２．集中メモリ型並列コンピュータ

2.1　ユニプロセッサからマルチコアへ

ノート PC、スマフォ、タブレットに搭載されている並列コンピュータの多くは小規模な集中メモリ型コンピュータ（UMA：Uniform Memory Access model）である。集中メモリ型並列コンピュータは、ユニプロセッサの自然な延長線上にある。このため、少しだけユニプロセッサの復習をしておこう。

図１にユニプロセッサの構成を示す。CPU に接続されるメモリは、高速大容量が理想なのだが、一般的に高速なメモリはビット単価が高く、容量の大きなメモリはアクセス（読み書き）が遅い。このため、複数の種類のメモリを使って記憶の階層を作る。すなわち、高速小容量のメモリに良く使うデータ（命令）を入れておき、アクセスしたいデータが存在しない場合に限って、階層の下から取ってくる。この高速小容量のメモリをキャッシュ（Cache）と呼ぶ。キャッシュ上に目標のデータが存在する場合をヒットと呼び、存在しないで下の階層に取りに行くに行くことをミスヒットあるいはミスと呼ぶ。最近の CPU は、ミスした時に取りに行く時間を減らすために、キャッシュ自体を複数の階層から構成し、CPU に近いものから順に L1,L2,L3 と番号を付ける。多くの CPU では L2 あるいは L3 までは同じチップ上に置く。これをオンチップキャッシュと呼び、CPU の論理回路と同じ半導体プロセスで作ることが可能な SRAM（Static Random Access Memory）で構成する。チップ外にもボード上に大容量の SRAM チップにより L3（L4）キャッシュを置き、オンボードキャッシュと呼ぶ。

オンボードキャッシュ上にアクセスするデータがなければ、主記憶がアクセスされる。主記憶は大容量の DRAM（Dynamic Random Access Memory）で構成される。最近の DRAM は、DDR SDRAM(Double Data Rate Synchronous DRAM)と呼ぶ、クロックの立ち上がり立ち下り両方の変化に同期してデータを転送する方式をとっており、高速な連続転送が可能である。記憶の階層は、主記憶までは CPU から見て透過（Transparent）である。すなわち、CPU から見ると、キャッシュは見えない存在で、ヒットすれば高速、ミスすれば遅くなる。一方、主記憶上にデータがない場合は、補助記憶に取りに行くことになる。補助記憶は従来、主に磁気的な記憶方式であるハードディスクを用いたが最近はフラッシュメモリを用いた SSD(Solid State Drive)の利用が増えている。主記憶と補助記憶の間のやり取りは、OS(Operating System)によって管理される。これを仮想記憶（Virtual Memory）と呼ぶ。ここではこの辺はあまり関係しないために復習を省略する。キャッシュの動作については後にまた触れる。

図１は論理的な記憶の階層を示すが、実際には図２のように、チップ外では複数のスイッチ（バス）により、キャッシュ、メモリ、入出力が互いに接続されている。高速性を要求される DRAM やグラフィックス装置は高速スイッチで、入出力装置は低速スイッチで接続される。前者を Intel の用語では North Bridge、後者を South Bridge と呼ぶ。これらのスイッチは後に解説するクロスバスイッチの一種と考えて良い。

さて、この構成を持つユニプロセッサをマルチコアにするにはどうすれば良いだろうか？まず図 3(a)に示すようにキャッシュを共有する方法が考えられる。しかし、(a) のように L1 キャッシュを共有する場合、アクセスの混雑が生じる。同時に読み出し可能なポートを複数持つマルチポートメモリを使うという方法も考えられるが、ポート数の多いマルチポートメモリはコストが大きく、高速動作も難しい。そもそも L1 キャッシュは CPU の高速動作に付いていくため、CPU の近くに配置しなければならない。このため図 2 (a)は、ほとんど現実的に実現不可能である。

では、(b)に示すように分散してはどうだろうか？この場合、アクセスは分散され、ユニプロセッサと同じ程度の高速性を実現可能だが、困った問題が生じる。キャッシュは CPU から見て透過であるため、それぞれの CPU がそれぞれのキャッシュにデータを書き込むと、それは他の CPU から見ることができない。図 2 (b) では、たまたま L3 キャッシュまでデータが書き戻され、それが他の CPU の L2,L1 に持ってこられた場合に限り、データの受け渡しができることになる。同じ番地であってもデータがてんでんばらばらでは、共有メモリとは言えない。この問題をキャッシュ一貫性 (コヒーレンス) 問題と呼ぶ。この問題を解決するため、L1 キャッシュのみを個別に持たせ、この間に一貫性を取るための特殊な機構を装備する。これをスヌープキャッシュと呼ぶ。この構成を (c) に示す。以下、まず (c) を実現するために CPU、キャッシュ間を接続するバス、クロスバについて解説する。次に一貫性問題を解決するスヌープキャッシュを紹介する。

2.2 共有バス

2.2.1 共有バスの基本

共有バスは、CPU、メモリを接続する最も基本的な方式である。バス (BUS) とは、複数の信号を伝搬する信号線の束のことを指し、路線バスと全く同じ意味である。普通のディジタル信号は一時期に一種類の信号しか伝搬できないので、コンピュータでバスと言ったら通常、時分割バスを指す。また、バス上の信号を複数モジュールで受けることは簡単なので、普通のバスは多数の受信者が居るマルチドロップ方式である。従来、バスは単純な信号線の束であり、ここに特殊な出力を用いて複数の送信者が信号を載せた。特殊というのは、通常のディジタル信号は複数の出力を接続することが電気的に許されないためである。このため、オープンドレインや 3 ステートゲートなど、複数の出力を接続して、時分割で信号を線路上に載せるための特殊な出力が必要になった。古典的なバスとしては、抵抗で電源にプルアップされた線路上に、基板を挿すためのスロットが配置されているバックプレーンバスが挙げられる。バックプレーンとは筐体の裏側を指す。古典的なコンピュータはバックプレーンに様々なモジュールを挿すことで、バス接続を実現した。

しかし現在、チップ内部ではバスは線路ではなく、論理ゲートで作られており、一種の選択回路、あるいはスイッチと考えて良い。両者を対比して図 4 に示す。論理ゲートで作られた場合でも一時期に一つの信号線しか載せない点で古典的なバスと機能は同じである。

通常、バスは以下の手順で利用する。①調停作業（アービトレーション）を行って、バスの利用権を獲得する（バスマスタになる）。②アドレストランザクションを行いアドレスをバス上に流す。③データを連続転送する。④終了トランザクションにより一連の転送を終了する。バス上の転送の一つの区切りをバストランザクションと呼ぶ。通常、バスはアドレスとデータを時分割転送するため、まずアドレストランザクションを、次にデータトランザクションを行うことになる。

2.2.2 アービトレーション

ユニプロセッサのバスでは、通常 CPU がバスの制御を司るバスマスタである。しかし、マルチコアではコアが複数あるため、誰がバスマスタになるのかを決める必要がある。これを行う操作を調停（アービトレーション）と呼び、このための回路を調停器（アービタ）と呼ぶ。コアはまずアービタに対して要求信号を出し、許可が出るとはじめてバスマスタになってバスを利用することができる。バスマスタ以外のモジュールをスレーブと呼ぶ。アービタは本質的に、優先順位付きエンコーダであり、複数の要求信号から優先順位に従ってどれかを選ぶ働きをするハードウェアである。バックプレーンバス上では、分散して決定する分散アービタが用いられたが、チップ内部で用いるアービタは集中型で、コア数が多いと、高速動作のためにツリー構造を持つ必要がある。代表的な回路を図 5 に示す。この図では数字の小さいものほど優先順位が高い。

アービタに付いた優先順位が固定であると、優先順位の低いコアは、要求が長時間連続的に受け入れなれずに待ち状態になることがある。これを飢餓状態（スタベーション）と呼ぶ。マルチコアでは基本的に優先順位は公平であることが望ましいので、アービトレーションの毎に順位を巡回させることが多い。この方法をラウンドロビンと呼ぶ。

アービタの動作時間が転送時間の増加に結び付かないように、動作は転送と重ね合わせ（オーバーラップ）して行う。すなわち、バスの転送時にもう次のバスマスタを調停しておくのである。この操作により、調停の時間は顕在化しなくて済む。この様子を図 6 に示す。

2.2.3 バストランザクション

最近のバスは高速性が重視されるため、単一のクロックに同期して転送を行う同期バス方式を採用する場合が多い。この方式では転送の最初と最後にストローブとアクノリッジと呼ぶ 2 本のハンドシェーク線を利用する。バスマスタは、アドレス・データ線に、アドレスを載せてから、ストローブをＨレベルからＬレベルに変化して、バス上のデータが有効であることを知らせる。スレーブは、アドレスを受け取ったら、アクノリッジをＬレベルからＨにしてこれをマスタに知らせる。これで両者の準備ができたことがわかるので、次からはクロックに同期して次々とデータが転送される。所定の数送る（あるいはストローブ線をＨレベルにする）と、アクノリッジ線をＬにして転送を終了する。この様子を図 7 に示す。最後に、エラー修正用コードなどを送る場合もある。

図 7 の例は受信側が一つを想定したが、マルチドロップ方式が一般的なバスでは受信者

が複数存在する場合が多い。この場合、オンチップ方式ならば、それぞれのモジュールがアクノリッジ線を持ち、それぞれの受信終了をマスタに知らせることができる。マスタは全員の準備ができたら、データの連続転送を始める。
一般的には、アドレス、データのバストランザクションは連続して行われる。しかし、これは、スレーブの準備に時間が掛かる（遅いメモリの読み出しなど）場合は、連続してバスを占領することを意味する。そこで、アドレスのみ転送して、次のデータを転送する前にバスを分割して別の転送を入れてやる場合がある。これをスプリットトランザクションと呼ぶ。この様子を図 8 に示す。スプリットトランザクションによりバスの利用効率を上げることができる。バストランザクションを一つずつ独立に送る方式を推し進めるとパケット転送に近づいていく。

2.3　クロスバスイッチ

クロスバスイッチは、バスの集合体と考えられる。図 9 に示すように複数のキャッシュに対して複数のコアからバスを引き、交点に ON/OFF 可能なスイッチを装備したものがクロスバスイッチである。クロスバスイッチは、宛先が違えば独立の経路を確保することができ、衝突しない。これをノンブロッキング性と呼ぶ。これを実現するため、n 入力 m 出力の場合、n×m の交点スイッチ（クロスポイント）が必要になる。

しかし、クロスバスイッチといえども、宛先が同じならば、衝突を起こす。このためバス同様アービタを使ってどれか一つの入力を選ぶ必要がある。この間、選ばれなかった方はどうなるか、というとバッファあるいは FIFO（First In First Out）と呼ばれる小規模なメモリに一時的に保存しておき、経路が空いたら通してやる。この様子を図 10 に示す。実は同様の機構はバスにも装備することができる。

バッファは、通常、クロスバスイッチの入力側に用意する入力バッファ方式を取るが、出力に用意する出力バッファ方式、交点に設けるクロスポイント方式もある。しかし前者は転送周波数の n 倍の動作周波数を用いる必要があり、交点に設ける方式はハードウェア量が多すぎる。

クロスバを用いてマルチコアとキャッシュを接続する場合、当然のことながらキャッシュが複数の塊に分割されていなければならない。このためには、アドレスを分散させるため、一定の単位（キャッシュブロックなど）で、順番にアドレス付けを行う。これをインターリーブと呼ぶ。この様子

クロスバは大規模になるとハートウェア量が大きくなる。n×mのクロスポイントが注目される場合が多いが、実際はアービタやバッファのコストも大きい。

2.3　スヌープキャッシュ

2.3.1 キャッシュの復習

次にキャッシュ一貫性問題を検討しよう。それぞれのＣＰＵがキャッシュを個別に持つ

とする。あるキャッシュに書き込んだデータは他のＣＰＵのキャッシュには知らされないため、何もしないとてんでんばらばらのデータが同じアドレスに書かれてしまうことになる。この様子を図 11 に示す。共有バスを利用してこれを解決するのがスヌープ（snoop）キャッシュである。

スヌープキャッシュを理解するために、簡単にキャッシュを復習しよう。スヌープキャッシュ機能は L1 キャッシュに装備し、L1 キャッシュのミスには L2 キャッシュが対応するが、ここでは混乱を避けるために、キャッシュにミスしたらメモリが対応するという古典的なシステムを念頭に置いて解説する。ここでの説明の共有メモリは L2 共有キャッシュと置き換えて考えてほしい。

コンピュータは、近くのアドレスがアクセスされやすいという空間的局所性の原則がある。キャッシュでは、一定の大きさ（32Ｂ－１２８Ｂ）の連続したアドレスのデータの塊をブロックと呼び、ブロック単位でデータをごっそり移動することで、空間的局所性を利用する。最も簡単な割り付け法であるダイレクトマップ方式は、メインメモリのブロック番号を、順にキャッシュのブロックに割り当てていき、一巡したらもとに戻ることを繰り返す。この方法では、ブロックサイズを 2^B、キャッシュに入るブロック数を 2^C、メインメモリに入るキャッシュのブロック数を 2^M である場合、CPU からのアドレスの下位 B ビットをブロック内アドレス、次の C ビットをキャッシュの位置を指し示すインデックス、上位の M-C ビットをメモリのブロック番号を示すタグ（キー）と呼ぶ。

キャッシュを管理するためにキャッシュディレクトリと呼ばれるテーブルを用意する。このテーブルはキャッシュのブロック数と等しい 2^C エントリを持っており、このエントリに現在、キャッシュ中に置かれているメモリのブロックの番号（タグ）を入れておく。CPU からのアドレス中のインデックスに相当する C ビットで、キャッシュブロックとディレクトリを同時に検索する。読み出したタグと、CPU から送られてきた上位 M-C ビット目が一致すれば、アクセスするブロックがキャッシュ中に存在することがわかる。これがヒットした場合であり、キャッシュから読みだしたデータはそのまま CPU に送ら、一致しなければミスとなる。この様子を図 12 に示す。キャッシュディレクトリの各エントリには、対応するブロックが有効かを示すビット（Valid:有効ビット）を付けておく。Valid ビットは 0 に初期化し、有効なブロックが入る際に 1 にセットする。

ダイレクトマップ方式ではメインメモリのブロックの割り付けられるキャッシュの場所は一か所に決まってしまうため、インデックスがたまたま同じになる二つのブロックは競合（衝突）を起こして同時にキャッシュに入れることができない。そこで、いくつかのブロックをセット（集合）として、集合単位に割り付けるセットアソシアティブと呼ばれる方法を使うのが一般的である。セット内のブロック数をウェイと呼び、2、４、8 ウェイを持たせる方法が良く用いられる。この場合、ディレクトリはウェイ数分だけ持たせて、同時に同じインデックスで検索する。

読み出しアクセスがキャッシュにヒットする場合、読みだしたデータをＣＰＵに持って

来るが、ミスすれば上位階層のキャッシュに取りに行く必要がある。取ってきたブロックは、今までそこにあったブロックを置き換える（あるいは書きつぶす)。この操作をリプレイスと呼ぶ。セットアソシアティブ方式では、どこのブロックと置き換えたら良いか判断する必要があるが、最近アクセスされなかったブロックを選ぶ LRU（Least Recently Used）という方法が一般的である。

一方、書き込みがヒットした場合、キャッシュに書いたデータを主記憶にそのまま送って更新するライトスルー方式と、キャッシュのみに書き込み、主記憶と違ったままにしておくライトバック方式がある。ライトバック方式では、後にそのブロックが置き換えられる際に、書き込みがあった場合には、ブロックの中身を主記憶に書き戻さなければならない。この必要性を判別するために、キャッシュディレクトリのすべてのエントリに Dirty bit と呼ばれる 1 ビットを付けて、最初にブロックを持ってきた時に 0：Clean に初期化しておく。そして、そのブロックに書き込みが起きた際に、このビットを 1 ：Dirty にし、主記憶と状態が異なっていることを示す。置き換えに際しては Dirty なビットが立っているブロックだけを書き戻すことにより、無駄な書き込みを避けることができる。

書き込みがミスした場合は、ブロックをキャッシュに持って来てから書く方法（ライトアロケート）と、持ってこないで、主記憶だけを書き換える方法（ライトノンアロケート）の二つがある。一般的に、ライトバックはライトアロケートの方針を取ることが多いが、ライトスルーキャッシュには、アロケート、ノンアロケートの二つの可能性がある。

2.3.2 スヌープキャッシュの元祖

スヌープキャッシュはライトスルー型から生まれた。図 13a）に示すようにマルチコアの共有バス型で、L1 キャッシュにライトスルー型を使うことを考えよう。図 13b)に示すように、ある CPU（P1）からの書きこみがヒットすると、そのアドレスとデータは共有バスを流れて、共有に送られる。ここで、共有バス上はアドレスとデータが流れていくことになり、他のプロセッサのキャッシュもこのアドレスの内容をバス経由で見ることができる。ここでキャッシュディレクトリをバス側にも設けておけば、アドレストランザクション毎にアドレスを検索することで、自分が同一アドレスのブロックを持っているかどうかを知ることができる。持っているとわかったら、このブロックに CPU が書き込んでしまったことがわかるので、Valid ビットを 0 にして、そのブロックを無効化する。無効化されたブロックに対して再びアクセスを行うとミスが発生して、主記憶から更新されたブロックを読み出されるので一致性は保たれる。　すべてのキャッシュが自分でアドレスを見て（盗み見て)、自律分散的に Valid ビットを 0 にすることからスヌープ（盗み見る）キャッシュと呼ばれる。(チャーリーブラウンに出てくる犬の名前と語源は同じである)。

ライトスルーキャッシュは、データを主記憶に送るので、無効化する代わりにこのデータをキャッシュに直接取り込んでしまうことで一致性を確保することができる。これを更新型スヌープキャッシュと呼び、無効化型スヌープキャッシュと区別する。更新型は、いわば

積極戦略であり、優れた点も持っているが共有バスを頻繁に使う必要が生じる問題点がある。

2.3.2 基本プロトコル

このようにスヌープキャッシュはライトスルー型のキャッシュを考えると自然な方法である。しかし、マルチコアにとってライトスルー型は、すべての書き込みアクセスが共有バス上に出てしまうという点で都合が悪い方法である。実際、ユニプロセッサではライトスルー型の性能はライトバック型に比べてさほど劣るわけではないのだが、マルチコアの場合は、書き込みアクセスが共有バスの混雑を招いて、極端に性能が低下する。このため、どうしてもライトバック型でスヌープキャッシュを構築する必要がある。

まずユニプロセッサと同様、Dirty/Clean を示すビット、Valid/Invalid を示すビットをそれぞれのエントリに持っているとする。ある CPU（P1）がデータを読み出してミスしたブロックは共有メモリ（L2 キャッシュ）からバスを経由して転送され、キャッシュに入る。この時状態は Clean となる。別の CPU（P3）も同一ブロックをキャッシュ上に読み出すことがあるが、両者ともに読むだけなら問題はない。（図 14a）

ここで P3 が書き込みを行うと、通常ライトバック型のキャッシュでは、この書き込みはキャッシュ内だけに留まるのだが、それでは他の CPU は書き込みがあったことを知ることができない。そこで、Clean なブロックに対して書き込みを行った時だけ、書き込んだという情報とそのアドレスだけを共有バスに流してやる。データはキャッシュ上にのみ置けばよいのでこのトランザクションはアドレスのみの簡単なもので良い。これを無効化トランザクションと呼ぶ。他のキャッシュは、この無効化トランザクションをスヌープして、自分のキャッシュに同一アドレスのブロックが存在する場合は、この Valid ビットを 0 にすることによって無効化する（図 14 b）。ブロックのコピーを複数の CPU が持っていたとしても一度に全て無効化されるので、書き込んだ P3 のブロックのみが Dirty な状態でシステム内に存在することになる。したがって、Dirty なブロックに対しての読み書きは外部のバストランザクションを伴わずに行うことができる。このことで、ライトスルーの場合よりも共有バスの混雑は少なくて済む。

さて、ここで P1 が無効化されたブロックに対して読み出しを行ったらどうであろう。無効化されているので、キャッシュはミスをし、共有メモリにブロックを取りに行く。ところが、P3 のキャッシュは Dirty なブロックを保持しており、共有メモリのブロックは最新のものではない。そこで、P3 は、P1 から読み出し要求のあったブロックのアドレスをスヌープして、それが自分のキャッシュ内の Dirty なブロックのアドレスと一致したら、書き戻しを行う必要がある。ここがライトバック型キャッシュの面倒な所であるがやむを得ない。P3 のキャッシュは P1 のキャッシュからの読み出し要求に待ったを掛けて、まず自分の Dirty なキャッシュの内容を共有メモリに書き戻し、次に P1 は書き戻したブロックを読み込む。最終的に両方のキャッシュ共に Clean な状態になってバストランザクションは終了する。

ちなみに効率向上のため、共有メモリと P1 のキャッシュに同時にブロックを転送可能である場合はやってしまってもいい。

では P1 が無効化されたブロックに対して読み出しではなく、書き込みを行ったらどうすれば良いだろう。読み出し同様にミスが起き、ライトバックは普通はライトアロケート型なので、共有メモリに対して書き込みミスの要求を発生する。これを Dirty なブロックを持っている P3 キャッシュがスヌープして、まず書き戻し、次に要求したキャッシュにブロックを転送する。ここまでは全く同じだが、書き戻した後に、このブロックは無効になる。要求を出した方は、ブロックを読み込んだ後にデータの書き込みを行い、ブロックの状態を Dirty にする。この様子を図 14d)に示す。上記の取り決めを守ってブロックの状態とデータ転送を制御すれば、キャッシュ一貫性が保たれる。このような取り決めをキャッシュ一貫性プロトコルと呼ぶ。一貫性プロトコルの動作を表現するのに状態遷移図を用いる場合がある。これは、それぞれプロセッサおよびバスからの要求で、キャッシュブロックの状態がどのように遷移するかを記述する方法である。図 15 に基本プロトコルに対応する状態遷移を示す。

2.3.3 Exclusive 状態の導入

無効化トランザクションは、アドレスと制御信号のみとはいえ、共有バスの混雑を増してしまう。Clean な状態のブロックと同じアドレスのブロックが他に存在しないことが保証されれば、このトランザクションは省略できる。そこで、キャッシュディレクトリに Exclusive(排他的)状態を設ける。L1 キャッシュミスが起きて共有メモリからブロックを持って来る時、それ以外のキャッシュは、自分が同じブロックを保持しているかどうかをスヌープにより確認する。保持することがわかればミスを起こしてブロックを読み込んだキャッシュにその旨通知する。通知が無ければ、このブロックは他にコピーがないので、Exclusive ビットをセットする（図 16a)。このビットが立っているブロックに書き込みが起きても無効化トランザクションを発生する必要はない（図 16 b)。このビットは一度セットされても、他のキャッシュが同一ブロックを読み込んだことを検知すると、リセットされる(図 16c)。Exclusive 状態を導入するには、互いに検知した結果を交換する機構が必要である。

Exclusive ビットがリセットされている状態を Sharable(共有可能性)と呼ぶ。Sharable だからと言ってコピーが必ず存在するとは限らない。コピーはリプレイスされた結果キャッシュ上からなくなってしまうかもしれず、これは他のキャッシュには検知できないためである。

基本プロトコルに Exclusive 状態を付けたプロトコルは、CS(Clean Sharable)、CE(Clean Exclusive)、DE(Dirty Exclusive)、I（Invalidate）の 4 状態で制御を行う。Dirty になる際に無効化が行われるので、Dirty Sharable は存在しない。

2.3.4 Ownership の導入

基本プロトコルは、ライトバックを基にしているので、ミスしたキャッシュは L2 キャッシュにブロックを取りに行き、これを Dirty のブロックを持っているキャッシュが一度ストップを掛けて、書き戻しを行わなければならない。この操作をスムーズに行うには、ミスが起きた場合にブロックを供給する責任（Ownership）を誰かに持たせれば良い。この責任者のことを Owner と呼ぶ。キャッシュのそれぞれのブロックには、Dirty か Clean かの代わりに Owned か Unowned かを示すビットを持たせ、OS(Owned Sharable), US(Unowned Sharable), OE(Owned Exclusive), I (Invalidate)の 4 状態で制御することができる。
この方法では、デフォルトの Owner は共有メモリであり、読み出しミスをしてブロックを読み出すと、それは Unowned Sharable になる。他のキャッシュがミスした場合も、コピーは全て同じ US 状態である（図 17a,b)。ここで、P1 が書き込み要求を出すと、基本プロトコル同様、無効化トランザクションが起きて、他のキャッシュは全て無効になる。書き込んだブロックは OE 状態となり、Owner となる。

ここで、無効化されたブロックを P3 が読み出すと、ミスを生じたキャッシュは共有メモリではなく、Owner にブロックを取りに行く。Owner である P1 のキャッシュと P3 のキャッシュ間で、直接ブロックが転送され、P1 のブロックは OS 状態、P3 のブロックは US 状態になる。この時共有メモリにブロックは書き戻されない。ここで、P3 の US 状態のブロックは Owner、すなわち P1 のブロックと一致しているが、共有メモリの内容とは一致していない点を注目されたい。この方法では、Owner のブロックがキャッシュ上に存在する限り、ブロックはそこから供給され、共有メモリへの書き戻しは起こらない。Owner のブロックがリプレイスされる時のみ、共有メモリへの書き込みが起きる。US 状態のキャッシュがリプレイスされる時は書き戻しの必要はない。

この手法は、制御に無理が少なく、L1 に比べて動作速度が遅い共有メモリに対して極力書き戻しを行わない点も優れている。

2.3.5 MOESI プロトコルクラス

Valid/Invalid, Exclusive/Sharable, Owned/Unowned の 3 ビットでほとんどのキャッシュプロトコルを表現することができる。Invalid なブロックについては他のビットは無意味であるので、有効な状態は 5 状態である。このうち Owned-Exclusive を M　(Modified)、Owned-Sharable を O (Owned)、Unowned-Exclusive を E (Exclusive)、Unowned-Sharable を S (Sharable) のそれぞれ一文字で表し、これに I(Invalid)を加えて、様々なプロトコルを表現する方法を MOESI プロトコルクラスと呼び、現在プロトコルの名称として広く使われている。各状態の包含関係を図 18 に示す。これまでに紹介したプロトコルは以下のようになる。

基本プロトコル（2.3.2）　MSI
Exclusive 状態の導入（2.3.3）　MESI
Ownership の導入（2.3.4）　MOSI

もちろん、MOESI すべての状態を使ってプロトコルを構成することも可能である。