特別研究報告書

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

指導教員　　富田 眞治 教授

京都大学工学部情報学科

福山 智久

平成 17 年 2 月 10 日

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

福山 智久

内容梗概

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．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．スラックは原則的には，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．

スラック予測器は主にスラック表と定義表からなり，さらに定義表はレジスタ定義表とメモリ定義表とからなる．スラック表は命令の過去のスラックを記憶する予測表本体である．一方，定義表には各データに対してそのデータを定義された時刻と定義した命令を記憶し，スラック表に記憶するスラック自体の計算に用いられる．スラックの予測は命令をフェッチする際にスラック表に記憶されたその命令のスラックの値を読み出すことで行われる．

本稿では，スラック予測を省電力アーキテクチャのための命令スケジューリングに応用する方法について述べた．整数演算器で実行される命令のうちスラックの値が 1 以上である命令を低速／低消費電力の演算器で実行することにより，演算器の性能を大きく低下させることなく省電力化を図る．本稿では，命令スケジューリングに応用する場合に用いるスラックの計算方法を原則的な方法と合わせて 3 つ提案している．第 1 の方法は，定義時刻と使用時刻の差によるもの．この方法では，発振が生じ効率よく命令を遅らせることができず不十分である．そこで第 2 の方法では，実効定義時刻を用い効率よく命令を遅らせることができるようにした．ところが今度は逆に，見かけ上クリティカルになっている命令の存在により命令を遅らせ過ぎ性能が大きく悪化してしまった．そして第 3 の方法では，見かけ上クリティカルになっている命令が存在する状況を発見しスラックの厳密な値を求めることとした．第 2，第 3 の方法は定義表にフラグを用意することで可能である．

本稿の命令スケジューリングのようにスラックの値が 1 以上の命令を遅らせる場

合，スラックの値が 0 であるかどうかが重要となる．そこで，スラック0 予測の精度を向上させるために飽和型 2ビット・カウンタについても述べる．

評価では，高速な演算器を 3 つ，低速な演算器を 3 つ，合計 6 つの演算器を用い本稿で述べた命令スケジューリングを実装しシミュレーションを行い，全ての演算器を高速なものにした場合や低速なものにした場合と比べた．全ての演算器を高速なものにした場合と比べたときの評価結果を次に示す．

- 第 1 の方法では，4.5%の IPC 低下で 14.5%の EDP を削減できた．また，低速な演算器で実行した命令の割合は整数演算器で実行した全命令の約 31%である．
- 第 2 の方法では，14%の IPC 低下で 37%の EDP を削減できた．また，低速な演算器で実行した命令の割合は 78%であった．
- 第 3 の方法では，4.5%の IPC 低下で第 1 の方法とほど同じであるが，EDP の削減はは 4.5%多い 19%であった．また，低速な演算器で実行した命令の割合は全体の 39%で全ての演算器を高速なものにした場合のスラック1 以上の命令の割合とほぼ一致している．

これらの結果は，第 3 の方法はより正確にスラックの計算ができていることを示している．にもかかわらず，性能が 4.5%低下している理由として次の 2 つが挙げられる．

- スラック予測自体のミス
- 高速な演算器を 6 つから 3 つに減らしたためにクリティカルな命令を 1サイクルにつき最大 3 命令までしか実行できなくなってしまったため．

評価結果は，性能を大きく低下させることなく省電力化を達成しており，スラック予測が省電力アーキテクチャに十分応用できることを示している．

Instruction Scheduling for Low-Power Architecture with Slack Prediction

Tomohisa Fukuyama

Abstract

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 as the slack of the instruction. Thus the slack of a critical instruction is zero cycles. The slack value is stored to the prediction table to be a predicted value for the next time.

A slack predictor is composed of a slack table and a definition table, and the definition table is composed of a register definition table and a memory definition table. The slack table is a prediction table itself, and stores slack values. On the other hand, the definition table is used for calculating the slack values.

This paper describes instruction scheduling for low-power architecture with slack. We propose that a microprocessor has some fast and power-hungry ALUs and some slow and power-efficient ALUs, and the instructions whose slack values are zero are executed in the fast ALUs and the instructions whose slack values are more than zero are executed in the slow ALUs. Using this scheduling strategy , we can reduce microprocessor energy consumption while maintaining its performance.

We propose three ways to calculate the slack value for instruction scheduling. In the first way , we use the basic slack calculation, that is, we calculate the slack value by subtracting a definition time from a use time. In the second way, we use a effective definition time. In the third way, the most accurate calculation is done by detecting apparently critical instructions. Adding a flag to the definition table enable these three calculations.

In the simulation, our microprocessor has three fast ALUs and three slow ALUs. We evaluate our instruction scheduling by comparing IPC and EDP with those of the case that all ALUs are fast ones.

Evaluation result shows :

- If the first way is used, EDP is reduced 14.5% at the cost of 4.5% IPC degradation, and 31% instructions are executed in slow ALUs.

- If the second way is used, EDP is reduced 37% at the cost of 14% IPC degradation, and 78% instructions are executed in slow ALUs.
- If the third way is used, EDP is reduced 19% at the cost of 4.5% IPC degradation, and 39% instructions are executed in slow ALUs. The rate of instructions executed in the slow ALUs is almost equal to that of instructions whose slack is more than one in the case that all ALUs are the fast ones.

These results show that with the third way we can calculate the slack value accurately. But although third way is most accurate calculation, IPC is decreased by 4.5%. This is caused by next two matters :

- The slack prediction makes a mistake.
- A number of critical instructions executed per cycle must not exceed three, because the microprocessor have only three fast ALUs.

The evaluation result shows that microprocessor energy consumption is reduced while maintaining its performance and the slack prediction can be applied to instruction scheduling for low-power architecture.

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

目次

第 1 章　はじめに **1**

第 2 章　クリティカル・パスとスラック **3**

2.1　クリティカル・パスに基づく方法 3

2.2　スラック予測に基づく方法 4

2.3　近似スラック 5

2.4　スラックの基本計算 7

第 3 章　スラック予測器 **7**

3.1　スラック予測器の構成 7

3.2　スラック予測器の動作 8

3.3　条件分岐命令 10

3.4　スラック表，メモリ定義表 のミス 10

第 4 章　スラックの計算方法 **11**

4.1　スラックの基本計算 11

4.2　見かけのスラック 13

4.3　見かけのスラック（その 2） 13

4.4　問題の発見 14

第 5 章　予測精度の向上 **15**

5.1　グローバル分岐履歴の導入 15

5.2　飽和型 2 ビット・カウンタの導入 16

第 6 章　評価 **16**

6.1　評価方法 16

6.2　評価結果 20

6.3　関連研究 23

第 7 章　おわりに **23**

謝辞 **24**

参考文献 **24**

第1章　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, 4, 5, 6, 7].

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．

（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)[8] 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 [9, 10]．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，スラック予測器による予測結果を，実際に省電力アーキテクチャのための命令スケジューリングに応用する方法について述べる．

省電力アーキテクチャへの応用　スラックが 1 以上の命令を低速/ 低消費電力の演算器で実行することによりプロセッサの性能を大きく低下させることなく省

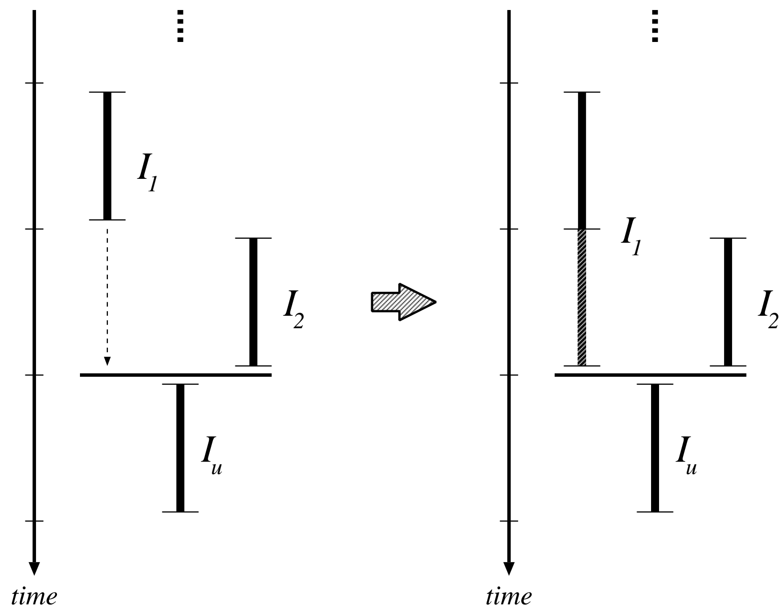

図 1: I_1 を 1サイクル遅らせた場合

電力化を図る．

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．

スラックが 1 以上の命令は，プログラムの実行時間を増大させることなく，その実行を遅らせることができる．例えば，図 1 左においてスラックが 1 と計算された命令 I_1 を，低速な演算器で実行することにより 1 サイクル遅らせた様子を図 1 右に示す.「1 サイクル遅らせる」とは，**その実行結果が使用可能になる時刻を**1 サイクル遅らせることを意味する．

図 1 右では，I_1 と I_2 が前回と同様のタイミングで実行された場合を示している．I_1 の実行結果が得られる時刻が 1サイクル遅れても，I_u の実行開始は遅れておらず，プログラムの実行時間は増大しないことに注意されたい．

上述したように，本来スラック予測とはクリティカリティを多値的に表現しているものである．しかし，本稿の命令スケジューリングでは，スラックが 0 であるか 1 以上であるかが重要となる．すなわち，ここで言うスラック予測とはスラック 0 予測と言えよう．

以下，第 2 章でクリティカル・パスとスラックについて，第 3 章でスラック予測器について，第 4 章でスラック予測を用いて省電力アーキテクチャ向け命令スケ

ジューリングを実現する際のスラックの計算方法について，第5章でグローバル分岐履歴と2ビット・カウンタの導入について述べ，第6章で本稿で述べた命令スケジューリングの評価結果を示す．

第2章　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，2.3節で近似スラックについて，2.4節でスラックの基本的な計算について述べる．

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 [11]．

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．

（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 2 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．また同図では，

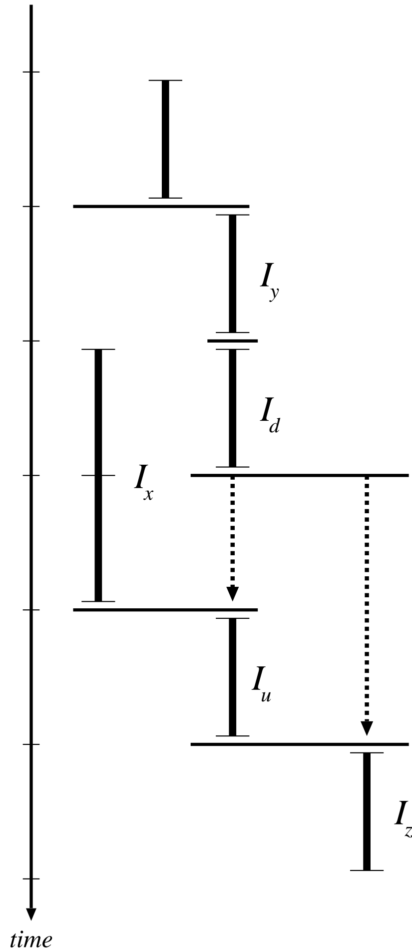

図 2: 命令実行のタイム・チャート

命令 I_z も I_d が定義した結果を参照しているが，I_u が先に I_d を参照しているので I_d のスラックは 2 とはならないことに注意されたい．

同図は，データ・フロー・グラフを示しているのではない．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[1)]．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．

2.4　スラックの基本計算

図 2 では，命令 I_d が定義した結果を最初に使用する命令は I_u となっている．この場合の I_d のスラック s は，原則的には，命令 I_d がデータを定義する時刻 t_d と，命令 I_u がデータを使用する時刻 t_u の差，つまり：

$$s = t_u - t_d - 1 \tag{1}$$

によって得られる．この式に従えば，図 2 の I_d のスラックは 1，I_u，I_x，I_y のスラックは 0 となる．

第 3 章　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，3.3 節で条件分岐命令の扱いについて述べ，最後に 3.4 節でスラック表，メモリ定義表のミスについて説明する．

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：

スラック表　命令のスラックを記録する表.

定義表　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. **フラグ** ：命令を遅らせて実行したかどうか

フラグについては，4.4 節で詳しく述べる．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

[1)] 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．

当する．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　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，定義命令，フラグ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とフラグの値を読み出せば，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，定義命令，フラグ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．

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：

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，定義命令，フラグを格納するフィールドを付加したものと考えてよい．

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，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 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 3 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 t_{d^1} および 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：

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：

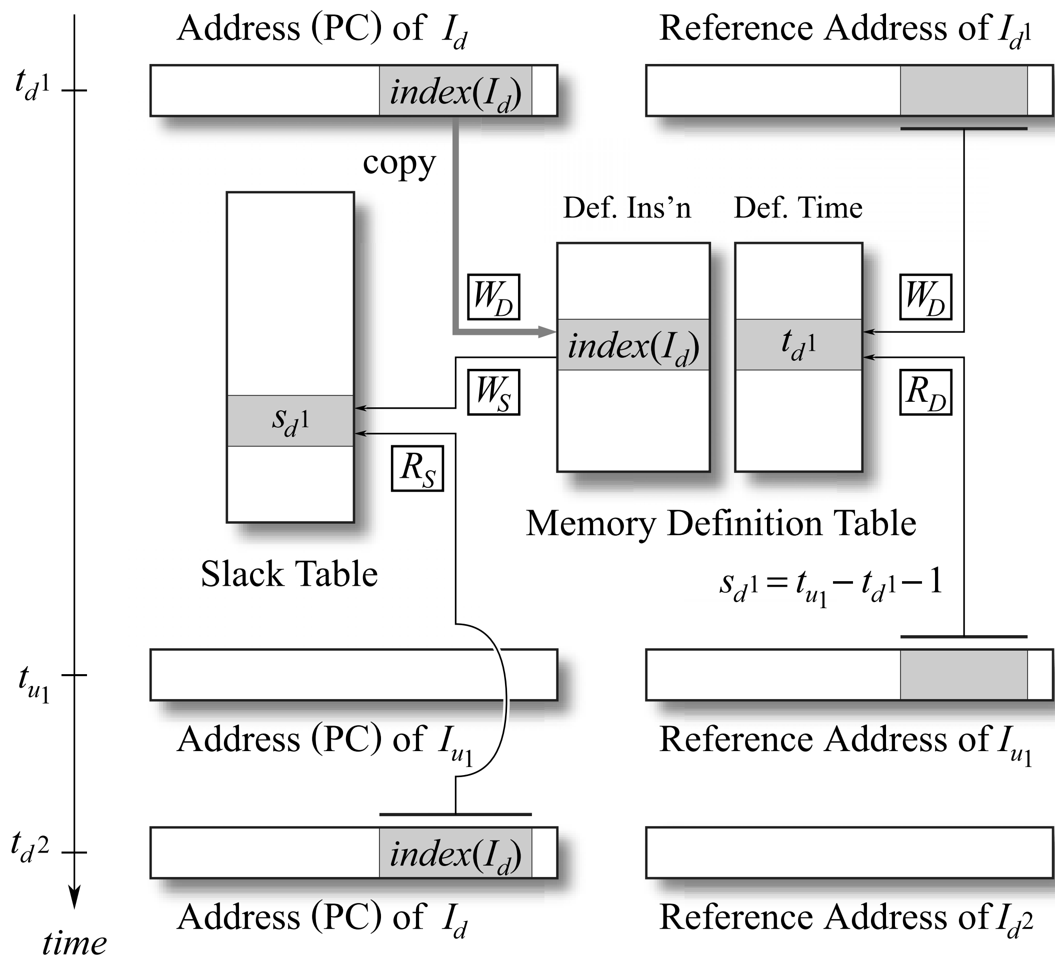

図 3: 予測器に対する操作（ストア命令の場合）

1. 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）を定義表に書き込む．
2. 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）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．

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．

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．したがって，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，以降同じエントリが繰り返し読み出されないようにする．このことは，メモリ定義表のエントリの有効利用にも効果がある．

3.3　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; フェッチ後，未完了のまま (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，MIPS R10000 プロセッサ [12] の場合，たかだか 4 個の条件分岐命令しか未完了にできない．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，その中では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．

3.4　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　定義表のエントリがリプレースされたのだから，定義側の命令を特定することができず，スラック表への登録を行うことはできない．

スラック表　スラック表ミス時には，スラックは 0 と予測するのが安全である．また，第 6 章節で述べるように，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．

第 4 章　スラックの計算方法

本章では，スラック予測を命令スケジューリングに応用する際に用いる 3 種類のスラックの計算方法を提案する．図 4 はそれぞれの計算方法を用いたときの状態の遷移を表わしている．

以下，4.1 節で基本的な計算方法，4.2 節で見かけのスラック，4.3 節で見かけ上クリティカルになっている命令について述べ，最後に，4.4 節で 3 つの計算がスラック予測器の定義表にフラグを用意することによって可能であることを示した．

4.1　スラックの基本計算

第 2 章で述べたように，スラック s の基本となる計算式は，定義命令 I_d がデータを定義した時刻 t_d とそのデータを最初に参照する命令 I_u がデータを使用する使用時刻 t_u の差，つまり式 (1)：

$$s = t_u - t_d - 1 \tag{1}$$

によって与えられる．

この式に従えば，図 4：左上の命令 I_1，I_2 のスラックはそれぞれ 1，0 と計算される．よって，次回実行時には図 4：左下に示すように，I_1 のみを 1 サイクル遅らせて実行する．

ただし，図 4：左下において，1 サイクル遅らせた命令 I_1 については，そのスラックを再計算する際注意が必要である．式（1）にしたがって計算すると，遅らせた命令 I_1 のスラックは 0 となる．すると次回 I_1 をスケジューリングする際には，I_1 はクリティカルであると判断され，実行時には 1 サイクル遅らせることなく

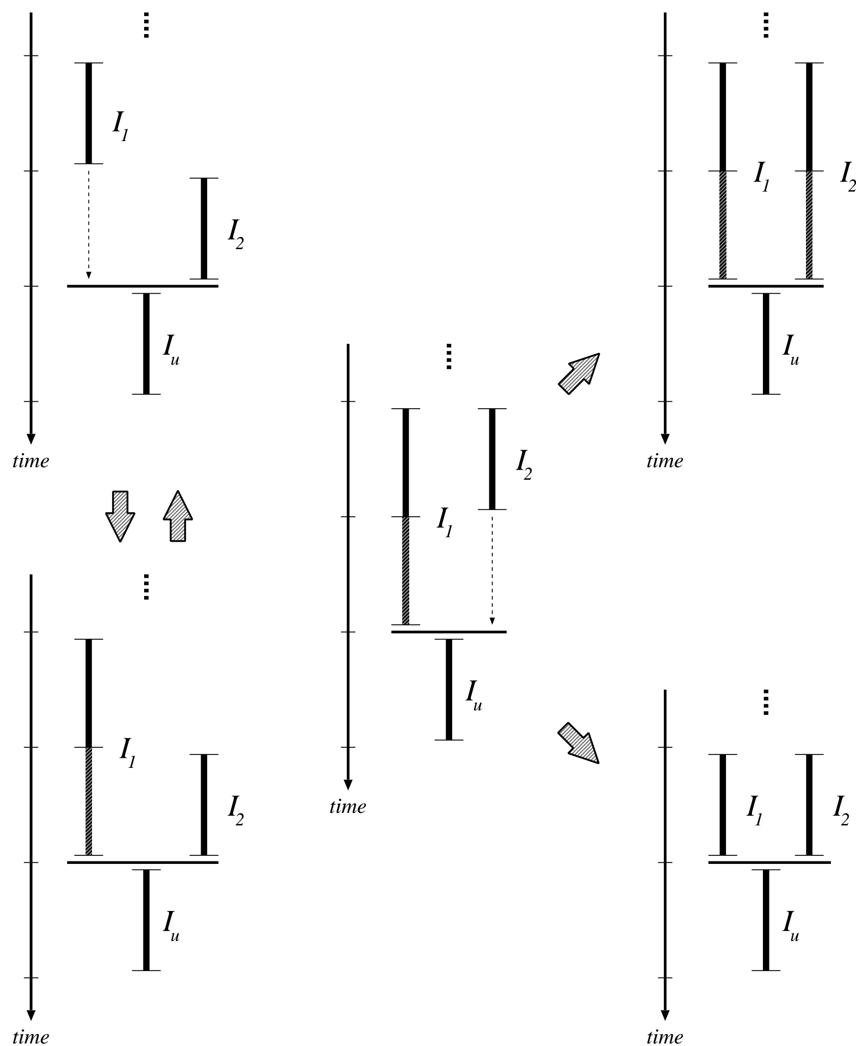

図 4: 状態の遷移

実行されてしまう．その結果，次回には再び図 4：左上のように実行され，以降，上下の状態を繰り返すことになる．

しかし，この繰り返しが発生するのは，遅らせた命令のスラックが式（1）によって 0 と計算される時のみ，つまり，遅らせなかった場合のスラックが 1 の命令のみである．よって，遅らせずに実行した場合にスラックが常に 1 となる命令は，毎回遅らせることができるにもかかわらず，2 回に 1 回しか遅らせることができなくなってしまう．

4.2　見かけのスラック

遅らせた命令 I_1 の 0 というスラックは，故意に 1サイクル遅らせたことによって生ずる「見かけの」のスラックであるといえる．I_1 の真のスラックは，図 4：左上の場合と同様，遅らせなかった場合の定義時刻をスラックを元に計算すればよい．この遅らせなかった場合の定義時刻を**実効定義時刻**と呼ぶことにする．つまり，遅らせなかった場合の定義時刻 t_e は：

$$t_e = t_u - 1 \tag{2}$$

によって与えられる．この場合のスラックは，使用時刻と実効定義時刻の差，つまり：

$$\begin{aligned} s &= t_u - t_e - 1 \\ &= t_u - t_d \end{aligned} \tag{3}$$

によって得られる．

このようにすれば，図 4：左の上下の状態を繰り返すことはなく，効率よく命令を遅らせることが可能となる．

4.3　見かけのスラック（その 2）

前節までの説明では，命令 I_1 と I_2 は，毎回同じタイミングで実行を開始されるとしていた．しかし，必ず前回と同じタイミングで実行開始されるとは限らない．図 4：中に示すように，I_1 と I_2 が同時に実行開始されることもあり得る．さらには，I_2 の方が早く実行される可能性もある．このような場合，前回スラックが 1 以上であったため 1サイクル遅らせた命令 I_1 が，今回クリティカルになってしまう．このような命令を**見かけ上クリティカルになっている命令**と呼ぶことにする．

図 4：中のような場合，I_1 のスラックは実効定義時刻に基づいた式（3）により，I_2 のスラックは基本となる式（1）により，それぞれ 1 と計算され，次回実行時には図 4：右上のようになる．しかし実際には，図 4：右下のように I_1，I_2 のスラックを共に 0 とし実行した方が I_u を 1サイクル早く実行することができる．

スラックを共に 1 と計算した図 4：右上の場合は，すべての命令を無条件に 1サイクル遅らせたのと変わりがない．また，一旦この状態になるとスラックが 1 のまま 0 に戻らないという可能性もある．

結局，図4：中のI_1、I_2のように見かけ上クリティカルになっている命令がある場合には，次回は図4：右下のようにするために，I_1、I_2のスラックを共に0とする．

図4：左下のように，遅らせていない命令（図ではI_2）がクリティカルになっている状況では，I_1は見かけ上クリティカルになっている命令とは言わない．逆に言えば，図4：中や右上のように，クリティカルになっている命令のすべて（図では，I_1とI_2）が遅らせた命令である場合に，これらの命令を見かけ上クリティカルになっている命令と言う．

見かけ上クリティカルになっている命令がある場合，命令を遅らせたサイクル数（図では1サイクル）だけ無駄に実行が遅れている．したがって各命令の実効スラックは，式（1），(3）により得られたスラックの値から遅らせたサイクル数を減ずればよい．

つまり，見かけ上クリティカルになっている命令が生じている場合，遅らせた命令のスラックは式（3）から1減じた式，つまり式（1）と同じ式により得られ，遅らせなかった命令は式（1）から1減じた式：

$$s = t_u - t_d - 2 \tag{4}$$

によって得られる．

4.4　問題の発見

２つの見かけのスラックの問題を発見するには，その命令が遅らせたものかどうかを表すフラグが必要になる．そこで，スラック予測器の定義表にフラグを用意し，命令を遅らせていない場合には0を，遅らせた場合には1を命令の実行時に記憶する．

まず、式（1）により暫定的なスラックを求める．この暫定的なスラックの値が0となる命令のフラグが全て1であるとき，見かけ上クリティカルになっている命令がある状況が生じていることとなる．

このように見かけ上クリティカルになっている命令がある状況はフラグを容易することで発見でき，この場合の実効スラックはフラグが0である命令に関しては暫定的なスラックから1減じた値，つまり式（4）で，フラグが1である命令に関しては暫定的なスラックの値，つまり式（1）で得ることができる．

見かけ上クリティカルになっている命令がない場合には、フラグが0の命令は暫

定的なスラックの値，つまり式（1）で得られる値が、フラグ１の命令は暫定的なスラックに 1 足した、つまり式（3）で得られる値がそれぞれ実効スラックとなる．

このようにして得られた実効スラックの値をスラック表に記憶する．

本章では，スラックの計算について 3 つの方法を提案した．

第 1 に 4.1 節で述べた基本となる計算方法である．しかし，この方法では常にスラックが 1 である命令に関して発振が生じ（図 4：左），遅らせることができる命令であるにもかかわらず遅らせずに実行してしまうという可能性があった．

発振が生じるのを防ぐために，第 2 の方法として 4.2 節で実効定義時間を用いて効率よく命令を遅らせる計算方法を提案した．だが，今度は逆に見かけ上クリティカルになっている命令が生じることにより（図 4:中），命令を遅らせ過ぎる状況が考えられた.（図 4：右上）

そこで，第 3 の方法として 4.3 節で見かけ上クリティカルになっている命令が生じた場合におけるより厳密なスラックの計算方法を提案し（図 4：右下），この計算で求められるスラックを実効スラックとした．

第 5 章　予測精度の向上

本章では，スラック予測の精度の向上を試みるために，スラック予測器にグローバル分岐履歴と飽和型 2ビット・カウンタを導入する方法について述べる．

5.1　グローバル分岐履歴の導入

スラック予測器にグローバル分岐履歴を導入するには，gshare 分岐予測器などと同様にする方法がまず考えられる；すなわち，スラック表のインデクスとして，命令のアドレスとグローバル分岐履歴の排他的論理和を用いるのである．ただし以下で述べるように，スラック表はタグを持っていることに注意する必要がある．

分岐予測で用いられる PHT (Pattern History Table) には，競合が発生する；すなわち，通常 PHT はタグを保持しておらず，異なる 2 つの命令が同一の PHT エントリを使用することを許している．

一方，値予測で用いられる VHT (Value History Table) では，タグ比較を行い，競合を許さないことが普通である[1]．

---

[1] タグは，一部省略できる可能性がある [13].

スラック表も，VHT と同様，現在ではタグを保持している．グローバル分岐履歴を導入するにあたっては，インデクスに命令のアドレスとグローバル分岐履歴との排他的論理和を用いるとともに，タグにもグローバル分岐履歴を加え，競合を完全に排除している．そのため，破壊的競合ではなく，ヒット率の低下によって予測精度が低下するおそれがある．

5.2 飽和型 2 ビット・カウンタの導入

これまで述べてきたスラック予測の方法では，スラックの最新の値 (last value) を記憶している．一方，第 1 章でも述べたようにスラックが 1 以上の命令を 1 サイクル遅らせる場合，スラックの厳密な値は必要なく，スラックが 0 かどうかを記憶しておけばよい．

結局，スラックが 0 かどうかを予測する上では，これまで述べてきた方法は，いわば 1 ビット・カウンタによる予測であると言える．

そこで，飽和型 2 ビット・カウンタを用いることにより，命令のローカルな履歴を反映させることが考えられる．スラックの計算による値が 0 であればカウンタを 1 減じ，1 以上であれば 1 足す．そして ALU を選択する際には，その命令のスラックの値が 0，1 であれば高速な ALU で，2，3 であれば低速な ALU で実行する．

第 6 章 評価

シミュレーションにより，第 4 章で提案した 3 種類のスラックの計算方法を実際に命令スケジューリングに用いた場合について評価する．

なお，シミュレーションではスラック予測器で用いたグローバル分岐履歴の履歴長を全て 2 ビットとした．スラック予測器にグローバル分岐履歴を用いることで予測精度が向上することが認められている [10] が，その度合いはわずかで今回の評価にはほとんど影響が現れなかったためである．

6.1 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本稿で述べた命令スケジューリングによる省電力アーキテクチャの効果を測定した．測定には表 1 に示す SPEC

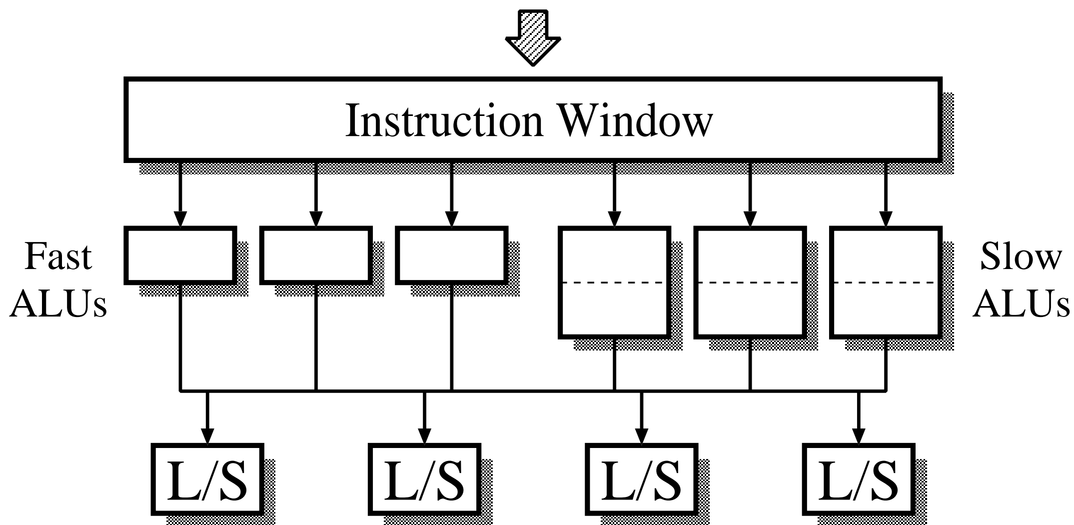

図 5: プロセッサのモデル

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　評価では，本稿で述べた命令スケジューリング方式を整数演算器に適用する．用いたプロセッサの整数系のブロック図を図 5 に示す．

同図に示すように，プロセッサは高速な ALU と低速な ALU をそれぞれ 3 個ずつ，合計 6 個持つ．高速な ALU と低速な ALU のレイテンシは，それぞれ，1 サイクル，および，2 サイクルである．低速な ALU は，高速な ALU をパイプライン

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	32M
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

動作させるものに相当し，レイテンシは 2 に増加するが，スループットは 1 のままである．これらの ALU の電源電圧は，それぞれ，1.1 V，および，0.7 V とした [?]．スラックが 0 と予測された命令を高速な ALU，1 以上と予測された命令を低速な ALU にスケジューリングする．

実際にキャッシュへのアクセスを行うロード / ストア・ユニットは，4 個あり，同図のように接続されている．すなわち，ロード / ストア命令のアドレス計算は，これらの ALU のいずれかで実行される．アドレス計算を実行した ALU とロード / ストア・ユニットの間は，完全結合とした．すなわち，アドレス計算を行った ALU とロード / ストア・ユニットの組み合わせには制限はない．

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．

プロセッサのそれ以外のパラメタを表 2 に示す．また，分岐予測には Gshare 予測器を用いた．エントリ数は 4K で履歴長は 12 とした．

テーブルのパラメータ　表 3 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

表 2: プロセッサの各パラメータ

パラメタ	サイズ
FETCH 幅	16
命令ウィンドウ	16
ISSUE 幅	8
COMMIT 幅	64
レイテンシ（Latency/Throughput）	iALU 1/1，iMULT 3/1，iDIV 20/19，fADD 2/1，fCMP 2/1，fCVT 2/1，fMULT 4/1，fDIV 12/12，fSQRT 24/24

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．

スラックの計算方法　それぞれの命令をどちらの ALU で実行するかを決定する際に用いるスラックの計算方法は第 4 章で提案した次の３つとする．

BASE　4.1 節で述べた基本となる計算方法（BASE）．発振し遅らせることができる命令でも遅らせずに実行されてしまう．

EDT　4.2 節で述べた実効定義時刻（EDT：Effective Definition Time）を用いる計算方法．BASE とは逆に命令を遅らせ過ぎる状況が発生する．

ES　4.3 節で述べた，見かけ上クリティカルになっている命令を発見することにより遅らせ過ぎる状況をなくすことにより，厳密な計算を行い実効スラック（ES：Effective Slack）を求める計算方法．

上記の３つのそれぞれの場合について，前回の履歴にのみ依存している場合と飽和型 2 ビット･カウンタを用いた場合の 2 通りをシミュレーションし，それぞれ **-1b**，**-2b** として区別する．

それぞれの場合について，性能，ALU における消費電力，低速の ALU で実行した命令の割合を測定し，全ての ALU を高速にした場合（FAST とする）や低速にした場合（SLOW とする）と比較することによって評価を行う．

表 3: 各表，キャッシュ，メモリのパラメータ

		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．

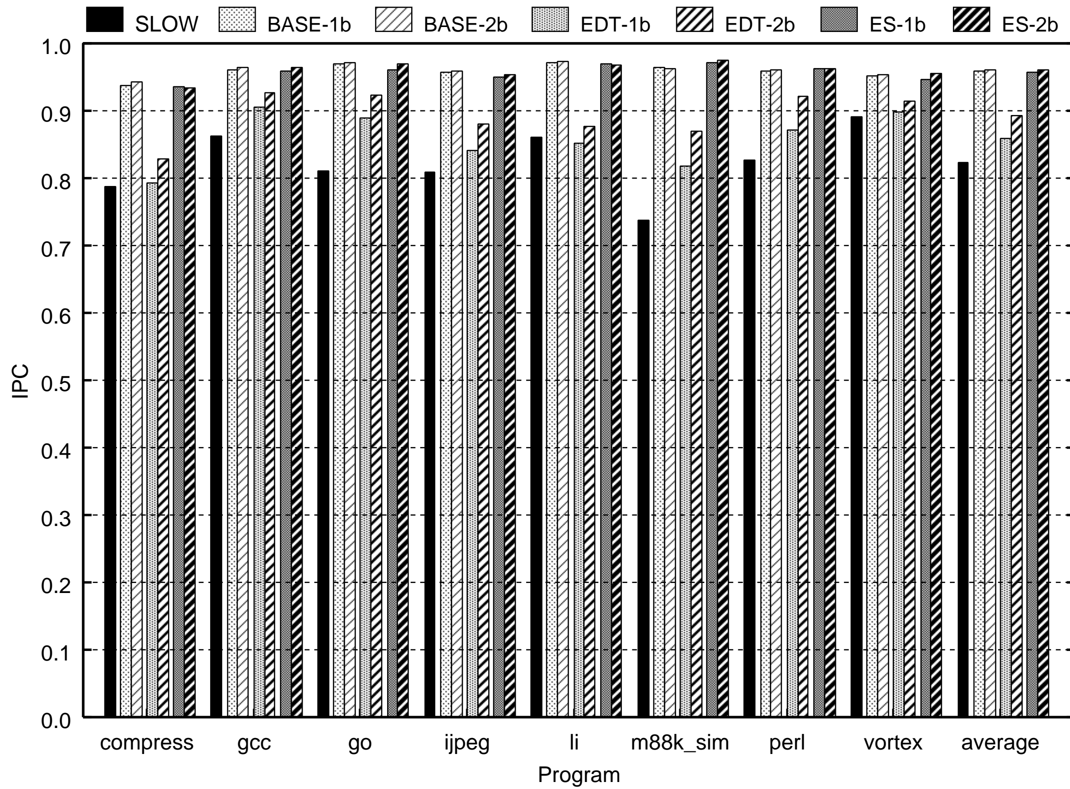

図 6: 性能の比

6.2 評価結果

図 6，図 7，図 8 に結果を示す．図 6 は性能を，図 7 は省電力アーキテクチャの効果を示す電力遅延積（EDP：Energy Delay Product）を，図 8 は低速な ALU で実行した命令の割合を示している．

図 6，図 7 には，9 組のバーがあり，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 1 組はそれらの平均である．図 6 は，FAST に対する IPC の比を，図 7 は EDP の比を表わしている．各組には，バーが 7 本ずつあり，それぞれ左から前節で述べた SLOW，BASE-1b，BASE-2b，EDT-1b，EDT-2b，ES-1b，ES-2b の場合のものを指している．

一方，図 8 には 4 組のバーがあり，左から FAST，BASE-1b，EDT-1b，ES-1b の場合の 8 つのプログラムの平均である．それぞれの組の 1 番左の薄いバーは整数演算器で実行した全命令に対する通常の整数演算命令の比で，2 番目のバーはロードの，3 番目のバーはストアのアドレス計算の比で 4 番目のバーはそれらの合計である．FAST の組の濃いバーはそれぞれの命令における全命令に対するスラックが 1 以上の命令の比で，その他の組の濃いバーは低速な ALU で実行した

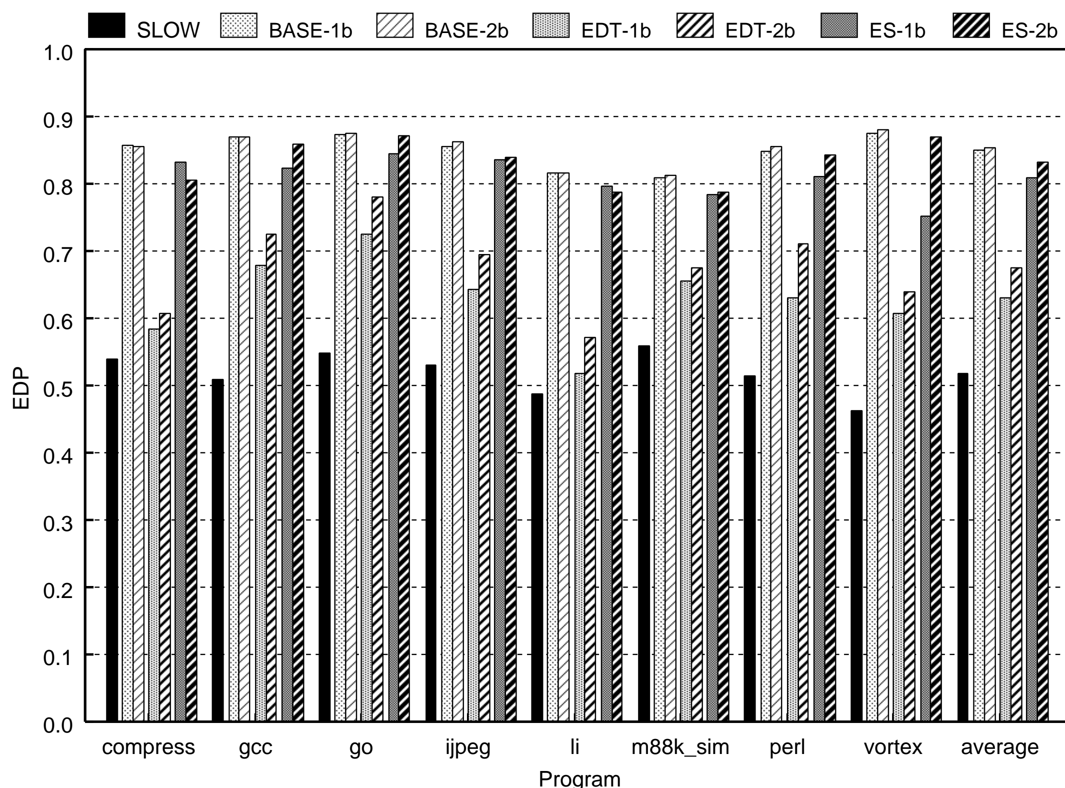

図 7: EDP の比

命令の比を表わしている．

図 6，図 7 を見ると，SLOW での IPC 低下は 18 %，EDP 削減は 48 %であるのに対し，

- BASE-1b では，FAST に比べ IPC，EDP がそれぞれ約 4.5 %，約 14.5 %低下した．
- EDT-1b では，FAST に比べ IPC が約 14 %低下しているものの，約 37 %もの EDP を削減できている．
- ES-1b は，IPC では BASE とほぼ同じ約 4.5 %であったが，EDP では約 19 %と BASE よりも 4.5 %多く削減できている．
- -2b では，それぞれの場合の飽和型 2 ビット・カウンタを用いなかった場合よりも IPC，EDP の比が共に若干減少した程度であった．

一方，図 8 を見ると，FAST では，整数演算命令の 44 %，ロード命令のアドレス計算部分の 31 %，ストア命令のアドレス計算部分の 42 %がスラック1 以上である．3 つを合わせると整数演算器を用いる全命令の 40 %がスラック1 以上であることがわかる．

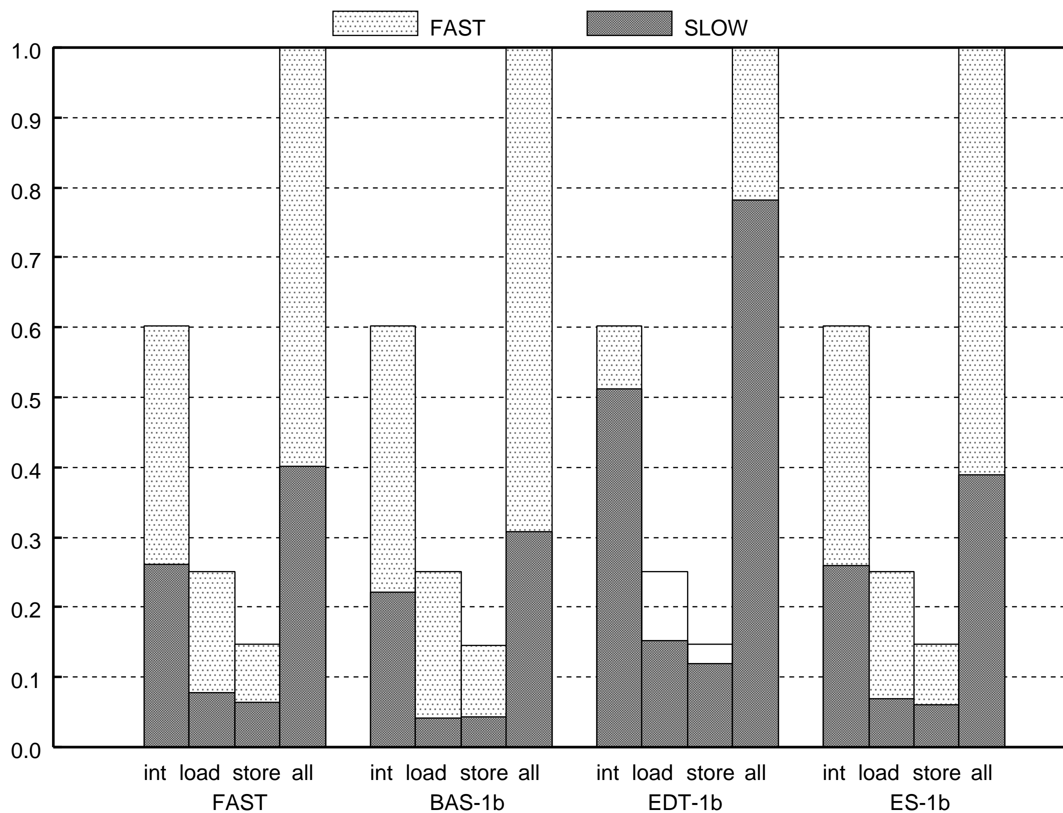

図 8: 低速な ALU で実行した命令の割合

これに対して，

- BASE-1b では低速な ALU で実行された命令は整数演算器で実行された全命令の 30 %で FAST においてスラック1 以上である命令よりも少ない．これは，発振が生じ遅らせることができる命令を遅らせられなかったためである．
- 逆に，EDT-1b では低速な ALU で実行された命令は 78 %と大幅に多い．これは，4.3 節で述べた見かけ上クリティカルになっている命令が存在し，命令を遅らせ過ぎているためである．
- ES-1b は 39 %の命令が低速な ALU で実行されており FAST のグラフとほぼ一致している．よって，正確にスラックの計算がされていると考えられる．

ES-1b は厳密にスラックを計算できているにもかかわらず，全ての ALU を高速なにしたときに比べ 4.5 %の性能が低下している．この原因として次の 2 つが挙げられる：

- 本当はクリティカルである命令を低速な ALU で実行した，つまりスラック予

測のミスによるもの．

- 高速な ALU を 6 つから 3 つに減らしたためにクリティカルな命令を 1サイクルにつき最大 3 命令までしか実行できない．それによって，使用する ALU に偏りが生じた場合には ALU が不足してしまう．

そこで，ALU の不足を解消するために高速な ALU を 10 個，低速な ALU を 10 個，合計 20 個の ALU を用いて，ES-1b と同様のシミュレーションを行った．すると，性能の低下は約 2 %まで減少した．これは，スラックの予測ミスによる性能低下と ALU 不足による性能低下が同程度影響していることを示している．

6.3　関連研究

千代延らは文献 7 ）において，小林らが提案しているデータフローグラフの最長パスを特定するためのパス情報テーブル [2]（PIT：Path info Table）をクリティカル・パス予測器として，省電力アーキテクチャの設計に用いている．クリティカル・パス予測器によりクリティカル・パス上にないと予測された命令を低速 / 低消費電力の ALU で実行し性能と消費電力の計測を行い，その結果を彼ら自身が提案しているクリティカル・パス予測器と比較することにより性能と省電力化の評価を行っている．PIT は約 5 %の性能低下で 20 %の EDP を削減できている．

使用したベンチマーク・プログラムやプロセッサのモデルが若干異なるものの本稿でのスラック予測を用いた命令スケジューリングは約 4.5 %の性能低下で 19 %の EDP を削減できており，PIT と同等の性能と省電力化を実現できている．

また，PIT は複雑なハードウェアを必要とし，実際に実装することはとても困難である．一方，これまでも述べた通りスラックは容易に求めることができることにも注意が必要である．

命令のクリティカリティを判定する際に，クリティカル・パスに基づいた PIT を用いた場合と，我々が提案する履歴に基づいたスラック予測を用いた場合とで，同様の結果が得られていることも興味深いところである．

第 7 章　おわりに

本稿では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について述べた．評価結果は，性能を大きく低下させることなく省電力化を達成しており, スラック予測が省電力アーキテクチャに十分応用できることを示して

いる．

また，2.3 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．第 2 章の図 2 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ることができ，本稿で述べた命令スケジューリングによりさらに多くの命令を遅らせることができる可能性がある．

なお，本稿の省電力アーキテクチャで述べた消費電力とは，演算器のみにおける消費電力でありプロセッサ全体の消費電力ではない．よって，スラック表や各定義表における消費電力は考慮していない．しかし，これまでも述べたようにスラック予測器は複雑なハードウェアを必要とせず，スラック予測器における消費電力は大きくないと考えられる．

謝辞

本研究の機会を与えていただいた富田眞治教授に深甚なる謝意を表します．また，本研究に関して適切なご指導を賜った森眞一郎助教授，中島康彦助教授，五島正裕助手，大阪工業大学の小西将人氏に心から感謝いたします．

さらに，日頃から御討論頂いた京都大学情報学研究科通信情報システム専攻富田研究室の諸兄に心より感謝いたします．

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] Casmira, J. and Grunwald, D.: Dynamic Instruction Scheduling Slack, *Kool Chips Workshop (in conjunction with MICRO-33)* (2000).

[9] 劉小路, 小西将人, 五島正裕, 中島康彦, 森眞一郎, 富田眞治: クリティカリティ予測のためのスラック予測, 先進的計算基盤システムシンポジウム SACSIS 2004, pp. 187–196 (2004).

[10] 福田匡則, 小西将人, 五島正裕, 中島康彦, 森眞一郎, 富田眞治: グローバル分岐履歴を用いたスラック予測器, 情報処理学会研究報告 2004–ARC–159 (SWoPP 2004), pp. 25–30 (2004).

[11] 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).

[12] Yeager, K. C.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, Vol. 16, No. 2, pp. 28–40 (1996).

[13] 佐藤寿倫, 有田五次郎: タグビット幅を考慮したデータ値予測機構のハードウェア量削減, 信学技報 CPSY 2000–3 (2000).

[14] M.Levy: SAMSUNG Twists ARM Past 1GHz, *Information Quarterly vol.1,no.1* (2002).