組込みリアルタイムシステム
を対象とした
マルチコアプラットフォーム技術

一場　利幸

組込みリアルタイムシステムを対象とした
マルチコアプラットフォーム技術

要旨

組込みリアルタイムシステムを開発するには，まず，アプリケーションが動作するための基盤となるハードウェアおよびソフトウェア（プラットフォームと呼ぶ）を定めてから，アプリケーションの開発を行う．プラットフォームを構成するソフトウェア（ソフトウェアプラットフォーム）の代表例は，OS である．ソフトウェアプラットフォームを利用することで，アプリケーションソフトウェアの保守性や再利用性の向上を実現することができる．組込みリアルタイムシステムでは，特に，実行時間の予測可能性を持ったリアルタイム OS を用いるのが通常である．これは，組込みリアルタイムシステムでは，満たすべき時間要件（時間制約）に従って動作する性質（リアルタイム性）を要求されるためである．パーソナルコンピュータやスーパーコンピュータなどの汎用システムで用いられている汎用 OS では，リアルタイム性を満たすことができない．

現在，汎用システムのほとんどは，マルチコアシステムとなっている．ハードウェアリソースへの制約が厳しい組込みシステムにおいても，その重要性が増している．これは，現在の半導体技術においては，消費電力の増大を抑えつつシステム全体のスループットの向上を図るためには，クロック周波数を上げるよりも，コア数を増やしたほうが有利なためである．

マルチコアシステムの重要性が増すにつれて，マルチコアシステムをサポートするマルチコアプラットフォームの重要性も増している．ソフトウェアプラットフォームの役割の 1 つは，ハードウェアの隠蔽である．マルチコアシステムにおいては，対象となるハードウェアに関する知識を持った上でプログラミングしなければ，マルチコアシステムの性能を十分に発揮できない．ソフトウェアプラットフォームを利用することにより，アプリケーション開発者に求められるハードウェア知識を減らすことができる．マルチコアシステム上のソフトウェアは，複雑なものになりやすく，マルチコアプラットフォームの果たす役割は大きいといえる．

マルチコアプラットフォームを構築するにあたって解決すべき課題は多く，組込みリアルタイムシステムで用いるために重要な点として，次の 3 つが考えられる．1 点目はリアルタイム性である． 組込みリアルタイムシステムで用いるために必要な具体的な性質は，プラットフォームとしてアプリケーションに提供する API

の最悪実行時間が定まることや，割込み応答時間が短いこと，である．2 点目は実行オーバヘッドである． これは，実行時間の増分である 組込みリアルタイムシステムにおいても，実行オーバヘッドを抑えることが要求される場合が多い．本論文ではコア数を増やしたことによる増分に着目する．3 点目は信頼性である．プラットフォームは様々なアプリケーションで利用されることを想定するため，その動作による不具合が存在すると多数のシステムに影響を与えてしまう．そのため，プラットフォームが提供する動作による不具合が発生しないことが求められる．特に，組込みリアルタイムシステムでは，製品化された後でソフトウェアの更新により不具合を修正することが困難であったり，非常に多くの費用を要したりする場合があるため，不具合のないことが重視されることが多い．不具合が発生しやすいのは，割込み処理や排他制御に関連する部分であり，マルチコアプラットフォームにおいてはデッドロックが発生しないことが重要である．デッドロックが発生するのは非常にまれなケースで，再現が困難であることが多いため，発生しないことを保証することが課題となる．以上の 3 つの課題を解決する，組込みリアルタイムシステムに適したマルチコアプラットフォームが必要である

本研究の目的は，組込みシステムに適したマルチコアプラットフォームを構築するため，3 つの課題を解決する技術を提案することである．特に，3 つの課題すべてに影響を与える，排他制御に着目する．排他制御による影響については，次の通りである．排他制御によって他のコアに処理を待たされると，API の最悪実行時間が長くなる．他のコアの実行状態が関わるため，最悪実行時間がいくつであるかを容易に知ることができず，リアルタイム性を損なう恐れがある．他のコアに処理を待たされない場合においても，排他制御のための処理を実行する時間が長いと，実行オーバヘッドも長くなる．さらに，排他制御はマルチコアプラットフォームの果たす大きな役割であるので，排他制御が正しく動くこと，信頼性のあることが重要である．

本研究の具体的な内容は以下の 3 つである．

まず，コア間の共有リソースを排他する際に用いる排他制御アルゴリズムを提案する．マルチコアプロセッサを用いたリアルタイムシステム向けの排他制御アルゴリズムとして，数多くのキューイングスピンロックアルゴリズムが提案されている．これらの多くは，単一のロックを取得する状況を想定しているため，複数のロックを同時に取得する場合には，コア数に対するスケーラビリティとリアルタイム性を両立できないという問題がある．本研究では，スピンロックが満たすべき要件を示し，提案アルゴリズムが全ての要件を満たすことを示す．さらに，

提案アルゴリズムをハードウェアで実装することで高速に実行できる手法について述べる．

2つ目として，マルチコアプラットフォームを構成するソフトウェアのテスト効率化手法を提案する．マルチコアシステムにおいては，ソフトウェアプラットフォーム内に各コアの実行順序に依存してパスが定まる分岐（実行順序依存分岐と呼ぶ）が存在する．この分岐は，各コアが並列に実行することによって発生する不都合な状態に対応するために，ソフトウェアプラットフォームの開発者が意図的に作成したものである．分岐を網羅する方法として，実行順序依存分岐を実行する可能性のあるプログラムを繰り返し実行する手法が考えられる．しかし，特定のパスを実行する可能性が低い場合は，実行順序依存分岐を網羅するために非常に多くの繰り返し回数を要する．プログラムを繰り返し実行する手法は，テストの実施や不具合の分析に多くの手間を要するため，テストの効率が悪い．本研究では，プログラムからコアの実行を制御できる機構を用いたテスト効率化手法を提案する．テストプログラムにコアの実行を制御するための記述を追加しコアの実行を制御することで，プログラム中の特定のパスを決定的に実行し，テストの効率化を実現できる．提案手法をマルチコアに対応したリアルタイムOSであるTOPPERS/FMPカーネルに適用し，提案手法の有用性を示す．

最後に，リアルタイム性と実行オーバヘッドの両立を目指した通信ミドルウェアを提案する．対象は，車載ソフトウェアアーキテクチャ仕様であるAUTOSARの通信ミドルウェアである．本研究では，まず，通信ミドルウェアをマルチコア拡張する際に求められる要件を示し，既存手法では，実行オーバヘッドとロック取得時間の低減を両立できないことと，実行並列性がないことを明らかにする．そして，これらの問題の解決を目指した手法を2つ提案し，実装および評価を行う．各手法の評価結果と要件とを比較し，各手法の特徴を明らかにする．また，既存手法は通信ミドルウェアが持っている定期通信の機能を考慮しておらず，リアルタイム性を損なわないように実装を行うと，実行オーバヘッドが大きくなってしまう問題があることを示す．

本研究では，マルチコアプラットフォームが果たす基本的な役割である排他制御について，リアルタイム性と実行オーバヘッドを両立する手法を提案した．また，マルチコアプラットフォームについて，テストの効率化手法を提案した．以上の内容により，マルチコアプラットフォームの構築に関する3つの課題を解決する技術を示した．

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究背景

各種の機器に組み込まれ，機器と一体で1つとみなされるコンピュータシステムを，組込みシステムと呼ぶ．組込みシステムの多くは，満たすべき時間要件（時間制約）に従って動作する性質（リアルタイム性）を要求されるリアルタイムシステムである．

現在，パーソナルコンピュータやスーパーコンピュータなどの汎用システムのほとんどは，複数のコアを用いて処理を実行するマルチコアシステムである．組込みシステムにおいても，大規模化・複雑化が進み，マルチコアシステムの重要性が増している．その背景には，消費電力の増大を抑えつつスループットの向上を図るためには，クロック周波数を上げるよりも，コア数を増やしたほうが有利であるという状況がある．

一方，組込みリアルタイムシステムのソフトウェア開発においては，リアルタイムOS（以下，RTOSと呼ぶ）やネットワークプロトコルスタックなどのソフトウェアプラットフォームを用いることが多い．プラットフォームを用いることで，プラットフォームから提供される限定的・明示的なインタフェースによってシステムに対する要求を実現することができ，アプリケーションソフトウェアの保守性や再利用性の向上を実現できる．

マルチコアシステムの重要性が増すにつれて，マルチコアシステムをサポートするプラットフォーム（以下，マルチコアプラットフォームと呼ぶ）の重要性も増している．マルチコアプラットフォームの役割は大まかに2つある．1つは，シングルコアシステムで動作していた従来のアプリケーションの再利用である．これにより，開発コストを削減できる．もう1つの役割は，ハードウェアの隠蔽である．マルチコアシステムでは，複雑なハードウェアの構成となっている場合や，マルチコア特有のハードウェア技術を用いている場合がある．そのため，マルチコアシステムのハードウェア知識を持たなければ，その性能を十分に発揮できないばかりか，正しく動作するソフトウェアを作成することもできない．特に，コア間

の排他制御を実現する適切な方法はハードウェアごとに異なり，その実現方法は，性能という点でも正しく動作するという点においても非常に重要である．マルチコアプラットフォームがハードウェアに関する部分を隠蔽することで，アプリケーション開発者に求められるハードウェア知識を減らすことができる．マルチコアシステム上のソフトウェアは，複雑なものになりやすく，マルチコアプラットフォームの果たす役割は大きいといえる．

マルチコアプラットフォームを構築するにあたって解決すべき課題は多い．組込みリアルタイムシステムで用いるためには，特に以下の項目が重要となる．

- リアルタイム性
  プラットフォームとして，リアルタイム性を保証するための仕組みを持っていなければ，システム全体としてリアルタイム性を保証することができない．具体的には，プラットフォームとしてアプリケーションに提供する API の最悪実行時間が定まることや，割込み応答時間が短いこと，といった性質を備えていることが求められる．

- 実行オーバヘッド
  実行オーバヘッドは，実行時間の増分のことで，本論文ではコア数を増やしたことによる増分に着目する．

- 信頼性
  プラットフォームは様々なアプリケーションで利用されることを想定するため， その動作による不具合が存在すると多数のシステムに影響を与えてしまう．そのため， プラットフォームが提供する動作による不具合が発生しないことが求められる．特に，組込みリアルタイムシステムでは，製品化された後でソフトウェアの更新により不具合を修正することが困難であったり，非常に多くの費用を要したりする場合があるため，不具合のないことが重視されることが多い．不具合が発生しやすいのは，割込み処理や排他制御に関連する部分であり，マルチコアプラットフォームにおいてはデッドロックが発生しないことが重要である．

マルチコアプラットフォームの課題のうち，実行オーバヘッドのみであれば，Linux や FreeBSD などの汎用システムを対象としたマルチコアプラットフォームで用いられている技術を流用できる．組込みリアルタイムシステムにおいては，リアルタイム性が求められるため，リアルタイム性と実行オーバヘッドの両立が重要な

課題となる．この 2 つがトレードオフとなって両立が困難な場合は，リアルタイム性の保証を重視する場合がある．言換えると，リアルタイム性を保証するために，実行オーバヘッドが多少増加することは，許容される場合がある．また，組込みリアルタイムシステムを対象としたプラットフォームは，リアルタイム性を保証するために割込みや排他制御に関連する処理が複雑になっている場合がある．信頼性を高めるためには，テストによってその処理が正しく動作することを検証する必要がある．少なくとも，すべての分岐を網羅するテストを実施する必要があると考えられる．

以上の課題に対応した，組込みリアルタイムシステムを対象としたマルチコアプラットフォームが必要である．

1.2　論文の概要

本研究の目的は，組込みリアルタイムシステムに適したマルチコアプラットフォームを構築するため，前節で述べた 3 つの課題を解決する技術を提案することである．特に，ソフトウェアプラットフォーム内で行う排他制御に着目する．排他制御と 3 つの課題について，次のような関係がある．排他制御によって他のコアに処理を待たされると，API の最悪実行時間が長くなる．他のコアの実行状態が関わるため，最悪実行時間がいくつであるかを容易に知ることができず，リアルタイム性を損なう恐れがある．他のコアに処理を待たされない場合においても，排他制御のための処理を実行する時間が長いと，実行オーバヘッドも長くなる．さらに，信頼性を高めるためには，適切に排他制御を行い，デッドロックのような不具合が発生しないことが重要である．プラットフォームは様々なシステムで利用されるため，プラットフォームが排他制御によって発生する，これらの課題を解決できなければ，システム全体としても課題を抱えたままとなる．そのため，プラットフォームとして課題を解決する手法が必要である．

排他制御によって発生する問題は，2 つに分けることができる．排他制御アルゴリズムの問題と，排他制御の使い方による問題である．本研究では，排他制御アルゴリズムと排他制御の使い方の 2 点において，リアルタイム性と実行オーバヘッドを両立するための方法を提案する．また，マルチコアプラットフォームの信頼性を向上させる取り組みとしてソフトウェアテストの効率化手法を提案する．

本研究の具体的な内容は以下の 3 つである．

まず，コア間の共有リソースを排他する際に用いる排他制御アルゴリズムを提

案する．マルチコアプロセッサを用いたリアルタイムシステム向けの排他制御アルゴリズムとして，数多くのキューイングスピンロックアルゴリズムが提案されている．これらの多くは，単一のロックを取得する状況を想定しているため，複数のロックを同時に取得する場合には，コア数が増加した際の実行オーバヘッドとリアルタイム性を両立できないという問題がある．本研究では，スピンロックが満たすべき要件を示し，提案アルゴリズムが全ての要件を満たすことを示す．さらに，提案アルゴリズムをハードウェアで実装することで高速に実行できる手法について述べる．

2 つ目として，マルチコアプラットフォームを構成するソフトウェアのテスト効率化手法を提案する．マルチコアシステムにおいては，ソフトウェアプラットフォーム内に各コアの実行順序に依存してパスが定まる分岐（実行順序依存分岐と呼ぶ）が存在する．この分岐は，各コアが並列に実行することによって発生する不都合な状態に対応するために，ソフトウェアプラットフォームの開発者が意図的に作成したものである．ソフトウェアプラットフォームの信頼性を向上させるためには，実行順序依存分岐のすべてのパスを網羅し，開発者が意図した通りに動作することを検証する必要がある．網羅する方法として，実行順序依存分岐を実行する可能性のあるプログラムを繰り返し実行する手法（連続試行手法と呼ぶ）が考えられる．テストプログラムによって各コアの実行順序を変動可能であることを前提とすれば，十分な数の試行を繰り返すことによって網羅性が期待できる．ここで，パス実行の可能性は一般に一様でないため，連続試行手法で実行順序依存分岐を網羅するための繰り返し回数は，実行順序に依存しない分岐の組み合わせを網羅する数と比較して，非常に多くの回数となる．したがって，連続試行手法は，テストの実施や不具合の分析に多くの手間を要するため，テストの効率が悪い．この問題に対して，プログラムからコアの実行を制御できる機構を用いたテスト効率化手法を提案する．テストプログラムにコアの実行を制御するための記述を追加しコアの実行を制御することで，プログラム中の特定のパスを決定的に実行し，テストの効率化を実現できる．提案手法をマルチコアに対応したリアルタイム OS である TOPPERS/FMP カーネルに適用し，提案手法の有用性を示す．

最後に，リアルタイム性と実行オーバヘッドの両立を目指したプラットフォームのマルチコア拡張手法を提案する．対象は，車載ソフトウェアアーキテクチャ仕様である AUTOSAR の通信ミドルウェアである．本研究では，まず，通信ミドルウェアをマルチコア拡張する際に求められる要件を示し，既存手法では，実行オーバヘッドとロック取得時間の低減を両立できないことと，実行並列性がない

ことを明らかにする．そして，これらの問題の解決を目指した手法を 2 つ提案し，実装および評価を行う．各手法の評価結果と要件とを比較し，各手法の特徴を明らかにする．また，既存手法は通信ミドルウェアが持っている定期通信の機能を考慮しておらず，リアルタイム性を損なわないように実装を行うと，実行オーバヘッドが大きくなってしまう問題があることを示す．

1.3　論文の構成

本論文の構成は，次の通りである．

2 章では，本研究の前提となる事項である，リアルタイム性や排他制御について述べる．さらに，本研究で用いるプラットフォームの概要について述べる．3 章では，実行オーバヘッドがコア数のリニアオーダである（コア数に対するスケーラビリティをもつ）こととリアルタイム性とを両立した排他制御アルゴリズムについて述べる．4 章では，マルチコアリアルタイム OS のテスト効率化手法について述べる．5 章では，リアルタイム性と実行オーバヘッドの両立を目指したプラットフォームの構築事例として，通信ミドルウェアのマルチコア拡張の手法について述べる．最後に，6 章で結論を述べる．

第2章　プラットフォームと組込みリアルタイムシステム

2.1　プラットフォーム

2.1.1　プラットフォームの定義

プラットフォームとは，その上でアプリケーションが動作するための基盤となるハードウェアおよびソフトウェアのことである．特に，プラットフォームを構成するソフトウェアを，ソフトウェアプラットフォームと呼ぶ．ソフトウェアプラットフォームは，アプリケーションから呼び出されたときに必要な処理を実行する．呼び出し方法はシステムコールや API としてあらかじめ規定されている．ソフトウェアプラットフォームの代表例は，OS である．OS の他には，通信スタックやファイルシステムなどのミドルウェアがあげられる．ハードウェアとしてシングルコアシステムを用いたプラットフォームをシングルコアプラットフォーム，マルチコアシステムを用いたプラットフォームをマルチコアプラットフォームと呼ぶ．

2.1.2　プラットフォームと排他制御

プラットフォームの上で動作するアプリケーションは複数の処理で構成されることが多い．また，アプリケーションが複数存在する場合もある．複数の処理が共有して利用するリソースが存在する場合，ある処理がリソースにアクセスしている間は，他の処理がアクセスできないように排他制御する必要がある．共有リソースとしては，入出力装置やメモリが考えられる．

排他制御の階層は少なくとも 2 つある．1 つはアプリケーションの処理間で，もう 1 つはプラットフォーム内部である．前者はアプリケーションが用いる共有リソースへの操作，後者はプラットフォームが管理するデータへの操作に一貫性を持たせるためである．アプリケーションの処理間での排他制御は，プラットフォームが提供するセマフォなどの機能を用いて実現される．この機能を実現するため

には，セマフォ変数などのプラットフォーム内部のデータについて，操作に一貫性を持たせる必要がある．このため，アプリケーションの処理間での排他制御は，プラットフォーム内部の排他制御によって実現される．一方，プラットフォーム内部の排他制御を実現する方法は，ハードウェアによって異なる．実現方法の詳細は，次節以降で述べる．

2.1.3　シングルコアプラットフォーム内部の排他制御

シングルコアプラットフォームでは，割込み禁止状態にすることで排他制御を実現できる．これは，割込み禁止状態では，割込み処理はもちろんのこと，割込み処理でない通常の処理を切り替える（ディスパッチする）ことをプラットフォームが実行しなくなるためである．

2.1.4　マルチコアプラットフォーム内部の排他制御

マルチコアプラットフォームでは，排他制御はコア内とコア間に分かれる．コア内の排他は，シングルコアプラットフォームと同様に割込み禁止によって実現できる．コア間の排他は，複数のコアの処理が同時にリソースにアクセスすることを防ぐ必要があるため，一般に，割込み禁止では不十分である．コア間排他制御を実現するためには，スピンロックを用いることが多い．

スピンロックは，共有リソースへのアクセス権を得るまで（ロックを取得するまで）チェックを繰り返す．チェックを繰り返す間は，ロック取得を待っている状態であり，本質的な処理を行っていない．ロックを取得したら，クリティカルセクションに入り，共有リソースを用いた処理を行う．そして，ロックを解放しクリティカルセクションを抜ける．

割込み禁止とスピンロックを用いることで，コア内とコア間の排他を実現することができる．しかし，これらは互いに関連しており，どちらを先に実行するかによって以下の問題がある．まず，割込み禁止後にスピンロックを取得する場合，スピンロックの取得待ちの間は割込み処理状態であるため，割込み処理の応答時間が長くなってしまう（図 2.1）．一方，スピンロックを取得後に割込み禁止とする場合，スピンロックを取得した後から割込み禁止状態となるまでの間に，割込みが発生して割込み処理を実行してしまう可能性がある．この場合，スピンロックを取得したまま，排他すべき処理とは全く関係のない割込み処理を実行してしま

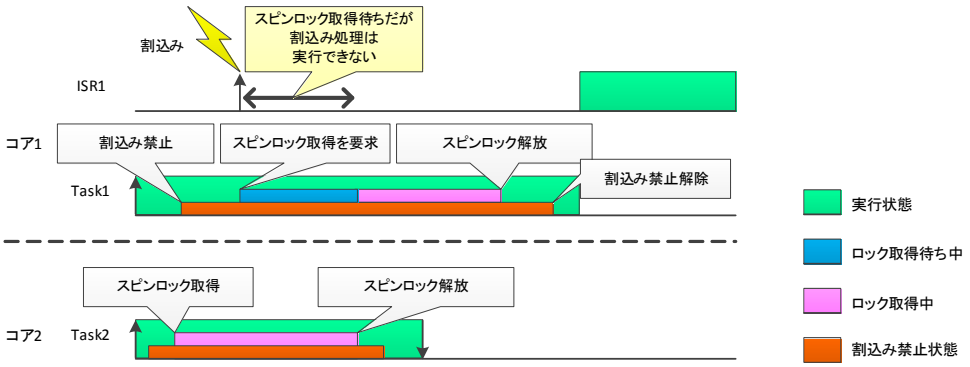


図 2.1: 割込み禁止後にスピンロックを取得する場合

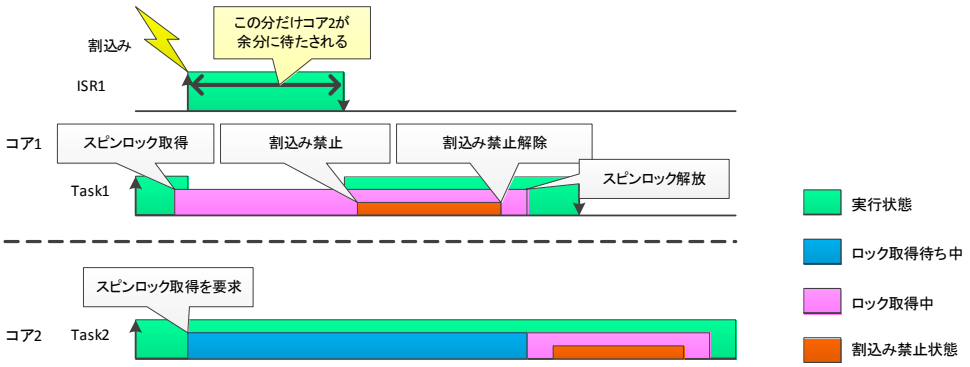


図 2.2: スピンロック取得後に割込み禁止する場合

うこととなり，他のコアではスピンロックの取得待ち時間が長くなってしまう（図 2.2）．

この問題を解決するためには，割込みを禁止してからスピンロックの取得待ちとなり，スピンロック取得待ちの間に割込みが発生した場合は割込み処理を実行すればよい．スピンロックを取得中に割込みが発生した場合，スピンロックを解放するまで割込み処理が実行されないことは許容するものとする．

2.1.5　アトミック命令によるコア間排他制御の実現

コア間の排他制御を実現するためには，ハードウェアによるサポートを用いるのが一般的である．具体的には，アトミック命令や排他制御のための専用回路などを用いる．

アトミック命令は，コア間で共有するメモリに対して，他のコアの影響を受けることなく，複数のメモリアクセスを実現する命令である．test-and-set (TAS) 命

令，compare-and-swap (CAS) 命令，Load-Linked/Store-Conditional (LL/SC) 命令などがある．

TAS 命令は，指定されたメモリアドレスの値を読出し，値をセットし，セットする前に読出した値を返す．値の読出しと値のセットが不可分に行われるため，返された値をチェックすることで，そのコアが TAS 命令によって値をセットできたのかを知ることができる．値がセットできなかったとき，つまり，返された値が別の値であったときは，別のコアが値をセットしたということを意味する．例えば，TAS 命令で，あるメモリアドレスについて，値が 0 である状態で 1 を書込むことを試す．期待する返却値は 0 であるが，1 が実際に返ってきたときには，同じ処理を別のコアが実行しており，そのコアでの TAS 命令の返却値が 0 であったことになる．返却値が期待値である 0 に一致するまで TAS 命令の実行を繰り返すことによって，スピンロックを実現できる．このスピンロックを TAS スピンロックと呼ぶ．

CAS 命令は，比較と書込みをアトミックに行える．比較と書込みの値，メモリアドレスの 3 つを指定する．CAS 命令は，指定されたメモリアドレスの値を読出し，その値が指定された値と一致したときに，指定した値を書込む．さらに，値を書込む前に読出した値を返す．CAS 命令は，TAS 命令に相当する処理を行えるので TAS スピンロックを実現できる．

LL/SC 命令は，LL 命令と SC 命令の組である．LL 命令では，指定されたメモリアドレスの値を返す．単に値を返すだけでなく，その後，指定されたメモリアドレスに対するアクセスを監視し，SC 命令ではそのアドレスへの書き込みがなかった場合にのみ，そのアドレスに値を書き込む．LL 命令と SC 命令の間に，既に書込みがあった場合には，SC 命令による値の書込みは失敗する．たとえ LL 命令で読出した値と同じ値を書き込んだとしても，SC 命令は失敗する点がこの命令の重要な点である．これにより，値が A から B に変化し，さらに A に戻ったことを検出することが可能になる．

CAS 命令と LL/SC 命令は，TAS スピンロックを実現できる．しかし，実際には TAS スピンロックを実装するために使われるのではなく，より複雑な排他制御アルゴリズムを実装するために使われる．

2.1.6　マルチコアプラットフォームにおける処理のコア割当て方法

組込みリアルタイムシステムを対象としたマルチコアプラットフォームにおいては，リアルタイム性を保証するため，処理のコア割当て方法に注意が必要である．リアルタイム性とは，満たすべき時間的な要件に従って動作する性質のことである．

時間制約が強い処理の場合は，実行時間の予測をしやすくするため，関連のある処理はできる限り 1 つのコアに割当てる方が良い．関連のある処理を複数のコアに割当てると，コア間での通信や排他制御などが必要となり，各コアの実行状態によって処理の実行時間が変動するため，実行時間の予測が困難となる．一方で，時間制約が弱い処理については，実行時間の予測可能性だけでなくスループットの向上も求められる．この場合は，各コアの負荷が同じになるように，処理をコアに動的に割当てることでスループットの向上が図れる．

組込みリアルタイムシステムでは，あらかじめ実行すべき処理がわかっているのが通常である．実行すべき処理内容とその実行時間が分かっているならば，各コアの負荷が同じになるよう，そしてコア間の通信や排他制御が少なくなるように，処理をコアに割当てることができる．

以上のことから，組込みリアルタイムシステムにおいては，マルチコアプラットフォームが処理をコアに動的に割当てることは避けられる．ただし，時間制約が弱い処理のスループット向上のために，コアを指定して動的に処理を割当てるシステムコールもしくは API をサポートすることがある．

2.2　本研究で対象とするプラットフォーム

2.2.1　TOPPERS/FMP カーネル

ITRON 仕様は，組込みシステム用の RTOS とそれに関連する仕様である．国内では，ITRON 仕様に準拠した多くの RTOS が開発，製品化されてきた．現在は，μITRON 4.0 仕様が公開されている．なお，μITRON 4.0 仕様は，シングルコアを対象としており，マルチコアは対象となっていない．

TOPPERS プロジェクト [24] では，μITRON 4.0 仕様に準拠した JSP カーネルをはじめ，様々な開発成果物をオープンソースで公開している．μITRON 4.0 仕様をベースとし，マルチコアやメモリ保護機能への対応などの拡張をした TOPPERS 新世代カーネル仕様を定めている．TOPPERS 新世代カーネル仕様に準拠したシ

ングルプロセッサ用の RTOS として TOPPERS/ASP カーネル（以下，ASP カーネル）をオープンソースで公開している．マルチプロセッサ用の RTOS としては，TOPPERS/FDMP カーネル [34]（以下，FDMP カーネル）と，FDMP カーネルをベースに作成された TOPPERS/FMP カーネル [35]（以下，FMP カーネル）を，オープンソースで公開している．FDMP カーネルと FMP カーネルは，本章で述べた割込み禁止と排他制御の問題と処理のコア割当てに関する問題が発生しないよう実装してある [34][35]．具体的には，前者についてはロック取得に失敗したときに一時的に割込み許可とすることで，ロック取得待ち中に割込み処理を実行することができる．後者については，コアごとに処理のスケジューリングを行い，コア割当ての変更が必要であれば専用のシステムコールをアプリケーションが呼ぶことができる．

本論文では，3 章および 4 章で，FMP カーネルを用いて評価を行っている．

2.2.2　AUTOSAR 通信ミドルウェア

AUTOSAR (AUTomotive Open System ARchitecutre) [1] は，車載システムにおけるソフトウェアアーキテクチャの標準化を目指す団体およびその仕様である．AUTOSAR 仕様には，RTOS や通信ミドルウェアなどが含まれており，AUTOSAR アーキテクチャを構成するモジュールごとに仕様を定めている．これらの仕様は，OSEK/VDX (Offene Systeme und deren schnittstellen fur die Elektronik im Kraftfahrzeug/Vehicle Distributed eXecutive) [4] によって定められた仕様をベースに拡張が行われている．AUTOSAR では，仕様は企業間の非競争領域であるが，その実装は競争領域であるとしているため，AUTOSAR としてはプラットフォームの実装を公開していない．AUTOSAR 仕様には，本章で述べた割込み禁止と排他制御の問題について言及がないため，この問題について詳細は不明である．処理のコア割当てに関しては，AUTOSAR OS の仕様上，静的に処理をコアに割当てることが決まっている．

本論文では，5 章で，AUTOSAR 仕様の通信ミドルウェアをマルチコアプラットフォームとして構築する方法を述べている．

2.3　組込みリアルタイムシステム

リアルタイムシステムにおいては，満たすべき時間的な要件に従って動作する性質（リアルタイム性）を持つことが重要となる．特に，ある処理の最悪実行時間を予測できることが重要である．これは，処理を終える時間が時間制約（デッドライン）を満たせなかった場合に，重大な損失を被る場合が存在するためである．例えば，自動車の衝突防止システムでは，前方の車両や障害物に衝突すると判断した場合にブレーキを制御して被害の軽減を図るが，ブレーキを制御するまでの処理が遅れた場合は人間の生命に危険が及んでしまう．

また，組込みシステムでは，機器を制御する必要から割込み処理を作成する場合が多く，割込み処理を含めた処理がデッドラインを満たす必要がある．割込み処理で問題となるのは，システムが割込みを禁止しているために，割込みの要求が発生してから実際に割込み処理を実行するまでの時間が長くなってしまい，デッドラインを満たせなくなることである．そのため，要求が発生してからの応答が高速であることが求められる．

排他制御のリアルタイム性について述べる．TAS スピンロックは，非常に簡単に実装できる排他制御アルゴリズムであるが，ロックを取得できる順番はランダムに決まる．このため，次にようにロック取得にかかる最悪実行時間が定まらない問題がある．コア 1，コア 2，コア 3 の 3 つのコアが TAS スピンロックにより同一のロックを取得する場合を考える．まず，コア 1 がロックを取得できたとする．そして，コア 1 がロックを解放した後には，コア 2 がロックを取得できたとする．次に，コア 2 がロックを解放する前に，コア 1 が再び同じロックを取得しようとすると，コア 2 がロックを解放した後に，コア 3 ではなくコア 1 がロックを取得する可能性がある．さらに，その後コア 2 が同じロックを取得しようとし，実際にロックを取得してしまう可能性もある．このように，ランダムにロック取得するコアが決まると，ロック取得の最悪実行時間は理論上，定めることができず，無限大となってしまう．なお，コア数が 2 つのシステムにおいては，交互にロックを取得できるため，ロック取得の最悪実行時間は容易にわかる．

ロック取得順を定めて，ロック取得の最大実行時間を定める方法としてキューイングスピンロックアルゴリズムが存在する．これは，ロック取得を要求した順番をキューで管理し，その順番通りにロック取得ができるアルゴリズムである．基本的なアルゴリズムとして，MCS ロック [16] が提案されている．キューイングスピンロックアルゴリズムを実装するには，一般に TAS 命令では不十分であり，CAS 命

令や LL/SC 命令を必要とする．そのため，これらの命令に相当する機能を備えたシステムでなければキューイングスピンロックアルゴリズムを用いることができない．

2.4　車載システム

CAN (Controller Area Network) は，自動車内のネットワークとして事実上の標準となっている通信プロトコルである．ドイツの Robert Bosch 社が 1980 年代に提唱し，その後，ISO により標準化された [9]．

CAN はバス型のシリアル通信プロトコルで，最大通信速度は 1Mbps である．CAN では，バスが空いていればすべてのノードが送信可能で，すべてのノードがメッセージを同時に受け取る．基本的には，バスに最も早くアクセスしたノードが送信権を得る．複数のノードが同時に送信を開始した場合には，優先度の高いメッセージを送信するノードがバスの使用権を得る．メッセージの優先度はメッセージ ID によって定まり，その値が小さいほど高優先度になる．最高優先度のメッセージは他のメッセージに邪魔されることなく送信できる．なお，CAN では優先度を比較する仕組み上，メッセージ ID はすべてのノード間でユニークなものにしなければならない．

自動車には，ECU (Electronic Control Unit) と呼ばれるコンピュータが搭載される．近年，1 台の自動車に搭載される ECU の台数は，増加傾向にある．これは，エンジンやインパネ，カーナビなど制御対象ごとに異なる ECU を用いることが多いこと，また，新しい機能を追加する際に必要な ECU を都度追加することが要因である．

ECU は CAN などの通信プロトコルを利用して相互に接続されており，電子制御によって実現される機能の多くは複数の ECU を用いて実現される．電子制御もしくは ECU によって実現される機能が，自動車の付加価値を決める要素となっており，車載システムの重要性が高まっている．

車載システムは，その用途によって求められる性質も異なる．最も厳しい性質を要求されるのは，エンジンやブレーキなどの制御系と呼ばれるシステムである．制御系には極めて高い信頼性，リアルタイム性が求められる．一方で，カーナビなどのマルチメディア系と呼ばれるシステムは，制御系ほどの信頼性やリアルタイム性は要求されない．このように，自動車は多くの機能・性質を持っており，それらが複雑に組み合わさって構成される，組込みシステムである．

第3章　中断可能な優先度継承キューイングスピンロックアルゴリズム

3.1　概要

マルチコア向けRTOSのリアルタイム性を実現するためのスピンロック方式に関して，様々な研究が行われており，単一のロックを取得する場合について，コア数に対するスケーラビリティとリアルタイム性を両立した，中断可能なキューイングスピンロックと呼ばれるアルゴリズムが提案されている [32].

FDMP カーネルでは，多くのシステムコールが2個のロックを同時に取得する必要がある [33]．しかしながら，2個ないしそれ以上のロックを取得する場合については，コア数に対するスケーラビリティを確保しつつリアルタイム性を満たすスピンロックアルゴリズムは提案されていない．

本章では，RTOS内部で2つのロックを取得する場合について，コア数に対するスケーラビリティとリアルタイム性を両立した，中断可能な優先度継承キューイングスピンロックアルゴリズムを提案する．提案手法は，エンジン制御などの最悪実行時間の予測性や高速な割込み応答性が求められるハードリアルタイムシステムに有効な手法である．アルゴリズムの一部をハードウェア化することにより，アルゴリズムの実現にCAS命令やLL/SC命令を必要としないため，様々なプロセッサで利用可能である．

3.2　RTOS向けスピンロック

3.2.1　要件

リアルタイムシステムにおいては，ある処理の実行時間や応答時間に上限があることが求められる．マルチコアプロセッサシステムにおいては，さらに，コア

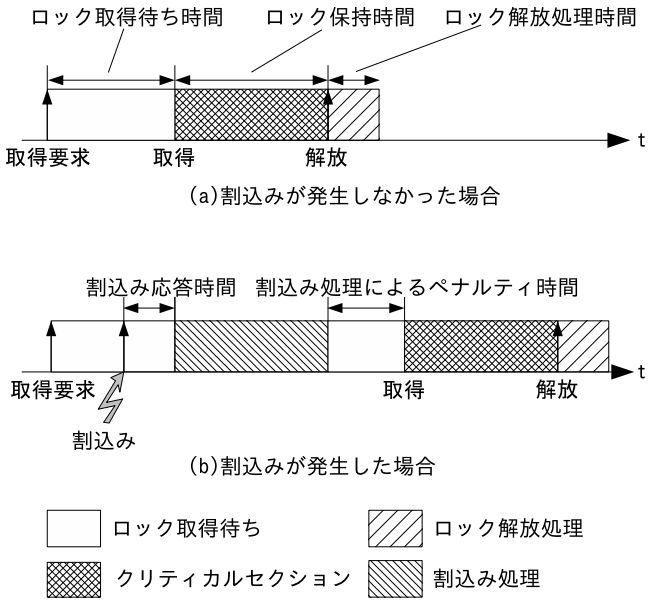


図 3.1: 3 つの要件に対応する時間

数に対するスケーラビリティを確保することが重要である．スピンロックについては，ロック取得・解放処理の時間に上限があり，かつ，その上限がスケーラビリティを持つことが求められる．文献 [33] では，マルチコア対応 RTOS が満たすべき要件を挙げており，これを基に，2 つの性質を両立させるために，スピンロックが満たすべき要件を以下のように定めた．

1. ロック取得待ち時間とロック解放処理時間の最悪実行時間がコア数のリニアオーダで抑えられる

2. クリティカルセクション以外では，割込みを受付けることができ，割込み応答時間がコア数に依存しない

3. 割込み処理によるペナルティ時間がコア数に依存しない

(1) に関して，まず，ロックが既にあるコアに保持されている場合は，解放されるまで他のコアはロックを取得することができない．ロック取得待ち時間は，ロック取得で競合するコア数によって変動するのは避けられないが，その最悪実行時間が定まることがリアルタイム性を満たすために必要である．さらにコア数に対するスケーラビリティを確保するためには，ロック取得待ち時間の変動をコア数

のリニアオーダで抑える必要がある．また，アルゴリズムによってはロック取得待ち時間だけでなくロック解放処理時間も競合するコア数によって変動するので，取得と解放両方の最悪実行時間がコア数のリニアオーダで抑えられることが求められる．図 3.1(a) に，ロック取得待ち時間とロック解放処理時間を示す．

(2) に関して，多くの場合，ロック保持時間に比べると割込み処理の時間は長いため，ロック保持中の処理であるクリティカルセクションは割込み禁止にする．クリティカルセクションに加えて，ロック取得待ちの間も割込みを禁止すると，競合するコアが多い場合にロック取得待ち時間と共に割込み禁止時間も長くなり，割込み応答性が低下する．ここで割込み応答とは，割込みが発生してから割込み処理が実行されるまでの時間で，図 3.1(b) で割込み応答時間とした部分である．割込み処理は高い応答性が求められるため，ロック取得待ちの間は割込み処理を受付けることが望ましい．また，割込み応答時間がコア数に依存するとシステムのスケーラビリティが著しく損なわれてしまう．

(3) について，まず，割込みを受付けることによりロック取得待ち時間が増える．割込みによって増加するのは，割込み処理に遷移してから復帰するまでの処理時間だけではない．割込みを受付けることでロックを取得できる順番が変化し，割込み処理から復帰した後からロック取得するまでの待ち時間が増加する場合がある．この増加時間を，割込み処理によるペナルティ時間と呼ぶことにする．図 3.1(b) で，割込み処理によるペナルティ時間とした部分が割込み処理から復帰した後に増えた待ち時間である．割込み処理によるペナルティ時間がコア数に依存すると，割込みが発生した場合のロックの取得待ちと解放処理の最悪実行時間がコア数のリニアオーダに抑えることができない．したがって，割込み処理によるペナルティ時間がコア数に依存しないことが求められる．

3.2.2　シングルロックのロックアルゴリズム

本論文では，単一のロックを取得することをシングルロックと呼ぶ．シングルロックの最も単純なスピンロックとして，TAS スピンロックがある．これは，ロック変数のチェックと書込みをアトミックに行える TAS 命令を利用し，ロックが取得できるまでチェックを繰り返すというものである．このアルゴリズムでは，ロックを取得できる順番はランダムに決まるため，要件 (1) を満たせない．要件 (2) を満たすように実装するのは容易である [34]．TAS スピンロックでは，ロック取得順がランダムなため，割込み処理によるペナルティ時間はなく，要件 (3) を満たす．

TAS スピンロックのようにロック取得の順番がランダムに定まる場合は，ロック取得待ち時間の上限を抑えることができない．そのため，ロックの取得順をキューで管理するキューイングスピンロックアルゴリズムが提案されている．単純なキューイングスピンロックアルゴリズムである MCS ロック [16] は，ロック取得待ちの間に割込み処理を受付けることができないため，要件 (2) を満たせない．

ロック取得待ち中に割込み処理を受付けることができるキューイングスピンロックアルゴリズムとして，2 種類の中断可能なキューイングスピンロックアルゴリズムが提案されている [32]．なお，要件 (3) については，アルゴリズムによって異なる．文献 [32] のアルゴリズム 1 は，割込み処理から復帰後にキューの最後尾に並ぶため，ロック取得までの待ち時間がコア数に依存する．従って，要件 (3) を満たさない．もう一方のアルゴリズム 2 は割込み処理から復帰後もキューの位置は変わらないため要件 (3) を満たす．

3.2.3　ネストロックの必要性

複数のロックをネストして取得することをネストロックと呼ぶ．

マルチコアシステムにおいては，一つのロックで排他制御を行う共有資源の単位（ロック単位と呼ぶ）を適切に設定する必要がある [34]．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，対応するロックを全て取得しなければならない．2 つの共有資源に同時にアクセスし，そのロック単位が異なる場合は，同時に 2 つのロックを取得できないため，ロックを 2 段階で取得する．つまり，次の手順で排他制御を行う．まず，1 段目のロック，2 段目のロックを順に取得する．そして，2 つのロックを取得した状態で 2 つの共有資源を使った処理を行う．最後に，2 段目のロック，1 段目のロックを順に解放する．2 つより多くのロックが必要な場合は，さらに段数（ネスト）が深くなる．

RTOS 内部で用いるロック単位について，全ての管理データをまとめて一つのロック（ジャイアントロック）にしてしまうと，コア内に閉じた処理であってもロックの競合が発生し，他のコアの処理を阻害してしまう．このように，ジャイアントロックでは，マルチコアの利点をいかせないため，最低でもコアごとに別々のロック単位を設定する必要がある．

FDMP カーネルでは，コアごとに 2 種類のロックを用意し，システムコールでは，最大 2 つのロックを同時に取得できれば，コア数に対するスケーラビリティとリアルタイム性を両立可能である [34][33]．そのため，本研究では，2 つのロック

を取得するネストロックのアルゴリズムを対象とする．

3.2.4 既存ネストロックアルゴリズムの問題点

コア数を N とすると，シングルロックにおいてロック取得までの時間が $O(N)$ のアルゴリズムであっても，2 つのロックをネストして取得すると，ロック取得までの時間が $O(N^2)$ となってしまう．この問題について，$O(N)$ の Totally FIFO アプローチが提案されており [23]，次の手順をとっている．1 段目のロック取得前にタイムスタンプを取得しておき，これを 1 段目と 2 段目のロック取得時の優先度とする．2 段目のロックは優先度順キューイングスピンロックであるが，タイムスタンプを優先度として用いることで，全体としてロックの取得順は FIFO 順となる．よって，Totally FIFO アプローチは要件 (1) を満たす．しかし，割込みについて言及がなく，そのままではロック取得待ちの間に割込みを受付けることができないため，要件 (2) を満たさない．つまり，Totally FIFO アプローチは，割込み応答性が低下する問題がある．これは，Totally FIFO アプローチでは，ロック取得待ちの間も割込み禁止となるため，ロック取得時間が長い場合に，割込み禁止時間も長くなるためである．

3.2.5 割込み受付け可能な Totally FIFO アプローチにおける優先度逆転

Totally FIFO アプローチを割込みが受付けられるように拡張するためには，1 段目のロックと 2 段目のロックで割込みを受付けられるアルゴリズムを使う必要がある．具体的には，中断可能なキューイングスピンロック [33] と中断可能な優先度順キューイングスピンロックが必要である．優先度順キューイングスピンロックは既に提案されており [31]，文献 [33] で提案された方法を適用すれば中断可能な優先度順キューイングスピンロックを実現することができる．この 2 つを用いて，割込み受付け可能に拡張された Totally FIFO アプローチを割込み受付け可能な Totally FIFO アプローチと呼ぶ．

割込み受付け可能な Totally FIFO アプローチでは，割込み処理を受付けることによって，2 段目のロック取得順が優先度順[1]とならない場合，つまり優先度逆転

[1]本研究では，タイムスタンプを優先度として使うため，優先度順はタイムスタンプ順と同義である

が発生する場合がある．割込み受付け可能な Totally FIFO アプローチは，優先度逆転の区間がコア数に依存する問題があるため，割込み処理によるペナルティ時間がコア数に依存してしまい，要件 (3) を満たさない．

優先度逆転の例を図 3.2 に示す．ロック L1 と L2 があり，割込み処理を受付けるよう拡張した Totally FIFO アプローチでコア 1 とコア 2 が L1，L2 の順にネストロックを取得し，コア 3，コア 4，コア 5 が L2 のシングルロックを取得する場合を考える．ここでは，優先度として用いるタイムスタンプは 1 から始まり，数が小さいほどロック取得の優先度が高いものとする．t1 においてコア 1 が優先度 2 で L1 を取得しようとしたときには，コア 2 が既に優先度 1 で L1 を保持している．そのため，コア 1 は L1 の取得待ちになる．その後，コア 1 に割込みが発生し，コア 1 は割込み処理を実行する．コア 1 が割込み処理を実行している間に，コア 2 は L2, L1 を解放する．t2 でコア 1 が割込み処理から復帰する前に，L2 について，コア 3 が優先度 3 で，コア 4 が優先度 4 で，コア 5 が優先度 5 で取得要求を出す．次に，コア 2 が優先度 6 で L1，L2 を順に取得しようとすると，コア 2 は L2 を取得する際にコア 3，コア 4，コア 5 に待たされる．そして，t2 でコア 1 が割込み処理から復帰して，L1 を再び取得しようとする．Totally FIFO アプローチでは，コア 1 のタイムスタンプが最も古いため他のコアより高い優先度を持つが，コア 2 が L1 を保持しているので，コア 1 は他の 4 つのコアに待たされてしまう．このように，高い優先度のコアが低い優先度のコアに待たされることを優先度逆転と呼ぶ．さらに，コア数が増え，そのコアがコア 2 より先に L2 を取得すると，優先度逆転の区間が長くなる．コア 1 が割込み処理から復帰したときに発生する優先度逆転の区間はコア数に比例して長くなるため，要件 (3) を満たすことができない．

3.3　PPIQL アルゴリズム

Totally FIFO アプローチをベースに，2 段のネストロックについて 3 つの要件を満たすよう拡張した PPIQL (Preemptive Priority Inheritance Queueing spin Lock) アルゴリズムを提案する．

3.3.1　アルゴリズム

一般に，優先度逆転を回避するための手法として優先度継承プロトコルと優先度上限プロトコルが知られている [22]．一般には優先度上限プロトコルの方が安全

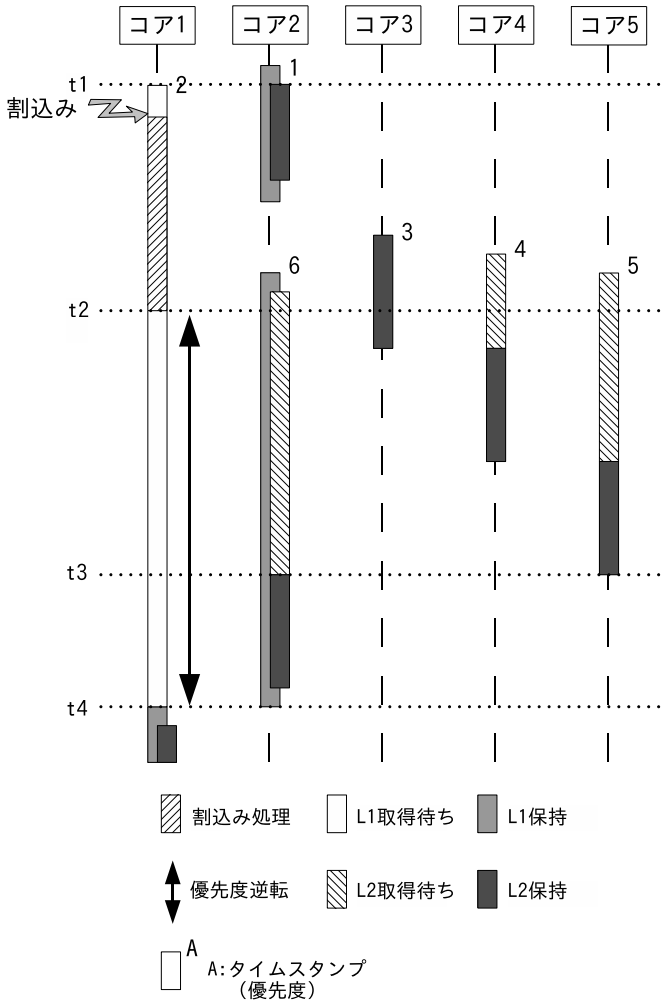


図 3.2: 割込み受付け可能な Totally FIFO アプローチで発生する優先度逆転の例

であるとされているが，次に述べる理由により，優先度継承プロトコルを用いることとした．

優先度上限プロトコルでは，あらかじめ上限となる優先度（上限優先度）を求める必要があるが，PPIQL アルゴリズムでは，タスクに固有の優先度が与えられないため上限優先度が定まらない．これは，ロック取得要求の順番でロックを取得するために，ロック取得要求時のタイムスタンプを優先度として用いるためである．一方，あらかじめ上限優先度を求めずに，特別な値（0 など）を上限優先度として用いる手法も考えられる．この手法では，例えば，タスク T1 がロック L1，L2 の順に，タスク T2 がロック L3，L2 の順にロックを取得し，タスク T3 はロック L2 のみを取得する場合，タスク T3 は通常の優先度で L2 の取得を試行するのに対し，タスク T1 とタスク T2 は上限優先度で L2 を取得することになる．タスク T1 とタ

スク T2 は交互に L2 の取得を繰り返せるため，タスク T3 が L2 を取得する最悪実行時間が定まらない [23]．したがって，この手法では要件 (1) を満たせない．

PPIQL アルゴリズムは，Totally FIFO アプローチを割込み処理の受付け可能に拡張し，さらに優先度継承処理を行うものである．言い換えると，ロック取得待ちの間に割込みが入らない場合は，優先度継承処理を除いて Totally FIFO アプローチと同じである．アルゴリズムを図 3.3 に示す．

1 段目のロックと 2 段目のロックに関わらず，ロック取得待ちの間に割込みが発生した場合は，ロック取得待ちを解除してから割込み処理を実行し，他のコアがロックを取得できるようにする（6，22 行目）．割込み処理から復帰した後はロック取得処理をやり直す．1 段目のロックについては，ロック取得待ち状態であることを解除してロック取得処理をやり直すだけでよい．2 段目のロックの場合は，1 段目のロックを保持しているため，ロック取得状態の解除だけでなく，1 段目のロックも解放する必要がある（23 行目）．これは，2 段目のロック取得待ちの間に割込み処理を実行する場合，1 段目のロックを解放してから割込み処理を実行しないと，他のコアが 1 段目のロックを取得する最悪実行時間が長くなってしまうためである．

1 段目のロックを解放し，ロック取得をやり直す場合は，1 段目のロックにより排他されるクリティカルセクションの実行（13 行目）を複数回行うことになる．複数回実行するかどうかは予測が難しいため，13 行目で実行できる処理は繰り返し実行しても問題ない処理に限られる．そのため，繰り返し実行すると問題のある処理は 2 つのロックを取得してから実行する必要がある．FDMP カーネルでは，1 段目のロックにより排他されるクリティカルセクションは，2 段目のロックの判定処理や RTOS が管理するデータの状態による場合分けなど繰り返し実行しても問題ない処理のみで記述できている．アプリケーションが PPIQL アルゴリズムを利用する際には，1 段目のロックにより排他されるクリティカルセクションが繰り返し実行しても問題ないことが必要であり，そうでない場合は PPIQL アルゴリズムを用いることができない．

2 段目のロック取得については，優先度継承処理が必要となる（18 行目）．優先度継承処理では，1 段目のロックについてロック取得待ちの中の最高優先度を探索する（19 行目）．もし，自コアの設定した優先度より高い優先度のコアが 1 段目のロック取得待ちになっている場合は，その優先度を継承し，2 段目のロック取得の優先度とする（20，21 行目）．2 段目のロックを取得する優先度が高くなるため，優先度逆転の区間はコア数に比例しない．

```
 1:  ロック取得の優先度（タイムスタンプ）を取得する.
 2: acquireL1:
 3:  1 段目のロック取得待ちキューに優先度順にならぶ.
 4:   1 段目のロック取得待ちとなる.
 5:  1 段目のロック取得待ちの間，a. を繰り返す.
 6:    a. 割込みが発生した場合
 7:      1 段目のロック取得待ちを解除し，待ちキューから外す.
 8:      割込み処理を実行する.
 9:      割込み処理から復帰したら acquireL1 に戻る.
10:  1 段目のロックを取得する.
11:   1 段目のロック取得待ちを解除し，待ちキューから外す.
12:
13:  1 段目のロックによるクリティカルセクションを実行する.
14:
15:  2 段目のロック取得待ちキューに優先度順にならぶ.
16:   2 段目のロック取得待ちとなる.
17:  2 段目のロック取得待ちの間，b. と c. を繰り返す.
18:    b. 優先度継承処理
19:      1 段目のロック取得待ちの中から最高の優先度を探索する.
20:      探索した優先度が自コアのロック取得優先度より高い場合,
21:      自コアのロック取得優先度を探索した優先度に設定する.
22:    c. 割込みが発生した場合
23:      1 段目のロックを解放する.
24:      2 段目のロック取得待ちを解除し，待ちキューから外す.
25:      割込み処理を実行する.
26:      割込み処理から復帰したら acquireL1 に戻る.
27:  2 段目のロックを取得する.
28:   2 段目のロック取得待ちを解除し，待ちキューから外す.
29:
30:  1 段目，2 段目のロックによるクリティカルセクションを実行する.
31:
32:  2 段目のロックを解放する.
33:  1 段目のロックを解放する.
```

図 3.3: PPIQL アルゴリズム

3.3.2　PPIQL アルゴリズムの適用例

図 3.2 のシナリオについて，PPIQL アルゴリズムを適用すると図 3.4 のようになる．PPIQL アルゴリズムと割込み受付可能な Totally FIFO アプローチとの違いは，2 段目のロック取得待ち時に優先度継承処理があるかどうかだけであるため，図 3.4 の t2 までは，図 3.2 と同じである．

コア 2 が優先度 6 で L2 の取得を要求する時，既にコア 3 が L2 を保持しているため，コア 2 は L2 を取得できずロック取得待ちとなる．ロック取得待ちとなるので，優先度継承処理を実行する（図 3.3 の 18 行目）．コア 2 は，L1 のロック取得待ちとなっている中から最高の優先度を探す（19 行目）．そして，探索した優先度がコア 2 の優先度 6 より高い場合は，その優先度をコア 2 のロック取得優先度とする（20 行目）この処理は，コア 2 が L2 のロック取得待ちの間に繰り返す（17 行目）．t2 までは L1 の取得待ちはいないので，コア 2 の優先度は 6 であるが，t2 以降は優先度 2 のコア 1 が L1 の取得待ちとなるので，コア 2 は優先度を継承して度 2 を継承し，L2 を取得する優先度を 2 に設定する．その後，t3 でコア 3 がロック L2 を解放すると，コア 4 とコア 5 よりコア 2 の優先度が高いためコア 2 が L2 を取得する．コア 2 が L1 を解放するとコア 1 が L1 を取得する．t4 から t5 までにコア 4 が L2 を取得できるため，t4 から t5 までコア 1 が L2 を取得できないのは避けられない．コア 1 を待たせる低優先度のコアは，コア 3，コア 2，コア 4 の 3 つであり，コア 5 には待たされない．このように，優先度継承を導入することによって，コア 1 が待たされる低優先度のコアを 3 つに抑えることできる．

3.4　PPIQL アルゴリズムのソフトウェア実装

優先度順キューイングスピンロックアルゴリズムに優先度継承を導入したソフトウェアアルゴリズムが既に提案されている [31]．しかし，このアルゴリズムは割込み処理を受付けることができないため，割込み処理を受付けるよう拡張すれば，PPIQL のソフトウェア実装が可能である．割込み処理を受付ける拡張については，シングルロックでコア数に対するスケーラビリティとリアルタイム性を両立した，中断可能なキューイングスピンロックアルゴリズム [32] が参考になる．

PPIQL をソフトウェア実装するためには，CAS 命令や LL/SC 命令が必要であり，TAS 命令では実装できない．LL/SC 命令を用いて PPIQL をソフトウェア実装した結果，2 段目のロック取得についてデータ定義等を除き 170 行程度の規模と

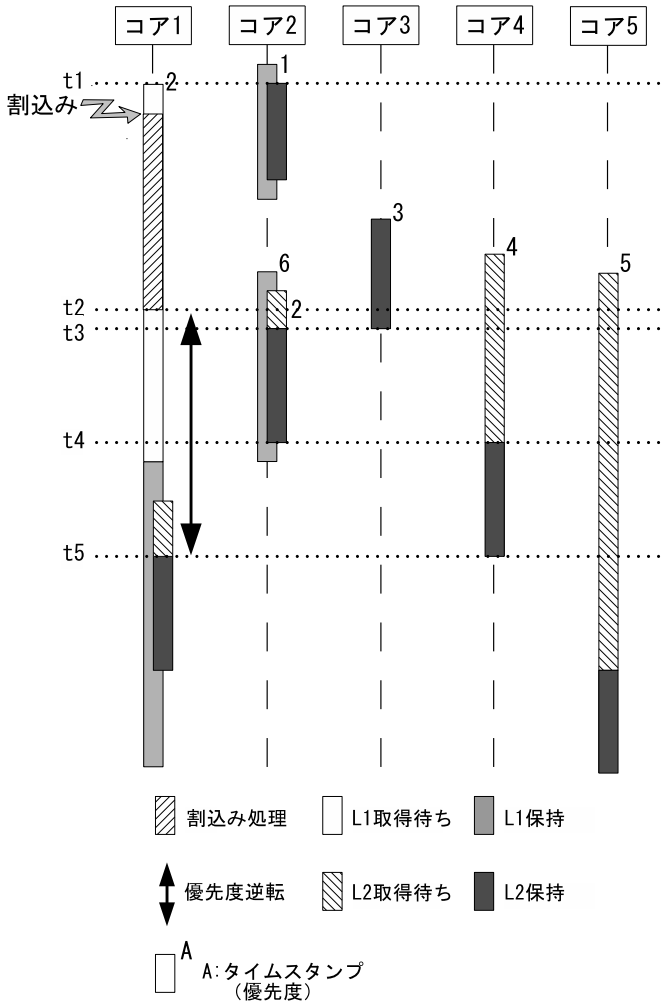


図 3.4: PPIQL アルゴリズムの適用例

なった．文献 [32] で提案されている 2 つのアルゴリズムが 50 行から 80 行程度の規模であるため，ロック取得待ち時間とロック解放処理時間が増加すると考えられる．PPIQL のソフトウェア実装では，コードサイズの増加による処理時間の増加だけでなく，アルゴリズムの複雑化による処理時間の増加も懸念される．PPIQL ではロック取得順は優先度順でなければならず，そのためにはキューのノードを走査する必要がある．また，優先度継承処理でもノードの走査が必要である．これらの処理には，コア数に依存した時間がかかるため，PPIQL アルゴリズムのソフトウェア実装では，ロック取得待ち時間とロック解放処理時間が長くなると予想される．

3.5　PPIQL アルゴリズムのハードウェア実装

本節ではハードウェアを用いた PPIQL アルゴリズムの実現方法を提案する．

まず，ハードウェア実装の利点と欠点について述べた後，PPIQL アルゴリズムを実現するためのハードウェアについて述べ，これを利用したロック取得と解放処理のドライバについて述べる．

3.5.1　ハードウェア実装

組込み向けプロセッサには，CAS 命令や LL/SC 命令を持たず，TAS 命令のみを持つものも少なからず存在している．このようなプロセッサで PPIQL を用いるためには，CAS 命令や LL/SC 命令を用意するか，提案ハードウェアを用いる必要がある．ここで，CAS 命令や LL/SC 命令を実現するためにはバスを複雑化する必要があるため，提案ハードウェアを用いることに有用性があると考えられる．また，専用ハードウェアにより排他制御を実現しているプロセッサにおいても，専用ハードウェアは TAS 命令相当の機能であるため（TAS ハードウェアと呼ぶ），PPIQL のソフトウェア実装を実現することができない．TAS ハードウェアを用意しているプロセッサにおいては，PPIQL のハードウェア実装を用いることで，より優れた排他制御手法を用いることができる．

TAS ハードウェアを用いている例として，SH7265（ルネサスエレクトロニクス社）では 32 個持つ．また，MC9S12XD（フリースケール社）ではメインの S12X とコプロセッサの XGATE 間のために 8 個，OMAP4430（テキサスインスツルメンツ社）では Cortex-A9，Cortex-M3，DSP 間のために 32 個の TAS ハードウェアを持つ．このように，ヘテロジニアスなマルチコアシステムでも TAS ハードウェアを用いていることが多い．

PPIQL をハードウェア実装し，アプリケーションから提案ハードウェアを直接用いたい場合には，アプリケーションからハードウェアをアクセス可能とするため，カーネル空間で実行する必要がある．一般的に RTOS は保護機能を持たず，アプリケーションをカーネル空間で実行するため，問題とならない．

システムで必要となるロックの数によって，必要となるハードウェアの数も変化する．RTOS 内部で用いる場合には，RTOS 設計時に必要なロックの個数が決まるため，この数を用意すればよい．前述のように，FDMP カーネルでは，コア数 ×2 個用意すればよい．

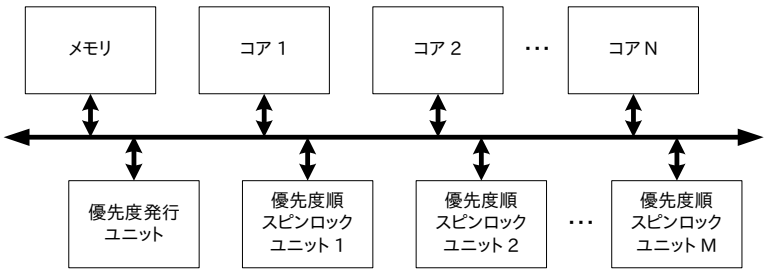


図 3.5: 提案ハードウェアを利用したシステムの例

一方，アプリケーションが提案ハードウェアを用いる場合には，アプリケーションによって必要となるロック数が変化するため，ハードウェアが提供すべきロックの数を予測しにくい．そのため，アプリケーションが提案ハードウェアを利用するシステムを構築する際には，十分な数のハードウェアを用意しておく必要がある．

3.5.2　ハードウェアの概要

提案ハードウェアは，優先度発行ユニットと優先度順スピンロックユニットで構成される．優先度発行ユニットはシステムに 1 つ必要で，優先度順スピンロックユニットはロック個数と同数だけ必要である．図 3.5 に提案ハードウェアを用いてロックを実現するシステムの例を示す．2 種類のユニットを用いてロックを取得する手順は次のようになる．ロックを取得するコアは，優先度発行ユニットから優先度を取得し，優先度順スピンロックユニットに設定する．この優先度はロックを取得する順序を決めるもので，優先度順スピンロックユニットは設定された優先度の中で最も優先度の高いコアにロックを与える．

優先度

優先度は，優先度発行ユニットと優先度順スピンロックユニットの両方で利用する値である．優先度発行ユニットから読み出した優先度を優先度順スピンロックユニットに設定する．優先度には，有効値と無効値があり，優先度発行ユニットからは有効値のみを読み出す．優先度順スピンロックユニットには，有効値もしくは無効値を設定することができる．無効値が設定されているコアはロック取得を要求していないことを表す．有効値は，値の小さいものほど高い優先度を持つことを表す．

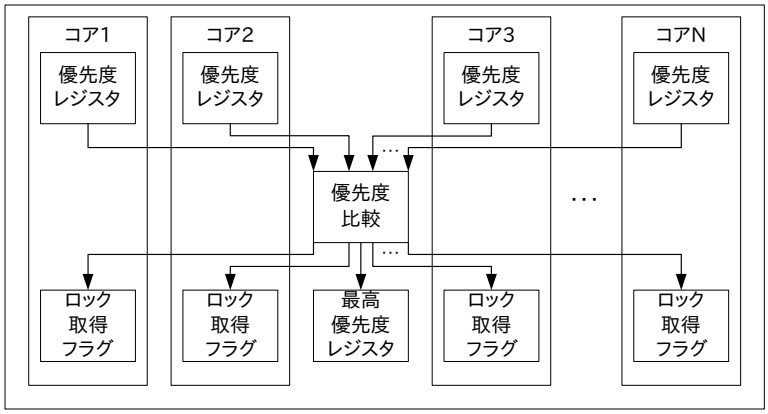


図 3.6: 優先度順スピンロックユニット

優先度発行ユニット

優先度発行ユニットは，ロックを取得しようとしているコアに対し，ロック取得の優先度を与える．優先度はユニット内部のカウンタにより保持されており，値を読み出すたびにインクリメントされる．値が小さいほど高い優先度のため，値を先に読み出したコアが高い優先度を持つ．優先度発行ユニットにより，FIFO 順にロックを取得するという原則が与えられる．

複数のコアが同時に優先度の読出しを行った際は，バスによる調停が行われる．読出しは単一命令で実現できることが多いため，この場合は複数のコアはそれぞれ異なる値を読み出すことが保証される．バスの調停方式は複数あるが，他のコアにバスを占有され，バスを使用できないコアが発生する可能性のある方式では，要件 (1) を満たせないため，ラウンドロビンなどの方式を用いる必要がある．

優先度順スピンロックユニット

優先度順スピンロックユニットは，ロック取得可能なコアを決定するユニットである．内部には，最高優先度レジスタと，コアごとに優先度レジスタとロック取得フラグの組を持つ．優先度順スピンロックユニットを図 3.6 に示す．

優先度順スピンロックユニットの状態遷移を図 3.7 に示す．状態はアンロック状態とロック状態の 2 状態を持ち，リセット時にはアンロック状態となる．アンロック状態の時は，有効値を保持している優先度レジスタの値を比較し，最も高い優先度を持つ優先度レジスタに対応するロック取得フラグをセットし，ロック状態に遷移する．全ての優先度レジスタの値が無効値である場合はアンロック状態のま

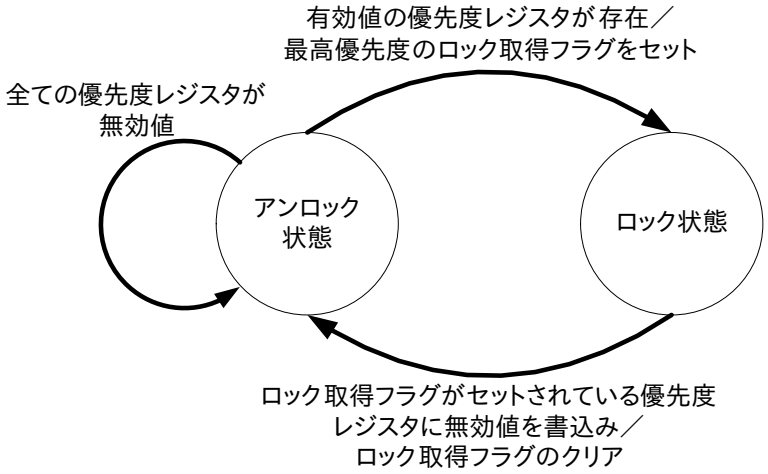


図 3.7: 優先度順スピンロックユニットの状態遷移図

ま遷移しない．ロック状態の時は，セットされているロック取得フラグに対応する優先度レジスタに無効値が書込まれると，ロック取得フラグをクリアして，アンロック状態に遷移する．最高優先度レジスタには，状態に関係なく，有効値の中で最高の優先度が設定される．

優先度の比較について，優先度がオーバフローすると単純な大小比較ではうまくいかない．そのため，比較には ICTOH[7] と同じアルゴリズムを用いている．

ロックを取得したいコアは，まず，優先度の有効値を優先度レジスタに設定する．ロック取得フラグをチェックし，セットされていれば，ロック取得に成功する．ロック取得フラグがセットされていないときは，ロック取得に成功するまでロック取得フラグをチェックする．クリティカルセクション実行後は，優先度レジスタに無効値を書込むことでロック解放を行う．

優先度比較器は，1 つの比較器を用いてコア数分の優先度を比較するのではなく，複数の比較器を組み合わせて実現している．コア数が多くなったときは，比較器の段数が深くなるため，結果を得るためのサイクル数が増えるか，動作クロックを遅くする必要がある．そのため，コア数が多くなるとスピンロック取得待ち時間が長くなる．

3.5.3　ドライバ

以下に述べるドライバの処理は，アプリケーションが直接利用することも可能であり，カーネル空間で実行されることを想定している．

ロック取得・解放

提案ハードウェアを利用したロック取得・解放のルーチンを図 3.8 に示す．図 3.8 では，全ての処理をまとめて記述している．19，20 行目は 1 段目のロックによるクリティカルセクション（CS_1 とする）で，35 行目が 1 段目と 2 段目のロックによるクリティカルセクション（CS_12 とする）である．coreid はコアの識別子を，lockid1 と lockid2 は優先度順スピンロックユニットの識別子をそれぞれ表す．lockid1 は 1 段目のロックに対応し，lockid2 は 2 段目のロックに対応する．pri 配列は各コアがロックを取得する優先度を保持する．優先度には無効値（INVALID，例えば 0）と有効値があり，無効値はそのコアがロック取得に関する処理を実行していないことを表す．spinning 配列は各コアが割込み処理を実行したかどうかを保持する．

ロック取得を要求したら，まず，優先度を取得する．pri 配列には無効値で初期化されているため，優先度発行ユニットから get_priority() 関数により優先度を読み出す（6 行目）．読み出した優先度は enter_spinlock() 関数により優先度レジスタに設定する（10 行目）．優先度順スピンロックユニットは，有効値の優先度を比較し，最高優先度のコアに対応するロック取得フラグをセットする．ロックが取得できたかどうかは，ロック取得フラグをチェックする try_spinlock() 関数で行う（11 行目）．try_spinlock() 関数はロック取得できない場合に真が返ってくる．偽が返ってきてロックが取得できていれば，クリティカルセクションの実行に移る．ロックを取得できなく，かつ，割込みが発生した場合，割込み処理を行うことで割込み応答性を向上させる．割込み応答性に関しては後述する．

2 段目のロックについても同様の処理を行う．ただし，Totally FIFO アプローチと同様に 1 段目の優先度で 2 段目のロックを取得する．また，優先度継承するため，2 段目のロック取得待ち時に 1 段目のロックの最高優先度レジスタを get_high_pri_on_spinlock() 関数により読み出す（30 行目）．読み出した優先度がそのコアが設定した優先度より高い場合は，優先度を継承し 2 段目のロック取得を行う．

ロックの解放は，release_spinlock() 関数で優先度順スピンロックユニットに対し無効値を設定し（37，38 行目），各配列に無効値を設定する（39，40 行目）．

ロック取得待ちの中断

ロック取得待ち中の割込み応答性を向上させるために，ロックが取得できなかった場合は割込み処理を行う．具体的には，try_spinlock() 関数でロック取得できていないことがわかった場合，一旦，割込みを許可する（12，24 行目）．割込みが発生していた場合はここで割込み処理にジャンプする．割込み処理では，図 3.9 に示す処理を実行した後，実際の割込み処理を行う．図 3.9 の処理では，suspend_spinlock() 関数により優先度レジスタに無効値が書込まれるため，優先度順スピンロックユニットによる優先度比較の結果，割込み処理を実行するコアにロックが渡されることがない．割込み処理の実行後は，get_priority() 関数を呼ばずに，割込み処理前と同じ優先度を優先度レジスタに設定する．

3.6　評価

3.6.1　評価項目

実機を用いた実験により，提案するアルゴリズムの性能評価を行う．比較には，MCS ロックと優先度順キューイングスピンロックアルゴリズムである Markatos ロック [14] を用いた Totally FIFO アプローチ (TF)，提案ハードウェアを用いたシングルロックを単純にネストロックにしたもの (simple)，割込み受付け可能な Totally FIFO アプローチ (TF/P)，PPIQL をソフトウェア実装したもの (PPIQL(SW)) を用いる．simple の基本となるシングルロックは中断可能なキューイングスピンロックであるため，割込み処理を受付けることができる．TF/P は，PPIQL の優先度継承の処理がないものである．RTOS 内部で用いるプリミティブなロックについて，評価対象としたアルゴリズム以外に割込み応答とコア数のスケーラビリティについて言及している手法は，私の知る限り存在しない．

それぞれのアルゴリズムについて，3 つの要件を満たすかという観点で評価を行う．要件 (3) については，優先度逆転が発生するケースについても評価を行う．また，提案ハードウェアのサイズの比較評価を行う．

```
 1:  /* 割込み禁止 */
 2:  disable_interrupt();
 3: retry:
 4:  /* 初回のみ優先度を取得する */
 5:  if( pri[coreid] == INVALID ) {
 6:    pri[coreid] = get_priority();
 7:  }
 8:  spinning[coreid] = true;
 9:  /* 1 段目のロック取得 */
10:  enter_spinlock(lockid1, coreid, pri[coreid]);
11:  while( try_spinlock(lockid1, coreid) ) {
12:    enable_interrupt();
13:    /* ここで割込み処理に移る */
14:    disable_interrupt();
15:    if (spinning[coreid] == false) {
16:     goto retry;
17:    }
18:  }
19:  /* ここから 1 段目のロックによるクリティカルセクション
20:      ここの処理は，割込みにより繰り返し実行される場合がある */
21:  /* 2 段目のロック取得 */
22:  enter_spinlock(lockid2, coreid, pri[coreid]);
23:  while( try_spinlock(lockid2, coreid) ) {
24:    enable_interrupt();
25:    /* ここで割込み処理に移る */
26:    disable_interrupt();
27:    if (spinning[coreid] == false) {
28:     goto retry;
29:    }
30:    high_pri = get_high_pri_on_spinlock(lockid1);
31:    if( high_pri < pri[coreid] ){
32:      enter_spinlock(lockid2, coreid, high_pri);
33:    }
34:  }
35:  /* 1 段目，2 段目のロックによるクリティカルセクション */
36:  /* ロック解放 */
37:  release_spinlock( lockid2, coreid );
38:  release_spinlock( lockid1, coreid );
49:  spinning[coreid] = false;
40:  pri[coreid] = INVALID;
41:  enable_interrupt();
```

図 3.8: ロック取得・解放ルーチン

```
1:  /* 無効値を設定する */
2:  suspend_spinlock( lockid2, coreid );
3:  suspend_spinlock( lockid1, coreid );
4:  spinning[coreid] = false;
```

図 3.9: 割込み処理入口ルーチン

3.6.2　評価環境

提案ハードウェアを VHDL で記述し，実装デバイスには StratixⅡ EP2S180 を用い，Altera 社の QuartusⅡ version 8.1 を用いて合成を行った．優先度は 16 ビットとした．コアには周波数 50MHz の NiosⅡ プロセッサを用いた．NiosⅡ プロセッサの命令キャッシュは 4KB で，データキャッシュはない．この構成の NiosⅡ プロセッサを 8 個合成したもので評価を行う．優先度発行ユニットを接続するバスの調停はラウンドロビンで行われ，優先度比較は 8 コアでも 1 サイクルで終了することを確認した．また，PPIQL(SW) の実装のため，LL/SC 命令を NiosⅡ にユーザ命令として追加している．

3.6.3　評価方法

ロック取得要求から解放までの処理時間がコア 1 で長くなるように，コア 1 とその他のコアで処理を変え，コア 1 の計測結果により各アルゴリズムの比較評価を行う．コア 1 はロック L1，L2 を取得するネストロックルーチンを実行する．その他のコアは，コア 1 と同じネストロックルーチンをコア 1 より先行して実行した後，ロック L2 を取得するシングルロックルーチンを 8 回繰り返す．以上の処理を 1 つの単位とし，1000000 回繰り返し実行する．このようにすると，コア 1 が 1 段目のロックを取得要求したとき，他のコアが先行して 1 段目のロックを取得する．さらに，1 段目のロックが解放された後も，コア 1 が 2 段目のロックを取得要求したとき，他のコアが 2 段目のロック取得を繰り返しているため，競合しやすい．クリティカルセクションは，ネストロックについて CS_1 と CS_12 の 2 つ，L2 によるシングルロックによるもの（CS_2 とする）で，計 3 つがある．3 つのクリティカルセクションの処理内容は全て空ループである．CS_1 と CS_12 は，実行時間が約 $18\mu s$ であり，CS_2 の実行時間は約 $34\mu s$ である．割込みはタイマによって $10ms$ ごとに発生し，割込み処理の実行時間は約 $19\mu s$ である．

アルゴリズムをハードリアルタイムシステムに適用するためには，最悪実行時

間を用いて評価する必要がある．マルチコアシステムにおいては，あるコアにおける処理の実行時間は，他のコアとの実行順序に依存して決まる．スピンロックアルゴリズムについては，システム中の全てのコアが同一のスピンロックの取得をほぼ同時に要求したときに真の最悪実行時間となる．そのような実行順序が発生する可能性は低く，また，そのような実行順序を狙って実行することも実機では困難である．したがって，マルチコアシステムで真の最悪実行時間を計測するのは困難である．

真の最悪実行時間を計測できないため，有限の計測回数によって得られる実行時間のうち，最大値だけで各アルゴリズムを比較することは公平でない．これは，偶然にも実行時間の長いケースを一度だけ計測した場合に，偶然計測したケースでアルゴリズムの比較をしてしまうからである．そこで本研究では，1000000 回繰り返し実行した中で，その時間までに実行を終えた割合が 99.999% 以上となる時間を用いて性能を評価する．言換えると，評価に用いる時間をデッドラインとした場合に，デッドラインをミスする割合は 0.001% である．評価に用いる時間より長い実行時間となった計測回数（デッドラインをミスした回数）は，最大 10 回ほど存在するが，これらの結果は評価に用いない．

3.6.4　評価結果

ロック取得待ち時間とロック解放処理時間

要件 (1) について，ネストロック全体の取得要求から解放までに要した時間を計測することにより評価する．そのため，計測結果には CS_1 と CS_12 の実行時間が含まれる．1 段目，2 段目のロックについて，ロック取得待ち時間とロック解放処理時間を計測するのが理想であるが，計測処理は状況によって実行時間が変化するため，全体の実行時間を計測した．

ロック取得要求から解放までの処理時間について，コア数を 1 から 8 まで変えた結果を図 3.10 に示す．図 3.10 はロック取得待ち中に割込みが発生しなかった場合である．図 3.10 で CS としているのは，ロック取得待ちやロック解放処理にかかる時間，バス衝突の影響などを無いとしたときの最悪実行時間である．例えば，4 コアの場合，他の 3 つのコアが先行してロックを取得する場合が最悪であり，自コアのクリティカルセクション実行時間を含めた $4 \times 2 \times 18\mu s = 144\mu s$ が最悪実行時間となる．

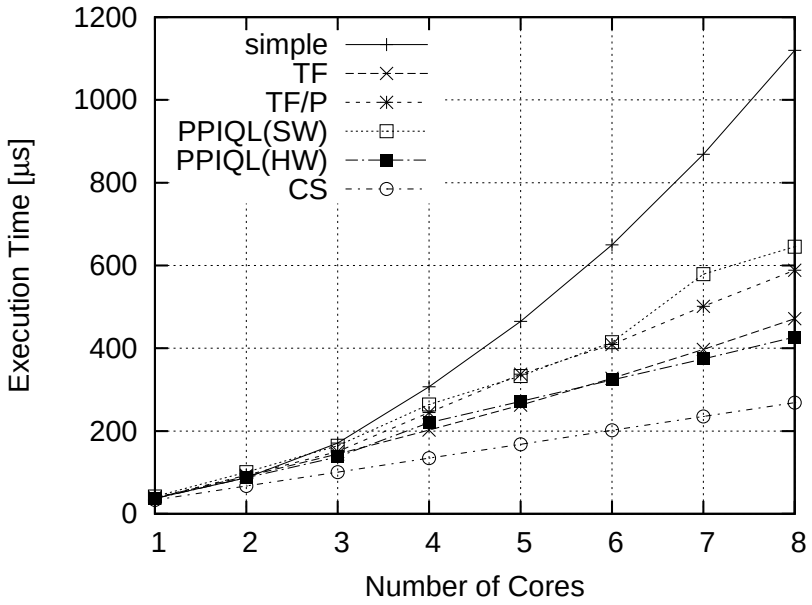


図 3.10: ロック取得要求から解放までの実行時間（割込みなし）

simple は $O(N^2)$ になってしまうため，コア数が増えたときの処理時間が他のアルゴリズムに比べて大きい．TF は処理時間が短いが，割込みを受付けることができない．TF/P と PPIQL(SW) は，TF を拡張したため，TF より処理時間が増えている．CS との差分で比較すると，PPIQL(HW) は PPIQL(SW) の 52.9% の実行時間となり，1.89 倍高速化した．

simple 以外のアルゴリズムは，コア数のリニアオーダになるはずだが，ややリニアオーダより悪くなっている．これは，コア数の増加によりデータアクセスについてバス衝突の可能性が高くなっているためと考えられる．バス衝突はハードウェア構成により発生するものであるため，スピンロックアルゴリズムとしては要件 (1) を満たしているとする．

割込み応答時間

要件 (2) については，ネストロック全体の取得要求から解放までの間に発生した割込みに対する応答時間により評価する．具体的には，3.6.4 節と同様に各コアがロック取得要求とロック解放を繰り返し，その間に発生した割込み応答時間を計測する．

割込みに対する応答時間について，コア数を 1 から 8 まで変えた結果を図 3.11

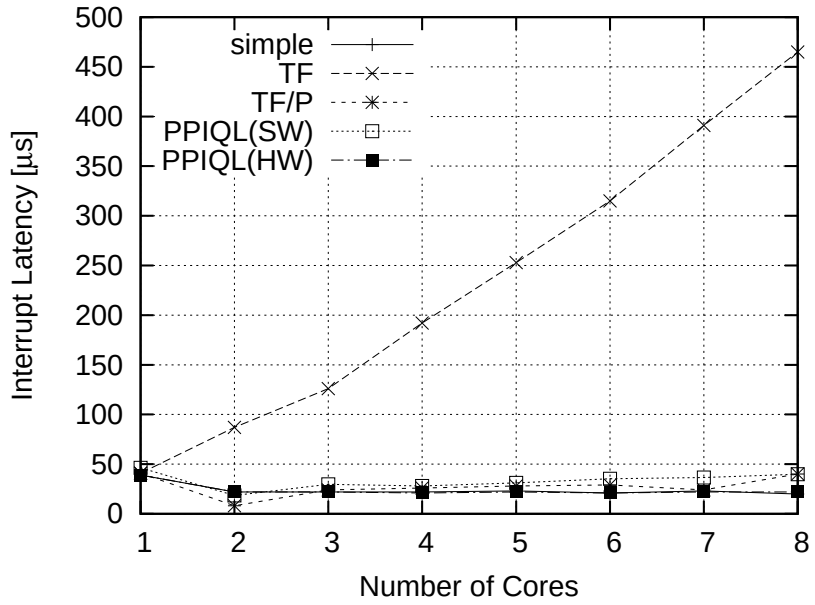


図 3.11: 割込み応答時間

に示す．TF はロック取得待ち中に割込み処理を実行することができないため，コア数の増加に応じてロック取得待ち時間とロック解放処理時間が長くなるほど割込み応答時間が悪くなる．クリティカルセクションを実行している $18\mu s$ の間以外は割込み処理を実行可能なため，TF 以外のアルゴリズムは高速に割込みを受付けていることがわかる．また，割込み応答時間がコア数に依存しておらず，要件 (2) を満たしている．

割込み処理によるペナルティ時間

要件 (3) については，ロック取得待ち中に割込みが 1 回だけ発生した場合のネストロック全体の取得要求から解放までに要した時間を計測し，割込みが入らなかった場合（図 3.10）と比較し評価する．

ロック取得待ち中に割込みが 1 回だけ発生した場合の，ロック取得要求から解放までの処理時間について，コア数を 2 から 8 まで変えた結果を図 3.12 に示す．割込み処理を実行した分だけ図 3.10 より実行時間が長くなっているが，傾向は図 3.10 と同じである．ただし，TF は割込みを受付けられないため図 3.12 にはない．クリティカルセクションの実行時間と割込み処理の実行時間を加えた結果を CS+で表しており，4 コアでは $144 + 19 = 163\mu s$ となる．

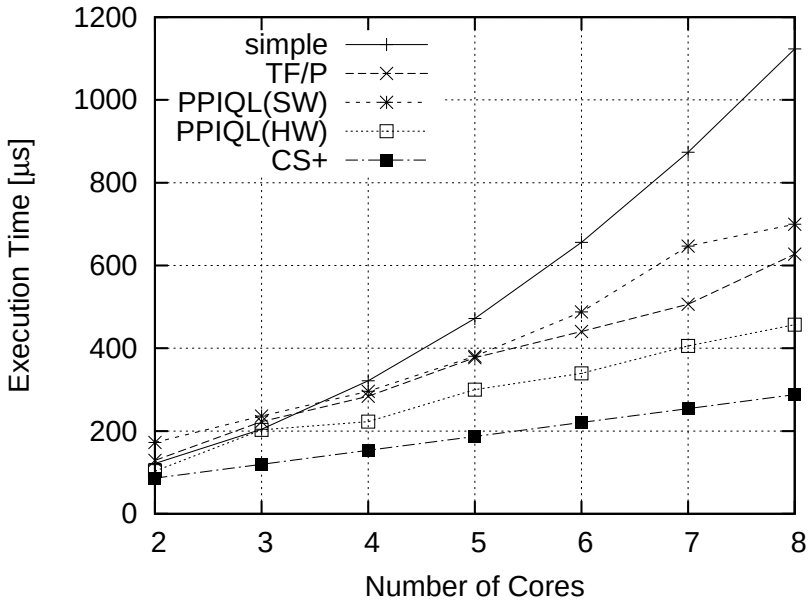


図 3.12: ロック取得要求から解放までの実行時間（割込みあり）

図 3.10 と図 3.12 の差分は図 3.13 のようになる．この差分は，割込み処理実行時間と割込み処理によるペナルティ時間である．コア数と実行時間の変動について規則性が見受けられないため，全てのアルゴリズムでコア数に依存していないといえる．したがって，この計測結果から TF 以外のアルゴリズムが要件 (3) を満たしていると考えられる．なお，多くのアルゴリズムで 3 コアのときの時間が長くなっているが，これは，前述のように今回の測定が 99.999% 以上となった時間を比較に用いているため，真の最悪値となっておらず，偶然に 3 コアの結果が他のコア数と比較して悪くなったためと考えられる．

優先度継承

図 3.10 では，TF/P と PPIQL(SW) に大きな違いがない．これは測定結果に優先度逆転が発生したケースが少ないためだと考えられる．優先度継承の効果を測定するため，次のように優先度逆転が発生しやすい状況にした．

- 割込みが発生する間隔を $10ms$ から $1ms$ に変更
- コア 2 は，L1 と L2 のネストロックルーチン（コア 1 と同じルーチン）を 8 回繰り返す

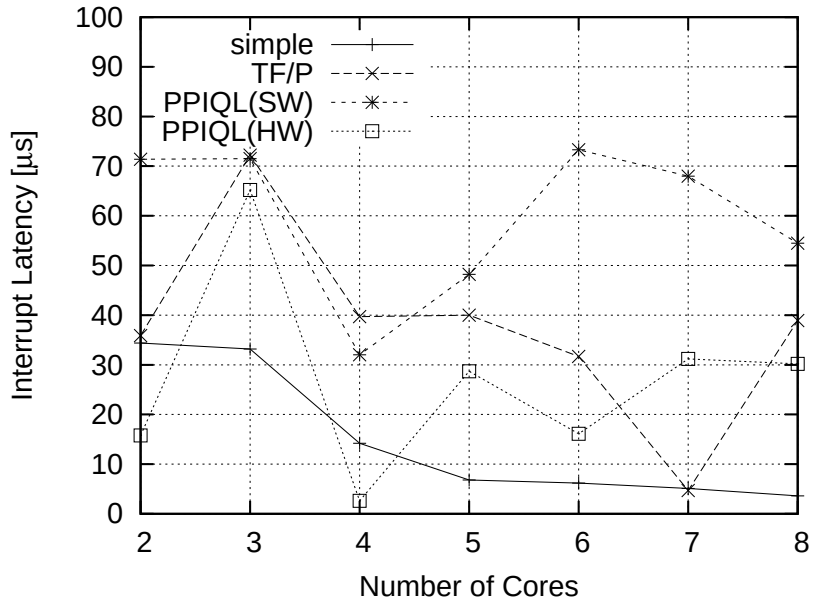


図 3.13: 割込み処理実行時間と割込み処理によるペナルティ時間

- コア 1 とコア 2 以外のコアは，L1 と L2 のネストロックルーチンを 1 回実行した後，L2 のシングルロックルーチンを 16 回繰り返す

このような状況でコア数を 2 から 8 まで変えたとき，ロック取得待ち中に割込みが発生した場合のロック取得要求から解放までの処理時間を図 3.14 に示す．前述したように，優先度継承の効果が現れるのはコア 1 以外に 3 つ以上（計 4 つ以上）のコアが存在するときである．優先度逆転の最大時間は，コア数が増えると長くなるため，コア数が増えると優先度継承の効果が大きくなる．5 コア以上の場合について，CS との差分で比較すると，PPIQL(HW) は PPIQL(SW) の 32.2% の実行時間となり，3.11 倍高速化した．

優先度逆転は，その発生する可能性が非常に低いと考えられるが，優先度逆転が発生する可能性のある TF/P は要件 (3) を満たすことができない．ネストロックで要件 (3) を満たすには PPIQL アルゴリズムを利用する必要がある．

ハードウェアサイズ

優先度順スピンロックユニットを 4 コア，8 コアそれぞれについてサイズを計測した．優先度順スピンロックユニットは，コアの数だけ優先度レジスタとロック取得フラグが必要である．16 ビットの優先度レジスタについて，4 コアと 8 コアの実

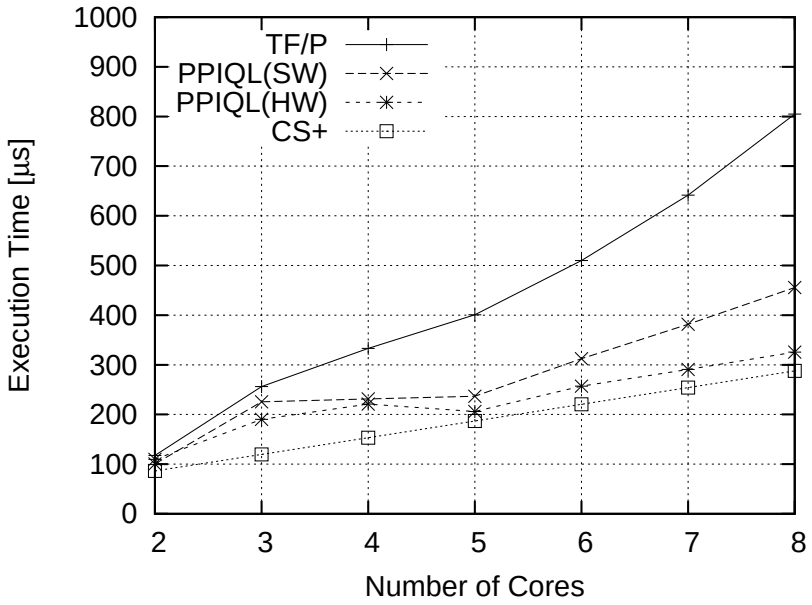


図 3.14: 優先度継承の効果

装を行った結果，表 3.1 のようになった．比較のために，Mutex 回路，1 チャネルの 32 ビットタイマ，4 コアと 8 コアのシステム全体のハードウェアサイズを計測した．優先度比較はコア間でそれぞれ行っているため，コア間の組み合わせが増える．具体的には 4 コア時には $6(= {}_4C_2)$ 個だった比較器が，8 コア時には $28(= {}_8C_2)$ 個になった．提案ハードウェアは，優先度順スピンロックユニットと優先度発行ユニットの組み合わせでシステム全体に占める割合が最大 2% 程度であり，許容できると考えられる．

3.6.5　考察

以上の評価より，各アルゴリズムの要件の充足について表 3.2 に示す．simple はロック取得待ち時間が $O(N^2)$ になってしまうため要件 (1) を満たさないが，要件 (2) と要件 (3) を満たす．TF は要件 (1) を満たすが，割込みを受付けることができないため，要件 (2) を満たさず，要件 (3) について評価ができない．TF/P は，TF をベースに割込み受付け可能な拡張をしたことで，要件 (1) に加えて要件 (2) も満たすが，優先度逆転により要件 (3) を満たさない．提案アルゴリズムである，PPIQL(SW) と PPIQL(HW) は，TF/P に優先度継承処理を加えたことで要件 (3) を含む全ての要件を満たす．

表 3.1: ハードウェアのサイズ

		ALUT数	Logic Register 数
Mutex		47	33
タイマ		144	120
優先度発行ユニット		34	16
優先度順スピンロックユニット	4 コア	201	85
	8 コア	754	182
システム全体	4 コア	18297	9805
	8 コア	38129	19285

表 3.2: 要件の充足

	simple	TF	TF/P	PPIQL(SW)	PPIQL(HW)
要件 (1)	×	○	○	○	○
要件 (2)	○	×	○	○	○
要件 (3)	○	–	×	○	○

PPIQL(SW) と PPIQL(HW) を比較すると，PPIQL(HW) では最大 3.11 倍程度の性能向上を得られるが，ハードウェアコストが増加するというデメリットもある．PPIQL を実装するためには CAS 命令や LL/SC 命令が必要なため，これらの命令が用意されていないシステムでは，ハードウェア実装が有効であると考えられる．一方，これらの命令が用意されているシステムにおいて，ソフトウェア実装による性能が要求を満たしている場合では，ソフトウェア実装を用いた方がハードウェアコストを抑えられる．このように，PPIQL の実装方法を選択する際には，ハードウェアコストと性能向上のトレードオフを考慮する必要がある．

3.7　関連研究

リアルタイム性を考慮した排他制御に関する研究は，2 種類に分けることができる．1 つは RTOS 内部で用いるプリミティブなロック（RTOS 向けロック）であり，もう 1 つはプリミティブなロックを用いて実現するロック（アプリケーション向けロック）である．アプリケーション向けロックは，RTOS の機能としてアプリケーションに提供されるのが一般的である．

RTOS 向けロックのソフトウェアアルゴリズムとしては，中断可能なキューイングスピンロックアルゴリズム [32]，Totally FIFO アプローチ [23]，優先度継承スピンロックアルゴリズム [31] などが提案されている．これらのアルゴリズムは，前述したように 3 つの要件を満たすことができない．RTOS の内部のクリティカルセクションは一般に短く上限時間が定まっているため，RTOS 向けロックアルゴリズムはスピンによりロックの取得を待つ．

RTOS 向けロックのハードウェアサポートによる実現方法が， Saglam らによって提案されている [19]．このアルゴリズムは，ロックを取得できなかった時にロック状況を繰返しチェックするのではなく，ロックを解放するコアが次にロックを取得するコアに対して割込みを発生させる機構をハードウェアで実現することにより，ロック取得待ちのバスアクセスを低減している．ロック取得の順序は FIFO を選択可能なため，ロック取得までの時間に上限があるが，割込み応答性については考慮していないため，要件（2）を満たさない．

アプリケーション向けロックのソフトウェアアルゴリズムとしては，優先度継承プロトコルをマルチコアシステム向けに拡張した MPCP[18][13] や FMLP 等 [6] の手法が提案されている．これらの手法は，アプリケーションの比較的長いクリティカルセクションを実現するために用いられる．そのため，ロックの取得をスピンで待つのではなく，タスクの状態を待ち状態として待つことで，プロセッサの利用効率を向上させている．タスクの状態を待ち状態とするには，タスクの管理データを操作する必要があるため，RTOS 内部のクリティカルセクションを実現する必要がある．また，FMLP は，クリティカルセクションが短い場合にはスピンで，長い場合はタスクの待ち状態でロックの取得を待つ．しかしながら，スピン時のスピンロックの手法については，OS が提供するスピンロックを使用することを前提している．すなわち，これらのロックを実現するには，RTOS 向けのロックが内部で必要となり，本研究で対象としている RTOS 向けロックとして用いることは難しいと考えられる．これら関連研究との詳細な比較は今後の課題とする．

3.8　まとめ

本章では，マルチコアシステムで必須となるコア間排他制御を実現するスピンロックを，リアルタイムシステムで用いる際に求められる要件を満たす，中断可能な優先度継承キューイングスピンロックアルゴリズムを提案した．さらに，提案アルゴリズムの一部をハードウェア化することにより，最大 3.11 倍高速化すること

を示した．スピンロックというプリミティブな排他制御を 3 倍高速化できたことは，高い応答性が求められるシステムに対する有用性が高いと考えられる．提案ハードウェアは CAS 命令や LL/SC 命令などを用いずにスピンロックを実現できるため，様々なマルチコアシステムで利用できる．

今後の課題として，3 つ以上のネストロックが挙げられる．FDMP カーネルでは，2 つのネストロックで良い．しかしながら，FDMP カーネルを発展させた，FMP カーネルでは，タスクマイグレーションをサポートした関係で，一部の機能の実現のため，3 つのロックを同時に取得する必要がある．そのため，今後は 3 つ以上のネストロックについても提案手法の拡張により対応できるかを検討する必要がある．

第4章　マルチコアシステム向けリアルタイムOSのテスト効率化手法

4.1　概要

近年，組込みシステムのソフトウェアを原因とする不具合が問題視されている中で，ソフトウェアの品質確保が重要な課題となっている．特に，RTOS はソフトウェアの中核をなすため，高い品質が求められる．そのため，ASP カーネル及び FMP カーネルの品質を高める取組みとして，テストスイートの開発が行われている [30]．テストスイートは，ソースコードの条件網羅[1]によるソースコードカバレッジ（以下，カバレッジ）を 100%とすることを 1 つの目標としている．ASP カーネルに対しては，ホワイトボックステストを行うテストプログラムを作成して実行することにより，カバレッジを 100%にすることができた [29]．

一方，FMP カーネルにおいては，各コアの実行順序に依存してパスが定まる分岐（以下，実行順序依存分岐）が存在するため，ASP カーネルと同様の手法では，カバレッジを 100%にすることができなかった．FMP カーネルではコア数に対する性能やリアルタイム性の確保のために，他の RTOS と比較しても多くの実行順序依存分岐が存在する．

実行順序依存分岐のすべてのパスを網羅する（以下，分岐網羅）手法としては，実行順序依存分岐を含む処理と，その分岐に影響を及ぼす可能性がある処理を繰り返し実行する手法が考えられる．分岐網羅をこの手法により達成するには，実行順序が十分に変動することを仮定している．例えば，テストプログラムによって各コアの実行順序を変動可能であれば，十分な数の試行を繰り返すことによって網羅性が期待できる．この手法では，テストの実施や不具合の分析に多くの手間を要するため，テストの効率が悪い．具体的には，実行順序依存分岐の特定のパス

[1]条件式の判定による真偽を少なくとも 1 回は実行する

を実行するコアの実行順序の発生する可能性が低い場合，分岐網羅となるまでに多くの繰り返し回数が必要となり，テストの実施に長い時間を要する．また，分岐網羅しなかった場合の原因の分析方法として，実行ログを解析する方法があるが，繰り返し実行するため大量の実行ログを解析しなければならない．

これらの問題は，発生する可能性の低いコアの実行順序を決定的に実現可能な手法があれば解決できる．テスト実施の時間の問題に対しては，実行順序依存分岐の各パス毎に 1 回の実行で分岐網羅可能である．不具合分析の手間に対しては，1 回分の実行ログを解析すればよいため，分析対象の手間が削減できる．

これまで，マルチコアにおいて各コアの実行順序を制御するデバッグハードウェアが提案されている [15]．しかしながら，この手法は任意のタイミングで割込みを発生させることができない．FMP カーネルには，各コアの実行順序に加えて割込みの発生順序にも依存する実行順序依存分岐があり，この分岐を網羅できない．

このような問題に対応するため，FMP カーネルにおいて効率的に実行順序依存分岐の分岐網羅を行う手法を提案する．提案手法では，各コアの実行順序や割込みの発生順序をテストプログラムにより制御する実行制御機構を用いる．分岐網羅するために，まず，実行順序依存分岐のパスを実行する，コアの実行順序や割込み発生の実行順序を調べる．次に，実行制御機構を用いて，その順序を実現するテストプログラムを作成する．そして，テストプログラムと実行制御機構を実装したシミュレータやハードウェアを用いて実行することにより，実行順序依存分岐のパスを決定的に実行する．本研究では，実行制御機構を実装した命令セットシミュレータ（以下，シミュレータ）を開発した．テストプログラムとシミュレータが協調して動作することで，FMP カーネルに存在する 83 件の実行順序依存分岐を効率的に分岐網羅できることを確認した．また，1 件の不具合を検出した．

4.2　FMP カーネルの分岐とパス

本章では，FMP カーネルのセマフォを取得する API である `wai_sem` を例に，まずコアの実行順序に依存しない分岐（状態依存分岐）とパス（状態依存パス）について説明する．次に，コアの実行順序や割込みの発生順序に依存する分岐（実行順序依存分岐）とパス（実行順序依存パス）について説明する．そして，実行順序依存パスを実行する手法について検討する．

4.2.1　状態依存分岐

状態依存分岐の例

説明のため簡略化した wai_sem のソースコードを図 4.1 に示す．wai_sem では，まず取得するセマフォを管理するデータ構造（SEMCB 型）を取得する（3 行目）．次に，セマフォ資源数（p_semcb->semcnt）をチェックし（7 行目），1 以上あれば資源を獲得し（8–10 行目），0 の場合は wai_sem を呼び出したタスクを待ち状態とする（13–22 行目）．前者のパスをセマフォ取得パス，後者のパスをセマフォ取得待ちパスと呼ぶ．

セマフォ取得パスは，資源数が 1 のセマフォ1 個とタスクを 1 個用意し，タスクからセマフォに対して wai_sem を呼び出すテストプログラムにより実行可能である．セマフォ取得待ちパスは，セマフォ資源数を 0 とした同様のテストプログラムにより実行可能である．

分岐網羅と状態依存パス

前節で述べたように wai_sem 内の 7 行目の分岐がどちらのパスに分岐するかは，wai_sem 呼び出し時のセマフォの資源数に依存する．FMP カーネル内の多くの分岐は，このように API 実行時のカーネルオブジェクトや OS の状態（以下，事前状態）によって実行パスが定まる．なお，OS の状態とは，割込み禁止やディスパッチ禁止等の OS が持つ状態のことである．

このように事前状態によって分岐のどのパスを実行するかが一意に定まる分岐を状態依存分岐と呼び，それぞれのパスを状態依存パスと呼ぶ．状態依存分岐を分岐網羅するためには，各状態依存パスを実行するための事前状態を調査し，その事前状態で API を呼び出すテストプログラムを実行すればよい．例えば，シングルコア向けの RTOS である ASP カーネルには状態依存分岐しか存在しないため，全ての状態依存分岐を網羅するテストプログラムを作成し，実行することでカバレッジを 100%にすることができた [29]．

```
 1:void wai_sem(ID semid)
 2:{
 3:   SEMCB *p_semcb = get_semcb(semid);
 4:   disable_int(); //割込み禁止
 5: retry:
 6:   acquire_olock(); // オブジェクトロック取得
 7:   if( p_semcb->semcnt >= 1 ) {
 8:     // <セマフォ取得パス>
 9:     p_semcb->semcnt -= 1;
10:     release_olock(); // オブジェクトロック解放
11:   }
12:   else {
13:     // <セマフォ取得待ちパス>
14:     //タスクロック取得
15:     if(!acquire_nest_tlock()) {
16:       // <オブジェクトロック再取得パス>
17:       goto retry;
18:     }
19:     make_wait(); // タスクを待ち状態へ
20:     release_nest_tlock(); //タスクロック解放
21:     release_olock(); // オブジェクトロック解放
22:     dispatch();
23:   }
24:   enable_int(); //割込み許可
25:}
```

図 4.1: FMP カーネルのセマフォ取得 API（wai_sem）

4.2.2　実行順序依存分岐

コアの実行順序に依存する例

FMP カーネルでは，API を実行中に割込み処理や他コアからカーネルオブジェクトが変更されるのを防ぐため，割込み禁止とロック取得を行ってから，API の処理を行う．ロックはタスクロックとオブジェクトロックの 2 種類がある．タスクを管理するデータ構造をアクセスする時にはタスクロックを，セマフォ等の同期・通信オブジェクトの管理ブロックをアクセスするときはオブジェクトロックを取得する．wai_sem では，まず API 実行直後に割込みを禁止し（4 行目），オブジェクトロックを取得する（6 行目）．その後，セマフォ取得待ちパスを実行する場合には，タスクの管理するデータ構造を操作するためにタスクロックも取得する（14 行目）．ロックはスピンロックで実装され，ロックの取得は図 4.2 に示すロック取得関数により行う．

オブジェクトロックは acquire_olock 関数により取得する．通常のスピンロックの実装では，ロック取得に成功するまでロック取得の試行を繰り返す．acquire_olock 関数をこのように実装すると，割込みが禁止されているため，ロック取得の試行の間に割込みを受け付けることができず，割込み応答性が悪化する．そこで，FMP カーネルでは，割込み応答性を確保するため，スピンロック取得の試行に失敗する毎に割込みを許可している．具体的には，acquire_olock 関数内で呼び出している try_olock 関数は，オブジェクトロックの取得を 1 回試みて，成功した場合に true を，失敗した場合に false をリターンする．8–10 行目は，ロック取得に失敗したときに実行されるロック取得失敗パスであり，割込みを許可する（9 行目）．割込み要求がペンディングしている場合はここで割込みを受け付ける．割込み処理を終了しリターンした後，再び割込みを禁止して（10 行目），ロック取得の試行を繰り返す．

ロック取得失敗パスを実行するシーケンスの例を，図 4.3 の（a）に示す．細い長方形はユーザプログラムを，太い長方形は OS 内部（API）を実行中であることを示す．セマフォ資源数が 2 個以上のセマフォに対して，コア 2 の task2 が wai_sem を呼び出し，acquire_olock 関数によりロックを取得した状態で，コア 1 の task1 が wai_sem を呼び出し acquire_olock 関数内で try_olock 関数を呼び出してロック取得を試みるという実行順序で各コアが実行すると，ロックの取得に失敗して 4 行目の分岐からロック取得失敗パスを実行する．このような各コア実行順序はテストプログラムによる事前状態だけでは実現することができない．そのため，

```
 1: void acquire_olock()
 2: {
 3:   while(true) {
 4:     if( try_olock() ){
 5:         // <ロック取得成功パス>
 6:         break;
 7:     }
 8:     // <ロック取得失敗パス>
 9:     enable_int();  //割込み許可
10:     disable_int(); //割込み禁止
11:   }
12: }
13:
14: bool_t acquire_nest_tlock()
15: {
16:   while(true) {
17:     if( try_tlock() ){
18:         // <ロック取得成功パス>
19:         break;
20:     }
21:     // <ロック取得失敗パス>
22:     enable_int(); //割込み許可
23:     disable_int(); //割込み禁止
24:     //オブジェクロック解放チェック
25:     if ( check_locked() ) {
26:       // <オブジェクトロック再取得パス>
27:       return(false);
28:     }
29:   }
30:   return(true);
31: }
```

図 4.2: FMP カーネルのロック取得関数

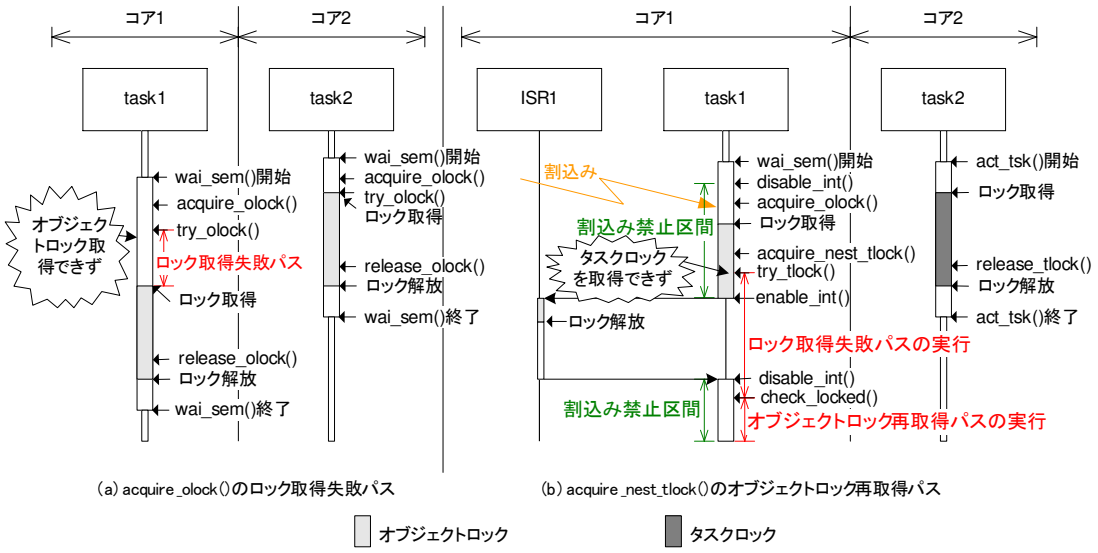


図 4.3: 実行順序依存パスが実行されるシーケンス

acquire_olock 関数の 4 行目の分岐はコアの実行順序によりパスが決まる実行順序依存分岐となる．

割込み発生の順序にも依存する例

次に，wai_sem のタスクロックの取得を例に，コアの実行順序に加えて割込みの発生順序により実行するパスが決まる分岐について説明する．図 4.1 の 14 行目に示すようにタスクロックはセマフォ取得パスにおいて，オブジェクトロックを取得した状態で図 4.2 の acquire_nest_tlock 関数により取得する．acquire_nest_tlock 関数は，acquire_olock 関数と同様に，割込み応答性を悪化させないように，ロックの取得に失敗した場合に割込みを許可する（22 行目）．割込みを受け付けた場合には割込みハンドラを実行するが，オブジェクトロックを取得しているため，割込みハンドラを実行している間他のコアがオブジェクトロックを取得できないという問題が発生する．この問題を解決するため，FMP カーネルでは割込みの入口処理でオブジェクトロックを解放する．そのため，割込み処理から戻り，再度割込みを禁止した後（23 行目）オブジェクトロックを取得しているかを check_locked 関数によりチェックし（25 行目），取得していない場合はオブジェクトロックの取得からやり直すために 26 行目のオブジェクトロック再取得パスを実行する．

オブジェクトロック再取得パスを実行するシーケンスの例を図 4.3（b）に示す．

task1 は wai_sem により資源数 0 のセマフォの取得を試み，task2 は task1 を act_tsk により起動させる[2]．act_tsk は task1 の管理ブロックをアクセスするため，タスクロックを取得する．task2 がタスクロックを取得している状態で task1 が acquire_nest_tlock 関数によりタスクロック取得を試みるとロックの取得に失敗し，ロック取得失敗パスを実行する．この時，22 行目で割込みを許可するタイミングで割込み要求がペンディングしていると，割込みハンドラ ISR1 が実行され，オブジェクトロックが解放される．その後，割込みハンドラからリターンすると，オブジェクトロック再取得パスを実行する．この例で 22 行目で割込みを許可するタイミングで割込み要求がペンディングしている条件は，図 4.1 の 4 行目で割込みを禁止した後から 24 行目までの実行の間で割込みが発生した場合のみである．このように，このパスを実行するかどうかは各コアの実行順序だけでなく割込みの発生タイミングにも依存している．

分岐網羅と実行順序依存パス

実行順序依存分岐により分岐するパスは，事前状態で実行可能なパスと各コアの実行順序と割込みの発生順序に依存して実行可能なパスに分けられる．

前者のパスは，適切な事前状態により実行可能であるため，従来のテストプログラムで決定的に実行可能である．例えば，acquire_olock 関数のロック取得成功パスについて，セマフォ 1 個とコアごとにタスク 1 個を用意し，コア 2 がタスク内で割込み禁止状態かつビジーウェイトとする事前状態で，コア 1 がタスクからセマフォに対して wai_sem を呼び出すと，ロック取得成功パスを実行できる．この事前状態では，コア 2 がロックを取得することはあり得なく，コア 1 がロック取得を必ず成功するため，ロック取得成功パスを決定的に実行することができる．このように，ロック取得成功パスは，状態依存パスと同じ方法で確実に実行することができる．したがって，ロック取得成功パスのように，事前状態で分岐可能な実行順序依存パスは，状態依存パスと同等であると考えることができる．そのため，このようなパスについても状態依存パスと呼ぶ．

一方，後者のパスは，4.2.2 節で述べたように，事前状態だけでは決定的に実行することができないパスである．このようなパスを実行順序依存パスと呼ぶ．

実行順序依存分岐を分岐網羅するためには，状態依存パスに加えて実行順序依存パスを実行する必要がある．FMP カーネルには，83 件の実行順序依存パスが

[2] task1 の起動要求がキューイングされる

存在するため，2,586 件のテストプログラムを実行してもカバレッジは 93.0%に留まった．

4.2.3　連続試行手法

実行順序依存パスを実行する方法として，実行順序依存パスを実行する可能性のある事前状態で実行順序依存分岐を実行するプログラムを複数回実行する方法が考えられる．これを連続試行手法と呼ぶ．

4.2.2 節で示した実行順序依存パスである `acquire_olock` 関数のロック取得失敗パスを連続試行手法で実行するには，コア 1 とコア 2 のそれぞれのタスクで `wai_sem` を繰り返し呼び出すようにする．それぞれのコアで処理を繰り返しているうちに，図 4.3 の（a）に示した実行順序になれば実行順序依存パスを実行する．このように，連続試行手法は，各コアが処理を繰り返し実行する中で，実行順序依存パスを実行する順序になることを期待した手法である．

連続試行手法には，2 つの問題がある．まず，実行順序依存パスを実行する各コアの実行順序の発生する可能性が低い場合，実行順序依存パスを実行するまでに長い時間を要する可能性がある．

もう 1 つの問題は，実行順序依存パスを実行しなかった場合の分析に手間がかかることである．連続試行手法を実施し，カバレッジ計測ツールを用いれば，実行順序依存パスを実行したことは確認できる．ここで，実行順序依存パスを実行しなかった場合，複数の要因が考えられる．まず，連続試行手法の試行回数の不足が挙げられる．また，テストプログラムもしくはテスト対象のプログラムにミスがあり，実行順序依存パスを実行しないことも可能性として挙げられる．これらの要因の切り分けは，カバレッジを計測するだけでは不可能である．また，実行順序依存パスを実行したことは確認できても，設計した通りのコアの実行順序で実行されたかどうかは分からない．コアの実行順序を確認するためには，実行履歴であるトレースログを解析する必要がある．しかしながら，連続試行手法ではプログラムを繰り返し実行するため，トレースログが長くなり，解析の手間が大きいという問題がある．

4.2.4　実行順序制御による実行順序依存パスの実行

何らかの方法により，各コアの実行順序や割込みの発生順序を制御できれば，実行順序依存パスを決定的に実行可能となり，連続試行手法で発生する問題は解決できる．

各コアの実行順序の制御方法として，ソフトウェアによるバリア同期が挙げられる．バリア同期を OS 内に適切に挿入することで，各コアの実行順序を制御できる．しかしながら，この手法は，テスト対象である OS を変更してしまうという問題がある．また，割込みの発生タイミングに依存した実順序依存パスを実行できないという問題もある．

4.3　マルチコア向け RTOS のテスト効率化手法

本章では，実行順序依存パスを決定的に実行するテスト効率化手法を提案する．まず，提案手法の概要について述べ，この手法を実現するために開発したシミュレータ TimingSim について説明する．シミュレータについては，コア実行順序制御機構の要件について述べ，その要件を実現するシミュレータ機構の設計と実装について述べる．なお本研究では，提案手法の有用性の確認を目的としているため，シミュレータを対象とし，ハードウェアによる実現は今後の課題とする．なお，組込みシステムではハードウェアが完成する前に，シミュレータを用いてソフトウェアを開発することがあるため，シミュレータによる提案手法の実現は有用であると考えている．

4.3.1　テスト効率化手法の概要

実行順序依存パスを決定的に実行するため，テストプログラムと実行順序制御を行うシミュレータを協調して動作させる．

実行順序依存パスを実行するためには，まず，対応する実行順序依存分岐を実行するテストプログラムが必要である．これは，連続試行手法と同じように，事前状態を作り出すテストプログラムを作成すればよい．次に，対象とするパスを実行する実行順序を調べ，これを実現する実行順序制御のための記述をテストプログラムに加える．シミュレータがテストプログラムに記述された実行順序を実現することで，対象とするパスを決定的に実行することができる．

例えば，acquire_nest_tlock 関数のロック再取得パス（図 4.2 の 26 行目）を実行したい場合，まず，このパスを実行するための事前状態とコアの実行順序を調べる．その結果，4.2.2 節で述べたように，セマフォ資源が 0 のセマフォが存在する事前状態において，そのセマフォの取得を試みる実行順序が図 4.3（b）のときに，ロック再取得パスを実行することが分かる．そして，この事前状態と実行順序を実現するテストプログラムを実装すれば，実行順序制御を行うシミュレータによって必ずロック再取得パスを実行することができる．シミュレータを用いた実行順序制御の実現については次節以降で，ロック再取得パスを実行するテストプログラムの詳細については 4.4.2 節で述べる．

4.3.2　コア実行順序制御の要件

前節で述べた各コアの実行順序や割込みの発生順序を制御する機構をコア実行順序制御と呼ぶ．FMP カーネルの実行順序依存パスを分析した結果，コア実行順序制御への要件を次のように定めた．

（1）特定のアドレスを特定のコアが実行した場合に，特定のコア（複数）の実行と停止が制御できること．

（2）特定のアドレスを特定のコアが実行した場合に，特定のコア（複数）への割込みを発生できること．

（3）同一アドレスで（1）（2）を複数実現できること．

（4）(1)(2) の実行順序制御や割込みの発生が一度実行された場合にその後の有効・無効を選択可能であること．

（5）実行順序制御に伴い，テスト対象のオブジェクトコードが変化しないこと．

要件（1）により，各コアの実行順序を制御出来るため，コアの実行順序に依存した実行順序依存パスを決定的に実行することができる．さらに，要件（2）より割込みの発生順序を制御出来るため，コアの実行順序に加えて割込み発生順序に依存する実行順序依存パスを決定的に実行することが可能である．

要件（3）については，同一コアの同じタイミングで実行制御と割込みを発生可能とするために定めた．また，将来的に変数チェック等の処理をアクションとして

追加することを考慮しており，要件（1）や変数チェックを同一アドレスで可能とする必要があると考え，定めた．

要件（4）については，テストによっては同じアドレスを複数回実行することがあり，その度に実行順序制御が必要な場合と一度の実行順序制御のみで良い場合があるため，制御を一度実行した後の有効・無効の選択が必要となる．

要件（5）については，テストのためにはテスト対象のソースコードを変更しないことが望ましい．少なくともオブジェクトコードが変更されないことが要求される．

4.3.3　実行制御機構の設計

前節で述べた要件から，シミュレータのコア実行順序制御の設計について述べる．まず基本構造について述べ，次に，テストプログラムが呼び出す実行制御関数について述べる．

基本構造

要件（1）と要件（2）を実現するために，シミュレータに対して，メモリ上の特定のアドレスを実行した場合のシミュレータの処理（アクション）を外部から登録できるようにする．アクションは要件（1）と要件（2）から，各コアの実行と停止（コア実行制御）もしくは各コアへの割込み発生（割込み発生）である．また，要件（2）から同一のアドレスに対して複数のアクションを登録可能とする．要件（4）から，アクションの登録の際には一度実行した後に登録を解除するか（ワンショット）の指定を可能とする．

シミュレータに対するアクションの登録はテストプログラムから実行順序制御関数を呼び出すことにより行う．実行順序制御関数には，アクションを登録するアドレスとアクションの種類，ワンショットの有無を引数として渡す．アドレスの指定方法は，例えば，関数の先頭にアクションを登録した場合には，関数ポインタを引数として渡せばよい．関数の途中を指定したい場合は，指定したい箇所に関数外からも参照可能なラベルを挿入し，そのラベルを引数としてアクションを登録する．このようなラベルの挿入は，テスト対象のソースコードを変更することになるが，オブジェクトコードに影響は与えないと考えられる．そのため，要件（5）を満たす．

実行順序制御関数

アクションをシミュレータに登録するための実行順序制御関数について述べる．まず，指定したアドレスに対してコア実行制御と割込み発生のそれぞれのアクションを登録する実行順序制御関数を用意する．なお，実行順序制御関数からシミュレータへの情報を受け渡し方法（インタフェース）については，4.3.4 節で説明する．

また，OS 内ではなく，テストプログラム内でアクションを実行したい場合があるため，特定のアドレスではなく実行順序制御関数呼び出し時にアクション（即時アクション）を実行する実行順序制御関数もそれぞれ用意する．実行順序制御関数の一覧を以下に示す．

（1）`add_pc_exccnt(uint_t *pc, uint_t active_prc, bool_t oneshot)`
pc で指定したアドレスに対して，コア実行制御をアクションとして登録する．active_prc で各コア実行制御を指定する．コア毎にビットが対応しており，値が 1 ならば実行，0 なら停止することを表す．oneshot でワンショットの指定をする．

（2）`add_pc_intraise(uint_t *pc, uint_t int_prc, uint_t intno, bool_t oneshot)`
pc で指定したアドレスに対して，割込み発生をアクションとして登録する．発生させる割込みの割込み番号は intno で指定する．int_prc で割込みを発生させるコアをビットで指定する．

（3）`add_cur_exccnt(uint_t active_prc)`
即時アクションとしてコア実行制御を実行する．active_prc で各コア実行制御を指定する．

（4）`add_cur_intraise(uint_t int_prc, uint_t intno)`
即時アクションとして割込み発生を実行する．int_prc で割込みを発生させるコアをビットで指定する．

4.3.4　TimingSim の実装

提案機構を実現したシミュレータである TimingSim の実装について述べる．TimingSim は文献 [28] の Windows PC 上で動作するマルチプロセッサシミュレータをベースとしており，ARM アーキテクチャをシミュレーションする．まず，アク

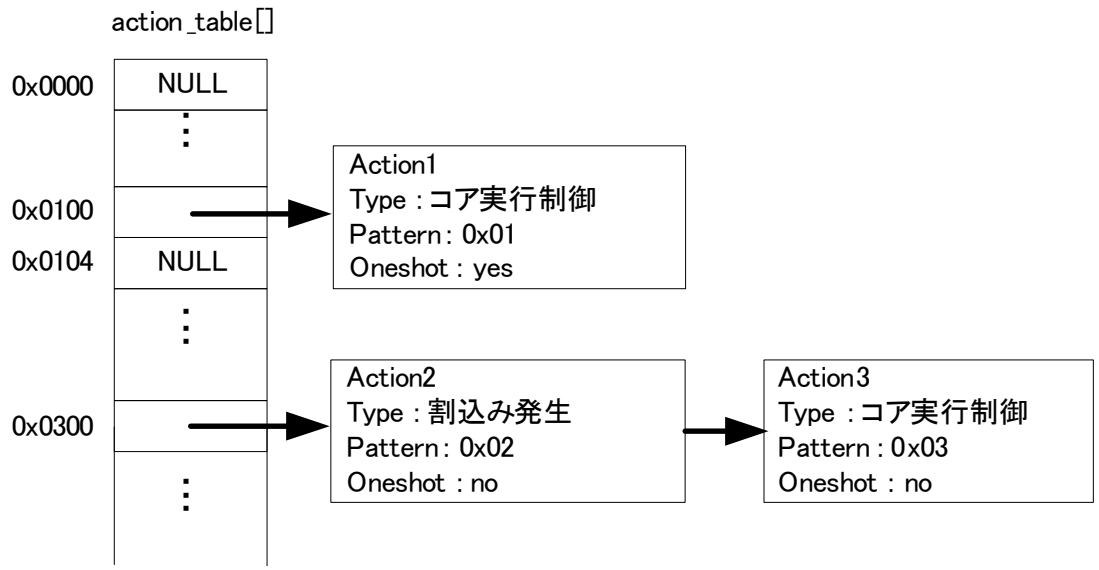


図 4.4: アクションを管理するデータ構造

ションの管理方法について説明し次に，テストプログラムが呼び出す実行制御関数とのインタフェースについて説明する．最後に，アクションを実行する TimingSim の実行フローについて説明する．

アクションの管理

TimingSim は登録されたアクションを図 4.4 に示す `action_table` と呼ぶデータ構造により管理する．`action_table` はキューヘッダの配列となっており，配列の各要素はメモリのアドレスに対応している．あるアドレスのキューヘッダには，そのアドレスに登録されたアクションの状態が登録される．提案手法では，同一のアドレスに複数のアクションを登録できるため，キュー構造となっている．アクションが登録されていないアドレスは NULL とする．

アドレス毎にキューを持つことにより，シミュレータ実行時に実行アドレスに登録されたアクションを容易に得ることができ，アクションの探索によりシミュレーション時間が長くなるのを低減できる．一方，`action_table` のサイズはシミュレーションするメモリサイズに比例して大きくなるというデメリットがある．しかしながら，TimingSim を実行する PC が数 Gbyte のメモリを搭載することが一般的であるのに対して，FMP カーネルは 100kbyte 程度のメモリがあれば動作するため，問題とならない．

実行順序制御関数とのインタフェース

テストプログラムが呼び出す実行順序制御関数は C 言語の関数として実現される．実行順序制御関数の引数は TimingSim に伝える必要がある．

TimingSim は命令セットシミュレータであるため，実行順序制御関数からはハードウェアとして見える．そのため，実行順序制御関数と TimingSim とのインタフェースはデバイスレジスタとした．具体的には TimingSim のシミュレーション対象のコアに周辺デバイスを追加する．周辺デバイスは複数デバイスレジスタを持ちそれぞれのデバイスレジスタが，実行順序制御関数の引数に対応する．実行順序制御関数は，引数の内容を対応したデバイスレジスタに書き込む．デバイスレジスタに値が書き込まれると，その情報を元に TimingSim は `action_table` にアクションを登録する．

なお，即時アクションが登録された場合には，次に実行する命令のアドレスへのワンショットのアクションとして `action_table` に登録する．

実行フロー

TimingSim のコア毎の実行フローを図 4.5 に示す．まずプログラムカウンター（PC）が示すアドレスから命令をフェッチして実行し，その結果により PC を更新する．次にタイマの更新等の IO 処理を行う．これらは基としたシミュレータと同じ動作である．

その後，`action_table` から PC が示すアドレスに登録されているアクションがあるかチェックする．アクションが登録されていればそのアクションに応じて実行制御ないし割込み発生を行う．この処理を PC が示すアドレスに登録されている全てのアクションに対して行う．コア実行制御のために，実行か停止かを管理する動作状態フラグをプロセッサ毎に用意しているため，コア実行制御の場合は，このフラグを変更する．割込み発生の場合は，指定したコアへ割込みを発生させる．PC が示すアドレスに登録されている全てのアクションを実行した後，自プロセッサの動作状態フラグをチェックし，停止要求があれば解除されるまで実行を停止する．

4.4　評価

提案手法の有用性の評価のため，まず，連続試行手法による FMP カーネルの代表的な実行順序依存パスの実行する割合を計測する．次に，提案機構を用いた実

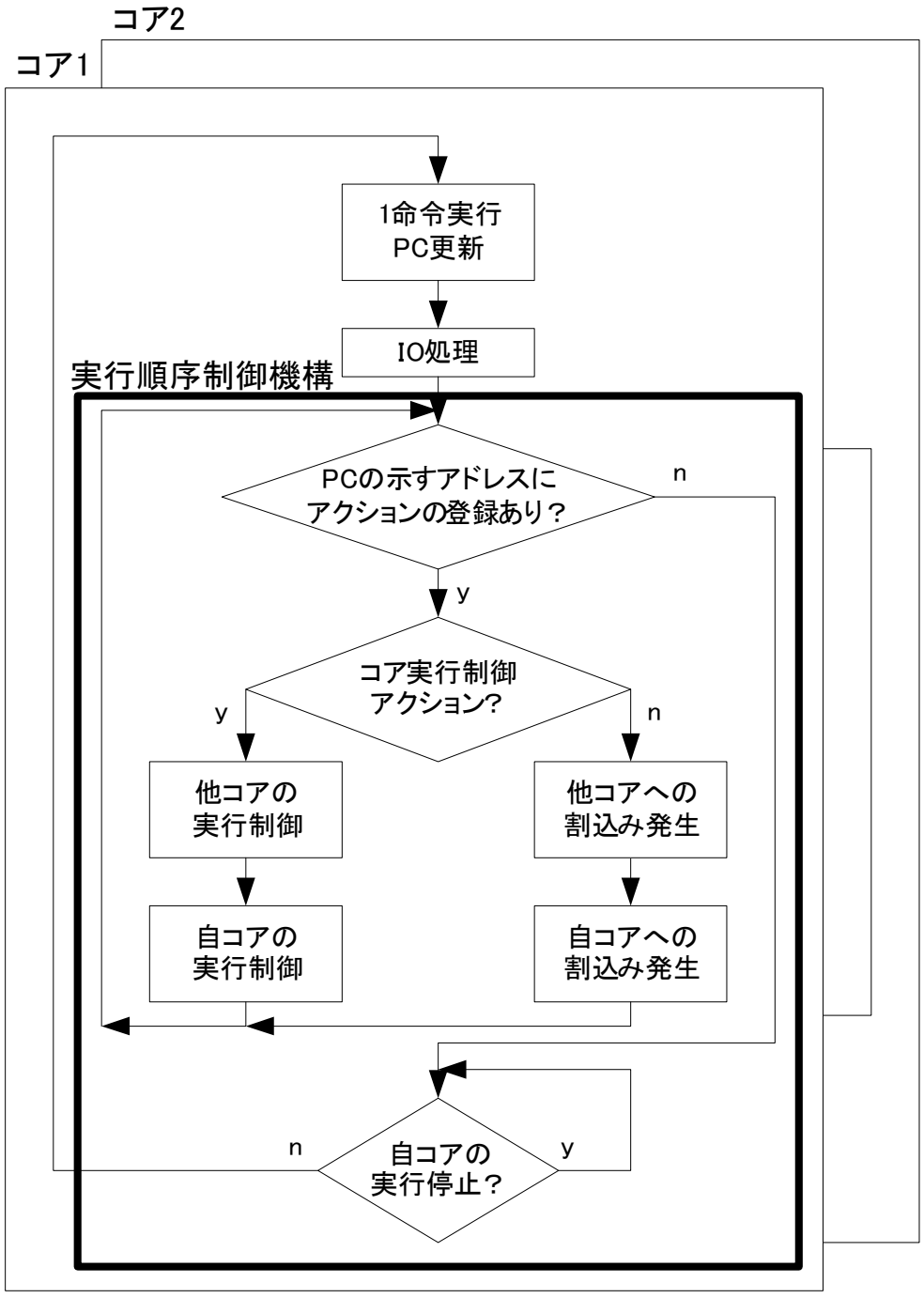


図 4.5: TimingSim の実行フロー

表 4.1: 連続試行手法による実行順序依存パスの実行割合

関数名	コア実行順序依存	割込み発生順序依存	実行コンテキスト	割合 [%]
acquire_olock	yes	no	タスク	24.1
t_acquire_tsk_lock_self	yes	no	タスク	14.1
ter_tsk（API）	yes	no	タスク	4.5
wait_tmout	yes	no	割込み	0.0
acquire_nest_tlock	yes	yes	タスク	0.0

行順序依存パスの実行例について述べる．そして，FMP カーネルの実行順序依存分岐の分岐網羅について述べる．

4.4.1　連続試行手法による実行順序依存パスの実行

連続試行手法により実行順序依存パスの実行を試みる．実行には，50MHz のプロセッサを 2 個搭載したシステムを用いた．プロセッサは NiosⅡ であり，FPGA 上で動作する．

実行順序依存パスが存在する OS 内の関数に対して，実行順序依存パスを実行する可能性がある事前状態で 100 万回呼び出した場合に実行順序依存パスを実行した割合を測定した．測定対象の関数は，特徴が異なる 4 種類の関数と 1 種類の API とした．それぞれの関数内の実行順序依存パスの実行する割合を表 4.1 に示す．表 4.1 には，それぞれの関数の実行順序依存パスが依存する実行順序の要因と関数の実行コンテキストを記載した．

実行順序依存パスを実行できた acquire_olock 関数，t_acquire_tsk_lock_self 関数，ter_tsk の 3 種類の関数は，全てのコアの実行順序にのみ依存し，かつ，タスクから実行される．これらは，実行順序依存パスを実行する条件と一致する可能性が低くなるにつれて，実行順序依存パスを実行する割合が低下していると考えられる．acquire_olock 関数と ter_tsk の違いについて，説明する．acquire_olock 関数については，他のコアがロックを取得している間にロックの取得を試みれば，実行順序依存パスは実行される．API 内部ではロックを取得してから必要な処理を行うため，他のコアがロックを取得している時間はおおよそ API の実行時間である．一方，ter_tsk の実行順序依存パ

スはデッドロック回避と呼ばれる処理を実行するときのパスである．API 内部でロックを一旦解放して再取得する間に，他のコアで関連する API が実行され解放したロックを取得する場合に実行される．ロックを解放してから再取得する間の命令は数命令であり，API の実行時間より小さい．実行順序依存パスを実行するのは，acquire_olock 関数については API を実行している間であり，ter_tsk についてはロックを解放してから再取得する間である．このため，acquire_olock 関数に比べて ter_tsk の実行順序依存パスを実行する割合が低くなったと考えられる．

wait_tmout 関数と acquire_nest_tlock 関数は，100 万回の試行での実行割合は 0%であった．さらに 6 時間実行を継続しても実行されなかった．wait_tmout 関数は，OS が管理する時間を更新するタイマ割込みで呼び出され，時間待ち状態のタスクを起床させる関数である．wait_tmout 関数内のデッドロック回避処理を行うパスが実行順序依存パスとなっている．この実行順序依存パスを実行するには，他のコアで起床対象のタスクの状態を変更する API が同時に実行される必要がある．FMP カーネルは，コア毎に時間を管理しているため，各コアに独立したタイマ割込みが入る．各タイマ割込みは多くの場合同じタイミングで発生して，それぞれ wait_tmout 関数を呼び出す．そのため，実行順序依存パスを実行するために必要な起床対象のタスクの状態を変更する API を他コアが発行する可能性は低くなる．以上の理由により，wait_tmout 関数の実行順序依存パスは実行されなかったのだと考えられる．

acquire_nest_tlock 関数については，オブジェクトロック再取得パスの実行はコアの実行順序だけでなく割込みの発生順序にも依存する．評価環境で発生する割込みは 1 ミリ秒毎に発生するタイマ割込みのみであり，オブジェクトロック再取得パスを実行する可能性はコアの実行順序だけに依存するパスより低い．このため，連続試行手法では実行されなかったと考えられる．

4.4.2　提案手法による実行順序依存パスの実行例

TimingSim を用いて，図 4.2 のオブジェクトロック再取得パスを実行する方法について述べる．テストプログラムを図 4.6 に，このテストプログラムを用いたシーケンスを図 4.7 に示す．

テストプログラムのタスク構成は，コア 1 で実行される task1 とコア 2 で実行される task2 であり，それぞれ実行されるとテストプログラム中の関数 task1 と task2

```
 1: bool_t startflg = false;
 2: void task1()
 3: {
 4:   barrier_sync(); //バリア同期1
 5:   add_pc_exccnt(acquire_nest_tlock,
 6:                  0x2, true); //a2登録
 7:   add_pc_intraise(acquire_nest_tlock,
 8:            0x1,INTNO1,true); //a3登録
 9:   barrier_sync(); //バリア同期2
10:   add_cur_exccnt(0x1); //a1即時実行
11:   startflg = true; //task2再開
12:   wai_sem(SEM1);
13: }
14:
15: void isr1(void)
16: {
17:   add_cur_exccnt(0x3); //a5即時実行
18: }
19:
20: void task2()
21: {
22:   barrier_sync(); //バリア同期1
23:   add_pc_exccnt(release_tlock,
24:                  0x1, true); //a4登録
25:   barrier_sync(); //バリア同期2
26:   while(!startflg);
27:   act_tsk(TASK1);
28: }
```

図 4.6: オブジェクトロック再取得パスの実行順序制御による実行

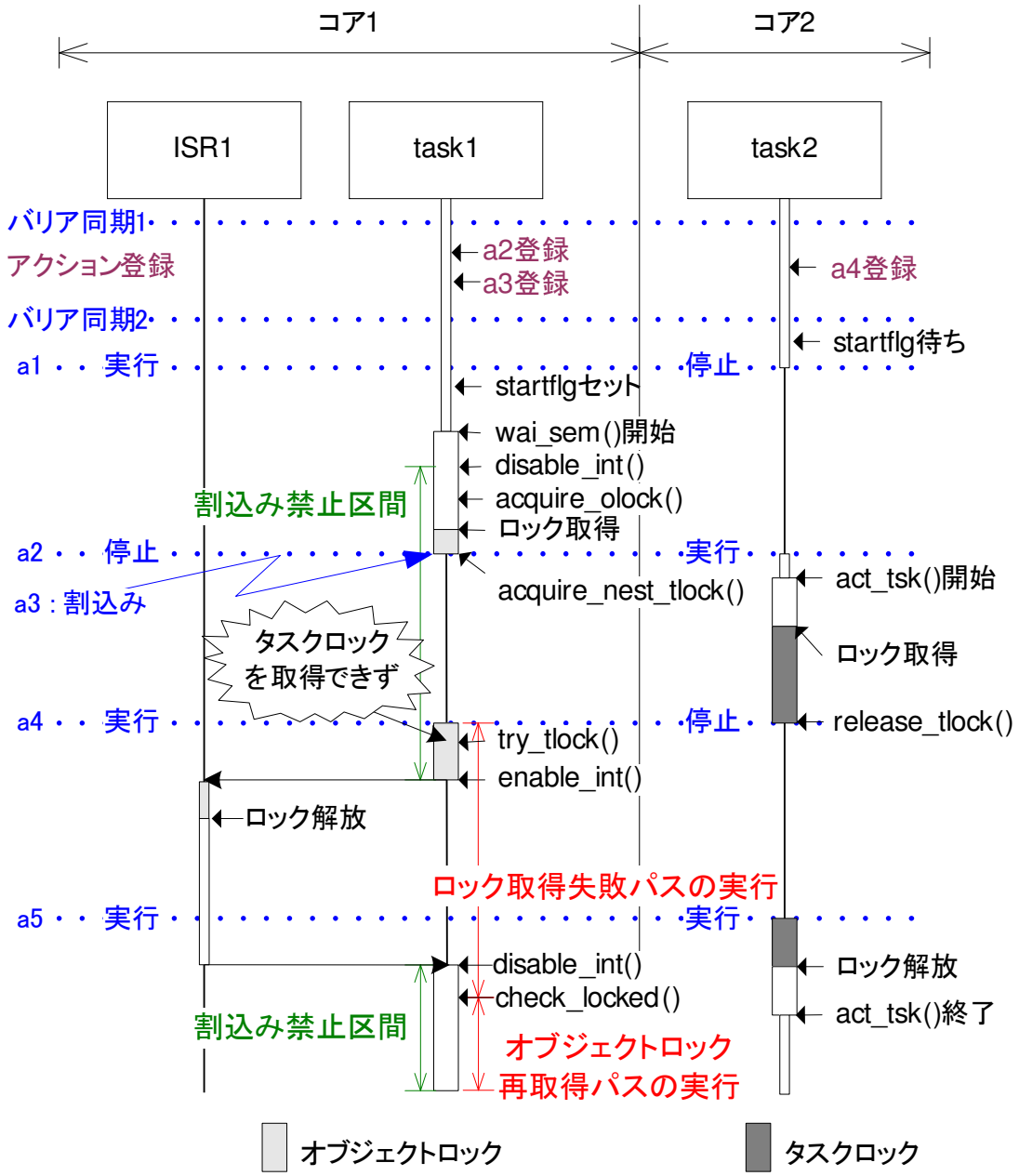


図 4.7: オブジェクトロック再取得パスの実行順序制御による実行

を実行する．コア 1 には割込み番号 INTNO1 の割込みが入ると実行される割込みハンドラ ISR1 が登録されている．ISR1 は実行されるとテストプログラム中の関数 isr1 を実行する．また，資源数 0 のセマフォ SEM1 が存在する．

テストプログラムが実行されると，コア 1 で task1 がコア 2 で task2 が実行される．task1 と task2 はまずバリア同期を行うライブラリ関数である `barrier_sync` 関数によりバリア同期を行う（バリア同期 1）．同期後，OS 内の関数に対してアクションを登録する．task1 は，コア 1 が `acquire_nest_tlock` 関数を実行した時にコア 1 を停止，コア 2 を実行するコア実行性制御アクション（a2）を登録する．次に，同様にコア 1 が `acquire_nest_tlock` 関数を実行した時にコア 1 に対して INTNO1 の割込みを発生させるアクション（a3）を登録する．task2 は，コア 2 が `release_tlock` 関数を実行した時にコア 1 を実行，コア 2 を停止するアクション（a4）を登録する．それぞれのタスクはアクションの登録が終わると，再びバリア同期を行う（バリア同期 2）．

バリア同期 2 の後，task2 は `startflg` がセットされるのをループで待つ．task1 は，即時アクション a1 によりコア 2 を停止させる．その後，`startflg` をセットする．`startflg` はセットされるが，a1 によりコア 2 は停止しているため，task2 はこの時点では動作を再開しない．

次に task1 は SEM1 に対して `wai_sem` を呼び出す．図 4.1 で示したように，`wai_sem` は，まず割込みを禁止した後に `acquire_olock` 関数によりオブジェクトロックの取得を試みる．この際コア 2 は停止しているため，`acquire_olock` 関数は確実にオブジェクトロックを取得可能である．

オブジェクトロックの取得後，SEM1 の資源数をチェックして（図 4.1 の 7 行目），資源数が 0 であるため，セマフォ取得待ちパスを実行し `acquire_nest_tlock` 関数を呼び出すことによりタスクロックの取得を試みる．`acquire_nest_tlock` 関数が呼び出されると，事前に登録していた，アクション a2 と a3 が TimingSim により実行される．a2 によりコア 1 は停止，コア 2 は実行（再開）される．また，a3 よりコア 1 に対して INTNO1 の割込みを発生する．

コア 2 の実行が再開されると，task2 は startflg が task1 によってセットされたため，実行を再開する．そして，task1 に対して `act_tsk` を呼び出す．`act_tsk` 内では，タスクロックを取得し，task1 の起動要求キューイングを追加したのち，タスクロックを解放するために，`release_tlock` 関数を呼び出す．`release_tlock` 関数が実行されるとタスクロックを解放する前に，アクション a4 が TimingSim により実行され，コア 1 が実行，コア 2 が停止となる．

コア 1 が再開されると，acquire_nest_tlock 関数の実行が再開され，図 4.2 の 17 行目にあるように try_tlock 関数によりタスクロックの取得を試みる．この時点でタスクロックはコア 2 が取得しているため，try_tlock 関数は失敗して，ロック取得失敗パスを実行する．そして，割込みを許可すると（図 4.2 の 22 行目），アクション a3 で発生させた INTNO1 の割込みがペンディングしているため，割込みを受け付ける．OS の割込み入口処理でオブジェクトロックを解放し，割込みハンドラ ISR1 を実行する．ISR1 は実行されると，即時アクション a5 により両コアを実行する．その後 ISR1 からリターンする．

割込み処理からのリターン後，再び割込みを禁止して（図 4.2 の 23 行目），check_locked 関数によりオブジェクトロックが解放されたかチェックする．オブジェクトロックは OS の割込み入口処理で解放されているため，実行の目標のパスであるオブジェクトロック再取得パスを実行する．また，コア 2 は関数を再開して，タスクロックを解放して，act_tsk を終了する．

以上のように，TimingSim とテストプログラム及び実行順序制御関数を組み合わせることにより，実行順序依存パスを決定的に実行することが可能である．

4.4.3　提案手法による実行順序依存分岐の分岐網羅

TimingSim を用いて，FMP カーネルの実行順序依存分岐の分岐網羅を試みた．

4.2.2 節で述べたように実行順序依存分岐には，状態依存パスと実行順序依存パスが存在する．状態依存パスに関しては，通常のテストプログラムを 2,586 件開発し，これを実施することで全て実行可能であった [27]．実行順序依存パスに関しては，実行順序制御関数を用いたテストプログラムと TimingSim を用いて，FMP カーネルに存在する全ての実行順序依存パスを決定的に実行可能であった．作成したテストプログラムの個数は 83 件であり，テストプログラム中で用いた実行順序制御関数は，4.3.3 節で説明した (1) の関数が 97 回，(2) の関数が 3 回，(3) の関数が 173 回，(4) の関数が 18 回であった．

提案手法を用いることにより，連続試行手法で 6 時間かけても実行できなかった wait_tmout 関数と acquire_nest_tlock 関数の実行順序依存パスを実行することができた．Intel Core2Duo E8400 (3.0GHz)，2GB メモリの環境において，テストプログラムの実行時間はそれぞれ 0.48 秒，0.43 秒であり，83 件全てでは 10.69 秒であった．このように，連続試行手法では実行順序依存パスの実行に長い時間を要する場合でも，提案手法を用いれば短時間で実行することができる．

提案機構を用いて wait_tmout 関数内の実行順序依存分岐の分岐網羅を行ったところ，実行順序依存パスに 1 件の不具合を発見した．この実行順序依存パスは，連続試行手法により 6 時間繰り返し実行しても一度も実行されなかったパスであり，提案機構を用いなければ発見が困難なバグであったと考えられる．

以上のことから，提案手法を用いることで，テスト実施の時間を効率化できることを示した．また，実行順序依存パスに存在する不具合を実際に発見することで，提案手法の有用性を示せた．

4.5　関連研究

実行順序依存パスを実行する方法として，ソフトウェアとハードウェアのアプローチが存在する．

ソフトウェアによる方法として，マルチスレッドアプリケーションの単体テスト用ツールである ConTest[8] が挙げられる．ConTest は，テストプログラムの実行をずらして繰り返し実行することで，実行順序依存パスの実行を試み，バグの検出をサポートする．ConTest は連続試行手法を効率化するものであり，ConTest を利用しない場合と比べてテストの実施時間を削減することが可能であるが，実行する可能性が低いパスを実行するには時間を要すると考えられる．また，実行順序依存パスを決定的に実行することができない．

マルチコア上で動作するソフトウェアのデバッグをサポートするデバッグハードウェアが提案されている [15]．このデバッグハードウェアを利用することでコアの実行順序を指定することができるため，コアの実行順序にのみ依存する実行順序依存パスを実行することができる．しかしながら，割込みを考慮しておらず，割込み処理に依存する実行順序依存パスを決定的に実行することができない．割込みを各コアに入力するハードウェアを実現・追加することで，TimingSim と同様に分岐網羅することができるが，ハードウェアによる実現では，あるアドレスに対するアクションの登録数に制限があると考えられる．TimingSim ではアクションをソフトウェアのキューにより管理しているため，論理的には登録数に制限がない．

また，ユーザプログラムに対して実行順序依存パスのないプログラミングモデルを提供する OS が提案されている [5]．これらの手法の対象はユーザプログラムであるため，OS の実行順序依存パスを無くすことはできない．同様の考え方で OS の実行順序依存パスを無くすためには，ハードウェアにより OS を決定的に動作させる必要があるが，そのようなハードウェアは私の知る限り存在しない．存在し

たとしても，性能が大きく限定されると予想される．

4.6　まとめ

複数のプログラムが並列もしくは並行動作する環境では，各コアの実行順序と割込みの発生順序に依存してパスが定まる，実行順序依存パスが存在する．実行順序依存パスを実行するには，コアの実行順序を制御する手法が有効である．本章では，コアの実行順序及び割込み発生順序を制御可能なシミュレーション機構を用いたテスト効率化手法を提案した．テストプログラムと提案機構を実装したシミュレータが協調して動作することで，マルチコア向け RTOS である FMP カーネルの実行順序依存分岐を効率的に分岐網羅できることを示した．

本研究では，FMP カーネルのソースコードのうち，ハードウェアに依存しない部分をテスト対象とした．ただし，ロック取得について，FMP カーネルでは TAS 命令を持つか TAS 命令に相当するハードウェアが用意されていることを想定している[3]ため，それ以外の機構を提供するハードウェアの場合は実行順序依存パスが変化する可能性がある．

本研究で用いた実行制御機構は，実機用デバッガが持つブレークポイント機能を用いることにより，実機に対しても実現可能と考えられるため，実機用の実行順序制御への拡張等が考えられる．また，提案機構は FMP カーネルには依存しておらず，他の OS やアプリケーションの実行順序依存分岐を網羅するために利用することも可能であると考えられる．これらの検証については，今後の課題とする．

[3]ほとんどのハードウェアはこの想定に合致する

第5章　車載システム向け通信ミドルウェアのマルチコア拡張

5.1　概要

自動車に搭載されるコンピュータである ECU は複雑化，高機能化が進んでいる．自動車 1 台に搭載される ECU の総数は増大が進んでおり，ソフトウェアも大規模化の一途をたどっている．ECU はネットワークを用いて相互に接続されており，ECU 間の通信プロトコルとして CAN [9] が広く用いられている．しかし，ECU の増加によって，ECU の設置スペースとそれらを接続する配線のスペースを確保するのが困難になってきている．このため，複数の ECU を 1 つの ECU に統合し，ECU の総数を減らす動きがある．

ECU の総数を減らすために統合を行うには，プロセッサの性能向上が必須である．マルチコアプロセッサは消費電力や発熱量の点でシングルコアプロセッサより優れており，他の組込みシステムより発熱に対する条件が厳しい自動車においても利用が検討されている．車載システムのアーキテクチャとして AUTOSAR が注目されている．現在最新の AUTOSAR Release 4.0 では，マルチコアに対応した OS が仕様化されたが，通信機能を提供するミドルウェア（通信ミドルウェア）を含む多くのソフトウェアが，1 つのコアで動作するアーキテクチャとなっている．通信ミドルウェアの動作しないコアは，通信ミドルウェアの動作するコアへ通信処理を依頼する必要があり，処理を依頼するための実行オーバヘッドが増えてしまう．処理を依頼する実行オーバヘッドに着目し，これをなくす手法としてジャイアントロックアプローチが提案されている [17]．しかし，リアルタイム性に関する言及が少なく，実際の車載システムで利用するための評価が不足している．

本章では，AUTOSAR 仕様の通信ミドルウェアをマルチコアシステムで実装する際のアーキテクチャについて議論する．想定するシステムは，CAN プロトコルを用いる 2 コアのシステムである．まず，通信ミドルウェアをマルチコア拡張する際に求められる要件を示す．AUTOSAR 仕様に従った通信ミドルウェアおよび

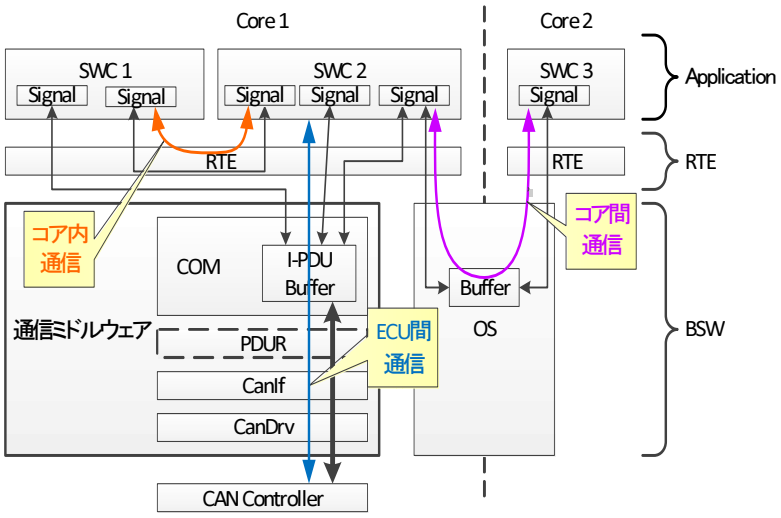


図 5.1: AUTOSAR における SWC 間の通信

ジャイアントロックアプローチでは，示した要件のうち，実行オーバヘッドとロック取得時間の低減を両立できないこと，さらに実行並列性がないことを明らかにする．そして，これらの問題の解決を目指した手法を 2 つ提案し，実装および評価を行う．評価により，提案手法が問題を解決していることを示す．また，ジャイアントロックアプローチは通信ミドルウェアが持っている定期通信の機能を考慮しておらず，リアルタイム性を損なわないように実装を行うと，実行オーバヘッドが大きくなってしまう問題があることを示す．

5.2 AUTOSAR 通信ミドルウェア

5.2.1 AUTOSAR における通信の概要

AUTOSAR のシステムは，アプリケーションを構成する SWC (software component)，RTE (runtime environment)，BSW（basic software）によって構成される（図 5.1）．BSW は OS や通信スタックなど，基本的な機能を提供するものである．RTE は SWC と BSW の仲介をする．

AUTOSAR のシステムにおいて，SWC 間の通信は，図 5.1 のようにコア内通信，コア間通信，ECU 間通信の 3 つに分類できる．コア内通信は同じコアにある SWC 間の通信であり，この通信は RTE が行う．コア間通信は同じ ECU 内で別コアにある SWC 間の通信である．この通信はコアをまたぐ通信であり，OS により実現

される．この機能は，AUTOSAR Release 4.0 で追加された．ECU 間通信は，通信ミドルウェアを用いて外部の ECU にある SWC 間で行う通信である．コア内通信とコア間通信は通信ミドルウェアを用いないため，通信ミドルウェアを用いるのは ECU 間通信のみである．

SWC はシグナルと呼ばれる単位で通信を行う．シグナルは通信データの最小単位であり，サイズなどは通信チャネルごとに定義する．

AUTOSAR のマルチコアアーキテクチャでは，図 5.1 のように通信ミドルウェアを単一のコア（コア 1）で動作させる．一方のコア（コア 2）は，通信ミドルウェアがなく，ECU 間通信を行う際にはコア 1 に通信を依頼する．例えば，図 5.1 では SWC 3 と外部の ECU にある SWC が通信を行う際には，まず，SWC 3 と SWC 2 がコア間通信を行い，そして SWC2 が通信ミドルウェアを用いて ECU 間通信を行う．コア 2 の SWC が ECU 間通信を行うには，コア 1 にプロキシとなる SWC が必要となる．

5.2.2　通信ミドルウェアの構成

通信ミドルウェアは COM，PDUR (PDU Router)，CanIf (CAN Interface)，CanDrv (CAN Driver) で構成される（図 5.1 ）．

CanIf 層と CanDrv 層は，それぞれ CAN プロトコル，CAN コントローラを抽象化し上位層へのインタフェースを与える．他の通信プロトコルを使う場合は，そのプロトコルのインタフェースやドライバに置き換わる．PDUR 層は，データのルーティングを行うもので，ゲートウェイとなる ECU で使われる．ゲートウェイでない ECU の場合は，PDUR 層を省略できる．省略した場合，PDUR 層の API はマクロで置き換える．例えば，送信で COM 層が PDUR 層の API を呼びだす箇所は，マクロにより CanIf 層の API に置き換わる．本研究では，簡単化のため，PDUR 層を省略した構成を前提とする．

COM 層は，通信プロトコルに適した I-PDU と呼ばれるデータの単位で CanIf 層とやり取りを行う．そのため，シグナルと I-PDU の変換を行う機能を持っており，送信処理では複数のシグナルを 1 つの I-PDU にパッキングする処理を，受信処理では 1 つの I-PDU から複数のシグナルにアンパッキングする処理を行う．COM 層は，これらの処理を行うため I-PDU バッファを持つ．サイズについては，CAN で通信できるデータは最大 64 ビットであるため，I-PDU とシグナルも最大 64 ビットである．

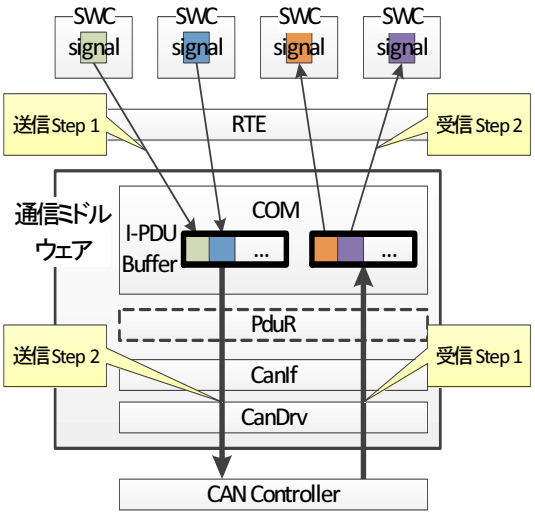


図 5.2: ECU 間通信処理の手順

COM 層，CanIf 層，CanDrv 層の処理を比べると，最も複雑なのは COM 層である．その要因の 1 つとして，パッキング処理（とアンパッキング処理）がある．シグナルと I-PDU の対応はビット単位で指定可能であり，エンディアンなどの違いも考慮する必要があるため，実装は複雑になりがちである．また，次節で述べるような通信機能の多くが，COM 層で提供されることも大きな要因である．

5.2.3 ECU 間通信の手順

通信ミドルウェアは，送信と受信でそれぞれ 2 段階の処理を行う（図 5.2）．送信は，以下の 2 段階の処理を行う．

送信 Step 1. SWC が RTE を経由して COM 層にシグナルを渡す．COM 層は，パッキング処理を行い，対応する I-PDU バッファを更新する

送信 Step 2. COM 層は CanIf 層に対して I-PDU を渡して送信を要求する．CanIf 層は CanDrv 層を経由し，I-PDU を CAN コントローラに書き込む

受信は，以下の 2 段階の処理を行う．

受信 Step 1. CAN コントローラが受信したデータを CanIf 層を経由して COM 層に渡す．COM 層は，受信データを CAN コントローラから I-PDU バッファにコピーする

受信 Step 2. COM は，アンパッキング処理を行い，SWC に受信の通知などを行う

送信と受信の Step2 の処理を行うタイミングは，I-PDU ごとに設定が可能である．Step1 と Step2 を連続して実行する場合を即時通信と呼び，一定周期で Step2 を実行する場合を定期通信と呼ぶ．さらに，通信が送信の場合は即時送信，定期送信と呼び，受信の場合は即時受信，定期受信と呼ぶ．周期通信を実現するため，`Com_MainFunctionTx` と `Com_MainFunctionRx` という API が COM 層に用意されている．この API はスケジューラから周期的（例えば 1ms ごと）に呼ばれ，実行回数をカウントする．実行回数のカウント値と I-PDU に設定された周期を比較して，設定された周期となった場合に Step2 を実行することで定期通信を実現できる．

5.3　マルチコア拡張の要件

AUTOSAR 通信ミドルウェアをマルチコア拡張するにあたって，以下の要件を定めた．

要件 1. ロック取得時間が短いこと
コア間で共有するリソースについては，ロックなどによる排他制御が必要である．あるコアがロックを取得している時間は，他のコアがロックを取得できず，ロックを取得できるまで待たなければならない．ロック取得待ちの時間は，有効な処理を行っていないため，できる限りロック取得時間を減らす必要がある．

要件 2. 割込み応答時間が短いこと
AUTOSAR 仕様は，自動車を対象としているため，割込み応答時間が長いとシステムに要求される時間制約を守ることができない．そのため，割込み応答時間が短いことが求められる．

要件 3. マルチコア拡張による実行オーバヘッドが小さいこと
従来の通信ミドルウェアと比較した際の実行時間の増分が小さいことが望ましい．マルチコアシステムにおいては，ロックの競合によって実行時間が長くなる場合がある．ここでの実行オーバヘッドには，ロックの競合によるオーバヘッドを含まないこととする．また，マルチコア拡張によってデータのコピー回数が増えることがあり，これも実行オーバヘッドの要因

となる．データのコピーが必要となるのは，処理を別のコアに依頼する場合である．この場合は，依頼した処理とその対象のデータをキューイングする必要があり，データをコア間で共有するメモリ領域にコピーする必要がある．

要件 4. 実行並列性が高いこと
マルチコアシステムでは，処理の並列化によりシステム全体のスループット向上を実現するため，通信ミドルウェアの処理にも実行並列性が求められる．本研究では，通信にかかる処理のうち，通信するコア自身で実行し，他コアの影響を受けない処理の割合が高いことを具体的な要件とする．これを言換えると，通信するコア自身がロックを取得せずに実行する処理の割合である．この割合が高いほど，各コアがそれぞれで処理を実行することができ，並列性を高くすることができる．

要件 5. メモリ使用量が少ないこと
組込みシステムでは，メモリ使用量への制約が強いことから，少ないメモリ使用量で通信ミドルウェアを実装することが望ましい．

要件 6. ソースコードの変更量が少ないこと
シングルコアシステムで動作する既存の通信ミドルウェアのソースコードに対する変更量が少なく，実装が容易であることが望ましい．これは，テスト実施の作業量を低減する効果もあると考えられる．

5.4　既存手法

2 つの既存手法と要件との対応について述べる．

5.4.1　AUTOSAR アプローチ

AUTOSAR アプローチは，シングルコアシステムで動作する既存の通信ミドルウェアに対して一切変更を行わずに，マルチコアシステムで通信ミドルウェアを利用する方法である．具体的には，既存の通信ミドルウェアを 1 コアだけで動作させ，他のコアが ECU 間通信を行う際には，コア間通信を行って通信ミドルウェアが動作するコアへデータを渡す．図 5.3(a) に AUTOSAR アプローチのアーキテクチャを示す．

要件との対応については以下の通りである．

要件 1 との対応　ロックを取得するのはコア間通信を行う間のみで良いので，ロック取得時間は短い．

要件 2 との対応　割込み禁止をするのは，コア間通信と通信ミドルウェアを実行するときである．シングルコアからの大きな変更はなく，割込み応答時間は短い．

要件 3 との対応　コア 2 の SWC が ECU 間通信を行うためにはコア 1 の SWC を経由する必要があり，実行オーバヘッドが大きい．コア 2 の SWC とコア 1 の通信ミドルウェアとの間では，シグナルのコピーが 3 回行われる．コア 1 の SWC が ECU 間通信を行う際の実行オーバヘッドは，シングルコアと同じである．

要件 4 との対応　通信ミドルウェアはすべてコア 1 で動作するので，並列性はない．

要件 5 との対応　通信ミドルウェアの動作するコアに対して処理を依頼するため，処理とデータのキューイングが必要である．データのキューイングのために必要なメモリサイズは，アプリケーションが必要とするコア間通信の頻度やデータの大きさによって決まる．

要件 6 との対応　ソースコードの変更はない

5.4.2　ジャイアントロックアプローチ

ジャイアントロックアプローチ [17] は，単一のロック（ジャイアントロック）を取得してから既存の通信ミドルウェアを実行する手法である．図 5.3(b) にジャイアントロックアプローチのアーキテクチャを示す．AUTOSAR アプローチではコア間通信のための実行オーバヘッドが増えるが，ジャイアントロックアプローチではマルチコア拡張による実行オーバヘッドはロック取得のみである．ただし，ロックを取得したまま通信ミドルウェアの処理を全て実行するため，ロック取得時間は長い．

要件との対応については以下の通りである．

要件 1 との対応　ロックを取得して通信ミドルウェアを実行するので，ロック取得時間は長い

要件 2 との対応 割込み禁止は，既存の通信ミドルウェアと同等である

要件 3 との対応 ロック取得と解放処理のみが実行オーバヘッドであり，これらの実行オーバヘッドは小さい

要件 4 との対応 ロック取得中は他のコアが実行できないため，並列性はない

要件 5 との対応 ロック取得と解放処理に必要なメモリ使用量は少ない

要件 6 との対応 ソースコードの変更はロック取得と解放処理を加えるのみである

5.5 PDUR サーバ方式と COM サーバ方式

本章では，通信ミドルウェアのマルチコア拡張の手法を 2 つ提案する．

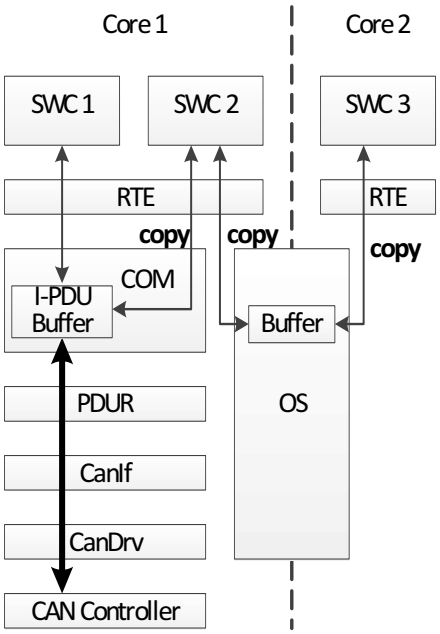
5.5.1 基本方針

AUTOSAR アプローチとジャイアントロックアプローチは，実行オーバヘッドとロック取得時間を両立できていない．また，共に実行並列性がないという問題もある．AUTOSAR アプローチで複数のコアが通信ミドルウェアを実行できないことは明白である．ジャイアントロックアプローチでは，ロックを取得してから通信ミドルウェアの処理を実行するため，複数のコアが同時に通信ミドルウェアの処理を行えない．このため，実行並列性がない．並列性を高めるためには，ロック取得時間を短くしつつ通信ミドルウェアの一部もしくは全てを両方のコアで動作させる必要がある．

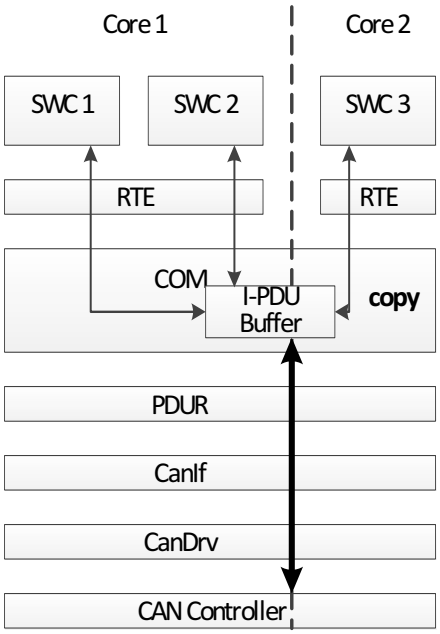
ここで，どちらの手法を基にし，改良するかを考える．

ジャイアントロックを改良する方法については，ロックの数を増やし，1 つのロックで排他制御を行う範囲（ロック単位と呼ぶ）を小さくする方法（細粒度ロックと呼ぶ）が考えられる．細粒度ロックでは，ロックの競合が発生しにくくなり並列性を高めやすい．しかし，細粒度ロックの実装は，通信ミドルウェアの実装方法に大きく依存し，設計及び実装が複雑である．AUTOSAR は仕様のみを定めているため，実装方法への依存の低い方が手法の汎用性および有用性が高くなると考え，本研究では，AUTOSAR アプローチを改良する方針をとった．

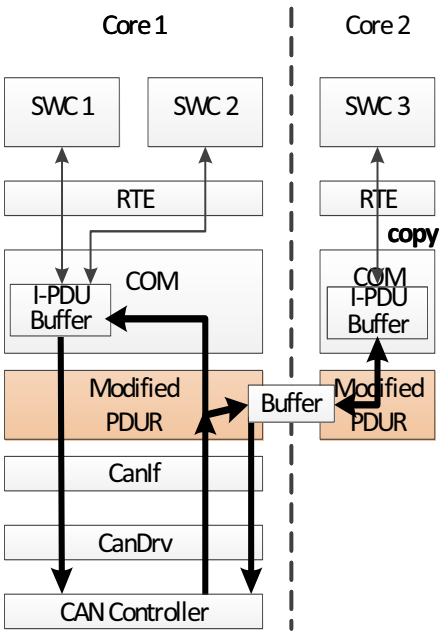
AUTOSAR アプローチを基に，改良するにあたっては，実装が容易な方法を検討する．そのためには，ソースコードの変更が少ないほうが良い．



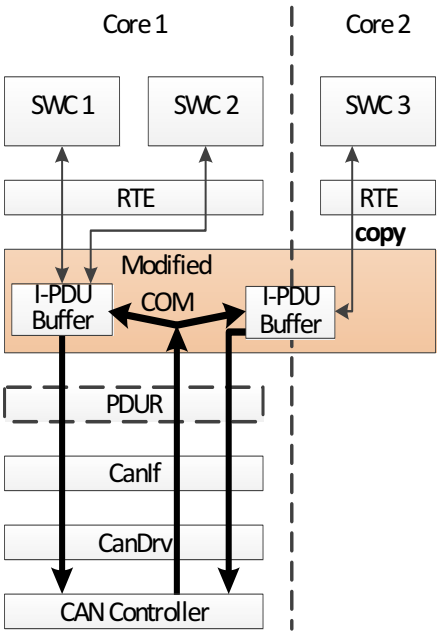
(a) AUTOSAR アプローチ



(b) ジャイアントロックアプローチ



(c) PDUR サーバ方式



(d) COM サーバ方式

図 5.3: マルチコア通信ミドルウェアのアーキテクチャ

5.5.2 PDUR サーバ方式

まず，ソースコードの変更をせずに実行並列性を高める方法を検討した．

各層の処理を変更せずにマルチコア拡張を行うには，いずれかの層の間でコア間通信を行う層を新たに追加する必要がある．1 つめの手法として，コア間通信は PDUR 層で行う PDUR サーバ方式を提案する．マルチコア拡張により追加される処理は，省略されていた PDUR 層ですべて実行する．コア間の通信のため，PDUR 内にバッファを持つ．PDUR サーバ方式のアーキテクチャを図 5.3(c) に示す．

コア 1 の SWC から別の ECU へデータを送信する場合は，従来通り PDUR 層を経由せずに COM 層から CanIf 層を直接呼び出す．受信の場合は，PDUR 層でどちらのコアの COM 層に受信データを渡すのか判別を行う必要がある．この判別処理が，コア 1 の SWC が受信する場合に実行オーバヘッドとして加わる．

コア 2 の ECU 間通信の手順について述べる．送信については，COM 層から PDUR 層を呼ぶと PDUR 層の内部でコア間通信を行い，コア 1 に対して残りの送信処理を依頼する．コア 1 は，コア間通信により送信データを取得し，CanIf 層に送信を要求する．COM 層までの処理と CanIf 層以降の処理は従来通りである．受信については，CanIf 層から PDUR 層を呼ぶと，PDUR 層は受信データの ID からコアの判別を行い，コア 1 の場合は従来通りの処理を行う．コア 2 の場合は，コア間通信を行いコア 2 へ受信データを渡す．そして，コア 2 は残りの受信処理を行うため，PDUR 層で受信データをバッファから取得し，COM 層へ渡す．CanIf 層までの処理と COM 層以降の処理は従来通りである．

PDUR サーバ方式において，コア 2 から定期送信する処理手順を図 5.4(a) に示す．まず，コア 2 の SWC が RTE 層を経由して COM 層を呼び出し，シグナルを COM 層に渡すと，COM 層は対応する I-PDU バッファを更新する．定期送信のため COM 層は I-PDU バッファの更新のみを行い，SWC に処理を戻す．その後，コア 2 で `Com_MainFunctionTx` が呼ばれると，送信周期になった I-PDU があるかどうかを探索する．送信周期になった I-PDU が見つかると，PDUR 層に I-PDU バッファへのポインタを渡す．PDUR 層の内部では，コア 2 からコア 1 へのコア間通信が行われると，I-PDU バッファのデータがコピーされ，コア 2 の処理は終了する．コア 1 がデータを受信すると，CanIf 層にデータを渡して送信を要求する．このように，PDUR サーバ方式では，I-PDU バッファとコア間通信用のバッファ間でデータをコピーする必要がある．コア 2 の SWC とコア 1 の通信ミドルウェアとの間では，シグナルと I-PDU でそれぞれ 1 回，計 2 回のコピーが必要となる．

要件との対応については以下の通りである．

要件 1 との対応　ロックを取得するのは，PDUR 層コア間通信を行う間のみであり，ロック取得時間は短い．

要件 2 との対応　割込み禁止は，コア間通信と通信ミドルウェアを実行する際に行う．シングルコアからの大きな変更はなく，割込み応答時間は短い．

要件 3 との対応　コア 2 の ECU 間通信では，PDUR 層でコア間通信を行う際に，データコピーが行われる．コア 2 の SWC とコア 1 の通信ミドルウェアとの間で，データのコピーが計 2 回行われる．コア 1 の ECU 間通信の実行オーバヘッドについて，送信はシングルコアと同じであるが，受信はコア判定の処理分だけ増加する．

要件 4 との対応　COM 層はそれぞれのコアで実行することができるため，AUTOSAR アプローチと比べて並列性が高い

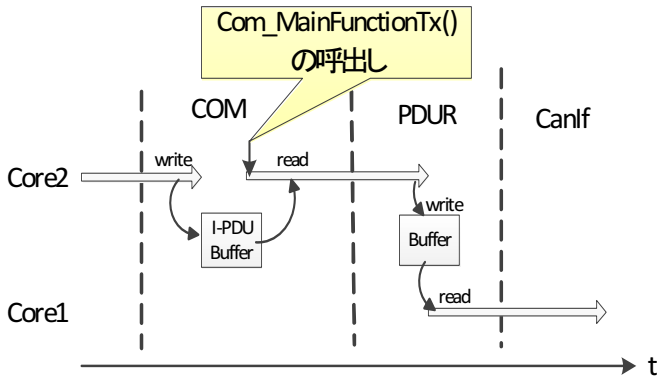
要件 5 との対応　COM 層をコアごとに持つため，メモリ使用量が大きい

要件 6 との対応　PDUR 層以外は変更する必要がなく，PDUR 層の処理も少ないのでソースコードの変更量は比較的少ない

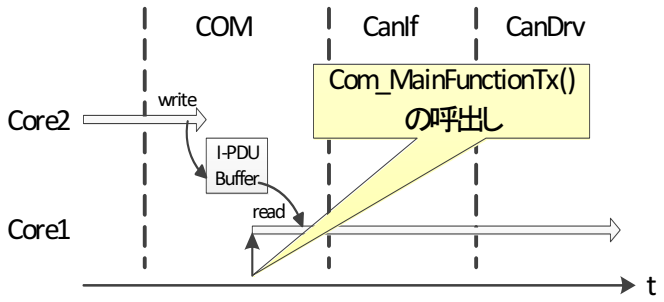
5.5.3　COM サーバ方式

AUTOSAR アプローチと PDUR サーバ方式で発生する，データコピーによる実行オーバヘッドを削減するため，I-PDU バッファに着目した．コア 2 の I-PDU バッファにコア 1 が直接アクセスすればデータコピーの回数を増やさずに実装することができる．この方式を COM サーバ方式と呼ぶ．アーキテクチャを図 5.3(d) に示す．コア 2 の ECU 間通信の処理を行うなかで，コア 2 の I-PDU バッファにアクセスする際にはロックが必要である．コア 1 の ECU 間通信では，コア 1 の I-PDU バッファに対するロックは不要である．これは，コア 1 の I-PDU バッファへはコア 2 からアクセスすることがないためである．

コア 2 の ECU 間通信の手順について述べる．送信については，ロックを取得してからシグナルを I-PDU バッファにパッキングし，コア 1 に対して送信を依頼する．コア 1 では，コア 2 の I-PDU バッファからデータを取り出して，CanIf 層に送信を要求する．受信については，COM 層の呼び出しまでは従来と同じである．COM 層



(a) PDUR サーバ方式



(b) COM サーバ方式

図 5.4: 定期送信の手順

は，受信データの ID からコアの判別を行い，コア 2 の場合にコア 2 の I-PDU バッファへ受信データをコピーする．そして，即時受信の場合はコア 2 へ受信処理を依頼する．定期受信の場合は，コア 2 が定期的に受信処理を行うため受信処理を依頼する必要がない．

COM サーバ方式で，コア 2 の定期送信する処理手順を図 5.4(b) に示す．コア 2 の COM 層を呼び出し，I-PDU バッファを更新する処理までは従来と同じである．その後，コア 1 で `Com_MainFunctionTx` が呼ばれ，送信周期になった I-PDU が見つかると，CanIf に I-PDU バッファへのポインタを渡し送信を要求する．図 5.4(a) と図 5.4(b) を比較すると，COM サーバ方式ではデータコピー回数が削減されていることがわかる．

COM サーバ方式ではコア間通信のために新たにバッファを作成せず，コア 2 の I-PDU バッファを共有する．そのため，コア 2 の I-PDU バッファに対して排他制

御が必要になる．送信処理では，コア 2 の I-PDU バッファにある I-PDU を CAN コントローラにコピーする必要があるため，コア 1 は COM 層から CanDrv 層の処理までを排他制御を行う必要があり，ロック取得時間が長くなってしまう．一方，受信処理では，I-PDU バッファへのアクセス時のみなので同様の問題は発生しない．

要件との対応は以下の通りである．

要件 1 との対応　受信ではロック取得時間は短いが，送信ではロックを取得したまま下位層を呼び出す必要があり，ロック取得時間が長い．

要件 2 との対応　割込み禁止は，シングルコアからの大きな変更はなく，割込み応答時間は短い

要件 3 との対応　コア 2 の ECU 間通信では，I-PDU バッファに対するロック取得と解放が実行オーバヘッドとなる．この実行オーバヘッドは小さい．コア 1 については，送信はシングルコアと同じであるが，受信時に COM 層でコア判定をする実行オーバヘッドがある．

要件 4 との対応　コア 2 では COM 層の一部を実行するため，AUTOSAR アプローチより並列性は高い．しかし，PDUR サーバ方式より並列性は低い．

要件 5 との対応　COM 層の処理をコア 1 とコア 2 で変更する分，メモリ使用量は増えるが，ソースコードの共有により増加分は少ない

要件 6 との対応　通信ミドルウェアの中で最も複雑な COM 層を変更する必要があり，ソースコードの変更量は多い

5.6　実装

各手法を AUTOSAR 仕様 Release 3.0.2 に従い作成された，シングルコアシステムで動作する通信ミドルウェアをベースに実装した．本節では，各方式を実装する上で課題となった点について述べる．

5.6.1　定期通信処理中の排他制御

定期通信の処理（`Com_MainFunctionTx/Rx` の処理）は，すべての I-PDU について設定された周期に達しているかチェックするため，実行時間が I-PDU の数に比

例して長くなる．ジャイアントロックアプローチと COM サーバ方式では，I-PDU バッファにアクセスする際には排他制御が必要である．定期通信処理の全てを，割込み禁止かつロック取得による排他状態のまま実行し続けると，割込み応答時間と他コアのロック取得待ち時間が長くなり，要件 1 と要件 2 を満たせない．これを解決するため，1 つの I-PDU をチェックする度にロックの解放と割込み禁止を解除することとした．したがって，ジャイアントロック方式と COM サーバ方式では，定期通信処理において，割込み禁止処理と，ロックの取得処理を I-PDU の数だけ行う．なお，文献 [17] では，定期通信の実装について言及がないため，ジャイアントロックアプローチでこのような実装が必要であることに触れられていない．

5.6.2 ソースコードの共有

PDUR サーバ方式では，COM 層を各コア上で動作させる必要がある．単純な実装方法として，コア 1 とコア 2 の COM 層のために，コアごとに異なるソースコードを作成することで，それぞれのコアで動作する COM 層を作成することができる．しかし，コア間で異なるのは，COM 層の管理情報や I-PDU バッファなどのデータだけであり，処理内容は全く同じである．ソースコードを別にしてしまうと，保守性が悪い．また，メモリ使用量については，COM 層の分だけ新たに必要となるため，大きく増えてしまう．コア間でソースコードを共有し，アクセスするデータをコアごとに変える方法としては，実行時にどのコアで実行しているかの情報を取得し，アクセスするデータを決める方法がある．この方法では，メモリ使用量は大きく増えないが，アクセスするデータを決める処理を追加するため，COM 層のソースコードの変更が必要である．

PDUR サーバ方式は COM 層のソースコードを変更する必要がないというメリットがあったため，実行時にアクセスするデータを決める方法を採用しないこととした．代わりに，ソースコードをコンパイルしたオブジェクトファイルをコア 1 とコア 2 用に用意することで，ソースコードを変更することなく共有する実装を行った．

オブジェクトファイルをコア 1 とコア 2 用に用意する，具体的な手順を述べる．本研究では GNU の開発環境を用いており，Binutils[2] を用いることで，オブジェクトファイル内のシンボル名を変更することができる．これを利用して，COM 層の全シンボルの名前を変更したオブジェクトファイルを生成する．これにより，通常通りコンパイルしたオブジェクトファイルと，変数や関数などの名前を変更したオブジェクトファイルの 2 つを生成できる．前者をコア 1 用，後者をコア 2 用と

することで，コア間でCOM層のソースコードを共有できる．ただし，この方法では，メモリ上に2つのCOM層を作成するため，メモリ使用量を減らすことはできない．

一方，COMサーバ方式では，コア1とコア2でCOM層の処理が異なる．また，コア1のCOM層はコア2のI-PDUバッファにアクセスする処理を加える必要があるため，既存のソースコードを変更することは避けられない．そのため，実行時にコア情報を取得して，どちらのコアであるかによって処理を分けることとした．この実装では，メモリ使用量の増分は追加した処理の分だけである．

5.7　評価

5.7.1　評価項目

即時通信と定期通信について計測を行う．それぞれ，送信と受信に分かれるため，計4つの通信パターンについて計測する．さらに，4つの通信パターンをコアごとに計測する．それぞれの計測を，AUTOSARアプローチ，ジャイアントロックアプローチ，PDURサーバ方式，COMサーバ方式について行う．それぞれの手法の計測結果を，autosar，glock，pdur，comと表記する．

それぞれの実行時間を計測し，実行オーバヘッド，ロック取得時間，並列実行性について比較を行う．また，メモリ使用量やコード変更量についても比較を行う．

5.7.2　評価方法

4つの通信パターンをコア毎に計測するため，8つのシグナルとI-PDUを定義し，1つのシグナルにつき1つのI-PDUを対応させた．サイズは，シグナルとI-PDUを共に32ビットとした．コアは，Altera NiosII 50MHzを2つ用意した．このコアは，4KBの命令キャッシュをもつが，データキャッシュはない．計測に使うタイマのクロックはCPU と同じ50MHzである．

計測範囲は通信ミドルウェアのみであり，別のコアに処理を依頼する際に通知する処理，具体的にはタスク起動もしくはイベントのセット処理などのOSが処理する時間は計測に含めない．ただし，ECU間通信を実現するためにコア間通信を行う場合には，そのコア間通信にかかる時間も含めて比較する必要があると考え，計測に含めることとした．

コアごとに 4 つの通信パターンの実行時間を 10000 回繰返し計測した．マルチコア拡張による実行オーバヘッドを評価するため，あるコアが通信をするとき，もう一方のコアがなにも処理を行わないようにした．それぞれの処理のうちロックを取得している時間を計測することで，ロック取得時間および並列実行性を評価する．また，コードの変更量は cloc[3] を使って計測した．

5.7.3 評価結果

実行オーバヘッド

コア 1 の計測結果を 図 5.5 に，コア 2 の計測結果を図 5.6 に示す．シングルコアの実行時間を基準にした実行オーバヘッドをまとめて表 5.1 に示す．コア 1 の即時通信については，どの方式も大きな違いはない．コア 1 の定期通信では，autosar は測定の誤差により小さな増減があると考えられる．glock では実行時間が増加した一方，pdur では減少した．そして，com では送信は増加し，受信は減少した．この原因は次の通りである．コアに限らず定期通信のときには，I-PDU バッファのチェックを行う `Com_MainFunctionTx/Rx` を呼び出すため，実行時間が I-PDU の数に比例する．glock は，I-PDU のチェックごとにロック取得と解放を繰り返すため，定期通信の実行時間が長くなった．一方，pdur と com では I-PDU バッファをコアごとに持つため，コア 1 における `Com_MainFunctionTx/Rx` の実行時間が glock の場合と比べて短くなった．ただし，com での送信は，`Com_MainFunctionTx` を呼び出すとコア 1 とコア 2 の I-PDU バッファをチェックする必要がある．そのため，com の定期送信の実行時間が長くなっていると考えられる．

コア 2 の即時通信では glock が最も良い結果である．autosar と pdur は 35%以上となる実行オーバヘッドが発生しており，データコピーの多さが主な要因である．一方，com では pdur より実行オーバヘッドが小さくなっており，これがデータコピー回数を低減した効果だと考えられる．コア 2 の定期通信では，pdur が I-PDU バッファを分割したことにより実行オーバヘッドを低減している．com では，送信の実行オーバヘッドが 14.9%と pdur より大きくなっている一方で，受信の実行オーバヘッドは 9.0%と他の方式と比べて小さい．

コア 1 とコア 2 の各通信にかかる実行オーバヘッドを単純に平均すると，小さいものから pdur，com，glock，autosar という順になった．つまり，既存手法よりも 2 つの提案手法が実行オーバヘッドが小さい．これは，定期通信における違いが大きい．即時通信の多いアプリケーションの場合は，pdur や com よりも glock の方

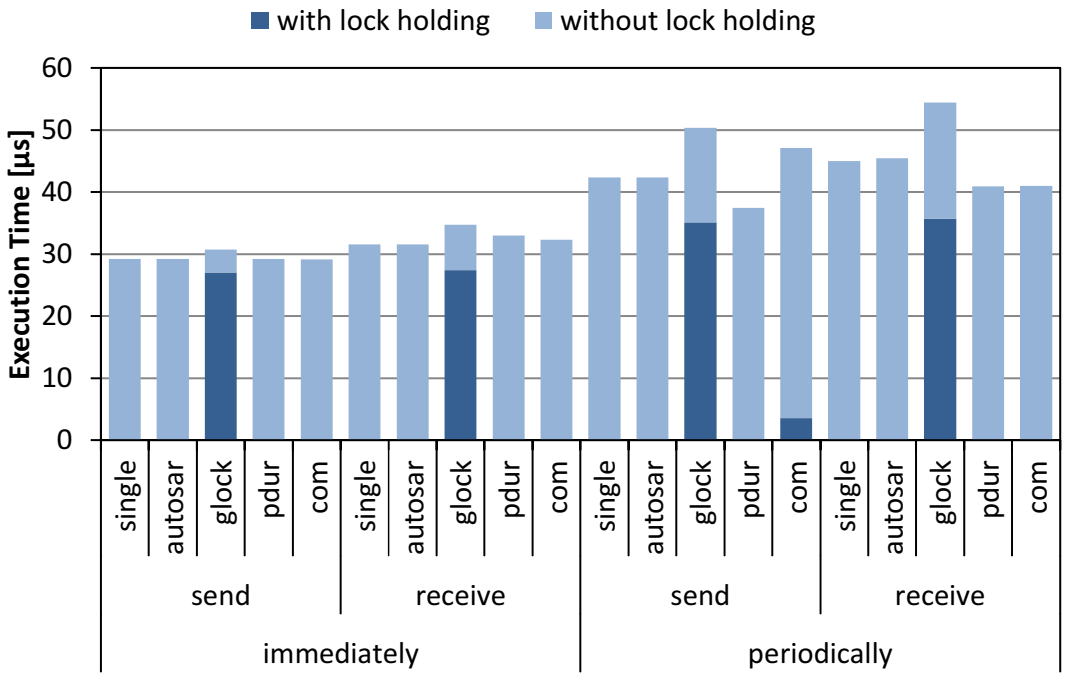


図 5.5: コア 1 の ECU 間通信にかかる実行時間

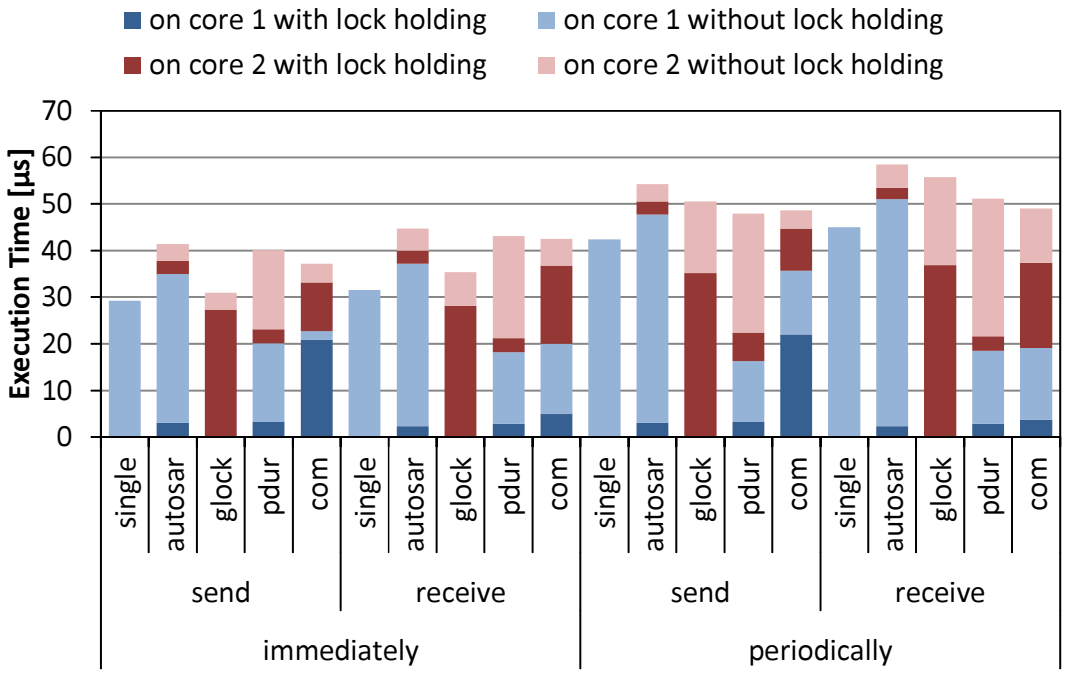


図 5.6: コア 2 の ECU 間通信にかかる実行時間

表 5.1: 実行オーバヘッド [%]

			autosar	glock	pdur	com
Core 1	immediately	send	0.0	5.2	0.0	-0.4
		receive	0.0	9.9	4.6	2.5
	periodically	send	-0.3	18.9	-11.5	11.3
		receive	0.9	20.9	-9.0	-8.9
Core 2	immediately	send	41.7	6.0	37.0	27.2
		receive	41.8	12.0	36.8	34.6
	periodically	send	27.9	19.3	13.2	14.9
		receive	29.9	23.9	13.5	9.0
Avg.			17.8	14.5	10.6	11.3

が適しているが，定期通信の多いアプリケーションの場合は，glock よりも pdur もしくは com を用いた方が適している．また，コア 2 の ECU 間通信がコア 1 の ECU 間通信より少ない場合は，autosar を用いることができる．

ロック取得時間

それぞれの通信処理にかかる時間のうち，ロックを取得して実行した最大実行時間を表 5.2 に示す．どの通信のときにロックを取得するかは図 5.5 と図 5.6 から読み取れる．コア 1 の ECU 間通信では，glock のとき，com の定期送信のときにロックを取得する．glock はロック単位が大きいため，最大ロック取得時間が長い．autosar と pdur は，コア間通信に用いるバッファアクセス時にだけロックを取得するため，最大ロック取得時間が短い．com は，送信時に I-PDU バッファをロックしたまま CanDrv を呼び出し，CanDrv にデータを渡す必要があるため，ロック取得時間は autosar や pdur より長くなった．一方で，com のロック取得時間は glock よりも短い．

並列実行性

通信にかかる処理のうち，通信するコア自身で実行し，他コアの影響を受けない処理の割合を表 5.3 に示す．この割合が高いほど，並列実行性が高いと考えられる．

表 5.2: 最大ロック取得時間 [μs]

	autosar	glock	pdur	com
Core1	3.1	27.1	3.2	20.9
Core2	2.8	27.3	3.2	13.7

表 5.3: ロックを取得せずに実行する処理の割合 [%]

			autosar	glock	pdur	com
Core 1	immediately	send	100.0	12.0	100.0	100.0
		receive	100.0	21.0	100.0	100.0
	periodically	send	100.0	30.3	100.0	92.5
		receive	100.0	34.4	100.0	100.0
Core 2	immediately	send	8.8	11.9	42.3	10.7
		receive	10.7	20.4	50.8	13.4
	periodically	send	6.7	30.5	53.2	8.2
		receive	8.7	34.0	57.8	23.8
Avg.			54.4	24.3	75.5	56.1

autosar と pdur では，コア 1 の ECU 間通信ではロックを取得しない．com では，定期送信でコア 2 の I-PDU バッファをチェックするため，コア 1 の ECU 間通信であってもロックを取得する．glock では通信にかかる処理のうちほとんどをロックを取得して実行するため，全般で割合が低くなり平均では 24.3%となった．各手法を平均で比較すると，割合が高いものから pdur，com，autosar，glock という順になった．

メモリ使用量

メモリ使用量をまとめて 表5.4 に示す．autosar はコア間通信のためのコードが，glock はロック取得のためのコードが増えた分だけ，メモリ使用量が増えている．pdur は COM 層をコアごとに持つため，他の手法よりメモリ使用量が多い．com でもは，COM 層をコアごとに持つが，可能な限りコードを共有するよう実装したため，pdur ほどメモリ使用量は多くない．

表 5.4: メモリ使用量 [Byte]

	text	data	bss	total	増率 [%]
single	40056	717	240	41013	–
autosar	40788	717	258	41763	1.8
glock	40264	717	246	41227	0.5
pdur	56896	1241	394	58531	42.7
com	44540	1237	390	46167	12.6

表 5.5: ソースコード変更量 [行]

	autosar	glock	pdur	com
modify	34	39	207	292
add	344	55	1383	1890
remove	10	26	295	367
total	388	120	1885	2549
既存のソースコードに対する割合 [%]	1.9	0.6	9.3	12.6

ソースコード変更量

マルチコア拡張によるソースコードの変更量を，表 5.5 に示す．autosar と glock は変更量が少ないが，pdur と com は変更量が多い．特に com では COM 内部を変更するため変更量が多い．

5.7.4 要件との対応

各手法を要件ごとに優劣をつけると表 5.6 のようになる．表 5.6 では 1 が最も優れていることを表している．要件 3 については，平均の実行オーバヘッドで比較を行った．これは，全ての通信で実行オーバヘッドが小さい手法が存在しなく，どの通信を重視するのかによって，各手法の優位性が変化するためである．なお，割込み応答時間については評価を行っていないので，要件 2 は空欄とした．

本研究で行った評価では，どの要件を重視するかによってどの手法が優れているかが変化する結果となった．

表 5.6: 各手法の要件ごとの優劣

	autosar	glock	pdur	com
要件 1	1	4	2	3
要件 2	–	–	–	–
要件 3	4	3	1	2
要件 4	3	4	1	2
要件 5	2	1	4	3
要件 6	2	1	3	4

5.8　関連研究

OS をマルチコア拡張することに関しては様々な研究が行われている．Linux や BSD など，オープンとなっているソースコードに基づいた改良が行われている．これらの OS では，当初，Big Kernel Lock と呼ばれる手法がとられていた．これはカーネルの排他制御を 1 つのロックのみで行う方式で，性能低下の原因となる．段階的に BKL が解消されていき，性能が向上した．TCP/IP プロトコルのマルチコアに関する研究も盛んに行われている．具体的には，IP パケットをフィルタリングする L7-filter[10] の性能向上や, カーネルのネットワーク割込み処理のスケーラビリティ向上のための RFS[12]，RPS[11] などがある．また，一連の処理を特定のコアへ割当てることで性能向上を図る MC-ORB[26] や A-TFN[25]，といった研究も存在する．これらの研究は，データの局所性の観点から特定のコアで処理を行うことでキャッシュのヒット率を向上させることで性能向上を図っている．しかし，車載システムではリアルタイム性の保証が困難であることからキャッシュを搭載しない場合があり，これらの手法が使えない．

AUTOSAR 通信ミドルウェアのように，仕様のみが規定され，実装がオープンとなっていないものを改良する際には，実装に依存した改良を行うよりも実装に依存しない汎用的なマルチコア拡張手法が望ましい．

AUTOSAR をマルチコアシステムで利用するために，様々な研究が行われている [20, 21]．AUTOSAR 通信ミドルウェアをマルチコア拡張した研究も存在する．文献 [17] では，ジャイアントロック方式が提案されている．これは，1 つのロックで通信ミドルウェアの全てを排他するロック方式である．Linux でもマルチコア拡張の初期にはジャイアントロック方式を採用し，段階的にロック粒度を細かくして

いった．文献 [17] では AUTOSAR のアプローチと比較して性能が向上したと述べている．しかし，最悪実行時間などのリアルタイム性に関する議論はしていない．

5.9　まとめ

AUTOSAR 仕様の通信ミドルウェアをマルチコアシステムで動作させる既存の手法では，実行オーバヘッドとロック取得時間を両立することができず，実行並列性がない．本研究では，このような問題を含めた 6 つの要件を，通信ミドルウェアのマルチコア拡張に求められる要件として定めた．そして，既存手法の問題の解決を目指した手法を 2 つ提案し，実装および比較評価を行った．評価の結果，どの手法も一長一短であり，どの要件を重視するかによって適切な手法が変化することが明らかになった．また，ジャイアントロックアプローチについては，定期通信を考慮しておらず，割込み応答時間やロック取得時間などのリアルタイム性を損なわないように実装を行うと，比較的大きな実行オーバヘッドとなることを示した．

今後の課題として，要件を全て満たす手法の検討が挙げられる．例えば，本研究で示した手法を組み合わせることで，より優れた手法となる可能性がある．また，コア数が 3 つ以上に増えたときの評価，より複雑なアプリケーションを用いた評価が今後の課題である．

第6章　結論

6.1　まとめ

本論文では，組込みリアルタイムシステムに適したマルチコアプラットフォームを構築するにあたって，以下の3つの提案を行った．

1つ目は，マルチコアプラットフォームがコア間の共有リソースを排他する際に用いる排他制御アルゴリズムとして，中断可能な優先度継承キューイングスピンロックアルゴリズム（PPIQL アルゴリズム）を提案した．本研究では，まず，コア数に対する実行オーバヘッドのスケーラビリティとリアルタイム性を両立するためにスピンロックが満たすべき要件を示した．そして，2個のロックを取得する場合に，既存のアルゴリズムである Totally FIFO アプローチは全ての要件を満たせない問題があることを示し，Totally FIFO アプローチを拡張することで，全ての要件を満たす PPIQL アルゴリズムを提案した．さらに，提案アルゴリズムをハードウェア実装する方法を述べた．実機による評価により，提案アルゴリズムの有用性と，ハードウェア実装によって最大 3.11 倍高速化することを示した．

2つ目は，マルチコアプラットフォームのテスト効率化手法の提案である．マルチコアプラットフォームでは，各コアの実行順序に依存してパスが定まる分岐（実行順序依存分岐）が存在する．この分岐は，各コアが並列に実行することによって発生する不都合な状態に対応するために，ソフトウェアプラットフォームの開発者が意図的に作成したものである．プログラムを繰り返し実行する連続試行手法によりその分岐を網羅することは，テストの実施や不具合の分析に多くの手間を要するため，テストの効率が悪いという問題がある．本研究では，テストプログラムからコアの実行を制御することで，プログラム中の特定のパスを決定的に実行する機構を用い，テスト効率化手法を提案した．具体的には，各コアの実行順序や割込みの発生順序をテストプログラムにより制御可能な機構をシミュレータ上で実装し，マルチコア向け RTOS である FMP カーネルの実行順序依存分岐を網羅できることを確認した．連続試行手法により，実行順序依存分岐を網羅することも試みたが，特定のパスについては，100 万回試行し，さらに 6 時間繰り返しても実行され

なかった．提案手法を用いると，実行順序依存分岐を網羅する 83 件のテストプログラムの実行時間は 10.69 秒であった．また，提案手法を用いることで，FMP カーネル内の wait_tmout 関数について，1 件の不具合を検出した．wait_tmout 関数は，タイマ割込みによって呼び出される関数である．wait_tmout 関数内にはデッドロック回避処理を行うパスが存在しており，このパスが実行順序依存パスとなっている．この実行順序依存パスを実行するには，そのコアでタイマ割込みの処理を実行中に，他のコアから特定の API を実行する必要がある．つまり，この不具合は，各コアの実行順序だけでなく割込みの発生順序にも依存するパスを実行したときに発生する不具合であった．このパスは，プログラムを繰り返し実行する手法では 6 時間繰り返しても一度も実行されなかったため，提案手法を用いなければ不具合の発見は困難であったと考えられる．

3 つ目は，リアルタイム性と実行オーバヘッドの両立を目指した，AUTOSAR の通信ミドルウェアのマルチコア拡張手法の提案である．本研究では，まず，通信ミドルウェアをマルチコア拡張する際に求められる要件を示し，既存手法の問題点を明らかにした．具体的には，実行オーバヘッドとロック取得時間を両立することができず，さらに，実行並列性がないことが問題であった．本研究では，実行並列性を高めた上で実行オーバヘッドとロック取得時間を両立することを目指し 2 つの手法を提案した．1 つは PDUR サーバ方式で，もう 1 つは COM サーバ方式である．既存手法を含めて実装を行い，比較評価を行った結果，すべての要件を満たす手法はなかった．例として，既存手法であるジャイアントロックアプローチと COM サーバ方式を比較すると，即時通信ではジャイアントロックアプローチの方が実行オーバヘッドは低いが，定期通信では COM サーバ方式の方が実行オーバヘッドは低くなる．なお，ジャイアントロックアプローチを提案する文献 [17] では，定期通信についての実装および評価を行っておらず，定期通信において実行オーバヘッドに課題があることは，本研究によって明らかになった．現状では，通信ミドルウェアをマルチコアシステム上で使うためには，アプリケーションの行う通信について，即時通信が多い場合はジャイアントロックアプローチを，定期通信が多い場合は COM サーバ方式を用いる，といった使い分けが必要である．実行オーバヘッド以外の要件もあるため，システム全体としてどの要件を重視するかを判断した上で，適切な手法を選択する必要がある．

本論文では，排他制御に着目し，排他制御アルゴリズムと排他制御の使い方の 2 点において，リアルタイム性と実行オーバヘッドを両立する手法を提案した．また，信頼性を向上させる取り組みとしてソフトウェアテストの効率化手法を提案

した．以上の内容により，マルチコアプラットフォームの構築に関する課題を解決する技術を示した．これらの成果が，組込みリアルタイムシステムを対象としたマルチコアプラットフォームを構築する技術の発展につながることを期待する．

6.2　今後の課題

今後の課題として，まず，3 章で述べた排他制御アルゴリズムについては，実際のマルチコアプラットフォームに適用することがあげられる．FMP カーネルが適用候補であるが，FMP カーネルでは 3 つのロックを同時に取得する場合があるため，この場合には提案手法をそのまま適用できない．このため，3 つのネストロックについても提案手法の拡張により対応できるかを検討する必要がある．

4 章で述べたテスト効率化手法については，FMP カーネル以外のプラットフォームやアプリケーションの分岐を網羅するために利用することが挙げられる．また，提案手法を用いるためには，実行順序依存パスを実行するようにプログラムを作成する必要があるが，これを作成するにはハードウェアとソフトウェアの両方について一定の知識を要する．実行順序依存パスを実行するプログラム作成をサポートする手法が必要である．

5 章で述べた通信ミドルウェアのマルチコア拡張については，本研究で示した要件を全て満たす手法を検討する必要がある．また，排他制御については TAS スピンロックで実現しており，コア数が増加したときに，通信ミドルウェアの提供するサービスの最悪実行時間が定まらない．そのため，3 章で提案した排他制御アルゴリズムを適用する必要があると考えられる．また，マルチコア拡張によって不具合が発生しないことを 4 章で述べたテスト効率化手法を用いて検証する必要がある．

謝辞

本論文に関する研究活動を通じて，御指導と御助言を頂きました名古屋大学大学院情報科学研究科情報システム学専攻の高田広章教授，同研究化附属組込みシステム研究センターの本田晋也准教授に，心から感謝致します．

本研究を進めるに当たり，日々御指導，御討論頂いた名古屋大学大学院情報科学研究科情報システム学専攻高田研究室，枝廣研究室の皆様と，附属組込みシステム研究センターの皆様に感謝致します．

本論文に関する研究は，一部，科学研究費補助金（21700026）による助成を頂きました．御礼申し上げます．

参考文献

[1] AUTOSAR. http://www.autosar.org/.

[2] Binutils. http://www.gnu.org/software/binutils/.

[3] Cloc - Count Lines of Code. http://cloc.sourceforge.net/.

[4] OSEK/VDX. http://www.osek-vdx.org/.

[5] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient System-Enforced Deterministic Parallelism. Technical report, 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10), Oct. 2010.

[6] Björn B. Brandenburg and James H. Anderson. An Implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP Real-Time Synchronization Protocols in LITMUS^{RT}. In *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 185–194, 2008.

[7] Alessio Carlini and Giorgio C. Buttazzo. An efficient time representation for real-time embedded systems. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pp. 705–712, 2003.

[8] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, Vol. 41, No. 1, pp. 111–125, 2002.

[9] "International Organization for Standardiztion". Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling. ISO 11898-1, 2003.

[10] Danhua Guo, Guangdeng Liao, Laxmi N. Bhuyan, Bin Liu, and Jianxun Jason Ding. A scalable multithreaded L7-filter design for multi-core servers. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pp. 60–68, New York, NY, USA, 2008. ACM.

[11] Tom Herbert. rps: receive packet steering. `http://lwn.net/Articles/361440/`, Sep 2009.

[12] Tom Herbert. rfs: receive flow steering. `http://lwn.net/Articles/381955/`, Sep 2010.

[13] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors. *Real-Time Systems Symposium, IEEE International*, Vol. 0, pp. 469–478, 2009.

[14] E. P. Markatos. Multiprocessor Synchronization Primitive with Priorities. *Proc, IEEE Workshop Real-Time Operating Systems and Software*, 1991.

[15] A. Mayer, H. Siebert, and K.D. McDonald-Maier. Debug support, calibration and emulation for multiple processor and powertrain control SoCs. In Proceedings of *Design, Automation and Test in Europe, 2005.* , Vol. 3, pp. 148–152, Mar. 2005.

[16] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, Vol. 9, No. 1, pp. 21–65, 1991.

[17] Daniel Lohmann Niko Böhm and Wolfgang Schröder-Preikschat. A Comparison of Pragmatic Multi-Core Adaptations of the AUTOSAR System. In *7th annual Workshop on Operating System Platforms for Embedded Real-Time Applications*, Jul. 2011.

[18] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th IEEE International Real-Time Systems Symposium*, pp. 259–269, 1988.

[19] Bilge E. Saglam and Vincent J. Mooney III. System-on-a-Chip Processor Synchronization Support in Hardware. *Design, Automation and Test in Europe Conference and Exhibition*, pp. 633–641, 2001.

[20] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst. System Level Performance Analysis for Real-Time Automotive Multicore and Network Architectures. IEEE Transactions on *Computer-Aided Design of Integrated Circuits and Systems*, Vol. 28, No. 7, pp. 979–992, Jul. 2009.

[21] J. Schneider, M. Bohn, and R. Rößger. Migration of Automotive Real-Time Software to Multicore Systems: First Steps towards an Automated Solution. In *Proceedings Work-In-Progress Session of the 22th Euromicro Conference on Real-Time Systems*, ECRTS'10, pp. 37–40, Jul. 2010.

[22] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, Vol. 39, pp. 1175–1185, 1990.

[23] H. Takada and K. Sakamura. Real-Time Scalability of Nested Spin Locks. *International Workshop on Real-Time Computing Systems and Applications*, pp. 160–167, 1995.

[24] TOPPERS プロジェクト. `http://www.toppers.jp/`.

[25] Wenji Wu, P. Demar, and M. Crawford. A Transport-Friendly NIC for Multicore/Multiprocessor Systems. *Parallel and Distributed Systems, IEEE Transactions on*, Vol. 23, No. 4, pp. 607–615, Apr. 2012.

[26] Yuanfang Zhang, C. Gill, and Chenyang Lu. Real-time Performance and Middleware for Multiprocessor and Multicore Linux Platforms. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pp. 437–446, Aug. 2009.

[27] 金ハンソル, 浅見侑太, 阿部真也, 金スンヨプ, 金榮柱, 金賢敏, 竹谷美里, 木村貴寿, 鴫原一人, 森孝夫, 山本雅基, 本田晋也, 高田広章. 組込みリアルタイムOS向けテストツールのマルチプロセッサ拡張. ソフトウェアテストシンポジウム 2011 東京, pp. 96–103, 2011.

[28] 安積卓也, 古川貴士, 相庭裕史, 柴田誠也, 本田晋也, 冨山宏之, 高田広章. オープンソース組込みシステム向けシミュレータのマルチプロセッサ拡張. コンピュータソフトウェア, 第 27 巻, pp. 24–42, Nov. 2010.

[29] 鴫原一人, 松浦光洋, 金ハンソル, 金スンヨプ, 馬鋭, 廉正烈, 金榮柱, 木村貴寿, 眞弓友宏, 本田晋也, 山本雅基, 高田広章. 組込みリアルタイム OS の API テストの実施. ソフトウェアテストシンポジウム 2010 東京, pp. 46–53, 2010.

[30] 鴫原一人, 一場利幸, 本田晋也, 高田広章. μITRON ベースのマルチプロセッサ向け RTOS のテスト. 情報処理学会論文誌, Vol. 53, No. 12, pp. 2682–2701, Dec. 2012.

[31] 王才棟, 高田広章, 坂村健. 優先度継承スピンロックアルゴリズムとその評価. 情報処理学会論文誌, Vol. 38, No. 11, pp. 2262–2273, 11 1997.

[32] 高田広章, 坂村健. 中断可能なキューイングスピンロックアルゴリズム. 電子情報通信学会論文誌 D-I, Vol. J78-D-I, No. 8, pp. 661–669, 1995.

[33] 高田広章, 坂村健. マルチプロセッサリアルタイムカーネルのスケーラビリティを重視した実現. 電子情報通信学会技術研究報告. CPSY, コンピュータシステム, Vol. 95, No. 603, pp. 1–6, 1996.

[34] 本田晋也, 高田広章. ITRON 仕様 OS の機能分散マルチプロセッサ拡張. 電子情報通信学会論文誌 D, Vol. 91, No. 4, pp. 934–944, Apr. 2008.

[35] 石田利永子, 本田晋也, 高田広章, 福井昭也, 小川敏行, 田原康宏. TOPPERS/FMP カーネル : リアルタイム性と高スループットを実現可能な組込みシステム向けマルチプロセッサ用 RTOS. コンピュータ ソフトウェア, Vol. 29, No. 4, pp. 219–243, 2012.

研究業績

学術論文

- 一場利幸，松原豊，本田晋也，高田広章，“中断可能な優先度継承キューイングスピンロックとそのハードウェア実装”，情報処理学会論文誌 コンピューティングシステム, Vol. 4, No. 3, pp. 133–146, May. 2011.
- 一場利幸，森孝夫，高瀬英希，鴫原一人，本田晋也，高田広章，“命令セットシミュレータの実行制御機構を用いたマルチプロセッサ RTOS のテスト効率化手法”，電子情報通信学会, Vol. J95-D, No. 3, Mar. 2012.
- 石川拓也，安積卓也，一場利幸，柴田誠也，本田晋也，高田広章，“TECS 仕様に基づいた NXT 用ソフトウェアプラットフォームの開発”, コンピュータソフトウェア, Vol. 28, No. 4, pp. 158–174, Nov. 2011.
- 石田利永子，山本雅基，海上智昭，森孝夫，本田晋也，一場利幸，高瀬英希，高田広章，“共同研究と公開講座による組込みソフトウェア技術者育成の取り組み”，日本工学教育協会 工学教育，Vol. 60, No 3, pp. 75–81, Jun. 2012.
- 鴫原一人，一場利幸，本田晋也，高田広章，“μITRON ベースのマルチプロセッサ向け RTOS のテスト”，情報処理学会論文誌，Vol. 53, No. 12, pp.2682–2701, Dec. 2012.

国際会議論文（査読あり）

- Toshiyuki Ichiba, Daniel Sangorrin, Shinya Honda and Hiroaki Takada, ”Comparison of Multicore Communication Stack for Automotive System,” The 16th IASTED International Conference on Software Engineering and Applications (SEA 2012), pp. 237–246, Nov. 2012.

国内会議発表論文（査読あり）

- 一場利幸，松原豊，本田晋也，高田広章，“中断可能な優先度継承キューイングスピンロックのハードウェア実装と評価”, 組込みシステムシンポジウム 2010 （ESS2010）, pp. 55–64, Oct. 2010.

研究会発表論文（査読なし）

- 一場利幸，高田広章，本田晋也，倉地亮，“AUTOSAR 通信ミドルウェアのマルチコア拡張”, 情報処理学会研究報告 第 14 回組込みシステム研究会, Jul. 2009.
- 一場利幸，松原豊，本田晋也，高田広章，“中断可能なキューイングスピンロックハードウェアの実装と評価”, 情報処理学会研究報告 組込み技術とネットワークに関するワークショップ (ETNET2010), Mar. 2010.
- 一場利幸，高瀬英希，鴫原一人，本田晋也，高田広章，“マルチプロセッ サ環境におけるタイミング依存のシナリオを実行可能なシミュレーション機構”, 組込み技術とネットワークに関するワークショップ ETNET2011, Mar. 2011.
- 石川拓也，安積卓也，一場利幸，柴田誠也，高田広章，“UML モデルの C 言語実装における TECS の適用事例”, 情報処理学会研究報告 第 78 回プログラミング研究会, Mar. 2010.
- 加藤寿和，一場利幸，本田晋也，高田広章，“ハードウェアの振舞いを考慮したスピンロックのモデル検査”, 情報処理学会研究報告 第 172 回ソフトウェア工学・第 21 回組込みシステム 合同研究発表会, May 2011.
- 太田貴也，Daniel Sangorrin，一場利幸，本田晋也，高田広章，“組込み向け高信頼デュアル OS モニタのマルチコアアーキテクチャへの適用”, 情報処理学会研究報告 第 117 回 OS 研究会, Apr. 2011.

受賞等

- 組込みシステムシンポジウム 2010 研究論文奨励賞，Oct. 2010.
- 平成 22 年度 情報処理学会 東海支部 学生論文奨励賞，Mar. 2011.
- 情報処理学会コンピュータサイエンス領域奨励賞, Jun. 2011.