Vol. 42 No. SIG 9(HPS 3) 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. 2001

共有メモリ・マルチプロセッサの分散シミュレーションのための参照フィルタ方式

今福 茂[†,☆] 大野和彦[†] 中島 浩[†]

本論文では，我々が開発中の共有メモリ・マルチプロセッサのための実行駆動型分散シミュレータ*Shaman*において，そのフロントエンドが生成するメモリ参照履歴を削減する方式を述べる．Shamanはフロントエンドとバックエンドから構成され，PCクラスタで実行される．フロントエンドではシミュレーションのワークロード・プログラムを，ソフトウェア分散共有メモリの技法を用いて並列実行し，シミュレーション対象システムのコヒーレント・キャッシュを部分的にシミュレートしてメモリ参照履歴を生成する．履歴はバックエンドに送られ，キャッシュを含む対象メモリ・システムの挙動がシミュレートされる．履歴削減の基本的なアイデアは，フロントエンドのキャッシュをフィルタとして用い，このフィルタ・キャッシュをミスした参照のみをバックエンドに送ることである．本論文ではキャッシュ/メモリのブロックに対するdata-race-free（DRF）の概念を導入し，DRFのブロックに関してはフィルタが正しく動作することを証明するとともに，DRFではないブロックの検出方法も示す．またSPLASH-2中の2つのカーネルを用いた評価により，最高で99.6%の参照が除去され，残りの参照の中で対象システムのキャッシュにヒットするものは1.4%以下であることも示す．

Reference Filtering for Distributed Simulation of Shared Memory Multiprocessors

Shigeru Imafuku,[†,☆] Kazuhiko Ohno[†] and Hiroshi Nakashima[†]

This paper proposes a method to reduce the amount of the memory references generated by the front-end of our distributed execution-driven simulator for shared memory multiprocessors named *Shaman*. The simulator consists of the front-end to execute programs in parallel and the back-end, driven by the memory references from the front-end, to simulate the behavior of the memory system of a target multiprocessor. For high performance simulation, the front-end runs on a PC cluster using software DSM technique and partially simulates the coherent cache of the target system. The key idea of the reference reduction is to use the caches in the front-end as a filter of the references. We prove that the filtering for a memory block is safe if it is accessed in *data-race-free* manner as the whole. We also show a method to detect *racing* blocks to inactivate the filtering. The preliminary experiment with SPLASH-2 kernels shows up to 99.6% of references are filtered out and redundant references are less than 1.4%.

1. はじめに

マルチプロセッサ・アーキテクチャの研究・開発において，新しいアイデアの有効性を検証するために，シミュレータは必要不可欠なツールである．このため，アーキテクチャ・レベル（あるいは機械命令レベル）のシミュレータが数多く提案されており，その多くではシミュレーションの高速化，換言すれば実機に比べた速度低下を小さくするための工夫がなされている．たとえば我々が研究対象としている共有メモリ・マルチプロセッサのための実行駆動型シミュレータの中には，実行時間が実機の10〜100倍程度の「高速」なものがいくつかある．

しかしこのような高速シミュレータであっても，実用的な観点からは必ずしも十分な性能であるとはいえない．すなわち，ほとんどのシミュレータが唱っている小さな実行時間比は，対象システムの単一プロセッサとの比率であり，システム全体の性能との比率ではないからである．たとえば実行時間比が実機の50倍であるシミュレータを用いて，64プロセッサからなる対象システムで1分を要するプログラムを実行したとすると，シミュレーション時間は3,200分となり2日以上を要することになる．

† 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，セイコーエプソン株式会社
Presently with Seiko Epson Corp.

したがって，従来のシミュレータとは異なるアプローチによってさらに高速化することが求められており，その有力な手段として並列あるいは分散処理があげられる．すなわち対象システムの1つあるいは複数のプロセッサを，並列/分散シミュレータの1つのノードに割り当て，対象システムに当然内在している並列性を利用して実行することは，きわめて自然な高速化手法であるといえる．しかし，この並列化/分散化を単純に行うと，きわめて多数のノード間通信が発生し，性能向上はほとんど望めない．すなわち，共有メモリ・マルチプロセッサではすべてのメモリ参照が潜在的なプロセッサ間通信であるため，これらの正しい順序付けや時刻管理のためにシミュレータ・ノード間で同期をとろうとすると，禁止的に大きなオーバヘッドが生じることは明らかである．

そこで我々が開発中の分散シミュレータ *Shaman*では，このメモリ参照の問題を以下の3点に着目して解決することとした[10),11)]．

(1) 正しく同期が行われる決定的なワークロード・プログラムに対しては，ソフトウェア分散共有メモリの技法を用いたシミュレーションにより，個々のプロセッサに関する限り正しく順序付けられたメモリ参照履歴を生成することができる．

(2) 対象システムのコヒーレント・キャッシュを部分的にシミュレートすることにより，対象キャッシュで必ずヒットするようなメモリ参照を履歴から除去するフィルタ操作を行うことができる．

(3) フィルタを通り抜けた履歴には，対象システムにおけるすべてのキャッシュ・ミスが含まれており，これらによってメモリ・システムの動作タイミングの正確な，あるいは十分な精度を保った，再現が可能である．

本論文の眼目は，上記(2)のフィルタ操作の提案とその正当性の証明である．すなわち従来は単一プロセッサ上で動作するシミュレータでのみ行われていたフィルタ操作を，分散シミュレータにおいても正しくかつ効率的に実現できることを示したことが，本論文による重要な貢献である．

以下本論文では，上記のメモリ参照履歴のフィルタ操作の提案，正当性の証明，および実験による評価について，次のような順序で述べる．まず2章では本研究の背景となる関連研究を概観し，次に3章でShamanとその分散実行技法であるソフトウェア分散共有メモリについて述べる．続く4章が本論文の核心であり，参照フィルタとその正当性について詳細に述べる．また5章では参照フィルタの有効性を評価するためにSPLASH-2中のカーネルを用いて行った実験結果を示し，6章で結論と今後の課題を述べる．

2. 関連研究

UhligとMudgeによる詳細なサーベイ[20)]に示されているように，共有メモリ・マルチプロセッサのための実行駆動型シミュレータは，これまでに数多く提案・開発されている．これらの多くは単一プロセッサ上で動作し，ワークロード・プログラムを実行してメモリ参照履歴を生成するフロントエンドと，履歴に基づいて対象マシンのメモリシステムをシミュレートするバックエンドから構成されている．フロントエンドでの実行方式は，対象プロセッサの機械命令をエミュレートするもの[7),15),21)]と，instrumentationを施したコードを直接実行するもの[4),8),18)]とに大別される．いずれの方式においても，高速なシミュレータでは10〜100倍の対実機実行時間比が達成されている[18),20)]．

フロントエンドとバックエンドは，1つのプログラムの構成要素としてリンクされるため，両者の間でのデータのやりとりはきわめて高速に行うことができる．したがってトレース駆動型とは異なり，参照履歴の授受にあたってファイル容量や入出力オーバヘッドの問題を顧慮する必要はほとんどない．そのため多くの場合，フロントエンドがすべてのメモリ参照履歴をバックエンドに渡すという単純な方法が用いられており，フィルタ操作などの履歴圧縮は行われていない．

一方，共有メモリ・マルチプロセッサには明らかに並列性が内在しているにもかかわらず，シミュレータの並列化や分散化を行った成功例はきわめて少ない．この主な理由は，対象プロセッサの論理構造がきわめて複雑であるため，分散論理シミュレーションで用いられるChandy-Misra法[6)]やTimeWarp法[12)]などの非同期的な手法の適用が困難なことである．実際，並列化の数少ない成功例の1つであるWisconsin Wind Tunnel（WWT）[17)]では同期的手法[3)]が用いられている．一般に同期的手法はシミュレータ・ノード間で頻繁に同期操作を行うため，オーバヘッドが大きく不利とされているが，WWTではホストであるCM-5が持つ高速なバリア機構を利用してこの問題を解決している．逆にいえば，WWTの方式はCM-5での実装を前提としたものであり，たとえばLANで結合されたPCクラスタのような安価な並列実行環境では，頻繁なバリア操作は性能に対して致命的なダメージを与える．

またWWTでは，コヒーレント・キャッシュを含む共有メモリの挙動を，Liのソフトウェア分散共有メモリ（S/W-DSM）[14)]に類似した技法を用いてシミュレート

している．すなわち，あるプロセッサにより書き込みが行われると，そのイベントはただちに書き込まれたブロックを共有するキャッシュを担当するシミュレータ・ノードに伝えられる．この方法の利点は，シミュレータ・ノード間で伝達されるメモリ参照に関するイベントの順序を，対象システムでの順序と一致させることが比較的容易なことである．しかし S/W-DSM の実現方法としては，lazy release consistency（LRC）[13] のように書き込みの伝達を後続する同期操作まで遅延させる方式が明らかに有利であり，特に PC クラスタのような環境では両者の性能差はきわめて大きい．

前述のように実行駆動型では参照履歴の削減はほとんど行われていないが，トレース駆動型では様々な削減方式が適用されている[20]．その 1 つとして，キャッシュを参照フィルタとして用いる方法は古くから知られており，たとえば Puzuk による文献 16) では，ある容量のダイレクトマップ・キャッシュをミスした参照履歴だけを，容量や連想度がより大きな対象キャッシュの解析のために用いる方法が提案されている．またこの文献では，解析に必要な参照が対象キャッシュをミスするようなものである場合，それらは必ずフィルタ・キャッシュをミスすること，すなわちキャッシュによるフィルタ操作の安全性も証明されている．この考え方は Wang らによって拡張され[22]，ライトバック・キャッシュにおけるメモリへの書き戻し事象も，フィルタ後の参照履歴を用いて正確に再現できることが知られている．

3. 分散シミュレータ Shaman

我々は PC クラスタのような安価な環境を用いた高速シミュレーションを目的とし，共有メモリ・マルチプロセッサのための分散シミュレータ Shaman を開発している．本章では，本論文での主題である参照フィルタ操作の適用対象例である Shaman と，その分散実行技法であるソフトウェア分散共有メモリについて述べる．

3.1 対象システムとワークロード

Shaman が対象とするシステムは，一般に**図 1** のような構成を持つ集中型あるいは分散型の共有メモリ・マルチプロセッサである．対象システムの各プロセッサ（P）はコヒーレント・キャッシュ（C）を持ち，プロセッサ/キャッシュとメモリは任意の構成を持つネットワークで結合される．Shaman でシミュレート可能なプロセッサは，現時点では単一パイプライン構成で SPARC Version 8 命令セット・アーキテクチャ☆に基づくものに限定されるが，原理的にはエミュレーション・モジュールの置き換えにより他の実行方式や命令セットにも対応可能である．

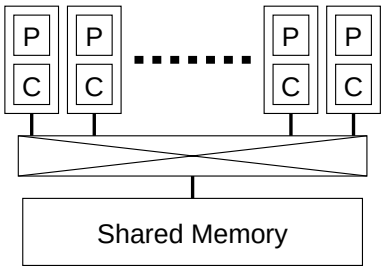

図 1 対象システム
Fig. 1 Target system.

キャッシュは，ライトバック型かつ MSI（Modified, Shared, Invalid）の 3 状態を含むような無効化型のコヒーレンス・プロトコル[19]に基づくものであることが望ましい．すなわちライトスルー型や更新型プロトコルであってもシミュレート可能ではあるが，後述する参照フィルタ操作の効果はきわめて小さなものとなる．他の構成パラメータ，たとえば容量，連想度，ブロック（ライン）サイズ，統合型か命令/データ分離型か，単一であるか階層型であるか，などは対象システムに応じて設定することができる．

Shaman が実行するワークロード・プログラムは，Solaris または POSIX のスレッド・ライブラリを用いて書かれたマルチスレッド・プログラムであり，シンボル・デバッグ情報が付加された Solaris のバイナリ実行形式として与えられる．ライブラリ中の同期プリミティブや基本入出力関数は，シンボル・デバッグ情報を参照することにより検出され，Shaman の内部で直接実行される．またバリア操作など，独自の同期プリミティブも用意されており，効率的な分散実行のためにやはり内部的に実行される．

後述するように Shaman の分散実行方式は，Keleher らによる lazy release consistency（LRC）[13]の機構に基づいている．したがってワークロード・プログラムは以下の定義に基づく data-race-free（DRF）[1),2)] の性質を有していなければならない．

定義 1 ある実行におけるメモリ参照 a と a' に関する以下の関係 $\overset{\text{po}}{\rightarrow}$ と $\overset{\text{so1}}{\rightarrow}$ の非反射的推移閉包を $\overset{\text{hb1}}{\rightarrow}$ とし，$a_1 \overset{\text{hb1}}{\rightarrow} a_2$ であるとき，またそのときに限り，a_1 は *happens-before-1* 関係において a_2 に先行するという．

- あるプロセスにおいて a がプログラム順で a' に先行するとき，またそのときに限り，$a \overset{\text{po}}{\rightarrow} a'$ である．

☆ 近い将来 Version 9 とする予定．

- a が *release*操作 (S_R)，a' が *acquire*操作 (S_A) であり，a と a' が対をなすとき，またそのときに限り，$a \overset{so1}{\rightarrow} a'$ である．

通常，S_Aと S_Rはそれぞれロック獲得とロック解放であり，たとえば `mutex_lock()` や `mutex_unlock()` として Shaman が検出することができる．また a_l がロック獲得（S_A），a_u がロック解放（S_R）であって，a_u が解放した相互排除オブジェクトを a_l が獲得したとき，a_u と a_l は対をなす．S_Aと S_Rはその他の同期操作にも付随するが，そのような操作は Shaman が検出可能な同期プリミティブ（あるいはその一部）でなければならない．

定義 2 ある実行における *data race*とは，2 つの競合するメモリ参照，すなわち少なくとも一方が書き込みであるような同じアドレスに対する参照の対が，happens-before-1 関係によって順序付けられないことをいう．ある実行中に data race が存在しないとき，またそのときに限り，その実行は *DRF*であるという．あるプログラムのどのような sequential consistent な実行も DRF であるとき，またそのときに限り，そのプログラムは DRF であるという．

さらに，意味のあるシミュレーションを行うために，ワークロード・プログラムは以下の定義により決定的でなければならない．

定義 3 あるプログラムの特定の数のプロセッサを用いた実行について，同期プリミティブに含まれないすべてのメモリ参照が実行によらず同じ結果をもたらすとき，またそのときに限り，そのプログラムは決定的であるという．

なおスピン・ロックのような同期プリミティブ中のメモリ参照については，非決定性が許容されることに注意されたい．

有用な並列プログラムの大部分は DRF であることが知られており，またほとんどの SPMD プログラムをはじめ多くのプログラムは決定的である．したがって，これらの制限を加えても，Shaman は十分に汎用性を持つといえる．

3.2 システム構成

対象システムの Shaman へのマッピングは，**図 2** に示すように行われる．Shaman は複数ノードからなるフロントエンドと，単一のバックエンド・ノード☆からなる．フロントエンドには対象システムのプロセッサが，またバックエンドにはキャッシュ，ネットワーク，およびメモリがマッピングされる．ただしバックエンドではメモリシステムの動作タイミングなどの物理的挙動のみをシミュレートし，共有メモリを介したデータ送受は 3.3 節で述べる LRC による S/W-DSM の機構を用いてフロントエンドがシミュレートする．また後述のように，フロントエンドではキャッシュの部分的シミュレーションも行う．

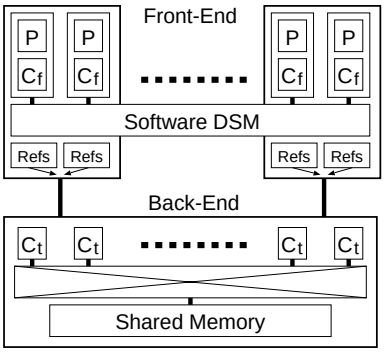

図 2 Shaman の構成
Fig. 2 Configuration of Shaman.

個々のフロントエンド・ノードは PC あるいはワークステーションであり，対象システムのプロセッサを 1 つあるいは複数担当し，メモリ参照履歴を生成する．なお対象システムのプロセッサ，あるいはフロントエンド・ノードでの対応する実行モジュールを，以後単にプロセッサという．対象システムで並列実行されるワークロード・プログラムは，Shaman のフロントエンドでも並列実行されるため，効率良くシミュレーションを行うことができる．またフロントエンド・ノードでのコンテキスト（プロセッサ）の切替えは，同期プリミティブの実行時にのみ行われるので，スケジューリングのオーバヘッドも小さい．

フロントエンドが生成した参照履歴はバックエンドに送られるが，すべての参照履歴を送信するときわめて大きな通信オーバヘッドが生じる．たとえば，対象システムを 64 プロセッサ構成，各プロセッサの性能を 1 GIPS，メモリ参照命令の出現頻度を 1/4，履歴の大きさを 1 つの参照あたり 4 B，ノード間を結合する LAN のスループットを 10 MB/sec と仮定すると，LAN での転送時間だけで実機との実行時間比が 6,400 倍となってしまう．

そこで個々のフロントエンド・ノードは，プロセッサごとにフィルタ・キャッシュ（C_f）を持ち，このキャッシュにミスした参照のみをバックエンドに送ることによって履歴送信量を削減する．このほか，スピン・ロックのように非決定性を持つ同期プリミティブは，スピ

☆ 原理的にはバックエンドの並列化も可能であり，将来の研究課題の 1 つである．

ン回数など実際の挙動を再現するためにプリミティブ単位でバックエンドに送られる．また送信される参照やプリミティブには，個々のフロントエンド・ノードに局所的なタイムスタンプが付加される．

バックエンドでは対象システムのメモリやネットワークだけでなく，対象システムのキャッシュと同じ構成の対象キャッシュ（$\mathsf{C_t}$）のシミュレーションも行う．すなわちフロントエンドから送られた参照に対し，対象キャッシュにより再度ヒット/ミスの判定を行い，真のミスだけが抽出されてメモリやネットワークのシミュレーションに用いられる．4 章で述べるように，対象キャッシュのミス（共有ブロックの書き込みを含む）を引き起こす可能性があるすべての参照は，必ずフィルタ・キャッシュを通過することが保証されるので，バックエンドはシミュレーションに必要なすべての参照を入手することができる．

対象キャッシュをミスした参照は，メモリやネットワークの挙動に応じて再スケジュールされ，局所時刻が大域時刻に変換される．この時刻変換は，blocking load/blocking cache のようにプロセッサがキャッシュ・ミスによりストールするものであれば，正確に行うことができる．一方ロード/ストアバッファなどメモリアクセス遅延を隠蔽する機構を有する場合には，高精度の時刻変換を行うために，フロントエンドでの部分的シミュレーションや遅延見積りによる局所的なスケジューリングを行う必要がある．

3.3 LRC によるソフトウェア分散共有メモリ

前述のように Shaman の S/W-DSM の管理機構は，Keleher らによって提案された LRC をベースとしている．これまでに提案されている様々な DSM 機構の中から LRC を選択したのは，以下の 3 つの理由による．第 1 の理由は性能面での優位性であり，PC クラスタのような通信コストが大きな環境では，書き込みの通知を即座に行わず，同期操作まで遅延させる方式が有利であることはよく知られている．

第 2 の理由は，コヒーレンス管理の単位であるページに対して複数のプロセッサが非同期的に書き込む write-write false sharing に対する性能的耐性が高いことである．Shaman の DSM はハードウェアによる共有メモリをシミュレートすることが目的であるため，一般の S/W-DSM のようにワークロードがページ単位コヒーレンス管理にチューニングされていることを期待できない．したがって，あるページへ書き込みを行うプロセッサが単一であるか否かをプログラマの指示により定めるようなプロトコル（たとえば Munin[5]）を採用することはできない．

第 3 の，そして最も重要な理由は，ある書き込みがどのプロセッサによっていつ行われたかを，その結果を読み出すプロセッサが完全に知ることができるという点である．この性質は 4 章で述べる参照フィルタ操作の効率や正当性の維持に不可欠であり，状況に応じて書き込みが生じたページ全体が送受されるようなプロトコル（たとえば AURC[9]）を採用することはできない．

以下，LRC の機構について概説するが，DRF プログラムに対する動作保証，実装，性能などについては文献 13) を参照されたい．

LRC では，ページと呼ばれる比較的大きなメモリ領域を単位として，コヒーレンスの制御を行う．すなわち，あるプロセッサによる参照 m によりページの一部が更新された場合，同じページに対して $m \overset{\mathrm{hb1}}{\rightarrow} m'$ なる参照 m' を行うすべてのプロセッサに，ページの更新情報が伝えられる．

たとえば，**図 3** に示すプロセッサ A，B，C による実行断片において，ページ π に含まれるアドレス x と y に関する読み出し（R）および書き込み（W）は，以下のように $\overset{\mathrm{hb1}}{\rightarrow}$ による順序付けを行うことができる．

$$\begin{aligned}&\mathrm{W}^A(x)\overset{\mathrm{hb1}}{\rightarrow}\mathrm{R}^B(x)\overset{\mathrm{hb1}}{\rightarrow}\mathrm{R}^C(x)\overset{\mathrm{hb1}}{\rightarrow}\mathrm{W}^C(x)\overset{\mathrm{hb1}}{\rightarrow}\mathrm{R}^A(x)\\ &\quad\Leftarrow \mathrm{W}^A(x)\overset{\mathrm{po}}{\rightarrow}\mathrm{S}^A_{\mathrm{R}}\overset{\mathrm{so1}}{\rightarrow}\mathrm{S}^B_{\mathrm{A}}\overset{\mathrm{po}}{\rightarrow}\mathrm{R}^B(x)\overset{\mathrm{po}}{\rightarrow}\mathrm{S}^B_{\mathrm{R}}\overset{\mathrm{so1}}{\rightarrow}\\ &\qquad\mathrm{S}^C_{\mathrm{A}}\overset{\mathrm{po}}{\rightarrow}\mathrm{R}^C(x)\overset{\mathrm{po}}{\rightarrow}\mathrm{W}^C(x)\overset{\mathrm{po}}{\rightarrow}\mathrm{S}^C_{\mathrm{R}}\overset{\mathrm{so1}}{\rightarrow}\\ &\qquad\mathrm{S}^A_{\mathrm{A}}\overset{\mathrm{po}}{\rightarrow}\mathrm{R}^A(x)\\ &\mathrm{W}^B(y)\overset{\mathrm{hb1}}{\rightarrow}\mathrm{R}^C(y)\overset{\mathrm{hb1}}{\rightarrow}\mathrm{R}^A(y)\\ &\quad\Leftarrow \mathrm{W}^B(y)\overset{\mathrm{po}}{\rightarrow}\mathrm{S}^B_{\mathrm{R}}\overset{\mathrm{so1}}{\rightarrow}\mathrm{S}^C_{\mathrm{A}}\overset{\mathrm{po}}{\rightarrow}\mathrm{R}^C(y)\overset{\mathrm{po}}{\rightarrow}\mathrm{S}^C_{\mathrm{R}}\overset{\mathrm{so1}}{\rightarrow}\\ &\qquad\mathrm{S}^A_{\mathrm{A}}\overset{\mathrm{po}}{\rightarrow}\mathrm{R}^A(y)\end{aligned}$$

したがって同図の実行断片は DRF であり，個々の読み出しについて以下の結果が得られることを保証しなければならない．

- $\mathrm{R}^B(x)$ と $\mathrm{R}^C(x)$ が得る値は $\mathrm{W}^A(x)$ が書き込んだものであること．
- $\mathrm{R}^A(x)$ が得る値は $\mathrm{W}^C(x)$ が書き込んだものであること．
- $\mathrm{R}^C(y)$ と $\mathrm{R}^A(y)$ が得る値は $\mathrm{W}^B(y)$ が書き込んだものであること．

このため，$\mathrm{S_R}$を行ったプロセッサは対応する $\mathrm{S_A}$を行うプロセッサに対して，以下の方法によりページ π の更新情報を伝える．

まず各プロセッサは，$\mathrm{S_A}$または $\mathrm{S_R}$を実行すると，プロセッサに固有の局所的なインターバルと呼ぶ構造を生成する．すなわちプロセッサ p の第 i 番目のインターバル σ_p^i は，$\mathrm{S_A}$または $\mathrm{S_R}$に挟まれた時間領域とその間に実行されたメモリ参照に対応する．なお文

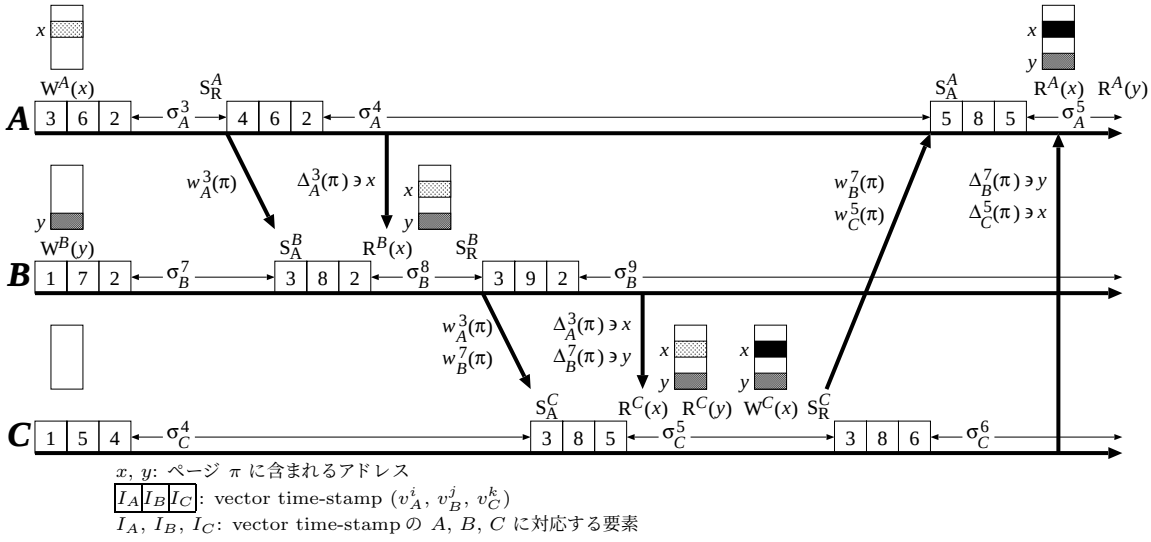

図 3 LRC の機構
Fig. 3 LRC mechanism.

献 13) では明確にされていないが，本論文では $\mathrm{S_R}$はインターバルの末尾に，$\mathrm{S_A}$はインターバルの先頭に，それぞれ位置するものとする．インターバル σ_p^i には，その間に更新されたページを示す書込通知と呼ぶ構造が付加される．また σ_p^i に付加されたページ π の書込通知 $w_p^i(\pi)$ には，必要に応じて σ_p^i の先頭と末尾における π の差分情報である diff（$\Delta_p^i(\pi)$）が付加される．

図 3 の例では，プロセッサ A が σ_A^3 において書き込み $\mathrm{W}^A(x)$ を行い，その結果書込通知 $w_A^3(\pi)$ と diff $\Delta_A^3(\pi)$ が生成される．続いてプロセッサ B が（たとえば）ロック獲得操作 $\mathrm{S_A^B}$ を行うと，そのロックを $\mathrm{S_R^A}$ によって解放した A から B へ $w_A^3(\pi)$ が送られる．この書込通知の受信により B は π の無効化を行い，π が A により（部分的に）更新されたことを認識できるようにする．その後，B は π に対する最初の参照である読み出し $\mathrm{R}^B(x)$ を行った時点で，π に対する A による更新を再度認識し，A に対して diff $\Delta_A^3(\pi)$ の送信を要求する．続いて受信した $\Delta_A^3(\pi)$ を用いて π を更新し，$\mathrm{R}^B(x)$ の正しい結果を得て実行を再開する．

同様の処理は，$\mathrm{S_R^B/S_A^C}$ のロック移送でも行われるが，この場合 $w_A^3(\pi)$ を B が中継する形で C に送る必要がある．同様に $\mathrm{S_R^B/S_A^C}$ のロック移送では，$w_B^7(\pi)$ を C が中継して A に送らなければならない．

どの書き込み通知を送信するかを正しく判断するために，σ_p^i を実行中のプロセッサ p は，インターバルの集合 $\Sigma_p^i = \{\sigma_q^j \mid \sigma_q^j \overset{\mathrm{hb1}}{\rightarrow} \sigma_p^i,\ q \neq p\}$ を管理している．ただし σ_q^j で実行された任意の参照 m と σ_p^i で実行された任意の参照 m' について $m \overset{\mathrm{hb1}}{\rightarrow} m'$ であるとき，またそのときに限り $\sigma_q^j \overset{\mathrm{hb1}}{\rightarrow} \sigma_p^i$ であるとする．たとえば図 3 の例では，$\Sigma_C^5 = \{\sigma_A^0, \ldots, \sigma_A^3, \sigma_B^0, \ldots, \sigma_B^8\}$ となる．この集合の管理のために，σ_p^i を実行中のプロセッサ p は vector time-stamp と呼ぶ配列 v_p^i を有する．配列の個々の要素はプロセッサに対応し，$q \neq p$ なるプロセッサ q に対応する要素 $v_p^i[q]$ は，以下の性質を満たす☆．

$$\forall p \forall q \forall i \forall j ((\sigma_q^j \overset{\mathrm{hb1}}{\rightarrow} \sigma_p^i) \leftrightarrow j \leq v_p^i[q]) \qquad (1)$$

すなわち $v_p^i[q]$ は，σ_p^i に $\overset{\mathrm{hb1}}{\rightarrow}$ 関係で先行する q の直近のインターバルを示す．また p 自身に対応する要素 $v_p^i[p]$ はつねに i に等しい．したがって図 3 の例では，σ_C^5 の直近の $\overset{\mathrm{hb1}}{\rightarrow}$ 先行インターバルは σ_A^3 と σ_B^8 であるので，$v_C^5[A,B,C] = (3,8,5)$ となっている．

上記の式 (1) を満足するために，$\mathrm{S_A}$を実行するプロセッサ a は，対応する $\mathrm{S_R}$を実行したプロセッサ r の vector time-stamp を用いて，自身の vector time-stamp を以下のように更新する．

$$v_a^i[p] \leftarrow \max(v_a^{i-1}[p], v_r^j[p])$$

ただし $\mathrm{S_A}$はインターバル σ_a^i の先頭に，$\mathrm{S_R}$は σ_r^j の末尾に，それぞれ位置するものとする．図 3 の $\mathrm{S_R^B/S_A^C}$ の例では，$v_B^8 = (3,8,2)$ かつ $v_C^4 = (1,5,4)$ であるので，$v_C^5 = (3,8,5)$ となる．

さらに，$v_a^{i-1}[p] < v_r^j[p]$ であるような p については，

☆ オリジナルの LRC では $(\sigma_q^j \overset{\mathrm{hb1}}{\rightarrow} \sigma_p^i) \rightarrow j \leq v_p^i[q]$ を満たすことのみが保証されるが，式 (1) を満たすようにすることは容易である．

インターバルの集合 $S_p = \{\sigma_p^k \mid v_a^{i-1}[p] < k \leq v_r^j[p]\}$ とそれらに付加された書込通知が，$\mathrm{S_R}$を行ったプロセッサ r から $\mathrm{S_A}$を行っているプロセッサ a に送られる．したがって図 3 の $\mathrm{S_A^C}$ の例では，C は $\{\sigma_A^2, \sigma_A^3\}$ と $\{\sigma_B^6, \sigma_B^7, \sigma_B^8\}$，およびそれらに付加された $w_A^3(\pi)$ と $w_B^7(\pi)$ を受け取る（σ_A^3 と σ_B^7 以外のインターバルでは書き込みはないものとする）．

上記の S_p を用いて，$\mathrm{S_A}$を行っているプロセッサ a はインターバル集合；

$$\Sigma_a^i = \Sigma_a^{i-1} \cup \bigcup S_p$$

を得ることができ，かつすべての S_p に含まれるインターバル，すなわち σ_a^i に $\overset{\mathrm{hb1}}{\rightarrow}$ 先行することが新たに判明したインターバルに付加された書込通知が示すページを無効化する．その後，プロセッサ a が無効化されたページへを参照した時点で，Σ_a^i を参照してページの書込通知を求め，さらに直近の書き込みを行ったプロセッサを特定する☆．続いて書き込みを行ったプロセッサに対して書込通知に対応する diff を要求し，それを用いてページを更新する．なお書き込みを行ったプロセッサ以外のプロセッサも diff を保有していることがあり，そのような場合には複数の diff を 1 つのプロセッサから得ることができる．たとえば図 3 の $\mathrm{R}^C(x)$ の例では，B が $\Delta_A^3(\pi)$ を A からすでに受け取っているので，$\Delta_B^7(\pi)$ とともに B から C へ受け渡すことができる．

4. 参照フィルタ

本章では，本論文の主題である参照フィルタについて詳しく述べる．まず 4.1 節では，ある対象キャッシュに対してフィルタ・キャッシュがどのように構成されるかを示す．次に 4.2 節では，キャッシュやメモリのブロックに関する DRF の概念を導入し，DRF であるようなブロックについてはフィルタ・キャッシュが正しく動作する，すなわち履歴から除去される参照は対象キャッシュに必ずヒットすることを証明する．続いて 4.3 節では，DRF ではないブロックの検出方法と，その正当性の証明を示す．

4.1 フィルタ・キャッシュ

3.1 節で述べたように，Shaman が扱う対象キャッシュ C_t はライトバック型かつ無効化型であり，コヒーレンス管理は MSI プロトコルまたはその（有意義な）拡張版に基づくものとしている．対象キャッシュのブロックサイズが l，連想度が w（すなわち w-way セット連想），容量が $w \times \gamma$ であるとき，フロントエンドのフィルタ・キャッシュ C_f はブロックサイズが l で容量が γ のダイレクトマップ型となる．また C_f のコヒーレンス管理は MSI プロトコルにより行われる．なおブロックサイズ，連想度，容量を上記のように設定することにより，Puzak が証明したように[16] C_t において容量性ミスを生じる参照は必ず C_f で容量性ミスを生じ，C_t での競合性ミスは C_f で競合性あるいは容量性ミスとなる．したがって C_f によるフィルタ操作の正当性に関する以後の議論は，コヒーレンス・ミスにのみ注目して行うことができる．

表 1 MSI プロトコルにおける状態遷移
Table 1 State transition of MSI-protocol.

	$trans(m,s)$				$hit(m,s)$	
s	R_l	W_l	R_g	W_g	R_l	W_l
I	**S**	**M**	**I**	**I**	0	0
S	**S**	**M**	**S**	**I**	1	0
M	**M**	**M**	**S**	**I**	1	1

MSI プロトコルにおけるキャッシュの状態遷移は，一般に**表 1** に示すように定義される．ここで $\mathcal{S} = \{\mathbf{M}, \mathbf{S}, \mathbf{I}\}$ はキャッシュ C のブロックがとりうる 3 つの状態，$\mathcal{R}_l = \{\mathrm{R}_l, \mathrm{W}_l\}$ は C を所有するプロセッサによる読み出しと書き込み，$\mathcal{R}_g = \{\mathrm{R}_g, \mathrm{W}_g\}$ は C の所有者以外のプロセッサによる読み出しと書き込みで C に可視であるものをそれぞれ意味する．一般に対象キャッシュのコヒーレンス管理が MSI プロトコルによる場合，メモリ・ブロック b に対する読み出しは b を状態 **M** で保持しているキャッシュに，また書き込みは b を状態 **S** または **M** で保持しているキャッシュに可視となる．

また同表の関数 $t = trans(m,s) : \mathcal{R}_l \cup \mathcal{R}_g \times \mathcal{S} \rightarrow \mathcal{S}$ は，状態が s であるようなブロックに対して参照 m が行われたときの遷移先状態 t を与える．また述語 $hit(m,s) : \mathcal{R}_l \times \mathcal{S} \rightarrow \{0,1\}$ は，状態が s であるようなブロックに対して行った参照 m が他のキャッシュに不可視であるとき，すなわち共有ブロックへの書き込みを含まない狭義のヒットであるとき，またそのときに限り真である．したがって $hit(m,s)$ が偽であれば，m は他のキャッシュに R_g または W_g として可視となりうる．

Shaman のフロントエンド上のプロセッサ p に付随するフィルタ・キャッシュ C_f も，前述のように MSI プロトコルにより管理され，表 1 に示した状態遷移を行う．ただしブロック b に対する参照が R_g または W_g として C_f に可視となる事象は，以下のように定

☆ $\overset{\mathrm{hb1}}{\rightarrow}$ 関係で相互に順序付けできない複数の直近の書き込みが存在する場合，このようなプロセッサは複数存在する．

義される．

- p 自身が行った b の更新を含むような diff を p が他のプロセッサの要求への応答として最初に渡したとき，b に対して C_f に可視の読み出し R_g が行われたと見なす．
- b の更新を含むような diff を p が他のプロセッサから受け取ったとき，b に対して C_f に可視の書き込み W_g が行われたと見なす．

また参照フィルタ操作は，C_f に関して $hit(m,s)$ が真であるような参照 m を履歴から除去する処理である．

4.2 DRF ブロックに対する参照フィルタ操作の正当性

本節ではフィルタ・キャッシュ C_f を用いて行う参照フィルタ操作が，DRF であるようなブロックに対する参照については正当であることを証明する．まず参照フィルタ操作の正当性とブロックに関する DRF を，以下のように定義する．

定義 4 m をキャッシュ C_x を持つプロセッサ p によるメモリ・ブロック b に対する参照とし，$\mathcal{S}_x$ を C_x の状態集合，$state_x(m) \in \mathcal{S}_x$ を m を行った時点での b の C_x における状態とする．述語 $miss_x(m)$ は $hit(m, state_x(m))$ が偽であるとき，またそのときに限り真である．

定義 5 キャッシュ C_f と C_t およびメモリ・ブロック b について，b に対するすべての参照 m が $miss_t(m) \rightarrow miss_f(m)$ を満足するとき，またそのときに限り，C_f は b に関して C_t の正しいフィルタであるという．

定義 6 ある実行におけるメモリ・ブロック b に対する任意の競合する参照の対 m と m' が，$m \overset{\mathrm{hb1}}{\rightarrow} m'$ または $m' \overset{\mathrm{hb1}}{\rightarrow} m$ と順序付けられるとき，またそのときに限り，b はその実行において DRF であるという．

次に C_f のフィルタ操作の正当性の議論において，対象キャッシュ C_t を MSI プロトコルで管理されるものに限定できること，すなわち以下により定義される MSI プロトコルの拡張を考慮しなくてもよいことを示す．

定義 7 状態集合 $\mathcal{S}_{ext}$ を持つキャッシュ C_{ext} のプロトコルは，以下の条件を任意の $s \in \mathcal{S}_{ext}$ に対して満たすような関数 $f : \mathcal{S}_{ext} \rightarrow \mathcal{S}$ が存在するとき，またそのときに限り，拡張 MSI プロトコルであるという．

$$\forall m \in \mathcal{R}_l \cup \mathcal{R}_g \quad (f(trans(m,s)) = trans(m, f(s))) \ \wedge$$
$$\forall m \in \mathcal{R}_l (hit(m, f(s)) \rightarrow hit(m,s)).$$

たとえば exclusive-clean，すなわちブロックが唯一（exclusive）かつメモリと整合している（clean）ことを示す状態 $\mathbf{E}$ を MSI に加えた MESI（または Illinois[19]）プロトコルは，$f(\mathbf{M}) = \mathbf{M}$，$f(\mathbf{E}) = \mathbf{S}$，$f(\mathbf{S}) = \mathbf{S}$ および $f(\mathbf{I}) = \mathbf{I}$ なる関数 f が存在するので，拡張 MSI プロトコルである．我々が知る範囲では，MSI プロトコルの有意義な拡張はすべて拡張 MSI プロトコルである☆．上記の定義 5 と定義 7 から，以下の定理がただちに導かれる．

定理 1 C_t を MSI プロトコルで管理されるキャッシュ，C'_t を拡張 MSI プロトコルで管理されるキャッシュとし，両者の違いはコヒーレンス・プロトコルのみであるとする．このとき，キャッシュ C_f がメモリ・ブロック b に関する C_t の正しいフィルタであるならば，C_f は b に関する C'_t の正しいフィルタである．

続いて参照フィルタ操作の正当性に関する，以下の重要な定理を証明する．

定理 2 C_t を MSI プロトコルで管理されるキャッシュ，C_f をフィルタ・キャッシュとし，両者の構成が 4.1 節に示したものであるとする．このとき，C_f は DRF であるような任意のブロックに関して C_t の正しいフィルタである．

証明： プロセッサ p がインターバル σ_p^i において行ったページ π に含まれるブロック b に対する参照を m とする．また p による b に対する参照の中で，プログラム順で m に先行する直近のものを m' とする．すなわち $m' \overset{\mathrm{po}}{\rightarrow} m$ であり，p による他の b に対する参照 m'' について $m'' \overset{\mathrm{po}}{\rightarrow} m'$ または $m \overset{\mathrm{po}}{\rightarrow} m''$ である．また m' が実行されたインターバルを σ_p^j $(j \leq i)$ とする．

ここでもし，p による b 以外のブロックに対する参照 m_v があって，$m' \overset{\mathrm{po}}{\rightarrow} m_v \overset{\mathrm{po}}{\rightarrow} m$ かつ m_v が b を C_t から追い出すようなものであれば，Puzak が証明したように $miss_t(m)$ かつ $miss_f(m)$ であり，したがって $miss_t(m) \rightarrow miss_f(m)$ である．

そうでない場合，$miss_t(m) \wedge \neg miss_f(m)$ を仮定すると，以下のいずれかが成り立つ．

(1) $state_t(m) = \mathbf{I} \ \wedge \ state_f(m) \neq \mathbf{I}$

(2) $state_t(m) = \mathbf{S} \ \wedge \ state_f(m) = \mathbf{M} \ \wedge \ m = \mathrm{W}_l$

(1) の場合，対象システムにおいて m' による C_t の状態遷移を完了後，かつ $miss_t(m)$ であることが判明する以前の時刻において（以下，単に m' と m の中間の時刻という），$q \neq p$ なるプロセッサ q によりインターバル σ_q^k で行われた b への書き込み m_w が C_t に

☆ ただし，キャッシュ・ブロックのキャッシュ間移送（migration）を行うような一部の例外を除く．

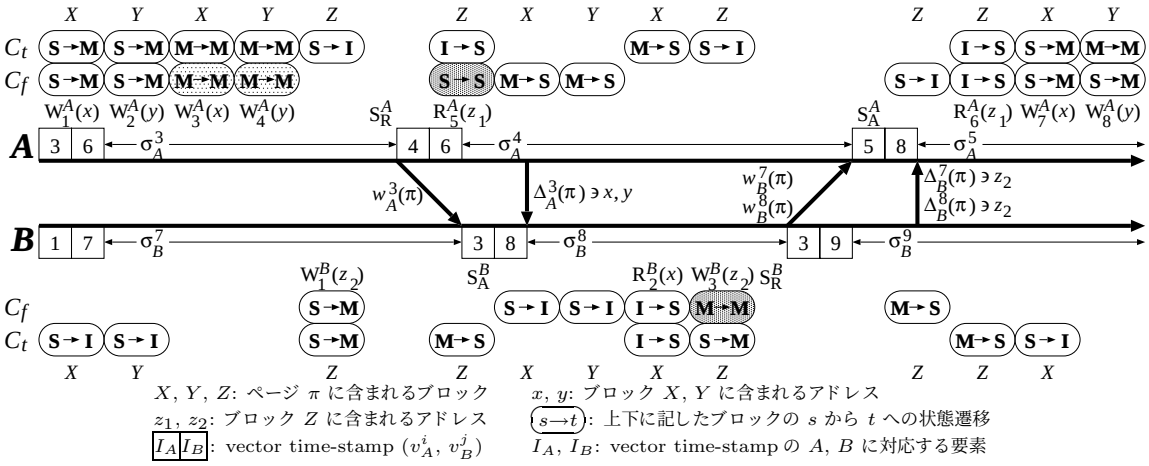

図 4 参照フィルタ操作の例
Fig. 4 Example of reference filtering.

可視となったはずである．ここで b は DRF であるので $m' \overset{\text{hb1}}{\rightarrow} m_w \overset{\text{hb1}}{\rightarrow} m$ であり，よって $\sigma_p^j \overset{\text{hb1}}{\rightarrow} \sigma_q^k \overset{\text{hb1}}{\rightarrow} \sigma_p^i$ である．したがってシミュレータでは，$j < i' \leq i$ なるインターバル $\sigma_p^{i'}$ の先頭に位置する $\mathrm{S_A}$の実行時に，σ_q^k とそれに付随する書込通知 $w_q^k(\pi)$ が p に伝達されており，m' と m の中間の時刻において diff $\Delta_q^k(\pi)$ の取得と C_f 中の b の無効化が行われている．よって b については m' が m に直近の先行参照であることから，$state_f(m) = \mathbf{I}$ であり $miss_f(m)$ が真となって仮定に矛盾する．

(2) の場合，$state_f(m) = \mathbf{M}$ であるので，p による b に対する書き込みの中で，プログラム順で m に先行する直近のもの m_w が存在し，それが実行されたインターバル σ_p^l について $l \leq i$ である．また $state_t(m) = \mathbf{S}$ であるので，対象システムでは m' と m の中間の時刻において，$q \neq p$ なるプロセッサ q によりインターバル σ_q^k で行われた b への読み出し m_r が C_t に可視となったはずである．よって b が DRF であることから，$m_w \overset{\text{hb1}}{\rightarrow} m_r \overset{\text{hb1}}{\rightarrow} m$ であり，$\sigma_p^l \overset{\text{hb1}}{\rightarrow} \sigma_q^k \overset{\text{hb1}}{\rightarrow} \sigma_p^i$ となる．したがってシミュレータでは，$k' \leq k$ なるインターバル $\sigma_q^{k'}$ において $\Delta_p^l(\pi)$ が q によって取得されている．一方 $\sigma_q^k \overset{\text{hb1}}{\rightarrow} \sigma_p^i$ であるので，p が $\Delta_p^l(\pi)$ を q あるいは他のプロセッサへ最初に渡した時点では，p のインターバルは $l < l' < i$ なる $\sigma_p^{l'}$ である．よってシミュレータでは，m_w と m の中間の時刻において $\Delta_p^l(\pi)$ を渡して b の状態を $\mathbf{S}$ としており，b については m' が m に直近の先行書込であることから，$state_f(m) \neq \mathbf{M}$ となる．したがって $miss_f(m)$ が真となって仮定に矛盾する． □

上記の定理が成り立つことは，DRF ブロックを 1 つのデータとして考えれば，LRC が DRF プログラムに対して正しく動作することを利用して直感的に示すこともできる．すなわちプロセッサ p が行った書き込みの結果が他のプロセッサ q によって参照されるのであれば，その値は必ず diff により q に伝達されるので，C_f の状態が p では **S**，q では **I** となることは明らかであろう．

なおこの定理が，$state_t(m) = state_f(m)$ あるいは $miss_t(m) \leftrightarrow miss_f(m)$ を主張するものではないことに注意されたい．実際，**図 4** に示すように DRF であるようなブロックに対して，$\neg miss_t(m) \wedge miss_f(m)$ であるような参照 m が存在しうる．同図において明るい陰を付した状態遷移を起こす書き込み $W_3^A(x)$，$W_4^A(y)$ については，C_t と C_f の両者でヒットするので履歴から除去される．一方，書き込み $W_8^A(y)$ は，C_t では状態が **M** であるのに対し C_f では **S** であるので，広義のミスと判定されて除去されない．

また同図において暗い陰を付した状態遷移を起こす読み出し $R_5^A(z_1)$ と書き込み $W_3^B(z_2)$ は，ブロック Z が DRF ではないことを無視してしまうと，誤って履歴から除去されてしまうことにも注意されたい．

4.3 非 DRF ブロックの検出

前節では C_f を用いた参照フィルタ操作が，DRF であるようなメモリ・ブロックに対しては正しく行われることを証明した．一方，図 4 に示したように，DRF ではないブロックに対してフィルタ操作を行うと，対象キャッシュの広義のミスを起こす参照を誤って除去してしまう．そこで，以下のアルゴリズムによって DRF ではないブロックを検出する．

(1) S/W-DSM の機構によりプロセッサ p にコピー

されている各メモリ・ブロック b に対して，最終参照インターバル $\tau_p(b)$ と呼ぶフィールドを付加する．ある時点において b への最後の参照がインターバル σ_p^i で行われたとき，またそのときに限り $\tau_p(b) = i$ である．

(2) 各々の diff $\Delta_p^i(\pi)$ に対し，vector time-stamp v_p^i を付加する．プロセッサ q が diff $\Delta_p^i(\pi)$ を取得する際，ブロック b の更新が $\Delta_p^i(\pi)$ に含まれていて，かつ $\tau_q(b) > v_p^i[q]$ であれば，b を非 DRF ブロックとしてマークする．

図 4 に示した例では，プロセッサ B は σ_B^7 でブロック X や Y の参照を行っていないので，$\Delta_A^3(\pi)$ を取得した時点での $\tau_B(X)$ と $\tau_B(Y)$ の値は，いずれも 6 以下である．したがってこの diff に付された vector time-stamp $v_A^3[B]$ が 6 であることから，X と Y には非 DRF のマークは付けられない．一方プロセッサ A は，σ_A^4 で $\mathrm{R}_5^A(z_1)$ により Z を参照しており，Δ_B^7 と Δ_B^8 を取得した時点では $\tau_A(Z) = 4$ である．この値は diff に付された $v_B^7[A] = 1$ や $v_B^8[A] = 3$ よりも大きいことから，Z は非 DRF ブロックとしてマークされる．

上記のアルゴリズムは，プロセッサ p による DRF ブロック b への書き込み m_w と，m_w に関する diff $\Delta_p^i(\pi)$ をプロセッサ q が取得する以前に行った参照 m について，必ず $m \overset{\mathrm{hb1}}{\rightarrow} m_w$ が成り立つことを利用している．すなわち m に後続し m_w に先行する q から p へのロック移送によって $\tau_q(b) \leq l$ なる q の l 番目のインターバルが p に伝えられ，これが $\Delta_p^i(\pi)$ に $v_p^i[q]$ として付加されるため，$\tau_q(b) \leq v_p^i[q]$ が必ず成立する．したがってこのアルゴリズムによりプログラムの実行が完了した時点において，あるブロックが DRF であることを判定できることが以下の定理により示される．

定理 3 ある実行の完了時点でメモリ・ブロック b が非 DRF とマークされていなければ，b は DRF である．

証明： プロセッサ p と q が各々のインターバル σ_p^i と σ_q^j で行った b に対する参照を，それぞれ m と m' とし，m は書き込みであるとする．また $\Delta_p^i(b)$ を m による更新を含む diff とし，q が $\Delta_p^i(b)$ を取得するインターバルを σ_q^k，また取得時点での $\tau_q(b)$ の値を l とする．

b は非 DRF とマークされなかったのであるから $l \leq v_p^i[q]$ であり，式 (1) により $\sigma_q^l \overset{\mathrm{hb1}}{\rightarrow} \sigma_p^i$ が導かれる．また $\Delta_p^i(b)$ に関する書込通知は σ_q^k の先頭かそれ以前に q に伝えられているので，$\sigma_p^i \overset{\mathrm{hb1}}{\rightarrow} \sigma_q^k$ が導かれる．さらに l は σ_q^k での b の最終参照インターバルであるので $l \leq k$ であり，m' を行ったインターバル σ_q^j との関係が $l < j < k$ であることはない．したがって $j \leq l$ であるか $k \geq j$ である．

ここで $\sigma_q^j = \sigma_q^l \vee \sigma_q^j \overset{\mathrm{hb1}}{\rightarrow} \sigma_q^l$ を $\sigma_q^j \overset{\mathrm{hb1}}{\Rightarrow} \sigma_q^l$ と表記すると，もし $j \leq l$ であれば $\sigma_q^j \overset{\mathrm{hb1}}{\Rightarrow} \sigma_q^l \overset{\mathrm{hb1}}{\rightarrow} \sigma_p^i$ であり，したがって $m' \overset{\mathrm{hb1}}{\rightarrow} m$ である．一方 $k \geq j$ であれば $\sigma_p^i \overset{\mathrm{hb1}}{\rightarrow} \sigma_q^k \overset{\mathrm{hb1}}{\Rightarrow} \sigma_q^j$ であり，したがって $m \overset{\mathrm{hb1}}{\rightarrow} m'$ である．よって b に対する任意の競合する参照対 m と m' について，$m' \overset{\mathrm{hb1}}{\rightarrow} m$ あるいは $m \overset{\mathrm{hb1}}{\rightarrow} m'$ が成り立つので，定義 6 により b は DRF である． □

さて上記の判定アルゴリズムでは，実行完了時にブロックが DRF であること，すなわち参照フィルタ操作が正しく行われたことが明らかになる．したがって非 DRF ブロックが発見された場合，以下の手順によりシミュレーション実行を 2 回繰り返す必要がある．

フェーズ 1 各フロントエンド・ノードでは，ワークロード・プログラムを実行しながら参照履歴を生成してバックエンドに送信する．その際，初期的にはすべてのブロックが DRF であると仮定し，フィルタ・キャッシュによる参照フィルタ操作を行う．この過程であるフロントエンド・ノードが非 DRF ブロックを発見すると，他のすべてのノードに参照履歴の生成と送信の中止を，またバックエンドにも履歴を用いたシミュレーションの中止を通知する．ただし，すべての非 DRF ブロックのマーキングを行うために，プログラムの実行自体は継続する．実行が完了すると，マークされた非 DRF ブロックのアドレスをすべてのノード間で交換し，各ノードが保持するブロックのコピーにもマーク付けを行う．またマークされたブロックが存在しなければ，シミュレーションは完了し，フェーズ 2 の実行は行われない．

フェーズ 2 各フロントエンド・ノードでは，ワークロード・プログラムを再実行し，参照履歴を生成してバックエンドに送信する．その際，フィルタ・キャッシュによる参照フィルタ操作は DRF ブロックに対してのみ行い，非 DRF とマークされたブロックへの参照はすべてバックエンドに送る．したがってバックエンドは，対象キャッシュをミスする可能性のあるすべての参照を得ることができる．

なお各フロントエンド・ノードのメモリ容量が十分に大きく，書込通知や diff のガベージ・コレクションが不要であるならば，以下のような方法によりフェーズ 2 の実行時間を短縮することができる．すなわち，フェーズ 1 で通信されたすべての書込通知を保存する

ことができれば，フェーズ 2 では S_R/S_A の処理をノード間通信を行わずに実行できる．またすべての diff をも保存することができれば，フロントエンド・ノード間での通信は完全に排除される．

5. 実　　験

参照フィルタ操作の有効性を評価するために，PC クラスタを用いた Shaman のプロトタイプ実装と，LU 分解と FFT の 2 つの SPLASH-2 カーネル[23]を用いたメモリ参照数の測定を行った．PC クラスタは，Pentium II（450 MHz，128 MB）をベースとする PC を 100Base-TX のイーサネットで結合したものである．対象システムは，バス結合された 4 プロセッサの共有メモリ・マルチプロセッサとした．また対象キャッシュは 1 レベルの命令/データ分離型のダイレクトマップ・キャッシュとし，容量は各々64 KB，ブロックサイズは 16 B，コヒーレンス管理は MESI プロトコルとした．

メモリ参照数の測定はデータ・キャッシュに対してのみ行い，総計（N_t），参照フィルタを通過した数（N_f），および共有ブロックへの書き込みを含む対象キャッシュの広義のミス回数（N_m）を，それぞれ読み出しと書き込みごとに集計した．なお，いずれのワークロードでもキャッシュを意識した処理分割がなされているので，ブロックに対する write-write false sharing などは生じず，したがって非 DRF ブロックは発見されなかった．

表 2 の測定結果に示すように，LU 分解では 0.4%，FFT では 1.4%の参照だけがフィルタ通過しており，バックエンドへはごく一部の参照だけを渡すことができている．したがって 3.2 節に示した参照履歴の通信オーバヘッドに関する仮定を用いれば，転送時間に関する限り FFT の場合でも 90 倍程度の対実機実行時間比が達成可能と見積もられる．またフィルタを通過した参照，すなわちフィルタ・キャッシュをミスした参照のほとんどが対象キャッシュもミスしており，バックエンドに渡される参照の有効率 N_m/N_f は LU 分解で 98.6%，FFT で 99.5%ときわめて高い．これらの結果から，フィルタ・キャッシュによりバックエンドにとって不要な参照をほぼ完全に除去できることと，参照履歴の削減効果がきわめて高いことが結論付けられる．

なお有効率が 100%ではないのは，対象キャッシュにヒットするような書き込みの一部が，以下の理由によりフィルタ・キャッシュではミス（共有ブロック書き込み）と判断されるからである．

(1) 対象キャッシュでの状態が **E**（exclusive-clean）であるブロックは，フィルタ・キャッシュではつねに **S**（shared）と見なされる．

(2) 対象キャッシュでの状態が **M**（modified）であるブロックが，フィルタ・キャッシュでは **S** と見なされることがある．すなわちブロックを含む diff の送受が行われると，diff を取得したプロセッサがそのブロックを実際に参照しなくても，フィルタ・キャッシュでは **M** から **S** への遷移が生じる．

表 2 評価結果
Table 2 Evaluation result.

workload		N_t	N_f	N_m	N_f/N_t	N_m/N_f
LU	(read)	1,090,896	4,187	4,187	0.4%	100.0%
	(write)	388,220	2,181	2,093	0.6	96.0
	(total)	1,479,116	6,368	6,280	0.4	98.6
FFT	(read)	2,105,634	33,300	33,300	1.6	100.0
	(write)	1,218,316	14,508	14,246	1.2	98.2
	(total)	3,323,950	47,808	47,546	1.4	99.5

6. おわりに

本論文では，共有メモリ・マルチプロセッサを対象とする分散シミュレータにおいて，フロントエンド・ノードが生成するメモリ参照履歴を削減する方式を提案した．この方式の眼目は，多くのワークロード・プログラムが持つ DRF の性質と，S/W-DSM のコヒーレンス管理方式の 1 つである LRC の実行機構を利用し，フロントエンド・ノードのキャッシュを参照フィルタとすることにある．また，このフィルタ・キャッシュが DRF ブロックに対して正しいフィルタ操作を行うこと，すなわち除去される参照は必ず対象キャッシュにヒットするものであることを証明した．さらに，DRF でないようなブロックの検出方法の提案と，その正当性の証明も行った．

我々が開発中の分散シミュレータ Shaman のプロトタイプによる評価では，フィルタ・キャッシュを通過する参照は 1.4%以下であり，参照履歴の削減効果がきわめて高いことが実証された．またフィルタを通過した参照の 98.6%以上が，実際にバックエンドでの対象メモリ・システムのシミュレーションに必要なものであることも明らかになった．

本研究の最大の課題は，Shaman を完成させ，参照フィルタ操作をはじめとする我々の分散シミュレーション技法の有効性を，シミュレーション速度の面でも実証することである．また Shaman のバックエンドに関するシミュレーション技法，すなわち遅延隠蔽機構の動作タイミングを高い精度で再現する手法や，バックエンドの並列化なども，今後の課題として残されている．

参 考 文 献

1) Adve, S.V. and Hill, M.D.: Weak Ordering—A New Definition, *Proc. 17th Intl. Symp. Computer Architecture*, pp.2–14 (1990).
2) Adve, S.V. and Hill, M.D.: A Unified Formalization of Four Shared-Memory Models, *IEEE Trans. Parallel and Distributed Systems*, Vol.4, No.6, pp.613–624 (1993).
3) Bailey, M.L. and Snyder, L.: A Model for Comparing Synchronization Strategies for Parallel Logic-Level Simulation, *Proc. Conf. Computer-Aided Design*, pp.502–505 (1989).
4) Brewer, E.A., Dellarocas, C.N., Colbrook, A. and Weihl, W.E.: PROTEUS: A High-Performance Parallel-Architecture Simulator, Technical Report MIT/LCS/TR-516, MIT (1991).
5) Carter, J.B., Bennet, J.K. and Zwaenepoel, W.: Implementation and Performance of Munin, *Proc. 13th ACM Symp. Operating System Principles*, pp.152–164 (1991).
6) Chandy, K.M. and Misra, J.: Asynchronous Distributed Simulation via a Sequence of Parallel Computations, *Comm. ACM*, Vol.24, No.11, pp.198–206 (1981).
7) Gabbay, F. and Mendelson, A.: Smart: An Advanced Shared-Memory Simulator—Towards a System-Level Simulation Environment, *Proc. MASCOTS'97* (1997).
8) Goldschmidt, S.R. and Hennesy, J.: The Accuracy of Trace-Driven Simulation of Multiprocessors, *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp.146–157 (1993).
9) Iftode, L., Dubnicki, C., Felten, E.W. and Li, K.: Improving Release-Consistent Shared Virtual Memory Using Automatic Update, *Proc. 2nd IEEE Symp. High-Performance Computer Architecture*, pp.14–25 (1996).
10) 今福　茂，大野和彦，中島　浩：共有メモリ型並列計算機の分散シミュレータ，情報処理学会研究報告，99-ARC-136, pp.37–42 (2000).
11) Imafuku, S., Ohno, K. and Nakashima, H.: Reference Filtering for Distributed Simulation of Shared Memory Multiprocessors, *Proc. 34th Annual Simulation Symp.*, pp.219–226 (2001).
12) Jefferson, D.R.: Virtual Time, *ACM Trans. Prog. Lang. Syst.*, Vol.7, No.3, pp.404–425 (1985).
13) Keleher, P.: Lazy Release Consistency for Distributed Shared Memory, Ph.D. Thesis, Dept. Computer Science, Rice Univ. (1994).
14) Li, K. and Hudak, P.: Memory Coherence in Shared Virtual Memory System, *ACM Trans. Comput. Syst.*, Vol.7, No.3, pp.63–79 (1992).
15) Magnusson, P. and Werner, B.: Efficient Memory Simulation in SimICS, *Proc. 28th Annual Simulation Symp.* (1995).
16) Puzak, T.: Analysis of Cache Replacement Algorithms, Ph.D. Thesis, Univ. Massachusetts (1985).
17) Reinhardt, S.K., Hill, M.D., Larus, J.R., Lebeck, A.R., Lewis, J.C. and Wood, D.A.: The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers, *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp.48–60 (1993).
18) Rothman, J.B. and Smith, A.J.: Multiprocessor Memory Reference Generator Using Cerberus, *Proc. MASCOTS'99*, pp.278–287 (1999).
19) Stenstrom, P.: A Survey of Cache Coherence Schemes for Multiprocessors, *Computer*, Vol.23, No.6, pp.14–24 (1990).
20) Uhlig, R.A. and Mudge, T.N.: Trace-Driven Memory Simulation: A Survey, *ACM Computing Surveys*, Vol.29, No.2, pp.128–170 (1997).
21) Veenstra, J.E. and Fowler, R.J.: MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors, *Proc. MASCOTS'94*, pp.201–207 (1994).
22) Wang, W.H. and Baer, J.L.: Efficient Trace-Driven Simulation Methods for Cache Performance Analysis, *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp.27–36 (1990).
23) Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd Intl. Symp. Computer Architecture*, pp.24–36 (1995).

(平成 13 年 1 月 31 日受付)
(平成 13 年 4 月 9 日採録)

今福　茂

2000 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．同年セイコーエプソン（株）入社．在学中は並列マシンの分散シミュレーションに関する研究に従事．

大野 和彦（正会員）

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）

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．