラックが 0 の命令

4 章で述べたように，スラックが 0 の命令は，全体の半分程度にもなる．そこで，スラックが 0 の命令を登録しないことによって，スラック表の少量化を図ることが考えられる．スラック表がミスした場合には，スラックは 0 と予測すればよい．

4 章で述べたように，スラックが 0 の命令は，全体の半分程度にもなるから，スラック表の容量を半減できる可能性がある．

5.4 メモリ定義表の少量化

4 章の評価に用いたメモリ定義表は，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，1次キャッシュの 1ワードにつき，書き込みを行ったストア命令のアドレスと定義時刻を記録することになる．そのため，このメモリ定義表の容量は，1次キャッシュのそれより大きく，あまり現実的とは言えない．

ただし，このような大きな容量としたのは，主にメモリ定義表のミスによる影響を評価結果から除外するためであり，実際にこれほどの大容量が必要な訳ではない．以下のように，少量化が可能である．

ストア命令は通常，(1) スタックに対するものなど，数サイクル程度の比較的小さなスラックを持つもの と，(2) 数十サイクル程度以上の非常に大きなスラックを持つもの に二分される傾向がある．

(1) に対しては，ごく最近のストアに関してのみ記録する小容量のテーブルがあればよい．そして，そのテーブルから溢れたストアに関しては，(2) と判断して，大きなスラックを予測すればよい．

特に，エントリ数がロード / ストア命令ウィンドウのエントリ数程度でよいなら，ロード / ストア命令ウィンドウにフィールドを付加することによって実現することができる．

今後は，メモリに関するスラックの分布を調べることで，メモリ定義表の最適な容量を探る必要がある．

5.5 他の予測器との機能の重複

スラック予測器は，分岐予測ヒット / ミスやキャッシュ・ヒット / ミスなどの結果を反映したスラックを出力する．そのため，それらの予測器と一部機能が重複することが考えられる．機能の重複に関しては，以下のようにまとめられる．

分 岐 予 測

スラック予測器の出力には，分岐予測ヒット / ミスの結果が繰り込まれる．当然のことではあるが，分岐予測器は，分岐の方向を予測するのであって，分岐予測のヒット / ミスを予測するのではない．したがって，スラック予測器と分岐予測器の間には，機能の重複はない．

両パス実行を行うプロセッサは，分岐予測ヒット率の低い分岐に対してのみ両パス実行を行うため，分岐予測ヒット / ミスを予測するテーブルを持つ．このテーブルが，スラック予測器に近い働きをする．

キャッシュ・ヒット / ミス予測

スラック予測器の出力はキャッシュ・ヒット / ミスを反映したものとなる．ただし，通常のキャッシュ・ヒット / ミス予測器は，ヒット / ミスを予測するだけで，スラックを予測するのではない．

したがって，スラック予測器の出力をもってキャッシュ・ヒット / ミス予測を行うことが考えられる．

5.6 大きなスラック

4.3 節の評価では，予測スラックが l 以上であれば，その命令の実行レイテンシを 1 サイクル増加させた．スラックが s と予測された命令の実行レイテンシを，1 ではなく，s サイクル増加させることも原理的には可能であるが，以下の理由により，通常のプロセッサでは現実的ではない；スラック予測器の予測精度は十分に高く，数十以上のスラックであってもかなり正しく予測できる．しかし，通常のプロセッサで実行レイテンシを数十サイクルも増加させると，未終了の命令によって命令ウィンドウ・エントリなどの計算資源が占有されるため，それによる IPC の悪化が許容できなくなる．

数十以上の大きなスラックに関しては，別の利用法を試す必要がある．その例としては，2 章で触れたように，残っている命令のスラックがすべて大きい値であった場合に DVS 制御をかけることなどが考えられる．

5.7 厳密なスラック

2.2 節で述べたように，本稿で用いたスラックは近似であり，スラックの厳密な定義には従っていない．図 1 の例では，命令 I_y の厳密なスラックは 1 であるが，本稿で述べた手法では 0 と計算される．

命令 I_d の近似スラックを命令 I_y に伝搬させるなどすれば，より厳密なスラックを漸近的に求められる可能性がないではない．しかし厳密なスラックは，その利用法自体 自明ではない．求め方とその利用法を合わせて考慮する必要がある．

6. おわりに

本稿では，スラック予測によるクリティカリティ予測について述べた．評価結果は，スラック予測器の予測精度は非常に高いことを示している．

ただし本稿では，スラックが高精度に予測できることを示しただけで，実際に役に立つのかは厳密には分からない．今回示したスラック予測器を，命令スケジューリングや省電力アーキテクチャなどの技術に応用する場合には，応用に応じた工夫が必要になるだろう．

しかし，本稿の評価は予備的なものとしては十分によい結果を示しており，今後様々な技術への応用が期待できる．

参 考 文 献

1) Keller, J.: The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, *9th Annual Microprocessor Forum* (1996).
2) 小林良太郎, 安藤秀樹, 島田俊夫: 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, 並列処理シンポジウム JSPP 2001, pp. 31–38 (2001).
3) 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).
4) Fields, B., Bodik, R. and Hill, M. D.: Slack: Maximizing Performance under Technological Constraints, *29th. Int'l Symp. on Computer Architecture (ISCA-29)* (2002).
5) 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).
6) Grunwald, D.: Micro-architecture Design and Control Speculation for Energy Reduction, *Power Aware Computing*, Kluwer, ISBN 0-306-46786-0, chapter 4 (2002).
7) 千代延昭宏, 佐藤寿倫: 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, 情報処理学会研究報告 2003–ARC–154 (SWoPP 2003), pp. 1–6 (2003).
8) 近藤正章, 藤田元信, 中村宏: 演算部とデータ供給部の動的周波数変更による低消費電力化手法の検討, 情報処理学会研究報告 2003–ARC–154 (SWoPP 2003), pp. 97–102 (2003).
9) Casmira, J. and Grunwald, D.: Dynamic Instruction Scheduling Slack, *Kool Chips Workshop (in conjunction with MICRO-33)* (2000).
10) 千代延昭宏, 佐藤寿倫, 有田五次郎: 低消費電力プロセッサアーキテクチャ向けクリティカルパス予測器の提案, 情報処理学会研究報告 2002–ARC–149 (SWoPP 2002), pp. 1–6 (2002).
11) Yeager, K. C.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, Vol. 16, No. 2, pp. 28–40 (1996).