Vol.2013-OS-126 No.18
2013/8/1

# CPU affinity による汎用 OS のリアルタイム性向上手法

山田真大[†1, †2]　林和宏[†1]　鈴木章浩[†1]　岡本幸太[†1]　小林良岳[†1]
本田晋也[†2]　高田広章[†2]

組込み向け機器に利用されるハードウェアの高性能化に伴い，組込み OS として Linux などの汎用 OS が搭載されるようになった．組込み機器では，リアルタイム性が重要視されるため，Linux を採用する場合，カーネルに改良を施すことでリアルタイム性を確保している. また, マルチコア CPU を搭載する組込み機器では, Linux が持つ CPU affinity の機能を用いることで，シングルコアでは不可能であった高負荷時におけるリアルタイム性も確保することが可能になった．しかし，CPU コア毎に存在するカーネルスレッドは CPU affinity を適用することができず，また，この処理がまれに引き起こすタイマのカスケード処理には多くの処理時間を必要とし，リアルタイム性を阻害する原因となる．本論文では，マルチコア CPU の各コアを，リアルタイム性を必要とする CPU コアと不要とする CPU コアに分割し，リアルタイム性を必要とする CPU コアでは，タイマのカスケード処理を発生させないよう事前に対策を施すことで，リアルタイム性を確保する手法を提案する．

# Improving the real-time performance of a general-purpose OS through CPU affinity

MASAHIRO YAMADA[†1,†2]　KAZUHIRO HAYASHI[†1] AKIHIRO SUZUKI[†1]
KOTA OKAMOTO[†1] YOSHITAKE KOBAYASHI[†1] SHINYA HONDA[†2]
HIROAKI TAKADA[†2]

With the increase in hardware performance of modern embedded systems, general-purpose operating systems (OS) such as Linux are commonly used as embedded OSs. Furthermore, the use of multi-core CPUs enables Linux to improve its real-time performance even on high-load scenarios which is rather hard to achieve on single-core CPUs thanks to its "CPU affinity" functionality. However, we found two issues in the current version of the Linux kernel: the CPU affinity of some kernel threads cannot be specified; and the use of timer cascading (use of multiple hardware timers to count time) increases the worst-case response time of real-time tasks. In this paper, we classify the cores in a multi-core CPU into 2 different groups: cores which require real-time performance guarantees; and cores which do not require such guarantees. Then, we propose and evaluate a method that improves the real-time performance of the system by disabling timer cascading on cores which require real-time performance guarantees.

## 1. はじめに

組込み機器の高性能化に伴い， Linux などの汎用 OS が搭載されるようになってきた．組込み機器は，リアルタイム性が重要視されるため，汎用 OS を利用する場合は，改良を施すことによってリアルタイム性を確保している．

Linux カーネルを使いつつリアルタイム性を確保する手段としては，RTOS と汎用 OS の共存によるものと，Linux 単体でのリアルタイム性確保によるものの２つがある．まず、RTOS と汎用 OS の共存によるものとしては，RTOS の 1 タスクとして汎用 OS を用いる方法[1]，RTOS と汎用 OS を用いてハイブリッド OS 構成とする方法，仮想化を用いる方法が存在する[2]，[3]．これらは高いリアルタイム性を確保できるといったメリットがある一方，開発環境の違いや問題が発生した際の原因特定の難しさといったデメリットも存在する．次に、単一の Linux カーネルでリアルタイム性を確保する手段には，Linux カーネルに特殊なインタフェースを持たせる RTAI[4]や，Linux カーネルの割込みやプリエンプション禁止区間を小さくする Real-Time Preemption Patch[5]がある．

プログラムを作成する観点では異なる OS や API の仕様は複雑性が増すため，単一の OS で実現したいという要求がある．そこで、以降は単純な構成で実現可能な Linux カーネルのみによるリアルタイム性を確保の手段に着目する．

Linux カーネルのみでリアルタイム性を確保するためには，ハードウェアの構成要素にも着目する必要がある．マルチコア CPU を搭載する組込み機器では、Linux カーネルが持つ CPU affinity の機能を用いて，マルチコア CPU の各 CPU コアを, リアルタイム性を必要とする CPU コア(以下, RT 用 CPU コア)と不要とする CPU コア(以下, 汎用 CPU コア)に分割して割り当てる手法が用いられる．これにより，リアルタイム性の確保が不要となる汎用 CPU コア側で高負荷な処理が発生しても, リアルタイム性を必要とする RT 用 CPU コアでは，リアルタイム性を確保できる[6].

しかし，Linux において，CPU コア毎に存在するカーネルスレッドは，CPU コアへの割り当てが固定になっており，CPU affinity を適用することができない. この処理にはリア

†1 (株)東芝
Toshiba Corporation.
†2 名古屋大学
Nagoya University

ルタイム性を確保する観点で重要となるタイマ処理が含まれている．そしてタイマ処理がまれに引き起こすカスケード処理は，多くの処理時間を必要とし，リアルタイム性を阻害する原因となる．

本論文では，複数の CPU コアに対し，CPU affinity を用いて，RT 用 CPU コアと汎用 CPU コアに分割し，さらに，RT 用 CPU コアでは，カスケード処理を発生させないよう事前に対策を施すことで，リアルタイム性を確保する手法を提案する．

2 章では，関連研究について述べる．3 章では，本論文で満たすべき要件について述べ，そのための課題を示す．4 章では，提案手法と実装方法について述べ，5 章では，その評価結果を示す．6 章では，本論文のまとめについて述べる．

## 2. 関連研究

本論文は，単一の Linux カーネルのリアルタイム性を向上させ保障するために，マルチコア CPU を RT 用 CPU コアと汎用 CPU コアに分類し，CPU affinity 機能を用いて，RT 用 CPU コアにおいてリアルタイム性を保障する手法についての研究である．以前よりこのような手法は研究されており，以下に 2 つの関連研究を紹介する．

### 2.1 Shielded CPU

Steve らは, Symmetric Multi Processor(以下 SMP)を前提として Shielded CPU という概念を導入した[6]．Shielded CPU に割り当てることができるのは，リアルタイム性を要求する種類のタスクや割り込みだけに限定されている．その結果，Shielded CPU で動作するタスクが，予測可能な実行時間や割込み応答時間になることを示した．実験によって，高負荷なネットワーク処理やグラフィクス処理を実行中であっても，リアルタイムタスク（以下 RT タスク）の割込み応答時間が 30ms 以下に抑えられることを示した．

### 2.2 ARTiS

ARTiS は，Linux カーネルのリアルタイム性の強化を目的としたプロジェクトである．ARTiS において Eric らは，SMP の各コアに RT CPU か NRT CPU という属性を割り当てて, 各 CPU コアを 2 種類に分類することで非対称なプロセッサとして扱い, RT CPU で動作する RT タスクの割込みの応答を低遅延にできることを述べている[7]．実験によって負荷が与えられている状態でも 104us 以下の遅延時間となることを示した．

## 3. 課題と要件

本章では，組込み機器に必要なリアルタイム性を確保，向上させる手法について述べるとともに，それらの手法を用いても発生する問題点について言及する．そして，問題解決のために満たすべき課題を列挙する．なお，以降の議論を簡略化するため，2 つの CPU コアを持ったマルチコア環境を前提とする．

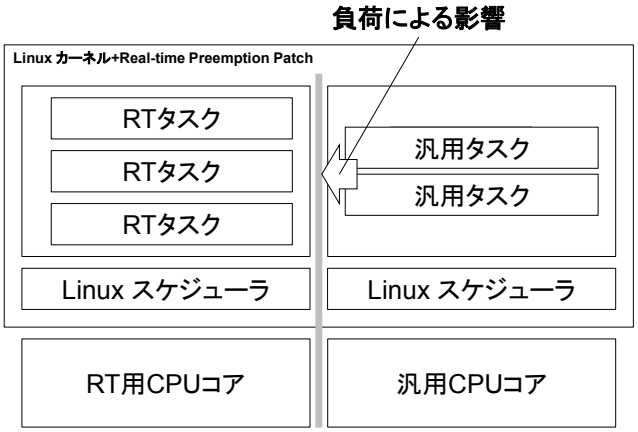


図 1　Real-time Preemption Patch と CPU affinity によってリアルタイム性を向上させた構成

### 3.1 Linux のリアルタイム性の向上方法

Linux においてリアルタイム性を確保する場合，一般に Real-Time Preemption Patch を用いる．このパッチは，Linux カーネルの中で発生する割込み禁止区間やプリエンプション禁止区間を，従来よりも細かく分割することで，割込み応答性能を大幅に改善できるものである[5].

しかし，Real-Time Preemption Patch を用いても，汎用タスクと RT タスクを共存させると，汎用タスクが発生させる負荷によって, RT タスクのリアルタイム性（応答開始時間）が影響を受けることがある．応答開始時間が遅れる要因は，汎用タスクが頻繁にスケジュールされたり，IO を発行後に割込み禁止区間が長くなる動作が行われたりすることが挙げられる．この負荷から保護するために，マルチコア環境におけるタスクの CPU affinity 機能を用いる．タスクは一般にマルチコア環境においてどの CPU コアで動作するかは定められていないが，この機能を用いるとタスクが動作する CPU コアを限定することができる. そこですべてのコアを，汎用 CPU コア，または RT 用 CPU コアの 2 種類に分割し, 汎用タスクの負荷は汎用 CPU コアに割り当てる．これにより，汎用タスクの負荷の影響を RT タスクが受けなくなるため，RT 用 CPU コアでのリアルタイム性が向上する．

本論文では，図 1 に示すように，Linux カーネルに Real-Time Preemption Patch を適用することで割込み遅延時間を低減し，さらに CPU affinity を用いることで，汎用タスクが引き起こす負荷が, RT タスクに影響することを防ぎ,

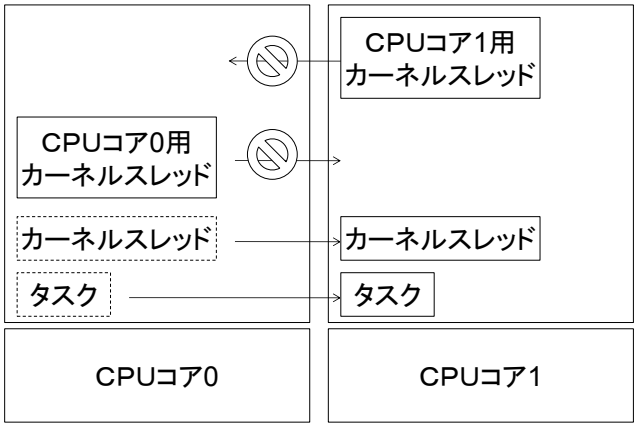


図 2 CPU affinity によるタスク／カーネルスレッドの移動

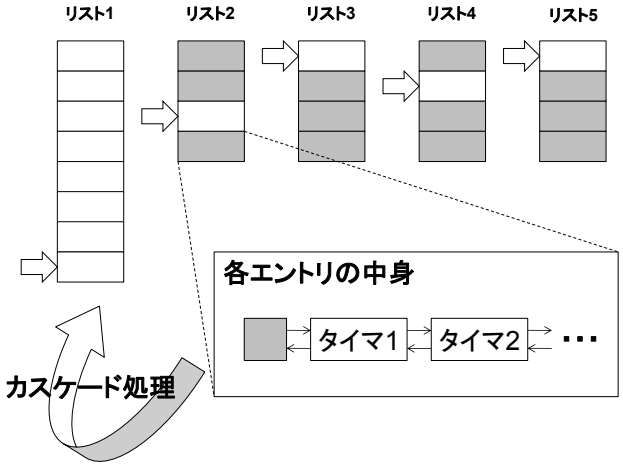


図 3　タイマのリスト管理とカスケード処理

RT 用 CPU コア上で動作するリアルタイム性を保障する．このような環境を構築するため，３つの状態を定義する．

- 初期状態 : Linux カーネルを起動した直後の状態．存在しているすべてのプロセスは，コア専用カーネルスレッドを除いて，CPU affinity は設定されていない．
- 移動状態 : CPU affinity を用いて，汎用タスクを汎用 CPU コアへ割り当て，RT タスクを RT 用 CPU コアへ割り当て，タスクがコア間を移動している状態．
- 稼働状態：目的とするコアへのタスク移動がすべて完了した状態．この状態において，RT 用 CPU コアでは，リアルタイム性が保障されている．

初期状態から，CPU affinity を用いてタスクを移動させ（移動状態），すべてのタスクマイグレーションが完了した時点で稼働状態となり目的とする状態に遷移する．

### 3.2 CPU affinity 機能の問題点

ユーザが作成したタスクやカーネルの機能を担うカーネルスレッドは，CPU affinity を自由に設定して動作する CPU コアを移動させることができる．しかし，図 2 に示すように CPU コア毎に固定されたカーネルスレッド(以下コア専用カーネルスレッド)は，CPU affinity を自由に設定することはできず，特定の CPU コアに固定されており，他の CPU コアでは動作することができない．このようなカーネルスレッドは，以下のような処理が存在する．

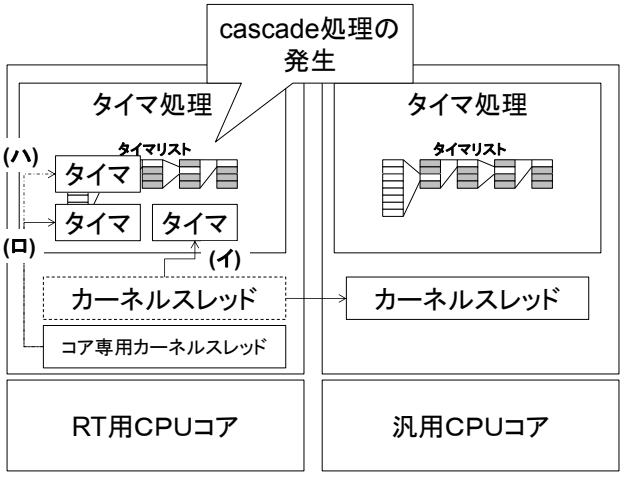


図 4　RT 用 CPU コアにおけるタイマ処理の問題点

- ドライバの遅延処理やポーリング処理を行うもの
- 高精度タイマ処理
- タイマ処理
- 割込み遅延処理
- タスクマイグレーション処理

コア専用カーネルスレッドの一つであるタイマ処理は，タスクやカーネルスレッドから登録されるタイマを CPU 毎にリストとして管理している．図 3 にその概要を示す．この図では，リストを 5 つ持ち，それぞれのリストの各エントリでタイマを管理することができる．各エントリは，タイマの発火時刻を示しており，このエントリに登録されているタイマは，発火時刻を過ぎると実行される．各リストはそれぞれ異なるタイムスケールを持ち，リスト 1 のエントリは 1ms 毎，リスト 2 のエントリは 256ms 毎というように異なっている．リスト 1 の最後のエントリまで参照を終えた場合，リスト 2 が参照しているエントリの次のエントリに登録されているすべてのタイマをリスト 1 に登録しなおすという処理が発生する．リスト 2 が最後のエントリまで参照を終えた場合も同様にリスト 3 の次のエントリに登録されているタイマをすべてリスト 2 へ登録しなおすという処理が発生する．この一連の処理をカスケード処理と呼ぶ[8]．カスケード処理は以下の理由からリアルアイム性を阻害する原因となる．

- 割込み禁止状態でタイマのリスト探索と付け替え処理を行う
- タイマの数に上限はないため，カスケード処理にかかる時間の上限が予測できない
- NO_HZ[9]では，定期的なカスケード処理が発生しな

いため，以前に行ったタイマ処理から間を空けてタイマ処理が発生した場合，その間に行われることがなかった分のカスケード処理が発生する

これらの理由から，RT 用 CPU コアで，カスケード処理が発生するということは好ましくない．カスケード処理は，セットされたタイマが発火したときを契機として実行される．そのため，一つ以上のタイマが RT 用 CPU コアのタイマのリストに登録されていれば，カスケード処理が発生する可能性がある．そこで，RT 用 CPU コアでは，タイマのリストにタイマが登録されない状態を維持する必要がある．

### 3.3 課題

一般にタイマは，そのタイマをセットするタスクが動作している CPU コアのリストに登録される．例外として，省電力のために，CPU コアがアイドル状態に入る予定があれば，別の CPU コアのリストに登録することもあるが，今回の件では適用しないため除外する．CPU affinity 機能によって汎用タスクやカーネルスレッドを汎用 CPU コアへ移動させた場合，これらが登録するタイマは，CPU コアを移動した後であれば，汎用 CPU コアに登録される．

しかし，図 4 に示したように RT 用 CPU コアのリストにタイマがセットされ，発火するケースとして以下の 3 点がある．

イ) 初期状態において RT 用 CPU コアで動作していたカーネルスレッドや汎用タスクがタイマをセットした後に，移動状態によって CPU affinity によって汎用 CPU コアへ移動した場合

ロ) 稼働状態において RT 用 CPU コアで動作するコア専用カーネルスレッドがタイマをセットする場合

ハ) 初期状態において RT 用 CPU コアで動作するコア専用カーネルスレッドがタイマをセットしていた場合

これらのどのケースにおいても，稼働状態でタイマが発火した時点でカスケード処理が発生する可能性があるため，(イ)〜(ハ)のケースにおいてタイマが発火する状態を防ぐ必要がある．

## 4. 設計・実装

3.3 節で提示した 3 つの課題(イ)〜(ハ)に対し，カスケード処理を未然に防ぐための手法を提案する．(イ)に対しての対策を 4.1 節，(ロ)に対しての対策を 4.2 節，(ハ)に対しての対策を 4.3 節で示す．これらの対策によって，RT 用 CPU コアのタイマリストが常に空の状態となり，タイマ処理が実行されることがなくなるため(full tickless 状態)，カスケード処理が発生することがなくリアルタイム性を確保できる．

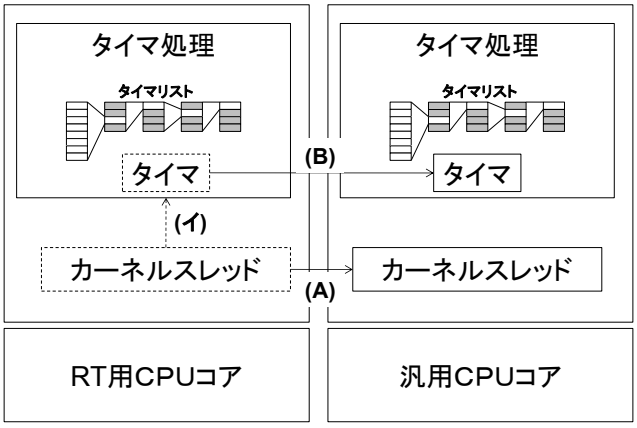


図 5　タイマのマイグレーション

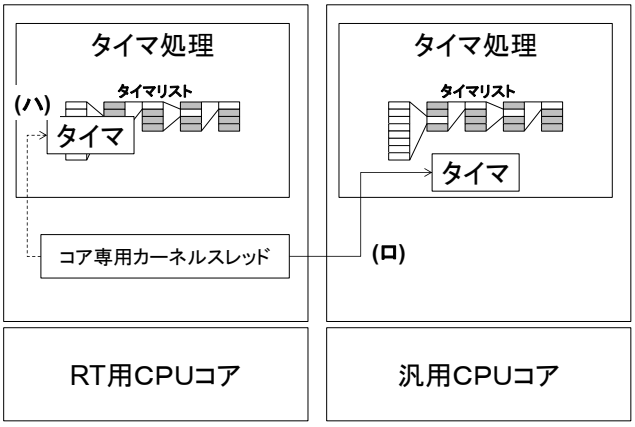


図 6　タイマの登録先を汎用 CPU コアへ限定

### 4.1 CPU affinity と連動したタイマのマイグレーション

RT 用 CPU コアは，汎用 CPU コア上で動作するタスクのためのタイマを発火させる必要がないため，そのようなタイマは，汎用 CPU コア上で発火するようにしたい．図 5 に示すように，移動状態において，CPU affinity 機能を用いてカーネルスレッドや汎用タスクが動作する CPU コアを汎用 CPU コアに決定して，(A)で示すようにタスクマイグレーションを実行した際，それらタスクが RT 用 CPU コアのタイマリストに登録していたタイマも，(B)のように汎用 CPU コアのタイマのリストへマイグレーションさせる機能を実装する．この機能によって，汎用 CPU コアで動作するタスクやカーネルスレッドのタイマが RT 用 CPU コアのタイマのリストに残ったままとなる状況がなくなり，不要なタイマが RT 用 CPU コアで発火することを防ぐ．

### 4.2 タイマ登録先を汎用 CPU コア上のリストに限定

図 6 に示すように，RT 用 CPU コアで動作しているコア専用カーネルスレッドが起床するために用いるタイマをセットする際，そのタイマの登録先を RT 用 CPU コアではなく，汎用 CPU コアの上で動作するタイマのリストにするこ

表 1 評価環境

| | |
|---|---|
| Board | Pandaboard ES |
| CPU | ARM Cortex A9 MPCore(1.2GHz) |
| OS version | Linux kernel 3.6.11-rt37 |
| Root file system | Busybox |

表 2 カーネルコンフィグ

| コンフィグ名 | 適用の有無 (Yes or No) |
|---|---|
| CONFIG_NO_HZ | Yes |
| CONFIG_HRTIMER | Yes |
| CONFIG_PREEMPT_RT_FULL | Yes |
| CONFIG_OMAP_32K_TIMER | No |

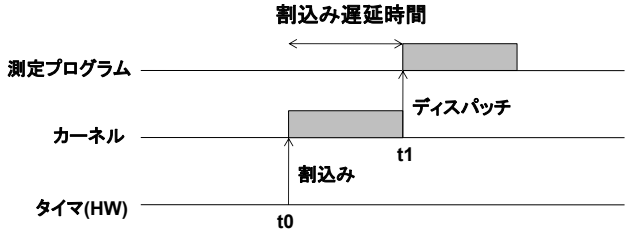


図 7 測定プログラム

とで，RT 用 CPU コア上でタイマが発火することを防ぐ．これは，稼働状態におけるタイマ登録関数の振る舞いを修正し，その登録先のリストを指定された RT 用 CPU コア以外から選択するように実装することで実現する．

### 4.3 RT 用 CPU コアのタイマリスト状態の確認

移動状態が完了して稼働状態になる前に，4.1 節によって汎用タスクがセットしたタイマを，すべて汎用 CPU コアのリストへ移動する．しかしタイマの中には RT 用 CPU コアで動作するコア専用カーネルスレッドが，登録していたタイマが存在する可能性もある．そこで，コア専用カーネルスレッドが登録したタイマが，リストに登録されていないかどうかを調べることでこの状態を確認する．タイマが存在している場合は，リアルタイム性を保障することができないため，リスト上のタイマがすべて発火し終わるまで待つようにし，すべて発火がおわった時点で，稼働状態とする．タイマが登録されていた場合，次のタイマの発火時刻を返し，そうでない場合は，0 を返す API を Linux カーネルの debugfs インタフェースとして実装し，ユーザプログラムへ提供することで，この機能の実現を行う．

## 5. 評価

### 5.1 評価環境

評価のために構築した環境を表 1 に示す．CPU は 2 コアあり，一方を RT 用 CPU コア，他方を汎用 CPU コアとして利用する．また，Linux のカーネルコンフィグレーションとして，主に表 2 に記載した内容を適用する．カスケード処理による影響をわかりやすくするために，NO_HZ を適用した．HRTIMER は高精度タイマを有効にするもので，測定プログラムが正確に起床するために必要である．PREEMPT_RT_FULL は，Real-Time Preemption Patch の機能を有効にするためのコンフィグである．また，OMAP アーキテクチャのカーネルコンフィグで，デフォルトで適用される 32K TIMER は，リアルタイム性を十分に確保できないため適用外とした．

### 5.2 評価方法

リアルタイム性を評価する方法として，割込み遅延時間を測定する．割込み遅延時間とは，図 7 に示すようにある時刻 t0 で発生した割込みから，カーネルのオーバヘッドやカーネルスレッドの処理を得て，測定プログラムが時刻 t1 に起床した時，(t1-t0)の時間を指す．測定プログラムを周期的に 20 万回繰り返し計測する．周期は 300μs とし，割込み遅延時間を計測する測定プログラム(RT タスク)は静的優先度を 98 とし，割込み遅延処理やタイマ処理を集約して行うコア専用カーネルスレッド ksoftirqd は静的優先度を 99 として，それぞれ SCHED_FIFO（静的優先度順スケジューリング）によって実行を行う．静的優先度は，値が大きいほど優先度が高く，今回のケースでは ksoftirqd が割込み遅延時間を計測する測定プログラムよりも優先度が高い．これは，割込み遅延時間を計測する測定プログラムが起床するために高精度タイマを用いており，ksoftirqd を先に実行しなければ測定プログラムの起床が遅れるためである．割込み遅延時間を計測する測定プログラム以外は，デフォルトの状態としており，ユーザが作成したタスクは存在しないが，カーネルスレッドは Linux カーネルの起動処理において，必要な分生成されている．割込み遅延時間の測定は，以下に挙げる 4 つのケースで行う．

I. CPU affinity を使わない．割込み遅延時間を測定する RT タスクを含むすべての処理が，どの CPU コアで動作するか保障はされていない．
II. CPU affinity を使って，割込み遅延時間を測定する RT タスクは RT 用 CPU コアに固定する．CPU コア専用のカーネルスレッド以外は，すべて汎用 CPU コアに固定する．ただしカスケード処理は発生しなかった場合の評価結果．
III. II と同様だが，カスケード処理が発生した場合の評価結果（発生した時点で測定を終了させる）．
IV. III と同様だが，本稿で開発した機能によって RT 用 CPU コアではカスケード処理が発生しなかった評価結果．

また，III と IV の違いを見るためにカーネル内部で発生

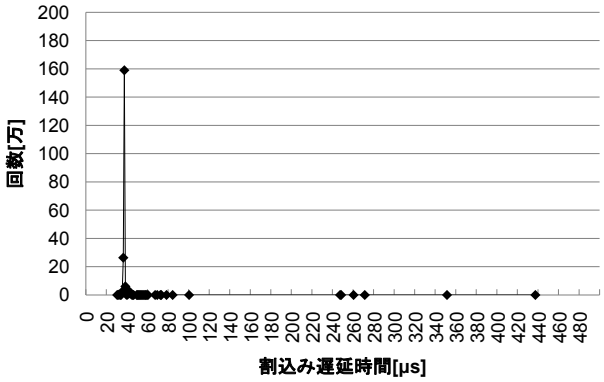


図 8　ケース I の割込み遅延時間

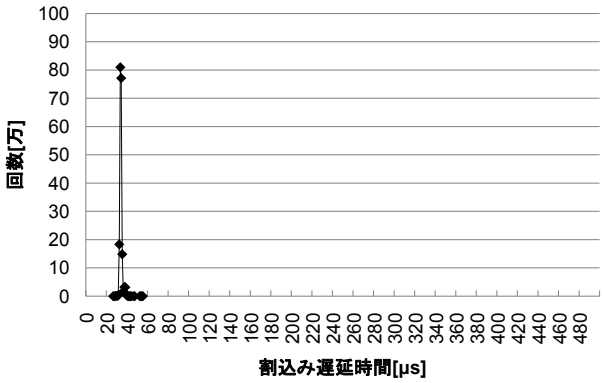


図 9　ケース II の割込み遅延時間

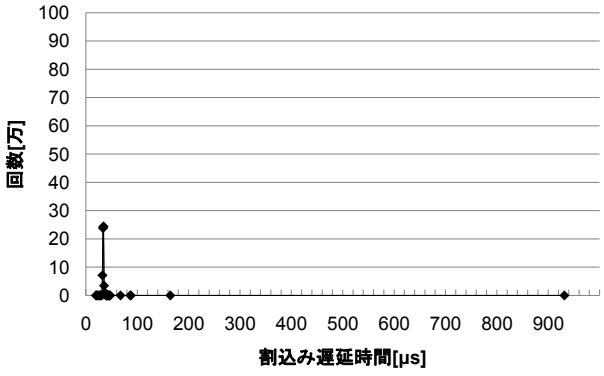


図 10　ケース III の割込み遅延時間

している処理内容をカーネルのトレース機能に使ってログを取得して示す．

### 5.3 カスケード処理の発生方法

今回の計測において，ケース III，ケース IV を評価するためにワークキュー[10]を用いる．ワークキューは，ドライバの遅延処理やポーリング処理を行うためのものである．また，ワークキューは，コア専用カーネルスレッドであり，各 CPU コアで動作する．ワークキューが遅延処理を行う場

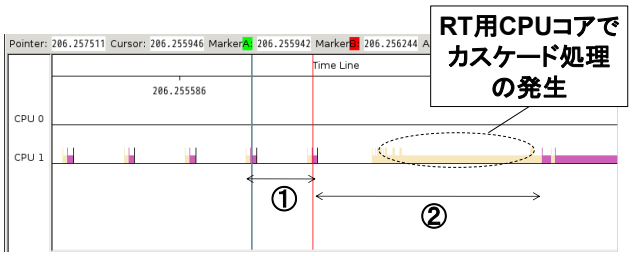


図 11　ケース III におけるトレースログ

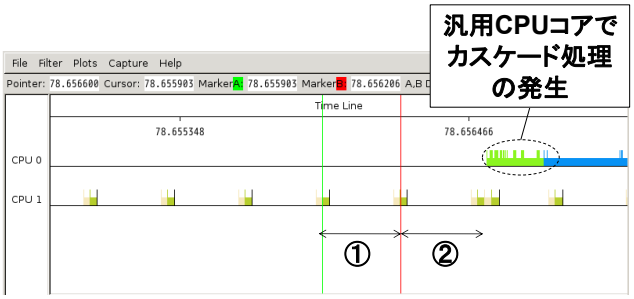


図 12　ケース IV におけるトレースログ

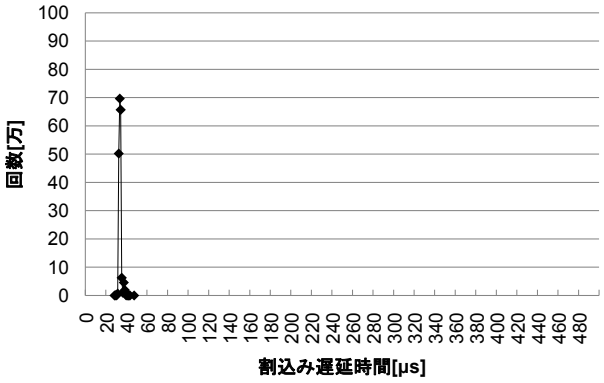


図 13 ケース IV の割込み遅延時間

合，遅延時間を指定したタイマを登録する．遅延時間が経過した後，発火したタイマがワークキューの所有するキューへ遅延処理をキューイングする．最後に，コア専用カーネルスレッドとして動作しているワークキューがキューイングされた遅延処理を実行する．ワークキューは，Linux カーネルの様々な機能から用いられており，それらがカスケード処理を発生させる要因となる．今回の評価では，測定プログラムを実行中に，任意でカスケード処理を発生させるために，測定プログラム開始後，RT 用 CPU コア上でワークキューの遅延処理を動作させるためのタイマ登録関数を実行する．

### 5.4 評価結果と考察

図 8 から図 10 に I から III までのケースにおける割込み遅延時間を計測したヒストグラムを示す．横軸は割り込み

遅延時間を μs 単位で，縦軸はその頻度を示している．図 8 では，CPU affinity を適用していないため，最大の割込み遅延時間が 440μs 近くまである．一方で，CPU affinity によって測定プログラムを RT 用 CPU コアで動作させた場合の割込み遅延時間は図 9 に示すように 60μs 以下となり，大きく性能が改善している．しかし，このようなケースにおいても，図 10 に示すように，カスケード処理が発生すると 900μs を超える割込み遅延時間が発生し，リアルタイム性を大きく損なっていることがわかる．図 11 は，図 10 のように割込み遅延時間が 900μs を超えた際のカーネルのトレースログを示したものである．横軸は時間軸であり，上段と下段の 2 つのラインは CPU0 と CPU1 に対応している．今回の実験に用いた構成は，CPU0 が汎用 CPU コア，CPU1 が RT 用 CPU コアである．それぞれの CPU 上で起きたイベントやタスクの実行状態がそのライン上の垂直線や矩形で示されている．カスケードが起きる前の周期は①に示すように 300μs 周期を満たしている．その後，ワークキューの遅延処理を行うためにタイマが RT 用 CPU コアで発火し，次の周期で起床する測定プログラムの前に点線で囲った箇所で大量のカスケード処理が発生しており，そのため，②の間隔が 900μs を超えるという状況が発生している．一方で，図 12 は，ケース III と同様にワークキューの遅延処理を行うためにタイマが発火したにも関わらず，本提案手法の機能によって，RT 用 CPU コア上ではカスケード処理が発生しておらず，②においてもリアルタイム性を損なうことなく 300μs 周期で動作できている．また図 13 はケース IV における割込み遅延時間を示しており，図 9 と同様に割込み遅延時間を 60μs 以下に抑えることができている．

## 6. おわりに

本稿では，CPU affinity を用いて，Linux におけるリアルタイム性を向上させる手法について提案した．CPU コア専用カーネルスレッドは，CPU affinity を適用することができず，中でもタイマ処理は割込み遅延時間に大きく影響を与えることがあるという問題提示を行った．本論文に提案した手法では，RT 用 CPU コアのタイマのリストを空の状態に維持することで，タイマの発火を防ぎ，カスケード処理が発生しなくなる．この手法により RT 用 CPU コア上で動作する RT タスクのリアルタイム性が保障できることを確認した．

本論文で提案した手法は，タイマ処理が及ぼすリアルタイム性への影響に着目したが，タイマ処理以外にも， 登録された処理をリストで管理し，適時実行するコア専用カーネルスレッドが存在する．このような処理は，実行中にカーネルを割込み禁止やプリエンプション禁止状態にすることがあるため，リアルタイム性に影響を及ぼす．そこで RT 用 CPU コアと汎用 CPU コア間で，登録される処理の量に差を設けて，極力 RT 用 CPU コアで割込み禁止状態やプリエンプション禁止状態を避けることで，さらにリアルタイム性の改善を期待することができると考えている．また，今回，実験に用いたワークキューは，タイマ処理への依存性が高く，発火したタイマが動作していた CPU コア上のワークキューに遅延処理を依頼するという仕組みになっていた．そのため，RT 用 CPU コア上ではワークキューの遅延処理を動作させることができなかった．今後は，タイマ処理とワークキューの遅延処理は，別の CPU コア上でも動作できるようにすることで RT 用 CPU コアでも従来通りワークキューの遅延処理が同様に使えるようにする必要がある．

## 参考文献

1) 保田 信長, 飯山 真一, 冨山 宏之, 高田 広章, 中島 浩: Linux と ITRON によるハイブリッド OS の設計と実装, 情報処理学会研究報告. SLDM, [システム LSI 設計技術], Vol.2004, No.33, pp.45-50 (2004).
2) 中嶋 健一郎, 本田 晋也, 手嶋 茂晴, 高田 広章: セキュリティ支援ハードウェアによるハイブリッド OS システムの高信頼化, 電子情報通信学会論文誌. D, 情報・システム, Vol.93, No.2, pp.75-85 (2010).
3) 金城 聖, 永島 力, 元濱 努, 片山, 吉章, 毛利 公一: リアルタイム仮想化ソフトウェア基盤におけるタイマ割込み通知機構, 情報処理学会研究報告. EMB, 組込みシステム, Vol.2008, No.116, pp.9-16 (2008).
4) RTAI - the RealTime Application Interface for Linux from DIAPM, https://www.rtai.org/isolation
5) Real-Time Linux Wiki, https://rt.wiki.kernel.org/index.php/Main_Page.
6) Brosky, S. and Rotolo, S.: Shielded processors: Guaranteeing sub-millisecond response in standard Linux, Parallel and Distributed Processing Symposium, 2003. Proceedings. International(2003).
7) Piel, Eric. et al.: Asymmetric scheduling and load balancing for real-time on Linux SMP, Parallel Processing and Applied Mathematics, pp896-903(2006).
8) Daniel P. Bovet, Marco Cesati et al.: 詳細 Linux カーネル第 3 版, オーム社, オライリー・ジャパン(2007).
9) Documentation/timers/NO_HZ.txt, http://lwn.net/Articles/549593/
10) 平田 豊: Linux デバイスドライバプログラミング, ソフトバンククリエイティブ株式会社(2008).