計算機アーキテクチャ 143－30
（２００１．７．２７）

スーパースケーラのための高速な命令スケジューリング方式の IPC の評価

西野賢悟† 小田累† 五島正裕†
中島康彦† 森眞一郎†
北村俊明† 富田眞治†

スーパースケーラは，命令スケジューリングのため，オペランドの有効性を追跡する *wakeup* と呼ぶロジックを持つ．我々は，従来のタグに基づく連想処理ではなく，命令間の依存関係を表す行列を読み出すことで *wakeup* を実現する方式と，その行列を縮小することにより遅延を IPC に対するペナルティに転化する手法を提案した．本稿では，シミュレーションによってそのペナルティを評価した．その結果，命令発行幅とウィンドウ・サイズを無限大にした場合でも行列のサイズは 16〜64 程度あればよく，提案方式によって，命令スケジューリングの遅延はウィンドウ・サイズとほぼ独立にできることが分かった．

Evaluation of IPC of a high-speed instruction scheduling scheme for superscalars

KENGO NISHINO,† RUI ODA,† MASAHIRO GOSHIMA,†
YASUHIKO NAKASHIMA,† SHIN-ICHIRO MORI,† TOSHIAKI KITAMURA†
and SHINJI TOMITA†

A superscalar has a logic called *wakeup* to manage availability of the data for instruction scheduling. We have proposed a new scheduling scheme which substitutes association of the tags by reading matrices which represents depndences between instructions, and a method which changes the delay of the matrices into IPC penalties. We evaluated the penalty by simulation. The result shows that the matrices size of 16 to 64 is enough even for the processor which have an infinity of the issue width and the window size, and that our scheme let the delay of instruction scheduling almost independent of the window size.

1. はじめに

現在のスーパースケーラでは，クロック速度が命令発行幅 (***IW***:Issue Width) とウィンドウ・サイズ (***WS***) を制限する主因となりつつある．スーパースケーラの構成要素のうち，***wakeup*** と呼ばれるロジックが，将来クロック速度を制限するものの 1 つになると予測されている．*Wakeup* は，命令発行ウィンドウの一部で，命令の発行に必要なソース・オペランドの有効性を追跡するロジックである．

従来の *wakeup* は，各命令が使用するデータに割り当てられた**タグ**による連想処理に基づくもので，RAM から読み出したタグで CAM をアクセスするという構造を持つ．

Wakeup を含む，スーパースケーラのほとんどすべての構成要素の遅延は，IW, WS の増加関数で与えられる．しかし，演算器自体の遅延は明らかに IW, WS とは独立であるから，その他の構成要素の遅延は IW, WS の増加に伴い相対的に長くなる．したがって，IW, WS の更に増加させるためには，より深い命令パイプラインを採用せざるを得ない．しかし，他の構成要素とは異なり，*wakeup* はパイプライン化不能である．*Wakeup* を構成する RAM と CAM は，パイプラインの深さに関わらず，0.5 サイクルの間に逐次的にアクセスされなければならない．その上，これらのメモリは，配線遅延の影響を強く受けるため，LSI の微細化の恩恵を受けにくい．以上の理由により *wakeup* は，IW, WS の増加，パイプラインの深化，LSI の微細化にともなっていっそうクリティカルになると予測されるのである．

このような背景から我々は，*wakeup* を高速化する全く新しい命令スケジューリング方式を提案した [1]．本方式では，タグによる連想処理ではなく，命令間の依存関係を表す行列に基づいてスケジューリングを行う．行列の更新はパイプラインのフロントエンドで済ませておくことができるため，*wakeup* は単にこの行列を読み出すだけで実現することができる．合わせて，*wakeup* の遅延を IPC に対するペナルティに変換する手法も提案した．本稿では，そのペナルティのより詳細な定量評価の結果を示す．

† 京都大学

2. 従来の動的命令スケジューリング方式

2.1 命令スケジューリングの原理

Out-of-order スーパースケーラは，論理的なレジスタとは別に，各命令の out-of-order な実行結果を一時的に保存するバッファ —— リオーダ・バッファ，あるいは，物理レジスタ・ファイル —— を用いる．Out-of-order 命令スケジューリングは，このバッファによる局所的なデータ駆動型の計算とみなすことができる；各命令にはバッファの 1 エントリが割り当てられ，実行結果はそのエントリに書き込まれる．一方結果を使用する命令は，プログラム・オーダとは独立に，ソース・オペランドに対応づけられたエントリに実行結果が書き込まれた時点で，実行を開始することができる．

命令の実行の開始時点を検出するため，左/右のソース・オペランドに対応づけられたエントリのそれぞれに対して，結果が書き込まれたかどうかを表すフラグ rdyL/R が，少なくとも論理的には，存在する．rdyL/R は，スケジューリングにおける中心的な役割を果たす．

命令スケジューリングの処理は，(1) ***rename***，(2) ***dispatch***，(3) ***wakeup***，(4) ***select***，(5) ***issue*** の 5 つのフェーズからなる．ある命令 I_c の rdyL/R に着目すると，スケジューリングの処理の流れは以下のように説明できる；なお，I_c の左ソース・オペランドは先行する命令 I_p の実行結果，右は即値であるとする：

(1) *Rename* 命令がフェッチされると，命令にはバッファのエントリが割り当てられる．同時に，左右のソース・オペランドに対応するエントリが求められる．この処理については，次節で詳しく述べる．

(2) *Dispatch* rdyL/R の初期化は，命令がウィンドウにディスパッチされるときに行われる．I_c の右オペランドは即値であるから，その rdyR は無条件に “1” に初期化される．一方，rdyL の初期値は，I_p が既に実行されているかどうかに依存する．既に実行され，その結果が利用可能である場合には，rdyL は “1” に初期化される．以降では，まだ実行されていなかった場合を考えよう．その場合，rdyL は “0” に初期化され，I_c は I_p が実行されるのを待って，ウィンドウ内で『眠る』ことになる．

(3) *Wakeup* やがて *select* によって I_p の発行が決定されると，その結果が利用可能になる時刻も決まる．すると I_c の rdyL は，適当なタイミングで “1” にセットされ，I_c は『起こされる』．本稿では，*wakeup* を，I_p の発行に伴う rdyL/R の更新処理と定義する．

(4) *Select* rdyL/R が共にセットされてる命令が，実行可能な命令である．*select* では，空いている実行ユニットに対して，発行可能な命令から実際に発行するものが選択される．

(5) *Issue* 選択された命令の情報がウィンドウを構成する RAM から読み出され，実行ユニットに送られる．

2.2 命令スケジューリングのパイプライン化

各フェーズのパイプライン化可能性は，フィードバックに依存する．フィードバックがない場合には自由にパイプライン化してよいが，ある場合には何らかの代償が必要となる．

命令スケジューリングには，以下の 2 つのフィードバックが存在する：

(1) 分岐 5 つのフェーズ全体は，分岐命令の実行から命令フェッチへのフィードバック・ループに含まれる．

(2) 発行 I_c に対する *wakeup* が開始できるのは，*select* が I_p の実行を決定してからである．すなわち，*select* の終わりから *wakeup* の先頭へは，分岐によるものより緊密なフィードバックが存在する．

命令スケジューリングの 5 つのフェーズのうち，発行のフィードバック・ループには含まれない，*rename*, *dispatch*, *issue* はパイプライン化可能である．パイプライン化の代償は，分岐予測ミス・ペナルティの増加であり，通常受け入れられる．実際 現存するスーパースケーラでは，*rename* の遅延のため，デコード・ステージに複数サイクルを充てることが普通である．

一方，*wakeup* と *select* は，パイプライン化すると IPC に深刻なダメージを与えかねない．*Wakeup* と *select* に合わせて 1 サイクル以上をかけると，先行する命令の結果を消費する命令は引き続くサイクルに発行できなくなる．このことは，レイテンシが 1 である演算器 —— 通常の構成では ALU —— からのオペランド・バイパスを取り除くことと等価である．詳細は 4 章で述べるが，それによる IPC の悪化は 5〜15%程度にもなり，クロック速度の向上に見合わない可能性が高い．このような観点から，レイテンシが 1 であるパスに関しては，*wakeup* と *select* は合わせて 1 サイクルで実行しなければならないとできる．

2.3 *Wakeup* とタグ

従来のスケジューリング方式は，**タグ**に基づく連想検索によって，前節で述べた *wakeup* の動作を実現する．タグとは，端的に言えば，バックエンドに存在するデータを一意に特定するための ID である．

タグはデータに付される ID の一種であるから，どんなシリアル番号でもその役割を果たすことができる．しかし，out-of-order な実行結果を格納するバッファのエントリ番号を用いるのが最も都合がよい．バッファのエントリ番号は，明らかに，そこに格納される実行結果と 1 対 1 の関係があるからである．

タグの割り当てとそのタグとしての使用は，*rename* と *wakeup* において行われる．したがって，前節の説明は以下のように補足することができる：

(1) *Rename* まず，命令にはバッファのエントリが割り当てられる．このエントリの番号を tagD と呼ぶ．同時に，左/右のソースに対応するエントリに割り当てられた tagD が求められる．これらのエントリの番号を，tagD と区別して，tagL/R と呼ぶ．

(3) *Wakeup* *Wakeup* はタグに基づく連想検索によって行われる．命令が発行されるときには，その tagD がウィンドウ内のすべての命令に対して放送される．ウィンドウ内で『眠っている』命令は，放送された tagD と自らの tagL/R を比較する．一致すれば，対応する rdyL/R がセットされる．

2.4 命令スケジューリングの論理

5 つのフェーズのうち，*dispatch*，*select*，*issue* の論理は自明であるので，本節では，*rename* と *wakeup* について述べる．*Rename* と *wakeup* の処理は共に，突き詰めれば，依存関係にある命令を検索する処理，すなわち，依存解析である．しかし，以下で述べるように，その複雑さはかなり異なる．

2.4.1 *Rename*

rename は，前述したように，tagD の割り当てと，tagL/R を求める処理からなる．割り当てるべき tagD は，十分前もって求めておくことができるから，性能上重要なのは後者である．

tagL/R を求める上では，論理レジスタ番号からタグへの写像を記録するレジスタ・マップ・テーブル (**RMT**) が中心的な役割を果たす．RMT の実装にはいくつかの方式があるが，本稿では RAM 方式について述べる．RAM 方式の RMT は，論理レジスタ番号をアドレス，タグを内容とする，2 · IW-read IW-write, $\boldsymbol{TW}$ b×WS word の RAM である．ただし，TW はタグのビット幅である．

I_p は，デスティネーションの論理レジスタ番号をアドレスとして，割り当てられた tagD を RMT に書き込む．一方 I_c は，基本的には，左/右ソースの論理レジスタ番号をアドレスとして RMT を読み出すことによって tagL/R を得ることができる．

ただし，同時にデコードされる命令間に依存がある場合には，RMT からは『古い』tagL/R が得られるので，以下の調整が必要である；比較器のアレイによって論理レジスタ番号の比較を行い，同時にデコードされる命令間の依存を検出する．依存が検出された場合には，RMT から読み出される『古い』tagL/R の代わりに，RMT に書き込まれようとしている『新しい』tagD を用いればよい．

2.4.2 *Wakeup*

通常の *Wakeup* の論理は，実行される命令の tagD を RAM から読み出し，それに一致する tagL/R を CAM から連想検索するという構造を持つ．

発行ウィンドウは，単一の論理として集中的に実装されるのではなく，実行ユニットのクラスごとに用意されたリザベーション・ステーション，あるいは，命令キューとして分散実装されることが多い．各命令キューの発行幅，サイズを $\boldsymbol{IW_q}, \boldsymbol{WS_q}$ とすると，各キューの RAM は，IW_q-read，IW_q-write，TW b×WS_q word，CAM のキー部は，IW_q-write，TW b×2 · WS_q word となる．CAM の比較入力ポート数は，IPC とのトレードオフとなる．例えば，MIPS R10000 では，整数演算，ロード/ストア (**LS**)，浮動小数点 (**FP**) 演算のそれぞれに，$IW_q = 2$，$WS_q = 16$ の命令キューを持つ．整数命令キューの CAM の比較入力ポート数は，3〜4（$IW_q + 1$〜2）としている．

2.2 節で述べたように，*wakeup* と *select* と合わせて 1 サイクルで実行する必要がある．これらのうち，*select* の遅延は専らゲート遅延からなるため，LSI の微細化に伴って順調に短縮されると予測される．一方 *wakeup* の遅延は，RAM と CAM のワード線，ビット線，マッチ線などの配線遅延からなるため，LSI の微細化の恩恵を受けにくい．以上の理由により *wakeup* は，LSI の微細化，パイプラインの深化にともなっていっそうクリティカルになっていくと予測されるのである．

3. 提案する動的命令スケジューリング方式

従来のスケジューリング方式では，依存関係にある命令の検索，すなわち依存解析を，*rename* と *wakeup* の 2 回に渡って行っている．*Rename* では，tagL/R を得るために，消費側の命令がそれぞれのソース・オペランドの生産側の命令を探す．*Wakeup* では，rdyL/R を更新するため，生産側の命令が消費側の命令のソース・オペランドを探す．

しかし，前章で述べたように，その複雑さはかなり異なる．それは以下の理由による：

(1) 検索の方向 *Wakeup* では，生産者側の命令は，複数存在し得るすべての消費者のすべてのソース・オペランドを探す必要がある．そのため *wakeup* における検索は，原理的に連想的にならざるを得ない．
一方 *Rename* では，消費者側の命令は，それぞれのソース・オペランドに対して，たかだか 1 つしか存在し得ない生産者を探せばよい．

(2) プログラム・オーダ レジスタ・リネーミングのため，ソース・オペランドである論理レジスタに書き込みを行う命令は複数存在し得る．しかし，ある命令の生産者はそれらのうち命令流上で最新のものである．各命令は *Rename* を in-order で通過するため，*rename* はその最新の生産者を知ることができる．

以上の 2 つの理由により，*rename* における依存解析は大幅に簡単化される．RMT では，論理レジスタごとに最新の生産者（の tagD）のみを覚えておけばよく，検索は直接的に行うことができる．

一方 *wakeup* における検索は，連想的かつ悉皆的にならざるを得ない．その上，*rename* はパイプラインのフロントエンドにあるため，必要であればパイプライン化することができるのに対して，*wakeup* はパイプライン化不能である．

Wakeup のための依存解析，すなわち，生産者側の命令からその消費者側の命令（のソース・オペランド）

を探す処理自体は，スケジューリングにとって本質的であり，省略することができない．しかし，その処理のすべてを必ずしも *wakeup* において実行する必要はない．したがって，*wakeup* を簡単化する鍵は，*wakeup* のために不可避である依存解析の『重い』部分を，*rename* と同様の方法によって，パイプラインのフロントエンドで済ませておくことにある．

本稿で述べる方式は，その具体的な方法の 1 つである．提案方式では，従来のタグによる連想処理に基づくものとは根本的に異なり，各命令間のデータ依存関係を直接的に表す**D行列**が，スケジューリングにおける中心的な役割を果たす．*Wakeup* に必要となる依存解析の大部分は，D行列の更新ロジックが，*rename* と同様の方法で，パイプラインのフロントエンドで済ませておくことができる．そのため *wakeup* は，単に D 行列を読み出すだけで実現することができる．以下本章では，D行列とその操作について詳しく述べる．

3.1 D 行 列

図 1 に，D行列の概念図を示す．D行列は，rdyL/R 用に各 1 つづつ用意された $WS \times WS$ の行列である．ウィンドウ内のエントリ ID $= p$ の命令 I_p の実行結果を ID = c の命令 I_c が消費する場合，c 行 p 列の要素は “1”，そうでなければ “0” とする．

Wakeup においては，発行される IW 個の I_p に対応する IW 列の bitwise-OR を求めれば，セットすべき rdyL/R を表す列ベクトルを求めることができる．

3.2 D行列の実装

D行列を実装するにあたっては，単に 1-read の RAM を用いればよい．複数行の OR を求めるための特別なロジックは必要ない．

図 2 に，D行列の回路図を示す．同図には，2 行 2 列分のセルが示してある．各セルの中央にある 4T セルの左側は，更新のための書き込みポートである．*wakeup* に関連するのは，4T セルの右下側にある読み出しポートである．

3.3 D行列の更新

D行列の更新は 2 つのフェーズからなり，第 1，第 2 のフェーズはそれぞれ *rename*，*dispatch* と同時に進行する．特に，*rename* と同時に進む第 1 のフェーズは，*rename* と同形の論理によって実現される．

今デコードされている I_c が ID $= c$ のエントリに格納されるとしよう．D行列の更新処理の 2 つのフェーズは，以下のようになる：

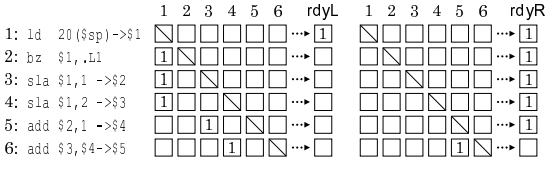

図 1 D行列

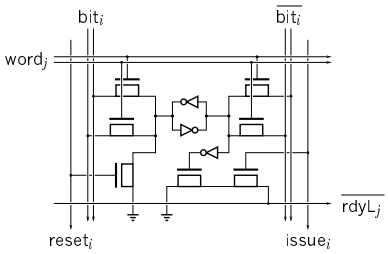

図 2 D行列のセル（rdyL 用）

(1) I_c の依存元の命令 I_p が格納されているエントリの ID $= p$ を求める．この処理は，前述した *rename* において tagL/R を求める処理と同様に，論理レジスタから生産者へのマッピング・テーブルによって実現される．より具体的には，2.4.1 項の説明において，『タグ』と『RMT』を『I_p のエントリ ID p』と『**生産者テーブル (PT)**』に読み替えればよい．PT は，RMT と同じく，論理レジスタ番号をアドレスとする，$2 \cdot IW$-read IW-write の RAM である．ただしその内容は，生産者のエントリ ID p である．

同時にデコードされる命令間の依存を検出するための検出器は，*rename* と共用することができるため，提案方式のために別途必要となるのは，主に，PT である．

(2) p をデコードし，行列の c 行に書き込む．

更新処理の遅延

更新処理の第 1，第 2 フェーズの遅延は，同時に行われる *rename*，*dispatch* の遅延と，それぞれ以下のように比較できる：

(1) バッファのエントリ数は WS の倍程度とすることが普通であるから，PT の内容であるエントリ ID は，RMT の内容であるタグより 0〜1b 程度短い．したがって，PT の遅延は RMT のそれより小さく，そちらがクリティカルになることはない．

(2) ウィンドウへの書き込みの遅延は，通常，ウィンドウを構成する RAM の行デコーダとワード線の遅延に支配される．p のデコードはこの行デコーダと等価な回路で実現できるから，その遅延が表面化する可能性は低い．

以上から，D行列の更新がクロック速度の低下やデコード・ステージ数の増加を招く可能性は低いと言える．

3.4 D行列による *wakeup*

図 2 からも明らかなように，D行列の読み出しポート部は，single-bitline の 1-read の RAM と基本的には同じものである．ただし，通常の RAM とは，以下の点が異なる：

構造上 書き込みポートと読み出しポートでは，位置関係が逆になっている．読み出しポートでは，ワード線が縦に，ビット線が横にひかれている．

動作上 通常の RAM では同時にはたかだか 1 つのワード線しかアサートされないのに対して，D行列では毎サイクル実行される I_p に対応する IW 本のワー

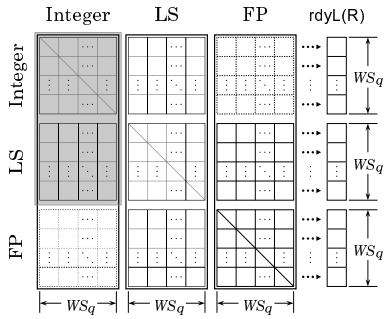

図 3　D行列の分散化

ド線 issue が同時にアサートされる．各ビット線 $\overline{\text{rdyL}}$ には，アサートされた issue に対応する IW 個のセルが接続され，いずれかのセルの出力が low であれば pull-down される．すなわち，単に複数のワード線を同時にアサートすることによって，対応する列の bitwise-OR を読み出すことができるのである．

3.5　D行列における投機失敗からの状態回復

最近のスーパースケーラでは，分岐予測による投機実行が必須である．命令スケジューリングの方式にとっては，投機失敗時の状態回復処理が容易であることが重要である．

D行列では，分岐予測ミス時には，無効化される命令に対応する列をすべて “0” にリセットするだけでよい．図 1 の例で ID = 2 の分岐命令が予測ミスを起こした場合には，ID = 3〜6 の命令が無効化される．D行列では，第 3〜6 列をリセットすればよい．

列のリセットは命令ウィンドウの他の情報の無効化と同程度のコストで実行できるから，投機失敗時に D行列に関連して新たなペナルティが生じることはない．

3.6　D行列アクセスの高速化

3.6.1　D行列の分散化

命令ウィンドウが q 本のサイズ WS_q の命令キュー分散される場合には，D行列は，q 個の WS 行 WS_q 列の部分行列に分割される．R10000 では，整数，LS，FP 命令に対して 3 つの命令キューを持つ．その場合の D行列の分散化の様子を図 **3** に示す．

D行列の分散化には，以下の 2 つのメリットがある：

パラメータの縮小　それぞれの D行列では，書き込みポート数が IW から IW_q に減少する．

レイテンシに合わせた最適化　レイテンシが 1 のパスでは，*wakeup* と *select* は合わせて 1 サイクルで実行する必要があるが，それ以外のパスでは適当にパイプライン化してよい．

R10000 の構成では，レイテンシが 1 であるパスは，整数から整数，整数から LS の 2 つであり，図 3 では影を付けた部分がこれに相当する．この部分を取りだし，これを **L-1**，残りの部分を **L-2 D行列** と呼ぶ．

L-2 D行列 は，適当にパイプライン化してよい．L-1 に 0.5 サイクルを充てるとすると，L-2 には 1.5 サイクル，すなわち，L-1 の 3 倍の時間をかけられる．したがって，L-2 がクリティカルになる可能性は低く，*wakeup* の遅延についての議論から除外することができる．

一方 L-1 D行列 は，元の D行列から比べると格段に小型化されるため，その読み出し遅延も短くなる．更に，L-1 D行列 に対しては，以下に述べる高速化手法を適用することができる．

3.6.2　L-1 D行列 の縮小

依存する命令間の距離は短い場合が多く，32 命令以下の場合が 90%程度以上を占めることが分かっている．この性質を利用して，*wakeup* を更に高速化することができる．具体的には，後続の w 命令に対するビットだけを L-1 D行列に残し，それ以外を L-2 D行列に移すのである．rdyL/R は命令間の距離が w 以下の場合には L-1 D行列によって，越えていた場合には L-2 D行列によって更新される．

図 **4** に，D行列の縮小の様子を示す．左は元々の，右が縮小後の D行列である．同図では，$WS = 8$，$w = 4$ である．元々の D行列から削除されるセルは，左図中で薄く示した．必要なセルを矩形領域に集める方法には任意性があるが，よりクリティカルであるビット線の長さを短縮することを優先して，図 4 右のようにするとよいであろう．　w をL-1 D行列の幅と呼ぶ．

図 4 右から明らかなように，縮小された D行列のワード線，ビット線は，それぞれ w 個のセルにしか接続されておらず，それぞれの長さは WS とは無関係となっている．したがって *wakeup* の遅延は，　WS とは独立に，w によって決めることができる．

Pace-Keeping Operation

L-1 D行列によって rdyL/R が更新できるのは，命令間の距離ではなく，ウィンドウ内のエントリ間の距離がL-1 D行列の幅以下の場合である．したがって，命令流上での距離とウィンドウ内での距離がある程度一致するように制御する必要がある．

このことは特に，命令ウィンドウを複数の命令キューに分散化する場合に問題となる．上記の条件を満たすためには，エントリをサイクリックに使用した上で，異なる命令キュー間ではエントリの消費の歩調を揃え

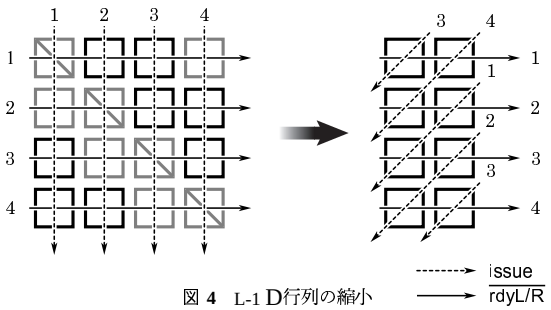

図 4　L-1 D行列の縮小

る (**keep pace**) 必要がある．R10000 の構成の場合，整数と LS キュー間では，別のキューにディスパッチされた後続の命令は，先行する命令が格納されたエントリの『横以降』に格納する必要がある．

4. IPC の評価

SimpleScalar ツールセット (ver.2.0) に対して，3.6 節で述べたL-1 D行列の縮小を実装し，SPEC ベンチマークを用いてL-1 D行列の幅に対する IPC の変化を測定した．**表 1** に示す CINT95 のプログラムを実行した．

ベース・モデルとしては，MIPS R10000 を用いた．R10000 は，整数演算，LS，FP 演算のそれぞれに命令キューを持ち，$(IW_q, IW, WS_q, TW) = (2, 4, 16, 6)$ である．命令/データ分離 1 次，および，統合 2 次キャッシュの，容量，ライン・サイズ，レイテンシは，それぞれ，32KB，32B，1 サイクル，および，1MB，64B，6 サイクルである．2 次キャッシュ・ミス時には，最初のワードに 18 サイクル，後続ワードには 2 サイクル/ワードが必要である．分岐予測には，ツールセットに用意されている 2b 飽和型カウンタによるもの (bimod) を用いた．これを，構成 R10K×1 と呼ぶ．更に，キャッシュ以外のすべての資源を 2 倍，すなわち，$(IW_q, IW, WS_q, TW) = (4, 8, 32, 7)$ とした構成 R10K×2 も合わせて測定した．また，w の上限を求めるため，すべての資源を無限大にし，キャッシュ，分岐予測を完全とした構成 R10K×∞ も測定した．

L-1 D行列の幅に対する IPC の比率を図 **5** に示す．なお，$w = WS_q$ の場合も本来不要な pace-keeping を行っており，その時の IPC の低下は pace-keeping のみの影響を示している．

R10K×1/2 の結果からは，$w \geq WS_q/4$ では，ほとんど pace-keeping の影響だけであり，IPC の低下は 1%程度以下に抑えられることが分かる．

また，R10K×∞ の結果からは，w は 16 あれば IPC の大きな落ち込みはなく，64 あれば十分であることが分かる．

5. おわりに

本稿では，命令間の依存関係を表す D行列を用いたスケジューリング方式とその IPC の評価について述べた．本方式では，*wakeup* に必要となる依存解析を D行列の更新ロジックがフロントエンドで済ませておくため，単に D行列を読み出すだけで *wakeup* を実現することができる．また，D行列を縮小することによってその遅延を IPC ペナルティに転化することができる．

本稿では，その手法における IPC の評価を行った．MIPS R10000 の構成から，IW_q, WS_q を無限にし，キャッシュや分岐予測を完全とした場合にも，D行列の幅は 16 あれば IPC の大きな低下はないことが分かった．実際には，IW_q, WS_q を十分に大きくしたとしても，キャッシュや分岐予測のミスによって性能はかなり低下するから，D行列の幅はもう少し小さくてもよい．

提案方式の遅延には，D行列の書き込みポート数として，IW_q の影響は若干残るものの，WS_q の影響はほとんどない．すなわち，提案方式によって，スーパースケーラの命令スケジューリング・ロジックの遅延はウィンドウ・サイズからはほぼ完全に独立となったと結論づけることができる．

謝辞

本研究の一部は文部省科学研究費補助金，基盤研究 (B)(2) #12480072，#12558027，#13480083 による．

表 1 各プログラムの実行命令数 (M)

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

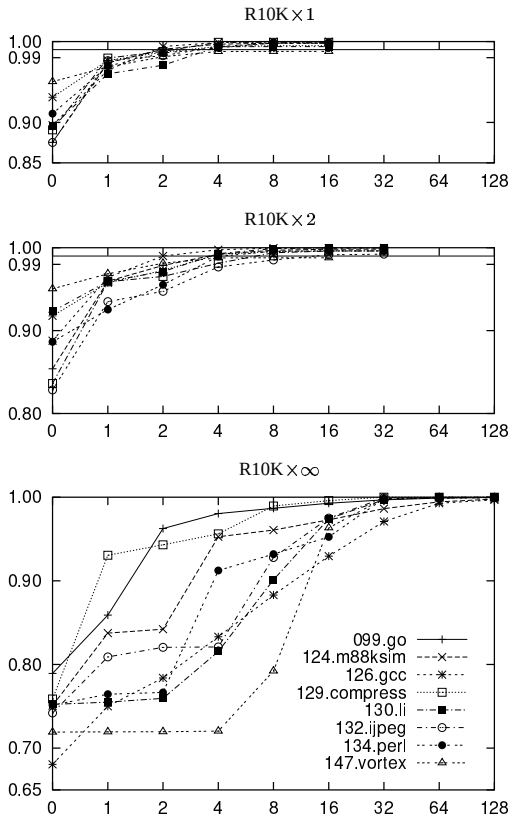

図 5 L-1 D行列の幅に対する IPC

参考文献

1) 五島正裕, 西野賢悟他: スーパースケーラのための高速な動的命令スケジューリング方式, 情報処理学会論文誌, Vol. 42, No. HPS 3 (2001).