Vol. 45　No. SIG 5(PRO 21)　　情報処理学会論文誌：プログラミング　　May 2004

非対称なスピンロックの提案とその Java への応用

河内谷　清久仁†　　古　関　　聰†　　小野寺　民也†

Java プログラムの実行では，スレッド間の排他制御のためのロック操作が頻繁に行われる．これを高速化することは，Java 処理系の性能向上に非常に重要である．筆者らは，Java でのロックの挙動として，各オブジェクトごとに特定のスレッドだけが頻繁にロックを行っているというスレッド局所性がみられることを発見し，これを利用した「予約ロック」という手法を提案している．これは，特定のスレッドにロックの予約を与え，そのスレッドによるロック処理を高速化するというものである．しかし，以前提案した予約ロックの実装では，予約を持ったスレッド以外がロックを行うと，それ以降のロック処理は通常の手法で行われてしまうため，スレッド局所性を完全には生かしきれていなかった．また，このモード遷移にはスレッドの一時停止という比較的重い処理が必要であった．本論文では，これらの点を考慮した新しい予約ロック手法について述べる．この手法のベースとして，まず１つのスレッドに限り高速な処理を行える「非対称」なスピンロック機構を提案し，次にこの非対称スピンロックを組み込んだ新しい Java ロック手法について説明する．このロック手法を商用の Java 処理系に実装し，いくつかのベンチマークを走らせたところ，以前の予約ロック実装と比べて予約成功率が向上し，スレッド一時停止による性能低下の危険性が排除されていることを確認できた．

Asymmetric Spin Lock and Its Application to Java

Kiyokuni Kawachiya,† Akira Koseki† and Tamiya Onodera†

Lock reservation, a powerful optimization for Java locks, is based on the observation that, in Java, each lock tends to be dominantly acquired and released by a specific thread. Reserving each lock for such a dominant thread, it allows the thread to acquire and release the lock without any heavy atomic operation. The algorithm recently proposed has embodied the idea and significantly reduced the synchronization overhead on reservation hit. However, when the second thread tries to acquire the lock, the reservation is canceled and the lock will not be accelerated from then. This cancellation also needs the first thread to be stopped, which incurs a performance penalty. In this paper, we propose a new algorithm of lock reservation which overcame the problems. We derive the algorithm in two steps. First, we create an *asymmetric* algorithm for spin lock, in which a specific thread can acquire the lock very quickly. Second, the new asymmetric spin lock is embedded into a state-of-the-art algorithm for Java lock. We have evaluated our algorithm in a production Java virtual machine. The benchmark results show that our algorithm achieves high performance close to the previous algorithm on reservation hit, exhibits no anomaly on reservation miss, and causes more reservation hits.

1. はじめに

Java 言語[1)]では，複数のスレッドを用いた並列処理を容易に記述することができる．この際のスレッド間の同期は，メソッドやブロックを「`synchronized`」と宣言することで指定される．スレッドがその部分を実行する際に，まず指定されたオブジェクトのロックが獲得され，実行が終わったときに解放される．

クラスライブラリを「スレッドセーフ」に作成しなければならないなどの理由から，Java ではロック処理が非常に頻繁に行われる．このコストを下げるために，多くの研究が行われてきているが，これらは，ランタイムによる手法とコンパイラによる手法に大別できる．前者は，ロック処理のアルゴリズムを改良して高速化しようというものであり[2)～6)]，後者は，コンパイル時の解析で不要なロック処理を取り去ってしまおうというものである[7)～12)]．

ランタイムによる高速化は，最もよくあるケースを最適化するという方針で行われてきている．Bacon らは，Java ではロックはほとんど衝突（contention）していないことを発見し，Thin ロック[2)]を開発した．これにより，衝突がないケースでは，わずか数命令でロックの獲得と解放が行えるようになった．Meta ロック[4)]

† 日本アイ・ビー・エム（株）東京基礎研究所
Tokyo Research Laboratory, IBM Research

や Relaxed ロック[5)]も，同様の手法である．

しかし，いずれのランタイム手法でも，compare_and_swap に代表される複雑な不可分命令が，ロックの実現に必要であった．この問題に対し筆者らは，各ロックごとに特定のスレッドに予約権を与え，そのスレッドによるロック処理は不可分命令なしに行い，高速化するという「予約ロック」を提案した[6)]．これは，Java ではロックがそれぞれ特定のスレッドにだけ獲得されているケースが多い（ロックの「スレッド局所性」）という発見を利用したものである．

ただし，文献 6) で提案した予約ロックの実装では，予約を与えられたスレッド以外がそのオブジェクトのロックを行うと，「予約解除」が起こり，それ以降のロック処理は通常のロック手法で行われてしまっていた．また，予約解除にはスレッドの一時停止を含む比較的重い処理が必要であった．

これらの点をふまえ，本論文では，予約解除を必要としない新しい予約ロックの実現法について述べる．この手法のベースとして，まず，「非対称なスピンロック機構」を提案する．これは，1 つのスレッド（予約スレッド）に限って，単純なメモリの読み書きだけでスピンロックの獲得を可能とするものである．予約スレッド以外は，通常どおり不可分命令に基づくスピンロックを行う．次に，通常の Java ロックアルゴリズム内のスピンロック部分をこの非対称スピンロックで置き換える．この 2 つのステップにより，予約スレッドは不可分命令なしに処理が行え，また予約解除という重い処理が起こらない Java ロックが実現できる．

提案したロック手法を商用の Java 処理系に実装し，評価を行った．新しいロック手法は従来の予約ロックにせまる高性能を発揮し，予約解除のペナルティがなく，予約成功率も向上していることを確認した．

2. 従来の予約ロック実装

まず，筆者らが文献 6) において提案した予約ロックの実装について説明する．

この予約ロックは，各オブジェクトのヘッダ内にロック処理用のワード（「ロックワード」）を持つ Java ロック手法を拡張する形で実装される．ロックワード内に，予約の有無を示す 1 ビット（Lock ReserVed: LRV ビット）を用意する（**図 1**）．LRV ビットが 1 の場合を「予約モード」，0 の場合を「ベースモード」と呼ぶ．ロックワードの残りのビットは，予約モードでは予約ロックによって管理され，ベースモードではベースになる通常のロック手法（ベース手法）によって管理される．

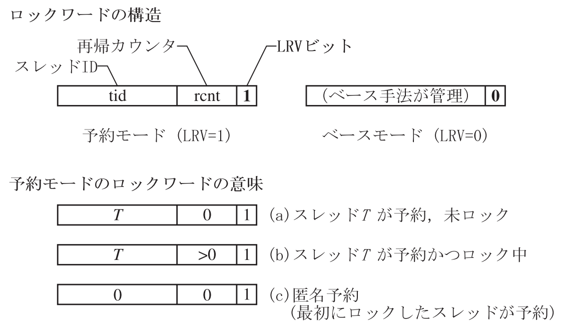

図 1 従来の予約ロック実装におけるロックワードの構造と意味
Fig. 1 Lockword semantics of the original lock reservation.

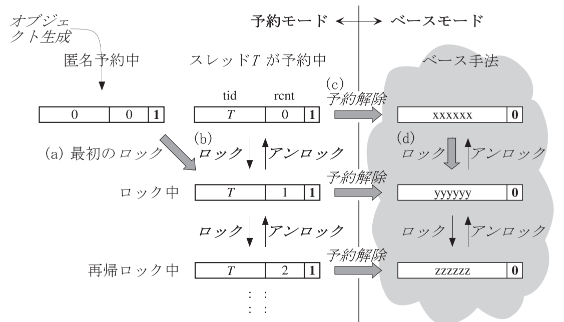

図 2 従来の予約ロック実装におけるロックワードの状態遷移
Fig. 2 State transition of the original lock reservation.

予約モードでは，ロックワードの残りの部分は，そのロックを予約しているスレッドを示す tid フィールドとロックの再帰獲得レベルを示す rcnt フィールドに分割されている．rcnt フィールドが 0 の場合（図 1 (a)），そのロックは予約されているが獲得されていない状態である．1 以上の場合（図 1 (b)）は，予約スレッド T がロックを行っている状態で，値はその再帰獲得レベルを示している．オブジェクトが生成される際，ロックワードは tid フィールドと rcnt フィールドがともに 0 の状態（図 1 (c)）で初期化される．これは，「匿名予約」という特別な状態で，そのオブジェクトを最初にロックしたスレッドが予約スレッドとなる．

図 2 に示したロックワードの状態遷移図で，従来の予約ロック実装の動作を簡単に説明する☆．なお，この図で太い矢印はその遷移に不可分命令が用いられることを示している．

ロックワードは匿名予約状態で初期化され，最初のロックによって予約スレッドが確定される（図 2 (a)）．この処理は，tid フィールドに不可分命令でスレッド

☆ ここでは，本論文の主題である新しい予約ロック手法の説明で必要となる部分についてのみ述べている．より詳細な内容については，元論文[6)]を参照してもらいたい．

ID を設定することで行われる．以後そのスレッドは rcnt フィールドを単純に増減することでロックの獲得と解放を行える（図 2 (b)）．この際に不可分命令を用いる必要がない☆ため，高速な処理が可能となる．

予約スレッド以外のスレッドがロックを試みると，予約が「解除」され，ロックワードはベースモードとなる．この状態遷移は，予約スレッドを一時停止させ，予約モードのロックワードを，ベースモードの対応する状態へと置き換えることで行われる（図 2 (c)）．この際に予約スレッドの停止位置を調べ，ロックワードを読み書きしようとしていた場合はその部分から追い出すことで，競合が起きないことを保証している．予約が解除された後のロック処理は，通常の（不可分命令を用いた）手法により行われる（図 2 (d)）．

筆者らの以前の調査では，Java ではオブジェクトは 1 つのスレッドにだけロックされているケースがほとんどで，予約解除が必要となるのはロック獲得処理全体の 0.05%以下であった．しかしながら，予約解除はスレッドの一時停止という比較的重い処理を含むため，これが頻発するようなプログラムがあった場合，性能が低下してしまう危険性をはらんでいる．また，予約がいったん解除されベースモードになると，以後のロック処理は（たとえそれが元の予約スレッドによるものであっても）通常の手法により行われてしまうため，スレッド局所性を完全には生かしきれていなかった．

これらの点を改善するため，次章以降では，予約ロック実装についての新しいアプローチを検討する．従来の予約ロックが，通常の Java ロック手法を LRV ビットにより拡張する形で実現されていたのに対し，新しいアプローチではまず，実装の基礎となるスピンロックに着目し，特定のスレッドに限り高速な処理が可能となるような「非対称性」を導入する．そして，この非対称なスピンロックを通常の Java ロックに組み込むことで，予約ロックを実現する．

3. 非対称なスピンロックの提案

本章では，新しい予約ロック実装の基礎となる非対称なスピンロックについて述べる．まず，一般的なスピンロックについて概観し，その後，非対称性を導入した新しいアルゴリズムの提案と議論を行う．

3.1 通常のスピンロック

スピンロックは，複雑な同期処理の基本となるプリミティブである．今日のプロセッサは，compare_and_swap や test_and_set に代表される不可分命令を備えており，スピンロックはこれを用いて実装されるのが一般的である．

```
 1 #define SUCCESS 1
 2 #define FAILURE 0
 3 typedef int thread_t;
 4
 5 int compare_and_swap(volatile thread_t *addr,
 6                      thread_t old, thread_t new) {
 7   /* 以下の処理が不可分に行われる */
 8   if (*addr == old) { *addr = new; return SUCCESS; }
 9   else return FAILURE;
10 }
11
12 void acquire(volatile thread_t *lock) {
13   while (try_acquire(lock) != SUCCESS)
14     continue; /* 成功するまでスピン */
15 }
16
17 int try_acquire(volatile thread_t *lock) {
18   return compare_and_swap(lock, 0, thread_id());
19 }
20
21 void release(volatile thread_t *lock) {
22   *lock = 0;
23 }
```

図 3 compare_and_swap のセマンティクスと，それを用いた単純なスピンロック
Fig. 3 Semantics of compare_and_swap and a simple spin lock using it.

図 3 に，以降の説明に用いる compare_and_swap のセマンティクスと，それを用いた単純なスピンロックを示す．ロックの保持者を示すための領域（lock で指される）をメモリ上に用意し，0 で初期化しておく．スレッドがロックを獲得するには，acquire を呼び出す．この関数は，lock で指される領域を 0 から自分のスレッド ID へと書き換えようとする（try_acquire 関数）．この処理は compare_and_swap により不可分に行われ，成功するまで繰り返される（スピン）．獲得したロックを解放するには，release を呼び出し，lock で指される領域を 0 に戻す．これにより，スピンしていたスレッドがロックを獲得できるようになる．

なお，実際に利用されるスピンロックでは，いくつかの拡張が行われているのが普通である．これには，再帰的なロック獲得のサポートや，ロック解放時のエラーチェックなどがあげられる．また，ロック獲得時のスピン待ち合わせを，メモリ読み出し命令と指数的バックオフ（exponential back-off）により行い[13]，スケーラビリティを向上させるのも一般的である．本章では単純化のため，これらの拡張については省略して説明を続ける．

3.2 非対称性の導入

次に，本論文の中心となる「非対称スピンロック」について述べる．これは，スピンロックに予約の概念を新たに導入し，特定のスレッド（予約スレッド）に限り不可分命令を使わない高速なロック獲得を可能にするものである．**図 4** がそのデータ構造で，スレッド

☆ 厳密にいうと，この手法はワードの読み書きが不可分に行えることを利用している．本論文でいう不可分命令とは，compare_and_swap のような，メモリ読み出し，チェック，書き込みを不可分に行う複雑な命令のことである．

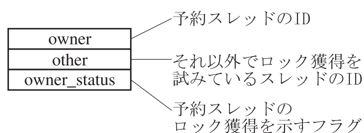

図 4　非対称スピンロックのデータ構造
Fig. 4　Data structure of the asymmetric spin lock.

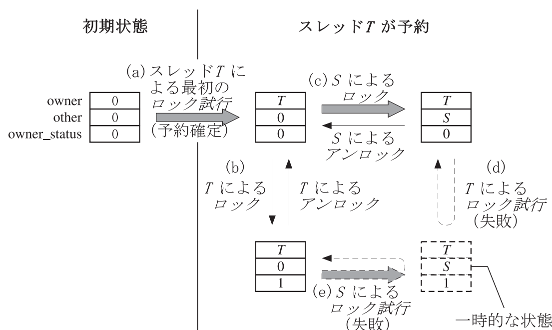

図 5　非対称スピンロックの状態遷移
Fig. 5　State transition of the asymmetric spin lock.

ID を保持する`owner`，`other` の 2 つのフィールドと，予約スレッドのロック獲得状態を示す`owner_status` フィールドからなる．

このデータ構造の状態遷移を**図 5** に示す．図 2 と同様，太い矢印はその遷移に不可分命令が用いられることを示している．

まず，3 つのフィールドはすべて 0 で初期化される．これは，従来の予約ロックの匿名予約に相当する状態である．最初にロックを試みたスレッドが，そのロックの予約スレッドとなり，その ID が`owner` フィールドに設定される（図 5 (a)）．複数スレッドによる競合をさけるため，この処理は不可分命令を用いて行われる．なお，このフィールドは，いったん設定されると二度と変更されない．

予約スレッドは，`owner_status` フィールドに 1 を書き込むことでロックを獲得する（図 5 (b)）．一方，それ以外のスレッドは，`other` フィールドに自分のスレッド ID を書き込むことでロックを獲得する（図 5 (c)）．こちらの処理は複数のスレッドにより試みられる可能性があるため，不可分命令を用いて行う．

これら 2 種のロック獲得処理は，同時に起こることがありうる．これによる競合を解決するため，上記のフィールド書き込みを行った後，もう一方のフィールド（`other` もしくは`owner_status`）を読み出す．これが 0 であった場合，ロック獲得が成功したことになる．もし 0 でなかった場合，自分が書き込んだフィールドを 0 に戻しスピンする（図 5 (d)，(e)）．

なお，ロックの解放は，自分が書き込んだフィールドを 0 に戻すだけで完了する．全体をとおして，`owner_status` フィールドは，予約スレッドによってのみ変更され，`other` フィールドは，それ以外のスレッドによってのみ変更される点に注意してほしい．

```
 1 typedef struct {
 2   thread_t owner;   /* 予約スレッドの ID */
 3   thread_t other;   /* それ以外で獲得を試みているスレッドの ID */
 4   int owner_status; /* 予約スレッドのロック獲得を示すフラグ */
 5 } asym_spin_t;
 6
 7 void acquire(volatile asym_spin_t *lock) {
 8   while (try_acquire(lock) != SUCCESS)
 9     continue; /* 成功するまでスピン */
10 }
11
12 int try_acquire(volatile asym_spin_t *lock) {
13   thread_t owner = lock->owner;
14   thread_t myID  = thread_id();
15
16   if (owner == 0) {    /* 予約の確定（最初のロック獲得） */
17     if (compare_and_swap(&lock->owner, 0, myID) != SUCCESS)
18       return FAILURE;
19     owner = myID;      /* 予約が成功すると下へ続く */
20   }
21   if (owner == myID) { /* 予約スレッドの場合 */
22     lock->owner_status = 1;
23     if (lock->other == 0)
24       return SUCCESS;
25     else {
26       lock->owner_status = 0;
27       return FAILURE;
28     }
29   }
30   else {               /* 予約スレッド以外の場合 */
31     if (compare_and_swap(&lock->other, 0, myID) != SUCCESS)
32       return FAILURE;
33     if (lock->owner_status == 0)
34       return SUCCESS;
35     else {
36       lock->other = 0;
37       return FAILURE;
38     }
39   }
40 }
41
42 void release(volatile asym_spin_t *lock) {
43   if (lock->owner == thread_id())
44     lock->owner_status = 0;
45   else
46     lock->other = 0;
47 }
```

図 6　非対称スピンロックのアルゴリズム
Fig. 6　Algorithm of the asymmetric spin lock.

以上の動作を C 言語で記述したものを**図 6** に示す．`try_acquire` 関数が処理の中心部分である．16〜20 行目で予約スレッドを確定する処理（図 5 (a)），21〜29 行目で予約スレッドによるロック獲得処理（図 5 (b)，(d)），30〜39 行目で予約スレッド以外によるロック獲得処理（図 5 (c)，(e)）を行っている．

ロックの衝突が起きていない場合，非対称スピンロックの性能は以下のようになる．ロックを獲得し解放するのに必要なメモリ操作は，予約スレッドの場合，3 回の読み出し（13，23，43 行目）と 2 回の書き込み（22，44 行目）でよく，不可分命令は不要である．それ以外のスレッドでは，3 回の読み出し（13，33，43 行目）と 1 回の書き込み（46 行目）に加え，1 回の不可分読み書き命令（31 行目）が必要となる．予約スレッドのロック操作には不可分命令が不要であるため，予約成功率が高ければ高速化が期待できる．

3.3　議　　論

次に，上で述べた非対称スピンロックについて，い

くつかの議論と拡張を行う．

3.3.1　Dekkerのアルゴリズム

不可分命令を用いず，単純なメモリ読み書きのみでスピンロックを行う方法は，実はいくつか知られている[14)～17)]．最も有名なものに，Dekkerのアルゴリズム[14)]がある．これは，2つのプロセス（スレッド）が1つのクリティカルセクションを排他的に実行するための仕組みである．それぞれのプロセスに，排他制御用のフラグを用意する．クリティカルセクションを実行するには，まず自分のフラグを立て，相手のフラグを調べる．立っていた場合は，自分のフラグを落としてやり直す．立っていなければ，クリティカルセクションを実行し，抜けるときに自分のフラグを落とす．

これらのアルゴリズムは，ロックを取り合うプロセスが2つの場合は比較的シンプルであるが，プロセス数が増えるとそれに比例して必要なメモリ操作の数が増えてしまい，高速性が損なわれるという問題があった．我々の非対称スピンロックは，スレッドを「予約スレッド」と「それ以外」に二分することでこの問題を解決したものと位置づけることができる．予約スレッドは，Dekkerのアルゴリズムと同様，メモリ書き込みとその後の読み出し確認のみでロックを獲得できる．それ以外のスレッドは，まず不可分命令により「代表者」を選び（31～32行目），勝ち残ったものが予約スレッドとDekker方式でロックを獲得しあっている．非対称性を導入し，不可分命令を組み合わせることで，多数のスレッドがある状況に対応した．しかし一方，不可分命令なしの処理が行えるのは予約スレッドによるロックのみとなる．

3.3.2　正当性について

我々の非対称スピンロックでは，最初のロック獲得時に`owner`フィールドを設定することで予約スレッドが決定される．これは`compare_and_swap`により行われる（17行目）ため，すでに設定された予約スレッドが上書きされてしまう事態は起こらない．`owner_status`フィールドは，予約が確定後に予約スレッドによってのみ変更されるため，競合は起こらない．

一方，`other`フィールドは，複数のスレッドがロック獲得時に変更しようとするが，これも`compare_and_swap`により行われる（31行目）ため，他のスレッドが設定した値を上書きしてしまうことはない．ロック解放時に`other`フィールドを0に戻す処理は，ロックを獲得しているスレッドのみが行うので，競合することはない．

予約スレッドとその他のスレッドの間で排他制御が行え，デッドロックが生じないことは，Dekkerのアルゴリズムにより保証されている．ただし，前節で示した非対称スピンロックの実装では，ライブロック状態におちいる可能性が残っている．予約スレッドと，もう1つのスレッドが同時にロック獲得を試み，予約スレッドが22，23，26，27行目（図5(d)），他方が31，33，36，37行目（図5(e)）でスピンを繰り返す状況である．

Dekkerのアルゴリズムでは，2つのプロセスが同時にロック獲得を試みた場合にどちらを優先するかを示す変数を追加することで，ライブロックを回避している．非対称スピンロックでもこの方法を用いることができるが，メモリ操作が増えてしまうため採用していない．3.4節で示す1ワード版の非対称スピンロックで，別の方法でライブロックを回避できることを示す．なお，Javaロックに組み込む場合，`try_acquire`が失敗すると，サスペンドロックに遷移するのが一般的である（4章を参照）ので，この実装のままでもライブロックは起こらない．

3.3.3　マルチプロセッサ環境での実装について

非対称スピンロックでは，予約が成功した場合，不可分命令なしに処理が行える．しかしこれは，メモリ書き込みと読み出しがプログラムどおりの順序で行われることに依存している．具体的には，予約スレッドが`owner_status`フィールドを1にする処理（22行目）と，`other`フィールドが0であることを確認する処理（23行目）は，この順で行われなければならない．

緩和メモリモデル[18),19)]を採用したマルチプロセッサシステムでは，この順序を保証するためにメモリバリア命令が必要だが，プロセッサによっては，ソフトウェア的な手法により実行順序の保証を行える場合もある．たとえばIBM 370では，`write into X`, `read from X`, `read from Y`という命令シーケンスによって，Xへの書き込みがYからの読み出しより前に行われることを保証できる[18)]．これらのソフトウェア手法は，一般にハードウェアによるメモリバリアよりも軽いことが多いが，どのようなものが使えるかはアーキテクチャによってさまざまである．

どのような手法で順序を保証するにせよ，それが`compare_and_swap`などの不可分命令よりも安価なものであれば，非対称スピンロックによる高速化が期待できる．

3.3.4　最　適　化

もし，`other`フィールドと`owner_status`フィールドを，まとめて比較，変更できる不可分命令があれば，予約スレッド以外のロック獲得処理をより単純化できる．具体的には，`owner_status`が0であることの確

認（33 行目）を，compare_and_swap（31 行目）の際に同時に行えばよい．

同様に，もし owner フィールドと owner_status フィールドを，まとめて比較，変更できれる不可分命令があれば，予約スレッドの確定処理（17 行目）に，ロック獲得処理をまとめてしまうことができる．

データ構造を工夫し，複数のフィールドを compare_and_swap が可能なワード内にまとめてしまうことで，double_compare_and_swap [20)]（DCAS，CAS2 [21)]）のような特別な命令がなくても，これらの最適化を行うことが可能である．次節で述べる 1 ワード版非対称スピンロックは，その実例である．

3.4　1 ワードへのデータ圧縮

前節までで述べた非対称スピンロックでは，説明を平易にするため，3 つのフィールドを独立したワードとして扱っていた．本節では，Java ロックに組み込むことを前提とし，データ構造を 1 ワード（32 ビット）に圧縮しメモリ効率を上げた改良版を示す．

1 ワード版の実装は，32 ビットのワード全体に対してメモリ読み書きと compare_and_swap を行うことができ，メモリの書き込みは 8 ビット，16 ビット単位でも（他の部分に影響を与えずに）行えることを前提としている．1 ワードは**図 7** のように分割して用いる．書き込みの最小単位である 8 ビットを owner_status に対して割り振り，残りの 24 ビットを owner と other に二分している．

図 8 に，1 ワード版非対称スピンロックのアルゴリズムを示す．owner_status と other の変更がお互いに影響を与えないように，前者の変更は 8 ビットのメモリ書き込みで行われ（26，45 行目），後者のクリアは 16 ビットのメモリ書き込みで行われる（48 行目）．このとき，owner フィールドの一部にも書き込みが行われてしまうが，このフィールドは予約スレッドが確定した後は変更されないので，問題は起こらない．

予約の確定と，予約スレッド以外によるロック獲得には，ワード全体に対する compare_and_swap が使用されている（23，37 行目）．これにより，複数フィールドを同時にチェックし変更することが可能となるので，3.3.4 項で述べた 2 つの最適化も行っている．

この最適化の副次効果として，1 ワード版ではライブロックが発生しない．予約スレッド以外がロックを獲得する場合，other フィールドの設定と owner_status フィールドが 0 であることの確認が compare_and_swap でまとめて行われる（37 行目）．そのため，other が 0 以外になるのは，そのスレッドがロック獲得に成功したときだけだからである．

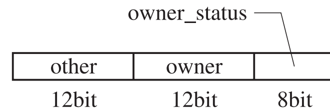

図 7　1 ワード版のデータレイアウト
Fig. 7　Data layout in the one-word variation.

```
 1 #define OWNER_MASK   0x000fff00
 2 #define OTHER_MASK   0xfff00000
 3
 4 /* 以下の定義はリトルエンディアンを仮定している */
 5 #define OWNER_STATUS_BYTE(lock)  (((char  *)lock)[0])
 6 #define OTHER_SHORT(lock)        (((short *)lock)[1])
 7
 8 /* thread_id() は，owner フィールドに合わせてシフト済みの
 9    0x00000100〜0x000fff00 の範囲の値を返すものとする */
10 #define OWNER_TO_OTHER(tid)      ((tid)<<12)
11
12 void acquire(volatile word_t *lock) {
13   while (try_acquire(lock) != SUCCESS)
14     continue; /* 成功するまでスピン */
15 }
16
17 int try_acquire(volatile word_t *lock) {
18   word_t l = *lock;
19   word_t myID = thread_id();
20
21   if ((l & OWNER_MASK) == 0) {          /* 予約の確定とロック */
22     word_t locked = myID | 1;
23     return compare_and_swap(lock, 0, locked);
24   }
25   else if ((l & OWNER_MASK) == myID) { /* 予約スレッドの場合 */
26     OWNER_STATUS_BYTE(lock) = 1;
27     if ((*lock & OTHER_MASK) == 0)
28       return SUCCESS;
29     else {
30       OWNER_STATUS_BYTE(lock) = 0;
31       return FAILURE;
32     }
33   }
34   else {                               /* 予約スレッド以外の場合 */
35     word_t unlocked = l & OWNER_MASK;
36     word_t locked = unlocked | OWNER_TO_OTHER(myID);
37     return compare_and_swap(lock, unlocked, locked);
38   }
39 }
40
41 void release(volatile word_t *lock) {
42   word_t l = *lock;
43
44   if ((l & OWNER_MASK) == thread_id())
45     OWNER_STATUS_BYTE(lock) = 0;
46   else {
47     word_t unlocked = l & ~OTHER_MASK;
48     OTHER_SHORT(lock) = OTHER_SHORT(&unlocked);
49   }
50 }
```

図 8　1 ワード版非対称スピンロック
Fig. 8　Algorithm of the one-word variation.

4. Java ロックへの組み込み

Java では多くのロックが，特定のスレッドによってのみ頻繁に獲得されるという「スレッド局所性」を持っていることが知られている [6)]．そのため，前章で提案した非対称スピンロックを利用することで，処理性能を向上できる可能性がある．

しかし，スピンロックをそのまま Java ロックに用いるのは現実的ではない．Java の同期機構はモニタ [22)] に基づいており，ロックによって保護されるクリティカルセクションは短時間で終わらない．そのため，スレッドスケジューラと連動して動作し，ロックが衝突した場合は獲得できるまでスレッドをサスペンドする「サスペンドロック」が必須である．

サスペンドロックで広く行われている最適化に，スピンロックと組み合わせて「ハイブリッド」化するという手法がある[23]．これは，まずスピンロックでロック獲得を試み，失敗した場合は数回のリトライの後サスペンドするというものである．

Java では，ロック操作の対象はオブジェクトである．ロック処理を高速に行うためには，各オブジェクトのヘッダ内にそのためのデータ構造を持つことがのぞましいが，ヘッダエリアは貴重であり多くのサイズを割くことは難しい．なるべく小さいデータ構造で高速なハイブリッドロックを実現するために Java 向けに考案された手法が,「バイモーダルロック[2),3),24)]」である．

このロック手法では，オブジェクトヘッダ内の 1 フィールド（ロックワード）のみを用い，それを 2 つのモードで使い分ける.「フラットモード」では，ロックワードはスピンロック用に使用され，ロックを獲得中のスレッド ID を保持する.「ファットモード」では，ロックワードはサスペンドロック用のデータ構造へのポインタを保持する．これら 2 つのモードは,「Shape ビット」と名づけられた 1 ビットにより区別される．

ロックがスレッド間で衝突していない場合は，ロックワードはフラットモードとなり，スピンロックと同等のコストでロック処理が行える．衝突が起こると，サスペンドロック用のデータ構造が用意され，ロックワードがファットモードへと遷移する．

バイモーダルロックの実装例の 1 つに，Tasuki ロック[3)]がある．これは，Thin ロック[2)]を改良したもので，フラットモード時のスピンロック処理に compare_and_swap が使われている．この部分を，非対称スピンロックで置き換えることで，予約をサポートした新しい Java ロックを実現できる．置き換えは，3.4 節に示した「1 ワード版」をベースに行うが，データレイアウトを**図 9** のように修正し，Shape ビットを組み込んでいる☆．

この置き換えにより，衝突がない場合には予約スレッドは不可分命令なしに処理が行え，また予約解除という重い処理が起こらない Java ロックが実現できる．なお，Tasuki ロックのアルゴリズム，および置き換えの詳細については，付録 A.1 を参照してもらいたい．

5. 性能評価

本章では，前章までで示した新しい予約ロックの性能評価を行う．

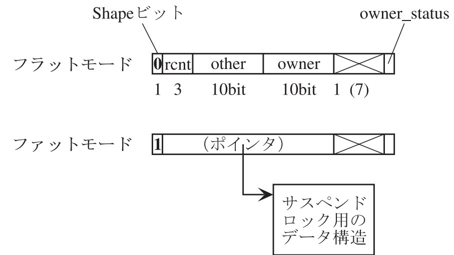

図 9 新しい予約ロックのためのロックワードの構造
Fig. 9 Lockword structure of the new reservation-based Java lock.

評価のベースとして用いた Java 処理系は，IBM Developer Kit for Windows, Java Technology Edition, Version 1.4.0[25)]である．この処理系に，2 章で述べた従来の予約ロック[6)]と，前章で述べた非対称スピンロックに基づく新しい予約ロックを実装し，この処理系が元々そなえている Tasuki ロック（ベース手法）を含めた 3 つのロック手法間の比較を行った．なお，処理系が内蔵している JIT コンパイラ[26)]も同時に修正している．

以後の測定はすべて，2 個の 933 MHz Pentium III プロセッサと 512 MB のメモリを搭載し，Windows XP Professional Edition SP1 が動作している IBM IntelliStation M Pro 上で行ったものである．

5.1 マイクロベンチマーク

まず，3 つのロック手法の基本性能と動作特性を明確化するため，マイクロベンチマークを行った．このベンチマークでは，2 つのスレッド T と S が生成され，同一の synchronized ブロックを 3 回ずつ実行する処理を交互に行う．各回の実行に要した時間を測定することで，さまざま予約状態におけるロック処理の性能を収集できる☆☆．

図 10 が測定結果で，3 つのロック手法が，それぞれの回において synchronized ブロックを実行するのに要した時間を示している．横軸は，何回目の実行であるかと，それを実行したスレッドを示しており，縦軸は，ベース手法における 1 回目の所要時間を 1 として正規化した値である．

まず，ベース手法の結果について検討する．このベ

☆ この図の rcnt フィールドは，スピンロックの再帰獲得をサポートするためのものである．

☆☆ synchronized ブロック内での処理は，カウンタをインクリメントするだけの単純なものである．なお，1 つのオブジェクトに対する synchronized ブロックの実行では速すぎて計測が難しいため，あらかじめ多数のオブジェクトを生成しておき，各回の実行ではそれらに対して順に synchronized ブロックを実行し，全体の所要時間を測定するという方式をとっている．

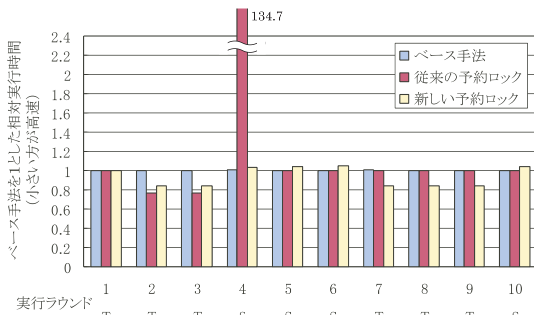

図 10　マイクロベンチマークの結果
Fig. 10　Micro-benchmark results.

ンチマークでは，2 つのスレッドが使われているが，交互に動作するためロックの衝突は起こらない．そのため，すべての回において 1 回の`compare_and_swap`だけでロックの獲得と解放が行え，均等な性能が発揮できている．

次に，従来の予約ロックの結果について検討する．1 回目の実行は，予約スレッドを確定する（図 2 (a)）のに`compare_and_swap`が使用されるため，ベース手法とほぼ同等の時間を要している．しかし，これにより，Tが予約スレッドとなるので，2 回目，3 回目の実行では不可分命令なしでロック処理を行える（図 2 (b)）．そのため，ベース手法と比べて所要時間が減っていることが分かる．これが予約ロックの意義で，この状態が長く続けば続くほど性能が向上する．

しかし，4 回目の実行で第 2 のスレッド Sが現れると，この手法の持つ危険性が表面化する．この回は，Tが持っている予約を解除する処理が行われる（図 2 (c)）ため，ベース手法と比べても 100 倍以上の時間がかかってしまう．さらに，いったん予約が解除されると，ベース手法と同様のアルゴリズムでロック処理が行われる（図 2 (d)）ため，5 回目以降の実行は，たとえTによる処理であっても，ベース手法と同等の性能になってしまっている．

最後に，新しい予約ロックの場合であるが，1〜3 回目の実行については，従来の予約ロックと同様の挙動となる．予約が成功している状況（2，3 回目）では，ベース手法に比べて所要時間が減り，従来の予約ロックに近い性能を発揮できている．若干の性能低下は，`other` フィールド確認の処理が必要なためと考えられる．

4 回目以降の実行で，従来の予約ロックとの大きな違いが現れる．まず，新しい予約ロックでは，予約解除が必要ないため，Sによる最初の実行（4 回目）が極端に遅くなることはない．さらに，依然として Tが予約スレッドであるため，7〜9 回目の実行も高速化されている．なお，Sによる実行には`compare_and_swap`が用いられるため，ベース手法とほぼ同等の性能となっている．

これらをまとめると，それぞれのロック手法の特性は以下のようになる．従来の予約ロックは，各オブジェクトを特定のスレッドだけがロックしているようなプログラムでは，図 10 の 2，3 回目にあたる処理がほとんどになるため，最も高速化が期待できる．しかし，複数のスレッドがロックを取り合うケースが多いプログラムでは，5 回目以降にあたる処理がほとんどになるため，高速化が期待できなくなる．ロックを行うオブジェクトの数が多いと，最悪の場合，4 回目にあたる予約解除処理により性能が低下してしまうおそれがある．一方，新しい予約ロックは，前者のプログラムでは従来の予約ロックほど高速にはならないが，ベース手法よりは速くなると期待できる．後者のプログラムでも，7〜9 回目にあたる部分のロック処理が多ければ高速化が期待できる．

これらをふまえ，次節ではより現実的なマクロベンチマークによる性能比較と，ロックの挙動調査を行った結果を示す．

5.2　マクロベンチマーク

測定に用いたマクロベンチマークは，SPECjvm98 [27] の 7 つのプログラムと，2 つの科学計算ベンチマークである．SPECjvm98 ベンチマークは，問題サイズを 100%に指定し，それぞれをアプリケーションモードで個別実行している．科学計算ベンチマークとしては，SPLASH-2 ベンチマーク [28] の Water と Barnes を文献 29) の著者らが Java で書き直したものを用いている．このうち，SPECjvm98 の`_227_mtrt`と，Water，Barnes の 3 つはマルチスレッドプログラムである．

まず，各ベンチマークのロック特性を調べた結果を**表 1** に示す．この情報は，測定に用いたものと同じ Java 処理系にイベント集計コードを追加し，各ベンチマークを走らせて収集した．SPECjvm98 ベンチマークは，実行回数を 3 に指定している．

この表から分かるように，`_201_compress`と`_222_mpegaudio`以外のベンチマークでは，多くのロック処理が行われている．1 章でもふれたとおり，シングルスレッドプログラムでもこの状況がみられるのが，Java の特徴である．

表 1 はまた，各ベンチマークにおいてロック予約が成功した率も示している．これは，最外ロックにおいて予約が成功した場合を全ロック数で割ったものである．再帰的なロック獲得は，元々不可分命令なしで行

表 1　マクロベンチマークのロック特性
Table 1　Lock statistics of macro-benchmarks.

プログラム名	ロックされたオブジェクト数	ロック処理の総数	従来の予約ロック 予約の成功率	従来の予約ロック 予約解除の総数	新しい予約ロック 予約の成功率
SPECjvm98					
_202_jess	12800	14977053	99.353%	187	99.356%
_201_compress	2462	35382	85.764%	127	86.868%
_209_db	66800	170834005	99.982%	52	99.982%
_222_mpegaudio	2111	31201	88.327%	91	89.028%
_228_jack	538631	46972114	95.822%	144	95.859%
_213_javac	133448	43820079	98.662%	1760	99.676%
_227_mtrt	3358	3528225	99.451%	114	99.548%
SPLASH-2					
Water	858230	4326541	43.668%	6022	44.342%
Barnes	216459	2064200	25.245%	78819	34.076%

えるため，予約スレッドによるものであっても，予約成功にはカウントしていない．いいかえると，この数値が各手法により高速化されるロックの割合となる．SPECjvm98 ベンチマークについては，2 つの予約ロック手法のいずれでも，ほぼすべてのロックにおいて予約が成功している．科学計算ベンチマークでは成功率が低下しているが，それでも全ロック処理の 25%以上で予約が成功している．

すべてのベンチマークにおいて，新しい予約ロックの方が予約成功率が高いことにも注目してほしい．これは，予約スレッドの「2 巡目」以降のロック（図 10 の 7〜9 回目にあたるケース）も高速化できるからである．この差は Barnes ベンチマークにおいて最も顕著である．

前節でも示されたように，従来の予約ロックでは予約解除が多いと性能が低下してしまう．そこで，各ベンチマークで起きた予約解除の回数についても集計を行った．SPECjvm98 ベンチマークでは，予約解除はほとんど起こっていない☆．_213_javac が若干多めであるが，ロックされたオブジェクトのうちの 1.3%程度であり，ロック処理自体が多いので，予約解除のペナルティは相対的には低くなると予想される．

一方，科学計算ベンチマークではかなりの予約解除が起きており，特に Barnes ではロックされたオブジェクトの 36%にものぼっている．この 2 つのベンチマークは，C で書かれたものを Java に移植したものであり，Java の標準クラスライブラリにあまり依存していない一方，ロックが本当に複数スレッド間の同期に必要な場合に使われているためだと考えられる．

☆ シングルスレッドプログラムでも予約解除が起きているのは，Java 処理系が生成する内部スレッドが一部のオブジェクトのロックを行うためである．

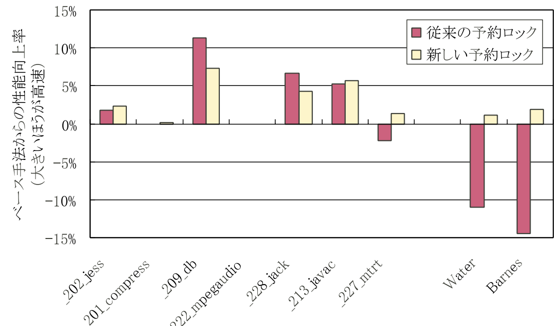

図 11　マクロベンチマークの結果
Fig. 11　Macro-benchmark results.

続いて**図 11** に，各ベンチマークのスコアが，2 つの予約ロック手法によってベース手法からどの程度向上したかを示す．これは，各ベンチマークを 3 つのロック手法を実装した Java 処理系で繰り返し実行し，それぞれのベストスコアどうしを比較したものである．

まず SPECjvm98 だが，従来の予約ロックは _209_db で 11.3%の性能向上を達成している一方，_227_mtrt ではベース手法よりもかえって遅くなっている．新しい予約ロックは，_209_db で 7.4%性能が向上し，_227_mtrt でも性能低下がみられない．_202_jess と _213_javac では，従来の予約ロックよりも良いスコアを記録している．_201_compress と _222_mpegaudio は，元々ロック処理自体が少ないため，いずれの予約ロック手法でも効果がみられない．

科学計算ベンチマークは，従来の予約ロックでは性能が低下してしまっている．特に Barnes は 14.4%も遅くなっており，予約解除のコストが無視できない場合があることを示している．それに対し，新しい予約ロックは予約解除のオーバヘッドがないため，これら

のベンチマークでも性能が向上している．

6. 関 連 研 究

ロックに関しては，これまでにも多くの研究や実装が行われてきている．本章ではこのうち，本論文と関係の深い 3 つのカテゴリについて，順に関連研究をあげていく．

6.1 メモリ読み書きによるロックの実現

ロックに関する初期の研究は，メモリ読み書きの不可分性を利用してプロセス間の排他制御を実現するというもので，Dekker のアルゴリズム[14]をはじめ，さまざまなアルゴリズムが考案された[15)〜17)]．

しかしこれらは，排他制御に関わるプロセス数が増えると，それに比例したメモリアクセスが必要となるため，スケーラビリティが乏しく，実用システムで用いるのは難しかった．そのため，test_and_set や compare_and_swap のような，ロック用の不可分命令の登場とともに，消滅していった．

3.3 節でも議論したが，非対称スピンロックは，スレッドを「予約スレッド」と「それ以外」の 2 つのグループに分け，後者に関しては不可分命令を用いて代表者を選ぶようにすることで，Dekker のアルゴリズムの発想を復活させたものということができる．

なお，同様のアイデアが，C 言語の並列拡張である Cilk 言語のタスクスティール機構の実装でも用いられている[30]．ここでは，タスクを保持中のスレッドに優先権（予約）を与え，不可分命令なしの排他処理を可能としている．本論文の非対称スピンロックの特徴としては，予約スレッド以外のスレッドがother フィールドに自分の ID を書き込むことが同時に Dekker 方式のロック処理の一部になっている点や，タスクスティールという限定された状況ではなく Java 言語の通常ロックへ適用しているという点があげられる．

6.2 マルチプロセッサ環境でのロック

不可分命令を用いることでロックが容易に実装できるようになると，マルチプロセッサ上のスピンロックの高速化が問題となった．

Anderson は，マルチプロセッサシステム上でさまざまなスピンロック手法の調査を行い，メモリ読み出しによるスピンや指数的バックオフといった技法について議論を行った[13]．また，Mellor-Crummey らは，各プロセッサが各自のローカルエリア上でスピンすることでバストラフィックを下げる手法を提案した[31]．これらの最適化技法は，本論文の非対称スピンロックにも適用可能である．

スピンロックとサスペンドロックを組み合わせてハイブリッド化するというアイデアは，Ousterhout によって提案された[23]．これにより,「スピン戦略」に関する研究が行われるようになった．Karlin らは，7 つのスピン戦略についてロック待ち時間の分散と処理時間の調査を行った[32]．Lim らは，さまざまな同期のパターンについてロック待ち時間の特性を調べ，最適なロック手法を導き出そうとした[33]．4 章で述べたとおり，Java のバイモーダルロックは，ハイブリッドロックをメモリ効率良く実現するためのものであり，これらのスピン戦略の研究結果を反映させることで，より性能を上げられる可能性がある．

6.3 Java ロックの高速化

Java の登場によって，ロックのアルゴリズムが再び脚光をあびるようになった．Java ではロックが頻繁に行われ，特に初期の処理系ではそのオーバヘッドがかなり大きかった[34]ためである．

Java ロックの高速化は，ランタイムによる手法とコンパイラによる手法に大別できる．前者は，ロック処理のアルゴリズム自体を改良して高速化しようというもので，予約ロックもこの範疇に入る．後者は，コンパイル時の解析で不要なロック処理を取り去ってしまおうというものである．

6.3.1 ランタイムによる手法

Bacon らは，Java ではロックはほとんど衝突していないことを発見し,「Thin ロック[2]」を開発した．これが，バイモーダルロックの最初のものである．衝突のない間は，ロックワードをcompare_and_swap で書き換えるだけでロックを獲得できるようになり，性能は大きく向上した．

Thin ロックでは，いったん衝突が起こってファットモードに移行したロックは，それ以降サスペンドロックで処理が行われる．これに対し，Java では，衝突が起こったとしても一時的である場合が多いということを発見し，改良を行ったのが，4 章でもふれた Tasuki ロック[3]である．この手法では，スレッド間の衝突がなくなった段階でロックワードがフラットモードに戻され，再び高速なロック処理が可能となる．Tasuki ロックは，Gagnon らの SableVM [24]でも彼ら独自の修正と共に用いられている．

ほかにも，Meta ロック[4]や，Relaxed ロック[5]などの Java ロック手法が提案されている．いずれの手法も，衝突がない状態では 1 つあるいは 2 つの不可分命令（と付随する数命令）でロック処理が行えるようになっている．

Tasuki ロックが衝突しているロックの挙動に注目したのに対し，衝突しないロックの挙動に着目したの

が，2 章でも述べた予約ロック[6]である．Java では，衝突しないロックの多くは，実は特定のスレッドからしか用いられていないという発見に基づき，ロックに「予約」という概念を導入した．そして，予約スレッドは不可分命令なしでロックを行える手法を提案している．

本論文は，この研究を発展させたものである．まず，ロック実装の基礎となるスピンロックに非対称性を導入し，この「非対称スピンロック」をバイモーダルロックに組み込むことで，予約解除が不要な新しい予約ロックが実現できることを示した．

6.3.2 コンパイラによる手法

次に，コンパイラによるロック除去手法についてあげる．Java において最もメジャーなロック除去手法は，脱出解析（Escape Analysis）[35]により，生成スレッドからしか見えないオブジェクトを発見し，そのオブジェクトに対するロック処理をすべて省略してしまうというもので，多くのアルゴリズムが提案されている[7)〜12]．この手法は，プログラム全体の挙動をコンパイル時に解析できる静的な処理系ではかなり有効であるが，Java は実行中のクラスロードなどが可能な動的言語であり，その環境下では仮想関数の呼び出しを完全に解析することは難しいため脱出解析にも限界がある．

別のコンパイラ手法としては，再帰ロックを除去するというものがある．たとえば，`synchronized` メソッドに別の`synchronized` メソッドをインラインしたときに，同期の対象が同じオブジェクトであることを確定できる場合は内側のロック処理を除去することができる．

これらの，コンパイラによるロック除去は，5 章の評価に用いた Java 処理系の JIT コンパイラ[26]でも行われている．しかし，表 1 から，それでも依然として多くのロックが残っていることが分かる．

7. おわりに

本論文では，予約に基づく新しい Java ロックの実装法を述べた．これにより，特定のスレッドがロックを予約し不可分命令を用いない高速なロック処理を行うことが可能となる．新しい実装法では，従来の予約ロック実装で問題となる可能性のあった，予約解除という重い処理が不必要である．またこれにより，予約の成功率も向上している．

新しい手法のベースとなるのは，予約の概念を導入した非対称なスピンロックである．これは，Dekker のアルゴリズムの発想を非対称に拡張し，1 つのスレッドに限って，単純なメモリの読み書きだけでスピンロックの獲得を可能としたものである．この非対称スピンロックを通常の Java ロックに組み込むことで，新しい予約ロックを実現した．

提案したロック手法を商用の Java 処理系に実装し，評価を行った．まず，マイクロベンチマークにより，予約が成功している場合は従来の予約ロック実装にせまる性能が発揮できており，予約解除のペナルティが存在せず，予約が成功するケースが増えていることを示した．マクロベンチマークでは，予約を行わない通常の Java ロック実装に比べて，最大 7.4%性能が向上した．さらに，従来の予約ロック実装では性能が低下した 2 つの科学計算ベンチマークに対しても，性能が向上することを確認した．

本論文の主な寄与点としては，以下の 3 点があげられる．

- 非対称スピンロックの提案
  予約という発想をスピンロックのレベルまで持ち込み，特定のスレッドに限り処理を高速化できる非対称なスピンロック機構を提案した．
- Java 向けの新しい予約ロックの実現
  非対称スピンロックを通常の Java ロックアルゴリズムに組み込むことで，予約スレッドは非常に高速なロック処理を行え，また予約解除のオーバヘッドのない Java ロックを実現できることを示した．
- 実装と性能評価
  新しい予約ロック手法を商用の Java 処理系に実装し，実ベンチマークによる情報収集と性能評価を行った．

今後の研究課題の 1 つに,「予約ポリシー」の検討があげられる．今回性能評価に用いた予約ロックの実装はいずれも，最初にロックを行ったスレッドに予約を与える方式になっている．これを修正することで，性能向上を試みる．

たとえば，従来の予約ロックでは，複数のスレッドからロックされる（つまり予約解除が起こる）可能性の高いオブジェクトは最初から予約を行わないという手法が考えられる．今回問題となった科学計算ベンチマークの性能低下は，この改良で解消できる可能性がある．また，新しい予約ロックでは，オブジェクトを最も頻繁にロックするスレッドが分かっている場合は，そのスレッドに最初から予約を与えておくという手法も考えられる．

ロックアルゴリズム自体の改良としては，予約スレッドを動的に変更していく手法や，環境に応じて複数の手法を切り替えるやり方なども研究の価値があるだ

ろう．

近年，Web サービスなどで，Java を用いた大規模なアプリケーションが利用されるようになってきている．このような環境でのロックパターンの調査，およびロック手法を含めたスレッド処理方式の改良なども，今後の課題としてあげられる．

謝辞 ふだんより有用な意見をいただいている，IBM 東京基礎研究所・ネットワークコンピューティングプラットフォームグループの皆様に感謝します．また，ベンチマークプログラムの使用を快諾してくれた，Alexandru Salcianu と Martin Rinard 両氏に感謝します．

参 考 文 献

1) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison Wesley (1996).
2) Bacon, D.F., Konuru, R., Murthy, C. and Serrano, M.: Thin Locks: Featherweight Synchronization for Java, *Proc. ACM PLDI '98*, pp.258–268 (1998).
3) Onodera, T. and Kawachiya, K.: A Study of Locking Objects with Bimodal Fields, *Proc. ACM OOPSLA '99*, pp.223–237 (1999).
4) Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y.S. and White, D.: An Efficient Meta-lock for Implementing Ubiquitous Synchronization, *Proc. ACM OOPSLA '99*, pp.207–222 (1999).
5) Dice, D.: Implementing Fast Java Monitors with Relaxed-Locks, *Proc. USENIX JVM '01*, pp.79–90 (2001).
6) Kawachiya, K., Koseki, A. and Onodera, T.: Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations, *Proc. ACM OOPSLA '02*, pp.130–141 (2002).
7) Aldrich, J., Chambers, C., Sirer, E.G. and Eggers, S.: Static Analyses for Eliminating Unnecessary Synchronization from Java Programs, *Proc. 6th Int'l Static Analysis Symposium* (*SAS '99*), pp.19–38 (1999).
8) Blanchet, B.: Escape Analysis for Object-Oriented Languages: Application to Java, *Proc. ACM OOPSLA '99*, pp.20–34 (1999).
9) Bogda, J. and Hölzle, U.: Removing Unnecessary Synchronization in Java, *Proc. ACM OOPSLA '99*, pp.35–46 (1999).
10) Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V.C. and Midkiff, S.: Escape Analysis for Java, *Proc. ACM OOPSLA '99*, pp.1–19 (1999).
11) Whaley, J. and Rinard, M.: Compositional Pointer and Escape Analysis for Java Programs, *Proc. ACM OOPSLA '99*, pp.187–206 (1999).
12) Ruf, E.: Effective Synchronization Removal for Java, *Proc. ACM PLDI '00*, pp.208–218 (2000).
13) Anderson, T.E.: The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors, *IEEE Trans. Parallel and Distributed Systems*, Vol.1, No.1, pp.6–16 (1990).
14) Dijkstra, E.W.: Co-operating Sequential Processes, *Programming Languages*, Genuys, F. (Ed.), pp.43–112, Academic Press, New York (1968).
15) Dijkstra, E.W.: Solution of a Problem in Concurrent Programming and Control, *Comm. ACM*, Vol.8, No.9, p.569 (1965).
16) Peterson, G.L.: Myths about the Mutual Exclusion Problem, *Information Processing Letters*, Vol.12, No.3, pp.115–116 (1981).
17) Lamport, L.: A Fast Mutual Exclusion Algorithm, *ACM Trans. Computer Systems*, Vol.5, No.1, pp.1–11 (1987).
18) Adve, S.V. and Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial, *IEEE Computer*, Vol.29, No.12, pp.66–76 (1996).
19) Culler, D.E., Singh, J.P. and Gupta, A.: *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann (1999).
20) Greenwald, M. and Cheriton, D.: The Synergy Between Non-blocking Synchronization and Operating System Structure, *Proc. USENIX OSDI '96*, pp.123–136 (1996).
21) Motorola Inc.: *M68040 User's Manual* http://e-www.motorola.com/files/32bit/doc/ref_manual/MC68040UM.pdf
22) Hoare, C.A.R.: Monitors: An Operating System Structuring Concept, *Comm. ACM*, Vol.17, No.10, pp.549–557 (1974).
23) Ousterhout, J.K.: Scheduling Techniques for Concurrent Systems, *Proc. 3rd Int'l Conference on Distributed Computing Systems*, pp.22–30 (1982).
24) Gagnon, E.M. and Hendren, L.J.: SableVM: A Research Framework for the Efficient Execution of Java Bytecode, *Proc. USENIX JVM '01*, pp.27–39 (2001).
25) IBM developerWorks: Java Technology Zone. http://www.ibm.com/developerworks/java/
26) Ishizaki, K., Takeuchi, M., Kawachiya, K., Suganuma, T., Gohda, O., Inagaki, T., Koseki, A., Ogata, K., Kawahito, M., Yasue, T., Ogasawara, T., Onodera, T., Komatsu, H. and Nakatani, T.: Effectiveness of Cross-Platform

Optimizations for a Java Just-In-Time Compiler, *Proc. ACM OOPSLA '03*, pp.187–204 (2003).
27) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/
28) Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd ACM Int'l Symposium on Computer Architecture* (*ISCA '95*), pp.12–23 (1995).
29) Salcianu, A. and Rinard, M.: Pointer and Escape Analysis for Multithreaded Programs, *Proc. ACM PPoPP '01*, pp.12–23 (2001).
30) Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *Proc. ACM PLDI '98*, pp.212–223 (1998).
31) Mellor-Crummey, J.M. and Scott, M.L.: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Trans. Computer Systems*, Vol.9, No.1, pp.21–65 (1991).
32) Karlin, A.R., Li, K., Manasse, M.S. and Owicki, S.: Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor, *Proc. ACM SOSP '91*, pp.41–55 (1991).
33) Lim, B.-H. and Agarwal, A.: Waiting Algorithms for Synchronization in Large-Scale Multiprocessors, *ACM Trans. Computer Systems*, Vol.11, No.3, pp.253–294 (1993).
34) Armstrong, E.: HotSpot: A New Breed of Virtual Machine (1998). http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html
35) Park, Y.G. and Goldberg, B.: Escape Analysis on Lists, *Proc. ACM PLDI '92*, pp.116–127 (1992).

付　　録

A.1 非対称スピンロックを用いた新しい予約ロック

4 章でふれた Tasuki ロックのアルゴリズムと，非対称スピンロックの組み込み方法について詳細を示す．

図 12 は，Tasuki ロックにおけるロックワードの使用法を示したものである．4 章でも述べたように，Shape ビットによりフラットモードとファットモードを区別する．**図 13** が，Tasuki ロックのアルゴリズムの概要である．なおここでは，説明を簡単にするため，フラットモードでの再帰ロックや，エラーチェックの機能については省略している．

オブジェクトのロックを獲得するには，`Java_lock_acquire` を使用する．この関数はまず，`try_acquire` でスピンロック獲得を試みる（21 行目）．ロックワードが 0 であれば，スピンロックが成功し処理は終了す

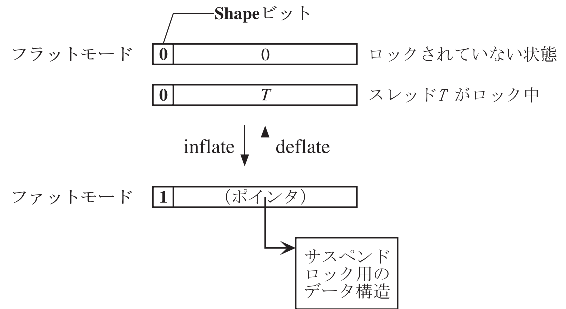

図 12 Tasuki ロックにおけるロックワードの意味
Fig. 12 Semantics of the lockword in Tasuki lock.

```
 1 #define SHAPE_BIT  0x80000000
 2
 3 int try_acquire(volatile word_t *lock) {
 4   return compare_and_swap(lock, 0, thread_id());
 5 }
 6
 7 void release(volatile word_t *lock) {
 8   *lock = 0;
 9 }
10
11 void inflate(volatile word_t *lock, monitor_t *mon) {
12   *lock = SHAPE_BIT | (word_t)mon; /* Shape ビットを立てる */
13 }
14
15 void deflate(volatile word_t *lock, monitor_t *mon) {
16   *lock = 0; /* Shape ビットを落とし，スピンロックも解放 */
17 }
18
19 void Java_lock_acquire(Object *obj) {
20   /* フラットモード */
21   if (try_acquire(&obj->lock) == SUCCESS) return;
22
23   /* ファットモード，もしくはファットモードへの移行 */
24   monitor_t *mon = obtain_monitor(obj);
25   monitor_enter(mon);
26   while ((obj->lock & SHAPE_BIT) == 0) {
27     obj->contention = 1; /* 衝突があったことを記録 */
28     if (try_acquire(&obj->lock) == SUCCESS) {
29       obj->contention = 0;
30       monitor_notify_all(mon);
31       inflate(lock, mon);
32     } else
33       monitor_wait(mon);
34   }
35 }
36
37 void Java_lock_release(Object *obj) {
38   /* フラットモード */
39   if ((obj->lock & SHAPE_BIT) == 0) {
40     release(&obj->lock);
41     if (obj->contention == 1) { /* 衝突があった場合 */
42       monitor_t *mon = obtain_monitor(obj);
43       monitor_enter(mon);
44       if (obj->contention == 1) monitor_notify(mon);
45       monitor_exit(mon);
46     }
47     return;
48   }
49
50   /* ファットモード，可能ならフラットモードへの移行 */
51   monitor_t *mon = obtain_monitor(obj);
52   if (no thread waiting on obj)
53     if (better to deflate)
54       deflate(&obj->lock, mon);
55   monitor_exit(mon);
56 }
57
58 monitor_t *obtain_monitor(Object *obj) {
59   word_t l = obj->lock;
60   if ((l & SHAPE_BIT) != 0)
61     return (monitor extracted from l);
62   else
63     return (monitor associated to obj);
64 }
```

図 13 Tasuki ロックのアルゴリズム
Fig. 13 Algorithm of Tasuki lock.

る．スピンロックが失敗した場合は，サスペンドロックの獲得が試みられる（25 行目）．この際，ロックワードがフラットモードだった場合は，`inflate` を呼び，ファットモードに遷移する．

オブジェクトのロックを解放するには，`Java_lock_release` を使用する．この関数はまずロックワードのモードをチェックする．フラットモードであった場合は`release`でスピンロックを解放（40 行目）し，衝突が起きている場合は追加処理を行う．ファットモードであった場合は，サスペンドロックの解放を行う（55 行目）が，その直前にフラットモードへの遷移が可能かどうかチェックし，可能なら`deflate`を呼ぶ．

モード遷移のために，オブジェクトヘッダ内の `contention` フラグと，サスペンドロックを利用した通知処理を行っている．この詳細については，元論文[3)]を参照してもらいたい．

Tasuki ロックで使用される`try_acquire`と`release`は，図 3 で示した`compare_and_swap`ベースのものである．ただし，スピンロック時に Shape ビットのチェックが同時に行われ，これが 1 であった場合，スピンロック獲得は必ず失敗するようになっている．また，モード遷移時にロックワードを変更するために，`inflate`，`deflate` という関数を新たに定義している．これらの関数を非対称スピンロック版で置き換えることで，予約をサポートした新しい Java ロックを実現できる．

置き換えは，3.4 節に示した「1 ワード版」をベースに行うが，データレイアウトを図 9 のように修正し，Shape ビットを組み込んでいる☆．

図 14 は，このデータ構造を用い Tasuki ロックに組み込むために微修正を行ったアルゴリズムである．傍線は，図 8 から変更された部分を示す．なお，説明の簡易化のため，`rcnt` フィールドの処理については省略している．5 章で測定に用いたバージョンでは，このフィールドも利用し，8 レベルまでの再帰ロックはフラットモードのまま行えるようになっている．

上でも述べたように，Shape ビットが 1 の場合，`try_acquire` 関数は必ず失敗しなければならない．そのため，ロック可能性をチェックする部分では，このビットを含めてチェックを行っている（16，20，22，32 行目）．`release` はフラットモードでしか呼ばれない

☆ ここで示したレイアウトでは，owner および other フィールドに 10 ビットしか割り振っていないため，同時に存在できる Java スレッドの数が 1023 個に制限されてしまう．図 9 では未使用としている 7 ビットも owner フィールドに使用し，rcnt フィールドを別ワードに移すことで，スレッド ID を 15 ビットまで拡張可能である．

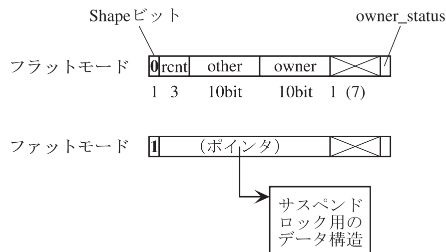

図 9　（再掲）新しい予約ロックのためのロックワードの構造
Fig. 9　(re-shown) Lockword structure of the new reservation-based Java lock.

```
 1|#define SHAPE_BIT   0x80000000
 2|#define OWNER_MASK  0x0003ff00
 3|#define OTHER_MASK  0x0ffc0000
 4
 5|#define SHAPE_OWNER_MASK (SHAPE_BIT | OWNER_MASK)
 6|#define SHAPE_OTHER_MASK (SHAPE_BIT | OTHER_MASK)
 7
 8 #define OWNER_STATUS_BYTE(lock)  (((char  *)lock)[0])
 9 #define OTHER_SHORT(lock)        (((short *)lock)[1])
10|#define OWNER_TO_OTHER(tid)      ((tid)<<10)
11
12 int try_acquire(volatile word_t *lock) {
13   word_t l = *lock;
14   word_t myID = thread_id();
15
16|  if ((l & SHAPE_OWNER_MASK) == 0){    /* 予約の確定とロック */
17     word_t locked = myID | 1;
18     return compare_and_swap(lock, 0, locked);
19   }
20|  else if ((l & SHAPE_OWNER_MASK) == myID) { /* 予約スレッド */
21     OWNER_STATUS_BYTE(lock) = 1;
22|    if ((*lock & SHAPE_OTHER_MASK) == 0)
23       return SUCCESS;
24     else {
25       OWNER_STATUS_BYTE(lock) = 0;
26       return FAILURE;
27     }
28   }
29   else {                            /* 予約スレッド以外の場合 */
30     word_t unlocked = l & OWNER_MASK;
31     word_t locked = unlocked | OWNER_TO_OTHER(myID);
32     return compare_and_swap(lock, unlocked, locked);
33|                  /* Shape ビットが立っていると失敗する */
34   }
35 }
36
37 void release(volatile word_t *lock) {
38   word_t l = *lock;
39
40   if ((l & OWNER_MASK) == thread_id())
41     OWNER_STATUS_BYTE(lock) = 0;
42   else {
43     word_t unlocked = l & ~OTHER_MASK;
44     OTHER_SHORT(lock) = OTHER_SHORT(&unlocked);
45   }
46 }
47
48|/* mon はロックワードの 8〜30 ビット目におさまるものとする */
49|void inflate(volatile word_t *lock, monitor_t *mon) {
50|  mon->spin_owner = *lock & OWNER_MASK; /* owner 情報を保存 */
51|  if (mon->spin_owner == thread_id())
52|    *lock = SHAPE_BIT | (word_t)mon;
53|  else while (1) {
54|    word_t l = *lock;
55|    word_t inflated = SHAPE_BIT | (word_t)mon | (l & 1);
56|    if (compare_and_swap(lock,l,inflated) == SUCCESS) return;
57|  }
58|}
59|
60|void deflate(volatile word_t *lock, monitor_t *mon) {
61|  if (mon->spin_owner == thread_id())
62|    *lock = mon->spin_owner;              /* owner 情報を復元 */
63|  else while (1) {
64|    word_t l = *lock;
65|    word_t deflated = mon->spin_owner | (l & 1);
66|    if (compare_and_swap(lock,l,deflated) == SUCCESS) return;
67|  }
68|}
```

図 14　Tasuki ロックに組み込むための非対称スピンロック
Fig. 14　Asymmetric spin lock adjusted for being employed in Tasuki lock.

ため，この修正は不要である．

`inflate`，`deflate` では，3 つの点に注意が必要である．まず，モード遷移の際に予約スレッド以外が `owner_status` フィールドを書き換えてはならない．そのために，`compare_and_swap` を用いている（56，66 行目）が，この部分は頻繁に通るパスではないので性能上の問題はない．また，`inflate` で予約スレッドの情報を保存（50 行目）し，`deflate` ではそれを復元（62，65 行目）しなければならない．さらに，`deflate` は Shape ビットを落とすだけでなくスピンロックの解放も行うので，予約スレッドの場合は `owner_status` フィールド，それ以外の場合は `other` フィールドをクリアしなければならない．

これら 4 つの関数で図 13 の対応する関数（3〜17 行目）を置き換えることで，衝突がない場合には予約スレッドは不可分命令なしに処理が行え，また予約解除が起こらない Java ロックが実現できる．

(平成 15 年 9 月 22 日受付)
(平成 15 年 11 月 14 日採録)

河内谷清久仁（正会員）

1963 年生．1987 年東京大学大学院理学系研究科情報科学専門課程修士課程修了．同年日本アイ・ビー・エム（株）入社．以来，同社東京基礎研究所にて，オペレーティングシステムやマルチメディア処理システム，携帯情報システム，Java 処理系ランタイム等の研究に従事．現在，同研究所専任研究員．1994 年情報処理学会全国大会奨励賞受賞．ACM 会員．

古関　聰（正会員）

1969 年生．1998 年早稲田大学大学院理工学研究科電気工学専攻博士課程修了．同年日本アイ・ビー・エム（株）入社．以来，同社東京基礎研究所において，Java Just-In-Time コンパイラの開発に従事．工学博士．ACM 会員．

小野寺民也（正会員）

1959 年生．1988 年東京大学大学院理学系研究科情報科学専門課程博士課程修了．同年日本アイ・ビー・エム（株）入社．以来，同社東京基礎研究所にて，オブジェクト指向言語の設計および実装の研究に従事．現在，同研究所シニア・テクニカル・スタッフ・メンバー．第 41 回（平成 2 年後期）全国大会学術奨励賞，平成 7 年度山下記念研究賞，各受賞．理学博士．日本ソフトウェア科学会，ACM 各会員．