情報処理学会論文誌 Vol.58 No.2 507–520 (Feb. 2017)

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

小川 真彩高[1,a)]　本田 晋也[1]　高田 広章[1]

受付日 2016年5月26日, 採録日 2016年11月1日

概要：車載制御システムのソフトウェアにおいて，エンジン等のパワートレインを制御するパワトレアプリは高機能化が著しく，マルチコアの導入が必要とされている．すでにパワトレアプリ用に複数のマルチコアのマイコンが存在するが，それぞれコア数やメモリ構成が異なり，車種ごとに適したマイコンを使用する．一方，パワトレアプリは開発コストが高いため，単一のソフトウェアをベースに少ない変更でこれらのマイコンをサポートしたいという要望がある．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，コア数や処理の配置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用し，同一のソフトウェアモデルから，マッピング記述やハードウェアモデルの変更のみでコア数や処理の配置の変更が実現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]

Received: May 26, 2016, Accepted: November 1, 2016

Abstract: Engineers have considered that it is necessary to use multi-core applications to apply high functionality in powertrains which control the likes of engines. However, as differing car models have different hardware architectures, it is expected to be difficult to apply this to all car models. In this study, we have developed a framework that enables changes in architectures by modeling powertrain applications and then automatically making runtimes from these models. We have confirmed that it is possible to create runtimes for the 2 processor architecture and 4 processor architecture by applying this framework to the same model that was made from a part of the actual powertrain application.

Keywords: vehicle control system, multicore, code generation

1. はじめに

自動車に搭載される複数の電子制御ユニット（ECU）のうち，エンジンやトランスミッションといったパワートレイン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）は，燃費の向上や排気ガス規制を満たすために重要な機能である．そ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 [1].

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．現在のパワートレインではシングルコアが用いられていることから，ソフトウェアの基本的な構成はどのマイコンでも大きく変わらず，条件コンパイルといった既存のプログラム言語の機構により対応している．

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

© 2017 Information Processing Society of Japan

燃費の向上，排ガス規制の強化，ハイブリットエンジンへの対応等，パワトレアプリへの要求は多く，今後も処理量が増加することが予想される [1]．一方，ソフトウェアを実行するマイコンはノイズや温度への耐性の関係で，微細化による動作周波数の向上が困難となってきており，上限は 300 MHz 程度といわれている．そのため，今後のパワトレアプリの高機能化を実現するには，マルチコアの利用が必要不可欠となる．すでに，2 ないし 3 個のコアで構成されたパワトレアプリ向けのマイコンがリリースされている [2], [3]．

前述のように，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ベースに ECU ごとの実装を実現する必要がある．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．

この問題を解決するためには，処理のコアへのマッピングとランタイムの生成を実現するツールが必要となる．処理のコアへのマッピングを自動化するツール [4] とマッピングの評価を行う性能見積りツールを用意してマッピングの決定後，ラインタイムを人手で生成する方法もあるが，複雑なコアやバスの衝突を考慮して精度の高い性能見積りを実現することは困難である．そのため，ある程度の数のマッピングの候補を実際に実装して評価する必要があるため，マッピングに従ってランタイムを生成するツールが必要となる．また，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も存在するが [5]，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．

本研究では，マルチコアに対応したパワトレアプリを実現するための第 1 段階として，ランタイム生成ツールを実現する．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を記述し，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比較，モデルの規模に対するスケーラビリティを評価する．

2 章では，既存のパワトレアプリの構成とマイコンについて説明する．3 章では，パワトレアプリのマルチコア化の要件について説明する．4 章では，PMPF のコンセプトについて説明する．続く 5 章で PMPF のランタイム生成について説明する．6 章では，実際のパワトレアプリの一部を用いた PMPF の評価について述べる．7 章では，関連研究との関係を説明する．8 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 の比較を行う．

2. パワートレインアプリケーション

本章では，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，パワトレアプリを実行するマイコンの特徴について説明する．

2.1 パワトレアプリのソフトウェア構成

現状のパワトレアプリはシングルコアで動作しており，RTOS を用いたマルチタスク構成となっている．

2.1.1 処理

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，機能ごとにモジュール化されている（機能モジュール）．各機能モジュールは複数の処理により構成されている．各処理は，時間もしくはエンジンのクランク角とい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する必要がある．

各処理には重要度が設定されており，重要度の低い処理は，設計時の周期より長い周期で実行することが許容されている．周期を長くすると制御の品質が下がるが，CPU 負荷を下げることができる．ECU 開発時に CPU 負荷が高く重要な処理がデットラインを満たさない場合は，重要性の低い処理の周期を長くすることで CPU 負荷を下げる．同様に負荷を分散させるため，オフセットの変更も行う．

各機能モジュールごとに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が用意されている．この関数は後述するランタイムから直接呼び出されるため公開関数と呼ぶ．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とも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
  実行が開始されると関数が終了するまで中断されない．

これらの変更を容易に実現するため，ランタイムは，**図 1** に示す構成となっている．まず，起動契機となるイベント

© 2017 Information Processing Society of Japan

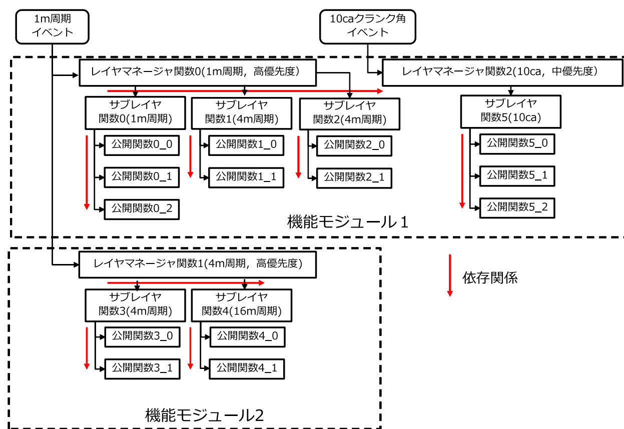


図 1　現状のパワトレアプリのランタイム構成

Fig. 1　Current runtime structure of a powertrain application.

と優先度が同じ公開関数をグループ化する．グループごとにタスクを用意してそのグループの優先度を設定する．タスクは実行されると，グループ内の公開関数を呼び出す．タスクから公開関数を呼び出すためのランタイムとして，レイヤマネージャ関数とサブレイヤ関数と呼ばれる関数を用意している．図 1 に示すように，レイヤマネージャ関数は，所属する機能とイベントに対する周期が同じ公開関数ごとに用意されている．レイヤマネージャ関数は呼び出されると，周期とオフセットを計算し，その起動タイミングで実行する必要があるサブレイヤ関数を実行する．公開関数の周期やオフセットを変更する場合には，サブレイヤ関数を変更すればよい．

ソフトウェア規模は公開関数が 1,000 個，レイヤマネージャが 100 個，タスクが 50 個程度である．

2.2　依存関係

公開関数には，実行順序に依存関係が存在する場合がある．現状のパワトレアプリは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，レイヤマネージャ関数およびサブレイヤ関数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．

2.3　パワトレ用マイコン

現状，パワトレアプリを実行するマイコンはシングルコアが主流である．ノイズ，実装面積，コスト等の理由からコアと ROM（FLASH），RAM を 1 チップにしなければならない．そのため，FLASH サイズは最大でも 4 MB，RAM サイズは 500 KB 程度である [6]．クロックの上限は，ノイズ耐性や動作温度の制約により 300 MHz 程度といわれている．

3.　パワトレアプリのマルチコア化

今後のパワトレアプリの処理量の増加に対応するには，パワトレ用マイコンのマルチコア化が必要である．現状の製品のコア数は 2，3 コアであるが [2], [3]，将来的には 8 コア程度になると予想されている．

本章では，パワトレアプリのマルチコア化に対する要件を説明し，その要件に満たす実現手法を検討する．

3.1　マルチコア化の要件

3.1.1　複数アーキテクチャのサポート

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 つの要件があげられる．

(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

(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．(2) も同様にマルチコアマイコンごとにメモリ構成が異なるために必要となる．

3.1.2　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（レイヤマネージャ関数・サブレイヤ関数は変更可能）

(4) 現状の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 [7] を利用すること

(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．(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．(5) は，AUTOSAR 仕様は車載ソフトのディファクトスタンダードであり，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．

3.1.3　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，以下の要件をシングルコアのときと同様に定める．

(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．

(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 RAM の使用量を抑えることが可能となるためである．(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．

© 2017 Information Processing Society of Japan

3.2 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) 公開関数ごとのタスク化手法

公開関数ごと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．処理へのコアの割当ての変更は，タスクのコアへの割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よい．

(b) 公開関数の選択実行手法

同一のレイヤマネージャ関数を全コアで実行し，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

レイヤマネージャ関数に相当するコードを，公開関数のコア割当てに応じて生成する方法である．

公開関数の数は 1,000 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 (7) と要件 (6) 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，本研究では (c) を採用する．

4. 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により，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．

4.1 設計フローとコンセプト

PMPF による設計フローを図 2 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，公開関数等と共有データのハードウェアへの割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．

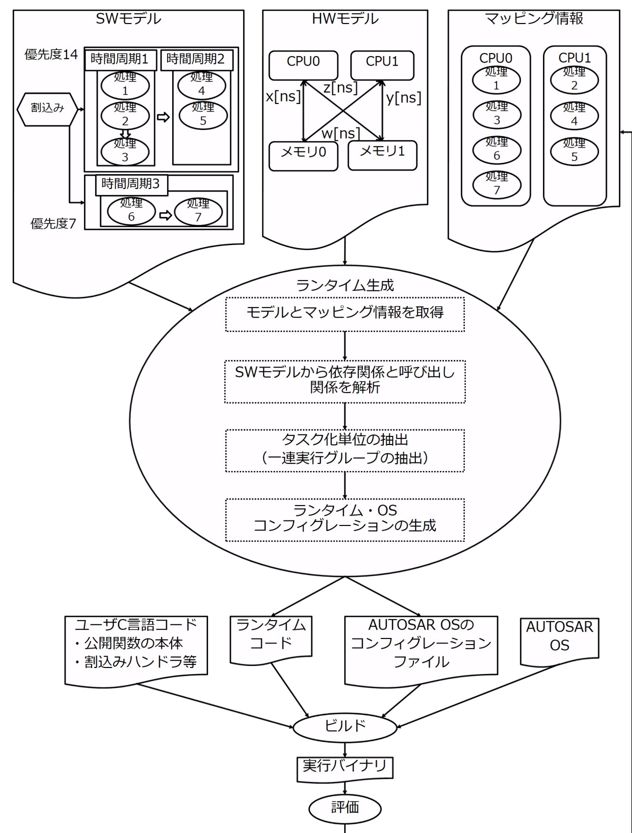


図 2 PMPF による設計フロー
Fig. 2 Design flow using PMPF.

前章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．

- 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
  現状の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，レイヤマネージャ関数，サブレイヤ関数，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 2 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．ランタイムではレイヤマネー

© 2017 Information Processing Society of Japan

ジャ関数・サブレイヤ関数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．

既存の設計では，設計者はターゲットとするマルチコアマイコンごとに公開関数のコアへのマッピングを決めた後，公開関数を依存関係を満たすようにタスクを用意して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．現状コア割当て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当てや，実行周期を変更する頻度が高く，そのつどランタイムの変更が必要となる．前述のように公開関数の数は 1,000 個程度あるため，ランタイムの開発や変更には大きな工数が必要となる．

一方，提案フレームワークは，SW モデルに対して公開関数のコアへの割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 等を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．たとえば，レイヤマネージャ関数やサブレイヤ関数の構成を記述したり，タスク化単位の抽出を自動的に行ったりするツールは存在しない．

4.2　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** に示す．各モデルの詳細については，4.2.1 項で説明する．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 4** に示す．コアごとに配置する公開関数をリストとして指定する．

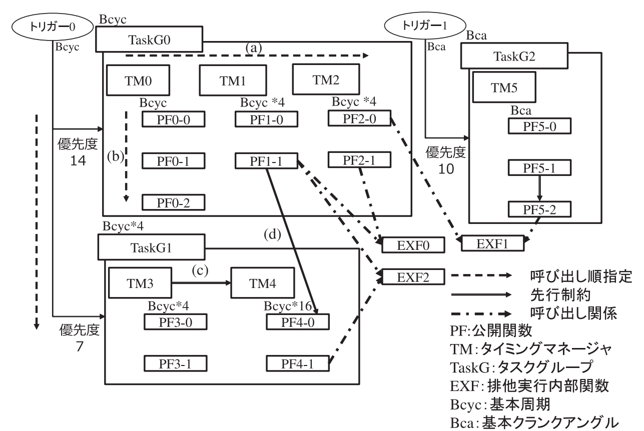
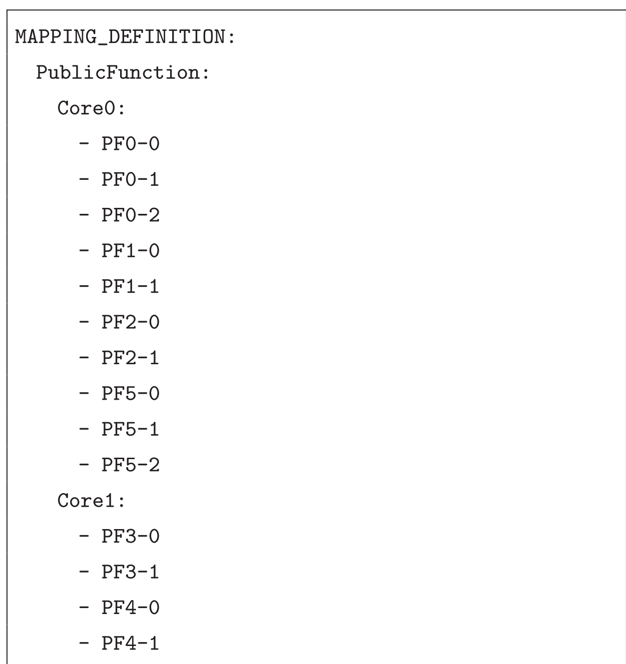


図 3　PMPF の SW モデル
Fig. 3　SW model for PMPF.



```
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 4　マッピング情報
Fig. 4　Mapping information.

4.2.1　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
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レイヤマネージャ関数に相当するオブジェクト．

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

© 2017 Information Processing Society of Japan

```
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 5 図 3 の依存関係の記述（一部）

Fig. 5 A part of dependency description of the SW model in Fig. 3.

状のパワトレアプリにおけるサブレイヤ関数に相当するオブジェクト．

公開関数 処理の呼び出し口となるオブジェクト．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．現状のパワトレアプリにおける公開関数と同等である．

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．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．C 言語記述においては，“排他実行内部関数” とした内部関数はその関数名の頭に “_” を加えるよう変更する．

4.2.2 依存関係の記述

既存のパワトレアプリには処理の依存関係が存在し，レイヤマネージャ関数によってサブレイヤ関数の実行順序が決められ，サブレイヤ関数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レイヤ関数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どうし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かか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 5** は，図 3 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 の依存関係 (a)〜(d) は図 5 の記述 (a)〜(d) に対応している．

5. ランタイム生成

PMPF のランタイム生成のフローを図 2 中に示す．

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およ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．

5.1 タスク化単位の抽出

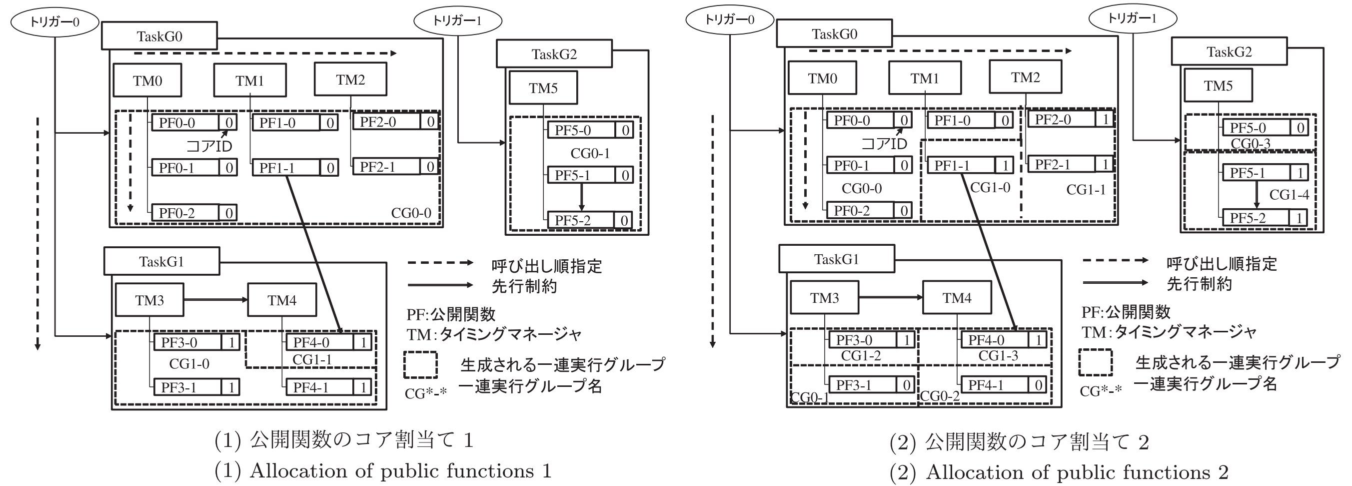
5.1.1 公開関数のタスクへのマッピングにおける課題

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する．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．たとえば，**図 6** (1) の例では，タスクグループ 1 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，この方法ではタスクグループ 1 は 1 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．

5.1.2 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

© 2017 Information Processing Society of Japan



(1) 公開関数のコア割当て 1
(1) Allocation of public functions 1

(2) 公開関数のコア割当て 2
(2) Allocation of public functions 2

図 6　公開関数割当ておよび一連実行グループ抽出
Fig. 6　Allocation of public functions and extraction of chain groups.

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 CH とすると．一連実行グループ CH 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 CH 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．

$$\forall f_i, \forall f_j \in CH, C(f_i) = C(f_j) \wedge TG(f_i) = TG(f_j) \quad (1)$$

ここで f_i は i 番目の公開関数，$C(f_i)$ は公開関数 f_i が割り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 CH 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 CH 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．

$$\forall f_i, \forall f_j \in CH, ED(f_i) = ED(f_j) \quad (2)$$

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．

$$ED(f_i) = \{f \mid path(f, f_i) \wedge f \in (F - CH(f_i))\} \quad (3)$$

ここで $path(f, f_i)$ は f から f_i へのパスが存在すること，$CH(f_i)$ 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 CH_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 CH_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 CH_n に含まれる公開関数は，一連実行グループ CH_n の外部依存公開関数集合または一連実行グループ CH_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 CH_n の外部依存公開関数集合に含まれる公開関数がすべて実行完了すると（依存関係が満たされると），一連実行グループ CH_n の公開関数は，一連実行グループ CH_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．

5.1.3　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 7** は図 3 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

© 2017 Information Processing Society of Japan

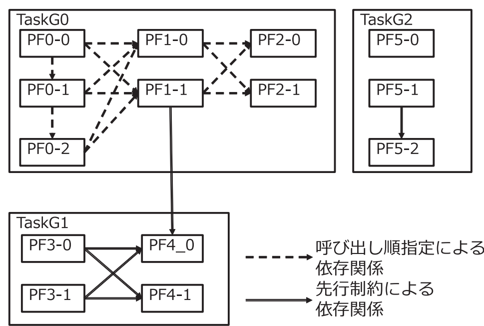


図 7 図 3 の SW モデルから作成したグラフ
Fig. 7 Graph made from the SW model in Fig. 3.

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 CHG=∅ 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 CH 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 CH 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 $CH_0, CH_1, ...$ 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．
$CH_0, CH_1, ...$ に対して，グラフ D において CH_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 CH_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 CH_i に含まれている

(4) CHG に $CH_0, CH_1, ...$ 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 CH_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) を満たしている．

(1-ii)，(2-iii) より，(2) の時点で CH_i に含まれる公開関数は CH_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 CH_i に挿入する公開関数は CH_i に含まれる公開関数のみから接続されているため，CH_i の外の公開関数からのパスはすべて (2) の時点で CH_i に含まれている公開関数を経由する．これにより，CH_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) を満たしている．

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 6 (1)）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 7 で示される．まず (1) では，$PF3\text{-}0$，$FP3\text{-}1$ が (1-i) を満たすため，$CH = \{PF3\text{-}0, PF3\text{-}1\}$ となる．次に (2) では，$PF3\text{-}0$，$PF3\text{-}1$ はともに (2-i)，(2-ii) を満たしているので $CH_0 = \{PF3\text{-}0, PF3\text{-}1\}$ となる．次に (3) では，$PF3\text{-}0$，$PF3\text{-}1$ から接続されている公開関数は $PF4\text{-}0$，$PF4\text{-}1$ だが，$PF4\text{-}0$ については (3-ii) を満たさないので $PF4\text{-}1$ のみを CH_0 に追加する．これ以上 CH_0 に公開関数を挿入できないので (4) に移る．(4) では CHG に $CH_0 = \{PF3\text{-}0, PF3\text{-}1, PF4\text{-}1\}$ を追加する．

再び (1) に戻ると，$PF4\text{-}0$ は (1-ii) を満たすため $CH = \{PF4\text{-}0\}$ となる．CH の要素はただ 1 つなので (2) では $CH_0 = \{PF4\text{-}0\}$ となる．(3) では $PF4 - 0$ 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 CHG に $CH_0 = \{PF4\text{-}0\}$ を追加する．

以上により，$CHG = \{\{PF3\text{-}0, PF3\text{-}1, PF4\text{-}1\}, \{PF4\text{-}0\}\}$ となり，$\{PF3\text{-}0, PF3\text{-}1, PF4\text{-}1\}$，$\{PF4\text{-}0\}$ 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 6 (2)）はコア割当て 1（図 6 (1)）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．

5.1.4 排他実行内部関数の実現

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を生成する．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させるため，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．

© 2017 Information Processing Society of Japan

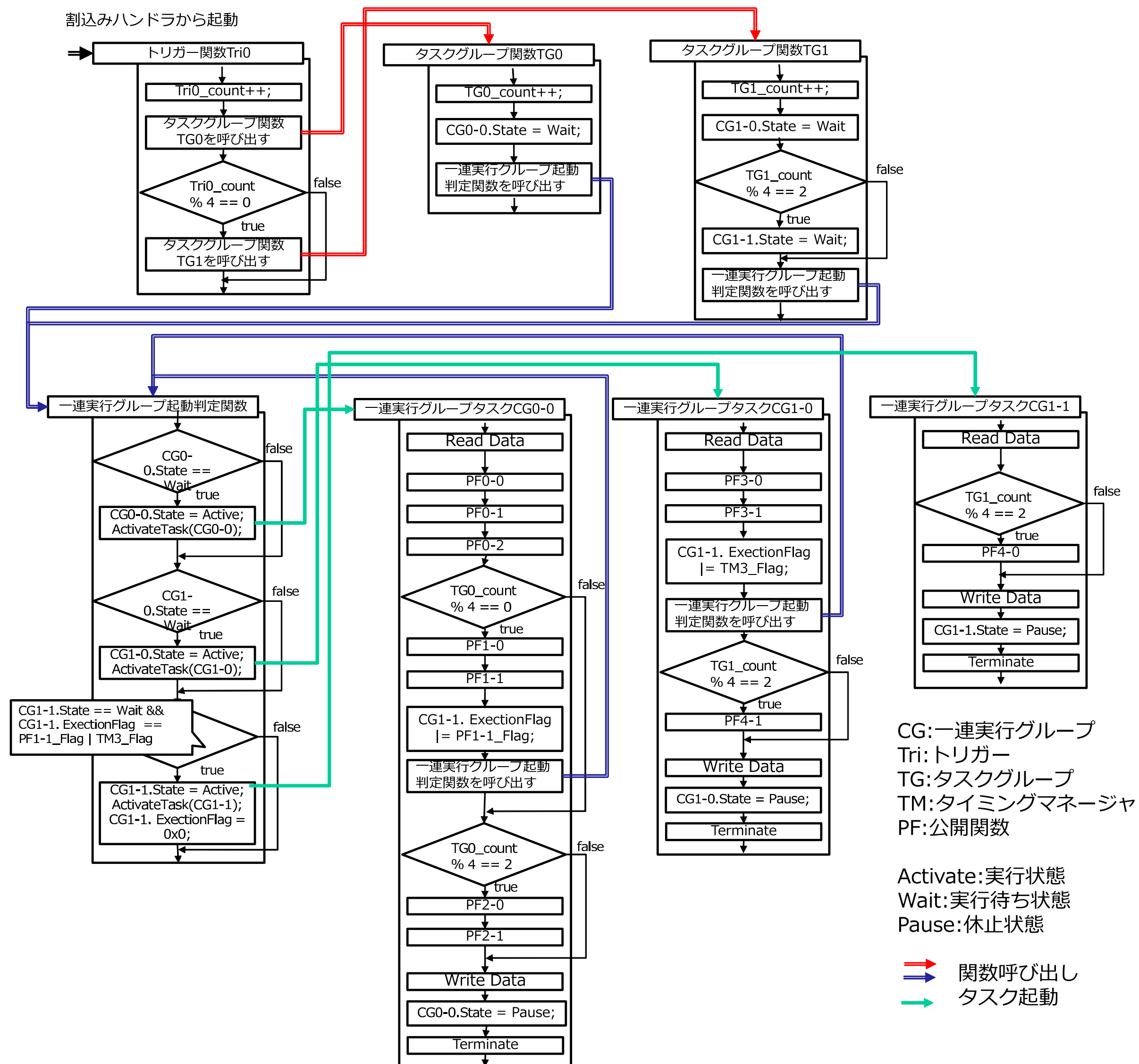


図 8　コア割当て 1 から生成されるランタイムのフロー
Fig. 8　Runtime flow generated from allocation of public function 1.

表 1　配置パターン，排他制御ごとの排他制御メカニズム
Table 1　Exclusion mechanism by combinations of task allocation patterns and types of exclusion.

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

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 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．

5.2　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 6 (1)）から生成されるランタイムのフローを**図 8** に示す．図 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 0 に関係するオブジェクトについてのみを表している．各構成要素については 5.2.1 項で述べる．

© 2017 Information Processing Society of Japan

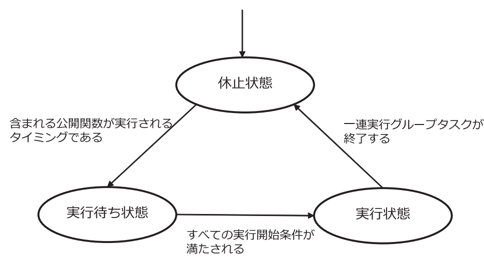


図 9 一連実行グループの状態遷移図
Fig. 9 Chain group state transition diagram.

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 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．

5.2.1 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．

- トリガ関数
  トリガごとにタスクグループ関数を呼び出すトリガ関数が生成される．各トリガごとにカウンタ変数を持ち，トリガ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
  タスクグループごと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起動待ち状態にする．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．

5.2.2 依存関係の実現例

図 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 6 (1)）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 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．

6. 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

© 2017 Information Processing Society of Japan

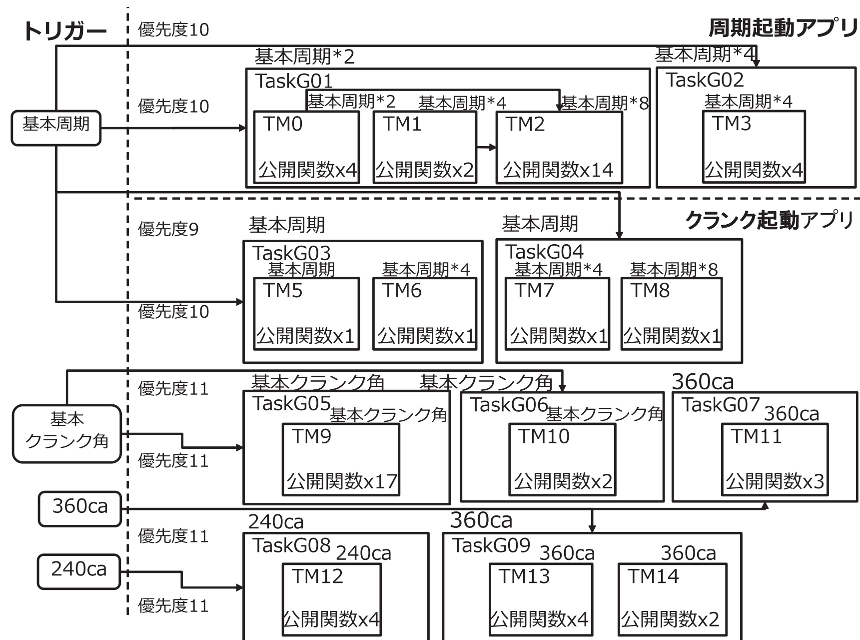


図 10　（評価 2）のモデル

Fig. 10　Model for evaluation 2.

- （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，公開関数ごと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 60 MHz 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 [8] を用いた．

6.1　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 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 10**，**表 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クランク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からタスクグループが起動

表 2　評価用モデル

Table 2　Models for evaluation.

評価	コア	オブジェクト [個数]			モデル [行数]		
		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

表 3　（評価 2）使用したモデルの詳細

Table 3　Detail of the model used in evaluation 2.

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	TaskG01	基本周期*2	4	9	15
		基本周期*4	2		
		基本周期*8	14		
	TaskG02	基本周期*4	4	3	3
	TaskG03	基本周期	1	1	1
		基本周期*4	1		
	TaskG04	基本周期*4	1	1	1
		基本周期*8	1		
クランク角同期処理	TaskG05	基本クランク角	17	1	1
	TaskG06	基本クランク角	2	1	1
	TaskG07	360ca	3	1	1
	TaskG08	240ca	4	1	1
	TaskG09	360ca	6	2	2
合計			60	20	26

© 2017 Information Processing Society of Japan

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．

6.2 （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 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 4 （評価 1）記述量評価の評価結果
Table 4 Evaluation result of amount of description.

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

6.3 （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．

クランク角同期の処理に関しては，1,000 rpm, 2,000 rpm, 3,000 rpm 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．またクランク角同期処理も，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 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 3,000 rpm の場合以外で 2，4 コアの両方において実行時間が削減できることを確認した．回転数が 3,000 rpm 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，クランク角割込みの間隔が短くなり，いくつかのクランク角期タスクが次のクランク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．

6.4 （評価 3）スケーラビリティ評価

表 2 に示した，評価 3 用のモデルを合成した結果を**表 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 1,000 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，4,000 個の場合であっても 2 時間半程度

表 5 （評価 2）一連実行グループ抽出評価の評価結果
Table 5 Evaluation result of extraction of chain group.

コア数	タスク化単位	タスク数	クランクの回転数 [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	1,000	134	67	-	0
			2,000	203	89	-	0
			3,000	190	90	-	153
	一連実行グループ	20	1,000	28	59	14	0
			2,000	34	75	18	0
			3,000	46	90	0.001	0
4	公開関数	60	1,000	129	65	-	0
			2,000	202	85	-	0
			3,000	186	82	-	163
	一連実行グループ	26	1,000	31	51	26	0
			2,000	43	69	22	0
			3,000	51	85	-4	0

© 2017 Information Processing Society of Japan

表 6 （評価 3）スケーラビリティ評価の評価結果
Table 6 Evaluation result of scalability.

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	
1,000	11	143	158	52,751	3,168	450
2,000	42	1,112	1,167	103,523	6,409	913
3,000	97	4,292	4,415	158,230	10,112	1,442
4,000	184	9,236	9,532	208,389	12,877	1,837

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．

7. 関連研究

文献 [1], [9], [10] では，本研究と同様に，既存のパワトレアプリをもと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等の情報から処理を 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がある．また，コア間の処理の依存関係を満たすような機構は持たない．

文献 [14] ではつね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でき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できないため，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 [11], [12], [13] 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当てが必要となる．

8. 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．

8.1 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．

8.2 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 5.1 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．したがって，PMPF の方が設計効率において優れていると考えられる．

8.3 コア割当てに関する比較

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またいで配置する必要があるが，AUTOSAR RTE では実現することができない．

8.4 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，スピンロック等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．

8.5 AUTOSAR RTE と提案手法の組合せ

前述のように，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．

9. おわりに

本研究では，パワトレアプリのマルチコア化を実現する

© 2017 Information Processing Society of Japan

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．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．

今後の課題としては，公開関数のコアへのマッピングやデータのメモリへのマッピングを自動化するツールを実現することがあげられる．

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の方々に深く御礼申し上げます．

参考文献

[1] Michel, L., Flaemig, T., Claraz, D. and Mader, R.: Shared SW development in multi-core automotive context, *ERTS2-2016*, Toulouse (Jan. 2016).
[2] NXP Semiconductors, MPC5777M.
[3] Infineon Technologies, AURIX TC27xT.
[4] Wang, W., Cotard, S., Gravez, F., Chambrin, Y. and Miramond, B.: Optimizing Application Distribution on Multi-Core Systems within AUTOSAR, *ERTS2-2016*, Toulouse (Jan. 2016).
[5] Umeda, D., Kanehagi, Y., Mikami, H., Hayashi, A., Kimura, K. and Kasahara, H.: Automatic Parallelization of Hand Written Automotive Engine Control Codes Using OSCAR Compiler, *17th Workshop on Compilers for Parallel Computing* (*CPC2013*), Lyon, France (July 2013).
[6] ルネサスエレクトロニクス RH850E1x (online), 入手先 ⟨http://japan.renesas.com/products/mpumcu/rh850/rh850e1x/index.jsp⟩ (参照 2016-01-28).
[7] AUTOSAR Consortium. Specification of multi-core OS architecture v1.0. AUTOSAR Release 4.0 (2009).
[8] TOPPERSプロジェクト TOPPERS/ATK2 (online), 入手先 ⟨http://www.toppers.jp/atk2.html⟩ (参照 2016-01-28).
[9] Claraz, D., Grimal, F., Leydier, T., Mader, R. and Wirrer, G.: Introducing Multi-Core at AutomotiveEngine Systems, *ERTS2-2014*, Toulouse (Feb. 2014).
[10] Mader, R., Armin, G. and Gerd, W.: AUTOSAR Based Multicore Software Implementation for Powertrain Applications, *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, 8.2, pp.264–269 (2015).
[11] Fuhrman, T., Wang, S., Jersak, M. and Richter, K.: On Designing Software Architectures for Next-Generation Multi-Core ECUs, *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, Vol.8, No.1, pp.115–123 (2015).
[12] 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 (2010).
[13] Monot, A. et al.: Multicore scheduling in automotive ECUs, *Embedded Real Time Software and Systems–ERTSS 2010* (2010).
[14] Panić, M., Kehr, S. et al.: RunPar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores, *Proc. 14th IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign and System Synthesis* (*CODES+ISSS*), ACM Press, New York, USA (2014).

小川 真彩高 (学生会員)

2016年名古屋大学大学院情報科学研究科博士前期課程修了．現在，同大学院情報科学研究科博士後期課程在学中．車載組込みシステムの研究に従事．

本田 晋也 (正会員)

2002年豊橋技術科学大学大学院情報工学専攻修士課程修了．2005年同大学院電子・情報工学専攻博士課程修了．名古屋大学大学院情報科学研究科附属組込みシステム研究センター助教等を経て，2014年より名古屋大学大学院情報科学研究科情報システム学専攻准教授，リアルタイムOS，ソフトウェア・ハードウェアコデザインの研究に従事．博士（工学）．2002年度情報処理学会論文賞受賞．ACM，IEEE，電子情報通信学会，日本ソフトウェア科学会各会員．

高田 広章 (正会員)

名古屋大学大学院情報科学研究科情報システム学専攻教授．1988年東京大学大学院理学系研究科情報科学専攻修士課程修了．同専攻助手，豊橋技術科学大学情報工学系助教授等を経て，2003年より現職．2006年より大学院情報科学研究科附属組込みシステム研究センター長を兼務．リアルタイムOS，リアルタイムスケジューリング理論，組込みシステム開発技術等の研究に従事．オープンソースのITRON仕様OS等を開発するTOPPERSプロジェクトを主宰．博士（理学）．IEEE，ACM，電子情報通信学会，日本ソフトウェア科学会，自動車技術会各会員．

© 2017 Information Processing Society of Japan