計算機アーキテクチャ 149－26
（２００２．８．２３）

直接依存行列型スケジューリングを適用した クラスタ化スーパースケーラ・プロセッサの評価

小西将人[†] 西野賢悟[†] 五島正裕[†]
中島康彦[†] 森眞一郎[†] 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．この２つの方式の組み合わせとして集中型・分散型・擬似分散型と呼ぶ構成が考えられる．また wakeup 遅延の問題に対し dependence-based architecture が提案されており，そのクラスタ化についても研究がなされている．本稿では集中型・分散型・擬似分散型と，クラスタ化 dependence-based architecture について IPC の評価・比較を行った結果を報告する．

Evaluation of Clustered Superscalar Processor with Direct Dependence Matrix-Based Scheduling Scheme

Masahito Konishi,[†] Kengo Nishino,[†] Masahiro Goshima,[†]
Yasuhiko Nakashima,[†] Shin-ichiro Mori[†] 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. And the logic called wakeup will be critical too, so we have proposed **Direct Dependence Matrix Based Scheduling Scheme** which decreases delay of wakeup. The structures, called Centralized, Decentralized, and Virtual Decentralized can be considered as clustered processor with out scheme. On the other hand, to the wakeup problem, dependence-based architecture is proposed, and clustering this architecure is studied too. This paper reports the result of comparison IPC of Centralized, Decentralized, Virtual Decentralized, and clustered dependence-based architecture.

1. はじめに

将来スーパースケーラ・プロセッサのクロック速度を制限する構成要素として，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，命令発行幅 (***IW***: Issue Width) とウィンドウ・サイズ (***WS***: Window Size) 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)]．

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)~8)]．また *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)]．

また *Wakeup* 遅延の問題に対してこれまでに **dependence-based architecture** が提案されており，そのクラスタ化についても研究がなされている [5),8)]．

本稿は依存行列型スケジューリングとクラスタ化の組み合わせとして**集中型**，**分散型**，**疑似分散型** と呼ぶ３つのモデルと，クラスタ化 dependence-based architecture 対して小林らの提案した**最長パスに着目した命令発行機構**[8)] を実装し，IPC の評価・比較を行った結果を報告する．

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．

† 京都大学
Kyoto University

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．

依存行列とクラスタ化

命令ウィンドウの構成と行列の適用方法によって，集中型，分散型，疑似分散型と呼ぶ 3 つのモデルが考えられる．

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，ウィンドウ毎に依存行列を分散して適用す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 *dispatch* 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．

疑似分散型は，集中型と同様ウィンドウを単一のロジックで構成しそこからすべてのクラスタに命令が発行可能である．その上でウィンドウのエントリをクラスタ毎に分割し，分散型と同様分散した依存行列を適用する．

dependence-based architecture

dependence-based architecture は命令ウィンドウを少数の FIFO で構成し，依存関係にある命令を同じ FIFO に *dispatch* することで *wakeup* の対象となる命令を FIFO の先頭に制限する方式である．本稿で比較・評価する最長パスに着目した命令発行機構は，クラスタ化 dependence-based architecture に対して，クラスタ間遅延を軽減させることを目的に提案された機構である．

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述べる．4 章で最長パスに着目した命令発行機構の概要について述べ，5 章でこれらの IPC の評価・比較を行う．

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．

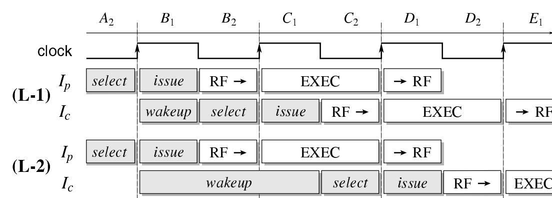

図 1 命令パイプラインにおける *wakeup*，*select*，*issue*

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 [9)]．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．

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．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

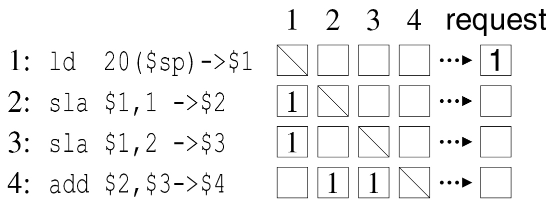

図 2 依存行列

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 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．依存行列は RAM に類似の回路によって実現できる.

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，多階層化について述べる.

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 [9),10)]．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.

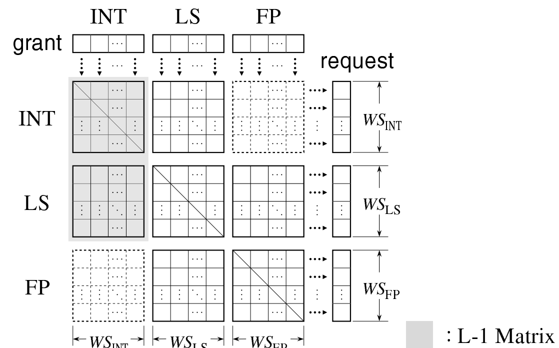

図 3 行列の分散化と多階層化

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 **3** に示す．R10000 では，INT，LS，FP の各サブウィンドウの発行幅とサイズはそれぞれ，$\boldsymbol{IW'} = IW/3 = 2$，$\boldsymbol{WS'} = WS/3 = 16$ である.

分散化の行列に対する効果は，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 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.

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

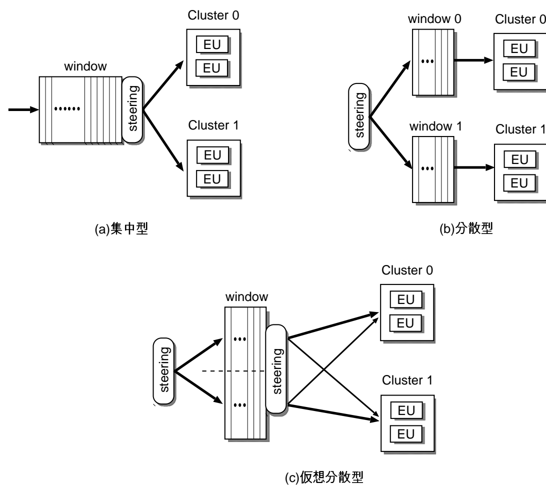

図 4 クラスタ化プロセッサの分類

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)~8)]．

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．図 **4**(a)(b) にそれぞれ、集中型、分散型の模式図を示す．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),6)]．

その一方で，ウィンドウを分散化することで *Wakeup* ロジックの遅延を削減することができる．これについては次項で詳述する．

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．

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．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．

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．このクラスタ化プロセッサの性質を集中型に応用した場合の**疑似分散型**と呼ぶ方式について述べる．

図 4 **(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．

3.4 比　　較

ここでは集中型，分散型，疑似分散型を比較し，その性質をまとめる．

***wakeup* 遅延**

分散型，疑似分散型では L-1 を分散することができ，wakeup 遅延に関して集中型より有利であると言える．

***dispatch* 時の命令割り振り**

分散型，疑似分散型で行う *dispatch* 時の割り振りは何らかのヒューリスティクスに基づいて行うが，依存関係を元に命令を割り振ることが普通である．これは依存関係にある命令をできるだけ同じクラスタに発行する試みで，これを行わない集中型よりクラスタ間遅延の発生頻度を下げる効用がある．ただし依存解析の精度によっては逆に集中型よりもクラスタ間遅延が頻発し IPC が悪化する可能性がある．

特に ILP が低く複数のクラスタで実行する必要がな

い場合，すなわち全て同じクラスタに割り振れば良いような場合には，*dispatch* 時の割り振りは不要な振り分けを行ってしまい逆に IPC に悪影響を及ぼす．

クラスタへの命令割り振りの効率

集中型では *select* フェーズで割り振りを行うため，クラスタの空きを考慮して命令を発行することが可能である．また疑似分散型でも，最終的には *select* フェーズで実行クラスタを決定するため集中型と同様に，空いているクラスタに発行することが可能である．

しかし分散型では，*dispatch* フェーズで割り振りを行うため，実行時のクラスタの利用状況を考慮して割り振ることができない．したがって，他のクラスタは空いているにも関わらず，対応するクラスタが busy 状態であるために命令が発行できないという状況が起こりうる．

4. 最長パスに着目した命令発行機構 [8]

本章では，次章で集中型・分散型・疑似分散型と評価・比較を行う小林らの提案した最長パスに着目した命令発行機構の概要について述べる．この機構はクラスタ化 dependence-based architecture に対しクラスタ間遅延の影響を軽減することを目的としており，命令ウィンドウ内の命令からなるデータフローグラフ (DFG) の最長パスを検出し，最長パス上の命令を同じクラスタに発行することを優先することで，クラスタ間遅延の影響を軽減することを目的としている．

DFG の表現

(1) DFG に存在する各合流ノードについて，そのノードに入るエッジのうち，実行終了時刻が最も遅いノードからのエッジを残し他を削除する．これにより DFG は木となる．

(2) 木を重なりのない部分パスに分解し，各部分パスに含まれるノードを FIFO 命令ウィンドウに格納する．部分パス間の結合情報は，別途表を用意し記憶し，最長パスを終点から始点まで辿るために利用する．

(3) ウィンドウ内の全命令中，実行終了時刻が最も遅い命令を最長パスの終点として別途記憶する．

***dispatch* フェーズでの動作**

上記 DFG を実現するために，*dispatch* される命令に対し，その実行終了時刻と，ソース・オペランドを生成する命令を求める必要がある．

このために各レジスタに対応するエントリを持つテーブルを用意し，そのレジスタを定義する先行命令へのポインタと，その命令の実行終了時刻を保持する．

dispatch される命令に対してこのテーブルを参照し，ソース・オペランドが揃う時刻と，これを生成する命令へのポインタを得る．ソースが複数存在する場合には実行終了時刻が最も遅い命令を選ぶ．

このようにして得られた依存元の命令の直後に，当該命令を挿入する．直後が空いていなければパスが分岐することを示しており，当該命令を空の FIFO の先頭に挿入しパス間の結合情報を保持する．このようにして，DFG が形成される．

また得られたソース・オペランドが揃う時刻に，当該命令のレイテンシを加えることで，実行終了時刻とする．当該命令がレジスタを定義する場合には，上記テーブルの対応レジスタのエントリに書き込む．

また実行終了時刻が，現在記憶している最長パスの終点の命令と比較して遅い場合には当該命令を最長パスの終点として記憶する．

***issue* フェーズでの動作**

このフェーズでは最長パスの先頭の命令を求め，この命令を優先して当該命令のソースを生成する命令と同じクラスタに発行する．

最長パスの先頭は，最長パスの終点を求め，そこから DFG を辿って始点を求める．終点は上記のように保持されており，そこからパス間の結合情報を用いて DFG を辿り先頭を求める．

5. 評　価

本章では，SimpleScalar ツールセット (ver.2.0)[11] に対して，3 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モデルと，4 章で述べた小林ら提案したの最長パスに着目した命令発行機構を実装し，SPEC ベンチマークを用いて IPC の評価を行った．

5.1 評価モデル

基準モデル

基準モデルは，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

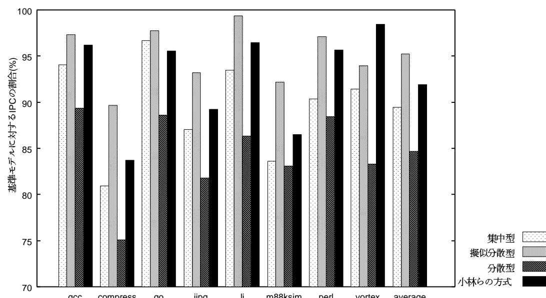

図 **5**　IPC: ×1

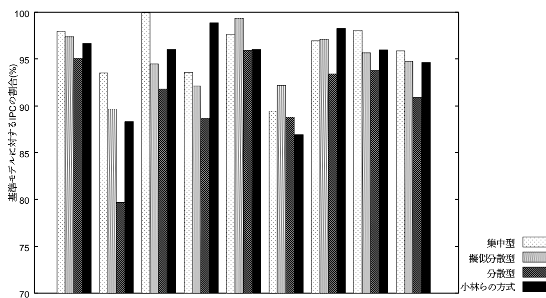

図 **6**　IPC: ×2

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 [7),8)]．

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．

小林らの方式の *select* 時の命令の割り振りは，最長パスの先頭に関しては 4 章で述べたようにそのオペランドを生成するクラスタとする．同サイクルに *select* された最長パスの先頭以外の命令に関しては，FIFO にクラスタの優先順位を持つ．

5.2　評 価 結 果

図 **5**，図 **6** にそれぞれ，×1，×2 をベースとするモデルの IPC の測定結果を示す．縦軸は基準モデルに対する IPC の割合を示している．4 本で組になっている棒グラフは左から集中型，擬似分散型，分散型，小林らの方式での IPC を示している．

×1 では疑似分散型，小林らの方式，集中型，分散型の順に高い IPC を示す傾向となった．平均では高い方から順に，基準モデルに対して 95.2% ，91.9% ，89.5% ，84.7% の IPC を示している．

また ×2 では，分散型が他のモデルと比較して IPC が低いという傾向が見られるが，集中型，疑似分散型，小林らの方式はプログラムによってかなり傾向が異なる．平均では集中型，疑似分散型，小林らの方式のそれぞれで，95.9% ，94.7% ，94.6% の IPC を示しており，あまり差がみられなかった．

6. お わ り に

本稿では，依存行列型命令スケジューリングとクラスタ化の組み合わせについて集中型・分散型・疑似分散型の 3 つの構成と，dependence-based architecture のクラスタ化プロセッサに対して小林らの提案した最長パスに着目した命令発行機構について，IPC の評価・比較を行った結果を報告した．×1 では疑似分散型，小林らの方式，集中型，分散型の順に高い IPC を示す結果となった．一方で資源を 2 倍に増やした ×2 の場合では，分散型が他のモデルと比較して IPC が低い傾向が見られたが，集中型，疑似分散型，小林らの方式についてはプログラムによって傾向が異なる結果となった．平均ではこの 3 つのモデルに関しては，1% 程度の差しかみられなかった．

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) Farkas, K. I., Chow, P., Jouppi, N. P. and Vranesic, Z.: The Multicluster architecture: reducing cycle time through partitioning, *MICRO-30* (1997).
7) Keller, J.: The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, *Proc. 9th Annual Microprocessor Forum* (1996).
8) 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).
9) Yeager, K.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, No. 4 (1996).
10) 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).
11) *SimpleScalar LLC: http://www.simplescalar.com/*.