計算機アーキテクチャ 121－4
設 計 自 動 化 82－4
（1996. 12. 12）

キャッシュ・コヒーレンス制御の並列処理に関する考察

福島 直人, 五島 正裕, 森 眞一郎, 中島 浩, 富田 眞治

京都大学大学院 工学研究科

キャッシュ・メモリを採用した共有メモリ型並列計算機において、キャッシュ・コヒーレンス制御の中に存在する並列性の指摘と、その抽出により得られる性能向上に関する考察を行う。この考察の結果、異なるブロックへのコヒーレンス制御の中に存在する時間並列性はハードウェアで抽出を行うが、同一ブロックへのコヒーレンス制御における並列性の抽出は放棄するのが良いと思われる。
本稿の後半ではコヒーレンス制御の並列処理の実現方法と、その実装例を述べる。

A Study of Parallel Cache Coherence Control

Naoto FUKUSHIMA, Masahiro GOSHIMA, Shin-ichiro MORI,
Hiroshi NAKASHIMA, Shinji TOMITA

Graduate School of Engineering, Kyoto University

In this paper, we point out parallelism within cache coherence control and make a study of performance increase with extracting it. As a result of this study, we believe that it is better to extract the parallelism within cache coherence controls to different memory block, and not to extract the one within cache coherence controls to a same block.
In the latter half of this paper, we show an implementation method of parallel cache coherence control and show an example of implementation.

1　はじめに

共有メモリ型の並列計算機は、メモリ・アクセスのレイテンシの低減のために、プロセッサ毎にキャッシュ・メモリをもつ構成となっているのがほとんどである。この場合、1つの共有データが計算機内に複数存在することになるので、キャッシュ間のコヒーレンスを保つ制御（キャッシュ・コヒーレンス制御）が必要となる。

システムの性能のためには、キャッシュ・コヒーレンス制御を計算機の並列性を保ったまま行うこと（コヒーレンス制御の並列処理）が望まれる。

このための技術の一つとして、コヒーレンス制御をパイプライン処理する方法がある。本稿では、パイプライン処理を導入することにより抽出できる並列性の指摘と、それらの抽出により得られる性能向上に関する考察を行う。

コヒーレンス制御における並列性を抽出するにあたり、容易に抽出可能なものと多くの努力を払わないと抽出困難なものがある。並列性を抽出するかどうかは、抽出にかかるコストとそれにより得られる性能向上を勘案してきめることである。本考察の結果、異なるブロックへのコヒーレンス制御の時間並列性の抽出および、ソフトウェアによる空間並列性の抽出は行うが、同一ブロックへのコヒーレンス制御の並列性の抽出は行わない方が良いと思われる。

以下2節で対象とする計算機の構成を示し、3節でコヒーレンス制御の並列処理を行うことに関する考察を述べる。その後この考察に基づいて、4節でコヒーレンス制御の並列処理の実装に必要なハードウェア機構と制御を、そして5節でその実装例を示す。

2　対象とする計算機

本稿の対象計算機の構成を図1に示す。すなわち、複数のネットワークが相互にトリー状に結合された多階層構成になっており、各階層のネットワークの結合部には物理的な記憶装置が接続されている。また、各ネットワークは図1中において一番下に存在しているネットワークをL1ネットワーク、以後下からL2ネットワーク、L3ネットワークと呼ぶ。

最下位のネットワークにはプロセッサが接続されており、また各プロセッサ毎に記憶装置が取りつけられている。

各記憶装置は、あるアドレス（システム内で一意なアドレス空間におけるアドレス。以後単にアドレスと述べた場合はこの意味で用いる）に対する主記憶として、また他の物理メモリに主記憶が存在するアドレスに対するキャッシュ・メモリとして利用される。しかし、本稿においては主記憶とキャッシュの区別は存在せず、以後は記憶装置のことを単にキャッシュと呼ぶことにする。また、プロセッサに直接取りつけられているキャッシュをL1キャッシュと呼び、以後プロセッサに近い位置の順にL2キャッシュ、L3キャッシュと呼ぶ。

データの共有管理はプロセッサからのアクセス単位よりは大きいブロック単位で行われ、各キャッシュもブロック単位（物理ブロック）に区切られている。そして、各物理ブロック毎に当該ブロックの他のキャッシュとの共有状態を表すタグがついており、また、キャッシュ間のデータ転送はブロック単位で行われる。

各プロセッサからのアクセスにより開始したコヒーレンス制御はパケットの移動により進行する。パケットにはコヒーレンス制御に必要な情報、例えば状態遷移の種類やキャッシュのデータの更新を伴う場合は更新データなどが書かれている。すなわち、プロセッサからのアクセスが開始するとパケットが生成され、パケットはネットワークを経由して各キャッシュに到達する。パケットを受け取ったキャッシュは必要に応じて状態の遷移やデータの更新を行う。

3　コヒーレンス制御の並列性の抽出とその効果の考察

あるプロセッサがあるブロックに対するストアを行うと、同一ブロックを所有する他のキャッシュとの間のデータの一貫性を保つために、他を無効化またはデータを更新するキャッシュ・コヒーレンス制御が行われる。

本節ではコヒーレンス制御の経路にパイプラインを導入することによるコヒーレンス制御の並列性の抽出と、それにより得られる性能向上に関し

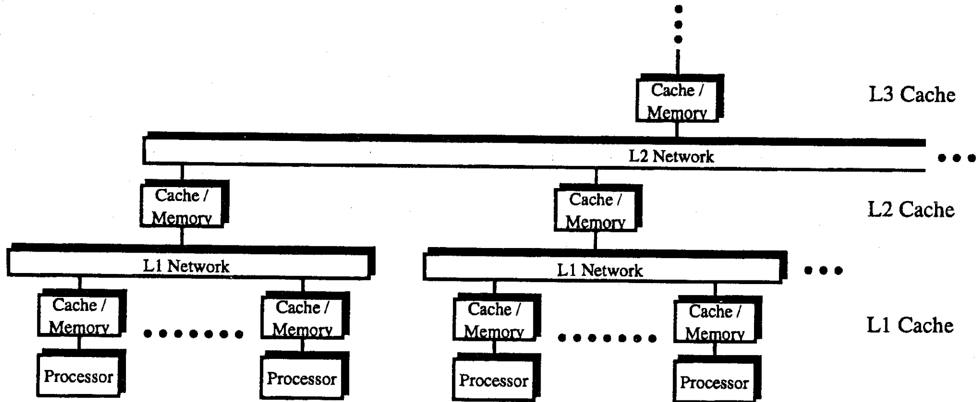


図 1: 対象とする計算機の構成

て考察を行う。

3.1 コヒーレンス制御のパイプライン処理

コヒーレンス制御をパイプライン処理するとは、プロセッサにおけるパイプラインと同様にパケットの通過経路を複数のステージに分割し、複数のコヒーレンス制御を並行に実行させることでキャッシュ･コヒーレンスの並列性を抽出する技術である。

コヒーレンス制御におけるパイプラインは、プロセッサの数だけ入り口（開始地点）があり、キャッシュの数だけ出口（終了地点）が存在する。また入り口から出口までの経路は一本道ではなく、数多くの分岐と合流が存在し、またある一つの入り口から開始して複数の出口に到達することもある（図 2)。

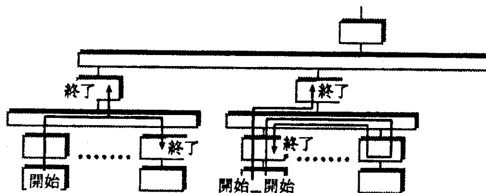


図 2: パイプラインにおける経路例

3.2 並列性の抽出とその効果の考察

コヒーレンス制御にパイプライン処理を導入する（以後単に、パイプライン処理を行うと呼ぶ）ことで、プロセッサにおけるパイプラインと同様に時間並列性を抽出できるとともに、数多くの開始地点と終了地点が存在することから、これを利用して空間並列性を抽出することもできる。

本節では、これらの並列性の抽出とそれにより得られる効果について考察する。

空間並列性の抽出　コヒーレンス制御の空間並列性を抽出するとは、各ブロックの共有状態の局所性を高めることである。

例えば考えられるのは、データの通信はなるべく近傍のプロセッサの間でしか起こらないようにプログラムのプロセッサへのマッピングを行うことである。このためには、ユーザがプログラム中における通信特性を調べる必要がある。

このような空間並列性の向上を図ることにより、コヒーレンス制御の通過経路が短くなる。さらに他のコヒーレンス制御との間の通過経路の重なりが減るので、コヒーレンス制御の進行が資源競合によって停止される確率も減る。これらのことから各コヒーレンス制御に要する時間を短くすることができる。

時間並列性の抽出　パイプラインにおいて依存関係が存在しない、すなわちある一つのブロックに対するコヒーレンス制御がパイプラインの中に複数存在しないときは、あるコヒーレンス制御の進行を妨げるものは、複数のコヒーレンス制御の通過経路の重なりによる資源競合のみである。空間

並列性の抽出の際に述べたように、この資源競合は減らすことが可能である。したがって、非常に高い並列性が期待できる。

一方で依存関係が存在しているときは、基本的にはどちらか一方の制御は他方の完了を待つ必要が生ずる。しかし、プロセッサにおけるパイプラインにおいては、レジスタ・リネーミングなどの技術により依存関係が存在する二つの命令を並行実行することが可能な場合があったように、コヒーレンス制御におけるパイプラインにおいてもこれと同様な技術の存在は考えられる。

しかし、同一ブロックに対する複数のコヒーレンス制御が存在する状況において、たとえそれらを並行実行させたとしても、それにより得られる効果は非常に少ないと思われる。その理由を以下で述べる。

コヒーレンス制御の依存関係は同一ブロックへのL1キャッシュにミスした各ロードと他のキャッシュに有効なデータが存在し得るときに行われた各ストアの間で生じ得る[1]。

まず、あるアドレスへのロードがL1キャッシュに対してミスすることが複数プロセッサにおいてほぼ同時に起こるのは事実上、プログラムの最初の方のインストラクション・フェッチの時くらいである。すなわち、恒常的に複数のプロセッサからほぼ同時に同一ブロックへのロード・ミスが行われることはない。

一方、複数のキャッシュにデータが共有されている状態で、あるプロセッサからのストアと別のプロセッサからのストアまたはロード・ミスがほぼ同時に起こり得るのはこのブロックがフォールス・シェアリングされている場合が多い。なぜなら、複数のプロセッサが同一アドレスに対してアクセスを行い、かつそのうち少なくとも一つがストアであるときは、このストアはクリティカル・セクションであることが多いので、他の同一アドレスへの各アクセスのタイミングをずらさなければならない。すなわち、複数のプロセッサがほぼ同時に同じアドレスへのアクセスを行いかつそのうち少なくとも一つはストアであるアクセスを行うことは正しいプログラムにおいては少ないと思われる。このように、あるブロックに対して複数のプロセッサからほぼ同時にアクセスが行われるのはそのほとんどがプログラムが間違っているか、フォールス・シェアリングによる場合である。

フォールス・シェアリングの発生は計算機の性能に大きな影響を与えることが知られている。この現象への対策としてはフォールス・シェアリングが発生しないようなプログラムを記述するか、フォールス・シェアリングが発生しても性能が低下しない機構をハードウェアで備えるかのどちらかである。しかし本稿では、ハードウェアによる手法より、フォールス・シェアリングを発生させない方が計算機の性能向上を図ることができ、またこれは近い将来コンパイラ技術の進歩によって実現可能であると仮定する。つまり、ハードウェアの設計にはフォールス・シェアリングは発生しないという前提を採用することにする。

また、間違ったプログラムを記述するのはプログラマの責任であり、そのようなプログラムに対してハードウェアで性能向上を求める必要性はない。

以上のように、フォールス・シェアリングが発生しないとすると同一ブロックへのコヒーレンス制御がほぼ同時に発生することはほとんどなく、また発生したとしても性能向上を求められない局面であることがほとんどである。

したがって、これらの制御に対しては結果の正当性だけを保証したなるべく簡単な実装方法を採用すべきであり、開発の努力は他の部分に回されるべきであると思われる。

4 コヒーレンス制御の並列処理の実現

前節の考察を基に、コヒーレンス制御の並列処理を行うために必要なハードウェアの機構及び制御について述べる。

4.1 パイプライン処理のための機構

本稿におけるコヒーレンス制御の並列処理はすべて、コヒーレンス制御がパイプライン処理されることを前提にしている。

このパイプライニングを行うためのハードウェ

[1]ロードがL1キャッシュにヒットした場合とストアを行ったときと、キャッシュの状態が（他のキャッシュには同一ブロックのデータは存在しないことを保証する）Exclusive状態の時には、コヒーレンス制御は発生しない。

ア機構を図3に示す。すなわち、各キャッシュと上位ネットワーク、下位ネットワークの間にそれぞれパケットの送信バッファと受信バッファを設けてパケットをバッファリングすることにより、コヒーレンス制御のパイプライン処理を可能にする。

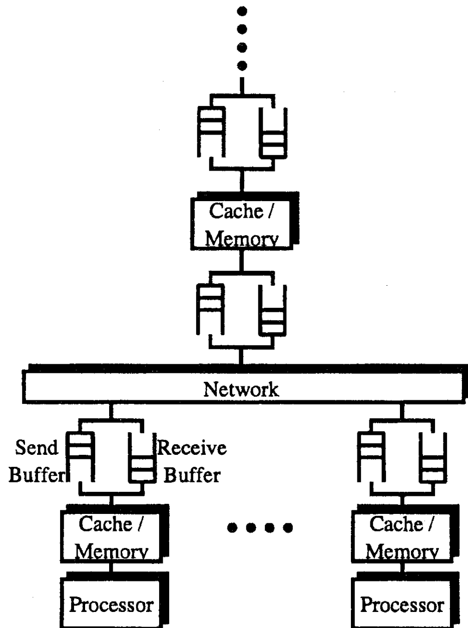


図 3: パイプライニングのための機構

4.2 コヒーレンス制御の並列処理の実現方針

先にも述べたように、コヒーレンス制御のパイプライニングを行う際には同一ブロックへのコヒーレンス制御が同時にパイプライン中に存在することがある。

前節で、パイプラインにおける依存関係の発生時には、性能向上の追求により得られるメリットは少ないので、なるべく簡単な実装で行うのが良いと述べた。すなわち、理想としては、異なるブロックに対するコヒーレンス制御はパイプライニングによりオーバラップさせて処理し、同一ブロックに対するコヒーレンス制御は他の同一ブロックへの制御に対して排他的に処理することである。

しかし、これをそのまま実装しようとすると、並列性を大きく損なう結果になる。なぜなら、このためには各コヒーレンス制御が発生した際、その制御をパイプラインに投入して良いかどうかを知らなければならない。このためには各ブロックに対する進行中のコヒーレンス制御の存在の有無をどこかで管理しておき、コヒーレンス制御の発生の度に管理地点に対して開始要求を行い、それが認められてから実際のコヒーレンス制御を開始する必要がある。しかし、これでは結局コヒーレンス・パイプラインに大量のバブルが発生することになる。

前節でも述べたように、同一ブロックに対するコヒーレンス制御は、そのほとんどはその処理に時間がかかってもシステムの性能には影響をおよぼさない。一方、コヒーレンス制御の発生時に同一ブロックへの制御の存在を想定した動作を行うと上で述べたような大量のバブルが発生する。

このことから、投機的なコヒーレンス制御を行う、すなわちコヒーレンス制御の発生後同一ブロックへのコヒーレンス制御は存在しないと仮定してパイプラインに投入し、同一ブロックへのコヒーレンス制御が既に投入されていたことが判明したときは、先行制御が行われた後に大きなペナルティを覚悟でつじつまを合わせる制御を行うのが性能の上では望ましい。

つじつまを合わせる制御は複雑ではあるが、投機的な状態遷移を行わないことによるペナルティは大きいので、この制御は行う方が良いと思われる。しかし、同一ブロックへのコヒーレンス制御が自分以外にも進行中であると知ったあとは、後につじつま合わせをしてまでコヒーレンス制御を進行させるメリットはほとんどない。すなわち、同一ブロックへのコヒーレンス制御が自分以外にもに存在していることを知った後は先行のコヒーレンス制御が先に行われることが確定できるまで後続のコヒーレンス制御の進行を停止させる。

4.3 コヒーレンス制御の並列処理の実現方法

上で述べた方法によりコヒーレンス制御の並列処理を実現するためには

1. 各コヒーレンス制御が、自分より前のコヒーレンス制御の存在を知ることができ、

2. これを知ったあと、前のコヒーレンス制御が各キャッシュに対して先に行われることが確定するまで、自分の進行を停止させ、

3. さらに、失敗した投機的状態遷移のつじつまを合わせる

三つのことが必要となる。ただし、つじつまあわせの機構は一般的な議論が困難であるから、本稿ではこれに関する議論は避ける。そのかわり、単純な例を 5節で示す。

以下、各機構/制御の実現方法を述べる。

自分より前のコヒーレンス制御の存在を知る機構 先にも述べたように、コヒーレンス制御の実体はパケットの移動である。このことから、あるコヒーレンス制御のパケットが計算機内のある地点を通過したとき、その地点を既に、現在進行中の他のコヒーレンス制御によるパケットが通過したかどうかを知ることができる機構が存在すればよい。以後、先行コヒーレンス制御の存在を知ることができる地点をシリアライズ・ポイントと呼ぶ。

この例を図4に示す。なおこの図以降、バッファはキャッシュの中に含まれているとする。この図は、まずP_1のメモリ・アクセスによるコヒーレンス制御が (1) に示す経路を通り、その後、P_2によるメモリ・アクセスによるコヒーレンス制御が (2) に示す経路を通るとする。この時、(1) が図の矢印で示した地点を通過する際にそのコヒーレンス制御の対象となるブロック・アドレスにフラグを立てておく機構が存在すると、(2) がこの地点を通過する際に、先行コヒーレンス制御の存在を知ることができる。

ところで、この手法により各コヒーレンス制御が自分自身より先のコヒーレンス制御があることを知るためには、先行コヒーレンス制御が通過した地点を自分自身も通過しなければならない。もし、シリアライズ・ポイントが各ブロックに対して計算機内に一点しかないとすると、各コヒーレンス制御はこの地点を必ず経由しなければならない。これは、コヒーレンス制御の経路が長くなることなので、性能上望ましくない。このことから、シリアライズ・ポイントは計算機内に幾つも存在させ、同時に存在し得る任意の二つのコヒーレンス制御がこのうちの少なくともどれか一つをともに通過するようにすれば、コヒーレンス制御の経路が長くなることは避けられる。

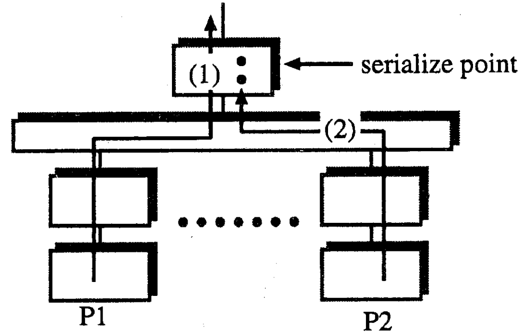


図 4: シリアライズ

コヒーレンス制御の進行を停止させる制御 シリアライズ・ポイントを通過した時点で、上で述べた機構により同一ブロックに対する別のコヒーレンス制御が進行中であることを知ったとき、もしあるキャッシュに対して後続コヒーレンス制御の方が先に状態遷移が行われることがあれば、この危険が解消されるまで、後続の制御の通過を待たせれば良い。

状態遷移順序の逆転はパケットの通過経路の違いや、パケットの追い越しの発生などの理由で発生する。この例を図5に示す。P_1,P_3からのメモリ・アクセスに伴うコヒーレンス制御がそれぞれ矢印に示される経路を通過し、P_1における状態遷移はそれぞれ丸印で示されたときに行われるとする。また、この計算機において、このネットワークがシリアライズ・ポイントに定められているとする。

今、P_1のコヒーレンス制御が先にネットワークを通過したとする。P_1の状態遷移はコヒーレンス制御がP_2を経由した後、P_1に戻ってきたときに行われる。よって、P_1のコヒーレンス制御がP_2で処理されているときにP_3がネットワークを通過すると、P_1の状態遷移における状態遷移の順序は先行/後続のコヒーレンス制御の間で逆転する。よって、P_1のコヒーレンス制御がP_1に戻ってくるまでは、P_3のコヒーレンス制御のネットワークの通過を停止させる制御を行う。

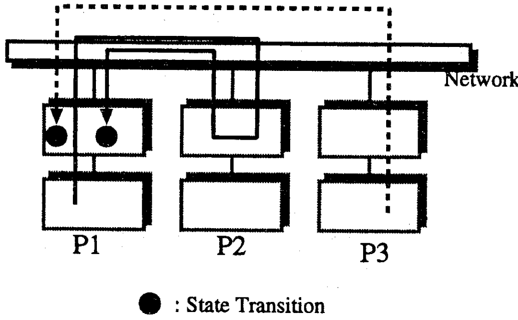


図 5: 状態遷移の順が逆転する例

5 実装例

前節で述べた機構と制御が実装の際に、どのように利用されるかを示す。

実装対象の計算機を図 6に示す。すなわち、各プロセッサにはプライベートなキャッシュがついており、これらと主記憶が一本のバスで接続されている。また、実装を行うキャッシュ・コヒーレンス・プロトコルは図 7に示すバークレイ・プロトコル [1] とする。

ロード・ミス→ストア この場合のコヒーレンス制御の流れを図 8に示す。各キャッシュの最初の状態は P_1が Valid、P_2は Invalid とする。まず、P_2がロードを行い、それによるパケットが主記憶に到達する (1)。それとほぼ同時に P_1がストアを行い、それによるパケットがバスを通過しようとする (2)。ところで、この計算機においてバスをシリアライズ・ポイントの一つに設定し、P_2からのパケットがバスを通過する際に当該ブロックへのコヒーレンス制御の進行を示すフラグを立てていると、P_1はここで先行コヒーレンス制御の存在を知ることができる。もし、主記憶からの応答がある前に、P_1からのパケットを直ちに通過させると P_2における状態遷移は、P_1からのコヒーレンス制御による遷移 (Invalid → Invalid) の方が先に起こる。よって、P_1からのパケットはバスの通過を待たせる。その後主記憶から P_2に応答が返ってくると (3)、シリアライズ・ポイントのフラグを解除し、P_1からのパケットを通過させる (4)。なお、P_1の状態は (2) の段階で投機的に Exclusive Dirty になるが、結局の最終状態も Exclusive Dirty なので、つじつま合わせの制御は必要ない。

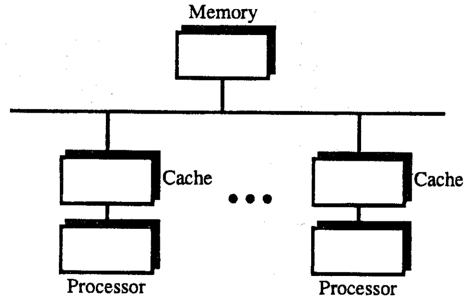
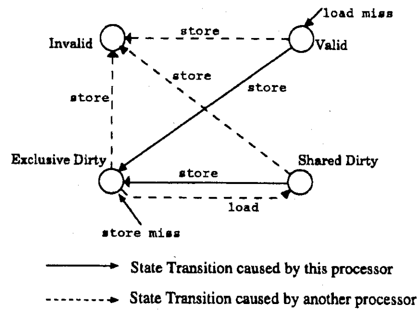


図 6: 実装対象の計算機



Invalid 当該キャッシュは無効化されている。

Valid 当該キャッシュに有効なデータが存在している。

Shared Dirty 当該キャッシュの内容は主記憶と異なる。
他のプロセッサからの読みだし要求に答える義務がある。

Exclusive Dirty 当該キャッシュの内容は主記憶と異なる。
他の物理メモリには当該アドレスのデータが
存在しないことが保証されている。

(注)：Exclusive Dirty は文献 [1] においてはただの Dirty と書かれている。しかし、本稿においては他のキャッシュに有効なデータが存在しないことを強調するため Exclusive Dirty と記述する。

図 7: バークレイ・プロトコル

シリアライズ・ポイントがバスでない例 先の例ではバスがシリアライズ・ポイントであったが、シリアライズ・ポイントがこれとは異なる例を示す。

今、P_1の状態がExclusive Dirty であるときに P_2がロードを行いそれによるパケットが P_1のキャッシュの直前に到達したとし (1)、これとほぼ同時に P_1がストアを行ったとすると (2)、この時は P_1からのコヒーレンス制御がバスには現れないため、キャッシュがシリアライズ・ポイントとなる。

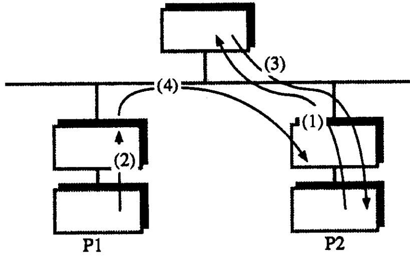


図 8: ロード・ミス→ストア

このように、シリアライズ・ポイントは各メモリ・アクセスが行われたときのキャッシュの状態によって異なるときがある。

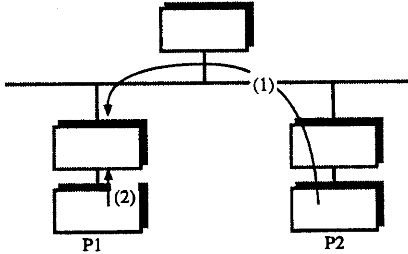


図 9: シリアライズ・ポイントがバスではない例

つじつま合わせが行われる例 投機的状態遷移の失敗によりつじつま合わせが行われる例を示す。

キャッシュの状態は最初 P_1, P_2ともに Valid であるとする。この時ほぼ同時に P_1と P_2がストアを行うと (1)(2)、キャッシュの状態はともに投機的に Exclusive Dirty となる。この場合の正しい最終状態はどちらかが Exclusive Dirty になることであるので、これは投機的な状態遷移が失敗していることを示す。

以下で、シリアライズ・ポイントをバスとした場合のつじつま合わせの手順を示す。まず、P_2からのコヒーレンス制御がバスを通過すると P_1の状態は Invalid になる (3)。この後、P_1から発せられるコヒーレンス制御により、P_2を無効化するだけでなく、P_1の状態も Exclusive Dirty にもどす (4)。

これは、P_1から発せられるコヒーレンス制御をあたかも始めからストア・ミスにより発せられたのと同じように扱うことで達成できる。ストア・ミス時には他を無効化し、自分を Exclusive Dirty にするコヒーレンス制御が行われる。すなわち、(3) が行われた時点で P_1のコヒーレンス制御の種類をストア・ミスによるものに変更することで、つじつま合わせが行われる。

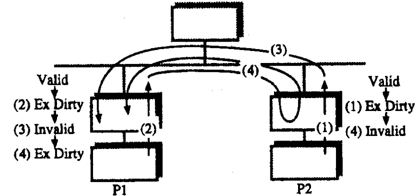


図 10: つじつま合わせが行われる例

6 おわりに

本稿はコヒーレンス・パイプラインの採用により、キャッシュ・コヒーレンス制御の中に現れる並列性を指摘し、その並列性抽出により得られる性能向上に関する考察を行った。この結果、同一ブロックへのコヒーレンス制御における並列性の抽出は放棄のが良いと思われる。その後、この考察に基づいてキャッシュ・コヒーレンス制御の並列処理の実現方法とその実装例を示した。

しかし本稿では、コヒーレンス制御の並列処理を実装する際はコヒーレンス制御の投機的実行が性能上必要となるが、その失敗によるつじつま合わせの機構についての一般的な議論は避けた。並列処理を実現するにあたり、このつじつま合わせが最も難しい部分であり、これをどう実現するかを考察することが今後の課題と言える。

参考文献

[1] 富田 眞治 著：コンピュータアーキテクチャI. 丸善株式会社. 1994 年。