優先度継承スピンロックアルゴリズムとその評価

王 才 棟[†,☆] 高田広章[†] 坂村 健[††]

共有メモリマルチプロセッサ上のリアルタイムシステムでプロセッサ間排他制御を実現する場合に，優先度逆転をなるべく短くするために，優先度順のスピンロックが使われる．ところが，複数のロックを順に獲得する場合に優先度順スピンロックを単純に用いると，プロセッサ間で上限のない優先度逆転の問題が発生する．本論文では，スピンロックにおいても上限のない優先度逆転の問題が起こることを指摘し，その問題を解決するために，優先度順スピンロックに優先度継承の仕組みを導入した優先度継承スピンロックを提案する．2種類の優先度順スピンロックアルゴリズムをベースに，2種類の優先度継承スピンロックアルゴリズムを提示し，それらの有効性を実機を用いた性能評価によって検証する．

Priority Inheritance Spin Lock Algorithms and their Evaluation

CAI-DONG WANG,[†,☆] HIROAKI TAKADA[†] and KEN SAKAMURA[††]

In realizing a real-time system on a shared-memory multiprocessor, priority-ordered spin locks are used to reduce priority inversions. However, simple use of priority-ordered spin locks can cause uncontrolled priority inversions among processors when they are used for nested spin locks. This paper points out the problem of uncontrolled priority inversions in the context of spin locks and proposes priority inheritance spin locks, priority-ordered spin locks that are enhanced with a priority inheritance scheme, to solve the problem. Two priority inheritance spin lock algorithms are presented based on two priority-ordered spin lock algorithms, and their effectiveness is demonstrated through performance measurements.

1. はじめに

処理結果の正しさが，出力される結果値の正しさに加えて結果を出す時刻にも依存するようなシステムを，リアルタイムシステムと呼ぶ．リアルタイムシステムにおいては，各処理に対して時間制約が与えられ，それを満たすように処理を実行しなければならない．多くの場合，処理に対する時間制約は，スケジューリングアルゴリズムにより静的に，ないしはタスクスケジューラにより動的に，優先度に翻訳される．ランタイムシステムは，その優先度に従って共有資源を処理に割り当てる．

共有メモリマルチプロセッサシステムにおいて，共有資源に対する排他的なアクセスを実現するための基本的なプリミティブの1つに，スピンロックがある．スピンロックは，共有資源をプロセッサに割り当てる機構であるため，その割当て順序に優先度を反映することが必要になる場合がある．すなわち，優先度の高い処理を実行するプロセッサが，優先的にロックを獲得できることが必要となる．本論文では，優先度の順にロックを獲得できるスピンロックアルゴリズムについて議論する．

スピンロックアルゴリズムについては，各種の前提のもとで多くの研究がなされてきた．本論文の議論は，共有メモリの単一ワード（ないしは，連続した数ワード）に対する不可分なリードモディファイライト操作（test&set，fetch&store (swap)，fetch&add，compare&swap など）をハードウェアがサポートしていることを前提とする．同じ前提のもとで数多くのスピンロックアルゴリズムが提案されており[1)~3)]，広く利用されている．また，各プロセッサが優先度の順にロックを獲得することができる優先度順スピンロックについても，これまでに3種類のアルゴリズムが提案されている[4)~6)]．

マルチプロセッサシステムにおいて処理の並列度を上げるためには，排他的にアクセスする必要のある共

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, School of Science, University of Tokyo
†† 東京大学総合研究博物館
The University Museum, University of Tokyo
☆ 現在，キヤノテック株式会社
Presently with CANOTEC CO., INC.

有資源をいくつかのロック単位に分割し，ロックの競合頻度を下げる方法が使われる．このとき，異なるロック単位に属する共有資源を同時にアクセスする場合には，それらのロックをすべて獲得する必要がある．ところが，複数のロックを順に獲得する場合に優先度順スピンロックを単純に適用すると，上限のない優先度逆転の問題が起こる．タスクスケジューリングにおける優先度逆転の問題は広く知られているが，本論文では，スピンロックにおいても類似の問題が起こることを述べる．

この問題を解決するために，本論文では，優先度継承の考え方をスピンロックに導入する．優先度継承は，タスクスケジューリングにおける優先度逆転の問題を解決する有力なアプローチであるが[7]，スピンロックにおける優先度逆転の問題も解決することができることを述べる．また，2 種類の優先度継承スピンロックのアルゴリズムを提案し，それらの性能評価を行う．

以下では，2 章においてスピンロックにおいても上限のない優先度逆転の問題が起こることを指摘した後，優先度継承がこの問題を解決できることを述べる．続く 2 つの章では，優先度継承スピンロックを実現する 2 種類のアルゴリズムについて述べる．最初に 3 章で Markatos のアルゴリズム[4]をベースにしたアルゴリズム[8]を提示し，次に 4 章では PR-lock[6]をベースにしたアルゴリズムを提案する．最後に 5 章において，これらのアルゴリズムの有効性を検証するために行った性能評価について述べる．

2. 優先度逆転と優先度継承

2.1 上限のない優先度逆転の問題

優先度逆転とは，優先度の高い処理が何らかの理由で優先度の低い処理に待たされる現象をいう．プロセッサ間の排他制御においては，優先度の低いプロセッサがロックを獲得していることによって，優先度の高いプロセッサが待たされる場合が優先度逆転にあたる．優先度逆転が望ましくない現象であることはいうまでもないが，獲得しているロックの横取りを許さない限り優先度逆転が起こることは避けられず，その発生をいかに短くおさえるかが課題となる．優先度逆転が継続する最大時間が決められない状況は上限のない優先度逆転と呼ばれ，リアルタイムシステムが時間制約を満たすうえで大きな障害となる．

異なるロック単位に属する共有資源を同時にアクセスするために，優先度順スピンロックアルゴリズムを用いて複数のロックを順に獲得しようとした場合，上限のない優先度逆転の問題が発生する．上限のない優先度逆転が起こる典型的な例は次のとおりである．

```
acquire_lock(L2);
// クリティカルセクション
release_lock(L2);
```

ルーチン (a)

```
acquire_lock(L1);
acquire_lock(L2);
// クリティカルセクション
release_lock(L2);
release_lock(L1);
```

ルーチン (b)

図 1 ネストしたスピンロックの例
Fig. 1 An example of nested spin locks.

4 つのプロセッサ P_1，P_2，P_3，P_4（P_1 が最高優先度，P_4 が最低優先度とする）が，図 **1** の 2 種類のルーチンをランダムに実行し，2 つのロック L_1，L_2 の獲得・解放を繰り返す場合を考える．P_1 がルーチン (b) の中で L_1 を獲得しようとしたときに，L_1 が P_4 に獲得されており，さらに P_4 が L_2 の獲得を待っている状況を考える．このとき，P_2 と P_3 は P_4 よりも優先度が高いため，P_4 より先に L_2 を獲得することができる．各プロセッサがロックを保持している時間に上限がある場合でも，P_2 と P_3 がかわるがわる L_2 を獲得する可能性はあり，P_4 が L_2 を獲得できるまでの時間には上限がない．この間，P_1 は L_1 で待たされているが，P_1 は P_2 および P_3 よりも優先度が高いため，この状態は上限のない優先度逆転である．

2.2 優先度継承スピンロック

上限のない優先度逆転の問題を解決するための有力なアプローチとして，優先度継承の導入がある．優先度継承の基本的な考え方は，優先度の低い処理が優先度の高い処理を待たせている場合には，前者の優先度を後者の優先度まで引き上げるというものである．スピンロックの場合に言い換えると，各プロセッサは，それが獲得しているロックを待っているプロセッサの中で，最高優先度のものの優先度を継承する．ここで，優先度継承は遷移的に行う必要がある．たとえば，3 つのプロセッサ P_1，P_2，P_3（P_1 が最高優先度，P_3 が最低優先度とする）があるとき，P_3 が P_2 を待たせており，P_2 が P_1 を待たせているならば，P_3 は P_1 の優先度を継承する．

上限のない優先度逆転の問題が優先度継承によって解決することを，前節の例で説明する．前節の例では，P_4 が L_1 を保持しており，P_1 が L_1 を待っていた．このとき，P_1 の優先度は P_4 よりも高いため，優先度継承のルールにより P_4 の優先度は P_1 の優先度まで引き上げられる．その結果，P_4 の優先度が P_2，P_3 の優先度より高くなるため，P_2，P_3 よりも先に L_2 を獲得できる．そのため，最も優先度の高い P_1 が，P_2 と P_3 がかわるがわるロックを獲得するのを待た

されることがなくなり，優先度逆転の最大時間をおさえることが可能になる．

優先度継承のルールをそのまま適用すると，プロセッサがロックを解放した場合には，そのプロセッサの優先度を設定しなおす必要がある．具体的には，ロック解放後の優先度は，そのプロセッサの元来の優先度と，他に保持しているロックを待っているプロセッサの中での最高優先度に設定する．保持しているロックをすべて解放した後は，元来の優先度に戻ることになる．ただし，次の 2 条件がともに満たされる場合，最後のロックを解放した場合を除いて，ロック解放時の優先度の設定を省略することができる．

(1) ロックの獲得・解放に 2-phase 性がある．
言い換えると，いったんロックを解放し始めたプロセッサは，獲得しているすべてのロックを解放するまで，他のロックを獲得することがない場合．

(2) 継承した優先度はスピンロックのためにだけに使われる．
より厳密には，あるプロセッサがあるロックを保持していることにより継承した優先度は，そのプロセッサが別のロックを獲得するための優先度としてのみ使い，タスクスケジューリングなどには使われない場合．

本論文では，簡単のために，この 2 条件が成り立っていることを仮定する．これらの仮定を取り除けるようにアルゴリズムを拡張することは，それによるオーバヘッドを覚悟すれば，困難なことではない．

以上より，優先度継承スピンロックに求められる振舞いを整理すると次のようになる．

(1) プロセッサは，優先度の順にロックを獲得することができる☆．

(2) プロセッサ P_1 がロック待ちを始めるときに，P_1 の優先度がそのロックを保持しているプロセッサ P_2 の優先度よりも高い場合には，P_2 の優先度を P_1 の優先度まで引き上げる．

(3) プロセッサ P_1 の優先度がロック待ちの間に引き上げられた場合，P_1 の新しい優先度がそのロックを保持しているプロセッサ P_2 の優先度よりも高い場合には，P_2 の優先度を P_1 の優先度まで引き上げる（遷移的な優先度継承処理）．

3. 優先度継承スピンロックアルゴリズム (1)

この章では，Markatos によるキューを用いた優先度順スピンロックアルゴリズム[4]（以下，Markatos lock と呼ぶ）をベースに優先度継承を導入し，前章で述べた仕様を満たすように拡張したアルゴリズムについて述べる．このアルゴリズムは，我々が文献 8) で提案したものである．

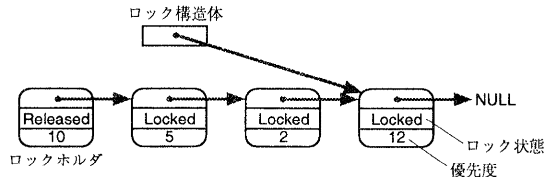

図 2 Markatos lock のデータ構造
Fig. 2 Data structures of the Markatos lock.

3.1 Markatos lock

Markatos lock は，代表的な FIFO 順のキューイングスピンロックアルゴリズム☆☆である MCS lock[2] をベースに，ロックが優先度順に渡されるよう改造したものである．ロック待ちキューは，MCS lock と同様に FIFO 順に構成される．ロックを保持するプロセッサ（以下では，ロックを保持するプロセッサをロックホルダと呼ぶ）は待ちキューの先頭にあり，新たにロックを待ち始めるプロセッサは待ちキューの末尾に入る．ロック構造体は，待ちキュー中の最後のプロセッサを指している（**図 2**）．ロックを解放するときには，ロックホルダはロック待ちキュー中で最も優先度の高いプロセッサを探し，そのプロセッサを待ちキューの先頭に移動した後，そのプロセッサにロックを渡す．

各プロセッサがローカルメモリ☆☆☆を持っている場合，Markatos lock ではローカルメモリ上の変数に対してスピンを行うようにできるため，コヒーレントキャッシュのないマルチプロセッサシステム上で用いても，共有バス（ないしは，インターコネクションネットワーク）にはわずかなトラフィックしか発生させないという特長を持つ．

なお，文献 4) に掲載されているアルゴリズムでは，ロック待ちキューを双方向リンクキューで実現しているが，実際には単方向リンクでも同じ機能を実現可能であることが分かったため，本論文では単方向リンクキューを用いたアルゴリズムをベースとする．

3.2 優先度継承の導入

Markatos lock に優先度継承を導入する場合に最も

☆ 優先度順のより厳密な定義は，文献 4) にある．

☆☆ キューイングスピンロックアルゴリズムとは，ロックを待っているプロセッサをキューを使って管理するスピンロックアルゴリズムのことをいう．

☆☆☆ ここでいうローカルメモリとは，その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を経由してアクセスできるメモリを指す．

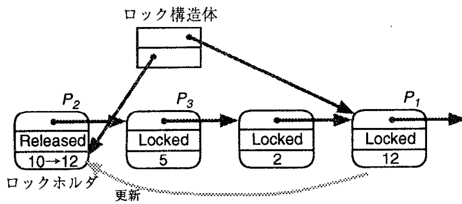

図 3 優先度継承導入のアプローチ
Fig. 3 An approach to incorporating priority inheritance.

問題となるのは，ロックホルダに優先度を継承する必要があることを知らせる方法である．最も効率が良いと考えられるアプローチとして，ロック構造体の中にロックホルダの優先度を管理する変数へのポインタを持たせ，そのポインタを経由してロックホルダの優先度を直接変更する方法が考えられる（**図 3**）．ところがこのアプローチでは，以下の問題により優先度継承を正しく実現することができない．

プロセッサ P_2 がロック L を解放しようとするのと同じタイミングで，より高い優先度を持ったプロセッサ P_1 が L を待ち始める場合を考える．ここで，次のような順序で処理が進む場合に問題が起こる．(1) P_2 は待ちキュー中から最高優先度のプロセッサ P_3 を見つけ，P_3 にロックを渡すことを決定する．(2) P_1 は待ちキューに入り，ロック構造体から P_2 の優先度変数を指すポインタを読み込む．(3) P_2 は P_3 にロックを渡し，ロック構造体中の優先度変数を指すポインタを更新する．(4) P_1 は P_2 に優先度を継承させる．この結果，すでにロックを解放している P_2 が優先度を継承し，優先度を継承すべきプロセッサ P_3 に継承されないという問題が起こる．このような場合にも P_3 が正しく優先度を継承できるようアルゴリズムを工夫することは可能であるが，P_2 が誤って優先度を継承してしまう問題を完全に解決することは難しい．

そこでもう1つのアプローチとして，優先度継承を行う必要があるかを，ロックホルダが定期的にチェックするアプローチを考える．具体的には，ロック構造体の中に，そのロックを待っているプロセッサの中での最高優先度を管理するフィールドを設ける．さらに，各プロセッサのローカルメモリ上に通知フラグと呼ばれる変数を用意し，ロックホルダの通知フラグを指すポインタをロック構造体中に管理する．ロックを待ち始めた（ないしはロック待ち中に他のプロセッサから優先度を継承した）プロセッサは，優先度がロックホルダよりも高い場合，ロック構造体中の最高優先度値を更新した後，ロック構造体から読み出したポインタを経由して通知フラグをセットし，ロックホルダに優

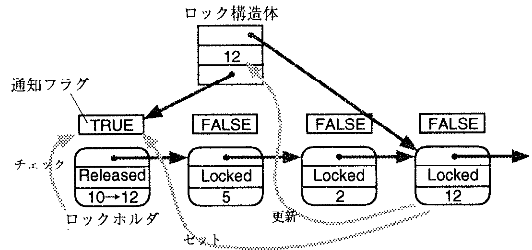

図 4 Markatos/PI における優先度継承処理
Fig. 4 Priority inheritance with Markatos/PI.

先度継承を行う必要がある（可能性がある）ことを知らせる．ロックホルダは通知フラグを定期的にチェックし，セットされていた場合には，ロック構造体中に管理されている最高優先度値を自分の優先度と比較し，前者の方が高い場合に優先度継承処理を行う（**図 4**）．通知フラグのチェックは，継承した優先度はスピンロック以外には使われないという条件から，別のロックを待っている間にのみ行えばよい．

この方法では，ロックホルダが自ら優先度を継承すべきかどうかのチェックを行うため，通知フラグが誤ってセットされた場合でも☆，誤って優先度を継承する問題は起こらない．逆に優先度の継承洩れを防ぐためには，ロックを受け取ったプロセッサが，通知フラグにかかわらず優先度継承を受けるべきかどうかを（一度だけ）チェックすればよい．

このアルゴリズムは，コヒーレントキャッシュを持たないマルチプロセッサシステムでも効率良く動作することができる．一方コヒーレントキャッシュを持つ場合には，ロックホルダがロック構造体中の最高優先度値と自分の優先度との比較を定期的に行い，通知フラグを省略する方法も可能である．

Markatos lock に優先度継承を導入したアルゴリズム（これを以下では Markatos/PI と呼ぶ）の詳細は，付録 A.1 で紹介する．

4. 優先度継承スピンロックアルゴリズム（2）

この章では，PR-lock [6] をベースに，優先度継承を導入したアルゴリズムを提案する．

4.1 PR-lock

Markatos lock がロック待ちキューを FIFO 順で作り，ロック解放時に最高優先度のプロセッサを探すのに対して，PR-lock では，ロック待ちキューを優先度順で作り，ロック解放時には待ちキューの先頭のプロ

☆ 先に述べたのと同じ理由で，通知フラグが誤ってセットされるのを防ぐことは難しい．

セッサにロックを渡す．すなわち，ロック構造体は待ちキューの先頭にあるロックホルダを指しており，新たにロックを待ち始めるプロセッサは待ちキュー中を検索し，自分より低い優先度のプロセッサを見つけたら，その手前に入る．そのため，優先度順に並べる処理が並列に実行できることになり，Markatos lock に比べて高いスループットが期待できる☆．PR-lock の場合も，Markatos lock と同様，各プロセッサがローカルメモリ上の変数に対してスピンを行うようにできる．

優先度順の待ちキューを実現するうえで起こる問題として，ロックを使わない同期において一般的に起こるA-B-A 問題[9]とポインタが指しているノードが他の目的に再利用されてしまう問題（迷子のポインタの問題）を，どのように回避するかという課題がある．これらの問題の回避策は何通りか考えられるが，本論文では，各プロセッサの実行速度比に上限があることを仮定して実行効率を上げる回避策を採用している☆☆．そのため，本論文で PR-lock と呼ぶものは，厳密には文献 6) に提示されている PR-lock とは一致していない．またその他にも，文献 6) のアルゴリズムを細部で修正しているが，本論文ではこれらの問題については省略する．

4.2 優先度継承の導入

PR-lock では，待ちキューが優先度順に作られているために，優先度継承によってロックを待っているプロセッサの優先度が変更された場合には，待ちキュー中での位置を移動させる必要がある．そのため，この部分で Markatos lock をベースにする場合と比べてオーバヘッドが大きくなることが予想される．優先度が変更されたプロセッサが待ちキュー中で位置を移動させる具体的な手順としては，挿入されているのとは別のノードを用意し，それを新しい優先度に対応する位置に挿入した後，元のノードをキューから外すという順序で行う．これとは逆に，ノードをキューから外した後に挿入しなおすという手順にすると，キューにまったくつながっていない状態が過渡的に生じるのが不都合であると考えた．

一方，Markatos lock と比べて優先度継承を導入する上で有利な点として，PR-lock ではロック構造体

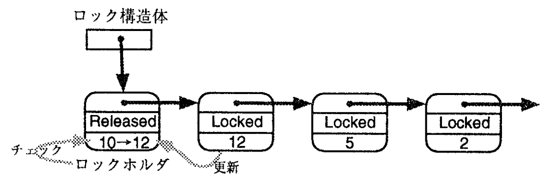

図 5 PR-lock/PI における優先度継承処理
Fig. 5 Priority inheritance with PR-lock/PI.

がロックホルダを指すポインタを持っているために，ロック構造体に新たなフィールドを追加する必要がないことがあげられる．また，アルゴリズムの性質上，Markatos lock を拡張する際に問題となった継承漏れは起こらない．一方，誤った優先度継承の問題は，A-B-A 問題を回避するのに用いた方法によって同様に回避される☆☆☆．そのため，優先度継承を通知する場合に，ロックホルダのノード中の優先度フィールドに直接書き込む方法をとることができ，新たな変数を用意してそれを管理するオーバヘッドを生じないという利点がある（図 5）．以上の理由から，優先度継承を通知する機構はきわめてシンプルに実現できる．

PR-lock に優先度継承を導入したアルゴリズム（これを以下では PR-lock/PI と呼ぶ）の詳細は，付録 A.2 で紹介する．

5. 実機による性能評価

この章では，実機を用いた性能測定により，提案したアルゴリズムの評価を行う．提案した 2 種類のアルゴリズムを，優先度継承を導入しないオリジナルのアルゴリズムと比較するとともに，2 種類のアルゴリズム相互の比較も行う．

5.1 評価環境の概要

性能評価には，プロセッサに TRON 仕様のプロセッサである Gmicro/200，共有バスに VME バスを用いた共有メモリマルチプロセッサシステムを用いた．プロセッサボード上には，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．プロセッサボード上に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各プロセッサごとのデータ領域は各プロセッサのローカルメモリ上に，システム全体で共有されるデータ

☆ ただし，著者らが知る限り，Markatos lock と PR-lock を実機を用いて性能比較した例はない．

☆☆ 具体的には，与えられた実行速度比の上限に対して，十分な数のデータ構造を用意することで，正しく動作するアルゴリズムを構築することができる．リアルタイムシステムにおいては，実行速度比に上限があるという仮定は満たされているのが通常である．

☆☆☆ Markatos/PI の場合と同様に，ロックを待っているプロセッサ中の最高優先度をロックホルダに再チェックさせることで，誤った優先度継承の問題を防ぐ方法も可能である．この場合にも，PR-lock では，最高優先度のプロセッサが待ちキュー中でロックホルダの直後にあるため，ロック構造体に新たなフィールドを追加する必要はない．

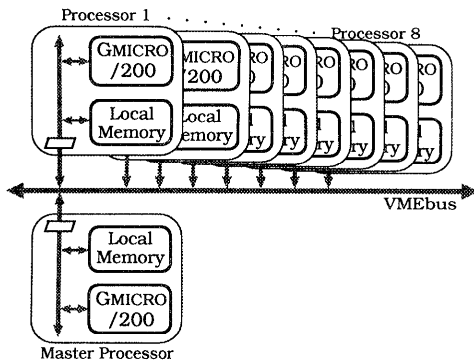

図 6　評価環境の概要図
Fig. 6　Overview of the evaluation environments.

領域についてはスピンロックに加わらないマスタプロセッサのローカルメモリ上に置いた（図 **6**）.

Gmicro/200 は，リードモディファイライト操作として，compare&swap を持っている．一方，Markatos lock で使う fetch&store は持っていないため，同等の機能を compare&swap を用いたループによって実現した．VME バスの調停方式としては，ラウンドロビン方式を用いた．今回の実験では，プロセッサの数を 1 個から 8 個まで変化させて性能評価を行ったが，VME バスでは最大 4 つのバスマスタまでしかラウンドロビンで調停できないため，8 個のプロセッサを 4 つの組に分け，それぞれの組の中では番号の大きい方がつねに優先となるようにした.

5.2 評価方法

性能評価は，各プロセッサに，繰り返し図 1 に示した 2 つのルーチンのうちの 1 つをランダムに選択して実行させ，その実行時間を計測することで行う．各プロセッサの優先度は，それぞれ異なる値（具体的には，プロセッサの ID 番号）に固定した．計測は各ルーチンの実行ごとに行い，実行時間の分布を求めた．各ルーチンのクリティカルセクション内では，数回の共有バスアクセスと，空ループを使った時間待ちを行う．ルーチン (b) の場合には，2 つのロックを獲得する中間にもこのような処理を入れた．スピンロックを行わない場合のクリティカルセクションの実行時間は，実行時間計測のためのオーバヘッドを含めて約 30 μs である．各プロセッサは，ルーチンを 1 つ実行した後，空ループを使ってランダムな時間待った後，次のルーチンの実行を開始する.

PR-lock および PR-lock/PI において，ロック解放中に，共有バスアクセスが繰り返される状態が一時的に起こる．この状態は，ロックを解放するプロセッサの実行が数命令進むと解消するきわめて一時的なものであるが，用いた評価環境においてはバスアクセスが不優先のプロセッサがあるため，そのようなプロセッサがロックを解放する場合にはバスの飽和を引き起こす場合がある．そこで，バスの飽和を防ぐために，付録 A.2 の図 14 および図 15 に (∗) で示す箇所に若干の待ちを入れた.

リアルタイムシステムの場合，最悪実行時間がアルゴリズムの性能を評価するうえでの重要な指標となる．ところが，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の非決定性のために，実験により最悪値を求めることは難しい．さらに，用いた評価環境では，共有バスアクセスにかかる時間に上限がないなどの理由で，最悪実行時間は原理上も押えられない．そこで以下では，最悪実行時間に代えて，その時間内に実行が終わる確率がある定数値 p 以上になるような時間を用いて，性能を評価する．この時間を p–信頼実行時間と呼ぶ．言い換えると，p–信頼実行時間をデッドラインとした場合，デッドラインを守る確率が p となる.

5.3 評価結果

図 **7**，図 **8** に，プロセッサ数を 1 個から 8 個まで変化させた場合に，最高優先度のプロセッサがルーチン (a)，(b) を実行するのにかかる 99.99%–信頼実行時間を示す．図 8 において，プロセッサ数が多い場合には，優先度継承を行わないアルゴリズムの性能が著しく低下している.

これが上限のない優先度逆転によるものであることを確かめるために，次の追加評価を行った．まず，評価用のプログラムに，各ロックを各プロセッサがどの順序で取得したかを記録するルーチンを追加し，図 8 と同様の結果が得られることを確認した．次に，優先度継承を行わないアルゴリズムに対して，ルーチン (b) の実行時間が顕著に長くかかったタイミングで評価プログラムを停止させ，取得した記録からその直前に起きた現象を調べた．その結果，顕著に長い実行時間がかかる直前には必ず，2.1 節で例示したような，高い優先度のプロセッサが低い優先度のプロセッサに待たされる間に，中間の優先度を持ったプロセッサが繰り返しロックを取得するという現象が起きていることが確認できた.

このことから，プロセッサ数が多い場合の性能低下は，上限のない優先度逆転によるものであるということができる．それに対してプロセッサ数が 4 以下の場合は，上限のない優先度逆転が顕著に起こらないために，優先度継承を行わないアルゴリズムの方が良い性能を示す．このときの性能差は，優先度継承の処理に

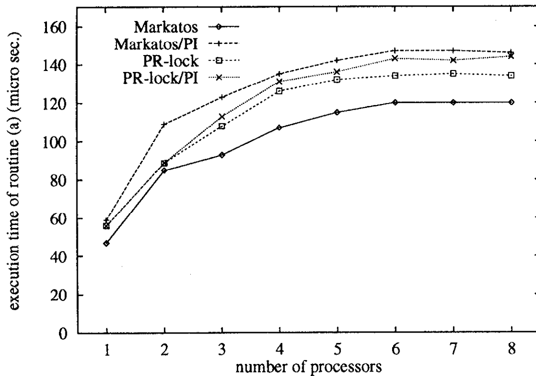

図7 ルーチン (a) の 99.99%–信頼実行時間
Fig. 7 99.99%–realiable execution times of routine (a).

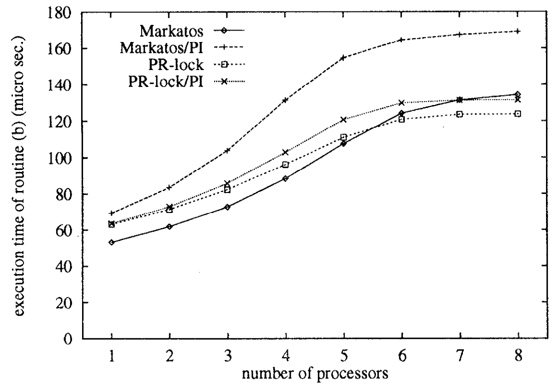

図9 ルーチン (b) の平均実行時間
Fig. 9 Average execution times of routine (b).

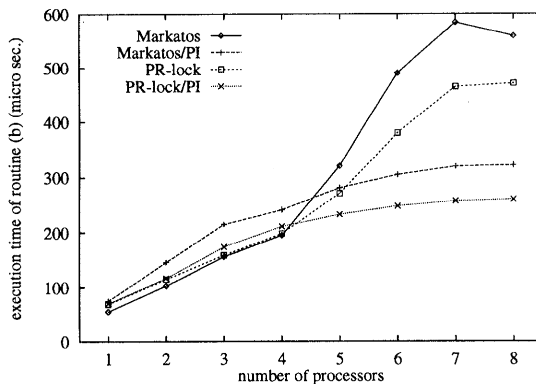

図8 ルーチン (b) の 99.99%–信頼実行時間
Fig. 8 99.99%–realiable execution times of routine (b).

かかるオーバヘッドと考えられる．

図9には，プロセッサ数を1個から8個まで変化させた場合に，最高優先度のプロセッサがルーチン (b) の実行にかかる平均時間を示す．これらの図から，優先度継承の効果は平均実行時間には現れないことが分かる．むしろ，優先度継承処理のオーバヘッドにより，平均性能は悪くなる．このオーバヘッドは，特にMarkatos/PIにおいて大きくなっている．

以上の性能評価の結果から，優先度順スピンロックでは上限のない優先度逆転の問題が起こること，優先度継承スピンロックがその問題を解消できることが検証できた．ただし，優先度継承スピンロックを用いると，優先度継承の処理にかかるオーバヘッドのために，平均実行時間は長くなる．そのため，平均性能が主たる興味であるアプリケーションには，優先度継承スピンロックを用いるべきではない．

またMarkatos/PIとPR-lock/PIを比較すると，いずれのケースでもPR-lock/PIの方が良い性能を示した．これは，Markatos/PIにおいて，ロック構造体を保守するオーバヘッドが大きいためと考えられる．ただし，4.1節で述べたように，本論文で用いたPR-lockは各プロセッサの実行速度比に上限があることを仮定して実行効率を上げているため，公平な比較とはいえない面が残る．

6. 結論と今後の課題

マルチプロセッサシステムにおいてプロセッサ間の排他制御をスピンロックで実現する場合に，優先度順スピンロックを単純に用いて複数のロックを順に獲得すると，上限のない優先度逆転の問題が生じる．上限のない優先度逆転の問題は，リアルタイムシステムが時間制約を満たすうえで大きな障害となるものである．我々は，この問題を解決するために優先度継承の考え方が有効であることを示し，それを適用した優先度継承スピンロックを提案した．また，2種類の優先度順スピンロックアルゴリズムをベースに，2種類の優先度継承スピンロックアルゴリズムを提案し，それらの有効性を実機を用いた性能評価によって確かめた．

本論文では，優先度の順にロックを獲得できるスピンロックについて議論したが，リアルタイムシステムに用いるスピンロックに求められる他の要求として，各プロセッサがロックを獲得するまでの時間に（なるべく短い）上限があるという性質がある．いうまでもなく，この2種類の性質は両立しないもので，どちらの性質が要求されるかはアプリケーションからの要求やシステムの設計手法に依存する．実際には，ロック獲得時間に上限があるスピンロックを優れたスケーラビリティを持って実現する場合には，優先度順スピンロックを用いる必要が生じる[10]．優先度継承スピンロックも，同時に獲得すべきロックの数が3個以上になる場合に必要となる．

本研究は，機能分散マルチプロセッサのためのリアルタイムカーネルに適用することを動機として行ったものであり，本論文で提案したアルゴリズムをリアルタイムカーネルへ組み込んでの評価も進行中である．

謝辞 PR-lock に優先度継承を導入する手法に関して議論に応じてくれた Univ. of Florida の Prof. Theodore Johnson に感謝する．

参考文献

1) 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).

2) 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).

3) Rhee, I.: Optimizing a FIFO, Scalable Spin Lock Using Consistent Memory, *Proc. Real-Time Systems Symposium*, pp.106–114 (1996).

4) Markatos, E.P.: Multiprocessor Synchronization Primitives with Priorities, *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software* (1991).

5) Craig, T.S.: Queuing Spin Lock Algorithms to Support Timing Predictability, *Proc. Real-Time Systems Symposium*, pp.148–157 (1993).

6) Johnson, T. and Harathi, K.: A Prioritized Multiprocessor Spin Lock, Technical Report TR-93-005, Department of Computer Science, University of Florida (1993).

7) Sha, L., Rajkumar, R. and Lehoczky, J.P.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization, *IEEE Trans. Computers*, Vol.39, No.9, pp.1175–1185 (1990).

8) Wang, C.-D., Takada, H. and Sakamura, K.: Priority Inheritance Spin Locks for Multiprocessor Real-Time Systems, *Proc. Int'l Symposium on Parallel Architectures, Algorithms, and Networks*, pp.70–76, IEEE CS Press (1996).

9) Prakash, S., Lee, Y.-H. and Johnson, T.: A Non-Blocking Algorithm for Shared Queues Using Compare-and-Swap, *Proc. 1991 Int'l Conference on Parallel Processing*, pp.II-68–II-75 (1991).

10) Takada, H. and Sakamura, K.: Real-Time Scalability of Nested Spin Locks, *Proc. 2nd Real-Time Computing Systems and Applications*, pp.160–167 (1995).

付　録

A.1 Markatos/PI アルゴリズムの詳細

この付録では簡単のために，1つのプロセッサが同時に獲得するロックの数が2個以下の場合の疑似コードのみを示す．任意個のロックを同時に獲得できるように拡張する場合には，プロセッサが保持しているロックをリストなどを用いて管理すればよい．

Markatos/PI で使うデータ構造およびロックを使うルーチンの例（図1のルーチン (b) の疑似コード）を**図10**に，またアルゴリズム本体の疑似コードを**図12**，**図13**に示す．これらの図中で，CAS は compare_and_swap の略記で，第1引数（ポインタ）が指すメモリ番地の内容を読み，それが第2引数と一致している場合に，その番地に第3引数を書き込む操作を不可分に行うものである．書き込みを行った場合に TRUE を返す．fetch_and_store は，第1引数（ポインタ）が指すメモリ番地の内容を読んで返り値とし，それと不可分にその番地に第2引数を書き込む．また and，or はそれぞれ conditional and，conditional or を表す．

これらの図中には acquire_first_lock 関数が含まれていないが，これは，acquire_second_lock 関数から，第3引数 lock1 と，ロック待ちの間に通知フラグをチェックし必要なら優先度継承処理を行う部分を省いたもので，容易に再構成可能と考え紙面の関係で省略した．

release_lock 関数中で，最初にロック構造体の最高優先度フィールドを最低優先度値（MIN_PRIO）に設定し，ロックを渡す相手プロセッサが決定した時点で，必要ならそのプロセッサの優先度に設定し直している．これは，単にロックを渡す相手プロセッサの優先度に設定するだけでは，ロックを渡す処理の間に，より高い優先度のプロセッサが待ちキューにつながった場合や，ロックを待っているプロセッサがより高い優先度を継承した場合に，その値が最高優先度フィールドに反映されなくなるためである．なお，ロックを保持しているプロセッサがない場合には，通知フラグへのポインタは NULL，最高優先度フォールドは MIN_PRIO に設定される．

A.2 PR-lock/PI アルゴリズムの詳細

PR-lock/PI で使うデータ構造およびロックを使うルーチンの例を**図11**に，またアルゴリズム本体の疑似コードを**図14**，**図15**に示す．Markatos/PI の場合と同様に，容易に再構成可能と考え acquire_first_lock 関数は省略した．

```
// ロック待ちキューのノード
type Node = record
  next: pointer to Node;    // 次のプロセッサ
  locked: (Released, Locked);    // ロックの状態
  prio: integer    // 優先度
end;

// ロック構造体
type Lock = record
  last: pointer to Node;    // ロック待ちキューの末尾
  maxprio: integer;    // ロック待ちプロセッサ中の最大優先度
  notifyp: pointer to boolean    // 通知フラグへのポインタ
end;
// last と notifyp は NULL に初期化する
// maxprio は MIN_PRIO に初期化する

type NodePtr = pointer to Node;
type LockPtr = pointer to Lock;

// システムで 1 つだけ用意する共有変数
shared var L1, L2: Lock;

// プロセッサごとに用意する共有変数
var I1, I2: Node;
var my_prio: integer;    // プロセッサの優先度
var my_notify: boolean;    // 通知フラグ

procedure routine_b();
begin
  // ここで my_prio を初期化する
  acquire_first_lock(&L1, &I1);
  acquire_second_lock(&L2, &I2, &L1);
  // クリティカルセクション
  release_lock(&L2, &I2);
  release_lock(&L1, &I1);
end;
```

図 10 Markatos/PI のデータ構造と使い方
Fig. 10 The data structures and a usage of Markatos/PI.

```
type Node = record
  next: pointer to Node;    // 次のプロセッサ
  locked: (Released, Locked);    // ロックの状態
  prio: integer    // 優先度
end;

type Lock = pointer to Node;

type NodePtr = pointer to Node;
type LockPtr = pointer to Lock;

// システムで 1 つだけ用意する共有変数
shared var L1, L2: Lock;

// プロセッサごとに用意する共有変数
var my_locknode[NUM_LOCKNODE]: Node;
var next_locknode: integer;    // 0 に初期化
var my_prio: integer;

procedure routine_b();
  var me1, me2: NodePtr;
begin
  // ここで my_prio を初期化する
  me1 = acquire_first_lock(&L1);
  // クリティカルセクション (前半)
  me2 = acquire_second_lock(me1, &L2);
  // クリティカルセクション (後半)
  release_lock(&L2, me2);
  release_lock(&L1, me1)
end;
```

図 11 PR-lock/PI のデータ構造と使い方
Fig. 11 The data structures and a usage of PR-lock/PI.

```
// lock の構造体中の最高優先度を newprio に引き上げる
// 実際に最高優先度を引き上げた場合に TRUE を返す
procedure raise_priority(lock: LockPtr,
                         newprio: integer): boolean;
  var prio: integer;
begin
retry:
  prio := lock→maxprio;
  if newprio > prio then
    if CAS(&(lock→maxprio), prio, newprio) then
      return TRUE
    end;
    goto retry
  end;
  return FALSE
end;

// 最高優先度を引き上げ，通知フラグをセットする
procedure raise_priority_notify(lock: LockPtr,
                                newprio: integer);
  var notifyp: pointer to boolean;
begin
  if raise_priority(lock, newprio) then
    notifyp := lock→notifyp;
    if notifyp ≠ NULL then
      *notifyp := TRUE    // 通知フラグをセット
    end
  end
end;

// entry を lock の待ちキューの先頭に移動する
// pred には entry の手前のノードを，
// oldtop には移動する前の先頭ノードを渡す
procedure move_to_top(lock: LockPtr, entry, pred,
                      oldtop: NodePtr);
  var succ: NodePtr;
begin
  succ := entry→next;
  if succ = NULL then
    // entry が末尾のノードの場合の処理
    pred→next := NULL;
    if CAS(&(lock→last), entry, pred) then
      entry→next := oldtop;
      return
    end;
    repeat succ := entry→next until succ ≠ NULL
  end;
  // entry の次にノード succ がある場合の処理
  pred→next := succ;
  enry→next := oldtop
end;
```

図 12 Markatos/PI のアルゴリズム本体（その 1）
Fig. 12 Pseudo-code of Markatos/PI (Part 1).

A-B-A 問題と迷子のポインタの問題を回避するために，Node 構造体を複数確保し，使い終わったものをすぐには再利用しないようにしている．my_locknode はそのための Node 構造体の配列で，各プロセッサの実行速度比の上限が大きい程，配列の要素数（NUM_LOCKNODE）を大きくする必要がある．next_locknode は最後に次に使うべき Node 構造体を記憶するための変数，get_next_locknode は次に使うべき Node 構造体を my_locknode から確保するための関数，return_locknode は直前に確保した Node 構造体を使わなかった場合に呼ぶべき関数である☆．PR-

☆ Node 構造体の無駄使いを気にしないなら必要ない．

```
// 2つめのロック lock を獲得する
// 最初に取ったロックを lock1 に渡す
procedure acquire_second_lock(lock: LockPtr,
                   me: NodePtr, lock1: LockPtr);
  var pred: NodePtr;
begin
  me→next := NULL;
  pred := fetch_and_store(&(lock→last), me);
  if pred ≠ NULL then
    me→locked := Locked;
    me→prio := my_prio;
    pred→next := me;
    // 優先度を継承させる処理
    raise_priority_notify(lock, my_prio);
    my_notify := TRUE;   // 最初に優先度継承をチェック
    repeat
      if my_notify then   // 通知フラグをチェック
        my_notify := FALSE;
        if lock1→maxprio > my_prio then
          my_prio := lock1→maxprio;
          me→prio := my_prio;
          // 優先度を継承させる処理（遷移的な継承）
          raise_priority_notify(lock, my_prio)
        end
      end
    until me→locked = Released
  else
    // ロックを保持しているプロセッサがなかった場合
    raise_priority(lock, my_prio)
  end;
  // 通知フラグへのポインタを自分の通知フラグに向ける
  lock→notifyp := &my_notify
end;

// ロックを解放する
procedure release_lock(lock: LockPtr, me: NodePtr);
  var top, entry, pred: NodePtr;
  var hentry, hpred: NodePtr;
  var hprio: integer;
begin
  lock→maxprio = MIN_PRIO;
  lock→notifyp := NULL;
  top := me→next;
  if top = NULL then
    // 待ちキュー中に他のプロセッサがなかった場合
    if CAS(&(lock→last), me, NULL) then
      return
    end;
    repeat top := me→next until top ≠ NULL
  end;
  // 待ちキュー中の最高優先度プロセッサを探す
  hentry := top;
  hprio := top→prio;
  hpred := NULL;
  pred := top;
  entry := pred→next;
  while entry ≠ NULL do
    if (entry→prio > hprio) then
      hentry := entry;
      hprio := entry→prio;
      hpred := pred
    end;
    pred := entry;
    entry := pred→next
  end;
  // この時点で hentry が最高優先度プロセッサ
  if hentry ≠ top then
    // hentry をキューの先頭に移動
    move_to_top(lock, hentry, hpred, top)
  end;
  raise_priority(lock, hprio);
  hentry→locked = Released
end;
```

図13 Markatos/PI のアルゴリズム本体（その2）
Fig. 13 Pseudo-code of Markatos/PI (Part 2).

```
// キューから外れたことを示すフラグをセットする
// (マークされる前のポインタを返す)
function mark(node: NodePtr): NodePtr;
  var succ : NodePtr;
begin
  repeat
    succ := node→next
  until CAS(&(node→next), succ, MARK(succ));
  return succ
end;

// 優先度を継承させる処理
procedure inherit(lock: LockPtr, me: NodePtr);
  var head: NodePtr;
  var prio: integer;
begin
  repeat
    head := *lock;
    if head = me then
      break
    end;
    prio := head→lockpri
  until prio ≥ my_prio
          or CAS(&(head→lockpri), prio, my_prio)
end;

// 2つめのロック lock を獲得する
// (最初に取ったロックのノードを me1 に渡す)
// (優先度継承をチェックすべきロックノードを返す)
function acquire_second_lock(me1: NodePtr,
                   lock: Lockptr): NodePtr;
  var me, me2, pred, succ, succ1, next: NodePtr;
  var prio: integer;
begin
  me := get_next_locknode();
  me→lockpri := my_prio;
again1:
  // me をロック待ちキューに挿入する
  pred := *lock;
  while pred = NULL do
    me→next := NULL;
    if CAS(lock, NULL, me) then
      return me
    end;
    pred := *lock
  end;
  // me1 から優先度を継承する処理をする
  prio := me1→lockpri;
  if prio > my_prio then
    my_prio := prio;
    me→lockpri := prio
  end;
  // me をロック待ちキューに挿入する (続き)
  me→locked := Locked;
  while (TRUE) do
    succ1 := pred→next;
    succ := UNMARK(succ1);
    if succ ≠ NULL and succ→lockpri ≥ my_prio then
      pred := succ
    elseif MARKED(succ1) then
(*)     goto again1
    else
      me→next := succ;
      if CAS(&(pred→next), succ, me) then
        break
      end
    end
  end;
  // ロックホルダに優先度を継承させる
  inherit(lock, me);
```

図14 PR-lock/PI のアルゴリズム本体（その1）
Fig. 14 Pseudo-code of PR-lock/PI (Part 1).

```
wait_loop:    // ロック待ちループ
  until me→locked = Released do
    // 優先度継承が必要かをチェック
    prio := me1→lockpri;
    if prio > my_prio then
      // 以下，me1 から優先度を継承する処理
      my_prio := prio;
      me→lockpri := prio;
      me2 := get_next_locknode();
      me2→lockpri := my_prio;
      me2→locked := Locked;
    again2:
      // me2 を待ちキューに挿入する
      pred := *lock;
      if pred = me then    // 移動の必要なし
        return_locknode(me2);
        inherit(lock, me);
        goto wait_loop
      end;
      while (TRUE) do
        succ1 := pred→next;
        succ := UNMARK(succ1);
        if succ = me then    // 移動の必要なし
          return_locknode(me2);
          inherit(lock, me);
          goto wait_loop
        end;
        if succ→lockpri ≥ my_prio then
          pred := succ
        elseif MARKED(succ1) then
(*)       goto again2
        else
          me2→next := succ;
          if CAS(&(pred→next), succ, me2) then
            break
          end
        end
      end;
      // ロックホルダに優先度を継承させる
      inherit(lock, me2);
    again3:
      // me を待ちキューから削除する
      pred := me2;
      while (TRUE) do
        succ1 := pred→next;
        succ := UNMARK(succ1);
        if succ ≠ me then
          pred := succ
        else
          next := mark(me);
          if CAS(&pred→next, me, next) then
            me := me2;
            goto wait_loop
          else
            me→next := next;
            goto again3
          end
        end
      end
    end
  end;
  return me
end;

// ロックを解放する
procedure release_lock(lock: LockPtr, me: NodePtr);
  var succ : NodePtr;
begin
  succ := mark(me);
  *lock := succ;
  if succ ≠ NULL then
    succ→locked := Released
  end
end;
```

図15 PR-lock/PIのアルゴリズム本体（その2）
Fig. 15 Pseudo-code of PR-lock/PI (Part 2).

lock/PIでは，優先度継承処理によって待ちキュー中での位置が移動する場合に，使っているNode構造体がacquire_second_lock関数内で変更になる．そのため，Node構造体はロックを獲得するルーチンの側で確保し，最終的にどのNode構造体を使ったかを，返り値として返す仕様としている．

PR-lockおよびPR-lock/PIでは，Node構造体のnextフィールドの余ったビット（通常は最下位ビット）を，そのノードがキューから外されたことを示すためのフラグとして利用しているが，MARK，UNMARK，MARKEDはそれぞれ，このビットをセットする，リセットする，セットされているかチェックする関数である．

（平成9年1月 6 日受付）
（平成9年9月10日採録）

王　才棟

1985年ハルピン科学技術大学自動制御学科卒業．1988年ハルピン工業大学大学院入学．1989年交換留学生として来日．1991年千葉工業大学大学院電子工学専攻修士課程修了．現在，キヤノテック（株）勤務．1994年から1997年まで，東京大学大学院理学系研究科情報科学専攻に在学．その間，ITRON仕様OS，マルチプロセッサシステムのためのリアルタイムOSの研究に従事．リアルタイムシステム，組み込みシステム，ネットワークシステムに興味を持つ．

高田　広章（正会員）

1986年東京大学理学部情報科学科卒業．1988年同大学院理学系研究科情報科学専攻修士課程修了．現在，同助手．マルチプロセッサシステムのためのリアルタイムOSとその実現アルゴリズム，リアルタイムスケジューリング理論などの研究に従事．リアルタイムシステム一般，組み込みシステム，コンピュータネットワークに興味を持つ．ITRONプロジェクトの活動に，中心的メンバとして参加．博士（理学）．IEEE，ACM，電子情報通信学会，日本ソフトウェア科学会各会員．

坂村 健（正会員）

東京大学総合研究博物館教授．1984年よりTRONプロジェクトを開始し，以来プロジェクトリーダとして，TRONアーキテクチャに基づいた新しいコンピュータ体系を作り上げるための活動を精力的に行う．生活環境のあらゆる場面にコンピュータが利用される時代に，コンピュータによって社会がどのように変わっていくかに興味を持つ．最近は，コンピュータ技術を駆使したデジタルミュージアムの構築を，東京大学総合研究博物館において手がける．工学博士．電子情報通信学会，ACM各会員．IEEEシニアメンバ．IEEE Micro編集委員．