Vol.2019-OS-146 No.11
2019/5/31

部分的なメモリ故障への耐性を有するインメモリキーバリューストア

下村 剛志[1,a)]　山田 浩史[1]

概要：メモリは外的要因によるビットフリップやモジュールの故障により保存したデータが呼び出せなくなることがある．特に In-Memory Key-Value Store(In-Memory KVS) はメモリを大量に使用するのでメモリページのエラーに遭遇する可能性が高いアプリケーションの一つである．しかし，In-Memory KVS はメモリ上に全ての key-value を展開しており，再起動にかかるコストが高い．既存研究では ECC などの誤り訂正符号を用いたエラーの回復を行っているが，広範囲に渡るメモリのエラーには対応できていない．本論文では，メモリに部分的な故障が生じたとしても，In-Memory KVS を継続して稼働可能にする手法を提案する．提案手法では OS Kernel と In-Memory KVS を連携させ，メモリページのエラー発生時に破損ページに保存されていたデータオブジェクトの回復処理を行い，動作を継続させる．本研究では Linux Kernel 4.13.9 と memcached 1.4.39 に提案手法の実装を行った．評価実験を行い，メモリエラー発生時でも約 3 秒のダウンタイムでスループットの劣化無しに動作が継続することを確認した．

1. はじめに

メモリの部分的な故障はサービスの可用性に大きく影響する．マシンのメモリ搭載容量は年々増加しているが，その一部分に訂正不可能なエラーが発生すると，保存した値に正常にアクセスできなくなり，その領域を使用していたアプリケーションは動作を正常に継続することができない．ハイエンドサーバは誤り訂正符号 (ECC) を持つメモリを搭載しており，エラー発生時に適切に反転たビットを訂正できるが，ECC では訂正できない多 bit のフリップやハードウェア故障によるエラーが発生するとメモリ内容が正しく訂正されない．こうしたエラーが生じると，多くの場合，オペレーティングシステム (OS) によってアプリケーションは停止され，再起動を余儀なくされる．Schroeder らの調査 [1] によると，1 年間でデータセンタにて運用されているマシンの約 32%にメモリエラーが生じ，その中でも約 14%が訂正不可能なハードエラーであった．

メモリ内に大量のデータを展開するビッグアプリケーションは使用するメモリ量が多いため，通常のアプリケーションに比べて相対的にメモリエラーに遭遇する確率が高くなる．ビッグメモリアプリケーションの代表格として，管理しているデータアイテムをメモリ上に存在する In-Memory Key-Value Store(In-Memory KVS) がある．In-Memory KVS ではデータを全てメモリ上で管理するため従来の HDD や SSD などのディスク装置を用いるデータベースよりも高い性能が実現できる．特徴として In-Memory である性質上， 大量のメモリを消費し，用途によっては TB 単位のメモリを 1 つのアプリケーションで使用することもある．データキャッシュとして用いられる例が多く，Facebook では，memcached や RocksDB といった In-Memory KVS 運用し，サービスの性能を向上させている [2].

In-Memory KVS がメモリの部分的な故障に遭遇した場合，サービスの品質を著しく低下させてしまう．再起動はメモリ内容すべての消失を伴うため，通常のアプリケーションよりもはるかにメモリ使用量の多い In-Memory KVS を再起動直前の状態に戻すには時間を要する．In-Memory KVS の場合，メモリ上に存在していた key や value などのデータ群すべてがメモリ上から失われてしまう．そのため，再起動後にディスクや他ノードのデータアイテムを利用して再構築する必要があり，使用していたメモリの量が多いほど正常動作までに時間を要してしまう．その間，スループットの低下が発生してしまいユーザからのリクエストに遅延や失敗が発生する．Facebook では約 2%の In-Memory KVS が再起動してしまうと，ユーザのクエリが部分的な失敗すると報告している [3].

本研究ではメモリの一部にハードウェア的な故障が発生しても動作を継続することのできる In-Memory KVS を提

[1] 東京農工大学
TUAT, Koganei, Tokyo 184–8588, Japan
[a)] tshimo@asg.cs.tuat.ac.jp

案する．故障したメモリページを検出し，保存されているメモリオブジェクトの種類に応じて，In-Memory KVS のメモリオブジェクトを再構成する．これにより，故障したメモリページを使用せずに In-Memory KVS の継続動作を可能とし，代替サーバが用意されるまでの臨時的な稼働を支援する．なお，マスターデータは別のストレージに保存されており In-Memory KVS をデータキャッシュとして用いる場合を対象としている．すなわち In-Memory KVS の一部の key，value のペアがメモリエラーで欠損し，アクセスができなくなってしまっても別のストレージから再度読み込むことで，スループットの低下は発生するが，In-Memory KVS 全体として見れば整合性の取れる場合を想定している．これにより In-Memory KVS の再起動によるダウンタイムや，メモリ上に展開されていたキャッシュがリセットされることによるスループットの低下を軽減させることが可能になる．

本論文の貢献は次のとおりである．

- メモリエラーが発生してもデータオブジェクトを再構成することで動作を継続することのできる In-Memory KVS を提案した．提案方式はメモリページのエラーが発生しても動作を継続することができ，再起動を行った場合に比べて性能の劣化が少ないという特徴を持つ．
- 提案手法を実現する機構を提案した．具体的には，メモリエラー発生時に対象のページを使っていたプロセスに通知を行い，メモリオブジェクトを再構成する機構を提案した．OS Kernel と In-Memory KVS が協調しエラーが発生した際にはプロセスのキルを行わずエラー領域の仮想アドレスを In-Memory KVS に通知する．In-Memory KVS ではその領域に保存されているデータの特性別の処理を行いメモリエラーから回復する．
- 提案手法を Linux Kernel 及び memcached 上に実装を行った．性能を評価する実験を行い，16GB のメモリ割り当て時にエラーからの回復時間を通常は 200 秒以上被るダウンタイムを 3 秒に短縮し，その後の性能劣化もみられなかった．

2. 背景

本節ではメモリエラーの種類や基本的な対策，またメモリを大量に使う In-Memory KVS について説明する．

2.1 メモリエラー

メモリエラーはその原因により大きくソフトエラーとハードエラーの 2 つに分類することができる．

2.1.1 ソフトエラー

ソフトエラーはメモリモジュール自体に破損は無いものの宇宙線 [4] などの外的要因によるビットフリップによって発生するエラーである [5]．特徴としては一時的にメモリの内容が書き換わってしまっているだけであるのでエラーの発生した領域に再びデータを書き込むことで正常な領域として用いることができることが挙げられる．主に 1 ビットだけがフリップしてしまうことが多く，データセンタなどで用いられるマシンでは後述するメモリエラー訂正符号によって修復され問題が表面化しない [6] ことも多い．

2.1.2 ハードエラー

メモリモジュール自体の故障でありハードエラーの生じた領域は永続的にエラーが発生するのが特徴である [7]．このエラーでは前述のソフトエラーとは異なり，複数ビットが同時に破損してしまうことが多く，そのためエラー訂正符号による修復ができないことも多い．ハードエラーが発生した場合，解決するにはメモリモジュール自体の交換が必要になる．

2.2 エラー訂正符号

現在，データセンタなどで用いられるサーバマシンのメモリにはエラー訂正符号 [8] が用いられていることが多い．現在主流となっている方式では 8Byte のブロックごとに 2bit のエラー検出と 1bit のエラー訂正が可能である．エラー訂正の機能の無いメモリと比較すると高価であるのでコンシューマ向けのマシンに用いられることは少ない．基本的にはエラー訂正が行われるタイミングは実際にデータにアクセスを試みて誤りを検出した場合のみである．

2.3 In-Memory Key-Value Store

In-Memory Key-Value Store (In-Memory KVS) はデータを key と value の組で管理するデータベースの中で， 特に key， value を全てメモリ上に展開するものを指す．In-Memory KVS では管理しているデータがメモリ上に存在するので従来の HDD や SSD などのディスク装置を用いるデータベースよりも高い性能が実現できる．In-Memory である性質上，大量のメモリを消費することが特徴で用途によっては TB 単位のメモリを使用することもある．

2.3.1 Memcached

Memcached はオープンソースな In-Memory KVS で主にキャッシュ用途として用いられる．採用例としては Facebook や Amazon[9] などに実際に利用されている．中でも Facebook は memcached を用いて秒間 10 億を超えるリクエストを処理 [2], [10] しており，スケーラビリティの高さを伺うことができる．memcached では slab と呼ばれる単位 (デフォルトでは 1MB) でメモリを確保し，必要に応じて様々な大きさに切り出してサイズごとに管理を行っている．図 1 のように slab は切り出したサイズごとに free リストを持ち，新たな要素の保存には free リストからメモリを割り当てることで malloc などシステムコールを呼び出す回数を減らし，動作速度を向上させている．また，key

図 1　memcached のメモリ管理イメージ

から value を検索する際には key のハッシュ値を計算し，ハッシュテーブルを用いて探索を行う．

2.3.2　再起動による性能の劣化

In-Memory KVS では再起動を実施するコストが高いアプリケーションの一つである．何故なら key-value のペアをメモリ上に展開しているのでアプリケーションの再起動が行われた場合，データ群をディスク装置などからメモリへと書き戻さなくてはならず元の性能と同等になるまでに時間を要してしまう．このため，In-Memory KVS では再起動をできるだけ避けたいが，訂正不可能なメモリエラーに遭遇した場合には再起動を余儀なくされている．また，メモリの使用量が非常に多いので他のアプリケーションに比べてメモリエラーに遭遇する可能性も高いと言える．

3.　関連研究

予期しないメモリエラーの発生に対処するために様々な研究が行われている．

メモリに関する調査としてベンダごとメモリエラーの発生条件などを調査した Memory Errors in Modern Systems[11] やメモリのエラー発生率を仮定し，それがシステムに与える影響を調査した論文 [12] がある．前者ではメモリベンダごとにメモリエラー発生の特性が異なることを示しており，後者ではエラーの発生する場所に偏りがあることを定量的に示している．

Heterogeneous-Reliability Memory Systems[13] ではデータセンタのハードウェア導入に掛かるコストを特徴の異なるメモリ使用することで削減することを提案している．メモリエラーが発生しても動作を継続できる可能性の高い部分に対して誤り訂正を持たないメモリを，それ以外の部分に誤り訂正機能を持つメモリを用いることでシステムの稼働率とコストの両立を図っている．

Software Based ECC[14] と Metadata Integrity Protection[15] では共にソフトウェアにメモリの改ざんを検知・修正する機能を持たせることで現在のシステムの信頼性を向上させている．前者は現在用いられているハードウェアによる誤り訂正符号 [16] よりも強力な誤り訂正符号をソフトウェアで実装することでメモリエラーからの回復率を向上させ，訂正不可能なメモリエラーの発生確率を低下させている．後者は主にファイルシステムに存在するバグによりメタデータが意図せず破壊されてしまうことをチェックサム [17] やデータの特徴を用いて防いでいる．

加えて Relax[18] によるとトランジスタの小型化に合わせてエラーの発生率が上昇傾向にあるがこれ以上複雑なエラー訂正技術をハードウェアに実装することはコスト・消費電力の面から難しくなってきている．また，複数ビットが同時に読み出せなくなってしまうことも多いハードエラーの発生率はソフトエラーよりも高く [1], [19]，発生した場合に既存のメモリエラーへの対策手法では正しく訂正できず，アプリケーション/OS カーネルの再起動が必要になる可能性が高くなってしまう．

再起動を余儀なくされるとダウンタイムが発生するだけでなく，前述した In-Memory KVS など，大量にメモリを使用するアプリケーションでは再起動後に大きな性能の劣化が生じる．

4.　提案

本研究ではメモリの破損が発生した場合でも動作を継続することのできる In-Memory KVS を提案する．提案手法により，破損したメモリページ上のオブジェクトを再構成させ，アプリケーションの動作を継続させることでダウンタイムの削減，再起動に伴うスループット劣化の低減を目指す．

図 2 に示すのが提案するメモリエラーからの回復機構の概要である．動作としては OS Kernel がメモリエラーを検出するとその領域を使用していたアプリケーションが In-Memory KVS であればその情報を伝える．通知を受けた In-Memory KVS は一部の key, value の要素の欠損を許容して内部のデータ構造を修正して動作の継続を行う．

本アプローチであると key, value の領域が破損してしまった場合にはアクセスできない要素が発生してしまうが，本研究のターゲットである In-Memory KVS のキャッシュ用途での利用であればディスク装置から再度読み込みを行えばよい．勿論，破棄された value を取得しようとした場合，ディスクへの読み込みを行う分だけシステム全体としての性能は劣化してしまうが In-Memory KVS 全体を再起動するよりは低い性能劣化で済む．

4.1　破損ページの通知

メモリが破損した場合，破損したページを使用していたプロセスを特定し通知を行う必要がある．しかし，アプリケーション自身が管理している情報だけでは

- 破損ページを自身が使用しているか
- 破損ページは自身のどの仮想アドレスに当たるか

を判断することができない．そこで，本手法では OS Kernel と連携してアプリケーションがアドレス情報を取得することでこれを解決する．

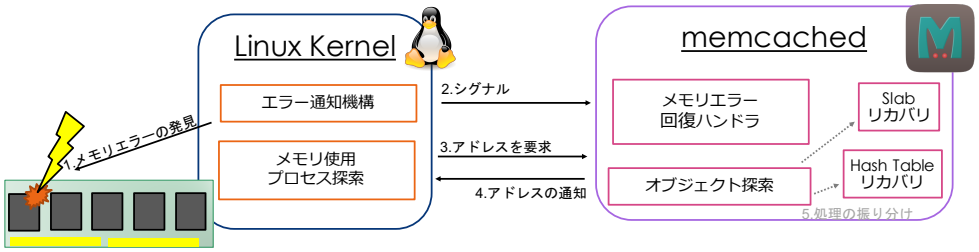

図 2　提案方式の概要

実際にメモリの破損が検出された場合はシグナルを利用して対象のメモリページを使用していたアプリケーションに対して通知を行う．その際，破損したページがアプリケーション上のどの仮想アドレスに割り当てられているかを同時に通知する．

4.2　データオブジェクトの再構成

破損したページ上にあるデータは正常に読み込むことができないので何らかの処理が必要になる．この時，既にアプリケーションは 4.1 より破損したメモリページの仮想アドレスを取得しているのでこれを利用して自身の持つデータオブジェクトの再構築を行う．また，これ以降の処理はアプリケーション (本研究では In-Memory KVS) のみで行う．これは実際に使用しているメモリの使用用途を把握しているのはアプリケーションであり，対象とするアプリケーションに応じてデータオブジェクトの再構成の方法が異なるからである．

アプリケーションでは予め

- 管理しているメモリの一覧
- 故障発生時の処理

を記録しておき，通知された仮想アドレス情報と照らし合わせてデータオブジェクトの再構成を行う．

4.3　技術的な課題

メモリエラーが生じたアプリケーションは破損したページにアクセスせずに動作を継続する必要がある．これは単に破損したページをアクセスできないようにすればよい (malloc で確保された領域に対して free を実行するなど) のではなく，破損したページのアドレスを持つ他のデータ構造 (memcached では LRU リストや free リストなど) も合わせて整合性を保てるように変更しなくてはならない．

Memcached の例を挙げると，図 3 のように，内部でメモリを管理する単位である slab は他の slab や slab を管理するテーブルからポインタで相互に参照していることがわかる．よって図 3 中の slab A にメモリページの破損が生じたと仮定した場合，LRU, free リストを再構築し slab A が含まれない形に修正することが必要である．

5.　設計

5.1　メモリエラー処理

5.1.1　対象プロセスの登録

アプリケーションがメモリエラーから回復し，動作を継続することのできる手法を持っていることを Kernel に示す必要がある．ここでは，メモリページの破損から回復可能であることを示すためにプロセスの起動時にシステムコールを用いて自身の PID を Kernel に持たせたリストに追加する．こうすることで，実際にメモリエラーが起き，破損したページを使用していたプロセスにシグナルを送る際にアプリケーションがメモリエラーから回復できるのか否かを Kernel が判断できるようになる．

5.1.2　対象プロセスの判別

メモリエラー発生時に予め登録されているプロセスには kill シグナルではなく別のシグナルを送信する必要がある．この時，5.1.1 に登録されたリストに存在するプロセスのみを探索することで，メモリエラーからの回復手法を持つアプリケーションが故障したページを使用していたかどうかを判断することができる．このリスト中の以外のアプリケーションが故障したメモリページを使用していた場合にはデフォルトの処理である kill シグナルを送信し，アプリケーションを強制終了させる．

また，破損したページを使用していたプロセスを探索する際に予め回復対象の PID が記録されたリストに記録されたプロセスのメモリマッピングのみを探索することで，起動している全てのプロセスのメモリマッピングを探索しなくてよいため，不要なコストがかからなくて済むようになっている．

5.1.3　プロセスへの通知

対象のプロセスにはシグナルを送り，アプリケーション独自のメモリエラー回復用のハンドラをを起動させる．この時点でシグナルを受信したプロセスは自身の管理するメモリの何処かにエラーが生じたことしかわからない．そこで，システムコールを発行し，自身のどの仮想アドレスのメモリページの破損が生じたのかを取得する．

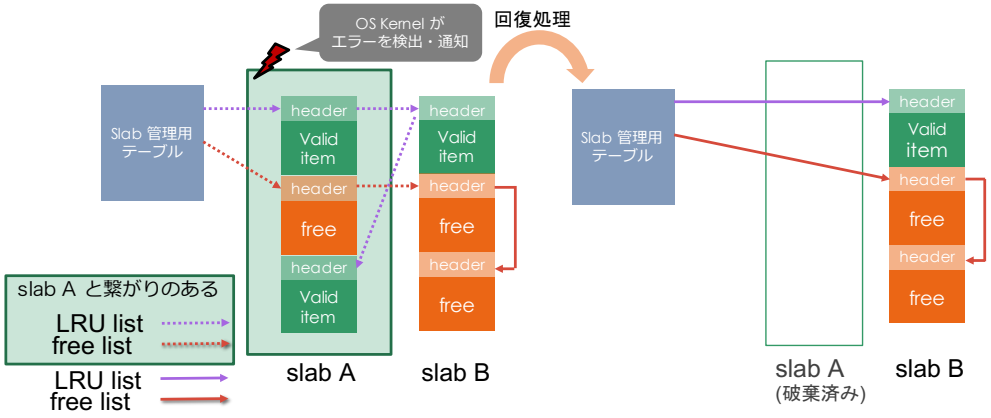

図 3　LRU, free リストと slab の関係

5.2　破損ページの処理

メモリページの故障に遭遇したアプリケーションが正常に動作を継続するためには破損したページに存在していたデータオブジェクトを再構成しなくてはならない．多くの場合これは取り扱うデータの性質や容量によって異なる処理を行う必要がある．本セクションでは memcached を例に挙げ，具体的な破損ページの処理の種類について紹介する．

5.2.1　再構築

破損したページ以外のデータオブジェクトから再構築が可能な場合にはこれを行う．memcached では検索用ハッシュテーブルがこれに当たる．ハッシュテーブルは管理する key から value を高速に検索するために用いるデータオブジェクトである．

ハッシュテーブルは検索用の key をハッシュ関数にかけた結果を用いて key，value へのポインタを保管している．そのため，ハッシュテーブルが破損してしまっても，保存されている全ての key，value を精査することで再構築を行うことが可能である．

また，memcached のハッシュテーブルではチェインハッシュが用いられている．チェインハッシュとは，同じハッシュ値を持つ要素があった場合，ハッシュテーブルから数珠繋ぎに要素を繋いでいく方法である．全ての要素を辿り，ハッシュテーブルを再構築すると，チェインハッシュの順番が入れ替わる可能性があるが，この場合でも memcached は整合性を保って動作を継続することができるので問題は発生しない．

5.2.2　破棄

対象のページに存在していたデータが無くてもアプリケーションが正常に処理を続けられる場合にこれを選択する．memcached では保管しているデータである key，value のペアを保存しているハッシュテーブルがこれに該当する．本研究では memcached をキャッシュとして運用する場合を対象としているのでメモリページの破損発生時に一

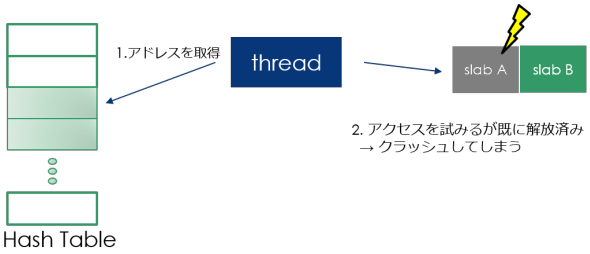

図 4　想定外の領域へのアクセス例

部の value が読み出せなくてもそのページ以外に保存されていた value にアクセスすることができれば問題はない．

この時，破損が発生した slab 領域以外にも free リストや LRU リスト，チェインハッシュやハッシュテーブル本体など，影響を受ける幾つかのデータオブジェクトも同時に整合性が取れるように変更しなくてはならない．例えば図 3 中で slab A が破損した場合は slab A と関係のあるポインタを変更しなくてはならない．この図では点線で描かれた LRU リストと free リストが変更しなくてはならないポインタに当たる．

5.3　アプリケーションの調整

データを保持するためのデータ構造以外に実行中の thread や lock の状態にも注意しなくてはならない．

5.3.1　Thread

Slab の再構築など，データ構造を修正している際に In-Memory KVS がクライアントからの要求を受け付けてしまうと予期せぬアクセスが発生してしまう可能性がある．例えば図 5 のようにハッシュテーブルに残されていたアドレスにアクセスを試みるが既に free されていることが挙げられる．ここではメモリエラーが発生した thread を含む worker thread 全てを一旦破棄し．データ構造の修正が完了した後に再び生成をおこなう．

5.3.2　Lock

Thread を実行中の状態を加味せずに破棄してしまうと dead lock が発生する場合がある．例えば memcached では

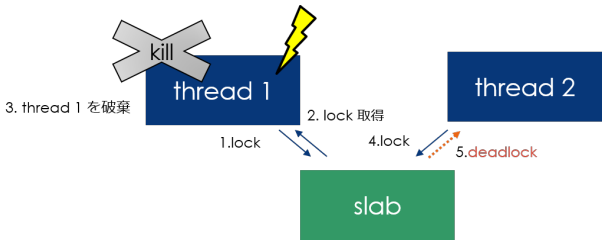

図 5　dead lock の例

SET 命令時に slab 単位で lock を取得しており，lock が取得されたままになっていると同じ slab に配置された別の要素へのアクセスもできなくなってしまう．この状況を解決するために lock 処理を hook し，thread が保持している lock の一覧を記録・管理し，thread 再生成時に unlock を行うことで解決を図る．

6. 実装

6.1 Linux Kernel

6.1.1 対象プロセスの管理

アプリケーションがエラーからの回復手法を持つことを Linux Kernel に示すために新たなシステムコールを実装する．追加するシステムコールは，実行したアプリケーションの pid を Kernel が保持するリストに登録する．

6.1.2 メモリエラー処理の追加

既存のメモリエラー処理にアプリケーションが独自の回復手法を持つ場合にはそれを優先するように改良を加える．具体的には Linux Kernel の mm/memory-failure.c にある memory_failure 関数の冒頭で 6.1.1 で追加したシステムコールにより記録した pid のリストを探索する．その結果に応じて対象のプロセスに送信するシグナルを決定する．

pid のリストに登録があった場合には，破損したメモリページを管理する page 構造体を引数に取り，予め記録した pid をのプロセスが持つ全ての page 構造体の割り当てをチェックする．そして該当するプロセスにおける仮想アドレスを取得する．Linux Kernel ではこの結果を元に対象のプロセスにシグナルを送信する．

6.1.3 対象プロセスへの通知

Linux Kernel は破損したページを使用していた対象のプロセスに対してシグナルを送信し，仮想アドレスを通知する．

Kernel からアプリケーションに破損したメモリの仮想アドレスを通知するために構造体を用いる．Kernel はアプリケーションにシグナルを送信する前に構造体に破損したメモリページの仮想アドレス，対象アプリケーションの pid を記録し，リストに追加する．

シグナルを受信したアプリケーションでは自身の pid を Linux Kernel が保持しているリストから探索する．Kernel の保持するリストから自身の pid を発見した後はユーザー空間のみの処理となり，その実装は各アプリケーションに依存する．

6.2 Memcached

メモリページの破損のアプリケーションの通知を実装した Linux Kernel と連携できるように In-Memory KVS である memcached に改造を加えていく．

6.2.1 監視対象プロセスへの追加

プロセスの起動時に自身がメモリエラーからの回復手法を持つことを Kernel に登録することでエラー発生時にシグナルを受信できるようする．

具体的な処理としては 6.1.1 で追加したシステムコールを用いて自身の pid を Kernel に登録する．

6.2.2 シグナルハンドラの登録

メモリエラー発生時に送られるシグナルを処理するためのハンドラを起動時に登録しておく．Linux では signal 関数を用いて対象のシグナルとそのシグナルを受信した時に呼び出される関数をハンドラとして指定することができる．

ここでは 6.2.1 の監視対象プロセスへの追加と合わせて memcached の起動時に main 関数の最初で登録処理を実行する．これによりクライアントとの通信を始める前にメモリの保護を開始することができる．

6.2.3 メモリ使用領域の記録

memcachd が実際に破損したメモリ上に存在するデータオブジェクトの修復を行う場合，その部分に存在していたデータの種類に応じて破棄や再構築などの処理を行う必要がある．破損したメモリページの領域が key-value であったのか，ハッシュテーブルであったのかなどを特定するために予めメモリを確保した時に使用領域を記録しておく．

使用領域を記録する関数では確保した領域の仮想アドレスと，その領域が破損した際に実行される関数ポインタを引数とし登録を行う．このリストはメモリページの破損時に実行する回復処理の特定に使われる．

6.2.4 ページ破損時の処理

Kernel から通知された仮想アドレスから故障したメモリオブジェクトを特定し対応する処理を行う．更に登録を行ったアドレスリストを探索する関数を用いて，破損したメモリページに存在するメモリオブジェクトの種類に応じた処理を呼び出している．

現状，メモリページの破損から回復できるのは破棄を行う key-value 及び再構築を行うハッシュテーブルのみである．ここでは例として key-value を保管している slab がメモリページの破損に遭遇した時の処理を紹介する．

slab の破損時には

- 破損した slab の削除
- free/LRU リストの再構築
- ハッシュテーブルの再構築

の 3 つの処理を順に行う．この内，上 2 つは破損した slab と同サイズのオブジェクトを管理していた slab 全体が，一

表 1　実験環境

機器名	DELL PowerEdge T110 II
CPU	Intel(R) Xeon(R) CPU E3-1270 V2 @ 3.50GHz
Cores	4C8T
RAM	32GB

番下では memcached 全体で有効な要素全ての再探索が必要になる．

6.2.5　破損した slab の削除

memcached では slab を特定のサイズに切り出して key-value を格納している．そのため slab に格納できるサイズごとに現在の割り当てを記録する構造体を保持している．まずはその slab を管理している構造体の一覧から破損したページの含まれている slab を探し出し，その slab を保持しているという情報を削除する．

6.2.6　free/LRU リストの再構築

しかし破損した slab を保持しているという情報を削除しただけでは問題は解決しない．memcached は新たなアイテムの追加が行われる時，既に割り当てている同容量の slab に未割り当ての領域があれば優先的に割り当てる (free リスト)．また，memcached の起動時に指定した最大使用メモリ量を超え，free リストが空の場合には LRU リストを用いて key-value の保存領域を確保している．

この時，破損したメモリページと同じサイズの要素を保存している slab を探索し，free リスト/LRU リストの再構築を行わないと新たなアイテムの追加時に既に開放された領域に対してアクセスしてしまう問題が発生する．

なお，LRU リストでは処理の高速化のために最終アクセス時間を確認せず探索した順番にリストに追加していく．そのため，slab 領域に対してメモリの再構築処理が実施されると一時的に LRU リストではなくなるため，時間的局所性を考慮したアクセスが行われた場合，スループットが低下してしまうことが想定される．

6.2.7　ハッシュテーブルの再構築

Free リスト/LRU リストの再構築を行うことで新たなアイテムの登録に対しての対策を取ることができたが，これだけでは memcached にアイテムを要求する際に不都合が生じる場合がある．

具体的には，ハッシュテーブルの再構築を行わないと，探索中に free リスト/LRU リストと同じく既に開放されている領域に対してアクセスを行ってしまう可能性がある．これを防ぐために memcached で管理している全ての key-value を探索し，新たなハッシュテーブルを再構築する必要がある．

7.　実験

7.1　実験環境

提案手法の評価を表 1 に示すマシンで行う．

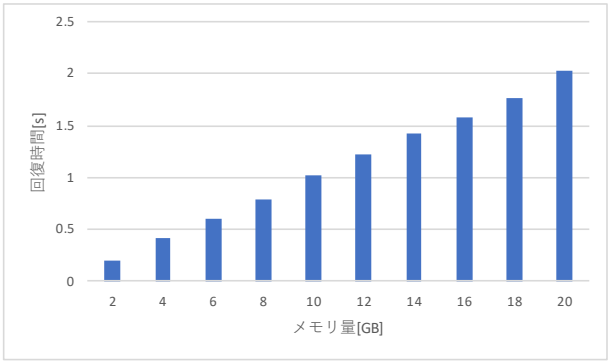

図 6　slab 破損からの回復時間の推移

また，OS は Ubuntu Server 18.04 を用い，使用する kernel は Linux 4.13.9 をベースに提案手法の実装を行ったものである．In-Memory KVS である memcached[20] はバージョン 1.4.39 を元にメモリ破損からの復旧のための提案手法の実装を行ったものを使用する．

7.2　slab 破損からの回復時間の評価

実装されているのは slab 破損からの回復とハッシュテーブルの再構築であるが，slab の破損からの回復を行うとハッシュテーブルの再構築も実行されるため，コストの大きい回復処理は slab 破損からの回復である．

Memcached のが slab に使用するメモリ量が多ければメモリページのエラー発生時にデータオブジェクトの再構築にかかる時間も多くなるはずである．

実験方法

本実験では memcached に割り当てるメモリ量を 2GB から 20GB まで変化させ，それぞれの容量で memcached が割り当てられたメモリを全て使用した状態で slab 破損が発生した時と同じ処理を行い，その時間を計測する．

実験結果

slab 破損からの回復処理にかかった時間の推移を図 6 に示す．この結果から，memcached が使用しているメモリ量が増加すると slab 破損からの回復処理にかかる時間が線形的に増加していることが確認できる．

今回の実験で計測した範囲では，20GB のメモリ使用時で 2.03 秒と実用上問題の無い値であったが，memcached の使用するメモリが増加すると回復時間が多くなり，再起動をした方が性能が高くなってしまう可能性がある．

図 6 のグラフはほぼ線形に増加しており，近似式を求めると $y = 0.1992x + 0.0087$ となる．これを用いて，1TB のメモリを memcached に割り当て slab 回復処理を実行した時のことを想定してみると，約 204 秒かかってしまう．実際にメモリを増加させたときに近似式に従って回復時間が推移するのか，memcached の再起動を行った場合と比較してスループットが悪化していないかなどを調査することが今後の課題として挙げられる．

表 2　slab の占めるメモリ割合

割当 RAM(GB)	全体 (MB)	slab(MB)	slab 率 (%)
1	1154	1024	88.7
2	2180	2049	94.0
4	4320	4099	96.9
8	8329	8299	98.4
16	16530	16399	99.2

7.3　slab のメモリ使用率

本研究では key-value のデータが実際に保存される slab 領域が使用するメモリの割合が著しく高いことを前提としてメモリエラーから保護する部分を決定した．ここでは memcached が実際にどの程度 slab 領域にメモリを使用しているかを確認する．

実験方法

memcached に割り当てるメモリ量を 1GB から 16GB まで変化させ，それぞれ割り当てられたメモリをすべて使用するように key-value の SET を行う．その時の memcached 全体のメモリ使用量, slab 領域のメモリ使用量をそれぞれ計測し，slab 領域がメモリ使用の面で支配的かどうかを確認する．

実験結果

実験結果と slab の占める割合を表 2 に示す．slab の占める割合の推移から，memcached へ割り当てるメモリ量が増加するほど slab 領域が占める割合が増えていることがわかる．結果から 16GB を memcached に割り当てた時点で slab 領域は全体の 99.2%のメモリを使用しており，十分に支配的だと言える．

これは割当量が増加してもハッシュテーブルのサイズやその他メタデータの増加は少量であるものの，key-value の使用する領域は同じペースで使用メモリを増やし続けるからである．

7.4　fault-injection からの復旧率

本研究では主に slab 領域が破損した際に動作を継続することを目標としている．前の実験から 16GB のメモリを memcached に割り当てた時に約 99.2%が slab 領域であることが分かっているので memcached の使用するメモリをランダムに選んだ場合，これに近しいメモリエラーからの回復率が求められるので確認する．

実験方法

Linux kernel 上で memcached が使用しているメモリページから 1 つをランダムに選びメモリエラーが起きたページとして memcahced に signal を送る．signal を受け取った memcached はメモリエラーからの回復を試みる．その結果を 500 回分記録し，メモリエラーからの復旧率を計測する．

実験結果

表 3 に示す通り，回復率は slab の占める割合と同じ99.2%が計測された．また，メモリエラーからの回復に失敗した 4 回の事例についてもすべてが slab 領域外のページが選ばれたことによるエラーであり，slab 領域が選択された後に回復に失敗した例は無かった．

表 3　slab の占めるメモリ割合

合計回数	回復成功回数	回復失敗回数	回復率 (%)
500	496	4	99.2

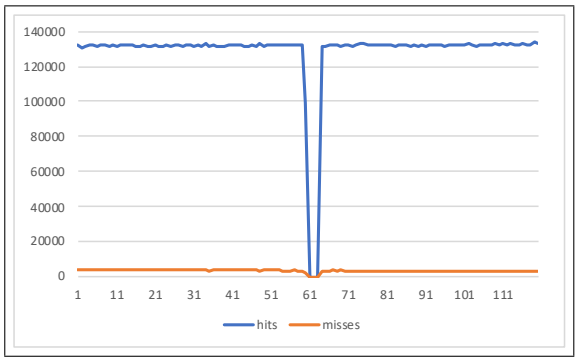

図 7　エラー回復前後でのスループット推移

7.5　メモリエラー発生前後のスループットの推移

メモリエラーからの回復処理の前後でキャシュの hit, miss 数がどのように推移したかを計測する．この実験ではベンチマークソフトとして Twitter のキャッシュサーバーをシミュレートした memcached 向けのワークロード Data Caching[21] を用いて実環境に近い形での性能を計測する．

実験方法

はじめに memcached に 16GB のメモリを割り当て，その領域すべてに Data Caching が生成するキャッシュを書き込む．その後，GET, SET の混在したベンチマークを実行し，その途中にメモリエラーが生じたことを仮定し回復処理を実行させる．

その際にキャッシュの hit 数, miss 数がどのていど変化したかを計測する．

実験結果

結果は図 7 に示すように約 3 秒のダウンタイムが発生するものの，エラー回復前後で hit, miss 数共に大きく変化していないことがわかる．これは memcached が割り当てている slab 数約 16000 に対して 1 つが欠損したに過ぎず In-Memory KVS 全体として影響が観測できなかったものと考えられる．

8.　おわりに

8.1　まとめ

本研究ではメモリエラー発生時に In-Memory KVS を継続動作させることにより性能の低下を押さえる手法を提案した．Linux Kernel とアプリケーションが連携し，Linux

Kernel がメモリページの破損が生じた場合にそのアドレスを通知することでアプリケーションにメモリの破損からの回復を委ね動作を継続させている．提案手法を Linux Kernel と memcached に実装し性能を計測したところ，メモリエラーから memcached が復帰できずに再起動した場合と比べて，メモリエラー回復後 10 秒で 82%，240 秒で 30%のスループット向上を達成した．

8.2 今後の課題

8.2.1 他の In-Memory KVS への適用

本論文では提案手法を memcached に適用したが，In-Memory KVS で有名なものとして他に Redis を挙げることができる．Redis も memcached 同様にハッシュテーブルを利用した In-Memory KVS であり，提案手法が適用できると考えている．

8.2.2 キャッシュ用途以外への利用

現状では slab 領域が破損すると保存されていた key-value の情報は失われてしまう．このため，データが別の場所に保存されているキャッシュ用途にしか用いることができない．

これを解決するためには，slab 領域に対してパリティ符号を用いればよい．しかし，計算したパリティ符号を保持するための領域が多くなればその分，memcached が key-value の保存に使用できるメモリ量が減少してしまう．多くの key-value をまとめて 1 つのパリティ符号とすることもできるが，メモリページの破損が生じた際に発生する回復にかかるオーバヘッドが高くなってしまう．

また，パリティを構成する要素が同時に破損してしまった場合には回復ができなくなってしまう．この点についても同一 slab からパリティを生成しないなどの工夫で解決ができると考えている．

参考文献

[1] Schroeder, B., Pinheiro, E. and Weber, W.-D.: DRAM Errors in the Wild: A Large-scale Field Study, *Commun. ACM*, Vol. 54, No. 2, pp. 100–107 (online), DOI: 10.1145/1897816.1897844 (2011).

[2] 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, *Proceedings of Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USENIX, pp. 385–398 (online), available from ⟨https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala⟩ (2013).

[3] Goel, A., Chopra, B., Gerea, C., Mátáni, D., Metzler, J., Ul Haq, F. and Wiener, J.: Fast Database Restarts at Facebook, *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, New York, NY, USA, ACM, pp. 541–549 (online), DOI: 10.1145/2588555.2595642 (2014).

[4] O'Gorman, T. J., Ross, J. M., Taber, A. H., Ziegler, J. F., Muhlfeld, H. P., Montrose, C. J., Curtis, H. W. and Walsh, J. L.: Field testing for cosmic ray soft errors in semiconductor memories, *IBM Journal of Research and Development*, Vol. 40, No. 1, pp. 41–50 (online), DOI: 10.1147/rd.401.0041 (1996).

[5] Baumann, R.: Soft errors in advanced computer systems, *IEEE Design Test of Computers*, Vol. 22, No. 3, pp. 258–266 (online), DOI: 10.1109/MDT.2005.69 (2005).

[6] Milojicic, D., Messer, A., Shau, J., Fu, G. and Munoz, A.: Increasing Relevance of Memory Hardware Errors: A Case for Recoverable Programming Models, *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, EW 9, New York, NY, USA, ACM, pp. 97–102 (online), DOI: 10.1145/566726.566749 (2000).

[7] Takeuchi, K., Shimohigashi, K., Kozuka, H., Toyabe, T., Itoh, K. and Kurosawa, H.: Origin and characteristics of alpha-particle-induced permanent junction leakage, *IEEE Transactions on Electron Devices*, Vol. 37, No. 3, pp. 730–736 (online), DOI: 10.1109/16.47779 (1990).

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

[9] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store, *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, New York, NY, USA, ACM, pp. 205–220 (online), DOI: 10.1145/1294261.1294281 (2007).

[10] Li, X., Shen, K., Huang, M. C. and Chu, L.: A Memory Soft Error Measurement on Production Systems, *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, Berkeley, CA, USA, USENIX Association, pp. 21:1–21:6 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1364385.1364406⟩ (2007).

[11] 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).

[12] 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).

[13] 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, *Proceedings of 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*, pp. 467–478 (online), DOI: 10.1109/DSN.2014.50 (2014).

[14] 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).

[15] Kumar, H., Patel, Y., Kesavan, R. and Makam, S.: High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System, *Proceedings of 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, USENIX Association, pp. 197–212 (online), available from ⟨https://www.usenix.org/conference/fast17/technical-sessions/presentation/kumar⟩ (2017).

[16] Chakraborty, K. and Mazumder, P.: *Fault-tolerance and reliability techniques for high-density random-access memories*, Prentice Hall PTR (2002).

[17] Mallory, T. and Kullberg, A.: Incremental updating of the Internet checksum (1990).

[18] de Kruijf, M., Nomura, S. and Sankaralingam, K.: Relax: An Architectural Framework for Software Recovery of Hardware Faults, *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, New York, NY, USA, ACM, pp. 497–508 (online), DOI: 10.1145/1815961.1816026 (2010).

[19] Al-Ars, Z., van de Goor, A. J., Braun, J. and Richter, D.: Simulation based analysis of temperature effect on the faulty behavior of embedded DRAMs, *Proceedings International Test Conference 2001 (Cat. No.01CH37260)*, pp. 783–792 (online), DOI: 10.1109/TEST.2001.966700 (2001).

[20] : memcached - a distributed memory object caching system. `http://www.memcached.org/`.

[21] : Data Caching. `http://cloudsuite.ch//pages/benchmarks/datacaching/`.