情報処理学会論文誌 Vol.58 No.2 531–543 (Feb. 2017)

RMT Processorの割込み起床機構を用いた低遅延リアルタイム実行

大沢 幸平[1,a)] 溝谷 圭悟[2,†1] 千代 浩之[3] 山﨑 信行[2]

受付日 2016年5月26日, 採録日 2016年11月1日

概要：組込みリアルタイムシステムにおいて，ハードリアルタイムタスクは時間制約を満たせなければシステムに致命的な障害を与える．ハードリアルタイムタスクの中でも，特にモータ制御のようなタスクは他のタスクと比較して高い優先度を持ち，短い周期かつ小さいジッタで実行することが求められる．優先度付き Simultaneous Multithreading アーキテクチャである Responsive Multithreaded Processor（RMT Processor）は，8 個の論理コアを持ち，非常に細かい時間粒度でリアルタイム処理を実現する．しかしながら，既存の RMT Processor 向けリアルタイム OS ではタスクをスケジューリングして実行するため，ms 程度の周期実行は可能であるが，数 10 μs 程度の周期実行は困難である．本研究では RMT Processor の持つ割込み起床機構を用いてスケジューラからタスクを分離し，短い周期かつ小さいジッタで実行する高優先度ハードリアルタイムタスクである Responsive Task を提案する．Responsive Task は RMT Processor が持つ 8 個の論理コアのうちの 1 個を専有し，割込みによるスレッド起床機構を用いることで数 10 μs 周期で周期タスクを実行可能にする．評価において，スケジューリングして実行する Real-Time Task と比較して，Responsive Task は小さいオーバヘッドとジッタになることを示す．

キーワード：低遅延リアルタイム実行，リアルタイム OS，RMT Processor

A Low Latency Real-time Execution with Interrupt Wake-up Mechanism on RMT Processor

Kohei Osawa[1,a)] Keigo Mizotani[2,†1] Hiroyuki Chishiro[3] Nobuyuki Yamasaki[2]

Received: May 26, 2016, Accepted: November 1, 2016

Abstract: Hard real-time tasks in embedded real-time systems cause fatal errors when the deadline miss occurs. An example of hard real-time tasks such as motor control tasks is executed with high priority, short periods and small jitter. Responsive Multithreaded Processor (RMT Processor) has a prioritized simultaneous multithreading architecture that has 8 logical processing cores for fine-grained real-time processing. Our conventional real-time OSes can schedule and execute real-time tasks with *ms* periods but cannot execute them with dozens of μs periods. In this paper, we propose the Responsive Task that is a high-priority hard real-time task that occupies one logical processing core by using the interrupt wake-up mechanism on RMT Processor and that can be executed with dozens of μs periods. Experimental evaluations show that the Responsive Tasks have smaller overhead and lower jitter than our conventional software-scheduled real-time tasks.

Keywords: low latency real-time execution, real-time OS, RMT Processor

[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
[2] 慶應義塾大学理工学部
Faculty of Science and Technology, Keio University, Yokohama, Kanagawa 223–8522, Japan
[3] 産業技術大学院大学産業技術研究科
Graduate School of Industrial Technology, Advanced Institute of Industrial Technology, Shinagawa, Tokyo 140–0011, Japan
[†1] 現在，任天堂株式会社
Presently with Nintendo Co., Ltd.
[a)] ohsawa@ny.ics.keio.ac.jp

1. はじめに

ヒューマノイドロボットに代表される組込みリアルタイムシステムには，ハードリアルタイムタスクを持つシステムが多い．モータ制御等のハードリアルタイムタスクはデッドラインミスをするとシステムに致命的な障害を与えるため，デッドラインまでにタスクの実行が完了することが要求される．このようなシステムでは，リアルタイム性を満たしつつ各タスクにプロセッサ時間を割り当てるリアルタイムスケジューラが必要とされる．**図 1** に一般的なハードリアルタイムタスクの周期実行の様子と，高精度なモータ制御タスクが要求する周期実行の様子を示す．図 1 の上矢印はタスクが実行可能となる時刻を表し，青いボックスはタスクを実行している様子を表す．図 1 に示すように，一般的なハードリアルタイムタスクは実行可能時刻から実行開始時刻までの時間が周期ごとに異なるため，実周期が大きく変動する．しかしながら，高精度なモータ制御タスクは短い周期で実行し，実周期を一定にするために小さいジッタであることが要求される．

多くの場合，組込みリアルタイムシステムで用いられるプロセッサは，コストや熱，消費電力等の制約から汎用のプロセッサよりも低い周波数で動作することが望ましい．組込み用プロセッサは低周波数で動作するために，スケジューリングのときの実行タスクの選択や実行タスクの切替え等で発生するオーバヘッドが相対的に大きくなる．

Simultaneous Multithreading（SMT）アーキテクチャ [11] にリアルタイム処理で用いる優先度を付加した優先度付き SMT アーキテクチャである Responsive Multithreaded Processor（RMT Processor）[12] は，8 個の論理コアを持ち，非常に細かい粒度でリアルタイム処理を行う．しかしながら，既存の RMT Processor 向けリアルタイム OS ではタスクをスケジューリングして実行するため，ms 程度の周期実行は可能であるが，数 10 μs 程度の周期実行は困難である．

そこで本研究では，RMT Processor の割込み起床機構を用いてタスクをスケジューラから分離し，短い周期かつ小さいジッタで実行する高優先度ハードリアルタイムタスクである Responsive Task を提案する．RMT Processor の割込み起床機構とは，予めスレッドの起床に使用する割込みと起床するスレッドを設定すると，割込み発生時に対応するスレッドが実行を開始する機構である．Responsive Task は従来のハードリアルタイムタスクの実行と共存しつつ，RMT Processor の割込み起床機構を用いることで数 10 μs 程度の周期実行を実現する．Responsive Task と，スケジューリングして実行する従来のハードリアルタイムタスクである Real-Time Task を同時に実行し，両タスクについて起床してから実行を開始するまでのオーバヘッドとジッタを評価する．

本論文の構成は次のとおりである．まず，2 章で背景について述べる．次に 3 章では本研究の対象とするプロセッサである RMT Processor について述べる．4 章では本研究で提案する Responsive Task について述べる．5 章では RMT Processor における Responsive Task の有効性の評価およびその考察を行う．最後に 6 章では結論を述べる．

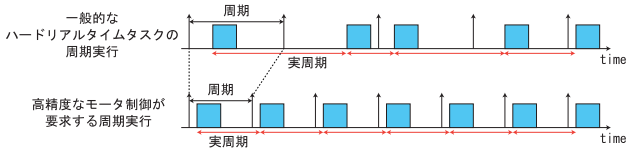


図 1 高精度なモータ制御タスクの要求
Fig. 1 Requirements of high precision motor control tasks.

2. 背景

2.1 システムモデル

本研究が対象とする組込みリアルタイムシステムのシステムモデルについて述べる．対象とするプロセッサは優先度付き SMT プロセッサであり，1 個のプロセッサと m 個の論理コアを持ち，k 番目の論理コアを LC_k とする．各論理コアは優先度を持ち，高優先度の論理コアは低優先度の論理コアよりも ALU 等のハードウェア資源を優先して獲得することができる．これにより，高優先度の論理コアのタスクが優先的に実行される．システムには n 個の独立した周期タスクで構成されるタスクセット $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ が与えられ，タスク τ_i は (C_i, T_i, D_i) のパラメータを持つ．C_i は最悪実行時間，T_i は周期，D_i は相対デッドラインであり，タスクの周期と相対デッドラインは等しいものとする．タスクのプロセッサ利用率は $U_i = C_i/T_i$，各論理コアに割り当てられているタスクのプロセッサ利用率は $U_{LC_k} = \sum_{\tau_i \in LC_k} U_i$ とし，プロセッサ全体の利用率は $U = \sum U_i = \sum U_{LC_k}$ と定義する．また，τ_i の j 番目のインスタンスを $\tau_{i,j}$ とし，そのリリース時刻を $r_{i,j}$，実行開始時刻を $s_{i,j}$，絶対デッドラインを $d_{i,j}$ とする．本論文では，取り扱うジッタを相対リリースジッタ（RRJ：Relative Release Jitter）として定義する．$\tau_{i,j}$ における相対リリースジッタ $RRJ_{i,j}$ は $RRJ_{i,j} = |(s_{i,j} - r_{i,j}) - (s_{i,j-1} - r_{i,j-1})|$ $(j > 1)$ と定義する．また，タスクを動的にマイグレーションするグローバル方式はオーバヘッドが大きいため [5]，指定する論理コアにタスクを静的に割り当てるパーティション方式を採用する．

2.2 関連研究

リアルタイム性を満たしつつ複数のタスクを処理するためには，リアルタイムスケジューリングアルゴリズムが重要である．代表的なリアルタイムスケジューリングアルゴリズムには Earliest Deadline First（EDF）[6] と Rate Monotonic（RM）[6] がある．リアルタイム OS はこれらの

ようなアルゴリズムを用いてリアルタイム性を満たす範囲で複数のタスクをスケジューリングする．モータ制御のようなハードリアルタイムタスクでは，制御の精度を高めるためにより短い周期での実行が求められる．実行周期が短いほどジッタの影響は大きくなるため，短い周期で実行されるタスクにおいてはジッタを抑えることも重要となる．しかしながら，スケジューリングしてタスクを実行する場合スケジューリングのオーバヘッドが発生する．さらに，スケジューリングするタスクの数が増えるとオーバヘッドとジッタが大きくなる傾向にある．**図 2** に RMTP 上でのスケジューラによるタスクの実行例を示す．図 2 の上矢印「リリース」はタスク τ_i の j 番目のインスタンスのリリース時刻 $r_{i,j}$ を表し，下矢印「デッドライン」は絶対デッドライン $d_{i,j}$ を表す．また，Real-Time Task1，2，3 をそれぞれ τ_1，τ_2，τ_3 とする．図中で上矢印と下矢印が重なっているのは，2.1 節 で述べたようにタスクの周期 T_i と相対デッドライン D_i が等しいため，タスク τ_i の j 番目のインスタンスのリリース時刻 $r_{i,j}$ がタスク τ_i の $j-1$ 番目のインスタンスの絶対デッドライン $d_{i,j-1}$ と一致することを表す．この例では同時に実行可能なスレッド数は 2 個とする．タスクのリリースから実行開始までにスケジューラによるタスクの選択等の処理を行うため，非常に短い周期実行は困難である．さらに，スケジューラの処理時間の変動により大きなジッタが発生する．

短い周期で小さいジッタの実行を可能とする Linux カーネルのリアルタイム拡張として ART-Linux [15], RT-Preempt Patch [7] 等があげられる．ART-Linux では，スケジューラによる遅延を削減するために，タスクをスケジューラから分離するシステムコールを実装している．これを利用することで ms 程度の周期実行を可能にする [4]．RT-Preempt Patch では，プリエンプション可能なロック機構の実装，優先度継承プロトコル [9] を実装している．しかしながら，これらは Linux カーネルをベースに設計されているためにオーバヘッドが大きい．

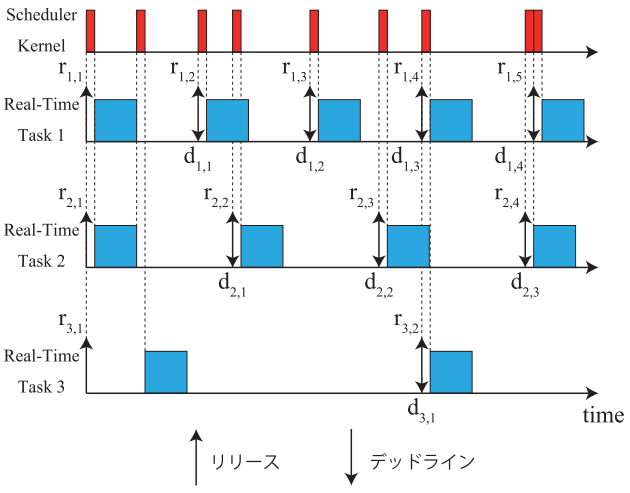


図 2 RMT Processor におけるスケジューラによるタスクの実行
Fig. 2 Task execution by a scheduler on RMT Processor.

次に，RMT Processor を対象としたリアルタイム OS について述べる．μITRON4.0 仕様 [14] に準拠した RMT Processor 向けに拡張されたリアルタイム OS として RTRON [16] がある．RTRON では，汎用レジスタやプログラムカウンタ等を含むコンテキスト情報を退避可能な専用のオンチップメモリであるコンテキストキャッシュ [12] を用いて，コンテキストスイッチのオーバヘッドを大幅に削減する．しかしながら，コンテキストキャッシュを利用した場合においても 10 μs 以上のスケジューラのオーバヘッドが発生する．また，RMT Processor の固有機能を活用する専用のリアルタイム OS [13] を独自に開発しているが，RTRON と同様に 10 μs 以上のスケジューラオーバヘッドが発生してしまう [5]．したがって，RMT Processor を対象としたリアルタイム OS についても数 10 μs 単位での周期実行は困難である．

3. Responsive Multithreaded Processor

3.1 RMT Processor の概要

本研究で対象とするプロセッサである RMT Processor [12] に関して述べる．RMT Processor はリアルタイム処理をハードウェアで支援する RMT Processing Unit 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 等）や制御用 I/O（PWM ジェネレータ，パルスカウンタ等）を 1 チップに集積した SoC（System-on-Chip）である．本研究では RMT Processor の 1 つである Dependable RMT Processor I（D-RMTP I）[10] を対象に実装を行う．D-RMTP I の構成を**表 1** に示す．D-RMTP I は 8 個の論理コアを持ち最大 8 スレッドまで同時実行可能であり，各論理コアのプロセッサ利用率 U_{LC_k} は最大 1 のため，D-RMTP I の全体のプロセッサ利用率 U は最大 8 である．しかしながら，表 1 より同時命令発行数および ALU の数が 4 であるため，Instructions Per Clock（IPC）の合計が 4 を超えることはできない．また，それぞれ 32 KByte の

表 1 D-RMTP I の構成
Table 1 Outline of the D-RMTP I.

Clock Frequency	62.5 MHz
Active Thread	8
Integer Register	32 bit × 32 entry × 8 set
Floating Point Register	64 bit × 8 entry × 8 set
Fetch Width	8
Issue Width	4
ALU	4 + 1(Divider)
FPU	2 + 1(Divider)
Branch Unit	2
Memory Access Unit	1

表 2 スレッド制御命令
Table 2 Thread control instruction.

命令	機能
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に退避する
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

命令キャッシュとデータキャッシュ，64 KByte の SRAM，64 MByte の SDRAM を搭載している．RMT Processor は優先度付き SMT アーキテクチャであり，ALU 等のスレッド間で競合が発生するハードウェア資源を 256 段階の優先度によりスレッドを調停して実行することが可能である．このハードウェア資源の調停に用いられる優先度を「ハードウェアコンテキストの優先度」とする．ハードウェアコンテキストの優先度はソフトウェアで決定し，リアルタイムスケジューリングアルゴリズムが決定するタスク実行の優先度とは異なるパラメータである．

3.2 スレッド制御機構

まず，RMT Processor のコンテキストキャッシュ [12] に関して説明し，次にコンテキストキャッシュを用いたスレッド制御機構について説明する．

コンテキストキャッシュとは，プログラムカウンタや汎用レジスタ等のコンテキスト情報を格納することが可能な RMT Processor のオンチップメモリであり，スレッド制御用の専用命令によってコンテキスト情報の退避および復帰を行うことが可能になる．RMT Processor でコンテキストスイッチを行う際，メモリアクセス命令を利用する場合は 590 クロックサイクルを要するが，スレッド制御命令を利用する場合は 4 クロックサイクルで実現可能になる．RMT Processor は 32 エントリのコンテキストキャッシュを搭載しているため，8 個の論理コア内にあるスレッドと合計して 40 個のコンテキストをプロセッサ内に保持することが可能になる．

表 2 に RMT Processor 固有のスレッド制御命令，**図 3** にこれらのスレッド制御命令による RMT Processor におけるスレッドの状態遷移図を示す．スレッドごとにコンテキストを持ち，Active Thread RUN 状態のスレッドは実行状態にある．Active Thread STOP 状態のスレッドはコンテキストを持つが実行されず，Cache Thread 状態のスレッドはコンテキストキャッシュに退避されている．RMT Processor は，図 3 に記載されている命令を発行すると，

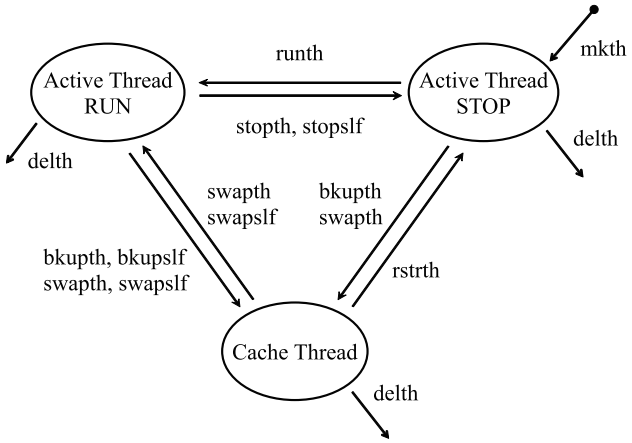


図 3 スレッドの状態遷移
Fig. 3 State transition of thread.

スレッドの状態を遷移することが可能である．

また，RMT Processor は，割込みによってスレッド起床することが可能な割込み起床機構を持つ．RMT Processor の割込み起床機構は，Active Thread STOP の状態のスレッドに起床割込みが発生した場合，対象のスレッドは 1 クロックサイクル後に Active Thread RUN 状態に遷移し，実行を開始する．つまり，割込み発生時に表 2 に示す runth 命令を実行することなく，図 3 に示す runth 命令と同様の状態遷移を 1 クロックサイクルで行うハードウェアの機構である．

4. Responsive Task

本研究で提案する Responsive Task は，3.2 節 で述べた RMT Processor の割込み起床機構を用いてタスクをスケジューラから分離し，短い周期かつジッタの小さいリアルタイム実行を実現するハードリアルタイムタスクである．Responsive Task は RMT Processor の割込み起床機構を用いて CPU タイマの割込みにより 1 クロックサイクルで起床し，周期タスクのリリースを行うことでオーバヘッドの小さい周期実行を実現する．RMT Processor は論理コアごとに CPU タイマを持っているため，Responsive Task は論理コアごとに異なる周期のタスクをリリースすることが可能である．また，割込み起床機構を用いて起床すること以外にオーバヘッドを削減するための工夫として，1 個の Responsive Task が RMT Processor の 8 個ある論理コアの 1 個を専有することとする．この設計によって Responsive Task の最大実行可能数は RMT Processor 上では 8 個に制限され，9 個以上のタスクを RMT Processor 上で実行する場合，Responsive Task のみをサポートするリアルタイム OS ではすべてのタスクを実行することができないという問題が生じる．この問題の解決方法として，Responsive Task の最大実行可能数である 8 個を超える数のタスクを実行する場合，Responsive Task として実行できない残りのタスクを Real-Time Task として実行可能と

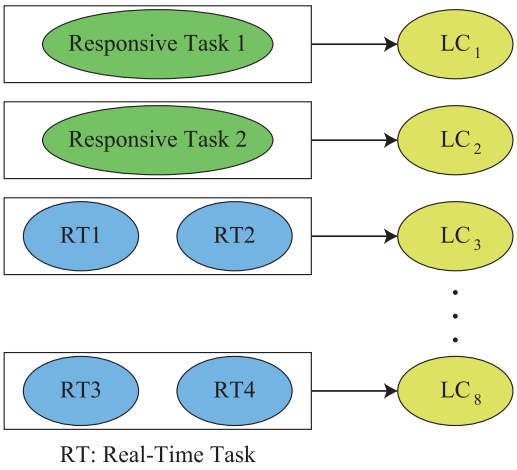


図 4 RMT Processor における Responsive Task と Real-Time Task の割当て例

Fig. 4 Example of assignment in Responsive Task and Real-Time Task on RMT Processor.

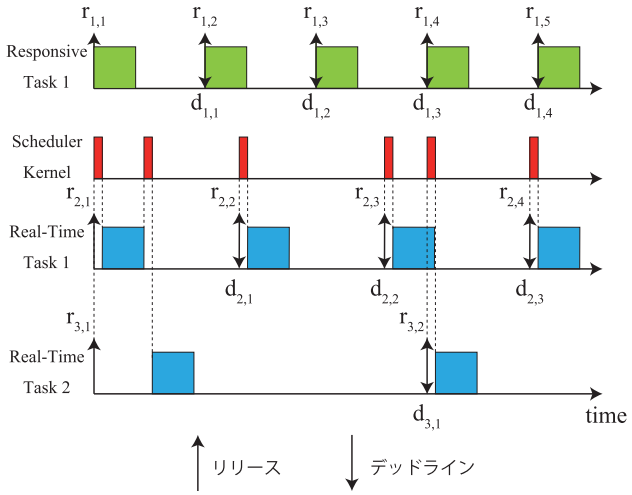


図 5 Responsive Task と Real-Time Task の同時実行

Fig. 5 Simultaneous execution of Responsive Task and Real-Time Task.

する．このとき，図 4 に示すように Real-Time Task を割り当てる論理コアが 1 個以上必要なため，9 個以上のタスクを RMT Processor 上で実行する場合 Responsive Task を最大 7 個実行し，残りの論理コアに Responsive Task でないタスクを Real-Time Task として割り当てる．この場合，Responsive Task に Real-Time Task よりも高い優先度を割り当てることで，ハードウェア資源の競合によるリアルタイム性の低下を抑制する．

図 4 に RMT Processor における Responsive Task と Real-Time Task の割当て例を示す．論理コア LC_1 には Responsive Task1，論理コア LC_2 には Responsive Task2 がそれぞれ割り当てられる．これに対して Real-Time Task は 1 個の論理コアを複数の Real-Time Task で共有して実行するため，図 4 の例では論理コア LC_3 には Real-Time Task1 と Real-Time Task2，論理コア LC_8 には Real-Time Task3 と Real-Time Task4 がそれぞれ割り当てられる．

図 5 に Responsive Task を 1 個と Real-Time Task を 2 個同時に実行する例を示す．図 2 と同様に図 5 の上矢印「リリース」はタスク τ_i の j 番目のインスタンスのリリース時刻 $r_{i,j}$ を表し，下矢印「デッドライン」は絶対デッドライン $d_{i,j}$ を表す．また，Responsive Task1 を τ_1，Real-Time Task1，2 をそれぞれ τ_2，τ_3 とする．図中で上矢印と下矢印が重なっているのは，図 2 と同様にタスクの周期 T_i と相対デッドライン D_i が等しいため，タスク τ_i の j 番目のインスタンスのリリース時刻 $r_{i,j}$ がタスク τ_i の $j-1$ 番目のインスタンスの絶対デッドライン $d_{i,j-1}$ と一致することを表す．Responsive Task1 は割込み起床機構を用いてスケジューラから分離するため，小さいオーバヘッドで実行可能である．これに対して，Real-Time Task1 は Real-Time Task2 と同じ論理コアで実行するため，スケジューラによるオーバヘッドが発生する．

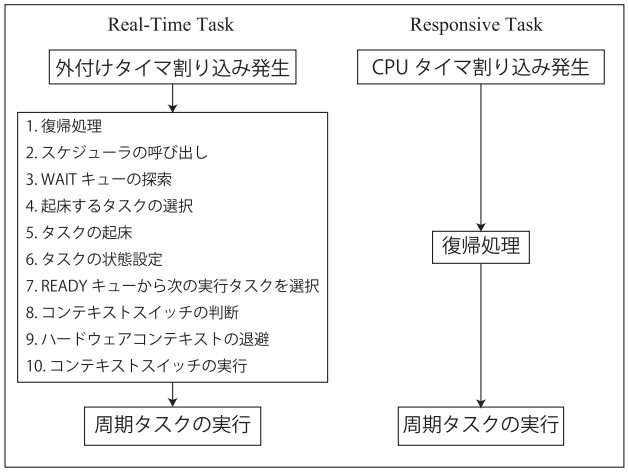


図 6 周期実行における Real-Time Task と Responsive Task の動作

Fig. 6 The behavior of Real-Time Task and Responsive Task on periodic execution.

4.1 設計

RMT Processor の割込み起床機構を用いて Responsive Task を実現するための設計について述べる．RMT Processor は 8 個の論理コアに対して CPU タイマを各々持つ．Responsive Task は専有する論理コアの CPU タイマの割込みが発生すると，RMT Processor の割込み起床機構によって Active Thread RUN 状態に遷移して周期実行を開始し，表 2 に示す stopslf 命令を用いて周期実行を終了することでスケジューラから独立した周期実行を実現する．

図 6 に割込み発生時における Real-Time Task と Responsive Task の動作を示す．Real-Time Task では READY キューと WAIT キューでそれぞれ実行可能状態のタスクと待機状態のタスクを管理する．Real-Time Task の場合タイマ割込みが発生すると，割込みベクタから復帰後にスケジューラを呼び出す．スケジューラは以下の処理を各論

理コアに対して行う．

(1) タスクの WAIT キューを探索し，起床するタスクを選択する．
(2) 選択したタスクを実行可能状態に更新し，READY キューに格納する．
(3) READY キューから実行するタスクを選択する．
(4) コンテキストスイッチの有無を判定し，必要があればコンテキストスイッチを行う．
(5) スレッド制御命令タスクの実行を開始する．

また，スケジューラは各キューの操作時にロックを獲得する．

一方 Responsive Task では，実行開始までのオーバヘッドを可能な限り抑えるために CPU タイマ割込み発生時は割込みベクタからの復帰処理のみを行う．RMT Processor では CPU タイマ割込みが発生すると 1 クロックサイクル後に割込みフラグがクリアされ，復帰処理のみで周期実行に戻ることが可能である．割込みからの復帰処理をした場合表 2 中の runth 命令と同等の動作をするため，明示的に runth 命令を実行しなくても Active Thread RUN 状態に遷移する．本節では Responsive Task を周期実行する際の周期的な CPU タイマ割込みについてのみ述べているが，他の I/O 等の割込みにより Responsive Task を起床させることも可能である．

Responsive Task と Real-Time Task の共存について述べる．Responsive Task と Real-Time Task は，ともにタイマ割込み発生時に周期実行を開始するための処理を行う．そのため，これら 2 つの処理を併用可能とするためには判定と分岐処理が必要となり，オーバヘッドが発生する．このオーバヘッドを削減するため，RMT Processor の機能を利用する．RMT Processor では例外および割込み発生時に，登録されたアドレスに例外および割込みの種類に応じたオフセットを加えたアドレスにプログラムカウンタを移す．この機能を用いて，Real-Time Task は RMT Processor 内蔵の制御用 I/O であるパルスカウンタを外付けタイマとして使用し，Responsive Task 用の割込みには論理コアが持つ CPU タイマを使用する．これらのタイマ割込みは上記で説明したオフセットの値が異なるため，別の割込みとして処理することで判定と分岐処理が不要になり，Real-Time Task と Responsive Task の共存を達成しつつオーバヘッドを削減可能である．

4.2 実装

4.1 節 で述べた設計に基づいて，RMT Processor 向けのリアルタイム OS [13] に Responsive Task の実装を行う．この OS には Real-Time Task を周期実行するための機構がすでに実装されており，4.1 節で述べたように外付けタイマ割込みの割込み処理で Real-Time Task のリリースを行う．各々の論理コアで Real-Time Task のリリース処理を行う場合，ハードウェア資源の競合が頻繁に発生し，リリース処理のオーバヘッドが大きくなってしまう [5]．Real-Time Task のリリース処理のオーバヘッドを可能な限り小さくするため，本 OS はスケジューラが必ず 1 個の論理コアを専有し，周期タスクのリリースおよびそれにともなう再スケジューリングの処理を行う．Real-Time Task のリリース時に行われる処理は図 6 のとおりである．ただし，次の外付けタイマ割込み発生までに実行中のタスクが終了した場合は該当する論理コア上でスケジューラが呼び出される．よって，各論理コア LC_1 から LC_8 が実行する内容は以下のようになる．

- LC_1：タスクリリースを行うスケジューラ
- LC_2〜LC_8：Responsive Task または Real-Time Task

この実装は，RTRON [16] のスケジューラの実装方法を参考にしている．

表 3 Responsive Task 用の API
Table 3 API for Responsive Task.

API 名	内容
create_task_resp	Responsive Task の生成
resp_wait_period	Responsive Task の実行を次の周期まで停止
resp_start_scheduler	CPU タイマの設定および開始

本 OS に Responsive Task を実装するにあたって，Real-Time Task と Responsive Task のどちらを生成するかユーザが選択可能とするため，Real-Time Task 向け API とは別に Responsive Task 向け API を実装した．**表 3** に実装した API を示す．次に各 API の実装内容について詳しく述べるとともに，Real-Time Task 向け API との差異について述べる．

- create_task_resp 関数
  Responsive Task の生成を行う．POSIX [3] の仕様にある pthread_create 関数の引数を参考に，Responsive Task を割り当てる論理コア ID，RMT Processor 固有機能であるスレッドの優先度と，周期実行に必要な情報である周期と最悪実行時間を引数として与えており，Real-Time Task を生成する API の引数と等しい．これは，周期実行に必要な情報が Real-Time Task と Responsive Task で差異がない点と，ユーザが API 名を変更するだけで生成するタスクの種類を切替え可能とするためである．Real-Time Task を生成する API は生成した周期タスクを各論理コアのタスクキューに挿入する．Responsive Task は割込み起床機構を用いてスケジューラから分離しているため，各論理コアへの割当てやタスクキューへの挿入等は不要である．代わりに Responsive Task には論理コアを専有するように実装するため，Responsive Task がどの論理コアを専有しているかを管理するテーブルを用意する．本 API は Responsive Task 生成時にこの管理テーブルを

参照し，指定した論理コアが Responsive Task によって専有されていない場合は Responsive Task の生成に成功し，管理テーブルにタスクの情報を登録する．

- resp_wait_period 関数
  Responsive Task の実行を，次の CPU タイマの割込み発生まで停止する．本 API は stopslf 命令を実行し，Responsive Task を Active Thread STOP 状態に遷移させる．4.1 節 で述べたように，Responsive Task は CPU タイマ割込みが発生すると RMT Processor の機能で自動的に Active Thread RUN 状態に遷移するため，スケジューラを呼び出すことなく周期実行を再開することが可能である．Real-Time Task が周期実行を完了したときに呼び出す API では，次の周期の開始時刻等のタスク情報を更新してスケジューラを呼び出す．
- resp_start_scheduler 関数
  生成した各 Responsive Task の周期に応じて CPU タイマの設定を行い，カウントを開始する．本 API 実行後は，論理コアごとに設定した周期で CPU タイマの割込みが発生し，対応する Responsive Task が起床する．Real-Time Task 用の同様の API ではスケジューラ用に外付けタイマの設定を行い，スケジューラを呼び出す．本 API の実装によって，4.1 節 で述べたように Responsive Task と Real-Time Task を異なるタイマの割込みで処理する．

5. 評価

本章では，Responsive Task の有効性を D-RMTP I 評価キットを用いて評価する．D-RMTP I の評価キット上で Responsive Task と Real-Time Task を同時に実行し，Real-Time Task のプロセッサ利用率を高くしたときの Responsive Task と Real-Time Task の性能を比較する．

5.1 評価環境

図 7 に評価で用いる D-RMTP I 評価キットの外観を示す．D-RMTP I 評価キットは D-RMTP SoC，SDRAM や Flash Memory 等を集積した D-RMTP SiP（System-in-Package）を搭載している．D-RMTP I の仕様は表 1 に示したとおりである．SiP のほかには各種 I/O ピンや FPGA，オシレータ等を搭載している．Responsive Task を実装するリアルタイム OS は SRAM に十分に格納できるサイズであるため，SDRAM よりメモリアクセスの遅延が短い SRAM 上で実行する．評価結果に関するデータは SRAM に格納できないサイズのため，容量が大きい SDRAM に格納する．

次に，実機評価に用いた各パラメータについて述べる．今回 Responsive Task の実装を行った RMT Processor 向けリアルタイム OS [13] は，4.2 節 で述べたようにスケジューラが必ず 1 個の論理コアを専有するため，Responsive Task や Real-Time Task を実行する論理コアは 7 個となる．Real-Time Task の割当てはタスクのプロセッサ利用率 U_i と各論理コアのプロセッサ利用率 U_{LC_k} から決定され，アルゴリズムには worst-fit algorithm [2] を用いる．また，各論理コアの Real-Time Task のスケジューリングには RM アルゴリズムを用いる．評価用タスクは ALU による加算，減算，乗算，除算を行う四則演算タスクと，主記憶に読み書きを頻繁に行う配列演算タスクの 2 種類を用意する．評価タスクは四則演算タスクと配列演算タスクのどちらか一方のみ実行し，Responsive Task の数を 1 個から 3 個，Real-Time Task を割り当てる論理コアを 1，2，4 個と変化させ，プロセッサ利用率を 0.10 から 1.50 まで 0.05 おきで設定する．ただし，Responsive Task と Real-Time Task のどちらかがデッドラインミスした場合は，そのときのプロセッサ利用率未満のデータを評価として用いる．また，Responsive Task の利用率の和以下のプロセッサ利用率を設定した場合は Real-Time Task が生成されないため，そのようなプロセッサ利用率の評価は行わない．スケジューラの tick を 5 ms，評価時間を 100 ms とする．

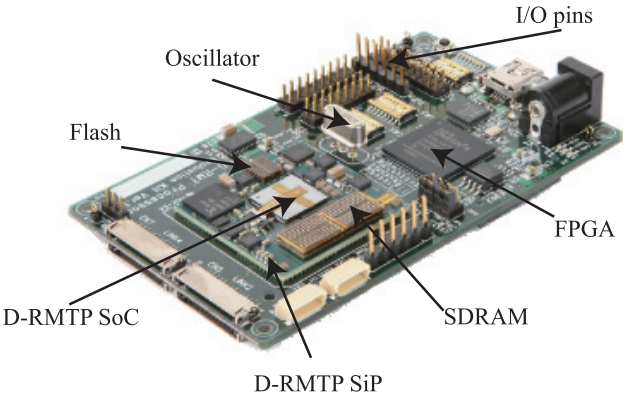


図 7 D-RMTP I 用評価キット
Fig. 7 Evaluation kit for D-RMTP I.

次に Responsive Task と Real-Time Task のパラメータについて述べる．Responsive Task の周期は，Responsive Task の数に応じて 50 μs から 70 μs まで 10 μs おきで設定する．たとえば，Responsive Task が 1 個の場合は周期 50 μs の Responsive Task を 1 個，Responsive Task が 2 個の場合は周期 50 μs と周期 60 μs の Responsive Task を 1 個ずつ生成する．Responsive Task の最悪実行時間は等しく 3 μs となるように設定し，プロセッサ利用率は周期の短い順に 0.06，0.05，0.04 となる．Real-Time Task の周期は，全タスクの周期の最小公倍数（ハイパーピリオド）を評価時間の約数とするために，スケジューラの tick に対して 1，2，4，5 倍，つまり 5，10，20，25 ms から無作為に決定する．このとき，各タスクのプロセッサ利用率が $[0.01, 0.10]$ の範囲となるような最悪実行時間を無作為に決定する．乱

数の生成にはメルセンヌ・ツイスタ法 [8] を用いて，シード値にはそのときの RMT Processor のカウンタ値を用いる．ここで各論理コアに割り当てられる Real-Time Task の数について述べる．各論理コアへの Real-Time Task の割当てには worst-fit algorithm を用いるため，各論理コアのプロセッサ利用率 U_{LC_k} はほぼ等しいと仮定することができる．論理コア数は m 個であるため，各論理コアのプロセッサ利用率は $U_{LC_k} = U/m$ となる．各タスクのプロセッサ利用率 U_i が $[0.01, 0.10]$ となるようにタスクを生成するため，各論理コアに割り当てられる Real-Time Task の数は $U_{LC_k}/0.10$〜$U_{LC_k}/0.01$ となる．たとえば Real-Time Task を割り当てる論理コアの数を 2 個，プロセッサ利用率を 0.80 とすると，各論理コアに割り当てられる Real-Time Task の数は 4〜40 個である．

次にハードウェアコンテキストの優先度の設定について述べる．Responsive Task については周期が 50 μs の Responsive Task を最高優先度とし，周期の長い Responsive Task ほど優先度を低く設定する．同様にスケジューラの優先度はすべての Responsive Task より低く設定し，Real-Time Task の優先度はすべて等しく，スケジューラの優先度より低く設定する．

5.2 評価指標

本論文で提案した Responsive Task は，短い周期かつ小さいジッタで実行するハードリアルタイムタスクである．そのため，評価では Real-Time Task のプロセッサ利用率を高くしたときに Responsive Task の実行に対する影響を示す必要がある．そこで，タイマ割込み発生からタスクが起床するまでの起床オーバヘッドと，相対リリースジッタを評価指標として用いる．Responsive Task は自身の CPU タイマのカウンタ値，Real-Time Task はスケジューラの CPU タイマのカウンタ値をそれぞれ取得し，タイマ割込みが発生してから周期タスクに処理が戻るまでのサイクル数を計測して起床オーバヘッドとする．相対リリースジッタは，計測した起床オーバヘッドを用いて 2.1 節 で定義した式から導出する．

比較対象には，Responsive Task と同時実行する Real-Time Task の中で最も周期の短いタスクを用いる．リアルタイムスケジューリングアルゴリズムとして使用する RM アルゴリズムでは，周期が最も短いタスクのジッタは理論上 0 である [1]．よって，各論理コア上で最も周期の短い Real-Time Task と比較することで Responsive Task の有効性を示す．

5.3 1 タスク実行時の評価

Real-Time Task のプロセッサ利用率を高くしたときに Responsive Task と最も周期の短い Real-Time Task それぞれに与える影響を比較するため，Responsive Task を 1 個のみ実行した場合と，Real-Time Task を 1 個のみ実行した場合の起床オーバヘッドと相対リリースジッタの測定を行った．**表 4** に，Responsive Task を 1 個のみ実行した場合と Real-Time Task を 1 個のみ実行した場合の起床オーバヘッドを示し，**表 5** に相対リリースジッタを示す．表 4 より，Responsive Task は四則演算タスクと配列演算タスクの両方のタスクにおいて 1 μs 未満の起床オーバヘッドであり，起床オーバヘッドの最小値と最大値には差がない．一方で，Real-Time Task は四則演算タスクと配列演算タスクの両方のタスクにおいて最大起床オーバヘッドは 18 μs 程度であった．また，表 5 より Responsive Task の相対リリースジッタはタスクによらず 0 μs であったが，Real-Time Task の相対リリースジッタは最大 0.016 μs であった．表 1 より D-RMTP I の動作周波数は 62.5 MHz であり，このとき 1 クロックサイクルは 0.016 μs である．よって，Real-Time Task ではスケジューラの処理に 1 クロックサイクルの揺らぎが生じている．

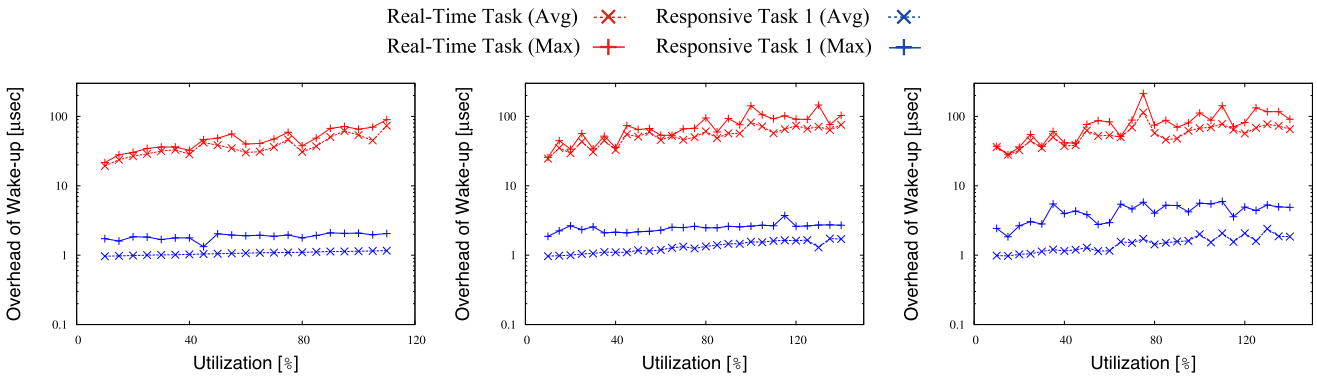
表 4 1 タスク実行時の起床オーバヘッド
Table 4 Overhead of wake-up of executing 1 task.

	タスク	最小値 [μs]	平均値 [μs]	最大値 [μs]
Responsive Task	四則演算	0.928	0.928	0.928
	配列演算	0.944	0.944	0.944
Real-Time Task	四則演算	18.432	18.444	18.448
	配列演算	18.656	18.668	18.672

表 5 1 タスク実行時の相対リリースジッタ
Table 5 Relative release jitter of executing 1 task.

	タスク	最小値 [μs]	平均値 [μs]	最大値 [μs]
Responsive Task	四則演算	0.000	0.000	0.000
	配列演算	0.000	0.000	0.000
Real-Time Task	四則演算	0.000	7.803×10^{-3}	0.016
	配列演算	0.000	8.207×10^{-3}	0.016

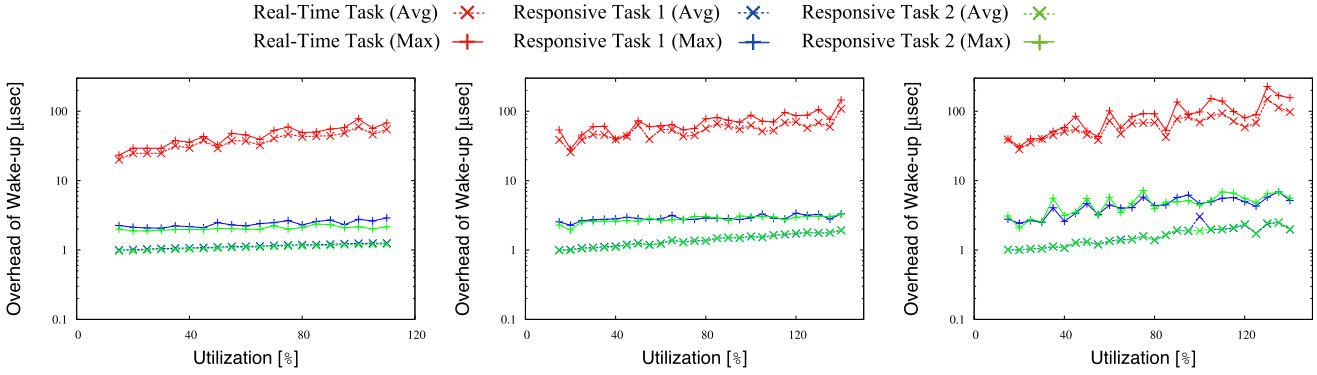
表 4，表 5 の評価結果を統合すると，Real-Time Task の場合，四則演算タスクよりも配列演算タスクの方が悪い結果となった．これは D-RMTP I のメモリアクセスユニットが 1 個しかないことが原因であり，頻繁にメモリアクセスを行う配列演算タスクは ALU による演算の多い四則演算タスクと比べて，より最悪に近い評価となる傾向にある．Responsive Task の場合，四則演算タスクと配列演算タスクの起床オーバヘッドが 1 クロックサイクル異なるのは，4.2 節 で述べた resp_wait_period 関数内で CPU タイマ割込み発生後に行う復帰処理とカウンタ値の取得命令のアラインメントが原因である．D-RMTP I のフェッチ幅は表 1 より 8 命令である．そのため，resp_wait_period 関数内の復帰処理とカウンタ値の取得命令が同じ 8 ワードアラインメントの場合は同時にフェッチされ，異なる 8 ワードアラインメントの場合は同時にフェッチされない．このとき，両者の間にはカウンタ値の取得命令が実行される時間に 1



(a) 論理コア 1 個　(b) 論理コア 2 個　(c) 論理コア 4 個

図 8　配列演算タスクの起床オーバヘッド（Responsive Task 1 個）

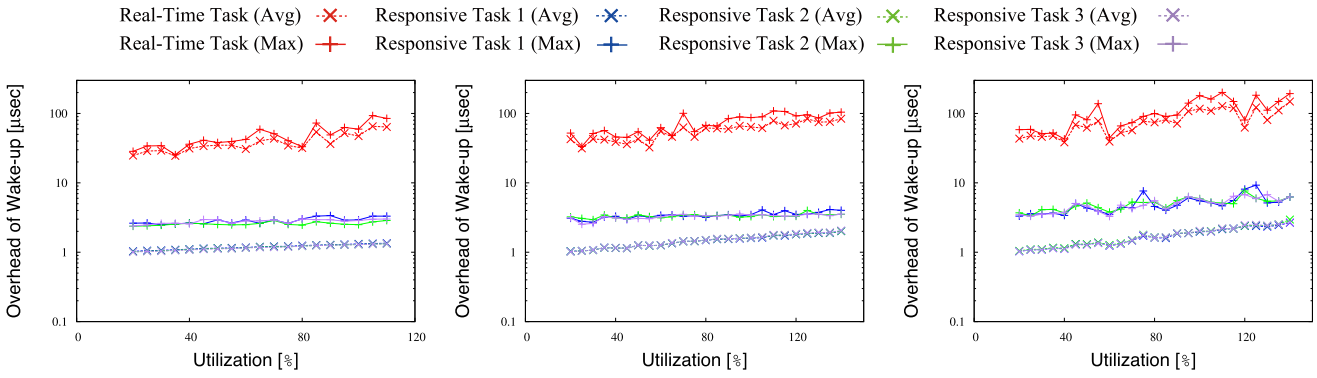
Fig. 8　Overhead of task of array access operations (1 Responsive Task).



(a) 論理コア 1 個　(b) 論理コア 2 個　(c) 論理コア 4 個

図 9　配列演算タスクの起床オーバヘッド（Responsive Task 2 個）

Fig. 9　Overhead of task of array access operations (2 Responsive Tasks).



(a) 論理コア 1 個　(b) 論理コア 2 個　(c) 論理コア 4 個

図 10　配列演算タスクの起床オーバヘッド（Responsive Task 3 個）

Fig. 10　Overhead of task of array access operations (3 Responsive Tasks).

クロックサイクルの差が生じる．これはタスク依存ではなくコーディング依存であり，コンパイル時に解析可能である．以上をふまえてより最悪の場合を想定するため，以後は配列演算タスクを実行した場合の評価を示す．

5.4　起床オーバヘッドの評価

Responsive Task と Real-Time Task について，それぞれ Real-Time Task のプロセッサ利用率を高くした場合の起床オーバヘッドを評価した．5.2 節 に基づいて，計測時間 100 ms の Responsive Task ごとの平均値と最大値を評価結果として用いた．Real-Time Task については，論理コアごとに最高優先度が割り当てられたタスクの平均値と最大値を集計したが，ほぼ同じ傾向を示したため，最も番号が小さい論理コアの値を評価結果として用いた．**図 8** に Responsive Task を 1 個実行し，Real-Time Task を実行する論理コアを 1，2，4 個とした場合の配列演算タスク

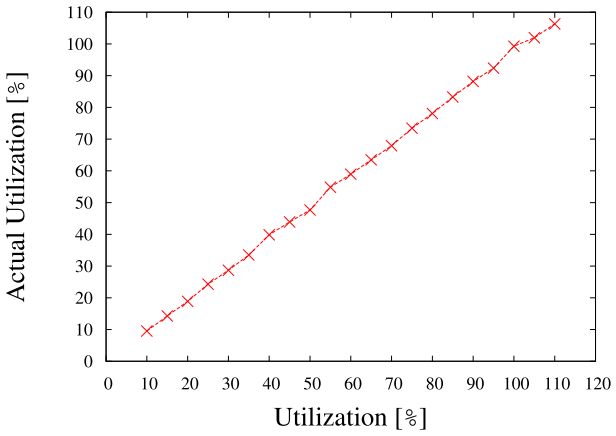


図 11 図 8 (a) の実際のプロセッサ利用率
Fig. 11 Actual utilization of Fig. 8 (a).

の結果を示す．縦軸は起床オーバヘッドを示し，横軸はプロセッサ利用率を示す．同様の条件で**図 9** に Responsive Task を 2 個，**図 10** に Responsive Task を 3 個実行した場合の結果を示す．

図 8，図 9，図 10 より Real-Time Task はプロセッサ利用率が低い場合でも起床オーバヘッドは数 10 μs 程度であり，プロセッサ利用率が高くなると最大で 200 μs 程度であった．また，図 8 (a)〜(c) より，Real-Time Task を実行する論理コア数が増えると最大起床オーバヘッドが大きくなることが分かる．これは，Real-Time Task を実行する論理コア数が増えるとタスクキューの数が増え，スケジューラがタスクキューの操作を行う処理時間が増大したことと，同時実行するスレッド数が増えたことでスレッド間のハードウェア資源の競合が多くなったことが原因である．

一方で，図 8 (a)，図 9 (a)，図 10 (a) と図 8 (b)，図 9 (b)，図 10 (b) より Responsive Task の起床オーバヘッドは Responsive Task の数に依存せず，Real-Time Task を 1 個または 2 個の論理コアで実行する場合はほぼ一定であることが分かる．これに対して図 8 (c)，図 9 (c)，図 10 (c) より，Real-Time Task を実行する論理コア数を増やした場合は低いプロセッサ利用率のときでも Responsive Task の起床オーバヘッドが大きい．Real-Time Task を実行する論理コア数が増えると Responsive Task と同時実行するスレッド数が増える．評価で用いた配列演算タスクはメモリアクセスを頻繁に行うタスクであり，表 1 より RMT Processor はメモリアクセスユニットを 1 個しか持たない．そのため，同時実行するスレッド数が増えると配列演算タスクのようなメモリインテンシブなタスクの場合はメモリアクセスユニットがボトルネックとなり，ハードウェア資源に空きがない可能性が高くなる．このとき Responsive Task が起床すると，メモリアクセスユニットに空きが生じるまでは Responsive Task は命令をメモリからフェッチできなくなるため，起床オーバヘッドが大きくなる原因となる．

次に，図 8 (a) において Responsive Task のプロセッサ利用率は 0.06 であるにもかかわらず，全体のプロセッサ利用率が 1.10 まで測定されている．タスクのプロセッサ利用率は 2.1 節の定義より $U_i = C_i/T_i$ で，この C_i は最悪実行時間を示している．タスクの実行時間は様々な要因で変動するため，通常，実際のタスクの実行時間は C_i より短く，実際のタスクのプロセッサ利用率は U_i より小さくなる．この実際のプロセッサ利用率とは，ジョブごとに変動するタスクの実際の実行時間を測定し，その最大値をタスクの周期で割った値である．ここで，**図 11** に図 8 (a) の実験における実際のプロセッサ利用率を示す．図 11 の縦軸「Actual Utilization」は実際のプロセッサ利用率を表す．図 11 より，プロセッサ利用率 $U = 1.1$ のときの実際のプロセッサ利用率は 1.06 程度となり，0.04 程度がプロセッサ利用率と実際のプロセッサ利用率の差である．この 1.06 のプロセッサ利用率のうち，Responsive Task のプロセッサ利用率はほぼ 0.06 であった．また，図 8 (a) で用いた実験結果では，プロセッサ利用率 $U = 1.1$ のときのスケジューラの実行時間は 100 μs 程度であった．スケジューラの tick は 5 ms のため，スケジューラの利用率は 0.02 程度となる．以上をまとめると，図 8 (a) の実験において $U = 1.1$ のときの実際のプロセッサ利用率は 1.06 で，その内訳は以下のとおりである．

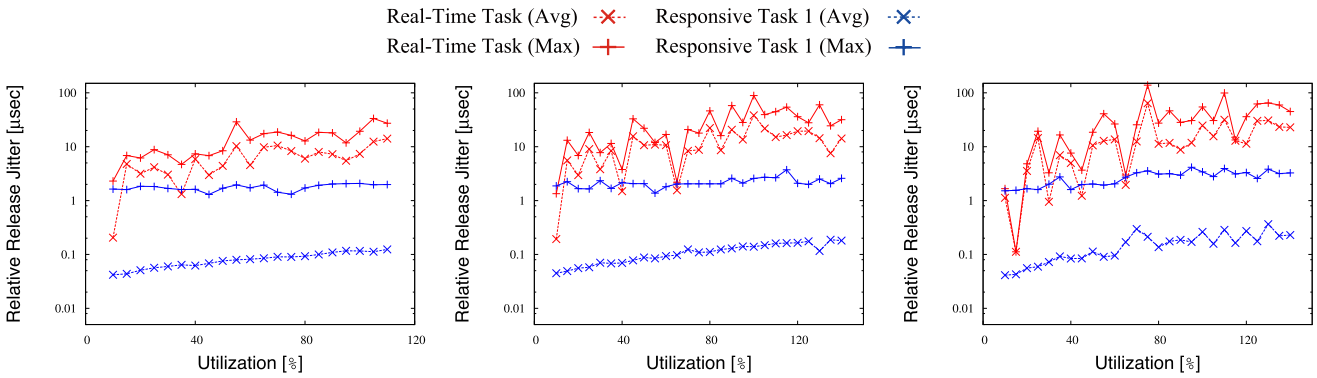
- 0.06：Responsive Task
- 0.98：Real-Time Task
- 0.02：スケジューラ

このように，Real-Time Task を割り当てる論理コアのプロセッサ利用率 U_{LC_k} が上限である 1 を超える場合でも，実際のプロセッサ利用率は上限を下回る 0.98 であったため，全体のプロセッサ利用率 $U = 1.1$ が得られた．

図 8，図 9，図 10 の評価結果を統合すると，同時に実行する Responsive Task の数よりも Real-Time Task を実行する論理コア数の方が Responsive Task の実行に与える影響が大きく，Responsive Task 間のハードウェア資源の競合はあまり多くないことが分かる．また，最も周期の短い Real-Time Task は高いプロセッサ利用率のときに起床オーバヘッドが最大で 200 μs 程度生じる．それに対して Responsive Task は，Real-Time Task のプロセッサ利用率が高い場合においても最大で 10 μs 以下の起床オーバヘッドである．5.1 節 で述べたとおり，図 8，図 9，図 10 は Responsive Task と Real-Time Task のどちらもデッドラインミスしないプロセッサ利用率における起床オーバヘッドの評価結果を示しているため，Responsive Task は時間制約を満たしつつ数 10 μs 単位の周期実行を実現することが可能である．

5.5 ジッタの評価

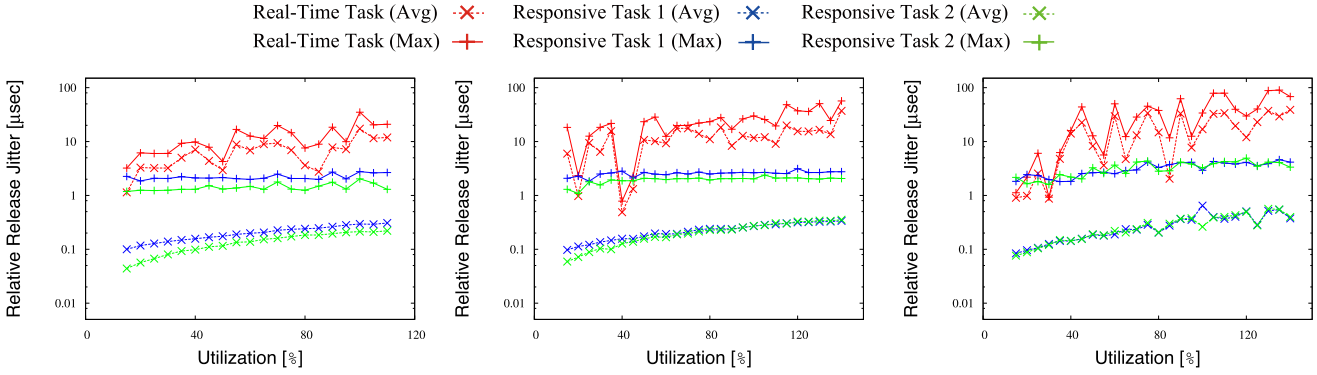
5.2 節 に基づいて，計測した起床オーバヘッドから相対リリースジッタを導出し，計測時間 100 ms における



(a) 論理コア 1 個　(b) 論理コア 2 個　(c) 論理コア 4 個

図 12　配列演算タスクの相対リリースジッタ（Responsive Task 1 個）

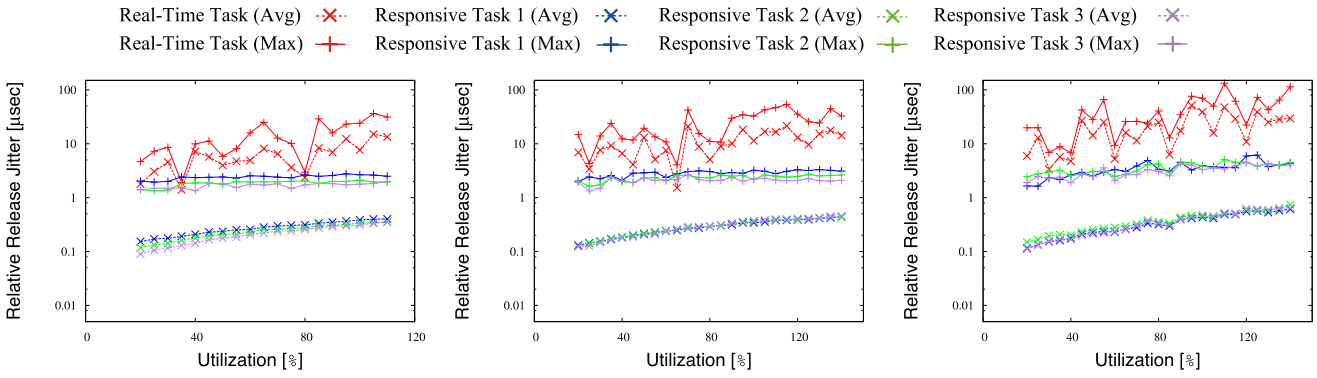
Fig. 12　Relative release jitter of task of array access operations (1 Responsive Task).



(a) 論理コア 1 個　(b) 論理コア 2 個　(c) 論理コア 4 個

図 13　配列演算タスクの相対リリースジッタ（Responsive Task 2 個）

Fig. 13　Relative release jitter of task of array access operations (2 Responsive Tasks).



(a) 論理コア 1 個　(b) 論理コア 2 個　(c) 論理コア 4 個

図 14　配列演算タスクの相対リリースジッタ（Responsive Task 3 個）

Fig. 14　Relative release jitter of task of array access operations (3 Responsive Tasks).

Responsive Task と Real-Time Task の平均値と最大値を評価結果として用いた．**図 12** に Responsive Task を 1 個実行し，Real-Time Task を実行する論理コアを 1，2，4 個とした場合の配列演算タスクの結果を示す．縦軸は相対リリースジッタを示し，横軸はプロセッサ利用率を示す．同様の条件で**図 13** に Responsive Task を 2 個，**図 14** に Responsive Task を 3 個実行した場合の結果を示す．

今回の評価で Real-Time Task のスケジューリングに利用した RM アルゴリズムは，5.2 節 で述べたように最高優先度タスクは理論上ジッタが発生させない．しかしながら，図 12 (a) に示すように，Responsive Task が 1 個，Real-Time Task を実行する論理コアが 1 個の場合でも最大で数 10 μs の相対リリースジッタが発生する．特に，図 14 (c) より Responsive Task を 3 個，Real-Time Task を 4 個の論理コアで実行した場合の最大相対リリースジッタは 100 μs 程度であった．

一方, 図 12 (a), 図 13 (a), 図 14 (a) と図 12 (b), 図 13 (b), 図 14 (b) より Real-Time Task を 1 個または 2 個の論理コアで実行する場合，Responsive Task の相対リリースジッタは Responsive Task の数に依存せず約 2 μs で一定であった．図 12 (c)，図 13 (c)，図 14 (c) より Real-Time Task を 4 個の論理コアで実行する場合は，プロセッサ利用率が高くなるに従って Responsive Task の相対リリースジッタも大きくなるが，最大でも 6 μs 程度となった．したがって，起床オーバヘッドの評価と同様に Real-Time Task を実行する論理コア数が Responsive Task の相対リリースジッタに与える影響は大きいが，RM アルゴリズムにおける最高優先度の Real-Time Task と比較して十分に小さい相対リリースジッタを達成している．

図 12，図 13，図 14 の評価結果を統合すると，Real-Time Task は最大で 100 μs 以上の相対リリースジッタが生じてしまうことが分かる．それに対して，Responsive Task はプロセッサ利用率が高い場合でも最大相対リリースジッタを 6 μs 程度に抑えることができる．また，5.4 節 と同様に図 8，図 9，図 10 は Responsive Task と Real-Time Task のどちらもデッドラインミスしないプロセッサ利用率における相対リリースジッタの評価結果を示しているため，Responsive Task は数 10 μs 単位の短い周期に対して時間制約を満たしつつ十分に小さいジッタで実行可能である．

6. 結論

本研究では，従来のハードリアルタイムタスクの実行と共存しつつ，RMT Processor の割込み起床機構を利用して短い周期かつ小さいジッタで実行するハードリアルタイムタスクである Responsive Task を提案した．実機評価により，62.5 MHz という低動作周波数において Responsive Task の起床オーバヘッドが最大 10 μs 程度と十分に小さいことを示した．また，Responsive Task の最大相対リリースジッタは 6 μs 程度であり，数十 μs 単位の短い周期の周期実行に対して十分に小さいジッタであることを示した．本研究で提案した Responsive Task により，時間制約を満たしつつ数十 μs 程度の周期実行を小さいジッタで実現することが可能になった．

今後の課題としては，Responsive Task を用いて I/O 処理を行った場合の性能調査を行う予定である．また，他の I/O 等の割込みにより Responsive Task を起床するソフトウェア機構の設計および実装を進めている．

謝辞 本研究の一部は科学技術振興機構 A-STEP 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．

参考文献

[1] Buttazzo, G.C.: Rate Monotonic vs. EDF: Judgment Day, *Real-Time Systems*, Vol.29, No.1, pp.5–26, Springer (2005).
[2] Coffman, E.G., Galambos, G., Martello, S. and Vigo, D.: Bin Packing Approximation Algorithms: Combinatorial Analysis, *Handbook of combinatorial optimization*, pp.151–207, Springer (1999).
[3] The IEEE and The Open Group: IEEE Std 1003.1, 2013 edition (2013).
[4] Kagami, S., Ishiwata, Y. and Nishiwaki, K.: ART-Linux for High-Frequency System Control, *Proc. IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*, pp.60–65 (2013).
[5] Kumura, Y., Mizotani, K., Takasu, M., Chishiro, H. and Yamasaki, N.: Overhead-Aware Schedulability Analysis on Responsive Multithreaded Processor, *Proc. 30th International Conference on Computers and Their Applications*, pp.379–386 (2015).
[6] 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).
[7] Luotao, F. and Robert, S.: RT PREEMPT HOWTO, Real-Time Linux Wiki (online), available from ⟨https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO⟩ (accessed 2014-05-07).
[8] Matsumoto, M. and Nishimura, T.: Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator, *ACM Trans. Modeling and Computer Simulation*, Vol.8, No.1, pp.3–30 (1998).
[9] Sha, L., Rajkumar, R. and Lehoczky, J.P.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization, *IEEE Trans. Comput.*, Vol.39, No.9, pp.1175–1185 (1990).
[10] 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).
[11] Tullsen, D.M., Eggers, S.J. and Levy, H.M.: Simultaneous Multithreading: Maximizing On-Chip Parallelism, *Proc. 22nd Annual International Symposium on Computer Architecture*, pp.392–403 (1995).
[12] Yamasaki, N.: Responsive Multithreaded Processor for Distributed Real-Time Systems, *Journal of Robotics and Mechatronics*, Vol.17, No.2, pp.130–141 (2005).
[13] 溝谷圭悟，上田陸平，高須雅義，千代浩之，松谷宏紀，山﨑信行：インプリサイス計算モデルにおける温度を考慮した動的電圧周波数制御の実機評価，情報処理学会論文誌，Vol.55, No.8, pp.1841–1855 (2014).
[14] 高田広章：μITRON4.0 仕様 Ver4.02.00，トロン協会 (2004).
[15] 石綿陽一，加賀美聡，西脇光一，松井俊浩：シングル CPU 用 ART-Linux 2.6 の設計と開発，日本ロボット学会誌，Vol.26, No.6, pp.78–84 (2008).
[16] 上田陸平，藤井 啓，千代浩之，松谷宏紀，山﨑信行：ITRON 仕様 OS の RMT Processor 向け実装，情報処理学会論文誌，Vol.54, No.7, pp.1835–1848 (2013).

大沢 幸平 （学生会員）

2015 年慶應義塾大学理工学部情報工学科卒業．現在，同大学大学院理工学研究科開放環境科学専攻博士課程に在籍．リアルタイムシステムにおけるオーバヘッド削減技術の研究に従事．

溝谷 圭悟

2013 年慶應義塾大学理工学部情報工学科卒業．2015 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．現在，任天堂株式会社勤務．リアルタイムシステムにおける温度制御，省電力化技術の研究に従事．

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）．2012 年度より 2013 年度まで慶應義塾大学理工学部情報工学科訪問研究員，日本学術振興会特別研究員（PD）．2014 年度より 2015 年度まで慶應義塾大学理工学部情報工学科助教．現在，産業技術大学院大学産業技術研究科情報アーキテクチャ専攻助教．リアルタイムシステム，オペレーティングシステム，ミドルウェア，トレーディングシステムの研究に従事．IEEE，ACM 各会員．

山﨑 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．