カーネルレベルにおける ECC-uncorrectable メモリエラー回復機構

井口 卓海[1]　山田 浩史[1]

概要：メモリにおける故障（メモリエラー）の中でも，ECC モジュールによって検知できても訂正できない *ECC-uncorrectable* **メモリエラー** と呼ばれるエラーは，ハードウェアやアプリケーションを管理するソフトウェアであるオペレーティングシステム（OS）のカーネルに深刻な被害を及ぼす．というのも，ECC-uncorrectable メモリエラーが OS カーネル内データを破壊して OS カーネルが停止してしまうと，その上で稼働していた全アプリケーションのスケジューリングが停止することでシステムがダウンしてしまう．この ECC-uncorrectable メモリエラーに対する最も基本的な対策として，OS カーネルそのものを再起動する方法が挙げられる．この再起動をベースとした方式は，メモリ上のデータがたとえダメージを負っていなくても全てのアプリケーションが再起動を強いられるため，インメモリデータベース (IMDB) のようなメモリを大量に使用するアプリケーションでは実行状態の回復に非常に時間がかかる．本論文では，カーネル領域における ECC-uncorrectable メモリエラーに対してリカバリによる OS カーネルの生存を指向する *Ev6* を提案する．Ev6 は ECC-uncorrectable メモリエラーが発生したメモリ上の内容をメモリオブジェクトの粒度で分離・再構築を施し，OS カーネルを可能な限り存続させる．この Ev6 のプロトタイプを xv6 上に実装し，フォールトインジェクションをはじめとする様々な実験を行った．実験の結果，Ev6 は低い回復時間とメモリスペースオーバヘッドで ECC-uncorrectable メモリエラーを模したフォールト下でも OS カーネルを継続動作できることを確認した．

キーワード：オペレーティングシステム, ECC-uncorrectable メモリエラー, リカバリ

1. はじめに

メモリにおける故障であるメモリエラーの中でも，Error Correction Code（ECC）[1] モジュールによって検知可能でも訂正不可能な *ECC-uncorrectable* **メモリエラー**はソフトウェアの信頼性に対して甚大な影響を及ぼす．メモリエラーは，宇宙線の衝突のような電磁気的な要因やメモリセルの物理的な損傷のようなハードウェア的な要因によって引き起こされる．メモリエラーの発生状況に関する先行研究によると，1 年間に平均してデータセンタ内のサーバマシンの 32% が訂正可能なメモリエラーを経験し，1.3% 以上のマシンで訂正不可能なエラーが発生している [2]．また，別の調査によると DRAM の微細化によってメモリエラーの発生率が悪化しているという調査結果や [3, 4]，近年開発されている 3D Xpoint 技術は DRAM に比べてメモリセルの損傷が起きやすいという報告もある [5, 6]．SEC-DED（Single Error-Correction Double Error Detection）[7] をはじめとする一般的に普及している ECC モジュールはメモリエラーの検出だけでなく訂正を行うことも可能であるが，SEC-DED における同一ワードにおける複数ビットの反転のように，訂正能力を超えた規模のメモリエラーは訂正できずに検出するに留まる．いくつかの先行研究によれば，ECC モジュールを付与していても検知可能でも訂正不可能なエラーは頻繁に起こることが報告されている [2, 8–10]．

ハードウェアやアプリケーションを制御する役割を担うソフトウェアであるオペレーティングシステム（OS）カーネル内部で発生した ECC-uncorrectable メモリエラーは，システム全体に対して特に深刻な被害をもたらす．OS カーネルに比べてメモリ使用量が多いアプリケーションは，メモリエラーに遭遇する確率が OS カーネルよりも多い一方，OS カーネルでの障害は OS カーネル上で稼働する全アプリケーションに致命的な影響が及ぶことから，アプリケーションにおける障害よりも深刻度が大きい．もし OS カーネル領域において ECC-uncorrectable メモリエラーが検知されると，OS カーネルは停止してしまい，それに伴いスケジューリングが停止することで OS カーネル上の全

[1] 東京農工大学
Tokyo University of Agriculture and Technology

ⓒ 2022 Information Processing Society of Japan

てのアプリケーションも道連れにクラッシュしてしまう．効率的なリカバリメカニズムを採用した既存手法やメモリエラーに対するデータ保護を提供するものも存在するもの の [11–14]，これらはアプリケーションレベルに留まるため，OS カーネル領域におけるメモリエラーには通常のシステムと同様にクラッシュしてしまう．

カーネル領域において ECC-uncorrectable メモリエラーが検知されると，OS カーネルは処理を停止してダウンし，ユーザに再起動を促すように設計されている．このため，もしユーザのアプリケーションに割り当てられたメモリ領域がたとえ健全であったとしても，システムダウンと再起動によってユーザアプリケーションは処理の中断を余儀なくされ，それまで処理していたメモリ上のデータも揮発してしまう．これにより，インメモリキーバリューストア（KVS）や大規模な機械学習，商用のクラウドサービスプラットフォームなどの数千 GB から数 TB スケールのメモリを使用するアプリケーションでは，マシンの再稼働までに長い時間を要することになる．このような点から，システム全体の信頼性と可用性を向上させるためには，OS カーネルそのものにも ECC-uncorrectable メモリエラーへの耐性を付与する必要があるといえる．

本論文では，カーネルのメモリ領域において ECC-uncorrectable メモリエラーが発生しても OS カーネルを可能な限り生存させる *Ev6* を提案する．ECC-uncorrectable メモリエラーが発生しても，他の大部分のメモリは健全な状態のままであるという点に着目し，ECC-uncorrectable メモリエラーの被害を受けたメモリオブジェクトを破棄した上で他の健全なメモリオブジェクトを用いて再構築することによって，システムを可能な限り継続稼働させる．本研究は ECC-uncorrectable メモリエラーへの耐性を OS カーネルに付与する試みの最初の一歩として位置づけており，オープンソースの OS である *xv6* [15] をケーススタディとしてメモリオブジェクトのセマンティクスや依存関係を調査し，リカバリハンドラを実装したプロトタイプを作成した．

本論文における貢献を以下に示す．

- カーネルアドレス空間における ECC-uncorrectable メモリエラーへの耐性を有する OS である *Ev6* を提案した．従来の ECC-uncorrectable メモリエラーへのアプローチとは異なり，Ev6 は回復時の OS カーネルの再起動を可能な限り回避することによるダウンタイムの削減，低メモリスペースオーバヘッドによるリカバリ機構の実現，及びハードウェアの変更を必要としないという 3 つの特徴を持つ（3 章）．
- 提案手法を実現するため，*per-object recovery handler*，*M-list* 及び *NMI shepherd* の 3 つのメカニズムを導入した．per-object recovery handler は ECC-uncorrectable メモリエラーにより破損したメモリオブジェクトのセマンティクスに合わせた回復処理を実施する．また，M-list は ECC-uncorrectable メモリエラーが検知された箇所のアドレスから対応するメモリオブジェクトを特定する役割を担い，NMI shepherd はマルチコア環境における複数の連続した ECC-uncorrectable メモリエラーの処理を補助する役割を担う（4 章）．
- xv6 に対して Ev6 のプロトタイプを実装し，フォールトインジェクションをはじめとする様々な実験を行った．実験の結果，Ev6 は ECC-uncorrectable メモリエラーに対する回復能力を低実行時オーバヘッドと低メモリスペースオーバヘッドで実現していることを確認した（5 章）．

2. 背景

2.1 メモリエラー

メモリエラーは，宇宙線など電磁気的要因や物理的破損・劣化などのハードウェア的要因によって，メモリに保管されているデータを正しく読み取ることができない状態を指す．メモリエラーは発生時に即座に発覚するわけではなく，メモリエラーに侵された箇所にアクセスを試みた時点で初めて誤ったデータの読み出しやシステムクラッシュといった影響が発生する．メモリエラーの発生率を調べた先行研究によると，微細化が進んだ DRAM ではメモリエラー発生率が悪化しているという調査結果や [3,4]，近年開発されている 3D Xpoint 技術は DRAM に比べ，メモリセルの損傷が起きやすいという調査結果も存在する [5,6]．

メモリエラーに対するシステムの信頼性を高めるために，メモリエラーの検知及び訂正を行う Error Correction Code（ECC）[1] モジュールをメモリに付与させる方法がある．ECC 機能は，保存するデータに加えて符号を記憶しておき，データの読み書き時に記録しておいた符号と保存データから計算される符号を比較することによって，メモリエラーなどによるデータの誤りを検出・訂正することができる機能である．例えば，代表的な ECC アルゴリズムである SEC-DED（Single Error-Correction Double Error-Detection）[7] は，同一ワード内の 1 ビットの誤りを訂正することができ，より高い誤り検知・訂正能力を持つ Chipkill [16] も存在する．ECC 機能が持つ誤り訂正・検知能力は採用している ECC アルゴリズムによって異なり，また ECC 機能によって発生するメモリスペースオーバヘッドも異なる．

しかし，たとえ ECC モジュールを装備したとしても，全てのメモリエラーに対処できるわけではない．SEC-DED における同一ワードにおける複数ビットの反転のように，訂正能力を超えた規模のメモリエラーは訂正できずに検出することしかできない．このように，ECC モジュールでも訂正できずに検知されるに留まるメモリエラーは特に *ECC-uncorrectable* **メモリエラー** と呼ばれ，ECC で訂正可

能なメモリエラーである *ECC-correctable error* と区別される．ECC-uncorrectable メモリエラーが検出されると，多くの場合 OS カーネルは該当するプロセスを強制終了したり，システムをクラッシュさせることで対処する．このため，OS カーネル上で稼働していたアプリケーションは強制終了を余儀なくされる．特に，Memcached [17] をはじめとするインメモリキーバリューストア（KVS），機械学習や大規模科学計算をはじめとするハイパフォーマンスコンピューティング，GoogleCE [18] や Amazon EC2 [19] をはじめとする商用のクラウド環境やウェブサービスはメモリ資源を大量に消費するため，再起動したマシンを稼働状態に復帰させるまでに非常に時間がかかる．例えば，Facebook の報告によれば，サーバ全体の 2% の再起動に 12 時間を要し，その間ユーザは依頼したクエリの一部のみしか結果を得ることができないという事態に陥ってしまう [20]．

このような被害をもたらすメモリエラーは頻繁に発生することが先行研究により報告されている．実環境におけるメモリエラーの発生状況の調査によると，1 年間に平均してデータセンタ内のマシンの 32% が ECC-correctable error を経験し，1.3% 以上のマシンで ECC-uncorrectable メモリエラーが発生していると報告している [2]．同様に，ECC-uncorrectable メモリエラーは DRAM や SSD のような記録媒体で頻繁に起こることが報告されている [2, 8–10]．以上の点から，OS カーネル自体に ECC-uncorrectable メモリエラーへの耐性を付与することの必要性は大きいといえる．

2.2 フォールトモデル

本論文では，既存のソフトウェアやハードウェア機構によって検知可能であるが訂正不可能なメモリエラーのうち [1, 7, 12, 21]，カーネルアドレス空間内で発生したものを対象とする．そのため，ECC-undetectable error やメモリエラーによって誤ったデータが読み出される Silent Data Corruption（SDC）は対象外となる．また，本論文ではメモリエラーが検知された後にメモリエラーがシステムに及ぼす影響を緩和する方法を調査する．つまり，ECC ハードウェアモジュールによって訂正できなかったメモリエラーが検出されると，OS カーネルは Non-maskable Interrupt（NMI）によってその発生が通知される．この時点で本論文が提案する手法にって OS カーネル内のオブジェクトに対するリカバリ処理が行われる．

2.3 関連研究

システムの信頼性や可用性を高めるため，現在に至るまで様々なソフトウェアベースの先行研究が実施されてきた．

primary-backup [22] や state machine replication [23, 24] のように，データの複製によって障害への耐性を向上させるアプローチが存在する．このアプローチでは，複数のマシン間でアプリケーションのデータを複製しておくことで，もしあるマシンが障害でクラッシュしたとしてもサービスを提供し続けることが可能である．しかし，レプリケーションは複製を取ることによってメモリ資源を 2 倍消費することになるため，メモリを大量に要求するアプリケーションでは信頼性を高めるために負うコストが大きくなるという欠点がある．また，Intel の Address Range Partial Memory Mirroring [25] は単一のマシン内で一定範囲のメモリ領域の内容を複製することで信頼性を向上することが可能である．しかし，ミラーリングによって複製可能な領域のサイズには限界があることから，メモリを大量に消費するアプリケーションを動かしている OS カーネルを複製することは困難である．

複製に似た技術として，プロセスの状態をスナップショットとして保存しておくチェックポインティングメカニズムも存在する [26–30]．これにより，もし OS カーネルにおける障害が起きたとしても，ストレージに保存しておいたプロセスの実行状態を復元し，再起動後に実行を再開することが可能になる．しかし，これらのスナップショットは主に低速なストレージ上に保管されることから，チェックポインティングを適用しても稼働状態への復帰には時間がかかってしまうことになる．他にも，このスナップショットをメモリ上に設置することで，プロセスだけではなくシステムの再起動やカーネルの更新の時間を短縮する手法も提案されている [31–33]．また，ShadowReboot [34] は子仮想マシン（VM）をフォークし，親の VM を再起動して新しい OS カーネルをバックグラウンドで適用することが可能である．しかし，これらの手法は OS カーネルの更新を目的とした手法であるため，OS カーネルの障害に適用するのは難しい．

マイクロカーネルをはじめとして OS カーネルの構造を抜本的に改良する手法も存在する．マイクロカーネルをベースとしたロールバックを行う先行研究 [35, 36] は，カーネルの構成要素に障害が起きても回復を行うことが可能になる．マイクロカーネルをベースとしていることにより，サーバとして切り離されたカーネルの要素を再起動することができる．しかし，プロセス間通信のようなカーネル領域内で稼働する機能内における ECC-uncorrectable メモリエラーに対してはシステムクラッシュが発生してしまうため，上で稼働しているソフトウェアも再起動を迫られるという点では通常のモノリシックカーネルと同様である．また，マイクロカーネルをベースとし，Rust [37] のような型安全な言語を用いて OS カーネルの信頼性と可用性を向上させるアプローチ [38–40] も行われている．これらの研究では，型安全な言語を採用することにより，メモリリークやバッファオーバフローのようなバグを除外できるほか，マイクロカーネルをベースとしてモジュール性を上げることで信頼性を向上させている．しかし，これらのアプロー

ⓒ 2022 Information Processing Society of Japan

チは現在普及しているモノリシックカーネルベースの OS カーネルから大きな変更を必要とするという弱点がある．

OS カーネルを再起動させる reboot-based アプローチをベースにした先行研究も数多く実施されてきた．Otherworld [41] は，ユーザ空間で稼働するプロセスんのメモリ内容を管理し，プロセス制御ブロック（PCB）を再生成することで OS カーネルのみ再起動させる手法である．しかし，プロセスの復元に必要なメモリオブジェクトが ECC-uncorrectable メモリエラーによって破損していた場合，OS カーネルを再起動できない．Phase-based Reboot [42] は，OS カーネルやサービスプロセスのブート時に段階ごとにスナップショットを取得しておき，次のブート時に再利用することで reboot-based recovery によるダウンタイムを短縮する手法である．また，Kexec [43] はハードウェアリセットを必要とせずに別の Linux カーネルを立ち上げることができる手法である．しかし，再起動によるダウンタイムを削減しても，OS カーネル上で稼働するアプリケーションの復帰に長い時間を要してしまうことは通常のシステムクラッシュによる再起動と変わりない．

OS カーネルとリカバリ機構の組み合わせも広く研究されてきた．Nooks [44] と Shadow Driver [45] はデバイスの障害を検知すると，アプリケーションからは透過的にそのドライバを再起動させることが可能である．また，Membrane [46] は定期的にチェックポインティングを行い，ファイルシステムのスナップショットをとっておくことで，ファイルシステムに障害が起きた際には直近のスナップショットまでロールバックすることでリカバリが可能となる．しかし，これらの手法はハードエラーを対象としておらず，このエラーが発生した場合にはリカバリが失敗してしまう．加えて，OS カーネル全体を回復対象としていないため，対象外の構成要素が破損した際にもリカバリはできない．他にもカーネルレベルのロールバックを実現した手法は存在するが [47–49]，これらもまた一時的なメモリエラーが対象である．このため，メモリエラーがソフトエラーではなくハードエラーであった場合，ロールバックを行っても再度同じエラー箇所にアクセスしてしまうため，ECC-uncorrectable メモリエラーに再度遭遇することになるという弱点を持つ．

3. 提案

本論文では，ECC-uncorrectable メモリエラーの発生下においても OS カーネルを可能な限り生存させる *Ev6* を提案する．Ev6 のデザインゴールを以下に示す．

- **リカバリによる稼働中の全ソフトウェアの再起動を回避する：**リカバリ時に再起動を強いる他の手法とは異なり，Ev6 は OS カーネルやその上で稼働するアプリケーションも含め，可能な限りリカバリによる性能低下を削減し，実行を継続させる．
- **大きなメモリスペースオーバヘッドをもたらさない：**様々な状況に対応するため，Ev6 は単一のマシン上で稼働し，リカバリ機構によるメモリ消費量を可能な限り小さくする．
- **OS 自体の機能を損なわない：**現在普及している OS カーネルは計算資源管理のために洗練されたソフトウェアメカニズムを有しており，Ev6 はそれらの OS の機能に干渉しない．
- **ハードウェアの変更を必要としない：**提案手法はソフトウェアで完結するアプローチを取り，Ev6 の下に位置するハードウェアに対して変更を加える必要がない．

ECC-uncorrectable メモリエラー に対し，Ev6 は OS を再起動するのではなく，メモリ上に存在するアプリケーションや Ev6 自身の実行状態を失わないよう，内部のメモリオブジェクトを再構築する．カーネル領域において ECC-uncorrectable メモリエラー が発生した場合，Ev6 はエラーが発生したメモリアドレス上に存在していたメモリオブジェクトを特定し，その破損したオブジェクトを切り離した上で他の健全なオブジェクトから再構築を行う．これらの処理をメモリエラーの影響を受けていない他のユーザプロセスから透過的に行うことにより，ECC-uncorrectable メモリエラー を処理した後もそれらのプロセスは処理を継続することが可能になる．

Ev6 はメモリエラーによって損傷したメモリオブジェクトの粒度で損傷箇所を破棄し，オブジェクトのタイプに合わせた回復処理を施す．カーネル領域上における ECC-uncorrectable メモリエラー の発生を Ev6 が NMI により検知すると，Ev6 はどの損傷したメモリオブジェクトが被害を受けたのかを特定してリカバリハンドラを呼び出し，OS カーネル生存のための回復処理を行う．この時，ECC-uncorrectable メモリエラー の被害を受けたメモリ領域上に置かれていたメモリオブジェクトは使用不可と判断し，オブジェクトの粒度で破棄する．これにより，メモリエラーが発生したメモリページ上に複数のメモリオブジェクトが置かれていた場合でも，健全なオブジェクトの移動は不要となる．

Ev6 を設計する上で，以下に示す 3 つのデザインチャレンジが生じる．

(1) 破損したメモリオブジェクトをどうやって矛盾なく回復するのか？
(2) 破損したメモリオブジェクトのタイプをどうやって特定するのか？
(3) マルチコア環境における複数の ECC-uncorrectable メモリエラー をどうやって処理するのか？

これらを踏まえ，OS カーネルに ECC-uncorrectable メモリエラー への耐性を付与する取り組みの最初の一歩として，MIT が開発した小規模な OS である xv6 [15] をケーススタディとして Ev6 のプロトタイプを設計した．以降の文章で

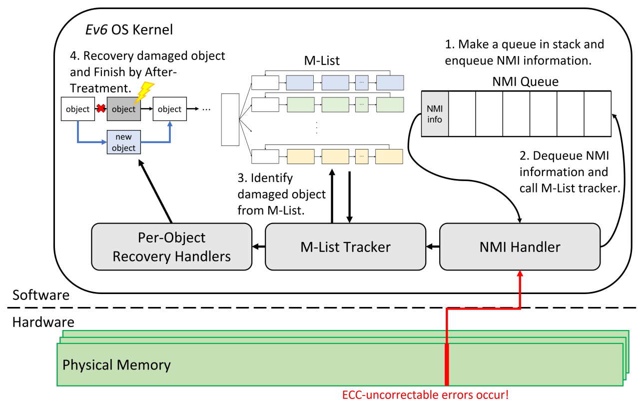

図 1: *Ev6* の全体像

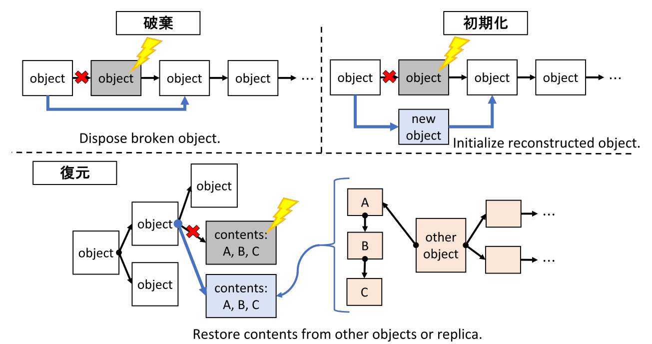

図 2: *Internal-Surgery* における 3 つの方針

は，Ev6 のプロトタイプに関する設計の詳細について説明した後，実施した実験について述べる．

4. 設計

Ev6 の全体像を図 1 に示す．本提案手法において鍵となるソフトウェアメカニズムは，*per-object recovery handler*，*M-list*，*NMI shepherd* の 3 つである．2.2 項において述べたように，ECC-uncorrectable メモリエラーが ECC モジュールにおいて検知されると，Ev6 は ECC モジュールから NMI によってその発生箇所のアドレスを受け取り，NMI ハンドラを呼び出す．Ev6 の NMI ハンドラは，連続した複数の ECC-uncorrectable メモリエラーを処理するため，受け取った物理アドレスをエンキューしておく．そして，キューから取り出した物理アドレスを用いて M-list を引き，破損した箇所の物理アドレスとメモリオブジェクトの対応関係を特定する．最後に，特定したオブジェクトのタイプに対応するリカバリハンドラを呼び出すことで，メモリエラーによって破損したオブジェクトを再構築する．本章の以降の文章では，本論文でケーススタディとした xv6 について簡単に説明した後，これらの 3 つのソフトウェアメカニズムについてそれぞれ詳細に説明する．

4.1 xv6 の概要

Xv6 は UNIX ライクな OS であり，これを構成する機能はファイルシステム，メモリマネージャ，コンソール，ロック，そしてプロセスマネージャの 5 つに大別され，それぞれの機能を実現するためのメモリオブジェクトがいくつも存在する．Xv6 のファイルシステムはジャーナリング方式を採用しており，open() や close()，read() や /write() といった UNIX ベースのシステムコールに対応している．ファイルシステムに含まれるオブジェクトは，`file` 構造体や `log` 構造体をはじめとする計 11 種類が存在する．メモリマネージャは空きメモリページの管理を司り，ページ管理にフリーリスト方式を採用している．メモリマネージャに含まれるオブジェクトは，`kmem` 構造体，`run` 構造体やページテーブルの計 3 種類である．コンソールは，文字通りコンソールにおける I/O を管理するもので，`devsw` 構造体，`cons` 構造体，`pr` 構造体の 3 種類が該当する．プロセスマネージャは xv6 で稼働するプロセスの管理やトラップのハンドリングを担当しており，プロセスや CPU のメタデータを保持したり，トラップ時などの汎用レジスタの中身を保存するオブジェクトが存在する．最後のロックは，前述の機能における排他制御を担当しており，`spinlock` 構造体と `sleeplock` 構造体がサポートする 2 種類のロックが用意されている．

4.2 破損したメモリオブジェクトのリカバリ

Xv6 のメモリオブジェクトはそれぞれ独自のセマンティクスや依存関係を有していることから，汎用リカバリハンドラのみで対処することは容易ではない．このため，個々のオブジェクトの再構築を行うために，Ev6 では対象のオブジェクトごとに 1 つのリカバリハンドラを用意する．

リカバリハンドラの詳細な処理はオブジェクト間で異なるものの，回復処理の段階は共通しており，*Internal-Surgery* と *After-Treatment* という 2 段階からなる．Internal-Surgery は，破損したメモリオブジェクトを切り離し，依存関係を持つ他の健全なオブジェクトから再構築する段階である．この段階では更にオブジェクトのセマンティクスによって，図 2 に示すように再構築の方針が以下に示す 3 つに分岐する．というのも，単純に破損箇所を切り離して破棄するだけでは，ジャーナリングを司る `log` 構造体のようにシステムの機能に不可欠なオブジェクトをリカバリ後に使用できず，システムをクラッシュさせてしまうためである．

破棄：この方針を採用するメモリオブジェクトは，ECC-uncorrectable メモリエラーが発生した場合には単純に OS カーネルが使用できないように破棄される．この方針は，フリーリストにおける空きページのメタデータを管理する `run` 構造体のような，メモリオブジェクトのうち単に取り除いても OS カーネルを含むシステムの継続実行に支障がないオブジェクトに適用される．

初期化：この方針では，メモリエラーに侵されたメモリオブジェクトを切り離した上で代わりの領域を用意して初

期化を行う．この方針は，バッファキャッシュを司る buf のように，オブジェクトが保持していた内容自体はシステムの継続動作に影響しないが，単なる破棄ではシステムを生存させられない場合に適用される．

復元：この方針では，メモリエラーに侵されたメモリオブジェクトを切り離して代わりの領域を用意するだけではなく，他のオブジェクトやレプリカからオブジェクトが保持していたデータも回復する．この方針は，ジャーナリングを司る log 構造体のように，オブジェクトが保持していた情報もシステム生存に必要な場合に適用する．

Internal-Surgery の後に位置する After-Treatment ではリカバリハンドラの処理をどうやって終えるのかを決定する．OS カーネルを生存させるためには，Internal-Surgery を終えたリカバリハンドラから OS カーネルの処理に復帰させる必要がある．しかし，メモリエラーのリカバリは通常の割り込みとは異なり，割り込みが発生した位置にそのまま返ることができない．なぜなら，メモリエラーに起因する割り込み（NMI）が発生した箇所に返ることは，再度破損したオブジェクトに触ることによって再度 NMI を引き起こしてしまうためである．これを防ぐため，Ev6 では以下に示す異なる終了方法を使い分ける．

Syscall Fail：一連の回復処理をシステムコールが失敗したとして終了する方法である．Internal-Surgery が終わると，リカバリハンドラはカーネルのコンテキストを破棄し，システムコールのエラーとして直接ユーザのコンテキストに復帰する．

Process Kill：回復処理を終えた後，メモリエラーが検知されたプロセス自体を強制終了する方法である．この方針が適用されるのは，ECC-uncorrectable メモリエラーが起きたカーネルのコンテキストに一貫性を保ったまま復帰することができないものの，他のプロセスの実行は継続できる場合に適用する．

Fail-Stop：回復処理を行っても矛盾なく OS カーネルを稼働し続けることができない場合や，破損したメモリオブジェクトの回復処理ができない場合にシステムを強制停止させる方法である．例えば，kfree() において kmem 構造体の破損が発覚した場合，複数ページにまたがる解放処理が中断されることによるメモリリークが発生する可能性がある．これを防ぐため，Ev6 は Fail-Stop によってシステムを強制停止する．

After-Treatment においてどの終了方法を選択するのかは，ECC-uncorrectable メモリエラーが発生するまでの関数呼び出しのコンテキストに依存する．例えば，ディスク上の内容から更新があったメモリオブジェクトが ECC-uncorrectable メモリエラーによって損傷した場合，ディスク書き込み内容の一部が失われる可能性があることから，リカバリ後も一貫性を保って実行することができなくなる．このため，リカバリハンドラではカーネルのコンテキストを踏まえた終了方法の選択を行う．具体的には，リカバリハンドラがカーネルコンテキストをコールスタックを遡ることで追跡し，NMI が呼び出されるまでの関数呼び出し履歴を獲得する．

この After-Treatment によるアプローチは，プロセスから見えるシステムコールの結果とカーネル生存のために行った処理との間でギャップを生じさせる恐れがある．というのも，あるプロセス上で発行されたシステムコールが成功を返しても，別のプロセス上で稼働したリカバリハンドラにより，更新されたメモリオブジェクトやデータブロックの内容を切り離してしまう可能性があるためである．そこで，システム管理者が Ev6 に対してプロセスコンテキストの整合性かシステムの存続かを選択できるよう，*Conservative* と *Aggressive* という 2 つのモードを用意する．Conservative モードでは，リカバリ処理によるシステムコールの結果との矛盾を許さず，矛盾が発生しうる状況では Fail-Stop のような強力な終了方法を選択する．Aggressive モードでは OS カーネルの生存を第一に考え，矛盾を残す可能性を許容して Ev6 を可能な限り強制的に稼働させる．buf 構造体を例に取ると，ダーティな buf 構造体の中身をディスクに書き込む関数である bwrite() 内で発生した NMI は，破損した buf 構造体に何のデータが格納されていたか特定できないため，Conservative モードでは Fail-Stop で対処するが，Aggressive モードでは Syscall Fail により終了する．

また，表 1 の最下部に示すように，10 種類のメモリオブジェクトは Ev6 による回復の対象外となっており，これらのオブジェクトの破損が検知された場合には Ev6 は Fail-Stop にて対処する．例えば，スタック上に割り当てられる superblock 構造体や dinode 構造体は，リカバリを行ってオブジェクトポインタをすり替えることができず，回復が困難である．OS カーネルの信頼性を高めるためには，これらのリカバリが困難なオブジェクトのリカバリ方法を模索する必要があるといえる．

4.2.1 ファイルシステム

表 1 に示すように，Ev6 におけるファイルシステムのメモリオブジェクトを 3 つの回復方針に分類し，リカバリハンドラの設計を行った．ファイルシステムのメモリオブジェクトのうち，log 構造体は**復元**に，残りは**初期化**に分類した．このうち，log 構造体のリカバリについてその詳細を述べる．

log 構造体は Ev6 のファイルシステムにおけるジャーナリングを担うメモリオブジェクトである．Xv6 はこの log のうち，メンバ logheader にジャーナリングの対象となるデータブロック番号を記録し，他のメンバにはジャーナリングを行う上でのメタデータが主に格納されている．ECC-uncorrectable メモリエラーによる log の破損が通知されると，Ev6 は**復元**による Internal-Surgery を

表 1: 各メモリオブジェクトにおけるエラーハンドリングの概要

オブジェクト名	セマンティクス	終了方法	Internal-Surgery の概要	After-Treatment の概要
buf (bcache)	バッファキャッシュ	初期化	新しい buf ノードを割り当て，メンバ lock を sleeplock のリカバリハンドラにより回復．損傷したノードと新しいノードを入れ替え，log や disk のポインタを更新，log と inode のロックを解放．	Fail Stop (Conservative (bfree(), bwrite())) Process Kill (exit(), iupdate()) Syscall Fail (21 個の他関数)
log	インメモリ ジャーナリングログの メタデータ	復元	新しい log を割り当て，ディスク上のログのメタデータをスーパーブロックから読み出し，メンバ lock を spinlock のリカバリハンドラで回復．logheader とメンバ outstanding はレプリカから内容を復元，最後に対応する buf と inode のロックを解放．	Fail Stop (Conservative (log_write(), fileclose() & begin_op())) Process Kill (end_op() & sys_chdir(), exit()) Syscall Fail (9 個の他関数)
file (ftable)	使用中ファイルのポインタ類	初期化	新しい ftable を割り当てて lock を initlock() で回復，破損した file ノードは初期化，残りは古い ftable からコピー．proc が持つポインタを更新，破損した file に対応する inode の参照カウントを減らす．	Fail Stop (Conservative (sys_write())) Process Kill (exit()) Syscall Fail (16 個の他関数)
inode (icache)	ファイルのメタデータ	初期化	新しい icache を割り当てて lock を initlock() で回復，破損した inode のノードは lock を initsleeplock() で回復，残りのメンバは初期化，残りのノードは古い icache からコピー．file や proc が持つポインタを更新，log の outstanding の値を減らす．Conservative モードでは，古い inode のロックが保持されていた場合には Fail-Stop により停止．	Fail Stop (T_DEVICE, Conservative (sys_write(), itrunc(), iput(), iupdate(), writei(), create(), holding sleeplock by other processes)) Process Kill (exit(), idup()) Syscall Fail (19 個の他関数)
pipe	パイプ処理のバッファ	初期化	新しい領域を割り当て，lock を spinlock のリカバリハンドラで回復．パイプ処理に関わっていた file を探し，pipe のメンバ readpoen と writeopen を復元，パイプ処理を通常の失敗時のフローに乗せる．	Syscall Fail
cons	コンソールのバッファ	初期化	新しい cons を割り当て，lock を spinlock のリカバリハンドラで回復．残りのメンバであるバッファやインデックスを初期化．	Fail Stop (consoleintr()) Syscall Fail (3 個の他関数)
devsw	特殊デバイスの R/W 関数ポインタ	復元	新しい devsw を割り当て，元と同じ関数ポインタを代入して復元．リカバリプロセスが保持していた buf，inode のロックを解放．	Syscall Fail
pr	printf() のロック	復元	新しい pr を割り当て，lock を spinlock のリカバリハンドラで回復．メンバ locking を panic() の有無で適切な値を代入．	Fail Stop (panic(), kerneltrap()) Process Kill (2 個の他関数)
spinlock	ロック (spinlock)	復元	新しく割り当てた領域か，引数で与えられた領域において，カーネルの関数呼び出し履歴からロックのステータスを復元．	spinlock をメンバに持つオブジェクトに依存．
sleeplock	ロック (sleeplock)	復元	新しく割り当てた領域か，引数で与えられた領域において，カーネルの関数呼び出し履歴からロックのステータスを復元．sleeplock を保護する spinlock は spinlock のリカバリハンドラで回復．	sleeplock をメンバに持つオブジェクトに依存．
kmem	メモリ割り当ての排他制御	復元	新しい kmem を割り当て，lock を spinlock のリカバリハンドラで，フリーリストの先頭ポインタを run の M-list から回復．カーネルコンテキストにより，パイプ中の file を閉じ，proc と log のロックを解放，outstanding の値を減らす．	Fail Stop (kfree() & initproc) Process Kill (freeproc(), kfree(), uvmalloc(), uvmcopy()) Syscall Fail (kalloc())
run	空きページのメタデータ	破棄	破損した run ノードを特定，フリーリストにおける前のノードをポインタを次ノードを指すよう変更，kmem のロックを解放．カーネルコンテキストにより，パイプ中の file を閉じ，proc，log のロックを解放，outstanding の値を減らす．	Fail Stop (kfree() & initproc) Process Kill (freeproc(), kfree(), uvmalloc(), uvmcopy()) Syscall Fail (kalloc())
page table	ページテーブル	復元	L2，L1：新しいページを割り当て，*PTDUP* が管理する複製からマッピング情報を復元． L0：新しいページを割り当て，Direct Segment-based メタデータからマッピング情報を復元． L2 では対応する proc のメンバ pagetable を更新し，L1 と L0 では上位の PTE と PTDUP の情報を更新．	Fail Stop (Conservative (bfree(), bwrite(), freeproc(), itrunc(), iput(), bfree(), iupdate(), sys_write(), writei(), bwrite(), create(), log_write(), kvminit(), fileclose() & begin_op())) Process Kill (exec(), mappages(), walk()) Syscall Fail (16 個の他関数)

リカバリ対象外のオブジェクト名 (セマンティクス)
cpu (CPU ごとの状態); proc (プロセスごとの状態); disk (Virtio ディスクデバイスメタデータ); elfhdr, proghdr (ELF ヘッダ); superblock (ディスク上のスーパーブロック); dirent, dinode (ディスク上の inode); trapframe (ユーザトラップ時に保存される汎用レジスタ); context (コンテキストスイッチ時に保存される汎用レジスタ)

行い，After-Treatment として状況に応じて 3 つの終了方法を使い分ける．具体的には，Internal-Surgery では，メンバ outstanding と logheader，lock 以外のメンバをディスク上のスーパーブロックを読み出して復元する．lock は spinlock のリカバリハンドラを呼び出すことで回復する．outstanding と logheader は，通常実行時に値が頻繁に入れ替わり，かつ不適切な値による復元はリカバリ後のシステムクラッシュを招く．このため，この 2 つのメンバは複製を取っておき，再構築時に複製の値をコピーすることで復元を行う．加えて，他に依存関係を持つオブジェクト（buf，inode）のロックを解放したり，参照カウントを減じることで整合性を保つ．

After-Treatment は基本的に Syscall Fail で終了するものの，exit() が呼ばれている，もしくは chdir 中に end_op() で NMI が発生した場合には Process Kill で対処する．この理由は，前者の場合には Process Kill の途中で ECC-uncorrectable メモリエラーが発覚したため，リカバリ後は再度 Process Kill を行えばよいためである．後者の場合には chdir を発行したプロセスのカレントディレクトリの inode が解放されているのにも関わらず，カレントディレクトリが移動していない状態となることから，Process Kill によってプロセス自体を強制終了することで対処するためである．また，更新の内容をメモリ上のログに記録する関数である log_write() 内で NMI が起きると，変更内容をロストする危険があるため，Conservative モードではシステムコールの結果との矛盾を防ぐために Fail-Stop で対処する．

4.2.2 メモリマネージャ

ファイルシステムのケースと同様に，表 1 に示すようにメモリマネージャに属する 3 種類のオブジェクトを分類した．kmem 構造体とページテーブルは**復元**の方針を，run 構造体は**破棄**の方針を採用した．このうち，ページテーブルのリカバリについて詳細を説明する．

2019 年版の RISC-V アーキテクチャでは 3 段のページテーブル（上から L2，L1，L0）を採用しており，Ev6 のリカバリハンドラはどの層のページテーブルにおいても複製を用いた復元を行う．これを実現するため，Ev6 は効率的にマッピング情報を複製・管理する機構である PTDUP を導入した．Internal-Surgery の段階では，損傷したページテーブルページの代わりとなる物理ページを割り当て，PTDUP が管理するメタデータをもとに破損前のマッピングを復元する．この時，もし破損したページテーブルが L2（最上位）ページテーブルであれば，proc 構造体のメンバ pagetable 内に格納されているページテーブルへのポインタを更新し，それ以外の層であれば上位のページテーブルと PTDUP が管理する複製のエントリを更新する．After-Treatment の段階では，リカバリハンドラはメモリエラーが発生したプロセスのカーネルスタックを走査し，Fail-Stop，Process Kill，Syscall Fail のどの方法で終了するのかを決定する．

ページテーブルのマッピング情報を全てコピーする方法は，最大でページテーブルサイズと同量の巨大なメモリスペースオーバヘッドを生じさせることになる．これに対処するため，PTDUP メカニズムでは，range-based メモリマッピングを行う Direct Segment [50] のアイデアをもとに， L0 ページテーブルの複製の大きさを縮小する工夫を行う．図 3 に示すように，仮想・物理アドレスが共に連続している複数のエントリは Direct Segment により 1 つのエントリにまとめ，非連続なエントリはそのまま複製をとることで，連続しているエントリの分だけ複製のサイズを圧縮することが可能になる．Ev6 で実装したプロトタイプでは，連続したエントリをまとめる際，PTE に割り当てられ

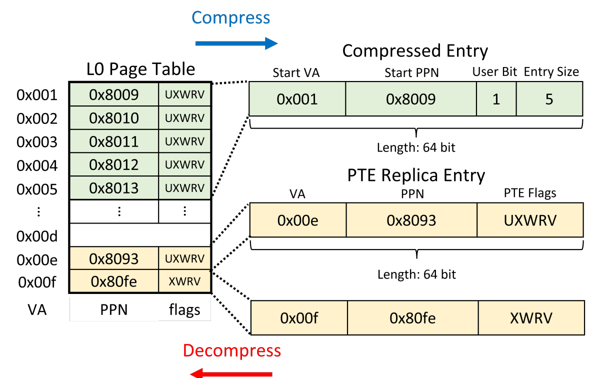

図 3: *PTDUP* における L0 ページテーブルの圧縮と復元の概観

ているフラグのうち User ビットのみを記録している．これは，他の Executable，Readale，Writable，Valid のフラグはユーザのページテーブル上では全て立っているとみられることから Internal-Surgery 時にその場で復元が可能であり，残りの Access，Dirty ビットはシステム生存の観点に絞った場合には欠落してもシステムを生かすことができると判断したためである．この圧縮したエントリは，図 3 に示すように L0 ページテーブルのリカバリの際に展開され，非連続なエントリの複製と組み合わせて L0 ページテーブルのマッピングを復元する．

4.2.3 コンソール

表 1 に示すように，コンソールに属する 3 種類のオブジェクトをそれぞれ**復元**と**初期化**に分類した．例えば，コンソールのような特殊デバイスと呼ばれるモジュールに対する読み書きを行う関数の関数ポインタを管理する devsw 構造体は復元に分類される．具体的なリカバリハンドラの動作としては，Internal-Surgery において新しい devsw 用の領域を確保し，オリジナルと同様の関数ポインタ（xv6 ではコンソールのみ）を代入する．そして，関係するメモリオブジェクトのロックを解放してから Syscall Fail によって回復処理を終了する．

4.2.4 ロック

Xv6 は *spinlock* と *sleeplock* という 2 種類のロックを採用しており，それぞれにロックの状態を示すメンバ lock が用意されている．表 1 に示すように，これらは共に**復元**に分類されている．リカバリハンドラでは，最初に新しいロック用の領域を割り当てるか，もしくはロックをメンバに持つオブジェクトの領域に再構築を行う．ロックの状態を示す lock の値を復元するには，カーネルのコンテキストに大きく依存するため，リカバリハンドラではカーネルスタックを走査することでロック状態の特定を試みる．具体的には，もしロックの破損がロック解放の前，もしくはロック取得の後だった場合にはロック済みとして初期化し，一方で解放後だった場合にはロックは未取得として初期化する．

ⓒ 2022 Information Processing Society of Japan

4.3 M-List

ECC-uncorrectable メモリエラーによって破損したメモリオブジェクトを特定し，適切なリカバリハンドラを呼び出すために Ev6 は M-list という機構を新たに導入する．これは，Ev6 内で生成されるメモリオブジェクトの先頭アドレスを記録しておき，ECC-uncorrectable メモリエラーが起きた際に走査することで，ECC モジュールから得られたアドレスと破損したメモリオブジェクトとを紐付ける役割を果たす．M-list により破損したオブジェクトのタイプ及び破損したノードが判明した段階で，オブジェクトのタイプに合ったリカバリハンドラを呼び出す．

M-list が正しくオブジェクトの割り当て状況を反映している状態を保つため，Ev6 はメモリオブジェクトの割り当てと解放を監視し，M-list を管理する．例えば，run 構造体はそれぞれ物理メモリページの解放と割り当てに合わせてオブジェクトの数が増減するため，それに合わせて M-list の run 構造体のアドレスリストから当該ノードのアドレスを登録・削除を実施する．

4.4 NMI Shepherd

Ev6 の信頼性を向上させるには，Ev6 を構成する各モジュールだけでなく，本論文で導入した機構における ECC-uncorrectable メモリエラーにも対処する必要がある．例えば，あるメモリオブジェクトにおける ECC-uncorrectable メモリエラーの処理中に別のオブジェクトや M-list におけるメモリエラーが検知される可能性がある．また，複数のプロセスが並列して稼働するマルチコア環境において，ECC-uncorrectable メモリエラーに起因する NMI が立て続けに別々のコア間で発生することが考えられる．これらのケースでは，最悪の場合，メモリエラーを処理する最中に別のメモリエラーが検知され続けることによって回復処理が終了せずに無限ループしたり，先行する回復処理が後続の回復処理で上書きされたりするという状況を招く危険がある．

この問題に対処するため, Ev6 ではマルチコア環境において複数の NMI を安全に処理するための機構である *bounded NMI shepherd* を導入する．図 4 に示すように，メモリオブジェクトのリカバリ中に別のプロセスで ECC-uncorrectable メモリエラーに起因する NMI が発生した場合，NMI ハンドラは回復に必要な情報をキューにエンキューしておき，先にリカバリハンドラに入ったプロセスに後続のプロセスで発覚したオブジェクト破損のリカバリを委任する．後続のプロセスは，自身のリカバリ要求が先行するリカバリプロセスによって処理されるまでスピンしておくことで，Internal-Surgery から After-Treatment までの一連の回復処理に入るプロセスを 1 つに制限する．もしキューが満杯であったり，リカバリプロセスで別のメモリエラーが検知された場合にはカーネルパニックを引き起こして Ev6 を強制終了させる．

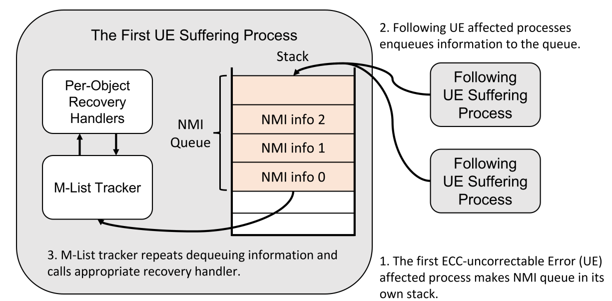

図 4: *NMI shepherd* の概観

NMI shepherd の設計を行う上で，ECC-uncorrectable メモリエラーによって NMI shepherd の動作が正しく完了できない場合の対処を考える必要がある．例えば，NMI ハンドラ内のテキスト領域で ECC-uncorrectable メモリエラーが発生した場合，場所によっては NMI キューにリカバリのための情報を積み続ける無限ループが発生する危険がある．

これに対処するため，Ev6 は NMI キューを最初に回復処理に突入したプロセスのカーネルスタック上に設置することで対処する．上記で述べたキューに積み続ける無限ループが発生した場合，いずれカーネルスタックからオーバフローし，カーネルスタックの上部に置かれているガードページに突入する．これにより，プロテクションフォールトが発生することで Ev6 はカーネルパニックによって強制終了し，無限ループから抜け出すことが可能になる．

5. 評価実験

Ev6 のプロトタイプを RISC-V アーキテクチャを採用した 2019 年版の xv6 に対して実装を行った．xv6 は NMI ハンドラを持たないため，提案機構に加えて NMI ハンドラを模した関数を用意し，kerneltrap() から呼び出せるように実装した．このプロトタイプを Ubuntu 5.4.0 上の QEMU 5.1.0 において 3 コア（harts），メモリ 5 GB の構成で稼働させ，評価実験を行った．本実験では，Ev6 における ECC-uncorrectable メモリエラーに対する回復能力，Ev6 による通常実行時のオーバヘッド，及び追加で発生するメモリスペースオーバヘッドを計測することで Ev6 の性質を明らかにする．

5.1 フォールトインジェクション

ECC-uncorrectable メモリエラーに対する Ev6 の回復能力を実証するため，ソフトウェアベースのフォールトインジェクションを実施した．調査した限りでは，OS カーネル内の任意のメモリアドレスに ECC-uncorrectable メモリエラーを模したフォールトを挿入できるフォールトインジェクタは存在しない．このため，Ev6 のコード内にシステムコールによりフラグを立てると発火する機構を挿入

ⓒ 2022 Information Processing Society of Japan

し，kerneltrap() を経由して意図的に破損したメモリオブジェクトのアドレスを引数として NMI ハンドラを呼び出すように実装した．

今回実施したフォールトインジェクションは，単一のフォールトを挿入する *Single* と複数のフォールトが発火する *multiple* の 2 つのケースを用意した．いずれのケースにおいても，こちらが想定したシナリオに基づき，エラーを挿入している．なお，Ev6 による ECC-uncorrectable メモリエラーからのリカバリが成功したと判断する基準として，xv6 全体にストレスをかけるリグレッションテストである usertests をリカバリが終わった後に全 46 テストをパスすることとした．

5.1.1 Single Fault Cases

ECC-uncorrectable メモリエラーを模した単一のフォールトインジェクションテストの結果を表 2 に示す．表から見て取れるように，Ev6 は多くのケースにおいてリカバリによるシステムの継続実行に成功しており，中でもルートディレクトリなどによる Fail-Stop のケースを除けば，実に 30 ケースで Aggressive と Conservative の両方でシステムが生存していることが分かる．

また，14 個のケースにおいて Ev6 は Fail-Stop によってシステムを停止させることに成功している．例えば，本研究において回復対象外となった `disk` 構造体やテキスト領域における ECC-uncorrectable メモリエラーを模したフォールトに対しても，M-list を走査した結果対象外として正しく弾き，Fail-Stop を選択できていることが分かる．

5.1.2 Multiple Faults Cases

複数の NMI が発生するフォールトインジェクションテストの結果を表 3 に示す．Multiple ケースは 2 つのカテゴリに分かれる．1 つ目は 1 つのメモリオブジェクトにおける ECC-uncorrectable メモリエラーに複数のコアで遭遇した場合である．この状況では，4.4 項において説明した NMI shepherd によって複数の連続した NMI を順番に処理できた場合，Fail-Stop によるシステムの強制終了を回避することができている．例えば，異なるコア間で `log` 構造体のメモリエラーが begin_op() において発覚したシナリオを例に取ると，最初のプロセスが回復処理に突入した後に別プロセスにて NMI が発行されると，用意しておいた NMI キューに回復要求をエンキューする．最初のプロセスにて発覚した NMI が処理されると，当該プロセスは NMI キューから積まれていた回復要求をデキューする．回復要求の対象である `log` 構造体は既に回復済みであるため，リカバリハンドラにおける再構築処理をスキップして終了する．

また，同一のメモリオブジェクト間だけでなく，異なるメモリオブジェクト間における ECC-uncorrectable メモリエラーを NMI shepherd がシリアライズ可能かを検証するため，異なるメモリオブジェクトのアドレスを引数として，直接 NMI を呼び出すシステムコール corrupted() を用意した．この corrupted() を複数プロセス間で繰り返し呼び出すことで，通常実行のコンテキストを無視して強制的に NMI が重なる状況を再現し，後続の NMI によって先行する NMI が横取りされないかを確認する．結果として，NMI shepherd は 4.4 項で述べたような，先行するリカバリ処理が後続の NMI による中断されたり，複数のリカバリ処理が並列に走る状況を防ぐことができていた．xv6 のプログラムソースを無視した強制的なフォールトインジェクションであったが，リカバリに成功して usertests を全てパスするケースも見られた．

2 つ目は，同一のコアにおいてリカバリハンドラ内で別の ECC-uncorrectable メモリエラーが発覚した場合である．例えば，M-list の走査中に M-list 自体の破損が発覚したり，NMI ハンドラ内で別のメモリエラーが発覚する可能性がある．この場合，Ev6 は NMI shepherd におけるガードページの利用や，回復処理中の NMI を検知することで Fail-Stop によりシステムを強制終了させる．

5.2 通常実行時オーバヘッド

Ev6 が通常実行時にどの程度のオーバヘッドを引き起こすのかを検証するため，通常の xv6 と Ev6 間で usertests の実行時間の比較を行った．usertests を順番に実行していく中で，各テストにおける実行時間を uptime() システムコールにより ticks ベースで測定する作業を 30 回繰り返した．

得られた結果を図 5 に示す．X 軸は usertests における各テスト名を，Y 軸はバニラな xv6 で正規化した Ev6 の実行時間を示している．図より分かるように，実行時最大オーバヘッドは 7 倍程度であり，平均値は約 1.4 倍であった．また，実行時オーバヘッドの大小は，メモリオブジェクトの生成・削除による M-list への登録・削除の量に依存していた．例えば，メモリ割り当て・解放を頻繁に行う exitwait，sbrkmuch といったテストは，`run` 構造体を頻繁に削除・生成するため M-list を頻繁に走査することによるオーバヘッドの影響が大きい．一方で sbrkbugs や bsstest はバニラな xv6 よりも性能が改善しているが，これはテストの実行時間が短すぎることにより，tick の値がテストの実行タイミングによって変化してしまったことが原因と考える．

5.3 メモリスペースオーバヘッド

Ev6 に起因するメモリスペースオーバヘッドを検証するため，Ev6 を構成する機構のメモリ消費量を調査した．追加のメモリスペースオーバヘッドは M-list と PTDUP が支配的と予想されるため，これらによるメモリ消費量の変化を観測しやすい 3 つの状況設定を用意した．1 つ目は，シェルが起動し終わった直後にメモリ消費量を測る「ブー

ⓒ 2022 Information Processing Society of Japan

表 2: フォールトインジェクションの実験結果（Single Fault Cases）

（*SF*：Syscall Fail，*PK*：Process Kill，*FS*：Fail-Stop）

オブジェクト名	挿入関数	シナリオ	結果
buf	bget()	バッファキャッシュとして空の buf を探索中にエラーに遭遇.	*SF*
	log_write()	log にダーティなブロック番号を書き込もうとして buf のエラーにアクセス.	
	bwrite()	ディスクにダーティな buf ノードのデータを書き込もうとしてエラーを触った.	*SF*（*Aggr*），*FS*（*Cons*）
log	begin_op()	log セクションに入るために log に触るタイミングでエラーに遭遇.	*PK*
	commit()	log の内容をコミットするタイミングで log の破損が発覚.	
	log_write()	メモリ上の log に更新されたブロック番号を書き込もうとしてエラーにアクセス.	*PK*（*Aggr*），*FS*（*Cons*）
	install_trans()	ディスク上の log の内容を正しい位置に反映する関数内でメモリエラーに触った.	*SF*
	write_log()	ディスク上のジャーナル領域に log を書き込もうとしてメモリエラーに遭遇.	
file	filealloc()	新しい file ノードを割り当てるタイミングでメモリエラーに触った.	*SF*
	fileread()	ファイルの内容を読もうとして損傷した file エントリに触った.	
	fileread()	パイプ処理の最中に読み込み側のファイルがメモリエラーに侵されていた.	
	fileclose()	exit() でプロセスを終了させる際に file 構造体が破損していた.	*PK*
	filedup()	ファイルディスクリプタの複製対象だった file ノードが破損していた.	
	filewrite()	ファイルを中身を書き込もうとして file におけるメモリエラーに遭遇.	*PK*（*Aggr*），*FS*（*Cons*）
inode	exec()	exec() の最中に開いたファイルの inode が破損していた.	*SF*, *FS*（T_DEVICE）
	idup()	fork() の際に inode の参照カウントを増やそうとしてエラーに遭遇.	*PK*, *FS*（*root* dir）
	writei()	変更が生じたデータ inode とディスクに書き込もうとして inode のエラーにアクセス.	*SF*（*Aggr*），*FS*（*Cons*, *root* dir）
	iget()	icache 上かディスク上の inode を走査中に inode の破損が発覚.	*SF*, *FS*（*root* dir）
	ilock()	inode のロックを確保しようとして損傷した inode にアクセス.	
	readi()	メモリかディスク上から inode を読み出そうとしてメモリエラーに遭遇.	*SF*
pipe	piperead()	パイプで連結されたファイルが pipe 内のデータを読み出そうとしてエラーに遭遇.	*SF*
	pipewrite()	パイプで連結されたファイルが pipe にデータを書き込もうとしてエラーに遭遇.	
spinlock	file, log, pipe, pr, sleeplock, kmem	spinlock をメンバに持つオブジェクトがメモリエラーに侵され，spinlock のリカバリハンドラが稼働.	*SF*, *PK*
sleeplock	buf, inode	sleeplock をメンバに持つオブジェクトがメモリエラーに侵され，sleeplock のリカバリハンドラが稼働.	*SF*, *PK*
cons	consoleread()	scanf() で入力された文字列を受け取ろうとして損傷した cons にアクセス.	*SF*
	consolewrite()	文字列をコンソール上に出力しようとして損傷した cons にアクセス.	
pr	printf()	カーネル空間における printf() を呼び出したことで破損した pr 内のエラーに遭遇.	*PK*
	kerneltrap()	プロテクションフォールトによるカーネルパニック時に破損した pr にアクセス.	*FS*
kmem	kalloc()	物理メモリページを割り当てようとしたタイミングで kmem 内のエラーが発覚.	*PK*
	kfree()	物理メモリページを解放しようとしたタイミングで kmem 内のエラーが発覚.	
run	kalloc()	フリーリストから物理メモリページを取り出そうとして損傷した run に触った.	*PK*
	kfree()	解放した物理メモリページをフリーリストに加えようとして損傷した run に触った.	
devsw	fileread()	コンソールからの入力データを受け取る際に，損傷した devsw 構造体に触ってしまった.	*SF*
	filewrite()	コンソールへデータを出力する際に，損傷した devsw 構造体に触ってしまった.	
ページテーブル	L2: bwrite()	MMU によるページテーブルウォークの際に L2 ページテーブルの破損が発覚した.	*PK*（*Aggr*），*FS*（*Cons*）
	L2: walk()	ソフトウェアによるページテーブルウォークの際に L2 ページテーブルの破損が発覚.	
	L1: walk()	ソフトウェアによるページテーブルウォークの際に L1 ページテーブルの破損が発覚.	
	L0: walkaddr()	ソフトウェアによるページテーブルウォークの際に L0 ページテーブルの破損が発覚.	*SF*（*Aggr*），*FS*（*Cons*）
disk	alloc_desc()	ディスク書き込みの処理中に破損した disk 構造体に触ってしまった.	*FS*
テキスト領域	bread()	bread() を呼んだ際にテキスト領域が損傷していた.	*FS*

ト直後」である．2 つ目は，「全割り当て / 解放」である．このシナリオでは，usertests 内のテストの 1 つで，1 つのプロセスが割り当て可能な全メモリを取得・解放する mem を用いて，全割り当て，全解放の前後のメモリ消費量を計測する．3 つ目は，「全割り当て / 解放（非連続フリーリスト）」である．このシナリオは 2 つ目のシナリオと同じ mem を用いるものの，その前にフリーリストの順序を乱して非連続なメモリページ割り当てが行われる状況を想定する．これにより，仮想・物理アドレスが連続したエントリをまとめる PTDUP 圧縮メカニズムによる圧縮効率が悪化するため，メモリ消費量が増加することが予想される．

実験結果を表 4 に示す．起動直後の Ev6 によるメモリスペースオーバヘッドは 572 KB であり，この時点では静的なオブジェクトの M-list 登録が全て完了している．2 つ目のシナリオである「全割り当て / 解放（連続フリーリスト）」では合計メモリ消費量が 696 KB から最大 736 KB まで増加しており，その増加分は PTDUP によるものであった．3 つ目のシナリオでは，最大合計メモリ消費量が 960 KB まで増加しており，PTDUP による増加量は 392 KB と 2 つ目のシナリオの倍以上であった．このことから，PTDUP 圧縮メカニズムによる圧縮効率は L0 ページテーブルに割り当てられるページのアラインメントに大きく依存することが分かる．

6. おわりに

本論文では，ECC メモリにおいて検知可能でも訂正不可

ⓒ 2022 Information Processing Society of Japan

表 3: フォールトインジェクションの実験結果（Multiple Faults Cases）

（*SF*：Syscall Fail，*PK*：Process Kill，*FS*：Fail-Stop）

オブジェクト名	挿入関数	シナリオ	結果
log	begin_op()	2 つのプロセスがそれぞれログセクションに入ろうとしてお互いに破損した log に触った．	*FS*（*NMI shepherd*），*PK*
	commit() & begin_op()	一方のプロセスがコミット中に別プロセスがログセクションに入ろうとして，お互いに破損した log に触った．	*FS*（*NMI shepherd*），*SF*（commit()），*PK*（begin_op()）
kmem	kfree() & kalloc()	一方のプロセスの kmem の回復中に別プロセスが新しい空きページを割り当てようとした．	*FS*（*NMI shepherd*），*PK*
devsw	filewrite()	複数のプロセスが並列にユーザ空間の printf() を呼び，破損した devsw にアクセス．	*FS*（*NMI shepherd*），*PK*
pipe	piperead()，pipewrite()	パイプ処理で繋がったファイル同士が並列に破損した pipe に触った．	*FS*（*NMI shepherd*），*SF*
cons	consolewrite()	複数のプロセスが並列にユーザ空間の printf() を呼び，破損した cons にアクセス．	*FS*（*NMI shepherd*），*SF*
cons & log	無し（corrupted() により生成）	一方のプロセスの cons の回復中に別プロセスが log のメモリエラーに遭遇．	*SF*，リカバリ失敗
devsw & log		一方のプロセスの devsw の回復中に別プロセスが log のメモリエラーに遭遇．	
proc	mem_nmi_handle_follow()	NMI ハンドラの処理中に proc 構造体におけるメモリエラーに遭遇．	*FS*
カーネルスタック	mem_nmi_handle_first()	NMI ハンドラの処理中にカーネルスタックにおけるメモリエラーに遭遇．	*FS*
M-list	mediator()	M-list の走査中に M-list 自体の破損が発覚．	*FS*
テキスト領域	recvr_log()	リカバリハンドラ中のテキスト領域がメモリエラーに侵されており，回復中に発覚．	*FS*
	nmi_handle()	NMI ハンドラ内のテキスト領域がメモリエラーに侵されていた．	
NMI キュー	mem_nmi_handle_follow()	NMI の処理要求を積む NMI キューが破損しており，NMI ハンドラ内でそれが発覚．	*FS*

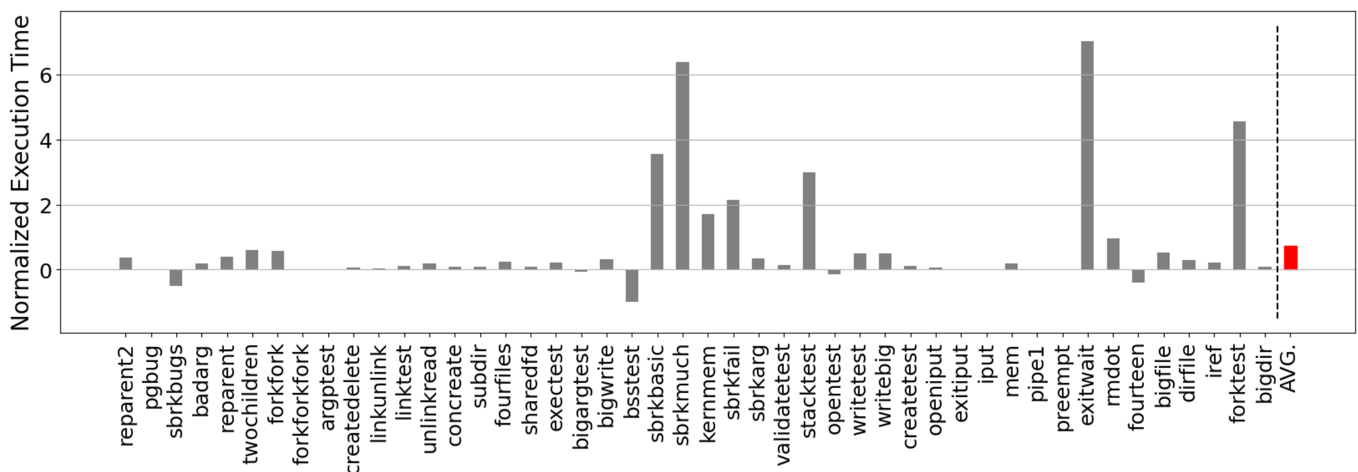

図 5: usertests による性能テストの結果

能なメモリエラーである，ECC-uncorrectable メモリエラーを検知した場合，リカバリを実施することで OS カーネルを含むシステム全体の生存を可能にする Ev6 を提案した．過去，システムの信頼性に着目した様々な既存研究が行われてきたが，リカバリによるシステムのダウンタイム，メモリを大量に消費するアプリケーションの再稼働までの時間，OS カーネルの構造を刷新する必要があるといった弱点があった．

本論文で提案した Ev6 では，ECC-uncorrectable メモリエラーに侵された OS カーネル内のメモリ領域をメモリオブジェクトの単位で切り離し，再構築を行う回復処理を施すことで，システム全体の再起動ではなく継続実行を指向することにより，システムの信頼性と可用性を向上させる．これを実現するため，破損箇所のアドレスに対応するメモリオブジェクトを特定する M-list，メモリオブジェクト間で大きく異なるオブジェクトセマンティクスや依存関係に対応した回復処理を施す per-object recovery handler，Ev6 の機構を含む様々なメモリエラーに対応するための NMI shepherd を導入した．

提案手法のプロトタイプを xv6 に対して実装し，ECC-uncorrectable メモリエラーを模したフォールトインジェクションをはじめとする評価実験を行った．その結果，用意したクラッシュシナリオに対して Ev6 はリカバリによる生存を低いメモリスペースオーバヘッドで可能にしたことが確認された．

本論文で紹介した After-Treatment は，システム管理者が指定した制限モードに応じ，ユーザアプリケーションとの連携をしないことを前提として設計した．このため，ユーザコンテキストを鑑みないことでその後のユーザアプリケーションの処理が失敗することが考えられる．例えば，ファイルの read() システムコールを発行した際に ECC-uncorrectable メモリエラーに対するリカバリが走ると，そのファイルに対応する `file` 構造体はクリアされてしまうため，Syscall Fail でユーザコンテキストに返っ

表 4: メモリスペースオーバヘッド

シナリオ	追加のメモリ消費量	内訳	平均オーバヘッド
ブート直後	624 KB	M-list：568 KB（run：516 KB，ページテーブル：8 KB），PTDUP：56 KB	0.0 %
全割り当て / 解放（連続フリーリスト）	全割り当て前：696 KB	M-list：568 KB（run：516 KB，ページテーブル：8 KB），PTDUP：128 KB	+ 4.7 %
	全割り当て後：736 KB	M-list：568 KB（run：516 KB，ページテーブル：8 KB），PTDUP：168 KB	
	全解放後：696 KB	M-list：568 KB（run：516 KB，ページテーブル：8 KB），PTDUP：128 KB	
全割り当て / 解放（非連続フリーリスト）	全割り当て前：696 KB	M-list：568 KB（run：516 KB，ページテーブル：8 KB），PTDUP：128 KB	+ 170.4 %
	全割り当て後：960 KB	M-list：568 KB（run：516 KB，ページテーブル：8 KB），PTDUP：392 KB	
	全解放後：696 KB	M-list：568 KB（run：516 KB，ページテーブル：8 KB），PTDUP：128 KB	

ても open() システムコールを発行しない限り，その後のファイルへの処理は全て失敗し続けることになる．今後の展望として，この問題に対処すべく，`file` 構造体における破損時には open() システムコールを再発行するというように，ユーザアプリケーションとの連動を前提とした After-Treatment によるユーザコンテキストを考慮したメカニズムの開発を行う必要がある．

参考文献

[1] Chen, C. L. and Hsiao, M. Y.: Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review, *IBM J. Res. Dev.*, Vol. 28, No. 2, pp. 124–134 (online), DOI: 10.1147/rd.282.0124 (1984).

[2] Schroeder, B., Pinheiro, E. and Weber, W.-D.: DRAM Errors in the Wild: A Large-scale Field Study, *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, New York, NY, USA, ACM, pp. 193–204 (online), DOI: 10.1145/1555349.1555372 (2009).

[3] Li, X., Huang, M. C., Shen, K. and Chu, L.: A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility, *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, Berkeley, CA, USA, USENIX Association, pp. 6–6 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1855840.1855846⟩ (2010).

[4] J. Meza, Q. Wu, S. K. and Mutlu, O.: Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field, *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pp. 415–426 (online), DOI: 10.1109/DSN.2015.57 (2015).

[5] Zhang, L., Neely, B., Franklin, D., Strukov, D., Xie, Y. and Chong, F. T.: Mellow Writes: Extending Lifetime in Resistive Memories through Selective Slow Write Backs, *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 519–531 (online), DOI: 10.1109/ISCA.2016.52 (2016).

[6] Zhang, M., Zhang, L., Jiang, L., Liu, Z. and Chong, F. T.: Balancing Performance and Lifetime of MLC PCM by Using a Region Retention Monitor, *Proc. of the IEEE International Symposium on High Performance Computer Architecture (HPCA '17)*, pp. 385–396 (2017).

[7] Hsiao, M. Y.: A Class of Optimal Minimum Odd-Weight-Column SEC-DED Codes, *IBM J. Res. Dev.*, Vol. 14, No. 4, pp. 395–401 (online), DOI: 10.1147/rd.144.0395 (1970).

[8] Patel, M., Kim, J. S., Hassan, H. and Mutlu, O.: Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices, *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 13–25 (2019).

[9] Hwang, A. A., Stefanovici, I. A. and Schroeder, B.: Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design, *Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, pp. 111–122 (2012).

[10] Sridharan, V., DeBardeleben, N., Blanchard, S., Ferreira, K. B., Stearley, J., Shalf, J. and Gurumurthi, S.: Memory Errors in Modern Systems: The Good, The Bad, and The Ugly, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, New York, NY, USA, ACM, pp. 297–310 (online), DOI: 10.1145/2694344.2694348 (2015).

[11] Taranov, K., Alonso, G. and Hoefler, T.: Fast and Strongly-consistent Per-item Resilience in Key-value Stores, *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, ACM, pp. 39:1–39:14 (online), DOI: 10.1145/3190508.3190536 (2018).

[12] Li, Y., Wang, H., Zhao, X., Sun, H. and Zhang, T.: Applying Software-based Memory Error Correction for In-Memory Key-Value Store: Case Studies on Memcached and RAMCloud, *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, New York, NY, USA, ACM, pp. 268–278 (online), DOI: 10.1145/2989081.2989091 (2016).

[13] Didehban, M., Shrivastava, A. and Lokam, S. R. D.: NEMESIS: A software approach for computing in presence of soft errors, *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 297–304 (online), DOI: 10.1109/ICCAD.2017.8203792 (2017).

[14] Ni, X. and Kalé, L. V.: FlipBack: automatic targeted protection against silent data corruption, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pp. 335–346 (online), DOI: 10.1109/SC.2016.28 (2016).

[15] Kaashoek, F., Morris, R. and Cox, R.: Xv6 (2006).

[16] Dell, T. J.: A white paper on the benefits of chipkill-correct ECC for PC server main memory, *IBM Microelectronics division*, Vol. 11, pp. 1–23 (1997).

[17] : Memcached, `https://memcached.org/`. Online. (accessed 2022-03-30).

[18] : Google Compute Engine.

[19] : Amazon Elastic Compute Cloud.

[20] Goel, A., Chopra, B., Gerea, C., Mátáni, D., Metzler, J., Ul Haq, F. and Wiener, J.: Fast Database Restarts at Face-

book, *Proc. of the 2014 ACM SIGMOD international conference on Management of data (SIGMOD '14)*, pp. 541–549 (2014).

[21] Schirmeier, H., Neuhalfen, J., Korb, I., Spinczyk, O. and Engel, M.: RAMpage: Graceful Degradation Management for Memory Errors in Commodity Linux Servers, *Proc. of the 2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing (PRDC '11)*, pp. 89–98 (2011).

[22] Budhiraja, N., Marzullo, K., Schneider, F. B. and Toueg, S.: *The Primary-Backup Approach*, pp. 199–216, ACM Press/Addison-Wesley Publishing Co. (1993).

[23] Bolosky, W. J., Bradshaw, D., Haagens, R. B., Kusters, N. P. and Li, P.: Paxos Replicated State Machines as the Basis of a High-Performance Data Store, *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*, pp. 141–154 (2011).

[24] Ongaro, D. and Ousterhout, J.: In Search of an Understandable Consensus Algorithm, *Proc. of the 2014 USENIX Annual Technical Conference, (USENIX ATC '14)*, pp. 305–319 (2014).

[25] Intel Corporation: Intel Address Range Partial Memory Mirroring (Accessed: 2021-11-16).

[26] Tsalapatis, E., Hancock, R., Barnes, T. and Mashtizadeh, A. J.: The Aurora Single Level Store Operating System, *Proc. of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, pp. 788–803 (2021).

[27] : CRIU: Checkpoint/Restore In Userspace.

[28] Vogt, D., Giuffrida, C., Bos, H. and Tanenbaum, A. S.: Lightweight Memory Checkpointing, *Proc. of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, pp. 474–484 (2015).

[29] Duell, J., Hargrove, P. and Roman, E.: Requirements for Linux Checkpoint/Restart, Technical Report 8, Berkeley Lab Technical Report, LBNL-49659 (2002).

[30] Rieker, M., Ansel, J. and Cooperman, G.: Transparent User-Level Checkpointing for the Native POSIX Thread Library for Linux, *Proc. of 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '06)*, pp. 492–498 (2006).

[31] Kashyap, S., Min, C., Lee, B., Kim, T. and Emelyanov, P.: Instant OS Updates via Userspace Checkpoint-and-Restart, *Proc. of the 2016 USENIX Annual Technical Conference (ATC '16)*, pp. 605–619 (2016).

[32] Siniavine, M. and Goel, A.: Seamless Kernel Updates, *Proc. of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, pp. 1–12 (2013).

[33] Terada, K. and Yamada, H.: Dwarf: Shortening Downtime of Reboot-based Kernel Updates, *Proc. of the 12th European Dependable Computing Conference (EDCC '16)*, pp. 208–217 (2016).

[34] Yamada, H. and Kono, K.: Traveling Forward in Time to Newer Operating Systems Using ShadowReboot, *Proc. of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '13)*, pp. 121–130 (2013).

[35] Bhat, K., Vogt, D., van der Kouwe, E., Gras, B., Sambuc, L., Tanenbaum, A. S., Bos, H. and Giuffrida, C.: OSIRIS: Efficient and Consistent Recovery of Compartmentalized Operating Systems, *Proc. of the 46th Annual IEEE/IFIP International Conference on Dependabile Systems and Networks (DSN '16)*, pp. 25–36 (2016).

[36] David, F. M., Chan, E. M., Carlyle, J. C. and Campbell, R. H.: CuriOS: Improving Reliability through Operating System Structure, *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 59–72 (2008).

[37] : Rust, `https://www.rust-lang.org/`. Online. (accessed 2022-04-01).

[38] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S.: SeL4: Formal Verification of an OS Kernel, *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pp. 207–220 (2009).

[39] Boos, K., Liyanage, N., Ijaz, R. and Zhong, L.: Theseus: an Experiment in Operating System Structure and State Management, *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, USENIX Association, pp. 1–19 (online), DOI: 10.5555/3488766.3488767 (2020).

[40] Narayanan, V., Huang, T., Detweiler, D., Appel, D., Li, Z., Zellweger, G. and Burtsev, A.: RedLeaf: Isolation and Communication in a Safe Operating System, *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, USENIX Association, pp. 21–39 (online), DOI: 10.5555/3488766.3488768 (2020).

[41] Depoutovitch, A. and Stumm, M.: Otherworld - Giving Applications a Change to Servive OS Kernel Crashes, *Proc. of the 5th ACM European Conference on Computer Systems (EuroSys '10)*, pp. 181–194 (2010).

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

[43] : Reboot Linux faster using kexec.

[44] Swift, M. M., Bershad, B. N. and Levy, H. M.: Improving the Reliability of Commodity Operating Systems, *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp. 207–222 (2003).

[45] Swift, M. M., Annamalai, M., Bershad, B. N. and Levy, H. M.: Recovering Device Drivers, *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 1–16 (2004).

[46] Sundararaman, S., Subramanian, S., Rajimwale, A., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H. and Swift, M. M.: Membrane: Operating System Support for Restartable File Systems, *Proc. of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, pp. 281–294 (2010).

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

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

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

[50] Basu, A., Gandhi, J., Chang, J., Hill, M. D. and Swift, M. M.: Efficient Virtual Memory for Big Memory Servers, *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*, Tel-Aviv, Israel, ACM, pp. 237–248 (online), DOI: 10.1145/2485922.2485943 (2013).

ⓒ 2022 Information Processing Society of Japan