ハイパーバイザ開発のためのフレームワーク

江頭 宏亮[1,a)]　福田 浩章[1,b)]

概要：現在，仮想化は一般的な技術になっており，物理資源の効率化が求められるデータセンタなどで活用されている．こうした仮想化を支えているのがハイパーバイザである．ハイパーバイザとは，仮想マシンを実現するためのソフトウェアである．近年，ハイパーバイザに負荷分散やマルウェア監視などの新たな機能を加え，より効率性と可用性の高いハイパーバイザが研究・開発されている．新しいハイパーバイザをゼロから開発する場合，仮想ハードウェアなどの多くの機能を実装しなければならないため，豊富な知識と多くの時間が必要となる．また，オープンソースのハイパーバイザに新たな機能を実装しようとしても，それらのハイパーバイザは長い時間をかけて多くの開発者によって改修されてきたため難しい．

そこで本稿では，ハイパーバイザの開発を容易にするためのフレームワークを提案する．提案フレームワークは，仮想ハードウェアを提供しハイパーバイザの制御を容易に変更可能にすることで，開発者の負担を軽減を目指す.

キーワード：ハイパーバイザ，フレームワーク，仮想化支援技術

A Framework for Supporting Hypervisor Development

Hiroaki Egashira[1,a)]　Hiroaki Fukuda[1,b)]

Abstract: A hypervisor is used in data centers because it allows multiple operation systems to share a single hardware. New hypervisor which has an additional functionality, has received much attention in recent years. New hypervisors which have a functionality of malware detection and load balancer, have been researched and developed. However, a developing of hypervisor is very difficult because it requires many implementations of functions. Additionally, an addition of new function to an existing hypervisor which is open(e.g. xen, VirtualBox and so on), is also very difficult because the existing hypervisors are very complicated. In this paper, we offer a framework for supporting hypervisor development which aims at developing a hypervisor easily without much knowledge and any implementations in kernel land.

Keywords: hypervisor, framework, virtualization technology

1. はじめに

現在，仮想化は一般的な技術になっている．仮想化技術を用いることで一台の物理マシンで複数の仮想マシン（VM）を実行できるため，物理資源の効率化が求められるデータセンタなどで活用されている．

一台の物理マシンで Linux と Windows を同時に実行するとき，それぞれの OS が１つの物理メモリを書き換えてしまうためそれぞれの動作に矛盾が生じる．そのため，一台の物理マシンで複数のオペレーティングシステム（OS）を同時に実行するには，複数の OS が物理資源を使用できるよう，物理資源の管理を行う必要がある．その役割を担うのがハイパーバイザである．ハイパーバイザは，１つの OS に１つの仮想ハードウェアを提供し，あたかも複数のハードウェアが存在するかのように振る舞う．例えば，ハイパーバイザは OS のメモリアクセスをフックし，それぞれの OS のために用意した仮想物理メモリへアクセスする．こうしたハイパーバイザによる物理資源のエミュレーショ

[1] 芝浦工業大学
Shibaura Institute of Technology
[a)] ma16018@shibaura-it.ac.jp
[b)] hiroaki@shibaura-it.ac.jp

ⓒ 2016 Information Processing Society of Japan

ンは，仮想物理メモリだけでなく仮想 CPU や仮想割り込みコントローラ，仮想プログラマブルタイマなどに対しても同様に必要となる．

近年，既存のハイパーバイザにシステムの負荷分散を行う機能 [1] や OS のマルウェア感染を検出する機能 [2][3] を加えた新たなハイパーバイザが研究・開発されている．しかし，ハイパーバイザをゼロから開発することは，ハードウェアに関する豊富な知識と多くの実装が必要となるため容易ではない．また，Xen[4] や KVM[5] などの既存のオープンソースのハイパーバイザに新たな機能を加えることで，目的のハイパーバイザを開発することも論理的には可能である．しかし，それらのハイパーバイザは，長い時間をかけて多くの開発者によって改修されてきたため，ソースコードは 70 万行以上あり，構造が非常に複雑である．そのため，ソースコードを理解して機能を加えることは極めて難しい．また，ハイパーバイザを開発するためにはカーネル空間で動作するプログラムとユーザ空間で動作するプログラムの両方を実装しなければならないため，開発者にとって大きな負担となる．

そこで，本稿ではハイパーバイザの開発を容易に行うためのフレームワークを提案する．提案フレームワークでは，3 つの機能を提供する．まず，ハイパーバイザの目的に依存しない一般的な仮想ハードウェアを提供する．次に，ユーザ空間のプログラムだけでハイパーバイザを開発可能にする．そして，ハイパーバイザの制御を容易に変更できるようにする．

提案フレームワークの有効性を示すために，提案フレームワークを用いて FreeBSD のハイパーバイザである Bhyve[6] を Mac OS X に移植した Xhyve[7] と同等の機能を持つハイパーバイザ（MyHyvervisor）を開発した．そして，提案フレームワークのオーバーヘッドを計測するために Xhyve と MyHypervisor で Tiny Core Linux[8] を実行し，一定回数処理をフックするまでの時間を測定した．さらに，どのくらいハイパーバイザの開発が容易になるのかを調べるために，コード行数の測定した．MyHypervisor が 100 回，1000 回，10000 回数処理をフックするまでにかかった時間は，Xhyve と比較して大きな差はなく，提案フレームワークのオーバーヘッドは少ないことがわかった．また，MyHypervisor は，1822 行で実装することができた．

本稿は，以下のように構成される．2 章でハイパーバイザの概要について述べ，3 章でハイパーバイザ開発における問題点について述べる．そして，4 章でハイパーバイザの開発を容易にするためのフレームワークを提案する．5 章で実装について述べ，6 章で実験について述べる．最後に，7 章で関連フレームワークについて述べ，8 章でまとめを述べる．

図 1　タイプ 2 ハイパーバイザのアーキテクチャ

2. ハイパーバイザの概要

ハイパーバイザは，動作方式と VM の実装方法により幾つかに分類される [9][10]．動作方式は，主にタイプ 1 ハイパーバイザとタイプ 2 ハイパーバイザの 2 つある．タイプ 2 ハイパーバイザのアーキテクチャを図 1 に示す．タイプ 2 ハイパーバイザは，物理マシンに Linux や Windows などの OS を実行し，そこでハイパーバイザを稼働させ，VM を実行する方式である．ハイパーバイザが動作する OS をホスト OS と呼び，VM で動作する OS をゲスト OS と呼ぶ．タイプ 2 ハイパーバイザは，OS がインストールされた PC に容易に導入できるため，本研究では動作方式にタイプ 2 ハイパーバイザを用いる．また，実装方法は主に動的命令変換と完全仮想化，準仮想化の 3 つある．本研究では，完全仮想化はホスト OS に変更を加えず，オーバーヘッドも少なく完全仮想化を用いる．完全仮想化とは，以下に述べる CPU の仮想化支援技術を用いる手法である．

2.1 CPU 仮想化支援技術

CPU の仮想化支援技術には，Intel VT[11] や AMD-V[12] があり，一台の物理マシンで複数の OS の同時実行をより効率的に行うことができる．本稿では，Intel VT を用いてハイパーバイザを開発することを想定する．

x86 アーキテクチャには，データや機能を予期せぬ障害やウィルスなどから保護するために，リングプロテクションと呼ばれるプロテクションモデルが導入されている．リングプロテクションのアーキテクチャを図 2 に示す．リングプロテクションは，0 から 3 までの 4 つのレベルで構成されている．リング 0 が最も特権が高く，リング 3 が最も特権が低い．一般的な OS は，カーネル空間がリング 0，ユーザ空間がリング 3 を使用している．リング間には特別なゲートがあり，低い特権から高い特権へ遷移するためには，ゲートを使用する必要がある．また，リング 0 でしか動作しない命令を特権命令と呼ぶ．

通常，一台の物理マシンで複数の OS を同時に実行する場合，OS はリング 0 で動作することを想定しているた

ⓒ 2016 Information Processing Society of Japan

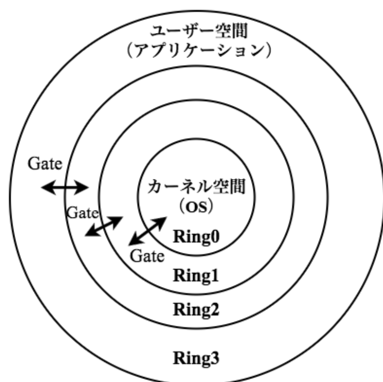

図 2 CPU のリングプロテクション

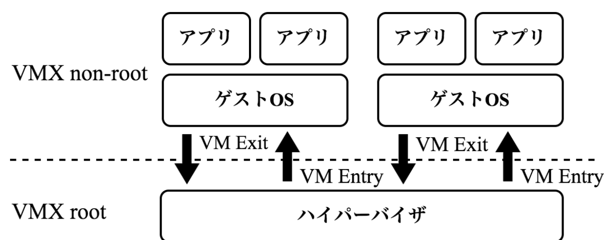

図 3 VMX のアーキテクチャ

め，それぞれ OS の動作がお互いに干渉してしまい動作に矛盾が生じる．そこで，Intel は CPU の仮想化支援技術として CPU に virtual-machine extensions(VMX) と呼ぶ新たなプロテクションモデルを追加した．VMX のアーキテクチャを図 3 に示す．VMX では，ハイパーバイザが動作するモード（VMX root）とゲスト OS が動作するモード（VMX non-root）を新たに設けている．そして，ゲスト OS がメモリなどのシステム資源の構成を変更しようとしたり，CPUID 命令などのシステム資源の構成に動作が依存している命令を実行しようとした場合に，VMX root モードに遷移し，ハイパーバイザで適切な処理を行えるようにしている．VMX root モードから VMX non-root モードへの遷移を VM Entry と呼び，VMX non-root モードから VMX root モードへの遷移を VM Exit と呼ぶ．

ハイパーバイザは，Virtual Machine Contorol Structure（VMCS）と呼ぶデータ領域を利用して VMX を制御する．VMCS は 4KB の連続したデータ領域で，CPU の個数分だけ用意する必要がある．VMCS は，主に Guest-state area, Host-state area, VM-execution control fields, VM-exit control fields, VM-entry control fields, VM-exit information fields の 5 つの領域で構成されている．ハイパーバイザの目的に応じて VMCS に適切な設定を行うことで，複数の OS を同時に実行させたときの例以外でも，VM Exit を発生させることができる．例えば，一般保護例外の発生時に VM Exit させたい場合，VM-exit control fields の exception bitmap の 13 番目のビットを 1 に設定することで，一般保護例外が発生したときに VM Exit を発生させることができる．

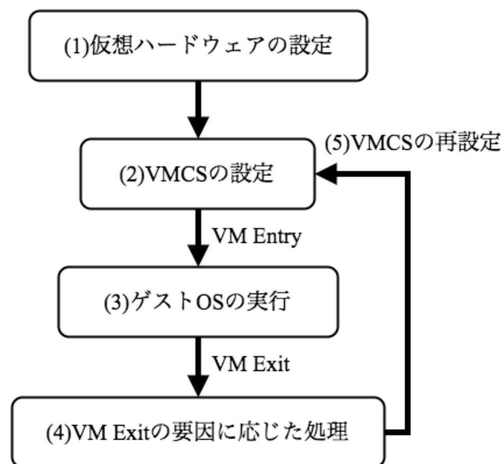

図 4 ハイパーバイザのライフサイクル

2.2 ライフサイクル

Intel VT を用いたハイパーバイザのライフサイクルを図 4 に示す．まず，VM を実行させる前に CPU の個数やメモリ容量など，(1) 仮想ハードウェアの設定を行う．次に，仮想ハードウェアの設定内容と VM Exit を発生させる要因を (2)VMCS に設定する．次に, VM Entry し, VMX non-root モードで (3) ゲスト OS を実行する．(2) で VMCS に設定した要因が発生すると VM Exit し，VMX root モードに遷移する．このとき，VM Exit の原因が VMCS の VM-exit information fields に保存される．そこで，ハイパーバイザは VM-exit information fields の (4)Exit reason に応じた適切な処理を行う．その後，必要に応じて (5)VMCS の再設定を行い，再び VM Entry して，ゲスト OS の実行を再開する．これらを繰り返すことでゲスト OS を実行する．

ハイパーバイザの振る舞いは，VMCS の設定と VM Exit 発生後の処理により決定する．例えば，一般保護例外の発生回数をカウントするハイパーバイザの開発を考える．まず，一般例外が起きたときに VM Exit が発生するように VMCS 設定を行い，VM を実行する，そして，一般保護例外が発生すると VMX non-root モードから，VMX root モードに遷移する．ハイパーバイザではカウントをインクリメントする処理を行ったあと，再び VM Entry して VM を実行する．このように，VMCS の設定と VM Exit 発生後の処理を変更することで目的のハイパーバイザを実現することが可能である．

VMCS に適切な設定を行うことで，様々な VM の実行状態を取得できるため，一台の物理マシンで複数の OS を実行させるだけでなく，マルウェアの解析などにも利用されている．

3. ハイパーバイザ開発における問題点

ハイパーバイザの開発における問題点は，非常に多くの機能を実装しなければならないことである．ハイパーバイザを開発する場合，ゼロから開発する方法とオープンソー

ⓒ 2016 Information Processing Society of Japan

スのハイパーバイザに修正を加えることで目的のハイパーバイザを開発する方法の 2 つが考えられる．

ゼロから開発する場合，仮想ハードウェアの実装とカーネル空間で動作するプログラムの開発が問題となる．仮想マシン（VM）を実現するためには，割り込みコントローラやプログラマブルタイマなどを含む，多くのハードウェアをエミュレーションしなければならない．そのため，それらのハードウェアに関する豊富な知識と多くの実装が必要になる．また，VMX に関する命令は，リング 0 で動作するためカーネル空間で動作するプログラムを実装しなければならない．カーネル空間で動作するプログラムはリング 0 で動作するため，CPU が提供する保護機能を利用することが出来ない．カーネル空間で動作するプログラムに不具合があった場合，その影響は OS にまで波及し，OS の動作に影響を及ぼす．そのため，カーネル空間で動作するプログラムの開発は容易ではない．

一方，Xen や KVM といったオープンソースのハイパーバイザに修正を加えることで目的のハイパーバイザを実現することもできる．この方法は仮想ハードウェアなどの機能を新たに実装する必要がないため，ゼロから開発する場合に比べて実装しなければならない機能は少ない．しかし，それらのハイパーバイザは，ソースコードの量が 70 万行を超え，構造が複雑であるため理解することは容易ではない．

ハイパーバイザの開発を支援するフレームワークには，Apple 社が提供している Hypervisor Framework[13] がある．Hypervisor Framework は，OS X Yosemite 以降で動作し，カーネル空間で動作するプログラムの実装を行わずに，ユーザ空間で動作するプログラムの実装だけでハイパーバイザを開発を可能にしている．したがって，Hypervisor Framework の提供する API を用いることで，カーネル空間で動作するプログラムを実装せずに，CPU の仮想化支援機能へのアクセスすることができる．しかし，仮想ハードウェアの提供はないため，開発者は依然として多くの機能を実装しなければならない．

4. 提案フレームワーク

本研究では，ハイパーバイザの開発を容易にすることを目的としたフレームワークを提案する．ハイパーバイザのライフサイクルに着目し，Application Programming Interface(API) を提供することでハイパーバイザの振る舞いを容易に変更できるようにする．提案フレームワークでは，以下の 3 つの機能を提供する．

(1) ハイパーバイザの目的に依存しない一般的な仮想ハードウェアを提供する
(2) ユーザ空間で動作するプログラムだけでハイパーバイザの開発を可能にする
(3) フック処理とエミュレーションを定義可能とし，ハイパーバイザの制御を容易に変更可能にする．

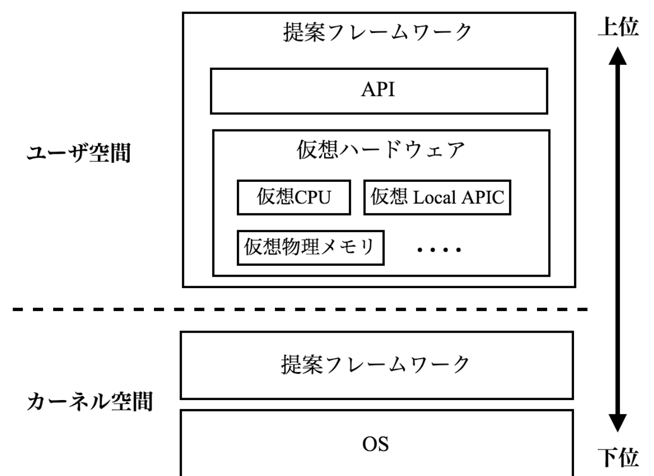

図 5 提案フレームワークのアーキテクチャ

4.1 仮想ハードウェアの提供

提案フレームワークのアーキテクチャを図 6 に示す．提案フレームワークでは，ハイパーバイザの目的に依存しない一般的な仮想ハードウェアとそれらにアクセスするための API を提供する．提供する仮想ハードウェアは，主に仮想 CPU と仮想 Local APIC，仮想物理メモリの 3 つである．それらの仮想ハードウェアは，VM Exit 時にエミュレーションを行うために必要である．

4.2 特権命令のラップ

図 6 に示したように，提案フレームワークはカーネル空間のプログラムとユーザ空間のプログラムで構成される．ユーザ空間のプログラムは，仮想ハードウェアと仮想ハードウェアへアクセスするための API の提供を行う．一方で，カーネル空間のプログラムはリング 0 から実行しなければならない VMX に関する命令の実行を行う．VMCS の読み書きを行う VMREAD 命令と VMWRITE 命令は，VMCS の設定時に用いるのでユーザ空間からそれらの命令を実行する API を提供する．その他の VMX に関する命令は提案フレームワークの内部で必要なときに実行されるので，開発者が実行することはない．したがって，それらの命令をユーザ空間から実行するための API は提供しない．

4.3 ハイパーバイザの制御の変更方法

2.2 節で述べたように，ハイパーバイザの制御は VMCS の設定と VM Exit 発生後の処理を変えることにより変更可能である．

VMCS の設定は，4.2 章で述べた API を用いて行う．また，VMCS の設定を行うには VMCS の領域を仕様 [11] で定められた番号を用いて指定する必要がある．そこで，VMCS の領域を番号で指定する煩雑さを解消するために，VMCS の領域をその名前で指定可能にする．例えば，ゲ

ⓒ 2016 Information Processing Society of Japan

スト OS の ES レジスタにアクセスしたい場合，従来であれば 0x800 を指定しなければならなかった．一方，提案フレームワークでは GUEST_ES といった VMCS の領域名でアクセス可能にする．

VM Exit 発生後の処理は，一般的に VM Exit 発生時に VMCS に保存される VM Exit field の Exit reason により異なる．そこで，Exit reason ごとに実行する処理を関数で定義し，それを提案フレームワークに登録することで，VM Exit 発生後の処理を柔軟に変更可能にする．

プロセッサの形式と機能を取得するために CPUID 命令という命令がある．CPUID 命令はシステム構成に動作が依存する命令であるため，ゲスト OS がその命令を実行すると VM Exit が発生する．例として，CPUID 命令による VM Exit 発生時に CPUID 命令のエミュレーションを行いたい場合を考える．まず，開発者は CPUID 命令のエミュレーションを行う関数を実装する．CPUID 命令の実行による VM Exit 発生時に，Exit reason は 10 が得られる．そのため，実装した関数を Exit reason に 10 がセットされているときに実行するように提案フレームワークに登録する．このようにすることで，提案フレームワークは CPUID 命令により VM Exit が発生したとき，CPUID 命令のエミュレーションを実行可能になる．

4.4 提案フレームワークを用いたハイパーバイザの開発

提案フレームワークを用いたハイパーバイザの処理の流れを図 6 に示す．まず，(1) 仮想 CPU の個数や仮想物理メモリの容量といった仮想ハードウェアの設定を提案フレームワークが提供する API を通して行う．次に，(2) 開発者は VMCS の設定時と VM Exit 発生時に提案フレームワークによって呼び出される関数を定義し，それらを提案フレームワークに登録する．そして，仮想 CPU の設定時に提案フレームワークに登録した (3)VMCS の設定が呼び出される．次に，ハイパーバイザは，VM Entry し，(4) ゲスト OS を実行する．CPU は VMCS に設定した要因が発生すると VM Exit するので，(5)Exit reason に応じて (2) で登録した関数が呼び出される．提案フレームワークは，この動作を (6) ゲスト OS が終了するまで繰り返す．

5. 実装

提案フレームワークの実装を図 7 に示す．提案フレームワークは，Xhyve[7] をベースに実装した．Xhyve は，Apple の Hypervisor Framework を用いて実装されており，CPU の仮想化支援技術である Intel VT を用いて VM を実現している．

提案フレームワークは C 言語を用いて実装され，ヘッダーファイルと共有ライブラリを提供する．

Xhyve は仮想 CPU や仮想 Local APIC，仮想物理メモリといった仮想ハードウェアを実装しているので，それらの仮想ハードウェアは Xhyve に実装されているものをそのまま利用する．また，仮想 CPU と仮想物理メモリへアクセスするための API は Xhyve に実装されているものを

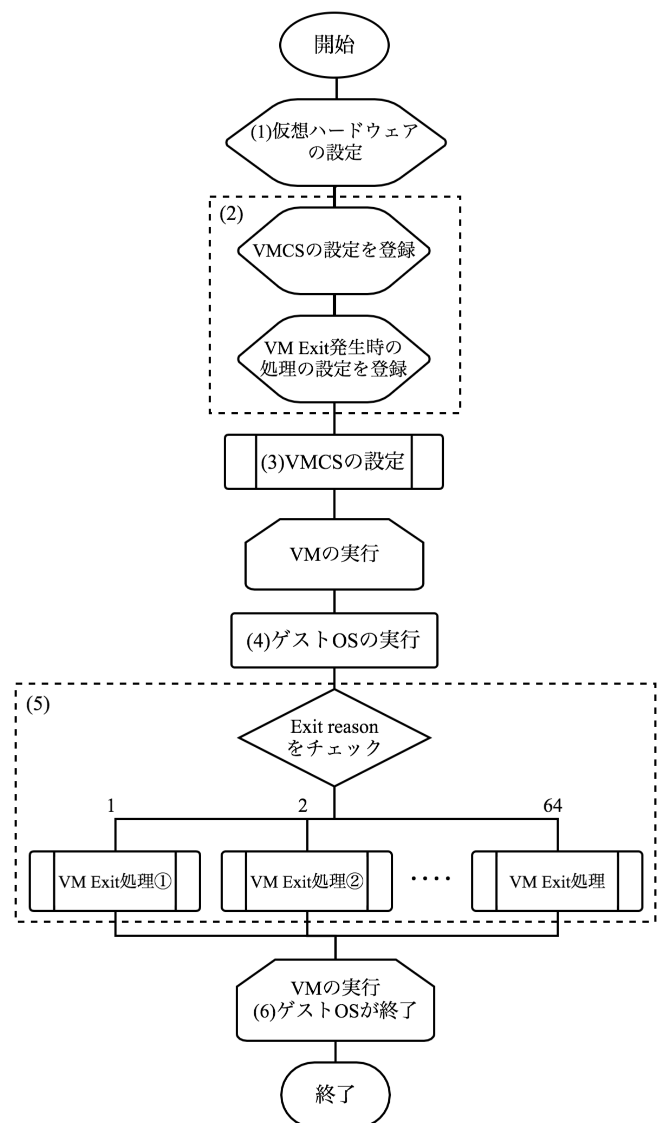

図 6 提案フレームワークを用いたハイパーバイザの処理の流れ

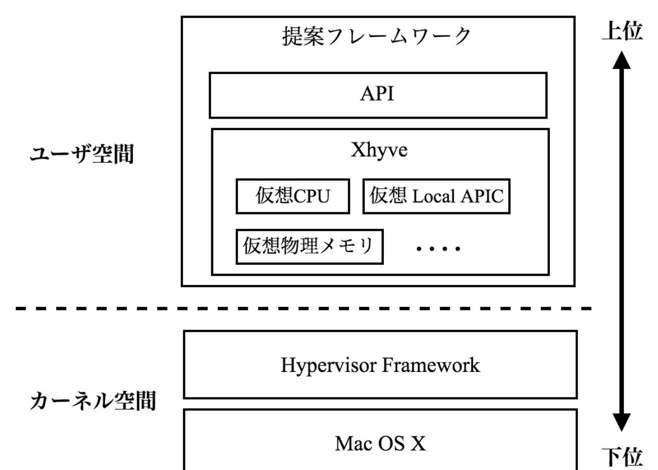

図 7 提案フレームワークの実装におけるアーキテクチャ

ⓒ 2016 Information Processing Society of Japan

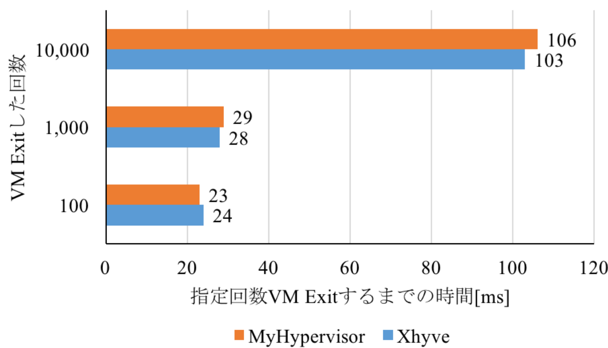

図 8 指定回数 VM Exit するまでの時間

ラップした．

VMCS へのアクセスは，VMCS の領域名と番号を紐付けた列挙体用いることで，VMCS の領域名で VMCS の各領域にアクセス可能な API を実装した．

VMCS の設定と VM Exit 時の処理をフレームワークに登録するための API を実装した．これらの API を用いることで Xhyve の適切な場所で，開発者が実装した処理が呼び出されるようになる．

6. 実験

提案フレームワークの有効性を示すために実行性能の計測とソースコード行数を測定する．実験では提案フレームワークを用いて Xhyve と同等の機能を持つハイパーバイザ（MyHypervisor）を開発した．そして，提案フレームワークが実行性能に及ぼす影響を調べるために，MyHypervisor オーバーヘッドを計測した．オーバーヘッドを計測するために，Xhyve と MyHypervisor で，Tiny Core Linux を実行し，VM Exit の回数をカウントし，100 回，1000 回，10000 回に達するまでの時間を測定した．実験は，Intel®Core^{TM}i5-3210M（2 コア，2.5GHz）の CPU，8G メモリ，512G の SSD を搭載した PC を使用した．

100 回，1000 回，10000 回 VM Exit が発生するまでの時間を図 8 に示す．100 回 VM Exit するまでの時間は MyHypervisor の方が 1ms だけ高速だった．一方で，100 回，1000 回 VM Exit するまでの回数は，Xhyve の方が僅かに高速だった．

提案フレームワークは，実行時にメモリに読み込まれてプログラムとリンクされる共有ライブラリで提供されているため，MyHyvervisor は遅くなったと考えられる．しかし，100 回 VM Exit が発生するまでの場合は，提案フレームワークの一部の API しか呼び出されていないので，影響が少なかったと考えられる．

7. 関連フレームワーク

関連フレームワークには，Hypervisor Framework がある．Hypervisor Framework は，カーネル空間のプログラムの実装を行わずにハイパーバイザを実装可能にすること目的としており，VMX に関連する命令をユーザ空間のプログラムから呼び出せる API を提供している．しかし，Hypervisor Framework は仮想ハードウェアを提供していないため，ハイパーバイザの開発を容易にするには不十分である．

8. まとめ

本稿では，ハイパーバイザの開発を容易にするためのフレームワークを提案した．仮想ハードウェアやカーネル空間で動作するプログラムをフレームワークで提供し，ハイパーバイザの制御を変更する処理を実装するだけで目的のハイパーバイザを実現可能になった．今後の課題は，マルチプラットフォームに対応させることである．

参考文献

[1] Chieu, T. C., Mohindra, A., Karve, A. A. and Segal, A.: Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment, *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on*, pp. 281–286 (2009).

[2] Jiang, X., Wang, X. and Xu, D.: Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction, *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, New York, NY, USA, ACM, pp. 128–138 (2007).

[3] Chubachi, Y., Shinagawa, T. and Kato, K.: Hypervisor-based Prevention of Persistent Rootkits, *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, New York, NY, USA, ACM, pp. 214–220 (2010).

[4] Xen (オンライン), 入手先 ⟨http://www.xenproject.org/⟩ (参照 2016-06-24).

[5] KVM (オンライン), 入手先 ⟨http://www.linux-kvm.org⟩ (参照 2016-06-24).

[6] FreeBSD: byve (オンライン), 入手先 ⟨http://bhyve.org/⟩ (参照 2016-06-24).

[7] Steil, M.: xhyve (オンライン), 入手先 ⟨https://github.com/mist64/xhyve⟩ （参照 2016-06-24）.

[8] Tiny Core Linux (オンライン), 入手先 ⟨http://tinycorelinux.net/⟩ （参照 2016-06-24）.

[9] Chisnall, D.: *The Definitive Guide to the Xen Hypervisor*, Prentice Hall (2007).

[10] 浅田拓也：ハイパーバイザの作り方～ちゃんと理解する仮想化技術～ 第 1 回 x86 アーキテクチャにおける仮想化の歴史と Intel VT-x，浅田拓也（オンライン），入手先 ⟨https://syuu1228.github.io/howto_implement_hypervisor/part1.pdf⟩ （参照 2016-06-24）.

[11] Intel Corporation: Intel 64 and IA-32 Architectures Software Developer 's Manual, Intel（オンライン）, 入手先 ⟨http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf⟩ (参照 2016-06-24).

[12] Advanced Micro Devices, Inc: AMD Virtualization (オンライン), 入手先 ⟨http://www.amd.com/en-us/solutions/servers/virtualization⟩ （参照 2016-06-24）.

ⓒ 2016 Information Processing Society of Japan

[13] Apple Inc.: The Hyeprvisor Framework（オンライン）, 入手先 ⟨https://developer.apple.com/library/mac/documentation/DriversKernelHardware/Reference/Hypervisor/⟩（参照 2016-06-24）.

ⓒ 2016 Information Processing Society of Japan