RTOSシミュレーションのための性能と移植性が高いスケジューラ

中村宏明† 佐藤直人† 田渕直†

システムレベル設計を対象としてソフトウェアタスクの遅延時間要求付き機能モデルのシミュレーションを行うためのスケジューリングアルゴリズムを提案する．アルゴリズムの主要な特長は高性能で移植性が高いことである．コンテクストスイッチの回数を大幅に減らすことによってシミュレーション時間を短縮する．またベースとなるシミュレーションプラットフォームが少数のプリミティブを提供していれば実装できるので移植性が高い．アルゴリズムをまず効率を示すための単純な形で示す．次に既存のシミュレーションプラットフォームに容易に統合できる移植性の高い形で示す．続いて2つの版の正当性と同値性の形式的な証明を行う．最後にシステムレベル設計言語での実現方法と評価結果を示す．

Efficient and Portable Scheduler for RTOS Simulation

Hiroaki Nakamura,† Naoto Sato† and Naoshi Tabuchi†

We propose a new task scheduling algorithm for timed-functional simulation of concurrent software tasks. The key features of our algorithm are its efficiency and portability: It attains efficiency by reducing the frequency of context-switching between concurrent tasks by exploiting a newly-developed 'switching-point determination' technique. It also provides a high-degree of portability in the sense that it only needs the underlying system to support a very small number of primitives. Our algorithm is presented in several different forms: First, we present it in a simple form, highlighting the efficiency feature. Then, we provide a 'portable' version for facilitating integration to existing simulation kernels. Succeedingly, we provide formal proofs of the correctness of the two versions and their equivalence. This way of algorithm certification makes a clear distinction between our approach and other similar efforts. Finally, we provide concrete implementations built on top of system-level design languages and show some results of evaluation.

1. はじめに

複雑な組み込みシステムの設計では，設計初期段階で性能解析を行っておくことが重要になる．機能の側面からモデリングを行い，その機能モデルの各所に**遅延時間要求**を付加することによって，システムの性能に関する要件（スループットやレスポンス時間）を解析することが可能になる．たとえばSystemC[10]のリファレンス実装に添付されているサンプルシミュレーションプログラム `simple_perf` のコード片（**図1**）には機能を実現する前半6行に対して最後に実行時間要求 `wait` が付加されていて，このコード片の機能の実行に1,000 ns要することを表現している．

遅延時間要求を付加したモデルは各**タスク**が個別の処理装置に割り当てられる場合，つまりハードウェアで実現される場合には期待どおりに動作する．しかし1個のCPUで動作するマルチタスクソフトウェアの場合には対象のタスクが他のタスクとCPUを共有していて，RTOSによってスケジュールされた結果として合計で指定されたCPU時間を消費する，というように解釈実行する必要がある．このため本稿ではwait(*delay*)と区別して *delay* 時間のCPU時間を指定するための新たな遅延時間要求 consume(*delay*) を導入する[4),☆]．たとえば，ラウンドロビンスケジューリングでは時刻0の consume(5) と時刻2の consume(2) が**図2**に示すように交互に実行された結果として，それぞれ7秒後と3秒後に終了することが期待される．

処理時間を遅延時間要求として表現するシミュレーションモデルをSystemCなどのプラットフォームで実現するには，従来はスケジューリングの影響を正

† 日本アイ・ビー・エム株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.

☆ 遅延時間を要求するタスクを明示する必要がある場合は2引数の consume(*delay*, *taskid*) を用いる．

```
int i = 1 + int(19.0 * rand() / RAND_MAX);
while (--i >= 0) {
 out->write(*p++);
 if (!*p) p = str;
 -- total;
}
wait(1000, SC_NS);
```

図 1 遅延時間要求の付いたコード片
Fig. 1 Code fragment with delay time demand.

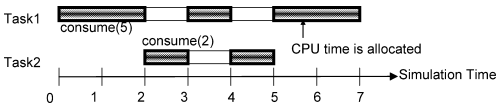


図 2 スケジュールされたソフトウェアタスクの実行
Fig. 2 Scheduled execution of software tasks.

確に遅延に反映させるためにソフトウェアタスクとタスクスケジューラモデルを個別のスレッドを持つモジュールとしていた[1]．スケジューラモデルが主導的に動作してタスクへの CPU 割当てを計算して，それに基づいてタスクに対して割込みをかけてタスクの状態（CPU 割当て中と割当て待ち中）の更新とタスクの状態に従って決定する残り時間の更新を行う．残り時間の更新後，ソフトウェアタスクは wait を再実行する．タスクの状態を変更するたびにコンテクストスイッチを起こす必要があるのでシミュレーション時間全体に対するコンテクストスイッチの時間の割合が大きく，シミュレーション速度が大きく低下する．

このような問題を解決するための RTOS 上で動作するソフトウェアのシミュレーションを行うためのタスクスケジューリングアルゴリズムを開発した．コンテクストスイッチの回数を減らすために，次回のコンテクストスイッチはあるタスクが処理を終了する時刻まで延期される．またこのアルゴリズムはベースとなるシミュレーションプラットフォームから少数のプリミティブが提供されていれば動作するので移植性が高い．

なお本研究ではタスクが RTOS でスケジュールされて動作した効果を遅延に反映させる仕組みだけを対象としている．ソフトウェアタスクはシミュレーションプラットフォームが提供するモジュール（動作単位）で実現することを仮定する．タスクの生成・終了やタスク間の同期はシミュレーションプラットフォームが提供する機能をそのまま使用する．

我々の技術的貢献は次の 3 点である：まず，我々が基にするスケジューリングアルゴリズムの概略を示し，それを SystemC などに移植しやすいアルゴリズムへ変換したものを示す．次に，スケジューリングアルゴリズムの形式的モデルを示し，2 つのアルゴリズムが同値であることを示す．さらに，代表的なシステムレベル設計言語である SystemC[10] と SpecC[9] での実装を示す．

本稿は以下のように構成されている：次章で関連研究を通して課題を明確にして，3 章で我々のアルゴリズムを 2 つの形で示す．4 章では 2 つの版の正当性と同値性の形式的な証明を行い，5 章でシステムレベル設計言語上での実現方法と評価結果を示す．6 章は結論である．

2. 関 連 研 究

Yoo ら[14] は，組み込みソフトウェアの時間を考慮したシミュレーションモデルを次の 3 種類に分類している：(1) 遅延付きソフトウェア機能モデル（functional simulation with delay annotation），(2) OS シミュレーションモデル（OS simulation model），(3) 命令セットシミュレーション（instruction set simulation）．遅延付き機能モデル (1) はシミュレーション環境の機能を使ってソフトウェアタスクのスケジューリングを行い，アプリケーションの実行時間を遅延として表現する．OS シミュレーションモデル (2) はターゲット OS と同等の機能を実現するシミュレータをホストワークステーションで動作させるものである．命令セットシミュレーション (3) は，ターゲット CPU の動作をホストワークステーションでシミュレートする．本稿では設計初期段階（実装レベルの設計やターゲットコードがない段階）でタスクの優先度や OS のスケジューリング方式などが与える影響を評価するための (1) 遅延付きソフトウェア機能モデルを対象とする．

遅延付きソフトウェア機能モデルは 3 分類の中で最も荒いシミュレーションモデルである．タスクスイッチやスケジューリングなどの OS のオーバヘッドが遅延時間として抽象化されてモデルに取り込まれるので，これらのオーバヘッドの見積り値の正確さがシミュレーションの精度に影響を与える．特にキャッシュの影響が反映されないことが精度を低下させる原因になる可能性がある．Yi ら[12] は動画像処理システムを例として使用して，タスクスケジューラを命令セットシミュレータで実行した場合と遅延付きソフトウェア機能モデルとして実現した場合の精度を比較した．スケジューリングモデルがキャッシュの影響を反映していないことによる誤差は予測値で 2.8%，実測値で最大 7.3%であったと報告している．

遅延時間要求 consume の実現方法の 1 つは，遅延要求を直列化することである．これにより，ある瞬間に動作しているタスクを最大 1 つにすることができる[3),6)]．この方法の欠点は図 2 のようなプリエンプションのあるスケジューリングを正確に扱えないこと

である．遅延時間が長くなるときにあらかじめ時間を分割しておくことによってこの問題は部分的に解決できる[7]が，次のイベントが起きる時刻を予測する必要があるので，この手法は詳細なシステム設計がなされていない設計初期段階では利用できない．

プリエンプションを正確に扱えるようにするには，スケジューラモデルが主導的に動作してタスクへのCPU割当てを計算して，それに基づいてタスクに対して割込みをかけてタスクの残り時間を更新し，タスクは更新された残り時間に従って再び遅延時間を発生させるようにする[1),5)]．この手法はプリエンプションがあるスケジューリングを正確に扱うことができる一方で，スケジューラモデルとソフトウェアタスク間のコンテクスト切替えが非常に多くなる．ソフトウェアタスクのスレッドにスケジューラの操作を実行させることによってソフトウェアタスクからスケジューラへのコンテクストスイッチを取り除くことができる[8)]が，ソフトウェアタスク間のコンテクストスイッチは取り除けない．

割込み発生時のソフトウェアタスクへのコンテクストスイッチ数を減少させる方法として，各タスクにローカルなクロックを管理させて，遅延終了後にCPUが割り当てられなかった時間でローカルクロックを調整する手法が提案されている[2)]．この手法ではイベントの順序を保つために割込み発生時にすべてのクロックの同期をとる必要があり，このときに割込み発生時刻より遅いクロックを持つタスクへのコンテクストスイッチが発生する．そのためモジュール数が多くなるとクロック同期の際にコンテクストスイッチが発生する割合が高くなると考えられる．文献2)のハードウェアとソフトウェアからなる衛星通信機を例とした評価では，シミュレーションのコンテクストスイッチ数は本来のコンテクストスイッチの89%と報告されている．またこの手法では個々のタスクが固有の時間を持つ必要があるため，各モジュールがグローバル時間を共有するようなシミュレーションプラットフォーム（SystemCやSpecC）で実現できない．

3. タスクスケジューリングアルゴリズム

本章ではまず我々が基にするスケジューリングアルゴリズムの概略を示し，次にこのアルゴリズムを既存のシミュレーションプラットフォームに統合できる移植性の高い形に変換したものを示す．

3.1 スケジューリングアルゴリズム（バージョン1）

図3に1つのCPUで動作する複数のソフトウェアタスクを管理するアルゴリズムを示す．

consume(dt, taskid) ソフトウェアタスクからのエントリポイントである．あるタスクがconsume(dt, taskid)を呼び出すと，CPUが合計でdt単位時間をこのタスクのために消費した後，呼び出し元に戻る．この手続きはタスクに割り当てる時間tasks[taskid].remaining_timeと，スケジューラがタスクの残り時間が0かどうかを調べるチェックポイントtasks[taskid].checkpointの初期値をセットする．最後にこの手続きは同期機構SUSPEND_AND_RESUME(scheduler)を呼び出してコンテクストをタスクからスケジューラに切り替えて，スケジューラの実行を再開する．その後，スケジューラがSUSPEND_AND_RESUME(task)を呼び出すと，ソフトウェアタスクtaskの実行が再開されて，consumeの実行は終了する．

update(t1, t2) 個々のスケジューリングポリシを実装していて，t1からt2までの時間を各タスクに割り当てる．その結果として各タスクの残り時間を減少

```
struct TASK {int remaining_time, checkpoint; bool active;};
const int ntask;
TASK tasks[ntask];
int curr_time, last_time;

void consume(int dt, int taskid) {
 tasks[taskid].remaining_time = dt;
 tasks[taskid].checkpoint = curr_time + dt;
 tasks[taskid].active = false;
 SUSPEND_AND_RESUME(scheduler);
}

void update(int t1, int t2 );
// requires 0 ≤ t1 ≤ t2
//  and ∀ i. 0 ≤ tasks[i].remaining_time
//  and ∀ i. t2 ≤ tasks[i].checkpoint ≤ t1 + tasks[i].remaining_time
// ensures ∀ i. tasks[i].remaining_time@pre ≥ tasks[i].remaining_time
//  and ∑_i (tasks[i].remaining_time@pre - tasks[i].remaining_time) ≤ t2 - t1
// (@pre indicates the value at the start of the execution)

void scheduler_run() {
 update(last_time, curr_time);
 last_time = curr_time;
 for(i=0;i<ntask;i++)
  if( tasks[i].checkpoint == curr_time )
   if( tasks[i].remaining_time == 0 ) tasks[i].active = true;
   else tasks[i].checkpoint = curr_time + tasks[i].remaining_time;
 for(i=0;i<ntask;i++)
  if( tasks[i].active ) SUSPEND_AND_RESUME(tasks[i]);
}

void scheduler_main() {
 curr_time = 0; last_time = 0;
 while(1) {
  scheduler_run();
  curr_time = min_task_checkpoint(); // min{tasks[i].checkpoint | i}
 }
}
```

図3 スケジューリングアルゴリズム（バージョン1）
Fig. 3 Scheduling algorithm (version 1).

```
void update(int t1, int t2 ) {
  h = 0; // 最優先タスク id
  for(i=0;i<ntask;i++)
    h = tasks[i].priority>tasks[h].priority ? i : h;
  tasks[h].remaining_time = tasks[h].remaining_time - (t2-t1);
}
```

(a) Priority Scheduling

```
i = 0;
void update(int t1, int t2) {
  for (;t1<t2;t1++,i++)
    tasks[i%ntask].remaining_time--;
}
```

(b) Fair Scheduling

図 4 手続き update(t1, t2) の例
Fig. 4 Examples of procedure update(t1, t2).

させる．手続き update の例を**図 4** に示すが，図 3 に示す条件を満たしている限り手続きの本体はどのように定義してもよい．

scheduler_main(), scheduler_run() スケジューラは自分自身のスレッドを持ち，タスクが再開される時間を決めることによってソフトウェアタスクを制御している．手続き scheduler_main は scheduler_run の呼び出しと curr_time の値の更新を繰り返す．手続き scheduler_run の本体はまず update(last_time, curr_time) を呼び出し，次に SUSPEND_AND_RESUME を呼び出すことによって残り時間が 0 になったタスクの実行を再開する．他のタスクに対しては，チェックポイントが次に残り時間をチェックする時刻になるように更新する．

このアルゴリズムでは手続きは consume が curr_time を変化させずにただちにスケジューラにスイッチするので，プリエンプションを遅延なく実現する．さらに手続き scheduler_run は時刻 last_time と curr_time の間にはコンテクストスイッチを起こさないので効率が良い．

一方このアルゴリズムは SUSPEND_AND_RESUME という低レベルでプラットフォーム依存の同期機構によってスレッドの実行を制御しているので，そのままでは SystemC や SpecC などの既存のシミュレーションプラットフォームで動作させることができない．次にこの制限を取り除く．

3.2 スケジューリングアルゴリズム（バージョン 2）

コードの一部がシミュレーションプラットフォームが提供するプリミティブで置き換えられるような同等のアルゴリズムになるように，元のアルゴリズムを**図 5** のように変更する．

手続き scheduler_run のコードは scheduler_main と consume の中に移動し，scheduler_run 自体はなくなっている．また consume は呼び出したタスク

```
void consume(int dt, int i) {
  tasks[i].remaining_time = dt
  while( tasks[i].remaining_time > 0 ) {
    tasks[i].checkpoint = curr˙time + tasks[i].remaining˙time; // (*)
    SUSPEND_AND_RESUME(scheduler);                            // (*)
    update(last_time,curr_time);
    last_time = curr_time;
  }
}

void scheduler_main() {
  curr_time = 0; last_time = 0;
  while(1) {
    curr_time = min_task_checkpoint();
    for(i=0;i<ntask;i++)
      if (tasks[i].checkpoint == curr_time)
        SUSPEND_AND_RESUME(tasks[i]);
  }
}
```

図 5 スケジューリングアルゴリズム（バージョン 2）
Fig. 5 Scheduling algorithm (version 2).

```
void consume(int dt, int i) {
  tasks[i].remaining_time = dt
  while( tasks[i].remaining_time > 0 ) {
    wait(tasks[i].remaining_time, i);
    update(last_time,curr_time);
    last_time = curr_time;
  }
}
```

図 6 シミュレーションプラットフォームの API を使用するように変更したアルゴリズム
Fig. 6 Modified algorithm for using APIs of simulation platforms.

のチェックポイントを更新して，残り時間が 0 にならなかった場合にはいったんスケジューラに制御を移すようになった．その結果として，consume は SUSPEND_AND_RESUME(scheduler) を 1 回以上呼び出す場合がある．

手続き scheduler_main はシミュレーション時間の進行を管理し，タスクが時間順に実行されるように制御する．多くの既存のシミュレーションプラットフォームの最上位は scheduler_main のようなループ構造になっており，SystemC の Functional Specification [11] ではスケジューラの手続き scheduler::execute として説明されている．また consume 中の (*) の部分は呼び出したタスクに遅延時間を割り当てるためのコードで，SystemC の Functional Specification でスレッドのベースクラスの手続き sc_thread::wait として説明されているものと同等である．つまり手続き scheduler_main と consume 中の (*) の部分はシミュレーションプラットフォームの提供する手続きに変更して，低レベルの同期機構を取り除くことができる．シミュレーションプラットフォームが提供する API で置き換えたアルゴリズムを**図 6** に示す．

4. アルゴリズムの正当性

タスクスケジューリングアルゴリズムの形式的モデルを定め，3 章で導入したアルゴリズムが‘正しい’ことをこのモデルの上で証明する．またアルゴリズムの 2 つのバージョンがモデル上では‘同じ’であることを示す．

4.1 タスクスケジューリングの形式モデル

各タスクスケジューリングの過程を**スケジューリング状態の遷移**の列としてモデル化する：まず，ある時点におけるタスク θ を，タスクの残り時間 ρ_θ と次のチェックポイント ν_θ の組 $(\rho_\theta, \nu_\theta)$ で表す．タスクの全体を Θ とする．また，スケジューリングの過程で 2 つの時刻 T_{last} と T_{curr} を扱う．T_{last} は Θ が最後にスケジュールされた時刻を，T_{curr} は（スケジューリングしようとしている）現在時刻を各々表している．現在時刻 T_{curr} において時間区間 $[T_{\mathrm{last}}, T_{\mathrm{curr}}]$ を分割してタスクに割り当てることが可能である．時刻 T_{last} と T_{curr} と集合 Θ によってスケジューリング状態 $s \triangleq (T_{\mathrm{last}}, T_{\mathrm{curr}}, \Theta)$ が定義され，スケジューリングの各ステップは 2 つの状態間の遷移として定義される．この定義を用いて正当なスケジューリング過程は次のように定義される．

正当な状態（valid state）：次の関係が成り立つとき，スケジューリング状態 $s = (T_{\mathrm{last}}, T_{\mathrm{curr}}, \Theta)$ は**正当**（*valid*）である：

$$0 \le T_{\mathrm{last}} \le T_{\mathrm{curr}} \tag{1}$$

$$0 \le \rho \quad \wedge \quad T_{\mathrm{curr}} \le \nu \le T_{\mathrm{last}} + \rho \quad (\text{for } \forall(\rho,\nu) \in \Theta) \tag{2}$$

正当な状態遷移：次の関係が成り立つとき，正当な状態 s_1 と s_2 の間の遷移 $s_1 \to s_2$ は**正当**である☆：

$$T^{s_1}_{\mathrm{last}} \le T^{s_2}_{\mathrm{last}} \ \wedge\ T^{s_1}_{\mathrm{curr}} \le T^{s_2}_{\mathrm{curr}} \tag{3}$$

$$\rho^{s_1} \ge \rho^{s_2} \ \wedge\ \nu^{s_1} \le \nu^{s_2} \quad (\text{for } \forall(\rho,\nu) \in \Theta) \tag{4}$$

$$\sum_{(\rho,\nu)\in\Theta} (\rho^{s_1} - \rho^{s_2}) \le T^{s_2}_{\mathrm{last}} - T^{s_1}_{\mathrm{last}} \tag{5}$$

ここで，式 (4) は遷移による各タスク $\theta = (\rho, \nu)$ の残り時間の減少とチェックポイントの延長（の可能性）を示しており，式 (5) は経過時間 $(T^{s_2}_{\mathrm{last}} - T^{s_1}_{\mathrm{last}})$ のタスクへの分配を示している．明らかに，上記式 (1)–(5) はすべてのスケジューリングが満たすべき必要条件である．一方，式 (1)–(5) を満たす任意の遷移列に対応するスケジューリングが実際に存在しうることは容易に納得できる．

☆ 上付き添字は状態を示す．たとえば $T^{s_1}_{\mathrm{curr}}$ は状態 s_1 での T_{curr} の値を表す．

4.2 アルゴリズムの正当性

スケジューリングアルゴリズムの正当性は，それによって生成される任意のスケジューリング過程が正当な状態遷移列になることに対応する．このことを次のステップで示す：まず，正当な状態 s_1 を別の正当な状態 s_2 に移す正当な状態遷移関数 E（*E*lapse），X_θ（ne*X*t），U^t（*U*pdate）を導入する．そして，スケジューリング過程中の任意の遷移が E，X_θ，U^t の組合せで定義されることを示す．まず状態遷移関数を定義する：

- (E) $\mathsf{E}(s_1)$ は $T^{s_1}_{\mathrm{curr}}$ を更新し，タスクのチェックポイントのうちの最小値に設定する．他の値は変化させない：$T^{s_2}_{\mathrm{curr}} = \min\{\nu \mid (\rho,\nu) \in \Theta\}$.
- (X_θ) $\mathsf{X}_\theta(s_1)$ はタスク $\theta = (\rho,\nu)$ のチェックポイントを $T_{\mathrm{last}} + \rho$ に延長する：$\nu^{s_2} = T_{\mathrm{last}} + \rho$.
- (U^t) $\mathsf{U}^t(s_1)$ は $T^{s_1}_{\mathrm{last}}$ から t（$T^{s_1}_{\mathrm{last}} \le t \le T_{\mathrm{curr}}$）までの時間区間 $[T^{s_1}_{\mathrm{last}}, t]$ をタスクに分配し，個々のタスク (ρ,ν) の残り時間 ρ をその分減少させる．チェックポイント ν は変化させない．また，T_{last} を t に更新する（$T^{s_2}_{\mathrm{last}} = t$）.

これらについて，s_1 が正当な状態であるとき（式 (1), (2) を満たすとき），$\mathsf{E}(s_1)$，$\mathsf{X}_\theta(s_1)$（$\theta \in \Theta$），および $\mathsf{U}^t(s_1)$ がやはり式 (1)，(2) を満たすこと，また遷移の前後で式 (3)–(5) を満たすことは容易に確かめられる．たとえば状態 $s_2 = \mathsf{U}^t(s_1)$ で式 (2) が成り立つことは次のように確かめられる：

$$\begin{aligned} T^{s_2}_{\mathrm{last}} + \rho^{s_2} &= T^{s_1}_{\mathrm{last}} + (T^{s_2}_{\mathrm{last}} - T^{s_1}_{\mathrm{last}}) + \rho^{s_1} \\ &\quad - (\rho^{s_1} - \rho^{s_2}) \\ &\ge T^{s_1}_{\mathrm{last}} + (T^{s_2}_{\mathrm{last}} - T^{s_1}_{\mathrm{last}}) + \rho^{s_1} \\ &\quad - (T^{s_2}_{\mathrm{last}} - T^{s_1}_{\mathrm{last}}) \\ &\ge \nu^{s_1} = \nu^{s_2} \end{aligned}$$

すなわち，これらの遷移関数は確かに正当な状態遷移を生成する．

さて，U^t のスケジューリング（時間の分配）の詳細は，実際には個々のポリシに依存して決まる．それらに対応した U^t を個別に定める代わりに，以下の条件を満たすもの全体を考えることにする：

$$\mathsf{U}^t \circ \mathsf{E}(s) = \mathsf{E} \circ \mathsf{U}^t(s) \tag{6}$$

$$\mathsf{U}^t \circ \mathsf{X}_\theta(s) = \mathsf{X}_\theta \circ \mathsf{U}^t(s) \tag{7}$$

$$\mathsf{U}^{t_2} \circ \mathsf{U}^{t_1}(s) = \mathsf{U}^{t_2}(s) \qquad (t_1 \le t_2) \tag{8}$$

直観的には，これらは U^t がタスクの ρ の値だけに依存することを示している．実際，様々なポリシは上の条件を満たすように定義でき，一方，以下の議論は条件を満たす任意の U^t について成り立つ．

遷移関数 E，X_θ，U^t を使ってスケジューリングアルゴリズム P のモデル $\mathcal{M}$ を定める．$\mathcal{M}[\![P]\!]$ は無限

状態遷移系であり，無限個のスケジューリング状態からなる．状態間の遷移は E，X_θ，U^t の合成として定められる．以下に $\mathcal{M}$ の概略を示す：P は，タスク集合 tasks，時間 last_time，curr_time の 3 つの変数と，それらを操作する手続き群からなる．$\mathcal{M}$ によって，3 つの変数は各々 Θ，T_{last}，T_{curr} に対応付けられる．task.active 変数やその操作については，ここでは実行時のタスク生成や消滅を考慮しないので無視する．各手続きについては，その中に現れる代入操作に着目し，コントロールフローは考えない（スケジューラの正しさには無関係であることを定理 1 で示す）．バージョン 1 のアルゴリズムの場合，consume(dt,taskid) はタスクの初期化 $(\rho_\theta, \nu_\theta) = (\mathsf{dt}, T_{\mathrm{curr}} + \mathsf{dt})$ に対応付けられる．ここで θ は taskid で指定されるタスクである．update(last_time,curr_time) は $\mathsf{U}^{\mathsf{curr\_time}}$ に対応付けられる．scheduler_run は $\mathsf{U}^{\mathsf{curr\_time}}$ と X_θ（$\theta \in \Theta$）の組合せに，scheduler_main の中の代入は E に各々対応付けられる．バージョン 2 についても，同様に E，X_θ，U^t を使ってモデルを定義できる．

以上より，アルゴリズム P の意味がモデル上で定められた．$\mathcal{M}[\![P]\!]$ は初期スケジューリング状態から始まる無限状態遷移系であり，すべての遷移は E，X_θ，U^t だけからなる．これらの遷移関数は正当な状態を正当な状態にうつすので，結局初期状態が正当であれば，生成されるスケジューリング過程全体は正当であり，これによりアルゴリズムの正当性が示された．

4.3 2 つのバージョンの同値性

3 章で示した 2 つのバージョンのアルゴリズムが意味的に等価であることを示す．すなわち，バージョン 2 のアルゴリズム P_2 が元のバージョン 1 のアルゴリズム P_1 とモデル上で同値である（$\mathcal{M}[\![P_1]\!] \sim \mathcal{M}[\![P_2]\!]$）ことを示す．まず状態の同値性を定義する．

定義 1. 2 つの状態 $s_1 = (T^{s_1}_{\mathrm{last}}, T^{s_1}_{\mathrm{curr}}, \Theta^{s_1})$ と $s_2 = (T^{s_2}_{\mathrm{last}}, T^{s_2}_{\mathrm{curr}}, \Theta^{s_2})$ に対して，$T^{s_1}_{\mathrm{last}} = T^{s_2}_{\mathrm{last}}$ が成り立ち，すべての $(\rho, \nu) \in \Theta$ に対して $\rho^{s_1} = \rho^{s_2}$ が成り立つとき，$s_1 \sim s_2$ と書くことにする．明らかに $\sim$ は同値関係で $s_1 = s_2$ は $s_1 \sim s_2$ を含意する．

$s_1 \sim s_2$ は，Θ がどのように T_{last} までスケジュールされたかとは関係なく，スケジューリングの観点からは時刻 T_{last} のとき s_1 と s_2 が同一視できることを示している．ここで $T^{s_1}_{\mathrm{curr}} = T^{s_2}_{\mathrm{curr}}$ は成り立つ必要がない．時刻 T_{last} 以降のスケジューリングはこれからなされるので，T_{curr} が 2 つの状態の同値性に無関係であるとするのは自然である．この関係を用いて P_1 と P_2 の同値性を保証する主要な定理を述べる準備ができた．

定理 1. $s_{(1,0)} \to s_{(1,1)} \to \cdots \to s_{(1,n)}$ と $s_{(2,0)} \to s_{(2,1)} \to \cdots \to s_{(2,m)}$ を E，X_θ，U^t の適用によって得られる 2 つの遷移列とする．正当な $s_{(1,0)}$ に対して次が成り立つ：

$$s_{(1,0)} = s_{(2,0)} \ \wedge \ T^{s_{(1,n)}}_{\mathrm{last}} = T^{s_{(2,m)}}_{\mathrm{last}} \ \Rightarrow \ s_{(1,n)} \sim s_{(2,m)}$$

直観的には上の定理は E，X，U に関する限りスケジューリングは初期状態だけに依存し（途中の状態には無関係で），任意の時点 T_{last} において両者の対応する状態が同値になることを主張している．すなわち，初期状態が同じ 2 つのスケジューリング過程が与えられると，それらは任意の時点で同値になる．したがって $\mathcal{M}[\![P_1]\!]$ と $\mathcal{M}[\![P_2]\!]$ の対応する状態がつねに同値になるという意味で 2 つのバージョン P_1，P_2 は等価である（$\mathcal{M}[\![P_1]\!] \sim \mathcal{M}[\![P_2]\!]$）．この節の残りでは以下の 2 つの補題を使って定理の証明を行う．

補題 1. $\mathsf{E}(s) \sim s, \quad \mathsf{X}_\theta(s) \sim s.$

定義 2. 状態 s に対して，関係 $\mathcal{R}_s$ を以下のように帰納的に定義する．

A. $(s, s) \in \mathcal{R}_s$

B. $(s_1, s_2) \in \mathcal{R}_s,\ \theta \in \Theta$

$$\Rightarrow \begin{cases} (\mathsf{E}(s_1), s_2) \in \mathcal{R}_s, & (s_1, \mathsf{E}(s_2)) \in \mathcal{R}_s \\ (\mathsf{X}_\theta(s_1), s_2) \in \mathcal{R}_s, & (s_1, \mathsf{X}_\theta(s_2)) \in \mathcal{R}_s \end{cases}$$

C. $(s_1, s_2) \in \mathcal{R}_s,\ t = T^{s_1}_{\mathrm{last}} \le \min(T^{s_1}_{\mathrm{curr}}, T^{s_2}_{\mathrm{curr}}) - 1$

$$\Rightarrow \left(\mathsf{U}^{t+1}(s_1), \mathsf{U}^{t+1}(s_2)\right) \in \mathcal{R}_s.$$

ここで ***C*** は，$\mathcal{R}_s$ に属する 2 つの状態 s_1，s_2 に対して単位時間分のスケジューリングを行い $T^{s_1}_{\mathrm{last}}$（$= T^{s_2}_{\mathrm{last}}$）を進めたものがやはり $\mathcal{R}_s$ に属することを定めている．

補題 2. 正当な状態 s に対して $(s_1, s_2) \in \mathcal{R}_s$ は $s_1 \sim s_2$ を含意する．

証明

$\mathcal{R}_s$ についての整礎帰納法によって示す．[**A**] 反射律により $s \sim s$ が成り立つ．[**B**] 補題 1 によって $\mathsf{E}(s_1) \sim s_1$ が成り立つ．したがって推移律により帰納法の仮定 $s_1 \sim s_2$ から $\mathsf{E}(s_1) \sim s_2$ が成り立つ．他の場合も同様．[**C**] $s_1 \sim s_2$ と仮定する．$\phi_1(s_1) = \phi_2(s_2)$ を満たす ϕ_1 と ϕ_2 を導入する．E と X_θ の組合せで容易に ϕ_1 と ϕ_2 が定義できる．式 (6) と (7) により $\mathsf{U}^{t+1} \circ \phi_1(s_1) = \phi_1 \circ \mathsf{U}^{t+1}(s_1)$ と $\mathsf{U}^{t+1} \circ \phi_2(s_2) = \phi_2 \circ \mathsf{U}^{t+1}(s_2)$ が成り立つので，補題 1 をつかって $\mathsf{U}^{t+1}(s_1) \sim \phi_1 \circ \mathsf{U}^{t+1}(s_1) = \mathsf{U}^{t+1}(\phi_1(s_1))$，$\mathsf{U}^{t+1}(s_2) \sim \phi_2 \circ \mathsf{U}^{t+1}(s_2) = \mathsf{U}^{t+1}(\phi_2(s_2))$ が導かれる．$\phi_1(s_1) = \phi_2(s_2)$ なので，$\mathsf{U}^{t+1}(s_1) \sim \mathsf{U}^{t+1}(s_2)$ が成り立つ．□

定理 1 の証明

$\vec{s_1}$ と $\vec{s_2}$ を 2 つの遷移列とする（$\vec{s_k} \triangleq s_{(k,0)} \to$

$s_{(k,1)} \to \cdots$）．まず $\vec{s_k}$（$k = 1, 2$）に出現する U^t を次のように置換する．$s_{(k,i)} \overset{\mathsf{U}^t}{\to} s_{(k,i+1)}$ を $s_{(k,i)}$ と $s_{(k,i+1)}$ の間の遷移と仮定すると，等式 (8) をもちいてこの遷移は以下のように置き換えることができる：

$$s_{(k,i)} \overset{\mathsf{U}^{t_0+1}}{\to} s_{(k,i_1)} \overset{\mathsf{U}^{t_0+2}}{\to} s_{(k,i_2)} \overset{\mathsf{U}^{t_0+3}}{\to} \cdots \overset{\mathsf{U}^{t_0+\Delta t}}{\to} s_{(k,i+1)}$$

$$(t_0 = T_{\mathrm{last}}^{s_{(k,i)}},\ \Delta t = t - t_0)$$

つまり t_0 と t の間で Θ をスケジュールする U^t を単位時間のスケジューリング列 $\mathsf{U}^{t_0+\Delta t} \circ \cdots \circ \mathsf{U}^{t_0+1}$ に分解する．置換後の $\vec{s_k}$ の初期状態および最終状態は，元の列のものと同一である．これら 2 つの列に対して，$(s_{(1,n)}, s_{(2,m)}) \in \mathcal{R}_{s_{(1,0)}}$ がただちに示される．したがって，補題 2 により $s_{(1,n)} \sim s_{(2,m)}$ が成り立つ．□

これまでスケジューリングの対象になるタスク全体 Θ が変化しないことで議論を単純化してきた．タスクの追加と削除を考慮した場合への定理の拡張を示す．

定理中では，タスクの追加と削除に対応する以下の状態遷移関数を使用する：

(D_θ) $\mathsf{D}_\theta(s_1)$ は，Θ^{s_1} から $\theta = (0, T_{\mathrm{curr}})$ を取り除く：$\Theta^{s_2} = \Theta^{s_1} \setminus \{\theta\}$

(A_θ) $\mathsf{A}_\theta(s_1)$ は，$T_{\mathrm{last}} = T_{\mathrm{curr}}$ となるとき Θ^{s_1} に $\theta = (\rho, T_{\mathrm{curr}} + \rho)$ を加える：$\Theta^{s_2} = \Theta^{s_1} \cup \{\theta\}$

定理 2. $s_{(1,0)} \to s_{(1,1)} \to \cdots \to s_{(1,n)}$ と $s_{(2,0)} \to s_{(2,1)} \to \cdots \to s_{(2,m)}$ を，E, X, U, D, A を適用することによって得られる遷移列とする．$s_{(1,0)} = s_{(2,0)}$ と仮定する．さらにタスクの追加と削除は 2 つの列で同時に起きると仮定する．すると $T_{\mathrm{last}}^{s_{(1,n)}} = T_{\mathrm{last}}^{s_{(2,m)}}$ が成り立つ時刻において，その時刻に他のタスクの追加・削除がない場合に $s_{(1,n)}$ と $s_{(2,m)}$ は同値になる．

5. 実装と評価

5.1 SystemC との統合

図 6 のアルゴリズムの SystemC による実装を**図 7** に示す．この実装では時間に関する操作を SystemC のプリミティブに置き換えた．時間データ型を int から sc_time に変更し，現在時刻の取得手段を変数 curr_time へのアクセスから関数 sc_time_stamp への呼び出しに変更した．またスケジューラを 1 つのクラスで実現して，タスクを割り当てる時間を Standard Template Library の map で管理することによって，SystemC のベースの C++ との親和性が高くなるようにした．

この実装（図 7）と元になるアルゴリズム（図 6）ではスケジューリングの対象になるタスク全体が変化しないと仮定していた．しかし実際のシステムでは実行中にタスクが生成・消滅し，また I/O 待ちのタスクには CPU 割当てが起きないため，スケジューリング対象のタスクが変化する．スケジューリング対象のタスクの追加・削除に対応するよう変更した実装を**図 8** に示す．クラス Scheduler の consume は，呼び出し時に enter でタスクを追加し，退出時に update 中でタスクを削除する．enter の直前には，追加するタスクのチェックポイントを現在時刻に基づいて設定するために update の呼び出しと curr_time の現在時刻への更新を行っておく．この追加されたコードは状態遷移関数 U に対応し，4 章で議論したように他のタスクに対するスケジューリングの正しさに影響を与えない．

```
class Scheduler {
protected:
  map<int, sc_time> remaining_time;
  sc_time last_time;
  Scheduler() : last_time(SC_ZERO_TIME) {}
  virtual void update(void) = 0;
public:
  void consume(sc_time dt, int task) {
    remaining_time[task] = dt;
    while (remaining_time[task] > SC_ZERO_TIME) {
      wait(remaining_time[task]);
      update();
      last_time = sc_time_stamp();
    }
  }
};
```

図 7 SystemC の API を使用するアルゴリズム
Fig. 7 Modified algorithm for using SystemC API.

クラス Scheduler のサブクラスは個々のスケジューリングポリシに従って update と enter を実装する．スケジューリングポリシは手続き update にカプセル化されている．割込み型優先度（Preemptive Priority）スケジューラの実装は 2 つのチェックポイントの間の時間全部を優先度が最高のタスクに割り当てる．enter が実行されるときにタスクの優先度がチェックされるので，consume を呼び出す前にタスクの優先度を示す属性を変更しておけば，それがスケジューリングに反映される．ラウンドロビンスケジューラの実装は TIME_SLICE の時間をスケジュールされているタスクに順番に割り当てる☆．本稿では 2 つのスケジューリングポリシを示しているが，他のポリシも同様に update の実装によってモデル化できる．

割込み型優先度（Preemptive Priority）スケジューリング

まず Task1（低優先度）が consume(5) を呼び出

☆ RoundRobinScheduler::update は 4 章の式 (8) を満たさないシンプルな場合を示す．

```
class Scheduler {
protected:
  map<int, sc_time> remaining_time;
  sc_time last_time;
  Scheduler() : last_time(sc_time_stamp()) { }
  virtual void update(void) = 0;
  virtual void enter(sc_time time, int task) = 0;
public:
  void consume(sc_time dt, int task) {
    update();
    last_time = sc_time_stamp();
    enter(task);
    remaining_time[task] = dt;
    while (remaining_time[task] > SC_ZERO_TIME) {
      wait(remaining_time[task]);
      update();
      last_time = sc_time_stamp();
    }
  }
};
```

```
class PreemptivePriority : public Scheduler {
  priority_queue<int,vector<int>,compare> tasks;
public:
  void update(void) {
    sc_time time_to_consume =
        sc_time_stamp() - last_time;
    if (tasks.size() > 0) {
      int task = tasks.top();
      remaining_time[task] =
          remaining_time[task] - time_to_consume;
      if (remaining_time[task] == SC_ZERO_TIME)
        tasks.pop();
    }
  }
  void enter(int task) {
    tasks.push(task);
  }
};
```

```
class RoundRobin : public Scheduler {
  queue<int> tasks;
public:
  void update(void) {
    sc_time time_to_consume =
        sc_time_stamp() - last_time;
    while (time_to_consume > SC_ZERO_TIME) {
      int task = tasks.front(); tasks.pop();
      sc_time step_time = min(TIME_SLICE,
          time_to_consume, remaining_time[task]);
      remaining_time[task] =
          remaining_time[task] - step_time;
      if (remaining_time[task] > SC_ZERO_TIME)
        tasks.push(task);
      time_to_consume -= step_time;
    }
  }
  void enter(int task) {
    tasks.push(task);
  }
};
```

図 8 タスクの追加・削除に対応した SystemC スケジューラ
Fig. 8 SystemC scheduler that handles adding and deleting tasks.

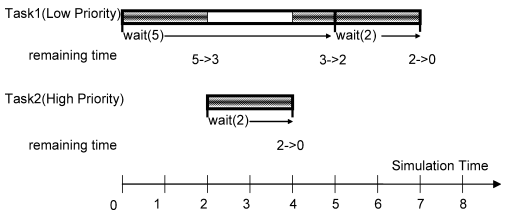


図 9 割込み優先度（Preemptive Priority）スケジューリング
Fig. 9 Preemptive priority scheduling.

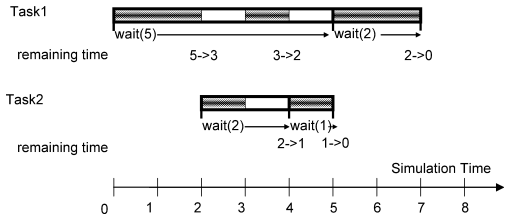


図 10 ラウンドロビンスケジューリング
Fig. 10 Round robin scheduling.

し，その 2 秒後に Task2（高優先度）が consume(2) を呼び出すと仮定する．2 つのタスクに呼び出された手続き consume は次のように処理される（**図 9**）．[**0 (Task1)**] Task1 の残り時間が 5 に初期化される．[**2 (Task2)**] 時刻 0 と 2 の間に Task1 は実行可能である．実行可能な時間には CPU 時間を割り当てることができる．CPU 時間の割当ての結果としてタスクの残り時間が減少する．Task1 の残り時間は 5 から 3 に更新される．Task2 の残り時間は 2 に初期化される．[**4 (Task2)**] Task2 は時刻 2 と 4 の間に実行可能で優先度が高いので，Task2 の残り時間は 2 から 0 に更新される．Task2 は終了する．[**5 (Task1)**] 時刻 4 と 5 の間に実行できるタスクは Task1 なので，Task1 の残り時間は 3 から 2 に更新される．[**7 (Task1)**] 時刻 5 と 7 の間に実行できるタスクは Task1 なので Task1 の残り時間は 2 から 0 に更新される．Task1 は終了する．

高優先度のタスクが終了してもただちに低優先度のタスクには切り替わらずに，高優先度のタスクの影響はあとで反映される．この例では時刻 4 で高優先度の Task2 が終了しているが，時刻 2 から時刻 4 までの時間は Task2 が実行されていた影響が Task1 に反映されるのは時刻 5 である．時刻 5 までに高優先度のタスクの終了が複数ある場合でも，それまでに一度も Task1 には切り替わらない．つまり低優先度のタスクが長時間動作して，その間に短時間で終了する高優先度のタスクが多数ある場合には，低優先度のタスクに対する高優先度のタスクの影響は 1 回にまとめて反映される．このような状況ではコンテクストスイッチの回数が減少する．

ラウンドロビンスケジューリング

まず Task1 が consume(5) を呼び出し，その 2 秒後に Task2 が consume(2) を呼び出すと仮定する．タスクに割り当てる時間の単位は 1 秒とする．2 つのタスクに呼び出された手続き consume は次のように処理される（**図 10**）．[**0 (Task1)**] Task1 の残り時間が 5 に初期化される．[**2 (Task2)**] 時刻 0 と 2 の間に Task1 は実行可能であり，Task1 の残り時間は 5 から 3 に更新される．Task2 の残り時間は 2 に初期化される．[**4 (Task2)**] 時刻 2 と時刻 4 の間に実行可能なタスクは Task1 と Task2 なので，Task2 の残り時間は 2 から 1 に，Task1 の残り時間は 3 から 2 に更新される．[**5 (Task2)**] Task2 は時刻 4 と 5 の間に実行可能なので，Task2 の残り時間は 1 から 0 に更新される．Task2 は終了する．[**5 (Task1)**] 時間は経過しなかったので，タスクの残り時間は更新されない．[**7 (Task1)**] 時刻 5 と 7 の間に実行可能なタスクは Task1 なので，Task1 の残り時間は 2 から 0 に更新される．Task1 は終了する．

Task1 と Task2 が実行可能な時刻 2 から 5 の間に CPU 時間が交互に割り当てられるが，コンテクストスイッチは 1 回しか起きない．特にタイムスライスを小さくしたときに潜在的なコンテクストスイッチの回

数を減らすことができる．

一方で，スケジューリングの影響を正確に遅延に反映させるために多量のコンテクストスイッチを発生させる従来手法と同じ正確さでスケジューリングの効果を計算できる．割込み優先度スケジューリングとラウンドロビンスケジューリングのどちらの場合にもconsume を終了させる時刻は従来手法と同じである．

5.2 SpecC との統合

ベースとなるシミュレーションプラットフォームが少数のプリミティブを提供すれば動作するようにアルゴリズムを設計したので，図 8 に示した実装を他のプラットフォームに容易に移植することができる．移植性を確認するために，まず SystemC で提案アルゴリズムを実現するスケジューラを実装し，次にこの実装に対して変更を行って SpecC で動作するようにした．SystemC では C++ を使ってシミュレーションプログラムを記述して，それが C++ ライブラリとして提供されるシミュレーション環境とリンクされる．一方 SpecC では専用言語で書かれたシミュレーションプログラムが C++ に変換されて，それがシミュレーションを実行する環境とリンクされる．ソフトウェアの遅延時間要求をシミュレーションプログラムからはconsume の呼び出しだけで使用できるようにするために，SystemC から SpecC の移植においてスケジューラをシミュレーションプログラム側から実行環境ライブラリ側に移動させた．

このような移植において変更を要した構成要素は**表 1** に示すものであった．時間データ型・時間 0 の表現・現在時刻の取得関数はシミュレーションプラットフォームによって必ず提供されるものである．また遅延時間要求は SystemC ではライブラリの API であるのに対して SpecC では実行環境の実装の一部であるという違いはあるが，シミュレーションプラットフォームに対して経過時間を指定する基本的な仕組みとして備わっている．このようにベースとなるシミュレーションプラットフォームに共通で少数のプリミティブしか要求しないので，提案アルゴリズムは移植性が高い．

表 1 SystemC と SpecC の構成要素
Table 1 Constructs of SystemC and SpecC.

	SystemC	SpecC
時間データ型	sc_time	sim_time
時間 0	SC_ZERO_TIME	0
現在時刻	sc_time_stamp()	now()
遅延時間要求	wait(delay)	_specc::waitfor(delay)

5.3 実装の改良

非同期割込み

ハードウェアとソフトウェアからなるシステムではハードウェアからソフトウェアへの割込みを扱える必要があるが，アルゴリズムをそのように改良できる．SystemC では手続き wait(time, event)（タイムアウト付きのイベント待ち）を使うと，消費時間の途中で割込みをとらえることが可能になる．イベントとタイムアウトのどちらが発生したかは wait(time, event) の前後での経過時間から判断できる．図 8 の while ループを次のように変更する：

```
while (remaining_time[task] > SC_ZERO_TIME) {
 t = sc_time_stamp() + remaining_time[task]; // 追加
 wait(remaining_time[task], event);
 update();
 last_time = sc_time_stamp();
 if (t != sc_time_stamp()) break;  // 追加
}
```

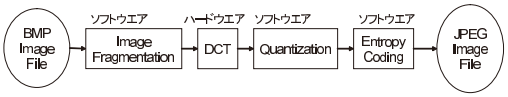
変更後の consume(time, event) は時間 time がすべてタスクに割り当てられるか，その間にイベントが発生するかのどちらかで終了する．また割込みハンドラの実行時間をスケジューリングに反映する必要がある場合には，イベント発生によって consume からリターンした後にハンドラを高優先度タスクとして再び consume を呼び出す．

OS のオーバヘッド

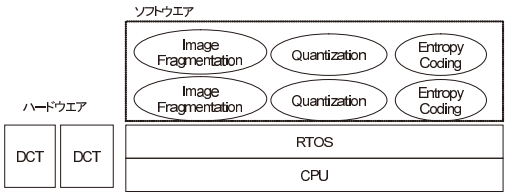
これまでは RTOS にオーバヘッドがないと仮定してきたが，RTOS のオーバヘッド（タスクコンテクストの保存，スケジューリング，タスクコンテクストの復元など）を扱うように拡張することが可能である．これは update の定義を，last_time と現在時刻の間の時間の一部分を RTOS の挙動に割り当てるようにすることで実現できる．

5.4 評　　価

我々のアルゴリズムの効果を，スケジューリングの影響を遅延に反映させるためにタイムスライスごとにコンテクストスイッチを発生させる従来手法と対比して評価した．対照スケジューラモデルではタイムスライスごとにスケジューラからソフトウェアタスクへのコンテクストスイッチと，各ソフトウェアタスクに割り当てる残り時間の再計算を繰り返す．残り時間の再計算の処理時間はコンテクストスイッチの処理時間と比して短いので，コンテクストスイッチ数の違いによるシミュレーション時間を比較した．対照スケジューラと同じコンテクストスイッチ数になるように，タイムスライスごとに _specc::waitfor(0) を呼び出すことによってコンテクストスイッチを発生させた．



(a) JPEG エンコーダの機能ブロック



(b) 2 並列 JPEG エンコーダの配置

図 11 実験に使用した JPEG エンコーダ
Fig. 11 JPEG encoder used in the experiment.

_specc::waitfor(0) は時間を進めないが，動作待ちのタスクがある場合にコンテクストスイッチを起こす．それ以外の処理は提案手法のものをそのまま使っているので，提案手法がコンテクストスイッチ数を減少させる効果だけを比較することになる．それ以外の違いは実験に反映されていない．

実験に SpecC の JPEC エンコーダのためのアーキテクチャレベル☆シミュレーションプログラム[13] を使用した．このプログラムではシステムの挙動はハードウェア部分とソフトウェア部分に分割されていて，プログラムの断片ごとに遅延時間がアノテートされている．我々はスケジューリングアルゴリズムの効果を評価する目的で，このプログラムに対して 2 画像を並列に処理するように変更を行った．変更後のプログラムではソフトウェア部分は 1 個の CPU を共有してスケジュールされるようにした（**図 11**）．またソフトウェア部分は遅延時間の指定に waitfor の代わりに consume を使用するように変更した．実験は IBM ThinkPad X40（メモリ 1,024 MB，OS Windows XP Professional, CPU Pentium M 1.4 GHz）☆☆で行った．シミュレーションプログラムに対する入力は 481 K のビットマップイメージで，81 K の JPEG ファイルにエンコードされる．

表 2 に示すように提案手法では対照 RTOS モデルが必要とする潜在的なコンテクストスイッチ数の多くを取り除くことができた．また対照 RTOS モデルでは，タイムスライスの大きさにほぼ反比例してコンテクストスイッチ数が増加する一方で，提案手法ではタイムスライスの大きさが 1/2 になってもコンテクストスイッチ数は 10%程度増加するだけである．これは提案手法では，発生するコンテクストスイッチはスケジューリングとは独立にアプリケーションの実行に必要なものが多くを占めていることを示している．その結果**表 3** に示すように，対照 RTOS モデルではタイムスライスの大きさの選択によってシミュレーション速度が変動するのに対して，我々のアルゴリズムではシミュレーション速度はタイムスライスの大きさとはほぼ関係なくなる．

表 2 タイムスライスサイズとコンテクストスイッチ数
Table 2 Time slice sizes and the number of context switches.

タイムスライス (単位時間)	コンテクストスイッチ数	
	対照 RTOS モデル	我々の手法
2,000	3.01 M	0.47 M
1,000	5.96 M	0.52 M
500	11.91 M	0.58 M

表 3 タイムスライスサイズとシミュレーション時間
Table 3 Time slice sizes and simulation time.

タイムスライス (単位時間)	シミュレーション時間	
	対照 RTOS モデル	我々の手法
2,000	18.8 s	18.4 s
1,000	20.7 s	18.5 s
500	24.5 s	18.5 s

6. 結　　論

本稿では RTOS 上で動作するソフトウェアタスクのシミュレーションを行うための新しいタスクスケジューリングアルゴリズムを提案した．我々のアルゴリズムではコンテクストスイッチの回数を大きく減らすことが可能で，効率の良いシミュレーションが可能になる．ベースとなるシミュレーションプラットフォームに少数のプリミティブしか要求しないので移植性が高い．またアルゴリズムの形式化と正当性の証明を行い，SystemC と SpecC を使ったアルゴリズムの実装について示した．

また我々は本稿のアルゴリズムを実際の業務に適用して有効性を確認している．複数のソフトウェアから使われる状況を考慮して画像処理用ハードウェアの性能の評価を行う際に，パラメータを少しずつ変化させた多数のシミュレーションを現実的な時間で実行することが可能になった．

☆ SystemC ではタイムド機能モデルに相当．

☆☆ IBM は IBM Corporation の商標．ThinkPad は Lenovo Corporation の商標．Windows は Microsoft Corporation の米国およびその他の国における商標．Pentium は Intel Corporation の米国およびその他の国における商標．

参 考 文 献

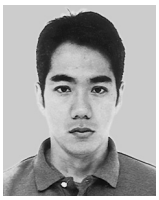
1) Bouchhima, A., Yoo, S. and Jeraya, A.: Fast and Accurate Timed Execution of High Level

Embedded Software Using HW/SW Interface Simulation Model, *Proc. Asia South Pacific Design Automation* (2004).
2) Cockx, J.: Efficient Modeling of Preemtion in a Virtual Prototype, *Proc. Rapid System Prototyping*, IEEE (2000).
3) Gerstlauer, A., Yu, H. and Gajski, D.D.: RTOS Modeling for System Level Design, *Proc. Design, Automation and Test in Europe* (2003).
4) Grotker, T.: Modeling Software with SystemC 3.0, *6th European SystemC User Group Meeting* (2002).
5) Hassan, M.A., Sakanushi, K., Takeuchi, Y. and Imai, M.: RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC, *Proc. Design, Automation and Test in Europe* (2005).
6) Hastono, P., Klaus, S. and Huss, S.A.: Real-Time Operating System Services for Realistic SystemC Simulation Models of Embedded Systems, *Proc. International Forum on Specification and Design Languages* (2004).
7) He, Z., Mok, A. and Peng, C.: Timed RTOS Modeling for Embedded System Design, *Proc. Real-Time and Embedded Technology and Applications Symposium* (2005).
8) Moigne, R.L., Pasquier, O. and Calvez, J.-P.: A Generic RTOS Model for Real-time Systems Simulation with SystemC, *Proc. Design, Automation and Test in Europe* (2004).
9) SpecC. http://www.specc.org/
10) SystemC. http://www.systemc.org/
11) SystemC Language Working Group: *Functional Specification for SystemC 2.0* (2002).
12) Yi, Y., Kim, D. and Ha, S.: Virtual Synchronization Technique with OS modeling for Fast and Time-Accurate Cosimulation, *Proc. Hardware/Software Codesign and System Synthesis* (2003).
13) Yin, H., Du, H., Lee, T.-C. and Gajski, D.D.: Design of a JPEG Encoder using SpecC Methodology, Technical Report ICS-TR-00-23, UC Irvine (2000).
14) Yoo, S., Nicolescu, G., Gauthier, L. and Jerraya, A.: Automatic Generation of Fast Timed Simulation Models for Operating Systems in SoC Design, *Proc. Design, Automation and Test in Europe* (2002).

(平成 19 年 1 月 9 日受付)
(平成 19 年 6 月 5 日採録)

中村　宏明（正会員）

平成元年東京大学大学院工学系研究科修士課程修了．同年日本アイ・ビー・エム（株）入社．ソフトウエア開発支援技術の研究に従事．著書『パターンとフレームワーク』（共立出版，共著）．ACM，日本ソフトウエア科学会各会員．



佐藤　直人（正会員）

1989 年東京大学理学部卒業．博士（理学）．現在，IBM 東京基礎研究所にて，ソフトウェア工学の研究に従事．ACM 会員．

田渕　　直（正会員）

1977 年生．2003 年東京大学情報理工学系研究科修士課程修了．同年より日本 IBM 東京基礎研究所に勤務．専門は型理論をはじめとするプログラム言語の基礎理論，プログラム解析．現在はスクリプト言語をはじめとする軽量・ドメイン特化言語および Web アプリケーションのセキュリティをテーマとして活動を行っている．