情報処理学会論文誌 Vol. 52 No. 8 2387–2401 (Aug. 2011)

時間保護のためのタスク起動遅延付き階層型スケジューリングアルゴリズム

松原 豊[†1] 本田晋也[†1] 高田広章[†1,†2]

分散リアルタイムシステムにおいて，個別に開発・検証された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して動作させるための階層型スケジューリングアルゴリズムが数多く提案されている．本論文では，統合前に，プリエンプティブな固定優先度ベーススケジューリングによりスケジュール可能なリアルタイムアプリケーションを対象に，優先度設計を変更することなく統合後もスケジュール可能であることを保証する階層型スケジューリングアルゴリズムを提案する．提案アルゴリズムの正当性を理論的に証明し，さらに，スケジューリングシミュレータを用いて，同一のアプリケーションに対するスケジュール可能性を従来アルゴリズムと比較した．その結果，従来アルゴリズムでは統合後にデッドラインをミスしてしまうアプリケーションが，提案アルゴリズムによりスケジュール可能であることを確認した．

Hierarchical Scheduling with Delayed Activation of Tasks for Temporal Protection

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

Many hierarchical scheduling algorithms have been proposed for integrating independently developed real-time applications into one processor. This paper presents a new hierarchical scheduling based on the Bandwidth Sharing Server (BSS). The presented algorithm supports fixed-priority local scheduler with delayed activation of tasks. Simulation results indicate that if an application is schedulable with fixed-priorities, then the application is in isolation also schedulable without modification of priority of each tasks when it is integrated with other applications into the same processor by using the proposed scheduling algorithm.

1. はじめに

代表的な分散リアルタイムシステムの 1 つである自動車制御システムには，走行性能の向上や運転支援を目的として，数多くの機能が搭載されるようになっている．現在は，ECU（Electronic Control Units；電子制御ユニット）と，ECU 上で動作する制御系リアルタイムアプリケーションをサプライヤが一体で開発し，自動車メーカがそれらを組み合わせることで，複雑な自動車制御システムを実現している．システムに新しい機能を追加する場合，センサやアクチュエータへのインタフェースと，それらを制御するマイコンが一体になった ECU を追加する必要がある．

近年，自動車制御システムの高性能化・高機能化により，1 台の自動車に搭載される ECU の数が急激に増加している．ECU が増加し，自動車制御システムの複雑さが増すと，個別に開発・検証された制御機能が，システムに組み込まれた後にも，要求される時間制約を満たして正常に動作することを検証する（結合検証と呼ぶ）ための工数が非常に多くなるという問題がある．また，ハードウェアコストの増加や，ECU 自体を搭載するためのスペースが不足するという問題も引き起こしている．

システムに搭載される ECU の数を削減するためのアプローチとして，個別に開発・検証されたリアルタイムアプリケーションを単一の ECU に統合して動作させる（アプリケーション統合と呼ぶ）ための手法が検討されている．アプリケーション統合を容易に実現するためには，先に述べた結合検証を容易に実施できることが望ましい．具体的には，制御機能を実現する複数のアプリケーションを統合する場合に，単体で時間制約を満たして動作することが確認できているアプリケーションについて，タスク構成や優先度設計を変更することなく，統合した ECU でも時間制約を満たして動作することが保証できると，結合検証の負担を大幅に減らすことができる．本研究では，この性質を実現するために，アプリケーションごとのプロセッサ時間を保護する（時間保護と呼ぶ）機能を提案し，統合 ECU のリアルタイム OS に適用するスケジューリングアルゴリズムの確立を目的としている．

時間保護を実現できるスケジューリングアルゴリズムとして，統合するアプリケーションのタスクをスケジューリングするローカルスケジューラと，アプリケーションをスケジュー

†1 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
†2 名古屋大学大学院情報科学研究科
Graduate School of Information Science, Nagoya University

リングするグローバルスケジューラを，階層的に配置した階層型スケジューリングが数多く提案されている．Lipari らの BSS（Bandwidth Sharing Server）[1] と，それを改良した PShED（Processor Sharing with Earliest Deadline）[7] は，ローカルスケジューリングとグローバルスケジューリングの両方に EDF（Earliest Deadline First）を用いる場合に，時間保護できることを保証している．しかしながら，ローカルスケジューリングにプリエンプティブな固定優先度スケジューリング（FPS と呼ぶ）を用いる場合には，時間保護を実現することができないことが知られている[8]．一般的なリアルタイム OS は FPS でタスクをスケジュールするものが多いため，既存のアプリケーションを BSS や PShED で統合してスケジュールすると，統合後のアプリケーションの振舞いが統合前と変わってしまう．実際の車載制御アプリケーションでは，実行時の処理オーバヘッドを削減するために，タスクの優先度関係において，低い優先度を持つタスクには実行を妨げられないことが分かっている場合に，高い優先度を持つタスクでは明示的な排他制御をしていないことがある．このようなアプリケーションを統合後に EDF でスケジュールすると，タスクの時間制約は満たすことはできても，排他制御が崩れてしまう．したがって，統合前に FPS で動作検証されたアプリケーションを統合する場合は，統合後にもタスク間の実行優先度関係を維持してスケジュールする必要がある．

階層型スケジューリングのグローバルスケジューリングに EDF を，ローカルスケジューリングに FPS を採用する場合でも時間保護を実現できるアルゴリズムとして，Deng らの Open System で用いられるアルゴリズム[10] や，これを拡張した，松原らの時間保護アルゴリズム[11] が提案されている．これらの手法を適用するためには，タスクの起動時刻が必要であるため，タスクの正確な起動時刻をあらかじめ知ることのできないアプリケーションに対しては適用できないという問題がある．

Shin らや Lipari らは，アプリケーションの実行要求に対して，プロセッサ時間を周期的に供給するリソースモデルを構築する手法を提案している[14),15]．この手法では，アプリケーションのリソースモデル (P, Q) を設計段階で構築し，実行時には，周期 P でプロセッサ時間 Q を割り当てることで，アプリケーション内のタスクがデッドラインを満たすことを保証できる．動作が単純であるためにスケジューリングのための処理オーバヘッドが小さいことと，実行時にタスクの起動時刻の情報を必要としないという利点がある．しかしながら，リソースモデルを構築する段階では，各タスクの起動周期（もしくは最小到着間隔），最悪実行時間，相対デッドラインの情報が必要である．加えて，リソースモデルのプロセッサ利用率（$\frac{Q}{P}$ で計算する）は，アプリケーションのタスクが必要とするプロセッサ利用率を合計した値よりも，数十%高く設定する必要がある[⋆1]．したがって，プロセッサ資源が限られる組み込みリアルタイムシステムには不向きである．

本論文では，アルゴリズムの適用条件の少ない BSS をベースに，統合前に FPS によりスケジュール可能なリアルタイムアプリケーションの優先度設計を変更することなく統合できる階層型スケジューリングアルゴリズムを提案する．階層型スケジューリングのグローバルスケジューリングに EDF を，ローカルスケジューリングに FPS を採用する場合でも時間保護を実現できる従来アルゴリズムに対する，提案アルゴリズムの優位性は，次の 2 つである．1 つ目は，アルゴリズムを適用するために必要なパラメータがタスクの相対デッドラインのみであるため，文献 10)，11) の手法に比較して制約が少なく，適用できるアプリケーションの対象が広いことである．2 つ目は，アプリケーションに含まれるタスクのプロセッサ利用率の合計値を超えるプロセッサ利用率を，アプリケーションに設定する必要がないため，プロセッサを効率的に利用できることである．たとえば，タスクのプロセッサ利用率（最小起動間隔に対する最悪実行時間の割合）の合計が 50%のアプリケーションに対して，文献 14)，15) の手法では，アプリケーションを周期実行する抽象化のために，数十%を加算したプロセッサ利用率を設定する必要がある．それに対して，提案アルゴリズムでは，50%を設定すれば，理論上，アプリケーションをスケジュール可能であることを証明している．プロセッサを効率的に利用できると，より多くのアプリケーションを統合できることや，性能の限られた組み込みシステムに適用できるという利点がある．

本論文では，提案アルゴリズムにより，統合後にすべてのタスクがスケジュール可能であることを理論的に証明し，さらに，スケジューリングシミュレータを用いて，BSS とスケジュール可能性を比較する．その結果，BSS では統合後にデッドラインをミスしてしまうアプリケーションを，提案アルゴリズムによりスケジュール可能であることを確認する．

以下，本論文の構成について述べる．まず，2 章で，関連研究について述べる．3 章で，アプリケーション統合におけるシステムモデルを整理する．4 章では，従来手法の問題点を指摘し，5 章で，提案アルゴリズムについて述べる．6 章では，提案アルゴリズムの正当性を証明する．7 章では，スケジューリングシミュレータを用いて，BSS とスケジュール可能性を比較する．最後に，8 章で，結論と今後の課題について述べる．

⋆1 必要となる追加プロセッサ利用率は，タスクの構成に依存する[15]．

2. 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[2)–5)]．これらのアルゴリズムは，もともと周期タスクのリアルタイム性保証と，非周期タスクの応答性の改善を目的としており，これらの手法をアプリケーション統合に適用して各タスクのリアルタイム性を保証するためには，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．したがって，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する必要がある．サーバアルゴリズムにおいて，1 つのサーバに複数のタスクを割り当てることにより，これらのプロセッサ利用率を保証することは可能である．加えて，割り当てられたタスクの周期の最大公約数をサーバの周期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デッドラインを満たせるようスケジュールできる．しかしながら，すべてのタスクの周期の最大公約数が非常に小さな値になる場合には，タスクの切替えが頻発するという問題がある．

Lipari らの BSS（Bandwidth Sharing Server）[1)] と，それを改良した PShED（Processor Sharing with Earliest Deadline）[7)] は，階層型スケジューリングにおいて，ローカルスケジューリングとグローバルスケジューリングの両方に EDF を用いる場合に，時間保護できることを保証している．しかしながら，ローカルスケジューリングに FPS を用いる場合には，時間保護を実現することができないことが知られている[8)]．

階層型スケジューリングのグローバルスケジューリングに EDF を，ローカルスケジューリングに FPS を採用する場合でも時間保護を実現できるアルゴリズムとして，Deng らの Open System [10)] で用いられるアルゴリズムや，これを拡張した，松原らの時間保護アルゴリズム[11)] が提案されている．これらの手法では，タスクの相対デッドラインに加えて起動時刻が必要であるため，タスクの正確な起動時刻をあらかじめ知ることができないアプリケーションに対して適用することができない．時間保護アルゴリズムでは，オープンソースのリアルタイム OS をベースに実装した研究がある[12)]．

Shin らや Lipari らは，アプリケーションの要求するプロセッサ時間を，モデル化するリソースモデルを提案し，そのリソースモデルを周期的に実行することで，アプリケーション内のタスクがデッドラインを満たす手法を提案している[14),15)]．これらの手法は，各アプリケーションを周期的に実行させるので実行時にタスクの起動時刻を必要としないが，起動周期を設計する段階では，各タスクの起動周期（もしくは最小到着間隔），最悪実行時間，相対デッドラインが必要である．また，アプリケーションに設定するプロセッサ利用率を，各タスクが必要とするプロセッサ利用率を合計した正味のプロセッサ利用率に比べて 30%程度高く設定する必要がある場合もある[15)]．したがって，プロセッサ資源が限られる組み込みリアルタイムシステムには向かない．リソースモデルを実装した研究としては，Kerstan らの仮想マシンモニタに適用した実装[16)] や，航空機制御システムの IMA（Integrated Modular Avionics）向け基盤ソフトウェアの標準規格である ARINC 653 [6)] に適用した実装[17)] がある．

3. システムモデル

本研究では，自動車制御システムで動作するリアルタイムアプリケーションを 1 つの高性能なプロセッサに統合することを想定する．自動車制御システムに搭載される ECU には，動作周波数が数十 MHz から数百 MHz の組み込みシステム向けのプロセッサが搭載されており，エンジン制御，ボディー制御，ドア制御などの制御系アプリケーションが動作している．このような制御アプリケーションを，数百 MHz から高くても 1 GHz 程度のプロセッサに統合することを目標とする．

分散リアルタイムシステムにおけるアプリケーション統合のシステムモデルを**図 1** に示す．独立に動作するコンピュータシステム（個別プロセッサと呼ぶ）上でデッドラインを満

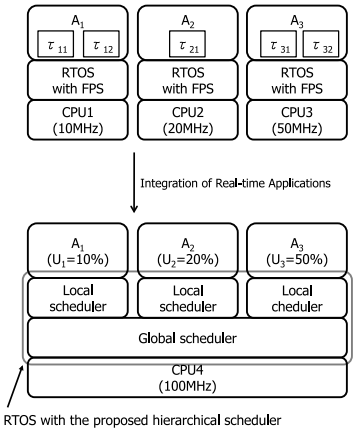

図 1　階層型スケジューリングによるリアルタイムアプリケーションの統合
Fig. 1　Integration of real-time applications using hierarchical scheduling.

たして動作するリアルタイムアプリケーションを，高性能なコンピュータシステム（統合プロセッサと呼ぶ）に統合して動作させる．統合プロセッサはシングルプロセッサである．なお，本論文では，アプリケーションの実行が別のアプリケーションに待たされるような同期通信は存在しないものとする．

本論文では，統合後に各アプリケーションがデッドライン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提案する．統合前に単独で動作する状況において，各アプリケーションがスケジュール可能であることを保証する手法については特に言及しないが，たとえば，レートモノトニック解析による理論的解析手法を用いて保証することが考えられる．この手法では，スケジュール可能性を検証する際にタスクの相対デッドラインのみではなく，タスクの最悪実行時間を測定して，最悪時におけるスケジュール可能性を検証しておく必要がある．

統合プロセッサでは，N 個のアプリケーション $A_1, A_2, \ldots, A_n$ を統合して動作させる．アプリケーション A_i は，2 項組 (U_i, d_i) で表される．ここに，U_i はシステム構築時に設定されるプロセッサ利用率，d_i はアプリケーションの絶対デッドライン（アプリケーションデッドラインと呼ぶ）である．アプリケーションデッドラインは，アプリケーションに属する，実行可能なタスクの絶対デッドラインの中で最も早い時刻であり，システムの動作中に変化する．

A_i は，タスクの集合 $\tau_i = \{\tau_{i1}, \tau_{i2}, \ldots\}$ である．本論文では，タスクの構成は設計時に決定され，動作中には変化しない．タスク τ_{ij} は，4 項組 $(P_{ij}, D_{ij}, C_{ij}, d_{ij})$ で表される．ここに，P_{ij} は実行優先度，D_{ij} は相対デッドラインであり，これらの値はタスクの設計時に決定される．C_{ij} は，性能 1 を持つ統合プロセッサにおける，タスクの最悪実行時間であり，定数である．プロセッサ性能とタスク処理時間の関係は単純化する．すなわち，タスクの処理時間は，プロセッサ性能に反比例するものとする．たとえば，性能 U_i を持つ個別プロセッサでの最悪実行時間は，$\frac{C_{ij}}{U_i}$ になる．C_{ij} は，6 章で述べる，アルゴリズムの正当性証明でのみ使用するパラメータであり，タスクをスケジュールする際には使用しない．d_{ij} は，タスクの絶対デッドラインで，システムの動作中に，タスクの起動時刻に相対デッドライン D_{ij} を加算して計算する．

本論文では，アプリケーション統合において，以下のことを前提とする．

(1)　アプリケーションはタスクのみで構成される．すなわち，割込み処理は考慮しない．

(2)　イベント待ちやセマフォ解放など，タスクが待ち状態になるようなタスク間同期通信はない．

(3)　アプリケーションには，実行前に正確な起動時刻を知ることのできないタスクが存在する．

(4)　タスクの優先度は，デッドライン・モノトニックにより割り当てられる．すなわち，τ_{ij} と τ_{ik} の相対デッドラインが，$D_{ij} < D_{ik}$ の関係にあるとき，τ_{ij} は，τ_{ik} より高い優先度（$P_{ij} > P_{ik}$）を持つものとする．

(5)　アプリケーション A_i は，統合プロセッサの性能を 1 とする相対性能 U_i を持つ個別プロセッサで FPS によりスケジュール可能である．すわなち，A_i が個別プロセッサでデッドラインを満たして実行可能であることは，たとえば，レートモノトニック解析による理論的解析手法を用いて保証されているものとする．

(6)　統合するアプリケーションに設定するプロセッサ利用率の合計は，1 以下とする．

これらの前提のうち，(1)，(2)，(5)，(6) は，リアルタイムアプリケーション統合に関する従来研究でも置かれている前提である．(3) と (4) は，提案アルゴリズムにおいて追加した前提である．文献 10)，11) の手法では，すべてのタスクが周期タスクであるという前提を置くことで，タスクの起動時刻を実行前に予測可能としているが，提案アルゴリズムではその前提を緩めていることから (3) の前提を追加している．したがって，(3) は制約条件ではないことに注意されたい．(4) は，提案アルゴリズムの正当性を容易に証明するために追加した前提である．実際の制御アプリケーション設計においても，タスク優先度をデッドラインモノトニックで決定する場合があるため，この前提が提案アルゴリズムの有効性を低下させるものではないと考える．

現実の自動車制御アプリケーションの中には，前提の (1)〜(6) にあてはまるアプリケーションが存在するが，その一方で，提案手法を適用できないアプリケーションも存在する．具体的には，割込み処理が多数存在するアプリケーション（前提 (1) にあてはまらない）や，アプリケーション間で同期通信をするアプリケーション（前提 (2) にあてはまらない）である．

4. BSS の問題点

BSS（および，それを改良した PShED）は，統合プロセッサでのアプリケーションの振舞いに関して，次の 2 つの性質を保証している．

- 性質 1：プロセッサ利用率の保証
  グローバルスケジューリングアルゴリズムとして EDF を採用すると，各アプリケーションは，設定されたプロセッサ利用率に基づくプロセッサ時間（バジェットと呼ぶ）

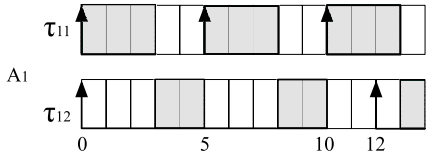

図 2　FPS によるアプリケーションのスケジューリング
Fig. 2　A scheduling of A_1 by FPS.

が割り当てられ，アプリケーションデッドラインまでに使いきることができる．

- 性質 2：タスクのスケジュール可能性の保証（時間保護の実現）
  各アプリケーションのタスクについて，統合前に，EDF によりスケジュール可能なタスクは，ローカルスケジューリングアルゴリズムとして EDF を採用する BSS により統合後もスケジュール可能である．

性質 1 は，アプリケーション間でプロセッサ時間が保護されることを保証している．しかし，アプリケーションに属するタスクがそれぞれのデッドラインを満たすこと（すなわち，時間保護を実現できること）は保証していない．このことを保証するのが性質 2 であるが，BSS の場合は，ローカルスケジューリングアルゴリズムが EDF であることを前提としており，一般的なリアルタイム OS で採用されている FPS には対応していない．

このことを簡単な例を用いて示す．いま，周期 5，（相対性能 0.5 の個別プロセッサにおける）最悪実行時間 3 の高優先度タスク τ_{11} と周期 12，最悪実行時間 4 の低優先度タスク τ_{12} の 2 タスクで構成されるアプリケーション A_1 と，周期 12，最悪実行時間 12 の 1 タスクで構成されるアプリケーション A_2 を性能 1 のプロセッサに統合することを考える．どちらのアプリケーションも，個別プロセッサでスケジュール可能で，A_1 に着目すると個別プロセッサでは**図 2** に示すようにスケジュールされる．ここで，A_1 と A_2 にそれぞれプロセッサ利用率を 50%割り当て，ローカルスケジューリングアルゴリズムに FPS を用いた BSS（BSS-FPS と呼ぶ）でスケジュールする．**図 3** に示すように，A_1 の τ_{12} は時刻 12 でデッドラインをミスしてしまう．これは，A_1 と A_2 の 2 つのアプリケーションが動作した結果，τ_{12} が，τ_{11} の 3 回目の起動の前に実行を完了できなかったことが原因である．時刻 12 までに各アプリケーションが使用したバジェットを考えると，A_1 と A_2 はともに 6 単位時間であり，プロセッサ利用率は 50%である．

得られたプロセッサ利用率は十分であるにもかかわらず，A_1 のタスクがデッドラインをミスしてしまった原因を考えると，τ_{11} の 3 回目の起動要求が発生したときのアプリケーションデッドライン（すなわち，τ_{12} の絶対デッドラインである時刻 12）より遅いデッドライン（時刻 15）を持つ高優先度タスク τ_{11} が，低優先度タスク τ_{12} を実行するためのバジェットを使って，低優先度タスクに優先して実行されたと考えることができる．

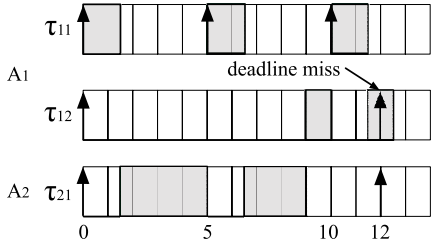

図 3　BSS-FPS によるアプリケーションのスケジューリング
Fig. 3　A scheduling of A_1 and A_2 by BSS with FPS local schedulers.

5. 提案アルゴリズム

5.1 提案アルゴリズムの概要

提案アルゴリズムでは，タスクが起動した時点で，その起動時刻から絶対デッドラインまでの間で，アプリケーションに設定されたプロセッサ利用率分の時間だけプロセッサ時間を獲得できるようにバジェットを管理する．

統合前にスケジュール可能であるという前提から，すべてのタスクは，起動時刻から絶対デッドラインまでに実行を完了できるはずである．統合後の実行時間は，プロセッサ利用率分だけ短くなるため，先のバジェット管理手法により，すべてのタスクに対して統合前と同等以上のプロセッサ時間を割り当てることで，スケジュール可能性を保証できる．

ただし，プロセッサ時間は，タスクごとではなく，アプリケーション全体で管理する．そのため，3 章で述べたように，同じアプリケーションの別タスクがプロセッサ時間を先に使ってしまうという現象が発生する．提案アルゴリズムでは，高優先度タスクの起動遅延の仕組みを導入することでこれを防ぐ．

5.2 提案アルゴリズムが満たす性質

提案アルゴリズムは，ローカルスケジューリングにおいて，アプリケーションのデッドラインより遅い絶対デッドラインを持つ高優先度タスクの起動時刻を遅延させる仕組みを導入することで，低優先度タスクがデッドラインをミスすることを防ぐアルゴリズムである．優先度の高いタスクの起動を遅延させて，実行中の低い優先度を持つタスクを，処理が完了

するまで優先的に実行するため，厳密な優先度ベーススケジューリングではない．

提案アルゴリズムは，優先度ベーススケジューリングの重要な性質の 1 つである，「高い優先度を持つタスクが，その実行中に低い優先度を持つタスクに邪魔されることはない」という性質を満たすことができる．実際の車載アプリケーションには，この性質を利用し，低い優先度を持つタスクには実行を妨げられないことを前提として，高い優先度を持つタスクでは（セマフォやミューテックスなどを用いて）明示的な排他制御をせずに，データの排他制御を実現している場合がある．このようなアプリケーションを，提案アルゴリズムを適用して統合しても，統合後に排他制御が崩れてしまうことはない．この点で，提案アルゴリズムの実用性は高いといえる．

そのほかにも，優先度ベーススケジューリングの性質としては，「アプリケーション内で最も高い優先度を持つタスクは，起動と同時に実行を開始し，実行を完了するまで他のタスクに実行を邪魔されることはないこと」や，「実行中に，あるタスクの実行時間が一時的に延びた場合に，実行時間が延長したタスクと同じか，より低い優先度を持つタスクがデッドラインをミスする」という性質がある．これらの性質は，提案アルゴリズムおよび BSS-FPS では満たすことができない．

このように，BSS-FPS と提案アルゴリズムでは，「高い優先度を持つタスクが，その実行中に低い優先度を持つタスクに邪魔されることはない」という性質を満たすことができる．しかし，それ以外の性質はどちらも満たすことができない．BSS-FPS と提案アルゴリズムで満たすことのできない性質が，実際のアプリケーションで要求されるかどうかについては，アプリケーションごとの要求，設計に強く依存する．提案アルゴリズムでは，少なくとも，BSS-FPS と同等の性質を満たすことができることから，提案アルゴリズムの有用性は高いと考える．

5.3　タスクモデル

提案アルゴリズムにおけるタスクは，**図 4** に示すように 4 つの状態がある．タスクにはイベント待ちやセマフォ解放など同期通信がないことを前提としているため待ち状態はない．システムの動作開始時には，すべてのタスクは T_DORMANT 状態である．タスクの起動要求が発生すると，そのタスクが起動遅延条件（詳細は，5.6 節で述べる）を満たすかどうかをチェックする．起動遅延条件を満たさない場合は，タスクは T_READY 状態に遷移する（図 4 (1)）．一方，起動遅延条件を満たす場合には，タスクは T_DELAYED 状態に遷移する（図 4 (2)）．T_DELAYED 状態のタスクは，起動遅延条件を満たさなくなった時点で T_READY 状態に遷移する（図 4 (3)）．T_READY 状態で最も優先度の高いタスクは

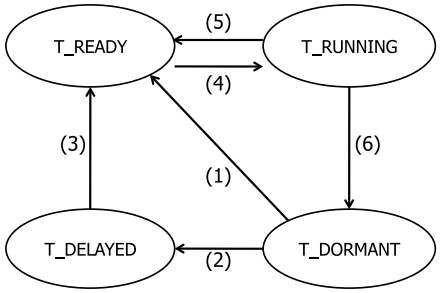

図 4　タスクの状態遷移図
Fig. 4　State transition diagram of a task.

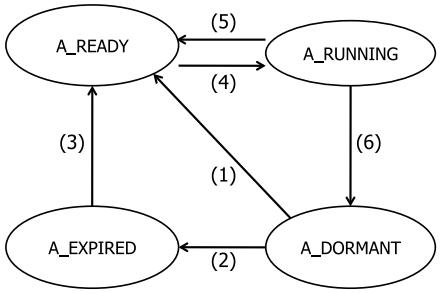

図 5　アプリケーションの状態遷移図
Fig. 5　State transition diagram of an application.

T_RUNNING 状態になる（図 4 (4)）．T_RUNNING 状態のタスクより高い優先度を持つタスクが T_READY 状態になると，T_RUNNING 状態のタスクは T_READY 状態になり（図 4 (5)），新しく起動した高優先度タスクが T_RUNNING 状態に遷移する．T_RUNNING 状態のタスクの実行が完了すると，T_RUNNING 状態から T_DORMANT 状態に遷移する（図 4 (6)）．

5.4　アプリケーションモデル

アプリケーションには，**図 5** に示すように 4 つの状態がある．アプリケーションの状態は，そのアプリケーションに属するタスクの状態とバジェットの量で決まる．まず，システムの動作開始時などで，実行できる状態（T_READY 状態もしくは T_RUNNING 状態）のタスクが存在しないアプリケーションは A_DORMANT 状態になる．A_DORMANT 状態のアプリケーションに属するタスクのうち 1 つでも T_READY 状態になり，かつ，バ

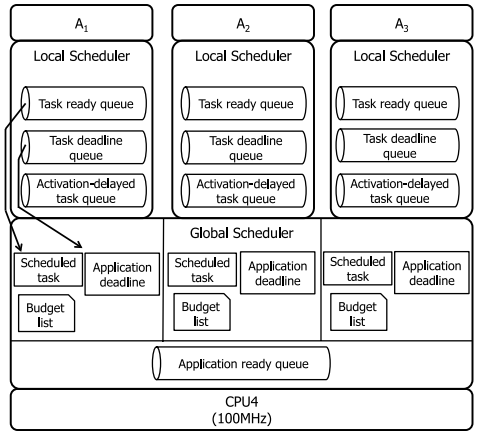

図 6　提案する階層型スケジューラの構成
Fig. 6　Construction of the proposed hierarchical scheduler.

ジェットが 0 でない場合，アプリケーションは A_READY 状態に遷移する（図 5 (1)）．一方，T_READY 状態のタスクが存在しても，バジェットが 0 の場合，アプリケーションは A_DORMANT 状態から A_EXPIRED 状態に遷移する（図 5 (2)）．A_EXPIRED 状態のアプリケーションにバジェットが割り当てられると，A_READY 状態に遷移する（図 5 (3)）．A_READY 状態のアプリケーションの中で最も絶対デッドラインの早いアプリケーションは，A_RUNNING 状態に遷移する（図 5 (4)）．A_RUNNING 状態のアプリケーションより，絶対デッドラインの早いアプリケーションが A_READY 状態になると，A_RUNNING 状態のアプリケーションは A_READY 状態になり（図 5 (5)），新しく A_READY 状態になったアプリケーションが A_RUNNING 状態に遷移する．A_RUNNING 状態のアプリケーションの T_RUNNING 状態のタスクが実行された結果，実行できるタスクがなくなった場合には，アプリケーションは A_RUNNING 状態から A_DORMANT 状態に遷移する（図 5 (6)）．

5.5　スケジューラ構成

提案アルゴリズムは，**図 6** に示すように，アプリケーションに属するタスクをスケジュールするローカルスケジューラと，どのアプリケーションのタスクをプロセッサで実行するかを決定する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リングアルゴリズムである．

ローカルスケジューラは，FPS にタスク起動遅延の仕組みを導入したアルゴリズムでタスクをスケジュールする．ローカルスケジューラでは，次の 3 つのキューを管理する．

- タスクデッドラインキュー（Task deadline queue）
  絶対デッドラインが設定されている状態（すなわち，T_DELAYED 状態，T_READY 状態，T_RUNNING 状態のいずれかの状態）のタスクが，絶対デッドラインの早い順に接続されたキューである．タスクの起動要求が発生した時点で接続される．
- タスクレディキュー（Task ready queue）
  実行できる状態（すなわち，T_READY 状態または T_RUNNING 状態）のタスクが，優先度の高い順に接続されたキューである．優先度が同じ場合は，FIFO 順に接続される．
- タスク起動遅延キュー（Activation-delayed task queue）
  タスクの起動遅延条件を満たしている（すなわち，T_DELAYED 状態）のタスクが，FIFO 順に接続されたキューである．

グローバルスケジューラは，EDF でアプリケーションをスケジュールし，最も早い絶対デッドラインを持つアプリケーションの T_RUNNING 状態のタスクを実行する．また，アプリケーションごとに使用できるバジェットを，バジェットリストで管理する（詳細は，5.7 節で述べる）．あるアプリケーションの絶対デッドラインが変わると，すべてのアプリケーションを再スケジュールする．加えて，新しい絶対デッドラインに対して使用できるバジェットを，そのアプリケーションに設定されたプロセッサ利用率を用いて計算する．グローバルスケジューラは，その時点での絶対デッドラインに対して割り当てられているバジェットを上限として，アプリケーションのタスクを実行する．アプリケーションのタスクを実行すると，その実行時間分だけバジェットを減らす．バジェットが 0 になると，次に絶対デッドラインの早いアプリケーションに，強制的に，実行を切り替える．

グローバルスケジューラでは，アプリケーションごとに次のデータ構造を管理する．

- 実行すべきタスク（Scheduled task）
  ローカルスケジューラでスケジュールされた T_RUNNING 状態のタスクである．アプリケーションがスケジュールされると，ここに格納されたタスクがプロセッサで実行される．
- アプリケーションレディーキュー（Application ready queue）
  実行できる状態（すなわち，A_READY 状態または A_RUNNING 状態）のアプリケーションが，絶対デッドラインの早い順番に接続されたキューである．アプリケーションが実行できる状態に遷移した時点で接続される．

- アプリケーションデッドライン（Application deadline）
  アプリケーションに属するタスクの絶対デッドラインの中で最も早い時刻（すなわち，ローカルスケジューラのタスクデッドラインキューの先頭に接続されたタスクの絶対デッドライン）である．グローバルスケジューラは，ここに格納された絶対デッドラインを用いて，アプリケーションをスケジュールする．
- バジェットリスト（Budget list）
  アプリケーションのバジェットを管理するためのデータ構造である．詳細は，5.7 節で述べる．

5.6　スケジューリングアルゴリズム

まず，タスクの起動要求が発生したときの流れを説明する．ここでは，時刻 t に，アプリケーション A_i のタスク τ_{ij} の起動要求が発生したとする．

(1)　τ_{ij} の相対デッドライン D_{ij} を用いて，次の計算式で絶対デッドライン d_{ij} を計算し，τ_{ij} をタスクデッドラインキューに挿入する．

$$d_{ij} = D_{ij} + t$$

(2)　τ_{ij} がタスクデッドラインキューの先頭に接続された場合，A_i の新しいデッドライン d_{ij} をグローバルスケジューラに通知する．グローバルスケジューラは，必要に応じて d_{ij} に対応するバジェット要素をバジェットリストに追加する．その後，アプリケーションの絶対デッドライン d_i を d_{ij} に更新する．

(3)　τ_{ij} と同じアプリケーションに属し，かつ時刻 t においてタスクレディキューに接続されたタスク τ_{ik} に対して，τ_{ij} が次の起動遅延条件を満たすかどうかをチェックする．
- τ_{ij} の優先度が τ_{ik} の優先度より高い（$P_{ij} > P_{ik}$）．
- τ_{ij} の絶対デッドラインが τ_{ik} の絶対デッドラインより遅い（$d_{ij} > d_{ik}$）．

(4)　τ_{ij} が起動遅延条件を満たす τ_{ik} が 1 つでも存在する場合は，τ_{ij} をタスク起動遅延キューの最後尾に挿入する．一方，τ_{ij} が起動遅延条件を満たさない場合は，τ_{ij} をタスクレディーキューに挿入する．

(5)　τ_{ij} がタスクレディーキューの先頭に接続された場合には，アプリケーション内で実行するタスクを切り替えるため，グローバルスケジューラに τ_{ij} を通知する．グローバルスケジューラは，アプリケーションの実行すべきタスクを τ_{ij} に更新する．

(6)　グローバルスケジューラは，絶対デッドラインの最も早いアプリケーションの実行すべきタスクをプロセッサで実行する．

次に，タスクの実行が中断もしくは完了したときの流れを説明する．ここでは，タスク τ_{ij} が時間 e だけ実行されたものとする．

(1)　τ_{ij} の属する A_i のバジェットリストを更新する．τ_{ij} の実行が完了した場合には，以降の (2) から (5) を実行する．

(2)　τ_{ij} をタスクデッドラインキューとタスクレディーキューから削除する．

(3)　タスク起動遅延キューの先頭に接続されているタスクから順に，そのタスク（τ_{ik} とする）が起動遅延条件を満たすかどうかをチェックする．

(4)　τ_{ik} が起動遅延条件を満たす場合は，そのままタスク起動遅延キューに接続しておく．起動遅延条件を満たさない場合は，τ_{ik} をタスク起動遅延キューから削除し，タスクレディーキューに挿入する．

(5)　同様に，τ_{ik} の次に接続されたタスクが起動遅延条件を満たすかどうかをチェックする．タスク起動遅延キューに接続されたすべてのタスクについて，起動遅延条件をチェックしたら終了する．

5.7　バジェット管理アルゴリズム

提案アルゴリズムでは，グローバルスケジューラでアプリケーションのバジェットを管理するために，バジェットリストを用いる．このアルゴリズムは，アプリケーションが使用できるバジェットを計算するもので，本質的には，BSS のバジェット管理アルゴリズム[1] と違いはない．

A_i のバジェットリストは，バジェット要素 $l_{ij} = (d_{ij}, b_{ij})$ で構成される．ここに，d_{ij} はバジェットの絶対デッドライン，b_{ij} は d_{ij} までのバジェットである．すなわち，l_{ij} は，A_i が d_{ij} までに b_{ij} のバジェットを使用できることを意味する．バジェットリスト内では，バジェット要素が絶対デッドラインの早い順に並ぶ．ここでは，バジェットリストに対する 3 つの操作を定義する．

バジェット要素の追加

A_i のローカルスケジューラが，グローバルスケジューラに対してアプリケーションのデッドライン d_{ij} を通知した時点（時刻 t とする）で，d_{ij} に一致するデッドラインを持つバジェット要素がバジェットリスト中に存在しない場合，次の処理により新しい要素 l_{ij} をバジェットリストに追加する．

(1)　次の条件を満たす，l_{ij} の挿入位置を探す．

$$\exists l_{i(k-1)}, l_{ik}\ d_{i(k-1)} < d_{ij} < d_{ik}$$

(2)　d_{ij} までに使用できるバジェット b_{ij} を次の式で計算する．A_i のデッドライン d_i が早い時刻になる場合（$d_i > d_{ij}$）と，遅い時刻になる場合（$d_i < d_{ij}$）とで計算式が異

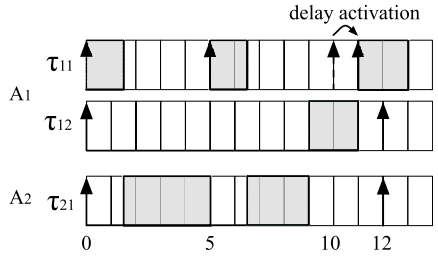

図 7　提案アルゴリズムによるスケジューリング
Fig. 7　A scheduling of A_1 and A_2 by the proposed scheduling algorithm.

なる．

$$b_{ij} = \begin{cases} \min\{D_i * U_i, (d_{ij} - d_{i(k-1)}) * U_i + b_{i(k-1)}, b_{ik}\} & (d_i > d_{ij}) \\ \min\{(d_{ij} - d_{i(k-1)}) * U_i + b_{i(k-1)}, b_{ik}\} & (d_i < d_{ij}) \end{cases}$$

（3）　l_{ij} を，$l_{i(k-1)}$ と l_{ik} の間に挿入する．

バジェットリストの更新

次の事象が発生したとき，バジェットリストを更新する．

- タスクの実行を完了したとき
- アプリケーションのバジェットが 0 になったとき
- アプリケーション内で実行するタスクを切り替えるとき
- 実行するアプリケーションを切り替えるとき

タスクを実行した時間を e，アプリケーションの絶対デッドラインに対応するバジェット要素 l_{ij} がバジェットリストの j 番目にあるとすると，次の処理を実行する．

- $b_{ik} \leftarrow b_{ik} - e$ $(k \geq j)$
- l_{ik} を削除する $(k < j \wedge b_{ik} > b_{ij})$．

バジェット要素の削除

バジェットリストのバジェット要素のうち，今後のバジェット計算で使用されない要素はバジェットリストから削除できる．時刻 t で，デッドラインが d_{ik} に一致するタスクがすでに完了しており，かつ，次のいずれかの条件を満たすとき，その要素 l_{ik} を削除できる[1)]．

- $d_{ik} \leq t$
- $b_{ik} > (d_{ik} - t)U_i$

5.8 動 作 例

図 3 のアプリケーションを提案アルゴリズムによりスケジュールした例を**図 7** に示す．時刻 10 で発生した高優先度タスク τ_{11} の起動要求は，実行中のタスク τ_{12} に対して起動遅延条件を満たすため，τ_{11} の起動は τ_{12} が完了するまで遅延する．その結果，τ_{12} は τ_{11} に優先して実行され，デッドラインである時刻 12 までに完了できる．また，起動が遅延した τ_{11} も，デッドラインである時刻 15 までに実行を完了できる．

6. 証　　明

提案アルゴリズムの正当性を証明するために，BSS の証明手法[8)] と同様に，4 章で述べた 2 つの性質が成り立つことを示す．性質 1 の証明は，グローバルスケジューリングアルゴリズムとバジェット管理手法に依存し，ローカルスケジューリングアルゴリズムには依存しない．そのため，BSS の証明方法をそのまま適用できる．そこで，本論文では，ローカルスケジューリングにタスク起動遅延の仕組みを導入した FPS を適用する場合に，性質 2 を満たせることを証明する．

まず，ある時間内におけるアプリケーションの処理要求に関して次の定義をする．

定義 1　区間 $[a, b]$ におけるアプリケーション A_i の処理要求を次のように定義する．

$$D_i(a, b) = \sum_{\substack{a_{ij} \geq a \\ d_{ij} \leq b}} C_{ij} \tag{1}$$

この処理要求の定義を用いて，アプリケーションに属するタスクのスケジュール可能性について，次の定理が成り立つ．

定理 1　性能 1 の統合プロセッサにおいてプロセッサ利用率 U_i を設定したアプリケーション A_i が次の式を満たすとき，A_i は提案アルゴリズムによりスケジュール可能である．

$$\forall a, b \; a < b \; D_i(a, b) \leq (b - a)U_i \tag{2}$$

（証明） 定理 1 の式 (2) を満たす状況において，A_i のタスクがデッドラインをミスすると仮定して矛盾を導く．いま，時刻 t_{ov} で A_i のバジェット l_{ij} が 0 になったとする．すなわち，バジェットリストには，$l_{ij} = (d_{ij}, 0)$ が存在する．ここで，次の条件を満たすバジェットリスト中の要素 $l_{ik} = (d_{ik}, b_{ik})$ について考える．

- $k \geq j$
- $b_{ik} = 0$

いい換えると，l_{ik} は，バジェットリストの中でバジェットが 0 である，最もデッドライン

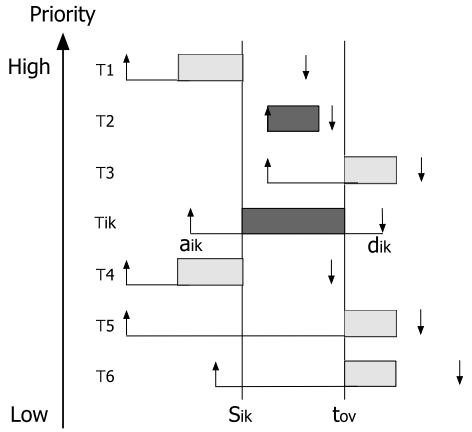

図 8　バジェット b_{ik} を使用して実行するタスク
Fig. 8　Tasks executed by using b_{ik}.

の遅いバジェット要素である．このとき，バジェットリストは次のようになっている．

$l_{i(k-1)} = (d_{i(k-1)}, b_{i(k-1)}) = (d_{i(k-1)}, 0)$
$l_{ik} = (d_{ik}, b_{ik}) = (d_{ik}, 0)$
$l_{i(k+1)} = (d_{i(k+1)}, b_{i(k+1)}) \leftarrow (d_{i(k+1)} \neq 0)$

l_{ik} がバジェットリストに追加された時刻を s_{ik} とすると，s_{ik} の時点で d_{ik} は最も早いデッドラインである．時刻 s_{ik} では，提案アルゴリズムのバジェット管理アルゴリズムにより，d_{ik} に対するバジェット b_{ik} が計算され割り当てられる．ここで，b_{ik} を使用して実行されるタスクの条件を考えると，**図 8** に示すように，$[s_{ik}, t_{ov}]$ で実行されるタスクの中で最も早く起動したタスクの起動時刻を a_{ik} とすると，a_{ik} より後に起動し，かつデッドラインが d_{ik} より早いタスクに限定される．その理由は次のとおりである．すなわち，タスク起動遅延付きの FPS では，τ_1 や τ_4 のようなタスクは τ_{ik} に優先して実行され，s_{ik} より早く完了するため，b_{ik} を使用して実行されることはない．また，τ_3，τ_5，τ_6 のようなタスクは，τ_{ik} が完了するまでは実行されず，バジェット b_{ik} を使用して実行されることはない．さらに，タスクの優先度がデッドラインモノトニックで割り当てられることを前提としていることから，τ_{ik} より優先度が高いかつ相対デッドラインが長いタスク，および，τ_{ik} より優先度が低いかつ相対デッドラインが短いタスクは存在しない．したがって，b_{ik} を使用して実行するタスクは，τ_2 と τ_{ik} 自身のように，a_{ik} より後に起動し，かつデッドラインが d_{ik} より早いタスクである．

時刻 t_{ov} でバジェット b_{ik} は 0 になるという前提から，b_{ik} を使用して実行するタスクの合計実行時間は b_{ik} を超えることになる．すなわち，以下が成立する．

$$\sum_{\substack{a_{ij} \geq a_{ik} \\ d_{ij} \leq d_{ik}}} C_{ij} > b_{ik} \tag{3}$$

ここで，b_{ik} の値は計算方法により，次の 3 つの場合に分かれる．それぞれの場合に，式 (3) が成立したときに矛盾が発生することを示す．

- 場合 A：$b_{ik} = b_{i(k+1)}$
  b_{ik} として，l_{ik} の次の要素の残りバジェット $b_{i(k+1)}$ が割り当てられる場合である．この場合，b_{ik} が 0 になった時点で，$b_{i(k+1)}$ も 0 になっているはずである．しかし，l_{ik} は，バジェットが 0 で絶対デッドラインの最も遅いバジェット要素であるから，この状況は発生しない．
- 場合 B：$b_{ik} = D_{ik} * U_i$
  s_{ik} の時点で，相対デッドライン D_{ik} に対するプロセッサ利用率分のバジェットが得られた場合である．このとき，次の式が成り立つ．

  $$\sum_{\substack{a_{ij} \geq a_{ik} \\ d_{ij} \leq d_{ik}}} C_{ij} > (d_{ik} - a_{ik})U_i$$

  これは，明らかに式 (2) に矛盾する．
- 場合 C：$b_{ik} = b_{i(k-1)} + (d_{ik} - d_{i(k-1)})U_i$
  l_{ik} の前のバジェット要素のバジェット $b_{i(k-1)}$ に依存する場合である．時刻 $s_{i(k-1)}$ で，l_{ik} の前の要素 $l_{i(k-1)}$ が挿入されたとする．$[s_{i(k-1)}, s_{ik}]$ の間に起動したタスクを実行した結果，s_{ik} の時点で，バジェット $b_{i(k-1)}$ を次のように更新する．

  $$b_{i(k-1)} \leftarrow b_{i(k-1)} - \sum_{\substack{a_{ij} \geq a_{i(k-1)} \\ d_{ij} \leq d_{i(k-1)}}} C_{ij}$$

  よって，バジェット b_{ik} は，次のように計算される．

$$b_{ik} = b_{i(k-1)} - \sum_{\substack{a_{ij} \geq a_{i(k-1)} \\ d_{ij} \leq d_{i(k-1)}}} C_{ij} + (d_{ik} - d_{i(k-1)})U_i$$

$b_{i(k-1)}$ について，さらに以下の 3 つの場合に分かれる．

– 場合 C-A：$b_{i(k-1)} = b_{ik}$
　場合 A と同様の理由で，このような状況はありえない．

– 場合 C-B：$b_{i(k-1)} = (d_{i(k-1)} - a_{i(k-1)})U_i$

$$\sum_{\substack{a_{ij} < a_{ik} \\ d_{ij} \leq d_{ik}}} C_{ij} > b_{ik} = b_{i(k-1)} - \sum_{\substack{a_{ij} \geq a_{i(k-1)} \\ d_{ij} \leq d_{i(k-1)}}} C_{ij} + (d_{ik} - d_{i(k-1)})U_i$$

$$= (d_{i(k-1)} - a_{i(k-1)})U_i - \sum_{\substack{a_{ij} \geq a_{i(k-1)} \\ d_{ij} \leq d_{i(k-1)}}} C_{ij} + (d_{ik} - d_{i(k-1)})U_i$$

$$\sum_{\substack{a_{ij} \geq a_{i(k-1)} \\ d_{ij} \leq d_{ik}}} C_{ij} > (d_{ik} - a_{i(k-1)})U_i$$

　となる．これは，$a = a_{i(k-1)}$，$b = d_{ik}$ と考えると，式 (2) に矛盾する．

– 場合 C-C：$b_{i(k-1)} = b_{i(k-2)}$
　さらに，前のバジェット $b_{i(k-2)}$ の計算方法により 3 つの場合に分かれるが，これまでと同様に $[a_{i(k-2)}, d_{ik}]$ の区間を考えると，場合 A もしくは場合 B のときには矛盾が発生する．場合 C が続くと，最終的には，タスクの最初の起動時刻までさかのぼることになるが，このときは，場合 B に該当するので，やはり式 (2) に矛盾する．

以上より，すべての場合で矛盾が発生することから，定理 1 は成立する．　□

次に，統合前の前提について，次の定理が成り立つ．

定理 2　個別プロセッサでアプリケーションが FPS でスケジュール可能であるとき，次の式を満たす．

$$\forall a, b \; a \leq b \; \frac{D_i(a,b)}{U_i} \leq (b-a) \tag{4}$$

（証明） 式 (4) は，EDF によるスケジュール可能性の必要十分条件であることが証明されている[1]．また，EDF は最適なスケジューリングアルゴリズムであることが知られており[9]，FPS でスケジュール可能なアプリケーションは EDF でもスケジュール可能である．よって，FPS でスケジュール可能なアプリケーションは式 (4) 満たす．以上より，定理 2 が証明された．　□

最後に，定理 1 と定理 2 より，次の定理 3 が成り立つ．

定理 3　性能 U_i の個別プロセッサでアプリケーション A_i が FPS（デッドライン・モノトニックによる優先度割当てを前提）によりスケジュール可能であるとき，性能 1 の統合プロセッサにおいてプロセッサ利用率 U_i を設定すれば，アプリケーション A_i のタスクは提案アルゴリズムによりスケジュール可能である．

（証明） 定理 2 の式 (4) より，次の式が成り立つ．

$$\forall \; a, b \; a \leq b \; \sum_{\substack{a_{ij} \geq a \\ d_{ij} \leq b}} \frac{C_{ij}}{U_A} \leq (b-a)$$

$$\forall \; a, b \; a \leq b \; \sum_{\substack{a_{ij} \geq a \\ d_{ij} \leq b}} C_{ij} \leq (b-a)U_i$$

よって，個別プロセッサで FPS によりスケジュール可能なアプリケーション A_i は定理 1 を満たすため，性能 1 の統合プロセッサにおいてプロセッサ利用率 U_i を設定すれば，提案アルゴリズムにより A_i はスケジュール可能である．以上より，定理 3 が証明された．　□

7. 評　　価

6 章で述べた証明の正当性を裏付けることと，4 章で述べた，BSS-FPS におけるデッドラインミスがどの程度の割合で発生するのかを明らかにすることにより，それを改善した提案アルゴリズムの有用性を確認する．従来研究には，提案アルゴリズムと適用条件が同じで，かつ，追加のプロセッサ利用率を必要とせずに効率的にプロセッサを使用できるという性質を持つアルゴリズムが存在しない．したがって，評価対象としては，できる限り提案アルゴリズムに近いアルゴリズムとして，提案アルゴリズムと同じ前提条件を持つ BSS-FPS を選択した．

我々は，タスクスケジューリングアルゴリズムの動作検証および評価を目的として，タスクスケジューリングに特化したスケジューリングシミュレータを開発している[13]．このスケジューリングシミュレータに提案アルゴリズムと BSS-FPS を実装して，単独での動作する場合に FPS でスケジュール可能なアプリケーションを複数統合し，スケジュール可能な

表 1　評価条件
Table 1　Evaluation conditions.

評価番号	テストベンチアプリケーション数	タスクの起動属性	平均プロセッサ利用率	平均タスク数
1	1	周期	89.49	4.08
2	1	非周期	89.49	4.08
3	1	非周期	96.61	8.45
4	3	非周期	94.03	3.80

表 2　シミュレーション結果
Table 2　Simulation results.

評価番号	評価対象アプリケーション数	スケジュール可能なアプリケーション数	
		BSS-FPS	提案アルゴリズム
1	10,000	8,330	10,000
2	10,000	9,355	10,000
3	10,000	6,142	10,000
4	1,000	978	1,000

アプリケーションの数を比較した．提案アルゴリズムにおける，タスクの切替え回数や，タスクの応答時間などの性能評価は今後の課題とする．

シミュレーションによる評価の流れは，次のとおりである．まず，スケジュール可能性を判定する評価対象のアプリケーションとして，単独で，FPS によりスケジュール可能なアプリケーションを生成する．評価対象アプリケーションは複数のタスクで構成される．シミュレーションに必要となる，タスクの起動周期と最悪実行時間のパラメータは，一様分布に従う乱数で生成した．相対デッドラインは，次の起動時刻に一致するものとした．次に，評価対象アプリケーションとテストベンチアプリケーションに対してプロセッサ利用率を割り当て，BSS-FPS と提案アルゴリズムの 2 つのアルゴリズムを用いてスケジューリングする．ここで，テストベンチアプリケーションとは，評価対象アプリケーションがスケジュールされるタイミングを，さまざまに変化させることを目的としたアプリケーションである．タスクは 1 つだけ存在し，このタスクの相対デッドラインを，一様分布に従う乱数で実行時に決定する．これにより，テストベンチアプリケーションの絶対デッドラインがさまざまに変化するため，評価対象アプリケーションが先に実行される状況や，逆に，評価対象アプリケーションに優先してテストベンチアプリケーションが実行される状況を作り出すことができる．最後に，生成したアプリケーション数に対するスケジュール可能なアプリケーション数（スケジュール可能率と呼ぶ）を比較した．

実施した 4 つの評価の条件設定を**表 1** に，シミュレーション結果を**表 2** に示す．評価 1 では，周期タスクのみで構成される，2 つのアプリを統合するという，最も基本的な状況下でスケジュール可能性を確認する．現実のアプリケーションには，非周期タスクが含まれることも多い．そこで，評価 2 では，提案アルゴリズムが周期タスクだけでなく，非周期タスクが存在するアプリケーションにも適用できることを確認するために，評価 1 のタスクを非周期タスクに変更したアプリケーションを用いる．評価 3 では，よりタスク数が多く，かつプロセッサ利用率の高いアプリケーションのスケジュール可能性を確認するため，評価 2 の条件に対して，各タスクのプロセッサ利用率を下げ，アプリケーション全体としては，タスク数が増加するようにした．評価 4 では，2 つより多いアプリケーションを統合する状況で評価するために，評価 2 の条件に対して，アプリケーション数を 4 つに増やした．これらの比較により，評価 1 の単純な状況に加えて，評価 2〜4 のさまざまな状況においても，提案アルゴリズムの性質が満たされることを明らかにする．

評価 1

評価 1 として，周期タスクのみで構成され，デッドラインモノトニックで優先度を割り当てた評価対象アプリケーションを 10,000 個生成した．アプリケーションを構成するタスクの起動周期は [10,50] の範囲の整数値，最悪実行時間は，[1,10] の範囲の整数値である．これらのアプリケーションは，FPS によりスケジュール可能であることを確認している．評価対象アプリケーションと，テストベンチアプリケーション 1 つの合計 2 つのアプリケーションを個別プロセッサの 2 倍の性能を持つ統合プロセッサで統合し，各アプリケーションのプロセッサ利用率を 50% に設定した．シミュレーション時間は，10,000 単位時間である．シミュレーションの結果，BSS-FPS のスケジュール可能率は約 83%であるのに対して，提案アルゴリズムでは 100%であった．すなわち，BSS-FPS では約 17%のアプリケーションにおいて，デッドラインミスが発生したことになる．この結果から，提案アルゴリズムは BSS-FPS に対して，スケジュール可能性を約 17%向上させることができた．

評価 2

評価 2 として，評価対象アプリケーションを構成するタスクを非周期タスクに変更してシミュレーションした．テストベンチアプリケーションの数，タスクのパラメータの範囲は，評価 1 と同様である．非周期タスクの起動間隔は周期タスクとは異なり，実行前に予測できないように，シミュレーションの実行時に次の式を用いて計算する．

$$p - \frac{\log(1 - rand())}{\lambda}$$

ここに，第 1 項 p は，タスクの起動周期として生成した値であり，タスク起動の最小起動間隔を意味する．第 2 項は，指数分布に従う乱数であり，最小到着間隔からの増加時間を意味する．$rand()$ は，[0,1] の範囲の実数をランダムに生成する関数である．λ は 4 に固定しており，最小到着間隔からの増加時間の平均は 2.5 単位時間，分散は 6.25 である．評価 2 で使用した評価対象アプリケーションの平均タスク数と平均プロセッサ利用率は評価 1 と同程度である．なお，プロセッサ利用率は，すべての非周期タスクが，最小到着間隔で起動した場合の値である．シミュレーションの結果，BSS-FPS のスケジュール可能率は約 93%であるのに対し，提案アルゴリズムは 100%であった．評価対象アプリケーションのプロセッサ利用率が評価 1 と同じであるにもかかわらず，BSS-FPS のスケジュール可能率が評価 1 に比べて高くなった理由としては，最小到着間隔に基づいて計算したプロセッサ利用率よりも，実際のシミュレーション時に動作するアプリケーションのプロセッサ利用率は低くなったことで，BSS-FPS でスケジュール可能なアプリケーション数が増加したと考えられる．それに対して提案アルゴリズムでは，すべてのアプリケーションをスケジュールすることができていることから，非周期タスクを含むアプリケーションに対しても，統合後のスケジュール可能性を保証できることを確認した．

評価 3

評価 3 として，評価対象アプリケーションを構成するタスク数を増やして，プロセッサ利用率を高めて評価するため，タスクを生成する際に用いるパラメータを変更してシミュレーションした．具体的には，アプリケーションを構成するタスクの起動周期の範囲を [20,50] に変更し，最悪実行時間の範囲を [1,4] に変更した．この変更により，個々のタスクのプロセッサ利用率は低下するが，アプリケーションあたりのタスク数が増加することで，アプリケーションのプロセッサ利用率を高めた．シミュレーションの結果，BSS-FPS のスケジュール可能率は約 62%に対して，提案アルゴリズムでは 100%であった．BSS-FPS のスケジュール可能率が評価 2 に比べて 21%程度低くなった理由としては，評価 2 に比べて，評価対象アプリケーションを構成するタスクの起動周期が長くなったことにより，4 章で述べた，BSS-FPS におけるデッドラインミスの状況が発生しやすくなったためと考えられる．提案アルゴリズムでは，タスク起動遅延の仕組みの効果により，すべての評価対象アプリケーションをスケジュールできている．

評価 4

評価 4 として，3 つ以上のアプリケーションを統合する状況で評価するため，評価 2 で用いた評価対象アプリケーションに対して，テストベンチアプリケーションを 3 つ統合した．評価 1 と評価 2 の場合と同様に，アプリケーションを構成するタスクの起動周期の範囲は [10,50]，最悪実行時間の範囲は [1,10] である．シミュレーション時間を 10,000 から 100,000 単位時間と，10 倍に増やした．合計 4 つのアプリケーションを個別プロセッサの 4 倍の性能を持つ統合プロセッサで統合し，各アプリケーションのプロセッサ利用率を 25%に設定した．シミュレーションの結果，BSS-FPS のスケジュール可能率は約 97%に対して，提案アルゴリズムでは 100%であった．BSS-FPS のスケジュール可能率が評価 2 に比べて高くなった理由としては，次の 2 つの要因により，4 章で述べた，BSS-FPS におけるデッドラインミスの状況が発生しにくくなったためと考えられる．1 つ目の要因は，アプリケーションの数が増えたことで，アプリケーション切替えの回数が増加したことである．2 つ目の要因は，統合プロセッサの性能が個別プロセッサの 4 倍になったことで，タスクの実行時間が相対的に 4 分の 1 に短くなり，低優先度タスクが高優先度タスクに邪魔される前に，実行を完了できる可能性が高まってことである．この結果から，提案アルゴリズムでは，4 つのアプリケーションを統合する場合においても，評価対象アプリケーションをスケジュール可能であることを確認した．

以上より，提案アルゴリズムの最も重要な性質である，統合後のスケジュール可能性を明らかにし，6 章の証明の正当性を確認した．さらに，BSS-FPS のスケジュール可能性も明らかにすることで，提案アルゴリズムの有用性を確認した．

8. おわりに

本論文では，統合前に FPS でスケジュール可能なリアルタイムアプリケーションを，その優先度設計を変更することなく統合できる階層型スケジューリングアルゴリズムを提案した．提案アルゴリズムにより，統合後にすべてのタスクがスケジュール可能であることを証明し，さらに，スケジューリングシミュレータを用いて，従来アルゴリズムとスケジュール可能性を比較した．その結果，従来アルゴリズムでは統合後にデッドラインをミスしてしまうアプリケーションを，提案アルゴリズムによりスケジュール可能であることを確認した．

今後の課題として，まず，提案アルゴリズムをリアルタイム OS に実装し，スケジューリングのオーバヘッドを評価することがあげられる．次に，実用性を評価するために，現実の車載アプリケーションに適用する．発展的な課題としては，適用条件を緩めてアルゴリズムの適用範囲を広げるために，待ち状態になるタスクをサポートすることや，アプリケーション間での同期通信を実現する仕組みを検討する．

参 考 文 献

1) Lipari, G. and Baruah, S.: Efficient Scheduling of Real-Time Multi-Task Applications in Dynamic Systems, *Proc. IEEE Real-Time Technology and Applications Symposium* (2000).
2) Spuri, M. and Buttazzo, G.: Scheduling aperiodic tasks in dynamic priority systems, *Real-Time Systems Journal*, Vol.10, pp.179–210 (1996).
3) 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).
4) Abeni, L. and Buttazzo, G.: Integrating multimedia applications in hard real-time systems, *Proc. IEEE Real-Time Systems Symposium* (1998).
5) Abeni, L. and Buttazzo, G.: Resource reservations in dynamic real-time systems, *Real-Time Systems*, Vol.27, pp.123–165 (2004).
6) Airlines Electronic Engineering Committee (ARINC): AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE PART 1 – REQUIRED SERVICES (2006).
7) 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).
8) Lipari, G.: Resource Reservation in Real-Time Systems, Ph.D. Thesis, Scuola Superiore S. Anna, Pisa, Italy (2000).
9) 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).
10) 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).
11) 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).
12) 松原　豊，本田晋也，冨山宏之，高田広章：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，情報処理学会論文誌組込みシステム工学特集，Vol.49, pp.3508–3519 (2008).
13) 佐野泰正，松原　豊，本田晋也，高田広章：リアルタイムアプリケーション向けタスク処理定義可能なスケジューリングシミュレータ，組込み技術とネットワークに関するワークショップ（ETNET2011），Vol.2010-EMB-20 (2011).
14) Lipari, G. and Bini, E.: A methodology for designing hierarchical scheduling systems, *Journal fo Embedded Computing*, Cenbridge International Science Publishing (2003).
15) Shin, I. and Lee, I.: Compositional Real-time Scheduling Framework with Periodic Model, *ACM Trans. Embedded Computing Systems*, Vol.7, No.3, Article 30 (2008).
16) Kerstan, T., Baldin, D. and Groesbrink, S.: Full Virtualization of real-time systems by temporal partitioning, *Proc. IEEE 6th International Workshop on Operating Systems Platform for Embedded Real-Time Applications*, pp.24–25 (2010).
17) Easwaran, A., Lee, I., Sokolsky, O. and Vestal, S.: A Compositional Scheduling Framework for Digital Avionics Systems, *Proc. IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (2009).

(平成 22 年 10 月 8 日受付)
(平成 23 年 5 月 14 日採録)

松原　豊

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研究員．2006 年名古屋大学大学院情報科学研究科博士前期課程修了．2009 年同博士後期課程単位取得満期退学．2009 年 4 月より現職．リアルタイム OS，リアルタイムスケジューリング理論に関する研究に従事．博士（情報科学）．

本田　晋也（正会員）

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助教．現在，同准教授．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，日本ソフトウェア科学会各会員．

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．