計算機アーキテクチャ 130－11
（1998. 8. 5）

投機的マルチスレッド処理の一手法

大津金光　古川文人
吉永　努　馬場敬信

近年の汎用マイクロプロセッサはクロック速度の向上と命令レベル並列性の活用により性能向上を果たしてきた。クロック速度は物理的限界、命令レベルでの並列度は制御依存による限界が存在するため、プロセッサのさらなる性能向上にはスレッドレベルでの並列性の活用が必須となる。本稿では投機的マルチスレッド実行を複数パスで行なうことによる逐次プログラムの性能向上を目指した実行モデルとそのためのハードウェア機構を提案する。

Speculative Multi-threading with Selective Multi-path Execution

KANEMITSU OOTSU, FUMIHITO FURUKAWA,
TSUTOMU YOSHINAGA and TAKANOBU BABA

Recent microprocessors' performance has been improved by their high-speed clock frequency and by exploiting instruction-level parallelism. Physical limitation of clock speed and semantical limitation of control dependencies impede the improvement of the performance. To overcome this difficulty, it is indispensable to make use of thread-level parallelism.

This paper proposes a speculative thread execution model and its hardware implementation that aims at speed-up of sequentical program execution by selective multi-path thread execution.

1. はじめに

近年の汎用マイクロプロセッサはその微細加工技術によるクロック速度の向上と ILP (Instruction Level Parallelism) の活用を行なうことにより格段の性能向上を果たしてきた。クロック速度の向上には物理的限界が存在するため今後は並列性の活用が必要不可欠となるが、ILP の並列度は投機実行を導入しても高々 5 ～ 10 程度[1] と言われており、さらに上位の並列性を活用しないことにはマイクロプロセッサの性能向上は不可能である。ILP の並列度が小さいのは同時に一つの命令流 (シングルスレッド) のみ追っているためであり、もしも複数命令流 (マルチスレッド) で実行を行なえば並列度を上げることが可能となる[2]。それを背景にして最近は制御フローに沿って投機的にマルチスレッド実行を行ない、並列度を向上させる研究が行なわれている。

制御フローに沿って投機的マルチスレッド実行を行なう場合、分岐によって分かれるパスの片方だけを投機実行の対象として採るのか両方のパスを採るのかで二つの選択肢がある。最も実行の可能性が大きい制御パス上のコードのみをマルチスレッドで実行するモデルを単一パスマルチスレッド実行モデル (以下、単一パスモデル)、それ以外のパスも実行するモデルを複数パスマルチスレッド実行モデル (以下、複数パスモデル) と呼ぶ。Multiscalar[3] や MUSCAT[4],SKY[5] などは単一パスモデルとして分類できる。

単一パスモデルの問題点は、投機実行を深く行なえば行なうほど実際にその実行結果が使われる可能性が小さくなっていくことである。(最先行した投機実行の結果が実際に使われるということがほとんど無いという状況はありえる) さらに単一パスモデルでは分岐予測が失敗した場合のペナルティコストの問題がある。これに対して予測が外れた方向も実行する複数パスモデルは、実行結果が使われる可能性が小さい投機実行を行なうよりはその実行に使用する資源を分岐予測失敗時のペナルティコスト削減のために使う方が良いという点で意味がある。

複数パスモデルの中で EE (Eager Execution) モデルは分岐により生じる双方のパスを全て実行してしまうというモデルである。EE モデルは原理上分岐の予測ミスの際に発生するペナルティコストが 0 であるが、分岐毎に倍の計算資源を必要とし、結果として許容できない規模 (指数的規模) のハードウェア量が必要となる。

DEE (Disjoint Eager Execution)[6] モデルは EE モ

† 宇都宮大学工学部情報工学科
Department of Information Science, Faculty of Engineering, Utsunomiya University

デルでの全てのパスを実行するという点を修正し、全てのパスの中から実行される可能性が高い順に選択して実行するモデルである。DEE モデルを要約すると、制御順序を木構造で表現し、その根から分岐の確率を累積して計算を行ない、実行の確率が大きいところを優先して投機的に実行してしまうということになる。

DEE モデルをそのままハードウェアで実装を行なうのは必要ハードウェア量と動作速度の面から現実的ではない。これは制御フロー木での根からの累積した分岐確率を計算するために各分岐に対応した乗算器が必要となり、さらにその結果を全て集計して分岐確率的に上位のものを選択するというハードウェアが必要になるからである。そこで文献 6) では静的に分岐木を決定してしまう方法を述べている。しかしながら静的に木を決定した場合でも、なお複雑で大量のハードウェアを必要とする。文献 6) では DEE モデルをハードウェア的に実現するアーキテクチャを挙げているが、これは一種のデータフローマシンである。

なお、DEE がその本領を発揮するのはプロセッサ台数がある程度大きくなってからである。一般的に分岐に大きな偏りが存在する場合にプロセッサ台数が小さい場合にはプロセッサで実行されるスレッドは全て分岐確率が大きい方向を採るパス上のものだけとなる。(文献 6) の例では 24 台以上のプロセッサが無い場合は 1 つのパスに沿って実行をするのと変わりない) そういう意味では DEE はプロセッサ台数が数十以上存在する場合に有効なモデルであると言える。そのためスレッドのプロセッサ資源への割り付け等の管理情報を集中管理するのはスケーラビリティの面から不利である。

そこで我々は DEE モデルの近似的な実現として DEE モデルに制限を加えたモデルを提案し、それを実現するハードウェアを考案した。

2. SMT 実行モデル

本稿で提案する SMT(Selective Multi-path Thread execution) 実行モデルを説明する。図 1 の制御フローにおいて各ノードはスレッドコードの境界 (分岐命令が存在する) である。各ノードから出ている辺はそれぞれのスレッドコードを示し、左辺のスレッドコードが右辺のスレッドコードより実行される可能性が大きいものとする。(つまり全ての左側の辺を通るパスが最も実行される可能性が大きく、逆に全ての右側の辺を通るパスが最も実行される可能性が小さい) 図において実線の辺を投機実行の対象とする。これらのパスは分岐において全て実行確率が高い方向を採る (0 次パス、あるいはメインパス) か、もしくは一度のみ実行確率の低い方向を採る (1 次パス) ものである。(以下 n 回のみ実行確率の低い方向を採るものを n 次パスと呼ぶ) 二度以上実行確率の低い方向を選択するパスを投機実行の対象から外すのは、それが実行される可能性が 1 次パスよりも小さいためである。

メインパス上のスレッドコードはその全てをマルチスレッドで投機実行を行ない、1 次パス上のスレッドコードに関してはその先頭の一部のみをマルチスレッドでの投機実行を行なう。(1 次パス上の先頭の数個を除く残りのスレッドコードは一時的にマルチスレッドでの実行はされない。これは 1 次パスがメインパスよりも実行される可能性が低いことを考慮すると妥当な選択だと言える。ただし、1 次パス側が真のパスになった場合にはマルチスレッドによる投機実行になる)

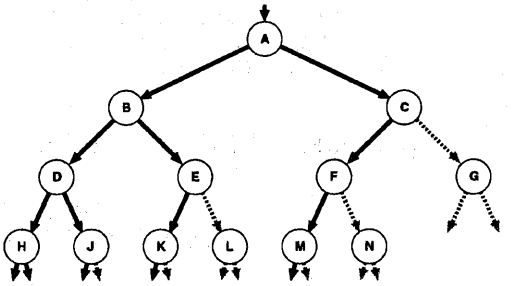

図 1　制御フローの例

図の場合で、分岐 A,B,D,H のメインパス上のスレッドコードは全てマルチスレッドでの投機実行を行なう。分岐 A,C,F,M のパス、A,B,E,K のパス、A,B,D,J の 1 次パス上のスレッドは先頭の k 個 (k はメインパスのスレッド数より小、パスによって可変) のスレッドのみをマルチスレッドでの投機実行を行なう。

ここで 1 次パスでのマルチスレッド実行を行なう先頭の数 k は場合によっては 0 であることがありえる。この場合、その 1 次パスに関してはスレッドの投機実行しないことを意味する。(分岐確率やプロセッサ台数にもよるが、ほとんどの分岐に関してその分岐確率は大きく偏りがあることを考慮に入れるとプロセッサ台数が数十台までは k の値は 1 もしくは 2 程度になると考えられる)

メインパス上のスレッド実行が間違っていた (実行されるべきでないと判断された) 場合は間違っていたスレッド以降の全てのスレッド実行を無効化し、直前の分岐からもう一方へのパス (1 次パス) を新たなるメインパスとして枝を延ばす。パスの変更にはスレッドの消滅生成等のペナルティコストがかかるが、1 次パス上の先頭のいくつかは既に投機的にマルチスレッド実行を開始しているため、その分で補償される。

図において、例えばメインパス上の分岐 B において実際の実行パスが反対のパスであった場合はメインパス上の分岐 B 以降のスレッドを全て無効化し、分岐 B から反対に延びる 1 次パスを新たなるパスとする。この際、既にその 1 次パス上の先頭側のいくつか、例えば分岐 B,E 間のスレッドコードのみ (前述の k=1 の場合) が既に投機的に実行を開始しているとすれば、分岐 E 以降の

1次パス上の枝を延ばしていく(子スレッドを生成していく)ことになる。この場合は分岐B,E間のスレッドがE,K間のスレッドを生成し、E,K間のスレッドはさらに自分の子スレッドを生成することになる。

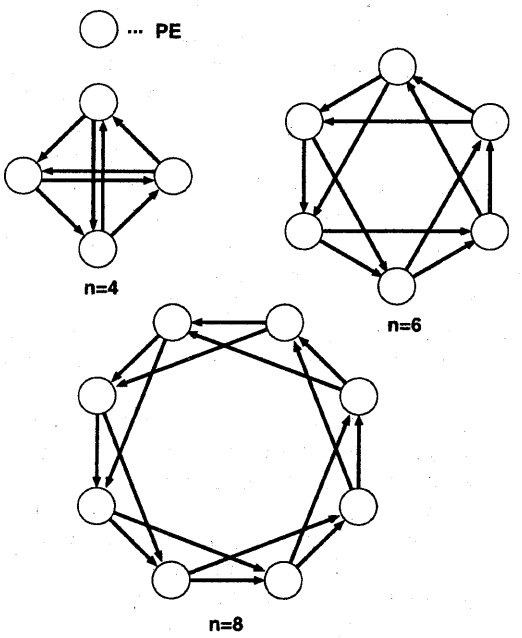

図2 プロセッサトポロジー

この実行モデルを実現するためのプロセッサ間の結合形態を図2に示す。図に示す通り各プロセッサがリング状に結合されている。各プロセッサからは単方向リング上での隣とその隣への2本のパスが出ている。(これは論理的な結合を示すものであって物理的に2本なければならないという意味ではない。実際に2本出すかどうかはハードウェア量と動作速度のトレードオフから判断しなければならない)各プロセッサから出る2本のパスが分岐の2方向に対応する。隣への1本のパスだけではプロセッサの飛び越しが不可能であるため、さらにその隣への2本目のパスを出すというのが本質的である。プロセッサ台数が4の場合は完全結合に近いが、6台、8台と台数が増えるに従って完全結合との差が明確になっていく。

図3にこのプロセッサ結合形態でのプロセッサへのスレッドの割り付けの例を示す。図(a)は制御フローグラフの一例である。この図において各ノードはスレッドコード、辺は分岐方向を示す。このグラフにおいてメインパスがA,B,D,Eであるとする。ここで分岐Aにおいて C側へのパスが別スレッドとして実行した方が性能が上がると判断された場合は図(b),(c)のようなプロセッサ割り付けが可能である。(別スレッドで実行しても性能が上がらないと判断した場合はA,B,D,Eの1つのパスをマルチスレッドで実行する)図(b)は1次パスである C,D,E側の先頭のCのみをスレッド実行することに対応し、図(c)はC,Dの2つをマルチスレッド実行することに対応する。(同様にマルチスレッドで実行する先頭の数を3,4,.. と増やしていく例が考えられる)これらの割り付け方はコンパイル時に静的に決定されているものであり、動的に切り替わるわけではない。

図(b)の場合、Aからの分岐方向が確定し、B側のパスが正しかった場合はCを実行しているプロセッサの実行が取り消される。逆にC側のパスが正しかった場合はB以降のB,D,Eを実行しているプロセッサの実行が取り消される。この際、開放されたプロセッサを使って C以降の D,Eのスレッド実行を開始する。(これが真のパスをメインパスから1次パスへの変更することに対応する)

図(c)の場合も図(b)の場合と同様である。この場合、B側を通って実行される Dと C側を通って実行される Dはコードとしては同じでも意味的には異なる(レジスタまたはメモリの Bで更新された内容と Cで更新された内容は別ものである)ものであるため、それぞれD1,D2の別のインスタンスとして実行を行なわなければならない。

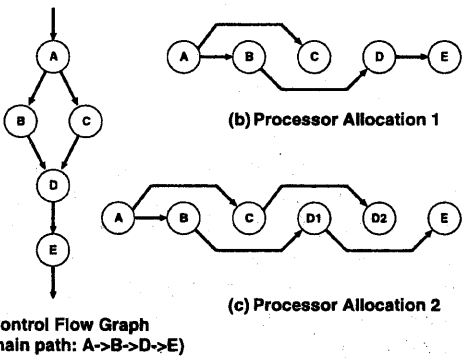

図3 プロセッサ割り付け例

3. 全体構成

前節で説明を行なったモデルを実行するためのアーキテクチャについて説明を行なう。図4に全体の構成を示す。各プロセッサコアは各々プライベートな1次キャッシュ(**PCache**)を持ち、投機的なメモリアクセスを把握するための情報を保持する **MAB**(Memory Access Buffer)を介して接続されており、その先にネットワーク(バスもしくはクロスバーネットワーク)を介して大容量の2次キャッシュ(**SCache**)に接続されている。プロセッサコアは ILP活用のためスーパースカラープロセッサをベースとしてレジスタファイル周辺、ロード

ストアバッファ周辺、およびスレッド制御のための拡張を施したものを想定する。プロセッサ間のデータ通信はレジスタおよびメモリを介して行なう。そのために各プロセッサのレジスタファイルと **MAB** 間にデータ転送のためのパスを用意する。各レジスタファイルおよび **MAB** からは先に述べたプロセッサトポロジーに応じて隣とさらにその隣へのデータパスが存在する。(図には無いが、スレッド制御用の信号線が同様のトポロジーで接続されている)

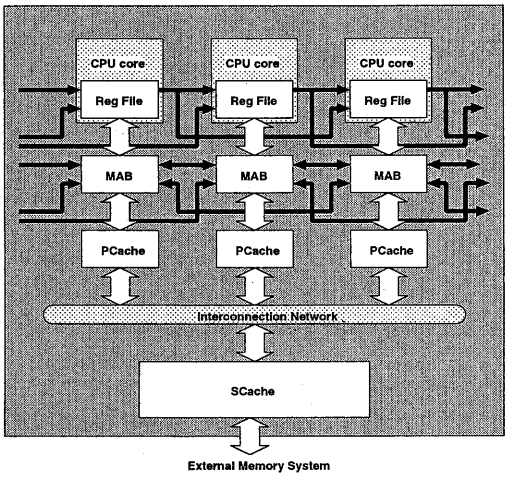

図4 全体構成

3.1 スレッド制御

各プロセッサは2つのスレッド生成先それぞれに対して1つずつ **fork buffer** を持つ。**fork buffer** はスレッド実行開始点、実行状態 (投機実行状態であるか確定実行状態であるか)、パスの **priority**(メインパス上のコードか、1次パス上のコードであるか)、**valid-bit**(スレッド生成要求が存在することを示す)、の情報を保持する。

これはスレッドの状態管理の役目を果すと同時に、スレッド生成要求を後続のプロセッサに対して発行した際に要求先のプロセッサがアイドル状態でなく、スレッドの生成が行なえない場合に一時的にそのスレッド生成要求を格納しておくためのバッファとして機能する。

valid-bit はスレッド生成要求の発行によりセットされ、後続のプロセッサによる当該スレッドの実行が終了する際にリセットされる。

各プロセッサはアイドル状態になった際に、先行するプロセッサの **fork buffer**(自プロセッサへのスレッド生成要求が格納される) を調べ、要求があれば (**valid-bit** がセットされている) そのスレッド実行を開始する。先行するプロセッサからの要求が無い場合は要求が来るまで待機する。

スレッド制御のために以下の命令拡張を行なう。

- **fork(.sp)** *id,pri,target_addr*
- **cfork(.sp)** *id,pri,target_addr*
- **kill** *id*
- **term**
- **commit** *id*

fork(.sp) 命令は子スレッドの生成を行なう。(**fork** 命令は確定状態でスレッド実行を開始し、**fork.sp** は投機状態で開始を行なう) *id* は0または1の値を取り、それぞれ隣のプロセッサへのスレッド生成と二つ隣のプロセッサへのスレッド生成に対応する。各 *id* に対応するプロセッサに対して1度のみスレッド生成要求を許す。*pri* は0または1の値を取り、生成するスレッドがメインパスに属するものであるか、1次パスに属するものであるのかを指定する。*target_addr* はスレッド実行の開始点を示す。**fork(.sp)** 命令は分岐・ジャンプ命令の拡張であり、*target_addr* はアドレスモードとしてそれらに準じるものを採る。スレッド生成先のプロセッサがアイドル状態でない場合は **fork buffer** に一時的に格納する。

cfork(.sp) 命令は子スレッドの生成を行なう。**fork(.sp)** 命令との違いは、条件付きでスレッド生成を行なう点である。**cfork(.sp)** 命令は自プロセッサがメインパスに属さないスレッドの実行を行なっている場合にのみスレッド生成要求を発行する。これは1次パス上のスレッドコードにおいて、真に実行すべきパスになった際にマルチスレッド実行を開始する潜在的なスレッド生成命令として使用する。

kill 命令は子スレッドの強制終了を行なう。これは投機実行を行なっていた子スレッドが正しくないコードを実行していた場合等に使用する。*id* は **fork(.sp)** 命令と同様に0,1を取る。もしも後続プロセッサがまだスレッド実行を開始していなかった場合 (**fork buffer** 内に要求が格納されたままの状態の場合) は **fork buffer** 内の要求を無効化する。**kill** 操作要求を送られてきた後続のプロセッサは実行中のスレッドを破棄する。同時に自プロセッサが生成したスレッドに対して **kill** 操作をフォワードする。これは自分がスレッド生成要求を発行したプロセッサ (**fork buffer** を調べれることで判別) に対しての **kill** 操作を行なうことで実現する。

term 命令は自スレッドの終了を行なう。もしも自スレッドが投機状態 (先行スレッドからの確定状態への状態変更指示がされていない) ならば確定状態になるまで停止する。**term** 命令実行後は現プロセッサはアイドル状態になり、スレッド生成要求があるまで待機する。

commit 命令は投機状態にある子スレッドを確定状態に移行させる。これは自スレッドの実行後、子スレッドが確実に実行されることが決まった際に子スレッドに対して使用する。*id* は **fork(.sp)**,**kill** 命令と同様0,1を取る。もしも後続プロセッサがまだスレッド実行を開始していなかった場合は **fork buffer** 内の要求を確定状態でのスレッド生成要求に変更するだけでよい。ここ

で **commit** によって投機状態から確定状態に遷移したスレッドが1次パス上のスレッドであった場合 (つまりメインパスが真のパスでなく、1次パス側が真のパスであったことを意味する) は子スレッドに対してメインパスになった旨を通知する。これにより1次パス上のスレッドで潜在的にスレッド実行を開始すべきだったもの (**cfork** 命令によるスレッド生成) が真にその実行を開始する。

3.2 レジスタデータフロー

レジスタファイル周辺を図5に示す。各プロセッサは通常レジスタファイルの他にもう1セットレジスタファイル (バックアップレジスタファイル) を持ち、実行開始時 (スレッド生成時) に直前のスレッドのレジスタの全イメージが両者に転送される。さらに実行開始後は後述のスレッド間レジスタ転送命令により先行スレッドから転送されてくるデータを両者に格納する。

通常レジスタファイルとバックアップレジスタファイルの違いは、前者が先行スレッドからの変更に加えてローカルに実行した結果によっての変更も受ける点である。バックアップレジスタファイルは先行スレッドのその時点での最終的なレジスタイメージを保持することにより、投機実行の rollback が発生した際 (先行するスレッドからのデータが到着していない場合に間違った値を使用して実行した場合に発生する) に使用される。

図において **port_in0** は先行する側での隣のプロセッサからのレジスタデータ、**port_in1** は二つ隣のプロセッサからのレジスタデータを受け取るためのポートである。どちらのポートからのレジスタデータを受け取るかは自分の親スレッドからのものであるかどうかで判定する。

各レジスタには同期ビット (full/empty bit) が付随しており、先行するスレッドからのレジスタデータが到着したかどうかの判定に使用する。スレッド実行開始時に全てのエントリが empty 状態に設定され、先行スレッドからのデータが到着する毎に、あるいはローカルな実行の結果を書き込む際に対応するエントリは full 状態になる。full 状態のレジスタ (既にローカルな実行によって変更を受けている状態である) への先行スレッドからのデータ転送は、バックアップ側にのみ書き込みを行なうことになる。(rollback が発生した際には先行するスレッドから送られてきたレジスタデータが必要)

スレッド間のレジスタ通信制御に以下の命令拡張を行なう。

- **send** *id,reg*
- **recv** *reg*

send 命令はレジスタ *reg* の値を *id*(0 または 1) で示されるスレッドの同名のレジスタに対してレジスタデータを転送する。**recv** 命令はレジスタ *reg* の同期ビットのチェックを行ない、先行スレッドからのデータが到着していればそのまま後続の命令実行を続け、到着していなければ到着するまで待機する。

send,**recv** 命令は対になって使用される。対応する **send** 命令の無い **recv** 命令が存在しないように注意してスレッドコード生成する必要がある。(逆に、対応する **recv** の無い **send** は問題ない)

先行スレッドが条件によってはレジスタデータを **send** するかどうかが不明な場合があるが、その場合は先行スレッド内でどのパスを通っても **send** が発行されるようにダミーの **send** を入れる必要がある。

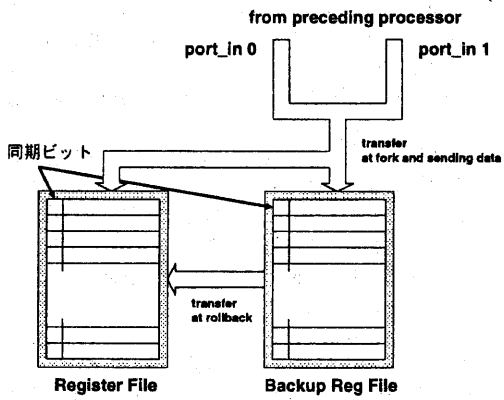

図5 レジスタファイル構成

3.3 メモリデータフロー

投機的なメモリアクセスはその結果が確定状態になるまで外部に反映してはならない。レジスタアクセスとは異なり、メモリアクセスはポインタ参照が存在するため、静的に全ての依存関係を追跡することが不可能である。そこでレジスタアクセスの場合とは異なった手法でスレッド間のデータ依存関係を動的に把握する必要がある。そのためこのアーキテクチャでは **MAB**(Memory Access Buffer)(図6) により投機的なメモリアクセスを追跡するメカニズムを持つ。**MAB** はメモリアクセスの対象アドレスをキーとする連想メモリであり、プロセッサから発行された投機的なメモリアクセスがスレッド間の依存を保っているかどうかの判定に使用する。

図6において **Potential Violation-bit**(以下 **PV-bit**) は投機アクセスが依存関係を保っていない可能性があることを示し、**Local Store-bit**(以下 **LS-bit**) はスレッドローカルに書き込みが発生し、先行スレッドとの依存が断ち切られていることを示す。

投機状態のプロセッサからのストアアクセスは **MAB** 内に保持され、確定状態に遷移するまでは1次キャッシュより下位のメモリ階層にはライトバックされない。(投機状態において、キャッシュコントローラは **MAB** の情報を参照する必要がある) 投機状態から確定状態に遷移した場合には **MAB** 内の格納されているストア処理に対応したキャッシュ上のデータのライトバックが可能となる。

各プロセッサから発行されるメモリアクセスリクエストにより **MAB** は次の動作を行なう。

確定実行状態のプロセッサからのリクエストに関して **MAB** は何もしないが、ストアアクセスの場合だけは後続スレッドを実行しているプロセッサの **MAB** に対してストアアクセスのフォワードを行なう。

投機実行状態のプロセッサからのリクエストの場合は以下の動作をする。

- ロードリクエスト
  MAB 内に一致するアドレスを持つエントリが存在する場合は1次キャッシュからデータを読み出しプロセッサに供給する。一致するエントリがない場合は先行スレッドのプロセッサの **MAB** に問い合わせて返送されてきたデータを自プロセッサに供給する。この際、**MAB** の1エントリを割り当て対象アドレスを格納する。(空きエントリが無い場合は空きが出来るまでロード処理は保留) プロセッサにデータを供給する際に **LS**-bit がセットされていなければ、**PV**-bit をセットし、将来発生し得る実行のやり直しに備える。
- ストアリクエスト
  対象アドレスを **MAB** に格納 (もしも一致するエントリがない場合は1エントリを割り当てる。空きエントリが無い場合は空きが出来るまでストア処理は保留) し、データを1次キャッシュに書き込む。この際、同時に **LS**-bit をセットする。同時に後続スレッドを実行しているプロセッサの **MAB** に対して現ストアアクセスをフォワードする。

各 **MAB** 間では以下の動作を行なう。

- 後続スレッドからのロードデータ要求時
  自分が確定状態であるならば、1次キャッシュからデータを読み出し要求元に返送する。自分が投機状態である場合に **MAB** 内でアドレスが一致するエントリがある場合はそのデータ1次キャッシュから読み出し要求元に転送する。一致するエントリが存在しない場合は先行スレッドを実行しているプロセッサの **MAB** に問い合わせを行ない、返ってきたデータを要求元に転送する。
- 先行スレッドからのストアデータ書き込み要求時
  MAB 内でアドレスが一致するエントリがある場合はその **PV**-bit を調べ、もしもセットされていれば自プロセッサに対して実行のやり直しを通知する。一致するエントリの **LS**-bit がセットされていた場合は既にそこで依存関係が断ち切られているため何もしない。一致するエントリが存在しない場合は後続スレッドへ現ストアデータ書き込み要求をフォワードする。

4. おわりに

投機的にマルチスレッド実行を行なう際に実行頻度が大きいパス上だけでなく、実行頻度の小さい方向のパス上も選択的に投機実行してやることで全体の実行速度を向上させる手法について提案を行なった。さらにこの手法を実現するためのアーキテクチャの提案を行なった。

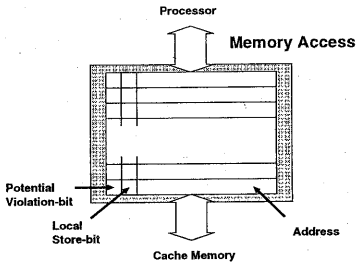

図6 メモリアクセスバッファ (MAB)

謝辞 本研究において有効かつ適切なアドバイスを頂きました (株) 東芝研究開発センター情報通信システム研究所 小柳滋氏に感謝致します。本研究は一部文部省科学研究費 基盤研究 (C) 課題番号 09680324, 基盤研究 (B) 課題番号 10558039, 奨励研究 (A) 課題番号 09780237, 並列・分散処理研究推進機構の援助による。

参考文献

1) Wall, D.W.: Limits of Instruction-Level Parallelism, Technical Report DEC-WRL-93-6, Digital Equipment Corporation, Western Research Lab (93).
2) Lam, M.S. and Wilson, R.P.: Limits of Control Flow on Parallelism, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 46–57 (1992).
3) Sohi, G. S., Breach, S. E. and Vijaykumar, T. N.: Multiscalar Processors, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414–425 (1995).
4) 鳥居淳, 近藤真己, 本村真人, 西直樹, 小長谷明彦: On Chip Multiprocessor 指向制御並列アーキテクチャ MUSCAT の提案, *Joint Symposium on Parallel Processing 97*, pp. 229–236 (1997).
5) 小林良太郎, 岩田充晃, 安藤秀樹, 島田俊夫: 非数値計算プログラムのスレッド間命令レベル並列を利用するプロセッサ・アーキテクチャ SKY, *Joint Symposium on Parallel Processing 98*, pp. 87–94 (1998).
6) Uht, A.K. and Sindagi, V.: Disjoint Eager Execution: An Optimal Form of Speculative Execution, *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 313–325 (1995).