計算機アーキテクチャ　99－12
(1993. 3. 12)

マルチプロセッサリアルタイムシステムにおける プロセッサ内同期とプロセッサ間同期

高田 広章　　坂村 健

東京大学 理学部 情報科学科
〒113 東京都文京区本郷 7-3-1

あらまし　非対称型のマルチプロセッサアーキテクチャ上にリアルタイムシステムを構築する場合、各プロセッサにおける高速な割込応答性と、予測可能性のあるプロセッサ間同期を両立させることが重要である。本稿ではまず、既存のスピンロックアルゴリズムを用いて両者を両立させることが困難であることを示し、この問題を解決するために、新しいスピンロックアルゴリズムを提案する。提案するアルゴリズムは、各プロセッサにおける割込禁止時間を最小限にすると同時に、プロセッサ間ロックを取るまでの時間に上限が与えられる。さらに、提案したアルゴリズムの有効性を、実機を用いた実験による性能評価によって検証する。

和文キーワード

Intra-Processor vs. Inter-Processor Synchronizations in Real-Time Multiprocessor Systems

Hiroaki Takada　　Ken Sakamura

Department of Information Science, Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113, Japan

Abstract　When a real-time system is implemented on an asymmetric multiprocessor architecture, both fast interrupt latency of each processor and predictable inter-processor synchronization are important. In this paper, we first show that they cannot be compatible using conventional algorithms for spin locks, and propose a new algorithm to solve this problem. Using the new algorithm, the blocking time on an inter-processor lock has an upper bound, while the interrupt masking duration on each processor is kept short. The effectiveness of the algorithm is demonstrated through empirical performance results.

英文 key words

1 はじめに

組み込みリアルタイムシステムの応用分野が広がるにつれて、大規模かつ高性能なリアルタイムシステムに対する要望が高まっている。とりわけ、プラント制御や航空機制御などの大規模制御システム、パケット交換機やネットワークルータなどの通信制御システムといった応用分野で、システムの大規模化、高性能化に対する要求が著しい。これらの応用分野では、多数の外部入出力装置がシステムに接続され、それらからの事象に対して短い時間でレスポンスを返す必要がある。このような要求に対して、マルチプロセッサシステムの採用は、有力な方法と考えられる。

これらの応用の多くにおいては、各入出力装置に伴って必要となる処理量を見積ることができるため、各入出力装置に関する処理を行なうプロセッサを限定し、入出力インタフェースをそのプロセッサのローカルバスに接続する非対称型のマルチプロセッサ構成が採られることが多い。非対称型のシステム構成においては、入出力装置を扱うプログラムのコード領域やデータ領域を、その入出力装置を処理するプロセッサのローカルメモリに置くことができる。そのため、共有バス (ないしは、インタコネクションネットワーク。以下同じ) の使用頻度を下げることができ、高速な共有バスや高価なキャッシュ機構を省略することで、システムのコストダウンが図れる。また、共有バスアクセスの衝突頻度が下がることは、リアルタイムシステムに不可欠な予測可能性を向上させる上からも利点が大きい。

現在我々は、(主として) このような非対称型のマルチプロセッサリアルタイムシステムに適用することを考慮して、組み込みリアルタイムシステム用カーネル仕様 ITRON のマルチプロセッサシステムへの拡張である ITRON-MP 仕様の検討と、その実験的な実装を行なっている [1]。

非対称型マルチプロセッサ上のリアルタイムシステムにおいて、外部事象に対する優れた応答性を実現するためには、各プロセッサが外部割込に対して高速に応答できることが不可欠である。外部割込に対する応答性を下げる主な要因として、同一のプロセッサ内で動作する他のタスクとの排他制御を実現するために、割込禁止区間を作ることが挙げられる。

一方、非対称型マルチプロセッサシステムにおいては、他のプロセッサで動作するタスクとの同期のために、スピンロックを用いた排他制御機構が用いられる。このプロセッサ間同期については、プロセッサ内同期ほどの高速な応答性は必要ないが、システム動作を予測可能にするには、プロセッサ間同期に関しても実行時間に上限のあるスピンロックアルゴリズム[1]を用いる必要がある。

本稿ではまず、既存のスピンロックアルゴリズムを用いて高速な割込応答性と実行時間が予測可能なプロセッサ間同期を両立させることが困難であることを示し、この問題を解決するために、新しいスピンロックアルゴリズムを提案する。提案するアルゴリズムでは、各プロセッサにおける割込禁止時間を最小限にすると同時に、ロックを取るまでの時間に上限を与えることができる。さらに、その性能を実機を用いた実験により評価し、有効性を示す。

[1]本稿で、スピンロックアルゴリズムの実行時間に上限があるという場合には、共有バスのアクセス時間に上限があることを仮定する。

2 マルチプロセッサリアルタイムシステムにおけるスピンロック

2.1 既存のアルゴリズム

マルチプロセッサシステムのためのスピンロックアルゴリズムについては、各種の前提の下で、多くの研究がなされてきた。本研究では、ハードウェアが持つマルチプロセッササポート機能として共有メモリの単一ワードの不可分なリードモディファイライト機構があることを前提として、スピンロックアルゴリズムの議論を行なう。リードモディファイライト機構を利用する代表的な命令としては、test_and_set, fetch_and_store (swap), fetch_and_add, compare_and_store などが挙げられる。

この前提の下で、J. M. Mellor-Crummey らは主要なスピンロックアルゴリズムを以下の 4 通りに分類している [2]。

- test-and-set ロック

  test_and_set 命令を用いて、共有メモリ上のロック状態を示す変数をセットする方法。セットに失敗した場合のポーリングの方法に、多くのバリエーションがある。

- チケットロック

  fetch_and_add 命令を用いて、共有メモリ上のロックを取る順番を示す変数をアクセスして、自分がロックを取る順番を得る方法。自分の順番が得られれば、ロックを取っている順番を示す変数が、自分の順番を示すまで待つ。待つ際のポーリングの方法に、いくつかのバリエーションがある。

- 配列を用いたキューイングロック

  ロックを取ろうとしたプロセッサを、配列を用いたキューにつないでいく方法。fetch_and_add 命令を用いるアルゴリズム [3] と、fetch_and_store 命令を用いるアルゴリズム [4] が提案されている。これらのアルゴリズムは、一貫性を保持するキャッシュを備えたマシン上では、共有バスのアクセス頻度を少なくし、バス競合の問題を解決することができる。

- ポインタを用いたキューイングロック

  ロックを取ろうとしたプロセッサを、ポインタを用いたキューにつないでいく方法。fetch_and_store 命令と compare_and_store 命令を用いる MCS ロックアルゴリズムが提案されている [2]。MCS ロックのアルゴリズムを図 1 に示す。ここで、キーワード **shared** はその変数がシステム全体で 1 つだけ作られることを、**private** は変数がプロセッサ毎に用意されることを示す。また、compare_and_store は 3 つの引数を取る関数で、第 1 引数の

```
type qnode = record
    next: pointer to qnode;
    locked: (Released, Locked)
end;

shared var L: pointer to qnode;

private var I: qnode;
private var pred: pointer to qnode;

I.next := NIL;
pred := fetch_and_store(L, &I);
if pred != NIL then
    I.locked := Locked;
    pred→next := I;
    repeat until I.locked = Released
end;
//
// critical section.
//
if I.next = NIL then
    if compare_and_store(L, &I, NIL) then
        goto exit
    end;
    repeat until I.next != NIL
end;
I.next→locked := Released;
exit:
```

図 1: MCS ロック

ポインタで指されるメモリの内容を第2引数と比較し、一致していた場合第3引数をそのメモリ領域に書き込み、True を返す。一致していなかった場合は、何もせずに False を返す。

MCS ロックでは、スピンロック待ちの際に参照する変数(図1の I.locked)を各プロセッサのローカルメモリ[2]上に置くことで、複数のプロセッサが同時にロックの取得を試みた場合でも、スピンロックに伴う共有バスへのアクセス回数に上限が与えられる。さらに、この上限はプロセッサ数に依存せずに決まる。そのため MCS ロックは、キャッシュを持たないマシンで特に有効なアルゴリズムとなっている。

これらのアルゴリズムのうち、test-and-set ロックについては、ロックが取れるまでの時間に上限が無いため、リアルタイムシステムには不適当と考えられる。

2.2 スピンロックと割込禁止

マルチプロセッサシステムにおいて、共有メモリ上のデータへアクセスするためのクリティカルセクションに入

[2]ここでいうローカルメモリとは、あるプロセッサからは共有バスを経由せずにアクセスでき、他のプロセッサからは共有バス経由でアクセスできるメモリのことである。

```
acquire_lock(L);
disable_interrupt;
//
// critical section.
//
enable_interrupt;
release_lock(L);
```

図 2: プロセッサ内同期優先方式

```
disable_interrupt;
acquire_lock(L);
//
// critical section.
//
release_lock(L);
enable_interrupt;
```

図 3: プロセッサ間同期優先方式

るまでの時間に上限があるためには、スピンロックアルゴリズムにおいて他のプロセッサを待つ回数に上限があることに加えて、他のプロセッサがロックを保持する時間に上限がなければならない。このうち、他のプロセッサを待つ回数については、前節で紹介したチケットロックないしはキューイングロックを用いることで、上限が与えられる。一方、プロセッサがロックを保持する時間に関しては、割込サービスとの関係で以下のような問題がある。

非対称型のマルチプロセッサシステムにおいては、外部入出力装置からの割込は、プロセッサ毎に発生する。複数の入出力インタフェースを1つのプロセッサに接続している場合には、それらからの割込は独立に発生する場合が多く、多重割込を考えた場合には、割込サービス時間の最大値はかなり大きい値になるのが通常である。そのため、プロセッサがロックを保持する時間に現実的な上限を設けるためには、ロックを保持する間は割込のサービスを禁止する必要がある。

この際に、割込禁止区間の開始とプロセッサ間のスピンロックのどちらを先に実行するかという問題が起こる。図2 の方法は、プロセッサ間のロックを取った後で割込禁止処理をする方法であるが、この方法では、プロセッサ間ロックを取る処理と割込禁止処理との間で割込が発生する可能性があり、上記の目的を達成できない。一方、図3の方法は、割込禁止処理を行なった後でプロセッサ間ロックを取る方法である。この方法を用いると、上記の目的通りロックを保持する間に外部割込のサービスが行なわれることはないが、スピンロックの実行時間が割込禁止区間に含まれるために、割込禁止時間がプロセッサの数に依存して長くなることになる。

この問題を解決するためには、プロセッサ間のロックを待つ間にも割込要求を受け付け、ロックが取れた時刻以降は割込禁止状態を保つようにする必要がある。より具体的

```
disable_interrupt;
while test_and_set(L) = Locked do
    if interrupt_requested then
        enable_interrupt;
        // service interrupt.
        disable_interrupt
    else
        delay
    end
end;
//
// critical section.
//
L := Released;
enable_interrupt;
```

図 4: 中断可能な test-and-set ロック

には、プロセッサ間ロックの取得待ちループの間で、割込リクエストがあるかをチェックし、リクエストがある場合にはスピンロックを中断して割込サービスを行なう方法が考えられる。この方法は、スピンロックを開始する際に共有データを操作しない test-and-set ロックには容易に適用することができる。test-and-set ロックを、中断可能となるよう改良したアルゴリズムを図 4 に示す [5]。

一方、同様の改良をチケットロックないしはキューイングロックにそのまま適用することはできない。次節では、キューイングロックを、中断可能となるよう改良したアルゴリズムについて述べる。

3 中断可能なキューイングロック

以下では、ロックを取れるまでの時間に上限のあるスピンロックアルゴリズムを、スピンロック中に割込サービスが可能となるように改良したアルゴリズムを提案する。

ロックを取れるまでの時間に上限のあるアルゴリズムはいずれも、自プロセッサがロックを取る順番を、スピンロックを開始する時点で共有データを操作することにより決定する。そのため、ロックを待っている間に割込リクエストを検出した場合に単に割込サービスルーチンに分岐するだけでは、自分がロックを取れる順番が来た際にすぐにクリティカルセクションの実行を開始できないため、プロセッサがロックを保持する時間に残りの割込サービス時間が含まれてしまうことになる。そこで、割込リクエストを検出した場合、割込サービスを開始する前に、共有データを操作することにより自プロセッサが割込サービス中であることを他のプロセッサに知らせる必要がある。

図 5 に、MCS ロックをベースに、割込サービスのために中断可能となるように改良したアルゴリズムを示す。ここで、CAS は compare_and_store の省略記法である。

このアルゴリズムでは、I.locked にスピンロック中断中であることを示す値 (Preempted) を書き込むことで、プロセッサが割込サービスに入ったことを他のプロセッサ

```
    type qnode = record
        next: pointer to qnode;
        locked: (Released, Locked, Preempted, R_P)
    end;

    shared var L: pointer to qnode;

    private var I: qnode;
    // I.locked must be initialized to Released.
    private var pred, succ: pointer to qnode;

    disable_interrupt;
retry:
    I.next := NIL;
    pred := fetch_and_store(L, &I);
    if pred = NIL then goto acquired end;
    I.locked := Locked;
    pred→next := I;
    if CAS(&pred→locked, R_P, Released) then
        I.locked = Released;
        goto acquired
    end;
    while (I.locked != Released) do
        if interrupt_requested and
                CAS(&I.locked, Locked, Preempted) then
            enable_interrupt;
            // service interrupt.
            disable_interrupt;
(#)         if ¬CAS(&I.locked, Preempted, Locked) and
(#)                 ¬CAS(&I.locked, R_P, Released) then
                goto retry
            end
        end
    end;
acquired:
    //
    // critical section.
    //
    if I.next = NIL then
        if CAS(L, &I, NIL) then goto exit end;
        repeat until I.next != NIL
    end;
    succ := I.next;
    while ¬CAS(&succ→locked, Locked, Released) do
(*)     if CAS(&succ→locked, Preempted, R_P) then
            if succ→next != NIL and
(**)                CAS(&succ→locked, R_P, Released) then
                succ := succ→next;
            else
                goto exit
            end
        end
    end;
exit:
    enable_interrupt;
```

図 5: 中断可能なキューイングロック

に知らせる。ロックを解放するプロセッサは、次にロックを取るべきプロセッサ P が割込サービス中であることを見つけると、P をロック待ちキューからはずし、さらに次のプロセッサにロックを渡す。P の他にロックを待っているプロセッサがない場合には、割込サービスが終了した時点ですぐにロックを取れることを示す値 (R_P: Released while Preempted) を P の locked 変数に書き込む。

一方、スピンロックを中断したプロセッサは、割込サービスが終了すると、I.locked を参照して割込サービスの間にロックの待ちキューからはずされたかどうかを調べる。すでにキューからはずされていた場合には、ロックを取るルーチンを再度先頭から実行する。はずされていなかった場合には、I.locked をスピンロック中であることを示す値に書きかえる。また、キューに新たにつながるプロセッサは、キューの1つ前のプロセッサが割込サービス中にロックを取れている状態 (R_P 状態) であるかチェックし、R_P 状態の時にはそのプロセッサをキューからはずし、自分がロックを取る。

このアルゴリズムでは、あるプロセッサが割込サービス中にロックを取る順番が来た場合には、他のプロセッサによってロック待ちのキューからはずされるため、割込サービスを終了後にロックを取る処理を再度最初から実行する必要がある。この際に、ロックを取る順番が最後に移動することになり、他のプロセッサのクリティカルセクション実行を待つ回数が増えることになるが、その増加時間を割込サービス時間に含めて考えることにより、時間制約の解析を行なうことができる。

図5 のアルゴリズムは、各プロセッサの実行速度にいかなる仮定も置かない場合に、間違った動作を起こす可能性がある。具体的には、あるプロセッサが (*) の行の実行が終ってから (**) の行を実行するまで間に、succ で指される qnode 領域が再利用され、他のプロセッサによって succ→locked に R_P が代入された場合に、(**) の行が間違ったロックを解除する可能性がある。この問題点は、qnode 構造体に (*) と (**) の間を実行中であることを示す Boolean 型のフィールドを1つ追加し、ロック取得中の適切な場所でその変数を参照して (*) と (**) の間の実行が終るのを待つことで解決できる。また、timing-based な排他制御 [6] の考え方を用いると、各プロセッサの実行速度比に上限を置くことで、このままの形でアルゴリズムの正当性が示せると考えている。

以上では、MCS ロックをベースに改良したアルゴリズムを提案したが、配列を用いたキューイングロックについても同様の改良が可能である。一方チケットロックについては、プロセッサ毎に割り当てられたメモリ領域を持たないために、同様の方法で改良することはできない。

4 実験による性能評価

本節では、実機を用いた実験により、前節で提案したスピンロックアルゴリズム (これを、アルゴリズム1 と呼ぶ) の性能評価を行なう。比較対象として、割込禁止区間なしの MCS ロック (アルゴリズム2)、図3 の方式を MCS ロックに適用したもの (アルゴリズム3)、図4 のアルゴリズム (アルゴリズム4) を用いる。図4 のアルゴリズムでは、test_and_set を実行する間隔を一定時間とした[3]。

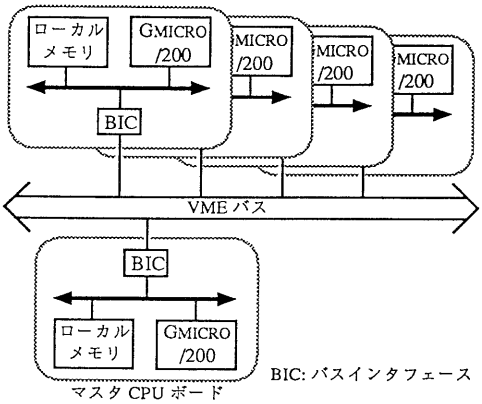

図 6: 評価環境の概要

4.1 評価環境の概要

実験には、プロセッサに TRON 仕様の CPU である GMICRO/200、共有バスに VME バスを用いたマルチプロセッサシステムを用いた。GMICRO/200 は、TRON 仕様の CPU としては最初に開発されたもので、20MHz のクロックで動作時に 6～7 MIPS 程度の性能を持つ。用いた CPU ボードは、ボード上のプロセッサと VME バスの両方からアクセスできるローカルメモリを持つ。また、CPU ボード上には、キャッシュメモリは持っていない。今回の実験では、すべてのプログラム領域およびプロセッサ毎に必要なデータ領域をこのメモリ上に置いた。システム全体で共有されるデータ領域については、スピンロックに加わらないマスタ CPU ボードのローカルメモリ上に置いた (図6)。

TRON 仕様 CPU は、マルチプロセッサをサポートするための命令として、bit_test_and_set, bit_test_and_clear, compare_and_store の3つの命令を持つ [7]。一方、MCS ロックおよびアルゴリズム1 を実装するために必要な fetch_and_store 命令は持っていないため、同等の機能を compare_and_store を用いたループにより実現した。そのため、MCS ロックの、共有バスへのアクセス回数に上限があるという特長は、今回の実験では実現されていない。また、アルゴリズム1 で、locked フィールドの値のコーディングを工夫することにより、図5 中の (#) で示した

[3] test-and-set ロックの場合、一般には test_and_set の実行間隔を指数的に増加させる方法がよいとされているが [2, 3]、リアルタイムシステムへの適用を考慮して一定間隔とした。

```
    for i := 1 to NoLoop do
(!)     acquire_lock_and_disable_interrupt;
        //
        // critical section.
        //
        release_lock;
(!!)    enable_interrupt;
        random_delay
    end;
```

図 7: 性能評価に用いたプログラム

2つの行の compare_and_store を1つの bit_test_and_set 命令により実現した。評価用プログラムは大部分を C 言語で記述し、これらのマルチプロセッササポート命令についてインラインアセンブラ機能を使って記述している。そのため、すべてのプログラムをアセンブラで記述する場合と比べて、この部分に若干のオーバーヘッドが生じている。

用いた CPU ボード上のローカルメモリは、ボード上のプロセッサからアクセスする場合と VME バスからアクセスする場合とではアクセスする際のアドレスが異なるため、アルゴリズム 1, 2, 3 で、next フィールドのポインタを読み / 書きする際にアドレスの変換が必要になった。またこの CPU ボードでは、実装上の制限のために、VME バスを経由してリードモディファイライトアクセスを行なっている間に、他のプロセッサがボード上のローカルメモリをアクセスした場合に、バスエラーを起こして衝突を回避する。今回の実験では、バスエラーから回復するためのオーバーヘッドを独立に測定し、バスエラーが起こった場合のデータにその分の補正を行なった。

VME バスのアービトレーション方式としては、ラウンドロビン方式を用いた。今回の実験では、プロセッサ数を、VME バスでラウンドロビンアービトレーションが可能な最大数である 4 個までの範囲で変化させて行なった。全般的に、プロセッサの速度と比較して、バスおよびメモリの速度はかなり遅く、VME バスを経由して他の CPU ボード上のローカルメモリをリード / ライトアクセスする際には、1μsec. 弱の時間がかかる。

4.2 評価方法

性能評価のための実験では、各プロセッサにタイマによる周期的な割込を発生させ、その割込サービスルーチン中で割込応答時間の分布を測定すると同時に、図 7 のプログラムを実行し、クリティカルセクション 1 回の実行にかかる時間分布 ((!) から (!!) までの実行にかかる時間の分布) を、途中で割込サービスを行なわなかった場合と行なった場合に分けて測定した。

クリティカルセクション内では、時間待ちのためのループ、正しく排他制御が行なわれていることを確認するための共有メモリアクセスおよび実行時間分布を計測するのための処理を行なう。プロセッサ間ロックを行なわない場合にクリティカルセクション 1 回の実行にかかる時間は、約 35μsec. である。また、割込サービスルーチン内にも、割込応答時間分布を計測する処理に加えて、時間待ちのためのループが入れてあり、1 回の割込サービスにかかる時間は約 40μsec. となっている。クリティカルセクションの実行を終ってから、次にクリティカルセクションに入るまでの時間には乱数要素を入れ、タイミング条件が変動するようにしている。その平均時間は、約 45μsec. である。また、タイマ割込の周期は約 1msec. であるが、プロセッサ毎に 1 ～ 2% 程度ずらすことで、各プロセッサでのタイマ割込の発生タイミングが同期しないようにした。測定の間は、タイマ以外の割込要因はマスクしている。

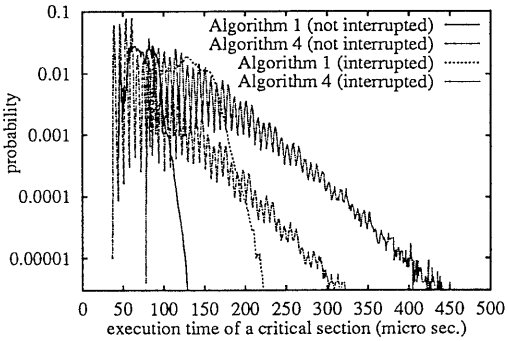

図 8: クリティカルセクション実行時間の分布

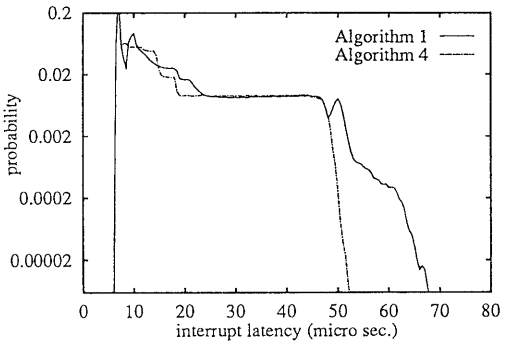

図 9: 割込応答時間の分布

4.3 評価結果

プロセッサ数が 4 個の場合の、アルゴリズム 1 および 4 による、クリティカルセクション 1 回あたりの実行時間の分布を図 8 に、割込応答時間の分布を図 9 に示す[4]。これらのグラフから、両方のアルゴリズムとも割込応答時間

[4] これらのグラフでは、縦軸 (確率密度) は対数目盛で取っていることに注意されたい。

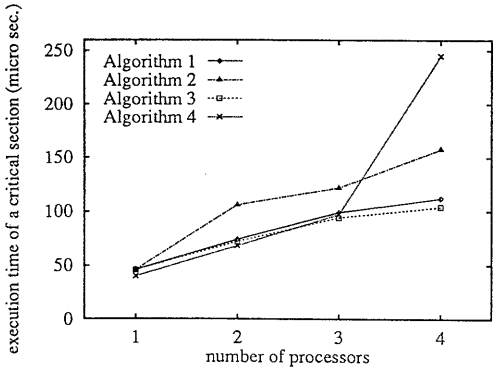

図 10: クリティカルセクション実行時間の比較

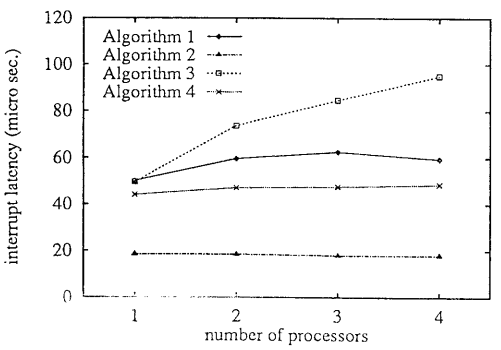

図 11: 割込応答時間の比較

については現実的な上限値があるのに対して、アルゴリズム 4 ではクリティカルセクション 1 回あたりの実行時間の上限値を定めにくい。これは、test-and-set ロックにおいては、各プロセッサがロックを取る順序が全くランダムに決まるという性質を反映している。図 8 で、アルゴリズム 4 の時間分布に小さい周期の増減が見られるが、これは、test_and_set を実行する間の待ち時間による効果である。

リアルタイムシステムの場合、平均実行時間により性能評価を行なうことは適当ではなく、本来は最悪の実行時間で評価すべきであると考えられる。しかし、スピンロックアルゴリズムの性能評価の場合には、マルチプロセッサシステムに不可避な非決定性のために、実験により最悪値を求めることは難しい。そこで以下では、最悪値に代えて、その時間内に実行が終らない確率 (ないしは、その時間内に割込応答が返らない確率) がある定数値以下になるような時間を用いて性能を評価する。つまり、その時間をデッドラインと置いた場合に、デッドラインミス率が設定した定数値となる。以下では、ミス率を 0.1% に設定した場合の結果を示す。

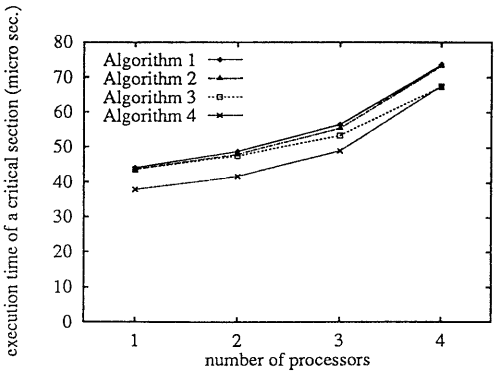

図 12: クリティカルセクション実行時間 (平均値) の比較

図 10 および図 11 は、プロセッサ数を 1 ～ 4 個まで変化させた場合の、各アルゴリズムの、上の方法で評価したクリティカルセクション 1 回あたりの実行時間 (途中で割込サービスを行なわなかった場合) および割込応答時間の変化を示す。

図 10 から、アルゴリズム 1 はプロセッサ数の増加に対して、クリティカルセクション 1 回あたりの実行時間がほぼリニアに増加し、スケーラビリティに優れていることがわかる。それに対して、アルゴリズム 2 では、他のプロセッサが割込サービスを行なう時間の影響を受けるために、全体に性能が悪くなっている。また、図 8 からもわかるように、アルゴリズム 4 では各プロセッサがロックを取る順番が全くランダムであるため、スケーラビリティが悪いという結果が得られる。

図 11 から、アルゴリズム 1 は、割込応答性に関してもスケーラビリティに優れていることがわかる。それに対してアルゴリズム 3 では、割込応答性がプロセッサ数に依存して悪くなる。割込応答性は、単一のプロセッサの応答性を決めるものであるから、システムのプロセッサ数に依存して変化することは好ましくない。

図 12 に平均値を使って評価したクリティカルセクション 1 回あたりの実行時間 (途中で割込サービスを行なわなかった場合) の比較を示す。平均実行時間で比較した場合、アルゴリズム 1 はアルゴリズム 4 に比べて 10 ～ 20% 程度、性能が悪くなっている。プロセッサ数の増加に対する振舞いは、各アルゴリズム間で有為な差は見られない。

5 アルゴリズムの適用性

前節まででは、クリティカルセクションを実行するために単一のロックを取ればよい状況を考えていた。しかし現実のシステムにおいては、ロック競合の頻度を下げるために、システムの共有資源をいくつかのロック単位に分けて管理するのが通常で、別単位に属する複数の資源をアクセスする場合には、複数のロックを取る必要がある。

このような状況においては、2 つめ以降のロックを待

つ間に割込要求があった場合に、それ以前に取得したロックについても中断可能な状態にする必要がある。提案したアルゴリズムは、このような状況に対応できるように拡張することが可能である。具体的には、2つめ以降のロックを待つ間に要求を検出した割込サービスに分岐する際に、それ以前に取得したロックを R_P 状態にし(ロック待ちプロセッサがある場合には、ロックを解放する)、割込サービスが終了した時点で R_P 状態にしたロックを元の状態に戻すことで実現できる(他のプロセッサにロックを取られていたら、再度そのロック待ちルーチンに戻る)。ただし、数多くのロックを取る必要がある場合には、オーバーヘッドが大きく実用的でなくなることが予想される。

割込禁止区間を短縮する方法の1つに、クリティカルセクションの途中で割込要求をチェックし、要求があった場合にはクリティカルセクションの実行を中断して割込サービスを優先させる方法がある(この場合、必要ならクリティカルセクションを先頭から再実行する)。このような方法も、提案したアルゴリズムを拡張することで容易に実現できる。

また、プロセッサ間ロックによるクリティカルセクションと、割込サービスルーチンとの排他制御が必要になる状況も考えられるが、提案したアルゴリズムはこのような状況に対しては、そのままの形で適用できる。

その他に、割込サービスルーチンの中で、他のプロセッサとの排他制御を行なう必要のある状況も考えられる。この場合、割込サービスにかかる時間が相当長くなることは避けられず、サービスしている割込の優先度以下の割込に対する応答性が著しく悪くなるため、極力避けるべきである。どうしても必要な場合には、提案したアルゴリズムを改造することでこのような状況にも対応することが可能である。

6 まとめと今後の課題

本稿では、非対称型マルチプロセッサ上にリアルタイムシステムを実現する際に必要となる、各プロセッサにおける割込禁止時間を最小限にすると同時に、ロックを取るまでの時間に上限のあるスピンロックアルゴリズムを提案し、その性能を実機を用いた実験により評価した。実験結果から、提案したアルゴリズムは当初の目的を満たす性質を備えていることが示された。

提案したアルゴリズムは、平均性能や割込応答性の面からは、test-and-set ロックをベースに改良したものと比べて性能が悪いが、実験環境が、fetch_and_store 命令を持たない、ポインタの変換が必要、といった、ポインタを用いたキューイングロックに不利な条件となっていることを考えると、適切なハードウェアを使用することで、性能差をさらに縮めることができると予想される。

今後の課題としては、提案したアルゴリズムの正当性の証明が挙げられる。特に、timing-based な排他制御の考え方を用いて、図5 そのもののアルゴリズムの正当性を示すことは、興味深い研究テーマである。

提案したアルゴリズムのさらなる拡張の方向性としては、優先度ベースのスピンロックアルゴリズム[8]との組合せが考えられる。

今回の実験では、プロセッサの数を 1～4 個まで変化させて行なったが、さらに多くのプロセッサ(10個程度)で実験することを計画している。また、提案したアルゴリズムを ITRON-MP 仕様のカーネルに組み込み、より現実的な環境での評価も必要と考えている。

参考文献

[1] H. Takada and K. Sakamura, "ITRON-MP: An adaptive real-time kernel specification for shared-memory multiprocessor systems," *IEEE Micro*, vol. 11, pp. 24–27,78–85, Aug. 1991.

[2] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, pp. 21–65, Feb. 1991.

[3] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, pp. 6–16, Jan. 1990.

[4] G. Graunke and S. Thakkar, "Synchronization algorithms for shared-memory multiprocessors," *IEEE Computer*, vol. 23, pp. 60–69, June 1990.

[5] K. Sato, H. Tsubota, O. Yamamoto, and K. Saitoh, "An experimental implementation of unified real-time operating system," in *Proc. of the Eighth TRON Project Symposium*, pp. 57–68, IEEE CS Press, 1991.

[6] N. Lynch and N. Shavit, "Timing-based mutual exclusion," in *Proc. Real-Time Systems Symposium*, pp. 2–11, Dec. 1992.

[7] K. Sakamura, *Specification of the Chip Based on the TRON Architecture*. TRON Association, Tokyo, 1989.

[8] E. P. Markatos, "Multiprocessor synchronization primitives with priorities," in *Proc. of the IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.