Vol.2015-OS-135 No.9
Vol.2015-EMB-39 No.9
2015/11/24

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

小川 真彩高[1,a)]　本田 晋也[1]　高田 広章[1]

概要：エンジン等を制御するパワトレアプリは高機能化が著しく，マルチコアによる処理が必要になるが，車種ごとにハードウェアのアーキテクチャのバリエーションが異なるため，そのすべてに対応させるのは難しいと予想される．そこで本研究では，パワトレアプリをモデル化し，そのモデルからランタイムを自動生成することにより，アーキテクチャの変更を可能とするフレームワークを開発した．実際のパワトレアプリの一部に本フレームワークを適応し，同一のモデルから，2 プロセッサ及び 4 プロセッサ用のランタイムが生成できることを確認した．

キーワード：車載システム，マルチコア，コード生成

Multicore Programing Framework for Vehicle Control Software

Masataka Ogawa[1,a)]　Shinya Honda[1]　Hiroaki Takada[1]

Abstract: For corresponding significant high functionality on powertrain applications such as engine control applications, powertrain applications engineers consider to use multi-core architectures. However, because different car models have different hardware architectures, it is expected to be difficult to apply all car models. In this study, we developed the framework that enables changes in architectures by modeling powertrain applications and then making automatically runtimes from models. We confirmed to make runtimes for 2 processor architecture and 4 processor architecture by applying this framework to the same model that made from a part of actual powertrain application.

Keywords: Vehicle control System, Multicore, Code generation

1. はじめに

自動車の高性能化に伴って，車載制御が高機能化，複雑化してきている．特にエンジンやトランスミッションなどを制御するパワートレイン・アプリケーション (パワトレアプリ) ではその傾向が強い．そのため，シングルコアでは処理が間に合わず，コア数を増加させる動きがある [1], [2]．パワトレアプリは多くの車種で共通的に使用され，車種毎に使用するハードウェアのアーキテクチャが異なる．また，マルチコアのハードウェアアーキテクチャはコア数やメモリ構成等のバリエーションが多く，そのすべてに単一のソースコードで対応させることは難しい．そこで本研究では，現状のパワトレアプリの構成をベースにモデルを定義し，そのモデルからアーキテクチャに応じたランタイムを生成するフレームワークを実現した．そして実際のパワトレアプリの一部に本フレームワークを適応し，同一のモデルから，2 プロセッサ及び 4 プロセッサ用のランタイムが生成できることを確認した．

[1] 名古屋大学大学院情報科学研究科, 名古屋市
Graduate School of Information Science, Nagoya University, Nagoya-shi, 464-8603 Japan
[a)] masa-bach@ertl.jp

2. パワートレインアプリケーション

本章では，本研究で対象とするパワトレアプリに関して，現状のソフトウェア構成について述べた後，今後のパワトレアプリの課題について説明する．

2.1 パワトレアプリのソフトウェア構成

パワトレアプリとは，車載におけるエンジンやトランスミッション等のパワートレインと呼ばれる機能を制御する

ソフトウェアである．パワートレインは車載においてもっとも重要な部分であり，高いリアルタイム性や信頼性が求められる．パワトレアプリのアルゴリズムを MATLAB で検証し，検証されたアルゴリズムを C 言語で実装する．そのため，処理に暗黙の依存関係が存在する．また，現状のパワトレアプリはマルチコアによる並列性を考慮したソフトウェア構成ではない．

ここで，現状のパワトレアプリの構成について説明する．パワトレアプリでは，時間もしくはエンジンのクランク角に同期して処理が実行される．パワトレアプリは非常に機能が多く，多数のモジュールで構成されており，そのモジュールの中の機能毎に機能契機や周期，オフセットが異なる．そのため，機能の追加，周期変更を容易に実現するため，現状のパワトレアプリは図 1 のような構成を持っている．以下，パワトレアプリの各要素について説明する．

レイヤ　エンジン，トランスミッションなど，同一の機能を実現するための処理の集合である．

レイヤマネージャ　タスクや ISR として実現する，機能を構成する処理の集合である．

サブレイヤ　レイヤマネージャから呼び出される，同一の周期や優先度を持つ処理の集合である．1 つのタスクは複数のサブレイヤを持ち，タスク内でサブレイヤの実行順序を決定する．

レイヤ内公開関数 (公開関数)　サブレイヤから呼び出される関数である．複数のサブレイヤから呼び出されることもあり，そのため，リエントラントが制限されるものも存在する．

内部関数　公開関数から呼び出される関数で，処理の最小単位である．

現状のパワトレアプリはシングルプロセッサを前提としているため，複数の公開関数が同時に動くことはない．そのため，同一のレイヤマネージャにおいて常に依存関係を満たす順番に実行される．

センサを制御する処理で受け取ったセンサ値を別の処理で使用するなど，処理間でデータの引き渡しをする必要がある．データの読み書きは負荷が大きく，またある処理がデータを読み込んでいる間にデータの値が書き込まれ，変更されるとデータの値が不整合なものとなってしまう．そのため，現状のパワトレアプリでは，処理を実行する前にその処理が必要とするデータをまとめて読み込み，処理が終了した後に書き換えたデータをまとめて書き込むようになっている．また，処理の前後のデータの読み書きでは割込み禁止などの排他制御を行う．現状のパワトレアプリでは内部関数単位でデータの読み書きを行う．

2.2 パワトレアプリの課題

FLASH の内蔵，熱耐性などの理由から，パワトレ向けマイコンの上限クロックは 200-300MHz 程度であると言われている．しかしハイブリッド車におけるモーター制御など，複雑な制御を行う必要があり，シングルコアでは限界が来ているため，車載用マイコンもマルチコア化が進めれれている．コア数は現状の製品は 2 コアであるが，将来的には 8 コアの製品も予想される．また，メモリ構成もキャッシュを使用するとリアルタイム性が損なわれるため，コア毎のローカルメモリとグローバルメモリを組み合わせて，変数のメモリ配置を工夫することにより性能の向上を実現する必要がある．

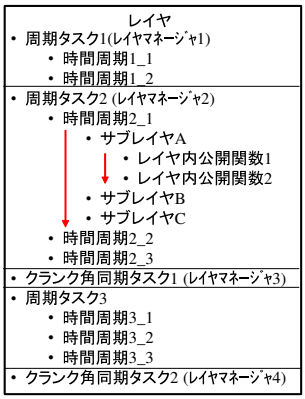


図 1　現状のパワトレアプリの構成

マルチコアに移行する際には，各公開関数をコアに割り付け，その上で依存関係を満たすようにソフトウェアを記述しなければならない．しかし，マルチコアのマイコンはシングルコアの時と比べコア数やメモリ構成のパターンが幅広くなるため，マイコン毎にソフトウェアを対応させることは困難である．また，処理のコア配置やデータのメモリ配置によってパフォーマンスが異なってくるため，これらを柔軟に変更できる仕組みが求められる．

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

前章で述べた課題に対応するため，現状のパワトレアプリの構成を元にパワトレアプリをモデル化し，2-8 コア程度の車載制御ソフト向けのマルチコアアーキテクチャを対象としたプログラミングフレームワーク，PMPF を提案する．

3.1 モデルおよび生成ツールの要件

ここでは，前述したパワトレアプリの課題に対応するために PMPF に必要な要件をまとめる．

- **モデル記述に関する要件**
  - (1) 既存のソフトの構成からあまり大きく変更せずにパワトレアプリの制御構造を記述できるモデルであること
  - (2) ハードウェアアーキテクチャに依存しない形で処理やデータの情報を記述できること
  - (3) 処理間の依存関係が記述できること

(4) 処理のコア配置やデータのメモリ配置といったマッピングを容易に変更できること

- **実装に関する要件**
  (5) タスク数をできるだけ少なくすること
  (6) スタックの使用量を抑えること
  (7) 排他処理は必要な箇所のみで実施すること
- **AUTOSAR OS の利用に関する要件**
  (8) AUTOSAR OS を使うこと
  (9) 待ち状態を使わないこと

(1) はパワトレアプリの構造変更により発生するコストを削減するためである．(2) はハードウェアの変更によってソフトウェアの情報を変更しなくても良いようにするためである．(3) は既存のパワトレアプリの処理が暗黙の依存関係を持つためである．(4) は評価によるマッピングへのフィードバックを容易にするためである．(5) はタスク起動による負荷を少なくするためである．(6) はパワトレ向けアーキテクチャの限られたメモリ資源を有効利用するためである．(7) は排他制御する処理を 2 つのコアから実行する場合はスピンロックが必要となるが，その処理が同じコアからしか呼び出されないことがわかっていれば割込み禁止のみで排他制御すればよいということになる．このように排他制御が必要な場合に適切な排他処理を判断することで，排他によるオーバヘッドを減らすことができる．(8) は車載ソフトウェアの標準となっている AUTOSAR と親和性の高いものにするためである．(9) は現状のパワトレアプリが待ち状態を使用していないためである．

3.2 実現方法検討

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
（b）公開関数毎の選択実装
（c）公開関数呼び出しコード生成

（a）は，公開関数毎にタスク化する方法で，タスク化した公開関数は実行するタイミングと依存関係が満たされたら起動される．タスクのコアへの割り当ては，プログラムではなく，OS のコンフィギュレーションを変更するだけで良いため，コア数やアーキテクチャに応じて容易に変更することができる．（b）はレイヤーマネージャを各コアに置いて，そのコアで実行するべき公開関数なら実行する方法である．公開関数のコア割り当てに応じてテーブルを生成すればよい．（c）はレイヤー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．

（a）（b）は追加のツールはなく実現できるが，実行オーバヘッドや使用メモリが大きくなると予想される．一方（c）は，ツールを作成する必要はあるが，オーバヘッドの問題は発生しないと予想される．パワトレアプリは，オーバーヘッドによるコストの低減が開発コストより重視されるため，本研究では（c）を採用する．

3.3 コンセプトと設計フロー

PMPF のコンセプトについて説明する．

パワトレアプリのモデル化　現状のパワトレアプリの構成を参考にしてパワトレアプリをモデル化する．モデルでは処理の依存関係や周期・オフセットなどの情報を扱い，処理の実装は別個に行う．

ランタイム生成　考案したモデルを入力としてランタイム生成を行うツールを作成する．ランタイムでは処理の実行タイミングや依存関係の整合性などの制御を行う．

マルチコアアーキテクチャへの柔軟な対応　ハードウェアの構成を記述し，そのハードウェア構成に従ってランタイムを生成する．ハードウェアの構成の記述を変更することで，柔軟に様々なハードウェア構成に対応できるようにする．

PMPF による設計フローを図 2 に示す．

まずパワトレアプリのモデル (SW モデル) と対象ハードウェアの構成の記述 (HW モデル)，マッピング情報を作成する．次に SW モデル，HW モデル，マッピング情報から PMPF のツールによってランタイムを生成する．ランタイムと処理の実体とリアルタイム OS から実行バイナリをビルドする．実行バイナリを実行して，実行結果の評価から，マッピングへのフィードバックを行う．

3.4 モデル

本章ではパワトレアプリのモデルについて説明する．現状のパワトレアプリの構成をベースにモデル化することで，既存のソフトウェア資源を効率的に利用できるようにする．モデルの概要を図 3 の [1] に示す．

3.4.1 モデルのオブジェクト

以下，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のような割込みから発生しタスクグループを起動する．現状のパワトレアプリにおけるレイヤに相当する粒度をもつ．

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レイヤマネージャに相当する粒度をもつ．

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レイヤに相当する粒度をもつ．

公開関数　処理の本体でモデル記述とは別に，公開関数の実装を C 言語で記述する．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．現状のパワトレア

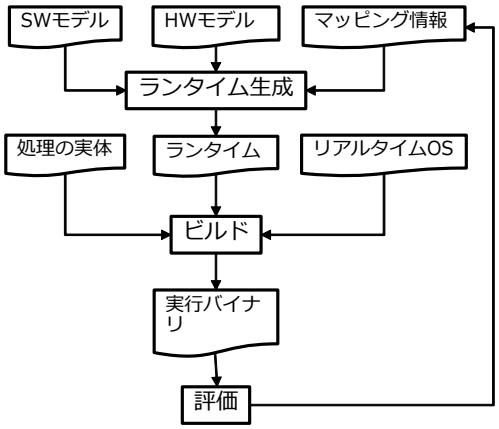


図 2 PMPF の合成フロー

プリにおけるレイヤ内公開関数に相当する粒度をもつ．

排他実行関数 複数の公開関数から呼び出される関数のうち，排他的実行の必要があるものであり，内部関数に相当する粒度を持つ．排他実行関数の設定により，以下のことを保証することができる．

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 実行が開始すると中断されない

リエントラント禁止，アトミック実行は排他実行関数それぞれに対して片方または両方設定できる．リエントラント禁止，アトミック実行の設定と排他実行関数を呼び出す公開関数の関係を参照し，自動的に必要な排他の実装方法を決めることができる．

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できる．

3.4.2 モデルの依存関係

既存のパワトレアプリには処理の依存関係が存在する．既存のパワトレアプリではレイヤマネージャによってサブレイヤの実行順序が決められ，サブレイヤによってレイヤ内公開関数の実行順序が決めるようになっている．しかし，処理の実行順序がレイヤマネージャ単位で決定されていると，マルチコアによる負荷分散の恩恵が少なくなってしまう．本研究では 2 つの依存関係を定めた．一つは，既存のパワトレアプリとの互換性のためにサポートする，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呼び出し順指定である．もう一つは既存のパワトレアプリにおけるサブレイヤ，レイヤ内公開関数に相当するタイミングマネージャ，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オブジェクト同士を並列に実行できるようにした先行制約である．これら 2 つの依存関係をまとめて実行開始条件と呼ぶ．

呼び出し順指定と先行制約について図 3[1] に従って説明する．呼び出し順指定には 3 つの種類がある．1.1 はタスクグループの呼び出し順指定である．各トリガーが起動するタスクグループを優先度順に起動する．図 3[1] の例では，タスクグループ 0 に含まれるタイミングマネージャのうち，そのタイミングで実行されるものがすべて終了した後，タスクグループ 1 が起動される．1.2 はタイミングマネージャの呼び出し順指定である．図 3[1] の例では，タイミングマネージャ 0, 1, 2 の順に実行されるように指定されている．しかしタイミングマネージャ 2 が実行されるタイミングではタイミングマネージャ 1 は実行されないため，そのタイミングではタイミングマネージャ 0,2 の順に実行され，タイミングマネージャ 1 は実行されない．1.3 は公開関数の呼び出し順指定である．図 3[1] の例では，タイミングマネージャ 0 内では公開関数 0_0, 0_1, 0_2 の順で実行される．2.1 はタイミングマネージャの先行制約である．図 3[1] の例では，タイミングマネージャ 3 内のすべての公開関数が実行された後，タイミングマネージャ 4 が実行可能となる．2.1 の先行制約がない場合はタイミングマネージャ 3，タイミングマネージャ 4 は並列して実行可能である．2.2.1 及び 2.2.2 は公開関数の先行制約である．図 3[1] の例では，2.2.1 はタスクグループを跨ぐ先行制約で，公開関数 1_1 が終了した後，公開関数 4_0 が実行可能となる．また，2.2.2 はタスクグループ内で完結する先行制約で，公開関数 5_1 が終了した後，公開関数 5_2 が実行可能となる．これらの依存関係がないタイミングマネージャ．公開関数同士は並列して実行可能である．

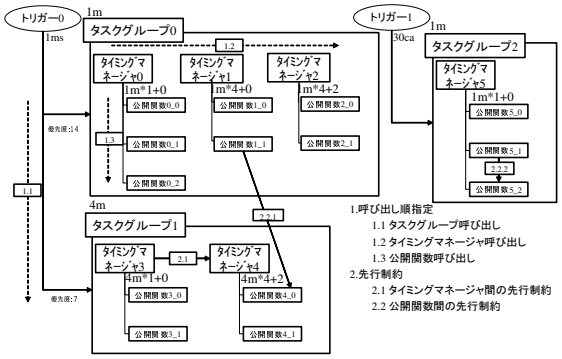
最終的には並列度の高い先行制約で実装することを想定しているが，このような並列性の高い依存関係に一度に対応することは困難であるため，まずは呼び出し順指定として，徐々に先行制約に移行していくことを想定している．

4. ランタイム生成

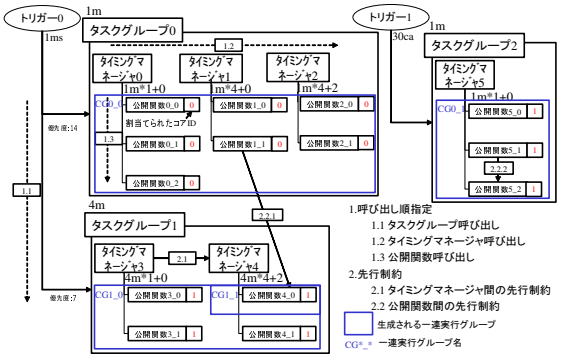
前章では，パワトレアプリの制御構造を容易に変更できるように，現状のパワトレアプリを参考にしてモデル化を行った．ユーザはパワトレアプリの構成を前章のモデルで記述し，公開関数や排他実行関数の処理を C 言語で実装する．本章ではモデルで示された制御構造やタイミングマネージャ/公開関数の依存関係を実現するランタイム生成について説明する．本研究ではリアルタイム OS を利用して実装を行うため，公開関数をタスクとして実装する．本章では，公開関数のタスク化，メモリアクセス，モデルオブジェクトのランタイム化について説明する．

4.1 公開関数のタスク化

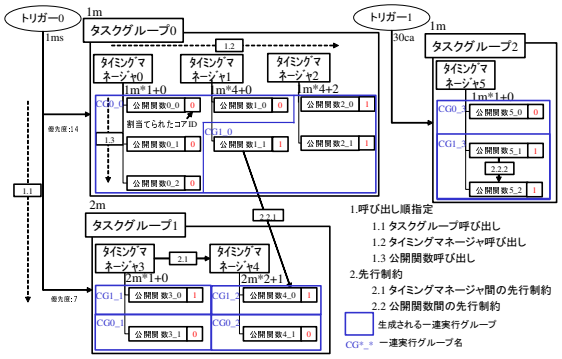
タイミングマネージャ単位でタスク化をする場合，タイ



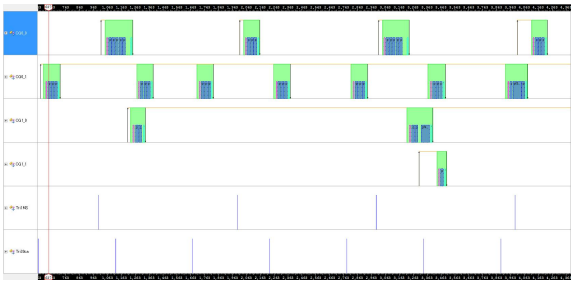
[1]PMPF の SW モデル



[2] 公開関数のコア割当て 1



[3] 公開関数のコア割当て 2



[4] コア割当て 1 でランタイム生成した場合の実行の様子

図 3 一連実行グループの生成と実行の様子

ミングマネージャ内で実行する公開関数のうち，そのタイミングマネージャ外に所属する公開関数が先行制約になっているものが存在する可能性があり，この場合タスクの途中で先行制約となる公開関数の実行を待つ必要が出てくる．公開関数単位でタスク化すれば，先行制約となる公開関数が終了したあとにタスクを起動し，タスク終了後にそのタスクで実行した公開関数が先行制約となっている公開関数を含むタスクを起動するため，タイミングマネージャ単位のときのような問題は起こらない．

しかし，公開関数を個別にタスク化すると，タイミングマネージャなどと比べタスク数が多くなる．タスク数が多いと，リアルタイム OS におけるオーバヘッドやタスク間の通信が増えるため，複数の公開関数を 1 つにまとめてタスク化する仕組みが必要となる．そこで，同じ実行開始条件をもつ公開関数の集合を一連実行グループと呼び，一連実行グループ単位でタスク化することにした．一連実行グループの実行を行うタスクを一連実行グループタスクと呼ぶ．同じ一連実行グループに含まれる条件は，同一のタスクグループに属していること (1)，同一のコアに割り当てられていること (2)，共通の実行開始条件を持つ公開関数であること (3) の 3 つである．(1) についてはタスク内に異なる優先度を持つ公開関数を含めないためである．(2) については，異なるコアに割り付けられた公開関数を同一のタスクに含むことは出来ないためである．(3) については，共通の実行開始条件をもつ公開関数を同じ一連実行グループに含めれば，タスク実行の途中で同期をとる必要がなくなるためである．図 3 の例では，タスクグループ 1 の公開関数がすべて同じコアに割り付けられた場合 (コア割当て 1, 図 3[2])，タスクグループ 0 が終了したあとに実行できる公開関数 3_0，3_1 と，公開関数 3_0，3_1 が終了したことで実行できるタイミングマネージャ 4 内の公開関数 4_1 が同じ一連実行グループとなる．公開関数 4_0 は公開関数 1_1 が終了しないと実行できないため，同じ一連実行グループには含まれない．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[3]) はコア割当て 1(図 3[2]) 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．コア割当て 1 のときの実行の様子を図 3[4] に示す．

4.2 メモリアクセス

PMPF のモデルでは，データアクセスは公開関数単位で記述するようにしている．共有データへのアクセスは公開関数の C ソースに PMPF が提供する専用 API を用いて行う．しかし，専用 API を呼び出すことで共有データから直にデータへアクセスするようにした場合，そのたびに排他制御を行う必要が出てくる．また，その共有データが複数回アクセスされる場合，メモリアクセスによるレイテンシの影響が大きくなる．そこで，PMPF によって生成されるランタイムでは，一連実行グループ単位で共有データ

アクセスを行うようにする．まず，一連実行グループに含まれている公開関数がアクセスする共有データに対応するローカルな変数を用意する．一連実行グループタスクが起動されると，一連実行グループに含まれる公開関数が読み込み指定している共有データをそのローカルな変数に読み込む．ランタイム生成時に公開関数の共有データ読み込みの重複を排除する．PMPF の API によって，公開関数内でローカルな変数への読み書きが行われる．共有データへの書き込みの実装は 2 つのパターンを用意する．1 つは一連実行グループタスク終了時に，一連実行グループに含まれる公開関数が書き込み指定している共有データの重複を排除して共有データ本体に書き込むというものである．もう一つは各公開関数を実行し終わった後に，その公開関数で書き込み指定している共有データについて共有データ本体に書き込むというものである．前者は共有データの通信回数を削減したい場合に使用し，後者は共有データの変更を即座に反映したい場合に使用する．

4.3 モデルオブジェクトのランタイム化

PMPF のランタイム生成ツールによって記述されたモデルをからランタイムを生成することができる．ランタイムではリアルタイム OS のコンフィグレーションファイル (1) とトリガーによるタスク起動処理 (2.1)，一連実行グループタスク (2.2)，タイミングマネージャ，公開関数の実行順序を解決するための同期機能 (2.3) などからなる中間コード (2) が生成される．(1) では，マッピング記述と一連実行グループ生成から一連実行グループタスクの登録とコア割り付けを行う．(2) は C 言語で記述されている．(2.1) ではそれぞれのトリガーから起動されるタスクグループ内のタイミングマネージャが起動するタイミングを確認し，実行されるタイミングマネージャを含む一連実行グループタスクを起動する．(2.2) ではその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，そうであれば実行する．また，前節で述べたように公開関数実行前に共有データの読み込みを行い，公開関数実行後に共有データの書き込みを行う．(2.3) については，まず，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 となっており，対応する実行開始条件が満たされた時に対応する実行開始フラグを TRUE とする．実行開始フラグ集合がすべて TRUE となったとき，対応する一連実行グループが起動される．公開関数が終了したとき，その公開関数が実行開始条件となっている一連実行グループの実行開始フラグを TRUE にする．また，何らかの一連実行グループの実行開始条件となる公開関数が終了するたびに実行されるタイミングである一連実行グループについて実行開始フラグ集合がすべて満たされ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起動する．

5. 評価

既存のパワトレアプリの一部を対象として，モデル化を行い，人手で 2 コア及び 4 コアにマッピング（Mapping1）して，提案ツールによりランタイムを生成した．評価結果を表 1 に示す．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，モデル記述は生成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．また 2 コアと 4 コアの記述の違いはマッピング指定のみであり，単一のモデル記述から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．また，公開関数のコア配置の見直しを行い評価を実施した（Mapping2）．実行時間は 2 コアでは 7.1%，4 コアでは 27.6%短縮することが出来た．マッピングを変更するのに必要な行数は 10 行以下であった．

次に公開関数ごとにタスク化するという他の実現方法と比較するために，ランタイム生成ツールにおいて一連実行グループの生成を行わず，公開関数ごとにタスク化することで，提案手法との比較を行った．評価結果を表 2 に示す．評価の結果，2 コア，4 コアで実行した場合のどちらでも 40%以上の実行時間の増加が見られた．このような結果になったのはタスク数が多くなったためにタスク起動などにかかるオーバヘッドが大きくなったことが挙げられる．

表 1 評価結果

Mapping	コア数	モデル [行数]	変更 [行数]	ランタイム [行数]	実行時間 [μsec]	改善率 [%]
1	2	1,660	-	16,295	815	-
	4	1,662	-	17,477	682	-
2	2	1,660	3	15,674	757	7.1
	4	1,664	9	17,202	494	27.6

表 2 公開関数単位でのタスク化との比較

コア数	タスク化単位	タスク起動回数	実行時間 [μsec]	悪化率 [%]
2	一連実行グループ	6	1447	-
	公開関数	10	2070	43
4	一連実行グループ	7	1303	-
	公開関数	10	1895	45

謝辞 本研究を進めるにあたりご協力いただいたトヨタ自動車株式会社に深く御礼申し上げます．

参考文献

[1] 相庭裕史, 本田晋也, 高田広章, ”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, Vol. 51, No. 12, pp. 2238-2249, Dec. 2010.
[2] Monot, Aurelien, et al. ”Multicore scheduling in automotive ECUs.” Embedded Real Time Software and Systems-ERTSS 2010. 2010.