Vol.2017-OS-140 No.9
2017/5/17

ソフトウェア若化を指向するインメモリDBMS

十文字 優斗[†1,a)]　山田 浩史[†1,b)]

概要：現在運用され利用されているプログラムには，少なからずバグが残ってしまうことが知られている．これらのバグには原因の特定が難しいものがあり，開発段階での完全除去は困難である．運用中の障害を緩和する手段の一つとして Software Rejuvenation が利用されている．これは再起動によりソフトウェアを安定した状態にすることで障害から復旧する手段である．再起動によって発生するメモリやリソースの解放は障害の原因であるソフトウェア老化を改善する．ただし，これはデータベース管理システム (DBMS) に対して相性が悪く性能低下を招く．そのため， DBMS については障害対策としてのソフトウェア若化が適用しにくい．そこで，本研究では Software Rejuvenation を指向した DBMS の設計手法を提案する．提案手法はデータの保持により性能劣化を抑える機能と書き込み制限によるクラッシュ対策の 2 点が中心となる．データ保持ではソフトウェア若化を考慮し，slab calcification への対応も行う．書き込み制限についてはチェックサムを併用し，性能の維持も図った．提案手法を Memcached 1.4.25 に実装した．実験を行ったところ，再起動後のスループットを維持してクラッシュへの耐性を持つことや性能劣化を抑えることを確認できた．

1. はじめに

現在のアプリケーションの運用ではプログラム上に少なからずバグが残ってしまうことを許容し，発見次第対処的にパッチなどによる修正を加えている．開発段階でこれらのバグを発見することは難しく，実際の運用ではこのバグによる障害に対してどのように対処するかが重要である．

このようなバグの対策として Software Rejuvenation(ソフトウェア若化) が利用されている [1]．これはソフトウェアを再起動により若化させるというもので，蓄積した異常を解放することで障害の発生率を低下させることができる．メモリリークの例であれば再起動により全てのメモリを解放することで解放を忘れているメモリを解放し，必要なメモリ確保のみを再度行うという対応が可能である．

ソフトウェア若化には Pro-active と Re-active があり，それぞれの特徴がある [2]． Pro-active なソフトウェア若化は異常が蓄積し障害などの稼働率に影響する問題が発生する前に若化を行うものであり，定期的なアプリケーションの再起動により動作を安定させる．これは深夜に稼働しないシステムであれば毎日営業時間後に再起動によってソフトウェアの若化を行うことにより，運用中の障害の発生率を減らすことができる．一方，Re-active は異常や障害が発生してからアプリケーションの再起動という対処を行うことであり，実際の運用の中ではこの手段が多く利用される．サービスの種類によっては 24 時間利用のために再起動によってサーバを停止させることができない場合があり，その場合はこちらが利用されやすい．

DBMS はソフトウェア若化に対して相性が悪く，ソフトウェア若化によってその性能が著しく低下する．これは DBMS が通常の性能を発揮するためのウォーミングアップに時間がかかるためである．ソフトウェア若化によってメモリの状態は正常に戻るが，それまでに用意されたキャッシュも全て解放してしまい，特にデータベースでは起動後にディスク上のデータベースの情報を読み出しメモリ上にキャッシュするまでの間にデータベースの容量に応じたの Disk I/O が発生する．大量のデータを扱うことの多いデータベースではこの問題が顕著に現れ，その影響も大きい．これは Pro-active でも Re-active でも問題となり Pro-active な手法をとるための時間的コストからこの対処を行わない結果障害発生率が上がったり，Re-active な対処に時間がかかることによる大きな稼働率の低下につながる．Facebook では DBMS の再起動に多くの時間がかかることを問題視しており [3]，再起動を利用しやすい DBMS が必要とされる．Facebook が利用する大規模な DBMS では，そのプログラムの更新のために再起動を必要とする機会が多い．これに対しソフトウェアは再起動にかかる時

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

間が大きくシステム全体で 12 時間程度かかっていた．これは DBMS が保存しているデータ量が多く，再起動後にディスク I/O が多量に発生することが原因である．これによって再起動によってシステムの稼働率が低下していたため，これに対する対策手段をとることとなっている．

本研究では，ソフトウェア若化を指向した DBMS の手法としてデータベースの一部をメモリ上に保持したままソフトウェア若化を行う手法を提案する．ここでは特にデータベースのキャッシュとして利用される Memcached[4] に対してこの機能を実装することでこの手法の有用性を確認する．ここでソフトウェア若化を指向するにあたって必要となる要素は再起動後の性能劣化の緩和とクラッシュ時のデータの整合性保証である．性能劣化の軽減としては再起動時に必要なデータであるデータベース自体を保持したまま再起動することで再起動直前に近い性能に復帰する手法を提案する．また，クラッシュ後の再起動を考慮したデータの整合性の保証も行う．

本論文の貢献は次の通りである．

- ソフトウェア若化を指向する DBMS の手法を提案した．提案手法は次の特徴を有する．DBMS の性能劣化を抑えるためのデータ保持．保持したデータがクラッシュの影響を受けないよう保護するためのデータ整合性保証．
- 提案方式を実現するメカニズムを提案した．共有メモリを利用したデータ保持手法と，データ整合性保証として Readonly slab および multi-checksum の手法を提案した．
- 以上の提案手法を実装し実験した．実装では Memcached 1.4.25 に対して提案手法を実現した．実験では 93% のスループットを維持しながら再起動直後の性能劣化の抑制とデータの整合性保証を確認した．

2. ソフトウェア若化

ソフトウェア若化はソフトウェアを再起動することによりアプリケーションを若化する運用手法である．この手法が有効な障害の原因として代表的なメモリリークであれば，そのメモリを一度解放して再構築することで問題を解消する．ただし，この手法は特にインメモリ DBMS では保持しているデータベースをすべて解放するため再度データベースが構築されるまで性能が低下した状態となる．また，この手法は大きく二つに分けることができる．

2.1 Pro-active なソフトウェア若化

定期的なソフトウェアの再起動などの障害を未然に防ぐソフトウェア若化を特に Pro-active な手法と言う．Pro-active な手法では多くの Mandelbug による老化を解消し，それによって引き起こされる障害を未然に防ぐことになる．

この手法では障害の発生を防ぐことで復旧やデータの整合性確認などの時間を短縮できる利点がある．一方でこの手法はコストの面から避けられる傾向にある．ソフトウェア若化ではソフトウェアの再起動が必要となり，これを実行している間はサービスを停止しなければならないため稼働率の低下をもたらす．その上で，実際に障害が発生する確率やその復旧にかかる時間は発生するまで想定することも難しく，発生してから対応する対処的な対応が利用されている．

2.2 Re-active なソフトウェア若化

障害が発生した際にソフトウェアを再起動することによるサービスの復旧を Re-active な手法という．このような対処的な対応はクラッシュなどのサービスの停止に対して再起動を実行することで，その障害から復旧することが可能である．しかし，この手法では復旧までに時間的コストが多くかかる問題がある．これはクラッシュ時にデータの不整合が発生していた場合には問題が発生するため，事前にこれに対する対応が必要となるためである．また，Pro-active のみを対応するのであれば再起動直前にデータを退避するなどの処理が可能であるが，Re-active な若化ではそのタイミングが存在しない．よって，データが常に継続的に保持される状態にする必要がある．

3. 目的

本研究では，DBMS においてソフトウェア若化の利用を前提とした基本機能を持たせる手法を提案する．提案手法では，対象の DBMS を Memcached[4] として，DBMS 特有のソフトウェア若化の問題点である性能劣化を抑え，その際に問題となるデータの損傷を未然に防ぐ．この機能においては性能劣化の低減とデータ損傷対策の 2 点に重点を置きそれぞれ DBMS の特徴や Slab allocator の特徴を考慮した手法を考案する．

3.1 性能劣化の抑制

DBMS に対してソフトウェア若化を行う上で性能劣化を抑制するためにはデータベース情報の再構築を高速に行う必要がある．特にインメモリ DBMS であればデータベース情報を保持してデータを解放しない手段が必要となる．データの再構築を高速化する上でデータをディスク上に退避させる手法はディスク I/O の時間的オーバヘッドが大きいことから現実的ではない．

そのため，再起動後までデータベース情報を保持し続けて性能劣化の抑制を目指す．

3.2 Pro-active な若化時のオブジェクト管理

Memcached のメモリ管理手法である Slab allocator には Slab calcification と呼ばれる問題が存在する．これは確保された slab が偏った状態になるものであり，Pro-active

な若化によって対処可能なソフトウェア老化である．しかし，この問題はデータを保持する手法とは相性が悪く，データが保持されれば偏りも維持されてしまう．

これに対処するため，再起動時に slab の一部を解放することで偏りの軽減を行う．

3.3 Re-active な若化時のデータ整合性保証

データの保存を行う上でそのデータの整合性保証が必要である．これは Re-active なソフトウェア若化を行う際にその直前の障害でデータに不正な書き込みが発生する可能性があるためである．メモリリークなどの問題によって障害が発生する際にはメモリ上の不正な値を参照する可能性が十分にあり，その書き込みを防ぐために保護を行い，また異常の発生を検出するための機能を搭載する．これによりデータの整合性を保証しながら再起動後に引き続くことが可能となる．

4. デザイン

提案手法では Pro-active のための性能劣化対策と，Re-active のための整合性保証の 2 つの要素で構成されるため，それぞれの設計を記述する．

4.1 Pro-active のための手法

Pro-active なソフトウェア若化で重要となるのは性能劣化対策であり，これを中心として slab calcification に対しても効果のある若化が必要である．本節ではこれらの対処手法をそれぞれ述べる．

4.1.1 データベース情報の保持

ソフトウェア若化後の性能劣化対策のためにメモリをデータベース情報とそれ以外のデータとに分類し，データベース情報に分類されたデータのみを保持されるメモリ上に保存する．これにより再起動後に必要なデータは常に保持されるメモリ上に存在し，再起動後にこれを取り出すだけで再利用が可能となる．

本手法ではデータベース情報とそれ以外のデータに分類する際に主にメモリ確保後の初期化以外に変更を行うかどうかを基準とする．データ構造を保持するメモリと異なり，key と value については一度確保し保存した内容については解放するまで変更することがないため，これにより分けることが可能である．

データの再確保の流れを図 1 に示す．再起動後にはデータの再利用が可能か確認し，異常のない item を再利用する．

4.1.2 Slab calcification の対策

データベースの保持を行いながら slab calcification に対応するために一部 slab の解放を行う．実際に利用される手法として Twitter ではランダムに slab を解放して再利用する手法 [5] を考案している．Twitter の手法であれは

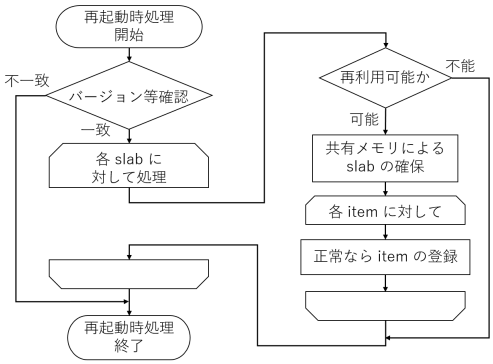

図 1 データベース保持のフローチャート

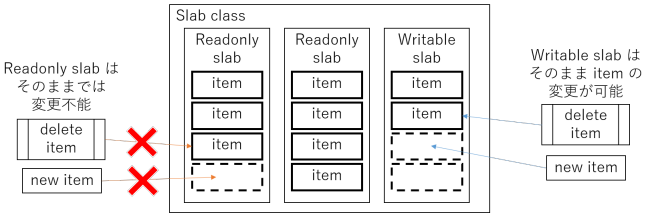

図 2 Readonly slab の構成

状況による偏りが発生しにくく，リクエストの偏りに常に対応した slab の確保を行える．

本手法ではこれを参考に，再起動時に各 slab class からいくつかの slab の解放を行う．これにより解放した分だけ slab の再分配が行えることにより，再起動後のリクエストの状況に応じた確保状況の調整が可能となる．

4.2 Re-active のための手法

Re-active なソフトウェア若化ではクラッシュや書き込み途中での終了を考慮した仕組みが必要となる．そのために保存されたデータの整合性保証のための整合性を確認する処理と，データの不正な変更を防止する．

4.2.1 Readonly Slab

データの不正な変更を減らすために通常時には書き込みが不能な readonly slab を利用する．各 slab の中で書き込みが完了したものを読み込み専用とし，再び変更の処理が必要になるまでの間に不正な上書きが発生しないようにする．OS のバッファキャッシュに対して読み書きを制限する手法 [6] を参考にする．読み込み専用の readonly slab ではデータベース情報の部分だけを読み込み専用とする．そのため，データ構造の変更は常に可能となり処理の並列性への影響を最小限に抑える．

各 slab は書き込み不能な readoly slab と書き込み可能な writable slab に分ける．これらの slab class 内の構成を図 2 に示す．通常の状態では writable slab は各 slab class に一つだけ存在し，item の確保とその中に値を書き込む処理はこの slab から行う．

4.2.2 Multi Checksum

Readonly slab では不正な書き込みを削減することはで

```
typedef struct _stritem {
    ...

    /* 本手法のために分離したメンバ変数 */
//  int         nbytes;   /* value のサイズ    */
//  uint8_t     nsuffix;  /* flag のサイズ     */
//  uint8_t     nkey;     /* key のサイズ      */
//  union {               /* value などの情報 */
//      uint64_t cas;
//      char end;
//  } data[];

    /* 追加したメンバ変数 */
    int         slab_num; /* slab の No.    */
    struct _stritem_main *datas;
           /* 対応した item_main へのポインタ */
} item;
```

図 3 構造体 item の変更点

きるが，writable slab に対するデータの破損を検知することはできない．この対策として Multi checksum を利用する．Multi checksum ではチェックサムを複数利用する手法で，各 item に対してその item の共有メモリに保持している情報から計算したハッシュ値を共有メモリ上の複数の配列に保存する．

複数のチェックサムを持つことでのいくつかに異常な値が含まれていても正しいハッシュ値が一つでも保存されていれば item の再利用が可能である．これは不正なアクセスなどによってハッシュ値のうちいくつかが変更された際に残りのハッシュ値が正しければそのデータが正しい可能性が高いことを利用している．

5. 実装

本研究では提案方式を Memcached 1.4.25 に実装している．データ保持手法とデータの整合性保証のために手法に分けてそれぞれ実装した．以下にその実装の詳細を記述する．

5.1 データ保持

データ保持については POSIX Shared Memory を利用する．共有メモリは確保したソフトウェアが終了してもデータを維持するうえ，メモリ上にデータを維持してディスク I/O を減らせるためこれを利用する．

本手法ではデータベース情報の保存を共有メモリに対して行う．これにより障害による予期せぬ再起動後でもデータを再利用できる．

5.1.1 データベースの分離

Slab からデータベースの情報を分離するうえで item の構造を変更する．本手法では slab allocator で利用される構造体 item に対して図 3 の通りに変更を加え，データベース情報は構造体 item_main に分離する．

Slab class は slab 確保時に構造体 item に利用するヒープメモリ領域の確保と，構造体 item_main の確保に利用する共有メモリ領域の確保を行う．このうちヒープメモリ領域を構造体 item のサイズごとに分割し，共有メモリ領域をその slab class が担当するメモリサイズごとに分割する．これは図 4 の通りにそれぞれ分割され一つの item を構築する．この中で処理に必要なデータなどはヒープメモリ領域に保存し，データベースは共有メモリ領域に保存する．

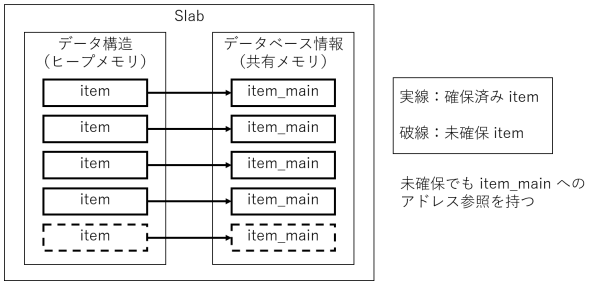

図 4 本手法での slab の構造

この際に構造体 item の datas メンバに，ペアとなる構造体 item_main へのアドレスとこの item がどの slab に属するかを示す整数値を保存する．構造体 item_main へのアドレスを持たせることで構造体 item のメンバと構造体 item_data のメンバへのアクセスを多くても 1 度のアドレス参照を追加するだけで可能とする．これはプログラムへの追加記述の量も少量で済むため移植での利点でもある．

このメモリ確保方法により，再起動後まで残したいデータベース情報は共有メモリ上に残り，そうするべきでないデータは再起動時に解放されるヒープメモリに保存されることとなる．

5.2 整合性保証

共有メモリでの情報の保存においてそのデータに異常な変更が加えられていないことの確認は必要であり，データの破損を可能な限り防いだうえで再起動後に共有メモリ上のデータが破損していないことを確認する必要がある．

5.2.1 Readonly Slab

共有メモリのデータの保護として書き込み制限を掛けることが可能である．これはメモリ全体に対して存在する書き込みの許可を利用する機能を利用する．

書き込み制限の取り方として各 slab class から slab のうちの一つを書き込み可能な writable slab とし，それ以外を書き込み許可のない Readonly slab としている．

新たな slab を確保した場合はそれを writable slab として，直前まで writable slab だったものは readonly slab にする．この時に直前まで writable slab に含まれた item が書き込み中だった場合はその書き込み後にその slab を読み取り専用にする．実験の中ではこの実装が完了しておらずマルチスレッドに対応していないため以降の実装や実験ではシングルスレッドで処理を行う．

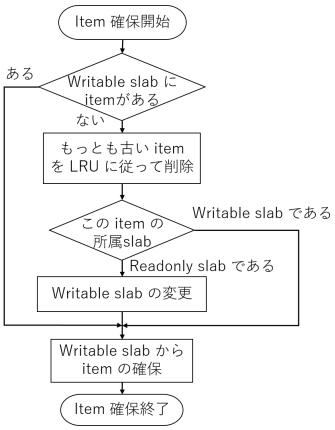

図 5　Item 確保のフローチャート

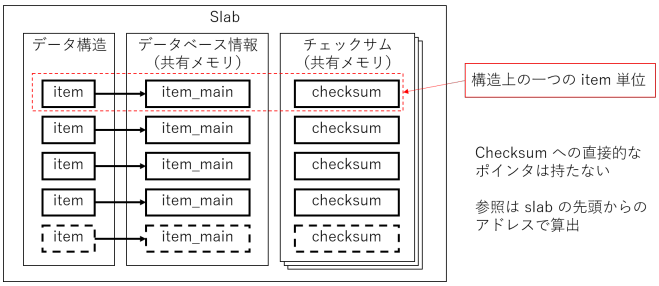

図 6　Multi checksum を含めた slab の構造

Item 確保時の変更の流れは図 5 のフローチャートに従う．まず，writable slab に未確保の item があればそれを確保して割り当てる．writable slab がすでに全 item を割り当てている場合は新たな slab を確保する．この時に，新たな slab を確保出来ればその slab を writable slab にする変更を行う．できなかった場合は既存の slab の中から最も古い item のある slab を次の writableslab に変更する．

5.2.2　Multi Checksum

チェックサムでは slab 同様の共有メモリ確保を行い，複数のチェックサム配列を作成する．この配列の数は設定可能としており，一つの配列であれば最小の 112Byte の item のデータに対して 32bit(4Byte) と 約 3% の情報を付加する．本実装では配列を 3 つ利用し，合計 12Byte のチェックサムを各 item に持たせている．Multi checksum を含めた slab の構造を図 6 に示す．

チェックサムの更新にかかわる関数はそれぞれ適切な関数に組み込むことで実装する．チェックサムの更新は共有メモリ内にデータを保存した直後にするべきであり，その時点で発生する関数 do_item_link() にチェックサムを更新する関数 update_item_hash() を追加する．これに対してチェックサムの破棄は item の破棄時であり，これには関数 slabs_free() 内にチェックサムを無効な値にする関数 delete_item_hash() を追加する．

チェックサムによる整合性の確認は再起動後の slab の再確保時に item が有効であるかの判定としてのみ利用する．

6.　実験

本章では，提案手法を評価するために行った実験について記述する．

6.1　実験環境

実験では 2 台のマシンを用いる．1 台の物理マシンをサーバマシン (Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, Memory 16GB) として Memcached を起動し，もう 1 台の物理マシンをクライアントマシン (Intel(R) Xeon(R) CPU E31270 @ 3.40GHz, Memory 16GB) としてベンチマークソフトを実行する．サーバマシンではアプリケーションは Memcached のみを起動し，メモリ容量から 8GB のメモリを割り当てて計測を行う．また，機能の制限のためオプションにより起動するスレッドは一つだけとして実行した．各マシンは Ubuntu 14.04 LTS を OS としてそれぞれのアプリケーションを実行した．各実験ごとでマシンを再起動し，共有メモリを解放した状態から実験を開始している．

6.2　オーバヘッド

6.2.1　目的

Memcached の標準機能であるデータの挿入・更新を行う際の時間的コストを評価するため，そのオーバヘッドを計測する．本実験では毎秒ごとの処理結果のうち，処理が完了したリクエスト数からそのスループットを算出する．計測されたオーバヘッドが小さければ手法を利用していない環境に本手法を取り入れることが容易となる．また，結果が十分に小さければ運用内での利用でも稼働率の上昇が見込める．

6.2.2　実験方法

通常の Memcached(origin)・共有メモリによる Pro-active な手法を考慮した Memcached(shm)・プロテクトを付けた本手法の Memcached(shm+protect) の三つについてそれぞれ計測を行った．計測ではサーバマシンでそれぞれの Memcached を起動しクライアントマシンでベンチマークを起動する．実験ごとにサーバマシンは再起動を行い共有メモリを解放した状態から開始した．

ベンチマークでは cloudsuite[7] の data caching を利用する．これは Twitter で利用されるシステムを参考にした Memcached のベンチマークアプリケーションである．設定では Memcached が 1 つのスレッドで処理を行うのに対し data caching の worker 数を 4 とし，常に Memcached が処理を行うような，負荷がかかった状態で計測を行う．クライアントサーバが送信するデータは get と set の比を 8:2 として 30 倍にスケールしたデータセットを利用して メモリ内に十分にデータがたまる状態で計測を行う．実験は Memcached 上にデータを貯めるための warm up

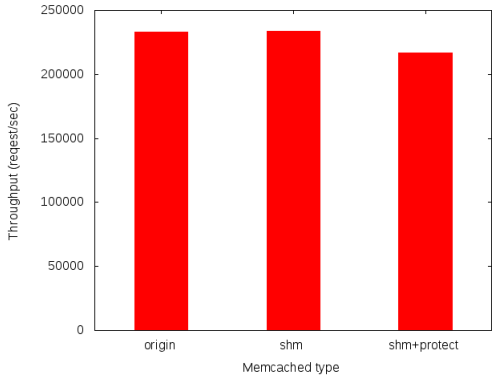

図 7　各手法でのスループット

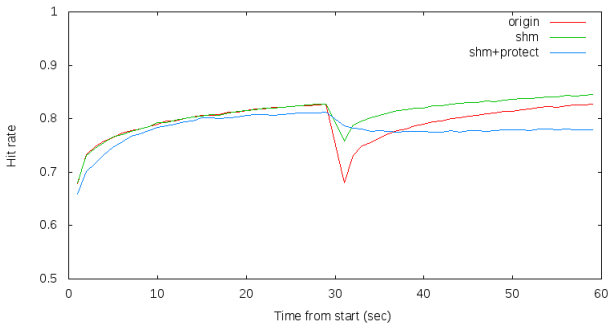

図 8　再起動前後の hit 率の推移

を行ったのち 30 秒間の計測によってスループットを算出する．

6.2.3　実験結果

各手法ごとの開始後 30 秒間のスループットの平均が図 7 である．shm は slab 確保時の処理は少量あるが，item の確保に対して追加された処理がほとんどないために origin との差は見られなかった．shm+protect については origin の 93% のスループットを維持している．こちらは origin と比べて item の確保時にチェックサムの計算やその登録を追加で行っている上，writable slab の変更なども含まれるために性能の低下は見られている．

6.3　再起動後の性能劣化

6.3.1　目的

再起動後までデータを保持することによって生まれる性能劣化の低減を評価する．この実験ではデータが消失することによって生まれる性能劣化と比較して，本手法を利用した際の性能劣化の程度を計測し，その低減を確認する．

6.3.2　実験方法

実験では前述の実験と同じく data caching を利用する．origin, shm, shm+protect の 3 手法についてそれぞれ実験を行う．これらに対して warm up を行い，再起動後に再び同じベンチマークを行った際の hit 率を比較する．

実験で利用する data caching の設定はオーバヘッドの実験と同じ内容で行う．1 度目の計測では起動したばかりの各 Memcached にたいして 30 秒間 hit 率を計測しながらデータを蓄積する．その後，各 Memcached を再起動し，再び同じ設定で 30 秒間のベンチマークを行う．

6.3.3　実験結果

図 8 が結果の hit 率の推移であり，再起動前の hit 率と比べて再起動後の変化にはそれぞれ違いが見られる．origin 以外では再起動後の hit 率の推移が 初回の起動後の hit 率の推移と大きく異なっている．一方で初回起動時は推移の形に大きな差がないことが確認できる．

再起動直後の hit 率 から shm と shm+protect では再起動直後の性能劣化を抑えられていることが確認できる．

表 1　3 つのフェーズ内容

内容	phase 1	phase 2	phase 3
送信 item サイズ (Byte)	512	1024	2048
送信 item の種類数 (種類)	160000	80000	40000
合計容量 (MByte)	80	80	80

origin では 再起動時にキャッシュしていたデータを解放しているために，再起動後の推移が一回目の起動時と同様の曲線を描いている．これは大きな性能劣化を示しており，利用するメモリ容量が大きければ大きいだけこの低下している時間は長くなる．

shm が origin より高い hit 率を保っているのに対して，shm+protect では origin と比較して全体的に hit 率が低下している．再起動後の 30 秒経過時点で shm の hit 率は origin と比べて 102% と向上しているが，shm+protect では 94.2% に低下している．これは各 item が共有メモリへのアドレスやチェックサムなどを保持するために容量が増え，同じメモリ容量内で保持できる item 数が低下していることが原因と考えられる．

6.4　Slab Calcification 対策性能

6.4.1　目的

Slab calcification による偏りを本手法によって軽減できることを確認する．Slab calcification 対策として加えた slab の解放により実際に hit 率の改善がみられることを確認する．

6.4.2　実験方法

Slab calcification が発生した状態を作るために，本実験では表 1 の 3 つのフェーズに分けて送信するデータサイズを変えながらデータの hit 率を計測する．3 つのフェーズではそれぞれ 512, 1024, 2048 Byte のデータを送信する．送信するリクエストでは get と set をそれぞれ 8:2 となるようにし，10000 リクエストごとに hit 率を確認する．合計 300000 リクエストを送る作業を 1 つのフェーズとしてデータサイズを変更しながら送信する．

Memcached は origin と shm+protect を比較する．それぞれ実行時はメモリ容量をデフォルトの 64MB に設定

し各フェーズの間で再起動を行う．origin に関しては再起動を行わない場合 (origin+noreboot) も計測しそれぞれの hit 率の変化を確認する．

6.4.3 実験結果

1 つ目のフェーズは図 9 であり，メモリ容量の上限までデータが入っていないためそれぞれの hit 率の推移に大きな差は出ていない．この部分では 512Byte のデータが入る slab が複数確保されメモリを消費させている．

2 つ目のフェーズは図 10 であり，origin+reboot 以外の hit 率が途中で上昇を止めている．これはこの時点でこの slab class に割り振りが可能な slab を割り振り切ったことを示している．shm+protect が最も早く 160000 リクエスト程度の時点で上限に達しているが，これはメモリのオーバヘッドによるものであると考えられる．shm+protect では共有メモリ部分へのポインタやチェックサムなどのメモリ確保が item の個数だけ増加している．これにより 1 つ目のフェーズで余計な slab を確保し，2 つ目のフェーズでも origin よりは少ない量の item しか確保できなかったと考えられる．

3 つ目のフェーズは図 11 であり，shm+protect は origin より高い hit 率になっていることが確認できる．これは 2 つ目のフェーズで確保していた slab の中からいくつかの slab を再利用できるようになったためである．origin ではこの slab class に割り当てられる最初の slab のみが利用された結果低い hit 率となっているが shm+protect の一部の slab 解放によってこの性能劣化を軽減できている．実験では set するデータのサイズが急激に変化する極端な例であるため origin のみについては hit 率が 70% 以上まで上昇している．

6.5 クラッシュ耐性

6.5.1 目的

プログラムがクラッシュした際にメモリへの異常な書き込みが発生する場合がある．再起動後にその異常な値が維持されることは問題となり得るため，これを可能な限り回避するための手段として提案手法の有用性を確認する．

6.5.2 実験方法

実験は shm と shm+protect とで異常な書き込みがあった際の結果を確認し比較する．これまでの実験と同じ環境において shm と shm+protect にそれぞれ 256Byte のデータを 10000 個保存し，その後これらのデータが保存されている共有メモリ内のランダムなアドレスに 32bit 分のランダムな数値を書き込むことで異常を挿入する．この時アプリケーションがクラッシュするか否かに関わらず Memcached を再起動し，再起動後に異常なデータが残っているかどうかを確認する．初めに送信するデータの量はメモリに余裕がある状態にすることで参照されない破損したデータが残っているかどうかを確認できる状態にする．

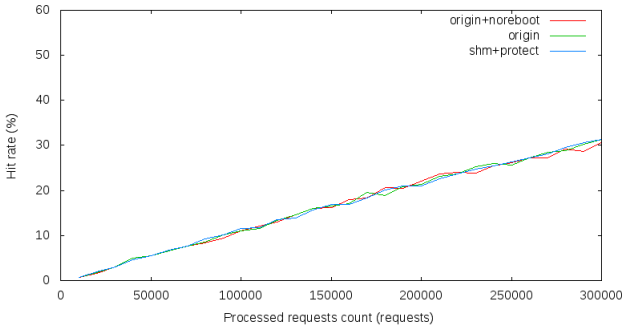

図 9 Slab calcification 時の hit 率推移 (phase 1)

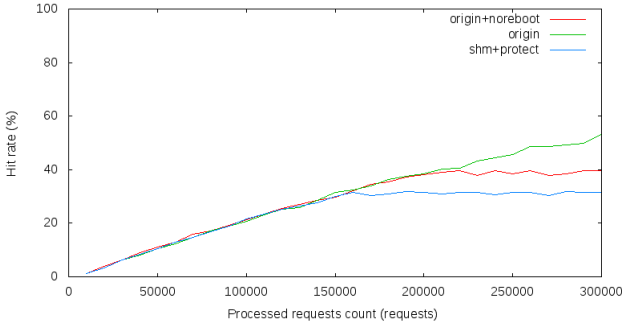

図 10 Slab calcification 時の hit 率推移 (phase 2)

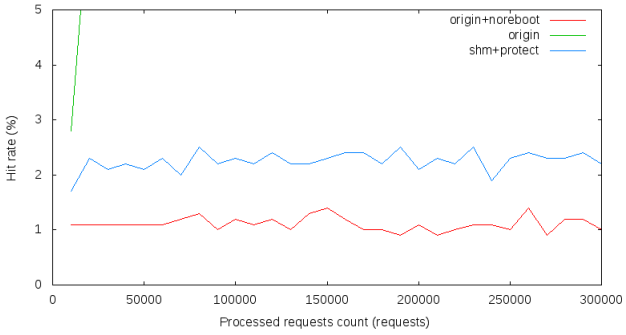

図 11 Slab calcification 時の hit 率推移 (phase 3)

6.5.3 実験結果

それぞれ 10 回ずつ異常挿入を行った結果について，種類ごとの回数が表 2 である．shm では key に異常な値が書き込まれたために参照はできないがメモリを消費する item が発生したほか，データの内容自体が書き変わったために不整合が発生したデータも存在した．特にデータの不整合は 30% を超える確率で発生しており，50% の確率でメモリリークの原因となる未解放な item が発生することが確認できる．それに対して shm+protect では 60% が例外を発生させてクラッシュした．これは Readonly slab への書き込みが発生したためであり，これらのデータが守られていることが確認できた．残りの中では 30% がチェックサムへの書き込みを行ったことが確認されたが，これに

表 2　クラッシュ結果

結果	shm	shm+protect
異常データ発生	5 回	—
データの異常発生	3 回	—
データの消失のみ	—	1 回
異常なし	2 回	9 回 (うち 6 回はクラッシュ)

よる item の消失はなく再起動後も正常に動作した．残りの 10%ではデータへの書き込みがあったためか item の消失が確認されたが，これが異常なデータとして残ることはなく再起動後も続けて動作が可能である．

結果から本手法による対策を行っていない場合は 30% の確率でデータの異常が発生するのに対して，保護とチェックサムによって異常の発生をほぼ抑制できることが確認できた．また，データの消失についても保護がない場合には合計 80% の確率で破損したデータが発生しているのに対して，保護により 10% まで発生率が下がっている．

7. 関連研究

Facebook が実際に運用している DBMS である Scuba[3] において高速で全体の性能の低下を抑えた再起動を行う仕組みを利用している．ここで利用している DBMS では，そのアプリケーションのアップデートのための再起動は多くの時間がかかっており問題となっている．この時に最も時間を必要とする処理は再起動後のデータのメモリへの展開とそのためのディスク I/O である．これを削減するために再起動処理開始時にメモリ内のデータを構造を圧縮しながら共有メモリに移動する．再起動後にこれをヒープメモリに再構築することでディスクアクセスを減らしている．この手法では障害による再起動を考慮しておらず，再起動のコマンドの実行に対してのみ性能を発揮する．そのため，Pro-active なソフトウェア若化を行えない．

Warm-Cache Reboot[8] は再起動による影響を最小限に抑えるためにゲスト OS について，ページキャッシュを維持したままの再起動を行う手法である．Virtual Machine(VM) 上のゲスト OS の再起動時に解放されるゲスト OS のディスクアクセスのキャッシュは再起動後も同じ内容を利用するが，これは解放されるため再起動後まで利用することができずこれが再起動後の性能の低下をもたらしている．この手法ではこの影響を低減するためにディスクアクセスによってキャッシュされた内容を VMM で保持し再起動後に再利用することで性能劣化を抑えている．この手法はゲスト OS 上のページキャッシュのみを保持するため，ディスク I/O を利用する DBMS への応用は可能であるがインメモリ DBMS への性能劣化の抑制には利用できない．

NVMcached[9] は NVRAM（Non-Volatile RAM）上に item とチェックサムを保持してクラッシュ後も item を利用可能にする手法である．しかし，この手法ではチェックサムの破壊が伴うクラッシュに対して対応していない上，NVRAM に寄るハードウェア上の制約がある．

8. まとめ

本研究では，Software Rejuvenation を指向した DBMS の設計手法を提案した．Software Rejuvenation を指向するにあたって，Pro-active なソフトウェア若化と Re-active なソフトウェア若化の双方に対応するための手法を考案した．Pro-active なソフトウェア若化への指向としてデータの保持と slab calcification への対応の手法を提案した．データの保持では共有メモリを利用してメモリ上のデータベース情報のみを分離して再起動後まで保持する機能を提案し実装した．Slab calcification の対応としては保持した slab の一部を再確保しない手法を提案，実装した．Re-active なソフトウェア若化への指向として Readonly slab と Multi checksum によるデータの整合性保証を提案した．Readonly slab ではメモリの書き込み制限を利用して異常な書き込みを防止する実装を行った．Multi checksum では Readonly slab でも保護しきれない異常な書き込みを再起動後に検出して不正な値を再利用しない手法を実装した．以上の実装を行った Memcached を用いて実験を行い，小さなオーバヘッドのみで再起動後の性能劣化の軽減と，クラッシュ時の異常な値の保持の対処を行えることを確認した．

参考文献

[1] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proc. of the IEEE 25th Int'l Symp. on Fault-Tolerant Computing (FTCS '95)*, pp. 381–390, 1995.

[2] D. Cotroneo, R. Natella, R Pietrantuono, and S Russo. A survey of software aging and rejuvenation studies. In *J. Emerg. Technol. Comput. Syst.*, pp. 8:1–8:34, January 2014.

[3] A. Goel, B. Chopra, C. Gerea, D. M ata ni, J. Metzler, F. U. Haq, and J. Wiener. Fast database restarts at facebook. In *Proc. of the 2014 ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD '14)*, pp. 541–549, 2014.

[4] Memcached. 入手先 ⟨http://memcached.org/⟩.

[5] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, Zhenlin Wang. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 57–69, Santa Clara, CA, 2015. USENIX Association.

[6] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G.Rajamani, and D.Lowell. The rio file cache: Surviving operating system crashes. In *Proc. of the 7th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, pp. 74–83, 1996.

[7] Cloud suite. 入手先 ⟨http://cloudsuite.ch/⟩.

[8] K. Kourai. Fast and correct performance recovery of operating systems using a virtual machine monitor. In *Proc.*

of the 7th ACM International Conference on Virtual Execution Environments (VEE '11), pp. 99–109, 2011.

[9] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, p. 18. ACM, 2016.