平成23年度 修士論文

仮想マシンを用いた IDSオフロードのための 資源管理機構

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 09M37028

新井 昇鎬

指導教員

千葉 滋 教授

平成23年1月26日

概要

仮想化技術の普及により、仮想マシン単位で貸し出しを行うクラウドサービスが普及している。一台の物理マシン上で複数の仮想マシンを動作している環境では、仮想マシン間の性能分離は非常に重要である。仮想マシン間の性能分離とは、各仮想マシンが設定された資源の制約を守り、他の仮想マシンの性能に影響を与えない環境を意味する。たとえば、最大 CPU 使用率 50 %という制約のある仮想マシンに対して CPU 資源の 50 %をしっかりと保証することができれば、この仮想マシンの性能は他の仮想マシンから分離できていることになる。

しかし、仮想マシンを利用して IDS のオフロードを行うと仮想マシン間の性能分離が難しくなる。IDS オフロードとは、監視対象の仮想マシンから侵入検知システム (IDS) を他の仮想マシンや仮想マシンモニタへ移動させ、監視対象の外部から監視する手法である。IDS が監視対象の仮想マシンの中で動作していないため、攻撃者の目に直接触れにくくなる。そのため、IDS 自体が攻撃されたり IDS のデータやログを改竄されることが難しくなり、セキュリティが向上する。しかし、既存の仮想マシンモニタでは仮想マシン単位の資源管理を行なっているので、オフロードした IDS を考慮することができない。IDS は監視対象の仮想マシンのために動作しているので、その消費した資源は監視対象の仮想マシンにカウントされるべきである。しかし、仮想マシン単位の資源管理ではその消費した資源はオフロード先の仮想マシンにカウントされる。このような資源管理では、オフロードした IDS と監視対象の仮想マシンの合計 CPU 使用率が監視対象の仮想マシンの CPU 制約を超えてしまう可能性がある。

本研究では IDS のオフロードを考慮するために仮想マシン単位ではなく、Resource Cage という新しい単位で資源管理を行うシステムを提案する。オフロードしたプロセスと監視対象の仮想マシンをひとまとまりにして資源管理を行う。仮想マシンとオフロードした IDS のプロセスが Resource Cage を持つことができる。オフロードした IDS を考慮するために、監視対象の仮想マシンの Resource Cage と IDS のプロセスの Resource Cage をグルーピングする。そして、そのグルーピングした Resource Cage に CPU 制約を与えることで IDS のオフロードを行なっても仮想マシン間の性能分離を守ることができる。

RC-Monitor と RC Credit Scheduler と RC-Limiter の 3 つの機構によって Resource Cage 単位の資源管理を可能にする。RC-Monitor はオフロードした IDS のプロセスの CPU 使用率を監視する。プロセスのページデレクトリのアドレスが保存されている CR3 レジスタを参照することで仮想マシンモニタレベルで CPU 使用率を記録する。RC-Limiter はオフロードした IDS に設定された CPU 制限を超えないように IDS を制御する。RC Credit Scheduler は仮想マシンが所属する Resource Cage を考慮しながら仮想マシンスケジューリングを行う。

Xen-4.1.0 上に Resource Cage を実装した。ドメイン 0 のバージョンは Linux 2.6.32.39 を使った。バケットフィルタリングを行う Snort とウイルスチェックソフトウェアの ClamAV をオフロードし、外部から httperf を利用して監視対象に負荷をかけて実験行った。 Snort と ClamAV をオフロードしても監視対象の仮想マシンと合わせて設定された CPU 制限を守れていることを確認した。

謝辞

本研究を進めるにあたり、研究の方針や論文の書き方について助言をしていただいた指導教員の千葉滋先生に心より感謝いたします。また、九州工業大学の光来先生にはシステムの設計・実装等の研究全般に渡り指導していただいき、深く感謝します。IBM 基礎研究所の堀江倫大氏と同期の別役浩平氏なしに私の研究生活はなかったと思えるくらい重要な存在でありました。心より感謝いたします。また、後輩の大久保君、金澤君、早船君、寺本君のおかげで最後の一年を楽しく乗り切ることができました。感謝しています。最後にとも研究活動を行った研究室の皆様に感謝します。

目 次

図 目 次

表 目 次

第1章　はじめに

本日、仮想化技術の普及により様々なクラウドサービスが普及している。クラウドサービスの一つである IaaS(Infrastructure as a Service) は、システムを構築する基盤となる仮想マシンなどをユーザに提供する。ユーザは様々なタイプの仮想マシンを利用することができ、その利用実績に応じて課金される。このような仮想マシン環境では、セキュリティや仮想マシン間の環境の分離などがとても重要となる。

仮想化環境のセキュリティを向上させる方法として、仮想マシンを利用した IDS オフロードがある。IDS オフロードとは、IDS を監視対象の仮想マシンの外で動作させ、別の仮想マシンや仮想マシンモニタなどから監視を行う。IDS オフロードを行うと IDS は監視対象の仮想マシンの中で動作していないため、攻撃者に直接攻撃されにくくなり、セキュリティが向上する。

しかし、IDS オフロードを行うと仮想マシン間の性能分離が難しくなる。仮想マシン間の性能分離とは、各仮想マシンが与えられた資源をしっかり使用できる環境である。つまり、ある仮想マシンの性能が他の仮想マシンの保証された性能に影響を与えないとき、仮想マシン間の性能が分離できていると言える。オフロードした IDS は監視対象の仮想マシンのために監視を行なっているので、IDS が消費した CPU 資源は監視対象の仮想マシンが消費したものと考えるのが望ましい。しかし、実際には IDS が動作しているオフロード先の仮想マシンが消費したことになる。そのため、監視対象の仮想マシンの CPU 消費量にオフロードした IDS の CPU 消費量を足すと、仮想マシンに設定された以上の CPU 資源を消費してしまう可能性がある。

本研究では、Resource Cage と呼ばれる新しい資源管理の単位を提案する。Resource Cage ではオフロードした IDS と監視対象の仮想マシンをひとつの資源管理単位とすることで、IDS オフロードを考慮した資源管理を行う。

Resource Cage は次の特徴をもつ。

- Resource Cage は VM 単位ではなく、Resource Cage 単位で資源管理を行う。現在、仮想マシンと仮想マシン内のプロセスが Resource Cage の単位になれる。また、オフロードした IDS のプロセスの Re-

source Cage と仮想マシンの Resource Cage をグルーピングすることで IDS のオフロードを考慮した資源管理が可能となる。

- Resource Cage は RC-Monitor と RC Credit Scheduler と RC-Limiter の 3 つから構成される。RC-Monitor はオフロードした IDS による CPU 使用量を監視する。RC Credit Scheduler はオフロードした IDS の Resource Cage も考慮して仮想マシンのスケジューリングを行う。RC-Limiter はオフロードした IDS が設定された CPU 制限を超えないように IDS のプロセスを制御する。

- Xen のドメイン 0 上からコマンドを入力することで Resource Cage の設定が可能である。各仮想マシンに対応する Resource Cage はその仮想マシン起動時に自動で作成される。プロセスに対応する Resource Cage はドメイン 0 上からコマンドを入力することで作成可能である。

本稿の残りは、次のような構成からなっている。第 2 章は 問題点と関連研究、第 3 章では、Resource Cage の提案、第 4 章では、Resource Cage のこれまでの実装、第 5 章では、Resource Cage を用いて IDS オフロード時の性能分離を確かめる実験、第 6 章では、まとめを述べる。

第2章　問題と関連研究

2.1　仮想化技術

2.1.1　仮想マシンモニタ

現在、Amazon Web Services[1] などのクラスド型のホスティングサービスや Openstack[5] や Eucalyptus[3] などのクラウドのオープンソフトウェアが普及している。それらの元となっているのが仮想化技術である。仮想マシンとはハードウェアを仮想化する技術である。仮想マシンを利用することにより、一つの物理マシンに複数のオペレーティングシステムを動作させることができる。たとえば、VMware という仮想化ソフトウェアを利用して Windows と Linux を同じ物理マシン上に動作させることができる。仮想化技術により、データセンターのサーバマシンの数や消費電力やスペースを削減できる。

物理マシンとオペレーティングシステムの間に仮想マシンモニタという層ができるので他の物理マシンにオペレーティングシステムを簡単に移動させることができる。この技術をマイグレーションという。マイグレーションにより、物理マシンの CPU に負荷がかかっているときは別の物理マシンに仮想マシンを移動させることができる。また、仮想マシンごとスナップショットを取ることにより、同じタイプの仮想マシンを簡単に作成することができる。

2.1.2　Xen Virtual Machine Monitor

代表的な仮想マシンモニタとして Xen[8] と KVM[4] がある。Xen はハイパーバイザー型 (図 2.3) の仮想マシンモニタでハードウェアの上で動作する。KVM はホスト型の仮想マシンモニタで Linux カーネルの一部として動作しているため、Xen と比べるとインストールが簡単である。KVM は Linux スケジューラやメモリ管理やデバイスドライバなどを共有している。

Xen にはドライバ 0 と呼ばれる特権ドメインが仮想マシンモニタ上に一つ存在して他の仮想マシンを管理している。ドメイン 0 以外の仮想マシ

ンのことをドメイン U と呼ぶ。ドメイン U はドメイン 0 を通してネットワークで通信したり、ハードウェアを利用したりする。

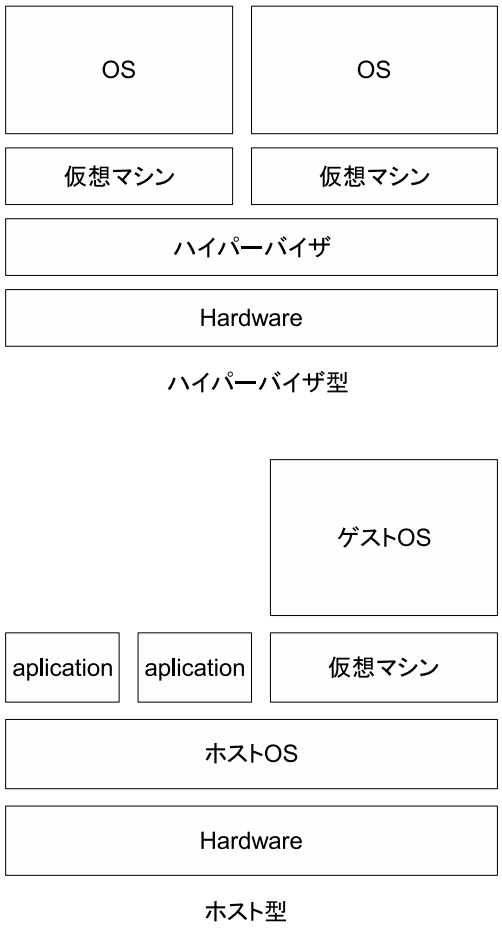

図 2.1: 仮想化の手法

2.1.3　Credit Scheduler

Xen の既存のスケジューラはクレジットスケジューラである (図 2.2)。クレジットスケジューラは SMP 環境にも考慮した仮想マシンスケジューラである。Xen の仮想マシンスケジューラは各仮想マシンに一つ以上の仮想 CPU を割り当てる。仮想 CPU とは仮想マシンが持つ仮想的な CPU

である。仮想 CPU を物理 CPU のランキューに入れてスケジューリングを行う。オペレーティングシステムではプロセスをランキューに入れてスケジューリングを行うが、Xen では仮想 CPU をランキューに入れてスケジューリングを行う。

クレジットスケジューラでは、各仮想マシンの仮想 CPU に配布する CPU 時間をクレジットとして表現する。10ms ごとに物理 CPU を使用している仮想 CPU のクレジットが減らされる。30ms ごとに各仮想マシンに設定された weight と cap を元にクレジットを計算し、仮想マシンが持つ仮想 CPU に配布する。weight は相対的な値で他の仮想マシンの weight の値と比較することで CPU 資源の使用割合を表す。cap は絶対的な値で最大 CPU 使用率を表す。

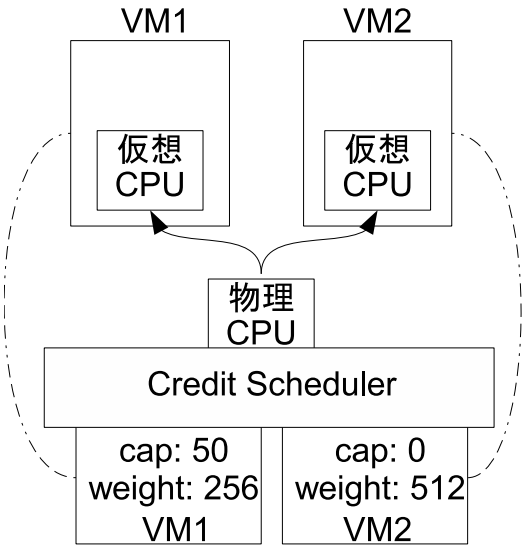

図 2.2: クレジットスケジューラ

2.2　仮想マシンを用いたセキュリティ機構オフロード

2.2.1　IDS のオフロード

侵入検知検知システム (IDS) は攻撃者の侵入に備えるためのセキュリティソフトウェアであり、侵入の兆候を検知して管理者に通知する。侵入者は IDS に検知されてしまうのを防ぐために、侵入後にまず IDS のデータやログを改竄することで IDS の無効化を試みる。このような攻撃に対

処するために、近年、仮想マシンを用いたオフロードが提案されている。IDS のオフロードとは IDS を監視対象の仮想マシンの外で動作させ、別の仮想マシンなどからオフロード元の仮想マシンを監視する手法である。

IDS のオフロードの研究として Livewire[11] がある。Livewire はホスト型 IDS の利点である監視対象ホストの内部状態を検知でき、またネットワーク型の利点である攻撃者に IDS 自体を攻撃されにくいというアーキテクチャを監視対象モニタを利用することにより実現している。監視対象仮想マシンから IDS をオフロードして、攻撃者が侵入した仮想マシンとは別の仮想マシンから監視対象仮想マシンの内部情報を仮想マシンモニタ経由で取得して検知する。

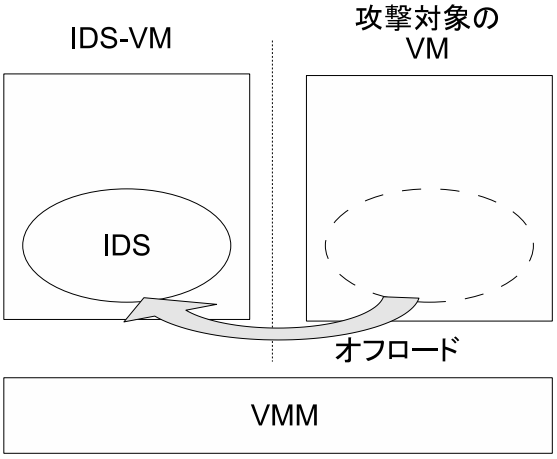

図 2.3: 仮想マシン (VM) を用いた IDS のオフロード

2.2.2　ファイヤーウォールのオフロード

ファイヤーウォールのオフロードの研究として xFilter[6] がある。xFilter は仮想マシン内にあるファイヤーウォールを仮想マシンモニタに移してパケットフィルタリングを行う。仮想マシンのメモリをマップすることで、仮想マシンモニタ内でも仮想マシン内と同様にきめ細かいパケットフィルタを行うことができる。

xFilter も仮想マシンモニタへオフロードを行なっているので、仮想マシ

ンへ侵入した攻撃者が xFilter を直接攻撃することは難しい。また、xFilter は仮想マシンモニタで動作しているので、上で動作する仮想マシンの情報をメモリ解析することで得られ、仮想マシン内で動いているファイヤーウォールと同じくらいきめ細かいパケットフィルタリングが可能となる。

2.2.3　アクセス制御のオフロード

OS では様々なアクセス制御で、ウイルスやクラッカーからファイルを保護している。しかし、OS 自体に脆弱性があると、これらのアクセス制御が機能しなくなる。たとえば、クラッカーがシステムに侵入して、OS の脆弱性を利用して管理者権限を奪うと、すべてのファイルにアクセスを許してしまう。

SAccessor[17] では、仮想マシンからアクセス制御を仮想マシンの外へオフロードして、仮想マシン内のファイルアクセスを管理する。SAccessor では仮想マシンを利用して作業 OS と認証 OS を動作させる。作業 OS が乗っ取られても、作業 OS 内の SAccessor の管理するファイルにはアクセスできない。

SAccessor にはファイルアクセスに関する 2 つのポリシーを記述できる。一つは空間に関するポリシーであり、もう一つは時間に関するポリシーである。空間に関するポリシーでは認証によってユーザーにアクセス許可されるファイルの範囲を限定することが可能である。時間に関するポリシーはアクセスの許可される期間を認証時に指定することができる

2.3　性能分離の問題

2.3.1　IDS オフロードによる性能分離の問題

VM を利用した IDS のオフロード (図 2.4) は IDS を監視対象の仮想マシンから他の仮想マシンや仮想マシンモニタに移動させて、監視対象の仮想マシンの外から監視する手法である。監視対象の仮想マシンの外で動作する IDS は、仮想マシンモニタの機能を利用することで監視対象の仮想マシンの情報を取得し、監視することができる。IDS をオフロードした先の仮想マシンは、外部に対するサービスを制限することで攻撃者に攻撃をうけないようにすることができる。例えば、ネットワーク型の IDS である Snort のオフロードを行う場合を考える。Snort を他の外部にサービスを提供しない仮想マシンや仮想マシンモニタにオフロードすると、Snort 自体のプロセスを停止されてしまうことを防ぐことができる。また、Snort が使用するルールやポリシも監視対象の仮想マシンの外に置かれるため、これらの改竄も防ぐことができる。

IDS のオフロードを行わければ仮想マシン間の性能を分離することができる。仮想マシン間の性能分離とは、ある仮想マシンの CPU 資源の消費が他の仮想マシンに保証された性能に影響を与えないことである。たとえば、仮想マシンが 2 つ存在しそれぞれの仮想マシンに最大 CPU 使用率 50 %を設定した場合は、各仮想マシンがその使用率を守り、CPU 資源の 50 %を使用できることが保証される。IDS が監視対象の仮想マシンの中で動作しているので、IDS がどれだけ CPU 資源を消費しても、割り当てられた 50 %の範囲内であり、他の仮想マシンに影響を与えることはない。

しかし、IDS を監視対象の仮想マシンの外へオフロードすると仮想マシン間の性能分離が難しくなる。オフロードした IDS は監視対象の仮想マシンのために動作しているので、IDS が消費した CPU 資源は監視対象の仮想マシンが消費したものを考えるのが望ましい。実際には、IDS はオフロードした先の仮想マシンの CPU 資源を消費することになる。たとえば、監視対象の仮想マシンが最大 CPU 使用率５０%を使用し、オフロードした IDS が 30 %の CPU を使用すると、合計で監視対象の仮想マシンのために 80 %の CPU 資源が利用されたことになる。もし、CPU 資源を５０%割り当てられた他の仮想マシンが存在していた場合、IDS のオフロードの影響により、その仮想マシンは保証された CPU 資源を利用できない可能性がある。

2.3.2　VM 単位の資源管理

IDS オフロードによる性能分離の問題は仮想マシンモニタが仮想マシン単位で CPU 資源の分配を行なっていることが原因である。例えば、Xen では各仮想マシンに CPU 使用率の上限と分配比率を設定するが、このような設定だけではオフロードした IDS を考慮して、監視対象の仮想マシンに CPU 資源を割り当てることができない。また、単に監視対象の仮想マシンとオフロードした IDS に別々に CPU を制限しても不十分である。オフロードした IDS に 20 %、監視対象の仮想マシンに 30 %という固定割り当てをすれば合計で 50 %に抑えることはできるが、オフロードした IDS が CPU 資源を使わないときに余った分を監視対象の仮想マシンに使わせることができない。

2.4　関連研究

2.4.1　SEDF-DC

Xen における I/O 処理を考慮した VM の性能分離を行うスケジューラとして SEDF-DC[12] がある。Xen ではドライバがドメイン 0 側のバック

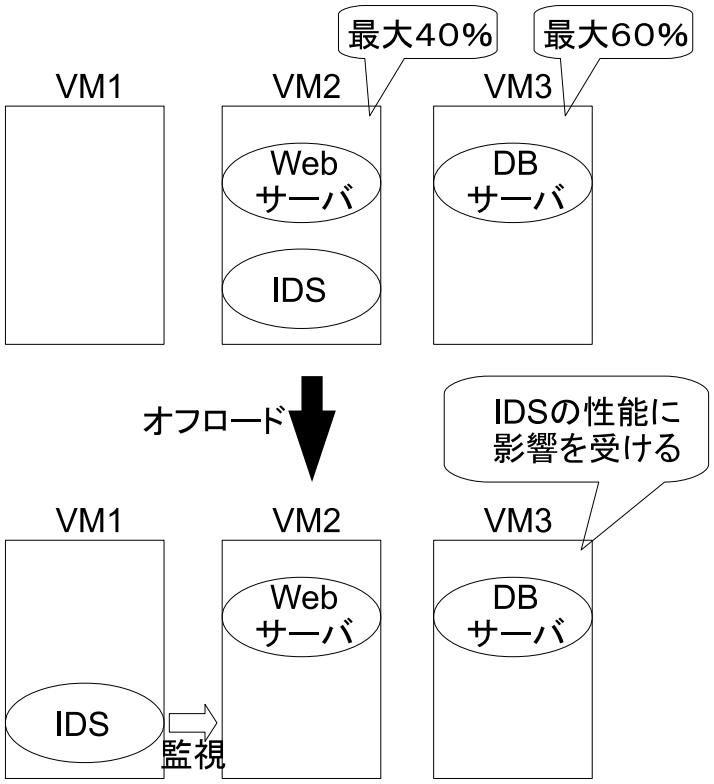

図 2.4: IDS オフロードの性能分離の問題

エンドとドメイン U 側のフロントエンドで分かれている。しかし、ドメイン 0 側のバックエンドドライバによる CPU 資源の消費は、I/O 処理を行なっているドメイン U に課金されない。そこで SEDF-DC では、ドメイン U のための I/O 処理で使用された CPU 使用量を測定し、そのドメイン U の使用量として課金する。

XenMon という機構が定期的にバックエンドドライバ内の処理の CPU 使用量を記録する。バックエンドドライバ内の各ドメインが使用した CPU 使用量を正確にカウントすることは難しい。そこで、XenMon ではドメイン 0 とドメイン U のドライバでの処理時メモリページ交換数からその CPU 使用量を推測する。また、ShareGuard という機構で各ドメインのための I/O 処理に使われるドメイン 0 の CPU 使用量を制限する。SEDF-DC というスケジューラで XenMon で得た情報を仮想マシンスケジューリングに反映する。本研究はオフローした IDS に注目しているが、SEDF-DC ではネットワーク I/O 処理に特化している。SEDF-DC ではパケット数から CPU 使用量を見積もり、パケットフィルタによって CPU 使用量を制

限する。

2.4.2 LRP

既存のネットワークアーキテクチャでは、負荷がかかったときに適切な優先順位、資源管理ができない。たとえば、パケット到着時のカーネル内のネットワーク処理による CPU 使用量はその処理に関係のあるプロセスにカウントされず、パケット到着時に動作していたプロセスにカウントされてしまう。また、パケット到着時処理がもっとも優先度が高いため、その処理が優先度の低いプロセスのためのものでも処理を行なってしまう。

LRP[10] はカーネル内のネットワーク処理に使われる CPU 消費量を適切なプロセスに課金できるようにしている。ネットワーク処理をよく行うプロセスがその処理の大部分をカーネルで行うことになる。しかし、カーネル内のネットワーク処理による資源の消費はそのプロセスのものとして課金されない。LRP では、カーネル内のネットワーク処理をプロセスのコンテキストで行うことで CPU 資源の消費を各プロセスに適切に課金する。本研究では、別仮想マシンに動作する IDS の CPU 資源の消費を監視対象仮想マシンに適切に課金する。

2.4.3 Resource Container

Resource Container[7] は LRP を拡張して OS に新しい資源管理の方法を導入している。アプリケーションの資源管理を行うとき、プロセス単位の管理が適切であるとは限らない。そこで Resource Container はプロセスとは異なる単位で CPU やメモリなどのシステムの資源を管理することができる。スケジューラはプロセスの属する Resource Container の資源の消費量など情報を利用してそのプロセスのスケジューリングを行う。Resource Container はオペレーティングシステムの資源管理の単位に注目しているが、本研究では仮想マシンモニタの資源管理単位に注目している。

2.4.4 Resource Pool

リソースプール (Resource Pool) では仮想マシン環境で階層構造の資源管理が可能となる。クラスタと呼ばれる論理的な仮想マシンの集まりを作成し、そのクラスタにルートの資源のプールの一部、子プールを与える。クラスタによって複数の仮想マシンをまとめて管理できるようになる。リソースプールは複数の仮想マシンを入れるための集合である。しかし、リ

ソースプールでは仮想マシン単位が最小単位であり、その中で動作するオフロードした IDS のプロセスを考慮することはできない。

2.4.5　Antfarm

Antfarm[13] は CR3 レジスタを利用することにより仮想マシンモニタ上からプロセスの状態を推測することができる。例えば、CR3 レジスタの値が変われば、プロセスのコンテキストスイッチが行ったと推測することができる。また新しい CR3 レジスタの値を取得したら新しいプロセスが生成されたと予測することができ、CR 3 レジスタに対応するページマッピングがなかった場合はプロセスが終了したと予測できる。この Antfarm を用いれば、仮想マシンからオペレーティングシステムのプロセススケジューリングの情報が得られたり、ゲストオペレーティングシステムのプロセスの数と Antfarm から取得できるプロセスの数を比較することでステルスなプロセスを発見できるようになる。

2.4.6　Task-aware Virtual Machine Scheduling

既存の仮想マシンスケジューラは、仮想マシンレベルのスケジューリングしか行えない。仮想マシン内のプロセスの情報が得られにくいので仮想マシンの中動作するタスクの種類を考慮したスケジューリングを行うことは難しい。そのため、I/O バウンドなタスクと CPU バウンドなタスクが混在するゲスト OS が存在すると、I/O バウンドなタスクのレスポンスが低くなることがある。

TAVS(Task-aware Vitual Machine Scheduling)[14] はゲスト OS 内の I/O バウンドなプロセスを考慮したスケジューリングを行う。Antram[13] と gray-box-knowledge[9] と呼ばれる技術を利用してプロセスの挙動から I/O バウンドなプロセスかどうかを仮想マシンモニタから推測する。ゲスト OS 内の I/O バウンドなプロセスと I/O 処理のイベントを関連づけて仮想マシンモニタ内に記録する。I/O 処理のイベントが発生したときに、その I/O 処理のイベントと対応するプロセスが存在する仮想マシンを一時的にブーストすることで I/O 処理の効率を上げる。

2.4.7　Monarch Scheduler

仮想化環境の普及により一台の物理マシンで複数仮想マシンを動作させることが多い。例えば、一台物理マシンに Web サーバと DB サーバ用の仮想マシンを動作させる。このような環境ではシステム全体でプロセス

の優先順位を考えることはサービスを提供する側にとって非常に重要である。仮想マシンモニタは仮想マシン単位で優先度を考慮できても、その中で動くプロセスの優先度は考慮できない。つまり、ある仮想マシンのプロセスと別の仮想マシンのプロセスの優先度をつけることは非常に難しい。

Monarch Scheduler[16] では、仮想マシンモニタから仮想マシン間にまたがるプロセススケジューリングを行う。仮想マシン内のオペレーティングシステムのプロセスのランキューを、仮想マシンモニタから操作することで実現している。ゲスト OS のスケジューリングと仮想マシンのスケジューリングを協調させることで、システム全体のプロセススケジューリングを可能とする。Monarch Schduler を利用するにあたってゲスト OS のカーネルを修正する必要はない

第3章　提案: Resource Cage

3.1　IDS のオフロードを考慮する資源管理システム

3.1.1　IDS のオフロードと性能分離の問題

セキュリティを向上させる方法として仮想マシンを用いた IDS のオフロードがある。IDS を監視対象の仮想マシンの外部から監視することで、IDS を攻撃者から直接攻撃されにくくなる。IDS だけではなく、様々なセキュリティ機構する研究がなされている。前章の関連研究で述べたように、ファイアウォールやファイルのアクセス制御もセキュリティを向上させるためにオフロードされている。たとえ監視対象の仮想マシンの外でセキュリティ機構が動作しても、仮想マシンモニタの機能を利用すれば監視対象の情報を取得できる。

仮想マシンを用いた IDS のオフロードはセキュリティを向上させるが、仮想マシン間の性能分離が困難になる。オフロードにより、監視対象の仮想マシンの外で IDS が動作するので攻撃者がシステム侵入後に IDS 自体を攻撃しにくくなる。このオフロードした IDS によって消費した CPU 資源は監視対象の仮想マシンのものとしてカウントされるべきである。しかし、実際にはオフロードした先の仮想マシンのものして消費されてしまう。このような資源管理では、オフロードした IDS の消費した CPU 資源と監視対象の仮想マシンの消費した CPU 資源の合計が管理が設定した CPU 資源の制約を超えてしまう可能性がある。たとえば、監視対象の仮想マシンに最大 CPU60 %の制約を与えた時、オフロードした IDS が 20 %の CPU を使うと合計で CPU 使用率が 80 %となり管理者の設定した制約を超えてしまう。オフロードした IDS と監視対象の仮想マシンに個別に CPU 制約を与えても不十分である。なぜなら、IDS が利用しなかった分を監視対象の仮想マシンが利用できないからである。

オフロードした IDS を考慮できない理由として仮想マシン単位の資源管理が挙げられる。仮想マシンモニタは仮想マシンを実行単位、かつ資源管理の単位としている。そのため、監視対象の仮想マシンの外で動作する IDS の消費した CPU 資源を監視対象の仮想マシンのものとしてカウントできない。仮想マシンモニタから別の仮想マシンの中で動作する IDS は直接参照することができないため、IDS によって消費した CPU をカウン

トすることは難しい。

3.1.2 Resource Cage

本研究では、オフロードした IDS と監視対象の仮想マシンを CPU 資源管理を単位とできるようにするための Resource Cage を提案する。Resource Cage はこれまでの仮想マシンモニタ内において仮想マシン単位で行われていた資源管理を拡張し、仮想マシンという実行単位から資源管理を切り離す。これにより、監視対象の仮想マシンとその外側で実行される IDS プロセスをひとまとまりとして扱うことができる。

Resource Cage に対して CPU 使用率の上限を設定することで、オフロードした IDS と監視対象の仮想マシンをひとまとまりとした性能分離を実現できる。たとえば、以下のコマンド例のようにして、オフロードした IDS の Resource Cage と監視対象の仮想マシンの Resource Cage をグルーピングする。そのグループの Resource Cage に 50 %の最大 CPU 使用率を設定し、IDS の Resource Cage に CPU を最大 30 %を使わせるといった制御ができる。オフロードしない環境では、Resource Cage は既存のシステムと同様に仮想マシン単位の資源管理を行う。

本論文では、オフロードする IDS としてパケット受信のイベントごとにチェックを行う Snort と、ウイルスチェックソフトウェアとしてシグネチャによるパターンマッチング方式でファイルシステムをチェックを行う ClamAV を選んだ。前述の関連研究のとおり、仮想マシンを利用したセキュリティ機構のオフロードの研究はいろいろ行われてる。ファイアウォールやファイルのアクセス制御のオフロードなどがその一例である。

```
user@zen:~# rc_init_process <PID>
0
user@zen:~# rc_group <PID> <Domain ID>
0
user@zen:~# rc_set_cap <Resource Cage ID (Group)> 50
0
user@zen:~# rc_set_cap <Resource Cage ID (Process)> 30
0
```

3.2 Resource Cage の構成

図 3.1 のように、3 つの機構により Resource Cage 単位の資源管理を実現している。RC-Monitor は仮想マシンモニタレベルでオフロードした IDS のプロセスの CPU 使用率を監視している。RC-Limiter はオフロー

ドした IDS のプロセスの動作を制御している。RC Credit Scheduler は Resource Cage の制約を考慮して仮想マシンのスケジューリングを行う。

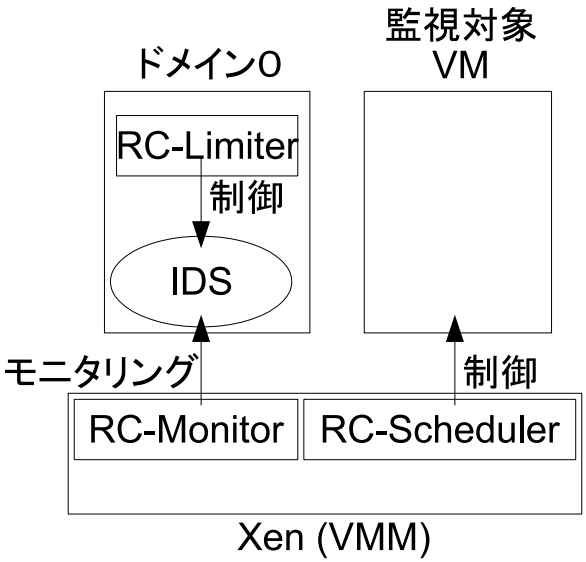

図 3.1: Resource Cage のアーキテクチャ

3.2.1　RC-Monitor

RC-Monitor は IDS プロセスの CPU 使用率を監視する。仮想マシンモニタで計測した IDS の CPU 使用率をその IDS が所属する Resource Cage に記録する。仮想マシンモニタから仮想マシン内のプロセスの CPU 使用率を取得するために、CR3 レジスタの変化に基づいてプロセスの実行時間を測定する。CR3 レジスタにはプロセスのページデレクトリの物理アドレスが格納されている。プロセスが切り替わる時には仮想マシンモニタが呼び出され、CR3 レジスタの値を次のプロセスのページデレクトリの物理アドレスの値に切り替える。IDS のプロセスの実行時間を記録するために、そのプロセスに対応するページデレクトリの物理アドレスが CR3 レジスタに格納されている時間を測定する (図 3.2)。

IDS プロセスの CPU 使用量はゲスト OS で測定するほうが一般的だと考えられるが、以下の 2 つの理由から Resource Cage では仮想マシンモニタを利用して計測している。一つはゲスト OS では仮想マシンの切り替えが考慮されていない場合があることである。たとえば、Linux の O(1) スケジューラの場合、仮想マシンがスケジューリングされていないときに

たまるタイマ割り込みがまとめて送られ、切り替えが直前、または切り替え直後に動作していたプロセスに次々と課金されてしまう。これは、O(1) スケジューラがタイマ割り込みのときに動いていたプロセスに課金する設計になっているためである。一方、Completely Fair Scheduler(CFS) を使った最近の準仮想化 Linux カーネルの場合、仮想マシンモニタから情報を利用することで VM の切り替えを考慮するようになっている。ただし、完全仮想化の場合は仮想マシンの切り替えを考慮するのは難しい [18]。

もう一つの理由は、正しく CPU を使用した時間が課金されていない場合があることである。O(1) スケジューラではタイマ割り込みの時にだけプロセスの実行時間に課金されるので、タイマ割り込みよりも前にプロセス切り替えが起こった場合には課金されない。そのため、I/O バウンドなプロセスはタイマ割り込みよりも短い間だけ動作していることが多いので、CPU を利用しているのに課金されないことがある。一方、CFS の場合はプロセス切り替え時に課金されるので上記のような問題は起こらない。

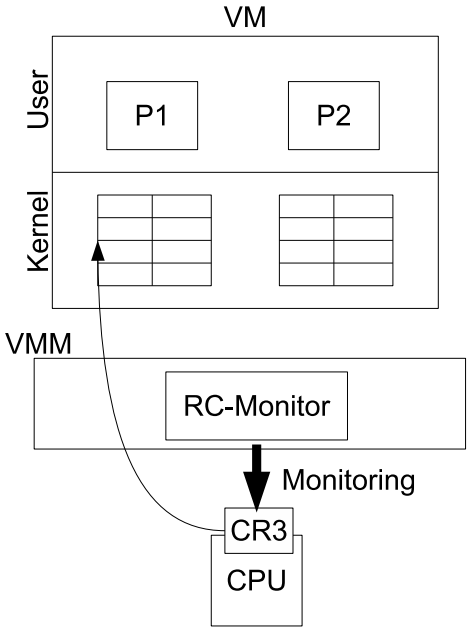

図 3.2: Resource Cage: RC-Monitor

3.2.2　RC Credit Scheduler

RC Credit Scheduler はオフロードした IDS の Resource Cage も考慮して仮想マシンのスケジューリングを行う。このスケジューラは Xen の

仮想マシンスケジューラであるクレジットスケジューラを改良して作成した。前述のとおり、クレジットスケジューラは Xen の現在のデフォルトの仮想マシンスケジューラである。クレジットスケジューラは各仮想マシンが持つ仮想 CPU(VCPU) に設定された cap、weight を元に計算したクレジットを配布することでスケジューリングを行う。

Resource Cage では仮想マシンではなく、Resource Cage に対して cap と weight を設定する。オフロード元の仮想マシンの cap はその Resource Cage のグループの cap とオフロードした IDS の CPU 使用率を元に計算される。例えば、Group に 50 という cap が設定されていてオフロードした IDS が 20 仮想マシンは 30 %まで使用できる。オフロードされていない環境では Resource Cage の cap の値だけを参照して既存のクレジットスケジューラと同様にスケジューリングされる。

3.2.3　RC-Limiter

監視対象の仮想マシンの動作は RC Credit Scheduler によって制御するが、オフロードした IDS のプロセスは仮想マシンの中で動作しているので仮想マシンモニタの仮想マシンスケジューラでは直接制御できない。そこで、Resource Cage ではオフロード先の仮想マシンに RC-Limiter という機構でオフロードした IDS のプロセスを制御する。このような制御する機構がない場合、オフロードした IDS だけでオフロードした IDS と監視対象の仮想マシンに与えれた CPU 資源を消費する可能性がある。

RC-Limiter は Resource Cage からオフロードした IDS の CPU 使用率を取得して Resource Cage に設定された上限値を超えていれば IDS の CPU 使用量を制限する。仮想マシンモニタで計測した実行時間を元に、オフロードした IDS のプロセスに対してシグナルを送ることで動作を制御する (図 3.3)。このプロセスを制限する方法は cpulimit[2] をベースにして実装している。SIGCONT シグナルでプロセスを動作させ、SIGSTOP プロセスでプロセスを一時的にストップさせる。

3.2.4　Resource Cage 用の コマンド

ドメイン 0 からコマンドを入力することによって Resource Cage を管理する。このコマンドから Xen のハイパーコールを通じて Resource Cage を操作する。ハイパーコールとは仮想マシンが仮想マシンモニタの機能を利用したいときに呼び出す API である。Resource Cage 用のハイパーコールを追加することによって、ドメイン 0 からのコマンドで Resource Cage 管理を可能にした。

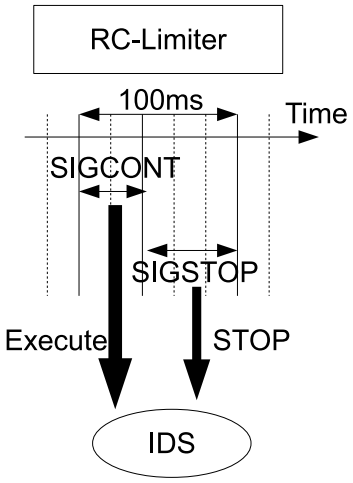

図 3.3: Resource Cage: RC-Limiter

rc_show

rc_show コマンドは現在の Resource Cage を出力する。現在の Resource Cage の構成や cap、weight の設定などが確認できる。以下は出力の例である。ドメイン 0 の Resource Cage の情報が出力されている。もう一つの例は ID が 1 の監視対象の仮想マシンとプロセス ID が 2000 のオフロードした IDS のプロセスのグループである。

```
Resource Cage
*******************************
* RC id:0, weight: 256, cap: 0
** Domain util: 0
*******************************

Resource Cage
*******************************
* RC id: 2001, tag: Group
** RC id: 1, weigth: 256, cap: 0
** RC id: 2000, util: 0
*******************************
* RC id:0, weight: 256, cap: 0
** Domain util: 0
*******************************
```

rc_init_process コマンド

rc_init_process コマンドはオフロードした IDS のプロセスのための Resource Cage を作成する。ドメイン用の Resource Cage 作成のコマンドはない。なぜならドメイン作成時に自動的に Resource Cage が作成されるためである。引数としてプロセス ID とオフロード元のドメインの ID が必要である。対応する ID が見つからない場合は何も作成されず、エラーを返す。作成された場合は 0 が出力される。

```
user@xen:/home/user$ rc_init_process <プロセス ID> <ドメイン ID>
0
user@xen:/home/user$
```

rc_group コマンド

rc_group コマンドはオフロードした IDS のプロセスと監視対象のドメインをひとまとまりにする。指定されたプロセスの Resource Cage とドメインの Resource Cage をまとめる Resource Cage を作成する。グループの Resource Cage の ID は IDS のプロセス ID と監視対象の仮想マシンの ID を足し合わせたものになる。グループの初期設定では、cap も監視対象の仮想マシンの cap とオフロードした IDS の Resource Cage の cap を足し合わせたものになる。引数としてオフロードしたプロセス ID と監視対象先の仮想マシンの ID を指定する。もしどちらかが見つからない、又はエラー時は-1 を返す。

```
user@xen:/home/user$ rc_group <プロセス ID> <ドメイン ID>
0
user@xen:/home/user$
```

rc_set_cap コマンド

rc_set_cap コマンドは ID によって指定された Resource Cage の cap を設定する。引数としては Resource Cage の ID と cap の値が必要である。もし cap の値が 0 ならば制限なしという設定になる。100 なら物理 CPU が一個分である。現在、マルチコア時の IDS オフロードには対応していない。

```
user@xen:/home/user$ rc_set_cap <Resource Cage ID> <cap の値>
0
user@xen:/home/user$
```

rc_set_weight コマンド

rc_set_weight コマンドはIDによって指定されたResource Cageのweightを設定する。引数としてはResource CageのIDとweight値が必要である。もしcapの値が0ならば制限なしという設定になる。100なら物理CPUが一個分である。現在、オフロード時のResource Cageのグループ内のweightによるshare schedulingは実現していない。

```
user@xen:/home/user$ rc_set_cap <Resource Cage ID> <capの値>
0
user@xen:/home/user$
```

第4章　実装

4.1　Resource Cage のデータ構造

Resource Cage は仮想マシンモニタの中でリストで管理されている。ドメイン (仮想マシン) もオフロードした IDS のプロセスも以下の rc 構造体で表現され、そのリストにつながれている。仮想マシンスケジューラによって、このリスト内の Resource Cage が参照され、仮想マシンの資源管理が行われる。Resource Cage のデータ構造は root/xen/include/xen/resource_cage.h に書かれている。これをインクルードすることで、他のソースコード内でも Resource Cage のデータや関数が使用可能となっている。

4.1.1　rc 構造体

rc 構造体 (図 4.1) は Resource Cage のメインとなるデータ構造である。RC Credit Scheduler はこの rc 構造体の情報を取得して仮想マシンのスケジューリングを行う。tag メンバーの値によって Resource Cage に入ってるエンティティを知ることができる。たとえば、tag の値が DOMAIN(=2) の場合、この Resource Cage にはドメインが入ってる。cap と weight は既存のクレジットスケジューラと同様で、cap は最大 CPU 使用率を表し、weight は他のドメインと比較するシェアを表す。IDS のオフロードが行われていない場合もこの rc 構造体の cap と weight の値を参照することで既存のスケジューラと同じように動作する。

4.1.2　rc_domain 構造体

rc_domain 構造体 4.1.2 はドメインの Resource Cage を表している。rc 構造体の tag が DOMAIN のときに共用体の u.domain のポインタは rc_domain 構造体を指す。ドメイン起動時に rc_domain 構造体は初期化され、対応する Resource Cage に保存される。また、この rc_domain 構造体は domain 構造体へのポインタも持っているのでこのドメインのシステム情報や所持している仮想 CPU をここから取得できる。後述する Future

```
/* 省力あり*/
struct rc {

  /* Domain ID or Process ID */
  int id;

  /* #define GROUP 0 */
  /* #define PROCESS 1 */
  /* #define DOMAIN 2 */
  uint16_t tag;

  /* Limit of CPU utilization */
  uint16_t cap;

  /* For share scheduling (Future work) */
  uint16_t weight;

  /* tag member indicates which union is included *
  union {
    struct rc_domain *domain;
    struct rc_process *process;
    struct rc_group *group;
  } u;

};
```

図 4.1: struct_rc 構造体

Work の実装方針では、仮想 CPU を Resource Cage で管理する。この domain 構造体から vcpu 構造体へのポインタを利用することで、実装が可能である。

4.1.3　rc_process 構造体

rc_process 構造体 4.1.3 はオフロードしたプロセスを表す。rc 構造体のメンバーである tag が PROCESS の場合に共用体 u.process はこの構造体を指す。address メンバーにはこのプロセスのページデレクトリのアドレスが保存される。この address を利用して仮想マシンモニタで CR3 レジ

```
/* 省力あり */
struct rc_domain {
  /* Domain ID *
  uint16_t domid;

  /* CPU utilization */
  int util;

  /* Pointer to struct domain */
  struct domain *dom;

}
```

スタを監視することで、IDS のプロセスの実行時間が計算できる。その実行時間が ptime_total で代入される。この ptime_total である一定の期間の間にどれくらい動いたかがわかり、CPU 使用率が計算できる。この rc_process 構造体は、ドメイン 0 からのコマンドによって rc_init_process() 関数が呼ばれ初期化される。

```
/* 省力あり */
struct rc_process {

  /* Value of CR3 register */
  /* Address of page directory */
  unsigned long address

  /* Process id */
  int pid;

  /* Period for execution */
  /* Used to caluculate utilization */
  s_time_t ptime_total;

  /* Pointer to the monitored domain */
  uint16_t domid;

}
```

4.1.4　RC-Monitor

RC_Monitor はオフロードしたプロセスの CPU 使用率を仮想マシンモニタレベルで計測する。定期的に int rc_calc_util(void) 関数を呼び出すことで CPU 使用率を計算し記録している。ある一定の期間にどれくらい rc_process 構造体の ptime_total メンバーが増加しているかを参照することで CPU 使用率を計算して、その util メンバーに記録する。

最近の Completely Fair Scheduling を利用している準仮想化 Linux カーネルの場合は仮想マシンレベルで CPU 使用率を正しく計測できるので Resource Cage 用のハイパーコール介して指定したプロセスの Resource Cage の util に直接記録できる。

4.1.5　RC Credit Scheduler

RC Credit Schedueler は前述のとおり既存のクレジットスケジューラを改良した。クレジットスケジューラでは図 4.2 のように各仮想マシンへ順番にクレジットを配布する (1)。仮想マシンに設定された cap や weight からクレジットが計算され (2)、その仮想マシンが持つ VCPU に均等に分配される (3)。このクレジットの分配は 30 ミリ秒ごとに行なっている。そして 10 ミリ秒ごとにタイマ割り込みが入り、そのとき物理 CPU を使用していた仮想 CPU のクレジットが減少する。また、クレジットがマイナスになった仮想 CPU は over という状態になり、まだクレジットが正の値の仮想 CPU は under という状態になる。この状態によってランキューの順序が変わる。ランキュー内では under の仮想 CPU が先になり、その後に over の仮想 CPU が続く。

RC Credit Scheduler では、(2) の cap と weight を利用してクレジットを計算する部分に変更を加えている。cap を利用して計算する部分では rc_schdule_cap を呼ぶ。この関数では、単にそのドメインに設定された cap の値を参照するのではなく、グルーピングされていた場合はグループの cap とグループ内のオフロードした IDS の Resource Cage の util を参照して計算する。例えば、グループに cap が 50 でオフロードした IDS が 20 %使用していた場合はこのドメインの cap の値は 30 となる。

4.1.6　RC-Limiter

RC_Limiter はオフロードした IDS の動作を制御して Resource Cage に設定された cap の値を超えないようにする。以下の式のように、CPU 使用率からプロセスを停止させる時間、動作させる時間を 100ms ごとに調整する。計算された動作時間だけプロセスを SIGCONT で動作させ、停

```
/* 省力あり */
static void csched_acct(void)
{
  /* (1) For each domain */
  list_for_each_safe(list_sdom, next_sdom, ...)
  {
    sdom = list_entry(iter_sdom, struct csched_dom, ...);

    credit_fair <- (2) calculated with cap and weight of this domain

    /* (3) divided by the number of VCPUs the domain has */
    credit_fair = (credit_fair + (sdom->active_vcpu_count - 1))
                  / sdom->active_vcpu_count;

    /* For each vcpu that the domain has */
    list_for_each_safe(list_vcpu, next_vcpu, ...)
    {
       svc = list_entry(iter_vcpu, struct csched_vcpu, ...)

       atomic_add(credit_fair, &src->credit);
    }
  }
}
```

図 4.2: Credit Sceduler

止時間だけ SIGSTOP でプロセスを停止させる。SIGCONT と SIGSTOP はプロセスに送るシグナルの種類である。このプロセスを制御する方法は cpulimit[2] をベースとして実装している。

$$\text{動作時間} = \min\left(100ms \times \frac{\text{上限値}}{\text{計測値}}, 100ms\right)$$

$$\text{停止時間} = 100ms - \text{動作時間}$$

4.2　CR3 レジスタの切り替えを監視

rc_log_ptime(図 4.3) はプロセス切り替え時に呼ばれる。具体的には CR3 レジスタが切りが変わるときである。引数の old_base_mfn には今まで動

作していたプロセスの CR3 レジスタの値が入り、new_base_mfn には新しく切り替わるプロセスの CR3 レジスタの値が格納している。

　この rc_log_ptime() 関数を CR3 レジスタが切り替わる場所に埋め込んだ。具体的には root/xen/arch/x86/mm.c にある int new_guest_cr3(unsigned long mfn) 関数と同じく root/xen/arch/x86/domain.c にある void context_switch(struct vcpu *prev, struct vcpu *next) 関数である。context_switch() 関数は仮想 CPU(VCPU) の切り替え時に呼ばれ、図 4.4 のようにして埋め込んだ。前の VCPU が所持している CR3 レジスタの値を pagetable_get_pfn() 関数で取得する。同様に新しく変わる VCPU が所持している CR3 レジスタの値も取得し、rc_log_ptime() 関数を呼ぶ。

```
/* 省力あり */
void
rc_log_ptime(unsigned long old_base_mfn, unsigned long new_base_mfn)
{
  struct rc_process *p =
  (struct rc_process)search_process(old_base_mfn);

  /* When target process is switched to another process */
  if (p != NULL && p->ptime_start != 0) {
    p->ptime_total += NOW() - p->ptime_start;
    p->ptime_start = 0;
  }

  p = (struct rc_process)search_process(new_base_mfn);

  /* When it is switched to target process*/
  if (target_p != NULL )
    p->ptime_start = NOW()
}
```

図 4.3: CR3 レジスタを監視する rc_log_ptime 関数

```
/* 省力あり */
void context_switch(struct vcpu *prev, struct vcpu *next)
{
  unsigned int cpu = smp_processor_id();
  cpumast_t dirty_mask = next->vcpu_dirty_cpumask;
  /* Resource Cage */
  unsigned long old_base_mfn, new_base_mfn

  ...

  local_irq_disable();

  /* Resource Cage */
  old_base_mfn = pagetable_get_pfn(prev->arch.guest_table);
  new_base_mfn = pagetable_get_pfn(next->arch.guest_table);
  rc_log_ptime(next);

  local_irq_eable();

  ...
}
```

図 4.4: VCPU を切り替える context_switch 関数

4.3　Resource Cage 作成時の動き

4.3.1　仮想マシンモニタ起動時

Xen 起動時に Resource Cage の初期化が行われる。各 Resource Cage のつなぐリストの初期化が行われる。正しく処理が行われると Xen カーネルには以下のようなメッセージが表示される。実際には int resource_cage_start(void) 関数が呼ばれ、各 Resource Cage をつなぐリストである rc_pri.active_rc が初期化される。

```
(XEN) Using scheduler: SCP Credit Scheduelr
(XEN) Resource Cage enabaled
(XEN) Detected 2798.097 MHz processor
```

4.3.2　仮想マシン起動時

仮想マシン起動時には仮想マシン用の Resource Cage の初期化が行われる。具体的には rc_init_domain() 関数が呼ばれる。weight や cap の設定や domain 構造体へのポインタを所持する。このポインタを所持することで Resource Cage から VCPU などのドメインに関する情報が得られる。無事に初期化が行われると、以下のようなメッセージが Xen カーネルから出力される。

```
(XEN) Initialization of RC of Domain 0
(XEN) is completed
```

4.3.3　プロセス登録時

プロセスを Resource Cage に登録するときは rc_init_process() 関数が後述するハイパーコールを介して呼び出される。このプロセスのページデレクトリの物理アドレスの情報や weight や cap の設定やどのドメインのために動作しているかといった情報が記録される。ページデレクトリの物理アドレスは CPU 実行時間を計測するときに用いられる。無事に初期化が行われると、以下のようなメッセージが Xen カーネルから出力される。

```
(XEN) RC for PROCESS 2789
(XEN) is completed
```

4.4　Resource Cage へのインターフェイス

4.4.1　ハイパーコールの追加

ハイパーコールとは仮想マシンが仮想マシンモニタの機能を利用したいときに使用する API である。ハイパーコールはオペレーティングシステム内のプロセスがオペレーティングシステムの機能を使用するときに呼び出すシステムコールに似ている。たとえば、ドメイン 0 が新しいドメイン U を作成したいとき、仮想マシンモニタにドメイン作成用のハイパーコールを呼ぶことで仮想マシンモニタがその新しい仮想マシンを作成する。

Resource Cage 用のハイパーコールとして図 4.5 の do_rc_op(int cmd), struct xen_domctl_rc *arg) を作成した。do_rc_op ハイパーコールは xenctrl.h で宣言されていて、第一引数の cmd に以下の値が入る。また、第ニ引数には構造体は図 4.6 のように定義されていてる。第 1 引数の種類によって、共用体に入る構造体が変化する。たとえば、RC_GROUP が第一引数に入っていた場合は、共用体には xen_rc_group 構造体が代入される。

```
#define RC_SHOW 1
#define RC_SET_UTIL 2
#define RC_REMOVE_PROCESS 3
#define RC_SET_CAP 4
#define RC_GROUP 5
#define RC_INIT_PROCESS 6
#define RC_GET_UTIL 7
#define RC_CALC_CR3 8
#define RC_GET_CAP 9
```

ドメイン 0 から xc_rc_op() 関数を呼ぶと以下のような流れで do_rc_op() 関数まで辿りつく。引数は省略してある。4 の hypercall() 関数以降に仮想マシンモニタ内のハイパーコールテーブルに飛び、Resource Cage 用の do_rc_op() ハイパーコールを呼び出す。すべての関数の引数にある xc_interface は/proc/xen/privcmd のファイルディスクリプタを保持しており、ここにあるハイパーコールテーブルはすべての仮想マシンで共有される。

1. xc_rc_op(xc_interface *xch, int cmd, struct xen_domctl_rc *rc_arg)
2. do_domctl(xc_interface *xch, struct xen_domctl *domctl)
3. do_xen_hypercall(xc_interface *xch, struct xen_domctl *domctl)
4. hypercall(xc_interface *xch, xc_osdep_handle h, privcmd_hypercall_t hypercall)

前章で紹介した Resource Cage 用のコマンドはそれぞれ do_rc_op() 関数の cmd に対応している。たとえば、rc_init_process コマンドは前述した関数を呼び出し、do_rc_op() 関数の rc_init_process() 関数にたどり着く。そしてこの関数の中でプロセス用の rc 構造体を作成、初期化してメインの Resource Cage のリストに挿入する。

```
int
do_rc_op(int cmd, struct xen_domctl_rc *arg)
{
  int ret = 0;
  switch(cmd)
  {
    /* Create Resource Cage for process */
    case RC_INIT_PROCESS:
    {
      ...
      ret = rc_init_process(mfn, pid, id, weight, cap);
      break;
    }
    /* Set cap to Resource Cage */
    case RC_SET_CAP:
    {
      ...
      ret = rc_set_cap(rc_id, cap);
      break;
    }
    /* Show status of Resource Cage */
    case RC_SHOW:
    {
      ...
      ret = rc_show();
      break;
    }
    /* Group Resource Cage */
    case RC_GROUP:
    {
      ...
      ret = rc_group_rc(rc1, rc2);
      break;
    }
    ...
  }
}
```

図 4.5: Resource Cage のハイパーコール: do_rc_op 関数

```
struct xen_domctl_rc {
  union {
    struct xen_rc_set_cap {
      int rc_id;
      int cap;
    }
    struct xen_rc_init_p {
      unsigned long mfn;
      int pid;
      int weight
      int cap;
    }
    struct xen_rc_group {
      int rc_id1;
      int rc_id2;
    }
    ...
  }
}
```

図 4.6: do_rc_op 関数の第 2 引数の構造体の定義

第5章　実験

IDS をオフロードしたときの仮想マシン間の性能分離を確かめる実験を行った。IDS としてイベント駆動型の Snort とウイルスチェックソフトウェアである ClamAV を Xen 環境でドメイン 0 へオフロードした。また、プロセスの CPU 使用率を計測するのに、仮想マシンモニタで CR3 レジスタを利用する方法とゲスト OS 内の proc ファイルシステムを利用する方法を比較する実験を行った。

CPU は Intel Core i7 2.80GHz のコアを一つ、メモリは 8GB、NIC はギガビットイーサ、HDD は SATA 1TB を搭載したマシンで実験を行った。Xen のバージョンは 4.1.0、ドメイン 0 の OS は Linux 2.6.32.39、ドメイン U の OS は Linux 2.6.16.33 であった。

5.1　Snort

Snort にドメイン 0 の仮想インターフェイス vif を監視させることで、ドメイン 0 へのオフロードを可能にした。Xen ではドメイン 0 が仮想マシンのネットワークの送受信を仲介している。これは、ドメイン 0 のみ物理 NIC にアクセスできるためである。ドメイン 0 は各仮想マシンがネットワーク通信できるようにそれぞれに対して vif を提供している。例えば、仮想マシン内のネットワークインターフェイスである eth0 に対応する vif1.0 がドメイン 0 内に作成される。

Resource Cage を利用する場合、オフロードした IDS と監視対象の仮想マシンをグルーピングした Resource Cage に対して最大 CPU 使用率 5030 %の最大 CPU 使用率を設定した。ドメイン 0 には CPU 使用率に関する制限を行わなかった。監視対象の仮想マシンを Web サーバとして動作させ、外部の別の物理マシンから httperf[15] を使用して Web サーバに負荷を与えた。httperf のリクエストレートは毎秒 4000 リクエストとした。外部のマシンは Intel Core i7 2.67GHz、メモリは 12GB、NIC はギガビットイーサネットであった。

図 5.1 と図 5.2 は、Snort をオフロードして Resource Cage を使用しなかった場合と利用した場合である。結果からわかるように Resource Cage を使用した場合はオフロードした Snort と監視対象の仮想マシンの合計

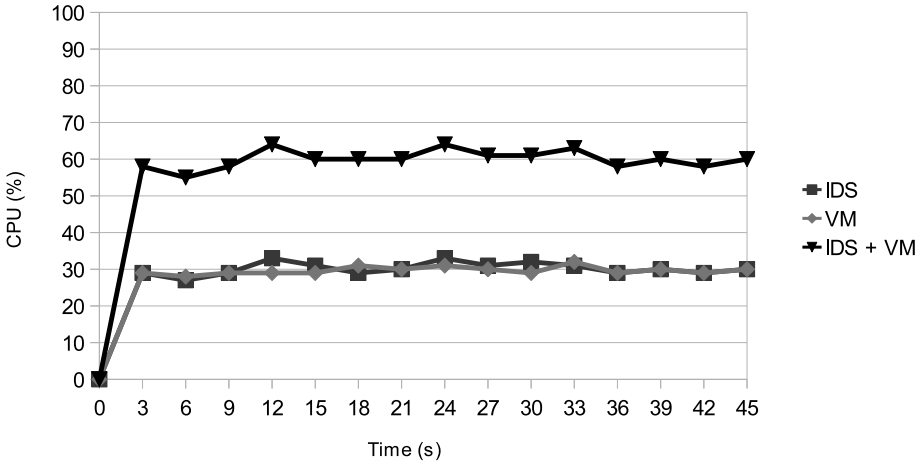

図 5.1: Snort のオフロード: Resource Cage なし

CPU 使用率が Resource Cage の制限である 50 %を守ることができている。しかし、Resource Cage を使用しない場合は合計 CPU 使用率が 50 %を大幅に超えている。監視対象の仮想マシンの CPU 使用率は 50 %以下であるが、オフロードした IDS はその制限と無関係にドメイン 0 の CPU 資源を消費した。

図 5.3 は監視対象の仮想マシンで動作せた Web サーバのスループットである。単純にオフロードした場合はオフロードした Snort の分だけ監視対象の仮想マシンが使用できる CPU 資源が増加する。その結果、オフロードしない場合よりも、Web サーバのスループットが増加している。

5.1.1　httperf について

httperf は Web サーバのパフォーマンスを測定するツールである。1 秒当たりのリクエスト数や総コネクション数を測定して使用することで、Web サーバに一定の負荷を与えることができる。実験のでは以下のコマンドを実行している。

- httperf –server 192.168.16.31 -port 80 -uri /../../../etc/passwd -rate 4000 -num-conn 100000 -num-call 1 -timeout 5

このコマンドは、192.168.16.31 というホストの 80 番ポートにアクセスして/etc/passwd ファイルを要求する。総コネクション数が 100000 に達するまで、1 秒当たり 4000 リクエストで負荷を与える。

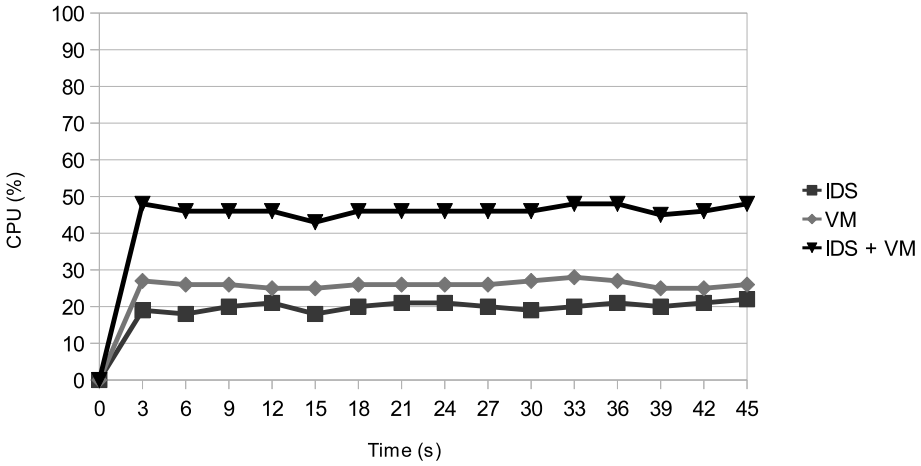

図 5.2: Snort のオフロード: Resource Cage あり

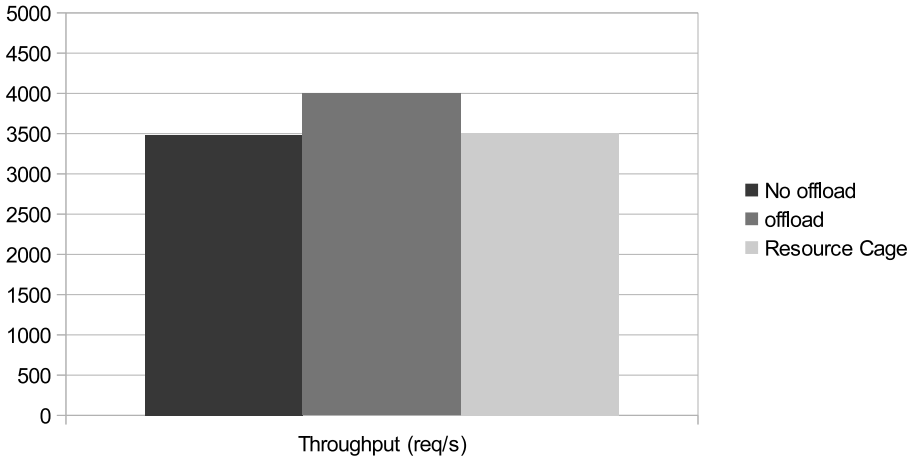

図 5.3: Snort のオフロード: ウェブサーバのスループット

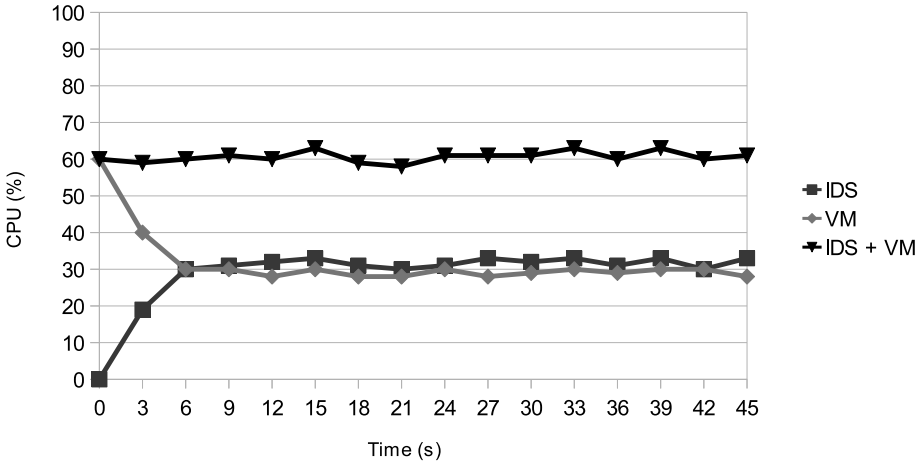

図 5.4: ClamAV のオフロード: Resource Cage あり

5.2　ClamAV

ClamAV に監視対象の仮想マシンが使用する仮想ディスク内のファイルシステムを検査することで、ドメイン 0 へのオフロードを実現した。Xen では、イメージファイルを仮想ディスクとして仮想マシンを作成することができる。ループバックデバイスを利用して、このイメージファイルを読み込み専用でマウントすることにより、ドメイン 0 上で監視対象の仮想マシンのファイルを参照することが可能になる。ただし、ドメイン 0 の OS が仮想ディスク内のファイルシステムを認識できなければならない。今回の実験では監視対象の仮想マシンでも Linux 標準の ext3 ファイルシステムを用いた。

オフロードした ClamAV と監視対象の仮想マシンをグルーピングした Resource Cage に 60 %の CPU 使用率を設定した。また、RC-Limiter で ClamAV は最大 30 %までしか CPU を利用できないように制限した。監視対象の仮想マシンには無限ループするプログラムが動作していて CPU 資源をできるだけ使うような環境になっている。

図 5.4 は、ClamAV をオフロードして Resource Cage を使用した場合である。オフロードした ClamAV と監視対象の仮想マシンの合計 CPU 使用率が 60 %の制限を守れていることが確認できる。

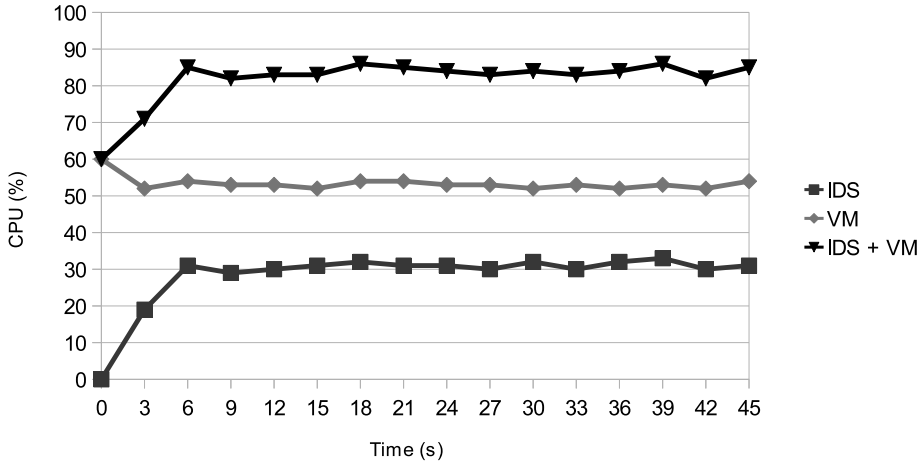

図 5.5: ClamAV のオフロード: Resource Cage なし

5.3　CPU 使用率の計測方法の比較

以下の実験は AMD Athlon 2.2GHz、メモリは 2GB、NIC はギガビットイーサネット、HDD は SATA 250GB を搭載したマシンで行った。Xen のバージョンは 3.3.0、ドメイン０の OS は Linux 2.8.18-8、ドメイン U は Linux 2.6.16.33 であった。

5.3.1　無限ループするプログラム

ある仮想マシンの中で動作する無限ループするプログラムの CPU 使用率を計測するのに仮想マシンモニタ内で CR3 レジスタを利用した場合と proc ファイルシステムを利用した場合を比較した (図 5.6)。とヒュウで別の仮想マシンで同じように無限ループするプログラムを動作した。２つの仮想マシンでそれぞれ CPU バウンドなプロセスが動作しているので、５０%ずつの CPU 使用率になるはずである。CR3 レジスタを利用して計測する方法では別の仮想マシンで無限ループが動作すると CPU 使用率が正しく５０%に下がる。しかし、proc ファイルシステムを利用して計測する方法では、O(1) スケジューラを用いた古いカーネルは別の仮想マシンで無限ループが動作しても CPU 使用率がほぼ１００%のままで下がらない。CR3 レジスタを利用して計測すれば正しく計測でき、オペレーティングシステムのスケジューラにも依存することはない。

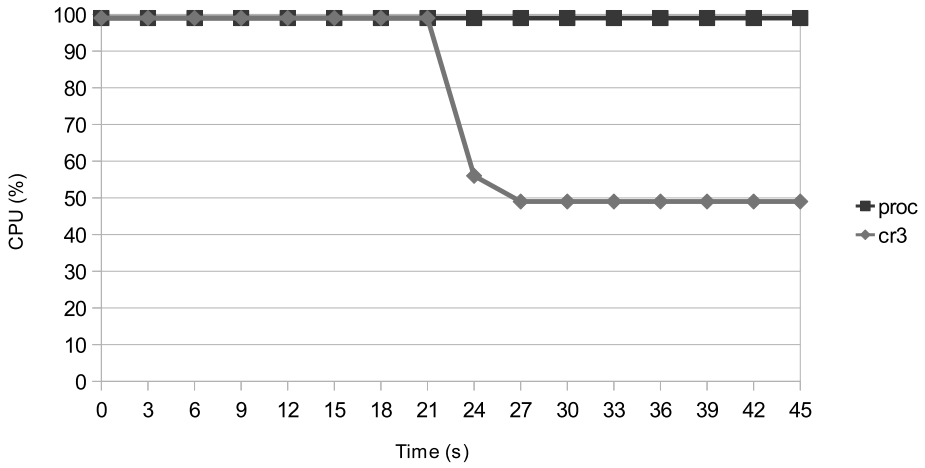

図 5.6: CPU 使用率の測定の比較 (proc と CR3 レジスタ)

5.3.2　Snort と Tripwire

図 5.7 と図 5.8 はある仮想マシンの中で動作する Snort と Tripwire の CPU 使用率を CR3 レジスタと proc ファイルシステムを利用して計測した結果である。これは先述したように O(1) スケジューラでは I/O バウンドなプロセスが使用した CPU 使用量を適切に課金できないことがゲインである。CR3 レジスタを利用すればオペレーティングシステムのスケジューラに依存することなく、課金できることがわかる。

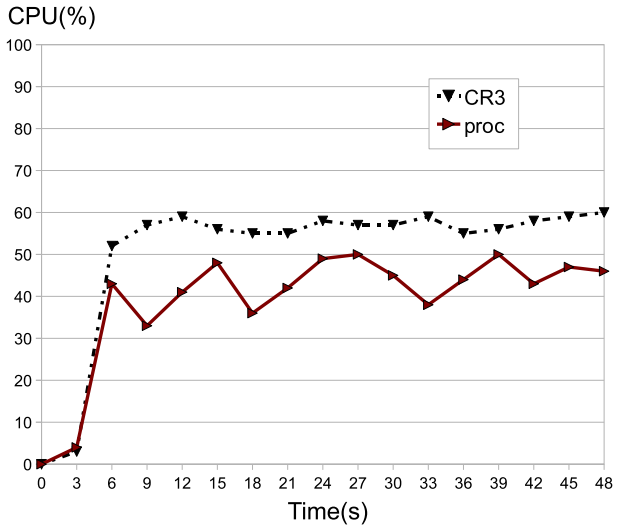

図 5.7: Snort の CPU 使用率の測定の比較 (proc と CR3 レジスタ)

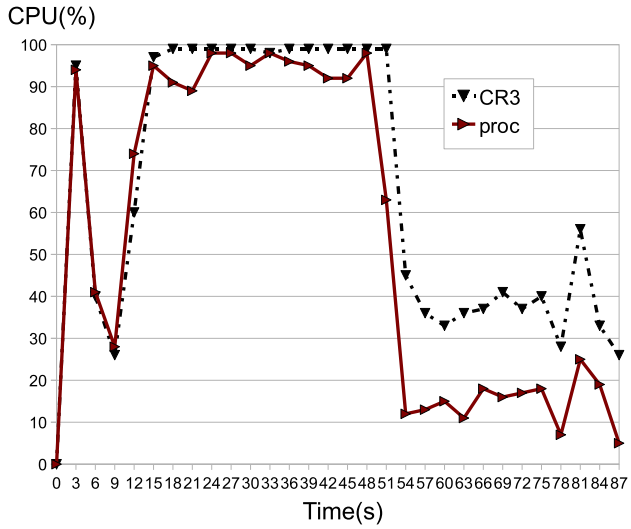

図 5.8: Tripwire の CPU 使用率の測定の比較 (proc と CR3 レジスタ)

第6章　まとめ

6.1　IDS オフロード時の仮想マシン間の性能分離の実現

仮想マシンを利用して IDS のオフロードを行うとセキュリティが向上する。IDS が監視対象の仮想マシンの外で動作するため、攻撃者に直接攻撃されにくくなる。しかし、オフロードした IDS が別の仮想マシンで動作するため、仮想マシン間の性能分離が難しくなる。本研究では、IDS オフロードを考慮する資源管理システムである Resource Cage を提案する。本システムでは仮想マシン単位ではなく、Resource Cage 単位で資源管理を行う。オフロードした IDS の Resource Cage と監視対象の仮想マシンの Resource Cage をグルーピングする。そのグルーピングした Resource Cage に CPU 制約を与えること仮想マシン間の性能分離を実現する。

IDS として Snort とウイルスチェックソフトウェアとして ClamAV をオフロードして性能分離を確かめる実験を行った。オフロードしたプロセスと監視対象の仮想マシンの CPU 使用率を計測して、グルーピングした Resource Cage に設定されている最大 CPU 使用率の制約を守れていることを確認した。

6.2　Future Work

オフロードした IDS と監視対象の仮想マシンをグルーピングした Resource Cage 内のシェアスケジューリングが実現できていない。現在の実装では IDS のプロセスと監視対象の仮想マシンが違うレベルで動作しているので、シェアスケジューリングを行うのは難しい。そこで、Resource Cage のもう一つ別の実装方法が考えられる。オフロードした IDS のプロセスのために仮想 CPU を新しく用意する。Linux の CGroup という機能を利用して IDS とその仮想 CPU を結びつける。こうすれば、オフロードした IDS のプロセスを仮想マシンと同じレベルでスケジューリングできる可能性がある。

実際にこの実装方法でプロトタイプを作成した。オフロードしたプロセスと結びつけた仮想 CPU を監視対象の仮想マシンが使用している物理

CPU のコアと共有させ、無限ループを利用して簡単な実験を行った。別の仮想マシンで動作するオフロードした無限ループとオフロード元の仮想マシンが同じ物理 CPU のコアで動作して期待した割合でシェアできていることを確認した。Resource Cage 単位で資源管理を行うには、仮想マシンごとにスケジューリングを行なっているクレジットスケジューラを Resource Cage ごとにスケジューリングを行うように改造する必要がある。また、各仮想マシンで管理されている仮想 CPU を Resouce Cage で管理する必要もある。

参考文献

[1] *Amazon Web Services*, http://aws.amazon.com/.

[2] *cpulimit: CPU Usage Limiter for Linux*, http://cpulimit.sourceforge.net.

[3] *Eucalyptus Cloud Computing Software*, http://www.eucalyptus.com/.

[4] *KVM: Kernel Based Virtual Machine*, http://www.linux-kvm.org/page/.

[5] *openstack*, http://openstack.org/.

[6] Azumi, T., Kourai, K. and Chiba, S.: A VMM-level Packet Filter Preventing Only Stepping-stone attacks, *IPSJ SIGOS* (2011).

[7] Banga, G., Druschel, P. and Mogul, J. C.: Resource containers: a new facility for resource management in server systems, *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, Berkeley, CA, USA, USENIX Association, pp. 45–58 (1999).

[8] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, ACM, pp. 164–177 (2003).

[9] Burnett, N. C., Bent, J., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Exploiting Gray-Box Knowledge of Buffer-Cache Management, *USENIX Annual Technical Conference, General Track*, pp. 29–44 (2002).

[10] Druschel, P. and Banga, G.: Lazy Receiver Processing(LRP): A Network Subsystem Architecture for Server Systems, *Operating Systems Design and Implementation*, pp. 261–275 (1996).

[11] Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Network and Distributed Systems Security Symp*, pp. 191–206 (2003).

[12] Gupta, D., Cherkasova, L., Gardner, R. and Vahdat, A.: Enforcing performance isolation across virtual machines in Xen, *Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, New York, NY, USA, Springer-Verlag New York, Inc., pp. 342–362 (2006).

[13] Jones, S. T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Antfarm: tracking processes in a virtual machine environment, *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, Berkeley, CA, USA, USENIX Association, pp. 1–1 (2006).

[14] Kim, H., Lim, H., Jeong, J., Jo, H. and Lee, J.: Task-aware virtual machine scheduling for I/O performance., *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, New York, NY, USA, ACM, pp. 101–110 (2009).

[15] Mosberger, D. and Jin, T.: httperf—a tool for measuring web server performance, *SIGMETRICS Perform. Eval. Rev.*, Vol. 26, No. 3, pp. 31–37 (1998).

[16] Tadokoro, H., Kourai, K. and Chiba, S.: A Secure System-Wide Process Scheduler across Virtual Machines, *Proceedings of the 2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*, PRDC '10, Washington, DC, USA, IEEE Computer Society, pp. 27–36 (2010).

[17] Takizawa, Y., Kourai, K., Chiba, S. and Yanagisawa, Y.: Secure File Access Control for Desktop PCs, *IPSJ Transactions on Advanced Computing Systems*, pp. 124–135 (2008).

[18] VMware, Inc: *Timekeeping in VMware Virtual Machines*, `http://www.vmware.con/pdf/vmware\_timekeeping.pdf` (2008).

付 録A　Resource Cageの使用例

ここでは、Resource Cage の利用例を説明する。Xen 上のドメイン 0 から以下のようにして利用する。まずは Xen カーネルからのメッセージを参照して Resource Cage が動作していることを確認する。

```
 __  __            _  _     _   ___
 \ \/ /___ _ __   | || |  / | / _ \
  \  // _ \ '_ \  | || |_ | || | | |
  /  \  __/ | | | |__   _|| || |_| |
 /_/\_\___|_| |_|    |_|(_)_(_)___/

(XEN) Xen version 4.1.0 (sungho@)
...
(XEN) Using scheduler: SMP Credit Scheduler (credit)
(XEN) Resource Cage enabled
(XEN) Detected 2798.090 MHz processor.
...
(XEN) Initialization of RC of Domain 0
...
```

次に PS コマンドなのでオフロードした IDS のプロセスを確認する。プロセス ID を確認したら、rc_init_process コマンドでオフロードした IDS のための Resource Cage を作成する。つぎに監視対象の仮想マシンの ID と xm list コマンドなどを利用して取得したら rc_group コマンドで Resource Cage をグルーピングする。

```
user@xen:~$ sudo rc_init_process <Process ID>
0
user@xen:~$ sudo rc_group <Process ID> <Domain ID>
0
user@xen:~$ sudo rc_set cap <Resource Cage ID> 60
0
```

IDS のオフロードのための Resource Cage を作成し CPU 制約を設定し終えたら、rc_show コマンドで現在の設定を確認する。オフロードした IDS

のプロセスの Resource Cage と監視対象の仮想マシンの Resource Cage がグルーピングされていて、CPU 制約が設定されていることを確認する。

```
**********************
* RC id: 2396, tag: Group, weight: 256, cap: 60
** RC id: 1, weigt: 256, cap: 0
** RC id: 2395, util: 0
**********************
* RC id: 0, weight: 256, cap:0
* Domain util:0
**********************
```