東京大学
情報理工学系研究科 創造情報学専攻
博士論文

利用可能性の高い仮想計算機転送技術

A technique of virtual machine migration with high applicability

高橋　一志
Kazushi Takahashi

指導教員　千葉　滋　教授

2013 年 2 月

要旨

既存の「遠隔仮想計算機転送システム」は「使う人と場所を選ぶ」システムであり，利用可能性が低いという問題がある．これは，仮想計算機転送システムの技術的な問題に起因する．本研究では，既存の 3 つの遠隔計算機転送システムに対して，それぞれ，3 つの着想（a）ほとんどすべてのデバイスに組み込み済みの Web ブラウザの JavaScript + HTML 上で動作するインタラクティブ性能の高いシンクライアントプロトコルの提案（b）Linux / KVM ドライバを，Linux エミュレーションレイヤを用いて「簡単」かつ「性能劣化無し」で Windows 上に移植することにより，Windows と Linux 間での VM ライブマイグレーション（c）VM の物理計算機間の往来という新たな振る舞いに着目した高速な差分転送，を元にして，既存の遠隔計算機転送システムの利用可能性を向上する．

「クラウド・コンピューティング」や「VDI : Virtual Desktop infrastructure」といった技術の普及に伴って「遠隔仮想計算機転送システム」がより重要な存在となりつつある．これらのシステムは，仮想マシンモニタ（VMM : Virtual Machine Monitor）によって生成される仮想マシン (VM : Virtual Machine) を，ネットワークを経由で，ユーザの持つ手元の端末でも利用可能にする技術である．クラウド・コンピューティングや VDI : Virtual Desktop infrastructure の場合，すべてのユーザは，中央集権的に集約された仮想マシンを，この，遠隔仮想計算機転送システムを通して，ネットワーク経由で操作することになる．そのため，このシステムは欠かすことが出来ない．そのため，遠隔仮想計算機転送システムに対して利用可能性の向上という貢献を行うのは重要な意義がある．

今日使用される遠隔仮想計算機転送システムは，画面転送によるシンクライアントシステム，VM そのものを他の物理マシンに転送する VM ライブマイグレーション，および，VM のストレージも含めて VM を他の物理マシンに転送する VM ストレージマイグレーションの 3 つであり**それぞれに利用可能性を低下させている技術的な問題点が存在する**．

まず，シンクライアントシステムの問題点は「導入が面倒」という点である．既存のシンクライアントシステムは，すべて，専用のアプライアンスや専用ソフトウェアのインストールと言う作業が必要になる．計算機に対して詳しくない人にとって，この作業は困難である．シンクライアントの研究は古くから行われているが，すべて専用のソフトウェアやアプライアンスを用いたものである．

また，既存の VM マイグレーション技術にも 2 つの問題点がある．1 つ目の原因は「コンシューマ向け OS とサーバ OS を結びつける VMM が無い」という点である．既存の VMM では，複数の異なる HostOS 上の VMM 間で，VM ライブマイグレーションを達成することができず，利用環境が同一 OS 上の VMM と限定されている．特定の OS 上でしか VM ライブマイグレーションが達成できないというのは利用可能性を低めている．また，2 つ目の原因としては，VM のストレージ転送に時間が掛かるという問題である．従来の VM ライブマイグレーションでは，この問題を解決するため，VM のディスクイメージを共有するために分散ファイルシステムや共有ファイルシステムを利用する必要があった．今までの VM ライブマイグレーションではこの制約が問題になることはなかったが，データセンタ間をまたいだ広域 WAN 環境におけるライブマイグレーションや，個人用の VM に VM ライブマイグレーショ

ンを利用するという研究も進んでおり，状況によっては，分散/共有ファイルシステムを利用するのが困難な状況下での VM ライブマイグレーションも珍しくなくなってきている．これを解決するために，VM ストレージマイグレーション技術という，VM のストレージを含めて，すべての VM ステータスを転送するという技術も開発されているが，VM のストレージは巨大であるため，転送には多くの時間がかかり，また，転送に使用する帯域幅も大きなものになる．当然，低速なネットワーク環境であれば，さらに多くの時間がかかり，これでは利用可能性が高いとは言えない．

このように，現存する遠隔仮想計算機転送システムでは利用可能性が低い．そこで，本研究では (1) JavaScript と HTML を用いたシンクライアント VoXY の実現 (2) Linux ドライバの Windows 移植による WinKVM の実装 (3) 差分転送による VM ストレージマイグレーションの高速化という 3 つの研究で，既存のシンクライアントシステム，VM ストレージマイグレーション，VM ライブマイグレーションのという 3 つの研究で，利用可能性を低下させている技術的な制約を取り除き，より多くの人が使える遠隔仮想計算機転送システムを目指す．

（1）の研究により，Web ブラウザという，ほとんどすべてのデバイスの初期段階で導入されているソフトウェア上で，新たなソフトウェアを導入することなく，使用出来る利用可能性の高いシンクライアントシステムを達成した．さらに，（2）の研究により，Windows と Linux 間といった異種 OS 間で動作する VMM 間で VM ライブマイグレーションを達成した．今までの VMM では，同一の HostOS 間で動作する同一の VMM 間でしか VM ライブマイグレーションを行うことができなかった．そのため，コンシューマ OS として広く普及する Windows や，サーバ用の OS として広く普及する Linux 間で VM ライブマイグレーションを達成することはできず，これは，利用可能性を低下させる技術的な要因であった．これを解決するため，本研究では，KVM と呼ばれる Linux ドライバとして実装されている Hybrid VMM を，Windows に移植することでこれを実現した．Windows と Linux 間でのライブマイグレーションを達成した．移植には Windows 上に Linux カーネルをエミュレーションするエミュレーションレイヤを構築することで KVM ドライバを「簡単」かつ「性能劣化」なく Windows 上で動作させることに成功した．最後の，（3）の研究では，一般に巨大な容量を持つ VM ストレージの転送時間と転送帯域を節約するために，VM の往来と呼ばれる今後 VM ライブマイグレーションに現れるであろう特質を利用した，新たな差分転送のメカニズムを提案した．これにより，従来の VM のストレージを含めた全転送時間を最大で 98% まで削減することに成功した．

この 3 つの貢献により，現在のクラウドコンピューティングや VDI といった仮想計算機の提供システムを，技術に明るくない人であっても使用させることが可能になる．普段は，VoXY を使用して Web ブラウザ上で VM を操ることができる．VM はクラウド上にあり，ユーザは電源管理や，ハードウェアの故障によるデータロストといった心配に悩まされることなくコンピュータを使うことができる．また，より詳しい人であれば，シンクライアントソフトウェアで利用するには少々困難であるソフトウェア（3D ゲーム，CAD，CAM）を利用したい場合，WinKVM と VM ストレージマイグレーションの高速化を利用して，一時的にクラウド上から VM を，手元の物理計算機に，実行コンテキストを維持したまま高速に移動させることができる．使用が終わった後は，ふたたび，安全なクラウド上に VM を移動させる．この

ように，VoXY と WinKVM と VM ストレージマイグレーションの高速という研究を組み合わせることで，より便利なコンピューティング環境をユーザに対して提供することができる

以下，3 つの研究についてより詳しく述べる．

（1）JavaScript と HTML を用いたシンクライアント VoXY では，Web ブラウザ上で動作するユーザに取って導入が容易なシンクライアントシステムを達成した．多数の OS インストールパッケージには，Web ブラウザが初期導入されているため，ほとんどすべての Web ブラウザ搭載端末で動作することが可能である．新たなアプライアンスやアプリケーションを別途インストールする必要はない．

VoXY の開発を始めた 2007 年当初，Web ブラウザ上ですべてのコンピューティングを行おうという潮流が存在し，クラウドコンピューティングというキーワードも誕生した．Web ブラウザ上で動作する，いわゆる Web アプリケーションには，ユーザは新たにソフトウェアをインストールする必要がないという利点が存在し，利用可能性の向上に大きく寄与した．この状況に倣い，Web ブラウザ上でシンクライアントシステムを実現すれば，シンクライアントシステムの利用可能性も同じく向上できると考えた．

しかし，JavaScript と HTML のみでインタラクティブ性能の高いシステムを実現するにあたっては，既存のシンクライアントシステムが採用しているプロトコルをそのまま流用することが不可能であった．なぜなら，貧弱な画像描画システムしか持たない JavaScript と HTML では，従来の，変更点のみを転送するシンクライアントシステムのプロトコルでは高いインタラクティブ性を実現することが不可能であった．もちろん，Web ブラウザに拡張性を持たせる専用のプラグインを導入することで，Web ブラウザに高機能な画像描画システムをもたせることは可能であるし，事実そのような研究も存在している．しかし，専用のプラグインは規格化されているわけではないので，限定的な環境でしか動作せず，ユーザにとって導入が容易であるとは言えない．そこで，われわれは，純粋に規格化された JavaScript と HTML を利用して，いかなる環境でもインタラクティブ性の高いシンクライアントシステムを実現する手法を提案する．具体的には，VoXY では，計算機のフレームバッファをある程度の大きさを持つタイルに区切り，タイルを単位とした転送を行うことで，この問題を解決する．従来のシンクライアントプロトコルに比べると，使用帯域は大きくなるが，HTML と JavaScript 環境が動作する Web ブラウザであれば，いかなる環境でも動作するシンクライアントシステムであるため，ユーザにとって導入が容易であり，使いやすいものである．そのため，「使う人を選ばず」「使う環境が限定されていない」遠隔仮想計算機システムを VoXY で実現することに成功する．

また (2) Linux ドライバの Windows 移植による WinKVM の実装では，Linux カーネルに付属する VMM (KVM: Kernel-based Virtual Machine) の Windows 移植手法について論じる．利用可能性の高い仮想計算機転送技術を実現するためには，様々なオペレーティングシステム上で動作する Hybrid VMM の実現が重要である．コンシューマ向けのオペレーティングシステムである Windows と，サーバ/サービス構築用として利用される Linux との間で VM ライブマイグレーションを実現することで，ユーザは，自由に自分の作業環境である VM（仮想マシン）を実行コンテキストを維持したまま移動させることができる．これにより．ユーザは，シンクライアントシステム VoXY 経由で扱うことが難しい CAD，CAM や 3D ゲームと

いったソフトウェアを使いたいときは，自分の環境を直ちに手元の計算機に転送することができる．これは，仮想マシンの利用可能性に寄与する．

しかし，VM ライブマイグレーションのプロトコルは，統一規格が存在せず，VMM 毎に独自プロトコルを用いている．そのため，現状，多くの VMM するが，それらすべての VM ライブマイグレーションの VM ライブマイグレーションプロトコルを統一するのは現実的ではない．そのため，本研究では Linux カーネルに標準搭載されている VMM である KVM を Windows 上に移植することで，Windows と Linux 間で VM ライブマイグレーションを実現する．

KVM を Windows に移植するためには Linux のデバイスドライバを Windows に移植する必要がある．KVM は特権命令である CPU の仮想化支援命令を使用しているため，KVM のコアの部分は Linux のデバイスドライバとして実装する必要があるのである．したがって，WinKVM を実現するためには，Linux デバイスドライバを Windows ドライバとして移植する必要がある．

デバイスドライバというプログラムは OS に強く依存するため，ユーザ空間で動作するプログラムに比べると移植が難しい．そのため，プログラマがドライバのソースコードを解読し，他の OS に移植するように書き換えるのは人的コストが高い．とくに問題となる (x) 発行可能な Privilege な命令の種類，(y) ユーザプログラムとのインタフェイス (z) カーネルとのインタフェイスという 3 つの OS 間の差異の 3 つである．

そこで，この研究では，WinKVM を実現するために，Windows カーネル上に Linux デバイスドライバを動作させるためのエミュレーションレイヤを手法を提案する．エミュレーションレイヤは上述した 3 つの差異を吸収し “簡単” かつ “性能劣化無し” で Windows 上で Linux ドライバである KVM ドライバを動作させることができた．しかし，この手法には制限もあり，Linux ドライバとして実装された KVM には 10 行程度の修正を加える必要がある．しかし，Linux / KVM ドライバを 10 行程度の修正でかつ性能劣化なく移植可能な本手法は，「使う人を選ばず」「使う環境が限定されていない」という限定的な遠隔仮想計算機転送技術である VM ライブマイグレーションに貢献する技術である．

最後に　(3) 差分転送による VM ストレージマイグレーションの高速化では，VM が特定の物理マシンの間を行ったり来たりする VM の往来と呼ばれる挙動に着目し，VM を含めた VM ストレージの転送を高速化の手法を提案した．従来，VM ライブマイグレーションでは，分散ファイルシステムや共有ファイルシステムといったシステムを併用し，転送先と転送元で，予め VM ストレージを共有しておくのが一般的であった．しかし，複数のデータセンタ間で VM を転送する，広域ライブマイグレーションや，個人用の VM ライブマイグレーションでは，分散ファイルシステムや共有ファイルシステムを利用することが困難となる．そこで，この問題を解決すべく，VM のストレージを含めて他の物理マシンに VM を転送する，VM ストレージマイグレーションと呼ばれる技術が提案されているが，VM のストレージは数十 GB になることも珍しくない．これほど巨大なデータを転送するには，VM が完全に転送し終わるまでに，多くの時間と帯域を要求する．そこで，本研究では，データセンタ間での広域ライブマイグレーションや，個人用の VM ライブマイグレーションでは，VM が特定の物理マシンの間を往来する可能性が高いという点に着目し，この性質を利用した，VM の差分転送メ

カニズムを開発する．

具体的には，VMM 内に DBT（Dirty Block Tracking）と，世代番号と呼ばれる概念を追加し，いつどの物理マシンで，ディスクの土の汚れたかを追跡しておき，VM ライブマイグレーション時には，差分のみを正確に取り出し，差分転送を行うメカニズムを開発する．差分転送というアイデアは，古今東西様々な場面で使用されてきた古典的な手法ではあるが，今回は，特定の物理マシン間を VM が往来するという現象に着目した差分転送を提案したことが貢献となる．ベンチマークの結果，90% 以上の転送時間を削減することができ，既存の VM ストレージマイグレーションと比べて高速な VM 転送を達成した．

以上，(1) VoXY では JavaScript + HTML で利用できる新たなシンクライアントプロトコルを開発することで，「誰でも容易に使える」Web ブラウザ上で動作するシンクライアントシステムを実現し，利用可能性を向上する．さらに，(2) WinKVM では，Linux/KVM ドライバを簡単かつ性能劣化なく Windows 上で動作する事ができるエミュレーションレイヤを開発し，使う環境を選ばない VM ライブマイグレーションを達成し，利用可能性を更に向上する，最後に (3) 差分転送による VM ストレージマイグレーションの高速化では，分散/共有ファイルシステムを使用できない環境下における VM ライブマイグレーションでは、VM が特定の物理計算機上を往来するという挙動を利用した高速な差分転送メカニズムを提案する．

この 3 つの研究によって，本博士論文で解決すべき問題である**現在の限定的である VM とユーザと結びつける遠隔計算機転送システムを，より多くの利用シーンでも利用できるような，遠隔仮想計算機転送システムを提案する**を達成した．「使う人を選ぶ」かつ「使う環境を選ぶ」遠隔仮想計算機の利用可能性を向上させたことが本博士論文の貢献である．

目次

第 1 章　序論 1
1.1 研究背景 1
1.2 本博士論文で解くべき問題点 2
1.3 解決手法 4
1.4 解決手法の実現 5
1.5 研究の問題点と達成した事柄のまとめ 7
1.6 本論文の構成 8

第 2 章　関連研究 9
2.1 仮想マシンモニタ (VMM) の歴史 9
2.2 VMM の技術的な要素 12
2.3 遠隔計算機操作システム 16
2.4 遠隔仮想計算機環境 18

第 3 章　JavaScript と HTML を用いたシンクライアント VoXY 21
3.1 研究背景 24
3.2 既存のシンクライアントプロトコルの問題点 25
3.3 新たなシンクライアントプロトコルの提案 26
3.4 VoXY : Vnc Over proXY の実装 28
3.5 評価 34
3.6 関連研究 38
3.7 JavaScript と HTML を用いたシンクライアント VoXY のまとめ 39

第 4 章　Linux ドライバの Windows 移植による WinKVM の実装 41
4.1 KVM (Kernel-based Virtual Machine) 移植の論点 43
4.2 エミュレーションレイヤを用いた KVM 移植による WinKVM の実装 54
4.3 移植性の高い Hybrid VMM を行うための指針 61
4.4 WinKVM と KVM のライブマイグレーション 62
4.5 KVM のライブマイグレーションプロトコル 62
4.6 WinKVM でのライブマイグレーション実装 65
4.7 評価 65

4.8 WinKVM の性能評価 66
4.9 ライブマイグレーションの性能評価 69
4.10 WinKVM 評価のまとめ 72
4.11 関連研究 73
4.12 Linux ドライバの Windows 移植による WinKVM の実装のまとめ 74

第 5 章 差分転送を用いた高速な VM マイグレーション 76
5.1 問題分析 78
5.2 提案手法 79
5.3 手法の詳細 81
5.4 実装 87
5.5 評価 87
5.6 関連研究 89
5.7 差分転送を用いた高速な VM マイグレーションのまとめ 91

第 6 章 結論 93
6.1 本博士論文で提案した成果 93
6.2 将来の展望 97
6.3 まとめ 97

発表文献と研究活動 99

参考文献 101

第 1 章

序論

本博士論文では，現在の技術的制約による不便さを排除し，利用可能性を向上する遠隔仮想計算機転送システムの構成法を明らかにしたものである．本論文では現在の主要な仮想計算機転送技術である，画面転送によるシンクライアント，VM（Virtual Machine）ライブマイグレーション，VM ストレージマイグレーションという 3 つの仮想計算機転送技術に対してそれぞれ貢献を行う．

1.1 研究背景

仮想マシンモニタ (VMM : Virtual Machine Monitor) とは一台の物理計算機上で複数の仮想計算機 (VM : Virtual Machine) を実現するためのミドルウェアである．VMM によって構築された VM は通常の物理計算機上で動作するソフトウェアを動作させる能力をもっている．つまり，物理計算機上で動作するためのソフトウェアを修正することなく，そのまま VM 上で動作させることができる．そのため，本来ならば，計算機を専有して管理する基本ソフトウェア (OS : Operating System) を単一の物理計算機上で同時に動作させることができる．

今日では，VMM を用いた，遠隔仮想計算機環境や VDI (VDI : Virtual Desktop Infrastructure) といった仮想化インフラの発展している．遠隔仮想計算機環境の具体例としてはとは，ユーザに対して，コンピュータネットワークを通して，仮想計算機 (VM : Virtual Machine) を計算機資源として提供するシステムのことである．このようなシステムの例としては，Eucalyptus[1] などがあげられる．また，大規模な遠隔計算機環境の商用事例としては Amazon 社の Amazon EC2[2], Microsoft 社の Windows Azure といったサービスも存在する．こういったシステムは「クラウドコンピューティング」といった用語で語られることがある．クラウドコンピューティングの場合は，インターネットを通して不特定多数の人間に対して VM リソースを提供するシステムを暗に示している一方で，VDI は，クラウドコンピューティングよりは，やや，利用者が限定された，特定の組織や企業が，特定の人員のみに提供する仮想化インフラという意味で使われる．クラウドコンピューティングと同じく，中央集権的なサーバに VMM を用いて，ユーザの作業環境である VM をすべて集約化するという点では同じだが，企業等，特定の組織が，組織の構成員のみに対して仮想化インフラを提供している場合，クラウドコンピューティングという言葉は使わずに，VDI という用語が使われることが

ある．VDI の具体例としては VMware 社の VMware View[3] や，Ctrix 社の XenDesktop[4] などがあげられる．

こういった，VDI やクラウドコンピューティングが利用される背景には，情報漏えい対策や，コンピューター管理コスト増大といった問題が存在する．

VDI，クラウドコンピューティング，いずれの場合にせよ，遠隔地（リモート）にある VM をネットワーク越しに，手元の計算機で操作しなければならない点が共通点である．そのため，VM（つまり，ユーザの作業環境と言い換えて良い）とユーザをつなぐための使いやすいインターフェイス (遠隔仮想計算機転送システム) を開発することが重要である．

この，遠隔地にある VM を手元の計算機から利用するためのシステムは様々なものが提案されている．例えば，古典的なテキストベースな telnet, rlogin, ssh などはその代表である．また，比較的新しい概念である，グラフィカルベースのものとしては，シンクライアントシステムという概念が挙げられる．また，シンクライアントとは別に，VM ライブマイグレーションと呼ばれる，VM 自体を他の物理マシンに，停止することなく移動する技術も開発されている．

しかし，**現在存在する遠隔仮想計算機転送システムは 2 つの意味で「限定的である」**その問題点を次節にて分析する．

1.2　本博士論文で解くべき問題点

現在の遠隔仮想計算機転送システムは 2 つの意味で「限定的である」その問題点について述べる．

導入には専門知識が必要　既存のシンクライアントシステムは，ユーザが利用の際に特殊なソフトウェアや機器をインストールしなければならない．これには専門知識が必要である．専門知識を備えた人間であれば，既存のシンクライアントシステムを使いこなすことは容易である．しかし，クラウドコンピューティングや VDI と言ったシステムは，必ずしもそういった専門知識がある人たちだけに対して恩恵をもたらすもではない．むしろ，そういった専門知識がない人たちこそ，こういったシステムを積極的に活用し，より便利なコンピューティング環境を築けるべきである．しかし，既存シンクライアントの導入に対して専門知識が要求され，利用可能性が低いままでは，クラウドコンピューティングや VDI は一部の限定された人々しか使うことの出来ないシステムのままである．そのため，シンクライアントシステムの利用可能性を向上する必要がある．

使う環境が限定されている　VM ライブマイグレーションは，VM の実行コンテキストを維持したまま，他の物理マシンに VM を転送する技術である．この技術はシンクライアントシステム上では利用が難しい，CAD や CAM，3D ゲームといったアプリケーションを使用したい場合，手元の物理的な計算機上で動作させることが出来れば VM のより柔軟な運用が可能になる．つまり，普段は耐故障性の高いハードウェア上で VM を動作させており，シンクライアントシステム経由で VM を利用しているが，必要なときは，VM をスムーズに手元の計算機にダウンロードできるようにすることで，利用可能性を向上することができる．しかし， VM ライブマイグレーションは，異なる OS 間

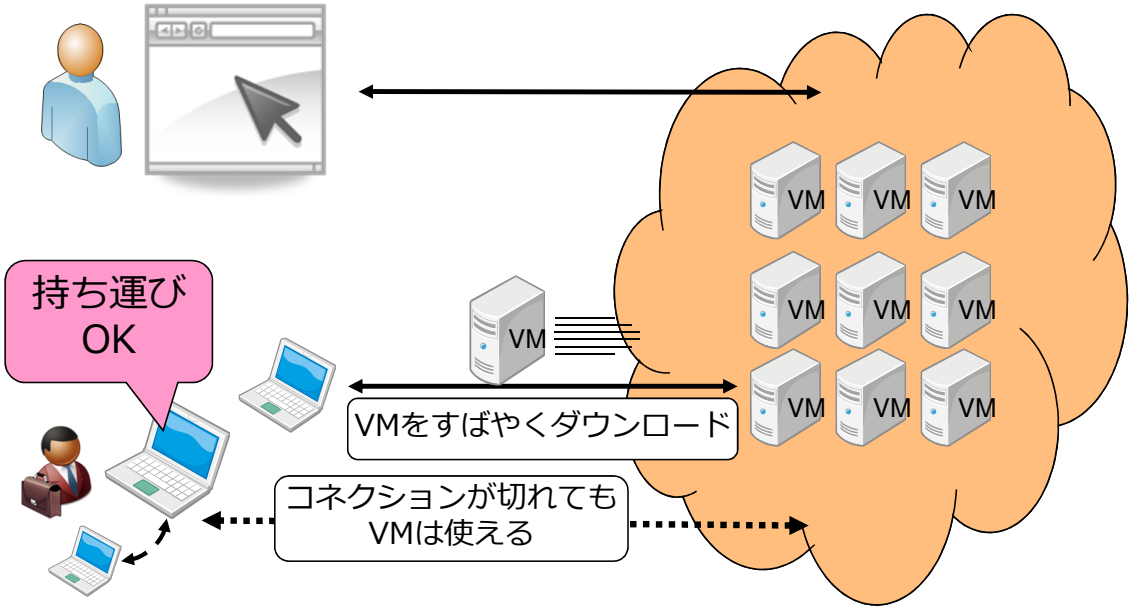


図 1.1. 将来の VMM を使ったコンピューティング環境

での VM ライブマイグレーションが可能な VMM が存在しない．クラウドコンピューティングや VDI といったシステムを構築する際には，オペレーティングシステムに，サーバ/サービス構築用として実績のある Linux が使用されることが多い．一方で，エンドユーザは，コンシューマ用途として広く普及している Windows が利用することが多い．そのため，Windows と Linux 間で VM ライブマイグレーションが可能である VMM が存在しないため，VM ライブマイグレーションが使用出来る環境が限定されている．結果的に，VM ライブマイグレーションのり用可能性を低下させている．また，VM のストレージデータは一般に巨大であり，数十 GB の容量を持つ．従来の VM ライブマイグレーションでは，分散ファイルシステムや共有ファイルシステムを利用して，これらのリソースを転送先と転送元で共有することが前提条件であったが，今日の VM の運用の変化，広域 WAN 環境でのデータセンターをまたいだ VM ライブマイグレーションや個人用の VM ライブマイグレーションといった VM の運用形態の変化によって，分散 / 共有ファイルシステムの利用が困難な状況下での VM ライブマイグレーションが現実的になっている．そのため，VM のストレージごと転送するという VM ストレージマイグレーションと呼ばれる技術の研究開発が進んでいるが，VM のストレージは巨大であるため，転送時間が増大したり，太い回線帯域が要求される．そのため，やはり「使う環境が限定」されており，利用可能性が低い．

これら 2 つの問題点を解決することで，より多くの利用シーンでも利用可能になる遠隔仮想計算機操作システムを実現することができる．イメージ図を**図** 1.1 に示す．ユーザはいかなるソフトウェアを追加インストールすることなく，手持ちの計算機からリモートにある自分の VM を遠隔操作することができる．また，もし，ネットワーク環境に依存しない計算機リソースが欲しい場合は，遠隔操作ではなく，直ちにリモートの VM を手元のクライアントにダウン

ロードして，以後は，ネットワークの接続を必要とすることなく VM を利用することができる．このように，本研究を利用することで，今までは限定的であった遠隔仮想計算機操作システムの利用可能性がより広がることが期待できる．

本博士論文では，VoXY，WinKVM と VM ストレージマイグレーションの高速化という 3 つの研究テーマを通して現在の限定的である遠隔仮想計算機転送システムを改良し，利用可能性を広げることが目的である．

1.3　解決手法

1.2 **節**で述べた問題点を，本論文ではどのように解決するのかについて述べる．

本研究では 3 **章**で述べる「JavaScript と HTML を用いたシンクライアント VoXY」4 **章**で述べる「Linux ドライバの Windows 移植による WinKVM の実装」そして，5 **章**で述べる「差分転送を用いた高速な VM マイグレーション」の 3 つの研究にて「使う環境が限定されており，使う人を選ぶ」遠隔仮想計算機転送技術を「より多くの環境で使え，より多くの人が使える」遠隔仮想計算機転送技術に改良することを目的とする．

シンクライアントシステムの利用可能性を低減させている原因である「導入には専門知識が必要」という問題に対しては，Web ブラウザ上でシンクライアントシステムを動作させるというアプローチでこれを解決する．VoXY の開発が始まった 2007 年ごろには，Web ブラウザ上でユーザのコンピューティングのすべてを行わせるという中央集権的なコンピューティングが再び普及しつつあった．Web ブラウザほとんどすべてのコンピュータにプリインストールされ，コモディティ化されており，専用のソフトウェア等を導入せずとも，ほとんどすべての端末，携帯端末でアプリケーションをきるという利点がある．そのため，Web ブラウザ上で動作するというアプリケーションは，追加のソフトウェアのインストール作業も必要なく導入が容易であり，そのため，エンドユーザに受け入れ易いものであった．

しかし，既存の Web ブラウザ上で動作するシンクライアントシステムは，Web ブラウザとは別にプラグインを導入が必須であった．後に詳しく述べるが，既存のシンクライアントのプロトコルはすべて，クライアント側に高度な図形描画インタフェイスを要求するものとなっているため，単純な画像描画のインタフェイスしか持たなかった 2007 年当初の Web ブラウザでは，図形描画インタフェイスを強化するための，プラグインでの機能拡張を行う必要があった．プラグインは Web ブラウザ以外のオペレーティング・システムやライブラリに依存するものであるため，Web ブラウザのみで完全に動作するものではない．そのため，Web ブラウザであればどこでも必ず動作するというわけではなく，利用可能性が低い．

そこで，われわれは，Web ブラウザにプラグインを導入せず，純粋に HTML + JavaScript のみでシンクライアントシステムである VoXY を実現した．しかし，既存のすべてのシンクライアントのプロトコルは，画面のフレームバッファの変更点のみをインクリメンタルに転送するという手法を採用している．この手法は，使用帯域を低減することができるが，JavaScript + HTML のようなリッチな画像描画インタフェイス持たない場合，このプロトコルは不適切である．そのため，変更部分のみをインタラクティブに転送するのではなく，フレームバッファを固定長のタイルに分割し，変更箇所のあった部分を，タイルごと転送するという新たなプロトコルを提案した．この手法では，プロトコルの要求帯域は大きくなるも

の，HTML + JavaScript でも十分なインタラクティブ性能を確保でき，利用可能性の高いシンクライアントを実現することができる．

次に，既存の VM マイグレーション技術の利用可能性を低減させている「コンシューマ向け OS とサーバ OS を結びつける VMM が無いという点である」という点については，異なる 2 つの OS 上で動作する移植性の高い VMM を実装し，複数 OS 上で移植が容易な VMM を設計する上での留意すべき点をまとめることで，異種 OS 間で VM ライブマイグレーションを達成することで利用可能性を向上するという手法を取る．

最後に，VM ストレージマイグレーションの利用可能性を低下させている原因である「転送に時間がかかる」という問題に対しては，既存の VM ストレージマイグレーションに対して差分転送という機構を新たに組み込む．これで，相手先にすでに転送済みの VM ディスクページが存在していれば，VM ディスクイメージのすべてを転送せずとも，転送先と転送元で変更のあった部分だけを検出して転送できるため，大幅に転送速度を削減することで利用可能性を向上するという手法を取る．

1.4 解決手法の実現

1.3 節で述べた解決手法の実装について，VoXY，WinKVM，VM ストレージマイグレーションの高速化の順で詳しく述べてゆく．

1.4.1 JavaScript と HTML を用いたシンクライアント VoXY

VoXY は，Web ブラウザ上で動作する，エンドユーザにとって導入が容易なシンクライアントソフトウェアである．VoXY は，Web ブラウザの標準プロトコルである HTML と JavaScript のみで高いインタラクティブ性を持つシンクライアントを実現している．そのため，追加のプラグイン等のインストールは必要ない．また，Web ブラウザは多数の OS のインストールパッケージや携帯単発に初期導入されているため，インストールレスで使えると言える．そのため，既存のシンクライアントシステムやと比較すると，専門知識が不要である．

しかし，この JavaScript と HTML とシンクライアントシステムを実行するためには，解決しなければならない問題が存在する．VoXY の開発を始めた 2007 年当初には，まだ JavaScript と HTML には，高度な図形描画処理機能が存在しておらず，そのため，既存の，変更点のみをインクリメンタルに転送するシンクライアントプロトコルでは Web ブラウザ上でインタラクティブ性の高いシンクライアントシステムを実現するのは困難であった．

この問題を解決するため，新たなシンクライアントプロトコルを考案した．従来の変更点のみを転送するのではなく，計算機のフレームバッファを固定長のブロック（タイル）に分割し，変更点が存在する箇所のブロックごと画像転送するプロトコルを設計した．この手法では，従来のシンクライアントプロトコルに比べると，要求帯域が高くなるという欠点が存在するが，純粋に JavaScript と HTML のみで高いインタラクティブ性を持つシンクライアントを実現することができるので，従来のシンクライアントシステムよりも，利用可能性が高くなる．

画面転送というレベルのみで，ユーザと VM を接続するには，まだ限定的であり不便である．シンクライアント VoXY は VM の画面転送という形で VM とユーザを接続する．しか

し，CAD やビデオ編集ソフトといったインタラクティブ性が求められるソフトウェアをユーザが利用したい場合，シンクライアントソフトウェアである VoXY では，ユーザに快適な操作性を提供することができない．こういったソフトウェアを利用する場合は，計算機リソース自体が，ユーザの手元にある計算機にダウンロードされ，その上で動作することが望ましい．これを実現するため，われわれは，VM ライブマイグレーションと呼ばれる技術を活用することにした．この技術を使えば，遠隔地にある計算機リソース (VM) を手元の計算機に即座にダウンロードすることができる．

しかし，現行のライブマイグレーションには問題がある．まず，コンシューマ向けのオペレーティングシステムである Widows と，システム構築用に使われる Linux との間でライブマイグレーションを実現するための仮想マシンモニタが存在しないのである．さらに，VM ライブマイグレーション自体は VM の仮想ディスクそのものを転送するため，転送完了までに，極めて多くの時間を消費する．われわれは，この 2 つの問題を解決するために WinKVM と差分転送による高速な VM ライブマイグレーションを実現した．

1.4.2　Linux ドライバの Windows 移植による WinKVM の実装

WinKVM は，Linux カーネルに付属する VMM (KVM: Kernel-based Virtual Machine) を Windows に移植したものである．

KVM は Linux デバイスドライバとして実装されているため，WinKVM を実現するためにはこの Linux デバイスドライバをいかにして Windows 上で動作させる必要がある．しかし，Linux にかぎらず，オペレーティング・システムのデバイスドライバというものは，OS の実装に強く依存するプログラムあるため，一般論としてドライバの移植というものは困難である．そのため，一般的には，プログラマがデバイスドライバのコードを読み込み，コードを移植してゆくのが一般的である．

本研究では，Linux デバイスドライバである KVM を Windows 上で動作させるため，Windows カーネル上に Linux カーネルを模倣するエミュレーションレイヤを構築して，Linux / KVM ドライバを Windows 上で動作させる手法を開発した．利用可能性の高い仮想計算機転送技術を実現するためには，様々なオペレーティングシステム上で動作する Hybrid VMM の実現が重要であることは既に述べた．そのため，Windows と Linux といった，まったく異なる設計思想を持つカーネル上にて同一の VMM を動作し，Linux と Windows 間で VM ライブマイグレーションが実現できるようになったため「使う人を選ばず」「使う環境が限定されていない」という限定的な遠隔仮想計算機転送技術である VM ライブマイグレーションという技術を改良することに成功した．

また，本エミュレーションレイヤを用いて，Linux / KVM ドライバを Windows 上で動作させることで，マルチ OS 対応の Hybrid VMM を設計する際に留意すべき点についてもまとめることができた．

WinKVM の研究により，コンシューマ向け OS である Windows と，サーバ向け OS として高い人気を誇る Linux 間でライブマイグレーションを可能とする VMM を容易に設計実装するための知見を得ることができた．共有ファイル・システム等を利用すれば，WinKVM と Linux 間で VM ライブマイグレーションを行い，計算機リソースを手元の計算機にダウンロー

ドすることも可能である．しかし，共有ファイル・システムという，ネットワークが常につながっている環境を想定しているシステムは，まだ，本博士論文が解決する問題として掲げている「使う環境が限定されている」を完全に解決したとはいえない．これを解決するためには，たとえネットワークコネクションが断線した後も継続して計算機リソースを利用できる仮想計算機転送システムを構築する必要がある．しかし，WinKVM だけでは，それを実現することはできないため，次節で解説する VM リソースのすべてを転送する技術を開発した．

1.4.3 差分転送を用いた高速な仮想計算機転送システム

上述した通り，WinKVM と VoXY の研究だけでは，ローカルとリモートがネットワークで常に接続されている必要がある．そのため，これらの研究はネットワークがなければ使用することができない．このままではまだ使用出来る環境が限定されている遠隔仮想計算機転送技術になってしまう．

そこで，次の研究としてネットワークが断線したとしても，引き続きエンドユーザが仮想計算機を利用できるような遠隔仮想計算機を実現する必要があると考えた．つまり，ネットワーク依存性を低くする必要がある．ネットワーク依存性を取り除くためには，VM の仮想ディスクファイルの共有に，共有ファイル・システムを利用するのではなく，VM の仮想ディスクを含めて仮想計算機のリソースをすべてまるごと他の他のマシン（ローカル側）にダウンロードする必要がある．これで，ネットワーク依存性を断つことが可能である．

本研究では，ネットワーク依存性を低くするために，VM ストレージマイグレーションと呼ばれる技術に着目した．この技術を使えば，VM の仮想ディスクを含めて，VM を他の物理マシンに転送する必要がある．しかし，VM の仮想ディスクは一般的に巨大なファイルサイズを持つため，ユーザが VM ストレージマイグレーションを開始してから，ネットワーク無しで VM が使えるようになるまで多くの時間がかかる．また，帯域の細い回線では，より多くの時間を必要としてしまう．

そこで，差分転送による VM ストレージマイグレーション技術に関する提案では，仮想マシンモニタに DBT (Dirty Block Tracking) と呼ばれる機構を取り付けることで VM のディスク書き込みをすべてトラッキングし，転送先と転送元で変更のあった部分のみを高速に，かつ，正確に検出し，転送する．VM ライブマイグレーションの転送時間でもっとも支配的な要素は VM の仮想ディスクファイルの転送にかかる時間である．そのため，DBT による差分の検出と転送により，高速な VM 転送を実現し「使う環境が限定されている」VM ストレージマイグレーションの技術をより多くの環境でも使えるように改良した．ベンチマークの結果，90% 以上の転送時間を削減することができ，既存の VM ストレージマイグレーションと比べると高速に VM をダウンロードすることに成功した．

1.5 研究の問題点と達成した事柄のまとめ

まとめると，本研究では VoXY：Vnc over proXY，WinKVM：Windows kernel-based Virtual Machine と差分転送による VM ストレージマイグレーションの高速化という 3 つの研究で，「使う環境が限定されており，使う人を選ぶ」遠隔仮想計算機転送技術を「より多く

の環境で使え，より多くの人が使える」遠隔仮想計算機転送技術に改良することに成功した．

シンクライアントシステムの利用可能性を低減させている原因である「導入が面倒」「特殊なプロトコルを利用し，使える環境が限定的」という問題に対しては，VoXY によって HTTP でインタラクティブ性の高いシンクライアントシステムを提供することでこの問題を解決した．クライアントとなる Web ブラウザはほとんどすべてのコンピュータにプリインストールされており，特別な整備無しで利用することが可能である．また，HTTP という一般的なプロトコルですべての通信を行うため，あらゆる環境で使用することができる．

次に，既存の VM マイグレーション技術の利用可能性を低減させている「コンシューマ向け OS とサーバ OS を結びつける VMM が無いという点である」という点については，WinKVM を設計実装の際に得た知見である異なる 2 つの OS 上で動作する移植性の高い VMM を実装手法の提案によって，容易に環境の異なる OS 間でも VM ライブマイグレーションが達成できるようになった．

最後に，VM ストレージマイグレーションの利用可能性を低下させている原因である「転送に時間がかかる」という問題に対しては，今までの VM ストレージマイグレーションに差分転送という機構を新たに組み込んだ．これで，相手先にすでに転送済みの VM ディスクページが存在していれば，VM ディスクイメージのすべてを転送せずとも，転送先と転送元で変更のあった部分だけを検出して転送できるため，90% 以上の転送速度を削減することに成功した．

VoXY，WinKVM と差分転送による VM ライブマイグレーションの高速化という 3 つの研究でもって，あらゆる利用シーンで，ユーザと仮想計算機を接続することができる仮想計算機転送技術を達成した．

1.6　本論文の構成

本論文の構成を以下に示す．はじめに，2 章にて関連研究について述べる．この章にて，仮想マシンモニタと遠隔仮想計算機転送システムの歴史について述べる．次に，3 章にて，エンドユーザにとって導入が容易なシンクライアントを，Web ブラウザ上で実現する新たなシンクライアントシステム VoXY を提案する．4 章 にて，Linux カーネルに付属する仮想化機構である (KVM: Kernel-based Virtual Machine) を Windows 上に移植した WinKVM について論じる．次に，5 章にてストレージマイグレーションの高速化手法について論じ，最後に 6 章でまとめと今後の課題を示す．

第 2 章

関連研究

仮想計算機モニタ (VMM: Virtual Machine Monitor) とは，物理的に存在するマシンと同様の処理を行える仮想計算機 (VM: Virtual Machine) を，物理マシン上に生成するミドルウェアシステムである．生成された VM はそれぞれ個別のオペレーティングシステム (OS: Operating System) をインストールすることができるため，複数 OS を同一物理マシン上で同時に動作させることが可能となる．

本章では仮想マシンモニタの歴史を簡単に振り返り，仮想マシンの研究がどのように勧められていったのかについて述べる．その後，本研究が歴史的にどのように位置づけられるかについて論じる．なお，この章に関しては，Mendel らによる総説 [5] を参考にして書かれている．

2.1 仮想マシンモニタ (VMM) の歴史

本節ではまず，仮想マシンモニタ (VMM: Virtual Machine Monitor) の黎明期から紐解いていく事にしよう．実は，VMM の概念は自体は古く，すでに 1960 年代には提唱されていたのである．

VMM (Virtual Machine Monitor) という概念は 1960 年の終わりに，IBM System/370 上に実装されたハードウェアアブストラクションレイヤーである VM/370 にその起源を見出すことができる．VM/370 の仮想マシンモニタに関しては Goldberg による総説 [6] が詳しい．VM/370 のアーキテクチャを**図** 2.1 に示す．

当時のシステムでは BARE MACHINE と呼ばれるコンピュータが提供する BASIC MACHINE INTERFACE 上に直接実装された VMM が多数の VM を生成する．生成された VM は BARE MACHINE が提供する BASIC MACHINE INTERFACE と全く同様のインターフェイスを提供する．そのため，BARE MACHINE 上で動作する PRIVILEDGE SOFTWARE NUCLEUS と呼ばれるソフトウェア (今日のオペレーティングシステムであると考えていよい) が動作する．PRIVILEDGE SOFTWARE NUCLEUS ソフトウェアは EXTENDED MACHINE INTERFACE を提供し，その上でユーザが利用するソフトウェアが動作する．

IBM の技術者たちは，彼らが開発した IBM System/370 上にタイムシェアリングシステムを実現するためのオペレーティングシステムである VM/370 を実装した．VM/370 は互いに

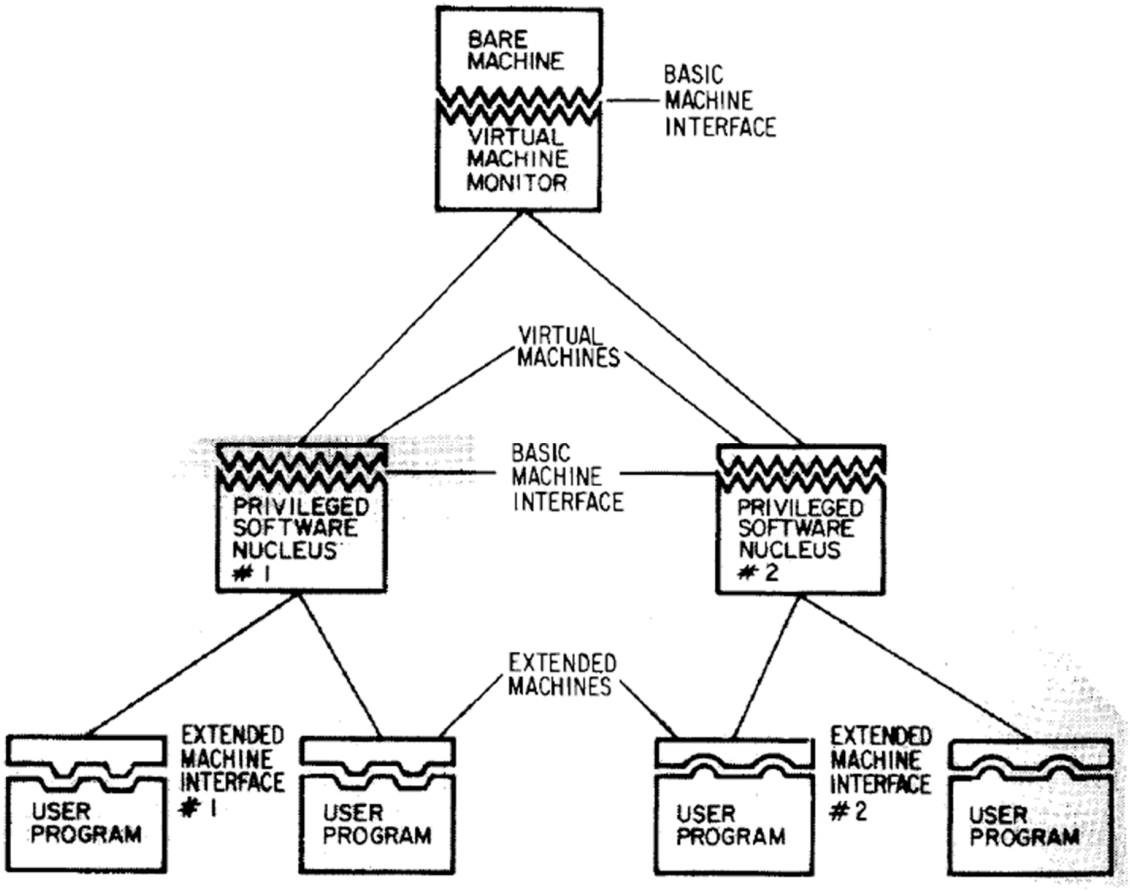


図 2.1. System/370 上に実装された VM/370（VMM）のアーキテクチャ図．Goldberg の論文 [6] より引用

完全に独立した複数の仮想的な IBM System/370 コンピュータをユーザに提供することができた．ユーザがコンピュータにログインすると，VM/370 によって新たな仮想マシンが作られる．その仮想マシンは一台の IBM System/370 と全く同様に動作し，ユーザはまるで，一台の IBM System/370 メインフレームコンピュータを独占して使っているかのように計算機を利用することができたのである．当時の様子は Creasy の文献 [7] に詳しい．

VM/370 の開発によって，独立した仮想計算機 (Virtual Machine) として IBM System/370 上で複数再現することが可能となり，ユーザがコンピュータに同時にアクセスできるようになったのである．VM/370 は今日のマルチタスクオペレーティング・システムの先駆けとも言える存在である．現在では，仮想計算機はプロセスと言う概念にとって代わり，それらを複数並行動作させることができるマルチタスクのオペレーティング・システムはありふれたものになっている．しかし，同時としては，タイムシェアリングシステムとして実装されたマルチタスクのオペレーティングシステムは画期的なものであった．

タイムシェアリングシステムを実現した基盤ソフトウェアとして脚光を浴び，盛んに研究された仮想マシンモニタであったが，1980 年から 1990 年にかけて，VMM は徐々に使われなくなっていった．モダンなマルチタスクオペレーティングの普及と，安価なハードウェアの急速

な普及により仮想マシンモニタの存在意義がなくなりはじめたのがその理由である．仮想マシンモニタを導入して自分の欲するアーキテクチャをエミュレーションするよりも，そのアーキテクチャを持つコンピュータを購入したほうが安上がりであり，また，モダンなマルチタスクオペレーティング・システムのおかげで，VMM 上でプログラムを走らせるよりも，特定のオペレーティングシステムのプロセスとしてプログラム動作させたほうが効率が良くなったのである．

1980 年代の終わりには，すでに，研究者もまた，産業界の人間も VMM を過去の遺物としてしか見ていなかった [5] のである．

ところが，2005 年になり，再び VMM が研究者や産業界の人々から注目を浴びることになる．コンピューティング環境の可搬性向上，セキュリティや続ける計算機の管理問題といった問題にたいし，VMM を使ったアプローチが有用であると考えられ始めたのである．多くのベンチャーキャピタルが VMM の使って，これらの問題を解決するための新たな技術を開発し始めている [5].

2005 年の少し前，1990 年代にスタンフォード大学の研究者は，プログラムが困難で，既存のオペレーティングシステムを再利用できない超巨大なマルチプロセッシングマシン（MPP）上でプログラムを動作させる手法として VMM を活用する手法を研究していたのである．彼らは，使いにくい（MPP）上のマシンでも，既存の，プログラマがよく使い慣れたアーキテクチャを VMM で擬似的に再現して，プログラマを補助する研究を行なっていたのである．つまり，MPP を扱いやすくするための手法として VMM を利用しようと考えており，そのプロジェクトが後に VMware 社の創業につながったという経緯がある．

現在，VMM は以下に示す観点から主に利用価値が見出されている．

1 つ目の利点は，プログラムの実行環境を隔離できる点である．VMM 上で動作する個々の VM は他の VM から完全に隔離されている．そのため，もし 1 つの VM が破滅的なソフトウェアクラッシュを引き起こしても，他の VM は影響を受けることはない.

2 つ目の利点は，VM の外から制御できるレイヤを追加できる点である．物理的に存在する計算機の場合，OS に依存しない，外からその計算機を制御するための統一したインターフェイスを作るのは困難である．しかし，VMM 上で動作する VM の場合，例えば，計算機に対する電源の ON/OFF やメモリへの読み書き，と言った制御は VMM を通してすべて制御することが出来る．そのため，VMM 上から VM ないのメモリをスキャンして，不正なプロセス (例えばウイルスプログラム) を監視できたり，柔軟な VM の電源 ON/OFF を利用した省電力化と言った研究がされている．

3 つ目の利点は，所有していないハードウェアを模倣させられる点である．VMM は実ハードウェア上に仮想ハードウェアを構築するため，現在では開発中であったり，既に販売が中止されてたり，といった，稀少なハードウェアを仮想ハードウェアとしてエミュレーションすることが出来る．古くから使われており，新しいソフトウェアに移行することが難しいソフトウェアの保守作業などでは，VMM のこの機能が特に使われていることがある．

現代の VMM は様々な観点から利便性が認められており，社会に受け入れられている．

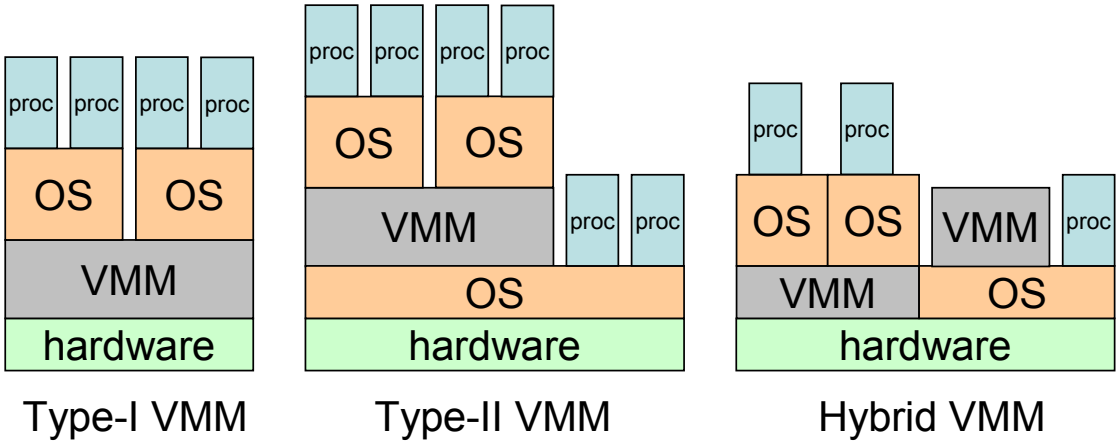


図 2.2. VMM の分類

2.2　VMM の技術的な要素

VMM に関する厳密な技術的な定義は存在しないものの，一般的に，VMM は CPU エミュレーターと異なるものとして扱われる．Bochs をはじめとする，いわゆる CPU エミュレータは，CPU の挙動をすべてソフトウェアでエミュレーションする．すなわち，CPU の基本動作である，命令フェッチ，デコード，実行をソフトウェアで模倣する．一方，VMM の場合，実 CPU 上で直接実行可能な部分は可能な限り実 CPU 上で実行し，実 CPU 上で実行不可能な命令のみをソフトウェアで模倣する．そのため，原則的に，VM の CPU のアーキテクチャは物理マシン上のそれと同じものしか生成できない．つまり，CPU エミュレータのように，Intel の X86 アーキテクチャ上で Sun SPARC アーキテクチャを模倣するといったことは不可能である．その代わりに，VMM の生成する VM は CPU エミュレーターと比べて高速に動作する．なぜなら，実行可能な命令は実 CPU に直接実行させるためである．

2.2.1　VMM の分類

VMM は様々な方法で実装されるため，実装法によって分類される．それぞれの分類を**図** 2.2 に示す．**分類には大きく分けて Type-In VMM (hypervisor 型), Type-II VMM, Hybrid VMM** の三つが存在する．これらの分類は

分類の解説に入る前に必要な用語を定義する．**ホスト OS** とは VMM の下位レイヤ，もしくは，VMM 自身と同じレイヤで動作する OS のことである．つまり，VMM はホスト OS 上で動作するシステムソフトウェアの一つである．一方，**ゲスト OS** とは，VMM が動作する仮想マシン上で動作する OS のことである．VMM によって，ゲスト OS はホスト OS と同じ OS でなければならないものから，そうでないものまで様々である．

Type-I VMM にはホスト OS が存在しない．VMM 自身がホスト OS の機能を兼ねているため，Type-I VMM を Hypervisor OS と呼ぶこともある．代表例としては，VMware ESX

Server, Xen, Hyper-V などが挙げられる．このタイプの VMM は仮想化に特化した OS を VMM 自身が持っているため，一般的に，大量かつ高速な VM を生成することが可能である．変わりに VMM のコード量は膨大である．例えば，Xen は Linux カーネルを丸ごと改造した VMM であり．コード量も Linux カーネル自身と変わらないほど巨大である．

Type-II VMM はホスト OS 上でユーザプロセスとして動作する VMM であり，代表例としては User Mode Linux (UML) [8] が挙げられる．UML はユーザ上で動作するように改造を施された Linux カーネルである．そのため，UML 利用者は，Windows 上で，気軽に Linux 環境を試すことが出来る．この種の VMM は，特定の OS をアプリケーションプログラムとして動作するように改造したものが多い．そのため，ゲスト OS として特定の OS しか実行できず，柔軟さに欠けると言った欠点がある．しかし，ゲスト OS が完全にネイティブアプリケーションとして動作するため，高速に動作することが見込まれる．

Hybrid VMM はホスト OS の一部として動作する VMM である．代表的な例としては KVM や，本論文で提案する WinKVM などが挙げれる．この種の VMM はゲスト OS として様々な OS を動作させることができる VMM が多い．なぜなら，この種の VMM はホスト OS のデバイスドライバとして実装されていることが多く，制限のないカーネル空間で動作する．そのため，Type-II VMM の実装手法の量と比べると，Hybrid VMM の実装手法のほうが取れる選択肢が多いからである．また，VMM の機能の一部にホスト OS の機能を利用するため，ソフトウェアの規模も，Type-I VMM と比べて小さいものが多い．事実，KVM のコアは数千程度であり，これは Xen と比べると格段に小さい．

2.2.2 VMM の技術的なチャレンジ

VMM で最も重要な課題はパフォーマンスである．物理的なマシンと互換性のあるアーキテクチャである仮想マシン（VM : Virtual Machine）を生成しつつ，物理マシンと遜色のないパフォーマンスを達成するのは困難である．

計算機を構成する主な要素は，CPU，メインメモリ，入出力（I/O）の 3 つである．そのため，互換性を維持するためにはこれら 3 つを物理マシンと互換性ができるように，エミュレーションする必要がある．この，互換性を維持しつつ，パフォーマンス達成するために，それぞれ技術的課題がある．本節ではそれらの課題について述べる．

CPU の仮想化

CPU の仮想を行うにあたって，物理マシンと VM の互換性を維持しつつ物理マシンと遜色のないパフォーマンスを達成するため，機械語命令をなるべく物理 CPU に直接実行させるという手法をとる．

CPU の命令は，特権命令（Privileged code）と非特権命令（Unprivileged code) に分類される．特権命令は基本的にオペレーティング・システムのカーネルが実行するものである．一方で，非特権命令はオペレーティング・システム上で動作する通常のアプリケーションが実行する．VMM は基本的にオペレーティング・システムと同等の特権レベルで動作する．すなわち，VMM は OS と同様，特権命令が発行できる特権レベルで動作する．一方で，VMM は非特権命令しか発行できない動作モードでゲスト OS を動作させる．ゲスト OS 内で特権命令が

実行された場合は，実 CPU が一般保護例外等の割り込みを発生させるため，の特権モードで動作している VMM がそれをトラップして必要なエミュレーションを行い，その後，ゲスト OS に処理を返す．それ以外の，非特権命令は直接実 CPU に実行させるという手法を取っている．これが基本的な VMM の CPU アーキテクチャの仮想化原理である．

ところが，現時点で広く普及する CPU アーキテクチャである x86 アーキテクチャでは，この手法をナイーブに適応することができない．x86 アーキテクチャは基本的に仮想化をサポートするようには設計されていなかったため，上述した CPU の仮想化手法が適応できないのである．

例えば，x86 命令である `POPF` 命令があげられる．これは，CPU の動作モードを規定するための EFLAGS レジスタの値を変更する命令である．外部割り込みの ON/OFF を切り替えるのもこの EFLAGS レジスタの値を変更することで行う．そのため，VM の CPU と x86 の実 CPU の互換性を保つためには，VMM が EFLAGS レジスタへのアクセスをトラップによって制御する必要がある．しかし，x86 はもともと仮想化をサポートするように設計された CPU ではなかったため，EFLAGS レジスタの値に影響を与える可能性のある `POPF` 命令をトラップする手法が存在しないのである．そのため，実 CPU と VM の互換性を維持するのは困難であった．

x86 アーキテクチャのように，仮想化を支援しない CPU 上で VMM を構築するための様々なテクニックが開発されてきた．最初に開発された手法は準仮想化（paravirtualization）[9] である．これは，VM 上で動作するゲスト OS の命令を書き換えることで，仮想化を容易にする手法である．例えば，x86 アーキテクチャの場合 `POPF` といった，トラップが不可能な命令を，他のトラップ可能な命令（`int3`:デバッグ命令）に置き換えることで，仮想化を容易にする手法である．VM 上で動作する特権命令を含むゲスト OS はそれらの特権命令を書き換える必要があるが，ゲスト OS 上で動作する，特権命令を含まない非特権命令からなるアプリケーションソフトウェアの場合は非修正のまま動作させることができる．

Disco[10] は MIPS アーキテクチャで動作する VMM である．MIPS プロセッサも x86 アーキテクチャと同様に仮想化には対応していない．そのため，MIPS プロセッサにも x86 アーキテクチャでいうところの `POPF` 命令に相当するものが存在するが，これをトラップしてエミュレーションすることが不可能である．MIPS アーキテクチャの場合，割り込みの ON/OFF を行うためには，特別なメモリ領域を書き換えることでこれを実現する．Disco の開発者は，この割り込みの ON/OFF を制御する特別なメモリ領域へのアクセス命令を，VM の割り込み ON/OFF 状況を保持するメモリ領域へのアクセスに置き換えるという手法を取っている．この手法により，特権命令をトラップするコストも削減できるため，パフォーマンスも向上する．

準仮想化技術の手法には問題点もある．それは，OS を書き換える必要が有るため，互換性が犠牲になるという点である．特定の VMM 向けに改造されたオペレーティング・システムしかゲスト OS として実行できないのである．そのため，改造を施されていない，非修正のオペレーティング・システムを動作させることはできない．

この問題を解決するため，つまり，非修正のオペレーティング・システムでも VMM 上動作させるため，VMware 社が新しい仮想化技術を開発した．それが，*on-the-fly binary translation* である．たいていのオペレーティング・システムの場合，OS の上で動作する通常

のアプリケーション・プログラムの動作モードは仮想化可能である．ゆえに，アプリケーション・プログラムの部分のみは実 CPU に直接実行させることができる．仮想化不可能な特権モードをエミュレーションするときには，特権モードで走るコードを動的に仮想化可能なアプリケーションのエミュレーションコードに置き換えるのである．この手法により，非修正の OS でも VMM 上で動作させることができるようになった．

他の，binary translation[11] のテクニックとは異なり，VMware の binary translation はもっとシンプルである．コードの解析等を行わず，実行対象のコードを特定の長さのコード篇に分割し，ident 命令（エミュレーションシなければ鳴らない命令），変換していくという手法がとられる．いわゆる Java の JIT コンパイラのようなものであるが，JIT コンパイラがやるような高度な解析や最適化は行わない．OS のコードはすでに十分に最適化されていると考えられるためである．

さらに，2007 年ごろから，もともと仮想化に対応していなかった x86 アーキテクチャ自身が仮想化に対応する拡張を導入した．これにより，仮想マシンの開発が容易になった．Intel 社が発表した Intel VT-x や AMD 社が発表している AMD-SVM などがそれに相当する．

メモリの仮想化

VMM は CPU のインストラクションのみを仮想化するのではなく，VM が使用するメモリも仮想化する必要がある．これには *shadow pageing* と呼ばれるテクニックが使用される．VMM は予め実 CPU が使用するページテーブルとは別に，VM 毎のページテーブル（これが *shadow page table* である）が用意される．VM 上で動作するゲスト OS が shadow page table に変更を加えた場合，それを VMM がトラップし，実 CPU のページテーブルに反映する．

また，実 CPU からページフォルトが来た時，VMM がそれを受け取り，VM の shadow page table を解析する．解析の結果，そのページフォルトが VM にパススルーすべきものであれば，VMM はそのページフォルトを VM 上のゲスト OS に通達する．そうでなければ，それは VMM 自身が処理すべきページフォルトであるため，VMM が必要な処理を行う．

I/O の仮想化

I/O のエミュレーションは VM のパフォーマンスに影響を与える最も重要な要素である．VMM は基本的に，入出力命令，例えば，x86 アーキテクチャにおける `in` 命令，`out` 命令が発行された時，その命令をトラップし，VMM が仮想的なエミュレーションする仮想的なデバイスにそれらの命令をパススルーするという手が使われる．

また，入出力は `in` 命令や `out` 命令以外にも，Memory Mapped I/O (MMIO) と呼ばれる機構も存在しており，これは，特定のメモリアドレスにデータを読み書きすると，そのデータがそのまま，デバイスのデータを読み書き出来るという仕組みである．たとえば，Standard VGA などはその代表例である．IBM-PC アーキテクチャの場合，`0x000A0000 - 0x000BFFFF` にデータが読み書きされると，それはビデオカードデバイスへのアクセスとみなされ，ディスプレイを制御することができる．MMIO のエミュレーションは，ページフォルトを利用してエミュレーションされる．MMIO の領域を常に書き込み禁止にしておき，ゲスト OS がこの

領域にアクセスを行うと，VMM がそれをページフォルトとしてトラップし，デバイスをエミュレーションするという手法である．また，公にはなっていないが，VMware 等は Binary translation の技術をつかって，VMM への通知命令に変換している可能性も考えられる．

CPU の IO 命令，MMIO の命令，いずれにせよ，トラップのコストが非常に高いため，VMM 上の I/O は，物理マシンのそれに比べると非常にパフォーマンスが悪くなる可能性がある．そこで，近年では，ゲスト OS のデバイスドライバレベルで VMM と直接通信を行い，トラップのオーバーヘッドを回避するという手法がとられるのが一般的である．ゲスト OS と VMM との通信に用いられる特殊なドライバはパラバーチャルドライバと呼ばれ，VMM ベンダーが，ゲスト OS 毎に自社の VMM の専用のドライバを開発して提供するのが一般的である．

2.3　遠隔計算機操作システム

さて，本節からは，VMM から離れ，仮想計算機システムについて解説する．

遠隔計算機操作システムとは，ネットワークを通じて，遠隔地にある計算機上のデスクトップ環境を，手元の計算機上に再現するためのインフラストラクチャのことである．遠隔計算機操作システムにより再現されたデスクトップ環境を**仮想デスクトップ**と呼ぶ．

遠隔計算機操作システムの概念は目新しいものではなく，代表例としてはシンクライアント，WebVDI などが挙げられる．**表** 2.1 に主要な遠隔計算機操作システムを示す．

2.3.1　シンクライアントシステム

シンクライアントは既存の OS やウィンドウマネージャをそのまま仮想デスクトップとして転送することが出来る．そのため，今まで使用してたソフトウェアやデータをそのまま引き継いで利用できるという利点がある．一般的に，リモートの計算機にサーバソフトウェアをインストールし，ローカルの計算機には専用のクライアントソフトウェアをインストールする，いわゆる，クライアントサーバモデルである．また，クライアントとして特殊なアプライアンス (専用小型機器) があるシンクライアントも存在する．OS の種類やウィンドウマネージャの種類によっては，初期導入の時点でシンクライアントのサーバソフトウェア相当の機能が付属しているものもある．

WebVDI は，デスクトップマネージャのような Web アプリケーションを通して，Web ブラウザ上でさまざまな Web アプリケーションを統合的に扱うことができるシステムである．具体例としては，WebShaka が開発した YouOS[12] などが挙げられる．**このように，Web ブラウザ上で仮想デスクトップを動作させることのできるシステムを，本論文では新たに WebVDI と定義する**．WebVDI は，ユーザが利用する端末として，コモディティ化された Web ブラウザと，その上で動作する JavaScript のみで実装された Web アプリケーションを使用する．そのため，いかなる計算機上でも動作する．なぜなら，現時点での OS パッケージには標準で Web ブラウザがほぼ必ず添付されているからである．

表 2.1. 主要な遠隔計算機操作システム一覧

カテゴリ	具体例
シンクライアント	VNC[13], SLIM (SunRay)[14, 15], RDP[16], X[17], THINC[18, 19]
WebVDI	YouOS[12], eyeOS, StartForce
CUI ベース	telnet, SSH, rlogin

2.3.2 VM マイグレーション

VM ライブマイグレーションは VMM がもたらす仮想化の中で最も重要な機能の 1 つである．VMM により生成された VM を他の物理マシンに移動させることができる．VM マイグレーションによって，物理マシンとその上で動作するゲスト OS の疎結合を実現することができる．そのため，物理マシンに何らかの障害が発生した場合でも VMM のライブマイグレーション機能を使えば，他のマシンにユーザのコンピューティング環境を移動することができる．

プロセスマイグレーション

ユーザの実行コンテキストを移動させるという技術は VM ライブマイグレーション以外にも存在する．1980 年代には，プロセスマイグレーション [20, 21, 22, 23, 24] 呼ばれる研究が盛んに行われていた．この技術は特定のオペレーティング・システム上で動作するプロセスを他の物理マシンで動作するオペレーティング・システムに移動させる技術である．

しかし，プロセスマイグレーションと VM ライブマイグレーションは根本的に異なるものである．プロセスマイグレーションは，その特性上，オペレーティング・システムの実装に強く依存する．例えば，レガシーなオペレーティングシステムに対して，プロセスマイグレーションを実装しようとした場合，そのオペレーティングシステムをプロセスマイグレーションに対応するように全面的にカーネルのコードを書き換える必要がある．また，VMM がハードウェアレベルで実行コンテキストを移動できるため，一度 VMM に VM マイグレーションの機能を実装してしまえば，その上で動作するすべてのゲスト OS が，結果的にマイグレーションできるのに対して，プロセスマイグレーションの場合，オペレーティングシステムシステム毎にカーネルの改造が必要になる．そのため，VMM による VM マイグレーションのほうがより普遍的な手法と言える．

VM マイグレーションには，コールド VM マイグレーションと，VM ライブ（ホット）マイグレーションの二つに分類することができる．

VM コールドマイグレーション

VM コールドマイグレーションとは，VM を一時的に停止して，その後に，たの物理ホストに移動する手法である．VM コールドマイグレーションの研究は多数存在している．Collective project[25] では，ADSL のような，家庭用の速度が遅いネットワーク環境に最適化された

VM コールドマイグレーションが提案されている．また，Internet Suspend / Resume[26] や，μDenali[27] などがあげられる．

また，複数のプロセスをまとめたドメイン (pods) を他の物理マシンに転送する Zap[28] と呼ばれる技術も存在している．Zap は特殊な Linux カーネルを用いて pods を転送することができる．Zap はどちらかと言うと，ハードウェアレベルで転送を行う VM コールドマイグレーションよりも，プロセスマイグレーションとして色合いが強いと思われるかもしれない．しかし，Zap の場合，プロセスの転送先のマシンと転送元が完全に独立している点が従来のプロセスマイグレーションとは異なる．つまり，既存のプロセスマイグレーションの場合，例え，プロセスを他のマシンに転送し終わったとしても，転送元のマシンが停止してしまった場合，転送したプロセスも停止してしまう．従来のプロセスマイグレーションは，転送先と転送元でプロセスが発行するシステムコールを RPC しているためである．そのため，Zap は移動後のプロセス独立性という観点から VM コールドマイグレーションに分類した．

VM ライブ（ホット）マイグレーション

VM ライブ（ホット）マイグレーションとは，VM を実行したまま，他の物理マシンに VM を移動させる手法である．*Staged migration* や Amoeba[21] の *pre-copy migration (approach)* などと呼ばれることもある．

駆動中の VM のバックグラウンドで，VM を構成するコンテキストを，転送先に送り続け，ある程度のコンテキストが送られた後に，VM を一時停止し，残りのコンテキストをすべて送り，転送元の VM を停止し，転送先の VM を再開するという手法である．厳密に言えば，VM は一瞬停止しているわけであるが，VM の停止時間（ダウンタイム）は極めて小さいため，傍目には VM が停止することなく VM が移動しているように見える．そのため，VM ライブ（ホット）マイグレーションと呼ばれる．

VM ライブマイグレーションとしては，VMware 社は *VMotion* と呼ばれる技術を開発しており，同社の VM マネジメントソフトウェアである VirtualCenter と呼ばれる管理ツールに付属しているツールである．

また，QEMU という様々なアーキテクチャに対応したエミュレータ上に実装された Quasar[29] と呼ばれるシステムも存在する．Quasar のユニークな点は，x86 アーキテクチャモードでエミュレートしている仮想マシンから，全く異なる他の，例えば PowerPC アーキテクチャモードでエミュレートしている仮想マシンへと，アーキテクチャをまたいでマイグレーションを行える点にある．

2.4　遠隔仮想計算機環境

遠隔仮想計算機環境とはサーバ側にある VM を，ネットワークを通して，ユーザに使用させるための技術のことである．サーバ側には VMM がインストールされており，管理者は VM を一括理することが出来る．一方，クライアント側では 2.3 **節**で解説した遠隔計算機操作システムがインストールされており，ユーザは遠隔計算機操作システムを用いて VM を操作することができる．

近年では，この遠隔仮想計算機環境が注目されており，様々な技術が開発されている．代

表的な例としては，VMware 社の VMware View[3] や, Citrix 社の Xen Desktop[4] などが挙げられる．VMware 社の VMware View は VMM として VMware ESX Server[30] を利用し，遠隔計算機操作システムにはシンクライアントである RDP や VNC を用いている．一方 Xen Desktop は Xen[31] と SLIM を採用している．これらの技術は VDI (Virtual Desktop Infrastructure) と呼称されることもある．また，それとは別に，先にあげた，Amazon 社の Amazon EC2[2] といった技術も挙げられる．Amazon EC2 は，インターネットが利用できる利用者であれば，誰でも仮想計算機資源を借り出すことができる基盤システムである．最近では，Eucalyptus[1] と呼ばれる，Amazon EC2 と互換性のあるオープンソースの基盤ソフトウェアも開発されており注目を浴びている．注目を浴びる理由は 2 つある．

第 1 にセキュリティ対策に関する有用性が挙げられる．パーソナルコンピュータの普及により，一人一台の計算機を割り当てることが可能になった．そのため，計算機の管理に関する適切な知識を持たない人が計算機を所有することになり，その結果，適切な管理が行われない計算機からの情報漏洩やウイルス感染といったリスクが社会問題となっている．この問題の原因は，計算機に対して詳しくない人に対して，セキュリティパッチの適応といった専門知識が要求される行為を強要している点である．遠隔仮想計算機環境は，管理者がユーザが使用する VM を一括管理すること可能である．そのため，ユーザに変わって知識のある管理者が VM を管理することが出来る．この種の用途では，先にあげた VDI が用いられることが多い．事実，VMware View や Xen Desktop と言った技術は，この問題に対する技術的な解決策の一つとして開発されている．

第 2 の理由としては，ユーザが計算機資源を借り出すためのプラットフォームとしての有用性が挙げられる．この有用性が論じられる際には**クラウドコンピューティング**と言った言葉が使用されることもある．こういったタイプの遠隔仮想計算機環境上では，ユーザはインターネットを経由して仮想計算機を借り出すことが出来る．このシステムは，計算機資源を必要とする人々が，自ら計算機を用意せずとも，気軽にインターネット経由で仮想計算機を借り出せると言う利点がある．

特に，この，第 2 の理由に関して，述べておかなければならないことがある．

これは，今までのサーバホスティングサービスとは大きく異なるものである．遠隔仮想計算機環境と言う概念が登場する以前から，サービス提供者が物理的なサーバを用意し，インターネット経由でそのサーバを貸し出す，いわゆるサーバホスティングサービスは存在していた．しかし，貸し出す計算機資源が，物理的な計算機から仮想計算機へと変わったことにより，遠隔仮想計算機環境にはサーバホスティングサービスには存在しなかった様々な付加価値が存在している．

サーバホスティングサービスにはない遠隔仮想計算機環境の利点は，サービス運用者側とサービス利用者側の双方に存在する．

サービス運用者側の利点としては，サービス利用者からのハードウェア追加要求がきたときに，物理的に計算機を容易せずとも，ソフトウェア的にすばやく (仮想) 計算機を追加することが出来る．さらに，ハードウェア追加の際にも，ユーザが要求する性能の計算機を追加購入するのではなく，運用者が管理しやすい計算機を導入することができる．なぜなら，VMM の資源多重化により，ユーザの求める性能を持つ (仮想) 計算機を自由に構築できるからである．

一方，サービス利用者の利点としては，先に挙げたように，サービス運用者が物理的な計算機を導入する手間がなくなるため，より細かな時間単位で外部計算機環境を調達，または，破棄できるという利点がある．そのため，特定の時間帯だけアクセス数が急上昇するような，例えば，4 月 1 日のエイプリルフールの日のみ，アクセス数が普段の 10 倍以上あるような Web サービスの場合，遠隔仮想計算機環境から補助 Web サーバを借り出し，それ以外の日は，補助 Web サーバを停止しておくことで，サーバ運用費用のコスト削減といったことが可能である．

これらの利点は，VMM により，仮想計算機の生成/削除/電源 On/off が柔軟に可能な遠隔仮想計算機環境だからこそ実現可能である．これが，今までの，いわゆるサーバホスティングサービスとは決定的に異なる点である．

第 3 章

JavaScript と HTML を用いたシンクライアント VoXY

クラウドコンピューティングの台頭に伴い，VoXY : Vnc over proXY の開発を始めた 2007 年ごろから，既存のソフトウェアをすべて Web ブラウザ上で動作させるというコンセプトのもと，様々な製品が開発された．ユーザはアプリケーションを OS 上にインストールして使用するのではなく，Web 上に作られた Web アプリケーションを使用して，コンピューティングを行うのである．

例を挙げると，Google 社の gmail, GoogleDocs, GoogleSpreadSheet などがあげられる．これらは，Web ブラウザ上でメールクライアントソフト，また，ワープロソフトウェア，表計算ソフトウェア等が動作する，いわゆる，Web アプリケーションであり，これらのサービスでは，ユーザが別途新たなソフトウェアをインストールすることなく，現在普及するほとんどすべてのデバイスに初期導入されている Web ブラウザのみで，これらの実用的なアプリケーションソフトウェアが利用できるものである．そのため，ユーザは他のアプリケーションソフトウェアを新たにインストールすることなく，初期導入されている Web ブラウザを使うだけで，作業に必要なアプリケーションを使用することができる．

また，これとは別に，eysOS, YouOS[12], StartForce と言った Web アプリケーションもあげられる．これらは，Web ブラウザ上に，Gnome や KDE といった統合デスクトップ環境を再現するものであり，すべて，JavaScript で書かれている．ユーザはこれらのソフトウェアを通して，Web ブラウザ上で，表計算，ワードプロセッサといったさまざまな Web アプリケーションを統合的に扱うことができる．

こういった技術が開発され，受け入れられていくにつれ，やがて，Web ブラウザ上ですべてのコンピューティングをこなしてしまおうではないかという新たな潮流が生まれた．これなら，ユーザは専用ソフトウェアを新たにインストールすることなく，既存のほとんどすべてのデバイスに導入済みの Web ブラウザだけですべての作業を行うことができる．新たに，ソフトウェアをインストールしたり，それに伴う面倒な設定を行う必要はない．また，パスワードと ID さえきちんと管理していれば，地球上，いつでもどこでも自分の作業環境を Web ブラウザ上に再現できるという利点もあった．そのため，ユーザは Web ブラウザ上でのコンピューティングを受け入れ初め，その流れは 2013 年の今現在でも続いているといえよう．

このように，Web ブラウザ上ですべてのコンピューティングを行おうという文化が生まれた背景には，2007 年台から始まった Web 技術の急速な技術革新が大きな影響を与えている．当時 Web ブラウザを開発している複数ベンダーが，こぞって独自の拡張機能を自前の Web ブラウザに導入し始めたのである．そのため，Canvas, SVG, VML と言った拡張技術が次々に登場し，Web ブラウザ上に実装されてゆき，今までの Web ブラウザ上での表現の幅が飛躍的に広がったのである．

この文化を画面転送による遠隔仮想計算機転送技術であるシンクライアントシステムにも導入することで，シンクライアントシステムの利用可能性を向上させることができる．後述するが，事実，そういったシンクライアントシステムがいくつか開発されている．Web ブラウザはほとんどすべてのデバイスに初期状態で導入されている一般的なアプリケーションである．その上で，シンクライアントシステムを動作させることが出来れば，より多くの人が，新たなアプリケーションやアプライアンスの導入なしで，遠隔地にある仮想計算機を利用することができ，現在の仮想計算機の利点をより多くの，計算機に対して詳しくない人に対しても提供可能であると考えた．

多くのシンクライアントプロトコルの研究が存在するものの，これらのプロトコルを直ちに Web ブラウザ上に実装することはできない．既存のシンクライアントプロトコルは，すべて，使用する帯域を最大限削減を主眼として設計されている．そのため，転送した画面を描画するクライアント側に，ペイントソフトウェアのような高度な図形描画インタフェイスが備わっていることが前提となっている．例えば，描画した画像の一部分のみを他の画像で上書きする，画像を切り取り別の場所に移動する，画像の一部分の色調を変えるといったインタフェイスを要求している．しかし，当時の Web ブラウザにはそのような高度な図形描画インタフェイスは存在せず，既存のシンクライアントプロトコルを Web ブラウザ上に実装すると，著しく画像の描画速度が低下し，インタラクティブ性の低い，すなわち，使いにくいシンクライアントになる．

そこで，われわれは，JavaScript と HTML といったすべての Web ブラウザで装されている最も基本的なインタフェイスのみで動作する新たなシンクライアントプロトコルを開発した．Web ブラウザは，ある程度の大きさを持つ画像であれば，ある程度の速度で描画できるという性質を持っている．そのため，フレームバッファ（画面）をある程度の大きさを持つタイルに区切り，ルごと転送するというプロトコルを新たに提案する．タイルの中の 1 変更が加えられれば，1 ドット変更されたタイルごとすべて転送する．代わりに，直感的に，既存のシンクライアントプロトコルと比べると，使用するネットワーク帯域は大きなものになる．しかし，われわれは，JavaScript と HTML による，一切のプラグイン等を導入することなく使用できる Web ブラウザ上でのシンクライアントでの利用可能性向上を優先した．

実装したシンクライアント VoXY は，プラグイン等を一切導入することなく，HTTP と JavaScript + HTML のみでインタラクティブ性の高いシンクライアントを実現した．**図 3.1** に示すように，さまざまな環境で利用することが可能である．また，画面をタイル上に分割転送するにあたって，最適なタイルサイズを実験により求め，最適なタイルサイズによって，ワープロ，ペイント，Web ブラウジング，予定記録管理ソフトウェアの操作と言ったコンピューティングにおいて，本章で提案するプロトコルが十分なインタラクティブ性能をもち，

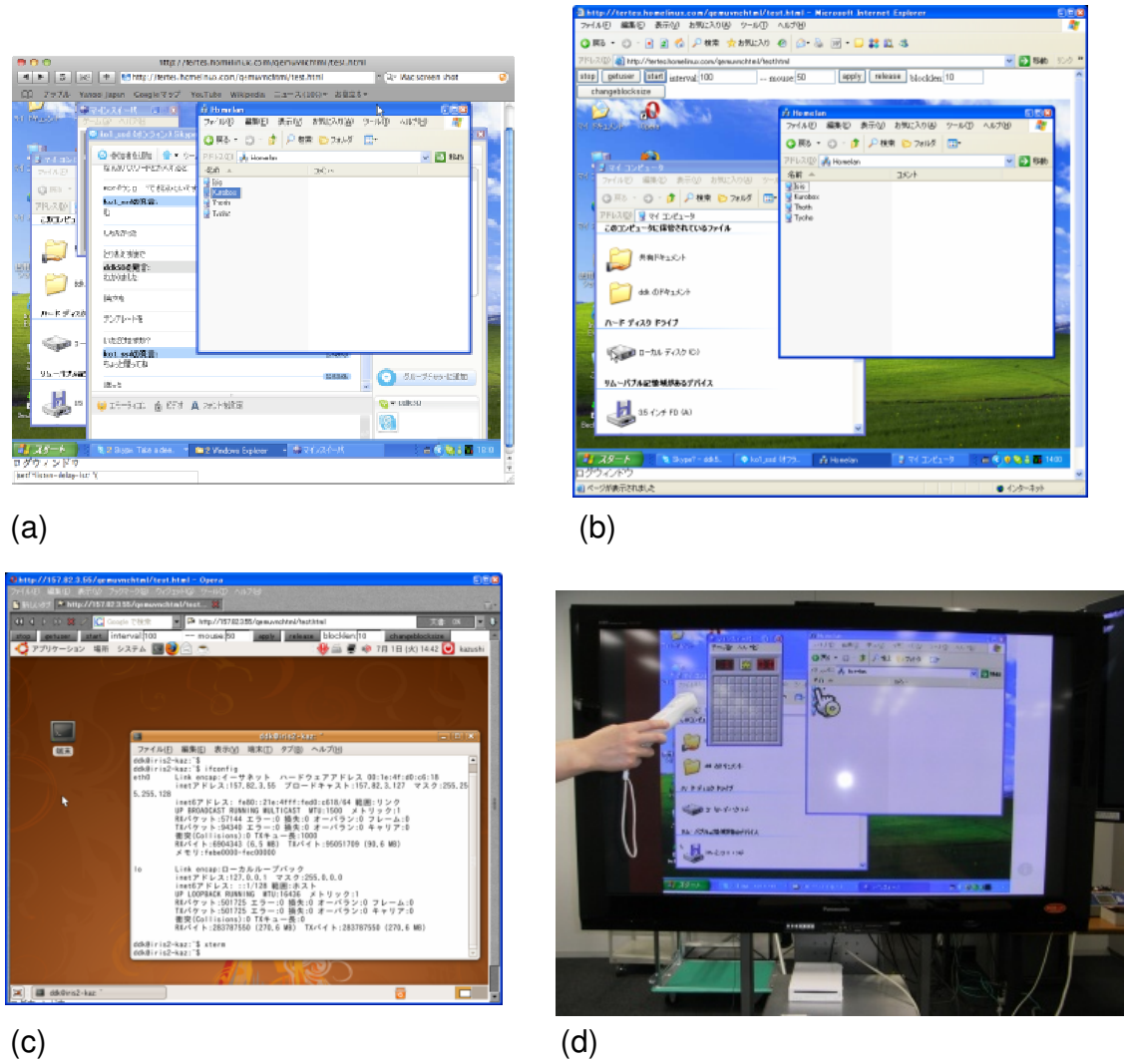


図 3.1. VoXY 使用例の数々: (a) WindowXP のデスクトップを Safari から．(b) WindowsXP のデスクトップを Internet Explorer から．(c) Ubuntu Linux のデスクトップを Opera から．そして，WindowsXP のデスクトップを任天堂 Wii から見ている．

結果，Web ブラウザ上で動作することができるシンクライアントプロトコルの利用可能性が向上させる．

また，VoXY は VMM との親和性が高くなるように実装されている．VoXY は Vnc over proXY の略称であるが，これは，現在の主要な VMM は VM のフレームバッファを VNC プロトコルで外部に出力することができるため，VNC のプロキシという形で実装することで，VMM との親和性を向上させている．しかし，VMM によっては VNC 以外のプロトコルを使用するものもあり，現在の実装では VNC 以外のプロトコルを使用する VMM には対応出来ていないが，これは，あくまで実装上の制約であり，本質的な制約ではない．すなわち，VNC 以外のプロトコルでも対応可能である．

本章の構成は以下のとおりである．まず，3.1 節で遠隔計算機操作システムの 1 つである既

存のシンクライアントプロトコルについて分析を行う．次に，3.2 節にて，既存のシンクライアントプロトコルを JavaScript + HTML 上に実装する際の問題点について論じる．3.3 節にて，今回提案する新たなシンクライアントプロトコルについて論じ，3.4 節にて，3.3 節にて提案したシンクライアントプロトコルの実装について論じる．3.5 節にて，提案したプロトコルがどの程度のインタラクティブ性能を持つかについて評価を行い，利用可能性に関する議論を行う．3.6 節にて，既存のシンクライアントプロトコルとの比較を行い，最後に，3.7 節にて本章のまとめを述べる．

3.1　研究背景

2007 年代から，Web ブラウザ上ですべての作業を行うという潮流が生まれた．この流行の背景には Web 技術の革新があり，Web ブラウザの開発者たちはこぞって独自の拡張機能や新たなプログラミングパラダイム[*1]の開発が活発に行われた．それに伴い，Web ブラウザ上で動作する Google 社の Gmail, GoogleDocs, GoogleSpreadSheet 等が開発された．これは，新たなソフトウェアを導入することなく，Web ブラウザだけで，メールソフトウェア，ワードプロセッサ，表計算ソフトウェアが使用できる．

この技術により，ユーザは新たにアプリケーションをインストールすることなく，初期段階でインストールされている Web ブラウザだけでコンピュータを利用することが可能となり，また，この技術と，クラウド・コンピューティングという中央集権的なデータ管理パラダイムを組み合わせることで，いつでもどこでも Web ブラウザさえ動作する環境があれば，自分を証明することができる何らかの認証機構[*2]を経て，いつでもどこでも自分の作業環境を呼び出すことができるようになった．このような利点から，Web ブラウザ上でのアプリケーション，Web アプリケーションは急速に普及し，今現在では Web ブラウザ上で作業を行うことも珍しくなく成り，多くのユーザに受け入れられている．

そこで，VoXY の目標としては，Web ブラウザ上で動作する Web アプリケーションの特性である，いつでもどこでも Web ブラウザ上であれば動作するという利点を，遠隔仮想計算機転送システムにも導入できると考えた．今までのシンクライアントシステムは，専用のソフトウェアやアプライアンスを導入する必要が有るため，これが，エンドユーザの導入の技術的な障壁となってきた．1 章でも論じたように，これからは，不特定多数の個々の技術レベルが様々な人々が，クラウド上の仮想マシンを利用する機会が増えてゆくことが期待される．その場合，技術的な導入困難さをなるべく低下させるために，ほとんどすべてのデバイス上で初期導入されている Web ブラウザ上でシンクライアントシステムが利用できることが望ましい．

本節では，まず，既存のシンクライアントプロトコルについて解説し，そのプロトコルを Web ブラウザ上に実装する際の問題点について述べ，問題分析を行う．

[*1] Ajax

[*2] ID やパスワードがよく使われる

3.1.1 既存のシンクライアントプロトコル

シンクライアントを実現するためのプロトコルとして代表的なものには X[17], ICA[32], RDP[16], VNC[13], SLIM[14], THINC[33], pTHINC[19] などがあげられる．これらはすべて**画面転送方式**のプロトコルである．この方式では仮想デスクトップの情報はすべてディスプレイに表示される画面データとして扱い転送する．

さらに，画面転送方式のプロトコルは，2 つに分類することができる．X や RDP は画面データとして画面描画命令そのものを転送する[*3]この手法は大きなデータを転送することなく，OS の画面を転送することができるため，高速であるという利点がある．しかし，この手法は使用する OS やウィンドウマネージャに強く依存し，移植が困難であるという問題点があるため，幅広いプラットフォームで使用できないという欠点がある．一方，ICA，SLIM では画面を単純な画像データとして扱う．そのため，特定システムに依存することなく，多くのプラットフォームに対応できるという利点がある．

VNC も**画面転送方式**のプロトコルである．しかし，VNC が利用するプロトコルは RFB protocol[13] として広く公開されており，多くの実装や亜種が存在している．そのため，同じ**画面転送方式**である ICA や SLIM と比べるとより一般的であるといえる．現に，現存のほとんどすべての VMM は VM のフレームバッファを RFB protocol として外部に出力する機能を持っており，そのため，VoXY も RFB protocol を使用した．これは後に詳しく述べるが，VoXY は RFB protocol のみに依存しているものではない．

これらの シンクライアントは専用のソフトウェアや専用ハードウェアとしても提供されている．RDP は Windows 上でリモートデスクトップとして使用できる．VNC や ICA はソフトウェアとして実装されている．また，SLIM はハードウェアとして実装されており，Sun Microsystems から SunRay[14] として販売されている．

3.2 既存のシンクライアントプロトコルの問題点

しかし，当時の JavaScript+HTML は，様々な大きさからなる画像ファイルを任意の場所に高速に描画するという図形描画インタフェイスを持ちあわせていなかった．つまり，比較的単純な，図形描画しかできなかったのである．そのため，既存のシンクライアントプロトコルを直ちに Web ブラウザ上に実装することは不可能であった．

3.1.1 **節**で述べたように，今まで多くのシンクライアントプロトコルが開発されているものの，これらのプロトコルは基本的に，クライアント側に高度な図形描画インタフェイスを要求している．フレームバッファ全画面のうち，一部分のみの更新，フレームバッファの一部を切り取り別の場所に移動，また，フレームバッファの一部分の色調を変えるといった，ペンとソフトウェアのような高度な図形編集インタフェイスを要求していたためである．

もちろん，Web ブラウザ上で動作するシンクライアントソフトウェアは 2007 年以前にも登場していた．しかし，そのシンクライアントは Web ブラウザ上に Java Applet プラグイン

[*3] フォント A を使用して，`(x,y)` の位置に背景色は白で “Hello world” と描画せよ．といった命令．

を導入し，Java Applet 上で使用出来る高度な図形描画インタフェイスを利用したものであった．そのため，純粋に JavaScript と HTML で実現されたシンクライアントソフトウェアではなく，これでは，プラグインという Web ブラウザとは独立したまた別のソフトウェアを導入する必要がある．そのため，純粋に Web ブラウザのみでシンクライアントを実現するという本研究とはまた異なるものである．

3.3　新たなシンクライアントプロトコルの提案

そこで，われわれは，JavaScript+HTML の画像描画特性に合わせた新たなシンクライアントプロトコルを提案する．Web ブラウザ上で図形を描画する手法はいくつかあるため，本節では，まず，すべての手法を列挙した上で，満足するインタラクティブ性能を出しつつ，利用可能性の向上が見込める手法について検討を行う．JavaScript を使用して，Web ブラウザに任意のピクセルデータを描画するいくつかの手法を以下に示す．

表 3.1. 主要ブラウザと拡張タグ一覧

ブラウザ	対応タグ
FireFox	svg, canvas
Safari	svg, (since Safari 3), canvas
Opera	svg, canvas
Internet Explorer	vml

(a) **1 × 1 の DIV タグによる描画**　1 画素に 1 つの DIV タグを割り当て，それを縦横のピクセル数分だけ敷き詰める手法である．画素の色は DIV の color エレメントで指定する．JavaScript による単純なドローツールの実装などは，この手法が使われていることが多い．この手法は単純ではあるが強力であり，ピクセル数が低ければ十分使用に耐える．しかしながら画面データのように，大きなピクセル数を描画する場合にはブラウザへの負荷が高い．簡単な実験を行ったところ，描画速度が極端に落ち，ブラウザが応答しなくなることがわかった．そのため本手法は採用しない．

(b) **拡張タグを使用する**　主要な Web ブラウザでは，JavaScript からピクセルデータを描画するための，拡張タグと API が用意されている．**表** 3.1 に主要ブラウザと対応する拡張タグの一覧を示す．いくつかの拡張タグがあるが，機能自体はどれも同様のものが用意されている．また，HTML5 からは Canvas タグが標準 [34] になる予定である．Canvas タグには，点を打つ，特定の四角形領域を特定の色で塗りつぶす，特定座標の四角形領域を，別の座標へコピーする，といったプリミティブな描画 API に加えて，画像データを，指定した座標へ表示するといったことも可能である．特定の画像データを扱う場合，画像データの指定の方法は，サーバにあるファイルのパスを指定する方法や，Base64 エンコードによるバイナリの表現などがある．実験のために，VNC サーバから送られてくる FramebufferUpdate メッセージごとに PNG ファイルを生成し，生成されたファイルごとに Canvas タグに描いてゆくという簡単なプログラムを作成し

た．その結果，描画速度が遅く[*4]．シンクライアントとして，十分なパフォーマンスを得ることができなかった．

(c) **DOM による IMG タグの動的加工** 画面全体をある程度の大きさをもつ正方形ブロックに分割する．Web ブラウザは IMG タグを使用して，分割された正方形ブロックを画面に敷き詰める．変更箇所があれば，該当する箇所を含むブロックを更新する．この手法は，単純な IMG タグの挿入と変更で実現されているため，少ない実装コストでほとんどすべての Web ブラウザに対応できる．また，試験的実装の結果では，シンクライアントとしてのパフォーマンスも良好であった．

以上，Web ブラウザで使用可能ないくつかの描画手法を検討してきた．その結果，**最終的に，パフォーマンスや現時点でのタグのサポート状況といった総合的な観点から (c) の手法を採用した**．(b) に関しても，(c) と同様にいくつかの正方形ブロックに区切る手法でシンクライアントとしてのパフォーマンスを上げることは可能であると考える．また，将来的には Canvas タグが HTML5 で標準化され，各ブラウザ間での互換性も整うものと思われる．そのため Canvas タグを使用した描画手法に関しては，今後検討したい．

上述の議論の結果，画面全体をある程度の大きさをもつ正方形ブロックに分割して，それぞれのブロックを HTTP の IMG タグを使用して Web ブラウザ上に表示すると言う手法 (c) によって，インタラクティブ性能と利用可能性が両立させることが可能であると結論づけられた．よってこの手法を採用する．

この正方形ブロックの手法においては，各ブロックの大きさをどう決定するかという問題がある．適切なブロックサイズを決定するために予備実験を行った．

実験を行うにあたって，実験用の CGI スクリプトと JavaScript コードを書いた．実験用の CGI スクリプトは，任意の縦横の長さを指定すると，その大きさを持つ PNG イメージと XML データを生成する．XML には生成した PNG ファイルを，ブラウザのどの座標 (`x,y`) に，どの大きさで (`width, height`) 描画するかが書かれている．JavaScript のコードは，CGI スクリプトが生成した XML のデータを元に DOM を操作し，IMG タグを挿入する．Web ブラウザは，IMG タグが挿入されると PNG データを httpd からダウンロードして画面に表示する．

いくつかの異なる縦横の幅を持つブロックで実験を行った．評価環境は，ブラウザに Internet Explorer 7.0 を使用し，httpd は Apache/2.2.8 を使用した．Internet Explorer と Apache 間のネットワークの遅延時間は考えない．画面解像度は 800 × 600 を使用した．

評価は二つの観点から行った．一つは，800 × 600 の全画面を更新したときにかかる時間に着目した．二つは，全画面更新ではなく，800 × 600 の一部分である 10 Kpixel 分のみを更新したときにかかる時間に着目した．この 10 Kpixel という値は，いくつかの実用的なアプリケーションでは，ユーザからのインプットの 50% 近くが 10 Kpixel よりも小さい値の更新しか引き起こさないという Schmidt らの研究 [15] を基にした値である．一番目の評価はブロックごとにそれぞれ 10 回測定して平均値を記録した値である．二番目の評価は一番目の更新時間を元に算出した値である．結果を**図** 3.2 に示す．

[*4] ここでの “描画速度の遅さ” は “Canvas タグそれ自体の描画速度の遅さ” を意味するものではない．この手法による実装の全体を見たときに，シンクライアントとして大幅な遅延を感じるという意味である．

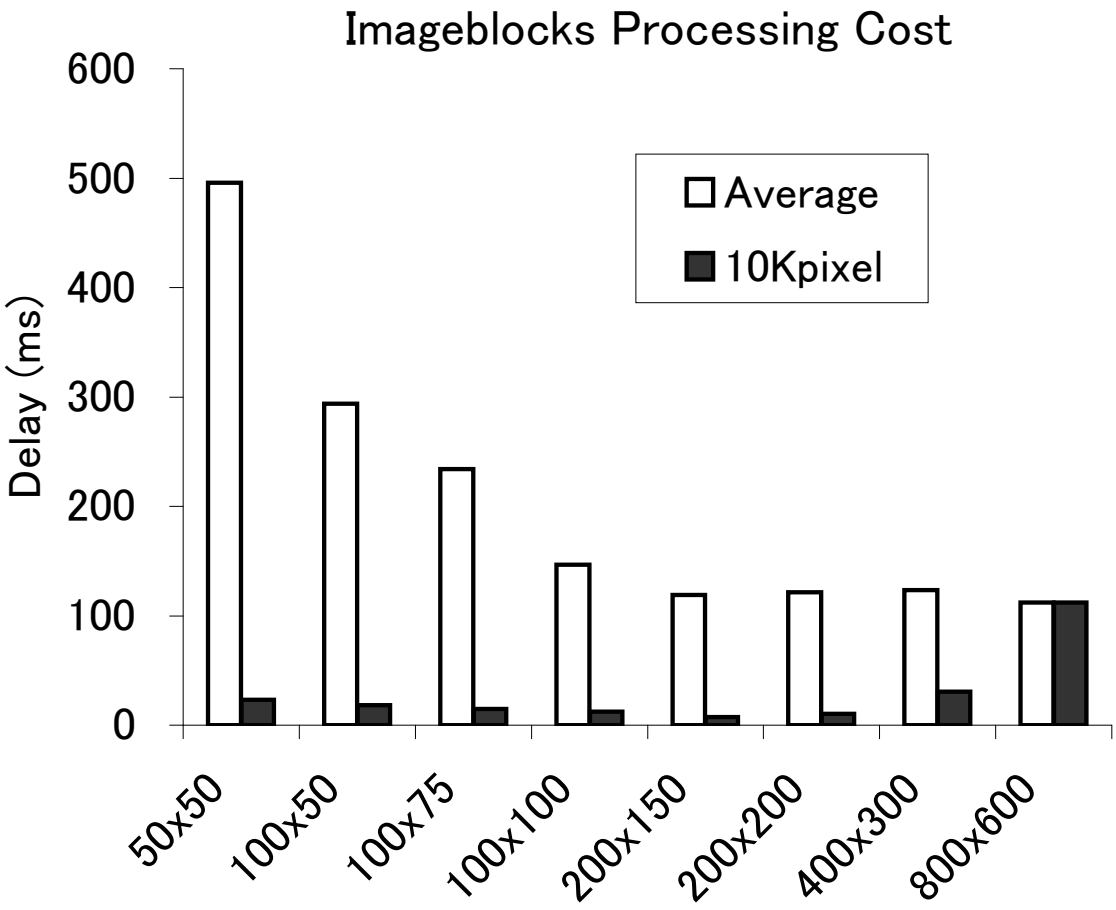


図 3.2. ベンチマーク結果

この結果によると，画面全体の更新時間は，200 × 150 以上の大きさでは大差がない．これに加えて `10 Kpixel` 分の更新時間を考慮すると，最適値は 200 × 150，200 × 200 のいずれかとなる．そこで，今回は正方形ブロックである 200 × 200 を選択した．

まとめると，200 × 200 のブロック上にフレームバッファを分割して転送するという手法を採用した．直感的に，この手法は，従来のシンクライアントプロトコルとくらべると，要求帯域が巨大になるという懸念が生じるが，それに関しては，後の評価の章にて，要求帯域の評価を行う．

3.4 VoXY : Vnc Over proXY の実装

3.3 **節**にて提案したプロトコルを利用したシンクライアントシステムである VoXY の実装を行なった．本章では VoXY の全体像と実装の細かな点について述べる．VoXY のシステム概念図を**図** 3.3 に示す．

VoXY は既存のソフトウェアやデータを引き継げる従来のシンクライアントの特性と，WebVDI が持つ純粋な HTTP のみによる通信の利点をそれぞれ持った新たなシンクライアントである．ユーザはクライアントとして Web ブラウザを使用できる．クライアントは 3.3 **節**

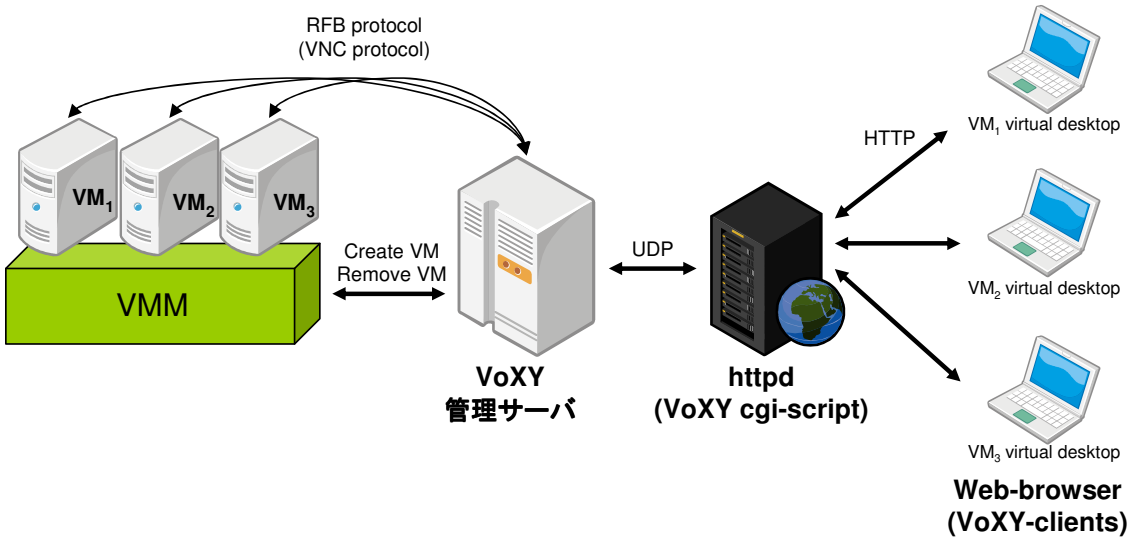


図 3.3. VoXY アーキテクチャの概念図

節で論じたように，JavaScript と HTML のみで実装されている．上述したように，利用可能性を犠牲にしないよう他のプラグイン等は一切必要としない．

VoXY と VM との接続には VNC で利用されている RFB protocol を使用する．VM の画面データを RFB protocol として出力できる VMM は多い．代表的なものとして．Xen[31], QEMU[35], KVM[36], VMware[30] などが挙げられる．また，これ以外にも RFB protocol 3.8[13] に準拠している VMM であればどれでも VoXY と接続可能である．ただ，VoXY は RFB protocol のみに依存するものではない．必要と有らば，この部分は，他のシンクライアントプロトコルであったも容易に対応することが可能である．

さらに，VoXY はマルチユーザへの対応や柔軟な拡張性を持っている．複数人のユーザそれぞれに 1 台ずつ VM を割り当てたり，複数人のユーザに対して特定の VM を共有させることも可能である．httpd は cgi スクリプトの実行に対応しているものであればどれでも使用することができるため，ロードバランサなども既存のものが使用できる．VMM と管理サーバは分離した設計になっているため，VMM の数を増やすことも容易である．このように，各コンポーネントを分離したアーキテクチャを採用したことで，VoXY 管理者は柔軟でスケーラブルなシステム構成を行うことが可能である．

3.4.1 VoXY のコンポーネント

本節では，VoXY を構成するコンポーネントの解説と，VoXY が使用するプロトコルや処理の流れについての解説を行う．

VoXY は大きく分けて，VMM の管理やユーザ管理などを行う VoXY 管理サーバと，Web ブラウザ上で動作する，JavaScript で実装された VoXY クライアント，VoXY クライアントと VoXY 管理サーバを接続するための，VoXY cgi スクリプトの 3 つのコンポーネントに分けられる．

VoXY 管理サーバ

VoXY 管理サーバは，VMM を管理する VoXY VMM Manager と，VoXY Window Manager の 2 つのコンポーネントに分けられる．それぞれの詳細を以下に示す．なお，VoXY 管理サーバの実装は C++ で行われており，4,000 行ほどである．

VoXY VMM Manager　VoXY VMM Manager は VMM に対して，仮想マシンの起動，新規作成，削除などの命令を発行する．仮想マシンの画面データを RFB protocol で出力できる VMM は多数存在しているものの，仮想マシン自体の制御[*5]に関して，VMM は統一されたインタフェースをもっているわけではない．そのためこの部分は VMM ごとに異なる実装を行う必要がある．本論文では QEMU 用の VMM Manager の実装を行った．

VoXY Window Manager　VoXY Window Manager は主に，ユーザの管理と画面データの管理を行う部分である．VoXY VMM Manager が管理する仮想マシンに対してユーザを割り当てたり，画面データの取得などを行う．

VoXY CGI スクリプト

VoXY cgi スクリプトは，httpd 上で動作する VoXY クライアントと VoXY 管理サーバ間の中継を行うための CGI スクリプトである．通信は UDP ソケットを使って独自のプロトコルで行う．UDP ソケットで通信を行う理由は，httpd からの要求に対して CGI の実行プロセスが立ち上がり，CGI スクリプト実行後は CGI の実行プロセスは終了するという CGI の性質を考えると，TCP ソケットによる通信では，ソケットの接続/切断が連続して発生するため，TCP のコネクションのコストが高いと考えたからである．

今回の実装では，Common Lisp で書いた CGI スクリプトを利用することで，VoXY 管理サーバと httpd の通信を行っている．しかし，この部分は，httpd と VoXY 管理サーバ間の通信を行える機構を備えたものであれば，どのようなものでもよい．また今回は，VoXY cgi スクリプトと VoXY 管理サーバを同一計算機内においているため，パケットロスの可能性は低い．そのため，UDP による通信を選択した．しかし，二つが分離し信頼性を持たせなければならなくなったとき，TCP に変更することは容易である．

VoXY クライアント

VoXY クライアントは Web ブラウザで動作するクライアントである．これは，仮想マシンの作成，削除といった VMM のインタフェースと，仮想マシンの画面データの表示や，仮想マシンに対して送信する Keyboard や Mouse のイベント取得などを行う．通信は Ajax を用いて，非同期に httpd 上にある VoXY cgi スクリプトへデータを送る．このクライアントは JavaScript を使用して実装されており，400 行ほどで実現されている．コードは多くの Web ブラウザが共通して持つ機能のみを使用しているため，多くのブラウザ間での互換性は高い．

[*5] パワーオン，リセット，仮想マシンの作成や削除といった操作．

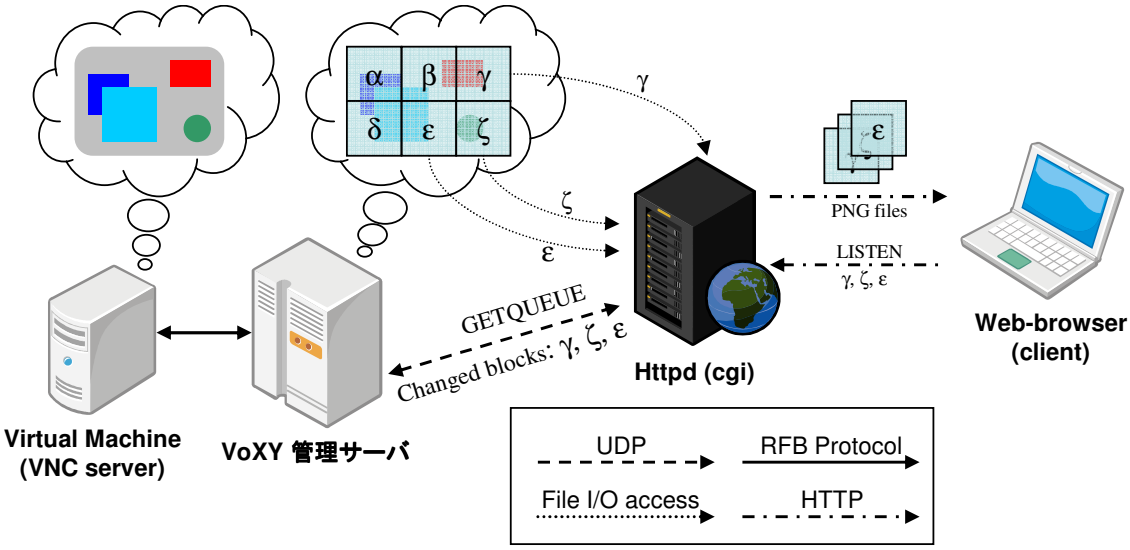


図 3.4. VoXY プロトコルの概略図

3.4.2 (VNC) RFB Protocol の概要

VNC の内部では RFB Protocol（Remote Frame Buffer Protocol）[13] と呼ばれるプロトコルが使用されている．現時点で最新の RFB Protocol は version 3.8 である．VoXY は RFB Protocol 3.8 に準拠するプロトコルを使用する VMM であれば，どれでも使用可能である．

RFB protocol はきわめてシンプルなプロトコルであり，クライアントをステートレスに実現可能である．画面データのフォーマットは，“与えられた x,y 座標に横, 縦分だけピクセルデータを描画する” という単一の描画規則に基づいている．

また，RFB protocol は Client-pull 型のプロトコルである．VNC クライアントが VNC サーバに向かって FramebufferUpdateRequest メッセージを送ると, VNC サーバから FramebufferUpdate メッセージが返る．VNC クライアントは FramebufferUpdateRequest メッセージの送信間隔を調整することができる．これにより，使用する回線状況に応じて通信トラフィックを調整することができる．

3.4.3 VoXY プロトコルの概要

3.4.1 節での議論を元に VoXY プロトコルを設計した．プロトコルの概略図を図 3.4 に示す．

管理サーバは VNC サーバへ TCP コネクションを確立する．VNC サーバ 1 台につき 1 つのスレッドを生成して，VNC サーバに向かって `FramebufferUpdateReqest` メッセージを送信する．その後は，VNC サーバからの返答を待つ．

VNC サーバから送られてきた `FramebufferUpdate` メッセージの画面データは，いったん管理サーバ内のメモリバッファに蓄えられる．ここで，画面データ全体は，複数の正方形タ

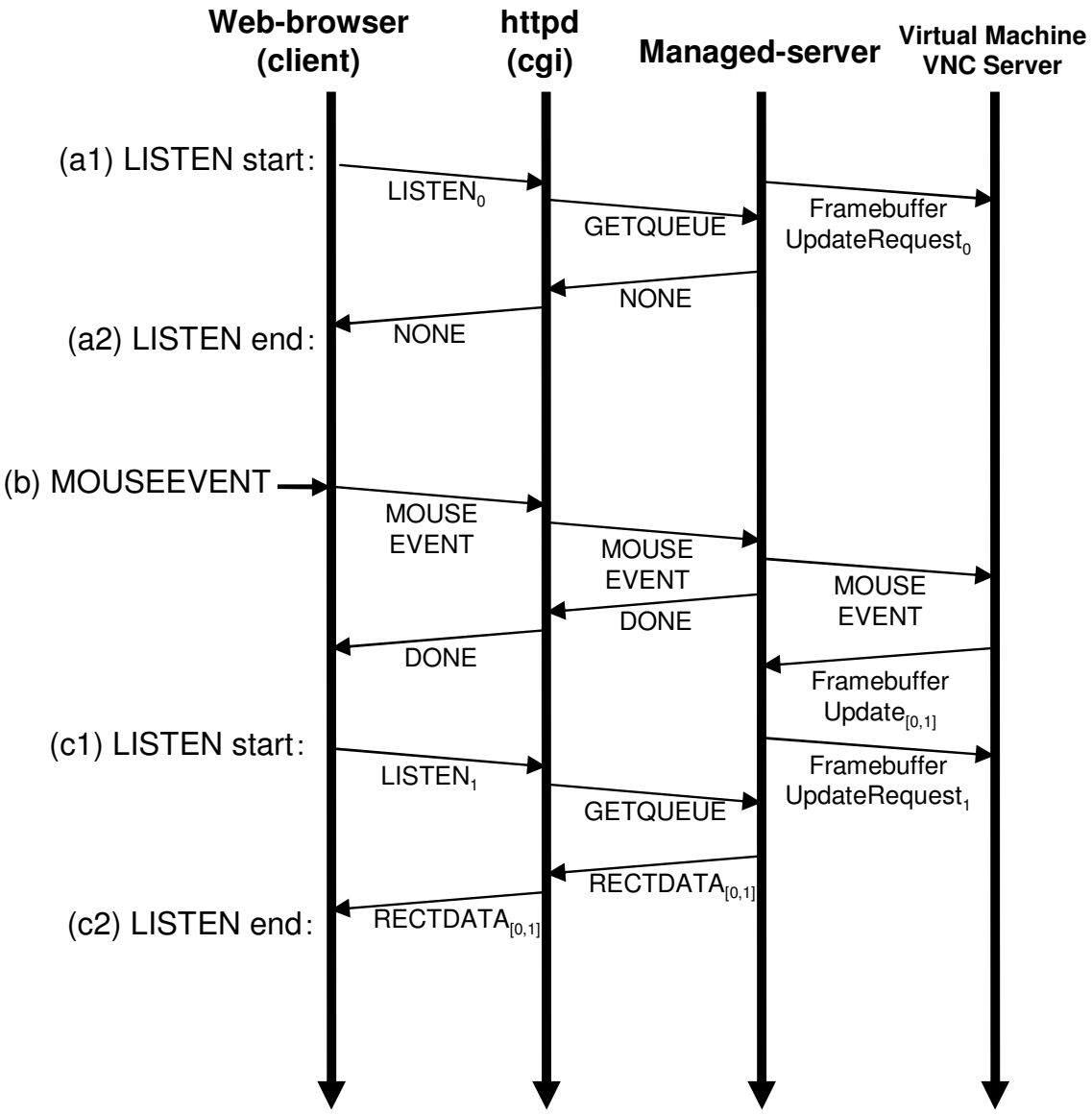


図 3.5. VoXY プロトコルシークエンス図

イルに分割されて管理される (**図 3.4**：$\alpha, \beta, \gamma, \delta, \epsilon, \zeta$)．それぞれの正方形タイルには，固有の ID と，ファイルディスクリプタが与えられる．つまり，各ブロックはそれぞれ 1 つの PNG ファイルに対応付けられる．また，それぞれのタイルの辺の長さは，**3.3 節**で議論したとおりである．管理サーバが VNC サーバから `FramebufferUpdate` メッセージを受け取ると，そのデータは，更新箇所を含むブロックへ送られる．管理サーバは変更があったブロックの内容を PNG ファイルに圧縮して書き出す．最後に，変更箇所があったブロックの ID を QUEUE に書き出す．

3.4.4 イベントドリブンなポーリング

図 3.5 に VoXY プロトコルのシークエンス図を示す．クライアントとなる Web ブラウザ側は，Ajax による非同期通信を使用して，定期的に httpd へとポーリングを行う

（**図** 3.5 : (a1) の `LISTEN end` から，(c1) の `LISTEN start` までがポーリングの間隔である）．これは，サーバ側からの擬似的なプッシュを実現するためである[*6]．

なお，ポーリングの間隔は 100ms である．ポーリングの際に送信するコマンドは，`LISTEN` コマンドである．クライアントが発行した `LISTEN` コマンドを受け取ると，httpd 上で実行されている cgi スクリプトは UDP ソケットを使用して，管理サーバへ問い合わせを行う．このときに使用されるコマンドは `GETQUEUE` である．`GETQUEUE` コマンドに対し，VoXY 管理サーバは，キューに入っている変更のあったブロックのファイル名を返す．（**図** 3.4:γ, ζ, ϵ）このとき管理サーバが返す内容は，変更点があったブロックに対応する PNG ファイル名である．PNG ファイルのバイナリデータは，管理サーバがファイルに書き出す．

最終的に，クライアントとなる Web ブラウザ側は，`LISTEN` コマンドの送信の結果として，変更のあったブロックの PNG ファイル名のリストを得る．クライアントは Web ページの DOM を操作して，変更のあったブロックを表示する IMG タグの `src element` を変更する．`src element` が変更されたことで，Web ブラウザは PNG のファイルのダウンロードを httpd に要求する．PNG ファイルのダウンロードが終わると，画像データがブラウザ上に表示される．

3.4.5 ユーザイベントの取得と送信

ユーザイベント（Keyboard，Mouse など）は，`LISTEN` イベントとは独立して送信される．マウスイベントが発生すると (**図** 3.5 : (b) の `MOUSEEVENT`）マウスイベント監視用のループがその値を送信する．マウスイベント監視のループは，JavaScript の `SetTimeout()` 機能を使用して，擬似的なスレッドを実現している．そのため，ループ間隔が短すぎるとブラウザへの負荷が高くなり応答しなくなる．ここでは，ループ間隔は 100ms としている．これは Schmidt らの先行研究が示した，アプリケーションに入力されるユーザイベントの 70% が 10Hz 以下である [14] という結果をもとにした値である．

3.4.6 VoXY VMM インタフェース

3.4.1 **節**節にて，仮想マシン上での KVM データをどのように取り扱うかに関する議論を行った．本節では，仮想マシンを管理する VMM へのコントロールを行う VoXY VMM Manager について解説する．

VoXY VMM Manager は VMM に対して，仮想マシンの起動，作成，破棄，コピー，一時停止，などのコマンドを発行する．ここは使用する VMM ごとに異なる実装が必要である．

[*6] ディスプレイの画面更新は，必ずしもユーザのアクションに対して行われるわけではないため，ポーリングによる監視が必要である

今回作成した VoXY VMM Manager は QEMU のみをサポートしており，`system()` 命令を使用して仮想マシンの起動を行っている．

また，多数の異なる VMM を統一したインタフェースで操作するための機構として libvirt が提案されている．これは，VMM 上で動作する仮想マシンの制御を抽象化したライブラリであり，サポートする VMM の数も多い．このライブラリが提供する API を使用すれば，統一したインタフェースで多くの VMM をコントロールすることができる．VoXY VMM Manager の libvirt 対応に関しては，今後検討したい．

3.4.7　進化した VoXY：N 面-VoXY

表 3.2. ベンチマークで使用したアプリケーション一覧

ジャンル	アプリケーション名
ドローソフトウェア	GIMP
Web ブラウザ	Mozilla FireFox
ワードプロセッサ	OpenOffice Writer
パーソナルマネジメント	KAddressBook

表 3.3. ユーザへ課したタスクの一覧

アプリケーション	ユーザに課したタスク
GIMP	人間の顔を描け
Mozilla FireFox	特定の Web サイトを閲覧せよ
OpenOffice Writer	SICP[37] の「はじめに」をタイプせよ
KAddressBook	自分のアドレス帳を作れ

3.5　評価

本節では，VoXY の評価を行った結果を記す．評価環境は以下のとおりである．

管理サーバと httpd，VMM はともに Intel(R) Core(TM)2 Quad Q6600 @ 2.40GHz, 2 Gbyte RAM, Intel Gigabit NIC からなるマシン上で動作させた．OS は Ubuntu Linux8.04 である．なお，VMM は QEMU 0.9.0 を使用しており，kqemu アクセラレータを使用した．その上でゲスト OS として Fedora Core 6 を動作させたものを使用している．httpd は Apache/2.2.8 を使用した．クライアント側もサーバと同じマシンを使用して，OS は Windows XP SP2 である．クライアントとなる Web ブラウザは，Internet Explorer 6 を p 使用した．クライアントとサーバをつなぐハブは Gigabit HUB を使用している．サーバ側からクライアント側へ ping コマンドで測定したネットワークの遅延平均は 0ms であった．そのためネットワークの遅延はないものとして扱う．

3.5.1 評価手法

システムを評価するにあたって，われわれは，まずはじめにいくつかの実用的なアプリケーションを選定した．選定基準はシンクライアントシステム上で利用してもストレスを感じにくいと思われ，かつ実用的なソフトウェアを選択した．3D ゲームや，CAD，CAM, 動画編集ソフトウェアといった，極めて高いインタラクティブ性が求められるソフトウェアは対象外である．

選定したアプリケーションを**表** 3.2 に示す．これらのアプリケーションを，それぞれ 1 回ずつユーザに 5 分間使ってもらい，その時の LISTEN コマンドが発行され，その結果として RECTDATA が返ってきてから，ブラウザにその RECTDATA が実際に描画されるまでの遅延時間を測定した．

図 3.5 で示すと，(a1) から (a2) 間の時間に，ブラウザが PNG ファイルをダウンロードしてきて表示する時間を足した値である．なお，VMM による遅延影響がどの程度あるかを測定するため，これらのアプリケーションを QEMU 上で動作させたときと，QEMU 無しで，アプリケーションを直接サーバ上で動作させ，X の VNC サーバを使用したときとを比較した．つまり，**表** 3.2 のアプリケーションにつき，VMM 有りと無しの 2 回づつ評価を行った．また，ユーザにはアプリケーションごとに課題を提示した．課題の詳細を**表** 3.3 に示す．なお，いずれのアプリケーションにおいても，画面解像度は 800×600 である．

3.5.2 VNC サーバにおける遅延測定

はじめに，VNC サーバを使用したときの測定結果を**図** 3.7 に示す．グラフを見たとき，全体的に時間がかかっているアプリケーションは GIMP であったこれは，絵を描くときに，更新される部分が多かったからと考えられる．また，FireFox も，画面スクロールや他のサイトへのジャンプなどといった操作で，画面の大部分が更新される割合が多いため 2 番目に遅延時間が大きい．一方，最も時間が少ないアプリケーションは word である．これは，キータイプによって更新される画面が，局所的なものに過ぎないためである．

全体的に見てもっとも時間がかかっている GIMP でも，全体の 94% が 50ms 以内の遅延時間で収まっている．また，もっとも更新時間の少ない word は 99% が 50ms 以内の遅延時間で画面更新を行えていることがわかる．

3.5.3 VMM 使用時における遅延測定

次に，QEMU を使用したときの遅延時間を測定した．結果を**図** 3.6 に示す．当初予想では，VMM より VNC を使用したほうが全体的に遅延時間が短くなると考えていた．しかし，結果は反対であった．この原因は，使用した RFB protocol の実装が異なる点にあると考える．

同様にもっとも時間がかかっているのは GIMP であり，全体の 99% が 50ms 以内の遅延時間で収まっている．また，もっとも更新時間の少ない word はほぼ 100% が 50ms 以内の遅延時間で画面更新を行えていることがわかる．

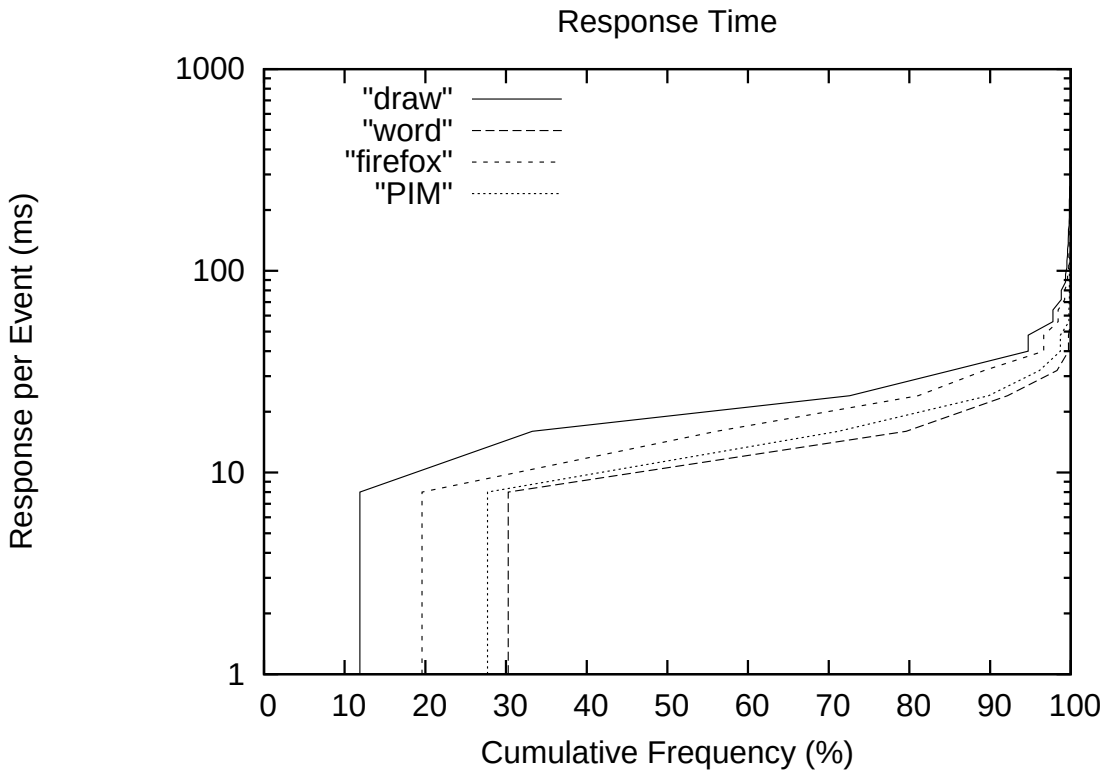


図 3.6. LISTEN イベントから，Web ブラウザ上の画面がアップデートされるまでの時間をまとめた累積度数分布図．ヒストグラムの階級は 8ms．(QEMU の場合)

3.5.4　遅延測定結果の考察

3.5.3 **節**と 3.5.2 **節**のまとめを示す．結果から VNC サーバを使用したときと，VMM を使用したときとで，VMM の部分がボトルネックなるような現象は観測されなかった．

シンクライアントの性能評価では対話応答性を評価することが重要である．ユーザインタフェースの研究によると，対話応答性の評価にあたっては，ユーザのイベントに対する更新の応答時間がよい検証値になり，人間が遅延を感じ始める時間は，50-150ms ぐらいから始まるという研究結果 [38] がある．3.5.3 **節**節と 3.5.2 **節**節で行った評価値 (これを Lt と表記する) では，LISTEN コマンドが発行された時点で，ユーザのイベント発生時から見て，すでに遅延が発生している．一方，対話応答性の評価に必要な値は，**図** 3.5 で示すと，(b) から，(c2) が終わってその後 Web ブラウザの画面更新が終わるまでの時間 (これを Rt と表記する) である．そのため，対話応答性となりえる遅延時間は，LISTEN のポーリング間隔を考慮して，最悪でも Lt+100ms 以下である．

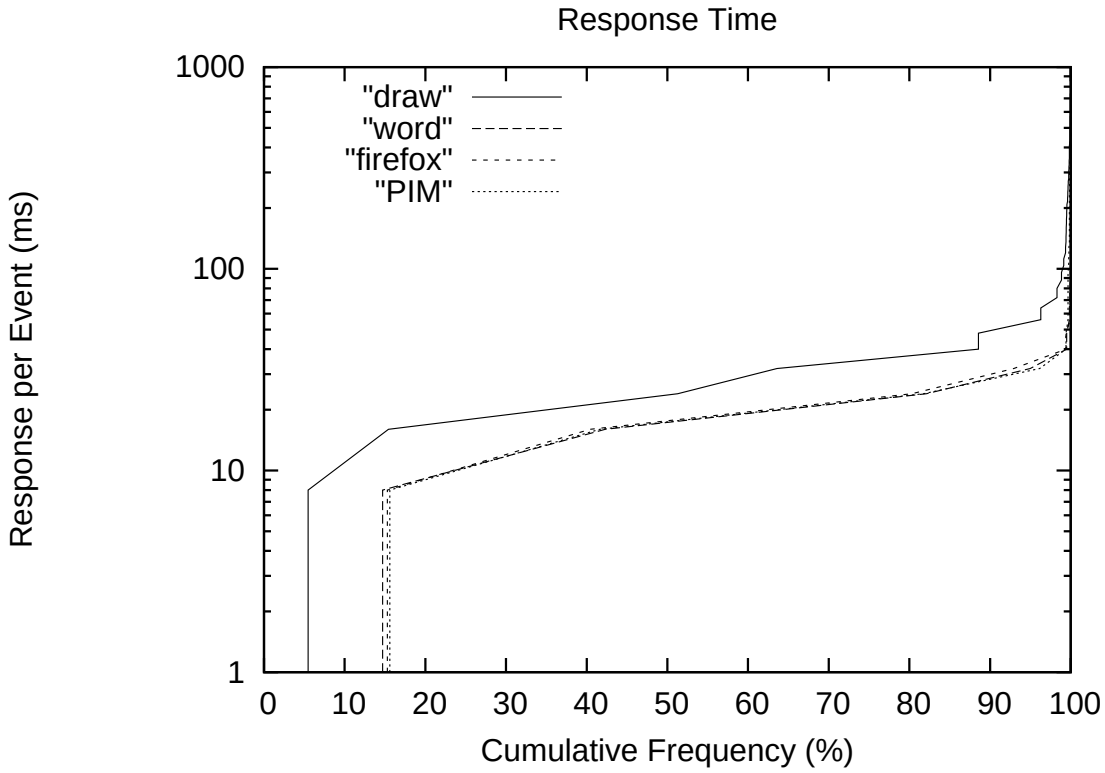


図 3.7. LISTEN イベントから，Web ブラウザ上の画面がアップデートされるまでの時間をまとめた，累積度数分布図．ヒストグラムの階級は 8ms．（VNC サーバの場合）

3.5.5 帯域測定

次に，各アプリケーションを使用したときの使用帯域を測定した．この評価では，解像度は 640 × 480 を使用しており，アプリケーションも 3.5.1 **節**で使用したものとは少々異なっている．**図** 3.8 に結果を示す．最も画面更新量が多い nautilus (firefox) でも，1.076Mbps 程度の帯域しか使用していない．また，kword(word) にいたっては 0.420Mbps 程度である．

3.5.6 帯域測定結果の考察

帯域測定の考察を示す．参考までに，一般的な家庭用ブロードバンド回線における下り帯域値を測定してみると，約 19.027Mbps であった．このことから，VoXY は今日普及する家庭用ブロードバンド回線上であれば十分に使用可能であることがわかる．そのため，VoXY はインターネット利用者がすべて利用者となりうる，広く開放されたサービス基盤として使用できることがわかった．

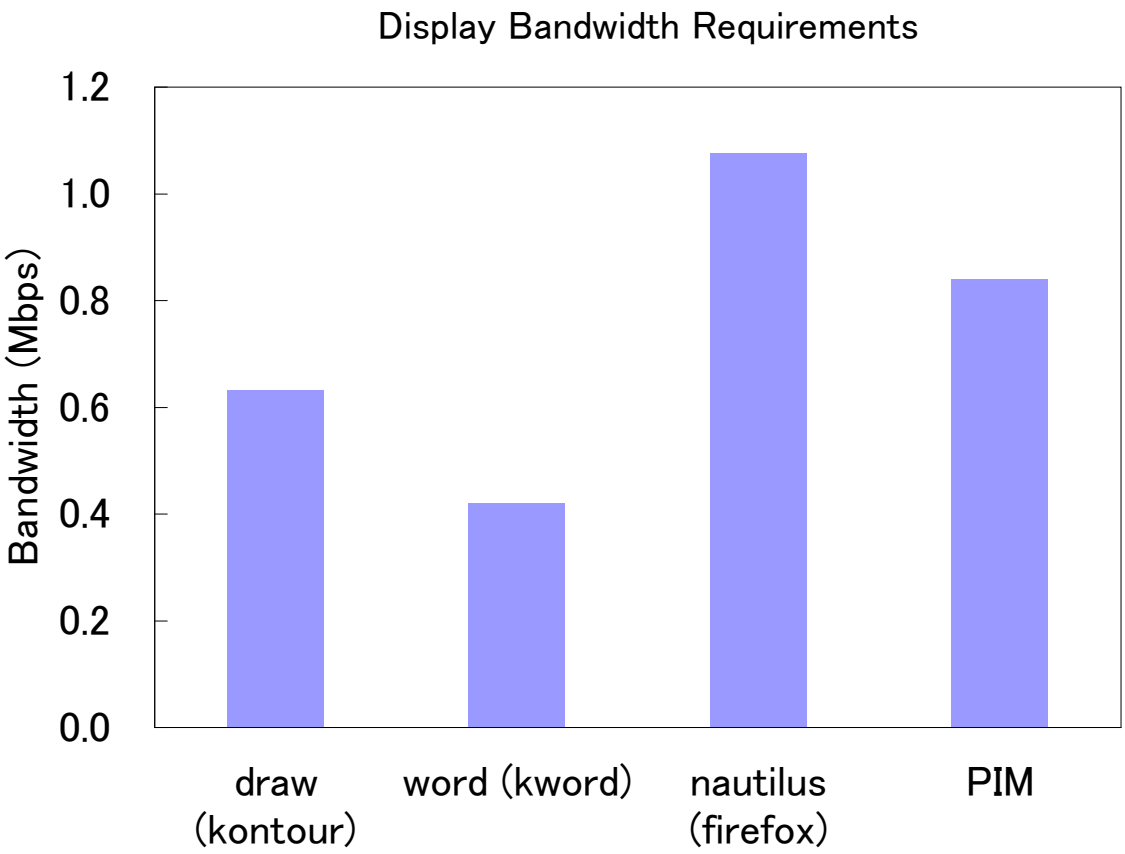


図 3.8. ベンチマークアプリケーションごとの平均消費帯域

3.6 関連研究

数多くのシンクライアントプロトコルが開発されている．X[17], ICA[32], RDP[16], VNC[13], SLIM[14], THINC[33], pTHINC[19]．しかし，これらのシンクライアントプロトコルは主に使用帯域削減を主眼にプロトコルを開発しており，シンクライアントのクライアントに高度な図形編集インタフェイスを要求する．具体的には，画面の一部分のみの変更，画面の一部を切り出した後移動，また，画面の一部の色調を変えるといった，言わば，ペイントソフトウェアのような高度な図形編集インタフェイスを要求しているのである．一方，VoXY は，ある程度の大きさを持つ画像ファイルを表示するだけであれば，高速に描画できるという Web ブラウザの画像描画システムの特性を生かして，ある程度の大きさを持つブロック（タイル）状の画像を転送する新たなシンクライアントプロトコルを提案したことが異なる．直感的に，転送する帯域幅が，従来のシンクライアントプロトコルに比べると巨大になるとかんがえられるが，評価の結果，今日普及する高速通信回線であれば十分に対応可能であることが判明した．

また，VoXY 全体を 1 つのソフトウェアとして見た場合，いくつか類似のシステムが存在しており，それについても述べていく．

FLOZ: Free Live OS Zoo[39] は，QEMU と Java Applet 版 VNC クライアント (VNC Viewer) を使用し，Web ブラウザ上にてゲスト OS をブラウザ上で動作させることのできるシステムである．利用者はサービス提供者があらかじめ用意した種々の OS イメージを自由に選ぶことができ，選択した OS の画面データが Web ブラウザ上の VNC Viewer に転送される．FLOZ は Web ブラウザ上で，プラグインとして動作する VNC クライアントを，QEMU が操る VNC サーバの TCP ポート (5900 版以降) に直結するという形を取っている．そのため，HTTP プロトコルの上で全てを運用することができる本システムとは性質が異なる．

The JPC Project[40] はオックスフォード大学によって開発されたシステムであり，Java Applet 上で x86 エミュレータを動作させ，その上で各種ソフトウェアを動作させようという試みである．本プロジェクトはテスト段階であるが，既に Web 上で FreeDOS が動作するデモを見ることができる．The JPC Project で実装された x86 エミューレータは，bochs と同じようにソフトウェア上で ISA レベルでのエミュレートを行うものである．そのため VMM を使用する本システムとは性質が異なる．

3.7 JavaScript と HTML を用いたシンクライアント VoXY のまとめ

本章では，一般的なコンピュータに導入済みである Web ブラウザをクライアントとすることで，導入が容易な遠隔計算機操作システムである VoXY: Vnc Over proXY について論じた．すでに Web ブラウザ上で動作するシンクライアントシステムは存在するが，プラグインを導入剃る必要があり，規格化された JavaScript と HTML のみでは動作しない．そのため，プラグインを導入するひつようがあるため，すべての環境で動作するわけではなく，利用可能性が低かった．

しかし，JavaScript と HTML のみでインタラクティブ性の高いシンクライアントを作成するのは困難である．なぜなら，VoXY を開発した 2007JavaScript と HTML には高度な図形描画機能が付属していなかった．そのため，従来のクライアントプロトコルである，インクリメンタルに差分のみを転送するプロトコルでは十分なインタラクティブ性能を出すことができないことが判明した．

そこで，本研究では，JavaScript と HTML で十分なインタラクティブ性を出すことができるシンクライアントプロトコルを設計した．変更点の差分のみを転送するのではなく，画面のフレームバッファをある程度の大きさ (200 pixel pixel) のタイルに分割し，変更点が含まれる部分のタイルごと転送するというプロトコルを提案する．結果，3D ゲームや，CAD，CAM，動画編集ソフトウェアといった極めて高いインタラクティブ性能を持たない，オフィスワーク系のソフトウェア程度であれば，十分利用に耐えるシンクライアントシステムを開発することに成功した．また，従来の差分のみのプロトコルよりは要求帯域は高いものの，今日普及する家庭用の高速な回線であれば十分対応できる程度であると考えている．

また，VoXY は VMM との親和性が高くなるように設計されており，VNC のプロトコルと HTTP の変換プロキシソフトウェアとして実装されている．なぜなら，1 章でも述べたとおり，VoXY はわれわれが新たに構想した遠隔計算機環境の一部として開発されているためで

ある．

本章の最後に，今後の課題について述べる．3.4.1 **節**節で述べたとおり，現在の VMM Manager は VMM として QEMU しかサポートしていない．そのため，QEMU 以外の VMM もサポートできるようにしたい．また，VMM として Xen を使用して，システムが一度にどの程度の数のユーザと仮想マシンをサポートできるかといった，スケーラビリティの観点からも評価を行いたい．

第 4 章

Linux ドライバの Windows 移植による WinKVM の実装

仮想マシンモニタ (VMM:Virtual Machine Monitor) の成熟に伴い，計算機の利用者に対し，データセンタ内にある VMM が管理する仮想マシン (VM:Virtual Machine) を割り当てるシステムが普及している [2].

このような基盤システムでは，VM はデータセンタ内に固定さえる．そのため，ユーザは 3 章で論じたシンクライアント VoXY を使って，ネットワーク越しに VM を操作する必要がある．しかし，計算機の利用形態は多種多様であるため，シンクライアント VoXY を用いたネットワーク越しでの利用でそれら全ての要求をかなえることは不可能である．例えば，高い対話応答性が求められる CAD ソフトウェアや動画編集ソフトウェアを利用するために一時的に手元の計算機資源を活用したいという要求や，ネットワークが使えない環境下でも VM を使用したいという要求に対して柔軟に対応することが出来ない.

VM ライブマイグレーションと呼ばれる技術を活用すれば，必要なときにはリモートにある VM を，シームレスに手元のコンピュータに移動させる事ができる．上述したようにシンクライアント VoXY だけでは幅広い利用シーンに対応できない．例えば，コンピュータゲーム，CAD (Computer Aided Design) ,CAM (Computer Aided Manufacturing) と言ったアプリケーションや，作業環境である VM を，出張等でネットワークが断絶する環境に一時的にダウンロードして運用することが出来ない．VM ライブマイグレーションであれば，実行コンテキストを維持したまま VM を移動させることができる．そのため，シンクライアント VoXY に加えて，VM ライブマイグレーションを利用すればより便利な仮想計算機の運用が可能となる.

しかし，現状の VM ライブマイグレーションの技術は，図 4.1 に示すように，同一の VMM 間でのみしか VM ライブマイグレーションが出来ない，そのため，異なる VMM 実装の間で VM ライブマイグレーションは基本的に不可能であるという問題がある．これは，VM マイグレーションの利用可能性を低下させている．エンドユーザが使用する OS（Windows) とサーバ用途向けの OS (Linux) 間のライブマイグレーションを実現することで，利用可能性の向上を実現することができる．ライブマイグレーション技術を使って，ユーザが仮想マシンを自由に移動させるためには，広く普及している Windows と Linux 上で動作する VMM 間で VM

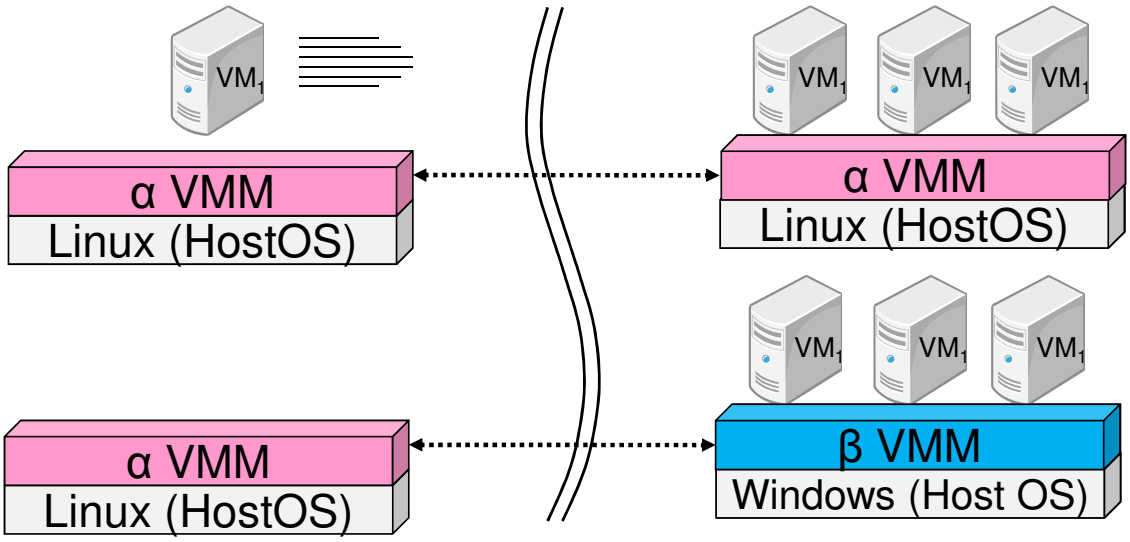


図 4.1. 同一の VMM 間でなければ VM ライブマイグレーションは不可能

ライブマイグレーションを実現することが必須である．そのため，Linux と Windows 間で動作する VMM 間で VM ライブマイグレーションを実現することで，シンクライアント VoXY では対応不可能なソフトウェアを利用したい場合，手元にある Windows がインストールされた計算機に，普段自分が使っている作業環境である仮想マシンを直ちにダウンロードすることができ，利用可能性を向上させることができる．

そこで，本研究では，Hybrid 型 Hypervisor である Linux/KVM に着目し，これを Windows に移植すれば，Windows と Linux 間のライブマイグレーションを実現するという手法を考えた．Windows はコンシューマ用途として広く普及するオペレーティングシステム（OS）である．一方で，Linux はサーバ用として広く普及する OS である．これらの OS の間で VM ライブマイグレーションを可能にすることで，既存の同一 OS 間でしかライブマイグレーションが出来なかったことによる利用可能性の低下という問題を解決する．

しかし，Linux/KVM の核となる部分は Linux デバイスドライバとして実装されている．一般論として，デバイスドライバはオペレーティングシステムに強く依存するプログラムである．そのため，ユーザ空間で動作するプログラムに比べると他のオペレーティング・システムに移植するのが難しい．そのため，KVM のコアである Linux デバイスドライバを Linux から Windows に移植するのは人的なコストが高い．

そこで，われわれは，“簡単” かつ “性能劣化を引き起こすことなく”Linux/KVM ドライバを Windows に移植する手法を開発した．具体的には，Linux のエミュレーションレイヤを Windows カーネル空間上に構築し，その上で Linux デバイスドライバを動作させる手法を取った．エミュレーションレイヤで，異なるアーキテクチャ A 向けに書かれたプログラムを，アーキテクチャ B 上で動作させるという方法は，ソフトウェア工学的にはかなり一般的な手法である．しかし，エミュレーションレイヤのオーバーヘッドを極力小さくし，その上で動作させるプログラムの性能劣化をなるべく引き起こさないというのが最も重要な論点となる．つまり，今回の WinKVM の場合，オリジナルの KVM とくらべて，今回開発するエミュレー

ションレイヤ上で動作する KVM（WinKVM）のパフォーマンスが致命的な性能劣化を引き起こさないことがもっとも重要な論点となる．今回は，Windows と Linux がデバイスドライバに対して提供するインタフェイスの違いを，エミュレーションレイヤだけでは解決することが出来なかったため，オリジナルの Linux/KVM ドライバ側に若干の変更を加える必要があるが，修正した LOC はわずか 10 行程度であり，簡単に移植できていることが示された．

また, 開発したエミュレーションレイヤが正しく動作していることを確認するため, nBench, fork-wait ベンチマーク，ApacheBench，ライブマイグレーションの VM 転送時間測定を行なって性能に関する定量的な評価を行なった．その結果，KVM と，それを Windows 上に移植した WinKVM のパフォーマンスにほとんど違いはなく，移植先に Windows の OS ノイズの影響に比べると，エミュレーションレイヤのオーバーヘッドはほぼ無視できるレベルであることが判明した，高速に Linux デバイスドライバである KVM を Windows 上で動作できることが示された．

4.1 KVM (Kernel-based Virtual Machine) 移植の論点

図 4.2 に示すように KVM はカーネル空間で動作する KVM ドライバとユーザ空間で動作する QEMU[35] から構成されている．KVM ドライバは特権命令である VT-x[41] や AMD-SVM[42] を操作しゲスト OS を動作させる．ユーザ空間にある QEMU は主に VM のデバイス (VGA や NIC) のエミュレーションを行う．ゲスト OS を実行するときには QEMU 側から KVM ドライバ側にシステムコールを発行し，そのシステムコールを受けて KVM ドライバが VT-x や AMD-SVM を操作しゲスト OS を実行する．

そのため，KVM ドライバ（Linux デバイスドライバ）を移植することが，KVM ドライバを Windows 上で動かす上で最大の論点となる．

デバイスドライバと呼ばれるソフトウェアは，オペレーティングシステムに強く依存するため，たの OS に移植するのは困難である．ユーザ空間で動作するアプリケーションプログラムの場合 C 標準ライブラリといったフレームワークが OS の差異を吸収してくれるため，他の OS で動いたプログラムが他の OS でもほとんど修正することなくで他の OS で動作させることは珍しいことではない．しかし，デバイスドライバの場合，オペレーティング・システムが提供する機能を直接利用するため，ソースコードをレベルで，他のオペレーティングシステムに移植するのは困難である．もちろん，プログラマが KVM ドライバのソースコードを一つ一つ解読してゆき，Windows 用のデバイスドライバに変換することで移植を達成することは可能である．しかし，この手法は人的なコストが掛かるため問題である．

そこで，われわれは，“簡単” かつ “性能劣化を引き起こすことなく”Linux/KVM ドライバを Windows に移植する手法を開発した．具体的には，Linux のエミュレーションレイヤを Windows カーネル空間上に構築し，その上で Linux デバイスドライバを動作させる手法を取る．エミュレーションレイヤで，異なるアーキテクチャ A 向けに書かれたプログラムを，アーキテクチャ B 上で動作させるという方法は，ソフトウェア工学的にはかなり一般的な手法である．Cygwin と呼ばれるプロジェクトは Linux 上のユーザ空間で動作するプログラムを Windows 上で動作させることができるエミュレーションソフトウェアである．これと同様に，Linux 上のカーネル空間で動作する Linux デバイスドライバを Windows 上のカーネル空間で

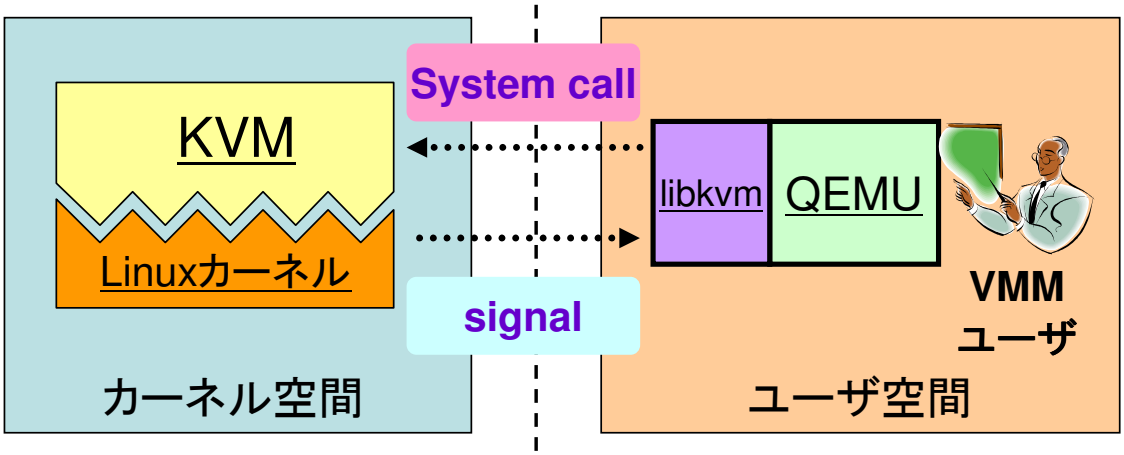


図 4.2. KVM のアーキテクチャ

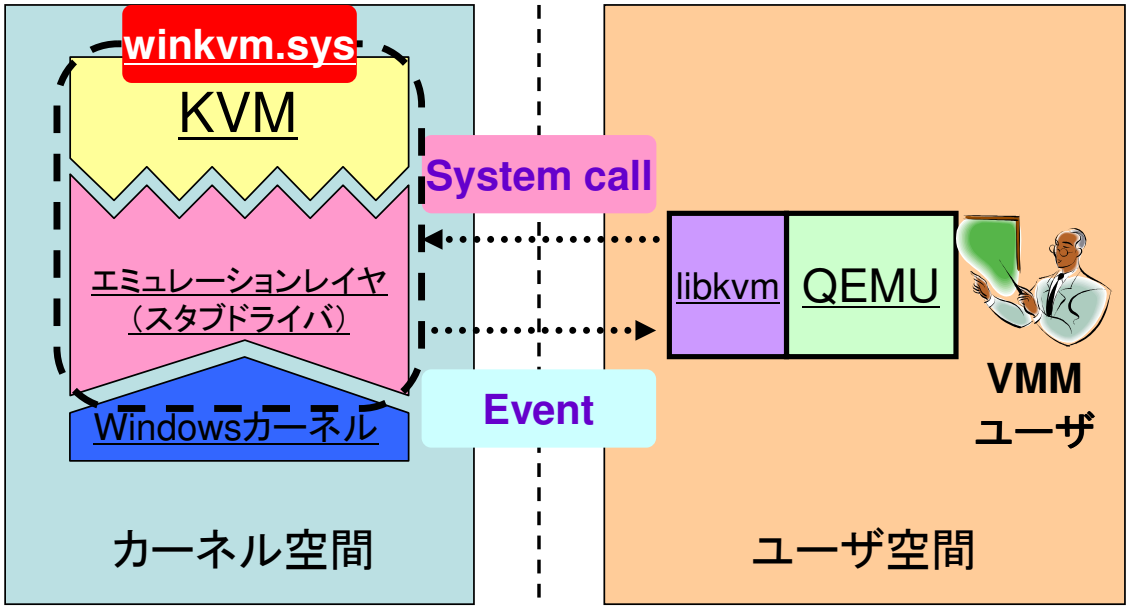


図 4.3. WinKVM のアーキテクチャ

動作させる手法を考えた．エミュレーションレイヤでソフトウェアを異なるアーキテクチャ上で再利用するという手法において，最も重要なポイントは “性能劣化を引き起こさないこと” である．そのため，移植にあたり，解決しなければならない問題がいくつか存在するが，それに関しては後述する．QEMU は Cygwin を使用して Windows 上でも動作させ，KVM ドライバは本章で論じる Linux カーネルエミュレーションレイヤで動作させる．最終的に，KVM 上で動作する WinKVM を得るのが本手法の概要である．

無論，エミュレーションレイヤを構築するのにはコストがかかる．しかし，一度 Linux エミュレーションレイヤを構築してしまえば，その上で様々な Linux デバイスドライバを動作させることができるし，また，KVM ドライバがバージョンアップしても追従してゆくのが容

易である．そのため，エミュレーションレイヤという手法は，一度構築してしまえば，後は，労力を掛けずに，素早くデバイスドライバを移植することができる．

4.1.1 乗り越えなければならない壁

Linux カーネルエミュレーションレイヤを構築するにあたっては，Windows カーネルと Linux カーネルの特性をそれぞれ分析した上で，何処に相違があり，どのようにしてそれを吸収すべきかを一つ一つ明らかにしてゆく必要がある．本研究では差異を，3 つのカテゴリーに分類する．これは，次の節で，これらの相違をどのように吸収すべきかについて論じる．

ユーザプログラムとのインタフェイス

デバイスドライバはユーザ空間で動作するプログラムと通信し，最終的にユーザに対して何らかのサービスを提供する．KVM ドライバに関しても，これは 4.1 節にて解説した通り，カーネル空間で動作する KVM ドライバと，ユーザ空間で動作する QEMU が協調動作することで，ゲスト OS の動作というサービスを提供している．

そこで，OS 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との間でコミュニケーションをとれるためのインタフェイスが用意されている．

UNIX 系のオペレーティング・システムで用意されている，`ioctl()`, `open()`, `close()`, `read()`, `write()` といったシステムコールはその代表例である．これらのシステムコールがユーザ空間側のプログラムで発行されると，予め対応付けられたデバイスドライバ内のハンドラが呼ばれることで，ユーザ空間とカーネル空間を結びつけることができる．これらのハンドラをユーザ側から呼び出されると，設定された，対応するカーネルドライバ側のハンドラが呼び出され，データをやり取りすることが可能である．

これらのシステムコールは，OS 独自のものであるため，ANSI/ISO C[43], SUS (Single UNIX Specification)[44], ISO/IEC 9945 (POSIX) [45] といった規格への準拠を掲げていない限り，オペレーティング間の互換性は存在しない．オペレーティングシステム A に存在していた $k()$ というシステムコールが，オペレーティングシステム B には存在しないということもありうる．そのため，オペレーティングシステム A のデバイスドライバを，オペレーティングシステム B 上で動作させたいと思った時，$k()$ というシステムコールの代替となるシステムコールが存在しない場合．$k()$ を効率よくエミュレーションする何らかの機構が必要となる．

また，それとは別に，メモリマッピング（MM）と呼ばれるインタフェイスも，モダンなオペレーティング・システムであればほぼ確実に搭載されている．基本的にデータのやり取りは，上述したハンドラを通してやり取りすれば良い．しかし，大容量のデータを高速にやり取りする NIC ドライバ等は，このようなハンドラを通してデータをやり取りすると，スループットの低下が生じる．なぜなら，システムコールを通したカーネルとアプリケーション間の通信は，システムコールを発行するコストが高いため，高スループットの通信が必要な場合，大量のシステムコールの発行による性能劣化が発生する．そこで，近代的なオペレーティングシステムであれば，ユーザ側のメモリとカーネル側のメモリを関連付ける，メモリマッピングと呼ばれる機能が提供されている．一度メモリマッピングが完成すれば，システムコールを発行するよりもはるかに低いコストで，ドライバ側とユーザ空間側でデータをやり取りすること

が可能である．

このメモリマッピング（MM）というインタフェイスは，OS のメモリ管理ポリシーによって，どのようなインタフェイスが提供されるかが大きく左右される．そのため，オペレーティングシステム A では許されているメモリマッピング手法が，オペレーティングシステム B では許可されないということが起こりうる．そのため，エミュレーションレイヤではこの違いを吸収する必要がある．

発行可能な Privilege な命令の種類

特権命令とはリング 0，すなわち最高の特権レベルで動作するプログラムのみ実行可能な CPU の命令のことである．一般的に，デバイスドライバと呼ばれるソフトウェアは，OS カーネルと同等の特権レベル，すなわちリング 0 で動作する．これは，Windows カーネルも Linux カーネルも同様である．しかし，OS は CPU のコンフィグレーションによって，リング 0 で動作するデバイスドライバであっても，特定の命令の実行を許可する・しないという，より細かな制御も可能である．すなわち，オペレーティングシステム A のデバイスドライバでは実行可能な命令が，別のオペレーティングシステム B のデバイスドライバでは実行不可能な命令というものが存在する可能性がある．

具体例を出すと，Windows 上で Linux カーネルエミュレーションを実現するにあたっては，これは深刻な問題である．具体的な例を挙げると，Linux カーネル上では実行可能な VT-x の命令である `VMLAUNCH` 命令や `VMRESUME` 命令が Windows カーネル上では実行できない．実行しようとすると，一般保護例外が発生し，命令の実行が許可されない．これは KVM ドライバをエミュレーションレイヤ上で動作させる時に大きな障害となる．これらの命令が実行できないのであれば，VT-x 命令を活用する KVM を動作させることが不可能になるからである．もちろん，VT-x 命令の挙動をすべてソフトウェア的にエミュレーションレイヤで模倣すれば，これらの命令を実行させる環境を再現することは可能である．しかし，この手法では，著しいパフォーマンスの低下が予想され，この手法を採用することはできない．そのため，これらの命令を，Windows デバイスドライバ上から直接実行できるようにする手法を考案する必要がある．

カーネルとのインタフェイス

カーネルとのインタフェイスはドライバがカーネルの機能にアクセスするためのカーネル内 API のことである．この API を呼び出すことで，ドライバはカーネル内で利用可能なメモリを確保したり，空きページを確保したりすることが可能である．当然，これらの API は OS 独自のものとなっており，アプリケーションレベルのプログラムでは利用される，ANSI/ISO C[43], SUS (Single UNIX Specification)[44], ISO/IEC 9945 (POSIX) [45] といった標準規格は存在しない．そのため，こういった OS 依存の API を使わざるを得ない点も，ドライバプログラムの移植を困難にしている要因の一つである．

表 4.1. Linux と Windows のシステムコールの対応関係

Linux	Windows	Explanation
Open/Close		
open() close()	CreateFile() CloseHandle()	VMインスタンスの作成と開放
Memory Mapping		
mmap() munmap()	該当API無し (DeviceIoControl()で代用する. MapViewOfFile(), UmmapViewOfFile()は使用不可)	VMのメモリ空間とプロセスメモリ空間のマッピング
Ioctls		
ioctl()	DeviceIoControl()	その他，VMへの各種制御コマンド発行

それ以外の細かな問題

Linux デバイスドライバのソースコードは Linux のコンパイラであ る GNU gcc に強く依存している．一方で，Windows のデバイスドライバを開発するためには Microsoft 製のコンパイラ (VisualC++ 等) を使用しなければならない．Microsoft 製のコンパイラで KVM のソースコードをコンパイルすることはできず，また，gcc を使って Windows のドライバをコンパイルすることもできない．そのため，この開発環境のを埋めるスキームを用意する必要がある．

4.1.2 解決手法

本節では 4.1.1 節で論じた課題に対して，どのように解決すれば Windows 上で Linux のデバイスドライバを動作させることができるかについて論じる．

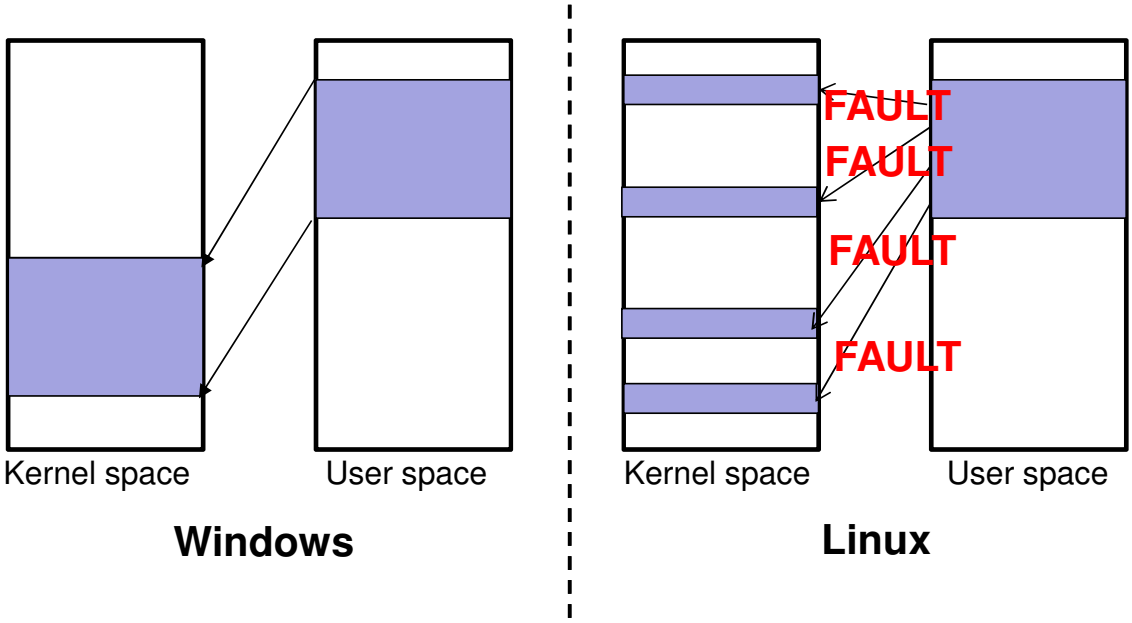


図 4.4. Winodws ではページフォルトフックによるメモリマッピングが不可能

ユーザプログラムとのインタフェイスの解決法

オペレーティング・システムにはユーザ空間で動作するプログラムとカーネル空間で動作するプログラムの間でデータや指令をやり取りするための仕組みがある．Linux の場合システムコールがそれに相当する．

Linux には UNIX 系のオペレーティング・システムで用意されている，`ioctl()`,`open()`, `close()`, `read()`, `write()` といったシステムコールが存在している．

一方で，Windows にもこれに一対一で対応付けられるシステムコールが存在している．**表** 4.1 に対応関係の表を示す．Linux の `ioctl()` には `DeviceIoControl()`API がそれに対応する．Linux の `open()`‘には `CreateFile()`API が対応する．さらに `close()` には `CloseHandle()`API が対応する．`read()` と `write()` 関数には，`ReadFile()`, `WriteFile()`API がそれに相当する．そのため，Linux デバイスドライバを操作するユーザ側のプログラムは，Linux 側のソースコードのシステムコールを受け取り，Windows 側の API に変換する，ラッパー関数を作るのは容易である

ユーザプログラムとカーネルプログラムのインタフェイスはシステムコールだけではない．上述したようにメモリマッピング（MM）と呼ばれるインタフェイスも存在しており，これは，OS のメモリ管理アルゴリズムのメモリ管理ポリシーに強く依存するものとなっている．そのため，システムコールのように，エミュレーションレイヤによる単純にラッパー関数の提供だけでは差異を吸収するのは不可能である．

具体的に Windows と Linux の MM のインタフェイスの違いについて**図** 4.4 を用いて解説する．

Windows と Linux の大きな違いは，ページフォルトハンドラをドライバのプログラマがト

ラップできるか否かという点にある．Linux の場合，ユーザプログラムが確保したユーザ空間のメモリ領域に対して，物理ページを割り当てる際に，ページフォルトハンドラを利用することができる．つまり，Linux の場合，ページフォルトハンドラを利用して，ユーザ空間とカーネル空間の間でメモリマッピングを行うことが許可されている．

一方で，Windows はページフォルトハンドラを利用した MM は不可能である．Linux の場合はページフォルトハンドラを利用して，プログラマが望む実メモリ領域に，ユーザ空間をヒモ付することが可能であったが，Windows の場合それが不可能である．Windows はカーネルが予め決め打ちで用意した連続の固定領域を，ユーザ空間にマッピングすることしかできない．

そのため，Linux のデバイスドライバがページフォルトハンドラを利用して，ユーザ空間とカーネル空間のメモリマッピング（MM）を行なっていた場合，これを単純にエミュレーションレイヤが提供するラッパー関数等でエミュレーションすることは不可能である．むろん，パフォーマンスを無視し，非効率な方法でもってエミュレーションするのであれば，エミュレーションレイヤという工学的手法を取っている以上，十分に可能である．しかし，本研究の最も重要な論点は “性能劣化を引き起こさないこと” である．そのため，性能劣化を引き起こさず，Windows 上で Linux カーネルのページフォルトによる MM のインタフェイスをエミュレーションする必要があるが，

結論から言うと，エミュレーションレイヤだけで，Linux のページフォルトによる MM を Windows 上で効率よく模倣するのは不可能である．これは本手法の限界である．そこで，本研究では，Linux デバイスドライバがページフォルトハンドラを利用していた場合，その部分のコードを，ページフォルトハンドラを利用しないメモリマッピングの手法に書きなおす必要がある．Linux のページフォルトインタフェイスを使わずに MM が利用できるように，移植元の Linux ドライバを書き直す必要がある．Linux/KVM ドライバの場合，3 行程度の書き換えでこれを実現することが出来た．

発行可能な Privilege な命令の解決策

われわれはこの問題を，RPL を 3 から 0 に低下させた DS 値と ES 値をゲスト OS からホスト OS への復帰ポイントに設定することで解決した．KVM のホスト復帰ポイントを設定している KVM のコードを部分的に改造し，ホスト OS の ES と DS の RPL を 3 から 0 に書き換えた値を，ホスト復帰ポイントとして設定するようにした．つまり，ゲスト OS 実行前の DS と CS の RPL 値は 3 であるが，ゲスト OS が実行された後では，その値が 0 になることになる．

この，手法について詳しく解説する．Linux/KVM ドライバがどのようにゲスト OS を実行するかを図**図** 4.5 に示す．初めに，Linux 上で動作する KVM ドライバは，Privilege 命令である `VMLAUNCH` 命令を発行して，ゲスト OS を作動させます．ゲスト OS の実行中に，VMM がエミュレートする必要がある Sensitive 命令がゲスト OS 内部で発行された場合，`VM Exit` と呼ばれる発生して KVM ドライバ側に実行が移ります．KVM ドライバは，`VM Exit` の探り，原因に応じたエミュレーションを行なった後，再びゲスト OS に動作を渡します．このサイクルを繰り返すことで，KVM はゲスト OS を実行する．

ところが，Windows の場合，上述したようにドライバの動作レベルがため，この VMLAUNCH 命令を発行することが不可能 (図 4.6) である．Windows ドライバの場合，VMLAUNCH 命令や VMRESUME 命令を発行した場合，一般保護例外が発生し，ゲスト OS を動作させることが出来ない．これは大きな問題である．

そこで，トランポリンコードと呼ばれる復帰ポイントを利用することでこの問題を解決した，図 4.7 に概念図を示す．エミュレーションレイヤは，Windows カーネルのスワップアウトされないメモリ空間にトランポリンコードを書き込み，個々を復帰ポイントとする．ゲスト OS から VM exit が発生した場合，直接 Windows ドライバ側に実行を移すのではなく，まず

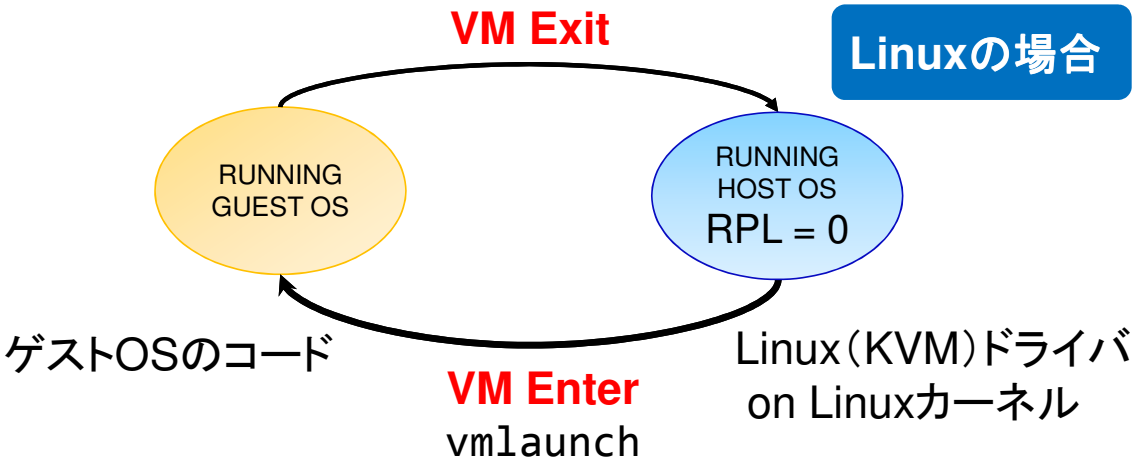


図 4.5. Linux カーネル内での Linux/KVM ドライバの Privilege 命令（VT-x 命令）の発行

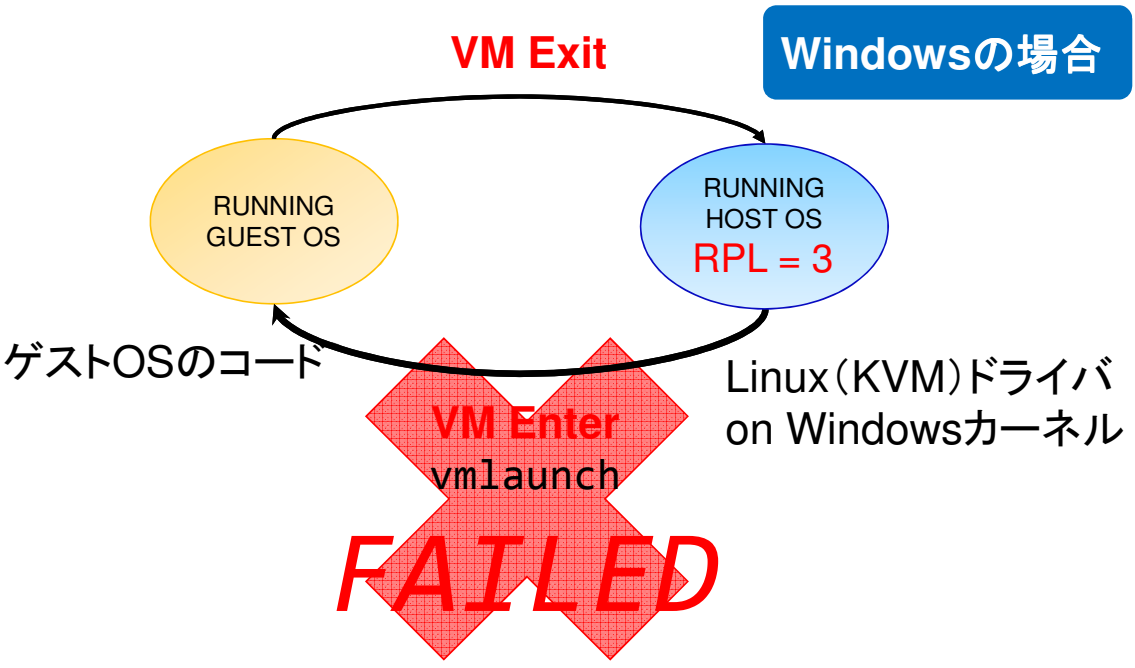


図 4.6. Windows カーネル上で修正無し KVM ドライバを動作させた時の問題点

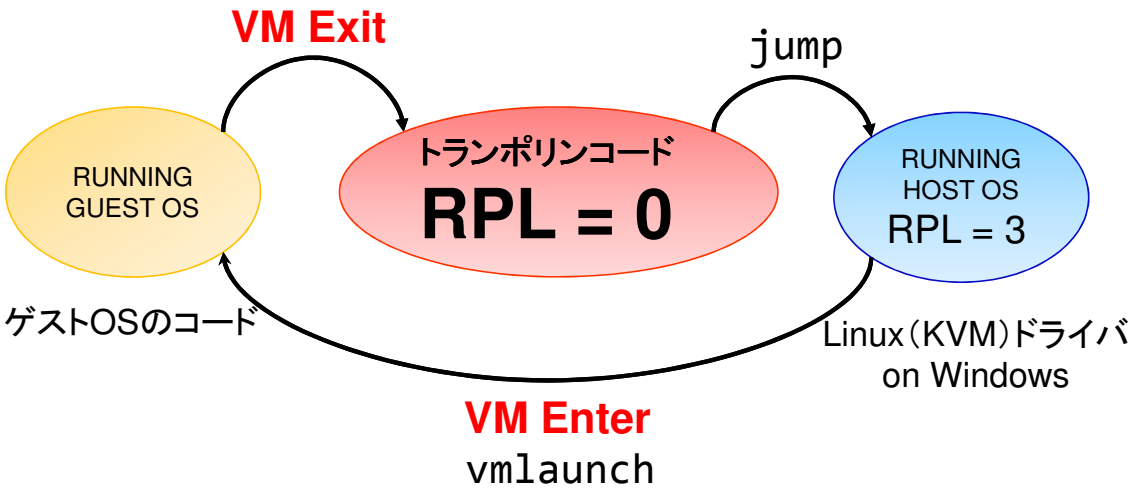


図 4.7. Windows ドライバ上では発行禁止の Privilege 命令をトランポリンコードを使って実行

は，このトランポリンコードに実行を移した後，改めて，Windows ドライバ側に実行を移すような手法を取った．KVM ソースコード自体を書き換える必要があるが，行数は 10 行程度である．

この手法では，本来ならば RPL が 3 で動作するはずの Windows ドライバのコードを一時的に RPL が 0 で実行する瞬間が発生することになる．しかし，この手法では致命的な問題点は発生しない．引き下げられた RPL の値は，Windows の割り込み時には 0 から 3 に戻るため，永続的に 0 にはならない．RPL は Requester's Privilege Level の略で不適切なシステムコール呼び出しを回避するためのものである．例えば，特権レベル 0 のセグメンテーション A と，特権レベルが 3 のセグメンテーション B, C，合計で 3 つ (A, B,C) のセグメンテーションを使用するオペレーティングシステムが存在するとして．特権レベルが 3 (B, C) のプログラムが特権レベル 0 上 (A) のセグメンテーションにあるメモリデータを直接盗み出すことはできない．しかし，特権レベル 0 (A) にシステムコール相当 (例えばコールゲート) が存在すると，そのコールゲートを経由することで，特権レベルが 3 (C) で動作しているプログラムが，特権レベルが 3 である (B) に特権レベル 0 (A) のメモリデータを書き込み，そのうえで，特権レベルが 3 (C) のプログラムが特権レベル (B) のデータを読むことができる．つまり，特権レベルが 3 であるプログラムがシステムコール相当 (コールゲート) を経由して特権レベル 0 のメモリデータを読めてしまうのである．RPL はこれを防止するためのものであり，システムコールの発行元の特権レベルを OS がチェックできるようにするためのものである．

しかし，近代のオペレーティングシステムは，セグメンテーションによる保護機構は使用しないのが一般的であり，主に，ページ単位による保護機構を利用している．RPL はセグメンテーションを活用するオペレーティングシステム上で利用されるためのものであるため，ページ単位による保護機構を使用している Windows では特に大きな問題は発生しないと考える．

なお，Windows 用の VMM である VirtualBox もわれわれの手法と同様のことを行って PRL の問題点を解決している．ゲスト OS からホスト OS への復帰ポイントには RPL が 0 になったホスト OS のセグメント値が代入されるため，ゲスト OS からホスト OS に VMexit

が発生したときには，一時的に VirtualBox 内の ES,DS が 0 である復帰ポイントに戻ることになる．この復帰ポイントが実行される間は割り込みを禁止しておき，VirtualBox 以外のプログラムが一切実行できない状態にする．そのあとで，退避しておいた Windows の元の値である ES 値と DS の値を復帰しホスト OS に戻るという形をとっている．

仮に, RPL が 0 に修正されたことによるセキュリティリスクが発生するとしても, WinKVM は VirtualBox と同様の手法でこれを回避することが可能である．そのため，今回のようにプロトタイプの実装においては致命的な問題は発生しない．

カーネルとのインタフェイスの解決策

上述したように，カーネル内で利用できる関数は OS の実装依存であり，基本的に OS 間の互換性はない．そのため，Windows カーネル内で利用できるネル内関数を，ただちに，Linux カーネル内で利用することは出来ない．

しかし，この問題を解決するのは比較的容易である．Linux カーネル API の代替となるような，何らかの API が Windows 内にも存在するからである．そのため，特定の Linux カーネル内関数 A が実行されたとき，その関数 A の代替となるような関数 A' を呼び出すするスタブ関数を用意すればよい．

表 4.2 に変換表を示す．この表では，代表的な API だけを示した．ほとんどの API は一対一で対応付けすることができる．たとえば，カーネルメモリアロケータである `kmalloc()` 関数は，Windows のカーネル API である `ExAllocatePoolWithTag()` 関数に置き換えることができる．

それ以外の細かな問題の解決策

4.1.1 **節**で述べたように，Linux のデバイスドライバのソースコードは GNU gcc に強く依存している．一方で，Windows のデバイスドライバは同じ C 言語で書かれているとはいえ，Windows の WDK に含まれる専用の C コンパイラに強く依存している．そのため，Linux のデバイスドライバのソースコードを Windows の WDK コンパイラでコンパイルすることは，コードを全面的に書きなおさない限り不可能である．当然，コードを書きなおすのは人的コストが高いため，コードを人の手で書きなおすという選択肢を撮ることはできない．

そこで，われわれは，Visual C++ と Cygwin gcc を併用することでこの問題を解決した．始めに Linux ドライバのソースコードを Cygwin gcc でコンパイルする．すると，COFF バイナリで Linux ドライバのバイナリフォーマットが出力される．われわれのエミュレーションレイヤは Visual C++ で書かれているが，COFF バイナリ形式でコンパイルされた Linux ドライバは Visual C++ でリンクすることができる．このように，gcc 依存である Linux ドライバのソースコードに修正を加えることなく，Visual C++ を使って Linux ドライバを Windows 用のドライバとしてビルドすることが可能である．

図 4.8 に Linux ドライバ（図中では KVM ドライバ）のソースコードから Windows ドライバへのビルドプロセスを示す．ビルドプロセスは 3 ステップに分けて行われる．

第 1 ステップ，Linux ドライバのコンパイルにはカーネルのヘッダーファイルが必要である．そのため，Linux ドライバで使用されているカーネル関数のヘッダーファイルを Linux

表 4.2. 主要 Linux カーネル API と Windows カーネル API の変換対応表

Linux カーネル API	Windows カーネル API
kmalloc() kzalloc() vmalloc()	ExAllocatePoolWithTag()
kfree() vfree()	ExFreePoolWithTag()
mutex_init() mutex_lock() mutex_unlock() mutex_trylock()	ExInitializeFastMutex() ExAcquireFastMutex() ExReleaseFastMutex() ExTryToAcquireFastMutex()
alloc_pages() free_pages()	ExAllocatePoolWithTag() ExFreePoolWithTag()
__va() __pa()	MmGetVirtualForPhysical() MmGetPhysicalAddress()
get_cpu() get_nr_cpus()	KeGetCurrentProcessorNumber() KeQueryActiveProcessorCountCompatible()
smp_call_function() smp_call_function_single() on_each_cpu()	local_irq_enable() local_irq_disable()
smp_wmb() smp_mb()	KeMemoryBarrier()

カーネルのソースコードから抽出した．また，カーネルドライバのコンパイル時には genksym とよばれるプログラムが実行され，これはドライバのソースファイルがコンパイルされた後に特殊なシンボルを付与する後処理である．Windows ドライバとしてビルドするにあたってこの後処理は不要であるため，カーネルの Makefile から genksym を走らせる処理を削除した．**図** 4.9 と**図** 4.10 に示したパッチを Linux の Makefile に当てることで Makefile を改造することができる．

第 2 ステップとして，Linux カーネルから抽出したヘッダーファイルと改造した Makefile とをあわせて，Cygwin gcc を用いてコンパイルする．コンパイルすると，COFF 形式でコンパイルされた Linux デバイスドライバのバイナリが手に入る．COFF バイナリは VisualC++ で解釈可能なバイナリであるため．次のステップにつなげることができる．

第 3 ステップは，われわれの書いたエミュレーションレイヤのプログラムと先ほど COFF 形式で吐き出された Linux ドライバのバイナリを VisualC++ で併せてコンパイルとリンクを行う．

以上の処理を行うことで，Linux のデバイスドライバのコンパイル済みバイナリとエミュレーションレイヤをリンクし，Windows 上で動作させる事ができる Linux ドライバのバイナ

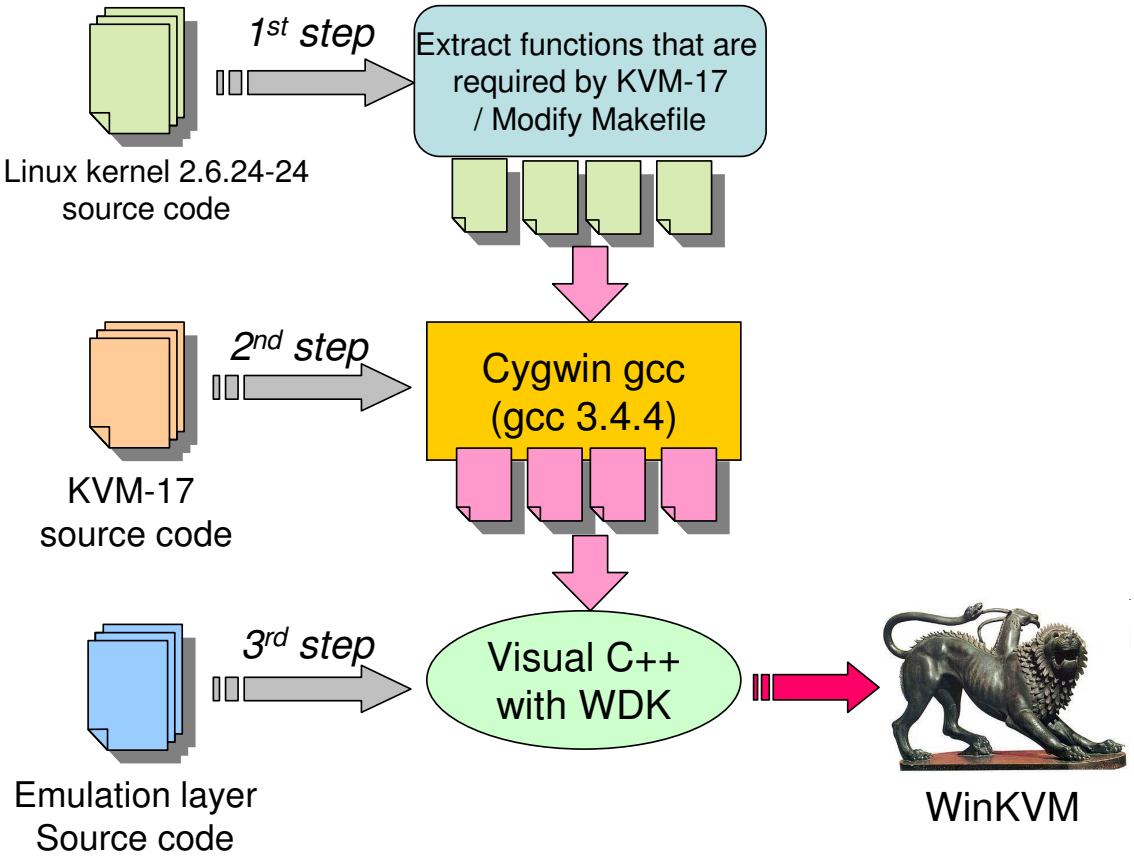


図 4.8. KVM コードの gcc 依存に対する解決手法

リを得ることができる．

4.2　エミュレーションレイヤを用いた KVM 移植による WinKVM の実装

実際に上述した手法にて Linux デバイスドライバである KVM ドライバを Windows 上で動作させた．その際，幾つか細かい点で KVM ドライバを改造しなければならない点があったので，それについて述べる．

図 4.3 に WinKVM のアーキテクチャを示す．KVM を移植するために，われわれは，Windows 上に Linux カーネルをエミュレーションするレイヤを構築しその上で KVM を動作させた．開発したエミュレーションレイヤと KVM ドライバをあわせてリンクして最終的に Windows のドライバとして動作する WinKVM を得た．

WinKVM を開発するに当たって KVM-17 を使用した．KVM-17 は VM のライブマイグレーションを実装した初めてのバージョンであり，最新の KVM バージョンに比べて比較的構造が単純である．そのため，プロトタイプの実装としては最適である．

4.1.2 節で述べたように，Linux のカーネルドライバを Windows 上で動作させるエミュレーションレイヤを動作させるための本手法には，限界が存在する．それは，カーネル空間とユー

```
*** fixed/linux/scripts/Makefile.build
--- original/linux/scripts/Makefile.build
***************
*** 178,184 ****
  cmd_cc_o_c = $(CC) $(c_flags) -c -o $(@D)/.tmp_$(@F) $<
  cmd_modversions =                           \
!   if false && $(OBJDUMP) -h $(@D)/.tmp_$(@F) | grep -q __ksymtab; then    \
        $(CPP) -D__GENKSYMS__ $(c_flags) $<         \
        | $(GENKSYMS) $(if $(KBUILD_SYMTYPES),          \
                  -T $(@D)/$(@F:.o=.symtypes)) -a $(ARCH)   \
--- 178,184 ----
  cmd_cc_o_c = $(CC) $(c_flags) -c -o $(@D)/.tmp_$(@F) $<
  cmd_modversions =                           \
!   if $(OBJDUMP) -h $(@D)/.tmp_$(@F) | grep -q __ksymtab; then \
        $(CPP) -D__GENKSYMS__ $(c_flags) $<         \
        | $(GENKSYMS) $(if $(KBUILD_SYMTYPES),          \
                  -T $(@D)/$(@F:.o=.symtypes)) -a $(ARCH)   \
```

図 4.9. Linux カーネルの Makefile に当てる patch1

```
*** fixed/linux/arch/x86/Makefile_32
--- original/linux/arch/x86/Makefile_32
***************
*** 37,43 ****
  endif
  CHECKFLAGS    += -D__i386__
! KBUILD_CFLAGS += -pipe -msoft-float -mregparm=3 -freg-struct-return
  # prevent gcc from keeping the stack 16 byte aligned
  KBUILD_CFLAGS += $(call cc-option,-mpreferred-stack-boundary=2)
--- 37,43 ----
  endif
  CHECKFLAGS    += -D__i386__
! KBUILD_CFLAGS += -pipe -msoft-float -freg-struct-return
  # prevent gcc from keeping the stack 16 byte aligned
  KBUILD_CFLAGS += $(call cc-option,-mpreferred-stack-boundary=2)
```

図 4.10. Linux カーネルの Makefile に当てる patch2

ザ空間との間でメモリマッピングを行うためのインタフェイスが Windows と Linux では著しく異なるため，レミュレーションレイヤを用いてこれを効率よくエミュレーションすることは不可能である．

4.1.2 **節**で述べたようにで述べたように，これは本手法の限界である．そのため，エミュレーションレイヤを用いて Linux デバイスドライバである KVM ドライバを Windows 上で動作させるためには，KVM ドライバのソースコードに若干の修正が必要である．本手法は Windows 上に Linux カーネルを模倣するエミュレーションレイヤを構築することで KVM ドライバを修正することなく Windows 上で動作させることを目標としている．しかし，エミュレーションレイヤではどうしても KVM を動作させることができない部分もあるため，KVM ドライバに数行の修正を加える必要がある．これは，4.1.2 **節**ですでに述べたように本手法の限界である．

また，ユーザ側で動作する QEMU と libkvm が存在しており，こちらも Windows 上で動作させる必要がある．QEMU は Windows 上で動作させるために改良を施した QEMU が存在するため移植する必要はない．しかし，libkvm に関しては，KVM ドライバと通信する部分であるため，4.1.2 **節**で述べたように，システムコールを呼び出す部分を変更する必要がある．`open()` は `CreateFile()`API に，`close()` を `CloseHandle()`API，`ioctl()` を呼び出している部分を，`DeviceIoControl()` 関数に置き換えた．もちろん，4.1.2 **節**で述べたように，Linux が用意するシステムコールをエミュレーションするラッパー関数を構築することで，QEMU と libkvm のソースコードを変更することなく動作させることも可能である．しかし，今回は，実装の手間を考え，ソースコードを直接書き換えることで対処した．Linux が用意するシステムコールをエミュレーションするラッパー関数を実装するのは容易であるため，本質的に困難な点ではない．そのため，実装のコストも考え，今回は QEMU と libkvm が直接 Linux のシステムコールを呼び出している部分，Windows のシステムコールに置き換えることで対処して問題はない．

KVM をエミュレーションレイヤを用いて動作させる時，最も困難となる点は 4.1.2 **節**の**ユーザプログラムとのインタフェイスの解決法**で述べた，エミュレーションレイヤを使用した Linux のメモリマッピングを，Windows でどのように効率よくエミュレーションするかという点にある．すでに，4.1.2 **節**でも，簡単に解説したが，今一度 KVM ドライバという具体例を用いて，どの点がエミュレーションレイヤを構築するにあたって困難であるかを解説する．

4.1 **節**でも述べたように，KVM はカーネル側で動作する KVM ドライバと，ユーザ側で動作する QEMU 側の 2 つに分けることができる．この 2 つのコンポーネントはゲストメモリ空間を共有している．しかし，Windows と Linux がドライバに対して提供するカーネルとユーザ間で同一メモリ空間を共有するメモリマッピング機構の性質の違いがエミュレーションレイヤを構築するにあたっての障害となる．

4.1.2 **節**でも述べたように，これはエミュレーションレイヤだけでこの差異を吸収することは不可能である．そのため，われわれは，やむなく KVM ドライバのソースコードを改造することで，この問題の解決を図った．本節では，われわれはこのメモリ共有メカニズムの差異をどのように解決したかについて論じる．

Linux のカーネルとユーザ間のメモリ共有メカニズムの概念図を**図** 4.11 の左に示す．図に

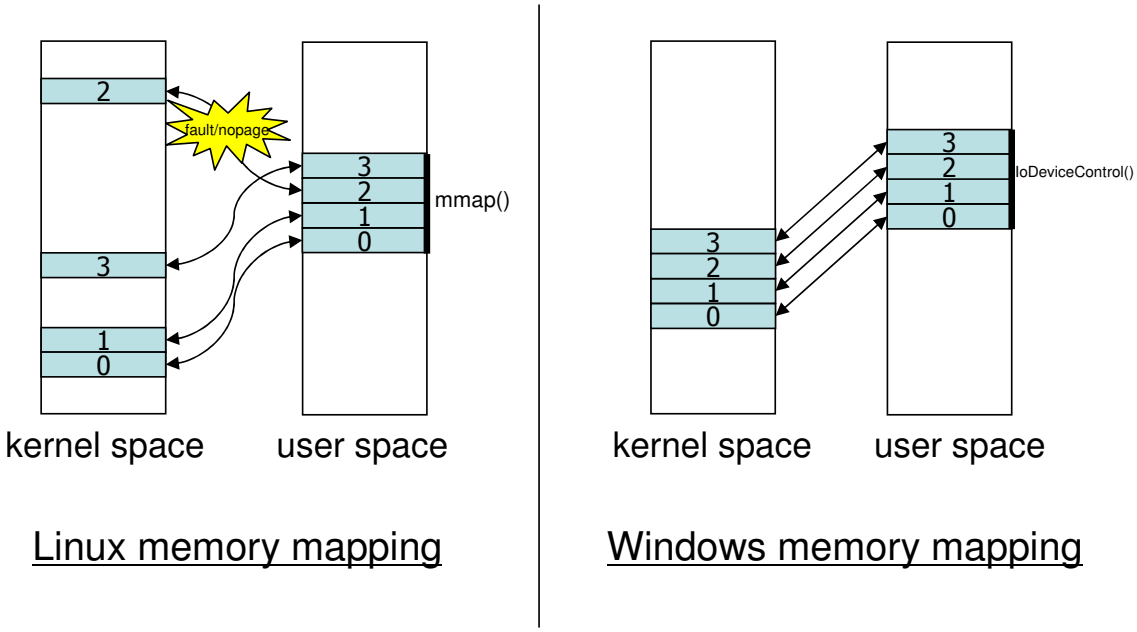


図 4.11. Linux と Windows のメモリマッピングの違い

あるように，Linux ではカーネル側で確保した非連続のページ領域を，ユーザ側の連続メモリ領域 (`mmap()` で確保した領域) にマッピングすることが可能である．

Linux の場合，デバイスドライバのプログラマーはドライバ内の `fault / nopage` ハンドラを使って，カーネル内ページがどのユーザメモリ領域に関連付けられるかを指定することができる．例えば，図 4.11 の左図にもあるように，`mmap()` で確保した 4 ページ分 (0 番〜3 番ページ) のメモリがあるとして，このとき，ユーザ側のメモリ領域から 2 番ページにアクセスがあったとき，Linux カーネルは `fault / nopage` ハンドラ呼ぶ．このときプログラマーは，2 番ページがどのカーネル内ページに割り当てられるかを指定することができる．結果的に，連続したユーザメモリ領域を非連続のカーネルメモリ領域に割り当てることができる．KVM-17 はこの `fault / nopage` ハンドラを KVM ドライバと QEMU 間のゲストメモリ共有に使用している．そのため，エミュレーションレイヤを構築するためには，この `fault / nopage` ハンドラをエミュレーションする必要がある．

一方，Windows のカーネルユーザ間のメモリ共有メカニズムの概念図を図 4.11 の右図に示す．Windows の場合，Linux では可能だった，非連続のカーネルメモリと，連続領域のユーザメモリのマッピングが不可能である．Windows では連続したカーネルメモリ領域を，同じく連続したユーザメモリ領域に割り当てることしかできない．どのユーザメモリ領域に，どのカーネルメモリ領域が割り当てられるのかは，Windows カーネルが暗黙的に決めてしまうのである．つまり，Linux ではあった fault ハンドラ / nopage ハンドラに相当するものが Windows には存在せず，エミュレーションレイヤ上でこのハンドラを高速にエミュレーションすることが難しい．さらに，一度にマッピングできるメモリ共有領域にも制限があり，例えば，2GBytes の物理メモリを搭載するマシンでは一度に 300MB のメモリ共有領域しか作ることができない．すなわち，WinKVM では VM に割り当てられるメモリ領域が制限されてし

まう．

このように，Windows と Linux 間ではユーザ空間とカーネル空間とのメモリ共有に大きな違いがあるため，2 つの問題が発生する．1 つ目は，KVM が使っている `fault / nopage` ハンドラを高速にエミュレーションすることができないという問題点である．このため，ゲスト領域とホスト領域のメモリマッピングが行われず，ゲスト OS を正しくエミュレーションすることが不可能になる．2 つ目は，Windows 上でのメモリマッピングが一度に 300MB 分しか作れないため，WinKVM の VM に割り当てられるメモリ領域が制限されてしまうという問題点である．

結局，`fault / nopage` ハンドラはわれわれの基本方針であるエミュレーションレイヤを使用して高速にエミュレーションすることは不可能であるという結論に至った．そのため，KVM 側のソースコードを修正し，KVM 側のコードが `fault / nopage` ハンドラを使用せずともカーネル側とユーザ側のメモリマッピングが可能になるように KVM のソースコードを修正ことでこの問題を解決した．むろん，ゲスト OS をソフトウェア的にエミュレーションする VMM である以上，低速な手法でこの fault /nopage ハンドラをエミュレーションすることは可能である．例えば，ゲスト OS のすべてのメモリアクセスをトレースし，エミュレータ上ではページが割り当てられていないとみなされたメモリ領域にアクセスが起こったとき，`fault / nopage` ハンドラを呼び出すといった手法が考えられる．しかし，この手法は著しいパフォーマンスの低下を引き起こすことが予想され，現実的な手法ではない．われわれの今回の手法は，エミュレーションレイヤにより，Linux カーネルのカーネル関数と Windows のカーネル関数を一対一で変換することにより，Windows カーネル上で高速な Linux カーネルのエミュレーションを行うことである．そのため，nopage / fault ハンドラを使わずともエミュレーションレイヤ上で KVM が動作できるように KVM を改造する手法をとった．

2 つ目の問題点を解決するためには，QEMU 側のソースコードと KVM のソースコードを根本的に書き直す必要がある．そのため，今回われわれがおこなったエミュレーションレイヤを用いた WinKVM 実装ではこの問題を解決できなかった．これは，Linux デバイスドライバをエミュレーションレイヤを使用して Windows 上で動作させる手法の限界である．しかし，エミュレーションレイヤを開発し KVM ドライバを Windows 上で動作させることで，Windows と Linux 間で動作する Hypervisor を構築する際に留意すべき点についての知見を得ることができた．これに関する詳細については 4.3 **節**で述べる．

はじめに，1 つ目の問題を解決するために KVM 側にあてたソースコードの修正 (パッチ) について詳解する．**図** 4.12 にエミュレーションレイヤと KVM に適用するパッチがどのように協調動作するかを図示する．

パッチは，KVM が内部に持つマッピング管理テーブルをエミュレーションレイヤ側の持つマッピング管理テーブルに置き換える作用を持つ．KVM は内部に独自のマッピング管理用のデータ構造を持っている．図中では，QEMU の 0 番仮想ページを a 番の物理ページに対応付けるというテーブルが KVM 内に存在している．`fault / nopage` ハンドラが呼ばれたときには KVM は内部的にこのテーブルを参照してどの QEMU の仮想ページにどの物理ページを対応付けるかを決定している．一方で，エミュレーションレイヤのほうでもマッピングの管理テーブルを持っている．図中では，QEMU の 0 番仮想アドレスに a 番の物理ページが対応付

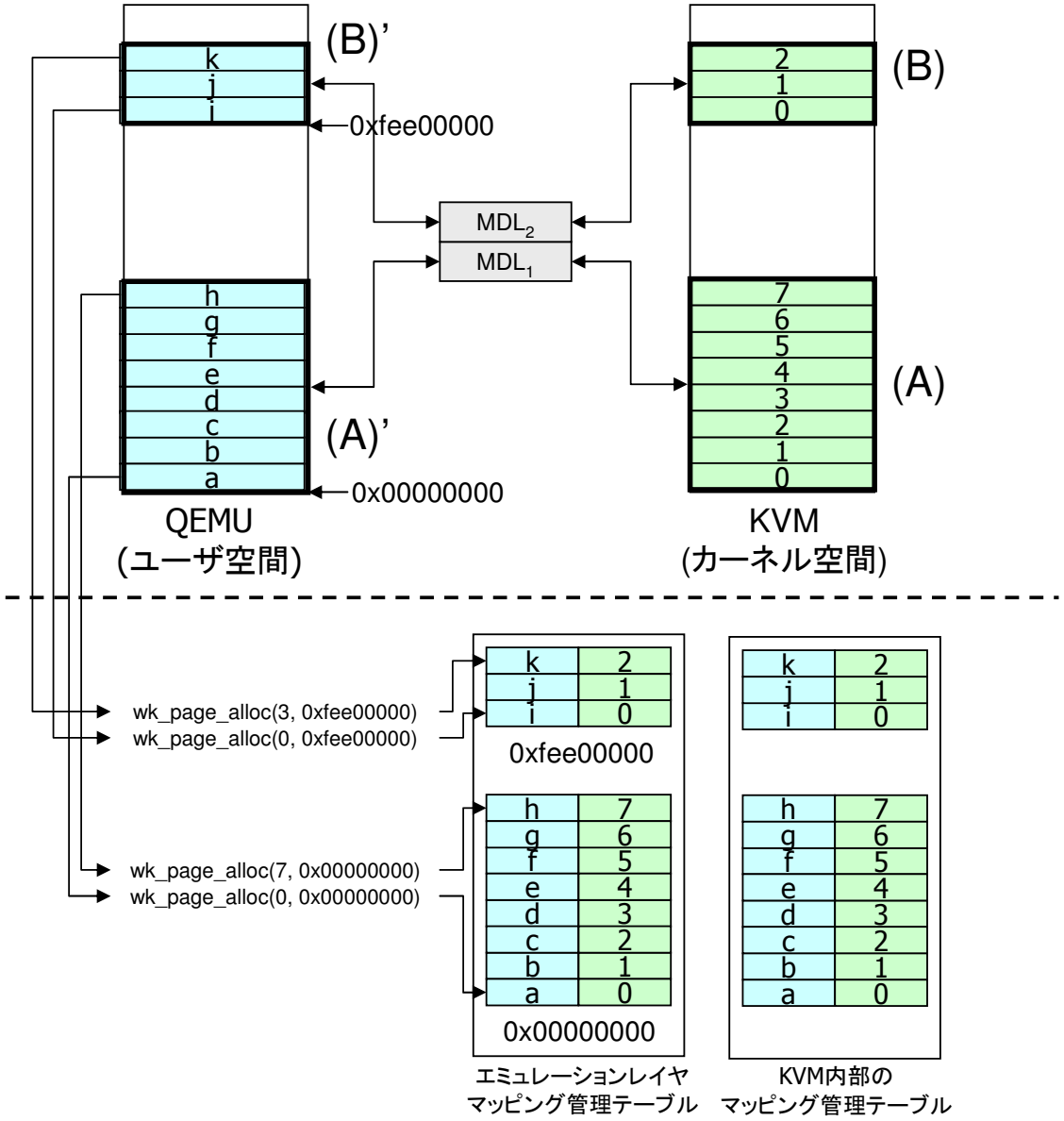


図 4.12. エミュレーションレイヤの機構

けられている

エミュレーションレイヤとパッチの実際の動作を順を追って解説する．これから記す一連の流れは，ユーザによってゲスト OS が起動されたときの初期化時に一度だけ行われるものである．

始めに，エミュレーションレイヤはカーネル空間とユーザ空間との間にメモリマッピング領域 (図中の (A)(B) (A)'(B)') を作成する．領域のサイズはゲスト OS に割り当てられる全メモリの量で決まる．例えば，ユーザがゲスト OS に 300MB のメモリ領域を割り当てた場合，300MB のメモリマッピング領域が作られる．MDL[*1] と呼ばれる Windows の機構を使うこと

[*1] セクションを使ってマッピングする方法もあるが，今回は使用できない．なぜなら，カーネル内で物理ページ

でメモリマッピングを行う．マッピングが行われると，エミュレーションレイヤ内のマッピングの管理テーブルに，QEMU の仮想アドレスのどのオフセットアドレスから何ページ分割り当てられているか，それらがどのカーネル内物理アドレスに割り当てられているかが記録される．図中では，2 つのマッピング (A) (B) が作られており，(A) はゲスト OS のメモリの 0x0 番地から 8 ページ分のメモリが割り当てられている．(B) には 0xfee00000 番地から 3 ページ分のメモリが割り当てられている．(A) と (B) に存在するページはカーネル内の物理アドレスに対応付けられている．そして，(A) 領域の 0 番仮想ページはカーネル内の a 番物理ページに割り当てられている．(B) 領域の 0 番仮想ページはカーネル内の i 番物理ページに割り当てられている．これらの割り当ては上述したとおり Windows カーネルが暗黙的に決定する．これらの割り当てはすべてエミュレーションレイヤのマッピング管理テーブルがすべて把握している．

次に KVM に修正を加える．これは KVM が内部でゲスト OS 用の物理メモリを割り当てるときに, QEMU のどの仮想アドレスを割り当てようとしているのかをエミュレーションレイヤに対して通知できるように改良を加えるものである．具体的には, KVM 内でゲスト OS 用のメモリ領域を割り当てる関数である `kvm_main.c` 内の `kvm_vm_ioctl_set_memory_region()` にパッチを適用した．図を使って説明する．KVM は QEMU の仮想アドレスに対して物理ページを割り当てるべく `alloc_pages()` と呼ばれる関数を複数回呼び出す．図では (A) 領域 と (B) 領域であわせて 11 ページ分のメモリがゲスト OS の領域として割り当てられているため，11 回 `alloc_pages()` 関数が呼ばれて KVM は 11 個の物理ページを得る．KVM が `alloc_pages()` を 1 回呼び出すごとに，どの仮想ページがどの物理ページに対応づけられるかが決定され，これは KVM 内部のマッピング管理テーブルに登録される．パッチはこの `alloc_pages()` 関数を `wk_alloc_pages(pnum, base)` に置き換える．この関数は 1 つの空ページを返す点では `alloc_page()` と同じであるが `pnum, basefn` の 2 つの引数をとる点で異なっている．これは，KVM が `wk_alloc_page()` を呼び出したとき，どの QEMU の仮想ページ $(basefn + pnum)$ にどの物理ページを割り当てようとしているのかをエミュレーションレイヤに伝えるためである．エミュレーションレイヤはこの二つの引数を受け取り，すでに存在するマッピングと整合性が取れるように適切な物理ページを返す．図を使って説明すると，`wk_page_alloc(0, 0x00000000)` が呼ばれたときには，エミュレーションレイヤは a 番の物理ページを返す．なぜなら，すでに，エミュレーションレイヤがメモリマッピングを作っているため，0 番仮想ページに割り当てる物理ページは a 番でなければならないからである．つまり，`wk_alloc_page()` 関数は，KVM がゲスト OS 用のメモリを割り当てるために使用する特別仕様の `alloc_page()` 関数であるといえる．

結果的に，エミュレーションレイヤのマッピング管理テーブルと KVM 内部のマッピングテーブルが完全に同一のものとなる．これで，KVM の nopage /fault ハンドラを使用せずとも Windows 上で QEMU と WinKVM ドライバ間でメモリマッピングを行うことができる．なお，KVM 内の `fault / nopage` ハンドラはエミュレーションレイヤが呼び出すことはない．つまり，WinKVM 上で nopage /fault ハンドラが使用されることはない．そのため，ンレイヤは Linux の `fault / nopage` ハンドラをエミュレーションすることなく，WinKVM

のアドレスを得ることができなくなってしまうからである．

を動作させることができた．

しかしながら，上述したように，改造を施した KVM でもゲスト OS に 300MB のメモリしか割り当てられない問題は解決しない．この問題を解決するためには，QEMU のメモリ割り当ての手法を根本的に変更する必要がある．これについては，4.3 節章にて解説する．

4.3 移植性の高い Hybrid VMM を行うための指針

われわれは，エミュレーションレイヤをつかって WinKVM を実現する道程で，Windows と Linux の双方で動作する移植性のある Hybrid Hypervisor を設計する上で守らなければならない重要な知見を得ることができた．

もっとも重要な点は，ゲスト用のメモリをホスト OS のユーザ空間上にて大きな一枚の連続したメモリ領域 (ポインタ) として管理するべきではない点にある．つまり，対象となる Linux デバイスドライバが `fault / nopage` ハンドラを利用している場合，本レミュレーションレイヤを利用して Windows 上で動作させることは不可能である．

KVM/QEMU の場合ゲスト用のメモリをホスト OS から見たユーザ空間上に一つの連続したメモリ領域 (ポインタ) として確保しており，それを前提としてコードが書かれている．すなわち，一部の QEMU デバイスがゲスト OS のメモリ領域を読み書きするとき，通常のポインタ操作でこれを実現している．このメモリ確保の手法では Windows と Linux 間で互換性のある HybridHypervisor を実装することができない．つまり，ゲスト OS 用のメモリとして 512MB の領域を割り当てること考えたとき，ホスト OS のユーザ側からこれを `mmap()` や `VirtualAlloc()` 等の関数で 512MB の 1 つの連続したメモリ領域として確保するべきではない．Linux だけで動作する Hybrid 型 Hypervisor を考えるのであれば，この設計で特に問題はない．しかし，Windows と Linux で動作する移植性のある Hybrid 型 Hypervisor を考えるとこのアーキテクチャは大きな問題を引き起こす．Windows では一度にマッピングできるメモリの領域が限られているため，ゲスト OS のメモリの割り当て制限を引き起こすのである．

移植性の高いメモリ管理手法を**図** 4.13 に示す．例えば，512MB のメモリをゲスト用として割り当てる状況を考える．このとき，512MB の領域を複数のメモリチャンクに分割して，例えば，4 つに分割した 128MB のメモリ領域としてホスト OS のユーザ側にそれぞれ独立してマッピングできるように設計する必要がある．つまり，4 つの独立した非連続のメモリチャンクを用意し，それぞれ独立してユーザ空間にマッピングする必要がある．Windows では一つのメモリチャンクがマッピングできるメモリの容量は限られている (2GByte の物理マシンを搭載したマシンで一度に 300MB のメモリチャンクしかマッピングできない)．しかし，4 つメモリチャンクを独立してそれぞれマッピングすることで，512MB のメモリ領域をユーザ空間に割り当てることができるようになる．なお，WinKVM にこの手法を適用するためには，上述したとおり，ゲスト OS 用のメモリを分割して確保できるように QEMU と KVM のコードを大幅に書き直す必要がある．上述した通り，ユーザ空間にある QEMU ではゲスト OS を一つの大きなメモリ領域として確保しており，ポインタを通した通常のメモリ操作でゲスト OS へのアクセスを行っている．そのため，本論文で述べたエミュレーションレイヤではメモリ割り当て制限を回避することはできない．

上記の点が守られない場合，WinKVM のように，2GBytes の物理メモリを搭載していても，ゲスト OS に 300MB のメモリしか割り当てられないという事象が発生する．この知見は KVM の移植のためだけでなく，VT-x や AMD-SVM を使用する全ての Hybrid 型 Hypervisor に関係する話である．なぜなら，これらの拡張機能を使う Hypervisor を設計するにあたって，ユーザ空間のコンポーネントとカーネル空間のコンポーネント間でゲストのメモリ空間を共有（マッピング）するアーキテクチャを採用することははほぼ必須だからである．

Linux と Windows で動作する Hybrid 型の Hypervisor である VirtualBox はゲスト用のメモリを単一の連続したメモリ領域としてユーザ側にマッピングしていない．VirtualBox では，ホスト OS のユーザ側で動作するコードがゲスト OS のメモリを読み書きする際には，すべて `PDMDevHlpPhysRead()`,`PDMDevHlpPhysWrite()` 関数を経由して読み書きしている．これらの関数は，確保されたゲスト OS のメモリチャンクが独立してマッピングされていても，正確にゲスト OS のメモリを読み書きできるようにするためのものである．VirtualBox はゲスト OS のメモリを確保するとき，複数の非連続なメモリチャンクを別々に確保している．例えば，900MB のゲストメモリ確保するときには，300MB ごとのメモリチャンクを 3 個別々にマッピングすることで，結果的にゲスト OS には 900MB のメモリが割り当てられるような設計になっている．そのため，WinKVM で発生しているような，2GByte の物理メモリを確保しているにもかかわらず，ゲスト用のメモリ 300MB のメモリしか割り当てられないといった不適切な制限は発生しない．

4.4　WinKVM と KVM のライブマイグレーション

ここまで，われわれはレミュレーションレイヤを利用した WinKVM の実現手法について述べてきた．WinKVM の完成度は高く，KVM と WinKVM 間のライブマイグレーション機能を動作させることも可能である．ライブマイグレーションの処理は，ほぼユーザ側で動作する QEMU 側から行われるため，カーネル側のエミュレーションレイヤに特に工夫を入れる必要はない．しかし，後述するライブマイグレーションの評価を理解するためには，KVM のライブマイグレーションのプロトコルについて詳細に理解する必要がある．そのため，ライブマイグレーションプロトコルについて詳解する．

4.5　KVM のライブマイグレーションプロトコル

Linux デバイスドライバを Windows 上で動作させるエミュレーションレイヤを開発したことで，KVM を Windows 上で動作させることに成功した．Linux は KVM がそのまま Windows 上で動作するため，当然，当初の目的であった，WinKVM と KVM 間のライブマイグレーションも達成することが可能である．

KVM のライブマイグレーションのシーケンス図を**図** 4.14 に示す．基本的な流れは VMware ESX Server の VMotion[46] や Xen のマイグレーション [47] と同様に Pre-copy と呼ばれる手法を用いている．

VM ライブマイグレーションを考える上で重要な点は VM のダウンタイムを可能な限り縮小することである．VM を構成するペイロードの中でもっとも支配的な要素は VM のメ

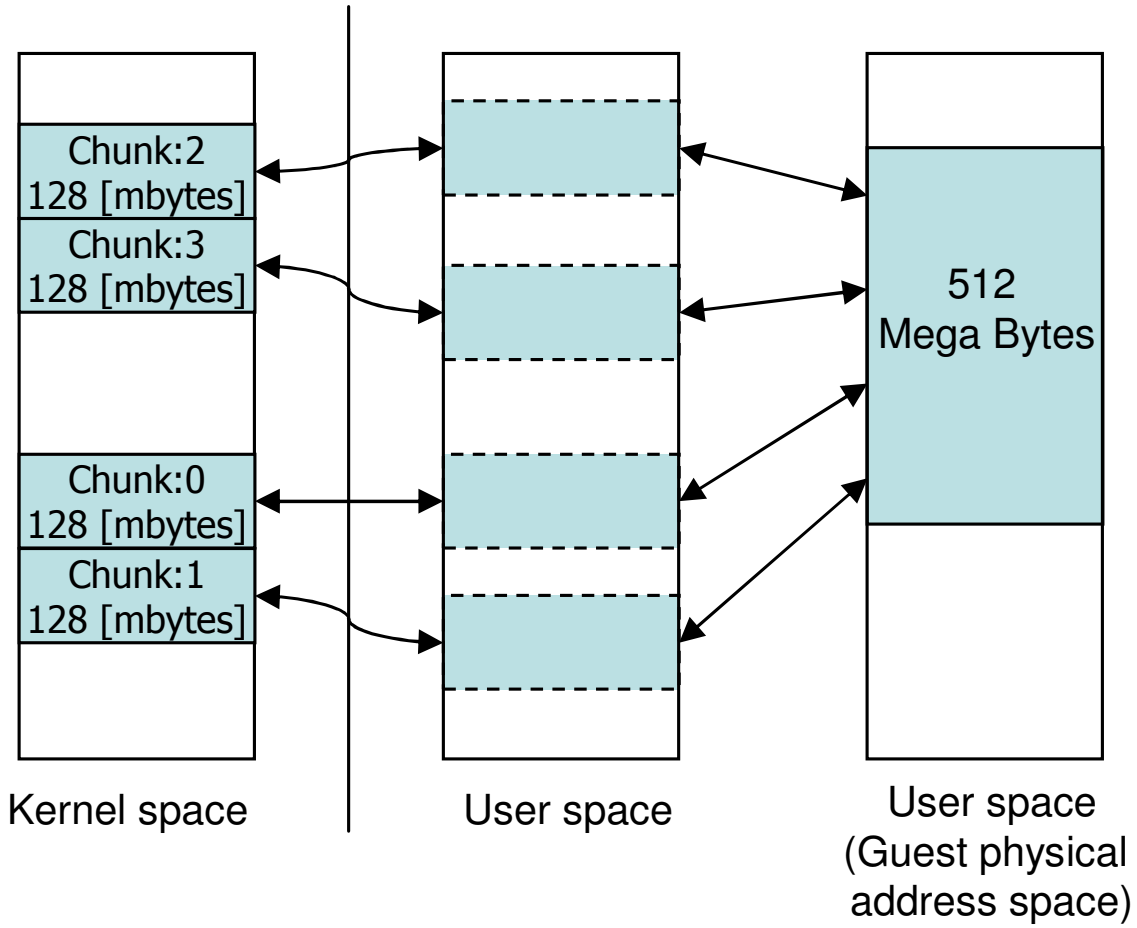


図 4.13. 移植性のある Hypervisor のためのメモリマッピング手法

モリデータである．これは巨大なデータであり，数ギガバイトになることもめずらしくはない．VM のダウンタイムを小さくするためには，このメモリ領域の転送手法を工夫する必要がある．

そのため，KVM のライブマイグレーション機構は VM を動作させつつメモリの転送 (Pre-copy) を行うことで，VM のダウンタイムを削減している．**図** 4.14 を用いて，ホスト B とホスト A 間でライブマイグレーションを行う例を考える．ライブマイグレーションのプロトコルは以下の 5 つのフェーズに分割して考えることが出来る．

1. 転送元ホストから転送先ホストへマイグレーション要求の送信．転送先ホストの資源の予約と初期化
2. メモリの転送
   (a) はじめに転送元 VM のメモリすべてを転送する (ファーストイテレーション)
   (b) さらにイテレーションを行う．このときは，前回のイテレーション時から更新されたページ (Dirty ページ) のみを転送する
   (c) Dirty ページが少なくなるまで (収束するまで) イテレーションを繰り返す

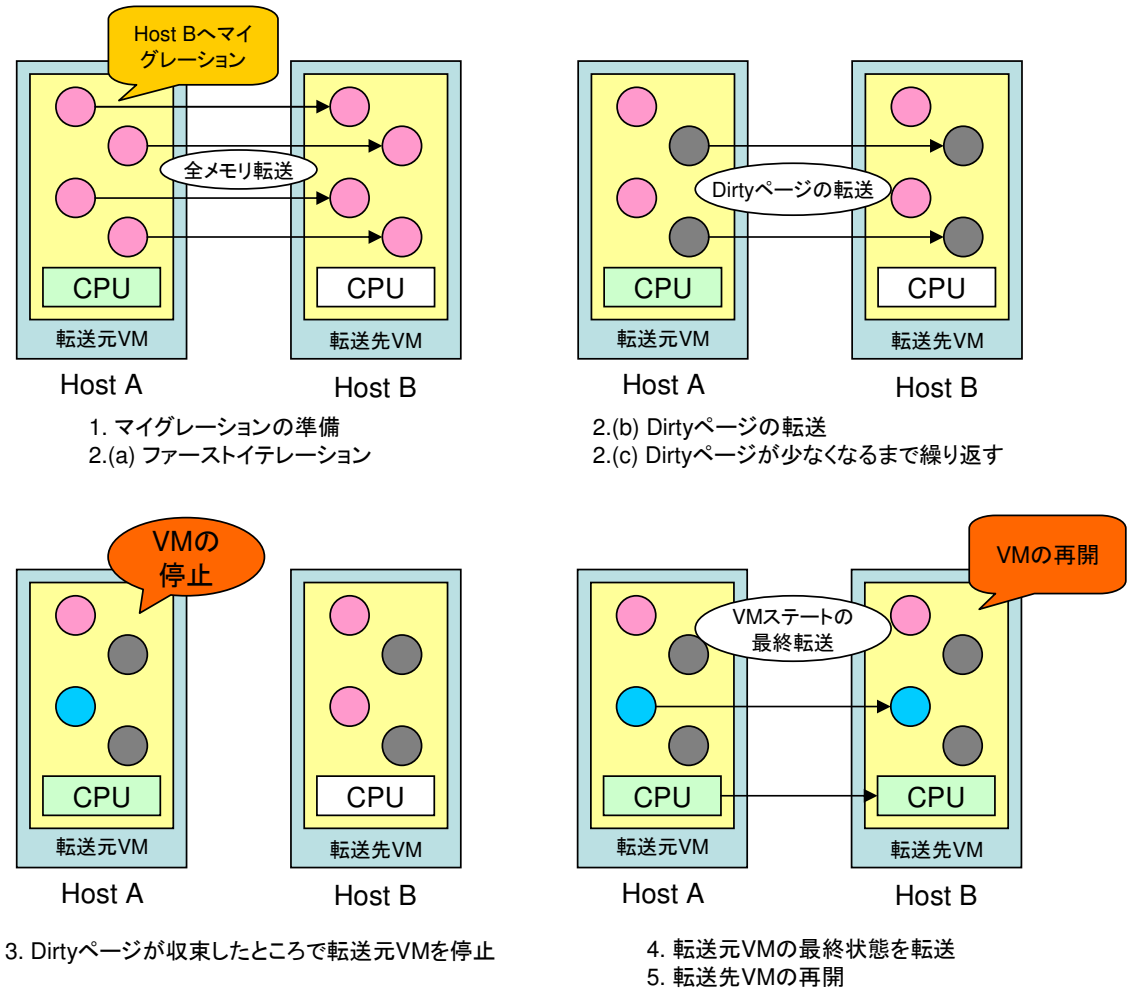


図 4.14. KVM の VM ライブマイグレーションメカニズムの全容図

3. 転送元 VM の停止．ある程度 Dirty ページが少なくなったところで転送元 VM を完全に停止する
4. 転送元 VM ステートの最終転送．CPU の状態や I/O デバイスの内部レジスタの状態などを転送する．転送元 VM が完全停止の直前に行ったイテレーションで見つかった Dirty ページもこの時転送される
5. VM の再開
   (a) 転送先 VM へのライブマイグレーションが成功すれば．転送元 VM を停止する．ネットワークがあれば，ARP ブロードキャストを使用して IP アドレスと MAC アドレスの対応付けを更新
   (b) 転送先 VM へのライブマイグレーションが失敗すれば．転送元 VM を再開する

2.(c) の収束条件を変えることによりライブマイグレーションの性能を変更することができる．Dirty ページの残りを少なく設定すれば (3) の実行時間が短くなるため，VM のダウンタイムが短くなる．しかし，少なく設定することにより (2) から (3) への遷移がいつまでたって

も起こらないという可能性がある．逆に大きく設定すれば (2) から (3) への遷移はすばやく完了するが，今度は (4) のフェーズに時間がかかってしまい，結果 VM のダウンタイムが長くなってしまう．

KVM は Dirty ページの数が 50 前後になった時点で (3) から (4) への遷移を行っているようである．この値は変更可能であり，ユーザがライブマイグレーションに求める性能に応じて調整可能である．以上が KVM のライブマイグレーションプロトコルの概要である．

4.6 WinKVM でのライブマイグレーション実装

ライブマイグレーションの処理はほぼユーザ側で動作する QEMU 側から行われるため，カーネル側のエミュレーションレイヤに工夫を入れる必要はない．WinKVM でライブマイグレーションを達成するためには，エミュレーションレイヤと，**図 4.2** に示した QEMU 側から WinKVM を制御するための `libkvm` を正確に移植する必要がある．例えば，KVM ドライバ側から VM の MSR を取得するための `kvm_get_msrs()` 関数の移植も正確に行う必要がある．この関数は，WinKVM 上で CentOS 5.4 を起動させるためには不要であるが，ライブマイグレーションを実現するためには必要不可欠である．

また，KVM-17 と WinKVM が使用する QEMU のバージョンを正確にそろえる必要がある．KVM-17 はデフォルトで QEMU 0.9.0 を使用しているが，一方で，WinKVM は Windows の移植版が存在する QEMU 0.9.1 を使用していた．なぜなら，Windows に移植されている QEMU に 0.9.0 は存在しなかったためである．QEMU のバージョンが違うと，送信受信間での QEMU デバイスのレジスタセットの整合性が取れなくなる．これは，両者の QEMU 間でレジスタセットをそのまま送信するライブマイグレーションにとっては致命的である．ライブマイグレーションを行うためには KVM が使用する QEMU を 0.9.0 から 0.9.1 に変更する必要がある．KVM-17 が使用する QEMU 0.9.0 には，KVM と連携を行うための修正が入っているが，QEMU 0.9.1 にはこのような修正が入っていない．そこで，QEMU 0.9.1 でも KVM-17 との連携を行うための修正を入れた．

4.7 評価

評価は大きく分けて 2 つの項目について行った．WinKVM の自体の性能評価と，VM ライブマイグレーションの性能評価の 2 つである．始めに，WinKVM 単体での比較を行い，われわれのエミュレーションレイヤが KVM に対して与えるオーバーヘッドを調査する．次に，エミュレーションレイヤが VM ライブマイグレーションが与える影響について調査する．これから述べてゆくベンチマーク結果を元に，本研究に論じた，エミュレーションレイヤによる移植手法は，性能劣化を引きこすことなく，Linux / KVM ドライバを Windows 上に移植することがとを議論する．

4.8　WinKVM の性能評価

始めに, WinKVM 自体の性能測定を行った．評価環境は以下のとおり, DELL LATITUDE D630 ラップトップコンピュータを SingleCore モードでブートして使用した．CPU は Intel Core 2 Duo T7300 2.0GHz で，物理メモリは 2GB である．HDD は WDC 製の 7200rpm HDD で 8MB のキャッシュメモリが搭載されている．ホスト OS は CentOS 5.4 で，カーネルのバージョンは 2.6.18-164.6.1.el5 (32bit, 100Hz の割り込みレート) と，Windows XP (Service Pack 3) を使用した．ゲスト OS はすべて CentOS 5.4 で，カーネルのバージョンは 2.6.18-164.6.1.el5 である．ゲスト OS に割り当てられたメモリは 300MB である．

4.8.1　Linux/UNIX nbench による評価

WinKVM の性能測定では，始めに，computation-intensive なベンチマークとして Linux/UNIX nbench プログラム [48] を使用する．このプログラムは Mayer 氏によって書かれたベンチマークプログラムであり，NUMERICSORT, STRINGSORT, NeuralNetwork シミュレーション，LU DECOMPOSITION となどいったいくつかのプログラムを含んでいる．nbench によるすべての評価の値は 30 回行ったうちの平均値である．

測定に当たって，Linux/UNIX nbench プログラムに 2 つの改良を加えた．これには 2 つの理由がある．

1 つ目に，nbench は対象コンピュータの性能を事前に測定し，最適なベンチマークのループ回数を決定する．また，結果として出るスコアは nbench オリジナルのスコアである．われわれは，この事前測定はベンチマーク結果に不明瞭な要素を与え，さらに，nbench のオリジナルスコアはわかりにくいと考えた．そのため，各ベンチマークプログラムが最低でも 20 秒以上の消費するようにループ回数を固定し，スコアはベンチマークプログラムが費やした秒数を表示するように nbench を修正した．

2 つ目に，一般的に Hypervisor 上での時計は不正確であるため，時計に依存しているベンチマーク結果も同様に不正確である．改造していない nbench を WinKVM 上と KVM 上で走らせてベンチマーク結果を得たところ，WinKVM がオリジナルの KVM よりも 10 倍以上高速であるという結果が出た．この結果はまったく信頼できない値である．正しい測定結果を得るため，われわれは，物理的なマシンを一台用意しそのマシンで上で時間測定を行った．より具体的に言うと，nbench プログラムが使用している `gettimeofday()` 関数を，ネットワーク経由の外部マシン上で実行できる仕組みを構築し，それを使用してベンチマーク結果をとった．この時，ネットワーク遅延による時間測定のずれを考慮する必要がある．この遅延をなるべく減らすため，以下に示す 2 つの条件下での実験を行った．1 つ目は，Nagle アルゴリズムを OFF にした状態で測定を行った．2 つ目は，実験で使用する LAN 環境上でベンチマーク測定のためのパケット以外は一切流れていない状態で測定を行った．むろん，以上の 2 つを考慮してもネットワーク経由の時間測定である以上，ある程度の遅延は避けられない．しかし，今回行った nbench によるすべての測定結果はまったく同一 LAN 環境下での実験結果であ

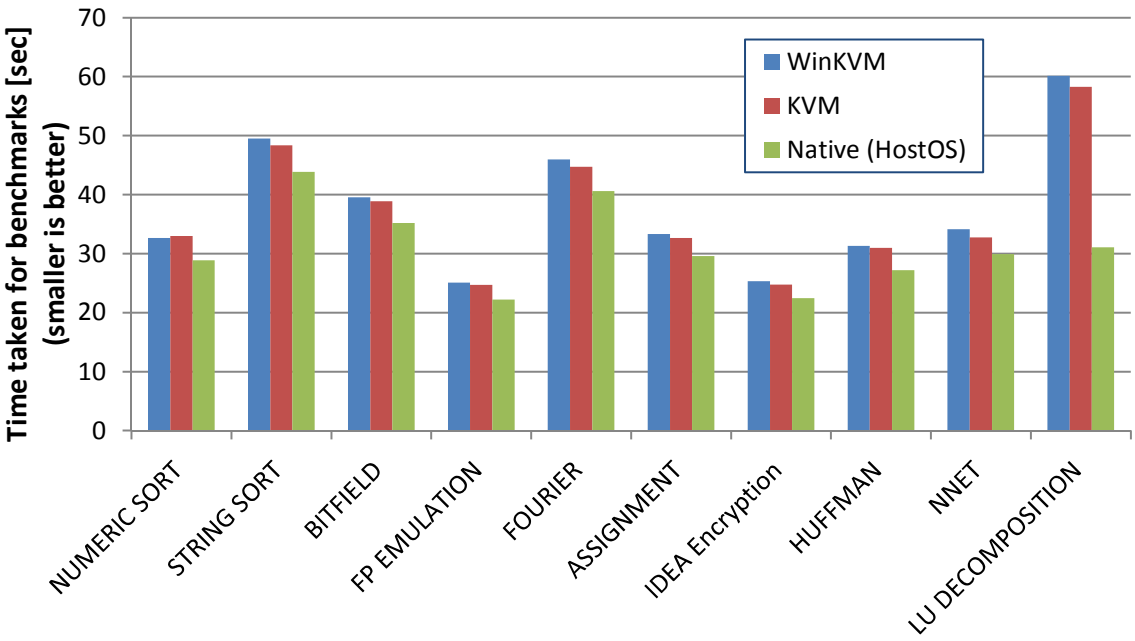


図 4.15. Linux/UNIX nbenchmark プログラムの測定結果

る．ping コマンドによる測定では，ネットワーク遅延の大きな乱れは見られなかったため[*2]，3 種類の値 (ホスト OS 上，KVM 上，WinKVM 上) に対してかかるネットワークの遅延は安定しているため，相対的にみると公平な値であり，比較して論じることが可能である．

図 4.15 に nbench の測定結果を示す．なお，参考までにホスト OS (CentOS 5.4) で実行したときのベンチマーク結果もグラフに加えた．全体的にみて，ベンチマーク結果からエミュレーションレイヤが WinKVM に致命的な影響を与えないことがわかる．最も性能低下を引き起こしている NNET ベンチマークでも，約 3.9% の性能低下にとどまっている．また，BITFIELD や FP EMULATION，HUFFMAN ベンチマークの性能低下は 2% 以内である．さらに，NUMERICSORT ベンチマークでは WinKVM と KVM の間にほとんど性能差がみられないという結果が出ている．つまり，nbencn のような computation-intensive なベンチマークでは，エミュレーションレイヤのオーバーヘッドは無視できるレベルであり，computation-intensive なプログラムは “性能劣化を引き起こすことなく”Linux / KVM ドライバを移植することが可能であることが分かった．

4.8.2　forkwait ベンチマークによる評価

次に，virtualization-sensitive なベンチマークとして，図 4.16 に示す FORKWAIT マイクロベンチマークを使用する．これはプロセスの生成と破棄を 40000 回繰り返すプログラムである．

評価の結果，ホスト OS での実行時間は 3.296 秒，KVM では 53.436 秒，WinKVM では

[*2] 測定機器へと 30 回 ping を送ったときの平均値は 0.283ms, 標準偏差は 0.0013 である

```
int main(int argc, char *argv[])
{
  int i;
  int status;
  for (i = 0 ; i < 40000; i++) {
    int pid = fork();
    if (pid < 0) return -1;
    if (pid == 0) return 0;
    waitpid(pid, &status, 0);
  }
  return 0;
}
```

図 4.16. FORKWAIT ベンチマークのソースコード

54.304 秒であった．WinKVM は 1.6% の性能低下しか引き起こしていないことがわかる．つまり，FORKWAIT のような virtualization-sensitive なプログラムでも “性能劣化を引き起こすことなく”Linux/KVM ドライバを Windows に移植できていることが分かった．

4.8.3　ApacheBench による評価

さらに，より現実的なプログラムを用いて WinKVM の性能を評価するため，Apache server benchmark を使用した．サーバに Apache/2.3.3 を使用し，10000 回の HTTP リクエストを同時接続数 1 として WinKVM と KVM-17 に対して処理させた．LAN 上にある計算機にて AB (Apache server benchmark) を実行して測定した．**図** 4.17 に評価結果を示す．10000 回のリクエストを処理する時間が，WinKVM が 102.33871 秒であるのに対して，KVM-17 (100Hz) は 39.831594 秒であることが判明した．つまり，WinKVM のほうが約 62.507 秒低速であるという結果が出た．

既存研究にて，OS の割り込みレートの違いによってベンチマークの測定値が変わるという現象が報告されている [49]．われわれは，ApacheBench において，WinKVM が低速である理由が Linux の割り込みレートと Windows の割り込みレートの違いにあると考えた．

この仮説を検証するために 2 つの実験を行った．1 つ目の実験は，異なる割り込みレートを持つ Linux カーネルを 3 つ (60Hz, 1000Hz, 1024Hz) 用意して ApacheBench による計測を行った．これらの割り込みレートは代表的な OS で使用されている値になっている．2 つ目の実験は，Windows カーネルの割り込みレートを `timeBeginPeriod()`API によって変更して，同様に ApacheBench による計測を行った．

1 つ目の実験結果を**図** 4.17 に示す．Linux の割り込みレートによって Apache Bench の結果が大きく変わることが判明した．全体的に，割り込みレートが増加すると，速度が低下する．KVM (1000, 1024 Hz) と KVM (60, 100 Hz) では約 2 倍の性能差が見られる．Linux の割り込みレートが 1024 Hz の場合，差はあるものの，WinKVM の速度とほぼ同等になる．

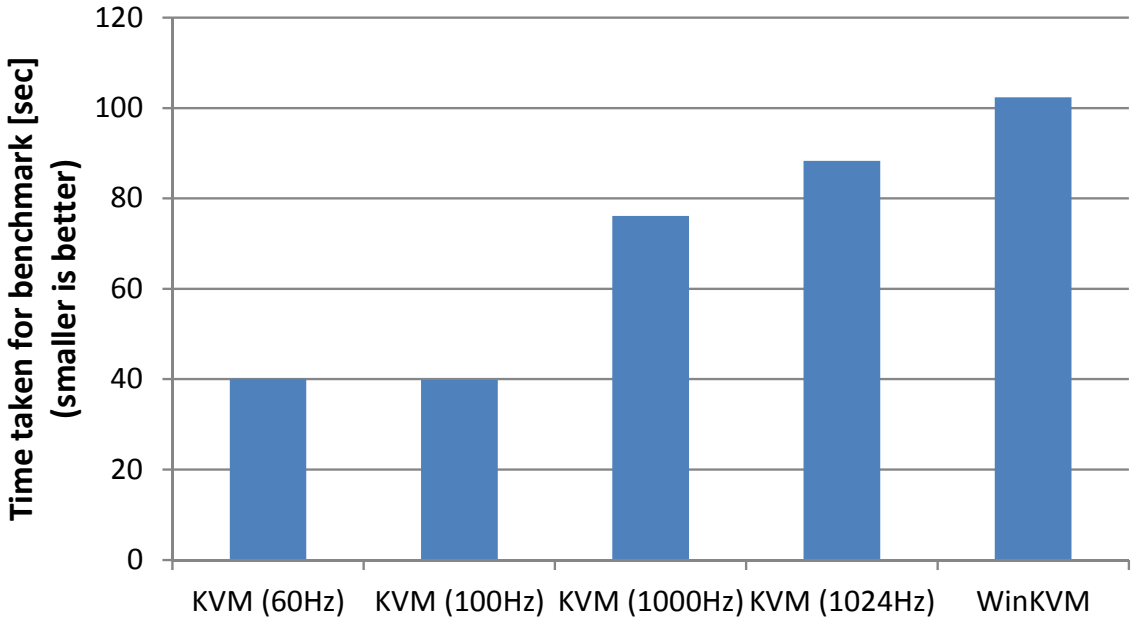


図 4.17. ApachBench で 10000 リクエストにかかる時間の平均値

2 つ目の実験結果を図 4.18 に示す．Windows 側の割り込みレートを変更した上で ApacheBench による計測を行った．全体的に，タイマーの分解精度を 100ms から 1ms へ変更してゆくと (割り込みレートを高くすると) ApacheBench の結果は KVM (割り込みレート 1000Hz) の値に近づくことが分かった．

上記の 2 つの実験結果より，WinKVM が低速である理由が Linux の割り込みと Windows の割り込みレートの違いにあることが証明された．

結果的に，WinKVM のほうが 2 倍高速であることがわかり，これまでのベンチマークと同様に “性能劣化を引き起こすことなく”Linux/KVM ドライバを Windows に移植できていることがわかる．

4.9　ライブマイグレーションの性能評価

われわれは次に，エミュレーションレイヤが，ライブマイグレーションに対して致命的なパフォーマンス低下を与えていないかを調査するための評価を行った．

4.4 節で述べたように，ライブマイグレーションのプロトコルは 5 つのフェーズに分けて考えることができる．評価は 2 つ行った．始めに，KVM-17 から KVM-17 に向けて VM ライブマイグレーションを行い，4 つのフェーズがそれぞれどの程度の時間を消費しているかを測定した．次に，KVM-17 側から WinKVM 側に VM ライブマイグレーションを行い，同様に 4 つのフェーズがどの程度時間を消費しているか測定を行った．最後の 1 フェーズは VM を再開するだけの処理なのでベンチマーク結果には含めない．なお，全てのベンチマークの値は 3 回行ったうちの最良値である．

測定環境には 2 台のコンピュータをネットワークで接続して行った．1 台目 VM 送信元の

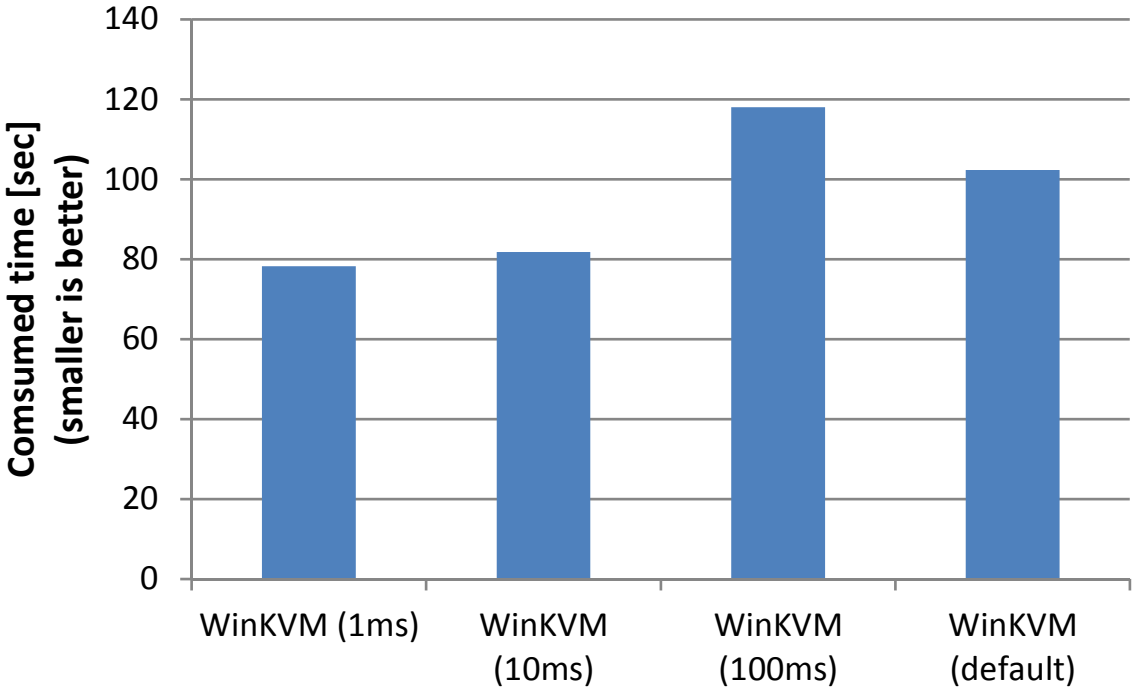


図 4.18. ApachBench で，WinKVM 上で異なる割り込みレートにおいて，10000 にリクエストかかる時間の平均値

コンピュータは IBM ThinkPad X60 を SingleCore モードでブートして使った，CPU は Inte Core DUO T2400 @ 1.83GHz．物理メモリ 2GB である．ホスト OS は CentOS 5.4 で，カーネルのバージョンは 2.6.18-164.6.1.el5 (32bit) である．2 台目のコンピュータは 4.8 節の評価で用いた DELLLATITUDE D630 ラップトップコンピュータである．ホスト OS はホスト OS は CentOS 5.4 で，カーネルのバージョンは 2.6.18-164.6.1.el5 (32bit) と WindowsXP (Service Pack 3) である．そして，これら 2 つのコンピュータは GIGABIT ハブで接続されている．ライブマイグレーションに使用した VM には 300MB のメモリが割り当てられており，ゲスト OS は RedHat 9 (カーネルバージョン 2.6.20) をベースにした QEMU のテスト用 Linux である．これは QEMU の Web ページからダウンロードすることができる．なお，それぞれ二つの VM には`/dev/zero` を 100MB 分 `mmap()` した後，確保した 100MB の領域を 4KB ページごとに線形に書き込んで行くという負荷をかけた上でライブマイグレーションを行った．

図 4.19 の中央に KVM-17 から KVM-17 へ VM ライブマイグレーションの結果を示す．VM ライブマイグレーションに費やした時間は 781 ミリ秒である．全ての処理の中でもっとも支配的な要素は 2nd フェーズの VM メモリ転送であり，全体時間の 91% を占めていることがわかる．次に支配的な要素が 3rd-4th フェーズである．これは VM の最終ステート転送であり，9% を占めていることがわかる．最後に，1st フェーズが 1% 未満でありほとんど時間を使用していないことがわかる．

次に，図 4.19 の左に KVM-17 から WinKVM への転送時間を示す．VM ライブマイグレーションに費やした時間は 703 ミリ秒である．全ての処理の中でもっとも支配的な要素は，同

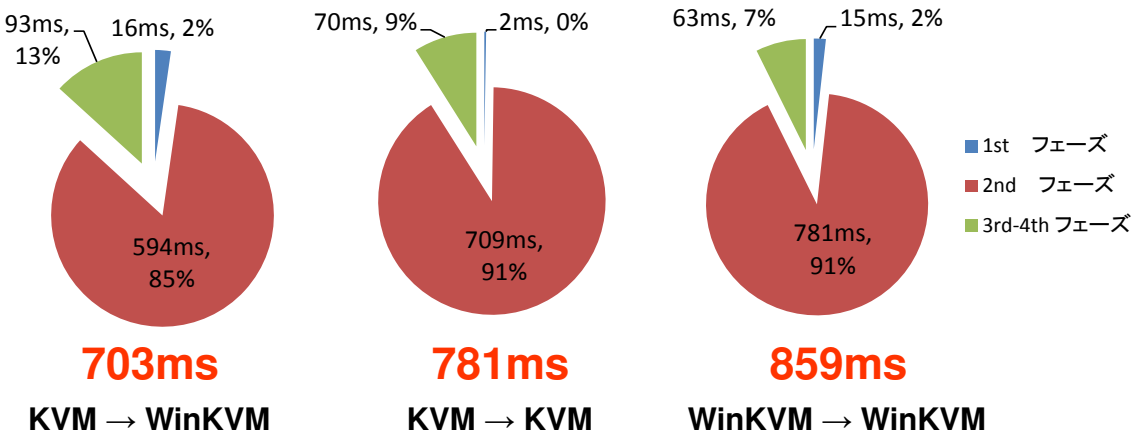


図 4.19. ライブマイグレーションの測定結果

様に 2nd フェーズの VM メモリ転送であり 85% を占めている．次に支配的な要素も同様に，3rd-4th フェーズであり 13% を占める．最後に，1st フェーズが 2% の時間を占める．KVM-17 から KVM-17 のマイグレーションを基準にして，マイグレーションの全体時間では 11% の性能変化，2nd フェーズでは 19% の性能変化，3rd-4th フェーズでは 24%，1st フェーズでは 87% の性能変化がみられる．

最後に，図 4.19 の右に WinKVM から WinKVM へのライブマイグレーションの結果を示す．VM ライブマイグレーションに費やした時間は 859 ミリ秒である．すべての処理の中で最も支配的な要素は，同様に 2nd フェーズであり全体時間の 91% を占めている．次に支配的な要素も同様に 3rd-4th フェーズでありこれは全体の 7% を占めている．最後に，1st フェーズが 2% の時間を占める．KVM-17 から KVM-17 へのマイグレーションを基準にして，マイグレーションの全体時間では 9% の性能変化，2nd フェーズでは 9% の性能変化，3rd-4th フェーズでは 11% の性能変化，1st フェーズでは 86% の性能変化がみられる．

3 つのライブマイグレーションの結果を比較する．エミュレーションレイヤによる性能変化は 1st フェーズで最大 87%, 2nd フェーズで最大 19%,3rd-4th フェーズでは 24% である．しかし，マイグレーション全体の性能変化は最大でも 11% の性能変化にとどまっている．また，実測値を比較すると，それぞれの差は数十秒ミリ秒とほとんど変わらず，エミュレーションレイヤはライブマイグレーションに対して致命的な性能低下を与えるものではないということがわかった．

評価の結果，KVM-17 から WinKVM へのライブマイグレーション速度のほうが，KVM-17 から KVM-17 へのマイグレーション速度より高速であるという結果が出た．

WinKVM-KVM 間のライブマイグレーションが高速である理由について考察する．われわれは，KVM (Linux) - KVM (Linux) 間と WinKVM (Windows) -KVM(Linux) の間ライブマイグレーションにおいて 2nd フェーズの占める割合が違うことに着目した．すなわち，KVM-WinKVM 間の TCP 転送のほうが，KVM-KVM 間の TCP 転送より高速であると考えられる．それ以外のフェーズである 1st, 3rd-4th フェーズは，全て KVM-WinKVM のライブマイグレーションのほうが低速という結果が出ている，これは QEMU が動作する Cygwin

環境のオーバーヘッドであると考えられる．まとめると，外部との通信をほぼ行わない 1st フェーズと，比較的少量の通信しか行わない 3rd-4th フェーズは Cygwin のオーバーヘッドにより低速になるが，大量の情報 (VM のメモリ情報) を転送する場合，TCP の性能差により 2nd フェーズが高速になり，結果的に KVM-WinKVM 間のライブマイグレーションが高速になるという結果が出るのではないかとの仮説を立てた．

この仮説の検証実験を行った．2nd フェーズの具体的な処理内容は，4101 バイト (4KB ページ + ヘッダ情報) の情報を連続的に送信するという処理内容である．この処理内容を擬似的に再現するシェルスクリプトを書き，Windows 上の Cygwin と Linux 間，Linux と Linux 間にて実行し，その間に要した時間を `time` コマンドで測定した．3 回測定しそのうちの最良値を採択した．実験の結果，Linux から Windows (Cygwin) への転送は 4.531 秒，Linux から Linux への転送は 4.616 秒であり，Linux から Windows (Cygwin) への転送のほうが 85 ミリ秒高速である．これにより，上記の仮説は正しいことが判明した．

結論として，われわれの評価環境においては，Linux から Linux への TCP 転送よりも，Linux から Windows への TCP 転送のほうが高速であるため，KVM-WinKVM 間の VM ライブマイグレーション転送のほうが高速になる．

また，エミュレーションレイヤがライブマイグレーションに与えるオーバーヘッドは無視できるレベルのものであり，同様に，“性能劣化を引き起こすことなく”Linux/KVM ドライバを Windows 上に移植できていることがわかる．

4.10　WinKVM 評価のまとめ

本章で行った評価をまとめる．Linux/UNIX nbench のような computation-intensive なベンチマークでは，エミュレーションレイヤが WinKVM に対して致命的なパフォーマンスの低下を引き起こしていないことが判明した．最も性能低下を引き起こしている NNET ベンチマークでも，約 3.9% の性能低下にとどまっている．次に，virtualization-sensitive なベンチマークとして，FORKWAIT ベンチマークを走らせた結果，WinKVM は 1.6% の性能低下しか引き起こさない．また，実用的なベンチマークとして ApacheBench を使用した測定を行ったところ，WinKVM のほうが約 60% 低速であるという結果を得た．この速度差はホスト OS の割り込みレートに起因するものであるという事がわかった．全体的にみて，エミュレーションのオーバーヘッドは無視できるレベルであることが判明した．

また，KVM-KVM, KVM-WinKVM, WinKVM-WinKVM のライブマイグレーションの評価結果では，それぞれ数十ミリ秒程度の変化しか見られず．同様に，エミュレーションレイヤがライブマイグレーションに与えるオーバーヘッドは無視できるレベルのものである．

結果として，性能評価の観点からは，本移植手法は，性能劣化なく Linux /KVM ドライバを Windows に移植できていることがわかった．また，人的コストのほうの “簡単” さに関しても，オリジナルの Linux / KVM ドライバ自体に修正を加える必要がある．このソースコードの書きえは，性能を犠牲にすることなく移植を達成するために必要なものであり，本手法の限界である．しかし，パッチの LOC は 10 行程度であり，本研究の，“簡単” かつ “性能劣化を引き起こすことなく”Linux/KVM ドライバを Windows に移植するという標は達成出来ている．

4.11 関連研究

ドライバのプログラムは，アプリケーションプログラムに比べて移植のコストが高いため，他のオペレーティング・システムのデバイスドライバを他のデバイスドライバに移植するという試みはいくつか行われている．

1 つ目の研究として，Linux ドライバを Solaris 上で動作させる PITS Library という研究が挙げられる [50]．この研究は，Linux のデバイスドライバのソースコードのコンパイル時に，PITS Library と呼ばれる，本研究で開発したエミュレーションレイヤ相当のプログラムををリンクさせ，Linux カーネルの互換レイヤを，移植対象となる Linux のデバイスドライバに提供することで，Solaris 上でも Linux ドライバが動作するシステムを提案している．しかし，この研究では，メモリマッピングに関する議論が含まれておらず，そこが本研究とは大きくことなる．そのため，PITS Library はメモリマッピングを利用する VGA ドライバ等はサポート対象外となっている．

2 つ目の研究としては，Linux と FreeBSD 上で，Windows の無線 LAN ドライバ (NDIS ドライバ) を動作させるプロジェクトが挙げられる．これらの研究は，それぞれ，Ndiswrapper[51],NDISulator (Project Evil)[52] と呼ばれている．第一に，これらのプロジェクトは Windows のドライバを Linux で動作させるシステムであるため，本研究とは逆の移植を行なっている．第二に，これらの研究は Windows の NIC ドライバのみを対象としている．Windows の NIC ドライバは，Windows カーネルが要求する NDIS ディスパッチハンドラを実装することで開発する．そのため，Windows の NIC ドライバを Linux で動作させるにあたって，本研究で議論した，実行可能な特権命令の違いや，メモリマッピングのエミュレーションは考慮する必要がない．そのため，これらの研究を比べると，本研究は，制限の弱い Linux カーネル上に実装された KVM ドライバを，比較的制限の強い Windows カーネル上に移植する手法について論じている．そのため，これらの研究とは異なる．

また，DSL（domain-specific language）を用いてドライバを記述すると，複数のオペレーティングシステムで動作するデバイスドライバをそれぞれ出力する研究/製品も存在する [53, 54]．これらの研究は，一度デバイスドライバのソースコードを DSL で書き直す必要がある．そのため，エミュレーションレイヤを用いて，既存の Linux / KVM ドライバを他の OS に移植する手法を提案した本研究とはそもそものアプローチが異なる．

また，さらに Linux / KVM を他の OS に移植という目標に直接的に関係する研究として，Porting Linux KVM to FreeBSD[55] があげられる．これは linux-kmod-compat とスタブドライバを利用して，KVM-17 を FreeBSD 上に移植したものである．linux-kmod-compat とは，Linux のデバイスドライバを FreeBSD 上でもコンパイルを可能にする互換レイヤである．このプロジェクトは Linux の互換レイヤを作るという点では本提案に似ている．しかし，Linux と FreeBSD を比較した時，Linux と Windows のドライバの相違の中でも，比較的エミュレーションが困難であった，ユーザプログラムとのインタフェイス，発行可能な Privilege な命令が Linux と FreeBSD の場合，linux-kmod-compat エミュレーションレイヤだけで十分に吸収可能であるため，本研究ほど違いを意識した設計を行う必要はない点が異なる．

また，Linux と Windows 間で動作する VMM の例として，VirtualBox があげられる．

VirtualBox は Windows や Linux をはじめとするさまざまな OS 上で動作する Hybrid 型 VMM である．VirtualBox は初めから高い移植性を持つように設計されており，予め入念な設計がなされたうえで，移植性を損なわないように設計/実装が行われている．これに対して本研究は，もともと移植性に関しては考慮されてない，Linux に強く依存している KVM を “簡単” かつ “性能劣化を引き起こすことなく”Windows に移植する手法について論じ，それが達成可能であることを示した．そのため，VirtualBox とは前提条件の違いにより複数 OS 間にて互換性をもたせるためのアプローチが異なっている．

4.12　Linux ドライバの Windows 移植による WinKVM の実装のまとめ

CAD，CAM やゲームといった，シンクライアント経由では利用が困難でるアプリケーションソフトウェアを利用するためには，手元にある物理的な計算機環境に VM を移動させ，手元の計算機上で直接動作させることが望ましい．

VM ライブマイグレーションと呼ばれる技術を活用すれば，必要なときにはリモートにある VM 自体をシームレスに手元のコンピュータに移動させることが可能である．しかし，現状の VMM では，異種 OS 間での VM ライブマイグレーションが不可能であり．これが利用可能性を低下させている．一般に，システム/サービス構築用には Linux が使われ，エンドユーザは，コンシューマ用として普及している Windows が利用される．そのため，Linux と Windows 間で VM ライブマイグレーションを達成することで，エンドユーザでもスムーズに VM を移動させて，上述したソフトウェアを VM 上でも，快適に利用することが期待できる．

そこで，本研究では，Linux で動作する KVM と呼ばれる VMM を Windows に移植することで，Windows と Linux 間での VM ライブマイグレーションを達成することを目指した．Windows に移植した KVM を WinKVM を名付けた．

KVM は Linux ドライバとして実装されている．一般論として OS のドライバはオペレーティングシステムに強く依存するプログラムである．アプリケーション・プログラムと比較すると，他のオペレーティングシステムへのドライバ移植は人的コストがかかる．そこで，本研究では，“簡単” かつ “性能劣化を引き起こすことなく”Linux/KVM ドライバを Windows に移植する手法を開発した．

具体的には，Windows と Linux のデバイスドライバアーキテクチャの違いを調べ，その違いを吸収する Linux レーションする Windows ドライバを開発した．そして，そのエミュレーションレイヤを用いて Linux/KVM を Windows 上で動作させることに成功した．

いくつかのベンチマークプログラムで WinKVM の性能を測定したところ，オリジナルの KVM との性能差は見られず，エミュレーションレイヤによる性能劣化は無いことがわかった．また，KVM-KVM, WinKVM-KVM, WinKVM-WinKVM 間のライブマイグレーションをそれぞれ比較したところ，性能劣化は数十ミリ秒程度に収まっている．つまり，本手法により “性能劣化を引き起こすことなく”Linux / KVM を Windows 上に移植することが可能であることが示された．

また，Linux エミュレーションレイヤを作るうえで直面した，開発環境の違いや，メモリアー

キテクチャの根本的な相違等，いくつかの困難か課題をどのように解決にしたかについて論じた．Linux の場合，カーネル空間とユーザ空間にてメモリマッピングを行う時，ページフォルトハンドラである `fault / nopage` ハンドラを利用することができる．一方で，Windows の場合，ページフォルトハンドラを使ってメモリマッピングを行うことはできないため，今回提案したエミュレーションレイヤでは，Windows 上で Linux の `fault / nopage` ハンドラを効率よく，本研究のエミュレーションすることは難しい．そのため，`fault / nopage` を利用したメモリマッピングを使用している Linux ドライバの場合，エミュレーションのみでは問題を解決できない．Linux / KVM ドライバ側のソースコードを修正する必要がある．これは，エミュレーションレイヤを使う本手法の限界である．しかし，修正に必要なパッチの LOC は 10 行程度であるため，すべての KVM プログラムを解読し，Windows ドライバとして移植する作業と比較すると，10 行程度の修正は “簡単” である．そのため，大きな問題にはならない．

また，これは本研究の目的とは直接的には関係しない副次的な成果ではあるが，Windows と Linux 間で互換性のある Hybrid 型 Hypervisor を設計する上で留意すべき知見を得ることもできた．もっとも重要な点は，ゲスト用のメモリをユーザ側にて大きな一枚の連続したメモリ領域として扱ってはいけないという点である．つまり，ゲスト OS 用のメモリとして 512MB の領域を割り当てること考えたとき，ユーザ側からこれを `mmap()` や `VirtualAlloc()` 等の関数で 512MB の 1 つの連続したメモリ領域として確保するべきではない．Linux だけで動作す る Hybrid 型 Hypervisor を考えるのであれば，この設計で特に問題はない．しかし，Windows と Linux で動作する移植性のある Hybrid 型 Hypervisor を考える際，このメモリアーキテクチャを採用すると，VM へのメモリ割り当て制限を引き起こしてしまうことを解明した．

今後の課題には，エミュレーションレイヤの完成度を向上し，最新版 KVM への対応を行うことが考えられる．現在のエミュレーションレイヤは KVM-17 が使用している Linux カーネル関数のみをエミュレーションしている．WinKVM が更なる発展を遂げるためには，最新の KVM がエミュレーションレイヤ上で動作できるようになることが望ましい．そのため，より包括的な Linux カーネルのエミュレーションが可能になるようにエミュレーションレイヤに追加を行うべきである．また，KVM に実装されている PCI-passthrought の機能に関しても，本手法と同様に “簡単” かつ “性能劣化を引き起こすことなく”Windows に移植可能かどうかも検証するべきであると考えている．PCI-passthrought は，ゲスト OS から物理マシンに接続されたデバイスを直接使用する技術である．VMM が，ゲスト OS に対して提供するデバイスのエミュレーションのコストが大幅に削減されるため，今後，仮想化インフラを支える主要な技術の一つになることが予想される．そのため，本研究で論じた同様の手法で PCI-passthrought 機能も含めた Linux / KVM ドライバが Windows に移植可能であるかどうかについても研究を進めるべきである．

第 5 章

差分転送を用いた高速な VM マイグレーション

3 章にて，専用ソフトウェアやアプリアンスをインストールするための技術的な知識を持たないユーザでも導入が容易なシンクライアントシステムを開発し，シンクライアントシステムの利用可能性を向上した．

次に，4 章では，異種 HostOS（Windows と Linux）間で動作する VM ライブマイグレーションを実現するために，Linux/KVM を Windows に移植し，これを WinKVM と名付けた．これで，コンシューマ用途として広く普及する Windows と，サーバ/サービス用途として広く普及する Linux 間での VM ライブマイグレーションが可能となり，VM ライブマイグレーションのり用可能性が向上した．

これら二つの研究に加えて，VM を他の物理マシンに完全転送する技術があれば，さらに利便性の高い仮想化環境を構築することが可能である．例えば，ユーザは，普段は耐故障性が高いクラウド上にて動作する VM を，シンクライアント VoXY を通して利用している．しかし，シンクライアントでは利用が難しいソフトウェア（CAD，CAM，3D ゲーム）や，出張で一時的にネットワークが利用出来ない環境でも自分の環境（VM）を持ち出したい場合には，VoXY 上で操作している VM を，スムーズに手元の計算機に VM をウンロードすることが出来ればより便利であると考える．

従来の（KVM，WinKVM 間を含め）VM ライブマイグレーションでは，VM のストレージデータが，ネットワーク透過な分散ファイルシステムや共有ファイルシステムで (NFS[56] や cifs）転送先と転送元との物理マシンとの間で予め共有されている必要があった．そのため，VM ライブマイグレーションを利用するためには，これらのファイルシステムが利用可能な状態にあることが前提条件であった．

しかし，分散ファイルシステムや共有ファイルシステムの利用/構築が困難な状況下において，VM ライブマイグレーションが必要とされる状況が存在する．図 5.1 に示すように，個人用の仮想化環境における VM ライブマイグレーションや，広域 WAN 環境における VM ライブマイグレーションである．後に詳しく論じるが，この場合，個人用に VM を使うエンドユーザが共有ファイルシステムや分散ファイルシステムを使い，て VM ライブマイグレーションを行うのは困難な点が多々存在するし，また，データセンタ間をまたぐ，広域 WAN 環境で

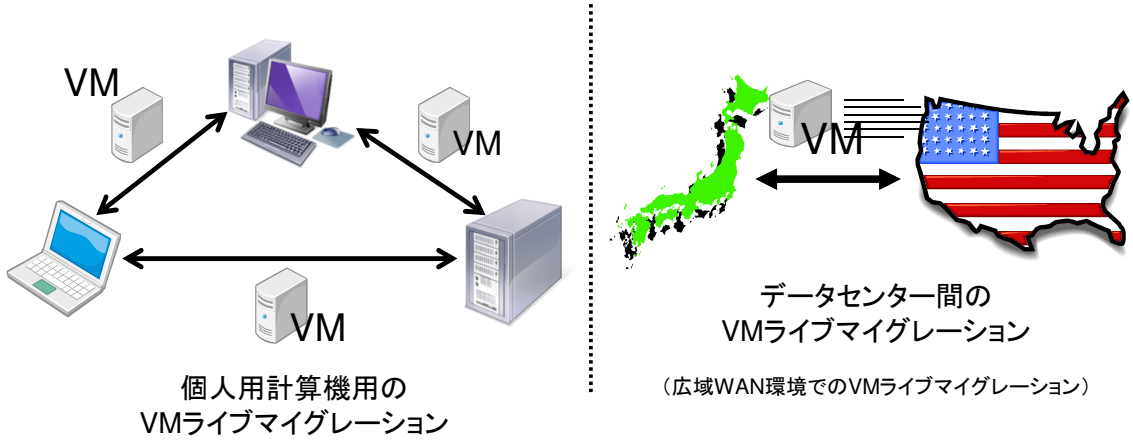


図 5.1. 広域 WAN 環境での VM ライブマイグレーションシナリオ

のライブマイグレーションでも，分散ファイルシステムや共有ファイルシステムを利用するのは，同様に困難である．

そのため，VM ストレージマイグレーション (VM ブロックマイグレーション) と呼ばれる，VM のストレージを含めて転送する技術が開発されている．これは，既存の VM ライブマイグレーションでは，RAM，CPU ステータス，デバイスステータスのみの転送に加えて，VM のストレージデータも含めて，他の VM 物理ホストにデータを転送する技術である．そのため，共有/分散ファイルシステムは必要なく，VMM 同士のみで VM を転送することができ，転送した VM は転送元とは完全に独立して動作することが可能である．

しかし，この新たな VM ストレージマイグレーションのメカニズムには転送に多くの時間や，広帯域なネットワーク環境要求されるという問題点が存在し，そのため，限定された環境下でしか使うことが出来ず，利用可能性が低い．VM のディスクイメージは一般的に巨大なファイルであり，数十 GByte になることも珍しくない．例えば，1Gbps の LAN 環境にて 20GB のディスクイメージを持つ VM をマイグレーションすると，VM ストレージの転送だけでも，数十分もの時間がかかる．マイグレーションが完了するまでに 10 分以上もの時間がかかるのはユーザの利便性を低下させる．

この問題点を解決するため，本章では，データセンタ間や，個人用 VM ライブマイグレーションでは，特定の物理マシンの間，VM が行ったり来たりする，VM の往来と呼ばれる挙動に着目し，シンプルな差分転送の手法を提案する．VMM から VM の I/O をフックする DBT (Dirty Block Tracking) と，VM のイメージに対して世代番号と呼ばれる概念を付加することにより，VM が，いつ（どの物理マシンで），どこが（VM ストレージの何処の部分が）変更されたかを検出し，転送の差異には，差分のみを正確に検出し，高速に転送する．日常的なオフィスワーク程度のコンピューティングであれば，コンピュータがディスクに与える影響は数 % であるため，この手法は VM ストレージマイグレーションの高速化に寄与する．

この手法を実際に Linux/KVM に実装し，日常的なオフィスワーク程度のコンピューティングであれば，VM ストレージマイグレーションを最大で 90% 以上高速化することに成功した．

差分転送を利用したブロックマイグレーション高速化のメカニズムを考案した [57]．例えば，A と B という 2 つの物理マシンが存在するとして，A から B に VM マイグレーションを行ったとする．その後，B から A へ再び VM ライブマイグレーションが行われたとき，転送元の A には，元の VM のディスクイメージが残されている．そのため，A から B の初期転送にかかる時間は大きいものの，その後，B から A に VM ライブマイグレーションが行われたときには，転送先の B で汚された VM のディスクページのみを転送すればよく，従来のブロックマイグレーションに比べて大幅に転送時間を短縮することが可能になる．なお，このユーザシナリオについては 5.1 節にて詳しく議論する．

5.1　問題分析

本章では，現状の VM ライブマイグレーション（ブロックマイグレーション）に何が不足しているのかを分析した後，われわれの提案が何を改良することを目的としているのかについて示す．この問題分析は，著者が以前に執筆した論文 [58] にも，同様のことが書かれている．

上述したように，すでに，VM ライブマイグレーションは，VM のストレージの転送に関して共有ファイルシステムや分散ファイルシステムに依存している．しかし，これらのファイルシステムの利用では対応できない事例が存在しており，今後それは増加してゆくと予想される．ここでは，具体的に，そのような事例の具体例を挙げる．

一つ目は，広域 WAN 環境におけるデータセンタ間でのライブマイグレーションのシナリオである．こういった環境では，移動元と移動先との物理計算機療法からアクセスできる共有ファイルシステムや分散ファイルシステムが必要になる．しかし，こういった，広域 WAN 環境を介してこれらのシステムを構築すると，LAN 環境よりも大きなネットワーク遅延により I/O 性能が低下してしまい，実用性を失う [59]．

二つ目は，VM ライブマイグレーションを個人用計算機環で利用するシナリオである．現在，ユーザに対して物理マシンを割り当てる代わりに VM を割り当て，ユーザは物理マシンを直接使用せず VM を使用する．実際に，MokaFive[60] というソリューションも販売されている．

このような個人用計算機環境上で使用される VM に対して VM ライブマイグレーションが導入されることで，より便利な個人用の仮想計算機環境が開発可能である．**図 5.2** に個人用計算機環境上で使用される VM ライブマイグレーションの概念を示した．これは，Intel の Shivani Sud らのレポート [61] に記載されていたシナリオのイメージ図である．彼らが挙げている個人用計算機環境上で使用される VM に対する VM ライブマイグレーションの導入シナリオを以下に引用する．

> A さんは，今朝のミーティングで提出する必要のあるプレゼンテーションを，自宅にあるパソコン (VM) を使ってレビューしている．出勤時間になったので，自宅のマシン上で使っていた VM 環境を丸ごと Netbook といった MID (Mobile Internet Device) に VM ライブマイグレーションを使ってシームレスに転送する．通勤途中の地下鉄の中でも，携帯端末上で自宅の PC 環境がそのまま使用できるため，プレゼンテーションの見直しを続けることができる．オフィスに到着すると，携帯端末上からオフィスにあ

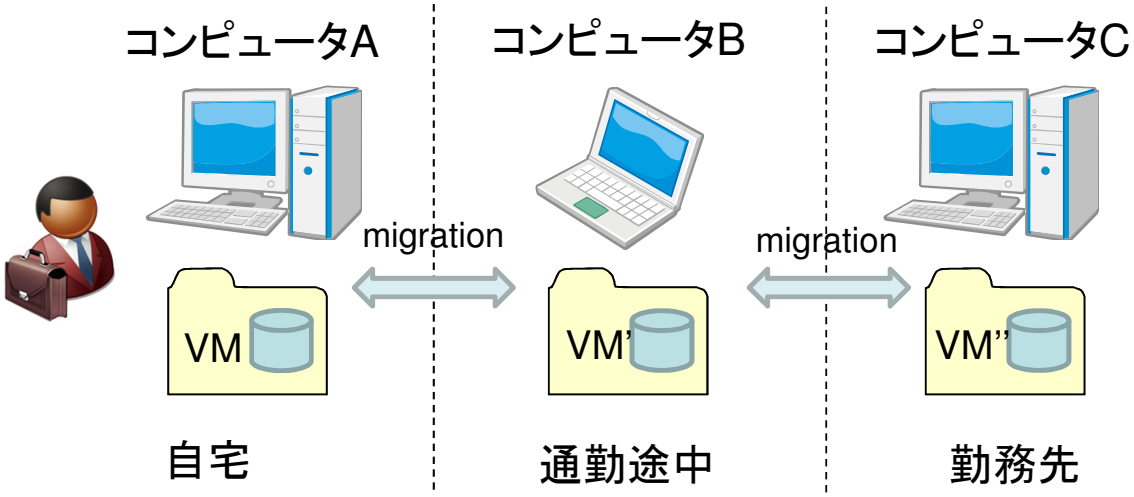


図 5.2. 個人用途のマイグレーション

る計算機へと VM ライブマイグレーションを使用して VM を転送して，通期途中の地下鉄の中で使っていた環境がオフィス上の計算機にシームレスに再現される．

上述した二つのケースでは，VM のストレージを含めて転送可能な VM ストレージマイグレーション（ブロックマイグレーション）といった技術が利用されることがある．しかし，既存のブロックマイグレーションには大きな問題点が存在する．それは，ブロックマイグレーションに要する全体の転送時間が大きくなってしまうことである．VM を構成するディスクイメージは一般的に巨大であり，数十 GByte の容量を持つものも少なくない．このレベルファイルサイズ (20Gbyte のディスクイメージ) をブロックマイグレーションで転送するには，現在普及している Gigabit の LAN 回線でも約 10 分もの時間を要する．

そのため，VM のライブマイグレーションに要する転送時間を低下させることが重要である．既存の VM ライブマイグレーション (ブロックマイグレーション) では，極論ではあるが，VM のダウンタイムさえ短ければ転送の全体時間は長くてもそれが問題になることはなかった．事実，既存の Pre-copy VM ライブマイグレーションプロトコル [47, 46] はそのような設計になっている．しかし，上述したケース，中でも，個人用途を想定した VM のライブマイグレーションの場合，転送の全体時間を低下させることは重要である．なぜなら，個人用の VM ライブマイグレーションにとって重要なことは，**ユーザが思い立った時に素早く VM の転送が行え，その後は一切の通信をすることなく VM が転送先が動作し，持ち運べることである．そのため，VM の転送に数十分もの時間を要するのは問題である．**

5.2 提案手法

本章では，5.1 節で議論した，分散/共有ファイルシステムの構築/利用が難しい状況下において**ストレージマイグレーション（ブロックマイグレーション）に要する全体転送時間をいかにして縮めるか**という問題点をどのように解決するかについて議論する．

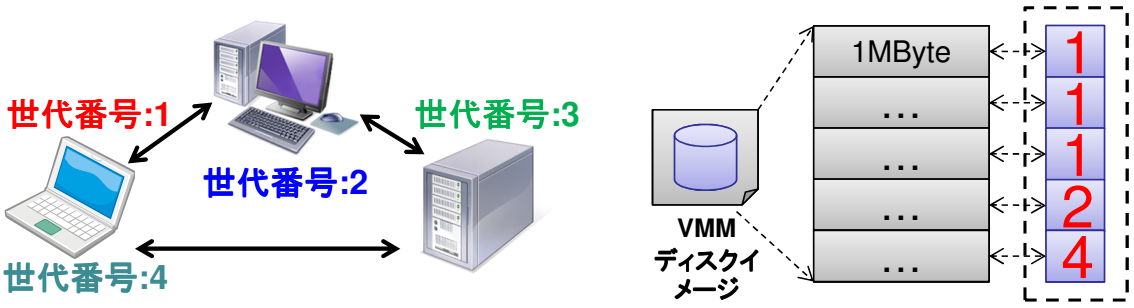


図 5.3. 提案手法の概要

5.2.1　提案手法の概要

本研究のアイデアの概念図を図 5.3 に示す．5.1 節で論じたように，特定の状況下では，VM ライブマイグレーションによって移動した VM が再び移動元の計算機に戻ってくる可能性が高い．すなわち，VM が特定の物理マシン間を往来する可能性があるという点について着目し，VMM から VM の I/O をフックする DBT（Dirty Block Tracking）と，VM のイメージに対して世代番号と呼ばれる概念を付加することにより，VM がいつ（どの物理マシンで），どこが（VM ストレージの何処の部分が）変更されたかを検出し，転送の差異には，差分のみを正確に検出し，高速に転送する手法を提案する．

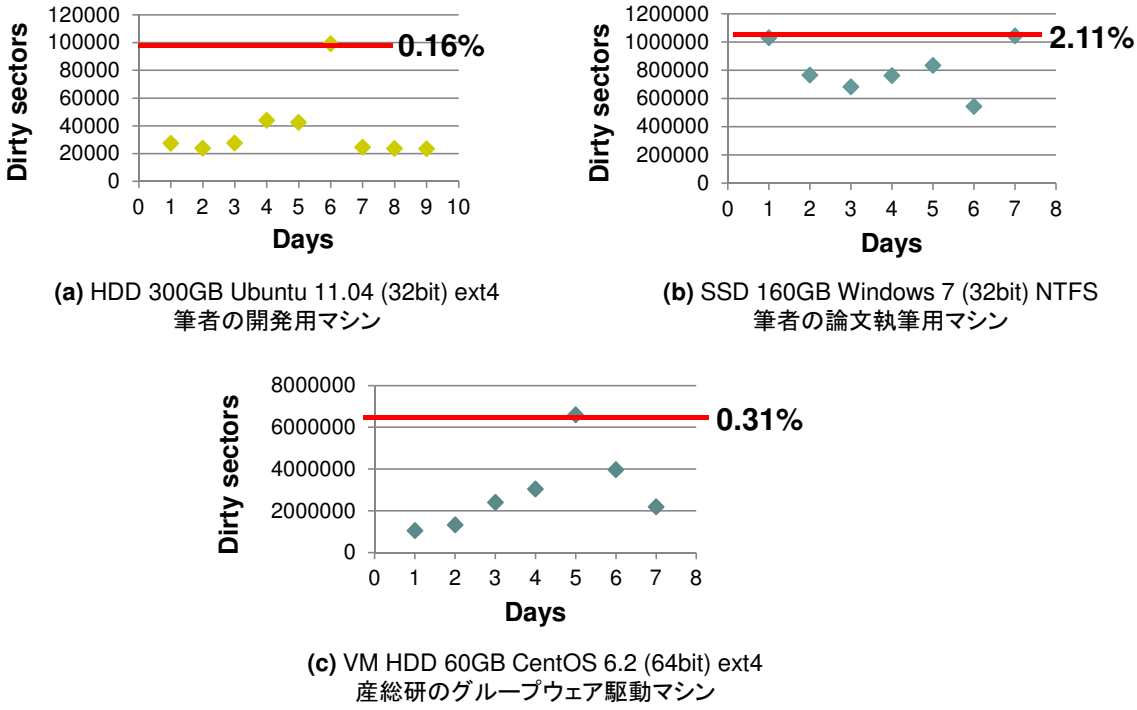
世代番号は VM が物理マシンを移動するたびにインクリメントされる．この世代番号は VM のストレージ（ディスクイメージ）に対してただ 1 つ割り当てられる番号である．この番号は，VM ストレージマイグレーションによって物理ホストを移動するたびにインクリメントされる．

次に，VM ディスクイメージは固定長のブロック（図中では 1MB）によって分割管理され，それぞれのブロックに対して，番号が 1 つずつ割り当てられる．もし，ディスクの書き込みが行われた時，その部分のブロックに割り当てられていた番号が，VM ディスクイメージが持つ 1 つの世代番号より小さかった場合，そのブロックの番号を世代番号と同じ値になるように書き換える．

つまり，どの物理マシンで，何処の部分のブロック変更されたかを検出することができる．差分転送の場合は，転送先の VM イメージの世代番号を確認した後，転送対象となるディスクブロックに割り当てられた番号を 1 つずつチェックし，転送先の VM イメージの世代番号よりも大きい数値を持つブロックのみを転送する．この仕組みによって，差分転送を行うことができる．

しかし，差分転送が実用的であるためには，VM 上で行われる日常的なコンピューティングにおいて，VM のディスクイメージの少量の部分のみが書き換えられられなければ効果を見出すことができない．極端な話，VM のディスクイメージが全て書き換わった場合，差分転送を利用しても，すべての VM ディスクイメージを転送する必要があり，差分転送による高速化は期待できない．

そのため，まず，日常的なコンピューティングにてどの程度のディスクページが汚されるか



(a) HDD 300GB Ubuntu 11.04 (32bit) ext4
筆者の開発用マシン

(b) SSD 160GB Windows 7 (32bit) NTFS
筆者の論文執筆用マシン

(c) VM HDD 60GB CentOS 6.2 (64bit) ext4
産総研のグループウェア駆動マシン

図 5.4. 3 つの実用的なコンピューティング環境における，一週間程度の一日におけるディスクのダーティブロック数

について予備調査をおこなった．3 つの実用的なコンピューティング環境 (VM1 つと，物理マシン) を用意し，そのディスクのブロックを 1MB に分割し，それぞれどの程度のブロックが汚れているのか数えた上で，セクタ数に換算した．なお，評価期間は約一週間である．

図 5.4 に実験結果のグラフを示す．**(a)** は 300GB の HDD を搭載しており，OS は Ubuntu 11.04 (32 bit) の筆者の開発用マシンである，このマシンでは主に Emacs テキストエディタによるテキストの編集と，GCC によるビルド作業が行われた．**(b)** は 160GB の SSD を搭載した論文執筆用のマシンである．このマシンでは主に，Emacs テキストエディタを使用したテキスト編集作業と，Microsoft Office 2010 による編集作業である．**(c)** は産業技術研究所の Web ベースのグループ管理ソフトウェアが動作する VM である．

調査の結果，すべての環境において，一日のコンピューティングにおいては，数 % 未満のディスクしか書き変わっていないことがわかった．そのため，VM が頻繁に往来する環境において，ディスクの差分転送は十分有効なアプローチであると考えられる．

5.3　手法の詳細

5.2.1 節にて論じた手法についてより詳細に述べる．

本研究では，差分転送による高速なブロックマイグレーションをサポートするための **diff ディスクイメージ** と名付けた新たな VM のディスクイメージを設計した．本章では初めに，DBT（Dirty Block Tracking）について述べ，次に，DBT を支援するための diff ディスクイメージがどのような構造を持つのかについて述べる．そして，diff ディスクイメージがどのよ

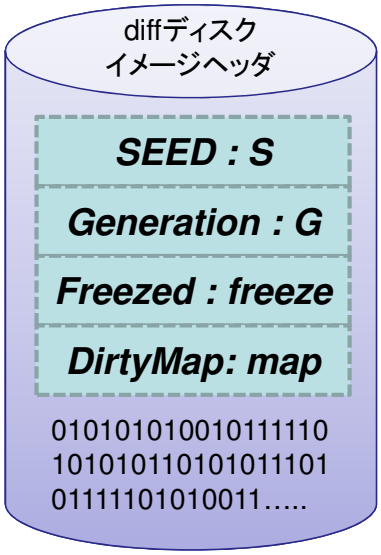


図 5.5. diff ディスクイメージのヘッダ

うにして差分転送による高速なディスクイメージ転送を行うかについて論じる．

5.3.1　ダーティーブロックトラッキング (DBT)

DBT は，ディスクの全書き込みを追跡しダーティマップにその記録を残すための機構である．DBT は VMM 側から捕捉できる VM のディスク書き込みをフックする．ディスクの全体領域を特定の大きさを持つブロックに区切り，1 ブロックにつき 8 ビットとしてダーティマップを構成することで DBT を実現する．8 ビットの空間には世代番号が書き込まれる．世代番号とは上述したとおり，VM の一つのイメージごとに割り当てられる一つの数値であり，VM が他の物理マシンにマイグレーションで移動したときや，VM が起動したときにインクリメントされる数値である．8 ビットの空間にはこの世代番号が書き込まれる．DBT とダーティマップによって，どの世代によって，どのディスクブロックが汚されたのかがすべて記録される仕組みになっている．

5.3.2　diff イメージフォーマットを用いたマイグレーション

diff イメージは，DBT で得たダーティマップを構造化して管理することにより，差分転送をサポートする VM ディスクイメージである．diff イメージの構造を図 5.5 に示す．

diff イメージヘッダには現在の世代番号 G, seed 番号 S, そのイメージが起動可能か否かを示す $freeze$, そして，どの世代によって，どのブロックが汚れたのかを記録するダーティページマップ Map によって構成されている．5.3.3 節にてこれらのデータ構造がどのようにしてブロックマイグレーションをサポートするかについて述べる．

5.3.3 マイグレーションアルゴリズム

本節ではわれわれが提案するマイグレーションが $A \longleftrightarrow B \longleftrightarrow C$ という 3 つの仮想マシン間でどのように動作するかについて詳解する．

図 5.6 に $A \longrightarrow B \longrightarrow C$ 間の転送がどのように行われるのかについて図示する．

初めに A で作られた diff ディスクイメージはユニークな seed 番号 S_0 (UUIDD : UUID generation daemon によって生成されたユニークな番号) が付与される[*1]．なお，初代の世代番号は 1 から始まる．

まず，A にある VM イメージが起動される．世代番号は OS が起動したときにインクリメントされ，1 から 2 になる (図中 A) この状態で，VM 上にて作業が行われ，世代番号 2 の汚れが DBT によって追跡され，ダーティマップに記録されてゆく．図の例では，2 番目と 4 番目が世代 2 によって汚されたことがわかる．

次に，$A \longrightarrow B$ へと転送が行われる．この時 B には差分転送に利用できる diff イメージはないので，通常の低速なブロックマイグレーションが行われ，さらに B の世代番号がインクリメントされる (図中 B)$A \longrightarrow B$ へのブロックマイグレーションが行われた後，A にある diff ディスクイメージは凍結される (図中 C) これはヘッダーの $freeze$ フラグを 1 にすることで行われる．$freeze$ フラグが 1 の diff ディスクイメージは一時的に VMM からは起動できないようになる．なぜ，この $freeze$ フラグが必要なのかは後述する．この時，A には $generation = 2$,$SEED = S_0$,$freeze = 1$ の diff ディスクイメージが残ることになる．B 上の VM イメージは，ライブマイグレーションにより VM が移譲され OS が起動するため，再び世代がインクリメントされる (図中 D) 結果的に $generation = 4$, $SEED = S_0$,$freeze = 0$ の diff ディスクイメージが作られることになる．B 上で作業を行うにつれて diff ディスクイメージのディスクブロックはどんどん汚れてゆく．図の場合では最終的に世代 4 によって 3 番目のディスクブロックが汚されていることが DBT により記録される (図中 D)

さらに，$B \longrightarrow C$ 間の転送がどのように行われるのかについて解説する．$A \longrightarrow B$ のときと同様に C にも差分転送に利用できる diff イメージはないので通常のブロックマイグレーションが行われ，さらに C の世代番号がインクリメントされる (図中 E) このマイグレーションが終わった後，A にあるイメージが凍結され，VMM から一時的に起動することができなくなる (図中 F) C 上の VM イメージは，ライブマイグレーションにより VM が移譲され，OS が起動することによって再び世代がインクリメントされる (図中 G) 結果的に，C 上には $generation = 6$,$SEED = S_0$, $freeze = 0$ の diff ディスクイメージが作れられることになる．C 上で作業が進むにつれ，ディスクが汚れてゆく．図の例では，世代 6 によって 5 番目のディスクブロックが DBT によって記録されていることがわかる (図中 G)

このように，計算機間でブロックマイグレーションが行われたり，OS の起動が起こるたびに世代番号 G がインクリメントされていき，それらの diff イメージファイルは共通 SEED 番号:S_0 を持っていることになる．また，DBT によって，どの世代でどのディスクブロックが汚されたかが記録されることになる．

[*1] qemu-img コマンドで diff ディスクイメージを作成したり，他のフォーマットから変換したときに付与される

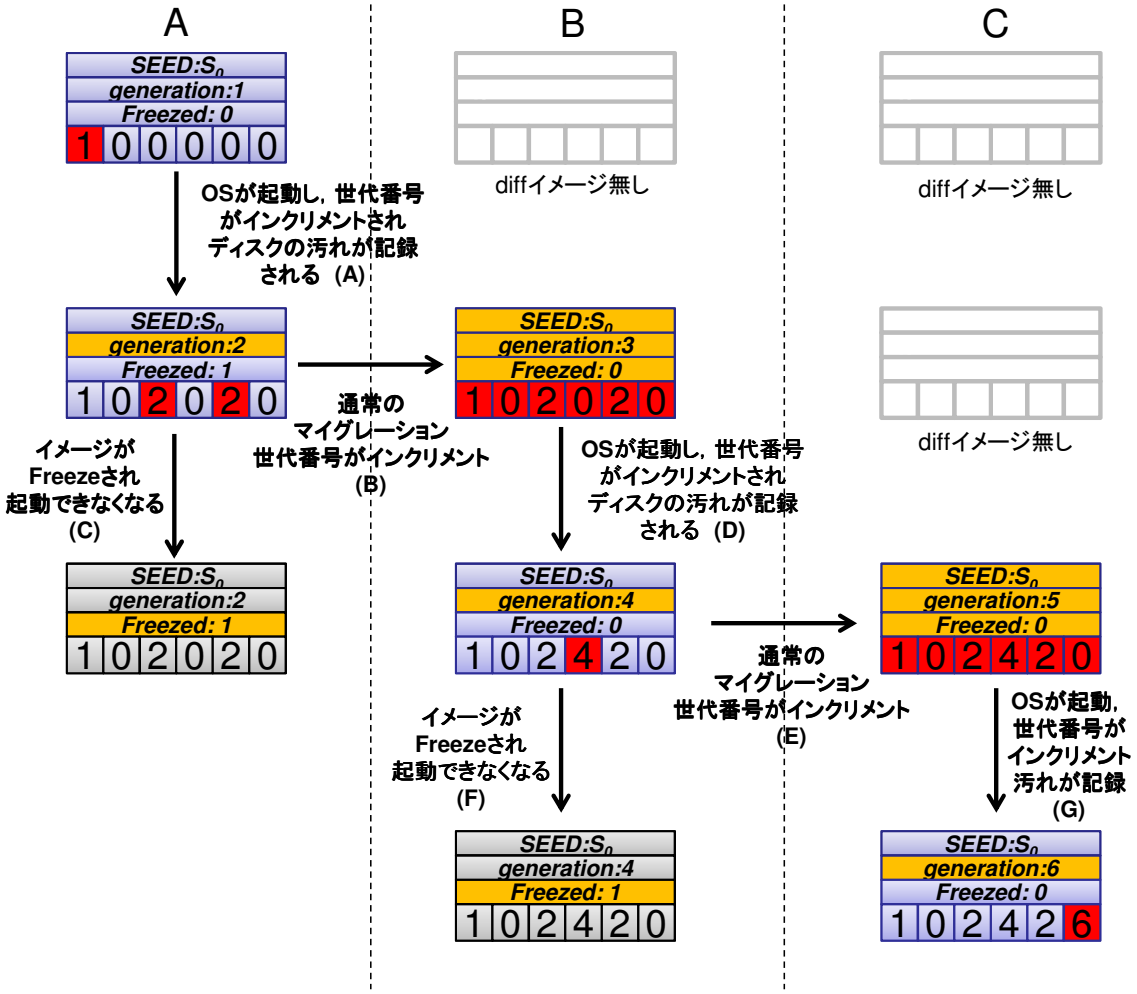


図 5.6. $A \longrightarrow B \longrightarrow C$ 間のマイグレーション

さらに，図 5.7 に $C \longrightarrow B \longrightarrow A$ 間のブロックマイグレーションについて解説する．

初めに，$C \longrightarrow B$ 間のブロックマイグレーションについて解説する．すでに，B 上には差分転送できる $generation = 4$ の diff ディスクイメージがあるので差分転送をつかった高速なブロックマイグレーションが可能である．まず初めに，C 上と B 上の diff ディスクイメージに記録されている SEED 番号:S_0 を見て，これが同一の SEED 番号である S_0 であることを確認する．仮に，S が異なる場合，C, B 間にはまったく異なるディスクイメージで，差分転送は不可能であると判定する．次に，ダーティマップを順に見てゆき，相手の世代番号 4 より大きい世代のブロックのみを転送する．そののち世代番号がインクリメントする (A)．今回の場合，5 番目の 6 ブロック目のみが世代番号が 6 となっており，相手の世代番号よりも大きくなっている．そのため，このブロックのみを転送する．これで，高速な差分転送が可能である．転送元は同じように $freeze = 1$ となり，このイメージから起動することが不可能となる (B)　次に転送された先では，OS が再び起動することで世代番号がインクリメントされ，世代番号は 8 となる．この状態でディスクの汚れが記録される．図の例では，3，4，5 番目のブロックが世代番号 8 によって汚されていることがわかる (C)

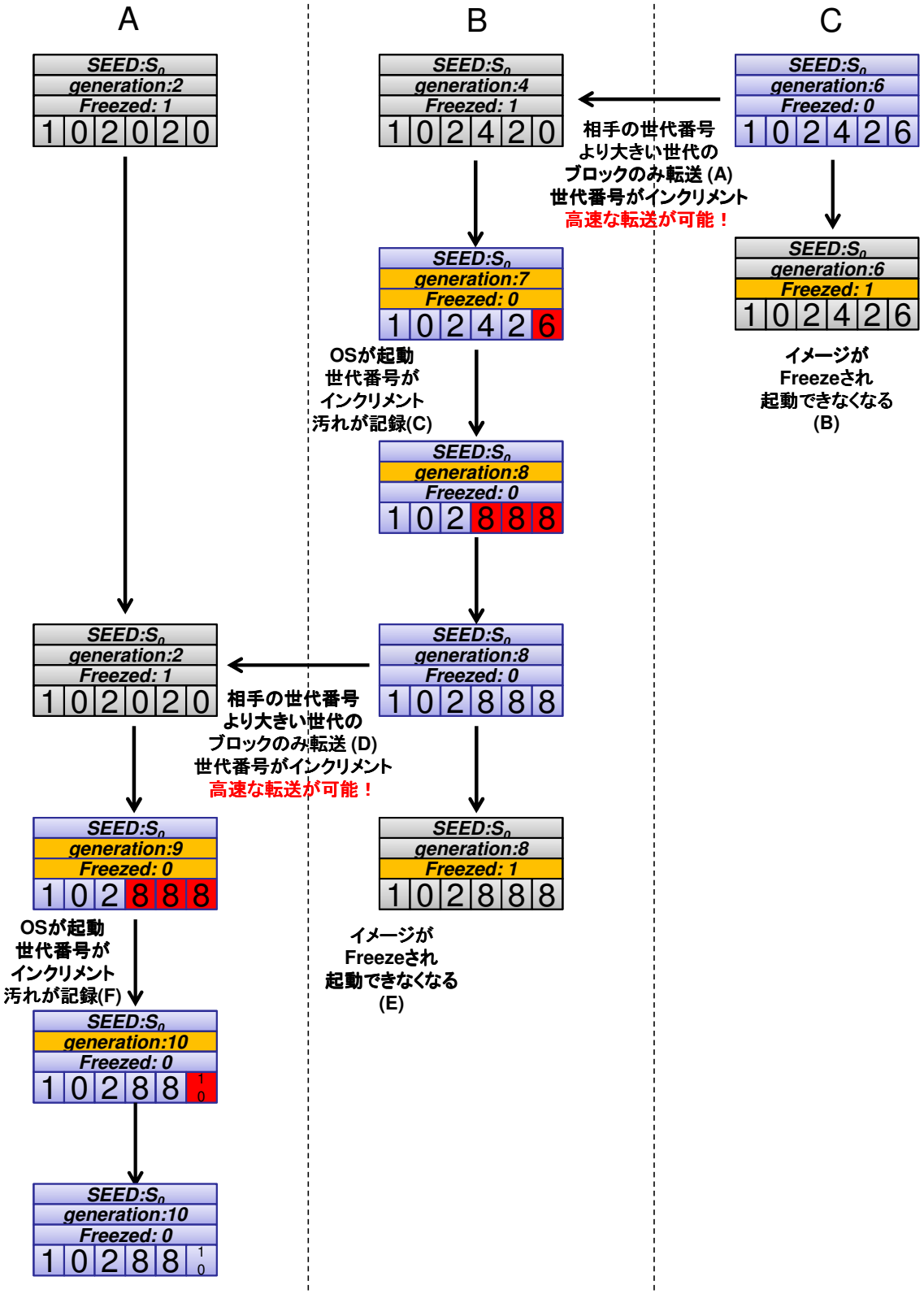


図 5.7. $C \longrightarrow B \longrightarrow A$ 間のマイグレーション

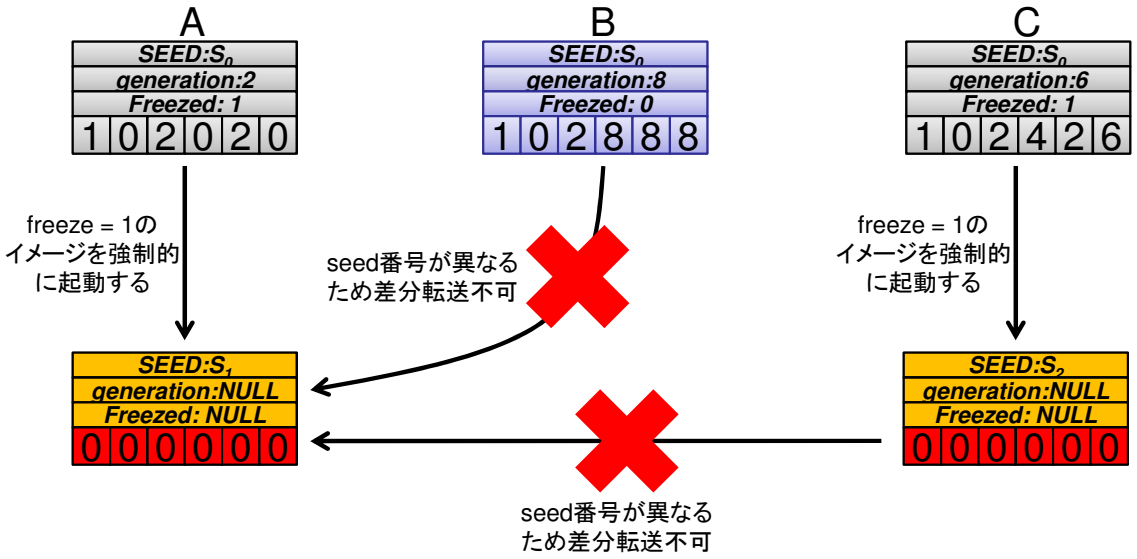


図 5.8. ディスクイメージの freezed を解除した場合の各仮想マシンの挙動

次に，$B \longrightarrow A$ の転送を考える．この場合も，ダーティマップを順に見てゆき，相手の世代番号よりも大きな世代番号を持つブロックのみを転送し，世代番号がインクリメントされる (D) 図の例では，世代番号 8 によって汚された，3，4，5 番目のブロックのみを転送することになる．同様に，転送元であった世代番号 8 のイメージは Freeze され，起動することができなくなる (E)．一方で，転送された VM は，OS が起動することで再び世代番号がインクリメントされたうえで，汚れが記録されることになる (F) 図の例では世代番号 10 によって，5 番目のブロックが汚されていることがわかる．

このようにして，今回提案するアルゴリズムでは過去にマイグレーションされた場所であれば，そこに存在する diff イメージとの差分を正確に割出，差分のみを転送し，高速なブロックマイグレーションが可能となる．

最後に，$freezed$ の役割について詳細に述べる．$freezed = 1$ となった diff ディスクイメージは基本的に起動不可能である．$freezed = 1$ のディスクイメージは基本的に，差分転送によるブロックマイグレーションを待機している状態であり，ブロックマイグレーションによる VM の移譲による起動以外の方法で起動することはできない．

もし，ユーザが強制的に $freezed = 1$ のディスクイメージを起動したときにはどのような現象が起こるのかを**図** 5.8 に示す．A, B, C の仮想マシン上で B のマシンが起動中であり，A, C の仮想マシン上のディスクイメージは $freezed = 1$ である．ユーザが，$freezed = 1$ のディスクイメージ上にある A を強制的に起動したとする．この時，VMM は A 上の仮想マシン上にあるディスクイメージに新たな SEED 番号である S_1 を割り振る．次に，$generation$ と Map も両方 NULL にクリアする．つまり，A 上の仮想マシン上にあるディスクイメージはまったく新しい別のディスクイメージとして生まれ変わることになる．双方のディスクイメージは SEED 番号が根本的に異なるため，起動中の B からの高速な差分転送は不可能となる．

5.3.4 議論

DBT による書き込み追跡の代わりに，転送時に `rsync`[62] と同様の手法である *Compare-by-hash* [63, 64] の手法を導入することが考えられる．つまり，転送先と転送元で，すべてのディスクブロックのハッシュ値 (SHA-1[65]，MD5[66] や Rabin-fingerprint[67] 等) を突合せ，ハッシュ値の異なるブロックをすべて転送するという手法をとれば DBT による書き込み追跡を行わずとも，差分転送が可能である．しかし，予備評価では 4GByte のディスクを 512KB 単位のブロックに区切って SHA-1 ハッシュ [65] をかけただけで 2 分近い時間が消費されてしまうことが判明したためこの手法は採用しない．

5.4 実装

われわれは，5.3 **節**で述べた diff フォーマットを Linux/KVM,QEMU に対して実装した．実装内容は以下に述べる 2 つの機能である．

1 つ目はダーティマップを作成するためのの DBT である．上記の二つのダーティマップは，仮想ディスク全体をある一定の大きさのブロック (セクタ) に区切り，各ブロック (セクタ) 1 つにつき 8 ビット[*2]としてビットマップを構成している．ちなみに，現在の実装では 2048 セクタを 1 ブロックとしている．なお，1 セクタあたりの容量は 512 バイトでありこれは QEMU のデフォルト値である．4Gbyte 程度の仮想ディスク容量であれば，4 キロバイト程度のビットマップに収まり，20Gbyte 程度の仮想ディスク容量であっても 20 キロバイト程度のビットマップに収まる．そのため，ビットマップが物理ハードディスク容量を圧迫することはない．

2 つ目が上記のダーティマップを QEMU のディスクイメージ階層とライブマイグレーション階層との間でやり取りするための API と，ライブマイグレーションが完了して，世代番号をインクリメントすることをディスクイメージに伝えるための 2 つの API が必要である．QEMU の実装はうまく構造化されている．例を挙げると，vmdk, qcow, qcow2 といったディスクイメージを操るディスクイメージ固有の処理を行う層は QEMU 内のディスクドライバとして書かれており，これらのドライバの開発者は QEMU 側から指定された API 群を記述していくことで新しい QEMU のファイルフォーマットを開発することができる．しかし，記録したダーティマップをディスクイメージ階層とライブマイグレーション階層とでやり取りする API や，世代番号をインクリメントするための API は存在しないため，これら二つの API の追加を行った．

5.5 評価

評価は 2 つの観点から行った．1 つ目の評価は DBT の書き込み追跡が VM の致命的なディスクの書き込み性能の低下を引き起こさないことを確認するためのものである．2 つ目の評価

[*2] これは自由に増減することができる

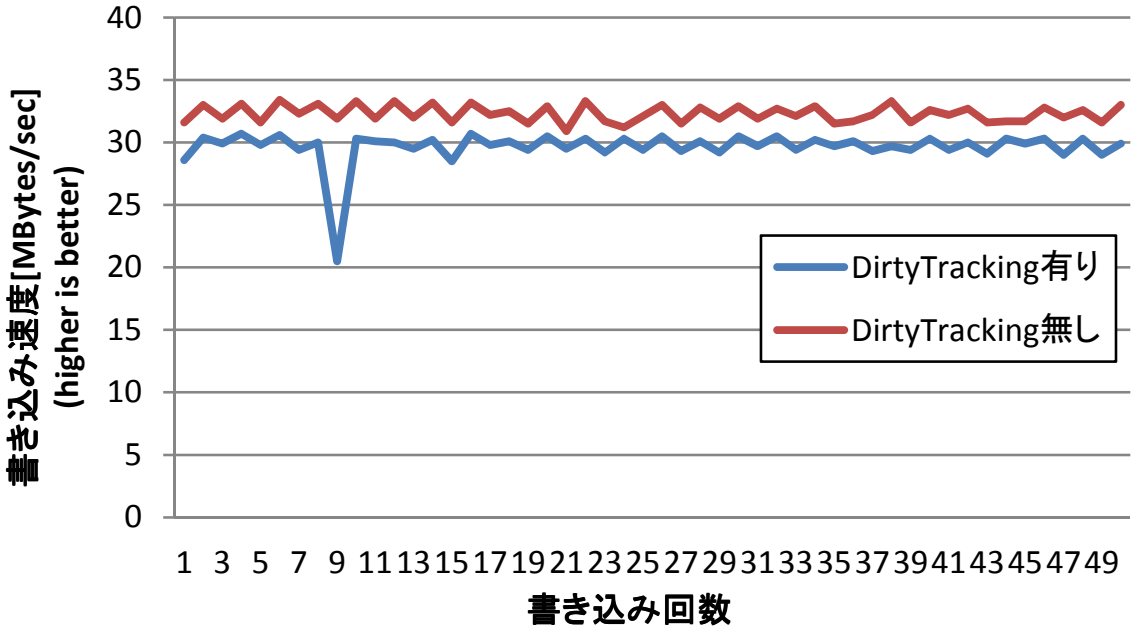


図 5.9. ダーティマップ更新時における dd に書き込み速度

は，さまざまなワークロードを VM を起動させ，ディスクブロック (1 ブロック 2048 セクタ) がどの程度汚されるのかを確認し，差分転送が，日常的なコンピューティングに使用されている VM に対して，程度効果を発揮するかについて評価する．

5.5.1　DirtyPage トラッキングの性能評価

本節では，DBT によるダーティマップを更新する処理が致命的なディスク書き込み性能の低下を引き起こさないことを確認する．評価環境には，Lenovo ThinkPad X220 ラップトップコンピュータを使用した．CPU は Intel Corei5-2430M@2.40GHz で，物理メモリは 4GB である．HDD は Seagate 製の 7200rpm HDD で 16MB のキャッシュメモリが搭載されている．ダーティマップの更新は書き込み処理時のみに行われるので，書き込み性能のみを評価した．dd コマンドを使い 1MB のブロックを 100 回書き込む評価を 50 回行った．評価結果を図 5.9 に示す．縦軸が書き込み速度 MBytes/sec で，横軸が書き込み回数である．トラッキング無しのときで平均 32.304MBytes/sec で，トラッキング有りの時が平均 29.656MBytes/sec であり，DBT による 8% 程度のディスク書き込み性能低下があることがわかる．

5.5.2　異なるワークロードによるブロックマイグレーションの速度

次に，実際に差分転送による効果の測定を行った．現実的なワークロードを考えて，以下に示す 4 つの作業を行った後で差分転送によるブロックマイグレーションを行った．パワーポイントファイルをダウンロードして Open Office による閲覧と編集と保存，夏目漱石の「こころ」(約 400KB のテキストファイル) を青空文庫よりダウンロードして閲覧，FireFox を使用した YouTube でのビデオ再生，約 3MB の PDF ファイルをダウンロードして閲覧．上記 4

表 5.1. Ubuntu 11.04 desktop (x86) 上でのブロックマイグレーションの測定結果

	差分転送なし	PDF	プレゼンテーション	YouTube	こころ
マイグレーション全体時間 (sec)	919.358	29.139	30.141	28.720	25.900
差分転送によるディスク転送量 (MBytes)	—	101	104	111	90

表 5.2. Windows 7 Professional (x86) によるブロックマイグレーションの測定結果

	差分転送なし	PDF	プレゼンテーション	YouTube	こころ
マイグレーション全体時間 (sec)	991.044	89.028	68.892	78.450	86.703
差分転送によるディスク転送量 (MBytes)	—	933	613	927	907

つのテストをユーザに 5 分間行ってもらった後，ライブマイグレーションを行った後で差分転送によるブロックマイグレーションを行い，通常のブロックマイグレーションと比べて，差分転送がどの程度有効であるかを確かめる測定を行った．

評価環境は，転送先が Lenovo の ThinkPad X60 ラップトップコンピュータで，CPU は Intel Core Duo CPU T2400 @ 1.83GHz で 2GBytes の物理メモリが搭載されている．転送元は DELL LATITUDE D630 ラップトップコンピュータで，CPU は Intel Core2Duo T7300 2.0GHz で，2GBytes の物理メモリが搭載されている．回線は 1Gbps の LAN 環境を使用した．

表 5.1 に Ubuntu 11.04 を使用したブロックマイグレーション結果を示す．一番時間がかかっているのはプレゼンテーションの 30.141 秒である．一方で，時間がもっともかかっていないのは「こころ」の 25.900 秒である．全体的に見て約 10 分間かかっていた転送が，差分転送の効果で数十秒に低下していることがわかる．差分転送によって転送されるディスクブロックも 20GB から数百 MB 内に縮小していることがわかる．

次に，**表** 5.2 に Windows 7 Professional を使用したブロックマイグレーション結果を示す．Ubuntu の時と比べると，Windows の場合，全体的に書き込まれているディスクブロックも多く，転送に比較的時間がかかっていることがわかる．一番時間がかかっているのは PDF 閲覧の 89.028 秒である．一方で，時間がもっともかかっていないのは，プレゼンテーションの 68.892 秒である．Windows (NTFS) の場合，Ubuntu (ex4) よりも多くのディスクブロックを汚すようである．また，Ubuntu のときはプレゼンテーションが一番時間がかかっていたのに対して，Windows の場合，PDF 閲覧が最も時間がかかる結果となった．また，Ubuntu の場合，もっとも時間がかかっていないのは，YouTube の 28.720 秒であったのに対して，Windows の場合，もっとも時間がかかっているのは PDF 閲覧であった．Windows の場合，数十秒に低下しているとは言えないものの，おおむね一分前後の転送時間であることがわかる．

5.6 関連研究

Linux/KVM 上のブロックマイグレーション以外にも，VM のディスクを含めてマイグレーションするための既存研究がいくつか存在する．Luo[68] や Bradford[69] らは，Xen 上で特殊なバックエンドデバイスドライバを実装することで，Linux/KVM のブロックマイグレーションとほぼ同等のことを実現している．つまり，共有ファイルシステムを使わずに VM ライブマイグレーションを達成している．しかし，転送先に過去のディスクイメージが存在していた

場合には差分転送のみを行うといったアプローチについては言及されておらず，本研究とは異なる．また，広渕ら [70] の提案では，Linux 上に独自のブロックデバイスドライバ/dev/nbd0 を実装することで，同様に共有ファイルシステムを使わない VM ライブマイグレーションを達成している．VM を起動するときローカルに存在する通常の VM イメージファイルの代わりに/dev/nbd0 を使うことで，ライブマイグレーション時には/dev/ndb0 が自動的に VM のディスクコピーも行う．われわれの研究との違いは，VM のライブマイグレーション先に過去のディスクブロックが存在していたとき，それらを転送しないことで高速化を図るというメカニズムを提案した点にある．

また，Sapuntzakis[25] らは，さまざまなユーザシナリオを想定した上で，仮想マシンのマイグレーションに関して，転送高速化に関するいくつかの提案を行っている．その中で，彼らはツリー構造を持つディスクイメージの管理手法を提案している．まず，ルートイメージと呼ばれる全ディスクイメージの元ファイルを作成する．ユーザは，このルートイメージをチェックアウトして，ルートイメージから見て子となるディスクイメージを作ることができる．さらに，ユーザが作成したディスクイメージから見てさらに子となる VM イメージも作ることができる．このようにして，ルートイメージ以外の，すべての VM イメージがそれぞれ親を持つというツリー構造が出来上がる．子は，ディスクに変更が加えられない限りは親へのポインタのみを持っており，実データは持たない構造になっている．マイグレーションのときは，親へのポインタのみを持つ子ノードファイルを転送することで転送量を減らす．転送後，必要なディスクブロックがあった場合，ネットワーク経由にて，オンデマンドで親をたどって必要なディスクブロックを見つけるという手法である．われわれの研究との違いは，彼らが，ネットワーク経由にて子ファイルを元にディスクブロックを後から要求できる状況を考えたうえで，ツリー構造を持つディスクイメージの管理手法を提案したのに対して，われわれは，初めからすべてのディスクイメージがマイグレーション元と先とで完全にコピーされる状況を考えた上で，ツリー構造よりもよりシンプルな差分記録による高速な転送手法を提案した点が異なる．

また，Intel Research から Internet Suspend/Resume (ISR) と呼ばれる手法に関するテクニカルレポートがいくつか出ている．ISR とは，VM を Suspend してから転送して，転送元で Resume することで，VM のコールドマイグレーションを実現する手法のことである．Kozuch[26] と Tolia[71] らは，Coda[72] と呼ばれる分散ファイルシステムを使って，ディスクイメージを含んだすべての VM のステートをすべて Coda 上にいったん格納し，転送先からネットワーク経由にてオンデマンドで VM のステートを要求することで ISR を実現している．Coda はファイルの先読み機能とキャッシング機構を備えているため，オンデマンドの転送以外にもバックグラウンドで VM のステート転送が行われ，やがてネットワークに接続せずとも Coda 経由で転送先から VM のステートを読みだすことができるようになる．われわれの研究との違いは，彼らが，ネットワーク経由にてディスクブロックを後から要求できる状況を考えたうえで VM のコールドマイグレーション手法を提案したのに対して，われわれは，初めからすべてのディスクイメージがマイグレーション元と先とで完全にコピーされる状況を考えた上で，よりシンプルな差分記録による高速な転送手法を提案した点が異なる．また，ISR であるため，ライブマイグレーションでないという点も異なる．

さらに，VMware 社の製品である VMware Storage VMotion[73, 74] が挙げられる．これ

は，ストレージを含む VM ライブマイグレーションを行うための機構である．われわれとの研究の違いは，配置先にすでに再利用可能なディスクイメージがあった場合，それを利用して高速な VM 転送を行う機構について提案している点である．

最後に，穐山ら [75, 76] の提案を挙げる．これは，われわれの仮定と同様に，いったん転送 (VM ライブマイグレーション) した VM は再び転送元へと戻ってくる可能性が高い．というユーザシナリオの基で，VM ステートの RAM 情報に着目し，VM の RAM 情報を転送元で保存しておいて，再び転送元へと VM が戻ってくるときは，汚れた DirtyPage のみを転送することで VM の転送速度の高速化を図っている．本研究との違いは，われわれは，これを VM のディスクストレージで行った点が異なっている．

5.7　差分転送を用いた高速な VM マイグレーションのまとめ

3 章にて，専用ソフトウェアやアプリアンスをインストールするための技術的な知識を持たないユーザでも導入が容易なシンクライアントシステムについて利用可能性を向上した．

また，4 章では，異種 HostOS（Windows と Linux）間で動作する VM ライブマイグレーションを実現するために，Linux/KVM を Windows に移植し，これを WinKVM と名付けた．これで，コンシューマ用途として広く普及する Windows と，サーバ/サービス用途として広く普及する Linux 間での VM ライブマイグレーションが可能となり，VM ライブマイグレーションの利用可能性が向上した．

上記の二つの研究に加えて，VM を他の物理マシンに完全転送する技術があれば，さらに利便性の高い仮想化インフラを構築することが可能であると論じた．例えば，ユーザは，普段は耐故障性が高いクラウド上にて動作する VM を，シンクライアント VoXY を通して利用している．しかし，シンクライアントでは利用が難しいソフトウェア（CAD，CAM，3D ゲーム）や，出張で一時的にネットワークが利用出来ない環境でも自分の環境（VM）を持ち出したい場合には，VoXY 上で操作している VM を，スムーズに手元の計算機に VM をダウンロードすることが出来ればより便利であると考える．

VM ストレージマイグレーションと呼ばれる技術を使えば VM のストレージを含めて転送することができるため，この技術を利用することが可能である．しかし，この新たな VM ストレージマイグレーションのメカニズムには転送に多くの時間や，広帯域なネットワーク環境要求されるという問題点が存在する．そのため，限定された環境下でしか使うことが出来ず，利用可能性が低い．VM のディスクイメージは一般的に巨大なファイルであり，数十 GByte になることも珍しくない．例えば，1Gbps の LAN 環境にて 20GB のディスクイメージを持つ VM をマイグレーションすると，VM ストレージの転送だけでも，数十分もの時間がかかる．マイグレーションが完了するまでに 10 分以上もの時間がかかるのはユーザの利便性を低下させる．

この問題点を解決するため，本章では，データセンタ間や，個人用 VM ライブマイグレーションでは，特定の物理マシンの間，VM が行ったり来たりする，VM の往来と呼ばれる挙動に着目し，シンプルな差分転送の手法を提案した．VMM から VM の I/O をフックする DBT (Dirty Block Tracking）と，VM のイメージに対して世代番号と呼ばれる概念を付加することにより，VM が，いつ（どの物理マシンで)，どこが（VM ストレージの何処の部分が）変

更されたかを検出し，転送の差異には，差分のみを正確に検出し，高速に転送する．日常的なオフィスワーク程度のコンピューティングであれば，コンピュータがディスクに与える影響は数 % であるため，この手法は VM ストレージマイグレーションの高速化に寄与することを確認した．ベンチマークの結果，日常的なコンピューティングに使用されている VM であれば，最大で 97% の高速化を達成することに成功した．

第 6 章

結論

本博士論文では，HTML+JavaScript と HTTP のみで動作するシンクライアントシステムである VoXY : Vnc over proXY と，異なる OS 間で VM ライブマイグレーションを可能とする WinKVM を実現するための，Windows 上で Linux カーネルドライバを動作させるための「簡単」かつ「性能劣化のない」移植手法を提案した．さらに，エミュレーションレイヤを構築して Linux/KVM ドライバを動作させることで，移植性の高い仮想マシンモニタの設計手法の提言も行なった．そして，VM ディスクイメージごと転送する技術である VM ストレージマイグレーションの高速化という 3 つの研究を通して，より多くの人が利用可能な遠隔仮想計算機転送技術について論じた．

6.1 本博士論文で提案した成果

現在の遠隔仮想計算機転送システムである VM ライブマイグレーションとシンクライアントシステムは，二つの意味で利用可能性が低い．第一に現在の遠隔仮想計算機転送システムは「使う人を選ぶ」専用のアプライアンスやソフトウェアが必要であったり，専門知識が必要であったりと万人向けではない．第二に「使う環境が限定されている」プロトコルの制約等で，いつでもどこでも直ちに使えるわけではないし，計算機リソースのダウンロードに時間がかかったり，ユーザが計算機をリソースを利用できるようになるまで時間かかったりと，利用可能性を低下させているのである．

そこで，本博士論文では**現在の限定的である VM とユーザと結びつける遠隔計算機転送システムを，より多くの利用シーンでも利用できるような，遠隔仮想計算機転送システムを提案した**．

既存の VM ライブマイグレーション技術が利用可能性を低下させている原因は二つある．

一つ目の原因は「コンシューマ向け OS とサーバ OS を結びつける VMM が存在しない」点にあった．既存の VMM では，複数 OS 間で VM ライブマイグレーションを達成することができず，利用環境が同一 OS 上の VMM と限定されている．特定の OS 上の間でしか VM ライブマイグレーションを達成できないという状態は利用可能性を低めている．

また，二つ目の原因は，VM のディスクイメージを共有するために，VM の転送先と転送元が常にネットワークで接続されている必要がある点にある．これは，持続的なネットワーク接

続に依存していると言い換えることもできる．VM ライブマイグレーションを利用して VM を転送するためには，cifs，NFS や iSCSI といった共有ファイルシステムを設定/構築した上で，VM のディスクイメージが転送先と転送元で共有されている必要がある．共有ファイルシステムを利用するためには，持続性のあるネットワーク環境が必要であり，いつでもどこでも VM が使えるわけではない．

共有/分散ファイルシステムに依存せずとも VM ライブマイグレーションを行える VM ストレージマイグレーションと呼ばれる技術が開発されている．これは，ネットワークが接続されている時に，VM のディスクイメージごとすべての技術を転送する技術である．しかし，この VM ストレージマイグレーションもまだ利用可能性が低い．なぜなら，VM のディスクイメージは一般的に巨大なファイルサイズであるため「転送に時間がかかる」のである．高速なネットワーク環境であったとしても，数十 GB ものディスクイメージを転送するのにはある程度の時間がかかり，また，帯域も圧迫する．そのため，利用可能性が高いとはいえなかった．

このように，現存する遠隔仮想計算機転送システムであるシンクライアントシステムと VM ライブマイグレーションとの利用可能性は低い．そこで，本研究では「JavaScript と HTML を用いたシンクライアント VoXY」「Linux ドライバの Windows 移植による WinKVM の実装」「差分転送を用いた高速な VM マイグレーション」という 3 つの研究で，**「使う環境が限定されており，使う人を選ぶ」遠隔仮想計算機転送技術を「より多くの環境で使え，より多くの人が使える」遠隔仮想計算機転送技術に改良することを目的とし，それを達成した**．

6.1.1　JavaScript と HTML を用いたシンクライアント VoXY で解決したこと

VoXY は Web ブラウザ上で動作するエンドユーザにとって導入が容易なシンクライアントシステムである．

VoXY の開発を始めた 2007 年頃には，クラウドコンピューティングの普及もあり，Web ブラウザ上ですべてのコンピューティングを行おうという潮流が存在した．この時代の流れは，シンクライアントシステムにも影響を与え，Web ブラウザ上で動作するシンクライアントシステムも開発されたことは上述した通りである．

しかし，既存のシンクライアントシステムが採用しているプロトコルをそのまま利用し，JavaScript + HTML で Web ブラウザ上にてインタラクティブ性の高いシンクライアントシステムを実現することは不可能であった．VoXY の開発を始めた当初，Web ブラウザの画面描画インタフェイスは貧弱であったため，従来の，インクリメンタルに送られてくるフレームバッファの差分を高速に Web ブラウザに表示し，インタラクティブ性の高いシンクライアントシステムを Web ブラウザ上で再現することが困難であった．

そこで, VM の画面を縦:200 pixel, 横:200 pixel (200×200) のタイル状に分割し, JavaScript の DOM 操作による HTML IMG タグの更新という操作を行うことで，ゲームや動画編集ソフトウェア，CAD，CAM と言った極めて高いインタラクティブ性を要求するソフトウェア以外であれば，十分に使用に耐える，高いインタラクティブ性を持つシンクライアントシステムを Web ブラウザ上に実装することに成功した．変更部分のみをインクリメンタルに転送するシステムに比べると使用帯域は増加するものの，貧弱な画像転送システムのみしか持たない Web ブラウザでも，十分にインタラクティブ性能があるシンクライアントシステムを実現で

きることを証明した.

VoXY には，既存のシンクライアントシステムとは異なる 2 つの利点が存在する．1 つ目の利点は，VoXY はクライアントに Web ブラウザを使用可能であるためユーザにとって導入が容易である．なぜなら，多数の OS インストールパッケージには，Web ブラウザが初期導入されている．そのため，ユーザは新たにソフトウェアをインストールする必要がない．2 つ目の利点としては，VoXY は，Web ブラウザの標準プロトコルである HTTP の通信だけで高いインタラクティブ性を持つシンクライアントを実現することができる点である．そのため，VoXY は，HTTP 以外の通信を拒絶するよう高い制限が存在し，従来のシンクライアントならば VPN (Virtual Private Network) やプロキシソフトウェアをインストールしなければならない環境でも，Web ブラウザだけで使用することが可能である．これらの利点により，VoXY はユーザにとって導入が容易であり，使いやすいものである．そのため，「使う人を選ばず」「使う環境が限定されていない」遠隔仮想計算機システムを部分的に実現することに成功した.

6.1.2 Linux ドライバの Windows 移植による WinKVM の実装

WinKVM は，Linux カーネルに付属する VMM (KVM: Kernel-based Virtual Machine) を Windows に移植したものである.

KVM は Linux デバイスドライバとして実装されているため，WinKVM を実現するためには，この Linux デバイスドライバを Windows 上で動作させる必要がある．しかし，これは Linux に限った話ではないが，オペレーティング・システムのデバイスドライバは，アプリケーションプログラムと比べると，OS に強く依存するプログラムある．そのため，一般論としてドライバの移植というものは困難である．そのため，プログラマがデバイスドライバのコードを読み込み，コードを移植してゆくという作業が必要になり，これには大きな人的なコストが掛かる.

本研究では，Linux デバイスドライバである KVM を Windows 上で動作させるため，Windows カーネル上に Linux カーネルを模倣するエミュレーションレイヤを構築して，Linux ドライバを Windows 上で動作させることを可能とした.

Linux オペレーティングシステムと Windows オペレーティングシステムは内部のアーキテクチャがかなり異なるものの，ほとんどの Linux カーネルの機能を，ラッパーライブラリを通して，Windows のカーネルの機能に変換することができ，さらに，ほとんどの性能劣化を引き起こすことなく Linux/KVM ドライバを Windows 上で動作させることができることを証明した.

しかし，完全に非修正の Linux / KVM ドライバを Windows 上で動作させることは困難であり．性能劣化を引き起こすことなくエミュレーションを行うためには，一部，数１０行程度のドライバコードの書き換えは必要である．これは本手法の限界である.

まず，Linux と Windows とでは，カーネル空間とユーザ空間のメモリ空間を結びつけるメモリマッピングと呼ばれる機構の性質が異なる．Linux の場合ページフォルトハンドラである `fault / nopage` ハンドラをプログラマが制御することでメモリマッピングを行うことができる．Windows ではページフォルトハンドラをプログラマが制御することができないため，

Linux ドライバがこの `fault / nopage` ハンドラが使用されていた場合，本エミュレーションレイヤでは，Windows カーネル上でこれを高速にエミュレーションすることができない．Linux / KVM ドライバはこの `fault / nopage` ハンドラを使用しているため，この点に関しては，Linux / KVM ドライバを修正することを余儀なくされた．また，一部の KVM が利用する一部の特権命令も，そのままでは Windows 上で動作させることが出来なかった．そのため，一時的に，トランポリンコードと呼ばれる，ゲスト OS のからホスト OS への復帰ポイントに代替となるコードを用意することで，これらの特権命令も高速に Windows 上でエミュレーションすることを可能とした．しかし，繰り返しになるが，一部の KVM コードを書き換えるという作業が必要であるため，これは本手法の限界である．

しかし，この貢献で，Linux と Windows 間で VM ライブマイグレーションが実現できるようになり「使う人を選ばず」「使う環境が限定されていない」という限定的な遠隔仮想計算機転送技術である VM ライブマイグレーションという技術を改良することに成功した．

さらに，WinKVM の実証実験を通して得られた知見は移植性の高い VMM を設計する上で重要な指標となる．利用可能性の高い仮想計算機転送技術を実現するためには，様々なオペレーティングシステム上で動作する Hybrid VMM の実現が重要であることは既に述べた．そのため，Windows と Linux といった，まったく異なる設計思想を持つカーネル上にて同一の VMM を動作し，異なる OS 上で動作する同一の VMM 上で VM ライブマイグレーションを行うという今回の実証実験を通して，マルチ OS 対応の Hybrid VMM を設計する際に留意すべき点についてまとめることもできた．

6.1.3　差分転送を用いた高速な VM マイグレーション

差分転送による VM ストレージマイグレーション技術に関する提案では，今日の VM の運用形態では特定の VM が特定の物理マシンをライブマイグレーションで往来する可能性が高いという点に着目した VM の差分転送の手法を提案した．仮想マシンモニタに DBT (Dirty Block Tracking) と呼ばれる機構を取り付け，さらに，VM ディスクストレージ（ディスクイメージ）に「世代番号」と呼ばれる番号を付加した．DBT と世代番号の作用によって，転送先と転送元で変更のあった部分のみを高速に，かつ，正確に検出し，転送することを可能にした．

VM ライブマイグレーションの転送時間でもっとも支配的な要素は VM の仮想ディスクイメージの転送にかかる時間である．そのため，DBT による差分の検出と転送により，高速な VM 転送を実現し「使う環境が限定されている」VM ストレージマイグレーションの技術をより多くの環境でも使えるように改良した．

ベンチマークの結果，日常的なオフィスワーク程度のコンピューティングであれば，コンピュータがディスクに与える影響は数 % であることが調査で判明した．そのため，この手法は VM ストレージマイグレーションの高速化に寄与し，90% 以上の転送時間削減を達成した．既存の VM ストレージマイグレーションと比べると高速に VM が転送が可能である．

6.2 将来の展望

これからの展望について述べる．本博士論文では隔仮想計算機転送システムの利用可能性を上げることが目的であった．そのため，ユーザの VM が中央サーバで動作している時に，VMM が積極的に VM に対して監視と動作制限を行うことでユーザのコンピュータへのメンテナンスを更に容易にするシステムを考えていく必要があると考えている．これは，高度な知識のないユーザでも安心して仮想マシン上の VM を使えるようにするたに必要である．

具体的には，VMM 側から VM の挙動を監視し，ユーザが意図しない，不正なプログラムが実行されようとしている時に，その実行を制限したり，ユーザに警告を発したりする仕組みが必要であると考える．すでに，VMM 側から VM の動作を監視して何らかのアサーションをユーザに対して発する研究は多く行われているが，クラウドコンピューティングのように，多数の VM が一つのサーバに集約し，それをすべて同時に監視し，アサーションを入れるようなシステムはまだ未発達の分野である．そのため，大量の VM の挙動を監視して，マルウェアの発見や，ポリシーにない不正な動作をしている VM を発見し，自動的に，ユーザに警告や，必要であれば回復処置を施すシステムの研究が必要である．

6.3 まとめ

本研究では 3 章で述べた「JavaScript と HTML を用いたシンクライアント VoXY」4 章で述べた「Linux ドライバの Windows 移植による WinKVM の実装」そして，5 章で述べた「差分転送を用いた高速な VM マイグレーション」の 3 つの研究にて「使う環境が限定されており，使う人を選ぶ」遠隔仮想計算機転送技術を「より多くの環境で使え，より多くの人が使える」遠隔仮想計算機転送技術を達成した．

シンクライアントシステムの利用可能性を低減させている原因である「導入に専門知識が必要」であり利用可能性が限定的という問題に対し，解決手法として，Web ブラウザ上でシンクライアントシステムを動作させるという手法に着目した．しかし，既存の Web ブラウザ上で動作させることができるシンクライアントシステムは Web ブラウザとは別に，特別なプラグインシステムを導入する必要があるため，「導入 が面倒」「特殊なプロトコルを利用し，使える環境が限定的」という問題点を完全に解決するには至っていない．そこで，われわれは，従来のシンクライアントプロトコルである変更点のみを転送するプロトコルを改良し，画面をある程度の大きさのタイル上に区切り，タイルの転送を行うプロトコルを提案した．この手法は．従来のシンクライアントプロトコルに比べると，要求帯域は大きくなるものの，HTML+JavaScript のみでもインタラクティブ性の高い Web ブラウザを実現することが可能になる．オフィスワーク系のソフトウェアであれば十分使用に耐えることができる．VoXY のクライアントとなる Web ブラウザはほとんどすべてのコンピュータにプリインストールされており，特別な整備無しで利用することが可能である．当然，HTTP という一般的なプロトコルですべての通信を行うため，あらゆる環境で使用することができる．

次に，コンシューマ向け OS とサーバ OS 間で VM ライブマイグレーションを可能な VMM を開発することで，より便利な仮想化インフラを実現することが可能であると考えた．これが

実現すれば，シンクライアント VoXY では対応するのが難しいコンピュータゲーム，CAD，CAM と言ったアプリケーションや，作業環境である VM を出張等でネットワークが断絶する環境に一時的にダウンロードして手元の計算機環境で運用したい時に，VM の実行コンテキストを維持したまま，ダウンロードすることが可能となる．そこで，Linux カーネルに実装された KVM と呼ばれる VMM を Windows に移植することで，Windows と Linux の HostOS 間で VM ライブマイグレーションの実現を目指した．Linux / KVM は Linux のデバイスドライバとして実装されている．デバイスドライバは対象の OS に強く依存するため，移植には人的なコストが掛かる．そこで，Linux / KVM デバイスドライバを Windows 上で動作させるエミュレーションレイヤを構築することで，Linux / KVM ドライバを Windows 上で動作させ，結果的に KVM を Windows 上で動作させることに成功した．エミュレーションレイヤを使い，特定のアーキテクチャのみで動作するように実装されたプログラムを，他のアーキテクチャでも動作させるようにするという手法はありふれたものである．しかし，この手法で最も重要視すべき点は「性能劣化を可能な限り防ぐ」という点である．本研究では，Linux と Windows の持つ性質の違いを分析し，Windows 上で，非修正の Linux デバイスドライバを効率よく動作させる手法を提案した．結果的に，完全に非修正の Linux / KVM ドライバを Windows 上で動作させることは出来なかったが，修正は数１０行程度にており，修正に掛かる人的コストは低く，さらに，性能劣化を引き起こすことなく Linux / KVM ドライバを Windows 上で動作させることに成功した．

最後に，VM ストレージマイグレーションの利用可能性を低下させている原因である「VM のストレージの転送に時間がかかる」という問題に対しては，今日の VM の運用形態では特定の VM が特定の物理マシンをライブマイグレーションで往来する可能性が高いという点に着目した VM の差分転送による高速転送を提案することでこれを解決した．これで，相手先にすでに転送済みの VM ディスクページが存在していれば，VM ディスクイメージのすべてを転送せずとも，転送先と転送元で変更のあった部分だけを検出して転送できる．オフィスワーク程度のコンピューティングであれば，ディスクイメージも大きく書き換わることはなく，差分転送のアプローチは有効であることが分かり，90% 以上の転送速度を削減することに成功した．

以上，**現在の限定的である VM とユーザと結びつける遠隔計算機転送システムを，より多くの利用シーンでも利用できるような，遠隔仮想計算機転送システムを提案する**を達成し「使う人を選ぶ」かつ「使う環境を選ぶ」遠隔仮想計算機の利用可能性を向上させたことが本博士論文の貢献である．

発表文献と研究活動

博士論文を構成する主要な査読付き論文

(1) 高橋一志, 笹田耕一: WinKVM : Hybrid Hypervisor 移植の一例　情報処理学会論文誌コンピューティングシステム (ACS36) Vol.5 No.1, pp.27-40 (2012.1).

(2) 高橋 一志, 笹田耕一, 竹内 郁雄:　VMM で WebVDI を実現するシステムの実装と評価, 情報処理学会論文誌コンピューティングシステム (ACS) Vol.2, No.1, pp.53-63 (2009.3)

(3) Kazushi Takahashi, Koichi Sasada and Takahiro Hirofuchi: “A Fast Virtual Machine Storage Migration Technique Using Data Deduplication” , The Third International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING 2012)

その他の論文

(4) Ruo Ando, Kazushi Takahashi, and Kuniyasu Suzaki: “Inter-domain Communication Protocol for Real-time File Access Monitor of Virtual Machine” , Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), Vol. 3, No. 1/2, pp. 120-137, March 2012

(5) Ruo Ando, Kazushi Takahashi, Kuniyasu Suzaki: ”A SOM based malware visualization system using resource access filter of virtual machine”, The 2011 International Conference on Computers, Communications, Control and Automation (CCCA 2011), Hong Kong, China, February 20-21, 2011

(6) 高橋一志, 笹田耕一: 差分転送を用いた高速な VM ディスクイメージ転送手法の提案, 情報処理学会コンピュータシステム・シンポジウム論文集 (ComSys 2011)

(7) 高橋一志, 笹田耕一 : WinKVM: 異なるホスト OS 間の VM ライブマイグレーション実現に向けて, 情報処理学会コンピュータシステム・シンポジウム論文集 (ComSys 2009), Vol.2009, No.13, pp. 59–66 (2009-11)

(8) 高橋一志, 笹田耕一, 竹内郁雄: VMM で WebVDI を実現するシステムの実装と評価, 情報処理学会コンピュータシステム・シンポジウム論文集 (ComSys 2008), Vol.2008, No.20, pp. 89-98 (2008-11)

(9) Kazushi Takahashi and Koichi Sasada: “WinKVM: Windows kernel-based virtual

machine” (KVM Forum’ 10) (2010)
(10) 高橋 一志, 笹田 耕一, 千葉滋: CAS ベースの VM ディスクイメージ 日本ソフトウェア科学会 (JSSST) 第 29 回大会
(11) 高橋 一志，笹田 耕一: 世代の概念を考慮したディスク書き込み追跡による高速な VM 転送機構の提案 第 53 回プログラミング・シンポジウム予稿集, pp. 37–45 (2012.1).
(12) 高橋一志, 笹田耕一: KVM を利用した異種ホスト OS 間上でのライブマイグレーション, 情報処理学会「システムソフトウェアとオペレーティング・システム」SWoPP 2010 (2010-8)
(13) 高橋一志: VoXY: Web ブラウザ上での仮想計算機システム, 先進的基盤ソフトウェア Vol.2, pp. 143-158 (2008-11)
(14) 高橋一志，笹田耕一，竹内郁雄: WebVDI を実現するための VNC Proxy, 情報処理学会研究報告 2008-OS-109, Vol.2008, No.77, pp. 85–92, (2008-8)
(15) 安藤類央, 高橋一志, 須崎有康 : emit 命令を用いた XEN 仮想マシン上のセキュリティインシデントの可観測化と可視化 (情報処理学会 コンピュータセキュリティ研究会 第 52 回研究報告) 2011 年 3 月
(16) 安藤 類央, 高橋 一志, 田端利宏, 須崎有康 : 統合仮想化システムモニタを用いたマルウェアのプロファイリング, 情報処理学会 コンピュータセキュリティシンポジウム 2009 年 10 月
(17) 須永高浩，上野康平，高橋一志，笹田耕一: Web ブラウザで動作するシンクライアント VoXY の実用化 SACSIS 2011（ポスター発表）2011 年 5 月
(18) 高橋一志，笹田耕一 : サーバ OS とクライアント OS 間の VM ライブマイグレーション, 情報処理学会「システムソフトウェアとオペレーティング・システム」ComSys 2010 (ポスター発表)
(19) Kazushi Takahashi and Koichi Sasada: “WinKVM: Virtual Machine Module for Windows”(Poster), Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’09), (2009.03).
(20) 高橋一志，山本知仁 : リッチクライアント技術を用いた仮想マシンモニタの提案と実装, 先進的計算基盤システムシンポジウム SACSIS 2007 (ポスター)

受賞等

(21) 2008 年　情報処理学会 第 109 回 OS 研究会　学生発表賞
(22) 2009 年　情報処理学会 コンピュータサイエンス領域奨励賞
(23) 2010 年　情報処理学会 ComSys2010 研究奨励賞

参考文献

[1] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of Cloud Computing and Its Applications*, October 2008.

[2] Amazon.com, Inc. Amazon EC2. `http://aws.amazon.com/ec2/` 2009.

[3] VMware, Inc. VMware View. `http://www.vmware.com/jp/products/view/` 2009.

[4] Citrix Systems, Inc. XenDesktop. `http://www.citrix.co.jp/products/xendesktop/` 2009.

[5] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, Vol. 38, No. 5, pp. 39–47, May 2005.

[6] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, Vol. 7, No. 6, pp. 34–45, 1974.

[7] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM J. Res. Dev.*, Vol. 25, No. 5, pp. 483–490, September 1981.

[8] The user-mode linux kernel home page. `http://user-mode-linux.sourceforge.net/` 2006.

[9] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, Vol. 36, No. SI, pp. 195–209, December 2002.

[10] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, Vol. 15, No. 4, pp. 412–447, November 1997.

[11] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Commun. ACM*, Vol. 36, No. 2, pp. 69–81, February 1993.

[12] WebShaka,Inc. Youos. `https://www.youos.com/` 2006.

[13] Tristan Richardson. The rfb protocol. `http://www.realvnc.com/docs/rfbproto.pdf` 18 June 2007.

[14] D. Tougaw and J. Sanders. Sunray: a cost-effective desktop computer solution. *Computing in Science & Engineering*, Vol. 4, No. 1, pp. 15–17, 2002.

[15] Brian K. Schimdt, Monica S. Lam, and J. Duane Northcut. The interactive perfor-

mance of slim: a stateless, thin-client architecture. *SIGOPS Oper. Syst. Rev.*, Vol. 34, No. 2, pp. 12–13, 2000.

[16] Microsoft Corporation, Redmond, WA. *Comparing MS Windows NT Server 4.0, Terminal Server Edition, and UNIX Application Deployment Solutions,*, 1999. Technical White Paper.

[17] Adrian Nye, editor. *X Protocol Reference Manual: Volume Zero for Xii, Release 6.* Oreilly & Associates Inc, 1994.

[18] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. Thinc: a virtual display architecture for thin-client computing. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pp. 277–290, New York, NY, USA, 2005. ACM.

[19] Joeng Kim, Ricardo A. Baratto, and Jason Nieh. pthinc: a thin-client architecture for mobile wireless web. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pp. 143–152, New York, NY, USA, 2006. ACM.

[20] Michael L. Powell and Barton P. Miller. Process migration in demos/mp. *SIGOPS Oper. Syst. Rev.*, Vol. 17, No. 5, pp. 110–119, October 1983.

[21] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the v-system. *SIGOPS Oper. Syst. Rev.*, Vol. 19, No. 5, pp. 2–12, December 1985.

[22] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, Vol. 6, No. 1, pp. 109–133, February 1988.

[23] Fred Douglis and John Ousterhout. Mobility. chapter Transparent process migration: design alternatives and the sprite implementation, pp. 56–86. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[24] Amnon Barak and Oren La'adan. The mosix multicomputer operating system for high performance cluster computing. *Future Gener. Comput. Syst.*, Vol. 13, No. 4-5, pp. 361–372, March 1998.

[25] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, Vol. 36, pp. 377–390, December 2002.

[26] Michael Kozuch, Mahadev Satyanarayanan, Thomas Bressoud, and Yan Ke. Efficient state transfer for internet suspend/resume. *Intellectual Property*, May 2002.

[27] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Grible. Constructing services with interposable virtual hardware. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pp. 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[28] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper.*

Syst. Rev., Vol. 36, No. SI, pp. 361–376, December 2002.

[29] Koichi Onoue. A virtual machine migration system based on a cpu emulator.

[30] VMware, Inc. VMware. `http://www.vmware.com/` 2009.

[31] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 164–177, New York, NY, USA, 2003. ACM.

[32] Boca Research, Inc, Boca Raton, FL. *Citrix ICA Technology Brief*, 1999. Technical White Paper.

[33] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. Thinc: a virtual display architecture for thin-client computing. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pp. 277–290, New York, NY, USA, 2005. ACM.

[34] Michael Smith, W3C. Html 5 publication notes. `http://www.w3.org/TR/2008/NOTE-html5-pubnotes-20080610/` June 2008.

[35] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, p. 41, Berkeley, CA, USA, 2005. USENIX Association.

[36] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pp. 225–230, 2007.

[37] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition (MIT Electrical Engineering and Computer Science)*. The MIT Press, 2 edition, July 1996.

[38] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction. (3rd ed.)*. Addison-Wesley, MA, 1998.

[39] FreeOsZoo-Project. Free live os zoo. `http://www.oszoo.org/wiki/index.php/Free_Live_OS_Zoo` 2006.

[40] JPCTeam. The jpc project. `http://www-jpc.physics.ox.ac.uk/index.html` 2007.

[41] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, Vol. 38, pp. 48–56, May 2005.

[42] AMD. Secure virtual machine architecture reference manual. Technical report, Advanced Micro Devices, May 2005.

[43] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[44] D. K. Fisher and K.M.Tran. Comparison of ieee portable operating system interface (posix) - part i and x/open single unix specifications (sus), 1995.

[45] ISO. *ISO/IEC 9945-1:1990 Information technology textemdash Portable Operating System Interface (POSIX)*. ISO, 1990.

[46] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, ATEC '05, pp. 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[47] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pp. 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[48] Uwe F. Mayer. Linux/unix nbench. `http://www.tux.org/~mayer/linux/bmark.html`

[49] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–13, New York, NY, USA, 2006. ACM.

[50] Sean McIlwain and Barton P. Miller. A tool for converting linux device drivers into solaris compatible binaries. *Softw. Pract. Exper.*, Vol. 36, No. 7, pp. 689–710, 2006.

[51] Fuchs P, Pemmasani G. Ndiswrapper. `http://ndiswrapper.sf.net/`

[52] Paul B. Ndisulator. `http://www.freebsd.org/doc/en/books/handbook/config-network-setup.html` 2003.

[53] Alt Richmond Inc. Scitech software snap device drivers. `http://www.altrichmond.ca/index.html`

[54] Project UDI. Uniform Driver Interface. `http://www.project-udi.org/`

[55] Checconi Fabio. Linux KVM on FreeBSD. `http://retis.sssup.it/~fabio/freebsd/lkvm/`

[56] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (nfs) version 4 protocol. 2003.

[57] 高橋一志, 笹田耕一. 差分転送を用いた高速な vm ディスクイメージ転送手法の提案. コンピュータシステム・シンポジウム (ComSys 2011), November 2011.

[58] 高橋一志, 笹田耕一. WinKVM : 異なるホスト OS 間の VM ライブマイグレーション実現に向けて. 情報処理学会コンピュータシステム・シンポジウム論文集 (ComSys 2009), 第 2009 巻, pp. 59–66. 日本ソフトウェア科学会, November 2009.

[59] 崇宏広渕, 宏高小川, 秀基中田, 智伊藤, 智嗣関口. 仮想計算機遠隔ライブマイグレーションのための透過的なストレージ再配置機構. 情報処理学会論文誌. コンピューティングシステム, Vol. 2, No. 2, pp. 152–165, July 2009.

[60] MokaFive. MokaFive Player. `http://www.moka5.com/`

[61] Shivani Sud, Roy Want, Trevor Pering, Kent Lyons, Barbara Rosario, and Michelle X. Gong. Dynamic migration of computation through virtualization of the mobile platform. In *MobiCASE*, pp. 59–71, 2009.

[62] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-

05, Australian National University, Department of Computer Science, June 1996. http://rsync.samba.org.

[63] J Black. Compare-by-hash: a reasoned analysis. *Proc 2006 USENIX Annual Technical Conference*, pp. 85–90, 2006.

[64] Val Henson and Richard Henderson. Guidelines for using compare-by-hash. 2005.

[65] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. April 1995. Supersedes FIPS PUB 180 1993 May 11.

[66] R. Rivest. The md5 message-digest algorithm, 1992.

[67] Michael O. Rabin. Fingerprinting by random polynomials. Technical report, Center for Research in Computing Technology, Harvard University, 1981.

[68] Yingwei Luo, Binbin Zhang, Xiaolin Wang, Zhenlin Wang, Yifeng Sun, and Haogang Chen. Live and incremental whole-system migration of virtual machines using block-bitmap. In *CLUSTER'08*, pp. 99–106, 2008.

[69] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pp. 169–179, New York, NY, USA, 2007. ACM.

[70] Takahiro Hirofuchi, Hirotaka Ogawa, Hidemoto Nakada, Satoshi Itoh, and Satoshi Sekiguchi. A live storage migration mechanism over wan for relocatable virtual machine services on clouds. *Cluster Computing and the Grid, IEEE International Symposium on*, Vol. 0, pp. 460–465, 2009.

[71] Niraj Tolia, Niraj Tolia, Thomas Bressoud, Thomas Bressoud, Michael Kozuch, Michael Kozuch, and Mahadev Satyanarayanan. Using content addressing to transfer virtual machine state. Technical report, 2002.

[72] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, David, and C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, Vol. 39, pp. 447–459, 1990.

[73] VMware, Inc. VMware Storage VMotion: Non-disruptive, live migration of virtual machine storage. `http://www.vmware.com/products/storage-vmotion/`

[74] Ali Mashtizadeh, Emré Celebi, Tal Garfinkel, and Min Cai. The design and evolution of live storage migration in vmware esx. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pp. 14–14, Berkeley, CA, USA, 2011. USENIX Association.

[75] 穐山空道, 広渕崇宏, 高野了成, 本位田真一. メモリの再利用により移動後の性能低下を抑えたライブマイグレーション. 先進的計算基盤システムシンポジウム論文集 (SACSIS), 第 2011 巻, pp. 258–259, May 2011.

[76] 穐山空道, 広渕崇宏, 高野了成, 本位田真一. 都鳥:メモリ再利用による連続するライブマイグレーションの最適化. コンピュータシステム・シンポジウム論文集 (ComSys), 第 2011

巻, pp. 2–11, November 2011.

謝辞

博士過程前期後期の 5 年間を戦い抜くにあたり，支えてくださった多くの方々に感謝の意をこめて．

何よりもはじめに，笹田耕一先生（現 Heroku, Inc.），千葉滋教授，竹内郁雄教授（現早稲田大学教授）に深く感謝します．先生方にはこの論文で論じたすべての研究に対して，日頃のディスカッションから，論文執筆，日頃の研究の進め方など，あらゆる点で関与していただきました．深く感謝いたします．

本論文の主査を引き受けてくださいました江崎浩教授，また，審査を引き受けてくださいました本位田真一教授，稲葉真理准教授、品川高廣准教授、中山英樹講師に感謝致します．

田辺良則先生（現国立情報学研究所），三好健文先生 (元創造情報学専攻特任助教授) にも深く感謝いたします．先生方からは，特に，3 章で述べた VoXY の研究に対して，実装段階のディスカッションから，論文執筆に関するアドバイスまで深く関与していただきました．深く感謝いたします．

竹内研究室 OB の村崎大輔氏にも深く感謝します．村崎氏には 4 章で論じた WinKVM の実装に関する深いアドバイスと，実装実験時に多用した RAM ディスク版 Linux のゲスト OS を作ってくださいました．WinKVM の研究と評価をうまく完遂できたのは氏のおかげです．深く感謝致します．

多忙の中，急な呼びつけに対しても嫌な顔ひとつせず，快く論文のチェックを引き受けてくださいました李相錫氏に感謝致します．

研究室後輩であった，胡益氏と顧モンテン氏にも感謝いたします．

また，元研究室メンバーの川島和澄氏，芝哲史氏にも深い感謝を．日ごろの研究活動において，実装段階で躓くとすぐにパニックに陥っていた私に対して，彼らは「心配するな，大丈夫だ，はやく，もう一度問題点を分析する作業にもどるんだ」と常に励ましの言葉を投げかけてくれました．彼らが居なければ私はくじけていたでしょう．深く感謝します．

最後の最後になりましたが，経済的かつ精神的に援助していただいた父（高橋洋一）と母（高橋秋恵）に深く深く感謝いたします．

ついでに，僕が博士論文執筆中にがぶ呑みしたラークとピースにも感謝を！