情報処理学会論文誌 Vol.54 No.7 1835–1848 (July 2013)

ITRON仕様OSのRMT Processor向け実装

上田 陸平[1,a)] 藤井 啓[1] 千代 浩之[1] 松谷 宏紀[1] 山崎 信行[1]

受付日 2012年11月5日, 採録日 2013年2月1日

概要：近年の組込みシステムの高機能化にともない，リアルタイム性に加えシステム全体の高スループット化が求められている．Responsive Multithreaded Processor（RMT Processor）は，このような組込みシステム向けに優先度付き Simultaneous Multithreading（SMT）アーキテクチャを採用しており，ハードウェアレベルでのスレッド制御を行うための機能を有している．本論文では，ITRON 仕様のリアルタイム OS を，RMT Processor 向けに設計，実装した．その際，スレッドの生成やディスパッチには RMT Processor 固有のスレッド制御命令を使用し，タスクの切替え時間を短縮した．また，同時実行スレッド数を増やした際の最悪割込み応答時間を一定にするため，割込み専用スレッドを設けた．評価により，ハードウェアによるスレッド制御によってコンテキストスイッチに関係する API の実行時間が短縮されたこと，それにともないスケジュール成功率が向上したこと，割込み専用スレッドにより最悪割込み応答時間を一定にできていることを示した．

キーワード：リアルタイム OS

Implementation of ITRON Specification OS for RMT Processor

Rikuhei Ueda[1,a)] Kei Fujii[1] Hiroyuki Chishiro[1]
Hiroki Matsutani[1] Nobuyuki Yamasaki[1]

Received: November 5, 2012, Accepted: February 1, 2013

Abstract: Recent advances in embedded systems demands high-performance under real-time constraints. Responsive Multithreaded Processor (RMT Processor) employs prioritized Simultaneous Multithreading (SMT) architecture with hardware-assisted context switch for such high-end embedded systems. In this paper, real-time OS based on ITRON specification is extended to support multithread processing on RMT Processor. Thread-control instructions, which are specific to RMT Processor, are used for thread creation and dispatch to reduce the context switching overhead. A dedicated thread is assigned to interrupt processing in order to maintain the worst-case interrupt response time when the number of threads executing simultaneously increases. Experimental results show that service calls that include context switch operations are accelerated by the hardware-assisted context switch, task schedulability is also improved accordingly, and the dedicated thread for interrupt processing can maintain the interrupt response time.

Keywords: real-time OSes

1. はじめに

近年の組込みシステムでは，ヒューマノイドロボットや携帯電話のように，高性能なアプリケーションが必要とされるのが一般的になりつつある．たとえば，ヒューマノイドロボットでは画像処理のような高いスループットを要求するタスクがあるように，リアルタイム性だけでなくシステム全体の高スループット化が求められる．その一方で，組込みシステムではハードウェア資源や消費電力に制限がある場合が多く，シングルプロセッサの動作周波数を向上させることで高スループットを実現するには限界がある．マルチコアプロセッサは，プロセッサの動作周波数ではなくプロセッサの並列性を高めることで，スループットを向上させるアプローチである．これは消費電力を抑えつつ性能向上を達成できる一方で，プロセッサコア数に比例して面積が大きくなるほか，プロセッサコア内部での空き資源

[1] 慶應義塾大学大学院理工学研究科
Graduate School of Science and Technology, Keio University, Yokohama, Kanagawa 223–8522, Japan
[a)] riku@ny.ics.keio.ac.jp

を他のプロセッサコアが使用することはできないといった問題がある．

プロセッサのマルチスレッド化も，マルチコアプロセッサと同様にプロセッサの並列性を高めることでスループットを向上させるアプローチである．複数のスレッドを同時に実行できる Simultaneous Multithreading（SMT）[14] は，従来のスーパスカラや細粒度マルチスレッディングに比べ 2〜3 倍のスループットを達成できることが知られている [5]．SMT は各スレッドがハードウェア資源を共有するため，並列実行にともない資源競合が発生する可能性があるが，空き資源が活用されないという事態は発生しない．また，ハードウェア資源量あたりのスループットもマルチコアプロセッサに比べて高くなるといった利点がある [5]．このような SMT プロセッサの例としては，POWER7 [15]，Intel Atom [7]，Responsive Multithreaded Processor（RMT Processor）[16] などがあげられる．特に，RMT Processor は組込みシステム向けに優先度を導入した優先度付き SMT アーキテクチャを採用しており，ハードウェアレベルでコンテキストスイッチを行うことで，タスク切替えにかかるオーバヘッドを大幅に削減できる．

組込みシステムにおいて，アプリケーションのリアルタイム処理を実現するにはリアルタイム OS（RT-OS）[3], [4], [11], [17], [20], [24] の利用が必要不可欠である．RT-OS は複数のタスクを管理するため，タスクの切替え時にコンテキストスイッチが発生する．このコンテキストスイッチにかかるオーバヘッドの増大は，タスクごとに時間制約のあるリアルタイムシステムにおいて，スケジュール成功率を低下させる要因の 1 つとなる．これまでに，機能分散マルチプロセッサ向け RT-OS [23] や RT-OS における一部機能のハードウェア化 [21], [22] が報告されているが，リアルタイムシステムで必須となる優先度制御を導入した優先度付き SMT アーキテクチャにおける RT-OS については報告されていない．文献 [23] では，タスクの各プロセッサへの割当てはユーザが定義し，プロセッサ内においては優先度を考慮してタスクを実行する．一方，優先度付き SMT アーキテクチャでは，最大 8 つのハードウェアコンテキストを用いた優先度付き SMT 実行によって，静的優先度付きスケジューリングを用いた複数スレッドのリアルタイム実行が可能であり，このためには SMT アーキテクチャ固有のスレッド制御命令を用いた RT-OS への拡張が必要である．

そこで本論文では，国内で広く普及している μITRON4.0 仕様 [20] に準拠した RT-OS を RMT Processor 固有のスレッド制御命令を用いて実現し，コンテキストスイッチにともなうオーバヘッドの大幅な削減およびスケジュール成功率の改善を実証する．また，マルチスレッドプロセッサの特徴を活かし，割込み専用スレッドを設けることにより，同時実行スレッド数の増加にともなう割込み応答時間の増大を抑制する．なお，RT-OS のマルチスレッド化にあたっては，ITRON 仕様 OS を機能分散マルチプロセッサ向けに実装した TOPPERS/FDMP カーネル [23] の実装を参考にした．

本論文の構成は次のとおりである．まず，2 章で既存の RT-OS を調査し，3 章では，本研究の実装対象である RMT Processor について説明する．4 章で本研究で提案する RMT Processor 向け ITRON 仕様 OS の拡張仕様について述べ，5 章で実装技術について述べる．6 章で実装した ITRON 仕様 OS の評価およびその考察について述べ，7 章で本論文をまとめる．

2. 既存の組込みリアルタイム OS

RT-OS は，リアルタイムシステム向けに時間資源の保護および実行時間の予測可能性を提供することを主眼に設計された OS であり，以下に示すような RT-OS が広く普及している．

OSEK/VDX [11] は，自動車のエンジンコントロールユニットに搭載されるソフトウェアの API を標準化することを目的とした RT-OS の仕様であり，欧州で広く利用されている．Non-preemptive/preemptive/Mix-preemptive の 3 つのスケジューリング機能を選択できる．

AUTOSAR OS [3] は，自動車メーカやソフトウェア開発企業により構成される標準化団体 AUTOSAR（AUTomotive Open System ARchitecture）により公開されている RT-OS 仕様である．選択するスケーラビリティクラス（SC）によって，時間保護やメモリ保護などの機能が追加される．2009 年にマルチプロセッサ向けの RT-OS 仕様も追加された．

汎用 OS をリアルタイムシステム向けに拡張した RT-OS の例としては，Linux をリアルタイム拡張したリアルタイム Linux があげられる [4], [17], [24]．リアルタイム Linux では，Linux のソフトウェアライブラリやデバイスドライバといったソフトウェア資産が再利用可能であるといった利点がある反面，Linux カーネルの更新にともない修正部分の設計を見直す必要があるため，将来的な拡張性に問題があるという欠点もある．

ITRON [20] は，1984 年に開始された TRON プロジェクトの一環として標準化されている組込み制御用の RT-OS 仕様である．1987 年に ITRON1 仕様が公開されてから様々な機能が追加されており，ITRON3.0 仕様からはより小規模なシステムを対象とした μITRON 仕様と統合し，現在では μITRON4.0 仕様が公開されている．ITRON 仕様に準拠した RT-OS は国内で多く開発および利用されており，トロン協会による 2010 年のアンケート調査によると，製品中に組み込んだ API の 51%が ITRON 仕様の API であるという結果が出ている [18]．また，ITRON 仕様 OS をマルチプロセッサ向けに拡張した例も報告されている [19], [23]．

しかし，リアルタイムシステムで必須となる優先度制御を導入した優先度付き SMT アーキテクチャにおける RT-OS については報告されていない．

これらの RT-OS においては，1) 複数のタスクを管理すること，2) タスク間の同期通信機構を提供することといった一般的な OS の機能に加え，3) タスクの実行を決められたデッドラインまでに完了させること，4) 高優先度のタスクを確実に実行することが要求され，このような制約下において，いかにスケジュール成功率を向上させるかが RT-OS およびリアルタイムスケジューリングの評価指標の 1 つとなっている．

本論文では，μITRON4.0 仕様に準拠した RT-OS である Hyper Operating System（HOS）[6] をベースとして，RMT Processor 向けに設計・実装を行う．組込みシステムではあらかじめアプリケーションが決まっていることが多く，汎用システムとは異なり，アプリケーションに合わせてハードウェアとソフトウェアを設計することで高効率なシステムを実現できる．本実装では，RMT Processor のハードウェア機能を用いることで，タスク切替えのオーバヘッド削減，スケジュール成功率の改善，割込み応答時間の安定化を実現する．

他にハードウェアの機能を利用して高速なタスク切替えを実現した例としては，ARTESSO（Advanced Real Time Embedded Silicon System Operator）RTOS [21], [22] がある．ARTESSO プロセッサは，ハードウェアでタスク情報とキュー情報を保持するなど，タスクの高速なキュー操作を実現するための機構を持っており，ARTESSO RTOS は ARTESSO プロセッサのハードウェア機能を用いて高い処理性能を実現している．ARTESSO RTOS は ITRON 仕様に互換性のある 30 種類の API を提供しているが，これは主として RT-OS におけるキュー操作の高速化が目的であり，本研究のようなハードウェア機能を利用した RT-OS のマルチスレッド化とは異なる．

3. Responsive Multithreaded Processor

3.1 概要

Responsive Multithreaded Processor（RMT Processor）[16] は，リアルタイム処理をハードウェアレベルで支援する RMT Processing Unit（RMT PU）をプロセッシングコアに持ち，コンピュータ用 I/O（DDR SDRAM I/F，DMA コントローラ，PCI64 I/F，IEEE 1394 I/F など），制御用 I/O（PWM ジェネレータ，パルスカウンタなど）を 1 チップに集積したシステム LSI である．RMT Processor における命令セットは MIPS の命令を踏襲しており，加えて RMT Processor 固有の命令を持つ．

3.2 優先度付き SMT

RMT Processor は，SMT 実行に優先度を導入した優先度付き SMT 実行によって，シングルスレッド実行時の性能向上と，複数スレッドの並列実行によるシステム全体の性能向上を実現している．従来のソフトウェアによるコンテキストスイッチを優先度付き SMT 実行に置き換えることにより，コンテキストスイッチを行わずに優先度の高いスレッドから順に同時実行することが可能である．また複数スレッドを同時に実行することで，プロセッサ全体の性能をスーパスカラと比較して大幅に高めることができ，一方で単一スレッドのみを動作させた場合でも，スーパスカラと同等の性能を発揮できる．**図 1** に RMT Processor におけるリアルタイム処理の様子を示す．

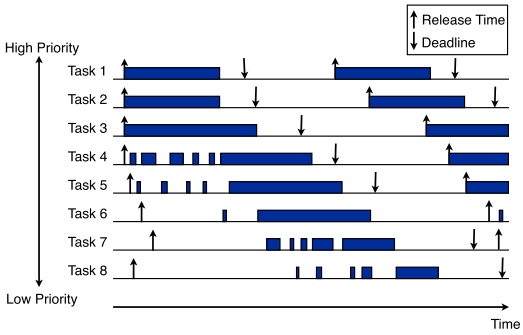


図 1 RMT Processor による 8 スレッド並列実行
Fig. 1 Parallel execution with 8 threads on RMT Processor.

RMT Processor は 8 つのハードウェアコンテキストを保持しているので，スレッド数が 8 以下の場合には，優先度付き SMT 実行によって静的優先度スケジューリングのようなリアルタイム実行が可能である．図 1 を見ると，優先度の高い Task 1〜Task 3 の実行はその他の低優先度スレッドの実行に影響を受けておらず，優先度の高いスレッドの実行効率の変動および低下を抑制できていることが分かる．優先度は，Rate Monotonic（RM）[8] スケジューリングアルゴリズムを効果的に適用するために必要とされる 256 段階で設定可能である．

3.3 コンテキストキャッシュ

通常コンテキストスイッチが発生すると，ソフトウェアによってメモリにコンテキストを退避させる必要があり，オーバヘッドが大きい．RMT Processor では，このようなコンテキストスイッチのオーバヘッドを削減するために，オンチップで 32 スレッドのコンテキスト情報を保持できるコンテキストキャッシュを搭載している．コンテキストキャッシュとハードウェアコンテキストとの間で専用バスを介してデータ転送を行うことで，コンテキストスイッチは 4 クロックで完了する．**図 2** にコンテキストキャッシュを使用したコンテキストスイッチの概略を示す．コンテキストスイッチで入れ替えるデータは汎用レジスタ，浮動小数点レジスタおよび制御レジスタである．ハードウェアコンテキストが割り当てられているスレッドをアクティ

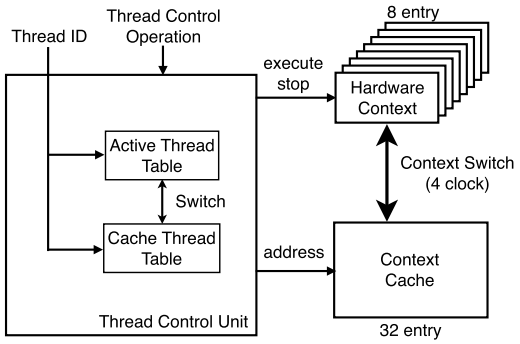


図 2 コンテキストキャッシュを使用したコンテキストスイッチ
Fig. 2 Context switch using context cache.

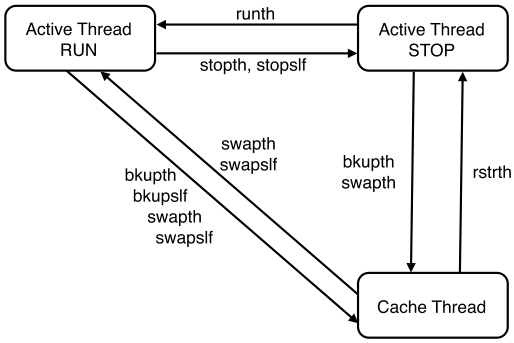


図 3 RMT Processor におけるスレッドの状態遷移図
Fig. 3 State transitions of threads on RMT Processor.

ブスレッド，コンテキストキャッシュに格納されているスレッドをキャッシュスレッドと呼び，これらのスレッドは Thread Control Unit 内で管理されている．スレッドを識別するために各スレッドにはスレッド ID が付与されており，アクティブスレッドとキャッシュスレッド間のコンテキストスイッチは RMT Processor 専用のスレッド制御命令を用いて，入れ替えるスレッドのスレッド ID を Thread Control Unit に伝えることで行われる．

3.4 スレッド制御機構

RMT Processor はスレッドの状態を**図 3** のように定義している．プロセッサが実行するスレッドは，レジスタファイルやプログラムカウンタなどの資源が確保されており，かつ実行状態（Active Thread RUN）にあるスレッドのみである．スレッドの状態は図 2 の Active Thread Table，Cache Thread Table と対応しており，図 3 に示されているスレッド制御命令によって状態遷移が行われる．スレッド制御命令の詳細を**表 1** に示す．

3.5 外部割込みによるスレッド起床

RMT Processor では，外部イベントによるスレッド制御を行う際の応答時間を短縮するために，停止状態（Active Thread STOP）にあるスレッドに対して外部割込みがかかった場合，そのスレッドを実行状態（Active Thread RUN）にすることができる．本研究では，この機能を用いて割込みに応答する専用スレッドを設けることで，割込みに対する応答性を向上させる．

表 1 スレッド制御命令
Table 1 Thread-control instructions.

命令	内容
mkth	新たにスレッドを生成する
delth	指定したスレッドを削除する
runth	指定スレッドを Active Thread RUN 状態に遷移させる
stopth	指定スレッドを Active Thread STOP 状態に遷移させる
stopslf	自身を Active Thread STOP 状態に遷移させる
bkupth	指定スレッドをコンテキストキャッシュへ退避する
bkupslf	自身をコンテキストキャッシュへ退避する
rstrth	指定したキャッシュスレッドを復帰させる
swapth	アクティブスレッドとキャッシュスレッドを交換する
swapslf	自身と指定したキャッシュスレッドを交換する

4. 拡張仕様

本章では，ITRON 仕様 OS を RMT Processor 向けに実装するにあたって，拡張した仕様について述べる．まずは，ITRON 仕様の機能分散マルチプロセッサ向け拡張である TOPPERS/FDMP カーネル [23]（以下，FDMP カーネル）の拡張仕様との比較を述べ，その後，拡張仕様の詳細について述べる．

4.1 マルチプロセッサ向け拡張仕様との比較

FDMP カーネルでは，μITRON4.0 仕様から以下の点を拡張している．

- カーネルオブジェクトのクラス分け
- ID 番号の割付け方法
- システムコール
- 静的 API
- システム状態

FDMP カーネルにおける拡張仕様の多くは，各プロセッサの保有するローカルメモリに格納されているオブジェクトに対し，他プロセッサから拡張前と同じ API でアクセスできるようにするためのものとなっている．たとえば，上記の「カーネルオブジェクトのクラス分け」では，同一のプロセッサに属するオブジェクトの集合をクラスと呼び識別しており，また「ID 番号の割付け方法」では，各オブジェクトに付けられる ID 番号の上位ビットでクラスを，下位ビットでクラス内におけるオブジェクトの識別番号を示すようにしている．これにより，拡張前と互換性のある API を，他プロセッサのオブジェクトを対象として使用できるようになっている．

一方，本拡張の対象となる RMT Processor では，カーネルオブジェクトは単一のメインメモリに格納され，どのスレッドからでも同様に参照が可能なため，これらのことを考慮する必要はない．本実装の拡張仕様を以下にあげる．

表 2 TCB 内の主要な変数
Table 2 Primary variables in TCB.

変数名	内容
tskid	タスク ID
task	タスク起動番地
tskstat	タスク状態
tskwait	待ち要因
tskpri	タスク優先度
que	所属キュー

表 3 TCB に追加した変数
Table 3 Variables added to TCB.

変数名	内容
period	タスク周期
wcet	最悪実行時間
deadline	デッドライン
tid	スレッド ID
rdqid	所属レディキュー

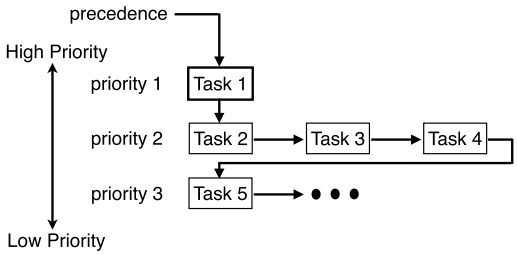


図 4 ITRON 仕様の優先度ベーススケジューリング方式
Fig. 4 Priority-based scheduling of ITRON specification.

- システム状態
- 周期タスク
- スケジューリング方式

FDMP カーネルと共通する拡張仕様としては，システム状態の拡張があげられる．ITRON 仕様では，割込みおよびタスクの切替えを禁止とする CPU ロック状態と，タスクの切替えのみを禁止とするディスパッチ禁止状態の 2 つを持つ．これらの状態は主に排他制御の際に使われるが，マルチスレッドプロセッサにおいては，システム全体で状態を管理するのはオーバヘッドが大きい．そのため，FDMP カーネルの拡張仕様では，これらのシステム状態をプロセッサごとに独立に管理している．また同時に，プロセッサ間の排他制御を実現するため，CPU ロック状態とディスパッチ禁止状態により排他制御を行っている部分を，スピンロックを用いた方法に置き換えている．本実装でも同様に，システム状態をスレッドごとに独立に管理し，スレッド間の排他制御はスピンロックで実現している．

周期タスクとスケジューリング方式については，以降の節で述べる．

4.2 周期タスク

ITRON 仕様ではタスクは周期の情報を持たず，実行が終わると休止状態に移行する．休止状態のタスクは API を用いて起動しない限り実行されないため，タスクを周期実行するには，周期ハンドラを用いてタスクを周期的に起動するなどの方法をとる必要がある．本実装では，タスクに周期の情報を持たせ，自動的に周期実行が行われるようにした．これを実現するため，タスクコントロールブロック（TCB）の拡張を行い，タスクに周期などの情報を追加した．TCB は，ITRON 仕様においてタスクに関する情報を格納する領域である．TCB 内の主要な変数を**表 2** に，本実装で追加した変数を**表 3** に示す．

追加した情報はタスクの周期実行のほかに，新たに実装したスケジューリング方式や，RMT Processor 固有命令を用いたコンテキストスイッチ部で使用する．タスクの周期と最悪実行時間は，タスクを生成する API である cre_tsk の引数で指定している．ただしプログラムの互換性を保つため，これらの値が指定されていなければ値 0 が入り，周期実行は行われない．また，残りのデッドライン，スレッド ID，所属レディキュー番号については，タスクを起動する API である act_tsk の実行時に設定する．

4.3 スケジューリング方式

ITRON 仕様では，タスクは与えられた優先度に基づいたプリエンプティブなスケジューリング方式によってスケジュールされる．ITRON 仕様 OS におけるスケジューリングの様子を**図 4** に示す．実行可能状態のタスクが複数ある場合には，その中で最も優先度の高いタスクが実行される．同じ優先度を持つタスク間では，First Come First Served（FCFS）方式によりスケジューリングを行い，先に実行可能状態になったタスクが高い優先順位を持つ．図 4 では最も優先度の高い Task 1 が実行状態となっており，Task 1 の実行が終了するか待ち状態に移行すると，Task 2 が実行状態に遷移する．その後 Task 1 が再び起動されると，Task 2 はプリエンプトされ，実行可能状態へと遷移する．

本実装では，従来の与えられた優先度に基づいたスケジューリング方式に加え，追加したタスクの周期とデッドラインの情報を利用して，Rate Monotonic（RM）[8] と Earliest Deadline First（EDF）[9] の 2 方式を実装した．どちらの方式でもタスク生成時に指定した優先度は使用せず，タスクの周期もしくはデッドラインの昇順にソートしてレディキューに挿入する．

5. 設計および実装

本章では，RMT Processor 向け HOS の実装技術について述べる．

5.1 スレッドの生成

本実装では，タスクとスレッドは 1 対 1 に対応させ，スレッドの生成はタスクを起動する act_tsk という API 内で行う．act_tsk は対象タスクを休止状態から実行可能状態へと移行させ，主に cre_tsk によって生成した直後のタスクに対して使用する．本実装では，act_tsk に以下の処理を追加している．

- スレッドの生成

- タスクのデッドラインを設定
- タスクのレディキューへの割当て

まず，act_tsk の内部で make_thread() 関数を呼ぶようにし，この関数の内容はプロセッサ依存部に記述している．RMT Processor の場合，make_thread() 関数内部では表 1 の mkth 命令を使用し，生成したスレッドの ID を TCB に格納する処理を行っている．コンテキストスイッチの際には，このスレッド ID を用いてタスクの切替えを行う．

本実装では，システムにおけるタスク数の上限をコンテキストキャッシュ数 32 個とアクティブスレッド数 8 個の合計 40 個までとしている．これは，タスクがコンテキストキャッシュから溢れてタスク切替えの速度が低下することで，リアルタイム性が低下するのを防ぐためである．後述するスケジューラ専用スレッドと割込み専用スレッド，6 つのサーバタスクの計 8 タスクをシステムで生成しているため，ユーザが生成し使用できるタスク数は 32 個となる．ユーザの生成するタスク数に 32 個という上限を持たせることは，本研究で提案した RT-OS のターゲットであるヒューマノイドロボットの小次郎 [10] における各制御ユニットで動いているタスクが，モータ制御や通信用タスクなどの計 6 個であることから，問題ないと判断した．

スレッドの生成を終えると，起動時刻とタスク周期を加算した値をタスクのデッドラインとして設定し，その後タスクをレディキューへと割り当てる．レディキューへの割当てについては次節で述べる．

5.2 タスクの割当て方式

本実装では，タスクの割当てにはパーティショニング方式を採用し，ハードウェアコンテキストごとにレディキューを生成している．タスクは act_tsk で起動された際に，いずれかのハードウェアコンテキストに割り当てられ，対応したレディキュー内でスケジューリングされる．その際，割り当てられたレディキューの番号を所属レディキュー番号として TCB に格納する．レディキューへの割当て方式は**表 4** の 4 つの方式を実装した．

これらの割当て方式の実装のため，レディキューに使用率の情報を追加した．レディキューの使用率は，対応したハードウェアコンテキストに割り当てられたタスクの使用率の総和で求められる．このとき，タスク τ_i の使用率 U_i は，τ_i の周期 T_i と最悪実行時間 C_i より，以下の式で表される．

$$U_i = \frac{C_i}{T_i} \tag{1}$$

レディキューの空きが十分かどうかの判定は，スケジューリング可能性判定により行われる．

表 4 レディキューへの割当て方式
Table 4 Task assignment policy for ready queue.

割当て方式	内容
First-Fit	最初に見つかった空きが十分なキューに割り当て
Worst-Fit	空きが最大（使用率が最小）なキューに割り当て
Best-Fit	空きが十分な中で使用率が最大のキューに割り当て
生成時指定	cre_tsk の引数で割り当てるキューを指定

5.3 コンテキストスイッチ

ITRON 仕様では，実行状態もしくは実行可能状態にあるタスクは，すべてレディキューにつながれている．レディキューは優先度ごとに分かれており，ディスパッチの際にはレディキューの先頭を優先度の高い方から順に調べていき，最初に見つかったタスクを実行タスクとする．このとき，実行タスクが現在実行状態にあるタスクと異なる場合には，コンテキストスイッチが発生する．

本実装では，従来のソフトウェアによるコンテキストスイッチの代わりに，RMT Processor のスレッド制御命令を使用し，コンテキストキャッシュを用いたコンテキストスイッチを実現している．HOS では各プロセッサでコンテキストスイッチ部が共通なため，コンテキストスイッチ部で context_switch() 関数を呼ぶように変更し，この関数内部はプロセッサ依存部に記述することでポータビリティの低下を防いでいる．RMT Processor においては，context_switch() 関数内部ではスレッド制御命令を用いてコンテキストスイッチを行う．**表 5** に使用するスレッド制御命令の例を示す．たとえば実行中のタスクと次に実行するタスクが異なる場合，swapth 命令を使用して実行するタスクを入れ替えることで，コンテキストスイッチを実現している．

5.4 排他制御

ITRON 仕様 OS では，API によりタスクやセマフォ，メールボックスなどのオブジェクトを生成することができる．API を用いてこれらのオブジェクトに対してアクセスを行うことで，様々な処理を実現する．このとき，各オブジェクトは全スレッドで共有されるため，オブジェクトにアクセスする際にはスレッド間の排他制御が必要となる．本実装では，排他制御にはスピンロックを使用し，各 API のオブジェクトにアクセスする前後でスピンロックの取得と解放を行っている．

また，ユーザがタスクと割込みハンドラ間の排他制御

表 5 コンテキストスイッチに使用するスレッド制御命令
Table 5 Thread-control instructions used for context switch.

実行中タスク	次タスク	使用する命令と処理内容
なし	なし	何もしない
なし	あり	rstrth 命令により次タスクを復帰
あり	なし	bkupth 命令により実行中タスクを退避
あり	あり	swapth 命令により実行タスクを入れ替え

表 6 タスクと割込みハンドラ間の排他制御を行う API
Table 6 API of exclusion between tasks and interrupt handlers.

API 名	返戻値	引数	内容
cre_spn	ER	ID spnid	ロックオブジェクトの生成
loc_spn	ER	ID spnid	スピンロックの獲得
iloc_spn	ER	ID spnid	スピンロックの獲得†
unl_spn	ER	ID spnid	スピンロックの解放
iunl_spn	ER	ID spnid	スピンロックの解放†

† 非タスクコンテキスト用

表 7 ロックの取得順序による API の分類
Table 7 API classificaiton by order of lock acquire.

分類名	ロックの取得順序	API 数
クラス A	ロックを取得しない	30 種
クラス B	オブジェクトロックのみ	18 種
クラス C	タスクロックのみ	13 種
クラス D	オブジェクト → キュー → タスクロック	42 種
クラス E	タスク → キューロック	9 種

を行うための機能として，スピンロックを獲得・解放する API を実装した．実装した API を**表 6** に示す．

5.5 デッドロックの回避

ITRON 仕様の API には，複数のオブジェクトにアクセスするものが存在する．そのような API では複数のロックを取得することになるが，その際に各スレッドの実行する API の組合せによっては，デッドロックが発生する可能性がある．

デッドロックを回避する手法は FDMP カーネル [23] の実装を参考にした．FDMP カーネルでは，タスク管理に関わる構造体のロックであるタスクロックと，セマフォやメールボックスなどの同期・通信オブジェクトに関わる構造体のロックであるオブジェクトロックの 2 つを設け，ロックの取得順序をタスクロック → オブジェクトロックの順と定めることでデッドロックを回避している．この順序でロックを取得できない API に関しては，タスクロックを取得した後，いったんタスクロックを解放して，再度オブジェクトロック → タスクロックの順にロックを取得する．

本実装では，タスクロック，オブジェクトロック，そしてキューロックの 3 種類のロックを設ける．キューロックとは，レディキューやオブジェクトに付随する待ち行列（セマフォの獲得待ち行列など）といったキューに関するロックである．キューロックが必要となるのは，HOS にはオブジェクトにはアクセスせず，対象となるタスクが現在所属している待ち行列に直接アクセスする API が存在するためである．

まず，ロックを取得する順番によって，HOS に実装されている ITRON 仕様の全 API を**表 7** のように分類する．

```
/* タスク ID からタスクオブジェクトを取得 */
task = get_task_object(tskid);
/* タスクロックの取得 */
acquire_lock(task->lock);

/* タスクの所属キューを取得 */
que = task->que

/* デッドロック回避のため，ロックを再取得する */
/* タスクの状態を TERTSK 実行中状態に変更 */
task->tskstat = STAT_EXE_TERTSK;
/* タスクロックをいったん解放 */
release_lock(task->lock);
/* 再度キュー，タスクの順にロックを再取得 */
acquire_lock(que->lock);
acquire_lock(task->lock);

/* タスクの状態が TERTSK 実行中状態であるか確認 */
if (task->tskstat == STAT_EXE_TERTSK) {
     /* Critical Section(ter_tsk の処理) */
}
release_lock(task->lock); /* タスクロックを解放 */
release_lock(que->lock);  /* キューロックを解放 */
```

図 5 デッドロック回避のコード例（ter_tsk）
Fig. 5 Example code for deadlock avoidance (ter_tsk).

表 7 の分類において，クラス D とクラス E の API 間でデッドロックが発生する可能性がある．たとえば，スレッド 1 がクラス D の API を発行し，セマフォ A のオブジェクトロックとセマフォ A に付随する待ち行列のキューロックを取得した段階で，スレッド 2 がクラス E の API を発行してタスク B のタスクロックを取得したとする．このとき，スレッド 1 がタスク B のロックを取得しようとし，スレッド 2 がセマフォ A の待ち行列のキューロックを取得しようとすると，デッドロックが発生する．

ここで，クラス D の API が取得するキューロックは，オブジェクトに付随する待ち行列のみが対象である．また，クラス E の API の中で，オブジェクトに付随する待ち行列のキューロックを取得する可能性のある API は，対象タスクの待ち状態を強制解除する rel_wai と，対象タスクを強制終了する ter_tsk の 2 種類のみである．つまり，クラス D の API とデッドロックを起こす可能性のある API はこの 2 種類だけであることから，本実装ではロックの取得順序をオブジェクトロック → キューロック → タスクロックの順に定め，これに反する rel_wai と ter_tsk ではロックの再取得を行い，デッドロックを回避する．**図 5** に ter_tsk におけるデッドロック回避のコードを示す．

デッドロック回避の必要な API では，タスクロックを取得した後にタスクの所属キューが分かった時点でいったんタスクロックを解放し，その後キューロック → タスクロックの順にロックを再取得する．このとき，ロックを解放し

てから再取得を行う間にタスクの状態が変わってしまう可能性がある．たとえば，タスクがセマフォ獲得待ち状態からメールボックス受信待ち状態に変わっていた場合には，ロックすべき待ち行列が変化してしまうため，再度ロックを取得し直さなければならない．しかし，再取得する間に再度タスクの状態が変化してしまう可能性がある．この問題を回避するため，ロックの再取得を行う前にタスクの状態を TERTSK 実行中状態もしくは RELWAI 実行中状態に変更し，デッドロック回避の必要な API を実行中であることを示す．これにともない，クラス C〜E の API では，タスクロック取得後にタスクの状態が TERTSK 実行中状態もしくは RELWAI 実行中状態であった場合，代わりに ter_tsk もしくは rel_wai の処理を行い，タスクの状態を変更する．一方，ter_tsk と rel_wai ではロックの再取得後にタスクの状態が TERTSK 実行中状態もしくは RELWAI 実行中状態のままであることを確認し，処理を続ける．もしタスク状態が変更されていれば，何も行わずに API の実行を終了する．

5.6 割込み専用スレッド

FDMP カーネルでは，ロックの取得時間を短くするため，ロックの取得中は割込みを禁止としている．これによって割込み応答時間が長くなるのを防ぐため，ロックの取得を試みるたびに割込み要求を確認している．FDMP カーネルにおける排他制御の様子を**図 6** に示す．

しかし，この手法には，割込みによってロックを取得する API の実行時間が伸びてしまうという問題がある．この問題を解消するため，本実装では 1 スレッドを割込み専用スレッドとして使用する．その際，多重割込みの応答性を改善するため，非周期サーバを実装し，割込みの受付処理と実際の割込み処置を分離している．具体的には，割込み専用スレッドは割込みを受け付けるとサーバタスクに対して割込みの内容を伝え，サーバタスクを起動する処理を行う．

```
retry:
  /* ロックの取得 */
  disable_interrupt();
  while(!test_and_set(lock)) {
    /* ロックの取得ができずかつ割込み発生時には， */
    /* 割込みを受け付け，その後 retry から再開する  */
    if (interrupt_request()) {
      enable_interrupt();
      goto retry;
    }
  }

  /* Critical Section */

  /* ロックの解放 */
  release_lock(lock);
  enable_interrupt();
```

図 6 FDMP カーネルにおける排他制御
Fig. 6 Exclusion technique in FDMP Kernel.

ここで，ITRON 仕様では，タイマ割込みの割込み処理でタスクのディスパッチが行われることがある．特に本実装ではタスクを周期実行するため，周期ごとにタスクのリリースが行われる．これをサーバタスクに実行させようとすると，サーバタスクの割当て処理などのオーバヘッドにより，タスクのリリースが遅れてデッドラインミスを招く可能性がある．これを防ぐため，さらにもう 1 スレッドをタイマ割込みにともなうスケジューラ専用スレッドとして割り当てている．よって，各スレッド TH0〜TH7 が実行する内容は以下のようになる．

- TH0：スケジューラ専用スレッド
- TH1：割込み応答専用スレッド
- TH2〜7：ユーザタスクおよびサーバタスク

上記のとおり，サーバタスクは他のユーザタスクと同じハードウェアコンテキストに割り当てられ，他のタスクと同じようにスケジューリングされる．システム起動時にサーバタスクは 6 個生成され，各ハードウェアコンテキストに 1 つずつ割り当てられる．割込み発生時に，割込み処理がどのサーバタスクが割り当てられるかは，実行されていないサーバタスクが優先的に割り当てられ，同条件下では割り当てられたハードウェアコンテキストに対応するレディキューの使用率が一番低いものが選択される．

非周期サーバは，サーバタスクの割当てにかかるオーバヘッドの少ない，Deferrable Server（DS）[12] と Constant Bandwidth Server（CBS）[1], [2] を実装した．前者はスケジューリング方式を従来の優先度ベーススケジューリングおよび RM とした場合，後者は EDF にした場合に使用する．

6. 評価

本章では，RMT Processor 向けに設計・実装を行った ITRON 仕様 OS の評価を行う．まずは，拡張にともなうコードサイズと実行時間のオーバヘッドを測定する．次に，割込み専用スレッドの評価として，最悪割込み応答時間と，割込みを一定間隔で発生させたときの API の実行時間を測定し，割込み応答時間が一定に定まることと，FDMP カーネルの手法と比べた優位性を示す．加えて，多重割込みのオーバヘッドを測定する．最後に，シミュレーションと実機評価によりスケジュール成功率を測定し，ハードウェアによるコンテキストスイッチによってスケジュール成功率が向上することを示す．

6.1 評価環境

評価に使用した RMT Processor の構成は**表 8** のとおり

表 8 RMT Processor の構成
Table 8 Outline of the RMT Processor.

Clock Frequency	31.25 MHz
Fetch Width	8
Integer register	32-bit × 32-entry × 8-set
Integer renaming register	32-bit × 64-entry
FP register	64-bit × 8-entry × 8-set
FP renaming register	64-bit × 64-entry
ALU	4 + 1（Divider）
FPU	2 + 1（Divider）
64-bit ALU	1
FP Vector Units	1（4FPU × 2 line）
Branch Unit	2
Memory Access Unit	1

表 9 カーネルのコードサイズ
Table 9 Code size of kernel.

カーネル	text	data	bss
ベース	48,762 Byte	0 Byte	16 Byte
拡張後	72,916 Byte	20 Byte	52 Byte

表 10 測定を行う API の分類
Table 10 Classification of API to measure.

API 名	API の処理内容	追加した主要な処理
act_tsk	タスクの起動	スレッド生成，タスク割当て
pol_sem	セマフォの獲得	オブジェクトロックの取得
wup_tsk	起床待ち状態の解除	タスクロックの取得
sig_sem	セマフォの返却	3 種類のロックの取得
ter_tsk	タスクの強制終了	デッドロックの回避

表 11 API の text サイズ
Table 11 Text size of API.

API 名	ベース	拡張後	取得ロック数	相対 text サイズ
act_tsk	544 Byte	1,176 Byte	1	2.16
pol_sem	280 Byte	336 Byte	1	1.20
wup_tsk	588 Byte	680 Byte	1	1.16
sig_sem	436 Byte	644 Byte	3	1.48
ter_tsk	472 Byte	792 Byte	2	1.68

である．また，実行するプログラムの生成には，gcc のクロス開発環境を使用した．gcc のバージョンは 3.4.3 である．

6.2 コードサイズ

ベースとなる HOS カーネルと，実装した RMT Processor 向け ITRON 仕様 OS のコードサイズを**表 9** に示す．text のサイズは HOS カーネルに比べ約 1.5 倍となっている．これはほぼすべての API に排他制御のためのロックの獲得と解放のルーチンが挿入されたためと，複数のスケジューリング方式・タスク割当て方式を実装したためである．

次に，API の text サイズを比較する．比較を行う API については，**表 10** のように選択した．比較結果を**表 11** に示す．act_tsk はマルチスレッド拡張にあたって多くの処理を追加したため，text サイズが大きく増加している．その他の排他制御の処理が追加された API については，取得するロック数が多いほど text サイズの増加率が大きくなっていることが分かる．ただし，ter_tsk はデッドロック回避の処理を行っているため，取得ロック数に比べてサイズの増加率が大きくなっている．

表 12 API の実行時間（コンテキストスイッチなし）
Table 12 Execution time of API without context switch.

API 名	拡張前	拡張後	取得ロック数	相対実行時間
act_tsk	15 μsec	26 μsec	1	1.73
pol_sem	7 μsec	8 μsec	1	1.14
wup_tsk	15 μsec	17 μsec	1	1.13
sig_sem	15 μsec	20 μsec	3	1.33
ter_tsk	14 μsec	19 μsec	2	1.36

6.3 API のオーバヘッド

本節では，RMT Processor 向けに HOS を拡張する際に API に追加した，排他制御やスレッドの生成といった処理のオーバヘッドを測定する．比較対象として，各 API とコンテキストスイッチの処理をベースとなる HOS から変更せずに，RMT Processor によってシングルスレッドで実行を行った場合の実行時間を測定し，拡張後の API の実行時間と比較を行う．実行時間の測定を行う API については，コードサイズの評価と同様に表 10 から選択した．

まず，コンテキストスイッチが発生しない条件下において実行時間を測定したものを**表 12** に示す．スレッドの生成やタスクの割当てなど，拡張にあたり多くの処理が追加された act_tsk は，text サイズと同様に実行時間のオーバヘッドも大きくなっている．それ以外の API については，取得するロック数が多いほど実行時間のオーバヘッドが大きくなっているが，ter_tsk はデッドロック回避の処理を行っているため，取得ロック数に比べてオーバヘッドが大きい．

表 12 にある API の中で，wup_tsk と sig_sem は，これらの API を発行することでコンテキストスイッチが発生することがある．たとえば，wup_tsk によって起床待ち状態から実行可能状態となったタスクが，所属するレディキュー内で最高優先度であればコンテキストスイッチが発生する．このようなコンテキストスイッチが発生する条件下において，wup_tsk と sig_sem を発行した際の実行時間を**表 13** に示す．ここで測定した実行時間とは，API を発行してから，コンテキストスイッチにより実行状態となったタスクが実行を再開するまでの時間である．

表 13 を見ると，拡張前より拡張後の方が実行時間が短くなっていることが分かる．これはコンテキストスイッチに RMT Processor のスレッド制御命令を使用し，コンテ

表 13 API の実行時間（コンテキストスイッチあり）
Table 13 Execution time of API with context switch.

API 名	拡張前	拡張後	取得ロック数	相対実行時間
wup_tsk	28 μsec	20 μsec	1	0.71
sig_sem	29 μsec	23 μsec	3	0.80

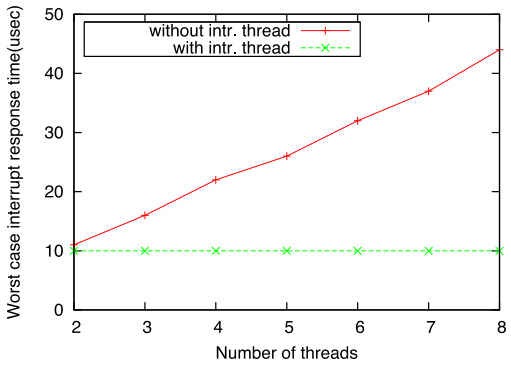


図 7 最悪割込み応答時間
Fig. 7 Worst-case interrupt response time.

キストキャッシュを活用することよってタスクの入替え時間を大幅に短縮したためである．

一方，コンテキストスイッチが発生しない場合には API の実行時間が伸びてしまうが，本 OS のターゲットである小次郎における要求仕様では，各ジョイントコントローラの制御周期が 10 μsec 以下であることが求められている [13]．ロックの取得にかかるサイクル数は約 30 cycle であり，ロックを追加したことによるオーバヘッドを考慮しても，要求仕様を満たすことは十分可能である．

6.4 割込み専用スレッド

本節では，割込み専用スレッドの評価として，最悪割込み応答時間，割込み発生時の API の実行時間，多重割込みのオーバヘッドをそれぞれ測定する．

6.4.1 最悪割込み応答時間

本実装では，1 スレッドを割込み専用スレッドとすることで，排他制御と割込み応答性を両立させている．割込み専用スレッドを使用しない場合は，ロックの取得時間を短くするため，ロックの取得中は割込みを禁止にすることが求められる．本項では，割込み専用スレッドを使用した場合と，割込み専用スレッドを使用せずにロック取得中の割込みを禁止にした場合における最悪割込み応答時間を測定し，比較を行う．測定結果を**図 7** に示す．

最悪割込み応答時間は，複数スレッドに同一のロックを取得する API を実行させ，一定時間間隔で発生するタイマ割込みに対する応答時間の最悪値を測定することで得られたもので，測定値は割込みを 1 万回発生させた結果である．図 7 を見ると，割込み専用スレッドを用いない場合は，同時実行スレッド数が増えるにつれて最悪割込み応答時間が

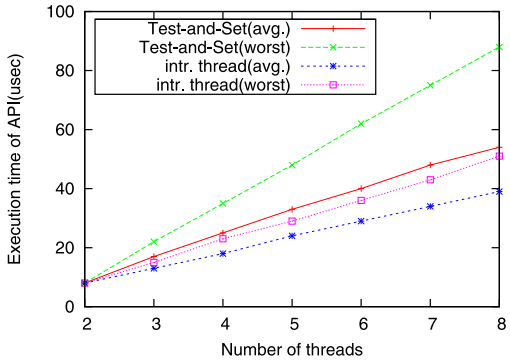


図 8 割込み発生時の API の実行時間
Fig. 8 Execution time of API when interrupts occurred.

大きくなってしまっている．一方，割込み専用スレッドを使用した場合には，最悪割込み時間は 10 μsec 付近で一定となっており，最悪割込み応答時間が同時実行スレッド数に依存していないことが分かる．

6.4.2 割込み発生時の API の実行時間

5.6 節で述べた FDMP カーネルの手法との比較として，同一のロックを取得する API を各スレッドに実行させ，タイマ割込みを 1 msec ごとに発生させたときの API の実行時間を測定した．その結果を，**図 8** に示す．Test-and-Set が FDMP カーネルの手法，intr. thread が割込み専用スレッドを用いた場合を表す．FDMP カーネルの手法では，ロックの取得待ち時間に加えて，取得待ち中に発生した割込み処理時間がかかるため，実行時間の増え幅が割込み専用スレッドに比べて大きくなっている．特に最悪値ではその傾向が顕著で，これはロックの取得中に割込みを受け付ける回数には上限がないことに起因していると考えられる．一方，割込み専用スレッドを用いた場合では，同時実行スレッド数を増やしてもロックの取得待ち時間が増大するのみであるため，実行時間の平均値・最悪値ともに FDMP カーネルの手法を用いた場合を下回っている．

6.4.3 多重割込みのオーバヘッド

本実装では多重割込みの応答性を向上させるため，非周期サーバを用いて割込み応答処理と割込み処理を分離している．多重割込みの応答性は各割込み処理の長さに依存するため，評価では割込み処理をサーバタスクに割り当てる処理にかかる時間を測定した．具体的には，割込みが発生してから，その割込み処理を行うサーバタスクの割当てが終了するまでの時間を，間に多重割込み 0〜7 回を発生させた場合についてそれぞれ 1 万回測定を行い，その最悪値を計測した．測定結果を**図 9** に示す．

図 9 中の CBS は Constant Width Server を，DS は Deferrable Server を用いた場合を表している．また，Common は CBS と DS に共通する処理にかかった処理時間を示している．図 9 を見ると，多重割込み数が増えるごとにオーバヘッドは増大している．また，Common は単一の割込み処理にかかるオーバヘッドと同等であるため，非周期サー

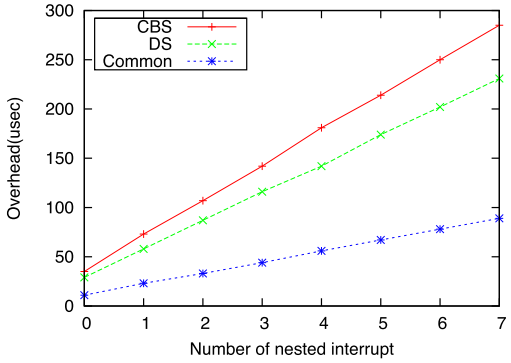


図 9 多重割込みのオーバヘッド
Fig. 9 Overhead of nested interrupts.

バを用いると，単一の割込み処理に比べてオーバヘッドが増大していることが分かる．しかし，非周期サーバを用いた場合では，割込み処理を TH2～TH7 に割り当てることで，単一の専用スレッドを用いた場合に比べ，結果的に応答時間が小さくなると考えられる．

6.5 スケジュール成功率

本節では，シミュレーションと実機のそれぞれについて，拡張した ITRON 仕様 OS の，多様なタスクセットに対するスケジュール成功率（Schedule Success Ratio，SSR）を測定する．SSR は以下の式で表される．

$$\mathrm{SSR} = \frac{\#\ \text{of successfully scheduled task sets}}{\#\ \text{of scheduled task sets}} \qquad (2)$$

測定は，コンテキストスイッチをソフトウェアで行う場合と，ハードウェアで行う場合の両方で行い，コンテキストキャッシュを用いることによりスケジュール成功率が向上していることを示す．

6.5.1 シミュレーションによる評価

シミュレーションによる評価では，スケジューリング方式には RM および EDF を用い，タスク割当て方式には First-Fit を使用する．具体的には，ハードウェアコンテキストに割り当てられたタスクの使用率（式 (1)）の総和が，RM を用いた場合は $n(\sqrt[n]{2}-1)$ 以下のとき（n はハードウェアコンテキストに割り当てられたタスク数），EDF を用いた場合は 100%以下のときに割当て可能と判定する．また，各々の表記は RM-FF，EDF-FF とする．

今回のシミュレーションでは，各ハードウェアコンテキストをプロセッサととらえている．このとき，プロセッサ数を M，タスクセット中のタスク数を n とすると，システムの合計使用率 U はタスク τ_i の周期 T_i と最悪実行時間 C_i を用いて以下の式で表される．

$$U = \frac{1}{M}\sum_{i}^{n}\frac{C_i}{T_i} \qquad (3)$$

シミュレーションでは，システム使用率 $U = 30\%$から 100%までのスケジュール可能性を評価するため，システム使用率 U に対して 1,000 個のタスクセットを生成した．また，各タスクのパラメータは，実機で測定したモータの PID 制御プログラムの周期と実行時間から決定した．各タスクの周期は 1～5 ms を想定して [1,000, 5,000] の範囲で，最悪実行時間は 10～50 μsec を想定して [10, 50] の範囲で無作為に決定する．タスクは，タスクセット中のタスクの使用率の合計が規定のシステム使用率 U に達するまで生成される．使用率の計算ののち，各タスクの最悪実行時間に，実機で測定したソフトウェアもしくはハードウェアによるコンテキストスイッチのオーバヘッドを加えている．オーバヘッドの値は，ソフトウェアの場合が 12 μsec，ハードウェアの場合が 1 μsec である．同時実行スレッド数は 8 であるため，ハードウェア資源の競合を考慮しなければプロセッサ数 M は 8 となるが，RMT Processor の持つ ALU が 4 つであることをふまえ（表 8），M を 4 とした場合の測定も行う．

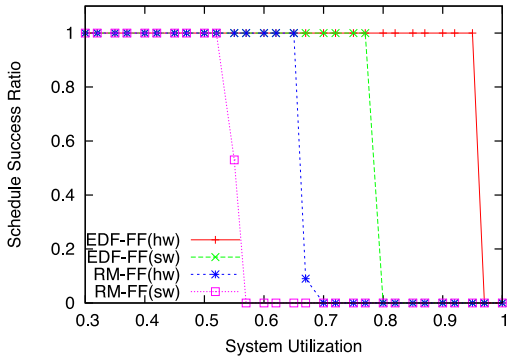


図 10 スケジュール成功率（$M = 4$）
Fig. 10 Schedule Success Ratio ($M = 4$).

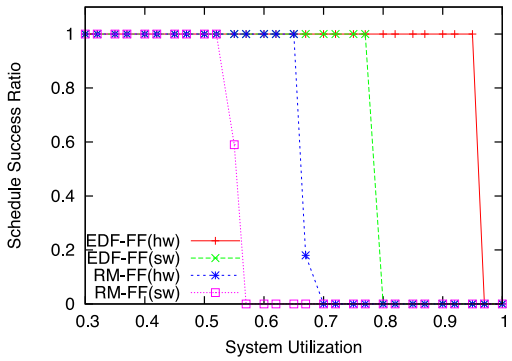


図 11 スケジュール成功率（$M = 8$）
Fig. 11 Schedule Success Ratio ($M = 8$).

測定結果を**図 10**，**図 11** に示す．図中の表記は，たとえばスケジューリング方式が RM-FF でハードウェアのオーバヘッドを加えた場合は RM-FF（hw），ソフトウェアのオーバヘッドを加えた場合は RM-FF（sw）としている．図 10 と図 11 を見比べてみると，プロセッサ数を変えても各スケジューリング方式のスケジュール成功率はほぼ同じ結果になっている．また，ソフトウェアによるコンテキストスイッチを使用した場合に比べて，コンテキストキャッ

表 14 デッドラインミス率
Table 14 Deadline miss rate.

	$U = 0.6$		$U = 0.7$		$U = 0.8$		$U = 0.9$		$U = 1.0$	
スレッド	SW	HW	SW	HW	SW	HW	SW	HW	SW	HW
T_3	0%	0%	0%	0%	95.2%	0%	98.1%	94.0%	99.3%	99.0%
T_4	91.3%	0%	99.0%	94.9%	99.4%	98.3%	99.4%	99.2%	99.6%	99.4%

シュを用いた場合の方がスケジュール成功率は向上していることが分かる．たとえば，図 10 の RM-FF（sw）はシステム使用率 U が 56%付近になるとスケジュール成功率が 0 になっているのに対し，RM-FF（hw）では U が 70%付近になるまでスケジュール成功率が 0 になっていない．これらの結果から，コンテキストキャッシュを用いることで，タスクのスケジュール成功率を向上させることができていることが分かる．

6.5.2 実機による評価

実機による評価では，行列演算を行うタスクをスレッド T_1 から T_4 にそれぞれ 8 個ずつ割り当て，4 スレッドで同時実行した際のデッドラインミス率を測定した．スケジューリング方式は EDF，タスク割当てには First-Fit を使用した．また，割り当てたスレッドごとにタスクの優先度を変えており，優先度は $T_1 > T_2 > T_3 > T_4$ となっている．タスクの最悪実行時間は事前測定の結果をふまえて 150 μsec とし，周期を [1,200 μsec, 2,000 μsec] の範囲で変更することで，CPU 使用率を変えて測定を行った．測定の結果を**表 14** に示す．

SW がソフトウェアによるコンテキストスイッチを使用した場合，HW がコンテキストキャッシュを用いた場合を表している．T_1 および T_2 ではデッドラインミスが発生しなかったため，表からは除外した．表 14 を見ると，どの条件においてもデッドラインミス率は $SW > HW$ となっており，6.5.1 項のシミュレーション結果と同様に，ハードウェアによるコンテキストスイッチによってデッドラインミス率が低下していることが分かる．

7. まとめ

RMT Processor は優先度を導入した優先度付き SMT アーキテクチャを採用しており，また，ハードウェアレベルでコンテキストスイッチを行うことでタスク切替えにかかるオーバヘッドを大幅に削減できるなど，組込みリアルタイムシステムに適した機能を有している．とりわけ，コンテキストスイッチのオーバヘッドの増大は，タスクごとに時間制約のあるリアルタイムシステムにおいてスケジュール成功率を低下させる要因の 1 つとなる．

これまでに，マルチプロセッサ向け RT-OS や RT-OS のハードウェア化について報告されているが，リアルタイムシステムで必須となる優先度制御を導入した優先度付き SMT アーキテクチャにおける RT-OS については報告されていない．優先度付き SMT アーキテクチャでは，最大 8 つのハードウェアコンテキストを用いた優先度付き SMT 実行によって，EDF や RM を用いた複数スレッドのリアルタイム実行が可能であり，このためには SMT アーキテクチャ固有のスレッド制御命令を用いた RT-OS への拡張と実装およびその評価が必要である．

本論文では，ITRON 仕様の RT-OS について，RMT Processor 向けにマルチスレッド拡張を行った．タスクコントロールブロックを拡張して周期や最悪実行時間などの情報を追加し，それらの情報を用いて RM や EDF といったスケジューリング方式と，First-Fit や Worst-Fit，Best-Fit といったレディキューへの割当て方式を実装した．コンテキストスイッチにともなうオーバヘッドの大幅な削減，および，スケジュール成功率の改善のために，スレッドの生成やディスパッチには RMT Processor のスレッド制御命令を使用した．排他制御は文献 [23] を参考にしてタスクロック・オブジェクトロック・キューロックの 3 種のロックを用いて実装した．その際，ロックの取得順序をオブジェクトロック → キューロック → タスクロックという順に定め，デッドロックを回避した．また，割込み専用スレッドを設けて非周期サーバを実装することで，排他制御や多重割込みにともなう割込み応答時間の増大を防いだ．

評価ではマルチスレッド拡張前と拡張後におけるカーネルのコードサイズと API の実行時間を測定した．その結果，マルチスレッド拡張によってカーネルコードサイズや一部の API の実行時間は増加したものの，ディスパッチをともなう API に関しては，コンテキストキャッシュを用いたスレッド制御命令によって逆に実行時間が短縮された．また，割込み専用スレッドによって，同時実行スレッド数が増えても最悪割込み応答時間を一定にできていることと，FDMP カーネルの手法と比較して，割込みによる API の実行時間の増大が抑制されていることを示した．さらに，RM や EDF を用いたタスクシミュレーションと実機による評価の結果，スレッド制御命令を用いた高速コンテキストスイッチによりタスクのスケジュール成功率が向上することを示した．

謝辞 本研究は科学技術振興機構 CREST の支援によるものであることを記し，謝意を表す．また，本研究の一部は文部科学省グローバル COE プログラム「環境共生・安全システムデザインの先導拠点」によるものであることを記し，謝意を表す．

参考文献

[1] Abeni, L. and Buttazzo, G.: Integrating multimedia applications in hard real-time systems, *Proc. 19th IEEE Real-Time Systems Symposium*, pp.4–13 (online), DOI: 10.1109/REAL.1998.739726 (1998).

[2] Abeni, L. and Buttazzo, G.: Resource Reservation in Dynamic Real-Time Systems, *Real-Time Syst.*, Vol.27, No.2, pp.123–167 (2004).

[3] AUTOSAR: *Specification of Operating System V4.0.0* (2009).

[4] Calandrino, J., Leontyev, H., Block, A., Devi, U. and Anderson, J.: LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers, *Proc. 27th IEEE Real-Time Systems Symposium*, pp.111–126, Washington, DC, USA, IEEE Computer Society (2006).

[5] Eggers, S., Emer, J., Levy, H., Lo, J., Stamm, R. and Tullsen, D.: Simultaneous Multithreading: A Platform for Next-Generation Processors, *IEEE Micro*, Vol.17, pp.12–19 (1997).

[6] HOS プロジェクト：Hyper Operating System, SourceForge.JP (online), 入手先 ⟨http://sourceforge.jp/projects/hos/⟩（参照 2011-04-13）.

[7] Islam, R., Sabbavarapu, A., Patel, R., Kumar, M., Nguyen, J., Patel, B. and Kontu, A.: Next Generation Intel© ATOM^{TM} processor based ultra low power SoC for handheld applications, *Solid State Circuits Conference (A-SSCC), 2010 IEEE Asian*, pp.1–4 (2010).

[8] Lehoczky, J., Sha, L. and Ding, Y.: The rate monotonic scheduling algorithm: Exact characterization and average case behavior, *Proc. Real Time Systems Symposium*, pp.166–171 (1989).

[9] Liu, C. and Layland, J.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *J. ACM*, Vol.20, pp.46–61 (1973).

[10] Mizuuchi, I., Nakanishi, Y., Sodeyama, Y., Namiki, Y., Nishino, T., Muramatsu, N., Urata, J., Hongo, K., Yoshikai, T. and Inaba, M.: An advanced musculoskeletal humanoid Kojiro, *7th IEEE-RAS International Conference on Humanoid Robots*, pp.294 –299 (2007).

[11] OSEK/VDX: *OSEK/VDX Operating System Specification Version 2.2.3* (2005).

[12] Strosnider, J.K., Lehoczky, J.P. and Sha, L.: The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments, *IEEE Trans. Comput.*, Vol.44, No.1, pp.73–91 (1995).

[13] Suito, K., Ueda, R., Fujii, K., Kogo, T., Matsutani, H. and Yamasaki, N.: The Dependable Responsive Multithreaded Processor for Distributed Real-Time Systems, *IEEE Micro*, Vol.32, No.6, pp.52–61 (2012).

[14] Tullsen, D., Eggers, S. and Levy, H.: Simulataneous Multithreading: Maximizing On-Chip Parallelism, *Proc. 22nd Annual International Symposium on Computer Architecture*, Vol.23, pp.392–403 (1995).

[15] Wendel, D., Kalla, R., Cargoni, R., Clables, J., Friedrich, J., Frech, R., Kahle, J., Sinharoy, B., Starke, W., Taylor, S., Weitzel, S., Chu, S., Islam, S. and Zyuban, V.: The implementation of POWER7^{TM}: A highly parallel and scalable multi-core high-end server processor, *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp.102–103 (2010).

[16] Yamasaki, N.: Responsive multithreaded processor for distributed real-time systems, *Journal of Robotics and Mechatronics*, Vol.17, No.2, pp.130–141 (2005).

[17] Yodaiken, V.: The RTLinux Manifesto (1999).

[18] T-Engine フォーラム：2010 年度組込みシステムにおけるリアルタイム OS の利用動向に関するアンケート調査報告書，T-Engine（オンライン），入手先 ⟨http://www.t-engine.org/ja/wp-content/themes/wp.vicuna/pdf/ja/TEF070-W001-110405.pdf⟩（参照 2012-02-25）.

[19] 鈴木貴久，上方輝彦：マルチプロセッサ向け μITRON OS の開発，情報処理学会研究報告．計算機アーキテクチャ研究会報告，Vol.2005, No.120, pp.57–61 (2005).

[20] 坂村　健（監修），高田広章（編）：μITRON4.0 仕様 Ver4.02.00，トロン協会 (2004).

[21] 丸山修孝，石原　亨，安浦寛人：RC-011 仮想キューによる高性能ハードウエア RTOS の実現（C 分野：ハードウェア・アーキテクチャ，査読付き論文），情報科学技術フォーラム講演論文集，Vol.9, No.1, pp.115–120 (2010).

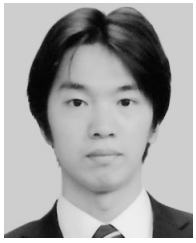
[22] 丸山修孝，石原　亨，安浦寛人：RTOS のハードウェア化によるソフトウェアベース TCP/IP 処理の高速化と低消費電力化（回路理論，回路解析），電子情報通信学会論文誌 A，基礎・境界，Vol.94, No.9, pp.692–701 (2011).

[23] 本田晋也，高田広章：ITRON 仕様 OS の機能分散マルチプロセッサ拡張，電子情報通信学会論文誌 D，情報・システム，Vol.91, No.4, pp.934–944 (2008).

[24] 石綿陽一，松井俊浩，國吉　康：高度な実時間処理機能を持つ Linux の開発，第 16 回日本ロボット学会学術講演会予稿集，pp.355–356 (1998).

上田 陸平　（学生会員）

2012 年慶應義塾大学理工学部情報工学科卒業．現在，同大学大学院修士課程に在籍．リアルタイム OS の研究に従事．



藤井 啓　（学生会員）

2010 年慶應義塾大学理工学部情報工学科卒業．2012 年同大学大学院理工学研究科開放環境科学専攻修士課程修了．リアルタイムシステムにおける省電力化技術等の研究に従事．

千代 浩之　（正会員）

2008 年慶應義塾大学理工学部情報工学科卒業．2012 年同大学大学院理工学研究科開放環境科学専攻博士課程修了．博士（工学）．現在，同大学訪問研究員の傍ら，2012 年 9 月よりノースカロライナ大学チャペルヒル校客員研究員を兼務．リアルタイムシステム，オペレーティングシステム，分散ミドルウェア等の研究に従事．

松谷 宏紀 （正会員）

2004 年慶應義塾大学環境情報学部卒業．2008 年同大学大学院理工学研究科開放環境科学専攻博士課程修了．博士（工学）．現在，慶應義塾大学理工学部情報工学科専任講師．2009 年度より 2010 年度まで日本学術振興会特別研究員 SPD．計算機アーキテクチャ，オンチップネットワークの研究に従事．

山崎 信行 （正会員）

1991 年慶應義塾大学理工学部物理学科卒業．1996 年同大学大学院理工学研究科計算機科学専攻博士課程修了．博士（工学）．同年電子技術総合研究所入所．1998 年 10 月慶應義塾大学理工学部情報工学科助手．同専任講師を経て，2004 年 4 月同助教授（現，准教授）．現在，同教授．リアルタイムシステム，プロセッサアーキテクチャ，並列分散処理，システム LSI，ロボティクス等の研究に従事．日本ロボット学会，IEEE 各会員．