ベアメタルクラウドにおける
物理マシン管理に関する研究

２０１８年　９月

深井　貴明

ベアメタルクラウドにおける
物理マシン管理に関する研究

深井　貴明

システム情報工学研究科

筑波大学

２０１８年　９月

概要

近年，仮想マシンだけではなく物理マシンを提供する IaaS クラウドであるベアメタルクラウドを提供する事業者が現れている．物理マシンはマシン仮想化による性能劣化が存在しないため性能面に優れている．また，物理マシン上ではユーザーがハードウェアの機能を直接利用することで，処理のさらなる高速化を実現できる．その一方で，ベアメタルクラウドはマシンの仮想化で実現していたマシン管理を行いにくくなっている．

本稿では， ベアメタルクラウドでマシン管理機能を提供するソフトウェアシステムである物理マシンモニタを提案する．物理マシンモニタは物理マシンの性能面および機能面の利点を維持しつつ，物理マシンを管理する機能を提供する．また，物理マシンは OS から独立したシステムとして動作することで，ユーザーの OS に依存しないマシン管理機能を提供する．このために，OS が物理ハードウェアを直接利用して動作するよう，物理マシンモニタはハードウェアを仮想化しない設計とした．

仮想マシンで実現されているマシン管理機能は，仮想マシンモニタが仮想ハードウェアを細かく制御することで実現していた．このため物理マシンモニタで仮想マシンと同様の機能を実現するには，物理マシンモニタで物理ハードウェアの細かく制御する必要がある．しかし物理ハードウェアは仮想ハードウェアと異なり，ソフトウェアから完全には制御できない．この課題を解決するための手法として，物理ハードウェアをその仕様に沿って間接的に制御する手法を示す．

ベアメタルクラウドで実現されていないマシン管理機能の中でもサービスの品質維持に大きく貢献すると考えられるライブマイグレーション機能と物理ハードウェアの保護機能に焦点を当てる．ライブマイグレーションは OS を無停止で別の物理マシンへ移動する機能である．この機能は IaaS において OS 無停止での物理マシンの計画メンテナンスを実現し，物理マシンの可用性に寄与する．また，物理ハードウェアの保護は提供する物理マシンのセキュリティを維持するために必要である．物理ハードウェアを悪意のあるユーザーから保護しなければ，他のユーザーが物理ハードウェアを通して攻撃の被害を受ける可能性がある．また，物理ハードウェアを起動不能にされることで，クラウドサービスが永続的な DoS の被害を受ける可能性がある．

物理マシンモニタと物理マシンモニタによるライブマイグレーションとハードウェア保護機能について，それぞれ設計を示す．また，これらのプロトタイプを実装し評価実験を行った結果を示す．評価結果は，物理マシンモニタが OS に依存しない形でかつ僅かな性能劣化で管理機能を提供できることを示している．

目次

図目次

表目次

第 1 章

はじめに

1.1 研究の背景

現在普及しているクラウドサービスの中に Infrastructure-as-a-Service (以下，IaaS) と呼ばれるインターネット経由でサーバーマシンを提供するサービスがある．IaaS クラウドのユーザーはクラウド事業者からサーバーマシンを借り，そのマシン上 に OS をインストールし利用する．また，ユーザーは OS 上で管理者権限を持つため，OS の差し替え，カーネルモジュールのロードやアンロード，OS の設定変更，ライブラリやアプリケーションのインストールが行える．このため，ユーザーは事業者が用意した OS 以外の OS もインストールできる．このように，IaaS は PaaS や SaaS といった他のクラウドサービス形態と比べ，ユーザーの自由度が高いことが特徴である．

IaaS のユーザーはネットワークを介して提供されるマシンにログインし利用できる．一般的に，IaaS では，Web UI や Web API が提供されており，これらを介してマシンの電源投入，シャットダウン，再起動などが行える． また，これらのインターフェイスを介して，新しいマシンを注文したり，借りているマシンを返却したりもできる．ユーザーは提供されるマシンに対して OS をインストールし，それを管理しつつ利用する．一般的に，主要な OS においては Web UI などで指定することで，事業者が用意した OS をインストールできる．

IaaS ではサーバーマシンとして仮想マシン (Virtual Machine, VM) を提供することが多い．仮想マシン (Virtual Machine, VM) とは，ソフトウェアによってエミュレートされた仮想的な計算機であり，仮想マシンモニタ (Virtual Machine Monitor, VMM) と呼ばれるソフトウェアで作成および管理される．仮想マシンを利用する利点として，一台の物理マシン上で複数の VM を作成し提供できる点と，マシンの管理がソフトウェア的に容易に行える点がある． VMM はゲスト OS から独立して，かつ，ゲスト OS よりも高い特権で動作し，VM をソフトウェア的に管理する．また，VMM はゲスト OS から別のゲスト OS や VMM 自身，さらに仮想マシンをホストしている物理マシンを保護する役割も担う．

VM 上で動作する OS は，一般的に物理マシン上で動作するよりものよりも処理性能が低下する．これは，物理マシン環境に比べ VMM によるハードウェアの仮想化処理が追加で行われているためである．現在は CPU や周辺デバイスが持つ仮想化支援機能を用いることで性能劣化は小さくなっているものの，依然として性能劣化は完全には取り除かれていない．特に，現在でも I/O デバイスの処理性能は物理マシン環境と比べて低下することが多い．

IaaS クラウドが普及するにつれ，IaaS 上で科学計算などの高負荷で大規模なワークロードを実行したいというユーザーが現れた． このような用途では，仮想化によるオーバヘッドが無視できない [1] ため，このようなユーザーは仮想マシンによる性能劣化を避け，より高速に

処理を実行したいという要望を持つ．このような背景から，仮想マシンだけではなく物理マシンも提供するベアメタルクラウドが登場した．ユーザーは物理マシンを利用することで，仮想化によるオーバヘッドを避けられる．実際に，IBM[2], Oracle[3], Internap[4], Rackspace[5], AWS[6] がベアメタルクラウドのサービスを提供しており，科学計算や映像処理の用途で用いられている．

ベアメタルクラウドは，セキュリティへの関心や危機感の強いユーザーにも適している．なぜなら，物理マシンはユーザー一人で一台の物理マシンを占有するため，別ユーザーとのマシン共有によるセキュリティリスクを回避できるためである．例えば，同一の物理マシンに複数ユーザーの VM が同時に動作するマルチテナント構成では，ユーザー間で情報漏洩が起きる可能性がある．マルチテナント構成を避けるために，一つの物理マシンで特定のユーザーの VM のみの動作を保証する Dedicated Instance というサービスを提供している場合もある．しかし，Dedicated Instance では VM を複数動かす必要がない場合でも，デバイスの仮想化によるオーバーヘッドが発生してしまう．このことは，本来不要な処理のためにマシンの計算資源を利用しているため，マシンの利用効率の観点でもよくない．

性能面以外での物理マシンの利点は，ユーザーが物理ハードウェアの機能を全て利用できる点である． 近年の NVMe SSD や 3D Xpoint といったストレージデバイスや 10 GbE や 40 GbE といったネットワークデバイス の性能は急速に改良されている．さらに，このようなハイエンドなデバイスには性能向上に役立つ高度な機能を提供している．例えば，マルチキュー，Single Root I/O Virtualization (以後，SR-IOV)，Remote Direct Memory Access (以後，RDMA) などがある．このような機能を活かし，高性能化した I/O デバイスの性能を十分に引き出すために，ユーザーモードドライバー, 新たな OS のアーキテクチャ，モダンなデバイスに最適化したアプリケーションなどが提案されている．[7, 8, 9, 10, 11, 12, 13]. これらのシステムは，ハイエンドな物理デバイスが提供する豊富な機能を利用して高い I/O 性能を実現しているため，ハードウェアが仮想化されている仮想マシンより物理ハードウェアの機能を直接利用できる物理マシンの方が望ましい．例えば，SeaSter[8], Scally DB[9] はハイエンドなネットワークデバイスが複数のデータ転送用キューを作成できることを利用している．その他のソフトウェアの例として，DPDK[7], IX[10], Arrakis[11], SPDK[14] などがある．また，その他のハードウェア機能の例として，最近のハイエンドネットワークデバイスは複数の MAC アドレスを扱うための機能や，SR-IOV と呼ばれるハードウェアによるデバイス仮想化支援機能がある．しかしこれらの手法の中には，仮想化環境では十分に活かせないものもある．例えば，仮想化によって物理デバイスが持つ機能をゲスト OS に対して隠蔽してしまったり，割り込みの仮想化によってゲスト OS が直接物理マシンの割り込みを制御することができなかったりする．これらを利用して高い I/O 性能を実現したいユーザーにとって，仮想マシンだけではなく物理マシンも提供するベアメタルクラウドが望ましい．

IaaS 事業者は，提供するサーバーマシンを管理する必要がある．例えば，ユーザーから Web UI や Web API を介して行われる要求に従ってマシンの電源投入，電源断，OS のデプロイ，といった操作が必要となる．また，サーバーマシン提供前にはハードディスクの初期化なども必要となる．これらを手動で行うことは大規模なサービスを提供する上で現実的ではない．そこで，上記のような作業をソフトウェア的にかつリモートで行うことで，マシンを一括で効率的に管理している．仮想マシン環境では，VMM で VM を制御することでマシンの一括管理を実現している．一方，物理マシン環境では IPMI や iPXE といった機構を使いマシンの管理を実現している．IPMI や iPXE は物理マシンの電源制御や OS のデプロイを実現できるものの，ユーザーの OS 動作中におけるマシン管理機能は仮想マシン環境に比べ充実していない．このためベアメタルクラウドの事業者は，仮想マシン環境では利用可能

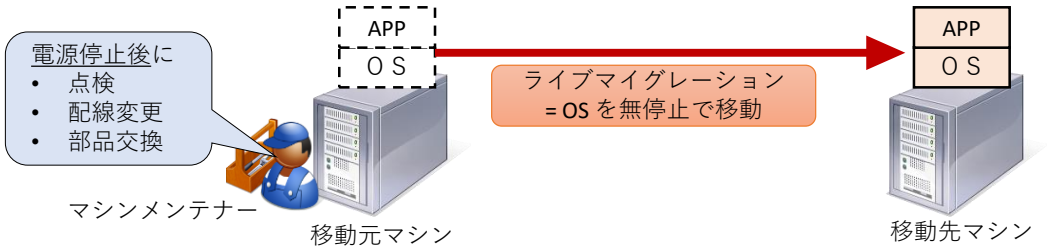

図 1.1　IaaS におけるメンテナンス時のライブマイグレーションの利用

なマシン管理機能のいくつかを使えない状態で運用している．このことはベアメタルクラウドのサービス品質に影響を与える．中でもライブマイグレーション欠くことにより主に可用性が，物理ハードウェアの保護機能を欠くことにより主にセキュリティが，それぞれ損なわれている．以下では，ベアメタルクラウドにおけるこれら 2 つの機能とサービス品質の関連について述べる．

ライブマイグレーションとは，ある物理マシン上で動作している OS を無停止で別の物理マシンへ移動する機能である．ライブマイグレーションは IaaS において，OS 無停止でハードウェアメンテナンスやファームウェアアップデートなどの作業を行う上で不可欠な機能である．ライブマイグレーションがあることで，クラウド事業者は可用性を損ねずに物理マシンの様々な管理作業を行える．クラウド事業者は物理マシンの計画メンテナンスで，ハードウェアの定期的な点検や交換，ファームウェアのアップデートを行ったりする．計画メンテナンスを行うことで，予期しない物理ハードウェアの故障を減らし，ユーザの OS が故障の影響を受ける可能性を抑えられる．しかしながら，これらの作業は物理マシンの電源停止が必要になり，ユーザーがメンテナンス対象のマシンを利用していると，動作中のユーザーの OS も同時に停止してしまう．このように，メンテナンスの度にユーザーの OS を停止してしまうと，ユーザーから見たサーバーマシンの可用性が損なわれてしまい，サービスレベル保証 (SLA) に直結する．そこで，クラウド事業者は図 1.1 にように，ライブマイグレーションを用いる．事業者はメンテナンス作業前にライブマイグレーションによってゲスト OS を別マシンに待避することで，OS を止めずにメンテナンス作業を始められる．実際， IaaS クラウドでは VMM の機能を用いて保守作業を行っていることを公表している例がある [15]．また，可用性向上のためのライブマイグレーションの用途として計画メンテナンス以外に proactive fault tolerance が提案されている [16, 17, 18] ．Proactive fault torelance は，温度や冷却ファンの状態といったハードウェアの様々な指標を監視し，ハードウェアが実際に故障する前に動作中の OS をライブマイグレーションによって別の物理マシンへ移動させることで，当該 OS がハードウェア故障の影響を受けないようにするものである．このようにライブマイグレーションは IaaS におけるサーバーマシンの可用性に寄与するものの，ベアメタルクラウドでは利用できない．このため，ベアメタルクラウドでは一般的な IaaS と比べサーバーマシンの可用性が低くなっていると言える．

IaaS におけるセキュリティを維持するには，提供するマシンを構成するハードウェアが正常に動作する必要があり，このためにユーザーから物理ハードウェアを保護する必要がある．マシンが仕様とは異なる動作をすると，そのマシン上で動作する OS やアプリケーションに影響を与え，データの破壊につながる可能性がある．例えば，ストレージデバイスが正しく動作しないと，ユーザーがストレージに格納したデータが破壊される可能性がある．さらに，

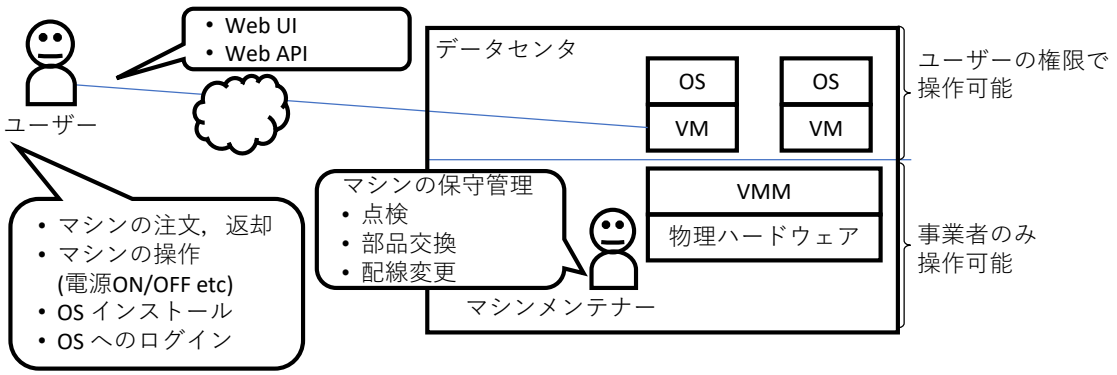

図 1.2　IaaS におけるユーザーと事業者の権限

ハードウェアが悪意のある動作をし，データを盗み出すような状態になると，機密性が保たれない．例えば，ネットワークデバイスが悪意のある動作をし，マシン上のデータを盗み出して外部に送信すると，機密性が損なわれる．マシンのセキュリティを維持するには，提供するマシンのハードウェアが仕様通りに正しく，かつ悪意なく動作する必要がある．仮想マシンにおいて，仮想ハードウェアが攻撃されたとしても，仮想マシンは返却された時点で削除され，ユーザーが変わるたびに VMM がソフトウェア的に新たな VM を作成するため，問題とならない．このため，仮想マシンを利用し始める際に，以前利用したユーザーの変更などが残っていることはなく，ユーザー間での影響もない．一方，物理マシンを構成する物理ハードウェアは一度返却され再度別のユーザーに提供する際に，ハードウェアの初期化は行うもののハードウェアを全て新品に入れ替えるわけではない．故に，一度ハードウェアの設定情報やファームウェアが改変されると，その後に物理マシンを利用する他のユーザーに影響が及ぶ可能性がある．以上のことから，ベアメタルクラウドにおいては，物理ハードウェアが仕様通り正しく動作する状態を維持するために，ユーザーからハードウェアを保護する必要がある．

1.2　本研究の目的とアプローチ

本研究の目的は，ベアメタルクラウドにおいて OS 動作中に物理マシンをより柔軟に管理できるシステムの実現である．現在ベアメタルクラウドで実現できていない管理機能の中でライブマイグレーションとハードウェアの保護はサービス品質への影響が大きいと考えるため，本稿ではこの二つの機能の実現に焦点を当てる．ベアメタルクラウドにおけるマシン管理を実現するには，以下の要件を満たす必要がある．

1. OS 非依存
2. 低オーバヘッド
3. 物理ハードウェアの提供

それぞれの要件について，以下で説明する．

一つ目の要件はマシン管理の機能が OS 非依存で動作するというものである．ベアメタルクラウドを含む IaaS クラウドにおいて，マシンの管理は OS に依存しない形で行う必要がある．OS 非依存でなければならない理由は，図 1.2 にある通り，クラウドのユーザーと事業者では権限の範囲が分離しているためである．ユーザーは，マシン上にインストールされて

いる OS において管理者権限持ち，これを操作，管理する．一方，事業者はユーザーがインストールした OS を直接操作できず，マシンのみを制御する．このため，管理機能の一部もしくは全てが OS に依存していると，事業者はユーザーに管理機能の操作を依頼することになる．ユーザーにとって，事業者の行うメンテナンス作業などにすべて対応することは現実的ではない．仮にクラウド事業者が用意したモジュールやエージェントプログラムを OS 内で動作させる仕組みにしたとしても，これらはユーザーの管理下で動作するため，ユーザーが停止や改変してしまう可能がある．さらに，ユーザーが悪意を持っていれば，これらのモジュールなどは容易に OS から取り除けるため，ハードウェアの保護は OS 内で実現しても悪意のあるユーザーから物理ハードウェアを保護できない．また，マシン管理機能が特定の OS の仕組みに依存すると，ユーザーは事業者が対応した OS しか選べなくなったり，ユーザーによる OS のカスタマイズの妨げになったりする．このようなことを避け，ユーザーの自由度を維持することも，OS 非依存であるべき理由である．

二つ目の要件は，低オーバヘッドである．仮想化による性能劣化を伴わない物理マシンの提供が，ベアメタルクラウドの大きな利点である．この利点を損なわないために，マシンの管理機能実現においても性能劣化をできる限り小さくすることが求められる．マシン管理機能の実現のためには OS 動作中にマシンを制御する必要があるものの，この処理を可能な限り少なくし，性能への影響を小さくする必要がある．

三つ目の要件は，物理ハードウェアの提供である．物理ハードウェアを直接利用できることはベアメタルクラウドの利点である．これを維持するために，物理ハードウェアが持つ機能は可能な限り隠蔽せずに OS に提供することが求められる．

現在 IaaS で利用されている VMM によるマシン管理機能は VM を制御することで管理機能を実現しているため，OS 非依存の要件を満たしたマシン管理機能を実現している．しかし，VMM のマシン管理機能はベアメタルクラウドで必要な低オーバヘッドや物理ハードウェアの提供という要件を満たせない．これまでの研究の中に，仮想化環境の高速化や OS による管理機能の実現を提案するものがある [19, 20, 21]．しかしこれらの研究では上記 3 つの要件の一部しか満たせていないため，ベアメタルクラウドでの利用には不向きである．例えば仮想化環境の高速化では，管理機能を維持しつつ性能向上を実現しているものの，物理ハードウェアの提供が不十分となる．また，OS による管理機能の実現は，仮想化環境を必要としないため性能と物理ハードウェア提供の要件は満たせるものの，OS 非依存の要件を満たせない．

本研究では，上記 3 つの要件を満たしつつ物理マシンを管理するために，物理マシンモニタ (Physical Machine Monitor, PMM) を提案する．物理マシンモニタは，OS から独立した形で物理マシン全体を制御および管理することで，VMM と同様の管理機能を提供することを目的に設計されている．VMM と同様に，PMM は OS やハードウェアから独立したソフトウェアとして動作する．一方で，仮想マシンモニタとは異なり，ハードウェアを仮想化せず，またハードウェアと OS の間の通信は基本的にパススルーする．このため，ゲスト OS は物理マシン上で動作するのと同様に，物理ハードウェアを直接利用できる．また，基本的に通信へ介入しないため，性能劣化もほぼ存在しない．

本稿では以下の二つの機能について実際にプロトタイプを実装し，その評価を行った．一つ目は，OS を無停止で別のマシンへ移動させるライブマイグレーション機能 [22] である．ライブマイグレーションは，移動元マシンで OS が利用しているハードウェアの状態を移動先マシンに転送し復元することで実現する．このために，PMM は物理ハードウェアの状態を二つ目は，ハードウェアの保護機能である．これを実現するには，ユーザーがハードウェアが持つ設定やファームウェアを不正に改変することを防ぎ，物理ハードウェアを保護する必要が

ある．設定情報やファームウェアを改変する機能は，本来ハードウェアの管理者がファームウェアをアップデートしたり，ハードウェアの設定を変更したりするために提供されている．ベアメタルクラウドにおいて，これらの機能はベンダー側のみが操作するべきであり，ユーザーにこれらの機能を利用することを完全に許可するべきではない．このため，物理マシンモニタで設定情報やファームウェアを改変する機能のみを OS から利用できないようにする．その一方で，データ送受信などのその他機能は物理ハードウェアと同様に利用可能とする．

本稿では，これら二つの機能のプロトタイプを実装し評価を行った．評価の結果から，性能劣化を抑えつつ，これらの機能を実現できたことを確認した．

1.3 本論文の構成

本稿における各章の構成は以下の通りである．第 2 章 では本研究と関連する研究について述べる．第 3 章 では本研究で提案する物理マシンモニタについて述べる．まずベアメタルクラウドにおいて物理マシンモニタに要求される要件を示す．その後，要件を満たすシステムの設計について述べ，その実装について述べる．この章で述べる設計と実装は物理マシンモニタにおける核となる部分についてである．これを基に管理機能を実現するためのそれぞれの設計と実装は第 4 章と第 5 章でそれぞれ述べる．

第 4 章 では物理マシンモニタによるライブマイグレーション機能について述べる．まず物理マシンモニタによってライブマイグレーションを実現するための課題を示し，これまでのライブマイグレーションに関する関連研究について示す．次に，課題を解決するためのアイデアとそれを基にした設計を述べ，設計を基にして行ったプロトタイプの実装ついて述べる．その後，プロトタイプを用いて提案手法を評価した結果を示す．最後に今後の応用の可能性などについての議論を示す．

第 5 章 では物理マシンモニタによるハードウェア保護について述べる．まずはベアメタルクラウドにおいてハードウェア保護の必要性についてより詳細に示すために，脅威モデルを示し関連する研究について述べる．次に，脅威モデルに対して有効なハードウェア保護が可能なシステムの設計を示し，それを基にしたプロトタイプ実装について述べる．その後，プロトタイプを用いて提案手法を評価した結果を示す．最後に，現在のハードウェアにおける制約や今後の発展の可能性についての議論を示す．

第 6 章 では，本稿の結論と今後の課題を述べる．

第 2 章

関連研究

この章では，本研究と関連する研究について述べる．ここでは，関連研究を仮想マシン環境での高速化，コンテナ型仮想化，OS でのマシン管理機能，およびハイパバイザを廃した IaaS 基盤の 4 つに分けて述べる．

2.1 仮想マシン環境での高速化手法

ベアメタルクラウドが登場した背景には，仮想マシンによる性能劣化があった．このため，仮想マシンの性能劣化を無くすことができれば，仮想マシンであっても現在のベアメタルクラウドの需要の一部を満たせる．故に，仮想マシン環境の高速化は IaaS で提供するマシンの高速化という点で本研究とモチベーションが共通している．このため，この節では仮想マシンの高速化に関する研究や関連技術について述べる．

ハードウェアによる仮想化支援機能が提供されるようになり，ハードウェアの支援による仮想マシン高速化が進んでいる．CPU に関しては， Intel 製 CPU は Intel VT-x, AMD 社製 CPU は AMD-V と呼ばれる仮想化支援機能をそれぞれ提供している．また，I/O デバイスについては，PCIe の規格で仕様が定められている仮想化支援機能である Single-root I/O virtualization (以後，SR-IOV) がある．SR-IOV を用いることで，I/O デバイスがそのインターフェイスを多重化し，それぞれのインターフェイスが別々のデバイスのようにふるまうことができる．SR-IOV 対応のデバイスは Physical function (以後，PF) と Virtual function (以後，VF) を持つ．PF は I/O デバイスが元々持っている機能であり，デバイス全体を制御するための機能が含まれる．また，PF は VF の生成や削除を行う機能を提供する．VF は SR-IOV の機能によって多重化されたインターフェイスで利用できる機能である．VMM における一般的な SR-IOV の利用方法は以下のとおりである．まず，VMM が PF を用いて VF を作成する．その後，この VF を VM に対して提供する．この際，VMM は VF を一つの PCI デバイスとして PCI-passthrough で提供する．このため，VM が VF を用いてデータの送受信を行う際には VMM は一切介在しないため，物理デバイスに近い性能を実現できる．

仮想化支援機能が提供されるようになっても，依然として仮想マシンモニタにはオーバヘッドが存在する．このオーバヘッドをさらに減らす研究の一つとして ELI [23] がある．この研究では，VMM による割り込みへの介入を避けることで，VMM による性能劣化を抑えることを提案している．

PCI-passthrough など一部の高速化手法は仮想マシンモニタが提供するマシン管理機能と共存できない形で実現されている．このため，高速化と管理機能の実現を両立することを目

的とした研究もある [19]．しかしながら，OS への改変が必要であったり仮想化による性能劣化が一部あったりと制約が残る．また，本研究の焦点であるベアメタルクラウドでは，ユーザにできるだけ物理ハードウェアの機能をそのまま利用可能にすることが望ましいものの，仮想マシン高速化のために SR-IOV などのハードウェア機能を利用すると，OS にこれらの機能を提供できなくなってしまう．

仮想マシンモニタの高速化とは別の観点として，ゲスト OS をクラウド環境に特化した設計とすることで高速化する研究もある．クラウド環境において，複数の VM を容易に構築できること，また負荷に合わせてアプリケーションをスケールアウトする運用が増えたことから，一つの VM 内では単一のアプリケーションのみを動作させる運用が増加した．この点に着目し，ゲスト OS 内のソフトウェアスタックを削減し，ゲスト OS 内のオーバーヘッドを削減するための研究がある．その一つとして，単一のアプリケーションのみを実行するライブラリ OS である OS^v がある．[24] OS^v は単一のアプリケーションのみを実行するように設計することで，汎用 OS のように複雑な仮想メモリ空間やプロセスの管理を大幅に削減している．しかし，単一のアプリケーションしか動作しない故に，複数プロセスが協調して動作するような複雑なシステムを運用できないことがある．

性能面とは別のアプローチとして，VMM の規模を小さくすることで VMM の脆弱性を減らし，よりセキュアなクラウド環境を構築しようとする研究もある．Steinberg らは [25] ハイパバイザの機能を分割し，最高特権で動作するコード規模を小さくしたハイパバイザである NOVA を提案した．これにより，最高特権で動作する Trusted Code Base (TCB) を小さくすることができる．NOVA の設計は，マイクロカーネルの思想をハイパバイザに取り込んだものとなっている．

2.2 コンテナ型仮想化

一つの OS のプロセス空間やファイルシステム空間を多重化することで，あたかも複数の OS 環境があるようにふるまう仮想化をコンテナ仮想化と呼ぶ．コンテナの例として，OpenVZ[26] や docker, LXC, Jail がある．

仮想マシンと異なる点としては，ホスト OS とゲスト OS でカーネルが共有されている点である．仮想マシン環境では，ホスト OS とゲスト OS は別々にカーネルを持ち，VM のユーザーはゲスト OS のカーネルを管理できるため，新たなカーネルモジュールロードしたり，カーネルを差し替えたりできる．一方コンテナ環境では，カーネルはホスト側の特権でのみ操作可能であり，ゲスト OS のユーザーからは操作できない．このように理由から，IaaS の提供のためにコンテナ型仮想化を利用しようとすると，仮想マシンの提供に比べ機能が制限されてしまう．また，カーネルが共有されているなどゲスト OS 間の隔離が弱いため，セキュリティに関する懸念やゲスト OS 間の性能の影響に関する懸念が強くなる．現在では PaaS サービスとしてコンテナでの動作環境を提供するクラウドサービスも存在している．

2.3 OS での管理機能の実現

これまでの研究で，IaaS での物理マシン管理に有用な機能を OS に実現する研究が行われている [20, 21]．例えば OS 自身に OS のライブマイグレーション機能を実現する研究 [20] や悪意のある挙動を示すハードウェアを OS で検知する研究 [21] がある．これらの研究は非仮想化環境でライブマイグレーションやハードウェアのセキュリティ対策の機能を実現しようとしている点で本研究と関連している．しかしながら，これらの研究は IaaS を想定してお

らず，IaaS 環境での要件である OS 非依存性は考慮されていない．第 1 章 で述べたように，OS に依存した手法は実際のベアメタルクラウドで事業者が利用することは難しい．

2.4 ハイパバイザを廃した IaaS 基盤

これまでの研究の中には，ハイパバイザを廃した環境で IaaS 基盤を提供しようとする研究もある．Keller ら [27] は各 VM 間の独立性が高いマルチテナントの IaaS 環境を実現するために NoHype というアーキテクチャを提案した．NoHype ではハードウェアを論理分割することで，VM 間の隔離を強固にしている．具体的には，各 VM で別々の CPU コアと物理デバイスを占有させることで，VM 間でのハードウェアの共有を減らしている．また，マシン全体を管理するモジュールは VM とコアを共有せずに動作し，VM の管理モジュールはコアごとに独立して動作する．このようにすることで，中央集権的なハイパバイザをシステムから廃している．

文献 [28] では拡張した NIC と Remote-Management Engine (RME) による物理マシンのホスティング手法を提案している．拡張 NIC と RME は IaaS 事業者に管理用機能を提供しつつ，ユーザーの OS には NIC やローカルストレージをパススルーしている．この研究ではマイグレーションに言及しているものの，これを実現するためにはゲスト OS での対応が必要である．また，ライブマイグレーションについては言及がない．

Azab らは SICE と呼ばれる BIOS と SMM のみを用いて OS から独立したセキュアな実行環境を提供する基盤を提案している [29]．SICE はソフトウェア的なハイパバイザなしで，OS から独立したセキュアな実行環境を提供する．

これらの研究は IaaS クラウドにおける VMM を小さくしたり取り除いたりしつつ管理機能を維持するという点が本研究と共通している．一方これらの研究の目的は，マルチテナントの IaaS において VM 間の独立性を高め，物理マシン共有による機密性や完全性の低下を防ぐことであり，ベアメタルクラウドは想定していない．それ故に，一台の物理マシンで一つの OS のみ動作するベアメタルクラウドとは想定が異なる．また性能に関しては， 一般的な VMM と比べて同程度かあるいは少し性能低下している．

また，IaaS 事業者が物理マシンを管理するための機能は IPMI などを介してハードウェアによっていくつか提供されている．例えば，ネットワークを介した物理マシンの電源制御や画面出力の機能が提供されている．また OpenStack などの IaaS 環境を構築するシステムでは，IPMI を介してネットワーク内の物理マシンをソフトウェアから管理する機能が提供されている．しかし第 1 章 で述べた通り，IPMI などで提供する機能は仮想マシンモニタに比べて少なく，ライブマイグレーションやハードウェアの保護といった機能は現在のところ提供されていない．

第 3 章

提案手法: 物理マシンモニタ

本章では，提案する物理マシンモニタについて述べる．まず，物理マシンモニタの概要を述べ，システムが満たすべき要件とこれを満たすために解決すべき課題について述べる．その後，物理マシンモニタの核となる部分の設計と実装について述べる．

3.1 提案システムの概要

本研究では，ベアメタルクラウドでのマシン管理の問題を解決するためのシステムを提案する．ベアメタルクラウドにおけるシステムの要件は，OS 非依存性，ユーザーへの物理ハードウェア提供，低オーバヘッド，の三つである．一つ目の要件は，第 1 章 で述べた通り，IaaS のユーザーと事業者の権限分離や OS の選択を制限しないために必要な要件である．二つ目の要件は，ベアメタルクラウドの利点の一つである物理ハードウェアの機能を直接利用できる点を維持するために必要な要件である．三つ目の要件は，物理マシンの性能の高さを維持するために必要な要件である．

これらの要件を満たすベアメタルクラウド向けマシン管理基盤システムとして物理マシンモニタ (Physical Machine Monitor, 以後 PMM) を提案する．PMM は，OS 非依存で物理マシンを制御し管理するための基盤となるソフトウェアである．PMM は BIOS やファームウェアが提供できていないライブマイグレーションやハードウェア保護といった機能を提供する．これらの機能は仮想マシンで既に実現されているものの，それらはデバイスの仮想化による手法でありベアメタルクラウドでの利用には適さない．本研究で提案する PMM はデバイスの仮想化に依らない手法でこれらの機能を実現することで，ベアメタルクラウドでの利用を可能にする．PMM は仮想マシンと同等の機能を提供するために，BIOS やファームウェアよりも細かく物理マシンを制御する必要がある．

3.2 提案システム実現の課題

上記の三つ要件を満たすには，ハードウェアをパススルーしつつ，OS に依存せずに物理マシンの管理機能を実現する必要がある．これまで，三つの要件の一部を満たしているシステムは提案されているものの，全ての要件を満たしているシステムは著者の知る限り提案されていない．三つの要件を満たすための課題を以下で二つ挙げ説明する．

物理ハードウェアのパススルー　これまで VMM によって管理機能が実現されていることからもわかる通り，ハードウェアの仮想化は OS 非依存でのマシン管理を容易にする．しかしハードウェアを仮想化すると，物理ハードウェアの提供という要件と低オーバ

ヘッドという要件も満たせなくなる．このため，本提案手法ではハードウェアを仮想化せずにパススルーした状態で管理機能を実現する必要がある．

物理ハードウェアの制御　OS 非依存な管理機能とするためには，OS を直接制御せずに機能を実現する必要がある．現在 IaaS で利用されている仮想マシンモニタによる管理機能は仮想マシンの状態を制御することで OS 非依存に動作している．これと同様に，OS に依存せずに物理マシンを管理するには，物理マシンの状態を制御する必要がある．また，物理マシンの状態を制御する上でも，OS の支援は受けられない．しかし，ソフトウェアで OS 非依存で物理ハードウェアを制御し管理機能を実現する手法は確立されていない．

マシン管理機能の実行前後において，OS が想定するハードウェア状態と実際のハードウェアの状態の間にずれがあってはいけない．仮に両者の間でずれがあると，OS によるハードウェアの制御処理が正しく動作せず，OS は動作を継続できなくなる．例えば，管理システムがデバイスを初期化したにも拘わらず OS のデバイスドライバがこれを検知せずに処理を続行すると，デバイスドライバの処理は正しく継続できない．OS の支援が受けられれば，OS を制御することでこのような両者のずれを無くせる．例えば，管理システムがハードウェアを初期化した場合でも，OS のデバイスドライバがこれを検知し初期化後のデバイスを扱うように対応すれば，OS は動作を継続できる． しかし，OS 非依存であるためには OS の制御なしで機能を実現する必要がある．OS の制御なしでハードウェアと OS 間のずれを無くすためには，ハードウェアの状態を制御する必要がある．

物理ハードウェアの状態を OS 非依存で制御することは，仮想ハードウェアの制御に比べて難しい課題である．仮想マシンにおいては，仮想ハードウェアは VMM がソフトウェア的にエミュレートし管理している．故に，VMM は仮想ハードウェアの全ての状態をデータとして扱うことができる．一方，物理マシンの状態は物理ハードウェアが管理している．物理ハードウェアの状態の多くはレジスタを介してソフトウェアから直接アクセスし制御できるものの，一部の状態はソフトウェアから直接アクセスできない．しかし，管理機能を実現する上では直接アクセスできない状態も制御する必要がある．

3.3　設計

この節では本稿で提案する物理マシンモニタの核となる部分の設計ついて述べる．

3.3.1　PMM のアーキテクチャ

本研究では，PMM は物理マシン環境を OS から独立して管理するための基盤として設計する．このために，PMM はハイパバイザと同様の特権モードで動作するソフトウェアとした．その一方で，ハイパバイザとは異なり PMM は OS に対して物理マシン環境を提供しつつ物理マシンを管理することが目的であるため，基本的には物理ハードウェアは OS が直接制御する．つまり，PMM は 物理ハードウェア仮想化せずに OS にパススルーしている状態となる．これは，PCI デバイスだけで割り込みコントローラなどについても同様である．また，一般的なハイパバイザとは異なり，複数の OS を同時に動かす機能を持たない．このようにすることで，従来のハイパバイザでは避けられなかった仮想化によるオーバヘッドを大きく低減できる．例えば，一般的なハイパバイザが持つ CPU，メモリ，デバイスといったハードウェア資源の仮想化，アクセスの調停，スケジューリングといった機能が PMM では

不要となる．このために，PMM は OS とハードウェアの間で発生する I/O, メモリアクセス，割り込みといった通信に介入する必要も基本的になくなる．また，PMM を軽量で小さなハイパバイザのようなシステムとして実装するために，準パススルーハイパバイザと呼ばれるアーキテクチャをベースとして採用した．

3.3.2 準パススルーアーキテクチャ

本稿で提案する物理マシンモニタは，準パススルーハイパバイザ [30] と呼ばれるアーキテクチャを基にしている．準パススルーハイパーバイザーは，元々クライアント PC 向けのセキュアハイパバイザとして設計されたものである．このため，性能劣化を抑えつつ OS から独立して動作するセキュリティ機能を実現するための設計となっている．本研究はクライアント向け PC のセキュリティではなくベアメタルクラウドでの物理マシン管理機能の実現が目的である．しかし，提案するシステムが満たすべき要件は準パススルーハイパバイザの特徴と共通点が多いため，提案システムのベースとして採用した．そこで，この節では PMM のベースとなる準パススルーハイパバイザのアーキテクチャについて述べる．

準パススルーアーキテクチャは，基本的に OS とハードウェアの通信をパススルーする．一方で，ハイパバイザが動作する必要最低限の処理と何らかの機能を実現するために処理が必要な場合のみ OS とハードウェアの通信に介入する．ハイパバイザができる限り通信に介入する必要がないようにするために，ハイパバイザは非常に簡素化されている．その最たる点は，単一 OS の動作にのみ対応し複数 OS の動作には対応しない点である．これは，他のハイパバイザとは大きく異なる点である．このことによって，ハイパバイザの処理は非常に簡素なものになる．

準パススルーハイパバイザで何らかの機能を実現するためには OS とデバイス間の通信に対する介入が必要な場合がある．このために，準パススルーハイパバイザは準パススルードライバと呼ばれるソフトウェアコンポーネントを持つ．準パススルードライバとは，物理デバイスと OS 間の通信を一部だけ捉えるためのドライバである．準パススルードライバは一部の通信を捉えるために，物理デバイスのレジスタのレイアウトやレジスタの役割といったデバイスの仕様に関する情報を持つ．このため，準パススルードライバは一般的なデバイスドライバと同様に，仕様の異なるデバイスごとに用意される．一方で，一般的なデバイスドライバとは異なり，準パススルードライバはデバイスを完全に制御するものではない．なぜなら，物理デバイスの制御はゲスト OS が行い，準パススルードライバは一部機能を実現するために必要な処理のみを行うためである．例えば，ストレージのバックグラウンド暗号化を実現するために HDD の読み出しや書き込みの I/O に介入し，読み書きしているデータを暗号化および復号化している．この場合でも，データの読み書き以外の部分は可能な限りパススルーとなっている．それ故に，準パススルードライバの規模は一般的なデバイスドライバと比べて格段に小さい．

準パススルーハイパバイザは，OS に依存せずに起動するようにするために，ホスト OS などを伴わず OS 起動前に単独で起動するようになっている．これは，Xen や VMWare ESXi のような Type I ハイパバイザの起動と同様である．

3.3.3 仮想マシンモニタとの比較

図 3.1 に物理マシン，仮想マシン，および提案手法である物理マシンモニタの比較を示している．図左の物理マシンでは，OS が物理マシンの直上で動作している．この環境では，ベ

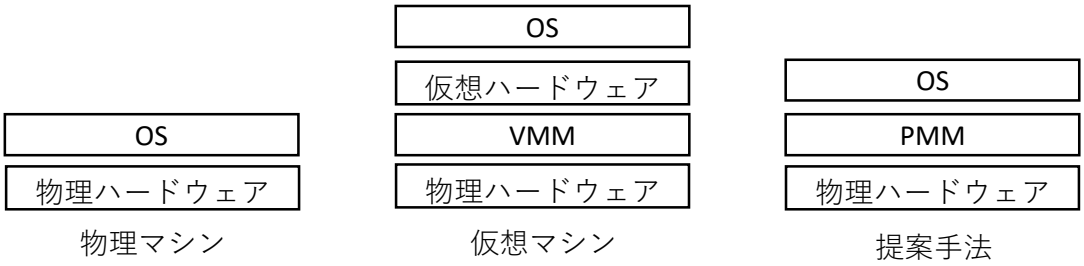

図 3.1　システムの比較

アメタルクラウドの事業者が利用できるマシンの管理機能はハードウェアが提供するもののみとなる．一方，仮想マシン環境においては，ゲスト OS は VMM に管理されている VM (図中の仮想ハードウェア) 上 で動作しており，事業者は提供している VM を管理するために VMM を用いることができる．VMM が提供する VM 管理の機能は物理ハードウェアが物理マシンを管理するために提供する機能と比べて豊富である．しかしながら，VMM は処理が複雑であるため，性能劣化を生む．PMM は，上記 2 つの中間となる手法である．PMM は OS より高い特権で，なおかつ OS から独立して動作する点で VMM に似ている．しかしながら，VMM とは異なり，PMM は仮想ハードウェアなどを作らず，物理ハードウェアを OS にそのまま提供する．その一方で，PMM は物理ハードウェアを制御し VMM と同様にマシンの管理機能を提供する．このため，ベアメタルクラウドの事業者は PMM を用いることで，物理マシンを管理しつつ物理マシンの高い性能をユーザーに提供し続けられる．

3.3.4　デバイスとの通信のパススルー

OS とデバイス間の通信には，I/O, DMA, 及び割り込みがある．仮想化環境では，OS は仮想デバイスに対してこれらの通信を行い，仮想マシンモニタは OS から仮想デバイスへの通信に介在し処理する．複数の仮想デバイスを作成し処理することで，物理的には一つしかデバイスなくても複数の VM に対して同時にデバイスを提供できる．VMM は各 VM の仮想デバイスに対して OS から行われる通信に介在し，複数 OS からのデバイスとの通信を調停しながら物理デバイスへの通信に変換する．実在するデバイスを完全にエミュレートする仮想デバイスは，既存のデバイスドライバが利用できる利点があるものの，仮想化の処理が複雑で性能劣化が大きい．そこで，仮想化が単純な処理で可能な実在しないデバイスを提供する準仮想化デバイスがある．準仮想化デバイスにおいても通信への介入で性能劣化は生じるものの，仮想化の処理がより単純になるために性能劣化が小さくなる．

デバイスの仮想化によって，複数 VM が同時に動作できるようになるものの，PMM は複数の VM を同時に動作させる必要はない．一方で，PMM は性能劣化を抑えるために，通信への介在は極力減らす必要がある．このため，PMM はデバイスを仮想化せず，OS とデバイス間の通信は基本的にパススルーする．また，PMM では複数 VM で共有するデバイスが一切ないため，割り込みへの介入や調停も必要なく，これもパススルーする．さらに，PMM は定期的に実行が必要な処理を持たないようにすることで，タイマー割り込みの受信を必要としない設計としている．これにより，PMM はタイマー割り込みについてもパススルーする．一時的に定期的な処理を必要とする場合には，割り込みとは別の方法で制御を得る．これについては後述する．

提案手法のアーキテクチャが性能面で有利であることを説明するために，図 3.2 に，一般

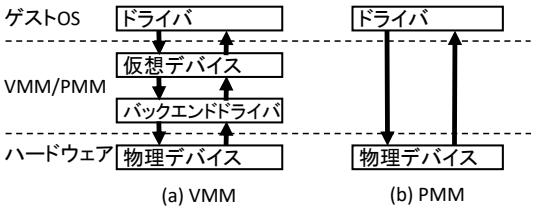

図 3.2　VMM と PMM の読み出し I/O 処理の比較

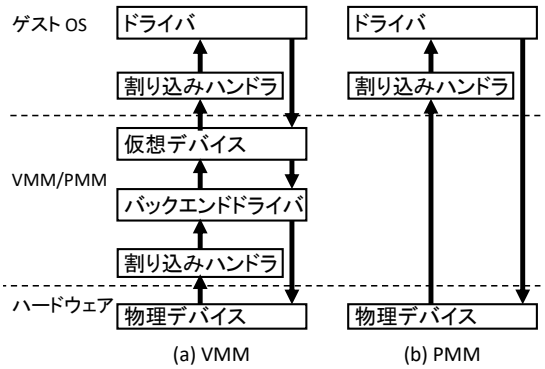

図 3.3　VMM と PMM の割り込み処理の比較

的な VMM と PMM における読み出し I/O の処理流れの比較を示す．この図では，ゲスト OS にあるデバイスドライバがデバイスに対して読み出し I/O を発行している場面を想定している．一般的な VMM では，ゲスト OS は VMM によって提供されている仮想デバイスに対して読み出し I/O を発行する．VMM では発行された I/O を VMM 内の仮想デバイスが受け取る．仮想デバイスは受け取ったリクエストを解析し，必要な場合には物理デバイスに対して実際に I/O を発行するバックエンドドライバにリクエストを送る．バックエンドドライバは VMM 内にある物理デバイスを扱うドライバである．バックエンドドライバは，依頼された処理に必要な I/O を物理デバイスに対して発行する．QEMU/KVM であればホスト OS が持つ物理デバイス用のドライバ，Xen であれば，Dom 0 が持つ物理デバイス用のドライバがバックエンドドライバとして動作する．バックエンドドライバは物理デバイスから I/O の結果を受け取り，その結果を仮想デバイスに反映する．仮想デバイスは受け取った結果を解析し，その結果をもとにして仮想デバイスの状態を更新し，ゲスト OS のデバイスドライバに結果を返却する．このように，I/O のたびに VMM が介在し処理を行う．VMM はこれらの処理のなかで，I/O の調停などを行う．このことが，性能劣化につながっている．

一方 PMM では，ゲスト OS のドライバは物理デバイスに対して直接読み出し I/O を発行する．物理デバイスはゲスト OS のドライバに対して直接 I/O の結果を返却する．このように PMM は I/O 処理に介在しないため，性能劣化も起きない．しかしながら，I/O に介在しないため，VMM ほど簡単に処理を挟み込むことができない．このことは，PMM で管理機能を実現する上での課題となる場面がある．

割り込み処理において，VMM と PMM の間にはより顕著の違いがある．図 3.3 に VMM と PMM での割り込み処理の比較を示す．この図では，物理デバイスからゲスト OS に対して割り込みを通知する場面を想定している．また，割り込みを受け取ったゲスト OS が割り込みに対する処理を終えた後，割り込み処理の終了をデバイスに通知することを想定している．一般的な VMM (図 3.3-(a)) においては，物理デバイスからの割り込みはまず VMM の割り込みハンドラが受信する．その後，割り込みハンドラがバックエンドドライバに割り込みを伝える．ハンドラは受信した割り込みが VMM に向けたものなのかゲスト OS に向けたものなのかを判断する．もし割り込みがゲスト OS に向けたものであれば，VMM の割り込みハンドラは仮想デバイス対して仮想デバイスの状態を更新するように要求を出す．仮想デバイスは割り込みエミュレーション処理を行い，ゲスト OS に割り込みを通知する．この仮想的な割り込みは，ゲスト OS の割り込みハンドラが受け取る．割り込みハンドラはデバイスドライバに割り込みを通知する．ゲスト OS 内のドライバが割り込みを受信した後，ドライバは割り込み処理が完了したことを割り込みコントローラーやデバイスに通知するための

書き込み I/O を発行する．この処理は，書き込み I/O によって行われる．仮想デバイスはバックエンドドライバに対して，物理デバイスへ割り込み処理の終了を通知するよう依頼する．バックエンドドライバは通知を受けて割り込み処理終了を物理デバイスに通知する．読み出しの I/O と同様に，この I/O もまた VMM により介入される．このように，VMM は割り込みと割り込み終了通知の I/O の処理に介在する．このために，VMM 上での割り込み処理では，ゲスト OS と VMM の中で多くのコンテキストスイッチが発生する．また，仮想デバイスと物理デバイスの間で多くの変換処理が必要となる．これらは仮想化のオーバーヘッドの要因となっている．

一方，PMM は割り込みにも介在しない．PMM において，物理デバイスからの割り込みは直接ゲスト OS に通知される．また，割り込み処理終了の通知もゲスト OS から物理デバイスに対して直接的に書き込み I/O を発行することで行われる．このような割り込み処理の違いは，特に I/O デバイスのレイテンシ性能に大きな影響を与える．

3.4 共通部分の実装

本稿では PMM による管理機能のプロタイプを実装し，これを用いて提案手法の評価を行う．このプロトタイプにおける PMM のコアとなる部分の実装について本章で述べる．

PMM は BitVisor[30][31] をベースに実装している．設計上，BitVisor はゲスト OS としてあらゆる OS が動作する．実際に，BitVisor 上で Windows, Mac OS, Linux, FreeBSD などが動作する．また，本稿で述べるプロトタイプ実装は Intel 社製 CPU にのみ対応したため，以下では Intel 社製 CPU での実装について述べる．しかしながら，AMD 社製の 同アーキテクチャの CPU においても同等の機能を備えているため同様の実装が可能だと考えられる．

3.4.1 CPU 管理

PMM は OS とは異なる特権モードで動作するために CPU の仮想化支援機能を用いている．CPU の仮想化支援機能では，ハイパバイザなどが動作する特権モードと仮想マシン上の OS などが動作する非特権モードが用意されている．Intel 社製 CPU の仮想化支援機能である Intel VT-x では，特権モードを root mode, 非特権モードを non-root mode と呼ぶ．それぞれのモードにおいて，x86 アーキテクチャにおける Ring 0 から Ring 3 の 4 つの特権レベルが用意されているため，OS は改変なしで非特権モードで動作できる．PMM の設計において，PMM は特権モードで，OS は非特権モードでそれぞれ動作する．

PMM は仮想化支援機能を用いて，OS が動作している非特権モードの CPU 状態を制御する．Intel VT-x では非特権モードの制御は，主に VMCS (virtual machine control structure) と呼ばれる構造のデータを介して行う．この VMCS は複数作成し管理することが可能であり，VMCS を切り替えることで非特権モードの CPU 状態を入れ替えることができる．このため，一般的な VMM では非特権モードで動作する CPU を仮想 CPU と見なし，仮想 CPU 毎に VMCS を作成し管理している．また，VMM では VM とそれに伴う仮想 CPU は不特定多数あり，さらに仮想 CPU と物理 CPU の対応は流動的である．このため，例えば動作途中にある仮想 CPU が物理 CPU 0 上で動いていても，別の時点では物理 CPU 1 で動作することもある．このような動作を vCPU マイグレーションと呼ぶ．vCPU マイグレーションは仮想 CPU に対応する VMCS をそれぞれの時点で異なる物理 CPU でロードすることで実現できる．vCPU マイグレーションにより仮想 CPU の処理を物理 CPU に柔軟に割り

振れる反面，仮想 CPU を移動する際に様々なキャッシュをフラッシュするため，CPU 性能の安定性が低下する．

PMM は一般的なハイパバイザとは異なり，VMCS は各物理コア毎に一つだけ管理し，物理 CPU でロードする VMCS は切り替えない．このようにすることで，PMM では vCPU マイグレーションは発生せず，キャッシュのフラッシュなどを避けられる．また，OS から識別している CPU コアと物理的な CPU コアの関係が変わらず，両者を同一視できるため，より物理マシン環境に近いと言える．また，ほとんどの CPU の機能は有効のまま OS は利用できる．このようにすることで，PMM 上の OS は物理マシン環境とほぼ同じ CPU を利用できる．

2 つの特権モードの遷移について，root mode から non-root mode への遷移は VM enter, non-root mode から root mode への遷移を VM exit と呼ばれる．VM enter は root-mode で専用の命令を実行することで起こる．一方，VM exit は様々な要因で発生する．例えば割り込みが入る，例外が発生する，ゲスト OS が特権を要するレジスタの値を操作しようとする，などがある．VM enter した後，VM exit が発生しない限り non-root mode で動作する OS は root-mode のソフトウェアの介入なしに動作し続ける．VM exit は non-root mode と root mode の状態遷移を伴うため，性能を改善するためには極力減らすべきイベントである．この状態遷移では，CPU のレジスタの値を差し替えるなどコンテキストスイッチの処理が行われるため，VM exit 自体が CPU サイクルを消費するイベントである．さらに，VM exit 後に root mode のソフトウェアの処理によって CPU サイクルを消費される．このために，VM exit が頻繁に発生すると，ゲスト OS の CPU サイクルが横取りされてしまい，結果として処理性能が低下する．また，デバイスとの通信において VM exit が発生してしまうと，レイテンシ性能に大きな影響がでる．

PMM は non-root mode で VM exit が極力発生しないようにしている．Intel VT-x において，root-mode で動作するソフトウェアはどのような要因で VM exit が発生するかは root-mode で設定できる．この機能を用いて，一般的な VMM であれば仮想化の処理のために必須である割り込みや例外発生時の VM exit においても，PMM では発生しないように設定している．このため，割り込みなどのイベントは PMM の介在なしに non-root mode で動作する OS に送られ，OS が直接処理する．このようにしたことで，PMM では VM exit の回数が大きく抑えられている．

3.4.2 メモリ管理

一般的に，Intel VT-x の root mode で動作する VMM は non-root mode でゲスト OS が利用する物理メモリ空間を制御するために extended page tables (以後，EPT) を用いる．EPT はゲスト OS が利用する物理アドレス空間 (以後，ゲスト物理アドレス空間) を仮想化するために，ハイパバイザ が用いる実際の物理アドレス (以後，ホスト物理アドレス) とゲスト物理アドレス空間の間でアドレス変換を行う機構である．アドレス変換のために，ゲスト物理アドレスとホスト物理アドレスのマッピングをページテーブルで指定する．このためのページテーブルの構造は，x86 アーキテクチャの仮想アドレスで用いられるものとほぼ同じである．VMM はこの機構を用いて，複数の VM のゲスト物理アドレス空間を一つのホスト物理アドレス空間上で共存させている．

PMM においては，OS は一つしか動作しないため，複数のゲスト物理アドレス空間を一つのホスト物理アドレス空間内で共存させる必要がない．また，PMM 自身は起動時に少量で固定長のメモリ空間を確保し，動作中に追加でメモリを確保しない．このようにすることで，

PMM はホスト物理アドレス空間上で OS が利用するメモリ領域と PMM が利用するメモリ領域を混在させずに分けることができる．このため，ゲスト物理アドレスとホスト物理アドレスは同一としており，PMM は EPT で二つのアドレス空間のストレートマッピングを作成している．ゲスト物理アドレスとホスト物理アドレスが同一にも関わらず EPT を用いているのは，EPT を用いて PMM の保護や後述する MMIO への介入を行うためである．

PMM は BIOS が返すメモリマップの変更と EPT の利用によって自身のメモリ領域を OS から保護している．先述の通り，PMM は起動時に固定長の物理メモリ領域を確保し，これを用いて処理を行う．今回作成したプロトタイプでは，PMM が確保する領域は 128 MB である．また，起動後にはこのメモリ領域以外からメモリを確保することはない．PMM はこのメモリ領域をゲスト OS が利用しないようにするために，BIOS が返すメモリマップの結果を変更し，このメモリ領域を Reserved としている．このため，一般的な OS であればこのメモリ領域を Reserved 領域として扱い利用することはない．仮に OS がこのメモリ領域にアクセスしようとした場合でもアクセスを防ぐために，PMM は意図的にこれらの領域に対する EPT のマッピングを作成しない．このようにすると，OS がこの領域にアクセスした時には VM exit が発生し PMM がハンドリングできるようになる．

また，PMM では OS 起動前に全てのメモリページに対する EPT マッピングを作成する．元々の BitVisor や一般的な VMM では， EPT のマッピングは当該物理ページの初回アクセス時に作成することで，利用されていないメモリ領域に対する無駄なマッピングを作成しないようにしている．その一方で，初回アクセス時には EPT マッピング作成処理のためにメモリアクセスが遅くなる．OS 起動前に全てのメモリページのマッピングを作成することで，動作時のメモリアクセス性能が安定しより物理マシン環境に近くなる．一方で，この実装には以下の二つのデメリットもある．一つ目はマシン起動時に追加の処理が行われるため，起動時間が増加する点である．この点については，ベアメタルクラウドの用途として頻繁にマシンを起動するユーザーは少ないと考えられることと，起動時間の増加が極端に大きくないため，影響は小さいと考えられる．二つ目は起動時に EPT のマッピングを全て作成するために，OS が利用していないページのマッピングも作成され，ページテーブルによるメモリ消費が不必要に増加する可能性がある点である．この点も，ベアメタルクラウドの用途としてメモリを多く使う用途が多いと考えられるため，作成したマッピングのほとんどが利用されマッピングが無駄になる割合は少なく影響は小さいと考えられる．このようの理由から，今回はオンデマンドなマッピングの作成ではなく事前にマッピングを作るよう実装を行った．

3.4.3 準パススルードライバ

マシンの管理機能を実現するために，PMM は一部の I/O に介入する必要がある．PMM ではこの介入処理を CPU の仮想化支援機能を用いて実現する．具体的には，ゲスト OS が特定の宛先へ I/O を発行した際に処理が PMM に移るようにする．

PMM は I/O 命令によって発行される programmed I/O (以後，PIO) とメモリアクセスとして発行される memory-mapped I/O (以後，MMIO) において VM exit を起こすためにそれぞれ異なるメカニズムを用いる．PIO への介入は，Intel VT-x が持つ機能である，指定された I/O ポートにアクセスされた際に VM exit を起こす機能を用いて実現する．PMM は介入が必要な I/O の宛先となる PIO のポートのみ指定し，この I/O ポートへのアクセスにのみ VM exit を発生させる．MMIO においては EPT を用いて VM exit を発生させる．EPT を用いている際，ゲスト OS がホスト物理アドレス空間へマッピングされてないゲスト物理アドレスにアクセスした場合，VM exit が発生するようになっている．このことを

利用して， PMM は 介入対象となる MMIO レジスタがあるメモリページを意図的に EPT でマッピングしないようにしている．ゲスト OS は PCI コンフィグレーション空間の base address レジスタ (BAR) を変更することで MMIO のベースアドレスを変更できるため，PMM はこの変更を追跡し，それに伴って EPT の設定も更新する．

PMM は上記の介入処理を，各物理デバイス用の準パススルードライバの中で行う．VMM のようにデバイスを仮想化している環境であれば，OS が利用する仮想デバイスのインターフェイスは物理デバイスの種類に依らず同じであり，それ故に介入する I/O やその際の処理も同じである．しかし，PMM においては介入すべき I/O や介入時の処理は物理デバイスによって異なる．なぜなら，OS は物理デバイスに対して直接アクセスするために，物理デバイスの仕様に沿った I/O を発行するためである．それ故に，仕様の異なる物理デバイス毎に準パススルードライバを用意し，この中で機能を実現するために介入すべき I/O の指定や，I/O に対する介入処理を記述している．デバイスドライバとは異なり，デバイスを完全に制御するわけではなく OS と物理デバイス間の I/O に一部介入し，機能を実現する役目を持つ．このため，実現する機能と関係ない I/O については，基本的に介入せずに OS が直接制御する．故に，機能を実現する際に介入すべき I/O の指定を少なくするほど，OS は物理デバイスと同様にデバイスを制御できるようになる．

第 4 章

物理マシンモニタによるライブマイグレーション

本章では，PMM によるベアメタルクラウドのためのライブマイグレーション手法について述べる．ライブマイグレーションはベアメタルクラウドでのサービス品質維持のために不可欠な機能の一つであり，PMM が提供すべき機能である．

この章の残りの部分は，以下の通りである．第 4.1 節 ではライブマイグレーション実現のための課題について述べる．次に，第 4.2 節 は関連研究について考察する．その後に，第 4.3 節 では，提案手法の設計について述べ，第 4.4 節 では，実装したプロトタイプの詳細について述べる．第 4.5 節 では，評価の結果を示し，第 4.6 節 では本手法の応用の可能性について議論する．

4.1 ライブマイグレーション実現における課題

PMM が提供するベアメタルクラウド向けライブマイグレーション機能は OS 非依存に動作しなければならない．このためには，OS 内のデータや状態を制御せずにハードウェアの状態のみ制御しライブマイグレーションを実現する必要がある．具体的には移動元マシンのハードウェアの状態を取得し，これを移動先マシンへ転送した後，移動先マシンで移動元マシンの状態を復元する，という処理の流れになる．実際に，VMM による VM 上の OS ライブマイグレーションは仮想ハードウェアの状態を制御するのみで実現しており，OS 非依存で動作する．VMM とは異なり，PMM は物理ハードウェアをパススルーし OS が直接制御している．故に，PMM は物理ハードウェアの状態を制御してライブマイグレーションを実現する必要がある．

提案するライブマイグレーションを実現するための主な課題は，物理デバイスの状態をどのように制御するかである．CPU とメモリの状態の制御は，CPU の仮想化支援機能を用いることで比較的容易に実現できる．それ故に，これらの状態の転送には，従来までのライブマイグレーション手法 [22][32] と同じ手法を用いればよい．しかし，ネットワークインターフェイスカード (以後 NIC) やタイマーデバイス，割り込みコントローラーといった物理デバイスの状態の中には，ソフトウェアから直接アクセスできないものもある．PMM もソフトウェアシステムであり ，これらの状態は直接読み出したり書き込んだりできない．この課題を解決するために，本研究では，物理デバイスの仕様に基づいてこれらの状態を間接的に制御する手法を提案し，これによってライブマイグレーションを実現しその評価を行った．

PMM が物理デバイスの状態を制御するには，デバイスの仕様に沿った処理が必要であり，

それ故に提案手法はデバイスに依存した手法となる．つまり，新たなデバイスに対応する度に，デバイス固有の処理を実装する必要がある．しかしながら，ベアメタルクラウドにおける用途においてこれは大きな問題でないと考えられる．これには二つの理由がある．一つ目は，デバイス状態をマイグレーションする処理は，一般的なデバイスドライバがデバイスを制御する処理よりも実装量が少ないためである．例えば，Realtek RTL8169 NIC のマイグレーション対応には C 言語のソースコード約千行を要したのみである．読み書き可能な状態を扱う処理は単純であり処理の実装は非常に容易である．一方，読み出しや書き込みが不可の状態を扱う処理は読み書き可能な状態と比べて複雑となる．しかし，デバイスの状態のほとんどは読み書き可能な状態であり，読み出しや書き出しが不可な状態の数は少ない．このため，マイグレーション処理はデバイスドライバの処理と比べて規模が小さくなる．

二つ目は，IaaS クラウドのデータセンターで用いられるデバイスの種類は一般に出回っているクライアントマシンのそれと比べて少ないことである．ほとんどの IaaS ベンダーはハイパバイザがサポートしている一般的なハードウェアを利用している．例えば，VMWare は vSphere ESXi ハイパバイザ用のデバイスドライバを開発および管理している．それ故に，サーバーハードウェアに向けて，デバイス依存のソフトウェアを管理することは現実的な選択肢であるといえる．

本提案手法によるライブマイグレーションは，移動元マシンと移動先マシンが同一のハードウェア構成になっていることを想定している．すなわち，移動元と移動先のマシンでは，CPU やマザーボードの型番が同じであり，同一の PCI バスに同一型番の PCI デバイスが接続していることを想定している．また，移動先マシンは，移動元のマシン以上の容量のメモリを搭載していることを想定している．ベアメタルクラウドにおいて，データセンター内には同一構成の物理マシンが大量に運用されていることが普通であり，この想定は現実的な想定であるといえる．また，この手法は移動元マシンと移動先マシンで，ライブマイグレーション専用に使えるネットワークがゲスト OS が利用するネットワークとは別に存在することを想定している．これも，実際の運用で見られる構成であり，現実的な想定であるといえる．

4.2 ライブマイグレーション手法における関連研究

この章では，仮想マシン，OS, コンテナ，およびプロセスのライブマイグレーションに関する研究について述べる．

4.2.1 仮想マシンのライブマイグレーション

主要な VMM はライブマイグレーションに対応している．例えば，VMWare の VMotion[33], XenSource の XenMotion[22] といった機能があり，同様の機能は KVM[34], Hyper-V にもある．ライブマイグレーションにより IaaS における OS 無停止での事前メンテナンス [15] やハードウェア障害発生前に OS を待避させる proactive fault tolerance[16, 17, 18] が実現され，IaaS のサービス可用性の向上が図れる．VMM によるのライブマイグレーションは既に確立された技術である．この技術は，仮想 CPU, メモリ，および仮想デバイスのすべての状態を移動元マシンの VMM から移動先マシンの VMM へ転送し新たな VM に状態を復元することでライブマイグレーションを実現するものである．仮想化環境において仮想 CPU, メモリ，および仮想デバイスの状態は全て仮想マシンモニタからアクセス可能であることがこの技術を実現可能にしている．しかし，一般的にマシンの仮想化処理は性能劣化が不可避である．第 2.1 節 で述べた通り，仮想化によるオーバヘッド低減の

ために，これまで多くの労力が注がれてきている．しかしながら，仮想マシンモニタによるオーバーヘッドはハイパフォーマンスコンピューティングなどの高い負荷を要求する用途では依然として無視できない [1, 35]．結果として，これらの用途に向けた IaaS としてベアメタルクラウドが登場するに至っている．

いくつかの研究ではダイレクトアクセスできるデバイスを伴う仮想マシンのライブマイグレーション手法を提案している [36, 37, 38]．PCI pass-through といった機能は，ゲスト OS が物理デバイスを仮想化なしに直接利用することで，仮想化によるオーバーヘッドは著しく削減する手法である．PCI pass-through デバイスを伴う環境でのライブマイグレーションを実現できれば，I/O の高速化とライブマイグレーションを両立できる．しかし，パススルーされている物理デバイスの状態を転送することが課題となる．なぜなら，VMM は パススルーデバイスと OS 間の通信に介在できず，また物理デバイスの状態を直接制御できないためである．この課題を解決するために，Nomad[36] では OS のデバイスドライバとユーザーレベルのライブラリを改変している． Kadav と Swift[37] は，ゲスト OS のカーネルに shadow driver という機構を導入している． この機構は，ゲスト OS のデバイスドライバの状態と復元を効率的に行うことができる．CompSC では，ゲスト OS 内の改変されたデバイスドライバと Xen ハイパーバイザー内のエミュレーションレイヤーと協調することでデバイスの状態を転送している．これらのアプローチは物理デバイスの状態に関する問題を効率的に解決しているものの，ゲスト OS に依存した手法となっている．先述の通り，ベアメタルクラウドにおいて，OS への依存は避けるべきである．

SRVM[19] は，SR-IOV デバイスを用いることで，この OS 依存への問題に対応している．SR-IOV は PCI デバイスの機能であり，デバイスのインターフェイスをデバイス自身で多重化するものである．典型的な用途では，それぞれの多重化されたインターフェイス (Virtual Function, VF) を PCI pass-through で各仮想マシンに割り当てる．SR-IOV と PCI-passthrough を組み合わせることで，OS と VF 間の通信に VMM はほとんど介入せず，また VM 間の調停はデバイスが行うため，性能劣化は無視できる程度に小さい．SRVM では，ダーティメモリの追跡と SR-IOV VF のチェックポインティングをゲスト OS のサポートなしで行うことで，物理デバイス状態の転送を実現している．しかしながら，SRVM は SR-IOV デバイスを PCI-passthrough で割り当て高速化しているのみで，他のハードウェアに関しては従来の VMM 同様に仮想化する必要がある．例えば，割り込みコントローラーやタイマーといったコアなデバイスに関しては依然として仮想化を要する．これは，SRVM がマルチテナント環境での VMM を想定しており，これらのデバイスを複数の VM や VMM で共有しなければならないためである．それに加えて，SR-IOV は一部のデバイスでのみ対応されている機能であること，SR-IOV の VF はゲスト OS で動作する専用のデバイスドライバ (VF ドライバ) を要することなど導入に際して制約がある．

4.2.2 OS のライブマイグレーション

OS のライブマイグレーション機能を OS 自身の機能として実装する手法も提案されている [39, 20]．Hansen[39] らは，OS でのライブマイグレーション手法のプロトタイプを 2 つ提案した．一つ目は，L4 マイクロカーネルと L4 タスクとして動作する Linux (L4Linux) を用いた手法である．このシステムは L4 Linux のメモリイメージを pre-copy で転送することでライブマイグレーションを実現している．このために，IPC を介したページング機構と再帰的な L4 アドレス空間を用いている．二つ目のシステムは XenoLinux と呼ばれる Linux の Xen ポーティングを用いた OS 自身のマイグレーション手法である．これらのシステムは

Linux に大規模な変更が必要である．それ故に，ゲスト OS に強く依存する手法となる．さらに，これらはメモリの状態しか転送しておらず，物理デバイスの状態は転送していない．

物理デバイス状態のマイグレーションの問題を解決するために，Kozuch らは OS のサスペンドとレジュームの機能を用いた手法を提案した [20]．彼らの提案する手法ではデバイスドライバによってデバイスとデバイスドライバの状態を転送する．OS レベルでのライブマイグレーション手法は，マシンの仮想化を要しないため，仮想化によるオーバヘッドを取り除くことができる．しかし，OS 自身でのライブマイグレーション手法は OS への依存が避けられないため，ベアメタルクラウドには不向きである．

OpenVZ のようにライブマイグレーションに対応しているコンテナ型仮想化システムもある [40]． コンテナはカーネルが提供している資源の一部分を隔離し，一つの仮想的な OS 環境を作る．例えば Linux におけるコンテナでは，プロセスの名前空間やファイルシステムの空間などの一部分を一つのコンテナとして隔離し，あたかもホスト OS から独立した Linux 環境として動作する．また，コンテナは一つのカーネル上で複数動作させられる．コンテナ内のファイルシステムやプロセスといった資源は，ホストや他コンテナから隔離され依存関係が少ないため，コンテナ内の資源を移動先マシンへ転送することでライブマイグレーションを実現できる．OS が提供する資源の一部を隔離して提供するコンテナ型仮想化は，仮想ハードウェアの組を作成しそのうえで別の OS を動作させるマシン仮想化に比べてオーバヘッドが小さい．しかし，仮想マシン環境と異なり，ゲスト OS とホスト OS のカーネルは共有されているため，仮に IaaS としてコンテナを提供すると，ユーザーからカーネルを選択したり改変したりできなくなる．

4.2.3 プロセスマイグレーション

プロセスマイグレーション [41, 42] とは，移動元の OS と移送先の OS が協調してプロセスをマイグレーションする手法であり，1980 年代に活発に行われていた研究テーマである．しかしながら，プロセスのマイグレーションは移動元 OS での依存関係の解決が大きな問題となった．例えば，あるプロセスをマイグレーションした後もそのプロセスは動作を維持するために移動元 OS にあるプロセスと通信する必要がある． この問題を解決するために，Zap [43] はプロセスドメインを導入した．プロセスドメインはプライベートな名前が付けられたプロセスのグループを提供する． これにより，プロセスグループを仮想化されたシステムの様に扱うことができるものの，この手法においても，ホスト OS の変更が必要となる．

4.3 提案システムの設計

本稿で提案するライブマイグレーションシステムの基本的なコンセプトは仮想マシンにおけるライブマイグレーション手法と同じである．つまり，マシン全体の状態をネットワーク経由で移動元マシンから移動先マシンへ転送することでライブマイグレーションを実現する．しかし，本研究で対象とする環境では移動元と移動先のマシンは仮想マシンではなく物理マシンであるため，物理マシンの状態，中でも物理デバイスの状態にアクセスするための新たな手法が必要となる．この章では，提案するライブマイグレーションシステムのアーキテクチャの概要を示し，マイグレーションを実現するために転送が不可欠な状態について議論する．その後，物理デバイスが持つ読み出し不可の状態と書き込み不可の状態をマイグレーションするための手法について述べる．

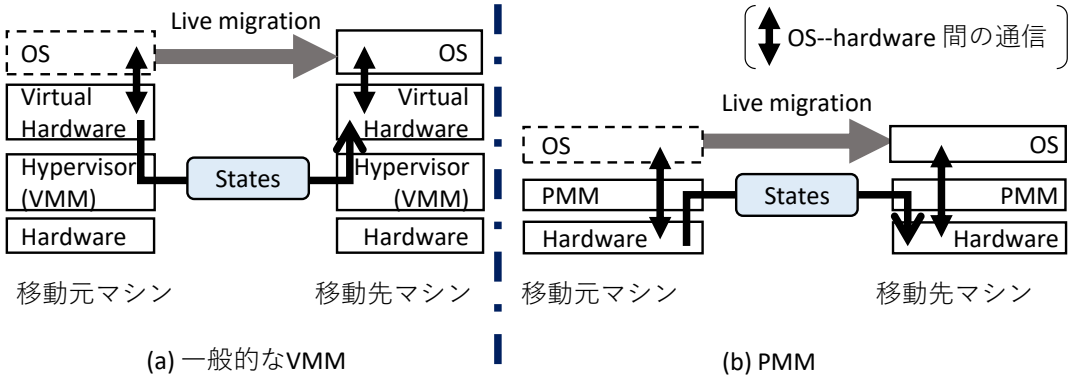

図 4.1　仮想マシンのライブマイグレーションと提案手法の比較

4.3.1　提案システムの全体像

先述の通り，PMM によるライブマイグレーションは物理マシンの状態を転送することで実現する．これを説明するために，図 4.1 に仮想マシンのライブマイグレーションと物理マシンのライブマイグレーションの比較を示す．現在実用化されている VMM による OS ライブマイグレーションでは，VMM が仮想ハードウェアの状態を移動元マシンから移動先マシンへと転送している．仮想ハードウェアの状態は，VMM によって全て管理されているため，この状態転送は容易に実現できる．一方，PMM が動作する環境では，OS は仮想ハードウェアではなく物理ハードウェアを扱う．このため，PMM によるライブマイグレーションでは，物理ハードウェアの状態を転送する必要がある．これを踏まえ，PMM によるライブマイグレーションの流れは以下のようになる．

1. 移動元マシンの PMM はライブマイグレーションの前に最小限の処理で物理デバイスの状態を保存する．
2. ライブマイグレーション実行中，移動元マシンの PMM はネットワークを介して移動先の PMM へマシンの状態を転送する．
3. 移動先マシンの PMM はマシン状態を受け取り，これをマシンの物理ハードウェアに復元する．
4. ライブマイグレーション実行後，移動元の PMM は次のマイグレーション要求が来るのを待機するかシャットダウンする．

ハードウェアの仮想化なしでライブマイグレーションするため，移動元マシンと移動先マシンで物理ハードウェアの構成が異なっている場合，この差異を OS に対して隠蔽できない．例えば，Intel 製 CPU のマシンから AMD 製 CPU のマシンへ移動すると，Intel 製 CPU にしかない機能や状態を AMD 製 CPU 上で仮想化なしで提供することは不可能である．そこで本手法は，移動元マシンと移動先マシンのハードウェア構成が同一であることを想定する．具体的には，移動元マシンと移動先マシンで，CPU の型番が同一であり，デバイスの型番や PCI バス上での位置が同一であることを想定する．また，メモリ容量に関しては，移動元マシンのメモリ容量より移動先マシンのメモリ容量が大きいことを想定している．ベアメタルクラウドのデータセンタにおいて，同一構成のマシンが大量に運用されているため，この条件は容易に満たせると考えられる．

ライブマイグレーションを実現するにあたり，PMM が転送しなければならないハードウェアの状態には CPU の状態，メモリ上のデータ，及び周辺デバイスの状態がある．状態の転送において，CPU の状態やメモリ上のデータの転送については，物理マシンモニタにおいても既存のライブマイグレーションと同様の手法が利用できる．CPU やデバイスの状態には，レジスタ内の値，CPU の動作モード，命令実行後のフラグ，各種機能の設定状態などがある．例えば，CPU が 64 ビットモードで動作しているか 32 ビットモードで動作しているかという設定状態や，CPU が直前に実行した命令の結果を反映したフラグレジスタの値などがある．PMM は CPU の仮想化支援機能を用いて CPU の状態の取得と復元が行える．また，PMM は物理メモリ空間からメモリ上のデータを読み書きすることで，メモリ状態の取得と復元を行う．このように，CPU とメモリに関する状態は，PMM においても比較的容易に転送できる．

一方，物理デバイスの状態は既存のライブマイグレーションとは異なる手法が必要となる．この理由は，物理デバイスはソフトウェアから直接読み出し不可な状態や書き込み不可な状態をもつことがあるためである．読み出し不可な状態は，移動元マシンの PMM が状態の転送前にどのように物理デバイスから状態を取得するかが問題となる．また，書き込み不可な状態は，移動先マシンの PMM が受信した状態をどのように物理デバイスに復元するかが問題となる．このため，本提案手法において，物理デバイスの状態を如何にして取得および復元するかが，PMM によるライブマイグレーションを実現する上で大きな技術的課題となる．

この課題を解決するために，PMM は物理デバイスをその仕様に基づいて観察および制御することで，間接的に目的の状態を取得及び復元する．この手法では，デバイスの動作とデバイスの状態が密接に関係していることを利用する．具体的には，デバイスの挙動はデバイスの状態によって変化すること，および，デバイスが動作するとデバイスの状態が変化することを利用する．読み出し不可の状態は， ゲスト OS からの読み出し不可状態への書き込みアクセスの監視，およびマシン切り替え時のデバイスの動作を監視することで間接的に状態を取得する．また，PMM は物理デバイスを制御し，内部の状態遷移を意図的に引き起こすことで，書き込み不可の状態を間接的に復元する．これらの処理を実現する上で知る必要のあるデバイスの動作と状態の関係は，デバイスの仕様として知ることができる．全ての物理デバイスを復元することは非常に難しいものの，ライブマイグレーションを実現するために不可欠な状態の転送は現実的なコストで実現可能であると考えられる． この不可欠な状態については，次の節で述べる．

4.3.2 転送するデバイスの状態

PMM によるライブマイグレーションにおいて，転送が不可欠な物理デバイスの状態は 2 種類ある．一つは設定状態，もう一つは処理状態である．一つ目の設定状態は，ソフトウェアがデバイスの動作を設定している状態である．一般的に，OS はデバイスの挙動を設定するためにこの状態を変更する．例えば，OS がネットワークデバイスを 100 Mbps ではなく 1000 Mbps で利用するために，該当する設定状態を変更する．設定状態は，移動先マシンのデバイスが移動元マシンのデバイスと同様の設定で動作し続けるために転送する必要がある．設定状態は，ソフトウェアが設定する状態であり，デバイスの処理によって状態が変化することはない．

二つ目の処理状態は，デバイスの処理にかかわる状態である．デバイスの処理にかかわる状態であるため，デバイスの処理に従ってデバイス自身によって更新され変化していく．また，逆にデバイスの処理はこの処理状態によって変化する．処理状態の例として，デバイスの

リセット処理が完了したか否かを示す状態がある．別の例としては，NIC がパケットを転送する際に，どのパケットを既に送信しており，次にどのパケットを転送するかの状態がある．

どちらの種類の状態もゲスト OS がライブマイグレーションの前後で動作を継続するために転送が必要なものである．もし設定状態が転送されなければ，ゲスト OS はマイグレーションの前後でデバイスの設定が変わっていることを把握していないため，OS のデバイスドライバは変更後の設定にそぐわない形でデバイスを利用および制御してしまう．例えば，100 Mbps の設定に替わってしまった NIC に対して，1000 Mbps の速度でデータ転送を要求する，といったことが考えられる．この場合，大量のパケットがドロップし，ネットワーク接続が不安定になる．また，もし処理状態が欠落すると，デバイスドライバが把握しているデバイスの状態と実際のデバイスの状態が乖離してしまう．例えば，リセット処理の状態が欠落すると，デバイスドライバはリセット完了前にデバイスの利用を始めてしまったり，逆にリセット完了を検知できずにいつまでもリセット完了を待ち続けて処理が止まってしまったりする可能性がある．

デバイスの状態すべてが OS の動作継続に不可欠であるわけではなく，上記の設定状態と処理状態以外は転送する必要がない．例えば，受信パケット数などの統計値は OS が継続的に動作するのに不可欠な状態ではない．なぜなら，統計値はデバイスの処理や処理状態に影響を与えないためである．また，デバイスの統計値は移動元のマシンと移動先マシンでそれぞれ持つべきであり，転送しない方が現実的である．移動元マシンの統計値を必要とするアプリケーションのために，PMM で統計値を転送し，その値を転送先で仮想化して見せることはできる．しかし，性能のオーバヘッドを伴うことと，移動元マシンの統計値を参照する必要があることは少ないと考えられるため，基本的には統計値の転送や仮想化は行わない．

転送が必要ではない状態の別の例として，割り込みの状態がある．割り込みの状態とは，どのデバイスからのどのような理由で割り込みが発生しているか否かという状態である．割り込みの状態は，個々の物理デバイスで起こる非決定的なイベントに強く依存するものであるため，割り込みの状態のみを直接転送するべきではなく，デバイスの設定状態と処理状態を適切に転送した結果として必要な割り込みが移動先で発生するようにすべきである．また，故障などが原因で発生している割り込みを転送すると，移動先のデバイスが故障していなくても，故障したかのように扱われてしまうため，このような割り込みの状態は転送するべきではない．例えば，デバイスの故障によって発生するマスク不可割り込みの状態は，移動先のデバイスが壊れていないため，転送するべきではない．

コマンドレジスタの値も，不要な状態である． コマンドレジスタは OS からコマンドを受け取るためのインターフェイスとしてふるまう．このため，コマンドレジスタの値はデバイス内部の状態とは直接関係しない．例えば I/O APIC のレジスタの中には，OS が割り込み処理の完了を通知するためのレジスタがあり，割り込みハンドラの処理が完了した際に OS がこのレジスタの指定されているビットに 1 を書き込む．この値は読み出し不可であるが，単にレジスタへの書き込みという形でソフトウェアから割り込みコントローラへ処理を要求しているだけのため，書き込まれた値を読み出すことには意味はない．コマンドレジスタのもう一つの例は，NIC が持つ送信要求を受けるためのレジスタである．このレジスタの場合，送信要求を受け取るタイミングとマイグレーションを開始するタイミングによっては問題が発生するように見えるものの，実際には，特に問題はない．以下で具体的に考察する．問題となり得そうなタイミングは，送信要求の直前でライブマイグレーションが行われた場合と，逆に送信要求直後にライブマイグレーションが行われた場合である．もし，ライブマイグレーションが送信要求の前に行われた場合，送信要求のコマンドはマイグレーション完了後に移動先のマシンで実行される．送信すべきパケットなどを含め，送信処理に関する状態もすべ

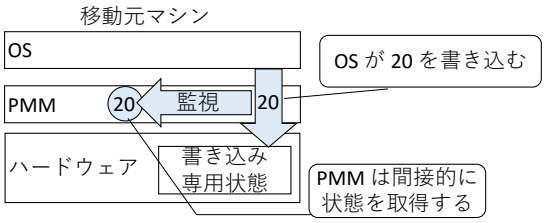

図 4.2　書き込み専用状態の取得

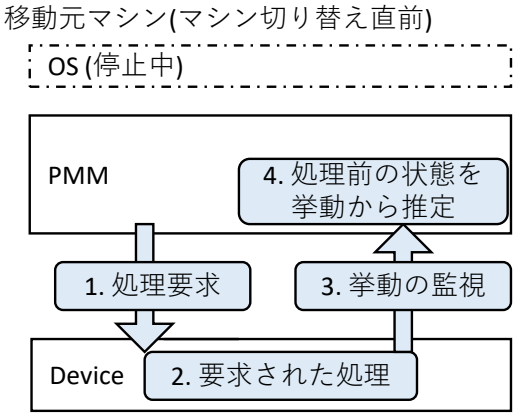

図 4.3　読み出し不可状態の取得

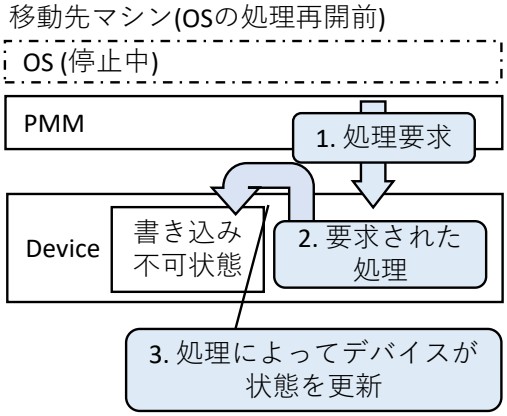

図 4.4　書き込み不可状態の復元

て転送されているため，送信要求はマシンが切り替わっていない場合と同様に移動先のマシンで実行される．逆に，ライブマイグレーションが送信要求のコマンド発行直後に行われた場合，移動元マシンのデバイスが送信要求を受けて送信要求を行う．ライブマイグレーションの際には，この処理の結果を反映した状態を転送する．それ故に，コマンドとライブマイグレーションのタイミングにおいては，特に問題はない．

4.3.3　読み出し不可状態の読み出し

読み出し不可の状態は 2 種類に分類できる．1 つ目は書き込み可能でかつ読み出し不可の状態，すなわち Write-Only な状態である．ソフトウェアはこの種類の状態をデバイスに直接書き込めるものの，デバイスから読み出すことはできない．これらの状態は主に OS によって更新される設定状態である．二つ目はソフトウェアから読み出しも書き込みもできない内部状態である．この種類の状態は主にデバイス自身によって更新される処理状態である．どちらの種類の状態においても，PMM は直接読み出すことによって状態を取得することはできない．

書き込み専用レジスタに格納されている状態を取得するために，PMM は OS から当該レジスタへの書き込み I/O を監視する．図 4.2 に書き込み専用の状態を取得する方法を示す．ゲスト OS が書き込み I/O を発行した際に，PMM はその書き込み I/O に介入し，書き込もうとしている値を記録した後，その書き込み I/O を物理ハードウェアに転送する．つまり，書き込み時の値を記録しておくことで，間接的に状態を取得している．PMM はライブマイグレーション実行時において，各書き込み専用レジスタの最後に書き込まれた値を移動先マシンの PMM に送信する．書き込み状態は設定状態であり，書き込まれた後にデバイスの処理によって状態は変わらないため，マシンの切り替え直前に値を読む必要はない．

監視すべき I/O のアドレスは，デバイスの仕様から決定できる．監視すべき I/O のアドレスはデバイス毎に異なり，デバイスごとに対応する必要があるものの，基本的に書き込み専用のレジスタについてのみ対応すればよいため，対応作業は，デバイスドライバを記述するのに比べ手間は少ない．書き込み I/O への介入はオーバヘッドを生じさせるものの，今日の PC アーキテクチャにおいては書き込み専用レジスタの数は少なく，それらに対するアクセスも頻繁ではないため，生じるオーバヘッドはごく小さなものである．実際の監視対象の I/O アドレスについては，第 4.4.4 節 章にて示す．

内部状態の取得について図 4.3 に示す．内部状態の取得は，PMM が物理デバイスの挙動を監視することで実現する．内部状態を取得するために，移動元マシンの PMM はライブマイグレーションでのマシン切り替え直前に物理デバイスを制御し，その挙動を観察する．デバイスの仕様と観察した物理デバイスの挙動から，PMM はデバイスの内部状態を推定できる．なぜなら，デバイスの挙動は，デバイスの状態に依存するためである．物理デバイスの挙動監視は性能劣化を生むため，ライブマイグレーションで OS の処理が移動元マシンから移動先マシンへ切り替わる直前にのみ行う．この時，移動元マシンでは OS が既に停止しているため，挙動の監視処理を行っても，OS の性能劣化は起きない．また，OS が停止してデバイスへ処理要求を出さなくなるため，デバイスの動作も停止する．デバイスが停止すると，PMM はデバイスの挙動を監視できないため，PMM は OS に代わってデバイスに処理要求を出す．PMM はこの処理要求に対するデバイスの挙動を元に，デバイスの状態を推定する．PMM はこの状態の推定を行うために，デバイスの処理と処理状態の状態遷移の関係をデバイスの仕様から把握している．

PMM がデバイスの処理を要求し，それに従ってデバイスが処理を行うことで，元々の処理状態は破壊されてしまう．しかし，PMM はこの処理の挙動から処理要求前の状態を推定するため，状態の取得において状態が破壊されることは問題とならない．また，状態の破壊は移動元マシンで OS が停止した後に起こり，その後移動元マシンで OS の動作を再開する必要はないため，状態が破壊されても OS の動作に影響しない．

この状態推定の処理は移動元マシンと移動先マシンの切り替え時に行うため，マシン切り替え時の処理が VMM によるライブマイグレーションと比べて多くなる．このため，マシン切り替え時における OS のダウンタイムが VMM による手法と比べて増加すると考えられる．しかし，第 4.5.5 節 に示す測定結果からダウンタイムへの影響は大きくないと言える．

デバイスの状態を取得する最中にデバイス状態が変化すると，状態が正しく取得できない．これを避けるためにも，状態取得の処理は OS の停止後に行う．また，デバイスの状態が非決定的に変化しないようにするために，ある程度デバイスの設定を変更する必要がある．例えば，NIC の受信処理に関する状態を取得する前には NIC がパケットを受信しないようにしておく．

4.3.4 書き込み不可状態の復元

書き込み不可状態の復元について図 4.4 に示す．PMM は物理デバイスを制御し，対象の書き込み不可状態を目的の状態へと変化させるような状態遷移を発生させることで間接的に状態を復元する．ライブマイグレーションのために転送すべき書き込み不可状態は，全て処理状態である．なぜなら，設定状態であれば OS から設定可能でなければならず，書き込み可能なためである．対象となる書き込み不可な状態は処理状態であるため，デバイスが処理することで対象の状態は変化する．そこで，デバイスに対象の状態を変化させるような処理を要求し，間接的に対象の状態を変化させる．書き込み不可状態の復元は，移動先マシンで

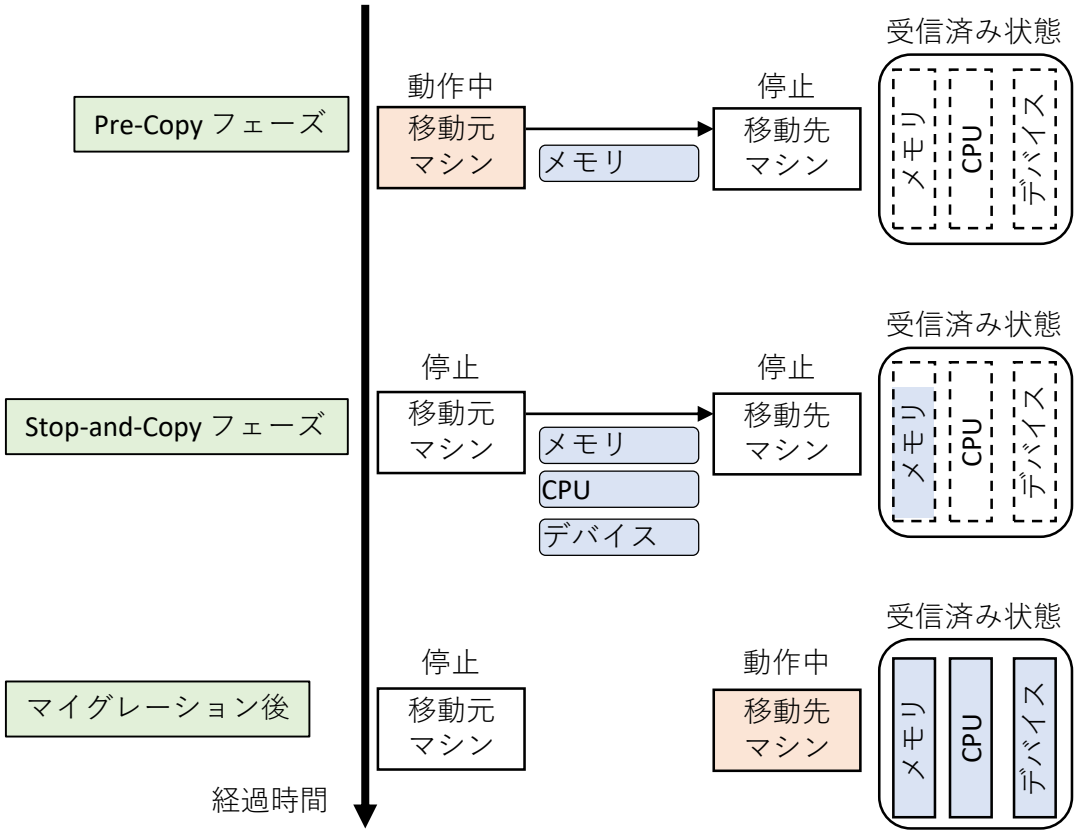

図 4.5　Pre-copy によるマイグレーションの処理流れ

PMM がデバイスに処理要求を出すことで行う．PMM はデバイスの仕様から，どのような処理がどのように処理状態を変化させるか把握している．

この処理は，移動先の PMM が OS の動作を再開させる前に行う．このため，これらの処理が OS の処理性能へ影響することはない．一方で，マシンの切り替えに要する時間が増大し，切り替え時における OS のダウンタイムが増大すると考えらえる．しかし，前節で述べたのと同様に，第 4.5.5 節 に示す測定結果からダウンタイムへの影響は大きくないと言える．

デバイスの状態を復元中にデバイス状態が不意に変化すると，状態が正しく取得できない．これを避けるために，前節の状態推定処理と同様に，状態が非決定的に変化しないよう，ある程度デバイスの設定を制御する必要がある．

4.4　実装

本提案手法を評価するために，物理マシンモニタによるライブマイグレーションのプロトタイプを実装した．本節ではこのプロトタイプの実装について述べる．以降では，本システムによるライブマイグレーションの流れ，CPU とメモリ状態の転送アルゴリズム及び物理デバイスの状態の転送に関する実装について述べる．また，マルチコアシステムへの対応やネットワーク接続の維持に関する実装についても述べる．

4.4.1 ライブマイグレーション処理の流れ

OS 非依存なライブマイグレーションでは，先述の通り CPU の状態, メモリ上のデータ，およびデバイスの状態を転送する必要がある．この中で，メモリ上のデータはサイズが大きく，転送に時間を要する．例えば，4GB のメモリデータを転送する場合，1GbE NIC を用いると 4 (giga byte) × 8(bit) ÷ 1(giga bit/sec) = 32 (sec) となり，少なくとも 32 秒の時間を要する．故に，メモリ上のデータの転送は OS が動作している状態で行わなければダウンタイムが大きくなってしまうため，メモリ上のデータは OS を動作させながらバックグラウンドで転送する．

ライブマイグレーションの実装にはメモリ転送手法で大きく 2 つに分けられる．1 つ目は Pre-Copy [22] である．これは，OS の処理が切り替わる前にメモリ上のデータを転送する手法である．移動元マシンで OS が動作している間にメモリを転送するため，転送した領域のメモリが転送後に書き換えられる可能性がある．これを移動先のマシンにも反映させるために，転送後に書き変わったメモリ領域は再送する．このため，OS が高い頻度で広範囲にわかってメモリ書き換えを行うと，再送が多発し，マイグレーションが終わらない事態が発生する．二つ目は Post-Copy と呼ばれる手法である [32]．Post-Copy では，OS の処理はメモリ転送前に移動元マシンから移動先マシンへ切り替わる．その後，移動先でメモリアクセスが発生したときに，アクセス先のメモリデータが転送されていなければ，直ちにそのデータを転送する．このため，Pre-copy のように再送が多発することはない．しかし，OS 移動後しばらくの間，メモリアクセスの速度が不確定に揺れることが難点である．

今回のプロトタイプは実装が比較的容易でかつ多くの既存システムで対応されている Pre-copy 方式で実装したため，ここでは Pre-copy でのライブマイグレーション処理の流れについて説明する．図 4.5 に Pre-copy におけるマイグレーションの処理流れを示す．Pre-copy による手法では，マシンを切り替える前にメモリ上のデータをバックグラウンドで転送する pre-copy フェーズと，マシンを切り替える stop-and-copy フェーズがある．pre-copy フェーズでは，PMM はメモリ上のデータを移動元マシンから移動先マシンへとバックグラウンドで転送する．この際，ゲスト OS は動作しているため，メモリ上のデータは転送後に一部書き換えられる．このままでは，ゲスト OS によるデータの変更が移動先マシンに反映されないため，PMM はゲスト OS により書き換えられたデータを再送する．再送を繰り返し，転送するメモリの残り容量が十分小さくなった時点で，システムは pre-copy フェーズから stop-and-copy フェーズに移行する．stop-and-copy フェーズでは，移動元の PMM は OS を停止し，残りのメモリデータと，CPU の状態，およびデバイスの状態を移動先マシンへと転送する．CPU とデバイスの状態は，データ量が少なく，またメモリ上のデータの残量も十分小さくなっているため，これらはごく短い時間で転送できる．このため，stop-and-copy フェーズ中には OS が停止するものの，この停止時間は 1 秒以内であることが多く，長くても数秒程度である．全ての状態の転送が終わると，移動先マシンの PMM が状態を復元し，OS の処理を再開する．

今回のプロトタイプ実装では pre-copy アルゴリズムを採用したものの，post-copy[32] を採用することも可能であると考えている．なぜなら，本研究で提案するシステムは主に stop-and-copy フェーズにおける物理デバイスの状態転送に関する部分が主であり，その他の部分は既存のアルゴリズムと同様であるためである．また，pre-copy においても post-copy においても物理デバイスの状態転送の処理に大きな差がないと考えられるためである．

4.4.2 CPU 状態の転送

本手法では，CPU は Intel VT-x や AMD-V といった仮想化支援機能を持つことを想定している．これらは CPU の状態をメモリ上に格納したり，メモリ上から CPU の状態をロードしたりする機能を提供している．例えば Intel VT-x では，virtual machine control structure (VMCS)，AMD では virtual machine control block (VMCB) と呼ばれるデータ構造によって CPU の状態を管理している．今回の実装では Intel 製 CPU 向けに実装したため，以下では Intel VT-x について説明する．

Intel VT-x の VMCS は一般的な命令では取得や復元が難しい CPU の状態を保持しており，専用の命令でこれらの状態を読み書きできる．例えば仮想 CPU の命令ポインタやスタックポインタ，フラグレジスタなどを `VMREAD` 命令や `VMWRITE` 命令で直接読み書きすることができる．このため，ライブマイグレーションの際にはこのような状態でも PMM は容易に転送できる．VMCS はメモリ上に格納されているものの，CPU によってキャッシュされていたり，CPU の型番によって内部構造が異なっていたりするため，全ての状態はメモリコピーではなく `VMREAD` や `VMWRIE` 命令によって読み書きする．このことは，CPU の状態の取得や復元に要する命令数が増える要因となるが，第 4.5.5 節 に示すダウンタイムの測定結果から，この影響は小さいことがわかる．一方，汎用レジスタや多くの model-specific register (MSR)，SIMD 命令用の状態 は VMCS で管理されていない．しかし，PMM はこれらの値も自由に読み書きできるため，マイグレーションの際には，直接値を読み書きし転送する．今回実装したプロトタイプでは Intel 社製 CPU の VT-x にのみ対応しているものの，AMD 社製 CPU の AMD-v を用いても同様の実装は可能であると考えられる．実際の多くのライブマイグレーションシステムは両方の CPU でライブマイグレーションを実装している

4.4.3 メモリ転送

第 4.4.1 節 で述べた通り，本プロトタイプは pre-copy によるライブマイグレーションを実装している．pre-copy において，メモリ転送中に OS が書き換えたメモリページ，つまり dirty ページを検知し，再送する必要がある．このために，PMM は Intel VT-x が提供する extended page table (以下，EPT) を用いている．EPT の各物理メモリページの管理構造体の中には当該ページがゲスト OS によって変更されたか否かを示す dirty bit と呼ばれるビットが存在する．ゲスト OS が当該ページに書き込んだ場合，ハードウェアがこのビットをセットする．PMM はこのビットを走査することで，OS によってどのメモリページが書き換えられたかを把握する．PMM は Dirty ページを Bitmap で管理している．この Bitmap の各ビットは 1 つのメモリページが書き換えられているか否かを示す．このため，Bitmap のサイズは概ね (物理メモリ容量) ÷ (ページサイズ) ÷ 8 となる．例えば，4GB のメモリを搭載しているマシンでは，4GB ÷ 4KB ÷ 8 = 128KB 程度となる．このビットマップは全てのコアで共有する．

あるメモリページが dirty か否かはページテーブルの木構造の葉にあたるページテーブルエントリ (PTE) を読むことで判明する．しかし一般的に，ある PTE がどのゲスト物理アドレスに対応するかは木構造をルートから辿ること (ページウォーク) をしなければわからない．dirty bit を走査するためにページウォークが多発すると非常に遅い処理となる．PMM においてゲスト物理アドレスとホスト物理アドレスは常に同一であり，この関係はライブマイグレーションの前後でも変わらない．このことを利用すれば，ページウォークせずに PTE

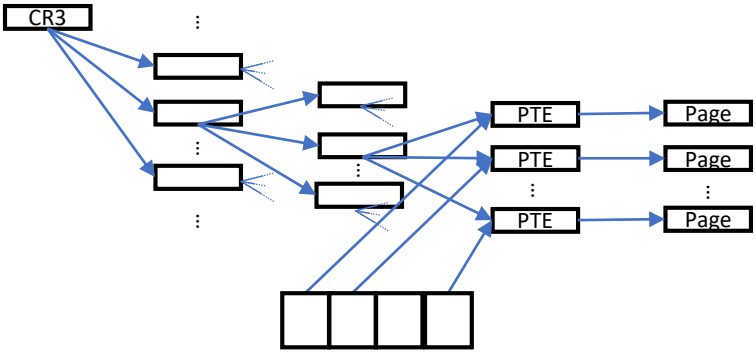

図 4.6　一次元配列による PTE アドレスの管理

のみ読んでもその PTE がどのゲスト物理アドレスに対応するかわかる．なぜなら，PTE が指すホスト物理アドレスがゲスト物理アドレスと同一であるためである．そこで，図 4.6 にあるように，木構造となっているページテーブルの末端の PTE へのポインタを一次元配列で管理し，これを走査することで高速化を図っている．

EPT エントリーの dirty bit を走査する際には，マルチコアシステムにおけるトランスレーションルックアサイドバッファ (以下，TLB) の扱いに注意を要する．各コアはゲスト OS が頻繁にアクセスするページの EPT エントリーを TLB 内でキャッシュする．また，あるコアは別のコアが持つ TLB の中身を読み出すことはできない．一般的な VMM では，ある瞬間において各コアは別々の VM を実行しておりコア毎にアドレス空間のマッピング異なるために，TLB はデータキャッシュと違いコア間のコヒーレンシを維持していない．このため，あるコアの書き込みによってセットされた dirty bit のうち，TLB でキャッシュされているものは，別のコアから走査できない．このような TLB のコヒーレンシに関する問題は，一般的に全てのコアで一斉に TLB の内容をメモリにフラッシュする TLB shootdown という手法で解決される．TLB shootdown でもこの問題は解決できるものの，TLB shootdown はコストが大きいため，Pre-Copy 中はすべてのコアが 1 秒に 1 回 dirty bit を走査し，その結果を共有メモリに反映する．また，Stop-and-copy 時，全てのコアが停止する直前に TLB をフラッシュする．このようにすることで，再送漏れを防ぐ．この実装は処理が複雑になることを避けられるものの，Stop-and-Copy 内で転送するメモリ容量の閾値を厳密に守ることができなくなる．具体的には，最後の TLB フラッシュで多くのメモリページが書き換えられていることが判明すると，Stop-and-Copy フェーズで転送するメモリ量は，閾値を超える．この問題は最後に TLB フラッシュしたのちに閾値を超えているか否かをチェックし，超えていれば再度 pre-copy フェーズに戻るという実装にすることで回避できる．今回のプロトタイプでは実装を簡易にするためにこの実装は行っていない．

今回実装したプロトタイプは，マイグレーション用の専用 NIC がマシンに搭載されていることを想定している．マルチコアシステムにおいて，全てのコアが NIC を使ってメモリ上のデータを転送しようとすると，コア間でデバイスのロックの競合が発生する．このロック競合は PMM の中で発生し OS の処理をブロックするため，マイグレーション中の OS の処理性能を著しく低下させてしまう．このロック競合を避けるために，本プロトタイプでは複数のコアではなく単一のコアで NIC を用いてメモリを転送する．ゲスト OS への性能劣化の影響を避けるには，最もアイドル時間が長いコアを選択してデータ転送を行わせるべきである．しかしながら，アイドル時間の長さの比較や，転送処理をコア間で移動させる実装は複雑と

なるため，本プロトタイプではメモリ転送を担当するコアは固定的に割り当てている．この実装が複雑となるのは，PMM は CPU スケジューラーのような機構を持たないためである．

他の解決策としては，マルチキュー機能がある NIC の利用も考えられるものの，今回用意した NIC にはマルチキュー機能がないため，単一コアで処理することとした．今回利用した NIC はマルチキューに対応していなかったものの，マルチキューに対応した NIC を用いれば，マルチコアでの転送処理の実現は NIC に関するロックなしに実現可能である．しかし，各コアでどのアドレスのメモリを転送するかを管理する機構が必要となるため，依然として課題は残ると考えられる．本稿ではマルチコアでのメモリ転送処理の並列化について多くは議論しないものの，マルチキュー対応 NIC を用いたうえで，コア間で適切にメモリ転送処理を分担することで，より効率的なメモリ転送処理を実現可能であると考えられる．

定期的に dirty bit を走査したりメモリ転送処理をしたりするには，pre-copy フェーズの間，PMM は定期的に制御を得る必要がある．VM exit が定期的に発生しないと，PMM は処理を進められないものの，提案手法において PMM は性能劣化を避けるために極力 VM exit が発生しないようにしている．PMM が pre-copy の定期的な処理を行うために，PMM は Intel VT-x が提供する機能の一つで，定期的に VM exit を発生させる機能である Preemption timer を用いる．今回実装したプロトタイプでは，メモリ転送を行うコアでは 1GbE のラインレートで転送するのに十分な頻度 (約 786 マイクロ秒に 1 回) で， その他のコアでは dirty bit の走査のために 1 秒に 1 回の頻度で VM exit を発生させている．また，Preemption timer はコアが C2 ステート以上のスリープ状態になると停止してしまう．これを避けるために，C2 ステート以上のスリープ状態に入ることを禁止している．

stop-and-copy フェーズを開始する際には，PMM は VM exit を介して制御を得る必要がある．しかしながら，もし PMM が何もしなければ，最悪の場合 VM exit は Preemption timer によって 1 秒後にしか発生しない．この場合，PMM 内ですべてのコアがこの VM exit を待つ必要があり，このままではダウンタイムが増大してしまう．この遅延を避けるために，メモリ転送を担当しているコアは stop-and-copy フェーズを開始する際にすべてのコアに対して inter-processor interrupt (以下 IPI) を送出する．この IPI の種別としてマスク不可割り込み (non-maskable interrupt, 以下 NMI) を選択する．また，PMM は通常の割り込みでは VM exit を起こさないようにする一方で，NMI を受信した際に VM exit を発生するように設定する．このようにすることで，全てのコアで先述の IPI を契機に VM exit が発生し，Preemption timer を待つことなく直ちに stop-and-copy フェーズに移行する．IPI に NMI を用いている理由は，NMI は VT-x の Exception Bitmap を用いて NMI のみで VM exit を発生させる設定ができるためである．NMI 以外の割り込みでは，VT-x は全ての割り込みで VM exit を発生させるか，逆にすべての割り込みで VM exit を発生させないかのどちらかの設定しかできなかった．

現在の実装において，pre-copy に関して以下の二つの制約がある．一つ目は，本プロトタイプはマイグレーションに必要最低限のメモリ領域を識別することなく，全てのメモリ上のデータを転送する．PMM は BIOS から得たメモリマップに従ってゲスト OS が利用可能な物理アドレス空間上のメモリを転送する．このため，pre-copy フェーズにおいて転送するデータ量は既存の VMM と比較して大きくなる．二つ目は pre-copy におけるデータ転送速度と pre-copy 終了の閾値が固定されており調整できないことである．現在の実装では，転送速度は NIC のラインレートである 1 Gbps であり，閾値は 64 MB である．これらの制約は設計に起因するものではなく実装に起因する問題である．また，これらの制約はライブマイグレーション実行中の性能やマシン切り替えのダウンタイムにのみ影響するものであり，通常時の性能には影響しない．QEMU/KVM などではこれらの機能が実装されている．

今回のプロトタイプ実装には，メモリ管理の実装に最適化の余地が残されている．EPT でのページ管理は 4KB, 2MB, 1GB と 3 種類のページサイズが選択できる．また，dirty ページは EPT のページ単位でしか把握できない．このため，ライブマイグレーション時の dirty ページの追跡に関しては，小さい単位で追跡した方が再送するメモリサイズを抑えられるため，4KB 単位でページ管理する方が望ましい．今回のプロトタイプ実装ではこのような理由から 4KB でのページ管理を行っている．一方で，2MB や 1GB ページを用いることで，EPT による物理アドレス変換の処理が高速になり，また TLB のヒット率も向上することが知られている．このことは，メモリアクセス速度の高速化に寄与するため，性能面では 2MB や 1GB のページを用いた方が有利である．このことから，ライブマイグレーション直前まで 2MB や 1GB 単位でページを管理し，pre-copy 時に 4KB ページングへと切り替える方針が考えられる．しかしながら，今回は実装を簡略化のためにこのような実装は行っていない． この実装の有無によって，通常動作時のメモリアクセス性能に影響すると考えられる．

4.4.4 物理デバイス状態の転送

物理デバイスの状態の取得

ライブマイグレーションの処理中，移動元の PMM は物理デバイスの状態を取得する必要がある．これを取得するために，デバイスがもつレジスタを 3 つに分類する．一つ目は読み出し可能なレジスタ，二つ目は書き込み専用レジスタ，そして三つ目は内部レジスタである．読み出し可能なレジスタの値は，PMM が直接読み出すことで取得可能である．書き込み専用レジスタの値は，ゲスト OS から当該レジスタへの書き込みを監視することで間接的に取得する．内部レジスタの値は，stop-and-copy フェーズ中にデバイスの挙動を観察することによって状態を取得する．

書き込み専用レジスタの状態の取得

書き込み専用レジスタの例としては programmable interval timer (PIT) の一部のレジスタがある．PIT は一定周期の割り込みを生成するために OS によって利用されるタイマデバイスである．割り込み周期を設定するためのレジスタが書き込み専用であるため，PMM は当該レジスタの値をゲスト OS が書き込んだ値を監視することによって間接的に取得している．

いくつかのデバイスのレジスタについては，PMM は一つの I/O だけでなく， I/O のシーケンスを監視しないといけない場合がある．なぜなら，これらの I/O は直前の I/O に依存して動作が変わるためである．programmable interrupt controller (PIC) はこのような処理が必要となるデバイスの一つである．

図 4.7 は PIC における書き込み専用状態である ICW の取得について示している．PIC の初期化において，ソフトウェアは initial control words (ICW) と呼ばれる 4 つの値 (ICW1–ICW4) を PIC のレジスタに書き込む．最初の ICW (ICW1) は 0x20 番ポートに書き込まれる．この際，0x20 ポートに書き込まれる値の特定ビットがセットされているとデバイスは ICW1 として扱う．もしこのビットがセットされていなければ，ICW1 として扱われず，別のレジスタへの書き込みとなる．残りの ICW (ICW2–ICW4) は 0x21 番ポートに連続して書き込まれる．また，0x21 番ポートへの書き込みは，ICW1 の直後に連続する 3 つの書き込みのみが ICW として扱われ，それ以外は別の値をして解釈される．このように ICW への書き込みは，別のレジスタと同じ I/O ポートを用いており，当該レジスタへの書き込みか否かは書き込み内容と直前の I/O によってしか判断できない．このため，ICW の状態を取得するには，常に特定ポートのへの書き込みを監視し，その中身を確認する必要がある．ICW1

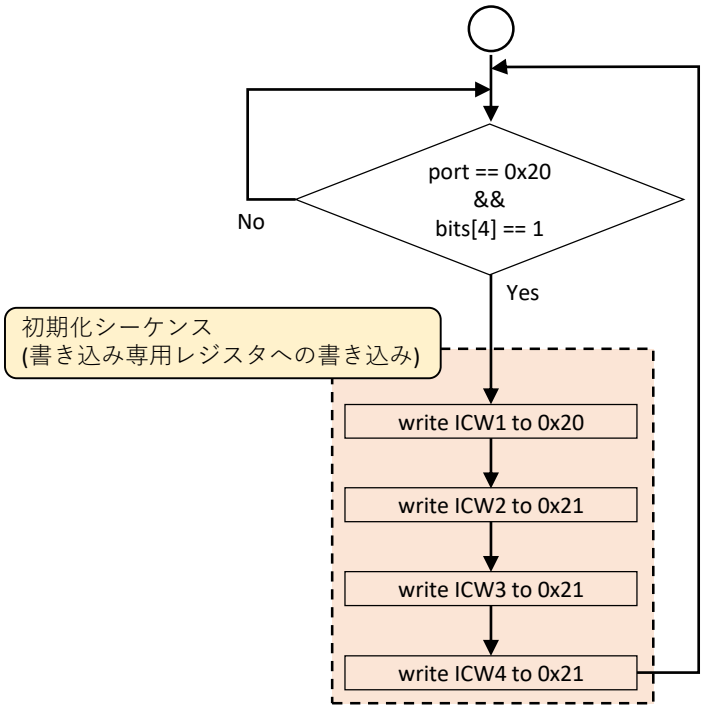

図 4.7　PIC における ICW の状態取得

表 4.1　PIO ports monitored during normal execution

Device	I/O port	State
PIC	0x20 and 0x60	ICW 1
PIC	0x21 and 0x61	ICW 2–4 and other configurations
PIT	0x40	Timer interval
PIT	0x43	Mode of the timer

が用いるポートには，End of Interrupt (EOI) と呼ばれる I/O も発生する．EOI は割り込み処理が完了したことを割り込みコントローラーに通知するための I/O であり，割り込みのたびに発行される．このため，この I/O が発行される頻度は非常に高く，この I/O ポートを監視するコストは小さくない．しかし，マルチコアシステムでは PIC ではなく I/O APIC を用いており，I/O APIC では同じアドレスを複数のレジスタで共有してないため，PIC と比べ状態の取得は容易である．故に，現在主流であるのマルチコアではこのことは問題とはなりにくい．

通常時に書き込み I/O を監視するとある程度オーバヘッドが生じてしまうものの，監視すべきレジスタは少ない．表 4.1 は PMM が監視するアドレス (I/O ポート) の一覧である．全てのアドレスはレガシーデバイス用の PIO ポートである．また，これらの PIO のポートには，別途読み出し専用レジスタも紐づけられている．レガシーデバイスのみが書き込み専用レジスタを割り当てる理由は，x86 アーキテクチャにおいて，I/O ポートは 64KB のアドレス空間しか存在せず，利用する I/O ポートの数を少なくする必要があったためと考えられる．結果として，少ない I/O ポートで多くのレジスタにアクセスできるようにするために，

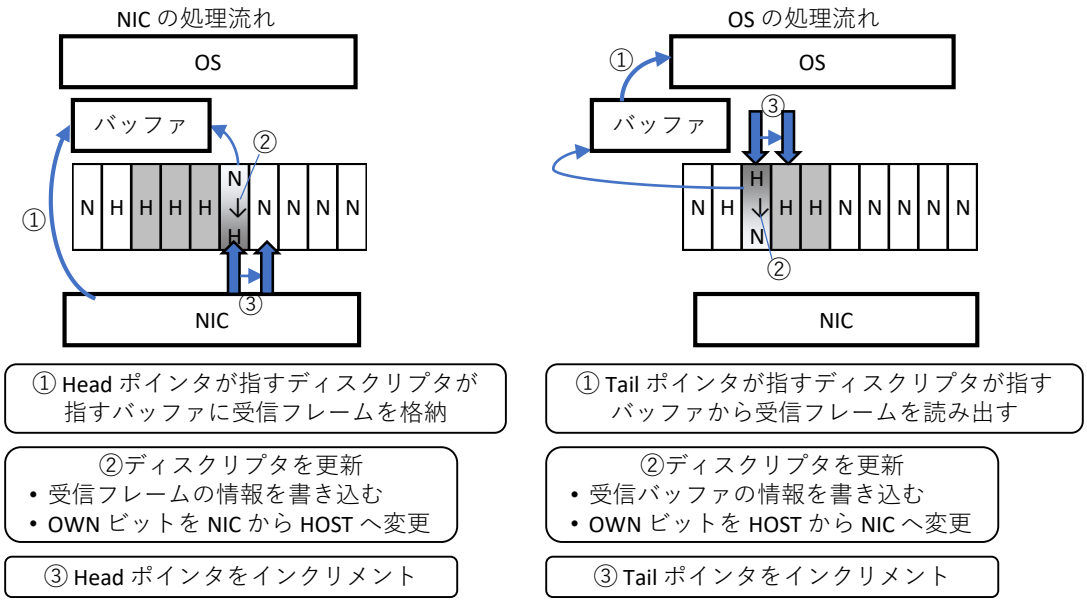

図 4.8　RTL8169 によるフレーム受信処理の流れ

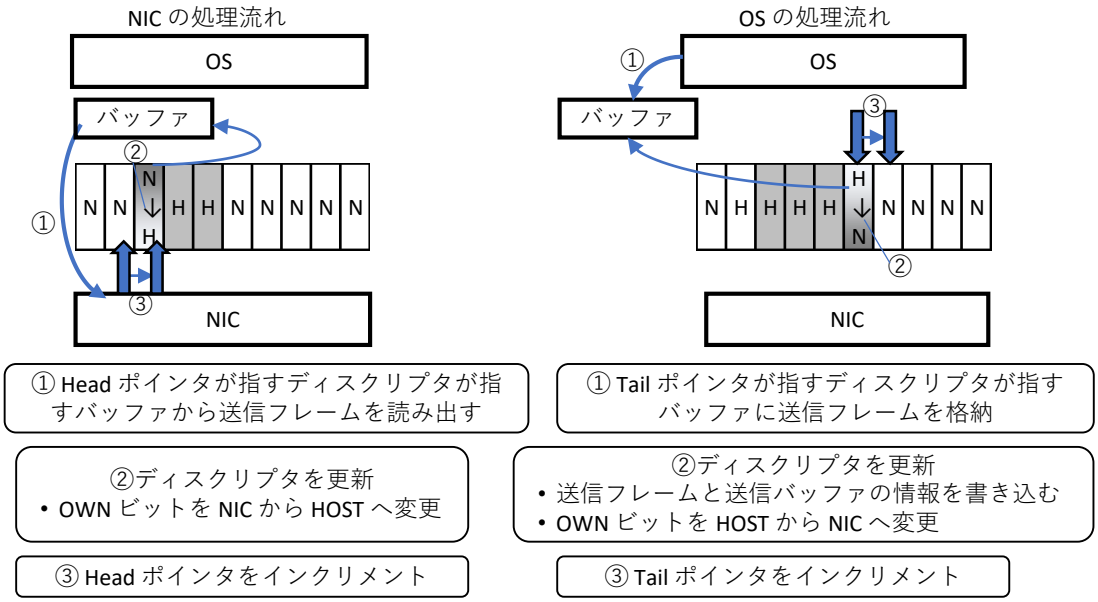

図 4.9　RTL8169 によるフレーム送信処理の流れ

これらのデバイスは 1 つの I/O ポートに別々の読み出し専用のレジスタと書き込み専用のレジスタを割り当てた．一方で，MMIO のアドレス空間は広大であり，そのために，読み出しと書き込みで別々のレジスタを一つにアドレス共有するような設計にはしていない．幸い，近年の OS は起動時にしかこれらのレガシーデバイスを用いないため，これらの I/O ポートへのアクセスはほとんどなく，これらへの介入のコストは無視できるほど小さい．

内部レジスタの状態の取得

内部レジスタの例には，Realtek 社製の RTL8169 シリーズの NIC が持つレジスタがある．このデバイスでは，内部レジスタはパケットの送受信に関連するものである．RTL8169

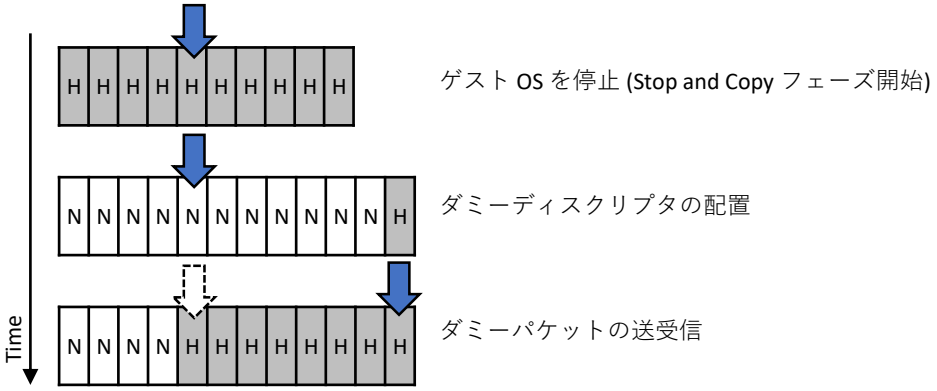

図 4.10　RTL8169 のリングバッファにおけるポインタの取得

において，NIC と OS はメモリ上にあるバッファを介してお互いにパケット渡しあう．ネットワークからパケットを受信したとき，まず NIC がメモリ上のバッファに受信したパケットを置く．その後，OS はそのバッファからパケットを読み出す．パケットをネットワークに送信する際は，この逆の手順を行う．このバッファはリングバッファで管理されるバッファのディスクリプタによって管理されている．NIC と OS がやり取りするためのリングバッファは 2 つある．一つはパケットの受信処理用，もう一つは送信処理用である．

NIC と OS はそれぞれリングバッファ上のポインタを管理しており，それぞれが次に操作するべきリングバッファ上のディスクリプタを指している．受信用リングバッファのヘッドポインタと送信用リングバッファのテールポインタは NIC 内部で管理されており，ソフトウェアからは直接アクセスできない．しかし，これらのポインタはマイグレーションの際に取得して転送しなければならない．なぜなら，ライブマイグレーション後にこのポインタの値が変わると，リングバッファの処理順序の整合性が取れなくなるためである．

これらの内部の状態を挙動の観察によって推定するには，まず RTL8169 によるパケット送受信の流れを知る必要がある．図 4.8 に RTL8169 におけるパケット受信処理の流れを示している．パケットの送受信において，OS と NIC は扱う Ethernet フレームについての情報が格納されるディスクリプタのリングバッファを介して非同期に協調して動作する．RTL8169 において，各ディスクリプタには `OWN` ビットと呼ばれるビットが存在する．このビットは当該ディスクリプタが次に NIC によって処理されるべきものか HOST (ここでは OS) によって処理されるべきものかを示している．このビットを介して，NIC と OS は互いにどのディスクリプタまで処理を終えているのかを把握しながら処理を進めていく．RTL8169 では，明示的にリングバッファの Head と Tail ポインタを格納するレジスタが存在せず，お互いの処理状況を把握する手段はこの `OWN` ビットのみである．しかし，NIC と OS はそれぞれ独自に Head ポインタと Tail ポインタを管理している．図左で示す NIC による処理は以下の流れになる．

1. Head ポインタが指すディスクリプタの OWN ビットを確認する．もし OWN ビットが HOST であれば，HOST の処理が進んでおらず，リングバッファが埋まっていることになるため，それを示す割り込みを起こす．
2. OWN ビットが NIC であれば，当該ディスクリプタが指すバッファに受信した Ethernet フレームを DMA でコピーする．
3. 当該ディスクリプタを更新する．具体的にはバッファに格納した Ethernet フレーム

に関する情報 (データ長など) をディスクリプタに書き込み， `OWN` ビットを NIC から HOST へ変更する．
4. 最後に，Head ポインタをインクリメントする．

一方，図右に示す一般的な OS による Ethernet フレームの受信処理は以下の流れになる．

1. Tail ポインタが指すディスクリプタの OWN ビットを確認する．もし OWN ビットが NIC であれば，NIC はパケットを受信しておらず，受信フレームは存在しないと判断する．
2. OWN ビットが HOST であれば，当該ディスクリプタが指すバッファから受信した Ethernet フレームをコピーする．
3. 当該ディスクリプタを更新する．具体的にはバッファに関する情報 (バッファ長など) をディスクリプタに書き込み， `OWN` ビットを HOST から NIC へ変更する．
4. 最後に，Tail ポインタをインクリメントする．

同様に，送信処理の流れを 図 4.9 に示す．NIC の処理流れは以下の通りである．

1. Tail ポインタが指すディスクリプタの OWN ビットを確認する．もし OWN ビットが HOST であれば，送信すべき Ethernet フレームが存在しないため，何もしない．
2. OWN ビットが NIC になっていれば，当該ディスクリプタが指すバッファから送信する Ethernet フレームを DMA で読み出す．
3. 当該ディスクリプタを更新する．主な更新は OWN ビットを NIC から HOST へ変更することである．
4. 最後に，Tail ポインタをインクリメントする．

OS の処理流れは以下の通りである．

1. Head ポインタが示すディスクリプタの OWN ビットを確認する．もし OWN ビットが NIC であれば，NIC の処理が追いついておらず空きディスクリプタがない状態であるといえる．
2. OWN ビットが HOST であれば，HOST はメモリ上に送信したい Ethernet フレームを置き，そのアドレスやデータ長を当該ディスクリプタに書き込む．
3. OWN ビットを HOST から NIC へ変更する．
4. 最後に Head ポインタをインクリメントする．

これらの内部状態を取得する方法の一つは，OS とデバイスの通信を継続的に監視する方法である．例えば，OS がこれらのリングバッファにアクセスするたびに PMM がそのアクセスに介在し，リングバッファがどのような状態になっているか観察するというものである．また，NIC によってもリングバッファは書き変わるため，OS の処理とは関係なく定期的な観察も必要となる．しかしながら，これらの通信量は非常に多く，それ故にこの監視処理は性能劣化を招くため避けるべきである．別の方法として，Stop-and-Copy フェーズにおいて PMM が能動的にデバイスを制御し，デバイスがそれに応じてどのように動作するかを監視する方法がある．このデバイスの観察は，破壊的な観察である．つまり，内部状態は PMM の観察により破壊されてしまう．しかしながら，この観察が破壊的であることは問題にはならない．なぜなら，ゲスト OS は，マイグレーション後に移動元マシンで動作する必要がないため，移動元マシンで状態を維持する必要がないためである．この破壊的な処理の前に PMM はディスクリプタやバッファ内のパケットを移動先のマシンに転送しているため，転

送先では破壊前の状態が復元できる．それ故に，送信前の送信パケットや受信済みの受信パケットは移動先でも保持される．タイミングの問題でいくつかの受信パケットはドロップしてしまうものの，イーサネットは元々規格上ベストエフォートでありパケットが必ず到着することを保証していない．また，stop-and-copy フェーズにおいてはマシン切り替えに伴うダウンタイムが発生し，この間に到達する受信パケットもドロップする．幸い，ドロップしたパケットは必要であれば上位のレイヤで再送される．ドロップしたパケットの数については，第 4.5.5 節 で測定した結果を示す．

RTL8169 NIC の内部状態を取得するために，移動元マシンの PMM はダミーパケットを送受信する．以下では，ダミーパケットの送受信を通して内部状態を取得する流れを示す．移動元の PMM は stop-and-copy フェーズに移行するにあたり，まずゲスト OS を停止する．その後，リングバッファのサイズを 2 つ増やし，最後のエントリーを除き全てダミーパケットを参照するディスクリプタへ置き換える．各ディスクリプタは EOR と呼ばれるリングバッファの終端を示すビットを持っており，リングバッファのサイズを増やすために，元々のリングバッファの終端の EOR ビットをクリアし，追加する二つ目のディスクリプタの EOR ビットをセットする．追加する二つのディスクリプタの役割は以下の通りである．一つ目のディスクリプタはダミーパケットの送信が完了したか否かを示すインジケーターとしての役割を果たす．二つ目のディスクリプタは送信処理を止め，ポインタのインクリメントを止めるターミネーターとしての役割を果たす．図 4.10 では，最後のエントリーを除く全てのエントリーが “N” となっている．これは，OWN ビットがセットされており，ディスクリプタの所有者が NIC となっていることを示す．その後，PMM は NIC に対してパケットの送受信処理の要求を出す．NIC はリングバッファのエントリーをシーケンシャルに処理していく．この際，NIC は元々内部のポインターが指していたエントリーから順に処理していく (図 4.10, 下)．この状態で NIC が送信を始めると，ポインタが指しているディスクリプタから，ポインタの値をインクリメントしながら，ターミネーターに到達するまでパケットを転送する．転送が完了したパケットのディスクリプタの OWN ビットは NIC からホストに変更される．一方，転送されなかったパケットのディスクリプタの OWN ビットはそのまま NIC のままである．PMM は NIC がダミーパケットの送信を完了したか否かを追加した 2 つのエントリーのうち 1 つ目のエントリーの OWN ビットを見ることで判定する．インジケーターのディスクリプタのパケットが転送されたのちに，物理マシンモニタはリングバッファ全体をチェックする．その後，PMM は処理されたリングバッファを走査し，OWN ビットが変わっている境界を見つけ出す．リングバッファの先頭から走査し，最初に OWN ビットがホストに変わっている部分が，元々ポインタが指していたディスクリプタであることがわかる．送受信処理が完了したことを追加のエントリーで判定することで，元のポインタがリングバッファの末端にあった場合でも容易に判定できる．受信処理用の内部ポインタについても同様の流れで取得できる．ダミーパケットの受信は，NIC ではなく PMM と協調できるリモートマシンに当該 NIC へのダミーパケットの送信を指示することで行う．

ダミーパケットの送受信処理には，通信相手を用意する必要がある．今回のプロトタイプ実装ではライブマイグレーションにおける移動先マシンの PMM を通信相手としている．しかし，設計上必ずしも移動先の PMM を用いる必要はなく，PMM と協調して動作できるマシンであればよい．その他の案としては，ライブマイグレーションを指示するためのコントローラノードや，移動元マシンに直結しているスイッチなどが考えられる．例えば，OpenFlow 対応のスイッチであれば，OpenFlow のプロトコルに従ってコントローラからスイッチに対して Packet-out メッセージを送ることで，スイッチが特定のポートにパケットを送ることができる．これを用いて，移動元マシンに対してダミーパケットを送信することができると考

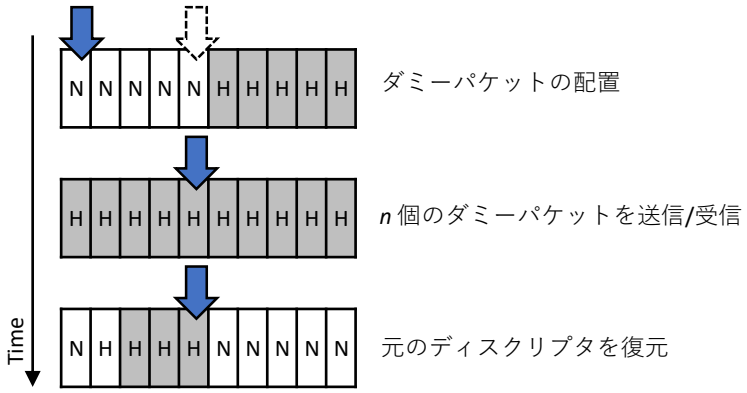

図 4.11　RTL8169 のリングバッファにおけるポインタの復元

えられる．

これまで，RTL 8169 NIC における状態の取得方法について述べてきたものの，全ての NIC でこのような処理が必要なわけではない．例えば，Intel Pro 1000 シリーズの NIC においては，上記のリングバッファのポインタは読み出し可能なレジスタに格納されているため， PMM は単にレジスタから値を読みだけでよい．

物理デバイスの状態の復元

ライブマイグレーションにおいて，移動先の PMM はデバイスの状態を再構築する必要がある．状態の再構築を考える上で，ここではデバイスの状態を書き込み可能な状態と書き込み不可能な状態の 2 つに分類する．書き込み可能な状態の再構築は PMM が単に値をレジスタに書き込むことで完了する．問題は書き込み不可能な状態である．書き込み不可能な状態は PMM がデバイスを注意深く制御することで再構築する．Realtek RTL8169 内部で保持している，送受信リングバッファのヘッドポインタやテールポインタは書き込み不可なレジスタに格納されている．これらの値を制御するために，移動先の PMM はダミーパケットを送受信する．

図 4.11 はダミーパケットの送受信による内部状態を制御する処理を示している．移動先マシンの PMM は，ダミーパケットのエントリーをポインタとして設定したい値と同じ数リングバッファに配置する．それぞれのエントリーは OWN ビットが NIC にセットされており，ダミーパケット用の送受信バッファが用意されている．図 4.11 では，内部のポインタに 5 を設定しようとしている例を示している．図の上部では，ポインタの値に合わせて，5 つのダミーパケット用エントリーを配置している．エントリーを配置した後に，PMM は NIC に対してパケットの送受信処理を要求する．NIC は要求に従ってパケットの送受信処理を行い，それに伴って，図の中段に示すように内部のポインタをインクリメントする．受信において，マルチキャストパケットやブロードキャストパケットなど予期しないパケットを受信すると，ポインタが必要以上に進んでしまう．そこで，この処理の間はマルチキャストパケットやブロードキャストパケットを受信しないように設定し，NIC は自身の MAC アドレスにマッチしたユニキャストのパケットのみを受信するように設定している．内部ポインタを設定した後に， PMM は元の送受信リングバッファに元のディスクリプタとバッファを復元する (図 4.11 下段)．

デバイスの状態の再構築にあたり，再構築する状態の順に注意する必要がある．なぜなら，

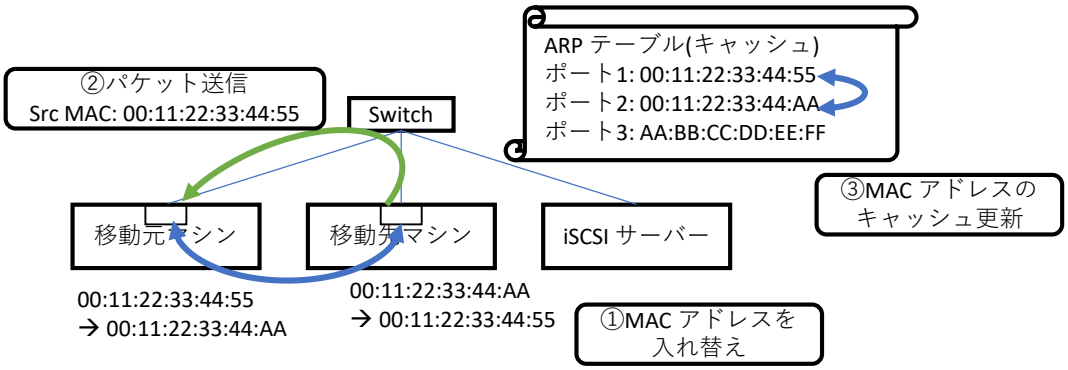

図 4.12　MAC アドレスの入れ替え

設定状態の設定は処理状態に影響を与えることが多いためである．例えば，NIC の設定状態を変更すると，それに関連する処理状態がリセットされる場合がある．一方，処理状態を変更しても設定状態に影響を与えない．それ故に，PMM は基本的に設定状態を再構築した後に，処理状態を再構築する．これは，デバイスの初期化と同様の順番でもある．

また，タイマデバイスの状態の復元については以下のとおりである．OS が割り込み頻度を指定し，その頻度で割り込みを発生させるタイマーデバイスについては，移動元で設定された値と同じ値を移動先マシンのタイマデバイスに設定する．Programmable Interval Timer (PIT) をシングルコアで動作させた場合はこのような動作をする．一方，OS が次回の割り込みタイミングを毎回指定し，指定されたタイミングで一度だけ割り込みを発生させる (ワンショット) 動作をするタイマデバイスについては，移動先マシンでは，次回割り込みまでの時間として指定可能な最小値を入れる．これは，Stop and Copy の時間は一般的にタイマー割り込みの間隔より長いため，移動先では直ちに割り込みを起こすためである．

4.4.5　ネットワーク接続の維持

マイグレーションの前後で，OS から透過的にネットワーク接続を維持するためには，NIC の MAC アドレスが変更されずに維持される必要がある．そこで，MAC アドレスも設定状態として扱い，移動元から移動先に転送し，移動先マシンの NIC の MAC アドレスを変更している．しかし，移動先の MAC アドレスを単に移動元の MAC アドレスに変更すると，移動元の MAC アドレスと衝突し，通信に支障が出る場合がある．そこで，マイグレーション時に移動元と移動先の MAC アドレスを入れ替えている．NIC の中には MAC アドレスを永続的に変更する必要があるものもある．このような NIC に対応するために，データセンターのコントローラはマイグレーションの際，移動元と移動先の MAC アドレスを記録しておき，NIC の変更を追跡しておく必要がある．

移動元と移動先の間にあるネットワークスイッチなどのネットワーク機器にも MAC アドレスの変更を知らせる必要がある．高機能なネットワーク機器であれば，データセンターのコントロール部分からネットワーク機器に対して明示的に MAC アドレステーブルの更新を指示できるものもある．しかしながら，そうではない物が含まれている場合，実際に MAC アドレスが変更されたことを知らせるために Ethernet フレームのソースアドレスが変更後の MAC アドレスとなったパケットを送る必要がある．このために，マイグレーション後，移動先マシンの PMM はゲスト OS に割り当てる NIC から移動元マシンのゲスト用 NIC に対して Reverse ARP を転送する．

表 4.2　ペイロードのフォーマット (メモリ)

オフセット (バイト単位)	0	1-9	9-1033
内容	状態種別 (メモリ)	アドレス	データ

表 4.3　ペイロードのフォーマット (その他状態)

オフセット (バイト単位)	0	1-n
内容	状態種別	データ

4.4.6　状態の転送処理

移動元の PMM と移動先の PMM はネットワークを介して状態のやり取りを行う．今回実装したプロトタイプでは，状態の転送は，UDP パケットで行う．TCP パケットではない理由は，実装の簡略化のためである．本プロトタイプ実装時，基となった BitVisor では TCP によって安定したネットワーク性能を実現することが難しかった．実際に運用する際には，TCP で実装すべきであるが，本研究の主眼は通信プロトコルの実装ではないため，今回は実装していない．

状態転送のペイロードのフォーマットは表 4.2 と表 4.3 の通りである．最初の 1 バイト目は状態の種別を表すためのコード番号が入っている．受信側である移動先マシンの PMM はこの番号を見ることで，届いたパケットがどのハードウェア状態に関するデータかを判断する．1 バイト目以降は状態種別によって異なる．メモリ上のデータであれば，次の 8 バイトはデータがあった物理アドレスを示す．その後 1KB (9 バイト目から 1033 バイト目まで) は先に指定された物理アドレス上にあったデータとなる．その他の状態に関しては，種別ごとにデータのサイズやフォーマットは固定のため，その他のメタ情報はなく直接データが配置されている．メモリは 1KB 単位ごとに転送しているのは，ヘッダーを含めて MTU である 1500 バイトを超えない範囲で管理がしやすい単位であったためである． 移動先の PMM はパケット受信ハンドラ内で，先頭の状態種別を示すコードを読み取り，それに沿った処理を行う．メモリ以外の状態は，一度バッファリングした後，Stop-and-Copy フェーズでまとめて復元する．そうしなければ，受信処理に遅れが生じてしまうためである．移動元マシンが転送終了のパケットを送信し，そのパケットを移動先マシンが受信したら，移動先マシンは受信したデータを基に状態の復元を開始する．

4.4.7　実装状況

提案手法を評価するため，本システムのプロトタイプを実装した．このプロトタイプはマルチコア CPU, 割り込みコントローラー (PIC および APIC), タイマーデバイス (PIT), および Realtek RTL8169 NIC に対応している．本実装は， Intel 製 CPU にのみ対応しているものの，AMD 製の CPU に対しても同様の実装が可能であると考えられる．これらのデバイスへの対応に際してデバイスの仕様を調査した限り，ストレージデバイスや高性能 NIC への対応も容易に可能であると考えられる．

RTL8169 NIC のマイグレーション処理を実現するソースコードは 1,176 行であった．このコードには前章で述べた読み出し不可状態の読み出しや書き込み不可の状態の復元処理も含まれている．このコード行数は一般的な OS のデバイスドラ

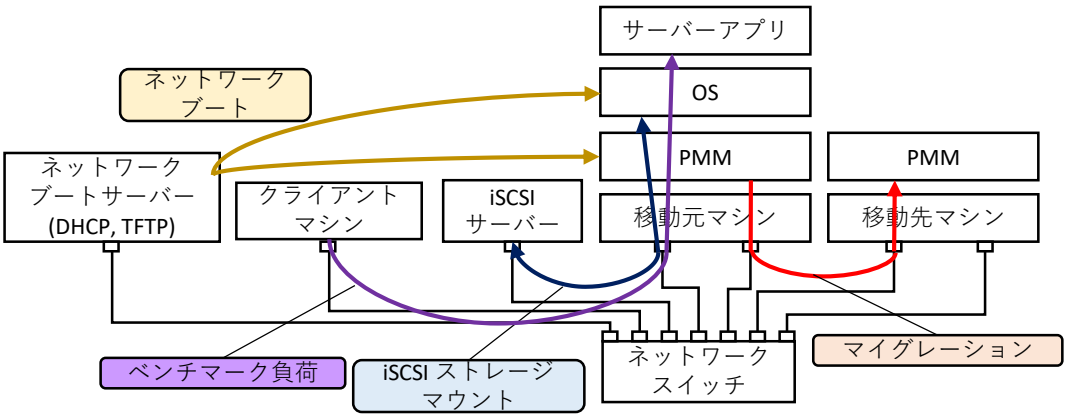

図 4.13　実験環境

イバと比べて非常に少ない．例えば，Linux 4.9 の RTL8169 向けデバイスドライバ (`drivers/net/ethernet/realtek/r8169.c`) は 6,800 行以上である．それ故に，提案手法のデバイス依存の実装は比較的容易に可能であると言える．

現在の実装では，ストレージデバイスはネットワークを介して iSCSI 接続することを想定している．それ故にストレージデータの内容はコピーすることなく移動元と移動先のマシンで引き継ぐことができる．ライブマイグレーションの前後で iSCSI サーバーのアドレスは替わることがなく，またネットワーク接続が継続しているため，ゲスト OS はストレージデバイスに継続的にアクセスすることができる．これはライブマイグレーションシステムにおいて長時間のダウンタイムを避けるために一般的な構成であるといえる．しかしながら，本手法はローカルの HDD や SSD を伴うライブマイグレーションも対応可能だと考えている．ホストコントローラーのデバイス状態はこれまで説明してきた手法で実現可能であると考えられる．また，保存されているデータについては，バックグラウンドでコピーする技術が既に確立しているため，これを組み合わせることで実現できると考えられる [35].

その他デバイス状態の転送については以下の通りである．マシンの時間に関する情報を維持するために，Time Stamp Count (TSC) は Stop and Copy フェーズに移行した直後に移動元マシンで値を読み，この値を移動先のマシンに書き込む．APIC の状態転送は転送対象の状態がすべての読み書き可能であったため，単純な読み書き操作のみで転送している．RTL 8169 では NIC のレジスタの値以外に MAC アドレスの変更，PCIe の MSI-X の状態の取得および復元，Phys デバイスの状態の転送を行う．また，デバッグのために，移動先マシンでシリアルポートの初期化を行っている．

4.5　評価

4.5.1　セットアップ

本稿ではライブマイグレーションを行わない通常処理時の性能と，ライブマイグレーション中の性能についての評価結果を示す．通常処理時の性能は，物理マシン，提案手法，及び KVM と比較した．KVM は PCI-passthrough されたネットワークデバイスを用いたものと，準仮想化デバイスである virtio-net を用いたものと比較した．

実験環境では，OS はディスクレスサーバーとして動作する．これは，今回実装したプロト

表 4.4　マイグレーション対象のサーバー構成

CPU	Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz
マザーボード	ASRock Z97 Extreme6
RAM	4GB
NIC (PMM, VMM 占有)	Intel I218V
NIC (ゲスト OS が利用)	Realtek RTL8111/8168B
ストレージ	なし (iSCSI ストレージを使用)

表 4.5　クライアントマシンの構成

CPU	AMD Phenom(tm) II X6 1090T Processor
RAM	8GB
NIC	Broadcom BCM57788, Intel 82541PI

表 4.6　評価対象システムが動作するマシンの構成

ゲスト OS (Linux)	Debian 7.8 , Linux カーネル 3.4.0 ,
ゲスト OS (Windows)	Windows Server 2016 Data center Edition
ホスト OS	Debian 7.8 , Linux カーネル 3.4.0
VMM	QEMU 1.1.2 (KVM 有効)

タイプがストレージデバイスに対応していないこと，ストレージ内のデータ転送には時間を要するという理由からである．ディスクレスの状態であれば，ストレージ内のデータはマイグレーションする必要はなく，ネットワーク接続を維持できてさえいればストレージへのアクセスを続けられる．本実験環境でのマシン構成は 図 4.13 の通りである．サーバーマシン上で OS と PMM をディスクレス構成で起動するために，以下のような起動処理を行う．まず，サーバーマシンが起動すると，iPXE によってネットワーク経由で PMM のイメージのダウンロードと iSCSI の接続を行う．このために，同一ネットワーク上に iPXE と PMM のイメージを配布するための DHCP サーバーと TFTP サーバーを設置している．その後，PMM を起動し，iSCSI の MBR を読み出し，ブートローダーを読みだし，これを起動する．さらに，ブートローダーは OS を起動する．この際も，ここまで接続してきた iSCSI の設定を引き継ぐために，おなじ iSCSI の設定をしている．

ライブマイグレーションの移動元マシンと移動先マシンとして，同一構成のマシンを 2 台用意した．構成は 4.4 の通りである．また，iSCSI サーバーとクライアントマシンの構成は 4.5 の通りである．iSCSI サーバーのストレージデバイスは，SSD Crucial CT512MX である．

ネットワークを介するベンチマークにおいては，サーバーマシンの RTL NIC を使用する．PMM が動作している際，Pro 1000 NIC は PMM によって隠蔽される．また，物理マシン動作時においては OS が Pro 1000 NIC を認識するものの，利用しないように設定する．また，KVM 動作時においては，後述する virtio-net を用いる構成においても物理的には RTL NIC のみをベンチマークに用いるようにする．

評価環境において，サーバーマシンの CPU の設定は以下のように行った．まず，CPU のスリープによる性能への影響を無くすために，CPU の C-state を無効にした．また CPU クロックの動的な変動による性能への影響を無くすために，CPU クロックが動的に変化しない

ように設定した．さらに，CPU 内部のスレッドスケジューリングの影響を排除するために，hyper-threading は無効としている．これらの設定をしないと，物理マシン環境と仮想環境である KVM との CPU 制御ポリシーの違いが性能測定の結果に反映されてしまう．具体的には，物理マシン環境では省電力化のためにスリープ状態に入ったりクロックを動的に下げたりする傾向が強く，これにより性能が低下する傾向がある．上記のような設定は物理マシン環境で最大性能を引き出すための設定として一般的なものであり，本実験向けの特別な設定ではない．また，これらの設定で，KVM 環境の性能が著しく低下することはない．

また，ネットワークデバイスからの割り込みはすべて同一のコアに通知されるように設定している．デフォルトの設定では，割り込みの通知先はシステムによって異なっている．具体的には，物理マシンとプロトタイプシステムでは，割り込みが 1 つのコアに集中し，KVM はすべてのコアに割り込みが分散する．Intel アーキテクチャの仕様上，物理マシン環境であっても割り込みか全てのコアに分散する設定ができるべきであるが，実験に用いたマシンではこの機能が動作しなかった．KVM は仮想的な割り込みコントローラによってこの仕様の動作を実現している．割り込みの分散の有無は I/O 性能に大きな影響を与えるため，今回の性能評価では割り込み分散の有無による影響を排除するために，KVM においても割り込みを一つのコアに集中するように変更した．

比較対象の KVM のセットアップについて以下で説明する．KVM では仮想 NIC を用いたセットアップと NIC をパススルーしたセットアップを用意し，それぞれ性能を測定し比較した．以後，仮想 NIC を用いた KVM のセットアップを KVM (virt), PCI パススルーを用いたセットアップを KVM (pass) と表記する．KVM では，仮想 CPU と物理 CPU は 1 対 1 対応するように設定している．つまり，仮想 CPU のマイグレーションによる性能の揺れは生じない．また，KVM はローカルのストレージから起動する．このため，KVM ホストによる iSCSI ストレージへの負荷は生じない．一方，VM については物理マシンや PMM 環境と同様に iSCSI に接続したディスクレスサーバー構成とした．

KVM において，メモリは可能な限り VM に提供する．また，KVM ホストは swap を無効にしている．このため，KVM ホストでスワッピングが発生し，性能劣化することはない．

また，KVM のデフォルトのクロックソースである kvm-clock を用いないようにした．これは，kvm-clock から時刻を読み取る処理が非常に遅く，計時処理が性能劣化につながるためである．この影響は，Sysbench による一部のベンチマークで無視できないほど大きかった．本実験では，kvm-clock に変わり，Time stamp counter (TSC) を用いた．この変更は，ゲスト OS の Linux の `/sys/devices/system/clocksource/clocksource0/current_clocksource` を変更することで実現している．

性能評価で用いた OS と VMM 環境を表 4.6 以下に示す．ゲスト OS は，Linux と Windows ともにカーネルは未変更であり，独自のカーネルモジュールなどは追加していない．

4.5.2 システムベンチマークでの性能評価

システムの性能として，CPU, メモリ，ファイル I/O の性能を測定した．測定では，ベンチマークソフトである Sysbench[44] のバージョン 1.0 を用いた．利用した OS である Debian のデフォルトパッケージに含まれる sysbench 0.4 は，memory ベンチマークに不具合があり，正しい性能を測ることができなかったため，このバージョンを用いた．Sysbench はマルチスレッドで動作するベンチマークソフトであり，そのスレッド数を指定することができる．

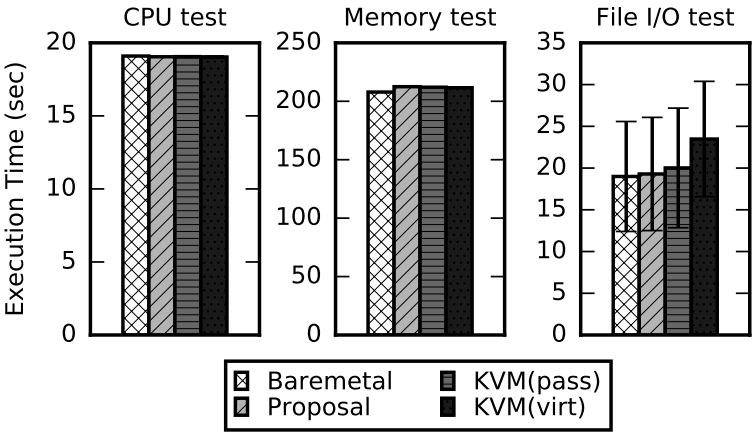

図 4.14　システムベンチマーク

今回の測定では，全ての測定において物理コア数と同じ 4 スレッドを指定した．

これらの測定は各 10 回行った．以降で示す測定結果は，10 回の測定の平均値と標準偏差を示している．図 4.14 の左のグラフは，Sysbench の CPU テストの実行時間を示している．このテストでは，素数を見つけ出す計算を行っている．結果から，全てのシステムにおいて実行時間に差は見られなかった．提案手法と KVM においては Intel VT-x を用いているため，数値計算のために必要な命令はほぼすべて CPU によって直接実行できる．故に，CPU だけを用いる計算では，PMM と KVM は共に VM exit を発生させないため，システム間での性能差が小さかったと考えられる．

図 4.14 の真ん中のグラフは，Sysbench のメモリテストの実行時間を示している．このテストでは，16 MB の空間にランダムにメモリ書き込みを行う．メモリ書き込みの総量は 100 GB である．結果，“Proposal”, “KVM (pass)”, “KVM (virt)” ではそれぞれ 2.21%, 2.01%, 及び 1.78% 実行時間が増加した．このオーバヘッドの理由は，EPT による 2 段階のアドレス変換によるものと考えられる．KVM では，これらのアドレス変換は VMM と複数の VM が単一の物理マシンに共存するために必須である．一方，PMM では，ストレートマッピングを用いるためアドレス変換は必ずしも必要ではない．PMM は Pre-copy フェーズにおける dirty page の追跡と PMM のメモリ領域の保護にのみ EPT を必要とする．このため，このメモリアクセスにおけるオーバヘッドもさらに緩和することができると考えられる．このことについては，第 5.6 節 で議論する．

図 4.14 の右図は，Sysbench のファイル I/O テストの結果を示している．このテストは，ランダムなファイルに対してのランダム I/O を行う．今回の性能測定ではファイルの数は 128 個，ファイルのサイズは 16MB とした．また，一回のアクセスでアクセスするサイズは 16 KB であり，アクセスの回数は 200,000 回，合計で約 3GB のアクセスを行う．また，読み出しと書き込みの比率は 3:2 であり, `fsync ()` システムコールは 100 回のアクセスの度に呼び出す．提案手法の実行時間は，1.6% のみ増加した．一方，KVM (pass) は 5.4%, KVM (virt) は 23.6% 実行時間が増加した．今回の実験環境では，ストレージは iSCSI によってネットワーク越しに接続されておりファイル I/O はネットワーク経由となるため，このベンチマークもネットワーク性能の影響を受ける．このため，オーバヘッドの違いは，ネットワークのレイテンシの差によるものと考えられる．iSCSI プロトコルでは，コマンド送信やレスポンスの送信のために小さなパケットのやり取りが多くなるため，ネットワークレイテンシの性能差が比較的大きく反映されると考えられる．

表 4.7　ゲスト OS が利用可能なメモリ容量

システム	メモリ容量 (KiB)
Baremetal	3,747,616
Proposal	3,612,640
KVM	3,433,204

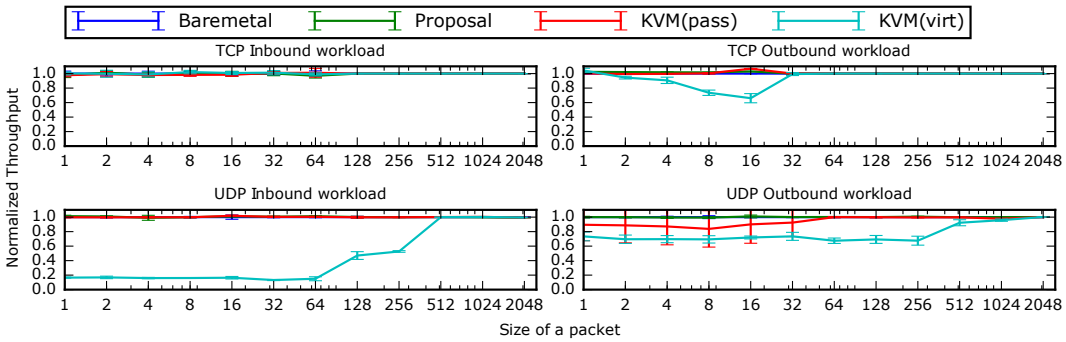

図 4.15　ネットワークスループットの比較

メモリ使用量

ゲスト OS が利用可能なメモリ容量は，メモリインテンシブな負荷では重要となる．しかしながら，仮想化レイヤも動作のためにある程度メモリを必要とする．この仮想化レイヤのメモリ消費を評価するため，ゲスト OS が利用可能であるメモリ容量を測定した．KVM では，メモリのオーバーコミットを避けるために，ホスト OS の swap を無効にし，VM に可能な限りのメモリを提供するように設定した．表 4.7 では，各システムにおいてゲスト OS が利用可能なメモリ容量を示している．このメモリ容量は，ゲスト OS 上で `free` コマンドを実行することで測定した．KVM (pass) は KVM(virt) と同じメモリ設定であり，表中では “KVM” と表記している．Baremetal と比較して利用可能なメモリ使用量は，提案手法では 131 MB，KVM では 307 MB 減少していた．この結果から，PMM は必要なメモリ容量が VMM に比べて少なく，ゲスト OS に対してより多くのメモリを割り当てることができることがわかる．これは，PMM のシンプルなアーキテクチャが寄与していると考えられる．

ネットワーク性能

ネットワークの性能評価として，通常動作時のネットワークスループットとネットワークレイテンシを測定した．測定には，ネットワークベンチマークソフトである Netperf [45] を用いた．全ての測定において，サーバープログラム (`netserver` コマンド) は，評価対象のシステム上で動作し，Netperf クライアントプログラム (`netperf` コマンド) は，クライアントマシン上で動作する．スループットの測定では，TCP と UDP についてそれぞれ 1 バイトから 2048 バイトのペイロードを持つパケットにおける，送信スループットと受信スループットを測定した．また，レイテンシの測定では 1 バイトのペイロードを持つパケットのラウンドトリップタイムを測定した．

図 4.15 は，ネットワークスループットの測定結果を示している．各システム間での差を見

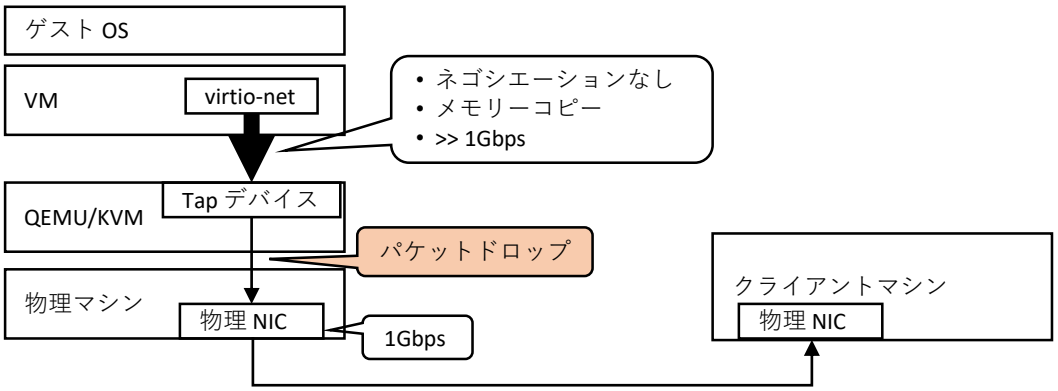

図 4.16　virtio-net を用いた環境での送信 UDP パケットのドロップ

るために，グラフでは物理マシンのスループットで正規化したスループットを示している．TCP の受信スループットにおいて，システム間で大きな差が見られなかった．これは，TCP スタックが Nagle アルゴリズムに従って複数の小さなパケットを集約して一つのパケットにして送信しているためであると考えられる．TCP の送信スループットにおいて，“KVM (virt)” のオーバーヘッドはパケットサイズを 1 から 16 に増加するに従って増加した．パケットサイズが 1 バイトの場合，送信マシン側で既に複数のパケットが一つのパケットに集約されており，このために実際に受信するパケット数は少なくなる．一方，パケットサイズが大きくなると，徐々に受信パケット数が増加する．パケットの数が増加するにしたがって，ACK パケットのための割り込み処理とパケットの処理が増大する．I/O と割り込みに介在し，エミュレート処理を行う “KVM (virt)” において，これらの処理の増加はオーバヘッドの顕在化につながる．しかし，32 バイト以上のパケットにおいては，ウィンドウサイズが十分に大きくなり，デバイスのラインレートである 1Gbps 程度のスループットとなっている．このため，仮想化によるオーバーヘッドは顕在化しなかった．一方，“Proposal” や “KVM (pass)” ではパケットサイズに依らずこのようなオーバヘッドは確認されなかった．

小さいパケットサイズにおける UDP のスループットではシステム間で差が見られる．特に，“KVM (virt)” において，受信スループットは物理マシンの 17 % 以下であり，送信スループットにおいては約 40 % のオーバヘッドがあった．これらのオーバヘッドは仮想化によるオーバヘッドである．“KVM (pass)” においても小さな UDP パケットの送信において 10 % のオーバヘッドが生じている．これは，割り込みエミュレーションによるものと考えられる．一方 UDP の受信スループットでは，“KVM (pass)” の性能は物理マシンと同等であった．また，“Proposal” は送受信，パケットサイズにかかわらず物理マシンと同等の性能が得られた．小さい UDP 送信パケットのスループットにおいて，“KVM (pass)” の標準偏差が非常に大きくなっている．これは iSCSI のハートビートパケットが通らなくなり，その結果 iSCSI クライアントのデーモンがネットワークインターフェイスをリセットし接続が一時的に切れたためである．この現象は，大量のパケットを送信した結果，割り込みエミュレーション処理が多発したために発生したと考えらえる．物理マシンや提案手法では，このような現象は発生しなかった．

KVM の virtio-net を用いた際，UDP の送信ワークロードにおいて，一部のパケットはドロップした．これについて図 4.16 に示し，以下で説明する．これは，本実験で利用した QEMU/KVM 環境において，virtio-net のインターフェイスとタップデバイスの間でネゴシエーションが機能せず，この結果，物理 NIC のリンクスピードを考慮しない速度でパケット

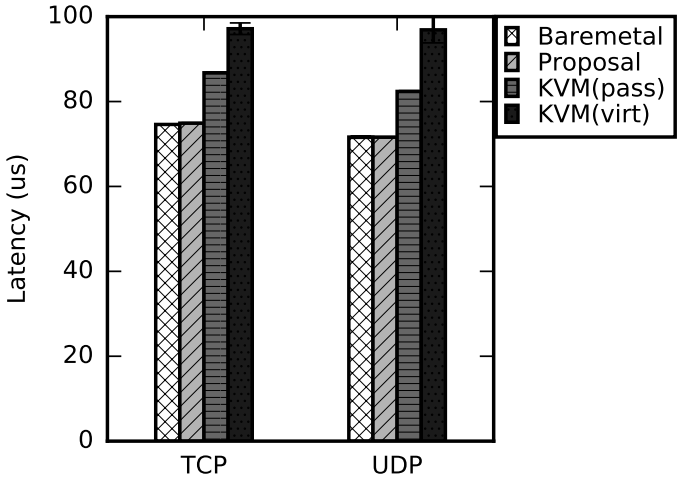

図 4.17　ネットワークレイテンシの比較

が転送されたためである．また，これらのインターフェイス間のパケット転送は，単にメモリコピーのみである．この結果，これらのインターフェイス間のスループットは物理 NIC が対応するスループットに比べて非常に高くなる．それ故に，パケットの一部はタップデバイスと物理デバイスの間でドロップしてしまう．しかしながら，Netperf が報告する送信スループットは virtio-net インターフェイスと tap デバイス間のスループットである．このため，グラフでは，Netperf が報告する送信スループットからタップデバイスと物理 NIC の間でドロップしたパケット分を差し引いて算出したスループットを示している．

図 4.17 は各システムにおける TCP と UDP のレイテンシを測定した結果である．TCP と UDP のそれぞれにおいて，提案手法のオーバーヘッドは無視できるほど小さかった (TCP: 0.4%; UDP: -0.04%) ．一方，“KVM (pass)” では約 15% (TCP: 16.3%; UDP: 15.0%), “KVM (virt)” では 30 % 以上 (TCP: 30.2%; UDP: 35.7%) の性能劣化を確認した．“KVM (virt)” でのオーバーヘッドはデバイス仮想化に関わる処理によるものと考えられる．仮想化されたデバイスでは，図 3.2 の通り，ゲスト OS とホスト OS の両方のデバイスドライバで処理され，またその間では virtio インターフェイスへの転送処理が必要である．PCI パススルーデバイスを用いることで，この性能劣化をある程度減じることはできるものの，完全に取り除けてはいない．これは，パススルーデバイスにおいても図 3.3 で述べた割り込みへの介入処理があるためと考えらえる．

4.5.3　VM exit の発生回数

仮想化レイヤによる性能劣化を分析するために，VM exit の数を測定した．この測定では，先述の Netperf による性能評価における，レイテンシー測定のワークロードを用いた．この測定のために，提案手法のプロトタイプシステムは VM exit の数を記録するためのカウンターの実装を追加している．また，このカウンターは，ゲスト OS から VMCALL 命令を発行することで読みだす．KVM の VM exit 数は, `kvm_stat` コマンドを用いて測定した．提案手法， KVM 共に，全てのコアでの VM exit の数を測定し，これを合算したものをシステムによる VM exit 数としている．

表 4.8 に提案手法，KVM (pass), 及び KVM (virt) により 1 秒間に発生した VM exit の数を示している．表に示す値は 10 回の測定の平均値である．KVM における VM exit の多

表 4.8　1 秒間の平均 VM exit 回数

Exit Reason	PMM	KVM (pass)	KVM (virt)
PAUSE	-	1613955.1	1594912.9
APIC access	-	62505.3	34355.9
IO instruction	-	-	9860.3
External interrupt	-	16933.9	7437.4
Interrupt window	-	5889.1	3.7
EPT violation	17.4	126.4	-
Exception or NMI	8.1	4.7	4.0
Control-register accesses	-	0.2	0.2
Total	25.5	1699459.1	1646606.0
Total (excluding PAUSE)	25.5	57502.7	51693.1

くはゲスト OS がアイドル状態になることによるもの (表中 “PAUSE”) である．この VM exit はゲスト OS がアイドル時のものであるため，性能への影響は問題とならないと考えられる．KVM (pass) により発生する他の VM exit は “APIC access,” “External interrupt,” “Interrupt window” など，割り込み処理に関連するものである．KVM (virt) で発生する “IO instruction” は，ゲスト OS が virtio デバイスに対し PIO で I/O アクセスした際に，KVM が介入し処理するために発生する．これらの結果は，図 3.2 や 図 3.3 に示す I/O や割り込みの処理が性能に大きな影響を与えていることを示している．一方提案手法では，秒間 26 回以下と VM exit の回数を非常に少なく抑えられている．このことから，提案手法では物理マシンと遜色ない性能を実現できることが確認できる．

VM exit の回数に加えて，提案手法における VM exit 時の処理における消費サイクル数を測定した．この測定では，VM exit が完了し，VM entry を行う直前までの消費サイクル数を計測した．測定の結果，EPT violation では 1 秒間で 平均 12,790.5 サイクル，Exception on NMI では 1 秒間で 6,981.1 サイクル消費していた．合計で，秒間 19.771.6 サイクル (1.24 マイクロ秒 / コア) 消費していることがわかった．このことから，提案手法における VM exit によるオーバヘッドは無視できるほど小さいといえる．

4.5.4　実アプリケーションでの性能評価

実際に利用されているサーバーアプリケーションの性能への影響を評価するため，Key-Value Store データベースである Redis[46] と Relational Database サーバーである MySQL[47] の性能を測定した．Redis の性能は YCSB[48] を用い，MySQL の性能測定では Sysbench の OLTP テストを用いた．ベンチマーククライアントである YCSB と Sysbench はクライアントマシンで実行し，Redis と MySQL は評価対象のマシンで実行し，ネットワークを介してサーバーへ負荷を掛けた．Redis については Linux における性能を，MySQL については Windows と Linux の両方で性能を測定した．

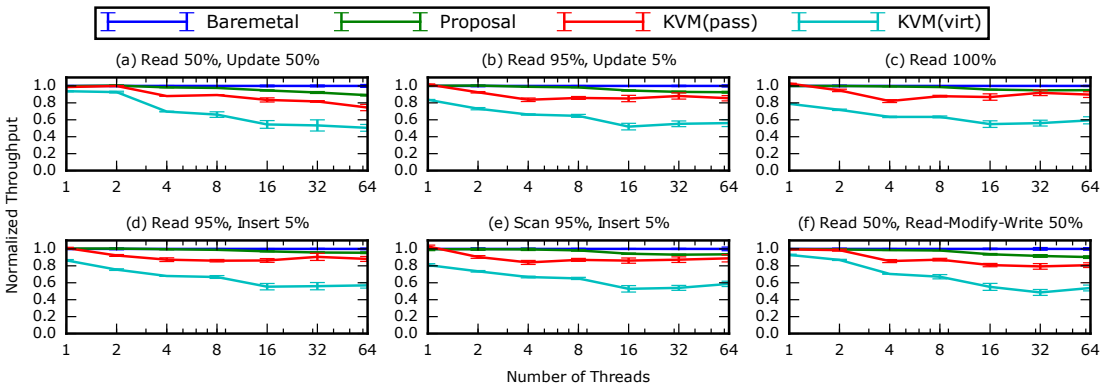

図 4.18　Redis のスループット

Redis

図 4.18 では Redis のスループットの測定結果を示す．ワークロードはそれぞれ以下のとおりである．

(a) アップロードが多いワークロード (read:upload = 50:50),
(b) 読み出しが多いワークロード (read:upload = 95:5),
(c) 読み出しのみのワークロード (read:upload = 100:0),
(d) 直近にアップロードされたデータを頻繁に読み出すワークロード (read:insert = 95:5),
(e) short ranges workload (scan:insert = 95:5),
(f) read-modify-write が多いワークロード (read ops:read-modify-write ops = 50:50).

ワークロード (d) 以外では，クライアントは読み出し対象や挿入対象のレコードは Zipf 分布に従って決定する．ワークロード (d) では，直近にアップロードされたレコードに対しての読み出しが多くなるように選ばれている．

この実験では，Redis はストレージ上にデータを格納しないように設定し，オンメモリキャッシュサーバーのように動作するように設定した．このため，この実験では，iSCSI ストレージを介したデータの読み書きは行われない．実験用のデータベースには 100 万レコード格納されており，ベンチマーククライアントは上記のワークロードをそれぞれ 10 秒間実行した．これらのワークロードのスループットは，クライアントのスレッド数を 1 スレッドから 64 スレッドまで変化させて測定した．このグラフはそれぞれの設定における 20 回の測定の平均と標準偏差を示している．グラフはスループットを示しており，高い値ほど性能が良いことを示している．また，グラフで示しているスループットは物理マシンのスループットを 1 として正規化したものである．

全てのワークロードにおいて，スレッドが増加するにしたがって，システム間の性能差が大きくなる傾向がみられた．64 スレッドにおいて，提案手法は 4.6 % から 11.0 % のオーバヘッドがあった．一方，KVM (pass) では，10.3 % から 25.5% のオーバヘッド，KVM (virt) では 41.0% から 49.5% のオーバヘッドがあった．全てのシステムにおいて，ワーストケースは 64 スレッドで (a) の更新が多いワークロードを実行したときであった．このワークロードはメモリアクセスが多発し，これに伴い TLB ミスも多発する．提案手法や KVM は EPT を用いており，TLB ミス時には EPT を用いた 2 段のアドレス変換が必要であるため，物理マシンと比べて重たい処理となる．このため，TLB ミスが多発するワークロードでは物

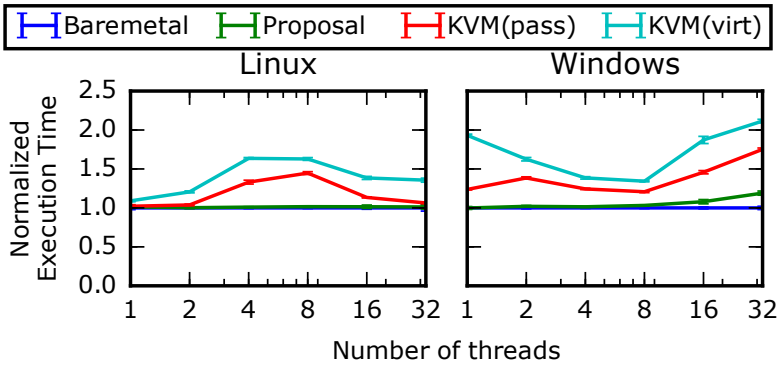

図 4.19　MySQL ベンチマークの実行時間

理マシンとの性能差が顕著になる傾向がある．また，KVM が提案手法よりも性能劣化が大きい理由は，I/O と割り込みのエミュレーションによるものであると考えられる．

MySQL

図 4.19 は Sysbench の OLTP テストの実行時間を示している．この実験において，実験用データベースは 1 万行のレコードを持ち，また 1 万回のトランザクションを実行した．各トランザクションは，14 回の read クエリ，(`SELECT`), 4 回の write クエリ (`INSERT`, `UPDATE`, `DELETE`) からなる．この実験では，クライアント数のスレッドを 1 から 32 へと変化させて実行時間は測定した．このグラフでは，各設定における 10 回の測定の平均実行時間と標準偏差を示している．横軸はクライアントのスレッド数，縦軸は実行時間を示している．本実験では，実行時間が短いほど，良い結果であるといえる．

Linux において (図左)，“Proposal” における実行時間は，ワーストケースで約 1.6 % 程度増加した．一方，“KVM (pass)” では 2.1% から 44.7%, “KVM (virt)” では最小で 9.0% (クライアント 1 スレッド時), 最大で 63.6% (クライアント 4 スレッド時) の実行時間の増加を確認した．Windows において (図右)，"Proposal” は 16 スレッド以下の場合には 8% 以下の実行時間の増加であった．また，32 スレッドでは 19.0% の実行時間増加をであった．一方，"KVM (pass)" は 20.7–75.2% ，"KVM (virt)" は 34.3–111.5% 実行時間が増加した．このワークロードはトランザクションごとにストレージにトランザクションの結果を反映するため，I/O インテンシブなワークロードであると言える．さらに，今回の実験環境では，ストレージは iSCSI を用いてネットワーク経由で接続されている．この結果，このワークロードではネットワークデバイスへの I/O が多発する．このことから，測定結果における KVM の大きな性能劣化は，I/O と割り込みのエミュレーション処理による性能劣化であると考えられる．

4.5.5　ライブマイグレーション中の性能

マイグレーション中のネットワークスループット

本システムにおけるライブマイグレーション中の性能を明らかにするために，ライブマイグレーション中のネットワークスループットを測定した．この測定においても，Netperf を用いた．Netperf のクライアントプログラムはクライアントマシンで動作させ，サーバープログラムは測定対象のサーバープログラムで動作させた．第 4.4.3 節で述べた通り，本プロトタイプはメモリの転送速度を調整しておらず，概ね NIC のラインレートである 1 Gbps でメモ

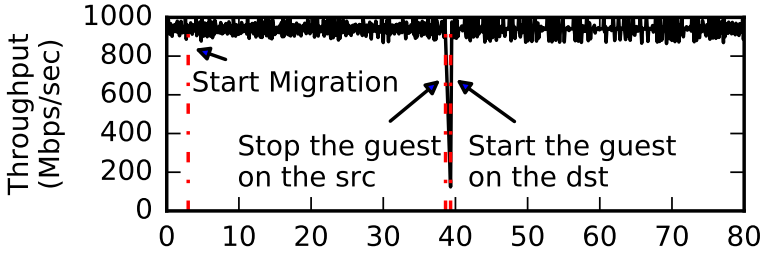

図 4.20　提案手法による Linux のライブマイグレーション中のネットワークスループット

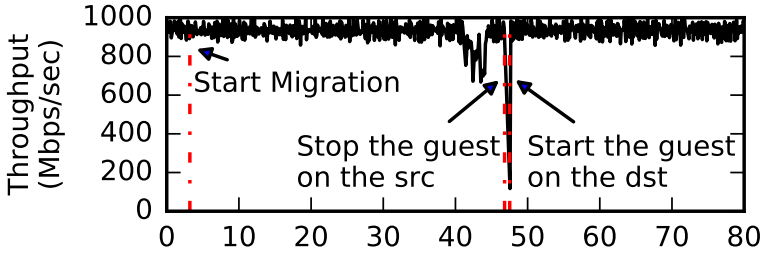

図 4.21　提案手法による Windows のライブマイグレーション中のネットワークスループット

リのデータを転送する．本評価環境において，OS 上で動作する Netperf プログラムが利用する NIC と PMM がライブマイグレーションのために用いる NIC は別であることに留意されたい．

図 4.20 と 図 4.21 に，それぞれ Linux と Windows のライブマイグレーション時の測定結果を示す．このグラフはライブマイグレーション前後の 100 ミリ秒毎のスループットを示している．ライブマイグレーションはグラフ内の 3 秒時点から開始した．Pre-copy フェーズは約 37 秒続き，その後，Stop-and-copy フェーズの処理が行われた．その後，マシンは移動先のものに切り替わっている．この結果から，提案するシステムにおけるライブマイグレーションの処理が，OS が利用する 1GbE ネットワークデバイスの性能へ大きく影響しないことがわかる．また，ライブマイグレーション完了後にも特に性能への影響がないことがわかる．

マイグレーション中の VM exit の回数と計算性能への影響

ライブマイグレーションの処理が OS やアプリケーションの処理性能にどの程度影響を与えるか解析するために，Linux のライブマイグレーション中における VM exit の回数を測定した．ライブマイグレーション中のワークロードは 第 4.5.3 節 と同様のものを用いた．測定の結果，VM exit の回数は Preemption timer によるものを除いてほぼ同程度に発生していた．また，preemption timer による VM exit は秒間平均で 837.5 回発生していた．注意として，preemption timer による VM exit のほとんどはメモリ転送を行うコアでのみ起き，その他のコアでは秒間 1 回程度しか起きていない．また，VM exit の処理のために費やした時間は全コアの合計で秒間約 320 ミリ秒 ($\approx$ 1,276 M cycles / 4.0 GHz) であった．これについても，ほとんどの時間はメモリ転送を行うコアが費やした時間であり，他のコアでの VM exit 処理時間は小さい．次の実験から，このことを示す．

次に，上記の VM exit による処理が CPU の性能にどの程度影響を与えるのかを Sysbench の CPU テストを用いた測定で明らかにする．ここではまず，シングルスレッドによる実行時間を通常時，ライブマイグレーション時のメモリ転送を行っている CPU 上，およびライブマイグレーション時のその他のコアで測定した．測定の結果，それぞれ 8.16 秒, 11.93 秒

(46.2% の増加), および 8.17 秒 (0.1% の増加) となった．この結果から，pre-copy 処理を行っているコア上では性能への影響は大きく，それ以外のコアにおける性能への影響は無視できるほど小さい．その他のコアにおける追加処理は dirty ページの記録を 1 秒間に一度行うのみである．このため，この結果はそれぞれのコアで dirty page を記録することによる性能への影響は大きくないこと示していると言える．

次に，マルチスレッドにおける実行時間を通常時とライブマイグレーション時それぞれにおいて測定した．スレッドの数は CPU のコア数と同じ 4 スレッドとした．実行時間はそれぞれ 2.04 秒と 2.22 秒 (8.8 % の増加) であった．この結果から，pre-copy 処理を行うコアでの性能劣化は大きかったものの，システム全体としての性能劣化は比較的小さく抑えられているといえる．これは，pre-copy 処理をしていない CPU が処理を肩代わりしているからであると考えられる．

ダウンタイム

最後に，提案手法のライブマイグレーションによるダウンタイムを測定した．ダウンタイムは，クライアントマシンから 100 ミリ秒ごとに ping パケットを送信することによって測定した．ダウンタイムの平均，最大値，および標準偏差はそれぞれ 0.861 秒，1.15 秒，および 0.104 であった．ダウンタイムの大半はメモリ上のデータを転送する処理によるものである．現在の実装では，pre-copy の閾値を 64 MB に設定しているため，stop-and-copy フェーズ内では 64 MB のメモリを転送する必要がある．1 GbE の NIC では，64 MB のデータ転送は 0.5 秒の時間を要する．$(64(MB) \times 8(bit) \div 1000(Mbit/sec) \approx 0.5(sec))$ その他のダウンタイムは，CPU やデバイスの状態取得，転送，および復元と経路上のスイッチの ARP テーブルの更新がある．状態の取得や復元の処理には，先述のダミーパケットの送信処理も含まれる．

ダウンタイムを詳細に調べるために，Stop-and-copy フェーズ中の損失パケットの数を測定した．測定には，tshark (WireShark の CLI 実装) を用いた．測定の結果，stop-and-copy フェーズにおいて 1,448 バイトのパケットを 31 個再送 (合計 44,888 バイト) されていることがわかった．これらの再送は 2 つの “TCP retransmission” パケットと，1 つの “fast retransmission” パケット，および 28 個の “selective ACK response” パケットが含まれていた．再送パケットは通常の通信時のパケットに比べて細かく分割されている．この結果から，再送パケットの数は許容範囲内といえる．

4.6 議論

本節では，本手法の制約や応用の方向性について議論する．

4.6.1 チェックポイント機能への応用

仮想マシンのチェックポイント機能はフォールトトレランスやイメージ管理のために有用な機能の一つである．チェックポイントとライブマイグレーションの機構の大半は共通している．両者の違いは，マシンの状態の転送先がファイルであるかネットワークであるかのみである．それ故に，本提案手法はわずかな変更でチェックポイントシステムへ応用できる．本手法は状態を破壊的に取得するため，状態取得後には元の状態が維持されない．このため，情報取得後そのままでは OS の動作を維持できない．しかしながら，本手法における状態の復元手法で同じマシンで状態を復元すれば，チェックポイント後にも OS の動作を継続できる．

4.6.2 デバイスへの対応

本手法はデバイスの仕様に依存した手法となっている．10GbE NIC や NVMe ストレージ，InfiniBand，GPU などの他のデバイスに対応するためには，状態の取得と復元を行うためのデバイス用のモジュールの開発が必要である．これらのモジュールの開発者はデバイスの仕様を理解する必要がある．しかし，これらのモジュールは状態の取得と復元に必要な処理のみ実装すればよく，デバイスドライバのようにデバイスを完全にコントロールする必要はない．このため，マイグレーションに必要なこれらのモジュールのコード行数はデバイスドライバのそれと比べて非常に小さい．将来的にデバイスドライバ開発支援や自動化の技術が開発されれば，それらの仕組みが必要なモジュールの開発にも役立つと考えられる．

4.6.3 PMM の動的な起動と終了

ライブマイグレーションの機能に限れば，ライブマイグレーション実行時以外で PMM が常に行う必要がある処理はない．しかしながら，PMM が起動しているだけで CPUID 命令などによる VM exit が発生することによる性能劣化 (4.5.3) や，EPT による 2 段のアドレス変換による性能劣化が存在する．もし通常動作時に CPU の仮想化支援機能を無効にし PMM を完全に停止できれば，通常時の性能劣化は完全になくなり物理マシンと同等の性能を実現できる．PMM の動的な起動と終了は以下の 2 つの問題を伴う．一つ目は PMM 自身の保護の問題である．現在の実装では，EPT によってメモリ上にある PMM のコードとデータを保護している．このため EPT を無効にしてしまうと，メモリ上に残っている PMM に関するデータを保護できない．このことは仮にゲスト OS が信頼できる状態ならば問題とならない．また，ARM CPU にある TrustZone のように特定のメモリを保護する CPU 機能があれば，このような機能を用いてより小さいオーバヘッドで PMM の保護を実現できる [49]．二つ目の問題は PMM をどのようにして起動するかである．仮想化支援機能は無効になっている状態では PMM を動作させることはできない．仮想化支援機能を有効にし PMM を起動するために， ゲスト OS にエージェントプログラムをインストール方法が考えられる．しかしながら，ゲスト OS 上で動くプログラムに依存すると，OS 非依存ではなくなり，IaaS クラウドで用いる手法としては適さない．本稿で述べた手法の発展として，Im ら [50] は BIOS を改変することで OS に依存せずに PMM の起動を行う手法を提案している．この論文内ではデバイスの状態転送が完全に OS 非依存になっていないものの，PMM の動的な起動と停止はデバイス状態の処理と直交するため，両者を組み合わせることは可能であると考えられる．

4.6.4 制約

ここでは本手法の制約について述べる．デバイスの状態転送における制約として，デバイスが持つ統計情報などを復元することは難しいことが挙げられる．例えば，NIC が持つエラーパケットの数などを復元するためには，それと同じ数のエラーパケットを受信させる必要がある．

また，本手法を導入する上での制約として，マシンを構成するデバイスの仕様を詳細がわかるものにしか対応できない点が挙げられる．本手法はデバイスの仕様に基づいて処理を記述する必要があるため，仕様に関する情報を入手できないハードウェアやデバイスに対して

は本手法を適用することができない.

第 5 章

物理マシンモニタによるハードウェア保護

ハードウェアの保護機能は提供する物理マシンにおけるセキュリティを維持するために必要な管理機能である．ここでは，保護されてないことによるセキュリティ面での問題，これまでの対応について述べた後に，本研究での提案手法，実装及び評価結果を述べる．

5.1 脅威モデル

現在のベアメタルクラウドで提供される物理マシンサーバーは x86 アーキテクチャの CPU を用いる IBM PC/AT を基にしたコンピュータアーキテクチャを採用している． このアーキテクチャは元々 PC を想定したコンピュータアーキテクチャである．PC は個人で占有利用したり，組織の限られたユーザーの間で共有されたりすることが想定される．つまり，PC の所有者と利用者は同一かそれに近い状況である．このため，メインフレームのように所有者とは異なる不特定多数のユーザーからリモートで利用されることは元々想定していない．しかしながら，ベアメタルクラウドではこの PC アーキテクチャのサーバーコンピュータを不特定多数のユーザーがリモートで利用しているのが現状である．本研究で想定する脅威モデルは，この用途の想定の違いから生まれた状況であると考えられる．

また，IaaS で提供される仮想マシンと物理マシンではライフサイクルが異なることもベアメタルクラウドでのセキュリティ問題に関連する．仮想マシンと物理マシンのライフサイクルについて図 5.1 と図 5.2 に示し以下で説明する．これまでの一般的な IaaS クラウドでは，ユーザーが利用するマシンは仮想マシンであった．この仮想マシンは，ソフトウェアによって注文された際に生成されている仮想的なコンピュータである．ユーザーが利用を止め仮想マシンを返却した際には，その仮想マシンは単に破棄され，再利用されることはない．一方，ベアメタルクラウドで貸し出されるマシンは物理マシンである．このため，マシンが返却されると，事業者はマシンのストレージ内のデータを消去し，BIOS の設定などをリセットした後に，別のユーザーに提供される．つまり，物理マシンが異なるユーザーに再利用される．このようなマシンの運用のため，ユーザーが物理マシンを利用する際には以前利用したユーザーによる変更の影響を受ける可能性がある．

上記の背景を踏まえ，本稿で想定するベアメタルクラウドでの脅威モデルについて図 5.3 に示し，以下で説明する．本稿の想定では，攻撃者はベアメタルクラウドの一般ユーザー，非攻撃者はクラウド事業者と他のクラウドユーザーを想定する．攻撃の流れは以下の通りである．攻撃者であるユーザーはクラウド事業者から物理マシンを注文し，そのマシンに対して

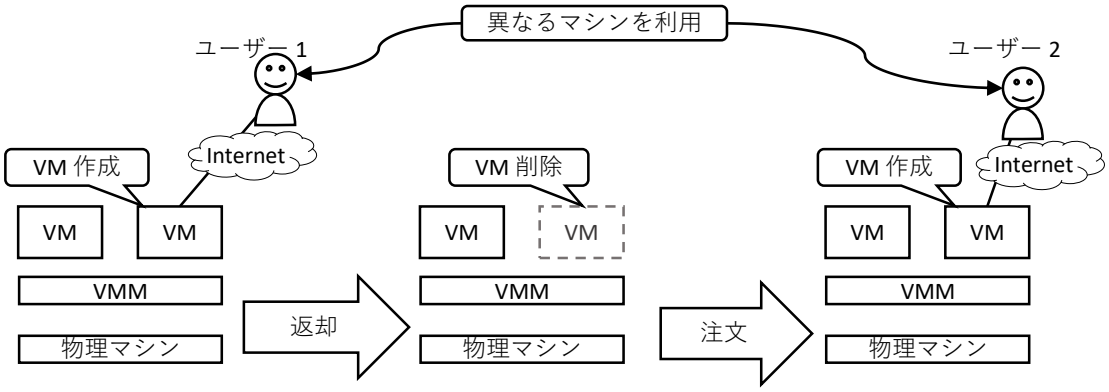

図 5.1　IaaS クラウドにおける仮想マシンのライフサイクル

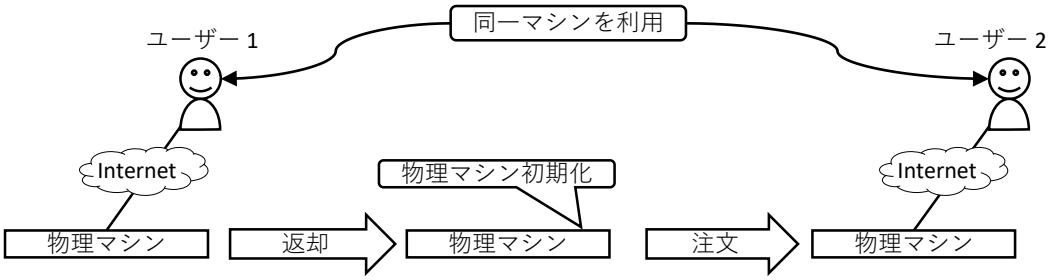

図 5.2　IaaS クラウドにおける物理マシンのライフサイクル

インターネットを介してアクセスする．攻撃者は，物理マシン上の OS における管理者権限を利用し，物理ハードウェアに対して攻撃する．攻撃としては，物理ハードウェアが恒久的に動作しなくなるような攻撃や，悪意のあるマルウェアをインストールするといった攻撃を想定する．このような攻撃を行った後，攻撃者であるユーザーは物理マシンをクラウド事業者へ返却する．仮に攻撃者が物理ハードウェアを動作不能にしている場合，クラウド事業者は返却されたマシンでサービスを提供できなくなる．このような攻撃は PDoS (Permanent DoS) と呼ばれている．また，仮に攻撃者が物理ハードウェアにファームウェアルートキットをインストールした場合，後にこの物理マシンを利用するユーザーがルートキットからの攻撃の被害を受ける．例えばルートキットによってデータを破壊されたり盗み出されることが考えられる．また，ベアメタルクラウドのサービスと合わせて仮想マシンを提供する IaaS や PaaS を提供している場合は，返却された物理マシンをこれら別のサービスに再利用することも考えられる．そうすると，クラウドベンダーの持つサービスにかかわる設定情報などの機密情報が盗まれる可能性もある．

この脅威モデルの中で想定している攻撃箇所は，物理マシンを構成する物理ハードウェア内にある不揮発領域である．ここで述べる不揮発領域とは， MAC アドレスなどの設定情報，スペック情報，およびファームウェアといった不揮発で，かつ，ハードウェアの動作上重要なデータが格納される箇所を指す．ストレージデバイスが格納しているユーザーデータなどは不揮発ではあるものの，今回の焦点を当てる対象ではない．この不揮発領域を改変することで，ハードウェアが恒久的に異常な動作するようにしたり，起動しなくしたり，ファームウェアルートキットをインストールしたりといった攻撃が可能である．ハードウェアの持つ状態はほとんどが揮発性であり，マシンを再起動したりデバイスをリセットしたりすることで初

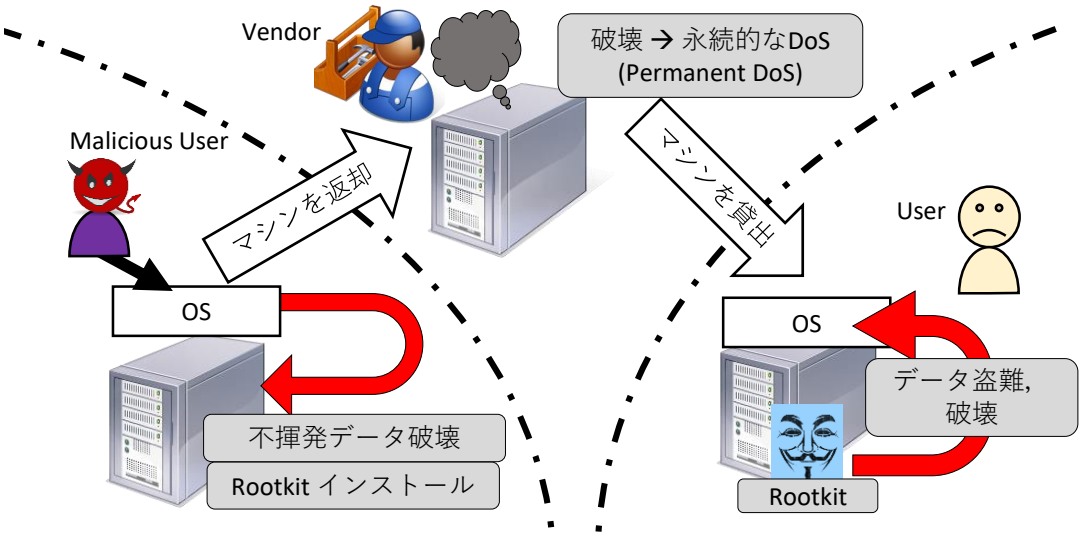

図 5.3　脅威モデル

期化される．このため，マシンを返却した後の処理に影響を及ぼさない．しかしながら，不揮発領域の改ざんは物理マシンを再起動しても変更が残るため，その影響もまた再起動だけでは取り除くことができない．ベアメタルクラウドにおいて，ユーザーは提供される物理マシンのハードウェア機能を全て利用できる．その中には，上記の不揮発領域へアクセスするための機能も含まれているため，攻撃者は比較的容易に不揮発領域を攻撃できると考えられる．

ハードウェアを機能不能にする例として NIC の EEPROM に対して不正な値を書き込むことによって NIC を動作不能な状態にできるといったものがある [51, 52, 53]．加えて，EEPROM はハードウェアの仕様上書き込み回数の上限があるため，単純に EEPROM に対して大量の書き込みを行うだけでハードウェアが故障する可能性がある．さらに， 不揮発領域内のデータを注意深く操作することで，攻撃者は当該デバイスを利用するシステムのファームウェア (BIOS や UEFI) がハングするようにし，物理マシンが起動しないようにすることもできる．

また，ファームウェアルートキットについても，技術的に難度が高いものの，実際にこのような攻撃はいくつも報告されている．例えば，Delugre[54] らは Broadcom NIC で動作するルートキットを実装した．また，別のセキュリティ企業は Intel 製マザーボード上の BIOS ROM に UEFI ルートキットをインストールできる悪意のある UEFI アップデーターを実装した [55]．加えて，NSA は DEITYBOUNCE と名付けられたマルウェアアプリケーションを実装した [56, 57]．これは，DELL 製のマシンにおいて，悪意のあるコードを BIOS にインストールすることで定期的に任意コード実装ができるものである．Zaddach[58] らは HDD のファームウェア内に物理的なアクセスなしでルートキットをインストールできることを示している．ファームウェアはソフトウェアスタックの中で最も高い権限を持つため，ファームウェアルートキットは検知や除去が難しい．さらに，ルートキットはマシン内にあるほとんどのデータにアクセスできるため，センシティブなデータを盗み出したり，ユーザーのデータを破壊したりすることもできてしまう．

さらに注目すべき点としては，これらの攻撃はクラウドベンダーが提供している API を用いることで自動化できる点である．なぜなら，これらの API を用いることで，プログラムか

らインスタンスを注文し，管理者権限でプロビジョニングスクリプトを実行し，インスタンスを返却するという処理ができるためである．それ故に，上記の攻撃は短期間で膨大な数の物理マシンに対して行うことができ，甚大な被害になる可能性がある．このため，ROM ライターでの復旧のようなコストが大きい復旧手段は現実的な対応策とは言えない．

現在の物理マシンで可能な対応策は，主にハードウェアによる保護機能とデータの書き戻しである．しかし，これらの対処法では物理マシンのハードウェアを保護するには不十分である．その理由を以下で説明する．ハードウェアの中には不揮発領域の write-protect を提供するものもある．例えば，チップセットは BIOS ROM を保護するためのハードウェア機能を提供している．一般的に，保護機能の有効化は BIOS や UEFI のようなファームウェア自身が行う．しかしながら，いくつかのファームウェアはこれらの保護機能を正しく有効化できてない．実際に，Intel によって作成されているシステムプラットフォームのセキュリティ検査ツールである CHIPSEC[59] によって研究室内のマシンを調査したところ，BIOS ROM への書き込み保護機能が有効になっていないなど，設定が不十分であった．さらに，実際のベアメタルクラウドが提供している物理マシンにおいてもこれらの機能が有効になっていないものがあった．つまり，攻撃者はベアメタルクラウドで提供されているこのようなマシンにおいては，BIOS ROM に容易にアクセスできる．ハードウェアによる保護機能が有効になっていたとしても，これらの保護機能に脆弱性が見つかる可能性もある．実際に，過去には SpeedRacer[60] と呼ばれる脆弱性が見つかっている．このような脆弱性のために，ハードウェアによる保護が有効であっても NVM が書き換えられる可能性がある．またハードウェアの脆弱性はソフトウェアのものと異なり，ハードウェアを入れ替える以外に対応策がない場合もある．さらに，周辺デバイスの中にはハードウェアによる保護機能を持たないものもある．このため，ハードウェアの保護機能のみで全ての攻撃を止めることは難しい．

また，マシン返却後に不揮発データを書き戻す方法も考えられる．不揮発データの書き戻しによって，軽微なデータの変更は修正できる．しかし，ハードウェアが起動しなくなるような不揮発データが改ざんをされた場合，書き戻し処理を実行できない．また，ファームウェアにルートキットがインストールされていると，ルートキットによってデータの書き戻し処理が止められ，書き戻し処理が正常に終わったかのように結果を偽装される可能性もある．ROM ライターなどのソフトウェアに依存しない方法で不揮発データを書き込みことは可能であるが，これは大変手間のかかる作業であり，物理マシンの運用として現実的ではない．

本研究での脅威モデルでは，攻撃者による物理マシンへの物理的なアクセス (マシンの分解など) は想定していない．なぜなら，攻撃対象の物理マシンはクラウド事業者が管理するデータセンター内に置かれており，一般的に物理アクセスは容易でないためである．また，クラウド事業者による攻撃も想定しない．

5.2 物理ハードウェアのセキュリティにおける関連研究

この章では，これまでに行われているハードウェアへの攻撃を対象とした研究を示す．

Loïc らの研究 [61] では，NIC の脆弱性を用いてマシンの制御を奪取できることを示している．また，この研究ではこのような攻撃を行うマルウェアを検知する手法として NAVIS を提案している．NAVIS は NIC のファームウェアに対して整合性のチェックを行うことでマルウェアの検知を実現している．Li らは [21] 周辺デバイスと OS 間でチャレンジレスポンスプロトコルを用いることでマルウェアを検知する VIPER と呼ばれる手法を提案している．VIPER では，OS からデバイスに対してチャレンジレスポンスを送信し，デバイスからの返答が到着するまでの時間を測定する．この返答までにかかる時間が一定以上大きければ，マ

ルウェアが存在していると判定する．VIPER は周辺デバイスに対する攻撃においてそれまで知られていたもの全てを検出できる．その中には，プロキシを用いた攻撃もある． 上記 2 つの手法はともにファームウェア内に存在するマルウェアに対するものであり，利用しようとする物理マシンが安全か否かを検査する上では有用である．しかし，これらの手法ではマルウェアのインストールを防げない．加えて，どちらの手法も OS が信頼できる環境を想定しているものの，今回の脅威モデルにおいて OS は信頼できない．それ故に，これらの手法はベアメタルクラウドで事業者が提供するマシンを保護するという用途には不向きである．今回提案する PMM による防御手法は OS には依存せずに不揮発領域への書き込みを防ぐことで，OS が信頼できない環境でマルウェアのインストールを防ぐ．この想定の違いは PC のように所有者と物理マシン上で動作する OS の特権を持つ利用者が同一であるという仮定が成り立たないベアメタルクラウドというマシンの特殊な利用形態に起因する．

Zhang ら [62] が提案した IOCheck は，CPU の最高特権モードである SMM モードを用いて周辺デバイスやファームウェアのチェックする手法である．SMM で実行されるコードは BIOS によって管理され OS の依存なしに動作するため，IOCheck もまた OS 非依存な手法である．SMM での実行コードを管理する BIOS が安全に起動するために，起動時に BIOS の検査も行う．しかしながら，この手法は BIOS を改造する必要があり，一般的にこの改造は困難である．加えて，この手法においてもマルウェアの検知のみを行い，マルウェアのインストールは防げない．本提案手法で用いる PMM は x86/x64 マシン上で動作するオープンソースの Type 1 ハイパバイザを基にした実装ができるため，BIOS を改造する手法に比べて導入が容易である．また，PMM は BIOS よりも細かく OS の動作を制御できるため，OS からデバイスやファームウェアへの攻撃を防ぐことができる．

本稿で提案する PMM はハイパバイザ同様 CPU の仮想化支援機能を用いており，ハイパバイザと共通する点も多い．これまで，ハイパバイザを用いてセキュリティを向上するための研究は数多く行われている [63, 64, 65, 66, 67, 68]．一般的にこれらの研究はハイパバイザが物理デバイスを隠蔽し不揮発領域への直接的なアクセスを防ぐことでデバイスを保護していた．しかし，一般的なハイパバイザは複数の VM を一つの物理マシンで動作させるためにその処理の多さと複雑さが増し，性能劣化と脆弱性を抱えるリスクから逃れられない．複数 VM を動作させるにはハードウェアの仮想化が必要であり，また，仮想 CPU や仮想デバイスのスケジューリングなどの処理を伴う．これらの多様で複雑な処理を要するために性能劣化は不可避であり，同時にハイパバイザの複雑さが増すことで trusted computing base (TCB) のサイズが増大することも避けられない．TCB 内のバグは脆弱性に直結するため，巨大で複雑なハイパバイザはセキュリティ対策には不向きである．

以下では，本研究に関連するマルウェア解析，検知，および防御手法についての研究において，上記以外のものについて述べる．Kirat ら [69] は VM-aware なマルウェアを解析するサンドボックスの構築手法として BareBox を提案した．近年，VM を用いてマルウェアを解析することが増えている．解析に VM を用いる理由は，VM は他の環境からの隔離が容易であること，VMM などの VM 外部から動作を監視できマルウェアによって監視を妨害されづらいこと，マルウェアによって攻撃されたシステムを攻撃前の状態に容易に復元できること，が挙げられる．しかしながら，VM-aware と呼ばれる特性を持つマルウェアは，VM 上で動作していることを検知すると，自身が解析されることを防ぐために悪意のある挙動をしないようにしている．このようなマルウェアを解析するために，BareBox は物理マシン上で直接マルウェアを動作させ，解析後に OS を攻撃前のクリーンな状態に復元する手法を提案している．しかしながら，BareBox が復元するのは OS のみであり，BIOS やデバイスの不揮発領域の復元には対応していない．このため，BareBox 上のマルウェアがこれらを攻撃した場

合，解析後にも攻撃の影響が残ってしまう．一方，本稿で提案する PMM による手法は物理マシン環境で BIOS やデバイスの不揮発領域を保護する．Bulygin ら [70] は virtualization ルートキットや SMM ルートキットを検知し除去する手法として DeepWatch を提案している．DeepWatch は チップセット内にある DRAM へのアクセスを制御するコントローラーを利用する．このアプローチは OS に依存しないものの，チップセットの内部を制御する必要があり，チップセットへの依存が大きい．一方 PMM による手法では，CPU の持つ仮想化支援機能のみで実現できる．Vasiliadis ら [71] は GPU-assisted マルウェアが実現可能であることを示している．GPU-assisted マルウェアはマルウェアの一部を GPU で実行することで CPU やメモリの監視による検知を困難にしている．周辺デバイスに着目した攻撃手法であるという点で，本研究と関連している．しかし，GPU で実行されるコードは揮発領域に置かれるものであり，システムを再起動することで容易に取り除くことができる．本研究では，攻撃によってファームウェアのデバイスの不揮発な情報を書き換えることを想定しているため，対象が異なる．

5.3 設計

この章では，提案する防御手法の設計について述べる．提案手法は物理ハードウェアの不揮発領域を小さいオーバーヘッドで保護する．

5.3.1 提案手法の概要

ベアメタルクラウドにおけるハードウェア保護システムは OS 非依存で，かつ軽量で小さくあるべきである．理由は以下の二つある．一つ目の理由は，クラウド事業者はベアメタルインスタンス上で動作する OS を信用することはできないことである．なぜなら，クラウドのユーザーはあらゆる OS をインストールし動作させることができるためである．それ故に，保護システムは OS とは隔離されたソフトウェアレイヤで行われる必要がある．同時に，保護システムはベアメタルインスタンスの性能を維持する必要があり，このためにオーバヘッドを小さく押さえる必要がある．加えて，システムの規模を小さくし，バグによる脆弱性が入り込む余地を減らすべきである．

OS 非依存でかつ低オーバヘッドで小さいシステムを実現するために，本提案手法では PMM によってハードウェア保護を実現する．PMM は OS 非依存に動作する設計であるため，OS に依存せずに不揮発領域を保護できる．不揮発領域を保護するため，PMM は不揮発領域への書き込みにのみに介入する．その他の I/O には極力介入しないようにすることで，性能劣化を最小限に押させる．この設計はシステムを小さく保つことにも寄与する．

PMM はほとんどの通信に介入しないものの，デバイスの不揮発領域を OS から保護するために，OS からデバイスへの書き込み I/O の一部には介入する必要がある．このために，保護対象の物理デバイスのための準パススルードライバは作成し，一部の通信を捉える．この準パススルードライバを用いて，デバイスの不揮発領域への書き込み I/O を捉え，その書き込みを許可するか拒否するかを判定する．

5.3.2 提案手法によるハードウェア保護

図 5.4 に PMM によるハードウェア保護の概要を示す．提案手法における PMM の役割は 2 つある．1 つ目は，ファームウェアが有効にしていないハードウェアの保護機能を有効

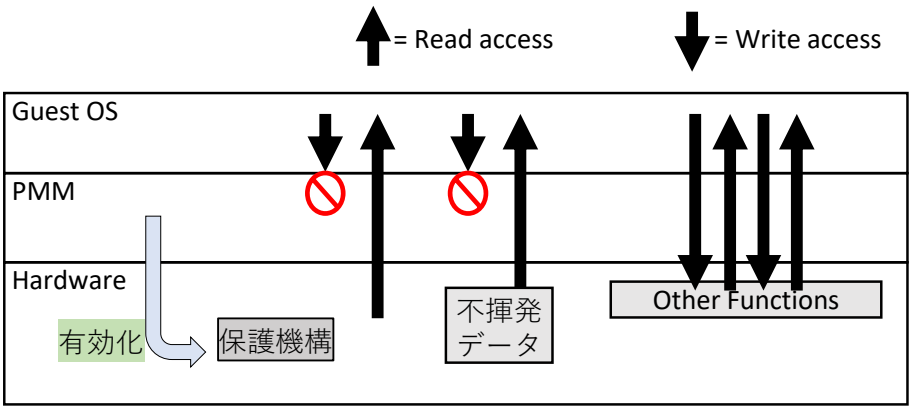

図 5.4　提案手法によるハードウェア保護

にすることである．PMM はデバイスが不揮発領域を保護するための機能を持っており，なおかつ，これが有効になっていなければ，OS 起動前にこれを有効にする．本来，これらの機能は BIOS や UEFI などのファームウェアによって有効にされていることが期待されるが，実際のマシンでは有効になっていないことが多い．ハードウェア自身によるこれらの保護機能は，一度有効にするとマシン全体を再起動するまで無効にすることができないものがほとんどである．このため，PMM がこれらの機能を有効にすると，OS から無効にすることはできない．しかしながら，これらの機能にも脆弱性がある可能性があるため，多重防衛のために PMM はこれらの機能の設定を OS からアクセスできないようにする．保護機能の有効化は，物理マシンモニタがハードウェア保護機能の仕様に従って有効化の処理を行う．具体的には，指定のレジスタに適切な値を書くという処理が主である．PMM の開発者はこれらの機能をどのように有効にするかは，デバイスの仕様書やデータシートを見ることで知ることができる．

2 つ目は，不揮発データへの書き込み要求を直接的に遮断することである．具体的には，ゲスト OS が不揮発データを書き換えるような要求をハードウェアに送ろうとした時に，物理マシンモニタでこれを遮断するというものである．その他の機能に関しては，基本的には全てパススルーとする．また，不揮発領域へのアクセスであっても，読み出しアクセスであれば準パススルードライバは介入せずゲスト OS が直接アクセスする．このため，OS は通常のワークロードでは物理マシンモニタの介入なしに物理ハードウェアを利用できる．また，通常のワークロードに置いては，物理マシンモニタによる介入は行われないので，性能劣化も小さい．

不揮発データへの書き込みアクセスは (1) メモリや I/O 空間を介したアクセス，(2) コマンドの処理依頼によるアクセス に分けられる，それぞれについて対応が必要である．(1) のメモリや I/O 空間を介したアクセスでは，ゲスト OS が特定のメモリアドレスや I/O アドレスに対して値を書き込むことで，不揮発データを書き換える．このため，物理マシンモニタは，この特定のアドレスへの書き込みを監視し，不揮発データを書き換えるアクセスは破棄する．(2) のコマンド処理依頼によるアクセスは，ゲスト OS がメモリ上に存在するコマンドキューを介して，デバイスに不揮発データの更新コマンドの処理を依頼することで不揮発データを書き換えるものである．(1) と異なり，アクセスするアドレスで判断することができないため，コマンドの内容を検査し，不揮発データを書き換えるコマンドであれば，これを無効なコマンドに置き換える．この結果，デバイスはファームウェアアップデートのコマンドを受け取ることなく，ファームウェアが書き換わることもなくなる．

5.3.3 PMM の起動と保護

PMM の起動のタイミングとして，OS の起動前と起動後の 2 つが考えられるものの，PMM は 2 つの理由から OS よりも先に起動する必要がある．一つ目の理由は，PMM は OS が起動する前にハードウェアの保護機構を有効にする必要があることである．一つ目の理由は，OS 起動後に PMM を起動しようとすると，一般的に OS に依存した手法になるためである．二つ目の理由は，PMM 自身を保護する必要があるためである．PMM を保護するためには，ストレージ内にある PMM のイメージの改ざんを防ぐ必要がある．マシンのローカルストレージに PMM のイメージを置く場合，PMM 自身でストレージデバイスを隠蔽したり PMM が格納されている領域へのアクセスを防いだりする必要がある．また，管理対象の物理マシン以外のマシンに PMM のイメージを格納し，物理マシン起動時に PMM のイメージをダウンロードし起動することで，物理マシン上で起動する OS からイメージファイルに直接アクセスできないようにすることでも保護できる．また，PMM が動作中にメモリ上に展開されているコードを改ざんされることも防ぐ必要がある．OS からの改ざんはネスティドページングで PMM のコードがある部分のアクセス権限を付与しないことで解決できる．また，デバイスの DMA から保護するためには，IOMMU で同様に PMM の領域へのアクセスを禁止することで解決する． さらに，起動時には BIOS が OS に通知するメモリマップを改変し，PMM が利用している領域を OS が利用不可な領域として報告するようにしている．これらの理由から，クラウド事業者は物理マシンのファームウェアにおいて PMM が最初に起動するように設定を行い，これをユーザーが変更できないように設定しておく必要がある．

5.4 実装

本章では PMM によるハードウェア保護システムの実装について述べる．PMM のプロトタイプは BitVisor[30, 31] をベースに実装した．今回実装した PMM のプロトタイプは Intel 製の CPU にのみ対応しているものの，提案する手法は AMD 製の CPU やその他 Intel VT と同等の仮想化支援機能を有する CPU 向けに実装できると考えている．

5.4.1 実際のハードウェアにおける Attack surface

今回保護の対象としたマシンの構成は以下の通りである．マザーボードは ASRock X99 Extream4 であり，このマザーボードは Intel C610/X99 チップセットを搭載している．また，CPU は Intel 製 Xeon CPU E5-2603 v4 (1.70GHz) であり，NIC は Intel 82574L Gigabit である．また，このマシンは BIOS による起動と UEFI による起動の両方に対応しているものの，今回は BIOS により起動するように設定した．今回実装したプロトタイプは BIOS ROM と Intel 製 NIC を保護している．現在はストレージデバイスなどのデバイスには対応してないものの，本手法を他のデバイスへ対応する手間は，そのデバイスの仕様に関する情報が入手できれば Intel 製 NIC へ対応するのと同程度であると考えられる．

不揮発領域を保護するためにはハードウェアによる不揮発領域の保護機能を有効にしておくべきである．そこでまず，物理マシン上で直接 Linux が動作している状態でどの機能が有効になっているか否かを調査した．この調査には，CHIPSEC[59] コマンドを用いた．調査の結果，今回の保護対象のマシンでは以下の保護機能は有効になっていないことが判明した．

BIOS の書き込み許可フラグ　このフラグがセットされている時に限り，ソフトウェアは

BIOS ROM に対して書き込みを行うことができる．BIOS ROM に対する書き込みを防ぐためには，このフラグのセットを禁止する必要がある．また，このフラグによる BIOS ROM の保護には競合状態における脆弱性が発見されている [60].

BIOS の書き込み保護　この保護機能が有効になっていると，ソフトウェアは全ての CPU が SMM モードで動作していない時には BIOS ROM に対して書き込みを行えなくなる．この機能を用いると，上記の脆弱性に対応することができる．

Serial Peripheral Interface (SPI) の Range Protection　この機能はチップセットが BIOS ROM へアクセスするためにソフトウェアに提供している Serial Peripheral Interface (SPI) を介して BIOS ROM にアクセスする際に，BIOS ROM の範囲ごとにアクセス権限を設定するものである．この機能で BIOS が格納されている領域の書き込み権限を全てなくすことで，ソフトウェアによる書き込みを防ぐことができる．しかしながら，今回の対象マシンではこの機能は使われていなかった．

SPI Configuration Lockdown　この機能を有効にすることで，上記の SPI Range Protection を含む SPI に関する設定をソフトウェアによって変更されることを禁止する．設定を変更するには，一度マシンを再起動するしかなくなる．今回の対象マシンではこの機能は有効になっていなかった．

PMM は BIOS ROM を保護するために，これらの機能を BIOS に替わって有効にする．これらの機能は，チップセットが持つレジスタ内にある対応するフラグをソフトウェアからセットすることで有効にできる．チップセット内のレジスタやフラグに関する情報は，チップセットの仕様書 [72] から知ることができる．一方，Intel 製の NIC にはこのようなハードウェアによる保護機能がそもそも存在しない．

PMM のもう一つの役割は，不揮発領域へアクセスするためのインターフェイスを全て塞ぐことである．ハードウェアの仕様書 [72, 73] によると，塞ぐべきインターフェイスは以下の通りである．

- BIOS ROM にアクセスするための SPI インターフェイス
- Intel NIC の不揮発領域にマルチバイトアクセスするために NIC が提供するレジスタ (EEWR)
- Intel NIC の不揮発領域がマップされたメモリ空間
- Intel NIC の Flash にアクセスするための SPI インターフェイス
- Intel NIC の EEPROM にアクセスするための SPI インターフェイス

Intel NIC は互換性を保つために不揮発領域にアクセスするためのインターフェイスを複数提供している．また，NIC の仕様上，不揮発領域として用いるデバイスとして EEPROM と flash の両方に対応しており，このために SPI インターフェイスはそれぞれのデバイス用に用意されている．今回実装したプロトタイプでは，以降の章で述べる通りこれらのインターフェイスを介したアクセスをブロックしている．

5.4.2　不揮発領域への書き込み I/O のブロック

書き込み I/O をブロックするために，PMM は書き込み I/O に介入する必要がある．この処理は，第 3.4.3 節 で述べた準パススルードライバで行う．書き込み MMIO の遮断を 図 5.5 に示している．PMM は保護対象のレジスタやメモリ領域に対しての書き込みがあった場合，

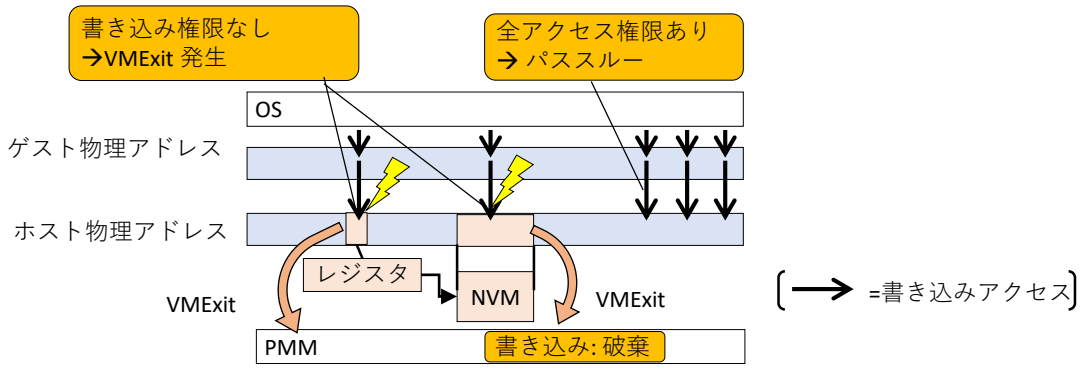

図 5.5　MMIO による不揮発領域への書き込みの遮断

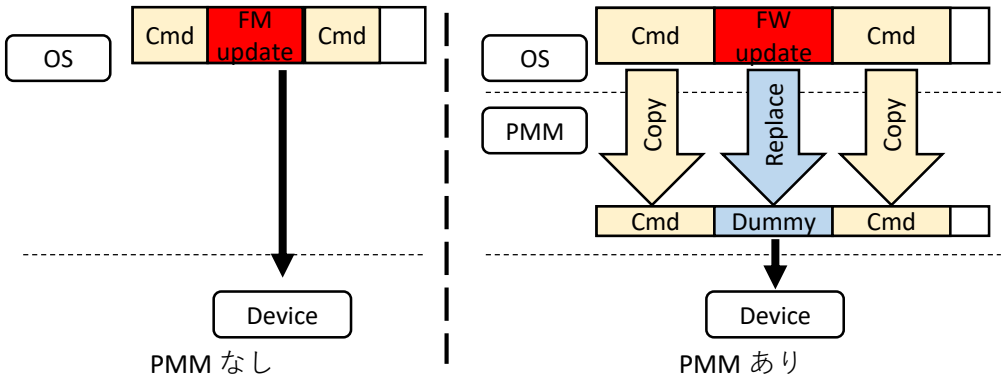

図 5.6　シャドーコマンドキューによる不揮発領域の改変防止

VM exit が発生するように設定する．また，このような書き込みで VM exit が発生した場合，基本的には単純に書き込み処理をスキップし，書き込み命令の次の命令から OS の処理を再開する．

不揮発領域の保護に関しては，基本的に読み出し I/O に介入する必要はなく書き込み I/O にのみ介入すればよい．EPT では読み出しと書き込みについて別々の権限を設定できるものの，従来の準パススルードライバは読み書きともに介入することのみを想定していた．そこで，MMIO への介入においては，EPT 内のパーミッションビットを用いて書き込み権限のみを付与しないことで VM exit を発生するようにした．一方，PIO への介入では読み書き別々に設定することができないため，当該レジスタからの読み出し I/O であっても VM exit が発生してしまう．この際には，PMM が元の読み出し I/O を代替して実行し，その結果を OS に返却する．

MMIO アクセスへの介入は上記の通り EPT を用いて行う．このため，介入対象のメモリ領域の設定がメモリページの単位でしか指定できない．それ故に，PMM が保護対象のレジスタへの書き込みアクセスに介入する場合，PMM はそのレジスタに隣接する MMIO レジスタへの書き込み I/O に対しても VM exit を起こしてしまう．このことは，頻繁にアクセスされるレジスタが保護対象のレジスタと同じページ内の配置されている場合，性能面で不利に作用する．実際に，このような状況は Intel NIC 内に存在する．しかしながら，これによる性能への影響は一般的な VMM による性能劣化と比べれば小さいものである．性能への影響については，第 5.5.3 節 と 第 5.5.4 節 での性能測定の結果で明らかにしている．また，この制約については第 5.6.1 節 でさらに論じる．

コマンドによる不揮発領域の改変を防止する手法を図 5.6 に示し以下で説明する．デバイスの中にはメモリ上にあるコマンドキューから OS がセットしたコマンドを読み出し，それを実行して動作するものがある．また，このコマンドの中にファームウェアをアップデートするためのコマンドなどがあり，これをデバイスが実行してしまうとファームウェアが更新されてしまう．これを防ぐために，PMM はコマンドキューをシャドーイングする．PMM が偽のコマンドキューを OS に見せ，OS には偽のコマンドキューにコマンドを置かせる．その後，OS がコマンドの実行要求をデバイスに転送した際に，PMM が本物のコマンドキューにコマンドをコピーする．この際，コマンドの中身を検査し，ファームウェアを更新するような有害なコマンドであれば，これをダミーのコマンドへと置き換える．この結果，デバイスはファームウェアを更新するようなコマンドを実行しなくなり，OS による不揮発領域の改変を防ぐことができる．

今回のプロトタイプ実装では，PMM は不揮発領域への書き込みを単純にブロックしている．この場合，ゲスト OS が不揮発領域へ書き込もうとした場合，インターフェイスによってはゲスト OS が正常に動作しなくなる場合がある．なぜなら，ゲスト OS が書き込み完了の通知を待ち受けているものの，単純に書き込みをブロックしてしまうと，この通知がゲスト OS に送られないためである．しかしながら，通常ベアメタルインスタンス上で動作する悪意のないソフトウェアはハードウェアの不揮発領域に書き込みを行わないため，単純にブロックするだけでも問題なく動作する．書き込みがあった場合でも正常に動作を継続させる必要がある場合には，ハードウェアの書き込み完了や書き込みエラー時の動作を部分的にエミュレートすることで解決できる．しかしこれを単純に実装しようとすると，読み出しアクセスにも介入する必要があり，性能劣化が大きくなる．この性能劣化を小さくするには，書き込み要求が行われてからエラーのエミュレートが終わる間でのみ読み出しアクセスへ介入するとよい．しかしながら，このような実装はメモリのアクセス権を頻繁に更新し，これを CPU の全コアに反映するために TLB シュートダウンを行う必要があるなど，実装が複雑になる．これまで，読み出しアクセスを全て介入することでエラーのエミュレートができることは確認しているものの，本稿での実験には利用していない．

5.4.3 PMM 自身の保護

PMM 自身を OS から保護するために，PMM のイメージは保護された領域に格納しておく必要がある．PMM によって PMM のイメージが格納されたストレージデバイスを保護することも可能ではあるが，今回はよりシンプルな方法としてネットワークブートを想定する．物理マシンが起動する際，PMM のイメージはネットワークサーバーからダウンロードされる．ネットワークブートの仕組み上，サーバー上のイメージを書き換えることはできないため，PMM のイメージは OS から保護される．ベアメタルクラウドにおいて，OS のデプロイなど既にネットワークブートで行っているため，ネットワークブートによる PMM の起動は容易であると考えられる．今回，ハードウェア保護に関する実験では PMM はローカルストレージから起動しているものの，マイグレーションに関する実験ではネットワークブートしているため，ネットワークブートは可能であるといえる．メモリ上のイメージについては，PMM 自身が EPT を用いて OS から保護する．具体的には，PMM が EPT によるマッピングを作成する際に，PMM 自身のメモリ領域に対するアクセス権限を全て無効にすることで保護を実現する．さらに，ゲスト OS 起動時に BIOS が OS に伝えるメモリマップの一部を改変し，PMM が利用している物理メモリ領域をソフトウェアから利用不可な領域として見せることで，OS が利用するメモリ領域と PMM が利用するメモリ領域が衝突しないよう

にしている．

また，I/O デバイスの DMA による書き替えに対しては，IOMMU を用いることで対応する．IOMMU とは周辺デバイス向けのメモリコントローラであり，CPU が用いる MMU と同様にデバイス用の仮想アドレス空間と物理アドレス空間のマッピングを管理する．また，物理アドレスへのマッピングが存在しない仮想アドレスへのアクセスは破棄される．保護の具体的な実現方法は IOMMU によってゲスト OS が制御しているデバイスによる DMA では PMM の利用領域へのマッピングを作成しないことで実現する．

5.4.4 PMM 起動の流れ

提案する保護システムにおける物理マシンの起動の流れは以下の通りである．まず，PMM が BIOS や UEFI, PXE (ネットワークブート用のファームウェア) といったファームウェアによって起動する．その後，PMM はハードウェアによる不揮発領域の保護機構を有効にする (第 5.4.1 節 参照)．次に，PMM は Intel VT-x を用い，ゲスト OS が用いる仮想 CPU の初期化を行う．この際，PMM はこの仮想 CPU から不揮発領域への書き込みができないように設定を行う．その後，仮想 CPU 上でブートローダを実行し，ゲスト OS の起動へと進む．この時点で，既にブートローダーやゲスト OS が用いる仮想 CPU は不揮発領域への書き込みができないため，ベアメタルクラウドのユーザーはブートローダーなどを改変しても不揮領域への書きみはできない．

5.4.5 PMM のサイズ

PMM のコアとなる部分のソースコードの行数はおよそ 4 万行である．またチップセットと Intel 製 NIC を保護するための準パススルードライバのコード行数は 5897 行であった．この中には，PCI デバイスを扱うための一般的なソースコードも含まれている．これらのコードサイズは，一般的なデバイスドライバや VMM と比べて小さい．

5.5 実験

本稿では提案手法における保護に関する実験と性能に関する実験を行った．

保護に関する実験は大きく分けて二つおこなった．一つ目はハードウェアによる保護機能が有効か否かの確認である．元々ファームウェアによってハードウェア保護機能が有効になっていなかった物理マシンにおいて，物理マシンモニタによって OS 起動後にハードウェア保護機能が有効になっているか否かを確認した．また，この保護機能を OS から無効にできるか否かを確認した．二つ目は OS による不揮発データへの実際の書き込みである．今回の実験では，Intel NIC の不揮発データのうち，MAC アドレスのデータの書き換えを実際に物理マシンモニタがない環境とある環境で行った．

また，性能評価においては，ネットワークデバイスの保護を実装したため，ネットワーク性能としてレイテンシを測定した．今回の測定では，TCP および UDP におけるネットワークスループット及びレイテンシを測定した．また，物理マシンと準仮想化ネットワークデバイスを用いる KVM を比較対象とした．

```
# chipsec_main
[...]
[!] None of the SPI protected ranges \
write-protect BIOS region
[...]
[CHIPSEC] Modules failed        2:
[-] FAILED: chipsec.modules.common.bios_wp
[-] FAILED: chipsec.modules.common.spi_lock
[...]
```

図 5.7　物理マシン上での chipsec_main の実行結果の抜粋

```
# chipsec_main
[...]
[+] PASSED: BIOS is write protected (by SMM and \
SPI Protected Ranges)
[...]
[CHIPSEC] Modules failed        0:
[...]
[+] PASSED: chipsec.modules.common.bios_wp
[+] PASSED: chipsec.modules.common.spi_lock
[...]
```

図 5.8　PMM 上での chipsec_main の実行結果の抜粋

```
# chipsec_util spi write 0x215270 data.bin
[...]
[CHIPSEC] writing to SPI flash memory at \
FLA = 0x215270 from 'data.bin'
[spi] writing 0x10 bytes to SPI at FLA = 0x215270 \
(in 4 0x4-byte chunks + 0x0-byte remainder)
[spi] writing chunk 0 of 0x4 bytes to 0x215270
[spi] writing chunk 1 of 0x4 bytes to 0x215274
[spi] writing chunk 2 of 0x4 bytes to 0x215278
[spi] writing chunk 3 of 0x4 bytes to 0x21527C
[CHIPSEC] completed SPI flash memory write
[CHIPSEC] (spi write) time elapsed 0.001
```

図 5.9　物理マシン上での chipsec_util spi write の実行結果の抜粋

5.5.1　実験のセットアップ

本実験において，物理マシンは第 5.4.1 節で紹介したマシンを用いる．また，ゲスト OS として Ubuntu 16.04.2, Linux カーネルのバージョンは 4.4.0 を用いた．性能評価における比較対象として Linux に組み込まれている VMM である QEMU/KVM を用いた．QEMU のバージョンは 2.5.0 を用いた．

5.5.2　保護の実験

提案手法の有効性を確認するための実験を行った．これらの実験では，ハードウェアによる不揮発領域保護機能が有効になっているか否かのチェックと，実際に不揮発領域へ書き込んだ際の挙動について確認した．これらの実験では，ハードウェアの機能についての調査には CHIPSEC 1.3.0 を，不揮発領域への書き込みは ethtool 4.5 を用いた．

CHIPSEC は chipsec_main というコマンドを提供しており，このコマンドはチップセッ

```
# chipsec_util spi write 0x215270 data.bin
[...]
[CHIPSEC] writing to SPI flash memory at \
FLA = 0x215270 from 'data.bin'
[spi] writing 0x10 bytes to SPI at FLA = 0x215270 \
(in 4 0x4-byte chunks + 0x0-byte remainder)
[spi] writing chunk 0 of 0x4 bytes to 0x215270
WARNING: SPI cycle not done
ERROR: SPI flash write cycle failed
[spi] writing chunk 1 of 0x4 bytes to 0x215274
WARNING: SPI cycle not done
ERROR: SPI flash write cycle failed
[spi] writing chunk 2 of 0x4 bytes to 0x215278
WARNING: SPI cycle not done
ERROR: SPI flash write cycle failed
[spi] writing chunk 3 of 0x4 bytes to 0x21527C
WARNING: SPI cycle not done
ERROR: SPI flash write cycle failed
WARNING: SPI flash write returned error \
(turn on VERBOSE)
[CHIPSEC] (spi write) time elapsed 0.772
```

図 5.10　PMM 上での `chipsec_util spi write` の実行結果の抜粋

```
# chipsec_util spi disable-wp
[...]
[CHIPSEC] trying to disable BIOS write protection..
[+] BIOS region write protection is disabled in \
SPI flash
[CHIPSEC] (spi disable-wp) time elapsed 0.000
```

図 5.11　物理マシン上での `chipsec_util spi disable-wp` の実行結果

```
# chipsec_util spi disable-wp
[...]
[CHIPSEC] trying to disable BIOS write protection..
[-] couldn't disable BIOS region write protection \
in SPI flash
[CHIPSEC] (spi disable-wp) time elapsed 0.000
```

図 5.12　PMM 上での `chipsec_util spi disable-wp` の実行結果

トが提供するセキュリティ機能が有効になっているか否かをチェックする．図 5.7 と 図 5.8 はこのコマンドを物理マシン上で実行した結果と PMM 上で実行したものをそれぞれ示している．このコマンドは，第 5.4.1 節 で示した attack interface の中から 3 つの項目についてチェックをしている．1 つ目の `SPI protected ranges` は SPI Range Protection において，少なくとも 1 つの保護領域が BIOS 領域を含んでいるか否かをチェックしている．2 つ目の `chipsec.modules.common.bios_wp` はチップセットの BIOS 書き込み保護に関するチェックを行っている．このチェックでは，BIOS Write Enable が無効になっているか否かと BIOS Write Protect が有効になっているかを調べている．3 つ目の `chipsec.modules.common.spi_lock` は SPI configuration lock down が有効になっているか否かをチェックしている．これが有効になっていれば，1 つ目の SPI range protection の設定が OS から変更されることはなくなる．図 5.7 と 図 5.8 の結果から，BIOS はこれらの機能を有効にしておらず，PMM はこれらの機能を有効にしていることがわかる．

```
~# ethtool -e enp3s0 offset 0 length 6
Offset          Values
------          ------
0x0000:         00 1b 21 53 84 3f
~# ethtool -E enp3s0 magic 0x10d38086 value 0x11 \
offset 0x0
~# ethtool -e enp3s0 offset 0 length 6
Offset          Values
------          ------
0x0000:         11 1b 21 53 84 3f
~#
```

図 5.13　物理マシン上での `ethtool` による Intel 製 NIC の NVM への書き込み処理の結果

```
~# ethtool -e enp3s0 offset 0 length 6
Offset          Values
------          ------
0x0000:         00 1b 21 53 84 3f
~# ethtool -E enp3s0 magic 0x10d38086 value 0x11 \
offset 0x0
Cannot set EEPROM data: Operation not permitted
~# ethtool -e enp3s0 offset 0 length 6
Offset          Values
------          ------
0x0000:         00 1b 21 53 84 3f
~#
```

図 5.14　PMM 上での `ethtool` による Intel 製 NIC の NVM への書き込み処理の結果

CHIPSEC の `chipsec_util spi write` コマンドは SPI インターフェイスを介して BIOS ROM へ書き込みを行うコマンドである．図 5.9 と 図 5.10 はこのコマンドを PMM なしの物理マシン上と PMM ありのマシン上で実行した結果である．図中のコマンドは BIOS ROM のアドレス 0x215270 に `data.bin` 内の 16 バイトのデータを書き込むものである．今回実験で用いたマシンでは，このアドレスには UEFI ブートする際に用いる UEFI ブートエントリに関する情報が格納されている．コマンドを実行した結果，PMM なしの状態では BIOS ROM に対する書き込みが成功した．一方で，PMM ありの状態では，コマンドによる書き込みは失敗した (“ERROR: SPI flash write cycle failed” というエラーメッセージが確認できる)．PMM なしの状態では書き込みに成功するものの，意図した通りのデータは書き込めていない．具体的には，BIOS ROM 内の値は変更されているものの，意図したデータとは異なる値になっている．意図した通りに書き込むには，チェックサムや署名などを考慮する必要がある．

CHIPSEC の `chipsec_util spi disable-wp` コマンドはチップセットが提供している BIOS Write protection の機能の無効化を行うコマンドである．このコマンドを PMM なしの状態と PMM ありの状態で実行した結果がそれぞれ 図 5.11 であり， 図 5.12 PMM なしでは Write Protection 無効化が可能であり，PMM ありでは無効化できなくなっていることがわかる．

ハードウェアによる保護機構がない Intel NIC においても PMM による保護が有効であるか否かを明らかにするために, `ethtool` というコマンドを用いて Intel NIC の不揮発領域を書き換える実験を行った．`ethtool` はイーサネットデバイスの不揮発領域を読み書きするための機能があり，本実験ではこの機能を用いる．今回使用した Intel NIC では，不揮発領域として EEPROM を使用しており， `ethtool` が EEPROM に書き込みを行う際は Intel NIC の EEPROM Write レジスタを用いる．図 5.13 と 図 5.14 はそれぞれ ethtool で不揮発領

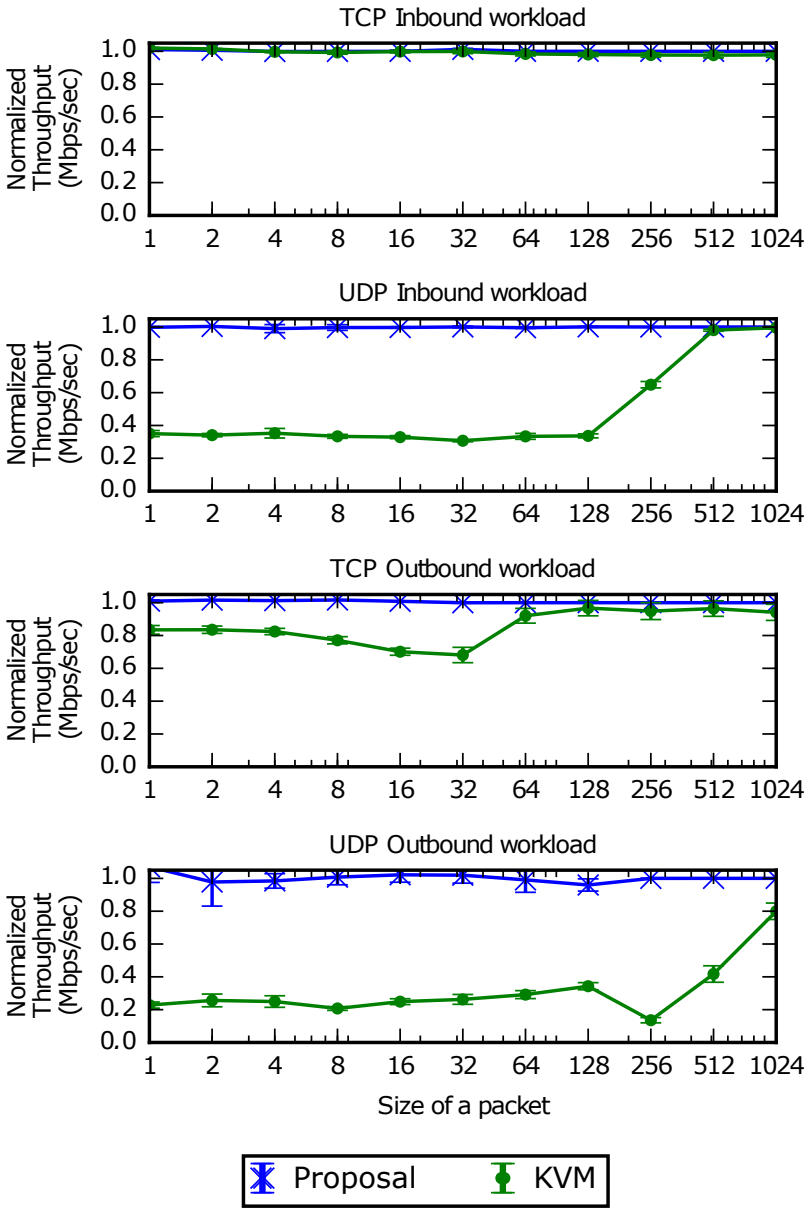

図 5.15　ネットワークスループットの測定結果

域へ値を書き込むコマンドを PMM なしと PMM ありの状態で実行した結果である．図中の 3 つのコマンドはそれぞれ，変更前の EEPROM の値を表示，EEPROM への書き込み，および書き込みコマンド実行後の EEPROM の値を表示，という処理を行うものである．結果から，PMM なしでは EEPROM の書き換えが成功しており，PMM ありでは EEPROM への書き込みが失敗していることがわかる．

5.5.3　ネットワーク性能

PMM によるハードウェアを保護処理がシステムの性能へどの程度影響するかを明らかにするために，PMM なしの物理マシン，PMM ありの物理マシン，および KVM においてネットワーク性能を測定し比較した．ネットワーク性能を測定するにあたり用いた対向マシ

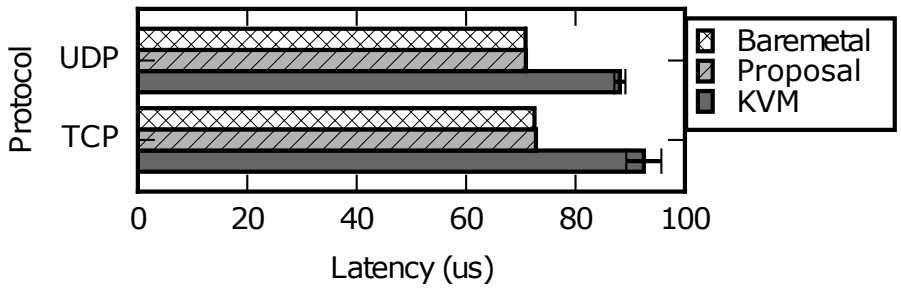

図 5.16　ネットワークレイテンシの測定結果

ンは，Intel Xeon E3 CPU，10 GbE NIC で構成されており，評価対象のマシンとはスイッチなどを介さず LAN ケーブル直結の状態である．KVM のネットワークデバイスは準仮想化デバイスの virtio-net であり，vhost-net が有効になっている．性能評価ではネットワークベンチマークツールである netperf を用いて，TCP と UDP におけるスループットとレイテンシ測定した．スループットにおいては，送信スループットと受信スループット共に測定した．全てのシステムにおいてネットワーク性能を最大化するために，評価対象のマシンにおいて netperf サーバープロセスとネットワークからの割り込みを処理するコアは別々のコアとなるように設定した．

図 5.15 はネットワークスループットの測定結果を示している．グラフの横軸は負荷を掛ける際のパケットサイズ，縦軸はスループットを示している．またグラフのスループットはベースラインとなる PMM なしの物理マシンでの性能を 1 として正規化している．このグラフは数値が大きいほど良い性能であることを示している．このスループットの測定において，5 回のウォーミングアップの後に，10 回の測定を行い，その平均と標準偏差 (エラーバー) に示している．

TCP の受信ワークロードにおいて，3 つのシステムの間に大きな性能差はなかった．これは TCP スタック小さなパケットを送出する際に， Nagle アルゴリズム [74] に従って小さなパケットを一つのパケットにマージするため，実際に物理デバイスが扱うパケットのサイズは大きく，パケットの数は減少するためである．一方 UDP の受信ワークロードでは，全てのパケットサイズにおいて PMM は 1% 未満の性能劣化であるのに対し KVM では 256 バイト以下のパケットサイズにおいて 65%–70% 程度の性能劣化が発生した．この理由は，TCP のように小さなパケットがマージされずパケットの数が多いワークロードになったことで，受信側のシステムはデバイスから大量の割り込みを受信し，割り込み処理におけるオーバヘッドが顕在化したためと考えられる．

TCP の送信ワークロードにおいて，PMM は 1% 未満のオーバヘッドであった．一方 KVM は 32 バイト以下のワークロードにおいては 16–32% 程度のオーバヘッドがあった．また，パケットサイズが 32 バイト以上であっても，KVM では 依然として 3.3–8.0% 程度のオーバヘッドがあった．TCP パケットの送信において TCP パケットの受信以上に性能差がでる理由として考えられるのは，パケットを送信している評価対象マシンにおいても Nagle のアルゴリズムに沿ってパケットをマージしているものの，依然としてパケットの数が多いためだと考えられる．さらに，パケットの送信処理においては，NIC がパケットを送信するたびに割り込みが発生し，OS はこの割り込みを処理する必要がある．これらのことから，通信量に対する割り込み処理などのパケット処理の回数が増加し，これらの処理におけるオーバヘッドが性能差として現れたと考えられる．

表 5.1　netperf レイテンシ測定ワークロードにおける PMM と KVM による 1 秒間の平均 VM exit 回数

Exit Reason	Proposal	KVM
PAUSE instruction	-	50748.0
Write to MSR	-	39854.7
External interrupt	-	33094.5
I/O instruction	-	10473.5
EPT Violation	28239.3	-
CPUID instruction	32.0	-
Exception or NMI	2.4	-
VMCALL	2.0	-
Total	28275.7	134170.3
Total (excluding PAUSE)	28275.7	83422.3

UDP の送信ワークロードの性能測定の結果は，UDP の受信ワークロードと似た傾向を示している．PMM は 5% 未満の性能劣化なのに対し，KVM では 512 バイト以下のパケットについて 58–86% の性能劣化が確認された．

図 4.17 は上記 3 つシステムにおいて UDP と TCP のネットワークレイテンシを測定した結果である．この測定では，パケットサイズ 1 バイトのラウンドトリップタイム (RTT) を測定した．横軸はレイテンシをマイクロ秒単位で示しており，この値は小さいほど良い値といえるものである．UDP と TCP 共に，PMM は 1% 未満の性能劣化しか確認できなかったのに対し，KVM は 24–27% の性能劣化が確認された．

PMM と KVM における性能差は I/O への介入の回数の違いに起因する．Intel NIC を介したネットワーク処理において，OS は Intel NIC とやり取りするために MMIO を発行する．このうち，PMM はその一部のみをハードウェア保護のために介入する．一方で，KVM は OS に対して仮想デバイスを見せ，仮想デバイスへの I/O を全てに介入し，これを処理し，物理デバイスに I/O を再度発行する．PMM でも 割り込みマスクレジスタと呼ばれる割り込み処理の際に書き込みが行われるレジスタへの書き込み I/O で VM exit が発生してしまう．これは，このレジスタが不揮発領域のインターフェイスとなるレジスタと同じメモリページに配置されているためである (第 5.4.2 節 参照)．にもかかわらず，性能面でのオーバヘッドは KVM のそれよりも大幅に小さくなっている．

5.5.4　VM exits 回数

VM exit の回数はシステムがゲスト OS の I/O に介入した数を意味する．この数字は性能にかかわる指標であるため，性能差が顕著であった UDP のネットワークレイテンシのワークロードにおける VM exit の数を測定した．この測定では，VM exit の回数を数えるために PMM ではソースコードの一部を改変している．また，KVM においては, `kvm_stat` というツールを用いた．

表 5.1 は 1 秒間に発生した VM exit の数をその原因別に示したものである．測定の際，VM exit の測定の前にウォームアップとして 5 回負荷を掛けた後に，UDP のネットワークレイテンシを測定したワークロードを実行し，その際の 1 秒間の VM exit を 10 回測定し

た．表に示す VM exit の回数はこの 10 回の平均値となる．表から，PMM における VM exit の数は KVM と比べて 79% 少ない (それぞれ 28275.7 回, 134170.3 回) ことがわかる．KVM はゲスト OS が CPU をアイドル状態にした際にも VM exit が発生する． この VM exit (表中 PAUSE instruction) を除いて計算しても，PMM の VM exit 回数は KVM よりも 66% 少ない (それぞれ，28275.7 回， 83422.3 回) PMM における VM exit の主な原因は EPT violation であった．これは，EPT において何らかの原因でメモリアクセスができなかった場合に発生する VM exit である．この VM exit のほとんどは割り込みマスクレジスタへの書き込み MMIO で VM exit が発生したものである．このレジスタは，不揮発領域へのインターフェイスとなるレジスタと同じメモリページに配置されているため，本来このレジスタへの MMIO を介入される必要はないのも関わらず VM exit が発生してしまう．また，今回のワークロードの中では，不揮発領域へのアクセスはなく，これによる VM exit は発生しなかった．

KVM における VM exit の “PAUSE instruction” 以外の主な原因は “External interrupt” “I/O instruction” そして “Write to MSR” であった．“External interrupt” と “I/O instruction” はデバイスの仮想化処理を行うために発生している VM exit である．一方，“Write to MSR” は IA32 TSC DEADLINE MSR レジスタと呼ばれるレジスタへのアクセスで発生したものである．このレジスタは，APIC タイマーが次回の割り込みをいつ起こすかを指定するための MSR レジスタである．これらの VM exit は第 5.5.3 節 で示した通り，性能劣化の主な原因となる．

5.6 議論

5.6.1 ページ内で共有されている MMIO レジスタ

第 5.4.2 節 で述べた通り，Intel VT-x は細粒度で MMIO レジスタへのアクセスに介入する機能を提供していない．このため，PMM は介入対象の MMIO レジスタを少なくとも 4KB ページの単位でしか指定できない．これは，もしセキュリティ上書き込みアクセスが行われても問題ないレジスタと保護のために書き込みを禁止すべきレジスタが同一ページに配置されている場合で，なおかつ，前者のレジスタに対して頻繁にアクセスされた場合，PMM による介入の性能への影響は大きくなってしまうことを意味する．一方，PIO や MSR に関しては，Intel VT-x はビットマップによるレジスタ単位での介入の設定が可能である．これは，不必要な VM exit を減らすためである．MMIO レジスタに関しても同様の機能が Intel VT-x に実装されれば，今以上に PMM の性能への影響は小さく抑えることができると考えられる．実際，Intel は今後より細粒度に EPT のアクセス権限を指定する機能を Intel VT-x に追加することを発表しており，この機能により PMM によるオーバヘッドは幾分か抑えられると考えられる．また，この問題に対するもう一つの解決策はデバイス設計者が MMIO レジスタの配置を決める際に，セキュリティ上保護される必要のあるレジスタと頻繁にアクセスされるレジスタを同一メモリページに配置されないようにすることである．これは，既に販売されているデバイスへの適応は難しいものの，これからデバイスを設計する際には容易に適応できる手法だと考えられる．

5.6.2 SR-IOV によるハードウェア保護

SR-IOV を用いることで，物理デバイスの性能を維持しつつ物理デバイスを保護する手法も考えられる．SR-IOV は PCI デバイス自身が自身のインターフェイスの一部を多重化し VM に直接制御させることで，仮想化による性能劣化を避けることができる機能である．SR-IOV を用いる場合，物理デバイスのすべての機能を提供するインターフェイスである physical function はハイパバイザが制御し，ゲスト OS は SR-IOV の機能で作成されたインターフェイスである Virtual Function をハイパバイザの介入なしに直接制御する．NIC の SR-IOV では, Virtual function もデータの送受信を行うための機能を提供しているため，ゲスト OS はハイパバイザの介入なしにデータの送受信処理に関する I/O を全て行うことができる．一方で，Virtual function は物理デバイスの不揮発領域へアクセスするためのインターフェイスは提供していない．このため，virtual function のみを提供されるゲスト OS は物理デバイスの不揮発領域に対して書き込むことができず，結果として物理デバイスの性能を維持しつつ物理デバイスを保護することができる．しかし今回，このような手法を選択しなかったのは以下のような理由からである．一つ目の理由は, SR-IOV は一部の I/O デバイスでしかサポートされておらず，チップセットなどではこのような手法は用いることができないことである．二つ目の理由はハイパバイザが SR-IOV を利用してしまうと，ゲスト OS が SR-IOV を用いることができなくなるためである．ベアメタルクラウドの一つの利点はゲスト OS が仮想マシン上では利用できない物理デバイスの機能を直接利用できる点である．実際に，SR-IOV は仮想化以外にも I/O 処理の最適化などでも利用する研究がある 今回提案した手法では，PMM は SR-IOV を用いないため，ゲスト OS が SR-IOV を用いることができる．

第 6 章

結論と今後の課題

本稿では，ベアメタルクラウドにおいて提供する物理マシンを管理性を改善するシステムの研究について述べた．ベアメタルクラウドでは，仮想マシン環境では利用可能なマシン管理機能の一部が利用できなかった．これは，仮想マシン環境における仮想マシンモニタのように，ユーザーの OS が動作中に OS から独立してマシン全体を制御するシステムが存在しなかったためである．このようなシステムをベアメタルクラウド向けに実現しようとした際，OS 非依存，小さな性能劣化，物理ハードウェアの提供という三つの要件を満たす必要があった．そこで本稿では，VMM のようにマシン管理機能を提供でき，かつ物理マシンの性能や物理ハードウェアの提供を両立できる新たなソフトウェアシステムとして物理マシンモニタ (PMM) を提案した．物理マシンモニタは，仮想マシンモニタのように OS に依存せずに動作し，一方で，デバイスを仮想化せず，ハードウェアと OS の通信をほぼパススルーすることで，性能劣化を抑えている．また本稿では実際にベアメタルクラウドで有用な管理機能であるライブマイグレーションとマシン保護機能のプロトタイプを実装しその評価を行った．評価の結果，これらマシン管理機能は OS 非依存で，かつ，ごく小さな性能劣化のみで実現できたことを確認した．

物理マシンモニタが上記三つの要件を満たすために，本研究では物理マシンモニタを準パススルーハイパバイザと呼ばれる軽量ハイパバイザを基に設計した．この設計では，物理マシンモニタは OS よりも高い特権でかつ独立して動作しつつ，OS は物理ハードウェアと直接利用して動作する．また，物理マシンモニタは仮想マシンモニタとは異なりハードウェアを仮想化せず，また OS とハードウェアの通信もほぼパススルーする．一方でハードウェアの保護を行う上で一部の通信を遮断する必要があるため，これを実現するために準パススルードライバと呼ばれる仕組みを用いた．また，物理マシンモニタは物理マシンを構成する全てのハードウェアを制御することで，マシンの管理機能を実現する設計となっている．この設計により，物理マシンは先の 3 つの要件を満たすことができた．

物理マシンモニタによるライブマイグレーション機能の実現での主な課題は物理デバイスの状態の制御にあった．その理由は，物理デバイスがソフトウェアから直接読み出しや書き込みのできない状態を持つことであった．この課題を解決する手法として，物理マシンモニタがライブマイグレーション実行時にデバイスを制御し間接的に読み出しできない状態を推定する手法と間接的に書き込みできない状態を更新し目的の状態に設定する手法を示した．またプロトタイプの実装においては，上記手法の実装に加え，物理マシンモニタでライブマイグレーションを実装する際の留意点や工夫についても述べた．プロトタイプを用いた提案手法を評価した結果，ライブマイグレーションが実際に Linux と Windows において動作していることが確認できた．また，仮想マシンでは性能劣化が見られるワークロードにおいて

も物理マシンモニタでは性能劣化が大幅に小さい，もしくはほぼ存在しないことが確認できた．ライブマイグレーションの手法についての議論では，本手法のチェックポイント機能への応用の可能性や，本手法の制約について示した．

ハードウェア保護について，本研究での脅威モデルでは，悪意を持つユーザーが物理マシンのハードウェアに対して攻撃することで，ベアメタルクラウドのサービスに対する永続的なDoS (PDoS) や攻撃したマシンを後に使ったユーザーに対するデータの破壊や盗み出しを行うことを想定した．また，物理ハードウェアへの攻撃は物理ハードウェアが持つ不揮発な設定情報やファームウェアを改変することで行うことを想定した．この脅威モデルに対して有効な物理ハードウェア保護を物理マシンモニタによって実現するために，提案した保護機構では物理マシンモニタによって 1) 物理ハードウェアが持つハードウェア保護機能を有効化，2) 設定情報やファームウェアが格納されている不揮発領域へのアクセスを遮断，を実現した．プロトタイプの実装では，不揮発領域へのアクセスの遮断の実装や物理マシンモニタ自身の保護を行った．その後，プロトタイプを用いて提案手法を評価した結果，ハードウェアの保護機構が有効になっていることと実際に不揮発領域への書き換えが失敗するようになったこと，つまり不揮発領域が保護されていることを確認した．また，性能評価の結果から，ネットワークデバイスを保護している状態でもネットワーク性能の低下はほとんどないことが確認できた．本手法に関する議論として，現在のハードウェアに対して本手法を適用する際の制約や，他のアプローチの可能性と本手法との比較について示した．

物理マシンモニタに関する今後の課題は以下の二つである．一つ目は管理機能をより多くのデバイスに対応することが挙げられる．本稿で評価に用いたプロトタイプ実装は非常に限られたデバイスにしか対応していない．このため，本手法の実用性を評価するためにはより多くのデバイスに対応し，他のデバイスでも同様の結果が得られるか否か確認する必要がある．複数のデバイスに対応するにあたり，その開発コストを抑えるアプローチとしてデバイスドライバの自動生成に関する研究成果 [75, 76, 77] を活用できる可能性が考えられる．本手法はデバイスの設計によって実装が異なる手法のため，デバイスの設計によって性能に与える影響が異なることも考えられる．その際は，どのような設計であればより性能劣化が小さいのかを明らかにする．

もう一つの課題は，実際のクラウド環境のように多数のマシンを運用している環境での評価が挙げられる．今回の実験はマシン一台や二台の環境での実験のみであり，実際のクラウドのような多数のマシンを集中管理するようなシステムとは接続していなかった．より実運用に近い環境での評価結果を得るために，これらの集中管理のシステムも交えた環境で実験を行う必要があると考えられる．また，科学計算のワークロードのように複数台のマシンによる並列分散処理における計算性能への影響についても評価する必要がある．

物理マシンモニタによるライブマイグレーションに関する今後の課題は，複数のデバイスに対応しその知見や結果から本手法を適応するためにデバイスが満たすべき要件についてより明確にすることである．デバイスが満たすべき要件を明らかにする上で，デバイスを状態機械とみなし有限オートマトンでモデル化し，どのようなモデルであれば要件を満たすのかといったアプローチも取り得ると考えられる．具体的には，読み出しできない状態の推定に関しては，与えられた有限オートマトンにおいて参照不可な状態から状態遷移を起こし，その結果辿り着く参照可能な状態から状態遷移前の状態を一意に特定できるか否かという問題に帰着できると考えられる．また，書き込みできない状態の推定に関しては，ソフトウェアから実行可能な状態遷移の組み合わせのみで初期状態から目的の状態に到達可能か否かという問題に帰着できると考えられる．デバイスのモデル化に際しても，先述のデバイスドライバ自動生成に関する研究成果 [75, 76, 77] を活用できる可能性が考えられる．

ハードウェア保護に関する今後の課題は以下の通りである．より細かい粒度での保護の実現が考えられる．不揮発領域に格納されている設定情報の中には，改変されても問題ないものがある．例えば，UEFI が持つブートエントリのみの変更は物理マシンに悪影響を及ぼさない．しかしながら，現在の保護機構では BIOS ROM への一切の書き込みを遮断するためこのような変更も遮断してしまう．このような無害な変更については利用可能な状態にしておいた方が物理マシン利用の自由度を維持できる．これを実現するためには，不揮発領域への変更を解析し変更内容が有害か否かを判定する必要がある．これに際して，どのような変更が有害であるか否かをより細かく判定するための基準を明らかにすることが求められる．

参考文献

[1] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High Performance VMM-bypass I/O in Virtual Machines," in *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006, pp. 29–42.

[2] "IBM Cloud IaaS," https://www.ibm.com/cloud/infrastructure.

[3] "Oracle Cloud." [Online]. Available: https://cloud.oracle.com/

[4] "Internap." [Online]. Available: http://www.internap.com

[5] "Rackspace." [Online]. Available: http://www.rackspace.com/

[6] "Announcing Amazon EC2 Bare Metal Instances (Preview)."

[7] "DPDK." [Online]. Available: http://dpdk.org/

[8] "Seastar." [Online]. Available: http://www.seastar-project.org

[9] "ScyllaDB." [Online]. Available: http://www.scylladb.com/

[10] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: A protected dataplane operating system for high throughput and low latency," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 49–65.

[11] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 1–16.

[12] L. Rizzo, "netmap: A novel framework for fast packet i/o," in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 101–112.

[13] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "Stackmap: Low-latency networking with the OS stack and dedicated NICs," in *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 43–56.

[14] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "SPDK: A development kit to build high performance storage applications," in *Proceedings of the 9th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2017)*, 2017, pp. 154–161.

[15] "Google Cloud Platform Blog: Google Compute Engine is now Generally Available with expanded OS support, transparent maintenance, and lower prices," http://googlecloudplatform.blogspot.com.au/2013/12/google-compute-engine-is-now-generally-available.html.

[16] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive Fault Tolerance for HPC with Xen Virtualization," in *Proceedings of the 21st Annual International Conference on Supercomputing*, 2007, pp. 23–32.

[17] A. Polze, P. Troger, and F. Salfner, "Timely Virtual Machine Migration for Proactive Fault Tolerance," in *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, 2011, pp. 234–243.

[18] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, "Proactive Fault Tolerance Using Preemptive Migration," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 252–257.

[19] X. Xu and B. Davda, "SRVM: Hypervisor support for live migration with passthrough SR-IOV network devices," in *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)*, 2016, pp. 65–77.

[20] M. A. Kozuch, M. Kaminsky, and M. P. Ryan, "Migration Without Virtualization," in *Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS 2009)*, 2009.

[21] L. Yanlin, M. M. Jonathan, and P. Adrian, "VIPER: Verifying the Integrity of PERipherals' firmware," in *Proceedings of the 18th ACM Conf. on Computer and Communications Security*, 2011, pp. 3–16.

[22] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*, 2005, pp. 273–286.

[23] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "ELI: Bare-metal performance for I/O virtualization," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 411–422.

[24] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har' El, D. Marti, and V. Zolotarov, "OSv—optimizing the operating system for virtual machines," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 61–72.

[25] U. Steinberg and B. Kauer, "Nova: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European Conference on Computer Systems*, 2010, pp. 209–222.

[26] "OpenVZ Linux Containers Wiki," http://openvz.org/.

[27] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "Nohype: Virtualized cloud infrastructure without the virtualization," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 350–361.

[28] J. C. Mogul, J. Mudigonda, J. R. Santos, and Y. Turner, "The NIC is the hypervisor: Bare-metal guests in IaaS clouds," in *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, 2013.

[29] A. M. Azab, P. Ning, and X. Zhang, "SICE: A hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011, pp. 375–388.

[30] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato,

"BitVisor: A Thin Hypervisor for Enforcing I/O Device Security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2009)*, 2009, pp. 121–130.

[31] "BitVisor: A Single-VM Lightweight Hypervisor," http://www.bitvisor.org/.

[32] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy Live Migration of Virtual Machines," *SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 14–26, 2009.

[33] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast Transparent Migration for Virtual Machines," in *Proceedings of the USENIX Annual Technical Conference*, 2005, pp. 25–25.

[34] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, 2007, pp. 225–230.

[35] Y. Omote, T. Shinagawa, and K. Kato, "Improving Agility and Elasticity in Bare-metal Cloud," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015)*, 2015, pp. 145–159.

[36] W. Huang, J. Liu, M. Koop, B. Abali, and D. Panda, "Nomad: Migrating OS-bypass Networks in Virtual Machines," in *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE 2007)*, 2007, pp. 158–168.

[37] A. Kadav and M. M. Swift, "Live Migration of Direct-access Devices," *SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 95–104, 2009.

[38] Z. Pan, Y. Dong, Y. Chen, L. Zhang, and Z. Zhang, "CompSC: Live migration with pass-through devices," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE 2012)*, 2012, pp. 109–120.

[39] J. G. Hansen and E. Jul, "Self-migration of Operating Systems," in *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.

[40] A. Mirkin, A. Kuznetsov, and K. Kolyshkin, "Containers checkpointing and live migration," in *Proceedings of the Linux Symposium*, 2008, pp. 85–92.

[41] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software: Practice and Experience*, vol. 21, no. 8, pp. 757–785, 1991.

[42] A. Barak and O. La'adan, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing," *Journal of Future Generation Computer Systems*, vol. 13, no. 4 – 5, pp. 361 – 372, 1998.

[43] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, 2002, pp. 361–376.

[44] "akopytov/sysbench: Scriptable database and system performance benchmark." [Online]. Available: https://github.com/akopytov/sysbench

[45] "The Netperf Homepage," http://www.netperf.org/netperf.

[46] "Redis." [Online]. Available: http://redis.io/

[47] "MySQL." [Online]. Available: https://www.mysql.com/

[48] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium*

on Cloud Computing, 2010, pp. 143–154.

[49] Y. Cho, J.-B. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices." in *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[50] J. Im, J. Kim, J. Kim, S. Jin, and S. Maeng, "On-demand virtualization for live migration in bare metal cloud," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 378–389.

[51] "Red Hat Bugzilla – Bug 459202 EEPROM/NVM of the e1000e becomes corrupted." [Online]. Available: https://bugzilla.redhat.com/show_bug.cgi?id=459202

[52] "Serious e1000e driver issue in SLE 11 beta 1 and openSUSE 11.1 beta 1." [Online]. Available: https://news.opensuse.org/2008/09/22/serious-e1000e-driver-issue-in-sle-11-beta-1-and-opensuse-111-beta-1/

[53] "Status of the e1000e issue." [Online]. Available: https://news.opensuse.org/2008/10/03/status-of-the-e1000e-issue/

[54] G. Delugrè, "How to develop a rootkit for Broadcom NetExtreme network cards," in *Recon*, 2011. [Online]. Available: http://esec-lab.sogeti.com/static/publications/11-recon-nicreverse_slides.pdf

[55] Intel Corporation, "Hacking Team's "Bad BIOS": A commercial rootkit for UEFI firmware?" Tech. Rep., 2015. [Online]. Available: http://www.intelsecurity.com/advanced-threat-research/ht_uefi_rootkit.html_7142015.html

[56] "DEITYBOUNCE." [Online]. Available: https://www.eff.org/files/2014/01/06/20131230-appelbaum-nsa_ant_catalog.pdf

[57] "Comment on der spiegel article regarding NSA TAo organization." [Online]. Available: http://en.community.dell.com/dell-blogs/direct2dell/b/direct2dell/archive/2013/12/30/comment-on-der-spiegel-article-regarding-nsa-tao-organization

[58] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas, "Implementation and implications of a stealth hard-drive backdoor," in *Proceedings of the 29th Annual Computer Security Applications Conf.*, 2013, pp. 279–288.

[59] "CHIPSEC: Platform security assessment framework." [Online]. Available: https://github.com/chipsec/chipsec

[60] C. Kallenberg and R. Wojtczuk, "Speed racer: Exploiting an intel flash protection race condition," January 2015. [Online]. Available: https://bromiumlabs.files.wordpress.com/2015/01/speed_racer_whitepaper.pdf

[61] L. Duflot, Y.-A. Perez, and B. Morin, "What if you can't trust your network card?" in *Proceedings of the 14th International Conf. on Recent Advances in Intrusion Detection*, 2011, pp. 378–397.

[62] F. Zhang, H. Wang, K. Leach, and A. Stavrou, "A framework to secure peripherals at runtime," in *Proceedings of the 19th European Symp. on Research in Computer Security*, 2014, pp. 219–238.

[63] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001, pp. 133–138.

[64] K. Kourai and S. Chiba, "HyperSpector: Virtual distributed monitoring environ-

ments for secure intrusion detection," in *Proceedings of the 1st ACM/USENIX International Conf. on Virtual Execution Environments*, 2005, pp. 197–207.

[65] K. Asrigo, L. Litty, and D. Lie, "Using VMM-based sensors to monitor honeypots," in *Proceedings of the 2nd International Conf. on Virtual Execution Environments*, 2006, pp. 13–23.

[66] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conf. on Virtual Execution Environments*, 2008, pp. 71–80.

[67] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based hidden process detection and identification using lycosid," in *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conf. on Virtual Execution Environments*, 2008, pp. 91–100.

[68] Y. Chubachi, T. Shinagawa, and K. Kato, "Hypervisor-based prevention of persistent rootkits," in *Proceedings of the 2010 ACM Symp. on Applied Computing*, 2010, pp. 214–220.

[69] D. Kirat, G. Vigna, and C. Kruegel, "BareBox: Efficient malware analysis on bare-metal," in *Proceedings of the 27th Annual Computer Security Applications Conf.*, 2011, pp. 403–412.

[70] Y. Bulygin and D. Samyde, "Chipset based approach to detect virtualization malware," *BlackHat Briefings USA*, 2008.

[71] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "GPU-assisted malware," in *Proceedings of the 5th International Conf. on Malicious and Unwanted Software*, 2010, pp. 1–6.

[72] "Intelő X99 Chipset Platform Controller Hub Datasheet." [Online]. Available: https://www.intel.com/content/www/us/en/chipsets/x99-chipset-pch-datasheet.html

[73] "Intelő 82574 gigabit ethernet controller family: Datasheet." [Online]. Available: https://www.intel.com/content/www/us/en/embedded/products/networking/82574l-gbe-controller-datasheet.html

[74] J. Nagle, "Congestion Control in IP/TCP Internetworks„" Internet Requests for Comments, RFC Editor, RFC 896, January 1984. [Online]. Available: https://www.rfc-editor.org/rfc/rfc896.txt

[75] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser, "Automatic device driver synthesis with termite," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 73–86.

[76] V. Chipounov and G. Candea, "Reverse engineering of binary device drivers with revnic," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10, 2010, pp. 167–180.

[77] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij, "User-guided device driver synthesis," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 661–676.

謝辞

本研究を行うにあたり、筑波大学システム情報工学研究科教授・加藤和彦先生には手厚いご指導とご支援をいただきましたこと心より感謝申し上げます．東京大学情報基盤センター准教授・品川高廣先生には研究方針や論文執筆について数多のご指導を頂いたこと，また研究テーマや技術的な問題について幾度となく議論させていただいたこと，深く感謝いたします先生からのご指導や先生との議論がなければ，本研究の成果を得ることはできませんでした．お忙しい中，本論文の副査を引き受けて頂きました筑波大学システム情報工学研究科教授・天笠俊之先生、同教授・和田耕一先生、同准教授・阿部洋丈先生、同准教授・大山恵弘先生に深謝いたします．また，日頃より研究室でお世話になりました筑波大学大学院システム情報工学研究科准教授・長谷部浩二先生、同准教授・岡瑞起先生に感謝の意を表します．

本稿のライブマイグレーションに関する研究は，科学技術振興機構 (JST) の研究成果最適展開支援プログラム (A-STEP) ハイリスク挑戦タイプ，研究課題「高セキュリティ・高信頼度のクラウドコンピューティング環境実現に向けた基盤システムソフトウェア『BitVisor』の研究開発」によるものです．本研究課題において企業側の責任者としてご尽力いただいた公立はこだて未来大学システム情報科学部准教授 (当時，株式会社イーゲル) の松原克弥先生に深く感謝いたします．

BitVisor の開発者として BitVisor の改良にご尽力いただき，また本研究を進める上で数多くの技術的助言を与えていただきました株式会社イーゲルの榮樂英樹氏に深謝いたします。研究室の先輩として研究方針に関するご助言や技術的な知識と知恵を与えて頂きました筑波大学加藤研究室の表祐志氏に深く感謝いたします．氏の与えてくださった知識と知恵がなければ，本研究の遂行は不可能でありました．研究室の中でも特に研究や技術的な話題で議論させていただいた筑波大学加藤研究室の竹腰開氏，東京大学品川研究室の味曽野雅史氏に深く感謝いたします．彼らとの議論は研究に関するアイデアの源となる素晴らしいものでした．また，日頃より研究生活における様々な面でご協力頂いた加藤研究室の皆様と品川研究室の皆様に感謝の意を表します．

在職中の博士論文執筆に理解を示していただいた日本アイ・ビー・エム株式会社に感謝の意を表します．最後に，博士課程進学に理解を示していただき，研究活動を応援してくれました両親に感謝いたします．

付録 A

公表論文リスト

査読付き雑誌論文

1. Takaaki Fukai, Takahiro Shinagawa, Kazuhiko Kato, "Live Migration in Bare-metal Clouds," IEEE Transactions on Cloud Computing (TCC), (Accepted for publication).

査読付き国際会議論文

1. Takaaki Fukai, Satoru Takekoshi, Kohei Azuma, Takahiro Shinagawa, Kazuhiko Kato, "BMCArmor: A Hardware Protection Scheme for Bare-metal Clouds," in Proceedings of the 9th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2017), pp. 322–330, 2017.
2. Takaaki Fukai, Yushi Omote, Takahiro Shinagawa, Kazuhiko Kato, "OS-Independent Live Migration Scheme for Bare-metal Clouds," in Proceedings of the 8th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2015), pp. 80–89, 2015. (Best Paper Award).

査読なし国内研究会発表

1. 深井 貴明，表 祐志，品川 高廣，加藤 和彦，「物理マシン間のライブマイグレーション手法の提案」，第 127 回 システムソフトウェアとオペレーティング・システム研究会，情報処理学会研究報告，第 2013-OS-127 巻，13 号，1–7 ページ，2013 年.

ポスター発表

1. Takaaki Fukai, Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato, "Live Migration of Bare-metal Instances," 5th Asia-Pacific Workshop on Systems, 2014.
2. 深井 貴明，表 祐志，品川 高廣，加藤 和彦，「物理マシン間のライブマイグレーション手法の提案」，第 11 回 ディペンダブルシステムワークショップ，2013 年.
3. 深井 貴明，表 祐志，品川 高廣，加藤 和彦，「物理マシン間のライブマイグレーション手法の提案」，第 25 回コンピュータシステム・シンポジウム，2013 年.