View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Waseda University Repository

2011年度

修士論文

リアルタイム仮想マシンモニタでのLock Holder Preemption問題の解決

早稲田大学基幹理工学研究科
情報理工学専攻

三嶽 仁

学籍番号 5110B117-6

提出 2012年1月31日

指導教員 中島 達夫 教授

Handling Lock-Holder Preemption Problem in Real-Time Virtual Machine Monitors

Hitoshi Mitake

Thesis submitted in partial fullfillment of
the requirements for the degree of

Master in Information and Computer Science

Student ID	5110B117-6
Submission Date	January 31, 2012
Supervisor	Professor Tatsuo Nakajima

Department of Computer Science
Faculty of Science and Engineering
Waseda University

概要

組込み機器の高機能化に伴い、それらの組込み機器のためのソフトウェアを開発するためのコストの増大が問題となってきている．組み込み機器は伝統的に無線通信をはじめとした実時間性を要求される処理を達成する必要がある場合が多い．現代的な組込み機器は、それらに加えてウェブブラウザなどの高機能なソフトウェアを実行することも要求される．そのような相反する要求を達成するために、リアルタイム OS と汎用 OS を単一のハードウェア上で仮想マシンモニタを用いてゲスト OS として同時に実行するという手法が注目されている．

そのような仮想マシンモニタを利用したシステムの開発手法は、リアルタイム性の保証、低い開発コスト、OS 間のアイソレーションなど、多くのメリットを持つ強力なものである反面、仮想化技術特有の問題点も持っている．その一例として、Lock Holder Preemption(LHP) と呼ばれる現象が挙げられる．この現象は、ゲスト OS のスレッドがビジーウェイトに基づいた排他制御を獲得し、その排他制御が保護しているクリティカルセクションの実行中に、OS が想定していないほど長時間プリエンプトされてしまうというものである．ゲスト OS が複数のコアを利用する場合、この現象によりあるプリエンプトされたコアが獲得している排他制御を他のコアが獲得しようと試み、無駄な CPU 時間を消費してしまうという事態につながる．これにより、ゲスト OS の性能が大きく低下してしまうという問題が発生する．

この問題は、特定の実装に固有ではなく、仮想マシンモニタ一般の問題である．なぜならば、基本的に仮想マシンモニタはゲスト OS のスレッドを個別には認識しておらず、仮想 CPU というより大きな単位でしか CPU の割り当てを行わないという原理が原因となって発生する問題だからである．仮想マシンモニタ一般の問題であるため、既にこの問題を解決するための手法は提案されている．しかし、それらの手法は主にサーバ用途で用いられるような仮想マシンモニタを対象としており、リアルタイム組込み機器に適用するには不適当である．

本研究では、LHP による性能の低下を防止する、リアルタイム組込み機器に適用可能な仮想 CPU スケジューリング手法を提案する．

Abstract

Modern embedded systems provide highly functional features. The increasing cost of the development of software for these devices became serious problem. Traditionally, embedded systems are required to execute real-time tasks such as wireless transmission. In addition, modern embedded systems are also required to execute highly functional software such as web browsers. For satisfying these conflicting requirements, the method of executing Real-Time OSes(RTOSes) and General Purpose OSes(GPOSes) on single hardware with Virtual Machine Monitors(VMMs) is considered as effective solution.

The method of development with VMMs are considered effective because it can establish real-time requirements, low development cost, isolation between guest OSes. But it also has the problems which are caused by the nature of VMMs. For example, VMMs cause the phenomenon called Lock Holder Preemption(LHP). This phenomenon is caused when the thread of guest OS which is executing the critical section protected by the mutex based on busy wait mechanism is preempted. If the guest OS has multiple CPUs, the thread on another virtual CPU (vCPU) may try to acquire the mutex which is already acquired by the thread on the preempted vCPU and wastes amount of CPU time. This phenomenon causes fatal degradation of the performance of guest OSes.

This problem is not implementation specific and can be found on any VMMs. Because VMMs do not aware of the threads of the guest OSes. They only schedule the CPU resources with the unit of vCPU. This fundamental principle is the reason of the problem.

The problem is the general one of any VMMs, so there are already existing solutions for solving the problem. But the existing solutions are targeting the VMMs for server side computing. Thus they can not be applied to the VMMs for real-time embedded systems.

In this work, we propose the solution which can avoid the performance degradation and be applied to real-time embedded systems.

目次

第 1 章　序論 **1**
1.1　背景 1
1.2　目的 1
1.3　本稿の構成 2

第 2 章　背景 **3**
2.1　複雑高度化する現代的な組込み機器 3
2.1.1　組込み機器向け VMM 3
2.1.2　組込み機器向け VMM と従来の VMM の違い 4
2.2　VMM の導入により発生する問題 4
2.2.1　LHP がゲスト OS のパフォーマンスへおよぼす影響 5
2.2.2　従来の VMM での解決方法 6
2.2.3　従来の VMM での解決方法を組込み向け VMM に適用出来ない理由 8

第 3 章　設計 **10**
3.1　概要 10
3.2　vCPU スケジューリング 10
3.2.1　割り込み 13
3.2.2　スタック 13
3.3　対象としたハードウェア 13
3.4　SH-4A の特徴 14
3.4.1　割り込み優先度の分割 14
3.5　抽象化する周辺デバイス 16
3.5.1　TMU 16
3.5.2　割り込みコントローラ 17
3.6　MMU の抽象化 17
3.6.1　ASID の分割 17

第 4 章　実装 **19**
4.1　API の呼び出しの仕組み 19
4.2　ゲスト OS をポーティングする際に修正を必要とする箇所 19
4.2.1　メモリ 19
4.2.2　割り込みハンドラ 19
4.2.3　idle タスク 20

4.2.4 トラップされたコンテキストの復元 20
4.3 vCPU マイグレーションのメカニズムとポリシー 21
4.3.1 Ondemand マイグレーションポリシー 21
4.3.2 Trap based マイグレーションポリシー 21
4.3.3 Ondemand マイグレーションと Trap based マイグレーションの両方に共通する処理 . . 23

第 5 章 評価 **24**
5.1 ベンチマークに使用したプログラムの概要 24
5.2 評価結果 24

第 6 章 考察 **26**
6.1 GPOS 内部での IPI の処理方法 26
6.2 受信側 CPU での IPI ハンドラの完了をビジーウェイトで待つ手法の問題点 27

第 7 章 関連研究 **28**
7.1 複数の OS のパーソナリティの共存 28
7.1.1 マイクロカーネル 28
7.1.2 仮想マシンモニタ 28
7.2 LHP の解決 29
7.3 低レイテンシと高スループットの両立 29
7.3.1 割り込みハンドラのスレッド化 29
7.3.2 汎用 OS での優先度継承の実装 29
7.3.3 RTOS と GPOS の組み合わせ 30
7.3.4 その他の試み 30

第 8 章 将来課題 **31**
8.1 IPI の到達の遅延の解消 31
8.2 GPOS によるキャッシュ汚染の RTOS のレイテンシへの影響 31
8.3 VMM を利用したキャッシュの有効活用 31
8.3.1 Computation Spreading 31
8.3.2 FlexSC 31
8.3.3 SPUMONE によるキャッシュの有効活用 32

第 9 章 結論 **33**

35

図 目 次

2.1 LHP による hackbench のスコアの低下 5

3.1 SPUMONE の概要 10
3.2 SPUMONE 上での LHP の発生 (1) 12
3.3 SPUMONE 上での LHP の発生 (2) 12
3.4 割り込み優先度の OS 間での分割 14
3.5 割り込み優先度の分割を行わなかった際の RTOS のレイテンシ 15
3.6 割り込み優先度の分割を行った際の RTOS のレイテンシ 16

5.1 各コンフィギュレーション下での hackbench のスコア 25

第1章　序論

1.1　背景

組込み機器の高度、複雑化に伴い、そのような機器で利用されるソフトウェアの開発が困難となりつつある．その理由としては、スマートフォンに代表されるような現代的な組込み機器は低レイテンシを要求される処理と高スループットを要求される処理というトレードオフの関係にある 2 つの要求を実現しなければならないからである．低レイテンシを要求する処理の例としては、無線通信のプロトコルスタックの実行などが挙げられる．高スループットを要求する処理の例としては、ウェブブラウザのような高機能なソフトウェアの実行などが挙げられる．

これらの処理を単一の OS で実行することは困難である．なぜならば、基本的に全ての OS は汎用的な用途に利用可能かつスループットを最大化するように設計された汎用 OS(GPOS) もしくはリアルタイムな処理を達成するためのリアルタム OS(RTOS) として開発されており、汎用 OS 上でタスクのリアルタイム性を保証することと、リアルタイム OS 上で多様なタスクのスループットを最大化することは困難だからである．

そのような個々の OS の制約を補うため、仮想マシンモニタ (VMM) を用いて単一のハードウェア上で GPOS と RTOS をゲスト OS として同時に実行するという手法が注目されている．この手法により、リアルタイム性が要求される処理は RTOS のタスクとして実行し、高機能なソフトウェアを GPOS で実行することにより、低レイテンシと高スループットの両方の要求を実現することが可能であると考えられている．

1.2　目的

VMM を用いて RTOS と GPOS を同時に実行する手法にも欠点がある．実装に依らない VMM 一般の特徴として、VMM はゲスト OS の状態を粗い粒度でしか識別せず、その特性によりゲスト OS の性能の低下を引き起こしてしまうという特徴がある．例えば、CPU 資源に関して、VMM はゲスト OS の各スレッドを認識せず、仮想 CPU(vCPU) という単位でゲスト OS のコンテキストを認識する．つまり、VMM はゲスト OS がある vCPU 上で実行しているスレッドの状態を考慮せず、vCPU のスケジューリングを行ってしまう．この特性は Lock Holder Preemption(LHP) という現象を発生させる．LHP は、ゲスト OS のスレッドがビジーウェイトに基づいた排他制御を獲得し、その排他制御により保護されているクリティカルセクションを実行している間に VMM によりプリエンプトされてしまうという現象である．この現象が共有メモリマルチコアプロセッサ (SMP) を利用するゲスト OS 上で発生すると、プリエンプトされた vCPU とは別の vCPU が、プリエンプトされた vCPU が獲得した排他制御を獲得しようと試み、大量の CPU 時間を無駄にしてしまい、結果としてそのゲスト OS の性能が低下するという問題が発生する．

上述の通り、このような問題は VMM 一般の実装に依らない問題である．そのため、既にこの問題を解決する vCPU スケジューリング手法は提案されている．しかし、既存の手法は主にサーバ用途などに用いられる VMM を対象としており、スループットの性能を向上させる目的で設計された．そのため、組込み機器向け

VMM にこの手法を適用すると、ゲストの RTOS のリアルタイム性を大きく損ねてしまうという問題がある．本研究の目的は、組込み機器向け VMM 上で、LHP による性能の低下を防ぐ vCPU スケジューリング手法を提案することである．

1.3 本稿の構成

本稿は以下のように構成される．第 2 章で詳細な背景を解説する．第 3 章では提案手法の設計を、第 4 章では評価に利用した VMM の実装を説明する．第 5 章では評価結果を示し、第 6 章ではその結果を考察する．第 7 章では関連研究と本研究との比較を行う．第 8 章では将来課題を挙げ、第 9 章では結論を述べる．

第2章　背景

2.1　複雑高度化する現代的な組込み機器

2012 年現在、プロセッサの高速化と低消費電力化や通信の高速化により、スマートフォン、タブレットのような新たな形態のコンピュータが社会に浸透してきている．これらのコンピュータの台頭により、ウェブブラウザ、メーラ、マルチメディアプレイヤー、ゲームといった多くのアプリケーションが PC を使わずに利用可能になりつつある．こういったコンピュータは一時代前の PC と同等あるいはそれ以上の能力を備えており、内部的には Linux を始めとした GPOS を採用している．

Linux などの高機能だが複雑な OS を採用しなければならない理由として、上述のウェブブラウザなどのアプリケーションをサポートしなければならないことが挙げられる．ウェブブラウザはウェブサーバの通信、HTML の字句解析、構文解析、JavaScript の実行、ウェブページのレンダリングを始めとして、様々な処理を行う必要がある．これらの処理は従来の組込み機器で利用されてきた RTOS 上で実行することが困難であるため、高機能な OS が採用されている．

従来の PC とこれらの機器との間には様々な違いがあるが、次の 2 つが典型的な例である．

1. バッテリー駆動を前提としているため、消費電力に強い制約がある
2. 電話機能を始め、無線通信機能を必ず備える

特に後者の無線通信機能をサポートするため、スマートフォンを含む典型的な携帯電話は無線通信のプロトコルスタックを実行するための DSP を実装している．

そのような DSP は用途が限定されているため、利用されていない場合は DSP の実装に利用されたチップ上の面積は有効活用することが出来ない．また、ハードウェアとしてプロトコルスタックを実装するとプロトコルのバージョンアップや新たなプロトコルに対応することが出来なくなる．

そのため、DSP を利用せずに汎用のプロセッサと RTOS を用い、その上で RTOS のタスクとして無線通信のプロトコルスタックを実装するという手法が効果的である．この手法であればプロトコルのバージョンアップや新たなプロトコルのサポートはソフトウェアをアップデートすることで対応出来、さらに通信が行われていない状況では空いたプロセッサを別の計算に利用可能になるという大きなメリットがある．

だが現状の組込み機器の開発ではアドホックな手法で RTOS と GPOS を組み合わせるという手段がとられていることが多く、RTOS が利用していないプロセッサを GPOS が有効活用出来ていない．この状況を改善し、複数の OS が 1 つの物理プロセッサを共有するための手法として、VMM が注目されている．

2.1.1　組込み機器向け VMM

以上のような動機により、単一のハードウェア上で RTOS と GPOS を共存させることが出来ればハードウェア資源を最大限活用することが出来る．そのため、従来ではデータセンターなどで利用されていた VMM の技術を組込み機器でも利用するという試みがなされつつある．

RedBend 社の提供する VLX はそのような目的のために開発された VMM である [10]．Open Kernel Labs 社の OKL4 は L4 マイクロカーネル [30] を基本とした仮想化技術であり、組込み機器への利用も対象としている [28].

注意すべき点として、RTOS と GPOS の共存以外にも、一台のスマートフォンの上に複数のユーザアカウントの設定を保存する目的にも仮想化技術が利用され始めているという潮流がある．この場合、ゲスト OS は RTOS と GPOS が 1 つずつという構成ではなく、GPOS が複数実行される．主な目的は、一台のスマートフォンにプライベート用の設定と、企業の業務用の設定を保存し、実行時にそれらを切り替えることで所有しなければならない機器の台数を削減するというものである．VMware 社はこのような目的のため複数のゲスト OS をスマートフォン上で実行可能な VMM を提供している [21]．Cells は厳密には VMM ではないが、複数のユーザアカウントの設定の保存と切り替えを目的とした技術である **??**．このような技術は上述の RTOS と GPOS を組み合わせるための VMM と共通点を持つが、根本的に違う目的を持っている．本研究で対象とするのは、RTOS と GPOS を組み合わせるための VMM である．

2.1.2 組込み機器向け VMM と従来の VMM の違い

組込み機器向け VMM と従来のデータセンターなどで利用されてきた VMM との違いを説明する．従来の VMM の主な目的は、本来であれば OS の数だけ必要だった計算機の数を統合により減らすこと、データセンター間での OS のライブマイグレーション、過去のソフトウェア資産の有効活用などである．そういった目的のために利用されている VMM の例としては、Xen [4]、KVM [6]、VMware [7] などがある．このような VMM は典型的には数十から百のゲスト OS をホストし、それらのゲスト OS に抽象化したハードウェア資源を平等にあるいは重みを付けて振り分ける．各ゲスト OS は優先度の違いはあっても本質的には同等の役割を果たす．ここが組込み機器向け VMM と従来の VMM との最大の違いである．組込み機器向けの VMM は RTOS と GPOS という役割の異なるゲスト OS をホストする．RTOS 上のタスクは実行可能になった時点で即座に実行される必要があるため、ゲスト OS に物理 CPU を割り当てるためのスケジューリングアルゴリズムは固定優先度になる．また、従来の VMM では各ゲスト OS を所有しているユーザが異なる場合がほとんどであったのに対し、組込み機器向け VMM では RTOS と GPOS は 1 人のユーザにより所有されている．

2.2 VMM の導入により発生する問題

上述のように VMM の導入は多くのメリットを持つが、同時に VMM が存在しなかった場合には発生しない問題をもたらす．代表的な具体例として、ファイルシステムや CPU 資源の抽象化による問題が挙げられる．Pfaff らは VMM のディスクイメージがホスト OS から見た場合にはゲスト OS のファイルシステムとは関連を持たないファイルとして扱われることにより、細かい粒度でのアクセス制限やバージョニングが行えない問題を解決するための手法を提案した [39]．Uhlig らは、VMM は vCPU という単位でしかスケジューリングを行わず、ゲスト OS のスレッドの状態を考慮しないことにより Lock Holder Preemption(LHP) という現象が発生し、それによるパフォーマンス上の問題の解決方法を提案した [2].

この 2 つの問題はどちらも VMM がゲスト OS を粗い粒度でしか認識しないことにより発生する Semantic Gap を原因としている．VMM のゲスト OS を抽象的に扱うことは強力なメリットでもある．Pfaff らの研究は仮想ディスクがホスト OS からは単なるバイナリファイルとしか認識出来ないという点を問題としているが、そ

のように扱うことで仮想ディスク全体のバックアップやバージョニングが容易であるという利点もある．また、VMM がゲスト OS のスレッドの状態を認識せずに vCPU という単位でコンテキストを認識するため、VM の状態のシリアライズが容易になり、ライブマイグレーションが行い易いという面もある．このような Semantic Gap は VMM を導入した際に様々な場面で見られるものである．

2.2.1 LHP がゲスト OS のパフォーマンスへおよぼす影響

具体的に、LHP がゲスト OS のパフォーマンスにどの程度の影響を及ぼすのかを、本研究で用いる VMM である SPUMONE [1] を例として示す．SPUMONE の詳細は第 3 章で解説する．

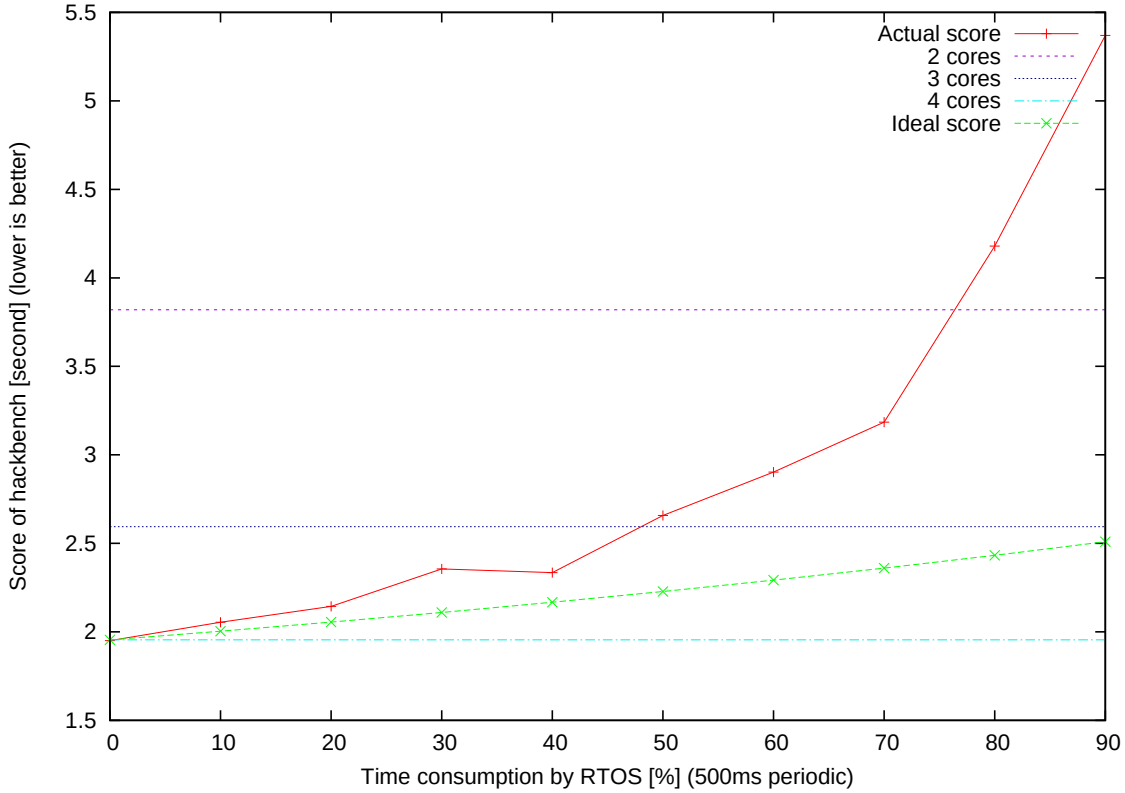

図 2.1: LHP による hackbench のスコアの低下

図 2.1 は、SPUMONE を 4 つの pCPU を持ったハードウェア上で実行し、4 つの vCPU を利用する SMP Linux と 1 つの vCPU を利用する RTOS TOPPERS を同時に実行させ、Linux 上で hackbench というベンチマークプログラムを実行し、TOPPERS 上で周期的に CPU 時間を消費するためのプログラムを実行した際のスコアを示したものである．縦軸は hackbench のスコアであり、これは特定の処理を終えるまでに必要とした時間であるので、低いほど良い．横軸は TOPPERS の周期タスクの CPU 使用率を表している．3 つの水平の線は Linux が pCPU を独占した際のスコアとなっている．最も低い線は 4 つの pCPU を独占した際のスコアであり、最も高い線は 2 つの pCPU を独占した際のスコアである．

hackbench は十分な並列性を持っているため、使用するコア数に比例して性能が向上する．このようなベンチマークを行った際、TOPPERS の CPU 使用率を $f(0 \leq f < 1)$ とすると、Linux 側が利用出来る CPU は $4 - f$ となる[1]．この時、hackbench の理想のスコアは $I(f) = \frac{S_1}{4-f}$ となる (S_1 は、1 つの pCPU を利用した時

[1] $f = 1$ の場合、Linux が利用出来る CPU は $4 - 1 = 3$ とはならない．これは、ゲスト OS である Linux 側から見ると 1 つの CPU

の hackbench のスコアである).

しかし、実際に計測されるスコアはこの理想のスコアとは大きく異なる．実際に計測されたスコアは、図 2.1 の TOPPERS + Linux とラベルされた折れ線で表現されている．この折れ線が表現するように、TOPPERS の CPU 使用率が 50%を越えた時点で hackbench のスコアは 3 つの pCPU を占有した時のスコアよりも悪くなっている．さらに、TOPPERS の CPU 使用率が 80%を越えると、hackbench のスコアは 2 つの pCPU を占有した状況よりも悪くなってしまう．

これは、LHP による影響により、TOPPERS と Linux に 1 つの pCPU を多重化させず、静的に CPU 資源をパーティショニングした方が、TOPPERS の CPU 使用率によっては有効に資源を利用出来、Linux の性能を高く保てることを意味する．この場合、TOPPERS に 1 つの pCPU を割り当て、Linux に 3 つの pCPU を割り当てるという静的な CPU のパーティショニングを行えば、TOPPERS のタスクの周期が 50%を越えた場合に、多重化した場合よりも良い性能が得られることになる．

しかし、静的なパーティショニングを行うと、TOPPERS 側が全く CPU 資源を利用していない状況でも Linux 側がその CPU を利用することが出来ないというデメリットが生ずる．そのため、可能な限り CPU 資源を有効活用するためにも、RTOS と GPOS に pCPU を共有させ、vCPU マイグレーションにより LHP を回避して 2 つの OS を共存させる、という手法が望ましい．

2.2.2 従来の VMM での解決方法

Uhlig らは LHP の回避方法を提案し、LHP によるパフォーマンスの問題を軽減することに成功している．彼等は準仮想化技術と完全仮想化技術のためにそれぞれ 1 つずつ手法を考案した．また、Wells らによりプロセッサに特殊な拡張機能を実装し LHP を検出する手法も提案されている [14, 40]．以下に、それらの手法の概要を述べる．

Delayed Preemption Mechanism

Delayed Preemption Mechanism は、ゲスト OS と VMM の両方に修正を加えることで実現される LHP の回避方法である．LHP はカーネル空間を実行中のスレッドがビジーウェイト系の排他制御を獲得している際に発生する現象なので、ゲスト OS のカーネル空間で利用される排他制御に修正を加え、実行中のスレッドがクリティカルセクションを実行中であるということを VMM から読むことの出来るデータ構造にマークする．そのデータ構造を用い、VMM は vCPU をプリエンプションする前にゲスト OS が実行中のスレッドがビジーウェイト系の排他制御を獲得の状況を調べる．もしもそのような排他制御を獲得していない状況であれば即座にプリエンプションが可能であるが、1 つ以上の排他制御を獲得している場合には、プリエンプションの前にクリティカルセクションの実行のための猶予を与える．また、ゲスト OS と共有されるデータ構造を通じて一度プリエンプションがキャンセルされたことを示す．ゲスト OS はクリティカルセクションの実行を終了し排他制御を開放する際、VMM によるプリエンプションのキャンセルの有無を調べる．もしもキャンセルされていれば、その時点で即座に vCPU を yield し、別の VM の vCPU に実行が移るようにする．

以上の方法で、ビジーウェイト系の排他制御により保護されたクリティカルセクションを実行中の vCPU のプリエンプションが発生せず、LHP による性能の低下が抑えられる．

の実行が止まったことと同じなので、処理を継続出来なくなり、そもそもベンチマークプログラムの実行が完了しなくなるからである．

Safe/Unsafe Scheme

上述の Delayed Preemption Mechanism は効果的な手法であるが、ゲスト OS の修正を必要とする．ゲスト OS のソースコードが入手出来ないなど、ゲスト OS の修正は必ずしも可能ではない．そういった場合には完全仮想化技術を用いることになるが、Delayed Preemption Mechanism は適用することが出来ない．

Uhlig らはそのような場合にも利用可能な解決策として、Safe/Unsafe Scheme という手法を提案した．この手法は以下の前提を基に成り立っている．

1. プリエンプションすると性能の低下に繋がる排他制御は必ずカーネル空間で実行される
2. ワークロードに依存するが、カーネル空間を実行中のスレッドはそのような排他制御を常に 1 つ以上獲得している傾向がある

完全仮想化を行う VMM はゲスト OS のスレッドの状態は知ることは出来ないが、vCPU の状態は当然読むことが出来る．実行中のスレッドがカーネル空間を実行しているかユーザ空間を実行しているかは、CPU 内の制御レジスタの特定のビットで表現されることが典型的である．この状態を VMM が読むことで、カーネル空間を実行中であればプリエンプションをキャンセルすることにより、LHP の発生を防ぐことが出来る．

また、カーネル空間を実行中であっても必ずプリエンプション可能な状態もある．ゲスト OS がアイドルタスクを実行している場合ならば、無条件でプリエンプションしても LHP の発生は起こらない．アイドルタスクが排他制御を獲得していることは無いからである．

Spin Detection Buffer

Wells らは Uhlig らの発見した経験則とは別の特性に注目し、プロセッサを拡張することで LHP を回避する手法を提案した [14].

上述の Uhlig らによる手法がビジーウェイト系の排他制御を獲得しているスレッドをプリエンプションしないという、事前に LHP を防ぐ方針であったのに対し、Wells らにより提案された手法は、LHP が発生した後に、プリエンプションされたスレッドが獲得した排他制御を獲得しようとして CPU 時間を浪費しようとしている状態にあるスレッドを検出し、LHP を回避するという方針に基づいている．

Wells らは、そのような排他制御の獲得のために CPU 時間を消費しているスレッドの実行は特徴的なパターンを持つことを発見した．

1. カーネル空間のスレッドは、1024 命令の実行を通じて、ロードされたメモリのアドレスが 8 つ未満もしくはストアされたアドレスが 8 つ未満であれば、プリエンプトされた vCPU が獲得した排他制御の獲得を試みている．
2. ユーザ空間のスレッドは、1024 命令の実行を通じて、ロードされたメモリのアドレスが 8 つ未満であれば、プリエンプトされた vCPU が獲得した排他制御の獲得を試みている．

つまり、あるスレッドがある程度の時間実行されても、ロードおよびストアが行われたアドレスが少なければ、そのスレッドによる計算は進行しておらず、プリエンプトされてしばらくは開放される見込みのない vCPU が獲得している排他制御の獲得を試みている、という判断が行われる．

ユーザ空間を実行しているスレッドについてはストアされたメモリのアドレスを数えない理由は、巨大な配列にインクリメンタルにアクセスするという実行パターンを誤検知してしまうことを防ぐためである．このよ

うなコードをスレッドが実行している時、配列のインデックスに利用される変数がレジスタに格納されている場合などは、メモリの更新がほぼ行われない状態になる．そのため、1024 命令の実行を通じて更新されたメモリアドレスが 8 つ未満であるということが排他制御の獲得以外にも発生し得るため、ユーザ空間とカーネル空間で検出のルールを変更している．

Wells らの手法は、このような判断を行うためにプロセッサに Spin Detection Buffer(SDB) という機能を実装するというものである．SDB はロード、ストアの 8 つのアドレスを格納するために 8 エントリの CAM(Content Addressable Memory) を 2 つ備え、プロセッサにより実行された命令の履歴を保存する．1024 命令毎にこの CAM の内容を調べ、実行中のスレッドが CPU 資源を無駄に消費していると考えられた場合、VMM 層にそれを通知し、vCPU のスケジューリングを行う．これにより、LHP による性能の低下を防ぐことが可能となる．

この検出方法の詳細は [40] に記載されている．

coscheduling

coscheduling とは古くから知られるタスクスケジューリングのアルゴリズムである．このアルゴリズムが考案された目的は、多くのプロセッサを持つ計算機上で、高頻度で RPC などの IPC で通信を行うプロセス群 (task force と呼ばれる) を実行した際、出来るだけ IPC の待ち時間を減らし性能を向上させるというものである [15].

このアルゴリズムは VMM 上での SMP VM のスケジューリングにも適している．上記の task force に属するプロセス間の通信と、VM に属する vCPU 間の通信が似ているためである．ある vCPU を実行する際、その vCPU が所属している VM の他の vCPU を同時に実行すれば、LHP による影響を軽減することが出来る．このため、ゲスト OS の情報を読みとることが難しい完全仮想化の VMM で採用されることが多い．

VMware ESX [7] はタイプ 1 の VMM であり、OS の上でアプリケーションとして動作するのではなく、ハードウェアを直接コントロールする種類の仮想化技術である．このような VMM は、ホスト OS を持たないため vCPU のスケジューリングを完全にコントロール出来る．そのため、coscheduling のような特殊な vCPU スケジューリングを行うことが出来る．VMware ESX は coscheduling を改良した vCPU スケジューリングアルゴリズムを採用している [16].

Sukwong らは、coscheduling の計算量を改良した balance scheduling というアルゴリズムを VMM 向けに提案している [17].

2.2.3 従来の VMM での解決方法を組込み向け VMM に適用出来ない理由

上述の手法のうち、Safe/Unsafe Scheme、Spin Detection Buffer、coscheduling は完全仮想化でも利用可能な技術である一方、vCPU の管理の粒度が粗いため、準仮想化技術に適用するメリットは少ない．

Uhlig らによる Delayed Preemption Mechanism は準仮想化技術にとって有効であり、基本的にはこの手法で LHP による性能の低下を防ぐことが出来る．しかし、プリエンプションを延期させるため、リアルタイム性を要求するゲスト OS が実行されている環境で利用すると、割り込みなどのイベントの配信が遅延されてしまうという欠点がある．また、この方法はゲスト OS のクリティカルセクションの長さに依存するという性質がある．つまり、RTOS と共存した GPOS の性質により、RTOS のレイテンシが受ける影響の度合いが変化する．

そのため、リアルタイム性を持つ VMM のためには、この手法とは異なる、ゲスト OS のリアルタイム性を損ねない手法が必要になる．この手法の提供が本研究の目的である．

第3章　設計

本章では本研究の手法を実装し評価に利用した VMM である SPUMONE の設計を述べる．

3.1　概要

SPUMONE [1] は SH-4A 上で RTOS と GPOS を共存させることを目的とした VMM である．特徴として、vCPU を実行時にマイグレーションする機能と OS 間のアイソレーションを提供しないという点がある．

実行時に vCPU をマイグレーションする機能は、ゲスト OS が利用していない pCPU を VMM のレベルで使用をやめ、消費電力の削減を計るといった利用方法や、本研究の目的である LHP の削減を目的として実装された．

多くの VMM は各ゲスト OS をユーザ空間で実行させることにより、一つの OS のバグが全体を停止させたり、一つの OS がマルウェアに感染することにより他の OS の脅威となってしまうことを防ぐが、SPUMONE はそれを行わない．そのような設計とした目的は、実行時のオーバーヘッドの削減とゲスト OS をポーティングする際のエンジニアリングコストの削減である．

ゲスト OS としては、複数の pCPU を利用する GPOS と単一の pCPU のみを利用する RTOS を想定している．また、GPOS と RTOS の両方に実行すべきタスクが無くなった場合、sleep 命令を実行するためだけの vCPU が実行される．この vCPU は SH-4A の sleep 命令を繰り返し実行する役割しか持たないが、SPUMONE から見た場合他のゲスト OS の vCPU と同じように扱われる．以降、この vCPU を idle vCPU と呼ぶ．

3.2　vCPU スケジューリング

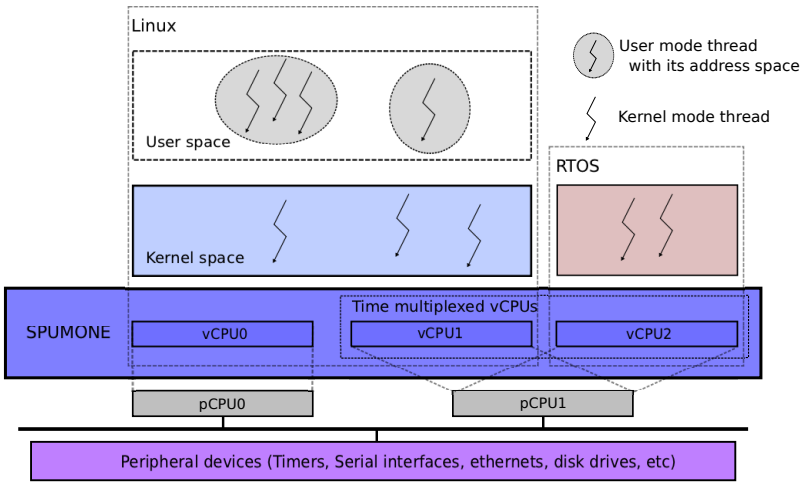

図 3.1: SPUMONE の概要

SPUMONE の概要を図示すると、図 3.1 のようになる．この例では、2 つの pCPU(pCPU0 と pCPU1) を持つプロセッサ上で、2 つの vCPU を利用する SMP GPOS と 1 つの vCPU を利用する RTOS が同時に実行されている．vCPU0 と vCPU1 は GPOS に属し、vCPU2 が RTOS に属している．vCPU0 は pCPU0 上を占有している．vCPU1 と vCPU2 は pCPU1 上で多重化されている．

各 pCPU は vCPU ランキューを持ち、その pCPU 上で実行可能な状態にある vCPU のリストとしている．各ランキューにはそれぞれの idle vCPU が必ず所属しており、実行可能な vCPU が無い場合は idle vCPU が選択され実行される．

また、各 vCPU はペリフェラルから SPUMONE には通知されたが vCPU(ゲスト OS) には通知されていない割り込みを保存しておくキューを持つ．基本的にはこれらのキューへの割り込みの追加は割り込みハンドラから行われる (IPI の場合は他の vCPU の API 呼び出しから行われる)．vCPU がスケジューラにより選択され実行される際、このキューが調べられ、まだディスパッチされていない割り込みが残っていた場合はそれを取り出しディスパッチする．

vCPU はスリープ、サスペンド、ランニングの 3 つの状態を取り得る．スリープ状態の vCPU は実行可能な処理を持っておらず、割り込みを契機として再びランキューに追加され、サスペンド状態を経てランニング状態になるまでは何も行わない．典型的には、ゲスト OS 上でアイドルタスクが実行された際にこの状態への遷移が発生する．サスペンド状態は、実行している処理があるためランキューに入っているが、より高優先度な vCPU が実行中であるため実行されていない状態である．より高優先度な vCPU が yield を行い pCPU を手放した時に再び実行される．ランニング状態は現在実行されている vCPU のみが取り得る状態である．例外が発生した場合、このランニング状態にある vCPU が原因であるが、割り込みが発生した場合にはこの vCPU へ配送されるべき割り込みであるとは限らない．そのため、対応する割り込みハンドラでその割り込みを通知すべき vCPU(1 つとは限らない) に割り込みをキューした後、vCPU のスケジューリングが行われ、より高い優先度の vCPU が実行可能となっている場合にはそちらをランニング状態に遷移させる．

各 OS は 1 つ以上の vCPU を持ち、トラップまたは OS 自身による API 呼び出しを契機として vCPU のスケジューリングが行われる．例えば RTOS の実行すべき処理が無くなった場合、RTOS のアイドルタスクが実行される．このアイドルタスクは後述の通り変更が加えられており、vCPU がスリープ状態に遷移したことを示す SPUMONE の API を呼び出す．この API 呼び出しにより、vCPU2 は pCPU1 のランキューから取り除かれ、次に実行可能な vCPU が選ばれる．GPOS 側に実行可能な処理があった場合、vCPU1 がディスパッチされ、処理を再開する．GPOS 側にも実行すべき処理が無い、つまり vCPU1 がアイドル状態であった場合は、idle vCPU が実行される．

LHP が発生する過程

この SPUMONE を具体例として、LHP が発生する過程を説明する．pCPU(pCPU0 と pCPU1) を 2 つ持ったハードウェアの上で、2 つの vCPU(vCPU0 と vCPU1) を持った SMP GPOS と 1 つの vCPU(vCPU2) を持った RTOS が同時に実行されているとする．pCPU0 は vCPU0 を実行しており、pCPU1 は vCPU1 と vCPU2 を多重化して固定優先度で実行している．つまり、RTOS と GPOS により共有されている．

図 3.2 の状況では、pCPU0 が vCPU0 を、pCPU1 が vCPU1 を実行している．vCPU2 はスリープ状態にある．vCPU1 上で、カーネル空間を実行しているスレッドがスピンロックなどのビジーウェイト系の排他制御を獲得し、その排他制御により保護されたデータを操作しているとする．

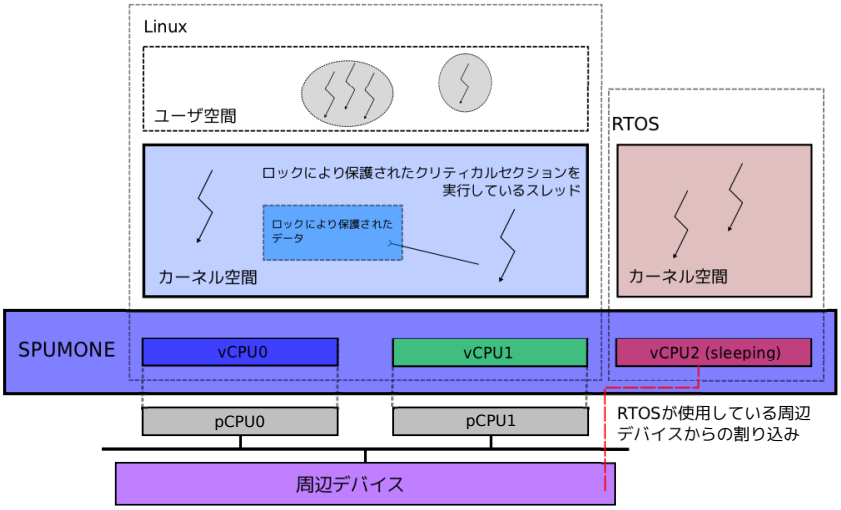

図 3.2: SPUMONE 上での LHP の発生 (1)

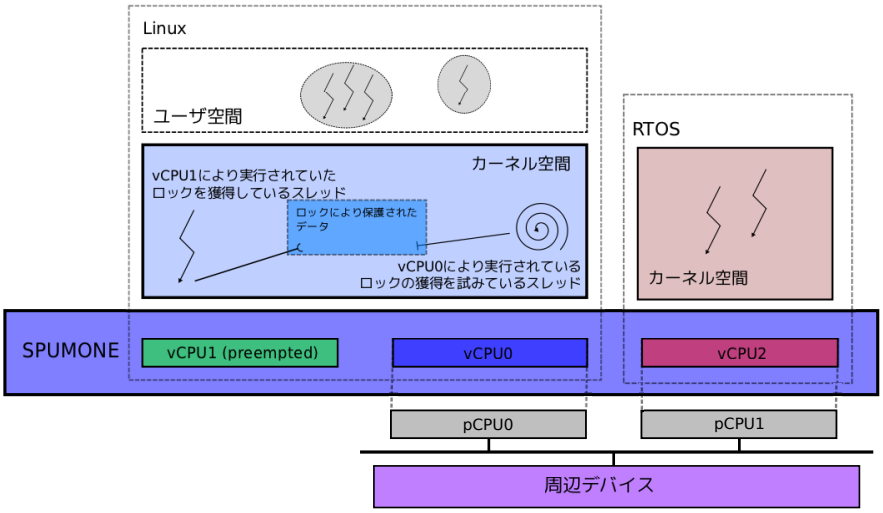

図 3.3: SPUMONE 上での LHP の発生 (2)

このような状態で、vCPU2 に RTOS が利用している周辺デバイスからの割り込みが通知されたとする．vCPU のスケジューリングアルゴリズムは固定優先度スケジューリングであるため、vCPU2 が即座に実行を開始する．

そして vCPU2 が実行を開始し、図 3.3 のような状況になる．この時、vCPU0 上のカーネル空間を実行中のスレッドが、vCPU1 上で実行されていたスレッドが獲得した排他制御の獲得を試みたとする．しかし、この排他制御を獲得しているスレッドは vCPU1 ごと SPUMONE によりプリエンプトされているため、クリティカルセクションの実行を続けることが出来ず、当然排他制御の獲得は行えない．

以上が LHP の発生の過程である．vCPU0 上で実行されているスレッドが、RTOS が CPU を手放すまでは開放されない排他制御の獲得のために CPU 資源を浪費するため、GPOS 側の性能が大きく低下することになる．特に SPUMONE のように vCPU を固定優先度でスケジューリングする VMM の場合、RTOS の vCPU が長時間 CPU を開放しない状況では、性能の低下は大きくなる．

3.2.1 割り込み

本稿では割り込みと例外全てを含んだ実行中のスレッドの中断をトラップと呼ぶ．SPUMONE はトラップが発生した際と、ゲスト OS による API の呼び出しを契機として実行される．一旦 SPUMONE の実行が開始されたら、全ての割り込みが禁止される．

後述の通り一部の vCPU マイグレーションポリシーが有効になっている際は SPUMONE 自体が一部の例外をハンドリングする場合もあるが、基本的に例外が発生した際は vCPU スケジューリングは行われず、現在実行されている vCPU に直接例外をディスパッチする．

割り込みが発生すると、事前に設定した割り込みとその割り込みを利用する vCPU の対応表を用いて、適切な vCPU に割り込みを配送する．vCPU が実行中であれば処理を中断し、vCPU がスリープ状態、つまりランキューに入っていない状態であればサスペンド状態に遷移させランキューに加え、ディスパッチ可能な状態にする．その vCPU が vCPU スケジューラに選択されディスパッチされる際、ゲスト OS の割り込みハンドラから処理が再開される．

3.2.2 スタック

上述のように SPUMONE が実行されている間は全ての割り込みが禁止されるため、SPUMONE の実行がプリエンプトされることは無い．そのため、vCPU はそれぞれのスタックを持たず、スタックは pCPU 毎に 1 つのみ利用され、メモリ資源の削減が計られている．通常の OS や VMM のようにタスク毎もしくは vCPU 毎にスタックを持実行モデルはプロセスモデルと呼ばれるのに対し、このように pCPU 毎にしかスタックを持たない実行モデルは割り込みモデルと呼ばれる．代表的な実装としては、Fluke マイクロカーネルが挙げられる [41].

3.3 対象としたハードウェア

本研究の評価には RP1 というマルチコアを備える組込み機器を想定して設計された試作ボードを利用した．このボードはプロセッサに 4 コアの SH-4A を持ち、128MB の DRAM を備える．周辺デバイスとして、割り込みコントローラ、タイマ、Ethernet コントローラ、SCIF などを持つ．他にもいくつかの周辺デバイスやコアローカルメモリといった特殊な資源を持つが、本研究で利用したのはこの 4 つのデバイスのみである．

3.4 SH-4A の特徴

SH-4A はルネサスエレクトロニクスが開発している RISC プロセッサであり、組込み機器用途に広く利用されている．命令は固定長であり、それぞれ 32KB の命令キャッシュとデータキャッシュを備える．TLB はデータ、命令共用 (UTLB)、命令用 (ITLB) のの 2 種類を持ち、UTLB は 64 ページ分のエントリ、ITLB は 4 ページ分のエントリを備える．

UTLB のミスヒットは例外として OS に通知される．ITLB のミスヒットはハードウェアが自動的にハンドリングし、UTLB の対応するエントリを ITLB に自動的にコピーする．その際に UTLB にも対応するエントリが存在しなかった場合、UTLB のミスヒットとして例外として OS に通知される．これは、一部の PowerPC や MIPS といった RISC アーキテクチャと共通する点である [20].

3.4.1 割り込み優先度の分割

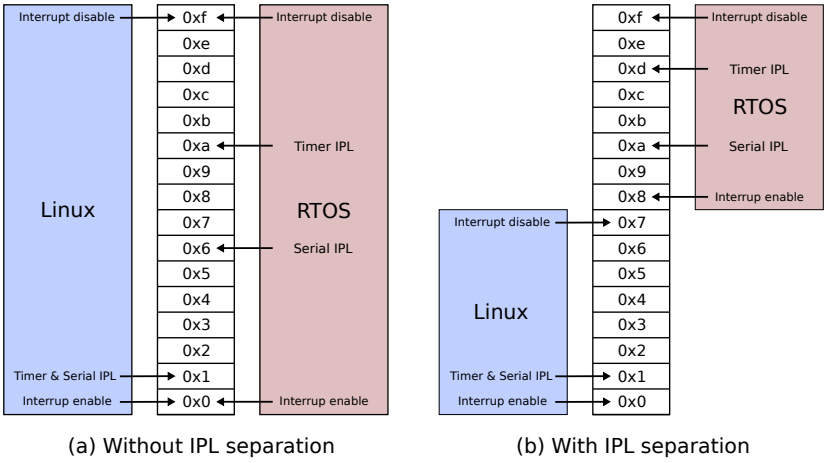

図 3.4: 割り込み優先度の OS 間での分割

SPUMONE は RTOS と GPOS を同時に実行するため、GPOS のアクティビティが RTOS のレイテンシを損うことがあってはならない．このため、ハードウェアの割り込み優先度の分割機能を用いる必要がある．SH-4A や ARM などのプロセッサアーキテクチャは、割り込みに複数の優先度を持たせ、割り込み禁止区間が禁止出来る割り込みを制限することを可能にする機能を持つ．SPUMONE はこの機能を有効活用することで、GPOS の割り込み禁止区間が RTOS への割り込みを禁止する．

具体的には、SH-4A は sr レジスタの 4 ビットの IMASK フィールドを利用し、禁止する割り込みのレベルを設定する．IMASK フィールドを 4 ビットの数値 (0 - 15) として解釈し、設定された値を越える優先度を持つ割り込みのみがプロセッサに通知される．IMASK フィールドが 0x0 である場合、全ての割り込み要因による割り込みが発生し得る．IMASK フィールドが 0x7 である場合は 0x8 以上の割り込みが発生し、0xf である場合は NMI(Non Maskable Interrupt) 以外の全ての割り込みの発生が禁止される．

図 3.4 は、典型的な割り込み優先度の分割の様子を示したものである．図の (a) が割り込み優先度を利用せず、GPOS も RTOS も全てのレベルの割り込みを受け付けられるようにした場合の例である．どちらの OS も割り込み許可区間では sr レジスタの IMASK フィールドは 0x0 に設定し、割り込み禁止区間では 0xf に設定する．この場合、GPOS が割り込みを禁止すると、RTOS が利用する TMU からの割り込みも禁止されてしまう．

また、RTOS が割り込み許可区間を実行している際、GPOS が利用するデバイスからの割り込みによりタスクの実行が中断されてしまう可能性がある．

図の (b) は OS 毎に割り込みの優先度を分割したものである．この場合、GPOS が利用する最大の割り込み禁止レベルは 0x7 である．GPOS が利用するデバイスの割り込み優先度は全て 0x7 以下なので、GPOS への割り込みが GPOS の処理を割り込むことは無い．一方 RTOS の割り込み許可区間では IMASK フィールドは 0x8 に設定される．このため、GPOS の割り込みが RTOS の割り込み許可区間を割り込むことは無い．また、RTOS の割り込み禁止区間では IMASK フィールドは 0xf に設定されるため、RTOS の利用するデバイスからの割り込みを含めて全ての割り込みが禁止される．

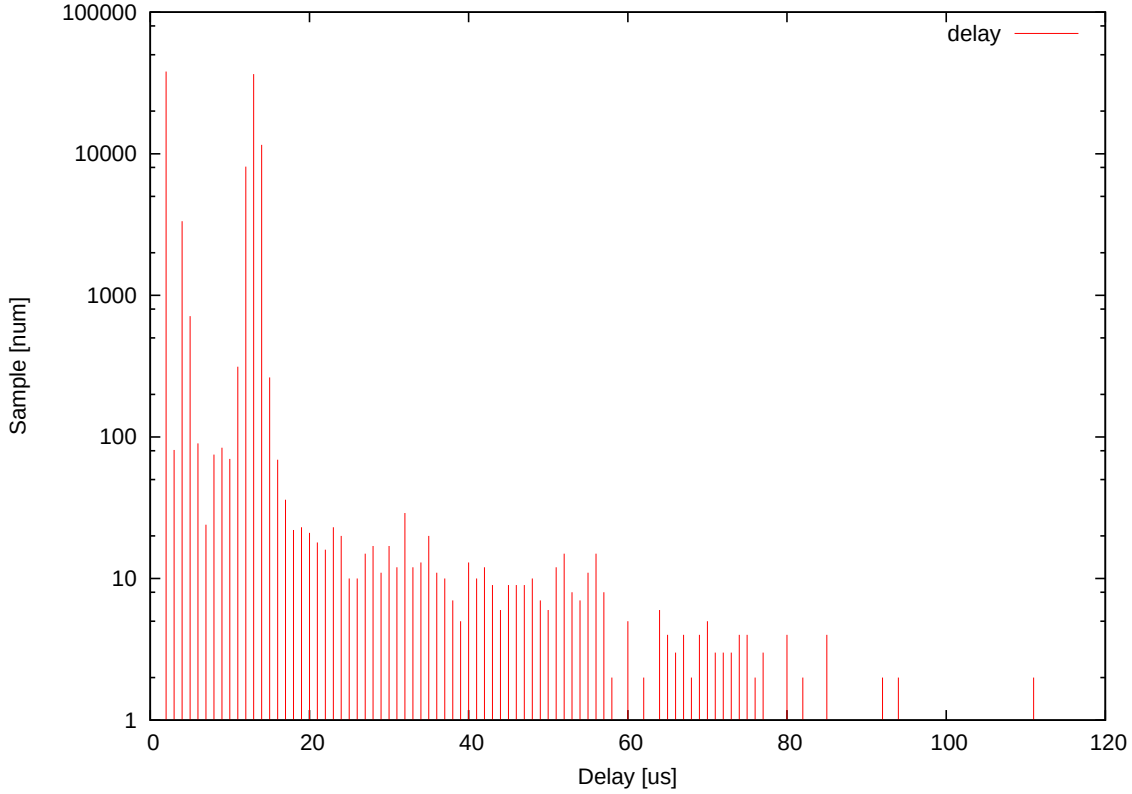

図 3.5: 割り込み優先度の分割を行わなかった際の RTOS のレイテンシ

図 3.5 と図 3.6 は、それぞれ割り込み優先度の分割をの有無による、RTOS のタスクのディスパッチのレイテンシのヒストグラムである．横軸はマイクロ秒単位の遅延であり、縦軸はその発生回数である．このレイテンシは、常時 TMU からの割り込みが発生するように設定した状態で、RTOS のタスクに CPU を yield させ、yield から処理が復帰するまでの時間を計測したものである．

図 3.5 の割り込みの優先度の分割を行わないコンフィギュレーションでは、大きなものでは 100 マイクロ秒を越える遅延が発生していることが分かる．一方、図 3.6 は割り込みの優先度の分割を行ったコンフィギュレーションでの遅延のヒストグラムであり、こちらは全ての遅延が 40 マイクロ秒以内の収まっていることが確認出来る．

このように、ハードウェアの提供する割り込みの優先度の分割を活用することで、RTOS と GPOS を 1 つの pCPU 上で実行しても、RTOS のタスクのディスパッチのレイテンシが保つことが出来る．

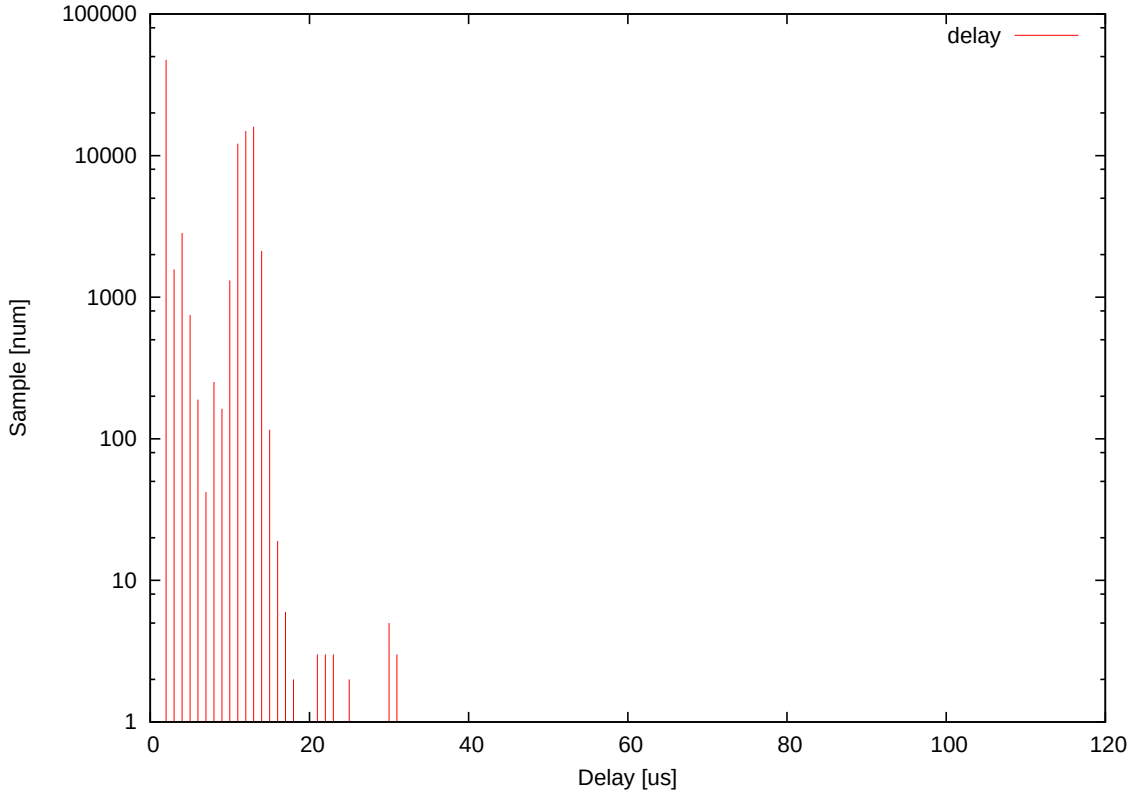

図 3.6: 割り込み優先度の分割を行った際の RTOS のレイテンシ

3.5 抽象化する周辺デバイス

SPUMONE はゲスト OS を特権モードで実行するため、各ゲスト OS は周辺デバイスのレジスタに直接アクセスすることが出来る．そのため、基本的にはビルド時に各ゲスト OS が使用するデバイスを調整しておき、複数のゲスト OS が 1 つのデバイスを利用し競合するというった状況を防ぐ必要がある．

だが、後述の目的のため TMU と割り込みコントローラの IPI を発行する機能は抽象化を行った．抽象化されたデバイスは、API を通じてゲスト OS に利用される．また一般的に、OS の割り込みハンドラは処理の中で割り込み発生源に対して割り込みのハンドリングが行われたことを通知し、発生した割り込みを停止させる．SPUMONE が抽象化したデバイスに関しては、この割り込みの停止は SPUMONE が行うので、ゲスト OS は行う必要が無い．そのため、抽象化されたデバイスのゲスト OS の割り込みハンドラは一部を変更する必要がある．

3.5.1 TMU

vCPU のマイグレーションを行うため、TMU の抽象化を行った．マイグレーションを行わない場合、基本的には 1 つの pCPU の vCPU ランキューに所属し得る vCPU は RTOS と GPOS の vCPU がそれぞれ 1 つのみである (正確には idle vCPU を含めると 3 つである)．だが、vCPU マイグレーションにより GPOS の vCPU がその vCPU を最初に実行した pCPU とは異なる pCPU で実行されるようになると、1 つの pCPU が 2 つの GPOS の vCPU を実行する状態が発生する．このような 1 つの pCPU に複数の vCPU が所属する状態をオーバーコミット状態と呼ぶ．GPOS の vCPU がオーバーコミットした状態では、TMU による割り込みをそれぞ

れの vCPU に平等に振り分ける必要がある．また、GPOS の vCPU はどれも同等の優先度を持つので、それぞれにクオンタムを持たせラウンドロビン方式でスケジューリングを行う．

3.5.2 割り込みコントローラ

vCPU マイグレーションが行われている状況では、IPI の送信をゲスト OS の従来の方法で行うことは出来ない．なぜならば、pCPUn が vCPUn を実行しているとは限らないので、ある vCPUm が vCPUn を送信先として pCPUn に IPI を送信しても、正しく配送されない可能性があるからである．

そのため、SMP GPOS をホストする場合、IPI のルーティングを行う必要がある．pCPUn が vCPUn に向けた IPI を受信した時、vCPUn が pCPUm で実行されている場合には、受信した IPI を pCPUm に転送してやる必要がある．

3.6 MMU の抽象化

SPUMONE は典型的には SMP GPOS と RTOS を 1 つずつ実行する．典型的な RTOS は MMU を利用しないので、各 pCPU に GPOS の vCPU が 1 つずつしか無い状況では、MMU を利用する vCPU は 1 つの pCPU 上に 1 つのみとなるため MMU を抽象化する必要は無い．

しかし、LHP の解決のために vCPU マイグレーションが実行される状況では、1 つの pCPU 上で複数の GPOS の vCPU が実行され得る．そのような状況では MMU を利用する vCPU が 1 つの pCPU 上に複数存在し得るため、MMU の抽象化が必要になる．

MMU の抽象化のため、vCPU のスイッチが発生した際には、MMUCR、PTEH、PTEL といったレジスタを含む MMU の状態は vCPU に関連付けられてメモリ上に退避される．しかし、ITLB と UTLB の内容は退避、復元は行われない．

3.6.1 ASID の分割

SH-4A は、プロセスのコンテキストスイッチが発生した際に全ての TLB をフラッシュせずに済むよう、各スレッドと TLB エントリに ASID(Address Space IDentifier) と呼ばれる 256 ビットのコンテキストを識別するための値を持たせている．異なるアドレス空間に属するスレッドが同じ仮想アドレスを使用することは良くあることであり、複数のスレッドの TLB エントリが MMU に混在している環境では、ある仮想アドレスに対応するエントリが複数存在する状況が発生し得る．このような状況である仮想アドレスを変換しようとした際、どのエントリが使用されるべきかを選択するために利用されるのが ASID である．

各 TLB エントリは ASID を持ち、また、実行中のコンテキスト (スレッド) の ASID は MMU のレジスタ PTEH の下位 8 ビットで表される．TLB の中から仮想アドレスの変換に使用するエントリを選ぶ際、仮想アドレスと PTEH の ASID を元に選択が行われる．

Linux のような通常の SMP GPOS では、各 pCPU は 0 から 255 の ASID 全てを使用する．スレッドはマイグレーションが行われると、その pCPU 上の新しい ASID を確保する．

しかし、SPUMONE のゲスト OS として実行される場合、OS 側にイベントが通知されることなく vCPU がマイグレーションされる可能性がある．そのような場合に、もしも各 vCPU が 0 から 255 全ての ASID を利用していると、複数のスレッドが 1 つの pCPU 上で同じ ASID を利用してしまい、間違ったアドレス変換が行わ

れてしまう可能性がある．そのため、SPUMONE 上で動作する SMP GPOS(Linux) は、各 vCPU 上で使用する ASID が重複することが無いように変更される必要がある．具体的には、RTOS と pCPU を共有する vCPU と、その vCPU がマイグレーションされる pCPU で元々実行されている vCPU は、利用する ASID をそれぞれ 0 から 127、128 から 255 と制限しておくことで、アドレスの誤変換を防止する．

第4章　実装

本章では本研究の手法を実装し評価に利用した VMM である SPUMONE の実装の詳細を述べる．

4.1　API の呼び出しの仕組み

API は vCPU の状態の変更を始めとする SPUMONE 内部のデータ構造の読み書きを行う．そのため、全てのゲスト OS から呼び出せる必要がある．この目的のため、どのゲスト OS が実行されていても同じデータを保存している vbr レジスタを利用する．vbr レジスタに 0x1000 を加算したアドレスが API のエントリを指すよう設定しておき、全てのゲスト OS に共通した API 呼び出しの仕組みを提供する．

API の呼び出しの前に、ゲスト OS は自身のスタック上に r1 から r7 を保存する．その後 vbr + 0x1000 を関数として jsr 命令で呼び出す．これにより制御が SPUMONE に移る．残りのレジスタは SPUMONE 内でゲスト OS 上のスタックあるいは vCPU 構造体に保存される．vCPU の切り替えが発生しない場合はスタック上に保存されたレジスタの内容を復元し、ゲスト OS に戻る．vCPU の切り替えが発生する場合は、ゲスト OS のスタック上に保存されていない部分は vCPU 構造体に保存され、次回その vCPU が実行される時にレジスタに復元される．

vbr を元に API のエントリのアドレスを計算するため、API を呼び出すコンテキストはカーネル空間でなければならない．そのため、ゲスト OS のユーザ空間のプロセスから API が直接呼び出されることは無い．また、呼び出し時のレジスタの内容が保存可能なスタックがあればどのようなコンテキストからも呼び出せるため、スタックの初期化さえ完了していれば割り込みハンドラからも呼び出すことは可能である．

4.2　ゲスト OS をポーティングする際に修正を必要とする箇所

4.2.1　メモリ

SPUMONE はメモリを抽象化しないため、ゲスト OS はそれぞれの物理メモリを直接管理する．複数の OS をゲスト OS として実行するためには、それぞれのゲスト OS が使用するメモリ領域が重複しないようにする必要がある．このために、典型的にはリンカスクリプトに与えるパラメータを変更し、重複を回避する．

各ゲスト OS は ELF ファイルとして SPUMONE に埋め込まれる．起動時に SPUMONE のローダが各ゲスト OS を OS 上に展開し、最初に実行されるべき命令のアドレスをその vCPU の最初に実行されるプログラムカウンタとして設定する．

4.2.2　割り込みハンドラ

SH-4A アーキテクチャは、vbr というレジスタを備え、そのレジスタに記録されたメモリアドレスに基づいて割り込み、例外といったトラップを処理するための OS のハンドラに実行を移す．TLB 以外が原因の例外

(一般例外) が発生した場合は vbr に 0x100 を加算した値がプログラムカウンタに設定される．TLB 例外が発生した場合は 0x400 が加算された値が設定される．ペリフェラルデバイスからの割り込みや NMI が発生した場合には、0x600 が加算された値が設定される．全ての場合において、トラップが発生した時点の sr の値は ssr に、r15 は sgr に、pc は spc に退避される．

通常、SH-4A 上で動作する OS は割り込みが発生する前の起動時の段階で ldc 命令を用いて vbr を適切な値に設定する．しかし、SPUMONE のゲスト OS はこのように直接 vbr を上書きすることは出来ない．もしゲスト OS が直接 vbr を上書きしたとすると、トラップ発生時に SPUMONE が処理を行うことが出来なくなってしまうからである．そのため vbr の登録は API 化され、各 vCPU 毎に異なる値を持つように設計されている．トラップが発生した直後に SPUMONE の割り込み、例外ハンドラに制御が移り、例外であれば現在実行中の vCPU の vbr の値に 0x100 もしくは 0x400 を加算してゲスト OS の例外ハンドラの位置を求めてジャンプする．割り込みであれば割り込みが通知される vCPU の vbr の値に 0x600 を加算しゲスト OS の割り込みハンドラの位置を求めてジャンプする．

4.2.3 idle タスク

通常の OS は、実行可能な処理が無くなると CPU をスリープさせるだけの命令 (SH-4A では sleep) を実行し続けるだけのタスクをスケジューラにより選択し、割り込みの発生を待つ．だが、SPUMONE のゲスト OS として動作する OS は、直接 sleep 命令を発行してしまうとより優先度の低い OS へ処理を移すことが出来なくなってしまうので、sleep 命令を API 化する必要がある．

そのため、SPUMONE のゲスト OS は sleep 命令を実行する箇所を変更し、API 化されたスリープ命令を呼ぶように修正される必要がある．この API が呼び出されると、SPUMONE は呼び出した vCPU をランキューから外し、スリープ状態に遷移させる．その後、別の実行可能な vCPU を選択し、その vCPU を実行する．ランキューから外された vCPU は、その vCPU が利用する割り込みが発生した際に、割り込みハンドラによってランキューに再び追加され、スケジュール可能なサスペンド状態に遷移する．

4.2.4 トラップされたコンテキストの復元

トラップが発生するとまず最初に SPUMONE に制御が移る．割り込みであった場合には、対応する割り込みハンドラを呼び出し、vCPU のスケジューリングを行い、その時点で実行可能な最も高い優先度を持つ vCPU を実行する．例外であった場合は、その時点で実行されている vCPU の例外ハンドラに直接制御を移す．

どちらの場合にも、一旦 SPUMONE 上にトラップが発生した時点でのレジスタの内容が保存され、ゲスト OS に処理が移る前に復元される．復元の処理が無駄に見えるが、これはゲスト OS のスタックにレジスタの内容を退避するためである．トラップ時のレジスタの内容はゲスト OS のハンドラで利用されるため、このような処理が必要になる．

このスタック上に退避されたレジスタの内容は、ハンドラの終了後、割り込まれたコンテキストの復元時にレジスタに復元される．コンテキストの復元自体は API 化される必要は無いが、後述の Trap based マイグレーションポリシーを有効にする場合には、ユーザ空間のコンテキストを復元する際には API を呼び出し vCPU マイグレーションを実行する必要がある．

4.3 vCPU マイグレーションのメカニズムとポリシー

SPUMONE は LHP を回避するために vCPU マイグレーションメカニズムを提供する．本研究ではそれを利用した 2 つのポリシーを実装し、評価を行った．2 つのポリシーで必要となるメカニズムはそれぞれ異なる部分があるため、ポリシー毎に説明する．

4.3.1 Ondemand マイグレーションポリシー

このマイグレーションポリシーを有効にした場合、RTOS と GPOS が共有している pCPU 上では、RTOS の vCPU がランニング状態になると必ず GPOS の vCPU があらかじめ設定された別の pCPU 上にマイグレーションされる．RTOS が API 化された sleep 命令により pCPU を手放すと、その共有された pCPU に GPOS の vCPU が戻ることを許可するために、それを表す変数を更新する．マイグレーションされた vCPU は、API 化された sleep 命令の中でその変数を読み、戻ることが可能であれば元の pCPU にマイグレーションにより戻る．

このようなポリシーにより、RTOS の vCPU が長時間に渡り GPOS の vCPU をプリエンプションすることが無くなり、LHP を回避することが可能となる．

4.3.2 Trap based マイグレーションポリシー

Trap based マイグレーションポリシーの考え方

Trap based マイグレーションポリシーは、ゲスト OS のスレッドがユーザ空間でプリエンプションされても LHP による性能の低下は発生しないという前提に基づいて設計されている．

ゲスト OS のスレッドがカーネル空間でプリエンプションされることが問題である、という経験則は Uhlig や Wells らによる手法を始めとした既存の手法も前提としているものである．基本的に現代の全ての OS は OS によりハードウェア資源が全て管理されているという前提の元に開発されるため、割り込み禁止区間での割り込みの発生や、プリエンプション禁止区間での長時間のプリエンプションを想定していない．そのため、典型的なビジーウェイトにより保護されたクリティカルセクションの実行が、OS が直接ハードウェア資源の全てを制御している状態では数十マイクロ秒で終わることを暗黙の前提としている．しかし VMM がそれらの OS をゲスト OS として実行している状況では、この暗黙の前提が破られてしまう．本来であれば数十マイクロ秒で終了するはずだったクリティカルセクションの実行が、別のゲスト OS に CPU 資源が割り当てられることにより、本来の数百倍もしくは数千倍という時間を要してしまうことも有り得る．それによりそのゲスト OS の他の vCPU の待ち時間も数百から数千倍と長くなり、本来であれば別の計算に利用出来た CPU 資源を同期のために消費してしまい、結果として性能が低下する．

ビジーウェイト系の排他制御により保護されたクリティカルセクションはカーネル内部ではないユーザ空間を実行中のスレッドにも存在する．しかし、ユーザ空間のスレッドはクリティカルセクションを実行中にプリエンプションされても問題とはならない．これは、基本的に全てのアーキテクチャではユーザ空間を実行中のスレッドは特権命令の実行が出来ないため、割り込みの禁止が不可能であるという前提の元にユーザ空間のプログラムが実装されていることによる．典型的には、ユーザ空間のビジーウェイト系の排他制御の獲得は、あらかじめ設定された回数だけロックの獲得を試み、獲得が出来なかった場合は CPU を別のスレッドに yield す

る、という処理を行う．このため、必要以上に CPU 資源の浪費が行われることがなく、OS によるスレッドのプリエンプションが性能の大きな低下を招かない．

Trap based マイグレーションポリシーはこの点に注目して設計された．RTOS の vCPU と pCPU を共有する GPOS の vCPU は、共有された pCPU 上ではユーザ空間の実行のみを行い、トラップによりカーネル空間に遷移した際に、あらかじめ設定された別の pCPU へマイグレーションし、別の GPOS の vCPU と pCPU を共有することで LHP の回避を図る．トラップが終了し、ユーザ空間のコンテキストの復元が行われる際、マイグレーションされていた GPOS の vCPU は RTOS が実行されている pCPU に再びマイグレーションされる．

Trap based マイグレーションポリシーには LHP の回避以外にもメリットがある．それは、RTOS が実行される pCPU 上で GPOS のカーネル空間を実行しないので、GPOS の割り込み禁止区間が実行されることが無いという点である．この特徴により、ハードウェアの割り込みの優先度を分割しなくても RTOS のレイテンシが GPOS により損われることが無くなる．つまり、マルチコアでさえあれば、例えば x86 アーキテクチャのように、ハードウェアが複数の割り込みの優先度をサポートしていなくても RTOS のリアルタイム性を向上させられる．

TLB ミスヒットの処理

本研究に利用した RP1 というボードは、SH-4A を 4 つ持ったプロセッサを搭載している．これらのコアはメモリと周辺デバイスを共有するが、互いの汎用レジスタやいくつかのコントロールレジスタ、また TLB といった資源にはアクセスすることが出来ない．

また、SH-4A の特徴として、TLB ミスヒットはハードウェアにより解決されるのではなく、例外として OS に通知され、OS が例外を起こした仮想アドレスに対応するページのための TLB エントリを登録しなければならない．

TLB への新規エントリの登録は ldtlb 命令の実行で行われる．その際、仮想ページ番号や物理ページ番号、ページサイズなどのパラメータは PTEH、PTEL、PTEA に事前に格納し、ldtlb 命令のパラメータとして与えられる必要がある．登録の際に TLB エントリに空きが無かった場合、LRU アルゴリズムにより消去されるエントリが選択され、消去されたエントリに新しく登録されたエントリが保存される．

この TLB ミスヒットの通知は例外なので、Trap based マイグレーションが有効になっている状態では、このイベントを契機として vCPU マイグレーションが発生する．以下では、マイグレーションされる vCPU が元々実行されていた pCPU を pCPUs(source)、マイグレーションされてカーネル空間の実行が行われる pCPU を pCPUd(destination) と呼ぶ．TLB ミスヒットが pCPUs で発生した場合、その vCPU が pCPUd にマイグレーションされる．

この pCPUd 上で TLB ミスヒットを解決し、ミスヒットの原因となった仮想アドレスに対応するページのエントリを登録する際、pCPUd の TLB エントリに登録するだけでなく、pCPUs の TLB エントリにも登録する必要がある．pCPUs の TLB エントリに登録を行わなければ、ユーザ空間の復元の際に pCPUs にマイグレーションした直後に再び TLB ミスヒットが発生し、処理の継続が出来なくなってしまう．また、同時に pCPUd の TLB エントリにも登録は行わなければならない．なぜならば、システムコールのコンテキストではユーザ空間のアドレスにアクセスすることは多々あり、マイグレーションされたカーネル空間の TLB ミスヒットに対応するためには pCPUd の TLB にも登録を行わなければならないからである．

IPI のルーティング

基本的な SMP を利用する GPOS は、リスケジューリングの要求、キャッシュや TLB のフラッシュ、関数の呼び出しなどを目的として pCPU 間で割り込みを送受信する．Trap based マイグレーションが有効になっている際には、vCPUn が pCPUn で実行されているとは限らない．そのため、単純に vCPUm が vCPUn へ IPI を送信しようとして、pCPUn へ割り込みを送信しても、vCPUn は pCPUn 上で実行されておらず、pCPUk 上で実行されている可能性がある．

そのため、pCPU は IPI による割り込みを受信した際、対象となる GPOS の vCPU がマイグレーションされている場合には、その vCPU を実行している pCPU に対して IPI を転送してやる必要がある．マイグレーション先の pCPU は 1 つなので、一回転送を行えば確実に目標となる vCPU に到達する．

4.3.3 Ondemand マイグレーションと Trap based マイグレーションの両方に共通する処理

マイグレーション先の pCPU 上では 2 つの GPOS の vCPU が実行されることになる．このような状況でも LHP は発生し得るが、RTOS と共有された pCPU 上で発生する LHP とは違い、性能の低下を防ぐことは容易である．Friebel らは、Xen を対象として LHP の回避方法を提案している [5]．Friebel らによると、LHP が発生している状況では、排他制御の獲得にかかる時間が通常の OS のワークロードでは見られないほど長時間かかるケースが観測される．そのため、ゲスト OS の排他制御の獲得の処理を変更し、特定の回数の獲得を試みてなお獲得に失敗する場合は、CPU を yield するハイパーコールを呼び出す、という処理に置き換えることにより、無駄に CPU 資源を排他制御の獲得に消費しないように出来る．

Friebel らの手法は、いくつかの前提に基づいており、GPOS と RTOS のような異種の OS を実行する VMM に持ち込むことは出来ない．その前提とは、

1. 全てのデバイスが抽象化されており、割り込みはメッセージとして vCPU に通知される．
2. pCPU に対して vCPU の数が多い．これは、yield に対する vCPU マイグレーションのコストが相対的に低くなることを意味する．

これらの前提は我々の対象としている環境では成り立たず、直接適用することは出来ない．だが、Trap based マイグレーションが実行されている状況では、GPOS の vCPU2 つが 1 つの pCPU 上で実行されるというオーバーコミット状態が発生するため、この手法をその pCPU 上でのみならば適用可能となる．

我々はこの手法を GPOS の vCPU がオーバーコミットする状況でのみ適用し GPOS 間での LHP の発生とパフォーマンスの低下を防いだ．GPOS と RTOS 間での LHP は Ondemand マイグレーションポリシーまたは Trap based マイグレーションポリシーにより防がれるため、性能の改善が可能であると考えた．

第5章　評価

この章では、hackbench と呼ばれる Linux のスケジューラと IPC のサブシステムを利用するベンチマークプログラムを用いた評価結果を示す.

5.1　ベンチマークに使用したプログラムの概要

hackbench の動作を説明する．hackbench は送信者と受信者となるスレッドあるいはプロセスを 20 ずつ含むグループを単位として作成し、その送信者と受信者が socketpair() もしくは pipe() で作成されたソケットを用いて read()、write() システムコールでデータの送受信を行う．一定のデータの送受信が行われたら各送信者と受信者は処理を終了する．hackbench のスコアとして得られる数値は、データの送受信が開始されてから完了するまでに消費された時間である．つまり、この数値が短いほど早く処理が完了したということであり、Linux の性能が良いことを意味する．注意するべきことは、スレッドもしくはプロセスの作成の時間はこのスコアに換算されないということである.

この評価ではグループを 1 つのみ作成して実行した．つまり、作成されたプロセス数は合計で 41(親プロセス: 1 + 送信者: 20 + 受信者: 20) である．送信者と受信者はアドレス空間を共有しないプロセスとして生成され、通信に用いられるソケットの作成には socketpair() が利用された.

Linux が hackbench を実行している間、RTOS である TOPPERS は周期的に CPU を消費するタスクを実行している．この評価では、500ms 周期で CPU を消費するよう設定した.

5.2　評価結果

図 5.1 は、横軸が TOPPERS のタスクによる CPU 使用率を表し、縦軸が hackbench のスコアを表している．つまり、縦軸の値が小さいほど良いスコアが出ている、ということになる.

Raw SPUMONE とラベルされた折れ線が LHP の対策を行わない SPUMONE のスコアである．Ideal とラベルされた折れ線が、第 2 章で説明された理想のスコアである．Ondemand based vCPU Migration とラベルされた折れ線が Ondemand マイグレーションポリシーを有効にした際のスコアであり、Trap based vCPU Migration とラベルされた折れ線は Trap based マイグレーションポリシーを有効にした際のスコアである.

このグラフが示す通り、Ondemand マイグレーションポリシーは、ある程度のオーバーヘッドが見られるが理想のスコアに近い値を示している．Ondeman マイグレーションポリシー利用時は、ほぼ全てのケースにおいて Linux が 3 コアを独占した状態よりも良いスコアが得られている．つまり、pCPU を静的にパーティショニングした場合と比較しても遜色無い結果が得られたということを意味し、pCPU を多重化することによるメリットを実現出来ている.

しかし、Trap based マイグレーションポリシーはほとんどの状況で通常の SPUMONE と差が少ないか、もしくは劣ったスコアを示している．これは、vCPU マイグレーションによる Linux の性能の向上が見られな

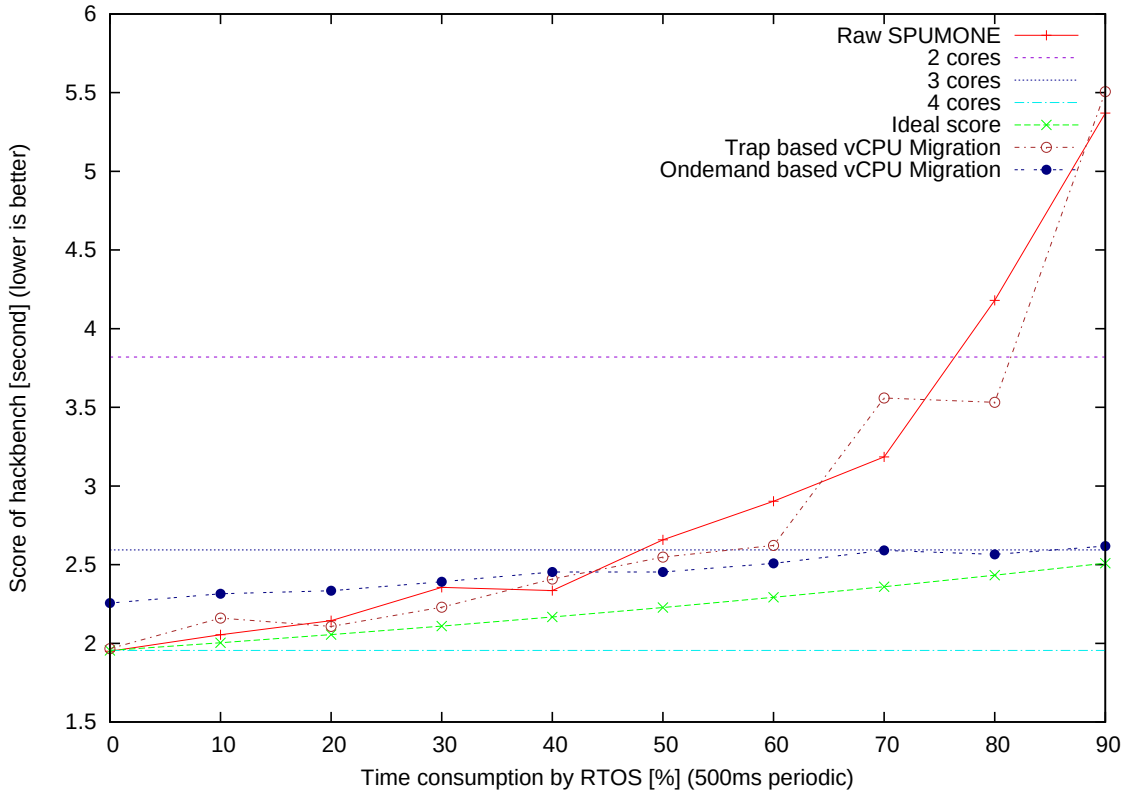

図 5.1: 各コンフィギュレーション下での hackbench のスコア

かったことを意味する.

第6章　考察

この章では結果に対する考察を述べる．主に、trap based マイグレーションポリシーの性能が何故良くないのか、という点に注目して考察する．

RTOS が使用する pCPU で、GPOS のカーネル空間を実行しないというポリシーにより、カーネル空間での RTOS による LHP は回避されている．また、閾値以上の回数ビジーウェイト系排他制御の獲得に失敗したら、vCPU を yield するという vCPU スケジューリングアルゴリズムの採用により、GPOS 間での LHP も回避されている．

LHP を回避しているにもかかわらず性能の向上が見られないという点について、考えられる要因として、IPI の同期による CPU 時間の浪費という問題がある．

6.1　GPOS 内部での IPI の処理方法

今回 SMP GPOS として評価に利用した Linux 2.6.16 は、基本的に 4 種類の IPI を利用する．

1. リスケジューリング
2. TLB のフラッシュ
3. キャッシュのフラッシュ
4. 関数の呼び出し

リスケジューリングは単純に IPI を送信するだけである．送信側は smp_send_reschedule() という関数を呼び出す．この関数は受信側の CPU 番号を引数として受け取る．IPI が到達した際に受信側で呼び出される smp_reschedule_interrupt() というリスケジューリングの割り込みハンドラは、何もせずに処理を終える．リスケジューリングの割り込みハンドラに限らず、Linux は全てのトラップハンドラの末尾で、割り込まれたスレッドにスケジューリングが必要というフラグが立っていた場合はリスケジューリングが行われる．典型的な利用例としては、ある CPU 上で実行されているスレッドから、別の CPU 上で実行されているスレッドにリスケジューリングが必要であるというフラグを立て、smp_send_reschedule() を呼び出しリスケジューリングの IPI を送信するというものである．この時、リスケジューリングの IPI が到達する前に、送信先のスレッドがトラップハンドラの実行を終え、リスケジューリングを行ってしまう可能性がある．しかし、この場合はリスケジューリングの IPI の到達が直接の原因では無いが、結果としてリスケジューリングが行われている．そのため、送信先の CPU で最低一回のリスケジューリングを行うことが出来る．

このリスケジューリング以外の 3 種類の IPI は、TLB のフラッシュは smp_flush_tlb_all()、キャッシュのフラッシュは smp_flush_cache_all()、関数呼び出しは smp_call_function() という関数で実装されている．それぞれの割り込みハンドラには smp_flush_tlb_all_interrupt()、smp_flush_cache_all_interrupt()、smp_call_function_interrupt() が対応する[1]．そして、送信側の関数の中でその関数の処理を保護するスピンロックを獲得する．

[1] 基本的に TLB とキャッシュのフラッシュは一部ではなく全ての CPU の TLB、キャッシュをフラッシュする．

また、これら 3 つの IPI は、それぞれの割り込みハンドラ (サフィックスが_interrupt の関数) の中で、IPI が完了した CPU 数を表現するグローバルなアトミック変数のインクリメントを行う．このアトミック変数はグローバルなので呼び出し側の関数からも読み取ることが出来る．呼び出し側の関数は IPI の送信後、そのアトミック変数が IPI が送信された CPU の数と一致するまでビジーウェイトする．これにより、IPI を送信した全ての CPU で割り込みハンドラが呼び出されたことを確認し、処理を終了する．

6.2 受信側 CPU での IPI ハンドラの完了をビジーウェイトで待つ手法の問題点

上述のような、受信側 CPU での IPI ハンドラの完了を、アトミック変数を用いたビジーウェイトで待つという手法は、ゲスト OS として SPUMONE 上で動作する際に問題を発生させると考えられる．

なぜならば、RTOS と pCPU を共有した GPOS の vCPU が受信側であった場合、RTOS による長時間の GPOS の vCPU のプリエンプションが、IPI の到達を遅延させる可能性が考えられるからである．このような現象はゲスト OS の性能の低下に繋がり得る．

LHP はクリティカルセクションの実行時間を大幅に長くしてしまうという Semantic Gap だったが、この現象は IPI の到達までの時間を大幅に長くしてしまうという Semantic Gap を生ずる．Ondemand マイグレーションポリシーでは RTOS がランニング状態に遷移した場合無条件で GPOS の vCPU をマイグレーションするため、この遅延の影響を受けていないと考えられる．つまり、この IPI の配送の遅延が、Trap based マイグレーションにより性能を向上させることが出来ていない主な原因ではないかと考えられる．

第7章　関連研究

本章では関連研究を挙げ、本研究との比較を行う．本研究と関連のある研究は、以下の 3 つを目的としたものに分類される．

1. 複数の OS のパーソナリティの共存
2. LHP の解決
3. 低レイテンシと高スループットの両立

7.1　複数の OS のパーソナリティの共存

7.1.1　マイクロカーネル

GPOS と RTOS の共存を始めとした複数の OS のパーソナリティを単一の計算機上に実装する試みは古くからある [37, 36, 35]．Mach は良く知られたマイクロカーネルの実装である．Mach はカーネルが提供するメカニズムは IPC やスレッドスケジューリングなどの最低限のものに止め、その上に UNIX のような OS のパーソナリティをプロセスとして実装するという方針に基づいて設計された．

だが開発が進むにつれ Mach や Chorus は IPC によるオーバーヘッドを削減し性能を向上させるため、IPC やスレッドスケジューリング以上の機能をカーネル空間で実装するようになっていった．Liedtke はこの方針を非難し、高性能な IPC を提供すればマイクロカーネル内部に実装される機能は最小限で良いと主張した [29, 30, 31, 44]．

7.1.2　仮想マシンモニタ

上述のマイクロカーネルによる複数の OS のパーソナリティの共存を目指すアプローチは、1990 年代を最後に急速に衰退していった．理由としては、ゲスト OS のポーティングに伴う修正量の多さや、アドレス空間をまたいだ IPC のオーバーヘッドの高さが考えらえれる．

Chorus の開発を行っていた開発者は、RTOS と GPOS を共存させるための手法として VLX という VMM のアプローチを採用した．また、L4 も現在では VMM として利用されており、組込み機器上での RTOS と GPOS の実行を目標としている [27, 28]．

また、Xen を ARM アーキテクチャに移植し、組込み機器で利用するという試みも行われている [12]．しかし、Xen のソースコードの複雑さやオーバーヘッドなどから、実用化は困難であると考えられている [13]．同様の目的を持ったプロジェクトとして、KVM を ARM に移植するという試みも存在する [11]．

7.2 LHP の解決

VMM 導入時の LHP による性能の低下を防ぐ手法は、第 2 で解説されているように複数ある．

Uhlig らは LHP という現象を VMM の文脈で改めて発見し、この名前を付け、Delayed Preemption Mechanism などの解決方法を提案した [2].

しかし、このようユーザレベルスレッドの問題としても古くから認識されていた．Anderson らは OS がユーザレベルスレッドを認識せず、カーネルレベルスレッドでしかスケジューリングを行わないという問題を解決するために Scheduler Activations というメカニズムを提案している [32].

Wells らは LHP が発生した際にゲスト OS のワークロードが特徴的な振舞いをする点に注目し、プロセッサを拡張することでその振舞いを検出し、LHP の検出とそれによる性能の低下の防止を実現する手法を提案している [14].

完全仮想化技術はゲスト OS を修正出来ないという制約により、Delayed Preemption Mechanism のような解決方法を採用することが出来ない．そのため、VMware ESX を始めとした完全仮想化の VMM は coscheduling というスケジューリングアルゴリズムをベースにしたアルゴリズムを vCPU のスケジューリングに採用している [16]．元々coscheduling は VMM のために設計されたものではなく、マルチコアプロセッサ上で頻繁に IPC を行うマルチプロセスのワークロードを対象としてスループットを向上させる目的を持ったスケジューリングアルゴリズムだった [15]．Sukwong らはシングルスレッドのアプリケーションであっても、VMM がホストする SMP のゲスト OS 上で実行される際には LHP による性能の低下が発生することを解決するため、Balance Scheduling という vCPU スケジューリングの手法を提案した [17]．これは coscheduling を改良したものである．

7.3 低レイテンシと高スループットの両立

7.3.1 割り込みハンドラのスレッド化

Kleiman らは割り込みハンドラに専用のスタックを持たせることで、割り込みハンドラのコンテキストをブロック可能にするという手法を提案した [33]．典型的な OS は、割り込みが発生した際にはその時点で実行されていたスレッドのカーネルスタックの上にそのスレッドの状態 (レジスタの内容など) を退避し、割り込みハンドラの実行を行う．このため、割り込みハンドラが、それが割り込んだスレッドに処理を yield することが出来なくなる．これは割り込みハンドラがブロッキングを伴う排他制御を獲得出来ないということを意味する．なぜならば、割り込みハンドラがブロッキングを伴う排他制御の獲得を試みた時、その排他制御が割り込まれたスレッドにより獲得されていた場合、処理を yield することが出来ずデッドロックとなるからである．割り込みハンドラをスレッド化することにより、割り込みハンドラにそれぞれ独自のスタックを持たせ、割り込まれたスレッドに処理を yield することが可能になる．

この考えをさらに拡張したものに、ハードウェアによりスケジューリングを行う Sloth という OS がある [34].

7.3.2 汎用 OS での優先度継承の実装

このように割り込みハンドラをスレッド化すると、割り込みハンドラと例外のコンテキストの排他制御をブロッキングを伴うものに変更出来る．そのような OS では優先度継承によりリアルタイム性を確保することが可能となる．この手法を採用した現代的な OS の 1 つに、Linux の rt パッチが挙げられる [8]．rt パッチを適

用した Linux は、POSIX の API を利用してソフトリアルタイムのアプリケーションを実装することが可能となり、低レイテンシを必要とするエンタープライズ分野などで普及が進んでいる．

しかし、このような優先度継承を利用した OS はレイテンシを低く抑えることが可能な一方、複雑な優先度継承のプロトコルによりスループットを低下させるというデメリットも持つ [9].

7.3.3 RTOS と GPOS の組み合わせ

Yodaiken は、優先度継承の複雑さはスループットの低下を招くと批判した [42]．RTLinux [18] およびその派生の RTAI [19] は、RTOS のアイドルタスクとして Linux を実行する技術である．RTOS と Linux は OS 間通信を行わない場合には同一の排他制御を獲得することは無いので、優先度継承の実装の必要が無く、低いレイテンシと高いスループットの両立が可能であると主張されている．

7.3.4 その他の試み

Manson らは、従来のロックに基づいた排他制御ではない、トランザクショナルメモリに基づいた Preemptible Atomic Regions(PARs) というメカニズムを、Java で書かれたアプリケーションのために提案している [38]．PARs は高優先度のタスクによるクリティカルセクションのプリエンプションが可能であり、そのためリアルタイム用途に適している．OS 内部のロックをこのような高機能な排他制御で置き換えることにより、LHP による性能の低下を防ぐことが出来る可能性もある．

Lee らは、Preemptibility-aware scheduler という OS の設計手法を提案している [23]．この手法はマルチコアに向けて提案されたもので、ハードウェアによる割り込み分散の機能を利用する．任意の時点で最低 1 つのコアが割り込みを受け付けられる状態を保つことで、割り込み禁止区間を減らし、汎用 OS のレイテンシを向上させることを目的としている．

第8章　将来課題

8.1　IPIの到達の遅延の解消

第6章で述べたように、現在のSPUMONEの実装ではIPIの到達が大幅に遅れるというSemantic Gapが生じている可能性がある．Trap basedマイグレーションポリシーをこの現象を考慮するように変更すれば、性能の向上が見込める可能性がある．

8.2　GPOSによるキャッシュ汚染のRTOSのレイテンシへの影響

提案手法はLHPの問題を解決する一方、RTOSとGPOSがpCPUを共有することにより発生するキャッシュ汚染を考慮していない．RTOSは全てのハードウェア資源を利用出来ることを前提に作られていることがあるため、キャッシュが他のOSやVMMにより利用されるだけでもレイテンシを損ねてしまう可能性もある．

そのようなRTOSのため、キャッシュのパーティショニングまで行うLHPの解決方法が望まれる．キャッシュのパーティショニング自体は既に提案されている手法はあり[43]、どの程度までLHPの解決と共存させられるかが問題である．

8.3　VMMを利用したキャッシュの有効活用

本研究ではLHPの解決を目的として、vCPUのマイグレーションによりカーネル空間を実行するCPUとユーザ空間を実行するCPUの分割を行った．LHPの解決などとは全く別の文脈で、このような技術を応用している研究が他に存在する．

8.3.1　Computation Spreading

Chakraborty らは、単一のOSのリソース管理のためにVMMを利用し、キャッシュやTLBなどの有効活用を図るComputation Spreadingという手法を提案している[24]．これは、本研究で提案されたvCPUマイグレーションと似た方法で、マルチコアプロセッサの一部のコアにはカーネル空間のみを実行させ、残りのコアではユーザ空間を実行させる、という手法である．これにより、典型的なウェブサーバやリレーショナルデータベースなどのワークロードの性能の向上を行っている．

8.3.2　FlexSC

Soares と Stumm は、FlexSCと呼ばれるシステムコールに伴なう例外を最小化するOSの設計手法を提案している[25, 26]．

FlexSC は、pthread 互換のスレッドライブラリと、それに対応したカーネルとして実装される．バイナリレベルで互換性があり、共有ライブラリを置き換えるだけで pthread を利用するプログラムに FlexSC を利用させることが出来る．

FlexSC を利用するプログラムは、スレッドの作成に pthread_create() を利用するが、pthread の場合とは異なり、この関数の呼び出しにより作成されるのはカーネルレベルスレッドではなくユーザレベルスレッドとなる．作成されたユーザレベルスレッドがシステムコールを発行を行う API を呼び出した場合、その時点ではシステムコールの発行を行わず、その呼び出しをカーネル空間と共有されるメモリ領域に記録し、CPU を別のユーザレベルスレッドに yield し処理を継続する．実行可能な処理を持つユーザレベルスレッドが無くなったら、その時点で初めてシステムコールを発行する．その後、カーネルレベルスレッドが記録されたシステムコールを順次実行していく．このような方針により、システムコールに伴う例外の数を最小化することで、例外に伴う TLB ミスヒットの増加やパイプラインのフラッシュなどを抑えることが出来る．

上の例はシングルコアの場合でも効果を発揮するが、マルチコアの環境ではさらにシステムコールの実行を担当するカーネルスレッドが利用する CPU を制限することにより、Computation Spreading と同じようにキャッシュの有効活用が可能になる．

8.3.3 SPUMONE によるキャッシュの有効活用

SPUMONE の本来の目的は GPOS と RTOS を組み合わせて実行することだが、上述の Computation Spreading のような目的にも利用が可能であると考えられる．また、Chakraborty らによる評価はシミュレータを利用しており、TLB ミスヒットによる影響を無視するなど、現実的な評価は行われていない．この手法の実際のハードウェア上での評価はまだ無いため、SPUMONE を用いて追従実験を行うことは有意義であると考えられる．

第9章　結論

本研究では、リアルタイム組込み機器向け仮想マシンモニタに適用可能な Lock Holder Preemption の解決方法を提案し、その有効性を検証した.

SMP GPOS と RTOS をマルチコアプロセッサ上で同時に実行し、固定優先度のスケジューリングを行うと LHP により GPOS 側のスループット性能が大きく低下する．我々はこの低下の度合いを調べ、解決のための 2 つの vCPU マイグレーションポリシーを提案し、実装と評価を行った.

Ondemand マイグレーションポリシーは GPOS の性能の向上を達成することが出来た．一方 Trap based マイグレーションポリシーは性能を向上させることに貢献出来ていない．これは、第 6 章で述べたように、RTOS の実行により IPI の到達が遅れるという Semantic Gap が原因であると考えられる.

Trap based マイグレーションポリシーは、割り込みの優先度の分割を行わないでも RTOS のレイテンシを低く保つことが可能な vCPU のスケジューリング手法なので、この IPI の到達の遅延を解決することにより、より効果的なリアルタイム組込み機器の開発手法の確立される可能性がある.

謝辞

本研究のテーマを始め、様々な機会を与えて下さり、研究の指導を含め幾度となく相談に乗って頂いた中島達夫教授に厚くお礼を申し上げます.

また、SPUMONE の最初の実装を行うことで研究の基盤を作って下さり、多くの助言を頂きました杵渕 雄樹氏に感謝致します.

最後に、研究が行き詰まった時の気晴らしに付き合ってくれた友人達にも感謝します.

[1] Hitoshi Mitake, Tsung-Han Lin, Hiromasa Shimada, Yuki Kinebuchi, Ning Li, and Tatsuo Nakajima. Towards co-existing of Linux and real-time OSes. In Proceedings of Ottawa Linux Symposium, 2011.

[2] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium (VM'04), 2004.

[3] Open Kernel Labs. OKL4 Microvisor. http://www.ok-labs.com/products/okl4-microvisor

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield. Xen and the art of virtualization. In Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003.

[5] Thomas Friebel and Sebastian Biemueller. How to Deal with Lock Holder Preemption. http://www.amd64.org/fileadmin/user_upload/pub/2008-Friebel-LHP-GI_OS.pdf

[6] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin Qumranet, Anthony Liguori. kvm: the Kernel-based Virtual Machine. In Proceedings of Ottawa Linux Symposium, 2007.

[7] VMware. http://www.vmware.com/

[8] Ingo Molnar. RT-patch. Index of /mingo/realtime-preempt:http://www.kernel.org/pub/linux/kernel/projects/rt/.

[9] Paul E McKenney. 'Real Time' vs. 'Real Fast': How to Choose? In *Proceedings of the Ottawa Linux Symposium*, 2008.

[10] François Armand and Michel Gien. A Practical Look at Micro-Kernels and Virtual Machine Monitors. In Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference, 2009.

[11] Christoffer Dall and Jason Nieh. KVM for ARM. In *Proceedings of Ottawa Linux Symposium*, 2010.

[12] The Xen ARM Project. http://xen.org/products/xen_arm.html

[13] Jupyung Lee. Private communication. 2011.

[14] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Hardware Support for Spin Management in Overcommitted Virtual Machines. In Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, 2006.

[15] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In Proceedings of Third International Conference on Distributed Computing Systems, 1982.

[16] VMware, Inc. VMware vSphere(TM) 4: The CPU Scheduler. in VMware(R) ESX(TM) 4 http://www.vmware.com/files/pdf/perf-vsphere-cpu_scheduler.pdf

[17] Orathai Sukwong and Hyong S. Kim. Is Co-scheduling Too Expensive for SMP VMs? In Proceedings of the ACM European conference on Computer systems (EuroSys '11), 2011.

[18] Victor Yodaiken. The RTLinux Manifesto. In Proceedings of the 5th Linux Expo, 1999.

[19] P. Mantegazza, E. L. Dozio, S. Papacharalambous. RTAI: Real Time Application Interface. In Linux Journal, volume 2000. Specialized Systems Consultants, Inc. Seattle, WA, USA.

[20] Jan Stoess, Jonathan Appavoo, Udo Steinberg, Amos Waterland, Volkmar Uhlig, and Jens Kehne. A light-weight virtual machine monitor for Blue Gene/P. In Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '11), 2011.

[21] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. 2010. The VMware mobile virtualization platform: is that a hypervisor in your pocket?. SIGOPS Oper. Syst. Rev. 44, 4 (December 2010).

[22] Jeremy Andrus, Christoffer Dall, Alex Vant Hof, Oren Laadan, and Jason Nieh. Cells: A Virtual Mobile Smartphone Architecture. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles, 2011.

[23] Jupyung Lee, Geunsik Lim, and Sang-bum Suh. Preemptibility-aware responsive multi-core scheduling. In Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11), 2011.

[24] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII), 2006.

[25] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10), 2010.

[26] Livio Soares and Michael Stumm. Exception-less system calls for event-driven servers. In Proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIXATC'11), 2011.

[27] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09), 2009.

[28] Gernot Heiser. Hypervisors for consumer electronics. In Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference (CCNC'09), 2009.

[29] Jochen Liedtke. Improving IPC by kernel design. In Proceedings of the fourteenth ACM symposium on Operating systems principles (SOSP '93), 1993.

[30] J. Liedtke. On micro-kernel construction. In Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP '95), 1995.

[31] Jochen Liedtke. Toward real microkernels. Commun. ACM 39, 9 (September 1996), 70-77.

[32] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In Proceedings of the thirteenth ACM symposium on Operating systems principles (SOSP '91), 1991.

[33] Steve Kleiman and Joe Eykholt. Interrupts as threads. SIGOPS Oper. Syst. Rev. 29, 2 (April 1995), 21-26.

[34] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. 2009. Sloth: Threads as Interrupts. In Proceedings of the 2009 30th IEEE Real-Time Systems Symposium (RTSS '09).

[35] M. Rozier , V. Abrossimov , F. Armand , I. Boule , M. Gien , M. Guillemont , F. Herrmann , C. Kaiser , S. Langlois , P. L 辿 onard , W. Neuhauser. CHORUS Distributed Operating Systems. Computing Systems, Vol. 1, No. 4, 1988.

[36] David Golub , Randall Dean , Alessandro Forin , All Dean , Ro Forin , and Richard Rashid. Unix as an Application Program. In USENIX Summer Conference, 1990.

[37] Mike Accetta , Robert Baron , William Bolosky , David Golub , Richard Rashid , Avadis Tevanian , and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In Proceedings of Summer Usenix. July, 1986.

[38] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible Atomic Regions for Real-Time Java. In Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05), 2005.

[39] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: getting beyond the limitations of virtual disks. In Proceedings of the 3rd conference on Networked Systems Design & Implementation, 2006.

[40] Philip M. Wells. Adapting to Dynamic Heterogeneity: Virtualization for the Multicore Era. PhD thesis, 2008.

[41] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the Fluke kernel. In Proceedings of the third symposium on Operating systems design and implementation, 1999.

[42] Victor Yodaiken. Against Priority Inheritance. FSMLabs Technical Report. http://www.yodaiken.com/papers/inherit.pdf. 2004.

[43] Jochen Liedtke, Hermann Härtig, Michael Hohmuth. OS-controlled cache predictability for real-time systems. In Proceedings of Real-Time Technology and Applications Symposium, 1997.

[44] Härtig, Hermann and Hohmuth, Michael and Liedtke, Jochen and Wolter, Jean and Schönberg, Sebastian. The performance of μ-kernel-based systems. In Proceedings of the sixteenth ACM symposium on Operating systems principles.