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

松原 豊[†] 本田晋也[††]
冨山宏之[†] 高田広章[†,††]

ハードリアルタイム性を要求される複数のアプリケーションを単一のプロセッサ上に容易に統合するためには，統合前の環境において時間制約を満たせるアプリケーションは，統合後の環境においても，時間制約を満たすことが望ましい．本論文では，QoS 制御されたタスクが存在する場合においても，このことを保証するスケジューリングアルゴリズムを提案する．まず，統合後の環境においてアプリケーションが時間制約を満たせなくなる要因を 3 つあげ，時間保護を実現するスケジューリングアルゴリズムが満たすべき要件を定義する．要因の中の 3 つ目では，QoS 制御されたタスクの存在により，アプリケーションが時間制約を満たせなくなる場合があることを指摘する．次に，各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前提として，時間保護の要件を満たし，かつシステムの設計段階において適用しやすいスケジューリングアルゴリズムを提案し，その正当性を証明する．

Real-time Scheduling Algorithm for Temporal Protection

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

In the field of embedded systems, it is required to integrate two or more real-time applications working on a dedicated processor onto a high performance single processor. To achieve it easily, application developers need a property that an application satisfy its temporal constraints after integration if the application satisfies it before integration. This paper presents a real-time scheduling algorithm to guarantee that all applications satisfy the property even if a Quality of Services (QoS) controlled task exists. Firstly, deadline miss causations after integration, occurred by applications satisfying temporal constraints before integration, are classified into only three factors. The third causation indicates that QoS controlled task can cause timing failure of other applications. Secondly, a requirement which scheduling algorithms for temporal protection should satisfy is defined. Thirdly, as the assumption that release time and deadline time of all tasks are known statically, we proposed a scheduling algorithm which is available in the application design phase. Finally, it is confirmed that the scheduling algorithm satisfies the requirement.

1. はじめに

近年，ハードリアルタイム性を要求される組み込みシステムの分野においても，ソフトウェアの大規模化，複雑化が著しく，その信頼性をどのように確保，向上させるかが大きな課題となっている．ハードリアルタイム性を要求される代表的な組み込みシステムの 1 つである自動車制御システムにおいても，低燃費化，走行性能の向上，排気ガス規制への対応などの目的から，ソフトウェアの大規模化，複雑化が急速に進んでいる．

自動車に搭載される制御用のコンピュータを ECU（Electronic Control Unit；電子制御ユニット）と呼ぶが，1 台の自動車に搭載される ECU の数は，多いもので，1998 年では 30 個程度であったが，2006 年には 100 個程度まで増加している．ECU の数はさらに増加する傾向にあり，コストの増大や ECU 搭載スペースの不足といった問題を起こしている．

自動車に搭載される ECU の数が増加している理由の 1 つとして，エンジン制御，ステアリング制御，ブレーキ制御といったアプリケーションごとに別々の ECU が用いられており，ECU に搭載できるプロセッサが高性能化しているにもかかわらず，それらが統合されていないことがあげられる．そのため，自動車に新しい機能を追加するたびに，ECU が追加されるこ

† 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
†† 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
Center for Embedded Computing Systems, Graduate School of Information Science, Nagoya University

とになる．

ECU の統合が進まない大きな理由として，ECU に使われている OS が保護機能を持っていないことがあげられる（そもそも，OS を使っていない ECU も多い）．OS が保護機能を持っていないため，複数のアプリケーションを 1 つの ECU に統合した場合，あるアプリケーションの問題により別のアプリケーションが障害を起こす可能性がある．そのため，各アプリケーションが別々に開発されている場合には，障害原因や責任の切り分けが非常に困難となる．

このような背景から著者らは，ハードリアルタイム性を要求される組み込みシステム向けに，保護機能を持ったリアルタイム OS の実現を目指している．リアルタイム OS に求められる保護機能としては，資源へのアクセスに対する保護機能（メモリ保護機能など）に加えて，時間多重される資源の利用時間に対する保護機能（これを時間保護機能と呼ぶ）がある．時間多重される資源の中で最も重要なのがプロセッサである．本研究は，プロセッサに対する時間保護機能を対象とする．

複数のアプリケーションを統合する場合，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していたアプリケーションが，統合することにより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．

そこで本論文では，統合前の環境において時間制約を満たせるアプリケーションは，統合後の環境においても，必ず時間制約を満たすことを保証するスケジューリングアルゴリズムを検討する．また本論文では，各アプリケーションは 1 つまたは複数のタスクで構成されており，各アプリケーション内では，プリエンプティブなタスクスケジューリングアルゴリズムが用いられていることを前提とする．

過去に，アプリケーション単位でプロセッサ時間を割り当てるさまざまなサーバアルゴリズムが提案されている[3]．これらのアルゴリズムの多くは，アプリケーションのプロセッサ時間を他のアプリケーションから保護することに重点を置いており，アプリケーション内のタスクが時間制約を満たすことを保証できるアルゴリズムは少ない．さらに，この保証に着目したアルゴリズムにおいても，システムの負荷に応じて処理内容が変化するタスク（このようなタスクを，QoS 制御されたタスクと呼ぶ）を考慮しているものはない．

そこで，まず，統合後の環境においてアプリケーションが時間制約を満たせなくなる要因を 3 つあげ，時間保護を実現するスケジューリングアルゴリズムが満たすべき要件を定義する．要因の中の 3 つ目は，QoS 制御されたタスクの存在により，アプリケーションが時間制約を満たせなくなる場合があることを指摘するもので，過去の研究では考慮されていない．現実の自動車制御システムにおいては，QoS 制御されたタスクが存在しており，この要因を考慮に入れて要件を定義することが不可欠である．

第 3 の要因を考慮に入れて提案されたものではないが，Deng らのアルゴリズム[4),5)] は，定義した要件を（結果的に）満たしている．このアルゴリズムは，各タスクのリリース時刻と実行時間の上限値が既知であることを前提としているが，システム設計段階においてタスクの実行時間の上限値を正確に見積もることは容易ではない．

そこで本論文では，時間保護を実現し，かつシステム設計段階で適用しやすいスケジューリングアルゴリズムとして，Deng らのアルゴリズムをベースに，見積もりにくい実行時間の上限値に代えてデッドラインを用いるように修正したアルゴリズムを提案する．さらに，提案アルゴリズムが時間保護の要件を満たすことを証明する．

以下，本論文の構成を述べる．まず 2 章で，時間保護の目的と，統合後にアプリケーションが時間制約を満たせなくなる要因を整理し，時間保護機能を実現するスケジューリングアルゴリズムが満たすべき要件を定義する．3 章では，時間保護機能の要件を満たすスケジューリングアルゴリズムを提案し，その動作について詳細に述べる．4 章では，提案アルゴリズムが時間保護機能の要件を満たすことを示す．5 章では，関連研究について紹介し，本研究との差異を詳細に述べる．最後に，6 章で今後の課題と結論を述べる．

2. 時間保護機能

2.1 用語の定義と目的

本論文の目的は，複数の低性能プロセッサ上で動作するアプリケーションを，単一の高性能プロセッサ上に容易に統合するための時間保護機能を提案することである．ここで，低性能プロセッサとは，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 ECU のプロセッサのことを，高性能プロセッサとは，複数のアプリケーションを動作させるための統合 ECU のプロセッサのことをいう．本論文では，プロセッサ性

表 1 アプリケーション A のタスクセット
Table 1 Task set of application A.

	Period	Deadline	Maximum Execution Time	Utilization	Worst-case Response Time
High Priority Task	4	4	2	50%	2
Middle Priority Task	5	3	1	20%	3
Low Priority Task	15	15	4	26.6%	15

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．

低性能プロセッサで動作するアプリケーションを，高性能プロセッサ上に容易に統合するためには，低性能プロセッサで時間制約を満たすアプリケーションは，高性能プロセッサへの統合後にも時間制約を満たすことが望ましい．ここで，アプリケーションが時間制約を満たすとは，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ことをいい，タスクが時間制約を満たすとは，いかなる状況においても与えられたデッドラインまでに処理を完了することをいう．

低性能プロセッサで時間制約を満たすタスクが，高性能プロセッサへの統合後にも時間制約を満たすことを保証できると，低性能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，統合後の高性能プロセッサ上でも有効となる．その結果，統合に際してアプリケーションの再検証の負担は大幅に軽減でき，統合が容易になったということができる．

2.2 時間制約を満たせなくなる要因と時間保護機能の要件

高性能プロセッサへの統合により，アプリケーションが時間制約を満たせなくなる原因として，次の 3 つの要因が考えられる．

1 つ目の要因は，別のアプリケーションがプロセッサ時間を使いすぎることにより，当該アプリケーションに割り当てられるプロセッサ時間が不足することである．これを防ぐためには，アプリケーションごとに利用可能なプロセッサ時間を管理し，それを超えてアプリケーションを実行しない機構が必要である．

2 つ目の要因は，アプリケーション内でのスケジュールが変化することである．アプリケーションが複数のタスクで構成されている場合には，アプリケーションに割り当てられるプロセッサ時間が十分であっても，タスクの実行順序が変化することで，一部のタスクが時間制約を満たせなくなる場合がある．

このことを**表 1** のアプリケーションを例に説明する．アプリケーション A は，3 つの周期タスクで構成

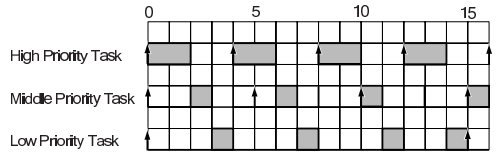

図 1 アプリケーション A の低性能プロセッサ上での動作
Fig. 1 Behavior of application A on a low performance processor.

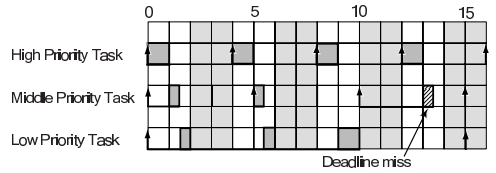

図 2 統合後のスケジュール変化によるデッドラインミス
Fig. 2 A deadline miss caused by changing of task scheduling after integration of processors.

され，固定優先度のプリエンプティブスケジューリングアルゴリズムに基づいてスケジュールされる．最大実行時間は低性能プロセッサにおける実行時間，最悪応答時間は Critical Instant 定理により導かれる最悪時の応答時間である．

アプリケーション A は，性能 1 の低性能プロセッサ上では**図 1** のように実行され，時間制約を満たせる．図中の上方向矢印はタスクのリリースを，縦の太線はデッドラインを（デッドラインがリリースと同じ場合は省略），リリースの矢印から横方向に伸びる下線はタスクが実行可能状態であることを示す．

ここでは，性能 2 の高性能プロセッサ上で，アプリケーション A をプロセッサ利用率 50%以内で時間制約を満たせるようにスケジューリングするアルゴリズムを検討する．まず，単純な方法として，高優先度タスクのリリース周期である 4 単位時間ごとに最大 2 単位時間のプロセッサ時間を与えるアルゴリズムを適用する．アプリケーション A は，与えられたプロセッサ時間を周期の最初から使えると仮定すると**図 2** のように実行される（この例では濃淡部分のうち，淡部で実行可能タスクを実行する）．時刻 0 で 3 つのタスクが実行可能になると，まず高優先度タスクが実行され，低性能プロセッサでの半分の時間で処理を完了する．次に，中優先度タスクが実行され，0.5 単位時間

表 2 QoS 制御されたタスクを含むアプリケーション A' のタスクセット
Table 2 Task set of application A' including a QoS controlled task.

	Period	Deadline	Maximum Execution Time	Utilization	Worst-case Response Time
High Priority Task	4	4	2	50%	2
Middle Priority Task	5	3	1	20%	3
QoS Controlled Task	15	15	4(6)	26.6(40)%	15

で処理を完了する．さらに，低優先度タスクが 0.5 単位時間実行される．時刻 2 になると時刻 0 で与えられたプロセッサ時間がなくなるため，アプリケーション A はこれ以上実行を継続できない．時刻 4 で，再び 2 単位時間のプロセッサ時間が得られると，このとき最も優先順位の高い高優先度タスクが実行される．

時刻 8 から時刻 12 の間のタスク実行順序に着目すると，高優先度タスク，低優先度タスクの順番で実行されており，低性能プロセッサでスケジュールした場合（高優先度タスク，中優先度タスクの順番）と異なる．その結果，時刻 13 で中優先度タスクがデッドラインをミスしてしまう．このように，アプリケーション A が得られるプロセッサ時間は十分であるにもかかわらず，タスクの実行順序が変更されることにより，低性能プロセッサで時間制約を満たすタスクが，高性能プロセッサでは時間制約を満たせなくなる場合がある．

これを防ぐ 1 つの方法として，PShED アルゴリズム[6] をあげることができる．PShED アルゴリズムでは，アプリケーションに対して周期的にプロセッサ時間を与えるのではなく，実行可能なタスクごとに，リリースからデッドラインまでの間で使用可能なプロセッサ時間を管理する．タスクが使用したプロセッサ時間を，そのタスクよりデッドラインが長いタスクの使用可能プロセッサ時間にも反映することで，アプリケーション内で最も長いデッドラインまでのプロセッサ利用率を設定値以下に制限する．また，タスクがリリースされた時点でプロセッサ時間を与えるので，タスクは優先度順に実行される．アプリケーション A を PShED アルゴリズムでスケジュールすると，**図 3** のように実行される．短い時間幅でのプロセッサ利用率は 50%を超えるが，時刻 0 から図中で最も長いデッドライン（時刻 15）までの間のプロセッサ利用率は 50%となる．このように，アプリケーション A に対して PShED アルゴリズムを適用することで，アプリケーションごとにプロセッサ利用率を制限するだけではなく，低性能プロセッサ上でのタスクの実行順序を維持し，かつすべてのタスクが時間制約を満たせる．PShED アルゴリズムは 2 つ目の要因に対応できるが，次に説明する 3 つ目の要因には対応できない．

3 つ目の要因は，システムの負荷に応じて処理内容

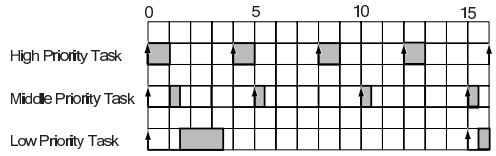

図 3 PShED アルゴリズムを適用したアプリケーション A の動作
Fig. 3 Behavior of application A scheduled by PShED algorithm on a high performance processor.

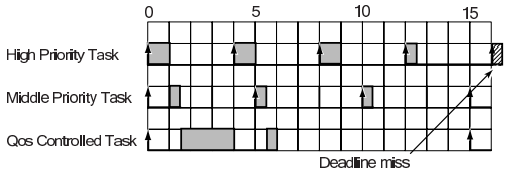

図 4 PShED アルゴリズムを適用した QoS 制御されたタスクを含むアプリケーション A' の動作
Fig. 4 Behavior of application A' including a QoS controlled task by PShED algorithm on a high performance processor.

を変化させている場合など，高性能プロセッサで他のアプリケーションと統合して実行した場合に，低性能プロセッサで実行したときと比べて，要求する処理量が変化するタスクの存在である．本論文では，このようなタスクを QoS 制御されたタスクと呼ぶ．たとえば，QoS 制御されたタスクは，実行中にデッドラインになった場合にはただちに実行を終了するが，他に実行可能タスクが存在しない場合にはさらにいくらか実行を継続する．

アプリケーション内に QoS 制御されたタスクが存在すると，PShED アルゴリズムを適用しても，統合後に一部のタスクが時間制約を満たせない場合がある．このことを**表 2** に示すアプリケーション A' をスケジュールした例で示す（**図 4**）．アプリケーション A' は，アプリケーション A の低優先度タスクを QoS 制御されたタスクに置き換えたものである．アプリケーション A' の QoS 制御されたタスクは，アプリケーション内で最も優先度が低く，プロセッサ利用率が高い場合は 4 単位時間（低性能プロセッサにおける処理時間）だけ要求するが，アイドル状態などプロセッサに余裕がある場合は 6 単位時間の処理を要求するタスクである．図 4 において，QoS 制御されたタスクは，

時刻 1.5 から時刻 4 までと時刻 5.5 から時刻 6 までの合計 3 単位時間実行され，将来リリースされるタスクのプロセッサ時間を先使いするため，高優先度タスクが時刻 16 のデッドラインをミスしてしまう．QoS 制御されたタスクによるプロセッサ時間の先使いを防ぐためには，タスクのリリース時刻をスケジューラが把握し，将来到着する高優先度タスクが時間制約を満たすために必要なプロセッサ時間を残しておくことが必要である．

以上の要因を考慮に入れ，本論文では，時間保護機能を実現するスケジューリングアルゴリズムが満たすべき要件を「QoS 制御されたタスクが存在する場合にも，低性能プロセッサで時間制約を満たすタスクは，高性能プロセッサへの統合後も時間制約を満たす」と定義する．この要件を満たすスケジューリングアルゴリズムを次章から詳細に述べる．

3. 時間保護を実現する階層型スケジューラ

3.1 前　　提

提案アルゴリズムでは，タスクのリリース時刻とデッドラインの 2 つの情報が既知であることを前提とする．タスクの待ち状態とタスク間通信は考慮しない．また，割込み処理時間はタスク処理時間に比べて非常に短いものと仮定し，割込み処理で使用されるプロセッサ時間は無視できるものとする．

3.2 用 語 定 義

ここでは，提案アルゴリズムに関する用語を定義する．

タスクのリリース時刻とデッドラインをタスクイベントと呼び，アプリケーションに含まれるすべてのタスクのタスクイベントをアプリケーションイベントと呼ぶ．さらに，ある時刻において最も早く発生するタスクイベントをそのタスクの次タスクイベントと呼び，最も早く発生するアプリケーションイベントをそのアプリケーションの次アプリケーションイベントと呼ぶ．

アプリケーションは，高性能プロセッサにおけるプロセッサ利用率（これをシェアと呼ぶ）と，実行可能なプロセッサ時間（これをバジェットと呼ぶ）の 2 つのパラメータを持つ．

シェアは，高性能プロセッサに対する低性能プロセッサの相対性能を意味する．たとえば，統合後の高性能プロセッサ上で N 個のアプリケーションを動作させる場合，統合前に性能 σ_i の低性能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．

バジェットは，統合後の高性能プロセッサ上でそのアプリケーションを実行可能なプロセッサ時間を意味している．バジェットは，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減少し，別のアプリケーションに実行が切り替わると，その後に実行されたタスクの実行時間分のバジェットが，切り替わった先のアプリケーションのバジェットから減少する．

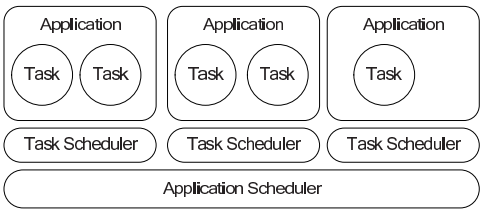

図 5 階層型スケジューラの構成
Fig. 5 Composition of Hierarchical Scheduler.

3.3 スケジューラ構成

提案アルゴリズムは，**図 5** のようにアプリケーションをスケジューリングするアプリケーションスケジューラと，アプリケーション内のタスクをスケジューリングするタスクスケジューラを階層的に配置した階層型スケジューラで実現する．

タスクスケジューラは，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．各アプリケーションのタスクは，アプリケーションごとに定められたプリエンプティブなスケジューリングアルゴリズムによりスケジューリングされる．タスクのスケジューリングは，低性能プロセッサと高性能プロセッサで同一のアルゴリズムを用いる．

アプリケーションスケジューラは，アプリケーションの実行順序を決定するスケジューリングと，アプリケーションに対するバジェットの割当ての 2 つの機能を持つ．

3.4 スケジューリングアルゴリズム

提案アルゴリズムのアプリケーションスケジューラは，バジェットが 0 でなく，次アプリケーションイベントが最も早いアプリケーションを最高優先順位アプリケーションとする．アプリケーションイベントでは，現在の時刻からそのアプリケーションの次アプリケーションイベントまでの時間とシェアの積をバジェットとして割り当てる．アプリケーションスケジューラは，実行中のアプリケーションのバジェットが 0 になると，次に優先順位の高いアプリケーションに実行を切り替える．

提案アルゴリズムでは，ハードウェアやリアルタイム OS に対して特別な機能を要求しない．たとえば，アプリケーションに対するバジェットの割当てや，バジェットが 0 になったときにアプリケーションを切り替

表 3 アプリケーション B のタスクセット
Table 3 Task set of application B.

	Period	Deadline	Maximum Execution Time	Utilization	Worst-case Response Time
Middle Priority Task	5	5	3	60%	3

える機能などは，多くのマイコンが備えるハードウェアタイマや割込みの機能を利用することで容易に実現できる．そのため，本研究の主な適用対象としている自動車制御システムの制御用コンピュータ（主に 16 ビットから 32 ビットのマイコンが多い）においても，リアルタイム OS のスケジューラを修正することで，本アルゴリズムを実装することは容易であると考えられる．

以下に，階層型スケジューラによる提案アルゴリズムの詳細を示す．ここでは，アプリケーション A_i（$i = 1, 2, \ldots, N$）のタスク T_{ij}（$j = 1, 2, \ldots, M$）の k 回目のジョブを J_{ijk}（$k = 1, 2, \ldots$）と表記する．

- 初期化
  - (1) すべてのアプリケーションのバジェットを 0 にする．
- タスク T_{ij} の k 回目のジョブ J_{ijk} が時刻 r_{ijk} でリリースされたときの動作
  - (1) タスクスケジューラはアプリケーション A_i 内のタスクをスケジューリングする．
  - (2) タスク T_{ij} の次タスクイベント NTE_{ij} をリリース時刻 r_{ijk} からデッドライン d_{ijk} に更新する．
  - (3) アプリケーション A_i の次アプリケーションイベント NAE_i を更新する．
  - (4) アプリケーション A_i のシェア S_i を用いて，アプリケーション A_i のバジェット B_i を以下の式を用いて更新する．
    $$B_i = (NAE_i - r_{ijk}) \times S_i$$
  - (5) アプリケーションスケジューラはアプリケーションをスケジューリングする．アプリケーション A_i が最高優先順位アプリケーションになった場合は，実行中アプリケーションのバジェットを保存する．
  - (6) 最高優先順位アプリケーションの最高優先順位タスクを実行する．
- ジョブ J_{ijk} の実行が完了したときの動作
  - (1) 次タスクイベント NTE_{ij} をジョブ J_{ijk} のデッドライン d_{ijk} から，次のジョブのリリース時刻 $r_{ij(k+1)}$ に更新する．
  - (2) 次アプリケーションイベント NAE_i を更新する．
  - (3) アプリケーション A_i に実行可能タスクがある場合は以下のように動作する．
    - (a) タスクスケジューラがタスクをスケジューリングする．
  - (4) アプリケーション A_i に実行可能タスクがなければ以下のように動作する
    - (a) バジェット B_i を 0 にする．
  - (5) アプリケーションスケジューラはアプリケーションをスケジューリングする．最高優先順位アプリケーションがアプリケーション A_i 以外のアプリケーションになった場合は，バジェット B_i を保存する．
  - (6) 最高優先順位アプリケーションの最高優先順位タスクを実行する．
- ジョブ J_{ijk} の実行中にバジェットが 0 になったときの動作
  - (1) ジョブ J_{ijk} の実行を中断する．
  - (2) アプリケーション A_i のバジェット B_i を保存する．
  - (3) アプリケーションスケジューラはアプリケーションをスケジューリングする．
  - (4) 最高優先順位アプリケーションの最高優先順位タスクを実行する．

3.5 動 作 例

提案アルゴリズムの動作を表 2 のアプリケーション A' と，**表 3** のアプリケーション B を高性能プロセッサ上に統合する例を用いて説明する．

まず，アプリケーション A' とアプリケーション B は，**図 6** に示すように，性能 1 の低性能プロセッサでそれぞれ時間制約を満たす．図には，アプリケーションごとの次アプリケーションイベントも示している．

次に，性能 2 の高性能プロセッサ上で，2 つのアプリケーションにそれぞれシェア 0.5 を設定し，提案アルゴリズムを適用すると，**図 7** に示すように，両アプリケーションともに時間制約を満たせる．

時刻 0 では，アプリケーション A' の次アプリケーションイベントは時刻 3 なので，バジェットは 1.5 単位時間となる．同様に，アプリケーション B の次アプリケーションイベントは時刻 5 なので，バジェットは 2.5 単位時間となる．このとき両アプリケーションともに実行可能タスクが存在するが，次アプリケーションイベントはアプリケーション A' の方が早いので，アプリケーション A' のタスクから実行される．アプ

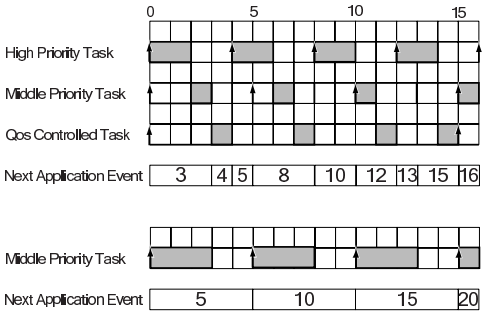

図 6 低性能プロセッサでのアプリケーションの動作
Fig. 6 Behavior of applications on a low performance processor.

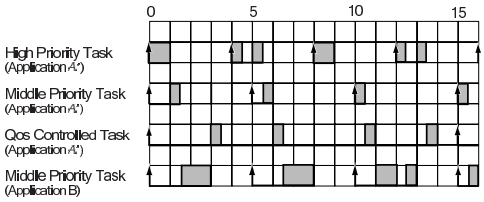

図 7 高性能プロセッサでのアプリケーションの動作
Fig. 7 Behavior of applications on a high performance processor.

リケーション A' は時刻 1.5 でバジェットを使い切り，次にアプリケーション B のタスクが実行される．

時刻 12 では，アプリケーション B のタスク実行中に，アプリケーション A' の高優先度タスクがリリースされると，アプリケーション A' の次アプリケーションイベント（時刻 13）が，アプリケーション B の次アプリケーションイベント（時刻 15）より早くなる．この場合は，アプリケーション B の実行を中断して残りバジェットを保存し，アプリケーション A' の高優先度タスクに実行が切り替わる．アプリケーション A' がバジェットを使い切ると，先に保存した残りバジェットを使ってアプリケーション B のタスク実行を再開する．このように，各アプリケーションがアプリケーションイベント間で利用可能なプロセッサ時間は制限されるため，プロセッサ利用率がシェアを超えて実行されることはない．その結果，アプリケーション間でプロセッサ時間は保護される．

提案アルゴリズムでは，タスクのデッドラインだけではなく，リリース時刻もアプリケーションイベントとして管理することで，QoS 制御されたタスクが，同じアプリケーション内のより優先度の高いタスクがリリースされる前に，バジェットを使ってしまうことを防ぐことが可能となる．その結果，PShED アルゴリズムを適用した場合には時刻 16 でデッドラインをミスしたアプリケーション A' の高優先度タスクは，提

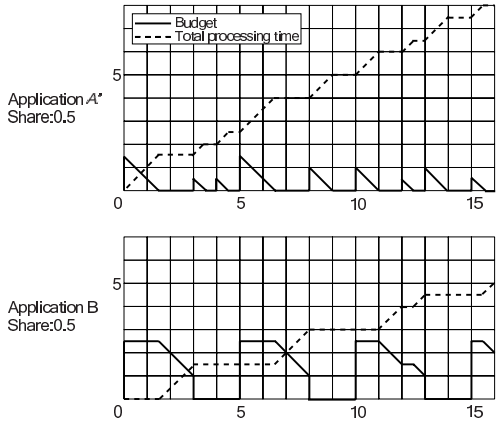

図 8 2 つのアプリケーションのバジェットの推移
Fig. 8 Budget transitions of applications.

案アルゴリズムを適用すると時間制約を満たせる．

また，図 6 と図 7 を比較すると，低性能プロセッサと高性能プロセッサでタスク実行順序が一致する．たとえば，時刻 0 から時刻 5 におけるアプリケーション A' のタスクの実行順序に着目すると，ともに高優先度タスク，中優先度タスク，低優先度タスク，高優先度タスクの順序で実行されている．このように，低性能プロセッサと高性能プロセッサでは，処理性能が異なるためにタスク実行の開始時刻は異なるが，実行順序は一致する．

図 8 はアプリケーションごとのバジェットと累積実行時間の推移を示している．統合前と統合後で，各アプリケーションが得られるアプリケーションイベント間の処理量は一致している．また，累積実行時間はアプリケーションイベントの時点で必ず一致している．たとえば，低性能プロセッサ上でのアプリケーション A' は，時刻 0 から時刻 10 の間で 10 単位時間を得ている．一方，高性能プロセッサでは，同時間内で 5 単位時間を得ており，性能差を考慮した処理量は一致している．

4 章では，タスクの実行順序が一致する性質と，アプリケーションイベント間で処理量が一致する性質をそれぞれ証明し，提案アルゴリズムが時間保護の定義を満たすことを示す．

4. 証　　明

提案アルゴリズムが 2.2 節で提示した時間保護の要件を満たすことを示す．まず，提案アルゴリズムによりスケジュールされるアプリケーションについて，以下の補題 1 が成り立つ．

補題 1 高性能プロセッサで動作するアプリケーションは，アプリケーションイベントで割り当てられたバ

ジェットを次アプリケーションイベントまでに使用できる．

（証明）提案アルゴリズムでは，アプリケーションイベントをそのアプリケーションのリリース時刻，次アプリケーションイベントをそのアプリケーションのデッドラインと考えると，アプリケーションの動作はタスクの動作と同様に扱うことができる．アプリケーションスケジューラは，次アプリケーションイベントをデッドラインとする EDF スケジューリングアルゴリズムであると考えると，アプリケーションのスケジュール可能性は，次のように示すことができる．

アプリケーション A_i（$i = 1, 2, \ldots, N$）のシェアは，高性能プロセッサの性能に対する A_i が動作する低性能プロセッサの性能 σ_i の割合と定義している．高性能プロセッサにおける A_i の最大プロセッサ利用率 U_i^f は，$\sigma_i / \sum_{j=1}^{N} \sigma_j$ となる．したがって，高性能プロセッサで動作するアプリケーションの最大プロセッサ利用率の合計は，以下のようになる．

$$\sum_{i=i}^{N} U_i^f = \sum_{i=1}^{N} (\sigma_i / \sum_{j=1}^{N} \sigma_j) = 1 \qquad (1)$$

プロセッサ利用率の合計が 1 以下となるタスクセットは，EDF スケジューリングアルゴリズムによりスケジューリング可能であることが知られており[3]，すべてのアプリケーションは提案アルゴリズムによりスケジュール可能である．よって，高性能プロセッサで動作するアプリケーションは，バジェットをアプリケーションイベント間で使用できる． □

さらに，以下の補題 2 と補題 3 が成り立つ．

補題 2 すべてのアプリケーションについて，アプリケーションイベント間の処理量は低性能プロセッサ上と高性能プロセッサ上で一致する．

（証明）バジェットの定義と補題 1 より，バジェットは高性能プロセッサにおけるアプリケーションイベント間での処理量の上限値であると考えることができる．2 つのアプリケーションイベント t_1 から t_2 における，プロセッサの性能差を考慮した処理量を $P(t_1, t_2)$ と表記すると，任意のアプリケーションイベント t において，低性能プロセッサの処理量 P^l と高性能プロセッサの処理量 P^h について以下が成り立つ．

$$P^l(0, t) \leq P^h(0, t) \qquad (2)$$

$P^l < P^h$ となる状況とは，低性能プロセッサ上に実行可能なタスクが存在せず，高性能プロセッサ上に実行可能なタスクが存在し，かつ，そのタスクが実行される場合である．そこで，時刻 t' で高性能プロセッサ上にこのような実行可能タスク T が存在すると仮定する．低性能プロセッサ上と高性能プロセッサ上では，タスクのリリース時刻が一致するので，時刻 t' までにリリースされるタスクも一致する．時刻 t' までに実行可能となるタスクが一致し，タスク T が高性能プロセッサ上で実行可能であるということは，時刻 0 から時刻 t' における低性能プロセッサの処理量が高性能プロセッサの処理量よりも多いことになる．すなわち，以下が成り立つ．

$$P^l(0, t') > P^h(0, t') \qquad (3)$$

これは明らかに式 (2) に矛盾するため，$P^l < P^h$ は成り立たない．よって，時刻 0 から任意のアプリケーションイベント t 間での処理量について以下が成り立つ．

$$P^l(0, t) = P^h(0, t) \qquad (4)$$

この関係は任意の 2 つのアプリケーションイベント t_1 から t_2 についてもいえるため，以下が成り立つ．

$$P^l(t_1, t_2) = P^h(t_1, t_2) \qquad (5)$$

以上より，すべてのアプリケーションについて，任意のアプリケーションイベント間の処理量は低性能プロセッサ上と高性能プロセッサ上で一致する． □

補題 3 タスクの実行順序は低性能プロセッサ上と高性能プロセッサ上で一致する．

（証明）補題 3 を示す方針として，両プロセッサ上でタスク切替えが発生する状況において，最高優先順位のタスクが必ず一致することを示す．提案アルゴリズムのタスクスケジューラはプリエンプティブなスケジューリングアルゴリズムを前提としているので，アプリケーション内のタスク切替えは，以下のいずれかの状況で発生する．

(1) 実行可能なタスクが存在しないときにタスクがリリースされる．
(2) 実行中のタスクよりも高優先度のタスクがリリースされる．
(3) 実行中のタスクの実行が終了する．

低性能プロセッサ上と高性能プロセッサ上ではタスクのリリース時刻が一致するため，(1) と (2) によるタスク切替えは同じ時刻で発生する．補題 2 より，アプリケーションイベントまでに実行可能になったタスクと，それらのタスクに対する処理量は両プロセッサで一致している．タスクスケジューラのスケジューリングアルゴリズムが両プロセッサで同じであるという前提より，実行したタスクとその順序も一致する．

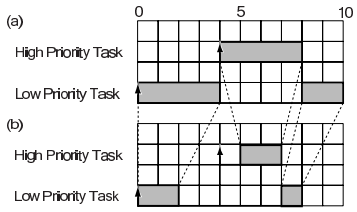

図 9 (a) 低性能プロセッサ上と，(b) 高性能プロセッサ上における同一アプリケーション内でのタスク切替え
Fig. 9 Task preemption. (a) On a low performance processor case; (b) On a high performance processor.

よって，アプリケーションイベントでは，実行可能なタスク（実行可能になった時刻も）とその優先順位は一致している．よって，低性能プロセッサ上でリリースされて最高優先順位となるタスクは，高性能プロセッサ上でも，アプリケーション内の最高優先順位タスクとなる．

(2) によりタスクがプリエンプトされる状況について具体的な例を用いて説明する．**図 9** は，(a) 性能 1 の低性能プロセッサ上と (b) 性能 2 の高性能プロセッサ上において，時刻 4 で同時にタスクがプリエンプトされる例である．高性能プロセッサの場合には，複数のアプリケーションが存在するため，アプリケーション内の最高優先順位タスクが切り替わっても即座に実行されるとは限らない．図の例では，高優先度タスクと低優先度タスクのデッドラインはともに時刻 10 とする．また，高性能プロセッサ上では，時刻 4 から時刻 5 の間に別のアプリケーションが実行されているものとする．

低優先度タスクリリースされる時刻 0 から，高優先度タスクがリリースされて低優先度タスクがプリエンプトされる時刻 4 までの間で，アプリケーションが得られる処理量は両プロセッサで一致している．したがって，低性能プロセッサ上でプリエンプトされるタスクは，高性能プロセッサ上でも必ずプリエンプトされるため，同一アプリケーション内でタスク切替えが発生する場合でもタスクの実行順序は一致する．

(3) はアプリケーションイベントの間で発生するためタスクの切替えタイミングは，両プロセッサ上で一致しない可能性がある．しかし，アプリケーションイベントでは実行可能なタスクとその優先順位が一致することから，実行中のタスクが終了した後に実行されるタスクは一致する．以上より，タスク切替えが発生する状況では，両プロセッサ上で同一のタスクが実行されることから，すべてのタスクの実行順序は一致するといえる． □

補題 2 と 3 より，以下の定理が導かれる．

定理 1 低性能プロセッサ上で時間制約を満たして実行可能なタスクは，高性能プロセッサ上でも時間制約を満たして実行可能である．

（証明） 補題 2 と補題 3 より，両プロセッサ上の任意のアプリケーションイベント間でタスクの実行順序と処理量が一致する．すると，任意のタスクのリリース時刻から，そのタスクの実行が終了した直後のアプリケーションイベントの間で実行されるすべてのタスクの実行順序と得られる処理量は一致することになる．よって，低性能プロセッサ上でリリース時刻からデッドラインまでに実行を完了するタスクは，高性能プロセッサ上でもデッドラインまでに実行を完了する．□

定理 1 により，低性能プロセッサで時間制約を満たせるタスクは高性能プロセッサ上でも時間制約を満たして実行可能であることが示された．よって，提案アルゴリズムは時間保護の要件を満たす．

5. 関 連 研 究

まず，複数の低性能プロセッサで動作するアプリケーションを高性能プロセッサに容易に統合することを目的とした過去の研究では，統合前の低性能プロセッサで時間制約を満たすタスクが統合後に時間制約を満たせなくなる要因を整理して，スケジューリングアルゴリズムが満たすべき要件を定義しているものは少ない．特に，本論文で指摘した QoS 制御されたタスクの存在を考慮している研究はない．

次に，本論文の提案アルゴリズムに関連する研究について議論する．

アプリケーションの QoS 制御を主な目的とした CPU リソース管理手法が数多く研究されている．たとえば，Oikawa らは CPU リソース管理機能を持つ小型のカーネル Resource Kernel を Linux に実装し，汎用 PC では無視できる程度のオーバヘッドで QoS 制御を実現した[7]．これらのリソース管理手法では，アプリケーションごとにプロセッサ時間を定率で割り当てる機能を実現できる．しかし，プロセッサ時間を割り当てるタイミングは周期的なものが多く，アプリケーション内のタスクが時間制約を満たすことを保証したものは少ない．また，QoS 制御されたタスクを考慮していないため，時間保護の要件を満たしていない．提案アルゴリズムでは，タスクのリリース時刻とデッドラインを用いてスケジューリングすることで時間保護の要件を満たしており，ハードリアルタイム性を要求されるシステムに対しては，リソース管理手法に比べて適用しやすいと考えられる．

リリース時刻が一定周期である周期タスクに対して

は，過去に提案されたサーバアルゴリズム[1),2),9),10)]でも，周期タスクの最大公約数の周期でサーバにプロセッサ時間を割り当てることで，サーバ上のすべてのタスクが時間制約を満たせると考えられる．しかし，QoS 制御されたタスクによるバジェットの先使いを考慮していないため，時間保護の要件を満たせない．

Lipari らは，タスクのリリース時刻が既知であることを前提とせず，デッドラインのみで，統合後の高性能プロセッサ上のすべての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ている[6)]．PShED アルゴリズムでは，本論文で整理した低性能プロセッサで時間制約を満たすタスクが高性能プロセッサで時間制約を満たせなくなる要因のうち，1 つ目と 2 つ目の要因を解決できると考えられるが，3 つ目の要因である QoS 制御されたタスクの存在を考慮していないため時間保護の要件を満たせない．

Deng らは，ハードリアルタイムシステム向けのアプリケーションを統合するための階層型スケジューリングアルゴリズムを提案している[4),5)]．この手法では，各タスクのリリース時刻と実行時間の上限値が既知であることを前提として，低性能プロセッサで時間制約を満たすタスクは高性能プロセッサでも時間制約を満たせることを証明しており，本論文で定義した時間保護の要件を結果的に満たしている．しかし，設計段階においてタスクの実行時間の上限値を正確に見積もることは容易ではなく，最大実行時間に代えてデッドラインを用いる提案アルゴリズムの方が現実のシステムに適用しやすいと考えられる．

6. おわりに

本論文では，ハードリアルタイム性を要求される組み込みシステムを対象として，複数のアプリケーションを単一のプロセッサ上に容易に統合するための時間保護機能と，それを実現するスケジューリングアルゴリムを提案した．

時間保護機能を実現するスケジューリングアルゴリズムの検討にあたり，まず低性能プロセッサで時間制約を満たすタスクが高性能プロセッサで時間制約を満たせなくなる要因を 3 つに整理した．3 つ目の要因は，QoS 制御されたタスクの存在を指摘するもので，現実の自動車制御システムの設計ではこのようなタスクを考慮する必要がある．次に，時間保護機能の定義として，3 つの要因を解決するためにスケジューリングアルゴリズムが満たすべき要件を定義した．そして，時間保護機能を実現し，かつアプリケーションの設計段階で適用しやすいアルゴリズムを提案した．提案アルゴリズムは，Deng らのアルゴリズムを修正したもので，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前提として時間保機能を実現するものである．最後に，提案アルゴリズムが時間保護の要件を満たすことを証明した．

今後の課題として，まず提案アルゴリズムを既存のリアルタイム OS へ実装し，スケジューリングやタスク切替えのオーバヘッドを測定して，提案手法の有効性を定量的に評価する必要がある．実装対象としては，オープンソースの組み込みシステム向けリアルタイム OS である，μITRON 仕様準拠の TOPPERS/JSP カーネル[8)] や OSEK/VDX 仕様準拠の TOPPERS/OSEK カーネル[8)] などを検討している．また，現状のアルゴリズムでは，タスクの実行終了時にアプリケーション内に実行可能なタスクがなければ，そのアプリケーションのバジェットは 0 にするが，余ったバジェットを別のアプリケーションが利用することで，プロセッサ利用率を向上させる手法などを検討していく．

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エア創造事業の援助を受けた．

参考文献

1) Abeni, L. and Buttazzo, G.: Integrating multimedia applications in hard real-time systems, *Proc. IEEE Real-Time Systems Symposium* (Dec. 1998).
2) Abeni, L. and Buttazzo, G.: Resource reservations in dynamic real-time systems, *Real-Time Systems*, Vol.27, pp.123–165 (2004).
3) Buttazzo, G.: *HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications*, 2nd Edition (2005).
4) Deng, Z., Liu, J.W.-S., Zhang, L., Mouna, S. and Frei, A.: An open environment for real-time applications, *Real-Time Systems*, Vol.16, pp.155–185 (1999).
5) Deng, Z., Liu, J.W.-S. and Sun, J.: A scheme for scheduling hard real-time applications in open system environment, *9th Euromicro Workshop on Real-Time Systems* (1997).
6) Lipari, G., Carpenter, J. and Baruah, S.: A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments, *IEEE Real-time Systems Symposium* (2000).
7) Oikawa, S. and Rjkumar, R.: Portable RK: A portable resource kernel for guaranteed and enforced timing behavior, *IEEE Real Time Tech-*

nology and Applications Symposium (1999).
8) TOPPERS Project. http://www.toppers.jp
9) Spuri, M. and Buttazzo, G.: Scheduling aperiodic tasks in dynamic priority systems, *Real-Time Systems Journal*, Vol.10, pp.179–210 (1996).
10) Strosnider, J., Lehoczky, J.P. and Sha, L.: The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments, *IEEE Trans. Comput.*, Vol.44 (1995).

(平成 18 年 10 月 10 日受付)
(平成 19 年 3 月 2 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．組み込みシステム向けのリアルタイム OS 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．

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）．

高田　広章（正会員）
名古屋大学大学院情報科学研究科情報システム学専攻教授．1988 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同専攻助手，豊橋技術科学大学情報工学系助教授等を経て，2003 年より現職．2006 年より大学院情報科学研究科附属組込みシステム研究センター長を兼務．リアルタイム OS，リアルタイムスケジューリング理論，組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．