情報処理学会論文誌　データベース　Vol. 2　No. 4　32–48 (Dec. 2009)

推薦論文

ロックフリーGCLOCKページ置換アルゴリズム

油井　誠[†1,†2]　宮崎　純[†3]　植村俊亮[†4]
加藤博一[†3]　山名早人[†5]

GCLOCKに基づくロックフリーなページ置換アルゴリズムNb-GCLOCKを提案する．バッファ管理モジュールへの並行アクセスは，CPUのプロセッサ数に対するデータベースのスケーラビリティを阻害する主要な要因である．本論文では，Nb-GCLOCKと無待機ハッシュ表の組合せにより，要求されたページをバッファフレームに固定する*bufferfix*処理をノンブロッキングに行う手法を提案する．実験結果により，既存のロックに基づくバッファ管理手法が16プロセッサ以上，プロセッサ数に対するスケーラビリティを示さないのに対して，我々の手法が64プロセッサまでほぼ線形のスケーラビリティを示すことを明らかにし，提案手法の有効性を示す．

A Lock-free GCLOCK Page Replacement Algorithm

Makoto Yui,[†1,†2] Jun Miyazaki,[†3]
Shunsuke Uemura,[†4] Hirokazu Kato[†3]
and Hayato Yamana[†5]

In this paper, we propose a lock-free variant of the GCLOCK page replacement algorithm, named Nb-GCLOCK. Concurrent access to the buffer management module is a major factor that prevents database scalability to processors. Therefore, we propose a non-blocking scheme for *bufferfix* operations that fix buffer frames for requested pages without locks by combining Nb-GCLOCK and a wait-free hash table. Our experimental results revealed that our scheme can obtain nearly linear scalability to processors up to 64 processors, although the existing locking-based schemes do not scale beyond 16 processors.

1. まえがき

マルチコアプロセッサやマルチスレッド化されたチップ設計に代表されるアプリケーションの性能向上をソフトウェアのマルチスレッド化に頼る近年のマイクロプロセッサの開発動向は，ソフトウェアの並列化への抜本的な方向転換をソフトウェア開発に迫っている[1)]．こうしたマイクロプロセッサの開発動向は，データベースの研究開発[2)]や実用にも影響を及ぼしており，産業界におけるデータベース利用への影響の1つが，CPUのプロセッサ数に対するDBMSのスケーラビリティ（以下，CPUスケーラビリティ）の問題に現れている．オープンソースのデータベース管理システムに関する経験的な研究[3)]や探究[4)]により，主要なオープンソースDBMSであるBerkeleyDB，MySQL，およびPostgreSQLはCPUスケーラビリティに乏しく，それぞれ4スレッド，8スレッド，16スレッド付近までしかプロセッサ数に対してスケールしないという問題が報告されている．その中で，データベース内部の共有データ構造（特にバッファ管理モジュール）への並行アクセスがデータベースのCPUスケーラビリティを阻害する主要な要因となっていることが分かっている．

バッファ管理の設計上の問題は，主にページをバッファフレームに固定する際に生じるロックの競合に起因する．一般的に，同期処理の並行性を向上させるための施策には次の3つがある．

(a) ロックを取得せずに，ロックを必要としないデータ構造とアルゴリズム[5)]を採用する．ロックを取得せずに同期処理を行う手法をノンブロッキング同期（non-blocking synchronization）という．

(b) ロックの粒度を小さくする．細粒度に分解したロックはロック自体の負荷（ロックの取得・開放の負荷の総和）を高める可能性があるが，ロックの競合を減らすことがで

†1 早稲田大学IT研究機構ITバイオ研究所
Research Institute of Information Technology Biology, Information Technology Research Organization, Waseda University
†2 日本学術振興会特別研究員（PD）
Research Fellow of the Japan Society for the Promotion of Science, PD
†3 奈良先端科学技術大学院大学情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology
†4 奈良産業大学情報学部情報学科
Department of Informatics, Faculty of Informatics, Nara Sangyo University
†5 早稲田大学理工学術院
Faculty of Science and Engineering, Waseda University

きる．

(c) より軽量なロックプロトコルを採用する．スレッドが短時間だけブロックされる場合，オペレーティングシステムのプロセススケジューリングやコンテキストスイッチのオーバヘッドを防ぐことができるため，スピンロックは効率的である．

PostgreSQL や MySQL のような従来のデータベースシステムは (b) と (c) の改善によってスケーラビリティの問題に対処してきた．本論文ではより急進的な解決策である，いっさいのロックを取得しないノンブロッキング同期 (a) によるスケーラブルなバッファ管理手法を提案する．

本論文は，データベースのバッファ管理にノンブロッキング同期を適用する初めての試みである．バッファ管理に関して述べた論文は，多様なワークロードに対して高いバッファ利用効率を導くことを主眼とする一方，本論文で扱うバッファ管理の並行性に関する詳細な議論は，個々の開発者の経験的知識に任されていた．これまでバッファ管理の並行性が十分に議論されてこなかった原因としては，大規模マルチプロセッサが普及していなかったことや，バッファヒット率を高め I/O 処理回数を最小化することに主眼が置かれていたことが考えられる．しかし，バッファ管理が担う要求ページをバッファフレームに固定する bufferfix 処理[6] は，必ずしも I/O 依存の問題ではない．bufferfix 処理でディスク I/O が発生するのは，ページ置換時に置換対象のページがディスクと同期が必要（ダーティ）である場合であるが，近代的な DBMS はダーティページのプリフラッシュやページ置換アルゴリズムの工夫（たとえば，ダーティでないページを優先的に置換対象とする）により，バッファ割当て時のダーティページの書き込み頻度を低減させている[6] からである．つまり，メモリが十分にあって十分なバッファプールを確保できれば，bufferfix 処理においてダーティページが置換の対象となる頻度は極小化できる．このため，bufferfix 処理は CPU 依存になる傾向にあり，マルチプロセッサ環境においてバッファ管理の CPU スケーラビリティの問題が生じる．実際，このバッファの固定/開放操作はデータベースシステムで最も頻繁に呼び出される基本操作の 1 つであり，排他制御の競合確率も高い[7] ため，バッファの固定/開放操作が高速であることがきわめて重要となる．

提案手法の有効性を主張するためには，提案手法が CPU 依存の処理と I/O バウンド処理の性能をいかに改善するかを明らかにする必要がある．I/O 処理がどれだけ支配的であるかを決める主要な要因の 1 つがディスク I/O の発生頻度であり，バッファ管理におけるディスク I/O の発生頻度は，バッファヒット率（と置換対象のページがダーティである割合）に依存する．そこで，本論文ではバッファヒット率の変化に着目し，バッファヒット率の変化に対して得られる CPU スケーラビリティについて考察する．

バッファの参照に用いるハッシュ表のノンブロッキングアルゴリズムは，すでにいくつか提案されている[8),9)]．本論文では，バッファの探索に既存の無待機（wait-free）ハッシュ表を用いるものとし，ページ置換アルゴリズムの並行性に焦点を当て，Generalized CLOCK（GCLOCK）ページ置換アルゴリズム[10)] をノンブロッキングに実現する Nb-GCLOCK ページ置換アルゴリズムを提案する．Sun UltraSPARC T2 を用いた評価実験により，既存のロックを用いるページ置換アルゴリズムでは 16 プロセッサ付近にスケーラビリティの限界があるのに対して，提案手法では 64 プロセッサまで，ほぼ線形のスケーラビリティを確保しうることを示し，提案手法が高い CPU スケーラビリティを有することを明らかにする．本論文の貢献は次の点にある．

- 同期基本命令を利用したノンブロッキングの GCLOCK アルゴリズムとノンブロッキングのバッファ管理手法の組を提示した．その中で (a) 単一競合点の問題を示し，CLOCK のカウンタストラインピングの必要性を論じ，(b) CAS 命令と pread システムコールによる楽観的な並行 I/O が，現代のハードウェアにおいて有効に機能する局面があることを示した．この楽観的な I/O をバッファ管理に導入したのは，他に先駆けるものである．
- 従来，バッファヒット率を向上させることが主眼であったバッファ管理において，LRU，2Q，CLOCK を例に出してバッファ管理における並行性の問題を明らかにし，ノンブロッキングデータ構造をデータベースのバッファ管理に初めて導入した．

2. 関連事項および既存手法の問題点

例をあげてバッファ管理のスケーラビリティを阻害する要因を説明する．一般的にバッファ管理は，**図 1** のように，バッファ参照表（buffer lookup table）とページ置換ポリシを管理するモジュール，およびバッファプール（buffer pool）から構成される．バッファ参照表は一般的にハッシュ表で構成される[6),11)]．ページ置換ポリシには，LRU や CLOCK，あるいはその派生が利用される．以下，ページ置換ポリシを管理するモジュールのことを（ページ）置換リストと呼ぶ．

2.1 バッファ管理の内部ロック

共有バッファへの並行アクセスは，マルチプロセッサ環境において，スケーラビリティの問題を引き起こす．具体的には，バッファ管理の危険領域（critical section）の排他制御で競合が発生する．

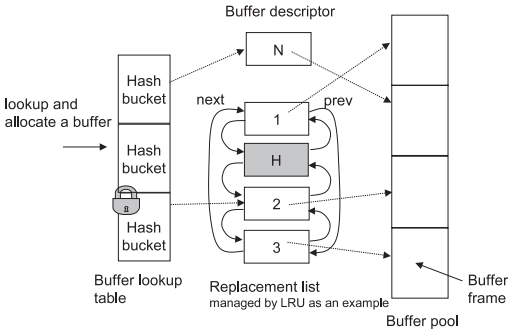


図 1　典型的なバッファ管理の構成
Fig. 1　Typical organization of a buffer manager.

排他制御の具体例としては，目的のページがバッファプールに存在するか探索する際に，バッファ参照表に共有ロック（shared lock）が取得される．また，バッファフレームへのページ割当てを変更する際に，バッファ管理に排他ロック（exclusive lock）が取得される．この排他ロックは，ページ置換リストの調整とバッファ参照表の更新にまたがって保持する必要がある．これはバッファフレームへのページ割当てを変更した直後は，異なるページ識別子からの参照がバッファ参照表に残っているためである．

LRU 連鎖の置き換えはつねに排他モードを必要とするため，バッファ管理にマルチユーザからの並行した操作要求が行われたとき，あるページング要求がキャッシュミスを起こし排他ロックを取得すると，その排他ロックは他のスレッドが共有ロックや排他ロックを取得するのを妨げてしまう．そのため，システム全体にわたるロック解放待ちが各スレッドのページスキャンのたびに発生し，そのことでロックの取り合いが生じる．高トラヒックのロックはコンボイ現象（ロック開放待ち行列）[7] を発生させる可能性がある．

PostgreSQL 8.2 と MySQL 5.0.30 は，バッファ管理におけるバッファ参照表のロックに，より細粒度のロックを採用することで，このロックの競合問題に対処した．ロックの競合を減らすために PostgreSQL と MySQL が採用した手段は，ハッシュ表のジャイアントロック（giant lock）をハッシュバケットごとに分割するロックストライピング（lock-striping）と呼ばれる，ロックの競合を減らす手段として基本的なものである．

2.2　バッファ置換アルゴリズムの並行性

Least Recently Used（LRU）と 2Q[12]，GCLOCK の 3 つを例にとり，ページ置換アルゴリズムの並行性を議論する．

図 1 にあるように，一般的に LRU はページ置換ポリシの管理に双方向リンクリスト（LRU 連鎖）を用いる．新しく割り当てられたバッファはリストの先頭に追加され，最後尾のバッファが置換される．リスト中にあるバッファが参照された場合，そのバッファはリストの先頭に移動される．

LRU アルゴリズムは，シングルスレッドからアクセスされるワークロードには問題なく動作するが，マルチスレッドのワークロードで応答性能が低下することが知られており，いくつかのシステムでは，LRU に近似した性能を示す CLOCK[13] が代替として利用される[14]．LRU が並行アクセスに弱いのは，双方向リンクリストにアクセスするときつねにリンクリスト全体に排他制御をかける必要があるからである．CLOCK は参照時に参照したエントリのカウンタを原子的（不可分）に変更するだけでよく，LRU のように全体にわたる排他制御を必要としない．GCLOCK は，通常の CLOCK が用いるビットフラグの代わりに，個々のページ P_i ごとにリファレンスカウントを利用する CLOCK の改良である．ページ P_i への参照は，対応するカウンタ $RC(i)$ を増分させる．$RC(i)$ は初期値 0 からなり，参照ごとに 1 ずつ増加していく．キャッシュミスが発生した際には，最初に $RC(i)$ が 0 であるページ P_i が発見されるまで，たどったページのリファレンスカウントを減少させながら循環バッファを探索していく．

LRU はシーケンシャル・スキャンで重要（ホット）なエントリを追い出してしまうことがあることが問題点として知られている[14]．スキャンに対して頑健なアルゴリズムである単純（simple）2Q アルゴリズムでは，キャッシュされるアイテム群をホットとコールドの FIFO キューに別けて管理する．完全（full）バージョンの 2Q アルゴリズムは，3 つの FIFO を利用する．Oracle のバッファ管理[15] は，単純 2Q と同様に LRU 連鎖を 2 つのホットとコールドのキューで管理する．2Q はシーケンシャル・スキャンに対して頑健であるが，LRU と同様にリンクリストの整合性を保つための広範な排他区間を必要とするため，マルチプロセッサ環境においてバッファ管理のスケーラビリティを阻害することがある．

シーケンシャルスキャンに対応するため，当初 PostgreSQL は 2Q アルゴリズムを採用していたが，マルチプロセッサ環境が一般的となったことで 2Q の同期処理のペナルティを無視できなくなったため，並行性に優れた CLOCK[13] ベースのアルゴリズムを採用することとなった．こうした事態は 2Q を採用していた我々が開発するデータベースシステム[16] においても確認されたことであり，バッファ管理の戦略としてバッファヒット率を重視しページ I/O を最小化することは，現在のハードウェアにおいて必ずしも最優先の要求ではなく，

CPU 依存の処理が全体の性能に重要な影響を与えるという本論文の主張を支持するものである．

3. ノンブロッキング GCLOCK ページ置換アルゴリズム

提案手法であるノンブロッキングの GCLOCK ページ置換アルゴリズム Non-blocking GCLOCK（略して Nb-GCLOCK）について説明する．Nb-GCLOCK アルゴリズムは，ノンブロッキングアクセスをサポートする以外は基本的に GCLOCK ページ置換アルゴリズム[10] の性質に従う．GCLOCK を我々のアルゴリズムの下地に選んだのは，次の理由による．

(1) 双方向リストを用いる LRU 等のバッファ置換ポリシと比較して，CLOCK（とその派生）は負荷が低く並行性が高いことが知られており，広く利用されている．そのため，CLOCK をノンブロッキングにすることは意義が大きい．

(2) CLOCK に由来するアルゴリズムの中で，GCLOCK は交差検定によりその性能が詳しく分析されている[10),17]．単純な CLOCK がバッファの利用頻度を考慮に入れないのに対して，GCLOCK は利用頻度が考慮される．

(3) 変数アクセスで競合が発生する確率が低い．ここで競合とは，複数のスレッドが同時に同じメモリロケーションにアクセスする状態を指す．たとえば，CLOCK では単一（あるいは少数の）ビットマップを共有し利用頻度を管理するが，ビットマップの更新時に偽共有（false sharing）が発生するため，キャッシュ一貫性が保たれる共有メモリマルチプロセッサにおいて非効率となる．GCLOCK ではバッファフレームごとに参照カウンタを持つため，競合が発生する確率が低い．

提案手法の Nb-GCLOCK は，ロックフリーのページ置換を実現する．

ノンブロッキング・アルゴリズムには次の 2 つの重要な性質がある[18]．第 1 にいくつかの処理を有限時間で終える場合は，そのノンブロッキング・アルゴリズムはロックフリー（lock-free）である．ロックフリーでは，少なくとも 1 つのスレッドがその役割を進行し続けることが保証される．第 2 にすべての処理を有限時間で終える場合は，そのノンブロッキング・アルゴリズムは無待機（wait-free）である．前者が活性（liveness）を後者が公平性（fairness）をそれぞれ保証する．この観点で，本論文で提案するバッファ管理手法はロックフリーである．

提案手法は，汎用のキャッシュ用途にも利用できるように，バッファプール中のページがすべて使用中である（ピンされている）ときには，ページをバッファフレームに割り当てるスレッドの実行権を一時放棄してから，ノンブロッキングの処理の試行を続ける戦略をとる．このとき，すべての処理が有限時間で完了することを保証することは，すべてのページがピンされている場合を考慮に入れると不可能である．すべてのページが利用中である場合の挙動は，キャッシュを利用するアプリケーションがバッファ割当て処理の失敗を許容するか否かによって選択すればよい．バッファ管理においては，すべてのページが利用中である場合，十分なバッファ容量が確保できれば起こりえない事象としてトランザクションのアボートが行われることが多い[6),14]．この状況は，4 章における実験や我々のデータベースシステムの運用において観察されたことはない．

我々は，提案手法を Java 5.0[19] で提供されている原子操作を利用して実装した．2 章で述べたように，バッファ参照表には無待機のハッシュ表を用いる．無待機のハッシュ表には，文献 20) でオープンソースソフトウェアとして公開されている実装（以下，NonBlockingHashMap）を利用する．NonBlockingHashMap で利用する putIfAbsent メソッドは，指定されたキーが値と関連付けられていないとき，指定された値を指定されたキーと関連付けてハッシュ表に格納する．戻り値では，指定されたキーに関連したマッピングがあればその以前の値，マッピングがなければ null が返る．remove メソッドは，キーが指定された値に現在マッピングされている場合にのみ，そのキーのエントリを削除する．値が削除された場合は真，そうでない場合は偽を返す．これらの操作は原子的に行われる．この無待機ハッシュ表の実装は，concurrent cuckoo hashing[18] やハッシュバケットごとにロックストライピングを行ったハッシュ表に置き換えることが可能であるが，その場合，前者ではライブロックが生じる可能性があり，後者ではロックのストライピング数の不足やハッシュ値の隔たりによってロック解放待ちが生じる可能性があることに注意が必要である．

3.1 Nb-GCLOCK アルゴリズム

Nb-GCLOCK のアルゴリズムを図 4 と図 5 で説明する．本提案の貢献は，GCLOCK をノンブロッキングに行うアルゴリズム（図 5）とバッファの割当てを行うアルゴリズム（図 4）からなるが，これらのアルゴリズムは相補的に関連しており，単独で有効に働くアルゴリズムではない．

元来の GCLOCK を用いた場合のバッファ管理では，バッファ参照表に該当するページが存在する場合にバッファ参照表の共有ロックとバッファフレームの短時間のスピンロックを取得するだけでよいという点を除けば，LRU 等を用いた古典的なバッファ管理手法[6] と同様の排他制御が必要である．ハードウェア上で並行して動作する 10 以上のスレッドが同時にバッファ管理にアクセスしているような状況では，全体のバッファヒット率が 50%でも，ほぼつねに排他ロックがかかることになるため，複雑なロックプロトコルを採用するこ

とはロック獲得コストが高い分だけ逆効果となる[3]．

一般的なバッファ管理システムでは，バッファを参照する際に初めにバッファ参照表に共有ロックを取得する．さらにバッファの置換が必要な場合，まず置換リスト（たとえば，LRU や GCLOCK において）に排他ロックをとり，その次にハッシュバケットのロックを取得し，ページを固定するバッファフレームに上位モジュールが要求するロックモードでロックを取得する[6]．仮に，無待機ハッシュ表によりバッファ参照表の排他ロックを取り除き，ある箇所のスループットが上がったとしても，他にロックが存在する限り処理の要求がそこにたまってスループットが低下してしまう．たとえば，ページ置換リストのページ置換時の排他ロックは，現在のデータベースシステムに残された課題の 1 つである．提案手法の貢献は，GCLOCK を拡張した Nb-GCLOCK アルゴリズムと，それと組にしたノンブロッキングのバッファ管理手法を用いることにより，バッファ管理に内在するロックを注意深く取り除いたことにある．

アルゴリズムの説明で利用する AtomicInteger，AtomicBoolean，AtomicArray，AtomicCounter へのすべての操作は，*compare-and-swap*（CAS）や *Load-Link/Store-Conditional*（LL/SC）のような同期基本命令（synchronization primitives）によって原子的に行われるものとする．我々が評価に利用するプラットフォームである SPARC V9 では，この原子命令は SPARC にある CAS 命令によって実現される．

3.1.1　バッファフレームの構成

図 4 の左半分は，キャッシュされるエントリを表す Frame クラスである．Frame インスタンスは，1 つのキーと値，そのほかに制御変数 2 つを利用する．バッファ管理での利用で，K はページ識別子，V はページ自体を表現する．refcount インスタンスは，エントリの参照カウントを原子的に管理する．pinning インスタンスは，バッファ管理の外側でそのエントリが利用中かどうか判断するのに利用される．ここで，pinning が-1 のときを特別に，そのエントリを追放可能状態（evicted）と見なす（**図 2** 参照）．図 2 のように，*evicted* 状態と *pinned* 状態を単一の *pinning* 値により表すのは，2 つの状態を同時に更新する操作を同期基本命令によって，ロックすることなしに不可分に行うためである．

本論文で用いる pin や unpin 操作は，それぞれバッファ管理で一般的に利用される FIX-USE-UNFIX プロトコル[6] の FIX，UNFIX 操作に従う．pinning 値を原子的に変更するのが，pin/unpin と tryEvict/evictUnshared メソッドである．これら 4 つのメソッドにより，pinning 値は図 2 に示すように状態遷移する．pin メソッドが呼び出す addIfGT 関数は，不可分に，pinning 値が 0 以上の場合に限って pinning 値を 1 増分する．図 2 の gt 1

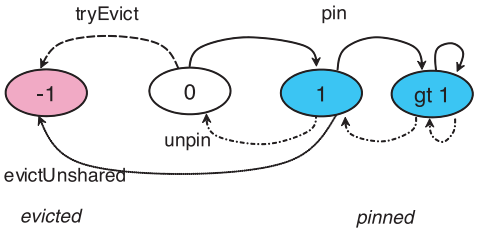


図 2　pinning インスタンスの状態遷移
Fig. 2　State machine of a pinning instance.

は，pinning 値が greater than 1 である，つまり 2 以上である状態を示す．この状態遷移にあるように，1 度 evict された pinning 状態は他の状態に遷移しない．unpin 操作が行われるとき，pinning 値は必ず 1 以上である．tryEvict が成功してエントリが evicted 状態となるのは，pinning 値が 0 である場合に限る．evictUnshared の呼び出しは，エントリの pinning 値が 1 の場合に限り，そのエントリを evicted 状態に遷移させる．一方，従来の GCLOCK を含む CLOCK の実装では，一般的にこのエントリ操作には排他制御を行っている．なお，本論文で GCLOCK を性能比較のベースラインに用いるとき，GCLOCK においても図 2 の技巧を利用する．

3.1.2　Bufferfix アルゴリズム

データベースにおいて，バッファフレームをページに固定する処理を慣例的に bufferfix[6] という．我々の Nb-GCLOCK において bufferfix 処理に相当するのが，これから説明する fixEntry 関数である．

図 4 の右半分の BufferCache クラスは，ノンブロッキングハッシュ表のインスタンス HASHTBL と，ページ置換リスト CLOCKBUF をメンバ変数として持つ．CLOCKBUF のアルゴリズムは，可読性のために，図 5 に分けて記述する．

BufferCache は，バッファフレームを取得する fixEntry とバッファフレームを割り当ててページを固定する addEntry の 2 つの関数を含む．addEntry 関数は，図 4 の 44 行目の条件判定が偽となる場合，つまり HASHTBL 中にキー *key* と関連付けられた evict されていないページが存在しない場合に呼ばれる．通常の GCLOCK（や CLOCK）においては，排他的な同期化が addEntry 関数の呼び出しにおいて必要である．*addEntry* のパージ処理（57 行目）で発生するバッファのフラッシュは，バッファから追い出されるページにダーティフラグが立っている場合に必要となる．このパージ処理で発生するディスク I/O は 1 章で

言及したように近代的な DBMS では極小化される．fixEntry メソッドの呼び出しは，任意のキーをバッファに固定し，固定した Frame の参照カウントを 1 増分する．fixEntry 処理は，定理 1 と系 1 を満たす．

定理 1. fixEntry メソッドによってすでに存在する *Frame* インスタンス F が返却されるとき，必ず，F の pinning 値が *1* 増分される．

定理 *1* の証明．すでに存在する Frame インスタンス F が *fixEntry* メソッドの呼び出しによって返却されるのは，**図 4** の 44 行目の条件式が真を返す，あるいは図 4 の 61 行目の条件が偽を返す場合に限る．これらの条件が成立するとき，F の pinning 値が *pin* メソッドの仕様に基づいて 1 増分されたあとである．□

系 1. 定理 *1* より，evictUnshared メソッド呼び出しによって成功裏に *evict* された *Frame* インスタンスは *Frame* クラスの外で決して利用されない．

3.1.3　CLOCK-sweep アルゴリズム

CLOCK において，バッファプールの中から置換対象のページを選択し置換する処理を *clock-sweep* という．**図 5** は，Nb-GCLOCK ページ置換ポリシを管理する ClockBuffer クラスを記述するものであり，4 つのメンバ変数を持つ．バッファプールであるフレーム配列とその原子的なアクセスを提供する POOL，バッファプールのフリースロットの数を表す原子的なカウンタ FREE，循環するクロックの指針としての原子的な CLOCKHAND，SIZE フィールドはバッファプールの容量を表す．

AtomicArray クラスは配列に対して原子的な操作を供給する．AtomicArray に対する CAS(i, expect, update) メソッド呼び出しは，添え字 i の指す配列要素の現在の値が期待する値（expect）に等しい場合，update の値に原子的に更新する．

ClockBuffer クラスは，単一の開始点として add メソッドを持つ．swap メソッドは既存のバッファページを GCLOCK ページ置換ポリシに基づいて新しいバッファページに置換するのに際して，add メソッドから呼び出される．swap メソッド呼び出しにおける 27 行目から 47 行目の for ループにおいて，Frame インスタンスの状態は**図 3** のように状態遷移する．この状態遷移に基づいて，いかなる並行アクセスも正しく処理される．moveClockHand メソッドは，原子的な add 命令列によって CLOCK の針を動かす．

ここで，*add* メソッドのアルゴリズムについて次の 2 つの定理が成り立つ．

定理 2. 図 *5* の *15* 行目で *FREE* インスタンスのデクリメント処理が成功したとき，必ず，

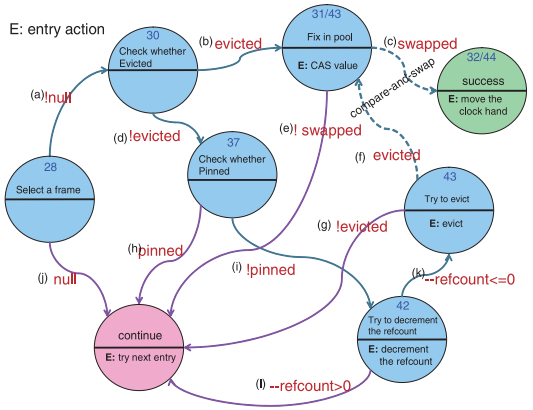


図 3　Clock-sweep における状態遷移
Fig. 3　State transitions in clock-sweep.

add メソッドの引数で与えられた *entry* がバッファプールの空きスロットに固定される．

定理 *2* の証明．15 行目で FREE インスタンスのデクリメント処理が成功する瞬間において，1 つ以上の空きスロットが存在することは明らかである．しかしながら，その成功の直後に，他のスレッドが 14 行目の *swap* に到達し，空きスロットを横取りする可能性がある．この場合に対処するため，28 行目で *swap* メソッドの呼び出しは空きスロットを利用することをせず，他のスレッドにその空きスロットを譲る．このようにして，*add* メソッドは，FREE インスタンスのデクリメントに成功するときはつねに，19 行目で引数の *entry* を空きスロットに固定する．□

定理 3. add メソッド呼び出しにおいて，同一の *Frame* インスタンスが異なる呼び出しに返却されることはない．

定理 *3* の証明．*add* メソッド呼び出しが非ヌル値を返すとき，*swap* メソッドが呼び出され，evict された Frame インスタンスが返却される．その evict されたインスタンスは，図 4 の 54 行目から 58 行目から明らかに，二度と置換リストで管理されない．

swap メソッドの for ループ中，27 行目と 47 行目の間で，Frame インスタンス e の状態は図 3 に示すように変化する．e が状態 '32/44' を通して返るとき，*compare-and-swap* 操

```
class Frame {
 1 K key; V value;
 2 AtomicInteger refcount = new AtomicInteger(1);
 3 AtomicInteger pinning = new AtomicInteger(1);
 4 Frame(K key, V value) {
 5  this.key = key;
 6  this.value = value;
 7 }
 8 V volatileGetValue() {
 9  memory fence for volatile load
10  return value;
11 }
12 boolean CASValue(V expect,V update) {
13  return CAS(value, expect, update);
14 }
15 void incrRC() {
16  refcount.increment();
17 }
18 boolean decrRC() {
19  return refcount.decrement();
20 }
21 boolean tryEvict() {
22  return pinning.CAS(0, -1);
23 }
24 void evictUnshared() {
25  pinning.CAS(1,-1);
26 }
27 boolean pinCount() {
28  return pinning.get();
29 }
30 boolean pin() {
31  return addIfGT(pinning,-1,1);
32 }
33 void unpin() {
34  pinning.decrement();
35 }
 } //end Frame
```

```
class BufferCache {
36 HashTable HASHTBL;
37 ClockBuffer CLOCKBUF;
38 BufferCache(int size) {
39  HASHTBL = new HashTable(size);
40  CLOCKBUF = new ClockBuffer(size);
41 }
42 Frame fixEntry(K key) {
43  Frame entry = HASHTBL.get(key);
44  if(entry != null && entry.pin()) {
45   entry.incrRC();
46   return entry;
47  } else {
48   return addEntry(key, null);
49  }
50 }
51 Frame addEntry(K key, V value) {
52  for(;;) {
53   Frame newEntry = new Frame(key, value);
54   Frame removed = CLOCKBUF.add(newEntry);
55   if(removed != null) {
56    if(HASHTBL.remove(removed.key, removed))
57     purge the removed page
58   }
59   Frame prevEntry = HASHTBL.putIfAbsent(key, newEntry);
60   if(prevEntry != null) {
61    if(!prevEntry.pin()) {
62     if(HASHTBL.replace(key,prevEntry,newEntry)) {
63      newEntry.setValue(prevEntry.getValue());
64      return newEntry;
65     }
66     newEntry.evictUnshared();
67     continue; //jump to line 53
68    }
69    newEntry.evictUnshared();
70    prevEntry.incrRC();
71    return prevEntry;
72   }
73   return newEntry;
74  }
75 } } //end BufferCache
```

図 4　バッファキャッシュの疑似コード
Fig. 4　Pseudo code of the buffer cache.

```
class ClockBuffer {
 1 AtomicArray POOL;
 2 AtomicInteger FREE;
 3 AtomicCounter CLOCKHAND = new AtomicCounter(0);
 4 int SIZE;
 5 ClockBuffer(int size) {
 6  this.POOL = new AtomicArray(size);
 7  this.FREE = new AtomicInteger(size);
 8  this.SIZE = size;
 9 }
10 Frame add(Frame entry) {
11  do {
12   int free = FREE.get();
13   if(free == 0)
14     return swap(entry);
15   if(FREE.CAS(free, free - 1))
16     break;
17  } while(true);
18  int idx = CLOCKHAND.get();
19  while(!POOL.CAS(idx%SIZE, null, entry))
20    idx++;
21  CLOCKHAND.increment();
22  return null;
23 }
23 Frame swap(Frame entry) {
24  int numpinning = 0;
25  int start = CLOCKHAND.get();
26  for(int i=start%SIZE;;i=(i+1)%SIZE) {
27   Frame e = POOL.get(i);
28   if(e == null) continue;
29   int pincount = e.pinCount();
30   if(pincount == -1) { // evicted?
31     if(POOL.CAS(i,e,entry)) {
32       moveClockHand(i, start);
33       return e;
34     }
35     continue;
36   }
37   if(pincount > 0) { // pinned?
38    if(++numpinning>=size)
39      yield this thread and allow others to execute
40    continue;
41   }
42   if(e.decrRC() <= 0) {
43    if(e.tryEvict() && POOL.CAS(i,e,entry)) {
44      moveClockHand(i, start);
45      return e;
46    }
47   }
48  } //end for
49 } //end swap
50 void moveClockHand(int curr, int start) {
51  int delta;
52  if(curr < start)
53    delta = curr + size - start + 1;
54  else
55    delta = curr - start + 1;
56  CLOCKHAND.add(delta);
57 }
} //end ClockBuffer
```

図 5　ClockBuffer の疑似コード
Fig. 5　Pseudo code of the ClockBuffer.

作はバッファプールから e を遷移（c）において削除する．したがって，同じ Frame インスタンスが異なるメソッド呼び出しの戻り値として返却されることがない．　□

3.2　アルゴリズムの正当性

線形化可能性とロックフリー性質は独立した関係であるが，どちらもノンブロッキングアルゴリズムにおいて重要な性質である[18]．本節では，提案手法である Nb-GCLOCK がロックフリーかつ線形化可能であることを示す．

3.2.1　線形化可能性

提案手法である Nb-GCLOCK が，線形化可能性の定義[21] を満たすことを証明する．線形化可能性はノンブロッキングの性質の 1 つであり，必要条件でも十分条件でもない（線形化可能でブロックされる実装も存在しうる）が，ある待機中の操作の呼び出しが他の待機中の操作の呼び出しの完了を待つ必要がないという性質を指す[18]．線形化可能性は合成可能である[18]．

線形化可能であることは，次を満たすことに等しい[18),22]．

- 線形化可能なオブジェクトへの各操作が，操作の開始と終了の間のある瞬間において，原子的に効果を及ぼす点（linearization point）を持つ．
- すべての操作は，linearization point において原子的に行われ，その呼び出し順序が保たれる挙動となる．

fixEntry のいかなる実行パスは，少なくとも 1 つの linearization point を持つ．*fixEntry* の実行パスは，46 行目，64 行目，71 行目，および 73 行目を通じて復帰する 4 通りであることは構文上明らかであるが，その各実行パスが通過する linearization point として次を選択する．

- 返却する Frame インスタンスの pin 処理を行う 44 行目
- 62 行目の replace 処理
- 返却する Frame インスタンスの pin 処理を行う 61 行目
- 59 行目の *putIfAbsent* 処理

ここで次のことが成り立つ．

補題 1. *44* 行目で *entry* が *null* かあるいはすでに追放済み（*evicted*）であれば，*pin* 操作は失敗する．さもなければ，*pin* 操作は成功し，実行は *45* 行目に遷る．ここで復帰する際に通過する *pin* 操作は *compare-and-swap* に基づく原子的な操作によるものであり，*linearization point* を持つ．

補題 2. *62* 行目で *prevEntry* が *HASHTBL* に存在しない場合，*replace* 操作は失敗する．さもなければ，*replace* 操作は成功し，実行は *63* 行目に遷る．ここで復帰する際に通過する *replace* はハッシュ表が提供する線形化可能な操作[⋆1]であり，*linearization point* を持つ．

補題 3. *61* 行目で *prevEntry* がすでに追放（*evict*）されている場合，*pin* 操作は失敗する．さもなければ，*pin* 操作は成功し，実行は *69* 行目に遷る．ここで復帰する際に通過する *pin* 操作は *compare-and-swap* に基づく原子的な操作によるものであり，*linearization point* を持つ．

補題 4. *59* 行目で *key* と関連付けられたエントリが *HASHTBL* 中に存在しない場合に限り，putIfAbsent は *null* を返し，実行が *73* 行目に遷る．ここで復帰する際に通過する *putIfAbsent* はハッシュ表が提供する線形化可能な操作であり，*linearization point* を持つ．

補題 1，補題 2，補題 3，および補題 4 より次の定理が成立する．

定理 4. 図 *4* の fixEntry アルゴリズムは線形化可能である．

3.2.2 Lock Freedom

Nb-GCLOCK のロックフリー性質（*lock-freedom*）は，他のスレッドが無限に進行（成功）し続けない限り，*fixEntry* 操作が有限ステップ数で終えることを意味する．つまり，ループ条件全般（ループ条件となりうる変数の集合 *LC*）のいずれかの値が変更されるまで *fixEntry* 操作の各ループが回り続ける場面が，他のスレッドが継続的に *LC* のいずれかの値を変更してその呼び出しを終える場合に限るならば，Nb-GCLOCK はロックフリーである．*LC* の状態を変更したスレッドは，その実行を終える必要がある．

ここでは，*fixEntry* からすべての無限ループする可能性があるループ箇所を洗い出し，各ループが他の復帰するスレッドによってのみ妨害されることを示すことによって，Nb-GCLOCK がロックフリーであることを示す．*fixEntry* メソッドに起因するループ箇所は次の 2 カ所である．

- *loop-in-addEntry* — 図 4 の 52–74 行目
- *loop-in-swap* — 図 5 の 26–48 行目

ここで，そのループに関して次の定理が成り立つ．

定理 5. loop-in-addEntry は，同じ key 値を引数に持つ別の *addEntry* メソッドの実行が並行して成功した場合に限り，実行をリトライする．

定理 *5* の証明．*loop-in-addEntry* が 67 行目に到達するのは，他の addEntry 呼び出しが成功したときのみである．なぜならば，62 行目の条件式が false を返すのは他の addEntry 呼び出しが成功したときに限るからである．*loop-in-addEntry* がループ処理をリトライするのは 67 行目を通してのみであるので，同じ *key* 値を引数に持つ別の addEntry メソッドの実行が並行して成功した場合に限り，*loop-in-addEntry* は実行をリトライする．　□

定理 6. loop-in-swap は，他の fixEntry 呼び出しが並行して成功しつづける場合に限り，無限にループする．

定理 *6* の証明．定理 2 と図 3 の *continue* に至る状態遷移から，*loop-in-swap* は，他の *fixEntry* 呼び出しが並行して成功しつづける場合に限り，無限にループする[⋆2]．　□

⋆1 利用するハッシュ表 20) が線形化可能であるとの前提をおく．ハッシュ表の正当な実装は linearization point を持ち，線形化可能である．線形化可能であることが証明されたノンブロッキングのハッシュ表[8),9)] でも代用可能である．

⋆2 各 CAS の成功は局所的な成功であり，各 CAS の失敗は他の CAS の成功を意味する．つまり，CAS が成功したときに状態機械の状態が進む．lock-freedom のノンブロッキング性質は，対象となる操作の制約から不可避なケース（たとえば，図 5 の 40 行目）でのリソーススタベーションの可能性を許容することに留意されたい．

```
1  Frame slot =
     PAGE_CACHE.fixEntry(pageId);
2  try {
3   V page = slot.volatileGetValue();
4   if(page == null) {
5     V update = read-in a page of the pageId from disk
6     slot.CASValue(null, update);
7   }
8   do application logic for the page
9  } finally {
10   slot.unpin();
11 }
```

図 6　提案手法におけるバッファの利用方法
Fig. 6　Usage of a buffer in our scheme.

定理 5 と定理 6 の成立によって，次の定理が導ける．

定理 7. 図 *4* の fixEntry 操作アルゴリズムはロックフリーである．

3.3　楽観的な並行 I/O

あるバッファフレームにページを固定する I/O 処理を複数のスレッドが同時に試みるという競合が発生することは，実際に起こりうる．慣例的なバッファ管理では，スレッドはバッファの固定に際して，他のスレッドが I/O 処理を開始して *io_in_progress* ロックを該当するフレームにすでに取得していた場合，先行スレッドによるロックが解放されるまで待機する[6]．Nb-GCLOCK は bufferfix 処理をノンブロッキングに実現するが，このページをバッファフレームに固定する *page-in* 処理は未解決の問題として取り残されたままに見える．

この *page-in* 処理をノンブロッキングに実現するため，我々のノンブロッキング手法は**図 6** に示すように楽観的に働く．既存のロックに基づく手法は，ページを読み込む前にロックを取得し，バッファフレームとページを関連付けた後にそのロックを解放する．*lseek* と *read* システムコールの組合せは排他制御を必要とすることもあり，ディスクからページを読み込む前にロックを取得するのは合理的といえるかもしれない．一方で，我々の手法では pread システムコールを利用して並行 I/O が発生することを厭わない．pread と pwrite は，同じファイルディスクリプタに対する複数のスレッドからの並列した I/O 要求を効率的に処理するためのシステムコールである．本論文の並行 I/O が有効に機能する可能性があるのは，次の 2 点による．

- 現在の二次記憶装置（SSD と安価な SATA ディスクを含む）が I/O 命令のキューイング機構を持ち，ディスクはヘッドの移動が最小限に済むような順序でアウトオブオーダ実行を行う．Serial ATA Revision 2.5 で定義された Native Command Queuing（NCQ）に対応する SATA ディスクが最大 32 段，SCSI ディスクは最大 255 段のコマンドをキューイングできる．
- I/O リクエストはただちにデバイスドライバに送られるのではなく，OS レベルでも I/O スケジューラによって I/O リクエストの並べ換えが行われる[23]．

表 1　SUN UltraSPARC T2 の仕様
Table 1　Specifications of Sun SPARC Enterprise T5120.

Operating System	Solaris 10 8/07
Core (Threads/Core)	8 (8)
Processor frequency	1.2 GHz
Main memory	16 GB
Disk	SAS (10,000 rpm)
L2 cache per core	4 M

これらの効果に対する複数のディスク装置や OS を用いた詳細な評価については，本論文の範疇ではないため割愛するが，4.1.2 項で，従来型のブロッキング I/O と提案手法の pread と CAS に基づく楽観的な並行 I/O について性能比較する．

4. 評価実験

Nb-GCLOCK の効果を評価するために，我々は Nb-GCLOCK と LRU，GCLOCK，および 2Q [12] を利用した場合の比較を行った．2Q [12] は完全バージョンを利用し，その *K1in* と *K1out* パラメータにはそれぞれ 20%と 30%を利用した．ブロッキング手法の排他制御には，効率的なスピンロックとして著名な指数バックオフ Test-and-Test-and-Set ロック[24]を利用した．

評価は，論文 12) にある例に従い，データベースのスキャンを含むワークロードを模擬する，Zipf 分布とスキャンを混ぜ合わせたワークロードを利用した．比較は，バッファヒット率とスループットに関して行った．我々の Nb-GCLOCK における置換ページの決定は，GCLOCK アルゴリズムと同様であるため，ヒット率は GCLOCK と同種の傾向を示す．そのため本論文における評価はスループットに重点をおく．

実験環境には，実機である**表 1** に示すスペックの Sun UltraSPARC T2 processor（Sun SPARC Enterprise T5120）を利用した．T5120 のプロセッサは，8 つの CPU コアを持ち，それぞれのコアが 8 スレッドを並行して扱うことができる．ゆえに，このプロセッサは 64 スレッドを並行して扱うことができる．我々の実装のランタイム環境には Sun JDK 1.6 を

利用した．

4.1　スキャンと Zipf 混在分布による実験

TPC-C のような OLTP ワークロードにおいて高い並行度のバッファのアクセスはランダム性が高くなる．ここでは，$\alpha = 0.5$ と $\alpha = 0.86$ を Zipf 分布[25] の入力に用いて，人工的に生成したワークロードを利用した評価を行う．この Zipf ワークロードは，独立参照モデル（Independent reference model）[26] に従う．独立参照モデルは，アクセス確率が過去の事象に依存しないという仮定をおくモデルであり，複数のプロセスやスレッドから利用されるキャッシュのヒット率解析において頻繁に利用される．

我々は，1,000 以上のクライアントから同時アクセスが行われるような状況[27] において，データベースの高いスケーラビリティを達成することを目標の 1 つとしている．こうした Web アプリケーションや情報システム一般において，Zipf 分布（特に 80/20 の分布）に近似するデータアクセスが頻繁に観察される（たとえば，論文 28)）ことが知られていることから，ページアクセスにも同様の傾向が現れると仮定して Zipf ワークロードを評価の対象とした．なお，論文 29) は，Zipf 分布に基づくデータアクセスが行われる環境におけるキャッシュヒット率に考察を与えている．

Zipf 分布では，N ページが存在するとき，アクセスされるページ番号が i かそれ未満となる確率は $(i/N)^{\alpha}$ である．$\alpha = 0.86$ は 80/20 の分布を生成し，$\alpha = 0.5$ の設定はより隔たりの少ない分布（おおよそ 45/20）を生成する．Zipf シュミレータを実行するとき，そのワークロードに適時スキャン処理が行われるように変更を加えた．我々の利用した混合ワークロードには，100 個の連続したページを要求するスキャン処理が 20%含まれる．ページサイズには 8 KB を共通して利用した．マルチユーザシナリオを模擬するため，ワークロードは複数のスレッドから並行して実行される．スレッド間でワークロードは共有せず，それぞれのスレッドが各自のシュミレータとワークロードを持つ．

今回用いたワークロードは読み込み専用であり，I/O 処理が支配的なケースについてはバッファヒット率による．書き込み処理を含むワークロードでは，図 4 の 57 行目において，削除されたページにダーティフラグが立っているケースでパージ処理が発生することがあるが，一般的に bufferfix 時のパージ処理は極小化される[6]．1 章で述べたように，I/O 処理がどれだけ支配的であるかを決める主要な要因の 1 つがディスク I/O の発生頻度であり，近代的なデータベース管理システムのバッファ管理におけるディスク I/O の発生頻度は，主にバッファヒット率に依存する．そこで，本論文ではバッファヒット率の変化に着目し，4,000,000 ページ（32 GB）からなるデータベースにおいて，バッファヒット率が比較的低い環境から高い環境について評価する．具体的には，バッファの容量が 4,096 ページスロットから 16,384 ページスロット，32,768 ページスロットまでを評価に利用した．メモリに読み込んだページに対するデータベースのタプルへのアクセスコストについては，ディスク I/O に対して相対的に小さいと考えられることからコストをゼロとして評価した．商用利用において，データベースサーバは 3 層構成にしてアプリケーションロジックと切り離されることが一般的であることから，アプリケーションロジックの CPU 消費については本論文では考慮しない．

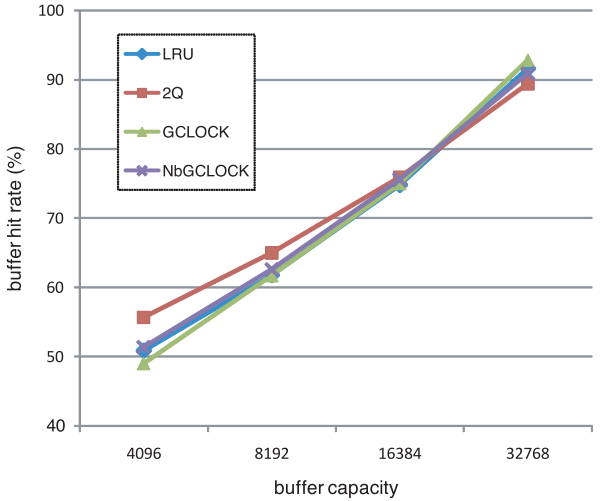


図 7　バッファ容量とバッファヒット率の関係（64 スレッド環境）
Fig. 7　Relationship between buffer capacity and buffer hit rate (64 threads).

4.1.1　バッファヒット率との関係

図 7 は，UltraSPARC T2 を利用して上記の 80/20 のワークロードを 64 スレッドから並行して走らせた場合のバッファ容量とバッファヒット率との関係を示す．バッファ容量が減少するほど，2Q は良好なバッファヒット率を示す．2Q はシーケンシャルスキャンに効果的であり，LRU（や CLOCK）に対してより良いバッファヒット率を提供するものと期待される[12] ことから，この結果は自然なものである．当然，十分なバッファ容量は図 7 に示すようにこれらの間の差を極小化する．さらに，高い並行度のバッファのアクセスはほとんどランダムアクセスを発生させる．2Q がバッファ容量が 32,768 のときにその優越を失っ

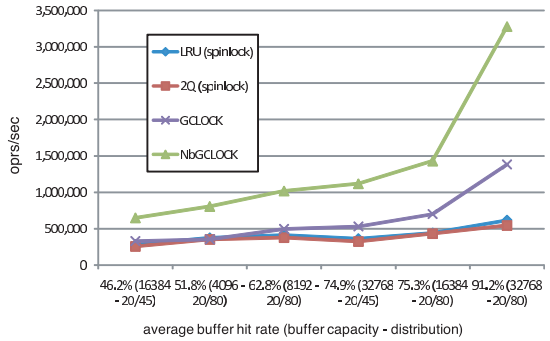


図 8　Zipf 分布とバッファ容量を変動させることによるスループットの変化
Fig. 8　Throughputs obtained when varying buffer capacity and workload distributions.

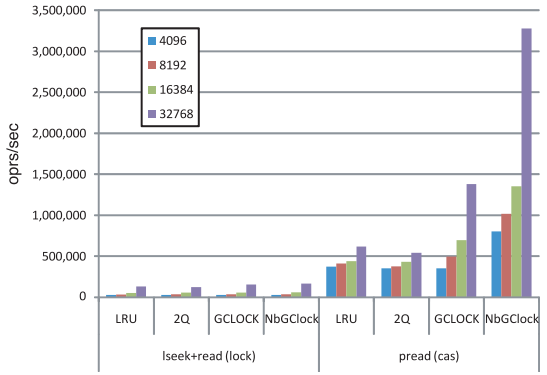


図 9　“lseek+read” と “pread” の性能比較
Fig. 9　Comparison between “lseek+read” and “pread”.

たのは，バッファ領域の見積りが不適切であったためと考えられる．2Q は 2 つの所定パラメータ（*K1in* と *K1out*）を持つが，最適な調整を静的に決定することが困難であることが欠点として知られている[30]．そのため，2Q のバッファヒット率はパラメータの設定の仕方によって期待できる上積みがある．

図 8 に，Zipf 分布とバッファ容量をそれぞれ変更し，その影響を測る実験で得られたスループットを示す．この結果から，GCLOCK とその派生である Nb-GCLOCK のスループットはバッファヒット率に主に依存すると見られるため，後続の実験ではバッファヒット率に対して得られる性能に焦点を当てる．

4.1.2　ブロッキング I/O と並行 I/O の性能比較

提案手法の特徴の 1 つに CAS 命令と pread システムコールによる楽観的な並行 I/O を行うことがあるが，楽観的な並行 I/O について，次の 2 点を明らかにする必要がある．

- クリティカルセクション（図 6 の 3〜7 行目）にどの程度の競合が発生するか．
- 極度に並行したワークロードに対してどちらの戦略が有効であるか．

図 9 に，UltraSPARC T2 上で 80/20 ワークロードを利用して提案手法と既存手法を比較した実験結果を示す．実験では，64 スレッドから 1 分間ワークロードを発行したため，理論的にとりうる最大の CPU 消費時間 CT_{max} は 3,840 秒（64 × 60 秒）である．図 9 で，それぞれの手法を “*pread*” と “*lseek+read*” と記述する．

最初の疑問への答えとして，我々は図 6 の 6 行目で発生する compare-and-swap 処理の失敗回数とその割合（Contention），および実際に消費した CPU 時間（CT と CT が CT_{max} に占める割合）を数え上げた．その結果を**表 2** に示す．提案手法の NbGCLOCK はノンブロッキングであるため，実行時間の 90%以上が CPU 処理に利用される．一方で従来のブロックされるページ置換手法は，同期処理で CPU にアイドル状態が発生するため CPU に費やした時間が低い．

スループットが異なるため単純に比較はできないが，Nb-GCLOCK は高スループット時（バッファ容量が 32,768 のとき）に GCLOCK に比べて，1 オペレーションあたり平均 1.44 倍の CPU 時間を消費している．ハードウェアスレッド数以上のスレッドが並行して実行され，かつ，CPU 利用率が高い状況下では，バッファ管理からみたアプリケーションロジック（他のデータベース処理）の CPU 消費次第でシステム全体としてのスループットが低下することが懸念される．ここで，Nb-GCLOCK が GCLOCK に対して余計に CPU サイクルを消費する可能性があるのは，バッファ置換時に他のバッファ置換処理スレッドの成功によってリトライが発生した場合であることに留意されたい．バッファ置換処理では，ページがダーティであれば I/O 処理が発生するため，他スレッドによる割込み（CPU 利用）の機会が存在する．バッファ管理以外で利用する CPU サイクルが問題となるケースでは，リトライが続いた場合に実行権限を一時放棄すること，スレッドプール等を利用してトランザクション処理スレッド数を適度に制限すること，ステージ型[31] のデータベース処理[32] を導入することが有効であると考えられる．

表 2 と図 9 から，我々のノンブロッキング手法は，楽観的な並行 I/O による冗長な I/O

表 2　pread より発生する競合
Table 2　Contentions generated by pread.

buffer capacity	LRU		2Q		GCLOCK		Nb-GCLOCK	
	Contention	CPU time (sec)	Contention	CPU time (sec)	Contention	CPU time (sec)	Contention	CPU time (sec)
4,096	10,619 (0.04%)	1,000 (26%)	5,342 (0.03%)	894.3 (23.3%)	10,177 (0.05%)	983.9 (25.6%)	88,396 (1.7%)	3,699.8 (96.3%)
8,192	5,650 (0.02%)	907.9 (23.6%)	3,329 (0.01%)	831.9 (21.7%)	7,993 (0.03%)	1,068.1 (27.8%)	65,034 (0.1%)	3,690.3 (96.1%)
16,384	2,477 (0.01%)	720.6 (18.8%)	2,077 (0.008%)	721.3 (18.8%)	6,157 (0.01%)	1,080.8 (28.1%)	53,028 (0.06%)	3,683 (95.9%)
32,768	733 (0.002%)	544.3 (14.2%)	998 (0.003%)	648.8 (16.9%)	5,538 (0.006%)	1,059.2 (27.6%)	60,447 (0.03%)	3,561.1 (92.7%)

処理が 2%近くの確率で発生するような場合においても有効に機能することが分かる．このことは，確かにある境界点まで提案手法がバッファ容量が増加するに従い，より効率的に機能することを示唆する．ここで，我々の実験が単一のディスクを利用したものであり，このアプローチは RAID 0 のような，より高いスループットのディスク設定において，より有効に機能する可能性があることに注意されたい．

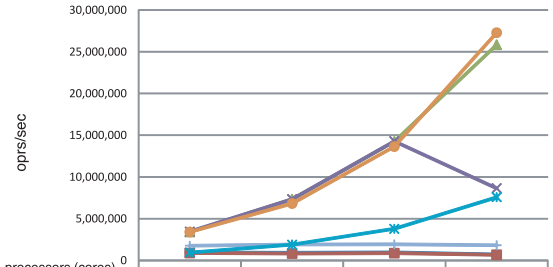
4.1.3　CPU のプロセッサ数に対して得られる性能のスケーラビリティ

CPU のプロセッサ数を変化させた場合の実験結果を示す．実験では，Solaris の *psradm* コマンドを利用してプロセッサの活性を制御した．最初の実験では，すべてのページがメモリ内に存在する場合のプロセッサ数に対するスケーラビリティを計測した．この実験は，提案するノンブロッキング手法をより高い I/O スループット環境に適用することを考慮に入れ，それぞれの手法のスケーラビリティの上限を測るためのものである．ここで，E$LRU と E$NbGClock は，それぞれ 8 プロセッサにおける性能を基にした LRU と Nb-GCLOCK が理想的に線形にスケールした場合のスケーラビリティを示す．

図 10 に示す実験結果は，NbGClock(stripe) と表記する我々のノンブロッキング手法が 64 プロセッサまでほぼ線形の CPU スケーラビリティを有することを示す．

図 5 において，compare-and-swap を利用した単純なカウンタである AtomicInteger を CLOCKHAND に利用した場合（NbGClock(atomic)）の指し示す性能が，32 プロセッサを超えて 64 プロセッサまでの範囲で減衰する理由は次のとおりである．

- 提案手法のナイーブな実装である NbGClock(atomic) は，CLOCKHAND に（局所的な）単一競合点がある．AtomicInteger は，カウンタの増減分に compare-and-swap インストラクションを利用するが，カウンタの更新に際してバスロックが発生してスケーラビリティを阻害する．
- 単純な共有カウンタは CPU 負荷の高い（書き込みバッファのフラッシュを誘発する）write 処理を含有するため，マルチプロセッサ環境における並列処理で競合し問題となる[33]．



processors (cores)	8 (1)	16 (2)	32 (4)	64 (8)
LRU	946267	917878	961255	695153
2Q	890992	819975	866009	662782
GCLOCK	1758605	1912000	1931268	1817748
NbGCLOCK(stripe)	3409819	7331722	14245524	25834449
NbGCLOCK(atomic)	3461203	7330622	14285105	8646410
E$LRU	946267	1892534	3785068	7570136
E$NbGCLOCK	3409819	6819638	13639276	27278552

図 10　すべてのページがメモリ内に存在する場合のプロセッサ数に対して得られる性能のスケーラビリティ
Fig. 10　Scalability to CPU processors when pages are resident in memory.

NbGClock（stripe）は，複数のストライプされたカウンタを利用することで上記の問題を解決したものであり，カウンタで発生する競合を低減させ，32 プロセッサを超えてスケールする．具体的には，単一のメモリロケーションでカウンタの値を管理せずに，書き込み時は複数のメモリロケーションによりカウンタの値を分散（stripe）させて管理し，読み込み時に複数のカウンタの値をまとめあげるアプローチを採用している．

上記の設定に加えて，ディスク I/O に *pread* システムコールを利用し，プロセッサ数を変化させたときの性能計測を行った．**図 11** に示す結果は，提案手法が 64 プロセッサまでのプロセッサ数の増加に対して少なくとも対数線形の性能が得られることを示すものである．この実験結果から，我々のノンブロッキングのバッファ管理手法は既存のブロッキング

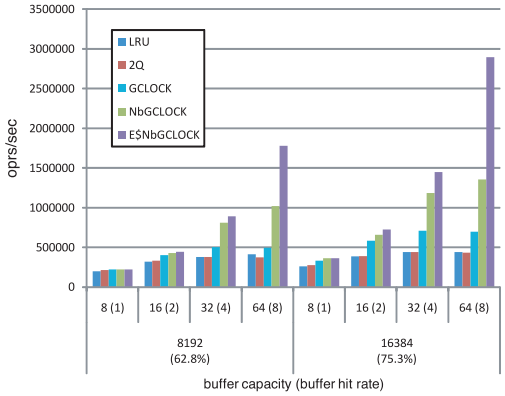


図 11　ディスク I/O に pread を利用した場合のプロセッサ数に対するスケーラビリティ
Fig. 11　Scalability to processors when using pread for disk I/O.

する手法と比較してよりスケーラブルであり，本論文で示した複数の状況（図 10 で示したような高い I/O スループット環境，および図 11 で示すようなディスク I/O が一応のバッファヒット率のもとに pread を利用して行われる環境）において有意な優位性を有すると，結論する．

4.2　x86-64 アーキテクチャにおける実験

4.1 節 の SUN UltraSPARC T2 を利用した実験で，我々のノンブロッキングのバッファ管理手法は有意な性能差を示した．しかしながら，他のアーキテクチャにおいて提案手法が同様に有効に働くかを明らかにする必要がある．この目的で，**表 3** に示す 2 つの異なる x86-64 アーキテクチャを利用した実験を行った．

実験では，すべてのページがメモリに存在する設定のもと（CPU プロセッサ数と同じ数である）8 個の並行アクセスがバッファ管理モジュールに発生するケースについて，Nb-GCLOCK を利用したノンブロッキングのバッファ管理手法と既存のロックに基づく手法の比較を行った．既存のロックに基づく手法が利用するページ置換アルゴリズムには LRU，2Q，GCLOCK を用い，80/20 のワークロードをそれぞれのアルゴリズムについて試行した．

図 12 が，その実験結果である．我々の手法 Nb-GCLOCK は既存手法である GCLOCK に対して，Xeon SMP アーキテクチャならびに Opteron ccNUMA アーキテクチャにおいてそれぞれ 3.84 倍，3.24 倍上回る性能を示した．なお，8 つの並行アクセスにおけるこのパフォーマンスの利得は，SUN UltraSPARC T2 において期待される数値（図 10 の 8 スレッド環境において GCLOCK の 1.94 倍）を超えるものである．この実験結果から，x86 アーキテクチャにおける中規模のマルチスレッド環境においても我々の提案するノンブロッキング手法の確かな有効性を確認した．

表 3　実験に利用した X86-64 計算機の仕様
Table 3　Specifications of each X86-64 machine.

Operating System	Linux 2.6.22 OpenSUSE 10.3	Linux 2.6.5 SuSE (SLES) 9
CPU model	Quad core Xeon E5420	Dual Core Opteron 880
Architecture	SMP	ccNUMA
Core (Chips)	8 (2)	8 (4)
Processor frequency	2.5 GHz	2.4 GHz
Main memory	8 GB	32 GB
Disk	SATA 2 (7,200 rpm, NCQ)	Ultra320 SCSI (10,000 rpm)
L2 cache per core	6 MB	1 MB

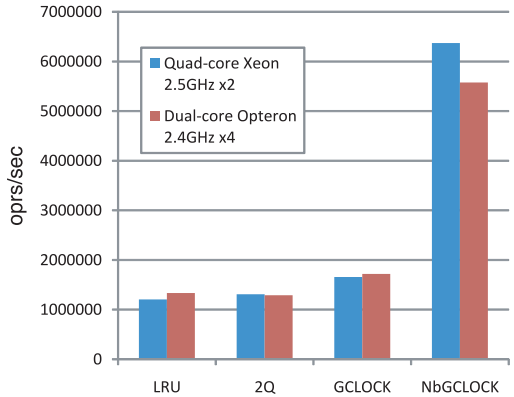


図 12　X86-64 アーキテクチャにおける実験（8 並行スレッド）
Fig. 12　Experiment on X86-64 architecture (8 threads).

5. 関 連 研 究

提案手法のベースとなる GCLOCK ページ置換アルゴリズムのバッファヒット率の詳細な評価は，論文 17) が詳しい．Nb-GCLOCK の置換されるページの決定方法は GCLOCK

と等しいため，GCLOCK と同様の性能傾向となる．

論文 34) で Tsuei らは，データベースサイズやバッファサイズ，CPU 数といった因子が性能（特にスループットとバッファヒット率）に与える影響について，SMP 環境で評価している．OLTP のベンチマークである TPC-C ワークロードにおける最適なバッファサイズについて議論しており，データサイズに対して約 10〜15%のバッファサイズを与えることで，おおむね 80%以上のバッファヒット率が得られることを述べている．図 8 に示した実験で，提案手法が明らかに有効となったのは約 7 割のバッファヒット率が得られた場合であった．このことから，扱うデータベースサイズの 1 割以上のメモリサイズをバッファ管理に確保できる場合は，提案手法の明らかな適用範囲となると考えられる．この要求は，64 ビット計算機の普及により扱えるアドレス空間が増えており，DRAM の高密度化と低価格化が進んでいることを考えると現実的に受け入れ可能な範囲と考える．

LRU の連結リストにおけるロックの競合を回避する目的で，ADABAS[35] はバッファプールを複数の区域に分割し，それぞれの区域を個別の LRU によって管理する．このアプローチはハッシュ値の分布に隔たりが生じた場合にバッファヒット率を著しく低下させる可能性があるが，LRU 連鎖の分割によってバッファヒット率がどのように変化するかについては議論されていない．

6. むすび

本論文では，ロックフリーの GCLOCK ページ置換アルゴリズムである Nb-GCLOCK を提案した．そして，Nb-GCLOCK と無待機のハッシュ表の組合せにより，要求されたページをバッファフレームに固定する処理をノンブロッキングに行う手法を開発した．実験結果により，独立参照モデルの一種である Zipf データ分布において提案手法が CPU 依存の処理と I/O バウンド処理の性能をいかに改善するかを明らかにし，既存手法が 16 プロセッサ以上のスケーラビリティを示さないのに対して，提案手法では 64 プロセッサまでほぼ線形の CPU スケーラビリティが得られることがあることを示した．また，CAS 命令と pread システムコールによる楽観的な並行 I/O が，現代のハードウェアにおいて有効に機能することがあることを示した．この楽観的な I/O という概念をバッファ管理に導入したのは，他の手法に先駆けるものである．マルチコアプロセッサと I/O がディスクと比較して高速な Solid State Drive（SSD）が普及過程に入っていることから，並行 I/O についてはより深い検討価値があると考える．

文献 6) で Gray らは，将来のデータベースシステムは大量のバッファプールを利用可能であるので，ページ置換をランダムに行うだろうという示唆を与えている．これは確かにあるケースに適用することができるが，ランダムのページ置換の挙動は予測不能であり，最悪のケースの挙動が問題となるようなシステムには適用することができない．我々の手法のようにロックフリーであることを保証することは，そのような現実の要求に応えるうえでも望ましいと考える．

謝辞　本研究の一部は，科学技術振興機構戦略的創造研究推進事業（CREST）「情報社会を支える新しい高性能情報処理技術」ならびに，日本学術振興会科学研究費補助金若手（B）（課題番号：19700094），日本学術振興会特別研究員奨励費の支援による．ここに記して謝意を表します．

参考文献

1) Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, *Dr. Dobb's Journal*, Vol.30, No.3 (2005).
2) Cieslewicz, J., Berry, J., Hendrickson, B. and Ross, K.A.: Realizing Parallelism in Database Operations: Insights from a Massively Multithreaded Architecture, *Proc. DaMoN* (2006).
3) Johnson, R., Pandis, I. and Ailamaki, A.: Critical Sections: Re-emerging Scalability Concerns for Database Storage Engines, *Proc. DaMoN* (2008).
4) 日本 OSS 推進フォーラム：2006 年度オープンソースソフトウェア（OSS）の性能・信頼性評価の成果 (2006). http://ossipedia.ipa.go.jp/capacity/CS0702010349/
5) Valois, J.D.: Lock-free Data Structures, Ph.D. Thesis, Rensselaer Polytechnic Institute, Troy, NY, USA (1996).
6) Gray, J. and Reuter, A.: *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1992).
7) Blasgen, M., Gray, J., Mitoma, M. and Price, T.: The Convoy Phenomenon, *SIGOPS Oper. Syst. Rev.*, Vol.13, No.2, pp.20–25 (1979).
8) Shalev, O. and Shavit, N.: Split-ordered lists: Lock-free extensible hash tables, *J. ACM*, Vol.53, No.3, pp.379–405 (2006).
9) Purcell, C. and Harris, T.: Non-blocking Hashtables with Open Addressing, pp.108–121 (2005).
10) Smith, A.J.: Sequentiality and Prefetching in Database Systems, *ACM Trans. Database Syst.*, Vol.3, No.3, pp.223–247 (1978).
11) Effelsberg, W. and Haerder, T.: Principles of Database Buffer Management, *ACM Trans. Database Syst.*, Vol.9, No.4, pp.560–595 (1984).
12) Johnson, T. and Shasha, D.: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, *Proc. VLDB*, pp.439–450 (1994).

13) Corbato, F.J.: A Paging Experiment with the Multics System, *In Honor of Philip M. Morse*, Feshbach and Ingard (Eds.), p.217, MIT Press, Cambridge, Mass (1969).
14) Ramakrishnan, R. and Gehrke, J.: *Database Management Systems*, 3rd ed., McGraw-Hill Science/Engineering/Math. (2002).
15) Bridge, W., Joshi, A., Keihl, M., Lahiri, T., Loaiza, J. and Macnaughton, N.: The Oracle Universal Server Buffer, *Proc. VLDB*, pp.590–594 (1997).
16) Yui, M., Miyazaki, J., Uemura, S. and Kato, H.: Efficient XML Storage based on DTM for Read-oriented Workloads, *Proc. IEEE International Workshop on Advanced Storage Systems* (*ADSS*), pp.559–564, IEEE CS Press (2007).
17) Nicola, V.F., Dan, A. and Dias, D.M.: Analysis of the generalized clock buffer replacement scheme for database transaction processing, *SIGMETRICS Perform. Eval. Rev.*, Vol.20, No.1, pp.35–46 (1992).
18) Herlihy, M. and Shavit, N.: *The Art of Multiprocessor Programming*, Morgan Kaufmann (2008).
19) Arnold, K., Gosling, J. and Holmes, D.: *The Java Programming Language, 4th Edition*, Addison-Wesley Professional (2005).
20) Click, C.: high-scale-lib. http://sourceforge.net/projects/high-scale-lib
21) Herlihy, M.P. and Wing, J.M.: Linearizability: A Correctness Condition for Concurrent Objects, *ACM Trans. Program. Lang. Syst.*, Vol.12, No.3, pp.463–492 (1990).
22) Herlihy, M.P. and Wing, J.M.: Axioms for Concurrent Objects, *POPL*, pp.13–26 (1987).
23) Axboe, J.: Linux Block IO — present and future, *Proc. Ottawa Linux Symposium*, p.51 (2004).
24) Anderson, T.E.: The Performance of Spin Lock Alternatives for Shared-Money Multiprocessors, *IEEE Trans. Parallel Distrib. Syst.*, Vol.1, No.1, pp.6–16 (1990).
25) Knuth, D.E.: *Art of Computer Programming*, Vol.3, Addison-Wesley Professional (1998).
26) Aven, O.I., Jr and Kogan, Y.A.: *Stochastic Analysis of Computer Storage*, Kluwer Academic Publishers (1987).
27) Kegel, D.: The C10K problem. http://www.kegel.com/c10k.html
28) Almeida, V., Bestavros, A., Crovella, M. and de Oliveira, A.: Characterizing reference locality in the WWW, *Proc. Parallel and Distributed Information Systems* (*DIS*), pp.92–107 (1996).
29) Vanichpun, S. and Makowski, A.M.: The Output of a Cache under the Independent Reference Model — Where did the Locality of Reference Go?, *SIGMETRICS Perform. Eval.*, Vol.32, No.1, pp.295–306 (2004).
30) Megiddo, N. and Modha, D.S.: ARC: A Self-Tuning, Low Overhead Replacement Cache, *Proc. FAST*, pp.115–130 (2003).
31) Welsh, M., Culler, D. and Brewer, E.: SEDA: An architecture for well-conditioned, scalable internet services, *SIGOPS Oper. Syst. Rev.*, Vol.35, No.5, pp.230–243 (2001).
32) Harizopoulos, S. and Ailamaki, A.: A Case for Staged Database Systems, *Proc. CIDR* (2003).
33) Herlihy, M., Lim, B.-H. and Shavit, N.: Scalable Concurrent Counting, *ACM Trans. Comput. Syst.*, Vol.13, No.4, pp.343–364 (1995).
34) Tsuei, T.-F., Packer, A.N. and Ko, K.-T.: Database buffer size investigation for OLTP workloads, *SIGMOD*, ACM, pp.112–122 (1997).
35) Schöning, H.: The ADABAS Buffer Pool Manager, *Proc. VLDB*, pp.675–679 (1998).

(平成 21 年 6 月 20 日受付)
(平成 21 年 10 月 10 日採録)

(担当編集委員　天笠 俊之)

油井　誠（正会員）

早稲田大学 IT 研究機構（IT バイオ研究所）客員研究員．日本学術振興会特別研究員 PD．博士（工学）．2003 年芝浦工業大学工学部工業経営学科卒業．同年（株）NEC 情報システムズ入社．グリッド・コンピューティングの研究開発に従事．2006 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．2008 年日本学術振興会特別研究員 DC2．2009 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．XML データベース，分散データベースの研究に従事．日本データベース学会会員．

宮崎　　純（正会員）

奈良先端科学技術大学院大学情報科学研究科准教授．1992 年東京工業大学工学部情報工学科卒業．1997 年北陸先端科学技術大学院大学情報科学研究科博士後期課程修了．博士（情報科学）．同大学助手を経て，2003 年より現職．2003〜2007 年科学技術振興機構さきがけ研究員（兼務）．2000〜2001 年テキサス大学アーリントン校客員研究員．高性能・高機能データ工学システムの研究に従事．電子情報通信学会，日本データベース学会，IEEE CS，ACM SIGMOD 各会員．

植村　俊亮（フェロー）

奈良産業大学情報学部情報学科教授．1964 年京都大学大学院工学研究科修士課程修了．同年電気試験所（現産業技術総合研究所）．マサチューセッツ工科大学電子システム研究所客員研究員，東京農工大学教授，奈良先端科学技術大学院大学教授を経て，2007 年から現職．データ工学，データベースシステムの研究に従事．工学博士．IEEE Fellow，電子情報通信学会フェロー．

加藤　博一（正会員）

奈良先端科学技術大学院大学情報科学研究科教授．博士（工学）．1986 年大阪大学基礎工学部制御工学科卒業．1988 年同大学大学院修士課程修了．1989 年同大学基礎工学部助手．1996 年講師．1998 年ワシントン大学客員研究員．1999 年広島市立大学情報科学部助教授．2003 年大阪大学大学院基礎工学研究科助教授を経て，2007 年より現職．拡張現実感，ヒューマンインタフェースの研究に従事．ヒューマンインタフェース学会，日本 VR 学会，ACM 等各会員．

山名　早人（正会員）

早稲田大学理工学術院教授．1993 年早稲田大学理工学研究科博士課程修了．博士（工学）．1993〜2000 年電子技術総合研究所．2000 年早稲田大学理工学部助教授．2004 年国立情報学研究所客員助教授．2005 年同研究所客員教授．2005 年早稲田大学理工学術院教授，現在に至る．IEEE，ACM，IEICE，DBSJ 各会員．