情報処理学会論文誌　Vol. 49　No. 10　3508–3519 (Oct. 2008)

リアルタイムアプリケーション統合のための柔軟なスケジューリングフレームワーク

松　原　　　豊[†1]　本　田　晋　也[†2]
冨　山　宏　之[†1]　高　田　広　章[†1,†2]

分散制御システムにおいて，各システムで単独動作するアプリケーションを 1 つのシステム上に統合する場合，アプリケーションのプロセッサ利用率やリアルタイム性を保証するための統合スケジューリングアルゴリズムが必要となる．その一方で，アプリケーションに要求される時間要件（リアルタイム，非リアルタイム）や，統合段階で既知であるパラメータ（アプリケーションに所属するタスクのリリース時刻，最悪実行時間，デッドラインなど）はそれぞれ異なる場合が多く，すべてのアプリケーションに対して同一の統合スケジューリングアルゴリズムを適用することは困難である．本論文では，アプリケーションの時間要件や統合段階で既知であるパラメータに応じて，アプリケーションごとに適切な統合スケジューリングアルゴリズムを選択できるスケジューリングフレームワークを提案する．提案するフレームワークのプロトタイプを実装し，評価した結果，許容できる程度のオーバヘッドで実現できることを確認した．

A Flexible Scheduling Framework for Integration of Real-time Applications

Yutaka Matsubara,[†1] Shinya Honda,[†2]
Hiroyuki Tomiyama[†1] and Hiroaki Takada[†1,†2]

In a distributed computer control system, a scheduling algorithm for temporal protection is required to isolate processing time among real-time applications and satisfy timing constraints of each application for integration of real-time applications. However, timing requirements and timing parameters (release time, worst-case execution time and deadline) known at integration phase vary according to applications. Therefore it is difficult to schedule diverse applications by a single scheduling algorithm. In this paper, we propose a flexible scheduling framework that allows each application to be applied the optimal scheduling algorithm. We developed a prototype of the framework based on an open-source real-time OS. The results of evaluation for the prototype indicate that the framework can be implemented with a little overhead.

1. はじめに

近年，代表的な分散制御システムの 1 つである自動車制御システムの開発では，その高性能化・複雑化により，ECU（Electronic Control Unit）と呼ばれる電子制御ユニットが増加し，コストの増加，設置スペースの不足が大きな問題となっている．そこで，システムに搭載される ECU 数を削減する手法として，複数の ECU でそれぞれ動作しているアプリケーションを単一の ECU 上に統合して動作させる手法が検討されている．たとえば，欧州の自動車メーカ，自動車部品メーカ，ツールベンダなどが中心となって設立した標準化活動団体である AUTOSAR（AUTomotive Open System ARchitecture）[1)]や，国内の自動車メーカが中心となって設立した JASPAR[2)] では，ECU の共通ソフトウェアプラットフォームの標準仕様を策定し，自動車制御アプリケーションの統合を目指している．

単独で，時間制約を満たして動作するリアルタイムアプリケーションを，1 つのプロセッサで複数動作させる（これをリアルタイムアプリケーション統合と呼ぶ）場合，あるアプリケーションが想定した以上のプロセッサ時間を使ったために，他のアプリケーションが時間制約を満たせなくなることを防ぐ必要がある．また，統合前に時間制約を満たして動作するアプリケーションが，統合することで時間的な振舞いが変化し，その結果時間制約を満たせなくなる場合があると，統合に際してアプリケーションの再検証（場合によっては再設計も）が必要となり，統合を進めることが困難になる．我々は，統合の前後でアプリケーションの振舞いが変化する要因を 3 つに整理し，アプリケーションの中に統合前と統合後で要求する処理量が変化するタスク（これを QoS 制御タスクと呼ぶ）が存在することが要因の 1 つになることを示した．さらに，QoS 制御タスクが存在する状況においても，統合前に時間制約を満たすリアルタイムアプリケーションが統合後にも時間制約を満たすこと（これを時間保護と呼ぶ）を実現する時間保護アルゴリズムを提案した[3)]．

†1 名古屋大学大学院情報科学研究科情報システム学専攻
Department of Information Engineering, Graduate School of Information Science, Nagoya University
†2 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
Center for Embedded Computing Systems, Nagoya University

© 2008 Information Processing Society of Japan

しかしながら，実際の自動車制御システムの開発では，要求される時間要件が異なるアプリケーションを統合する場合がある．たとえば，リアルタイム性を要求される制御アプリケーションと，バックグラウンドで動作してシステムの故障を検出する非リアルタイムアプリケーションを統合する状況が考えられる．このような場合においては，適切なプロセッサ時間の保護機能（プロセッサ利用率保護，時間保護）はアプリケーションごとに異なる．

また，複数のベンダのアプリケーションを統合する場合や，統合の前後でアプリケーションの動作環境が異なる場合には，統合段階で既知であるアプリケーションのパラメータ（アプリケーションに含まれるタスクのリリース時刻，デッドライン，最悪実行時間（WCET），平均実行時間，クリティカルセクションの実行時間など）は，異なる場合が多い．このような状況では，適用できる統合スケジューリングアルゴリズムも異なるため，統合するすべてのアプリケーションに対して，同一のスケジューリングアルゴリズムを適用することは困難である．

本論文では，アプリケーションに要求される時間要件と，統合段階で既知であるパラメータから，アプリケーションごとに適切なスケジューリングアルゴリズムを選択できるスケジューリングフレームワークを提案する．

提案フレームワークのスケジューラは，ローカルスケジューラとグローバルスケジューラを，2 階層に配置した階層型スケジューラである．アプリケーションごとにローカルスケジューラが対応し，アプリケーションの時間要件と統合段階で既知であるパラメータから，適切なプロセッサ時間保護機能を選択できる．たとえば，リアルタイムアプリケーションに対しては時間保護機能を，非リアルタイムアプリケーションに対してはプロセッサ利用率の保護機能を選択できる．提案フレームワークをリアルタイム OS に適用することで，時間要件の異なるアプリケーションを単一のプロセッサ上に統合できる．

以下，本論文の構成について述べる．まず 2 章で，アプリケーションに要求される時間要件と，プロセッサ時間の保護機能を整理する．さらに，整理した保護機能を実現できる統合スケジューリングアルゴリズムと，適用するために必要なパラメータを整理し，提案するフレームワークにおける保護機能の選択方針を述べる．3 章では，提案するフレームワークの設計について詳細に述べる．4 章では，プロトタイプの実装について述べ，メモリ消費量とタスク切替え処理時間などの観点で評価する．5 章では，リアルタイムアプリケーション統合に関連する従来研究について述べ，6 章で結論と今後の課題について述べる．

2. リアルタイムアプリケーションの統合と保護機能

2.1 状況設定

本論文では，**図 1** に示すように，複数の個別プロセッサ上で動作するアプリケーションを，単一の統合プロセッサ上に統合する状況を想定する．ここで，個別プロセッサとは，1 つのアプリケーションのみを動作させるための統合前のプロセッサのことを，統合プロセッサとは，複数のアプリケーションを動作させるための統合後のプロセッサのことをいう．通常，統合プロセッサは，個別プロセッサに比べて高い処理性能を持つ．プロセッサ性能と処理量の関係は単純化し，単位時間あたりの処理量はプロセッサの性能に比例するものとする．たとえば，性能 1 のプロセッサで，2 単位時間で実行を完了するタスクは，性能 2 のプロセッサでは，1 単位時間で実行を完了できるものとする．

2.2 プロセッサ時間の保護機能

統合対象のアプリケーションは，リアルタイム性を要求されるタスクが含まれるリアルタイムアプリケーションと，リアルタイム性を要求されないタスクのみで構成される非リアルタイムアプリケーションに分類できる．要求される時間要件の異なるアプリケーションを統合する場合，アプリケーションごとに，適切なプロセッサ時間の保護機能を選択して適用する必要がある．本論文では，アプリケーション統合に必要なプロセッサ時間の保護機能として，非リアルタイムアプリケーションに適するプロセッサ利用率保護と，リアルタイムアプリケーションに適する時間保護の 2 つを用意する．

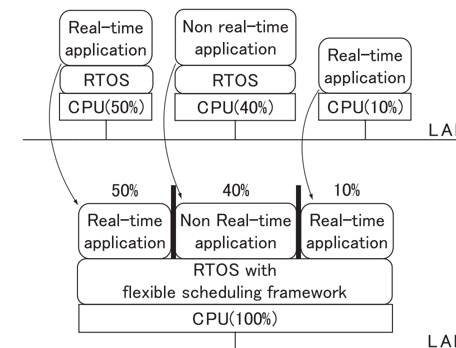

図 1　アプリケーション統合
Fig. 1　Integration of applications on a processor.

© 2008 Information Processing Society of Japan

プロセッサ利用率保護

アプリケーションに所属するタスクが，ある周期ごとに，特定の割合の時間分だけ実行できることを保証する保護機能をプロセッサ利用率保護と呼ぶ．非リアルタイムアプリケーションを統合する場合，タスクは厳密なデッドラインを持たないため，アプリケーションのプロセッサ利用率を保証できれば十分である場合が多い．アプリケーションのプロセッサ利用率を保護できると，統合の前後で，アプリケーションが得られる処理量が等しくなることを保証できる．

しかし，統合プロセッサでは複数のアプリケーションが動作するため，アプリケーションの周期の中で，タスクが実行される時刻までは保証されない．すなわち，デッドラインを持つタスクが存在する場合，そのタスクのデッドラインまでにプロセッサ時間が得られるとは限らない．したがって，リアルタイムアプリケーションを統合する場合の保護機能としては，プロセッサ利用率保護だけでは不十分である．

時 間 保 護

プロセッサ利用率の保護に加えて，アプリケーションに所属するタスクが時間制約を満たすことを保証する保護機能を時間保護と呼ぶ．具体的には，QoS 制御タスクが存在する状況においても，統合前に時間制約を満たすリアルタイムアプリケーションが統合後にも時間制約を満たすことを保証する機能のことをいう[3)]．リアルタイムアプリケーションに時間保護を適用することで，個別プロセッサにおけるアプリケーションの動作検証の結果は，統合後の統合プロセッサにおいても有効となる．その結果，アプリケーション統合における再検証の負担を大幅に軽減できるため，リアルタイムアプリケーションを統合する場合の保護機能としては，時間保護が適すると考える．

2.3　保護機能の選択方針

アプリケーションごとに，時間要件によって選択すべきプロセッサ時間の保護機能が異なることに加え，アプリケーションのパラメータの中で，統合段階で既知であるものも，それぞれ異なる場合が多い．たとえば，周期タスクのみで構成されるアプリケーションでは，リリース時刻は既知であるが，イベント駆動タスクを含むアプリケーションでは，リリース時刻は既知ではない．そのため，既知のパラメータの異なるアプリケーションに対しては，同一の統合スケジューリングアルゴリズムを適用することは困難である．

そこで，本研究では，保護機能を実現する統合スケジューリングアルゴリズムを複数種類用意し，アプリケーションごとに，適用条件を満たす統合スケジューリングアルゴリズムを選択できるフレームワークを提案する．**表 1** に，実現できる保護機能ごとに，提案フレームワークで選択できる統合スケジューリングアルゴリズムと，必要なアプリケーションのパラメータを整理する．

表 1　プロセッサ時間保護と統合スケジューリングアルゴリズム
Table 1　Protection functions of processing time and scheduling algorithms for integration of real-time applications.

保護機能	アルゴリズム	必要なパラメータ
プロセッサ利用率保護	アプリケーション周期実行	アプリケーションの周期
時間保護	Lipari らの設計手法を用いた周期実行	タスクの周期と WCET
	Open System	タスクのリリース時刻と WCET
	既存の時間保護アルゴリズム	タスクのリリース時刻とデッドライン

アプリケーション周期実行アルゴリズム

アプリケーション周期実行アルゴリズムは，アプリケーションに対して，プロセッサ利用率に加えてアプリケーションを実行する周期を設定し，その周期ごとにプロセッサ利用率に応じた一定量のプロセッサ時間を要求するアルゴリズムである．アプリケーションのタスクを実行した結果，アプリケーションに割り当てられたプロセッサ時間が 0 になると，次の周期でプロセッサ時間が補充されるまでそのアプリケーション内のタスクは実行されない．このアルゴリズムを適用することで，アプリケーションのプロセッサ利用率の保護を実現できる．

時間保護アルゴリズム

時間保護を実現できるアルゴリズムは複数存在するが，アルゴリズムの適用条件が異なるため，統合段階で既知であるアプリケーションのパラメータにより，適用できるアルゴリズムを選択する．たとえば，タスクのリリース時刻と正確な WCET が得られる場合は，Lipari らの設計手法[4)] に基づく周期実行アルゴリズム，もしくは Open System[5),6)] アルゴリズムを適用できる．正確な WCET が得られず，リリース時刻とデッドラインのみ既知である場合は，我々の提案した時間保護アルゴリズム[3)] を適用できる．

また，周期タスクのみで構成され，かつ QoS 制御タスクを含まないアプリケーションに対しては，周期タスクの周期の最大公約数をアプリケーションの実行周期として設定することで，アプリケーションの周期実行アルゴリズムでも時間保護を実現できる．しかし，一般に周期タスクの周期がハーモニックでない場合も多く，すべてのタスクの時間制約を満たすためには，アプリケーションの周期を非常に短くしなければならない．この場合，アプリケーション切替え回数の増加によるオーバヘッドが問題となるため，アプリケーションの周期実行アルゴリズムは，時間保護を実現するアルゴリズムとしては適さない．

© 2008 Information Processing Society of Japan

3. スケジューリングフレームワーク

3.1 準　　備

まず，提案フレームワークの前提として，タスクやアプリケーションは互いに独立であり，タスク間のリソース共有や，アプリケーション間の通信はないものとする．また，割込み処理は考慮しない．さらに，個別プロセッサにおけるタスクのスケジューリングアルゴリズムは，プリエンプティブであるものとする．

次に，提案フレームワークに関連する用語を整理する．アプリケーションごとに割り当てられるプロセッサ時間をバジェットと呼ぶ．バジェットは，アプリケーションのタスクを実行できる残りプロセッサ時間を意味する正の整数値である．アプリケーション内のタスクが実行されると，その実行時間分だけバジェットが減少する．

統合プロセッサにおけるアプリケーションのプロセッサ利用率をシェアと呼ぶ．シェアは，システムの設計段階で決定し，統合プロセッサに対する個別プロセッサの相対性能を意味する 1 以上 100 未満の整数値である．たとえば，統合プロセッサ上で N 個のアプリケーションを動作させる場合，性能 σ_i の個別プロセッサで動作するアプリケーションのシェアは $\sigma_i / \sum_{j=1}^{N} \sigma_j$ となる．

3.2 概　　要

提案フレームワークの概要について述べる．提案フレームワークでは，**図 2** に示すように，ローカルスケジューラとグローバルスケジューラを 2 階層に配置した階層型スケジューラを採用する．

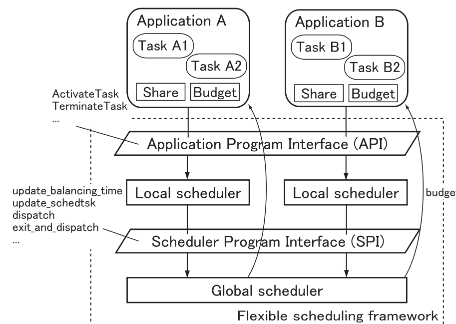

図 2　スケジューリングフレームワークの構成
Fig. 2　Construction of the hierarchical scheduling framework.

ローカルスケジューラは，アプリケーションごとに割り当てられ，所属するタスクをスケジューリングするタスクスケジューリング機能と，グローバルスケジューラに対してバジェットを要求する機能を持つ．本論文では，タスクの実行順序を決定するためのスケジューリングアルゴリズムをタスクスケジューリングアルゴリズム，アプリケーションがバジェットを要求するためのスケジューリングアルゴリズムをバジェット要求アルゴリズムと呼ぶ．

ローカルスケジューラのバジェット要求アルゴリズムは，アプリケーションにバジェットを補充してほしい時刻を決定し，それをグローバルスケジューラに通知する．このバジェットの要求時刻を，「アプリケーションを個別プロセッサと統合プロセッサで同時に実行を開始した場合に，個別プロセッサで得る処理量と統合プロセッサで得る処理量が一致する時刻」という意味で平衡時刻と呼ぶ．

グローバルスケジューラは，どのアプリケーションのタスクを実際にプロセッサで実行するかを決定するアプリケーションスケジューリング機能と，アプリケーションに補充するバジェット量を決定するバジェット補充機能を持つ．本論文では，アプリケーションの実行順序を決定するためのスケジューリングアルゴリズムをアプリケーションスケジューリングアルゴリズムと呼び，アプリケーションに補充するバジェット量を計算して補充するアルゴリズムをバジェット補充アルゴリズムと呼ぶ．

ローカルスケジューラとグローバルスケジューラ間のインタフェースをスケジューラプログラムインタフェース（SPI）と呼ぶ．ローカルスケジューラは，グローバルスケジューラに対して，SPI を介して情報の登録やタスク切替えなどの処理を要求する．提案フレームワークでは，SPI を，ローカルスケジューラのバジェット要求アルゴリズムに依存しないよう定義することで，特定の統合スケジューリングアルゴリズムに依存せず，多様なアプリケーションに対応できる．

なお，提案フレームワークは，自動車制御システム向けリアルタイム OS の標準的な仕様である OSEK/VDX OS 仕様（以下，OSEK OS 仕様）をベースにしているが，本質的な部分で OSEK OS 仕様には依存しない．そのため，たとえば μITRON 仕様など，他のリアルタイム OS の仕様に対しても容易に適用できる．以下，提案フレームワークの詳細な設計について述べる．

3.3 タ　ス　ク

タスクは，提案フレームワークにおける最小の実行単位であり，いずれかのアプリケーションに所属する．タスクの状態は，*suspended*，*ready*，*running* の 3 状態ある．タスクがローカルスケジューラのタスクスケジューリングアルゴリズムによりスケジュールされる

© 2008 Information Processing Society of Japan

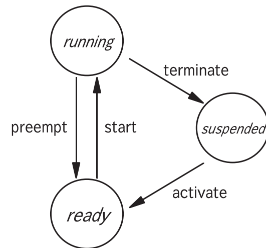

図 3　タスクの状態遷移図
Fig. 3　State transition diagram for a task.

と，**図 3** に示す状態遷移図に従って，その状態が遷移する．初期状態は *suspended* で，システムサービスにより実行可能になると *ready* に遷移する．アプリケーション内の *ready* タスクの中で最も優先度の高いタスクが *running* に遷移する．同一優先度の *ready* タスクが複数存在する場合には，最も早く *ready* になったタスクが *running* となる．*running* タスクは，実行が終了すると *suspended* に遷移し，*ready* タスクの中で最も優先度の高いタスクが *running* に遷移する．タスクスケジューリングアルゴリズムは，プリエンプティブであることを前提としているため，*running* タスクの実行中に，より優先度の高いタスクが *ready* になると，*running* タスクは *ready* に遷移し，優先度の高いタスクが *running* に遷移する．

3.4　アプリケーション

アプリケーションは 1 つ以上のタスクで構成される，タスクの集合である．アプリケーションの状態は，**表 2** に示すように，*suspended*，$replenishment-waiting$，*ready*，*running*，*exhausted* の 5 状態がある．アプリケーションの状態は，アプリケーションに所属するタスクの状態遷移とバジェット消費にともない，**図 4** の状態遷移図に従って遷移する．アプリケーションの初期状態は *suspended* である．アプリケーションに *ready* タスクが存在すると，$replenishment-waiting$ に遷移してバジェットの補充を待つ．グローバルスケジューラが，$replenishment-waiting$ アプリケーションにバジェットを補充すると，そのアプリケーションは *ready* に遷移する．*ready* アプリケーションの中で，平衡時刻の最も早いアプリケーションは *running* に遷移する．グローバルスケジューラは，*running* アプリケーションの *running* タスクを実行する．*running* アプリケーションのすべての *ready* タスクの実行が完了するか，バジェットが 0 になると，そのアプリケーションは *exhausted* に遷移し，*ready* アプリケーションの中で次に平衡時刻の早いアプリケーションが *running*

表 2　アプリケーションの状態
Table 2　States of an application.

状態名	説明
suspended	バジェットが 0 で，実行できるタスク（*ready* タスクもしくは *running* タスク）が存在しない状態．
$replenishment-waiting$	バジェットが 0 で，実行できるタスクが存在する状態．平衡時刻が決定すれば，バジェットが補充されて *ready* に遷移する．
ready	バジェットが 0 ではなく，実行できるタスクが存在する状態．ただし，*running* タスクは，実際にプロセッサで実行されてはいない．
running	*running* タスクがプロセッサで実行されている状態．
exhausted	バジェットが 0 になり，平衡時刻が経過するのを待っている状態．所属するすべての *ready* タスクの実行が完了するか，バジェットが 0 になると，この状態に遷移する．

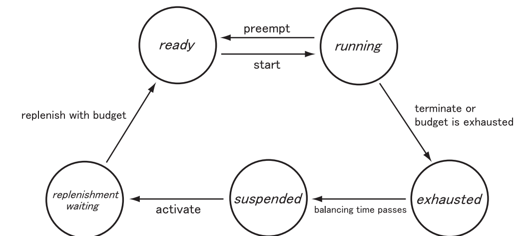

図 4　アプリケーションの状態遷移図
Fig. 4　State transition diagram for an application.

に遷移する．システム時刻が *exhausted* アプリケーションの平衡時刻を経過すると，そのアプリケーションは *dormant* に遷移する．アプリケーションが *dormant* に遷移した際に，*ready* タスクが存在する場合は，即座に $replenishment-waiting$ に遷移して，バジェットの補充を待つ．

3.5　ローカルスケジューラ

ローカルスケジューラはアプリケーションと 1 対 1 に対応し，アプリケーションに所属するタスクをスケジューリングするタスクスケジューリング機能と，平衡時刻を決定しグローバルスケジューラに通知するバジェット要求機能を持つ．また，バジェット，シェア，平衡時刻の 3 つのパラメータを管理する．

タスクスケジューリングアルゴリズムは，アプリケーションに所属するタスクをスケジュールし，SPI を介して *running* タスクの ID をグローバルスケジューラに通知する．タスク

© 2008 Information Processing Society of Japan

表 3　バジェット要求アルゴリズムと平衡時刻
Table 3　Definition of balancing time for budget requirement algorithms.

バジェット要求アルゴリズム	平衡時刻
アプリケーションの周期実行	アプリケーションの次のリリース時刻
Lipari らの設計手法に基づく周期実行	アプリケーションの次のリリース時刻
Open System	タスクのリリース時刻と WCET の中で最も早い時刻
時間保護アルゴリズム	タスクのリリース時刻とデッドラインの中で最も早い時刻

スケジューリングの具体的なアルゴリズムは，統合前と同一とする．

システム設計者は，アプリケーションの時間要件と，統合段階で既知であるパラメータから，適用するバジェット要求アルゴリズムを表 1 から選択する．各バジェット要求アルゴリズムは，**表 3** に示すように，アプリケーションのパラメータを用いて平衡時刻を決定し，SPI を介してそれをグローバルスケジューラに通知する．

たとえば，プロセッサ利用率保護を適用するためにアプリケーションの周期実行アルゴリズムを選択すると，バジェット要求アルゴリズムは，システム設計者により設定されたアプリケーションの相対周期に従い，アプリケーションの次の起動時刻を計算し，それをアプリケーションの平衡時刻としてグローバルスケジューラに通知する．また，時間保護アルゴリズムを選択した場合，バジェット要求アルゴリズムは，アプリケーションに所属するタスクのリリース時刻とデッドラインを管理し，その時刻の中で最も早い時刻を，平衡時刻としてグローバルスケジューラに通知する．

3.6　グローバルスケジューラ

グローバルスケジューラは，どのアプリケーションの *running* タスクを実行するかを決定するアプリケーションスケジューリング機能と，アプリケーションに対してバジェットを計算して補充するバジェット補充機能を持つ．

アプリケーションスケジューリングアルゴリズムは，*ready* アプリケーションの中で，平衡時刻の最も早いアプリケーションの *running* タスクを実行する．バジェット補充アルゴリズムは，$replenishment - waiting$ アプリケーションの中で，平衡時刻が登録されたアプリケーションに対して，その時点のシステム時刻から平衡時刻までの時間と，シェアの積をバジェットとして計算し，補充する．

このように，グローバルスケジューラは，アプリケーションの状態，平衡時刻，*running* タスクのみ管理することで，ローカルスケジューラのタスクスケジューリングアルゴリズムやバジェット要求アルゴリズムには依存せずに動作する．

表 4　スケジューラプログラムインタフェース（SPI）
Table 4　Scheduler Program Interface (SPI).

関数名	パラメータ	機能
`dispatch`	なし	*running* タスクを中断して，タスク切替えを要求
`exit_and_dispatch`	なし	*running* タスクを終了して，タスク切替えを要求
`update_balancing_time`	アプリケーション ID，平衡時刻	平衡時刻を更新
`update_schedtsk`	アプリケーション ID，タスク ID	実行するべきタスクの ID を更新

3.7　スケジューラインタフェース（SPI）

ローカルスケジューラとグローバルスケジューラのインタフェースである SPI のうち，スケジューリングに関連する SPI の一覧を**表 4** に示す．ローカルスケジューラは，SPI を介してグローバルスケジューラに情報を通知したり，処理を要求したりする．

update_schedtsk 関数は，ローカルスケジューラのタスクスケジューリングの結果，アプリケーション内で実行するべきタスク（*running* タスク）が変化した場合に呼び出され，新しく *running* なったタスクの ID を通知するために使用する．

dispatch 関数は，update_schedtsk 関数の後に呼び出される関数で，実行中の *running* タスクの処理を一時中断し，新しい *running* タスクに実行を切り替えるために使用する．exit_and_dispatch 関数は，*running* タスクの実行が終了したときに呼び出される関数で，次に優先度の高いタスクに実行を切り替えるために使用する．

update_balancing_time 関数は，グローバルスケジューラに登録されている，アプリケーションの平衡時刻を更新するために使用する．ローカルスケジューラが呼び出すタイミングは，そのバジェット要求アルゴリズムごとに異なる．たとえば，時間保護アルゴリズムの場合には，タスクの起動や終了するタイミングで呼び出し，アプリケーションに対して周期的にバジェットを要求する場合は，その補充周期で呼び出すことになる．

3.8　動　作　例

異なるバジェット要求アルゴリズムを適用した場合の，アプリケーションの動作の違いを具体的な例を用いて説明する．ここでは，アプリケーション 1 とアプリケーション 2 の 2 つのアプリケーションを統合するものとし，アプリケーション 1 の動作に着目する．アプリケーション 1 は，タスク 1 とタスク 2 の 2 つのタスクで構成され，タスク 2 はタスク 1 より高い優先度を持つ．各タスクの相対デッドラインは，タスク 1 が 10，タスク 2 が 14 とする．ローカルスケジューラのシェアは 50%が設定され，固定優先度ベースのプリエンプティブスケジューリングアルゴリズムでタスクをスケジュールする．

© 2008 Information Processing Society of Japan

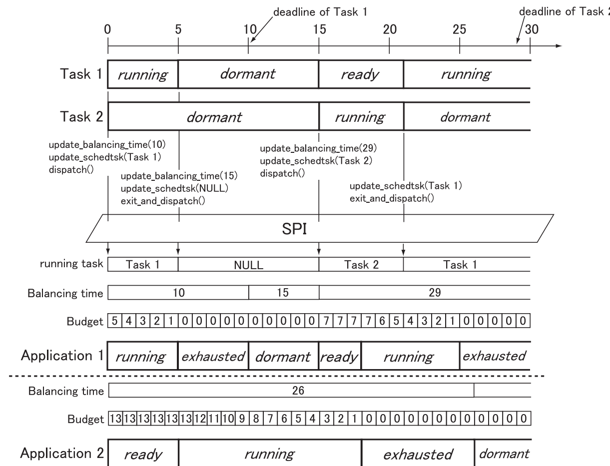

図 5　時間保護アルゴリズムを適用した場合のアプリケーションの動作例

Fig. 5　An example of schedule of an application in the case of applying the temporal protection algorithm.

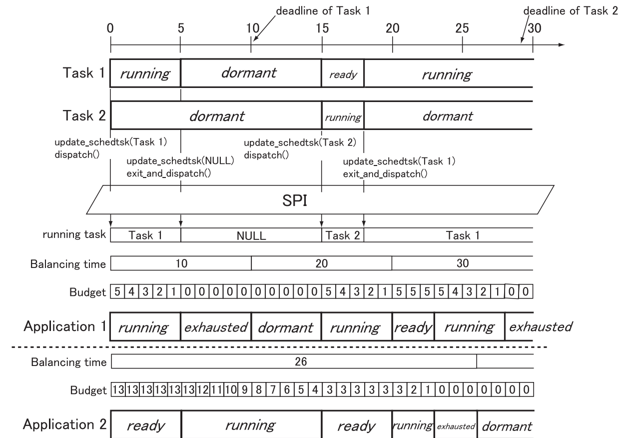

図 6　周期実行アルゴリズムによるアプリケーションの動作例

Fig. 6　An example of schedule of an application in the case of applying the cyclic execution algorithm.

まず，バジェット要求アルゴリズムとして，時間保護アルゴリズムを適用した場合の動作例を，**図 5** に示す．時間保護アルゴリズムの場合は，タスク起動により平衡時刻が変化するタイミング（時刻 0，時刻 15）や，タスクの実行終了により平衡時刻が変化するタイミング（時刻 5）で，update_balancing_time を呼び出し，アプリケーションの平衡時刻の変化を通知する．時刻 21 では，タスク 2 の実行が終了して，タスク 1 に実行が切り替わるが，平衡時刻は変化しない（タスク 2 のデッドラインである時刻 29 のまま）ため，update_balancing_time を呼び出す必要はない．タスクスケジューリングにより *running* タスクが変化するタイミング（時刻 0，時刻 5，時刻 15，時刻 21）では，update_schedtsk を呼び出し，グローバルスケジューラに *running* タスクの ID を通知することに加え，dispatch もしくは，exit_and_dispatch を呼び出して，実行するタスクの切替えを要求する．なお，アプリケーションが *ready* であっても，より平衡時刻の早いアプリケーションが存在する場合には，*running* に遷移しない．たとえば，時刻 15 でアプリケーション 1 は *ready* であるが，平衡時刻が時刻 29 よりも早いアプリケーション 2 が *running* であるため，アプリケーション 2 がバジェットを使いきる時刻 18 までは *ready* のままである．

次に，バジェット要求アルゴリズムとして，周期実行アルゴリズムを適用した場合の動作例を，**図 6** に示す．アプリケーション 1 のバジェットの補充周期は 10 とする．この場合，通常であれば，ローカルスケジューラは定期的にバジェットを要求するために，周期 10 で update_balancing_time を呼び出す必要があるが，提案フレームワークではこれをグローバルスケジューラ内で自動的に平衡時刻を更新する仕組みを取り入れているため，明示的に update_balancing_time を呼び出す必要はない．タスクスケジューリングにより *running* タスクが変化する時刻（時刻 0，時刻 5，時刻 15，時刻 18）では，update_schedtsk を呼び出して，*running* タスクの ID をグローバルスケジューラに通知し，dispatch もしくは，exit_and_dispatch を呼び出す．

4. 実装と評価

4.1 実　　装

提案フレームワークのプロトタイプを，OSEK OS 仕様準拠の TOPPERS/OSEK カーネルをベースに実装した．プロトタイプのタスクスケジューリングアルゴリズムは，OSEK OS 仕様の固定優先度ベースのプリエンプティブスケジューリングである．

ターゲットボードは，Renesas 社製 M32R-II ソフトコアプロセッサ（動作周波数：10 MHz，キャッシュ：OFF）を搭載した M3A-ZA36 ボードである．メモリは，32 K バイトのコア内

© 2008 Information Processing Society of Japan

蔵SRAMと，64Mバイトの外部SDRAM（動作周波数：10MHz）を持つ．このプロセッサの性能は，現在ハイエンドな組み込み向け機器に用いられるものに比較すると大分劣る．特に，動作周波数については，実際の自動車制御システムの制御系で用いられるマイコンの動作周波数の1/4から1/10程度である．そのため，性能評価の段階ではこの点を考慮する．

プロトタイプの実装においては，アプリケーションのバジェットを管理するタイマと，システム時刻を管理するタイマの2つのハードウェアタイマを使用した．また，バジェット管理用タイマの割込みは，カーネルが管理する割込みの中で最も優先して受け付けるように設定する必要があるため，タイマごとに割込み優先度を設定できる割込みコントローラが必要となる．

4.2 評　　価

実装したプロトタイプの性能を評価する．評価に用いたアプリケーションは，周期タスクのみで構成されるリアルタイムアプリケーションと，周期タスクにバックグラウンドタスクを加えた非リアルタイムアプリケーションを用意する．リアルタイムアプリケーションに対しては時間保護アルゴリズムを適用し，非リアルタイムアプリケーションに対しては，周期実行アルゴリズムを適用する．

実行形式サイズ

まず，提案フレームワークの実装によるメモリ消費量を評価する．TOPPERS/OSEKカーネルとプロトタイプの実行形式ファイルを**表5**に示す．ここでは，実行形式ファイルにアプリケーションの領域は含めず，カーネル本体のみを測定対象とする．プロトタイプのROMとRAMの使用量は，TOPPERS/OSEKカーネルに比較して，ROMが6.4KB，RAMが0.7KB増加した．主な要因は，スケジューラを階層化することによるコードの増加や，バジェット管理用タイマを操作するコードなど追加したことである．測定結果より，ROMとRAM使用量の増加量は非常に少なく，実用上許容できる範囲であると評価する．

タスク切替え時間

次に，提案フレームワークの主な処理であるスケジューリング，ディスパッチ，タスク切替え処理時間を評価する．アプリケーション数は2つとし，それぞれ周期実行アルゴリズムと時間保護アルゴリズムを適用する．各アプリケーションのタスク数は，それぞれ2個（シェアはそれぞれ40%），4個（シェアはそれぞれ20%），8個（シェアはそれぞれ10%）と変化させる．

TOPPERS/OSEKカーネルとプロトタイプについて，同一アプリケーションのタスクへの切替え処理（OSEKサービスコールのActivateTaskの実行）時間を**表6**に示す．スケジューリング時間とは，*running*タスクより優先度の高いタスクを起動するサービスコールを呼び出してから，タスクコンテキストの退避処理を開始するまでの時間である．ディスパッチ時間とは，タスクのコンテキスト退避処理を開始してから，優先度の高いタスクの処理が開始されるまでの時間である．測定結果によると，同一アプリケーションのタスクへの切替え処理では，周期実行アルゴリズムの場合は，スケジューリング時間が41 μs，ディスパッチ時間が17 μs，タスク切替え全体の時間としては57 μsそれぞれ増加した．また，時間保護アルゴリズムの場合は，スケジューリング時間が141 μs，ディスパッチ時間は17 μs，タスク切替え全体の時間としては161 μsそれぞれ増加した．また，実行中のアプリケーションがバジェットを使いきり，別のアプリケーションに切り替えるための処理時間は，130 μs程度であった．同一アプリケーションのタスクへの切替えでは，タスクスケジューリングをO(1)で処理できるため，これらの測定結果は，アプリケーションのタスク数により変化しない．

表5　カーネルの実行形式（ELF）ファイルサイズ
Table 5　Size of excutable files (ELF format).

	ROM（KB）	RAM（KB）
TOPPERS/OSEKカーネル	11.6	1.6
プロトタイプ	18.0	2.3

表6　同一アプリケーションのタスクへの切替え処理時間
Table 6　Processing time of task switching in a same application.

	スケジューリング時間（μs）	ディスパッチ時間（μs）	タスク切替え時間（μs）
TOPPERS/OSEKカーネル	49	25	77
周期実行アルゴリズム	90	42	134
時間保護アルゴリズム	190	42	234

次に，時間保護アルゴリズムにおいて，別のアプリケーションのタスクへ切り替える場合のタスク切替え処理時間を**表7**に示す．各アプリケーションが独立であるという前提のもとで，別アプリケーションのタスクへ切り替える状況とは，実行中のアプリケーション内で実行するタスクを切り替えた結果，平衡時刻が更新され，その更新された平衡時刻が別のアプリケーションの平衡時刻より遅くなったために，実行するアプリケーションが切り替わる場合である．別のアプリケーションのタスクへの切替え処理では，スケジューリング時間が，さらに58 μs増加した．

評価ボードのプロセッサ性能を考慮すると，測定した処理時間については実用上許容でき

© 2008 Information Processing Society of Japan

表 7　時間保護アルゴリズムにおける別アプリケーションのタスクへの切替え処理時間
Table 7　Processing time of task switching between applications.

	スケジューリング時間（μs）	ディスパッチ時間（μs）	タスク切替時間（μs）
時間保護アルゴリズム	248	42	292

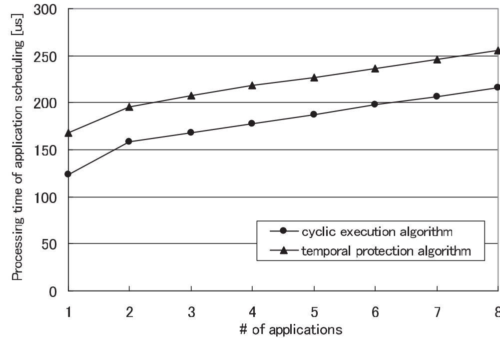

図 7　アプリケーションスケジューリング時間
Fig. 7　Processing time of application scheduling.

る範囲のオーバヘッドであると考えるが，特に，時間保護アルゴリズムについては，今後さらにオーバヘッドを小さくするよう実装方法を改善する必要がある．

アプリケーションスケジューリング時間

統合するアプリケーションの数により変動するオーバヘッドとしては，グローバルスケジューラにおけるアプリケーションのスケジューリング時間がある．アプリケーションのタスク数を 4 つに固定し，アプリケーションの数を変化させたときの最長スケジューリング処理時間を**図 7** に示す．なお，スケジューリングが最長となる状況とは，実行可能になったアプリケーションの平衡時刻が，実行可能アプリケーションの中で最も遅い状況であり，このときの処理時間が最長処理時間となる．

プロトタイプでは，実行可能なアプリケーションを管理するキューに対して，新たに実行可能になったアプリケーションの挿入位置を線形探索するため，スケジューリングの最長処理時間は，統合するアプリケーション数に比例して増加する．測定結果によると，統合するアプリケーション数の増加に対して，約 10 μs 程度のオーバヘッドで済む．

5. 関連研究

アプリケーション統合に適用できるアルゴリズムとして，バジェットをタスク単位で管理するサーバアルゴリズムが提案されている[7)–10)]．さらに，タスクに割り当てたバジェットが余る場合に，それらを他のタスクが利用することで，統合プロセッサの利用率を向上させるアルゴリズムも提案されている[11)–14)]．

これらのアルゴリズムは，主に，周期タスクのリアルタイム性の保証と，非周期タスクの応答性の改善を目的としており，タスク単位でバジェットを保護できる特徴がある．これらの手法でアプリケーションを統合する場合，各タスクのリアルタイム性を保証するためには，統合するすべてのタスクの動作を考慮してスケジュール可能性を検証する必要がある．そのため，統合するアプリケーションの組合せが変化するたびに，スケジュール可能性を解析し直さなければならない．また，これまで提案されたバジェット再利用アルゴリズムでは，余りバジェットをどのタスクが再利用するかを実行時に決定するため，同一のアプリケーションに所属するタスクが利用するとは限らない．また，タスク数が多い場合は，各タスクのバジェットや余りバジェットを管理するための計算が多くなるという問題がある．

一方，提案フレームワークでは，アプリケーション単位でプロセッサ利用率保護や時間保護を選択でき，他のアプリケーションの動作に関係なく保護できる．そのため，統合するアプリケーションの組合せが変化する場合にも，スケジュール可能性の検証を最小限にとどめることができる．さらに，アプリケーション単位でバジェットを管理するため，余りバジェットは必ず同一のアプリケーションに所属するタスクが再利用できる．

複数のリアルタイムアプリケーションを単一のプロセッサに統合することを目的とした階層型スケジューラがいくつか提案されている．Deng らは，すべてのタスクのリリース時刻と WCET が既知であることを適用条件として，プロセッサ利用率保護と時間保護を実現できる Open System を提案している[5),6)]．しかし，適用条件に合わないタスクが存在するアプリケーションを統合できないという問題がある．

Lipari らは，タスクのデッドラインのみを利用してアプリケーションのプロセッサ利用率と，タスクが時間制約を満たすことを保証する PShED アルゴリズムを提案した[15)]．適用条件がタスクのデッドラインのみであるため，多くのアプリケーションに適用できると考えるが，QoS 制御タスクが存在する場合，より高い優先度を持つタスクが時間制約を満たせなくなるという問題がある．我々は，タスクのリリース時刻とデッドラインが既知であることを適用条件として，QoS 制御タスクの存在を考慮した時間保護アルゴリズムを提案し

© 2008 Information Processing Society of Japan

た[3]．しかしながら，時間保護アルゴリズムでも，すべてのタスクに対して同じ前提条件を定めており，それに合わないタスクを含むアプリケーションを統合できないという問題がある．また，時間保護アルゴリズムの実装による評価がなされていない．

Lipari らは，タスクのリリース周期と WCET を利用して，アプリケーション内のすべてのタスクが時間制約を満たす (P,Q) の組を求める手法を提案している[4]．P はアプリケーションの実行周期，Q は周期ごとに必要なプロセッサ時間である．この手法は，QoS 制御タスクが存在する場合にも時間保護を実現でき，実行時にはすべてのアプリケーションを周期実行するため比較的容易に実装できるという特徴がある．しかし，アプリケーション内に周期の長いタスクと短いタスクが混在すると，アプリケーションの実行周期を周期の短いタスクに合わせて設定する必要があり，PShED アルゴリズムや我々の時間保護アルゴリズムのようなデッドラインを用いる手法に比べて，デッドラインに余裕があるタスクの切替え回数が増加するという問題がある．また，タスクのデッドライン情報が得られる場合でも，これを有効に活用する手段がない．本論文では，これらのアルゴリズムをバジェット要求アルゴリズムとして採用し，アプリケーションごとに選択できるフレームワークを提案している点で，従来の提案手法とは異なる．

HLS[16] と S.Ha.R.K カーネル[17)–19] は，複数のタスクスケジューリングアルゴリズムを混在させることができるフレームワークである．いずれのフレームワークにおいても，時間保護を実現するアルゴリズムは扱われておらず，本論文が目標としているリアルタイムアプリケーション統合の容易化を実現するものではない．

Oikawa らは，マイクロカーネル上で複数のリアルタイム OS を動作させ，OS レベルでリアルタイムアプリケーションを統合する手法を提案している[20]．OS の多重化は，統合するアプリケーションの構造を変更することなく適用できるという利点がある．しかし，ローカルスケジューラを多重化する手法に比べると，メモリ消費が多く，OS 切替えによる処理オーバヘッドも大きいと考えられる．また，リアルタイムアプリケーションのタスクが時間制約を満たすことを保証するものではないため，複数のリアルタイムアプリケーションを統合する場合には，短時間で頻繁に OS を切り替える必要がある．したがって，リソース制約が厳しいリアルタイムシステムに適用する場合，提案フレームワークの方が有利であると考える．

6. 結　　論

本論文では，分散システムにおいて，各システムで単独で動作するアプリケーションを 1 つのプロセッサ上に統合することを目的としたスケジューリングフレームワークを提案した．提案フレームワークでは，アプリケーションごとに適切なバジェット要求アルゴリズムを選択することで，アプリケーションに要求される時間要件や，統合段階で既知であるパラメータが異なるアプリケーションに，柔軟に対応できる．そのため，従来の統合スケジューリングアルゴリズムでは，容易に統合できなかった分散システムに適用できると考える．

提案フレームワークを OSEK OS 仕様準拠のリアルタイム OS をベースに実装し，メモリ消費量，タスク切替え処理時間，アプリケーションスケジューリング時間の観点から評価した結果，実用上許容できる程度のオーバヘッドで実現できることを確認した．よって，提案フレームワークは小型のマイコンに対しても適用できると考える．

実システムに適用するための課題として，まず，提案フレームワークのオーバヘッド削減のための実装チューニングを行う．次に，OS のオーバヘッドと割込みハンドラ処理時間を考慮したスケジューリングアルゴリズムを検討する．実際の自動車制御システムには，1 回の割込みハンドラの処理時間は短いが，割込み回数が非常に多いアプリケーションが存在する．また，時間制約が厳しいため，プロセッサ利用率の設定においては，OS のオーバヘッドを軽視できない．提案フレームワークでは，OS のオーバヘッドと割込みハンドラの処理時間は，タスク処理時間に比較して非常に短いという前提を置いているため，これらの時間は考慮していないが，実システムへ適用するためには，これらを考慮したスケジューリングアルゴリズムの検討が必要であると考えられる．

謝辞　本研究は，情報処理推進機構（IPA）が実施した未踏ソフトウェア創造事業の援助を受けた．

参　考　文　献

1) AUTOSAR. http://www.autosar.org
2) JASPAR. http://www.jaspar.jp/
3) 松原　豊，本田晋也，冨山宏之，高田広章：時間保護のためのリアルタイムスケジューリングアルゴリズム，情報処理学会論文誌：コンピューティングシステム，Vol.48, No.SIG 8(ACS18), pp.192–202 (2007).
4) Lipari, G. and Bini, E.: A methodology for designing hierarchical scheduling *systems, Journal fo Embedded Computing*, Cenbridge International Science Publishing (2003).
5) Deng, Z., Liu, J.-S. and Sun, J.: A Scheme for Scheduling Hard Real-Time Applications in Open System Environment, *Proc. 9th Euromicro Workshop on Real-Time Systems*, pp.191–199 (1997).

© 2008 Information Processing Society of Japan

6) Deng, Z., Liu, J.W.-S., Zhang, L., Mouna, S. and Frei, A.: An Open Environment for Real-Time Applications, *Real-Time Systems Journal*, Vol.16, pp.155–185 (1999).
7) Strosnider, J., Lehoczky, J. and Sha, L.: The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments, *IEEE Trans. Comput.*, Vol.44, No.1 (1995).
8) Spuri, M. and Buttazzo, G.: Scheduling aperiodic tasks in dynamic priority systems, *Real-Time Systems Journal*, Vol.10, pp.179–210 (1996).
9) Abeni, L. and Buttazzo, G.: Integrating Multimedia Applications in Hard Real-Time Systems, *Proc. IEEE Real-Time System Symposium* (1998).
10) Abeni, L. and Buttazzo, G.: Resource Reservations in Dynamic Real-Time Systems, *Real-Time Systems Journal*, Vol.27, No.2, pp.123–165 (2004).
11) Lipari, G. and Baruah, S.: Greedy Reclaimation of Unused Bandwidth in Constant-Bandwidth Servers, *Proc. IEEE 12th Euromicro Conference on Real-Time Systems*, pp.193–200 (2000).
12) Caccamo, M., Buttazzo, G. and Sha, L.: Capacity sharing for overrun control, *Proc. IEEE 21th Real-Time System Symposium* (2000).
13) Nogueira, L. and Pinho, L.M.: Capacity Sharing and Stealing in Dynamic Server-based Real-Time Systems, *Proc. International Parallel and Distributed Processing Symposium* (2007).
14) Caccamo, M., Buttazoo, G.C. and Thomas, D.D.: Efficient reclaiming in reservation-based real-time systems with variable execution times, *IEEE Trans. Comput.*, Vol.54, No.2, pp.198–213 (2005).
15) Lipari, G., Carpenter, J. and Baruah, S.: A framework for acheving inter-application isolation in multiprogrammed, hard real-time environments, *Proc. IEEE Real-time System Symposium*, pp.217–226 (2000).
16) Regehr, J. and Stankovic, J.A.: HLS: A Framework for Composing Soft Real-Time Schedulers, *Proc. IEEE 22nd Real-Time System Symposium* (2001).
17) Gai, P., Abeni, L., Giorgi, M. and Buttazzo, G.: A New Kernel Approach for Modular Real-Time systems Development, *Proc. IEEE 13th Euromicro Conference on Real-Time Systems* (2001).
18) Lipari, G., Gai, P., Trimarchi, M., Guidi, G. and Ancilotti, P.: A Hierarchical Framework for Component-Based Real-Time Systems, *Proc. IEEE International Symposium on Component-based Software Engineering* (2004).
19) Gai, P., Lipari, G., Abeni, L., di Natale, M. and Bini, E.: Architecture for A Portable Open Source Real Time Kernel Environment, *Proc. 2nd Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial* (2000).
20) Oikawa, S., Ishikawa, H., Iwasaki, M. and Nakajima, T.: Providing Protected Execution Environments for Embedded Operating Systems Using a u-Kernel, *Proc. International Conference on Embedded and Ubiquitous Computing*, pp.153–163 (2004).

(平成 19 年 12 月 27 日受付)
(平成 20 年 7 月 1 日採録)

松原　豊（学生会員）

2005 年名古屋大学大学院情報科学研究科情報システム学専攻博士前期課程修了．2006 年より同博士後期課程．リアルタイム OS，リアルタイムスケジューリング理論，自動車制御システムの研究に従事．

本田　晋也（正会員）

名古屋大学大学院情報科学研究科付属組込みシステム研究センター助教．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．2005 年名古屋大学情報連携基盤センター名古屋大学組込みソフトウェア技術者人材養成プログラム産学官連携研究員．2006 年から現職．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．電子情報通信学会会員．

冨山　宏之（正会員）

1999 年 3 月九州大学大学院システム情報科学研究科博士後期課程修了．同年米国カリフォルニア大学アーバイン校客員研究員．2001 年（財）九州システム情報技術研究所研究員．2003 年名古屋大学大学院情報科学研究科講師．現在，同准教授．SOC や組み込みシステムの設計技術に関する研究に従事．電子情報通信学会，ACM，IEEE 各会員．博士（工学）．

© 2008 Information Processing Society of Japan

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み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，組み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各会員．

© 2008 Information Processing Society of Japan