AGAMOTTO

missing flush/fence pattern

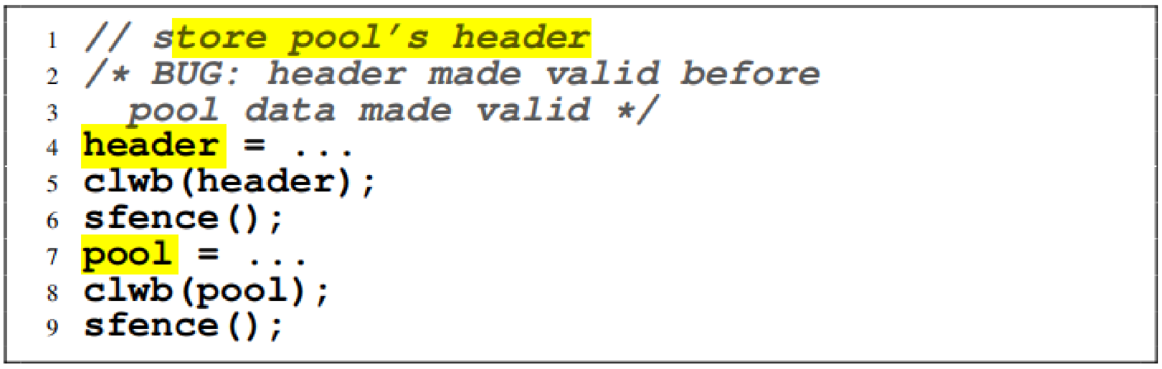
```
x:=1;
```

extra flush/fence pattern

```
int x=1;
flush x;
flush x;
//sfence();
```

Also includes flush, clwb operations in loops;

other(don't know how to simplify): in these cases, data is correctly flushed to PM and there are no redundant flush operations, but the application misuses PM, leading to performance or correctness issues.



```
// store pool's header
/* BUG: header made valid before
   pool data made valid */
header = ...
clwb(header);
sfence();
pool = ...
clwb(pool);
sfence();
```

Listing 4: An example correctness bug adapted from PMDK Issue #14.

PMRace

PM Inter-thread Inconsistency

```
Thread 1  | Thread 2
x := A;   |
          | y := x;
          | clwb(&y);
          | sfence();
#If crash happens here, it is possible that y != x in PM;
clwb(&x); |
sfence(); |
```

PM Synchronization Inconsistency

```
Thread 1  | Thread 2
lock(&g)  |
x := A    |
unlock(&g)|
          | lock(&g)
#If crash happens here, everything stucks because of lock mechanism;
          | y := x // x+=1 ??
          | clwb(&y);
          | sfence();
clwb(&x); |
sfence(); |
```

Pmemcheck

Memory overwrites: check in here;

This refers to the case where multiple modifications to the same persistent memory location occur before the location is made persistent, **especially when the later modification depends on the previous value**.

```
x := 1;
x++;
flush(&x);
```

Witcher

persistence ordering violation:

not flush in time

```
//for example, x == 0, cnt == 0 initially, cnt is the number of times x has been
modified
int x = 0, cnt = 0;
x +=3;
cnt++;
//if crash happens here, it might happen that x is persisted while cnt is not.
flush(&x);
flush(&cnt);
```

persistence atomicity violation:

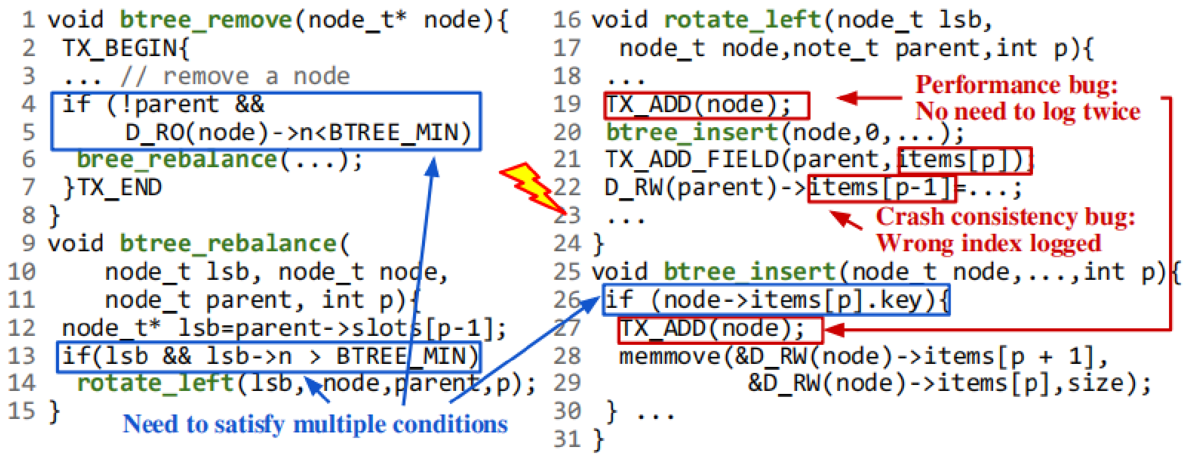
Sometimes due to the logic of the program, two or more operations are considered to be atomic. So it is possible that the crash consistency problems happen after recovering from crash.

```
//for example, if we have to make sure x+y = 10
x := A;
y := 10 - x;
//line 2&3 have to be atomic. if crash happens after x := A but before y := 10 -
A, there might be persistence atomicity violation.
flush(&x);
flush(&y);
```

PMFuzz (don't know how to simplify)

crash consistency bug: the program updates the (p-1)-th item (line 22) but logs the p-th item by mistake (line 21). In case of a failure at line 22, the item being modified can be lost as it has not been backed up by the log.

performance bug: log the same node twice (line 19 and 27);



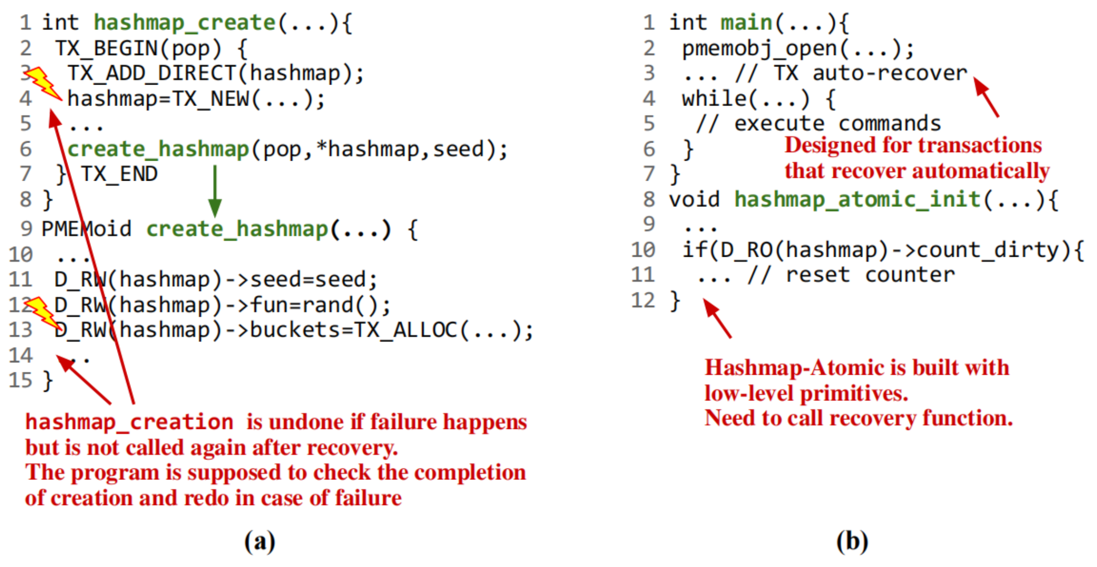
```
 1 void btree_remove(node_t* node){
 2  TX_BEGIN{
 3  ... // remove a node
 4  if (!parent &&
 5       D_RO(node)->n<BTREE_MIN)
 6   bree_rebalance(...);
 7  }TX_END
 8 }
 9 void btree_rebalance(
10      node_t lsb, node_t node,
11      node_t parent, int p){
12  node_t* lsb=parent->slots[p-1];
13  if(lsb && lsb->n > BTREE_MIN)
14   rotate_left(lsb,node,parent,p);
15 }
```

```
16 void rotate_left(node_t lsb,
17   node_t node,note_t parent,int p){
18  ...
19  TX_ADD(node);
20  btree_insert(node,0,...);
21  TX_ADD_FIELD(parent,items[p]);
22  D_RW(parent)->items[p-1]=...;
23  ...
24 }
25 void btree_insert(node_t node,...,int p){
26  if (node->items[p].key){
27   TX_ADD(node);
28   memmove(&D_RW(node)->items[p + 1],
29           &D_RW(node)->items[p],size);
30  } ...
31 }
```

Figure 1: A buggy PM-based B-Tree (Example 1).

a: shown clearly in Figure 14;

b: main() is a driver program for PMDK's key-value store; hashmap_atomic_init(), is a procedure in Hashmap-Atomic. It has crash consistency bug, since main() assumes all key-value store structures can automatically recover using transactions, but overlooks the low-level-primitive-based Hashmap-Atomic.(They can't automatically recover using transactions)



```
int hashmap_create(...){
 TX_BEGIN(pop) {
  TX_ADD_DIRECT(hashmap);
  hashmap=TX_NEW(...);
  ...
  create_hashmap(pop,*hashmap,seed);
 } TX_END
}
PMEMoid create_hashmap(...) {
 ...
 D_RW(hashmap)->seed=seed;
 D_RW(hashmap)->fun=rand();
 D_RW(hashmap)->buckets=TX_ALLOC(...);
 ...
}
```

hashmap_creation is undone if failure happens but is not called again after recovery. The program is supposed to check the completion of creation and redo in case of failure

(a)

```
int main(...){
 pmemobj_open(...);
 ... // TX auto-recover
 while(...) {
  // execute commands
 }
}
void hashmap_atomic_init(...){
 ...
 if(D_RO(hashmap)->count_dirty){
  ... // reset counter
}
```

Designed for transactions that recover automatically

Hashmap-Atomic is built with low-level primitives. Need to call recovery function.

(b)

Figure 14: New crash consistency bugs found by PMFuzz: (a) Bug 1 and (b) Bug 6.

DURINN

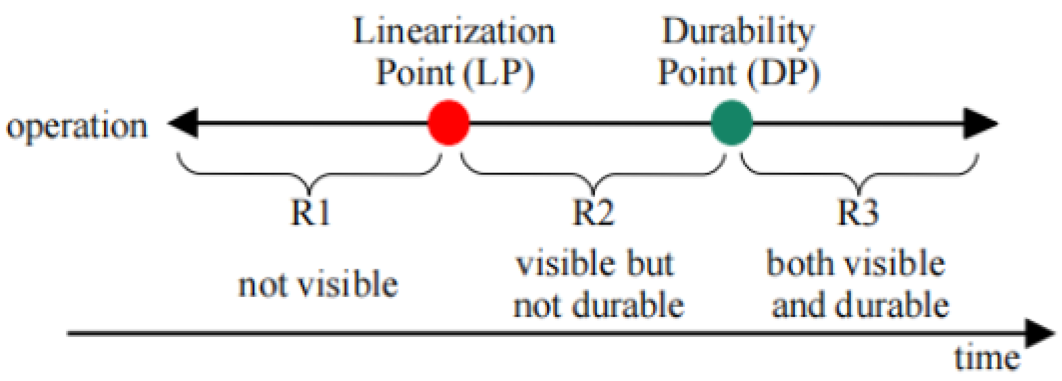


Figure 2: Linearization point and durability point split an operation into three-time intervals as per its visibility and durability guarantees.

misusing of operations & happens in R3:

```
new_hush = hashtable_create();
clflush_next_check(new_hush);
fence();
add_root(h, new_hush);//set root pointer h to a new hash table, making its
effect visible
clflush(h);
//if crash happens here, the inserted key-value data to new_hush might be lost
since clflush_next_check at line 8 does not flush all the updated NVM data in the
new hash table and leaves a part unpersisted.
```

An Unrecovered-Durable Bug & happen in R1 and R2 :(not sure this is the right way to simpilify)

```
insert(K, V);
CAS();//synchronization operation in lock-free program, making the effects of
insert visible;
//crash happens
get(K);//after recovery, if it returns V rather than NULL, then here is a bug;
```

happens in R2: maybe is similar to the one in PMRace

```
Thread 1  | Thread 2
x := 1;   |
          | if(x == 1) x++;
//If crash happens here, y != x in PM;
clwb(&x); |
sfence(); |
```

Px86

1.

```
x1=1;
flush_opt(x2);//x1 and x2 are in the same cache line;
y=1;//y is not in the same cacheline with x1 and x2;
//if crash happens here, it is possible that y:= 1 has executed and persisted but
flush_opt hasn't. So after recovery, x1,y are likely not be equal to 1;
```

2.

```
Thread 1  | Thread 2
y := 1;   | if(y == 1) z := 1;
//If a crash occurs after z:= 1 has executed and persisted but before y:= 1 has
persisted, it is possible to observe y=0, z=1 after recovery.
```

raw example:

$$\begin{array}{l||l} x := 1; & \\ \mathbf{flush}_{\mathrm{opt}}\ x'; \dagger & \mathsf{a} := y; \\ \mathbf{sfence}; & \mathbf{if}\ \mathsf{a}\ \mathbf{then} \\ y := 1; & \quad z := 1; \end{array}$$

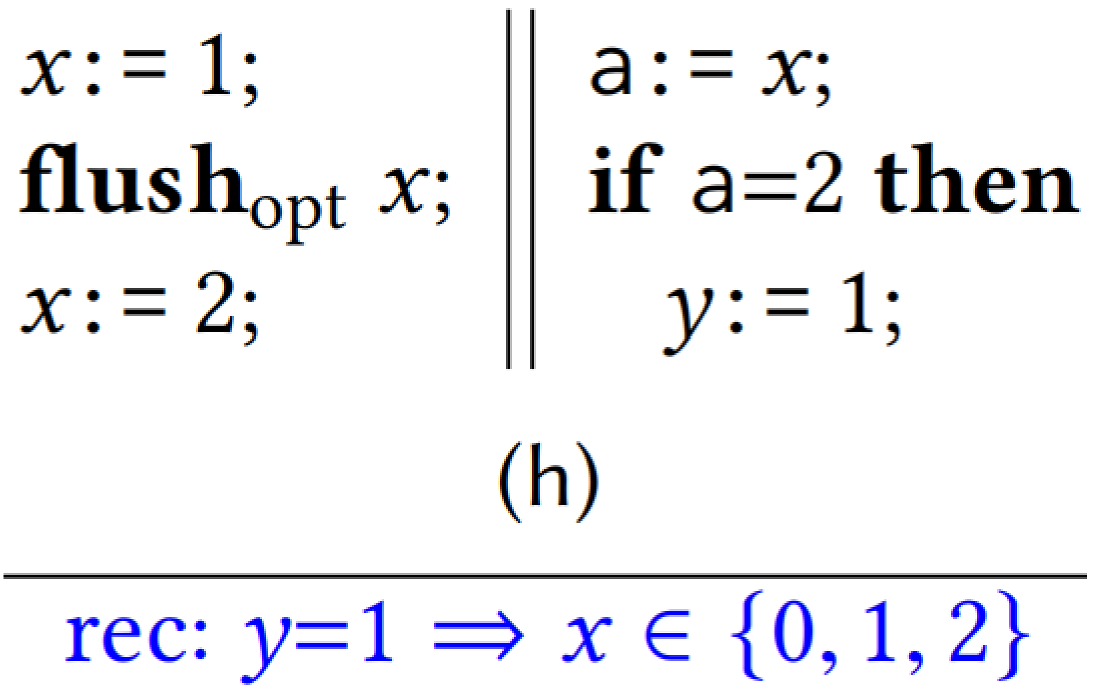
(g)

rec: $z=1 \Rightarrow (x=1 \wedge y \in \{0,1\})$

3.

```
x := 1;
flush_opt(&x);
x := 2;
//notice that x=2 and flush_opt x can be reordered. So if crash happens here, the value of x is uncertain.
```

raw example:



```
x := 1;          ||  a := x;
flush_opt x;     ||  if a=2 then
x := 2;          ||    y := 1;
```

(h)

$$\text{rec: } y=1 \Rightarrow x \in \{0, 1, 2\}$$

PMDebugger

no durability guarantee: A persistent memory location is not persisted after the last write to it. similar to missing flush/fence pattern in paper **AGAMOTTO**.

Multiple overwrites: similar to the one in **Pmemcheck**.

No order guarantee: some programs have to make sure some operations are in a fixed order, while they can not guarantee that. Similar to persistence atomicity violation in paper **Witcher**.

Redundant flushes: similar to extra flush/fence pattern in paper **AGAMOTTO**.

Flush nothing: A CLF instruction does not persist any prior store. For example, the memory location persisted by the instruction does not exist in the bookkeep space.

```
x := 1;
flush(&y);
```