情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2010-EMB-19 No.3
2010/12/6

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確認するため，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

近年，代表的な分散リアルタイムシステムのひとつである自動車制御システムの高性能化・高機能化により，一台の自動車に搭載される ECU（Electronic Control Units; 電子制御ユニット）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事なく，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]，および，それを改良した PShED（Processor Sharing with Earliest Deadline）[2] 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

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

© 2010 Information Processing Society of Japan

することができない[3]．一般的なリアルタイム OS では，FPS でタスクをスケジュールするものが多く，そこで動作するアプリケーションを BSS や PShED で統合スケジュールすると，統合後のアプリケーションの振る舞いが統合前と変わってしまう．実際のアプリケーションでは，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も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，高い優先度をも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せるが，排他制御が崩れてしまうことになる．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ことが望ましい．

本論文では，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．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する．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する．

以下，本論文の構成について述べる．まず 2 章で，アプリケーション統合におけるシステムモデルを整理する．3 章では，従来手法の問題点を指摘し，4 章で，提案アルゴリズムについて述べる．5 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．6 章で関連研究について述べ，7 章で結論と今後の課題について述べる．

2. システムモデル

本研究では，図 1 に示すように，分散リアルタイムシステムにおいて，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ことを想定する．

統合プロセッサはシングルプロセッサで，N 個のアプリケーション $A_1, A_2, ..., A_n$ を動作させる．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もっとも早い時刻であり，システムの動作中に随時変化する．

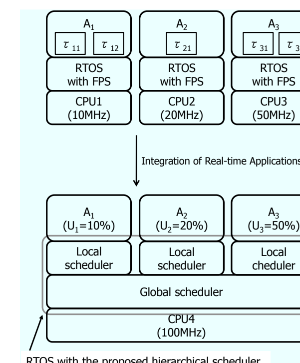

図 1 階層型スケジューリングによるリアルタイムアプリケーションの統合
Fig. 1 Integration of real-time applications using hierarchical scheduling.

A_i は，タスクの集合 $\tau_i = \{\tau_{i1}, \tau_{i2}, ...\}$ で構成される．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，ともにタスクの設計時に決定されるものとする．タスク τ_{ij} と τ_{ik} の優先度が $P_{ij} > P_{ik}$ であるとき，τ_{ij} は，τ_{ik} より高い優先度をもつものとする．C_{ij} は，性能 1 をも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，定数とする．なお，タスクの処理時間は，プロセッサ性能に比例するよう単純化する．したがって，性能 U_i をもつ個別プロセッサでの最悪実行時間は，$\frac{C_{ij}}{U_i}$ になるものとする．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を加算することで計算する．

その他に，本論文では以下を前提とする．

- 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．
- タスクは待ち状態にならない．
- タスクの起動要求は周期的に発生するとは限らない．
- タスク間の通信は非同期とする．タスクが待ち状態になるような同期通信はない．
- アプリケーションのタスク構成は設計時に決定され，システム動作中は変わらない．
- タスクの優先度は，デッドライン・モノトニックにより割当てられる．
- 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 をもつ個別プロセッサで FPS によりスケジュール可能である．
- 統合するアプリケーションに設定したプロセッサ利用率の合計は，１以下である．

© 2010 Information Processing Society of Japan

3. 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る舞いに関して，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）が割当てられ，アプリケーションデッドラインまでに使い切ることができる．
- 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 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 の二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 は共に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関わらず，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）をもつ高優先度タスク 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．

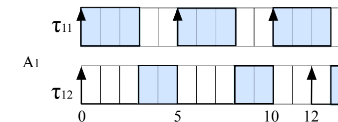

図 2 FPS によるアプリケーションのスケジューリング
Fig. 2 A scheduling of A_1 by FPS.

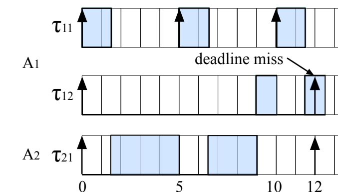

図 3 BSS のローカルスケジューラに FPS を使用した場合のスケジューリング
Fig. 3 a scheduling of A_1 and A_2 by BSS with FPS local schedulers.

4. 提案アルゴリズム

4.1 満たされる性質

本章では，ローカルスケジューリングにおいて，アプリケーションデッドラインより遅いデッドラインをもつ高優先度タスクの起動時刻を遅延させることで，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を提案する．高優先度タスクの起動を遅延させ，実行中の低優先度タスクを，処理が完了するまで優先的に実行する．この意味で，提案アルゴリズムのローカルスケジューリングアルゴリズムは，厳密な優先度ベーススケジューリングではない．しかしながら，優先度ベーススケジューリングの重要な性質の一つである，高い優先度を持つタスクが，その実行中に低い優先度をも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．したがって，非同期なタスク既存関係がある場合において，低い優先度をもつタスクには実行を妨げられないことを前提に，高い優先度をもつタスクでは明示的な排他制御をしていない場合でも，統合後に排他制御が崩れてしまうこ

© 2010 Information Processing Society of Japan

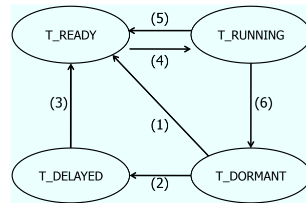

図 4 タスクの状態遷移図
Fig. 4 State transition diagram of a task.

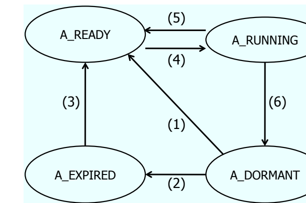

図 5 アプリケーションの状態遷移図
Fig. 5 State transition diagram of an application.

とはない．

4.2 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．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，4.5 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 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もつタスクが 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)）．

4.3 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一つでも A_READY 状態になり，かつ，バジェットが 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)）．

4.4 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．

© 2010 Information Processing Society of Japan

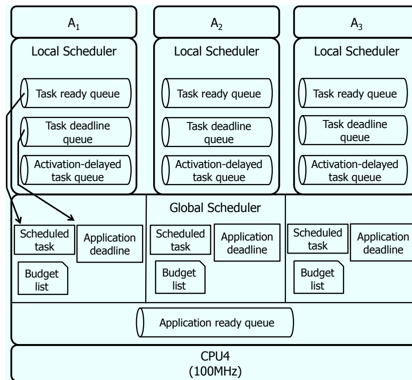

図 6 提案する階層型スケジューラの構成
Fig. 6 Construction of the proposed hierarchical scheduler.

- 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も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，4.6 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，アプリケーション毎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，T_READY 状態または T_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もっとも早い時刻（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，4.6 節で述べる．

4.5 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時の流れを説明する．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，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} が一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

© 2010 Information Processing Society of Japan

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時の流れを説明する．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．

4.6 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．バジェットリストに対する 3 つの操作として，バジェット要素の追加，更新，削除を定義する．バジェットの更新と削除については，BSS[1] と同じであるため，ここではバジェット要素の追加のみ説明する．

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も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．

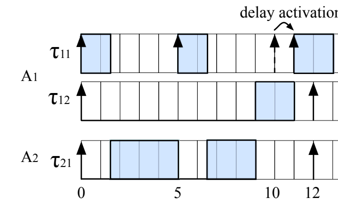

図 7 提案アルゴリズムによるスケジューリング
Fig. 7 A scheduling of A_1 and A_2 by the proposed scheduling algorithm

（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．

4.7 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，デッドライン時刻 15 までに実行を完了できる．

5. 評　　価

同一のアプリケーションをスケジューリングし，そのスケジュール可能性を比較するため，ローカルスケジューリングに FPS を採用した BSS と，提案アルゴリズムを実装したタスクスケジューリング・シミュレータを開発した．

シミュレーションによる評価の流れは，次の通りである．まず，スケジュール可能かどう

© 2010 Information Processing Society of Japan

Vol.2010-EMB-19 No.3
2010/12/6

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	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

かを判定する評価対象のアプリケーションとして，FPS で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，起動周期や最悪実行時間などのパラメータは，一様分布に従う乱数で生成した．また，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を統合し，それぞれに同じプロセッサ利用率を割当て，ローカルスケジューリングに FPS を採用した BSS と，提案アルゴリズムの２つのアルゴリズムでスケジューリングした．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り，タスク１つで構成される．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とで，テストベンチアプリケーションの絶対デッドラインもさまざまに変化する．その結果，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した．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

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周期の最大値は 50 単位時間で，最悪実行時間の最大値は 10 単位時間である．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つを統合し，それぞれ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 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．

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を生成する際の最大値は，評価１と同様である．非周期タスクの起動間隔は，シミュレーションの実行時に，次の式で計算している．

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

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，ここでは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る．λ は４に固定した．$rand()$ は，[0,1] の実数をランダムに生成する関数である．シミュレーションの結果，BSS 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．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と同じであるにも関わらず，BSS 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，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，シミュレーション中のプロセッサ利用率が低いため，BSS 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．

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評価するため，評価２のタスク生成時のパラメータ設定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周期の最大値は 50 単位時間で変更していないが，最小値を 20 単位時間に設定した．また，最悪実行時間の最大値を 10 単位時間から４単位時間に変更した．BSS 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 のスケジュール可能率が評価２に比べて低くなった理由として，評価対象アプリケーションを構成するタスク数が評価２に比べて増加したことで，３章で述べた，BSS

© 2010 Information Processing Society of Japan

でデッドラインをミスする状況が発生しやすくなったためと考えられる．評価 3 においても，提案アルゴリズムでは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．また，シミュレーション時間を 10,000 から 100,000 単位時間に増やしてシミュレーションした．動作するアプリケーションが 4 つに増加したため，プロセッサ利用率をそれぞれ 25%を割当てた．アプリケーションを構成するタスクの周期の最大値は 50 単位時間，最悪実行時間の最大値は 10 単位時間で変更していない．BSS 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．

以上より，すべての評価において，提案アルゴリズムを適用することで，評価対象アプリケーションをスケジュールできることを確認した．

6. 関連研究

階層型スケジューリングの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[5) や，これを拡張した時間保護アルゴリズム[6) 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予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．

Shin や Lipari らが提唱するリソースパーティショニングに基づきアプリケーションの起動周期を設計する手法[7)8)] は，各アプリケーションを周期的に実行させるため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[8)]．したがって，プロセッサ資源が限られる組込みリアルタイムシステムには向かない．

提案アルゴリズムの従来手法に対する優位性は，主に 2 つある．一つ目は，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，適用できるアプリケーションの対象が広いことである．二つ目は，正味のプロセッサ利用率（各タスクが必要とする最低限のプロセッサ利用率を合計した値）を越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，プロセッサの資源を有効に利用できることである．したがって，プロセッサ資源が限られる組込みリアルタイムシステムにも適用できる．

7. おわりに

本論文では，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した．スケジューリングシミュレータを用いて，提案アルゴリズムと従来アルゴリズムの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．今後の課題としては，提案アルゴリズムをリアルタイム OS に実装し，オーバヘッドを評価する予定である．

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) 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).
3) Lipari, G.: Resource Reservation in Real-Time Systems, *Ph.D Thesis*, Scuola Superiore S.Anna, Pisa, Italy, (2000).
4) Liu, C.L. and Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *Journal of the ACM (JACM)*, v.20 n.1, pp.46–61 (1973).
5) 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).
6) 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).
7) Lipari, G. and Bini, E.: A methodology for designing hierarchical scheduling systems, *Journal fo Embedded Computing*, Cenbridge International Science Publishing (2003).
8) Shin, I., and Lee, I.: Compositional Real-time Scheduling Framework, *Proc. IEEE Real-time Systems Symposium*, pp.57–67, (2004).

© 2010 Information Processing Society of Japan