計算機アーキテクチャ　98－17
設　計　自　動　化　65－17
(1993. 1. 22)

イベント対応型キャッシュ・コヒーレンス制御方式の応用事例とその基本性能

齋藤 秀樹†, 森 眞一郎†, 中島 浩†, 富田 眞治†
田中 高士‡, David FRASER‡, 城 和貴‡

†：〒606-01 京都市左京区吉田本町
京都大学工学部情報工学科

‡：〒600 京都市下京区中堂寺南町 17 京都高度技術研究所内
(株) クボタ コンピュータ事業推進室

あらまし

大規模な密結合型マルチプロセッサ・システムを構築する上で、キャッシュ・コヒーレンス制御のオーバヘッドを軽減することは最重要課題の一つである。筆者らはデータの持つ意味や参照パターンなどの性質を利用することによりこの問題に対処することを考え、既にイベント対応型キャッシュ・コヒーレンス制御方式(ECCC)についての提案を行っている。本稿では、ECCCの新たな応用事例としてロック操作を取り上げその原理と動作について述べるとともに、発表済みの応用事例と併せてシミュレーションによる実験結果で示される基本性能についても論じる。

和文キーワード　イベント、キャッシュ・コヒーレンス、ロック操作、バリア同期、シミュレーション、性能評価

An Implementation of the Event Correspondent Cache Coherency Scheme and its Performance Analysis

Hideki SAITO†, Shin-ichiro MORI†, Hiroshi NAKASHIMA†, Shinji TOMITA†
Takashi TANAKA‡, David FRASER‡, and Kazuki JOE‡

†: Department of Information Science
Faculty of Engineering, Kyoto University
Yoshida-hon-machi, Sakyo-ku, Kyoto 606-01 Japan
E-mail: {saito, moris, nakashim, tomita}@kuis.kyoto-u.ac.jp

‡: Office of Computer Business, KUBOTA Corporation
ASTEM RI　17 Chudoji, Minami-machi, Shimogyo-ku, Kyoto 600 Japan

Abstract

It is essential to reduce the overhead of cache coherence, especially in large scale shared-memory multiprocessors. The authors have already proposed the Event Correspondent Cache Coherency Sheme(ECCC), which reduces the redundant cache coherence overhead, reflecting the characteristics of memory accesses. In this paper, we present another implementation of ECCC and analyze its basic performance based on simulation results. We also analyze the basic performance of our barrier synchronization method which we had presented before.

英文 key words　Event, Cache Coherence, Lock, Barrier Synchronization, Simulation, Performance

1 はじめに

プロセッサとメモリとの速度差は広がる傾向にあり、それを埋める役割を果たすキャッシュは、プロセッサの性能を十分に生かすために必要な要素の一つとなっている。大規模な密結合型マルチプロセッサ・システムにおいてもキャッシュを用いるものが一般的となりつつある [10] [3] [2] が、大規模なシステムでは、バス型のような小規模なシステムと比べて、一般にブロードキャストのコストが大きいと考えられ、そのことに起因するキャッシュ・コヒーレンス制御のオーバヘッドは大きな問題となっている。筆者らは不要なコヒーレンス動作をなくすことによってこの問題に対処することを考えており、既にイベント対応型キャッシュ・コヒーレンス制御方式 (ECCC) についての提案を行っている [8]。

本稿では、ロック操作に ECCC を応用する方法を述べ、その基本性能について報告するとともに、前回発表のバリア同期操作についてもその基本性能の報告を行う。

まず、第 2 章で ECCC の概略について説明を行い、第 3 章でその新たな応用事例として ECCC に基づいたロック操作の原理と動作について述べる。第 4 章ではバリア同期操作およびロック操作に ECCC を応用した例についてシミュレーションによる実験を行った結果を述べ、第 5 章で簡単なまとめを述べる。

2 イベント対応型キャッシュ・コヒーレンス制御方式の概略

従来のキャッシュでは、データの持つ意味やデータに対するアクセスのパターンとは無関係に、キャッシュ中のすべてのデータに対して同一のプロトコルに従ってコヒーレンス制御を行っていた。そのことに起因する不必要／不適切なコヒーレンス動作の発生は、プロセッサ—メモリ間ネットワークにおいて本質的には不要なトラフィックの発生として現われ、システムの性能に大きな影響を与える。特に大規模なシステムにおいては、こうしたオーバヘッドは致命的な問題となる。

ところが実際には、特有のアクセス・パターンや特殊な意味を持ったデータ等が多数存在し、それぞれのデータに対しては最適なコヒーレンス制御プロトコルというものが存在する。こうしたデータの属性をキャッシュのコヒーレンス制御に反映させ、不必要なコヒーレンス動作をなくすとともに、必要なコヒーレンス動作に対しては最適なプロトコルでそれを実現するものとして、筆者らはイベント対応型キャッシュ・コヒーレンス制御方式 (Event Correspondent Cache Coherency scheme: ECCC) を提案している [8]。

ECCC において、イベントとは、キャッシュあるいはその制御機構がコヒーレンス制御を開始する誘因となるものであり、データの属性によって区別される [1]。各イベントに対しては対応するコヒーレンス制御、すなわち、

1. コヒーレンス動作を行うか否か
2. 制御プロトコル（コヒーレンス動作を行う場合）

が定義される。

イベントの種類に応じた最適なプロトコルを事前に定義し、イベント発生時にそれに従った制御を行うことで、無駄のないキャッシュ・コヒーレンス制御が可能となる。

[1]属性の与え方などは排他的なものではなく、組み合わせて新たな属性として用いることも可能である。

3 ECCC のロック操作への応用

密結合型マルチプロセッサ・システム、特に大規模なものでは、プロセッサ間あるいはプロセス間の同期を効率良く行うことが非常に重要である。文献 [8] ではバリア同期操作に対して ECCC を応用した例についての報告を行った。本稿では排他制御のために用いられるロック操作に対して ECCC を応用した例について取り上げる。

ロック操作についての研究はかなり古くから行われており、ソフトウェアによるロック・アルゴリズム、ハードウェアによるロック支援機構など、様々な研究が既に報告されている [5] [7] [4]。本稿では ECCC を応用することによりロック操作の効率を高める手法を 2 つ提案する。1 つは一般のキャッシュを持つシステム、もう 1 つはディレクトリ方式のキャッシュを持つシステムを設計する場合に適用可能である。

3.1 排他制御

元来、ロック操作は一般に排他制御を必要とする資源にアクセスする場合に用いられるものであるが、密結合型マルチプロセッサ・システムを考える場合、メモリに対する排他制御を効率良く行うことはシステム設計における最重要課題の一つである。

クリティカル・セクション内における排他的なデータ操作は、以下のように大別できる。

- 大きなサイズのデータに対して任意の演算を行う。
- ごく小さなサイズのデータに対して簡単な演算を行う。

これらに対してはそれぞれ異なった排他制御方法を用いるのが妥当だと思われる。まず、大きなサイズのデータに対して演算を行う場合には、ロック操作を行い、データをキャッシングするのが効率良いと考えられる。次に、1 word 程度のごく小さなデータに対して不可分な Fetch&Operation のプリミティブとしてサポートされるような簡単な演算を行う場合には、そのプリミティブを用いることにより排他制御を実現することが可能であり、ロック操作の必要性はないと思われる。

3.2 ECCC のロック操作への応用

3.2.1 Prefetch と Eager Write-Back によるロック／アンロックの高速化

あるプロセッサあるいはプロセス（以下ではプロセッサとして記述し、特に区別しない。）が排他制御を要求する場合には他のプロセッサも同様の要求を出す可能性のあることが多いと考えられる。こうした要求が頻発し、さらに当該データの値が書き換えられる場合、当該データのキャッシュ・コピーは実質的にはロックを認められたプロセッサのみが保持していることとなり、データがアンロックされた後にロックを獲得して新たにクリティカル・セクションに入ったプロセッサはほぼ確実にキャッシュ・ミスを起こすこととなる。この場合、従来のコヒーレンス制御方式では、プリフェッチによるキャッシュ・ミス防止を効率良く行うことはできない。そこで筆者らは、ロック／アンロック操作がハードウェアにより認識可能で、かつその対象となるデータもハードウェアにより認識可能な場合に、ロック／アンロック操作とその対象となるデータに対するコヒーレンス制御に対してイベントの概念を導入し、そのイベントに対して最適な制御プロトコルをキャッシュ制御機構に実装することを提案する。

従来方式のWrite-Invalidate型キャッシュを用いる場合、クリティカル・セクションの前後では、

1. ロックを獲得
2. ロックを獲得したプロセッサがキャッシュにデータを要求し、キャッシュ・ミスヒット発生
3. キャッシュはメモリにデータを要求する。
   - プロセッサからの要求がLoadである場合
     (a) データがDirtyであればメモリは書き込みが行われたキャッシュにWrite-Backを要求し、Write-Backが終了すると要求したキャッシュにデータをCleanな状態で渡す。この時点で当該データのキャッシュ・コピーは2つ存在している。
     (b) データがCleanであればメモリは要求したキャッシュにデータを渡す。この時点で当該データのキャッシュ・コピーは複数存在している可能性がある。
   - プロセッサからの要求がStoreである場合
     (a) データがDirtyであればメモリは書き込みが行われたキャッシュにWrite-Back-Invalidateを要求し、Write-Backが終了すると要求したキャッシュにOwnershipとともにデータをDirtyな状態で渡す。
     (b) データがClean-Cachedであればメモリはキャッシュ・コピーを持っているキャッシュに対してInvalidateを要求し、全てのInvalidateが終了するのを待って要求したキャッシュにOwnershipとともにデータをDirtyな状態で渡す。
4. 最初のデータ要求がLoadであり、その後、当該データに対して最初のStoreが行われる場合
   (a) キャッシュにヒット
   (b) キャッシュはメモリにキャッシュ・コピーのInvalidateを要求
   (c) メモリは他の全てのキャッシュ・コピーに対してInvalidateを要求
   (d) 全てのInvalidateが終了するとメモリは当該ラインをDirtyにしてOwnershipをキャッシュに渡す。
   (e) Ownershipが戻りデータの書き込みが行われるとStore終了
5. （クリティカル・セクション内部の実行）
6. アンロック
7. 他のプロセッサがロックを獲得（以下、同様）

という手順となる[2]が、書き込みを行った場合にアンロックしたプロセッサがデータをDirtyな状態で持っている必要はなく、アンロック後のWrite-Back要求を待つことは、ロックを獲得したプロセッサにおける命令実行終了を無意味に遅らせるのみである。また、ロックを獲得したプロセッサのみが当該データに対するアクセスを行う場合、ロック獲得時に必要とされるべき最新のデータを得られる保証があればInvalidateの要求を出す必要性もない。

そこで、ロック獲得およびアンロックをそれぞれイベントとして考え、それに対応するコヒーレンス動作として強制プリフェッチとライト・バックをそれぞれ行うことにより無意味なレイテンシをなくし、余分なトラフィックをなくすことを提案する。より具体的に手順を示すと、

1. ロック獲得を示す値をプロセッサに送るとともに、当該ロックに対応するデータをキャッシュに送り強制的にフェッチさせる。このとき、ロック獲得を示す値はデータと同時に届くようにする。
2. プロセッサがロック獲得。この時点で既にデータはキャッシュに強制的にプリフェッチさせられており、必ずヒットする。
3. （クリティカル・セクション内部の実行）
4. プロセッサがアンロックを発行。
5. キャッシュはそれを認識してWrite-Backを開始する。このとき、Write-Backのデータがアンロック要求と同時に到着するようにする。
6. アンロック要求がメモリに到着すると、当該データはCleanの状態になっている。
7. 次にロックを与えるプロセッサ（クラスタ）が分かれば、当該データを送るとともにロック獲得を通知する。

アンロックの際に当該ラインをInvalidateするかどうか、また、Invalidateしない場合に、新たにロックを獲得したプロセッサが書き込みを行う時にとるべきコヒーレンス動作については、当該データの使われ方に依存し、それぞれに最適な動作が存在する。

こうすることにより、キャッシュ・ミスによるトラフィックの増加とメモリ・アクセス・レイテンシの増大を防ぐことができる。半強制的にキャッシングさせられるラインに対応するセット内の全てのウェイが埋まっており、これら全てがDirtyである場合には問題が生ずると思われるが、この問題に対しては、ライト・バック用のバッファを設けるかまたは新しいラインを放棄してキャッシュ・ミスを発生させることによって対処することが可能だと思われる。

3.2.2 ロック・リクエストをキューイングすることによる高速化

ロック獲得の方法については、Test&Setを用いる単純なアルゴリズムに対して同期変数をキャッシングすることによる高速化技法が提案されており、また、その他のロック・アルゴリズムなども提案されている[5]が、どれも多数のメモリ・アクセスを伴っており、一般にメモリ参照コストが大きいとされる大規模並列機に対しては十分とはいえない。また、Goodmanらの提案によるQueued On Lock Bit(QOLB)[7]はハードウェアによるロック・キューを必要とし、コストの面で問題があると思われる。そこで我々は、QOLBの考えを用い、ディレクトリ方式のコヒーレンス制御を行うキャッシュを持つシステムに対して、ディレクトリをロック・キューとして用いることを提案する。本提案はチェインド・ディレクトリ方式およびリミティッド・ディレクトリ方式にも適用可能であるが、ここではフルマップ・ディレクトリ方式のシステムを例にとり、以下により具体的な説明を行う。

1. 既にロックされているラインに対してロックを要求された場合、ディレクトリ中の要求したプロセッサ（クラスタ）に対応する存在ビットをセットすることにより、リクエストをキューイングする。
2. アンロックされる時、当該ラインのディレクトリにセットされているビットがあれば、そのうちの一つを選んでロック獲得を通知し、対応するビットをリセットする。
3. セットされたビットのなかから1 bitを選択するアルゴリズムとして筆者らが検討しているのは、アンロックを発行したプロセッサ（クラスタ）の番号に基づいたラウンドロビン方式である。

ここでは、アンロック操作が一つのイベントであり、それに対して、同一ラインの複数のキャッシュ・コピーのうち、ただ一つを選択してアップデートするというコヒーレンス動作を行っていることとなる。

これにより、ロック要求のFIFO処理を要求されないものについては、同じシステムでQOLBを実装することと比較して、より少ないハードウェア量でほぼ同等の性能を得ることが可能であると思われる。

スタンフォード大学のDASHにもQueue-Based Lockと呼ばれるディレクトリをロック・キューに用いる機構がある[4]が、両者はそれぞれ独自に考案されたものであり、DASHでは次にロックを与えるクラスタを、free

[2] イリノイ・プロトコル[1]を想定。ここではvalid-exclusiveとsharedを特に区別せずCleanとした。

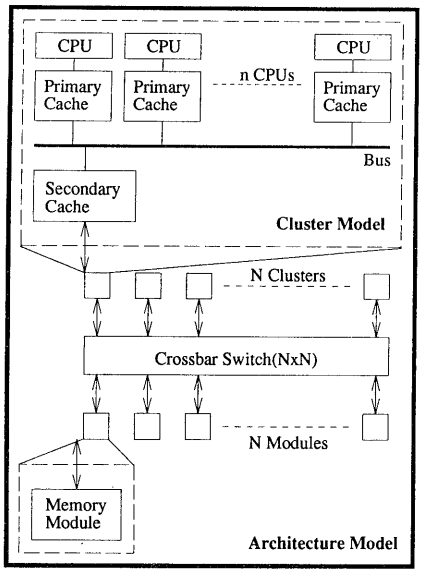

図 1: 性能評価に用いたアーキテクチャ

running hardware counter を用いてランダムに決定している点で、starvation free をアルゴリズムの上で保証している筆者らの提案とは異なっている。

4 応用事例の基本性能

ECCC の有効性を示すための第一歩として、その応用事例であるバリア同期操作とロック操作について図 1 に示されるアーキテクチャのモデルにおける基本的な性能を評価することとする。筆者らは、基本性能を調査するために並列ループ・シミュレータを作成し、シミュレーションによる実験を行い後述の結果を得た。

4.1 応用事例

4.1.1 バリア同期操作への応用 [8]

集中型バリア同期アルゴリズムを採用し、同期変数をキャッシングし同期待ちのスピンをキャッシュ上で行う。具体的には、同期に参加している各スレッドによる同期変数の更新時ごとのコヒーレンス動作を止め（一時的な inconsistency を許す）、当該同期変数に対する最後の更新時、即ち、「バリア同期成立時」にのみ write-update 型のコヒーレンス動作を行う。

4.1.2 ロック操作への応用

ロック要求をクラスタ単位でまとめ、ロックを獲得したクラスタ内でプロセッサ単位でのロックを実現する。以上は SCache で行われ、プロセッサ（プロセス）は意識する必要はない。これはフェッチしたデータをクラスタ内で有効に使うための最適化である。各クラスタからのロック要求をディレクトリ中の存在ビット・ベクタにキューイングする。ロック獲得時にプリフェッチを行う。

4.2 シミュレーションによる実験

4.2.1 並列ループ・シミュレータ

筆者らの作成した並列ループ・シミュレータは、図 1 に示されるアーキテクチャのモデルにおける並列ループの実行時オーバヘッドを計測するものである。クロスバ網は単方向のものをクラスタ→メモリ用とメモリ→クラスタ用の 2 セット用いた。シミュレータは以下のような特徴を持つ。

- Doall 型並列ループの各イタレーションの命令数をもとにした実行時オーバヘッドの計測が可能。
- リリース・コンシステンシ・モデル [3] を採用。
- 並列ループのスケジューリング法 [9] としてプレ・スケジューリング（スタティック・スケジューリング）とセルフ・スケジューリングを使用可能。
- システムのコンフィギュレーションをパラメータ・ファイルにより指定可能
- 並列ループの（同期変数以外の）グローバル・データに対するメモリ・リクエストやキャッシュ・ヒット率に関する平均的挙動をパラメータ・ファイルにより指定可能。
- バリア同期法として、バリア同期信号線を用いたハードウェア・バリア、キャッシュを使用しない集中型バリア同期アルゴリズム、ECCC を応用したキャッシュを使用する集中型バリア同期アルゴリズムの 3 手法を使用可能。

シミュレータ本体は、ドライバとシミュレーション・エンジンの 2 つに分かれる。シミュレーション・エンジンはさらに CPU、Primary Cache（以下、PCache と呼ぶ）、バス、Secondary Cache（以下、SCache と呼ぶ）、ネットワーク、メモリの各部に分かれる。ドライバはシミュレータのクロック 1ns 毎にシミュレーション・エンジンを呼びだし、エンジン内各部は、シミュレータのクロックがパラメータ・ファイルで指定されるサイクルと同調する毎に各部の 1 サイクル分のシミュレーションを実行する。

本シミュレータはバリア同期操作およびロック操作の性能を調査するためのものであるので、同期変数については忠実なシミュレーションを行っている。それ以外の一般のデータに対しては、確率的なモデル化を施し、スケジューリングとトラフィックおよびオーダリングのみについて考慮した。プロセッサ・プライベートなデータに対しては PCache に 100%ヒットするものとした。

一般のグローバル・データに対するシミュレーションがどのように行われているかを、自クラスタの SCache に存在しないラインに対してプロセッサが Store を行った場合を例にとって説明する。

プロセッサは残り命令数を保持する命令バッファをデクリメントすることによって命令をフェッチする。次に、この命令がメモリ・リクエストであるかどうか、メモリ・リクエストである場合には Load か Store かを確率的に判断する。メモリ・リクエストでない場合には 1 cycle で実行を終了し命令フェッチ可能状態となる。Load の場合にはプロセッサはデータ待ち状態となる。Store の場合には、キャッシュに受け付けられていない Store 命令の数を保持する Write Buffer をインクリメントし命令フェッチ可能状態となる。

PCache はバス側の入力バッファが空である場合、プロセッサからの要求にそれと同じサイクルで応答する。バスからもプロセッサからも要求が来ない場合、Write Buffer が空でなければそれに対する処理を行う。Write Buffer に対する処理を行う場合、まず Write Buffer をデクリメントし Store を受け付ける。それがローカル・データに対するものかグローバル・データに対するものか、グローバル・データに対するものであればヒット

するかしないか、ヒットした場合にはダーティであるかどうか、ミスヒットした場合にはリプレース対象のラインがダーティかどうかを確率的に判断する。以下、ミスヒットしたものとして説明を続ける。PCacheはバスに対するリクエストをバス側の出力バッファに出力する。リクエストにOwnership要求が含まれていれば、Ownership request counterをインクリメントする。このカウンタはOwnershipが得られるとデクリメントされる。Write BufferとOwnership request counterの両方が0になるとreleaseの実行を終了してよい。

バスはラウンドロビン方式の優先度にしたがってPCache, SCacheのリクエストを受け付ける。PCacheの出力バッファからStoreミスヒットのリクエストがバスに出ると、SCacheでダーティ・ヒットするかしないかをバス上で確率的に判断し、ダーティ・ヒットであれば一様乱数で決定されるクラスタ内のPCacheの入力バッファにWrite Backリクエストを入れる。これは、実際のバス型マルチプロセッサではスヌーピングとして行われている。PCacheからのリクエストはそのままSCacheの入力バッファへ入れられる。ただし、ダーティ・ヒットした場合にはその情報も付加される。

SCacheはバス側の入力バッファの処理とクロスバ網側の入力バッファの処理を交互に行う。バス側の入力バッファからPCache Storeミスのリクエストが来ると、それがダーティ・ヒットしていなければ、クリーン・ヒットかミスヒットか、ミスヒットの場合にはリプレース対象のラインがダーティかどうかを確率的に判断する。以下、ミスヒットしたものとしてさらに説明を続ける。SCacheはクロスバ網に対するリクエストをクロスバ網側の出力バッファに出力する。

SCache→メモリ用のクロスバ網は各SCacheの出力バッファの処理をラウンドロビン方式の優先度にしたがって行う。

メモリは入力バッファからのリクエストを処理する。StoreがSCacheでミスヒットした場合については、当該ラインがクリーンかダーティか、クリーンの場合にはキャッシングされているか、クリーンでキャッシングされている場合にはキャッシュ・コピーがいくつあるかを確率的に判断する。クリーンでかつキャッシングされていない場合にはすぐにOwnershipおよびデータ・リターンを返す（出力バッファに出力する）。クリーンでかつキャッシングされている場合には、キャッシングしているクラスタをキャッシュ・コピーの数だけ一様乱数で決定し、Invalidate要求を出力バッファに出力する。ダーティの場合には、ダーティ・コピーを持っているクラスタを一様乱数で決定しWrite Back要求を出力バッファに出力する。Inavlidate要求を出した場合には、対応するInvalidate終了が全て戻ってきた時にOwnershipと（必要ならば）データを出力バッファに出力する。Write Back要求についても同様である。

SCacheに対するInvalidate要求は必ずバスに出力され、バスに出力された時点でSCacheにおけるInvalidateが終了することとする。

シミュレーションに用いたパラメータを表1、表2、表3に示す。メモリ／キャッシュ・パラメータは全て正規分布に従う確率変数であり、表の値は平均値である。キャッシュ・コピー数の標準偏差は20とし、それ以外のものの標準偏差は平均値の1割程度とした。

4.2.2 シミュレーション

前述の並列ループ・シミュレータを用いて、プロセッサ数やループのイタレーション回数、イタレーション内の命令数を変化させた時のキャッシュ・コヒーレンス制御やイベント・オーダリングのオーバヘッドを含めたバリア同期操作およびロック操作のランタイム・オーバヘッドを計測した。行ったシミュレーションは以下の通りである。

表1: システム・パラメータ

プロセッサ数	（表4、表5参照）
プロセッサ動作周波数	32MHz
ロード・ディレイ	2cycle
ブランチ・ディレイ	3cycle
PCache動作周波数	32MHz
PCache Block Size	32B
バス動作周波数	32MHz
バス・バンド幅	256MB/s
SCache動作周波数	16MHz
SCache Block Size	1KB
メモリ動作周波数	16MHz
クロスバ網動作周波数	32MHz
1回線あたりの クロスバ網バンド幅	256MB/s

表2: プログラム・パラメータ

グローバル・メモリ・リクエスト率	7.0%
（メモリ・リクエスト中の）ストア率	30.0%

バリア同期操作 スケジューリング・アルゴリズムはプレ・スケジューリング。各プロセッサが終了する命令数が静的に決まる。この命令数の実行が終了すると、コンシステンシ制御のためにReleaseを発行しStoreの終了を待つ。全てのStoreに対するAckが得られると同期点に達したこととなり、バリア同期操作を実行する。

- **hwps** バリア同期信号線を用いたハードウェア・バリア。これをオーバヘッド0の基準とする。バリア同期に参加する全てのプロセッサが同期点に達すると、電気的信号遅延レベルの時間で同期成立を知ることが可能。
- **eps** ECCCを応用したキャッシュを用いた集中型バリア。
- **cps** キャッシュを使用しない集中型バリア。

ロック操作 バリア同期操作はハードウェア・バリア、スケジューリング・アルゴリズムはスタティック・スケジューリング。スケジューリングを行う際にロック操作を行うことにより、ロック操作のオーバヘッド計測に用いる。

- **hwsss** Test&Set命令を用いたいわゆるシンプル・ロック
- **hwssq0** ECCCを応用したロック（プリフェッチなし、ロック時データ転送1KB）
- **hwssq1** ECCCを応用したロック（プリフェッチなし、ロック時データ転送1 word）
- **hwsse** ECCCを応用したロック（プリフェッチあり、ロック時データ転送1 word）

筆者らの提案するキューイング・ロック、複数ブロックサイズ・サポートおよびプリフェッチの効果をそれぞれ調べるため、ECCCを応用したロックについて3通りの実験を行った。この場合、使用するデータが1 wordのみであることから、ロック時データ転送を1 wordのみとするものを用いた。

4.3 シミュレーション結果

4.3.1 バリア同期操作

シミュレーションに用いた総プロセッサ数とクラスタ内プロセッサ数およびクラスタ数の関係を表4に示す。クラスタ内プロセッサ数については、商用小規模マル

表 3: メモリ／キャッシュ・パラメータ

PCache ヒット率	80.0%
PCache ダーティ率（ストア・ヒット時）	75.0%
SCache ヒット率	95.0%
SCache ダーティ率（ヒット時）	75.0%
メモリ・ダーティ率	10.0%
キャッシュ・コピー存在率	80.0%
キャッシュ・コピー数	5

表 4: 実験のパラメータに用いたプロセッサ数

総プロセッサ数	クラスタ内プロセッサ数	クラスタ数
32	4	8
64	4	16
128	4	32
256	4	64
512	4	128
1024	8	128

チプロセッサ機の多くがプロセッサ数 4 であることを考慮している。

バリア同期法およびプロセッサ数と台数効果との関係を図 2、図 3、図 4、図 5に示す。イタレーション内の命令数は 100 命令固定とし、イタレーション回数を 1024,4096,8192,16384 と変化させた。

なお、台数効果は、命令数とキャッシュ・ヒット率およびキャッシュ・ミス・コストから計算によって求めたシングル・プロセッサによる実行時間を、シミュレーションによって計測された並列実行時間で割った値である。シングル・プロセッサにおけるキャッシュヒット率などは並列のものと同じ値を用いた。PCache ミス・コストは 7 プロセッサ・サイクル、SCache ミス・コストは 140 プロセッサ・サイクル [3] と仮定した。

並列ループが必要とする実行時間（ハードウェア・バリアによる実行時間）をもとにバリア同期のオーバヘッドを考えると、筆者らの提案する ECCC を応用したバリア同期法は、集中型バリア同期アルゴリズムの約 1/4 から 1/3 のオーバヘッドであることがわかる。例えば 512 プロセッサ、1024 イタレーションのものについては、筆者らの提案するバリア同期法のオーバヘッドが並列ループが必要とする実行時間の約 15%であるのに対し、集中型バリア同期アルゴリズムでは約 62%となっている。これは主に、同期のためのトラフィック発生量の差によるものであると思われる。

各グラフともに、1024 プロセッサ時に性能が伸びていないことがわかるが、これは通常のデータ・アクセスにおいて既にバスが飽和していることによると思われる。

4.3.2 ロック操作

シミュレーションに用いた総プロセッサ数とクラスタ内プロセッサ数およびクラスタ数の関係を表 5に示す。

ロック法およびプロセッサ数と台数効果との関係を図 6、図 7、図 8に示す。イタレーション回数は 1024 に固定し、イタレーション内命令数を 30,100,300 と変化させた。図 6のスケールが他の 2 つと異なることに注意されたい。

[3] 1KB のデータがネットワークを流れるのに最低必要なサイクル数が 1KB/64bit=128network cycles であることを考慮した。

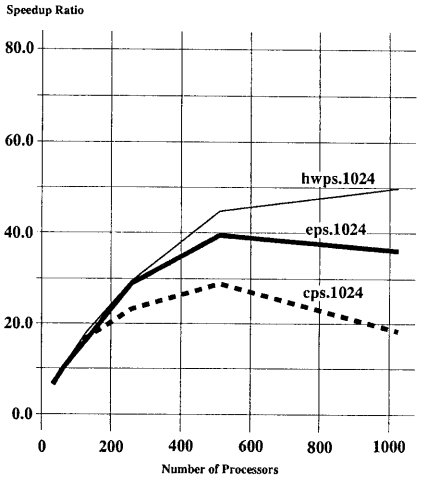

図 2: プロセッサ数と台数効果の関係（100 命令、1024 Iterations）

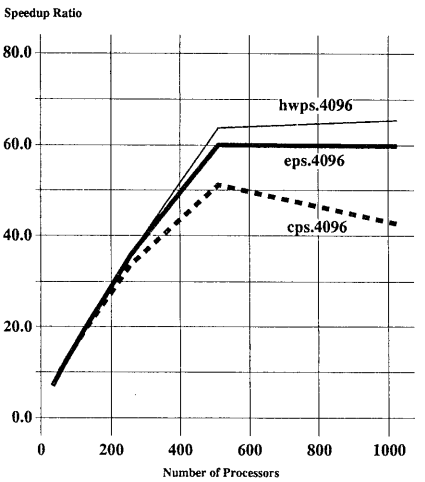

図 3: プロセッサ数と台数効果の関係（100 命令、4096 Iterations）

競合が起こりやすい条件での実験であり、提案した手法はどれも高い効果を発揮したことがわかる。ブロック・サイズ変更の効果が大きいことから、この条件では実行時間がクロスバ網の速度に依存しているということが言えるのではないだろうか。

また、クラスタ内プロセッサ数が 6 である 24 プロセッサのとき、台数効果の挙動が他と異なっているのは、バスが飽和しかかっていることが理由として考えられる。プリフェッチなしで 1KB のデータを転送するモデルにおいて 24 プロセッサの時に性能が向上しているのは、ロックのために SCache にキャッシングされたデータがクラスタ内で有効利用されていることを示していると思われる。

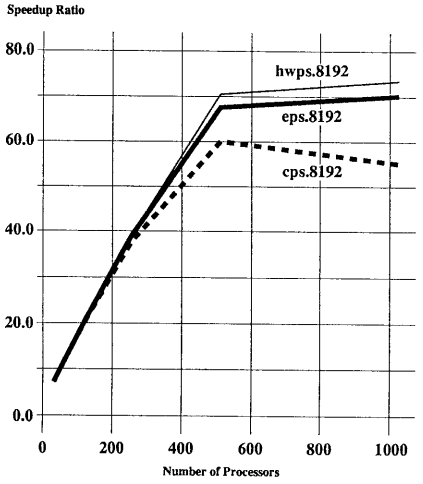

図 4: プロセッサ数と台数効果の関係（100 命令、8192 Iterations）

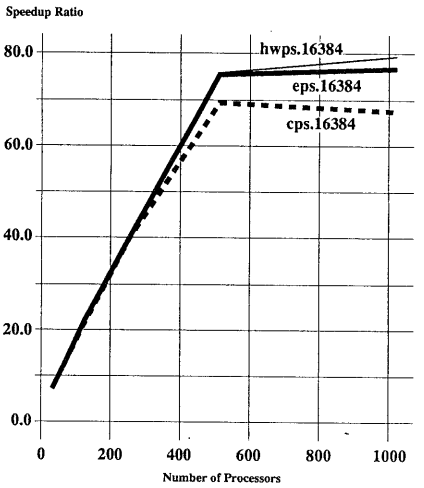

図 5: プロセッサ数と台数効果の関係（100 命令、16384 Iterations）

5 おわりに

従来行われているロック操作に対して問題を提起し、ロック操作に ECCC を応用することについて提案を行った。また、既に発表済みの ECCC を応用したバリア同期操作と併せて、シミュレーション結果に基づく基本性能を示した。

今後は、システム構成やプログラム、キャッシュ、メモリなどのパラメータを様々に変えた実験を行う予定である。他のバリア同期あるいはロック・アルゴリズムとの比較検討を行うことが今後の大きな課題である。

また、並列ループ・スケジューリング法として、ガイデッド・セルフ・スケジューリング [6] や Fetch&Decrement を用いたセルフ・スケジューリングも使用可能にしたい。

表 5: 実験のパラメータに用いたプロセッサ数

総プロセッサ数	クラスタ内プロセッサ数	クラスタ数
4	2	2
8	4	2
16	4	4
24	6	4
32	4	8
64	4	16

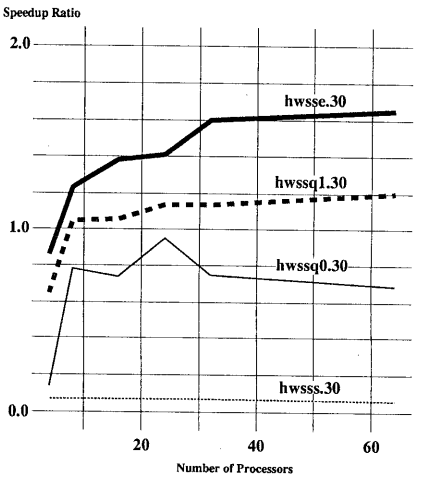

図 6: プロセッサ数と台数効果の関係（30 命令、1024 Iterations）

謝辞

本研究を進めるにあたり、御指導、御討論頂いた名古屋大学 阿草清滋教授、（株）クボタ 山口宗之部長、（株）クボタ ASURA プロジェクト・チーム の諸氏に感謝致します。また、文献 [4] の入手に御協力いただいたスタンフォード大学 Anoop Gupta 教授、シミュレータの作成について御教授をお願いしたイリノイ大学 Carl Beckmann 氏に感謝致します。最後に、日頃ご討論頂く京都大学 富田研究室 の諸氏に感謝致します。

参考文献

[1] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298. November 1986.

[2] David R. Cheriton, Hendrik A. Goosen, and Patricck D. Boyle. Paradigm: A highly scalable shared-memory multicomputer architecture. *IEEE Computer*, pages 33–46, 1991.

[3] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, pages 63–79, 1992.

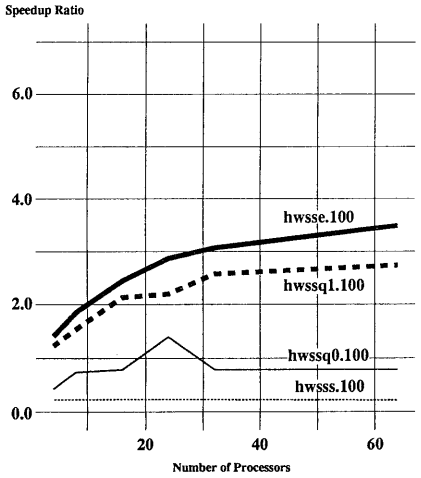

図 7: プロセッサ数と台数効果の関係（100 命令、1024 Iterations）.

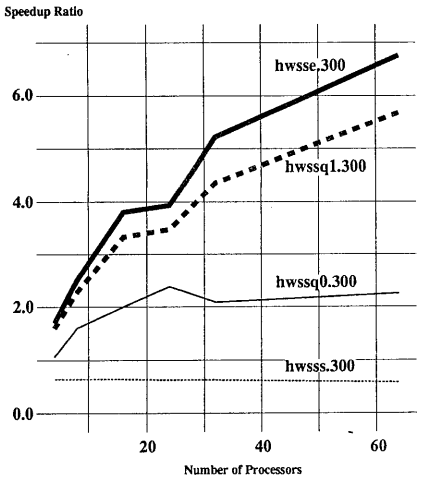

図 8: プロセッサ数と台数効果の関係（300 命令、1024 Iterations）

[4] Daniel E. Lenoski. The design and analysis of DASH: A scalable directory-based multiprocessor. Technical Report CSL-TR-92-507, Computer Systems Laboratory, Stanford University, February 1992.

[5] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[6] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[7] Philip J. Woest and James R. Goodman. An analyssis of synchronization mechanisms in shared-memory multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 152–165, April 1991.

[8] 齋藤 秀樹, 森 眞一郎, 富田 眞治, 田中 高士, David Fraser, and 城 和貴. イベント対応型キャッシュ・コヒーレンス制御方式とそのバリア同期への応用. Technical Report 92-ARC-95-2, 情報処理学会, 1992.

[9] 笠原 博徳. 並列処理技術. コロナ社, 1991.

[10] 森 眞一郎, 齋藤 秀樹, 五島 正裕, 富田 眞治, 田中 高士, David Fraser, 城 和貴, and 新田 博之. 分散共有メモリ型マルチプロセッサ「阿修羅」の概要. Technical Report 92-ARC-94-6, 情報処理学会, 1992.