永続メモリのリカバリ・コード検査を容易にするクラッシュ・インジェクタの提案

坂本 颯一朗[1,a)]　鈴木 慶汰[1,b)]　河野 健二[1,c)]

概要：永続メモリは，バイト単位のアクセスを可能にし，データ構造を永続化できるため，多くのアプリケーションで用いられるようになっている．しかし，永続メモリを扱うアプリケーションを正確に記述することは難しく，結果として Correctness Bug や Performance Bug を含むことが多い．これらのバグは Crash Consistency Bug と呼ばれる．クラッシュによってデータ更新等の操作が中断された結果，データの一貫性が失われるというバグである．これらのバグを発見・修正するため，既存の研究では主に静的解析を用いてより広範囲のコード領域をカバーし，より短時間で効率よく検証を行う手法を提案している．静的解析を用いてこのバグを検証する場合，例えば非同期処理に伴うタイミング依存のバグなど全てを発見することは難しい．本論文では，永続メモリにおける Crash Consistency Bug の検証を動的に行うクラッシュインジェクタというツールを提案する．実行時にクラッシュを起こす関数をランダムに挿入することで，データの一貫性を回復するリカバリコードを呼び出す．動作の結果をもとに，リカバリコードに関わる Crash Consistency Bug が存在するかを確認する．クラッシュインジェクタを用いて，PMDK のサンプルデータ構造に対して意図的に挿入したバグを検出可能であることを確認した．

1. はじめに

永続メモリと呼ばれる不揮発性メモリが広く用いられるようになっている．たとえば，Intel 社の Optane Persistent Memory などが実用化されている．永続メモリの特徴は，従来の DRAM と同様にバイト単位でのアクセスが可能であり，扱うデータ構造をそのまま永続化できるという点である．ここでいう永続性とは，電源が遮断されても保存されたデータが失われることはないという性質である．そのため，永続メモリを用いたプログラミングでは，不慮の電源遮断等により永続データの一貫性が損なわれる懸念がある．例えば線形リスト構造を永続メモリに保持している場合，新たな要素を追加する操作の最中に電源が遮断された場合，ポインタが本来想定しない宛先に繋がれたまま永続化される可能性がある．すなわち，クラッシュが起きるタイミングによっては永続メモリ上のデータの一貫性が失われる可能性がある．そのため，任意の時点でクラッシュが発生しても，永続メモリ上のデータに矛盾がないこと保証する必要がある．クラッシュが発生しても永続メモリ上のデータに矛盾が発生しないようにすることを Crash Consistency が保証されているという．

永続メモリでは，Crash Consistency を損なわないよう write-ahead logging を用いてデータを管理するのが一般的である．Intel 社の提供する PMDK[4] というライブラリでは，アトミックに永続化すべき操作をトランザクション内で実行するようにしている．トランザクション領域では，まず更新する永続メモリ上のデータをログに書き込む．その後クラッシュ等の遮断が起きた場合には，ログを参照して内容を巻き戻すことで Crash Consistency を維持する．そのため，もしログへの書き込みが正しくなされなかった場合には巻き戻す情報が存在せずに Crash Consistency Bug となる．

永続メモリにおけるバグの検出を行う研究は例えば，[1]，[2]，[3]，[5]，[6]，[7] が挙げられ，Crash Consistency Bug は永続メモリを扱う上で重要な課題となっている．これらは，膨大なコード領域でバグの検出を行うために，ユーザのアノテーションを用いた解析や，静的解析を用いて効果的な絞り込みを行いつつバグの検出を行う．例えば，PMDebugger[2] では実際のアプリケーション（memchached）にて新しく 19 のバグを発見し，PMDK においては 2 つのバグを新しく発見したと報告している．

Crash Consistency の検査の問題点として，クラッシュ・電源遮断が起きて初めてリカバリコードが実行されるとい

[1] 慶應義塾大学
Keio University
[a)] soichiro.sakamoto@sslab.ics.keio.ac.jp
[b)] keitasuzuki.park@sslab.ics.keio.ac.jp
[c)] kono@sslab.ics.keio.ac.jp

う点がある．そのため，通常のソフトウェアテストを実行しても Crash Consistency Bug を検査することはできない．PMDK においてはトランザクションの際にログに正しく書き込めているか，という点を通常の検査で発見することはできない．

そこで，本論文では永続メモリを対象としたクラッシュインジェクタを提案する．プログラム中でクラッシュを作為的に起こすことによって，リカバリコードを呼び出して Crash Consistency の検査を行うことを容易にする．作為的なクラッシュの挿入による利点として，特定のコードに絞った検査を効果的に行うことができることができる点が 1 つ挙げられる．またポインタ解析などの静的解析に依存しないため，大規模なソフトウェアやマルチスレッディング，非同期処理などを行うソフトウェアに対しても適用することができる，という利点もある．既存の研究では，基本として静的解析に依存した検査を行うため，以上のような点を達成するのは困難となっている．

PMDK のサンプルデータ構造に対してバグを用意し，それらをクラッシュインジェクタで検出できるかどうかを確認した．用意した 8 つのバグのうち，6 つは正しく発見することができ，残り 2 つに関してはクラッシュインジェクタのみでは発見できない種類のバグであると結論づけることができた．

以下に本論文の構成を示す．2 章では，永続メモリにおけるプログラミング手法およびバグについて説明を述べる．3 章では，クラッシュインジェクタの概要とその実装について説明する．4 章では，評価を行い，クラッシュインジェクタを用いた検査が Crash Consistency Bug を発見するのに有効であることを示す．5 章では，関連研究の紹介を行う．6 章では，本論文のまとめを行う．

2. 永続メモリにおけるプログラミング手法

2.1 永続化を行う操作

永続メモリを扱うプログラムは，`store`/`flush`/`fence` を正しく用いることで永続性を保証する．`store` は，メモリロケーションやデータオブジェクトの書き換えを行う．次に `flush` の操作を行い，キャッシュライン上にある更新済みのデータを消去し，永続メモリに書き戻すことでメモリロケーションやデータの永続化を行う．最後に `fence` の操作を行う．この操作によって，`flush` によるデータの永続化が正しい順序で完了されることを保証する．

Intel 社の提供する PMDK というライブラリでは，これらの操作を簡単に扱うための関数を提供する．libpmemobj というライブラリは，プログラマが実用的に利用できる関数を提供している．アトミックな操作のみで永続性を保証する操作例を**図 1** にあげる．`D_RW` というマクロは，`DIRECT_READANDWRITE` の略で，永続メモリのアドレス領域に直接アクセスしてデータの読み書きを行う関数である．

```
1 void insert(struct llist *prev){
2   // generates a new linked list element
3   struct llist *new = llist_new();
4
5   // updates the pointer
6   D_RW(prev)->next = new;
7
8   // makes the updated pointer persistent
9   pmemobj_persist(D_RW(prev)->next,
10                  sizeof(*D_RW(prev)->next));
11 }
```

図 1 各データに対して毎回永続化を行う例

```
1  void insert(struct llist *prev){
2    // generate a new linked list element
3    struct llist *new = llist_new();
4
5    TX_BEGIN(pop){
6      // write to undo log
7      TX_ADD(prev);
8
9      // update the pointer
10     D_RW(prev)->next = new;
11   }TX_END
12 }
```

図 2 トランザクションを用いたプログラムの例

永続メモリ上の更新可能なポインタを `D_RW` で取得可能であると言い換えることもできる．`pmemobj_persist` という関数は flush/fence をアトミックに行う操作を提供し，直前に更新したアドレス領域を永続化する関数として用いられる．ただし，現在のハードウェア特性ではアトミックな操作を保証できるサイズが 8 バイトに制限されている．そのため，`pmemobj_persist` のみを用いて永続化を行う場合，データサイズを考慮して関数を用いる必要がある．PMDK では，それらのデータサイズを考慮せずに永続化を行う手法としてトランザクションを提供している．**図 2** にその例をあげる．トランザクション領域は `TX_BEGIN` と `TX_END` で囲まれており，この中で永続メモリ上のデータを扱うクリティカルな操作を行う．まず，`TX_ADD` を用いて更新するデータをログに書き込む．`TX_ADD` では `pmemobj_persist` を代替する操作も行うため，トランザクション内で明示的に他の関数を呼び出して永続化を行う必要はない．その後トランザクション内でデータの更新を実行し，何もなければ操作は完了される．トランザクション内でクラッシュが起きる場合には，ログに書かれた古い情報をもとにロールバックを行うことで Crash Consistency を回復する．PMDK では，以上のようにして write-ahead logging を行い Crash Consistency を保証している．

2.2 永続メモリを扱うプログラムにおけるバグ

永続メモリを扱うプログラムに特有のバグとして，Crash Consistency Bug が挙げられる．それを細分化すると Correctness Bug と Performance Bug の 2 種類に分けることができる．

Correctness Bug は，クラッシュ時に Crash Consistency を失わせるバグである．先ほど示したコードを例にとって説明を行う．アトミックな操作を行うコード (図 1) の場合，`pmemobj_persist` の記述がない場合が Correctness Bug に当てはまる．永続化を行う操作が実行されないため，クラッシュなどが起きた際に永続化されるべき情報が失われてしまう．その状態で永続化されるはずだったデータを参照した場合，例えば前後のデータのバージョン情報が異なる場合に本来参照されるべきでない解放後のアドレス領域が読み込まれてしまう可能性がある．この場合，クラッシュによって一貫性が失われたと考えることができ，Crash Consistency Bug として報告される．トランザクションを用いる場合 (図 2) でも同様である．トランザクション領域で `TX_ADD` の記述がない場合，データの更新前の情報がログに保存されずに操作が完了する．そのトランザクション内でクラッシュが起きた場合，ログ内にはさらに古いバージョンの情報があるか，もしくは何も情報が保存されていない場合が考えられる．どちらの場合もロールバックを行なった際に不適切な情報を参照することになり，データの一貫性を失う結果となる．

Performance Bug は，クラッシュ時に Crash Consistency を失わせることはないが，冗長な操作によってパフォーマンスを損なう可能性のあるバグである．例えば，一度永続化したメモリロケーションを再度永続化してしまうという操作である．また，永続化する必要のないデータを永続化してしまうというバグが起きる可能性もある．つまり永続性の保証は可能であるが，その手法が不適切でありパフォーマンスを低下させる要因になるバグである．

これらの Crash Consistency Bug は検出するのが困難である．その理由は，1 つはクラッシュ時に初めて検出可能であるためである．クラッシュが起きない限りにおいては見かけ上は正しく動作し永続化されているように見えるため，通常のテストケースでバグを検出することはできない．もう 1 つの理由としては，ある実行パスでは冗長な操作であるか，もしくは適切に永続化されない場合でも，別の実行パスを通る場合には結果として適切な操作となってしまいバグがマスクされるためである．例えば条件分岐のうちいずれかにのみ影響するバグが含まれる場合，実行パスによってはそのバグの領域を通らない可能性もある，という場合である．

```
1  int
2  btree_map_insert(PMEMobjpool *pop, TOID(struct btree_map) map,
3      uint64_t key, PMEMoid value){
4
5    struct btree_map_node_item item = {key, value};
6    TX_BEGIN(pop){
7      ...
8      /* position at the dest node to insert */
9      int p = DEST_NODE;
10     ...
11     btree_map_insert_item(dest, p, item);
12   }TX_END
13
14   return 0;
15 }
16
17 /* inserts and makes space for new item */
18 static void
19 btree_map_insert_item(TOID(struct tree_map_node) node, int p,
20     struct tree_map_node_item item)
21 {
22   TX_ADD(node);
23   /* call crash_function */
24
25   if(D_RO(node)->items[p].key != 0){
26     memmove(&D_RW(node)->items[p + 1], &D_RW(node)->items[p],
27         sizeof(ITEM_SIZE));
28     /* call crash_function */
29   }
30   ...
31 }
```

トランザクション領域が実行の対象

トランザクションの内部関数も考慮

図 3 btree_map.c

3. PM Crash Injector

3.1 クラッシュインジェクタ

本論文では，永続メモリを扱うプログラムにおけるリカバリコードの検査を行う，クラッシュインジェクタを提案する．クラッシュインジェクタおよびバグの検出の流れを**図 4** に示す．クラッシュインジェクタが行うのは図 4 中で緑で示される部分であり，プログラム中にクラッシュを意図的に挿入することである．実際にクラッシュを起こすことでリカバリコードを呼び出し，その動作の検証を容易にすることが目的である．クラッシュインジェクタを起動するごとにランダムな位置でクラッシュが起きるようにすることで，より多くのパターンでの検査を可能にする．機能としてはクラッシュをランダムに起こすのみであるが，それだけではバグの検証を行うことができないため， 本論文ではバグが存在するかの検査を行う例も示す．具体的には，クラッシュを挿入してプログラムを異常終了させた後，再度クラッシュを挿入せずに実行する．Crash Consistency が失われている場合には再実行時に何らかのエラーが出ると考えられるため，コード中に Correctness Bug が存在していると判断することができる．以上で示した流れで，一回クラッシュを起こしそれに対応するバグがあれば一つバグを検出，という動作を実現できる．この動作を大量に繰り返すことでクラッシュをコード中の様々な場所で起こし，クラッシュ位置に対応するバグの検出を行う．ただし，Performance Bug に関しては実際に実行した際にエラーという目に見える形で情報を取得することができない．そのため，本論文で提案するクラッシュインジェ

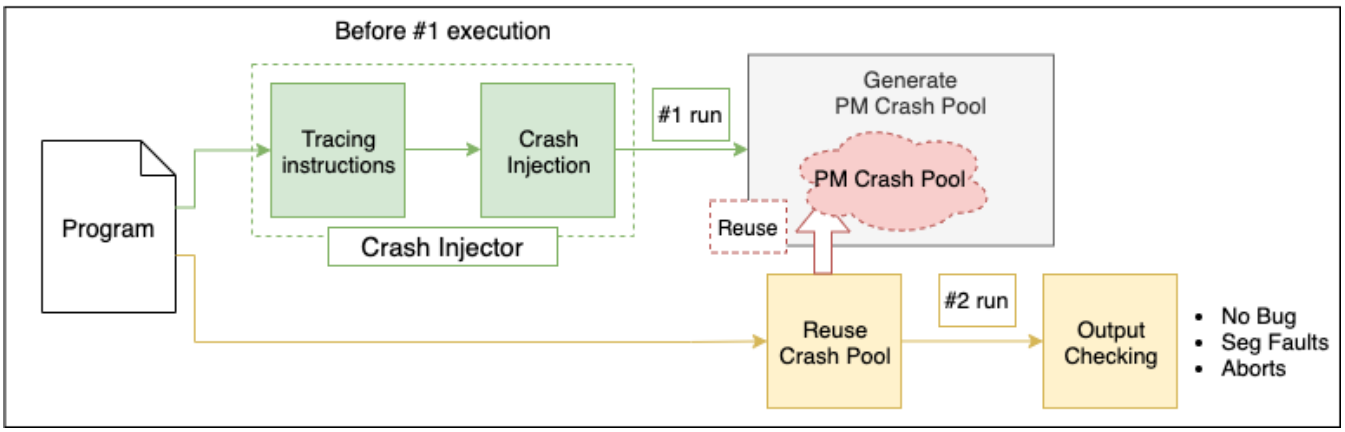

図 4 クラッシュインジェクタとバグ検査の流れの例

クタは，Correctness Bug の検査を行うために用いることを前提としている．

3.2 クラッシュの挿入

図 4 で示すように，クラッシュインジェクタはプログラムの実行前にクラッシュを挿入し，実行時に意図的なクラッシュを起こす．しかし，コード領域全体を対象としてクラッシュを挿入していくことを考える場合，挿入すべき対象が膨大になってしまい不必要な点にまでクラッシュが挿入されてしまう可能性がある．そのため，プログラム中でクラッシュを挿入すべき点を事前に構文解析を用いて選定し，絞り込みを行う必要がある．永続メモリを扱う多くのプログラムは PMDK のトランザクションを用いて実装されているため，本論文ではトランザクションを用いる場合に限定して説明を行う．

Crash Consistency が失われる可能性のある点を考察していく．まず，トランザクション外での操作に関しては永続メモリ上のデータを更新することはない，という約束がある．そのため，トランザクション外領域はクラッシュの挿入対象から除外する．次にトランザクション内での操作に関して．ここでは，ログへの書き込みを行う操作と，永続メモリ上のデータの更新を直接行う操作，そして永続メモリには関わらないデータの更新を行う操作が考えられる．ログへの書き込みを行う操作は，Crash Consistency を保つために行われる．その操作を行う関数が正しくログへの書き込みを行い，またプログラマが正しくその関数を扱う場合には必ず Crash Consistency は保存される．しかし，ログへの書き込みが適切に行われない場合やログに書き込むべき変数を誤った場合には，適切な Crash Consistency が保存されない可能性がある．そのため，ログへの書き込みを行う操作はクラッシュを挿入する対象に含める．次にデータの更新を行う操作に関して．永続メモリ上のデータの更新を直接行う操作に関しては，適切にログへの書き込みが行われない場合 Crash Consistency を損なう可能性があるためクラッシュの挿入対象に含める．それ以外の更新を行う点に関しては，永続メモリ上に保存されるデータ構造を書き換える可能性がないため，本来は操作の対象に含める必要はない．ただ，現状はアドレスを参照して永続メモリを扱う操作かどうかを判別するなどの手法を実装できていないため，本論文の段階ではこの操作もクラッシュの挿入対象に含まれてしまっている．

PMDK のサンプルデータ構造である btree_map.c の一部を**図 3** に簡略化して示す．このうち，6 行目から 12 行目までのトランザクション領域をクラッシュの挿入対象とする．内部で呼ばれる関数内の操作も対象とする．対象となる領域のうち，ログへの書き込みを行う 22 行目の `TX_ADD`, 直接永続メモリ上のデータを書き換える 25, 26 行目の `memmove` がクラッシュを挿入すべき対象として選定される．クラッシュの挿入対象として選定された点それぞれの直後に，クラッシュを起こす関数 (crash_func) の呼び出しを行う．同様にして，コード中の全ての候補点に対して crash_func を呼ぶようにする．関数側でランダムな分岐を起こすための操作を行う．

3.3 バグの検査

永続メモリを扱う際，永続的なメモリリークを発生させないようにメモリプールを用いてメモリを管理している．プールとは，ある一定のサイズのアドレス領域を永続メモリ上に一度に確保してしまって，その上でデータの管理を行うというものである．このプールは特定の名前をつけて管理することができ，新たな名前を指定した場合には新たなプールの作成から操作を開始する．既に存在する名前のプールを用いる場合には，プールが管理するアドレス領域に保存されたデータを用いることが可能となる．永続メモリを扱うプログラムでは，永続化したデータを用いるため基本的にプログラムごとに特定名のプールを作成してデータの管理を行う．そのため，あるタイミングでバグを起こしたデータ構造が永続化されてしまった場合，次回以降の

実行で同じバグが何度も引き起こされることになる．この状態は永続メモリ特有のバグであり，バグに関わるデータを修正する，ロールバックして回復を行う，等の操作が行われない場合，永続的なバグとして残されることになる．

これを踏まえて，実際にバグが存在しているかの検査を行う．具体的な流れは図 4 に示した通りである．まず，クラッシュインジェクタを動かしクラッシュを挿入する．その状態でプログラムを実行し，クラッシュが起こる場合にはプログラムは異常終了し，Crash Consistency Bug を含む可能性のあるメモリプール (PM Crash Pool) が永続メモリ上に残される．次に，クラッシュインジェクタを動かさずに通常通り実行を行う．この際，先ほどと同一名のプール (PM Crash Pool) を用いることで，バグが含まれる可能性のあるデータを参照することができる．実行の結果，何も問題なく動作が完了すればリカバリコードによって正しく Crash Consistency が回復されたと判断することができる．もし再実行の途中で何らかのエラーによって実行が進まなくなった場合，リカバリコードによる回復が正しくできていないと判断できるため，この場合に Crash Consistency Bug があると判断する．以上のようにして，クラッシュインジェクタによるリカバリコードの動作の検証を行うことができる．

3.4 実装

クラッシュインジェクタにおける構文解析・クラッシュの挿入は，LLVM コンパイラパスを用いて実装を行なった．LLVM を用いて解析を行い，選定した挿入点に対応するインストラクションを見つける．各インストラクションの呼び出しの直後に，クラッシュを起こすための関数 (crash_func) を挿入する．関数内では，乱数と関数の呼び出し回数を用いてランダムにクラッシュを起こすかどうかを決定する．これによって，意図的なクラッシュをコード中のランダムな位置で起こす操作を実現する．クラッシュとしては，C 言語の関数である `abort()` を用いた．

4. 評価

この章では図 4 で示した流れを用いて，クラッシュインジェクタによる検査が実際にリカバリコードの検証を容易にできるかどうかの評価を行う．

クラッシュインジェクタの検査対象としては，PMDK が提供するサンプルのデータ構造を用いた．これらのデータ構造は全て PMDK のトランザクションを用いて実装されており，今回はそのプログラムに作為的にバグを入れた場合，クラッシュインジェクタによって検査が可能であるかを検証した．用いるバグは，XFDetector の研究チームが用意したバグパッチを適用して行った．**表 1** に用いたデータ構造およびバグパッチで挿入するバグ，およびバグが検出された割合を示す．用いたバグは全て Correctness Bug を起こす種類のものである．

表 1 データ構造・バグパッチとバグの検出割合
(M: Missing TX_ADD, P: misPlace TX_ADD)

	パッチ名	バグの種類	検出割合 (%)
btree	nobug	—	0
	race1	M	2.27
	race2	P	48.26
	race3	M	0
	race4	M	33.08
ctree	nobug	—	0
	race1	P	0
rbtree	nobug	—	0
	race1	M	37.31
hashmap_tx	nobug	—	0
	race1	M	0.08
	race2	P	0.29

実験環境は次の通りである．

- CPU : Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz
- PM : 512GB Intel OptaneDC DCPMM
- DRAM: 16GB DDR4
- OS : Ubuntu 20.04

4.1 バグの検出結果

各サンプルデータ構造をアイテム数 200 で実行し，クラッシュインジェクタによる検査を 10000 回実行した結果を表 1 に示している．nobug は比較用に実行したバグを入れない状態での検査を行なった結果，race はそれぞれバグを入れた結果を示す．まず，バグパッチを当てずに通常実行を行なった場合，用いたデータ構造では実行結果にエラーを含むものがなかった．次に，バグを入れて実行した場合，2 つのバグを除いて実行時に何かしらのエラーを検出することができた．以上の結果より，本論文のクラッシュインジェクタで，今回用いたデータ構造に含まれる未報告の Crash Consistency Bug を発見することはできなかった．しかし意図的にバグを挿入した結果を踏まえると，種類にもよるが Crash Consistency Bug をクラッシュインジェクタによって正しく発見できるという結果を得られた．

4.2 結果の考察

まず，クラッシュインジェクタの目的を踏まえて，バグの検出割合による違いについて考察していく．データ構造とバグの組み合わせ次第で，0%のものを除くとバグを発見できる割合が 1 %未満から 48 %前後までと大きく異なっていた．クラッシュインジェクタの目的は，クラッシュによってリカバリコードを呼び出すことでその検査を容易に行うことである．今回バグがない場合にはクラッシュが起きても問題は起こらないとわかったため，一回でもバグとなる場合には何かしらのバグがプログラム中に含まれてい

```
1  static void
2  btree_map_insert_empty(TOID(struct btree_map) map,
3    struct tree_map_node_item item)
4  {
5    // Bug: Remove TX_ADD_FIELD
6    // TX_ADD_FIELD(map, root);
7    D_RW(map)->root = TX_ZNEW(struct tree_map_node);
8    ...
9  }
10
11 int
12 btree_map_insert(PMEMobjpool *pop, TOID(struct btree_map) map,
13     struct tree_map_node_item item)
14 {
15   TX_BEGIN(pop){
16     if(BTREE_MAP_EMPTY){
17       btree_map_insert_empty(map, item);
18     }
19     ...
20   }TX_END
21 }
```

図 5 btree_map.c に btree_race3.patch を適用した場合

ると考えることができる．そのため，たとえ一万回中の一回でもバグが起きた時点で何かしらのバグが含まれていると報告することができるため，バグの検出能力と検出割合に関係はないと考えることができる．ただし，バグの検出割合が高いということはそれだけ多くの実行パスがバグによる影響を受けるということなので，その点ではバグ修正の優先度と捉えることもできる．

次にクラッシュインジェクタで検出できなかったバグについて考察する．`btree_race3.patch` を適用したプログラムを**図 5** に示す．このバグは，`btree_map_insert_empty` 内でポインタを更新する際，ログへの書き込みを行う `TX_ADD_FIELD` という関数を取り除くというものである．仕事としては `TX_ADD` と同等である．この場合，バグが挿入される関数が呼び出されるタイミングが重要である．`btree_map_insert` で関数呼び出しを行う際，呼び出される条件は `BTREE_MAP_EMPTY` が成立している場合であり，これを言い換えると `btree_map` というデータ構造がまだ空の状態でのみ実行されるパスである．つまり，この条件分岐は必ずデータ構造を作成する初回のみで成立するため，他のノード等とのデータの依存関係が存在していない．例えば 7 行目でクラッシュが起きる場合，割り当てたデータは全て失われる．しかし依存関係があるデータがそもそも存在していないので，永続メモリ上に Crash Consistency Bug として保存されることはない．他の位置でクラッシュが起きる場合，考慮しているデータ領域が既にログ・永続メモリに書き込まれていれば Crash Consistency Bug が生じることはない．ログに書き込まれていない状態で他のデータと依存関係が生じていればバグとなるが，そのような状況は発生し得ないと考えられる．理由は，例えば木構造に新たに葉を追加する際，根の更新も行う必要がありその時点でログへの書き込みが行われるためである．ログに書き込まれない場合はその点がバグであると言えるため，今回考慮すべき内容と別のバグが生じていることになる．以上のことより，他のデータとの依存関係が確立する前に起こるバグは，クラッシュインジェクタによる検査で見つけることは困難であると考えられる．`ctree_race1` も同様のバグを挿入するパッチであり，同じ理由で今回バグの検出ができなかったと考えられる．

4.3 課題点

本論文の課題として，まず実験に用いた対象プログラムが少ないことが挙げられる．今回は PMDK のサンプルデータ構造を用いた実験データのみを示した．PMDK のトランザクションを用いたアプリケーションとして，代表例として他に Redis-pmem が挙げられる．そのため，まずは Redis-pmem に対してもクラッシュインジェクタを用いてバグ解析を行うことを考えており，より実用的な範囲でのクラッシュインジェクタの有用性を示すことができると考えている．

次に，バグの挿入点の選定をより細かく行うことができるという点である．3 章でも示した通り，バグを挿入する対象を選定することでより効率的な検証を行うことができる．しかし，アドレスの情報を用いるというような厳密な絞り込みを実装できていないため，永続化の必要がない操作もクラッシュの挿入候補として選ばれてしまう可能性がある．そのため，不必要な操作を取り除くことで，同じ回数でもより効率よくバグの検出を行うことができるようになると考えられる．

最後に，本論文で提案するクラッシュインジェクタは，実行の対象を PMDK のトランザクションを用いるプログラムに限定しているという点である．これは 2 章にも示しているが，PMDK を用いて実装を行う場合にはサンプルデータ構造を含めトランザクションを用いる場合が多いためである．しかし，トランザクションを提供する `libpmemobj` ではないライブラリを用いて実装されたアプリケーションとして memchached-pmem があり，必ずしも全てのプログラムがトランザクションを用いているわけではない．PMDK の特定のライブラリを用いたプログラムに限定した実装では，クラッシュインジェクタが適用可能なプログラムも大きく制限を受けると考えられる．そのため，より広範囲のプログラムを検査するため，まずはトランザクションに限らない実装手法を考える必要があると考えられる．最終的には，PMDK に限らず永続メモリを扱うプログラム全てに対する検査を行うことができるよう拡張を行いたいと考えている．

5. 関連研究

5.1 バグ検出

PMTest[7] は，永続メモリを扱うプログラムをトレースし，得られた永続メモリの更新操作とプログラマが定義したメモリ更新のルールと照合を行うことでバグの解析を行うツールである．柔軟で高速な解析ができる反面，プログラマ側でルールを設定する必要があるため，プログラマ

側の習熟度も解析結果に影響しかねないという問題がある．XFDetector[6] は，フォールトインジェクションの考えを用いて永続メモリの解析を行う．フォールトインジェクションは，クラッシュインジェクションと同様の考えで，動的にプログラム中にバグを挿入することでその後の動作を検査し，バグの解析を行うというものである．ただ XFDetector では実際にプログラム中にフォールトを大量に入れるのではなく，ShadowPM という考えを用いて擬似的な動的解析を行う．2 つのプロセスを管理し，フォールトを挿入するタイミングで同じ操作をトレースしている ShadowPM のプロセスに遷移し，フォールト後にどのような動きとなるかを検査する．こちらもプログラム中で解析する領域を限定するためにプログラマ側でアノテーションを挿入する必要がある．そのため，こちらもある程度プログラマ側の習熟度が要求される．ユーザ側のアノテーションを必要としないツールの例として，PMDebugger[2] が挙げられる．永続メモリにおける `flush` の操作の間を 1 区間として，配列と AVL 木を用いてより高速で広範なバグ検出を行うことができる．XFDetector は動的解析を行うツールであるが，以上であげた以外の研究 [1][3][5][9] でも基本の考えとして静的解析を用いており，いかに全ての実行パスを検査せず効果的にバグ検出を行うことができるかを模索している．本論文では，Crash Consistency Bug を動的に解析する手法を提案し，静的解析では探索できないバグの検出やプログラムへの適用が可能である．

5.2 バグ修正

Hippocrates[8] は，永続メモリにおけるバグの修正を行うツールである．対象とするバグは本論文でいう Correctness Bug に限定しており，それらのバグを自動的に修正する．考えとしては簡単で，永続化を行う操作がない部分を検出し，その部分に永続化を行う記述を追加していくということである．また，ヒューリスティックを用いて最適化を行うことで，修正時に追加したコードによるパフォーマンスの低減を防止する．

6. まとめ

永続メモリを扱うプログラムにおいて，Crash Consistency Bug という問題がある．突然のクラッシュや電源の遮断によって操作が強制的に終了され，操作が正しく実行されない場合，扱っていたデータの一貫性が失われるというバグである．既存のツールは主に静的解析を用いてバグの検出を行うが，動的に変化する実行パスへの対応やシングルスレッド以外のプログラムに対しては適用が難しいものである．本論文では，永続メモリにおいて Crash Consistency Bug の検証を動的に行うためのクラッシュインジェクタを提案した．具体的には，意図的なクラッシュを起こすことでリカバリコードを呼び出す．その後のデータの一貫性を確認することで，リカバリコードに関わるバグの検出を容易に行うことを可能にした．PMDK のトランザクションを用いるサンプルデータ構造に対して実行を行い，クラッシュインジェクタを用いてバグが検出可能であることを確認した．

謝辞 本研究は，JST，CREST，JPMJCR19F3 の支援を受けたものである．

参考文献

[1] Choi, B., Burns, R. and Huang, P.: Understanding and Dealing with Hard Faults in Persistent Memory Systems, *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, New York, NY, USA, Association for Computing Machinery, p. 441–457 (online), DOI: 10.1145/3447786.3456252 (2021).

[2] Di, B., Liu, J., Chen, H. and Li, D.: Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs, *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, New York, NY, USA, Association for Computing Machinery, p. 503–516 (online), DOI: 10.1145/3445814.3446744 (2021).

[3] Fu, X., Kim, W.-H., Shreepathi, A. P., Ismail, M., Wadkar, S., Lee, D. and Min, C.: Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores, *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, New York, NY, USA, Association for Computing Machinery, p. 100–115 (online), DOI: 10.1145/3477132.3483556 (2021).

[4] Intel: pmem.io.

[5] Liu, S., Mahar, S., Ray, B. and Khan, S.: PMFuzz: Test Case Generation for Persistent Memory Programs, *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, New York, NY, USA, Association for Computing Machinery, p. 487–502 (online), DOI: 10.1145/3445814.3446691 (2021).

[6] Liu, S., Seemakhupt, K., Wei, Y., Wenisch, T., Kolli, A. and Khan, S.: Cross-Failure Bug Detection in Persistent Memory Programs, *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, New York, NY, USA, Association for Computing Machinery, p. 1187–1202 (online), DOI: 10.1145/3373376.3378452 (2020).

[7] Liu, S., Wei, Y., Zhao, J., Kolli, A. and Khan, S.: PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, New York, NY, USA, Association for Computing Machinery, p. 411–425 (online), DOI: 10.1145/3297858.3304015 (2019).

[8] Neal, I., Quinn, A. and Kasikci, B.: Hippocrates: Healing Persistent Memory Bugs without Doing Any Harm, *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, New York, NY, USA, Association for Computing Machinery, p. 401–414 (online), DOI: 10.1145/3445814.3446694 (2021).

[9] Neal, I., Reeves, B., Stoler, B., Quinn, A., Kwon, Y., Peter, S. and Kasikci, B.: AGAMOTTO: How

Persistent is your Persistent Memory Application?, *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, pp. 1047–1064 (online), available from ⟨https://www.usenix.org/conference/osdi20/presentation/neal⟩ (2020).