情報処理学会論文誌 Vol.57 No.8 1752–1764 (Aug. 2016)

アスペクト指向プログラミングによるリアルタイムOSスケジューラのカスタマイズ

原田 祐輔[1,a)] 阿部 一樹[1,†1] 兪 明連[1,b)] 横山 孝典[1,c)]

受付日 2015年11月19日, 採録日 2016年5月17日

概要：組込みシステムは様々な用途に使用されるため，アプリケーションによって必要となるリアルタイムOSの機能は異なることが多い．しかし，リソース制約により，アプリケーションに応じて必要最小限の機能を備えるリアルタイムOSを提供することが望まれる．本論文では，リアルタイムOSの機能としてタスクスケジューリングを取り上げ，アスペクト指向プログラミングを用いることで，既存のソースコードを直接修正せずに，アプリケーションに適したスケジューリングにカスタマイズする手法について述べる．具体的には自動車制御分野向けのOSEK OSを対象に，仕様で規定された固定優先度スケジューリングをEDFまたはRMCLスケジューリングにカスタマイズするとともに，リソースアクセスプロトコルをそれらのアルゴリズム向けに変更するアスペクトを提案する．そして，実用上問題ないオーバヘッドとメモリ消費量で実装可能であることを示す．

キーワード：組込みシステム，リアルタイムOS，スケジューリングアルゴリズム，リソースアクセスプロトコル，アスペクト指向プログラミング

Aspect-oriented Customization of the Scheduler of a Real-time Operating System

Yusuke Harada[1,a)] Kazuki Abe[1,†1] Myongryun Yoo[1,b)] Takanori Yokoyama[1,c)]

Received: November 19, 2015, Accepted: May 17, 2016

Abstract: Embedded systems are used in various domains and the required functionalities of real-time operating systems (RTOSs) are different depending on applications. Most RTOSs adopt just fixed priority scheduling. Some applications, however, require dynamic scheduling algorithms. The paper presents a method to customize the scheduling algorithm and the resource access protocol of an OSEK OS using aspect-oriented programming (AOP). We define aspects to replace the fixed priority scheduling mechanism of the OSEK OS with an EDF (Earliest Deadline First) scheduling mechanism or a RMCL (Rate Monotonic Critical Laxity) scheduling mechanism. We also define aspects to customize the resource access protocol for EDF scheduling and RMCL scheduling. By using the aspects, we can customize the scheduling algorithm and the resource access protocol without modifying the original source codes. This improves the maintainability of the RTOS family. The evaluation results show that the overhead of AOP is small enough.

Keywords: embedded system, real-time operating system, scheduling algorithms, resource access protocols, aspect oriented programming

1. はじめに

組込みシステムは様々な用途に使用されるため，必要となるリアルタイムOSの機能もアプリケーションによって異なることが多い．しかし，リソース消費量の制約が大きい組込みシステムにおいて，それらすべての機能を備えたリアルタイムOSを搭載することは困難である．そこでア

[1] 東京都市大学
Tokyo City University, Tokyo 158–8557, Japan
[†1] 現在，株式会社PFU
Presently with PFU LIMITED
[a)] g1581519@tcu.ac.jp
[b)] myoo@tcu.ac.jp
[c)] tyoko@tcu.ac.jp

プリケーションの要求に応じて必要最小限の機能を備えたリアルタイム OS を提供することが望まれる．このため，単一のリアルタイム OS ではなく，必要な機能のみを備えたリアルタイム OS を選択可能なリアルタイム OS ファミリが望まれ，リアルタイム OS ファミリ化を効率的に実現する手法が求められている．

アプリケーションに応じて取捨選択可能とすべき機能の 1 つにタスクスケジューリングがある．既存のリアルタイム OS の多くは，固定優先度スケジューリングを採用している．しかし RM（Rate Monotonic）スケジューリング [1] に代表される固定優先度スケジューリングがすべてのアプリケーションにとって最適とは限らず，動的スケジューリングが求められる場合もある．

リアルタイム OS をカスタマイズすることを目的に横断的関心事を分離してモジュール化することのできるアスペクト指向プログラミング [2] を用いた研究がなされている．アスペクト指向プログラミングを用いることで，ソースコードを直接修正することなく，リアルタイム OS の機能の追加・変更が可能になる．これによりリアルタイム OS のソースコードの保守性を向上させることができる．

Afonso らは，リアルタイム OS の同期（排他制御）やロギングにアスペクトを適用している [3]．Park らは，プログラミング言語非依存なアスペクト指向プログラミング環境 AOX を開発し，カスタマイズ可能なリアルタイム OS への適用を提案している [4]．また Saito らは，既存のリアルタイム OS のソースコードを修正せずに，分散リアルタイム OS を実現するためのアスペクトを提案した [5]．

Beuche らは，アスペクト指向プログラミングを用いてアーキテクチャ非依存なリアルタイム OS を実現する手法を提案している [6]．その後，Lohmann, Spinczyk らのグループは，彼らが開発したアスペクト指向言語 AspectC++ [7] によるリアルタイム OS の実装を行い，そのオーバヘッドが十分小さいことを示した [8], [9]．リアルタイム OS ファミリの開発には最初からアスペクトを意識した設計が必要として Aspect-Aware Design に基づき AUTOSAR OS の機能のうち必要な機能のみを取捨選択可能としているリアルタイム OS ファミリの開発をしている [10], [11]．

Brandenburg らは，リアルタイム OS のスケジューリングアルゴリズムのプラグインとして構成し動的に変更可能とした [12] が，動的に変更するため ROM 化することができない．これに対しアスペクト指向プログラミングを用いてコンパイル時に静的に処理を織り込めば，ROM 化に対応できる．

しかしこれまでのところ，アスペクト指向によるリアルタイム OS のアルゴリズムを変更をするカスタマイズに関する研究は行われておらず，実現可能かどうかの予測はできていない．また Lohmann らのリアルタイム OS ファミリ [10], [11] は，あらかじめ想定される AUTOSAR OS 仕様内での機能の取捨選択は可能であるが，アルゴリズムを変更するような想定されていない仕様への変更は扱っていない．

そこで我々の研究の目的は，既存のリアルタイム OS のソースコードを直接修正せずに，アスペクト指向プログラミングを用いてスケジューリングをカスタマイズする手法を提案することである．本論文では固定優先度スケジューリングアルゴリズムを EDF または RMCL スケジューリングにカスタマイズするアスペクトを提案するとともに，実用上問題ないオーバヘッド，メモリ消費量で実装可能であることを示す．

以下本論文の構成は次のとおりである．まず 2 章でアスペクト指向プログラミングを用いたリアルタイム OS ファミリのスケジューリングアルゴリズムとリソースアクセスプロトコルのカスタマイズ方法の概要について述べる．3 章でスケジューリングの機構を説明し，カスタマイズするためのアスペクトについて述べる．4 章でリソースアクセスプロトコルについて説明し，カスタマイズするためのアスペクトについて述べる．5 章で，アスペクト指向プログラミングでカスタマイズしたリアルタイム OS の評価を示す．6 章で，関連研究との比較について述べる．7 章で本論文のまとめについて述べる．

2. アスペクト指向プログラミングによるリアルタイム OS のカスタマイズ

2.1 カスタマイズ対象

我々は，自動車制御分野の標準である OSEK OS 仕様 [13] に基づくオープンソースのリアルタイム OS である TOPPERS/ATK1 [14] を対象に固定優先度スケジューリングを動的優先度スケジューリングにカスタマイズする．

動的優先度アルゴリズムの多くは，デッドラインあるいは余裕時間を用いて優先度を決定している．前者の代表として，動的スケジューリングアルゴリズムで最もよく用いられている EDF（Earliest Deadline First）スケジューリング [1] をとりあげ，カスタマイズ対象とする．一方後者の代表である LLF（Least Laxity First）スケジューリングは単位時間ごとにスケジューラを起動するため実行効率に問題があり，実用性に欠ける．そこで余裕時間を用いる手法として，タスクの起動や終了などのタイミングでスケジューラを起動すればよい RMCL（Rate Monotonic Critical Laxity）スケジューリング [15] をカスタマイズ対象として選ぶ．

2.2 アスペクト指向プログラミングによるカスタマイズ方法

TOPPERS/ATK1 の大部分は C 言語で記述されているため，C 言語ベースのアスペクト指向言語 ACC（AspeCt-oriented C）[16], [17] を用いる．ACC は AspectJ [18] や

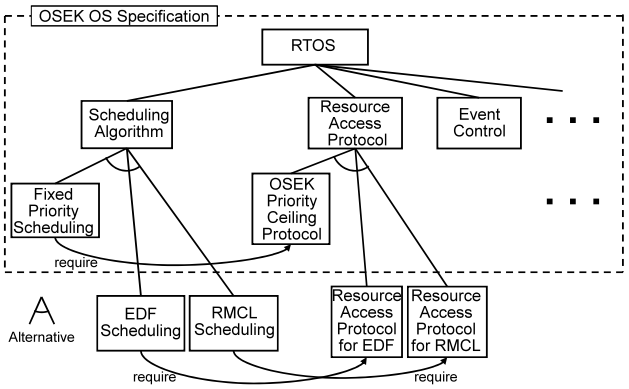


図 1 RTOS ファミリのフィーチャモデル
Fig. 1 Feature model of a real-time operating system family.

Aspect C++と同様な，ジョインポイントモデルに基づくアスペクト指向言語である．

ACC で扱えるジョインポイントは，関数の呼び出し（call）と実行（execution），変数への値の書き込み（set）と値の読み出し（get）である．アスペクトはポイントカット（pointcut）とアドバイス（advice）からなる．ポイントカットはジョインポイントの集合を指定するもので，アドバイスはポイントカットに合致するジョインポイントで実行する処理を記述したものである．ACC は before，after，around の 3 つのアドバイスをサポートしており，対象ジョインポイントの前後あるいはその代わりに，アドバイスコードを実行することができる．

ACC はトランスレータとして実装されており，ACC および C のソースファイルを入力し，織り込みを行った後，C のソースファイルを出力する．出力されたファイルを C コンパイラによりコンパイルすることでオブジェクトコードを生成する．

2.3 カスタマイズ項目とアスペクト

リアルタイム OS ファミリが提供する機能はフィーチャモデル [19] で表現することができ，その一部を**図 1** に示す．リアルタイム OS は多くのフィーチャを持つが，スケジューリングアルゴリズムとリソースアクセスプロトコルのフィーチャをカスタマイズ可能とする．図の破線内が OSEK OS 仕様内のフィーチャである．スケジューリングアルゴリズムのオルタナティブフィーチャを，固定優先度スケジューリング，EDF スケジューリング，RMCL スケジューリングとし，それらから選択可能とする．また，リソースアクセスプロトコルのオルタナティブフィーチャを OSEK 優先度上限プロトコル，EDF 向けリソースアクセスプロトコル，RMCL 向けリソースアクセスプロトコルとし，スケジューリングアルゴリズムに応じて選択する．

EDF スケジューリングと RMCL スケジューリングにカスタマイズするためのアスペクトの一覧を**表 1** に示す．EDF スケジューリング向けに 10 個のアスペクトを，RMCL スケジューリング向けにも 11 個のアスペクトを定義する．うち 2 個のアスペクトは共通である．

表 1 スケジューリングアルゴリズムとリソースアクセスプロトコルのためのアスペクト
Table 1 Aspects for customization of scheduling algorithm and resource access protocol.

Aspect		EDF	RMCL
Scheduling	Ready Queue Operation	x	x
	Selection of Task to Run	x	x
	Preemption	x	x
	Scheduler Call	x	x
	Remaining Execution Time Maintenance	-	x
	Absolute Deadline Update	x	
	Task Initialization	x	x
Resource	Resource Management	x	x
	Ceiling Priority Maintenance	x	-
	Search Shared Resource	-	x
	Resource Occupation Maintenance	x	
	Resource Initialization	x	x

```
<file> ::=
  <OIL_version>
  <implementation/definition>
  <application_definition>
  …
<parameter_list> ::=
  /* empty definition */
  | <parameter>
  | <parameter_list><parameter>
<parameter> ::=
  <attribute_name> = <attribute_value><description>";"
<attribute_name> ::= <name> | <object>
<attribute_value> ::=
  <name>
  …
  | <number>
  …
<name> ::=
  …
<number> ::= <dec_number> | <hex_number>
```

図 2 OIL の構文
Fig. 2 Syntax of OIL description.

2.4 コンフィグレーション

OSEK OS では，アプリケーションに応じた設定（コンフィギュレーション）のために OIL（OSEK Implementation Language）[20] を用いる．タスクの宣言やイベントの設定などを OIL で記述し，SG（System Generator）に通すことで，アプリケーション依存の構成データを記述したソースコードを得る．本研究では，EDF および RMCL スケジューリングに対応するために OIL を拡張する．

図 2 は OIL の構文を BNF 記法で表記したものである．OS のパラメータ（parameter）は属性名（attribute_name）とその値（attribute_value）の組合せで記述する．スケジューリングアルゴリズムを選択するための OS オブジェクトの属性名として SCHEDULER を追加する．また値として固

```
/* Definition of OS object */
OS sample_os {
  STATUS = EXTENDED;
  STARTHOOK = TRUE;
 /* Attribute to select
       scheduling algorithm */
  SCHEDULER = RMCL;
  ......
}

/* Definition of task object */
TASK sample_task1 {
  AUTOSTART = TRUE {
    APPMODE = sample_mode1;
  };
  RESOURCE = sample_res1;

  /* Attribute to declare
       relative deadline and Worst Case Execution Time */
  DEADLINE  = 10;
  WCET      = 3;
......
}
```

図 3 OIL 記述例
Fig. 3 Example OIL description.

定優先度を表す FPRIORITY，EDF を表す EDF，RMCL を表す RMCL を記述可能とする．次に，TASK オブジェクトの属性名として，EDF および RMCL で必要となる相対デッドラインを表す DEADLINE を，RMCL で必要となるタスクの最悪実行時間を表す WCET（Worst Case Execution Time）を追加する．**図 3** に拡張した OIL の記述例を示す．この例では OS オブジェクトで RMCL スケジューラを選択している．タスクオブジェクトでは sample_task1 の相対デッドラインを 10，最悪実行時間を 3 に設定している．

また，OIL 記述から EDF スケジューラまたは RMCL スケジューラで用いるタスクのデッドラインなどをタスクコントロールブロック（Task Control Block，TCB）へ追加するために SG を拡張した．

3. スケジューリングアルゴリズムのカスタマイズ

3.1 固定優先度スケジューリング

図 4 に TOPPERS/ATK1 の固定優先度スケジューリングの機構を示す．実行可能状態のタスクは，優先度ごとに用意されるレディキューに格納される．タスクの優先度は TCB に設定する．実行タスクが終了すると，関数 search_schedtsk() が優先度が最も高いレディキューの先頭タスクを選び実行する．システムコール ActivateTask()，ChainTask() またはアラーム機構によってタスクを起動すると，関数 make_active() が呼ばれる．次に関数 make_runnable() がタスクを実行可能状態にし，優先度に応じたレディキューへ格納する．起動するタスクの優先度が実行中タスクの優先度よりも高い場合はプリエンプションが発生する．プリエンプションは関数 preempt() が行い，プリエンプトされたタスクを優先度に応じたレディキューに格納する．

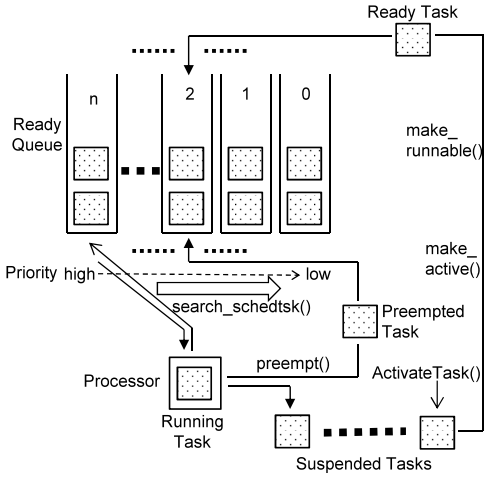


図 4 固定優先度スケジューリングの機構
Fig. 4 Fixed priority scheduling mechanism.

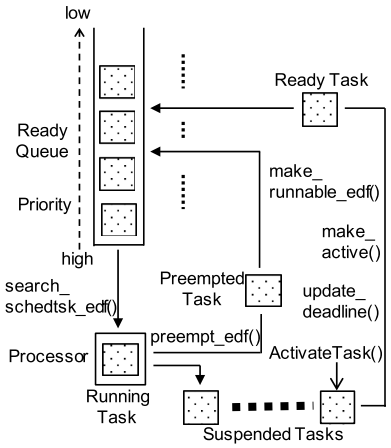


図 5 EDF スケジューリングの機構
Fig. 5 EDF scheduling mechanism.

3.2 EDF スケジューリング

EDF スケジューリングではタスクの優先度を絶対デッドラインで表す．前述のように，OIL によって宣言されたタスクの相対デッドラインは SG によりコンフィグレーションデータに変換され記憶されている．EDF スケジューラはタスク起動時に，相対デッドラインをもとに絶対デッドラインを求めて TCB に設定し，タスクの優先度として用いる．

EDF スケジューリングの機構を**図 5** に示す．タスクを起動すると関数 update_deadline() が，タスクの相対デッドラインをもとに，タスクの絶対デットラインを算出し TCB に設定する．そして，関数 make_active() が呼ばれた後 make_runnable_edf() が起動タスクを実行可能状態へ遷移させ，起動タスクと実行中タスクの絶対デッドラインを比較し，絶対デッドラインが近いタスクを実行タスクとし，もう一方のタスクを単一レディキューへ格納する．

これにより，タスクが実行可能状態になると，絶対デッドラインの近いものから順に単一レディキューに格納され，実行中のタスク以外で最も近い絶対デッドラインを持

表 2 EDF スケジューリングのためのアスペクト
Table 2 Aspects for EDF scheduling algorithm.

Aspect	Join Point	Advice
EDF-Based Ready Queue Operation	execution(make_runnable())	around
EDF-Based Selection of Task to Run	execution(search_schedtsk())	around
EDF-Based Preemption	execution(preempt())	around
EDF Scheduler Call	execution(Schedule())	around
Absolute Deadline Update	execution(make_active())	before
Task Initialization for EDF	execution(task_initialize())	around

```
/* replace the fixed-priority-based
     ready queue operation for the ready task
     with the deadline-based ready queue operation */
BOOL around(TaskType tskid) :
    execution(BOOL make_runnable(TaskType))
    && args(tskid) {
  return(make_runnable_edf(tskid));
  /* make the state of the task ready and
     enqueue the task into the ready queue
             accoording to the absolute deadlines */
}
```

図 6 EDF 向けレディキュー操作のためのアスペクト
Fig. 6 Aspect for EDF-based ready queue operation.

つタスクがレディキューの先頭になる．プリエンプションが発生する場合は preempt_edf() が実行中タスクを単一レディキューへ格納しプリエンプションを行い，絶対デッドラインの近いタスクを単一レディキューの先頭から取り出し実行タスクに設定する．実行中のタスクが終了またはプリエンプトされると，search_schedtsk_edf() がデッドラインの近い順番にソートした単一レディキューの先頭からタスクを取り出し，実行タスクに設定する．プリエンプトされたタスクはレディキューの絶対デッドラインに応じた位置に格納する．

図 4 の固定優先度スケジューリング機構を図 5 の EDF スケジューリング機構にカスタマイズするためのアスペクトの一覧を**表 2** に示す．Join Point の欄はアスペクトのポイントカットが指定したジョインポイントを示し，Advice の欄はアドバイスの種類を示している．

EDF 向けレディキュー操作のアスペクト（EDF-Based Ready Queue Operation）のソースコードを**図 6** に示す．このアスペクトは around アドバイスで起動タスクを実行可能状態にし，実行タスクを決める関数 make_runnable() を make_runnable_edf() の呼び出しに置き換える．

EDF 向け実行タスク選択のアスペクト（EDF-Based Selection of Task to Run）は around アドバイスで，最高優先順位探索する関数 search_schedtsk() を search_schedtsk_edf() の呼び出しに置き換える．

```
/* update the absolute deadline of the activated task */
before(TaskType tskid) :
    execution(BOOL make_active(TaskType)) && args(tskid) {
        update_deadline(tskid);
}
```

図 7 絶対デッドライン更新のアスペクト
Fig. 7 Aspect for absolute deadline update.

EDF 向けプリエンプションのためのアスペクト（EDF-Based Preemption）は aroud アドバイスで，関数 preempt() を preempt_edf() の呼び出しに置き換える．

EDF 向けスケジューラ呼び出しのためのアスペクト（EDF Scheduler Call）は around アドバイスで，スケジューリングを行うシステムコール Schedule() を Schedule_edf() の呼び出しに置き換える．Schedulei_edf() は，ノンプリエンプティブタスクが混在する場合，プリエンプションを発生させ絶対デッドラインの近いタスクを実行タスクに設定する．

絶対デッドライン管理のためのアスペクト（Absolute Deadline Update）のソースコードを**図 7** に示す．このアスペクトは before アドバイスで，起動タスクの起動処理を行う関数 make_active() の実行前に関数 update_deadline() を呼び出す．このアスペクトは EDF と RMCL 共通で用いる．

EDF 向けのタスクデータ初期化も必要なので，そのためのアスペクト（Task Initialization for EDF）は around アドバイスで EDF 向けのタスクデータの初期化を行う．

3.3 RMCL スケジューリング

RMCL スケジューリングはデッドラインまでの時間と残り実行時間の差である余裕時間を使用する．前述のように，OIL で宣言したタスクの相対デッドラインと最悪実行時間は SG によりコンフィグレーションデータに変換され記憶されている．残り実行時間は TCB に記憶する．タスク起動時に，最悪実行時間をタスクの残り実行時間として記憶するとともに，スケジューラがタスクの残り実行時間を更新する．RMCL では実行中のタスクの残り実行時間より余裕時間が小さいタスクが発生した場合，それをクリティカルタスクと呼び，その優先度を最高優先度とする．

RMCL スケジューリングの機構を**図 8** に示す．タスクを起動すると関数 update_deadline() がタスクの絶対デッドラインを設定する．そして，関数 make_active() が呼ばれた後，make_runnable_rmcl() が起動タスクを実行可能状態にし，実行中タスクとともにレディキューへ格納する．その後レディキュー中からクリティカルタスクの探索を行い実行タスクを決める．そして，search_schedtsk_rmcl() がレディキュー中からクリティカルタスクの探索を行い，あ

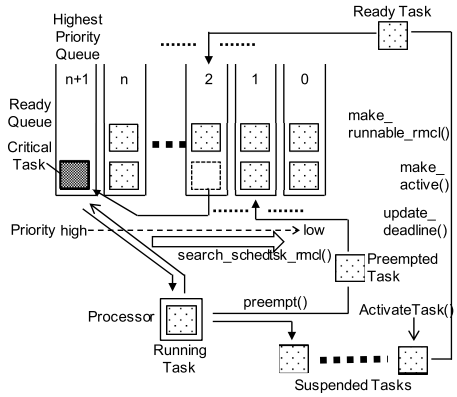


図 8 RMCL スケジューリングの機構
Fig. 8 RMCL scheduling mechanism.

表 3 RMCL スケジューリングのためのアスペクト
Table 3 Aspects for RMCL scheduling algorithm.

Aspect	Join Point	Advice
RMCL-Based Ready Queue Operation	execution(make_runnable())	around
RMCL-Based Selection of Task to Run	execution(search_schedtsk())	around
RMCL-Based Preemption	execution(preempt())	before
RMCL Scheduler Call	execution(Schedule())	around
Remaining Execution Time Maintenance	execution(make_runnable())	before
	execution(search_schedtsk())	before
	execution(search_schedtsk())	after
Absolute Deadline Update	execution(make_active())	before
Task Initialization for RMCL	execution(task_initialize())	before

る場合はクリティカルタスクに設定されているリソースの空き状況を確認する．使用中の場合はクリティカルタスクを最高優先度レディキューへ格納し，リソースを獲得しているタスクの優先度を引き上げて先に実行タスクに設定する．その後，レディキューからクリティカルタスクを取り出して実行する．クリティカルタスクがない場合は OSEK OS の固定優先度スケジューリングと同様の動作を行う．

図 4 の固定優先度スケジューリング機構を図 8 の RMCL スケジューリング機構にカスタマイズするためのアスペクトの一覧を**表 3** に示す．

RMCL 向けレディキュー操作のアスペクト（RMCL-Based Ready Queue Operation）は around アドバイスで，起動タスクを実行可能状態にし，実行タスクを決める関数 make_runnable() を make_runnable_rmcl() に置き換える．

RMCL 向け実行タスク選択のアスペクト（RMCL-Based Selectioin of Task to Run）は around アドバイスで，最高優先順位タスクを探索する関数 search_schedtsk() を search_schedtsk_rmcl() の呼び出しに置き換える．

RMCL 向けプリエンプションのアスペクト（RMCL-Based Preemption）は before アドバイスで，プリエンプションを行う関数 preempt() の前に次に実行するタスクの優先度の更新を行う処理を追加している．クリティカルタスクの探索をプリエンプションの処理内で行うためである．

RMCL 向けスケジューラの呼出のアスペクト（RMCL Scheduler Call）は around アドバイスで，スケジューリングを行うシステムコールの Schedule() を Schedule_rmcl() に置き換える．Schedule_rmcl() は最高優先度タスクを探索するとともに，クリティカルタスクを探索し実行タスクを決める．

タスクの残り実行時間の管理のアスペクト（Remaining Execution Time Maintenance）は before アドバイスで関数 make_runnable() 実行前に最悪実行時間でタスクの残り実行時間を設定する．また before アドバイスと after アドバイスで search_schedtsk() の実行前に残り実行時間の更新を行い，実行後にタスクの実行開始時刻の設定を行う．これらはタスクの余裕時間の算出に用いられる時間である．

絶対デッドライン更新のアスペクト（Absolute Deadline Update）は EDF と共通のアスペクトである．

RMCL 向けのタスクデータ初期化も必要なので，そのためのアスペクト（Task Initialization for RMCL）は before アドバイスで RMCL 向けのタスクデータの初期化を行う．

4. リソースアクセスプロトコルのカスタマイズ

4.1 OSEK 優先度上限プロトコル

OSEK OS のリソースアクセスプロトコルは，スタックリソースポリシ [21] をベースに固定優先度向けに簡略化したもので，OSEK 優先度上限プロトコル（OSEK Priority Ceiling Protocol）と呼ばれる．リソースの上限優先度はそのリソースにアクセスするタスクの中から最も高い値が設定される．システムコール GetResource() によりリソースを獲得し，実行中タスクの優先度がリソースの上限優先度よりも低い場合，タスクの優先度をリソースの上限優先度へ引き上げる．システムコール ReleaseResource() によりリソースを解放するとタスクの優先度はリソースを獲得する前の優先度に戻る．

図 9 に OSEK 優先度上限プロトコルの実行例を示す．TaskA，TaskB，TaskC はリソース Res を共有している．TaskA の優先度は TaskB よりも高く，TaskB の優先度は TaskC よりも高い．リソース Res の上限優先度は TaskA の優先度と同じ値となる．t1 で TaskC が起動される．TaskB が t2 で起動され，TaskC をプリエンプトする．t3 で TaskB が GetResource(Res) を呼び出すと，TaskB の優先度は Res の上限優先度に引き上がる．TaskA が t4 で起動されるが，

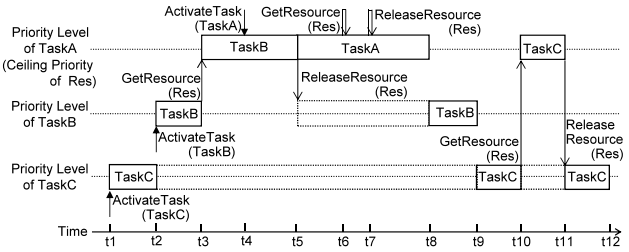


図 9 OSEK 優先度上限プロトコル
Fig. 9 OSEK priority ceiling protocol.

TaskB はプリエンプトされない．次に t5 で TaskB が ReleaseResource(Res) を呼び出すと，TaskB の優先度はもとの優先度へ戻り，TaskA がプリエンプトし，実行が始まる．TaskA が終了すると，t8 で TaskB の実行が再開する．t9 で TaskB が終了すると TaskC の実行が再開する．t10 で TaskC が GetResource(Res) を呼び出すと TaskC は上限優先度に引き上がり，t11 で TackC が ReleaseResource(Res) を呼び出すともとの優先度へ戻る．

4.2 EDF 向け優先度上限プロトコル

EDF スケジューリングでは，タスクの優先度は絶対デッドラインで表され，動的に変化する．そこで，リソースを要求したタスクの中で絶対デッドラインが最も近いタスクのデッドラインを上限優先度としてリソースの管理を行う．

まずリソースの上限優先度管理について説明する．タスクがリソースを獲得するとき，そのタスクの優先度をリソースの上限優先度に設定する．リソースを共有するほかのタスクが起動され，そのタスクの優先度のほうが高い場合は上限優先度を更新する．リソースの上限優先度はタスクがリソースを解放するまで，リソースを共有するタスクの中で最高の優先度が設定される．どのタスクもリソースを獲得していないときはリソースの上限優先度は設定しない．

次にタスクの優先度管理について説明する．タスクの優先度は獲得したリソースの上限優先度に一致させる．リソースを共有するほかのタスクが起動されると，リソースの上限優先度を引き上るとともに，その時点でリソースを獲得しているタスクの優先度を上限優先度まで一時的に引き上げる．タスクがすべてのリソースを解放すると，一時的に引き上げられたタスクの優先度はもとの優先度へ戻る．

図 10 に EDF 向け優先度上限プロトコルの動作例を示す．TaskA，TaskB，TaskC はリソース Res を共有している．TaskA の絶対デッドラインは TaskB よりも早く，TaskB の絶対デッドラインは TaskC よりも早い．t1 で TaskC が起動されると Res の上限優先度は TaskC と同様の値になる．TaskB が t2 で起動されると Res の上限優先度は TaskB の優先度と同様になる．t3 で TaskB が GetResource(Res) を呼び出す．TaskA が t4 で起動されると Res の上限優先度

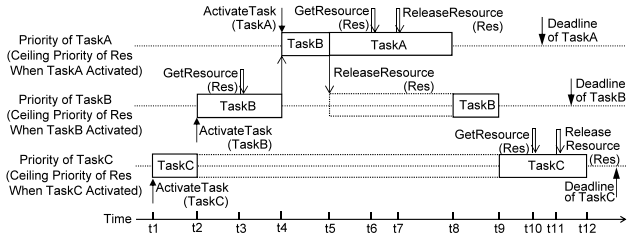


図 10 EDF 向け優先度上限プロトコル
Fig. 10 Priority ceiling protocol for EDF.

表 4 EDF 向けリソースアクセスプロトコルのためのアスペクト
Table 4 Aspects for resource access protocol for EDF.

Aspect	Join Point	Advice
Resource Management for EDF	execution(GetResource())	around
	execution(ReleaseResource())	around
Ceiling Priority Maintenance	execution(update_deadline())	after
Resource Occupation Maintenance	execution(GetResource())	before
	execution(ReleaseResource())	before
Resource Initialization for EDF	execution(resource_initialize())	before

```
/* replace the original resource management functions
     with the resource management functions
                                  for EDF scheduling */

StatusType around(ResourceType resid) :
    execution(StatusType GetResource(ResourceType))
    && args(resid) {
  return(GetResource_edf(resid));
  /* function to get the resource based on the
       resource access protocol for EDF scheduling */
}

StatusType around(ResourceType resid) :
   execution(StatusType ReleaseResource(ResourceType))
    && args(resid) {
  return(ReleaseResource_edf(resid));
  /* function to release the resource based on the
       resource access protocol for EDF scheduling */
}
```

図 11 EDF 向け優先度上限プロトコルのためのアスペクト
Fig. 11 Aspect forpriority ceiling protocol for EDF.

は TaskA と同じ値になり，TaskB の優先度は上限優先度に引き上がる．TaskB が t5 で ReleaseResource(Res) を呼び出すと，TaskB の優先度はもとに戻り，TaskA の実行が始まる．

OSEK 優先度上限プロトコルを EDF 向け優先度上限プロトコルに置き換えるアスペクトを**表 4** に示す．

EDF 向けリソース管理のためのアスペクト（Resource Management for EDF）のソースコードを**図 11** に示す．このアスペクトは around アドバイスで，リソースの獲得を行うシステムコール GetResource() を GetResource_rmcl() に置き換える．GetResource() は獲得したリソース ID に

応じてビットマップを更新しデッドラインをリソースの上限優先度へ一時的に引き上げる．また同様に around アドバイスで，リソースの解放を行うシステムコール ReleaseResource() を ReleaseResource_rmcl() に置き換える．ReleaseResource() は解放するリソース ID に応じてビットマップを更新し，一時的に引き上げたタスクの上限優先度をもとの優先度へ戻す．

EDF 向け上限優先度管理のためのアスペクト（Ceiling Priority Maintenance）は after アドバイスでタスクの絶対デッドラインを更新する関数 update_deadline() 実行後に ref_deadline() を呼び出す．ref_deadline() はデッドラインが更新されたタスクとリソースを共有しているタスクが，リソースを獲得していないかを調べる関数である．また，他のタスクがリソースを獲得していて，そのタスクのデッドラインが実行状態になるタスクより遅い場合，リソースを獲得しているタスクのデッドラインを早めて実行を継続する処理も行う．

リソース使用状況管理のためのアスペクト（Resource Occupation Maintenance）は before アドバイスで，リソース獲得を行う GetResource() またはリソースの解放を行う ReleaseResouece() の実行前にリソースの使用状況の更新を行う．リソースの獲得時はリソースを使用状態に更新し，どのタスクがリソースを獲得するかを設定する．リソースの解放時はリソースを解放状態に更新し，リソースが空いている状態に設定する．このアスペクトは RMCL の優先度上限プロトコルと共通のアスペクトである．

EDF 向けのリソースデータ初期化のためのアスペクト（Resource Initialization for EDF）は before アドバイスで EDF 向けのリソースデータの初期化を行う．

4.3 RMCL 向け優先度上限プロトコル

RMCL スケジューリングアルゴリズムでのリソースアクセスプロトコルを OSEK 優先度上限プロトコルをベースに設計する．クリティカルタスクがない場合は OSEK OS における OSEK 優先度上限プロトコルと同じ動作をする．ここではクリティカルタスクがある場合の動作について説明する．

まず，リソースの上限優先度管理について説明する．あるタスクがクリティカルタスクになると，獲得しているリソースと他のタスクと共有しているリソースの上限優先度を，クリティカルタスクの優先度と同様にタスクの中で最高の優先度に設定する．リソースを解放するとリソースの上限優先度をもとに戻す．

次に，タスクの優先度管理について説明する．クリティカルタスクがリソースを獲得する場合は，タスクの優先度はタスク内での最高優先度のまま変更されない．あるタスクがクリティカルタスクになる前に，リソースを共有している他のタスクがリソースを獲得している場合は，タスク

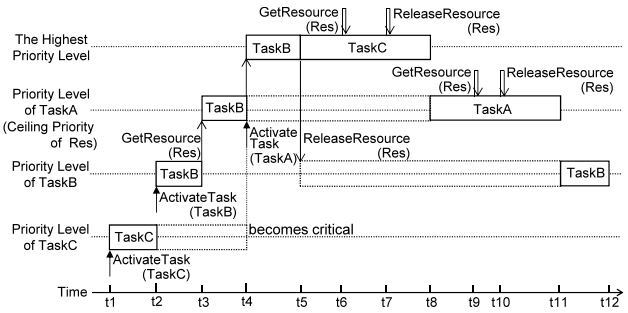


図 12 RMCL 向け優先度上限プロトコル
Fig. 12 Resource access protocol for RMCL scheduling.

表 5 RMCL 向けリソースアクセスプロトコルのためのアスペクト
Table 5 Aspects for resource access protocol for RMCL.

Aspect	Join Point	Advice
Resource Management for RMCL	execution(GetResource())	around
	execution(ReleaseResource())	around
Search Shared Resource	call(Search_schedtsk())	after
Resource Occupation Maintenance	execution(GetResource())	before
	execution(ReleaseResource())	before
Resource Initialization for RMCL	execution(resource_initialize())	before

がクリティカルタスクとなった時点で，クリティカルタスクとリソースを共有しているタスクの優先度をタスク内での最高優先度に引き上げる．クリティカルタスクとリソースを共有するタスクがリソースを解放すると優先度はもとの優先度に戻る．クリティカルタスクがリソースを解放すると優先度はタスク内での最高優先度のまま変更されず実行を続ける．

図 12 に RMCL 優先度上限プロトコルの動作例を示す．タスクとリソースは図 10 と同様である．この例では t4 で TaskA がクリティカルタスクになる．TaskC を最高優先度に引き上げる前に，TaskB の優先度を最高優先度に引き上げる．これは TaskB が Res を獲得しているためである．t5 で TaskB が ReleaseResource(Res) を呼び出すと，TaskB の優先度はもとの優先度へ戻り，TaskC の実行が始まる．T8 で TaskC 終了すると，TaskA の実行が始まる．t11 で TaskA が終了すると，TaskB が再開する．

OSEK 優先度上限プロトコルを RMCL 向け優先度上限プロトコルに置き換えるアスペクトを**表 5** に示す．

RMCL 向けリソース管理のアスペクト（Resource Management for RMCL）のソースコードを**図 13** に示す．このアスペクトは around アドバイスで，リソースの獲得を行うシステムコール GetResource() を GetResource_rmcl() に置き換える．GetResource_rmcl() はクリティカルタスクの場合はリソースの上限優先度をクリティカルタスクの優

先度と同様のタスク内での最高優先度に設定する．また同様に around アドバイスで，リソースの解放を行うシステムコール ReleaseResource() を ReleaseResource_rmcl() に置き換える．ReleaseResource_rmcl() は，クリティカルタスクのときはリソース解放後も最高優先度から変更しない．クリティカルタスクとリソースを共有しているタスクの場合は，タスクの優先度に応じたレディキューへ格納しクリティカルタスクを最高優先度レディキューから取り出し実行タスクに設定する．

RMCL 向け共有リソース探索のアスペクト（Search Shared Resources）は after アドバイスで，RMCL 向け実行タスク選択のアスペクト内のクリティカルタスクを探索する関数 search_criticaltsk() の実行後にクリティカルタスクに設定されている共有リソースの空き状況を確認し優先度管理を行う．

リソース獲得状況管理のためのアスペクト（Resource Occupation Maintenance）は EDF の優先度上限プロトコルと共通のアスペクトである．

RMCL 向けのリソースデータ初期化のためのアスペクト（Resource Initialization for RMCL）は before アドバイスで RMCL 向けのリソースデータの初期化を行う．

```
/* replace the original resource management functions
     with the resource management functions
                                for RMCL scheduling */

StatusType around(ResourceType resid) :
    execution(StatusType GetResource(ResourceType))
    && args(resid) {
  return(GetResource_rmcl(resid));
  /* function to get the resource based on the
       resource access protocol for RMCL scheduling */
}

StatusType around(ResourceType resid) :
   execution(StatusType ReleaseResource(ResourceType))
    && args(resid) {
  return(ReleaseResource_rmcl(resid));
  /* function to release the resource based on the
       resource access protocol for RMCL scheduling */
}
```

図 13 RMCL 向けリソース管理のためのアスペクト
Fig. 13 Aspect resource management for RMCL.

5. 実装および評価

5.1 実験環境

本研究で評価に用いた実験環境を以下に示す．マイクロコントローラ H8S/2638F を搭載した評価ボードを使用した．H8S/2638F は 256 kB の ROM，16 kB の RAM を内蔵し，クロック周波数は 20 MHz である．カスタマイズする RTOS として TOPPERS/ATK1 Release1.0 を用いた．C コンパイラは GCC 3.4.6，最適化オプションは ATK1 の Makefile で指定している O2 を用いた．アスペクト指向言語には ACC ver0.9 を用いた．

5.2 オーバヘッドの評価

アスペクト指向プログラミングによるオーバヘッドを評価するため，アスペクト指向によりカスタマイズした機能に関わるシステムコールについて実行時間を測定し，ソースコードを直接書き換えて実装した同一機能のリアルタイム OS の実行時間と比較する．実行時間の測定には 5 MHz のハードウェアタイマを用いた．

表 6 にシステムコールの実行時間を示す．RMCL の場合はクリティカルタスクの存在する場合と存在しない場合に分けて示す．また ActivateTask についてはタスクスイッチが発生する場合と発生しない場合に分けて示す．参考のため，オリジナルの TOPPERS/ATK1 での固定優先度の場合の実行時間も示す．表に示した値は各システムコールの発行から終了までにかかる時間を 100 回計測した平均値である．

システムコールとシステムコールが呼び出す関数を**表 7** に示し，その実行時間を**表 8** に示す．

EDF の Rewrite と RMCL の Rewrite と Fixed のそれぞれの差がアルゴリズムの違いによる実行時間の差を示す．また，各スケジューラでの AOP の実行時間と Rewrite の実行時間の差がアスペクト指向プログラミングによるオー

表 6 システムコールの実行時間
Table 6 Execution time of system call.

System Call		Execution Time [μsec]						
		EDF		RMCL				Fixed
				with a critical task		without a critical task		
		AOP	Rewrite	AOP	Rewrite	AOP	Rewrite	
ActivateTask()	with task switch	74.0	66.5	138.8	132.2	123.1	116.6	49.2
	without task switch	64.8	57.2	149.5	143.0	-	-	40.5
TerminateTask()		29.7	27.3	149.5	143.0	61.1	56.6	40.5
ChainTask()		81.0	70.4	180.5	168.9	162.5	150.8	60.4
Schedule()		56.6	53.6	104.2	98.9	88.5	83.3	51.2
WaitEvent()		36.8	33.8	90.6	85.4	67.6	64.6	45.2
GetResource()		27.1	27.1	31.2	31.2	31.1	31.1	20.7
ReleaseResource()		36.3	36.3	133.7	128.5	115.0	109.8	21.7

表 7 システムコール内呼び出し関数
Table 7 Functions in system call.

システムコール	呼び出し関数
ActivateTask()	make_active() make_runnable()
TerminateTask()	search_schedtsk()
ChainTask()	search_schedtsk()
Schedule()	preempt() search_schedtsk()
WaitEvent()	search_schedtsk()

表 8 システムコール内呼び出し関数の実行時間
Table 8 Execution time of function in system call.

Function	Execution Time [μsec]				
	EDF		RMCL		Fixed
	AOP	Rewrite	AOP	Rewrite	
make_active()	26.8	22.4	18.6	18.6	7.4
make_runnable()	25.6	22.6	118.6	112.0	20.8
search_schedtsk()	9.2	6.2	42.2	37.2	20.4
preempt()	23.8	20.8	76.2	71.0	26.6

バヘッドを示す．

RMCL スケジューラでは固定優先度スケジューラをベースに拡張しているため，RMCL スケジューラの実行時間の方が大きい．一方，固定優先度スケジューラと EDF スケジューラとは処理が異なる．そこで固定優先度スケジューラと EDF スケジューラの実行時間の違いについて説明する．固定優先度スケジューリングと EDF スケジューリングのオーバヘッドを比較する．タスク起動のシステムコール ActivateTask()，ChainTask() は関数 make_active() と make_runnable() を呼び出す．EDF スケジューラの方が固定優先度スケジューラより実行時間が長いのは，EDF スケジューリングでは起動タスクのデッドラインを算出し，デッドラインの近い順にレディキューへ格納する必要があるため，起動処理の計算量が大きくなる．Schedule() は，関数 preempt() と search_schedtsk() を呼び出す．デッドラインの比較を行い，プリエンプトをするかどうかを判断する必要があるため EDF の方が計算量が大きい．GetResoruce()，ReleaseResource() では固定優先度スケジューリングで行わないリソース取得状況を管理しているため，計算量が大きくなる．また，TerminateTask() は EDF スケジューリングの方が固定優先度スケジューリングよりもオーバヘッドが小さい．これは，タスクキューの先頭からタスクを取り出すだけのためである．固定優先度スケジューラだと，最高優先度を参照し，その優先度のレディキューから次に実行するタスクを取り出すため，固定優先度アルゴリズムよりも計算量が小さい．

アスペクト指向プログラミングにより実装すると，ACC から出力されるアスペクトに記述したアドバイスコードは C コードの inline 関数に変換される．そのため，関数呼び出しの回数が増えることはない．しかし，アドバイスコードで引数を用いる場合，引数を一時変数に格納してから参照する処理が追加され，戻り値を必要とする場合，一時変数に戻り値を格納してからリターン値として返すため，オーバヘッドが発生する．そのため，同じポイントカットでアスペクトを複数織り込むとオーバヘッドが大きくなる．しかし，アスペクトでも直接書き換えても（引数などを除いて）直接書き換えたものと同じコードになるが，最適化がされにくくなる．コードが単純で同程度の最適化が可能な場合は同じになり，コードが複雑で同程度の最適化ができない場合は直接書き換えた場合は最適化され，アスペクトを使った場合は最適化されないことがある．

表 9 メモリ消費量
Table 9 Memory consumption.

RTOS		Memory Consumption [Byte]		
Scheduling	Customized by	Code	Data (ROM)	Data (RAM)
EDF	AOP	15080	832	144
	Rewrite	13672	832	144
RMCL	AOP	15948	848	139
	Rewrite	14124	848	139
Fixed	Original	12436	812	133

EDF と RMCL の GetResource() と EDF と ReleaseResource() でアスペクト指向プログラミングでカスタマイズした場合と，直接書き換えた場合で同じ実行時間になるのは，同じように最適化されたからである．

アスペクト指向プログラミングにより実装した場合の実行時間は直接書き換えて実装した場合に比べ ChainTask() で 15%以下の増大であり 10 μ*sec* 程度である．10 μ*sec* は固定優先度スケジューラのタスク起動でタスクスイッチがある場合とない場合の差と同程度で，タスクスイッチ 1 回程度分の時間である．また ChainTask() は指定のタスク起動をするために 1 度だけ呼ばれ，複数回呼ばれるものではないため，実用上問題ない程度と考える．

5.3 メモリ消費量の評価

アスペクト指向プログラミングによるメモリ消費量を評価する．**表 9** に，メモリ消費量を示す．

アスペクトを用いず人手でカスタマイズした OS と比較すると，コードサイズは 10〜15%程度の増加となる．これは，アスペクトを C 言語コードへ変換する際，処理内容を記述したアドバイスコードを関数に変換するためで，直接 OS のコードを書き換えるよりもメモリ消費量は大きくなる．増大量は 1.4 kB〜1.8 kB で，アプリケーションを含めたシステム全体のメモリ消費量を考えると（H8S/2638F の場合，コードが書き込まれる ROM 容量の 256 kB に対して 1%以下），許容範囲であると考える．

以上の評価から，アスペクトを用いたカスタマイズでの

メモリ消費量の増加は，実用上問題ない範囲と考える．

5.4 アスペクト指向プログラミングによる効果

ソースコードを直接書き換えてカスタマイズした場合は，スケジューラ以外のものも含めて同じファイルがオリジナルリアルタイム OS，EDF スケジューラ搭載リアルタイム OS，RMCL スケジューラ搭載リアルタイム OS の 3 つでき，オリジナルと EDF，オリジナルと RMCL，EDF と RMCL での共通部分が含まれてしまい，管理対象ファイルが 3 倍になり，さらに重複コードも含まれているため，構成管理が困難になる．アスペクトでカスタマイズすることによって，共通部分のないファイルのみとなり，スケジューラ間で重複するソースコードのデッドライン管理とリソース獲得状況管理を共通機能アスペクトとして分離させることにより保守性を向上させ，構成管理を容易にしている．

また，本研究のように，オープンソースのリアルタイム OS をカスタマイズする場合は，アスペクト指向プログラミングが有効である．直接修正すると，オリジナルのソースコードがバージョンアップされた場合への対応に工数がかかる．アスペクトで記述すれば差分が明らかで対応が容易になる．

6. 関連研究との比較

Afonso らは，リアルタイム OS を対象に同期（排他制御）やロギングにアスペクトを適用している [3]．しかしながら同期（排他制御）やロギングは，アプリケーションプログラムにおいてすでにアスペクト化がなされており，アスペクトによる分離記述が容易な処理である．

Park らは，ハードウェア依存部をリアルタイム OS から分離し，プログラミング言語非依存なアスペクト指向プログラミング環境 AOX を開発し，カスタマイズ可能なリアルタイム OS への適用を提案している [4]．また Beuche らは，アスペクト指向プログラミングを用いてハードウェア依存部を分離し，アーキテクチャ非依存なリアルタイム OS を実現する手法を提案している [6]．しかしリアルタイム OS のハードウェア依存部の置き換えは，これまでマクロによるコンパイルスイッチで広く実現されてきている．一般に，マクロで実現できることはアスペクトでも容易に実現できる．

Saito らは，既存のリアルタイム OS のソースコードを修正せずに，追加の機能として分散リアルタイム OS 環境に対応するためのアスペクトを提案した [5]．元の処理を残したまま，新しい処理を追加することは，ポイントカットによる織り込み箇所の指定ができれば，アスペクトでの実現はそれほど難しくはない．

Lohmann らは AUTOSAR OS の機能をアスペクトとしてモジュール化し，そこから必要な機能のみを取捨選択可能なリアルタイム OS ファミリーの提案をした [10], [11]．しかし，当初からアスペクト指向プログラミングによる実装を想定して設計されており（Lohmann らは Aspect-Aware Development と呼んでいる），設計時に想定していない機能を実現することは対象としていない．

以上の関連研究に対し，本研究では，ベースとしたリアルタイム OS の実装時には想定されていない仕様変更となるとともに，直接手作業で書き換える以外に実現する方法は知られていない，スケジューリングアルゴリズムやリソースアクセスプロトコルの変更を行っている．

これらの変更は，単純な処理の追加や関数の置き換えでは実現できず，本論文で述べたように，複数のアスペクトを組み合わせて 1 つの機能を実現するとともに，当初の設計では想定されていない処理への織り込みも行っている．たとえば，オリジナルの優先度上限プロトコルは，GetResource() と ReleaseResouce() 呼び出し時にのみ処理を行うことで実現できていたが，EDF や RMCL 向けのリソースアクセスプロトコルを実現するため，それらのみでなく，スケジューリングアルゴリズム変更のために新規に追加した処理にリソース管理のための処理を織り込んでいる．このように，従来研究で提案されていた手法をそのままあるいは単純に組み合わせて適用するのみでは，本研究の目的は達成できない．

7. おわりに

本論文では OSEK OS 仕様に基づく TOPPERS/ATK1 の固定優先度スケジューリングを，アスペクト指向プログラミングを用いて EDF または RMCL スケジューリングにカスタマイズする手法を提案した．これにより，既存のソースコードを直接修正せずに EDF または RMCL スケジューリングにカスタマイズできる．そして，実用上問題ないオーバヘッドとメモリ消費量で実現可能であることを示した．

今後の課題として，他のスケジューリングアルゴリズムやリソースアクセスプロトコルのアスペクトの提案，またはカスタマイズの対象機能を増やし，より多くのシステム向けに最適化できるリアルタイム OS を実現することがあげられる．

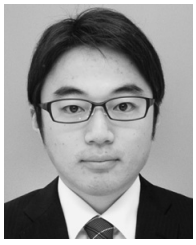
謝辞 本研究で使用した TOPPERS/ATK1 の開発者と ACC の開発者に感謝する．本研究は JSPS 科研費 2450046 および 15K00084 の助成を受けたものである．

参考文献

[1] Liu, C.L. and Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *J. ACM*, Vol.20, No.1, pp.46–61 (1973).

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-oriented programming, *ECOOP'97 – Object-Oriented*

Programming, Akşit, M. and Matsuoka, S. (Eds.), Lecture Notes in Computer Science, Vol.1241, pp.220–242, Springer Berlin Heidelberg (1997).
[3] Afonso, F., Silva, C., Montenegro, S. and Tavares, A.: Applying Aspects to a Real-time Embedded Operating System, *Proc. 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS '07*, New York, NY, USA, ACM (2007).
[4] Park, J. and Hong, S.: Building a Customizable Embedded Operating System with Fine-grained Joinpoints Using the AOX Programming Environment, *Proc. 2009 ACM Symposium on Applied Computing, SAC '09*, New York, NY, USA, ACM, pp.1952–1956 (2009).
[5] Saito, N., Yoo, M. and Yokoyama, T.: A distributed real-time operating system built with aspect-oriented programming for distributed embedded control systems, *2014 20th IEEE International Conference on, Parallel and Distributed Systems* (*ICPADS*), pp.436–443 (2014).
[6] Beuche, D., Fröhlich, A.A., Meyer, R., Papajewski, H., Schön, F., Schröder-Preikschat, W., Spinczyk, O. and Spinczyk, U.: On Architecture Transparency in Operating Systems, *Proc. 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System, EW 9*, New York, NY, USA, ACM, pp.147–152 (2000).
[7] Spinczyk, O., Gal, A. and Schröder-Preikschat, W.: AspectC++: An Aspect-oriented Extension to the C++ Programming Language, *Proc. 40th International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications, CRPIT '02*, Darlinghurst, Australia, Australia, Australian Computer Society, Inc., pp.53–60 (2002).
[8] Spinczyk, O. and Lohmann, D.: Using AOP to Develop Architectural-neutral Operating System Components, *Proc. 11th Workshop on ACM SIGOPS European Workshop, EW 11*, New York, NY, USA, ACM (2004).
[9] Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O. and Schröder-Preikschat, W.: A Quantitative Analysis of Aspects in the eCos Kernel, *SIGOPS Oper. Syst. Rev.*, Vol.40, No.4, pp.191–204 (2006).
[10] Lohmann, D., Hofer, W., Schröder-Preikschat, W. and Spinczyk, O.: Aspect-aware Operating System Development, *Proc. 10th International Conference on Aspect-oriented Software Development, AOSD '11*, New York, NY, USA, ACM, pp.69–80 (2011).
[11] Lohmann, D., Spinczyk, O., Hofer, W. and Schröder-Preikschat, W.: Transactions on Aspect-Oriented Software Development IX, Springer-Verlag, Berlin, Heidelberg, chapter The Aspect-aware Design and Implementation of the CiAO Operating-system Family, pp.168–215 (2012).
[12] Spliet, R., Vanga, M., Brandenburg, B.B. and Dziadek, S.: Fast on Average, Predictable in the Worst Case: Exploring Real-Time Futexes in LITMUSRT, *Real-Time Systems Symposium* (*RTSS*), *2014 IEEE*, pp.96–105 (2014).
[13] OSEK/VDX, Operating System, Version 2.2.3 (2005).
[14] TOPPERS Project, available from ⟨http://www.toppers.jp/⟩.
[15] 加藤真平，山崎信行：Linux カーネル用リアルタイムスケジューリングモジュール，情報処理学会論文誌コンピューティングシステム（ACS），Vol.2, No.1, pp.75–86 (2009).
[16] Gong, M.Z.C. and Jacobsen, H.A.: Systems Development with AspeCt-oriented C (ACC), *Connections 2007* (*ECE Graduate Symposium, University of Toronto*) Talk 5.6 (2007).
[17] AsceCt-oriented C, available from ⟨https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc/⟩.
[18] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *Proc. 15th European Conference on Object-Oriented Programming, ECOOP '01*, London, UK, UK, pp.327–353, Springer-Verlag (2001).
[19] Kang, K., Cohen, S., Hess, J., Novak, W. and Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University (1990).
[20] OSEK VDX, System Generation OIL: OSEK Impremention Language Version 2.5.
[21] Baker, T.P.: Stack-based Scheduling for Realtime Processes, *Real-Time Syst.*, Vol.3, No.1, pp.67–99 (1991).



原田 祐輔 （学生会員）

2015 年東京都市大学知識工学部情報科学科卒業．現在，同大学大学院工学研究科情報工学専攻修士課程在学中．組込みシステム向けソフトウェアの研究に従事．IEEE 会員．

阿部 一樹

2011 年東京都市大学知識工学部情報科学科卒業．2013 年同大学大学院工学研究科情報工学専攻修士課程修了．同年株式会社 PFU 入社．インフラ構築業務に従事．

兪 明連 （正会員）

1994 年安東国立大学校工学部コンピュータ工学科卒業．1996 年浦項工科大学大学院情報通信専攻修士課程修了．同年安東情報大学講師．2002 年嶺南大学コンピュータ工学専攻博士課程修了．2006 年早稲田大学大学院情報生産システム研究科博士後期課程修了．2007 年武蔵工業大学．現在，東京都市大学准教授．スケジューリング理論の研究に従事．博士（工学）．電子情報通信学会，IEEE 各会員．

横山 孝典 （正会員）

1981 年東北大学工学部通信工学科卒業．1983 年同大学大学院工学研究科電気及通信工学専攻修士課程修了．同年（株）日立製作所入社．1987 年から 1990 年まで（財）新世代コンピュータ技術開発機構出向．2004 年武蔵工業大学．現在，東京都市大学教授．組込みシステム，分散システム，ソフトウェア工学等の研究に従事．博士（情報科学）．電子情報通信学会，IEEE，ACM 各会員．