Simultaneous Multi Thread（SMT）技術を用いるCPU上で動作する計算資源予約システム

井上 拓† 森山 孝男†
根岸 康† 小原 盛幹†

Simultaneous Multi Thread（SMT）技術を用いる CPU では，計算資源の利用率を高めるために，複数のプロセッサスレッドが 1 つの CPU 上で同時に実行される．このとき，プロセスが一定時間内に得ることができる計算量は，同時に動作している他のプロセッサスレッドの影響を受けて大きく変動してしまう．そのため SMT 環境下ではリアルタイムプロセスに対してあらかじめ必要な CPU 時間を予約してスケジューリングを行っても，計算時間としては予約されているにもかかわらず，実際には意図しただけの計算量が得られないという問題が生じてしまう．そこで本研究では SMT 環境下でも予約された計算量の割当てを保証しつつ SMT の機能を活かしシステムのスループットを向上させることを目的とした CPU 資源の予約手法を提案する．

CPU Resource Reservation System for CPU Using Simultaneous Multi Thread

Hiroshi Inoue,† Takao Moriyama,† Yasushi Negishi† and Moriyoshi Ohara†

Simultaneous Multi Thread (SMT) techniques increase the CPU usage by allowing multiple threads to run on a physical processor at a very fine granularity. Operating systems, however, can control the resource allocation only at a coarser granularity than SMT mechanisms control the resource. Thus, the progress of a thread is not independent from that of other threads on the same processor. As a result, existing reservation-based real-time scheduling schemes cannot guarantee the resource allocation as they can do in non-SMT systems. In this paper, we propose a new resource reservation scheme for SMT systems. Our experimental results have shown that our scheme improves the system throughput by exploiting SMT facilities while guaranteeing the allocation of reserved resources.

1. はじめに

リアルタイム OS の 1 つの形態としてデッドラインを持つリアルタイムプロセスに最悪値での計算時間をあらかじめ申告させその計算時間を予約することで，デッドライン要求を確実に満たすようにスケジューリングを行うものがある．しかし 1 つの物理 CPU 上で複数のプロセッサスレッドが同時に動作する SMT 技術を用いた CPU では，一定時間内に行える計算量は同じ物理 CPU 上で同時に動作している他のプロセッサスレッド（以下，裏プロセッサスレッドと表記）との競合により大きく変動してしまう．その結果，SMT 環境下では計算時間としては予約されているにもかかわらず，実際には予約時に意図しただけの計算量が得られないという問題が生じてしまう．従来の CPU 資源予約システムに関する研究として，たとえば Rajkumar らによる Resource Kernel [1),2)] では，周期とその中でリアルタイムプロセスが使用すべき計算時間を明示して資源を予約し，予約されたプロセスに対しては確実に CPU 時間を割り当てることで，リアルタイムプロセスのデッドライン要求を満たすことを保証した．しかし，Resource Kernel ではプロセスがディスパッチされている間は CPU を占有できることを前提としているため，その前提が崩れる SMT 環境下ではそのままでは適用することができない．この問題は Resouce Kernel 以外の CPU 資源予約システム[3)] や，Rate Monotonic スケジューリング[4)] をはじめとするリアルタイムスケジューリングに関しても同様である．SMT を考慮したリアルタイムスケジューリングにつ

† 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.

いての研究として，Jain ら[5] によるソフトリアルタイムスケジューリングの研究があげられる．Jain らはプロセス間の干渉の影響が小さくなるように SMT の CPU 上に同時にディスパッチするプロセスを選択するというアプローチで，リアルタイムプロセスのデッドラインミスを減少させられることを示した．しかし，Jain らの手法では統計的にデッドラインミスを減らすことはできるが Resouce Kernel などのようにリアルタイムプロセスへの計算量の割当てを確実に保証できるわけではない．

SMT 環境下で確実な計算量の割当てを保証するためには単純には CPU 時間を予約しているリアルタイムプロセスがディスパッチされる間，SMT の機能を無効にし裏プロセッサスレッドを使用しないことで回避することができる[6]．しかし，システム内にはデッドラインを持たずベストエフォートで動作する非リアルタイムプロセスも存在し，リアルタイムプロセスへの資源の割当てを保証しながらも，これらの非リアルタイムプロセスにより多くの資源を割り当てることが求められる．SMT 技術は本来複数のプロセスを同時に実行することで CPU 内の計算資源の利用効率を高めることを目的としており[7]，スループット向上のためには予約されたリアルタイムプロセスと同時に他の非リアルタイムプロセスを裏プロセッサスレッドで動かすことが望ましい．すなわち，裏プロセッサスレッドを使いながらも，リアルタイムプロセスがデッドラインまでに予約しただけの計算量を獲得できることを保証する必要がある．

本研究では OS のタスクスケジューラにおいて，リアルタイムプロセスにあらかじめ予約した CPU 時間に相当する計算量の割当てを保証しながら，可能な限り裏プロセッサスレッドの利用率を高める手法を提案する．SMT を用いる CPU では，あるプログラムが実行されているとき，裏プロセッサスレッドにおいて同じ計算資源（演算ユニットなど）を要求するプログラムが走っている場合には，その資源での競合により単独で実行されている場合よりもそれぞれの計算速度が低下するが，まったく違う資源を要求するプログラムが走っている場合や，アイドルしている場合にはこのような競合は生じず計算速度の低下は小さい[8]．裏プロセッサスレッド上で同時にどのようなプログラムが実行されるかは，一般にプログラマには予測不可能であるため，計算速度の低下をあらかじめ考慮した資源予約を行うことは困難である．このような資源競合による予測不可能な計算速度の変動は，通常の SMP のシステムでも共有バスの競合などで生じるが，SMT のプロセッサではその影響が顕著になる．そこで本研究では，プログラマによる資源予約の時点では，裏プロセッサスレッドによる影響はないものとして予約を行い，実行時にその影響を補償する手法を提案する．

2. デッドラインを保証するためのスケジューリング手法

2.1 スケジューリング手法

本研究で提案するスケジューリング手法について示す．リアルタイムプロセス P1 が，デッドライン時刻 $t_{deadline}$ までに，$c_{reserve}$ だけの計算量を得ることを予約しているとする．P1 が CPU にディスパッチされたとき，まずデッドラインまでの余裕時間 $t_{slack} = T - C$ を計算する．ここで T はデッドラインまでの残り時間（$T = t_{deadline} - t_0$，t_0 は現在時刻），C はデッドラインまでに得るべき残り予約計算時間（$C = c_{reserve} - c_0$，c_0 は現在までに得た計算量に相当する計算時間）を表す．ただし計算時間 C については裏プロセッサスレッドの影響がない場合の計算時間をもって表すとする．ここで余裕時間が 0 以上の場合には裏プロセッサスレッドに他のプロセスをディスパッチすることで，システムのスループット向上に寄与する．

裏プロセッサスレッドに他のプロセスをディスパッチしている間は，予約された CPU の割当てを保証するために余裕時間の監視を行っていく必要がある．もし監視中に余裕時間がなくなった場合には，それ以降裏プロセッサスレッドを強制的にアイドル状態にすることで主プロセッサスレッドへの影響を取り除きリアルタイムプロセス P1 への予約された計算量の割当てを保証する．余裕時間の監視は一定間隔で発生するタイマ割込みを用いて行うか，可変の間隔で外部タイマ割込みを発生させてチェックすることにより行う．

2.2 外部タイマ割込みを用いるスケジューリング手法

余裕時間の監視に可変間隔の外部タイマ割込みを用いる場合は，タイマ割込みを発生させる時刻 t_{next} は次式で表される．

$$t_{next} = t_{slack} + t_0 \qquad (1)$$

この時刻 t_{next} に発生する割込みの割込みハンドラ内で余裕時間を確認し余裕時間がある場合には引き続き裏プロセッサスレッドを他のプロセスが使用し続けられる．余裕時間がなくなっている場合には，裏プロセッサスレッドをアイドル状態にし，予約されたプロセス P1 に全共有資源を占有使用させることでデッドラインまでに予約された計算量の割当てを保証する．裏プロセッサスレッドを他のプロセスが使用し続ける

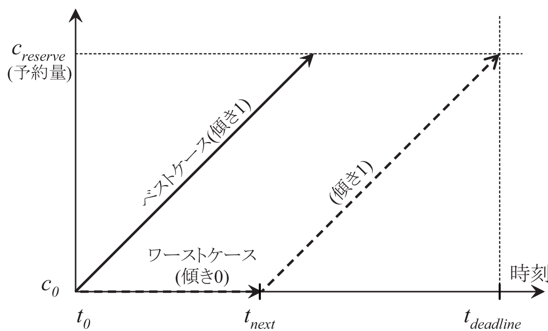


図 1 獲得した計算時間の推移
Fig. 1 Schematic image of calculation progress.

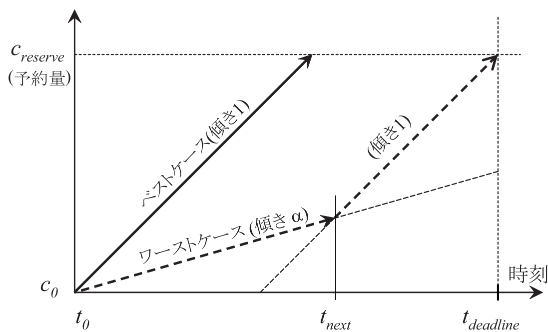


図 2 獲得した計算時間の推移（割込み削減時）
Fig. 2 Schematic image of calculation progress with reduction of timer interrupt.

場合には，確認時に再び次の割込みタイマの設定を行う．**図 1** に時刻と得られる計算量の関係について示す．グラフの横軸は時刻，縦軸はその時刻までに得られた計算量に相当する計算時間を示す．縦軸の計算時間は裏プロセッサスレッドの影響がない場合の時間で表しているので，裏プロセッサスレッドとの競合が生じないベストケースではグラフの傾きは 1 となる．一方，裏プロセッサスレッドの影響で計算時間をまったく得られないワーストケースでは傾きは 0 となる．しかし，その場合でも割込みのかかる時刻 t_{next} から先，裏プロセッサスレッドをすべてアイドル状態にして予約したプロセスに資源を占有させれば，デッドラインまでに予約された計算量が得られることが保証されるように t_{next} を定めている．t_{next} の時刻にタイマ割込みがかかり，まだ余裕時間（$t_{slack} = T - C$）が残っていて裏プロセッサスレッドを他のプロセスが使用し続ける場合には，t_{next} を新たな t_0，t_{next} までに実際にプロセスが得た計算量に相当する時間を新たな c_0 とし，式 (1) に従ってタイマ割込みを発生させる時刻を再度設定する．

余裕時間の確認のために，実際にプロセスが得た計算量を取得する必要がある．これには

(1) プログラムに自己申告させる，
(2) CPU のパフォーマンスモニタなどにより外部から情報を取得する，

という 2 つの方法がある．(1) ではプログラムの変更が必要になるが，パフォーマンスモニタなどの機能を持たない CPU においても実現が可能である．

2.3 タイマ割込み回数の削減

この手法ではデッドラインまでに計算時間を確実に割り当てられる代わりに，タイマ割込みによるオーバヘッドが生じる．このオーバヘッドを軽減するために，SMT を実装するプロセッサの性質として，裏プロセッサスレッドの影響がある場合にも最低限ある保証された計算性能が得られることが分かっている場合には，t_{next} の算出において以下の式を用いることで，割込みの回数を削減することができる．

$$t_{next} = (T - C)/(1 - \alpha) + t_0 \qquad (2)$$

ここで，α は裏プロセッサスレッドを使用した場合でも得られる計算性能の最低保証量を，裏プロセッサスレッドを使わなかった場合との比で表した値であり，$0 \leq \alpha < 1$ の範囲の値をとる．α が 1 に近いときは，裏プロセッサスレッドの影響をほとんど受けないことを表し，0 の場合には裏プロセッサスレッドにすべての資源をとられ実効性能がまったく得られない可能性があることを表す．

図 2 にこの場合の時刻と得られる計算量の関係を示す．図 1 と同様に縦軸の計算時間は裏プロセッサスレッドの影響がない場合の時間をもって表しているので，裏プロセッサスレッドとの競合が生じない場合にはグラフの傾きは 1 となる．一方，ワーストケースではグラフの傾きは α になる．図 2 に示すように，この割込みがかかった時刻 t_{next} から先，裏プロセッサスレッドをすべてアイドル状態にすれば，デッドラインまでに予約された計算量が得られることが保証される．この場合も先ほどと同様に，t_{next} の時刻の割込みがかかった場合には，再帰的に割込みタイマの設定を行う．

2.4 動 作 の 例

このスケジューリング手法の動作例を**図 3** に示す．この例では物理 CPU 上で実行されるプロセッサスレッド数が 2 のシステムにおいて，現在時刻を 0 とし，あるリアルタイムプロセスが 10 ms 後のデッドラインまでに 6 ms 分の計算時間を予約しているとする．また，余裕時間の確認は 2.2 節で述べた可変間隔のタイマ割

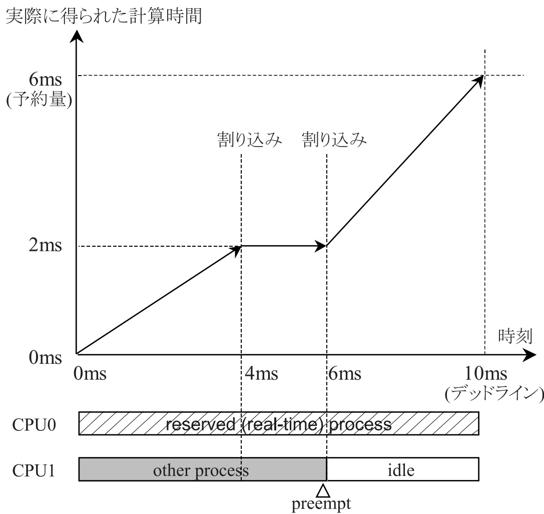


図 3　スケジューリング例
Fig. 3　Sample case.

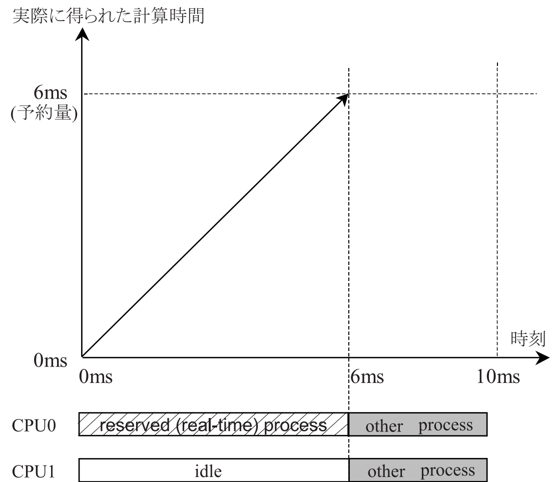


図 4　スケジューリング例（SMT 無効法）
Fig. 4　Sample case (idling method).

込みを用いる方法で行う．定期的に確認を行う場合でもこの例と同様に考えることができる．

まず，リアルタイムプロセスは CPU0 に優先的にディスパッチされ，この場合の余裕時間 $t_{slack} = T - C = 10\,\mathrm{ms} - 6\,\mathrm{ms} = 4\,\mathrm{ms}$ であり $t_{slack} > 0$ であるため，裏プロセッサスレッド（CPU1）を他のプロセスが使用できる．また，次の割込みを発生させる時刻は $t_{next} = t_{slack} = 4\,\mathrm{ms}$ である．次の割込みが生じる 4 ms の時点で，それまでに実際に獲得できた計算量を計測したところ 2 ms 相当の計算時間をすでに得ており，残りの予約時間が 4 ms だったとする．この場合には，余裕時間は $t_{slack} = 6\,\mathrm{ms} - 4\,\mathrm{ms} = 2\,\mathrm{ms}$ であり，まだ $t_{slack} > 0$ であるため，CPU1 は引き続き他のプロセスが使用できる．次の割込みを発生させる時刻は，$t_{slack} + 4\,\mathrm{ms} = 6\,\mathrm{ms}$ である．4 ms から 6 ms までの間，計算時間が獲得できず，6 ms の時点での残り予約時間が変わらず 4 ms だったとすると，この時点で余裕時間 $t_{slack} = 0$ であり，そのため以後 CPU1 をアイドル状態にすることで予約プロセスに資源を独占させ，予約された計算量がデッドラインまでに割り当てられることになる．この例では，デッドラインを持つリアルタイムプロセスは 1 つだけとして考えたが，これが複数になった場合には，デッドラインまでの余裕時間として，他のプロセスの処理時間を考慮する必要が生じる．

一方，この計算をリアルタイムプロセスの動作中は裏プロセッサスレッドを単純にアイドル状態にするという手法でスケジュールした例を**図 4** に示す．この場合，時刻 0 にリアルタイムプロセスが CPU0 にディスパッチされた時点で，CPU1 は強制的にアイドル状態となる．こうすることにより，CPU0 は CPU1 の影響を受けず，予約分の計算時間を 6 ms で獲得できることになる．そして残った 4 ms 分の時間は他のプロセスが CPU0，1 ともに使用できる．この手法はリアルタイムプロセスの動作中は SMT の機能を無効にしているものといえ，リアルタイムプロセスの負荷が大きい場合には，CPU が備える SMT の機能はほぼつねに使用できなくなる．

3. 実装および評価

3.1 実装の概要

前章で述べた手法を検証するためにデッドラインを明示してリアルタイムプロセスのスケジューリングを行うタスクスケジューラを Linux kernel 2.4.21 上に実装し評価を行った．このスケジューラの基本的な機能としては周期とその中で割り当てるべき CPU 時間およびデッドラインを明示的に指定することで予約を行い，予約された時間内では他のプロセスよりも優先的に CPU を割り当てるというものである[1),2)]．周期内での予約時間を使いきった場合や，周期内で行うべき計算が終了して自ら使用権を放棄した場合などには，次周期まで CPU の使用権を取得できない．マルチプロセッサシステムへの対応としては，予約時にどの CPU 上で時間を予約するのかを明示し，予約されたプロセスはその CPU 上でのみスケジュールされるようにした．

予約されたリアルタイムプロセスが動作するときの，裏プロセッサスレッドの取扱い方を以下の 3 種類とした．

(1) 考慮しない（以下，“従来法” と表記）．
(2) 図 4 の例のように，必ずアイドル状態にする（以下，“SMT 無効法” と表記）．
(3) 図 3 の例のように，可能な限り他のプロセスをディスパッチし，余裕時間がなくなったところでアイドル状態にする（以下，“余裕時間監視法” と表記）．

本研究で提案している手法である余裕時間監視法の実装について示す．余裕時間の監視に用いる外部タイマ割込みとしてはシステムタイマの割込み（IRQ0）を用いた．そのため，システムタイマを標準 Linux の一定周期で割込みを発生する設定から，指定した時間後に 1 度だけ割込みを行う設定に変更し，システムタイマの割込みハンドラにおいて次回の割込みを設定するように変更を加えた．また，割込みタイマの時刻を決定するためには式 (1) を用いており，2.3 節で述べた式 (2) を用いるタイマ割込み頻度の削減はここでは行っていない．余裕時間の確認はプログラムに自己申告させる方法をとり，リアルタイムプロセスが専用のシステムコールを用いて一定時間間隔で進捗を報告するようにした．また，式 (1) では右辺を 0 としており余裕時間が 0 になるまでチェックを繰り返すことになるが，右辺を 0 とすると非常に細かい間隔で割込みが多発してしまいオーバヘッドが大きくなることがあるため，この実装では割込みハンドラの処理時間を考慮して余裕時間が $10\mu s$ 以下になった時点で，裏プロセッサスレッドをアイドル状態にするようにした．

3.2 評価

本研究では Intel 社の SMT の実装である Hyper Threading（以下 HT と表記）を有効にした Xeon プロセッサを搭載したワークステーションを用いて評価を行った．この環境では物理的に搭載された 1 つの CPU が OS からは 2 つの論理 CPU として認識される．

評価はシステム内にデッドラインを持つリアルタイムプロセスは 1 つのみで，あとはベストエフォートで動作する非リアルタイムプロセスであるという条件で行った．非リアルタイムプロセスの種類を変え，リアルタイムプロセスへの予約計算量の割当ての可否，および非リアルタイムプロセスのスループットの 2 点について評価を行った．CPU を予約して動作するリアルタイムプロセスとしては，double 型を要素とする 200 行 × 200 列の行列どうしの行列積を各周期に 1 回ずつ計算するプログラムを用いた．このプログラムを実行し各周期での計算時間を測定することでリアルタイムプロセスへの予約計算量の割当てについての評価を行った．このプログラムでは SIMD 演算命令（SSE 命令）は用いず，通常の浮動小数点演算命令で計算を行っており，ループ・ブロッキングなどの計算を高速にするための最適化は行っていない．評価に用いたマシンでは 1 回の行列積計算におよそ 50 ms の時間を要する．この計算を 70 ms 周期で繰り返し行わせ，次周期が開始されるまでに前の計算を終えることをデッドラインと定義した．CPU 予約としては 70 ms 中の 55 ms をこのプロセスに予約して計算を行った．すなわち，単独で実行したときに 55 ms かかるだけの計算量を得るまではリアルタイムプロセスが優先的にディスパッチされる．SMT を用いないシングルプロセッサシステムで本計算を行った場合には，負荷になる非リアルタイムプロセスを同時に実行しても，CPU 予約が適切に動作しデッドラインミスは発生しないことを確認している．

3.2.1 予約された計算量の割当ての保証

まず，SMT 環境下においてシステム内の非リアルタイムプロセスの影響を受けずに，リアルタイムプロセスに予約した計算量が割り当てられることを確認する．**図 5** に各周期での計算時間の推移を非リアルタイムプロセスとして異なる 3 種類のプログラムを実行した場合についてそれぞれ示す．横軸は周期単位で表した時刻，縦軸はその周期での計算終了までにかかった時間である．この計算時間がデッドラインである 70 ms 以下であるとき，デッドラインまでに予約した計算量が割り当てられ計算が終わっていることを意味する．ここで非リアルタイムプロセスとして用いたプログラムは次の 3 種類であり，それぞれ．図 5 のグラフに順に対応する．

(a) int 型を要素とする行列の行列積を計算するプログラム
(b) double 型を要素とする行列の行列積を計算するプログラム（リアルタイムプロセスと同じプログラム）
(c) IDE 接続のハードディスク（ext3 ファイルシステム）上で Linux カーネルのコンパイルを繰り返すプログラム

図 5 より，手法によって明らかな差が生じていることが分かる．まず，裏プロセッサスレッドの影響を考慮しない従来の予約手法では，非リアルタイムプロセスとしてどのプログラムを実行した条件でもデッドラインミスが多発している．これはリアルタイムプロセスは片側のプロセッサスレッドに優先的にディスパッチされるが，裏プロセッサスレッド上でも他のプロセスが動作してしまうために，それらのプロセス間で CPU

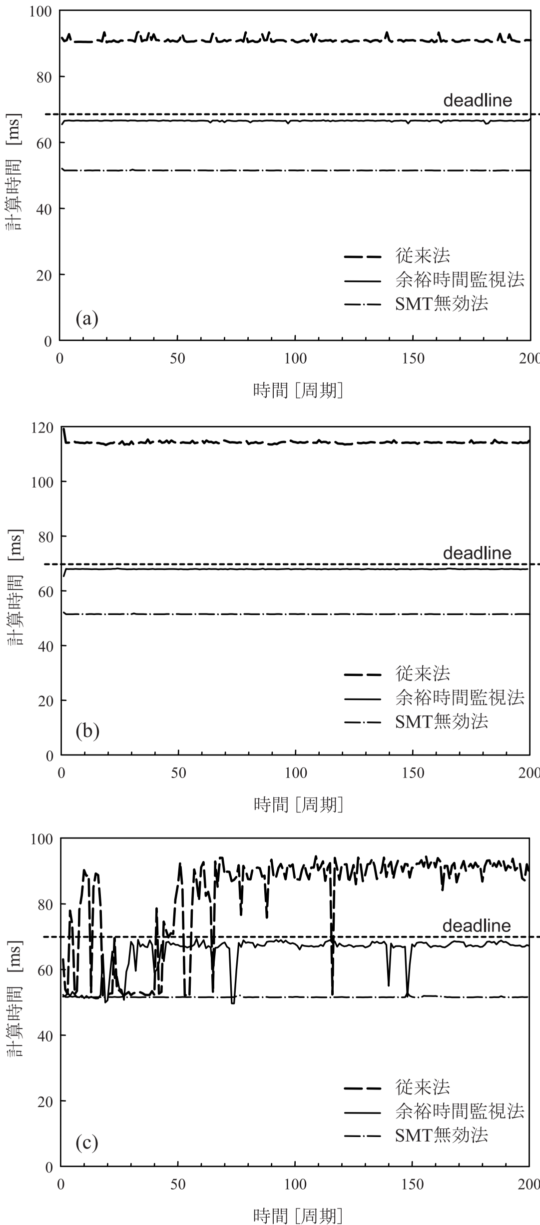


図 5 各周期での所要計算時間の推移．(a) 整数行列演算，(b) 浮動小数点行列演算，(c) カーネルコンパイル

Fig. 5 Calculation time in each cycle. (a) integer calculation, (b) floating point calculation, (c) kernel compile.

内の共有資源の競合が発生したことによる．すなわち，この資源の競合により裏プロセッサスレッドが使用されなかった場合と比較して計算が遅くなりデッドラインまでに計算が終了しなかったものである．この結果より，SMT 環境下での CPU 資源を予約してのプロセスの実行を考えるためには，同じ物理 CPU 上で実行される他のプロセスの影響を考慮することは必須といえる．

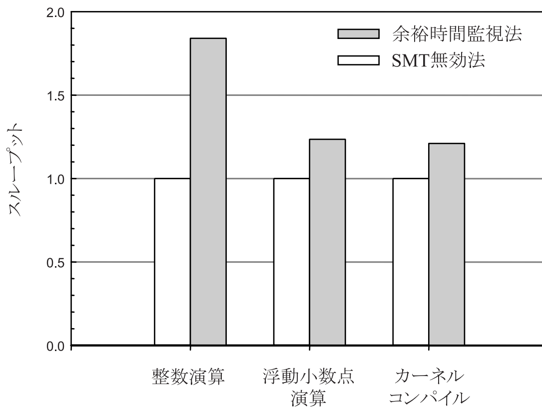
SMT 無効法を用いた場合には，裏プロセッサスレッドの影響がほぼ完全に取り除かれ，非リアルタイムタスクの種類によらずつねに一定の計算速度が得られておりデッドラインミスは生じていない．これはリアルタイムプロセスの動作中に裏プロセッサスレッドをアイドル状態にしたことにより，SMT の機能がないシングルプロセッサシステムで実行しているのと同様になったためである．

余裕時間監視法を用いた場合には，裏プロセッサスレッドを可能な限り他のプロセスに使わせるために，他のプロセスの影響を受けて計算速度が変動しながらも，デッドラインミスは発生していないことが確認される．これらの結果から，余裕時間監視法は単純に裏プロセッサスレッドをアイドル状態にする SMT 無効法と同様に，SMT 環境下で予約された計算量を確実に割り当てることができる手法であるといえる．

図 5 で余裕時間監視法の結果として示したものは，割込みタイマの時刻を決定するために式 (1) を用いているが，前述のように HT で 2 つのプロセスを同時に実行した場合に最低保証される計算性能が確定できれば，式 (2) を用いてタイマ割込みの頻度を減らすことができる．現在の HT の実装では共有する L1，L2 キャッシュでの競合は場合によっては非常に大きな性能悪化をもたらすため，一般的には計算性能の保証はできない．しかし本研究で用いているプログラムでは最悪でも 1/2 までの性能悪化しか生じないことが確認されたため，式 (2) を用いて割込みの削減を行った場合の評価も行い，この場合にも，デッドラインの保証はなされていることを確認した．また，周期ごとの割込み回数は，たとえば非リアルタイムプロセスとして整数計算が実行されている場合で平均 14.3 回から 6.6 回に減少した．

3.2.2 非リアルタイムタスクのスループット

リアルタイムプロセスに対してデッドラインまでに予約された計算量を割り当てることが可能な 2 つの手法（SMT 無効法，余裕時間監視法）について，リアルタイムプロセスが使用しなかった資源を用いてベストエフォートの動作をする非リアルタイムプロセスのスループットを比較する．前項の条件で実行を行った際の各非リアルタイムプロセスのスループットをそれぞれ**図 6** に示す．値は SMT 無効法の結果を 1 とした場合との比により表した．ここで整数および浮動小数点の行列演算のスループットは単位時間あたりに計算できた行列数で定義した．カーネルのコンパイルについてはコンパイルが終了するまでの時間を time コマ



	余裕時間監視法	SMT無効法
整数演算	1.84	1.00
浮動小数点演算	1.23	1.00
カーネルコンパイル	1.21	1.00

図 6 非リアルタイムプロセスのスループット
Fig. 6 Throughput of non real-time process.

ンドで測定し，その逆数をスループットとした．この計測中リアルタイムプロセスのデッドラインミスはどちらの手法を用いた場合でも発生していないことを確認している．

図 6 より余裕時間監視法を用いた場合には SMT 無効法と比較して整数演算で 80%程度スループットが向上していることが確認される．これは本手法ではリアルタイムプロセスの動作中も裏プロセッサスレッドを使用可能にしたことにより，CPU の計算資源の利用率が高まったことによる．一方，浮動小数点演算については，スループットの向上は整数演算の場合よりも小さい．これはリアルタイムプロセスと裏で動作する非リアルタイムプロセスが同じ特徴を持つプログラムであり，同じ資源を要求したため，SMT を用いることによる性能向上が小さいためである．このように余裕時間監視法では計算資源の利用率を高めることで，SMT 無効法と比較しリアルタイムプロセスのデッドラインを保証しながらもスループットの向上が得られ，その向上率は同時に実行されるプロセス間で資源が競合する割合に依存するといえる．

また余裕時間監視法において式 (2) を用いて割込みの削減を行った場合についても同様にスループットの計測を行った．その結果，図 6 に示した割込みの削減を行わない余裕時間監視法の場合との間に有意な差は見られなかった．本条件ではこの割込みの削減により，1 秒間あたりの割込みの回数が 100 回程度減少しているが，1 回あたりの割込みオーバヘッドが大きくないため，明確なスループットの向上が見られなかったものと考えられる．

より現実的なアプリケーションといえる Linux カーネルのコンパイルでは，余裕時間監視法を用いた場合は SMT 無効法を用いた場合と比較して 20%程度スループットが向上していることが確認される．ディスクアクセスによる I/O ブロックを含むような一般的なワークロードの非リアルタイムプロセスに対しても余裕時間監視法は有効であることが確認できる．

本研究ではシステム内にリアルタイムプロセスが 1 つであるという条件の下に評価を行った．今後の研究としては，より一般的に複数のリアルタイムプロセスが存在し，その間に優先度の関係があるような場合についても詳細な検討を行う必要がある．また，SMT のプロセッサで同時に実行されるプロセッサスレッド数が 2 を超えて増えた場合や，SMT のプロセッサが物理的に複数搭載された場合を対象とした資源予約システムも今後の課題である．

4. おわりに

本研究では SMT 技術を用いる CPU 上で，タイマによる外部割込みを使用してリアルタイムプロセスへ予約された計算量の割当て保証する手法を提案した．SMT を用いるプロセッサでは，あるプロセスが一定時間に獲得できる CPU の性能が裏プロセッサスレッドの影響で大きく変動してしまう．そのためディスパッチされている間は CPU を占有できることを前提としている過去の CPU 資源予約システムに関する研究はそのままでは適用が困難である．本研究で提案した手法はデッドラインまでの余裕時間を確認して可能な限り裏プロセッサスレッドを使用することで，予約された計算量割当ての保証とスループットの向上を両立させた．この手法を Linux に実装し，HT を使用する CPU 上で検証を行った結果，本手法ではリアルタイムプロセスへの予約された計算量の割当てを保証すると同時に，SMT の機能を活かし非リアルタイムプロセスを含むシステム全体のスループットを向上させることが確認できた．

SMT 技術は今後，リアルタイム処理を必要とする組み込み向け CPU などでも採用されるのみでなく，システムの性能を引き出すためには SMT の機能をうまく使いこなすことが重要になってくると考えられる．そのため SMT を用いるプロセッサ上でのリアルタイムスケジューリングについても，今後その重要性が拡大すると思われる．

参 考 文 献

1) Rajkumar, R., Juvva, K., Molano, A. and Oikawa, S.: Resource Kernels: A Resource-Centric Approach to Real-Time Systems, *Proc. Conference on Multimedia Computing and Networking*, SPIE/ACM (1998).
2) Oikawa, S. and Rajkumar, R.: Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior, *Proc. Real-Time Technology and Applications Symposium*, IEEE (1999).
3) Jones, M.B., Rosu, D. and Rosu, M.: CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities, *Proc. Symposium on Operating Systems Principles*, ACM (1997).
4) 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).
5) Jain, R., Hughes, C.J. and Adve, S.V.: Soft Real-Time Scheduling on Simultaneous Multithreading Processors, *Proc. Real-Time Systems Symposium*, IEEE (2002).
6) 井上　拓：SMT 環境下における CPU 資源予約，プログラミングおよび応用のシステムに関するワークショップ，JSSST (2003).
7) Ungerer, T., Robi, B. and Silc, J.: A survey of processors with explicit multithreading, *ACM Computing Surveys*, Vol.35, No.1, pp.29–63 (2003).
8) Snavely, A. and Tullsen, D.: Symbiotic Jobscheduling for a Simultaneous Multithreading Processor, *Proc. Architectural Support for Programming Languages and Operating Systems*, ACM (2000).

(平成 15 年 7 月 31 日受付)
(平成 15 年 10 月 31 日採録)

井上　　拓（正会員）

1977 年生．2000 年慶應義塾大学理工学部システムデザイン工学科卒業．2002 年同大学院理工学研究科総合デザイン工学専攻修士課程修了．同年日本アイ・ビー・エム（株）に入社．現在同社東京基礎研究所に勤務．オペレーティングシステム，ハイパフォーマンス・コンピューティングシステムの研究に従事．

森山　孝男（正会員）

1962 年生．1985 年東京工業大学工学部情報工学科卒業．1987 年同大学院修士課程修了．同年日本アイ・ビー・エム（株）に入社．現在同社東京基礎研究所に勤務．並列マシンのオペレーティングシステム，グラフィックスの並列処理の研究に従事．

根岸　　康（正会員）

1964 年生．1987 年東京工業大学理学部情報科学科卒業．1989 年同大学院修士課程修了．同年日本アイ・ビー・エム（株）に入社．現在同社東京基礎研究所に勤務．システムソフトウェア，通信プロトコルの研究に従事．

小原　盛幹

1963 年生．1986 年東京大学工学部計数工学科卒業．同年日本アイ・ビー・エム（株）に入社．1996 年米国スタンフォード大学 Ph.D. 現在同社東京基礎研究所に勤務．マルチプロセッサ・システム，超高解像度ディスプレイの研究に従事．