Vol.2011-DBS-153 No.35
2011/11/3

インメモリDBへの適用に向けた実用的なロックフリーハッシュテーブル

新田 淳[†1,†2] 石川 博[†1]

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を，ロックフリーで行う API と処理方式を提示する．これは，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 を実装するための基本部品となる．提案方式の特徴は，データ登録位置とホーム位置のハッシュテーブルエントリを CAS 命令で操作することによりさまざまな競合を検出するところにある．また，データの登録管理情報と参照カウンタを一体操作することで，参照と削除の危険な干渉を回避している．前提となる実行環境の要件を最小限に抑えてあり，汎用のサーバ OS 上だけでなく，組み込みを含む幅広い実行環境で実装可能である．ロックを用いる処理との性能比較を実施し，その効果を確認した．

Practical Lock-free Hash Table with Open Addressing for In-memory DBMS

Jun Nitta[†1,†2] and Hiroshi Ishikawa[†1]

We present the first lock-free hash table API and processing with open addressing that can handle arbitrary <key, value> pairs. One can implement a highly-concurrent in-memory DBMS or KVS using this technique. It detects conflicts between competing processing by using CAS instruction to modify both registering position and home position counters. It also eliminates the dangerous side effects of the residual reference by integrating hash table manipulation and reference counting in a single hash table entry. Throughput measurement shows that this lock-free version outperforms mutex-locked version under various conditions.

†1 静岡大学情報学部
Faculty of Informatics, Shizuoka University
†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

1. はじめに

本稿では，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，実用的なロックフリー操作方式を提示する．オープンアドレッシングでは，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つなぐのではなく，テーブル内の別の空き位置に格納する．この方法では，ハッシュテーブルサイズを超えるエントリを格納することはできないが，キャッシュメモリアクセスの高い局所性を期待できる．

本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の管理モジュールを想定している．ミドル層への応用としては，特にインメモリの RDBMS や KVS(Key-Value Store) の実装において有効である．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エアの進化とそれを利用するソフトウエア処理方式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要性は乏しかった．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，ロックフリーのインメモリデータ処理方式の探求が重要性を増してきている．

提案方式の要点は，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，並列処理の競合を検出・回避する制御の単位として扱うことである．エントリ追加位置のデータ構造を CAS(Compare and Swap) 命令で操作することで空きに対する競合を制御すると共に，ホーム位置のデータ構造を CAS 命令で操作することにより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．本方式は，再帰的な処理完了補助機構を持たないため，ロックフリーではあるがウエイトフリーではない．

提案方式のもう一つの特徴は，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，残留参照（詳細後述）の外部漏洩をなくすことができ，安全性と自己完結性の高いパッケージングを実現したことである．

これまでにも，様々なデータ構造に対するロックフリー処理方式が提案されてきている．

© 2011 Information Processing Society of Japan

本稿もそれらに対して 1 事例を加えるものであるが，特に強調しておきたいのは，高い実用性を目指した点である．一般にコアとなるデータ構造とアルゴリズムが提示された場合，それを実用プログラムで利用するには，利便性や保守性を担保するために付加的なデータ項目や処理を追加する必要がある．ところが，ほとんど全てのロックフリーアルゴリズムは，データ構造やプログラムが微細に変更されただけで成り立たなくなる改変過敏性を持つ．このため，それが実用的に安心して使えるものであるかどうかは，コアアルゴリズムの提示だけでは不十分で，実用提供可能なレベルまで完成されたパッケージングの形を示さなければ判断できない．

本稿では，まず 2 章で関連するロックフリーアルゴリズムの研究事例を示し，3 章でロックフリーアルゴリズムに求められる実用性の条件を明確にする．4 章では外部インタフェースとそれを実現する内部のデータ構造とアルゴリズムを提示する．5 章では性能評価を行い，6 章で総括する．

2. 関 連 研 究

基本的なデータ構造に対するロックフリー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．

アドホックアプローチでは，個別のデータ構造ごとにロックフリー操作アルゴリズムを開発する．ハードウエアで提供される CAS のようなシリアライズ命令[1] 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．また，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手法も 1980 年代に報告されている[2]．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なかなか見つからなかったが，2002 年に Michael が簡潔で有効な方式を示した[3]．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[4]．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ー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．CAS 命令が，このようなデータ構造に対するウエイトフリー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[5]．

本報告が主要な応用対象と想定しているプラットフォームおよびミドルレイヤでの共有資源の管理に多く用いられるのは，アレイとリンクドリストとハッシュテーブルであり，上記の既存研究によって基本的な課題はほぼ言及され部分的には解決されていると言える．ただし，Michael や Purcell の方式には 3 で詳述する残留参照の課題があり，また，Purcell の方式ではキーだけのハッシュテーブルしか扱えない．実用的なロックフリーハッシュテーブル操作は，残留参照なしに < キー, 値 > ペアを扱える必要がある．

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目指す．Herlihy[6] ら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．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 FTSM/OTSM の実装方法を示した[7][8]．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ことも可能であるが，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問題がある．また，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的に扱うことになり，処理の複雑性とオーバヘッドが肥大化しやすい課題もある．

上記，CAS 命令を利用した処理方式群の他に，現在利用可能なハードウエアではほとんどまたは全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[9]．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，実用性は低いと言わざるを得ない．

本稿で述べる方式は，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，新田他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[10] を元にいくつかの改良を加えたものである．

3. 実用性条件

提案処理方式の解説に入る前に，本稿で重要視している「実用性」について，より具体的な定義を述べる．

条件 1: 必要十分な機能を備えたインタフェース (API) を提供すること
実用的であるためには，上位モジュールが規定する < キー, 値 > ペアを操作（登録，検索，削除）できる機能セットを提供する必要がある．キーだけ管理できるのでは不十分であるが，< キー, 値 > ペアより複雑なデータ構造を意識する必要性は少ない．この条件に関連して考慮が必要なことが；

条件 1-1: 部品としての汎用性と上位セマンティクスの取り込みのバランスを取ること
本稿で示すような汎用部品としてのロックフリーデータ処理は，上位が規定する < キー, 値 > ペアのデータ群を，管理データ構造（リストやハッシュなど）にロックフリーで登録・検

© 2011 Information Processing Society of Japan

索・削除するだけの機能を提供する．登録されるデータがどのようなセマンティクスを持って利用されるかは上位の問題であるとして関知しない．しかし，上位レイヤでのデータ処理まで含めて全体として最適なロックフリー性を追求しようとすると，両者をレイヤ分離せずに一体化した個別ロジックとして実装するほうが望ましい場合もあるだろう．どの API レイヤで切るのがよいかは，応用分野に応じた設計バランスの問題である．

条件 2: 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．本稿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．

条件 3: 前提となる動作環境条件は最低限のものであること
サーバ機で動作する OS や DBMS といった層のソフトウエア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目指すため，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．例えば，ガーベジコレクション，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 4: 依存性の高い全ての機能モジュールを内包したパッケージングであること
商用製品で用いられることを想定すると，10 年以上にわたって保守・拡張が行われる状況で安全に使える必要がある[⋆1]．このため，コアのアルゴリズム実装部分だけでなく，メモリ管理など改変時にコアロジックと同期して手を入れる必要がある機能群を一つのパッケージ内にまとめて自己完結性を高める必要がある．ロックフリー処理では，単純なソースコードのクロスレファレンス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が存在し，これらをパッケージ化することが保守性を担保する上で必須である．これに関して特に注意が必要なことは；

条件 4-1: 残留参照をパッケージ内部に閉じ込めること
検索と削除処理が競合した場合，多くの既存方式では，検索処理スレッドがデータ要素へのポインタを保持した状態で別スレッドによる同じデータ要素の削除処理が完了してしまう状況が発生する．これを，ここでは残留参照と呼ぶ．残留参照があると，削除処理スレッドは，削除したデータ要素のメモリ領域を開放したり別目的に再利用したりすることが自由にできなくなる．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，条件 4 によれば，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隠蔽されて，残留参照のタイミングがパッケージのインタフェース外部に漏洩しないようにする必要がある．本稿の処理方式では，データ要素への参照カウンタの管理をコアのハッシュテーブル操作ロジックと一体化することで API 利用者が残留参照を意識しなくてすむようにしている．

⋆1 大規模な製品プロジェクトでは，関連する開発者数は $O(10^1) \sim O(10^2)$ 人規模に及び，しばしば入れ替わりがあることを考慮しなければならない．

4. データ構造とアルゴリズム

本章では，まず実用性条件 1 を満たすパッケージ全体の API を示し，次に実用性条件 2,3,4 を満たす主要部分の実装を解説する．

4.1 インタフェース

付録 A.1 にロックフリーハッシュテーブル操作パッケージ（LFH: Lock Free Hash Table）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全て網羅されており，実用性条件 1 が満たされる；

4.1.1 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，ロックフリーでもスレッドセーフでもない．

- `newLFH()` ：ハッシュテーブル本体とその管理構造体（LFH ディスクリプタ）のファクトリ．LFH はデータ要素の内部構造を意識しないため，上位からデータ構造中のキーを操作する関数（getKey(), equalKey(), hashKey()）を与えてもらう．ハッシュは 2 段階で行い，API 層ではその第 1 段階（キー⇒符号なし 1 語長整数へのマップ）を扱う．第 2 段階（第 1 段階で得た整数⇒ハッシュテーブルエントリ位置へのマップ）は LFH 内部処理になる．
- `freeLFH()` ：ハッシュテーブルとディスクリプタのデストラクタ．
- `clearLFH()` ：ハッシュテーブルのイニシャライザ．

4.1.2 データ操作 API:

ハッシュテーブルの操作をロックフリーで行う関数群．

- `LFH_get()` ：データ要素のキー一致検索（参照カウンタを加算しポインタを返却）
- `LFH_put()` ：データ要素のハッシュテーブルへの登録（参照カウンタを 1 に設定）
- `LFH_release()`：参照カウンタの減算

© 2011 Information Processing Society of Japan

- `LFH_delete()` ：データ要素のハッシュテーブルからの削除（参照カウンタが 1 の場合のみ可能）
- `LFHiterator()`：イテレータの作成・初期化
- `LFH_getnext()`：データ要素の逐次検索（参照カウンタを加算しポインタを返却）

登録の場合，上位プログラムはまず LFH_get() で既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 LFH_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() で戻されるバージョン番号を put() に渡すインタフェースとしている．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()（引数としてバージョン番号を取らず，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．

4.2 操作アルゴリズム

ロックフリー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 LFH ディスクリプタからのポインタ参照（pLFH− >）を省略してある．

4.2.1 逐次化のプリミティブ命令

ロックフリー処理を実現するためのハードウエ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．本稿の評価で使用した Intel プロセッサの場合は，CMPXCHG/CMPXCHG16B 命令がほぼ同等機能を持つ[11]．

4.2.2 ハッシュテーブル構造

図 1 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，倍語× 2，または語× 4 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 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

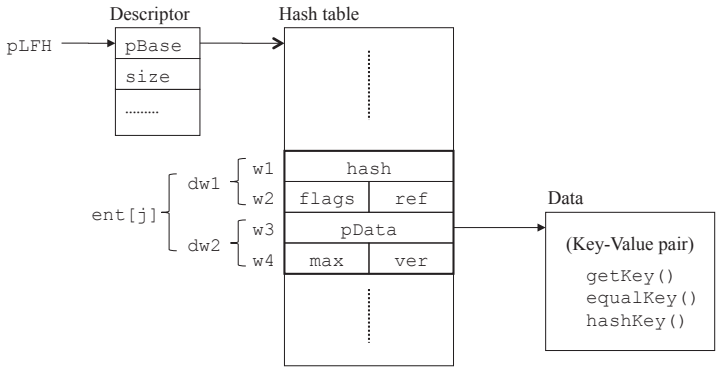

図 1 ハッシュテーブルの構造

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臓し，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 4 が満たされる．

ハッシュ値 h を持つデータ要素の 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関数 pos() によって与えられる；

$N = 4m + 3$ の素数のとき[12]

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

ここでは，衝突発生時の次候補位置を求めるのに 2 次剰余探索を採用しているが，他の方法を採用することも可能である．pos(h,0) がホーム位置である．

pData はデータ要素 < キー, 値 > ペアへのポインタである．LFH 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一切意識しない．max は，この位置をホームとするシノニムセット（pos(h,0) が同じ値であるキーの集合）を全て探索するのに必要な

© 2011 Information Processing Society of Japan

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，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（新規登録または削除）が行われるたびに加算されてゆく．全ての処理は冒頭でホーム位置の 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．

4.2.3 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．この処理で特徴的なのは，ハッシュ値の一致するエントリの参照カウンタを行 16 で加算して競合する削除を防ぎながら，改めて行 18 でキーの同一性を調べていることである．もしキーが不一致であれば参照カウンタを減算して探索ループを続行する（行 19〜行 23）．このようにするのは，キーを直接ハッシュテーブルエントリに取り込むことによって，キーのデータ型に窮屈な制限を与えることを防ぐためである．ハッシュ関数をうまく選べば，キー不一致の可能性は十分小さくできる．また，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，全てハッシュテーブル内のメモリアクセスで処理がすむため，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．

4.2.4 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バージョン番号とホーム位置の現時点での値を比較し，一致していればバーション番号を加算して（行 19）処理完了とする（行 27）．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）．上位プログラムは再び get() からやり直す．仮登録から処理完了までの間にキー一致検索がこのエントリを参照すると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）．

登録処理が，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，登録処理が有限時間内に完了することは保証されていない．この意味で，本方式はロックフリーではあるがウエイトフリーではないと言える．ウエイトフリーにするためには，競合する他の更新処理でこの登録処理を代替するような再帰的な救援処理ロジックを設ければよいが，そのような拡張が可能であるかどうかは更なる検討が必要である．

4.2.5 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），ホーム位置のバージョンカウンタを加算する（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 20）．

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この問題を解決したわけである．このアイディアは他のデータ構造にも応用できるはずであるが，実際に参照カウンタをデータ構造に組み込むことが可能かどうかは（特にアドホックアプローチの場合）個々のアルゴリズムの詳細に依存するので一概には可否を断定できない．

4.2.6 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，擬似コードは省略する．また，キー一致検索だけでなく，ロックフリーな順次検索（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．

4.3 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挙げておく；

(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している．

(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（特に 64 ビット環境）であろう．

(4)　ハッシュテーブル上のバージョン番号（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構わない．API 引数でのバージョン番号は 1 語長の符号付整数となっており，負の値は意味を持つ．

© 2011 Information Processing Society of Japan

(5) 擬似コード上の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，投機的命令実行を行うプロセッサにおいては，メモリアクセス順序制御命令[⋆1]を適宜挿入する必要がある．
(6) 擬似コードから明らかなように，実装ではシリアライズの手段として CAS/CAS2 だけを利用しており，また OS や言語処理系のサービスは一切コールしておらず，実用性条件 2 と 3 を満たしている．厳密に言うと，擬似コードを掲載していない初期化処理 newLFH() 内で malloc() 相当の実行時ヒープ割り当て機能を利用しているが，これも必要であれば静的割り当てなど代替手段への簡単な置き換えが可能である．

5. 評　　価

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[⋆2] X5570: 4 コア× 2 ソケット）を用いた実験を行った．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，スレッドに CPU アフィニティを設定して 1 コア〜8 コアで実行し，一定回数（測定条件により 7,5000 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．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．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．また，実際の使用条件に似せるため，ハッシュテーブル操作にかかる CPU 時間の全体の処理時間に対する比率をいくつか変えて測定した．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 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 2 から図 4 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 コアでのロックフリー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

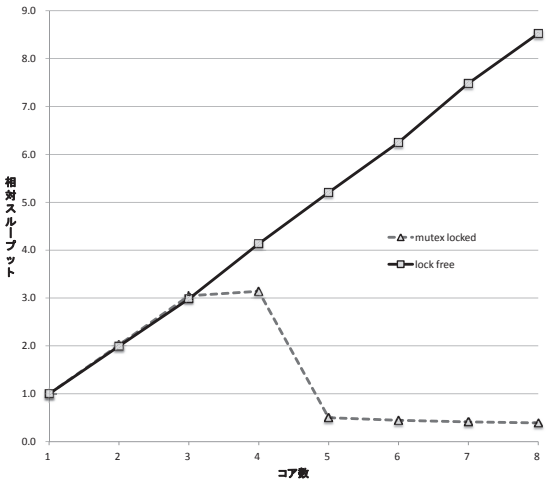

図 2 性能データ (ハッシュ処理比率≒ 1%)

1%の場合は（図 2），ロックフリー版が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．テーブル操作率が 5%になると（図 3），ロックフリー版でも 5 コア以上で伸びが鈍ってくる．テーブル操作率が 90%以上という極端な環境では（図 4），ロックフリー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，ロックフリー版の方が低下傾向が緩やかである．ロック版における 4 コア超でのスループットの極端な悪化は，実験環境での pthread の実装に依存する部分が大きいため，絶対値にこだわることはあまり意味がないが，だいたいの傾向はつかめるであろう．機械命令（CMPXCHG）であろうが，ソフトウエア（mutex）であろうが，コア数に対するスループットのスケーラビリティは，一義的には処理全体の CPU サイクルに占めるメモリアクセスがシリアライズされる部分の CPU サイクルの比率に依存する．ロックフリー版は，ロック版に対してこのシリアライズされる CPU サイクルを大きく削減することで性能を確保している．

6. 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ロックフリーに操作するア

⋆1 Intel プロセッサの場合だと MFENS/SFENCE/RFENCE 命令がこの機能を持つ
⋆2 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．

© 2011 Information Processing Society of Japan

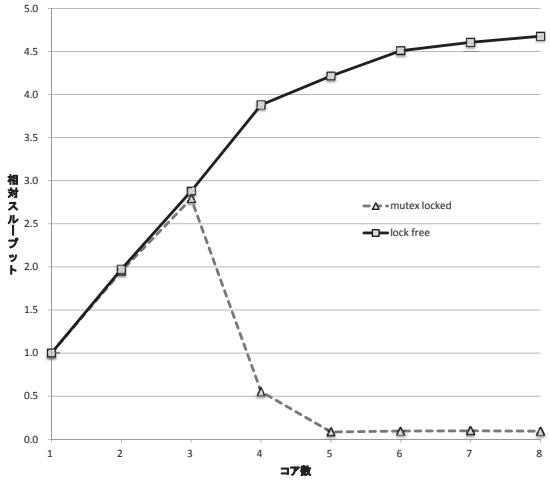

図 3 性能データ (ハッシュ処理比率≒ 5%)

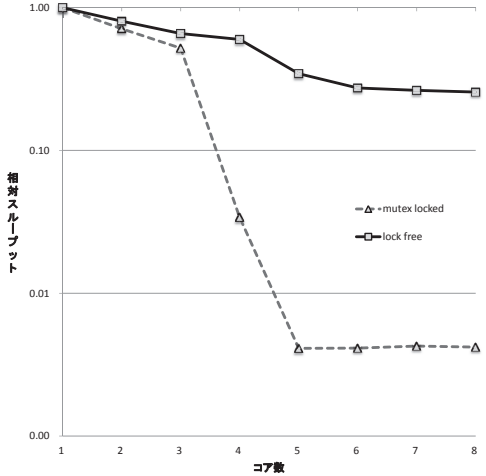

図 4 性能データ (ハッシュ処理比率≧ 90%)

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．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，アルゴリズムを改良してウエイトフリー化ができないか検討してみたい．さらに，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リンクドリスト等の他の汎用的な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．機会が許せば，インメモリ DBMS/KVS の実装に提案方式を組み込んだ環境で効果を確かめたい．

参 考 文 献

1) IBM: *System/370 Principles of Operation* (1970). manual number: GA22-7000.
2) Obermarck, R.L., Palmer, J.D. and Treiber, R.K.: Extended atomic operation (1989). U.S. Patent 4847754.
3) Michael, M.M.: High Performance Dynamic Lock-Free Hash Tables and List-Based Sets, *Proceedings of the 14th Annual Symposium on Parallel Algorithms and Architectures (SPAA '02)*, pp.73–82 (2002).
4) 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).
5) Herlihy, M.: Wait-Free Synchronization, *ACM TOPLAS*, Vol.13, No.1, pp.124–149 (1991).
6) Herlihy, M.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proceedings of the 20th Annual International Symposium on Computer Architecture (ICSA '93)*, pp.289–301 (1993).
7) Fraser, K.: Practical lock-freedom, PhD Thesis, Computer Laboratory, University of Cambridge, Cambridge, UK (2003). also available as Technical Report UCAM-CL-TR-579 (2004).
8) Fraser, K. and Harris, T.: Concurrent Programming Without Locks, *ACM TOCS*, Vol.25, No.2, pp.1–59 (2007).
9) Greenwald, M.: Non-Blocking Synchronization and System Design, PhD Thesis, Computer Science Department, Stanford University, Palo Alto, US (1999). also available as Technical Report STAN-CS-TR-99-1624 (1999).
10) 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.
11) Intel: *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*

© 2011 Information Processing Society of Japan

Instruction Set Reference A-M (253666-037US) (2011).
12) Radke, C. E.: The Use of Quadratic Residue Research, *Communications of the ACM*, Vol.13, No.2, pp.103–105 (1970).

付　　録

A.1 ロックフリー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
        } LFH_t                   // LFH descriptor
typedef struct {
            LFH_t      *pLFH;
            int         position;
        } LFHiterator_t           // for sequential access

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

// housekeeping functions
LFH_t *newLFH(int size, getKey_t *get, equalKey_t *equal,
              hashKey_t *hash);   // constructor
int freeLFH(LFH_t *pLFH);         // destructor
int clearLFH(LFH_t *pLFH);        // initializer

// data manipulation functions
void *LFH_get(LFH_t *pLFH, const void *pkey, ver_t *pversion);
int LFH_put(LFH_t *pLFH, const void *pData, const ver_t version);
int LFH_release(LFH_t *pLFH, const void *pData);
int LHF_delete(LFH_t *pLFH, const void *pData);
LFHiterator_t *LFHiterator(LFH_t *pLFH);
void *LFH_getnext(LFH_t *pLFH, LFHiterator_t *piterator);
```

A.2 ロックフリー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:
//    hash table entry type as depicted in fig1
// dw1_t, w2_t, w4_t:
//    hash table entry subcomponent types as depicted in fig.1

// flags: INSERTING=0x0001U

// 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 *LFH_get(LFH_t *pLFH, 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));
```


© 2011 Information Processing Society of Japan

```
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 LFH_put(LFH_t *pLFH, 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 LHF_delete(LFH_t *pLFH, 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 }
```


© 2011 Information Processing Society of Japan