Vol. 41 No. SIG 5(HPS 1) 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 Aug. 2000

specMEM：同期操作に対するメモリ・アクセスの投機的実行機構

佐藤 貴之[†,☆] 松尾 治幸[†]
大野 和彦[†] 中島 浩[†]

本論文では，共有メモリ型並列マシンにおける同期処理オーバヘッドを削減する手法として，同期操作に後続するメモリアクセスを同期成立確認以前に投機的に実行する機構 **specMEM** を提案する．この機構の特徴は，投機失敗の検出やそれにともなう計算状態の復元を，コヒーレント・キャッシュに簡単な拡張を施すことにより実現することにある．またキャッシュラインの状態を記憶するタグを簡単な機能メモリを用いて実装することにより，投機の開始，成功，失敗にともなう操作を，投機的に行ったアクセスの数によらず定数時間で実行できることも重要な特徴である．また本論文では，バリア同期を対象とした投機的アクセスの実装モデルを，SPLASH-2 ベンチマークを用いてシミュレーションにより評価した結果についても述べる．評価の結果，LU 分解では 13%の性能向上が得られ，負荷の変動によって同期区間が伸縮するようなプログラムについて specMEM が有効であることが明らかになった．

specMEM: A Mechanism for Speculative Memory Accesses Following Synchronizing Operations

TAKAYUKI SATO,[†,☆] HARUYUKI MATSUO,[†] KAZUHIKO OHNO[†]
and HIROSHI NAKASHIMA[†]

In order to reduce the overhead of synchronizing operations of shared memory multiprocessors, this paper proposes a mechanism, named **specMEM**, to execute memory accesses following a synchronizing operation speculatively before the completion of the synchronization is confirmed. A unique feature of our mechanism is that the detection of speculation failure and the restoration of computational state on the failure are implemented by a small extension of coherent cache. It is also remarkable that operations for speculation on its success and failure are performed in a constant time for each independent of the number of speculative accesses. This is realized by implementing a part of cache tag for cache line state with a simple functional memory. This paper also describes an evaluation result of specMEM applied to barrier synchronization. Performance data was obtained by simulation running benchmark programs in SPLASH-2. We found that the execution time of LU decomposition, in which the length of period between a pair of barriers significantly varies because of the fluctuation of computational load, is improved by 13%.

1. はじめに

共有メモリ型並列マシンは，ごく普通のロード/ストアによってプロセッサ間通信が実現できるという，プログラマにとってきわめて便利な特質を持っている．また，この細粒度かつ高速な通信は，コヒーレント・キャッシュをはじめとする様々なハードウェア機構により実現されているため，ソフトウェアのオーバヘッドは皆無に近く，通信の遅延も隠蔽あるいは削減される．

しかし，プロセッサ間通信には，ロード/ストアによる共有メモリのアクセスだけではなく，同期操作を必要とする．すなわち異なるプロセッサによる共有変数のアクセスと，プログラムが要求する依存関係を充足するようにアクセスを順序付ける同期操作によって，通信が実現される．したがって，ある同期操作を開始すると，その同期が成立したことが確認されるまで，同期に関連する共有変数のアクセスを行うことはできない．また実装の簡便さを保つために，多くの場合はより広い範囲の操作，たとえばあらゆるメモリ・アクセスが，同期成立確認まで禁止される．

この同期操作は一般に通常のメモリ・アクセスより

† 豊橋技術科学大学情報工学系
Department of Computer and Information Sciences, Toyohashi University of Technology
☆ 現在，ソニー株式会社
Presently with Sony Corporation

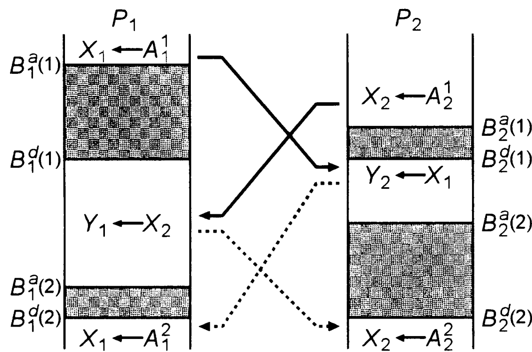

図 1 バリア同期によるデータ依存制約の充足
Fig. 1 Satisfying data dependency constraint by barrier synchronization.

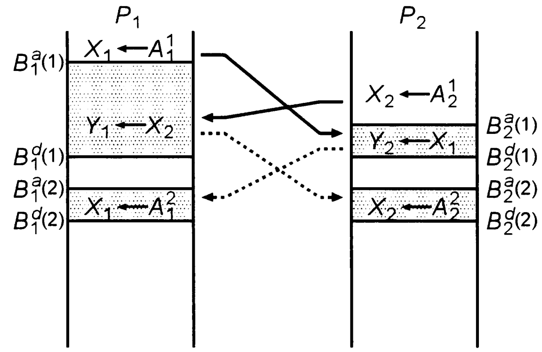

図 2 投機的メモリ・アクセスの成功
Fig. 2 Successful speculation of shared memory accesses.

も時間を要するため，同期操作の遅延隠蔽や削減は共有メモリのアーキテクチャとプログラミングの双方にとって重要な技術的課題である．このための自然なアプローチとして考えられるのは，同期操作の実行頻度を削減するために通信の粒度を大きくすることである．この方法は，分散メモリマシンにおけるメッセージの集合化などの技術と共通する発想に基づいており，同様の効果をもたらすように見える．

しかし通信の粗粒度化は，細粒度通信という共有メモリの特質とは整合しないため，有効な方法であるとは限らない．すなわち同期成立確認までの操作禁止によって，不必要なオーバヘッドが生じることがしばしばある．たとえば**図 1** はバリア同期を用いたプロセッサ P_1 と P_2 の間の通信を示したものであるが，X_1 のフロー依存制約（図中の実線矢印）は P_2 よりも P_1 が最初のバリアに先に到達しているので（$B_1^a(1) < B_2^a(1)$），明らかに充足されている．したがって，P_2 がバリア同期成立を確認するために要する時間 $B_2^d(1) - B_1^a(1)$ は，無駄なアイドル時間であるといえる（図中の暗い影付きの部分）．また X_2 のフロー依存制約は，P_1 がバリアへ到達した時点 $B_1^a(1)$ では満たされていないが，同期成立確認時点 $B_1^d(1)$ よりも以前に満たされているので，やはり無駄なアイドル時間が生じている．さらに同様の現象は，X_1 と X_2 の逆依存制約を満たすための 2 番目のバリア同期についても見ることができる．

このような無駄なアイドル時間は，フロー依存や逆依存などのデータ依存制約を，同期という一種の制御依存制約に置き換えて充足しようとするために生じたものであると考えることができる．そこで我々は，制御依存による遅延を除去する方法として一般的に用いられている投機的実行[1),2)]を応用することにより，無駄なアイドル時間を除去あるいは削減する機構 **specMEM** を考案した[3)~5)]．

以下本論文では，specMEM における投機的アクセスの方式，すなわち同期の成立を確認する以前に，データ依存制約が満たされていることを仮定してメモリ・アクセスを行ってオーバヘッドを除去する方式について述べる．まず 2 章では，投機的アクセスの概要を成功例と失敗例のそれぞれについて述べる．続いて 3 章では，コヒーレント・キャッシュに簡単な拡張を施すことによる現実的かつ効率的な実装モデルを示す．4 章では，SPLASH-2 に含まれるいくつかのベンチマークによる評価結果とそれに対する考察を述べ，5 章では関連研究についての議論を行う．

2. 投機的メモリ・アクセス

2.1 投機の成功と高速化

我々が提案する specMEM では，同期操作を実行してもプロセッサは停止せず，その完了が確認されるまでの間は**投機モード**に移行して処理を続行する．したがって，共有変数へのアクセスを含むすべてのメモリ・アクセスは，同期操作により充足されるべきデータ依存制約がすでに満たされているものと仮定して，投機的に実行される．

たとえば，図 1 に示したバリア同期による通信は，投機的アクセスによって**図 2** に示すように実行される．この例では，X_2 のフロー依存制約を充足するためのバリア同期に P_1 が到達すると（$B_1^a(1)$），P_1 は投機モードに移行して処理を続行する（図中の明るい影付き部分）．その結果，同期成立確認が $B_1^d(1)$ でなされる以前に P_1 は X_2 を読み出す．しかし X_2 の書き込み/読み出しのタイミングが図に示すようにフロー依存制約を満たす場合，アクセスによって得られる値は正しく，投機は成功する．また $B_2^a(1)$ と $B_2^d(1)$ の間

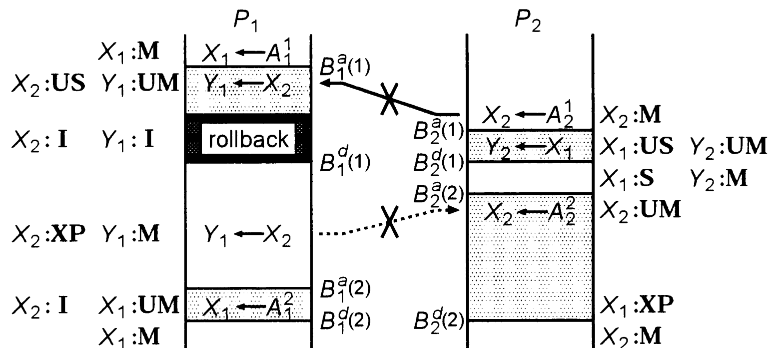

図3 投機の失敗と計算状態の復元
Fig. 3 Speculation failure and rollback.

に行われる P_2 による X_1 のアクセスについても同じように投機は成功し，この結果どちらのプロセッサについてもアイドル時間が完全に除去される．

また同様に，X_1 と X_2 の逆依存制約を充足するためのバリア同期に P_1 と P_2 が $B_1^a(2)$ と $B_1^a(2)$ で到達後，これらへの書き込みが $B_1^d(2)$ と $B_2^d(2)$ での完了確認以前に投機的に行われる．この場合も読み出し/書き込みの順序が逆依存制約を満たしているので，やはり投機は成功する．

この例に示すように，投機的アクセスは同期成立確認に要する時間を削減するものであるので，バリア到達時刻にずれが生じる場合，特に負荷変動などによって到達順序が変化する場合に有効である．また負荷が均衡している場合も，多数のプロセッサがバリアに参加することにより確認のための遅延が大きくなれば，遅延隠蔽の効果が顕著に現れる．

2.2 投機の失敗

前節の例では充足すべきデータ依存制約が，投機的に行われたすべてのアクセスについて満たされたため，プログラムの意味を保存しつつオーバヘッドを除去することができた．しかし投機である以上，依存制約を満たさないアクセスが行われる可能性はつねにあり，その場合にもプログラムの意味が保存されるような措置が必要である．

たとえば**図3**に示す例では，P_1 による X_2 の投機的読み出しが P_2 による X_2 への書き込みに先行し，その結果不正な値を読み出してしまっている．またこの不正な値は Y_1 に格納されるので，Y_1 を参照する操作があれば不正値が次々に伝搬する．このような場合，まず不正な投機的読み出しを行ってしまったことを検知し，続いて不正値によって生じたあらゆる計算状態変化を無効化し，正しい値による計算を再度行う必要がある．

この計算の無効化と再実行を，動的命令スケジューリングを行うプロセッサに用いられるロード/ストア・バッファに類似した機構を用いて実現することは可能である[6)~9)]．すなわち，潜在的に不正なロード命令のアドレスを連想バッファに記憶して他のプロセッサからの更新通知と比較し，かつ計算状態を保存するためにストア命令のアドレスとデータ（あるいはそのアドレスの旧値）を別の連想バッファに記憶すればよい．しかしこの方法は連想バッファに要するハードウェア・コストの制約から，バッファ容量が必然的に小さい値となり，同期成立確認までの大きな遅延を隠蔽するには不十分であることが予想される．また投機が成功したことが確認されたとき（あるいは失敗が検出されたとき），ストア・バッファに記憶しておいたすべての書き込み操作をバースト的に実行する必要があり，投機成功に関する手間が投機的書き込みの数に比例することとなる．

そこで3章で詳しく述べるように，この投機失敗の検知と無効化をライトバック型のコヒーレント・キャッシュを拡張して実現することとした．まず，投機モードでアクセスされたキャッシュ・ラインすべてに，潜在的に危険であることを示すマークが付けられる．上記の例では，X_2 の読み出し前の状態が **S**（Shared）であるとすると，投機的読み出しの結果 **US**（Unsafe Shared）という特別な状態に遷移する．また Y_1 の状態も，**M**（Modified）に対応する状態 **UM** となる．この **U** が付された状態にあるラインに対する他のプロセッサからの書き込み通知を受けとると，不正値を読み出していた可能性があることが判明する．すなわち上記の例では，P_2 による X_2 への書き込み通知によって，P_1 のキャッシュが不正読み出しを検知する．

次に行う計算状態変化の無効化は，キャッシュのライトバック機構を利用して行う．すなわち，ラインの

状態が **M** から **UM** または **US** に変化する際に，ラインの値をメモリに書き戻すことにより，投機開始時点の値がメモリに保存されるようにする．投機失敗が検知されると **UM** のラインはすべて無効化され，以後の操作では保存された真値が参照されるようにする☆．この結果，メモリに関する計算状態変化はすべて無効化されるので，他の計算状態をシャドーレジスタ[10),11)]などの機構を用いて投機開始時点で保存しておけば，その復元によりロールバックを行うことができる．ロールバック後の実行再開は，複数のプロセッサが互いを投機的にロールバックさせてデッドロックに陥ることを防止するために，図3に示すように同期成立が確認されるまで抑止される．

この方法は前述のロード/ストア・バッファを用いるものと比べ，十分に多くの投機的アクセスを許容できるという利点がある．また後述のようにマスク付きの一括リセットができるような簡単な機能メモリを用いることによって，投機失敗時の一括無効化を定数時間で実現できることもメリットである．この機能は投機の成功時，すなわち同期成立が確認された時点で，**U** が付されたラインをすべて普通の状態に戻す操作も定数時間で実現する．

2.3 共有変数の投機的書き込み

図3の例では，P_1 でのロールバックの後，X_2 の値が再び読み出される．一方 P_2 では最初のバリアに関する投機は成功し，続いて2番目のバリアに関する投機的アクセス，すなわち X_2 への書き込みを行う．この書き込みは図に示すように P_1 による X_1 の読み出しに先行してしまっているので，逆依存制約を満たしていない．

この不正な書き込みを行ってしまったことは，無効化型のプロトコルを用いれば，前節と同様に **UM** 状態のラインへの他プロセッサからの読み出し要求によって検知できる．したがって，前節と同様にロールバックすることもできるが，**UM** 状態のラインについては書き込み前の値がメモリに保存されていることを利用して，正しい値を参照元プロセッサに返すことができる．すなわち図の例では，P_2 のキャッシュにある書き込み後の値 A_2^2 ではなく，$B_2^a(2)$ の時点での値である A_2^1 をメモリから P_1 へ返すことが可能である．

しかしこの値は，2番目のバリア同期成立確認までに読み出される限りは正しいが，それ以降は逆に不正に古い値となってしまう．すなわち（図には示されていない）3番目のバリア以降では，X_2 の値は A_2^2 でなければならない．しかし P_2 は X_2 への書き込みの時点で書き込み通知を送ってしまっているので，P_1 はキャッシュした値 A_2^1 を無効化（あるいは更新）する機会を失っている．

そこで，他のプロセッサのキャッシュに **UM** 状態のラインが存在するためにメモリから値を得たラインについては，**XP**（eXPiring）という特別な状態を割り当て，次のバリア同期に到達した際に（図では $B_1^a(2)$）一括して無効化する．この結果，P_1 が X_2 を改めて読み出す際にはキャッシュミスとなり，P_2 のキャッシュから正しい値である A_2^2 を得ることができる．なおこの一括無効化は，前述の投機成功/失敗時の一括状態変更と同様に，簡単な機能メモリを用いて定数時間で行うことができる．

また図の例では，P_2 が X_2 に書き込む時点で P_1 は X_2 をキャッシュしていないが☆☆，投機的書き込みの対象が他のキャッシュに存在することもある．その場合には，書き込み通知に投機的であることを示す情報を付加し，他のキャッシュに存在するコピーの状態を直接 **XP** に遷移させる．これにより，投機的書き込みによる無駄な無効化にともなうキャッシュ・ミスを防止できる☆☆☆．

3. 実装モデル

3.1 概　　要

前章で述べたように，specMEM はライトバック型のコヒーレント・キャッシュを拡張する形で実装される．本節ではこの実装モデルを，コヒーレンス・プロトコルが write-invalidate であり，キャッシュのベースとなる状態が **M**，**S**，および **I**（Invalid）であるとして述べる．なおこの前提は議論を簡潔にするためのものであり，ベース状態の拡張（3.4節参照）や write-update プロトコルの採用を妨げるものではない．

このベースとなるキャッシュ状態 **M**，**S**，**I** のそれぞれに対応する形で，投機的アクセスに関する特別な状態 **UM**，**US**，**XP** が加えられる．これらの状態間の遷移の概要は，以下のとおりである（図4）．

(1) 通常の状態 {**M**,**S**,**I**} にあるラインに対して投機モードでの読み出し（$r(s)$）を行うと，投機的読み出しを記憶するためにラインは **US** へ遷移する．また元の状態が **M** であれば，ライトバックにより投機失敗に備えてラインの状態を

☆ 後述のように現在の実装モデルでは U が付されたラインはすべて無効化する．

☆☆ ロールバックの際に無効化されている．

☆☆☆ write-invalidate の場合．write-update であれば不正な更新が防止される．

表 1 キャッシュの状態遷移
Table 1 State transition of cache line.

from	to I	to S	to M	to US	to UM	to XP
I	$r(s,s)+RB, \sigma^b, \sigma^e, RB$	$r(n,n)$	$w(n)+W(n)$	$r(s,n)$	$w(s)+W(s)$	$r(n,s)$
S	$W(n), v$	$r(n), \sigma^b, \sigma^e, RB$	$w(n)+W(n)$	$r(s)$	$w(s)+W(s)$	$W(s)$
M	$W(n,c), v+WB$	$R(c)$	$r(n), w(n), \sigma^b, \sigma^e, RB$	$r(s)+WB$	$w(s)+WB$	$W(s,c)+WB$
US	$W(*)+RB, v+RB, RB$	σ^e	—	$r(s)$	$w(s)+W(s)$	—
UM	$W(*,m)+RB, v+RB, RB$	—	σ^e	—	$r(s), w(s), R(m)$	—
XP	$r(s)+RB, w(s)+RB,$ $W(n), RB, \sigma^b, \sigma^e, v$	—	$w(n)+W(n)$	—	—	$r(n), W(s)$

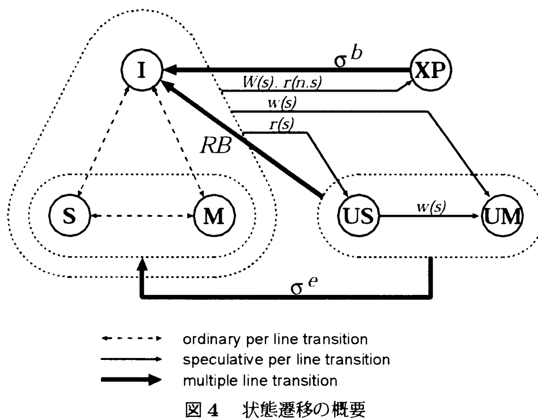

図 4 状態遷移の概要
Fig. 4 Overview of state transition.

メモリに保存する．

同様に投機モードでの書き込み（$w(s)$）では **UM** に遷移し，旧状態が **M** であればライトバックを行う．またこの投機書き込みを通知された他のキャッシュでは（$W(s)$），保持している値がいずれ無効になることを示す状態 **XP** に遷移する．その際，旧状態が **M** であればやはりライトバックを行う．この **XP** への遷移は，キャッシュミスにより **UM** 状態のラインを得ようとした場合にも生じ（$r(n,s)$），その際にはメモリに保存された投機開始前の値が返される．

(2) 同期の成立確認によって投機が成功すると（σ^e），**US** のラインはすべて **S** に，また **UM** のラインはすべて **M** に遷移する☆．この結果投機的状態は解消し，次の投機開始までは通常の状態遷移を行う．

(3) 一方，**US** または **UM** のラインに対する他プロセッサの書き込み，これらのラインのリプレース，あるいは **XP** のラインへの自プロセッサからの投機的アクセスが生じると，プロセッサは投機開始時点までロールバックする（RB）．同時にすべての **US** と **UM** のラインは **I** に遷移し，メモリに保存した値を参照できるようにする☆☆．なお投機失敗の直接原因となったラインを除いて，**US** → **I** の遷移を **US** → **S** とすることもでき，4.4.1 項で述べるように性能面では優れている．しかしこの一括状態遷移を実現すると，3.3 節で述べる機能メモリのハードウェア・コストが一般的には増加する．

(4) 次の同期点に達すると（σ^b），**XP** のラインの値はすでに古いものとなっている可能性があるため，すべての **XP** のラインが **I** に遷移し，最新の値を参照できるようにする．

3.2 状態遷移の詳細

表 1 は前節で述べた状態遷移を，以下の記号で示す遷移要因事象のすべてついて完全に示したものである．また + に続く記号は，遷移にともなう動作を表す．

- $r(\{s|n\}\ [, \{s|n\}])$
  自プロセッサからの読み出し．第 1 引数が自プロセッサが投機モードにあるか（s）否か（n）を示す．キャッシュミスが生じた場合は，第 2 引数がシステム中に **UM** 状態のラインが存在するか（s）か否か（n）を示す．
- $w(\{s|n\})$
  自プロセッサからの書き込み．引数は自プロセッサが投機モードにあるか（s）否か（n）を示す．
- $R(\{c|m\})$
  他プロセッサからの読み出し．引数はキャッシュの値を返すか（c），メモリの値を返すか（m）を示す．
- $W(\{s|n|*\}\ [, \{c|m\}])$
  他プロセッサからの書き込み．第 1 引数が要求元プロセッサが投機モードにあるか（s）否か（n），あ

☆ 後述の表 1 に示すように，**XP** のラインもすべて **I** に遷移するが，この遷移は本質的に必要なものではない．

☆☆ 前項と同様に **XP** のラインもすべて **I** に遷移するが，本質的に必要なものではない．

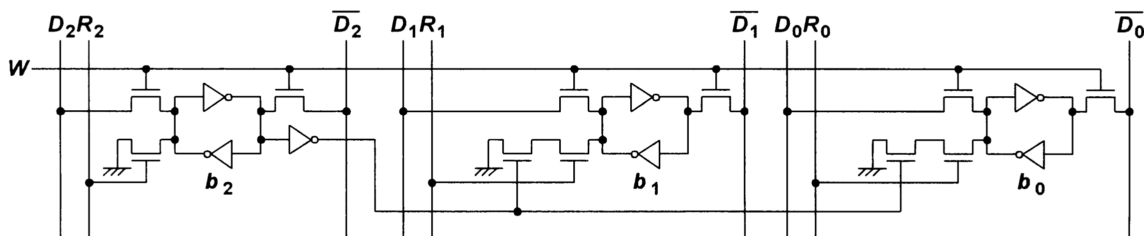

図5 状態ビット用のメモリセル
Fig. 5 Memory cells for cache state bits.

るいはモードに無関係であるか（$*$）を示す．キャッシュラインを返す場合には，第2引数がキャッシュの値を返すか（c），メモリの値を返すか（m）を示す．また $+W(\{n|s\})$ は，自プロセッサからの書き込み時に他プロセッサへ投機的（s）または非投機的（n）な書き込み要求を出すことを示す．

- v
  リプレース．
- σ^b
  バリアへの到着（$B_i^a(j)$）など，同期操作の開始への到達．
- σ^e
  バリアの成立確認（$B_i^d(j)$）など，同期操作の完了．
- RB
  ロールバック．$+RB$ は状態遷移とともにロールバックが生じることを示す．
- $+WB$
  ライトバックをともなう状態遷移であることを示す．

なお，ロールバックは表に示していない他のイベントによって引き起こされることもある．たとえば，メモリ・アクセス例外が生じたならば，それはおそらく投機的参照によって得た不正なポインタに起因するものであろうから，ロールバックすべきである．また性能の観点からは，TLB ミスについても同様にロールバックが必要である．

3.3 機能メモリによる実現

以上述べた状態遷移のうち，σ^b，σ^e，RB により引き起こされるものは，複数のキャッシュラインに対して行われる．この一括状態遷移は，以下の機能を備えた簡単な機能メモリを用いることにより実現できる．

(1) すべてのワードについて，あるビット b_r を 0 にする機能：$reset(b_r)$

(2) すべてのワードについて，あるビット b_m が 1 であれば別のビット b_r を 0 にする機能：$masked_reset(b_m, b_r)$

すなわちこれらの機能を用い，**表2** に示すように状態

表2 状態のエンコード
Table 2 Encoding of cache states.

state	$b_2b_1b_0$	σ^b	σ^e	RB
I	000	**I** (000)	**I** (000)	**I** (000)
S	001	**S** (001)	**S** (001)	**S** (001)
M	010	**M** (010)	**M** (010)	**M** (010)
XP	100	**I** (000)	**I** (000)	**I** (000)
US	101	—	**S** (001)	**I** (000)
UM	110	—	**M** (010)	**I** (000)

をエンコードすることにより，以下の操作によって一括状態遷移を行うことができる．

$$\sigma^b : reset(b_2)$$
$$\sigma^e : reset(b_2)$$
$$RB : masked_reset(b_2, b_1);\ masked_reset(b_2, b_0);\ reset(b_2);$$

図5 は，このような機能を実現するためのメモリセルの構成例であり，b_i の通常アクセスはワード線 W とビット線 D_i により行われ，（マスク付き）リセットは R_i を Hi にすることにより行われる．図に示すように，通常の CMOS-SRAM に付加されるトランジスタ数はワードあたり 7 個，すなわちほぼ 1 ビットの増加分にすぎず，ハードウェアの実装コストは許容範囲であるといえる．またリセット操作に要する時間を通常アクセスよりも長くしても性能に大きなダメージは与えないため，許容される電力消費量に見合った設計も可能である．

3.4 ベース状態の追加

本節では，前節までに述べたベース状態 {**M**,**S**,**I**} に新たなベース状態を加えたコヒーレント・キャッシュに対する，specMEM の実装モデルについて述べる．まず **E**（Exclusive）を追加した MESI[12]を例にあげて，対応する投機的状態 **UE** に関する状態遷移の定義方法を示す．以下 $W(*) + RB :$ **UE** $\rightarrow$ **I** のような表記は，':' の左辺の遷移要因事象（3.2 節参照）によって右辺の状態遷移が起こることを示す．また一般に追加される状態を **x**，またそれに対応する投機的状態を **Ux** とする．

(1) $r(s,n): \mathbf{I} \rightarrow \mathbf{UE}$（ただし他のキャッシュにコピーが存在しない場合）
他のキャッシュにコピーが存在しない場合の非投機的な読み出しミスの遷移 $r(n,n): \mathbf{I} \rightarrow \mathbf{E}$ に対応する．一般に $r(n,n): \mathbf{I} \rightarrow \mathbf{x}$ に対応して $r(n,s): \mathbf{I} \rightarrow \mathbf{Ux}$ となる．

(2) $r(s): \{\mathbf{E}, \mathbf{UE}\} \rightarrow \mathbf{UE}$
E に関する非投機的遷移 $r(n): \mathbf{E} \rightarrow \mathbf{E}$ に対応する．一般には clean なライン **x** について $r(n): \mathbf{x} \rightarrow \mathbf{x}$ であるので，$r(s): \{\mathbf{x}, \mathbf{Ux}\} \rightarrow \mathbf{Ux}$ となる．

(3) $r(s) + WB: \mathbf{M} \rightarrow \mathbf{UE}$
M に関する投機的読み出しヒットは状態保存のためのライトバックをともなう．その結果ラインは投機的，排他的かつ clean になるので，**UE** に遷移する．一般には dirty な非投機的状態 $\mathbf{x}_d$ とそれを clean にした状態 $\mathbf{x}_c$ について $r(s) + WB: \mathbf{x}_d \rightarrow \mathbf{Ux}_c$ となる．

(4) $w(s): \{\mathbf{E}, \mathbf{UE}\} \rightarrow \mathbf{UM}$
E に関する非投機的遷移 $w(n): \mathbf{E} \rightarrow \mathbf{M}$ に対応する．一般には $w(n)[+W(n)]: \mathbf{x} \rightarrow \mathbf{x}_M$ に対応して $w(s)[+W(s)+WB]: \{\mathbf{x}, \mathbf{Ux}\} \rightarrow \mathbf{UM}$ となる．すなわち非投機的書き込みヒットでの遷移先にかかわらず必ず **UM** に遷移し，他のキャッシュのコピーはすべて **XP** に遷移する．また **x**，**Ux** が dirty であればライトバックをともなう．

(5) $R(c): \mathbf{UE} \rightarrow \mathbf{US}$
E に関する非投機的遷移 $R(c): \mathbf{E} \rightarrow \mathbf{S}$ に対応する．一般には $R(\{c|m\}): \mathbf{x} \rightarrow \mathbf{x}'$ に対応して $R(\{c|m\}): \mathbf{Ux} \rightarrow \mathbf{Ux}'$ となる．

(6) $W(s,c): \mathbf{E} \rightarrow \mathbf{XP}$
一般に他プロセッサからの投機的書き込みにより $W(s,\{c|m\}): \mathbf{x} \rightarrow \mathbf{XP}$ となり，**x** が dirty であればライトバックをともなう．

(7) $\{W(*) + RB,\ v + RB,\ RB\}: \mathbf{UE} \rightarrow \mathbf{I}$
一般に他プロセッサからの書き込みまたはリプレースによってロールバックが生じ，$\mathbf{Ux} \rightarrow \mathbf{I}$ の一括遷移が行われる．

(8) $\sigma^c: \mathbf{UE} \rightarrow \mathbf{E}$
一般に同期操作完了によって $\sigma^c: \mathbf{Ux} \rightarrow \mathbf{x}$ の一括遷移が行われる．

上記以外の操作や状態に関しては，表1に示した状態遷移が行われる．また上記の一般化は，さらに多くの状態を持つキャッシュにも適用できる．たとえば write-invalidate 型の新 Keio プロトコル[13]では，MESI に dirty-shared-owner（**D**）と clean-shared-owner（**O**）の2状態が加えられるが☆，**D**，**O**，**UO** に関する状態遷移は**表3**のように定義できる☆☆．またベース状態が n ビットにエンコードされ，**I** の表現が $(0\ldots0)$ であるならば，3.3 節に示した方法に従って投機状態を示すビットを付加した $n+1$ ビットの機能メモリを用いれば，一括状態遷移を実現することができる．

表3 新 Keio プロトコルへの適用
Table 3 Application to New-Keio protocol.

base transition	speculative transition
$r(n): \mathbf{D} \rightarrow \mathbf{D}$	$r(s): \mathbf{D} \rightarrow \mathbf{UO}$
$r(n): \mathbf{O} \rightarrow \mathbf{O}$	$r(s): \{\mathbf{O},\mathbf{UO}\} \rightarrow \mathbf{UO}$
$w(n) + W(n): \mathbf{D} \rightarrow \mathbf{M}$	$w(s) + W(s): \mathbf{D} \rightarrow \mathbf{UM}$ $+ WB$
$w(n) + W(n): \mathbf{O} \rightarrow \mathbf{M}$	$w(s) + W(s): \{\mathbf{O},\mathbf{UO}\} \rightarrow \mathbf{UM}$
$R(c): \mathbf{D} \rightarrow \mathbf{D}$	
$R(c): \mathbf{O} \rightarrow \mathbf{O}$	$R(c): \mathbf{UO} \rightarrow \mathbf{UO}$
$W(n,c): \{\mathbf{D},\mathbf{O}\}$ $\rightarrow \mathbf{I}$	$W(s,c) + WB: \mathbf{D} \rightarrow \mathbf{XP}$ $W(s,c): \mathbf{O} \rightarrow \mathbf{XP}$
	$W(*) + RB: \mathbf{UO} \rightarrow \mathbf{I}$ $v + RB: \mathbf{UO} \rightarrow \mathbf{I}$ $RB: \mathbf{UO} \rightarrow \mathbf{I}$
	$\sigma^c: \mathbf{UO} \rightarrow \mathbf{O}$

表4 評価に用いたマルチプロセッサモデル
Table 4 Architectural parameters for evaluation.

プロセッサ数	4
プロセッサ ISA	SPARC V8
キャッシュ	
容量	64 KB
ラインサイズ	16 B
連想度	4 way
コヒーレンス	MESI
命令実行コスト（サイクル数）	
一般命令	1
バリア同期	+10
状態保存	+5
ロールバック	+10
キャッシュミス	
メモリ → キャッシュ	+20
キャッシュ → キャッシュ	+10
共有ライン無効化	+5
ライトバック	+10

4. 評　　価

4.1 評価対象ハードウェア・モデル

評価のために，**表4**に示す仕様に基づく集中共有メモリ型マルチプロセッサの命令駆動型のシミュレータを構築した．シミュレータの単純化と結果の解析を

☆ dirty-exclusive-owner は **M**，clean-exclusive-owner は **E** に相当する．
☆☆ **UD** に相当する状態はない．

容易にするために，プロセッサはサイクルあたり1命令発行かつ in-order 実行とし，メモリ・モデルは sequential consistency[14]とした．データキャッシュ3.4節に示した MESI に基づくものとし，命令キャッシュについては完全にヒットするものと仮定した．

バリア同期ついては単純な線バリア[15]をサポートするハードウェア機構の存在を想定し，バリア同期命令によってこのバリア機構とキャッシュにバリア到達が通知されるとともに，レジスタなどの状態保存が行われるものとした．また最後のプロセッサがバリアに到達すると，表に示すバリア同期コストを経過した後にすべてのプロセッサとキャッシュに同期成立が伝達されるものとした．

なおこのように単純かつ小規模の集中共有メモリのシステムを評価対象としたのは，単に性能評価を迅速に行うためであり，specMEM の適用対象がこのようなシステムに限定されることを意味するものではない．たとえば weak consistency[16]のような緩和されたメモリ・モデルの採用は容易であると同時に，通常はバリア同期の際に必要な先行アクセスの完了確認を延期することもできる．すなわちバリア同期成立条件を，すべての先行アクセスが完了し，かつ全プロセッサがバリアに到達したこととすれば，アクセス完了確認に要する遅延を隠蔽することができる．なお未完了ストアの対象アドレスは，他プロセッサからは投機的にしかアクセスされないため，仮に旧値を読み出してしまったとしてもロールバックが生じ，プログラムの意味は保存される[17]．実際，後述するワークロードの1つである LU 分解を対象に，weak consisitency を採用したシステムについても評価した結果，投機的アクセスによる性能向上率が若干ではあるが増加した．

また specMEM をディレクトリ型の分散共有メモリなど，より大規模なシステムに適用するも容易である．実際，一般的な分散型コヒーレンス・プロトコルに対する追加は，状態 **UM** のラインを所有するキャッシュが他のキャッシュからのライン取得要求を受け取ったときに，ディレクトリに対してメモリの内容を応答するように依頼することだけである．

4.2 ワークロード

評価のためのワークロードとしては，SPLASH-2[18]の中から以下の3つのプログラムを選択し☆，投機的アクセスの有無による実行時間（サイクル数）を計測した．

☆ シミュレータのメモリ容量などの制約により，問題の大きさを縮小している．

- LU 分解
  負荷の偏りがあり，かつ高負荷のプロセッサが変動するため，投機的アクセスの効果が高いと予想される．
- FFT
  負荷の偏りがまったくないため，投機的アクセスはほとんど行われないと予想される．したがって，specMEM が性能について「中立的」であることの試験となる．
- Radix Sort
  負荷が特定のプロセッサに偏るため，他のプロセッサによる投機的アクセスが行われるが，高負荷プロセッサの実行時間は短縮されない．したがって，投機的アクセスの悪影響が（もしあれば）現れるものと予想される．specMEM の中立性に関するより厳しい試験となる．

4.3 評価結果

図6 は，前述の3つのプログラムの実行時間（サイクル数）とその内訳を，投機的実行を行わない場合の実行時間を100として正規化して示したものである．内訳の中で rollback は命令再実行を含むロールバックのコストであり，cache miss には共有ライン書き込みや状態保存ライトバックのペナルティも含む．また図には，投機的アクセスを行った場合のロールバック頻度も示されている．以下，各々の結果に関する考察を述べる．

4.3.1 LU 分解

予想どおり投機的アクセスによってアイドル時間が短縮され，その結果13%程度の速度向上が得られることが明らかになった．このプログラムでは，右下隅の対角小行列ブロックの分解を行うプロセッサ $P3$ の負荷が最も高いが，$P3$ 以外のプロセッサも担当する対角小行列を分解する際にはバリアの最終到達者となる．したがって，2.1節の図2に示したような効果が生じ，アイドル時間が短縮される．

なお，最大負荷を割り当てられた $P3$ においてもアイドル時間が0にならないのは，バリアへ到達した際に直前のバリア同期の成立確認を行っているため，複数のバリア区間をオーバラップできないことによる．複数の投機的状態の保持に要するハードウェア・コストが大きいと判断したためこの制約を設けているが，制約の除去によりさらに7%程度の性能向上が見込めるため，コスト/性能のトレードオフについて今後さらに検討が必要である．

また，$P1$ と $P2$ においてロールバックのコスト（命令再実行コストを含む）が無視できない値となってい

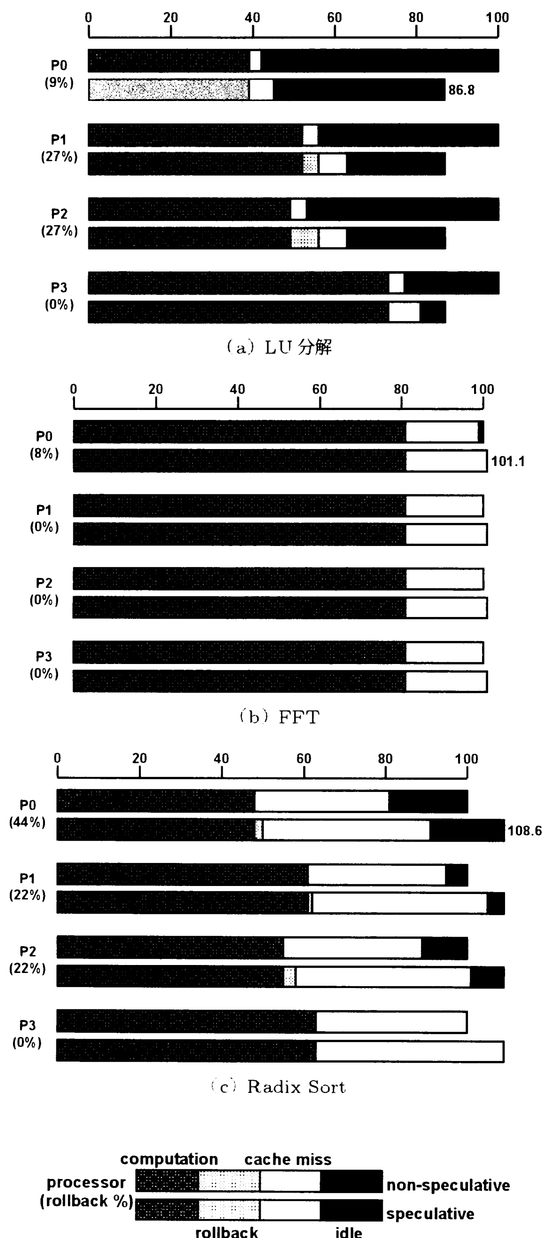

図 6 SPLASH-2 による評価結果
Fig. 6 Performance results of SPLASH-2 benchmarks.

るが，全体の性能への直接の悪影響はない．一方キャッシュミス・ペナルティはすべてのプロセッサで約 2 倍になっており，最大負荷を有する $P3$ の実行時間，すなわち全体の実行時間を約 4%増加させている．この問題については 4.4 節で議論する．

LU 分解についてはメモリ・モデルを weak consistency とし，プロセッサあたり 4 個の未完了アクセスを許容するようなシステムについても評価を行った．なおバリア同期は release と aquire を結合したものとしてモデル化されるため，release consistency [19]を採用しても同じ結果となる．評価の結果，投機的実行によって実行時間が 14.6%短縮され，sequential consistency での値 13.2%を若干上回ることが明らかとなった．この理由としては，先に述べたバリア操作にともなうアクセス遅延の隠蔽効果が考えられるが，詳細な解析は今後の課題である．

4.3.2 FFT

負荷がほぼ完全に均衡しているため，投機的アクセスの効果がまったく現れないのは予想どおりである．すなわちこのような場合，投機的アクセスにより隠蔽できるのはバリア同期自体に要するコストであるが，1 回あたりのコストが 10 サイクルという小さい値としており，またバリア同期の実行回数もわずか 12 回であるので，隠蔽効果は 1%未満にすぎない．逆にいえば，プロセッサ数の増加，あるいはハードウェア・サポートの欠如によって同期コストが大きくなれば，隠蔽効果が顕著に現れることも期待できる☆．

一方この評価の目的である specMEM の中立性については，ごくわずかの性能劣化が観測されたものの，おおむね検証できたものと考えられる．このわずかな性能劣化の要因であるキャッシュミス・ペナルティの増加については，4.4 節で議論する．

4.3.3 Radix Sort

二分木による一種のリダクション操作を行うため，そのルートである $P3$ がつねに性能上問題となるバリアの最終到達者となる．したがって，FFT と同様に投機的アクセスによって隠蔽できるコストは小さく，効果が現れないことは予想どおりである．しかし問題であるのは，キャッシュミス・ペナルティの顕著な増加，特にクリティカル・パスを実行している $P3$ での約 25%もの増加である．

後述するように投機的アクセスにともなうキャッシュミス・ペナルティの増加要因はいくつか存在するが，$P3$ についてはそのほとんどがキャッシュ・ミス回数自体が 25%以上増加したことによる．これは 4.4 節で議論するように，一種の false sharing によるものである．

4.4 キャッシュミス・ペナルティの増加

4.4.1 ロールバックと状態保存による増加

投機的アクセスを行うことにより，メモリ・アクセスのタイミングや回数が変化し，その結果としてキャッ

☆ 同期成立がキャッシュに可視であることは必要であるが，そのためのハードウェア・コストはバリア同期機構自体に比べて格段に小さい．

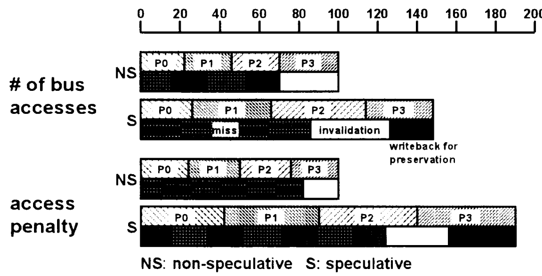

図7 LU分解のバスアクセス回数とペナルティ
Fig. 7 Bus traffic and penalty in LU decomposition.

シュミス・ペナルティが増加する可能性がある．増加要因の1つはキャッシュミス回数自体の増加であり，もう1つは投機開始時点での状態保存のためのライトバックである．そこでLU分解について，バスのアクセス回数とペナルティをプロセッサと要因ごとに詳細に測定した．**図7**はその結果を示したもので，投機的実行を行わない場合の値を100として正規化している．

この図から明らかなように，大きなロールバック・コストを生じている$P2$においてバスアクセス回数が約2倍に増加している．また$P2$でのバスアクセスの要因の内訳を調べると，キャッシュミスが約60%増加していることが明らかになった．この原因は，ロールバックが生じると投機的にアクセスされたすべてのラインが無効化されることにより，再実行時のミス率が大きくなることである．実際，$P2$の再実行時のミス率は2.1%であり，それ以外でのミス率0.6%を大きく上回っている．一方，状態保存のためのライトバックも，バスアクセス回数の増分の約半数を占めており，ミス回数の増加と同程度の比重でバスやメモリの競合を増やしていると考えられる．

この2つのバスアクセス増加は，ロールバックや状態保存を行うプロセッサだけではなく，他のプロセッサの実行時間にも影響を及ぼす．すなわち，クリティカルパスを実行する$P3$では，ミス回数がほとんど変わらないにもかかわらず，ミスペナルティが他のプロセッサと同様に約2倍となり，全体の実行時間を4%程度低下させる要因となっている．そこで，バスアクセスの増加を抑制するために，以下の改良を施すことが考えられる[17]．

- 再実行時のミス回数削減

  投機的にアクセスされたラインの中で，ロールバック時に本質的に無効化が必要なものは，直接のロールバック要因となったラインと状態が**UM**のラインのみである．しかし評価に用いたMESIベースの実装モデルでは，ラインの状態を$\{\mathbf{XP}, \mathbf{US}, \mathbf{UM}, \mathbf{UE}\} = \{100, 101, 110, 111\}$とエンコードし，3.3節に示した方法でロールバック時の一括状態遷移を行っているので，**XP**，**US**および**UE**のラインも無効化される．そこで一括状態遷移の操作を

  $$RB : masked_reset(b_2, b_1);\ reset(b_2)$$

  と変更すれば，ロールバック時の遷移は$\{\mathbf{XP}, \mathbf{UM}\} \rightarrow \mathbf{I}$および$\{\mathbf{US}, \mathbf{UE}\} \rightarrow \mathbf{S}$となり，無効化の範囲を必要最小限にとどめることができる☆．また，この変更は3章で述べたMSIベースのモデルにも適用できる．

  なお一般的には，本質的に無効化が必要なラインのみが対象となるようにするには，状態ビットの追加あるいは複数のビットによるマスクが必要となり，機能メモリのハードウェア・コストが若干増加する．たとえば評価に用いた実装モデルにおいて，ロールバック時の遷移を$\{\mathbf{XP}, \mathbf{UM}\} \rightarrow \mathbf{I}$，$\mathbf{US} \rightarrow \mathbf{S}$，$\mathbf{UE} \rightarrow \mathbf{E}$とするには，マスク操作のためにラインあたりトランジスタを1個追加しなければならない．

- 状態保存の回数削減

  評価に用いたモデルでは，$\mathbf{M} \rightarrow \{\mathbf{UM}, \mathbf{UE}\}$の遷移の際に状態保存のためのライトバックを行っている．このうち$\mathbf{M} \rightarrow \mathbf{UE}$については，ロールバックの際に**UE**のラインが無効化されることに起因しており，投機的な書き込みを行っているわけではないので，状態保存は必ずしも必要ではない．そこで**UM**の意味を「投機的読み出しを行った**M**」とし，これまで**UM**としてきた状態を**SM**（Speculatively Modified）としたうえで，以下のような状態遷移を行えば，投機的書き込みを行ったラインだけが状態保存の対象となる．

$$
\begin{aligned}
r(s) &: \{\mathbf{M}, \mathbf{UM}\} \rightarrow \mathbf{UM} \\
w(s) + WB &: \{\mathbf{M}, \mathbf{UM}\} \rightarrow \mathbf{SM} \\
w(s) &: \{\mathbf{E}, \mathbf{UE}, \mathbf{SM}\} \rightarrow \mathbf{SM} \\
w(s) + W(s) &: \{\mathbf{S}, \mathbf{I}, \mathbf{US}\} \rightarrow \mathbf{SM} \\
R(c) &: \mathbf{UM} \rightarrow \mathbf{US} \\
W(*, c) + RB &: \mathbf{UM} \rightarrow \mathbf{I} \\
W(*, m) + RB &: \{\mathbf{UE}, \mathbf{US}, \mathbf{SM}\} \rightarrow \mathbf{I} \\
v + WB + RB &: \mathbf{UM} \rightarrow \mathbf{I} \\
v + RB &: \{\mathbf{UE}, \mathbf{US}, \mathbf{SM}\} \rightarrow \mathbf{I} \\
RB &: \mathbf{SM} \rightarrow \mathbf{I}
\end{aligned}
$$

☆ プログラムが同期的であれば**XP**のラインは次の同期区間までアクセスされないので，無効化は性能に影響しない．

$$\{\mathbf{UM}, \mathbf{UE}, \mathbf{US}\} \rightarrow \{\mathbf{M}, \mathbf{E}, \mathbf{S}\}$$
$$\sigma^{c} : \mathbf{SM} \rightarrow \mathbf{M}$$
$$\{\mathbf{UM}, \mathbf{UE}, \mathbf{US}\} \rightarrow \{\mathbf{M}, \mathbf{E}, \mathbf{S}\}$$

このためには **SM** を表現するために状態ビットを1ビット追加する必要があるが，状態保存の頻度の削減が期待できる．

- 2次キャッシュの導入
  再実行時のミスや状態保存によるバスアクセスを削減する別のアプローチとして，非投機的な2次キャッシュを導入して状態保存を行うことも考えられる．この場合，2次キャッシュにはベースとなる状態（たとえば MESI）に加えて,「1次キャッシュの対応ラインが **UM/XP** の可能性がある」☆ことを意味する状態 **U** と **X** を追加し，1次キャッシュの状態が **UM** である場合には投機開始時点での値を保持するようにする．状態 **U** のラインに対する他プロセッサからのスヌープ要求に対しては必ず1次キャッシュの状態を参照し，その結果に応じて1次/2次キャッシュのどちらの値を返すかを決定する．すなわち状態 **U** のラインの真の状態は1次キャッシュの状態で定まるため，2次キャッシュに関しては一括状態遷移が不要である．また状態 **X** のラインは，他プロセッサからの書き込み要求を1次キャッシュに伝達する以外は **I** と同等とすることにより，やはり一括状態遷移が不要となる．この方法では，状態保存や再実行時の保存されたラインへのアクセスは，2次キャッシュに対してのみ行われるため，他のプロセッサによるバスやメモリのアクセスを阻害することはない．

4.4.2 false sharing による増加

キャッシュミスの増加はロールバック後の再実行だけが要因ではなく，ロールバックと false sharing の複合によっても生じる．実際，これが主要因となって Radix Sort のミスペナルティを増加させており，しかもハードウェアでの解決は容易ではない．すなわち Radix Sort では以下のような事象が生じ，クリティカル・パスを実行する *P3* のミス回数が投機を行っていないにもかかわらず25%も増加したことが明らかになった．

(1) *P3* のキャッシュに配列要素 $x[i]$ と $x[i+1]$ が属するライン L があり，状態は **M** である．
(2) *P3* がバリアに到達する前に，他のプロセッサ（たとえば *P2*）が $x[i]$ に対して投機的書き込みを行い，キャッシュミスする．この結果，*P3* のキャッシュでは L の状態が **M** → **XP** と遷移する．
(3) *P3* はバリアに到着前に他の変数への書き込みを行い，その変数の投機的読み出しを行っていた *P2* がロールバックする．この結果 *P2* のキャッシュでは L が無効化される．
(4) *P3* がバリアに到着し，*P3* のキャッシュでは L が無効化される．
(5) *P3* が $x[i+1]$ に書き込みを行い，キャッシュミスする．
(6) *P2* では再実行が行われ，$x[i]$ の書き込みを再度行い，キャッシュミスする．

以上の過程において L に関するミスが3回生じ，そのうちの1回は *P3* のキャッシュで生じていることに注意が必要である．すなわち投機的実行をしなければ *P2* において1回だけしかミスが生じないのに対して，false sharing とロールバックが組み合わされることにより2回もミスが増えてしまっている．

さらに，*P3* ではバリア到着前に i の値を計算しているため☆☆，*P2* での投機的書き込みはデータだけでなくアドレスも不正であることすら少なくない．すなわち上記のシーケンスに対応する非投機的実行では，そもそも *P2* が $x[i]$ への書き込みを行うことがなく，したがってまったくキャッシュミスが生じないという事象も観測されている．

これらの事象は，ハードウェアのみの判断による盲目的な投機的実行の限界を示している．したがって，コンパイラの最低限の支援として，時間的に近いアドレス生成と参照の依存関係がバリアを越えて存在する（可能性が高い）場合には，投機を行わないようにハードウェアに指示する機能が必要である．一方ハードウェアに関しては，投機の成功/失敗の履歴に基づき投機の可否を適応的に判断する機構を付加すれば，上記のような行き過ぎた投機の抑制に役立つのではないかと考えられる．

5. 関連研究

5.1 細粒度通信による同期遅延隠蔽

1章でも触れたように，specMEM は共有メモリを介した粗粒度通信のオーバヘッドを除去することを目

☆ 上記の **SM** を導入すれば「**SM** の可能性がある」という意味になる．

☆☆ 正確には *P2* が i を求めるために必要な配列の計算．

的としている．これに対して，1 つの同期操作にかかわる共有データを少なくすることにより，同期成立確認までのアクセス抑止対象を本質的に必要なものに限定する，細粒度の同期/通信に関する研究も数多く行われている．

このような細粒度同期/通信の典型例としては，I-structure[20]とその共有メモリマシンへの応用[21]があげられる．この方法は，メモリの各々の語に同期用ビットを付加し，通常のロード命令によって書き込み/読み出しの依存関係が充足されていることをチェックするものである．したがって，specMEM が期待しているのと同様に，書き込みと読み出しの時間間隔が十分に大きければ，同期のための待ちが生じることはない．同様の効果は，release consistency（RC）[19]や entry consistency（EC）[22]の枠組みのもとで同期区間や同期対象の共有変数を細分化することによっても実現できる．たとえば EC では，個々の共有変数に対して固有の同期変数を割り当てることができ，ハードウェアやソフトウェアの適切な支援，たとえば十分な数の未完了アクセスの許容により，同期待ちの遅延を隠蔽することができる．

またバリア同期に関する研究の多くも，この細粒度同期/通信の効率化を目的としている[23]．なかでも Fuzzy Barrier[24]などの面バリアは，バリアの入口と出口の間での命令の実行を許容する点で，specMEM と一定の類似性を持っている．しかし specMEM とは異なり，操作の正当性が静的に保証されるもののみが許容されるため，同期操作の遅延隠蔽効果は小さい．実際 specMEM の効果が大きい LU 分解では，バリア同期直後に行われる局所的な操作はループ制御やアドレス計算などの 10 命令程度であり，ほとんど遅延隠蔽効果はない．また specMEM においても入口と出口を分離し，投機的実行を出口から開始することは可能である．一方 Elastic Barrier[25]は，入口と出口にはさまれた区間をオーバラップできるという specMEM にはない特徴を有しており，RC や EC と同じような同期区間の細分と同期遅延の隠蔽が可能である．

しかしこれらの細粒度同期/通信の機構は，同期遅延の隠蔽（あるいは削減）のために特別なハードウェア機構を必要とするだけではなく，プログラマやコンパイラによる同期操作の細かくかつ適切な挿入が必要である．たとえば

```
parallel for (p=0 ; p<2 ; p++) {
    for (i=0 ; i<n ; i++) a[p][i] = ... ;
    for (i=0 ; i<n ; i++) ... = a[1-p][i] ;
}
```

のようなプログラムに対して，配列 a に関する書き込み/読み出しの同期遅延を隠蔽するためには，2 つの for ループを適切に細かく分割して同期操作を挿入しなければならない．一方 specMEM では，2 つのループの間に単にバリア同期を挿入するだけで，同期遅延を隠蔽することができる．

5.2 コヒーレント・キャッシュと投機

コヒーレント・キャッシュと投機的実行を関係付けた研究は，適切なコヒーレンス維持操作の選択に関するものと，specMEM のように潜在的には危険なアクセスを投機的に行うものに大別される．前者は様々なコヒーレンス維持操作のバリエーションの中から，命令ごとの[26]，あるいはメモリ・ライン（ブロック）ごとの[27),28]履歴に基づいて最適な操作を予測し，それを投機的に行うことによって遠隔アクセスの遅延を削減しようというものである．したがって specMEM とは目的や手法が大きく異なるが，4.4.2 項で触れた投機の可否の適応的制御に，これらの手法を応用できる可能性がある．

一方後者の例としては，sequential consistency（SC）[14]を要求するプログラムにおけるメモリ・アクセス順序を投機的に入れ換える，Gniady らの SC++[7]があげられる．SC++には，SC に基づくプログラムに潜在する同期操作の成立を仮定すること，投機の失敗を他のプロセッサからの書き込み要求によって検出することなど，specMEM との共通点がいくつか存在する．しかし投機的アクセスの履歴を連想バッファに記憶する点が大きく異なり，2.2 節で述べたように大きな同期遅延を隠蔽することができない．

別の例としては，分岐予測に基づく逐次プログラムの投機的マルチスレッド実行を，集中共有メモリ型マルチプロセッサをベースに実現する Gopal らによる Speculative Versioning Cache（SVC）[29]がある．SVC は我々の初期の報告[3]と同時期に提案されたもので，他の研究[8),9]が SC++と同様の連想バッファを用いているのに対し，キャッシュを利用した投機の成否の検出や投機的書き込みの実現法など，specMEM との共通点が多い．

しかし投機的マルチスレッド実行では多数の細粒度スレッドが並列実行されるため，1 つのメモリ・アドレスに対して各々のキャッシュがそれぞれ異なった値を同時に持ちうることが必要となる．そのため，スレッドに応じて適切なキャッシュを選択する集中的な制御論理が必要であり，プロセッサの数は自ずから限定される．一方 specMEM は粗粒度並列タスクを対象としているので，1 つのアドレスに対する値はたかだか 2

種類であり，先に述べたように大規模な分散共有メモリにも簡単に適用できる．

6. おわりに

本論文では，バリア同期に対する投機的なメモリ・アクセスを行う機構specMEMと，SPLASH-2に含まれるベンチマークを用いた評価について述べた．specMEMはコヒーレント・キャッシュに簡単な拡張を施すことにより実現でき，投機の開始，成功，失敗にともなう操作を定数時間で実行できるという特徴を持っている．評価の結果，LU分解のように負荷の変動によって同期区間が伸縮するようなプログラムについては，specMEMが有効であることが明らかになった．また投機によるキャッシュミス・ペナルティの増加についても解析を行い，それに基づくspecMEMの改良法についても考察した．

一方，FFTではわずかに，またRadix Sortでは少なからず性能が悪化することも明らかになった．特にRadix Sortについて悪化要因を解析した結果，一種のfalse sharingが原因であること，過度の投機を抑制するためのコンパイラによる支援が必要であることを明らかにした．

今後の課題としてはまず，上記のハードウェア機構の改良とその評価．およびコンパイラ支援に基づく投機可否判断の方式検討があげられる．このほか，プロセッサ数の増加やそれにともなう分散共有メモリの採用，ワークロードの多様化などが，評価に関する課題としてあげられる．また同期区間のオーバラップ，ロックなどバリア同期以外の同期操作に対する適用など，方式や機構に関する検討も行う予定である．

謝辞　本研究の一部は，並列分散処理コンソーシアム（PDC）の研究テーマ「超並列共有メモリ型マルチプロセッサの研究」による．

参考文献

1) Smith, M.D., Johnson, M. and Horowitz, M.A.: Limits on Multiple Instruction Issue, *Proc. Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, pp.290–302 (1989).
2) 中島　浩：投機に投資しよう，情報処理，Vol.40, No.2, pp.195–201 (1999).
3) 佐藤貴之，中島　浩：同期操作に対するメモリ・アクセスの投機的実行の提案，情報処理学会研究報告，98-ARC-129, pp.19–24 (1998).
4) 佐藤貴之，中島　浩，大野和彦：同期操作に対するメモリ・アクセスの投機的実行の評価，情報処理学会研究報告，99-ARC-133, pp.25–30 (1999).
5) Sato, T., Ohno, K. and Nakashima, H.: A Mechanism for Speculative Memory Accesses Following Synchronizing Operations, *Proc. Parallel and Distributed Processing Symp.*, pp.145–154 (2000).
6) Smith, J.E.: Dynamic Instruction Scheduling and the Astronautics ZS-1, *Computer*, Vol.22, No.7, pp.21–35 (1989).
7) Gniady, C., Falsafi, B. and Vijaykumar, T.N.: Is SC + ILP = RC?, *Proc. 26th Intl. Symp. Computer Architecure* (1999).
8) Franklin, M. and Sohi, G.S.: ARB: A Hardware Mechanism for Dynamic Reordering Memory References, *Proc. Intl. Symp. High-Performance Computer Architecutre* (1998).
9) 玉造潤史，松本　尚，平木　敬：Loopを並列実行するアーキテクチャ，情報処理学会研究報告，96-ARC-119, pp.61–66 (1996).
10) Seo, K. and Yokota, T.: Pegasus: A Risc Processor for High-Performance Execution of Prolog Programs, *Proc. Intl. Conf. Very Large Scale Integration*, pp.261–274 (1987).
11) Smith, M.D., Lam, M.S. and Horowitz, M.A.: Boosting Beyond Static Scheduling in a Superscalar Processor, *Proc. 17th Intl. Symp. Computer Architecure*, pp.344–355 (1990).
12) Stenstrom, P.: A Survey of Cache Coherence Schemes for Multiprocessors, *Computer*, Vol.23, No.6, pp.14–24 (1990).
13) Terasawa, T., Ogura, S., Inoue, K. and Amano, H.: A Cache Coherece Protocol for Multiprocessor Chip, *Proc. IEEE Intl. Conf. Wafer Scale Integration*, pp.238–247 (1995).
14) Lamport, L.: How to Make a Multiprocessor Computer that Correctly Execute Multiprocessor Programs, *IEEE Trans. Computers*, Vol.28, No.9, pp.690–691 (1979).
15) O'Keefe, M.T. and Dietz, H.G.: Hardware Barrier Sylıchronization: Static Barrier MIMD (SBM), *Proc. Intl. Conf. Parallel Processing*, Vol.1, pp.35–42 (1990).
16) Adve, S.V. and Hill, M.D.: Weak Ordering – A New Definition, *Proc. 17th Intl. Symp. Computer Architecure* (1990).
17) Nakashima, H., Sato, T., Matsuo, H. and Ohno, K.: Implementation Issues of the Speculative Memory Access Mechansim specMEM, `http://www.para.tutics.tut.ac.jp/~nakasima/papers/specmem-imp.ps.gz` (2000).
18) Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methdological Considerations, *Proc. 22nd Intl. Symp. Computer Ar-*

chitecure, pp.24–36 (1995).

19) Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Hennessy, J.: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Proc. 17th Intl. Symp. Computer Architecure*, pp.15–26 (1990).

20) Arvind, Nikhil, R.S. and Pingali, K.: I-structure: Data Structures for Parallel Computing, *ACM Tans. Prog. Lang. and Syst.*, Vol.11, No.4, pp.598–632 (1989).

21) Goshima, M., Mori, S., Nakashima, H. and Tomita, S.: The Intelligent Cache Controller of a Massively Parallel Processor JUMP-1, *Innovative Architecture for Future Generation High-Performance Processors and Systems*, Veidenbaum, A. and Joe, K. (Eds.), pp.116–124, IEEE (1997).

22) Bershad, B.N., Zekauskas, M.J. and Sawdon, W.A.: The Midway Distributed Shared Memory System, *Proc. IEEE COMPCON* (1993).

23) 山家　陽，村上和彰：バリア同期モデル—Taxonomyと新モデルの提案，および，モデル間性能比較，並列処理シンポジウムJSPP'93，pp.119–126 (1993).

24) Gupta, R.: The Fuzzy Barrier, *Proc. Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, pp.54–63 (1989).

25) Matsumoto, T., Tanaka, T., Moriyama, T. and Uzuhara, S.: MISC: A Mechanism for Integrated Synchronizatin and Communication, *Proc. Intl. Conf. Parallel Processing*, Vol.1, pp.161–170 (1991).

26) Kaxiras, S. and Goodman, J. R.: Improving CC-NUMA Performance Using Instruction-Based Prediction, *Proc. 5th Intl. Symp. High-Performance Computer Architecture* (1999).

27) Mukherjee, S.S. and Hill, M.D.: Using Prediction to Accelerate Coherence Protocols, *Proc. 25th Intl. Symp. Computer Architecure* (1998).

28) Lai, A.C. and Falsafi, B.: Memory Sharing Predictor: The Key to a Speculative Coherent DSM, *Proc. 26th Intl. Symp. Computer Architecure* (1999).

29) Gopal, S., Vijaykumar, T., Smith, J.E. and Sohi, G.S.: Speculative Versioning Cache, *Proc. Intl. Symp. High-Performance Computer Architecutre* (1998).

(平成12年1月28日受付)
(平成12年6月 2 日採録)

佐藤　貴之

1974年生．1999年豊橋技術科学大学大学院工学研究科情報工学専攻修士課程修了．同年ソニー（株）入社．在学中は共有メモリ型並列計算機のアーキテクチャに関する研究に従事．

松尾　治幸

1977年生．豊橋技術科学大学大学院工学研究科情報工学専攻修士課程在学中．共有メモリ型並列計算機のアーキテクチャに関する研究に従事．

大野　和彦（正会員）

1970年生．1998年京都大学大学院工学研究科情報工学専攻博士後期課程修了．同年豊橋技術科学大学助手．並列プログラミング言語の設計と最適化に関する研究に従事．博士（工学）．

中島　　浩（正会員）

1956年生．1981年京都大学大学院工学研究科情報工学専攻修士課程修了．同年三菱電機（株）入社．推論マシンの研究開発に従事．1992年京都大学工学部助教授．1997年豊橋技術科学大学教授．並列計算機のアーキテクチャ等，並列処理に関する研究に従事．工学博士．1988年元岡賞，1993年坂井記念特別賞受賞．IEEE-CS，ACM，ALP，TUG各会員．