Vol.2013-SLDM-160 No.22
Vol.2013-EMB-28 No.22
2013/3/14

アスペクト指向による組込みOSスケジューラのカスタマイズ

阿部 一樹[1,a)]　安島 光紀[†1]　兪 明連[1,b)]　横山 孝典[1,c)]

概要：一般に組込みシステム上のアプリケーションは複数のタスクから構成されるが，アプリケーションによって異なるタスクスケジューリングアルゴリズムが要求される場合がある．しかし多くの組込み OS は，固定優先度スケジューリングしか提供していない．そのため，スケジューリングアルゴリズムの選択が可能な組込み OS が求められる．本研究では，アスペクト指向プログラミングにより，OS のソースコードを直接書き換えることなく，組込み OS のスケジューラをアプリケーションに応じてカスタマイズする手法を提案する．自動車制御向け組込み OS である OSEK OS を対象に，固定優先度のスケジューラを EDF に切り替えるとともに，EDF に対応した排他制御を実現するために，排他制御のプロトコルを変更するアスペクトを開発した．そして，実際にそれらのアスペクトを適用した OS の評価を行い，実用上問題の無いオーバヘッドで実現可能であることを確認した．

キーワード：組込み OS，リアルタイム OS，スケジューリングアルゴリズム，アスペクト指向プログラミング

1. はじめに

組込みシステムの多様化にともない，様々な用途に対応した組込みアプリケーションが開発されている．一般に組み込みアプリケーションは，並行処理される複数のタスクで構成され，それらのタスクは組込み OS のスケジューラによってスケジューリングされる．組込み OS のほとんどは固定優先度スケジューリングのみをサポートしている．しかし，RM(Rate Monotonic) スケジューリングのような固定優先度スケジューリングは最適とは限らない．アプリケーションによっては，最適スケジューリングアルゴリズムである EDF(Earliest Deadline First) スケジューリング [1][2] が求められる場合がある．

しかし，1 つの組込み OS のスケジューラに複数のスケジューリングアルゴリズムを実装すると，組込みシステムの厳しいリソース制約を満たせなくなる可能性がある．そこで，目的に応じて組込み OS のスケジューラを静的にカスタマイズする手法が求められている．

[1] 情報処理学会
IPSJ, Chiyoda, Tokyo 101–0062, Japan
[†1] 現在，東京都市大学
Presently with Tokyo City University, Tokyo 158–8557, Japan
[a)] g1181501@tcu.ac.jp
[b)] yoo@tcu.ac.jp
[c)] yokoyama@tcu.ac.jp

システムの機能を静的に変更する手法として，アスペクト指向プログラミング [3] が注目されている．アスペクト指向プログラミングでは，システム中に横断的に散在する関心事を，アスペクトとしてモジュール化して記述できる．

これまでにアスペクト指向プログラミングを，OS の構築に応用する研究がいくつかなされている [4][5][6]．例えば Park らは，ファイルシステムとメモリマッピングシステムを対象にカスタマイズを行った [7]．また Lahmann らは，アスペクト指向に基づいた組込み OS の設計を行い，コンフィギュレーションが容易な組込み OS を提案している [8]．しかしこれらの研究は，タスクスケジューリングアルゴリズムの切り替えは対象にしていない．組込み OS のスケジューラをアスペクトによって切り替える提案 [9] はあるが，実装は今後の課題としている．

本研究の目的は，アプリケーションに応じて，組込み OS のスケジューリングアルゴリズムを静的に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を採用している OSEK OS[10] を対象に，そのスケジューラを EDF スケジューリングアルゴリズムに置き換えるアスペクトを提案する．これらのアスペクトによってカスタマイズした OS の実行性能やリソース消費量を評価し，実用性の検討を行う．

Vol.2013-SLDM-160 No.22
Vol.2013-EMB-28 No.22
2013/3/14

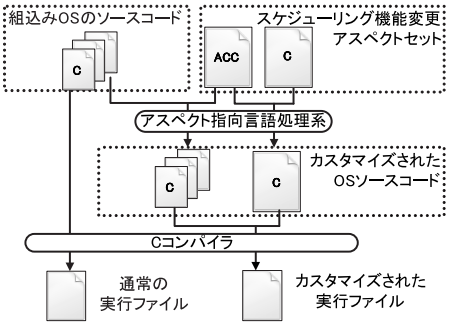

図 1 アスペクトによるカスタマイズの流れ
Fig. 1 Flow of Customization

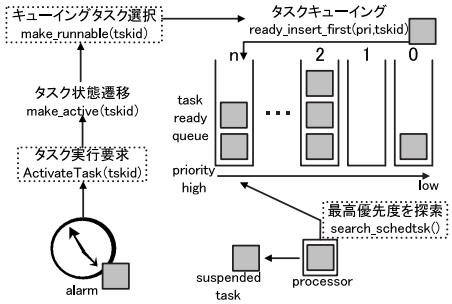

図 2 TOPPERS/OSEK のスケジューラ
Fig. 2 Scheduler of TOPPERS/OSEK

2. カスタマイズの概要

2.1 OSEK OS

本研究のカスタマイズ対象 OS は，自動車制御システム用組込み OS の仕様である OSEK OS に準拠した，TOPPERS/OSEK カーネル [11] とする．OSEK OS は，固定優先度のタスクスケジューリングアルゴリズムを採用している．タスクはアラームオブジェクトによって周期的に起動できる．また，リソースオブジェクトをタスクが獲得することで，優先度を一時的に引き上げ，排他制御機能が提供されている．

タスクやアラーム, リソース等のオブジェクトは, コンフィギュレーションファイル内で宣言することで静的に生成する．コンフィギュレーションファイルは OIL(OSEK Implementation Language) という専用言語で記述し, SG(System Generator) を用いてソースコードに変換する．

本研究では，OSEK OS の固定優先度スケジューリングを，EDF スケジューリングに置き換えるアスペクトを開発する．

2.2 ACC

本研究では, C 言語で実装されている TOPPERS/OSEK をカスタマイズするため, アスペクト指向言語として ACC(AspeCt-oriented C)[12] を用いる．ACC は，GCC をベースにアスペクト指向に拡張した言語である．ACC では, call, execution, get などのポイントカットと, before, `after` , `around` などのアドバイスが利用できる．

2.3 カスタマイズの流れ

図 1 に，アスペクトによるカスタマイズの流れを示す. この図は，スケジューリングアルゴリズムを変更するアスペクトのファイルセットを用意し，TOPPERS/OSEK に適用する際の手順を表している．

アスペクト指向言語処理系により，組込み OS のソースコードにアスペクトを織り込むことで，スケジューリングアルゴリズムが変更された OS の C ソースコードが生成される．生成されたソースコードをコンパイルすることで，変更が適用された OS の実行ファイルを作成できる．この手法を用いると，組込み OS の機能変更を，組込み OS 自体のソースコードを書き換えることなく，アスペクトの織り込みによって実現することができる．

ソースコードを直接書き換える手法と比べ，変更部分が分離されているため保守性に優れる．また，OS の基幹部分を変更せずに部分的な変更を適用することができるため，OS のバージョン管理が容易になる．

3. スケジューラのカスタマイズ

3.1 OSEK OS のスケジューリング

図 2 に，TOPPERS/OSEK のタスクスケジューリング機構の概略を示す．この図は，アラームが指定タスクの起動要求を周期的に発行し，タスクが実行状態になるまでの流れを表している．タスク起動要求システムコール `ActivateTask` が呼び出されると，タスクの状態を遷移する関数 `make_active` が実行され，タスクが実行状態になる．その後実行される `make_runnable` では，実行状態になったタスクと現在実行中のタスクとの優先度を比較し，低い方を実行待ちタスクキューに挿入する．キューイングを行う関数は `ready_insert_first` で，`make_runnable` から呼び出される．タスクは，それぞれの優先度に対応したタスクキューに格納され，実行を待つ．

ディスパッチが発生する際は，次に実行するタスクの探索を行う関数 `search_schedtsk` が実行される．この関数は，タスクキューを優先度の高いものから順に走査し，タスクが入っているキューのうち最も優先度の高いものを探索する．ディスパッチ時は，そのキューの先頭のタスクを取り出し，次に実行するタスクに指定する．

3.2 アスペクト

本研究で提案する手法では，タスクの実行要求からディスパッチまでの一連の流れで呼び出される関数を，アスペクトによって変更する．

デッドラインを更新する関数を呼び出すアスペクトを図 3 に示す．本アスペクトは，`ActivateTask` に `before` アド

```
// デッドラインを更新する
before(TaskType tskid) : execution(StatusType ActivateTask(TaskType))
                         && args(tskid) {
    if(callevel == TCL_ISR2){
        update_deadline(tskid);
    }
}
```

図 3 デッドライン更新関数を呼び出すアスペクト
Fig. 3 Aspect for Calling Deadline Update Function

```
StatusType around(TaskType tskid):  execution($ ActivateTask(TaskType)
                                    && args(tskid)){
        //デッドラインの遅いタスクをキューイングするように指定する関数
        return( make_runnable_edf(tskid) );
}
```

図 4 タスクキューイング関数を置き換えるアスペクト
Fig. 4 Aspect for Replacing Queueing Operation

```
around():  execution($ search_schedtsk(TaskType)){
        //タスクキューの先頭を参照する関数
        search_schedtsk_edf();
}
```

図 5 実行タスク探索関数を置き換えるアスペクト
Fig. 5 Aspect to Extract a Task to be Executed

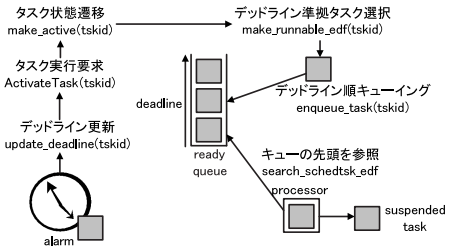

図 6 カスタマイズ後のスケジューラ
Fig. 6 Customized Scheduler

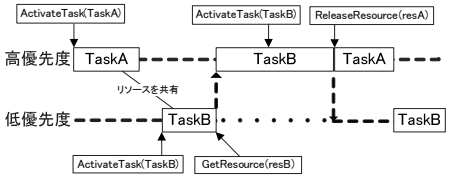

図 7 TOPPERS/OSEK の排他制御
Fig. 7 Mutual Exclusion of TOPPERS/OSEK

バイスで，デッドライン更新関数 update_deadline の呼び出しを記述する．update_deadline は，システムの現在時刻に相対デッドラインを追加することで，タスクの絶対デッドラインを算出する処理を行う関数である．これにより，タスクの起動要求があると，まずそのタスクのデッドラインを最新のものに更新するようになる．なお，以下単にデッドラインと呼ぶ場合は，相対デッドラインではなく，絶対デッドラインを指すものとする．

タスクキューイング判定をデッドラインにより行うアスペクトを図 4 に示す．本アスペクトは make_runnable に around アドバイスで，関数 make_runnable_edf を記述している．この関数は，実行状態になったタスクを受け取り，実行中のタスクとデッドラインを比較して，より遅い方をタスクキューに挿入する，という処理を行う関数である．make_runnable に代わりキューイングを行う関数だが，その条件をデッドラインに変更している．

なお，EDF スケジューリングで用いるタスクキューは，固定優先度と違い単一のキューである．格納するタスクはデッドライン順に整列する．このキューイングおよび整列を行う関数は enqueue_task であり，ready_insert_first に代わって make_runnable_edf から呼び出される．この一連の変更により，タスクの起動要求システムコールを EDF スケジューリングに対応させることができる．

単一タスクキューに対応させるアスペクトを図 5 に示す．本アスペクトは search_schedtsk に around アドバイスで，関数 search_schedtsk_edf を記述している．この関数は，EDF スケジューリングのキューイングに対応し，単一キューの先頭のみを参照する．これにより，ディスパッチ時には最もデッドラインの近いタスクを選択するようになる．

3.3 変更後のスケジューラ

カスタマイズ後のスケジューリング機構は図 6 のようになる．EDF スケジューラは，タスクの起動要求を受けた際に，静的に定義されたタスクの相対デッドラインを参照し，絶対デッドラインを算出する．アラームオブジェクトから ActivateTask が発行されると，現在時刻と相対デッドラインから絶対デッドラインを算出し，実行状態に遷移する．実行中のタスクのほうがデッドラインが早い場合は，単一のタスクキューに挿入し，デッドライン順に整列する．新たなタスクを実行する際は，タスクキューの先頭にあるデッドラインの近いタスクから順にディスパッチしていく．

4. 排他制御のカスタマイズ

4.1 OSEK OS の排他制御

OSEK OS のリソース機能による排他制御は，優先度上限プロトコル，正確にはスタックリソースポリシー [13] を採用している．TOPPERS/OSEK の排他制御機能の図 7 を用いて説明する．この図は，高優先度タスク TaskA と低優先度タスク TaskB が共有するリソース res1 を，TaskB が獲得し，解放する際の流れを表している．タスクがリソース獲得システムコール GetResource を呼び出すと，そのタスクの優先度がそのリソースを共有するタスクのうち最高優先度のタスクと同じ優先度まで引き上げられる．ReleaseResource を呼び出すとリソースを解放し，元の優先度に戻る．

ところが，EDF スケジューリングでは，上限優先度を静的に定めることができず，OSEK OS の優先度上限プロトコルでは対応できない．そこで EDF スケジューリングのリソース機能は，リソースを獲得しているタスクが，その

```
after(TaskType tskid) :
execution(void update_deadline(TaskType)) && args(tskid) {
    //優先度継承を行う関数
    ref_deadline(tskid);
}
```

図 8 優先度継承機能を追加するアスペクト
Fig. 8 Aspect for Priority Inheritance

```
タスクのリソース共有状況ビットマップ
tinib_resource[0] … 0000010000001001  (a)
tinib_resource[1] … 0000000010011000  (b)
                  ⋮
                  ⋮
                  ⋮

リソースの獲得状況ビットマップ
rescb_resource    … 0000000000001010  (c)

タスク0の起動要求
        ↓
タスク0のビットマップと，リソース獲得状況ビットマップを論理和
tinib_resource[0] & rescb_resource = 0000000000001000 (d)
```

図 9 ビットマップの構造
Fig. 9 Data Structure of Bitmaps

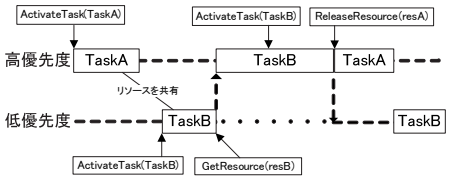

図 10 EDF スケジューリングに対応した排他制御
Fig. 10 Matual Exclusion for EDF Scheduling

```
TASK Task1 {
        AUTOSTART = TRUE {
                APPMODE = AppMode2;
        };
        PRIORITY = 4;
        STACKSIZE = 0x00a0;
        ACTIVATION = 8;
        SCHEDULE = FULL;
        RESOURCE = TskLevelRes;
        RESOURCE = IntLevelRes;
};
```

(a) タスクの定義

```
ALARM ActTskArm {
        COUNTER = SysTimerCnt;
        ACTION  = ACTIVATETASK {
                TASK = Task1;
        };
        AUTOSTART = TRUE {
                APPMODE = AppMode1;

                ALARMTIME = 100;
                CYCLETIME = 100;
        };
};
```

(b)アラームの定義

図 11 オブジェクトの OIL 記述
Fig. 11 OIL Description of Objects

リソースを共有するタスクのうち，その時点で起動されている最もデッドラインが近いタスクの優先度を継承するプロトコルを実現する．

4.2 切り替えを行うアスペクト

リソース機能のシステムコールの動作を変更するアスペクトを図 8 に示す．デッドライン更新関数 `update_deadline` に after アドバイスを用いて織り込む関数 `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させる処理も行う．

他のタスクがリソースを獲得しているかを調べるために，リソース獲得状況をビットマップに記憶する．ビットマップの構成を図 9 に示す．ビットマップは，各タスクがどのリソースを共有しているかを表す静的ビットマップ `tinib_resource[タスク ID]` と，現在どのリソースが獲得されているかを表す動的ビットマップ `rescb_bitmap` の 2 種類がある．ビット一（LSB から何ビット目か）がそのままリソース ID として扱われる．

図 9 の例を説明する．タスク ID0 のタスク（以下「タスク 0」）がリソース ID3 のリソース（以下「リソース 3」）を共有する場合，ビットマップ `tinib_resource[0]` の LSB から 3 ビット目が 1 になる（図中 (a)）．また，このリソースが獲得された時は，`rescb_bitmap` の 3 ビット目が 1 になる（図中 (c)）．関数 `ref_deadline` 内でリソースの共有状況を確認する際には，`tinib_resource[0]` と `rescb_bitmap` の論理積をとる（図中 (d)）．3 ビット目が 1 となるため，リソース 3 が獲得されていることが分かる．リソース 3 を獲得しているタスクのデッドラインを参照し，それがタスク 0 のデッドラインより遅い場合，そのタスクにタスク 0 のデッドラインを継承させる．

以上のような処理をタスク起動要求時に行うことで，リソースに関わるタスクのデッドラインを常に確認し，EDF スケジューリングの排他制御を実現できる．

また，リソース獲得状況ビットマップを常に更新するため，`GetResource` を `GetResource_edf` に，`ReleaseResource` を `ReleaseResource` に，それぞれアスペクトで置き換える．これらの関数では，獲得および解放されたリソース ID に応じて，`rescb_bitmap` を更新する機能をもっている．また，リソース獲得時に，そのタスクのデッドラインを保存し，リソース解放時に元に戻す．

4.3 EDF 環境下のプロトコル

カスタマイズ後の排他制御プロトコルの動作を図 10 を用いて説明する．TaskB がリソース res1 を獲得中に，より優先度の高い（デッドラインが近い）TaskA を起動されると，TaskB は TaskA の優先度（デッドライン）を継承することで，TaskA の実行開始は，TaskB がリソース res1 を解放するまで遅延する．

5. コンフィギュレーション

5.1 OSEK OS のコンフィギュレーション

OSEK OS では，タスクやアラーム，リソース等は，OIL により静的に宣言する．タスクの定義では，優先度，タスクが共有するリソースオブジェクトなどを記述する．アラームオブジェクトでは，アラームの開始時刻や周期，時刻到達時に行う処理などを記述する．図 11 に，OIL で記述したタスク定義の例を示す．図 11(b) の例は，(a) で定義したタスク Task1 の起動要求を周期的に発行する記述をしている．

表 1　追加された属性
Table 1　Added Elements

属性名	属性の種類	用途
tinib_deadline	定数	タスクの相対デッドラインを表す定数
tcb_deadline	変数	タスクの絶対デッドラインを格納する変数
tinib_resource	定数	タスクのリソース共有状況を表すビットマップ
rescb_bitmap	変数	リソース獲得状況を表すビットマップ

```
TASK Task1 {
        AUTOSTART = TRUE {
                APPMODE = AppMode2;
        };
        DEADLINE = 500;
        STACKSIZE = 0x00a0;
        ACTIVATION = 8;
        SCHEDULE = FULL;
        RESOURCE = TskLevelRes;
        RESOURCE = IntLevelRes;
};
```

図 12　タスクの拡張 OIL 定義
Fig. 12　Extended OIL Description of Task Object

5.2 EDF スケジューリングのための OIL の拡張とコンフィギュレーション

EDF スケジューリングを実現するため，コンフィギュレーションファイル内に追加した属性の例を表 1 に示す．tinib_deadline と tcb_deadline は，絶対デッドラインの算出および格納に用いる．tinib_resource および rescb_bitmap は，リソース数に応じた配列として定義する．論理和の計算をより効率的に行うため，変数の型は CPU に応じて決める．

EDF スケジューリングに対応するため，既存の OIL を拡張してタスクの相対デッドラインを定義可能にする．図 12 に，拡張 OIL でのタスク定義の例を示す．相対デッドラインは定数として出力し，絶対デッドラインの算出に用いる．また，出力するソースコードには，絶対デッドラインを格納するための配列や，タスクが共有しているリソースを参照するためのビットマップなどを追加する．

6. 評価および考察

本研究で開発した OS を評価するため，評価用ボード上での性能，およびリソース消費量を計測した．比較のため，アスペクトを用いずに直接ソースコードを書き換えて同じカスタマイズを施した組込み OS を実装し，同様の計測を行った．

6.1 評価環境

本研究で評価に用いた環境は表 2 に示すとおりである．表中の ACC 用 C コンパイラとは，アスペクト処理系が織り込みを行う際に呼び出す C コンパイラである．ACC の織り込みは GCC 環境 [14] に依存しているため，実装環境によらず GCC を用いる必要がある．織り込み後のファイルから実際に実行ファイルを出力するコンパイラは NC30 である．

表 2　評価環境
Table 2　Experimental Environment

種類	環境名
評価用ボード	OAKS16-MINI（ROM64KB RAM2KB）
プロセッサ	M16C/26（20MHz）
OS	toppers/OSEK ver.1.1
アスペクト処理系	ACC ver.0.9
ACC 用 C コンパイラ	GCC ver.4.2.2
クロスコンパイラ	NC30 ver.5.10
アセンブラ	AS30 ver.4.20
リンカ	LN30 ver.4.10

表 3　ActivateTask 実行時間の評価
Table 3　Execution Time of *ActivateTask*

（単位：μ s）	タスク切替なし	タスク切替あり	デッドライン継承あり
アスペクトを用いて EDF 化した OS	49.10	52.95	+17.66
アスペクトを用いず EDF 化した OS	48.20	49.82	+16.96
固定優先度 OS	25.40	26.50	（なし）

表 4　リソース機能システムコールの実行時間の評価
Table 4　Execution Times of System Calls for Resource Control

（単位：μ s）	GetResource	ReleaseResource
アスペクトを用いて EDF 化した OS	14.92	19.37
アスペクトを用いず EDF 化した OS	13.37	17.84
固定優先度 OS	10.64	12.11

6.2 実行時間の評価

表 3 および表 4 に，EDF スケジューリングに変更した 2 つの OS の各システムコールの実行時間を比較した結果を示す．参考のため，固定優先度の場合の値も示す．時間の計測には，タスクとリソースがそれぞれ 10 個ずつある評価用アプリケーションを用いた．各システムコールの発行から終了までにかかる時間を 50 回ずつ計測し，その平均を求めた．アスペクトを用いた場合は，直接ソースコードを書き換えた場合と比較して，実行時間は 1〜10 % 程度増加している．これはアスペクト自体のオーバヘッドを意味しているが，十分小さい値となっている．ref_deadline 内でデッドラインの継承が行なわれる場合は実行時間が増大するが，リソース機能のシステムコールの実行時間と同程度であり，実用上許容できる範囲ではないかと考えている．

6.3 リソース消費量の評価

6.3.1 OS 部分のリソース消費量

表 5 に，EDF スケジューリングに対応した組込み OS の

表 5　リソース消費量の評価
Table 5　Memory Consumption of Customized Scheduler

（単位：Byte）	RAM 領域	ROM 領域	コードサイズ
アスペクトを用いて EDF 化した OS	442	99	9881
アスペクトを用いず EDF 化した OS	442	99	9767
固定優先度 OS	418	74	9085

表 6　追加された属性のリソース消費量
Table 6　Memory Consumption of Added Elements

属性名	属性の種類	リソース消費量
tinib_deadline	定数	T × 32bit
tinib_almid	定数	T × 8bit
tinib_resource	定数	T × (R/16+1) × 16bit
tcb_deadline	変数	T × 32bit
tcb_prev	変数	T × 8bit
rescb_bitmap	変数	(R/16+1) × 16bit
rescb_usingtask	変数	R × 8bit
rescb_prevdl	変数	R × 32bit
T：タスクオブジェクト数　R：リソースオブジェクト数		

リソース消費量の評価結果を示す．これは，OS のアプリケーション部を空の main 関数のみにして，OS 部分のみで実行ファイルを作成した時の値である．アスペクトを用いない場合と比べてメモリ消費量は 1% 程度の増加にとどまっており，無視しても問題ない程度の差といえる．

6.3.2　コンフィギュレーションファイル内の追加属性

コンフィギュレーションファイルに追加した属性のリソース消費量を，表 6 に示す．ビットマップ配列 `tinib_resource` および `rescb_bitmap` のデータ型は，論理積の計算をより効率的に行うため，CPU に応じて決める．今回は 16 ビットの CPU を採用したため，配列の 1 つの要素を 16 ビットとして定義した．

表 1 の変数および定数は全て，アスペクトを用いずにカスタマイズした場合にも必要になる属性である．また，アスペクトを用いてカスタマイズしたことが原因で，新たな属性が必要になることはない．

以上の評価から，アスペクトを用いたカスタマイズでのリソース消費量の増加は，実用上問題の無い範囲と言える．

7.　まとめ

アスペクト指向プログラミングを用いて，組込み OS の固定優先度のスケジューラを EDF 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，固定優先度と EDF のスケジューリングを，アプリケーションに応じて静的に変更することが可能になる．また，アスペクトを用いてカスタマイズした OS を，アスペクトを用いずにカスタマイズした OS と比較し，評価した結果，リソース消費量，実行速度ともに，実用上問題のない値であるとわかった．

今後の課題として，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組込み OS を実現することが挙げられる．

謝辞　本研究のベースとした TOPPERS/OSEK カーネルの開発者に感謝する．本研究の一部は JSPS 科研費 24500046 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, Journal of the ACM, Vol.20, No.1, pp.46-61, 1973.
[2] Dertouzos, M. L., Control Robotics: The Procedural Control of Physical Processes, Proceedings of IFIP Congress 1974, pp.807-813, 1974
[3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes,Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Proc. 11th European Conference on Object-Oriented Programming, pp. 220-242, 1997. International Conference on Aspect-Oriented Software Development 2011, pp.69-80, 2011.
[4] Beuche, D., Frohlich, A. A., Reinhard, M., Papajewski, H., Schon, F., Schroder-Preikschat, W., Spinczyk, O. and Spinczyk, U., On Architecture Transparency in Operating Systems, Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating system, pp. 147-152, 2000.
[5] Coady, Y., Kiczales, G., Feeley, M. and Smolyn G., Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code, Proceedings of the 8th European Software Engineering Conference, pp.88-98, 2001.
[6] Afonso, F., Silva, C., Montenegro, S. and Tavares, A., Applying Aspects to a Real-Time Embedded Operating System, Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software, Article No.1, 2007.
[7] Park J and Hong S , Customizing Real-Time Operating Systems with Aspect-Oriented Programming Framework , Proceedings ofSoC Design Conference, 2003.
[8] Lohmann, D., Hofer, W., Schroder-Preikschat, W. and Spinczyk, O.,Aspect-Aware Operating System Development, Proceedings of the 10th
[9] Hatun, K., Bockisch, C., Sozer, H. and Aksit, M., A Feature Model and Development Approach for Schedulers, Proceedings of the 1st Workshopon Modularity in Systems Software, pp.1-5, 2011.
[10] OSEK/VDX : *http://www.osek-vdx.org/*
[11] TOPPERS Project Inc.:toppers/OSEK kernel, *http://www.toppers.jp/osek-os.html*,2006.
[12] AspeCt-oriented C: *http://research.msrg.utoronto.ca/ACC*,2006.
[13] Baker, T. P. "A Stack-Based Resource Allocation Policy for Realtime Processes". Proceedings of IEEE Real-Time Systems Symposium, PP. 191-200. 1990.
[14] GNU Project , *http://www.gnu.org/*