特別研究報告書

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

指導教官　　富田　真治　教授

京都大学工学部情報学科

小田 累

平成13年2月8日

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

小田 累

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、Wakeup ロジックと Bypass ロジックの遅延である。

Wakeup ロジックとは、動的命令スケジューリングのため、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である。この Wakeup ロジックは、配線遅延に支配されるために、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 ロジックは、今後 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。

Bypass ロジックとは、演算器内でのレジスタ及び ALU の間でのデータの転送を行う Bypass のロジックのことである。Bypass ロジックでも、同様に配線遅延に支配されるために、LSI の微細化の恩恵を受けにくい。よって、Bypass ロジックも同様に、今後 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。

wakeup ロジックを高速化させる手法としては、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。一方で、Bypass ロジックの遅延の対策としては、スーパースケーラ ・プロセッサのクラスタ化を行う研究が行われている。

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場合について議論する。

クラスタ化スーパースケーラ ・プロセッサでは、命令キューの構成として集中、分散の 2 種類に分類できる。分散されている場合には、各キューのエントリ数も軽減でき、wakeup の遅延を更に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べて IPC が高くなると考えられる。

直接依存行列型スケジューリング方式では、パイプライン化が不可能である場合と、パイプライン化可能である場合のそれぞれに、個別の依存行列を適用する。前者に対しては、wakeup ロジックの遅延を減少させるために、さらに依存行列の縮小化を行う。この場合、依存行列の縮小化による wakeup ロジックの

遅延の減少とIPCとはトレード・オフであり、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、命令キューのエントリ数によって決まる。

本稿では、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・プロセッサに実装するにあたり、集中、分散の2つの命令キューの構成に加えて、両方の利点をあわせ持つと考えられる仮想的に命令を分散する方式を提案する。

今回、集中、分散の命令キューの構成及び、提案する仮想的分散キューの構成を持つ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を導入し、wakeupの遅延及びIPCの測定を行った。

その結果、命令キューを分散化することによるwakeupロジックの遅延の減少は行われたが、それ以上に命令キューを集中した方式が、仮想的に命令キューを分散化した方式、及び命令キューを分散化した方式よりも高いIPCを示した。これは、プログラムの命令並列性が低く、スーパースケーラ・プロセッサの演算器の稼働率が低いため、1つのクラスタだけで十分な命令処理が行われているからだと考えられる。

Direct dependence matrix scheduling scheme on clustered superscalar processors

RUI ODA

Abstract

Delay of Wakeup logic and Bypass logic is thought as what will restrict clock speed among the elements of a superscalar processor in the future.

Wakeup logic is the logic which pursues the validity of data required for execution of instructions because of dynamic instructions scheduling. The wakeup logic is influenced with wiring delay, so it can not receive the benefit of miniaturization of LSI easily. Moreover, unlike many of other elements of a superscalar processor, it cannot divide into two or more pipeline stages.

Bypass logic is the logic of Bypass which transmits the data between the register in operation units and ALUs. Like wakeup logic, the Bypass logic is governed by wiring delay, so it cannot receive the benefit of miniaturization of LSI ,too.

Therefore, it is expected that Wakeup logic and Bypass logic will become much more critical with enhancement of the miniaturization and pipeline of LSI from now on.

As a technique which makes wakeup logic accelerated, there is a direct dependence matrix based scheduling scheme. On the other hand,the research of clustered superscalar processor is done as a measure of delay of Bypass logic.

This thesis discusses the direct dependence matrix based scheduling sheme on clustered superscalar processor.

By the direct dependence matirix based scheduling sheme, an individual dependence matirix is applied to each in the case where the latency of the path is 1 cycle, and in the case where the latency of the path is one or more cycle.

The former dependence matrix is further narrowed in order to decrease delay of wakeup logic. Then,the delay of wakeup logic by narrowed dependence matrix and IPC are a trade-off, so it is decided by the number of entries of instruction queue how far delay can be decreased, suppressing decrease of IPC to tolerance level.

a clustered superscalar processor can classify into the two type. One is the

type of concented instruction queue and the other is the type of decentralized instruction queue. In the latter type,the number of entries of each queue can also be distributed and it is considered that delay of wakeup decreases further. However,compared with IPC of the decentralized instruction queue type, IPC of centralized instruction queue is presumed to become high, because of the different components between the two type.

In this thesis, when mounting direct dependence matrix based scheduling sheme in a clustered superscalar processor, in addition to the two type of concented and distributed instruction queue, it is proposed that the type of virtually decentralized instruction queue, which is considered to have both of advantages.

This time,we have measured the wakup logic latency and IPC of the three type of clustered superscalar processor, the concented,decentralized, and virtual decentralized instruction queue type, with direct dependence matrix based scheduling.

The result is that IPC of the centralized instruction queue type is higher than that of virtually decentralized,and decentralized instruction queue. The reason of the result is considered that only one clustered superscalar processer sufficiently process instructions because of the low operating ratio of the operation unit of a superscalar processor origined by low instruction level parallelism.

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

目次

第 1 章　はじめに　1

第 2 章　クラスタ化スーパースケーラ ・プロセッサ　2

2.1　クラスタ化スーパースケーラ ・プロセッサ 2

2.2　クラスタ化スーパースケーラの構成 4

第 3 章　直接依存行列型スケジューリング方式　4

3.1　動的命令スケジューリング方式 5

3.1.1　動的命令スケジューリングの原理 5

3.1.2　動的命令スケジューリングとパイプライン 6

3.2　直接依存行列型スケジューリング方式 7

3.2.1　D 行列 の概要 7

3.2.2　D 行列の高速化 8

第 4 章　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の組み合わせ　11

4.1　D 行列と集中キュー型スーパースケーラ ・プロセッサ 12

4.1.1　D 行列の導入方法 12

4.1.2　Steering ロジック 12

4.2　D 行列と分散キュー型スーパースケーラ ・プロセッサ 12

4.2.1　D 行列の導入方法 12

4.2.2　Steering ロジック 14

4.3　D 行列と仮想分散キュー型スーパースケーラ ・プロセッサ 15

4.3.1　D 行列の導入方法 15

4.3.2　Steering ロジック 15

4.4　各方式の wakeup の遅延と IPC の比較 16

4.4.1　wakeup の遅延の比較 17

4.4.2　IPC の比較 17

4.4.3　集中キュー型、分散キュー型に対する仮想分散キュー型の位置付け 20

第 5 章　性能評価 **21**
5.1　評価モデル 21
5.2　評価結果 21
第 6 章　まとめ **23**
謝辞 **24**
参考文献 **24**

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。スーパースケーラ ・プロセッサの性能を向上させるためには、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として、命令発行幅 (**IW** :Issue Width) 及びウィンドウサイズ (**WS** :Window Size) を増やす方法がある。実際、初期のスーパースケーラ ・プロセッサは、トランジスタ数の許す範囲で IW,WS を増やすことで大幅に 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、徒に IW, WS を増加させれば、クロック速度の低下をまねき、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考えられているのが、Bypass ロジックと Wakeup ロジックの遅延である [1]。

Bypass ロジックとは、演算器内でのレジスタ及び ALU の間でのデータの転送を行うロジックのことである。Bypass ロジックでも、同様に配線遅延に支配されるために、LSI の微細化の恩恵を受けにくい。よって、Wakeup ロジックと同様に今後 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。バイパス遅延を解決策として、クラスタ化スーパースケーラ ・プロセッサが研究されている [2] [3] [4] [5]。クラスタ化スーパースケーラ ・プロセッサは、命令キューの構成として集中、分散の 2 種類に分類できる。分散されている場合には、各キューのエントリ数も軽減でき、wakeup の遅延を更に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べて IPC が高くなると考えられる。

Wakeup ロジックとは、動的命令スケジューリングのための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である。この Wakeup ロジックは、配線遅延に支配されるために、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 ロジックは、今後 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。Wakeup のロジックを高

速化させる方法のひとつとして、直接依存行列型スケジューリング方式がある。[6] 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、依存行列を縮小化することによる wakeup ロジックの遅延の減少は、IPC とトレードオフであり、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、命令キューのエントリ数によって決まる。

本稿では、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・プロセッサに実装するにあたり、集中、分散の 2 つの命令キューの構成に加えて、両方の利点をあわせ持つと考えられる仮想的に命令を分散する方式を提案する。そして、クラスタ化スーパースケーラ・プロセッサの 3 つの方式について、各方式の IPC の比較、及びキューを分散化した場合における wakeup ロジックの遅延の減少の効果測定を行う。

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について詳細を述べる。第 4 章では、クラスタ化スーパースケーラ・プロセッサに対する直接依存行列型スケジューリング方式の導入について述べ、第 5 章では、第 4 章に対する実装及び性能評価を行う。そして、最後に第 6 章で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

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述べる。一般に、配線遅延は、配線の長さの長くなるにつれ大きくなること、及び WS 及び IW が増えるにつれて Bypass ロジックの配線が長くなることから、Bypass ロジックの遅延は IW が増大するにつれて今後ますますクリティカルになると予想される。

この Bypass 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次節から述べる。

2.1　クラスタ化スーパースケーラ・プロセッサ

この節では、前節で述べた Bypass ロジックの遅延を解決するための手法であ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。

図 1(a)(b) 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

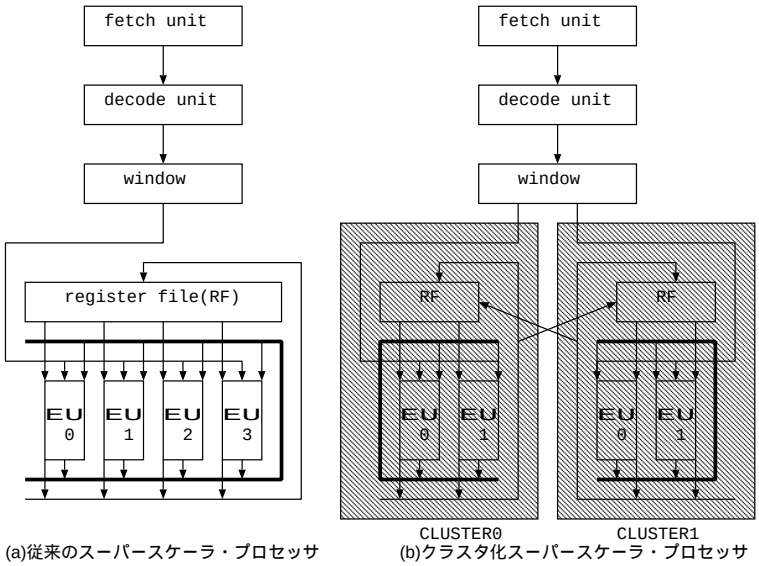


図 1: クラスタ化スーパースケーラ

パースケーラ ・プロセッサを、図 1(b) は、(a) をクラスタ数 2 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 においてはスーパースケーラ ・プロセッサは、 フェッチ ・ユニット、デコード ・ユニット 命令ウィンドウ、 レジスタ ・ファイル (**RF** :Register File)、実行ユニット (**EU** :Execution Unit) を示す。図 1(b) の網掛け部分がクラスタであることを示す。また、図 1 の太線は、ローカルバスを示しており、ローカルバスは、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/ n 個の実行ユニットを持つ n 個のクラスタに分割した場合、バイパス先の実行ユニット数は、1/ n 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、各クラスタの命令発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。

クラスタ化された場合のローカルバスは、図 1 が示すように、クラスタ化されていないスーパースケーラと比較して短くなる。その結果、クラスタ内の Bypass ロジックの遅延を減少させることができる。

一方、異なるクラスタ間の場合、CLUSTER0 の EU0 結果を CLUSTER1 の EU1 が使用する場合には、CLUSTER1 の EU1 は、CLUSTER0 にあるレジスタファイルから結果を参照しなくてはならない。よって、異なるクラスタ間での転送には同じクラスタ間での通信と比べて 1 サイクル余計にペナルティが生じ

てしまうことになる。今後、このクラスタ間の転送によるペナルティを **cluster delay** と呼ぶ。

このクラスタ化の技術は、今後、IW が増えるにつれてクリティカルとなっていく Bypass ロジックの遅延を低減する上でますます重要となっていくと予測される。

2.2 クラスタ化スーパースケーラの構成

演算器にクラスタリングを施した場合、演算器がクラスタリングされる以外にもスーパースケーラの構成が変化する。

まず、演算器のクラスタリングを行った場合、フェーズのある段階で必ず命令をどのクラスタ演算器に振り分けるかを判定するロジックが必要となる。このロジックを **Steering** という。一般に、Steering を行うフェーズは、dispatch と issue の場合の 2 つの場合がある。以下、dispatch で行う Steering を **Dispatch_Steering** 、issue で行う Steering を **Exec_Steering** と呼ぶ。

演算器をクラスタ化させる場合に、Dispatch_Steering の場合には、各クラスタに対応した命令キューに命令を割り当てるため、命令キューを各クラスタ毎に分散化させる。この Dispatch_Steering を行い命令キューを分散化したものを、「分散キュー型」と呼ぶ。同様に Exec_Steering を行う場合は、各クラスタに命令を割り当てる直前に Exec_Steering を行うため、特に命令キューを分散化させる必要はなく、命令キューは 1 つに集中している。これを「集中キュー型」と呼ぶ。

この集中キュー型と分散キュー型の方式の違いは、直接依存行列スケジューリングをクラスタ化スーパースケーラ・プロセッサに導入する際に依存行列の構成と深く関係する。これについては、第 4 章で詳しく述べる。

第 3 章　直接依存行列型スケジューリング方式

第 3 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、その解決方法のひとつとして我々の提案する直接依存行列型スケジューリング方式の詳細を述べる。

3.1 動的命令スケジューリング方式

スーパスケーラの構成する基本構造のうち、演算器それ自体以外のほとんど全ての遅延は、IW, WS の増加関数で与えられる。構成要素としては、キャッシュ、命令フェッチ ・ロジック、レジスタ ・ファイル、オペランド ・バイパス、そして、この章で述べる動的命令スケジューリングを行うロジックなどがある。そのうちいくつかの処理に対しては、パイプライニングやクラスタリングなどの技術によって、 1 サイクルに終えなければならない処理の遅延を大幅に短縮できる。

これらの技術は、ロジックの遅延を、一部の命令の実行レイテンシや、何らかのペナルティに転化するものである。したがって、これらの技術が有効であるためには、クロック速度の向上に対して、IPC の悪化の度合いが十分に小さい必要がある。

しかし、動的命令スケジューリングを行うロジックに対しては、このような技術は効果的でない。本節では、その理由について述べる。以下、まず、3.1.1 項においてスーパースケーラの動的命令スケジューリングの原理についてまとめ、3.1.2 項でスケジューリングの処理と命令パイプラインについて説明する。

3.1.1 動的命令スケジューリングの原理

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、空いている機能ユニットに対して、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そろ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が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。

動的スケジューリングの処理のうち特に動的命令スケジューリングに関わるフェーズは、**(1)dispatch (2)wakeup (3)select** の 3 つのフェーズである。ある命令 ***Ic*** 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、***Ic*** の左ソース ・オペランドは、先行する命令の ***Ip*** の結果、右は即値であるとする。

(1)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。***Ic*** 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

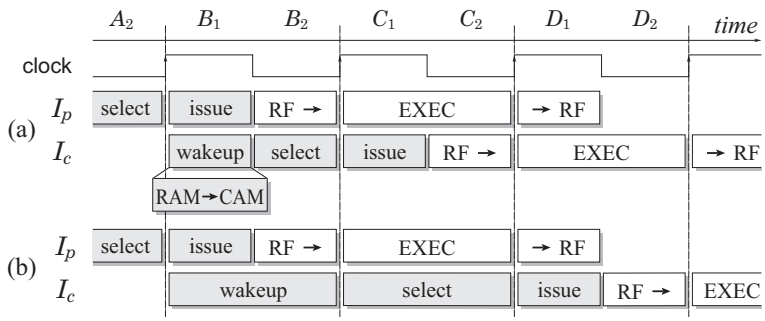


図 2: 命令パイプラインの構成

れる。一方、rdyL の初期値は、Ip 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。既に命令が実行されている場合、及び即値であれば、初期値には”1”を、まだ命令が実行されていない場合は、rdyL は”0”に初期化され、Ic は、Ip が実行されるのをまって、ウィンドウ内で 『眠る (sleep)』ことになる。

(2)wakeup

やがて、select によって命令 Ip の発行が決定されると、その結果が利用可能となる時刻も決まる。すると Ic の rdyL は、適当なタイミングで”1”にセットされ、Ic は、 『起こされる (wakeup)』ことになる。本稿では、wakeup は、命令の発行が決定した際の rdyL/R の更新処理と定義する。

(3)select

rdyL/R が共にセットされている命令が、実行可能な命令である。select では、あ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。

3.1.2 動的命令スケジューリングとパイプライン

この節では、動的命令スケジューリングの各処理をパイプライン化することを考える。

命令スケジューリングの 3 つのフェーズのうち、特に問題となるのが wakeup,select のフェーズであり、パイプライン化が必ずしも可能ではない。以下、パイプライン化について説明を行う。

wakeup,select,issue のパイプライン化

図 2 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。[7] 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。

図 2(a) は、wakeup+select を 1 サイクルで行った場合、図 2(b) は、wakeup+select を 1 サイクル以上かけて行った場合を示す。

遅延とパイプライン化

wakeup と select は、一般にはパイプライン化することができない。図 2 からわかるように、wakeup と select は合わせて 1 サイクル以下でなければ、***Ic*** を ***Ip*** の次のサイクルに実行することができなくなる。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。この場合、IPC の悪化は 5 % ~ 15 %程度にもなることがわかっており [7]、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。

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、いっそうクリティカルになっていくと予測されている。次節では、この wakeup に対して我々が提案する手法を述べる。

3.2 直接依存行列型スケジューリング方式

我々の提案する手法では、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 (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。以下は、D 行列について説明を行う。

3.2.1 D 行列 の概要

図 3 に、D 行列 の概念図を示す。D 行列は、**rdyL/R** 用に各 1 つずつ用意された WS × WS の行列である。ウィンドウ内のエントリ ID = p の命令 ***Ip*** の実行結果を、ID = c の命令 ***Ic*** 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。

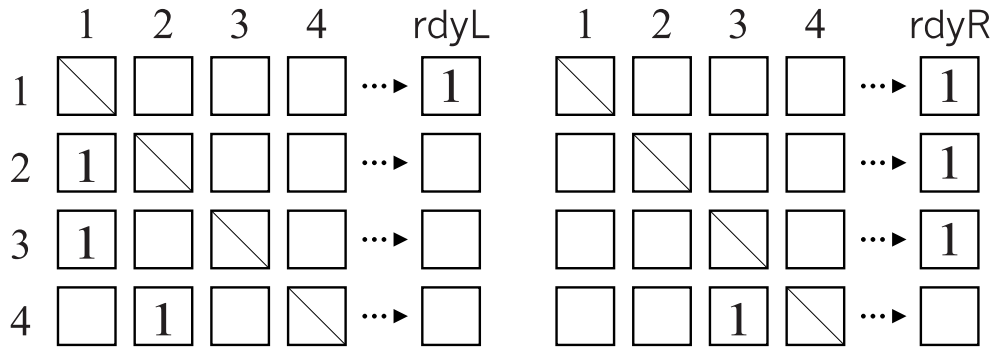


図 3: 直接依存行列の概念図

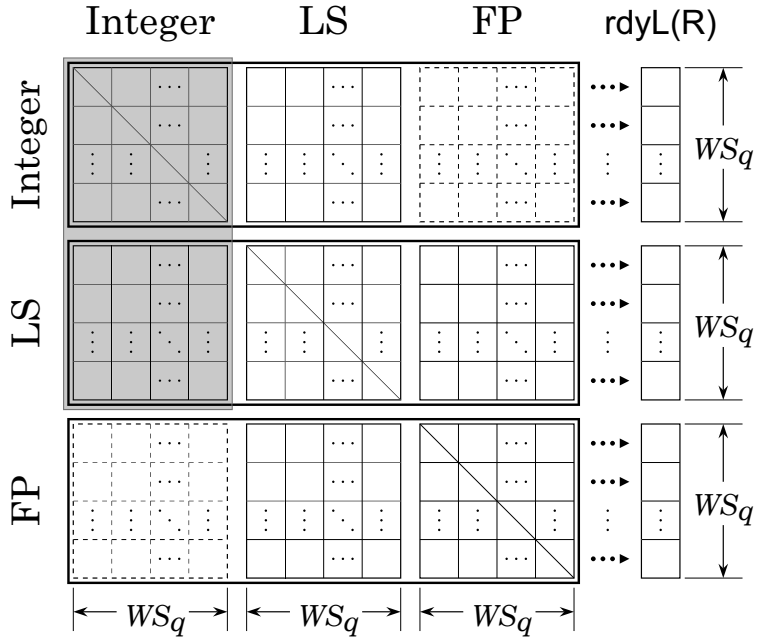


図 4: D 行列の分散化

wakeup においては、発行される IW 個の Ip 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。

なお、D 行列 を実装するにあたっては、従来の CAM+RAM 方式のかわりに単に 1-read の RAM を用いればよく wakeup を高速化できる。複数行の OR を求めるための特別なロジックは必要ない。

3.2.2 D 行列の高速化

通常、動的命令スケジューリングでは、命令ウィンドウを分散化することが可能である。この命令ウィンドウの分散化を行うことで、D 行列も合わせて分散化することができ、その結果、wakeup の遅延を減少させることができる。ま

ず、命令ウィンドウの分散化について述べ、その後、D 行列の分散化の方法を述べる。

命令ウィンドウの分散化

命令ウィンドウを分散化する場合、出現命令に偏りがある場合にIPC が低下するという欠点がある。

Integer、Floatpoint、Load/Store の 3 つのクラス毎に命令ウィンドウを分散化する場合を考える。命令ウィンドウのエントリ数が全体で30、各クラスごとの命令ウィンドウのエントリ数を 10 とする。このとき、Integer の命令のみが 20 個続けて出現した場合、命令ウィンドウを分散化していない場合は、全ての命令を命令ウィンドウに割り当てることが可能である。一方、分散化をしている場合は、Integer 用の命令キューのエントリ数が 10 しかないため命令の割り当てができない。しかし、上記の例は極端な例であり、実際に命令ウィンドウの分散化を行った結果、IPC に対する影響は非常に小さく個々のロジックを大幅に簡単化できることがわかっている。

よって、実際の命令ウィンドウは、クラス毎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化されることが多い。ロジックの遅延を考えるにあたっては、この分散化を避けては通れない。

分散化の効果には、次の 2 つがある ：

(1) 構成要素のパラメータが大幅に縮小される。

複数のキューが分散化した場合、個々の構成要素パラメータが大幅に縮小される。各命令キューの命令発行幅とサイズをIWq、WSq とした場合、パラメータは以下のようになる。:

- 命令キューのサイズは、WS から WSq に縮小される。
- 命令キューの命令発行幅は、IW から IWq に縮小される。
- select ロジックは、WS → IW から WSq → IWq に縮小される。ただし、WS → IW とは、WS 個の発行要求から IW 個を選択することを表す。

(2) 演算器の実行レイテンシに合わせた最適化が可能になる。

実行レイテンシが 1 である演算器からはのオペランド ・バイパスがあるパスでは、wakeup と select は合わせて 1 サイクルで実行する必要がある。R10000 の場合、この条件を満たすのは、

- 整数から整数

- 整数からロード/ストア

の 2 ヶ所である。この 2 ヶ所以外では、wakeup と select を適当にパイプライン化してよい。次に D 行列の分散化について述べる。

D 行列の分散化

命令ウィンドウが q 本のサイズ WSq の命令キューに分散化されたとき、D 行列は、図 4 のように、q 個の WS 行 WSq 列の部分行列に分割される。この分割によって得られるメリットには以下の 2 つがある。

パラメータの減少

書き込みポート数が IW から IWq に減少する。

レイテンシにあわせた最適化

3.1.2 節で述べたようにパイプライン化が不可能な場合は、命令が最もクリティカルな場合、つまり、Ip の実行レイテンシが 1 の場合である。この条件を満たすのは、Integer から Integer、Integer から Load/Store の 2 ヶ所である。パイプライン化の可 ・不可によって D 行列をわけることで、それぞれに適した D 行列を構成することが可能となる。

具体的には、Integer から Integer、Integer から Load/Store の 2 ヶ所とそれ以外で依存行列を分ける。前者の D 行列を **L-1D** 行列、後者の D 行列を **L-2D** 行列と呼ぶ。

L-2D 行列においては、パイプライン化が可能であるため、wakeup の遅延がクリティカルになることは少ないと考えられる。一方、L-1 D 行列は、パイプライン化が不可能なため、wakeup の遅延 がクリティカルになる。

L-1 D 行列は、元の D 行列からすると格段に小型化されているためその読み出し遅延も短くなる。さらに、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。

L-1 D 行列の縮小

一般的に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 %以上を占めることがわかっている。この性質を利用して wakeup を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 にうつすのである。**rdyL/R** は、命令間の距離が w 以下の場合は L-1 によって、それ以外では L-2 D 行列によって更新される。

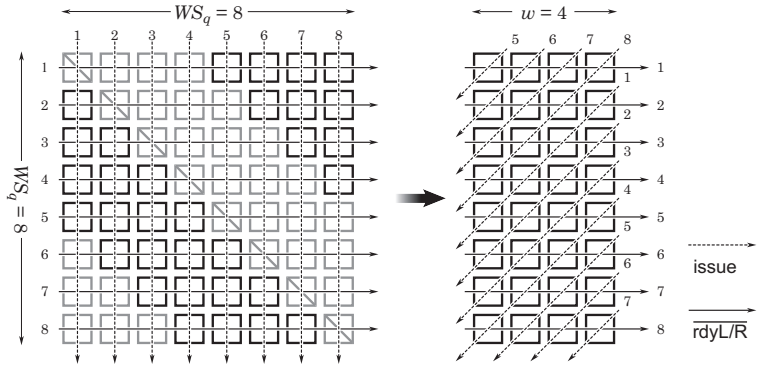


図 5: D 行列の縮小化

図 5 に、L-1 D 行列の縮小の様子を示す。左は元々の、右が縮小後の L-1 D 行列である。同図では、WS = 8、w = 4 である。元々L-1 D 行列から削除されるセルを、左図で薄く示している。必要なセルを矩形領域に集める方法としては、よりクリティカルであるビット線の長さを短縮するために右図のように行う。ここで、w を L-1 D 行列の幅と呼ぶ。

図 5 の右図から明らかなように、縮小された L-1 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 とは無関係で、w によってのみ決めることができる。このため、L-1D 行列の遅延は、WS とは無関係となり、幅wによって決めることができる。ただし、IPC の悪化を許容範囲に抑えつつ幅wをどこまで縮小できるかは、WS の大きさによる。

第 4 章　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。

まず、集中キュー型、分散キュー型のスーパースケーラ ・プロセッサに対して、第 3 章で述べた直接依存行列型スケジューリングを導入方法について述べる。また、本稿で提案する 「仮想分散キュー型」についても述べる。仮想分散キュー型とは、集中キュー型の命令ウィンドウを仮想的に分散化することによって、集中キュー型と分散キュー型の利点を共に持つと考えられる方式である。

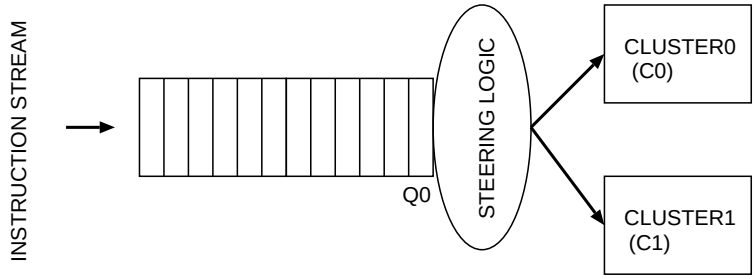


図 6: 集中キュー型

その後、各方式の IPC 及び wakeup の遅延に影響を及ぼす要素についての比較を行う。これは、直接依存行列型スケジューリング方式を適用した、集中キュー型、分散キュー型、仮想分散キュー型の各方式を比較する際、IPC の悪化を許容範囲に抑えつつ、どこまで wakeup の遅延を減少できるかが焦点となるからである。

4.1 D 行列と集中キュー型スーパースケーラ ・プロセッサ

4.1.1 D 行列の導入方法

図 6 に集中キュー型の構成を示す。図 6 に示すように集中キュー型の場合は、クラスタ化を行っている以外は、従来型のスーパースケーラ ・プロセッサと構成が等しいため、D 行列も従来のスーパースケーラ ・プロセッサと同様に導入すればよい。

4.1.2 Steering ロジック

第 3 章でも述べたように、集中キュー型においては Exec_Steering を用いる。今回導入している Exec_Steering のロジックは以下の通りである。

- CLUSTER を番号順に走査していき、空いている実行ユニットがあれば、その実行ユニットへ issue する。

4.2 D 行列と分散キュー型スーパースケーラ ・プロセッサ

4.2.1 D 行列の導入方法

図 7 に分散キュー型を示す。

分散キュー型の場合は、Dispatch_Steering によって、異なる命令ウィンドウに依存元命令 Ip と依存先命令 Ic をそれぞれ割り当てた際、Ip、Ic は、そのまま異なるクラスタの実行ユニットに割り当てられるため cluster delay が生じ

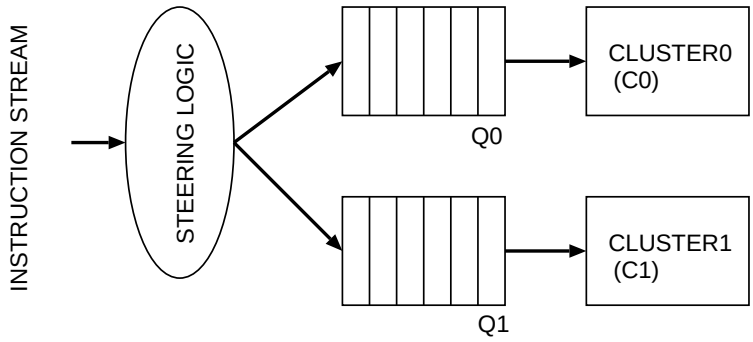
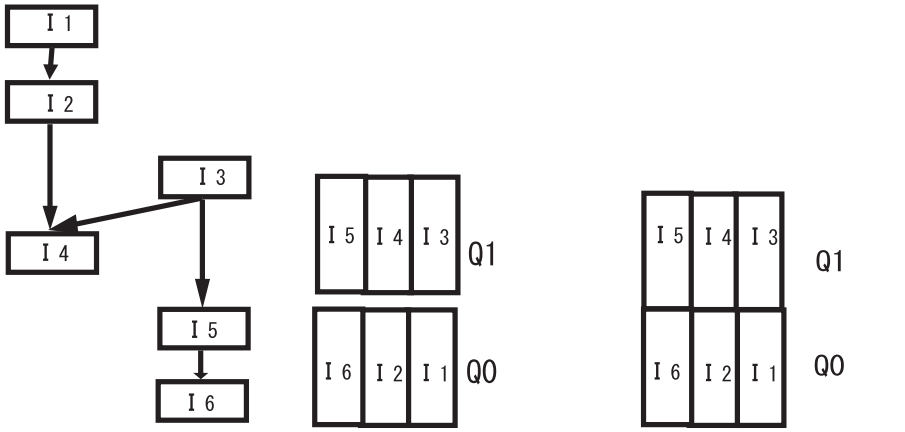
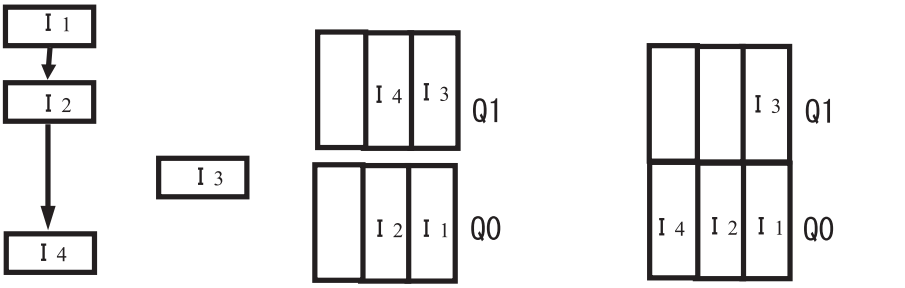


図 7: 分散キュー型



(A)命令順序と依存関係　(B)分散キュー型の結果(IW=3×2)　(C)仮想分散キュー型の結果(IW=6)



(D)命令順序と依存関係　(E)分散キュー型の結果(IW=2×2)　(F)仮想分散キュー型の結果(IW=4)

図 8: Dispatch_Steering ロジック

る。異なる命令ウィンドウに命令を割り当てる場合は、命令 Ic からみると、命令 Ip は cluster delay が生じるため実行レイテンシ 2 の命令と等価であるとみなせる。この場合、wakeup+select を 1 サイクル以上かけることが可能となる。一方、同一命令ウィンドウに命令が割り当てられた場合は、クラスタと対応する命令ウィンドウの構成は、従来型のスーパースケーラ ・プロセッサと等しい

ため、L-1D 行列を用いるのは、Ip の実行レイテンシが 1 の場合である。以上から、分散キュー型スーパースケーラ・プロセッサの場合、wakeup+select を 1 サイクル以内で行わなければならないのは、命令 Ip、Ic が共に同じ命令キューに存在し、かつ Ip の実行レイテンシが 1 の場合である。

第 3 章で述べたように、L-1 D 行列を使うときは、wakeup+select がクリティカルになる場合、つまり、wakeup+select を 1 サイクル以内で行う場合であった。つまり、L-1 D 行列は、同じ命令キューの命令の実行においてのみ用いればよく、L-1 D 行列は各クラスタに分散化できることがわかる。

以下に L-1 D 行列を分散化する方法を示す。

ｎ個のクラスタに実行ユニットが分散されたとし、分散キュー型の場合には、分散化された命令キューのエントリ数は、WSq/ｎ となる。

この場合、(Wsq -1)b × WSq word の L-1D 行列を、ｎ個のサイズの (WSq/n -1)b × WSq/ｎ word の L-1D 行列に分割する。

4.2.2 Steering ロジック

分散キュー型で導入している Dispatch_Steering のロジックを、図 8 に示す。

図 8(B) に割り当てる命令を図 8(A) に、図 8(E) に割り当てる命令を図 8(D) に示す。図 8(A)(D) において命令をつなぐ矢印は依存の方向を表し、「$Ip \rightarrow Ic$」のように依存元から依存先へつながっている。また、命令 I1 ~ I6 は、命令順に番号を付けている。

なお、クラスタ数ｎ、スーパースケーラ・プロセッサ全体の命令発行幅を IW、命令キューのエントリ数を WS とした場合、分散キュー型の命令キューは、ｎ個の命令発行数 IW/ｎ、各命令キューの最大エントリ数 WS/ｎ の命令キューに分散化されると考えられる。

図 8(B) は、分散キュー型において、各命令キューが命令発行幅 3、エントリ数 3 である場合の Dispatch_Steering の例を、図 8(E) は、分散キュー型において、各命令キューが命令発行幅 2、エントリ数 3 である場合の Dispatch_Steering の例を示している。

分散キュー型で用いている Dispatch_Steering は、以下の通りである。

1. 命令 Ic を issue する場合、依存元 Ip が存在しない場合は、最も命令キューの空きの多い命令キューに dispatch する。
2. 依存元命令 Ip がある場合は、Ip がある命令キューのうち最も空きの多い命令キューに dispatch する。

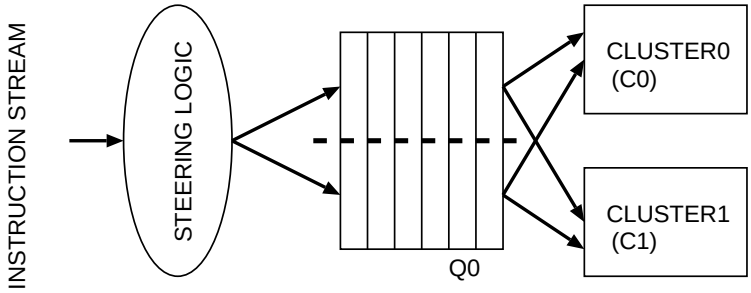


図 9: 仮想分散キュー型

3. 各命令キューの空きが等しい場合は、キュー番号の若い命令キューに命令を割り当てる。
4. 条件 1、2、3 にあてはまった際に、命令キューのエントリ数を超える、あるいは命令キューへの命令発行数を超える場合は、別の命令ウィンドウに命令を割り当てる。

図 8(b) において、I3 を Dispatch_Steering するときが条件 1、I4 を Dispatch_Steering するときが条件 2、I1 を Dispatch_Steering するときが条件 3、I6 を Dispatch_Steering するときが条件 4、のロジックをそれぞれ用いている。

4.3 D 行列と仮想分散キュー型スーパースケーラ ・プロセッサ

4.3.1 D 行列の導入方法

本稿において提案する仮想分散キュー型は、図 9 に示すように、L-1 D 行列を分散化するために命令キューを仮想的に分散化しているが、分散キュー型と異なり命令キューに対応するクラスタのみでなく他のクラスタにも、命令を割り当てることができるという方式である。

L-1D 行列の分散化の方法は、4.2 節で述べた通りである。

4.3.2 Steering ロジック

仮想分散キュー型では、Dispatch_Steering 及び Exec_Steering の両方で Steering を行う。

分散キュー型で導入している Dispatch_Steering のロジックを、図 8 に示す。

図 8(C) に割り当てる命令を図 8(A) に、図 8(F) に割り当てる命令を図 8(D) に示す。

命令キュー全体のエントリ数を WS、命令発行幅を IW、クラスタ数を 2 とする。分散キュー型では、命令キューは、WS/2 に分散化され、各命令キューへ

の命令発行数は、最大でIW/ 2 、全体としてIW であった。一方、集中キュー型では、命令キューへの命令発行数は、全体でIW、エントリ数は、全体でWS となる。

仮想分散キュー型は、各命令キューを仮想的に分散化するため、各命令キューのエントリ数は、WS/2 となる。しかし、仮想分散キュー型では、実際には命令キューは分散化されておらず、集中キュー型と同様に1つの命令キューだとみなせるので、命令キューの命令発行幅は、全体で最大IW となる。

図8(C) は、仮想分散キュー型において、各命令キューがエントリ数3で、全体の命令発行数が6である場合のDispatch_Steering の例を、図8(F) は、仮想分散キュー型において、各命令キューがエントリ数3で、全体の命令発行数が4である場合のDispatch_Steering の例を示す。

仮想分散キュー型で用いているDispatch_Steering は、分散キュー型と同じロジックを用いている。しかし、各命令キュー毎に命令発行数が決められている分散キュー型の場合と比べ、仮想分散キュー型は全体で命令発行幅が決められているため、図8(E)(F) のように、依存関係にある命令を同一命令キューにより多く割り当てることができる。

Dispatch_Steering については分散キュー型と同じロジックであるため、以下は、Exec_Steering について述べる。

1. 命令 ***Ic*** をissue する場合、命令キューに対応するクラスタの演算器が実行可能であれば、同クラスタ内に命令をissue する。そうでない場合は、クラスタ番号順に走査し、空いている実行ユニットのあるクラスタにissue する。

仮想分散キュー型では、Dispatch_Steering の段階で依存関係を基に優先度を決定しているため、依存関係を考慮したExec_Steering を行うことができる。

4.4 各方式のwakeup の遅延とIPC の比較

本節では、集中キュー型、分散キュー型、仮想分散キュー型についてwakeup の遅延とIPC についての比較を行う。これは、直接依存行列型スケジューリング方式を適用した、集中キュー型、分散キュー型、仮想分散キュー型の各方式を比較する際、IPC の悪化を許容範囲に抑えつつ、どこまでwakeup の遅延を減少できるかが焦点となるからである。

wakeup の遅延の減少に影響を与えているものとしては、命令キューの分散化があげられている。

また、IPC に影響を与えているものとしては、Steering の効率性、あるいはパイプラインにおけるペナルティなどがあげられる。

4.4.1 wakeup の遅延の比較

wakeup の遅延については、D 行列の分散化を行うかどうかが影響を及ぼす。

分散キュー型、及び仮想分散キュー型で命令キューを分散化した場合、L-1D 行列の分散化が可能である。L-1D 行列の分割により分散化された個々の L-1D 行列の大きさは、分散化する前の L-1D 行列よりも小さくなる。よって、分散化された L-1D 行列は、分散化されていない L-1D 行列と比較して、幅 w をより縮小させることができ、その結果 wakeup の遅延を減少できる。以上より、分散キュー型、仮想分散キュー型の wakeup の遅延は、集中キュー型よりも減少すると考えられる。

4.4.2 IPC の比較

IPC については、Steering の効率とパイプラインが影響を及ぼす。まず、Steering の効率が IPC に及ぼす影響について述べる。

Steering の効率

Steering には、Exec_Steering と Dispatch_Steering がある。Exec_Steering の場合は、実行ユニットへ直接命令を割り当てるため、実行ユニットの空きを常に把握しながら命令を割り当てることができる。一方、Dispatch_Steering の場合、ある命令 Ip を Dispatch_Steering した時と、実際にその命令 Ip が実行される時には必ずタイムラグが存在する。そのため、dispatch の段階で先読みして命令を割り当てた場合や、キャッシュミスのために Dispatch_Steering の予測とは異なる状況に陥いる場合に、実行ユニットに空きがなかった場合は、他のクラスタの実行ユニットに空きがある場合でも命令が select されないことになる。

以上から、集中キュー型と比較して、分散キュー型の方が IPC の低下が大きいといえる。また、仮想分散キュー型は、Dispatch_Steering を行った際にクラスタの優先度が決定され、Exec_Steering の際に、まず優先度の高いクラスタに命令を割り当てようとするが、分散キュー型と違い他のクラスタにも命令を割り当てることが可能であるので、分散キュー型より IPC が大きく、集中キュー型と同程度の IPC となる。

次に、パイプラインで起こりうる状況について各方式の比較を行う。

cluster delay とパイプラインの状況

第 2 章で述べたようにスーパースケーラ・プロセッサをクラスタ化したい場合、

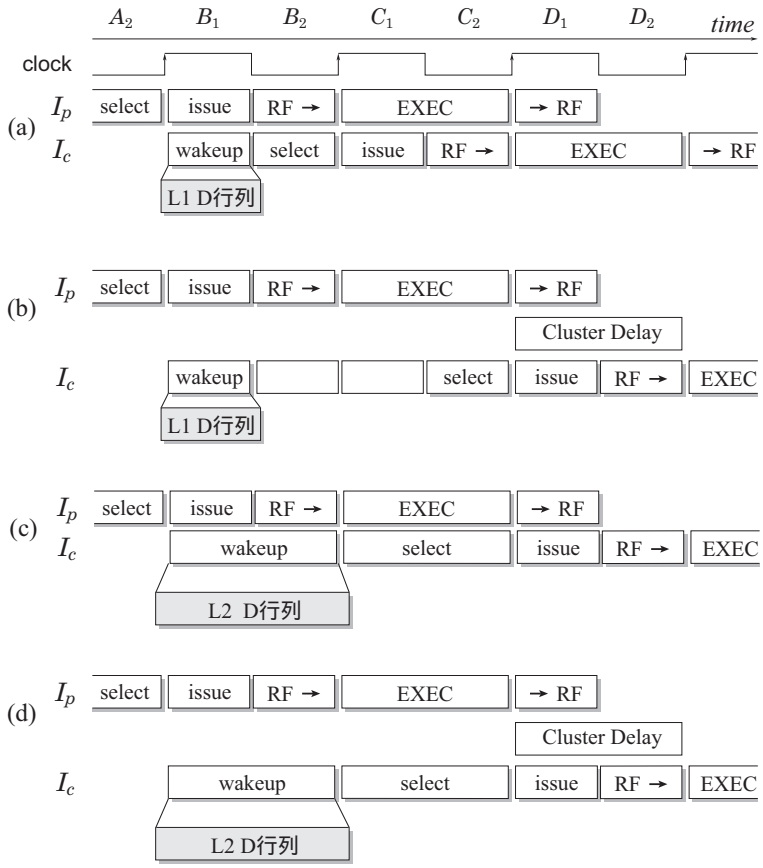


図 10: cluster delay とパイプライン

異なるクラスタ間の転送には cluster delay が生じる。この cluster delay が生じることで、3.1.2 節で述べた、従来のスーパースケーラ ・プロセッサのパイプラインの状況に加えて更に新たなパイプラインの状況が生じることになる。本項では、まずこのパイプラインについてまとめ、その後各方式の比較を行う。

図 10 は、命令が最もクリティカルな場合、つまり、Ip 及び Ic の実行レイテンシが 1 の場合のパイプラインの状況を表している。ここでは簡単のため、L-1D 行列の幅は w = WSq の場合 、つまり、L-1D 行列の縮小は行っていない場合を考える。よって、Ip と Ic の命令間の距離が L-1D 行列の幅 w を超えることはないものとし、L-1D 行列を用いるのは、命令が同一命令キューにある場合、L-2D 行列を用いるのは、命令が異なる命令キューにある場合とする。

wakeup+select を行う際、L-1D 行列を用い 1 サイクル以内に行う場合と、L-2D 行列を用い 1 サイクル以上で行う場合の 2 通りがある。また、異なるクラスタに命令 Ip と Ic を割り当て cluster delay が生じる場合と、同一クラスタに命

令 ***Ip*** と ***Ic*** を割り当て cluster delay が生じる場合の 2 通りがある。

これらの状況をあわせると、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用いた場合にパイプラインの状況として考えられるのは、$2 \times 2 = 4$ 通りである。これらのパイプライン化の様子が図 10 の (a) ~ (d) である。

次にそれぞれの場合について説明をおこなう。

(a)L-1D 行列を用いて cluster delay がない場合

L-1D 行列で wakeup がなされ、同一クラスタに命令が割り当てられた場合であり特に問題はない。

(b)L-1D 行列を用いて cluster delay がある場合

L-1D 行列で wakeup がなされ、異なるクラスタに命令が割り当てられた場合である。この場合、***Ic*** のパイプラインをそのまま進めると、***Ip*** の結果が ***Ic*** の実行結果に間に合わなくなる。そのため、何らかの方法で、***Ic*** の実行を遅らせる必要がある。

(c)L-2D 行列を用いて cluster delay がない場合

L-2D 行列で wakeup がなされ、同一クラスタに命令が割り当てられた場合である。L-2D 行列を用いているために wakeup 直後に ***Ic*** が実行できなくなり、L-1D 行列を用いた場合と比べて 1 サイクル遅れている。

(d)L-2D 行列を用いて cluster delay がある場合

L-2D 行列で wakeup がなされ、異なるクラスタに命令が割り当てられた場合である。示しているようにやはり 1 サイクル遅れている。

以上をまとめると、(a) の場合は、特にペナルティなく wakeup を行うことが可能であり、(b) ~ (d) の場合は、いずれも 1 サイクルのペナルティが生じるといえる。

集中キュー型において、パイプライン化の際に起こりうるパターンは、(a)(b) の 2 通りである。(b) に陥る頻度は、Exec_Steering の性能に依存する。単に空き実行ユニットが存在するクラスタにランダムに割り当てる場合が最悪であろう。依存関係まで考慮し、できだけ同一のクラスタに割り当てようとする場合には、割り当てようとするクラスタに空き実行ユニットが存在するかどうかに依存する。

分散キュー型において、パイプライン化の際に起こりうるパターンは、(a)(d)

の2通りである。(d) に陥るのは、Dispatch_Steering で依存関係にある命令が異なるウィンドウにdispatch された場合である。この頻度は、まずDispatch_Steering の依存解析の性能による。依存解析があたった場合で、更に (d) に陥る場合は、割り当てようとしたウィンドウの空きが問題となる。

仮想分散キュー型において、パイプラインの際に起こりうるパターンは、(a)(b)(c)(d) の全通りである。異なるウィンドウに命令を割り当てる (c)(d) に関しては分散キュー型の議論と同様である。また、(b) に関しても、集中キュー型の議論と同様である。

仮想分散キュー型は、集中キュー型と比べ、パイプラインの状況で (c)(d) に陥る分は、集中キュー型よりもIPC が低下すると考えられる。また、分散キュー型と比べ、パイプラインの状況で (b) に陥る分は、分散キュー型よりもIPC が低下すると考えられる。

なお、実際にどの程度IPC が下がるかは、パイプラインのペナルティの発生の頻度による。

4.4.3 集中キュー型、分散キュー型に対する仮想分散キュー型の位置付け

最後に本節のまとめとして、集中キュー型、分散キュー型に対して、提案する仮想分散キュー型がどのような位置付けにあるのかについてまとめる。

仮想分散キュー型は、集中キュー型と比較して、L-1D 行列の分散化がなされている分、wakeup の高速化ができる。命令キューの制限がなく、両方式において完全に依存関係を解析できるという理想状態の場合であれば、仮想分散キュー型と集中キュー型は、Steering の効率も、パイプラインの状況も一緒となり、IPC は等しく、L-1D 行列の分散化により wakeup の高速化が可能であるため、仮想分散キュー型の方が性能は高いと考えられる。しかし、現実的には、仮想分散キュー型は、集中キュー型と比較して、パイプラインの状況で (c)(d) で生じるペナルティ分IPC が低下する。よって、仮想分散キュー型は、現実的には、wakeup の高速化の利点の方が、パイプラインの状況で (c)(d) で生じるペナルティによるIPC の低下という欠点より効果が高ければ、集中キュー型よりも性能がよいといえる。

仮想分散キュー型は、分散キュー型と比較して、Exec_Steering することでSteering の効率を高めIPC を高めるという利点が、Exec_Steering することで新たにパイプラインの状況で (b) に陥りペナルティが生じるという欠点よりも効果があれば、分散キュー型よりも性能がよいといえる。

各方式の性能を比較するために、次章では実際にIPCの測定を行う。

第5章　性能評価

Simple Scalar ツールセット (ver.2.0) に対して、第4章で述べた分散キュー型 ・集中キュー型 ・仮想分散キュー型のスーパースケーラ ・プロセッサにL-1D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の評価を行った。

5.1　評価モデル

以下に述べるクラスタ化スーパースケーラ ・プロセッサを、集中キュー型、分散キュー型、仮想分散キュー型の各方式を実装したものを評価する。

ベースとしては、MIPS R10000を用いるが、パラメーラに変更を加えてある。

整数演算、LS、FP演算のそれぞれに命令キューを持ち、(IWq,IW,WSq) = (4,8,32) である。命令/データ1次キャッシュ及びデータ統合2次キャッシュの容量は、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、7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12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、履歴長12、4Kエントリのgshareを用いた。

スーパースケーラ ・プロセッサのクラスタ数は2、実行ユニット数は各方式全て同一で、各クラスタの命令発行幅は、IWq = 2とする。

各方式のSteeringの実装に関しては、第4章で述べた通りである。分散キュー型と仮想分散キュー型のウィンドウサイズは、それぞれ WSq/2 = 16 である。

集中キュー型を基準モデルとし、基準モデルのL-1D行列の幅を縮小させなかった場合、つまりw = 32の場合のIPCに対する各方式のIPCの比率を求める。

5.2　評価結果

集中キュー型、仮想分散キュー型、分散キュー型の各方式において、基準モデルに対するIPC比率と幅wの比率を測定した結果が、図11、図12、図13である。横軸はL-1D行列の幅w、縦軸が基準モデルに対するIPCの比率である。また、図14は、基準モデルに対する各方式のIPC比率の相加平均を比較したものである。

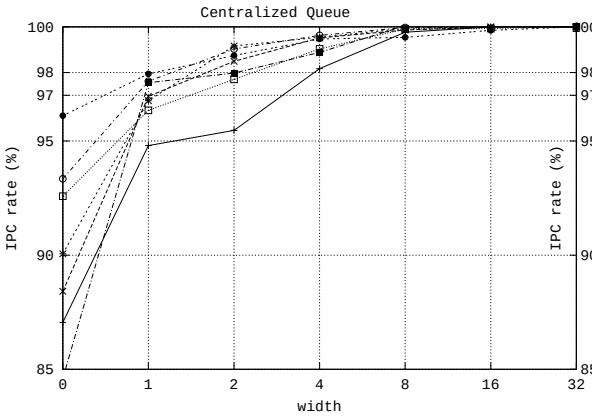


図 11: 集中キュー型 IPC 比率

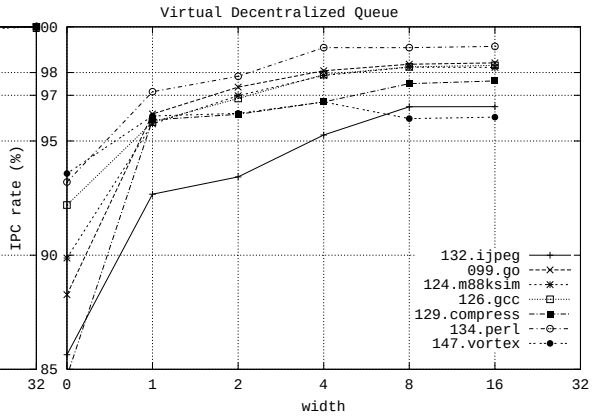


図 12: 仮想分散キュー型 IPC 比率

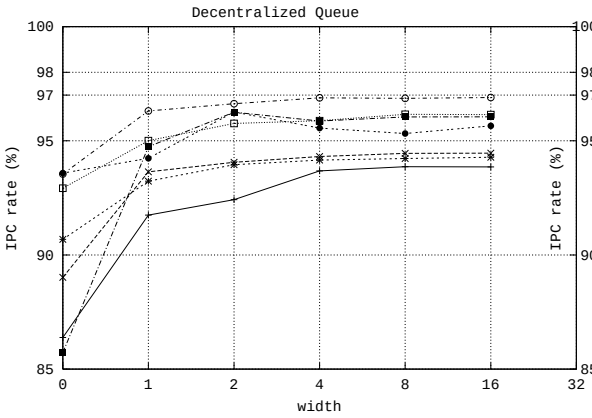


図 13: 分散キュー型 IPC 比率

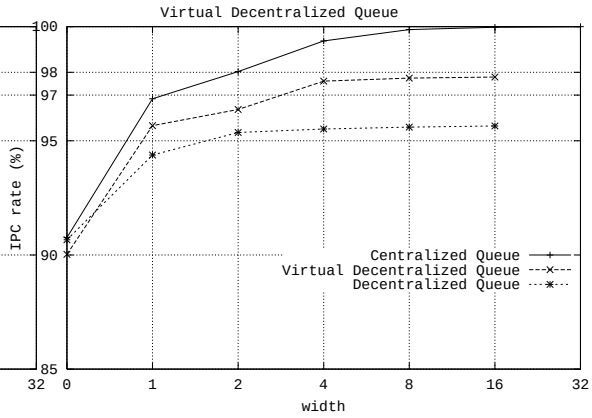


図 14: 各方式の平均 IPC 比率

基準モデルに対する各方式の **IPC** 比率の比較

各方式において L-1D 行列を縮小しない場合、つまり、集中キュー型では w = 32、分散キュー型、仮想分散キュー型では w = 16 の場合を比較する。この場合、基準モデルである集中キュー型と比較して、仮想分散キュー型では、IPC の比率は 1 % ~ 4 %程度、平均して 2.3 %程度悪化している。分散キュー型では、IPC の比率は 3.1 % ~ 5.6 %程度、平均して 4.5 %程度悪化している。

各方式の L-1D 行列の幅 w を等しくして比較してみたとき、どの幅 w においても集中キュー型と比べて、分散キュー型、仮想分散キュー型の IPC は落ちていた。これは、全実行ユニットの使用率が低く、１つのクラスタ内で十分命令処理を行えるため、依存関係を考慮に命令を各クラスタに割り当てる分散キュー型、仮想分散キュー型 Steering よりも、クラスタの番号の若い順から空いてい

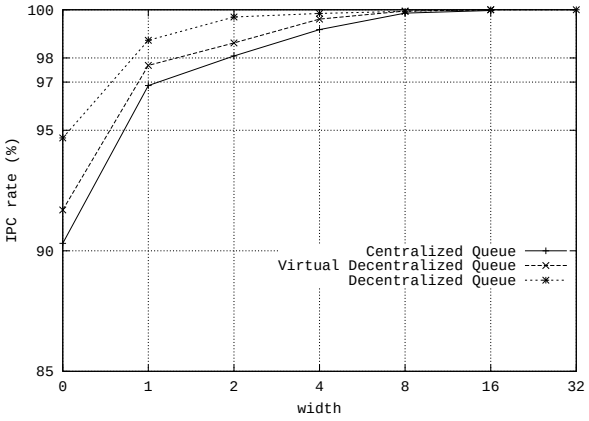


図 15: 各方式の平均 IPC の悪化の傾向

る実行ユニットを探す Steering の方が、cluster delay の頻度が少ないからだと考えられる。

なお、分散キュー型と比較して、仮想分散キュー型の方が、平均して 2 %程度 IPC が向上している。

各方式における **L-1D** 行列の幅と **IPC** の悪化の傾向

図 15 は、各方式においてそれぞれの依存行列の幅を縮小しない場合を基準モデルとし、それに対して、各方式の IPC 比率を見たものである。

結果より、全ての幅 w において、命令キューの分散化によって依存行列の幅に対する IPC の低下を抑えられている。仮想分散キュー型が分散キュー型と比較して IPC の低下が大きいのは、Dispatch_Steering の命令キューの命令発行幅が大きいため、命令キュー内にある命令同士の距離幅が大きくなりやすく、L-1D 行列の幅 w を超えペナルティが生じる場合が多くなるためだと考えられる。

第 6 章　まとめ

本稿では、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。クラスタ化スーパースケーラ ・プロセッサの命令キューが分散化されている場合、各命令キューに直接依存行列を分散化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、命令キューを分散化することで、クラスタへの命令振分け効率が悪くなるという問題に対して、仮想的に命令キューを分散化させる方式を提

案した。

今回の評価結果から、仮想分散キュー型は、集中キュー型と比べて、L-1D 行列の分散化による利点よりも、集中キュー型よりもパイプラインについてのペナルティが多く IPC が低下するという欠点の方が大きいことがわかった。これは、現在実装している Steering の性能によるものもあるが、それ以上にプログラムの命令並列性が少ないために、一度に処理できる命令数に限界があり、その結果、実行ユニットの空きが十分で、1 つのクラスタで十分命令処理を行えていることを意味する。

一方、仮想分散キュー型は、分散キュー型と比べて、wakeup の高速化の程度が低いにも関わらず、IPC は分散キュー型よりも大きかった。これは、Steering を効率よく行うことの方が、パイプラインのペナルティよりも効果が高いことを意味する。

各方式の比較をより詳細に行うためには、命令並列性を高める必要があると思われる。今後は、Load の投機的実行を実装し、命令並列性を高めて各方式の評価を行いたい。

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。そして、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から感謝致します。また、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] Palacharla, S., Jouppi, N. P. and Smith, J. E.: Quantifying the Complexity of Superscalar Processors, Technical report, Univ. of Wisconsin-Madison (1996).

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

[3] G. A. Kemp and M.Franklin : PEWs : A Decentralized Dynamic Scheduler

for ILP Processing,*Proc. Int. Conf. on Parallel Processing*(1996).

[4] 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).

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

[6] 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).

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

[8] 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: スーパースケーラのための高速な動的命令スケジューリング方式の IPC の評価, 情処研報 2001–ARC–144 (SWoPP 2001) (2001).

[9] Duaflow Project Team: Dualflow シミュレータ仕様書 (1999).