特別研究報告書

ロード命令の投機を行う
クラスタ化スーパースケーラ・プロセッサの
性能評価

指導教官　　富田 眞治 教授

京都大学工学部情報学科

芦川 司

平成15年2月3日

ロード命令の投機を行う クラスタ化スーパースケーラ・プロセッサの 性能評価

芦川 司

内容梗概

スーパースケーラ・プロセッサの構成要素のうち，将来クロック速度を制限するものとして考えられているのが，バイパスロジックと *wakeup* ロジックの遅延である．

バイパスロジックとは，ALU 間でのデータ転送を行うロジックのことである．このロジックは，配線遅延に支配されるため，LSI の微細化の恩恵を受けにくい．よって，バイパスロジックは，LSI の微細化に伴って，一層クリティカルになっていくと予想される．

そこで，このバイパス遅延の解決策として，演算器のクラスタ化が研究されている．クラスタ化とは，一般のスーパースケーラ・プロセッサのデータパスを，独立した複数のデータパスで構成することをいう．

一方，*wakeup* ロジックとは，動的命令スケジューリングのために，命令の実行に必要なデータの有効性を追跡するロジックのことである．この *wakeup* ロジックもバイパスロジックと同様に，配線遅延によって支配されるために，LSI の微細化の恩恵を受けにくい．また，他の構成要素とは異なり，このロジックは複数のパイプライン・ステージに分割することができない．よって，*wakeup* ロジックも同様に，LSI の微細化，パイプラインの深化に伴って，一層クリティカルになっていくと予想される．

この *wakeup* ロジックを高速化させる手法の 1 つとして，我々の提案した直接依存行列型スケジューリング方式がある．直接依存行列型スケジューリングは，依存行列を縮小化することで，*wakeup* ロジックの遅延の減少を実現している．この場合，依存行列の縮小化による *wakeup* ロジックの遅延の減少は，IPC とトレード・オフの関係にあり，IPC の悪化を許容範囲に抑えつつ，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って決まる．

本稿では，このクラスタ化スーパースケーラ・プロセッサと，直接依存行列型スケジューリング方式を組み合わせた機構について議論する．クラスタ化スーパースケーラ・プロセッサは，命令ウィンドウの構成法の違いにより，集中型

と分散型に分類できる．命令ウィンドウが分散されている場合，各サブウィンドウのエントリ数は軽減され，更に $wakeup$ の遅延も減少すると考えられる．しかし一方で，その構成の違いから，集中型は分散型と比較してIPCが高くなると考えられる．また，両方式の利点を併せ持つ，疑似分散型という方式も提案されている．

SPECベンチマークを用い，クラスタ数: 2，各クラスタ内EU数: 2のモデルでIPCを評価したところ，最大のものでも2程度であった．したがって，このモデルでは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合いが薄れていると考えられる．そこで，もう1つのクラスタの使用頻度を高め，プロセッサのクラスタ化について考察するために，ロード命令の投機的実行によりIPCを向上させた．

ロード命令の投機的実行とは，あるストア命令に続くロード命令の，メモリ参照の曖昧性の問題を緩和するものとして提案されている方式である．具体的には，まず，先行するストア命令と後続するロード命令のメモリアドレスの一致/不一致を予測する．その上で，不一致と予測されたものに対して，その後続ロード命令を先行ストア命令よりも先に実行する．つまり，ロード命令の投機を行うと，命令並列性が上がりIPCも向上する．このロード命令投機実行を，依存行列方式を持つクラスタ化プロセッサに適用する．その上で，命令ウィンドウの構成法によって分類される集中型，分散型，疑似分散型の3方式について議論する．

評価データによると，ロード非投機に対するロード投機のIPCの伸び率は，疑似分散型が最も高かったが，IPCの絶対値に関しては，集中型が最も大きかった．つまり，分散型/疑似分散型よりも集中型の方が高い性能を示す結果となった．

Evaluation of clustered superscalar processors with speculation of load instructions

TSUKASA ASHIKAWA

Abstract

The delay of a bypass logic and a *wakeup* logic among the components of a superscalar processor are thought as what will restrict the clock speed in the future.

The bypass logic transmits data among ALUs. This logic is governed by the wire delay, so it can not get the benefit of the miniaturization of LSI. Therefore, the bypass logic will be much more critical with the miniaturization of LSI in the future.

The research of a clustered superscalar processor is being done as a measure against the delay time of bypass logic. The clustered means that the data passes of a general superscalar processor are made up of independent plural data passes.

On the other hand, the *wakeup* logic manages availability of the data for dynamic instruction scheduling. Like the bypass logic, this *wakeup* logic is dominated by the wire delay, so it can not receive the benefit of the miniaturization of LSI easily. Moreover, unlike other components of a superscalar processor, it can not divide into two or more pipeline stages. Therefore, it is expected that the *wakeup* logic will become much more critical with the miniaturization of LSI and the deepening of pipelines from now on.

We have proposed Direct Dependence Matrix Based Scheduling Scheme which decreases the delay time of *wakeup* logic. It can decrease the delay of *wakeup* logic with a dependence matrix reduction. In this case, the delay of *wakeup* logic 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.

This paper describes the case where our scheduling scheme based on a direct dependence matrix is applied to clustered superscalar processors. The clustered superscalar processor can be classified into two types by the difference in instruction window structure. They are called the centralized type and the

decentralized type. In the decentralized type, the number of entries to each subwindow is distributed and the delay time of *wakeup* also becomes shorter. On the other hand, the centralized type will get higher IPC than the decentralized type. And the virtual decentralized type is proposed, which has both types' good points.

The IPC was at most 2.0 instructions/cycle in 2 clusters model which has 2 EUs each. In other words, this superscalar processor can deal with the process by only one cluster, so it don't have to think the dependence between instructions. In this paper, to evaluate the clustered superscalar processor, road instructions were excecuted speculatively.

A speculation of load instructions relieves the problem of ambiguous reference to memory for load instructions following a store instruction. Concretely, first, it is predicted whether a memory address of a following load instruction coincides with that of a preceding store instruction or not. In case both the memory addresses are not the same, the following load instruction will be executed before the preceding store instruction. Therefore, the instruction parallel will improve and IPC will rise if load instructions are executed speculatively. We applied the load speculation to the clustered superscalar processor with the direct dependence matrix. And we discuss the three types in the clustered superscalar processor.

According to the results, the IPC rate of increase is the highest in the virtual decentralized type, but the absolute value of IPC is the largest in the centralized type. After all, the centralized type has the highest performance in the three types.

ロード命令の投機を行う
クラスタ化スーパースケーラ・プロセッサの
性能評価

目次

第1章　はじめに

スーパースケーラ・プロセッサの性能は，IPC(Instructions per Cycle) とクロック速度の積で表すことができる．故に，スーパースケーラ・プロセッサの性能を上げるためには，IPC 及びクロック速度を向上させればよい．

スーパースケーラ・プロセッサの IPC を最も直接的に向上させるための方法は，命令発行幅 (***IW***: Issue Width) 及びウィンドウ・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 の微細化に伴って，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，いたずらにこれらを増やせば，クロック速度の低下を招き，かえって全体の性能を悪化させることになる．

スーパースケーラ・プロセッサの構成要素のうち，将来クロック速度を制限するものとして予測されているのが，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である．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．また配線遅延は，LSI の微細化に伴って，全体の遅延を支配していくと考えられている [1][2][3]．以下に，これらのロジックの説明と，これらの遅延の増大に対する解決法を紹介する．

オペランド・バイパスは，ALU 間でのデータの転送を行う．また，このロジックは配線遅延によって支配されるために，LSI の微細化の恩恵を受けにくく，今後 LSI の微細化に伴って一層深刻な影響を受けると予想されている．

そこで，バイパス遅延の増大に対する解決策として，DEC Alpha 21264 で採用された，演算器の**クラスタ化**がある [3][4][5][6][7]．クラスタ化とは，一般のスーパースケーラ・プロセッサのデータパスを，独立した複数のデータパスで構成することをいう．別のクラスタに振り分けられた 2 つの命令は，利用すべきバイパスが存在しないため，続けて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，クラスタ化するメリットとして挙げられるのは，バイパスの配線長が数分の 1 に短縮されることである．こ

れにより，バイパスロジックの遅延を大幅に短縮することができる．

一方 *wakeup* ロジックとは，動的命令スケジューリングのために，命令の発行に必要なデータの有効性を追跡するロジックのことである．このロジックも配線遅延によって支配されるために，LSI の微細化の恩恵を受けにくい．また *wakeup* ロジックは，他の多くの構成要素とは異なり，複数のパイプライン・ステージに分割することができない．よって，*wakeup* の遅延は，*IW*, *WS* の増大，LSI の微細化，パイプラインの深化に伴って，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主要因の 1 つとなると予測されるのである．

この *wakeup* を高速化させる方法の 1 つとして，**直接依存行列型スケジューリング方式**がある [1]. 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 $\boldsymbol{w}$ 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，続けて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，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．

本研究では，直接依存行列型スケジューリングをクラスタ化スーパースケーラ・プロセッサに実装した機構において議論する．クラスタ化スーパースケーラ・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 つに分類できる．命令ウィンドウが分散されている場合，各ウィンドウのエントリ数も軽減でき，更に *wakeup* の遅延も減少すると考えられる．しかし一方で，その構成の違いから，集中型は分散型に比べて IPC が高くなると考えられる．また，これら両方の利点を併せ持つ**疑似分散型**という方式も提案されている [8].

第 6 章の評価結果でも述べるが，クラスタ数が 2 個で，各クラスタ内の EU 数が 1 個のモデル ($\times 1$ と呼ぶ) では，疑似分散型の IPC は他 2 構成よりも良好な結果を示した．しかし，クラスタ数が 2 個で，各クラスタ内の EU 数が 2 個のモデル ($\times 2$ と呼ぶ) 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，EU 数を 2 倍にしたことで，命令ウィンドウへの命令割り振りの意味合いが薄れ，命令間の依存関係を考慮する必要性が低くなったためと考えられる．

そこで，この ×2 モデルに対して，ロード命令の投機的実行を適用し，命令並列性を上げ全体の IPC を向上させた．その上で，集中型，分散型，疑似分散型の，より詳細な比較を行った．ロード命令の投機的実行については，第 5 章で詳しく述べる．

以下，第 2 章では，クラスタ化スーパースケーラ・プロセッサについて紹介し，第 3 章では，直接依存行列型スケジューリング方式について述べ，その依存行列の高速化手法についても詳述する．第 4 章では，クラスタ化スーパースケーラ・プロセッサに，直接依存行列型スケジューリング方式を導入することについて述べ，集中型，分散型，疑似分散型の 3 方式についても説明する．第 5 章では，ロード命令の投機的実行の背景と概要を説明し，第 4 章のクラスタ化スーパースケーラ・プロセッサに対して，ロード命令の投機を実行した動機について述べる．第 6 章では，第 4 章のクラスタ化スーパースケーラ・プロセッサにおいて，第 5 章で述べたロード命令の投機的実行を施した場合の評価を行う．そして最後の第 7 章では本稿のまとめを行う．

第2章　クラスタ化スーパースケーラ・プロセッサ

この章では，クラスタ化スーパースケーラ・プロセッサについて述べる．一般に，IW, WS が増えるにつれてバイパスロジックの配線は長くなり，配線の長さが長くなるにつれ配線遅延は大きくなる．故に，IW が増大するにつれて，バイパスロジックの遅延は今後ますますクリティカルになると予想される．

次に，このバイパスロジック遅延の解決方法として研究されている，クラスタ化スーパースケーラ・プロセッサについて述べる．

2.1　クラスタ化スーパースケーラ・プロセッサの概要

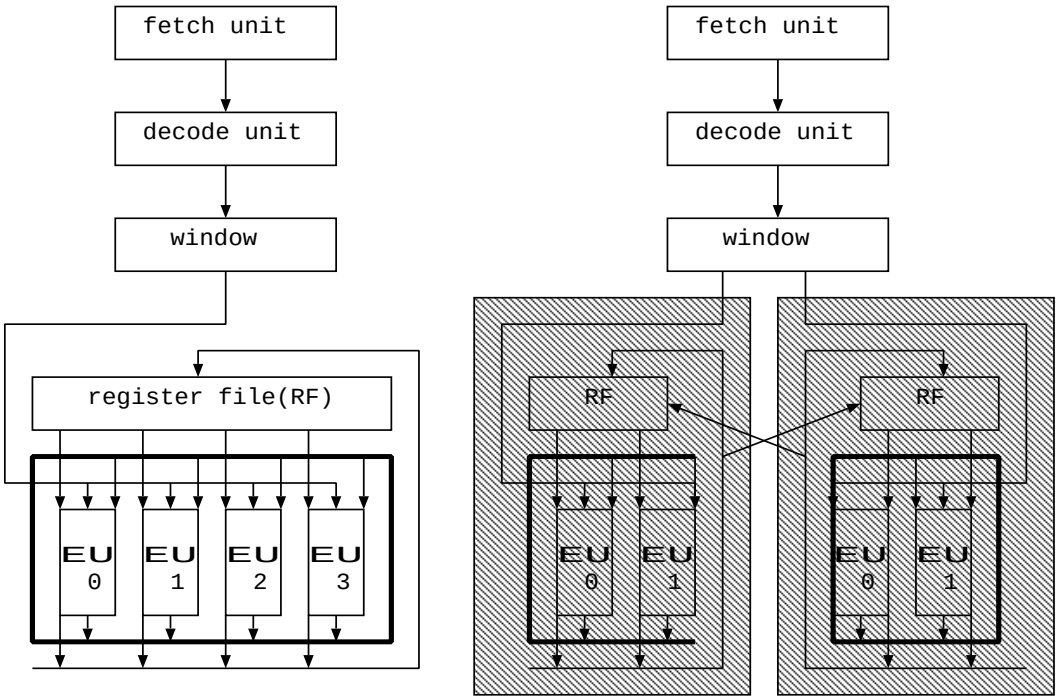

図1: クラスタ化

図1 にクラスタ化の例を示す．図1 (a) は4命令同時発行の従来のスーパースケーラ・プロセッサを，図1 (b) は2個のクラスタで分割したクラスタ化スーパースケーラ・プロセッサを示している．

図1において，フェッチ·ユニット，デコード·ユニット，命令ウィンドウ，実行ユニット (**EU**: Execution Unit)，レジスタ·ファイル (**RF**: Register File) の順に，命令が伝わる．また，図1 (b) の網掛け部分がクラスタ化していることを示す．そして，図1の太線はローカルパスを示しており，ローカルパスはそれぞれのクラスタ内でのみバイパスされている．

一般に，命令発行幅がNのスーパースケーラ·プロセッサを，n個のクラスタに分割すると，各クラスタ内の実行ユニット数はN/n個となり，クラスタ化していない場合の1/n倍となる．一方で，クラスタ化スーパースケーラ·プロセッサ全体の命令発行幅は，各クラスタの命令実行幅を足し合わせたNであり，クラスタ化する前のそれと変わらない．

2.2 クラスタ化スーパースケーラ·プロセッサの特徴

そもそもクラスタ化とは，具体的には，クラスタ間にまたがるオペランド·バイパスを省略することである．故に，依存する2つの命令が，別のクラスタに割り振られると，利用すべきバイパスが存在しないため，その2つの命令は続けて実行することができない．そして，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．実際に依存関係にある2つの命令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$\boldsymbol{NC}$個のクラスタに分割すると，バイパスの配線長は$1/NC$程度にまで短縮され，その遅延を大幅に短縮することができる [3][4][5][6][7].

今後IWが増えるにつれて，クリティカルとなっていくバイパスロジックの遅延を低減する上で，このクラスタ化の技術はますます重要となっていくと予想される．

第3章　直接依存行列型スケジューリング方式

この章では，まず動的命令スケジューリング方式について述べ，その中で *wakeup+select* ロジックの遅延がクリティカルになる理由を述べる．その後，そのロジック遅延の解決法の１つとして，我々が提案した直接依存行列型スケジューリング方式について述べる．

3.1　動的命令スケジューリング方式

Out-of-Order スーパースケーラにおける動的命令スケジューリングとは，端的にいえば，左/右のソースオペランドに対応するバッファのエントリに，データが揃っている命令を選択することである．したがって，原理的には，左/右のデータが使用可能かどうかを表すフラグ **rdyL/R** のテーブルが存在し，スケジューリングにおける中心的な役割を果たす．

動的命令スケジューリングのフェーズ

動的命令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 *Ip* の結果を命令 *Ic* が消費するものとする：

1. ***rename***：出力依存や逆依存を動的に解消することを目的に，論理レジスタ番号を，個々のデータに割り当てられる ID に付け替える処理を ***rename*** と呼ぶ．*rename* は，各命令がフェッチされた後，命令ウィンドウへ格納される前に実行される．
2. ***dispatch***：その後，命令は命令ウィンドウに格納される．*Ip* がまだ実行されていない場合，*Ic* は *Ip* 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 *Ip* の発行が許諾されると，*Ic* は『起こされ』，自らの発行を要求する．
4. ***select***：rdyL/R がともにセットされている命令が，演算器に対して発行することができる命令である．rdyL/R がともにセットされている最大 *WS* 個の命令の中から，その内の *IW* 個を選択し，それらに発行許可を与える処理を ***select*** と呼ぶ．
5. ***issue***：*select* によって選択された命令は，実際に演算器に対して発行される．***issue*** は，具体的には，命令ウィンドウを構成する RAM から選択された命令の情報を読み出す処理である．

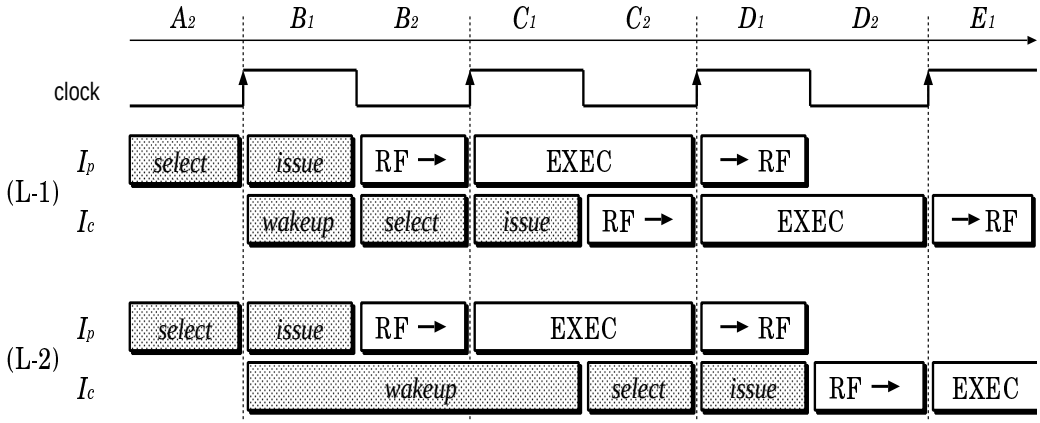

図2: 命令パイプライン

図2 (L-1) に，*wakeup*，*select*，*issue* の命令パイプラインでの位置付け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 はそれぞれ物理レジスタ・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の場合，すなわち，実行のレイテンシが 1 である命令 Ip の次のサイクルで Ic が実行される場合を示している．Ip が生成したデータは，オペランド・バイパスを通って Ic の実行に使用される．

遅延とパイプライン化

図 2 **(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，$(A_2)Ip$ に対する *select* が終わった後にしか，$(B_1)Ic$ に対する *wakeup* は開始できない．その結果，Ic の発行は 1 サイクル遅れ，Ip と Ic は続けて実行することができない．

同図では，Ip 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 Ic の実行に使用される．すなわち，*wakeup*+*select* に 1 サイクルより多くのサイクル数を割り当てることは，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．この場合，IPC の悪化は 5~15%程度にもなることが分かっており [1]，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 サイクルで実行されなければならないといえる．

現在，スーパースケーラのパイプラインはますます深くなる傾向にある．

wakeup と *select* は，一般にはパイプライン化することができないので，パイプラインが深くなると相対的にクリティカルになる．

このうち，*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ていくと予測されている．一方，*wakeup* の遅延は，主に RAM 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．このような理由から，*wakeup* は，LSI の微細化，パイプラインの深化に伴って，一層クリティカルになっていくと予測されているのである．次節では，依存行列を用いた *wakeup* について詳述する．

3.2 直接依存行列型スケジューリング方式

我々が提案した手法では，命令ウィンドウ中の各命令間のデータ依存関係を直接的に表す依存行列が，スケジューリングにおける中心的な役割を果たす．以下では，この依存行列の概要と，行列アクセスの高速化手法について説明する．

3.2.1 依存行列の概要

図 3 に依存行列の概念図を示す．行列は，rdyL/R に各 1 つずつ用意された $WS \times WS$ の行列であり，対角要素は使用しない．命令ウィンドウの p 番エントリに格納された命令 Ip の実行結果を，c 番の命令 Ic が消費する場合，p 行 c 列の要素は “1”，そうでなければ “0” とする．*wakeup* においては，各列で 1 である行に対応するすべての命令の発行が許諾されれば，その列に対応する命令は実行可能であり，rdyL/R フラグをセットしてその命令を要求すればよい．

図 3 に示す例では，連続する 4 つの命令が ID $= 1$ のエントリから順に格納された場合を表している．この場合，例えば，ID $= 1$ の命令が生産する $1 を ID $= 2, 3$ の命令が左オペランドとして消費するので，左の行列の 1 行目では 2, 3 列の要素が “1” となる．その他の要素も同様に求められる．

3.2.2 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，上記の順に適用するという流れで各手法について説明する．

行列の分散化

実際のプロセッサでは，整数 (**INT**)，ロード / ストア (**LS**)，浮動小数点 (**FP**) 命令の系統ごとに命令ウィンドウを分散化している．このような分散化は，わ

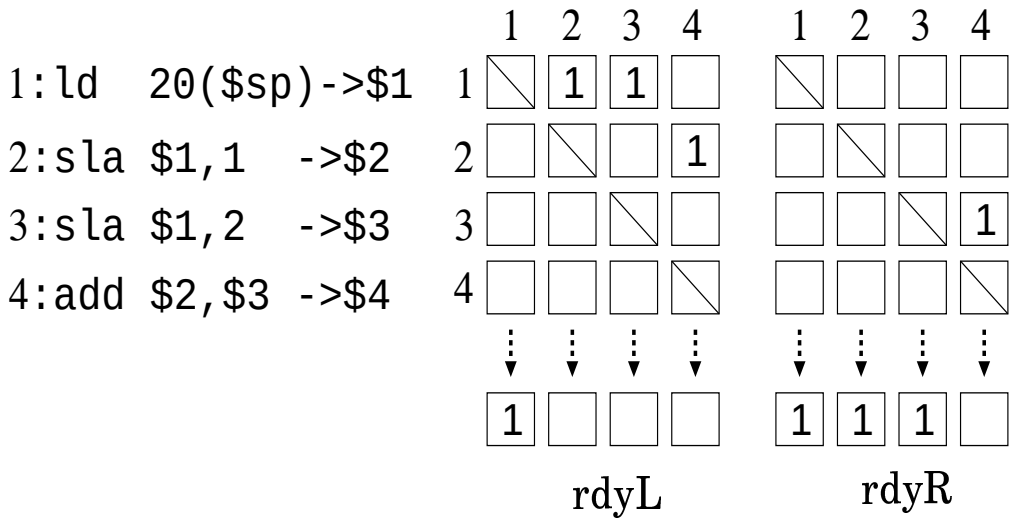

図 3: 依存行列の概念図

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．MIPS R10000 と同様の分散化を施した場合の様子を図 **4** 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，$\boldsymbol{IW}_q = IW/3 = 2$, $\boldsymbol{WS}_q = WS/3 = 16$ 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_q 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_q 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，次項で述べる行列の多階層化が可能になる．

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．この部分を取り出し，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 2 (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

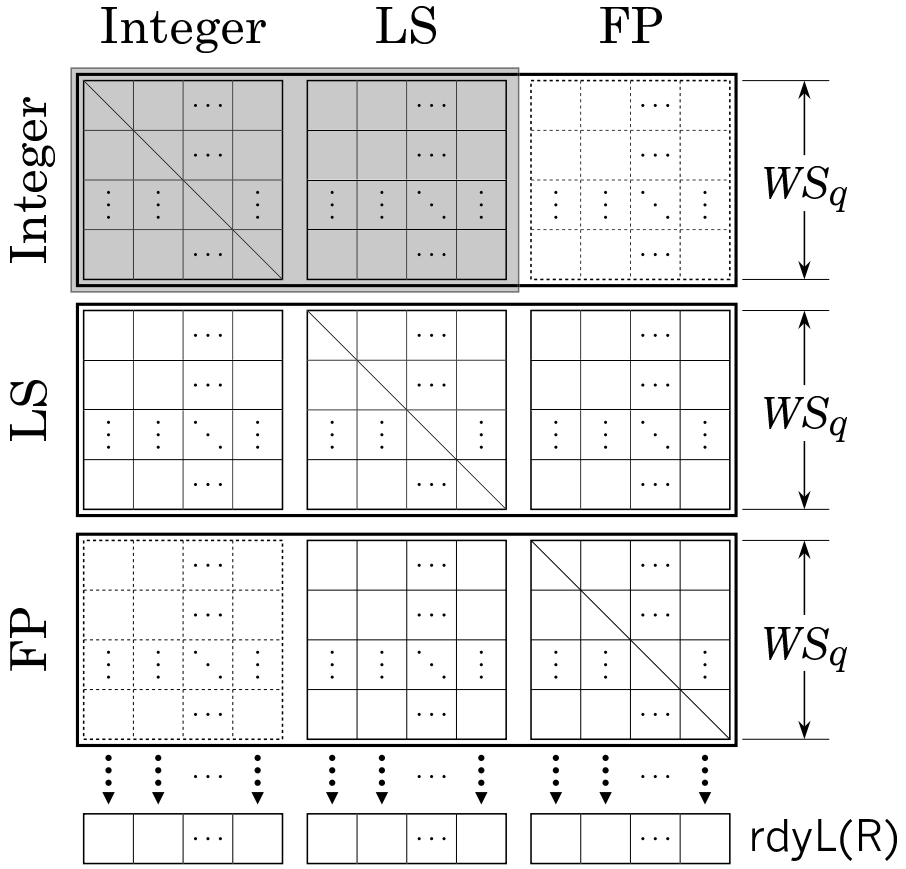

図 4: 直接依存行列の分散化と多階層化

縮される．更に，L-1 に対しては次項で述べる狭幅化を適用することができる．

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 を縮小し，更にその遅延を短縮することを考える．すなわち，各命令キューにおいて，自命令の後の $\boldsymbol{w}(\mathbf{1} \leq \boldsymbol{w} \leq \boldsymbol{WS}_q - \mathbf{1})$ 個の命令に対するビットだけを L-1 に残し，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，$WS_q = 8$, $w = 4$ の場合の L-1 の縮小の様子を示す．この図は，INT から INT の部分のみであるが，INT から LS の部分も同様である．左は元々の，右が縮小後の L-1 である．元の L-1 行列から削除されるセルは，左図中で薄く示してある．必要なセルを矩形領域に集めるにはいくつかの方法が考えられるが，よりクリティカルであるビット線 $\overline{\mathrm{rdyL/R}}$ を短縮することを優先して，同

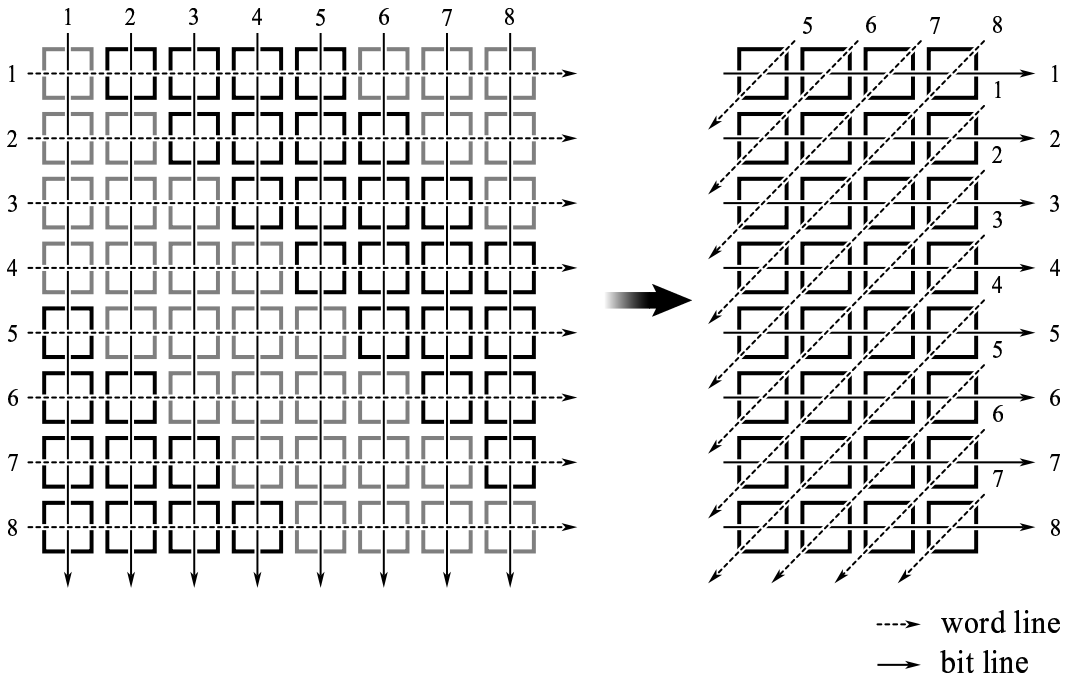

図5: L-1 行列の狭幅化

図右のようにするとよい. w を L-1 の**幅**と呼ぶことにし，この技術を **L-1 行列の狭幅化**という.

図5右から明らかなように，狭幅化された L-1 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. ただし，IPC の悪化を許容範囲に抑えつつ，幅 w をどこまで小さくできるかは，WS の大きさによる.

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. $w \geq \min(WS_q/4,64)$ 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，その IPC の低下は 1~2% に抑えられることが分かっている [1].

第4章　クラスタ化と行列方式の組み合わせ

この章では，第2章，第3章でそれぞれ述べた，クラスタ化スーパースケーラ・プロセッサに，直接依存行列型スケジューリング方式を組み合わせる方法について述べる．

4.1　クラスタ化プロセッサの分類

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6 (a),(b) はそれぞれ集中型，分散型を表したものである．この図から分かるように，集中型では単一のロジックとして命令ウィンドウを構成し，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として命令ウィンドウを構成している．そして，集中型では命令は単一のウィンドウから全てのクラスタに対して発行されるのに対し，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．

また集中型と分散型とでは，命令をクラスタに割り振るタイミングが異なる．つまり，集中型では命令の割り振りは *select* フェーズにて行われるが，分散型では何らかのヒューリスティクスに基づいて，*dispatch* フェーズにて行う必要がある．IPC はその精度に依存する [3][4][5]．

しかし，その一方で，分散型は命令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*** このロジックでは，WS 個の命令から IW 個を選択する必要があるものが，n 個のクラスタに分割すると 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 [10]．
- ***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，これらの処理は *wakeup*, *select* とは異なり，パイプライン化可能であるので，*wakeup*+*select* と比較すると，その影響は決定的ではない．

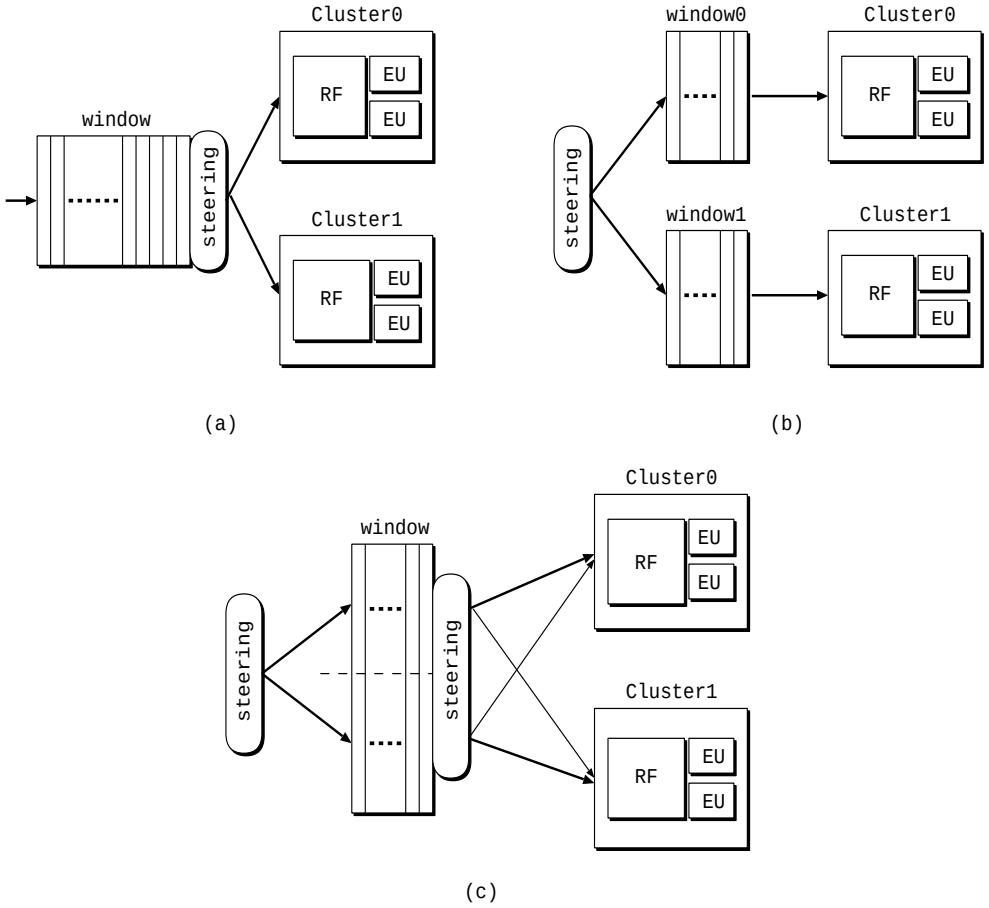

図6: クラスタ化プロセッサの分類

4.2 分散型と行列方式の組み合わせ

依存する2つの命令 Ip と Ic 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，Ip と Ic は続けて実行することはできない．したがって，Ic 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2 (L-2) は，ちょうどその場合を表しているとみなすこともできる．このとき，同図のように Ic の $wakeup+select$ に合わせて2サイクルをかけても，新たなペナルティが生じることはない．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またがる $wakeup+select$ 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のは容易ではない．

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み合わせは，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いくと仮定すると，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．

4.3 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**という方式について説明する．

図 6 (c) は，疑似分散型の模式図である．疑似分散型は，物理的には集中型の一種である．すなわち，命令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，命令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．そして，$select$ 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り，どの領域の命令もすべてのクラスタに対して発行できる．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ればよく，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．

4.4 各方式の特徴

以下に，疑似分散型の，集中型，分散型に対するメリット・デメリットをまとめる．

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，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はできない．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．従って，上記のような問題は起こらない．

デメリット

メリットとして *dispatch* 時の命令割り振りの付加を挙げたが，その精度によっては逆にペナルティとなることがある．つまり，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じる．すると，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，IPC が悪化する可能性がある．

第5章　ロード命令の投機的実行

この章ではまず，ロード命令の投機的実行の背景と概要を述べ，ロード命令の投機的実行の具体例も説明する．そして，前章で述べたクラスタ化スーパースケーラ・プロセッサにおける3方式に対して，ロード命令の投機的実行を行った動機について述べる．

5.1　ロード命令の投機的実行の背景と概要

スーパースケーラ・プロセッサの性能向上を妨げる要因の1つに，命令間のデータ依存関係の問題がある．データ依存関係には，2つのタイプが存在する．1つはレジスタを介した依存関係であり，もう1つはメモリを介した依存関係である．この内，レジスタを介した依存関係は，デコード時に命令内のオペランド・フィールドを評価することで検出することが可能である．一方，メモリを介した依存関係は，同一アドレスを参照するメモリアドレス命令間の依存となるが，参照アドレスが実行時にならないと決定されないため，命令スケジューリング時に検出することはできない．

したがって，データ依存関係を満たすことを保証するには，あるストア命令に続くロード命令は，先行するストア命令が完了しなければ実行することができない，という制約が必要である．これは，異なるアドレスに対する後続ロード命令でさえも，先行ストア命令に依存することになる．これが曖昧なメモリ参照の問題であり，命令レベル並列性(ILP:instruction level parallelism)を引き出す際の大きな妨げとなっている．

このメモリ参照の曖昧性の問題を緩和するために，ロード命令の投機的実行という方式が提案されている．これは，先行ストア命令と後続ロード命令のアドレス一致/不一致を予測し，不一致と予測されたロード命令を投機的に実行する方式である．**図7**を用いて具体的に説明する．

以下では説明を簡略化するため，ロード命令とストア命令しかないものとする．図7では，ある時刻における，4つのロード/ストア演算器の使用状況，また，命令列の“Ready”と“Not Ready”はそれぞれ，ロード/ストア命令のソース・オペランドが揃っているか否かを表している．この時刻では，$1のLOADはLS演算器で実行中である．そして，$2のSTORE，$5のLOADはソース・オペランドが揃っていないため，“Not Ready”状態である．一方，$3，$4の

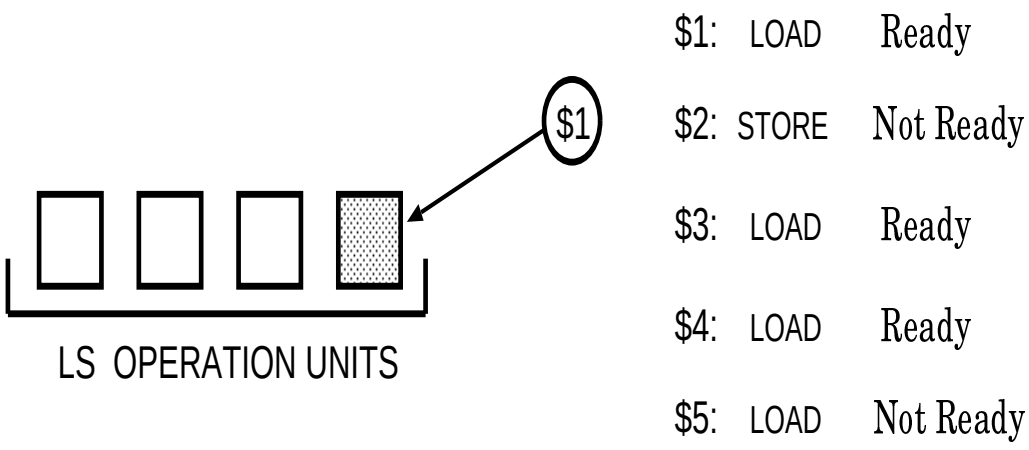

図 7: ロード命令の投機的実行

LOAD は，既にソース・オペランドが抽出されているため“Ready”状態にあり，実行待ちである．

ここでまず，ロードの投機を行わない場合を考える．上で述べた，あるストア命令に続くロード命令は，先行するストア命令が完了しなければ実行することができない，という制約より，“Ready”状態にある$3，$4 の LOAD は，“Not Ready”状態にある先行する STORE がまだ実行完了していないので，LS 演算器に 3 つの空きがあるにもかかわらず，実行開始ができない．

しかし，ロードの投機を適用する場合，先行ストア命令$2:STORE と，後続ロード命令$3:LOAD，$4:LOAD とのアドレス一致/不一致を予測し，不一致と予測されたなら，$3，$4 の LOAD は実行できる．そして時間が進行して，$2 の STORE が“Ready”状態になり，メモリアドレス値が分かると，すでに実行された LOAD のメモリアドレス値と一致/ 不一致を確かめる．もし，すべて不一致なら何も問題はなく，一致しているものがあれば，再度その LOAD を実行する．

これまでは，ロード命令とストア命令しかないものとして話を進めてきたが，実際は他にも命令があり，もし先行ストア命令と後続ロード命令のメモリアドレスが一致していれば，そのロード命令とそれによって実行可能となる命令を再実行する必要がある．しかし，先行ストア命令と後続ロード命令のメモリアドレスが一致していることは少なく，先行ストア命令よりも先に実行した後続ロード命令を再実行する頻度は少ない．

5.2 動機

クラスタ化スーパースケーラ・プロセッサにおいて，クラスタ数が2個で，各クラスタ内のEU数が2つのモデルに対して，クラスタ化を施していないベース・モデルを考える．この場合，ベース・モデルのEU数は，$2 \times 2 = 4$個となる．前にも述べたが，クラスタ化とは，クラスタ間のオペランド・バイパスを省略することであり，これにより，僅かなIPCを犠牲にしてバイパスの遅延を大幅に短縮することができる．つまり，クラスタ化をしていないモデルのIPCは，クラスタ化をしているモデルのIPCよりも多少高くなる．

次章の表1より，このクラスタ化をしていないベース・モデルに対して，SPECベンチマークを用いてIPCの評価を行ったところ，最大のものでも約2.3であった．このベース・モデルに対してクラスタ化を施した場合(クラスタ数：2，EU数：2)は，上記のことを踏まえると，IPCは平均して2ぐらいになると容易に予想できる．そして，この場合，分散型，疑似分散型よりも，集中型のIPCの方が高くなると考えられる．以下にその理由を述べる．

平均してIPCが2前後で，各クラスタ内演算器数が2つなら，評価に用いるベンチマーク・プログラムの一連の命令は，1つのクラスタだけでもほぼ実行できる，と考えられる．つまり，命令をクラスタに割り振る必要性が少なくなり，命令間の依存関係を考慮せずに，単純に空いているクラスタに命令を割り振る集中型の方が，他の2方式よりも高いIPCが得られると予測できる．

そこで，もう一方のクラスタの使用頻度も高めて評価を行うために，ロード命令に投機的実行を施すことで全体のIPCを向上させた．

第6章　評価

本章では，SimpleScalar ツールセット (ver.2.0)[11] に対して，第4章で述べたクラスタ化スーパースケーラ・プロセッサの各方式に依存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.

6.1　評価モデル

ベース・モデル

ベース・モデルは，INT 命令，LS 命令，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，4，32 である．全体の発行幅は $4\times3 = 12$ であるが，フェッチ幅は 8 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 $\times2$ と呼ぶことにし，フェッチ幅，発行幅，ウィンドウ・サイズと EU 数を半分にしたモデルを $\times1$ と呼ぶ．本研究の評価は主に $\times2$ モデルである.

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

各方式のクラスタへの命令割り振りは，以下のアルゴリズムに基づく：

- 集中型：命令の割り振りは，*select* 時であり，より古い命令からクラスタ 0 の EU に優先的に割り振る．クラスタ 0 が空いていなければ，クラスタ 1 の EU に割り振る.
- 分散型：命令の割り振りは，*dispatch* 時である．フェッチされた命令間において，依存元となる命令が存在する場合には，これと同じ命令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.

- 疑似分散型：*dispatch* 時のクラスタへの命令割り振りは，分散型と同じである．しかし，この *dispatch* 時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* 時である．この *select* 時の命令割り振りでは，ある領域の命令は，それに対応するクラスタに高い優先順位を持つものとする．

6.2 評価結果

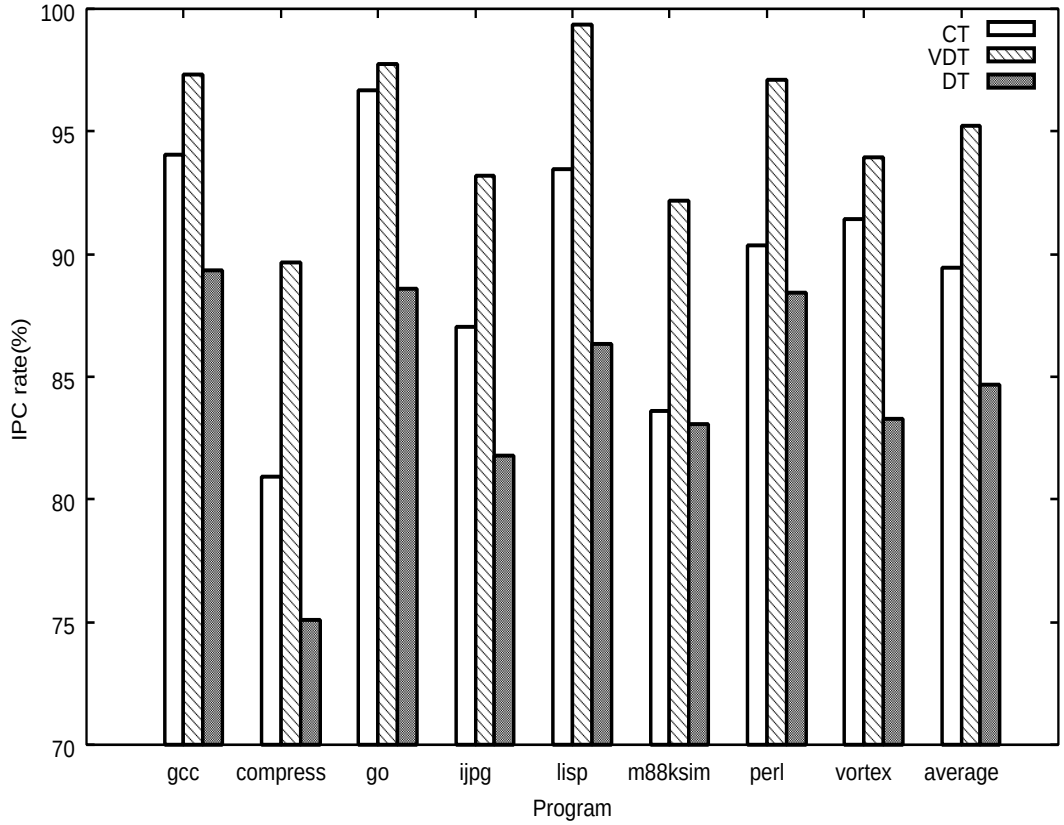

図 8: IPC 率，集中型 (CT), 疑似分散型 (VDT), 分散型 (DT)，$\times 1$

w は L-1 行列の幅である．クラスタ化プロセッサの各方式，つまり，集中型 (CT:Centralized Type)，疑似分散型 (VDT:Virtual Decentralized Type)，分散型 (DT:Decentralized Type) 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：$w = 16$) について比較した．図 8，図 9 の横軸は，評価に用いた SPEC ベンチマークと，全ベンチマークにおける平均，縦軸は，クラスタ化を行わない場合 (ベース・モデル) に対する，クラスタ化プロセッサの 3

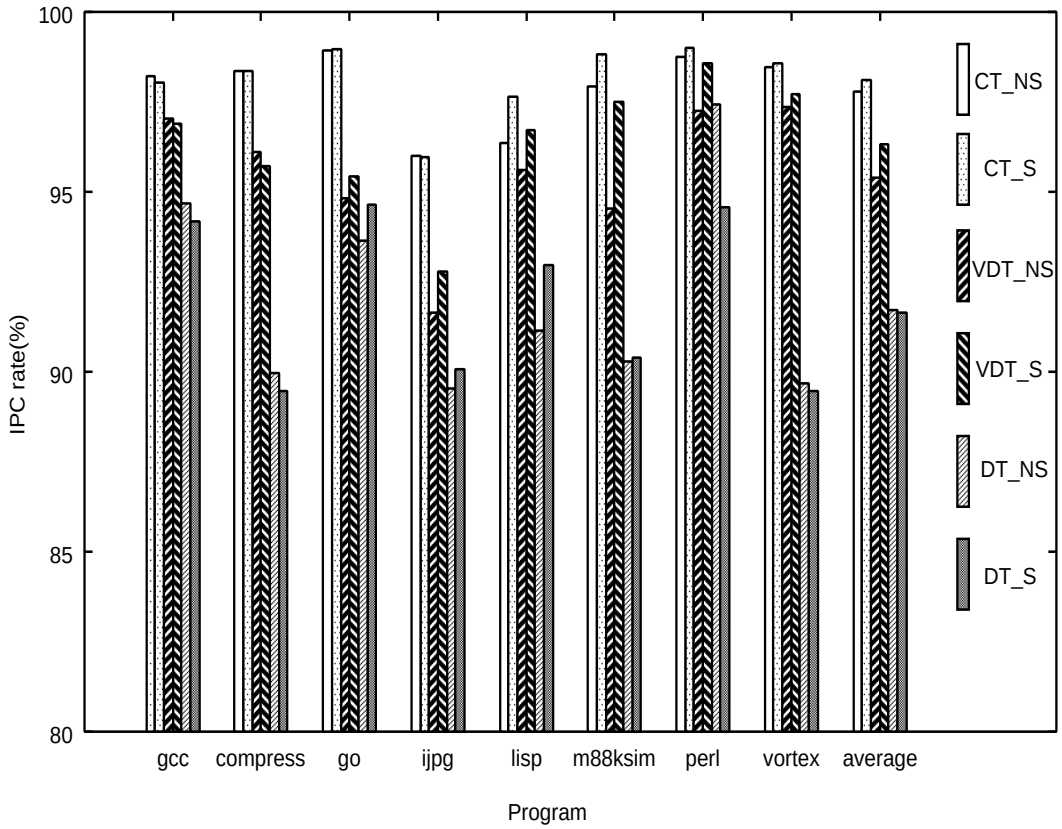

図9: IPC 率，各方式 (非投機/投機)，×2

表1: IPC，ベース・モデル，×2

Program	IPC	Program	IPC
gcc	1.300556	lisp	1.817902
compress	2.131910	m88ksim	2.166313
go	1.541727	perl	1.762126
ijpg	2.307741	vortex	2.066653

方式の IPC 比率を示した. また，図 9 は，8 つのベンチマーク・プログラムに対して，左から順に，集中型 (CT_NS:ロード非投機時)，集中型 (CT_S:ロード投機時)，疑似分散型 (VDT_NS:ロード非投機時)，疑似分散型 (VDT_S:ロード投機時)，分散型 (DT_NS:ロード非投機時)，分散型 (DT_S:ロード投機時) の IPC 比率を表したグラフである.

また表 1 は，×2 におけるベース・モデルの，各ベンチマーク・プログラムにおける IPC をまとめたものである. これらの値はロードの投機を行っていない

場合のIPC値である．第5章でロード命令の投機をした動機についての説明では，この表1を用いた．

図8，9では，すべてのグラフのIPC比率は100%より小さくなっている．これは前章でも述べたが，EUのクラスタ化によって，バイパス遅延を大幅に短縮しており，その代償としてIPCの減少が伴う．そのため，クラスタ化モデルを非クラスタ化モデルで正規化すれば，IPC比率が100%未満となるのは当然で，クラスタ化プロセッサでは，そのIPC低下以上の，バイパスロジックにおける遅延短縮が期待できる．

まず，図8では，×1のモデルに対する3方式の，クラスタ化していないベース・モデルで正規化したIPC比率を表している．同図は，L-1行列の狭幅化を行わない場合について比較したものであるが，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%悪化する結果となった． つまり同図では，疑似分散型の優位性が明確であり，第4章で述べた疑似分散型のメリットが強調された結果と考えられる．

しかし，図9から分かるように，×2のモデルでは集中型，疑似分散型，分散型の順に高いIPCを示している． ロード命令の投機的実行を行うと，平均してIPC比率が上がるのが分かる．ロード命令の投機をしているにもかかわらず，gcc，compressのようにIPC比率が下がっているのがあるが，これはあくまでクラスタ化していないモデルで正規化したためであって，IPCの絶対値そのものは，ロードの投機をした場合，全ての方式で向上した．

本研究は主に×2のモデルについての評価であったので，図9について考察する．実験データが得られる前は，同図のaverageのように，疑似分散型のIPC比率が他方式よりも上昇するだろうと予想できた．理由として，疑似分散型のロード投機前のIPC比率の低さや，ロード投機による命令並列性の向上で，クラスタへの命令割り振りにおける重要度の増加が挙げられる．しかし，実際は疑似分散型のIPC比率の向上も見られたが，それでも集中型のIPCには及ばなかった．同図より分かるが，集中型のIPC比率も向上しており，集中型の性能の高さがうかがえる．

第7章　まとめ

本稿ではまず，スーパースケーラ·プロセッサの構成要素のうち，将来クロック速度を制限するものとして考えられる，バイパスロジックと $wakeup$ ロジックについて，なぜそれらが将来クロック速度を制限するのかについて述べた．そして，それらの遅延に対して提案されている方式についてそれぞれ説明した．具体的には，バイパスロジックの遅延に対しては，スーパースケーラのクラスタ化という研究がされており，$wakeup$ ロジックの遅延に対しては，我々の提案した直接依存行列型スケジューリング方式がある．

その説明をした上で，クラスタ化スーパースケーラ·プロセッサに対して，直接依存行列型スケジューリング方式を適用する場合について述べた．そして，クラスタ化スーパースケーラ·プロセッサにおける3方式，つまり，集中型，分散型，疑似分散型の構成を説明し，これらの特徴についても述べた．実際に，クラスタ化プロセッサの3方式に対してIPC評価をすると，$\times 1$ のモデルでは，全てのベンチマーク·プログラムに対して，疑似分散型は集中型よりも良好な結果を示した．しかし，$\times 2$ のモデルでは，集中型と比較して疑似分散型の優位性は見られなかった．これは，EU数を2倍にしたことで，命令割り振りの意味合いが薄れ，命令の依存関係を考慮する必要性が低くなったためと考えられる．

そこで，$\times 2$ のモデルに対してロード命令の投機的実行を行い，命令並列性を高め，IPCを向上させた上で，集中型，分散型，疑似分散型の性能を再度比較し，3方式の特徴をより詳しく調べた．その結果，ロード命令の非投機時→投機時のIPC増加率は，疑似分散型が最も高くなったが，IPCの絶対値に関しては，ロード非投機時/投機時とも，集中型が最も大きな値を示した．つまり，ロードの投機を行って命令並列性を向上させ，もう片方のクラスタの使用頻度を高めても，分散型，疑似分散型より，シンプルに命令を割り振った集中型の方が優れていたことを示唆している．

今後の興味ある評価として，命令並列性の高いプログラムの使用や，ロード命令の投機的実行以外の方法でIPCを向上させた際の各方式の比較も考えられる．

謝辞

本研究を進めるに当たり，多くの御指導を賜わりました富田眞治教授に深く感謝の意を表します．また，日頃より熱心に指導して下さり，本報告書の作成に対しても多大なる助言を頂きました五島正裕氏に心より感謝致します．そして，日頃より技術的，精神的に支援して下さり，数多くの助言を頂いた富田研究室の皆様に感謝致します．

参考文献

[1] 五島正裕, 西野賢悟, グェンハイハー, 縣亮慶, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: 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: ハイパフォーマンスコンピューティングシステム, Vol.42, No.SIG 9(HPS 3), pp.77-92(2001).

[2] Palacharla, S., Jouppi, N.P. and Smith, J.E.: Quantifying the Complexity of Superscalar Processors, Technical Report, Univ. of Wisconsin-Madison (1996).

[3] Palacharla, S., Jouppi, N.P. and Smith, J.E.: Complexity-Effective Superscalar Processors, *Proc. 24th Int'l Symp. on Computer Architecture (ISCA24)* (1997).

[4] Kemp, G.A. and Franklin, M.: PEWs: A Decentralized Dynamic Scheduler for ILP Processing, *Proc. Int. Conf. on Parallel Processing* (1996).

[5] Farkas, K.I., Chow, P., Jouppi, N.P. and Vranesic, Z.: The Multicluster architecture: Reducing cycle time through partitioning, *Proc. 30th Int'l Symp. on Microarchitecture* (1997).

[6] Keller, J.: The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, *Proc. 9th Annual Microprocessor Forum* (1996).

[7] 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 *JSPP2001* (2001).

[8] 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: 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: 並列処理シンポジウム *JSPP2002* (2002).

[9] Yeager, K. C.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, No.4, pp.28-40 (1996).

[10] Henry, D.S., Kuszmaul, B.C., Loh, G.H. and Sami, R.: Circuits for Wide-Window Superscalar Processors, *Proc. 27th Int'l Symp. on Computer Architecture (ISCA27)* (2000).

[11] *SimpleScalar LLC: http://www.simplescalar.com/.*