Vol.2018-OS-144 No.4
2018/7/30

カーネルクラッシュからのシームレスなプロセスリカバリ手法

寺田 献[1,a)]　山田 浩史[1,b)]

概要：OS（オペレーティングシステム）カーネルのクラッシュからの一般的な復帰方法は OS の再起動であるが，再起動を行うとメモリ上に展開されているデータは失われてしまう．実行されていたアプリケーションプログラム（App）の実行状態を失うことで App の可用性は低下し，そのサービスの停止によってサービス提供者に損害を与える．App 復元のためには，そのユーザ空間のメモリ内容を復元することに加え，カーネル空間内にあるプロセスメタ情報をも復元する必要がある．App が発行したり受け取ったりした I/O を失うと App 復元後にそのメモリ内容との整合性が取れなくなるため，プロセスメタ情報には I/O に関する情報も含まれる．本研究では App のシステムコールと OS の I/O を補足することで，カーネル内の I/O に関するプロセスメタ情報を保存し，OS の再起動後に I/O の一貫性を取ったプロセスリカバリを行う．ShadowBuddy は数%のオーバヘッドで，OS 透過に App のメモリ保護を行い，発行した I/O を記録することができる．

キーワード：オペレーティングシステム，信頼性

1. はじめに

OS はハードウェアの特権を持って実行されるため，マシン全体の信頼性の根幹を担っている．しかしながら OS のバグは数多く報告されており，バグが実行されることで障害が発生する可能性は常に存在する．使用するハードウェア特有だったり一部の競合状態であったりと，特定の条件下でのみ発症するバグもあるため報告されたバグすべてがマシン利用者にとって致命的なものであるとは言えないが，その分バグの特定・修正が遅れることもある．OS 内に存在するバグがによっていつ障害が発生するかはわからず，信頼性を低下させる要因の一つとなっている．Windows では 1000 行あたり 20 箇所程度のコード上の欠陥が存在すると推定されており [1]，またオープンソースの OS で利用数も多い Linux では，数ヶ月ごとにマイナーバージョンアップが行われ，各マイナーバージョンに対して数千から 1 万個程度のパッチが作成される [2]．プログラムのバグを修正するためにはアップデートが行われるが，OS のアップデートが他のバグを含んでしまうという場合も確認されており，14.8%から 25.0%の修正には新たなバグを含んでいるという分析 [3] もある．

OS のバグによって発生する障害の影響は様々である．メモリリークは本来使用できるはずのメモリが使用できなくなることでメモリの枯渇が発生し，マシン全体の処理性能が低下する．システムコール中などにデッドロックが発生すると App に実行が戻らなくなり，App の実行が継続できなくなる．SDC（Silent Data Corruption）は，バグの実行時点では障害とならないものの，後に破壊されたメモリオブジェクトが参照されることで障害となる可能性がある．クラッシュは，ハードウェアからの例外やフォルトなどを OS が適切に処理できない場合や，メモリオブジェクトが破壊されるなどして Invariant が崩れたことを検知した場合などに引き起こされる．それ以上の OS 実行は継続することができない，あるいは継続することが不適切であるという状態である．

OS の可用性が失われると，その上で動作する App が提供するサービスの可用性も失われてしまう．App は OS の機能を利用して動作するため，App が障害が発生した OS の機能を利用した時点で App の動作に影響が出る．メモリの枯渇であればスワップアウトが多発することで App の実行速度が大幅に低下し，デッドロックでは App の実行が停止，クラッシュではそのマシン上で実行されていたすべての App の実行が停止される．

[1] 東京農工大学
Tokyo University of Agriculture and Technology
[a)] terada@asg.cs.tuat.ac.jp
[b)] hiroshiy@cc.tuat.ac.jp

ⓒ 2018 Information Processing Society of Japan

OS のバグによる障害を緩和するために，いくつかの手法が提案されている．しかしながら，既存の手法では OS の障害によって App の実行状態が破壊される可能性があり，OS の障害から App の保護を行うことはできていない．あるいは，App 透過ではなく人間の仲介を必要とする．OS 障害の検知後に再起動した OS がメモリ上のデータオブジェクトを引き継ぐ手法 [4] では，OS 内のデータオブジェクトを直接コピーして使用するため，OS 内のメモリオブジェクトが破壊される障害では誤った実行状態で App を再開させてしまう可能性がある．OS の障害と復元処理が切り離されておらず，復元の信頼性が低い．

OS の障害が発生した後に App の実行状態を復元する手法 [5], [6], [7], [8] では，あらかじめ App の実行状態を定期的に保存しておき，障害発生後に App の実行状態を復元する．これらの手法は App 内に障害があることを想定したもので，OS 内の障害から App の実行状態を保護することはできない．

OS 内の障害が App の実行状態を破壊することを防ぐ手法 [9], [10], [11], [12], [13], [14] では，OS の挙動によって App の実行状態が不正な値で上書きされることが無いように，OS より特権の強い仮想マシンモニタ（VMM）やハードウェアの保護機能である Intel SGX を用いて App のユーザ空間のメモリを保護する．これらの手法では OS 内のメモリオブジェクトを破壊する障害や，OS の実行が継続できなくなるような障害に対しては効果が無く，OS の実行が継続できなくなれば App も再起動しなければならない．また App の実行状態が OS から保護されているために，OS を信頼すること無く Checkpoint を利用することができない．

本論文では App の実行状態を保護しつつ，障害発生時に App の実行を再開させる手法を提案する．App の実行状態と I/O の実行状態は整合性を保ち，復元した App の I/O は障害発生の直前の状態を持っており，App を変更することなく App 透過に復元することを保証する．我々の提案手法では OS 内のデータ破壊が起こったとしてもサービスを復旧することができる．App の実行状態は OS から保護され，OS の障害によって破壊されることはない．想定する障害は，OS の再起動によって復旧することができるものとする．

提案手法を評価するために，I/O を伴う App の保護中に発生するオーバヘッドについて計測を行う．

2. 関連研究

障害が起こった場合に OS を高速に再起動して App の実行を再開する手法 [15], [16] では，OS の再起動にかかる時間を短縮することで App のサービスダウンタイムを短縮することができる．これらの手法では App の実行状態が失われるため，復元される App の状態は App 自身の実行状態復元機構に依存する．そのような機構が存在しない場合には App の状態は初期化される．実行状態を復元することができたとしても，メモリ上のキャッシュを失うことはサービスの再開後のスループット低下につながる．たとえば DBMS では，ディスクやネットワークを経由して得た DB の内容をメモリ上にキャッシュとして保持しており，このキャッシュを失えばキャッシュを再構築する時間が必要になる．キャッシュを再構築するためには多くの I/O が発生し，その間 I/O の競合によって DB のスループットが低下する．大量のメモリを使用するインメモリ DB ではより顕著で，再起動後に要求されたスループットを達成するまでの時間は長期化する．

Otherworld [4] は OS の障害を App から隠蔽し，障害発生後に透過的に App を復元する．OS の障害が起こるとメモリの予約領域を使って OS を再起動させ，障害が発生した OS 内のメモリオブジェクトを引き継いで App の実行状態を復元する．OS 内の App 実行状態であるプロセスメタ情報は保護の対象となっておらず，OS の障害が発生した時にこれらの情報が破壊されれば，正しく App を引き継ぐことはできず再起動後も同じバグを実行して障害を引き起こす可能性がある．OS の障害によって App のユーザ空間のメモリが破壊されることを防ぐために，OS 実行中は App のユーザ空間のメモリに書き込み保護を行う．しかしこの保護機構自体が OS の機能として実装されているため，OS の障害がこの保護機構を迂回して App のユーザ空間のメモリ保護を破壊する可能性がある．Supervisor Mode Access Prevention（SMAP）とは，カーネルモード中の CPU がユーザ空間に割り当てられたメモリに読み書きをすることを防ぐ CPU の機能である．この保護機能も同様に OS が設定する保護機構であるため，OS の障害から守られているとは言えない．また Linux などが持つストレートマッピングは，すべての物理ページがカーネル空間からアクセスできる仮想アドレスに割り当てられているため，これらの保護機構は容易に迂回される．

OS 内で障害が発生したときに再起動すること無くを元に戻す手法 [17] では，あらかじめ OS の実行状態を更新する際にアンドゥログを作成しておき，偶発的に発生するような OS 内の障害を検知した時にログから OS の実行状態を巻き戻して再実行する．OS 再起動を伴わず App の実行が中断されないためサービスダウンタイムは短いが，復帰可能な障害は再実行すれば回避できるものに限定される．SDC などによってメモリオブジェクトがアンドゥログの取得開始前に破壊されていた場合には，一般的に同じバグが発生するため復元することができない．FGFT [18], Phoenix [19] はデバイスドライバのロールバックを可能とするものである．OS 内の変数のほか，デバイスの状態についても考慮して戻すことで，障害発生前の状態を再現する．しかしこれらは App の実行状態については考慮され

ⓒ 2018 Information Processing Society of Japan

表 1 既存手法と提案手法 ShadowBuddy の比較

	App 変更なし	プロセスの保護	プロセスの実行継続	I/O 状態の継続
Phase-based Reboot [15], ShadowReboot [16]	✓	×	×	×
Otherworld [4]	✓	✓	✓	✓
RecoveryDomains [17], FGFT [18], Phoenix [19]	✓	×	✓	✓
BLCR [5], DMTCP [6], CRIU	✓	×	✓	×
Overshadow [10], InkTag [12]	✓	×	×	×
AppShield [13], Haven [11], SCONE [20]	×	✓	×	×
ShadowBuddy	✓	✓	✓	✓

ていないため，デバイス以外の OS 障害に対しては効果がない．

App の Checkpoint/Restart によって，OS の障害が発生した後に，OS 障害の前に取得しておいた Checkpoint から App の実行状態を復元する事が可能である．Checkpoint は App が持つユーザ空間のメモリと，OS 内にあるプロセスメタ情報の集合である．BLCR [5] などの Kernel-level Checkpoint では，主に OS 内のプロセスメタ情報を直接保存して Checkpoint を作成する．OS 内の障害で破壊されたデータオブジェクトを保存してしまった場合，その Checkpoint を使って Restart をしても正しく App を再開させることができない．DMTCP [6]，CRIU などの User-level Checkpoint では，主にシステムコールを追跡して Checkpoint のプロセスメタ情報を作成する．User-level からは参照することができない OS 内のプロセスメタ情報が存在するため，OS 内の全実行状態を再現できるわけではない．また，最新の I/O 実行状態を復元できない．たとえば TCP プロトコルで通信しているネットワーク接続に受信パケットが到着した場合，接続先のマシンには正しく受信したことを通知する確認応答を返す．しかしこの後から Checkpoint を作成するまでの間に OS 内で障害が発生すると，取得していた Checkpoint には受信したパケット情報が欠落しており，通信相手も確認応答を受け取ったため送信バッファを解放している可能性がある．仮想マシンの Checkpoint を取得する Remus [21] では，発生した I/O は実行せずにログに記録し，Checkpoint を正しく取得した時点で実行する．通信相手が送信バッファを解放しないため正しく復元することができる半面，Checkpoint の取得間隔がネットワーク I/O の遅延に直結し，遅延を低くしようと間隔を短くすれば Checkpoint のオーバヘッドが増大するというトレードオフが存在する．いずれの手法でも Checkpoint はファイルに書き込まれるが，ファイルは OS の機能であるため Checkpoint は OS の障害から保護されていない．

Shielded Execution は特権を持つソフトウェアから App のメモリ内容を保護する．障害の発生した OS や脆弱性をつかれて Rootkit が動作している OS が，App のメモリ内容を読み取ったり書き込んだりすることを防ぐ．RSA の秘密鍵やパスワードなどを保護機構の内側で扱うことで，信頼できない OS が App の機密情報を漏洩させたりサービスを改ざんすることができなくなる．OS よりも特権の強い VMM から App を保護する手法 [10], [12], [13] では，OS の挙動を監視して App の実行中は保護を外し，OS の実行中は保護をかける．App が持つユーザ空間のメモリを暗号化する手法 [10], [12] では，App の実行前に復号し，OS の実行前に暗号化する．メモリ空間を分割する手法 [13] では，App 用のメモリと OS 用のメモリとを区別し，保護対象の App が持つメモリをその App 以外のページテーブルにマッピングすることや，App 以外のメモリページを App の仮想アドレス空間にマッピングすることを禁止する．ハードウェアの支援機能である Intel SGX [22] を用いて App を保護する手法 [11], [20], [23] では，App が自らのメモリに保護をかけ，ハードウェアが App に対する OS の不正な読み書きを遮断する．いずれの手法も OS の障害が発生して停止した場合には App の実行を継続することはできない．また OS 内にある App の実行状態は保護対象外であり，OS の障害によって OS 内にある App の実行状態は破壊される場合がある．**表 1** に既存手法の特徴をまとめる．プロセスコンテキストを復元したり保護したりする手法が提案されているが，これらを両立する手法は存在しない．

3. 提案

本論文では OS のクラッシュ後に，クラッシュ直前の状態から App 再開させる ShadowBuddy を提案する．OS のクラッシュによって引き起こされるメモリオブジェクトの破壊から App のプロセスコンテキストを保護し，App のプロセスコンテキストと I/O デバイスの状態の一貫性を保った復元を行う．ShadowBuddy は OS よりも強いハードウェアの特権を持って動作する VMM で，既存手法の弱点を克服するために OS クラッシュ時にプロセスコンテキストの保護ならびに実行の継続を両立する．デザインゴールは以下の通りである．

- OS クラッシュがコンテキストを破壊しない
- Application-Consistent に実行を引き継ぐ
- OS を信頼しない保護機構を実現する

ⓒ 2018 Information Processing Society of Japan

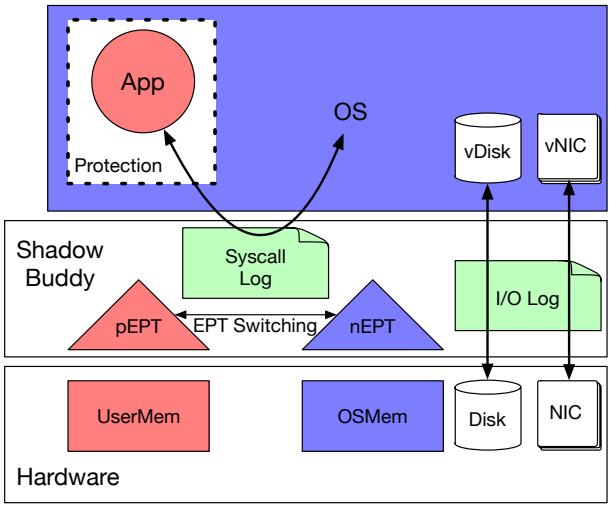

図 1 ShadowBuddy によるメモリ保護とチェックポインティング

- App に改変を必要としない

OS クラッシュが App のコンテキストを破壊することを防ぎ，また OS クラッシュ発生後に Application-Consistent に App の実行状態を復元することで，App は OS クラッシュの発生後に正しく実行を再開することができる．また OS レベル・App レベルの保護機構である場合は，特権を持って実行される OS のバグが保護機構を迂回して App の実行状態を破壊する可能性がある．App の改変を必要としないことで，既存 App や商用 App にも用意に適用することができる．

図 1 のように App のユーザ空間のメモリを OS からの不正な書き込みから保護し，システムコールのログを取り OS 内のプロセスメタ情報を OS クラッシュ後に再現する．I/O デバイスの状態はシステムコールの内容と実際に発行された I/O 処理の比較を行うことで取得し，OS 再起動後に必要があれば最小限の I/O 処理を再実行する．

3.1 EPT Separation

OS はマシンに割り当てられているハードウェア・メモリに対するアクセス権限を持っているため，ページテーブルや SMAP などによる保護機構は十分に動作しない可能性がある．そこで本研究では VMM で使われている MMU 仮想化技術の Extended Page Table（EPT）を用いて，物理ページレベルの保護機構を導入する．また性能上の観点から仮想化環境上のシステムコールには原則 VMM が介入することはない．そこで App が発行するシステムコールを App 透過に，OS の改変をせずに取得するために EPT を活用する．

3.1.1 EPT Configure

EPT は VMM が複数 VM 間の Isolation を実現するために作られた MMU の仮想化支援技術である．OS がページテーブルで複数 App 間の Isolation を実現するように，一般的には他の VM が持つメモリへのアクセスを遮断するように EPT を構築する．OS からの不正な書き込みを防ぐために，図 1 に示すように ShadowBuddy によって保護された App が動作するための EPT（pEPT）と，それ以外の App や OS が動作するための EPT（nEPT）の 2 つを用意する．App の初期化時点で ShadowBuddy に対してハイパーコールを**表 2** に示すように，保護対象の App が使用するメモリは nEPT 上では全て書き込みを禁止することで OS からの不正な書き込みを防ぐ．また OS から保護対象の App に実行が移った瞬間を補足するために，保護対象の App が持つ実行可能なメモリ領域は，nEPT 上では全て実行を禁止する．このように，EPT を活用することで App のメモリ保護とシステムコールの捕捉の 2 つを同時に達成する．

表 2 2 つの EPT の設定

	nEPT	pEPT
OS-Data	rwx	rw-
OS-Code	rwx	rw-
App-Data	r–	rwx
App-Code	r–	rwx

3.1.2 EPT Switching

保護対象 App の実行中は pEPT，それ以外は nEPT を利用するために，App の挙動に合わせて EPT の切り替えを行う．App 実行中に OS のメモリ空間を実行した場合は EPT Violoation が発生して VMM に制御が移る．App からのシステムコールの実行のほか，タイマや I/O デバイスなどからの割り込みが発生した場合も，ハンドラがカーネル空間内に存在するため EPT Violation が発生する．EPT のエントリや TLB のエントリは CPU にキャッシュされるため，EPT Violation が起こり nEPT と pEPT を切り替えるときには，TLB と EPT エントリをすべて削除してから OS と App の制御を切り替える．

3.1.3 ToggleCache

EPT と TLB キャッシュを EPT Violation のたびにフラッシュして失うことは実行速度の低下につながる．フラッシュするために要する時間が長いことと，キャッシュが存在しないことで EPT やページテーブルを読んでキャッシュを再構築する時間がかかるためである．これらをフラッシュしない場合は速度低下を低減することができるが，不適切なキャッシュ残っていることでシステムコールの捕捉ができなかったり保護された App のメモリへの OS からの書き込みが成功したりしてしまう．そこで EPT と TLB キャッシュに存在するタグを利用する．これは VM 間の Isolation のために使われているもので，ShadowBuddy では OS-App 間の Isolation のために用いる．タグの値は Intel VMCS の VIRTUAL_PROCESSOR_ID（VPID）が用いられるため，OS-App 間の遷移が起こるたびにこの値を更新する．VM 間の Isolation の場合は VM のコンテキストスイッ

ⓒ 2018 Information Processing Society of Japan

チが起こるとほとんどのキャッシュが残らないため，一般的にはこの値を常にインクリメントする．ShadowBuddy ではこの遷移が頻繁に起こるため，OS と App にそれぞれ割り当てた VPID を交互に切り替えてキャッシュが失われないようにする．

3.2 Syscall Logging

保護対象 App が発行したシステムコールは全て ShadowBuddy が捕捉する．OS と App の実行が切り替わる時点で EPT Violation が発生して VMM に制御が移るため，VMM の中でシステムコールのログを残す．このログは VMM の中に記録されるため，OS の障害によって破壊されることはない．

システムコールは ShadowBuddy の対応によって以下のように分類できる．

- (a) 無視するもの…getpid など
- (b) ログを変更するもの…mmap，munmap，open，close，dup2，exit など
- (c)ShadowBuffer を作成してからログに追加するもの…read，recv など
- (d)ShadowBuffer にコピーしてからログに追加するもの…write，send など

3.2.1 ShadowBuffer

read システムコールなどは，システムコールの結果を OS が App が持つユーザ空間の仮想アドレスに書き込む．しかし ShadowBuddy は OS がユーザ空間のメモリに直接書き込むことを禁止しているため，結果の書き込み時に EPT Violation が発生してしまう．そこでこのようなシステムコールが発行された時点で，nEPT 中の該当仮想アドレスに対応するメモリページについて，読み書き実行のすべてを許可した新しいメモリページを割り当てる．OS はこの一時的に割り当てられた ShadowBuffer に書き込むため，read システムコールの結果を EPT Violation を起こさずにユーザ空間に書き込むことができる．read システムコールなどが終了してユーザ空間に戻ろうとした時点で再び EPT Violation が発生し，App が read システムコールで受け取ろうとしたメモリ領域に ShadowBuddy が ShadowBuffer からデータを書き込んで ShadowBuffer を解放する．

この手法は一時的に OS の EPT 上にバッファ用のメモリを上書きするものとなっており，マルチスレッド環境などで他のシステムコールと同時に実行された場合に意図しない挙動となる恐れがある．たとえば，スレッド 1 で read(addr=0x60100, size=10) とした場合 0x60100〜0x60109 のメモリ空間に対して，システムコールの結果を受け取るための ShadowBuffer を割り当てる．この時点でスレッド 2 が write(addr=0x60000, size=100) とした場合，0x60000〜0x60099 の内容を OS が参照してファイルに書き込む．しかし OS が読み込むユーザ空間のメモリはスレッド 1 で作成された ShadowBuffer であるため，App が実際に書き込もうとした I/O 内容ではない．そこで OS にメモリ内容を読み込ませるシステムコールでは，対応する仮想アドレスに ShadowBuffer が存在すれば，オブジェクトが存在するメモリページから ShadowBuffer に書き込む．

3.2.2 Log Structure

ログは VMM の内部でデータオブジェクトとして管理する．スレッドごとに CPU レジスタの値を持ち，EPT Violation が起こるたびに値を更新する．EPT Violation の原因がシステムコールである場合，発行された時点ではほとんどのシステムコールはログを更新しない．OS から App にシステムコールが返った時点で，システムコールが成功していれば実際にログを更新する．たとえば匿名領域を割り当てる mmap システムコールの場合，

mmap(addr = NULL, length = 0x10000,
prot = PROT_READ|PROT_WRITE,
flags = MAP_PRIVATE|MAP_ANONYMOUS,
fd = -1, offset = 0);

のように実行される．成功したときの返り値が実際に割り当てられたメモリアドレスであるため，ログにはその情報を加えて保存する．

OS クラッシュが発生したとき，ShadowBuddy 内のログを加工して OS 内のデータオブジェクトを再構築するために必要なシステムコールのリストを作成する．たとえば mmap の例では，

{’syscall’: ’mmap’, ’addr’: 0x80000,
’length’: 0x10000, ’prot’: PROT_READ|PROT_WRITE,
’flags’: MAP_PRIVATE|MAP_ANONYMOUS,
’fd’: -1, ’offset’: 0}

のような形式で OS に渡す．再起動後の OS はこれらの情報と App のメモリ内容を ShadowBuddy から受け取り，プロセスの実行状態を再構築する．

3.2.3 I/O Log

一部のシステムコールと I/O デバイスの状態は密接な関係にある．システムコールとしては I/O が発行されたが実際には I/O が実行されていないということがあり，ShadowBuddy ではシステムコールのログと I/O のログを照らし合わせて再起動後の OS が実行するべき I/O を決定する．ShadowBuddy はディスクと NIC について対応する．

たとえば TCP の接続をしているソケットがあった場合，再起動後の OS ではクラッシュ直前のシーケンス番号と確認応答番号を得る必要がある．しかしこれらの値はシステムコールインタフェースには存在せず，システムコールのログだけでは再構築することができない．そこで，accept/connect といったシステムコールでソケットが作成されたときには，直前に送受信していた SYN-ACK パケットを読んでシーケンス番号と確認応答番号の初期値を

ⓒ 2018 Information Processing Society of Japan

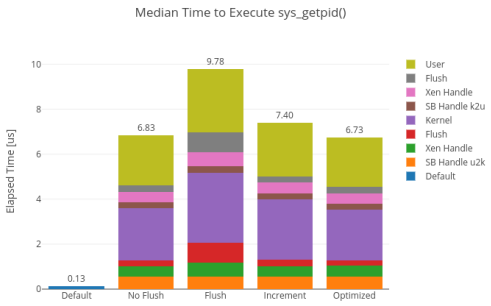

図 2 getpid の実行時間中央値

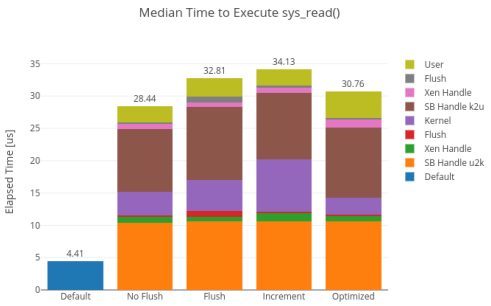

図 3 read の実行時間中央値

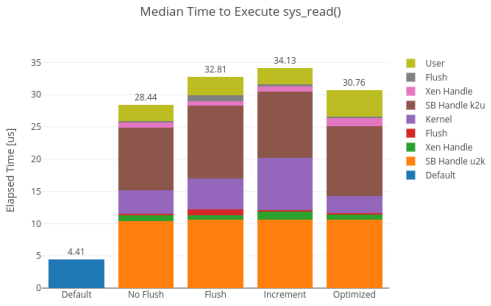

図 4 read の実行時間中央値

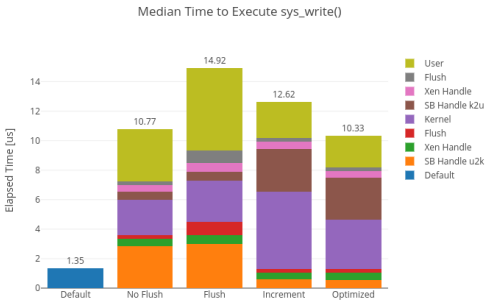

図 5 write の実行時間中央値

取得する．send/sendto/sendmsg といったシステムコールが発行されたときは送信内容のシステムコールログを作成し，通信相手から ACK が返ってきたログは解放することができる．

Linux では受信パケットを App が受け取る前に，OS がパケットを受信した時点で ACK を返すことがあり，通信相手が送信バッファを解放する可能性がある．recv システムコールなどで App がパケットの内容を受け取るまでの間に OS 内で障害が発生すると，受信したパケットのデータが App のメモリ内容にも通信相手の送信バッファからも残らない．そのため ShadowBuddy は通信相手に返す ACK の番号は，App が受け取ったパケットまでの ACK 番号とすることでこの問題を解決する．App がどこまでのパケットを受け取ったかは recv システムコールのログから取得することができ，通信相手に返す ACK 番号に上書きする．

ディスクについては App が書き込んだ内容と発行された I/O の内容を比較することが困難であるため，fsync/fdatasync といったシステムコールが発行され App にシステムコールが返った時点で該当するファイルディスクリプタに関連するログを解放する．Linux には fifo_expire_sync や fifo_expire_async といったパラメータがあり，ディスクへの書き込みは一定時間内に書き込みが開始されるようスケジューリングされるため，これらのパラメータを見て楽観的にログを解放することも可能である．

4. 実装

ShadowBuddy は Xen 4.5.2 上に実装を行い，約 3500 行を追加した．App の保護機構・システムコールの捕捉機構・I/O の捕捉機構については OS の改変を行わず，すべて VMM レイヤに実装を行った．

5. 実験

ShadowBuddy によって発生するオーバヘッドを評価するために，nEPT と pEPT の切り替えや EPT の更新にかかる処理が，App の実行速度をどれだけ低下させるかを計測した．実験は Xeon E3-1270 v2(8-core, 3.5GHz), 12GB RAM, 240GB SSD のマシンで行った．仮想マシンには 6vCPU, 2GB RAM を割り当て，ゲスト OS は改変を行っていない Linux 4.9.68 を用いた．

5.1 Micro Benchmark

getpid, open, read, write の各システムコールを 1,000,000 回繰り返して実行し，実行時間を計測した．VM のスケジューリングによって大幅に所要時間が増えることがあるため，図にはシステムコール 1 回の所要時間の中央値を示す．ToggleCache を用いることで，EPT や TLB キャッシュをフラッシュしない程度の時間でシステムコールが実行できている．保護無しと ShadowBuddy の保護ありを比べると，7.0 から 52 倍の実行時間がかかっていること

ⓒ 2018 Information Processing Society of Japan

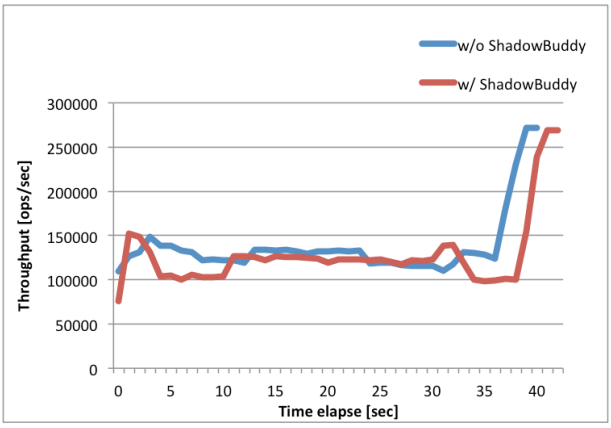

図 6 memcachedb のスループット

がわかる．OS 内の処理が軽い getpid システムコールでは ShadowBuddy のオーバヘッドが目立つが，read や write といった重いシステムコールではオーバヘッドが比較的小さくなる．

5.2 Macro Benchmark

実アプリケーションでの ShadowBuddy のオーバヘッドを調査するために，memcachedb に memtier_benchmark をかけてスループットを計測した．ワークロードは，SET:GET=1:10，キーの分布はランダム，データサイズは 32Bytes である．6 に示した通り ShadowBuddy によるオーバヘッドは小さく，スループットの平均で 6.4%程度である．

6. 結論

OS クラッシュ発生時には App のユーザ空間メモリを保護し，再起動後に App の実行を保証する手法は存在しなかった．OS 内のプロセスメタ情報や I/O デバイスの状態が破壊されたり，Checkpoint データが破壊される可能性があったり，OS 停止時には利用できなかったりしたためである．ShadowBuddy はユーザ空間のメモリ保護とシステムコールの監視によるチェックポイントの作成を行う．OS 再起動後の App 再実行を保証するため，OS クラッシュ時にも利用可能である．ShadowBuddy は OS による App 保護機構と独立した保護機構を提供することができた．従来ハードウェア利用効率の向上を担ってきた VMM に対し，ShadowBuddy は App の可用性向上という新しい役割を与えることができ，信頼性の高い App 実行環境を提供することができる．

参考文献

[1] Dacey, R. F.: Effective Patch Management is Critical to Mitigating Software Vulnerabilities, Technical report, Government Accountability Office (2003).
[2] Corbet, J., Kroah-Hartman, G. and McPherson, A.: Linux Kernel Development – How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It (2010). `https://www.linuxfoundation.org/docs/lf_linux_kernel_development_2010.pdf`.
[3] Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S. and Bairavasundaram, L.: How Do Fixes Become Bugs? – A Comprehensive Characteristic Study on Incorrect Fixes in Commercial and Open Source Operating Systems, *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pp. 26–36 (2011).
[4] Depoutovitch, A. and Stumm, M.: Otherworld - Giving Applications a Change to Servive OS Kernel Crashes, *Proc. of the 5th ACM European Conference on Computer Systems (EuroSys '10)*, pp. 181–194 (2010).
[5] Duell, J., Hargrove, P. and Roman, E.: Requirements for Linux Checkpoint/Restart, Technical Report 8, Berkeley Lab Technical Report, LBNL-49659 (2002).
[6] Rieker, M., Ansel, J. and Cooperman, G.: Transparent User-Level Checkpointing for the Native POSIX Thread Library for Linux, *Proc. of 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '06)*, pp. 492–498 (2006).
[7] Baumann, A., Lee, D., Fonseca, P., Glendenning, L., Lorch, J. R., Bond, B., Olinsky, R. and Hunt, G. C.: Composing OS Extension Safely and Efficiently with Bascule, *Proc. of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, pp. 239–252 (2013).
[8] Vogt, D., Giuffrida, C., Bos, H. and Tanenbaum, A. S.: Lightweight Memory Checkpointing, *Proc. of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, pp. 474–484 (2015).
[9] David, F. M., Chan, E. M., Carlyle, J. C. and Campbell, R. H.: CuriOS: Improving Reliability through Operating System Structure, *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 59–72 (2008).
[10] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J. and Ports, D. R. K.: Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems, *Proc. of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)* (2008).
[11] Baumann, A., Peinado, M. and Hunt, G.: Shielding Applications from an Untrusted Cloud with Haven, *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pp. 267–283 (2014).
[12] Hofmann, O. S., Kim, S., Dunn, A. M., Lee, M. Z. and Witchel, E.: InkTag: Secure Applications on an Untrusted Operating System, *Proc. of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pp. 265–278 (2013).
[13] Cheng, Y., Ding, X. and Deng, R. H.: Efficient Virtualization-Based Application Protection Against Untrusted Operating System, *Proc. of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS '15)*, pp. 345–356 (2015).
[14] Hunt, T., Zhu, Z., Xu, Y., Peter, S. and Witchel, E.: Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data, *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 533–549 (2016).

ⓒ 2018 Information Processing Society of Japan

[15] Yamakita, K., Yamada, H. and Kono, K.: Phase-based Reboot: Reusing Operating System Execution Phases for Cheap Reboot-based Recovery, *Proc. of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, pp. 169–180 (2011).

[16] Yamada, H. and Kono, K.: Traveling Forward in Time to Newer Operating Systems using ShadowReboot, *Proc. of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '11)* (2011).

[17] Lenharth, A., Adve, V. and King, S. T.: Recovery Domains: An Organizing Principle for Recoverable Operating Systems, *Proc. of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pp. 49–60 (2009).

[18] Kadav, A., Renzelmann, M. J. and Swift, M. M.: Fine-Grained Fault Tolerance using Device Checkpoints, *Proc. of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pp. 473–484 (2013).

[19] Smith, R. and Rixner, S.: Surviving Peripheral Failures in Embedded Systems, *Proc. of the 2015 USENIX Annual Technical Conference (ATC '15)*, pp. 125–137 (2015).

[20] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., DanO 'Keeffe, Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C. : SCONE: Secure Linux Containers with Intel SGX, *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 689–703 (2016).

[21] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N. and Warfield, A.: Remus: High Availability via Asynchronous Virtual Machine Replication, *Proc. of the 5th USENIX Symposium on Networked Systems Design and implementation (NSDI '08)*, pp. 161–174 (2008).

[22] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C., Shafi, H., Shanbhogue, V. and Savagaonkar, U.: Innovative Instructions and Software Model for Isolated Execution, *Proc. of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*, pp. 1–8 (2013).

[23] Orenbach, M., Lifshits, P., Minkin, M. and Silberstein, M.: Eleos: ExitLess OS Services for SGX Enclaves, *Proc. of the 12th ACM European Conference on Computer Systems (EuroSys '17)*, pp. 238–253 (2017).

ⓒ 2018 Information Processing Society of Japan