オペレーティングシステムの
ソフトウェアバグからアプリケーションの
実行を保護する手法に関する研究

髙山 献

東京農工大学大学院
電子情報工学専攻
博士（工学）の学位申請論文

2019年3月

オペレーティングシステムのソフトウェアバグから アプリケーションの実行を保護する手法に関する研究

高山 献

論文要旨

コンピュータシステムの信頼性は主に，狭義の信頼性（Reliability），可用性（Availability），保守性（Serviceability），保全性（Integrity），機密性（Security）で評価され，一般的には故障間隔，稼働率，修理時間などで評価される．オペレーティングシステム（OS）はマシン全体を管理するため，OS の信頼性が高いことは，その上で稼働するアプリケーションプログラム（App）の信頼性を担保するための必要条件である．

OS のソフトウェアバグはシステム全体の信頼性を低下させる要因である．例えば，OS カーネル内でバッファオーバーフローが発生すると，不正なアドレスにメモリ書き込みを行ってしまうこともある．OS 内のメモリオブジェクトが破壊されると，App のユーザ空間のメモリを破壊したり，破壊されたメモリオブジェクトを参照した場合に OS がクラッシュしたりする．障害が発生した OS での実行を続ければ，エラーは更に拡大し App や OS が意図しない挙動をするようになる．エンドユーザにとって，OS のバグを解決する現実的な方法は OS の再起動である．OS の再起動によって破壊された実行状態を捨てて，新しい OS の実行状態を作り出すことができる．またバグを修正するためのアップデートを適用するためにも，再起動は有効である．

OS の実行状態の中には App の実行状態も含まれており，OS の再起動によって稼働していた App の実行状態も失われる．Checkpoint によって再起動前の段階で App の実行状態を保存することができていれば，OS 再起動の後に App の実行を再開することができる．しかし Checkpoint の取得頻度が高ければランタイムオーバヘッドが増し，低ければ再開した App の実行状態が古いため障害発生直前の状態になるまで時間がかかるようになるというトレードオフが存在する．OS カーネルのアップデートを OS カーネルの実行中に適用する Dynamic Patching では，App の実行を停止すること無くアップデートが適用でき，OS の障害を引き起こすようなバグを修正することも可能である．しかしすべてのアップデートで利用可能なわけではなく，コードのみ修正を行うものは得意であるものの，データ構造の修正を伴うものには対応していないこともある．

本論文では，App の実行環境を抽象化したプロセスという概念は歴史的に変わっていないことに着目し，App の実行に必要なプロセスの実行状態を *Essential Context* として定義する．Essential Context を用いることで，OS は再起動後も App の実行を継続することができる．まず，信頼できない OS 上で動作する App の Essential

Context を作成する手法，*ShadowBuddy* を提案する．ShadowBuddy は OS よりも強いハードウェア権限を持ち，OS のエラープロパゲーションから App の実行状態を保護しつつ，App と OS の挙動を監視して Essential Context を更新する．ShadowBuddy では保護のためにランタイムオーバヘッドが発生し，App のスループットが低下する．そこで OS カーネルのバグの修正のために行われるアップデート時に絞り，ランタイムオーバヘッドをなくしつつサービスダウンタイムを低減する最適化手法 *Dwarf* を提案する．両手法の提案によって，OS のソフトウェアバグによってシステム全体の信頼性が低下することを防ぐことができた．App の実行状態が破壊されることを防ぐことによって保全性を高め，ダウンタイムを短縮することによって可用性と保守性を高めることができる．両手法は仮想マシンモニタ（VMM）の Xen と広く用いられている OS の Linux に対して実装を行い，OS 再起動時にも App を継続して実行できることを示す．

目 次

図目次

表 目 次

第1章　はじめに

1.1　背景

コンピュータシステムには高い信頼性が求められている．1000 人以上を雇用する大企業を対象とした調査では，コンピュータシステムに期待する稼働率を，99.99% 以上と答えた企業は 79%，99.999%と答えた企業は 17%であった [1]．99.99%とは 1 年間のうち停止している時間が約 52 分以下，99.999%とは約 5 分以下であるよう求めているということである．あるサービスの停止時間がサービス提供者にどれだけの損失を与えるかは，そのサービスによって異なる．同じ調査では，1 時間のサービス停止によって発生する損害は，10 万ドル以上と答えた企業は 98%，30 万ドル以上と答えた企業は 81%，100 万ドル以上と答えた企業は 33%であった [1]．ホスティングサービスやクラウドサービスの提供者は，QoS を満たすことができない程の性能低下が起これば，利用者との契約に従って違約金を支払う必要もある．

本論文では OS 内の実行が停止する状況を Fault と呼ぶ．OS が Kernel Panic を引き起こした場合のほか，OS のアップデートを適用するために OS を停止させる場合も含む．また停止する状況を OS 内で以上が発生した前者の場合に限ったものを Failure と呼ぶ．

OS はコンピュータシステム全体の信頼性の要であり，高い信頼性が求められる．OS はハードウェアを管理し，App に割り当てることでハードウェア資源の効率的な利用を可能とするソフトウェアである．App は OS の機能を利用してサービスを展開するため，OS 内で Fault が起こると，App の実行が停止したり，OS や App 内にある App の実行状態が破壊されたりすることがある．App の稼働時間が短くなればサービスの可用性が低下し，App の実行状態が破壊されれば保全性が低下する．これら OS の Fault によって発生する信頼性の低下は，App のバグに関わらず発生する．

OS の Fault の原因の 1 つはソフトウェアバグである．OS のバグは日々数多く報告されており，修正されていない脆弱性のあるコードが実行されることで Failure が発生する可能性は常に存在する．OS 内に存在するバグによっていつ Failure 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いる [2]．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 [3]．Linux カーネルのソースコード量は年々増加しており，2011

年 8 月にリリースされた 2.6.36.9 では 1453 万行，2014 年 9 月にリリースされた 3.16.2 では 1888 万行，2018 年 11 月にリリースされた 4.19.5 では 2559 万行に達している．過去の Linux のパッチを遡って分析した結果，バグが導入されてから修正されるまでにかかる時間は平均 1.8 年であった [4]．また 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，バグの修正のうち 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 [5] もある．このように OS カーネルの中には今なおバグが数多く含まれており，バグを含むコードが実行されることで，いつ Failure が発生してもおかしくない．現実にバグのない OS が存在しないが，そのバグを少しでも減らすこと，バグがあっても正しく App を実行すること，が望まれている．

OS のバグによって発生する Failure の影響は様々である．OS 内の Failure は，ソフトウェアバグの含まれたコードが実行された場合に引き起こされるほか，実行中のコードにはバグが無いが不正なメモリオブジェクトを参照したこと，あるいはその両方で引き起こされる．ソフトウェアバグに分類されるものは，バッファオーバーフロー，Off-by-one エラー，メモリの解放忘れによるメモリリーク，ロックの解放忘れによるデッドロック，整数演算時の意図しないオーバーフロー，などが存在する．不正なメモリオブジェクトを参照する原因としては，実行しているコード以外でのソフトウェアバグによってメモリが破壊されていた場合や，レジスタやメモリなどでのビットフリップ，ディスクなどのストレージ上のデータが破壊される SDC，などが挙げられる．Kernel Panic 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たことを検知した場合などに意図的に引き起こされる．

OS 内で Failure が発生した場合の解決方法は OS の再起動である．本来行うべき実行フローが書き換わってしまい正しく実行できなかったり，破壊されたメモリオブジェクトを読み込んでしまうことでメモリ破壊が広がっていくエラープロパゲーションは発生したりするためである．OS の実行状態が信頼できないこのような状況では，OS の再起動が唯一の解決方法である．また既知のバグを修正するためのアップデートを行うことで，Failure が発生することを未然に防ぐこともできる．OS のアップデートを適用することは OS の再起動を伴う．OS の Failure が発生した場合にも，Failure を未然に防ぐためにも，OS の再起動が重要な役割を果たしている．

OS の再起動はその上で稼働しているすべての App を停止させ，ハードウェアを再起動させる．この時揮発性メモリ上のメモリオブジェクトは全て失われるため，OS の再起動後は App の起動を初期化からやり直さなければならない．App が実行状態を不揮発性のストレージ等に保存している場合は，App が利用するメモリオブジェクトはメインメモリに配置しなければならない．メモリ上のキャッシュを失うことで処理速度が劣化したり，再計算が必要になったりするプログラムでは，OS 再起動によるキャッシュの損失はスループット低下の原因となる．例えばメモリ上に大量のキャッシュを持つ In-memory DB では，起動後にキャッシュを構築する時

間がかかり，Facebook で利用している In-memory DB では 120GB のキャッシュを構築し直すために 2.5〜3 時間の時間を要する [6]．またメモリ上で大量の計算を行う機械学習プログラムなどでは，計算の途中結果をファイルに出力することがあるものの，処理が中断されれば最後の途中結果から同じ計算を再度行わなければならない．

OS のバグを修正するためにはカーネルのアップデートが必要となる場合があるが，OS のアップデートはマシン再起動を必要とし，再起動中のサービスダウンタイムはサービス提供者にとって損失となってしまう．OS のアップデートを行わなければ脆弱性のあるコードが残り続け，クラッシュに至るようなバグを実行してしまったり，脆弱性をついた攻撃が成立してデータの改竄やサービスが異常停止するなどといったことも起こりうる．OS のアップデートを適用するためには，脆弱性のあるカーネルの実行を停止してマシンを再起動し，修正を加えたカーネルをロードする．これにより脆弱性のあるコードが OS カーネルから排除され，よりバグの少なくなることが期待されている．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，バグの修正のうち 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 [5] もある．このような理由からアップデートを行わないユーザもおり，そのスキルや経験に関わらず 70%のマシン管理者はアップデートを行っていなかった [7] という調査もある．

1.2 本研究の動機

OS の再起動は不可欠なものであるが，再起動に長い時間を要したり，稼働していた App の実行状態を失ったりと短所もある．Dynamic Patching [8, 9, 10, 11, 12, 13, 14, 15, 16] は OS の実行中にカーネルのコードを修正することで，OS アップデート時の再起動にかかる時間を省略し，App の実行を継続することができる．しかしながら，Dynamic Patching が利用できない場合も存在する [17] ため，通常の再起動によるアップデート適用も行われている．Dynamic Patching はコードや関数を置き換えるアップデートに対しては有効であるものの，データオブジェクトを修正するアップデートではすでに作成されたメモリオブジェクトにも変更を加えなければいけないことから困難で，またマクロなどで展開されたコードがカーネル全体に散らばっている場合にはすべての箇所を変更しなければならず，アップデート適用可能な瞬間を判定することが困難になる．OS の再起動を行う場合でも，再起動の時間を短縮する手法 [18, 19, 20, 21] を用いることで，サービスダウンタイムを短縮することもできる．しかしながら，App の状態は失われてしまい，サービスの再開は各 App の実装依存となってしまう．App の実行状態を残すために Checkpoint/Restart [22, 23, 21, 17] を利用することもできる．しかしメモリ上のキャッシュなどを失った状態から App が再開されるため，初期化に時間がかかったり，再起動前のスループットに到達するまでに時間がかかったりするプログラムも存在する [6].

Failure が発生した場合にも OS の実行を可能な限り続ける手法 [24, 25] では，OS の再起動を省略し，App の実行を継続することができる．Failure の原因はハードウェアの一時的な不具合など確率的なものも含まれるという知見から，OS 内のメモリ全体を新しい OS に引き継いで処理を再実行したり，Failure の原因となったメモリオブジェクトを復元して処理を再実行したりする．しかし破損したメモリオブジェクトを再度参照してしまったり，破損したメモリオブジェクトの復元に失敗したりした場合に，同じ Failure が再発してしまう可能性もある．このような場合には利用することができないため OS の再起動を行わなければならず，App の実行状態も失ってしまう．定期的な Checkpoint を行っておき，OS の再起動後に Restart する手法 [26, 27] を用いて App の実行状態を失うことを避けることもできる．いつ発生するかわからない OS 内の Failure で OS が停止したとしても，取得しておいた最新の Checkpoint から実行を再開することで，復元後の再計算のコストを抑制している．しかし Checkpoint の頻度の高さによって，再計算のコストは減るが Checkpoint 取得によるスループットのランタイムオーバヘッドは増加傾向にありトレードオフの関係がある．OS の再起動を行ったとしても App の稼働時間を長く保つためには，小さなランタイムオーバヘッドで高頻度の Checkpoint を行うことが不可欠である．

Failure の起こりにくいカーネルを構築するために，脆弱性のあるコードを検出する手法 [28, 29, 30, 31, 32] もある．これらの手法はそれぞれ特定のバグを判定する式を定義し，定義したコードが存在しないということを定理証明ソフトウェアを使って証明する．特定のバグを検出する手法としては有用であるものの，すべてのソフトウェアバグを網羅した検査を行うことができるわけではないため，バグフリーな OS であることを証明することはできない．また証明可能なバグは証明用言語によって記述可能なものに限られるため，Buffer Overflow や Use After Free などよく知られたものについては有効であるものの，未知のバグであったり定式化が行いにくいバグに対しては利用できない．したがって特定のバグが無いことが証明された OS を使用したとしても，すべてのバグが存在しないというわけではなく，OS の Failure は発生してしまう．また新たにバグが特定された場合にも，稼働していた修正前の OS にアップデートを適用するためには再起動を行わなければならず，この再起動は Fault となる．

また OS の構成方法によって，Fault が起こったとしてもその影響範囲を狭めることもできる．Micro Kernel [33, 34, 35, 36, 12] では，ある OS サーバでの Fault が他のサーバに伝搬しないよう Isolation が行われている．しかしあるサーバで Fault が起きれば，そのサービスを使うすべての App に影響が出てしまう．特定の App 向けに特化した OS を App を同一空間にロードする Library OS [37, 38, 39, 40, 41] を使うことで，Fault の範囲をその Library OS だけに留めることも期待できる．OS カーネルを機能によって分割し，階層構造に配置した Layered Kernel [42, 43, 44, 45, 46, 47] もまた，発生した Fault を分割したレイヤに留めることができ，より特権の強いレイヤに Fault の影響を与えないことが可能である．しかしいずれの手法も App に

対しては保護が行われておらず，また Fault によって失われる OS 内のメモリオブジェクトに含まれる App の実行状態は失われてしまう．

1.3 本研究の目的

本研究では，信頼できない OS 上で App を実行する場合でも，システム全体の信頼性を高く保つ手法を提案する．提案手法では App の実行を継続させるために必要な App の実行状態，*Essential Context* を定義する．OS は長い歴史の中で App の実行環境をプロセスとして抽象化しており，それを表現する OS 内の細かなデータ構造は変わっているものの，概念としては変わっていない．具体的には，App の挙動を定義するコード，変数を格納するメモリ，計算を行うプロセッサの状態，メモリのアドレッシング，仮想化された I/O デバイスを表すファイルなどである．Essential Context はこのことに着目し，プロセスという概念を特定の OS バージョンに依存しない形式で表現したものである．App はシステムコールを通してプロセスの状態を変更させることを OS に依頼する．よってシステムコールを監視し，その結果を解析することで Essential Context を抽出することが可能である．また I/O デバイスの状態も監視することで，非同期の I/O デバイスを扱うファイルの状態をも取得することができる．

1.3.1 継続的な Essential Context の収集

OS カーネル内で Failure が発生したことを検知した時点で，すでにエラープロパゲーションによって OS 内のメモリオブジェクトが破壊されている可能性があり，また App のユーザ空間内のメモリも破壊されている可能性がある．そのため OS の Failure が発生する前から Essential Context を抽出しておく必要がある．しかしながら，Checkpoint のように App がユーザ空間に持つメモリ内容の複製を定期的に作成すると，複製に時間がかかってランタイムオーバヘッドが発生し，また Checkpoint の保存先として高いメモリオーバヘッドが発生する．

そこで本研究では，App のユーザ空間のメモリが OS の Failure によって破壊されないよう保護し，メモリ内容の複製は行わないようにする．また OS 内にメモリオブジェクトとして存在する App の実行状態は，監視しておいたシステムコールと I/O のログを利用し，OS 再起動後に再現可能にする．これらの保護と監視によって，破壊されていない App のユーザ空間のメモリ内容，プロセッサの実行状態，メモリマッピング，ファイルの状態を取得可能になり，Essential Context を抽出することができる．

OS よりも強い権限で App のユーザ空間のメモリを保護するために，ハードウェア仮想化技術の VMM を活用する．VMM から OS の権限に制約を設けることで OS が App のメモリ内容を破壊しないようにする．また App が発行したシステム

コールの引数とその結果を解析し，適切なログを残す．非同期な I/O デバイスの状態はシステムコールの時点では反映されない場合もあるため，OS が実際に発行した I/O の内容と合わせて実行状態を更新する．例えば TCP の通信を行う App があり，OS がパケットを受信して ACK を返してから App が実際にパケットデータを受け取るまでの間に OS に Failure が発生した場合を考える．パケットの送信元は ACK を受け取ったため送信バッファが残っている保証はなく，受信したパケットのバッファが確実に存在するのは Failure を起こした OS の中にのみである．App の挙動だけを監視している場合にはこの受信したが App が受け取っていないパケットについて追跡することができず，OS を再起動するとパケットのバッファを失ってしまう．OS がどのようなパケットを送信したのか，受信したのかを監視することで，OS の再揮毫後にも App の実行状態と一貫した I/O の状態を復元することができる．

本論文では OS や App を改変することなく，App の挙動を追跡して Essential Context を収集する機構 *ShadowBuddy* を提案する．App のユーザ空間のメモリ保護を行いつつ，システムコールの引数と結果を追跡して Essential Context を作成する機能を Xen 4.6.1 に実装した．システムコールの発行が少ない App ではそのランタイムオーバヘッドは小さく，実行時間の増加は 1%未満であった．またネットワーク I/O を多発する memcached でも，スループットの低下は 40%程度に抑えることができた．

1.3.2 瞬間的な Essential Context の収集

継続的な Essential Context の抽出では App と OS の挙動を監視する必要があり，平常稼働時にもランタイムオーバヘッドが発生していた．そこで OS カーネルのアップデートのみに利用範囲を制限することで，ランタイムオーバヘッドとダウンタイムをさらに短縮する最適化を考える．OS カーネルのアップデートは OS 内の Failure とは異なり，事前に予期可能なイベントである．また OS 内でメモリオブジェクトが破壊されていることは想定しないため，OS 内のメモリオブジェクトを参照して Essential Context を作成することが可能である．

本論文では，OS アップデートの直前に Essential Context を取得し，短時間でアップデート後の OS 上で App の実行を再開させる機構 *Dwarf* を提案する．App の Essential Context を抽出・再開する処理は OS 内に記述する．また VMM を導入することで，アップデート後の OS を App 実行のバックグラウンドで起動させておき，App を実行する OS を高速に切り替えることでダウンタイムを短縮する．具体的には，アップデート後の OS は VMM が起動した別 VM 上で起動し，I/O デバイス無しで起動できる段階までブート処理を行っておく．その後アップデート前の OS は App を停止させて Essential Context を収集し，VMM を経由して Essential Context をアップデート後の OS に引き継ぐ．VMM はアップデート前の OS に割り当てられていた I/O デバイスもアップデート後の OS に割り当てを切り替え，アップデート

後の OS のブート処理を再開させる．アップデート後の OS は Essential Context を利用して App の復元を行い，App の実行を再開させる．冗長なマシンやハードウェアを導入することなく，すべての OS アップデートで利用することができ，App のサービスダウンタイムを短縮することが本研究の目標である．また既存の App でも利用可能にするために，App のソースコードへの改変は必要としない．

本論文では OS 再起動による App のサービスダウンタイムを短縮する機構 *Dwarf* を提案する．Essential Context を取得・復元するモジュールを Linux 2.6.39.4 と Linux 4.1.6，OS の起動をバックグラウンドで行い，Essential Context の引き継ぎを行うモジュールを Xen 4.5.0 に実装し，実験を行った．Essential Context の作成・復元にかかるを計測したマイクロベンチマークでは所要時間はメモリサイズに大きく依存することがわかり，1GB のメモリを使うプログラムではいずれも約 20ms，12GB のメモリを使うプログラムではそれぞれ約 160ms と約 20ms と短かった．再起動中に発生したアプリケーションのサービスダウンタイムを計測したマクロベンチマークでは，通常の再起動では 26s 程度だったが，提案手法によって約 5s まで短縮することができた．

1.4 本研究の貢献

コンピュータシステム全体の信頼性を担保するために，OS の Fault に対処する，既存研究では OS アップデート時の App のサービスダウンタイムを短くするということが取り組まれてきた．しかしながら，App の復元が理論的に不可能であったり，App のサービスダウンタイムが長かったり，App の通常実行時のランタイムオーバヘッド・メモリオーバヘッドが大きかったり，App の改変を必要としたり，特定の OS 構成方法に依存したり，冗長なマシンを必要としたりしていた．そのため，実環境では利用されていなかったり，利用が制限される場合があったりしていた．本研究では，App の復元が理論的にも可能であり，App のサービスダウンタイムが短く，ランタイムオーバヘッド・メモリオーバヘッドを抑え，特定の OS 構成方法に依存せず，冗長なマシンを必要としない手法を提案する．

提案手法 ShadowBuddy は，OS を信頼することなく App のユーザ空間のメモリ内容を保護する．OS の Failure によって発生するエラープロパゲーションは App のメモリ内容を破壊することが無いことを保証し，ユーザ空間のメモリは Fault 発生直前のものを再利用する．これにより Fault 発生の前後での App の保全性を保証する．また Checkpoint でオーバヘッドの原因となっていたユーザ空間のメモリの複製をする必要はなく，ランタイムオーバヘッド・メモリオーバヘッドの両方を抑えることができる．

提案手法 Dwarf は OS カーネルのアップデートに対して最適化を行い，すべての OS アップデートの適用を可能にしつつ，App のサービスダウンタイムを既存研究以上に短縮する．Dynamic Patching では利用できるアップデート内容に制限が

あったが，本研究では OS の再起動を行うため App の互換性が保たれるすべてのアップデートで利用可能である．

本研究で提案した機構は，実際の OS や VMM 上で実装し実験を行っている．また実験には実際のアプリケーションを用いており，実用を想定して有効性を検証している．

本研究と関連研究は表 1.1 のように位置づけることができる．OS カーネル内の Failure とアップデートのすべてに対応することのできる手法は，OS の再起動 (OS Reboot) である．Fast Reboot は OS 再起動時間を短縮することでダウンタイムを削減し，Checkpoint/Restart は App の実行状態を残すことでダウンタイムを削減する．OS-level Rollback は Failure からの復帰が高速であるものの，対応できる Failure はハードウェアの一時的な故障や Concurrency Bugs など同じ処理を再実行した場合に回避可能なものに限られる．Checkpoint/Restart は OS 内の Failure でもアップデートでも利用可能であるため Fault への耐性が強いが，実行状態の保存に時間がかかるため App のスループットが大幅に低下し，復元を行う際には OS の再起動にかかる時間は App のサービスダウンタイムとなる．Process Migration は App のサービスダウンタイムが短いものの，OS のアップデートでは利用できるがいつ発生するかわからない OS の Failure に対しては利用できない．Dynamic Patching は App のサービスダウンタイム無しに実行を継続できるが，利用できないアップデートも存在する．本研究は OS の Failure から App の保護を行い，OS の再起動を行う場合に短いサービスダウンタイムで App のサービス再開を行うことができる．App の互換性が保たれているカーネルアップデートであれば大幅なメジャーバージョンアップデートでも利用でき，実際に Linux 2.6.39.4 から 3.18.35，2.6 から 4.1.6, 3.18.35 から 4.1.6 に App の実行を移すことができた．App のユーザ空間のメモリは OS による書き込みから保護されており，また OS カーネル中のメモリ破壊が起こるような Failure であっても App の復元が可能である．

1.5 本論文の構成

本論文の構成は以下の通りである．2 章では関連研究について述べる．具体的には OS の Failure 発生時の可用性・完全性の低下を抑制し保守性を向上させるための OS-level Rollback，Shielded Execution，OS のアップデート適用時の可用性低下を抑制させるための Process Migration，Dynamic Patching，そのどちらにも有効な Checkpoint/Restart，Fast Reboot，OS のコード中のバグを検出して OS 自体の信頼性を向上させる Verification，OS 構成方法によって信頼性を向上させる OS Architecture である．3 章では OS 再起動後の App の再開に必要な Essential Context について述べる．4 章では ShadowBuddy について述べる．OS の Failure からのユーザ空間のメモリ保護，システムコールに合わせた保護について述べ，実装と実験の結果を示し，その保護機構の最適化について考察する．5 章では Dwarf について

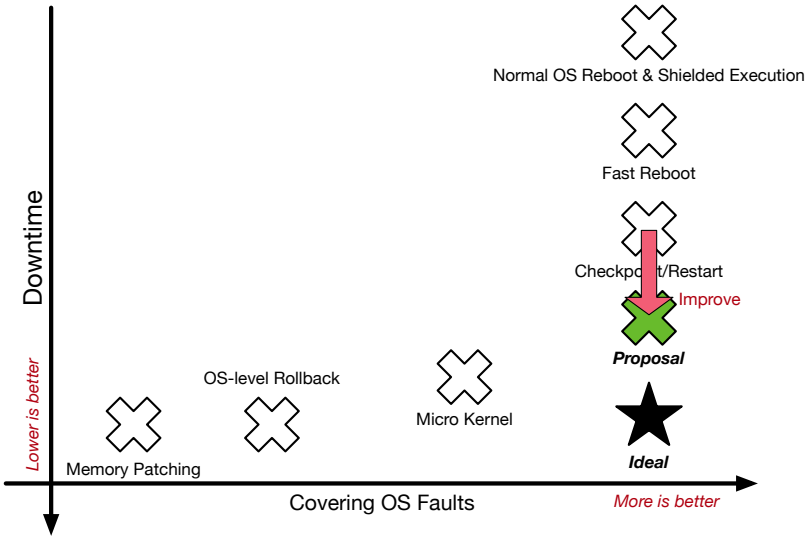


図 1.1: 既存研究からの貢献

述べる．OS のアップデート適用時に App のサービスダウンタイムを短縮させる手法について述べ，実装と実験の結果を示す．最後に 6 章で本論文をまとめる．

第2章　関連研究

本章では，OS 内の障害発生時にコンピュータシステム全体の信頼性が低下することを抑止する既存研究について述べる．Fast Reboot は OS 再起動にかかる時間を削減することでシステムの保守性を向上させる．Shielded Exexution は OS 内の障害が App のユーザ空間のメモリにアクセスし，内容を読み込んだり不正な値で上書きするすることを防ぐことで保全性と機密性を向上させる．OS-level Rollback と Memory Patching は OS 内の破壊されたメモリオブジェクトを復元することで，OS の実行を継続させることができる．Checkpoint/Restart は App の実行状態を保存しておき再起動後の OS 上で App の実行を再開させ，保守性を向上させる．Process Migration は別マシン上に App を移送することで，OS のアップデートを可能とする．Verification はプログラムのバグを検出する手法で，OS 自体の信頼性を向上させる．最後に OS Architecture の構成方法を変更することによってシステム全体の信頼性を高める方法について述べる．

2.1　Fast Reboot

OS カーネルのバグに対応するものではないが，OS のアップデートや障害が発生した後に必要となる OS 再起動を早めることによって App の可用性低下を抑止することもできる．高速再起動を実現する手法 [18, 19, 20] はいずれも，ブート処理は一般的に変化が少なくメモリ状態はほぼ一定である，という経験則を利用している．

Kexec は BIOS を介したハードウェアの初期化を省略することができ，この仕組を利用した kboot [18] はブートローダの代わりとなって新しい OS カーネルを起動することができる．Phase-based Reboot [19] はリブート処理を段階ごとに分け，それぞれの処理で得られた結果を保存し，再起動時には可能な限り再利用している．ShadowReboot [20] ではこれをログインシェルの直前まで可能とし，OS カーネルの再起動にかかる時間は通常の再起動に比べ 90%以上削減できている．

ShadowReboot では仮想マシンモニタを導入し，ある VM で動作する OS を再起動する際にはバックグラウンドで立ち上げた VM 上でブートしておく．初期化処理が終了しログイン画面となった瞬間に，元の VM 上に OS カーネルのコンテキストを上書きする．この際，ある VM 上で動作する OS カーネルにディスク装置上でアップデートを適用しておき，バックグラウンドでアップデート後の OS をブート

すれば，OS カーネルのアップデートとしても利用できる．ただしログインシェルを始め App の実行状態は引き継がれないため，App の再起動にかかる時間は App のサービスダウンタイムとなってしまう．

Seamless Kernel Updates [21] は kboot を利用して，App の状態をメモリ上に残しておくことで，若干 OS の再起動を高速化しつつ，App の状態を再現している．

2.2 Shielded Execution

App がユーザ空間に持つメモリ内容を OS から隠蔽する機構を Shielded Execution [48, 49, 50, 51, 52, 53, 54, 55, 56, 41] と呼ぶ．主に App の機密性を保証するために導入され，パスワード，暗号化鍵，個人情報といった Sensitive Data をシステムソフトウェアによる読み書きを防止するために用いられる．

VMM を用いて Untrusted OS から App 内の機密情報を保護する Overshadow [48], TrustVisor [49]，InkTag [50]，AppShield [51]，Ryoan [52] では，App と OS の遷移時に VMM がトラップし，ユーザ空間のメモリ内容を暗号化/復号を行ったり CPU の保護機構を設定したりして，OS が App の Sensitive Data にアクセスすることを禁止する．VMM を導入せず Untrusted OS から App 内の Sensitive Data を保護する Haven [53]，SCONE [54]，Eleos [55]，Graphene-SGX [41] では，App が Intel SGX を利用して自ら Sensitive Data を保護領域 (Enclave) を作成する．

Shielded Execution は App と OS の Isolation を高めることができ，OS のバグが App のユーザ空間のメモリ内容を書き換えることを防ぐ．しかし OS のカーネル空間内にある App の実行状態は保護されず，Enclave 内のメモリ内容は外部から参照不可能になり Checkpoint/Restart や Process Migration が困難になることから障害発生後に App の再実行をすることがより困難になる．

2.3 OS-level Rollback

プログラムのバグによって動作が停止したり，プログラムの実行状態が破壊された場合に，特定の実行状態に遡って再度実行を行う手法を Rollback と呼ぶ．このような手法では，Rollback を行って再実行した場合に，必ず同じ症状が起こるわけではないということを仮定している．Grottke [57] らは，同じ実行状態であれば確実に発症するバグを Bohr バグ，確率的に発症するバグを Mandel バグと呼んでおり，Rollback は特に OS カーネル内の Mandel バグに対して効果的である．

Rollback で扱うことのできるカーネルバグの種類は，メモリ上のデータオブジェクトに異常な値が書き込んだことを検知した場合である．監視対象のカーネル内のデータオブジェクトについて書き込みの変更履歴を取り，異常な値を検知した場合に適切な値まで遡って再度実行する．Rollback を行うことであたかも処理が行われていなかったかのようなメモリ状態を再現することが目標である．本節で

はRollback可能な範囲がカーネル全体であるものとデバイスドライバ内であるものとで分類する．

2.3.1 カーネル全体のバグ

Otherworld [24] はFailure発生に関与すると推定されるデータオブジェクトは全て破棄するという手法である．Rollback処理は行わないが，Failureが発生した際に全てのメモリを新しいOSにコピーして引き継いでそのOS上で実行を再開する．Failure発生の原因となったApp以外は新しいOS上で再びFailureを発生させる可能性は低いため，Failureが発生しても実行を継続させることができる．Failureが発生したコアで実行したAppを復元させることはできず，また再起動にかかる数十秒がAppダウンタイムとなるが，通常実行時のオーバヘッドは無しで利用することができる．

Otherworldの復元成功率は約95%であり，復元に失敗することもある．また復元の成功としているものも，復元処理の後一定時間稼働し続けたというものであり，コピーしたメモリオブジェクトの中に破損したオブジェクトが存在しないという保証でもない．したがって，Otherworldによって復元した場合であっても，Otherworldの仕組みを利用したことによってさらなる障害が発生する可能性がある．

Recovery Domains [25] では，DBMSで培われたRollbackの知見をカーネル内のデータオブジェクトへのRollbackに適用した．OS開発者はコード中にドメイン（Rollback可能な領域）と監視対象のデータオブジェクトを指定することで，そのドメイン内では該当データオブジェクトへの変更は元の値と書き換え後の値のペアとしてログが作成される．ドメイン内でFailureが発生した場合には，ログを元に該当データオブジェクトへの書き込みを順次もとに戻し，ログを削除して処理を再実行する．

図2.1にログの取得と解放，Failure発生時のRollbackの処理を示す．青色の四角形は各ドメインを示しており，あるデータオブジェクトの変更履歴を記録するコード領域を示している．あるドメインの途中で別のオブジェクトを監視対象に追加する場合には階層的にドメインを構築できる．より濃い色のドメインはそういった下層のドメインを表しており，下層のログは上層のドメインが持つログに統合される．全てのドメインが終了した段階で，メモリの変更履歴を実際のデータオブジェクトに反映（Commit）させ，ログを解放する．ある階層のドメインでFailureが起こった場合は，その階層のドメインが始まる直前までメモリの内容を戻せば良い．これはそのFailureを発生させた要因がその階層のドメイン内にあると推測されるためである．

この変更ログの作成によって発生するオーバヘッドは10%〜400%と監視対象の多さによって大きくなることがあるため，Reversible DomainとTransparent Domainという2つの最適化が行われている．メモリの復元処理を別の処理で記述できるドメインをReversible Domainと呼んでいる．例えばメモリ確保を行う関数kmalloc()

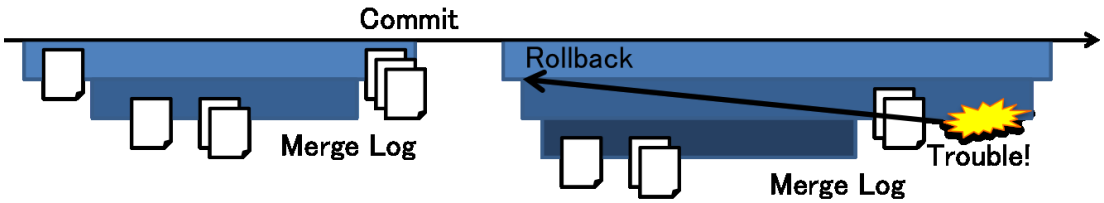


図 2.1: Recovery Domain におけるログの取得と Rollback

を実行した後に Failure が起こり，kmalloc() を行う前に遡ることを考える．kmalloc() はフリーリストからメモリ領域を管理するデータオブジェクトを抜き，使用中のフラグを立てるなどのメモリ書き込みを行う．しかし kmalloc() の処理だけをもとに戻すためには，kfree() を呼ぶだけで十分である．また，中には復元処理を行う必要がないデータオブジェクトも存在し，復元処理を記述する必要のないドメインを Transparent Domain と読んでいる．例えば LRU リストの順序変更など，最適な資源管理のためには復元処理を行った方が良いが，必ずしも復元しなくても良いデータオブジェクトなどが対象となる場合である．

実装上は OS を対象としたものではないが，ClearView [58] は App の通常実行時の挙動を監視して学習し，App 内で Failure が発生した場合に通常実行時の学習結果からメモリ上の値を書き換えて実行を再開させる．OS を監視するソフトウェアの導入と，OS の挙動解析を行うことができれば，OS-level Rollback に応用することができる．

OS 内の Failure が発生した場合に，Rollback によって可能な限り OS の実行を継続することは App の可用性を向上させるために好ましいことである．しかし OS 内の Failure のすべてを Rollback で解決することはできていない．例えば OS 内のバッファオーバーフローで OS 内のメモリオブジェクトが書き換えられ，そのオブジェクトを別のスレッドが読み込んで Failure を検知した場合，Rollback されるのは正しい挙動をしていたスレッドとなってしまう．またどこまで Rollback すれば正しく OS の実行を継続できる状態になるのかは明確な基準があるわけではない．Failure への対応を誤ったまま実行を継続すると，エラープロパゲーションが拡大し更に大きな Failure を引き起こしてしまったり，ユーザ空間のメモリに不正な書き込みを行うなどの危険も高まってしまう．またソフトウェアが意図的に値の検査を行った場合や CPU や周辺機器などのハードウェアが例外や割り込みなどによって Failure を検知した場合に限られる．値の検査ができるということは，正しいと推定される値が事前に分かっているということであり，それ以外のメモリオブジェクトに対しては利用できない．Failure の検知を行うことができずに Rollback 可能なコード領域を出てしまうと Rollback 不可能になってしまう．

以上のように，Rollback は OS 内の Failure 対策について完全な解決策であるとは言えない．Rollback に失敗した場合は App の可用性が低下し，また Failure 検知

後も不完全な Rollback によってエラーが更に伝搬する可能性もある．

2.3.2 デバイスドライバ内のバグ

FGFT [59] はデバイスドライバ内の Checkpoint を作成し，Failure 発生時に Checkpoint の内容に戻す．デバイスドライバ内の連続した処理をトランザクションのように処理できるように，一定の処理が終了するまでは状態遷移を Checkpoint 内に記録する．Failure が発生した場合は，トランザクションの先頭まで Rollback させる．

Phoenix [60] は FGFT と同様の手法であるが，主に組込みシステムの周辺機器を想定しており，ログのためのメモリ使用量を抑えつつデバイス操作を適切に元に戻すことを目的としている．デバイスの読み書きなど，ログの取得は細かく行う．プログラムのメモリ変更履歴はハードウェアのメモリ保護機能を用いて取得し，復元時にはログを逆順に辿ってメモリを復元する．適切な復元処理を作成するために，加速度センサなど外部に直接影響を与えない Stateless Peripherals，ブザーなど外部に影響を与える Ephemeral Peripherals，モーターなど追加の操作が行われない限り状態を維持する Persistent Peripherals，ディスプレイなど操作の履歴が残る Historical Peripherals に分類している．Stateless Peripherals や Historical Peripherals はデバイス状態の復元は不要だが，Ephemeral Peripherals は適切な初期状態を選んで上書きする．Persistent Peripherals はデバイス状態の変更が必要であるかを確認し，必要であれば適切な状態に上書きする．

これらの手法は Rollback の対象をデバイスドライバに限定しており，その他の OS コード領域で起こる Failure には一般的に利用することができない．

2.4 Dynamic Patching

プログラムを稼働させたまま，プログラムのコードに変更を加えることを Dynamic Patching と呼ぶ．通常はコードの変更のためにはプログラムを再起動する必要があり，特に OS カーネルのアップデートを行うための再起動は，その上で稼働する App を終了させ，再起動後の OS で再度 App を起動しなければならないことから，大きなコストとなっている．App の再起動は，App が稼働できる時間を短くし，メモリ上のキャッシュを失うことで App のスループット低下を招く．Dynamic Patching は再起動を省略することで，App の安定稼働を保証しつつ，OS カーネルが抱えるバグを修正することで OS の安定稼働という信頼性を向上させることができる．

Dynaic Patching が対応するのは主にオープンソースソフトウェアのパッチである．これはアップデートの 1 つの形態で，パッチ利用者は手元にあるソースコードに対してパッチを適用してソースコードを修正し，コンパイルする．バイナリファイルを直接修正するパッチや，バイナリファイルを置き換えるアップデート，

大幅な変更や機能追加を含むアップグレードなどは，既存の Dynamic Patching 研究では対象外である．

OS カーネルの Dynamic Patching を可能とする手法はいくつかに分類できる．本節では，Dynamic Patching 可能な OS 構成方法，VMM とカーネルモジュールを用いる Dynamic Patching について述べる．

2.4.1 OS 構成方法による Dynamic Patching

アップデート可能な OS 構成方法を採用した OS には，K42 [8, 9, 10, 11]，Proteos [12]，TTST [61]，Barrelfish/DC [13] がある．K42 と Proteos は Micro KernelOS で，サーバへの Dynaic Patching を可能としている．Barrelfish/DC は Barrelfish [62] に対していくつかの改変を加えたもので，Barrelfish がコア間のマイグレーションを行う際に App の実行状態をハードウェア操作のデータオブジェクトから抽象化するという特徴を活かし，App の実行状態を維持しつつカーネルのコードも変更できるようにしている．

OS 構成方法による Dynamic Patching では現状，Micro Kernel か Multi Kernel である必要がある．また，Micro Kernel 本体には Dynamic Patching は利用できない．このように，利用環境が限られることが欠点である．

2.4.2 VMM を用いる Dynamic Patching

カーネルのコードを変更するために，より強いハードウェア特権を持った VMM から変更を加えるのが LUCOS [14] である．OS のアップデートを内容を特殊なパッチに書き換えてコンパイルし，OS の挙動を監視して適切なタイミングでアップデート内容を挿入する．処理の切り替えは，アップデート前の関数の先頭にアップデート後の関数への jump コードを挿入して行う．関数の引数や返り値を変更する場合にはさらにその上位の関数自体も変更する必要があり，パッチのコード量が少なかったとしても Dynamic Patching 用のパッチのコード量は多くなる．

2.4.3 カーネルモジュールを用いる Dynamic Patching

カーネル自身には組み込まず，必要な時にメモリ上にロードして実行するための仕組みをカーネルモジュールと呼ぶ．通常はデバイスドライバなど，必要に応じてロードすることが好ましいプログラムに対して用いられる．カーネルモジュールを用いる Dynamic Patching では，アップデート内容をカーネルモジュールとしてコンパイルし，適切なタイミングでアップデート対象のコードへ実行フローを切り替える．DynAMOS [15] はカーネルモジュールを用いることで，LUCOS で必要となっていた VMM レイヤ無しに Dynamic Patching を可能とした．Ksplice [16]

はさらに，通常のパッチから Dynamic Patching 用のカーネルモジュールを自動生成することを可能とした．

2.4.4 Dynamic Patching 可能なタイミング

Dynamic Patching が利用可能なタイミングはいくつかの研究で言及されている．K42 は OS のコード中にアップデート可能な箇所を記述する．LUCOS は別の関数を呼び出し中である場合を含め，スレッドがアップデート対象の関数を実行している場合は Dynamic Patching を行わず，関数の実行が終わった後に Dynamic Patching を行う．これはスレッドが使用するコールスタックを遡ることで判定することができ，コード中にアップデートタイミングを明示することなく自動的なタイミング判定を行うことを可能にした．Proteos は Micro KernelOS のサーバは無限ループで動作することを利用し，ループの先頭で Dynamic Patching を行うことでプログラムの実行中にコードが変更されることを防ぐ．

2.4.5 データオブジェクトへの Dynamic Patching

データオブジェクトの変更は，カーネルのコードを差し替えるよりも困難である．LUCOS はアップデート対象のデータオブジェクトへのアクセスを検知し，適切なデータオブジェクトの変換処理（メンバの追加，削除，配置変更，値の更新など）を行う．このようなコードは通常 Transfer Code と呼ばれ，パッチとは別に記述しなければならない．Proteos はデータ構造の変更内容を LLVM を用いて判定し，メンバの削除などについては Transfer Code の自動作成を可能とした．

Dynamic Patching は App のサービスダウンタイムも 1ns 程度と大幅に削減しつつアップデートが利用でき，OS のオーバヘッドも小さい．Dynamic Patching が利用できる際には利用することが好ましい．

しかし利用可能なアップデート内容に制限があるという欠点がある．ソースコードを公開していないプログラムでは利用できず，オープンソースソフトウェアのパッチであっても全てで利用できるわけではない．データオブジェクトの変更を伴う場合にはデータオブジェクト内に入っている値の意味が理解できなければならず，不適切な Transfer Code を実行してしまうと Failure が発生し OS の信頼性を損ねる可能性もある．Dynamic Patching は変更内容がコード領域のみである場合には容易に利用できるが，データオブジェクトの変更を伴うパッチの適用に対しては信頼性の観点から適用が好ましくない．Kashyap らが Linux の kpatch を用いて Ubuntu が提供する Linux Kernel Package の Dynamic Patching を行ったところ，23 回のアップデートのうち成功は 2 回だけだった [17] ことから分かる通り，修正箇所が多いアップデートやデータ構造の変更を伴うアップデートに対しては利用できないことがある．

2.5 Checkpoint/Restart

Appの実行を停止，再開させる手法をそれぞれCheckpoint，Restartと呼ぶ．Checkpointで取得したAppのプロセスコンテキストに関する情報をCheckpointと呼ぶこともある．Checkpoint/RestartはOS内のFailureにもOSのアップデートによる再起動できるものがあり，Faultが発生してもAppを再開させることが可能である．AppのCheckpoint/Restartを行うツールはBLCR [22]，mini-ckpts [63]，DMTCP [23]，CRIU [64]などがあり，対応しているAppも多く存在する．本章ではこれらの詳細について述べ，Checkpoint/Restartを活用したOSのFault対策と，OSのバグ修正について述べる．

2.5.1 Checkpoint/Restartツール

Checkpoint/Restartツールには，ユーザレベルで行うものとカーネルレベルで行うものの2つに大別することができる．ユーザレベルで行うツールではOSの改変を行わず，Appが行うシステムコールを追跡してCheckpointを作成する．システムコールの追跡だけではカーネルが持つAppの情報の全てを追うことはできず，完全なRestartを行うことができない可能性がある．例えばプロセスIDはプロセスやスレッドの作成時にカーネルが自動決定するため，Checkpoint/Restartツールで再構築したAppのプロセスIDは元通りにならない可能性もある．カーネルレベルで行うツールではOSの改変を必要とし，AppはCheckpointの取得タイミングについて関知しない．カーネル内の情報を活用できる反面，ユーザ空間のメモリ内容などはわからないためAppが意図しないタイミングでCheckpointを取得してしまう可能性がある．BLCRはカーネルレベル，DMTCPとCRIUはユーザレベルのCheckpoint/Restartである．

Checkpoint時はAppが持つCPUのレジスタ，メモリ領域（開始アドレス，長さあるいは終了アドレス，読み書き実行の属性など），展開したファイル（ファイル識別子，ファイルパス，読み書き実行の属性，読み書き位置など），といった情報を収集し，Checkpointに追加，更新していく．メモリ領域をunmapしたり，ファイルをcloseした場合は，Checkpointから削除する．Checkpointの保存先は主にファイルで，特にメモリ領域はプロセスのforkを行うことでメモリ上に残したり，ファイルにマップしてディスク上に残したりする．

Restart時は取得していたCheckpointからプロセスコンテキストを再構築する．プロセスの親子関係を修復した後，メモリ領域をCheckpoint時のものに上書きし，ファイルを再展開し，Appの実行を再開させる．

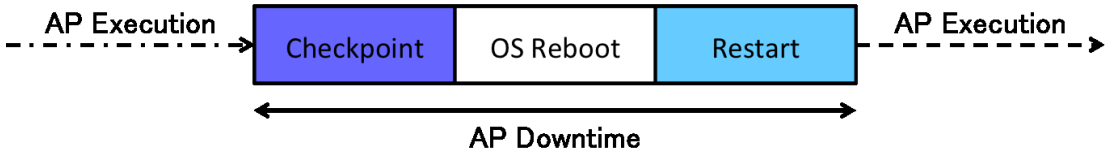


図 2.2: OS カーネルのアップデートと Checkpoint/Restart

2.5.2 カーネルアップデート時の利用

カーネルアップデート時には OS の再起動が行われるが，稼働していたプログラムの実行状態が失われるという欠点がある．これを克服するために，Seamless Kernel Updates [21]，KUP [17] では，アップデートのための再起動直前に Checkpoint を行い，再起動後のカーネル上で Restart を行う．Seamless Kernel Updates はカーネルレベル，KUP はユーザレベルの Checkpoint/Restart である．

ユーザレベルの Checkpoint/Restart は，API が変化しない OS アップデートで利用できる．対してカーネルレベルの Checkpoint/Restart は，カーネル内のデータ構造変更に Checkpoint/Restart ツールが対応しなければならず，原理的には対応できるアップデートはユーザレベルの Checkpoint/Restart より少ない．

Checkpoint の取得回数はアップデートごとに 1 回で良く，一般に Checkpoint 取得時間よりも OS 再起動にかかる時間の方が長いため，Checkpoint 取得にかかるコストより再起動時間を短くする焦点が置かれている．Seamless Kernel Updates と KUP はいずれも kexec という Linux の高速再起動ツールを用い，それぞれ 10 秒程度と数秒程度のダウンタイムを達成した．

2.5.3 Failure 対策での利用

プログラムの動作が停止したり，プログラムの実行状態が破壊されることがある．このような場合に，事前に取得していた正しい実行状態を用いてプログラムの実行を再開させることができる．Failure が発生した後に App の状態を直前の Checkpoint の状態に戻し，App の実行をを再開させる．Checkpoint を取得した時点から Failure が発生するまでに実行された App の処理は，Restart 後に再度実行される．Bascule [26]，LMC [27]，OSIRIS [65]，Remus [66, 67]，Tardigrade [68] などがこれに該当する．

Checkpoint の取得時にはオーバヘッドが発生することは避けられず，高頻度な Checkpoint を行っても App のスループットを低下させないことが課題である．実装上のコストが低いことから，いくつかの Checkpoint/Restart ツールでシステムコールの fork を行い，実行用と Checkpoint 用のプロセスを作成する方法がよく用いられている．ハードウェアのページング機能を利用して，更新されたページを追跡してページ単位でメモリの Checkpoint の更新する．これに対し，App がレジスタ

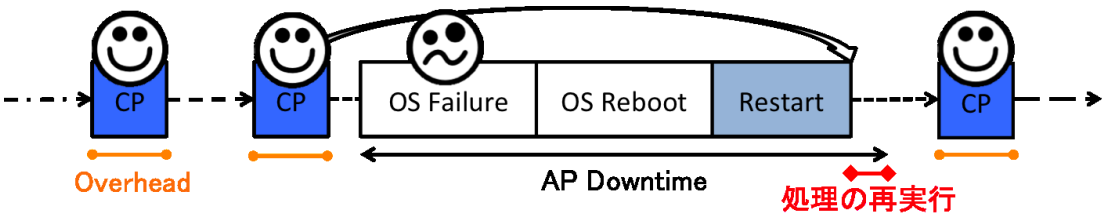


図 2.3: App 実行中の Checkpoint と OS 内の Failure 発生時の Restart

の値をメモリに書き込む命令を追跡して更新範囲単位でメモリの Checkpoint を更新する手法 [27] もある．また Checkpoint の保存先を不揮発性ディスク上のファイルとした場合，メモリのデータ書き込み速度よりディスクの書き込み速度の方が一般的に低速であるため，オーバヘッドはさらに増加する．不揮発性のディスクではなく不揮発性メモリ (Non Volatile Memory) を用いる手法 [69, 70] も存在する．ディスクに比べて NVM は高速であるため，より高頻度の Checkpoint を行ってもオーバヘッドを抑制することができるが，やはり高頻度の Checkpoint は App のスループットを大幅に低下させる．

App 内のソフトウェア Failure を想定した Checkpoint 手法 [27] では，OS 内で Fault が起こった場合には復元できない場合がある．たとえば Checkpoint の保存先をメモリ内としている場合には，OS を再起動するとその Checkpoint 内容は消滅する．

App の Restart はユーザによる手間のかかる処理である．ネットワークやディスクなど非同期の I/O を利用したサービスを提供する App を Restart してしまうと，I/O の最後に Checkpoint を取得してから Failure が発生するまでの間に発生した I/O が実際に処理されたのか，あるいは OS 内にバッファとして残り続けているのかを判断しなければならない．

既存の Checkpoint ツールは OS の Failure から完全に守られているとは言えない．OS 内で発生したエラーのプロパゲーションによってユーザ空間のメモリが破壊される可能性がある．例えば Linux はカーネル空間とユーザ空間が同一のメモリ空間内にあり，さらにカーネル内にはすべての物理ページをページテーブルにマッピングしたストレートマッピングが存在する．このようなアドレスを通して，カーネルの実行中にバッファオーバーフローなどで直接ユーザ空間のメモリに書き込みが発生する可能性がある．さらに Checkpoint は不揮発性のディスクに書き込むことが一般的であるが，ファイルに書き込む際には OS のファイルシステムに依存するため，Checkpoint が破損する可能性もある．

App の Checkpoint ではなく，OS と App の両方を Checkpoint の対象としたものは VM Snapshot と呼ばれる．Xen [45] や VMware [44] など数多くの VMM が標準で対応している機能である．VMM という OS の外部から Checkpoint を作成するため，OS 内で Failure が起こっても Checkpoint が破壊されることはない．しかしなが

らAppのCheckpoint/Restartに比べ，(1)メモリオーバヘッドが大きい，(2)Appのスループットが低下する，(3)Restart後に同じFailureが発生しやすい，(4)OSアップデートでは利用できない，といった欠点がある．マシンの電源喪失やハードウェアの故障などを想定し，バックアップ先として別のマシンを用意する手法は特にVM Replicationと呼ばれ，Remus [66, 67] やTardigrade [68] が挙げられる．VM ReplicationではVM Snapshotの欠点に加え，(4)冗長なマシンを必要とする，という欠点を持つ．

メモリオーバヘッドが大きい

VM SnapshotではVMの状態を保存するため，Checkpointを取得したいAppのメモリ内容に加え，OSや他のAppのメモリ内容もまたCheckpointの保存対象となっている．これを抑制するために，VM内でCheckpointが必要なメモリページを識別することでオーバヘッドを削減する取り組み [71] もある．しかしCheckpointのサイズがAppだけを対象とする場合に比べて大きくなることには変わらず，AppのCheckpoint/Restartに比べてメモリオーバヘッドが大きくなりがちである．

Appのスループットが低下する

VM Snapshotがランタイムオーバヘッドとなる主要な原因は二つある．一つは先述の通り，VM Snapshotのサイズが大きくなり，保存しなければならないデータ量が増えるためである．Checkpoitのために CPUリソースやメモリリソースが多く使われ，実行中のVMのCPU・メモリリソースと競合する．

もう一つは，I/OのCheckpointを行うためにI/O内容を一時的にVMM内に保留するためである．I/Oを保留せずにCheckpointを行いFailure発生後にRestartすると，すでにFailure発生前に発行されたI/Oと同じものが発行されてしまい，誤作動の原因となるためである．特にネットワークI/Oでは，送信されたパケットは取り消すことができず，送信相手にACKを返してしまった受信パケットは復元することができない．そのため，Checkpointを取得してから次のCheckpointを取得するまでの間に発生したI/Oは一時的にVMM内に保存され，次のCheckpointの取得が始まってからI/O処理が開始される．Checkpointの取得頻度が低いとI/Oのレイテンシが増加し，取得頻度が高いとCheckpointのオーバヘッドが増加するという欠点がより顕著である．

Restart後に同じFailureが発生しやすい

VM SnapshotではOSのメモリイメージを再利用するため，Otherworld [24] と同様同じFailureが発生しやすい．Checkpointを取得した時点でメモリ破壊が発生していたがErrorやFailureにはなっていなかったという場合には，Restart後に破壊

されたメモリを参照した時点で Failure が発生する場合があるためである．また破壊されたメモリを参照してさらに別のメモリを破壊するなど，エラープロパゲーションによってさらに被害が拡大する可能性がある．OS の再起動を行っておけば，このような破壊されたメモリを再利用することによる被害を防ぐことができる．

OS アップデートでは利用できない

VM Snapshot では OS と App が一体となっているため，OS のアップデートでは利用することができない．この原因には，App が使っているユーザ空間のメモリ内容を効率よく取得する方法がないこと，OS 内に存在する App の実行状態の Checkpoint を取得できないこと，が挙げられる．VMM からは一般的に，あるメモリ領域が App によって使われているのか OS によって使われているのかを判断する方法はない．Intel 社製の CPU においてはページテーブルエントリに存在する U/S フラグを用いて，App が利用しているページなのか OS が利用するページなのかを判断することは可能である．しかし OS がこの機能を利用している場合に限られ，またすべてのページテーブルをくまなく参照することはコストが大きく大きなオーバヘッドとなってしまう．さらに，OS 内に存在する App の実行状態は OS を再起動した時点で失われてしまう．VMM からは OS 内のメモリオブジェクトがどのような意味を持ち，何に利用されているのかは一般的に判断できない．Linux には DEBUG_KERNEL というコンパイルオプションがあり，これを有効にした上でコンパイルしたイメージと System.map というカーネル内のシンボルテーブルを用いれば，これを解決することは可能である．しかし再起動後の OS 内にそれらのメモリオブジェクトを適切に展開すること，OS アップデートの内容を把握してデータ構造の変更にも対応することは現実的ではない．

2.6 Process Migration

OS アップデート前にその上で稼働している App を別マシン上の OS に移送しておく [72, 73, 74, 75] ことで，OS のアップデート中も App の実行を継続する事が可能である．OS アップデート以外にも，マシンメンテナンス [73] でも利用可能である．Microvisor [74] では OS のアップデートにしぼり，VMM を導入して同一マシン内で VM を立ち上げて App を移送することでより高速な Process Migration を実現しているが，40s 程度のダウンタイムが発生する．Intel SGX などの Shielded Execution を利用している場合，OS や VMM から App のユーザ空間のメモリを取得することはできず，Checkpoint が失敗してしまう．SGX Migration [75] では Intel SGX によって保護された Enclave 内からチェックポイントを作成することで App のマイグレーションを可能とした．また近年 docker [76] コンテナのマイグレーションが Checkpoint/Restart ツールの CRIU [64] を利用することで可能となっている．しかしコンテナを停止して Checkpoint を取得し，ネットワークを経由して Checkpoint

イメージを転送し，別マシン上で起動するといういわゆる Stop-and-Copy となっているため，多くのメモリを使う App がコンテナ内で稼働しているほどダウンライムが長くなってしまう．VMM の VM Migration ではメモリ内容の転送と App の実行を同時並行で行う Live VM Migration [77, 78] が実装されており，数百 MB のメモリを持つ VM であっても数百 ms 程度のダウンタイムを達成することができているが，Container や Process Migration では実現できていない．

2.7 Verification

定理証明ソフトウェアを用いて証明することを Verification（検証）と呼ぶ．特に特定の脆弱性が存在しないことや，動作要件を満たすことを証明することを Formal Verification（形式的検証）と呼ぶ．証明する論理次第では，ロックの解放忘れや初期化していない変数を読むなどといったことがあるかを発見したり，一定 CPU サイクル以内にプログラムが終了するか，開発者が意図した結果を返すか（Functional Correctness）を検証したりすることも可能である．ただし，証明する内容は論理式で定義できなければならない，証明を行うためにはいくつかの仮定を置かなければならない，プログラムの構成に制約を付けなければならない，証明が現実的な時間で終わるために証明範囲を限定しなければならない，などといった場合が多く，万能ではない．特定の脆弱性に対して証明を行うことができれば，強力なデバッグ方法である．

定理証明ソフトウェアを用いずに，コードを検証する方法もある．Engler らはポインタは NULL ではない，unlock するときはロックが取られているなどといった暗黙の了解を用いて，システムソフトウェアのコード検証を行った [79]．こういった規則はコード上の欠陥を発見するために用いられており，Verification でも使われていることがある．

しかし検証できる内容が少ないこと，その検証範囲が狭いことは Verification の欠点である．seL4 も CertiKOS も Micro KernelOS のカーネルのみを対象としており，Ironclad は Linux のネットワークスタックのみを対象としている．検証対象が狭いことは，信頼性が保証される範囲も限定的であるという意味であり，App やマシン管理者にとっての恩恵は多くはない．検証範囲が狭い原因は主に，証明スクリプトの記述量や証明にかかる時間が増加し現実的でなくなること，検証を行うための制約をかけられない領域が存在することである．

2.7.1 定理証明ソフトウェア

証明内容の記述，検証対象のプログラムの記述はそれを得意とする言語で記述することが多い．証明の前提となる公理を選んで証明する内容を記述すれば，ソフトウェアが自動で証明を行う．定理証明ソフトウェアで有名なものは Agda，Isabelle,

Coq などがある．各ソフトウェアは証明スクリプトの記述のために複数の言語に対応していることが多く，Gallina，pCic，Vernacular，Isar といった言語から適切なものを選び，組み合わせて使うことができる．

公理と定理の間にある差を埋めるために証明スクリプトを記述する．前提とする Assumption，そこから当然に導かれる結果 Corollary，補助命題 Lemma などを記述して，常に満たさなければならない条件式 Invariant や定理 Theorem を検証する．何を証明するのか，どこまでを前提条件とするのか，といったことが研究の焦点となる．Invariant は主にバグ検出のために，Theorem は主に信頼性の保証のために用いられる．

2.7.2 OS Kernel

Functional Correctness を証明した seL4 [28, 29] や Jitk [30]，Hyperkernel [80]，デッドロックやスタベーションなどがなく並行して実行される全てのタスクが必ず実行されること（Concurrency）を証明した CertiKOS [31]，データリークやバッファオーバーフローなどがないネットワーク通信を証明した Ironclad [32] などが存在する．

seL4 は OS の実装をトップダウンに行い，トップレベルで Functional Correctness を検証している．まず Isabelle/HOL で OS の関数に関する抽象的なプロトタイプ宣言（関数が何をするのか）を行い，次に Haskell で条件の判定などを含めたコーディングを行い，最後に C 言語で詳細な実装を行う．検証のために Haskell から Isabelle/HOL に機械翻訳し，ハードウェアシミュレータと合わせて Isabelle/HOL での抽象的なプロトタイプ宣言と挙動が一致するかを検証する．したがって実装のためのコーディング上のバグが無いという検証ではなく，Haskell での実装が Isabelle/HOL でのプロトタイプ宣言と一致するかという検証を行っている．

また C 言語で書かれた OS が実際にコンパイルされた時，機械語での記述と元の C 言語との Functional Correctness を保証した CompCert [81, 82] というコンパイラが存在する．一部の研究ではこのコンパイラを利用することで，C 言語で実装した OS カーネルがコンパイル後も Functional Correctness を保っていることを保証している．

いずれの手法も，特定の OS の実装や条件に絞って Functional Correctness や Concurrency を保証しているため，広く用いられている Linux や Windows といった OS に対しては適用できず，ソフトウェアバグのすべてを検知することもできていない．検証対象はほとんどが Micro Kernel のコアや特定のサブシステムに絞られており，OS 全体の検証は行われていない．また，各研究において達成された証明内容は前提条件が異なっており，他のものと組み合わせることができない場合もある．

2.7.3 File System

ファイルシステムを対象とした検証も行われている．Functional Correctness でゼロクリア忘れや復元不可能な書き込みを排除した [83]，事前に収集した FS 挙動の特徴分析から，許される挙動を特定し他のファイルシステムに対して証明を行う [84] などが存在する．これらは正しいディスク操作の定義と，実装上の手続きの定義を比較することで，実装されたコードの Functional Correctness を検証している．

これらの研究は OS のサブシステムであるファイルシステムのみを対象としており，OS 全体に適用できるものではない．ファイルシステムの挙動はその他の OS 全体とは大きく異なっており，バグの無い OS を保証することのできる Functional Correctness は未だに定義されていない．

2.7.4 最適化

証明対象のプログラムよりも証明用スクリプトの方が長くなることもあり，証明用スクリプトを自動生成することや，短い記述で多くの証明を行うことが求められている．例えば seL4 で検証対象とした OS カーネルは Haskell で 8,700 行であるのに対し，証明用スクリプトは合計 165,000 行である．CertiKOS で検証対象とした OS カーネルは C 言語とアセンブラで 6,500 行であるのに対し，証明用スクリプトは 90,000 行である．証明用スクリプトの量を減らすことで，証明用スクリプト内にバグが存在することを防ぐ必要がある．

Micro-grammer [85] はプログラミング言語の文法規則を利用してスクリプトの一部をワイルドカードで記述できるようにし，証明用スクリプトの行数を従来の 100 分の 1 程度に削減しつつ既存の商用ツールよりも多くのバグを検出した．Bedrock [86] は証明用に使う Invariant をいくつかの仮定から自動生成した．Yggdrasil [87] はディスクの状態には一貫性があることに着目し，正しい状態を定義する Invariant を自動生成した．

これらの最適化はプログラミング言語の文法やハードウェアの使用などといった Assumption や Corollary を活用している．あらゆる場合に利用できるものではないが，証明用スクリプトの記述量を減らすことは実装コストの削減とスクリプト内へのバグ混入を防ぐことにつながり，より信頼性の高い検証を可能とする．

2.8 OS Architecture

OS 構成方法は，Monolithic Kernel，Micro Kernel，Layered Kernel，Library OS，Multi Kernel，に大きく分類できる．図 2.4 にその概念図を示す．様々な App を高速に動作させることが可能であるのは Monolithic Kernel であるが，Failure に弱いという欠点がある．Fault の影響範囲を狭めることができるのは Micro Kernel や

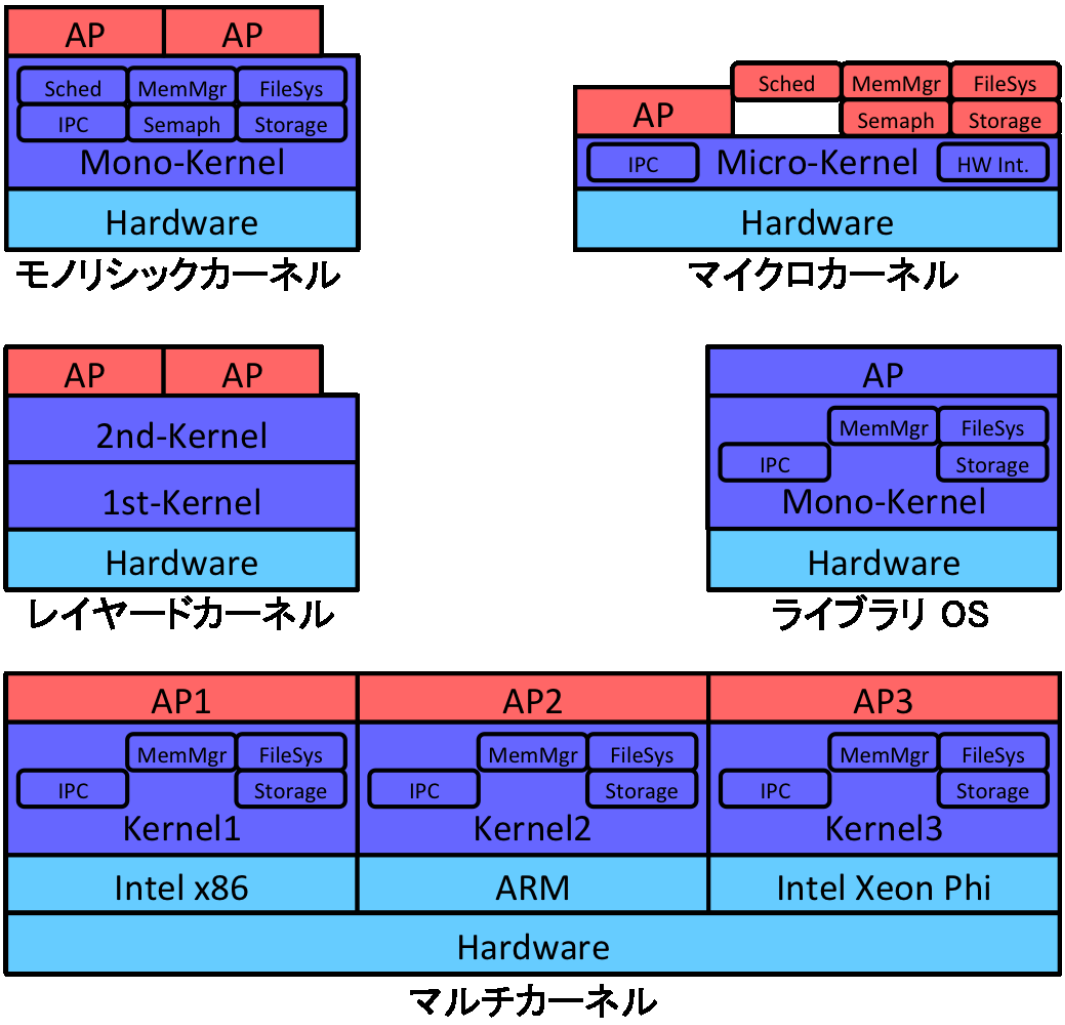


図 2.4: 各カーネルの構築方針

Layered Kernel であるが，発生する Monolithic Kernel に比べオーバヘッドが大きく App の処理性能を低下させがちである．特定の App に特化してその性能を最大限に高めることができるのは Library OS であるが，効果的に利用できる App や実行環境は多くない．Multi Kernel は特定の App のスループットやスケール性を向上させる可能性があるが，Library OS 同様効果的に利用できる App や実行環境は多くない．Library OS と Multi Kernel は App ごとに OS の状態を持つため，App や OS 内の Fault が他の App には影響しない．

2.8.1 Monolithic Kernel

最も原始的な OS 構成方法で，OS が提供する機能（コンポーネント）を全て単一のプログラムで実現する設計方針である．OS は特権的なモードで実行され，マ

シン全体を適切に管理しながら各 App に対して資源を割り当てる．Linux，古い Macintosh(MacOS 9 まで) や Windows(Me まで) などがこれに該当する．モードの切替やコンテキストスイッチが少ないため高速な動作が期待でき，App の処理性能を最大限に活かすことが可能である．

カーネル内の脆弱性には弱い．必ずしも特権を必要としないプログラムも特権モードで実行されるため，1 箇所でもバグや脆弱性が存在すると OS 全体に不具合が伝播する可能性があり，OS 全体の信頼性が損なわれる．

2.8.2 Micro Kernel

特権を持った状態で実行するプログラム量をなるべく減らそうという設計方針である．コンポーネントをサーバという単位に分け，特権を必要としない部分は非特権モードで実行する．サーバ同士はハードウェアの保護機能などを利用して分離させ，あるサーバでの不具合が他のサーバに直接影響することはない．L4 [33, 34], Minix [35]，CuriOS [36]，Drawbridge [37] などがこれに該当する．

OS が管理する資源を App が利用する時はサーバに要求を送るが，サーバが実行されるためにコンテキストスイッチが発生することが多く，原理的に Monolithic Kernel より実行速度が遅いという欠点がある．スイッチングコストを下げる研究が長く続けられた結果このコストは減少したが，同じ機能を持つ場合に原理的に遅くなることは避けられない．

OS のサーバは App と同じように実装されているため，サーバが持つ実行状態を適切に保持することができれば，Failure が発生した場合でも該当サーバのみを再開すること [36] や，サーバのアップデートによって Fault した場合でも利用可能 [12, 61] である．

最近の Macintosh(MacOS X 以降) と Windows(Windows NT 以降) は主要な機能はカーネル内で実装しそれ以外をサーバとして実装することで，オーバヘッドを抑えつつ Monolithic Kernel と Micro Kernel の良い所を併せ持っている．設計方針としては Monolithic Kernel と Micro Kernel の中間と言える．

Micro Kernel に依存する信頼性向上手法はいくつかある．例えば 2.7.2 で述べた seL4 [28, 29] は，証明対象を Micro Kernel とすることで証明範囲を狭め，証明を可能としていた．また Proteos [12] は Micro Kernel のサーバがループ構造を持っていることに着目し，ループ部分ではサーバの主要な機能が実行されておらずアップデートに適したタイミングであることを利用していた．

2.8.3 Layered Kernel

複数のカーネルを階層構造に配置し，ハードウェア管理，保護機能，API の提供などを複数のカーネルで分担する設計方針である．Micro Kernel と似ているが，それぞれの OS の機能が独立しているという点で異なる．下層のカーネルほど高い特

権を持つ．下層のカーネルが仮想マシンモニタ（VMM）機能を持ち，上層のカーネルはハードウェアインタフェースを通して仮想マシン（VM）を操作する方式のほか，特殊なインタフェースを通してカーネル間の通信を行う方式がある．信頼性向上のために階層構造を作もの [43, 37, 88, 42, 89] と，ハードウェアの利用効率を上げるために階層構造を作るもの [44, 45, 46, 90, 91, 76] に分類することができる．後者はさらに，OS 資源の仮想化を行う Container [92, 90, 91, 76] と，ハードウェアの仮想化を行う VMM [44, 45, 46] とに分類することができる．また VMM の上に別の VMM を配置する Nested Virtualization [93, 47, 94, 95] によって，資源管理をより柔軟にすることもできる．

このような構成方法が登場した背景の一つには，OS が豊富な機能を提供すればするほど OS のコード量が増えて複雑になり，脆弱性も増えてしまうというトレードオフが挙げられる．上層のカーネルは App に対して豊富な機能を提供しつつ，下層のカーネルはハードウェアを適切に管理することに専念するという役割分担により，上層のカーネルでの脆弱性がマシン全体に及ぶことを防ぐことができる．同じ OS 上で実行する必要のない App を別の OS 上で実行し，OS で発生した Fault の影響範囲を狭めることができる．

初期の Container は主に単一のカーネル上で動く複数の App 間の Isolation を強め，ある App の資源を他の App と分離することで機密性を向上したり，ある App が他の App のパフォーマンスを低下させることを防ぐために用いられた．また近年の Container は VMM に対して高速に動作する [96] という点で再び注目されている．ハードウェアの仮想化を行う VMM のランタイムオーバヘッドが大きく，また同一マシン上で複数の OS カーネルが動作することによりメモリオーバヘッドも大きいためである．複数の Container が同一カーネル上で動作するため，カーネルの脆弱性をついた攻撃が成功すると，ある Container から別の Container に攻撃することができるという欠点を持つ．

OS のバグによって App が持つメモリを直接書き換えたり，メモリ内容を漏洩させることを防ぐこともできる [48, 53]．他の階層には修正を加えること無く，保護を行う階層がハードウェア保護機能を利用して App のメモリ内容の保護を行うことができる．OS の実行中は App のメモリ保護を行い，App の実行中は保護を解除する．しかし保護機構のランタイムオーバヘッドによって App のスループットは 50%以上低下し，スループットを重視する App では利用することができない．

Layered Kernel で用いるそれぞれのカーネル構成方法には特に制限が無いため，階層構造にするためのインタフェースに対応すればどのようなカーネルであっても利用でき，App にも制約はない．しかしながら，階層が増えれば増えるほどオーバヘッドは大きくなり，原理的には Monolithic Kernel や Micro Kernel よりもオーバヘッドが大きい．

2.8.4 Library OS

OS は App に特化した機能を持ち，App の性能を最大限に高めることを目的とした設計方針であり，Layered Kernel の一種と位置づけることもできる．App と OS を 1 対 1 で対応付けることで，その App が利用しない OS コンポーネントや条件分岐を削除したり，その App にとって最適なコンフィグを適用することが可能となる．サーバ App では単一マシン上で単一 App しか利用しないという場合も多いため，特にサーバ App を想定した Library OS が多い．Drawbridge [37]，Mirage [38]，OSv [39]，Graphene [40, 41] などがこれに該当する．

特定の App に特化した Library OS を構成するために，App が利用する API を分析し，OS 各機能のコンフィグを指定する．この際，不要な OS コードは削除することができ，条件分岐の削除による高速化とカーネルのバイナリを削減することによるメモリ効率の向上が可能である．またコード量が減ることでバグも減ることが期待できる．App と Library OS はリンクされ，起動時には単一のイメージとしてロードする．コードの追加を禁止することで，コードインジェクション攻撃を防ぐことができる．

特定の App に特化するため利用できる環境が限られるが，原理的には App のスループットを向上させることが可能である．プロセス間通信は必ずネットワークを経由しなければならないため低速になったり，複数の App でファイル資源を共有する場合にはオンラインストレージが必要になったりと不便な点も多い．適切な運用ができない場合には，マシンリソースの利用効率が落ちたりスループットも低下したりするため，どのような環境でも利用可能であるとは言い難い．

App を割り当てるマシンは VMM が提供する VM であっても良いため，ハードウェアの多重化は VMM が行い，Library OS が App に対して抽象化したコンピュータ資源を提供する，という利用方法も可能である．これは前節で述べた Layered Kernel とのハイブリッドである．近年の Library OS はクラウドコンピューティングの普及に合わせて VMM 上での動作を想定しており，Drawbridge，Mirage，OSv はいずれも VMM 上での動作を想定している．

2.8.5 Multi Kernel

ヘテロジニアスマルチコア環境でのスケール性を向上させ，最適なコア上に処理を移送することのできる設計方針である．CPU コアには処理速度や消費電力，価格などのトレードオフが存在する場合がある．例えば，処理速度は速いが消費電力も高いという CPU と，処理速度は低いが消費電力も低いという CPU を組み合わせることで，スループットが求められる処理は前者の上で，求められない処理は後者の上で実行することで，全体として電力消費を抑えつつ高速に実行することが可能となる．このように特徴の異なる演算装置を組み合わせた環境をヘテロジニアスマルチコアと呼ぶ．演算装置には CPU アーキテクチャの他，GPU などの

コプロセッサなども含まれる．Multi Kernel では演算装置のコアごとに OS と App を実行するため，同一マシン上で複数の OS カーネルが実行される．Barrelfish [62] がこれに該当する．

演算装置のコアを変更するためには，App の動作を維持しつつ CPU やメモリの状態を別のコアに翻訳する．このような処理をマイグレーションと呼び，マイグレーションをするためにかかるダウンタイムがその後 App のスループット向上よりも小さい場合に，マイグレーションによって App の処理性能を向上させることができる．App の処理内容の特徴が途中で変わるような場合には，処理の特徴が変わるところでマイグレーションを行えば，App 単体の全体のスループットを向上させることができる．例えば，並列性の少ない処理を行うフェーズはクロック数が高い演算装置上で実行し，並列性の高い処理を行うフェーズではコア数が多い演算装置上で実行することなどが考えられる．

またマイグレーションが可能であるということは，ハードウェアの操作を行うデータオブジェクトが OS や App の実行状態から抽象化されているということを意味する．この特徴を活かし，抽象化されたハードウェア操作のデータオブジェクトを元に App の実行を継続しつつ OS のアップデートを実行することも原理的に可能である [13].

Multi KernelOS は様々なアーキテクチャに対応しなければならず，挙動も複雑になることから一般的に信頼性は低い．Multi Kernel では他アーキテクチャへのマイグレーションを可能とするために，アーキテクチャ間でハードウェアの操作オブジェクトを翻訳する必要があり，翻訳ができない場合はその機能の利用かマイグレーションのどちらかを諦めなければならない．また，Multi Kernel はヘテロジニアスマルチコアにおいて複数の App を同時に実行した際に全体のスループットを向上させることを目的としているが，実行する App が少ない場合には App が特定のコアで固定されることとなり，最初からそのコアを持ったマシン上で実行すれば良い．コンソリデーションに対してスケール性が高いが，特定の App のスループットを向上させるというわけではない．

2.9 その他

デバイスドライバなどカーネルモードで実行されるプログラムについても保護する仕組みが必要である．バグ修正について調査した結果デバイスドライバのバグはカーネル全体の 7 倍である [4] ことや，Windows XP の Failure 原因の 85%はデバイスドライバである [97] など，デバイスドライバのバグが原因となる Failure は多い．例えデバイスドライバでバグが発生したとしても，その Failure による影響をカーネルや App など他のプログラムに波及させないことが好ましい．Nooks [97, 98] や Shadow Driver [99, 100] は，ハードウェアの保護機能を利用してデバイスドライバのコードが他のメモリ領域に書き込むことができないような保護を行っている．このような仕組みは Library OS 以外の全てのカーネル構成方法で利用できる．

よく使われるコードはバグが少ないという仮定を元にカーネルモード中により信頼性の高いコードのみを実行することでOSの信頼性を高めることもできる．Lock-in-Pop [101]では，Appのコードを変換するNaCl [102]とRepy SandBox [103]を利用して，Appの実行をOSカーネルのうちよく利用されるコードのみに限定している．OS内にはバグが残り続けるが，バグが実行されてFailureとなることを防ぐことができる．

CPUアーキテクチャの保護機構を利用してAppとOS間のIsolationを高め，OSの信頼性を高める手法は他にもある．VMI(Virtual Machine Introspection)とはVMMからAppのシステムコールを監視してOSへの攻撃検知を行う機構のことである．Ether [104]やShadowContext [105]はIntel VTを利用して監視対象のAppやOSを改変すること無くAppの不審な挙動を追跡することができる．SIM [106]は監視対象のVM内で複数の仮想アドレス空間を持つことでIsolationを行い，監視システム側の機密性を保ちつつ，監視によって発生するランタイムオーバヘッドを抑制している．Extended Page Table(EPT)を用いてAppからOS内の機密情報を保護するEPTI [107]では，Appが実行されるEPTとOSが実行されるEPTを分離してAppが実行されるEPTにはOSの物理アドレスを参照不可能にし，CPU投機的実行を利用してAppがカーネル空間のメモリ内容を推測するMeltdown [108]攻撃を防ぐ．OSがCPUの設定を行うことで，OSがユーザ空間のメモリにアクセスすることを防ぐこともできる．Intel SMEP(Supervisor Mode Execution Protection)ではカーネルモード中にユーザ空間のメモリを実行することを防ぎ，Intel SMAP(Supervisor Mode Access Protection)ではユーザ空間の読み書きを防ぐことができる．予め仮想アドレスのアクセス範囲をレジスタに設定しておき命令実行中に範囲外を参照した場合にCPUが検知するIntel MPXは，カーネル空間内でのバッファオーバーフローの検知に用いることができる．いずれもOSのバグによって意図しない挙動をした場合に検知することができるが，これらの設定はOSが行うことであるため迂回することができ，信頼できないOSではAppのメモリ空間の保護を保証することができない．

2.10 まとめ

本章で述べた通り，OSのソフトウェアバグによって引き起こされるAppの信頼性低下は，様々な手法が提案されているものの解決がされていない課題である．各関連研究によって解決されていること，解決されていないことを表2.1にまとめた．Fast Rebootはマシン・OSの再起動を高速化するものであるが，Appの実行状態を失うため別途Checkpoint/Restartなどと組み合わせなければならない．OS-level RollbackはOS内のFailureが発生しても，OSの実行を継続可能な状態まで戻すことでOSの再起動やAppの状態損失を防ぐことが可能である．しかしRollbackはすべてのFailureから正しく復元できるわけではなく，不完全なRollbackによって更にメモリ破壊が進む可能性もある．またOSのアップデートに対しては利用

表 2.1: 既存研究と本研究の比較

	App の保護	App の実行継続	ダウンタイム	対応可能な Fault
OS Reboot	×	×	×	✓
Fast Reboot	×	×	×	✓
OS-level Rollback	×	✓	✓	×
Shielded Execution	✓	×	×	×
Checkpoint/Restart	×	✓	×	✓
Process Migration	×	✓	✓	×
Dynamic Patching	×	✓	✓	×
本研究	✓	✓	✓	✓

できない．Shielded Execution は OS から App への攻撃を防ぐために App の保護を行う．しかしながら OS が停止すると App は実行を継続することができなくなる．Checkpoint/Restart は App の実行状態を保存し再開させることができるが，OS の Failure によって Checkpoint が破壊される可能性もあり Checkpoint 自体の信頼性が損なわれる可能性がある．また Checkpoint の頻度によって Checkpoint によるランタイムオーバヘッドと復元後の再実行時間はトレードオフの関係がある．Process Migration は OS アップデート前に予め App の実行状態を他のマシン上に移送することで，その下で実行されている OS をアップデートさせたり，アップデート済みの OS 上に移送することで，App の実行を継続しつつ OS アップデートを可能とする．しかしいつ起こるか予測不可能である Failure に対しては利用できない．Dynamic Patching は OS の再起動なく OS カーネルのアップデートを適用するものであるが，すべてのアップデートで利用できるわけではない．大規模なアップデートでは利用できないことが多く，特にデータ構造の変更を伴う場合には利用が困難である．

Verification を用いることで対象とするソフトウェア内の特定のコーディング違反を検出することができ，本研究と補完的である．OS Architecture の構成方法によってこれまでに述べた手法が利用しやすくなることもある．しかし特定の OS 構成方法に依存することは利用範囲を狭めてしまうこととなるため，本研究では特定の OS 構成方法になるべく依存しない手法を提案する．

第3章　提案手法

OS 内の障害や OS アップデートが引き起こす OS の再起動によって，App の実行状態が失われることや稼働時間が減少することによって，コンピュータシステム全体の信頼性が低下することを抑制する．Checkpoint/Restart が再起動後の OS でも App の実行を再開させられることに着目し，OS の障害やアップデートによって App のが稼働できない時間を最小限にする．本章では OS 再起動後の App 復元に必要な Essential Context を取得する手法を述べる．前章で述べた Checkpoint/Restart 手法と同様，App の挙動を記録して Checkpoint を作成し，復元時には OS 内のデータオブジェクトを再構築しユーザ空間のメモリ内容を再展開する．まず OS 内で障害が発生した場合でも App の実行状態を保存し再起動後の OS で復元する手法について述べ，次に OS アップデート時に App の実行が停止するサービスダウンタイムを最小化する手法について述べる．

3.1　Essential Context

Essential Context は従来のプロセスコンテキストとは異なり，App の復元に最小限必要な App の実行状態である．従来のプロセスコンテキストは，OS カーネル内の大量のメモリオブジェクトから構成されており，App の復元には必ずしも必要のないものも多く含まれる．たとえば図 3.1 に示すとおり，Linux ではプロセスのスレッド情報を管理する task_struct という構造体には PID，TID，CPU のレジスタ値，プロセスの実行可能・実行不可能・終了のステータス，スケジューラの優先度，などといった情報が含まれている．App の復元に必要となるのはこれらのうち PID，TID，レジスタ値である．復元の前後で PID が変わってしまうとシグナル等を利用する App では誤作動の原因となってしまうため，またレジスタ値を失ってしまうとどの処理から App を再開すれば良いのかがわからなくなるためである．たとえスケジューラの優先度が失われたとしても，App の実行が再開されればその値は再度スケジューラによって更新されていき，適切な優先度が設定されるため復元する必要はない．

Essential Context は OS 内のメモリオブジェクトを直接参照することなく取得することができる．たとえば task_struct のメモリオブジェクトを読めば PID やレジスタ値は判明するが，App が getpid システムコールを実行すれば，App が OS 内のメモリオブジェクトを直接読み込むことなく PID を知ることができる．レジスタ

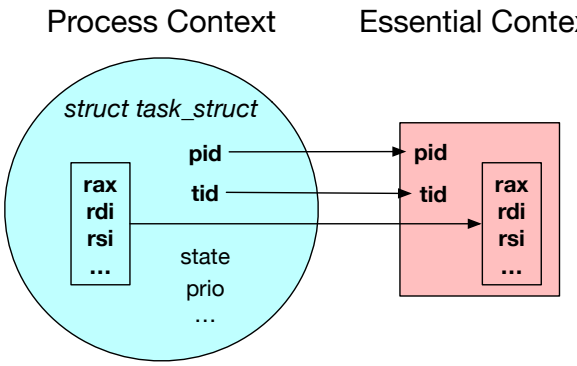


図 3.1: Process Context と Essential Context の違い

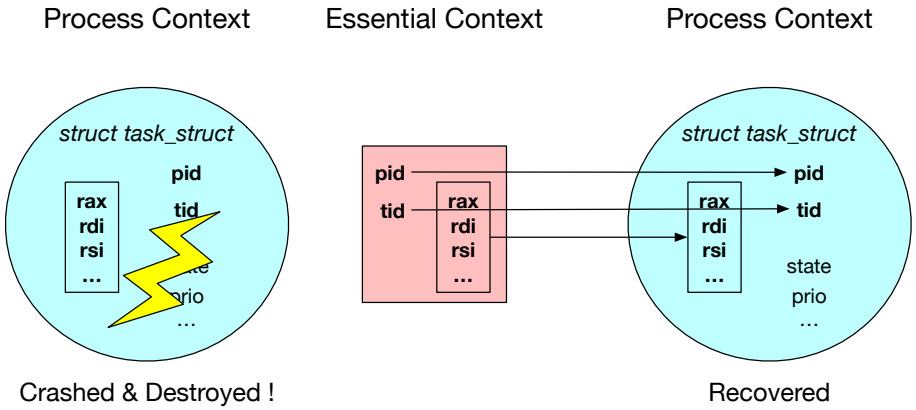


図 3.2: Essential Context を用いた OS クラッシュからの App 復元

値は，OS より権限の強い VMM を導入すれば，App が直接アクセスすることができないものも含めて参照することができる．

OS カーネルのクラッシュが発生して OS 内のメモリオブジェクトが破壊されたとしても，Essential Context を用いれば App の実行を復元することができる．OS カーネル内でクラッシュが発生した場合，OS 内のエラープロパゲーションによってメモリオブジェクトが破壊されてしまっている可能性がある．そのため，クラッシュが発生した時点でのメモリオブジェクトを使用して App の復元を行うと，破壊されたメモリオブジェクトを参照してさらなるエラーが発生する可能性もある．OS 内のどのメモリオブジェクトが破壊されていてどのメモリオブジェクトは破壊されていないのかといった判断を行うことは不可能で，信頼できるリカバリ手法ではない．そこで図 3.2 に示すとおり，App が実行されてから OS カーネルのクラッシュが発生するまでの間に Essential Context を取得しておき，クラッシュが発生した場合には Essential Context から App の実行を復元することで，信頼できるリカバリが達成できる．

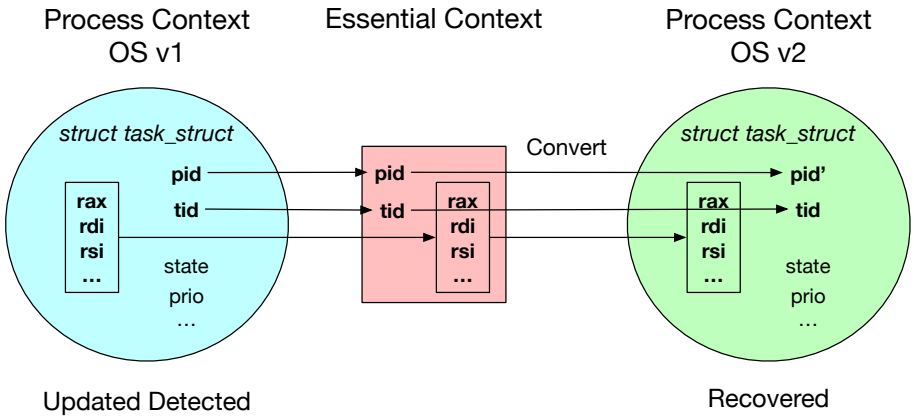


図 3.3: Essential Context を用いた OS カーネル後の App 復元

OS のアップデートが起こったとしてもプロセスという抽象化は変わっていないため，Essential Context も変わりにくい．Essential Context が変わらない限り，再起動後のアップデート後の OS で App を復元させることが可能である．たとえばファイルパスを構築する OS 内のデータ構造が変化した場合，古いバージョンの OS が持つメモリオブジェクトを新しいバージョンの OS が持つメモリオブジェクトに直接代入することはできない．データ構造内のエントリが増減して配置が変わったりすると，古いメモリオブジェクトの持つ値を新しいメモリオブジェクトのどの位置に挿入するのか，そのメモリオブジェクトの値が持つ意味が変わった場合にはどのような値に変更しなければならないのか，といったバージョン間の違いを埋めるリカバリ処理を記述しなければならなくなってしまい，バージョンの数が増えれば現実的ではなくなってしまう．そこで図 3.3 に示すとおり，Essential Context を経由して古いデータ構造から新しいデータ構造へと変換すれば，異なるバージョン間の OS が持つメモリオブジェクトも正しく復元することができる．ただし後方互換性が失われるような OS のアップデートにおいては，App と OS のインタフェースが変更されてしまい復元できない場合もある．このような OS アップデートでは一般的に App のアップデートも必要となり，既存の Checkpoint と同様そもそも App の実行を継続させることはできない．

Essential Context に含めるべき実行状態は，App が使用するハードウェアと使用するシステムコールのインタフェースから決定することができる．App が使用するデバイスには，CPU，MMU が挙げられる．CPU についてはレジスタの値を保存する．MMU はセグメントとページテーブルによって仮想アドレスと物理アドレスを変換するため，仮想アドレスと物理アドレスのマッピングを保存する．またユーザ空間内のメモリオブジェクトも保存する．たとえば Linux のシステムコールでは，kill や wait といった pid を引数とするシステムコールが存在するため，pid は Essential Context に含まれる．同様にして，open システムコールが存在するため

ファイルのパス・展開フラグ・ファイルディスクリプタ番号，write・read・lseek システムコールが存在するためファイルの読み書きオフセットと読み書きした内容，mmap システムコールが存在するため仮想アドレス領域の開始位置・長さ・保護フラグ，socket・bind・listen・access・connect システムコールが存在するためソケットで使用するプロトコル識別情報，そのプロトコルで使用する接続管理情報（たとえば UDP であれば 16bit 長のポート番号，TCP であればさらに 32bit 長のシーケンス番号など），clone システムコールが存在するため子プロセスや子スレッドの PID，sigaction システムコールが存在するためシグナルハンドラの関数ポインタ・ブロックマスク，set_robust_list・get_robust_list システムコールが存在するため robust_list へのポインタ，が Essential Context には必要である．

3.2 ShadowBuddy

OS カーネル内でバグが発症して障害が起こった際に再起動した OS が App の実行状態を引き継ぎ，App の実行を継続する手法を提案する．OS カーネル内で障害が発生することを想定しているため，カーネルレベルの Checkpoint を使うことはできず，OS よりハードウェア権限の低いユーザレベルの Checkpoint も利用することができない．そこで OS カーネルより権限の強い VMM を導入し，VMM から App の Checkpoint を作成する．OS 内で障害が発生したとしても VMM はハードウェアによって保護されており，エラープロパゲーションの影響を受けること無く App の再開を行うことができる．

OS を信頼すること無く App の Checkpoint を作成するため，Checkpoint の作成を行うための改変を OS に加えることはしない．また対応する App を増やすために App やライブラリのソースコードへの改変は行わない．また Checkpoint 作成のためにかかるランタイムオーバヘッド・メモリオーバヘッドを削減するために，App が持つユーザ空間のメモリ内容を複製しない Checkpoint 手法を提案する．一般的な Checkpoint/Restart では App 内で発生する障害が App のユーザ空間のメモリ内容を破壊してしまうことを想定しているため，一定間隔でユーザ空間のメモリ内容すべてを Checkpoint に保存したり，App 実行中に発生するメモリ書き込みのたびに Checkpoint を更新する必要があった．本研究では App のユーザ空間のメモリを OS カーネルから保護する手法を導入することで，OS の障害が App のユーザ空間のメモリ内容を破壊してしまうことを防ぐ．OS より強いハードウェア権限で動作する VMM による保護であるため，OS はこの保護手法を迂回して App のユーザ空間のメモリを破壊することはできない．また OS のメモリ内容を直接再利用する VM Replication とは異なり，App の実行状態から必要な OS のメモリオブジェクトを再構築する手法を採用し，障害によって OS 内のメモリオブジェクトが破壊された場合でも正しく App の実行を再開することを保証する．

3.3 Dwarf

バグを修正するために行われる OS のアップデートは，障害の発生を未然に防ぎ OS の信頼性を向上させることができる．しかしながら OS のアップデートを適用するためには長い時間を要する OS の再起動が必要であり，App の稼働時間が減ることから可用性は低下してしまう．OS の再起動を行うためには，稼働している App を停止させ，稼働している OS を停止させ，マシンを再起動し，アップデート後の OS を起動し，稼働していた App を起動するという手順を踏む．App の再開を行う機構を持っている場合や Checkpoint を行う場合は実行状態を低速な不揮発性のディスク等に書き込むため，稼働している App を停止させることは時間を要する．また OS の起動終了後は App の実行を再開させるために，メモリオブジェクトをディスクからメモリに読み込んだりすることで App の再開にも時間がかかる．これら時間を要する処理が必要となる理由は，マシンの再起動によって不揮発性メモリの内容が失われるためである．

App が実行できない間に生じるサービスダウンタイムを短縮することでコンピュータシステムの可用性を向上させるために，VMM を導入する．新しい VM 上でアップデート後の OS を起動することでマシン再起動を省略しつつ，OS の起動と App の停止を並列実行することで App のサービスダウンタイムを短縮することができる．同一マシン上で起動した VM であるため，起動した OS は稼働していた App のユーザ空間のメモリを直接参照する事が可能であり，App の停止をする際には低速なディスク等に実行状態を書き込む必要はない．またユーザ空間のメモリ内容を復元する処理も，ユーザ空間のメモリサイズが大きいほど時間がかかってしまう．そこで App がアクセスした時にユーザ空間のメモリを復元することで，大量のメモリを使用する App であっても高速に復元することを可能とする．

3.4 まとめ

本章では，本研究で用いる手法がコンピュータシステム全体の信頼性を向上させることができることについて述べた．OS の障害によって App の実行状態を失うことを防ぐことで，可用性と保守性を向上させる．また OS の障害発生時に発生するエラープロパゲーションが App の実行状態を破壊することを防ぐことで，保全性を向上させる．アップデートを適用するための OS 再起動の時間を短縮し高速に App を再開させることで，可用性を向上させる．また OS のアップデートがバグを修正するため，狭義の信頼性を向上させることができる．本研究では VMM レベル，カーネルレベルの両方で Essential Context を取得する手法を提案した．これまでは App の実行状態を OS から保護しつつ，OS の障害発生後に App を再開させることはできていなかったが，ShadowBuddy では App のユーザ空間のメモリ保護を行いつつ，信頼できないカーネル内のメモリオブジェクトを再利用すること無く App を再開することができる．また Dwarf では OS の再起動によって発生する App

のダウンタイムを既存研究以上に短縮することができた．次章以降，これらの手法の具体的な実現方法について述べる．

第4章　ShadowBuddy

OS カーネルのクラッシュは，OS 自身だけでなくその上で動作する全てのアプリケーションを停止させ，アプリケーションが提供するサービスの可用性を低下させる．全てのアプリケーションが OS の機能に依存しており，OS のクラッシュによって停止した機能をアプリケーションが利用した時点で動作が停止する．本章では，OS の障害時に発生するカーネルエラープロパゲーションから App の実行状態を保護し，再起動後の OS での App 再開を保証する手法 *ShadowBuddy* について述べる．OS のコードにもバグが存在することは事実であり，このバグを実行してしまうと OS が異常な挙動をする場合があり，OS の実行を継続できなくなることもある．

4.1　提案

OS 内で障害が発生した場合でも，App の実行を再起動後の OS で継続する．ShadowBuddy は以下の特徴を持つ．

- **OS で障害が発生しても App の実行状態を失わない:** ユーザレベル・カーネルレベルの Checkpoint/Restart はカーネル以下の権限で行われるため，カーネルの障害からは守られていない．また障害によって破壊されてしまったかも知れないメモリオブジェクトを再利用する手法とは異なり，カーネル空間内のメモリオブジェクトを直接参照して再利用することはしない．
- **障害発生後の再起動でも I/O View を一貫させる:** 再起動を行う前と後とで，App から認識される I/O の状態は一貫していなければならない．例えばネットワークコネクションも維持する必要がある．TCP/IP の通信ではお互いにシーケンス番号と ACK 番号を持っており，復元後の OS でもコネクションが途切れることなく App を再開することができる．
- **ランタイムオーバヘッド・メモリオーバヘッドを小さく保つ:** 既存手法のように保護によって大幅にスループットを低下させるのではなく，ハードウェアの機能を活用することでランタイムオーバヘッドを抑制する．

4.2 課題

OS クラッシュ発生後，再起動した OS で App の実行状態を復元し，クラッシュ直前の実行状態を再現する必要がある．正しく App の復元を行うためには，OS クラッシュが App のメモリを破壊しないようにする必要がある，OS の障害に対してカーネルレベルの App 保護手法 [24] を導入したとしても，カーネルは App の全メモリに対するアクセス権を持っているため，App のメモリが破壊されないという保証にはならない．また Intel SGX のような App がハードウェアの保護機構を利用する場合，App のメモリが破壊されることを防ぐことはできるが，App の実行状態を再開することはできなくなる．

クラッシュ発生時の実行状態を使うことなく OS の実行状態を再現する必要がある，たとえば App がディスク上のファイルに書き込んだ場合，OS 内のバッファキャッシュには書き込まれても，その内容が実際にディスクに反映されるまでには時間がかかる．この間にクラッシュが起こってしまうと，App はディスクに書き込んだと認識しているものが，実際にはディスクに書き込まれていないという状態，すなわち I/O View が一貫しない状態となる．I/O View を一貫させるためには，再発行するべき I/O があるのかを判定し，実行しなければならない．DBMS などでは fdatasync システムコールを用いることで，OS に対してディスクへの書き込みを強制しているが，すべての I/O を同期的に実行すると膨大な待ち時間が発生し，スループットが大幅に低下してしまう．しかしながら OS 内のバッファキャッシュがどこに存在し，いつ書き込まれたのかを OS 外から特定することは困難である．

4.3 設計

OS で起こった障害から App の実行状態を保護するために VMM を導入する．VMM は OS よりも高いハードウェアの権限で動作するため，OS はこの保護を迂回して App のメモリを破壊することはできない．また Checkpoint の作成も VMM から行い，OS が Checkpoint のオブジェクトを破壊することができないようにする．I/O View の保証をするために VMM から I/O の監視を行い，再発行しなければならない I/O を特定する．本章ではこれらの方針の実現方法を述べる．実証のために OS は Linux，VMM は Xen を用いて設計・実装を行ったが，別の OS や VMM であってもそれぞれに合わせた設計や実装の変更は可能である．

4.3.1 EPT-level Protection

SMEP や既存研究 [24] など，ページテーブルを利用した保護では OS の障害から App の持つメモリを保護できるとは言えない．これらの保護機構は OS によって実現されているため，OS は容易に保護を迂回して App のメモリ空間にアクセスすることができるためである．そこで OS よりハードウェア権限の強いモードで実

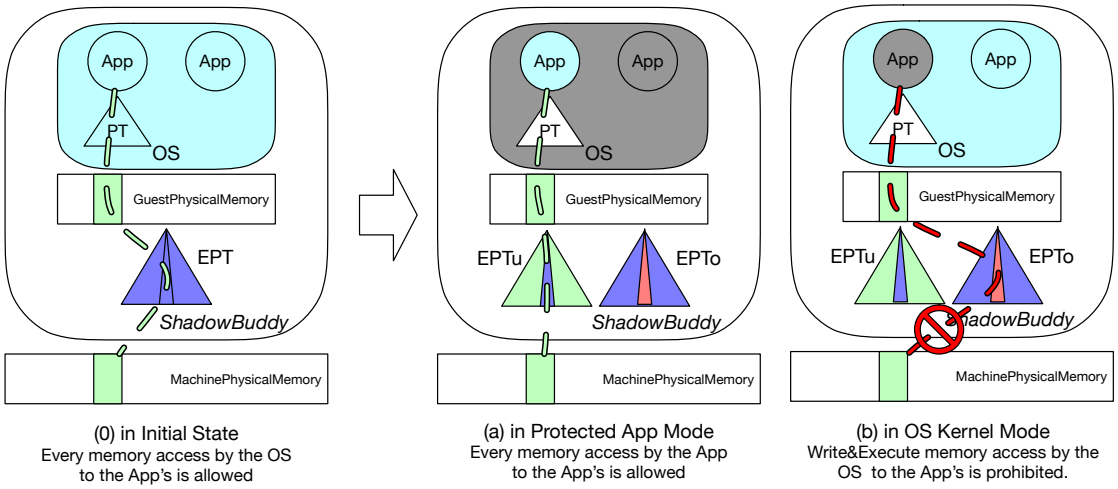


図 4.1: EPTo と EPTu の複数 EPT 構成

行される VMM を導入し，OS による App のユーザ空間のメモリへの不正な書き込みを防ぐ．

この保護を行うために，CPU によるメモリの仮想化支援技術 EPT を用いる．EPT はゲスト OS が認識するゲスト物理ページと，実際にマシンが認識するマシン物理ページを変換する．VMM は予め EPT を設定しておくことで，ゲスト OS の実行中メモリアドレスの変換は CPU アーキテクチャによって行われる．OS がページテーブルを導入することで App 間の Isolation を行うように，EPT は一般的に VM 間のメモリの Isolation のために用いられ，複数の VM 上で別の OS が動作している場合にある OS が別の O S のメモリ内容を読み書きすることができないようにすることができる．初期の VMM はこれをソフトウェア上で実装していたが，ゲスト OS 上での TLB ミス時には必ず VMM に制御が移り，オーバヘッドの原因となっていた．EPT を導入することでこの処理を CPU 側で対処することができ，VM Exit を発生させずゲスト OS と VMM の性能上のメリットが大きい．

本研究ではこの EPT を，OS と App 間の Isolation のために活用する．図 4.1(0) のように，通常は OS と App は同じ EPT を通してメモリアドレスの変換が行われる．ShadowBuddy による保護を開始する時，OS と通常の App が動作する EPTo と，保護対象の App が動作する EPTu に分ける．保護対象の App がユーザ空間に持つメモリは，(a)App からは読み書き実行を行うことができるものの，(b)OS や保護対象外の App からは書き込みと実行ができなくなる．

EPT の設定は表 4.1 にまとめた．OS からの不正な書き込みを防ぐために，保護対象の App が使用するメモリは EPTo 上では全て Write を禁止する．次項で述べる Syscall Logging のために，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，EPTo 上では全て Execute を禁止する．保護対象の App 実行中は EPTu で動作するため，OS が持つ

表 4.1: EPT の設定とアクセスの禁止理由

ページ所有者		OS	Protected App	理由
EPTo	R	✓	✓	
	W	✓	×	App のメモリ保護
	X	✓	×	OS から App への遷移検知
EPTu	R	✓	✓	
	W	✓	✓	
	X	×	✓	App から OS への遷移検知

ページテーブルを構成するマシン物理ページは EPTu にも割り当てる．

App は新しく導入したハイパーコールによって，ShadowBuddy の保護の開始を要求する．ShadowBuddy は EPTu として空の EPT を作成し，App の CR3 レジスタと対応付ける．ShadowBuddy は App が実行のために必要としたマシン物理ページのみを EPTu に割り当てる．具体的には，CR3 レジスタに格納されているゲスト物理ページ番号を用いて App が動作しているページテーブルを参照し，App のユーザ空間のメモリ（Linux においては 0x000000000000-0x7fffffffffff）に割り当てられている各マシン物理ページに対し，EPTo は実行権限を消し，EPTu は全アクセス権限を設定する．またページテーブルを構成するゲスト物理ページに対応するマシン物理ページは EPTu にも割り当て，App の動作中も CPU がページテーブルを参照して仮想アドレスのアドレス変換を行うことができるようにする．また，OS によってページテーブルが破壊されることを想定し，EPTu にユーザ空間のマシン物理ページを割り当てる時に，仮想アドレスとマシン物理ページの対応をログに保存する．

OS の動作中に App のメモリに書き込むと，EPTo には書き込み禁止の設定がなされているため EPT Violation が発生し VMM 内のハンドラに制御が移る．App が意図しないメモリ書き込みは OS の障害によって発生したエラープロパゲーションの可能性があるため，VM を新しく起動して OS を再起動し App の実行を再開させることができる．App の動作中に OS のメモリを実行すると，EPTu には実行禁止の設定がなされているため EPT Violation が発生する．例えば App がシステムコールを発行した場合や，ハードウェアの割り込みによって OS 内の割り込みハンドラに制御が移った場合などが考えられる．この場合には 4.3.3 項の手法で Checkpoint の更新を行う．

4.3.2 CPU Cache Synchronization

TLB キャッシュと EPT のキャッシュは CPU が参照する EPT を変更した後も残り，このキャッシュが残っていた場合は OS も App のユーザ空間に書き込みを行うことができてしまう．ShadowBuddy では，(1) このキャッシュを消さない場合，(2)App と OS の遷移時に毎回キャッシュを消す命令を実行した場合，(3)VPID をスイッチの度にインクリメントすることでキャッシュを無効にする場合，(4)OS 用と App 用の VPID を別々に確保してスイッチの度に代入した場合，の 4 つで実装を行った．(1) の方法は保護が不完全である．(2) の方法は保護ができる反面，ランタイムオーバヘッドが高い．(3) と (4) の方法は (2) のランタイムオーバヘッドを抑えるための最適化である．

Xen ではゲスト OS の切り替えや，ゲスト OS がプロセスのコンテキストスイッチを行うために CR3 を切り替えた場合に，VPID の値をインクリメントしている．TLB や EPT Cache は VPID によってタグ付けされており，VPID を変更することで古いキャッシュを無効にしつつ，実際にフラッシュするためにかかる時間を短縮している．(3) の手法では，ShadowBuddy 利用時に OS と App を切り替える場合にも VPID をインクリメントし，OS 実行中に App 用のキャッシュが利用されないようにしている．したがって，VPID はシステムコールなどが発生する場合，以下のように推移する．

OS(VPID=101) → App(VPID=102) → OS(VPID=103) → App(VPID=104)...

しかし，OS と App 間の遷移の度にキャッシュを失ってしまうとスイッチ後に TLB ミスが頻発し，App や OS の実行速度が低下する．そのため，OS と App それぞれの VPID を記録しておき，スイッチを行う場合には記録しておいた VPID を代入することでこの速度低下を抑制する．したがって，VPID は以下のように推移する．

OS(VPID=101) → App(VPID=102) → OS(VPID=101) → App(VPID=102)...

Xen が App ごとに VPID を記録せずに VPID をインクリメントする理由は，一度 OS の切り替えやコンテキストスイッチが発生した後，もう一度その App を実行しキャッシュを利用するまでの間に TLB や EPT Cache が上書きされている場合が多いためである．

4.3.3 Syscall Logging

App が発行するシステムコールは OS 内のメモリオブジェクトを変更するため，ShadowBuddy は Checkpoint を更新しなければならない．しかし Intel VT-x では通常，システムコールを直接捕捉する仕組みは搭載されていない．そこで表 4.1 のように EPT を設定したことで，ユーザ空間とカーネル空間で実行が推移する瞬間

を捕捉することができる．推移のタイミングで，原因がシステムコールであるか，割り込みであるかの判定を行う．実行しているOSに合わせてシステムコールのインタフェースと割り込み内容の理解が必要となり，それに応じて適切なログを残す必要がある．EPToでは保護対象Appのメモリが，pEPTではOSのメモリが実行禁止となっている．そのため，Appがp EPTで動作中にシステムコールを実行すると，カーネル空間のメモリが実行禁止に設定されているためにEPT Violationが発生する．

ユーザ空間からカーネル空間に実行が移動した時，そのIP(Instruction Pointer)やVMCS内の割り込みベクタを見ることで発生原因を特定することができる．高速システムコールの呼び出しSYSCALL/SYSENTER命令は，それぞれそのジャンプ先がレジスタのLS_LSTAR，GUEST_SYSENTER_EIPに保存されている．IPがLS_LSTAR，GUEST_SYSENTER_EIPのいずれかの値と一致している場合，あるいは割り込み番号が0x80である場合はシステムコールである．それ以外の場合で割り込みベクタに値が設定されている場合は，割り込みである．

ShadowBuddyを導入したことによって発生するEPT Violationの原因は，システムコールや割り込み以外にも多岐にわたる．これらの発生原因の判別方法と対象方法について，以下にその代表的なものを示す．

- **システムコールが発行された場合:** Executeの EPT Violationが発生し，かつIPの値が先述したシステムコール用レジスタの値と一致している場合．レジスタ値を保存し，システムコールの内容に合わせて適切なログを残す．システムコールに突入する瞬間，一部のシステムコールではその引数に合わせてCheckpointを更新する．参照渡しの引数がある場合，ShadowBufferの設定を行う．

- **割り込みが発生した場合:** Executeの EPT Violationが発生し，システムコール以外である場合．スケジューリングのためにタイマ割り込みが発生したり，ユーザ空間中でページフォルトが発生した場合など．レジスタ値の保存は行うが，ログは残さない．

- **システムコールが完了しユーザ空間に返った場合:** Executeの EPT Violationが発生し，直前にシステムコールが発行されていてユーザ空間の仮想アドレスに戻った場合．システムコール発行時に保存されていたレジスタの値から，システムコール番号とその引数，またカーネルが設定した返り値からCheckpointを更新する．成功と失敗の両方があるほとんどのシステムコールで，Checkpointはこの時点で更新する．

- **割り込みハンドラが完了した場合:** レジスタの保存や対処は行わない．

- **App実行中にEPTuに存在しないユーザ空間のページにアクセスした場合:** Read，Write，Executeの EPT Violationが発生し，フォルトを起こした仮想ア

ドレスはユーザ空間内で，該当するゲスト物理ページに対応するマシン物理ページは EPTu には EPT エントリが無く，EPTo には EPT エントリが存在する場合．ページフォルトのハンドラが実行された後に頻繁に発生する．App がページフォルトを起こした後カーネルは該当する仮想アドレスにページテーブルを割り当てるが，この時点では EPTu には反映されないため再度この EPT Violation が発生する．EPTo から EPTu にエントリを複製し，EPTo での権限は読み込みのみ，EPTu での権限は読み書き実行のすべてを許可する．

- **App がカーネル空間のメモリにアクセスした場合:** Read，Write，Execute の EPT Violation が発生し，フォルトを起こした仮想アドレスはカーネル空間内で，該当するゲスト物理ページに対応するマシン物理ページは EPTu には EPT エントリが無く，EPTo には EPT エントリが存在する場合．EPTo から EPTu にエントリを複製し，両方共読み書き実行すべてを許可する．App で障害が発生することや，App が OS カーネルの脆弱性をついて攻撃するといった状況は想定してないため，App から行う OS メモリへの参照は許可する．カーネルが事前に指定した一部の仮想アドレス以外への読み書き実行は，一般的にページテーブルレベルで参照がなかったりアクセスが禁止されていたりするため，EPT-level Protection を行わなくても問題無い．

- **EPTu 内に存在しないページテーブルエントリにアクセスした場合:** App 内で Read の EPT Violation が発生し，その原因がページテーブルの変換中（npfec_kind_in_gpt）であった場合．ページテーブルを構成するゲスト物理ページを CR3 から探索し EPTo から EPTu にエントリを複製し，ページテーブルを構成するゲスト物理ページに対応するマシン物理ページは EPTo も EPTu での権限は読み書き実行のすべてを許可する．途中でページテーブルのエントリが存在しない場合はそこで探索を中断する．これは Demand Paging によって OS が実際にゲスト物理ページを割り当てていない場合に起こる．ShadowBuddy 側はエントリの複製を中断することで OS にページフォルトが通知され，OS が適切にページフォルトの処理を行うことができる．OS 側のページフォルトへの処理が終わって App に実行が戻ると，EPTu には更新されたページテーブル分のエントリが存在しないため，再びこの EPT Violation が発生する．

ここでは Checkpoint の更新が必要なシステムコールのログ作成機構のみ記述する．システムコールは発行時だけでなく，完了時の返り値も踏まえてログを作成しなければならない．システムコールが利用可能なものであるか，許可されたものであるか，といった判断は全て OS 内で行っており，ShadowBuddy がそれを発行時に判定してログを作成することは不適切であるためである．各システムコールに対して ShadowBuddy で行うべき Checkpoint の更新の一部を，表 4.2 に示す．CPU のレジスタにはシステムコール番号と引数がそれぞれ設定されており，RAX はシ

ステムコール番号，RDI・RSI・RDX・R10・R8・R9 はそれぞれ第一から第六引数を格納している．ユーザ空間のメモリに OS から書き込みを行う read などのシステムコールでは，OS がユーザ空間のメモリに書き込んだ瞬間に EPT Violation が発生してしまい，障害だと検知されてしまう．そこで 4.3.4 項の *ShadowBuffer*(以下 sbuf) を用いることでこれに対処する．App が OS から値を受け取る際に引数でポインタを用いている場合は，OS が一時的に書き込むための sbuf を設定する．また App が OS にポインタで値を渡す場合には，該当仮想アドレスに sbuf が設定されていないかを確認し，もし sbuf が存在すればその領域にコピーしておく．

4.3.4 ShadowBuffer

OS カーネルがユーザ空間のメモリに書き込むようなシステムコール全般に，適切に対処するために ShadowBuffer を用いる．EPT-level Protection が有効に働いている場合，OS から App のメモリに書き込みが発生すると EPT Violation が発生する．その時 Syscall Log を参照し，App が発行したシステムコールの中に，該当領域への書き込みを要求するものがあればこれは正しいものであると判断できる．しかしながら，サイズの大きいメモリ書き込みを行う場合，何度も EPT Violation が発生してしまいランタイムオーバヘッドが大きくなる．

そこで，App のユーザ空間のメモリに書き込みを発生させるシステムコールが発行された場合は，書き込み先の仮想アドレスに対応する EPTo の物理ページに書き込み用の新しいメモリページ (ShadowBuffer) を挿入する．OS がこの領域にメモリコピーを行っても EPT Violation が発生することはなく，かつ App に割り当てられているメモリページとは別のメモリページであるため OS のメモリ書き込みが App のメモリを破壊することはできない．システムコールが終わり App に遷移する時には，App が指定した書き込み先の仮想アドレスに対し，ShadowBuddy が ShadowBuffer からメモリコピーを行う．書き込む長さはシステムコールの種類や引数と返り値に依存する．たとえば pipe システムコールであれば int 型で長さ 2 の配列分のメモリサイズをコピーする．また read システムコールであれば，App が指定したバッファの最大長と OS が指定した返り値のうち小さい方のサイズ分をコピーする．ShadowBuffer から App のユーザ空間へのコピーが完了した後，他にその ShadowBuffer を利用するシステムコールが存在しなければ ShadowBuffer のメモリページは解放する．

ShadowBuffer はページサイズの 4KB 単位で設定されるため，OS は同じページに含まれる App のメモリ内容にアクセスすることができなくなる．例えば，あるスレッドが read(3, 0x80000, 16) というシステムコールを発行すると，仮想アドレス 0x80000 に対応する物理ページに ShadowBuffer が設定される．このシステムコールが終わる前に，別のスレッドが write(4, 0x80016, 16) というシステムコールを発行すると，仮想アドレス 0x80016 と 0x80000 は同じ物理ページが使用されるため，OS は App が実際に持っているメモリではなく ShadowBuffer から読み込みを行っ

表 4.2: システムコールへの対処例

システムコール名	発行時	完了・成功時
open	なし	ファイル名と展開フラグを記録.
read	読み込み先のアドレスに sbuf を設定.	sbuf から RDI が指すアドレスに書き込む. シーク位置を増加.
write	書き込み内容をログ. sbuf に内容をコピー.	シーク位置を増加
lseek	なし	先頭からのシーク位置が返り値で, Checkpoint のシーク位置を変更する.
close	なし	App はもうアクセスしないため 閉じられたことを示すフラグを立てる
mmap	なし	返り値は割り当てられた仮想アドレスで, 展開フラグ・サイズ等と合わせて記録する.
exit/exit_group	すべての保護を無効にする	なし
getpid	なし	なし
socket	なし	ファイルディスクリプタ番号と プロトコル番号を記録する.
bind	sbuf に受信情報をコピー	該当するソケットに割り当てられた アドレスと受信相手を記録する.
listen	なし	バックログの長さを記録する.
accept/ accept4	接続情報を受け取るポインタに sbuf の設定を行う.	I/O ログから接続相手の 識別情報を取得する.
connect	sbuf に接続情報をコピー. ソケットに送信先を記録する.	I/O ログから接続相手の 識別情報を取得する.
pipe	ファイルディスクリプタの 格納先に sbuf を設定する.	接続された両端の ファイルディスクリプタを記録する.
fsync/ fdatasync	なし	ファイルディスクリプタに対応する I/O ログを削除する.

てしまう．したがって，OS が App のユーザ空間のメモリを読み込むシステムコールでは，その仮想アドレス領域がすでに存在する ShadowBuffer 内に含まれているかということを検査しなければならない．もし該当領域に ShadowBuffer が存在した場合は，App が実際に持っているメモリから ShadowBuffer に必要な分のメモリコピーを行っておき，OS が正しくバッファを読み込めるようにする．

ShadowBuffer は EPT Violation が頻発することを防ぐが，ランタイムオーバヘッドが増加する原因となる．新しく EPTo に挿入した ShadowBuffer を OS が確実に使用するためには，TLB と EPT のキャッシュをフラッシュしなければならない．また ShadowBuffer を解放した後は，再度 OS が直接 App のユーザ空間のメモリを読み込み専用で参照できるよう EPTo のエントリを変更する．この際にも TLB と EPT のキャッシュをフラッシュしなければならず，スループット低下の原因となってしまう．そこで，あえて ShadowBuffer の解放を行うことはせず，EPTo に ShadowBuffer を残すことで TLB と EPT のキャッシュをフラッシュする回数を減らす．ShadowBuffer がすでに設定されている領域にメモリコピーを行うシステムコールが発行された場合は，EPTo のエントリを更新する必要がなく 2 回のフラッシュを削減することができる．この最適化オプションは，TLB や EPT のキャッシュをフラッシュする回数を減らす代わりに，ShadowBuffer が増え続けれメモリオーバヘッドが増加することに加え，メモリのコピー量が増加するという欠点も持つ．

4.3.5 I/O Logging

OS が I/O デバイスをどのように操作したかということも，App の Restart を行うために必要となる場合がある．例えば App が TCP/IP の socket を通じてパケットを送信した後に OS 内で障害が発生した場合，(1) パケットは通信相手に到達していた場合，(2) パケットは通信相手に到達していなかった場合，とで Restart 時に行う処理が変わる．前者の場合にはそのパケットを再度送る必要はないが，後者の場合には送らなければならない．

OS がネットワークパケットをどう処理するか，シーケンス番号の初期値をどの値にするか，といったことは ShadowBuddy が OS が持つメモリオブジェクトの内容を把握しなければならず特定は現実的ではない．そこで OS 内の実行を監視することなく，必要な情報を残すために I/O Logging を行う．OS が発行するデバイス I/O の内容からこれらの情報を特定し，クラッシュ時に活用できる情報を残す．OS は仮想 NIC に対して I/O を行うが，そのバックエンドは VMM 内に存在する．そのため，ここで I/O の内容を監視することで，OS 内のメモリ全てを把握することなく情報を収集することが可能である．例えば TCP/IP パケットの送受信時には表 4.3 のようにして対処する．

Syscall Logging で得た bind，listen，connect システムコール履歴を元に，送受信されたパケットはどのソケットに関するものなのかということを特定することが可能である．特に SYN&ACK パケットを監視することで，TCP の 3-way Handshake

表 4.3: 各パケットヘッダに合わせた I/O ログの作成

監視対象		対処方法
送信	SYN	なし
	SYN&ACK	bind 情報を元にパケットをデータ通信用ソケットに紐付ける TCP の初期シーケンス番号・ACK 番号をソケット情報に記録する
	DATA	なし
	DATA&ACK	App が受取済みのシーケンス番号でパケットの ACK を上書きする
受信	SYN	なし
	SYN&ACK	connect 情報を元にパケットをソケットに紐付ける TCP の初期シーケンス番号・ACK 番号をソケット情報に記録する
	DATA	なし
	DATA&ACK	send システムコール時に作成した送信バッファを解放する

が完了したことを特定することができる．TCP プロトコルでは送受信されるパケットの順序を保証するためにシーケンス番号が用いられている．セキュリティ上の観点からこの初期値はランダムな値となっており，ShadowBuddy では SYN&ACK パケット中のシーケンス番号と ACK 番号を，ソケット Checkpoint のシーケンス番号・ACK 番号の初期値とする．

ShadowBuddy では VM Replication [66, 67, 68] でネットワーク I/O のレイテンシ増加の原因となっていた I/O のバッファリングは行わず，OS が発行した I/O は随時実行されることを可能とする．データの送信を行うために App が send システムコールなどを発行した場合は，送信バッファの内容をソケット Checkpoint に追加する．ACK を含むパケットを受信した場合は，ACK が返された範囲の送信バッファをソケット Checkpoint から解放する．

送信する ACK 番号は，App が recv システムコールなどで受け取った範囲までの番号で上書きする．これは，OS がパケットを受信して ACK を返したものの，App が受け取る前に障害が発生してしまった場合に受信パケットが失われるという問題に対処するためである．ACK 番号を上書きすることで通信を行っているリモートのマシンは送信バッファを解放することがなく，再起動を行った後に TCP の再送メカニズムによって該当パケットを受け取ることができる．受信したパケットの複製を保存しておくことでもこの問題に対処することができるが，障害が発生しないほとんどの場合においてメモリコピーのオーバヘッドが増加する，ソケットの復元を行う時に受信データを挿入しなければならず復元処理が複雑化する，という欠点があり ACK 番号を上書きする手法を採用した．

4.3.6 Recovery

ShadowBuddy が作成した Checkpoint を元に，再起動後の OS で App の実行状態を復元する．CPU，Memory，I/O の各資源の方法は以下の通りである．

CPU の復元

レジスタの値は通常上書きするだけで良い．ただし，動作中の AP のレジスタの値を直接書き換えるコードを実行すると誤作動の原因となるため，OS から App に実行が戻る時に利用される，struct pt_regs の値を更新する．ただし，呼び出し規約で Callee Saved Register に設定されている一部のレジスタは，更新した pt_regs の値を利用せずカーネルスタック上に push/pop されてしまう．カーネルスタック上に積まれたレジスタの値を書き換えることは現実的ではないため，OS 内のリカバリ処理が終わりユーザ空間に戻った瞬間に ShadowBuddy がトラップし，VMM からレジスタの値を復元する．

Memory の復元

仮想アドレス領域を復元することで，プロセスのメモリ空間を復元する．Checkpoint に入っている仮想アドレス領域それぞれには，開始アドレス，領域のサイズ，アドレス領域の使用フラグ，プロテクションのフラグ，マップされたファイルの情報が格納されている．これらを引数として mmap システムコール，mprotect システムコールを実行することで，仮想アドレス領域の復元を行う．

また仮想アドレスに対応するメモリ内容の復元も行う．障害を起こしたが使っていたページテーブルは，App の CR3 レジスタの値を用いることですべて参照することができる．

I/O 資源

各プロセスのファイルディスクリプタ番号に対応するファイル資源を割り当てる．一般ファイルであれば Checkpoint 内にファイルパスと展開フラグが保存されているため，これらを引数として open システムコールを実行する．seek 可能なファイルに対しては，open の後 lseek システムコールを実行することで操作位置を変更する．パイプのように seek 不可能な I/O 資源においてはその初期化を行わず，障害発生時点で Checkpoint に残っていたパイプの書き込みバッファ分を，障害発生後に書き込みをしておく．TCP/IP を利用するソケットのようにマシンの内外で通信が行われる場合には，接続相手との送受信パケット・発行されたシステムコールの内容を元に再発行するべき I/O を特定する．詳細は 4.3.5 項で述べたとおりである．

4.4 実験

以上の提案手法のうち，EPT-level Protection，CPU Cache Synchronization，Syscall Logging，I/O Logging は Xen 4.11.2 に実装し，ソースコードの変更量は 6820 行で

表 4.4: ShadowBuddy の実験環境

	Host	Guest
CPU	Xeon 6cores	6vCPU
Memory	16GiB	4GiB
Disk	500GB HDD	120GB vDisk
NIC	1Gbps	vNIC
OS	Linux 3.18.75/Xen 4.11.2	Linux 4.9.68

あった．I/O Logging はホスト OS にも実装し，168 行の変更を行った．Restart はゲスト OS に実装を行い，1,872 行を変更した．

作成した ShadowBuddy のプロトタイプの有用性を確認するため，本節ではランタイムオーバヘッドの計測と App の復元についての実験を行う．実験環境は表 4.4 に示す．

4.4.1 マイクロベンチマークでのランタイムオーバヘッド

ShadowBuddy のオーバヘッドがどこに存在するかを評価するために，まず 10,000,000 回のループの中でレジスタを用いて加算を行うプログラム，getpid システムコールを繰り返し呼び出すプログラムを実行し，その所要時間を計測した．この実験では EPT-level Protection のみを有効にし，Syscall Logging，I/O Logging は有効にしていない．図 4.2 にその結果を示す．レジスタでの演算所要時間は 1%未満のオーバヘッドであった．CPU のレジスタ操作では ShadowBuddy のオーバヘッドが発生することは無く，タイマ割り込み等によって発生するオーバヘッドが影響したものと考えられる．getpid は最も単純なシステムコールで，App と OS 間のメモリ書き込みもないが，それでも約 120%程度のオーバヘッドが発生した．syscall/sysret 命令が発行されるたびに VM Exit/VM Resume が発生しているためで，これ以上システムコールのオーバヘッドを削減することは困難である．

次にシステムコールごとのオーバヘッドの違いを評価するため，open，read，write，getpid の各システムコールを繰り返し 20 回実行するベンチマークプログラムを作成した．VM Exit が発生してから Syscall Logging がシステムコールの引数から Checkpoint の更新を行う時間 (HandSyscall)，Syscall Logging の後 VM Enter をするまでの時間 (VMMSyscall)，カーネル空間に推移する前に CPU Cache Synchronization を行う時間 (App-¿OS)，VM Resume の直前から VM Exit で VMM に制御が移るまでのカーネル空間中にいる時間 (OS)，VM Exit が発生してから Syscall Logging がシステムコールの返り値から Checkpoint の更新を行う時間 (HandSysret)，Syscall Logging の後 VM Enter をするまでの時間 (VMMSysret)，ユーザ空間に推

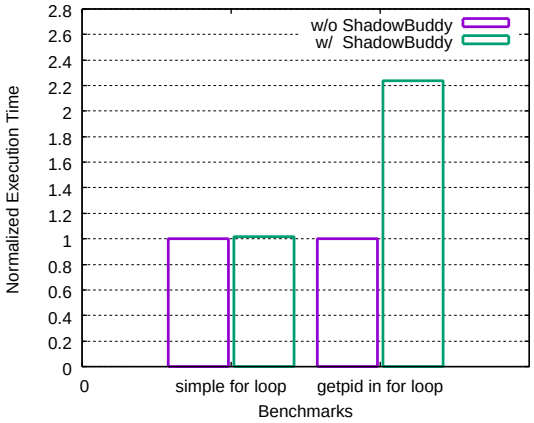


図 4.2: レジスタでの演算，システムコールの発行を行う際の所要時間

移する前に CPU Cache Synchronization を行う時間 (OS-¿App)，VM Resume の直前から VM Exit で VMM に制御が移るまでのユーザ空間中にいる時間 (App)，のそれぞれを計測した積み上げグラフを図 4.3 から図 4.6 に示す．CPU Cache Synchronization の実装は 5 種類行い，それぞれ比較を行った．w/o Protection は保護を有効にせずユーザ空間の中で時間を計測した．CPU Cache Synchronization 実行時の理想的な実行時間を示す参考として計測した．Flush は App・OS 間の遷移が起こるたびに TLB と EPT をフラッシュするもの，Increment は Tagged TLB と Tagged EPT を活用し VPID を加算することで両キャッシュを無効化するもの，Toggle は OS 用のと App 用にタグを割り当て VPID に代入するもの，Full は Toggle に加え ShadowBuffer の解放を行わない最適化を施したものである．

この実験では，VMM の VM Exit ハンドラ内に多くの時刻取得コードを埋め込んだため，また Syscall Logging と I/O Logging を有効にしたため前の実験に比べオーバヘッドが大きくなっている．またベンチマークの実行中にスケジューリングが発生したり，タイマ割り込みなどによって極端に長い時間がかかる場合もあり，ここでは区間ごとの所要時間の中央値を使用した．w/o Protection は 20 回のループでの平均値を算出した．ShadowBuddy の保護なしと保護の最適化を行った場合で比較すると，全体の所要時間は 3 倍から 36 倍となっている．CPU Cache Synchronization の最適化によって高速化しているものの，VM Exit のコストの高さ，ShadowBuffer を解放するために行う TLB や EPT フラッシュによる速度低下，ShadowBuddy の処理によって CPU のキャッシュが汚れてしまうことによるユーザ・カーネル空間中の速度低下，が顕著である．read システムコールでは同じメモリ領域に対して read システムコールを実行したため ShadowBuffer を解放しなかったことの恩恵が得られている．ShadowBuffer の割り当て・解放の処理が省略されるだけでなく，OS の EPT を変更したことで OS 用の TLB と EPT を無効にする必要がなかったことで CPU キャッシュが性能改善に貢献している．

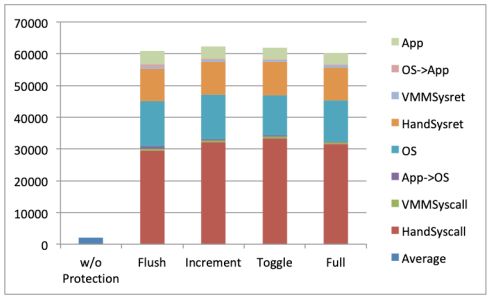


図 4.3: Syscall Logging によって発生するオーバヘッド (open)

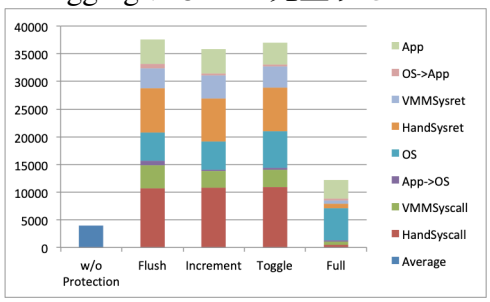


図 4.4: Syscall Logging によって発生するオーバヘッド (read)

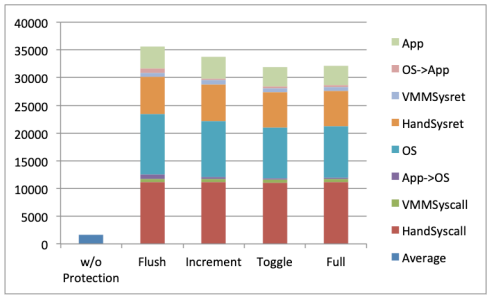


図 4.5: Syscall Logging によって発生するオーバヘッド (write)

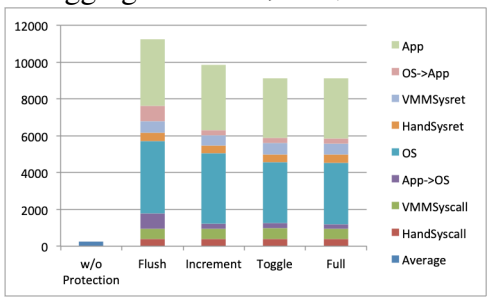


図 4.6: Syscall Logging によって発生するオーバヘッド (getpid)

表 4.5: 各ベンチマーク内容

App	Intensive	Workload
matmul	CPU, Memory	500x500 の行列計算の所要時間
tar	Memory, Disk I/O	Linux のソースコードのアーカイブ化の所要時間
gzip	Memory, Disk I/O	上記 tar ファイルの圧縮所要時間
memcached	Memory, Network I/O	memtier ベンチマーク実行時のスループット
memcachedb	Network I/O, Disk I/O	memtier ベンチマーク実行時のスループット
caffe	CPU, Memory	機械学習の所要時間

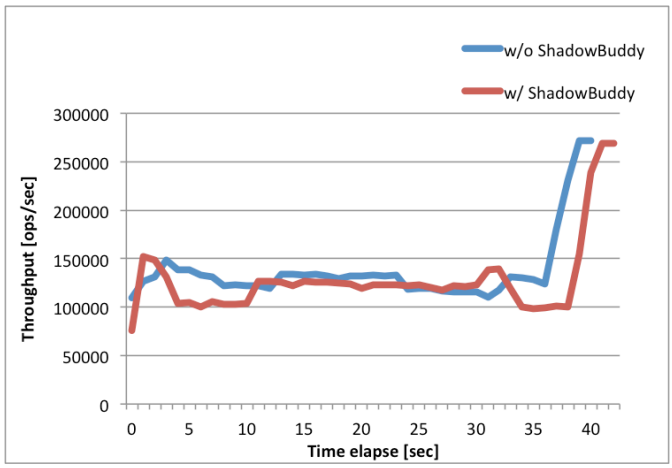


図 4.7: memcached に対する memtier ベンチマークのスループット

4.4.2 実アプリケーションでのランタイムオーバヘッド

ShadowBuddy のランタイムオーバヘッドが，実アプリケーションではどれだけの大きさになるのかを評価する．マイクロベンチマークではシステムコールが多発するような場合のワーストケースを示しており，実際のアプリケーションではそれほど多くのシステムコールを発行することは少ないためオーバヘッドは小さくなる．matmul，tar，gunzip，grep，memcached，memcachedb，caffe を ShadowBuddy の保護あり，保護なしのそれぞれで実行し，実行時間やスループットを計測した．表 4.5 に各アプリケーションの特性とベンチマーク内容を示す．

実験を行った結果，matmul，tar，gzip，caffe の所要時間増加はいずれも 5%未満であった．memcached のスループットは図 4.7 に示す．スループットのオーバヘッドは 6.4%程度であった．ShadowBuddy はシステムコール発行時にオーバヘッドが増加するが，実際のアプリケーションが発行するシステムコールの量はそれほど多くないためオーバヘッドは限定的である．

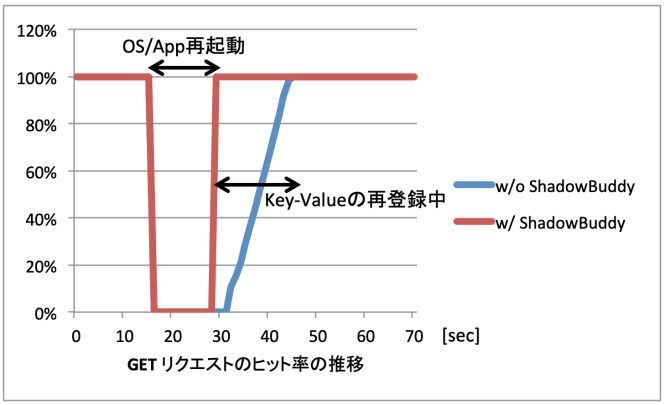


図 4.8: memcached に対する memtier ベンチマークのヒット率

4.4.3 Key-Value Store でのヒット率

メモリ上にデータを保存する Key-Value Store である memcached は，再起動を行うとメモリ上のデータを失うため，GET リクエストに対して結果を返すことができなくなってしまう．この実験では，ShadowBuddy による保護ありと保護なしとで，GET リクエストに対するヒット率を計測した．memcached を立ち上げた後，1024 バイトの Value を 100,000 個登録しておき，ヒット率の推移を計測した．ShadowBuddy による保護が無い場合はヒット率が 0%となってしまうため，memcached を通常の方法で再起動した後 Key-Value の登録を行った．

結果を図 4.8 に示す．OS と App の再起動中は GET リクエストに対する返答が無いためヒット率が 0%になるものの，ShadowBuddy によって App の復元を行うとヒット率はすぐに 100%に改善された．一方 ShadowBuddy による保護が無い場合は memcached への Key-Value の登録を行いながら GET リクエストが発生したため，ヒット率が改善されるまでに時間がかかっている．また TCP コネクションが切断されたために，memtier ベンチマークも再度コネクションを作成し直す必要があった．

4.4.4 フォルトインジェクション

OS 内でカーネルクラッシュが発生した場合であっても，ShadowBuddy が取得していた Essential Context から App の実行状態を再開させることができることを示すために，フォルトインジェクションを行った．フォルトインジェクションには Swifi を用いた．Swifi は Linux カーネル内に，カーネルパニックを引き起こすコードや，ロックの解放を行わないなどと行ったフォルトを挿入する．ベンチマーク対象の memcached の実行を開始した後 5 秒後からフォルトを発症させ，カーネルクラッシュを起こさせた．挿入するフォルトの種類に合わせて，3 種類のベンチマー

表 4.6: memcached に対するフォルトインジェクションの結果

Set	NUM	No Crash	Crash Not Detected	Recovered	Failed
A	200	15	2	178	5
B	100	0	0	100	0
C	100	0	0	100	0

クを用意した．

- Set-A: テキスト領域をビットフリップ，スタック上の値をビットフリップ，ランダムに命令を nop に変更する，alloc 系の関数が NULL を返すようにする，条件分岐 jmp 系の命令のいくつかを nop にする，use-after=free か double-free を引き起こす，unlock 系の命令を飛ばす，という 7 種類を 30 箇所ずつランダムに発生させる

- Set-B: 即座にカーネルパニックを引き起こす

- Set-C: 即座に無限ループを起こしてウォッチドッグタイマが異常を検知させる

結果を表 4.6 にまとめた．フォルトが発症せずベンチマークが稼働し続けた場合は No Crash，フォルトが発症したものの ShadowBuddy からは検知できなかった場合は Crash Not Detected，フォルトが発症して ShadowBuddy によって復元ができた場合は Recovered，復元に失敗した場合は Failed として計測した．

OS カーネルのクラッシュが起こった場合のリカバリ成功率は，378/383 で約 98.7% であった．Set-B と Set-C ではフォルトは確実にクラッシュとして検出され，またフォルトの種類はメモリ破壊を引き起こすものではなかったため，リカバリに失敗することはなかった．フォルトを挿入したものの，App は稼働を続けた場合 (No Crash) が Set-A で 15 回あった．また一部の OS のコンポーネントが停止したためネットワークが不通となり，memtier ベンチマークでのスループットは 0 になったものの，OS カーネル自体はクラッシュしておらず ShadowBuddy から検知できなかった場合 (Crash Not Detected) が Set-A で 2 回あった．No Crash と Crash Not Detected が起こる原因は，Set-A で挿入したフォルトが必ずしも毎回カーネルクラッシュを伴うものではないためである．App の復元に失敗したのは Set-A で 5 回あった．

App の復元に失敗した原因はユーザ空間内で起こっており，どのフォルトによってエラーが発生したのか完全な特定はできないが，原因は何点か考えられる．たとえば App がシステムコールを発行した間や，App 実行中に割り込みハンドラが実行されている最中にフォルトが発症して CPU のレジスタが破壊されたが，クラッシュとしては検出されず破壊されたままユーザ空間に戻ってしまった場合には ShadowBuddy では保護することができない．呼び出し規約で Callee-saved Registers に設定されているレジスタは，カーネルがそのレジスタを使う時にはカーネルスタックに push し，その関数から戻る時に pop される．このとき，フォルトによって

カーネルスタック上にある push した値が書き換えられていると，不正な値が pop されてしまいレジスタが破壊される．また一部の命令が nop になったり条件分岐 jmp が nop になるフォルトでも，レジスタが破壊されることもありうる．特に rbp, rbx，rcx はアドレスアクセスに使われる事が多い Callee-Saved Register であり，これらのレジスタが不正な値になった状態でユーザ空間に戻ると，エラープロパゲーションがユーザ空間内で起こる可能性が高い．フォルト発症直後にエラープロパゲーションがユーザ空間内で起こらなかったとしても，再び OS に実行が移った時点で Essential Context 内の CPU レジスタ情報が破壊された値で更新されることもある．ユーザ空間のメモリ内容の保護は達成できているが，CPU レジスタの破壊については課題として残っている．App から OS に遷移した時点の CPU レジスタと，OS から App に遷移した時点の CPU レジスタを比較して破壊されたかどうかを検出することも可能ではあるが，例えばシグナルハンドラの処理を行う場合などには OS が意図的に CPU レジスタの値を変更する場合もあり，レジスタ破壊が起こったとして誤検知されてしまう．

4.5 考察

4.5.1 障害からのページテーブルの保護

ページテーブルはカーネル空間から直接読み書きが可能であり，ShadowBuddy のプロトタイプでは守られていない App の実行状態であった．ShadowBuddy では，EPTo から EPTu にエントリを移動する際に，仮想アドレスと物理アドレスの対応付けを Checkpoint に保存していた．4KB のページごとにこの対応付けを Checkpoint に保存することはメモリオーバヘッドとなり，事実上ゲスト OS が持つページテーブルと同じものを持つということと同じである．そこで Checkpoint にこのマッピングを保存する代わりに，OS が持っていたページテーブルを OS の不正な書き込みから保護し，再起動後に利用する．App が持つユーザ空間のページと同様，ページテーブルを構成する物理ページについても，OS からの書き込みを禁止するよう EPTo を設定することで，OS の障害がページテーブルを破壊することを防ぐことができる．しかし OS は正常な動作であってもページテーブルの書き込みを行うことはある．ページテーブルへの書き込みを禁じて OS が操作できなくなれば，App は正常に動作しなくなってしまう．

OS がページテーブル操作を行った際に VMM がトラップし，以下の方針で変更を許可・禁止する．変更を加えたページテーブルにはエントリが存在しない場合は，変更を許可する．変更を加えたページテーブルにはエントリが存在した場合，すでにその仮想アドレスが munmap システムコールによって解放されていれば変更を許可する．それ以外の場合はページテーブルを破壊しようとしたとみなし，OS を再起動する．

しかし Kernel Same-page Merging(KSM) や，Transparent Huge Page(THP) など，App が意図しないタイミングでページテーブルが変更されることがあり，これらの機能がページテーブルを操作した時点で障害と誤検知してしまう可能性がある．ShadowBuddy を利用しつつこれらの機能の動作を許可することは，今後の課題である．

4.5.2 Checkpoint 頻度とランタイムオーバヘッドのトレードオフ

既存研究では，Checkpoint の頻度が高ければ高いほど，Restart 後に障害発生前の実行状態を再現するための時間が短いという利点がある反面，App の実行時間に占める Checkpoint の時間が長くなることで高いランタイムオーバヘッドが発生するというトレードオフが存在した．ShadowBuddy はシステムコール・割り込みが発生するたびに Checkpoint を更新するため頻度が高い．しかし実験で示したとおり，ランタイムオーバヘッドはほとんどの App で数%程度と小さい．ShadowBuddy は App の Restart が高速であるにもかかわらず平常時のランタイムオーバヘッドが低い，という既存研究では達成できたいなかった特徴を持っている．

4.5.3 ランタイムオーバヘッドの削減

ShadowBuddy では，App のメモリ保護と Checkpoint の更新のため 0-30%程度のランタイムオーバヘッドが発生した．これは EPT Violation 発生後に実際にハンドラが起動されるまでの所要サイクル数が長いことが一つの原因である．EPT Violation ではない別の手法で保護と Checkpoint の更新を行うことで，オーバヘッドの削減を行うことが見込める．本研究ではより多くの App で利用可能とするために，App の改変をしないという方針で EPT Violation を利用したが，App から OS へのシステムコール時に EPT Violation によって VMExit が発生する syscall ではなく，vmcall や vmfunc を実行して直接 VMM のハンドラを呼び出せばオーバヘッドは減る．原則システムコールを発行するのは libc などのライブラリからであるため，これらを変更しておくことで App の改変を行うこと無く高速化をすることが見込める．コンパイル時に LLVM などを用いて命令を置き換えるほか，NaCl [102] のようなバイナリの書き換えを行う手法を利用することも可能である．

また ShadowBuffer の導入とその最適化オプションによって，TLB と EPT のキャッシュをフラッシュする回数を減らす代わりにメモリコピー量が増えるという問題があった．そこで OS が書き込みに使う仮想アドレスと，読み込みに使う仮想アドレスが同一メモリページに配置されないよう App の改変を行うこともできる．あるいは Shielded Execution のように，SIM と呼ばれる App と OS 間のメモリの読み書きを仲介するメモリ領域を作成することもできる．ShadowBuddy の仕様に合わせて App や OS を改変すれば，App のスループット低下を抑制することは可能である．

表 4.7: CPU cycles on each Hypercall

CPU Model(Release Year)	Cycles	Reference
Intel Xeon X5550 @2.66GHz (2009)	961	Hyperkernel [80]
Intel Xeon E5-1620 @3.6GHz (2011)	765	
Intel Core i7-3770 @3.40GHz (2012)	760	
Intel Xeon E5-1650 v3 @3.5GHz (2013)	540	
Intel Core i5-6600K @3.50GHz (2015)	568	
Intel Core i7-7700K @4.20GHz (2016)	497	
AMD Ryzen 7 1700 @3.0GHz (2017)	697	
Intel Xeon E5-2630 v3 @2.4GHz (2014)	1,188	NEVE [94]
AMD ARMv8-A @2.4GHz (2016)	2,729	
Intel i7-6700k @4.00GHz (2015)	631	MemSentry [56]

4.5.4 TCB の増加

VMM の導入によって TCB が増えることは信頼性を低下させるという懸念がある．しかし OS に比べて VMM はコード量が小さく，その増加は比較的小さい．今回は Xen を実装対象としたが Xen 特有の機能に依存しているわけではなく，NOVA [43] のようなコード量の小さい VMM であっても実装は可能である．

4.5.5 VMM のバグへの対応

VMM は OS に比べコード量が小さいので，そもそもアップデート自体が少ない．さらに RootHammer [109] を用いれば稼働中の VM の実行状態を維持したまま VMM の再起動が可能であり，Swap and Play [110] を使えば VMM のライブアップデートが可能である．また VMM 内で障害が発生した時に，VM を稼働させたまま VMM コンポーネント単位で再起動する ReHype [111]，VMM コンポーネントの Rollback を行う NiLiHype [112] を利用することで数十 ms のダウンタイムで 96%の復元が可能である．

4.5.6 VMM の導入によるオーバヘッド

昨今の VMM は最適化されており，ベンチマークにもよるがランタイムオーバヘッドは数%程度と小さい．また CPU アーキテクチャの進歩により VMExit/VMEnter などの必要サイクル数は減っている．表 4.7 には様々なアーキテクチャでの Hypercall にかかる CPU サイクル数の推移を示す，傾向として，新しいアーキテクチャであればより高速に割り込み処理が可能である．

本研究では VMM の導入を行うため，仮想化によるオーバヘッドが増加することは避けられない．しかし Isolation を VMM に任せ OS は App の実行に必要な機能を担う Layered Kernel（特に VMM を利用する Library OS）とは相性が良い．

ShadowBuddy は特定の OS 実装方法に依存するものではないため，LightVM [95] のように Container よりも高速な VMM に採用することも可能である，

第5章　Dwarf

本章では，OS のアップデート時に発生する App の稼働時間低下を抑制する機構 *Dwarf* について述べる．OS カーネルのアップデートを行うことは日常的な出来事である．OS のアップデート時には App・OS・マシンをすべて再起動することが原則であるが，これらは長い時間を要する処理であるため App の実行ができないサービスダウンタイムが長期化してしまう．実行されている App によってサービスダウンタイムが引き起こす損失の度合いは異なるが，短いほど良いという点はすべての App で共通している．

5.1　提案

OS のアップデートを確実に適用しつつ，App のサービスダウンタイムを短縮する手法を提案する．Dwarf は以下の長所を持つ．

- **利用可能な OS アップデートが多い:** OS カーネルへの Dynamic Patching は数 ms のダウンタイムで OS のアップデートを適用できる反面，利用可能なアップデートには制限があり，利用できない場合にはアップデートの適用を諦めるか通常の再起動を行わなければならない．Dynamic Patching が利用できない場合に Dwarf を利用することで，可用性の低下を最小限にする．Dwarf はアップデート後の OS を通常通りの方法で起動するため，利用できない OS アップデートは存在しない．ただ OS に後方互換性が無く，稼働していた App の実行状態が利用できなくなる場合には利用不可能である．このようなアップデートにおいては既存研究においても対応することができていないため，対象外とする．また OS アップデート以外にも，OS のコンフィグ変更やソフトウェア若化のためにも利用することができる．

- **App の実行状態は維持される**アップデート前の OS 上で稼働していた App は一時停止され，数 s のサービスダウンタイムの後に一時停止直前の実行状態から再開することが可能である．App が持つメモリ上のキャッシュや CPU のコンテキストが失われることはなく，実験では App 再開後に有意なスループット低下が発生することは無かった．これは不揮発性のディスク装置などに実行状態を書き出す手法とは異なり，ユーザ空間のメモリを直接再利用しているためである．

- **既存の OS で利用可能である:** Dwarf は OS アーキテクチャの種類に依存すること無く利用可能である．実装は Monolithic Kernel の Linux に行ったが，Linux 特有の機能や Monolithic Kernel 特有の特徴を利用したわけではない．特定の OS 構成方法に限定した手法とは異なり，本研究のアイディアはすべての OS 構成方法で利用可能である．

- **冗長なハードウェアを必要としない:** Dwarf は追加のディスク装置やネットワーク機器などを必要とすることは無く，広く普及しているハードウェアを用いる．Process Migration や VM Migration は App の移送先となる別マシンを必要としたが，追加のハードウェアを用意することは追加の費用となってしまう．また特殊なハードウェアに依存してしまうと，Dwarf を採用可能な環境に制限を加えることになるため利用範囲を狭めてしまう．広く普及したハードウェアを利用することで Dwarf はあらゆる環境で利用可能になる．

これらの要件を満たすよう，Dwarf では以下の手法を提案する．

- **One-time Checkpoint/Restart:** アップデート前の OS を停止させる直前に一度 Checkpoint を取得し，アップデート後の OS で Restart する．OS のバージョンに依存しない Checkpoint/Restart を行うために，App の実行状態は独自に定義した Essential Context に保存する．

- **Background Boot:**

- **Attachment-pipelining:**

Dwarf を構成する機能のうち，Essential Context の作成・App の復元を行う機構は OS に，OS の高速再起動と Essential Context の受け渡しを行う機構は VMM に導入する．Dwarf のプロトタイプは，前者は Linux 2.6.39.4 と Linux 4.1.6，後者は Xen 4.5.0 に実装を行った．

5.2 課題

OS のあるバージョンのデータ構造をそのまま Essential Context として保存することは適切ではない．理由は 2 つあり，1 つ目は OS アップデートが OS 内のデータ構造を変更する可能性があるため，2 つ目は App の再開に不要なメモリオブジェクトも含まれているためである．Essential Context は特定のバージョンに依存することなく定義しなければならない．

アップデート前の OS とアップデート後の OS では，多重化できない I/O デバイスを同時に利用することはできない．一般的な VMM は VM を立ち上げる時に，起動するべき OS のイメージが入ったディスク装置を必要とする．OS に割り当てられているディスク装置やネットワーク装置はそれぞれ 1 つしかなく，ブートローダー

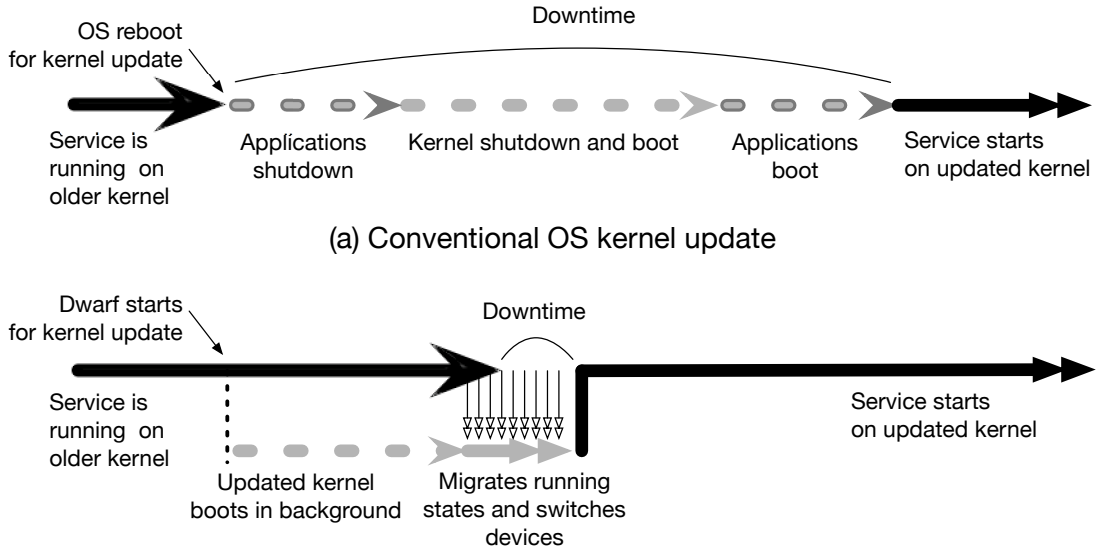


図 5.1: Dwarf による高速な App 再開

やアップデート後の OS イメージが格納されたルートデバイスが無ければ，アップデート後の OS を起動することができない．ブートローダーの実行前の段階で VM の起動を止めてしまった場合，vCPU コアと割り当てるメモリの確保を行う VMM の処理までは実行することができるが，OS カーネルのイメージをメモリに展開し，OS 内の初期化を行う処理は実行することができない．これでは App のサービスダウンタイムの短縮効果が限定的になってしまう．

5.3 設計

5.3.1 OS Update Flow

OS カーネルのアップデートを行う場合には，App の停止，OS の停止，マシンの再起動，OS の起動，App の起動，という手順を踏む必要があった．Dwarf では図 5.1 のように，App を稼働させた状態でアップデート後の OS を並列に起動し，App の実行状態を受取可能になった時点で App を一時停止，実行状態をアップデート後の OS に移して App の実行を再開させる．

本節では，App の実行状態を取得する One-time Checkpoint，アップデート後の OS を並列に起動する Background Boot，OS が利用していたデバイスを高速に切り替える Attachment-pipelining について述べる．

5.3.2 One-time Checkpoint

アップデート後のOSでAppの実行を再開するために，DwarfではCheckpoint/Restartと同様，アップデート前のOSカーネル内のメモリオブジェクトからAppの実行状態を抽出し，アップデート後のOSで再構築する．OS内の障害と異なりOSのアップデートは予測可能なイベントであるため，アップデートを適用する直前に一度だけCheckpointを取得すれば良い．本節ではCheckpoint，Recoveryのそれぞれについて述べ，既存のCheckpoint/Restartツールと異なる点についても述べる．

Checkpointは Appの実行状態を取得し，Appの再実行が可能な状態で保存することである．Dwarfでは OS内のメモリオブジェクトを参照して，Appの CPU，Memory，I/Oに関する実行状態からEssential Contextを抽出する．具体的には以下の通りである．CPUに対しては，レジスタの値，プロセスID，プロセスの親子・兄弟・グループ関係，登録されたシグナルハンドラ．Memoryに対しては，利用している仮想アドレス領域の範囲・読み書き実行などの属性，またそのメモリ内容である．I/Oに対しては展開している各ファイルディスクリプタに対応する情報である．ファイルであればファイル名，読み書きの展開フラグ，シーク位置，などである．パイプであれば，接続され対になっているファイルディスクリプタ番号，書き込まれたが読み込まれていないバッファ，などである．ソケットであれば，通信プロトコル，接続相手の識別情報，TCPなど順序情報を持つプロトコルであればシーケンス番号やACK番号，などである．これらの情報を格納するメモリオブジェクトはOSカーネル内に散在しており，Dwarfのカーネルモジュールが各メモリオブジェクトを収集する．

OSのバージョンに依存しない形式でEssential Contextを作成するにあたり，Dwarfでは以下の基準でCheckpointのデータ構造を定義した．

- **メモリオブジェクトの値を直接保存可能なもの:** ABIやプロトコルによって定義されている値はOSのバージョンによって変更されることはほとんどないため，値を直接利用することができる．たとえばファイルの展開フラグ（O_RDONLY，O_WRONLY，O_RDWR）などは，Appから参照可能なヘッダファイルなどに記述されている．一般的なOSでは互換性のため，一度定義した値を変更することはなく，該当する機能が廃止されない限り残り続ける．本研究では後方互換性を保証したOSのアップデートに限定しているため，このようなフラグの値は変わることがない．他にも，例えばTCPのシーケンス番号やACK番号などの扱いはすべてプロトコルによって定められており，そのビット長，フラグに使われる値の意味はOSのバージョンによらず一定である．

- **メモリオブジェクトの値を独自形式に変換して保存するもの:** 上記以外の場合は，アップデート前のOSのデータ構造からEssential Contextで独自に定義したデータ構造に，Essential Context独自のデータ構造からアップデート

後の OS のデータ構造に変換する処理を，それぞれアップデート前の OS とアップデート後の OS に組み込む必要がある．これに該当する OS 内のデータ構造であっても，API やハードウェアインタフェースに定義された形式に変換を行うことができれば OS のバージョンに依存しない Essential Context の宣言が可能である．たとえばあるファイルのパスを表すデータ構造が OS 内でどのように宣言されているかは，OS のバージョンによって異なる可能性がある．実際に Linux 2.6.39.4 と 4.1.6 とでは struct dentry や struct inode などのデータ構造の宣言が異なっているが，パスを char の配列として保存することで OS のバージョン間で差異無く保存することができる．

5.3.3 Restart

アップデート後の OS は Essential Context を VMM から受け取り，OS 内のメモリオブジェクトを再構築する．Restart の方法は ShadowBuddy と同様であるため，詳細は 4.3.6 を参照．本項では App のサービスダウンタイム削減することを目的とした最適化手法について述べる．

On-demand Page Table Recovery

App が該当するメモリにアクセスした時に，該当する仮想アドレスと物理ページのマッピングを作成することで，App が持つメモリサイズに比例しない復元処理ダウンタイムを達成することができる．特に匿名のヒープの領域を大量に使う App が多い場合に有効である．仮想アドレス領域の復元を行う mmap を行う際に，Dwarf によって後でメモリページが復元されることを示すフラグを立てておく．App が実際にその仮想アドレスにアクセスするとページフォルトが発生する．ページフォルトハンドラ中で，このフラグが立っていた場合は新しいメモリページを割り当てるのではなく，保存しておいた App のメモリページからメモリ内容の復元を行う．

この復元はページ単位で行うことが基本であるが，ページフォルトが大量に発生することから復元後の App のスループットが低下するという懸念もある．VM Migration の Post Copy [78] では bubbling という手法が採用されている．これは空間局所性に着目し，ページフォルトを起こした仮想アドレス付近は再びアクセスされやすいため優先的にメモリページを復元していくという手法である．Dwarf のメモリページ復元においても利用可能で，ページフォルトを起こしたメモリページの付近を予め復元しておくことで，ページフォルトが頻発して App のスループット低下を抑制することが可能である．

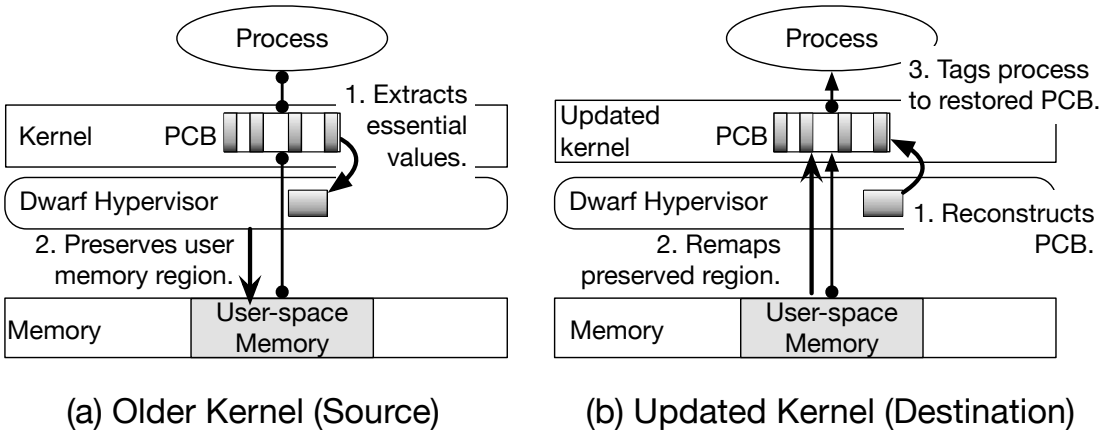


図 5.2: Zero-copy Memory Recovery によるメモリ復元

Zero-copy Memory Recovery

障害を起こした OS と新しく起動した OS は同一マシン上で実行されているため，App が使用していた物理ページは直接再利用することができる．図 5.2 に，コピーを行わないメモリ復元の手法を示す．新しい OS はゲスト物理ページフレームを確保した後，そのページに古い OS が持っていたメモリページを取得するよう Dwarf に依頼する．Dwarf は Checkpoint を参照して該当する物理ページを取得し，そのアドレスを通知する．OS はその物理ページをページテーブルに設定することで，メモリコピーを行うこと無くメモリページの復元を行うことができる．

この手法では新しい OS が持つと判断される物理ページが増え続けてしまうため，復元によって割り当てた分のメモリページを新しい OS から回収する必要がある．そのため現在の実装では，OS がゲスト物理ページフレームを確保する際に OS に割り当てられていた物理ページを回収し，復元先の物理ページと交換することで解決した．ある仮想アドレスに対応する物理ページを切り替えたため，復元処理の後は TLB フラッシュを行う．

One-copy Memory Recovery

前項の Zero-copy Memory Recovery では，再起動後の OS が仮想アドレスに対応する物理ページのマッピングを作成，ShadowBuddy に通知して対応する物理ページを割り当てるという処理が必要であった．しかし再起動後の OS に物理ページを引き継ぎ続ければ，VM に割り当てたメモリ量に齟齬が生じるため，再起動後の OS に割り当てていた物理ページを取り除いてからメモリページの復元を行う．この物理ページが CPU のキャッシュに存在する場合は，復元後も取り除かれた物理ページへのアクセスを続けてしまう可能性があるため，CPU キャッシュをフラッ

シュする必要がある．しかし 1 つのメモリページの復元を行うたびに TLB キャッシュをフラッシュすることはランタイムオーバヘッドとなってしまう．

CPU アーキテクチャによっては，CPU キャッシュをフラッシュするためにかかる時間とその後に発生する TLB ミスなどによるランタイムオーバヘッドが，単純に 4KB のメモリをコピーする時間より長い場合がある．このような場合にはあえて Zero-copy Memory Recovery を行うより，メモリのコピーによる復元を行うことで App のスループット低下を抑制することができる．

復元がされていないメモリに App がアクセスするとページフォルトが発生し，OS のページフォルトハンドラに飛ぶ．OS は通常通りメモリを割り当てた後，Dwarf にメモリ内容のコピーを依頼する Hypercall を発行し，ハンドラから App に実行を戻すことができる．4KB 分のメモリコピーが行われるため CPU のキャッシュが汚染されるが，TLB キャッシュを失うより高速に動作する場合がある．古い OS が持っていたページは不要となるため解放される．しかし古い OS はもう再稼働することがないため，TLB フラッシュを行う必要はない．新しい OS でもページマッピングの対応を無効にしなければならないエントリが存在しないため，TLB フラッシュを行う必要はない．

5.3.4 Shortening Downtime

本項では App のダウンタイムを削減するための 2 つの手法，Background Boot と Attachment-pipelining について述べる．Backgroud Boot は App の Checkpoint 取得を行う前に予めアップデート後の OS を起動しておく手法で，サービスダウンタイムの削減の大半を占める．Attachment-pipelining は時間のかかるデバイスの引き継ぎ処理をパイプライン化することで，アップデート後の OS の初期化処理をなるべく早く実行するために役に立つ．

Backgroud Boot

VMM の一般的な機能を用い，アップデート後の新しい OS カーネルをバックグラウンドで起動する．カーネルの起動はルートデバイスを利用する直前で一時停止させ，デバイスがアップデート前の OS から引き継がれるまで待つ．この処理は古いカーネルがプロセスを実行している最中から行うことができるため，カーネルの起動にかかる時間の大半をダウンタイムから除外することができる．図 5.3 に通常の OS 再起動と，Background Boot によるダウンタイムの少ない OS 起動方法の違いを示す．

新しい OS カーネルの準備には，利用する VMM やブート方法によって様々な手法を採用することができる．例えば Xen では，ホスト OS 内に存在するカーネルイメージを用いて VM を起動する事ができる．OS がアップデートしたカーネルのイメージを作成してホスト OS に転送しておくことで，アップデート後の OS の

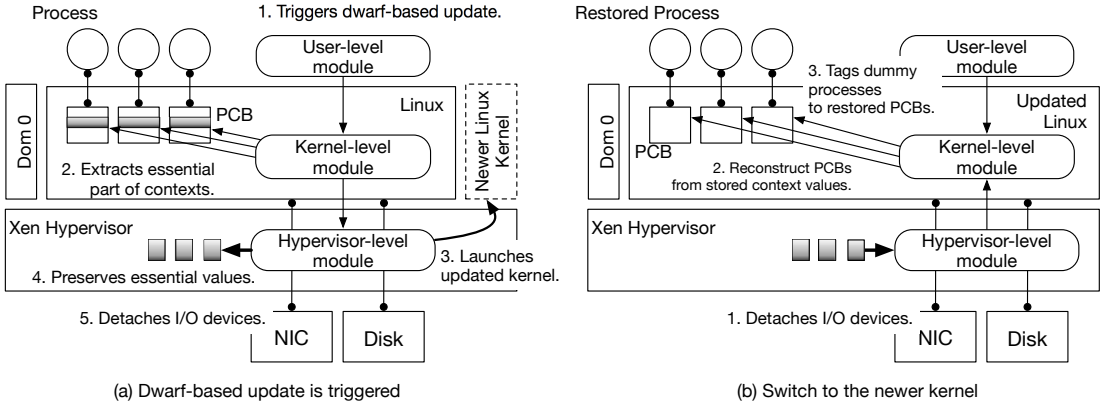


図 5.3: Background Boot した OS への App の移送

Background Boot が可能である．また NFS Boot を使用している場合には，NFS 側に新しい OS カーネルのイメージを用意すればそちらを用いて VM を起動することもできる．このように，新しい OS を起動するためには必ずしもディスク装置が必要なわけではなく，OS のブート処理のうち CPU の初期化やメモリの初期化といった処理はデバイス無しでも行うことができる．

Attachment-pipelining

デバイスの引き継ぎは，古い OS カーネルからのデバイスのデタッチ，新しい OS カーネルへのデバイスのアタッチによって行われる．デタッチ処理は古いカーネルがデバイスドライバの解放を行い，キャッシュのフラッシュなどが発生する場合がある．このためアタッチ処理に比べて長時間かかることがある．

図 5.4 に Attachment-pipelining で NIC とディスクの切り替え手順を示す．複数のデバイスを引き継ぐ場合，ディスク装置のうちルートデバイスのデタッチ＆アタッチが終了した時点でカーネルの初期化は再開させることができる．先にルートデバイスのデタッチ＆アタッチを終えてから新しい OS カーネルの初期化を行わせ，それと並行してネットワークなど他のデバイスをデタッチ＆アタッチすることで，ダウンタイムを削減することができる．引き継ぎを行うデバイスには，PCI などで接続された実デバイス（Disk，NIC など）の他，VMM によってソフトウェア的に作られた仮想デバイス（Virtual Disk，Virtual NIC），ネットワーク経由で Disk を利用することのできる NFS なども利用可能である．これらのデバイスを安全なタイミングで古い OS から切断し，新しい OS への再接続後に Essential Context を参照して状態を再現する．例えば TCP/IP の通信を行っている App があればソケットを作成し，シーケンス番号や ACK 番号を適切に設定した後ネットワーク通信を再開させる．

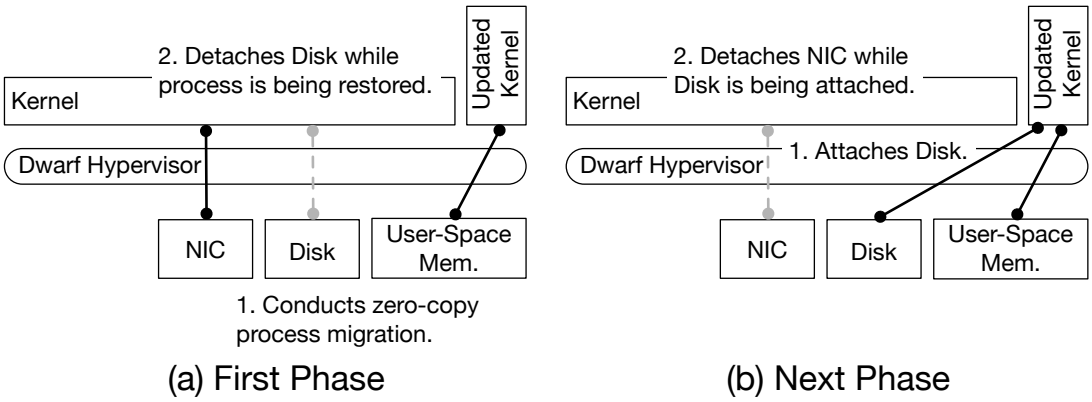


図 5.4: Process Migration と Attachment-pipelining の並列実行

表 5.1: Dwarf の実験環境

	Host	Guest
CPU	Xeon E3-1240 4cores 8threads	8vCPU
Memory	16GiB	12GiB
Disk	500GB HDD	240GB SSD(PCI Pass)/40GB vDisk
NIC	1Gbps	1Gbps(PCI Pass)/vNIC
OS	Linux 3.10.7 on Xen 4.5.0	Linux 2.6.39.4/3.18.35/4.1.6

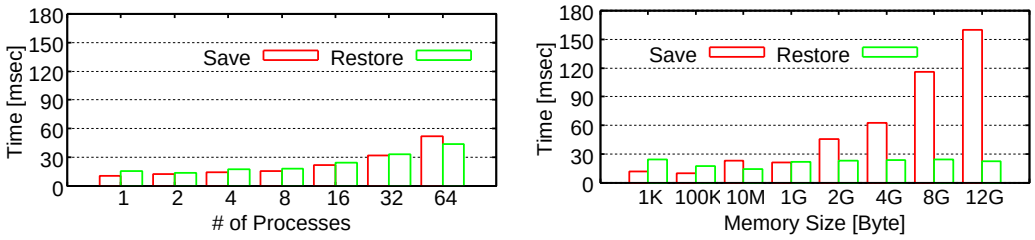
5.4 実験

Dwarf は機能に合わせて OS と VMM のそれぞれに実装を行った．One-time Checkpoint/Restart は OS に実装を行い，Linux 2.6.39.4 は 2719 行，Linux 3.18.35 と Linux 4.1.6 は 2714 行の改変を行った．Background Boot と Attachment-pipelining は主に VMM に実装を行い，スクリプトなども含めて 500 行程度の改変を行った．

Dwarf が App の実行状態を保存し，短いダウンタイムで App を再開することができることを実験によって確認する．実験環境は表 5.4 に示す．

5.4.1 マイクロベンチマーク

App の Checkpoint/Restart 時間は App の特徴によって異なる．どのような特徴の App が長い時間を要するのか，またその時間はどれほどかを調べるためにマイクロベンチマークを実行した．ベンチマーク用のプログラムは自作し，プロセス数を 1 から 64 個まで，メモリサイズを 1KB から 12GB まで，展開ファイル数を 1 から 4000 個まで，展開ソケット数を 1 から 4000 個まで，変化させた．評価を行う



(a) Process Generation (b) Memory Allocation

(c) File Opening (d) Socket Opening

図 5.5: 各資源の使用量を変化させた時の Checkpoint/Restart の所要時間

ために，Checkpoint/Restart それぞれの所要時間と，App の Checkpoint が開始してから Restart が終わるまでのサービスダウンタイムを計測した．この実験では，古い OS も新しい OS もともに Linux 2.6.39.4 で行った．結果を図 5.5 に示す．

メモリを復元する時間以外はすべてその量と正の相関がある．メモリ量を変化させた実験だけ Restart にかかる時間だけが一定になっているのは，新しい OS 上で行うメモリの復元がオンデマンドに行われているためである．App の Restart 時にはメモリのマッピングは復元しないため，Restart 後に復元されていないページにアクセスするとページフォルトが発生する．OS は Essential Context からページフォルトを起こした仮想アドレスに対応する物理ページ番号を調べて Hypercall を発行し，VMM は古い OS の物理ページの内容を用いてメモリを復元する．Checkpoint/Restart の所要時間の支配項となっているのはメモリマッピングの保存時間である．

5.4.2 サービスダウンタイムの削減

Background Boot と Attachment-pipelining による OS 再起動時間の削減が，どれだけ App のサービスダウンタイムを削減することができるのか実験を行った．この実験は Linux 2.6.39.4 で，Disk，NIC ともに PCI Passthrough を利用した環境で実行した．

図 5.6 にその結果を示す．通常の再起動を行うと 26sec 程度のダウンタイムが発生していたが，Dwarf を利用することで 5sec 程度に削減することができた．PCI Passthrough で物理デバイスを直接 VM に渡す場合，OS が直接デバイスを操作するため高スループットを達成することができる．物理デバイスの初期化処理は仮

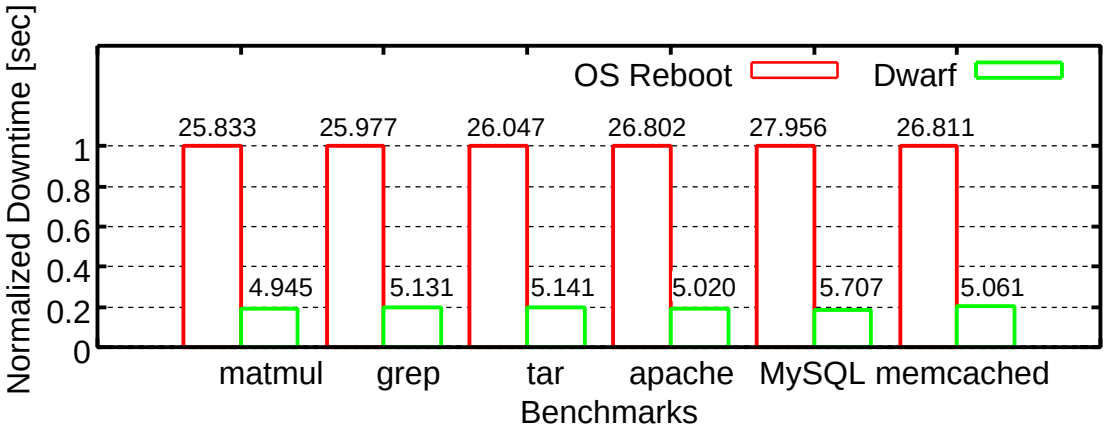


図 5.6: Dwarf によるダウンタイムの削減

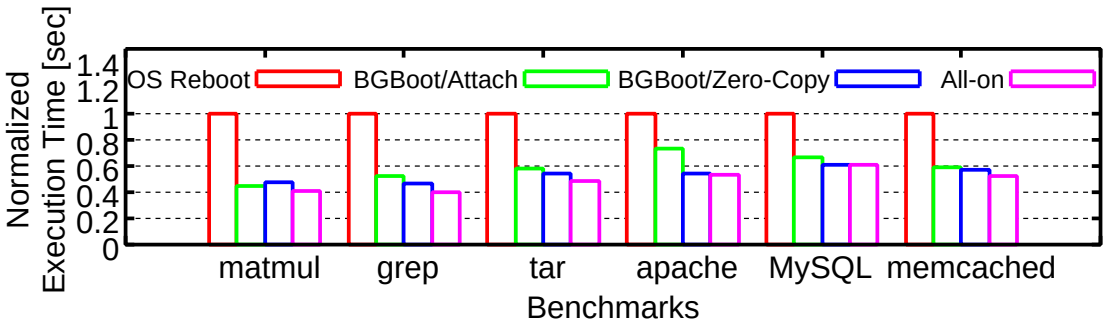


図 5.7: 各最適化手法の有効/無効を変化させたときのダウンタイムの変化

想デバイスの初期化処理に比べて時間を要するため，ダウンタイムが 5sec と長引いてしまった．

次に Background Boot，Attachment-pipelining，Zero-copy Memory Recovery の効果を確かめるために，(1) 通常の再起動，(2)Background Boot と Attachment-pipelining を有効にしたもの，(3)Background Boot と Zero-copy Memory Recovery を有効にしたもの，(4)3 手法全てを有効にしたもの，でダウンタイムの比較を行った．結果を図 5.7 に示す．最もダウンタイムを削減することに貢献した手法は Background Boot であり，ダウンタイムの半分程度を削減している．Attachment-pipelining と Zero-copy Memory Recovery の貢献は数百ミリ秒程度であるがダウンタイムの削減に貢献している．利用するデバイスが増えたり App が使用するメモリ量が増えれば，これらの手法による削減効果も大きくなることが見込まれる．

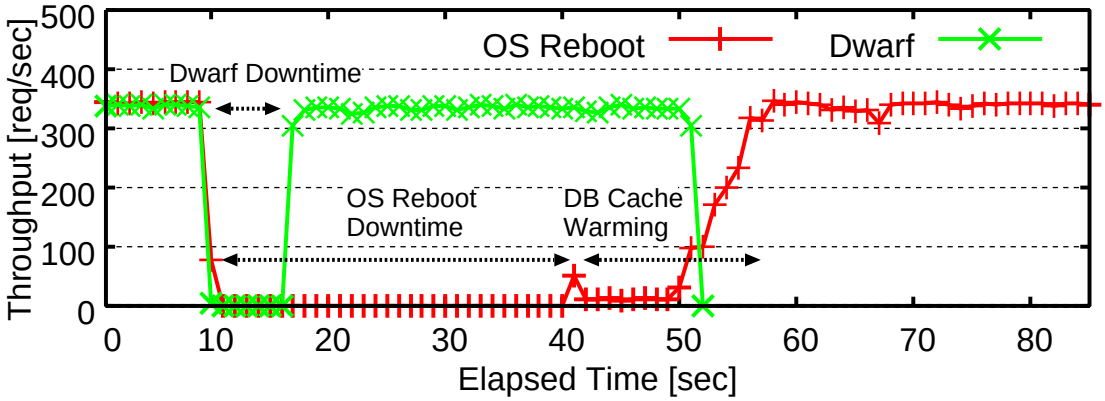


図 5.8: キャッシュを持つ App のスループット

5.4.3 App スループットの維持

DBMS はユーザ空間のメモリ上にキャッシュを持ち，DB へのアクセスを高速化している．DBMS の再起動後はキャッシュが失われてしまうため，MySQL では再起動直後にキャッシュの再構築を行う．キャッシュを再構築する処理の間にもリクエストが到来すれば，その間スループットは低下してしまう．Dwarf を利用することで MySQL はメモリ上のキャッシュを維持することができ，再起動の後もスループットが低下しないことを示す．ベンチマークには Sysbench の OLTP のワークロードを用い，1Gbps のネットワークで接続されたマシンから 8,000,000 回の SELECT クエリを実行した．

図 5.8 に示す通り，ダウンタイムは Dwarf が 5sec，通常の再起動が 30sec である．しかし通常の再起動を行った後は MySQL のキャッシュが無く，その再構築を行っている間スループットが大幅に低下している．Dwarf ではダウンタイムが短いという利点だけでなく，メモリ上のキャッシュを失わなかったことでスループットが再起動前の水準を維持することができている．

5.4.4 OS アップデートの適用

Apache Web Server を稼働させたまま，Linux 2.6.39.4 から Linux 3.18.35 にアップデートを行う．Linux はこのバージョンの間で改良され続け，特にネットワーク I/O はスループットが飛躍的に向上している．ベンチマークには ApacheBench を用い，Web サイトへの GET リクエストを繰り返したときのスループットを計測した．本実験は，デバイスは物理デバイスの PCI Passthrough 接続ではなく，仮想デバイスにて実験を行う．

結果を図 5.9 に示す．デタッチや初期化の速い仮想デバイスを用いることでダウンタイムは更に短くなり，3sec となっている．これは Apache 自体は 2.13sec で復元

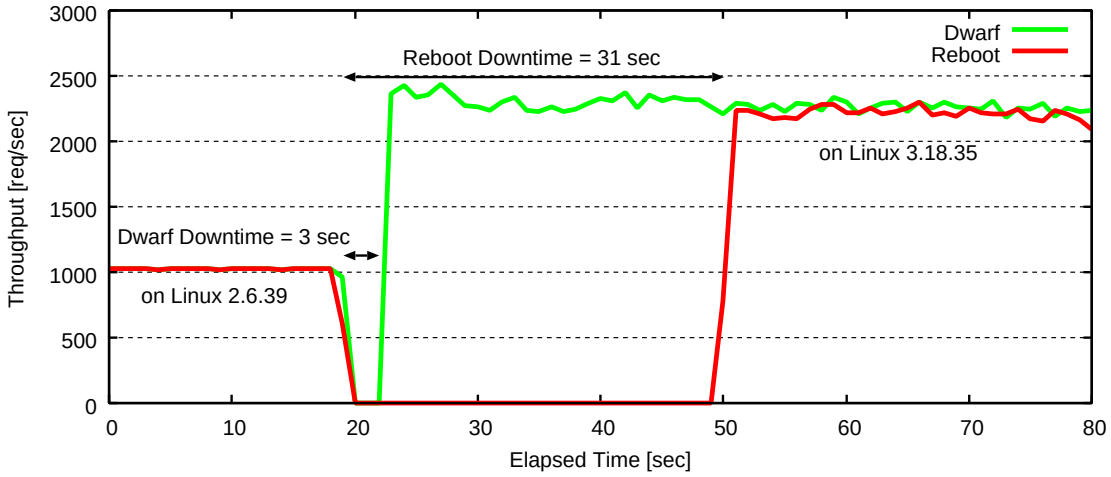


図 5.9: OS アップデート前後の ApacheBench のスループット

できたものの，TCP の再送間隔が 1sec，3sec，…となっているためである．スループットはアップデートによって 2 倍以上と飛躍的に向上し，App の実行を継続しつつアップデートの恩恵を受けることができた．実験は Linux 2.6.39.4 から Linux 4.1.6 へのアップデートも行い，同様の結果を得ることができた．Linux 2.6.39.4 の公開から Linux 3.18.35 が公開されるまでの間は約 3 年であり，大量のアップデートを含んでいる．このような大規模なアップデートを一度に適用することは Dynamic Patching にはできない．

5.5 考察

5.5.1 ダウンタイムのさらなる削減

ダウンタイムは可能な限り短いことが好ましい．Dwarf では約 2sec のダウンタイムを達成したが，この 2sec のうち多くを占めるのはデバイスのデタッチ&アタッチと OS の初期化処理である．OS のデバイスドライバ手を加えることでデタッチ&アタッチの所要時間を減らしたり，ブート処理の順序を変更することでさらなる並列化を可能とすることが考えられる．

また近年は 1TB を超えるメモリを搭載したマシンも登場しており，大量のメモリを使用する App のダウンタイムを増加させないことが必要である．Dwarf ではメモリサイズに比例して App の Checkpoint にかかる時間が増加しており，大量のメモリを使う App が稼働している場合には Checkpoint の時間がダウンタイムの支配項となる可能性がある．メモリサイズに比例しないダウンタイムを達成する Checkpoint 手法を実現することで，そのような環境であっても短いダウンタイムを達成することが可能となる．

5.6 まとめ

本章では，OS アップデート時に App の実行を引き継ぎ，そのダウンタイムを抑制する Dwarf について述べた．Background Boot は App の稼働中に稼働中にアップデート後の OS を立ち上げ，App の停止，OS の停止，マシンの再起動の 3 つの時間を並列化して隠蔽することができた．Attachment-pipelining はルートデバイス以外のデバイス切り替え処理を並列化することでダウンタイムを削減した．特にデタッチに時間がかかるデバイスに対して有効である．物理デバイスを PCI Passthrough で接続した場合は 5sec 程度，仮想デバイスを利用した場合には 2sec 程度と短いダウンタイムを達成することができた．OS のバージョンへの依存度が低い Essential Context を用いることで，3 年分のアップデートを一度に適用しつつ App の実行を継続することが可能である．

Essential Context は OS ごとに異なる形式で定義しなければならないが，Dwarf は冗長なハードウェアや特殊なハードウェアを必要とせず，また Dwarf は特定の OS 構成方法に依存するわけではないため，多くの OS で採用することができる仕組みである．また Dynamic Patching とは異なり，大規模アップデートでも利用可能であることを実証した．

第6章　結論

6.1　まとめ

コンピュータシステムの信頼性の要はOSである．しかしOSには未発見のものも含め多くのバグが含まれていることがよく知られており，コンピュータシステム全体の信頼性を低下させる要因となっている．バグの入ったコードをCPUの高い実行権限で実行するため，またOS内のFailureはその上で稼働するすべてのAppに影響を及ぼすため，OS内のバグによって引き起こされるFailureはより深刻である．OSのバグはアップデートによって取り除かれるが，OSのアップデートを適用するための再起動はAppの稼働時間を低下させ，可用性を損なってしまう．

そこで本研究では，Appの実行を再起動後のOSに引き継ぐために必要なEssential Contextを定義した．これはOSのFailureが発生した場合でも，OSのアップデートを適用する場合にも利用することができる．OSの再起動を行うことで，Failureが発生して実行が継続できないOSに変わってAppの継続実行が可能となり，またあらゆるアップデートを適用可能となる．Appは再起動直前の実行状態を保持したまま，再起動後のOS上で再開させることができる．

ShadowBuddyとDwarfの実装により，Essential ContextはVMMによっても，OSカーネルによっても作成することができることを示した．OS内でFailureが発生することを想定する場合にはVMMからAppの保護とEssential Contextの作成を行う必要があるが，OSカーネルのアップデート時に利用する場合はどちらのレイヤから作成しても良い．

本論文で提案したEssential Contextは，OSカーネル内のFailureとOSカーネルのアップデートという，どちらもコンピュータシステムの信頼性を低下させるイベントに対し，その信頼性低下を抑制することで統一的な解決を行うことができる．OSカーネル内でFailureが発生しても，再開したAppの実行状態はFailure発生前のものを維持することで保全性が保証されており，Failureから高速に復帰できることから高い保守性を実現することができる．またOSのアップデートによって発生するAppのダウンタイムについても，これを短縮することで可用性の低下を抑制することができる．DwarfではAppの一時停止から再開までのダウンタイムを2sec程度まで達成しており，通常通りのOS再起動に比べて約90%程度のダウンタイムを削減することができた．

6.2 今後の展望

6.2.1 機密性の向上

Essential Context は，短いダウンタイムで App を再開させることで可用性を向上させ，OS を確実にアップデートさせることで狭義の信頼性を向上させ，発生した Failure からの App の復元を可能とすることで保守性を向上させ，OS 内の Failure によって発生するエラープロパゲーションから App の実行状態を保護することで保全性を向上させた．そこで機密性についても向上させることで，さらなる信頼性の向上を達成することができる．

Malware の中には OS の脆弱性をつき，OS の権限を乗っ取ったり本来アクセスできない資源へのアクセスを可能とするものも存在する．OS の権限でアクセスがなされるため，App 間の Isolation を超えてアクセスが可能である．App のメモリ内容を Malware が取得することで，パスワードや暗号鍵などの Sensitive Data が流出してしまう可能性がある．OS は Attack Surface が広いため，その脆弱性をついた Rootkit による攻撃が成立しやすい．Rootkit が狙う OS と App のインタフェース，すなわち API は多岐にわたっている．例えば Linux では 300 を超えるシステムコールが実装されており，またその引数も豊富で OS 内の関数の処理を切り替えることができる．そのため十分にテストされていないコードも存在し，脆弱性が残されたままになることもある．

ShadowBuddy の EPT-level Protection では，OS 内の Failure を想定して OS のエラープロパゲーションが App の持つユーザ空間のメモリを不正に書き換えることを防いでいた．そこで EPT の設定を変更し，OS が App のメモリ空間の読み込みを行うことも禁止することで Malware の攻撃によって App のメモリ内容が流出することを防ぐことが可能である．システムコールに利用する引数やそのメモリ内容は流出する可能性があるが，その他のスタックやヒープ上のメモリオブジェクトは OS から参照することができなくなる．すべての Malware からの攻撃を防ぐことができるわけではないが，OS の脆弱性をついたメモリ読み込みを行う Malware の攻撃は防ぐことができる．

6.2.2 Failure や App への攻撃の検知

OS 内の Failure を検知する方法には，OS 内でメモリオブジェクトに不正な値が格納されていないかを検査したり，ハードウェアで起こった例外のハンドラ内で判断を行ったりするものが存在する．OS 内で発生した Failure や脆弱性をついた攻撃を素早く検知し，OS を再起動することで信頼できない OS 上での App の稼働をより安全にすることができる．

ShadowBuddy では新たに，App のメモリ空間への書き込みを禁止することで，App が意図しない領域への OS からの書き込みを Failure として検知することがで

きた．OS のメモリ空間内に存在しているページテーブルは ShadowBuddy のプロトタイプでは保護の対象外であった．そこで OS 内のページテーブルを書き込み禁止にし，その変更を逐一監視する手法によって，ページテーブルを不正に操作することによって App の Control Flow を攻撃する Iago Attack [113] を検知することができる．ただし，

Essential Context とは本来 App の実行再開に必要な Checkpoint を指していたが，これを作成するために採用された ShadowBuddy という手法が Failure や App への攻撃を検知できるという副次的な効果を得ることができた．OS からの App への不正な書き込み，OS によるページテーブルの破壊以外にも検知可能な Failure・攻撃を増やすことで，より信頼性の高いコンピュータシステムを構築することが可能となる．

6.2.3 OS の Multi-version Execution

複数バージョンのプログラムを並列に実行し，その挙動の履歴からプログラムの不具合を検出する手法は Multi-version Execution，N-version Execution，Multi-Variant Execution などと呼ばれている．あるプログラムのアップデートによってバグが導入されてしまった場合，その前のバージョンとアップデート後のバージョンとを平行あるいは並列に実行した場合，プログラムの挙動が変わることがある．この挙動の変化を検知し原因を特定することで，バグの検出をすることができる．またあるバージョンで Fault が発生してしまったとしても，他のバージョンのものは引き続き実行可能であり，Fault に強い実行環境とすることができる．さらにあるバージョンの脆弱性をついた攻撃が実行されたとしてもその結果はバージョンごとに異なり，その結果の違いから攻撃を検知することもできる．このように Multi-version Execution は狭義の信頼性と保全性を向上させることができる．

Multi-version Execution は App に対しては実現されている [114, 115, 116] が，OS に対しては実現されていない．複数バージョンの OS を実行することで，OS 内のバグの検出や Malware からの攻撃の検知，Fault が発生しても別バージョンでの継続実行など，信頼性を向上させることができる．OS のアップデートが発生した時には，並列実行しているバージョンに加える．Essential Context によって新しいバージョンの OS においても同じ App の実行状態で起動することができる．あるバージョンの OS 内で Fault が発生したとしても，Essential Context は OS から保護されているため別のバージョンには影響がない．Essential Context を Multi-version Execution と組み合わせることで，より信頼性の高いコンピュータシステムとすることが可能である．

6.2.4　他のシステムソフトウェアへの適用

本研究は特定の OS 構成方法に依存しないよう設計を行ったが，実際には Linux と Xen に対して実装を行った．他の OS にも本研究の手法を適用することで，この目的を達成していることを検証することができる．特に Library OS や Xen より軽量な VMM に適用することで，高速かつ高信頼な App 実行環境を達成することが見込める．

謝辞

本論文をまとめるまでに多くの方々からご協力とご指導をいただきました．お世話になった方々に，心から感謝いたします．

特に直接指導をしていただいた山田浩史准教授に深く感謝いたします．東京農工大学学部 4 年生から博士課程 3 年までの 6 年間にわたって指導をいただきました．実装がうまくいかない時，アイディアが行き詰まった時，間違った方向に研究を進めそうになった時，適切なアドバイスをしていただきました．修士課程までで卒業し就職しようかと思っていたときも，もっと楽しい研究ができると進学の道を示してくださいました．心が折れそうになった時にも励ましていただき，時には夜遅くまで作業をしていただきました．深く感謝いたします．

廊下ですれ違うたびに，学会でご一緒するたびに，研究の進み具合を尋ねてくださった並木美太郎教授にも感謝します．その幅広い分野への知識から自分の研究の位置づけを教えていただき，またその専門知識から自身の研究の行く先まで知ることができました．

また，藤波香織教授，近藤敏之教授，藤田桂英准教授には，貴重な時間を割いて本論文を査読していただき，ご指導とご助言をいただきました．深く感謝いたします．

同じ研究室で学んだメンバーにも感謝します．ゼミや普段の会話を通して知識を共有し，強く刺激を受けました．自分の研究とは少し離れた分野についての知識を得ることができ，そのアイディアのいくらかは自身の研究に取り込むことができました．

妻には迷惑をかけ，また寂しい思いをさせました．没頭すると時間を気にしなくなり，夜遅くや朝まで作業を続けてしまい，すれ違うこともありました．まともに家事ができない時もありました．それでも励まし支え続けてくれたこと，とても心強く感じました．

学位を申請するにあたり，支えてくださったすべての方に，心から感謝したします．

論文目録

定期刊行誌掲載論文

- Ken Terada and Hiroshi Yamada, “Shortening Downtime of Reboot-based Kernel Updates using Dwarf”. In *IEICE TRANSACTIONS on Information and Systems*, Vol.E98-A, No.1, pp.1–14, Sep. 2018.

国際会議論文

- Ken Terada and Hiroshi Yamada, “Dwarf: Shortening Downtime of Reboot-based Kernel Updates”. In *Proc. of the 12th European Dependable Computing Conference (EDCC ’16)*, pp.208–217, Sep. 2016.

研究会報告

- 寺田 献, 山田 浩史, “OS クラッシュからのプロセスコンテキスト保護・再開手法”, 第 141 回情報処理学会 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, Vol.2017-OS-141, No.5, pp.1–7, Jul. 2017. (最優秀学生発表賞)

- 寺田 献, 山田 浩史, “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”, 第 144 回情報処理学会 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, Vol.2018-OS-144, No.4, pp.1–8, Jul. 2018. (CS 領域奨励賞)

参考文献

[1] ITIC: Hourly Downtime. http://itic-corp.com/blog/2017/05/hourly-downtime-tops-300k-for-81-of-firms-33-of-enterprises-say-downtime-costs-1m/.

[2] Robert F. Dacey. Effective Patch Management is Critical to Mitigating Software Vulnerabilities. Technical report, Government Accountability Office, Sep. 2003.

[3] Jonathan Corbet, Greg Kroah-Hartman, and Amanda McPherson. Linux Kernel Development – How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It, Dec. 2010. https://www.linuxfoundation.org/docs/lf_linux_kernel_development_2010.pdf.

[4] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Oct. 2001.

[5] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How Do Fixes Become Bugs? – A Comprehensive Characteristic Study on Incorrect Fixes in Commercial and Open Source Operating Systems. In *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 26–36, Sep. 2011.

[6] Aakash Goel, Bhuwan Chopra, Ciprian Gerea, Dhrúv Mátáni, Josh Metzler, Fashim Ul Haq, and Janet L. Wiener. Fast Database Restarts at Facebook. In *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*, pages 541–549, Jun. 2014.

[7] Olivier Crameri, Nikola Knežević, Dejan Kostić, Ricardo Bianchini, and Willy Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. In *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 221–236, Oct. 2007.

[8] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Providing dynamic update in an operating

system. In *Proc. of the USENIX Annual Technical Conference (USENIX '05)*, pages 279–291, Apr. 2005.

[9] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a Complete Operating System. In *Proc. of the 1st ACM European Conference on Computer Systems (EuroSys '06)*, pages 133–145, Apr. 2006.

[10] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. In *Proc. of the USENIX Annual Technical Conference (ATC '07)*, pages 337–350, Jun. 2007.

[11] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma M. Da Silva, Orran Krieger, David J. Edelsohn Marc A. Auslander, Ben Gamsa, Gregory R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1), 2003.

[12] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and Automatic Live Update for Operating Systems. In *Proc. of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 279–292, Mar. 2013.

[13] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling Cores, Kernels, and Operating Systems. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 17–31, Oct. 2011.

[14] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live Updating Operating Systems Using Virtualization. In *Proc. of the 2nd ACM International Conference on Virtual Execution Environments (VEE '06)*, pages 35–44, Jun. 2006.

[15] Kristis Makris and Kyung D. Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *Proc. of the 2nd ACM European Conference on Computer Systems (EuroSys '07)*, pages 327–340, Mar. 2007.

[16] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *Proc. of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, pages 187–198, Apr. 2009.

[17] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. Instant OS Updates via Userspace Checkpoint-and-Restart. In *Proc. of the 2016 USENIX Annual Technical Conference (ATC '16)*, pages 605–619, Jun. 2016.

[18] Werner Almesberger. kboot—A Boot Loader Based on Kexec. In *Proc. of the Ottawa Linux Symposium (OLS '06)*, volume 1, pages 27–38, Jul. 2006.

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

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

[21] Maxim Siniavine and Ashvin Goel. Seamless Kernel Updates. In *Proc. of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, pages 1–12, Jun. 2013.

[22] Jason Duell, Paul Hargrove, and Eric Roman. Requirements for Linux Checkpoint/Restart. Technical Report 8, Berkeley Lab Technical Report, LBNL-49659, May 2002.

[23] Michael Rieker, Jason Ansel, and Gene Cooperman. Transparent User-Level Checkpointing for the Native POSIX Thread Library for Linux. In *Proc. of 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '06)*, pages 492–498, Jun. 2006.

[24] Alex Depoutovitch and Michael Stumm. Otherworld - Giving Applications a Change to Servive OS Kernel Crashes. In *Proc. of the 5th ACM European Conference on Computer Systems (EuroSys '10)*, pages 181–194, Apr. 2010.

[25] Andrew Lenharth, Vikram Adve, and Samuel T. King. Recovery domains: An organizing principle for recoverable operating systems. In *Proc. of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 49–60, Mar. 2009.

[26] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing OS Extension Safely and Efficiently with Bascule. In *Proc. of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, pages 239–252, Apr. 2013.

[27] Dirk Vogt, Cristiano Giuffrida, Herbert Bos, and Andrew S. Tanenbaum. Lightweight Memory Checkpointing. In *Proc. of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, pages 474–484, Sep. 2015.

[28] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 207–220, Oct. 2009.

[29] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32:1–70, Feb. 2014.

[30] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A Trustworthy In-kernel Interpreter Infrastructure. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 33–47, Oct. 2014.

[31] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 653–669, Nov. 2016.

[32] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 165–181, Oct. 2014.

[33] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, Sep. 1996.

[34] Jochen Liedtke. On μ-Kernel Construction. In *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 237–250, Dec. 1995.

[35] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Reorganizing UNIX for Reliability. In *Proc. of the 11th Asia-Pacific Computer Systems Architecture Conference (ACSAC '06)*, pages 81–94, Sep. 2006.

[36] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 59–72, Dec. 2008.

[37] Donald E. Porter, Galen Hunt, Jon Howell, Reuben Olinsky, and Silas Boyd-Wickizer. Rethinking the Library OS from the Top Down. In *Proc. of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 291–304, Mar. 2011.

[38] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proc. of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 461–472, Mar. 2013.

[39] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv— Optimizing the Operating System for Virtual Machines. In *Proc. of the 2014 USENIX Annual Technical Conference (ATC '14)*, pages 61–72, Jun. 2014.

[40] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *Proc. of the 9th ACM European Conference on Computer Systems (EuroSys '14)*, pages 1–14, Apr. 2014.

[41] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proc. of the 2017 USENIX Annual Technical Conference (ATC '17)*, pages 645–658, Jul. 2017.

[42] Ruslan Nikolaev and Godmar Back. VirtuOS: an operating system with kernel virtualization. In *Proc. of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 116–132, Nov. 2013.

[43] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proc. of the 5th ACM European Conference on Computer Systems (EuroSys '10)*, pages 209–222, Apr. 2010.

[44] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proc. of the USENIX Annual Technical Conference (USENIX '01)*, pages 1–14, Jun. 2001.

[45] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and Art of Virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, Oct. 2003.

[46] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux Virtual Machine Monitor. In *Proc. of the Ottawa Linux Symposium (OLS '07)*, pages 225–230, Oct. 2007.

[47] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The Xen-Blanket: Virtualize Once, Run Everywhere. In *Proc. of the 7th ACM European Conference on Computer Systems (EuroSys '12)*, pages 113–126, Apr. 2012.

[48] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, Mar. 2008.

[49] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. of the 2010 IEEE Symposium on Security and Privacy (S&P '10)*, pages 143–158, May 2010.

[50] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proc. of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 265–278, Mar. 2013.

[51] Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. Efficient Virtualization-Based Application Protection Against Untrusted Operating System. In *Proc. of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS '15)*, pages 345–356, Apr. 2015.

[52] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 533–549, Nov. 2016.

[53] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proc. of the 11th USENIX Symposium*

on Operating Systems Design and Implementation (OSDI '14), pages 267–283, Oct. 2014.

[54] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O 'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 689–703, Nov. 2016.

[55] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *Proc. of the 12th ACM European Conference on Computer Systems (EuroSys '17)*, pages 238–253, Apr. 2017.

[56] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proc. of the 12th ACM European Conference on Computer Systems (EuroSys '17)*, pages 437–452, Apr. 2017.

[57] Michael Grottke, Dong Seong Kim, Rajesh Mansharamani, Manoj Nambiar, Roberto Natella, and Kishor S. Trivedi. Recovery from software failures caused by mandelbugs. *IEEE Transactions on Reliability*, 65(1):70–87, Mar. 2016.

[58] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically Patching Errors in Deployed Software. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 87–102, Oct. 2009.

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

[60] Rebecca Smith and Scott Rixner. Surviving Peripheral Failures in Embedded Systems. In *Proc. of the 2015 USENIX Annual Technical Conference (ATC '15)*, pages 125–137, Jul. 2015.

[61] Cristiano Giuffrida, Cüalin Iorgulescu, Anton Kuijsten, and Andrew S. Tanenbaum. Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer. In *Proc. of the 27th USENIX Large Installation System Administration Conference (LISA '13)*, pages 89–104, Nov. 2013.

[62] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 29–44, Oct. 2009.

[63] David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Mini-Ckpts: Surviving OS Failures in Persistent Memory. In *Proc. of the 2016 International Conference on Supercomputing (ICS '16)*, pages 1–14, Jun. 2016.

[64] CRIU. `https://www.criu.org/`.

[65] Koustubha Bhat, Dirk Vogt, Erik van der Kouwe, Ben Gras, Lionel Sambuc, Andrew S. Tanenbaum, Herbert Bos, and Cristiano Giuffrida. OSIRIS: Efficient and Consistent Recovery of Compartmentalized Operating Systems. In *Proc. of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '16)*, pages 25–36, Jun. 2016.

[66] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proc. of the 5th USENIX Symposium on Networked Systems Design and implementation (NSDI '08)*, pages 161–174, Apr. 2008.

[67] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. RemusDB: Transparent High Availability for Database Systems. *The international Journal on Very Large Data Bases*, 22(1):29–45, Feb. 2013.

[68] Jacob R. Lorch, Andrew Baumann, Lisa Glendenning, Dutch T. Meyer, and Andrew Warfield. Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services. In *Proc. of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, pages 575–588, May 2015.

[69] Bogdan Nicolae and Franck Cappello. BlobCR: Efficient Checkpoint-Restart for HPC Applications on IaaS Clouds using Virtual Disk Image Snapshots. In *Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pages 1–12, Nov. 2011.

[70] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems. In *Proc. of the 48th International Symposium on Microarchitecture (MICRO '15)*, pages 672–685, Dec. 2015.

[71] Lei Cui, Tianyu Wo, Bo Li, Jianxin Li, Bin Shi, and Jinpeng Huai. PARS:A Page-Aware Replication System for Efficiently Storing Virtual Machine Snapshots. In *Proc. of the 11th ACM International Conference on Virtual Excution Environments (VEE '15)*, pages 215–228, Mar. 2015.

[72] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 361–376, Dec. 2002.

[73] Shaya Potter and Jason Nieh. Reducing downtime due to system maintenance and upgrades. In *Proc. of the 19th USENIX Large Installation System Administration Conference (LISA '05)*, pages 47–62, Dec. 2005.

[74] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable Virtual Machines Enabling General, Single-Node, Online Maintenance. In *Proc. of the 11th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '04)*, pages 211–223, Oct. 2004.

[75] Jinyu Gu, Zhichao Hua, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. Secure Live Migration of SGX Enclaves on Untrusted Cloud. In *Proc. of the 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '17)*, pages 225–236, Jun. 2017.

[76] docker. `https://www.docker.com/`.

[77] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proc. of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, pages 273–286, May 2005.

[78] Michael R. Hines and Kartik Gopalan. Post-Copy Based Live Virtual Macine Migration Using Adaptie Pre-Paging and Dynamic Self-Ballooning. In *Proc. of the 2009 ACM International Conference on Virtual Execution Environments (VEE '09)*, pages 51–60, Mar. 2009.

[79] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Oct. 2001.

[80] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-Button Verification

of an OS Kernel. In *Proc. of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, pages 252–269, Oct. 2017.

[81] Xavier Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Proc. of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*, pages 42–54, Jan. 2006.

[82] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep Specifications and Certified Abstraction Layers. In *Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*, pages 595–608, Jan. 2015.

[83] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proc. of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, pages 18–37, Oct. 2015.

[84] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proc. of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, pages 38–53, Oct. 2015.

[85] Fraser Brown, Andres Nötzli, and Dawson Engler. How to Build Static Checking Systems Using Orders of Magnitude Less Code. In *Proc. of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, pages 143–157, Mar. 2016.

[86] Adam Chlipala. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. In *Proc. of the 32nd ACM Conference on Programming Language Design and Implementation (PLDI '11)*, pages 234–245, Jun. 2011.

[87] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button Verification of File Systems via Crash Refinement. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 1–16, Nov. 2016.

[88] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Maziéres, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 335–348, Oct. 2012.

[89] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proc. of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, pages 191–206, Mar. 2015.

[90] Linux-VServer Project. `http://linux-vserver.org/`.

[91] Linux Containers. `https://linuxcontainers.org/`.

[92] Oracle Solaris Container. `http://oracle.com/solaris`.

[93] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, and Nadav Har 'El. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 423–436, Oct. 2010.

[94] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proc. of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, pages 201–207, Oct. 2017.

[95] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *Proc. of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, pages 218–233, Oct. 2017.

[96] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proc. of the 2nd ACM European Conference on Computer Systems (EuroSys '07)*, pages 275–287, Mar. 2007.

[97] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 207–222, Oct. 2003.

[98] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems (TOCS)*, 23:77–110, Feb. 2005.

[99] Michael M. Swift, Muthukaruppan Annamalai, Braian N. Bershad, and Henry M. Levy. Recovering Device Drivers. In *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 1–16, Dec. 2004.

[100] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering Device Drivers. *ACM Transactions on Computer Systems (TOCS)*, 24:333–360, Nov. 2006.

[101] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path. In *Proc. of the 2017 USENIX Annual Technical Conference (ATC '17)*, pages 1–13, Jul. 2017.

[102] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proc. of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*, pages 79–93, May 2009.

[103] Justin Cappos, Armon Dadgar, Jeff Rasley, Justin Samuel, Ivan Beschastnikh, Cosmin Barsan, Arvind Krishnamurthy, and Thomas Anderson. Retaining Sandbox Containment Despite Bugs in Privileged Memory-Safe Code. In *Proc. of the 17th ACM Conference on Compute and Communications Security (CCS '10)*, pages 212–223, Oct. 2010.

[104] Artem Dinaburg, Paul Royal †, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proc. of the 15th ACM Conference on Compute and Communications Security (CCS '08)*, pages 51–62, Oct. 2008.

[105] Rui Wu, Ping Chen, Peng Liu, and Bing Mao. System Call Redirection: A Practical Approach to Meeting Real-world Virtual Machine Introspection Needs. In *Proc. of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*, pages 574–585, Jun. 2014.

[106] Monirul Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure In-VM Monitoring Using Hardware Virtualization. In *Proc. of the 16th ACM Conference on Compute and Communications Security (CCS '09)*, pages 477–487, Nov. 2009.

[107] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs. In *Proc. of the 2018 USENIX Annual Technical Conference (ATC '18)*, pages 255–266, Jul. 2018.

[108] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User

Space. In *Proc. of the 27th USENIX Security Symposium (USENIX Security '18)*, pages 973–990, Aug. 2018.

[109] Kenichi Kourai and Shigeru Chiba. A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines. In *Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pages 245–255, Jun. 2007.

[110] Franz Ferdinand Brasser, Mihai Bucicoiu, and Ahmad-Reza Sadeghi. Swap and Play: Live Updating Hypervisors and Its Application to Xen. In *Proc. of the 6th ACM Cloud Computing Security Workshop (CCSW '14)*, pages 33–44, Dec. 2014.

[111] Michael Le and Yuval Tamir. ReHype: Enabling VM SurvivalAcross Hypervisor Failures. In *Proc. of the 7th ACM International Conference on Virtual Excution Environments (VEE '11)*, pages 63–74, Mar. 2011.

[112] Diyu Zhou and Yuval Tamir. Fast Hypervisor Recovery Without Reboot. In *Proc. of the 48th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '18)*, pages 115–126, Jun. 2018.

[113] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proc. of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 253–264, Mar. 2013.

[114] Petr Hosek and Cristian Cadar. Safe Software Updates via Multi-version Execution. In *Proc. of the 35th International Conference on Software Engineering (ICSE '13)*, pages 612–621, May 2013.

[115] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. In *Proc. of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, pages 339–353, Mar. 2015.

[116] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. Taming parallelism in a multi-variant execution environment. In *Proc. of the 12th ACM European Conference on Computer Systems (EuroSys '17)*, pages 270–285, Apr. 2017.