# 車載制御マルチコアシステム
# におけるマッピングと
# ランタイムコード生成技術

小川　真彩高

# 車載制御マルチコアシステムにおけるマッピング とランタイムコード生成技術

## 要旨

近年，自動車制御に用いられるアプリケーションは，排ガス規制や低燃費化，そして安全性への要求の高まりといった要件から非常に複雑化，高度化してきている．中でもパワートレインの制御に用いられるアプリケーション（パワトレアプリ）は，その傾向が強い．従来のパワトレアプリはシングルコアアーキテクチャ上で動作していたが，耐熱性，耐ノイズ性の観点からシングルコアの動作周波数は限界に達しており，上述の要件を満たすためにはパワトレアプリにマルチコアアーキテクチャを用いることが必要となってきている．そのため，従来のシングルコアアーキテクチャ向けのパワトレアプリをマルチコアアーキテクチャに対応させる手法が必要である．

既存のマルチコアアーキテクチャ上のパワトレアプリの開発フローでは，まず使用するアーキテクチャを決定し，次に処理間のデータ通信の方法やタスク間の依存関係を成立させるための同期方法を決定するためにランタイム構造の設計を行う．そしてシングルコアアーキテクチャ向けにすでに作成されているアプリを複数のコア上で分散実行するために手動でアプリを分割し，分割した機能のコアへの配置を決定する．その後，実機上で実行できるようにランタイム実装を行い，実機による動作検証を行う．動作検証の結果，デッドラインミスのような問題が発生する場合には，その問題の原因を究明し，コアマッピングにフィードバックを行う．このようにして，最終的な実装を得る．

しかし既存の開発フローには，いくつかの課題が存在する．まず，コアの異なる処理間の実行依存関係を成立させるには同期が必要となるため，コア間同期による性能の悪化や制御の複雑化が課題となる．また，マルチコアアーキテクチャはコア数やメモリ構成が多様であり，アーキテクチャごとにこのような同期を実装することは非常に開発コストがかかる．さらに，マッピングの変更や機能の追加がある場合にランタイムを再実装する必要があることも課題となる．次に，安全性への影響が高いタスクのデッドラインミスが起きない処理のマッピングを探索する必要があることが挙げられる．ここで並列性を出すために細かい粒度でタスク分割するとマッピングのパターンが膨大なものとなることも課題となる．最後に，所属するコアの異なるタスク間通信では，他のタスクの実行状況やコア間で共有

するリソースへのアクセスのために必要な排他制御によって通信のタイミングが変動するため，動作検証を行うことが困難である．検証容易とするため，通信のタイミングが決定的となるようなランタイム構造が必要となる．

本論文では，これらの課題を解決できるような開発フローを提案する．提案する開発フローでは，パワトレアプリと実装するマルチコアアーキテクチャをモデル化し，それらに加えて処理のマッピング情報を用いてランタイムを自動生成する．マッピング情報としては最悪応答時間解析によってデッドラインミスが生じないように保証されたものを使用する．さらに，生成されるランタイムでは，通信のタイミングを決定的とするために，あらかじめ決められたタイミング (LET 区間) で通信を行う Logical Execution Time (LET) を用いて実装される．

提案する開発フローを実現するために本論文では 3 つの研究を行った．1 つ目は，Powertrain Multicore Programing Framework （PMPF）の提案である．PMPF では，パワトレアプリとハードウェアのアーキテクチャをモデル化し，それに加えて処理やデータのマッピング情報を入力として，ランタイムコードを自動生成する．これにより様々なマルチコアアーキテクチャへの対応を容易にすることが可能となる．PMPF では，タスクの途中で同期が必要ないように処理をタスクに分割し，タスク間の実行依存関係はタスク起動によって実現することで同期を不要としている．

2 つ目は，最悪余裕時間の静的解析の結果を用いて処理のコアマッピングの候補を選定する手法の提案である．まず，解析の候補を限定するために処理の集合を実行依存関係や排他アクセスが必要なデータ数によって PF グループという単位に分割する．そして PF グループ単位でのコアマッピングに対して最悪余裕時間解析を行う．最悪余裕時間が 0 以上であれば，デッドライン内で処理が完了することを保証することができる．最悪余裕時間の解析には既存手法の MIF をマルチコア向けに拡張したものを用いる．解析によって得られた最悪余裕時間が長い処理のコアマッピングを，実機に実装するコアマッピングとして選定する．

3 つ目は，LET をパワトレアプリに適用する手法の提案である．LET を実現するための既存手法では，処理の負荷を柔軟に変更することを可能とするサブスケジューリングやエンジンが高回転の場合に生じる安全性への影響が低いタスクのデッドラインミスに対応していない．そこで本研究では，サブスケジューリング下での適切な LET 区間の設定方法とデッドラインミスによって不整合なデータが生じないようなランタイム構造を提案する．また，既存手法では LET の通信を実現するための処理は実行オーバヘッドが大きいため，そのような処理を複数のコア

に分散し，効率的に実装する手法を提案する．

以上3つの研究により，前述した課題に対応した開発フローを実現することが可能となった．

# 目 次

# 図 目 次

# 表 目 次

# 第1章　序論

## 1.1　研究の背景

近年，世界的な環境保全への意識の高まりから，自動車にも排ガス規制への対応や低燃費化が求められるようになってきた．そのような背景から，欧州を中心に従来のガソリンエンジンやディーゼルエンジンから Electronic Vehicle (EV) への移行が進められてきているが [1]，充電インフラの普及，バッテリ容量およびバッテリの充電速度に問題があるため [2]，EV への完全な移行には時間がかかる．

排ガス規制，低燃費化，そして，安全性への要求の高まりといった要件により，パワートレインを制御するためのアプリケーション（パワトレアプリ）は高度化，複雑化してきている．従来のパワトレアプリはシングルコアプロセッサ上で動作していたが，車載マイコンのシングルコアプロセッサの動作周波数は耐熱性，耐ノイズ性の観点から限界に達しているため，より高度な機能を実現していくためにパワトレアプリをマルチコアアーキテクチャ上で動作させることが必要となってきている．しかし，パワトレアプリはソフトウェア規模が非常に大きく，現在 1000 個程度の依存関係を持つ処理が複雑に相互作用して動作しているため [3][4]，マルチコアアーキテクチャ向けにフルスクラッチで作り直すのは現実的ではない．そのため，従来のシングルコアプロセッサ向けのパワトレアプリを制御の処理や構成を変えずにマルチコアアーキテクチャに対応させる手法が必要である．ここで，すでにシングルコアプロセッサ向けのパワートレインの制御アプリが存在するという前提で，マルチコア上の制御アプリの開発について考える．マルチコアアーキテクチャ上へのパワトレアプリの既存の開発フローを図 1.1 の A に示す．

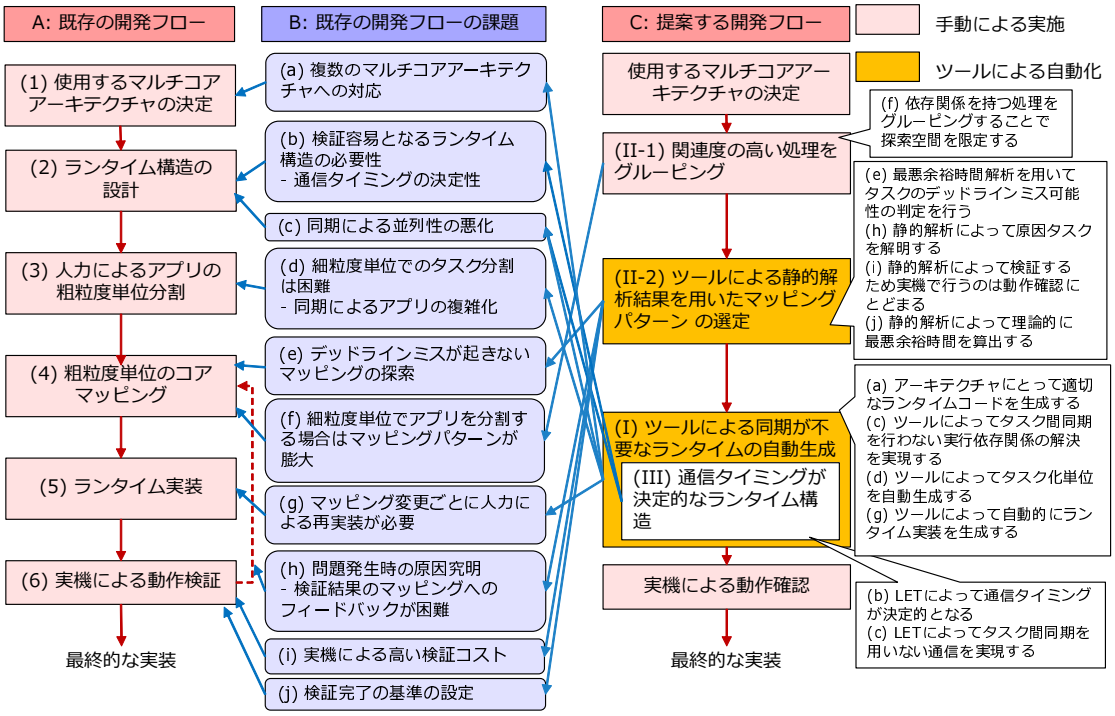


図 1.1: 既存および提案するパワトレアプリのマルチコア開発フロー

まず，(1) では，使用するマルチコアアーキテクチャを決定する．マルチコアアーキテクチャはバリエーションが多いため，その中から目的にあった構成をもつものを選択する．(2) では，処理間のデータ通信の方法やタスク間の依存関係を成立させるための同期方法を決定するためにランタイム構造の設計を行う．(3) では，シングルコアプロセッサ向けにすでに作成されているアプリを，複数のコア上で分散実行するために手動で分割する．(4) では，(3) で分割されたアプリのコアマッピングを実施する．(5) では，実機上で動作が可能となるように (4) で決められたコアマッピングに従ってランタイム実装を行う．(6) では，実機実行によって動作検証を行い，アプリが想定通りの動作をし，またタスクのデッドラインミスのような問題が発生するかどうかを検証する．検証の結果，問題が発生する場合にはその問題箇所を特定し，(4) のコアマッピングにフィードバックする．このように (4) から (6) のイテレーションによって，最終的な実装を得る．

このような既存の開発フローにはいくつかの課題（図 1.1 の B を参照）が存在する．(1) では，(a) 複数のマルチコアアーキテクチャに対応することが挙げられる．パワトレアプリを動作させる Electronic Control Unit （ECU）やセンサは，搭載する車種によって様々である．パワトレアプリは大規模であることに加え，高価

な HILS や実際のエンジンによる動作検証が必要なため開発コストが高く，単一のソースコードを複数の ECU で再利用できるような仕組みが必要である．ここで，マルチコアアーキテクチャはシングルコアアーキテクチャと比べ，コア数やメモリ構成が多様であり，アーキテクチャによって適切なタスク構成が異なってくる．さらにシングルコア上で動作していた処理の実行依存関係を保ったままマルチコアに移行すると，コア間を跨ぐ実行依存関係を成立させるためにコア間同期を用いる必要があり．制御が複雑となる．したがって手動による複数マルチコアプラットフォームへの対応は非常に困難であると言える．

(2) では，まず (b) 検証容易となるランタイム構造が必要であるということが挙げられる．属するコアの異なるタスク間の通信は，それぞれ通信のタイミングが自身より優先度の高いタスクの実行状況や実行パス，スピンロックのような排他制御，バスアクセス競合によって変動するため，通信タイミングの検証を行うことが難しい．これにより，通信タイミングが決定的であるようなランタイム構造が求められる．通信のタイミングを決定するために，通信を行うタスク間の実行順序を同期によって決定するというということも可能だが，その場合は (c) 並列性が悪化し，マルチコアを用いるメリットが小さくなるという問題がある．

(3) では，(d) 細粒度単位でのタスク分割が困難であるということが挙げられる．これは，シングルコアプロセッサ上で決定されていた処理の実行順序を守るためには同期が必要となり，同期によってアプリが複雑化し，実行時間の予測も困難となるためである．そのため，既存の開発フローでは大まかな機能単位での分割を行っている．

(4) では，(e) デッドラインミスが起きないマッピングの探索が必要であるということが挙げられる．パワトレアプリは安全性への影響度が高いハードリアルタイムな処理を多く含むため，これらのデッドラインミスが生じないようにしなければならない．しかし，実機による検証の場合，極稀に生じるタスクの実行パターンを見逃す可能性があるため，必ずデッドラインミスが生じないということを保証することが難しい．また，細粒度単位でアプリが分割される場合には (f) マッピングのパターン数が膨大となるという問題もある．

(5) では，(g) マッピング変更や機能の追加が行われる度にランタイムを再実装する必要があるということが挙げられる．特に同期処理の変更が必要となるためマッピング変更のコストが大きいという問題がある

(6) では，(i) 実機による検証は高価な HILS や実際のエンジンが必要なためにコストが高く，また実機検証では必ず問題が発生しないという確証を得るのが難し

いため，(j) 検証完了の基準の設定が必要であるということが挙げられる．さらに，(h) デッドラインミスのような問題が確認されたときは，その原因を究明し，マッピングにフィードバックする必要があるが，実機実行のログからその原因を特定することは非常に労力を要する．

## 1.2 論文の概要

本論文では，前節で述べた既存の開発フローの課題に対応した開発フローの提案およびその開発フローを実現するための研究 (I)(II)(III) を行った．

提案する開発フローを図 1.1 の C に示す．提案する開発フローでは，まず，既存の開発フローと同様に使用するマルチコアアーキテクチャを決定する．次に，(II-1) 関連度の高い処理のグルーピングを行う．実行およびデータの依存関係のある処理は本質的に同じコアに配置される必要性が高いためこれらをグルーピングし，そのグループ単位でのマッピングを行うことで探索空間を限定し，(f) の課題に対処する．

次に，(II-2) ツールによって静的解析結果を用いたマッピングパターンを選定する．具体的には最悪余裕時間解析を用いて各タスクのデッドラインミスの可能性の判定を行い，(e) の課題に対処する．ここで，静的解析の結果によりどのタスクがデッドラインミスする可能性があるかがわかるため，それにより (h) の課題に対処できる．また，静的解析によってデッドラインミスの可能性を検証するため，実機での動作検証は不要であり，理論的に最悪余裕時間を算出するため，(i)(j) の課題を解決できる．

次に，(I) ツールによって同期が不要なランタイムの自動生成を行う．使用するマルチコアアーキテクチャのコア数やメモリ構成を入力として与え，それらの情報を用いてランタイムを生成するため，(a) の課題に対応できる．また，ツールによってタスク化単位を自動抽出し，タスク間では同期を用いずに実行依存関係の解決を実現できるようなランタイムを自動生成する．同期を用いないため (c) の課題に対処でき，ツールによって人力より細かい粒度にタスク分割できるため (d) の課題に対処でき，さらにマッピングを決定すると自動的にランタイムが生成されるため (g) の課題に対処できる．さらに，生成するランタイムは (III) 通信タイミングが決定的な構造を持つ．具体的には Logical Execution Time （LET）[5] という通信手法を用いる．LET は，処理間のデータ通信のタイミングを LET 区間によって決定的にすることで，通信のタイミングを決定的にする手法である．LET 区間

によって通信のタイミングが固定されているため，通信のタイミングは他のタスクの実行状況や排他制御，メモリ競合によって影響を受けず，また処理の追加や処理のマッピングの変更が行われても変化しない．このようにして LET は (b) の課題を解決でき，また LET は同期を用いずに通信を実現するため (c) の課題への対処に寄与することができる．

このようにして得られたランタイム実装を実機によって動作確認し，最終的な実装とする．提案する開発フローは，前節で挙げた課題に対応でき，かつ自動化によって開発効率を向上させることができる．

上述の仮説の下に，提案するフローを実現するために研究 (I)(II)(III) を行った．各研究の番号は，提案するフローの各ステップにつけられた番号に対応している．また，研究 (I)(II)(III) は，本論文の 3 章，4 章，5 章で述べる研究に対応している．

研究 (I) は，パワトレアプリのソフトウェアと実装するハードウェアアーキテクチャをモデル化し，それらのモデルと処理，データのコア，メモリマッピングの情報を入力として与え，実装するハードウェアアーキテクチャに適したランタイムを生成するツールである，Powertrain Multicore Programing Framework（PMPF）の提案である．PMPF では与えられた入力情報を用いて，タスク実行の途中で実行依存関係による同期が必要ないように処理を分割する．タスク間の実行依存関係はタスク呼び出しによって実現するため，同期を用いずに実行依存関係を成立させることができる．そして，タスクの実装コードを含むランタイムコードを自動生成する．PMPF によって，同期が不要なランタイムを自動生成することが可能となった．また，PMPF では必要なタスク数を抑え，タスク起動による OS オーバヘッドの削減を可能とした．さらに，少ない変更かつ現実的な時間で複数のマルチコアプラットフォームに対応することや処理，データのマッピングを変更することが可能となった．具体的には PMPF によって，数行の変更でマッピングの変更が可能であり，現状のパワトレアプリの規模であれば 2 分半程度でランタイムコードの生成が可能であった．

研究 (II) は，最悪余裕時間の静的解析の結果を用いてマッピングの候補を選定する手法の提案である．研究 (II) は (II-1) 関連度の高い処理のグルーピング，(II-2) ツールによる静的解析結果を用いたマッピングパターンの選定の 2 つのステップに分割できる．(II-1) では，静的解析を行う候補を限定するために，処理間の実行依存関係やコア間排他が必要となるデータ数を考慮して処理の集合を複数のグループ（PF グループ）に分割する．PF グループ内の処理は同じコアに配置される必然性の高いものであるため，PF グループ単位でのコア配置を考えることでマッピ

ングの探索空間を限定することができる．(II-2) では，各 PF グループのコア配置パターンに対して最悪余裕時間の解析を行い，その解析結果を用いてマッピングパターンを選定する．最悪余裕時間を指標として用いるのは，パワトレアプリは一般的にハードリアルタイムなシステムであるため，各タスクの最悪余裕時間が長くなれば，正しくデッドライン以内に完了できる可能性が高まると考えられるからである．最悪応答時間の解析は，既存手法の MIF [6] をマルチコア向けに拡張した手法によって行う．具体的にはスピンロックオーバヘッド，共有するデータへのメモリアクセス時間，タスク起動 ISR オーバヘッド，タスク切り替えオーバヘッドを新たに計算に含める．この手法により，現実的な時間でマルチコアアーキテクチャ上に実装する場合の処理のコアマッピングの候補を得ることが可能となった．実際に評価した結果，解析に要する時間は現状の規模のパワトレアプリを 4 コアに配置する場合でも 250 秒程度であり，短い時間での解析が可能であった．

研究 (III) は，パワトレアプリに対して LET を実現するためのランタイムの提案である．LET の実装方法として，LET の通信を担当する専用のタスクもしくは ISR である LET 処理（LET-P）を用いる方法が提案されている [7]．また，LET による通信オーバヘッドを削減するために，ダブルバッファリングを用いる手法が提案されている [8]．しかしこれらの既存手法には 2 つの課題がある．1 つは，サブスケジューリングに LET を対応させることである．サブスケジューリングはタスクの周期とは別に，処理に周期とオフセットを与える手法である [9]．サブスケジューリングによって，処理のタスク配置を変更させることなく重要度の低い処理の周期を長くしたり，同じ周期の処理でもオフセットをずらすことで処理の負荷を分散させることができる．車載マイコンのリソース制約が厳しい中で処理の負荷を分散させるサブスケジューリングは有用であるが，既存の LET の実装方法は，サブスケジューリングを考慮していない．もう 1 つは，パワトレアプリで起こりうるデッドラインミスを考慮することである．パワトレアプリは厳しい時間制約のもとで動作するアプリであるが，現実的にはエンジンの速度が高い場合に安全性への影響の低いタスクのデッドラインミスが起こりうる．ダブルバッファリングを用いた LET の実装ではデッドラインミスが生じると不整合なデータが発生するという問題がある．本研究では，サブスケジューリングを用いたパワトレアプリに対して最適な LET 区間の設定方法を提案する．次に，デッドラインミスによる不整合なデータの発生に対応するために，LET を用いて通信されるデータに対して，そのデータを更新する処理がデッドライン内で更新されたかどうかを監視し，不整合なデータが生じないようにするためのランタイムとして Deadline

Miss Tolerant LET（DMT-LET）提案する．ここで，LET-P は約 10,000 個ものデータを扱うことから実行時間が大きく，かつ LET-P が完了するまですべてのコアのタスクを待たせる必要があるため，オーバヘッドが非常に大きい．そのため，DMT-LET を適用しつつ，LET-P を各コアに効率的に分散する手法として HDLP（Hybrid Distributed LET Process）を提案する．これらの手法によってパワトレアプリに LET を適用することによって生じる問題に対処することが可能となった．

## 1.3　論文の構成

本論文の構成を以下に示す．まず，第 2 章では，車載制御システムの特徴と，車載制御システムに対して使用される既存の手法について説明する．第 3 章では，研究 (I) の，マルチコアの複数プラットフォームに対応するためのランタイム生成フレームワークについて説明する．第 4 章では，研究 (II) の，最悪余裕時間を用いた処理のマッピング決定手法について説明する．第 5 章では，研究 (III) の，パワトレアプリに LET を効率的に適用する手法について説明する．第 6 章では，本論文のまとめと今後の課題について説明する．

# 第2章　車載制御システムとそのマルチコア化

本章では，本論文の前提となる技術を説明する．まず，本論文が対象としている車載制御システムについて説明する．ここでは，本論文で特に焦点を当てているパワートレインアプリケーションの構成とパワートレインアプリケーションを動作させるマルチコアハードウェアアーキテクチャについて説明する．次に，本論文でも使用している車載ソフトウェアのデファクトスタンダードである AUTOSAR について説明する．次に，研究 (I) に関連して，シングルコアプロセッサ向けパワトレアプリ開発について説明する．次に，研究 (II) で使用する，既存の最悪応答時間解析手法について説明する．最後に，研究 (III) で使用する通信手法である，Logical Execution Time について説明する．

## 2.1　車載制御システム

1970 年代頃に車載制御の分野で電子制御が用いられるようになって以来，電子制御はより複雑で高度な車載制御を実現してきた [10]．現在，自動車には多くの Electronic Control Unit (ECU) が搭載されており，その数は 70 以上であると言われている [11]．それらの ECU がネットワークを構成し，相互通信を行っている [12].

車載制御システムはエンジンやトランスミッションのようなパワートレインを制御する制御系，ドアやウィンドウのような車体を制御するボディ系，カーナビやカーオーディオのようなマルチメディアを担当する情報系によって構成され，そして近年ではそれらに加えて自動ブレーキなどの Advanced Driver Assistance System (ADAS) も重要な要素となっている．中でも，制御系は安全性への影響が大きいため，高いリアルタイム性が求められている．そのため，パワートレインを制御するパワートレインアプリ (以下，パワトレアプリ) には， Real-Time Operating System (RTOS) が用いられる．

近年，パワトレアプリは排ガス規制や低燃費化などの要件により複雑化，高度化してきている．しかし，ECU の動作周波数の限界は，耐熱性や対ノイズ性などにより 300MHz 程度であると言われており，これ以上の動作周波数の向上は難しい．そのため，車載制御の分野においてもマルチコアアーキテクチャの利用が求められるようになってきた．

### 2.1.1　パワトレアプリのアーキテクチャ

ここでは，本論文で前提とする構成を持つパワトレアプリ（以下対象パワトレアプリ）について説明する．対象パワトレアプリは実際のパワトレアプリの構造を継承しており，中に含まれる処理やデータについても実際のパワトレアプリと同等の規模となっている．対象パワトレアプリは後述する AUTOSAR OS の使用を前提としているが，柔軟に処理の負荷を調整するためにサブスケジューリングという機構を持っており，ランタイムおよびアプリケーションは AUTOSAR とは異なる構造を持っている．以下に，対象パワトレアプリの処理とデータおよび処理間の依存関係について説明する．

#### 処理

対象パワトレアプリは，エンジン制御やトランスミッション制御といった，機能毎にモジュール化されている．

各モジュールは複数の処理により構成されている．これらの処理の中には燃料の噴射や点火などのハードリアルタイムな処理が含まれている．各処理は，時間周期もしくはエンジンの回転角と言ったイベントに同期して実行される．それぞれの処理には，起動の契機となるイベントと，イベントに対する周期とオフセットが設定されており，その設定に応じて実行される．処理個別に用意されている周期とオフセットは，後述するサブスケジューリングによる処理の負荷の軽減や分散に利用される．

対象パワトレアプリは機能の追加，周期変更を容易に実現できるように図 2.1 のような構成を持っている．図 2.1 の基本回転角は回転角同期タスクの実行間隔のベースとなる回転角である．例えば，図 2.1 ではエンジンが基本回転角分進むごとにタスク 3 が起動される．

各モジュールには実行タイミングや実行優先度ごとに実施する処理をまとめた関数（公開関数）が用意されている．公開関数とは，ある機能を実現するための

まとまった処理の単位であり，各公開関数はランタイムから直接呼び出される．公開関数は 1 つ以上のタスクによって呼び出される．本論文では公開関数の実行時間は実際に測定したときの平均実行時間とする．最悪実行時間を使わないのは，全ての処理が最悪実行時間であるというプログラムの構成上起こり得ない悲観的な解析を行うことになるためである．公開関数の分岐実行の組み合わせを考慮する方法は，組み合わせ数が膨大となり，解析が困難であるため，本論文では取り扱わない．

以下に，対象パワトレアプリの構成について説明する．まず，各モジュールは機能を構成する処理を管理するセットマネージャを持つ．セットマネージャはイベントの発生間隔に対する周期を持っている．例えば図 2.1 ではセットマネージャ1 の周期が基本周期*2 であるため，セットマネージャ1 の実行間隔は基本周期イベント 2 回分である．現状のパワトレアプリでは，同一のセットマネージャに関する処理は RTOS 上の 1 つのタスクとして実現される．セットマネージャは時間や回転角のようなイベントによって起動される ISR 上に含まれ，セットマネージャの周期に従って対応するタスクを起動する．タスク実行のデッドラインはセットマネージャの周期である．セットマネージャから呼び出されるタスクはサブセット (サブレイヤ) で構成される．サブセットはセットマネージャに対して同じ周期とオフセットを持っている公開関数の集合である．例えば図 2.1 のサブセット 2 は周期 4，オフセット 2 であるため，セットマネージャ1 が 2 回実行されたあと 4 回実行されるたびにサブセット 2 に含まれる公開関数が実行される．セットマネージャ1 の周期は基本周期*2 であるため，最終的にサブセット 2 に含まれる公開関数は周期が基本周期*8，オフセットが基本周期*4 のタイミングで実行されることになる．これにより，対象パワトレアプリでは，後述するサブスケジューリングが実現されている．

ランタイムから直接は呼び出されず，公開関数からのみ呼び出される関数を内部関数と呼ぶ．内部関数は，以下に示す 2 種類の排他的な実行が必要な場合がある．片方のみが要求される場合と両方共に要求される場合もある．

- リエントラント禁止
  ある公開関数が呼び出している間は他の公開関数からの呼び出しをペンディングする

- アトミック実行
  実行が開始すると関数が終了するまで中断されない

ソフトウェア規模は公開関数が 1000 個，サブセットが 100 個，セットマネージャ

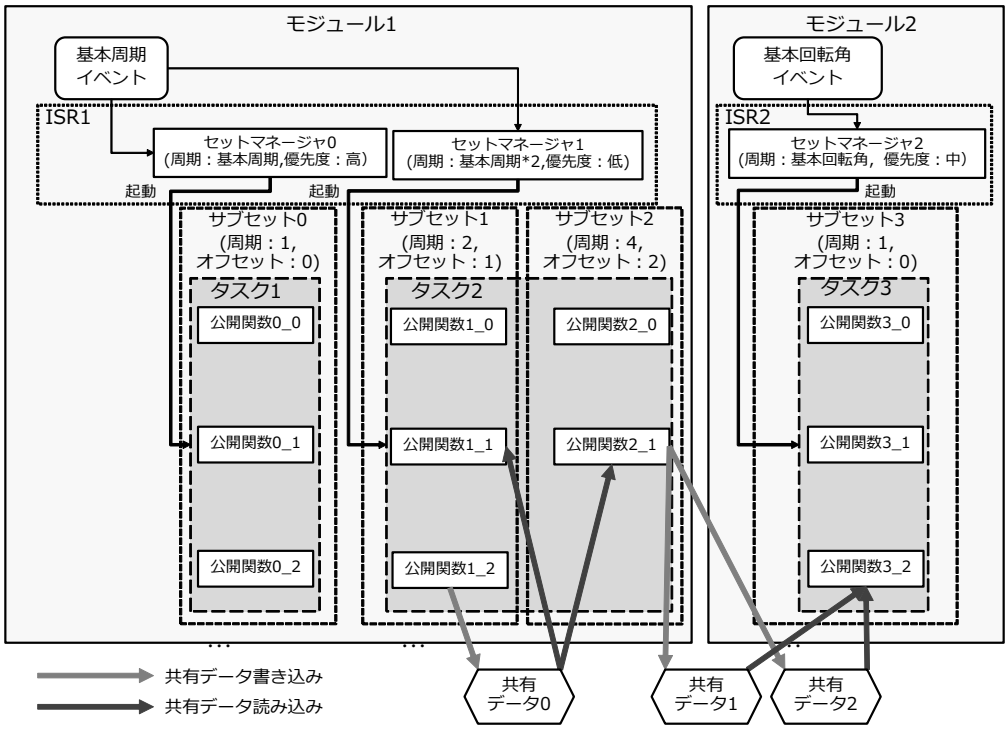


図 2.1: 現状のパワトレアプリのランタイム構成

（タスク）が 50 個程度である．

### データ

センサを制御する公開関数で読み取ったセンサ値を別の公開関数で使用するなど，公開関数間でデータの共有をする必要がある．複数の処理単位（対象パワトレアプリでは公開関数）で共有するデータを共有データと呼ぶ．図 2.1 に公開関数間での共有データのアクセス例を示す．共有データは一貫性や安定性を保つため排他制御をする必要がある．

図 2.1 の共有データ 0 のように同一のセットマネージャつまり同一タスクからのみアクセスされる場合は排他制御の必要はない．しかし，共有データ 1, 共有データ 2 のように複数セットマネージャつまり複数優先度の処理からアクセスされる場合は低優先度のタスクが共有データアクセス中に高優先度の処理にディスパッチして共有データを変更する可能性があるため，データアクセスの前後で割込み禁止を行う必要がある．

### 依存関係

公開関数間には，実行依存関係が存在する場合がある．対象パワトレアプリは元々シングルコアを前提としているため，次の方法で依存関係を実現してる．

- あるタスク内の公開関数間に依存関係がある場合は，セットマネージャ及びサブセットで実行順序の制約を満たす順に呼び出す．
- 異なるタスク間の公開関数間に依存関係がある場合は，先に実行する必要がある公開関数が所属しているタスクの優先度を高く設定する．

なお，公開関数間の実行依存関係は，ドキュメント化されているものも存在するが，パワトレアプリには車載にマイコン初めて使われた 70 年台後半から存在する処理もあり，そのような古い処理には，依存関係がドキュメント化されていないものも多く，現状の呼び出し順から変更した場合に問題が生じるかどうかの判断ができない．

### サブスケジューリング

対象パワトレアプリは Rate Monotonic Scheduling （RMS）[13] を採用しており，周期が短いタスクほど高い優先度に割当てられる．設計時に，CPU 負荷が高く重要な処理の実行がデッドラインを満たすことができない場合，いくつかの重要でない公開関数のタスク配置を変更する．具体的にはソフトリアルタイムな公開関数はより長い周期のタスクに移行される．タスクのコア配置が変更された場合，元々の設計と異なるタスクに配置された公開関数の順序は想定されていた順序とは異なる可能性がある．また，周期の長さごとにタスクを生成するとタスク数が増加し，OS 実行オーバヘッドが大きくなるという問題もある．

このような問題に対処し，実行順序を変更せずに CPU 負荷を下げることができるように，サブスケジューリングという方法を用いる．図 2.2 ではサブスケジューリングされた公開関数の実行の様子を示している．サブスケジューリングでは公開関数は所属しているタスクの周期より長い周期を持つことができる．各公開関数はサブ周期とサブオフセットを持つ．サブ周期は公開関数が配置されているタスクの起動に対するその公開関数の実行頻度である．一方，サブオフセットは公開関数の最初の実行タイミングである．各タスクに対し，同じ周期とオフセットを持つ公開関数の集合をサブセット（サブレイヤ）と呼ぶ．図 2.2 では，タスク Task1

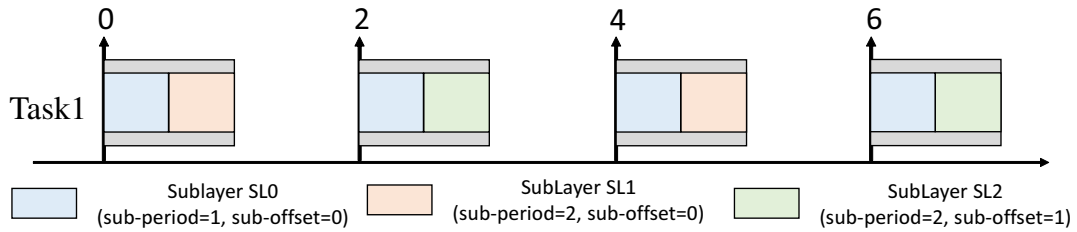


図 2.2: サブスケジューリング

の周期が 2 で，サブレイヤ SL2 のサブ周期が 2，サブオフセットが 1 であるため，SL2 は 2ms，6ms，10ms のタイミングで実行される．サブスケジューリングでは，公開関数の実行周期が長くすることで CPU 全体の負荷を低減することが可能であり，その上サブ周期が異なる公開関数も同じタスク内で実行されるため，実行順序は維持されている．

ここで，サブスケジューリングされた公開関数のデッドラインはその公開関数が実行される周期（サブスケジューリングの周期）ではなく，タスクの周期となることに注意する必要がある．

### 2.1.2　車載制御向けマルチコアアーキテクチャ

車載向けのマルチコアアーキテクチャの種類としては，各コアごとにローカルメモリを持ち，ローカルメモリへはローカルバスによって短時間でアクセスすることができるものや 2 から 4 個程度のコアがクラスタとしてまとめられ，クラスタごとにクラスタ内からは短時間でアクセス可能な RAM が用意されるものがある．また，ローカルメモリやグローバルメモリへのアクセス時間はアーキテクチャによって異なる．

車載マルチコアアーキテクチャの例を図 2.3 に示す．図 2.3 のとおり，各コアは自身が高速にアクセスできるローカルメモリ（LMEM）を持つ．あるコアが他のコアの LMEM にアクセスする場合はバスを介するためアクセス速度が遅い．また，全てのコアから等しい時間でアクセス可能な共有メモリ（SMEM）が存在する．一般的に，SMEM へのアクセス時間は各コアが自身の LMEM へアクセスするために要する時間より長い．

本論文で対象としている車載マルチコアアーキテクチャは高いリアルタイム性を保証するため，実行時間ができるだけ変動しないように設計されている．その

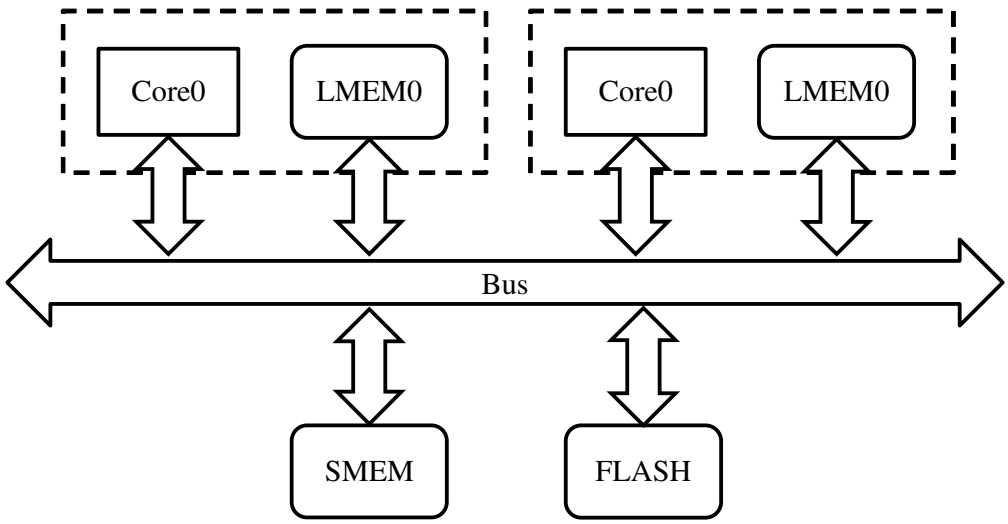


図 2.3: 対象とするハードウェアアーキテクチャの例

ため，データキャッシュおよび命令キャッシュや複数 ALU，投機的実行のような機構を持っていない．パイプライン機構は持っているものの，実行すべき命令列が決まっていれば実行時間は大きく変動しない．

現状，リリースされている車載マルチコアマイコンのコア数は，2〜3 コアであり [14][15]，6 コアのものもリリースされた [16]．

本論文で対象とするマルチコアアーキテクチャは図 2.3 に示すような構成を持つクラスタを持たない最大 4 コアのアーキテクチャである．今後，車載システムの機能が増え，5 コア以上が必要となる可能性はあるが，その場合は 4 コア程度のクラスタを複数組み合わせたアーキテクチャ[17] となり，構成が大きく変わるため，本論文の対象とはせず，対応は今後の課題とする．

## 2.2 AUTOSAR

### 2.2.1 概要

パワトレアプリのソフトウェア規模は車載システムの中では非常に大きく，1,000 個もの依存関係を持つ処理で構成されている [4]．また，自動車に含まれるソフトウェアのソースコードの総ライン数は 1 億行にも上ると言われている [11]．このように膨大かつ複雑となっている車載ソフトウェアの生産性を向上させるために，欧州の自動車メーカーが中心となって AUTomovite Open System ARchitecture (AUTOSAR)

[18] が設立された．AUTOSAR は OSEK/VDX 仕様 [19] の上位互換となるものであり，車載ソフトウェアの複雑さを軽減するためのソフトウェア基盤の標準仕様を策定することを目的としている．AUTOSAR 仕様は従来の RTOS をベースとしたエンジン制御等をターゲットとしている Classic Platform (CP) と，ADAS や自動運転をターゲットとし，POSIX ベースの OS を用いる Adaptive Platform (AP) に分かれている．本論文はパワトレアプリのマルチコア化に焦点を当てているため，CP を用いる．

CP のアーキテクチャを図 2.4 に示す．CP では，アプリケーションをソフトウェアコンポーネント (SW-C) という単位で記述し，SW-C 間および Basic Software (BSW) との通信は AUTOSAR のランタイム環境 (RTE) を介して行う．SW-C は単一の機能を実現する処理であるランナブルの集合によって表現される．ランナブルの粒度は対象パワトレアプリの公開関数に相当する．ランナブルの振る舞いは C 言語の関数によって記述される．AUTOSAR では，ランナブルは RTE ジェネレータによって OS のタスク上にマッピングされる．BSW は主にサービス層，ECU 抽象化層，マイクロコントローラー抽象化層，コンプレックスドライバに分かれており，SW-C は RTE および BSW の各層を介してマイクロコントローラーにアクセスする．

### 2.2.2　AUTOSAR OS

CP の OS（以下，AUTOSAR OS）は RTOS であり，BSW のサービス層内に含まれている．AUTOSAR OS は車載システムの要件に特化した OS であり，アラームやスケジューリングの機能が充実している．

AUTOSAR OS は OS が提供する機能に応じて SC1，SC2，SC3，SC4 の 4 つのスケーラビリティクラス（SC）が定義されている．SC1 は標準的な機能のセットであり，SC2 は SC1 に加えてタイミング保護機能を搭載している．SC3 は SC1 に加えてメモリ保護機能を搭載しており，SC4 は SC2 および SC3 を組み合わせたものである．本論文では SC1 の AUTOSAR OS を使用することを想定している．

AUTOSAR OS ではタスクもしくは割り込みサービスルーチン（ISR）をスケジューリングの単位として AUTOSAR のモデルに記述する．

タスクの状態遷移について，図 2.5 に示す．タスクの状態としては現在 CPU 上で実行中であることを示す実行状態，タスクが終了していることを示す休止状態，タスクがイベント待ちによって中断されていることを示す待ち状態，タスクが起

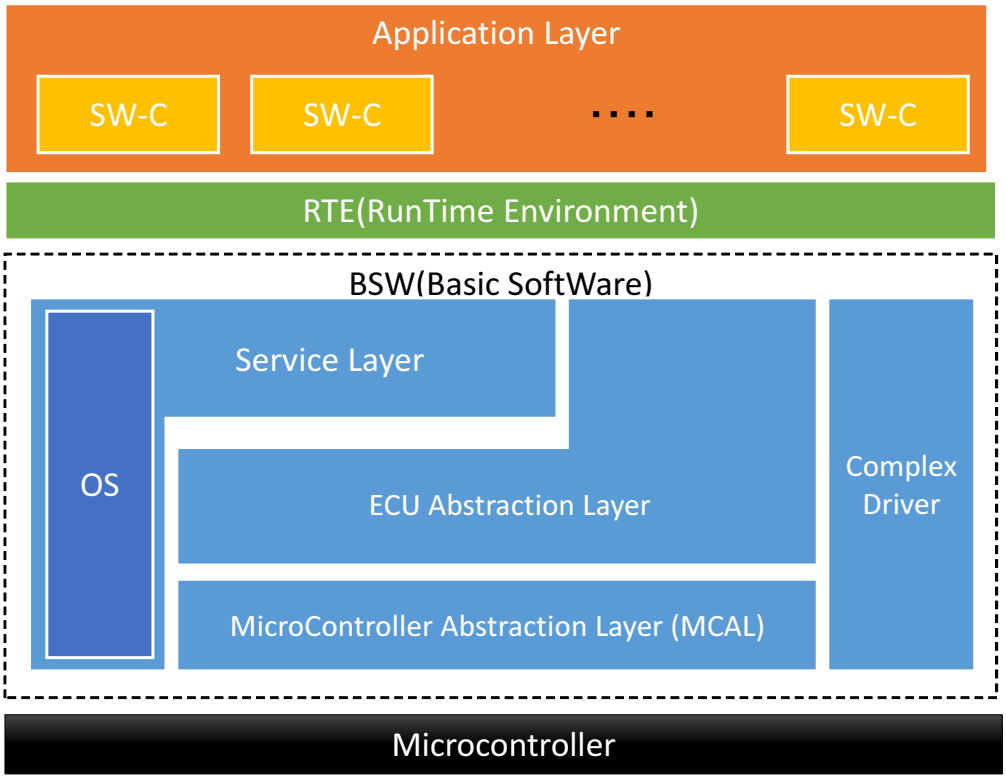


図 2.4: AUTOSAR Classic Platform

動され，実行中のタスクが終了もしくは待ち状態に入り，自身が実行できるまで待っていることを示す実行可能状態がある．

タスクに設定できる属性として，優先度，起動要求上限，スケジューリングポリシーがある．AUTOSAR OS は固定優先度スケジューリングを採用しており，再スケジューリングが発生したときに，実行状態，実行可能状態にあるタスクの中で最も優先度が高いタスクが実行状態となる．

起動要求上限はタスクが保持できる起動要求数を示している．起動要求上限が 2 以上の場合，タスクが休止状態以外のときにタスクの起動要求が行われると起動要求がキューイングされ，タスクの終了時に再び起動要求が行われる．起動要求がキューイングされるように設定すると，タスクがデッドラインミスした場合，デッドラインミスしたタスクは終了時に続けてもう一度起動される．しかしその場合，タスクが起動してから次のデッドラインまでの期間が短くなるため，再びデッドラインミスする可能性が高いと考えられる．したがって本論文では，タスクの起動要求上限を 1 と設定し，タスクが休止状態以外のときに起動要求された場合はその起動要求を棄却することにする．

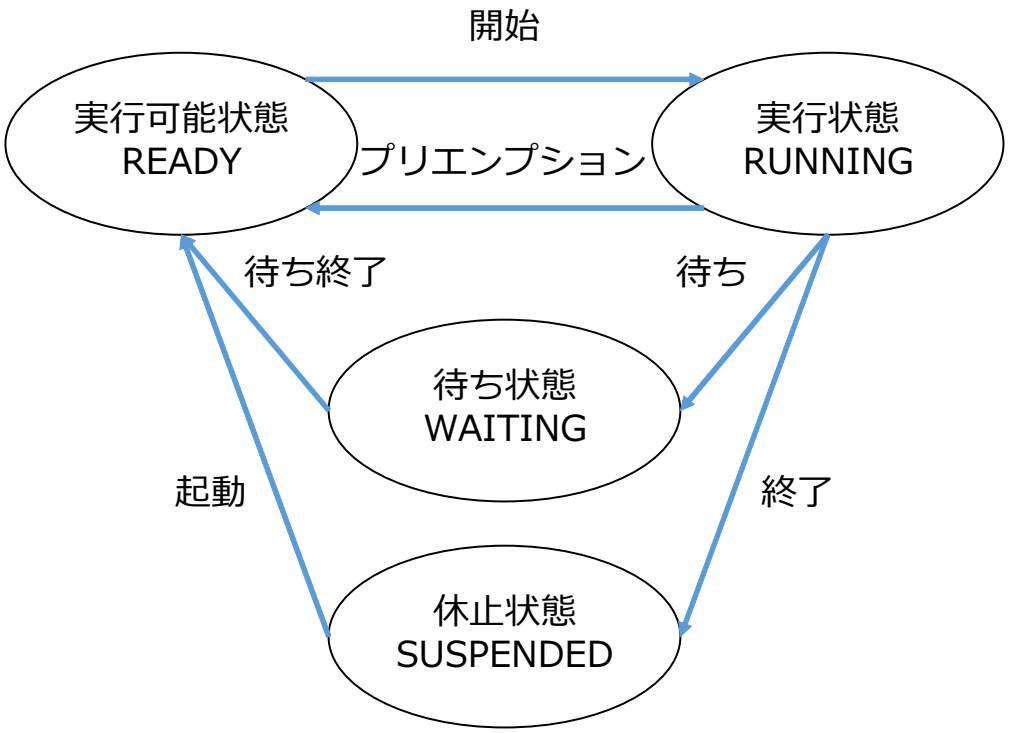


図 2.5: タスクの状態遷移

また，AUTOSAR OS では高優先度タスクが実行可能状態となったときに再スケジューリングを行うかどうかをタスクごとに設定できる．本論文では，すべてのタスクが高優先度タスクの実行可能状態への遷移に伴って再スケジューリングを行うフルプリエンプティブスケジューリングを想定している．

タスク間同期のため，AUTOSAR OS ではイベントというオブジェクトが用意されている．イベントはタスクごとに用意されたイベントマスクによって実装される．タスクは OS の API によってイベント待ちを実施し，待ち状態となる．そして OS の API によって対応するイベントがセットされると待ち状態が解除され，タスクの状態は実行可能状態へ遷移する．

タスクの種類として，基本タスクと拡張タスクの 2 種類がサポートされている．拡張タスクは待ち状態となることを許容しているタスクであり，そうでないタスクを基本タスクとする．複数の同じ優先度のタスクが基本タスクである場合，その基本タスク間ではディスパッチが生じないため，スタックを共有し，メモリ使用量を削減することができる．一般的に車載向けマイコンはメモリサイズが小さいため，本論文では，タスクはスタック共有の恩恵を受けることができる基本タスクとする．

ISR は割り込み発生時に呼び出される関数であり，AUTOSAR OS では ISR1 と ISR2 が用意されている．ISR1 は OS のコードを経由せずに呼び出される関数であり，ISR1 は OS の API を呼び出すことができない代わりに高速に割り込み処理を実行することができる．ISR2 は OS のコードを経由して呼び出される関数であり，ISR1 と比べて応答速度は遅いものの，OS の API を呼び出すことができる．本論文で扱う ISR は時間周期または特定のクランクの回転角によって発生する割り込みによって起動されるものであり，主にタスクを起動するために用いられる．したがって本論文では ISR2 を想定している．

### マルチコア対応

車載向けマイコンのマルチコアアーキテクチャ化に伴い．AUTOSAR OS もマルチコアに対応できるように拡張されている．マルチコア上のタスク実行の様子を図 2.6 に示す．図のように，異なるコアに配置されるタスク同士は互いに独立して実行することが可能である．マルチコア対応 OS の中には，タスクを実行するコアを動的に変更するタスクマイグレーションの機能を持つものもあるが，AUTOSAR OS はタスクマイグレーションに対応していない．

マルチコアアーキテクチャ上でアプリを実行する場合，異なるコアに配置されたタスクがともに共有するリソースにアクセスする可能性を考える必要がある．あるタスクがアクセス中のリソースに他のタスクがアクセスした場合，不整合な結果を得てしまう可能性があるため，異なるコアに割り付けられたタスクおよび ISR 間の排他制御を行う仕組みが必要となる．

AUTOSAR OS では，コア間排他を実現する仕組みとして，スピンロックを仕様として定めている．スピンロックは「スピンロックの獲得」,「スピンロックの解放」の 2 つの機能によって実現される．共有リソースにアクセスするタスクおよび ISR は，まずスピンロックの獲得を行う．そして共有リソースにアクセスへのアクセスが完了したあとにスピンロックの解放を行う．あるタスクもしくは ISR のスピンロック獲得中に別のタスクもしくは ISR がスピンロックを獲得しようとした場合，スピンロック待ちが発生し，そのタスクもしくは ISR はスピンロックが解放されるまでビジーウェイトをする．ここで，スピンロックの状態はスピンロック変数によって管理される．複数のスピンロック変数を使用してコア間排他を行う場合，デッドロックが生じないように注意する必要がある．AUTOSAR OS はデッドロックの発生を回避するためにスピンロックの獲得順序を管理する機能を提供

している．

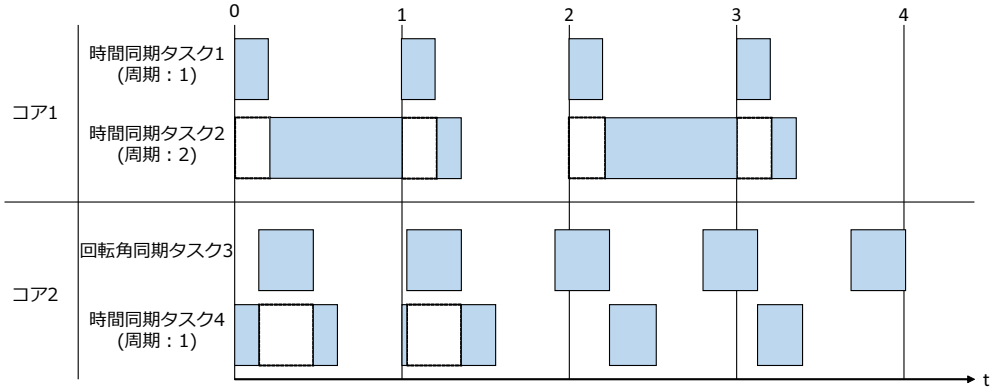


図 2.6: マルチコア上のタスク実行

**TOPPERS/ATK2**

オープンソースで利用できる AUTOSAR OS のカーネルとして，NPO 法人 TOPPERS プロジェクトが提供している TOPPERS/ATK2 Kernel がある [20]．現在 TOPPERS プロジェクトでは SC1，SC2，SC3 および SC4 のカーネルが開発されており，マルチコアに対応したカーネルもリリースされている．本論文では SC1 でマルチコア対応のカーネルである，TOPPERS/ATK2-SC1-MC を利用する．

### 2.2.3　通信メカニズム

AUROSAR では，ランナブル間の共有データの通信方法として直接通信（Explicit 通信）および Implicit 通信が用意されている．直接通信（Explicit 通信）は，ランナブルが共有データを直接アクセスする方法である．直接通信の例を図 2.7 に示す．直接通信は Implicit 通信と比べて簡素な実装であり，追加のオーバヘッドがないという利点がある．Implicit 通信は，データの一貫性を保持するために提案された通信方式である．Implicit 通信の例を図 2.8 に示す．Implicit 通信ではまず，タスクの先頭で共有データを読み込み，そのランナブルに対応したローカル領域にコピーする．この一連の処理をコピーインと呼ぶ．次にタスク実行中は共有データの代わりにローカル領域のデータにアクセスする．最後に，タスクの末尾でローカル領域のデータを共有データにコピーする．この一連の処理をコピーアウトと呼ぶ．Implicit 通信では，ランナブルで使用するデータはローカル領域に確保されている

ため，途中で他のランナブルによって値が変更されることはない．そのためデータの一貫性を保証することが可能である．Implicit 通信ではローカルメモリの使用量が増加することや，コピーイン，コピーアウトによるデータアクセスオーバヘッドがかかってしまうことに注意する必要がある．直接通信では，複数のコアで共有されるデータにアクセスする際にはスピンロックによるコア間排他制御を行う必要がある．Implicit 通信では，ローカル領域へのアクセス時にはコア間排他制御は不要であるが，コピーイン，コピーアウトでは直接通信と同様にコア間排他制御は必要である．

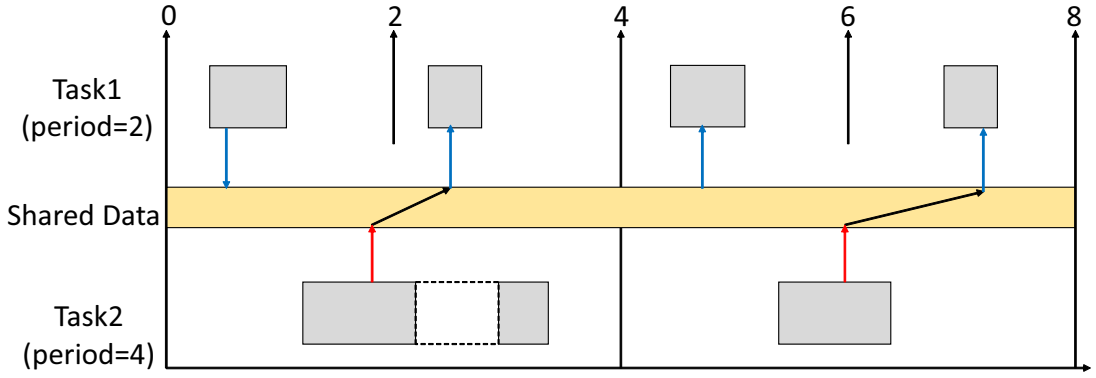


図 2.7: Explicit 通信

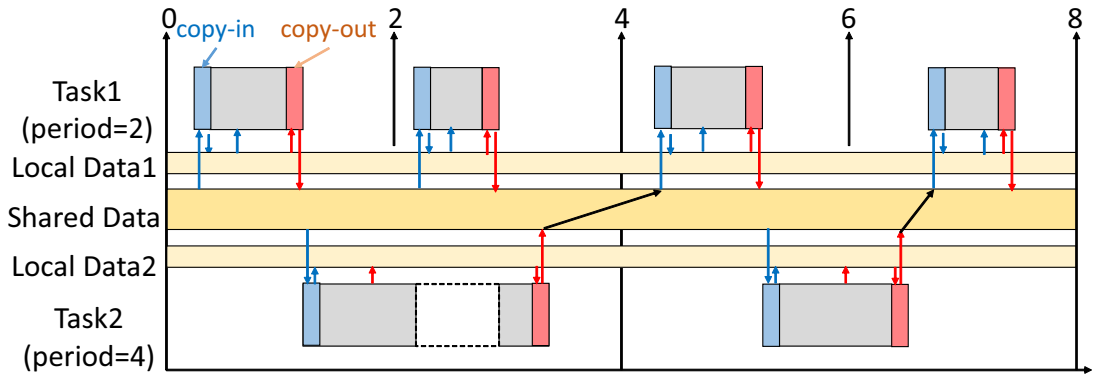


図 2.8: Implicit 通信

## 2.3　シングルコアプロセッサ向けパワトレアプリ開発

パワトレアプリのソフトウェア規模はカーナビゲーションシステムを除けば車載システムで最も大きく，1,000 個程度の依存関係のある処理で構成されている [4]．パワトレアプリは，ソフトウェア規模が大きいこと，開発に高価な HILS や実際のエンジンを動作させる必要があることから開発コストが高い．一方，車種により使用するエンジンやセンサが異なり，使用されるマイコンも異なる．そのため，単一のソースコードをベースに ECU ごとにカスタマイズすることにより，開発コストを抑えている．

シングルコアプロセッサでは，ECU ごとに構成は大きく変わらないため，使用する ECU の変更によって，ソフトウェアを大幅に変更する必要はない．したがって，ECU ごとに必要な設定を実行するコードを記述し，条件コンパイルによって使用する ECU を切り替えることで，複数のシングルコアアーキテクチャへの対応を実現することができる．

また，既存の設計では，設計者は公開関数のコアへのマッピングを決めた後，依存関係を満たすように公開関数が配置されたタスクを用意し，公開関数を呼び出すコードや他のタスクを起動するコード（ランタイム）を開発する．現状，公開関数のマッピングは自動化されていないため，最初のマッピングが最適でなく実装後に評価を行い見直す必要があることや，頻繁に機能追加がなされることから，開発中に公開関数のコア割り当てや，実行周期を変更する頻度が高く，その都度ランタイムの変更が必要となる．

## 2.4　既存の最悪応答時間解析手法

### 2.4.1　シングルコア実行時の最悪応答時間解析

車載システムが大規模化，複雑化していく中で，システムの品質を保証するためには体系的なリアルタイム性保証技術が必要となってくる．ここでは，既存の車載向けシングルコア向けの最悪応答時間解析を行い，リアルタイム性を保証する手法について述べる．タスクが常にデッドライン以内に収まっていることをスケジュール可能と呼び，スケジュール可能であるかどうかを解析することをスケジューリング解析と呼ぶ．対象パワトレアプリはエンジン点火のタイミングなどの時間制約を持つ処理によって構成されており，時間制約が満たされなかった場合

に人命に関わるような重大な事故につながる可能性のあるハードリアルタイムシステムである．そのため，それぞれのタスクの時間制約が満たされていることを保証する必要がある．各タスクの最大応答時間がデッドライン（周期処理の場合次の起動タイミング）以内に収まっていれば，その車載システムはリアルタイム性が保証されている．Critical Instant 定理 [21] を満たしているマルチタスクセットについて，タスク $\tau_i$ がスケジュール可能であることを示すためには以下の必要十分条件を満たせばよい．

$$\sum_{j \in hp(i)} C_j \left\lceil \frac{T_i}{T_j} \right\rceil \leq T_i \tag{2.1}$$

ここで，$T_j$ は優先度が $j$ 番目のタスクの周期，$C_j$ は優先度が $j$ 番目のタスクの最大実行時間，$hp(i)$ はタスク $\tau_i$ よりも高優先度なタスクの集合ある．

### 2.4.2　マルチフレームタスクモデル

タスク上で実行される公開関数はサブセットの周期とオフセットにしたがって起動タイミングが異なってくる．例えば周期 4，オフセット 2 のサブセットに含まれる公開関数が起動するタイミングは 2，6，10... である．タスク上で実行される公開関数を含むサブセットは複数個である場合も多く，タスクは起動するタイミングによって処理内容が大きく変化し，タスクの最大実行時間と平均実行時間には大きな差が出る可能性がある．そのため，(2.1）式をそのまま用いると悲観的な結果となることが予想される．このような問題に対処するため，タスクが複数の実行時間を持つことを前提としたマルチフレームタスク（MF タスク）モデル [22] を導入する．

マルチフレームタスクモデルとは，周期ごとの最大実行時間があるパターンにしたがって変化するようなタスクを扱うためのモデルである．$P_i$ をタスク $\tau_i$ のフレーム長，$C_i^j$ をタスク $\tau_i$ の $j$ 番目のフレームの最大実行時間，$N_i$ をタスク $\tau_i$ のフレーム数とすると，タスク $\tau_i$ のマルチフレームタスク（MF タスク）は $((C_i^0, ..., C_i^{N_i-1}), P_i)$ と表現される．例えば MF タスク $((1, 2, 3), 4)$ の場合，時刻 0 で最大実行時間 1, 時刻 4 で最大実行時間 2, 時刻 8 で最大実行時間 3, 時刻 12 で最大実行時間 1 のタスクとなる．MF タスクの最大応答時間を調べるためには，そのタスクより優先度が高いすべてのタスクについてどのフレームから開始したときに最もそのタスクの実行を邪魔するかを確認しなければならない．MF タスクセットのスケジュール可能性判定を効率よく行う方法として Maximum Interference Function（MIF）[6] を用いたものが提案されている．MIF $M_i(t)$ は長さ $t$ の間にタスク $\tau_i$ がタスク $\tau_i$ より優

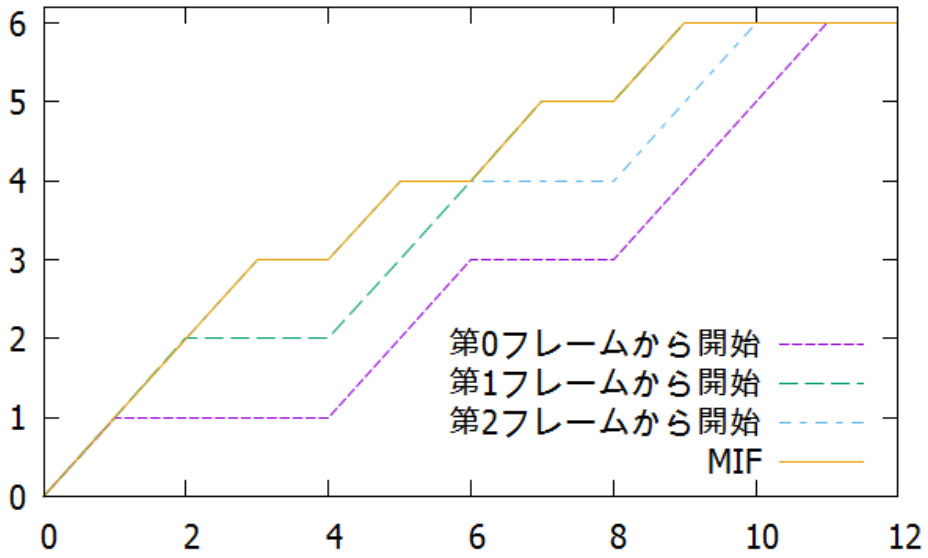


図 2.9: MF タスク $((1,2,3),4)$ の MIF

先度の低いタスクの実行を妨げる最大時間である．図 2.9 に MF タスク $((1,2,3),4)$ がフレーム 0,1,2 それぞれから開始したときの CPU 利用時間を示す．各フレームから開始したときの CPU 利用時間の最大値をとる関数が MIF の値となる．

MF タスクのスケジュール可能性を判定するためには，MIF を用いて（2.1）式を変更し，フレーム $k$ におけるタスク $\tau_i$ に関して以下の式を充たせばよい．

$$\sum_{j \in hp(i)} M_j(D_i) + C_j^k \leq D_i \tag{2.2}$$

ここで，$D_i$ をタスクのデッドライン，$C_j^k$ をタスク $\tau_i$ の $k$ 番目のフレームとする．この式を充たすタスクがスケジュール可能であることは [6] で証明されている．

## 2.5　Logical Execution Time

タスクの起動タイミングはタスクの周期によって決定されるが，タスクが実際に共有データにアクセスするタイミングは高優先度のタスク（特に非同期なタスク）および ISR の起動やタスク内の実行パス，バス競合やコア間排他制御に影響される．ここで，ランナブル間の共有データ通信のタイミングが決定的である性質を時間的決定性と呼ぶ．時間的決定性があれば，データ通信のタイミングが固定化されるため予測可能性が向上する．しかし時間的決定性を持たない場合は逆に，

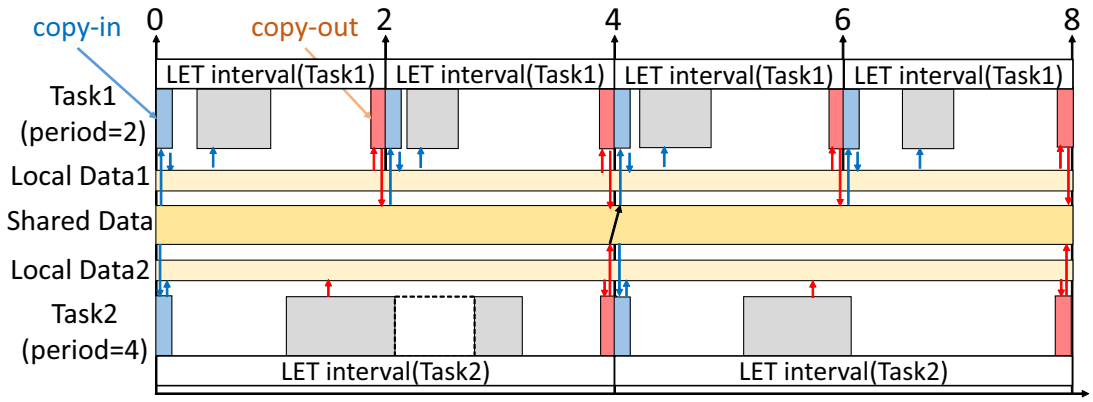


図 2.10: Logical Execution Time

考えられるすべての実行タイミングを想定してする必要が出てくるため，データ通信が複雑となる．したがって，時間的決定性を保証できるような通信方法が求められている．

時間的決定性を保証する通信方法として LET（Logical Execution Time）通信 [5] がある．LET の例を図 2.10 に示す．LET ではあるタスクについて，一定の決定的な時間間隔（一般的にはタスクの周期）を LET と定め，LET の開始時にコピーインを実行し，タスクではローカル領域にアクセスし，LET の終了時にコピーアウトを実行する．LET では，タスクごとに一定の時間間隔を LET 区間と定める．また，ランナブルごとに，そのランナブルがアクセスする共有データと同等のサイズのローカルデータを用意する．各ランナブルは共有データではなく，自身のローカルデータに書き込みを行う．LET ではコピーインとコピーアウトがそれぞれ LET 区間の先頭と末尾で行われる．コピーインは共有データを読み込み，ランナブルのローカルデータにコピーする操作である．コピーアウトはランナブルが更新したローカルデータを共有データにコピーする操作である．これらの操作は，LET 区間中に実行されるランナブルが使用する共有データすべてに対して行われる．LET ではすべての共有データアクセスの実行タイミングが決定的となるため，時間的決定性が保証される．LET は前述の通信方法と異なり，ランナブル間の共有データ通信の end-to-end latency のジッタはわずかである [23].

# 第3章　車載制御システム向けマルチコアプログラミングフレームワーク

## 3.1　概要

本章では，1 章で述べた研究 (I) について説明する．

2.3 節で述べたとおり，パワトレアプリの開発コストは高いため，マルチコアにおいても，単一のソフトウェア記述を複数の ECU で再利用する必要がある．マルチコアではコア数がマイコンごとに異なるため，条件コンパイルやランタイムコードによる対応では，余分な条件判定やタスク数の増加等により実行オーバヘッドが大きくなるという問題がある．

本章では，複数のマルチコアアーキテクチャにパワトレアプリを対応させるためのランタイム生成ツールを提案する．具体的には，単一のソフトウェア記述を定め，マッピングに従ってランタイムを生成するプログラミングフレームワーク Powertrain Multicore Programing Framework（PMPF）を実現する．PMPF では既存のパワトレアプリの構成を基にしたモデルを定義し，このモデルによりパワトレアプリの構成を記述し，それに加えてコア数やメモリ構成といったマイコンのモデルと処理やデータのコアとメモリへのマッピング指定を入力して，ランタイムを生成する．ランタイム生成においては，使用リソースを削減するためタスク数を最小限とし，実行に必要なタスクを抽出して処理を割り当てる．実現したモデルとランタイム生成機能の評価として，記述量や他の手法との比較，モデルの規模に対するスケラビリティを評価する．

研究 (I) によって，1 章で述べた課題 (a)(c)(d)(g) に対応しつつ，少ない変更量で，現実的な時間でランタイムを自動生成することが可能となる．

## 3.2　マルチコア化の要件

本節では，パワトレアプリをマルチコアに対応させる場合の要件について述べる．

### 3.2.1　複数アーキテクチャサポート

既存のシングルコア向けのパワトレアプリと同様に単一のソフトウェア記述から様々な車種に対応した実装を実現する必要がある．具体的には次の 2 つの要件が挙げられる．

(1) コア数を任意の数に設定でき，公開関数の各コアへのマッピングを短時間で変更できること

(2) 様々なメモリ構成に対応できること

要件（1）は，様々な公開関数のマッピングを短期間で評価するために必要となる．要件（2）も同様にマルチコアマイコン毎にメモリ構成が異なるために必要となる．

#### マルチコア化のコスト低減

シングルコアからの移行コストを低く抑えるために，次の要件を定める

(3) 公開関数の構成や内容は変更しないこと（セットマネージャ・サブセットは変更可能）

(4) 対象パワトレアプリにおけるタスク内の公開関数の実行順で実行する機能を持つこと

(5) AUTOSAR OS[24] を利用すること

要件（3）は 1000 個にも及ぶ公開関数の構成や内容を変更することは現実的ではないためである．要件（4）はマルチコア化のプロセスとして，まず既存の実行順序を踏襲し，その後，コード解析等により判明した不要な依存関係を取り除くという方法を想定しているためである．要件（5）は，AUTOSAR 仕様は車載ソフトのディファクトスタンダートであり，現在のパワトレアプリにおいても AUTOSAR OS を利用しているためである．

**リソース制約への対応**

マルチコアマイコンにおいても処理やメモリに対するハードウェア資源の制約は依然厳しいため，以下の要件を定める．

(6) スタックの使用量を抑えること（タスクの待ち状態を使わないこと）

(7) タスク数を抑えること．

要件（6）は，AUTOSAR 仕様では，待ち状態を使用しないタスクは基本タスクと呼び，基本タスクでは同一コアかつ同優先度のタスク間ではスタックを共有してメモリの使用量を抑えることが可能となるためである．要件（7）はタスクの起動・終了の実行オーバヘッドを抑えるために必要である．

### 3.2.2　実現手法検討

前述の要件を満たしてパワトレアプリをマルチコア化する方法としては次の 3 種類の方法が考えられる．

**（a）公開関数毎のタスク化手法**

公開関数毎にタスク化する方法で，各タスクは実行タイミングと依存関係が満たされると起動される．処理へのコアの割り当ての変更は，タスクのコアへの割り当てを変更すれば実現でき，AUTOSAR 仕様で定義されたコンフィギュレーションを変更するだけで良い．

**（b）公開関数の選択実行手法**

同一のセットマネージャを全コアで実行し，公開関数の呼び出し時にテーブルを参照して，実行されているコアに配置されていれば呼び出すようにする方法である．公開関数のコア割り当ては，テーブルを変更することで変更可能である．

**（c）公開関数呼び出しコード生成手法**

セットマネージャに相当するコードを，公開関数のコア割り当てに応じて生成する方法である．

公開関数の数は1000 前後にも及ぶため,（a）の手法ではタスクの起動・終了の実行オーバヘッドが大きくなり，要件（7）を満たせないと予想される.（b）に関しては，依存関係がある公開関数が他のコアに配置された場合，待ち状態を使用してその公開関数の終了を待つ必要がある．これは要件（6）に反する．一方（c）は，ツールを作成する必要はあるが，待ち状態が必要ないように公開関数をグループ化してタスク化すれば，要件（6）と要件（7）を満たすことが可能と予想される．その他の要件についても満たすことができるため，本章では（c）を採用する．

## 3.3　車載制御向けマルチコアプログラミングフレームワーク

公開関数呼び出しコード生成手法を実現するフレームワークとして，前章で述べた要件を実現する Powertrain Multicore Programing Framework（PMPF）について説明する．

### 3.3.1　設計フローとコンセプト

PMPF による設計フローを図 3.1 に示す．まず特定のマイコンの構成に依存しない形でパワトレアプリのモデルを記述する（SW モデル）．次に，使用するマイコンのハードウェアの構成モデル（HW モデル）を定義する．これらの記述に加えて，公開関数と共有データのハードウェアへの割り付けを記述したマッピング情報を入力として，PMPF によってランタイムコードおよび OS コンフィグレーションを生成する．それらに加えて公開関数の本体および割込みハンドラ等を記述したユーザ C 言語コードと AUTOSAR OS をビルドして実行バイナリを生成する．次に，実行バイナリを実行して結果を評価し，マッピング情報へのフィードバックを行う．

前節の要件を満たすための PMPF のコンセプトについて説明する．なお，本論文では，各モデルにおいてはメモリとデータを扱うが，ランタイム生成に関しては処理のみを扱い，データのメモリへの配置に関しては今後の課題とする．

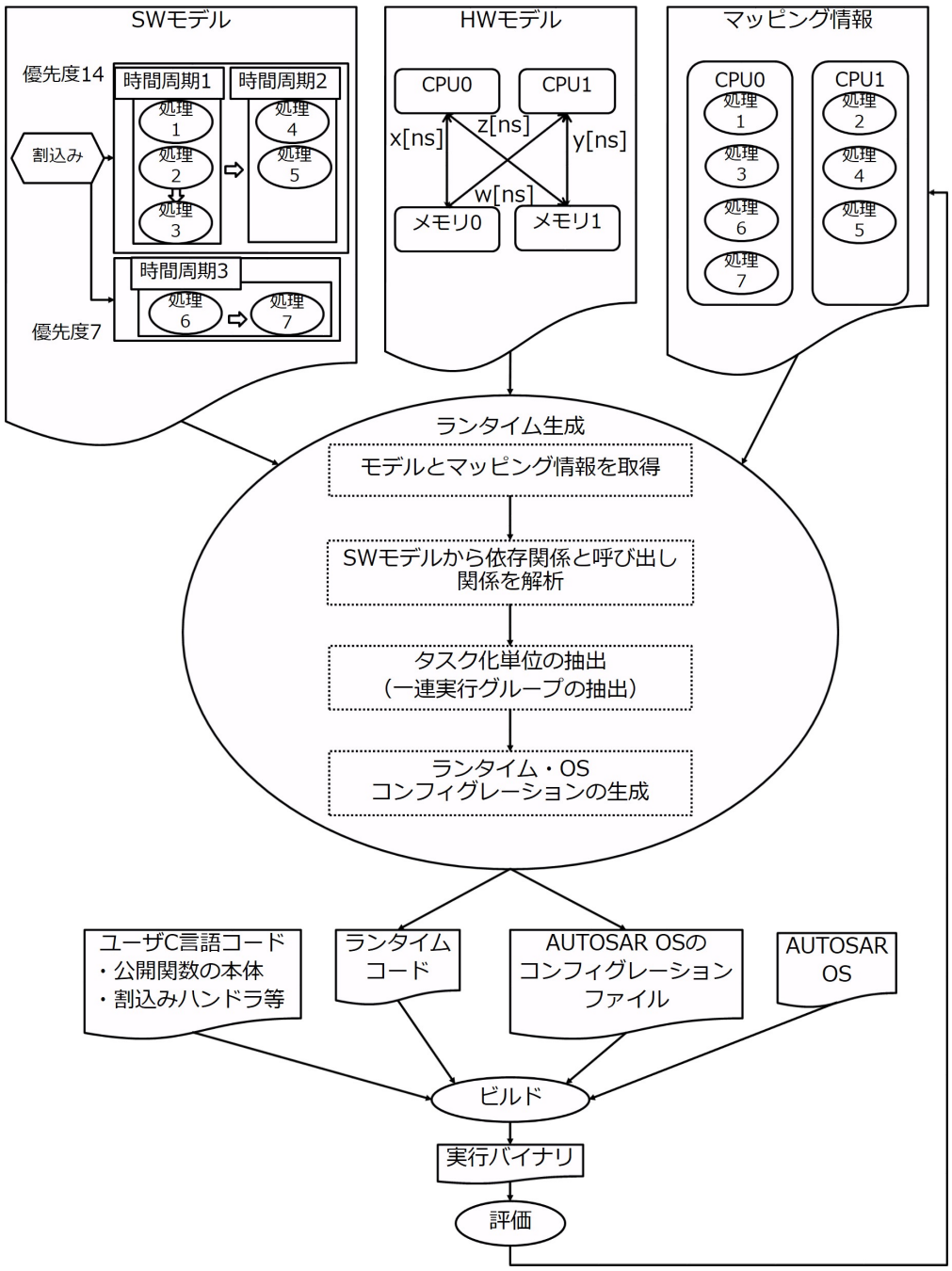


図 3.1: PMPF による設計フロー

- 多様なマルチコアアーキテクチャへの対応:

  要件 (1) と要件 (2) を満たすため，ソフトウェアモデルをコア数やメモリ構成とは独立して記述する．また，コア数やメモリ構成を記述したモデルを用意する．

- **パワトレアプリのモデル化**

  対象パワトレアプリの構成を踏襲してパワトレアプリをモデル化する．要件 (4) を満たすため，モデルでは，セットマネージャ，サブセット，公開関数に相当する構成要素を持ち，それぞれ依存関係を設定可能とする．なお，要件 (3）から，公開関数の実体はモデルとは別に記述する．

- **マッピング指定**

  要件 (1）を満たすため，パワトレアプリのモデルに記述した公開関数のコア配置やデータのメモリ配置を決定するマッピング情報を記述できるようにする．

- **ランタイム生成**

  図 3.1 に示したように，上記の SW モデル，HW モデル，マッピング情報を入力としてランタイム生成を行うツールを作成する．ランタイムではセットマネージャ・サブセットに相当する処理を実現する．いくつかの公開関数をまとめてタスク化することで，タスク数が抑えられるようにランタイムを生成する．これによりタスク起動・終了のオーバヘッドを削減し要件 (7）を満たす．要件 (6）を満たすため，途中で待ちが必要とならないような公開関数のグループを抽出し，そのグループ単位でタスク化する．

  また，要件 (5）を満たすため，AUTOSAR OS に対応したランタイムを生成する．

2.3 節で述べたように，既存の設計ではマッピングの変更や機能の追加がある度にランタイムの変更を行う必要があり，そのために大きな工数が必要となる．一方，PMPF は，SW モデルに対して公開関数のコアへの割り付けを指定することにより，必要なタスクの決定とランタイムを自動生成する．これにより手作業による開発と比較して開発工数を削減することが可能となる．同様にモデルからランタイムを生成する方法として，Simlink 等用いたモデルベース開発においても，モデルからランタイムを生成する機能を持つツールが存在する．しかしながら，既存のツールではパワートレインアプリケーションの構成を特性を考慮していない．例えば，セットマネージャやサブセットの構成を記述したり，タスク化単位の抽出を自動的に行うことに対応していない．

### 3.3.2　モデル

本節では PMPF における各モデルについて説明する．モデルは，SW モデル，HW モデル，マッピング情報の 3 種類がある．モデルやマッピング情報は YAML と呼ばれるデータ形式により記述する．

HW モデルには，使用するマイコンのコア数や，メモリアーキテクチャを記述する．SW モデルは，現状のパワトレアプリの構成をベースとしたモデルを定義することで，既存のソフトウェアからの移行を容易化する．パワトレアプリの SW モデルを図 3.2 に示す．マッピング情報には，公開関数のコアへの配置を記述する．マッピング情報の例を図 3.3 に示す．コア毎に配置する公開関数をリストとして指定する．

以下に，SW モデルの詳細を説明する．

#### SW モデルの構成要素

SW モデルの構成要素について説明する．

**トリガー**

処理の契機となるオブジェクトで，周期割込みやクランク角割込みを契機に発生しタスクグループを起動する．

**タスクグループ**

優先度が同じ処理の集合で，同優先度つまりデッドラインが同じ処理をまとめて扱うためのオブジェクトである．現状のパワトレアプリにおけるセットマネージャに相当する．

**タイミングマネージャ**

同一タスクグループ内で周期とオフセットが同じ処理の集合でタスクグループの中でも実行タイミングが同じ処理をまとめて扱うためのオブジェクトである．現状のパワトレアプリにおけるサブセットに相当する．

**公開関数**　2.1.1 節で説明したものと同等である．モデルとしては，実行時間やアクセスするデータについての情報を記述する．

**排他実行内部関数**　内部関数のうち排他的実行の必要があるものを定義するオブジェクトである．必要となる排他的実行の種類（リエントラント禁止/アトミック実行）を指定する．

**共有データ**

公開関数間で共有されるデータオブジェクトで，データへのアクセスは公開関数内にツールで用意されるデータ・アクセス用 API を記述して行う．

**データグループ**

共有データの集合で，含まれる共有データに関して排他を設定できる．排他の種類はジャイアントロック/細粒度ロックがあり，排他の種類はアクセスする公開関数の関係で自動選択される．

### 依存関係の記述

対象パワトレアプリには処理の依存関係が存在し，セットマネージャによってサブセットの実行順序が決められ，サブセットによって公開関数の実行順序が決められている．

PMPF では，以下の 2 つの依存関係を定めた．これら 2 つの依存関係をまとめて実行開始条件と呼ぶ．

- 呼び出し順指定
- 先行制約

呼び出し順指定は，同じタスクグループ/タイミングマネージャに含まれるすべてのタイミングマネージャ/公開関数の実行順序を完全に決める．これは現状のパワトレアプリの依存関係と同等のものであり，要件（4）を満たすために用意する．

先行制約は既存のパワトレアプリにおけるサブセットに相当するタイミングマネージャと，公開関数について必要な依存関係のみを個別に設定し，依存関係を持たない構成要素同士を並列に実行できるようにしたものである．先行制約は同一トリガーから呼び出されるものであれば，呼び出し元のタスクグループやタイミングマネージャに関わらず設定可能である．

マルチコア化のプロセスとしては，シングルコア時の実行順序を実現する呼び出し順指定で依存関係を記述して，不要な依存関係を取り除き，徐々に先行制約

に移行していくことを想定している．依存関係はSWモデルの各オブジェクトの属性として設定する．図3.4は，図3.2のSWモデルを表すYAML記述から依存関係に関する部分を抜き出したものをである．図3.2の依存関係(a)∼(d)は図3.4の記述(a)∼(d)に対応している．

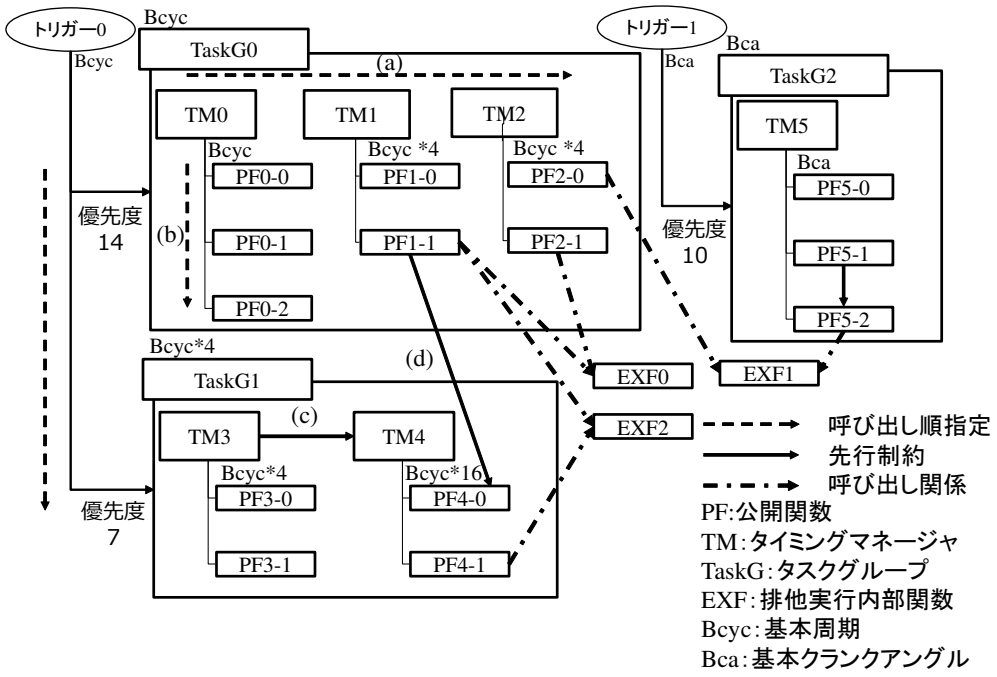


図 3.2: PMPF のSWモデル

```
MAPPING_DEFINITION:
  PublicFunction:
    Core0:
      - PF0-0
      - PF0-1
      - PF0-2
      - PF1-0
      - PF1-1
      - PF2-0
      - PF2-1
      - PF5-0
      - PF5-1
      - PF5-2
    Core1:
      - PF3-0
      - PF3-1
      - PF4-0
      - PF4-1
```

図 3.3: マッピング情報

```
TASKGROUP_DEFINITION:
   TaskG0:
      TimingManagerCallSeq: #(a)
         - TM0
         - TM1
         - TM2

TIMINGMANAGER_DEFINITION:
   TM0:
      PublicFunctionCallSeq: #(b)
        - PF0-0
        - PF0-1
        - PF0-2
   TM4:
      PreceedenceTimingManager: #(c)
        -   TM3

PUBLICFUNCTION_DEFINITION:
   PF4_0:
      PreceedencePublicFunction: #(d)
        - PF1-1
```

図 3.4: 図 3.2 の依存関係の記述 (一部)

## 3.4　ランタイム生成

本節では，PMPF のランタイム生成について説明する（図 3.1 を参照）．

まず，SW/HW モデルとマッピング情報を読み込む．次に，SW モデルの情報から各オブジェクトの呼び出し関係と公開関数及びタイミングマネージャの依存関係を解析する．その後，各種情報と制約から必要なタスクを抽出する．これをタスク化単位の抽出と呼ぶ．最後にタスクの本体となるランタイムコードと，タスクの生成情報を記載した AUTOSAR OS のコンフィギュレーションファイルを生成する．

以降，タスク化単位の抽出，ランタイムのコード生成の詳細について説明する．

### 3.4.1　タスク化単位の抽出

#### 公開関数のタスクへのマッピングにおける課題

シングルコアを対象とした既存のパワトレアプリでは，優先度と起動契機となるイベントが同じ公開関数のグループを抽出し，そのグループ単位でタスク化している．PMPF のモデルに置き換えると，タスクグループ単位でタスクとしている．この手法をマルチコアへ適用すると，同タスクグループかつ同コアに配置された公開関数単位でタスクとすることになる．この方法では 3.2 節で述べた (b) の方法と同様に待ち状態が必要となり要件（6）を満たせない．例えば，図 3.5 の例では，タスクグループ TaskG1 に含まれる公開関数はすべてコア 1 に配置されているため，この方法では TaskG1 は一つのタスクで実現される．しかし，公開関数 PF4-0 は公開関数 PF1-1 の終了を待つ必要があるため，ここで待ち状態が必要となる．

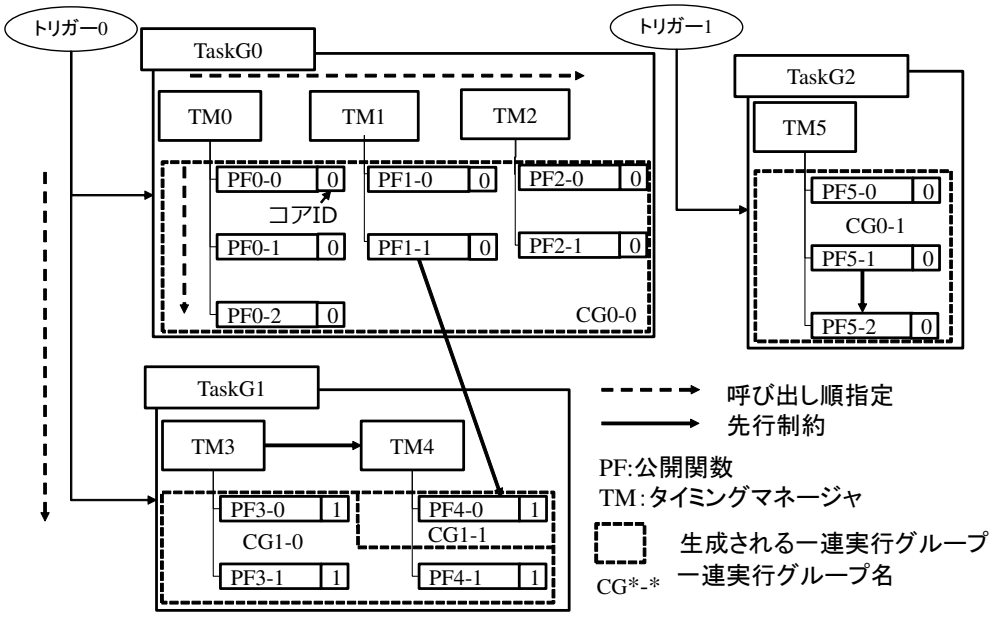


図 3.5: 公開関数のコア割当て 1

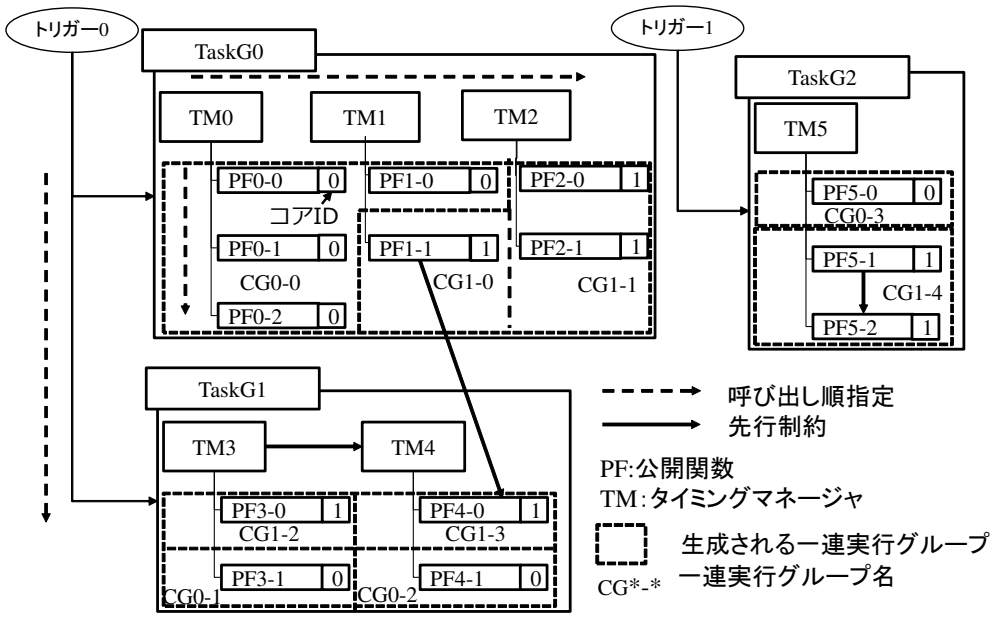


図 3.6: 公開関数のコア割当て 2

### 一連実行グループ

公開関数の依存関係は有向非巡回グラフとして表現できる．公開関数の依存関係のグラフを $D = (F, A)$ とすると，頂点集合 $F$ の要素は公開関数であり，辺集合 $A$ の要素は公開関数間の依存関係である．ここで，要件を満たすようなタスク化単位の抽出を行うために，以下の条件 (a)(b) を満たす公開関数の最大集合である一連実行グループを定義する．

**条件 (a)** 同一コアに割り当てられているかつ同一タスクグループ (同一優先度かつ同一トリガーにより起動) に所属している公開関数の集合

**条件 (b)** 公開関数 $f_i$ へのパスが存在しかつ公開関数 $f_i$ とは異なる一連実行グループに含まれる公開関数の集合が，公開関数 $f_i$ と同一の一連実行グループに含まれる各公開関数において等しい．

一連実行グループを $CG$ とすると．一連実行グループ $CG$ に含まれる公開関数の数が 1 のとき，条件 (a) は自明である．一連実行グループ $CG$ に含まれる公開関数の数が 2 以上のとき，条件 (a) は以下のように表すことができる．

$$\forall f_i, \forall f_j \in CG, C(f_i) = C(f_j) \wedge TG(f_i) = TG(f_j) \tag{3.1}$$

ここで $f_i$ は $i$ 番目の公開関数，$C(f_i)$ は公開関数 $f_i$ が割当てられているコア，$TG(f_i)$ は公開関数 $f_i$ が所属しているタスクグループを表している．

条件 (b) の”公開関数 $f_i$ へのパスが存在しかつ公開関数 $f_i$ とは異なる一連実行グループに含まれる公開関数の集合”を $f_i$ の外部依存公開関数集合と定義する．一連実行グループ $CG$ に含まれる公開関数の数が 1 のとき，条件 (b) は自明である．一連実行グループ $CG$ に含まれる公開関数の数が 2 以上のとき，条件 (b) は以下のように表すことができる．

$$\forall f_i, \forall f_j \in CG, ED(f_i) = ED(f_j) \tag{3.2}$$

ここで $ED(f_i)$ は $f_i$ の外部依存公開関数集合を表している．$ED(f_i)$ は以下の式で定義できる．

$$ED(f_i) = \{f \mid path(f, f_i) \wedge f \in (F - CG(f_i))\} \tag{3.3}$$

ここで $path(f, f_i)$ は $f$ から $f_i$ へのパスが存在すること，$CG(f_i)$ は公開関数 $f_i$ を含む一連実行グループを表している．

ある一連実行グループに含まれる公開関数の外部依存公開関数集合は条件 (b) よりすべて同一である．このことから，一連実行グループ $CG_n$ に含まれる公開関数の外部依存公開関数集合を一連実行グループ $CG_n$ の外部依存公開関数集合と定義する．

一連実行グループ $CG_n$ に含まれる公開関数は，一連実行グループ $CG_n$ の外部依存公開関数集合または一連実行グループ $CG_n$ の公開関数にのみ依存関係がある．また，上記の条件 (b) から，一連実行グループ $CG_n$ の外部依存公開関数集合に含まれる公開関数が全て実行完了すると（依存関係が満たされると），一連実行グループ $CG_n$ の公開関数は，一連実行グループ $CG_n$ 内の依存関係を満たす順序で待ち状態なく（他の一連実行グループの公開関数の終了を待つことなく）実行することが可能となる．そこで，一連実行グループごとにタスクを定義して，外部依存公開関数集合の実行終了を起動条件としてタスクを起動する．

一連実行グループを実行するタスクを一連実行グループタスクと呼ぶ．

### 一連実行グループ抽出アルゴリズム

ここでは，PMPF の入力となる SW モデルとマッピング情報から一連実行グループを抽出するアルゴリズムを説明する．まず，SW モデルに記述した公開関数と公

開関数間の依存関係から有向非巡回グラフ $D = (F, A)$ を作成する．頂点集合 $F$ は公開関数の集合を表し，辺集合 $A$ は公開関数間の依存関係集合を示す．このとき，タイミングマネージャ間の依存関係を公開関数間の依存関係に展開して辺集合 $A$ に公開関数間の依存関係として追加する．図 3.7 は図 3.2 の SW モデルの公開関数とタイミングマネージャ間の依存関係を展開したものを含む公開関数間の依存関係から作成したグラフである．また，マッピング情報により，各頂点 f に対して割り当てられているコアの情報を設定する．

このグラフ $D$ を入力として，Algorithm 1 を実行することで，一連実行グループを抽出する．

**Algorithm 1** 一連実行グループ抽出アルゴリズム

一連実行グループ集合 $CGG$=$\emptyset$ として，すべての公開関数がいずれかの一連実行グループに含まれるようになるまで，(1)∼(4) を繰り返す.

1. いずれの一連実行グループにも含まれていない公開関数のうち，以下の条件の内どちらかを充たすものの集合 $CG$ を抽出する.

   **(1-i)** グラフ $D$ においてどの公開関数からも接続されていない

   **(1-ii)** 接続されている同一タスクグループ内のすべての公開関数が，いずれかの一連実行グループに含まれている

2. 集合 $CG$ を，以下の (2-i) かつ，(2-ii)(2-iii) のどちらかの条件を充たすような部分集合 $CG_0, CG_1, ...$ に分割する.

   **(2-i)** 同一タスクグループに属し，同一コアに割り当てられている

   **(2-ii)** グラフ $D$ においてどの公開関数からも接続されていない

   **(2-iii)** グラフ $D$ において接続されている公開関数が過不足なく同じである

3. 以下を挿入できる公開関数がなくなるまで繰り返す.
   $CG_0, CG_1, ...$ に対して，グラフ $D$ において $CG_i$ に含まれている公開関数が接続している公開関数について，以下の条件を満たしていれば $CG_i$ に追加する.

   **(3-i)** 同一タスクグループに属し，かつ同一コアに割り当てられている

   **(3-ii)** 接続されている公開関数がすべて $CG_i$ に含まれている

4. $CGG$ に $CG_0, CG_1, ...$ を新たな一連実行グループとして追加する.

(2-i),(3-i) により，同じ集合 $CG_i$ に含まれる公開関数は同一タスクグループに属し，かつ同一コアに割り当てられているため，一連実行グループの条件 (a) を充たしている.

(1-ii),(2-iii) より，(2) の時点で $CG_i$ に含まれる公開関数は $CG_i$ 以外の同一の公開関数から接続されているので，外部依存公開関数集合が等しい．(3-ii) により，(3) で $CG_i$ に挿入する公開関数は $CG_i$ に含まれる公開関数のみから接続されてい

るため，$CG_i$ の外の公開関数からのパスはすべて (2) の時点で $CG_i$ に含まれている公開関数を経由する．これにより，$CG_i$ に含まれる公開関数は外部依存公開関数集合が等しくなるので，一連実行グループの条件 (b) を充たしている．

以上よりこのアルゴリズムによって得られる一連実行グループは，前節で示した一連実行グループの条件を満たしている．

ここで，コア割当て 1(図 3.5) の TaskG1 について，一連実行グループを抽出した場合について説明する． アルゴリズムの入力となるグラフは，図 3.7 で示される．まず (1) では,$PF$3-0，$FP$3-1 が (1-i) を充たすため，$CG = \{PF\text{3-0}, PF\text{3-1}\}$ となる．次に (2) では，$PF$3-0，$PF$3-1 はともに (2-i)(2-ii) を満たしているので $CG_0 = \{PF\text{3-0}, PF\text{3-1}\}$ となる．次に (3) では，$PF$3-0，$PF$3-1 から接続されている公開関数は $PF$4-0，$PF$4-1 だが，$PF$4-0 については (3-ii) を充たさないので $PF$4-1 のみを $CG_0$ に追加する．これ以上 $CG_0$ に公開関数を挿入できないので (4) に移る．(4) では $CGG$ に $CG_0 = \{PF\text{3-0}, PF\text{3-1}, PF\text{4-1}\}$ を追加する．

再び (1) に戻ると，$PF$4-0 は (1-ii) を充たすため $CG = \{PF\text{4-0}\}$ となる．$CG$ の要素はただ 1 つなので (2) では $CG_0 = \{PF\text{4-0}\}$ となる．(3) では $PF$4-0 はどの公開関数にも接続していないので何もしない．(4) では $CGG$ に $CG_0 = \{PF\text{4-0}\}$ を追加する．

以上により，$CGG = \{\{PF\text{3-0}, PF\text{3-1}, PF\text{4-1}\}, \{PF\text{4-0}\}\}$ となり，$\{PF\text{3-0}, PF\text{3-1}, PF\text{4-1}\}$，$\{PF\text{4-0}\}$ が一連実行グループとなる．

公開関数のコア割当てが異なる場合，一連実行グループ割当ての結果も異なる．コア割当て 2(図 3.6) はコア割当て 1(図 3.5) とは公開関数のコア割当てが異なるため，一連実行グループ割当ての結果も異なっている．

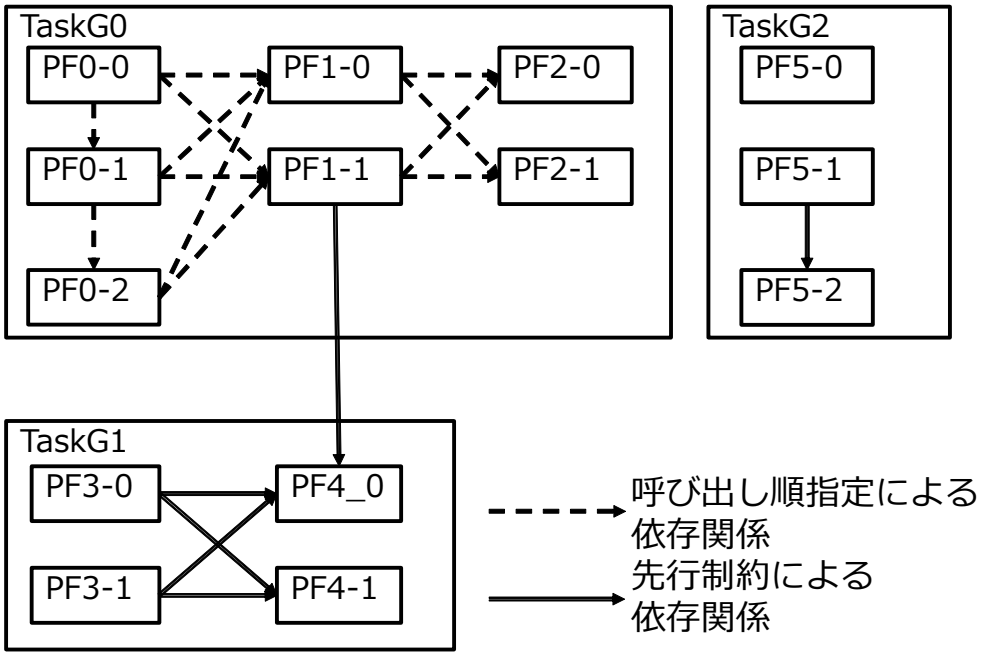


図 3.7: 図 3.2 の SW モデルから作成したグラフ

### 3.4.2　排他実行内部関数の実現

内部関数の排他制御を実現するために，ラッパ関数を生成する．ラッパ関数では，呼び出す公開関数のコア配置やタスク配置により適切な排他制御を行った後，指定された内部関数を呼び出す．ラッパ関数はモデルで指定された名前で生成され，公開関数の C 言語記述はこのラッパ関数を呼び出すように記述する．

実行オーバヘッドを低減するため，排他制御メカニズム（排他制御の実現方法）は呼び出し元の公開関数のコアへの配置やタスクグループへの所属のパターンに応じて可能な限り実行オーバヘッドが小さいものを決定する．配置パターンは以下の 3 通り存在する．

- 単一タスクグループ＆同コア
  呼び出す公開関数が同一のタスクグループのみに所属していて，同一コアに割り付けられている場合．

- 複数タスクグループ＆同コア
  呼び出す公開関数が他のタスクグループに所属していて，同一コアに割り付けられている場合．

- 複数コア
  呼び出す公開関数が他のコアに割り付けられている場合.

配置パターン，排他制御の種類によってどの排他制御メカニズムが選択されるかを表 3.1 に示す.

複数コアのタスクから呼び出され，リエントラント禁止が必要な排他実行内部関数の場合，あるタスクが排他実行内部関数を実行中に，他コアのタスクが排他実行内部関数を実行してはならないのでスピンロックおよび割込み禁止を用いて排他実行する必要がある．単一タスクグループ (同優先度) かつ同コアの複数タスクから呼び出され，リエントラント禁止が必要な排他実行内部関数の場合，あるタスクでその排他実行内部関数を実行されている間，他コアでは実行されず，また同コア中のその排他実行内部関数を実行する他のタスクにディスパッチされないので，排他メカニズムは不要である.

表 3.1: 配置パターン，排他制御ごとの排他制御メカニズム

| 排他制御の種類 | 単一タスクグループ&同コア | 複数タスクグループ&同コア | 複数コア |
|---|---|---|---|
| アトミック | 割込み禁止 | 割込み禁止 | 割込み禁止 |
| リエントラント禁止 | None | 割込み禁止 | SpinLock/割込み禁止 |
| アトミックかつリエントラント禁止 | 割込み禁止 | 割込み禁止 | SpinLock/割込み禁止 |

### 3.4.3　ランタイムコードの生成

ランタイムコードは C 言語のソースコードの形式で生成し，タスクの実体や依存関係や実行タイミングを満たす制御機構を実現する．コア割当て 1（図 3.5）から生成されるランタイムのフローを図 3.8 に示す．図 3.8 のフローは実際に生成されるランタイムを簡略化したものであり，またトリガー 0 に関係するオブジェクトについてのみを表している.

一連実行グループタスクの起動を制御するために，一連実行グループに実行状態，実行待ち状態，休止状態の 3 種類の状態を定義する．状態の遷移について，図

3.9 に示す．初期化時はすべての一連実行グループで休止状態である．実行状態は実行中もしくは他のタスクが終了したときに即座に実行することができる状態である．実行待ち状態は実行されるタイミングであるが，実行開始条件を満たしていない状態である．休止状態は処理が終了している状態である．

### ランタイムコード

生成するランタイムコードの詳細について説明する．

- トリガー関数

  トリガー毎にタスクグループ関数を呼び出すトリガー関数が生成される．各トリガーごとにカウンタ変数を持ち，トリガー関数が呼び出されるごとにカウンタをインクリメントする．カウンタ変数の値を確認し，実行されるタイミングであるタスクグループに関して，対応するタスクグループ関数を呼び出す．トリガー関数はユーザが記述したイベントに相当する割込みハンドラによって呼び出すことを想定している．

- タスクグループ関数

  タスクグループ毎にタスクグループ関数が生成される．各タスクグループごとにカウンタ変数を持ち，タスクグループ関数が呼び出されるごとにカウンタをインクリメントする．タスクグループ関数ではカウンタ変数の値を確認し，タスクグループ内のタイミングマネージャの周期とオフセットから，実行される公開関数を決定する．そして実行される公開関数を含む一連実行グループについて，一連実行グループタスクを実行待ち状態にする．その後，一連実行グループ起動判定関数を呼び出す．

- 実行開始フラグ

  一連実行グループはそれぞれその実行開始条件に対応するフラグ (実行開始フラグ) の集合 (実行開始フラグ集合) を持つ．実行開始フラグは初期化時ではすべて FALSE である．

- 一連実行グループタスク

  一連実行グループタスクでは．一連実行グループに含まれる公開関数について実行されるタイミングであるかどうかを確認し，そうであればその公開関

数を実行する．一連実行グループタスク終了時には一連実行グループを休止状態にする．

各公開関数終了時に，その公開関数が実行開始条件となる一連実行グループがあれば，その一連実行グループの対応する実行開始フラグを TRUE にする．その後，一連実行グループ起動判定関数を呼び出す．

- 一連実行グループ起動判定関数

  実行待ち状態の一連実行グループタスクについて，実行開始条件を満たしているかどうかを確認し，そうであれば一連実行グループタスクを実行状態にして起動する．その後，対応する実行開始フラグ集合内の実行開始フラグをすべて FALSE にする．

**依存関係の実現例**

図 3.8 により依存関係の実現方法を説明する．

コア割当て 1(図 3.5) の例では，CG1-1 はタイミングマネージャ TM3，公開関数 PF1-1 に対応する実行開始フラグ (TM3_Flag,PF1-1_Flag) を持つ．そのため，CG1-1 が起動されるためには一連実行グループ起動判定関数実行時に CG1-1 の実行開始フラグ集合 (CG1-1.ExectionFlag) が PF1-1_Flag, TM3_Flag を満たしている必要がある．

PF1-1 の終了後，CG1-1.ExectionFlag の PF1-1_Flag に該当するビットが TRUE となる．その後一連実行グループ起動判定関数が呼び出されるが，TM3_Flag が満たされていないため，CG1-1 は起動されない．次に CG1-0 で実行される公開関数のうち，PF3-0,PF3-1 が終了したとき，TM3 に含まれるすべて公開関数が終了したため，CG1-1.ExectionFlag の TM3_Flag に該当するビットが TRUE となる．その後一連実行グループ起動判定関数が呼び出され，CG1-1.ExectionFlag が TM3_Flag,PF1-1_Flag の両方を満たしているため，CG1-1 が起動される．

図 3.8 の例では TM3 に含まれるすべての公開関数が同じ一連実行グループ CG1-0 に含まれているため，後に実行される公開関数実行後に TM3_Flag を True とすればよい．しかし TM3 に含まれる公開関数が異なる一連実行グループに含まれる場合も存在する．その場合，タイミングマネージャごとに内包する公開関数に対応するフラグの集合を持たせ，公開関数が終了するたびにそのフラグの集合の該当ビットを True にする．その後，終了した公開関数を内包するタイミングマネージャ

に関して，すべてのフラグがセットされているかどうかを確認し，そうであれば TM3_Flag を True にする．

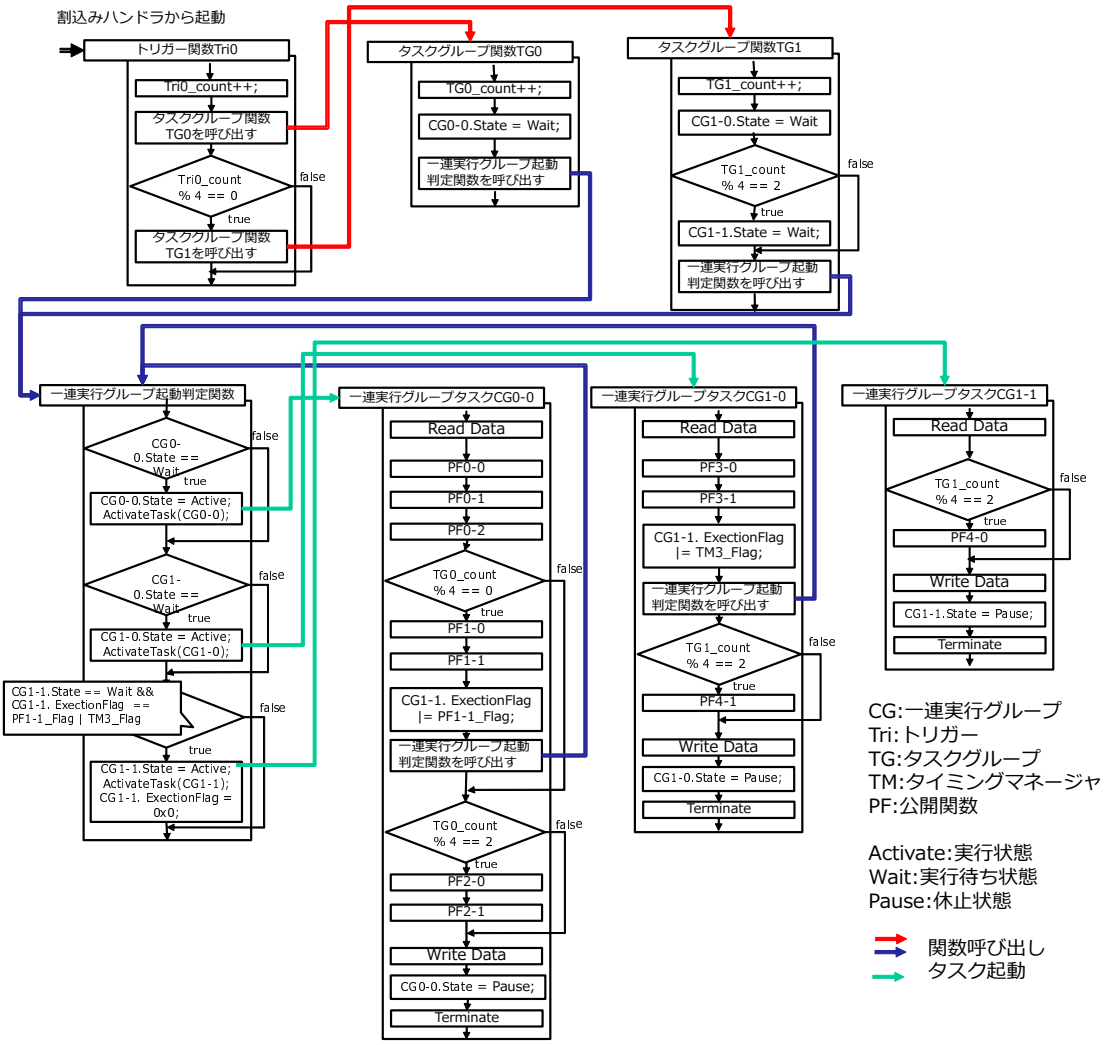


図 3.8: コア割当て 1 から生成されるランタイムのフロー

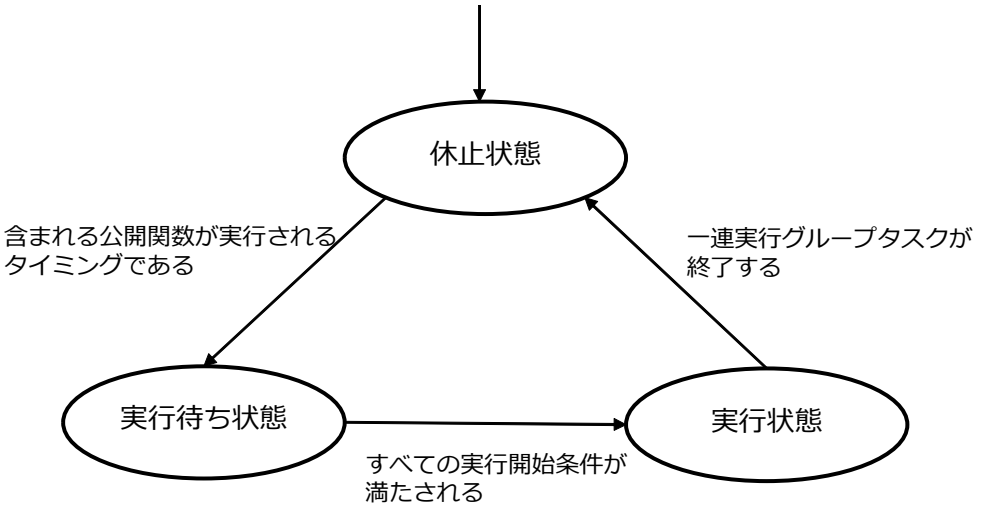


図 3.9: 一連実行グループの状態遷移図

## 3.5　評価

PMPF について，以下の 3 つの評価を実施する．

- （評価 1）記述量評価
- （評価 2）一連実行グループ抽出評価
- （評価 3）スケーラビリティ評価

評価 1 では，要件 (1) を満たしているかどうかを確認するため，マッピングやコア数を変更するための記述量について評価する．評価 2 は，要件 (7) を満たしているか評価するため，公開関数毎にタスク化した場合とのタスク数や実行時間を比較する．評価 3 では，要件 (1) より，短時間でランタイム生成が可能であるか評価するため，現状のパワトレアプリの規模と今後の機能増加を想定した規模で評価を行う．

評価環境としては，動作周波数 60MHz の Altera 社の NiosII プロセッサを用いた．コンパイラは GCC 4.1.2，AUTOSAR OS としては，TOPPERS/ATK2 SC1-MC 1.2.2[20] を用いた．

表 3.2: 評価用モデル

| | オブジェクト [個数] | | | | モデル [行数] | | |
|---|---|---|---|---|---|---|---|
| 評価 | コア | 公開関数 | タイミングマネージャ | タスクグループ | SW モデル | HW モデル | マッピング |
| 1 | 2 | 24 | 4 | 2 | 1597 | 34 | 28 |
| | 4 | | | | | 36 | 30 |
| 2 | 2 | 60 | 14 | 9 | 2542 | 34 | 58 |
| | 4 | | | | | 36 | 69 |
| 3 | 2 | 1,000 | 200 | 50 | 10,894 | 34 | 1,008 |
| | 2 | 2,000 | 400 | 100 | 21,744 | 34 | 2,008 |
| | 2 | 3,000 | 600 | 150 | 32,594 | 34 | 3,008 |
| | 2 | 4,000 | 800 | 200 | 43,444 | 34 | 4,008 |

### 3.5.1　評価用モデル

各評価で使用したモデルの情報およびモデルとマッピング情報の記述量を表 3.2 に示す.

評価 1 では，実際のパワトレアプリの一部をモデル化した記述を使用する．HW モデルは 2 コアと 4 コアを用意し，マッピングは人手で実施した．評価 2 で使用するモデルの詳細を，図 3.10，表 3.3 に示す．このモデルは実際のパワトレアプリの時間同期処理と回転角同期処理の一部分から抽出したものである．評価 2 でも評価 1 と同様にマッピング情報は人手で用意する.

評価 3 で使用するモデルは，公開関数，タイミングマネージャ，タスクグループ，トリガーの数が決められた個数となるようにし，各オブジェクトの依存関係および呼び出し関係をランダムで生成したものである．本評価に使用したモデルはすべて単一のトリガーからタスクグループが起動されることを前提としている．また，オブジェクトの依存関係はすべて呼び出し順指定のものとし，マッピングは公開関数をランダムに配置した.

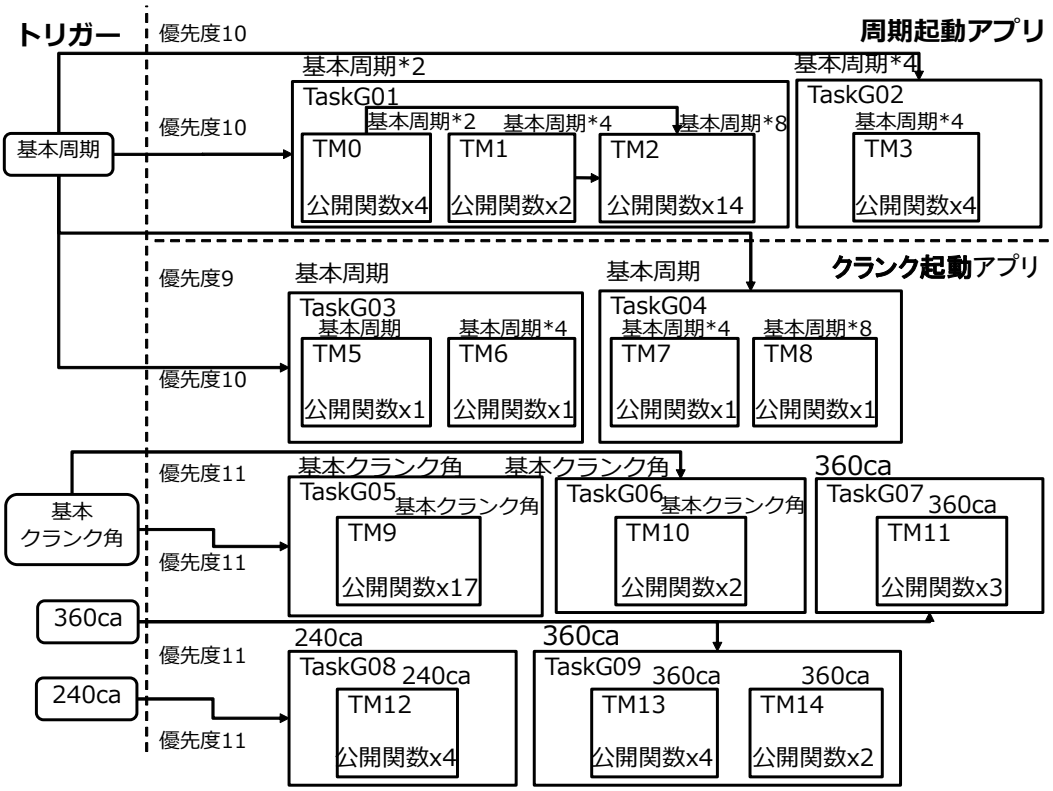


図 3.10: (評価 2) のモデル

### 3.5.2　（評価 1）記述量評価

評価用のモデルに対して，2/4 コアのハードウェアを対象にそれぞれ机上検討でマッピングを決定し，PMPF によりランタイムを生成して実行時間を測定した（Mapping1）．次に，実行結果から各コアの負荷が平準になるよう，2/4 コアそれぞれでマッピングを見直し（Mapping2），再評価を実施した．評価結果を表 3.4 に示す．表中の実行時間はある周期における全処理の終了までの時間を示す．

評価の結果，マッピングの変更は，生成されるランタイムの規模と比較すると十分小さい変更で実現できることを確認した．また，同一の SW モデルから，HW モデルとマッピングを変更することで，コア数の異なるランタイムが生成できることを確認した．さらに，モデル記述は生成されたランタイムと比較して，記述量が 10 分の 1 程度となることを確認した．

以上の結果より，提案手法は，人手による記述量を削減することができ，要件 (1) を満たすことができると考えられる．

表 3.3: (評価 2) 使用したモデルの詳細

| アプリ | タスクグループ | 実行間隔 | 公開関数数 | 一連実行グループ抽出時のタスク数 (2 コア) | 一連実行グループ抽出時のタスク数 (4 コア) |
|---|---|---|---|---|---|
| 時間同機処理 | TaskG1 | 基本周期*2 | 4 | 9 | 15 |
| | | 基本周期*4 | 2 | | |
| | | 基本周期*8 | 14 | | |
| | TaskG2 | 基本周期*4 | 4 | 3 | 3 |
| | TaskG3 | 基本周期 | 1 | 1 | 1 |
| | | 基本周期*4 | 1 | | |
| | TaskG4 | 基本周期*4 | 1 | 1 | 1 |
| | | 基本周期*8 | 1 | | |
| 回転角同期処理 | TaskG5 | 基本回転角 | 17 | 1 | 1 |
| | TaskG6 | 基本回転角 | 2 | 1 | 1 |
| | TaskG7 | 360ca | 3 | 1 | 1 |
| | TaskG8 | 240ca | 4 | 1 | 1 |
| | TaskG9 | 360ca | 6 | 2 | 2 |
| 合計 | | | 60 | 20 | 26 |

### 3.5.3　（評価 2）一連実行グループ抽出評価

一連実行グループ抽出の効果を評価するため，公開関数ごとにタスク化する手法との比較を行った．

回転角同期の処理に関しては，1000rpm, 2000rpm, 3000rpm の場合について評価を行った．

計測区間を最初の時間同期割込みが発生してから，基本周期の 2 倍の時間が経過するまでの間とした．すべての時間同期理処理のオフセットを 0 としているため，最初の時間同期割込みによってすべての時間同期処理が実行される．また回転角同期処理も，最低 1 回は計測区間中に実行されるようにオフセットを調整した．

評価結果を表 3.5 に示す．評価の結果，提案手法は，公開関数ごとにタスク化する手法と比較してタスク数が少なく，タスク起動・終了に必要なオーバヘッドが削減でき，回転数が 3000rpm の場合以外で 2，4 コアの両方において実行時間が削減

表 3.4: （評価 1）記述量評価の評価結果

| Mapping | コア数 | マッピング変更 [行数] | ランタイム [行数] | 実行時間 [$\mu$sec] |
|---|---|---|---|---|
| 1 | 2 | - | 16,295 | 815 |
| | 4 | - | 17,477 | 682 |
| 2 | 2 | 3 | 15,674 | 757 |
| | 4 | 9 | 17,202 | 494 |

表 3.5: （評価 2）一連実行グループ抽出評価の評価結果

| コア数 | タスク化単位 | タスク数 | 回転の回転数 [rpm] | タスク起動回数 | 計測区間に対する実行時間割合 [%] | 削減率 [%] | 破棄されたタスク起動要求数 |
|---|---|---|---|---|---|---|---|
| 2 | 公開関数 | 60 | 1000 | 134 | 67 | - | 0 |
| | | | 2000 | 203 | 89 | - | 0 |
| | | | 3000 | 190 | 90 | - | 153 |
| | 一連実行グループ | 20 | 1000 | 28 | 59 | 14 | 0 |
| | | | 2000 | 34 | 75 | 18 | 0 |
| | | | 3000 | 46 | 90 | 0.001 | 0 |
| 4 | 公開関数 | 60 | 1000 | 129 | 65 | - | 0 |
| | | | 2000 | 202 | 85 | - | 0 |
| | | | 3000 | 186 | 82 | - | 163 |
| | 一連実行グループ | 26 | 1000 | 31 | 51 | 26 | 0 |
| | | | 2000 | 43 | 69 | 22 | 0 |
| | | | 3000 | 51 | 85 | -4 | 0 |

できることを確認した．回転数が 3000rpm のときに公開関数単位でのタスク化のほうが実行時間が少なくなっているのは，回転角割込みの間隔が短くなり，いくつかの回転角期タスクが次の回転角割込みが入るまでに終了せず (デッドラインを満たせず)，タスクの起動要求が破棄されてしまっているためである．

以上の結果から提案手法は，必要なタスク数を抑え，タスク起動等のオーバヘッドを小さくすることで，要件（7）を満たしていることを確認した．

### 3.5.4　（評価 3）スケーラビリティ評価

表 3.2 に示した，評価 3 用のモデルを合成した結果を表 3.6 に示す．ランタイム生成にかかる時間は現状のパワトレアプリの規模である公開関数が 1000 個の場合では 2 分半程度であり，短時間でランタイム生成が可能である．ランタイム生成にかかる時間は公開関数数 $n$ に対して $O(n^2)$ で増加するものの，4000 個の場合で

あっても 2 時間半程度であるため，ツールの実行時間に関しても要件（1）を満たし，現実的な時間でランタイム生成が可能であることを確認した．

表 3.6: （評価 3）スケーラビリティ評価の評価結果

| 公開関数の数 | 生成時間 [s] | | | ランタイム [行数] | | 一連実行グループ数 |
|---|---|---|---|---|---|---|
| | 一連実行グループ抽出 | コード生成 | 合計 | C 言語コード | コンフィグレーション | |
| 1000 | 11 | 143 | 158 | 52,751 | 3,168 | 450 |
| 2000 | 42 | 1,112 | 1,167 | 103,523 | 6,409 | 913 |
| 3000 | 97 | 4,292 | 4,415 | 158,230 | 10,112 | 1,442 |
| 4000 | 184 | 9,236 | 9,532 | 208,389 | 12,877 | 1,837 |

## 3.6　関連研究

文献 [25] ではアプリを並列実行するために，並列化コンパイラを使用して細粒度の並列性を抽出して，マッピングとラインタイムを生成するアプローチを提案している．しかし，現状のパワトレアプリは，すでに 1000 個程度に処理が分割されており，数個のコアに対しては，十分な並列性があるため，現時点での必要性は低い．

文献 [4][26][27] では，本研究と同様に，既存のパワトレアプリを元に処理をタスクにマッピングし，ランタイムを生成するツールを提案している．提案ツールでは制御依存関係などの情報から処理を AUTOSAR OS のタスクに割り付ける．本研究と比較すると，タスクの抽出は行わず，人手でタスクをコアへ割り当ける必要ある．また，コア間の処理の依存関係を満たす様な機構は持たない．

文献 [3] では常にシングルプロセッサ時に同じタスクに所属していた処理 (同一グループ処理) のみ並列で実行し，全コアで同一グループ処理が終了するまで待ち合わせ，次の同一グループ処理を実行する．本研究と比較すると，既存のシングルコアタスクスケジューリングが使用出来るためシングルコアからの移植性は高いが，同一グループ処理でなければ並列実行出来ないため，本研究と比べてマルチコア化による速度向上の恩恵は少なくなると思われる．

文献 [28][29][30] では，処理のコアへの動的な割り当てを提案しているが，パワトレアプリの場合，信頼性の確保のために，処理の再現性が求められるため，本研究で実現したような静的な処理の割り当てが必要となる．

## 3.7　AUTOSAR RTE との比較

本章では AUTOSAR 仕様のランタイム環境生成ツールである RTE ジェネレータと PMPF について比較を行う．

### 3.7.1　AUTOSAR RTE

AUTOSAR RTE(Run-Time Environment) とは AUTOSAR アプリに提供されるランタイム環境である．AUTOSAR RTE では，処理をランナブルとして記述し，機能ごとのランナブルの集合を SWC(SoftWare Component) と定義する．排他実行についてはランナブルの属性を設定することで実現可能である．

AUTOSAR RTE を用いた実装を行う際には，開発者は以下について定義し，RTE ジェネレータを実行する．

- タスクの定義
- タスクのコアへのマッピング
- ランナブルのタスクへのマッピング
- 排他実行の実現メカニズム

RTE ジェネレータはこれらの定義と SWC の定義を読み込み，タスク本体等の実装を生成する．

PMPF における公開関数と AUTOSAR RTE におけるランナブルはともに処理の記述であるため，本章ではランナブルは公開関数に対応するものとして定性的な比較を行う．

### 3.7.2　処理のタスクへのマッピングに関する比較

AUTOSAR RTE では，タスクの定義とランナブルのタスクへのマッピングについては，設計者が行う必要がある．一方，PMPF では 3.4.1 節でも示したように，

公開関数のコア割当てと依存関係に応じて，必要なタスクが最小となるようにタスクの定義と公開関数へのマッピングを自動で行う．従って，PMPF の方が設計効率において優れていると考えられる．

### 3.7.3　コア割り当てに関する比較

現状の AUTOSAR RTE では仕様では，同一の SWC に所属しているランナブルは同一のコアにのみ割り当てることができるという制約があるが，PMPF にはそのような制約はない．現状のパワートレインにおいては，マルチコアの有用活用するために同一機能の複数の処理をコアを跨いで配置する必要が有るが，AUTOSAR RTE では実現することができない．

### 3.7.4　排他実行に関する比較

AUTOSAR RTE 仕様では，ランナブルの排他実行はサポートしているものの，割込み禁止，スピンロックなどの排他メカニズムのうち，どれを使用するかは開発者が指定する必要がある．一方，PMPF では内部関数の排他制御については，コア割当てと排他の目的を指定することにより適切な排他メカニズムを自動的に選択してラッパ関数を生成する．このことから，PFPF の方が開発効率が高いと考えられる．

### 3.7.5　AUTOSAR RTE と提案手法の組み合わせ

前述の様に，PMPF は AUTOSAR RTE では自動化されていない，

- タスク化単位の抽出
- 排他メカニズムの選択

を自動化している．AUTOSAR RTE による設計に対して，PMPF の上記機能を取り出して組み合わせることにより，設計効率を上げることが可能である．

## 3.8　おわりに

本章では，パワトレアプリのランタイムの自動生成を実現するツールである PMPF について述べた．PMPF を用いることにより，単一のソフトウェア記述からコア数や処理のマッピングが異なるランタイムを生成することが可能である．また，タスク数の増大を抑えるためのタスクの抽出アルゴリズムを提案した．課題 (a) は，HW モデルやマッピング情報を変更するだけで異なるマルチコアアーキテクチャに対応することができるため対処することができている．課題 (c) は，待ち状態を用いた同期ではなく，タスクの起動によってタスクの実行依存関係の成立を実現しているため対処することができている．課題 (d) は，一連実行グループを用いることで対処することができており，課題 (g) は PMPF によるランタイムの自動生成により対処することができている．評価により，少ない記述量でコア数や処理のマッピングを変更することを示した．また，大規模なパワトレアプリに対しても現実的な時間で適用可能であることも示した．

# 第4章　マルチコア車載制御システムにおける最悪余裕時間解析手法を用いたマッピング決定

## 4.1　概要

本章では，1 章で述べた研究 (II) について説明する．

パワトレアプリは多数の実行依存関係を持った処理と処理間で共有するデータ（共有データ）で構成されており，マルチコアアーキテクチャに実装する場合にはそれらの処理のコア配置と共有データのメモリ配置を決定する必要がある．コア配置を決める要因として CPU 利用率やタスクの最悪応答時間などが挙げられる．パワトレアプリの処理の数は 1000 個程度であり，これらのコア配置のパターン数は多く，全ての配置パターンを実装して評価するのは現実的ではない．また，実機による有限時間の測定で得られる最も長い応答時間が最悪応答時間となるとは限らない．そのためすべてのタスクや ISR の実行の組み合わせを考慮した静的な解析を用いてコア配置の候補を絞り，それらの候補に対して実機で評価する方法を行うのが望ましい．

パワトレアプリの処理は時間制約にしたがって動作し，重要な処理が時間制約を満たさない場合，人命に関わる重大な事故に繋がる可能性がある．これがすべてのタスクや ISR の実行の組み合わせを考慮した解析を行う理由である．本章では処理のコア配置のための指標として，余裕時間（デッドラインと最悪応答時間の差）の最小値を求めることにより，リアルタイム性を保証する．シングルコアシステムについてはすでにリアルタイム性を解析する手法が提案されている（2.4.1 節を参照）．この手法によって処理の実行時間と周期，オフセットから各タスクの最悪応答時間を計算することができる．マルチコアシステムでは処理のコア配置と共有データのメモリ配置の関係によってコア間排他やメモリアクセスによる実行オーバヘッドが変化する．この変化を考慮しない場合，最悪応答時間が実機実

行より短く算出される場合がある．

しかし実際にはスピンロックのアイドル時間などパワトレアプリの構成次第では解析が困難であるオーバヘッドがある．そのような場合に最低限どのようなオーバヘッドを考慮すれば解析によって得られる最悪応答時間が実機実行で得られる応答時間の最大値より長くなるかを見極めることが必要となる．

本章ではパワトレアプリの評価ソフトについて，これらのオーバヘッドを考慮した最悪余裕時間解析手法を提案する．具体的には，まず組合せ数を削減するために処理を幾つかの集合に分割する．次にその集合単位で処理をコアに配置するすべてのパターンに対して共有データの配置を行い，その後最悪応答時間の解析を行い，処理が時間制約を満たさないパターンを排除する．最後に，残った配置パターンに関して最悪応答時間から余裕時間を計算し，その中から最小値を算出する．

研究 (II) によって，1 章で述べた課題 (e)(f)(h)(i)(j) に対応しつつ，現実的な時間で解析を実施し，マッピングの候補の選定をすることが可能となる．

## 4.2　タスクのコア配置および共有データのメモリ配置の方針

本章で対象とするソフトウェア構成は 2.1.1 節で示したとおりである．マルチコアで対象パワトレアプリを実行する場合は公開関数のコア配置と共有データのメモリ配置を決定する必要がある．公開関数を実行するタスクの応答時間がデッドラインを超えないように公開関数のコア配置を決定する必要がある．また，共有データはできるだけアクセス時間が短くなるようにメモリに配置する必要がある．そのためアクセス頻度の高いコアの LMEM に配置したり，各コアでのアクセス頻度が偏っていない場合には SMEM に配置するなどの工夫が必要である．

## 4.3　公開関数グループ（PF グループ）

公開関数は現状 1,000 個程度であり，それらのコア配置パターンは組合せ爆発によって膨大な数になってしまい，リアルタイム性解析に必要な計算時間が膨大となることが予測される．例えば 1,000 個の公開関数を 2 コアに配置するパターン数は約 $1.07 \times 10^{301}$ にも及ぶ．そのため，公開関数のコア配置パターンを削減する仕組

みが必要である．ここで，設計者が公開関数の集合を適切にいくつかのグループに分割し，そのグループ単位でコア配置を行うようにする．その公開関数の集合を公開関数グループ（PF グループ）と呼ぶ．図 4.1 は公開関数の集合を PF グループに割当てたときの例を示している．同 PF グループに含まれ，同セットマネージャから呼び出される公開関数を一つのタスクで実行する．

### 4.3.1　公開関数グループへの分割の目的

公開関数間には実行依存関係やデータ依存関係の 2 つ依存関係が存在する．

- 実行依存関係: 公開関数 A が先に実行され，終了した後でないと公開関数 B が実行できないという制約がある場合，公開関数 A と公開関数 B は実行依存関係を持つと呼び，この時公開関数 A を依存元，公開関数 B を依存先と呼ぶ．
- データ依存関係: 公開関数間に共通して使用される共有データが存在するという関係である．各データに関してデータの供給元 (センサー値を書き込む公開関数など) とデータの利用先 (センサー値を使用して処理を行う公開関数など) が存在する．データの利用先はデータの更新を考慮することなく，実行時にデータを読み込んで処理を行う．

文献 [31] で示されているように，待ち状態を使用することは好ましくないため，それぞれの処理を別のタスクとして実装し，依存元の処理をもつタスクが終了した後に依存先の処理を持つタスクを起動するようにしなければならない．しかしこのようにするとタスク数が増大し，タスク起動および終了に要する OS オーバヘッドが大きくなってしまうという問題がある．また異なるコアに配置されたタスク間に実行依存関係がある場合は制御構造が複雑になり，解析に要する時間や工数が増大する．逆にそのようなタスク間に実行依存関係がなければ，他コアのタスクの終了待ちによる遅延がなくなるため，コアごとに独立した解析が可能となる．これらの理由から互いに実行依存関係を持つ公開関数は同じコアに配置されるべきであるため，このような公開関数を同じ PF グループに割当てるようにする．

### 4.3.2　公開関数グループへの分割方法

公開関数の集合を公開関数グループへ分割する際に，以下の 3 つの評価観点について考慮する．

- PF グループ単位の平均処理負荷
  PF グループに含まれる公開関数の実行時間を公開関数の実行周期で割った値の合計である．これは新しい機能を追加する余地があるかどうかの指標となる．極端に大きすぎたり小さすぎたりすると選択できる PF グループのコア配置パターンが少なくなるためそれぞれ平均的になるようにする．

- ある PF グループを単一のコアに配置したときのコア最悪余裕時間
  単一のコアに PF グループを配置したしたときに極端にコア最悪余裕時間 (4.4.1 節参照) が短い場合にはその PF グループだけで 1 つのコアを専有することが決まってしまい，コア配置の幅を狭くしてしまうため，これを考慮する．

- 同一 PF グループのみによってアクセスされる共有データ数
  このような共有データはその PF グループが配置されるコアのローカルメモリに配置すればよくなるため，スピンロックが不要になり，バスを介してアクセスされなくなる．従ってこれが多いほどスピンロックオーバヘッドとメモリアクセスオーバヘッド (4.6 節参照) が削減できる．

次に，公開関数を PF グループに割当てる方法を示す．

(i). すべての公開関数からなる集合を集合間で公開関数の実行依存関係を持たないように分割する

(ii). (i) で得た集合を一定の個数の PF グループに前述の評価観点のパレート解となるように割当てる

(iii). (i) で得た集合すべてを PF グループに割当てたときに PF グループへの割当ては終了する

PF グループの数は実装対象のハードウェアのコア数より多くした上で組み合わせ数が膨大にならないような値にする必要がある．

### 4.3.3　PF グループの有用性

公開関数を PF グループに割り当てる方法の有用性を示すために，以下の 2 つとの比較を行う．

(I). 公開関数単位でのコア配置

(II). 公開関数をランダムにグループ化

（I）（II）の場合と PF グループを用いる場合について，以下の 3 つに関して定性的な比較を行う．

(a). 他コアに配置された公開関数の終了待ちの有無

(b). 最悪応答時間解析に要する時間

(c). 解析によって得られるシステム最悪余裕時間

（a）について，(I)（II）の場合ではコアを跨ぐ公開関数の実行依存関係が残るため，他コアの公開関数の終了待ち時間が発生する．そのためコアごとに独立した解析はできなくなる．PF グループを用いる方法では，互いに実行依存関係を持つ公開関数は同じ PF グループに割当てられるため，コアを跨ぐ公開関数の実行依存関係なくなり，他コアの公開関数の終了待ちは発生しない．

（b）について，(I）と比較すると，(I）は公開関数単位での配置であるため，公開関数のコア配置パターンが非常に多くなり，組み合わせ爆発が起こるため，現実的な時間で最悪応答時間を解析できない可能性がある．PF グループを用いれば，組み合わせ数を削減し，現実的な時間での解析が可能となる．(b）について，(II）と比較すると，(II）では実行依存関係を解析し，他の公開関数の終了待ちの時間を計算するプロセスが必要となり，そのプロセスの分だけ PF グループを用いる方法と比較して最悪応答時間解析に要する時間が増加する．

（c）について，(I）（II）は他のコアの公開関数の終了待ちの分だけ公開関数を PF グループを用いる方法と比べてシステム最悪余裕時間（4.4.1 節を参照）が悪化する．また，コアごとに独立して解析を行うことができないので，解析が複雑となる．ただし，(I）では PF グループを用いた方法よりも柔軟な公開関数のコア割り当てが可能なので効率のよいパターンを見つけることができる可能性がある．

## 4.4　最悪余裕時間解析の要件と方針

本章では車載マルチコアシステムに実装する対象パワトレアプリに対する最悪応答時間解析手法が充たすべき要件を示し，その要件を充たすような最悪応答時間解析を行うための方針について説明する．

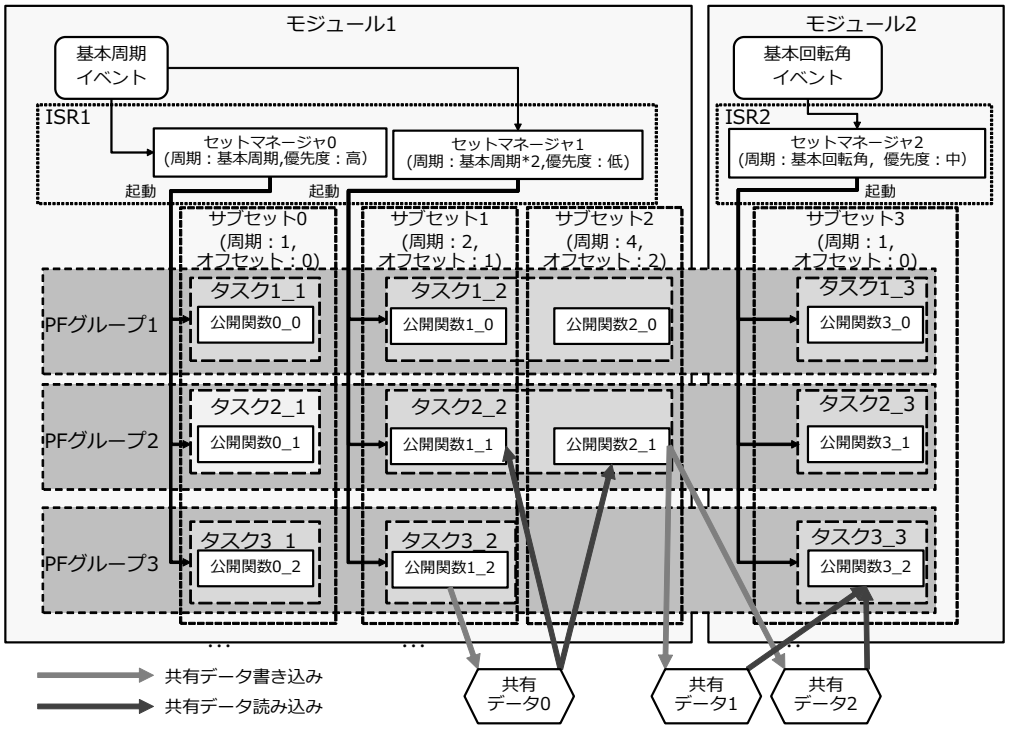


図 4.1: 公開関数グループ（PF グループ）

### 4.4.1　要件

タスクごとのリアルタイム性を解析するために，最悪応答時間とデッドラインを比較する．あるタスクについて，そのタスクのデッドラインと応答時間の差を余裕時間と呼ぶ．余裕時間に関して以下のように用語を定義する．

- タスク最悪余裕時間: あるタスクが取りうる最小の余裕時間
- コア最悪余裕時間: あるコアに配置されているすべてのタスクのタスク最悪余裕時間の中での最小値
- システム最悪余裕時間: すべてのコアのコア最悪余裕時間の中での最小値

システム最悪余裕時間の例を図 4.2 に示す．システム最悪余裕時間が小さいコア配置ほど，デッドラインミスする可能性が高い．そのため，実機で評価するコア配置パターンとしてはシステム最悪余裕時間が長いものを選択する．

ここで，本手法が充たすべき要件を以下に示す．

(1) 現実的な時間（数分〜数時間）で解析可能なこと

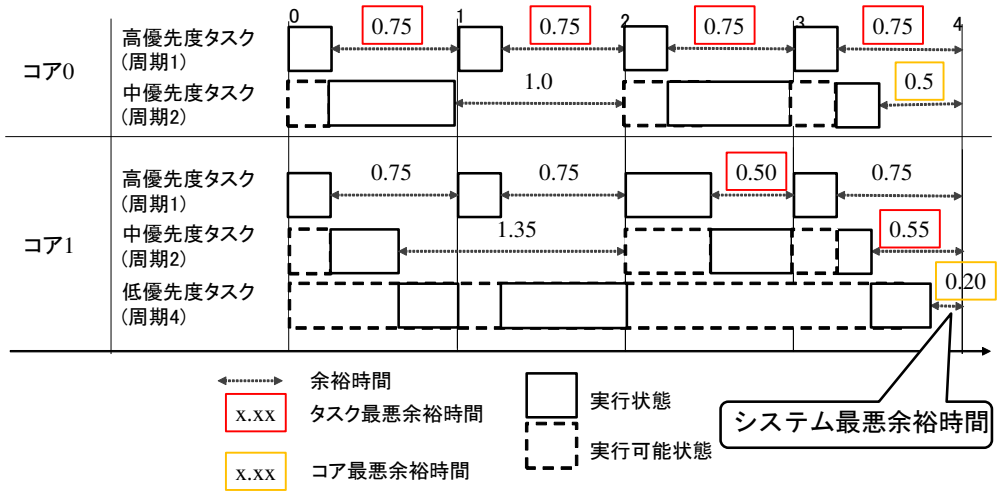


図 4.2: システム最悪余裕時間の例

ソフトウェア，ハードウェアが変更されるたびに最悪応答時間解析を再実行する必要があるため，一回あたりの最悪応答時間解析時間が短くなるようにする必要がある．

(2) すべてのタスクや ISR の実行の組み合わせにおける最悪応答時間を算出できること

車載システムを安全に実行するためにはすべてのタスクや ISR の実行の組み合わせにおける最悪応答時間を導き，リアルタイム性を保証する必要がある．

(3) 解析結果と実機実行結果に相関があること

本手法での解析結果においてシステム最悪余裕時間が長いものについて実機においても検証を行うため，相対的に良い結果となるような PF グループの配置パターンを見つけることができればよい．したがって，本手法では絶対的な解析値の正しさは必要ない．

(4) 解析結果が実機実行より厳しい結果となること

最悪応答時間解析の結果リアルタイム性が保証された PF グループの配置パターンを実機に実装したときにタスクのデッドラインミスが起こるようなことがあってはならないということが本要件を設定した目的である．スピンロックのアイドル時間など実際のパワトレアプリの構成次第では解析が困難であるオーバヘッドがあるが，それらの影響が小さく，考慮しなくても本要件を充たすことができるのであれば，そのようなオーバヘッドは無視することが

できる．したがって，要件 (4) を満たすためにどのようなオーバヘッドを考慮しなければならないかを明らかにする必要がある．

ここで，本要件を充たすための条件を以下に示す．時刻 0 から $t$($t$ は正の実数) まで実機において計測したときの全タスクの中で最小の余裕時間を $s_t$ とする．このとき，すべての $t$ に対して以下が成り立つ．

$$s_t \geq WCST \tag{4.1}$$

ここで WCST は本論文の解析手法で得られるシステム最悪余裕時間である．実機による実行時間が長いほどより短い余裕時間を見つけることが可能であるので $s_t$ は $t$ に従って単調減少する．そのため，ある時刻 t まで計測して上記の式が成り立つならば，時刻 $t$ 以下についても上記の式は成立する．

また，すべての配置パターンにおいてリアルタイム性が保証できない場合は，機能を減らしたり，いくつかの処理の実行周期を長くしたりして全体の負荷を下げる必要がある．そのような修正の後に再度解析を行い，リアルタイム性が保証できる配置パターンが見つかったとする．そのパターンを実機に実装して評価したときにデッドラインを充たすことができていなければ再び前述の修正を実施して再度解析を実施しなければならない．このような手間が必要なくなるように，本要件が必要となる．

### 4.4.2　方針

ここでは，4.4.1 節で述べた要件を充たすための方針について説明する．

要件（1）については，不要な解析を打ち切ることで解析に要する時間の短縮を図る．具体的にはデッドラインミスするタスクが見つかった時点で解析を打ち切るようにする．

要件（2）については，Maximum Interference Function（MIF）[6] を用いることで理論的な最悪応答時間を導出し，その値を用いてシステム最悪余裕時間を計算する．MIF はタスクの実行時間が周期（フレーム）ごとに変化するマルチフレームタスク（MF タスク）[22] のリアルタイム性保証を効率よく行うための手法である．パワトレアプリ評価ソフトにおけるタスクはサブセットによって実行するタスクがフレームごとに変化するため，MF タスクとして扱うことができる．タスク $\tau \in T$（$T$ はすべてのタスクの集合）とすると，タスク $\tau$ の MIF $M_\tau(t)$ はタスク $\tau$ が自分より優先度の低いタスクの実行を妨げる最大時間を表す関数であり，タスク

$\tau$ が各フレームから開始したときの時刻 0 から $t$ までの CPU 利用時間の最大値を取る関数である．‘ ここでタスク $\tau$ の最大応答時間は，タスク $\tau$ の実行時間と，タスク $\tau$ より優先度の高いタスクの時刻 0 からタスク $\tau$ のデッドラインまでの MIF の合計値である．

要件（3）（4）については，OS やハードウェアのオーバヘッドを考慮した解析を行う．

## 4.5　対象パワトレアプリのランタイム

本節では対象パワトレアプリのランタイムについて述べる．

対象パワトレアプリは RTOS 上のタスクおよび ISR からなるランタイムとして実行される．対象パワトレアプリのランタイムを図 4.3 に示す．対象パワトレアプリのタスクは優先度ベースのプリエンプティブスケジューリングにより実行される．起動，終了，プリエンプションによるタスクおよび ISR の切り替え時にはコンテキストの保存・復帰などの OS オーバヘッドが発生する．

時間周期または回転回転角などのイベントを契機に ISR が実行される．時間同期のタスクは同一の時間同期の ISR によって起動される．回転角同期の ISR およびタスクは実際には非同期的に実行されるが，エンジンがある一定の回転数で動いているときの周期を用いることで，時間同期のタスクおよび ISR と同様に扱うことができる．ISR ではセットマネージャによって指定された周期にしたがって各タスクを起動する．ISR の実行時間は起動するタスクを選択するための前処理にかかる時間とタスク起動による OS オーバヘッドに分かれており，ISR の実行時間の大きな割合を占めているのは後者である．

タスクは公開関数処理ブロック（図 4.3 中 PF），共有データ読み込み処理ブロック（図 4.3 中 R），共有データ書き込み処理ブロック（図 4.3 中 W）で構成されている．公開関数実行ブロックではタスクに含まれている公開関数をその公開関数が所属するサブセットが指定する周期とオフセットによって決められたタイミングで実行する．対象パワトレアプリではタスクが使用している共有データの内容はタスク実行中に別のコアに配置されたタスクによって変更されないようにしなければならない．そのためにタスクの先頭の共有データ読み込みブロックで使用する共有データをローカル領域に読み込み，タスクの末尾の共有データ書き込みブロックで共有データの変更を書き込むようにしている．共有データ読み込みブロックおよび共有データ書き込みブロックでは各共有データに対してスピンロックも

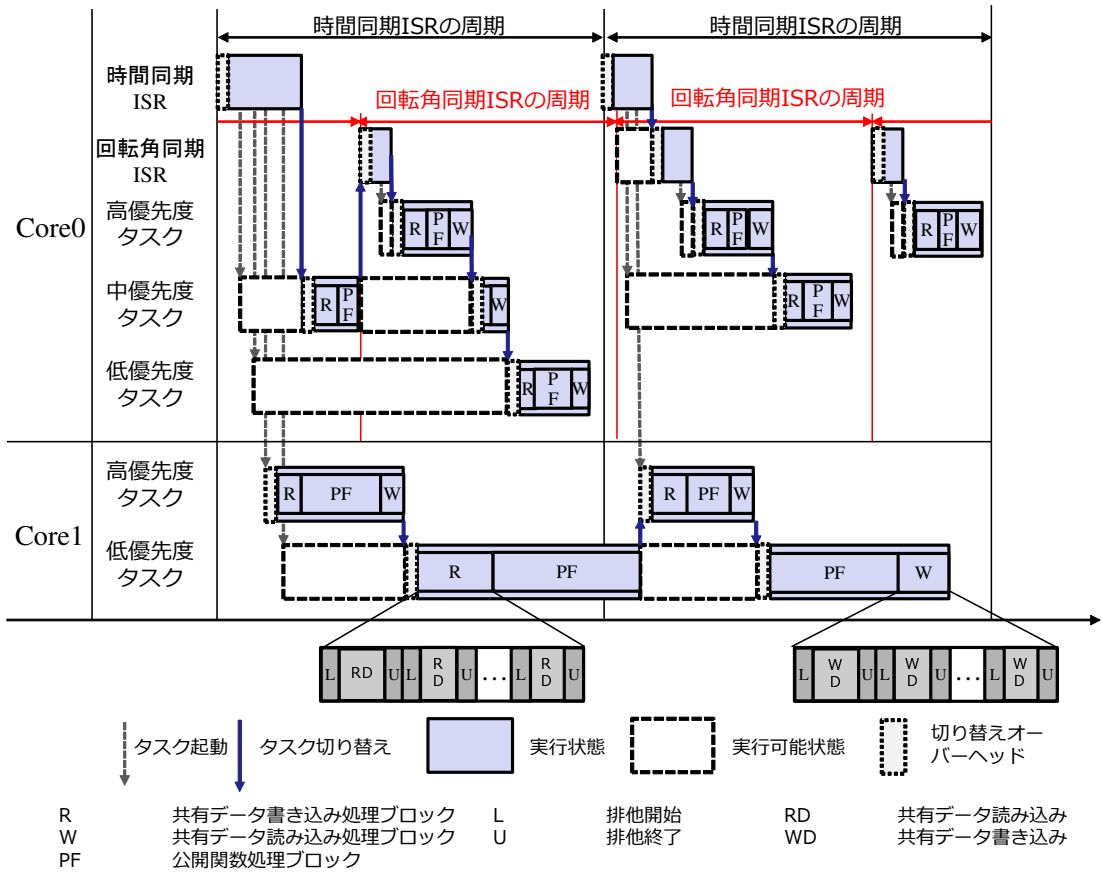


図 4.3: 対象パワトレアプリのランタイム

しくは割込み禁止を用いて排他制御を行う．

## 4.6　考慮する実行オーバヘッド

本章では 4.4.1 節で述べた要件を充たすために必要となる車載マルチコアアーキテクチャや対象パワトレアプリの情報を明らかにし，それらの情報を含むような性能評価用モデルについて述べる．

マルチコアシステムでは共有データのメモリ配置の結果によって排他制御によるオーバヘッドやメモリアクセス時間が大きく変化する．よってシングルコアでの最悪実行時間解析とは異なり，これらについてタスクの実行時間とは別に考慮する必要が出てくる．以下に要件（3）（4）を充たすために公開関数の実行時間とは別に本手法で考慮しなければならないオーバヘッドを示す，オーバヘッドにはハードウェアによるものと OS によるものが存在し，ハードウェアオーバヘッドに

は共有データアクセスによるメモリアクセスオーバヘッド，OS オーバヘッドにはスピンロックオーバヘッド，タスク切り替えオーバヘッド，タスク起動 ISR オーバヘッドがある．

### 共有データアクセスによるメモリアクセスオーバヘッド

共有データは数が多く，複数のタスクによって頻繁にアクセスされるため，共有データが配置されているメモリへのアクセス時間の合計は大きい．シングルコアで実行する場合は共有データが配置されるメモリは一定であるため公開関数の最大実行時間にメモリアクセス時間を含めることが可能であるが，マルチコアの場合，共有データが配置されるメモリと共有データへアクセスするタスクのコア配置によってメモリアクセス時間が異なる．公開関数のコア配置に対し適切に共有データをメモリに配置する必要があるが，その場合公開関数の実行時間が共有データの配置パターンによって変動するので，共有データへのアクセス時間は公開関数の最大実行時間とは別に考慮する必要がある．

また，共有データを配置するときに LMEM を可能な限り用いることでメモリアクセス時間を削減する．解析を行う際には，共有データをアクセスする公開関数のコア配置とメモリアクセスのオーバヘッド時間を用いて各 PF グループの配置パターンごとに共有データのメモリへの適切な配置を決定する必要がある．本手法ではランタイムの仕様に従い，各公開関数が共有データにアクセスする回数は読み込み・書き込み共に 1 回までであるという前提で解析を行う．

### スピンロックオーバヘッド

スピンロックにかかるオーバヘッドはスピンロックの獲得，解放を行うために必要な処理を実行する API の実行時間と他のコアがスピンロックを解放するまでのアイドル時間で構成される．スピンロックによるオーバヘッドはコア数が多いほど性能に与える影響が大きい [32]．今後，コア数が増加したときを考慮して本手法ではスピンロックのオーバヘッドの影響を考慮できるようにする．ただし，すべての共有データについてスピンロックを用いる必要はなく，複数のコアからアクセスされる共有データに関してのみスピンロックによる排他制御を行えばよい．

ここで，本来であれば API にかかる時間とアイドル時間の両方について考慮すべきであるが，競合状態によるアイドル時間はタスクの実行状況によって大きく変化するため，解析を行うことは難しい．すべてのスピンロックが衝突するという仮

定で解析を行うという方法も考えられるが，その場合は解析結果が悲観的となり実機で実際に実行したときの結果と乖離する可能性が高い．ここで，要件（4）を充たすような解析が可能ならば，必ずしもパワトレ評価ソフトの性能に与える影響が少ない要因については考慮する必要はない．対象パワトレアプリのランタイムはタスクの先頭，末尾に共有データアクセスを集約させることでスピンロックなどの排他制御が必要な区間や回数を抑えている．このようにスピンロックの獲得・解放頻度が低くなるような実装の仕方をしていれば，一般的にコア数が少ないほどスピンロックが衝突する確率は低くなるため，2〜4 コア程度で実行する分にはスピンロックのアイドル時間は短くなると予想できる．コア数が多くなれば，考慮の必要性が出てくる可能性はあるが，そのような場合は 2.1.2 でも述べたようにアーキテクチャの構成が大きく変化すると考えられるため，本章では対象とせず，今後の課題とする．

したがって，本手法ではスピンロックオーバヘッドは API の時間のみを考慮する．

### タスク切り替えオーバヘッド

PF グループに分割したことで，シングルコアのときと比べタスク数が増加している．タスク数が多い場合，タスク起動，終了時によるプリエンプションの回数が多くなり，その分タスク切り替えのオーバヘッドが増加する．そのため本手法ではタスク切り替えのオーバヘッドも考慮して解析を行う．

### タスク起動 ISR オーバヘッド

4.5 節のとおり，タスクはセットマネージャの周期にしたがってタスク起動を行う ISR（タスク起動 ISR）により起動される．タスク起動 ISR の実行時間は起動するタスク数にしたがって変動する．AUTOSAR OS のタスク起動に要するオーバヘッドは [33] によると，クロック数が 60MHz 程度の CPU の場合 4 μ s から 9 μ s 程度かかる．PF グループへの分割のためタスク数は増えており，多数のタスクが同時に実行される場合無視できないオーバヘッドになると予想できる．そのため本手法ではタスク起動によるオーバヘッドを考慮して解析を行う．

表 4.1: 対象パワトレアプリのモデルの属性

| 構成単位 | 属性 |
|---|---|
| イベント | 起動するセットマネージャ |
| PF グループ | 含まれる公開関数 |
| セットマネージャ | 優先度，周期，実行するサブセット |
| サブセット | セットマネージャに対する周期，<br>オフセット，所属する公開関数 |
| 公開関数 | 最大実行時間，読み込む共有データ，<br>書き込む共有データ |
| 共有データ | データサイズ |

表 4.2: RTOS, ハードウェアの情報

| | 考慮するオーバヘッド |
|---|---|
| RTOS | スピンロック API オーバヘッド（$SpinApi$）<br>およびタスク切り替え時間（$OH^{Chg}$）<br>およびタスク起動オーバヘッド（$OH^{Act}$） |
| ハードウェア | コア数，各メモリの各コアからの<br>読み込み（$Latency^{R}(c,m)$）および<br>書き込み時間（$Latency^{W}(c,m)$） |

### 4.6.1　対象パワトレアプリの性能評価用モデル

2.1.1 節で示した対象パワトレアプリから解析を行う上で必要な情報を抽出し，モデル化を行う．対象パワトレアプリの各構成単位をモデルした際に必要な属性を表 4.1 に示す．また，4.6 節から，OS，ハードウェアについて表 4.2 の情報が必要となる．表 4.2 の情報は，実機上でベンチマークソフトを実行することで事前に計測しておく．

## 4.7　マルチコア最悪応答時間解析

本章では，最悪応答時間解析によってマルチコアで実行される対象パワトレアプリの PF グループ単位での公開関数のコア配置を決定する手法について述べる．

本手法ではPF グループのすべてのコア配置パターンについて解析を行い，得られた結果を配置決定に使用する．具体的には本手法を実行するためのツールを実現し，解析の結果優良であったいくつかの配置パターンを実機に実装して評価を行うことを想定している．解析は 4.6.1 節の性能評価用モデルを用いて行う．

### 4.7.1　解析ツールのフロー

本手法は各PF グループのコア配置パターンに対し，以下の手順を実行することで解析を行う．

(a). 共有データの配置決定

(b). 各タスクの最悪応答時間解析

(c). システム最悪余裕時間の算出

（a）ではまずPF グループの配置に対して適切な共有データの配置を決定する．(b）では各タスクについて最悪応答時間を計算する．また，要件（1）により，最悪応答時間がデッドラインより長い場合はそのPF グループの配置パターンに関する解析を打ち切る．(c）では各タスクの最悪応答時間からシステム最悪余裕時間を計算する．

### 4.7.2　（a）共有データの配置決定

共有データ $d$ について以下のアルゴリズムを適用する．

i. アクセスするコアが単一である場合，そのコアのLMEM に $d$ を配置する．

ii. それ以外の場合，以下により，コア $c$ から共有データ $d$ への読み込み頻度 $Freq^R_{d,c}$ 書き込み頻度 $Freq^W_{d,c}$ を計算する．ただし $F^R_{d,c}$ はコア $c$ に配置され共有データ $d$ を読み込む公開関数の集合，$F^W_{d,c}$ はコア $c$ に配置され共有データ $d$ に書き込む公開関数の集合，$p_f$ を公開関数 $f$ の周期とする．

$$Freq^R_{d,c} = \sum_{f \in F^R_{d,c}} \left(\frac{1}{p_f}\right), Freq^W_{d,c} = \sum_{f \in F^W_{d,c}} \left(\frac{1}{p_f}\right) \tag{4.2}$$

iii. 以下により，共有データ $d$ をメモリ $m$ に配置したときのメモリ $m$ に関する合計アクセス時間 $T_{d,m}$ を計算する．ここでコア $c$ からメモリ $m$ へ読み込み時間を $Latency^R(c,m)$，書き込み時間を $Latency^W(c,m)$，共有データ $d$ のデータサイズを $Size(d)$ とする．

$$T_{d,m} = \sum_c \{Freq^R_{d,c} \cdot Latency^R(c,m) \cdot Size(d) + Freq^W_{d,c} \cdot Latency^W(c,m) \cdot Size(d)\} \tag{4.3}$$

iv. 対象とするアーキテクチャのメモリの集合を $M$ とする．$T_{d,\mu} = \min\{T_{d,m} | m \in M\}$ となるようなメモリ $\mu$ に共有データ $d$ を配置する．

### 4.7.3　（b）各タスクの最悪応答時間解析

各配置パターンにおける最悪応答時間を解析し，そこから PF グループのコア配置決定に使用するシステム最悪余裕時間を計算する．ここでは，タスク $\tau$ の最悪応答時間 $WCRT_\tau$ の計算方法を示す．

#### 最悪応答時間解析の概要

対象パワトレアプリにおけるタスクは，異なるサブセットから呼び出される公開関数を含み，呼び出すタイミングによって実行時間が大きく変動するため，MF タスクとして扱うことができる．タスク $\tau$ の $k$ 番目のフレームの実行時間 $C^k_\tau$ は以下の式で与えられる．

$$C^k_\tau = \sum_{f \in F_\tau} \{C_f \cdot a_f(k)\} \tag{4.4}$$

ただし $F_\tau$ をタスク $\tau$ から呼び出される公開関数の集合，$C_f$ を公開関数 $f$ の最大実行時間とする．また，$a_f(k)$ は公開関数 $f$ がフレーム $k$ で実行されるなら 1，そうでないなら 0 を与える関数である．

そして，タスク $\tau$ の $k$ 番目のフレームの実行時間 $C^k_\tau$ は 4.6 節で述べた各種オーバヘッドを加味して以下の式で与えられる．

$$C^k_\tau = \sum_{f \in F_\tau} \{(C_f + OH^{Spin}_f + OH^{Mem}_f) \cdot a_f(k)\} + OH^{Chg} \tag{4.5}$$

ただし公開関数 $f$ におけるスピンロックオーバヘッドを $OH^{Spin}_f$，共有データアクセス時のメモリアクセス時間を $OH^{Mem}_f$ とする．タスク切り替えオーバヘッドは各タスクが実行状態に移るときと終了するときに必ず 1 回ずつ発生するため，それらを合計した値を $OH^{Chg}$ として各タスクの実行時間に加算すれば良い．

タスク $\tau$ のフレーム数を $N_\tau$ とする．ここで，タスク $\tau$ の最大応答時間 $WCRT_\tau$ は以下の計算式で計算される．

$$\begin{aligned} C_\tau &= \max\{C_\tau^k | k = 0, 1, ..., N_\tau - 1\} \\ WCRT_\tau &= \sum_{\mu \in hp(\tau)} M_\mu(D_\tau) + C_\tau + OH_\tau^{Isr} \end{aligned} \tag{4.6}$$

$D_\tau$ はタスク $\tau$ のデッドライン，$OH_\tau^{Isr}$ は時刻 0 から時刻 $D_\tau$ までのタスク起動 ISR によるオーバヘッド，$hp(\tau)$ はタスク $\tau$ と同じコアに含まれ，タスク $\tau$ よりも高優先度なタスクの集合である．$M_\mu(D_\tau)$ はタスク $\mu$ の時刻 0 から $D_\tau$ までの MIF の累計であり，タスク $\mu$ の各フレームの実行時間から計算することができる．

### スピンロックオーバヘッド（$OH_f^{Spin}$）

公開関数 $f$ に関するスピンロックオーバヘッド $OH_f^{Spin}$ は公開関数 $f$ が読み込む共有データ，書き込む共有データのうち，スピンロックが必要なデータに関するスピンロック獲得・解放 API の合計となる．したがって公開関数 $f$ が読み込む共有データの集合を $SD_f^R$，書き込む共有データの集合を $SD_f^W$，$SpinApi$ をスピンロックの獲得・解放の API 実行時間とすと，

$$\begin{aligned} OH_f^{Spin} = &\sum_{d \in SD_f^R} \{SpinApi \cdot IsSpin(d)\} \\ &+ \sum_{d \in SD_f^W} \{SpinApi \cdot IsSpin(d)\} \end{aligned} \tag{4.7}$$

である．また，$IsSpin(d)$ を共有データ $d$ にスピンロックが必要なら 1 そうでないなら 0 となる関数とする．

### 共有データアクセス時のメモリアクセスオーバヘッド（$OH_f^{Mem}$）

公開関数 $f$ に関する共有データアクセス時のメモリアクセスオーバヘッド $OH_f^{Mem}$ は公開関数 $f$ が読み込む共有データ，書き込む共有データすべてについて，公開関数 $f$ が配置されているコアから対象の共有データが配置されているメモリへのメモリアクセスレイテンシと共有データのサイズの積の合計となる．したがって配置メモリを $Mem(d)$，公開関数 $f$ の配置コア（公開関数 $f$ を含む PF グループの配置コア）を $Core(f)$ とすると，

$$\begin{aligned} OH_f^{Mem} = &\sum_{d \in SD_f^R} \{Latency^R(Core(f), Mem(d)) \cdot Size(d)\} \\ &+ \sum_{d \in SD_f^W} \{Latency^W(Core(f), Mem(d)) \cdot Size(d)\} \end{aligned} \tag{4.8}$$

である．

#### タスク起動 ISR オーバヘッド（$OH_{\tau}^{Isr}$）

タスク起動 ISR は，フレーム長をタスク起動 ISR が起動する全タスクの周期の最大公約数とした MF タスクとして扱うことができる．タスク起動 ISR の実行時間はフレームごとに起動するタスク数に比例する．タスク起動 ISR $i$ の $k$ 番目のフレームの実行時間を $C_i^k$，起動するタスク数を $A_i^k$，タスク起動オーバヘッドを $OH^{Act}$，タスク起動の前処理にかかる時間を $Const_i$ とすると，

$$C_i^k = OH^{Act} \cdot A_i^k + Const_i + OH^{Chg} \tag{4.9}$$

となる．タスク起動 ISR への切り替えについてもオーバヘッドがかかるため，タスクと同様に実行時間に $OH^{Chg}$ を加算する．タスク $\tau$ と同じコアに配置されたタスク起動 ISR は，起動するたびにタスク $\tau$ の実行を妨げる．そのときの妨げる最大時間は MIF によって計算することができる．したがって，タスク $\tau$ と同じコアに配置されたタスク起動 ISR の集合を $I_\tau$ とすると，$OH_{\tau}^{Isr}$ は以下の式によって計算される．

$$OH_{\tau}^{Isr} = \sum_{\mu \in I_\tau} M_\mu(D_\tau) \tag{4.10}$$

### 4.7.4　（c）システム最悪余裕時間の算出

システム最悪余裕時間 $WCST$ は以下で求められる．

$$WCST = \min\{D_\tau - WCRT_\tau | \forall \tau \in T\} \tag{4.11}$$

$WCST$ が負となる場合にはデッドラインミスが生じる可能性が出てくる．ここで，デッドラインミスの原因となるタスクは $D_\tau - WCRT_\tau$ が負となるタスクである．

## 4.8　評価

本章では，車載マルチコアシステムの最悪応答時間解析手法に関して，実機実行との比較評価を行う．性能評価に用いる OS, ハードウェアによるオーバヘッドは，実機環境で測定したものを使用する．本章では以下の 4 つの評価を行う．

**（評価 1）** 解析時間の評価

**（評価 2）** 実機と解析値の比較

**（評価 3）** 実行オーバヘッド考慮による影響の評価

**（評価 4）** 適用性の評価

評価 1 では最悪応答時間解析ツールによる解析によって実際に要した時間を計測する．この評価により，要件（1）を充たしていることを確かめる．評価 2 では，実機で余裕時間を計測し，解析値との比較を行う．すべての PF グループのコア配置パターンについて比較を行い，要件（3）（4）を充たしていることを確認する．評価 3 では，各種オーバヘッドによる影響を評価するために，実機による計測結果および 4.6 節で示したオーバヘッドについてそれぞれ考慮していない場合についての解析結果の比較を行う．評価 4 では，本手法を適用するために必要なパワトレアプリの性質を明らかにすることで本手法の適用性を評価する．

本章では評価 2，評価 3 について，実機実行した際に全タスクで最小であった余裕時間と解析によって得られたシステム最悪余裕時間を比較する．実機実行の計測区間はほとんどのタスク起動のパターンを網羅できると考えられる，実行開始から 5 分までの間とした．

### 4.8.1　評価環境

本研究の評価を行う実機環境としてコアごとに 32kB のローカル RAM を持ち，バスを介してアクセスできる 256kB の共有 RAM と 4MB の FLASH を持つ 192MHz の周波数で実行される車載ホモジニアスデュアルコアマイコンを用いた．実機への実装には PMPF[31] を用いた．リアルタイム OS として AUTOSAR OS を用いており，最適化オプションありのコンパイルによって実行ファイルを生成した．解析についても同様の実行環境を想定している．

### 4.8.2　評価モデル

本評価で使用するソフトウェアモデルは，1,000 個程度の公開関数で構成されている．合計 30 個程度の周期タスクと回転角同期タスクがあり，公開関数はそれぞれそのいずれかに配置されている．回転角同期タスクはエンジンの回転数が 2,000〜5,000rpm で安定していると想定したときの周期で実行される．ただし上記の評価環境と評価モデルを用いて評価を行ったとき，2,000rpm の場合はすべての PF

グループのコア配置パターンでデッドラインミスが起きず，5,000rpm の場合は逆にすべてのパターンでデッドラインミスが起こった．そのためこれらの回転数の場合での評価を行わず，3,000rpm および 4,000rpm での評価を行った．なお，パワトレアプリは高負荷時には安全性に影響がないソフトリアルタイムタスクのデッドラインミスが避けられない場合があり，ソフトリアルタイムタスクのデッドラインミスはある程度許容できる構成となっている [29]．そのため回転数が 5,000rpm の場合にはどのような PF グループのコア配置であってもデッドラインミスするタスクが存在するようになっていると考えられる．

対象パワトレアプリを分析した結果，公開関数は 7 つの PF グループに分類することができた．公開関数は PF グループ単位でコアに配置される．PF グループをコアに配置するパターン数は第二種スターリング数で求められ，7 つの PF グループを 2 コアに配置する場合のパターン数は 63，4 コアに配置する場合のパターン数は 350 通りである．このパターン数に対して手作業で最悪応答時間解析を行うことは困難であるが，解析ツールを用意することで，十分に最悪応答時間解析可能なパターン数となる．

### 4.8.3　（評価 1）解析時間の評価

4.8.2 節のモデルに対し，2 及び 4 コアに PF グループを配置する場合について，解析に要した時間を計測した．解析を行う環境は，OS は Windows®10 64bit，CPU は Intel ®Core™i7 2.60GHz，メモリは 16GB である．4 コアでの計測では，機能が拡張されているという前提で行うため，各公開関数の実行時間を 2 倍とした．

結果を表 4.3 に示す．2，4 コアのどちらの場合でも現実的な時間で解析が可能であることがわかった．また，解析の打ち切りにより解析に要する時間が約 2/3 に短縮され，要件（1）に貢献していることがわかる．

### 4.8.4　（評価 2）実機と解析値の比較

評価 2 では 7 つの PF グループを 2 コアに配置する全パターン（63 パターン）に対して，エンジン回転数が 3,000rpm の場合と 4,000rpm の場合について実機実行の最小余裕時間と解析のシステム最悪余裕時間の比較評価を行った．結果を図 4.4，図 4.5 のグラフに示す．図 4.4，図 4.5 のグラフは実機実行の結果で昇順ソートされ

表 4.3: 評価 1： 解析に要する時間

| コア数 | 全配置パターン数 | 打ち切り | 解析時間 [s] |
|---|---|---|---|
| 2 | 63 | あり | 47.7 |
| | | なし | 67.6 |
| 4 | 350 | あり | 252.4 |
| | | なし | 382.7 |

ている．また，解析においてリアルタイム性が保証できないとされた PF グループの配置パターンについては解析を打ち切っている．

結果として，解析においてリアルタイム性が保証できるとされた配置パターンについては解析のシステム最悪余裕時間は実機実行の最小余裕時間より小さい値となり，要件（4）を充たしている．解析でシステム最悪余裕時間が長いものを上位 10 個選択して比較すると，表 4.4，表 4.5 のようになる．全体として解析でシステム最悪余裕時間が長いものは実機実行の最小余裕時間も長くなる傾向があった．順位は必ずしも一致しないが，解析でシステム最悪余裕時間の長さが上位であった PF グループのコア配置パターンの集合の中に実機実行の最小余裕時間の長さが上位であった PF グループのコア配置パターンが含まれている．解析でシステム最悪余裕時間の長さが上位であったいくつかの配置パターンについて実機でも検証を行うため，実装者は実機実行の最小余裕時間の長さが上位である配置パターンを選択することができる．この意味で，要件（3）を充たしていると言える．順位が一致しない理由としては MIF の計算によって実際には発生確率が低いタスク実行のタイミングの組み合わせまで考慮しているためであると考えられる．

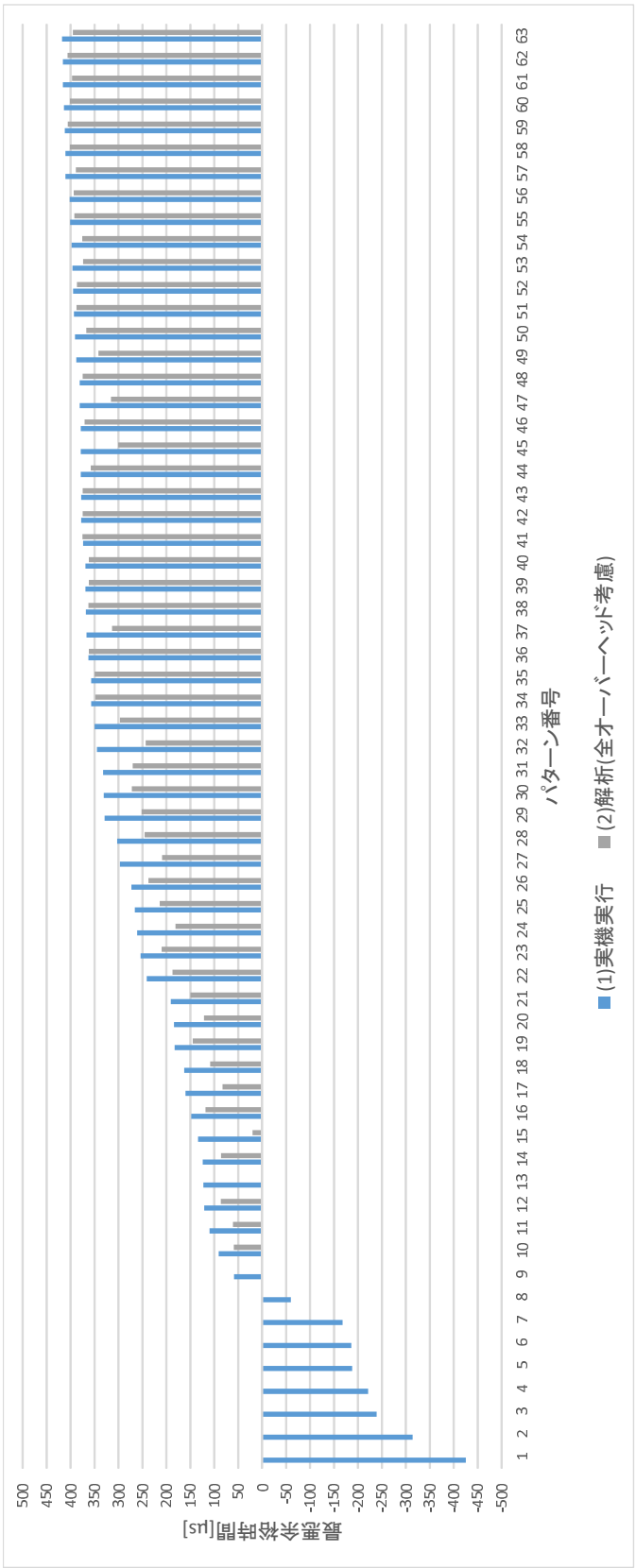


図 4.4: 評価 2（3,000rpm）： 実機実行の最小余裕時間と解析のシステム最悪余裕時間

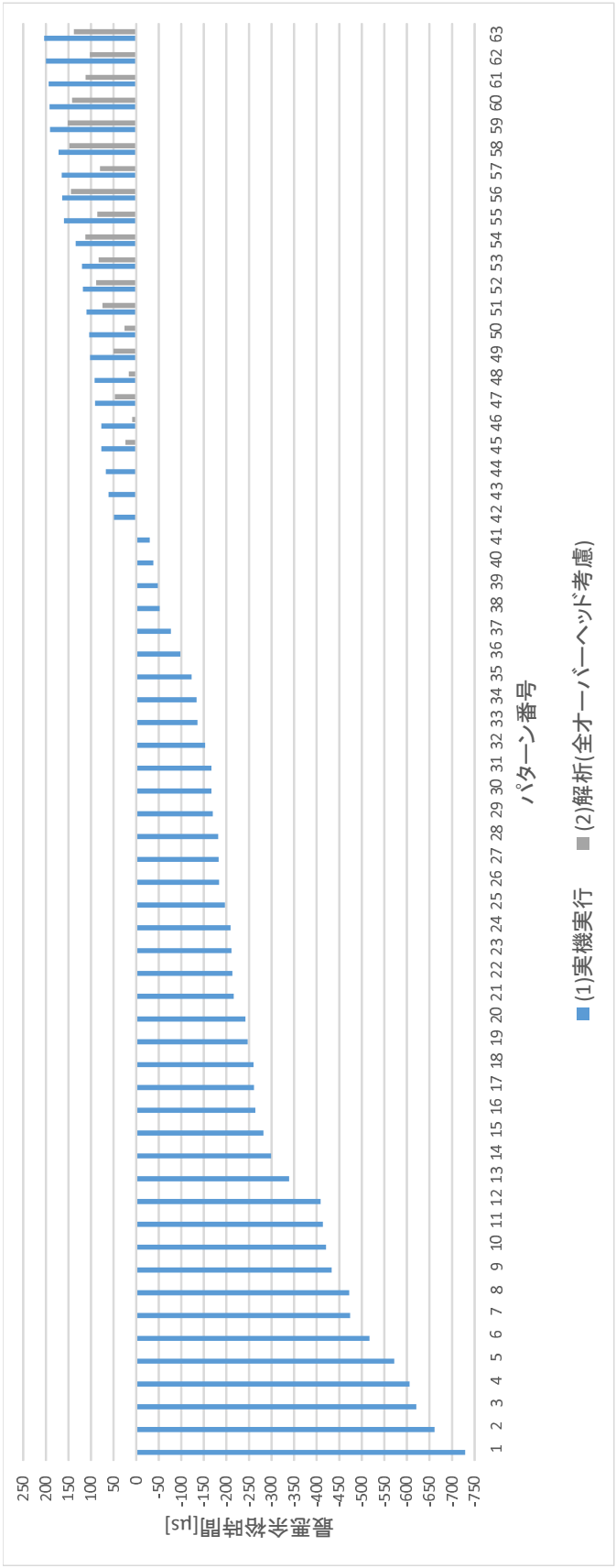


図 4.5: 評価 2（4,000rpm）： 実機実行の最小余裕時間と解析のシステム最悪余裕時間

表 4.4: 評価 2（3,000rpm）：解析結果上位 10 位

| パターン番号 | 解析での順位 | 実機実行での順位 | 解析のシステム最悪余裕時間 [μ s] | 実機実行の全タスクでの最小余裕時間 [μ s] |
|---|---|---|---|---|
| 62 | 1 | 2 | 406.3 | 416 |
| 44 | 2 | 5 | 405.9 | 412 |
| 48 | 3 | 4 | 401.7 | 414 |
| 54 | 4 | 6 | 401.5 | 411 |
| 51 | 5 | 3 | 397.0 | 416 |
| 57 | 6 | 1 | 395.1 | 418 |
| 56 | 7 | 8 | 393.7 | 402 |
| 50 | 8 | 9 | 392.1 | 401 |
| 43 | 9 | 7 | 388.9 | 411 |
| 36 | 10 | 13 | 387.9 | 393 |

### 4.8.5　（評価 3）実行オーバヘッド考慮による影響の評価

評価 3 では実機実行によって得られた最小の余裕時間を長い順に 10 個の PF グループのコア配置パターンを抽出し，4.6 節で示したオーバヘッドについてそれぞれ考慮していない場合の解析結果との比較を行った．結果を図 4.6，図 4.7 のグラフに示す．図 4.6，図 4.7 は各オーバヘッドを考慮していない場合の解析のシステム最悪余裕時間と実機実行での最小の余裕時間を比べたときの増減率を表している．

公開関数の実行時間のみを考慮した場合（図 4.6，図 4.7（7））は実機実行に対する増減率が非常に高く，オーバーヘッドの考慮はパワトレアプリの性能評価を行う上で必須である．スピンロックオーバヘッド以外を考慮した場合（図 4.6，図 4.7（3））および共有データへのメモリアクセス時間以外を考慮した場合（図 4.6，図 4.7（4））は本手法で対象としているオーバヘッドをすべて考慮した場合（図 4.6，図 4.7（2））と比べて差は小さいものの，解析のシステム最悪余裕時間が実機実行の最小余裕時間より大きい場合がある．タスク起動 ISR オーバヘッド以外（図 4.6，図 4.7（5）），タスク切り替えオーバヘッド以外（図 4.6，図 4.7（6））を考慮した場合は解析のシステム最悪余裕時間が実機実行の最小余裕時間より大きくなる場

表 4.5: 評価 2（4,000rpm）：解析結果上位 10 位

| パターン番号 | 解析での順位 | 実機実行での順位 | 解析のシステム最悪余裕時間 [μ s] | 実機実行の全タスクでの最小余裕時間 [μ s] |
|---|---|---|---|---|
| 59 | 1 | 5 | 151.1 | 191 |
| 58 | 2 | 6 | 148.0 | 172 |
| 56 | 3 | 8 | 144.4 | 164 |
| 60 | 4 | 4 | 142.0 | 192 |
| 63 | 5 | 1 | 138.0 | 204 |
| 54 | 6 | 10 | 112.3 | 134 |
| 61 | 7 | 3 | 112.2 | 194 |
| 62 | 8 | 2 | 102.6 | 200 |
| 52 | 9 | 12 | 88.7 | 118 |
| 55 | 10 | 9 | 86.3 | 160 |

合が多い.

以上のことから要件（4）を充たすためにはスピンロックオーバーヘッド，共有データへのメモリアクセス時間，タスク起動 ISR オーバヘッド，タスク切り替えオーバヘッドを考慮する必要がある．また，4.6 節でスピンロックのアイドル時間は考慮しないとしたが，実際に考慮せずとも要件（4）を充たすことが可能であることを確認できた.

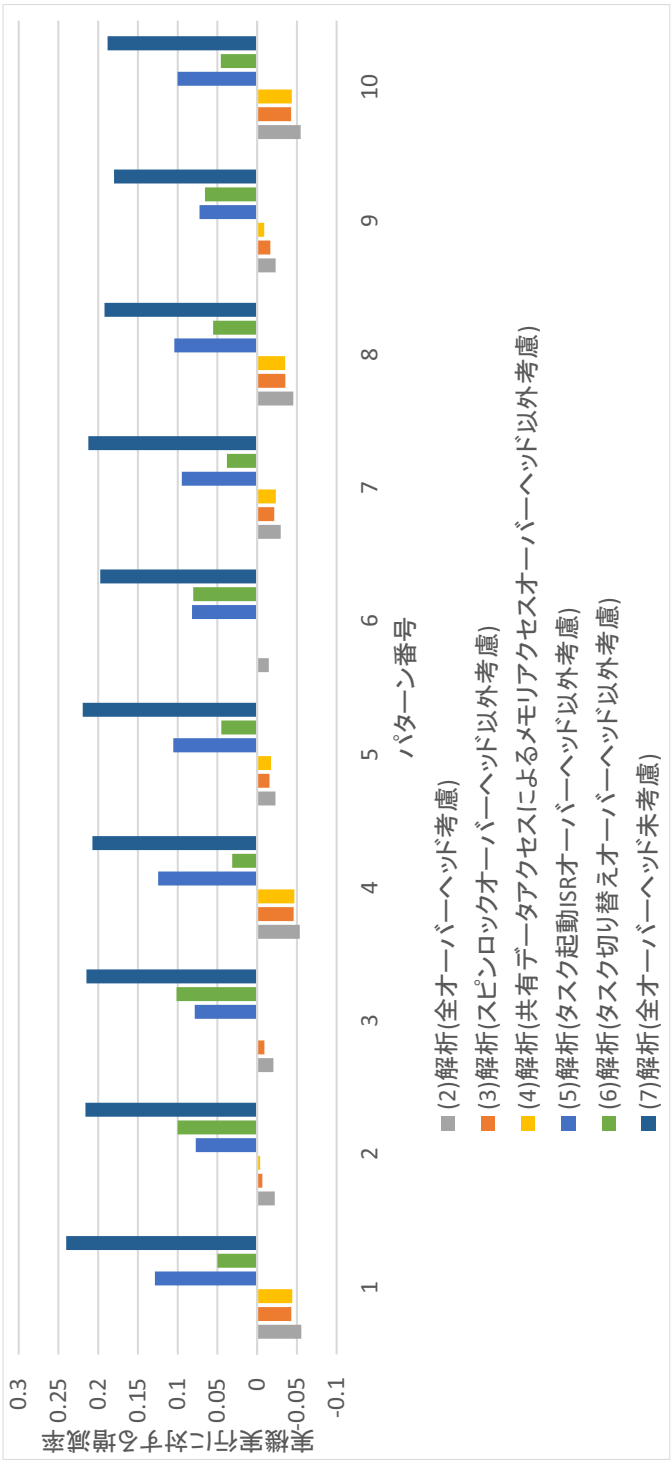


図 4.6: 評価 3（3,000rpm）： 各オーバヘッドを考慮していない解析の実機実行に対する増減率

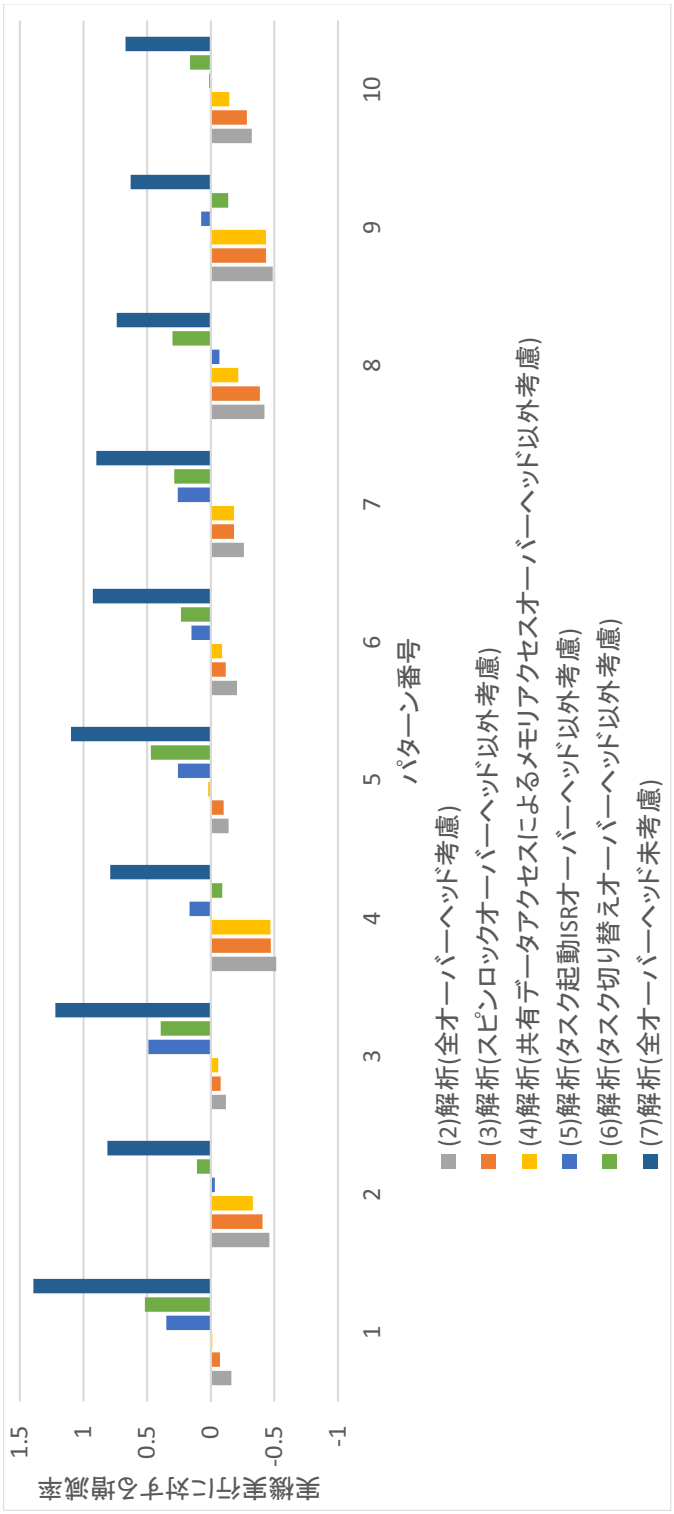


図 4.7: 評価 3（4,000rpm）： 各オーバヘッドを考慮していない解析の実機実行に対する増減率

### 4.8.6　（評価 4）適用性の評価

本手法の適用性の評価として，本手法が適用できるパワトレアプリの性質を示す．本手法を適用するためには，2.1.1 節で述べた対象パワトレアプリと同じような構成を持ち，2.1.2 節で述べたような車載マルチコアアーキテクチャ上に実装され，4.3.2 節の方法で公開関数に相当する処理単位の集合を PF グループに割当てることができ，4.6.1 節で示したような情報を取得できるパワトレアプリである必要がある．また，評価 3 では対象パワトレアプリの構成上，スピンロックのアイドル時間の影響を考慮せずとも要件（4）を充たすことが可能であったが，スピンロックの取得頻度が高かったり，スピンロックによる排他時間が長いようなアプリの構成であった場合はスピンロックのアイドル時間を考慮する必要が出てくる．

以上のことから，本手法が適用できるパワトレアプリの性質は以下のとおりである．

- RTOS 上の複数のタスクおよび ISR によって構成されており，各タスクおよび ISR は優先度を持ち，優先度ベーススケジューリングが行われること
- タスクおよび ISR の起動周期がわかっていること
- タスク実行中に待ち状態とならないこと
- 回転数が固定の状態では非同期な処理が存在しないこと
- 公開関数に相当する処理単位の実行時間と実行依存関係がわかっており，処理単位を他のグループの処理単位と実行依存関係を持たないようにコア数より多い数のグループに分割することができること
- 実装するハードウェアの各メモリの各コアからのアクセス時間がわかっていること
- すべての公開関数に相当する処理単位について，各共有データに書き込みもしくは読み込みを行う回数がわかっていること
- すべての公開関数に相当する処理単位について，スピンロックを獲得・解放する回数がわかっていること
- スピンロックの衝突回数が解析で考慮しなくてよいレベルで少ないこと

## 4.9　関連研究

マルチコア車載ソフトの処理や共有データの配置を決定する方法としては，本手法のように最悪応答時間解析を用いたもの以外にも，各処理の実行周期や実行時間などの情報を用いて配置を決定するマッピングツールを用いたものがある．[34] では焼き鈍し法 [35] や遺伝的アルゴリズム [36] などのヒューリスティックアルゴリズムによって AUTOSAR [24] のランナブルとデータのコア，メモリ配置を決定する手法を提案している．[34] は本研究と比べて自由度の高い配置パターンを，評価値を計算することでを検出することができるが，ヒューリスティックなアルゴリズムであるため局所最適解に陥る可能性がある．本研究の手法でも公開関数はグループ化してコアに配置されるため，公開関数のグループ化が適切に行われなかった場合には大域最適解を得られない可能性がある．しかし 4.3.1 節でも説明したとおり，互いに実行依存関係のある公開関数は同じコアに配置されるべきである．PF グループによるグルーピングの目的の 1 つがそのような公開関数を同じコアに配置することなので，同じ PF グループに含まれる公開関数は同じコアに配置される必要性が高いものであると考える．

また，解析にかかる実行時間については，[34] では 1,000 個程度の公開関数の配置をランダムな初期配置から優良な配置になるまでイテレーションを行うのは時間がかかると考えられる．将来コア数やアプリケーションの規模が増加した場合，[34] では最適解に収束するための時間が増大していく．本手法でも PF グループをコアへ配置する組み合わせ数が増えれば解析時間も増加することが予想される．しかし本論文の手法では PF グループの数をコア数に対して多くしなければ 4.8.3 節の評価 (1) の結果にあるように 4 コアであっても解析時間は長くはない．4 コアを超えるような場合については本論文の 2.1.2 節にあるとおり，ハードウェアの構成が大きく変化することが予想されるため，本研究の対象外としている．タスク数や公開関数が増えた場合でも PF グループの数を多くしなければ，組み合わせのパターン数が変わらないため解析時間の増加は大きなものにならない．ただし 1 つの PF グループに含まれる公開関数の数が増えるため，コア配置の自由度は相対的に低くなる．

[37] では，NSGA-II [38] という遺伝的アルゴリズムを用いた最適化アルゴリズムによって mNRT(maximum Normalized Response Time)，タスクの待ち時間，コア間通信頻度のパレート解となるようなタスクのコア配置を求める手法を提案している．mNRT はすべてのタスクの中で応答時間の最大値をデッドラインで割っ

た値が最大のものであり，本論文のシステム最悪余裕時間と近い概念である．遺伝的アルゴリズムのイテレーションごとにシミュレーションツールによってこれら 3 つのパラメータの取得を行う．[37] の手法と本手法を比較すると，[37] の手法では遺伝的アルゴリズムのイテレーションごとにシミュレーションを行うため，静的な解析を行う本手法と比べ，最終的なコア配置を得るまでに必要な時間が長いと考えられる．また，[37] の手法はコア間を跨ぐランナブルの実行依存関係がなくなるような仕組みを持たないため，PF グループを用いる本手法と異なり，待ち状態を考慮しなければならない．最後に [37] の手法はシミュレーションを用いているために，非常に低い確率で発生するタスクや ISR 実行の組み合わせについて評価できていない場合があり，最悪応答時間を取得できていることを保証できない．対して本論文の手法では対象のタスクと同等以上の優先度を持つタスクおよびタスク起動 ISR ついて MIF を計算し，対象のタスクの実行時間との合計を出すことで最悪応答時間を計算しているため，すべてのタスクおよびタスク起動 ISR 実行の組み合わせについて評価できている．

[3] ではランナブルの実行依存関係を充たしつつ，ヒューリスティックアルゴリズムによって配置を決定する手法を提案している．[3] の配置アルゴリズムは各ランナブルについて CPU 利用率の高いものから順に配置するもので，各 CPU のシステム最悪余裕時間について考慮することはできない．車載ソフトでは安全に実行するためにシステム最悪余裕時間が重要な指標となるため，本研究には適さない．

[39] は MAST[40] などのオープンソースの解析ツールを用いてマルチ ECU を前提とした最悪応答時間解析手法について提案している．[39] はオープンソースを用いることで開発コストを削減しているが，マルチコアを前提としていないためメモリアクセス時間や排他競合時間を考慮することができない．

車載ソフトの性能を実機で実行する前に把握する方法としてはマルチコアシミュレータ [41][42] がある．しかしシミュレータでは一つのパターンに関する結果を得るために必要が時間が多いことと，理論的な最悪値を出すことができないことから，配置決定のための性能把握方法としては不適格である．

## 4.10　おわりに

本研究ではマルチコアで実装された対象パワトレアプリの最悪応答時間解析に必要な要因として OS およびハードウェアのオーバヘッドについて述べた．PF グループを用いることで公開関数のコア配置パターンを削減し，PF グループの全配

置パターンについて最悪応答時間を解析し，システム最悪余裕時間を計算することで公開関数のコア配置の決定を支援するツールを実現した．課題 (e) は，本手法が要件（4）を満たしていることからデッドラインミスが生じないマッピングを選定することができるため，対応することができている．課題 (f) は，公開関数を PF グループへ分割することにより対応することができている．課題 (h) は，4.7.4 節で示したとおり，デッドラインミスの原因となるタスクを検出することができるため，対応することができている．要件（2）（4）により本手法によって得られたマッピングではデッドラインミスしないため，実機での検証コストを削減することができる．したがって課題 (i) は対応することができている．課題 (j) は，本手法は要件（2）を満たしており，すべての ISR やタスクの実行パターンを考慮しているため，対応することができている．評価によって解析は現実的な時間で行うことができ，実機実行結果と相関があることからコア配置決定に有用であることを示した．

# 第5章　車載制御ソフトウェアに適した効率的なLET実装

## 5.1　概要

本章では，1章で述べた研究 (III) について説明する．本章における LET の実装方法は AUTOSAR に対応するために AUTOSAR のモデルを想定している．つまり処理の単位としてランナブルが作成され，ランナブルはタスクに配置される．ランナブルは公開関数と同等の粒度であるため，本章の手法は研究 (I) のランタイム生成に活用することができる．

1章で述べたように，通信タイミングが決定的となるランタイム構造を実現する方法として LET（ 2.5 節を参照）を用いることができる．LET によって 1 章で挙げた課題 (b)(c) に対処することが可能である．LET を実現するための手法はすでに提案されている．文献 [7] では LET で行う共有データ通信を専用のタスクもしくは ISR に担当させることで LET を実現する手法が提案されている．本論文ではこの専用のタスクもしくは ISR を LET 処理と呼ぶ．また，Beckert らは，ダブルバッファリングを用いて LET を効率的に実現する手法について提案した [8]．しかし，LET 処理では膨大な量のデータ通信を行うため実行時間が非常に長く，より効率的な LET 処理の実装が求められる．また，実際のパワトレアプリに LET を適用するためには以下の 2 つの課題がある．1 つは，実際のパワトレアプリの中には CPU 利用率を軽減させるためにサブスケジューリング (2.1.1 を参照) を用いているため，これを考慮した LET の実現方法を検討する必要があるということである．もう 1 つは，低優先度タスクがデッドラインミスすることを想定した LET の実装が必要であるということである．一般的に車載システムはハードリアルタイムシステムであるため，デッドラインミスが許容されていない，しかし，パワトレアプリは回転数に応じて CPU 負荷が変動し，高回転時には一部の低優先度なタスクはデッドラインミスする可能性がある．そのため，そのような低優先度なタスクについてはデッドラインミスが生じる可能性があることを前提として制御設

計がなされている．しかし，既存研究ではデッドラインミスが起きた場合について考慮されていない．その結果，不整合なデータが他のタスクに公開されてしまう可能性がある．

本章では，実際のパワトレアプリに LET を適用するために，サブスケジューリングに対応した LET の実装を提案する．また，タスクがデッドラインミスしたときの影響を軽減できるような LET 通信の実装方法として， Deadline Miss Tolerant LET (DMT-LET) を提案する．次に CPU あたりの LET 処理の負荷を軽減するために，LET 処理を複数の CPU に分散する手法を提案する．同期を用いない LET の分散手法としてハードリアルタイムな処理を対象とした Asynchronized Distributed LET process (ADLP)，同期を用いた LET の分散手法としてソフトリアルタイムな処理を対象とした Synchronized Distributed LET process (SDLP) を提案する．また，SDLP と ADLP を組み合わせてハードリアルタイムおよびソフトリアルタイムな処理が混在するシステムを対象とした Hybrid Distributed LET Process (HDLP) を提案する．最後に， HDLP を用いたときのメモリ使用量と CPU 利用率についての評価を行う．

## 5.2　既存手法の適用

### 5.2.1　LET 処理

Biondi らは，LET 通信のコピーインおよびコピーアウトを専用のタスクによって実装することを提案している [7]．ここで，LET 通信におけるコピーイン，コピーアウトを行う処理単位を LET 処理 (LET-P) と呼ぶ．LET-P の例を図 5.1 に示す．LET-P はタスク起動イベントごとに用意され，高優先度タスクもしくは ISR として実装される．LET 処理の周期は対応するイベントの周期とする．例えば，周期 1ms の時間イベントに対応する LET 処理の周期は 1ms となる．LET-P はその LET-P に対応するイベントによって起動されるランナブルによってアクセスされるローカルデータについて，コピーイン・コピーアウトを行う．LET-P には，前回の LET 区間中のローカルデータ書き込みに対するすべてのコピーアウトの後にコピーインを行うという制約がある．これは，コピーアウトによる共有データの更新を行う前にコピーインが行われないようにするためである．この制約によって，LET-P はコピーアウトとコピーインの順序を守るために単一の CPU によって実行されなければならない．しかし LET-P では 10000 個にも及ぶ共有データのアク

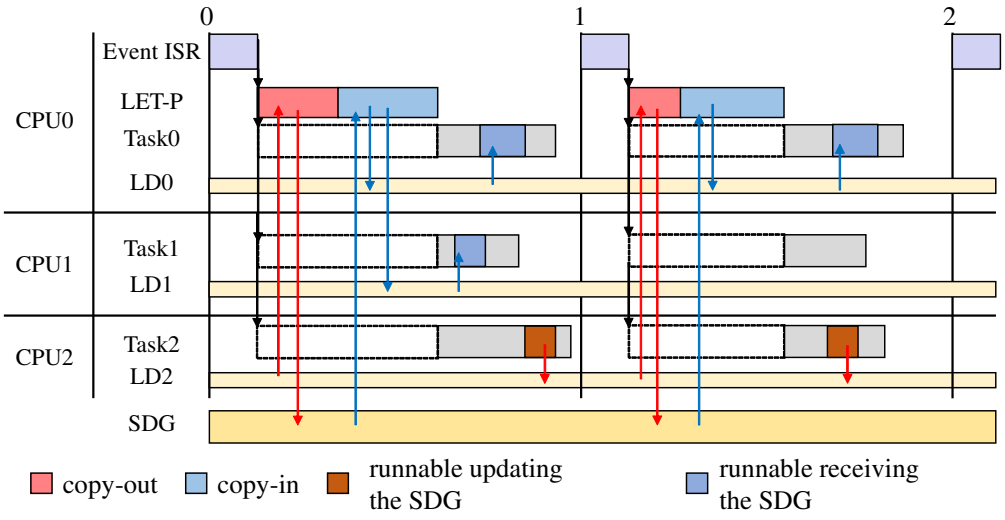


図 5.1: LET 処理

セスを行うため，LET-P の実行時間は大きい．また，コピーインを行う前にタスクによって共有データが利用されるのを避けるため，すべてのタスクは LET-P が完了するまで待つ必要がある．これらの理由から，LET-P による実行オーバヘッドは大きく，より効率的に LET-P を実装することが必要である．

### 5.2.2　ダブルバッファリング

LET 通信を効率的に実現する方法として，ダブルバッファを用いた手法が提案されている [8]．ダブルバッファを用いた通信モデルについて図 5.2 に示す．本手法では，まず，読み込みおよび書き込みを行うランナブルが同じである共有データを共有データグループ （SDG）とする．そして SDG ごとに 2 つの分離されたバッファ(ダブルバッファ) を用意する．このバッファのうち片方をリードバッファ（RB），もう片方をライトバッファ（WB）とする．RB，WB は SDG ごとに用意されたリードバッファポインタ（RBP），ライトバッファポインタ（WBP）によって参照される．読み込み側のランナブルは LET-P のコピーインによって RB からデータを受け取り，一方書き込み側のランナブルは直接 WB を更新する．そして LET-P はコピーアウトの代わりに SDG ごとに用意された 2 つバッファポインタのスワップを行う．このようにすることで 2 つのバッファに対するアクセスは競合せず，WB に書き込まれた値はスワッピングによって読み込み可能となる．ダブルバッファによって，多数の共有データのコピーアウトがそれより少数の SDG の

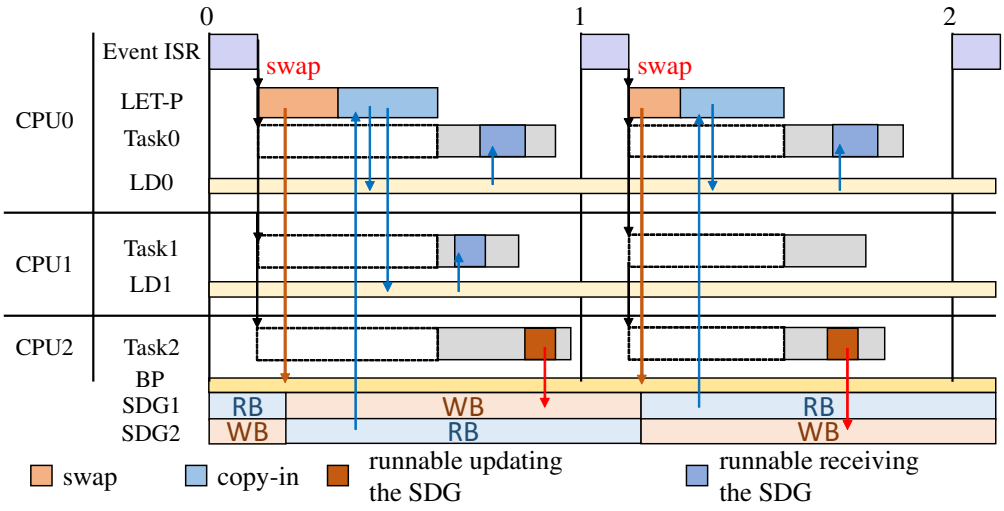


図 5.2: ダブルバッファリングを用いた LET

バッファポインタのスワップに置き換わるため，LET-P の実行時間を削減することが可能である．本論文の手法はダブルバッファを用いた LET を対象とする．

## 5.3 実際の車載制御システムへの LET の適用要件

タスクはリアルタイム性の要件に応じて，ソフトリアルタイムタスクとハードリアルタイムタスクに分類される．エンジンが高回転であるとき，すべてのタスクをスケジュール可能とするのは困難であるため，比較的安全性に対する重要度の低いソフトリアルタイムタスクのデッドラインミスは許容される．一方，ハードリアルタイムタスクは安全性の要求のために必ずデッドライン内に完了できなければならない．

一般的に，ソフトリアルタイムタスクは低優先度に設定され，ハードリアルタイムタスクは高優先度に設定される．本論文のパワトレアプリは RMS を採用しているため，ソフトリアルタイムは長周期，ハードリアルタイムタスクは短周期に設定される．ここで，ソフトリアルタイムタスクがデッドラインミスした場合の要件を以下に示す．

**(R1):** デッドラインミスしたタスクが終了するまでそのタスクの起動は行われない．

**(R2):** タスクがデッドラインミスした場合であってもタスク内のサブレイヤの実行タイミングは変化しない.

**(R3):** デッドラインミスしたソフトリアルタイムタスクが書き込み側であれば,対応する読み込み側のタスクは書き込み側のタスクが前回の周期で生成した共有データを用いて計算を行う.

ハードリアルタイムタスクがデッドラインミスした場合はシステムの OS の機能によるシャットダウンなどで対応するため，本論文でそれらのタスクのデッドラインミスに対応する必要はない.

現状の LET はソフトリアルタイムタスクがデッドラインミスを想定していないため，上記の要件を満たすような LET の実装を検討する必要がある．R1 はイベントによるタスク起動時にタスクの状態を確認し，休止状態でなければタスク起動をスキップするようにすることで満たすことが可能である．R2 はサブレイヤの実行タイミングとタスクの実行が独立するように実装すれば満たすことができる．例えばサブレイヤの実行タイミングを管理するためのカウンタ変数を，そのサブレイヤを含むタスクではなくタスクを起動するイベントによってインクリメントするように実装すればよい．ここで，もしランナブルがサブレイヤによって定められた次のランナブルの実行タイミングより前に完了することができるならば，そのランナブルを含むタスクがデッドラインミスした場合であってもランナブルの実行はスキップされない.

一方，R3 はダブルバッファを用いた場合，既存の LET では満たすことができない．図 5.4 では， 時刻 10，12 で起動したサブレイヤ SLR 内のランナブルは本来時刻 8 で起動したサブレイヤ SLW 内のランナブルが生成したデータを使用するべきである．しかし，サブレイヤ SLW を含むタスクがデッドラインミスしたため，R3 により時刻 4 で起動した SLW が生成したデータを使用すべきであるが，それよりも前の時刻 0 で起動した SLW 内のランナブルが生成したデータを使用することになる．本論文では R3 を満たすことができるようなダブルバッファを用いた LET 通信の実装方法について提案する.

## 5.4　対象車載制御アプリケーションへの LET 適用

本章では，実際のパワトレアプリに LET を適用するための実装方法を示す．まず，実際のパワトレアプリで採用されているサブスケジューリングに対応した LET

の実現方法を示す．次に，R3 を満たし，ソフトリアルタイムタスクがデッドラインミスすることを許容できる LET の実装方法として Deadline Miss Tolerant LET (DMT-LET) を提案する．

### 5.4.1　サブスケジューリングへの対応

サブスケジューリングが適用されているパワトレアプリでは，ランナブルが実行されるタイミングはタスクの周期ではなく，サブレイヤの起動周期およびオフセットに依存するため，サブレイヤごとに LET の区間を設定する必要がある．ここでは，サブスケジューリング下のパワトレアプリにおける LET 区間の設定方法について述べる．

最初に考えられるのは，サブレイヤの LET 区間をサブレイヤの実行間隔に設定するという方法である．例えば，サブレイヤのサブ周期を 4，そのサブレイヤを含むタスクの周期を 2ms とした場合，サブレイヤの LET 区間はサブレイヤの実行周期と同じ 8ms と設定する．しかし，この方法ではサブ周期が長く設定されている場合に LET 区間が長大になり，共有データが更新されてからその更新後の値が読み込み可能となるまでの時間が増大するという問題がある．そのため，より効率的な LET 区間の設定が必要となる．

ここで，書き込み側のランナブルは，デッドラインミスが発生しない限り，そのランナブルが配置されているタスクのデッドライン以内に共有データの更新を行うことができるはずである．したがって，本論文で提案する手法では，LET の区間をサブレイヤの起動タイミングからタスクの周期経過した時刻までの区間に設定する．これにより，サブレイヤの LET の区間はタスクの周期分の長さとなり，サブレイヤの実行間隔に設定する場合と比較して速やかに他のタスクに更新結果を反映させることができる．

サブスケジューリング下の LET 区間について図 5.3 に示す．図 5.3 ではサブレイヤ SL2 はサブ周期が 2，サブオフセットが 1 で，SL2 を含むタスクの周期が 2 であるため，LET の区間は $[2, 4], [6, 8], ...$ となる．

### 5.4.2　Deadline Miss Tolerant LET (DMT-LET)

DMT-LET のコンセプトは SDG を書き込むサブレイヤを含むタスクがデッドラインミスしたときに LET-P によるバッファポインタのスワップを行わないように

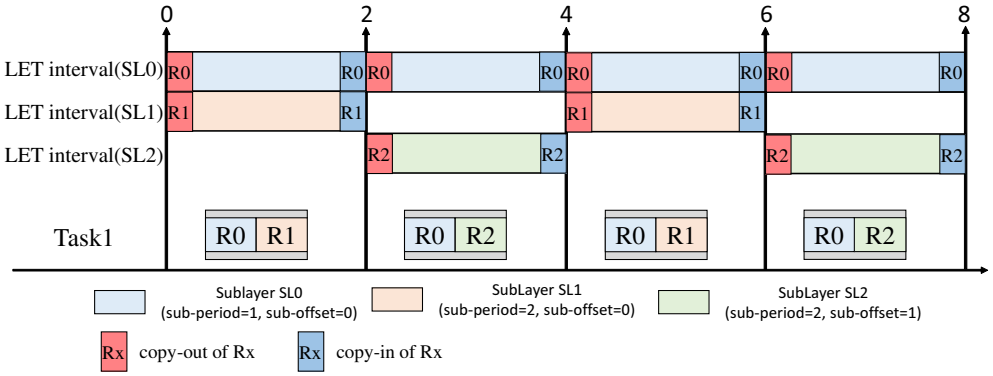


図 5.3: サブスケシューリング下での LET 区間

することである．図 5.5 に DMT-LET を用いた場合の LET を示す．DMT-LET では，各 SDG に対し対応するデータ更新フラグを用意する．DMT-LET における LET-P の処理について説明する．まず，データ更新フラグをチェックする．このとき，データ更新フラグが True のときはスワップを行い，False の場合はスワップを行わない．次にデータ更新フラグを False にする．最後にコピーインを行う．書き込み側のランナブルはダブルバッファリング適用によって WB に直接書き込みを行うが，WB に対するすべての共有データの更新が終了した際に対応する SDG のデータ更新フラグを True とする．

このようにすることで，LET-P 実行時にバッファポインタをスワップするタイミングで SDG のデータ更新フラグが False であれば，その SDG に対応する WB に書き込むランナブルがデッドラインミスしたことがわかる．そして，デッドラインミスした場合はその SDG に関してバッファポインタのスワップを行わず，次の LET-P 実行時に SDG が更新されていれば（つまり，データ更新フラグが True になっていれば）バッファポインタのスワップが行われる．このようにすることで，図 5.5 の例では，SLR は少なくとも SLW がデッドラインミスしたときの 1 つ前に SLW によって更新された SDG の値を読み込むことができ，R3 を満たすことができる．具体的には，図 5.5 では，DMT-LET によって，時刻 10 で実行された LET-P は書き込み側ランナブルが更新する WB に対応する SDG のデータ更新フラグをチェックし，データ更新フラグが False になっているため SDG のバッファポインタはスワップされない．その結果，時刻 10 で起動した SLR 内のランナブルは時刻 4 で起動した SLW 内のランナブルが更新したデータを使用することが可能になっている．

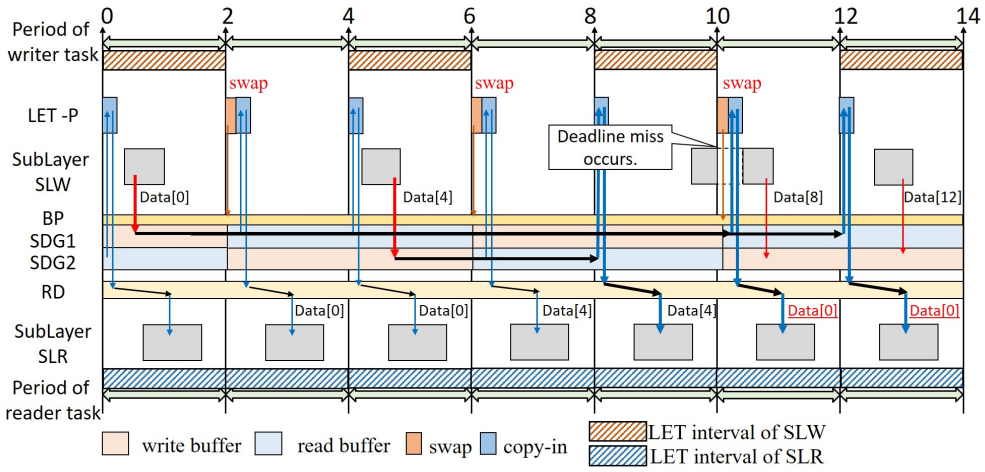


図 5.4: デッドラインミスが生じたときに起こる問題

また，WB を書き込むランナブルを持つタスクがデッドラインミスした場合でも，サブレイヤのサブ周期が 2 以上であれば，デッドラインミスによってそのランナブルの実行がスキップされない限り SDG のバッファポインタのスワップ後は正しい更新データを受け取ることができる．図 5.5 では，時刻 8 で起動された SLW 内のランナブルがデッドラインミスしているが，次の SLW の起動タイミングである時刻 12 までに WB の更新が完了しているため，時刻 12 で起動された SLR は想定通り時刻 8 で起動された SLW 内のランナブルが更新したデータを受け取ることができる．

このように DMT-LET によってタスクのデッドラインミスに対応することが可能となるが，データ更新フラグのチェックや False への更新によって LET-P の実行時間が増加していることに注意する必要がある．現状のパワトレアプリでも SDG の数は 300 個程度あり，SDG 一つ一つに対するこれらの操作を LET-P の起動ごとに行う必要があるため，の実行時間の増加は大きい．このことは各 CPU に LET-P を分散して効率的に LET-P を実装する必要性を高めている．

## 5.5　LET 分散手法

数千ものデータを扱う LET-P はダブルバッファリングを適用し，コピーアウトによる実行オーバヘッドを削減したとしても，非常に実行負荷が大きく，LET-P の処理が完了するまで元々のタスクが実行できないことからオーバヘッドも大き

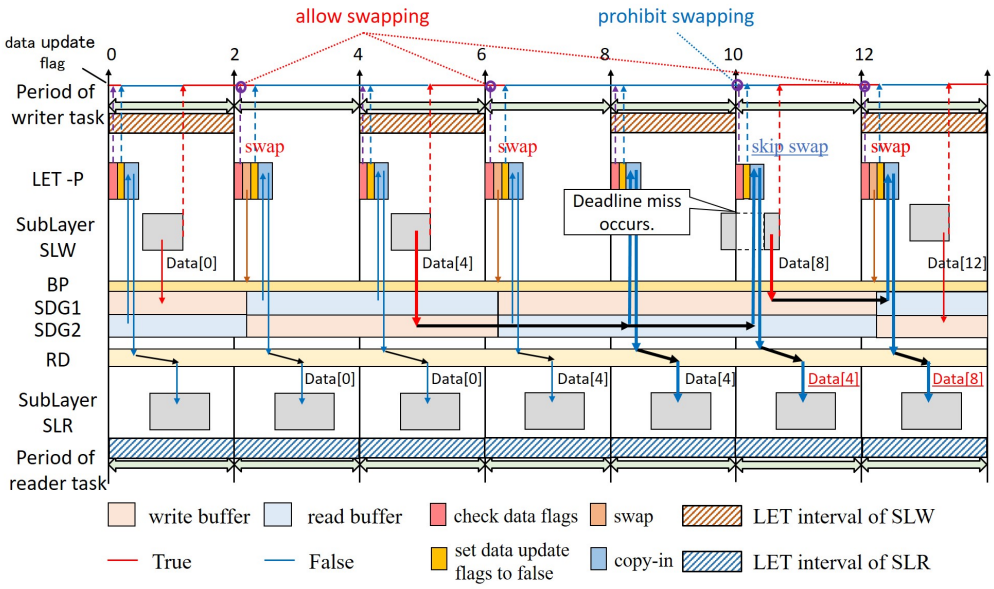


図 5.5: Deadline Miss Tolerant LET (DMT-LET)

い．各 CPU の LET-P による実行負荷を軽減するためには，LET-P の処理を複数 CPU に分散することが考えられる．LET-P の分散の基本的な考えは，各 CPU に対して LET-P を用意し，各 CPU に配置されているランナブルに対する SDG への操作をその CPU に用意された LET-P が実施するというものである．しかし，5.2.1 節でも述べたように，データの整合性をとるためにコピーアウト（ダブルバッファリング適用下ではバッファポインタのスワップ）とコピーインの実行順序を守らなければならないという制約がある．本節では，DMT-LET によってデッドラインミスの監視および対応しつつ，上記の制約を満たしつつ LET-P を複数 CPU に分散する手法を提案する．まず，すべてのタスクがハードリアルタイムタスクである場合に適用できる手法である，ADLP(Asynchronized Distributed LET-P) について述べる．次に，すべてのタスクがソフトリアルタイムタスクである場合を対象とした，SDLP(Synchronized Distributed LET-P) について述べる．最後に，実際のパワトレアプリのタスク構成である，ハードリアルタイムタスクおよびソフトリアルタイムタスクが混在するシステムを対象とした HDLP(Hybrid Distributed LET-P) について述べる．

### 5.5.1 Asynchronized Distributed LET Process (ADLP)

ADLP は LET 区間によってバッファポインタのスワップのタイミングが静的に決まっているというコンセプトに基づいている．そのため，SDG にアクセスするランナブルや LET-P は，現在時刻を確認し，その時刻に応じて自分がアクセスすべきバッファを選択する．図 5.6 に ADLP による LET-P の分散を示す．ある時刻において SDG を読み込むべきランナブルが実行される CPU に配置された LET-P は現在時刻からどちらのバッファが RB であるかを判断し (図 8(b) の場合は SDG2)，RB から値を読み込み，ローカルデータにコピーする．現在時刻が 0 から 1 までの値であれば、LET-P は RB として SDG2 を選択し、RB の値を各ランナブルに対応するローカルデータにコピーする．また，書き込みランナブルもまたどちらのバッファが WB であるかを判断し (図 8(b) の場合は SDG1)，直接そのバッファの値を更新する．現在時刻が 0 から 1 までの値であれば、SDG を更新するランナブルは WB として SDG1 を選択し，WB の値を直接更新する．次の区間 (時刻 1 から 2) では，各ランナブルや LET-P はもう一方のバッファを選択する．

LET によって RB と WB の切り替えのタイミングは静的に決まっている．具体的には，対象の WB を更新するランナブルを含んでいるサブレイヤの LET 区間の末尾において RB と WB が切り替わる．サブレイヤの実行周期はサブレイヤのサブ周期とそのサブレイヤを含むタスクの周期の積であり，その倍の周期がバッファ選択の周期となる．以下に RB の選択の条件式を示す（RB として選択されなかった方が WB となる）．SDG $d$ の 2 つのバッファをそれぞれ $d_0$，$d_1$ とする．SDG $d$ を書き込むサブレイヤ $s_w$ を含むタスク $\tau_w$ の周期を $p_{\tau_w}$，$s_w$ の周期を $p_{s_w}$，オフセットを $of_{s_w}$ とする．ここで，サブレイヤ $s_w$ の実行周期を $P_w = p_{\tau_w} \cdot p_{s_w}$ とすると，バッファ選択の周期は $2P_w$ である．時刻 $t(0 \leq t < 2P_w)$ のときの SDG $d$ のリードバッファ $d_r(t)$ は下のとおりである．

$$d_r(t) = \begin{cases} d_0 & \text{if } (of_{s_w}+1)p_{\tau_w} \leq t < (of_{s_w}+p_{s_w}+1)p_{\tau_w} \\ d_1 & otherwise \end{cases} \tag{5.1}$$

この手法では，LET-P によるバッファポインタのスワップが不要となるため，LET-P の実行時間を大幅に短縮でき，なおかつスワップとコピーインの順序制約はなくなり，各 LET-P 間で非同期的に実行することが可能である．しかし，ADLP では DMT-LET を適用することができない．なぜならば，タスクのデッドラインミスによりデータを書き込むサブレイヤの実行がスキップされてしまった場合，バッファの切り替えを行うべきタイミングが変化するため，式 (5.1) によって静的に決

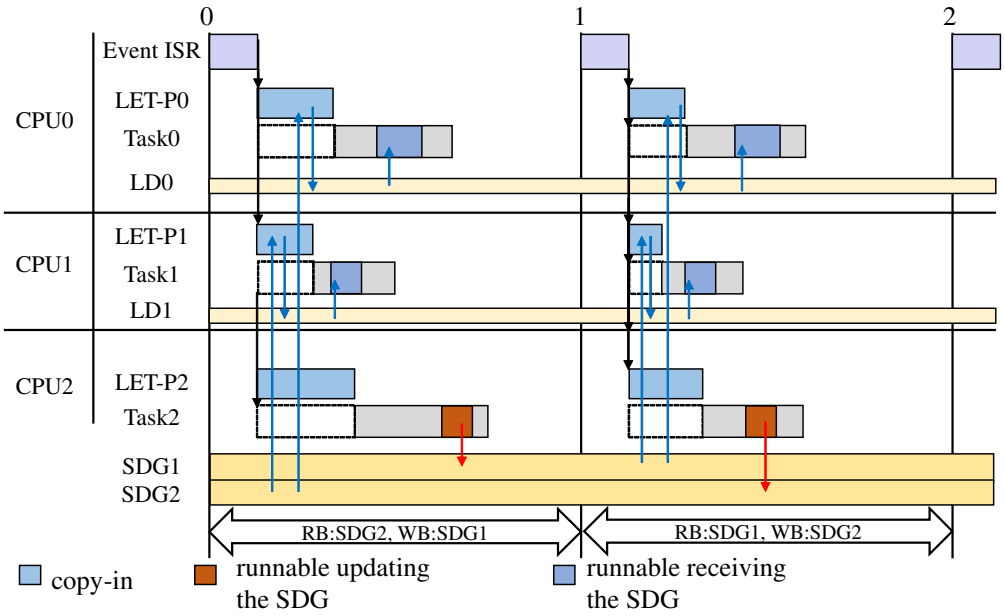


図 5.6: Asynchronized Distributed LET Process

められたタイミングによってリードバッファを変更することができないからである．したがって本手法の適用対象はハードリアルタイムタスクに限定される．

### 5.5.2 Synchronized Distributed LET Process (SDLP)

ソフトリアルタイムタスクが存在する場合に適用する分散手法として，SDLP(Synchronized Distributed LET-P) を提案する．SDLP は DMT-LET を適用しつつ，LET-P の制約を満たすために LET-P 間で同期を行う手法である．図 5.7 に SDLP による LET 通信の分散を示す．各 LET-P では，スワップおよびデータ更新フラグのチェックおよび False への更新を行い，その後他の CPU に配置された LET-P による同様の操作が完了するまで同期によって待つ．これらの操作がすべて完了した後，各 LET-P はコピーインを実施する．このようにして，LET-P を分散しつつ，スワップとコピーインの実行順序の制約を満たすことを可能とする．SDLP はスワップおよびデータ更新フラグのチェックおよび False への更新の操作が必要であり，同期による待ち時間が発生することから ADLP よりも LET-P の実行時間は長くなることに注意する必要がある．

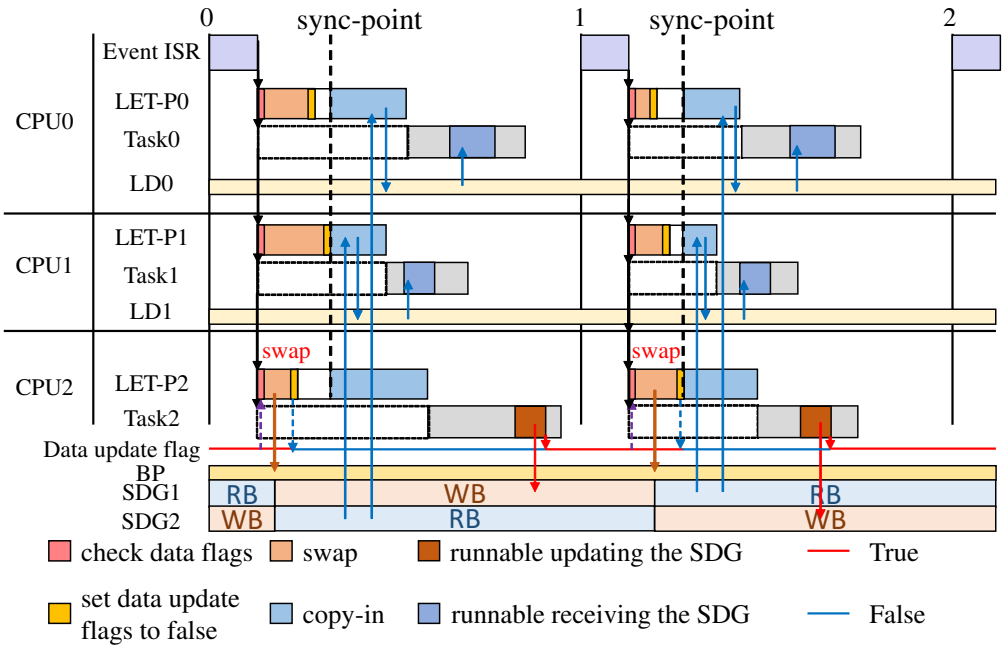


図 5.7: Synchronized Distributed LET Process

### 5.5.3 Hybrid Distributed LET Process (HDLP)

ADLP は DMT-LET との両立は困難であるが，バッファポインタのスワップや同期が不要なことから LET-P の実行時間の削減効果が大きいと考えられるため，ADLP のコンセプトをできるだけ利用したい．対象のパワトレアプリではハードリアルタイムタスクおよびソフトリアルタイムタスクが混在しているため，ハードリアルタイムタスクによって更新される SDG に対しては ADLP を適用し，一方ソフトリアルタイムタスクによって更新される SDG に対しては SDLP を適用することを考える．この考えに基づき，SDLP と ADLP を組合せた LET-P の分散方法として，Hybrid Distributed LET Process (HDLP) を提案する．図 5.8 に HDLP が適用されたときの LET-P を示す．各 LET-P は，ソフトリアルタイムタスクによって更新される SDG について，データ更新フラグのチェック，バッファポインタのスワップおよびデータ更新フラグの False への更新を行い，同期フラグを立てる．その後，ハードリアルタイムタスクによって更新される SDG に対するコピーインは LET-P 間で同期される必要がないため，すべての LET-P の同期フラグが揃うのを待たずに式 (5.1) で選択した RB に対して行われる．すべての同期フラグが揃った後は，ソフトリアルなタスクによって更新される SDG に対するコピーイ

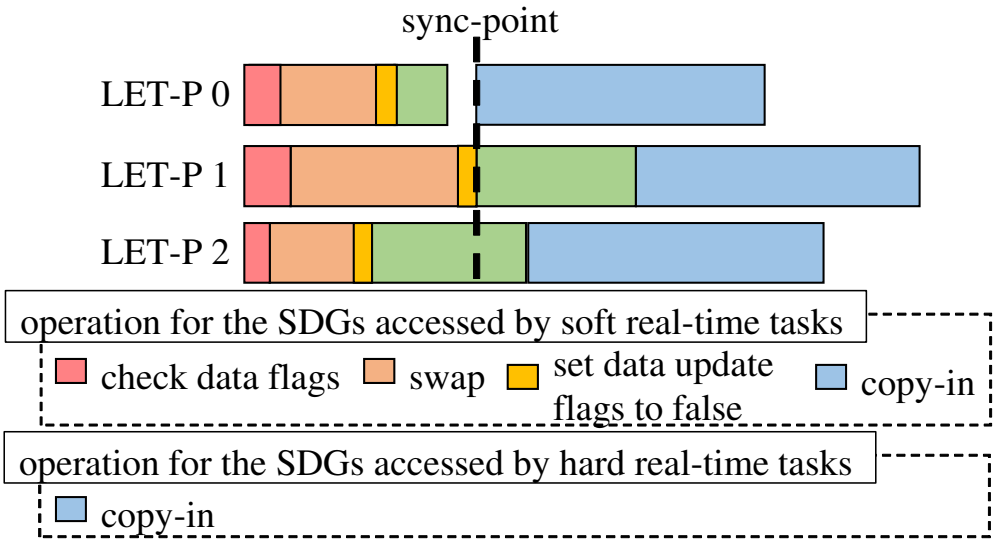


図 5.8: Hybrid Distributed LET Process (HDLP)

ンを開始する．また，SDG を更新するランナブルがハードリアルなタスクに配置されている場合は式 (5.1) で選択した WB に対して更新を行い，ソフトリアルタイムタスクに配置されている場合は WBP が指し示す WB に対して更新を行う．

バッファポインタのスワップの回数の削減と同期による待ち時間の短縮により，HDLP は DMT-LET によるデッドラインミスへの対応をしながら SDLP より LET-P の実行時間を削減することが可能である．

## 5.6　評価

本論文では，SDLP，ADLP，HDLP を適用した場合のメモリ使用量および LET-P の CPU 利用率を評価する．また，比較対象として 1 つタスクで LET-P を実装した場合 (単一タスク) についても評価を行う．ここで，ADLP は本来 R3 を満たさないため適用することはできないが，すべてのタスクがハードリアルタイムタスクであると仮定で適用している．

### 5.6.1　評価環境

評価環境となるハードウェアは，AURIX アーキテクチャを持つ評価用ボードである TC299B である [14]．AURIX アーキテクチャは，最大 300MHz で動作する CPU

を 3 つ持ち，各 CPU は自身のローカル RAM に対して短い遅延でアクセスする一方，他の CPU のローカル RAM やグルーバル RAM に対するアクセスには長い遅延時間が必要となる．また，ソフトウェアモデルは 2.1.1 節で説明したソフトウェア構成とスケジューリング機構を持つパワトレアプリを想定している．評価ソフトウェアは実際のパワトレアプリの構成を模擬したものである．評価ソフトウェアには数十個の時間同期タスクと数個の回転角同期タスクが含まれ，LET-P は SDLP，ADLP，HDLP が適用されている．また，評価ソフトウェアは AUTOSAR OS の 1 つである TOPPERS/ATK2-SC1-MC [20] 上で動作する．ここで，回転角イベントの発生間隔は 4,000rpm でエンジンが動作しているという想定で回転角イベントの発生頻度が設定されている．また，LET 通信はダブルバッファによる実現方法を想定しており，各コアごとに時間イベント，回転角イベントに対応する LET 処理を行う LET タスクを用意する．各 SDG は最もその SDG に高い頻度でアクセスする CPU が持つローカル RAM に配置されている．また，評価に用いられるスピンロック獲得・解放 API オーバヘッドと各コアからのメモリアクセス時間は実機の KIT_AURIX_TC299_TRB 上で計測した．

### 5.6.2　メモリ使用量の評価

ここでは，SDLP，ADLP，HDLP を用いて実装した場合のメモリ使用量の評価を行う．各手法におけるメモリ使用量を図 5.9 に示す．結果，それぞれの手法でメモリ使用量に大きな差はなかった．SDLP を用いた実装では，メモリ使用量の観点では単一 LET-P で実装した場合と差はないため，それらのメモリ使用量は同じである．次に，ADLP を用いた実装では，WBP，RBP およびデータ更新フラグが不要となるため，すべての実装方法で一番メモリ使用量が小さくなる．最後に，HDLP を用いた実装では，一部の SDG について WBP，RBP およびデータ更新フラグが不要となるため，SDLP を用いた場合と ADLP を用いた場合の中間程度のメモリ使用量となる．

### 5.6.3　LET 処理の CPU 利用率の評価

ここでは LET-P に ADLP，SDLP および HDLP を適用した場合の全 LET-P 合計の CPU 利用率の評価を行う．まず，すべてのタスクがハードリアルタイムタスクであるという想定で評価を行う．これは ADLP を適用することと等価である．次

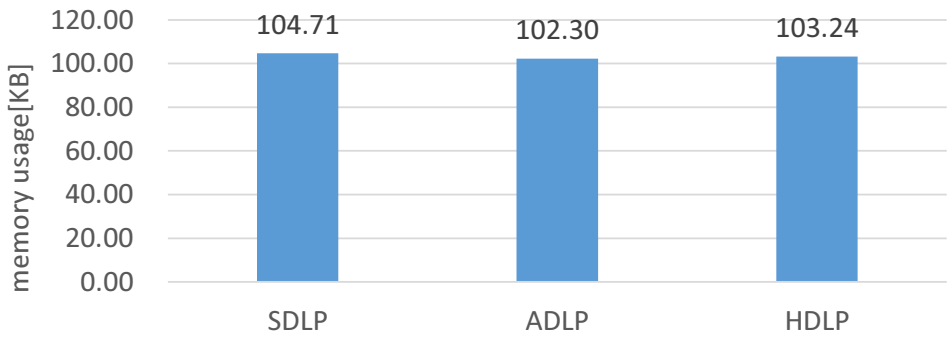


図 5.9: 各通信方法におけるローカルメモリに置かれるデータの合計サイズ

に，低優先度，つまりソフトリアルタイムタスクとして扱われる可能性の高いタスクから順にソフトリアルタイムタスクに変更していく．このとき，ソフトリアルタイムタスクとハードリアルタイムタスクが混在するため，HDLP を適用する．最終的に，すべてのタスクがソフトリアルタイムタスクとなる．これは SDLP を適用することと等価である．

結果を図 5.10 に示す．LET-P の合計実行時間はソフトリアルタイムタスクの比率に比例して増加している．したがって，HDLP は SDLP に比べて LET-P の実行時間を削減することができている．また，LET-P の実行時間は ADLP を適用した場合に最も短くなる．しかしながら，ハードリアルタイムタスクのみのシステムは現実的ではない．したがって，LET-P の実行時間とタスクが本当にハードリアルタイムタスクである必要があるかどうかを考えてソフトリアルタイムタスクおよびハードリアルタイムタスクの割当てを決定していく必要がある．

## 5.7　関連研究

LET 通信を車載制御ソフトウェアに適用する試みはすでに行われてきている．現在，LET はすでに AUTOSAR CP R4.4.0 に仕様として組み込まれている [43]．AUTOSAR では，LET 区間の設定方法についての仕様が決められており，区間の長さや開始するタイミングを決定するオフセットによって LET 区間を定義する．また同じ LET 区間を持つランナブルをエクスキュータブルエンティティクラスタ（EEC）として定義している．AUTOSAR 仕様で定められる LET 区間は必ずしも連続的に繰り返される必要はないため，本論文で提案したサブスケジューリングに対応した LET 区間の設定方法を適用することは可能である．その場合，サブレ

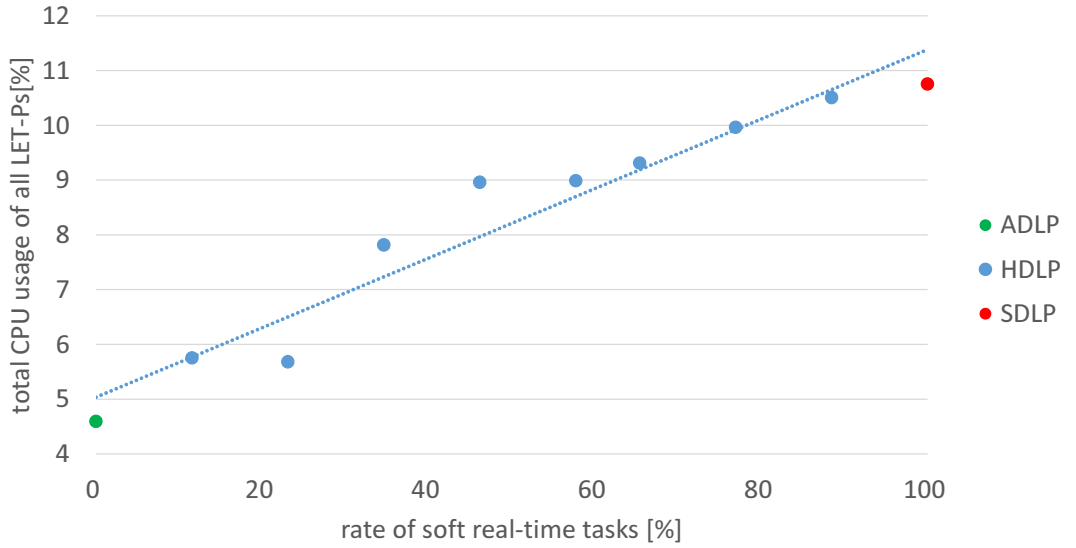


図 5.10: 分散方法ごとの LET 処理の CPU 利用率

イヤを EEC とみなし，LET 区間をそのサブレイヤを含むタスクの周期，最初に LET 区間が開始される時間をオフセットによって定義する．

また，WATERS Industrial Challenge 2017 [44, 7, 45, 46, 47, 48] では，LET の実装や評価の手法についての検討が行われた．ここでは，本論文でも触れた LET のデータ通信を専用のタスクで実現する手法 [7] の他にも，LET の通信用のランナブルをタスクに挿入する方法 [48] が提案されている．また，end-to-end latency [23] の評価手法 [7, 46, 47, 48] や LET の適用によるメモリ使用量の評価 [7, 47] についても議論されている．文献 [45] では LET のモデルを組み込みリアルタイムシステム向け同期言語である PRELUDE [49] に変換することについて述べている．また，文献 [48] では LET による通信のタイミングを解析し，必要なタイミングでのみデータの送受信を行う手法について説明している．デッドラインミスが生じた場合には通信のタイミングは変化してしまうため，この方法を実際のパワトレアプリに適用する場合にはハードリアルタイムタスク間の通信に限定されると考えられる．

最後に，Resmerita らは LET によって追加で必要となるローカル領域を削減する手法を適用することで LET の効率化を実現している [8]．この手法では，LET による通信のタイミングを静的に解析し，通信を行うタスクの LET 区間の重なり方に応じて読み込み側もしくは書き込み側のバッファを取り除く．この手法を実際のパワトレアプリに適用する場合には，デッドラインミスにどのように対応してい

くかを検討していく必要がある．

## 5.8　まとめ

本論文では，LET の実装方法を示すことで課題 (b)(c) に対処した．具体的には，サブスケジューリング下のサブレイヤの LET 区間を，通信の遅延を抑えるためにサブレイヤの実行タイミングとそのサブレイヤを含むタスクの次の実行までの区間として設定した．また，タスクがデッドラインミスしたときの影響を軽減する手法として DMT-LET を提案した．そして LET-P を分散し，各 CPU の負荷を軽減する手法として ADLP，SDLP，HDLP を提案した．ADLP はハードリアルタイムタスクのみで構成されたアプリに適用できる手法であり，各 CPU の負荷の軽減効果が大きい．SDLP は DMT-LET を適用しつつ，LET-P を分散する手法であり，同期による CPU 負荷の増大が課題となる．HDLP はハードリアルタイムタスクとソフトリアルタイムタスクが混在しているアプリを想定しており，SDLP より CPU の負荷の軽減効果が大きい．

評価として，ADLP，SDLP および HDLP を適用した場合の LET-P のメモリ使用量および CPU 利用率の計測を行った．評価の結果，各手法でのメモリ使用量にほとんど差はないが，SDLP を適用した場合と比較して，HDLP を適用した場合はメモリ使用量が少なかった．また，HDLP によって SDLP よりも LET-P の実行時間を削減できることを確認した．

# 第6章　結論

## 6.1　まとめ

本論文では，既存のマルチコア向けパワトレアプリ開発のフローの課題 (a)〜(j) を示し，それぞれの課題に対処できる開発フローを提案した．提案する開発フローでは，まず使用するマルチコアアーキテクチャを決定したら，関連度の高い処理をグルーピングし，ツールによる静的解析結果を用いてマッピングパターンを選定する．次にツールによって同期が不要なランタイムを自動生成する．ここでランタイム生成には，通信タイミングが決定的なランタイム構造を用いる．最後に，得られたランタイム実装を実機によって動作確認し，最終的な実装とする．

本論文では提案する開発フローを実現するために研究 (I)(II)(III) を行った．

研究 (I) は，複数のマルチコアハードウェアアーキテクチャに効率的に対応することを可能とするランタイム自動生成ツールである PMPF の提案である．PMPF では，ソフトウェアの構成，ハードウェアの構成のモデルおよび処理，データのコア，メモリマッピングを入力として与え，元のアプリの処理の依存関係を保持したまま処理をタスクにマッピングする．研究 (I) によって課題 (a)(c)(d)(g) に対応することが可能となった．評価によって PMPF によって少ないモデル記述の変更によって使用するハードウェアアーキテクチャを変更したり，処理のコアマッピングを変更することが可能であることを示した．また，PMPF によって必要なタスク数を抑え，タスク起動による OS オーバヘッドを削減することが可能であることを示した．さらに実際の規模のパワトレアプリに対して現実的な実行時間でランタイムを生成することが可能であることを示した．これらにより，PMPF がパワトレアプリの構成を柔軟に変更することを可能にするということが示された．

研究 (II) は，最悪余裕時間を静的解析した結果を用いて処理のコアマッピングを決定する手法の提案である．本手法では，依存関係や必要なスピンロック数を考慮して公開関数の集合を PF グループに分割し，PF グループ単位でのコア割り当てを評価することによってコア割り当ての探索空間を限定する．そして各 PF グループのコア割り当てに対してシステム最悪余裕時間を算出し，静的解析によるシ

ステム最悪余裕時間が上位である割り当てを実機評価を行う対象とする．研究 (II) によって課題 (e)(f)(h)(i)(j) に対応することが可能となった．評価によって，現実的な時間で静的解析を行うことができることを示した．また，実機実行と静的解析による解析値を比較して，両者に相関があることを示し，静的解析によるシステム最悪余裕時間が上位であるマッピングが実機実行による評価に用いるマッピングに適していることを示した．そして各オーバヘッドを考慮したときの解析による解析値を評価することにより，マルチコア上のパワトレアプリのシステム最悪応答時間の計算する際にスピンロックオーバヘッド，共有データへのメモリアクセス時間，タスク起動 ISR オーバヘッド，タスク切り替えオーバーヘッドを考慮する必要があることを示した．また，スピンロックのアイドル時間による影響は低く，これを考慮せずとも解析結果が実機実行より厳しい結果となるという要件を満たすことが可能であることを示した．これらにより，本手法のシステム最悪余裕時間の解析によって現実的な時間で実機評価を行うための有用な処理のコアマッピングを得られるということが示された．

研究 (III) は，実際のパワトレアプリに LET を適用する手法についての提案である．実際のパワトレアプリに LET を適用するためには，サブスケジューリングへの対応とソフトリアルタイムタスクのデッドラインミスへの対応という課題がある．前者については，LET 区間を最低限必要な長さに限定することで，共有データの更新をできるだけ早く反映させることを可能とした．後者については，デッドラインミスに対応した LET の機構として DMT-LET を提案した．DMT-LET によって，ダブルバッファリングが適用された場合においてもデッドラインミスによる不整合なデータの発生を防ぐことを可能とした．また，LET を実現するための処理である LET-P には，全 CPU に対するオーバヘッドが大きいという問題があったため，LET-P を複数の CPU に効率的に分散する手法である HDLP を提案した，研究 (III) によって課題 (b)(c) に対応することが可能となった．評価によって，HDLP は単一の LET-P で LET を実現する場合と比較して各 CPU の実行オーバヘッドを大幅に削減できることを示した．これらにより，実際のパワトレアプリに LET を適用する際に生じる課題に対応するための手法が示された．

以上により，課題 (a)〜(j) に対応した開発フローを実現することが可能となった．

## 6.2　今後の課題

上記の 3 つの研究に対する今後の課題を以下に示す．研究 (I) に対する今後の課題として，PMPF の機能を AUTOSAR の RTE ジェネレータに組み込むことで設計効率を向上させることが挙げられる．近年，AUTOSAR 仕様を用いた車載ソフトウェア開発が一般化されてきている．PMPF は AUTOSAR OS を用いることを前提としているものの，AUTOSAR RTE の仕様に準拠しているわけではない．そのため，様々な車載ソフトウェア開発に本手法を適用できるように，RTE ジェネレータを拡張して PMPF のタスク化単位の抽出や排他メカニズムの選択の手法を可能とすることが求められると考える．

研究 (II) 対する今後の課題として，公開関数内部の分岐や本論文では考慮していないスピンロックアイドル時間を考慮することで，より精度の高い解析を実現することが挙げられる．また，本論文の静的解析手法はクラスタ化された複雑なアーキテクチャを想定していない．しかし，今後パワトレアプリに対する要求がさらに高まり，8 コアを超えるようなマイコン上にパワトレアプリを実装することが求められるようになる可能性がある．したがってそのような複雑なアーキテクチャに対応できるような静的解析手法が求めれれるようになると考える．

研究 (III) に対する今後課題として，LET の end-to-end latency を軽減できるような手法を提案することが挙げられる．長い LET 区間内での LET による共有データ通信は，ランナブルがデータを生成してから実際に他のランナブルに公開されるまでの遅延が非常に長くなる．このような問題に対処するには，あらかじめランナブル間の実行順序を決定し，ランナブル間の通信を Implicit 通信に置き換えるという方法が考えられるが，この方法では並列性を犠牲にすることと，アプリの複雑さを増加させるという問題がある．LET に関する他の課題としては本論文の PMPF と静的解析手法に LET を対応させることが挙げられる．

# 謝辞

本論文に関する研究活動を通じて，学部 4 年からの長期に渡って多大な御指導，御鞭撻をして頂きました，名古屋大学大学院情報学研究科情報システム学専攻の高田広章教授，本田晋也准教授に，心から感謝申し上げます.

本論文の執筆にあたり，貴重な時間を割いて論文全体の構成について御指導，御助言をして頂きました名古屋大学大学院情報学研究科情報システム学専攻の枝廣正人教授に深く感謝致します.

日頃より研究生活についての御支援や研究についての議論をして頂きました名古屋大学大学院情報学研究科情報システム学専攻の松原豊准教授を始めとした高田・松原研究室および枝廣・本田研究室の皆様に厚く御礼申し上げます.

本研究を進めるに当たり，多くの御支援，御助力をして頂きましたトヨタ自動車株式会社および株式会社サニー技研の皆様に深く御礼申し上げます.

最後に，長期に渡る研究生活を支えてくれた家族に対し心から感謝致します.

# 参考文献

[1] 黒川文子 et al. Ev へのシフトと co 排出量に関する考察. **環境共生研究**, (11):25–36, 2018.

[2] 内田英明, 藤井秀樹, and 吉村忍. マルチエージェント交通シミュレーションにおける充電を考慮した ev の経路選択. **人工知能学会論文誌**, 32(5):AG16–I_1, 2017.

[3] Miloš Panić, Sebastian Kehr, Eduardo Quiñones, Bert Boddecker, Jaume Abella, and Francisco J Cazorla. Runpar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2014 International Conference on*, pages 1–10. IEEE, 2014.

[4] Lothar Michel, Torsten Flaemig, Denis Claraz, and Ralph Mader. Shared SW development in multi-core automotive context. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[5] Thomas A Henzinger, Benjamin Horowitz, and Christoph M Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[6] 飯山真一, 高田広章, 菅沼英明, et al. エンジン制御システムのためのリアルタイム性検証手法. **情報処理学会論文誌**, 43(6):1915–1724, 2002.

[7] Alessandro Biondi, Paolo Pazzaglia, Alessio Balsini, and Marco Di Natale. Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores. *The WATERS industrial challenge 2017*, 2017.

[8] Matthias Beckert, Mischa Möstl, and Rolf Ernst. Zero-time communication for automotive multi-core systems under SPP scheduling. In *Emerging*

*Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, pages 1–9. IEEE, 2016.

[9] 飯山真一, 遠藤友悟, 高田広章, 菅沼英明, et al. Rma 手法のエンジン制御システムへの適用に関する研究. **情報処理学会研究報告システムソフトウェアとオペレーティング・システム** *(OS)*, 2001(21 (2000-OS-086)):51–58, 2001.

[10] 加藤光治. **図解 カーエレクトロニクス *[*上*]* システム編**. 日経 BP 社, 2010.

[11] Robert N Charette. This car runs on code. *IEEE spectrum*, 46(3):3, 2009.

[12] 櫻井剛. 自動車の組込みソフトウェアの現状: Autosar および iso 26262 (組込みシステムの信頼性・安全性). **日本信頼性学会誌 信頼性**, 36(4):197–205, 2014.

[13] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[14] AURIX™ Family. `https://www.infineon.com/cms/jp/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc2xx/`. Accessed: 2019-1-15.

[15] NXP Semiconductors. MPC5777M. `https://www.nxp.com/jp/products/processors-and-microcontrollers/power-architecture-processors/mpc5xxx-55xx-32-bit-mcus/ultra-reliable-mpc57xx-32-bit-automotive-and-industrial-microcontrollers-mcus/ultra-reliable-mpc5777m-mcu-for-automotive-industrial-engine-management:MPC5777M`. Accessed: 2019-1-15.

[16] Infineon Technologies AG. AURIX™ 2nd Generation - TC3xx. `https://www.infineon.com/cms/jp/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/`. Accessed: 2019-1-15.

[17] 宮田仁, 島岡護, 見神広紀, 西博史, 鈴木均, 木村啓二, 笠原博徳, et al. 自動車リアルタイム制御計算の複数クラスタ構成マルチコア上での並列化. **研究報告システム・アーキテクチャ** *(ARC)*, 2017(30):1–6, 2017.

[18] AUTOSAR. https://www.autosar.org/. Accessed: 2018-11-26.

[19] TOPPERS Project. TOPPERS/OSEK カーネル. https://www.toppers.jp/osek-os.html. Accessed 2017-4-17.

[20] TOPPERS/ATK2. http://www.toppers.jp/atk2.html. Accessed: 2018-6-29.

[21] 飯山真一, 高田広章, et al. システム構成を考慮した can の最大遅れ時間解析手法. **情報処理学会論文誌コンピューティングシステム** *(ACS)*, 45(SIG01 (ACS4)):66–76, 2004.

[22] Aloysius K Mok and Deji Chen. A multiframe model for real-time tasks. *IEEE transactions on Software Engineering*, 23(10):635–645, 1997.

[23] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[24] AUTOSAR Classic Platform Release 4.4.0 SRS Operating System. https://www.autosar.org/standards/classic-platform/classic-platform-440/. Accessed: 2019-1-15.

[25] Dan Umeda, Yohei Kanehagi, Hiroki Mikami, Akihiro Hayashi, Keiji Kimura, and Hironori Kasahara. Automatic parallelization of hand written automotive engine control codes using oscar compiler. In *17th Workshop on Compilers for Parallel Computing (CPC2013)*, 2013.

[26] Denis Claraz, Franck Grimal, T Laydier, Ralph Mader, and Gerhard Wirrer. Introducing multi-core at automotive engine systems. *ERTS2, Feb*, 2014.

[27] Ralph Mader, Armin Graf, and Gerd Winkler. Autosar based multicore software implementation for powertrain applications. *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, 8(2015-01-0179):264–269, 2015.

[28] Thomas Fuhrman, Shige Wang, Marek Jersak, and Kai Richter. On designing software architectures for next-generation multi-core ecus. *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, 8(2015-01-0177):115–123, 2015.

[29] 相庭裕史, 本田晋也, 高田広章, et al. 対称型マルチコアシステムのエンジン制御ソフトウェアへの適用. **情報処理学会論文誌**, 51(12):2238–2249, 2010.

[30] Aurélien Monot, Nicolas Navet, Bernard Bavoux, and Françoise Simonot-Lion. Multicore scheduling in automotive ecus. In *Embedded Real Time Software and Systems-ERTSS 2010*, 2010.

[31] 小川真彩高, 本田晋也, 高田広章, et al. 車載制御システム向けマルチコアプログラミングフレームワーク. **情報処理学会論文誌**, 58(2):507–520, 2017.

[32] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[33] TOPPERS Project. ATK2 性能評価結果 NiosII 版 (online). `http://www.toppers.jp/a-osbench.html`. Accessed 2017-4-17.

[34] Wenhao Wang, Sylvain Cotard, Fabrice Gravez, Yael Chambrin, and Benoit Miramond. Optimizing application distribution on multi-core systems within autosar. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[35] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[36] David E Goldberg. Genetic alogorithms in search. *Optimization & Machine Learning*, 1989.

[37] Andreas Sailer, Stefan Schmidhuber, Michael Deubzer, Martin Alfranseder, Matthias Mucha, and Jürgen Mottok. Optimizing the task allocation step for multi-core processors within autosar. In *Applied Electronics (AE), 2013 International Conference on*, pages 1–6. IEEE, 2013.

[38] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[39] Saoussen Anssi, Sara Tucci-Piergiovanni, Stefan Kuntz, Sébastien Gérard, and François Terrier. Enabling scheduling analysis for autosar systems. In *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 152–159. IEEE, 2011.

[40] MAST website (Mast.unican.es).

[41] Abir Ben Khaled, Mohamed El Mongi Ben Gaïd, Daniel Simon, and Gregory Font. Multicore simulation of powertrains using weakly synchronized model partitioning. In *E-COSM'12 IFAC Workshop on Engine and Powertrain Control Simulation and Modeling*, 2012.

[42] Rafael Ubal, Julio Sahuquillo, Salvador Petit, and Pedro Lopez. Multi2sim: A simulation framework to evaluate multicore-multithreaded processors. In *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)*, pages 62–68. IEEE, 2007.

[43] AUTOSAR Classic Platform Release 4.4.0 RS TimingExtensions. `https://www.autosar.org/standards/classic-platform/classic-platform-440/`. Accessed: 2019-1-15.

[44] The industrial challenge of WATERS 2017. `http://waters2017.inria.fr/challenge/`. Accessed: 2018-6-29.

[45] Frédéric Boniol, Julien Forget, and Claire Pagetti. WATERS Industrial Challenge 2017 with Prelude. *The WATERS industrial challenge 2017*, 2017.

[46] Juan M Rivas, J Javier Gutiérrez, Julio L Medina, and Michael González Harbour. Comparison of Memory Access Strategies in Multi-core Platforms Using MAST. *The WATERS industrial challenge 2017*, 2017.

[47] Kai Björn Gemlau, Johannes Schlatow, Mischa Möstl, and Rolf Ernst. Compositional Analysis of the WATERS Industrial Challenge 2017. *The WATERS industrial challenge 2017*, 2017.

[48] Jorge Martinez, Ignacio Sañudo, Paoloa Burgio, and Marko Bertogna. End-To-End Latency Characterization of Implicit and LET Communication Models. *The WATERS industrial challenge 2017*, 2017.

[49] J. Forget, F. Boniol, D. Lesens, and C. Pagetti. A multi-periodic synchronous data-flow language. In *Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium (HASE '08)*. IEEE, 2008.

# 研究業績

## 学術論文

- 小川真彩高 , 本田晋也, 高田広章, 車載制御システム向けマルチコアプログラミングフレームワーク, 情報処理学会論文誌, Vol.58,No.2, pp. 507-520, Feb 2017.

- 小川真彩高, 本田晋也, 高田広章, マルチコアパワートレインアプリケーションにおけるコア配置決定のための最悪応答時間解析手法, 情報処理学会論文誌, Vol.59,No.2, pp. 748-761, Feb 2018.

## 国際会議論文（査読あり）

- Masataka Ogawa, Shinya Honda, Hiroaki Takada, Efficient Approach to Ensure Temporal Determinism in Automotive Control Systems, 8th International Symposium on Embedded computing and system Design, Dec 2018

## 研究会発表論文（査読なし）

- 小川真彩高, 本田晋也, 高田広章, 車載制御向けマルチコアプログラミングフレームワーク, 第 135 回 OS・第 39 回 EMB 合同研究発表会 , Vol.2015-EMB-39,No.9, pp. 1 - 6 , お茶の水女子大学, Nov 2015.

- 小川真彩高, 本田晋也, 高田広章, 車載マルチコアシステムの性能見積もり手法, 第 217 回システム・アーキテクチャ・第 179 回システムと LSI の設計技術・第 44 回組込みシステム合同研究発表会, Vol.2017,No.11, pp. 1-6, 沖縄, Mar 2017.

- 小川真彩高, 石野正敏, 岡本卓也, 山本椋太, 本田晋也, システムレベル設計環境による Trax プレーヤの設計事例, 平成 29 年度 電気・電子・情報関係学会 東海支部連合大会, pp. C5-4, 名古屋, Sep 2017.

- Masataka Ogawa, Yusuke Kato, and Shinya Honda: ”CAN Controller Sharing Mechanism for Mixed Critical Systems”, Embedded Systems Workshop 2017, 東京, Dec 2017

- 小川真彩高，本田晋也，高田広章，車載制御システムにおける時間的決定性保証手法の検討, 第 48 回組込みシステム研究発表会, 東京，Jun 2018
- 小川真彩高，本田晋也，高田広章，車載制御システムを対象とした Logical Execution Time の実現手法, 組込み技術とネットワークに関するワークショップ (ETNET2019)，鹿児島，Mar 2019