クラスタ化スーパースケーラ・プロセッサにおける直接依存行列型スケジューリング方式

小西将人[†] 小田累[†] 西野賢悟[†]
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・プロセッサにおけるバイパス・ロジックは将来的にクロック速度を制限すると予測されており，この問題に対してプロセッサをクラスタ化する研究が行われている．一方，我々は，やはりクリティカルになると予測される命令スケジューリング機構の wakeup と呼ばれるロジックに対し，これを高速化する手法である直接依存行列型スケジューリングを提案している．本稿では，クラスタ化プロセッサと，依存行列型スケジューリング方式を組み合わせた場合について議論する．我々の行列を用いる方式では，wakeup の遅延は IPC とトレード・オフの関係にあり，IPC の悪化を許容範囲内に抑えつつ，どこまで遅延を減少できるかは，命令ウィンドウのエントリ数による．クラスタ化プロセッサでは，命令ウィンドウの構成として集中/分散の 2 種類に分類でき，分散されている場合には，各ウィンドウのエントリ数も減少でき，更に wakeup の遅延も減少すると考えられる．一方で，その構成の違いから，分散型は集中型と比較して IPC は落ちると考えられる．そこで，両方式の利点を併せ持つと考える仮想的に命令ウィンドウを分散する方式を提案する．

Direct Dependence Matrix-Based Scheduling Scheme on Clustered Superscalar Processors

Masahito Konishi,[†] Rui Oda,[†] Kengo Nishino,[†] Masahiro Goshima,[†]
Yasuhiko Nakashima,[†] Shin-ichiro Mori,[†] Toshiaki Kitamura[†]
and Shinji Tomita[†]

It is predicted that the Bypass logic in a superscalar processor will restrict clock speed in the future, and to this problem, clustering processor is studied. On the other hand, the logic called wakeup will be critical, so we have proposed **Direct Dependence Matrix Based Scheduling Scheme** which decreases delay of wakeup. This paper describes about the case where our scheme is applied to clustered processors. By our scheme, the delay of wakeup and IPC have a relation of a trade-off. It is dependent on size of instruction window, how far the delay can be decreased, suppressing aggravation of IPC in tolerance level. Clustered processors can be classified into two kinds such as Centralized/Decentralized instruction window. In Decentralized type, the size of each window is small, and the delay of wakeup also can be decreased. One side considered that IPC aggravates as compared with Centralized, because of difference in the composition. So we propose new type of clustered processors, that has virtually distributed instruction window. We think that this type has advantage of both IPC and the delay of wakeup.

1. はじめに

スーパースケーラ・プロセッサの IPC (Instructions Per Cycle) を向上させる最も直接的な方法は，命令発行幅 (***IW***: Issue Width) とウィンドウ・サイズ (***WS***: Window Size) を増やすことである．実際 初期のスーパースケーラ・プロセッサは，トランジスタ数が許す範囲で IW, WS を増やすことにより，大幅に IPC を向上させてきた．

しかし現在では，LSI の微細化にともなって，トランジスタ数ではなく，クロック速度が IW, WS を制限する主因となりつつある．IW, WS を増やしても単純に IPC が向上するわけではないので，いたずらに増加させれば，かえって性能を悪化させることになる．

将来クロック速度を制限する構成要素としては，オペランド・バイパスと，命令スケジューリング・ロジックの一部である ***wakeup*** と呼ばれるロジックの 2 つが挙げられる．これらの遅延は，IW, WS の増加関数で与えられる上，配線遅延の影響を強く受ける．配線遅延は，LSI の微細化にともなって，全体の遅延を支配していくと考えられている [1)~5)]．

† 京都大学
Kyoto University

バイパスの遅延の増大に対する対処法としては，DEC Alpha 21264 で採用された，EU のクラスタ化がある [5)~9)]．*Wakeup* ロジックの遅延に対しては，我々は直接依存行列型と呼ぶ方式を提案した [1)~3)]．本稿では，これらの 2 方式の組み合わせについて議論する．

クラスタ化スーパースケーラ・プロセッサ

EU のクラスタ化とは，具体的には，クラスタ間に跨るオペランド・バイパスを省略することである．別のクラスタに割り振られた 2 つの命令は，利用すべきバイパスが存在しないため，back-to-back に実行することができない．そのため，実際にそれらの命令がクリティカル・パス上に乗ってしまった場合には通常 1 サイクルのペナルティが発生する．その一方で，バイパスの配線長は数分の 1 に短縮されるため，その遅延を大幅に短縮することができる．

直接依存行列型スケジューリング方式

直接依存行列型スケジューリング方式は，ウィンドウに格納された最大 WS 個の命令間の依存関係を表す，WS 行 WS 列の行列を用いる．この方式では，この行列を格納する RAM を読み出すだけで *wakeup* を実現することができる．

また，行列の狭幅化と呼ぶ技術により，この RAM の実効幅を大幅に縮小することができる．IPC の低下は，狭幅化後の行列の幅 w に依存する．詳しい理由は後述するが，行列を WS 列から w 列へと縮小する場合，w 命令以上離れた 2 つの命令はやはり back-to-back に実行することができなくなる．そのため，クラスタ化の場合と同様に，1 サイクルのペナルティが発生することがある．ただし，$w \geq WS/4$ であれば，IPC の低下は 1 ~ 2% 程度に抑えられることが分かっている．その一方で，やはりクラスタ化の場合と同様に，RAM の配線長は数分の 1 に短縮され，その遅延を大幅に短縮することができる．

組み合わせ

クラスタ化プロセッサは，命令ウィンドウの構成により，集中型と分散型の 2 つに分類することができる．集中型ではウィンドウを集中化された単一のロジックで構成するが，分散型ではクラスタに合わせてウィンドウを複数のロジックとして分散配置する．

分散型では，ウィンドウ自体が小型化されるため遅延の点では集中型より有利であるが，IPC の点では集中型より不利である．分散型は，命令のウィンドウへのディスパッチ時にその命令を実行するクラスタを決定する (steering) 必要があるため，ディスパッチから発行までの状況の変化に対応できない．

依存行列型スケジューリング方式と組み合わせる際には，分散型には，遅延の点で更に有利な点が生まれる．分散型では，ウィンドウに合わせて，行列も分散することができる．その幅 w のベースとなる WS も小さくできるため，より小さい w に対して耐性を持つと考えられるのである．

本稿の目的は，これらの効果を定量的に評価することにある．以下まず，2 章で依存行列型スケジューリング方式について概説した後，3 章でそれとクラスタ化の組合わせについて詳しく議論する．最後に 4 章で IPC の評価を行う．

2. 依存行列型スケジューリング方式 [1)~3)]

文献 1)~3) で提案した依存行列型命令スケジューリングは，スケジューリングの処理のうち，*wakeup* と呼ばれる部分を高速化する．以下まず 2.1 節では，一般的な out-of-order スケジューリングについて述べ，*wakeup* の処理と，なぜその遅延の短縮が重要であるかについて明らかにする．その後 2.2 節から，依存行列を用いた *wakeup* について詳述する．

2.1 Out-of-Order スケジューリング

Out-of-orderßは，論理的なレジスタとは別に，各命令の out-of-order な実行結果を保存するための物理レジスタを必要とする．物理レジスタは，リオーダ・バッファ，あるいは，物理レジスタ・ファイルなどによって実現される．

Out-of-order スケジューリングは，この物理レジスタを用いたデータ駆動型の計算とみなすことができる．すなわち，命令ウィンドウ中で『眠っている』命令は，ソースに割り当てられた物理レジスタにデータが書き込まれることによって『起こされ (*wakeup*)』，プログラム・オーダとは独立に実行を開始することができる．

Out-of-Order スケジューリングのフェーズ

Out-of-order スケジューリングの処理は，以下の 5 つのフェーズに従って進む．命令 I_p の結果を命令 I_c が消費するものとする：

(1) *Rename* フェッチされた命令に対して，まずレジスタ・リネーミングが施される．

(2) *Dispatch* その後命令は，命令ウィンドウに格納される．I_p がまだ実行されていない場合，I_c は I_p の実行を待ってウィンドウ内で『眠る』．

(3) *Wakeup* 次の *select* によって I_p の発行が許諾されると，I_c は『起こされ』，自らの発行を要求する．

(4) *Select* *Select* では，発行要求が調停され，選択された命令の発行が許諾される．

(5) *Issue* 発行を許諾された命令の情報が命令ウィンドウから読み出され，実行ユニットに送られる．

図 **1 (L-1)** に，*wakeup*，*select*，*issue* の命令パイプラインでの位置とその動作の様子を示す．同図は，MIPS R10000 のパイプライン構成に準ずる [10)]．図中，Exec は実行を，RF→と→RF は物理レジスタ・ファイルに対する読み出しと書き戻しを表す．同図は，タイミングが最もクリティカル，すなわち，レイテンシが 1 である命令 I_p の次のサイクルで I_c が実行される場合を示している．I_p が生成したデータは，オペランド・バイパスを通って I_c の実行に使用される．

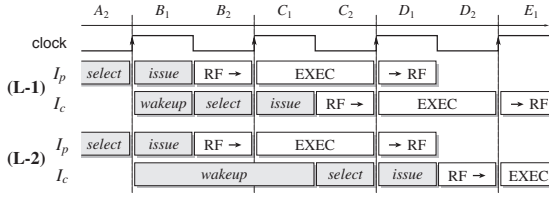

図 1 命令パイプラインにおける *wakeup*，*select*，*issue*

フェーズとパイプライン

命令スケジューリングの 5 つのフェーズのうち，*rename*，*dispatch*，および，*issue* は，必要であれば，パイプライン化を施すことによってシステムのクリティカル・パスから外すことができる．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ß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，実際上パイプライン化できない．図 1 **(L-2)** に，*wakeup* に 1 サイクル余分にかけた場合の命令パイプラインの様子を示す．*Select* から *wakeup* へのフィードバックにより，(A_2) I_p に対する *select* が終わった後にしか，(B_1) I_c に対する *wakeup* は開始できない．その結果，I_c の発行は 1 サイクル遅れ，I_p と back-to-back に実行できなくなる．

同図では，I_p が生成したデータは，オペランド・バイパスを通る必要はなく，レジスタ・ファイルを介して I_c の実行に使用される．すなわち，*wakeup+select* に 1 サイクルより多くを割り当てることは，IPC の観点からは，レイテンシが 1 である EU —— 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．それによる IPC の悪化は最大 15%程度にもなり [1)~3)]，動作周波数の向上に見合わない可能性が高い．したがって，レイテンシが 1 であるパスに関しては，*wakeup* と *select* からなるループ一周の遅延は 1 サイクル以内でなければならないと結論づけられる．逆に言えば，このループ一周の遅延がクリティカルである場合には，それがシステムの動作周波数を決定することになる．

このうち *select* の遅延は，専らゲート遅延からなり，LSI の微細化にともなって順調に短縮されると予測される．一方従来の *wakeup* の遅延は配線遅延に支配されている．そのため，将来 *wakeup* の遅延がよりクリティカルになると考えられるのである [1)~5)]．

2.2 依存行列による *wakeup*

依存行列によるスケジューリングでは，命令間の依存関係を表す行列を用いて *wakeup* を実現する．図 **2** に，依存行列の概念図を示す．行列は，*WS* 行 *WS* 列であるが，対角要素は使用しない．ウィンドウの p 番エントリに格納された命令 I_p の実行結果を c 番の命令 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，各行で 1 である列に対応するすべての命令の発行が許諾されれば，その行に対応する命令は実行可能であり，request フラグをセットしてその発行を要求すればよい．

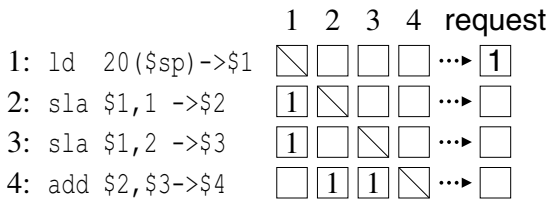

図 2 依存行列

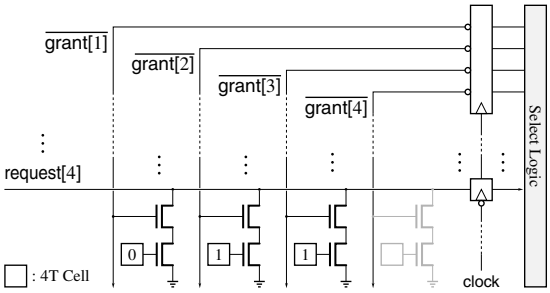

図 3 依存行列のロジック

行列は，図 **3** に示す，RAM に類似のアレイ状の回路によって実現される．同図は，図 2 の 4 行目に対応している．request[4] は high にプリチャージされ，$\overline{\text{grant}[2/3]}$ が双方ともアサートされた場合には high に保たれるが，そうでない場合にはプルダウンされる．

2.3 行列アクセスの高速化

本節では，行列アクセスの高速化手法として，行列の分散化，多階層化，狭幅化について述べる．それぞれの手法は本来独立に適用可能であるが，組み合わせて用いることでより効果を発揮する．そのため以下では，上記の順序で順に適用するという流れで各手法について説明する．

2.3.1 行列の分散化

実際のプロセッサでは，命令ウィンドウは，EU の種類ごとに設けられたリザベーション・ステーションや命令キューとして分散実装されることが多い．実際，最近のレジスタ・リネーミング方式のプロセッサの多くは，整数 (**INT**)，ロード/ストア (**LS**)，浮動小数点 (**FP**) 命令の系統ごとにウィンドウを分散化している．例えば MIPS R10000 は，INT 用，LS 用，FP 用のサブウィンドウを持つ [10),11)]．このような分散化は，わずかな IPC のペナルティを犠牲に，ロジックを大幅に縮小できるため，非常に重要である．

R10000 と同様の分散化を施した場合の行列の分散化の様子を図 **4** に示す☆．R10000 では，INT，LS，FP

☆ SPARC など，INT–FP レジスタ間の転送命令が無い ISA では，同図中破線で記した部分行列を省略できる．

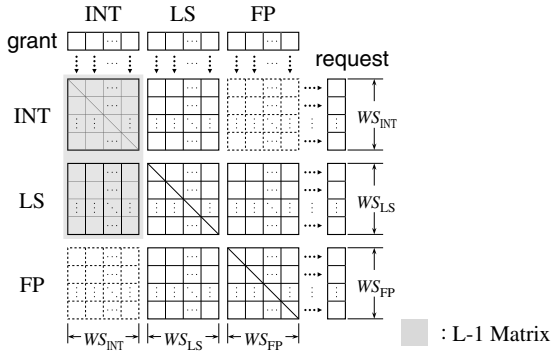

図 4　行列の分散化と多階層化

の各サブウィンドウの発行幅とサイズはそれぞれ，$IW' = IW/3 = 2$，$WS' = WS/3 = 16$ である．

分散化の行列に対する第一の効果は，書き込みポートの削減である．INT 命令を *dispatch* するときに書き込まれる行は，対応する WS' 行に制限される．この部分のセルの書き込みポート数は，IW から IW' へと 1/3 に削減される．LS，FP についても同様である．

また，R10000 のように INT のみのサブウィンドウを持つ場合，次項で述べる多階層化が可能になる．

2.3.2　行列の多階層化

R10000 の構成では，レイテンシが 1 であるパスは，INT から INT，INT から LS の 2 つである．図 4 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**，残りを **L-2** 行列と呼ぶ．

L-1 アクセスは *select* と合わせて 1 サイクル以内に行う必要があるが，L-2 はパスのレイテンシに合わせて適当にパイプライン化してよい．図 1 (L-2) は，レイテンシが 2 の場合にあたる．同図では，L-2 には 1.5 サイクル，すなわち，L-1 の 3 倍の時間をかけており，L-2 がクリティカルになる可能性は極めて低い．

一方 L-1 は，元の 行列から比べると格段に少量化され，その分だけ遅延も短縮される．更に，L-1 に対しては次節で述べる狭幅化を適用することができる．

2.3.3　**L-1** 行列 の狭幅化

依存する命令間の距離は短い場合が多い．この性質を利用して，L-1 を更に縮小することができる．各行において，先行する w 命令に対するビットだけを残し，それ以外を L-2 に移すのである．*Wakeup* は，命令間の距離が w 以下の場合には L-1 によって，そうでない場合には L-2 によって行われる．後者の場合には，1 サイクルのペナルティが生じる．

図 **5** に，L-1 の縮小の様子を示す．左は元々の，右が縮小後の L-1 である．L-2 に移されるセルは，左図中で薄く示してある．メモリのラインのうちでは，書き込みポートのビットラインが $2 \cdot IW'$ 本と最も本数が多い．したがって図 5 では，それらが直線になるように，L-1 に残されるセルを矩形領域に集めている．w は，L-1 を構成するアレイの幅にあたる．そのため，この技術を **L-1** 行列の狭幅化と呼ぶ．

図 5 右から明らかなように，狭幅化された L-1 のワード，ビットラインは，それぞれ w 個のセルにしか接続されていない．したがって *wakeup* の遅延は，WS とは独立に，w によって決まる．

なお，各サブウィンドウ間では，エントリ消費の歩調を揃える **pace-keeping** と呼ぶ操作が必要であり，ウィンドウ・エントリ の使用効率が若干悪化する．$w \geq \min(WS'/4, 64)$ であれば，L-1 ミスではなく，主に pace-keeping によって IPC は低下し，それ IPC の低下は 1 ~ 2%に抑えられることが分かっている [1)~3)]．

3.　クラスタ化と行列方式の組み合わせ

EU のクラスタ化とは，具体的には，クラスタ間に跨るオペランド・バイパスを省略することである．依存する 2 つの命令 I_p と I_c が別のクラスタに割り振られた場合には，利用すべきバイパスが存在しないため，I_p の実行結果を I_c はその次のサイクルで受け取ることができず，I_p と I_c は back-to-back に実行することができない．クラスタ間でデータを引き渡すには，通常 1 サイクルのクラスタ間遅延 (cluster delay) を要する．実際に I_p と I_c がプログラムのクリティカル・パス上に乗ってしまった場合には，1 サイクルのペナルティが発生することになる．その一方で，EU を NC 個のクラスタに分割すると，バイパスの配線長は $1/NC$ 程度にまでに短縮さ，その遅延を大幅に短縮することができる [5)~9)]．

3.1　クラスタ化プロセッサの命令ウィンドウの構成

クラスタ化プロセッサは，命令ウィンドウの物理的な構成により，大きく集中型と分散型の 2 つに分類することができる．集中型では単一のロジックとして構成される命令ウィンドウを，分散型では各クラスタに 1 対 1 に対応する複数のサブウィンドウに分散化する．集中型では命令は単一のウィンドウからすべてのクラスタに対して発行されるのに対して，分散型では命令は各サブウィンドウから 1 対 1 に対応するクラスタに

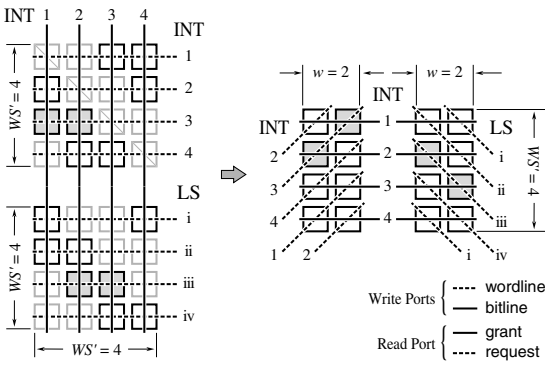

図 5　L-1 行列の狭幅化

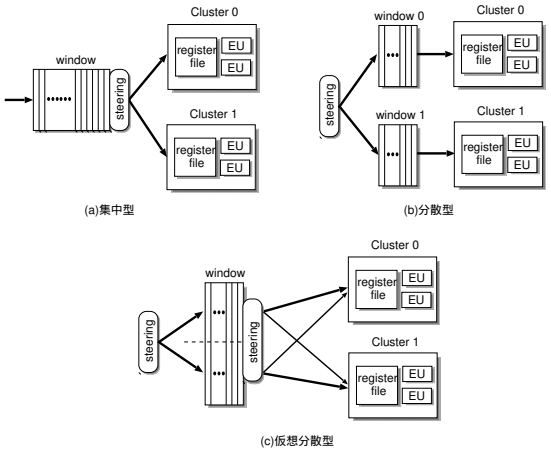

図 6　クラスタ化プロセッサの分類

対してのみ発行される．

集中型と分散型では，命令のクラスタへの割り振りのタイミングが異なる．集中型では命令の割り振りは *select* フェーズで行われるが，分散型では，何らかのヒューリスティクスに基づいて，*dispatch* フェーズで行う必要がある．IPC はその精度に依存する [5)~7)]．

その一方で，ウィンドウを分散化することで，ウィンドウ・ロジックの遅延を削減することができる

Wakeup　次節で詳しく述べる．

Select　*Select* ロジックでは，*WS* 個の命令から *IW* 個を選択する必要があるものが，*WS*/*n* 個の命令から *IW*/*n* 個を選択するだけでよくなる．*Select* ロジックの遅延は $O(\log WS \times \log IW)$ で与えられ，*WS*，*IW* の削減ともに，対数オーダの効果がある [12)]．

Dispatch，***Issue***　ウィンドウを構成する RAM 自体が小型化されるため，*dispatch*，*issue* の遅延も削減される．ただし，2.1 節で述べたようにこれらの処理はパイプライン化可能であるので，*wakeup+select* に比べるとその影響は決定的ではない．

なお，ここで述べたクラスタに対するウィンドウの集中/分散と，2.3.1 項で述べた命令の系統に対する集中/分散とは，直行関係にあることに注意する必要がある．例えば DEC Alpha 21264 では，INT+LS と FP の 2 つの命令系統に分散化した上で，INT+LS の EU を更に 2 つのクラスタに分割している．INT+LS のサブウィンドウは，実際にはクラスタに対しては分散化されていないが [8)]，分散化することも可能である．

3.2　分散型と行列方式の組み合わせ

依存する 2 つの命令 I_p と I_c が別のクラスタに割り振られた場合には，クラスタ間遅延のため，I_p と I_c は back-to-back に実行することができない．したがって，I_c に対する *wakeup+select* も 1 サイクルで実行する必要がなくなる．図 1 (L-2) は，ちょうどその場合を表しているともみなすこともできる．同図では，クラスタ間遅延のため I_c は I_p の次のサイクル (D) には実行できず，その次のサイクル (E) に実行される．したがって，同図のように I_c の *wakeup+select* に合わせて 2 サイクルをかけても，新たなペナルティを生じることはない．すなわち，*wakeup+select* を合わせて 1 サイクルで行う必要があるのは，同一のクラスタに割り振られた命令間に限られるのである．

クラスタ化プロセッサの持つこの性質は，分散型により有利に働く．分散型では，同一のクラスタに割り振られる命令は同一のサブウィンドウに格納されている．したがって，同一のサブウィンドウ内での *wakeup+select* は 1 サイクルで，別のサブウィンドウに跨る *wakeup+select* は 2 サイクルで実行するようにロジックを組めばよい．一方集中型では，この性質を利用することは簡単ではない．

分散型と行列方式を組み合わせる場合には，L-1 行列は各サブウィンドウに分散し，クラスタ間の *wakeup* は L-2 で行えばよい．

分散型と狭幅化

分散型と行列方式の組合わせは，狭幅化に対しても有利に働く．分散型における *dispatch* 時の各サブウィンドウに対する命令の振り分けがごくうまく行くと仮定すると，命令は互いに依存関係にある命令のグループに分けられて各サブウィンドウに置かれることになる．各サブウィンドウ内での依存する命令間の距離は平均して $1/NC$ 程度に削減されると期待できる．したがって，L-1 行列の幅を w/NC まで削減しても，同程度の L-1 ヒット率を期待することができる．

3.3　疑似分散型

前述したように，*wakeup+select* を合わせて 1 サイクルで行う必要があるのは同一のクラスタに割り振られた命令間に限られる．前節では，このクラスタ化プロセッサの性質は，集中型に応用することは簡単ではないと述べたが，以下ではこの性質を利用できる疑似分散型と呼ぶ方式について述べる．

図 **6 (c)** に，疑似分散型の模式図を示す．疑似分散型は，物理的には集中型の一種である．すなわち，ウィンドウは単一のロジックで構成され，命令はそこからすべてのクラスタに対して発行される．

その上で，ウィンドウのエントリをクラスタごとに領域分割し，各領域ごとに L-1 行列を用意する．すなわち，同じ領域内に置かれた 2 つの命令は 1 サイクルで *wakeup+select* できるが，異なる領域内に置かれた 2 つの命令間の *wakeup+select* には 2 サイクルを要する．*Dispatch* 時には，各領域に対して命令を振り分ける必要があるが，これには分散型と同じアルゴリズムを用いることができる．

Select 時には，発行可能な命令を各クラスタに割り振る必要がある．疑似分散型は，物理的には集中型の一種であり，どの領域の命令もすべてのクラスタにでも発行することができる．しかし，ある領域の命令は

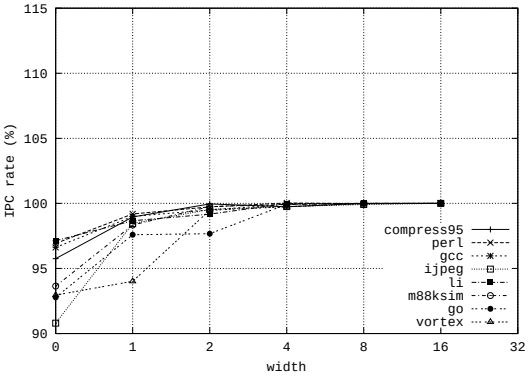

図 7　IPC: 集中型，×1

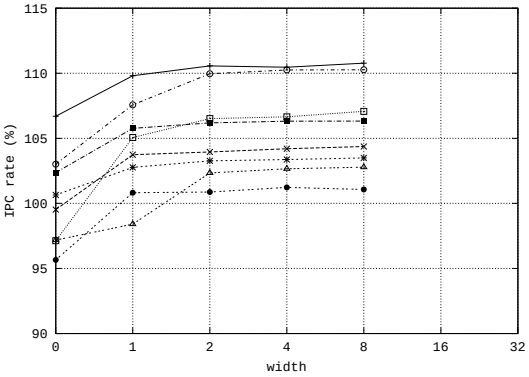

図 8　IPC: 疑似分散型，×1

それに対応するクラスタに優先的に割り振るとよいであろう．クラスタに空きがある場合には，別の領域の命令を発行することができる．

疑似分散型の，集中型，分散型に対するメリット，ディメリットは，以下のようにまとめられる．

メ　リット

疑似分散型では L-1 は分散型と同様分割した領域ごとに割り当てるため，遅延に関して集中型と比較して有利である．また疑似分散型と狭幅化について，前項の分散型と狭幅化の議論をそのまま用いることができ，集中型と比べてより小さい幅 w に対する耐性を持つ．

またクラスタへの命令割り振りに関して，疑似分散型は集中型に *Dispatch* 時の振り割りを加えたものといえる．*Dispatch* 時に依存関係を考慮して命令を割り振ることで，依存関係にある命令が同一クラスタで実行される頻度は集中型と比較して高くなり，クラスタ間遅延の発生頻度が低くなると考えられる．

また分散型では *Dispatch* 時に命令のクラスタへの割り振りを決定するため，クラスタに空きがある場合でも，対応クラスタ以外には命令を発行することができない．しかし疑似分散型の *Dispatch* 時のクラスタへの命令割り振りは優先度を求めるだけであり，最終的な決定は集中型と同様 *Select* 時に行われる．したがって，上記のような問題は起こらない．

ディメリット

メリットとして *Dispatch* 時の命令割り振りを付加を挙げたが，その精度によっては逆にペナルティとなることがある：集中型では L-1 を用い 1 サイクルで *wakeup+select* される命令の組み合わせが，疑似分散型では *Dispatch* 時の割り振りで異なる領域に置かれる可能性が生ずる．疑似分散型では異なる領域内に置かれた 2 つの命令間の *wakeup+select* には 2 サイクルを要するため，*Dispatch* 時の命令割り振りにより IPC が悪化する可能性がある．

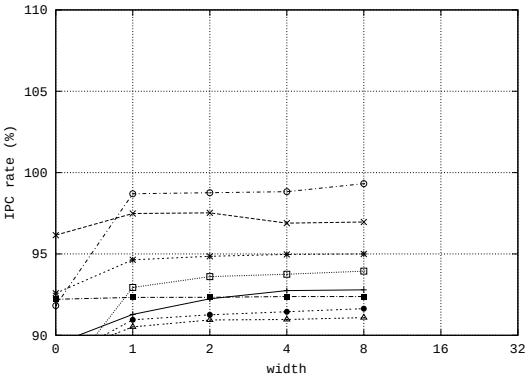

図 9　IPC: 分散型，×1

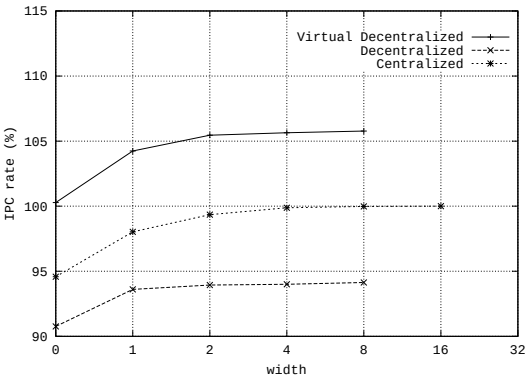

図 10　IPC: 各プログラムの相加平均，×1

4. 評　　価

本章では，SimpleScalar ツールセット (ver.2.0)[13] に対して，前章で述べたクラスタ化プロセッサの各方式に行列方式を適用したものを実装し，SPEC ベンチマークを用いて IPC の評価を行った．

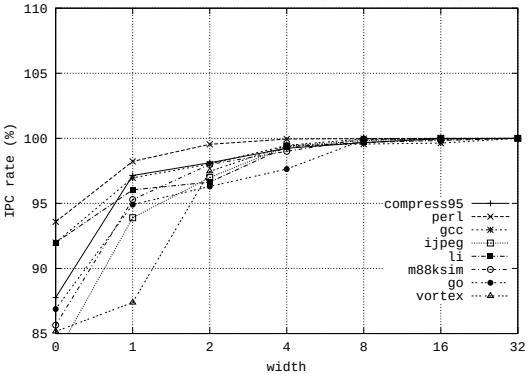

図 **11** IPC: 集中型，×2

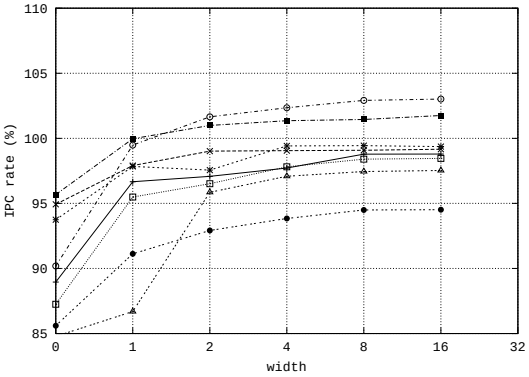

図 **12** IPC: 疑似分散型，×2

4.1 評価モデル

ベース・モデル

ベース・モデルは，INT，LS，FP 命令のそれぞれにサブウィンドウを持っており，その発行幅とウィンドウ・サイズは，それぞれ，2，16 である．全体の発行幅は $2 \times 3 = 6$ であるが，フェッチ幅は 4 である．命令/データ分離 1 次，及び，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，64B，2 サイクル，及び 8MB，64B，8 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 32 サイクル，後続ワードには 6 サイクル/ワードを要する．分岐予測には、履歴長 12，エントリ数 4K の gshare を用いた．このモデルを ×1 と呼ぶ．これに加えて，フェッチ幅，発行幅，ウィンドウ・サイズと EU 数を 2 倍に増やしたモデルを評価した．これを ×2 と呼ぶ．

クラスタ化

このモデルに対し，クラスタ数 $NC = 2$ のクラスタ化を施した．INT，LS，FP 命令のそれぞれの EU を 2 つのクラスタ，0 と 1 に分割する．各クラスタ内の INT，LSEU 間のバイパスは完全結合とし，クラスタ間遅延は 1 サイクルとした．

命令の割り振り

分散型，疑似分散型における，*dispatch* 時の命令の割り振りは，以下のアルゴリズムに基づく： 命令ウィンドウ内に依存元となる命令が存在する場合には，これと同じウィンドウに割り振る．依存しない命令，もしくはソース・オペランドが既に利用可能となっている命令は，優先的にクラスタ 0 に割り振る．

集中型における *select* 時の命令の割り振りは，より古い命令からクラスタ 0 の EU に優先的に割り振る，単純なアルゴリズムに基づく [8),9)]．

疑似分散型における *select* 時の命令の割り振りでは，ある領域の命令は対応するクラスタに高い優先順位を持つ．

4.2 評 価 結 果

各構成において，L-1 の幅 w を変化させた場合の IPC の変化を計測した．図 **7**，図 **8**，図 **9** にそれぞれ

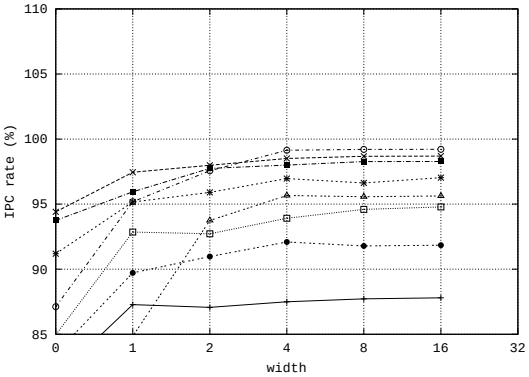

図 **13** IPC: 分散型，×2

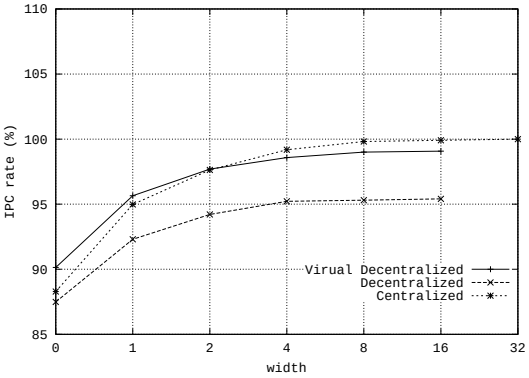

図 **14** IPC: 各プログラムの相加平均，×2

×1 をクラスタ化した集中型，分散型，疑似分散型の結果を示す．横軸は L-1 の幅 w である．縦軸は，集中型において L-1 の狭幅化を行わない場合に対する IPC の比率を示した．1 つの曲線は，1 つのプログラムに対応する．また図 **10** に，各構成ごとに，すべてのプログラムに対する IPC の比率の相加平均を示した．

また ×2 についても同様に，図 **11**，図 **12**，図 **13** にそれぞれ集中型，分散型，疑似分散型の結果を，図 **14** に構成ごとの IPC の比率の相加平均を示した．

×1 については図が示す通り，全ての L-1 の幅において疑似分散型，集中型，分散型の順に高い IPC を示している．狭幅化を行わない場合 (集中型:w = 32，疑似分散型/分散型:w = 16) について比較した場合も疑似分散型は他 2 構成よりも良好な結果を示している．この場合集中型より疑似分散型は平均で 6.0%良好，また分散型は 4.8% 悪化するとの結果となった．

また ×2 において狭幅化を行わない場合 (集中型:w = 32，疑似分散型/分散型:w = 16)，疑似分散型は分散型より高い性能を示している．しかし集中型と比較すると，li と m88ksim では集中型よりも高い性能を示しているものの，他のプログラムでは悪化する結果となった．また同じ幅で比較した場合については，図 14 が示す通り 2 以下では疑似分散型が勝っているがその差は 1% 未満と微かなものとなった．

×1 において疑似分散型は集中型よりも良い性能を示す一方，×2 では逆に殆どのプログラムで悪化する結果となった．これは評価に用いたベンチマークの IPC が低かったためであると考えられる．実際 ×2 のクラスタ化を行わないモデルで IPC を評価したところ，最大のもので 2 程度であった．したがって ×2 では 1 つのクラスタで十分処理でき，命令を割り振ることの意味が薄れていると考えられる．これにより 3.3 節で述べた，疑似分散型の *Dispatch* 時の命令割り振りのデメリットの方が大きく現れてしまい IPC が悪化したものと考えられる．

5. おわりに

本稿では，クラスタ化プロセッサに対して，我々の提案した依存行列型命令スケジューリング方式を適用について述べた．クラスタ化プロセッサのサブウィンドウが分散されている場合には，各ウィンドウに依存行列を分散して適用することで，*wakeup* の遅延を減少させることができる．一方，サブウィンドウを分散し *Dispatch* 時に命令を割り振ると，発行時までの状況変化に対応できないという問題に対し，擬似分散型を提案した．命令ウィンドウの集中/分散，に加え提案した疑似分散型に依存行列を適用したモデルに対し IPC の評価を行ったところ，ベースモデルをクラスタ化した場合では疑似分散型は集中型，分散型と比較して良好な結果を得た．しかし，EU 数を 2 倍に増やしたモデルでは，集中型と比較して疑似分散型の優位性はみられなかった．これは評価に用いた SPEC ベンチマークの命令並列性が低く，*Dispatch* 時の命令割り振りのデメリットの影響が大きくなったためと考えられる．今後より詳細な比較を行うため，メモリアクセスの投機的実行を実装し命令並列性を上げて評価していく予定である．

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) #13480083 による．

参考文献

1) 五島正裕ほか: 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 HPS 論文誌, Vol. 42, No. SIG 9(HPS 3) (2001).
2) Goshima, M. et al.: A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors, *MICRO-34* (2001).
3) 五島正裕ほか: 行列を用いた Out-of-Order スケジューリング方式の評価, *JSPP2002* (2002).
4) Palacharla, S. et al.: Quantifying the Complexity of Superscalar Processors, Technical report, Univ. of Wisconsin-Madison (1996).
5) Palacharla, S. et al.: Complexity-Effective Superscalar Processors, *ISCA24* (1997).
6) Kemp, G. A. and Franklin, M.: PEWs: A Decentralized Dynamic Scheduler for ILP Processing, *ICPP'96* (1996).
7) Farkas, K. I., Chow, P., Jouppi, N. P. and Vranesic, Z.: The Multicluster architecture: reducing cycle time through partitioning, *MICRO-30* (1997).
8) Keller, J.: The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, *Proc. 9th Annual Microprocessor Forum* (1996).
9) 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).
10) Yeager, K.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, No. 4 (1996).
11) Farrell, J. et al.: Issue logic for a 600-MHz out-of-order execution microprocessor, *IEEE J. of Solid-State Circuits*, Vol. 33, No. 5 (1998).
12) Henry, D. et al.: Circuits for Wide-Window Superscalar Processors, *ISCA27* (2000).
13) *SimpleScalar LLC: http://www.simplescalar.com/*.