Reverse VMRPCによるゲストOSの操作

山本 諒裕[1]　新城 靖[1]　Oscar Fernando Garcia Alvarado[1]

概要：本論文はホスト型仮想計算機モニタ (Virtual Machine Monitor, VMM) において，ホスト OS からゲスト OS を操作するための Remote Procedure Call (RPC) を提案する．この RPC を Reverse VMRPC と呼ぶ．Reverse VMRPC の特徴は，ネットワークを用いない点，またカーネル空間で実行される手続きを呼び出すことができるという点である．Reverse VMRPC は本研究室で開発された Virtual Machine Remote Procedure Call (VMRPC) という技術をベースにして設計および実装が行われている．また Reverse VMRPC の実装は OS に依存しない．VMRPC を提案手法の実装に用いるにあたっては，ホスト OS からゲスト OS への処理の遷移について課題がある．本研究ではこの課題を VM Long Polling (VMLP) を用いることによって解決する．提案手法はゲスト OS として Linux および FreeBSD で動作している．そして，Reverse VMRPC を用いてゲストのメモリの情報を取得したり，ファイルシステムのキャッシュを解放するといった操作が可能となっている．

1. はじめに

仮想計算機モニタ (Virtual Machine Monitor, VMM) は，OS を実行するための環境を仮想的に構築し動作させるソフトウェアである．VMM は計算機の資源を有効に活用するための不可欠の技術となった．

VMM はハイパーバイザ型とホスト型の 2 種類に大別される．ハイパーバイザ型 VMM は，ハードウェアの上にハイパーバイザという層を作成し，その上で仮想計算機を実行する．これに対しホスト型 VMM は，ハードウェア上で動作している OS (ホスト OS) を基盤として，仮想計算機を構築する．ホスト型 VMM を用いると，物理計算機上で動作するホスト OS と，仮想計算機上で動作する OS (ゲスト OS) を同時に利用することができる．また 1 台の物理計算機上で複数のゲスト OS を同時に動作させることも可能である．本研究はホスト型 VMM を対象としている．

ホスト OS からゲスト OS の動作状況を得たいという要求や，ゲスト OS のカーネルに対して操作を行いたいという要求が存在する．例えばゲスト OS のメモリ消費量や負荷をリアルタイムに取得したり，バッファキャッシュの解放を行うことが考えられる．文献 [1] では，ゲスト OS が持つ様々なメモリ利用統計を用いて仮想計算機のワーキングセットサイズ (Working Set Size) を見積もっている．また文献 [2] では，ゲスト OS の仮想メモリ管理を高速化するために，ホスト OS がゲスト OS のページの情報を取得している．

ホスト型 VMM において，ゲスト OS とホスト OS は互いに独立している．そのため一般的に，ゲスト OS からはホスト OS のメモリ空間は隠蔽されている．逆に，ホスト OS からゲスト OS のメモリを参照することは可能ではあるが，容易ではない．両者が情報の交換を行うためには，SSH や Network File System (NFS) など，物理的に異なる計算機間の通信と同様の手法を用いることが一般的である．これらの手法は，TCP/IP などでネットワーク通信を行う．これらネットワークによる手法を用いるには，ホスト OS とゲスト OS の双方がネットワーク通信機能を持ち，安全に接続されている必要がある．また通信にかかるオーバーヘッドも大きい．

ホスト OS からゲスト OS ではなく，ゲスト OS からホスト OS を利用する手法として，本研究室では Virtual Machine Remote Procedure Call (VMRPC) を提案している [3][4]．VMRPC はゲスト OS からホスト OS に対して Remote Procedure Call (RPC) を行う手法である．VMRPC により，ネットワークによらず高速にゲスト OS からホスト OS のカーネルの機能を利用することができる．しかし，逆にホスト OS からゲスト OS のカーネルの機能を利用することはできない．

ネットワークによらず，ホスト OS からゲスト OS のカーネルを直接操作する手法として，SymCall が提案されている [2]．SymCall は VMM がゲスト OS 上の手続きを同期的に呼び出すことを可能にするインターフェースで，システムコールと可能な限り近付けるように設計が行われてい

[1] 筑波大学
University of Tsukuba

る．ただし，SymCall の設計と実装は複雑である．

本研究では，ネットワークを用いず，高速かつ容易にホスト OS からゲスト OS に対し RPC を行う手法を提案する．この手法を，VMRPC の逆方向であることから Reverse VMRPC と呼ぶ．Reverse VMRPC は，オーバーヘッドが非常に小さく，またユーザレベルの手続きみならず，カーネル空間で実行される手続きを呼び出すことも可能である．具体的にはゲストのメモリ情報を取得する，ファイルシステムのキャッシュを解放するといった操作が可能である．SymCall と異なり，Reverse VMRPC は VMRPC という既存の仕組みを拡張する形で実現する．Reverse VMRPC は全てを C 言語のコードで実装しており，VMRPC に関係するものを除けば，カーネルに標準で装備された API のみを用いて実装可能である．

本論文は次のような構成となっている．2 章では Reverse VMRPC のベースとなる VMRPC について述べる．3 章では Reverse VMRPC の実装に際しての課題を述べる．4 章および 5 章で Reverse VMRPC の設計と実装についてそれぞれ述べ，6 章では Linux ゲスト，7 章では FreeBSD ゲストに対し Reverse VMRPC を用いて実際にゲスト OS を操作する．8 章で研究を評価し，9 章で関連研究について述べ，10 章で本論文をまとめる．

2. VMRPC

本研究室では，ゲスト OS からホスト OS の手続きを Remote Procedure Call (RPC) の形式で呼び出す VMRPC を開発している．本研究では，VMRPC をベースとして Reverse VMRPC の設計と実装を行う．この章では，VMRPC の概要を述べる．

VMRPC はアウトソーシング (Outsourcing) を実現するための手段として設計された [4]．アウトソーシングは，ホスト OS に対してゲスト OS の高水準の要求の処理を委譲する手法である．従来の準仮想化手法がデバイスドライバという低水準のモジュールを置き換えるものだったのに対し，アウトソーシングはソケット層など，より高い水準のモジュールを置き換える．そのため，ホスト OS はゲスト OS の要求をより柔軟に処理することができる．現在までにソケットオブジェクトや仮想ファイルシステムのアウトソーシングが実装されている [4][5]．

VMRPC では，実行させたい手続きをあらかじめホスト OS にサーバとして登録しておくことで，ゲスト OS がクライアントとしてその手続きを呼び出せるようになる．サーバの手続きの本体は，ユーザ空間，カーネル空間のいずれに存在していてもよい．ホスト OS 上の手続きを呼び出すための API として，ゲスト側のクライアントは vmrpc_perform() および vmrpc_perform_kernel() を持つ．前者はユーザ空間 (VMM のプロセスの中)，後者はカーネル空間の手続きを呼び出す API である．vmrpc_perform() および vmrpc_perform_kernel() は，引数として手続き番号と，リクエストメッセージの先頭番地を取る．分散システムにおける RPC とは異なり，VMRPC では引数にポインタ (ゲスト OS 内の変数の番地) を含むことができる．ホスト OS は，ポインタを使ってゲスト OS 内の変数を読むことができる．VMRPC の結果も，ポインタの先の変数に書き込むことができる．VMRPC では，ゲスト OS 内の変数を読み書きするための API として copy_from_guest() および copy_to_guest() を提供する．前者はゲストのメモリ空間からホストのメモリ空間へデータのコピーを行い，後者はホストのメモリ空間からゲストのメモリ空間へコピーを行う．

これらの API で用いられるアドレスは Guest Vitual Address (GVA) である．ホスト OS は GVA を直接扱うことができない．ホスト側のサーバは get_hva() という API を持っており，GVA をホスト OS で直接扱える Host Virtual Address (HVA) に変換することができる．copy_from_guest() および copy_to_guest() は内部で get_hva() を使用している．

図 1 と図 2 は，VMRPC を利用してゲスト-ホスト間通信を行うコードの主要部分である．これらのコードは，ゲスト OS のカーネルが保持する文字列データを，ホスト OS のカーネルログへ書き込むという処理を行っている．図 1 はゲスト側のクライアント，図 2 はホスト側のサーバである．なお，本論文に示すコードは，エラー処理の記述を省略している．

図 1 に示したゲスト側のコードでは，vmrpc_bind_module() でサーバとのバインディングを行い，vmrpc_perform_kernel() で RPC を実行している．vmrpc_perform_kernel() には引数として，バインディングで得られたサーバの識別子，手続き番号 PRINT および文字列 str の先頭番地とその長さを与えている．図 2 に示したホスト側のコードでは，copy_from_guest() でゲストのメモリ空間に存在する文字列を，ホストのメモリ空間に確保した領域 buf へコピーしている．関数 print_server の引数 arg には，ゲスト側コードの vmrpc_perform_kernel() の引数 str に格納された値と等しい値が格納されている．先述の通り，この値は GVA である．

本研究で利用する VMRPC は，Intel VT-x の仮想化支援機能を用いて RPC を実現している．vmrpc_perform() および vmrpc_perform_kernel() の実装では VMCALL を実行し，ゲストからホストへのコンテキストスイッチ (VM Exit) を引き起こす．ホスト側のサーバは，ゲストから受け取った手続き番号と引数に従って個別の手続きを呼び出す．手続きが完了すると，ホストからゲストへコンテキストスイッチ (VM Entry) を引き起こす．その後ゲストでは vmrpc_perform() または vmrpc_perform_kernel() 以降に記述された処理が続行する．

```
int client(void) {
    int handle;
    char str[] = "Hello from Guest";
    handle = vmrpc_bind_module_kernel(
        "MOD_PRINT", "1");
    vmrpc_perform_kernel(handle, PRINT,
        str, sizeof(str));
    vmrpc_unbind_module_kernel(handle);
    return 0;
}
```

図 1 VMRPC クライアントの例 (ゲスト)

```
int print_server_print(void *rpcdata,
    gva_t arg, size_t bytes) {
    char buf[BUFSIZE];
    copy_from_guest(
        rpcdata, buf, arg, bytes);
    printk(KERN_INFO "print: %s\n", buf);
    return 0;
}

int print_server(void *rpcdata,
    int func, gva_t arg, size_t bytes) {
    switch(func) {
    case PRINT:
        ret = print_server_print(
            rpcdata, arg, bytes);
        break;
    case ...

    case ...

    default:
        ret = 0;
    }
    return ret;
}
```

図 2 VMRPC サーバの例 (ホスト)

3. Reverse VMRPC 実装にあたっての課題と対策

Reverse VMRPC の実装にあたっては，VMRPC における vmrpc_perform() や vmrpc_perform_kernel() に相当する API が必要である．しかし，VMRPC と同様のアプローチでこれらの API を実装することは困難である．この章では Reverse VMRPC 実装にあたっての課題と，課題への対策について述べる．

3.1 ホスト OS からゲスト OS への処理の遷移

Reverse VMRPC の実装ではまず，任意のタイミングで VM Entry を発生させる方法が課題となる．2 章で述べたように，VM Exit を発生させるためには VMCALL を用いればよいため，任意のタイミングで VM Exit を発生させることは容易である．これに対し，VM Entry を任意のタイミング発生させることは容易ではない．

この課題への対策として，Reverse VMRPC では vmrpc_perform_kernel() で呼び出されたホスト側の手続きがリターンした時点で発生する VM Entry を任意のタイミングで発生させる．ホスト側は，リクエストがなされるまで自身のスレッドをスリープ状態にする．クライアントがリクエストを行った時点で．それをゲスト OS のメモリ空間に書き込み，リターンする．

3.2 ホスト側でのスリープ

上記で述べたように，VM Entry を任意のタイミングで発生させるためにホスト側でスリープしたい．しかしここにも課題が存在する．

VMRPC は，ホストがゲストからの要求に対し瞬時に処理を行い，結果を返すことが前提となる仕組みである．プロセッサが VMX root モードでホスト側の関数を実行している間，ゲスト側にプロセッサは割り当てられないため，ゲスト OS はフリーズ状態となる．VMRPC では，スリープを伴う関数をホスト側で使用できないという制約がある．例えば，まだ届いていないメッセージを受け付ける recvmsg() やセマフォ操作の down() などがスリープを伴う関数にあたる．一方，Reverse VMRPC を実現するにあたっては前節で述べた通り，ホスト側でのスリープが必要となる．そのため，従来の VMRPC の API では，Reverse VMRPC 実装にあたって必要な要件をクリアできない．

本研究では，上記の課題に対応するために，VM Long Polling (VMLP) という新たな仕組みを用いる．次節で VMLP の概要を述べる．

3.3 VM Long Polling (VMLP) の概要

VMLP は VMRPC においてホスト側でのスリープを可能にするための仕組みである．VMLP では，VMM のゲスト-ホスト間で分散システムにおける Long Polling と同等の機構を実現する．

分散システムにおける Long Polling は，サーバからのプッシュを擬似的に実現する手法である．例えば一般的な HTTP では，ユーザからのリクエストに対しサーバが即座にレスポンスを返すが，Long Polling では，サーバはリクエストを受け取ると，クライアントとのコネクションをを保持したままスリープする．クライアントへ通知すべき情報が発行されたタイミングではじめて，サーバはクライアントへレスポンスを返す．

VMLP では，ゲストに予め 2 つ以上の仮想 CPU (Virtual CPU, vCPU) を割り当てる．例えば vCPU を 2 つ設定した状態で VMM を起動した場合，ゲスト OS からは CPU が 2 つあるように見える．ゲスト OS が 1 つの CPU で

VMCALL を実行したとき，vCPU のうち 1 つはホスト OS 側の処理に回る．しかしもう 1 つの vCPU はゲスト OS 側で動作を続ける．このため Reverse VMRPC で，ホスト OS でスレッドがスリープされた状態でゲスト OS を実行し続けることができる．

本研究では, Linux KVM[6] において, 次のように VMLP を実装した．KVM では，vCPU はホスト OS のカーネルスレッドで実装されている．各 vCPU には，1 つのカーネルスレッドが割り当てられている．通常の KVM では，vCPU に対応したカーネルスレッドがスリープすると，仮想計算機全体がフリーズするという問題がある．その理由は，VM Exit した際，vCPU に対応したカーネルスレッドが大域的なロックを保持しているからである．本研究では KVM を修正し，スリープする前にロックを解放し，スリープから回復後に再びロックを保持するようにした．

4. Reverse VMRPC の設計

本研究で提案する Reverse VMRPC は，VMRPC の反対方向，すなわちホスト OS からゲスト OS の手続きを呼び出す仕組みである．すなわち，ホスト OS はクライアントとしてリクエストメッセージを生成し，ゲスト OS がサーバとしてメッセージに従い処理を行う．Reverse VMRPC が NFS や SSH 等，ネットワークを用いた手法と異なるは，カーネル空間の手続きを，ネットワークを用いずに呼び出すことが可能な点である．加えて，カーネル内の情報を，ユーザ空間を介さずに取得することができる．このため例えばゲスト OS のカーネル内の，逐次更新される情報を非常に低いレイテンシで取得することが可能である．

4.1 概要

本研究では, ホスト OS として Linux, VMM として KVM を用いて Reverse VMRPC を実装する．Reverse VMRPC はホスト OS のカーネル空間に組み込まれるホスト側モジュールと，ゲスト OS のカーネル空間に組み込まれるゲスト側モジュールで構成される．

Reverse VMRPC の構成要素を図 3 に示す．Reverse VMRPC の実装には VMRPC および VMLP を用いる．本研究で用いる VMRPC および VMLP は，KVM のために実装されたものである．KVM は Intel VT-x の機能を利用している．Reverse VMRPC の上に，それを利用してゲスト OS のメモリ資源や CPU 資源の利用状況をモニタするアプリケーションやバッファキャッシュを解放するアプリケーションを実装する．

Reverse VMRPC は全てを C 言語のコードで実装しており，VMRPC に関係するものを除けば，カーネルに標準で装備された API のみを用いて実装可能である．Reverse VMRPC の実装には VMLP の API を呼び出すコードは含まれていないが, VMLP の働きにより VMRPC でスリープ

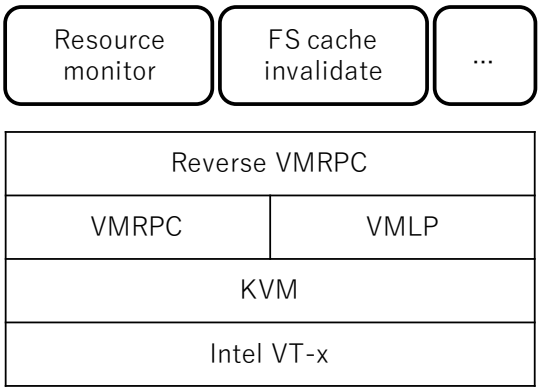

図 3 Reverse VMRPC の構成要素

できることを期待してる．したがって，VMRPC と VMLP が動作する VMM と OS であれば，Reverse VMRPC もそのまま動作する．本研究では VMM として KVM，ホスト OS として Linux, ゲスト OS として Linux および FreeBSD で Reverse VMRPC を動作させることができた．

4.2 Reverse VMRPC のクライアント

図 4 に Reverse VMRPC のクライアントのコードの例を示す．このコードは，クライアント側からサーバ側にメッセージを送っている．サーバは，クライアントから送られてきたメッセージをそのまま返す．クライアントは，サーバから受け取ったメッセージを printk() でカーネルログに書き込んでいる．

クライアントは Reverse VMRPC の API である rvmrpc_call() を呼び出している．rvmrpc_call() は VMRPC における vmrpc_perform() および vmrpc_perform_kernel() に相当する関数である．その引数は以下の通りである．

- サーバのモジュール番号
- 手続き番号
- 要求メッセージの先頭番地
- 要求メッセージの長さ
- 応答メッセージを受け取るためのメモリの先頭番地
- 応答メッセージを受け取るためのメモリの長さ

クライアントはホスト OS のカーネル内で動作するプログラムである．カーネル内で動作するプログラムであれば，その形態は限定されない．Loadable Kernel Module (LKM) として実装すれば，カーネル本体を再ビルドする必要はない．クライアント内でキャラクタデバイスを生成し，ユーザプログラムとのインターフェースを定義することでユーザ空間のプログラムからゲスト OS を操作することもできる．6 章ではクライアントを proc ファイルシステムとして実装する．

4.3 Reverse VMRPC のサーバ

図 5 に Reverse VMRPC のサーバのコードの例を示す．server_init() はサーバの初期化関数であり，この中で

```
int echo_client(void) {
    char str[] = "Hello from Host";
    char back_str[BUFSIZE];
    rvmrpc_call(MOD_ECHO, ECHO, arg, 16,
        back_str, BUFSIZE);
    printk(KERN_INFO "echo back: %s\n",
         back_str);
    return 0;
}
```

図 4 Reverse VMRPC クライアントの例 (ホスト)

register_server_module() を呼び出している．

register_server_module() はサーバにモジュールを登録する Reverse VMRPC の API で，モジュール番号と関数ポインタを引数に指定する．サーバはクライアントからリクエストを受け取ると，ここで登録したモジュール番号に対応した関数を呼び出す．

関数 echo_server() は Reverse VMRPC のディスパッチ関数であり，モジュール番号 MOD_ECHO で登録されたものである．この関数は手続き番号 req->func を解析し，個々の手続きを呼び出す．

関数 echo_server_echo() は手続き番号 ECHO に対応した手続きである．その引数は，要求メッセージを表す構造体へのポインタと，応答メッセージを格納するための構造体へのポインタである．これら構造体の詳細は 5.1 節で述べる．この関数は printk() でゲスト OS のカーネルログへ受け取った文字列を書き込んでいる．そして，受け取った文字列をそのまま応答メッセージにコピーしている．

サーバにモジュールを登録するには，ゲスト OS のカーネル内にディスパッチ関数を記述し，手続き番号を割り振る．クライアントから受け取るリクエストメッセージにはモジュール識別番号が含まれており，サーバはモジュール識別番号に一致するディスパッチ関数を選択して呼び出す．この関数は以下に示す条件に従う限り，特別な制約を受けることはない．

- 第 1 引数にリクエスト用バッファの先頭番地，第 2 引数にレスポンス用バッファの先頭番地を取る．
- クライアントから与えられパラメータはすべてリクエスト用バッファの中に存在する．
- クライアントへ返却したいパラメータはすべてレスポンス用バッファに置く必要がある．

5. Reverse VMRPC の実装

5.1 リクエストとレスポンスの受け渡し

2 章で述べたように，従来の VMRPC では，ホストのサーバで手続きを実行している間に copy_from_guest() や copy_to_guest() でリクエストメッセージやレスポンスメッセージの読み書きをしていた．これらの手続きでは，一時的にゲスト OS のメモリをマップし，データのコピーを行い，それが終わるとアンマップしていた．この方法でリクエストメッセージやレスポンスメッセージを読み書きする方法は，ホスト OS では可能であるがゲスト OS では不可能である．

```
int server_init(void) {
    ...
    register_server_module(MOD_ECHO,
        echo_server);
    register_server_module(..);
    ...
}
int echo_server(
    struct request *req,
    struct response *res) {
    switch(req->func) {
    case ECHO:
        ret = echo_server_echo(req, res);
        break;
    case ...
    case ...
    default:
        ret = 0;
    }
    return ret;
}
int echo_server_echo(struct request *req
    struct response *res)  {
    printk(KERN_INFO "echo: %s\n", req->arg);
    memcpy(res->res, req->arg, req->arglen);
    res->reslen = req->arglen;
    return 0;
}
```

図 5 Reverse VMRPC サーバの例 (ゲスト)

そこで Reverse VMRPC の実装では，次の方法でこの問題を解決した．

- ゲスト OS のアドレス空間内にリクエストメッセージおよびレスポンスメッセージを保存するためのバッファを確保する (図 6)．
- Reverse VMRPC の初期化時に，確保したバッファの番地と大きさをゲスト OS からホスト OS に vmrpc_perform_kernel() で渡す．
- ホスト OS は受け取った番地をもとに，バッファをホスト OS のアドレス空間にマップする．

Reverse VMRPC は，リクエストメッセージをバッファへ格納する際に，以下のフィールドを持つデータ構造を用いる．

mod モジュール番号

func 手続き番号

arglen リクエストメッセージのデータサイズ (バイト単位)

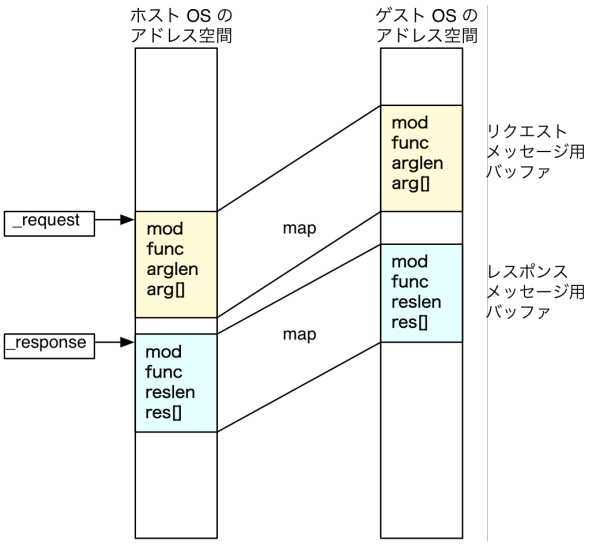

図 6 リクエストメッセージ用バッファおよびレスポンスメッセージ用バッファのマッピング

arg リクエストメッセージ本体を格納する場所

また，レスポンスメッセージをバッファへ格納する際には以下のフィールドを持つデータ構造を用いる．

mod モジュール番号

func 手続き番号

reslen レスポンスメッセージのデータサイズ (バイト単位)

result レスポンスメッセージ本体を格納する場所

5.2 Reverse VMRPC における制御の流れ

Reverse VMRPC は，以下のようなゲスト側で 1 つ，ホスト側で 2 つのスレッドを用いて実装する．

サーバスレッド ゲストに常駐している．VMRPC のクライアント．

プロキシスレッド KVM で vCPU を実装しているカーネルスレッド．ホストに常駐している．VMRPC のサーバ．

クライアントスレッド ホストで必要な時に動作する．

図 5 に示した Reverse VMRPC のサーバは，上記のサーバスレッドで実行される．これは，ゲスト OS では専用の CPU で実行される．VM Exit が生じると，ホスト OS では KVM の vCPU を実装しているカーネルスレッドに制御が移る．すなわち，上記のプロキシスレッドとなる．

図 4 に示した Reverse VMRPC のクライアントは，クライアントスレッドで動作する．これは専用のカーネルスレッドではなく，通常のプロセスに対応したカーネルスレッドであることもある．例えば 6.4 節の実験では，ホスト OS で ioctl() システムコールを実行したユーザプロセスに対応したカーネルスレッドが，Reverse VMRPC のクライアントとなる．

Reverse VMRPC における RPC は以下に示すような流れで行われる．概略を図 7 に示す．

(1) サーバスレッド:
リクエストメッセージおよびレスポンスメッセージを保存するためのバッファを確保する．その後，確保したバッファの番地と大きさを引数として vmrpc_perform_kernel() を呼び出す．

(2) プロキシスレッド:
vmrpc_perform_kernel() で受け取ったバッファをホスト OS のアドレス空間にマップする．その後スリープする．

(3) クライアントスレッド:
rvmrpc_call() を呼び出し，リクエストメッセージを 5.1 節で述べたバッファに書き込みプロキシを起こす．その後自身はスリープする．

(4) プロキシスレッド:
コンテキストをゲスト OS に戻す．

(5) サーバスレッド:
リクエストをリクエストメッセージ用バッファから読み出し，対応する処理を実行し，レスポンスをレスポンスメッセージ用バッファに書き込む．その後再び vmrpc_perform_kernel() を呼び出す．

(6) プロキシスレッド:
クライアントを起こし，自身はスリープする．

(7) クライアントスレッド:
レスポンスメッセージ用バッファを読み出し結果を受け取る．

これ以降は 3. から 8. を繰り返す．

5.3 Reverse VMRPC におけるゲストの実装 (サーバスレッド)

ゲストではサーバが常時稼働しており，全てのクライアントからの要求を単一のスレッドで処理する．Reverse VMRPC におけるサーバは，VMRPC のクライアントの 1 つとして実装されている．

図 8 にサーバスレッドの実装の概略を示す．これは，2 章で述べた VMRPC のクライアントである．このスレッドは，まず VMRPC のクライアントとしてバインディングを行う．次に Reverse VMRPC で用いるリクエストメッセージとレスポンスメッセージのための領域を確保する．確保した領域の番地を vmrpc_perform_kernel() でプロキシスレッドに渡す．この呼び出しは，ホスト OS からリクエストがなされるまでリターンしない．

ホスト OS がリクエストを行うと，サーバスレッドは無限ループに突入する．このループではまず，call_module() で図 5 に示したようなサーバのディスパッチ関数を呼び出す．その関数からリターンした時には，レスポンスメッセージが用意されている．最後に vmrpc_perform_kernel() を実行する．この実行は，レスポンスメッセージをホスト

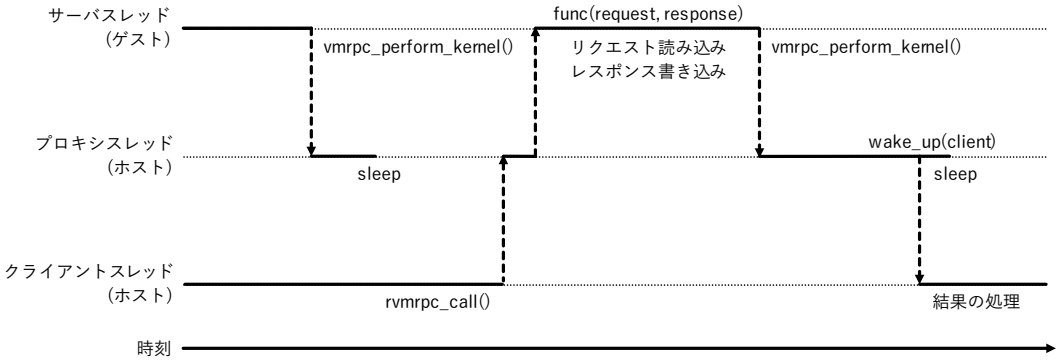

図 7 Reverse VMRPC の制御の流れ

```
void rvmrpcg_main(struct rvmrpc_info *info) {
    mhandle = vmrpc_bind_module_kernel(
        "MOD_RVMRPC", "1");
    buffer_request = __get_free_pages(
        GFP_KERNEL, PAGE_COUNT);
    buffer_response = __get_free_pages(
        GFP_KERNEL, PAGE_COUNT);
    info.request = rvmrpc_request;
    info.response = rvmrpc_response;
    info.request_page_count = PAGE_COUNT;
    info.response_page_count = PAGE_COUNT;
    vmrpc_perform_kernel(
        mhandle, INIT_HOST,0, 0);
    while (1) {
        ret = call_module(
            info.request, info.response);
        if (ret == EXIT_RVMRPC)
            break;
        vmrpc_perform_kernel(mhandle,
            RVMRPC_MAIN, &info,
            sizeof(struct rvmrpc_info));
    }
}
```

図 8 サーバスレッド (ゲスト) 実装の概略

OS にレスポンスメッセージを送ると同時にホスト OS からのリクエストメッセージを待つ.

5.4 Reverse VMRPC におけるホストの実装 (プロキシスレッド)

図 9 に，プロキシスレッドの実装の概略を示す．これは，2 章で述べた VMRPC のサーバである．このスレッドは，VMRPC のサーバとして proxy() という関数を実行する．

この関数は手続き番号 INIT_HOST で呼び出されたとき，ゲスト OS から Reverse VMRPC で用いるリクエストメッセージ用バッファとレスポンスメッセージ用バッファの番地を受け取る．この番地を init_host() という関数に渡し，ホスト OS のアドレス空間にマップする．以後このバッファは，以下に示すポインタ変数でアクセス可能になる．

```
int proxy(const rpck_data *rpcdata, int func,
    gva_t arg, size_t bytes) {
    int ret;
    switch (func) {
    case INIT_HOST:
        ret = init_host(rpcdata, arg, bytes);
        down(&sem_proxyy)
        break;
    case RVMRPC_MAIN:
        up(&sem_client);
        down(&sem_proxy);
        ret = 0;
        break;
    default:
        ret = 0;
    }
    return ret;
}
int init_host(const rpck_data *rpcdata, gva_t
     arg, size_t bytes) {
    struct rvmrpc_info info;
    vmrpck_copy_from_guest(rpcdata
        &info, arg, sizeof(info));
    pages_req = kmalloc(
        sizeof(struct page *)*info.req_npages,
        GFP_KERNEL);
    pages_res = kmalloc(
        sizeof(struct page *)*info.res_npages,
        GFP_KERNEL);
    _request = rvmrpc_allocate_pages(
        rpcdata, (gva_t)info.req,
        pages_req, info.req_npages);
    _response = rvmrpc_allocate_pages(
        rpcdata, (gva_t)info.res,
        pages_res, info.res_npages);
    return 0;
}
```

図 9 プロキシスレッド (ホスト) 実装の概略

_request ゲスト OS にあるリクエストメッセージ用バッ

```
int rvmrpc_call(
    u32 mod, u32 proc,
    u32 arglen, const void *arg,
    void *res, int res_buflen) {
    down(&lock);
    _request_->mod = mod;
    _request_->proc = proc;
    _request_->arglen = arglen;
    if (arg)
        memcpy(_request_->arg, arg, arglen);
    up(&sem_proxy);
    down(&sem_client);

    /* ----after back---- */
    if (response_->reslen > 0)
        memcpy(res, _response_->result,
            _response_->reslen);
    up(&lock);
    return 0;
}
```

図 10 rvmrpc_call() の実装の概略

ファが，ホスト OS のアドレス空間にマップされている番地を指す．ホスト OS でここにリクエストメッセージを書き込むと，ゲスト OS からアクセス可能になる．

_response ゲスト OS にあるレスポンスメッセージ用バッファが，ホスト OS のアドレス空間にマップされている番地を指す．ホスト OS でここを読み込むと，ゲスト OS から送られてきたレスポンスメッセージにアクセスできる．

init_host() からリターンすると，セマフォ down() でスリープする．次に手続き番号 RVMRPC_MAIN が呼ばれたき，このスレッドはセマフォ down() によりスリープし，クライアントスレッドからの要求を待つ．クライアントスレッドが rvmrpc_call() を呼び出すと，このスレッドは起床する．リクエストメッセージは，既に rvmrpc_call() により _request の指す先に用意されている．すると，この関数はリターンし，ゲスト OS に制御を移す．

次にこの関数が呼ばれたとき，_response の先にレスポンスメッセージが保存されている．すると，この関数はセマフォ up(&sem_client) でクライアントを起す．自身は再びセマフォ down() でスリープし，クライアントスレッドからの要求を待つ．

5.5 Reverse VMRPC におけるホストの実装 (クライアントスレッド)

rvmrpc_call() の実装の概略を図 10 に示す. この関数は，複数のクライアントスレッドから呼び出されることがある．しかし現在の Reverse VMRPC 実装では，リクエストメッセージとレスポンスメッセージを保持するバッファと，それを処理するスレッドは 1 組しかない．したがって，この関数は一度に高々 1 つの RPC しかなされないように，まずセマフォによりロックを行っている．

この関数は次に，ポインタ _request の先に，引数で与えられたリクエストメッセージをコピーしている．その次にセマフォの up() でプロキシスレッドを起こし，さらにその次にセマフォの down() でこのスレッド (クライアントスレッド) をスリープ状態にしている．

RPC が終わると，このスレッドが起きる．このとき，レスポンスメッセージはポインタ _response の先に保存されている．この関数は，このポインタの先にあるレスポンスメッセージを引数 res で与えられた場所にコピーしている．最後にロックを解除している．

6. proc ファイルシステムを通じた Linux ゲストの操作

本研究では Linux ゲストを操作するため，proc ファイルシステムを用いることにした．ゲスト OS を操作するためには，proc ファイルシステムを使わずに直接カーネル内の関数を呼び出したり変数をアクセスする方法もある．そうした方法と比較して，proc ファイルシステムを利用する方法の利点は，第 1 にカーネルの進化に追随することが容易であることが挙げられる．カーネル内の関数や変数の名前はしばしば変更されるのに対し，proc ファイルシステムの内容はあまり変化しない．Linux カーネル開発者はシステムコールのインターフェースや proc ファイルシステムの内容など，ユーザ空間のプログラムに影響を与えることを極力避けることが知られている．proc ファイルシステムにアクセスする Reverse VMRPC のサーバは，カーネルのバージョンによらず動作することが期待できる．proc ファイルシステムを使う第 2 の利点は，RPC サーバの開発が容易であることである．ユーザ空間のプログラムを記述する方法と同じロジックで RPC のサーバを記述することができる．proc ファイルシステムを用いる問題点は，直接カーネル内の関数を呼び出したり変数をアクセスする方法よりもオーバヘッドが大きいことである．

6.1 meminfo

ゲスト OS のワーキングセットサイズを見積もることは，仮想計算機の集約レートを高めるために課題である [1]．そのためには，ゲスト OS のメモリ利用状況を観測する必要がある．Linux は直接的にワーキングセットサイズを提供していないが，それを計算するために重要な情報，例えば Committed_AS を proc ファイルシステムの meminfo で提供している．Committed_AS は，全プロセスの匿名メモリページ (anonymous memory pages) の数である．本研究では，ゲスト OS の meminfo を，ホスト OS からアクセス可

```
struct meminfo_res {
    int len;
    char buf[BUFSIZE];
};
int read_meminfo_server(
    struct meminfo_res *res) {
    struct path init_path;
    struct file *file;
    int size;
    loff_t pos = 0;

    user_path("/proc/cpuinfo",
        &init_path);
    file = dentry_open(&init_path, O_RDONLY,
         current_cred());
    size = vfs_read(file, res->buf,
        BUFSIZE, &pos);
    res->buf[size] = '\0';
    res->len = size + 1;
    return 0;
}
```

図 11 meminfo を取得するサーバの概略

能にする．

図 11 に，ゲスト OS (Linux) の meminfo の内容をホスト OS に返す Reverse VMRPC のサーバの概略を示す．なお，このコードではエラー処理の記述を省略している．ゲスト OS がホスト OS から Reverse VMRPC で要求を受け取ると，関数 read_meminfo_server() が呼ばれる．この関数の引数は，結果を格納する共有メモリの先頭番地である．この関数は，ホスト OS からの RPC の引数を取らない．この関数はまず，カーネル内でファイルを開く API である dentry_open() を利用してファイル/proc/meminfo を開いている．次にその内容を vfs_read() で読み出し，Reverse VMRPC の結果を格納するメモリに保存し，ファイルを閉じている．最後に Reverse VMRPC の結果の長さを設定し，関数をリターンしている．

6.2 cpuinfo と loadavg

cpuinfo は CPU の情報を得るための proc ファイルシステムである．cpuinfo を vfs_read() で読み出すことで，CPU のモデル名やクロック数，キャッシュサイズなどの情報を得ることができる．また loadavg はロードアベレージ (Load Average) を得るための proc ファイルシステムである．cpuinfo と同じく vfs_read() で読み出すことで，過去 1 分間，5 分間，15 分間のロードアベレージを得ることができる．

これを読み出すための Reverse VMRPC プログラムは，6.1 節のプログラムで，参照する proc ファイルシステムのパスを/proc/cpuinfo および/proc/loadavg に置き換えるのみで実装することができる．

6.3 drop_caches

drop_caches は，メモリ内に残るファイルシステムキャッシュの削除を行う proc ファイルシステムである．ゲスト OS がホスト OS のファイルシステムを NFS や WFS[5] でマウントしている場合を考える．ホスト OS でファイルを追加したり削除したとしても，ゲスト OS にキャッシュが残っているために，なかなかそれが反映されないことがある．そこで本研究では，このような場合にゲスト OS のバッファキャッシュを削除する機能を Reverse VMRPC で実装した．

meminfo や cpuinfo では vfs_read() を用いるが，drop_caches では/proc/sys/vm/drop_caches に対し vfs_write() で書き込みを行う．

6.4 性能

Reverse VMRPC による proc ファイルシステム操作の性能を測定した．Reverse VMRPC によってゲスト OS の/proc/meminfo を開き，ホスト側のクライアントで値を得るまでの時間を計測した．

測定は以下の環境で行った．

- CPU: Intel Core i5-2450M 2.50GHz (2 cores)
- RAM: 4GB
- ホスト OS: Linux 3.12.10 (Ubuntu 16.04)
- ゲスト OS: Linux 3.12.10 (Ubuntu 14.04)

10 回の試行の結果得られた最短，最長，平均時間を表 1 に示す．また，同様の処理を SSH によって実行し，時間を測定した．これによって得られた平均実行時間は 587ms であった．

このように，Reverse VMRPC を使う方法では，proc ファイルを使ったとしても SSH を利用する方法と比較して非常に高速にゲスト OS の操作が可能なことがわかった．また Reverse VMRPC を使う方法は，ゲスト OS でネットワーク機能を無効化していても利用可能である．ゲスト OS を，インターネットからの攻撃に晒されないように安全に設定する手間も不要である．

7. FreeBSD ゲストの操作

Reverse VMRPC は OS に依存する技術ではなく，ホストとゲストで異なる OS が稼働している場合も利用可能である．本章ではホストで Linux，ゲストで FreeBSD が稼働している環境上で Reverse VMRPC を利用可能であることを示す．

表 1 meminfo の読み込みの実行時間

	最短	最長	平均
実行時間	0.45 ms	2.73 ms	1.03 ms

7.1 FreeBSD 用の Reverse VMRPC 実装

FreeBSD 用に実装された VMRPC ゲスト側モジュールが存在する．このモジュールを利用し，Reverse VMRPC を FreeBSD ゲストで稼働させる．

5 章で述べた Linux ゲスト用のサーバのソースコードに部分的な変更を加え，FreeBSD ゲスト用のサーバを実装した．変更点は主に，関数名やインクルードするヘッダファイルの置き換えである．例えば Linux のカーネルプログラムでは，任意長のメモリ空間を確保するために kmalloc() を用いるが，FreeBSD では malloc() 関数を用いる．またキャラクタデバイスの生成に，Linux は device_create() などを用いるのに対し FreeBSD は make_dev_p() などを用いる．

7.2 FreeBSD カーネルからのロードアベレージ取得

FreeBSD ゲストのロードアベレージを取得する Reverse VMRPC プログラムを作成した．ロードアベレージは単位時間あたりの実行待ちおよび実行中のスレッドの数を示す値で，システムの負荷状況を表す指標として用いられる．

図 12 に，FreeBSD ゲストで動作し，ロードアベレージを返す Reverse VMRPC のサーバの概略を示す．なお，このコードではエラー処理の記述を省略している．

Linux ゲストの場合とは異なり，このプログラムでは proc ファイルシステムを用いず，カーネル内のパラメータを直接参照している．このプログラムによって取得できるのは，過去 1 分間，5 分間，および 15 分間のロードアベレージである．

ゲスト OS がホスト OS から Reverse VMRPC で要求を受け取ると，関数 read_loadavg_server() が呼ばれる．この関数はホスト OS からの RPC の引数を取らない．この関数はカーネル内の構造体 averunnable のフィールド ldavg と fscale を，結果を格納するメモリにコピーしている．averunnable に含まれる配列 ldavg の 3 つの要素はそれぞれ過去 1 分間，5 分間，および 15 分間のロードアベレージを含んでいる．ただし配列に含まれる値は浮動小数で保存しているのではなく，fscale の倍数 (実現に用いたカーネルの構成では 2048) で保持している．

```
struct loadavg_res {
    long ldavg[3];
    long fscale;
};

static int read_loadavg_server(
    struct loadavg_res *res) {
    res->ldavg[0] = averunnable.ldavg[0];
    res->ldavg[1] = averunnable.ldavg[1];
    res->ldavg[2] = averunnable.ldavg[2];
    res->fscale = averunnable.fscale;
}
```

図 12 FreeBSD ゲストのロードアベレージを取得するサーバの概略

8. 評価

本研究の目的は，ネットワークを用いず高速かつ容易にホスト OS からゲスト OS に対し RPC を行う手法を提供し，ゲスト OS を操作することである．本研究では，これを既存の仕組み VMRPC 拡張する形で Reverse VMRPC として実現した．本研究では，分散システムにおける Long Polling と類似の仕組みを用いて VMRPC のサーバを Reverse VMRPC のクライアント，VMRPC のクライアントを Reverse VMRPC のサーバとして利用した．Reverse VMRPC は，全てを C 言語のコードで実装しており，VMRPC に関係するものを除けば，カーネルに標準で装備された API のみを用いて実装した．Reverse VMRPC は，ホスト OS として Linux，ゲスト OS として Linux および FreeBSD で動作している．これらのゲスト OS で，メモリの状態や CPU の使用率を調べる操作やファイルシステムのバッファキャッシュをフラッシュする操作を実現した．Linux における実験の結果，Reverse VMRPC はネットワークを用いる方法と比較して非常に高速であることが確認された．よって，本研究の目標は達成された．

現在の Reverse VMRPC の実装にはいくつかの制限事項がある．まず，ホスト OS からゲスト OS への RPC は，1 度に高々 1 つのクライアントしか行うことができない．複数のクライアントが RPC を行おうと試みると，ホスト OS 内で逐次化され，待機することになる．この制約は，ゲスト側のサーバですぐにリターンするような手続きを実行するようなアプリケーションでは大きな問題にはならないが，改善の余地がある．

Reverse VMRPC の実装では，ゲスト OS 内に RPC の引数と結果を保存するための領域を確保し，それをホスト OS のアドレス空間にマップした．この方法では，RPC を行う時に，引数と結果をコピーしなければならない．このことは，ホスト OS とゲスト OS で大きなデータをやりとりする時に，性能が低下する原因となる．今後は，このコピーを省略する方法を探っていきたい．

Reverse VMRPC を用いて，いくつかのゲスト OS の操作を行った．ただし，現状これらの操作は，ゲスト OS に依存したものになっている．たとえば，Linux ゲストで動作する meminfo は，FreeBSD ゲストでは動作しない．今後は，ゲスト OS の種類によらず，同じインタフェースで RPC の手続きを実装することで，この問題を解決する．

Reverse VMRPC により，ゲスト OS のカーネル内の関数を呼び出すことができるようになったので，今後，様々なアプリケーションを実装したいと考えている．たとえば，アウトソーシングの逆に，ゲスト OS のネットワークスタックやファイルシステムをホスト OS から利用できる

ようにしたいと考えている．

9. 関連研究

9.1 共生仮想化

共生仮想化 (Symbiotic Virtualization) という手法が Lange らによって提案されている [2]．これは，VMM とゲスト OS とを協調的に動作させる仮想化手法である．共生仮想化では，従来の完全仮想化と同様のハードウェア仮想化をゲストに提供するのに加え，ゲスト-ホスト間に共生仮想化インターフェースという新たなインターフェースを設ける．

共生仮想化インターフェースは，ホスト-ゲスト間の情報伝達を行うためのもので，SymCall は共生仮想化インターフェースの 1 つである．SymCall のインターフェースはシステムコールと近くなるように設計されており，ホストは関数呼び出しと同じ感覚でゲストに対する手続き呼び出しを行える．

また文献 [2] では，SymCall を利用した VMM 拡張機能として SwapBypass というサービスが実装されている．SwapBypass はゲスト OS のスワップ機構を改善するものである．ゲスト OS でスワップアウトされたページをスワップデバイスの代わりに VMM のメモリ空間に配置することで，スワッピングによる性能低下を低減する．

SymCall と比較して，本研究で実現した Reverse VMRPC の相違点は，VMRPC という既存の仕組みを拡張する形で設計されているという点にある．Reverse VMRPC は，全てを C 言語のコードで実装しており，VMRPC に関係するものを除けば，カーネルに標準で装備された API のみを用いた．また本研究では，実現した Reverse VMRPC を用いて，Linux ゲストおよび FreeBSD ゲストの操作を行った。

9.2 仮想計算機を用いたデバイスドライバの再利用

ある OS 用にビルドされたデバイスドライバを，VMM を用いて他の OS で使用する研究が LeVasseur らによって行われている [7]．この研究では，ゲスト OS のカーネルに中継プログラムを追加することで，ホスト OS からゲスト OS 上のデバイスドライバを使用できるようにしている．

ホスト OS からは VMM がデバイスドライバのように見えており，デバイスへの命令は VMM への命令に変換される．VMM はゲスト OS にその命令を転送し，ゲスト OS が実際のデバイス処理を実行する．ゲスト OS のデバイスドライバ自体には変更を加える必要がない．

LeVasseur らの研究は仮想化環境として L4 マイクロカーネルを用いているが，本研究では Linux KVM を用いた．

9.3 協調型仮想計算機モニタ

協調型仮想計算機モニタは，従来の仮想計算機モニタの隔離という目的を緩和し，同一ハードウェアで実行されるホスト OS とゲスト OS が協調して動作することを支援する [8]．このシステムでは OS の境界を超えて動作する Cross-OS Program (COP) が動作する．COP はユーザ空間で動作し，ホストとゲストの間で RPC で通信を行う．この RPC はアウトソーシングの機能を利用し，ホスト OS の Unix Domain Socket と SunRPC を使って実装されている．

この研究と本研究の共通点は，ホスト OS とゲスト OS が RPC で通信できる点，および実装に VMRPC が使われていることである．この研究と本研究の相違点は，Reverse VMRPC がカーネル空間で動作し，高速である点にある．

10. おわりに

本研究では，ホスト型 VMM においてゲスト OS の操作を行うための Reverse VMRPC という手法を提案・実装した．Reverse VMRPC の実装には既存の仕組みである VMRPC で提供される API を用いた．この実装方針によって，実装を簡略化することができた．また，VMLP によってホスト OS からゲスト OS の手続きを呼び出すことの困難な点を解決した．また，ゲスト OS を操作する例として，Linux ゲストの proc ファイルシステムを操作可能なことを示した．これに加え，FreeBSD ゲストを操作することができた．

今後は複数クライアントへの対応や，ゲスト OS のネットワークスタックやファイルシステムをホスト OS から利用することを目指す．

参考文献

[1] Jui-hao Chiang, Tzi-Cker Chiueh and Han-Lin Li: Memory Reclamation and Compression Using Accurate Working Set Size Estimation. In *Proceedings of 2015 IEEE 8th International Conference on Cloud Computing*, pp.187-194 (2015).

[2] John R. Lange and Peter a Dinda: SymCall: Symbiotic Virtualization Through VMM-to-Guest Upcalls. In *The 7th ACM SIGPLAN/SIGOPS International Cognference on Virtual Execution Environments (VEE 2011)*, pp.193-204 (2011).

[3] 齋藤剛, 新城靖, 榮樂英樹, 佐藤聡, 中井央, 板野肯三: 仮想計算機におけるアウトソーシングのためのゲスト-ホスト間 RPC. 情報処理学会第 20 回コンピュータシステム・シンポジウム (ComSys2008) ポスターモセッション (2008).

[4] Hideki Eiraku, Yasushi Shinjo, Calton Pu, Younggyun Koh and Kazuhiko Kato: Fast Networking with Socket-Outsourcing in Hosted Virtual Machine Environments. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC '09)*, pp.310-317 (2009).

[5] 豊岡拓, 新城靖, 齋藤剛: ホスト型仮想計算機環境におけるファイル入出力の VFS アウトソーシングによる高速化. 情報処理学会第 21 回コンピュータシステム・シンポジウム (ComSys2009), pp.33-40 (2009).

[6] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin and Anthony Liguori: kvm: the Linux virtual machine moni-

tor. In *Proceedings of the Linux Symposium*, pp.225-230 (2007).

[7] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess and Stefan Götz: Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI '04)*, pp.17-30 (2004).

[8] 白石光隆, 新城靖, 齋藤剛, 豊岡拓, 五明将幸: 協調型仮想計算機モニタとその Linux KVM における実装. 情報処理学会論文誌 コンピューティングシステム, Vol.3, No.2 (ACS 30), pp.147-162 (2010).