計算機アーキテクチャ 133－5
（1999．5．21）

同期操作に対するメモリ・アクセスの投機的実行の評価

佐藤貴之[†,☆] 中島 浩[†] 大野和彦[†]

共有メモリ型並列計算機における同期処理オーバヘッドを削減する手法として，我々は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を提案している。この機構の特徴は，投機失敗の検出やそれに伴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，投機的に行な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。

今回の報告では，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述べる。評価の結果，LU 分解では 15％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我々の機構が有効であることが明らかになった。

Evaluation of Speculative Memory Accesses Following Synchronizing Operations

Takayuki Sato,[†,☆] Hiroshi Nakashima[†] and Kazuhiko Ohno[†]

In order to reduce the overhead of synchronizing operations of shared memory multiprocessors, we have proposed a mechanism 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, 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 describes an evaluation result of our speculative access mechanism 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, is improved by 15％.

1. はじめに

共有メモリ型並列計算機におけるプロセッサ間通信は，共有メモリのアクセスと同期操作の組合せによって実現される。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なうことはできない。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。

この同期成立確認までの操作禁止は，操作の正当性を保証するためのものであるため，禁止対象や期間を必要最小限に留めることが困難であり，そのため不必要なオーバヘッドが生じることがしばしばある。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 $(B_1^a(1) < B_2^a(1))$ で，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 $B_2^g(1) - B_1^a(1)$ は，無駄なアイドル時間であるといえる。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 $B_1^g(1)$ 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。

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

† 豊橋技術科学大学情報工学系
Dept. of Computer and Information Sciences, Toyohashi Univ. of Tech.
☆ 現在，ソニー(株)
Presently with Sony Corp.

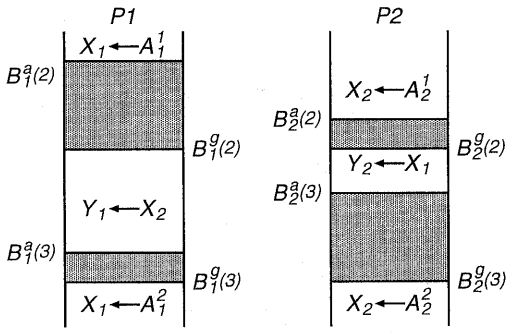

図 1　バリア同期によるデータ依存制約の充足

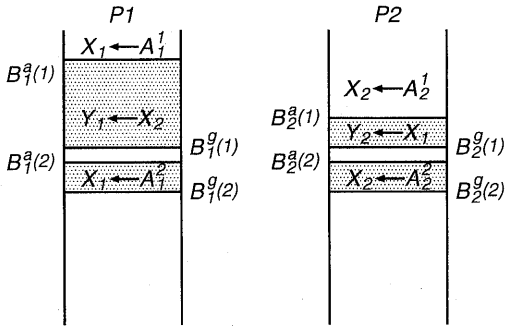

図 2　投機的メモリ・アクセス

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方式を提案している[3)]。この方式では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・アクセスを行なってオーバヘッドを除去し，実際には依存制約が満たされていない時に限って再実行により操作の正当性を保証する☆。

以下本報告では，第 2 章で投機的アクセスの概要を，第 3 章では実装モデルをそれぞれ示し，第 4 章で 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る。

2. 投機的メモリ・アクセス

2.1 投機による高速化

我々が提案している投機的メモリ・アクセス方式では，同期操作によって充足されるデータ依存制約が，同期成立確認以前にも充足されていることを仮定し，同期操作以降の処理を投機的に行なう。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_i $(i = 1, 2)$ のフロー依存制約を充足するためのバリア同期に $B_j^a(1)$ $(j = 1, 2)$ で到達すると，その成立を確認することなく後続の操作を続行する。その結果，同期成立確認が $B_j^g(1)$ でなされる以前に P_j は X_i を参照する。しかし X_i の更新／参照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，この投機は成功してアイドル時間が完全に除去される。

また同様に，X_i の逆依存制約を充足するためのバリア同期に $B_j^a(2)$ で到達後，P_j による X_i の更新も $B_j^g(2)$ 以前に投機的に行なわれる。この場合も参照／更新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。

☆ 正確には満たされていることが保証できなくなった時に再実行を行なう。

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，投機的に行なわれた全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なわれる可能性は常にあり，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 の投機的参照が，P_2 による X_2の更新に先行し，その結果不正な値を参照してしまっている。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，まず不正な投機的参照を行な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なう必要がある。

我々が提案している機構では，この投機失敗の検知と無効化をコヒーレント・キャッシュを拡張して実現している。まず，バリア到達から確認までの期間においてアクセスされたキャッシュ・ラインには，全て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 の参照前の状態が S (Shared) であるとすると，投機的参照の結果 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更新要求を受けとると，不正値を参照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 の更新要求によって，P_1 のキャッシュが不正参照を検知する。

次に行なう計算状態変化の無効化は，キャッシュの

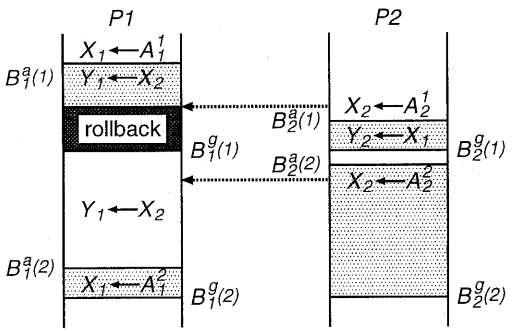

図 3　投機の失敗

ライトバック機構を利用しておこなう。すなわち，状態 UM のラインについては投機開始時点の値がメモリに保存されているようにし，投機失敗が検知された時点で UM のラインを全て無効化することにより，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☆。このため UM への遷移の際には，必要に応じてライトバックを行なう。この結果，メモリに関する計算状態変化は全て無効化されるので，他の計算状態をシャドーレジスタ[4]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なうことができる。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。

なおメモリの投機的更新は，アドレスと更新値（あるいは旧値）を連想検索可能なバッファに記録することによっても実現できるが[5]，キャッシュによる実現には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全て普通の状態に戻す操作も定数時間で実現する。

2.3　不正な投機的更新の防止

図 3 の例では，P_1 でのロールバックの後，X_1 の値が再び参照される。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 の更新を行なう。この更新は図に示すように P_1 による X_1 の参照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。

この不正な更新を行な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参照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更新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，更新後の値 A_2^2 ではなく，$B_2^a(2)$ の時点での値である A_2^1 をメモリから返すことが可能である。

しかしこの値は，図のように 2 番目のバリアに関する投機が成功すると，今度は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が，P_2 による更新操作は既に完了しているので，P_1 はキャッシュした値 A_1^2 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で UM 状態にあるラインを参照して得たラインは，XP (eXPiring) という特別な状態とし，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，X_2 を改めて参照する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なうことができる。

また図の例では，P_2 が X_2 を更新する時点で P_1 は X_2 をキャッシュしていないが☆☆，投機的更新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，更新要求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，投機的更新によって無駄なキャッシュミスが生じることを防止できる。

3.　実装モデル

前章で述べたように，我々が提案している投機的メモリ・アクセス機構は，コヒーレント・キャッシュを拡張する形で実装される。表 1 は，コヒーレンス・プロトコルを write-invalidate とし，ベースとなる状態を MSI (Modified/Shared/Invalidate) とした時のキャッシュの状態遷移を示したものである。状態遷移に関する表中の記号は以下の意味を持つ。

- $r(n/s\ [,n/s])$
  自プロセッサからの読出。第 1 引数が自プロセッサが投機状態 (s) か否か (n) を示す。キャッシュミスが生じた場合は，第 2 引数がラインを供給したキャッシュの状態が投機状態 (s) か否か (n) を示す。
- $w(n/s)$
  自プロセッサからの書込。引数は自プロセッサが投機状態 (s) か否か (n) を示す。
- $R(c/m)$
  他プロセッサからの読出。引数はキャッシュの値

☆ 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全て無効化する。

☆☆ ロールバックの際に無効化されている。

表1　キャッシュの状態遷移

from	to					
	I	S	M	US	UM	XP
I	$r(s,s)$+RB, B^a, B^g, RB	$r(n,n)$	$w(n)$+$W(n)$	$r(s,n)$	$w(s)$+$W(s)$	$r(n,s)$
S	$W(n)$, v	$r(n)$, B^a, B^g, RB	$w(n)$+W	$r(s)$	$w(s)$+$W(s)$	$W(s)$
M	$W(n,c)$, v+WB	$R(c)$	$r(n)$, $w(n)$, B^a, B^g, RB	$r(s)$+WB	$w(s)$+WB	$W(s,c)$
US	$W(*)$+RB, v+RB, RB	B^g	—	$r(s)$	$w(s)$+$W(s)$	—
UM	$W(*,m)$+RB, v+RB, RB	—	B^g	—	$r(s)$, $w(s)$, $R(m)$	—
XP	$W(n)$, B^a, v	—	$w(n)$+$W(n)$	—	—	$r(n)$

を返すか (c)，メモリの値を返すか (m) を示す。

- $W(n/s\ [,c/m])$
  他プロセッサからの書込。第1引数が要求元プロセッサが投機状態 (s) か否か (n) 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込時に他プロセッサへ投機的 (s) または非投機的 (n) な書込要求を出すことを示す。
- v
  リプレース。
- B^a
  バリア同期への到達。
- B^g
  バリア同期の成立確認。
- 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。
- $+WB$
  ライトバックを伴う状態遷移であることを示す。

また状態遷移の内，B^a, B^g, RB によるものは，複数のキャッシュラインに対して行なわれる。この一括状態遷移は；

(1)　全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)　全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：$msk\_reset(b_m, b_r)$

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。すなわちキャッシュラインの状態情報を表2に示すようにエンコードすると，一括状態遷移は以下のように実現できる。

B^a : $reset(b_2)$
B^g : $reset(b_2)$
RB : $msk\_reset(b_2, b_1)$; $msk\_reset(b_2, b_0)$;
　　$reset(b_2)$;

このような機能を実現するために通常の CMOS-SRAM に付加されるトランジスタ数はビットあたり7個であり，ほぼ1ビットの増加分に過ぎない[3]。

表2　状態情報のエンコード

state	$b_2b_1b_0$	B^a	B^g	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)	—	—
US	101	—	S (001)	I (000)
UM	110	—	M (010)	I (000)

4. 評　　価

4.1 評価の方法

評価のために，表3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シミュレータを構築した。なお前章で述べたキャッシュ状態遷移はMSIをベースとしたものであるが，評価に用いたキャッシュはMESIをベースとしたものである。ただし，この Exclusive Clean の導入に伴う投機的状態の追加はほぼ自明であり，キャッシュタグに要するハードウェア・コストは全く増加しない。

バリア同期ついては，単純な線バリアをサポートするハードウェア機構の存在と，その同期成立がキャッシュに可視であることを想定した。また表に示すバリア同期コストは，最後のプロセッサがバリアに到達してから同期成立が確認されるまでの時間とした。

評価のためのワークロードとしては，SPLASH-2[6]の中から以下の3つのプログラムを選択し，投機的

表3　評価に用いたマルチプロセッサモデル

プロセッサ数	4
キャッシュ	
容量	64KB
ラインサイズ	16B
連想度	4way
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	+20

アクセスの有無による実行時間（サイクル数）を計測した。

- 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
  負荷の偏りが全くないため，投機的アクセスはほとんど行なわれないと予想される。
- 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なわれるが，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。

4.2 評価結果

図4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，投機的実行を行な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。また図にはバリア同期の数と，投機的アクセスを行なった場合のロールバック回数も示されている。以下，各々の結果に関する考察を述べる。

4.2.1 LU 分解

予想通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 15 %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主としてプロセッサ $P3$ が高負荷となるが，$P3$ 以外のプロセッサが最終到達者となるようなバリアもあるため，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なっているため，複数のバリア区間をオーバラップできないことが主な要因である。複数の投機的状態の保持に要する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，無視できない値のロールバックのコスト（命令再実行コストを含む）と，それに伴うキャッシュミス・ペナルティの増加が見られるが，いずれも全体の性能を悪化させるには至っていない。

4.2.2 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全く現れないのは予想通りである。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回当たりのコストが10サイクルという小さい値としているため，隠蔽効果は0.4%程度に過ぎない。逆に言えば，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☆。

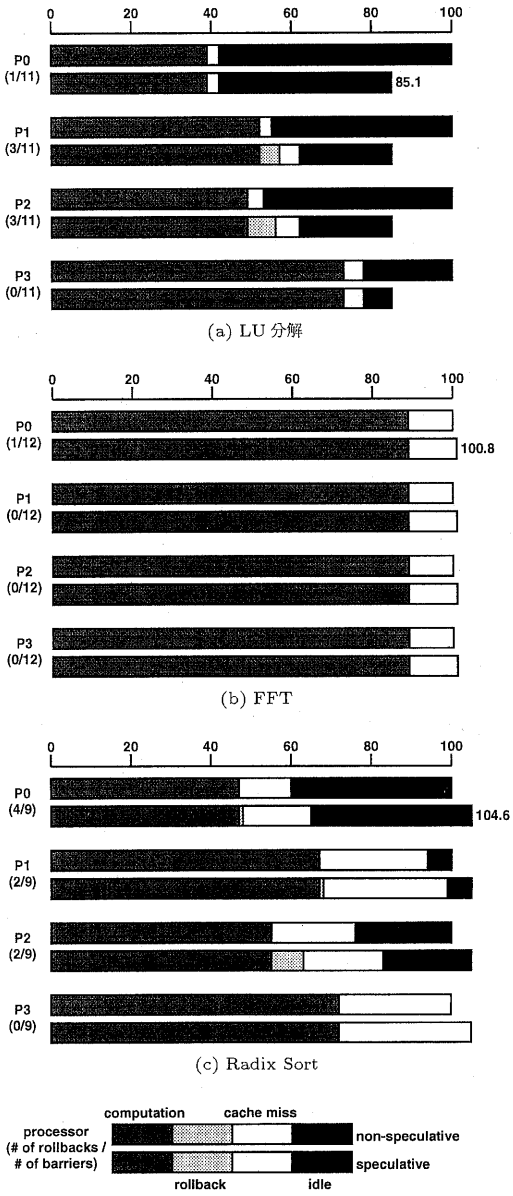

図4　SPLASH-2 による評価結果

一方，投機的アクセスによってキャッシュミス・ペナルティがごく僅かではあるが増加し，その結果全体の実行時間が僅かに増加してしまっている。この問題については後述する。

☆ 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。

4.2.3 Radix Sort

全てのバリアについて $P3$ が最終到達者となるため，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通りである。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 $P3$ での増加である。

後述するように投機的アクセスに伴うキャッシュミス・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・ミス回数自体の増加により説明することができる。すなわち false sharing が要因であると考えられるが，詳細な解析は今後の課題である。

4.3 キャッシュミス・ペナルティの増加

投機的アクセスを行なうことにより，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。増加要因の一つはキャッシュミス回数自体の増加であり，もう一つはメモリ・トラフィックの増加である。

キャッシュミス自体の増加要因としては，ロールバックと false sharing の二つが挙げられる。まずロールバックが生じると，投機的にアクセスされた全てのラインが無効化されるため，再実行時のミス率は非常に大きな値となる。たとえば Radix Sort では，$P0$ における再実行時のミス率は 32 %であり，それ以外の部分でのミス率 2.5 %を大きく上回っているため，ミスペナルティが顕著に増加している。ただし投機的にアクセスされたラインの中で，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 のラインのみであり，機能メモリのハードウェア・コストを若干増やすことにより，ミス率やペナルティを大幅に削減できるものと期待できる。

もう一つのミス増加要因である false sharing は，同一ラインに含まれる変数のアクセス・タイミング変化に関係する。すなわち変数 x と y が同じラインに含まれていて，プロセッサ P_x が x を参照し，並行して P_y が y を更新するものとする。また投機的アクセスが行なわれない場合，この参照／更新が実時間でこの順序で生じるとする。したがって x の参照ではキャッシュミスは生じない。しかし投機的アクセスを行なうと，y の更新タイミングが早くなることがあり，その結果 x の参照時にキャッシュ・ミスが生じてしまう。Radix Sort での $P3$ における ミス率の増加は，このような参照／更新タイミングの変化によるものと考えられるが，さらに詳細な解析が必要である。

一方メモリ・トラフィックの増加の要因としては，上述のミス増加によって引き起こされるものの他に，M → UM の状態遷移によるライトバック，すなわち状態保存のためのメモリやバスのスループット消費が挙げられる。今回の評価ではミス増加によるトラフィック増加が支配的であったため顕在化しなかったが，状態保存は投機的アクセスに本質的でトラフィック削減は困難であるため，ミス率削減時にどの程度顕在化するかなどを解析する必要がある。

5. おわりに

本報告では，バリア同期に対する投機的なメモリ・アクセスを行なう機構と，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。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，我々の機構が有効であることが明らかになった。一方，FFT や Radix Sort については性能が僅かではあるが悪化することも明らかになった。これらのプログラムに対して投機的アクセスの効果が小さいことは予想された結果であるが，本来は性能に悪影響を及ぼしてはならないため，悪化の要因であるキャッシュミス・ペナルティ増加について今後の詳細な解析や，それに基づく機構の改善が必要である。

この他，プロセッサ数の増加やそれに伴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挙げられる。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なう予定である。

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. ASPLOS'89*, 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，情処研報，98-ARC-129, pp. 19–24 (1998).
4) Smith, M. D., Lam, M. S. and Horowitz, M. A.: Boosting Beyond Static Scheduling in a Superscalar Processor, *Proc. ISCA'90*, pp. 344–355 (1990).
5) 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，情処研報，96-ARC-119, pp. 61–66 (1996).
6) Woo, S. C., Ohara, M., Torrie, E., Singh, J. P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methdological Considerations, *Proc. ISCA'95*, pp. 24–36 (1995).