割込み起床機構を用いた低遅延リアルタイム実行

渡邊 大記[1,a)]　溝谷 圭悟[1,†1,b)]　羽鳥 雄介[2,c)]　千代 浩之[1,d)]　山﨑 信行[1,e)]

概要：近年の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とソフトリアルタイムタスクを両方持つシステムが多く存在する．多くの場合において，ソフトリアルタイムタスクは低優先度を持ちスループット重視で実行される一方，ハードリアルタイムタスクは高優先度を持ち短い周期で実行する．優先度付き Simultaneous Multithreading (SMT) 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 ではスケジュールすることでタスクを実行するため，ms 程度の周期実行は可能であるが，数十 μs 程度の周期実行は困難である．本研究では RMT Processor 上で割込み起床機構を用いて，スケジューラから分離することで，数十 μs 周期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 1 個を専有することで，数十 μs 周期で実行可能になる．評価結果では，スケジュールすることで実行される 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：リアルタイム OS，Responsive Multithreaded Processor，Simultaneous Multithreading，低遅延

A Low Latency Real-Time Execution with Interrupt Wake-up Structure

Watanabe Hiroki[1,a)]　Mizotani Keigo[1,†1,b)]　Hatori Yusuke[2,c)]　Chishiro Hiroyuki[1,d)]
Yamasaki Nobuyuki[1,e)]

Abstract: In recent embedded real-time systems, there are many systems with both hard real-time tasks and soft real-time tasks. While soft real-time tasks are executed with long periods and low priority, hard real-time tasks are executed with short periods and high priority. Responsive Multithreaded Processor (RMT Processor) is a prioritized SMT architecture and has 8 logical cores to achieve fine-grained real-time processing. However, existing real-time OSs execute tasks by scheduling, and hence these real-time OSs can execute tasks with *ms* periods and cannot execute them with dozens of μs periods. In this paper, we propose Responsive Task, which is a high-priority hard real-time task to occupy one logical core with the interrupt wake-up structure on RMT Processor and can be executed with dozens of μs periods. Experimental evaluations show that Responsive Tasks have smaller overhead and lower jitter than Real-Time Tasks by scheduling.

Keywords: Real-Time OS, Responsive Multithreaded Processor, Simultaneous Multithreading, Low Latency

[1] 慶應義塾大学 理工学部
Facalty of Science and Technology, Keio University
[2] 慶應義塾大学 大学院理工学研究科
Graduate School of Science and Technology, Keio University
[†1] 現在，任天堂株式会社
Presently with Nintendo Co., Ltd.
[a)] watanabe@ny.ics.keio.ac.jp
[b)] mizotani@ny.ics.keio.ac.jp
[c)] hatori@ny.ics.keio.ac.jp
[d)] chishiro@ny.ics.keio.ac.jp
[e)] yamasaki@ny.ics.keio.ac.jp

1. はじめに

近年の組込みリアルタイムシステムでは，ハードリアルタイムタスクとソフトリアルタイムタスクを持つシステムが多く存在する．ハードウェア制御に代表される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必ず完了することが要求される．多くの場合ハー

ドリアルタイムタスクは高優先度かつ短い周期での実行が求められる．一方，動画像処理のようなソフトリアルタイムタスクは，デッドラインミスを起こした場合でもシステムに致命的な障害を与えるわけではない．リアルタイム性よりもスループットを重視する場合が多い．多くの場合ソフトリアルタイムタスクはハードリアルタイムタスクよりも低優先度を持ち，比較的長い周期で実行される一方，ハードリアルタイムタスクは高優先度を持ち短い周期で実行される．

このようなハードリアルタイムタスクとソフトリアルタイムタスクを持つシステムでは，リアルタイム性とスループットの両立が求められる．そのため，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当てる（リアルタイム）スケジューラが必要とされ，特に高精度の制御を要求するようなハードリアルタイムタスクではより短い周期で，かつ小さいジッタでの実行が求められる．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．しかしながら，低周波数で動作するために，スケジューラによる実行タスクの選択などで発生するオーバヘッドを無視できない．従って，ms 単位での周期実行は可能だが，数十 μs 単位の周期実行は困難である．

Simultaneous Multithreading (SMT) アーキテクチャ [7] 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）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 ではスケジュールすることでタスクを実行するため，ms 程度の周期実行は可能であるが，数十 μs 程度の周期実行は困難である．

そこで，本研究では RMT Processor 上で割込み起床機構を用いて，スケジューラから分離することで，より短い周期で実行され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 1 個を専有することで，数十 μs 程度の周期実行を実現する．RMT Processor において Responsive Task の有効性を評価する．既存のスケジューラで管理される Real-Time Task と論理コアを専有する Responsive Task を同時に実行し，両タスクについて起床されてから実行が開始されるまでのオーバヘッドやジッタの測定を行う．

本論文の構成は次のとおりである．まず，2 章では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 の有効性の評価及びその考察を行う．最後に 6 章では結論を述べる．

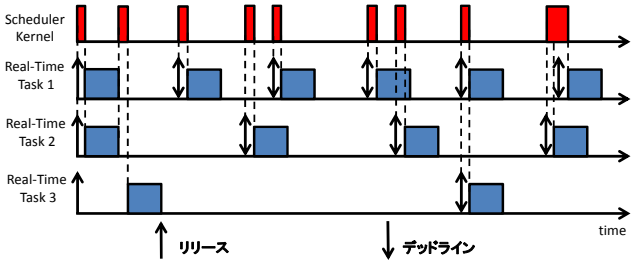


図 1 スケジューラによるタスクの実行
Fig. 1 Task Execution by Scheduler.

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におけるシステムモデルについて述べる．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 C_k を持つ．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，ALU 等のハードウェア資源を優先して獲得することで，リアルタイム処理を実現する．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る．タスクの周期と相対デッドラインは等しいものとする．タスクの利用率は $U_i = C_i/T_i$，論理コア 1 個におけるプロセッサ利用率は $U = \sum_{\tau_i \in C_k} U_i$ 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 $s_{i,j}$，実行終了時刻を $f_{i,j}$ 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 [2]，指定する論理コアにタスクを静的に割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）[3] と Rate Monotonic（RM）[3] がある．これらのようなアルゴリズムを用いてスケジューラがリアルタイム性を満たす範囲で複数のタスクを管理し，システム全体のスループットを向上させる．一方でハードウェア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．特に，モータ制御等のタスクでは小さいジッタで実行されることが望ましい．また，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，スケジューラを介してタスクが実行される場合，スケジューラのオーバヘッドが発生する．さらに，スケジュールするタスクの数が増えるとオーバヘッドが増えるため，ジッタが大きくなる傾向にある．**図 1** にスケジューラによるタスクの実行例を示す．この例では同時に実行可能なスレッド数は 2 個とする．タスクがリリースされてから実行開始するまでにスケジューラによるタスクの選択などの処理を行うため，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．

小さいジッタで短い周期実行を可能とする Linux カーネルのリアルタイム拡張として ART-Linux [11], RT-Preempt Patch [4] 等が挙げられる．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．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 [5] の実装している．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は大きく，数十 μs 単位で周期実行することは困難である．

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 [10] 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[12] 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 [8] を用いることで，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 10μs 以上のスケジューラのオーバヘッドが発生するため，数十 μs 単位での周期実行は困難である．また，RMT Processor の固有機能を最大限に活用する専用リアルタイム OS [9] を独自に開発しているが，RTRON と同様にスケジューラによるオーバヘッドが発生してしまう．

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 [8] について述べる．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，RMT Processor はコンピュータ用 I/O（DDR SDRAM I/F, DMA コントローラ, PCI64 I/F, IEEE1394 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 LSI である．また，RMT Processor における命令セットは MIPS の命令を踏襲しつつ，RMT Processor 固有の命令を持つ．本研究では RMT Processor の 1 つである Dependable RMT Processor I（D-RMTP I）[6] を対象に実装を行う．D-RMTP I の構成を**表 1** に示す．D-RMTP I はスレッドを最大 8 個まで同時実行可能である [*1]．それぞれ 32KByte の命令キャッシュとデータキャッシュ，64KByte の SRAM，64MByte の SDRAM を搭載している．また，D-RMTP I は優先度付き SMT アーキテクチャであり，ALU 等の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になる．

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

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

3.2 スレッド制御機構

まず，RMT Processor 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 [8] に関して説明する．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，プロセッサ内に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 専用のオンチップメモリである．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 クロックが必要であるが，スレッド制御用の専用命令を利用する場合は 4 クロックで実現可能になる．RMT Processor では 32 エントリのコンテキストキャッシュを搭載している．従って，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 2** にこれらのスレッド制御命令による D-RMTP I におけるスレッドの状態遷移図を示す．スレッド毎にコンテキストを持ち，Active Thread RUN 状態のスレッドは実行状

[*1] 本論文では，D-RMTP I におけるソフトウェアが管理するハードウェアコンテキストのことを論理コアと呼ぶ．ただし，D-RMTP I でスレッドを実行する場合は，空き状態のハードウェアコンテキストの中で最も小さいハードウェアコンテキストの ID にスレッドを割当てる．従って，論理コアの ID とハードウェアコンテキストの ID は異なる可能性があることに注意されたい．

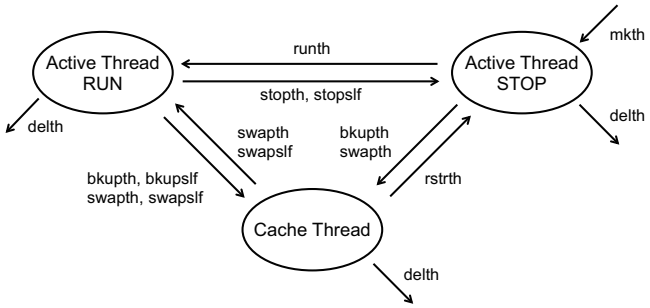


図 2 スレッドの状態遷移
Fig. 2 State Transition of Thread.

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

態にある．Active Thread STOP の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 の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 2 に記載されている命令を発行することで，スレッドの状態を遷移することが可能になる．さらには，割込みによってスレッドの状態を遷移することも可能である．例えば，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 Active Thread RUN 状態に遷移し，割込みベクタで割込み処理を行う．

4. Responsive Task

本研究では，RMT Processor の割込み起床機構を用いて，スケジューラから分離して，非常に短い周期と小さいジッタでのリアルタイム実行を目的とする Responsive Task を提案する．1 個の Responsive Task は D-RMTP I の 8 個ある論理コアの 1 個を専有し，ハードウェアタイマを用いた割込みにより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．D-RMTP I において Responsive Task は最大 8 個まで実行することが可能である．しかしながら，D-RMTP I において 9 個以上のタスクを実行する場合は，Responsive Task では対応できない問題がある．そこでスケジューラで管理される Real-Time Task を同時に実行することで，ハードリアルタイムタスクである Responsive Task とソフトリアルタイムタスクである Real-Time Task を同時に扱うことが可能になる．Responsive Task に Real-Time Task よりも高い優先度を割当てることにより，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．

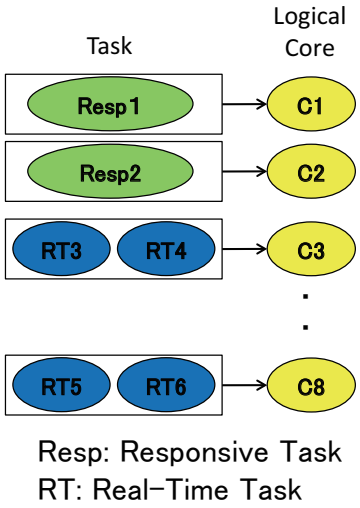


図 3 D-RMTP I における Responsive Task と Real-Time Task の割当て例
Fig. 3 Example of Assginment in Responsive Task and Real-Time Task on D-RMTP I.

図 3 に D-RMTP I における Responsive Task と Real-Time Task の割当て例を示す．論理コア C1 には Responsive Task 1，論理コア C2 には Responsive Task 2 がそれぞれ割当てられる．これに対して，論理コア C3 には Real-Time Task 3 と Real-Time Task 4，論理コア C8 には Real-Time Task 5 と Real-Time Task 6 がそれぞれ割当てられる．Responsive Task は 1 個の論理コアを専有することで低遅延で実行可能になる．これに対して，Real-Time Task は 1 個の論理コアを共有して実行するため，スケジューラにより Real-Time Task を管理する必要がある．

図 4 に 1 個の Responsive Task と 2 個の Real-Time Task を同時に実行する例を示す．Responsive Task 1 は論理コアを専有することにより，スケジューラを呼び出すことなく低遅延で実行可能である．これに対して，Real-Time Task 2 は Real-Time Task 3 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．

4.1 設計及び実装

D-RMTP I において Responsive Task を実現するための割込み起床機構の設計及び実装について述べる．特に，Responsive Task におけるハードリアルタイムタスクと Real-Time Task によるソフトリアルタイムタスクを両方扱うことを目的とした設計及び実装を行う．D-RMTP I は 8 個の論理コアに対してハードウェアタイマを各々持つ．Responsive Task は専有する論理コアのタイマと表 2 に示

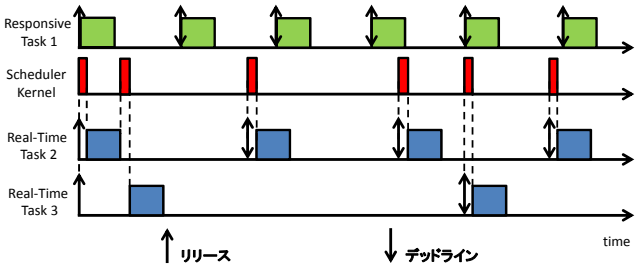


図 4 Responsive Task と Real-Time Task の同時実行
Fig. 4 Simultaneous Execution of Responsive Task and Real-Time Task.

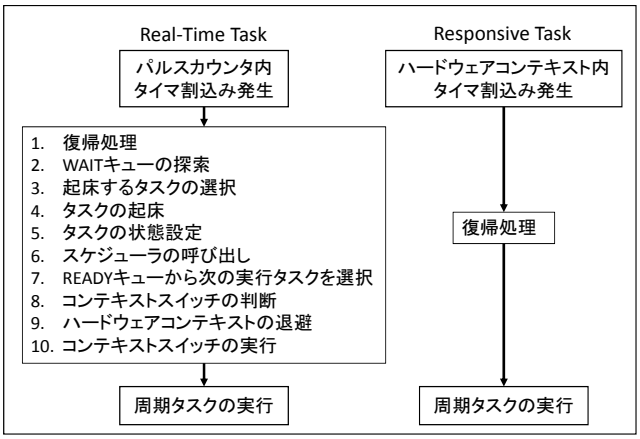


図 5 周期実行における Real-Time Task と Responsive Task の動作
Fig. 5 The Behavior of Real-Time Task and Responsive Task on Periodic Execution.

す stopslf 命令を用い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．Responsive Task は C 言語により実装する．

図 5 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．これに対して，Responsive Task は論理コアを専有するため，これらのキューを必要としない．Real-Time Task は割込みが発生する場合，まずスケジューラが割当てられているスレッドが割込みベクタから復帰し，タスクの WAIT キューを探索して次に起床するタスクを選択する．そして，タスクを RUNNING 状態に設定して READY キューに格納する．スケジューラ本体が呼び出され，READY キューから次に実行するタスクをスケジューラが選択する．選択されたタスクについてコンテキストスイッチが必要かどうかの判定をし，必要であれば実行中のタスクの論理コアの退避を行ってからコンテキストスイッチにより実行タスクを入れ替えて周期タスクの実行に戻る．スケジューラにより実行される Real-Time Task では，タイマ割込み発生後に以上のような処理を実行する．一方で 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タイマ割込み発生時は割込みベクタからの復帰処理を行う．D-RMTP I では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るため，復帰処理のみで周期実行に戻ることができる．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実行を再開できる．本節では Responsive Task を周期実行する際の周期的な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．

タイマ割込み発生時にスケジューリングの処理を行うため，これらの 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る．このようなオーバヘッドを削減するため，D-RMTP I 上に異なる 2 種類の割込み処理を実装する．D-RMTP I では例外及び割込み発生時に，例外及び割込みの種類に応じて登録されたアドレスに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．Real-Time Task は RMT Processor に含まれる制御用 I/O であるパルスカウンタによる外部タイマ，Responsive Task 用の割込みには論理コアが持つ内部タイマを使用する．これらの割込みは上記で説明したオフセットの値が異なり，別の割込みとして処理を行う．この機能により，判定と分岐処理が不要になるため，オーバヘッドを削減することが可能になる．このように設計及び実装を行うことで Real-Time Task と同時に実行可能な Responsive Task を実現する．

4.2 利用例

図 6 に Responsive Task の Real-Time Task の利用例を示す．紙面の都合のため，変数の宣言や初期化処理，エラーチェック等は省略している．create_task_resp 関数は Responsive Task, create_task_rt 関数は Real-Time Task を作成する．両方の関数の引数は同じで以下になる．

- core_id: 論理コア ID
  タスクが実行する D-RMTP I における論理コアの ID を示す．create_task_resp 関数は，指定された論理コアに既に Responsive Task または Real-Time Task が存在する場合は，タスクの作成に失敗し，エラーコードを返す．この理由は，新しく作成した Responsive Task が論理コアを専有できないためである．何もタスクが割当てられていない場合は，タスクの作成に成功する．create_task_rt 関数は，指定された論理コアに既に Responsive Task がある場合はタスクの生成に失敗するが，create_task_resp 関数と同様に何もタスクが割当てられていない場合，または Real-Time Task がある場合はタスクの作成に成功する．
- priority: 優先度
  D-RMTP I に搭載されているタスクの優先度を 0-255

```
int create_task_resp(int core_id,
                     int priority,
                     unsigned long period,
                     unsigned long wcet,
                     void *(*entry)(void *),
                     void *arg);
int create_task_rt(int core_id,
                   int priority,
                   unsigned long period,
                   unsigned long wcet,
                   void *(*entry)(void *),
                   void *arg);

void *entry1(void *arg)
{
  /* execute Responsive Task */
}

void *entry2(void *arg)
{
  /* execute Real-Time Task */
}

void *entry3(void *arg)
{
  /* execute Real-Time Task */
}

int main(void)
{
  create_task_resp(core_id1, priority1,
                   period1, wcet1, entry1, arg1);
  create_task_rt(core_id2, priority2,
                 period2, wcet2, entry2, arg2);
  create_task_rt(core_id2, priority3,
                 period3, wcet3, entry3, arg3);
  return 0;
}
```

図 6 Responsive Task と Real-Time Task の利用例
Fig. 6 Example of Usage of Responsive Task and Real-Time Task.

の 256 段階で設定する．値が大きいほど優先度は高くなり，D-RMTP I における ALU 等のハードウェア資源を優先的に実行することが可能になる．ここで，D-RMTP I に搭載されているタスクの優先度は，EDF や RM 等のスケジューラが管理するタスクの優先度とは異なる．上記以外の値を設定する場合，タスクの作成に失敗し，エラーコードを返す．

- period: 周期 [μs]
  タスクの周期を μs 単位で設定する．0 以下の値を設定した場合はエラーコードを返す．また，create_task_rt 関数では，スケジューラが管理する時間単位である tick の整数倍以外の値を設定した場合はエラーコードを返す．これに対して，create_task_resp 関数では，Responsive Task はスケジューラにより管理しないため，tick の整数倍以外の値でも周期の設定に成功する．
- wcet: 最悪実行時間 [μs]
  タスクの最悪実行時間を μs 単位で設定する．0 以下の値，または周期より大きい値を設定した場合はエラーコードを返す．
- entry: 実行関数のポインタ
  作成するタスクが実行する関数のポインタを示す．関数ポインタの引数と戻り値は両方とも void 型のポインタとする．関数の実装は，UNIX ベースの OS に実装されている共通の API である POSIX [1] の仕様にある pthread_create 関数の第 3 引数を参考にしている．
- arg: entry 関数の引数
  entry 関数の引数を設定する．entry 関数と同様に pthread_create 関数の第 4 引数を参考にしている．

create_task_resp 関数と create_task_rt 関数の戻り値は int 型のタスク ID となる．これらの関数の戻り値が 0 以下の場合はタスクの作成が失敗したことを示す．main 関数では，create_task_resp 関数を 1 回，create_task_rt 関数を 2 回呼び出すことで，それぞれ 1 個の Responsive Task と 2 個の Real-Time Task を作成している．1 個の Responsive Task は論理コア core_id1 を専有して entry1 関数を実行する．2 個の Real-Time Task は指定した同じ論理コア core_id2 において RM や EDF 等によりスケジュールされ，それぞれ entry2 関数及び entry3 関数を実行する．entry1 関数，entry2 関数，entry3 関数の引数には，それぞれ arg1，arg2，arg3 を設定する．

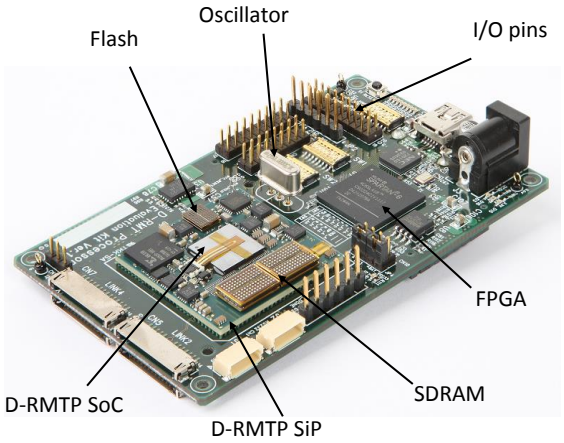


図 7 D-RMTP I 用評価キット
Fig. 7 Evaluation Kit for D-RMTP I.

5. 評価

5.1 評価環境

本研究で提案する Responsive Task の有効性を評価する．

図 7 に 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（System on Chip），SDRAM や Flash Memory 等を集積した SiP（System in Package）を搭載する．D-RMTP I の仕様は図 1 に示されている．SiP の他には各種 I/O 用のピンや FPGA，オシレータなどを搭載しており，この評価キットを用いて実機評価を行う．Responsive Task を実装するリアルタイム OS は，SRAM に充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 上でソフトウェアを実装する．

次に，実機評価の際に用いた各パラメータについて述べる．Real-Time Task は 1 個の論理コアにおいてスケジュールされることで，実行する．Real-Time Task のスケジュールに利用するリアルタイムスケジューリングアルゴリズムは，シングルコア向けの EDF と RM 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．Real-Time Task に割当てられるタスクの実行時間は四則演算タスクが 150μs 程度，配列演算タスクが 300μs 程度である．

各タスクとリアルタイムスケジューリングアルゴリズムの組合せについて，Real-Time Task の数を 0 個から 4 個，Responsive Task を 0 個から 3 個実行する．Real-Time Task は全て周期 5ms でシングルスレッド実行する．Real-Time Task を 4 個同時実行した場合のプロセッサ利用率は四則演算タスクについては 10%程度，配列演算タスクについては 24%程度で設定したため，EDF と RM 共にスケジューリング可能である．Responsive Task は優先度の高い順に 50 μs（High），60μs（Middle），70μs（Low）で実行し，この順番で Responsive Task の数を増やす．Responsive Task に割当てられるタスクの実行時間は四則演算タスクが 1.5μs 程度，配列演算タスクが 3μs 程度である．ジッタが発生しやすい環境で評価を行うために，Responsive Task は異なる周期でそれぞれ実行する．一方で Real-Time Task は，全て同じ周期で実行する．評価時間は 100ms とする．最悪実行時間を想定した環境で評価を取るためにデータキャッシュと命令キャッシュを無効にして実行する．スケジューラの処理中は割込み起床機構とのオーバヘッドの比較評価の公平性のために命令キャッシュを有効にする．

5.2 オーバヘッドの評価

Responsive Task と Real-Time Task について，それぞれタスクが到着してから実行を開始するまでの時間をオーバヘッドとして計測する．Responsive Task はハードウェアタイマのカウンタ値，Real-Time Task はスケジューラのハードウェアタイマのカウンタ値をそれぞれ取得し，タイマ割り込みが発生してから最高優先度の周期タスクに処理が戻るまでのサイクル数を計測した．測定時間 100ms における平均値を評価結果として用いた．計測するオーバヘッドはリアルタイムスケジューリングアルゴリズムやタスクの種類に依存しないため，1 組のリアルタイムスケジューリングアルゴリズムとタスクの種類の組合わせで評価を行う．**表 3** に EDF スケジューラと四則演算タスクを用いて Responsive Task と Real-Time Task をそれぞれシングルスレッド実行した場合の結果を示す．表 3 より，Real-Time Task については実行するタスク数が 1 個の場合でも実行開始までのオーバヘッドは平均して 13μs 程度発生しており，タスク数が増えるにつれてオーバヘッドは大きくなっている．これはタスク数が増えたことによりスケジューラによるオーバヘッドが大きくなったためである．一方 Responsive Task を 1 個のみでシングルスレッド実行した場合のオーバヘッドは 1μs を下回っている．タスク数が 1 個の場合は 13μs 程度のオーバヘッドで実行される Real-Time Task と比較すると，Responsive Task は充分に小さいオーバヘッドで周期実行可能である．

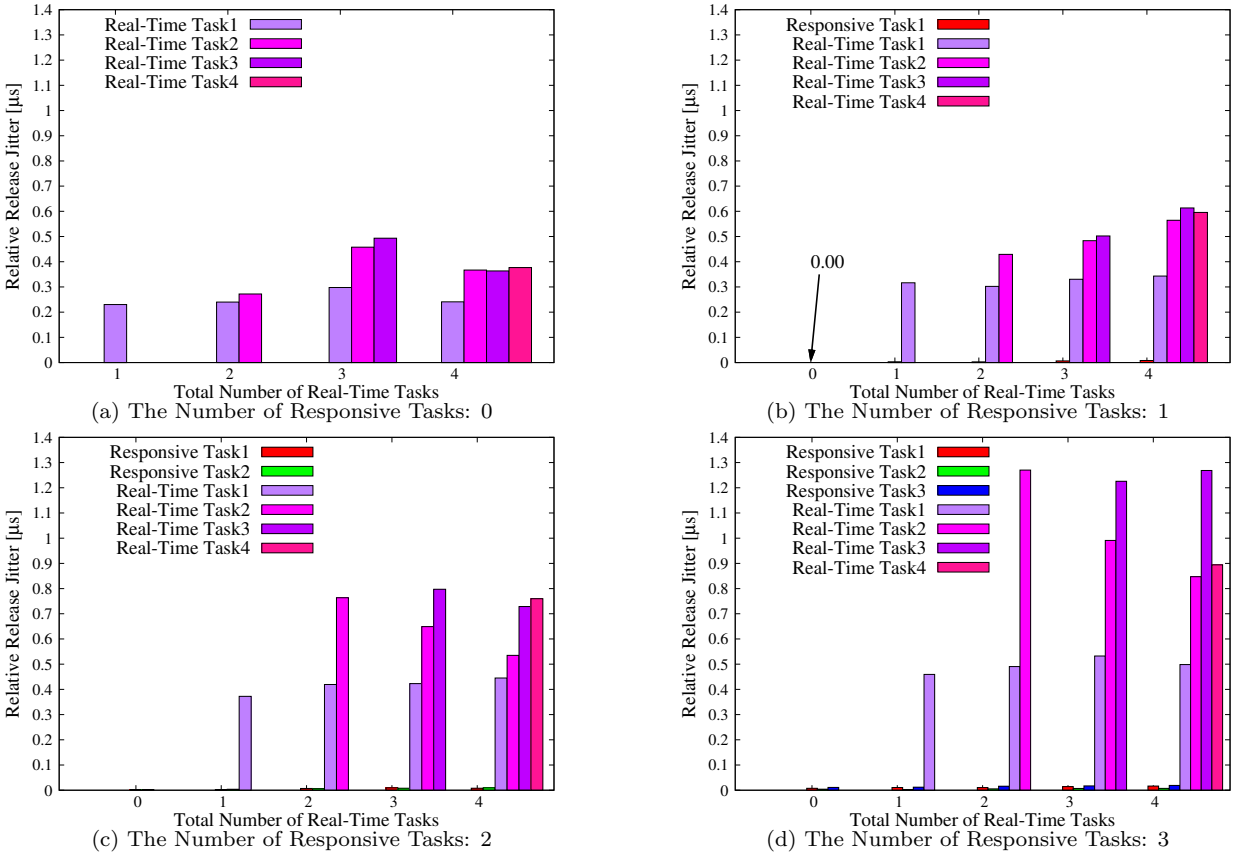
表 3 オーバヘッドの比較
Table 3 The Comparisons of Overhead.

タスク	個数	最小値 [μs]	平均値 [μs]	最大値 [μs]
Responsive	1	0.928	0.928	0.928
Real-Time	1	13.248	13.494	16.832
Real-Time	2	17.856	18.099	21.792
Real-Time	3	25.088	25.375	30.224
Real-Time	4	33.248	33.475	36.736

5.3 ジッタの評価

Responsive Task の数を 0 個から 3 個まで変化させ，それぞれの場合で Real-Time Task の数を最大 4 個まで変化させて各タスクの相対リリースジッタを評価した．オーバヘッドの評価と同様にハードウェアタイマのカウンタ値を用いて評価時間 100ms における相対リリースジッタの計測をした．**図 8**，**図 9**，**図 10**，**図 11** にそれぞれ，EDF 及び RM で四則演算タスクと配列演算タスクを実行した場合の各タスクの相対リリースジッタの評価結果を示す．縦軸は相対リリースジッタの平均値を，横軸はシステムが同時実行する Real-Time Task の数をそれぞれ示す．リアルタイムスケジューリングアルゴリズムとタスクの種類の各組合せにおいて Real-Time Task の数ごとに結果を示す．また **表 4** と **表 5** に EDF と RM におけるタスクの種類において Responsive Task を 3 個，Real-Time Task を 4 個実行した場合についての Responsive Task の相対リリースジッタの最小値，平均値，最大値を示す．

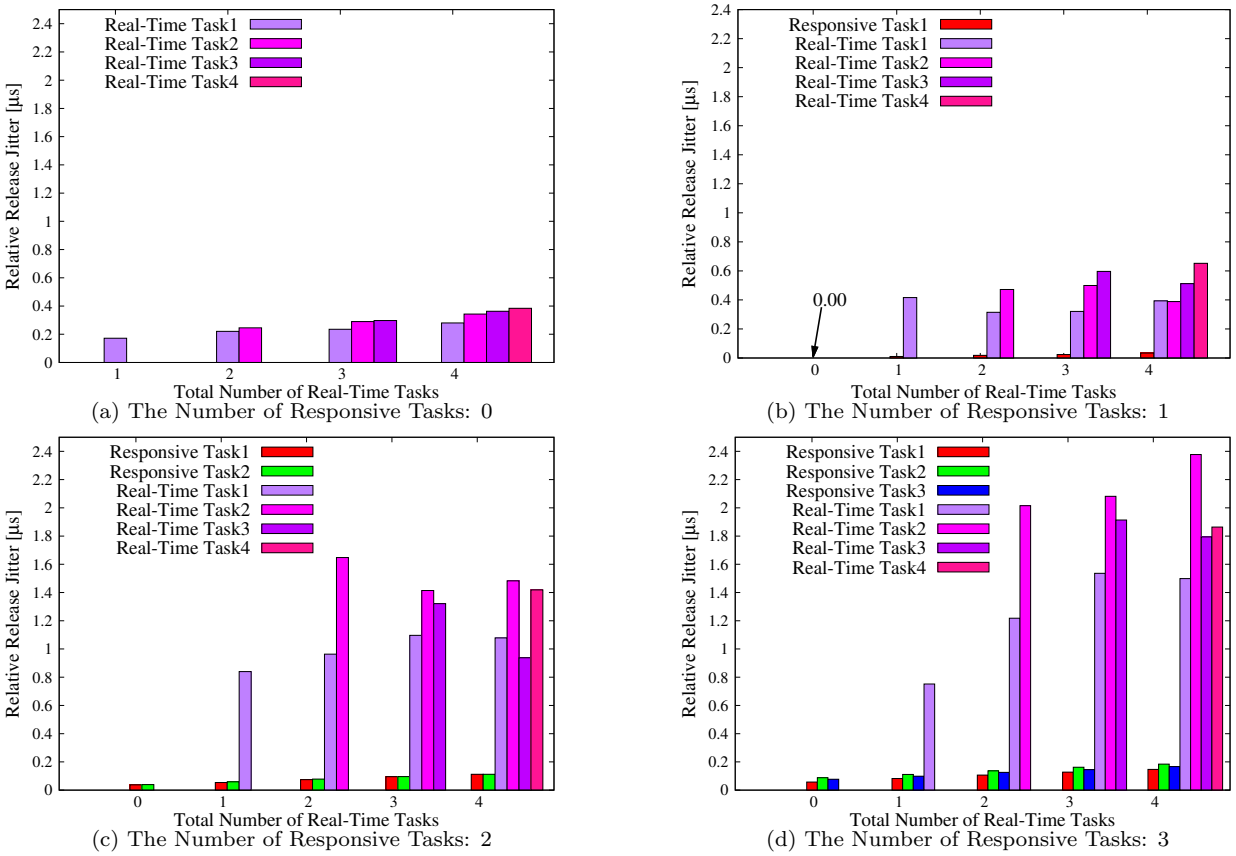
図 8，図 9，図 10，図 11 より，システム内で実行するタスクの数が多くなるほど相対リリースジッタが増加する傾向である．四則演算タスクと配列演算タスクを比較



(a) The Number of Responsive Tasks: 0
(b) The Number of Responsive Tasks: 1
(c) The Number of Responsive Tasks: 2
(d) The Number of Responsive Tasks: 3

図 8 四則演算タスクの平均相対リリースジッタ（EDF）

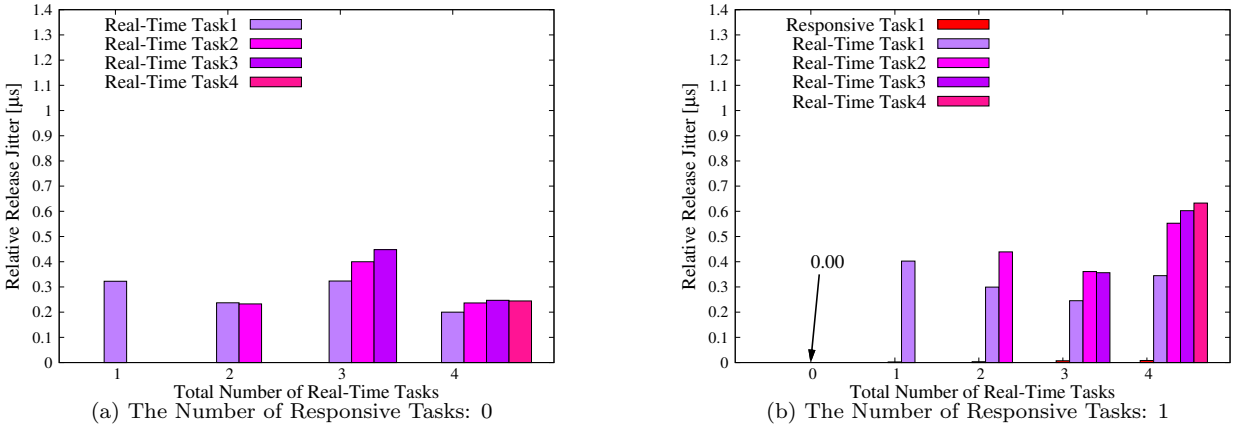
Fig. 8 Average Relative Release Jitter of Task of Four Arithmetic Operations (EDF).



(a) The Number of Responsive Tasks: 0
(b) The Number of Responsive Tasks: 1
(c) The Number of Responsive Tasks: 2
(d) The Number of Responsive Tasks: 3

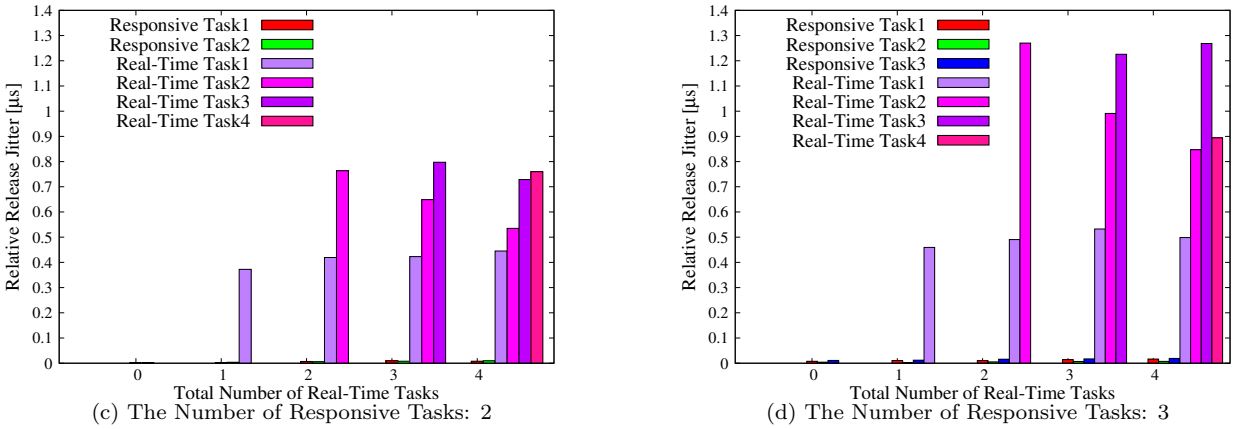
図 9 配列演算タスクの平均相対リリースジッタ（EDF）

Fig. 9 Average Relative Release Jitter of Task of Array Access Operations (EDF).



(a) The Number of Responsive Tasks: 0

(b) The Number of Responsive Tasks: 1

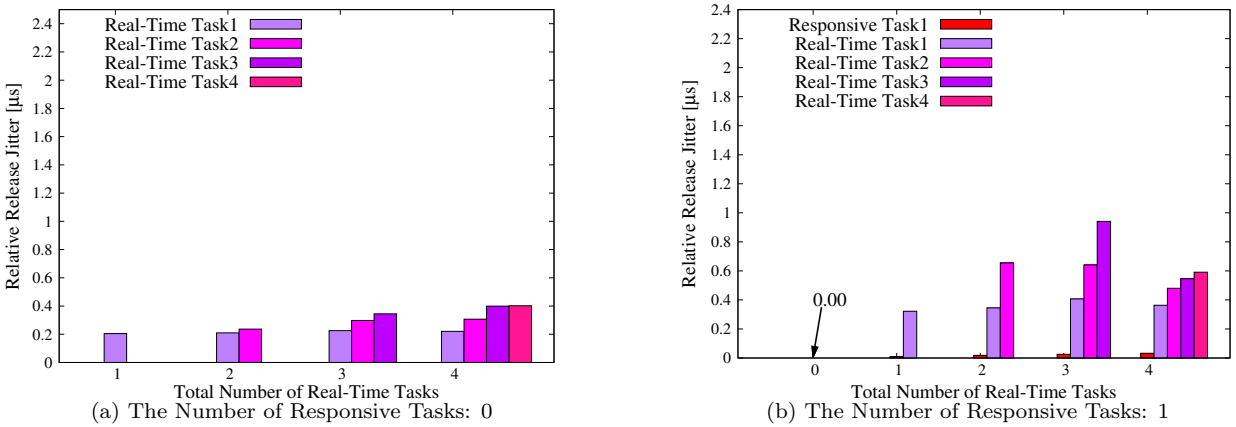


(c) The Number of Responsive Tasks: 2

(d) The Number of Responsive Tasks: 3

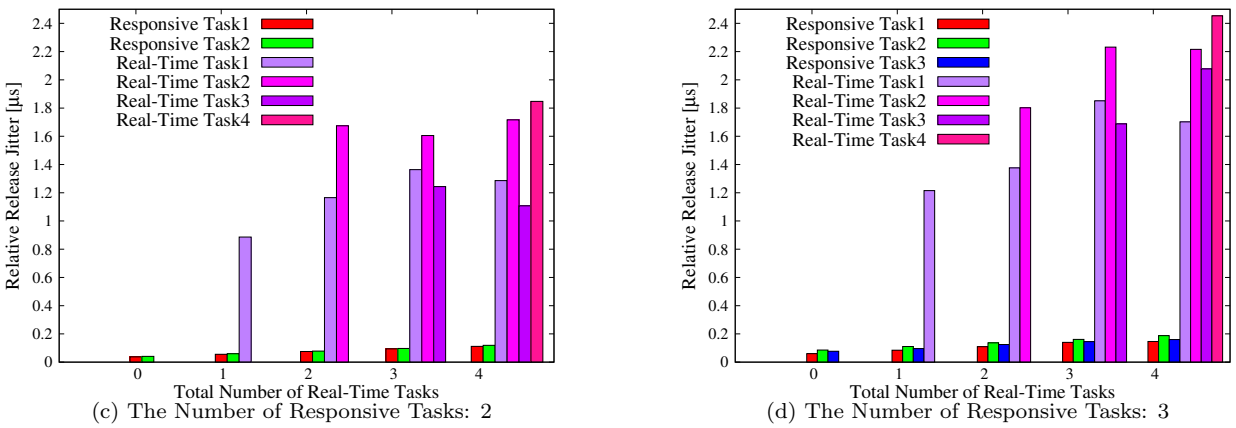
図 10 四則演算タスクの平均相対リリースジッタ（RM）

Fig. 10 Average Relative Release Jitter of Task of Four Arithmetic Operations (RM).



(a) The Number of Responsive Tasks: 0

(b) The Number of Responsive Tasks: 1



(c) The Number of Responsive Tasks: 2

(d) The Number of Responsive Tasks: 3

図 11 配列演算タスクの平均相対リリースジッタ（RM）

Fig. 11 Average Relative Release Jitter of Task of Array Access Operations (RM).

表 4 Responsive Task における相対リリースジッタ（EDF）
Table 4 The Relative Release Jitter of Responsive Task (EDF).

タスク	優先度	最小値 [μs]（%）	平均値 [μs]（%）	最大値 [μs]（%）
四則演算	High	0.000 (0.00)	0.016 (0.03)	0.912 (1.82)
	Middle	0.000 (0.00)	0.007 (0.01)	0.480 (0.80)
	Low	0.000 (0.00)	0.019 (0.03)	0.896 (1.28)
配列演算	High	0.000 (0.00)	0.146 (0.29)	1.472 (2.94)
	Middle	0.000 (0.00)	0.184 (0.31)	1.680 (2.80)
	Low	0.000 (0.00)	0.167 (0.24)	1.424 (2.03)

表 5 Responsive Task における相対リリースジッタ（RM）
Table 5 The Relative Release Jitter of Responsive Task (RM).

タスク	優先度	最小値 [μs]（%）	平均値 [μs]（%）	最大値 [μs]（%）
四則演算	High	0.000 (0.00)	0.020 (0.04)	0.992 (1.98)
	Middle	0.000 (0.00)	0.001 (0.00)	0.816 (1.36)
	Low	0.000 (0.00)	0.022 (0.03)	0.880 (1.26)
配列演算	High	0.000 (0.00)	0.146 (0.29)	1.470 (2.94)
	Middle	0.000 (0.00)	0.187 (0.31)	1.616 (2.69)
	Low	0.000 (0.00)	0.159 (0.23)	1.968 (2.81)

した場合，後者のほうが相対リリースジッタが大きいが，これは表 1 で示したように ALU よりもメモリアクセスユニット資源が少なく，競合が発生しやすいためである．Responsive Task を複数実行した場合にジッタの大きさと設定した優先度に規則性がない．これは図 5 に示したように，Responsive Task は割込みが発生してから周期タスクに戻るまでに復帰命令しか実行しないため，このオーバヘッドに関しては優先度制御の効果は低いと考えられる．図 8，図 9，図 10，図 11 のそれぞれの (b) のグラフにおいて，Responsive Task を 1 個だけ実行すると相対リリースジッタは必ず 0.00 となり，ジッタの発生を抑制することが可能になる．表 4 と表 5 より，それぞれのリアルタイムスケジューリングアルゴリズムとタスクの種類の組合せにおいて，四則演算タスクでは最小 0.00%，平均 0.04%，最大 1.8%と，配列演算タスクでは最小 0.00%，平均 0.4%，最大 3.2%になった．また，今回の評価の全組合せにおいて Responsive Task の相対リリースジッタが四則演算タスクの場合は平均 0.01μs，最大 0.992μs，配列演算タスクの場合は平均 0.15μs，最大 1.968 μs になった．これらの結果から Responsive Task は数十 μs 単位の短い周期に対して時間制約を満たしつつ充分に小さいジッタで実行可能である．

6. 結論

本研究では D-RMTP I を用いて割込み起床機構によって実現されるハードリアルタイムタスクである Responsive Task を実装した．実機評価により，62.5MHz の動作周波数において相対リリースジッタが四則演算タスクの場合は最大 0.992μs，配列演算タスクの場合は最大 1.968μs であり，数十 μs 単位の短い周期に対して充分に小さいジッタであることを示した．また Responsive Task の実行開始処理にかかるオーバヘッドは約 1μs と充分に小さいことを示した．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 程度の周期実行を実現することが可能になった．

今後の課題としては，ソフトリアルタイムタスクである Real-Time Task が，ハードリアルタイムタスクである Responsive Task のオーバヘッドやジッタに関して，どの程度の影響を与えるのかを詳細に調査する予定である．また，モータ制御や通信のような I/O 処理向けに Responsive Task を適用する予定である．

謝辞 本研究の一部は科学技術振興機構 CREST の支援に依るものであることを記し，謝意を表す．また，本研究の一部は慶應義塾 学事振興資金と一般財団法人 慶應工学会の支援に依るものであることを記し，謝意を表す．本研究に関して有益な助言を頂いた慶應義塾大学 大学院理工学研究科 大沢幸平様に感謝する．

参考文献

[1] The IEEE and The Open Group: *IEEE Std 1003.1*, 2013 edition (2013).
[2] Kumura, Y., Mizotani, K., Takasu, M., Chishiro, H. and Yamasaki., N.: Overhead-Aware Schedulability Analysis on Responsive Multithreaded Processor, *Proceedings of the 30th International Conference on Computers and Their Applications*, pp. 379–386 (2015).
[3] Liu, C. L. and Layland, J. W.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, *Journal of the ACM*, Vol. 20, No. 1, pp. 46–61 (1973).
[4] 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).
[5] Sha, L., Rajkumar, R. and Lehoczky, J. P.: Priority inheritance protocols: An approach to real-time synchronization, *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1175–1185 (1990).
[6] 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).
[7] Tullsen, D. M., Eggers, S. J. and Levy, H. M.: Simultaneous Multithreading: Maximizing On-Chip Parallelism, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 392–403 (1995).
[8] Yamasaki, N.: Responsive Multithreaded Processor for Distributed Real-Time Systems, *Journal of Robotics and Mechatronics*, Vol. 17, No. 2, pp. 130–141 (2005).
[9] 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).
[10] 高田広章：μITRON4.0 仕様 Ver4.02.00，トロン協会 (2004).
[11] 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).
[12] 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).