Vol.2022-OS-155 No.3
2022/5/26

耐メモリ故障性の強化を支援するフォールトインジェクタ

根津 直也[1]　山田 浩史[1]

概要：メモリ容量の増大や，電圧の微細化などの原因によるメモリエラーの増加に伴い，ソフトウェアにメモリエラー耐性を持たせるための手法が多く提案されている．フォールトインジェクタは擬似的にメモリエラーを発生させる事ツールであり，これらの対策手法の評価・検証に有効である．既存のフォールトインジェクタの多くはメモリ上のランダムな位置にエラーを挿入するという特徴がある．たとえば，LLFI では中間コード内のランダムに選ばれた動的命令の結果に対してエラーを挿入するため，どのデータに対してエラーが挿入されるかはランダムである．このようなランダムにエラーを挿入する手法では，特定のエラーケースを意図的に再現することが難しいと言った問題点がある．本研究では，指定したメモリオブジェクトに対して直接エラーを挿入でき，特定のエラーケースを容易に再現できるフォールトインジェクタを提案する．本フォールトインジェクタは memcached を対象として，メモリオブジェクト情報の表示，指定したメモリオブジェクトへの２種類のエラー挿入，エラー挿入後に検証したい処理への遷移などの機能を持つ．また，既存の耐メモリエラー機構である software-based ECC に対して，作成したフォールトインジェクタを適用する事でその有効性の検証も行った．

キーワード：memcached，フォールトインジェクタ，メモリエラー

1. はじめに

メモリエラーは宇宙線による干渉やハードウェアの故障などにより引き起こされるメモリ内のデータを正しく読み取れなくなってしまう障害である．１年の間でデータセンター内のサーバの 32%がメモリエラーによる影響を受け，それらうちの 14%は修正できないエラーであるという研究もある [1]．メモリエラーは，Silent Data Corruption（SDC）と呼ばれる検知できない誤りを引き起こす可能性があり，これによりシステムの信頼性が著しく損なわれてしまう可能性がある．特にメモリ上でデータの管理を行う In-memory key-value store（KVS）はメモリを大量に使用するするため，メモリエラーによる被害が大きい．

メモリエラーに対する対策として，これまで多くの研究がなされてきた．対策手法には大きく分けて，ハードウェア上で対策するもの [2] と，ソフトウェア上で対策するもの [3] [4] [5] の２種類がある．ハードウェア上で実装する対策手法の代表例としては，Error Correcting Code（ECC）メモリを用いる手法 [6] がある．ECC メモリにより一定のビット数まではハードウェアでエラーを検知して訂正することが可能である．また，ソフトウェア上でもメモリエラーに対する対策は可能で，重要なオブジェクトを複製する手法 [7]，ソフトウェア上で ECC を実装する手法 [8] など，様々な手法が提案されてきた．

フォールトインジェクタは，擬似的にメモリエラーを挿入するツールである [9]．これにより，上記に述べたような対策手法を組み込んだシステムのメモリエラー耐性の評価や，脆弱性の発見を通して，堅牢なシステムの構築を支援する．既存のフォールトインジェクタの例としては，LLFI [10] などがある．LLFI は，LLVM でのコンパイル時に生成される中間コード内の動的命令が扱うデータに対してランダムにエラーを挿入する．

LLFI などの既存のフォールトインジェクタの問題点は，フォールトをランダムに挿入するということである．ランダムにエラーを挿入する手法だと，狙ったところにエラーを発生させるのが困難であるという問題がある．開発者側にメモリエラーを発生させたい場所が明確にある場合，既存のランダムにフォールトを挿入する手法では開発効率が悪い．たとえば，特定のメモリオブジェクトにエラーが発生した場合の挙動を検証したいといった場合に，LLFI のようにランダムに選ばれた動的命令の出力結果に対してエラーを挿入する手法では，意図的にその状況を作り出すことが困難である．

そのため，そのメモリオブジェクトにアクセスしそうな

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

ⓒ 2022 Information Processing Society of Japan

命令の選別が必要になったり，繰り返し実行して特定のケースになるのを待つ必要があるため，開発時の検証効率が悪くなってしまう．

そこで，本研究では指定したメモリオブジェクトに対して，直接エラーを挿入できるフォールトインジェクタを提案する．提案するフォールトインジェクタは，指定したメモリオブジェクトに対して直接エラーを挿入することにより，開発者が想定したメモリエラーを効率的に再現することを可能にする．指定したメモリオブジェクトに対するエラー挿入を実現するために，本研究では，In-memory KVS である memcached [11] を対象として，ユーザ側へのメモリ上のオブジェクト情報の表示，オブジェクトへのエラーの挿入という２つの機能を持つフォールトインジェクタを実装した．エラーの挿入については，単純な bit 反転によるエラーと，ECC で修復できないが検知はできるメモリエラーである ECC-uncorrctable Error（UE）の２種類のエラーを再現する．また，効率的な開発支援として，より細かなエラーケースの実現を可能にするために，エラー挿入後に指定された処理を呼び出す機能も加えて実装した．

本研究の貢献は以下の通りである．

- Memcached を対象に，指定したメモリオブジェクトに対して直接エラーを挿入できるフォールトインジェクタを提案した．
- フォールトインジェクタは，オブジェクト情報の表示，指定したオブジェクトへのエラーの挿入，エラー挿入後の任意の処理への遷移機能を持ち，単純な bit-flip によるエラーと Uncorrectable Error の２種類のエラーを再現可能である．
- 既存のメモリエラー対策手法である Software based ECC に対して，実装したフォールトインジェクタを適用しその実用性を検証した．

本論文では，はじめに第２章で本論文の背景，第３章で関連研究，及びこれまでの研究での問題点を示す．次に第４章では，指摘した問題点を改善するための提案手法とその Design Challenge を示し，第５章で提案手法を実現するためのシステム設計，第６章で詳細な実装方法を述べる．第７章では，本研究で設計・実装を行ったフォールトインジェクタの実用性を検証するための実験を行う．最後に，第８章にて本研究のまとめと，今後の展望について述べる．

2. 背景

2.1 メモリエラー

メモリはハードウェアの故障や予期しないビット反転により，保存された値を正しく読み出すことができない場合があり，この障害はメモリエラーと呼ばれる．エラーが生じた領域を利用しているアプリケーションは本来意図したデータとは異なるデータを読み取るため，正常な動作が期待できなくなる．具体的な例として，アドレスデータにエラー生じた場合に引き起こされる不正アドレスへのメモリアクセスによる segmentation Fault や，数値データにエラーが生じる事により引き起こされるゼロ除算等による arithmetic error 等のハードウェア例外処理によるシステムクラッシュのケースや，SDC の発生によりプログラム自体は正常終了するが誤った結果が出力されるケースなどが挙げられる．データセンターのサーバなど大規模なシステムを扱うマシンでのメモリエラーの発生は，誤ったデータがクライアント側に転送されることで，誤りが拡散していく危険性がある．

メモリエラーの発生件数は近年増加傾向にある [12] [13]．考えられる原因として，まずメモリ容量の肥大化が挙げられる．サイズが大きくなれば，当然メモリエラーが発生する確率は高くなる．また，技術の微細化によるトランジスタの性能のばらつき，リーク電流の増加なども原因として挙げられる．

これまで多くのメモリエラーの対策に関する研究・開発がなされてきた．ハードウェア上の対策の１つである ECC メモリを用いた手法 [6] では，パリティ bit を導入することによりメモリ上で発生する複数ビットのエラーの検知と，single ビットのエラー修正を行う事ができる．検知可能だが修正できないエラーは ECC-uncorrectable Error（UE）と呼ばれる．UE は大規模なシステムでは頻繁に発生するという研究も出ており [14] [15] [16]，UE 発生時には通常プログラムは強制終了させられる．また，プログラム上でエラーの検出・訂正を行う sofware-based ECC [8] などのソフトウェアによるメモリエラーの対策手法も提案されている．

2.2 フォールトインジェクタ

フォールトインジェクタは擬似的にメモリエラーを再現するためのツールであり，フォールトトレラントなシステムの構築を支援する [17]．前節で示したようなメモリエラーに対する対策手法を開発する際には，フォールトインジェクタを用いてエラーを再現し，エラー耐性の評価・検証を行う．

フォールトインジェクタには，ハードウェア上で実装するものとソフトウェア上で実装するもの（SWiFI）の２種類があるが，拡張性や移植性に優れているという点から SWiFI [18] [19] [20] が主流となっている．SWiFI では，アプリケーションの実行中に何らかの方法で fault injection コードを実行することでハードウェアエラーをエミュレートする．代表的なフォールトモデルとして，bit-flip モデルがある．これはあるデータの bit をソフトウェア上で反転させることでハードウェア上でのエラーを擬似的に再現するというものである．

ⓒ 2022 Information Processing Society of Japan

2.3 memcached

本研究では，memcached [11] を対象としたフォールトインジェクタを設計する．本節では memcached の概要，内部のデータ構造の仕組み・管理方法を述べる．

2.3.1 memcached の概要

Memcached は，フリーでオープンソースの高性能な分散型の In-memory KVS である．In-memory-KVS は一般的にはデータベースの負荷を軽減して動的な Web アプリケーションを高速化するためのキャッシュサーバとして使用することを目的としている [21] [22] [23] [24]．KVS は Key-Value Store の略であり，キーと値の組み合わせでデータを管理する仕組みのことである．RDB（Relational DataBase）に代わるデータ管理手法である NoSQL の 1 つで近年注目を浴びている．また In-memory 型のシステムとは，データをハードディスクなどには書き込まずにメモリ上で管理する仕組みのことで，通常の RDB などに比べて高速にデータを出し入れできるという特徴がある．複数の memcached サーバを並列に動作させることで分散キャッシュシステムを実現することが可能になっている．

2.3.2 memcached のデータ構造

memcached では，図 1 に示すように slab と呼ばれる単位でメモリ領域を割り当て，順次そこに key-value データを保存していく．slab の管理は slabclass というクラスごとに行う．メモリの割り当ては slab 単位で行うが，実際にメモリ領域を使用する際には更に細かいブロックである chunk と呼ばれる単位に分割して使用される．各 slabclass は独自の chunk サイズを持っている．

memcached において，保存される Key-value データは item と呼ばれる形で保存され，その中身は key と value の値に加えて，item の情報を保持する metadata を加えた 3 ブロックからなる．各 slab における詳細な item の割り当て方は図 2 のようになる．新たな item が割り当てられる際には 3 ブロックの合計サイズを満たすような最小の chunk サイズを持つ slabclass を決定し item を割り当てる．その slabclass が保持する slab 領域に空きがある場合には，free item を参照して item を割り当て，slab を保持していない又は既存の slab に空き chunk が無い場合には，その slabclass に新たに slab を割り当てた後に item を格納する．

memcached で item を削除する際には，メモリ領域は解放されずフラグによって item を無効化し，その領域は free item リストに加えられることで再利用される．

2.3.3 memcached のデータ管理

Memcached はデータベースのキャッシングシステムとして使われるため，メモリに格納できる item が一杯になった時に，いくつかの item をメモリ上から追い出す必要がある．memcached では LRU(Least Recently Used) アルゴリズムを用いて，バックグラウンドスレッドで item の追い出し処理を行う．保存される item にはアクセスされた回数

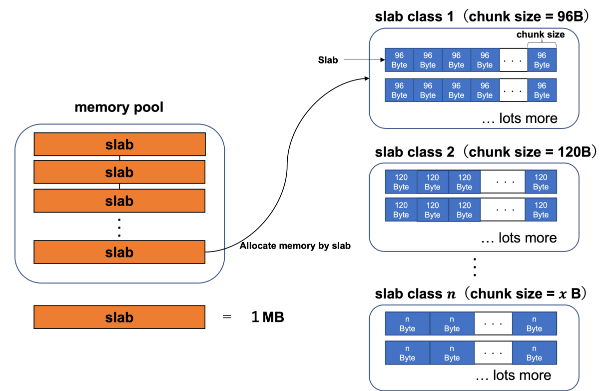

図 1: memcached におけるメモリ領域の割当

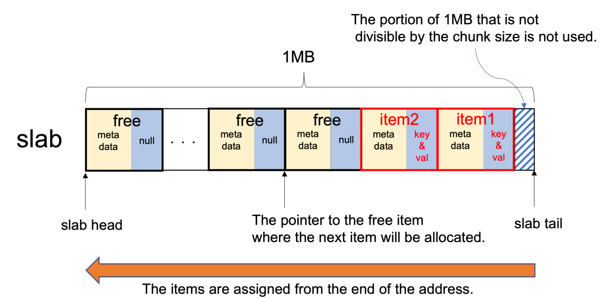

図 2: memcached における各 slab 内での item の割当

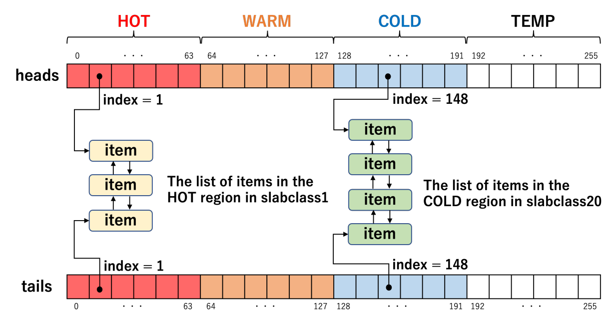

図 3: LRU リストによる item の状態管理

に応じて HOT，WARM，COLD の 3 つの状態が与えられ，キャッシュから evict される item は COLD 状態の item の中から選ばれる．memcached では LRU リストと呼ばれるデータ構造を用いて，図 3 に示すように item の状態ごと・slabclass ごとにリストを形成することで管理している．

また，memcached では Hash テーブルを用いていることで，item の高速な検索を実現している．item の読み出しを依頼された場合には，key の hash 値を元に検索を行う．各 item はハッシュ用のポインタ（h_next）を metadata 領域内に保持しており，ハッシュ値が重複した場合は item はリスト構造をとるチェイン法を用いて管理される．

3. 関連研究

3.1 既存のフォールトインジェクタ：LLFI

LLFI は，LLVM を用いたフォールトインジェクタである [10]．LLVM [25] は C/C++を含む複数の言語で書かれた

ⓒ 2022 Information Processing Society of Japan

ソースコードの解析と最適化を可能にするコンパイラツールであり，コンパイル時に LLVM IR と呼ばれる中間コードを生成する．LLFI では，この中間コードである LLVM IR に対してフォールトインジェクションを行う．

具体的なエラー挿入のアルゴリズムについて説明する．LLFI でのエラー挿入はメモリに対して直接行われるのではなく，命令実行後の結果が格納されているデスティネーションレジスタの値を bit-flip することにより行われる．どの命令の実行結果に対してエラーを挿入するかはランダムで，LLVM IR 内で実行される全ての動的命令の中からランダムに選ばれた命令の実行結果に bit-flip を発生させることでエラーを挿入する．

3.2 問題点

LLFI は中間コードである LLVM IR 上で動作するため，アプリケーションの挙動に沿ったエラーを挿入することが可能であるが，課題として特定のメモリオブジェクトに対して，意図的にエラーを挿入することが困難であるという点が挙げられる．中間コード内のランダムに選択した動的命令の結果に対してエラーを挿入するため，エラーが挿入されるメモリオブジェクトもランダムになってしまう．このランダムにエラーを挿入する方式は，ソフトウェア開発者が特定の箇所にエラーを挿入してその挙動を検証したい場合には，狙ってエラーを挿入できるわけではないため検証の効率が悪いという欠点がある．

4. 提案

本研究では，memcached を対象とした開発者がより効率的なエラー挿入を行うことが可能なフォールトインジェクタを提案する．提案手法では，memcached 内のデータオブジェクト情報を開発者側に提供し，それらを元に指定されたオブジェクトに対してエラー挿入を行うことで，開発者の意図に応じた円滑なエラーの再現を実現する．

4.1 エラー挿入の対象となるメモリオブジェクト

本研究で提案するフォールトインジェクタでは，item やそれらを管理するメタデータなどのヒープ領域で動的に確保されるメモリオブジェクトを対象にメモリエラーの挿入を行う．理由としては，memcached はキャッシュサーバであるため，メモリ上の大半を占めるデータはヒープ領域で管理される key-value データである．そのため，生じるエラーの多くもこれらのメモリオブジェクト上で発生すると考えられるため，本研究ではメモリオブジェクトを対象にメモリエラーの挿入を行う．

4.2 フォールトインジェクタの機能

従来のフォールトインジェクタとは異なり，本研究では，指定したメモリオブジェクトに対して狙ってエラーを挿入

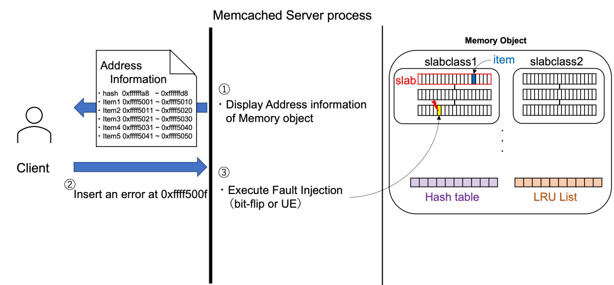

図 4: 提案するフォールトインジェクタの概要

できる機能を持つフォールトインジェクタを提案する．本研究で提案するフォールトインジェクタの概要を図 4 に示す．エラーを挿入する処理は大きく分けて以下の 2 種類に分かれる．

- メモリオブジェクト情報の出力
- 指定したメモリオブジェクトへのエラー挿入

指定したメモリオブジェクトへ挿入されるエラーについては，ECC メモリによる保護機能を持たないシステムと持つシステムの両者におけるエラーを再現するために，以下の 2 種類のエラーを挿入可能にする．

- ソフトウェア上での単純な bit 反転 (bit-flip 型)
- ECC メモリで修正できない UE（UE 型）

また円滑なエラー検証を補助するために，bit-flip 型のエラーを挿入する場合にはエラー挿入後に任意の処理を実行することができる機能を実現する．この機能により，エラーデータがどの処理によってアクセスされるかを開発者が指定することができるため，検証したいケースを容易に再現することが可能になる．

4.3 デザインチャレンジ

提案したフォールトインジェクタを実装する上で，設計課題となる点は主に以下の 3 つである．

- **メモリオブジェクト情報の取得**
  エラーの挿入に必要な情報として，メモリオブジェクトのアドレス情報などを開発者側に提供する必要がある．その際に，存在する膨大なメモリオブジェクトの中から，どのようにして開発者側の目的に沿ったオブジェクトのみの情報を出力するのかといった問題がある．この問題に対しては，開発者の目的に応じた item へのアクセス方式を 3 通り用意することで解決する．
- ***UE* の再現**
  UE による SIGNAL の送信などの処理は bit 反転が生じた際に起こるのではなく，プログラムがエラーデータを読み取った際に発生する．そのため，UE を再現する際に課題になってくるのが，どのようにプロセスのメモリアクセスを追跡し，エラーデータへのアクセスの有無を確認するかという点である．この課題に対しては，Dynamic Binary Instrumentation（DBI）の一種

である Intel Pin [26] を用いることで解決する．DBI は実行時にバイナリの内容を動的に書き換えることで，CPU 命令単位でのプログラム実行のトレースなどを行う手法であり，これを用いることで memcached のメモリアクセス処理をフックすることが可能になる．

- **エラー挿入後の処理遷移**
  本研究では，開発者が想定したエラーケースを効率よく検証可能にするため，エラー挿入直後に，検証したいケースに応じた処理を実行するという機能を実現する．この機能を実現するにあたり課題になるのが，他のスレッドからの干渉をどう制御するかという点である．Memcached はマルチスレッドで動作するため，エラー item に対して指定した処理を実行したい場合に，他のバックグラウンドスレッドなどによって先にエラー item へアクセスされてしまう可能性がある．本研究では item へのアクセスが頻繁に行われるバックグラウンドスレッドの特定と，それらをフォールトインジェクション時には停止させることでこれらのスレッドによる干渉を制御する．また，エラー挿入後にこれらのスレッドによる処理を検証したい場合には，メインスレッド上でこれらの処理を実行することで，他スレッドの影響を排除しながらエラー発生時の特定処理の挙動の検証を可能にする．

5. 設計

5.1 エラー挿入の指定方法と流れ

本研究で提案するフォールトインジェクタでは，クライアントからコマンドを通してフォールトインジェクションの処理依頼を行い，コマンドを受け取ったサーバ側で実際にエラーの挿入処理を行うという操作形式を採用した．このような仕様にする利点として，インタラクティブなデバッグが可能になるということがあげられる．サーバプロセスとクライアントプロセスを同時に立ち上げながら，インタラクティブにフォールトインジェクションを行うことができるため，リアルタイムにシステムの挙動を確認しながらデバッグ作業を行うことが可能である．

実際のフォールトインジェクションの流れを図 5 に示す．まず，memcached 上で現在有効なメモリオブジェクトの情報を取得するためにクライアント側からメモリオブジェクトの情報を取得するコマンドを実行する．コマンドを受け取ったサーバは，メモリオブジェクトに関するアドレス情報をクライアントに返す．ユーザは，その情報を元にクライアント側から，エラーを挿入するオブジェクトのアドレスを引数としてエラー挿入コマンドを実行する．再びコマンドを受け取ったサーバは，指定されたアドレスに対してメモリエラーを挿入する処理を行うといった流れで全体の処理が行われる．

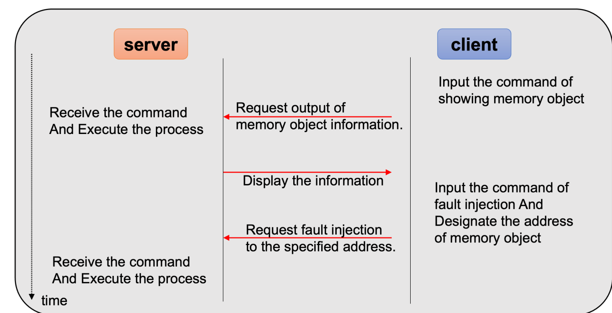

図 5: フォールトインジェクション時の処理の流れ

5.2 メモリオブジェクト情報の出力

本節ではエラー挿入前のメモリオブジェクトの情報を出力する段階の処理についての詳細を述べていく．

5.2.1 出力する情報

サーバはオブジェクト情報の出力コマンドを受け取ったら，オブジェクトのアドレスを出力する．提案手法では，fault injection の際にアドレスを指定することで，そのアドレス内のデータに対してエラー挿入を行う．したがって，ユーザ側にはどのメモリオブジェクトがアドレス空間のどこに位置しているかの情報を出力する必要がある．4 章 1 節でも述べたように，本研究ではエラーを挿入する対象を，item とそれらを管理するデータ構造の 2 種類に絞っている．その各々についてどのような形式・内容でアドレス情報を出力するのかを以下に示す．

- *item*
  item の中身は，item を管理する metadata，key データ，value データの 3 ブロックから構成される．item の中でも，3 ブロックのうちのどの部分にエラーを挿入するのかを選択可能にするため，これらのブロックごとのアドレス領域は別々に出力する．また，meta data ブロックの中はより細かなデータブロックに分割されている．たとえば，LRU リストにおける当 item の next や prev につながる item へのポインタに関する情報や，ハッシュリストのポインタ，TTL，データサイズなどの情報である．従ってより詳細な metadata 情報が必要な場合には、meta data ブロックの中身は保持するデータごとにアドレスを分けて出力する．
- **データ管理オブジェクト**
  item を管理する LRU リストや Hash リスト，slabclass を管理する slabclass array といったデータ管理オブジェクトは，item へのポインタをデータとして持つ配列である．配列のアドレス範囲，データ数，データのサイズを出力する．

5.2.2 出力方式

メモリオブジェクト情報の出力は，slab，LRU リスト，Hash リストの 3 つのデータ管理オブジェクトで管理している item ごとにアドレス情報を表示する設計を施した．図 6 に示すように，memcached において開発者が特定の item

ⓒ 2022 Information Processing Society of Japan

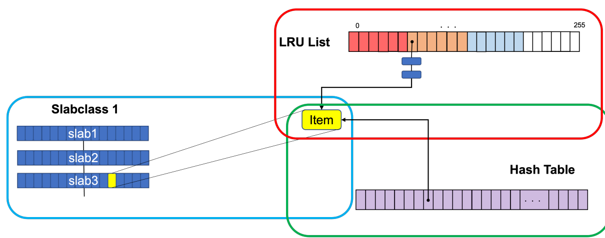

図 6: memcached 上での item の参照

を参照する際にはいくつかのアプローチが存在する．開発者がエラーを挿入する際にはその目的に応じて，これらの複数のアプローチから各 item のオブジェクト情報を参照できると非常に使い勝手が良いと考えられる．たとえば，LRU リストの先頭に位置している item や，同じハッシュ値をもつ item が２つ以上存在する item に対してエラーの挿入を行いたいと言った場合に，それぞれ異なるアプローチで item を検索できると非常に便利である．

したがって，上記の３つの管理オブジェクトからそれぞれ item を検索して出力する方式を実装した．各出力方式の詳細を以下に述べる．

- ***slab* から出力**
  slabclass と slab を指定し，その slab 内に格納されている item のアドレス情報を一覧で出力する．
- ***LRU* リストから出力**
  slabclass と item の状態を指定し，その slabclass 内に存在している指定した状態の item の一覧を LRU リストを探索して出力する．
- ***Hash* テーブルから出力**
  Hash 値の範囲を指定し，その範囲内のハッシュ値を持つ item を全て出力する．

これらの３つの出力方式において，各 item のアドレス情報を詳細に表示すると，item の量によっては出力する情報量が非常に大量になってしまう可能性があり，開発者が必要な情報を探す手間が増えてしまう恐れがある．そこで，これらの方式で item に対してアクセスした場合には，metadata ブロック，key ブロック，value ブロックの３ブロックのアドレス範囲を簡易的に出力し，metadata 領域の詳細な情報は出力せず，metadata 領域内の各データフィールドのアドレス情報の詳細を取得するための出力方式として，直接 item を指定して出力する方式も加えて設計した．

- ***item* を直接指定して出力**
  item の key を指定することで，key ブロック，value ブロックの情報に加えて、metadata ブロック内のデータごとのさらに細かいアドレス情報を出力する．

5.3 メモリエラーの挿入（bit-flip 型）

bit-flip 型のエラー挿入では，クライアント端末からコマンド入力を通してエラーを挿入するアドレスと，そのアドレス内のデータの何 bit 目にエラーを挿入するかを指定する．サーバ側では，クライアントからのコマンド引数で受け取ったアドレスに格納されているデータの，指定された bit に対して bit 反転を行う．bit 反転は単純な bit 演算を用いて行う．

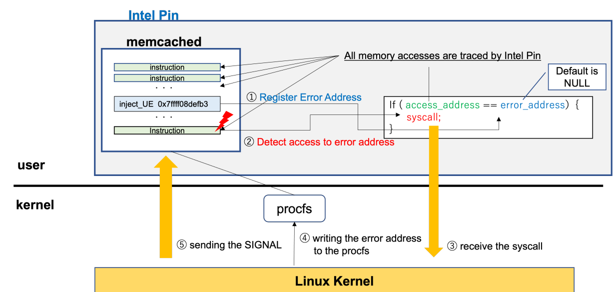

図 7: UE 型のエラー挿入時の概要図

5.4 メモリエラーの挿入（UE 型）

まず，通常の UE 発生時の挙動について説明する．UE は ECC メモリで修正不可能な破損データにアクセスした場合に発生し，UE を検知したカーネルは破損データのアドレスからその領域を使用しているプロセスを特定し，通常は kill シグナルを送ることでプロセスを強制終了する．

そこで，UE 型のエラー挿入ではプロセスのメモリアクセスを監視し，事前に指定しておいた破損データにアクセスした場合にカーネル空間から特定の SIGNAL をプロセスに対して送信することで UE の挙動を再現する．memcached, Intel pin，Linux Kernel でのエラー挿入時の役割と挙動を図 7 に示す．

5.4.1 intel pin 内での処理

メモリアクセスの監視には Intel Pin を用いる．Intel Pin により memcached 内で実行される命令全ての処理をフックすることで，メモリアクセスを監視する事が可能になる．Intel Pin は memcached 内で UE 型の injection コマンドが実行されたことを検知すると，指定されたアドレスを error address として登録しておく．それ以降の処理では，毎命令ごとにオペランドのアドレスと登録した error address を比較することで，エラー領域へのアクセスを監視する．そしてエラーアドレスへのアクセスが行われた場合には，専用のシステムコールを呼び出すことでカーネルに処理を移行する．

5.4.2 linux Kernel 内での処理

システムコールを受け取った Linux Kernel 内での処理は，procfs へのエラーアドレス書き込み，プロセスへの SIGNAL 送信の２つである．

本来であれば，エラーデータを保持しているプロセスに対して，kill SIGNAL を送信すれば UE 時の挙動を再現することはできている．しかし，本研究で提案するフォールトインジェクタはメモリエラー耐性を保持する堅牢なシステムの開発を支援するためのものであるため，UE 発生時

の挙動を再現するとともに，耐性機構の開発を円滑に進めるための支援機能が必要となる．支援機能として procfs へのエラーアドレス書き込み処理を組み込むことで，プロセスに対してどのアドレスが破損したのかを伝える事が可能になり，UE によるクラッシュ耐性を保持するような機構の開発支援を実現する．

5.5 エラー挿入後の処理遷移

本研究で提案するフォールトインジェクタは，意図的に多様なエラーケースの検証を可能にするためエラー挿入後に呼び出したい処理へ遷移するという機能も持つ．この機能は，bit-flip 型のエラー挿入と併せてのみ使用することが可能となる．遷移対象の処理，実現方法について説明していく．

5.5.1 遷移の対象とする処理

memcached 上で稼働する lru_maintainer と lru_crawler の２種類のバックグラウンドスレッドを, エラー挿入後の遷移対象となる処理とする．まず, get や append などのコマンドによる item へのアクセスケースについては，fault injection の実行後にクライアント側からコマンドを入力することで再現可能であるため，対象としない．また，memcached 上で稼働するバックグラウンドスレッドの内，slab の再配置を行うものや，hash テーブルのサイズを拡張するものなど，item へのアクセスを行わない処理についても対象としない．

lru_maintainer は item の状態の確認・更新のため恒常的に活動しており, lru_crawler も TTL の切れた item をキャッシュから削除するためにバックグラウンドスレッド内で定期的に処理を行っているため，本機能における遷移対象の処理とする．

5.5.2 処理遷移の実現方法

まず，遷移の対象となる lru_maintainer と lru_crawler の２つのバックグラウンドスレッドはコマンドにより停止できるようにしておく．この２つのバックグラウンドを停止しておく理由は，コマンド処理によるエラー item へのアクセスの検証を容易にするためである．具体的には，エラー挿入後に get や append などのコマンド処理によるエラー item へのアクセス時の挙動を検証したい場合に，これらのバックグラウンドスレッドが稼働していると，スレッドの処理によってエラー item に先にアクセスしてしまう可能性がある．そのため，必要に応じてバックグラウンドスレッドを停止しておくことで，スレッド処理が干渉することなくコマンド処理によるエラー item へのアクセス時の挙動の検証を行うことが可能になる．

lru_maintainer と lru_crawler によるエラー item へのアクセスを検証したい場合には，メインスレッド上で fault injection を行った直後にそのままこれらの処理へ遷移することで実現する．バックグラウンドスレッドを起動するのではなくメインスレッド上で処理を遷移することで，並列処理に伴う他のスレッドからの影響を排除して，これらの処理をエラー item に対して実行した時の挙動を検証することが可能である．fault injection により呼び出された場合は自ら return するようにスレッド関数の中身を修正することで，メインスレッドの正規の処理への復帰も実現する．

6. 実装

提案手法は memcached-1.6.9，Intel Pin3.18，Linux5.10.0 のソースコードに改良を加えることで実装した．フォールトインジェクタの実装の詳細について，ユーザインタフェース部，メモリオブジェクトの出力部，エラー挿入部（bit-flip 型），エラー挿入部（UE 型），処理遷移部の５つに分けて説明する．

6.1 ユーザインタフェース部

ユーザ側から処理依頼を行うためのコマンドは，アイテムオブジェクト情報の出力を行うコマンドとエラー挿入の依頼を行うコマンドの２種類である．アイテムオブジェクト情報の出力を行うコマンドは，item に対して slabclass, LRU リスト，Hash テーブル，直接指定のどれからアクセスするかによって以下の４種類のコマンドが存在する．

- show_slab_addr
- show_lru_addr
- show_hash_addr
- show_item_addr

各出力方式における item の出力範囲は，コマンド引数内で指定する事が可能である．

また，エラーの挿入を行うコマンドは，bit-flip 型のエラーを挿入する injection，UE 型のエラーを挿入する injection_UE というコマンドをそれぞれ用意した. bit-flip 型の injection については，エラー挿入アドレスに加えてデータの何 bit 目にエラーを挿入するかも引数により指定することができる．

6.2 メモリオブジェクト出力部

メモリオブジェクト情報を取得する処理は大きく分けて，指定された条件に該当する item を探索する処理と，該当する item の中身をブロックごとに分割して出力する処理の２つに分割できる．前者の処理は，slabclass，LRU リスト，Hash テーブル，直接指定の４種類からのアプローチによってそれぞれ異なる．後者の処理は，簡易的な情報を出力する場合と，metadata 領域の中身を含む詳細な情報を出力する場合の２通りがあるが，基本的な実装は同じである．

6.2.1 該当 item の探索処理

slabclass，LRU リスト，Hash テーブル，直接指定の各アプローチごとの探索方法を以下に示す．

- *slabclass*

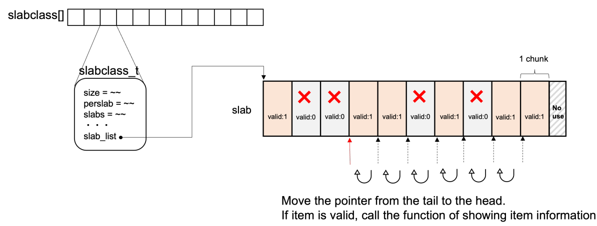

図 8: slabclass からの item の探索

図 8 に示すように，コマンド引数で指定された slabclass の slab を特定し，その slab の末尾の chunk から先頭の chunk までを順に調べていき，有効な item のみを取得する．削除コマンドなどにより全ての item が連続した chunk に格納されている訳ではないため，先頭から末尾までの slab 内の全ての chunk を調べる必要がある．

- ***LRU* リスト**
  LRU リストは各 slabclass・状態ごとのリストの先頭と末尾を保持しておく heads と tails という 2 つの配列から構成されているため，これらの配列を参照して該当リストを順に見ていくことで item を探索する．
- ***Hash* テーブル**
  ハッシュテーブルの指定されたハッシュ値の範囲部分を探索することで item を探索する．
- **直接指定**
  get コマンド時の動作と同様にして，指定された key から Hash テーブルを参照して該当 item を見つける．

6.2.2 item 内のアドレス情報の出力処理

この処理では，引数として item 領域の先頭アドレスを受け取り，その中身をデータの種類毎に出力する処理を行う．データの出力は，基本的には metadata 領域，key データ領域，value データの領域の 3 種類のアドレス範囲を出力する．詳細な情報を出力する場合には、metadata 領域内の役割毎の細かなデータブロックも分割して出力する．

key データと value データのアドレス範囲の参照は，これらのデータの先頭アドレスと，その大きさを参照することにより算出する．先頭アドレスは既存のマクロである ITEM_key() と ITEM_data() を用いることで取得でき，データサイズについては，item の構造体のフィールドを参照することで取得可能である．

詳細なアドレス情報を取得する際の，metadata 領域のアドレス範囲の参照は，図 9 に示すように，metadata 内の各データ領域のサイズを事前に保持しておくことで先頭アドレスから算出を行う．metadata 領域には，item を管理する構造体のデータが格納されているため，そのサイズと各フィールドのデータの配置は全ての item で共通である．従って，各フィールドのデータサイズを事前に配列で保持しておくことで，metadata 領域の先頭アドレスから各フィールドのアドレス範囲を算出することが可能になる．

前項で述べた item の探索処理の中で発見した item に対して，この関数を呼び出すことにより，指定された条件下での item 一覧とそれらの item の詳細なアドレス情報をユーザに提供する処理が実現される．

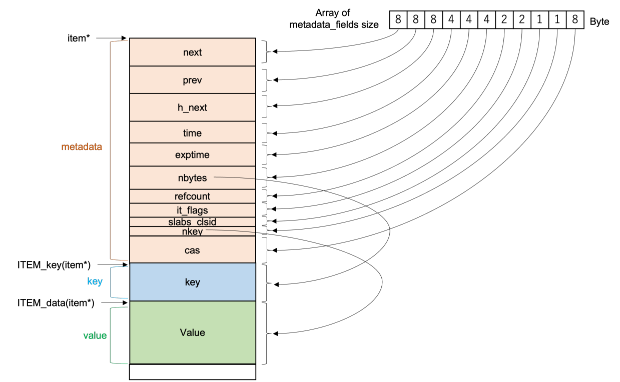

図 9: item 内の各データブロックへのアクセス

6.3 エラー挿入部（bit-flip 型）

bit-flip 型のエラー挿入部は，指定されたアドレスにエラーを挿入する関数を実装した．この処理の実装は単純で，コマンドの引数として与えられたアドレスに格納されるデータの指定された bit に対して，bit-flip を行いエラーを挿入する．具体的には，指定されたアドレス内のデータを char 型で取得し，そのデータに対して bit 演算を行うことでエラーを発生させる．

6.4 エラー挿入部（UE 型）

UE 型のエラー挿入は，memcached，Intel Pin，Linux Kernel の 3 層からなるため，各層における実装を説明していく．

6.4.1 memcached 上の実装

memcached 上では，inject_UE コマンドを受け取ると，エラーアドレスへのポインタを引数として fault_inject_UE と呼ばれる関数を呼び出す．fault_inject_UE は，Intel Pin がエラーアドレスが登録されたことを検知するために使用する関数であるため，関数内で特定の処理をするわけではない．詳細は次節の Intel Pin 上の実装で述べる．

6.4.2 Intel Pin 上の実装

Intel Pin で提供される API を利用したエラーアドレスの登録，メモリアクセスの追跡の 2 つの処理の実装を説明していく．

- **エラーアドレスの登録**
  エラーアドレスの登録には，Intel Pin が提供する RTN_InsertCall と呼ばれる関数呼び出しをフックする機能を持つ API を用いる．この API では指定した関数名を持つ関数が実行された瞬間に，任意の処理を実行することが可能である．Intel Pin は memcached 上で fault_inject_UE が呼び出された事を検知すると，その関数の引数で渡されているアドレスデータを取得し，

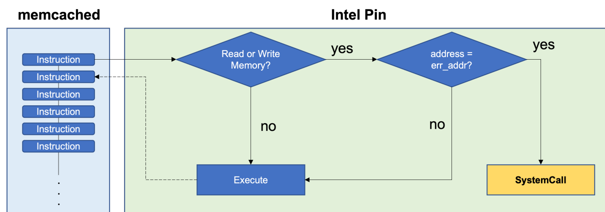

図 10: Intel Pin によるメモリアクセスの追跡処理

エラーアドレスとして Intel Pin 内のグローバル領域の変数（err_addr）に保存しておく．

- **メモリアクセスの追跡**
  簡略化したメモリアクセス追跡の全体の処理の流れを図 10 に示す．メモリアクセスには，Intel Pin が提供する，(1)INS_MemoryOperandIsRead, (2)INS_MemoryOperandIsWritten, (3)INS_InsertPredicatedCall の 3 つの API を主に使う．(1) と (2) は実行命令がメモリに対して読み込み，書き込みを行っているかどうかを確認する API であり，(3) は命令が実行される前に任意の処理を挿入するための API である．処理の流れとしては，まず memcached 上で実行される全ての命令に対して (1)(2) の API でメモリアクセスの有無を確認する．メモリアクセスがある場合には (3) の API によって，自作した checkMemRead と checkMemWrite と呼ばれる関数を呼び出す．CheckMemRead と CheckMemWrite はアクセスを行うアドレスとグローバル変数の err_addr を比較し，一致する場合にはエラーデータへのアクセスとして systemcall を呼び出すという処理を実行する．

6.4.3 Linux Kernel 上の実装

Linux Kernel 上には新たに raise_error という systemcall を実装した．raise_error は Intel Pin がエラーを検知した場合にエラーアドレスを引数として呼び出され，procfs へのエラーアドレス書き込み，プロセスへの SIGNAL 送信を行う．procfs への書き込み処理は，task 構造体に新たに err_addr というフィールドを追加し，procfs 内の stat ファイルへの書き出し処理部分に新たに err_addr の中身を出力する処理を加えることで実装する．プロセスへの SIGNAL 送信については send_sig 命令によりプロセスへ SIGNAL を送信する．

6.5 処理遷移部

6.5.1 バックグラウンドスレッドの停止

lru_maintainer と lru_crawler の 2 つのバックグラウンドスレッドの稼働を制御をするに当たり，lru_crawler は memcached に標準で搭載される lru_crawler コマンドによってスレッドの停止・再開を制御する事ができる．lru_maintainer はそのようなコマンドが用意されていないため，lru_crawler コマンドと同様の形式で，スレッドを停止・再開できるコマンドを実装した．

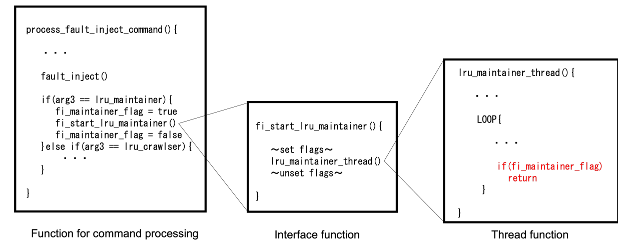

図 11: 処理遷移用の関数呼び出しの流れ

これらのコマンドを使う事で，クライアント側からバックグラウンドスレッドの制御を行う事が可能になる．fault injection を行う前にこれらのコマンドを用いてスレッドを停止する事で，スレッドによる干渉をされずに望んだエラーケースの再現が可能になる．

6.5.2 処理移行のためのインタフェース

エラー挿入後に，lru_maintainer と lru_crawler をメインスレッド上で実行するためのインタフェースとなる fi_start_lru_maintainer と fi_start_lru_crawler という 2 つの関数を実装した．これらの関数では主に，スレッド処理の中で使用されるフラグの設定などを行った後，メインの処理を実行する関数を呼び出す．

これらの関数は，クライアントからの inject コマンドを処理する関数の中で呼び出され，エラー挿入が実行された直後にこれらの関数が呼び出される．

6.5.3 関数の終了

メインスレッド上で指定された処理に移行した後に，その処理からメインの処理に自ら帰ってこれるようにスレッド関数に修正を加えた．lru_maintainer と lru_crawler のメインの処理を行う関数である lru_maintainer() と lru_crawler() は，元々新たに生成されたスレッドにより呼び出される関数のため他のスレッドからのシグナルなどがない場合には基本的に関数を終了せず，ループ分の中で繰り返し処理を行う．

メインスレッド上でこれらの処理を実行するためには，自ら関数を終了して戻ってくるように関数を修正する必要がある．そこで,injection のコマンドによって呼び出された場合には，一連の処理を終えた後にループには入らずに return するように関数を修正した．具体的には，新たに fi_maintainer_flag，fi_crawler_flag と呼ばれるフラグを定義し，これらが true だった場合，すなわち inject 命令により呼び出された場合にはループに入らず処理をメインに返す挙動をするように関数を修正した．

injection コマンドを受け取ってからの一連の関数の呼び出しをまとめると図 11 のようになる．

7. 実験

本章では，提案手法に対する評価・検証を行う．実験は，

指定したメモリオブジェクトへのエラー挿入の検証と，既存のメモリエラー耐性機構を実装したシステム上での動作チェックの２種類の検証を行う．本実験では改良を加えたmemcached-1.6.9 を Ubuntu20.04 上で動作させた．

7.1 実験１：指定オブジェクトへの Error 挿入

7.1.1 実験準備

実験１では作成したフォールトインジェクタをシステム上で実際に動かし，その挙動を確認する．実験方法としては 500 個の item を格納した状態の memcached に対して，item 情報の出力と，item へのエラー挿入を複数のケースで実行し，その挙動を確認する．500 個の item の生成には、Memcached クライアント用の python のパッケージである pymemcache を用いる．item を生成した状態で，サーバに接続したクライアント端末から実際の操作を行い，フォールトインジェクション用のコマンドを実行していき，その挙動を確認する．具体的には，４つの item 出力用のコマンドを実行した後，item の value 領域，metadata ブロック内の next 領域，exptime 領域にエラーを挿入したケースの挙動を確認する．

7.1.2 実験結果

まず，生成した item の情報を出力するため，４つの show コマンドを実行した様子を図 12 に示す．サイズの都合上，上位 2item の出力結果のみを切り出している．各コマンドから item のアドレス情報が出力できているのが確認できる．また，データ管理オブジェクトからアクセスする 3 つのコマンドによる出力結果では、先頭に管理オブジェクトのアドレス範囲も出力されており，show_item_addr では，meta データ領域の情報が詳細に出力されているのが確認できる．

出力された情報を元に，bit-flip 型のエラーを挿入した際の出力結果を図 13 に示す．ここでは，item の value 領域にエラーを挿入した場合の出力結果を示す．図 13 からもわかるように，inject コマンドにより value の中身が”value”から”walue”となっているのが確認できた．item の metadata ブロック内の next 領域や exptime 領域などにも同様にしてエラーを挿入したところ，領域外参照によるクラッシュや期限切れの item が evict されないなど，挿入されたエラーが正常に作動しているのが確認できた．

inject_UE コマンドによる UE 型のエラー挿入を行った場合にも，get コマンドなどによるエラー item へのアクセスが検知され，memcached に対してカーネル空間から SIGNAL が送信されているのが確認できた．

7.2 実験２：Memory Error 耐性機構の動作チェック

7.2.1 実験準備

実験２では，既存のメモリエラー耐性機構上で提案手法が正しく動作し，従来のフォールトインジェクタでは再現が困難だったエラーケースの再現・検証が可能であるかどうかの確認を行うことを目的とする．

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
slabclass_array :
        Address : 0x55f817babb00 ~ 0x55f817bac4ff , element size:  40 Byte , number of elements :    64
slab class   1: chunk size =>    96 perslab => 10922 slabs =>  1
--------------------------------------------------------------------------------
slab1 : 0x7f9871e08010 ~ 0x7f9871f0800f
key1 =>
        meta : 0x7f9871f07f70 ~ 0x7f9871f07fa0, key : 0x7f9871f07fa8 ~ 0x7f9871f07fac, val : 0x7f9871f07fad ~ 0x7f9871f07fb5,
key2 =>
        meta : 0x7f9871f07f10 ~ 0x7f9871f07f40, key : 0x7f9871f07f48 ~ 0x7f9871f07f4c, val : 0x7f9871f07f4d ~ 0x7f9871f07f55,
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
LRU_heads_List :
        Address : 0x55f817bb8a40 ~ 0x55f817bb923f , element size:   8 Byte , number of elements :   256
LRU_tails_List :
        Address : 0x55f817bb8240 ~ 0x55f817bb8a3f , element size:   8 Byte , number of elements :   256
--------------------------------------------------------------------------------
slabclss 1 : COLD List
key500 =>
        meta : 0x7f9871efc450 ~ 0x7f9871efc480, key : 0x7f9871efc488 ~ 0x7f9871efc48e, val : 0x7f9871efc48f ~ 0x7f9871efc499,
key499 =>
        meta : 0x7f9871efc4b0 ~ 0x7f9871efc4e0, key : 0x7f9871efc4e8 ~ 0x7f9871efc4ee, val : 0x7f9871efc4ef ~ 0x7f9871efc4f9,
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Hash_table :
        Address : 0x7f9875fe0010 ~ 0x7f987606000f , element size:   8 Byte , number of elements : 65536
--------------------------------------------------------------------------------
hash_value : 4
key273 =>
        meta : 0x7f9871f01970 ~ 0x7f9871f019a0, key : 0x7f9871f019a8 ~ 0x7f9871f019ae, val : 0x7f9871f019af ~ 0x7f9871f019b9,
hash_value : 190
key387 =>
        meta : 0x7f9871efeeb0 ~ 0x7f9871efeee0, key : 0x7f9871efeee8 ~ 0x7f9871efeeee, val : 0x7f9871efeeef ~ 0x7f9871efeef9,
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
key1 details=>
        key : 0x7f9871f07fa8 ~ 0x7f9871f07fac, val : 0x7f9871f07fad ~ 0x7f9871f07fb5,
        meta :
                  next => 0x7f9871f07f70 ~ 0x7f9871f07f77
                  prev => 0x7f9871f07f78 ~ 0x7f9871f07f7f
                h_next => 0x7f9871f07f80 ~ 0x7f9871f07f87
                  time => 0x7f9871f07f88 ~ 0x7f9871f07f8b
               exptime => 0x7f9871f07f8c ~ 0x7f9871f07f8f
                nbytes => 0x7f9871f07f90 ~ 0x7f9871f07f93
              refcount => 0x7f9871f07f94 ~ 0x7f9871f07f95
              it_flags => 0x7f9871f07f96 ~ 0x7f9871f07f97
           slabs_clsid => 0x7f9871f07f98 ~ 0x7f9871f07f98
                  nkey => 0x7f9871f07f99 ~ 0x7f9871f07f99
                   cas => 0x7f9871f07fa0 ~ 0x7f9871f07fa7
```

図 12: show_slab_addr，show_lru_addr，show_hash_addr，show_item_addr の出力結果

```
get key1
VALUE key1 0 6
value1
END
inject 0x7f9871f07fad 1
get key1
VALUE key1 0 6
walue1
END
```

図 13: value 領域にエラーを挿入した結果

実験方法としては既存のメモリエラー耐性機構として Software-based ECC [8] を Memcached 上に実装する．

Software-based ECC では，Memcached 上の全ての item データに対して bch 符号化をソフトウェア上で適用することでエラーの検知・修正を行う．具体的には，item の生成時，更新時に bch 符号化によりパリティブロックを生成し，item にアクセスする際には毎回 bch 符号化によりパリティブロックの比較を行うことでエラーの有無を確認し，エラーが発生している場合にはシンドロームの再計算によりエラーブロックを修正する．

Software-based ECC を Memcached 上に実装し，1000 個の item を格納した状態でプログラムを実行し，挿入したエラーが耐性機構で正しく検知・修正されているかを検証した．実験はフォールトインジェクタにより以下の３つのエラーケースを再現し，システムの挙動を観察する．

- **コマンドからエラー *item* にアクセスする場合**
  inject コマンドにより特定の item の value 領域にエラーを挿入した後に，get コマンドを用いてその item へのアクセスを行う．

- ***lru_maintainer* からエラー *item* にアクセスする場合**
  inject コマンドで, アドレスの指定とエラー挿入後の処理として lru_maintainer を指定することで，lru_maintainer 内からエラー item へのアクセスを行う．
- ***lru_crawler* からエラー *item* にアクセスする場合**
  inject コマンドで，アドレスの指定とエラー挿入後の処理として lru_crawler を指定することで，lru_crawler 内からエラー item へのアクセスを行う．

7.2.2 実験結果

３つのケースにおけるエラー item へのアクセス時の，それぞれの挙動を示す．

- **コマンドからのアクセス**
  エラー挿入後にエラー item を get コマンドで取得したところ，挿入されたエラーが検出・修正されて正しい元の値が出力されるのが確認できた．実行時の様子をデバッグツールを用いて確認すると，get コマンドを受理した後に Hash テーブルを検索し，Hash リスト内の item を探索する際に Software-based ECC のパリティブロック演算によってエラーが検知・修正されているのが確認できた．これにより，実験１に示したような誤った値が出力されることなく，格納時の正しい value が出力された．
- ***lru_maintainer* からのアクセス**
  エラー挿入後に lru_maintainer の処理を実行したところ，エラーを挿入する item によってアクセスされるものとされないものが存在した．これは，lru_maintainer スレッドは全ての item を精査しているわけではなく，tails 配列が直接参照している item の状態のみを毎秒参照しているため，LRU リストの末尾以外の item は lru_maintainer ではアクセスされないのが理由であると考えられる．逆に，tails 配列が直接参照している item に対して inject を実行した際には，Software-based ECC により正しくエラーが検知・修正されているのが確認できた．
- ***lru_crawler* からのアクセス**
  エラー挿入直後に lru_crawler を実行したところ, crawler の処理の中でエラー item を検知・修正できているのが確認できた．lru_crawler は格納されている全ての item に対して TTL が切れているかどうかのチェックを行う．そのため，lru_maintainer と違い，全ての item に対してアクセスを行うため，アクセス時にパリティブロックの再計算によりエラー item が検知・修正される．

上記の結果からも分かるように，挿入されたエラーはエラー耐性機構により正しくエラーと認識され，修正されている事が確認できた．従って，作成したフォールトインジェクタは，実際にこれらの機構の開発時にメモリエラー検証ツールとして使用する事が可能であると言える．

また３つのエラーケースの結果に示すように，従来のフォールトインジェクタでは実現が困難だった，エラーの挿入箇所とその後の処理（検知のされ方）を指定したエラー挿入も実現できていた．これにより，開発者が想定したエラーケースを意図的に作り出し，開発対象のエラー耐性機構がそのようなケースに対応できるのかの検証を容易に行う事が可能であることが確認できた．

8. おわりに

本研究では memcahced を対象とし，指定したメモリオブジェクトに対して直接エラーの挿入が可能なフォールトインジェクタを提案した．メモリオブジェクト情報の提供，メモリオブジェクトへの２種類のエラー挿入，エラー挿入後の処理の指定という３つの機能を実現し，開発者が想定したエラーケースを容易に再現できる設計を施した．

実際に memcached 上にフォールトインジェクタを実装し，動作検証を行った結果，提案したフォールトインジェクタが正しく動作している事が確認できた．また，既存の耐メモリエラー機構である Software-based ECC が実装された memcached に対しても動作確認を行い，耐メモリエラー機構の機能の検証にも有効的に使えることを確認した．

本研究の今後の展望としては，汎用性の向上が挙げられる．本研究では memcached のみを対象とした FI を開発したが, 対象とするソフトウェアを拡張していき，より汎用性を高めていくことが課題になる．具体的には，memcached と同じ In-memory KVS である Redis [27] などへの適用が考えられる．

参考文献

[1] 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*, ACM, p. 193–204 (online), DOI: 10.1145/1555349.1555372 (2009).

[2] Luo, Y., Govindan, S., Sharma, B., Santaniello, M., Meza, J., Kansal, A., Liu, J., Khessib, B., Vaid, K. and Mutlu, O.: Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory, *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE, pp. 467–478 (online), DOI: 10.1109/DSN.2014.50 (2014).

[3] Carter, N. P., Naeimi, H. and Gardner, D. S.: Design techniques for cross-layer resilience, *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, IEEE, pp. 1023–1028 (online), DOI: 10.1109/DATE.2010.5456960 (2010).

[4] de Kruijf, M., Nomura, S. and Sankaralingam, K.: Relax: An Architectural Framework for Software Recovery of Hardware Faults, *SIGARCH Comput. Archit. News*, Vol. 38, No. 3, p. 497–508 (online), DOI: 10.1145/1816038.1816026 (2010).

[5] Cho, H., Leem, L. and Mitra, S.: ERSA: Error Resilient System Architecture for Probabilistic Applications, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 31, No. 4, pp. 546–558 (online), DOI:

ⓒ 2022 Information Processing Society of Japan

10.1109/TCAD.2011.2179038 (2012).

[6] Qin, F., Lu, S. and Zhou, Y.: SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs, *11th International Symposium on High-Performance Computer Architecture*, IEEE, pp. 291–302 (online), DOI: 10.1109/HPCA.2005.29 (2005).

[7] Taranov, K., Alonso, G. and Hoefler, T.: Fast and Strongly-Consistent per-Item Resilience in Key-Value Stores, *Proceedings of the Thirteenth EuroSys Conference*, ACM, p. 1–14 (online), DOI: 10.1145/3190508.3190536 (2018).

[8] 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*, ACM, p. 268–278 (online), DOI: 10.1145/2989081.2989091 (2016).

[9] Carreira, J., Madeira, H., Silva, J. and Informtica, D.: Xception: Software Fault Injection and Monitoring in Processor Functional Units, *Proceedings of the 5th IFIP Working Conference on Dependable Computing for Critical Applications* (2001).

[10] Lu, Q., Farahani, M., Wei, J., Thomas, A. and Pattabiraman, K.: LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults, *2015 IEEE International Conference on Software Quality, Reliability and Security*, IEEE, pp. 11–16 (online), DOI: 10.1109/QRS.2015.13 (2015).

[11] : memcached-a distributed memory object caching system, `https://memcached.org/`. (accessed 2022-01-08).

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

[13] Li, X., Huang, M. C., Shen, K. and Chu, L.: A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility, *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, USENIX Association, (online), available from ⟨https://www.usenix.org/conference/usenix-atc-10/realistic-evaluation-memory-hardware-errors-and-software-system⟩ (2010).

[14] 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, *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, p. 111–122 (online), DOI: 10.1145/2150976.2150989 (2012).

[15] 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)*, IEEE, pp. 13–25 (online), DOI: 10.1109/DSN.2019.00017 (2019).

[16] 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, *SIGPLAN Not.*, Vol. 50, No. 4, p. 297–310 (online), DOI: 10.1145/2775054.2694348 (2015).

[17] Sangchoolie, B., Pattabiraman, K. and Karlsson, J.: One Bit is (Not) Enough: An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors, *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, pp. 97–108 (online), DOI: 10.1109/DSN.2017.30 (2017).

[18] Kanawati, G., Kanawati, N. and Abraham, J.: FERRARI: a tool for the validation of system dependability properties, *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, IEEE, pp. 336–344 (online), DOI: 10.1109/FTCS.1992.243567 (1992).

[19] Skarin, D., Barbosa, R. and Karlsson, J.: GOOFI-2: A tool for experimental dependability assessment, *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, IEEE, pp. 557–562 (online), DOI: 10.1109/DSN.2010.5544265 (2010).

[20] Tsai, T., Iyer, R. and Jewitt, D.: An approach towards benchmarking of fault-tolerant commercial systems, *Proceedings of Annual Symposium on Fault Tolerant Computing*, IEEE, pp. 314–323 (online), DOI: 10.1109/FTCS.1996.534616 (1996).

[21] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S. and Paleczny, M.: Workload Analysis of a Large-Scale Key-Value Store, *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, Association for Computing Machinery, p. 53–64 (online), DOI: 10.1145/2254756.2254766 (2012).

[22] Berg, B., Berger, D. S., McAllister, S., Grosof, I., Gunasekar, S., Lu, J., Uhlar, M., Carrig, J., Beckmann, N., Harchol-Balter, M. and Ganger, G. R.: The CacheLib Caching Engine: Design and Experiences at Scale, *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, pp. 753–768 (online), available from ⟨https://www.usenix.org/conference/osdi20/presentation/berg⟩ (2020).

[23] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T. and Venkataramani, V.: Scaling Memcache at Facebook, *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, USENIX Association, pp. 385–398 (online), available from ⟨https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala⟩ (2013).

[24] Yang, J., Yue, Y. and Rashmi, K. V.: A large scale analysis of hundreds of in-memory cache clusters at Twitter, *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, pp. 191–208 (online), available from ⟨https://www.usenix.org/conference/osdi20/presentation/yang⟩ (2020).

[25] : The LLVM Compiler Infrastructure, `https://llvm.org/` (accessed 2022-01-06）.

[26] : Pin - A Dynamic Binary Instrumentation Tool - Intel, `https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html`. (accessed 2022-01-15).

[27] : Redis, `https://redis.io/` (accessed 2022-01-06）.

ⓒ 2022 Information Processing Society of Japan