情報処理学会論文誌　データベース　Vol.5　No.3　126–140 (Sep. 2012)

インメモリDBへの適用に向けた実用的で安全な高並列オープンアドレスハッシュテーブル

新田 淳[1,2,a)]　石川 博[3]

受付日 2012年3月19日, 採録日 2012年6月26日

概要：任意の < キー，値 > ペアを管理対象としたオープンアドレッシング方式のハッシュテーブル操作を，高並列に行う実用的で安全な API と処理方式を提示する．マルチプロセッサ計算機上でコア数に比例した性能を出すインメモリ DBMS や KVS を実装するための汎用的な基本部品として使うことを想定している．ハッシュテーブル上に管理対象データの登録情報と参照カウンタを併せ持ち，それを一体操作することで，高並列データ処理に特有の複雑な競合タイミングを部品内部に局所化して隠蔽できるという特徴を持つ．この部品を利用することで，DBMS の様々な機能コンポーネントの開発者は，開発・保守の生産性を落とすことなく，かつ十分な品質を保持しながら，システム全体をマルチコアスケーラブルに性能向上させることが可能となる．ロックを用いた処理との比較を実施し，性能向上効果を確認した．

キーワード：インメモリ DBMS，ハッシュテーブル，マルチコア並列性

Practical and Safe Highly-concurrent Hash Table with Open Addressing for In-memory DBMS

Jun Nitta[1,2,a)]　Hiroshi Ishikawa[3]

Received: March 19, 2012, Accepted: June 26, 2012

Abstract: We present the highly-concurrent yet practical and safe hash table API and processing with open addressing that can handle arbitrary <key, value> pairs. One can implement a multicore-scalable in-memory DBMS or KVS using a common toolkit implementing this technique. It alleviates the burden of worrying about cumbersome race conditions that are characteristic of highly-concurrent algorithms from general DBMS developers by integrating hash tabe manipulation and reference counting in a single hash table entry. Encapsulating potentially dangerous timing problems within a common toolkit may enable development team to build a multicore-scalable DBMS safely without much affecting their productivity. Throughput measurement shows that this highly-concurrent version outperforms mutex-locked version under various conditions.

Keywords: in-memory DBMS, hash table, multicore concurrency

1. はじめに

本論文では，マルチコア計算機上の高性能インメモリデータ処理を安全に開発することを目的として，衝突をオープンアドレッシングによって解消するハッシュテーブルに対する実用的な高並列操作方式を提示する．この方式の主な適用先としては，マルチコアプロセッサを用いたサーバ上で商用の業務システムを稼働させる計算機利用環境での，プラットフォーム（ハイパーバイザや OS）やミドル層（DBMS や AP サーバなど）の内部処理でのプロセス・スレッド間の共用資源を管理するモジュール群が利用する，汎用的なデータ処理部品を想定している．ミドル層への応用としては，特にインメモリの DBMS や KVS（Key-Value Store）のキャッシュ管理機能，ロック管理機

[1] 静岡大学創造科学技術大学院自然科学系教育部
Shizuoka University, Graduate School of Science and Technology, Educational Division, Hamamatsu, Sizuoka 432–8011, Japan
[2] 株式会社日立製作所情報・通信システム社
Hitachi, Ltd., Information and Telecommunication Systems Company, Yokohama, Kanagawa 244–0817, Japan
[3] 静岡大学情報学部
Faculty of Informatics, Shizuoka University, Hamamatsu, Sizuoka 432–8011, Japan
[a)] jun.nitta.wg@hitachi.com

能，セッション管理機能，トランザクション管理機能，ログ管理機能などの実装に広く適用可能な共通部品を実現することをめざしている．

I/O 処理を除くミドル層の性能は，2000 年代初頭までは主にマイクロプロセッサの高速化とメモリの大容量化というハードウェアの進化と，それを利用するソフトウェア処理方式の改善によって向上してきており，細粒度の並列処理を採用する必然性は乏しかった．最近のプロセッサアーキテクチャの潮流変化（クロック高速化からマルチコア化へ）と，引き続くメモリ大容量化（OLTP 環境で利用する DB の多くをインメモリに配置することが可能になってきた）の 2 つの要因により，高並列なインメモリデータ処理方式の探求と応用が重要性を増してきている．

マルチコア環境では，メモリ上の共有オブジェクトの排他制御が大きな性能ボトルネックとなる．これに対応するため，既存ソフトウェアを改良して共有オブジェクトに対するロックを分割したり，ロック占有期間を短縮したりする努力が行われているが，近年の多コア計算機[*1]を有効に使いきるには不十分であり，より並列度の高いデータ処理方式をメインパス部分に組み入れることが避けて通れない．我々の目標の 1 つは，少なくとも 8 コア程度まで（できればそれ以上，16 コアあるいは 32 コアまで）性能がスケールするデータ管理製品の実用化である．

ロックフリー処理などのマルチコア環境向け高並列アルゴリズムを，DBMS の開発現場で様々な機能コンポーネントに組み入れようとする場合に，課題の 1 つとなるのが各種の競合タイミングを考慮しなければならない設計負荷の増大である．また，我々の経験からは，タイミングの絡む製品不良は時間が経ってもなかなか根絶できないものであることが分かっており[*2]，製品の保守・拡張性を担保するうえでも大きな負担となる可能性が高い．この問題を低減するには，複雑なタイミングが絡む処理部分を共通部品プログラム内に局所化することが有効であろう．その部品の開発・保守は，CAS（Compare and Swap）命令 [1] をはじめとするハードウェア逐次化命令とそれを利用した高並列データ処理に精通したごく一部のプログラマが担当し，他の多くの開発者（その多くはマルチコア向けプログラミングの知識をある程度持っているが必ずしもその専門家ではない）はその部品を利用することで，危険な競合状況をできるだけ意識せずに自分たちが担当する機能の実装に専念できるという状況を実現することが，開発・保守生産性と品質を犠牲にせずにマルチコアスケーラブルな高性能を達成するための近道であると，我々は考える．

本論文の提案方式は，そのような安全で高性能な共通部品を提供する試みの 1 つであり，DBMS の内部処理で広く使われるハッシュマップを対象とする．ハッシュの衝突を解決する方法としては，オープンアドレッシングを採用する．この方式では，ハッシュ値の衝突が発生した場合，当該エントリをテーブル外部のリスト構造に追加するのではなく，テーブル内の別の空き位置に格納する．ハッシュテーブルサイズを超えるエントリは格納できないが，メモリアクセスの高い局所性を期待できる．また，基本データ構造がポインタチェーンではなく配列であるので，高並列アルゴリズムのロジックが比較的単純ですむ．

提案方式の要点は，参照カウンタとハッシュテーブル操作を一体化したことにより，応用目的に適した API の提供と危険な競合タイミングの外部漏洩の回避を両立させて，安全性の高いパッケージングを実現したことである．ここでは，あるテーブル位置（ホーム位置）で衝突を起こすハッシュ値を持つエントリの集合（シノニムセット）を単位として競合の検出を行う．エントリ登録・追加位置のデータ構造を CAS 命令で操作することでエントリに対するアクセス競合を制御するとともに，ホーム位置のデータ構造を CAS 命令で操作することでシノニムセットに対する追加・削除の競合を検出し，必要であればリトライする．登録処理の一部が参照処理をブロックするタイミングがあるため，完全なロックフリーアルゴリズムではないが，全面的にロックに依存する方式と比較して優位な高並列性能を発揮する．

これまでにも，様々なデータ構造に対する高並列な処理方式が提案・報告されてきているが，その多くはアルゴリズムの正当性と性能向上効果を検証することに重点が置かれており，それを製品開発現場に取り入れる場面での生産性，拡張性，保守性といった観点から論じたものは，我々の知る限りではほとんど皆無である．次章で述べる CAS 命令を使った従来の高並列アルゴリズムの実装は，少なくとも我々の評価では，平均的プログラマから見て複雑で理解が難しいものである．スケーラブルなマルチコア向けプログラミングが多くの DBMS 開発者にとって避けられないものとなってきた現在，この複雑さをどのように克服しながら開発現場に取り入れてゆくかの具体的な対応策が求められている．

本論文では，まず 2 章で関連する高並列アルゴリズムの研究事例を示す．3 章では高並列アルゴリズムを製品開発現場へ持ち込むにあたっての課題を解説し，4 章で高並列アルゴリズムに求められる実用性の条件を明確にする．5 章では提案方式の外部インタフェースとそれを実現する内部データ構造とアルゴリズムを提示する．6 章では実用性と性能の評価を行い，7 章で総括する．

[*1] 現在市場で一般に調達可能なサーバ計算機は，最小構成で 4 コアからとなっているものが多い．

[*2] 筆者の 1 名が身近に体験した事例では，バージョンアップを繰り返しながら広く長く使われてきたソフトウェア製品で，20 年以上前に実装された部分のタイミングバグが新たに発覚して事故発生に至ったというものがある．これほどではなくても，古いバージョンから潜伏していたタイミングバグが，数年以上の時を経て発覚するということはそれほど珍しい事件ではない．

2. 関連研究

基本的なデータ構造に対する実用的な高並列操作アルゴリズムのこれまでの研究は，アドホックと汎用の 2 つのアプローチに大別される．なお，マルチプロセッサ向けプログラミング技術全般の解説と最近までの網羅的な参考文献リストについては，Herlihy らによる教科書 [2] が参考になる．この中では，オブストラクションフリー，ロックフリー，ウエイトフリーといった用語の正確な定義も与えられている．

アドホックアプローチでは，個別のデータ構造ごとに高並列操作アルゴリズムを開発する．ハードウェアで提供される CAS のようなシリアライズ命令を用いて，単純なカウンタ操作やスタック（LIFO）を実装する方式は，1970 年代の古くから知られており，OS 内部処理などで利用されてきた．Obermarck らは，スケジューリングキューのような複雑なデータ構造を扱う実用的な手法[*3]を 1980 年代に報告した [3]．基本的で応用範囲の広い片方向リンクドリストを扱う方式は，2002 年に Michael が簡潔で有効な方式を示した [4]．また，オープンアドレッシング方式のハッシュテーブルをロックフリーに操作する方法は 2005 年に Purcell らが報告している [5]．外部チェイニングで衝突を解決するハッシュテーブルは，片方向リンクドリストのアンカ部分のアレイと見なすことができ，Michael の方法が適用できる．これをさらに一歩進めて，Michael のリスト構造を基本に拡張可能ハッシュテーブルを実現する方法を，Shalev らが示した [6]．これらのアルゴリズムは，その API が提供するタイミング上のセマンティクスを利用者が十分に理解して利用する限りにおいて有効である．CAS 命令がこのようなデータ構造に対するノンブロッキングアルゴリズムを実装するための重要な基盤になることの理論的解明は，Herlihy が行った [7]．CAS などのシリアライズ命令は，10^2 程度のクロックサイクルを消費する処理時間のかかる命令であるため，アドホックアルゴリズムではその使用を必要最小限にとどめる工夫が，実行性能を確保するうえで必須である．

一方，汎用アプローチでは，任意の離散したメモリ領域をマルチスレッドから原子的に参照・更新する汎用のインタフェースと処理機構の提供をめざす．Herlihy によるトランザクショナルメモリの考え方が代表的である [8]．最近では，Harris，Fraser らが，基本的な CAS 命令を用いて，複数の離散メモリを CAS と同じセマンティクスで扱える MCAS，およびより汎用的な FSTM/OSTM のソフトウェアによる実装方法を示した [9], [10]．汎用機構であるため，どのようなデータ構造も扱うことができ，また単機能のモジュールを組み合わせてより複雑な処理を実現する合成能力も高いが，逆にリンクドリストやハッシュテーブルのような基本的なデータ構造を扱う場合は，余分な処理オーバヘッドがかかるという問題がある．Fraser らの手法では，実装上の制限で原則としてポインタ型のデータしか処理対象にできないため，データ構造体を要素単位に間接参照しながら扱うことになり，処理の複雑性とオーバヘッドが肥大化しやすい．汎用アプローチでは，アドホックアプローチと比べて，同じ機能を実現するのに多くの場合 2 倍以上の CAS 命令を使用する．

上記，CAS 命令を利用した処理方式群のほかに，現在利用可能なハードウェアではほとんどまたはまったく実装されていない DCAS のような機械命令を前提とした提案もある [11]．これらの命令が利用できれば，各種のロックフリーアルゴリズムを簡潔にプログラミングすることが可能となるが，実用性は低いといわざるをえない[*4]．

本論文で提案する方式は，ハッシュテーブルを対象としたアドホックアプローチに属するものであり，新田らがメインフレームのマルチプロセッサ化に対応するため 1989 年に公開したもの [12] を基にいくつかの改良を加えたものである．参考のため，改良前のデータ構造を付録 A.3 に示す．

3. 高並列処理を実用的に取り込むにあたっての問題点

従来提案されてきたようなロックフリーデータ処理方式を，製品プログラム開発現場に幅広く適用しようとした場合に発生する問題の一部を，実例を示しながら解説する．以下では，Purcell のハッシュテーブルと Michael のリンクドリストを取り上げるが，ここでの議論はそれ以外の場合（たとえば Herlihy の教科書に掲載されているすべてのロックフリーアルゴリズム）でも広く適用可能である．要点は，多くのロックフリーアルゴリズムは，ソースコード上で陽に把握することが難しいが，それを利用した処理方式の設計上は十分な考慮を必要とするタイミング上の注意点を持つことである．

3.1 対象とする課題の定義

まず，本論文で解こうとしている課題を改めて定義しておく．我々の課題は，DBMS の内部処理で多用されるハッシュテーブルを，マルチコア環境で操作するための高性能でかつ安全な汎用部品を提供する，ということである．「高

[*3] ここで Obermarck らの示したアセンブラプログラムは，後年ロックフリーアルゴリズムで重要なテクニックとして多用される他スレッドによる補助処理—Obermarck らはこれを「義務の伝達（passing an obligation）」と呼んでいる—の概念を内包している．このことを指摘した文献は我々の知る限りでは存在しない．

[*4] 高機能な機械命令があればソフトウェアを書くのが楽になるという根拠だけでは，ハードウェア開発者を納得させることはできないと我々は考える．その機械命令が，従来より効率的な高並列処理を実現するのに必要不可欠であるという確証が必要となるであろう．

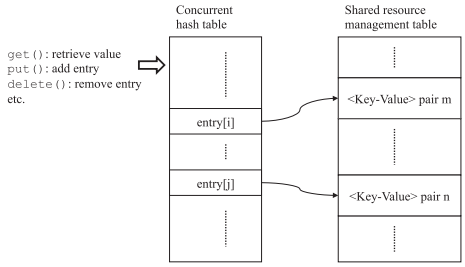

図 1　ハッシュテーブルによるメモリ上の共用資源の管理
Fig. 1　Management of shared memory resource by a hash table.

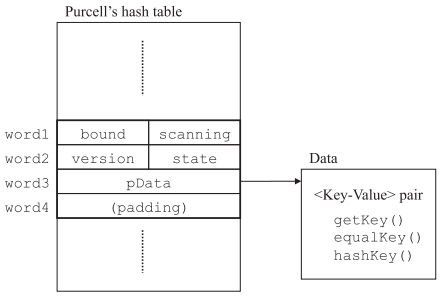

図 2　Purcell によるハッシュテーブルを拡張する試み
Fig. 2　Enhancement of a lock-free hash table by Purcell.

性能」とは，コア数に比例したスループット向上を意味する．「安全」とは，部品利用者（DBMS の各機能コンポーネント実装者）が，高並列処理に特有の危険な競合タイミングをできるだけ意識せずに正しく動くプログラムを書けるということを意味する．また，「汎用」である，すなわち応用ごとに個別ロジックを開発するのではなく，共通の部品として提供するという点にも留意が必要である[*5]．これらの特性を備えた部品は，従来と変わらない生産性と拡張・保守性を保ちつつ，マルチコアスケーラブルな DBMS を開発するための有効な助けとなると，我々は考える．

図 1 に，想定するハッシュテーブルの応用場面を示す．ハッシュテーブルは，DBMS 内部処理で用いるメモリ上の各種共用資源を検索するために利用される．たとえば，キャッシュ管理でのページディスクリプタ，ロック管理での資源テーブル，トランザクション管理でのトランザクションテーブル，セッション管理でのセッションテーブル，ログ管理でのログテーブルなどである．汎用部品としてのハッシュテーブルは，任意形式のキーと < キー，値 > ペアへのポインタを登録・検索・削除する機能を，安全かつ高並列に提供すればよい．

ここで注意が必要なことは，これら各種の共有資源管理データ（< キー，値 > ペア）は，部品利用者である各上位機能コンポーネントの持ち物であるということである．それらのデータ構造（配列か，リンクドリストかなど），そのライフサイクル（確保・初期化・解放など），使われ方（アクセスの逐次化が必要か共有可能かなど）は，上位が規定・制御するものであり，部品としてのハッシュテーブルはそれと干渉すべきでない．ハッシュテーブルは，これらの資源管理テーブルエントリをキーを用いて検索するための，外部の補助的なデータ構造でしかない．逐次化環境では，部品とその利用者の間で，機能責任範囲の分担が明確で独立性が高くなるように API セットが規定される．高並列環境では，それに加えて，タイミング上の責任範囲が明確で独立性が高くなるような API とセマンティクスを規定する必要がある．安全な部品化を実現するためのタイミング上の独立性という概念は，従来のロックフリーアルゴリズムの文献ではほとんど考慮されてこなかったものである．

3.2　失われた検索結果問題

Purcell らによるハッシュテーブルを，我々の目標とする汎用部品化することを考えてみよう．**図 2** に，ここでの議論に適するように原論文から多少変更したデータ構造を示す．Purcell 方式の特徴は，word1（オープンアドレッシングでのサーチ上限と一時状態フラグ）と word2（更新カウンタと状態フラグ）をそれぞれ CAS 命令で操作することで，ロックフリーなテーブル操作を実現していることである．ただし，そのままではキーだけしか扱えないため，キーフィールドを < キー，値 > ペアへのポインタ（pData）に変え，検索 API を $bool\ find(key)$ から $void\ *get(key)$ に拡張する必要がある．ここまでは，原論文のデータ構造とアルゴリズムにほぼ自明で簡単な変更を加えて実現できる．

問題は，異なるスレッドによる同一キーに対する検索と削除が競合した場合，$get()$ が正常終了してきた時点で，検索したキーを持つエントリがすでにハッシュテーブルから削除されているタイミングが発生することである．これをここでは「失われた検索結果」と呼ぶ[*6]．このタイミング上のセマンティクスは，我々の応用目的からすると不便であり，$get()$ で見つけたエントリが削除されないようハッシュテーブル上に固定する機能追加（しばしば $lock()$，$fix()$，$pin()$ などと呼ばれる）が必要となる．API は同じまま，$get()$ のタイミング上のセマンティクスを変えるだけであるが，この変更を実現するのは自明ではない．たとえば，状態フラグに「使用中」を追加してそれを CAS 命令で操作する案が考えられるが，全体のロックフリー性を壊さないよう整合性を保って処理を変更する簡単な解は見

[*5] 我々は，ソフトウェアトランザクショナルメモリのように低レベルなメモリアクセス層ではなく，ハッシュテーブルというデータ処理機能層での汎用性をめざしている．

[*6] 同様に追加と検索が競合すると，検索が NOT_FOUND でリターンしてきた時点で，求めるキーに対応するエントリがすでに追加されているタイミングも発生する．しかし，この場合，まだハッシュテーブルにキーが登録されていないと判断した検索スレッドが自分で新たに登録を行おうとすると，$put()$ が KEY_DUPLICATE でエラーリターンしてくるため，削除との競合に比べると危険性は小さい．また，追加と削除が競合すると，$put()$ が正常終了してきた時点ですでにそのキーが削除されてしまっている状況も起こりうる．

つかっていない[*7]．さらに，検索処理が CAS 命令を使うようになると性能特性も大きく変わってくるはずである．Purcell 自身も，辞書機能（キーだけではなく < キー，値 > ペアを操作すること）は今後の課題であるとしている．

上記のようなタイミング上のセマンティクスでも，応用によっては十分な場合がある．Laarman らは，検証モデル解析への応用を示した [13]．この応用では，エントリの削除機能が不要であるため，ロジックが非常に単純化できる．DBMS への応用では，Yui らがキャッシュ管理（バッファ管理）への適用を示した [14]．ここでは，削除処理そのものは必要であるが，その呼ばれ方がキャッシュ管理特有のパターンに限定されることにロジックが依存している．具体的には，エントリ削除の契機はページリプレースメントのときだけ発生し，かつ，リプレースメント処理はハッシュテーブルを経由せずに別データ構造（バッファプールアレイ）を利用して対象エントリをアクセスして状態を「追い出し済み」に変更するという仕組みになっている．ハッシュテーブルからの削除は，検索処理の延長でこの「追い出し済み」印の付いたエントリを検出したときにだけ呼ばれる．すなわち，ハッシュテーブル単体でなく，それを一部分として含むキャッシュ管理ロジック全体でタイミング上の整合性を確保している．いずれも，我々の目標とする，ハッシュテーブル操作レベルで自己完結した「安全性」と「汎用性」を十分に満足するものではない．ハッシュテーブルの各応用ごとに個別にタイミング問題を回避するロジックを組み立てていては，タイミング上の不良を作りこむ確率と保守の手間が，応用数に比例して増大してしまう．

ここでの事例が示すように，高並列アルゴリズムでは，わずかなデータ構造の変更，API シグネチャの変更，API のセマンティクスの変更を，ロックフリー性を保ったままで実現することが簡単にできる場合もあるが，非常に難しい（有効な解が簡単に見つからない）場合もある[*8]．これは，高並列アルゴリズムを比較するにあたっての注意点にもなる．たとえば，単に「ロックフリーなハッシュテーブル」という表現だけで同等なものと考えることは危険である．API とそのタイミング上のセマンティクスの詳細な部分まで注意深く考慮したうえでないと，同列での比較は意味がない．

3.3　残留参照問題

前節の「失われた検索結果」タイミングを削除スレッド側から見ると，*delete*() が正常終了してきた後も，ハッシュテーブルから削除した < キー，値 > ペアに対して検索処理スレッドがアクセスしてくる可能性が残っており[*9]，その記憶領域を自由に変更したり解放したりできないという状況を意味する．これを，削除処理スレッドから見て削除したエントリに「残留参照」が存在するとここでは表現する．残留参照は，削除処理と同一のキーに対する競合する *get*() 正常終了後の上位ユーザ処理によって引き起こされるだけでなく，異なるキーに対する *get*() 処理内部からも発生する場合がある．後者の例を，Michael のリンクドリストを用いて説明しよう．

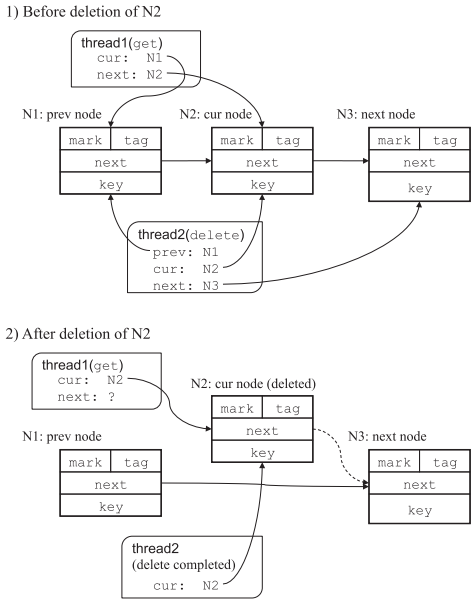

図 3　Michael によるリンクドリストでの残留参照
Fig. 3　Residual reference on a linked list by Michael.

図 3 は，検索（スレッド 1）と削除（スレッド 2）が競合した場合に起こりうる状況を示している．それぞれのノードは < キー，値 > ペアデータ構造全体を含むが，図中では簡単のためキーしか持たないように描いてある．図 3 で，検索処理も削除処理もリストを左から右に順番にたどっている．検索処理の最終目標はノード N3 であり，削除処理の対象はノード N2 とする．1) では，検索処理がノード N1 でキー一致とならず，next ポインタをローカル記憶域にロードしてノード N2 にアクセスしようとしている．この時点でスレッド 1 が何らかの原因によって中断し，その間 2) のように並行する削除処理が進み，ノード N2 をリストから削除したとしよう．スレッド 2 は，削除処理が正常リターンした後，ノード N2 の記憶領域を解放したい，もしくは任意の目的で再利用したいと考えるであろう．登録処理では呼び出し元がノードの記憶領域を確保してハッシュ処理に渡すので，これに対応して削除処理後にその記憶領域を解放しようとするのは，十分に考えられる処理シーク

[*7] 最悪，汎用的なソフトウェアトランザクショナルメモリの実装テクニックを使って解決することは可能なはずだが，その結果はオリジナルの処理方式とはロジックも性能も大きくかけ離れたものになる．
[*8] この問題は，高並列データ処理においてアドホックアプローチを採用する限り避けられない．アドホックアプローチで提供されるデータ処理機能は，非常に狭い利用条件においてだけしか有効ではない．
[*9] *get*() の戻り値が値渡しであればこの問題は起こらないが，我々の応用用途では，ほとんどすべての場合において共有資源を管理する構造体への参照渡しとなる．

エンスである．しかし，並行する検索処理（スレッド1）が，いつ処理を再開してこの記憶領域を参照するか分からない（ノードN2にスレッド1からの参照可能性が残留している）ので，それは不可能である．残留参照は，ソースコード上で明示的に把握することが難しく，我々の目標である「安全性」要件を充足しない．

Michaelの処理方式では，検索処理がノードN2の<mark, tag, next>フィールドを参照して期待した値と異なることを検出し，検索ループを最初からやり直すことで全体の整合性を保っている．このロジックが正しく動くには，ノードN2の記憶領域は解放されることがなく，かつ，<mark, tag, next>のデータ内容が，リストから外れた状態を含むいかなる場合でも定められた規約に従って更新される必要がある．このため，Michaelを含む従来方式では，ノードの記憶領域を自由に再利用可能な領域として扱わず，特定のメモリ管理ロジックを適用するという制限を導入している．1度ノードとして利用された記憶領域は，プログラム全体をリスタートしない限り他の用途には使えない．この問題は，Javaのようなガーベジコレクション機能を備えた言語処理環境では表面上見えにくくなる．ただしこの場合，処理方式のロックフリー性や実行性能を評価するには，ガーベジコレクタを含めて議論しなければならない．

「失われた検索結果」でも，「残留参照」でも，一般プログラマから見た場合の最大の問題点は，危険な競合が発生していることがソースコード上で明示的に分からないということである[*10]．この状況は，新規開発を担当するプログラマにとっても十分に問題であるが，完成したソースコードを他のプログラマが拡張・保守しようとする場合はさらに深刻となる．

4. 実用性条件

前章で述べたような問題を考慮し，本論文で重要視している「実用性」の定義をやや一般化した形で述べる．現実的なシステム開発環境で，単純なロック方式に比較して有意な性能向上を実現する高並列データ処理部品を取り込もうとする場合，その部品は以下の3つの条件を満たす必要がある；

条件1： 商用プロセッサで広く提供されている機械命令だけを使用すること

実際に利用可能なプログラムの提供が目的であるから，これは自明の要件である．本論文では現在広く利用されているプロセッサ群が提供するCAS相当命令（1語および2語長オペランド）を基本的なシリアライズ命令として利用する．直接CAS命令が提供されていないプロセッサでも，ほとんどの場合CAS相当機能を簡単に実装できる命令セットを備えている．従来の提案方式のうち，DCAS命令やハードウェアトランザクショナルメモリを前提とするものはこの条件に抵触する．なお，高並列データ処理プログラムでは，CAS命令を用いる部分だけでなく，通常のメモリアクセス（load/store）でもアクセス順序が大きな意味を持つ．このため，メモリアクセス順序の一貫性を保証する補助的な機械命令も利用する．これによりプログラムで記述した順にメモリアクセスが行われることが保証される．ほぼすべての商用プロセッサがこの機能を持つ命令を提供している．

条件2： ガーベジコレクタなどの高機能な実行環境を前提としないこと（OSカーネル内部と同程度の実行環境しか期待しない）

サーバ計算機で動作するOSやDBMSといった層のソフトウェアの内部処理で利用することが主目的であり，また，組み込み環境でも使えることをめざすため，前提となる実行環境の条件は最小限に抑える必要がある．OSや言語処理系のサービスは原則として利用しない．たとえば，ガーベジコレクション[*11]，実行時データ型管理，特定のプロセススケジューリングなどは前提としない．

条件3： 危険な競合タイミングをパッケージ内に隠蔽し，一般のミドル層プログラマが開発作業と保守作業において理解しやすいAPIとセマンティクスを提供すること

ここで，ミドル層プログラマとは，システムコールなど低レベルの機能を使ったプログラミングに業務層プログラマよりは精通しており，ロックを用いたマルチスレッドプログラミングの知識も持っているが，CAS命令とそれを利用した高並列処理の専門家ではない，という利用者像を意味する．我々の目的は，このようなミドル層プログラマに対して，高並列で安全な汎用データ処理部品（ここではハッシュマップ）を提供することである．危険なタイミングを内部に隠蔽し，かつ必要十分なデータ操作機能を上位に提供するAPIセットとセマンティクスを規定することが実用上の要点となるが，このことの重要性は，これまでほとんど指摘されてこなかった．

インメモリDBMS/KVSを主な応用先と考えた場合，汎用であるためには，上位モジュールが規定する任意の＜キー，値＞ペアの基本操作（登録・検索・削除）が可能なAPIセットを提供する必要がある．複数のスレッドが競合しながら共用リソースをアクセスする環境であるため，逐次環境と完全に同じAPIとセマンティクスを提供することはほぼ不可能であるが，安全であるためには，高並列処理に特有の危険な潜在タイミングを，利用者ができるだけ設計時に考慮しなくてすむようになっていなければならない．

[*10] たとえていえば，見えないグローバル変数が存在して，各所で危険な副作用を起こしているような状況である．または，タイミング上のスパゲティコードと表現してもいいだろう．

[*11] Herlihyらの教科書では，ガーベジコレクション機能のあるJavaを使用してアルゴリズムを記述している．これにより，多くのプログラムの記述は簡単になるが，ガーベジコレクタ自身の高並列性をどう実現するかは，別問題として残っている．

最低限，危険な競合が発生した場合はAPI呼び出し元がエラーコードなどで顕在的に認識できる必要がある．
この部品は，商用製品で用いられることを想定すると，長期にわたって保守・拡張が行われる状況でも安全に使える必要がある[*12]．大規模な製品プロジェクトでは，関連する開発者数は 10^1～10^2 人規模に及び，しばしば入れ替わりがあることも考慮しなければならない．このような状況に対応するためには，関連して修正を施す必要のある依存性の高いモジュール群を，できるだけ1つのパッケージ内にまとめることが望ましい．特に，高並列データ処理では，単純なソースコードの相互参照やデータフロー解析では補足しきれないタイミング上の依存関係を持つモジュール群をパッケージ化することと，危険なタイミングをAPI利用者から隠蔽することが，保守性を担保するうえで必須である．
具体的に避けるべき危険な潜在タイミングとしては；
条件 3-1：　APIユーザから見て「失われた検索結果」のタイミングを防ぐこと
キーを指定した検索が正常リターンしてきた場合，見つかったエントリが削除されないように固定する機能を，ハッシュテーブル処理側が提供しなければならない．
条件 3-2：　APIユーザから見て「残留参照」のタイミングを防ぐこと
削除処理が正常リターンしてきた場合，呼び出し元は，他スレッドの残留した参照を心配することなく，ハッシュテーブルから削除した＜キー，値＞ペアの記憶領域を自由に使えなければならない．
残留参照に関して，これまで提案された方式では，特定のメモリ管理ロジックと組み合わせて使うことで問題を回避しているが，そのようなメモリ管理モジュールはコアロジックと同じパッケージ内に同梱して，残留参照のタイミングがパッケージのAPI外部に漏洩しないようにする必要がある．この場合，APIセットは，登録・検索・削除に加えて，＜キー，値＞ペアの確保・解放機能を提供することになる[*13]．もしくは，本論文で提案するハッシュテーブルのように，特定のメモリ管理モジュールを必要としない処理方式を実装しなければならない．

高並列データ処理部品のAPIの水準について補足しておく．ソフトウェアトランザクショナルメモリは，（オーバヘッドの問題を考慮しないとして）高並列処理基盤として十分に汎用的であるが，一般のミドル層プログラマが直接使う部品としては，APIのレベルが低すぎる．メモリ領域のload/storeレベルではなく，ハッシュテーブルやリンクドリストといった汎用データ処理レベルで高並列性を提供することがより望ましい．一方，システム全体の保守性や拡張性をある程度犠牲にしても性能を重視しようとすると，安全な汎用部品を使わず，キャッシュ管理やセッション管理といった上位レイヤでのデータ処理まで含めた個別最適な高並列ロジックを採用する選択もあるだろう．Laarmanらや Yuiらがそのような例である．どのレイヤで高並列処理を実現するのが最適かは，それぞれのプログラムに求められる様々な要件を総合的に勘案して判断するしかなく，その判断には幅広い視野でのシステム設計のバランス感覚が要求される．我々は，社会の基盤システムで使われるようなDBMS製品を継続して安全に提供してゆくためには，難しい競合タイミングの問題を局所化した汎用部品を使うアプローチが最も望ましいと考えている．

本論文では，ハッシュテーブルを対象としているが，この実用性条件の議論は，他の高並列データ処理方式を製品レベルのソフトウェアに取り込む場合にも広く適用が可能である．

[*12] 我々の経験によると，商用のプラットフォーム/ミドル層ソフトウェア製品では，10年以上継続して拡張・保守が行われ，利用され続けることもしばしばである．
[*13] ＜キー，値＞ペアの持ち主である上位モジュールの様々な要求に応えることのできる汎用性と効率性を備える確保・解放機能を提供するのは，それだけで挑戦的な課題である．

5. データ構造とアルゴリズム

本章では，まず実用性条件3を満たすパッケージ全体のAPIを示し，次に実用性条件1，2，3を満たす主要部分の実装を解説する．

5.1 インタフェース

付録A.1に高並列ハッシュテーブル操作パッケージ（HCH: Highly Concurrent Hash）のAPIを示す．APIは大きく2つに分類される．この2種類のAPI群で，必要なハッシュテーブル操作はすべて網羅されている；

5.1.1 ハッシュテーブル管理 API：

ハッシュテーブル全体の管理を行う以下の3関数．これらはスレッドセーフでなく，逐次処理されることを上位が保障する必要がある．

- `newHCH()`　：ハッシュテーブル本体とその管理構造体（HCHディスクリプタ）のファクトリ．HCHはデータ要素の内部構造を意識しないため，上位からデータ要素中のキーを操作する関数（$getKey()$, $equalKey()$, $hashKey()$）を与えてもらいディスクリプタ内に記憶する．
- `freeHCH()`　：ハッシュテーブルとディスクリプタのデストラクタ
- `clearHCH()`　：ハッシュテーブルのイニシャライザ

5.1.2 データ操作 API：

ハッシュテーブルの操作を高並列に行う関数群．

- `HCH_get()`　：データ要素のキー一致検索（発見したデータ要素の参照カウンタを加算しポインタを返却）
- `HCH_put()`　：データ要素のハッシュテーブルへの登録（登録したデータ要素の参照カウンタを1に設定）

- HCH_release()：参照解除（対象データ要素の参照カウンタを減算）
- HCH_delete()　：データ要素のハッシュテーブルからの削除（対象データ要素の参照カウンタが 1 の場合のみ可能）
- HCHiterator()：逐次検索用イテレータの作成と初期化
- HCH_getnext()：データ要素の逐次検索（発見したデータ要素の参照カウンタを加算しポインタを返却するとともに新たな検索開始位置を設定）

検索処理は，与えられたキーと一致する固定された（他スレッドで削除されない）データ要素への参照を返す．この *get*() 機能を実装するためには，データ要素への何らかの参照管理が必須となる．そのために導入する参照カウンタは，後に示すように残留参照の外部漏洩を防止するための仕掛けにも利用される．この参照カウンタは，あくまでハッシュテーブル操作に限った範囲で意味を持つものであり，一般的な < キー，値 > ペアへの参照を管理するものではない．

検索で見つかったデータ要素を，呼び出し元がどのように利用するか，ハッシュテーブル部品はいっさい関知しない．ただし，検索処理スレッドは，必要なデータ処理を完了した後に *release*() を呼んで固定解除を行う必要がある．通常の並列ハッシュテーブル操作（java.util.concurrent.ConcurrentHashMap< K, V >[15] など）と比べて余計な手間となるが，自分で固定したものを解除するという処理セマンティクスは，検索したエントリが他スレッドによっていつ削除されるか分からないというタイミングを考慮するよりも，通常のプログラマにとって理解しやすいものであると我々は考える．

登録の場合，上位プログラムはまず *get*() ですでに重複キーが登録されていないことを確認して，データ要素を準備し，*put*() で登録する．*get*() がリターンしてきてから *put*() を呼ぶまでの間に競合する *put*()/*delete*() が発生したかどうかを確認するために，*get*() で戻される更新カウンタ値を *put*() に渡すインタフェースとしている[*14]．競合発生を示す *put*() のエラーリターンを検出した場合，利用者は *get*() からリトライすることになる．この要素的な *get*()/*put*() を用いて，自己完結型の *put*()（引数として更新カウンタ値をとらず，キー重複登録の場合はエラーリターンする）を提供することは簡単にできる．

5.2　操作アルゴリズム

高並列処理の中心となる検索・登録・削除処理について，付録に示す擬似コードを参照しながら解説する．擬似コードでは，半語/1 語/倍語長の符号なし整数のデータ型をそれぞれ UHALF/UWORD/UDOUBLE で表す．また，簡単のため HCH ディスクリプタからのポインタ参照（pHCH– >）を省略してある．

5.2.1　逐次化のプリミティブ命令

高並列処理を実現するためのハードウェアレベルの逐次化機能として CAS 命令を利用する．1 語長オペランドの CAS 動作は以下のように定義される；

```
bool CAS(UWORD *var, UWORD *old, const UWORD new)
do atomically {
    if (*var == *old) {*var = new; return true;}
    else {*old = *var; return false;}
}
```

CAS2() は同じ動作を倍語長オペランドに対して行う．本論文の評価で使用した Intel プロセッサの場合は，CMPXCHG/CMPXCHG16B 命令がほぼ同等機能を持つ [16]．

5.2.2　ハッシュテーブル構造

図 4 にハッシュテーブルと関連する構造体を示す．ハッシュテーブルエントリは 4 語長であり，それをプログラム中で 4 語長の構造体（ent[j]），倍語 × 2（dw1，dw2），または語 × 4（w1～w4）の変数として参照するためにいささか煩雑な定義が必要となる（擬似コード中では簡略化）．dw1 は CAS2，w2 と w3 と w4 は CAS による操作対象となる．w1～w3 は，この位置に登録されるデータ要素を管理するために用いられるのに対し，w4 はこの位置をホームとするシノニムセットを管理するためにある．

hash は，この位置に登録されているデータ要素のキーのハッシュ値であり，データ要素が登録されていない（空き）の場合は 0 となる．flags は一時的な状態を表し，ref はこの位置に登録されたデータ要素の参照カウンタである．ハッシュテーブルエントリに参照カウンタを内蔵し，それを他のデータ要素と一体操作することで実用性条件 3 が満たされる．

ハッシュ値が衝突した場合の登録位置の探索には，シノニムの偏在が起こりにくい 2 次剰余探索 [17] を採用しているが，高並列性には直接関係しないため他の方法を採用しても問題ない．具体的には，ハッシュ値 h を持つデー

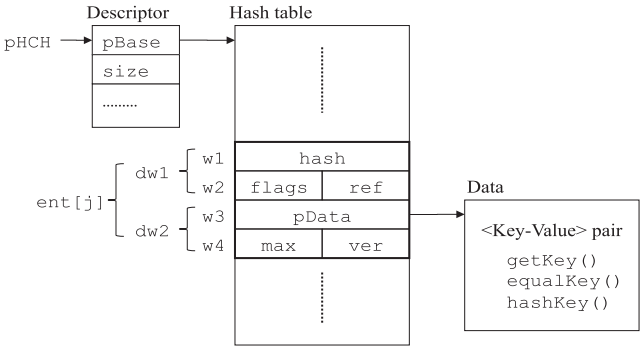

図 4　高並列ハッシュテーブルの構造
Fig. 4　Structure of a highly-concurrent hash table.

*14 CAS と並んで代表的なシリアライズ命令である LL/SC（Load Linked/Store Conditional）と似たセマンティクスといえる．

タ要素の i 番目の登録位置候補は，N をハッシュテーブルサイズとするとき[*15]次の関数 $pos()$ によって与えられる $(0 \leq i \leq N-1)$；

N が $Nmod4 = 3$ を満たす素数のとき

$$pos(h,i) = \begin{cases} (h+i^2)modN & : i <= N/2 \\ (h+i(N-i))modN & : i > N/2 \end{cases}$$

$pos(h,0)$ がホーム位置である．また，$pos(h,0)$ が同じ値となるキー値を持つデータ要素の集合がシノニムセットとなる．

pData はデータ要素 < キー，値 > ペアへのポインタである．HCH 処理では，限定されたキー操作を除いてデータ要素の内部構造をいっさい意識しない．また，残留参照がハッシュテーブル操作内部に隠蔽されているため，データ要素 < キー，値 > ペアそのものの記憶領域管理には任意のものを用いてよい．したがって，データ要素の記憶管理モジュールを部品パッケージ内に取り込む必要はない．

max は，この位置をホームとするシノニムセットをすべて探索するのに必要なループ回数の最大値を保持する．これは，キーが登録されていないことをテーブル全体を探索しないで判定するために用いる．ver は更新カウンタであり，この位置をホームとするシノニムセットにおいて，何らかの更新処理（登録または削除）が行われるたびに加算されてゆく．

5.2.3　処理方式の概要

処理方式の基本的な骨格は以下のようになる；

```
get/put/delete() {
    ホーム位置の ver を退避;
    for (pos(h,i) に従い順番に走査)
        if (目的のエントリ) {pData を参照/設定/消去して break};
    if (更新処理) {ホーム位置の ver を加算};
}
```

すべての処理は冒頭でホーム位置の更新カウンタ（ver）を記憶しておき，探索ループを抜けた時点での ver の値と比較する．一致しなければ処理中に競合する他の更新が発生したことを示し，リトライやエラーリターンを行う契機を与える．

5.2.4　キー一致検索

キー一致検索の擬似コードを付録 A.2.2 に示す．ホーム位置から始めてシノニムセットを走査し，キーが一致するデータ要素が見つかれば参照カウンタを増分してそのポインタを返す．検索ループは，ハッシュテーブルエントリ中のハッシュ値を比較し（行 10），一致すれば参照カウンタを CAS で更新して（行 16），改めてデータ要素をアクセスしてキーの一致を確認する（行 18），という手順で進む．もしキーが不一致であれば参照カウンタを減算してループを続行する（行 19〜行 23）．ハッシュ関数をうまく選べば，後半のキー不一致の可能性は十分小さくできる．このようにすることで，最後のキー一致判定以外ではデータ要素へのアクセスは起こらず，すべてハッシュテーブル内のデータ処理ですむため，キャッシュミスによる性能低下を最小限に抑えることができる．また，キー一致判定前に参照カウンタを増分しているので，競合する削除を失敗させ，「失われた検索結果」を防止することができる．ハッシュテーブルエントリ内にハッシュ値だけを保持するのは，キー本体を取り込むことによって，エントリサイズが不必要に大きくなることを防ぐためでもある[*16]．キー値発見または未発見の正常ケースでは，この検索ループを実行した時点での更新カウンタの値が返却される（行 25 と行 30）．行 11〜行 14 では，一時状態フラグを参照してエラーリターンしている．この部分は次に述べる登録処理によってブロックされる可能性があるため，この処理は純粋なロックフリー性要件を満たさない．

5.2.5　登録

登録処理は，付録 A.2.3 に示すロジックで実現できる．ハッシュテーブルをホーム位置から順に探索して，空きの位置を見つけたら CAS でその位置を仮に占拠する（行 12）．次に引数で与えられた更新カウンタ値とホーム位置の現時点での値を比較し，一致していれば更新カウンタを加算して（行 17〜行 19）処理完了とする（行 27）．この登録により探索最大数（max）が大きくなる場合は，ver と max をあわせて更新する．不一致の場合は，最新の $get()$ から現在までに他の競合する更新処理が成功したことを意味するため，仮登録を取り消してエラーリターンする（行 24）．このエラーを検出した上位プログラムは，再び $get()$ からやり直す．仮登録からホーム位置の更新カウンタを加算するまでの間にキー一致検索がこのエントリを参照すると，同一キーのデータ要素を 2 重登録するという矛盾が起こる可能性があるため，一時的に「登録処理中」のステータスフラグを立てる（行 13 と行 26）．フラグが立っている間は，このエントリを参照する検索処理はエラーリターンする．

5.2.6　削除

削除処理の擬似コードを付録 A.2.4 に示す．登録位置のデータをクリアし（行 11），ホーム位置の更新カウンタを加算する（行 20〜行 21）．もし競合を検知しても，同じデータ要素に対する削除処理との競合でなければ再試行を繰り返す．削除処理が少々複雑になるのは，サーチカウンタ（max）を正確に更新しようとするためである（行 15〜行

[*15] 2 次剰余探索法では，ハッシュテーブルサイズ N に制限が生じる．ここで示した Radke によるもののほかに $N = 2^m\,(m = 1, 2, \ldots)$ とする方法もある．

[*16] たとえば URL をキーとして使うと，その記憶域には最低でも 300 バイト程度必要となり，ハッシュテーブル操作のメモリ参照局所性を損なう．

20)．*delete*() の引数が，$*pKey$ ではなく，$*pData$ であることに注意してもらいたい．キーを引数にすると，削除対象データ要素を発見するために *get*() と同様の処理が必要となり CAS 発行回数が増えてしまうため，データ要素のアドレスを渡すインタフェースにしてある．

データ要素の削除は，参照カウンタ = 1 である場合だけ，すなわち削除処理スレッドだけが参照している場合に実行されるので，残留参照問題は発生しない．ハッシュテーブル操作とメモリ管理のための参照カウンタを統合したデータ構造とアルゴリズムにしたことで，残留参照が発生するタイミングを排除したわけである[*17]．削除処理は，他の処理をブロックすることはない．

登録・削除処理が，同一シノニムセットに対する連続した他の登録・削除処理と競合した場合，スレッドのスケジューリング具合いによっては，長時間リトライを繰返す場合がありうる．すなわち，登録・削除処理が有限時間内に完了することは保証されていない．

5.2.7　その他の処理

参照解除（*release*()）は，指定されたデータ要素に対応するハッシュテーブルエントリを探して参照カウンタを CAS で減算するだけの簡単な操作であるため，擬似コードを省略する．また，キー一致検索だけでなく，順次検索（*getnext*()）も可能である．ハッシュテーブルを指定された位置から順次走査してゆき，最初に見つかった空きでないエントリの参照カウンタを加算してデータポインタを返せばよい．キー一致検索の簡単な変形で実装できるのでこの疑似コードも省略する．

5.3　実装上の注意事項

擬似コードに関する実装上のいくつかの注意点をあげておく；

(1) 1 語長は 64 ビットを想定しているが，32 ビットでも問題なく動作する．

(2) ハッシュテーブル上のハッシュ値（hash）のデータ型は符号なし整数であるが，0 を未登録の意味に使っているので，*hashKey*() 関数の戻り値は符号付の正の整数に限定する必要がある．

(3) ハッシュテーブル上の参照カウンタ（ref）とサーチカウンタ（max）はどちらも半語長の符号なし整数であり，上限が $2^{32}-1$（64 ビット環境）または $2^{16}-1$（32 ビット環境）となる．実用上はほとんど問題にならないであろう（特に 64 ビット環境）．擬似コード上では，簡略のため上限値のチェック処理を省略してある．

(4) ハッシュテーブル上の更新カウンタ（ver）も同じく半語長の符号なし整数であるが，こちらは上限を超えてラップアラウンドしてもよい．API 引数中の更新カウンタ値は 1 語長の符号付整数（ver_t）となっており，負の値にリターンコードの意味を持たせている．

(5) ハッシュテーブルに対するメモリアクセスでは，CAS 部分だけでなく通常の load/store 順序も意味を持つ．このため，メモリアクセス順序制御命令[*18]を適宜挿入する必要がある．擬似コードでは，登録処理の行 14 と行 21 に入っている．この命令は CAS 同様クロックサイクルを大きく消費するので，使用を必要最小限にとどめる必要がある．

6.　評価

6.1　実用性条件の充足評価

提案方式が，4 章に示した実用性条件をどのように満たすかを評価する；

条件 1：　商用プロセッサで広く提供されている機械命令だけを使用すること

擬似コードから明らかなように，本実装ではシリアライズの手段として CAS/CAS2（およびメモリフェンス命令）だけを利用しており，実際の IA サーバ機上で動作するプログラムである．したがって条件 1 は満足される．

条件 2：　ガーベジコレクタなどの高機能な実行環境を前提としないこと

本実装では，OS や言語処理系のサービスはいっさいコールしておらず，またガーベジコレクションも前提としないため条件 2 を満たす．厳密にいうと，擬似コードを掲載していない *newHCH*()/*freeHCH*() 内で *malloc*()/*free*() 相当の実行時ヒープ割当て/解放機能を利用しているが，これも必要であれば静的割当てなど代替手段への簡単な置き換えが可能であり，条件 2 を破る程度の問題ではない．

条件 3：　危険な競合タイミングをパッケージ内に隠蔽し，一般のミドル層プログラマが開発作業と保守作業において理解しやすい API とセマンティクスを提供すること

API は，インメモリ DBMS/KVS の実装に必要なハッシュテーブル基本操作（登録・検索・削除）を網羅しており，検索 API はテーブル上に固定されたデータ要素への参照を返す機能を持つ．また，ハッシュテーブル操作と参照カウンタ管理を一体化したことで，危険な残留参照タイミングが API 利用者に潜伏して漏洩することはない．避けられない競合が発生した場合は，API のリターンコードで顕在化される．したがって，条件 3 は満たされている．この条件を満足するセマンティクスを備えた API と処理方式を示したのは，おそらく本論文が最初である．

API に関しては，単純な検索と登録に加えて，ハッシュテー

[*17] その代わりに，参照カウンタ ≥ 2 である，すなわち他スレッドによる参照が併存しているうちは，*delete*() コールはエラーリターンする（危険な競合タイミングが顕在化される）．*delete*() を成功させるタイミングで呼ぶのは上位の部品利用者の責任となる．また，参照カウンタ = 1 の場合だけ削除が可能であるという条件は，いわゆる ABA 問題を回避する役目も果たしている．ABA 問題については Herlihy ら [2] を参照のこと．

[*18] Intel プロセッサの場合だと MFENS/SFENCE/RFENCE 命令 [16] がこの機能を持つ．

ブル操作レイヤでの原子的な比較更新（read-modify-write）操作を提供することがより有用であるという意見もある．このような API を使えば，上位プログラムが，＜キー，値＞ペアをロックフリーに更新する処理を実装できる可能性が生まれる．単純なアトミック read/write 操作だけではこれは不可能である．我々の *get*()/*put*() API は，LL/SC 命令に似たセマンティクスを持っているので，read-modify-write 操作を提供する新しい CAS 命令的な API を設けるのではなく，既存の *get*()/*put*() を拡張することで対応できる可能性がある．

6.2　性能実測

提案方式の効果を検証するために，マルチコア IA サーバ（Xeon[*19] X5570：4 コア × 2 ソケット）を用いた実験を行った．centOS 5.4/gcc 4.1.2 の pthread ライブラリの mutex ロックを用いてシリアライズするプログラムと，本提案方式を Intel 64 アーキテクチャの CMPXCHG/CMPXCHG16B 命令を用いて実装したプログラムを比較している[*20]．OS スケジューラの影響を可能な限り排除してアルゴリズム本体の性能特性を得るため，各コアに 1 スレッドを CPU アフィニティ付きで割り当てて 1 コア〜8 コアで実行し，一定回数（測定条件により 75,000 回〜1,000,000 回）のハッシュテーブル操作を行うために要する時間を計測した．スレッドは，できるだけ 2 つのプロセッサのコアに均等に分配されるように割り当てている．インメモリ DBMS では，I/O 待ちが発生しない（または従来のディスクベースの処理に比べて I/O 待ちの発生が大幅に少ない）ため，コア数以上のスレッドを実行するのは OS オーバヘッドが増えるだけであまり意味がなく，この測定条件は実際の利用状況に近いと考える．ハッシュテーブル操作は，検索（*get*()[OK] とそれに対応する *release*()），登録（*get*()[NOTFOUND] とそれに続く *put*()），削除（*delete*()）の 3 種類をそれぞれ 1 操作として数え，2:1:1 の比率でランダムに混合して実行させている[*21]．使用したハッシュテーブルのサイズは，8219 エントリである．ハッシュ値の衝突や各種処理が競合する機会が多くなるように，キー値としては，64 ビットでランダムに発生させたものを下位 16 ビットに制限して使用している．

実際の利用環境では，ハッシュテーブル操作は全体の処理の一部分でしかない．この状況を擬似的に実現するため，各実行スレッドにローカルなメモリ処理を加え，ハッシュテーブル操作にかかる CPU 時間が全体の処理時間に占める比率を変えられるようにした．どの程度の比率が現実的であるか公開されている情報は多くないが，Harizopoulos らは，TPC-C [18] のような比較的単純な OLTP 処理において，ラッチ処理が全体の 14.2%であると報告している [19]．この「ラッチ処理」が具体的にどのような処理を含むのか詳細不明であるが，1 つの参考にはなる．我々の経験からは，DBMS の各種内部処理で利用可能なところにすべて今回のハッシュテーブルを適用した場合，OLTP ワークロードではその処理比率は全体の数%と見積もるのが妥当だと推定する[*22]．なお，オープンアドレッシング方式のハッシュテーブル操作を mutex ロックで行う場合，テーブル全体に対するロックが必要なことを注意しておく．登録操作において，ホーム位置と実際の登録位置の 2 カ所のエントリをこの順番にロックする必要があるため，エントリごとにロックを設ける方式ではデッドロックが発生するためである．

図 5，**図 6**，**図 7** に測定結果を示す．各グラフは，コア数に対するハッシュテーブル操作数の相対スループット（1 コアでの操作数を基準）を表しており，コア数に対してスループットが比例して増加することが理想である．1 コアでの高並列版とロック版のスループットの絶対値の差はほとんど無視できる程度に小さい．ハッシュテーブル操作が全体の約 1%の場合は（図 5），高並列版が 8 コアまでほぼコア数に比例してスループットが増加するのに対し，ロック版では 3 コアまでしか性能が伸びない．ハッシュ処理比率が 5%になると（図 6），高並列版でも 5 コア以上で伸びが鈍ってくる．現実的とはいえないが参考のため測定したハッシュ処理比率が 90%以上という極端な高競合環境では（図 7），高並列版もロック版もともにスループットは 1 コアよりも低下するが，高並列版の方が低下傾向が緩やかで

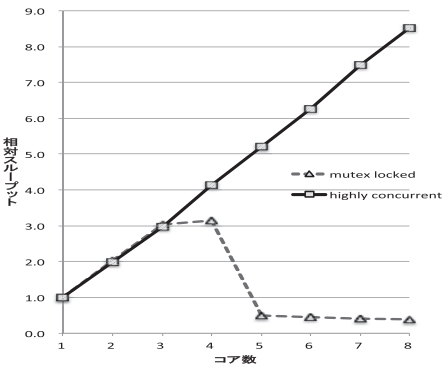

図 5　性能実測データ（ハッシュ処理比率 ≒ 1%）

Fig. 5　Performance measurement (hash processing ratio ≒ 1%).

[*19] Intel Xeon は，アメリカ合衆国およびその他の国における Intel Corporation の商標です．

[*20] 最新のロックフリーハッシュテーブル操作アルゴリズムとの比較ができれば望ましいが，ロックフリー性を失わずかつオリジナルとかけ離れない範囲で我々と同じ API とセマンティクスを満たすように改変する良い方法がまだ見つからないため今後の課題とする．

[*21] この検索と更新処理の比率は，我々の経験に基づいて任意に選んだものであり，特定の処理モデルを前提とした定量的評価から決定したものではない．実際には，もう少し検索処理の比率が高いと考えているが，更新処理比率が高いほうが競合が発生しやすくより厳しい実験条件となるため，この比率を採用している．

[*22] 厳密な値ではなく，既存製品のソースコード上からおおまかに机上で見積もったものである．桁数のレベルでは有効だと考える．

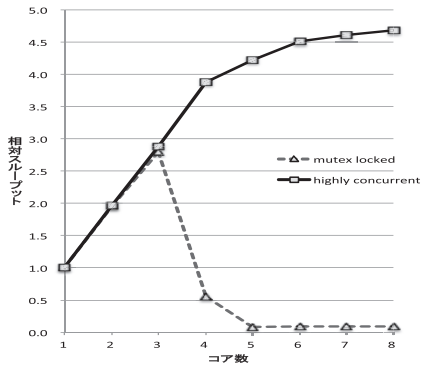

図 6　性能実測データ（ハッシュ処理比率 ≒ 5%）

Fig. 6　Performance measurement (hash processing ratio ≒ 5%).

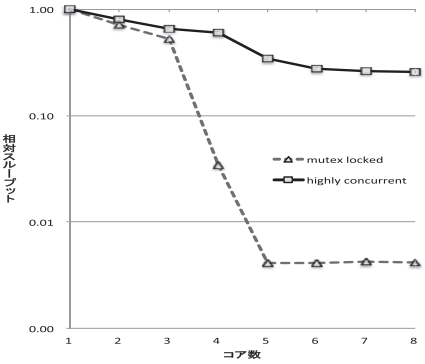

図 7　性能実測データ（ハッシュ処理比率 ≧ 90%）

Fig. 7　Performance measurement (hash processing ratio ≧ 90%).

ある．

一般に並列処理アルゴリズムの性能は，他の条件が同じであれば，全体の処理時間に対する逐次化される処理時間の比率に左右される．したがって，ハッシュテーブル処理全体が逐次化されるロック版に比べて，CAS 命令実行部分だけがハードウェア的に逐次化される[*23]高並列版の方が，コア数に対してスケーラブルに性能が伸びる傾向が高いという結果は，期待に沿うものである．具体的に，どの程度の並列度まで性能が伸び，またその後どの程度性能が劣化するかの詳細な振舞いは，測定に利用する環境（プロセッサ，メモリシステム，OS，pthread を実装する C ライブラリ）に依存する．このため，図 5 から図 7 に示した結果は，上述の実測条件に特有のものであり，他の環境では異なったものになる可能性が高いと考えられる．特に，擬似コードに示したプログラムがすべてユーザモードで実行される高並列版に比べて，*pthread_mutex_lock/unlock*() の C ライブラリ関数を利用するロック版は，測定するプラットフォームの環境に依存する度合いが大きい．ロック版における 4 コア以上でのスループットの極端な悪化は，実験環境での pthread の実装に依存したものであると推測される．一方，提案した高並列方式は，現実的な負荷のもとで少なくとも 8 コアまでは十分な性能向上が期待できる程度に逐次化部分を小さくすることに成功しているといえよう．

[*23] 厳密にいうと，競合による CAS 命令失敗時のリトライループ処理に起因する CPU 時間増大も性能劣化に寄与する．ただし，高並列版における CAS 命令の競合確率は，ロック版における mutex ロックの競合確率に比べると大幅に小さい．

7. まとめと今後の方針

オープンアドレッシングで衝突を解決するハッシュテーブルを高並列かつ安全に操作するアドホックな処理方式を提案した．本方式は，実用に必要十分な API と機能を備え，商用プロセッサで広く利用可能な機械命令だけを用い，実行環境の前提を最低限に抑え，また危険なタイミングを隠蔽した安全なパッケージングになっている．このため，インメモリ DBMS/KVS の実装において開発や拡張・保守の生産性を落とさずに高並列処理を取り入れるための汎用データ処理部品として利用できる可能性が高い．実際の利用条件に近い場合，マルチコア計算機上でコア数に比例して性能が向上する効果が確認できた．

今後計算機環境が確保できれば，8 コア以上でどこまで効果があるかの検証を行いたい．また，参照カウンティングとデータ要素操作を一体化することにより残留参照を回避する技法が他のロックフリーアルゴリズムに適用可能かどうか試みたい．特に，Purcell のようなロックフリーハッシュと我々の安全な高並列ハッシュをうまく融合させることができないか追求してみたい．機会が許せば，最終目標であるインメモリ DBMS/KVS の実装に提案方式を組み込んだシステムで効果を確かめたい．本論文に示すような高並列データ処理方式を DBMS 主要部分に組み込むことにより，8 コア程度まで性能をスケールさせる目処はあるが，それ以上のコア数の計算機を有効に利用するためにはどのようなシステム構成が良いか[*24]についてはまだまだ議論の余地が多く，今後も探求を続けたい．

[*24] 1 プロセッサチップに載るコア数までは VM で切り分けずに使いきるというのがデータ管理ミドル層ソフトウェアの 1 つの目標となるだろう．

参考文献

[1] IBM: *System/370 Principles of Operation* (1970). manual number: GA22-7000.
[2] Herlihy, M. and Shavit, N.: *The Art of Multiprocessor Programming*, Morgan Kaufmann, San Francisco (2008).
[3] Obermarck, R.L., Palmer, J.D. and Treiber, R.K.: Extended atomic operation (1989). U.S. Patent 4847754.
[4] Michael, M.M.: High Performance Dynamic Lock-Free Hash Tables and List-Based Sets, *Proc. 14th Annual Symposium on Parallel Algorithms and Architectures* (*SPAA '02*), pp.73–82 (2002).
[5] Purcell, C. and Harris, T.: Non-blocking hashtables with Open addressing, Technical Report UCAM-CL-TR-639, Computer Laboratory, University of Cambridge, Cambridge, UK (2005).
[6] Shalev, O. and Shavit, N.: Split-Ordered Lists: Lock-Free Extensible Hash Tables, *J. ACM*, Vol.53, No.3,

pp.379–405 (2006).
[7] Herlihy, M.: Wait-Free Synchronization, *ACM TOPLAS*, Vol.13, No.1, pp.124–149 (1991).
[8] Herlihy, M.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. 20th Annual International Symposium on Computer Architecture* (*ICSA '93*), pp.289–301 (1993).
[9] Fraser, K.: Practical lock-freedom, Ph.D. Thesis, Computer Laboratory, University of Cambridge, Cambridge, UK (2003). Also available as Technical Report UCAM-CL-TR-579 (2004).
[10] Fraser, K. and Harris, T.: Concurrent Programming Without Locks, *ACM TOCS*, Vol.25, No.2, pp.1–59 (2007).
[11] Greenwald, M.: Non-Blocking Synchronization and System Design, Ph.D. Thesis, Computer Science Department, Stanford University, Palo Alto, US (1999). Also available as Technical Report STAN-CS-TR-99-1624 (1999).
[12] 新田　淳，山本章治，米田　茂：共有資源の管理方法 (1989). 日本国特許 特開平 1-303527.
[13] Laarman, A., van de Pol, J. and Weber, M.: Boosting Multi-Core Reachability Performance with Shared Hash Tables, *Proc. Formal Method in Computer Aided Desigin 2010* (*FMCAD*), pp.247–255 (2010).
[14] Yui, M., Miyazaki, J., Uemura, S. and Yamana, H.: Nb-GCLOCK: A Non-blocking Buffer Management Based on the Generalized CLOCK, *Proc. 26th IEEE International Conference on Data Engineering* (*ICDE 2010*), pp.745–756 (2010).
[15] Oracle: *Java Platform, Standard Edition 7 API Specification* (2011), available from ⟨http://docs.oracle.com/javase/7/docs/api/⟩.
[16] Intel: *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z (325383-043US)* (2012), available from ⟨http://download.intel.com/products/processor/manual/325383.pdf⟩.
[17] Radke, C.E.: The Use of Quadratic Residue Research, *Comm. ACM*, Vol.13, No.2, pp.103–105 (1970).
[18] Transaction Processing Performance Council: *TPC Benchmark C – Standard Specification, Revision 5.11* (2010), available from ⟨http://www.tpc.org/tpcc/spec/tpcc_current.pdf⟩.
[19] Harizopoulos, S., Abadi, D., Madden, S. and Stonebraker, M.: OLTP Through the Looking Glass, and What We Found There, *Proc. 2008 SIGMOD Conference on Management of Data* (*Vancouver, B.C.*), New York, pp.981–992, ACM Press (2008).

付　　録

A.1　高並列ハッシュテーブル操作 API

```
// key manipulation functions
typedef const void *getKey_t(const void
*pData);
typedef bool equalKey_t(const void *key1,
const void *key2);
typedef long hashKey_t(const void *key);
// must be > 0

// data types
typedef long  ver_t;
typedef struct {
            void       *pBase;    // hash
table address
            int         size;     // hash
table size
            getKey_t   *getKey;   // key
            equalKey_t *equalKey;
//   manipulation
            hashKey_t  *hashKey;
//     functions
        } HCH_t                   // HCH
descriptor
typedef struct {
            HCH_t      *pHCH;
            int         position;
        } HCHiterator_t           // for
sequential access

// return code
// OK=0, RETRY=-1, NOTFOUND=-2, FULL=-3,
INVALID=-4

// housekeeping functions
HCH_t *newHCH(int size, getKey_t *get,
equalKey_t *equal, hashKey_t *hash);

// constructor
int freeHCH(HCH_t *pHCH);
// destructor
int clearHCH(HCH_t *pHCH);
// initializer

// data manipulation functions
void *HCH_get(HCH_t *pHCH, const void *pkey,
ver_t *pversion);
int HCH_put(HCH_t *pHCH, const void *pData,
const ver_t version);
int HCH_release(HCH_t *pHCH, const void
*pData);
int HCF_delete(HCH_t *pHCH, const void
*pData);
HCHiterator_t *HCHiterator(HCH_t *pHCH);
void *HCH_getnext(HCH_t *pHCH, HCHiterator_t
*piterator);
```

A.2　高並列ハッシュテーブル操作疑似コード

A.2.1　実装用内部データ型と補助関数

```
// UHALF/UWORD/UDOUBLE:
//    half-word/word/double-word length
unsigned integer type

// hashEntry_t, dw1_t, w2_t, w4_t:
//    hash table entry and member types as
depicted in fig1

// flags: INSERTING=0x00000001U

// primitive serialization instructions
bool CAS(UWORD *var, UWORD *old, const UWORD
new);
bool CAS2(UDOUBLE *var, UDOUBLE *old, const
UDOUBLE new);

// i-th position by quadratic residue search
int pos(const long hash, const int i);
```

A.2.2　検索

```
01 void *HCH_get(HCH_t *pHCH, const void
*pKey, ver_t *pversion)
```

```
02 {
03   hashEntry_t *ent = pBase;
04   long hkey = hashKey(pKey);
05   int j = home = pos(hkey, 0);
06   do {
07     w4_t home_w4 = ent[home].w4;
08     for (int i = 0; i <= home_w4.max; j =
pos(home,++i)) {
09       dw1_t old_dw1 = ent[j].dw1;
10       while (old_dw1.hash == hkey) {
11         if (old_dw1.flags & INSERTING) {
12             *pversion = RETRY;
13             return NULL;  // temporarily
conflicting with put
14         }
15         dw1_t new_dw1 = {old_dw1.flags,
old_dw1.ref + 1};
16         if (!CAS2(&ent[j].dw1, &old_dw1,
new_dw1))
17           continue;
18         if (!equalKey(getKey(
ent[j].pData), pKey) {
19           w2_t old_w2 = ent[j].w2;
20           do {
21             w2_t new_w2 = {old_w2.flags,
old_w2.ref - 1};
22           } while (!CAS(&ent[j].w2,
&old_w2, new_w2));
23           break;
24         }
25         *pversion = home_w4.ver;
26         return ent[j].pData;  // key found
27       }
28     }
29   } while (ent[home].ver != home_w4.ver);
30   *pversion = home_w4.ver;
31   return NULL;  // key not found
32 }
```

A.2.3　登録

```
01 int HCH_put(HCH_t *pHCH, const void
*pData, const ver_t version)
02 {
03   hashEntry_t *ent = pBase;
04   long hkey = hashKey(getKey(pData));
05   int j = home = pos(hkey, 0);
06   if (ent[home].ver != version)
07     return RETRY;  // already updated
since last get
08   w4_t home_w4 = ent[home].w4;
09   for (int i = 0; i < size; j = pos(home,
++i)) {
10     if (ent[j].pData != NULL) continue;
11     UWORD old = NULL;
12     if (CAS(&ent[j].pData, &old, pData)) {
13       ent[j].w2 = { ent[j].flags |
INSERTING, 1};
14       // SFENCE required here
15       ent[j].hash = hkey;
16       w4_t old_w4 = home_w4;
17       w4_t new_w4 = {i > home_w4.max ? i :
home_w4.max,
18                      home_w4.ver + 1};
19       if (!CAS(&ent[home].w4, &old_w4,
new_w4)) {
20         ent[j].hash = 0;
21         // SFENCE required here
22         ent[j].w2 = {0, 0};
23         ent[j].pData = NULL;
24         return  RETRY;   // conflicting
with other updates
25       }
26       ent[j].flags &= ~INSERTING;
27       return OK;  // successful insertion
28     }
29   }
30   if (ent[home].ver == home_w4.ver)
31     return FULL;  // hash table overflow
32   else
33     return RETRY;  // conflicting with
other updates
34 }
```

A.2.4　削除

```
01 int HCH_delete(HCH_t *pHCH, const void
*pData)
02 {
03   hashEntry_t *ent = pBase;
04   long hkey = hashKey(getKey(pData));
05   int j = home = pos(hkey, 0);
06   w4_t home_w4 = ent[home].w4;
07   for (int i = 0; i <= home_w4.max; j =
pos(hkey, ++i)) {
08     if (ent[j].pData == pData) {
09       dw1_t old_dw1 = ent[j].dw1;
10       while (old_dw1.hash != 0 &&
old_dw1.ref == 1) {
11         if (CAS2(&ent[j].dw1, &old_dw1,
0)) {
12           ent[j].pData = NULL;
13           w4_t old_w4 = home_w4;
14           for(;;) {
15             for (int k = home_w4.max; k >=
0; --k) {
16               UWORD h = ent[pos(hkey,
k)].hash;
17               if (h != 0 && pos(h, 0) ==
home)
18                 break;
19             }
20             w4_t new_w4 = {k >= 0 ? k : 0,
home_w4.ver + 1};
21             if (CAS(&ent[home].w4,
&old_w4, new_w4))
22               return OK;
// successful deletion
23             else
24               home_w4 = old_w4;
25           }
26         }
27       }
28       return RETRY;  // conflicting with
other updates
29     }
30   }
31   return NOTFOUND;   // could not find the
specified data
32 }
```

A.3　1989 年版のハッシュテーブル構造

図 A·1 に 1989 年提案 [12] のハッシュテーブル構造を示す．図 4 のハッシュテーブルと比較すると以下の点が異なる；

(1) 図 A·1 のテーブルエントリは，図 4 の dw2 相当部分からなる 2 語長の構造体である．

(2) 図 4 の w2 は，図 A·1 では pData で参照されるユー

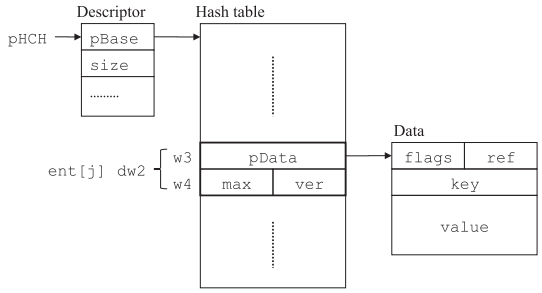

図 A·1　改良前のハッシュテーブル構造

Fig. A·1　Structure of a highly-concurrent hash table (1989 version).

ザデータ内の先頭に配置される．

(3) 図 4 の w1 は図 A·1 には存在しない．

図 4 の方が，ユーザデータ側に余分な管理情報を保持する必要がなく，またほとんどの処理がハッシュテーブル内で完結するのでメモリアクセスの局所性が高くなる．

新田 淳　（学生会員）

1982 年東京大学大学院理学系研究科物理学専攻修士課程修了．同年株式会社日立製作所入所．以来，メインフレームおよびオープンプラットフォーム上での OLTP システムミドルウエア（TP モニタ，DBMS，Web AP サーバ），ストレージ管理ソフトウェア等の研究開発に従事．2011 年より静岡大学創造科学大学院自然科学系教育部（情報科学専攻）に社会人学生として在籍．

石川 博　（フェロー）

静岡大学情報学部情報科学科教授．東京大学理学部情報科学科卒業．東京都立大学を経て 2006 年より現職．東京大学博士（理学）．著書に『データマイニングと集合知—基礎から Web，ソーシャルメディアまで』（共立出版），『集合知の作り方・活かし方—多様性とソーシャルメディアの視点から』（共立出版），『次世代データベースとデータマイニング』（CQ 出版社），『JavaScript によるアルゴリズムデザイン』（培風館），『データベース』（森北出版）等．国際的論文誌 ACM TODS，IEEE TKDE，国際学会 VLDB，IEEE ICDE 等を含め学術論文多数．1994 年情報処理学会坂井記念特別賞，1997 年科学技術庁長官賞（研究功績者）受賞．情報処理学会データベースシステム研究会主査，情報処理学会論文誌（データベース）共同編集委員長，International Journal Very Large Data Bases Editorial Board，日本データベース学会理事歴任．情報処理学会フェロー，電子情報通信学会フェロー．ACM，IEEE 各会員．

（担当編集委員　的野 晃整）