Vol.2014-ARC-210 No.9
Vol.2014-OS-129 No.9
2014/5/15

仮想マシン環境における応答時間変動原因の分析

小比賀亮仁[†1†2] 篠田陽一[†1] 菅原智義[†2]

我々は，仮想マシンとして実行される通信サービス(NFV)の品質向上に取り組んでいる．通信サービスを仮想マシン上で実行すると，応答時間が急激に変動し，その結果として通信サービスの品質劣化をもたらす．我々は，KVM 上で実行される SIP アプリケーションを通信サービスの具体例とし，特に Hypervisor の挙動に着目して応答時間変動原因を分析した．その結果，Hypervisor に応答時間の変動を引き起こす構造上の問題があることを特定した．本稿では，Hypervisor 上の応答時間の変動原因の分析結果を示すとともに，仮想マシン環境において，応答時間の変動を解消するための機能を議論する．

Analysis of Response Time Fluctuation Factors on Network Function Virtualization

AKIHITO KOHIGA[†1†2] YOICHI SHINODA[†1]
TOMOYOSHI SUGAWARA[†2]

We are working on improvement of telecommunication quality on Network Function Virtualization (NFV). It is known that a telecommunication service running on a virtual machine sometimes has the fluctuation of round trip time. This fluctuation leads to loss of quality on telecommunication service. We investigated what causes the fluctuation of round trip time, based on Asterisk SIP server on KVM Hypervisor as an example of NFV instance. We found the fact that KVM Hypervisor has some structural problems related to stacking packets that cause the fluctuation of round trip time. In this paper we explain the result of this investigation and argue what function we need for NFV era.

1. 背景

近年, Network Function Virtualization（以下, NFV と呼称）と呼ばれる通信サービスの仮想化が注目されている．携帯電話を他の携帯電話と接続するための電話交換機網を EPC（Evolved Packet Core）と呼ぶ．また，電話交換機網の中で運用される通話を中心とした通信サービスを IMS（IP Multimedia Subsystem）と呼び，IMS，EPC を総称して Network Function と呼ぶ. Network Function を仮想計算機上で実行することを NFV と呼ぶ.

通信サービスを仮想化することによって，１．通信サービス処理に用いられるハードウェアコストの削減，２．急激な通信リクエスト負荷の上昇に対する柔軟な対応が可能となる. 東日本大震災では通常の 60 倍の通話要求が発生し, 電話交換機網が輻輳状態となり, 疎通率が著しく低下した. 我々は電話交換機網の輻輳状態を緩和すべく，EPC，IMS を仮想計算機上で実行し，急激な負荷上昇に対しても柔軟な処理容量の増強によって輻輳状態が緩和されることを示した 1).

通信サービスを仮想化すると，応答時間が激しく変動することが知られている 2). 通信サービスの応答時間が変動すると，通信サービスの品質劣化を引き起こす．よって，できるだけ応答時間を一定に保つ必要がある．通信サービスの具体例として通話サービスを挙げる．通話サービスは二つの部分から構成される．一つは制御通信フェーズであり，もう一つはデータ通信フェーズである．制御通信フェーズでは，携帯電話間のセッションを作成する．電話番号を入力してから呼び出し音が鳴るまでがこのフェーズにあたる. データ通信フェーズでは, 音声データを送受信する. 携帯電話で会話をしている間がこのフェーズにあたる．制御通信フェーズにおける通信サービスの品質劣化とは，電話番号を入力しても一向に呼び出し音が鳴らずに待たされる状態のことを指し，データ通信フェーズにおける品質劣化とは，音声にガウスノイズが混ざったり，音声が細切れになったりすることを指す.

2. 課題

仮想マシン上で通信サービスを実行すると, 応答時間が激しく変動する．仮想マシン環境の応答性向上に関する研究は数多く存在する 3) 4) 5) 6)が, 既存研究では明らかにされていない. 文献 2)は TCP リクエストに対する応答時間の変化を問題としている．このような応答時間の変化は UDP リクエストを基本とする通信サービスでも同様に起こると考え，文献 1)の研究において応答時間の変化を測定した．その結果，ある一定の負荷を超えるとリクエストに対する応答時間の激しい変動を確認した.

本稿では, 仮想マシン上で実行される通信サービスの具体例として SIP サーバを対象として，応答時間の変動を分析し原因を特定する．東日本大震災では主に制御通信フェー

†1 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology
†2 日本電気株式会社
NEC Corporation

ズにおける輻輳が問題となった．輻輳が発生したのはシステムの許容量を超えてリクエストが到着したことが主原因である．しかしながら，輻輳は，ある一定の負荷を超えるとCPU使用率が100%を超えなくても輻輳が発生する．このような輻輳発生原因の一つに応答時間の激しい変動があると考えられる．つまり，応答時間が変動し，偶然，応答時間がタイムアウトに達した通話リクエストはキャンセルされる．そうすると，ユーザは再度通話を試みる．このようなキャンセルと再試行の繰り返しによって単位時間当たりのリクエスト数は指数的に増加すると考えられる．応答時間を一定に保つことによって，突発的な負荷上昇に対しても,輻輳耐性の高い通信サービスを提供できると考える．

本稿では実際に発生した輻輳事例を勘案し,制御通信フェーズの応答時間を分析する．制御通信フェーズの代表的なアプリケーションはSIP（Session Initiation Protocol）サーバであるので，本稿では，アプリケーションへのプローブ挿入も考慮し，オープンソースのサーバであるAsteriskを対象アプリケーションとして用いた．

3. 分析内容

分析は次の手順に従う．

1. SIPサーバに対する負荷特性の調査
2. 事前評価によるAsteriskに対する負荷の決定
3. 応答時間変動原因の分析
   (ア) 応答時間変動に影響する機能ブロックの特定
   (イ) 当該機能ブロックが影響する原因の特定
   (ウ) 影響を取り除くための改善方法実装・評価
   (エ) 取り除くことができなかった応答時間変動原因の追加調査

最初に，分析を行う際にどの程度の負荷が分析に適しているかを決定する必要があるので,SIPプロトコル(Asterisk)の持つ負荷に対する性質について調査した．負荷が低すぎると対象アプリケーションに現れる変化が微小で改善施策を施しても，どの程度改善されたかがわかりにくくなる．逆に負荷が高すぎると，対象とするシステムの許容量を超えてしまい，対象アプリケーションに現れる現象自体をとらえることができなくなる．よってSIPプロトコルの持つ負荷に対する性質について調査することによりどの程度の負荷を与えればよいかが決まる．ここで決定した負荷を使って，いくつかの改善施策の試行による効果を比較するために事前評価を行った．

次に，応答時間変動の原因を分析した．応答時間変動原因の分析は次のような工程である．最初は，個々のリクエスト処理において，応答時間の変動と追従して処理時間の変動する機能ブロックが応答時間の変動に大きな影響を与えていると推定して，各リクエストにかかる処理時間について，機能ブロック毎の処理時間の割合を算出した．機能ブロックとは，割り込み処理，UDP,IPなどのプロトコル処理など，一連の処理のうち，大きな区切りとなる区分のことを指す．次に，当該機能ブロックの処理時間がなぜ変動するのかを分析して問題点を抽出した．次に，問題点に対して改善方法を簡易実装することによって前記分析が正しいかどうかを判断した．改善方法の実装・評価で改善できなかった部分の変動についても分析を行って問題点を抽出した．

分析の結果，応答時間の変動は，**パケットの滞留とAsterisk の構造上の問題が原因**であると特定した．アプリケーションの構造上の問題はアプリケーション個々の実装方法の問題であるので，本稿では深く取り上げない．本稿最後の議論では，Hypervisor の構造上の問題について取り上げる．

3.1 実験環境

実験環境としては，単純構成な NFV の具体例としてKernel-based Virtual Machine(以下 KVM)を仮想マシン環境として選択し，ターゲットアプリケーションとしてフリーの SIP サーバである Asterisk を選択した．また，Asterisk への負荷発生用プログラムとして sipp を用いた．

表 1 に負荷発生用プログラムの実行環境を示す．また，表 2 に Asterisk が動作するサーバ環境を示す．

表 1 負荷発生用クライアント環境

OS	CentOS6.3 (kernel-2.6.32-279.22.1.el6.x86_64)
負荷発生用プログラム	Sipp-v3.3
負荷シナリオ	UAC(User Agent Client)
メモリ	16GB
ネットワークカード	Broadcom10GNIC

表 2 サーバ環境

ゲスト OS	OS	CentOS6.3 (kernel-3.12.6)
	アプリケーション	Asterisk-11.0.1
	メモリ	8GB
	ネットワークドライバ	virtio_net
Hypervisor	CPU	Xeon E5-2690 2.9Ghz
	メモリ	32GB
	ネットワークカード	Broadcom10GNIC
	OS	CentOS6.3 (kernel-3.12.6)
	ネットワーク仮想化機構	bridge, tun, vhost_net
	ネットワークドライバ	bnx2x

負荷シナリオは sipp に標準の UAC(User Agent Client)シナリオを用いた．本稿の目的は仮想マシン環境上の応答時間変動を対象としているので，ネットワークスイッチによる応答時間への影響を防ぐため負荷発生用クライアントとAsterisk を実行している物理サーバは，ネットワークスイ

ッチを経由せず直接接続されている．

3.2 Round Trip Time（RTT）計測の詳細

3.2.1 SIP プロトコル

SIP(Session Initiation Protocol)プロトコルは，通話を開始する際のプロトコルとして用いられる．

図 1 に SIP プロトコルの処理シーケンス(UAC)を示す．

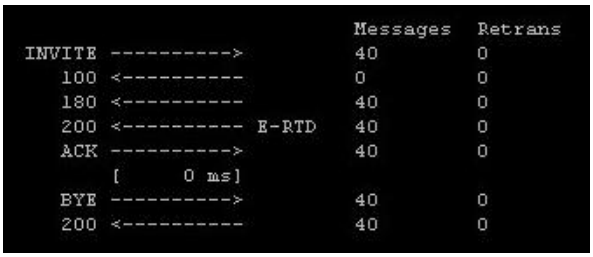

図 1SIP シーケンス（UAC）

図中，矢印左側がユーザ側（sipp）であり，右側がサーバ側（Asterisk）である．sipp が Round Trip Time（以下，RTT）として計測する区間は，最初の INVITE リクエストから，最後の BYE が送信されて，200OK が返信されるまでの区間であり，INVITE から 200OK が返ってくるまでの間に送受信されるリクエスト数は 6 つ存在する．これらのリクエスト（もしくは応答）の送受信はすべて Asterisk の SIP 処理のために設けられた専用スレッドが実行する．

UAC シナリオを使った通話テストでは，着信側に相当する端末は存在しない．INVITE 処理の終了を示す ACK（図中上から 5 番目の矢印）後はユーザの通話時間（前述のデータ通信フェーズ）となるが，本稿では制御通信フェーズが分析の対象となるため，通話時間は 0 秒（図中の[0ms]）とした．

図 2 に SIP ヘッダのデータ構造を示す．

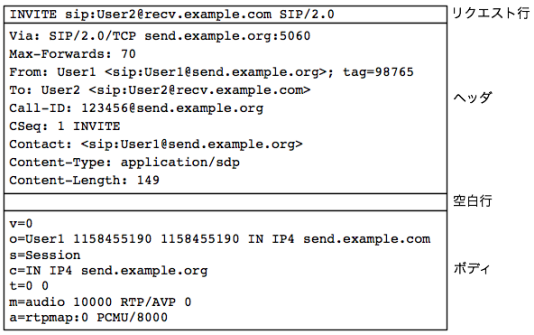

図 2 SIP ヘッダ

SIP ヘッダは Layer5 以上のテキストフォーマットとして表現される．SIP サーバは，ここに記載されている情報を元にユーザ間のセッションを管理する．図中，Call-ID は図 1 に示した一連のシーケンスを通して付与される ID であり，CSeq は個々のリクエスト（100Trying や 180Ringing など）に一つ付与される ID である．後述の分析では Call-ID と Cseq を使って個々の SIP リクエストについて，処理に費やされる時間を分析する．

3.2.2 SIP プロトコルの負荷に対する特徴

Asterisk に与える妥当な負荷を決めるために，SIP プロトコル（Asterisk）の持つ負荷に対する性質について調査する．調査は，低負荷・中負荷・高負荷の 3 段階の負荷をかけて，RTT のグラフを確認し，Asterisk の負荷に対する特性を確認する．

低負荷：図 3 に，sipp から 100 call/s の sip リクエストを送信した際の応答時間の変化を示す．図中，平均応答時間は約 2ms，最大応答時間は約 7.5ms となっている．図中に現れる応答時間のばらつきは，負荷を上げていくと数が増えていく．

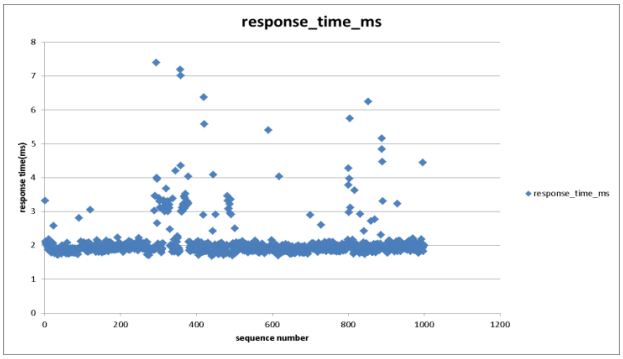

図 3 低負荷に対する応答時間の変動

中負荷：図 4 に，sipp から 500call/s の sip リクエストを送信した際の応答時間の変化を示す．低負荷において，時間のばらつきがグラフ中にまばらに表れることを説明した．低負荷状態からさらに負荷を上げていくと，のこぎりの歯のようなグラフ形状になり，そこからさらに負荷を上げると図のような形状になる．このグラフでは応答時間が最大で約 380ms となるが，INVITE リクエストのタイムアウト時間である 500ms は超えていないので，再送は発生していない．

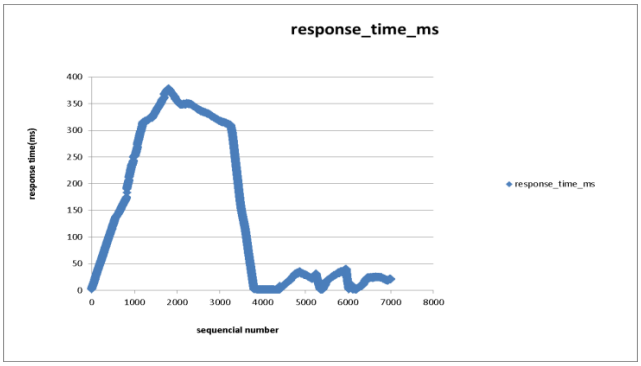

図 4 中程度の負荷に対する応答時間の変動

高負荷：図 5 に 700call/s の sip リクエストを送信した際の応答時間の変化を示す．図中，500ms までは RTT が線形に上昇し，500ms を超えた時に負荷上昇の傾きが 1.5 倍程度に変化する．500ms は SIP の最初の再送タイマー(T1)のデフォルト値である．500ms を超えても返信のないリクエストに対して，sip クライアントは再送の INVITE リクエストを送信する．この再送リクエストがさらに sip サーバの

処理容量を圧迫することによって，このような RTT の急激な変化が現れることになる．図 5 の状態では，sip サーバの CPU 使用率は 100%で推移し，大量のリクエストをドロップする．図 5 のグラフはドロップされずに RTT を計測できたリクエストだけが表示されている．

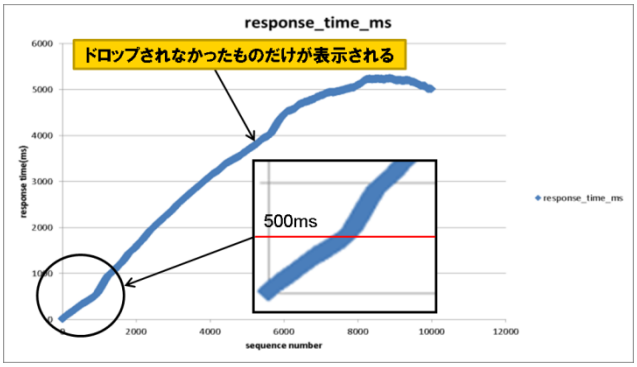

図 5 高付加に対する応答時間の変動

以上の結果から，Asterisk に送信する sip リクエストは毎秒 400～500call/s 程度が妥当であると判断した．

3.3 事前評価

図 6 に事前評価結果を示す．図 6 は sipp から 400call/s の負荷をかけた際の RTT の推移を示している．RTT が最も高い部分で約 43ms であり，最も低い部分で約 1ms である．これを元となるデータとして，次節以降でこの応答時間の変化の原因を分析する．

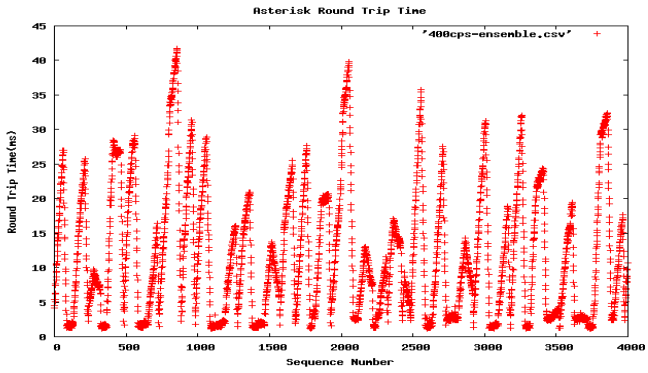

図 6 400cps の負荷に対する RTT の変化

4. 分析結果

4.1 処理時間の変動が大きい機能ブロックの特定

まず，個々のリクエスト処理において，応答時間の変動と追従して処理時間の変動する機能ブロックが応答時間の変動に大きな影響を与えていると推定して，各リクエストにかかる処理時間の中で機能ブロック毎の処理時間の割合を算出した．

調査内容としては，リクエストがハイパーバイザー，ならびにゲスト OS 上の各処理に到達した時刻を記録し，それぞれの時刻の差分を計算することによって，どの処理で最も時間がかかっているかを調査した．時刻記録のために プローブ関数を挿入した場所は，次の通りである．

- Hypervisor にリクエストが到着した時刻
- vhost_net が処理を開始した時刻
- GuestOS がプロトコル処理を開始した時刻
- GuestOS がプロトコル処理を終了した時刻
- Asterisk がリクエストを受け取った時刻
- Asterisk がリクエストに対する応答を送信する時刻

図 7 は上記の時刻記録箇所を図示したものである．systemtap のプローブ関数は，当該 SIP リクエストのプローブ時刻，プローブされた関数の関数名，Call-ID+CSeq の組み合わせを記録する．

到着時刻を記録する際の注意点は以下の通りである．時刻は TimeStampCounter（TSC）の値を用いた．通常，GuestOS の保持する TSC は Hypervisor の保持する TSC にオフセットとなる値を足したものになっており，GuestOS と Hypervisor の各々でそのまま TSC の値を取得すると時刻がずれる．よって，GuestOS 起動時に Hypervisor 上の vmx_write_tsc_offset 関数の offset 値に 0 を代入することで TSC のオフセットを 0 に設定した．Call-ID と CSeq の組み合わせは sipp から送信する際に IP ヘッダのオプションフィールドに格納した．理由は，systemtap によるプローブ処理をできるだけ軽くするためである．SIP プロトコルは Layer5 以上のテキストデータとして表現されている．よって，Call-ID と CSeq を SIP ヘッダから読み取るには，SIP プロトコルのテキスト処理が必要になる．一方で，予めこれらを IP ヘッダのオプションとして保持しておけば，ヘッダ上の定位置からデータをコピーするだけなので，テキスト処理と比較して処理が軽くなる．また，プローブを挿入する関数もできるだけ少なくする必要がある．プローブの挿入個数が多いと，グラフの波形が変化したり，systemtap がいずれかの理由でプローブをスキップしたりする場合がある．本稿の分析環境では，グラフ波形の変化やプローブのスキップの起こらない程度のプローブ挿入可能個数は GuestOS，Hypervisor それぞれ約 3～4 個程度であった．

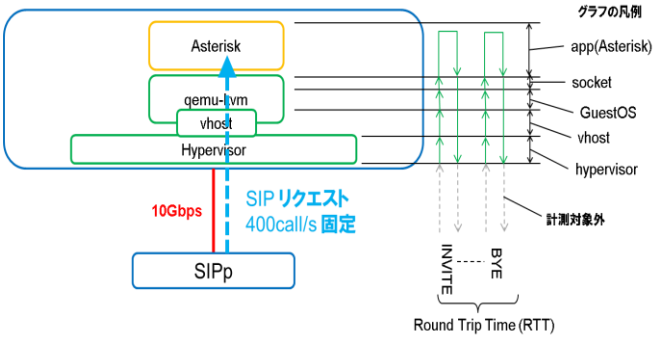

図 7 計測箇所とグラフの凡例

調査の結果，GuestOS 上で UDP 処理が終了した後，当該パケットがユーザプログラム（Asterisk）に渡されるまでの待ち時間で最も時間を費やしていることがわかった．図 8 は各機能ブロックが使用する実行時間の推移を示している．Y 軸が TSC の値（クロックサイクル）を表しており，X 軸

は処理したリクエストのシーケンス番号である．図中，応答時間の変動に最も影響を及ぼしているのが socket の時間であることがわかる．socket は UDP 処理が終了してから Asterisk が当該パケットを受け取るまでの時間であることから，GuestOS 上で UDP 処理が終了した後，Asterisk が SIP 処理を開始する前に長い間待たされていることが原因で応答時間が激しく変動していることがわかった．

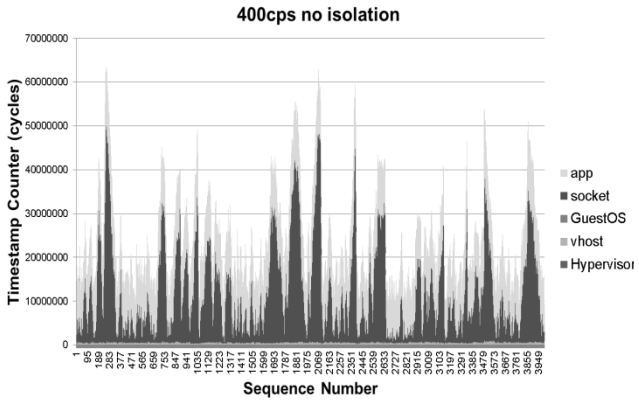

図 8 各機能ブロックが使用する実行時間の推移

UDP 処理が終了した後に長時間待たされる原因として，ユーザプログラムの実行遅延が推測される．Hypervisor から送られてきた SIP リクエストは，GuestOS 上の割り込み処理とプロトコル処理用カーネルスレッドを経由してユーザプログラムのパケットバッファに格納される．割り込み処理とプロトコル処理用カーネルスレッドは同一仮想 CPU 上で動作し，かつ最優先で実行される．プロトコルが処理されたパケットがソケットバッファに格納された後は，遅延なしに try_to_wake_up 関数でパケット受信先のプロセスが起動される．しかし，仮想 CPU は Hypervisor 上のユーザスレッドとして動作するので，仮想 CPU に対して物理 CPU 時間が割り当てられるまでに一定の遅延時間が発生する．この理由によってユーザプログラムの実行開始が遅延しているものと推測する．

4.2 ユーザプログラムの待機時間増加の原因分析

UDP 処理が終了した後，当該パケットがユーザプログラム（Asterisk）に渡されるまでの待ち時間が長くなる原因を調査する．待ち時間が長くなる原因としては，物理 CPU 時間が仮想 CPU に割り当てられていないと推測した．よって，SIP リクエスト処理中に仮想 CPU（以下，vCPU スレッドと呼称）と物理 CPU が割り当てられていない箇所を調査した．

まず，物理 CPU 時間が割り当てられていない箇所を kernelshark で確認したところ，SIP リクエスト処理中に数多くの箇所で Hypervisor の idle プロセスが実行されていることがわかった．idle プロセスは vCPU スレッドから schedule 関数によって切り替えられている．

次に，vCPU スレッドの実行が idle プロセスの実行に切り替えられている原因を調査した．その結果，Hypervisor が GuestOS 上の HLT 命令をトラップして VM_EXIT 処理を実行し，vCPU スレッドから idle プロセスの実行に処理が切り替わっていることがわかった．また，GuestOS 上の HLT 命令の実行は，Asterisk による polling 処理において，polling イベントが一定時間到着していなかったことに起因する．

原因分析結果をまとめると，ユーザプログラムの待機時間が増加する際には，次のような遷移が起こる．

- GuestOS 上の遷移
1. Asterisk の polling 処理がタイムアウトによって idle プロセスに切り替わる
2. idle プロセスは HLT 命令を発行する
- Hypervisor 上の遷移
1. HLT 命令がトラップされ，Hypervisor によって VM_EXIT が処理される（その後，vCPU スレッドを実行する）
2. vCPU スレッドは schedule 関数を実行して idle プロセスに物理 CPU を明け渡す

GuestOS 上の polling 処理において，イベントが無ければ HLT 命令が発行されるのは OS の一般的な処理手順である．しかし，Hypervisor による HLT 命令のトラップは，HLT 命令の性質を考慮していないタスクスケジューリングに置き換えられるため，これがユーザプログラムの待機時間増加の原因になると考えられる．HLT 命令は，物理 CPU を停止させるハードウェア命令である．停止状態からの回復時間は，OS のタスクスケジューリング時間と比較するとほんのわずかな時間である．よって，Hypervisor が vCPU スレッドと idle プロセスのタスクスケジューリングを実行している間に SIP リクエストが滞留していくものと考える．

4.3 vCPU スレッドの物理 CPU 明け渡し抑制実験

調査の結果，GuestOS 上で発行された HLT 命令を Hypervisor がトラップし，vCPU スレッドが自身の持つ物理 CPU を他のスレッドに明け渡すことによって他のスレッド実行中に到着パケットが蓄積し，RTT が増大することが分かった．

これを立証するために，HLT 命令をトラップした vCPU スレッドが CPU を明け渡さなければどうなるのかを実験した．HLT 命令をトラップした vCPU スレッドは，kvm_vcpu_block 関数において vCPU スレッドを wait キューに登録して，schedule 関数を呼び出して自身の持つ物理 CPU を明け渡す．kvm_vcpu_block 関数内の条件分岐によって当該条件が成立したときに，vCPU スレッドは wait 状態から復帰する．この schedule 関数の呼び出しと条件分岐は無限ループ内で実行される．よって，この schedule 関数をコメントアウトすることによって，条件が成立するまで kvm_vcpu_block 関数は busy loop を実行することになる．

図 9 は HLT 命令を受け取った vCPU スレッドが busy loop

を実行した際の応答時間の変化を示している．図のグラフは，Y 軸が応答時間を表し，X 軸がリクエストのシーケンス番号を表している．応答時間の最大値は，約 17ms であり，SIP リクエストの実行開始直後に現れる．平均応答時間は，約 1ms となった．標準偏差は 0.13 となった．グラフが示す通り，ほとんどの応答時間の変化がなくなっていることがわかる．ただし，vCPU スレッドが実行されている物理 CPU の CPU 使用率は常に 100%で推移する．

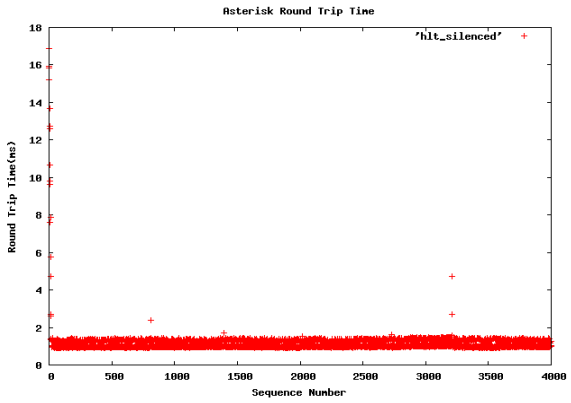

図 9 vCPU スレッドを busy loop させた場合の RTT の変化

図中，SIP リクエスト処理の開始直後に大きな応答時間の変化が残っていることがわかる．これをスパイク上のブレと呼ぶこととする．次節ではこのスパイク上のブレの原因を調査する．一方で，スレッドの切り替え抑制によって終息したブレを定常的なブレと呼ぶこととする．

4.4 スパイク状のブレの原因特定

基本的に，応答時間の遅れは SIP リクエストが Hypervisor か GuestOS 上のどこかに滞留することによって発生する．3.6 節の vCPU スレッドの物理 CPU 明け渡し抑制実験の結果からもわかるように，vCPU スレッドが常に動き続けて SIP リクエストを処理すれば応答時間は変動しない．vCPU スレッドが常に動き続けているにも関わらず応答時間が変動するということは，vCPU スレッドに到着する前にすでにパケットが滞留しているものと推測できる．

よって，スパイク状のブレの原因特定では，最初に SIP リクエストがどこで滞留しているのかを調査した．SIP リクエストが Hypervisor 上で滞留すると予想される箇所としては，次の 2 つが考えられる．

1. 仮想マシン宛てのパケットを受け取る vhost_net
2. GuestOS が用意している virtio のリングバッファ

まず，vhost_net に滞留するパケット量を調べる．vhost_net は handle_rx 関数の中で tun ドライバの tun_recvmsg 関数を用いて tun ドライバのキューからパケットを取り出して，GuestOS 上の virtio が提供するリングバッファ（vring）に当該パケットをコピーする．したがって，vhost_net が tun_recvmsg 関数を実行する際にすでに蓄積されているパケット数をカウントすることによって，パケットの到着がどのように推移するのかを調査した．

図 10 は tun ドライバ上に蓄積されているパケット量の推移を示すグラフである．Y 軸は vhost_net がパケットを取り出す際，すでに蓄積されているパケットの量を示している．X 軸は tun_recvmsg を実行した回数である．グラフでは，SIP リクエストの処理開始時点で多くのパケットが蓄積されていることを確認した．滞留しているパケットの最大値は高々5 個であるが，SIP リクエストの開始時点では滞留数が 4〜5 個の状態が続いて次第に滞留数が少なくなっていく．

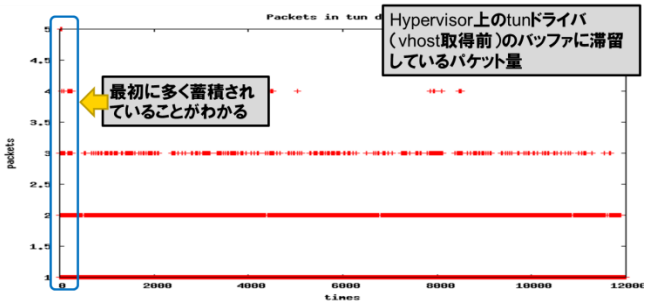

図 10 Hypervisor 上に蓄積しているパケット量

次に GuestOS 上で virtio が用意しているリングバッファに滞留しているパケット量を調べる．ソケットバッファ構造体のキューからパケットを取り出す際に実行される関数は netif_receive_skb 関数である．ftrace は，netif_receive_skb 関数の実行イベントを記録できるので，netif_receive_skb 関数が実行された時刻を記録し，当該関数実行の時間間隔を見ることによって，どの程度パケットが滞留しているのかを調査した．

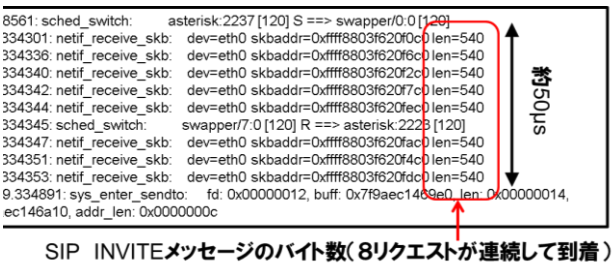

図 11 ftrace によるパケット滞留の確認

図 11 に ftrace による netif_receive_skb 関数のトレース結果の一部を示す．トレース結果は，左から実行時刻，関数名，トレース情報となっている．netif_receive_skb 関数の場合のトレース情報は，デバイス名，取得したソケット構造体のアドレス，パケットの長さから構成される．図中，パケットの長さが 540 バイトのパケットは SIP の INVITE リクエストを示している．図を見ると，約 50μ秒の間に 8 つの INVITE リクエストが到着していることがわかる．このような INVITE リクエストの塊は SIP リクエストの処理開始時点で少なくとも 2 個確認できた．つまり，16 個の SIP リクエストが束になって到着していることになる．したがって，virtio のリングバッファに滞留している SIP リクエストの数は，Hypervisor に滞留しているリクエストの数よりもはるかに多い．

virtio のリングバッファに滞留している SIP リクエストの数が，Hypervisor に滞留しているリクエストの数よりもはるかに多い理由としては，SIP リクエストが経由するスレッドの処理遅延が推測できる．ここで，Hypervisor 上の各スレッドの起動時刻を kernelshark にて確認した．

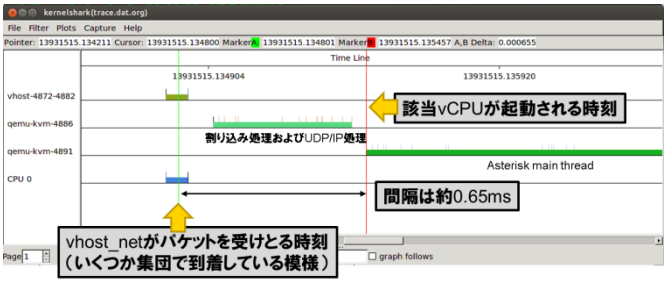

図 12 各スレッドの起動時刻の遷移

図 12 に SIP リクエストに関連する各スレッドの起動時刻の遷移を示す．図に示す通り，SIP リクエストに関連する Hypervisor 上のスレッドは 3 つ存在する．1 つは vhost_net ドライバのカーネルスレッドである．2 つ目は，GuestOS 上の割り込み処理とプロトコル処理を行う vCPU スレッド，3 つ目は，Asterisk が実行されている vCPU スレッドである．図を見ると，vhost_net がパケットを受け取ってから，Asterisk が実行を開始するまでに約 0.65ms かかることがわかる．Asterisk が実行を開始してすぐにパケットの受信が開始されるわけではないので，実際のパケット受信処理はもっと後に行われる．vhost_net でリクエストを取得する時点ではすでに SIP リクエストが数個滞留している状態であり，この起動遅延によってリクエストの滞留個数が増加するものと考える．このような起動遅延は，SIP リクエストの処理中に何度も現れる．よって，結果的に 10～20 個の SIP リクエストを一斉に処理しなければならないという状況になる．

起動遅延を無くすことによって，応答時間の変動が小さくなることは，4.3 節の実験によっても明らかであるが，スレッドの起動遅延とパケットの滞留量については曖昧さが残るので，さらなる調査が必要である．

一方で，滞留したパケットが Asterisk 上でどのように処理されるかを調査した．図 13 は Asterisk が実行される GuestOS 上の仮想 CPU に着目して，当該仮想 CPU の CPU 割り当てを kernelshark にて図示したものである．

図の黒色部分は Asterisk の SIP リクエスト受信スレッドの実行を表しており，黒色以外は，着信側クライアントとの送受信用スレッドの実行時間を表している．Asterisk は SIP リクエストを受け取ると，子スレッドを生成して着信側クライアントとの送受信に用いる．今回の実験では，着信側クライアントは存在しないので，当該子スレッドは何もせずに処理を終了する．図に示される通り，スパイク状の RTT が現れるまでの間，多くの部分が子スレッドの実行に割り当てられていることがわかる．よって，Asterisk の持つ仮想 CPU 時間は，SIP リクエストを処理と並行して，子スレッドの生成と実行を含むことがわかる．スレッドは pthread_create で生成されるため，一度生成されるとキャッシュによって，次のスレッドは生成時間が短くなる．よって，スレッドの生成と実行がある程度続くと図中の黒色で示された仮想 CPU 時間のうち SIP リクエスト処理に使える時間が徐々に長くなる．

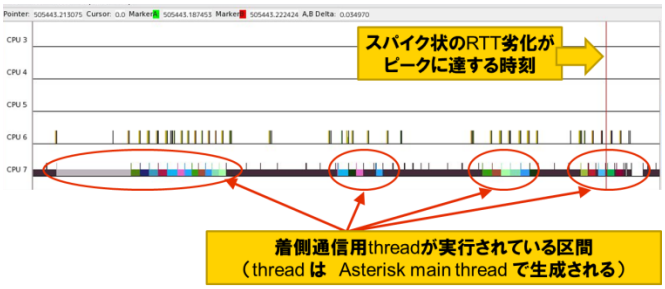

図 13GuestOS 上のスレッド実行時間の経過

図 13 の結果から，スパイク状の RTT が SIP リクエスト開始直後に現れる理由は，パケットが最も滞留しやすい時間帯であることと，pthread_create 実行時にキャッシュが形成されていないことによって，子スレッド生成に時間がかかることに起因することがわかった．

4.4 節の分析をまとめると，スパイク状のブレの原因は，1．スレッドの起動遅延によるパケットの滞留，2．Asterisk の構造上の問題であると考える．

5. 仮想マシン環境におけるリアルタイム性向上のための技術課題

我々は，4 節の分析結果から，リクエスト応答時間の変動は，**パケットの滞留と Asterisk の構造上の問題が原因**となって発生すると結論づける．パケットの滞留原因は Hypervisor の構造上の問題であり，大きくは以下の 2 つに集約されると考える．

1. 仮想マシンの idle 状態の振る舞い
2. 遅延でできるだけ小さくする仮想マシンの起動方法

次節ではこれらの問題に対する我々の考えを述べる．

5.1 Hypervisor がハードウェア命令を正しくエミュレートできているのか？

HLT 命令は，処理すべきタスクが存在しない場合に，電力消費，CPU の発熱等を抑えるために idle タスク上で呼び出されるアセンブラ命令である．HLT 命令が発行された CPU は必要最低限の回路以外への電力供給を停止する．割り込み信号など何らかの外部イベントによって当該 CPU は停止状態から回復する．停止状態からの回復時間はごくわずかである．

しかしながら，Hypervisor によってトラップされる GuestOS の HLT 命令は，Hypervisor 上の idle タスク実行に置き換えられるので，vCPU スレッドは物理 CPU と同じ速度で停止状態から回復することはできない．

本研究において，GuestOS による HLT 命令の実行が原因となって，SIP リクエストの応答時間が変動した．GuestOS 上の HLT 命令実行は OS として正しい動作であるが，HLT 命令を Hypervisor がトラップしてから，再度 GuestOS に実行主体が戻る（VM_ENTER）には，物理 CPU が停止状態からの回復時間と比較して，はるかに長い時間を要する．

よって，応答時間の悪化を防ぐには，GuestOS のハードウェア命令を Hypervisor が正しくエミュレーションできているかどうかを改めて考え直す必要がある．

5.2 スレッドの「先読み」起動機能を考える

本研究では，SIP リクエストの処理に関連する各スレッドの起動遅延によってパケットが滞留し，応答時間を悪化させることがわかった．さらに，GuestOS 上の Asterisk が到着したパケット毎にスレッドを生成するためスレッドの生成と実行時間がさらに応答時間を悪化させることもわかった．Asterisk の構造上の問題はアプリケーション依存なので，ここでは議論しないが，安定した応答時間を必要とするアプリケーションは，動的なスレッド生成は行わず，worker モデルを利用して，できるだけ応答時間に影響が及ばないようにすべきである．

パケットの滞留は，Hypervisor 上の各スレッドの起動遅延を解消することによってある程度防ぐことができる．起動遅延の解消による効果は，4.6 節の抑制実験によって立証されている．4.6 節では vCPU スレッドが物理 CPU を保持し続けている．つまり，起動し続けたままパケットが到着するのを待機している状態であり，この状態で vCPU スレッドに起動遅延はない．他の手法としては文献 7)のように, virtio のような仮想デバイスを Hypervisor にアウトソーシングすることによって，パケットのコピー回数を減らす手法がある．この手法は，リクエスト処理に関連するスレッドの数を減らすことができるので，本研究における起動遅延の低減に効果があると考える．

起動遅延を解消するには，SIP リクエストに関連するスレッドを「予め」起動しておくとよい．今回のように，GuestOS 上でどのようなアプリケーションが実行され，当該アプリケーションにどのような特性があるかを事前に把握できれば,「予め」起動しておくスレッドを選択することができる．しかしながら，NFV では多種多様なアプリケーションが Hypervisor 上で動作するので，GuestOS 内がブラックボックスであるような場合には，GuestOS 上のタスク配置やスケジューリング状況を透過的に Hypervisor に伝える仕組みが必要になると考える．

6. 関連研究

仮想マシン環境のリアルタイム性向上に関する研究は数多く存在する 3) 4) 5) 6)．これらの多くは，車載システムのように，マイクロ秒単位の応答性を確保しなくてはならないようなアプリケーションに適用される．アプリケーションや Hypervisor の挙動を分析するという我々の研究によって，目標とする応答時間を満たすために，最低限必要な機能は何かということを見極められる．

また，NFV などネットワークサービスの仮想化に伴い，仮想マシン環境のネットワークジッタも問題になりつつある 2)．文献 2)は複数の仮想マシンを Hypervisor 上で実行した際に TCP リクエストの応答時間が変動することを問題にしている．

7. おわりに

本稿では，単体の仮想マシンを Hypervisor 上で実行し，一定の負荷をかけた際の応答時間の変動原因を分析した．

NFV では，様々なサービスが実行される．このような環境において，応答時間の安定性を保つための研究課題は次のものがあげられると考えている．

1. ２台以上の仮想マシンが一台の Hypervisor 上で動作し，物理 CPU を共有しない場合でも一定の応答時間を保つことができるか？
2. ２台以上の仮想マシンが一台の Hypervisor 上で動作し，一つの物理 CPU を共有しつつ一定の応答時間を保つにはどのような機能が必要か？
3. 複数の物理マシンから構成される NFV のシステム環境全体で一定の応答時間を保つにはどのような機能が必要か？

本稿で上げた応答時間変動原因を解決したうえで，上記研究課題を解決することによって，我々は社会インフラとして相応しい NFV の提供が可能になると考えている．

参考文献

1) 菅原，水越，岩田：大規模災害における移動通信サービスの輻輳解決に向けた取り組み，NEC 技報，Vol.65 No.3，2013 年 2 月
http://jpn.nec.com/techrep/journal/g12/n03/pdf/120327.pdf

2) Luwei Cheng, Cho-Li Wang, Sheng Di : Defeating Network Jitter for Virtual Machines, UCC'11 Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing, pp.65-72, 2011.

3) Jan Kiszka: Towards Linux as a Real-Time Hypervisor, KVM forum 2011, Aug 2011.

4) Cong Xu, et.al: vSlicer: latency-aware virtual machine scheduling via differentiated-frequency CPU slicing, HPDC '12 Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, pp.3-14, 2012.
http://old.lwn.net/lwn/images/conf/rtlws11/papers/proc/p18.pdf

5) Alex Landau, Muli Ben-Yehuda, Abel Gordon: SplitX: Split Guest/Hypervisor Execution on Multi-Core, WIOV'11 Proceedings of the 3rd conference on I/O virtualization, 2011.

6) Wind River: NFV と SDN による高パフォーマンスのオープンスタンダード仮想化, 2013
http://www.windriver.com/japan/announces/open_virtualization_profile/pdf/OVP_whitepaper_130924.pdf

7) Hideki Eiraku, et.al : Fast Networking with Socket-Outsourcing in Hosted Virtual Machine Environments, SAC '09 Proceedings of the 2009 ACM symposium on Applied Computing pp.310-317, 2009