Vol.2016-OS-136 No.18
2016/3/1

リブートを伴うOSアップデートにおける ダウンタイム削減手法

寺田 献[†1,a)]　山田 浩史[†1,b)]

概要：脆弱性の修正や高速化のためにソフトウェアのアップデートを行うことは，サービスの信頼性を確保するために必要不可欠である．しかしながら OS のアップデートのために行う再起動は，その上で稼働する全てのプロセスを再起動することになり，その間に発生するサービスのダウンタイムによって可用性を損なってしまう．本研究では，短いサービスダウンタイムを達成するために，アップデート後の OS をバックグラウンドで用意する Background Boot，プロセスの引き継ぎにかかる時間を短縮する Zero-copy Process Migration，デバイスの引き継ぎをプロセスの引き継ぎと平行して行う Attach-pipelining を提案する．memcached を用いた実験では，プロセスを引き継ぎつつ再起動にかかる時間の約 87%を削減することができた．

キーワード：オペレーティングシステム，アップデート，信頼性

1. はじめに

オペレーティングシステム（OS）のアップデートは，バグや脆弱性の修正，機能の追加，プログラムの高速化といった点で必要不可欠である．コンピュータ全体を管理する OS にアップデートが反映されないことは，システム全体の信頼性を大幅に低下させる．実行されるアプリケーションプログラム（AP）が適切にアップデートされていたとしても，OS 内のバグによるサービスの異常終了，脆弱性を利用された攻撃による情報漏えい，低速な OS がボトルネックになるなど，様々な問題が発生する．

しかしながら，OS のアップデートを適用するための再起動に伴い，動作していた AP のプロセスが持つメモリ上のコンテキストは失われ，OS の再起動が終わってプロセスの再起動が完了するまではダウンタイムが発生する．OS の再起動には起動していたプロセスの停止，OS の終了，ハードウェアの停止，OS の起動，ハードウェアの初期化，といった処理を伴うため，この間のダウンタイムは数十秒から場合によっては数分かかることもある．また AP 再起動の際には低速なディスクなどからプログラムやデータを読み込んだり，ネットワークコネクションを再接続したりするが，これらの時間もまたサービスのダウンタイムを伸ばす原因である．

本論文では OS のアップデートを適用しつつ，プロセスのダウンタイムを短縮する手法を提案する．OS 再起動中のダウンタイムを削減するために，退避処理を行う前にアップデート後の OS をバックグラウンドの仮想マシン上で起動しておき，各プロセスを引き継がせる．同時に利用できないハードウェアの割り当ては，プロセスの引き継ぎと同時に切り替える．プロセスやデバイスの引き継ぎ処理は同一物理マシン内で完結するため，メモリ内容やストレージ内容を転送する必要がなく，通常のプロセスマイグレーションのように大量のネットワークトラフィックを発生させることはない．アップデートに伴う OS 内のデータ構造の変更に対応するために，アップデートによって変更されることのないプロセスの実行状態を退避するデータ構造を定義する．プロセスの復元に最低限必要なメモリオブジェクトを *Essential Context* と呼ぶ．OS のアップデートによって変更されることのない，API やハードウェアインタフェース，プロトコルに準拠した形式で保存する．

提案手法を Xen 4.5.0[15] および Linux 4.1.6 に実装した．退避処理・復元処理の所要時間はプロセスの持つメモリサイズに強く依存し，1GB 程度のメモリを持つプロセスであれば退避処理に 33ms，復元処理に 0.70ms 程度でプロセスコンテキストの維持が可能であった．通常のリブートでは memcached の停止から再起動を経て再開するまでのダウ

†1 現在，東京農工大学
Presently with Tokyo University of Agriculture and Technology
a) terada@asg.cs.tuat.ac.jp
b) hiroshiy@cs.tuat.ac.jp

© 2016 Information Processing Society of Japan

表 1 プロセスコンテキスト中から選出した Essential Context
Table 1 General Essential Context

資源	内容
CPU	レジスタの値，プロセス ID，親子・兄弟関係
メモリ	メモリ領域の使用状況
I/O	パイプ，ファイル，ソケットなど
通知	メッセージ／シグナルのコールバック関数

ンタイム 43.4 秒を，提案手法によって新しく起動した OS 上で高速に復元し 5.8 秒に短縮した．

2. 背景

OS のアップデートを行うことで，脆弱性を修正して信頼性を向上したり，機能の追加，既存のプログラムの高速化をしたりすることができる．しかしながら，OS のアップデートは再起動を伴う．その上で動作していたプロセスも終了する必要があり，プロセスの終了，OS の再起動，プロセスの再起動中はプロセスが提供していたサービスが利用できないダウンタイムとなる．プロセスの終了に伴って，メモリ上のコンテキストは失われる．サーバプログラムではキャッシュやネットワークコネクション，ワードプログラムでは Undo ログなどを失うことで，性能劣化や機能低下につながる．さらに OS のアップデートによってプロセスとの互換性がなくなり，プロセスが動作しなくなるということもある．このような背景もあり，OS のアップデートを敬遠するユーザも多く，その結果既知の脆弱性を攻撃されることにも繋がっている．既知の脆弱性を利用した攻撃は全体の 90%を占めている [7] ことからも分かる通り，OS のアップデートが適用されないことは深刻な問題である．

3. 提案

OS のアップデートを適用し，高速にプロセスの引き継ぎを行う手法を提案する．

3.1 Essential Context

アップデート後の OS でプロセスコンテキストを復元するために最低限必要となるプロセスコンテキストを Essential Context と呼ぶ．Essential Context 中に含めるべきプロセスコンテキストは次の通りである．実行中や停止中といったプロセスの実行状態，仮想メモリアドレスと実際に利用している物理メモリアドレス，そのページの実行可能，読み込み可能，書き込み可能といった属性，展開しているファイルのパスや展開フラグなどが挙げられる．これらは OS の設計や実装に依存するものであるが，Windows，Macintosh OS X や Linux などの Unix 系といった汎用の OS では共通して**表 1** に示す情報によってプロセスを管理している．これらに関連するデータ構造や，プロセスが実行するシステムコールに必要となる引数などの情報を元に，各 OS に対して退避・復元すべきプロセスコンテキストを選定する．

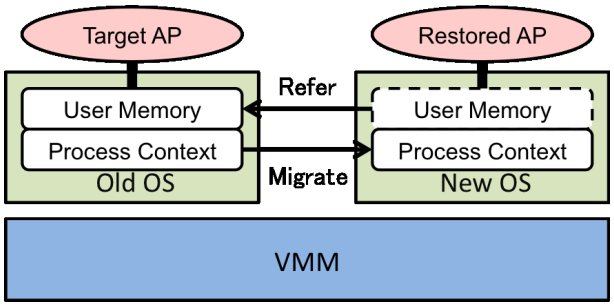

図 1 メモリ参照によるプロセスマイグレーション
Fig. 1 Zero-copy Process Migration

プロセスコンテキストは OS カーネル内のデータ構造で管理されている．このデータ構造は，OS のアップデートによって変更する可能性がある．Essential Context として保存するデータを OS のバージョンに依存しない形式とするために，API や ABI，ハードウェアインタフェース・プロトコルに準拠して定義を行う．OS 内の関数やデータ構造が変わったとしても，API や ABI といったプロセスと OS との連携インタフェースはほとんど変化しない．また後方互換性を確保するために，追加されることはあっても，削除されることは無い．ハードウェアも同様で，OS のバージョンが変わってもその仕様が変わることはない．これらのインタフェースに準拠する形式で Essential Context を保存することで，OS のバージョンに依存しないプロセスコンテキストを定義することができる．

3.2 Zero-copy Process Migration

通常のプロセスマイグレーションでは，あるプロセスが稼働中のマシンから別のマシンに，プロセスのメモリやディスク装置などの中身をネットワークを経由して送信する．しかし提案手法では，同一マシン内でプロセスの引き継ぎを行うため，メモリ内容の転送を行う必要はない．新しく起動した OS は，ページテーブルを正しく設定することで，**図 1** のようにプロセスが持っていたメモリを直接参照することができる．

3.3 Background Boot

主に OS の終了・起動にかかる時間を省略，並列実行することで，プロセスの再開までの時間を短縮する．①ゲスト OS はアップデートを適用した OS イメージを作成しハイパーバイザに通知する．②ハイパーバイザはバックグラウンドで VM を作成し，アップデートを適用した OS を起動する．③アップデート後の OS はデバイスを利用する直前で OS の初期化処理を一時停止する．④ゲスト OS は Essential Context を OS 内から抽出してプロセスを停

© 2016 Information Processing Society of Japan

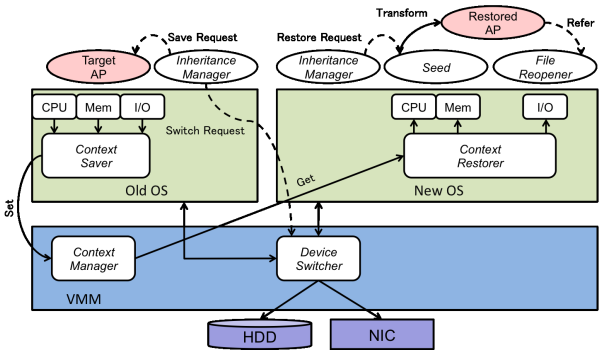

図 2 提案機構
Fig. 2 Proposal System

止させる，⑤ハイパーコールによってハイパーバイザに保存する．⑥ハイパーバイザはアップデート後のゲスト OS にデバイスを引き渡し，起動を再開させる．⑦アップデート後の OS は復元ハイパーコールでハイパーバイザから退避していた Essential Context を受け取る．⑧Essential Context を元にプロセスを復元する．プロセス ID など，コンフリクトが懸念される資源は，該当資源の初期化時に使用フラグを立てるなどしてコンフリクトを避ける．

3.4 Attach-pipelining

OS に割り当てられているデバイスの切り替えを，プロセスの引き継ぎと平行して行う．ディスク装置や NIC は仮想化していないため，同時に複数の OS から操作されると誤作動を起こす．そのため，アップデート前の OS がデバイスの利用を終えた後，動作していた VM からデタッチし，アップデート後の OS が動作する VM にアタッチする．ディスク装置などはバッファキャッシュを持つため，アップデート前の OS はデバイスの状態を同期してからデタッチを行う．アップデート後の OS はデバイスを受け取った後，マウントや IP アドレスの取得など，デバイスを利用する OS の初期化を行う．

4. 実装

ハイパーバイザに Xen 4.5.0, ホスト OS に Linux 3.10.87, 準仮想化ゲスト OS に Linux 4.1.6 を使用した．ハイパーバイザのコードを約 500 行，ホスト OS のコードを約 50 行，ゲスト OS のコードを約 3,500 行変更した．ユーザ空間でプロセスの退避・復元を管理するプログラムは約 1,000 行である．Linux でプロセスコンテキストを Essential Context に退避し，プロセスコンテキストを復元する手法について述べる．**図 2** に提案システムの構成を，**図 3** に Essential Context を格納するデータ構造の構成を示す．

4.1 プロセスコンテキストの退避

プロセスコンテキストの退避処理は，Inheritance Manager から各プロセスに退避開始を通知するシグナルを送信

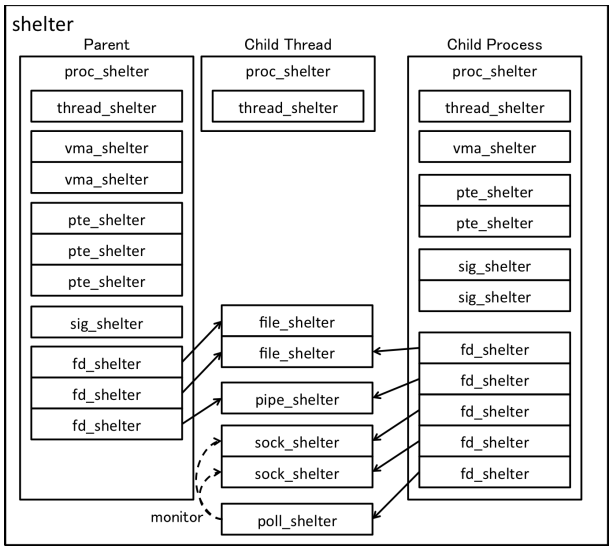

図 3 Linux 用の Essential Context のデータ構造
Fig. 3 Linux specific Essential Context

することによって始める．シグナルを受け取った各プロセスは，自らのプロセスコンテキストを取得した上でプロセスの実行を停止させる．Essential Context 全体を格納するデータ構造として shelter 構造体，プロセスとスレッドを格納するデータ構造として proc_shelter 構造体を定義する．

CPU レジスタ値を格納する thread_struct 構造体, pt_regs 構造体は動作している CPU に大きく依存する．各種レジスタのビット幅はアーキテクチャに依存しているため，OS のアップデートの前後では型が変化しない．したがって，各レジスタの値は通常の unsigned long 等の 64 ビットの型に直接代入することができる．両構造体から得られた値は，thread_shelter 構造体として定義したデータ構造に格納する．

メモリ領域の保存のために，プロセスのテキスト・静的・ヒープ・スタック領域の開始・終了アドレスなどを保存する mm_struct 構造体と，連続するメモリ領域の仮想アドレスの開始・終了アドレス，読み書きの属性の情報を保存する vm_area_struct 構造体からデータを読み込む．mm_struct 構造体内のプロセス独自のコード，テキスト，ヒープ，スタック領域の開始・終端アドレスは proc_shelter 構造体に，vm_area_struct 構造体内の領域の開始・終端アドレス，読み書き実行の参照属性，参照位置についての情報は vma_shelter 構造体内に格納する．仮想アドレスは OS のバージョンに依存せず，またフラグは API で利用するフラグを利用する．Linux のページテーブルエントリは pteval_t 型の変数であり，この値は MMU によって直接参照される値である．ハードウェアインタフェースによって定められた形式であるため，直接値を保存し，再利用することができる．ページが割り当てられているメモリ領域に対して，この値と 4KB ごとの仮想アドレスとを対にして保存する．

© 2016 Information Processing Society of Japan

ファイルはパスを取得し，展開モードとともに file_shelter に保存する．パイプは 2 つのファイルディスクリプタを持つが，それぞれから展開モードを取得し，パイプバッファ内のデータとともに pipe_shelter に保存する．ソケットは各プロトコルを識別した上で，UDP/IP であれば送信元・送信先のポート番号・IP アドレスを保存する．TCP/IP であれば，加えてシーケンス番号と受信応答番号を保存する．イベントポールは，監視対象のファイルへのポインタがカーネル内のリストに保存されているため，この監視対象を調べて保存する．これらの資源は複数プロセスファイル資源はプロセス間で共有される可能性があるため，proc_shelter 構造体の外に定義する．proc_shelter 構造体内にはファイルディスクリプタテーブルの代わりに，これらのデータ構造を参照するための fd_shelter 構造体を定義する．

シグナルハンドラは sighand_struct 構造体のメンバである k_sigaction 構造体配列内にデータが格納されており，これにシグナル番号を追加したものを sig_shelter_t 構造体としている．ユーザが独自に設定していないシグナルハンドラは取得する必要が無い．シグナル受信時に呼び出されるコールバック関数が指定されている場合は，この関数のアドレスを保存する．この値は呼び出すべきプロセスの関数のアドレスを指しているため，OS のアップデートによって変化することはないため，直接この値を保存することができる．

4.2 プロセスコンテキストの復元

CPU のレジスタ値はビット幅がハードウェアによって決まっているため，アップデート後の OS で対応する変数に直接代入することができる．またプロセスの ID，プロセスやスレッドの親子関係を維持するために，指定した ID を用いて fork や clone を行う事ができるシステムコール my_fork と my_clone を追加した．ユーザプログラムの Inherit Manager はこのシステムコールを利用して予め親子関係を作る Seed を実行し，作成後に Inherit Manager はプロセスコンテキストの復元を開始するシグナルを送信する．

メモリの復元は Zero-copy Process Migration によって，ユーザ空間のメモリ内容をコピーすることなく行う．mmap_pgoff システムコールに対応する OS 内の関数の引数に開始仮想アドレスとメモリ領域のサイズ，読み書き実行の属性を指定してメモリ領域を確保する．Linux には不要なページ割り当てを避けるためにデマンドページングという機能が備わっており，この機能を利用して各ページテーブルエントリの復元はページフォルトが発生してから行うことで，復元処理に大きなオーバヘッドが発生することを防ぐ．

予め全てのファイル資源を File Reopener が復元しておき，そのデータ構造への参照をプロセスに与える．ファイルは保存していたファイルパスと展開パスを利用して，open システムコールに対応する OS カーネル内部の関数を呼ぶ．読み書き位置の調整は，必要があれば lseek システムコールに対応する OS カーネル内部の関数を呼ぶ．パイプは pipe システムコールに対応する関数を実行し，保存しておいたパイプバッファを write システムコールで書き込む．ソケットの再展開は，TCP/IP と UDP/IP，アクティブ側とパッシブ側で，それぞれ行う処理が異なる．アクティブ側とパッシブ側の違いは，そのソケットが最初に SYN を送ったのか，SYN を受け取ってソケットを作ったのかの違いである．UDP/IP のアクティブ側は，UDP/IP の通信を行うプログラム側がパケットの送信のたびに接続相手を指定することになっているため，接続情報を保持する必要が無い．UDP/IP のパッシブ側は，socket システムコールを実行した後，IP アドレスとポート番号を指定して bind システムコールを実行する必要がある．TCP/IP のアクティブ側は，socket，bind システムコールを実行した後，sock 構造体，inet_sock 構造体，inet_connection_sock 構造体，tcp_sock 構造体の通信情報を更新する必要がある．また ACK が返ってきていないバッファがあれば，通信情報を更新した後に send システムコールによって送信を行う．TCP/IP 固有の通信情報は，次に送信する送信パケットのシーケンス番号，次に受信する予定の受信パケットのシーケンス番号が挙げられる．復元処理でも同様に socket，bind，listen システムコールを行ったソケットを作成し，このソケットの複製を作成して通信情報を上書きする．

シグナルハンドラの再登録のためには，sigaction システムコールで利用される k_sigaction 構造体に，sig_shelter 構造体から保存情報をすべて代入し，シグナル番号とともに sigaction システムコールに対応する OS カーネル内部の関数を呼ぶ．ブロックされているシグナルについての情報は task_struct 構造体内の sigset_t 構造体のメンバである sig に値を代入する．

5. 実験

提案手法の評価を行うために，2 つの実験を行った．物理マシンの CPU は Intel Xeon E3-1270 V2（動作周波数 3.50GHz，4 コア，8 スレッド，キャッシュ 8MB），メモリは 16GB である．仮想マシンの構成は，仮想 CPU1 コア，仮想メモリ 4GB，スワップ領域を作成しない，とした．ゲスト OS に PCI パススルーで割り当てたデバイスは，ディスク装置に 2TB のハードディスク（TOSHIBA MD04ACA200, SATA, 7200RPM），NIC に 1Gbps の Intel E1G42ET(コントローラは SR-IOV 対応の 82576EB) を使用した．高速に復元するプロセスはゲスト OS 上で実行する．ゲスト OS の設定は，ASLR の機能は使用しない，シングルユーザモードで起動する，ページテーブルの操作は Direct Mode と Writable Page Table Mode を利用する，

表 2 プロセスとファイル等の退避にかかった時間 [ns]
Table 2 Process & File Save time[ns]

SIZE	THREAD	CHILD	SIGNAL	MEMORY	FILEs
1KB	22791	592	7431	5946844	10499
100KB	23039	609	7284	6304570	11042
10MB	22392	593	8780	5221311	14108
1GB	66806	2490	21103	33606300	12568

表 3 ファイル等の復元にかかった時間 [ns]
Table 3 File Restore time[ns]

SIZE	PID	FILE	PIPE	SOCKET	POLL
1KB	140	20300	309	240	313
100KB	68	15129	238	266	285
10MB	40	8511	72	118	3727
1GB	32	9133	164	174	167

表 4 プロセスの復元にかかった時間 [ns]
Table 4 Process Restore time[ns]

SIZE	MEMORY	FILEs	SIGNAL	THREAD	CHILD
1KB	274944	4853	370	6551	121
100KB	683132	10088	551	14284	189
10MB	873198	10511	614	14226	198
1GB	669140	13061	352	6988	252

PCI パススルーによって HDD と NIC を直接操作する，名前空間は使用しない，とした．

5.1 実験 1 各資源の退避と復元

作成したプログラムは，CPU，メモリ，ファイル，パイプ，ソケット，シグナルについての情報をすべて再現することができた．ここではそのプログラムの性能を評価するために，各資源の退避，復元にかかる時間を計測することで，どの資源の維持がボトルネックとなっているのか調査する．

メモリの容量を変化させて計測を行った．計測には Linux カーネル内で ns 単位で時間計測を行う getnstimeofday() を利用し，退避，復元処理の各フェーズごとに所要時間を計測した．引き継いだプロセスは，指定されたバイト数のヒープを確保してスリープするだけの，計測用に作成したプログラムである．malloc() 関数を用いて確保するヒープサイズは，1KB，100KB，10MB，1GB と変化させた．ただし実際のに確保されるヒープメモリのサイズは異なる場合がある．また，退避・復元を行うのはヒープ領域に加え，プログラムのテキスト領域，静的領域，スタック領域と，その他のライブラリの領域を合わせた領域である．

プロセスとファイル資源の退避にかかった時間を**表 2** に示す．THREAD は CPU の情報を取得する時間，CHILD は子プロセスと子スレッドにシグナルを送り取得完了を待つ時間，SIGNAL はシグナルハンドラを取得する時間，MEMORY はメモリレイアウトとページテーブルを取得する時間，FILEs は通常ファイル，パイプ，ソケット，イベントポールを取得する時間である．

ファイル資源の復元にかかった時間を**表 3** に示す．PID はプロセス ID を予約する時間，FILE は通常ファイルを再展開する時間，PIPE はパイプを復元する時間，SOCKET はソケットを復元する時間，POLL はイベントポールを復元する時間である．メモリ以外の資源は変更していないため，これらは本来同じ値となる．

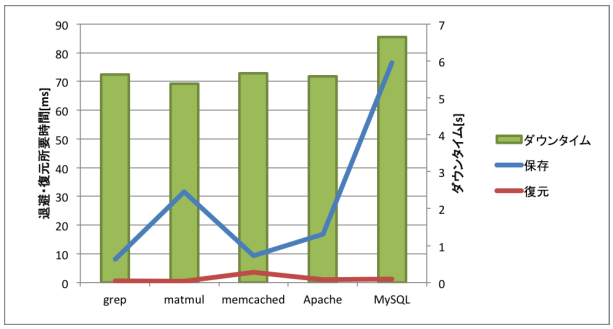

図 4 各プログラムのダウンタイムと，退避・復元にかかる時間
Fig. 4 Downtime and Duration for Save/Restore process

プロセスの復元にかかった時間を**表 4** に示す．MEMORY はメモリレイアウトを復元する時間（ページテーブルの復元は含まない），FILEs はファイルディスクリプタに復元したファイル資源を割り当てる時間，SIGNAL はシグナルハンドラを再登録する時間，THREAD は CPU の情報を復元する時間，CHILD は子プロセス，子スレッドに復元開始を通知する時間である．

5.2 実験 2 リアルアプリケーションへの適用

実際に使用されているアプリケーションに対して，提案手法を用いてダウンタイムを短縮した OS の再起動を行った．

実験に使用したプログラムは，ファイルから文字列の検索を行う grep，行列演算を行う matmul，メモリキャッシュシステムの memcached，HTTP サーバの Apache，データベース管理システムの MySQL である．実験中の OS 再起動は全て，提案手法を用いて高速に行っている．grep は，Linux-4.3.0 の全ソースコードから「linux」の文字列を探索し，検索開始 1 秒後に停止，OS を再起動し，復元を行った．matmul は，matmul_v3.py を引数 500 で実行し，実行開始 1 秒後に停止，OS を再起動し，Python インタプリタの復元を行った．memcached は 3 スレッドで動作し，リモート環境から memslap を同時接続数 1 で実行し，実行開始 5 秒後に停止，OS を再起動し，復元を行った．Apache は 2 スレッドで動作し，リモート環境から ApacheBench を同時接続数 1 で実行し，実行開始 5 秒後に停止，OS を再起動し，復元を行った．MySQL は 16 スレッドで動作し，リモート環境から SysBench の OLTP を同時接続数 10 で実行し，実行開始 5 秒後に停止，OS を再起動し，復元を行った．

各プログラムの退避・復元の所要時間について，結果を**図 4** に示す．退避処理と復元処理は左側の軸，退避処理の開始から復元処理の終了までのダウンタイムは右側の軸を使用する．

各プログラムの退避にかかる時間は 8.2ms（grep）から 77ms（MySQL）と差があった．いずれのプログラムも，プロセスの退避にかかる所要時間の 97%以上はメモリの保存にかかる時間で，このうちページテーブルエントリの取得がほとんどの割合を占める．また，MySQL はスレッド数が多かったため，各スレッドの停止にも 1.6ms（退避処理の約 2%程度）の時間を使用した．この結果から，退避にかかる処理時間の支配項はメモリの保存であるといえる．

各プログラムの復元にかかる時間は 0.47ms（matmul）から 3.5ms（memcached）と差が小さかった．これは，プロセスの復元にかかる時間と，I/O の復元にかかる時間との和である．プロセスの復元にかかる処理のうち，メモリの復元にかかる時間が 90%以上を占めるという傾向はメモリの保存と変わらないが，占有割合は低下している．これは，メモリ内容の復元は実際にページフォルトが発生するまで先送りにされているため，この段階ではページテーブルをたどって復元していないためである．matmul と memcached はほとんどファイルを展開しないが，grep，MySQL，Apache はファイルやパイプ，ソケットの展開数が多かった．そのため I/O の復元にかかる時間は，前者は復元処理全体の約 3%，後者は復元処理全体の約 95%と，傾向に大きな違いが見られた．

各プログラムの停止から再開までの時間，すなわちプログラムのダウンタイムは，いずれも 5.4sec（matmul）から 6.6sec（MySQL）とほぼ変わらず，その差の発生原因は OS の初期化にかかる時間であった．ファイルの復元数が多いと，ディスク I/O を多く発行する OS の初期化と競合し，OS の初期化もプログラムの再開も遅くなる傾向が見られた．

削減できたダウンタイムについて評価するために，通常のリブートと，提案手法による高速なリブートとの比較を行った．memcached を起動した状態で，提案手法を用いてプロセスの退避・復元を行いつつ，OS の再起動は通常通り行った結果，43.4sec かかった．各プログラムのダウンタイムが約 5.8sec 程度であるので，約 87%の OS 再起動中のプログラムのダウンタイムを削減することができた．

6. 関連研究

ダイナミックパッチング [2], [3], [4], [6], [9], [11], [14] では，アップデートの適用をプログラムの再起動無しに行う．DynAMOS[11] と Ksplice[9] では，アップデート内容は OS カーネルモジュールとして用意し，アップデートのタイミングで適用対象の関数への CALL 命令を OS カーネルモジュールへのジャンプコードに書き換えることで，処理を切り替えている．DynAMOS ではアップデートパッチの分類を行い，適切なアップデート適用タイミングについて議論しており，多くのものが自動決定できることを示している．この決定には，アップデート適用対象の関数・データ構造が現在参照されていない，参照を行うプログラムがスタック上にない，といったことが関与する．DynAMOS は通常のアップデートパッチから専用のコードを書く必要がある．Ksplice ではアップデートはパッチからコンパイラによって生成した機械語レベルで適用しており，多くの処理を自動化している．メモリ上でアップデート箇所のみを更新するため，アップデートに伴うダウンタイムはマイクロ秒単位である．しかしながら，ダイナミックパッチングはアップデート内容によって利用できないものがある．データ構造への変更は自動化できておらず，適切な初期化方法がない場合もある．また，OS のソースコードや実行イメージの差分が取得できなければならず，ソースコードが公開されていない商用ソフトウェアや，そのようなパッチが存在しないメジャーアップデートの場合には利用できない．

プロセスマイグレーション [1], [16], [17] は，プロセスを別のマシン上で動作している OS カーネルに移動する．再起動対象となっている OS 上で稼働中のプロセスを別の OS 上に移動し，再起動を行うことでプロセスの実行を継続したまま OS のアップデートを行うことができる．プロセスのライブマイグレーションができればダウンタイムはほとんど無い．Zap[17] や AutoPod[16] は，プロセスと OS カーネルのレイヤの間に，プリセス ID やファイルディスクリプタ番号などを仮想化するレイヤを挿入する．OS カーネル内での ID のコンフリクトを避けつつ，プロセスから見える ID を変えない，といったことを可能である．Zap では新しい資源として移送先のマシンが必要となる．またマイグレーション中は送信元と送信先との間で大量のネットワークトラフィックが発生し，両マシン上で動作する他のプログラムや，この経路を利用するネットワークプログラムが競合の影響を受ける．Microvisor[5] は，ハイパーバイザを用いて同一物理マシン内でプロセスマイグレーションを行う．しかしデバイスは仮想化されいなければならず，単一マシン内では処理が完結していないという点では Zap と同じである．

OS の再起動によるサービスのダウンタイムを減らすために，高速に再起動を行うことを目的とした研究 [8], [10], [12], [19] も存在する．ShadowReboot[8] では，ハイパーバイザを用いて別 VM 上でアップデートパッチを適用した OS カーネルを起動し，OS のメモリイメージをアップデート後のものに切り替えることで高速にアップデート後の OS に移行する．Phase-based Reboot[10] 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．いずれの手法も，プロセスの状態は失われる．Seamless Kernel Updates[12] は BIOS を介したハードウェアの初期化を省略する Kexec という手法を利用しつつ，プロセスの状態をメモリ上に残しておくことでプロセスの状態を再現

する．この手法では再起動にかかる時間を短縮しているものの，10 秒以上のダウンタイムが発生する．

オフラインパッチング [13], [18], [20] では，停止しているプログラムにアップデートパッチを適用する．アップデートが適用されることなく放置されやすい，停止しているマシン上のプログラムにアップデートを適用することができる．Nüwa[20] は仮想マシンの貸し出しを行う IaaS クラウド環境を想定し，停止中の VM イメージに対してアップデートを適用する手法を提案している．VM イメージを外部の VM からマウントし，アップデートプログラムを安全に適用する．また複数の VM イメージに対してはバッチ処理を行うことで，高速にアップデートを適用する手法も導入している．オフラインパッチングではアップデートを行う対象が停止していることを想定しており，本研究で想定している常時稼働が期待されるようなマシンに対しては利用できない．

7. 結論

Essential Context は既存のインタフェースに基いた保存形式とすることで，OS のアップデート内容に依存しない方法でプロセスの引き継ぎを行う手法を提案した．プロセスのメモリ領域の複製を行わない Zero-copy Process Migration，アップデートが適用された OS カーネルを並列して起動する Background Boot，デバイスの引き継ぎをプロセスの引き継ぎと平行して行う Attach-pipelining によって，アップデートに伴うダウンタイムを削減する手法を提案した．通常のリブートでは memcached の停止から再起動を経て再開するまでのダウンタイム 43.4 秒を，提案手法によって新しく起動した OS 上で高速に復元し 5.8 秒に短縮した．

参考文献

[1] Ammon Barak and Richard Wheeler. MOSIX: An Integrated Multiprocessor UNIX. In *Proceedings of the USENIX Winter 1989 Technical Conference*, pages 101–112, 1989.

[2] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications(OOPSLA '12)*, pages 249–264, 2012.

[3] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and Automatic Live Update for Operating Systems. In *Proceedings of the 18th international conference on Architectural support for programming languages and operating systems (ASPLOS '13)*, pages 279–292, 2013.

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

[5] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable Virtual Machines Enabling General, Single-Node, Online Maintenance. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS '04)*, pages 211–223, 2004.

[6] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments (VEE '06)*, pages 35–44, 2006.

[7] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 ACM SIGCOMM Conference*, pages 193–204, 2004.

[8] Hiroshi Yamada and Kenji Kono. Traveling Forward in Time to Newer Operating Systems using ShadowReboot. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments (VEE '13)*, pages 121–130, 2013.

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

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

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

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

[13] Microsoft. Virtual Machine Servicing Tool (VMST) 2012. https://technet.microsoft.com/en-us/library/jj149757.aspx

[14] 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 *Proceedings of the 1st ACM European Conference on Computer Systems (EuroSys '06)*, pages 133–145, 2006.

[15] 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, 2003.

[16] Shaya Potter and Jason Nieh. Reducing Downtime Due to System Maintenance and Upgrades. In *Proceedings of the 19th Large Installation System Administration Conference (LISA '05)*, pages 47–62, 2005.

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

[18] VMware. VMware vSphere Update Manager: Compli-

ance & Configuration Management.
https://www.vmware.com/products/vsphere/features/update-manager

[19] Werner Almesberger. kboot—A Boot Loader Based on Kexec. In *Proceedings of the Linux Symposium*, Volume One, pages 27–38, 2006.

[20] Wu Zhou, Peng Ning, Xiaolan Zhang, Glenn Ammons, Ruowen Wang, and Vasanth Bala. Always Up-to-date - Scalable Offline Patching of VM Images in a Compute Cloud. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*, pages 377–386, 2010.

© 2016 Information Processing Society of Japan