結果に再現性のある計算機システムエミュレータ

清水 勇希[1] 高井 峰生[2] 木村 啓二[1]

概要：コンピュータネットワークの研究では，プロトコルやアプリケーションを評価する際にしばしばネットワークシミュレータが利用される．ネットワークシミュレータを使用することで，実機による評価と比べて再現性を高められ再実験も容易になる．また，プロトコルなどの評価対象をモデル化することでシミュレーション時間の削減を期待できる．しかし，評価対象システムの規模によってはモデル化の労力が大きくなり，実装の難易度も高くなる．また，モデル化による誤差のためにシミュレーション結果の傾向が実際とは異なる可能性がある．以上の問題を解決し複雑なシミュレーションを可能とするため，ネットワークシミュレータと計算機システムエミュレータを組み合わせ，評価対象であるアプリケーションを計算機システムエミュレータ上でコードを変更せずに直接実行するという手法が提案されている．問題点として， ネットワークシミュレータでは入力と設定が同じであればシミュレーション結果は常に同じである必要があるが，計算機システムエミュレータでは再現性が保証されておらず，シミュレーションの再現性が低下することが挙げられる．そこで，本稿では計算機システムエミュレータにおける挙動の再現性を確保する手法を提案する．

1. 研究背景

コンピュータネットワークの研究では，プロトコルやアプリケーションを評価する際にしばしばネットワークシミュレータが利用される．ネットワークシミュレータを使用することで，実機による評価と比べて再現性を高められ再実験も容易になる．さらに，ネットワークシミュレータを使用する際にプロトコルなどの評価対象をモデル化する必要があるが，実施対象の実験とは本来関係ない事項を省くことでシミュレーション時間の削減を期待できる．しかし，評価対象システムの規模によってはモデル化の労力が大きくなり，実装の難易度も高くなる．また，モデル化による誤差のためにシミュレーション結果の傾向が実際とは異なる可能性がある．

以上の問題を解決し複雑なシミュレーションを可能とするため，ネットワークシミュレータと計算機システムエミュレータを組み合わせ，評価対象であるアプリケーションを計算機システムエミュレータ上でコードを変更せずに直接実行するという手法が提案されている．例えば，[2] で紹介されている GNS3 や Cloonix 及び imunes などのネットワークシミュレータでは VMware や QEMU などの仮想マシンを端末として利用することもできる．問題点として， ネットワークシミュレータでは入力と設定が同じであればシミュレーション結果は常に同じである必要があるが，計算機システムエミュレータでは実行プログラムが同じでも OS による割り込みやディスクアクセスのタイミングが常に同じとなることが保証されていないので，シミュレーションの再現性が低下することが挙げられる．

計算機システムエミュレータにおける挙動の再現性を確保する手法として，オープンソースの QEMU を対象としたものがある [1]．本手法では QEMU のスナップショットを利用したシステム状態の復元と QEMU を構成するスレッドの実行をノンプリエンプティブ化することで QEMU 上でのプログラム実行時間の再現性の向上を図った．本手法を実装した QEMU 上で計算主体のプログラムと I/O 主体のプログラムを用いて評価を行い再現性の向上を確認したものの，完全な再現性の確保には至らなかった．

本稿では [1] の結果を基にして，さらに以下の機能を実装した．

- I/O 処理関数の実行タイミングの固定化
- QEMU 内のスレッド間の確実な切り替え
- 仮想 CPU スリープ時の時間進行の固定化
- I/O 処理命令の実行タイミングの固定化

上記の機能を実装した QEMU により，計算主体のプログラムと I/O 主体のプログラムを用いて評価した．いずれの場合も再現性を確保することができた．

以降，本稿では第 2 節で QEMU の概要を示し，第 3 節

[1] 早稲田大学
[2] Space Time Engineering, LLC

Copyright (c) 2017 by the Information Processing Society of Japan

で先行研究について述べる．第 4 節では再現性確保手法を提案し．第 5 節では再現性の評価する．第 6 節ではまとめについて述べ，今後の課題について議論する．

2. QEMU の概要

QEMU とは C 言語のオープンソースのソフトウェアで，計算機システムエミュレータとしても仮想化ソフトウェアとしても使用できる．計算機システムエミュレータではホスト環境とは異なるアーキテクチャの OS を含めたプログラムをゲスト環境で動作させることができる [8].

本稿では，再現性を確保する計算機システムエミュレータとして QEMU2.2.1 を対象とする．計算機システムエミュレータはソフトウェア的に別のハードウェアを実現する技術であるため，例えば，Intel CPU の PC 上で Intel CPU 用の OS を動作させることはもちろん，Intel CPU の PC 上で ARM 用の OS やプログラムを動作させることもできる．計算機システムエミュレータとしての QEMU が実行できるアーキテクチャは Intel 386 や x86_64，ARM，PowerPC，SPARC など様々なものに対応している．本稿ではエミュレーションを行う CPU として Intel CPU と ARM を用いた．

2.1 スナップショット機能について

QEMU では実行途中のシステムの状態（CPU や RAM，デバイスなど）を保存し，後で保存した状態を復元し実行を再開することができるスナップショットという機能がある．本稿ではこの機能を使用しすることで，毎回同じシステム状態からプログラムを実行する．

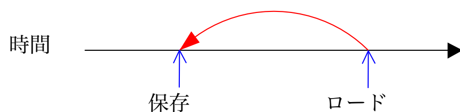

図 1 スナップショットのイメージ図

2.2 QEMU 内のスレッド

今回使用するバージョン 2.2.1 の QEMU は MAIN スレッド，VCPU スレッド，WORKER スレッドの 3 種類のスレッドによって構成されている．MAIN スレッドはイベントループを持ち，モニターの制御やイベント（キーボード・マウス入力やパケットなど）に対するコールバック関数の実行などを行う．VCPU スレッドは仮想マシンの CPU をエミュレートし，ゲスト命令のホスト命令への変換並びに実行，ハードウェアの割り込み，デバイスのエミュレーション，I/O 処理の発行などを行っている．WORKER スレッドでは CPU スレッドで発行された非同期の I/O 処理を実行している．I/O 処理を CPU スレッド内で実行してしまうと，処理の長い I/O 処理をしている間ゲストの実行が止まってしまう．そこで WORKER スレッドで実行させることで，QEMU のパフォーマンスの向上が期待できる．

2.3 QEMU における時刻管理

QEMU は Intel CPU で実装されている TSC(Timestamp Counter) や HPET(High Precision Event Timer)，ARM CPU で実装されているハードウェアタイマをエミュレートすることが可能である．これらをエミュレートする際，オリジナルの QEMU ではホスト環境の実際のハードウェアタイマを参照しているが起動時のオプションで icount を有効にすることでホスト環境の時間を参照せずに QEMU 内の時刻を進めることができる．

icount とは Instruction Counter の略で，ゲストの命令数 1 つを実行するのに 2^N[ns]（N はオプションで指定）要するものとして時間を進める方法である．仮想マシンの時間は命令数を保存してある icount（instruction counter）に基づいて進む．例えば，N として 0 を指定すると 1ns に 1 命令を実行する仮想 CPU を得ることができる．icount を使用することで仮想マシンの時間の進行は，ホストの時間ではなくゲストで実行された命令数に依存するので再現性を保証できる．すなわち仮想マシンで発行された命令列が同じだった場合，ホストの時間参照の場合はホストの影響を受けるので仮想マシン内における実行時間は変動するが，icount では命令数が同じだったら実行時間は同一になる．ゲストの命令列をホストの命令列に変換しないと icount を使用することができないので，icount を有効にできるのは計算機システムエミュレータとしての QEMU である．本稿では再現性を確保するために icount を有効にして QEMU を使用する．

3. 先行研究

3.1 プログラムの実行再現手法

福意等 [1] は計算機システムエミュレータ (QEMU) の再現性を確保するために，スナップショットの利用と QEMU のノンプリエンプティブ化を行った．一般的にプログラムを実行するときに毎回同じシステム状態（時間やメモリ，スケジューリング，CPU，I/O 関連，キャッシュなど）で始まることはないので，同じプログラムを実行しても再現性が得られない．そのため，2.1 節で説明した QEMU のスナップショット機能を利用することでプログラムの実行を毎回同じシステム状態から始めることができる．しかし，QEMU では 2.2 節で説明した MAIN スレッドと VCPU スレッド，WORKER スレッドが pthread ライブラリを用いプリエンプティブスレッドとして並列に動いている．そのため，スレッドのスケジューリングはホスト OS の状態に依存しており，スレッドの実行順序は決定的でない．そこ

Copyright (c) 2017 by the Information Processing Society of Japan

で，[1] では QEMU で使用するスレッドをプリエンプティブスレッドからノンプリエンプティブに切り替えることで，スレッドのスケジューリングが決定的になり再現性が向上した．ノンプリエンプティブスレッドを提供するライブラリは pthread ライブラリとの互換性を意識して開発された nPth ライブラリ [4] を使用した．

[1] の手法では ARM アーキテクチャに対する再現性は確保できたものの，Intel アーキテクチャに対する再現性を確認してみたところ再現性は得られなかった．そのため，本稿では [1] の手法を前提として Intel アーキテクチャ上でも再現性を得られる手法を提案する．

3.2 QEMU の record/replay 機能

QEMU2.8.0 では record/replay 機能 [3] があり，QEMU 内部での実行を記録し分析やデバッグなどのために後で同じ実行を何回も再生できる．record 機能により，非決定的なイベント（キーボード入力やパケット入力など）のみをログファイルに出力し，replay 機能を使用する時それらのイベントを再現することで再現性を確保する．目的の一つとしてはハイゼンバグ[*1]などのように観察するのが困難なバグを再現性のある QEMU を使用することでなくすことである．icount を有効にした場合にこの機能を使用でき，キーボード・マウス入力やホストの時間なども再現できる．

本稿の目的であるネットワークシミュレータとの接続においては有効な手段であるが，現在のところ起動の途中で止まってしまい完全に再現されていないため使用できない．

3.3 VMSimInt 及び SliceTime

ネットワークシミュレータと計算機システムエミュレータを統合した先行研究としては，VMSimInt[5] や SliceTime[6] などが挙げられる．

VMSimInt はネットワークシミュレータ IKR SimLib と計算機システムエミュレータ QEMU を統合した先行研究である．この手法では QEMU の処理速度を無限大と仮定して，QEMU での実行時間は考慮せずにシミュレーション時間のみを元に進行することで再現性のあるシミュレーションを行った．しかし，この仮定ではアプリケーションを評価する際に端末での実行時間を考慮しないため，実際の挙動とは異なるシミュレーションになってしまう．

SliceTime ではネットワークシミュレータとして ns-3 を，仮想マシンとして Xen を採用した．複数の仮想マシンとネットワークシミュレータをバリア同期によって定期的に同期をとることで時間の進行を一致させている．この手法では，大規模な通信システムを評価することができる．しかし，仮想マシンはシミュレーション時間に合わせて進行しているが再現性はない．本稿では，端末での実行時間を考慮し再現性のあるシステムを目標とする．

[*1] デバッグしているときは生じないがデバッグなしの場合だと生じるような観測が困難なバグ

4. 提案する再現性確保手法

4.1 再現性確保の阻害要因

本稿が対象とするネットワークシミュレータと計算機システムエミュレータを組み合わせたシステムで再現性のあるシミュレーションをするには，計算機システムエミュレータの再現性が重要である．すなわち，毎回のシミュレーションで同一の実行時間が再現可能であることが重要である．スナップショットを利用することでプログラムの開始時点では同じシステム状態とすることができるが，それだけでは不十分である．再現性確保の阻害要因としては，I/O の処理やハードウェア割り込み処理，ホストの時刻参照，QEMU のスレッド間のスケジューリングなどがある．

非同期の I/O 処理は WORKER スレッドで行われるが，VCPU スレッドで I/O 処理を発行した後に WORKER スレッドで実行するタイミングはホスト OS のスレッドのスケジューリングによるため，その時のホストの状態に依存してしまう．また，icount を使用することでゲストの命令による再現性は保証されるが，その他の部分でホストの時間を参照している場所があり時間を変動させる原因となる．

さらに先行研究 [1] では，ARM アーキテクチャと Intel アーキテクチャで大きく再現性が異なったが，これはアーキテクチャが違うことで QEMU 内で実行する関数やスレッドの使い方が大きく違うためである．

以下，このような再現性確保の阻害要因をなくし QEMU における実行で再現性を確保する手法について述べる．

4.2 QEMU 内スレッドのスケジューリング

先行研究 [1] で導入した Non-preemptive スレッドライブラリ nPth を使用することで QEMU 内のスレッド実行は逐次的になり再現性は向上した．しかし，いくつかの問題点によりこれだけでは再現性が保証できていない．この問題を解決する手法を述べる．

4.2.1 I/O 処理関数の実行タイミングの固定化

4.1 節で述べたように，VCPU スレッドで I/O 処理を発行した後に WORKER スレッドで実行するタイミングはホスト OS のスレッドのスケジューリングによるため，その時のホストの状態に依存してしまう．そのため，再現性を向上させるには WORKER スレッドで I/O 処理を実行するタイミングを毎回同じにする必要がある．

ここで，QEMU の WORKER スレッドで I/O 処理が行われる様子を説明する．まず，I/O 処理発行時に I/O 処理を行うコールバック関数は VCPU スレッド内の thread_pool_submit_aio 関数でキューに登録される．そして WORKER スレッドに切り替わったときにキューから登録された関数を取り出し実行する．スレッドライブラリ nPth を利用した QEMU ではスレッドを切り替えるとき npth_usleep 関

Copyright (c) 2017 by the Information Processing Society of Japan

数を使用している．この関数は次にどのスレッドに切り替えるかはホスト OS に決定されるため，スレッドの実行順序は完全に決定的でない．図 2 に上記の処理の流れの例を示す．

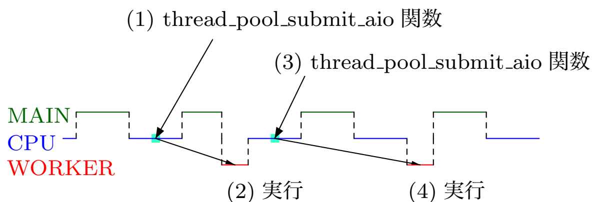

図 2 WORKER スレッドの実行タイミング（改良前）

例えば，図 2 の (1) の時点で thread_pool_submit_aio 関数により登録されたコールバック関数は，VCPU スレッドから MAIN スレッドへ，そして MAIN スレッドから WORKER スレッドへと切り替わった (2) の時点で実行される．同様に，(3) の時点で登録されたコールバック関数は (4) の時点で実行されている．しかし，npth_usleep 関数では次にどのスレッドに切り替わるかは OS に依存するので，WORKER スレッドでの実行のタイミングは例で示した時ではなく前後してしまう可能性がある．つまり，図 2 では (2) と (4) の時点で実行されたが，毎回そのタイミングで実行される保証はない．この実行のタイミングのばらつきが VCPU スレッドで作成されるゲストの命令列に影響を与えるので icount もずれてしまう．

そこで，図 3 のように WORKER スレッドの実行タイミングを thread_pool_submit_aio 関数でキューに登録した直後に WORKER スレッドに切り替え，実行するように固定化する．これにより，毎回同じタイミングでき，WORKER スレッドにおける非同期の I/O 処理が icount に影響を与えることがなくなる．

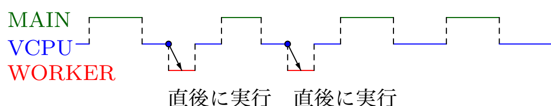

図 3 WORKER スレッドの実行タイミング（改良後）

これを実現するためスレッドの切り替えに npth_usleep 関数は使用せず，シグナルを利用したスレッドのサスペンドと再開を実装した．これを利用し決定的にスレッドを切り替える処理の流れを以下に示す．

まず，VCPU スレッドから WORKER スレッドへ切り替えるためには MAIN スレッドをサスペンドしておく必要がある．その後，VCPU スレッドを wait 状態にすることで WORKER スレッドのみがアクティブなスレッドになり WORKER スレッドへと切り替わる．WORKER スレッドでの実行が終了したら VCPU スレッドのスリープを解除し，今度は WORKER スレッドが wait 状態に入ることで VCPU スレッドに実行が移る．MAIN スレッドはサスペンドしているので MAIN スレッドに移る可能性はない．VCPU スレッドに実行が戻ったら MAIN スレッドを再開させる．また，次に thread_pool_submit_aio 関数に入るまで WORKER スレッドを呼ばないようにするために WORKER スレッドをサスペンドさせる．再び thread_pool_submit_aio 関数に入るときにサスペンドしていた WORKER スレッドを再開させ，同じ処理を繰り返すことで WORKER スレッドの実行タイミングを固定することができる．

4.2.2　スレッド間の確実な切り替え

4.2.1 節で述べたようにスレッドライブラリ nPth を用いた QEMU では，スレッドの切り替えを行うとき npth_usleep 関数を使用している．この関数は指定された秒数だけスリープする関数でスレッドの切り替わりを保証する関数ではない．そのため，ホスト OS の状態によりスレッドの実際のスケジューリングが決定する．例えば，図 4 のように VCPU スレッドから MAIN スレッドへの切り替えが期待される時点で切り替わらないことがある．実行の度にスレッドスケジューリングが変わると処理の順番が異なる．そのため，ゲスト命令列も変化し再現性が低下する．

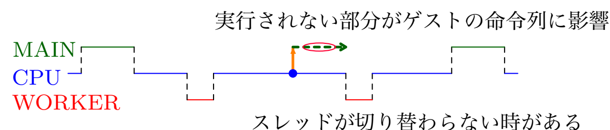

図 4 スレッドが切り替わらない場合

そこで，npth_usleep 関数を使うのではなく npth_cond_wait 関数と npth_cond_signal 関数を使用することで図 5 のようにスケジューリングを決定的にする．npth_cond_wait 関数と npth_cond_signal 関数は pthread_cond_wait 関数と pthread_cond_signal 関数に対応するもので，npth_cond_wait 関数は特定の条件が真になるのを待ち，npth_cond_signal 関数で送られるシグナルで条件が満たされたら実行を再開する．この 2 つの関数でスレッドを切り替えるようにすることで，スレッドスケジューリングが決定的になり再現性が向上する．

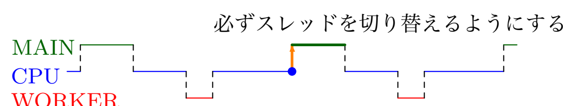

図 5 スレッドが必ず切り替わる場合

Copyright (c) 2017 by the Information Processing Society of Japan

4.2.3 main loop の回転数の固定化

MAIN スレッド，VCPU スレッド，WORKER スレッドは仮想マシン実行中，それぞれ無限ループを実行している．MAIN スレッドのループでは，QEMU 内の期限の切れたタイマー関数の実行や読み書き可能なファイルディスクリプタを定期的にチェックしそれに対応するコールバック関数の実行などを行う．VCPU スレッドのループでは，ゲスト命令列のホスト命令列への変換並びに実行や I/O 処理との同期などを行う．WORKER スレッドのループでは，VCPU スレッドで発行された I/O 処理を非同期に行う．オリジナルの QEMU ではプリエンプティブスレッドなので，これら 3 つのスレッド（ループ）をホスト OS のスケジューリングにより実行する．一方スレッドライブラリ nPth を利用した QEMU では，ノンプリエンプティブスレッドなので遷移のタイミングを指定し順番に切り替える必要がある．

切り替える際，MAIN スレッドのループ（以降 main loop と呼ぶ）はスレッドライブラリ nPth を利用した QEMU において，実行権が MAIN スレッドに移ってから VCPU スレッドに移るまでの回転数が決まっていない．MAIN スレッドで実行される関数はモニタの制御のように icount に影響を与えないものもあるが，ディスクアクセスのコールバック関数の実行などのように icount に影響を与える関数も実行されることがあり，MAIN スレッドの回転数が決まっていない場合 icount が変動する可能性がある．そこで，main loop の回転数を決め，その回転数になったら VCPU スレッドに切り替えるようにした．

4.3 仮想 CPU スリープ時の時間進行の固定化

QEMU では仮想 CPU がスリープする際，仮想時間（icount）も進まなくなるとスリープ開始時点以降に先に生じるタイマ割り込みを実行することができなくなり，スリープ状態から抜け出せなくなってしまう．この問題を解決するためにスリープしている間も仮想時間を進める qemu_clock_warp 関数がある．オリジナルの QEMU ではスリープしている間も時間を進めるために，図 6 のようにホストの時間を参照しながら進行している．ホストの時間を参照すると実行ごとに異なる時間の進み方をするため，再現性を確保するためにはホストの時間を参照しないようにする必要がある．

解決策として，VCPU スレッドをスリープさせずにタイマー割り込み処理へとジャンプすることでホストの時間が関係なくなり再現性が得られる．

4.3.1 復元時の qemu_clock_warp 関数の実行回避

オリジナルの QEMU では時刻を進めるため，スナップショットを復元し始めてから実際に VCPU スレッドでゲストの命令列の処理が始まる前までに qemu_clock_warp 関数が実行される．しかし，この関数を実行すると固定値だけ時間が進んでしまうため，スナップショットを保存した時と復元した後の icount がずれてしまう．そこで，スナップショットを復元し始めてから VCPU スレッドでの処理が始まるまでは qemu_clock_warp 関数を実行しないようにした．

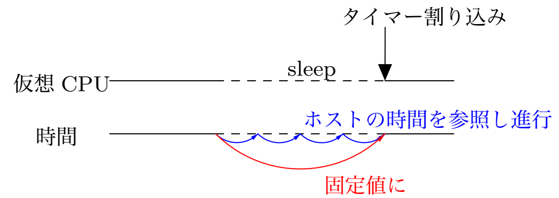

図 6 qemu_clock_warp 関数の動き

4.4 I/O 処理命令の実行タイミングの固定化

QEMU 内部にはマシンの設定や CPU の状態，デバイス，時刻などのデータを保持するいくつもの構造体がある．その中に CPU のコア数やスレッド数などの CPU の状態を保存している CPUState という型の構造体がある．CPUState 型のメンバの一つに can_do_io 変数があり，この変数は I/O 実行の可・不可を表している．

図 7 に can_do_io 変数の利用例として I/O 読み込み処理のコードの一部を示す．図 7 の 13 行目の if 文の条件式に cpu->can_do_io 変数が使われている．この条件式が true の場合，cpu_io_recompile 関数が実行される．この関数は icount が有効な場合に I/O 処理用の命令群を TB（Translation Buffer）の最後に行うようにする関数である．TB の最後に実行することで毎回 I/O 処理命令の実行タイミングが同じになり再現性が得られるようになる．しかし，cpu->can_do_io 変数に対する処理が正しく行われておらず，この関数を呼び出し I/O 処理用命令群が TB の最後にならず再現性が失われていた．

そこで，図 8 のように tcg_qemu_tb_exec 関数（10 行目）の前後の行（9 行目と 11 行目）に cpu->can_do_io 変数に値を代入する．icount を有効にしているとき，use_icount 変数には 2 が代入されているので tcg_qemu_tb_exec 関数の実行前には cpu->can_do_io 変数には 0 が入り，実行後には 1 が入る．このようにすることで，tcg_qemu_tb_exec 関数を実行しているときは前述の cpu_io_recompile 関数に入ることができる．そのため，I/O 処理用命令群を TB の最後に置けるため再現性が確保される．なお，この変更は QEMU2.8.0 のソースコードを参照して取り入れたものである．

5. 再現性の評価

5.1 スナップショットを利用した評価方法

第 4 節で述べた再現性確保の手法の評価方法として，評

Copyright (c) 2017 by the Information Processing Society of Japan

```
static inline DATA_TYPE
glue(io_read, SUFFIX) (CPUArchState *env,
                       hwaddr physaddr,
                       target_ulong addr,
                       uintptr_t retaddr)
{
   uint64_t val;
   CPUState *cpu = ENV_GET_CPU(env);
   MemoryRegion *mr = iotlb_to_region(cpu->as,
        physaddr);

   physaddr = (physaddr & TARGET_PAGE_MASK) +
       addr;
   cpu->mem_io_pc = retaddr;
   if(mr != &io_mem_rom && mr != &
       io_mem_notdirty && !cpu->can_do_io){
      cpu_io_recompile(cpu, retaddr);
   }

   cpu->mem_io_vaddr = addr;
   io_mem_read(mr, physaddr, &val, 1 << SHIFT
       );
   return val;
}
```

図 7 I/O 処理における読み込み関数

```
/* Execute a TB, and fix up the CPU state
    afterwards if necessary */
static inline tcg_target_ulong cpu_tb_exec(
    CPUState *cpu, uint8_t *tb_ptr)
{
    CPUArchState *env = cpu->env_ptr;
    uintptr_t next_tb;

    /* 省略 */

    cpu->can_do_io = !use_icount; // <- 追記
    next_tb = tcg_qemu_tb_exec(env, tb_ptr);
    cpu->can_do_io = 1; // <- 追記

    /* 省略 */

    return next_tb;
}
```

図 8 cpu_tb_exec 関数

価対象プログラムの QEMU 内における実行時間の変動を計測した．QEMU 内における実行時間とは，実際にかかった実行時間ではなく QEMU 内部の icount をもとに計算される仮想マシン内の実行時間である．

まず，仮想マシン内で評価対象のプログラムを実行する．その直後にスナップショットを取りシステムの状態を保存する．この評価対象のプログラムでは実行の最後に実行にかかった仮想時間を出力する．保存したスナップショットを復元し，最後まで実行させ仮想マシン内における実行時間を出力することを 100 回繰り返す．出力された実行時間を毎回記録し，100 回中何回同じ出力になったかを調べる．なお，QEMU の icount を有効にするなどの起動オプションは全て同じにし，実行している間キーボード・マウス入力とネットワークによる通信は遮断した環境で行う．

評価対象プログラムとして，C 言語のベンチマークと dd コマンドを実行するスクリプトを用意した．C 言語のベンチマークとしては SPEC[9] が提供する 179.art, 183.equake, 470.lbm と MediaBench[10] の mpeg2dec, mpeg2enc を同時に実行することで，計算による再現性を調べる．また，dd コマンドはファイルシステム等のデータに直接アクセスを行いコピーや変換を行うプログラムであり，I/O による再現性の影響を調べる．

5.2 評価環境

5.2.1 ホストの環境

QEMU 自体を走らせるホストの評価環境を表 1 に示す．

表 1 ホストの環境

OS	Ubuntu 14.04 64bit
Kernel	3.13.0
CPU	Intel Xeon E5-1650
コア数	6
RAM	16GB

5.2.2 ゲストの環境

ARM CPU，Intel 386，x86_64 をエミュレートするゲスト（QEMU 内）の環境をそれぞれ表 3，表 2，表 4 に示す．

表 2 ARM CPU の仮想マシンの環境

OS	Debian wheezy
Kernel	3.2.0
CPU	ARM Cortex-A9
コア数	1
RAM	1GB

表 3 Intel 386 の仮想マシンの環境

OS	Debian wheezy
Kernel	3.16.0
CPU	Intel 386
コア数	1
RAM	1GB

Copyright (c) 2017 by the Information Processing Society of Japan

表 4 x86_64 の仮想マシンの環境

OS	Debian wheezy
Kernel	3.16.0
CPU	x86_64
コア数	1
RAM	1GB

5.3 再現性の評価結果

オリジナルの QEMU2.2.1 と先行研究 [1] の Non-preemptive QEMU 及び本稿の提案手法を取り入れた QEMU に対して仮想マシンのアーキテクチャを ARM CPU と Intel CPU それぞれに対して評価を実施した．C 言語のベンチマークと dd コマンドを実行するスクリプトの評価プログラムにおける評価結果をそれぞれ表 5， 表 6 に示す．なお，表の中の数字は 100 回中何回同じだったかを表している．

表 5 ベンチマーク実行時の再現性

	ARM	Intel 386	x86_64
オリジナルの QEMU	0	0	0
Non-preemptive QEMU	100	52	4
提案手法による QEMU	100	100	100

表 6 dd コマンド実行時の再現性

	ARM	Intel 386	x86_64
オリジナルの QEMU	0	0	0
Non-preemptive QEMU	100	6	0
提案手法による QEMU	100	100	100

5.4 オーバーヘッドの評価

オリジナルの QEMU では MAIN スレッド，VCPU スレッド，WORKER スレッドはプリエンプティブスレッドとして並行して動いていたが，Non-preemptive QEMU や提案手法による QEMU では逐次的に動くようにノンプリエンプティブスレッドにしたので，オリジナルの QEMU に対してオーバーヘッドが生じる．また，仮想マシンとして QEMU を使用するとゲスト命令を動的にホスト命令に変換するので実機で実行した時に比べて大幅に遅くなる．

そこで，実機と 3 つの QEMU において同じプログラムを実行させた時にかかる実際の時間を測定し比較する．なお，実際の時間とは仮想マシン内で表示される QEMU 内における実行時間ではなくプログラムを実行するのに実際にかかった時間のことである．また，実機での実行時間の測定は QEMU 実行時のホスト環境である表 1 の環境上でのみ行った．

表 7 ベンチマーク実行時に実際にかかる実行時間 [s]

	ARM	Intel 386	x86_64
実機	-	-	1.192
オリジナルの QEMU	27.779	38.367	21.951
Non-preemptive QEMU	32.270	39.320	22.479
提案手法による QEMU	27.347	37.699	21.323

表 8 dd コマンド実行時に実際にかかる実行時間 [s]

	ARM	Intel 386	x86_64
実機	-	-	1.670
オリジナルの QEMU	26.167	4.991	4.111
Non-preemptive QEMU	881.333	26.187	15.123
提案手法による QEMU	37.779	7.386	6.507

5.5 考察

表 5，表 6 からわかるように，提案手法による QEMU では ARM，Intel 386，x86_64 の全てにおいて再現性を確保することができた．このことから，WORKER スレッドでの I/O 処理関数の実行タイミングの固定化やスレッドの確実な切り替え，仮想 CPU スリープ時の時間進行の固定化，I/O 処理命令の実行タイミングの固定化をすることは特に，Intel CPU における再現性を確保する上で有効であることが確認できた．

表 9 は，提案手法による QEMU で C 言語のベンチマークを評価したときに出力された QEMU 内における実行時間である．art と equake はナノ秒オーダーで一致し，その他はマイクロ秒オーダーで同じ実行時間になっている．もし，途中で再現性がなくなり異なる実行をしていたら icount がずれ，結果としてこの実行時間も違う値になったはずである．しかし，提案手法による QEMU 上では毎回同じ値になったので毎回同じ実行をしていたことが保証される．また，今回の評価では他の QEMU と比較するために 100 回のみの測定結果にしてあるが，提案手法による QEMU だけの結果としては 1000 回評価した場合も全て同じ実行時間になった．

表 9 ベンチマーク実行時の QEMU 内における実行時間 [s]

	ARM	Intel
art	24.825060360	26.265812
equake	10.486911610	7.888500
lbm	74.908794	33.574500
mpeg2dec	72.292593	57.845241
mpeg2enc	74.773814	61.984059

オーバーヘッドの評価をした結果である表 7，表 8 では，そもそもオリジナルの QEMU での実行時間は実機に比べかなり遅いことがわかる．これは計算機システムエミュレータであるためゲストの命令列を動的にホストの命令列

Copyright (c) 2017 by the Information Processing Society of Japan

に変換していたことが原因である．特に x86_64 について見ると，ベンチマークの実行時間ではオリジナルの QEMU は実機より約 20 倍遅くなるが，dd コマンドの実行時間は約 4 倍になっている．この違いは C 言語によるベンチマークプログラムは計算主体であり，ゲストマシンの命令列をホストマシンの命令列に変換する処理がボトルネックになるのに対し，I/O 主体の dd コマンドでは実行時間が I/O に多く費やされているためである．

QEMU 同士の比較としては，オリジナルの QEMU と提案手法による QEMU の実行時間の差は表 7 のようにベンチマーク実行時ではほとんど変わらないのに対し，表 8 のように dd コマンド実行時では遅くなっている．このことから，本稿ではノンプリエンプティブなスレッドにしたため I/O に関する処理を行う WORKER スレッドと VCPU スレッドを並行して行うことができなかったため dd コマンド実行時では遅くなっていることがわかる．また，提案手法による QEMU は Non-preemptive QEMU より大幅に実行時間を短くできたことがわかる．

6. まとめ

コンピュータネットワークのシミュレーションによる評価を行う際，実アプリケーションのように複雑でモデル化が困難なシステムをモデル化しないで QEMU のような計算機システムエミュレータで直接実行することで，モデル化するのはネットワークの部分だけですむようになる．

しかしながら，計算機システムエミュレータではデバイスアクセスや割り込み，ホストの時間参照，QEMU 内のスレッドスケジューリングなどのために実行の再現性が得られない．本稿では先行研究で提案された QEMU のスナップショット利用とノンプリエンプティブ化に加えて，QEMU 内のスレッドスケジューリングの固定化，仮想 CPU のスリープ時における時間進行の固定化，I/O 処理をする命令列の実行タイミングの固定化を提案した．これらの改良を加えた QEMU 上で改めて ARM と Intel CPU に対して評価をしたところ，ARM，Intel 386，x86_64 全てにおいて 100 回中 100 回同じ実行時間を得られた．このことから，Non-preemptive QEMU に加えた改良点は Intel CPU の再現性の向上に有効だったことが確認できた．

今後の課題として，ネットワークシミュレータと接続するためのインターフェースを作りパケットのやり取りをしても再現性が保たれるような手法の開発が挙げられる．

参考文献

[1] 福意大智, 水本旭洋, 西本真介, 金田茂, 高井峰生, 木村啓二：計算機システムエミュレーションにおける再現性の評価, マルチメディア, 分散, 協調とモバイル (DICOMO2015) シンポジウム (2015).

[2] Vesel, P., Karovi, V. and Karovi ml, V. : Tools for modeling exemplary Network Infrastructures, *Proc. the 7th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (2016)*

[3] QEMU, QEMU Emulator User Documentation, 入手先 ⟨https://qemu.weilnetz.de/doc/qemu-doc.html⟩ 入手先 ⟨http://wiki.qemu.org/Features/record-replay⟩, （参照 2017-05-09）

[4] npth - the new pth library, 入手先 ⟨https://github.com/gpg/npth⟩ （参照 2017-05-09）

[5] Werthmann, T., Kaschub, M., Khlewind, M., Scholz, S. and Wagner, D. : VMSimInt: A Network Simulation Tool Supporting Integration of Arbitrary Kernels and Applications, *Proc. the 7th International ICST Conference on Simulation Tools and Techniques (2014).*

[6] Weingrtner, E., Schmidt, F., Lehn, H.V, Heer, T. and Wehrle, K. SliceTime: A platform for scalable and accurate network emulation, *Proc. the 8th USENIX conference on Networked systems design and implementation (2011).*

[7] Maier, S., Herrscher, D. and Rothermel, K. : Experiences with node virtualization for scalable network emulation. *Computer Communications (2007).*

[8] Bellard, F. : QEMU, a Fast and Portable Dynamic Translator, *Proc. the annual conference on USENIX Annual Technical Conference (2005).*

[9] Standard performance evaluation corporation, 入手先 ⟨https://www.spec.org/⟩ （参照 2017-05-09）

[10] MediaBench Consortium, 入手先 ⟨http://mathstat.slu.edu/ fritts/mediabench/⟩ （参照 2017-05-09）

Copyright (c) 2017 by the Information Processing Society of Japan