Vol.2012-OS-123 No.12
Vol.2012-EMB-27 No.12
2012/12/5

組込みマルチコア向け仮想化環境における性能低下抑止手法

太田 貴也[1]　Daniel Sangorrin[1]　本田 晋也[1]　高田 広章[1]

概要：今日の組込みシステムは，適用分野が増加しており，一部の適用分野では，多機能性とリアルタイム性といった相反する性質の両方が求められる場合がある．我々は，こうした要求に答えるための組込み向け仮想化環境である SafeG を研究開発している．SafeG を用いてリアルタイム OS(RTOS) と汎用 OS を同一システム上で実行することにより，多機能性とリアルタイム性を満たすシステム構築が可能である．SafeG はシングル・マルチコアの両方に対応しているが，マルチコア SafeG 特有の問題として，汎用 OS の同期処理の遅延 (LHP 問題) や負荷の各コアへの不均等な割り当て (不均等負荷問題) による性能低下が発生する．本研究では，こうした問題を解決するための，組込みシステムに適した手法を設計・実装した．具体的には，LHP 問題に関しては，RTOS の処理を行うコアを動的に変更する方法で，不均等負荷問題については，汎用 OS のロードバランサの最適化により性能低下抑止を実現できることを確認した．

1. はじめに

近年の組込みシステムは，適用分野が増加しており，求められる性質もさまざまである．カーナビゲーションシステムや工作機械といった，一部の適用分野においては，多機能性とリアルタイム性の求められる処理が混在している．こういった分野の要求を満たすシステムを単一のコンピュータで実現するのは難しく，リアルタイム処理を実現するために，別のハードウェアが用意されていることも多い．

OS 仮想化により，単一のコンピュータで，異なる性質を持つ OS の混在が可能となる．具体的には，リアルタイム OS(RTOS) と汎用 OS を仮想化技術を用いて同時実行し，リアルタイム性の求められる処理を RTOS で，多機能性の求められる処理を汎用 OS で実行する方法が考えられる．しかし，組込みシステムでは RTOS の信頼性を確保する必要があり，既存の OS 仮想化技術ではこれらの要求を全て満たすのは難しい．このため我々は，組込みシステムに適用可能な OS 仮想化環境として，SafeG[7] [8] を設計・実装し，研究を行っている．SafeG は，シングルコアプロセッサまたはマルチコアプロセッサ上で汎用 OS と RTOS の同時実行を実現する．

一方で，現在組込みシステムではマルチコア化が進んでおり，その重要性が増している．この背景として，消費電力の増大を抑えつつ，処理性能の向上をはかるためには，クロック周波数を上げるよりも，コア数を増やした方が有利であるという現状がある．特に，複数のコアを 1 つの LSI 上に集積したオンチップマルチコアプロセッサは，処理性能面からも利点が大きく，広範な組込みシステムへの適用が期待される．

我々は，こうした近年の組込みシステムのハードウェアの変化に対応するために SafeG をマルチコア拡張した [11]．マルチコア対応 SafeG は対象型マルチコアプロセッサ (SMP) システムを対象としており，SMP システム上で実行可能な汎用 OS と RTOS の同時実行をサポートする．しかし，マルチコア特有の問題として，汎用 OS で以下のような性能低下問題が生じることが分かっている．

- Lock Holder Preemption 問題
- 不均等負荷問題

Lock Holder Preemption(LHP) 問題は，同期処理に関する問題であり，ロック取得までの時間が延びることで，同期を必要とする処理の性能低下を招く．また，不均等負荷問題は，負荷バランスに関する問題であり，マルチコアプロセッサの各コアへの不均等な負荷 (タスク) の割り当てにより，タスクの実行が不均等になる．

本研究では，次のようにして 2 つの問題を解決した．LHP 問題については，RTOS の処理を行うコアを動的に変更することで発生を抑えた．また，不均等負荷問題については，RTOS のコア毎の CPU 使用率を元に，汎用 OS のコア間の負荷バランスを調整することで解決を行った．

以降，2 節で SafeG にの詳細なアーキテクチャについて述べ，3 節で Lock Holder Preemption，4 節で不均等負荷問題による性能低下を抑止するための手法について順に述べる．

[1] 名古屋大学大学院情報科学研究科
Graduate School of Information Science, Nagoya University

2. SafeG

SafeG は，組込みシステム向け OS 仮想化環境を実現するためのモニタソフトウェアである．SafeG を用いて，単一のシステム上で汎用 OS と RTOS の同時実行を実現できる．SafeG の機能は，割込み監視と OS 切り替えに限定されており，低オーバヘッドで OS の仮想化を行うことができる．機能の実現には，ARM アーキテクチャのセキュリティ支援技術である ARM TrustZone[14] を活用している．

2.1 マルチコア向け SafeG

マルチコア向けの SafeG は，コードセクションを全てのコアで共有する．ただし，OS 切り替え時に必要なコンテキストの保存領域，スタックなどのデータ領域については，各コアごとに個別の領域を持つ．

マルチコア対応 SafeG は，仮想 CPU のマイグレーションをサポートしない．TrustZone では各コアごとにひとつずつのセキュア/ノンセキュア仮想 CPU を持つ．物理コアごとに用意された一部のハードウェアに関しては，各コア内の仮想 CPU 間で共有されており，設定を他のコアへマイグレーションすることはできない．また，同様の理由により，ある OS の仮想 CPU を同時に一つの物理コアに割り付けることもできない．

SafeG で現在サポートされているマルチコアプロセッサは Cortex-A9 MPCore であり，ゲスト OS は，汎用 OS として Linux を，RTOS として TOPPERS/FMP[5] カーネル (以降，FMP カーネルと呼ぶ) をサポートしている．

2.1.1 OS スケジューリング

SafeG はすべてのコアで RTOS を優先実行する．RTOS として使用する FMP カーネルは，各コアにタスクを静的[*1]に割り当て，コア毎に優先度スケジューリングを行う．あるコアで実行すべき処理 (タスク) がなくなった場合に汎用 OS を実行する．

図 1 にマルチコア対応 SafeG の OS スケジューリングの優先順位を示す．この図では各 OS のカーネル及び処理単位[*2]のシステム全体での実行順を示している．例えば，T1_2 はコア 1 で実行される FMP カーネル上の 2 番目のタスクを表している．P1_1 などは Linux 上で実行されるプロセスまたはスレッドを表している．

FMP カーネル，Linux 共に SMP システムをサポートした OS であり，これら OS のカーネルも SafeG 同様に全てのコアで共有されている．各 OS はそれぞれのコアごとに異なるコンテキストを持っており，OS 内部の実行単位のスケジューリングは各々の OS 内部のスケジューラに依存する．

[*1] FMP カーネルは自動的な負荷分散の仕組みを持たず，基本的にタスクを実行するコアを静的に割り当てる．ただしタスクマイグレーション API が用意されており，タスクからこの API を呼び出すことで，動的な負荷分散を実現できる．

[*2] タスクやスレッドなどの，スケジューラがスケジューリング対象とする処理の単位

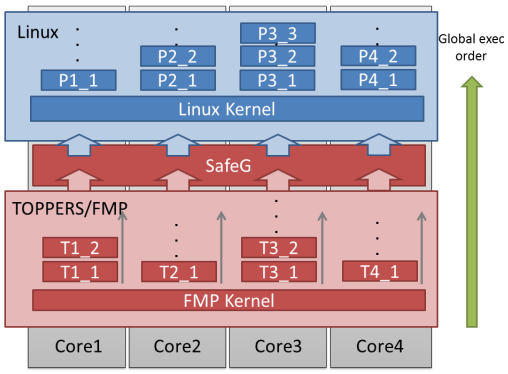

図 1 マルチコア対応 SafeG
Fig. 1 SafeG for Multicore architecture

3. Lock Holder Preemption (LHP)

性能低下問題のうち，まず LHP 問題について，その抑止手法について述べる．LHP はスピンロックの性能低下を招く問題であり，SafeG においては汎用 OS 側の性能低下を招く．

3.1 スピンロック

スピンロックは，マルチコアシステムで使用される同期処理の一つであり，比較的単純な実装により同期処理を実現できるため，多くのマルチコア対応 OS で採用されている．スピンロックではロック取得に失敗した場合に，そのロックが解放されるまでビジー状態になる (これをスピンウェイトと呼ぶ) ことで同期を行う．すなわち，ロック取得待ちのコアは処理をそれより先に進めることができないという特徴を持つ．こうした特徴を持つため，一旦ロックを取得したコアは，早期にロックを解放するように実装する必要があり，通常はロック取得中のコアでは，タスク切り替え (タスクのプリエンプト) が禁止されている．

3.2 LHP とは

LHP は，マルチコアシステム上の仮想環境で生じる同期処理の問題である．LHP の発生により，仮想マシン上のスピンロックの待ち時間が非仮想化時に比べ非常に長くなり，結果として性能劣化を生じる．

LHP が発生する原因は，仮想化により，スピンロック取得コアがプリエンプト禁止状態であるという前提が崩されることである．**図 2** に示すように，仮想環境下では OS 切り替えにより，本来プリエンプト禁止状態であるはずのコアで，スピンロックのクリティカルセクション中のプロセスが，OS ごとプリエンプトされる現象が発生する．この時ロック保持者 (Lock Holder) がプリエンプト (Preemption) されるため，この現象を LHP と呼ぶ．LHP の発生したコ

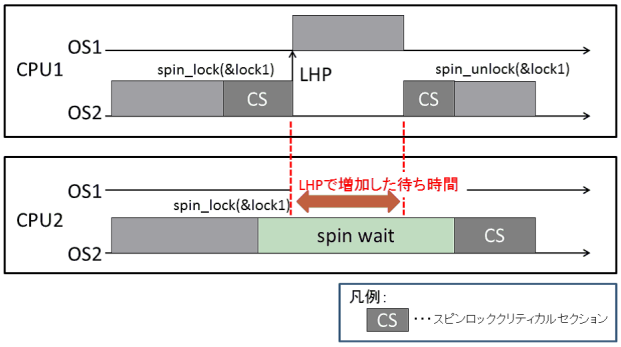

図 2 Lock Holder Preemption (LHP)
Fig. 2 Lock Holder Preemption (LHP)

アでは，他 OS の実行時間分だけスピンロックの解放が遅れ，結果的にそのロックを待っている他のコアのプロセスは，他 OS の実行時間分待ち時間が延びる．

SafeG においては，RTOS が優先的に実行されるため，RTOS がスピンロックを取得中のコアで OS 切り替えが発生することはない．逆に汎用 OS がスピンロックを取得中のコアにおいては，RTOS の処理が発生すれば，汎用 OS がプリエンプトされて RTOS の処理が優先される．したがって，LHP による長時間のスピンロック待ちは汎用 OS 側で発生する問題である．

3.3 関連研究

LHP は，汎用システム向けの仮想化技術でもよく知られた問題で，既存研究が多く存在している．

Delayed Preemption Mechanism(DPM)[2] は，ある物理 CPU で実行中の OS がスピンロックを取得した時に，その物理 CPU で OS の切り替えを一定時間遅延させることで，LHP を抑止しようとした手法である．

co-scheduling は，VMware[3] などの商用仮想マシンモニタ (VMM) で，仮想マシンのスケジューリングアルゴリズムとして採用されている．この手法では，あるロックを取得している仮想 CPU が実行中に，全物理 CPU でその仮想 CPU が属する仮想マシンの仮想 CPU を実行するようにする．これにより，ロック取得コアのみがプリエンプトされ，ロック待ちコアが実行されている状況が起きないようにしている．同様の研究として文献 [4] が挙げられる．

以上の既存研究は，汎用システム向けの研究であり，SafeG での利用は難しい．DPM では，汎用 OS がスピンロックを取得すると OS 切り替えを禁止するため，RTOS の処理が遅延する可能性がある．co-scheduling は，汎用 OS がスピンロック取得中に全てのコアで汎用 OS を実行することになるため，DPM 同様に RTOS の処理が遅延する可能性がある．

3.4 LHP 抑止手法の実現

本節で，LHP の発生抑止を実現するための提案手法について述べる．

3.4.1 要件

本研究では，LHP 抑止手法の実現にあたり，以下のような要件を設定した．

要件 1：RTOS のリアルタイム性 汎用 OS と RTOS のうち，RTOS の方が重要な処理を行うため，手法実現による RTOS のリアルタイム性への影響を最小限とする．

要件 2：汎用 OS への修正量 汎用 OS のバージョンアップ時の追従を容易にするために，手法実現のために加える汎用 OS への変更は必要最小限かつ特定のバージョンに依存しないものとする．

要件 3：適用範囲 汎用 OS では，さまざまな種類のアプリケーションが実行されており，手法適用時にどのようなアプリケーションが動作しているかを予測するのは難しい．このため，手法の適用範囲は，汎用 OS 上で実行されるアプリケーションの特性に依存しないものであることが望ましい．

3.4.2 提案手法

SafeG のターゲットとする，多機能性とリアルタイム性の求められる組込みシステムでは，リアルタイム処理を RTOS が担当しており，リアルタイム処理が必要な場合のみ RTOS を実行する．こうしたリアルタイム処理の処理時間は短く，複数のコアで同時に RTOS のタスクを動作させなくてはならないケースは稀である．

この特性を利用して，本研究では RTOS の処理を行うコアを動的に変更することにより，LHP 抑止を実現した．具体的には，RTOS の処理が発生した場合に，その処理を汎用 OS の非スピンロック取得コアで起動させるようにすることで，LHP を抑止し，ロック取得待ちコアのスピンウェイト時間が長くならないようにした．SafeG の OS 切り替えは割込みを契機として行われる．RTOS の割込みが発生したコアでは，汎用 OS 実行中であっても SafeG により即座に RTOS に実行 OS が切り替えられる．従って，RTOS の処理を行うコアを動的に変更する方法としては，以下の 2 つが考えられる．

- **RTOS 割込みターゲットコアマイグレーション方式** - RTOS の使用する割込み入力信号がターゲットとするコアそのものを動的に変更する
- **RTOS タスクマイグレーション方式** - RTOS への割込みは予め決めたコアに固定し，リアルタイム処理を行うタスクの実行のみを動的に変更する

1 つ目の RTOS 割込みターゲットコアマイグレーション方式では，RTOS の割込みハンドラやタスクの全ての RTOS の処理を汎用 OS の非ロック側コアで起動することが可能であるため，多くの LHP の発生を防ぐことが可能になると期待できる．しかし，この方法を実現するためには，汎用 OS があるコアでロックを取得する度に，割込みターゲッ

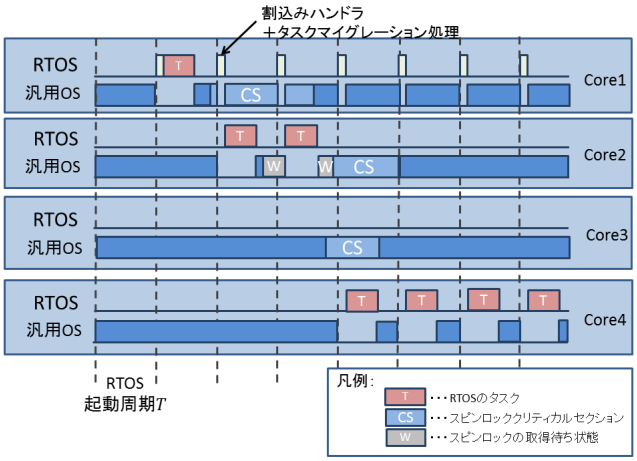

図 3 RTOS タスクマイグレーション方式
Fig. 3 RTOS task migration method

トコアから自コアを排除するように割込みコントローラの設定を変更する必要がある．割込みコントローラの設定は SafeG または RTOS からしか実施できないため，汎用 OS がロックを取得する度に SafeG を呼び出す必要があり，非常に多くのオーバヘッドがかかると考えられる．

2 つめの RTOS タスクマイグレーション方式では，割込みハンドラや，タスクをマイグレーションするための機構はいつも決まったコアで実行されるため，それらの実行時に LHP が発生する可能性がある．しかし，実際のタスクの処理自体を他のコアで実行することで，LHP による長いスピンウェイト時間を削減し，LHP による性能低下の抑止が期待できる．

そこで，本研究では RTOS タスクマイグレーション方式を選択し，実装・評価を行った．この機構を実現するために，FMP カーネルのタスクマイグレーション機能を利用した．**図 3** に実施例を示す．この図では，RTOS 用のタイマ割込みが常にコア 1 でハンドリングされ，そのタイマ割込みで駆動する周期ハンドラにより，一つのタスクが一定周期で実行されている様子を示している．RTOS 上のタスクは最初に，前回起動時のコアで起動を試みられる．もし，前回起動時のコアに属する実行中の汎用 OS のプロセスが，スピンロックを取得していた場合には，汎用 OS がロックを持っていないコアを探し，非ロックコアでタスクを起動する．

全てのコアがロックを取得していた場合については，前回起動時のコアでタスクを起動する．どこかのコアがロックを解放するまで RTOS タスクの起動を遅延させる方法を取ることで，こうした場合でも LHP 発生を防ぐことができる．しかしその場合，リアルタイム性確保が難しくなり，要件 1 を満たせなくなるため，一部の LHP 発生は許容した．

本手法により，RTOS の処理に影響を与えると考えられるのは，マイグレーションによるオーバヘッドである．FMP カーネルのタスクマイグレーションによるオーバヘッドについては文献 [9][10] で述べられており，この文献に従い，マイグレーションオーバヘッドは十分小さいものと仮定して設計を行った．実際のオーバヘッドの計測については，評価の節で述べる．

3.5 設計

提案手法の実装には，汎用 OS のカーネルとして Linux-2.6.35.9 を，RTOS として FMP カーネル 1.2.1 を使用した．

3.5.1 汎用 OS に必要な機能

汎用 OS には，RTOS に対して各コアのスピンロック取得状況を通知するための機構を実装した．この仕組みの実現のために，スピンロックの取得，解放のそれぞれのコードを修正した．ロックの通知用データは，OS 間で共有可能なメモリ領域に配置し，Linux からこのデータに対してデータの更新，FMP カーネルから読み込みを行う．このデータ構造は各コアごとに個別の領域が用意されており，コア間の排他が不要となっている．スピンロック取得状況は「waiting」,「locked」,「unlocked」で表し，それぞれロック取得待ち状態 (スピンウェイト)，ロック取得状態 (クリティカルセクション実行中)，ロック未取得を意味する．Linux のスピンロックの実装は，アーキテクチャに依存したインラインアセンブラで書かれている．スピンロックの取得を行う関数として，arch_spin_lock() 関数，arch_spin_trylock() 関数がある．解放を行う関数は，arch_spin_unlock() 関数である．スピンロック取得要求時にロック通知データを waiting に，スピンロック取得時に locked に更新する．また，スピンロック解放時には unlocked に更新される．

3.5.2 RTOS に必要な機能

RTOS には，汎用 OS により更新された各コアのロック状態を参照するためのインタフェースとして，次のような get_lock_status() という関数を用意した．

- unsigned int get_lock_status(void)

この関数を実行すると，返り値として，汎用 OS 側のロック状態が得られる．この 32 ビットの返り値を 2 ビットずつのブロックに区切り，各ブロックが汎用 OS 側のロック状態 (waiting, locked, unlocked) を保持するようにしている．タスク起動時には，get_lock_status() 関数を利用して，スピンロック取得状況が locked ではないコアを探し，そのコアでタスクを起動するようにした．

3.6 評価

設計した手法の評価のために，4 コアの環境で評価を行った．プロセッサは動作周波数 1.0GHz の RCar-H1(Cortex-A9 × 4) である．L1 キャッシュはコア毎に命令・データそれぞれ 32KB であり，L2 キャッシュは全コア共有で，1MB(Unified) である．また，DRAM として，デュアル

チャンネルの DDR3 を 1GB 搭載している．評価のために，hackbench[13] と NAS Parallel Benchmarks[6] の 2 種類のベンチマークを使用した．いずれも複数のスレッドを生成し，スレッド間で同期処理が頻繁に発生するベンチマークであるため，LHP による性能低下の影響を受けやすいと考えられる．なお，Parallel Benchmarks は複数のベンチマークから構成されるベンチマークスイートである．

具体的な評価内容は以下の通りで，いずれも手法有効時 (migration) と無効時 (static) それぞれで計測を行い，結果を比較している．

評価 1　hackbench の実行において，長時間のスピンロック取得待ちが発生した回数

評価 2　hackbench および，Parallel Benchmarks それぞれのベンチマークスコア

評価 3　FMP カーネルのタスク起動までのオーバヘッド

評価 1 では，LHP が原因と考えられる，長時間のスピンロック待ちが発生した回数を計測している．ここで述べる長時間のスピンロック待ちとは，Linux を SafeG を用いて RTOS と同時実行せず，単体で実行した場合の，最長のスピンロック待ち時間以上の待ち時間を示している．評価 2 では，手法の適用により，Linux の性能低下を抑止できたかを確認するために，ベンチマークスコアを計測した．評価 3 では，要件 1 を満たしているかを確認するために，FMP カーネルのオーバヘッドを計測した．マイグレーション実施時には，FMP カーネルのタスク起動時のオーバヘッドが増加すると予想できるため，タスク起動までのオーバヘッドを計測した．

FMP カーネル側ではタイマ割込みにより駆動される周期ハンドラから，一定周期でタスクを起動する．ここで起動するタスクは一定時間処理を行うダミータスクであり，指定した時間ビジーループを回った後，処理を終了する．手法無効時の場合，タスクは常にコア 1 で起動させられる．手法有効時には，周期ハンドラで Linux のロック状態を確認してから，非ロックコアにマイグレーションして実行される．

3.6.1　(評価 1) 長時間のスピンロック待ち発生回数

評価 1 では，FMP カーネルのダミータスク起動周期のみを変更して複数回行い，それをヒストグラムにまとめた．

図 4 にタスクの起動周期と長時間のスピンウェイト発生回数の関係を示す．ダミータスクすなわち FMP カーネルの起動周期が短くなればなるほど，Linux 側で長時間のスピンウェイトが発生しやすくなっていることが分かる．手法無効時 (static) の方が，周期を短くした場合に極端に長時間のスピンウェイト発生回数が増加している．これに対し，手法有効時 (migration) では，同様に周期が短い程，長時間のスピンウェイトが増加する傾向があるが，いずれの周期でも手法無効時の場合よりも発生回数は少ない．

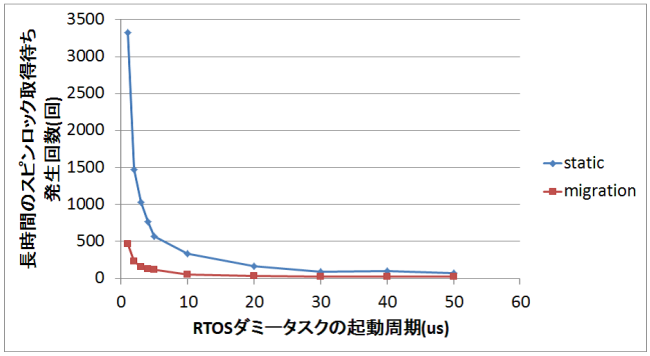

図 4　(評価 1) 長時間のスピンウェイト発生回数
Fig. 4　Evaluation 1: frequency of long spin wait occurrences

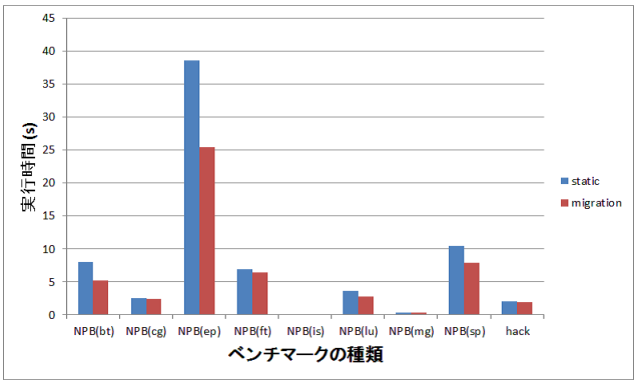

図 5　(評価 2) 各ベンチマークの実行時間
Fig. 5　Evaluation 2: Parallel Benchmarks execution results

3.6.2　(評価 2) ベンチマークスコア

評価 2 では，FMP カーネルのダミータスクの起動周期を $1ms$，負荷時間を $500\mu s$ で固定した上で，hackbench および NAS Parallel Benchmarks を実行した際のベンチマークスコアの計測を行った．

図 5 に hackbench と NAS Parallel Benchmarks の実行結果を示す．NAS Parallel Benchmarks は複数のベンチマークから構成されているため，複数の結果がある．グラフ上では NPB(ベンチマーク名) と表示している．いずれに関しても，手法有効時の方が常によいベンチマークスコアとなっていることを確認できた．

3.6.3　(評価 3)FMP カーネルのオーバヘッド

評価 3 では，FMP カーネルのオーバヘッドがどの程度増加したかを計測するために，周期ハンドラでタスクの起動要求を出してから，実際にタスクが起動するまでの時間を計測した．汎用 OS 側では，hackbench を実行している．計測は FMP カーネルの起動から，Linux で hackbench の実行が終了する間で行い，それをヒストグラムにまとめた．

図 6 に結果を示す．手法有効時 (migration) の方がオーバヘッドが増加しているが，ピーク時の値は数 μs 程の違いであり，大きなオーバヘッド増加は見られなかった．最悪値について比較すると，手法無効時 (static) で $14\mu s$，有効時 (migration) で，$21\mu s$ であった．手法無効時には，FMP

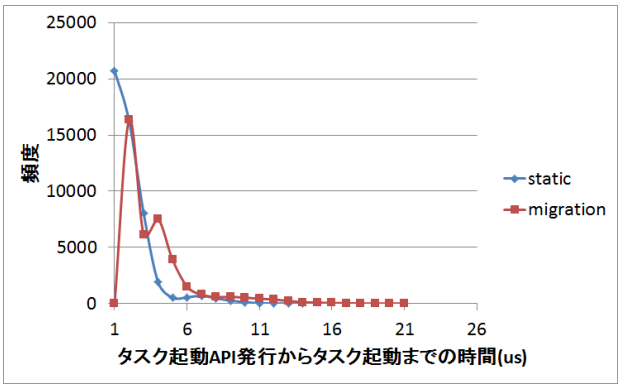

図 6 (評価 3) FMP カーネルの実行オーバヘッド
Fig. 6 Evaluation 3: The execution overhead of FMP Kernel

カーネルのタスクは，いつも同じコアで実行されているため，ピーク以降 ($6\mu s$ 以上) の大きなオーバヘッドについては，キャッシュミスによるものと考えられる．手法有効時には，2 つのピークがある．手法を有効化することで，FMP カーネルのオーバヘッドが増加する原因としては，LHP 発生の確認を行うためのコード実行とタスクマイグレーションが考えられ，左側のピークは LHP 抑止のためのマイグレーションが不要だった場合，右側のピークについては，マイグレーションを行った場合であると考えられる．ピーク以降 ($6\mu s$ 以上) の大きなオーバヘッドについては，手法無効時と同様にキャッシュミスによるものと考えられる．

3.7 考察

手法の実現のために設定した要件を満たしているかについて述べる．まず，要件 1 の RTOS の処理の遅延時間についてだが，本手法ではマイグレーションオーバヘッド以外に RTOS の処理が遅延させられることがない．評価 3 で手法有効時のオーバヘッドを確認した．手法有効時には，確かにオーバヘッドが増加しているものの，最悪応答時間の増加は $10\mu s$ 以下の値に抑えられており，数十 μs 精度のリアルタイム処理については影響がないと言える．

要件 2 の汎用 OS への変更量については，本手法の実現のために必要な修正はスピンロックコードへのロック状態の更新コードのみである．Linux において，ARM アーキテクチャ向けのスピンロックは 2.6.4 以降のバージョンでサポートされているが，本研究で使用した 2.6.35.9 までのバージョンアップの過程で，ロックの取得・解放といった基本的な排他制御の操作の仕組みや実装方法は変化していない．今後のバージョンアップでもスピンロックの実装は大きく変わらないと予想され，提案手法の最新バージョンへの追従は容易であると考えられるため，要件 2 を満たしている．

最後に，要件 3 だが，FMP カーネルでは Linux のロック状態のみを監視しており，Linux のアプリケーションとして何を実行しているかを判断する必要はない．このため，要件 3 に関しても満たしていると言える．

4. 不均等負荷問題

以降で，2 つ目の性能低下問題である，不均等負荷問題について，その問題定義と関連研究および抑止手法について述べる．

4.1 不均等負荷問題とは

一般に SMP システムを対象とする汎用 OS では，動的な負荷分散をサポートしており，各コアへ同じ数のタスクを割り当てようとする．SMP システムでは，すべてのコアの性能が同じであるため，性能を最大化するためには，なるべく均等にタスクを割り振る方がよいためである．

一方，SafeG の OS スケジューリングでは，すべてのコアで RTOS を優先実行する．このため，RTOS の処理が頻繁に行われるコアでは，その分汎用 OS の使用できる物理 CPU 時間が少なくなる．各コアごとの RTOS の CPU 使用率は均一とは限らない．このため汎用 OS の各コアの性能は一定ではなくなる．

しかしながら，汎用 OS は自分自身が仮想化されていることを知るすべがないので，各コアへ同じ数のタスクを割り当てようとする．RTOS の処理が多い物理コアでは相対的に汎用 OS の性能が低下するため，割り付けるタスクの数を減らすべきであるが，そうはなっておらず，不均等なタスク数の割り当てになっていることになる．これが SafeG における不均等負荷問題である．

4.2 関連研究

OS 仮想化技術は汎用システムでも広く使用されており，その多くはマルチコアシステムをサポートしている．ただし，汎用システムでは，全てのゲスト OS は汎用 OS を前提としているため，SafeG とは OS スケジューリングの方法が大きく異なっており，SafeG のような不均等負荷は生じない．Xen[1] の OS スケジューリングでは，個々の仮想マシンごとの重みにしたがって，全 CPU 時間が公平に各仮想マシンに分配される．仮想マシンが複数の仮想 CPU を持つ構成となっていた場合，その仮想マシンに割り振られた CPU 時間は，仮想マシンに属する仮想 CPU に等分配される．したがって，それぞれの仮想 CPU は公平に実行時間を得ることができるため，仮想 CPU 間で性能差が生じることはほとんどなく，不均等負荷問題が生じない．汎用システムでは個々の仮想マシンに公平に実行時間を割り当てることが重要であるため，こうした手法は適切なものと考えられる．しかし，SafeG の OS スケジューリングでは，RTOS の処理がある時に汎用 OS を実行できないため，CPU 使用率を一定に保つことは難しく，同等の手法で不均等負荷を抑止することはできない．

また，非対称型マルチコアプロセッサ (AMP) システムでは不均等負荷問題によく似た状況が発生する．AMP システムでは，搭載されている各コアの性能比が異なる．従って，AMP システムをサポートする OS では，各コアの性能が異なることを考慮した負荷バランスが必要となる．文献 [12] では，AMP システムに対して，Linux のスケジューラを最適化するための手法が述べられている．この研究では，スケジューラのロードバランサが各コアの性能比にしたがって負荷を割り振るようにすることで適切なスケジューリングを実施するようにしている．

4.3 不均等負荷抑止手法の実現

本節で，不均等負荷問題を抑止するための提案手法について述べる．

4.3.1 要件

3 節の LHP 抑止手法と同様に不均等負荷抑止手法も，SafeG への適用を前提とするため，ターゲットとするシステムや求められる性質は同じである．従って，不均等負荷抑止手法の実現にあたり，3.4.1 節で述べたものと同様の要件を設定した．

4.3.2 提案手法

先に述べたように，不均等負荷問題は，汎用 OS のスケジューラが，自分の利用しているコアが全て同じ性能であるという前提で動作していることに起因する問題である．従って，汎用 OS のスケジューラが，仮想化により発生する各コアの性能差を理解し，負荷バランスを実施するように変更することで不均等負荷による性能低下を抑止した．

汎用 OS から見た各コアの性能比は，各々のコアの RTOS の実行が占める CPU 時間との差分により求めることができる．すなわち，RTOS の CPU i における CPU 使用率を U_i^{rtos} とすると，汎用 OS の CPU i の CPU 使用率は $U_i^{gpos} = 1 - U_i^{rtos}$ と表すことができる．汎用 OS の任意のコア i とコア j 間の性能比は $U_i^{gpos} : U_j^{gpos}$ で得られる．

不均等負荷問題が生じないようにするために，汎用 OS のスケジューラのロードバランサが，各コアに割り当てる負荷の比率は，各コアの性能比にそのまま等しくなるようにする．つまり，性能の高いコアほど多くの負荷を与えるようにすればよい．

4.4 設計

提案手法の実装には，汎用 OS のカーネルとして Linux-2.6.35.9 を，RTOS として TOPPERS/FMP カーネル 1.2.1 を使用した．

4.4.1 前提条件

本研究においては，RTOS の処理は周期処理のみであり，かつ RTOS の CPU 使用率は変化しないと仮定する．RTOS の負荷が変動する動的負荷変動の場合については，今後の課題とする．

4.4.2 RTOS の CPU 使用率の見積もり

前提条件より，RTOS の処理は周期処理のみであり，RTOS の各周期における実行時間を一定とする．RTOS の起動周期を T, 各周期における実行時間を C とおいた場合，RTOS の CPU 使用率は $U^{rtos} = C/T$ として見積もることができる．

4.4.3 汎用 OS の変更

Linux では，さまざまな省電力化を実現するための各種ドライバが用意されている．その一つとして，プロセッサの動的周波数変動をサポートしている．マルチコアプロセッサで動的周波数変動が有効な場合，一部のコアの周波数が，他のコアより低くなる状況が発生する．この際，コア間の性能差が生じるため，SafeG で発生する汎用 OS の各コアの性能不均一化同様の状況が発生する．Linux でこうしたマルチコアの動的周波数変動を有効化すると，各コアの負荷バランスはそれぞれのコアの性能比率で割り振られる．従って，こうしたシステム向けのスケジューリングを SafeG 向けの Linux で実施させることで，不均等負荷問題による性能劣化を抑止することが可能になる．

Linux のスケジューラのコードには cpu_power と呼ばれるパラメータが存在する．このパラメータは各コアごとに用意されており，それぞれのコアの性能を示す値である．SMP システムにおいて，全コアの cpu_power 値は一定である．

提案手法を実現するために，cpu_power 値をそのコアの FMP カーネルの使用率に従って修正した．例えば，FMP カーネルの CPU 使用率が 50 %の場合は，そのコアの cpu_power をデフォルト値の 1/2 に設定する．これにより，cpu_power が 1/2 に設定されたコアには，半分の負荷がいつも割り当てられるようにロードバランサは振る舞う．

4.5 評価

評価実験として，4 つの物理 CPU のうち，コア 1 のみ FMP カーネルの負荷をかけた場合の Linux でのベンチマークスコアを計測した．計測の条件として，Linux カーネルの cpu_power 値に修正を加えた場合 (手法適用時) と，修正を加えなかった場合 (手法未適用時) を比較した．本評価で使用したベンチマークは hackbench である．hackbench 実行時には，複数のスレッドが生成され，各スレッドはスケジューラのロードバランサにより実行するコアが選択される．このため，cpu_power 修正による影響を受けやすいと考えられる．RTOS の負荷を変更し，以下の 2 つの評価を行った．

評価 1 コア 1 における, FMP カーネルの負荷率を 50 %とした場合の hackbench スコア

評価 2 コア 1 における, FMP カーネルの負荷率を 80 %とした場合の hackbench スコア

図 7 に評価結果を示す．hackbench のスコアはベンチ

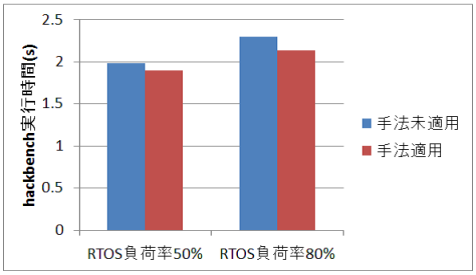

図 7 hackbench の実行結果
Fig. 7 hackbench result

マークの処理が終了するまでの実行時間で表され，実行時間が短い程，速く処理が終わったことを示している．結果が示すように，2 つの評価とも cpu_power に修正を加えた場合 (手法適用時) の方がよいスコアになっていることが確認できた．評価 1 と評価 2 を比較すると，評価 2 の方がスコアの改善率が高く，FMP カーネルの負荷が高い場合，すなわち Linux の使用できる CPU 時間の不均等度合が大きい程，手法の効果が大きいことが分かった．

4.6 考察

手法実現のために設定した要件を満たしているかについて述べる．まず，要件 1 については，本手法実現には汎用 OS 側のロードバランス調整で対応しており，RTOS を常に優先実行するという OS スケジューリングについては変更がない．このため，RTOS のリアルタイム性への影響はないため要件を満たしている．

要件 2 については，手法実現のために使用したのは Linux の動的周波数変動向けの既存機能であり，パラメータの修正のみで対応可能であった．従って，修正量は十分少なく，Linux のバージョンアップ時の追従も容易と考えられる．

要件 3 については，本手法では汎用 OS のロードバランサのコードを修正したのみであり，実際に汎用 OS 上で動作しているアプリケーションを意識する必要はないため，要件を満たしていると言える．

5. おわりに

本研究では，組込みマルチコア向け仮想化環境下で生じる性能低下抑止手法について述べた．性能低下の原因は LHP 問題と不均等負荷問題の 2 つに大別可能で，とくに LHP は汎用システム向けの仮想化環境でもよく知られた問題である．本研究では，組込みシステムの観点からこれら 2 つの問題の抑止手法を提案した．

LHP 問題・不均等負荷問題それぞれで改善すべき課題がある．LHP 問題では，今回読み手と書き手を区別しない spin_lock を対象としたが，Linux に実装されているスピンロックには読み手と書き手を区別する読み書きスピンロックがある．この読み書きスピンロックは更新の少ない共有データの排他制御には適しているが，使用できるケースが限られているため，spin_lock 程の使用頻度はないものと考えられる．しかし，この読み書きスピンロックでも依然として LHP が発生していると考えられ，性能低下に影響を与える可能性があるため，今回の手法により性能低下抑止が可能であるかを確認する必要がある．

不均等負荷問題では，今回 RTOS の負荷が一定であるようなケースに限り性能向上を確認した．今後の課題として，RTOS の負荷が動的に変わる場合にも対応可能にするべきである．RTOS の負荷が動的な場合に対応するためには，RTOS 自身で自分の負荷率を計測し，それを汎用 OS へ通知する仕組みが必要になると考えられる．

参考文献

[1] Barham, Paul and Dragovic, Boris and Fraser, Keir and Hand, Steven and Harris, Tim and Ho, Alex and Neugebauer, Rolf and Pratt, Ian and Warfield, Andrew: *Xen and the art of virtualization*, SIGOPS Oper. Syst. Rev (2003).
[2] Uhlig, Volkmar and LeVasseur, Joshua and Skoglund, Espen and Dannowski, Uwe: *Towards scalable multiprocessor virtual machines*, Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3 (2004)
[3] VMware vSphere 4: The CPU scheduler in VMware(R) ESX(TM) 4 入手先 ⟨http://www.vmware.com/files/pdf/perf-vsphere-cpu_scheduler.pdf⟩
[4] Weng, Chuliang and Liu, Qian and Yu, Lei and Li, Minglu: *Dynamic adaptive scheduling for virtual machines* Proceedings of the 20th international symposium on High performance distributed computing (2011)
[5] TOPPERS 新世代カーネル統合仕様書: 入手先 ⟨https://www.toppers.jp/documents.html⟩
[6] NASA Advanced Supercomputing Division, *NAS Parallel Benchmarks* 入手先 ⟨http://www.nas.nasa.gov/publications/npb.html⟩
[7] 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 (リアルタイムシステム)“, 情報処理学会研究報告. EMB, 組込みシステム (2008)
[8] TOPPERS/SafeG: http://www.toppers.jp/safeg.html
[9] 石田利永子，本田晋也，高田広章，福井昭也，小川敏行，田原康宏，“TOPPERS/FMP カーネル リアルタイム性と高スループットを実現可能な組込システム向けマルチプロセッサ用 RTOS”, コンピュータソフトウェア, Vol.29,No.4, pp. 219-243, Oct 2012.
[10] 相庭裕史，本田晋也，高田広章，“対称型マルチコアシステムのエンジン制御ソフトウェアへの適用”, 情報処理学会論文誌, Vol.51,No.12, pp. 2238-2249, Dec 2010.
[11] 太田貴也，Daniel Sangorrin，一場利幸，本田晋也，高田広章，“組込み向け高信頼デュアル OS モニタのマルチコアアーキテクチャへの適用”, 情報処理学会 第 117 回 OS 研究会, Apr 2011.
[12] Tong Li and Dan Baumberger and David A. Koufaty and Scott Hahn: *Efficient Operating System Scheduling for Performance-Asymmetric MultiCore Architectures*, in SC '07, 2007
[13] hackbench: 入手先 ⟨http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c⟩
[14] ARMLtd, Building a Secure System using TrustZone Technology, PRD29-GENC-009492C