2008－OS－107 (14)
2008／1／31

仮想計算機におけるデバイスエミュレーションの特化による高速化

山本 悠輔[†] 新城 靖[†,*] 榮樂 英樹[†] 板野 肯三[†,*] 佐藤 聡[†] 中井 央[‡] 加藤 和彦[†,*]

†筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
‡筑波大学図書館情報メディア研究科研究科図書館情報メディア専攻
＊科学技術振興機構

要旨

仮想計算機において、仮想計算機モニタは、ゲストOSに対して実機と同じ抽象を提供するために、デバイスのエミュレーションを行う。このデバイスのエミュレーションは、重たい処理を含み、このため入出力性能が低下する。この問題を解決するために、準仮想ドライバと呼ばれる、仮想計算機モニタの機能を利用したデバイスドライバをゲストOSに組込む方法がとられている。この方法では、入出力の性能はよいが、デバイスドライバの開発に大きな労力を必要とする。この論文は、仮想計算機において高速なデバイスドライバを自動生成する方法を提案している。この方法では、まず、ゲストOSの既存のデバイスドライバと仮想計算機のデバイスエミュレータを、関数呼出しにより結合する。次に結合されたプログラムを、特化 (specialization) または部分評価により高速化する。提案手法は、x86 を対象とした仮想計算機モニタ LilyVM において実装が進められている。

Speeding up device emulation by specialization in virtual machines

Yusuke Yamamoto[†] Yasushi Shinjo[†,*] Hideki Eiraku[†] Kozo Itano[†,*]
Akira Sato[†] Hisasi Nakai[‡] Kazuhiko Kato[†,*]

†Department of ComputerScience, University of Tsukuba
‡Graduate School of Library, Information and Media Studies, University of Tsukuba
＊Japan Science and Technology Agency

Abstract

In a virtual machine, the virtual machine monitor (VMM) performs device emulation to provide the same abstraction as a physical machine for guest operating systems. This device emulation requires a large amount of resources, and decreases the I/O performance. To address this problem, dedicated device drivers called para-virtual device are installed into guest operating systems. However, this method requires large efforts to develop such device drivers that are specific to individual VMMs. This paper proposes a new method to generate fast device drivers for virtual machines. First, this method links an existing device driver in a guest OS with the device emulator in the VMM by regular function calling. Next, the method speeds up the linked program by specialization or partial evaluation. The proposed method is being implemented in the virtual machine monitor LilyVM for the x86 architecture.

1 はじめに

仮想計算機 (Virtual Machine, VM) において、入出力性能を改善することは重要な課題になっている。伝統的な仮想計算機では、実機と同じ抽象を提供するために、仮想計算機モニタ (Virtual Machine Monitor, VMM) において入出力デバイスのエミュレーションが行われている。たとえば、VMware Workstation[14] では、ディスクについては標準的な IDE や SCSI デバイスのエミュレーションを行っている。このデバイスのエミュレーションは、重たい処理を含み、仮想計算機の入出力性能を大きく低下させる原因になっている。この方式に基づく仮想計算機の入出力方式を、**エミュレーション方式**と呼ぶことにする。

仮想計算機の入出力性能を改善する方法として、準仮想化に基づき特殊なデバイスドライバをゲスト OS(operating system) に組込む方法がある。準仮想化 (para-virtualization) とは、実機を仮想化する際に仮想化が困難な部分についてはあえて仮想化を行わず、実機とは異なる抽象を提供するような仮想化である。準仮想ドライバとは、ゲスト OS の中で他の一般のデバイスドライバと同じインタフェースを実装しているが、入出力においてハードウェアのデバイスを操作するのではなく VMM の機能を利用するものである。たとえば、VMware Workstation では、グラフィックスの性能を改善するために、準仮想ドライバを用いている。この方式に基づく仮想計算機の入出力方式を、**準仮想方式**と呼ぶことにする。

準仮想方式を使うと、エミュレーション方式と比較して入出力性能を大きく改善することができる。しかしながら、準仮想方式には、VMM ごとにデバイスドライバを開発しなければならないという問題がある。たとえば、VMware Workstation では、Linux, Solaris, FressBSD, および Windows のために準仮想ドライバを提供しているが、他の OS には提供していない。また、その Linux 用のデバイスドライバも、Linux KVM[7] や Xen[2] 等の他の VMM で Linux を動作させたとしても利用することはできない。

この論文では、エミュレーションによる性能低下の問題と準仮想化における移植の問題を同時に解決する方法を提案する。具体的には、まず既存のゲスト OS デバイスドライバと VMM のデバイスエミュレータがともに多くの場合 C 言語で記述している点に注目し、両者を関数呼出しにより結合する。次に、特化 (specialization) または部分評価 (partial evaluation) を用いることにより、両者を合わせて高速化する。

現在、提案手法を、我々が開発している LilyVM において実装している。LilyVM は、x86 アーキテクチャを対象とした、準仮想化に基づく VMM である [4, 5]。LilyVM の特徴は、言語処理系を利用することで、ゲスト OS のうちアーキテクチャ依存部分の移植の労力を削減していることにある。LilyVM では、ゲスト OS と同じアドレス空間に実行時ライブラリを置く。この言語処理系を利用することと実行時ライブラリの存在が、本提案手法に適している。

2 特化を用いた高速な準仮想ドライバの自動生成

システムソフトウェアにおいて、特化を用いた高速化は、様々な場所に適応されてきた [9, 10, 13]。これらの研究では、RPC スタックにおけるコピーのオーバヘッドやマルチスレッドプログラムにおけるスレッド固有データへのアクセスのオーバヘッドが削減されている。本研究では、仮想化にともなうオーバヘッドを削減する。

図 1 に、提案方式の概要を示す。本方式では、まず、既存の実機用のデバイスドライバを用意する。このデバイスドライバは、ゲスト OS に固有のコードと入出力命令の発行を含む。次に、VMM からデバイスのエミュレータを抜き出す。このエミュレータは、ゲスト OS からは独立しており、入出力命令を受け取ると動作する手続きの集合になっている。このエミュレータは、ゲスト OS と同じアドレス空間で動作し、最終的に入出力が必要な時には、ハイパーバイザコールを用いてホスト OS の機能を利用する。たとえば、ディスクデバイスでは、セクタの読み書きを行う時にホスト OS の機能を利用する。

本方式の第 1 段階では、この既存のデバイスドライバと VMM から抜き出したデバイスエミュレータを、スタブを追加することで結合する。スタブは、ゲスト OS において入出力命令を発行する手続きと同じインタフェースを持つが、内部の実装は異なり、エミュレータの手続きを呼び出すものである。

次に、本方式の第 2 段階では、結合したプログラムを特化により高速化する。特化により、RPC ス

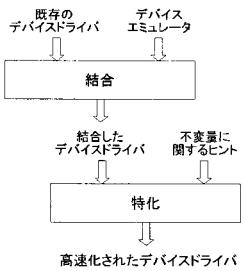

図 1: 特化を用いた高速な準仮想ドライバの自動生成

タックと同様に、手続きがインラインで展開され、いくつかの層がつぶされる。また、if 文が削除されたり、ループが展開されることもある。結果として、高速なデバイスドライバが自動生成されることになる。本方式は次の利点を持つ。

- 既存のゲスト OS 用のデバイスドライバを再利用することができる。ゲスト OS を VMM に移植する手間が削減される。ゲスト OS と VMM のメンテナンスを独立に行える。
- 生成されたデバイスドライバは、特化が理想的に行われれば、手書きの準仮想ドライバと同じ性能を持つ。

手書きの準仮想デバイスドライバの内部にも、もともと特化により高速化可能な部分を含んでいる可能性もある。その点も考慮すれば、生成されたデバイスドライバは、特化を用いない手書きのものよりも高速に動作する可能性もある。

2.1 Tempoによるコンパイル時特化と実行時特化

本研究では、特化を行う言語処理系として Tempo を用いる。Tempo は、C 言語、および、Java 言語を対象とした部分評価、または、特化を行う言語処理系である [3]。Tempo は、**コンパイル時特化** (Compile-time specialization) と**実行時特化** (run-time specialization) の両者を行うことができる。Tempo は、C 言語で記述されたプログラムと C 言語、および、ML 言語で記述されたヒントを読みこむ。ヒントは、主に不変量 (invariants) を記述したものである。Tempo は、束縛時解析 (Binding Time Analysis) [6] と呼ばれる手法で、プログラムの中の変数が不変量にのみ依存していることを発見する。不変量がすべてコンパイル時にわかる場合、コンパイル時特化に基づき、より高度な C 言語のプログラムを出力する。不変量が実行時（関数を呼ぶ前）にわかる場合、実行時特化を行い、その結果として、C 言語で記述された 2 つのプログラムを出力する。1 つは、テンプレートと呼ばれ、内部に後で埋めるべき「穴」を持つ。もう 1 つは、実行時特化プログラムであり、テンプレートを元に特化されたプログラムを動的に生成する。

本研究では、デバイスドライバとデバイスエミュレータを結合し、全体を Tempo を用いて特化を行うが、基本的にはコンパイル時特化だけを用いたいと考えている。その理由は、実行時のコード生成のオーバヘッドがないことや生成されたコードの管理が不要であることがあげられる。コンパイル時に特化を行うためには、コンパイル時に知られる不変量が多い方がよい。しかしながら、デバイスのエミュレーションにおいては、実行時 (OS 起動時) になって初めて決定される不変量もある。本研究では、このような不変量のいくつかをコンパイル時に決定する。この点について詳しくは、6 章で述べる。

3 LilyVM

2 章で提案した方式を、LilyVM において実現する [4, 5]。LilyVM は、我々が開発してる、x86 アーキテクチャを対象とした Type II の VMM である。本研究の提案手法を適用する VMM として LilyVM を選択した理由は、次の 2 点である。

- LilyVM のコード変換と特化の相性がよい。LilyVM では、コンパイル時にアセンブリ言語のレベルでコード変換を行い、特権命令や関連したセンシティブな非特権命令をライブラリ呼出しに置き換える。この静的にコード変換を行うことは、ゲスト OS を再コンパイルすることに相当する。この再コンパイルは、本研究で行う特化でも必要になる。
- LilyVM では、VM ライブラリと呼ばれる VMM のモジュールを、ゲスト OS のアドレス空間内で動作させる。この VM ライブラリの中にデバイスのエミュレータを置くことで、

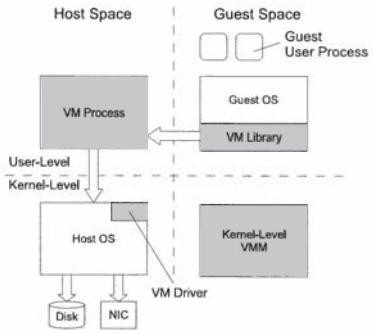

図 2: LilyVM の構成

手続き呼出しの形でエミュレータを動作させることができる。

LilyVM の構成を図 2 で示す。図において、ホスト空間 (Host Space) は、ホスト OS のカーネルとユーザプロセスが動作するアドレス空間を表し、ゲスト空間 (Guest Space) は、ゲスト OS のカーネルとそのユーザプロセスが動作するアドレス空間を表している。LilyVM は以下の 4 つのモジュールから構成される。

- VM ライブラリ (VM Library): ゲスト OS カーネルと同じアドレス空間と特権レベル（ユーザモード）で動作し、特権命令のエミュレーションを行う。
- カーネルレベル VMM (Kernel-level VMM): ゲスト空間において高い権限レベル（カーネルモード）で動作し、ゲスト OS に対して CPU 例外のリダイレクションを行う。
- VM ドライバ (VM Driver): VM プロセスとカーネルレベル VMM の間の通信を中継する。
- VM プロセス (VM Process): ホスト OS 上でユーザプロセスとして動作し、ホスト OS に対してシステムコールを発行する。

3.1 LilyVM における準仮想化方式による入出力

現在、LilyVM のディスクおよびネットワークの入出力処理は、準仮想化方式により行っている。

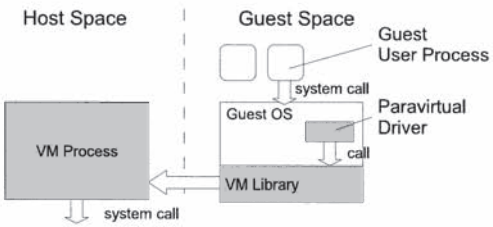

図 3: LilyVM における準仮想化方式による入出力

LilyVM における準仮想化方式による入出力方式を図 3 に示す。入出力の流れは以下の通りである。

1. ゲスト OS のユーザプロセスは、ファイル入出力およびネットワーク通信のシステムコールを実行する。
2. ゲスト OS カーネルは、準仮想ドライバを呼び出す。
3. 準仮想ドライバは、VM ライブラリを呼び出す。
4. VM ライブラリは、共有メモリを使い VM プロセスに入出力要求を送る。また、カーネルレベル VMM と VM ドライバを経由して制御を VM プロセスに移す。
5. VM プロセスは、VM ライブラリから共有メモリを通して入出力を受ける。そして、ホスト OS のシステムコールを利用して、入出力を行う。

LilyVM の VM ライブラリは、ゲスト空間で動作している。ゲスト空間はホスト OS が動作するホスト空間とは完全に独立しているので、ゲスト空間からはホスト OS に対してシステムコールを発行することはできない。したがって、図 3 のように VM プロセスにおいてシステムコールを発行する。

現在、LilyVM では、ディスクとネットワークの入出力をこの方式で実装している。ディスクの場合、VM プロセスは、通常のファイルに対して入出力を行う。ネットワークの場合、VM プロセスは、TUN/TAP デバイスに対して入出力を行う。このように現在の実装では、インストールするゲスト OS に準仮想ドライバをインストールすることが必須になっている。本研究では、デバイスのエミュレータを用いることで、この制約を無くする。

3.2 LilyVMにおけるエミュレーション方式による入出力

現在、LilyVMでは、次のようなデバイスのエミュレーションを行っている。

- 割込みコントローラ (PIC(Programmable Interrput controller))
- タイマ (PIT(Programmable Interval Timer), RTC(Real Time Clock))
- シリアルポート

これらのデバイスのエミュレーションは、主にVMライブラリの内部で行われており、必要な時にのみカーネルレベルVMMやVMプロセスを呼び出す。たとえば、シリアルポートのエミュレーションにおいては、VMプロセスに対してキーボードや画面に対する入出力要求を行う。本研究では、ディスクとネットワークについて、VMライブラリ内にエミュレータを追加する。

3.3 LilyVMの問題点

2章で述べた手法を適用するに当たって、現在のLilyVMには次のような問題点がある。

(1) VMライブラリのメモリ容量が不足

現在のLilyVMのVMライブラリは、ゲストOS上では、ちょうどBIOS ROMが存在場所に固定されている。この部分の大きさが、64 KBしかなく、ディスクデバイスやネットワークデバイスのエミュレータを置くにはメモリが不足している。

(2) PITのエミュレーションが不完全

現在のLilyVMはIntel i8259 PITを部分的にエミュレーションをしている。マスターデバイスのIRQ0から7までは割り込みはエミュレートされているが、スレーブデバイスのIRQ8から15まではエミュレートされていない。このためATAデバイスに使用されるしばしばIRQ14および15の割り込みを生成することができない。

(3) 拡張性の欠如

現在のLilyVMのソフトウェア構成は、拡張性を考慮せずに設計されており、新たなデバイスのエミュレータを追加するための枠組みが提供されていない。たとえば、通常のOSであれば、デバイスドライバやプロセスファイルシステムという枠組みでモジュールを追加することが容易になるように設計されている。しかしながら、現在のLilyVMにはそのような枠組みは存在しないので、機能を追加するにはプログラム全体を把握する必要がある。

本研究では、提案手法を実現する前に、これらのLilyVM問題点を解決する。

4 LilyVMに対するディスクデバイスとネットワークデバイスのエミュレータの追加

2章で述べた特化に基づく高速なデバイスドライバ生成手法をLilyVMにおいて実現するために、本研究では、まず、ディスクデバイスとネットワークデバイスのエミュレータをLilyVMのVMライブラリに追加する。対象としてディスクとネットワークを選んだ理由としては、両者の高速化に対する要求が高いこと、および、LilyVMに対するゲストOSの移植において、両者のデバイスドライバの移植が大きな労力を占めていることにある。

本研究で実現する、デバイスエミュレータの動作を図4に示す。図3の準仮想化方式と異なる点は、既存のゲストOSデバイスドライバをそのまま使用していること、およびVMライブラリにデバイスのエミュレータを組み込んでいることである。

VMライブラリに組込んだデバイスのエミュレータは、基本的には、QEMU[1]等のPCエミュレータに組込まれているデバイスエミュレータと同じような動作を行う。すなわち、内部にデバイスのモデルとなるオートマトンを持ち、入力命令や出力命令の実行を契機に状態を遷移する。QEMUのエミュレータとの違いは、第1に、入出力命令の実行というイベントをゲストOSから関数呼出しの形で受け取る点にある。第2の違いは、外界との相互作用が必要な時には、VMプロセスに依頼する点にある。

4.1 PCIバスのエミュレーション

本研究では、ディスクデバイスとネットワークデバイスのエミュレーションを行うが、それにはPCIバス[11]のエミュレーションが不可欠である。なぜならば、ディスクデバイスやネットワークデバイスは、PCIバスに接続するものが多くあり、また、ゲストOSとなるOSも、そのようなデバイス用のド

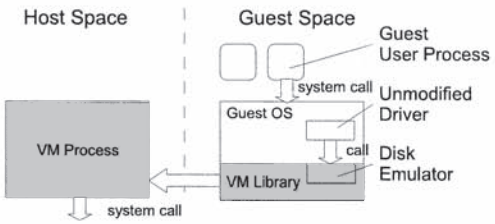

図 4: LilyVM におけるデバイスエミュレーション方式による入出力

ライバを標準的に備えているからである。

PCI バスに接続された個々のデバイスを利用するためには、多くの場合、次の 3 段階を踏む。

1. OS 起動時に、個々のデバイスを検出する。
2. 検出されたデバイスについて、コンフィギュレーション空間と呼ばれる、個々のデバイスを操作するために専用の I/O 空間を割り当てる。
3. 個々のデバイスのドライバは、ゲスト OS から利用すべきコンフィギュレーション空間と割込み番号を得て、デバイスを設定する。

PCI バスのエミュレータは、上の 1 と 2 に対してハードウェアと同じように振る舞う。

4.2 ディスクデバイスのエミュレーション

本研究では、ディスクデバイスとして Intel 82371SB PIIX3 IDE ディスクコントローラのエミュレーションを行う。このデバイスを選択した理由は、QEMU においても同じデバイスのエミュレーションを行っているので、参考になるからである。

本ディスクエミュレータで解釈するコマンドは、次の 6 種類に分類される。

1. Non-data: デバイスとホスト間とで、データとやり取りのないもの。
2. DEVCE RESET: デバイスをリセットするもの。
3. EXECUTE DEVICE DIAGNOSTIC: デバイスを診断するもの。
4. PIO(Programmed I/O) data-in: デバイスからホストへのデータ転送で入力命令用いるもの。
5. PIO(Programmed I/O) data-out: ホストからデバイスへのデータ転送で出力命令を用いるもの。
6. DMA(Direct Memory Access): デバイスとホスト間のデータ転送を DMA 転送を用いるもの。

これらのコマンドのうち、1 から 3 では、入出力が発生しない。したがって、ディスクのエミュレーションにおいても、VM ライブラリ内で処理が完結する。それに対して、4 から 6 では、入出力要求が発生する。この場合、本エミュレータは、LilyVM の VM プロセスにあるモジュールを呼び出す。このモジュールは、本来は、準仮想ドライバと対話するために設計されたものである。本研究は、そのコードを再利用して、ディスクデバイスのエミュレーションのためのバックエンドとして用いる。

4.3 ネットワークデバイスのエミュレーション

ディスクデバイスについては、ATA などの標準規格が存在する。これに対して、ネットワークデバイスについては、ATA のような規格は存在しない。そこで本研究では、既存のデバイスドライバを再利用するために、実在するデバイスを選択し、そのデバイスをエミュレートする。本研究では、ネットワークデバイスとして、Realtek RTL8139 をエミュレートする。このデバイスを選択した理由は、多くの OS でデバイスドライバが供給されているからである。

5 デバイスドライバとデバイスエミュレータの結合

本研究では、特化により高速化を行うことに先立ち、まず、ゲスト OS のデバイスドライバと VMM のデバイスエミュレータを結合する。両者は、ともに C 言語で記述させているが、デバイスドライバは、アセンブリ言語で入出力命令を実行しているので、単純には結合することはできない。よく設計されたデバイスドライバは入出力命令を行う部分を少数の関数に押し込めているという特徴がある。具体的には、inb() や outb() といった名前の関数やマクロを呼び出して、入出力命令を実行している。

本研究では、入出力命令が少数の関数で行われているという特徴を活用し、スタブを追加することで、デバイスドライバとデバイスエミュレータを結合す

る。スタブは、デバイスドライバが用いている入出力を行う関数やマクロと同じ引数をとり、同じ結果を返すが、実際には入出力を行わずにエミュレータ内の手続きを呼び出す。

スタブへの置き換えは、デバイスドライバのソースプログラムを修正するのではなく、C 言語のヘッダファイルを置き換えることで実現する。このヘッダファイルは、ゲスト OS ごとに作成する必要があるが、内容はそれほど複雑なものにはならない。

6 コンパイル時特化に適したデバイスのエミュレーション

本研究では、特化に Tempo を用いる。2 章で述べたように、Tempo は、コンパイル時特化と実行時特化の 2 種類をサポートしている。本研究では、コンパイル時特化に適したデバイスエミュレーションを行う。

PCI バスには、個々の PC ごとに様々なデバイスが装着される。したがって、通常の OS は、起動時にプローブを行い、デバイスに対して動的にポート番号と割込み番号を割り当てている。このポート番号と割込み番号は、不変量であり、これを利用して特化を行うことができる。しかしながら、動的に決定されるため、コンパイル時には不明であり、本来ならば 2 章で述べたように実行時特化が必要になる。

本研究では、本来ならば動的に決定されるポート番号を形式的に決定することで、コンパイル時特化を可能にする。形式的なポート番号とは、Tempo による解析を行う時にのみ有効なポート番号である。このような形式的なポート番号は、Tempo による解析が完了し、特化が完全に行われれば、出力されるコードには現れることはない。たとえば、次のようなコードを考える。

```
portno = 0x1f00;
netdev_driver() {    /* device driver  */
  lilyvm_vmlib_outb( data1, portno );
}

lilyvm_vmlib_outb( data, port ) {
  switch( port ) {
  case 0x1f00:
     netdev_emu( data, port & 0xff );
     ...
  }
}
netdev_emu( data, port ) {
  data を使ったエミュレーションのコード;
}
```

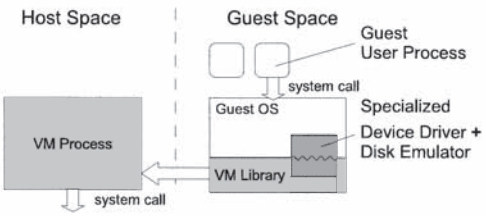

図 5: LilyVM における本提案方式に基づく入出力

ここで、0x1f00 は、このデバイスに割り当てた型式的なポート番号である。コンパイル時特化により、Tempo は次のようなコードを生成する。

```
netdev_driver() {
  data1 を使ったエミュレーションのコード;
}
```

このように、形式的に割り当てたポート番号は特化により削除され、最終的なコードには現れない。

このような特化の結果、全体の構成は、図 5 のようになる。図 4 と比較すると、デバイスドライバと VM ライブラリの境界がなくなり、複数の層がつぶされていることがわかる。

割込み番号についても、ポート番号と同様に形式的に割り当てることで、コンパイル時特化の対象となる。これにより、割込み処理のディスパッチを高速化することができる。

7 関連研究

Paravirt_ops[12] は、Linux 2.6.20 (x86) で導入された関数の表である。この表は、仮想化することが困難な命令を含む。たとえば、割込みの禁止・許可、ページテーブルの設定を行う命令を含んでいる。この表は、OS の起動時に各 VMM ごとに切り替えられる。現在、実機、Xen、および、VMware についてこの表が実装されている。この表は、入出力命令を含んでいない。Paravirt_ops と比較して本研究の特徴は、特化を行っていること、入出力命令を扱っていること、および、Linux 以外の OS にも対応できることである。

VMware VMI(Virtual Machine Interface)[15] は、標準的な準仮想化を目指して提案されたインタフェースである。これは、BIOS ROM のように使われることを想定して設計されている。ゲスト

OSをこのインタフェースを使うように書き換えれば、VMware Workstation等のこのインタフェースを実装したVMMでは高速に動作する。このインタフェースは、Paravirt_opsに含まれているような特権命令の他に、入出力命令も含んでいる。VMware VMIと比較して、本研究の特徴は、特化を行っていること、および、言語処理系により書き換えの手間を削減していることにある。

Afterburner[8]は、LilyVMの技術[4]を元にして、アセンブリ言語のレベルでの変換を行うことで仮想化を行っている。Afterburnerでは、ゲストOSカーネルの中のセンシティブな命令を見つけると、その後にエミュレーションで必要なメモリをnop命令を埋め込むことで確保する。ゲストOSをメモリにロードする時に、nop命令をVMMごとに合わせたエミュレーションコードに置き換える。Afterburnerと比較して、本研究の特徴は、C言語のレベルで特化を行っていることにある。

8 おわりに

この論文では、ゲストOSのデバイスドライバとVMMのデバイスエミュレータの結合し、特化を用いて高速化する方式について述べた。本方式では、まずゲストOSの既存のデバイスドライバと仮想計算機のデバイスエミュレータを、関数呼出しにより結合する。次に結合されたプログラムを、特化(specialization)または部分評価により高速化する。本研究では、特化を行う言語処理系として、Tempoを用いる。Tempoは、コンパイル時特化と実行時特化の2種類の特化をサポートしているが、本研究では、主にコンパイル時特化を用いる。このため、コンパイル時特化に適したデバイスのエミュレータを開発している。

提案方式を、我々が開発しているLilyVMにおいて実装している。LilyVMは、コンパイル時(アセンブル時)に命令を書き換えることで仮想化を行っているVMMであり、コンパイル時特化と適合性が高い。また、LilyVMのVMライブラリは、ゲストOSから直接呼び出すことができる点も、本提案方式に適している。

今後の課題は、デバイスのエミュレータを実現し、本提案方式の適用範囲を確認することである。LilyVMは、ゲストOSとして、Linux、FreeBSD、および、NetBSDを動作させることができるので、それらのOSについて本提案手法を適用する。そして、手書きの準仮想ドライバと本研究で特化により生成したデバイスドライバの性能を比較する。

参考文献

[1] Bellard, F.: QEMU, a fast and portable dynamic translator, *Proceedings of the USENIX Annual Technical Conference 2005*.

[2] Cambridge, U.: Xen 3.0 User Manual (2005).

[3] Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S., Volanschi, E.-N., Lawall, J. and Noyé, J.: Tempo: specializing systems applications and beyond, *ACM Comput. Surv.*, pp. 1299–03 (2003).

[4] Eiraku, H. and Shinjo, Y.: Running BSD kernels as user processes by partial emulation and rewriting of machine instructions, *Proceedings of the BSD Conference 2003*, pp. 10–10 (2003).

[5] 榮樂英樹, 新城靖, 加藤和彦カーネル・レベル・コードによるユーザ・レベルVMMの移植性の向上, 情報処理学会 第104回システムソフトウェアとオペレーティング・システム研究会, pp. 17–24 (2007年1月).

[6] Jones, N. D., Gomard, C. K. and Sestoft, P.: *Partial evaluation and automatic program generation*, Prentice-Hall, Inc. (1993).

[7] Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: kvm: the Linux Virtual Machine Monitor, *Proceedings of the Linux Symposium*, Ottawa, Ontario (2007).

[8] LeVasseur, J., Uhlig, V., Chapman, M., Chubb, P., Leslie, B. and Heiser, G.: Pre-Virtualization: Slashing the Cost of Virtualization (2005).

[9] Marlet, R., Thibault, S. and Consel, C.: Mapping software architectures to efficient implementations via partial evaluation, *Proceedings of the 12th international conference on Automated software engineering*, pp. 183–192 (1997).

[10] Marlet, R., Consel, C. and Boinot, P.: Efficient incremental run-time specialization for free, *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pp. 281–292 (1999).

[11] PCI-SIG: An Evolution of the Conventional PCI Local Bus Specification 3.0 (2002).

[12] Russell, R.: Paravirt_ops (2006).

[13] Shinjo, Y. and Pu, C.: Achieving Efficiency and Portability in Systems Software: A Case Study on POSIX-Compliant Multithreaded Programs, *IEEE Transactions on Software Engineering*, Vol. 31, No. 9, pp. 785–800 (2005).

[14] Sugerman, J., Venkitachalam, G. and Lim, B.-H.: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, *Proceedings of 2002 USENIX Annual Technical Conference*, pp. 1–14 (2001).

[15] VMware: Paravirtualization API Version 2.5 (2006).