Vol.2019-ARC-240 No.47
Vol.2019-SLDM-191 No.47
Vol.2019-EMB-53 No.47
2019/2/28

アプリケーションプロセッサを用いたミックスドクリティカルシステム向けTDMAベースのスケジューリング手法の提案

三浦功也[1]　本田晋也[2]

概要:
組込みシステムの大規模化，複雑化，コスト削減要求に対応するために，従来使用されてきた組込み向けプロセッサ（ MPU）ではなく，高性能なアプリケーションプロセッサ（ APU）を活用することが検討されている．APU は高価であるため，既存システムのコストを増加させないためには，これまで複数の MPU で実現されていた個別のアプリケーション（ アプリ ）をまとめて単一の APU で実現するアプリ統合が必要となる．各アプリは要求する安全度水準やリアルタイム性が異なるため，アプリ統合後のシステムはミックスドクリティカルシステムとなる．アプリ間の Freedom from Interference（ FFI）を実現するためには，従来の組込みシステムで用いられてきた優先度ベースのスケジューリングではなく，TDMA スケジューリングが必要となる．しかしながら，APU は最悪実行時間と平均実行時間との差が大きいため，各アプリに最悪実行時間で CPU 時間を割付けて TDMA スケジューリングを行うと，平均 CPU 利用率が悪くなるという問題がある．そこで本研究では，各アプリの進捗を管理し，進捗に応じて割付け時間変更することにより問題を解決する TDMA ベースのスケジューリング手法を提案する．

キーワード： スケジューリング, TDMA, ミックスドクリティカル, 性能向上

1. はじめに

組込みシステムは，年々その開発規模が増大していること，また，量産製品であることからコスト削減が求められている．同時に，AI 関連の技術が進歩するに伴って，これらの機能を搭載し，ユーザに対する訴求機能として用いたいという新たな要求も出てきている．

組込みシステム開発では，既存の開発資産を流用して開発をすすめることが一般的であるため，AI 関連の機能についても，同様に既存システムへ追加実装されることが予想される．AI 関連の機能は，従来のシステムが担っていた機能（ モータ制御機能や省エネ制御機能など）と比較して，多くの演算性能を必要とするが，既存の組込みシステム向けプロセッサ（ MPU）は，性能よりもリアルタイム性を重視して設計されている．そのため，これらの制御を実行するには，MPU では性能不足であり，MPU の変更によって性能差を補うという解決策を取ることも困難である．よって，AI 関連の機能のみを専用に実行する MPU を用意して機能実現をする手法が取られるが，コスト制約に厳しい組込みシステムでは，容易に MPU の数を増やすことは望ましくない．

また，処理負荷の高い AI 関連の機能を既存システムに追加することから，システムの複雑化，大規模化が今まで以上に加速することが想定される．そのため，大規模な業務システムのように，複数部署や複数企業に跨ってアプリケーション（ アプリ ）を実現することが想定される．大規模なアプリを開発する場合，アプリが提供すべき機能を複数に分割し，複数人で個別に開発した機能を統合して，全体のアプリを作成する開発手法が取られる．これを容易に実現するためには，特定の機能の設計変更時に，それが他の機能に影響を与えないようにする必要がある．

これらの問題を解決するために，スマートフォンなどに搭載されているアプリケーションプロセッサ（ APU）上に，処理負荷の小さい従来のシステムが提供する機能と，処理負荷の大きい AI 関連の機能をまとめた単一のアプリを実装し，それぞれの機能を時間的および空間的に分割して実行することで，プロセッサ性能への要求を満たすとともに，他の機能の動作の影響を受けないようにする手法がある．

[1] 三菱電機（ 株）情報技術総合研究所
[2] 名古屋大学 大学院情報学研究科

Vol.2019-ARC-240 No.47
Vol.2019-SLDM-191 No.47
Vol.2019-EMB-53 No.47
2019/2/28

APU は高性能であるが，MPU に比べて高価である．よって，既存システムのコストを増加させないためには，これまで複数の MPU を組み合わせて実現されていたアプリを単一の APU 上で実現するアプリ統合を行い，MPU の数を削減することでコスト削減を行う必要がある．

統合前の各アプリは要求する安全度水準やリアルタイム性が異なることから，アプリ統合後のシステムはミックスドクリティカルシステムとなる．ミックスドクリティカルシステムを APU 上に統合するには，アプリ間の Freedom from Interference(FFI) を実現する必要があり，従来の組込みシステムで用いられてきたスケジューリング方法である優先度ベースのスケジューリングではなく，TDMA スケジューリングが必要となる．

しかし，APU には，最悪実行時間と平均実行時間が大きく乖離するという特性があるため，各アプリの最悪実行時間を基にして CPU 時間を割付けると，平均的な状況では CPU 時間が余ってしまい，結果的に CPU 利用率が悪くなるという問題や，性能の高い APU が必要となりコストが増加してしまうという問題がある．

本研究では，この問題を解決する手法として，統合されたそれぞれの処理の進捗を確認し，割付ける CPU 時間を動的に変更することによって，システムが最低限提供すべき安全性を担保しつつ，CPU の利用効率を上げることができるフレームワークおよび，スケジューリング手法を提案する．

本書の構成を以下に示す．2 章では，アプリ統合の概要と，アプリ統合を実現するために必要となる TDMA スケジューリングを APU に適用する際の課題について述べる．3 章では，関連する先行研究について述べる．4 章では，課題を解決するためのフレームワーク，および，フレームワーク上でのスケジューリング手法を提案する．5 章で，提案するフレームワークの評価を行う．

2. アプリ統合とその課題

本章では，統合の対象となるアプリの特性と，アプリ統合に使用する APU の特性，及び，優先度ベースでスケジューリングを行うことによるアプリ統合の課題について述べる．

2.1 アプリ統合方針

本検討におけるアプリ統合の方針を以下に示す．

(a) 統合前アプリが統合後アプリの 1 機能となるように統合する

アプリ統合の際，統合前アプリを統合後アプリが持つ 1 機能として実装するものとする．すなわち，統合後アプリが提供する機能として，統合前アプリを持つ構造となる．

(b) 統合前アプリの構造を統合後アプリに引き継ぐ

アプリ統合の際，統合前アプリの構造を統合後アプリにそのまま引き継ぐものとする．すなわち，統合後アプリが提供する 1 機能は，統合前アプリと同じソフトウエア構造となる．

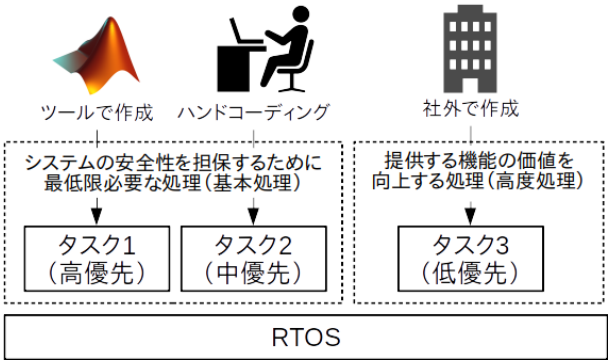

図 1　統合前アプリの概要

2.2 統合前アプリの特徴

本検討における統合前アプリの概要を図 1 に示す．本検討で統合前アプリは以下の特徴を有するものとする．

(a) 複数のタスクで構成されている

特定の機能を提供するアプリは多くの場合，機能を提供するために必要となる特定の処理単位(タスク) が複数存在し，タスク間でデータを授受することで動作する．統合前アプリも同様に，複数のタスクで構成されているものとする．

(b) 基本処理と高度処理で構成される

特定の機能(例えばモータ制御) を提供する場合，その機能を構成する要素には，電流フィードバック制御や電流異常検出制御といった，システムの安全性を担保するために最低限必要となる処理(基本処理) と，省エネ制御や騒音低減制御といった，システムが提供する価値を向上するための処理(高度処理) が存在する．統合前アプリも同様に，基本処理と高度処理で実現されているものとする．

(c) 優先度ベースでスケジューリングが行われている

大規模なアプリを MPU 上で実現する場合，ITRON や VxWorks などのリアルタイム OS(RTOS) を用いて機能実現することが一般的である．例えば，自動車業界で広く用いられている AUTOSAR は，AUTOSAR Classic Platforom 仕様の AUTOSAR OS として，RTOS が用いられている．よって，統合前アプリも同様に，RTOS 上で動作するアプリとして実装され，優先度ベースでスケジューリングが行われているものとする．

(d) バックグラウンドタスクが存在する

RTOS 上で優先度ベースのスケジューリングを行う場合，最低優先度のタスクは他のタスクが存在しない場合にのみ実行される．本検討では，システムの安全性を保証するために，基本処理を高優先度のタスクとして，高度処理を低優先度のタスクとして実行しているものとする．

Vol.2019-ARC-240 No.47
Vol.2019-SLDM-191 No.47
Vol.2019-EMB-53 No.47
2019/2/28

(e) 基本処理と高度処理の安全度水準が異なる

基本処理はシステムが提供すべき安全性を担保するために最低限必要となる処理である一方で，高度処理はシステムが提供すべき安全性には影響しない．そのため，例えば，基本処理は ISO26262 で規定されている安全度水準である ASIL D 相当の処理として実装し，高度処理は同水準の ASIL B 相当の処理として実装する，というように，これら 2 つの処理は異なる安全度水準を持つ可能性が高い．よって，統合前アプリの基本処理は高度処理よりも安全度水準が高く，潜在的なバグは存在しないものとする．

(f) 複数の開発環境を用いて実現されている

組込みシステムの大規模化に伴い，MPU に実装されるアプリも複数人で開発することが増えてきている．このとき，ソースコードを作成する方法として，(1) MATLAB/Simulink などのツールを用いて作成する (2) 自社でコーディングする (3) 外注により作成するなど，様々な手法で実現することが考えられる．よって，統合前アプリも同様に，複数の異なる開発環境で実現されるものとする．

2.3 アプリケーションプロセッサ（APU）

APU とは，スマートフォンなどに用いられている高性能なプロセッサであり，分岐予測，パイプライン，キャッシュ，マルチコアなどの高速化技術を導入することで，従来の MPU と比較して，高い実行性能を実現している．

表 1 に，同一ベンダにおける，MPU と APU のスペック，および，Coremark の値を示す[*1]．Coremark はソートや行列計算，CRC の計算などを実行するベンチマークであり，プロセッサの演算性能の計測を行うために用いられる．表 1 からわかるように，APU の価格が MPU と比較して 3 倍〜4 倍程度であるのに対し，APU の性能は MPU の 10 倍以上となっており，単一の APU 上に，複数の MPU に搭載されたアプリを統合することで，コスト削減に寄与できることがわかる．

2.4 優先度ベースのスケジューリングによる統合の問題

2.2 節で述べたように，統合前アプリは優先度ベースのスケジューリングで実行されているが，統合後に優先度ベースのスケジューリングを用いると，**図 2** に示す以下の問題が発生する．

(a) 高優先度タスクの不具合が波及する

優先度ベースのスケジューリングでは，統合後アプリに実装された高優先度タスクに不具合がある場合に，その不具合が統合前の他アプリに波及するという問題がある．図 2 の (a) は，統合前アプリの高優先度タスクに存在していた潜在的な不具合（無限ループ）が，アプリ統合により顕在化した場合の例を示す．優先度ベースのスケジューリングでは，高優先度タスクの実行中は，それよりも優先度の低いタスクは動作できないため，不具合が発生したタスクよりも優先度の低いすべてのタスクが実行されない．よって，統合前アプリ 1 に実装されていたタスクの不具合が，統合後ではアプリ 2 に波及してしまい，統合前アプリが提供していたアプリ 2 の機能を統合後アプリでは提供できなくなってしまう．複数の企業や部署に跨って 1 つのアプリを実装することから，これをアプリ全体で調整して不具合の波及を阻止し，安全性を担保することは難しい．よって，アプリ間での不具合の波及を生じさせない実行環境が必要である．

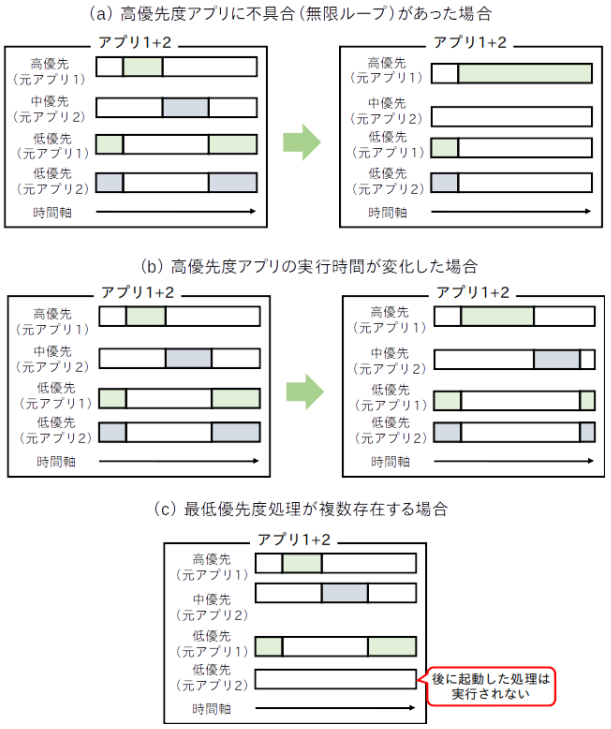

図 2 優先度ベースのスケジューリングによる統合の問題

(b) 高優先度タスクの動作変更が低優先度タスクに波及する

優先度ベースのスケジューリングでは，高優先度タスクが完了するまでは，それよりも優先度の低いタスクを実行しない．図 2 の (b) は，キャッシュミス，設計変更などの理由で，アプリ統合後に高優先度タスクの実行時間が変動した場合の例を示す．実行時間が変動したタスクよりも優先度の低いタスクの開始時間が変動するため，統合前アプリ 1 に実装されていたタスクの動作変更が，統合後ではアプリ 2 に波及してしまう．従来の開発では，タスクの動作変更時には，同一環境で動作する他のタスク全体を見直し，再設計を行うことで対応していたが，複数の企業に跨って開発をする場合には，同一環境で動作する他アプリの細かな動作仕様を知ることができないために，タスクの再設計が困難になってしまう．そのため，それぞれのアプリを時間的に独立させ，他アプリに実装されたタスクの実行時間

[*1] https://www.eembc.org/coremark/view.php?benchmark_seq=2853,1053

Vol.2019-ARC-240 No.47
Vol.2019-SLDM-191 No.47
Vol.2019-EMB-53 No.47
2019/2/28

表 1　MPU と APU の比較

CPU	コア名	コア数	周波数	Coremark 値	参考価格
AM5728	ARM Cortex-A15	2	1500MHz	15788.22	40〜50 ドル
LM3S9B96	ARM Cortex-M3	1	80	127.59	13 ドル

の変動に影響されることのない，アプリの実行環境を提供することが必要である．

(c) 最低優先度タスクがいずれか一つしか実行されない

優先度ベースのスケジューリングでは，同一優先度のタスクは起動時にキューに蓄えられ，キューの先頭にあるタスクが完了すると次のキューのタスクを実行する．大規模かつ複数の組織に跨って分散開発を行ったり，既存機能を統合する場合，統合前アプリのタスクに割付けられていた優先度をそのまま用いると，統合後アプリではタスク間で優先度が競合し，本来実行されるべき機能を実行できなくなるという問題がある．図 2 の (c) は，統合によって最低優先度のタスクが競合した場合の例を示す．アプリ 1 の低優先度タスクがアプリ 2 の低優先度タスクよりも先に起動すると，アプリ 2 の低優先度タスクはアプリ 1 の低優先度タスクが完了するまで実行することができないために，統合前アプリが提供していたアプリ 2 の機能を統合後アプリでは提供できなくなってしまう．そのため，それぞれのアプリを空間的に独立させ，アプリ間でタスクに割当てられた優先度が競合することのないタスクの実行環境を提供することが必要である．

2.5　TDMA スケジューリング

前述の優先度ベースのスケジューリングの問題を解決するために，TDMA スケジューリングと呼ばれるスケジューリング手法の活用が検討されている [1]．TDMA スケジューリングは，時分割，空間分割を提供できるスケジューリング方式である．TDMA スケジューリングには，システム周期，および，タイムウインドウ時間と呼ばれる時間単位が存在する．システムで最短の処理周期をシステム周期と呼び，システム周期をいくつかの時間単位に分割したものをタイムウインドウ時間と呼ぶ．アプリには特定のタイムウインドウ時間が割当てられ，システム周期の時間のうち，タイムウインドウ時間として割当てられた時間だけプロセッサを利用できる．タイムウインドウ時間が割当てられたそれぞれのアプリは時間的，空間的に独立しており，また，タイムウインドウ時間として割当てられた時間はプロセッサを利用できることが保証されることから，特定のアプリの動作変更がその他のアプリの動作に影響を与えることはない．

図 3 に，システム周期を 100us，タイムウインドウ時間をそれぞれ 40us(アプリ 1 への割当て時間)，60us(アプリ 2 への割当て時間) とした場合の例を示す．この場合，まず，アプリ 1 を 40us 実行した後に，アプリ 2 を 60us 実行する．その後，次のシステム周期にて再びアプリ 1 を 40us 実行する，という流れで処理をすすめる．

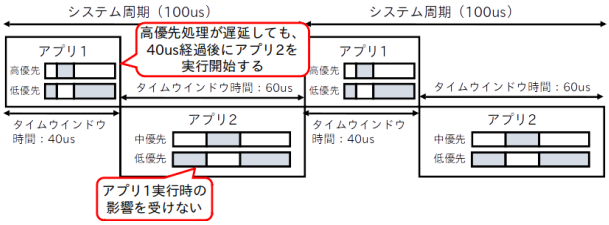

図 3　TDMA スケジューリング

2.6　APU 上で TDMA をする際の課題

APU は MPU と比較して容易に性能を向上できるが，リアルタイム性を重視した MPU と比較して，最悪実行時間の推定がしづらいという問題がある．いくつかの APU を対象として，その実行時間を計測した研究 [2] によると，メモリアクセスを頻発するプログラムを Raspberry Pi3 上で実行した場合，メモリアクセスの頻度により，実行時間に 300 倍以上の開きがあることが確認されている．また，APU の実行性能は，キャッシュミスやコア間の干渉などのプロセッサの影響だけでなく，バス幅やメインメモリへのアクセス速度など，ペリフェラルの性能にも影響を受ける．そのため，同一のプロセッサを用いるデバイスであっても，チップベンダごとに最悪値が異なり，最悪実行時間を推定することを困難にする一要因となっている．

組込みシステムでは，最低限保証すべき安全性を担保するために，プロセッサ上での実際の実行時間が最悪値になった場合にも，その処理の最悪実行時間を超えないことを保証できるように，処理の優先度やその処理内容を設計する必要がある．TDMA スケジューリングを行うには，システム周期およびタイムウインドウ時間を決定する必要があるが，最悪実行時間を保証するために従来同様の設計を APU 上で行う場合，以下の問題が発生する．

プロセッサ利用効率が悪化する

アプリは多くの場合，平均的な実行時間で完了し，最悪実行時間となる場合は稀である．そのため，タイムウインドウ時間を最悪実行時間を基に決定した場合，平均実行時間で処理が完了する多くの場合には，プロセッサに空き時間が生まれ，利用効率の低下を招く．

デッドラインミスをする可能性がある

APU の実行時間は，キャッシュミスやペリフェラル性能の影響を強く受けるため，アセンブラの実行サイクル数をカウントする等の従来の方法を用いても，その真の最悪

実行時間を算出することはできない．よって，実際にアプリを動作させた際に計測できた最悪実行時間を基にしてタイムウインドウ時間を決定することになるが，計測時よりもキャッシュミスが頻発した場合などにはデッドラインミスをする可能性がある．

3. 先行研究

異なる優先度を持つ複数の機能を同一のプロセッサで実行するシステムはミックスドクリティカルシステムと呼ばれ，先行研究がいくつか存在する．Lara らは，バスアクセスに関するミックスドクリティカルシステムを対象としたスケジューリング手法を提案している [3]．この研究では，プロセッサコアのバスアクセスに関して，他コアとバスを共有できるモード（ Shared Mode）とバスを専有するモード（ Isolated Mode）の 2 つのモードを用意している．デッドラインを守れなくなるまでは Shared Mode で実行し，デッドラインを守れなくなることを検出すると，それ以降の処理を Isolated Mode で実行することにより，バスの競合による処理遅延を防ぐことでデッドラインを守るという手法が取られている．Crespo らは，仮想マシンを対象にミックスドクリティカルシステムを実現する際のスケジューリング手法を提案している [4]．この研究では，特定の機能がデッドラインミスする段階であることを検出すると，他のコアの処理を止めることにより，優先すべき処理のデッドラインを保証するという手法を取っている．Cilku らも，優先すべき処理のクリティカルセクション実行中に他のコアの実行を止めることにより，優先すべき処理の実行時間を保証するという手法を提案している [5]．これらの手法はいずれも，ハードウエアに主眼を置いた研究である．また，いずれのスケジューリング手法も，優先すべき処理を保証するために他のコアでの動作を止める必要があり，APU に適用する場合には，プロセッサを効率よく利用できないという問題がある．

ソフトウエアを対象としてミックスドクリティカルシステムを実現した研究には，Ali らの提案手法 [6] がある．この研究では，クリティカルな処理については排他的にすべてのコアを専有して実行させるとともに，アイドル処理はメモリアクセス速度に制限をつけて実行することで，ミックスドクリティカルシステムを実現している．この手法はソフトウエア上で実現しているものの，クリティカルセクション実行中は他の処理を実行できなくなるため，プロセッサを効率よく利用できないという問題がある，また，メモリアクセス速度が制限されていることから，キャッシュミスに伴うメモリアクセスが頻発するような AI 関連の機能に適用すると，最悪実行時間と平均実行時間の乖離が進むおそれがある．

4. 提案フレームワーク

2.6 節で述べた課題に対応するために，本研究では，統合後アプリに統合された，個々の統合前アプリの基本処理の進捗を管理し，統合前アプリの基本処理のタイムウインドウ時間の割当てを動的に変動させるフレームワークを提案する．なお，以降では，説明のために統合後アプリに統合された，個々の統合前アプリのことを，単にアプリと表記する．

提案フレームワークの基本的な振る舞いは次の通りである．各アプリはまず基本処理を実行する．基本処理が終了すると，他のアプリで基本処理が実行されていれば，自身のタイムウインドウ時間の割当てを 0 として，その分，他のアプリのタイムウインドウ時間を増加させる．全てのアプリの基本処理が完了すれば，全てのアプリにタイムウインドウ時間を割当て高度処理を実行する．

このようなスケジューリングを行う事により，プロセッサの利用効率を上げるとともに，基本処理のデッドラインを保証する．

4.1 前提とするプラットホーム

本フレームワークは，TDMA スケジューリングをサポートした ITRON ベースの RTOS である，TOPPERS HRP3 カーネル上で設計及び実装を行う．HRP3 カーネルは，各アプリに割当てられたタイムウインドウ時間を切り替えながら，アプリのタスクを周期的に起動する機能を有する．

HRP3 カーネルでは，各アプリへのタイムウインドウの割当て方法である，システム動作モードを定義できる．システム周期は統合後アプリの設計時に静的に決定する必要があるが，システム動作モードは，コンフィギュレーションファイルと呼ばれるファイルに静的 API を用いて記述することで，静的に複数のパターンを定義できる．記載されたコンフィギュレーションファイルの情報は，コンフィギュレータにより RTOS の構成情報へ変換される．

また，アプリが専用の API を呼び出すことにより，次のシステム周期の切り換えタイミングでシステム動作モードが RTOS により変更され，アプリの動作中にタイムウインドウ時間を動的に切り換えることができる．HRP3 カーネルは，割付けた処理がすべて完了すると，システムのアイドル処理を実行するため，高度処理が完了してもプロセッサがアイドル状態になることはない．

4.2 アプリモデル

アプリケーションは基本処理と高度処理で構成されている．基本処理は最悪実行時間と起動周期を持ち，デットラインは起動周期と同じとする．高度処理は要求する CPU 利用率を持つ．

Vol.2019-ARC-240 No.47
Vol.2019-SLDM-191 No.47
Vol.2019-EMB-53 No.47
2019/2/28

表 2　システムを構成するアプリ例

アプリ名	基本処理		高度処理
	最悪実行時間	周期	CPU 利用率
アプリ 1	2ms	5ms	30%
アプリ 2	5ms	15ms	10%
アプリ 3	10ms	50ms	60%

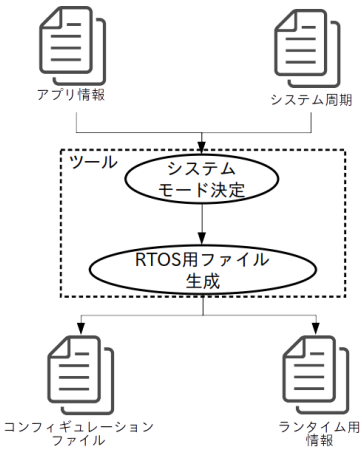

図 4　ツールの動作フロー

以降の節では，表 2 に示す 3 つのアプリで構成されるシステムを対象として説明する．

4.3　フレームワークの構成

提案するフレームワークはツールとランタイムにより構成されている．ツールの動作フローを図 4 に示す．ツールは表 2 の情報を含むアプリ情報と，システム周期を入力とし，コンフィギュレーションファイル，及び，ランタイム用の情報を出力する．

ツールはまず，各アプリの進捗（ 基本処理が完了したか否かを示す情報）に応じたシステム動作モードを定義する．次に，定義したシステム動作モードに対応したコンフィギュレーションファイルと，ランタイムが使用する，アプリの進捗に対応したシステム動作モードの情報（ ランタイム用情報）を C 言語ファイルとして生成する．システム動作モードの詳細は，4.4 節を参照すること．

ランタイムの構成を図 5 に示す．ランタイムは進捗管理機能とシステム動作モード管理機能で構成されている．進捗管理機能は，各アプリケーションの進捗を管理する機能である．各アプリから処理の進捗を API を介して受け取り，アプリの進捗状況を管理する．システム動作モード管理機能は，次のシステム周期のシステム動作モードを決定する機能である．ユーザーアプリケーションとしてシステム周期の最後に実行され，進捗管理機能から受け取った各アプリの進捗と基本処理の周期の情報をもとに，システム動作モードを決定する．

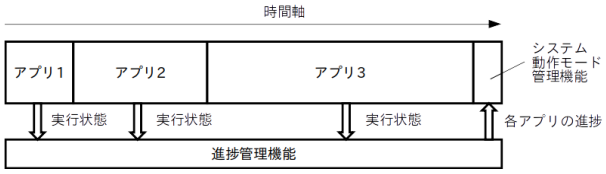

図 5　ランタイムの構成

4.4　システム動作モード定義

本フレームワークでは，基本処理を高度処理よりも優先して実行するために，各アプリの進捗の組合せを網羅できるようにシステム動作モードを定義する．本フレームワークで 3 つのアプリを動作させる場合のシステム動作モードの例を表 3 に示す．

システム動作モードは，基本処理を実行している状態（ B0〜B6）と，高度処理を実行している状態（ A0）に大別される．まず，アプリ全体で最短のデッドライン（ 5ms）をシステム周期として設定する．次に，基本処理を実行している状態でのシステム動作モードを決定する．基本処理を実行している状態では，各アプリは実行するべき基本処理の有無により，タイムウインドウ時間の割当てが異なるため，1 つ以上の基本処理を実行している状態となる，$2^{NumApp} - 1$（ NumApp はアプリ数）のシステム動作モードが必要である．B0 は全てのアプリの基本処理を実行している状態である．B0 では，各アプリには，最悪実行時間で実行した場合に周期内に処理が終了する時間を割当てた後，残った時間を均等に割当てる．表 2 の場合，アプリ 1 には 5ms 周期で 2ms の時間を与えることができるように 2ms の時間を割当て，同様にして，アプリ 2 には 1.7ms を，アプリ 3 には 1ms の時間を割当てる．割当て済みのタイムウインドウ時間の合計時間が 4.7ms であるため，アプリ 1 の実行周期である 5ms までには 0.3ms だけの余裕がある．よって，これをそれぞれのアプリに均等に分配する．B1〜B6 では，B0 の割当て結果を基に，処理が完了したアプリのタイムウインドウ時間を他のアプリに均等に割付ける．例えば B1 の場合，アプリ 1 は基本処理を完了しているので，アプリ 1 のタイムウインドウ時間（ 2.1ms）をアプリ 2 とアプリ 3 に均等に分配する．B6 の場合は，アプリ 1 とアプリ 2 は基本処理を完了しているので，使用できるすべてのタイムウインドウ時間をアプリ 3 に割当てる．

次に，高度処理を実行している状態でのシステム動作モードを決定する．高度処理では，各アプリが要求する CPU 利用率に応じてタイムウインドウを配分するために，システム動作モードを 1 つのみ定義する．システム周期が 5ms なので，この時間と CPU 利用率を基にしてタイムウインドウ時間を決定する．

4.5　ランタイム

アプリから進捗管理機能へ自身の進捗を通知するため

Vol.2019-ARC-240 No.47
Vol.2019-SLDM-191 No.47
Vol.2019-EMB-53 No.47
2019/2/28

表 3　システム動作モード

ID	実行状態			タイムウインドウ時間(ms)		
	アプリ 1	アプリ 2	アプリ 3	アプリ 1	アプリ 2	アプリ 3
B0	基本処理実行中	基本処理実行中	基本処理実行中	2.1	1.8	1.1
B1	基本処理完了	基本処理実行中	基本処理実行中	0	2.85	2.15
B2	基本処理実行中	基本処理完了	基本処理実行中	3	0	2
B3	基本処理実行中	基本処理実行中	基本処理完了	2.65	2.35	0
B4	基本処理実行中	基本処理完了	基本処理完了	5	0	0
B5	基本処理完了	基本処理実行中	基本処理完了	0	5	0
B6	基本処理完了	基本処理完了	基本処理実行中	0	0	5
A0	高度処理実行中	高度処理実行中	高度処理実行中	1.5	0.5	2

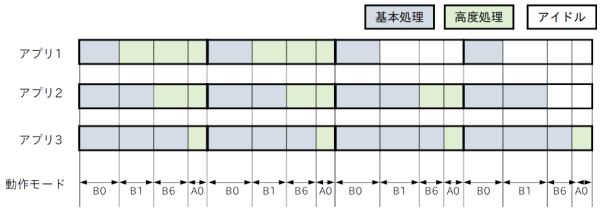

図 6　動作例

に，基本処理終了通知 API を提供する．アプリが基本処理終了通知 API を呼び出すことで，進捗管理機能は，呼び出したアプリの実行状態を基本処理完了状態にする．

システム動作モード管理機能は，進捗管理機能から受け取った各アプリの進捗から，次のシステム周期のシステム動作モードを決定する．なお，基本処理および高度処理の起動自体は周期起動機能によりアプリ自身で行う．

4.6　動作例

本フレームワークを用いて，表 2 に示す 3 つのアプリを動作させる場合の動作例を図 6 に示す．各アプリの実行状態の変化に伴い，表 2 で定義したシステム動作モードを順次切り替えながら，それぞれのアプリを実行する．

5.　評価

本提案の有効性を示すために，2 アプリの場合を例として，提案手法を実際に TOPPERS HRP3 カーネル上に実装し，その評価を行った．本章では，その評価項目及び評価結果について述べる．

5.1　評価環境

本評価を行った環境を表 4 に示す．本評価は，前提とする RTOS である TOPPERS HRP3 カーネル上で行った．ターゲットボードには Xilinx 社の Zybo z7 Zynq-7010 Development Board を用いた．このターゲットボードは，アプリケーションプロセッサである ARM Cortex-A9 MPCore が搭載されている．

5.2　評価項目

本評価における評価項目を以下に示す．

表 4　評価環境(Zybo z7 ZYNQ-7010 Development Board)

大項目	小項目	内容
ハードウエア	プロセッサ	ARM Cortex-MPCore(667MHz)
	L1 キャッシュ	33KByte
	L2 キャッシュ	512KByte
	メモリ	1GByte DDR3
ソフトウエア	RTOS	TOPPERS HRP3 3.2.0

表 5　評価用アプリが想定する基本処理の実行時間

アプリ名	最悪実行時間	デッドライン
アプリ 1	2ms	5ms
アプリ 2	5ms	12ms

ソフトウェア規模

進捗管理機能とシステム動作モード管理機能の実装に必要なソフトウェア規模を評価するため，ステップ数カウンタを用いて，進捗管理機能とシステム動作モード管理機能に相当する処理のステップ数を計測した．

実行オーバヘッド

進捗管理機能とシステム動作モード管理機能の実行時間，および，アプリケーションに提供する API である進捗通知機能の実行時間を計測した．

フレームワーク適用による効果

フレームワークの有無による効果を評価するために，適用前後における基本処理の実行時間を計測した．

5.3　評価用アプリ

評価を行うために使用する基本処理と高度処理に相当するアプリケーションは，一定時間ビジーウェイトするアプリケーションとして作成した．

5.4　評価結果

本節では，フレームワークの評価結果について述べる．

5.4.1　ソフトウェア規模

進捗管理機能とシステム動作モード管理機能のソフトウェア規模(ステップ数) を表 6 に示す．進捗管理機能は 77 ステップ，システム動作モード管理機能は 40 ステップで実現でき，実装負荷が小さいことが確認できた．

Vol.2019-ARC-240 No.47
Vol.2019-SLDM-191 No.47
Vol.2019-EMB-53 No.47
2019/2/28

表 6　提案フレームワークのソフトウェア規模

機能名	ステップ数
進捗管理機能	77
システム動作モード 管理機能	40

表 7　フレームワーク適用結果

フレームワーク	アプリ名	実行時間 (us)	
		最大値	最小値
あり	アプリ 1	1590	1497
	アプリ 2	10960	8132
なし	アプリ 1	1375	1333
	アプリ 2	10181	10049

5.4.2 オーバヘッド

進捗管理機能とシステム動作モード 管理機能，および，進捗通知 API の実行時間を 10000 回計測した．進捗管理機能，システム動作モード 管理機能，進捗通知 API はいずれも，アプリの実行状態を管理する変数を読み書きする程度の処理であるため，その実行時間はいずれも 1us〜2us 程度であった．オーバヘッドも小さく，フレームワークとして提供した場合のアプリケーションへの影響も小さいことが確認できた．

5.4.3 フレームワークの効果

本フレームワーク適用前後における実行時間の変化を**表 7**に示す．各アプリの基本処理完了までの時間を，アプリ 1とアプリ 2の実行回数の合計が 10000 回になるまで計測し，その実行時間の最大値と最小値を計測した．計測結果から，アプリ 2の実行時間最小値が 2ms 程度高速化されている一方で，基本処理の実行時間最大値は適用前後で大きな変化がないことが確認でき，デッドラインを守りながら，プロセッサを有効活用できる手法として一定の効果があることが確認できた．アプリ 1の実行時間については，適用前後で処理完了時間が 200us 程度遅くなっている．現在の実装では，進捗管理機能とシステム動作モード 管理機能を周期実行するタスクとして実装しているため，アプリの実行状態通知発生から進捗管理機能による検出，および，進捗管理機能による検出からシステム動作モード 管理機能による検出までの時間や，タイムウインドウ時間切り 替えによるオーバヘッドなどが原因と考えられる．

6. おわりに

本研究では，システムが提供する機能を基本処理と高度処理に分類し，それらの進捗を確認することにより，APU の利用効率を向上できる，TDMA ベースのフレームワークを提案した．本フレームワークでは，基本処理を高度処理よりも優先して処理を行うことでシステムに必要な安全性を確保するとともに，基本処理の進捗を基にして，アプリに割当てたタイムウインドウ時間を変動させることにより，動的なスケジューリングを可能にする．評価の結果，提案するスケジューリング手法により，基本処理完了までの時間を早めることができ，提案手法に一定の効果があることを確認することができた．今後の課題を以下に示す．

適切な時間割当て方法の検討

本提案では，タイムウインドウ時間を複数の基本処理に分配する場合，その実行時間を平均的に割付けている．しかし，他の割当て方法として，例えば，デッドラインが短い処理に優先的に割当てるという方式も考えられる．これら他の方式についても検討し，より効率的な割当て方式を検討する必要がある．

RTOS の 1 機能としての実装

本提案では，進捗管理機能とシステム動作モード 管理機能をそれぞれ，RTOS の機能を用いたアプリケーションとして作成している．そのため，RTOS のスケジューラがもつ 1 機能として実装するなど，ユーザが意識することなく本機能を使用可能にする必要がある．

実プログラム上での評価

本評価では，ビジーウェイトするサンプルアプリを用いて評価を行っているため，例えば，単純な PI 制御と，画像認識プログラムなどの，実際に適用を想定しているプログラムを用いて評価を行い，有効性を検証する必要がある．

マルチコアへの対応

アプリケーションプロセッサの多くは，高速化のためにマルチコア化していることが多い．本研究では問題の単純化のためにシングルコアのみを対象にしているが，今後，マルチコアにも対応できるようなプラットフォームにする必要がある．

参考文献

[1] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein and M. Wolf, ”Special Session: Future Automotive Systems Design: Research Challenges and Opportunities,” 2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2018

[2] M. Bechtel and H. Yun, ”Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention,” 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2019

[3] E. Lara et al., ”A New Approach to Guarantee Critical Task Schedulability in TDMA-Based Bus Access of Multicore Architecture”, 2019 IEEE Latin American Test Symposium, 2019

[4] A. Crespo et al., ”Execution Control in Mixed-Criticality Systems”, Int’l Conf. Embedded Systems, Cyber-physical Systems, & Applications ESCS’18, 2018

[5] B. Cilku, A. Crespo, P. Puschner, J. Coronel and S. Peiro, ”A TDMA-Based arbitration scheme for mixed-criticality multicore platforms,” 2015 International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP), 2015

[6] W. Ali and H. Yun, ”RT-Gang: Real-Time Gang Scheduling Framework for Safety-Critical Systems,” 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2019