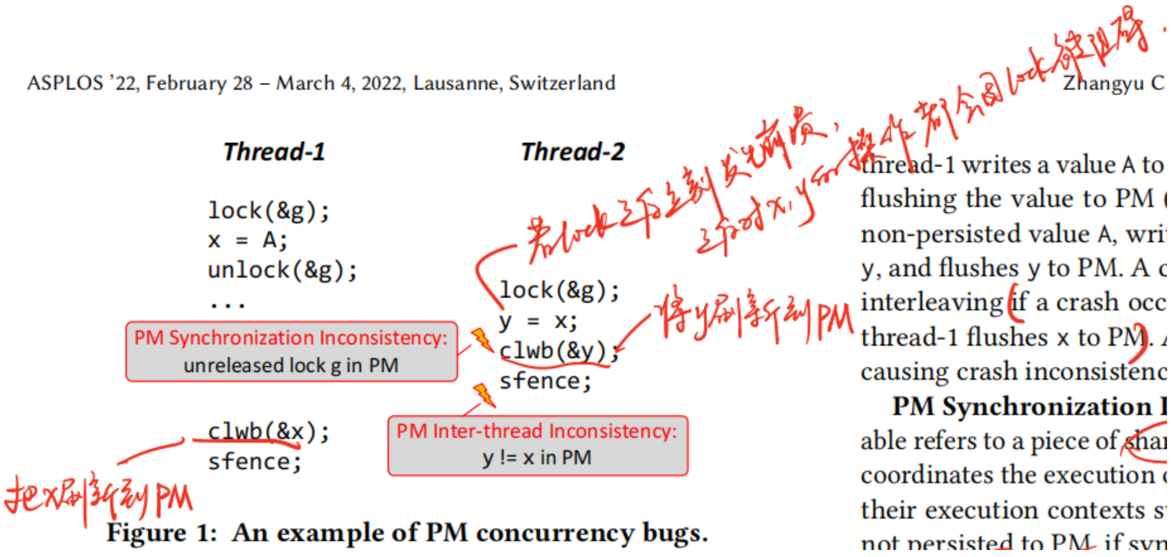
PMRace

1. PM Inter-thread Inconsistency

One thread makes durable side effects based on non-persisted data written by another thread.

2. PM Synchronization Inconsistency

If synchronization variables are persisted to PM and recovered after crashes, the inconsistency between synchronization variables and the new threads in the recovered PM application induces an inconsistency.



ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland — Zhangyu C

Thread-1

```
lock(&g);
x = A;
unlock(&g);
...

clwb(&x);
sfence;
```

Thread-2

```
lock(&g);
y = x;
clwb(&y);
sfence;
```

PM Synchronization Inconsistency: unreleased lock g in PM
PM Inter-thread Inconsistency: y != x in PM
若lock之后立刻发生崩溃，之后对x,y的操作都会因lock被阻塞。
将y刷新到PM
把x刷新到PM

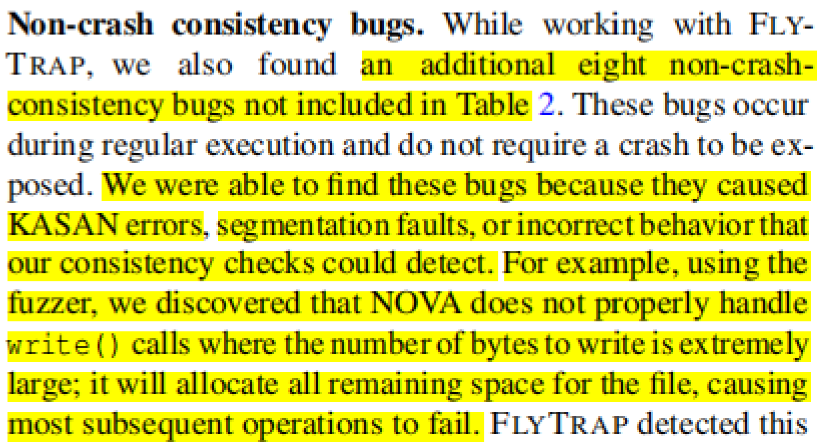
Figure 1: An example of PM concurrency bugs.

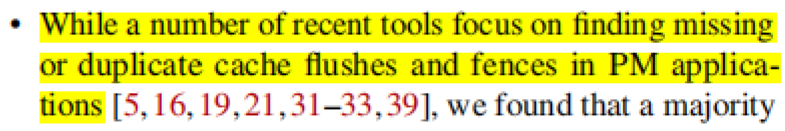
thread-1 writes a value A to
flushing the value to PM
non-persisted value A, wri
y, and flushes y to PM. A
interleaving if a crash occ
thread-1 flushes x to PM.
causing crash inconsistenc
PM Synchronization I
able refers to a piece of shar
coordinates the execution
their execution contexts s
not persisted to PM if syn

FLYTRAP(not so important)

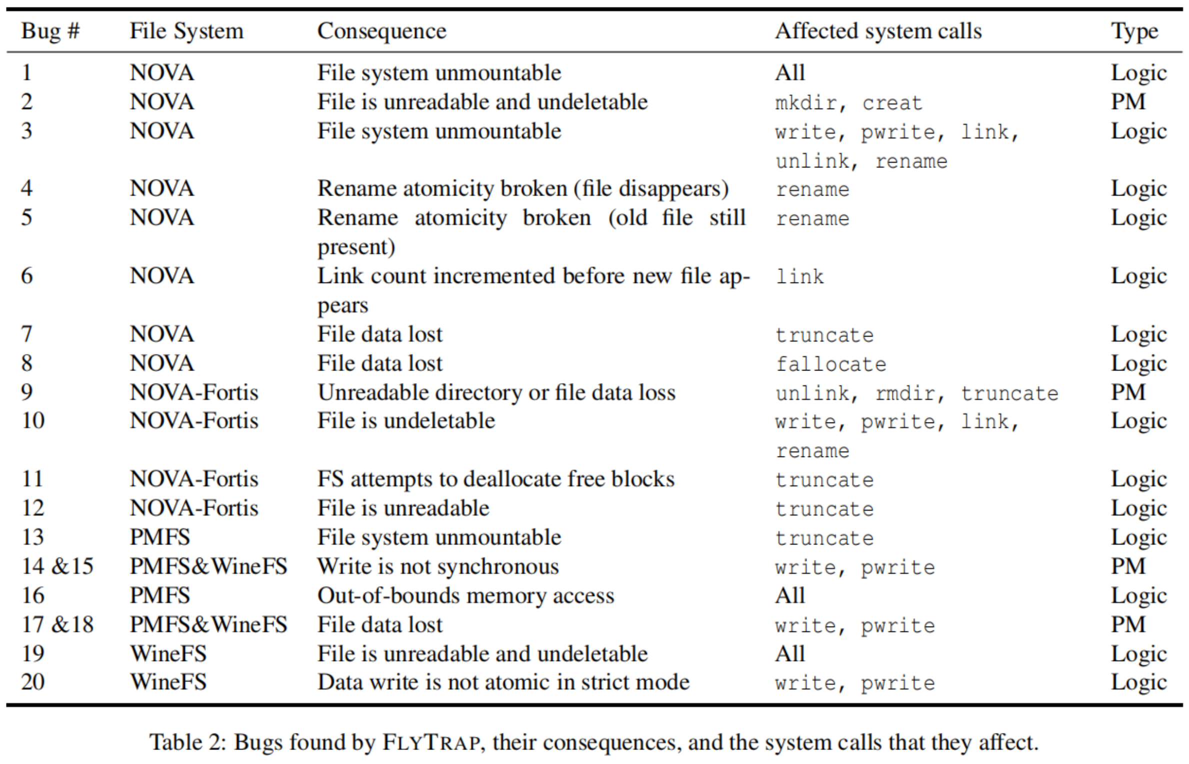
在论文**Finding and Analyzing Crash-Consistency Bugs in Persistent-Memory File Systems**中：

Observation	Associated bugs
Many bugs occur due to logic or design issues, not PM programming errors.	1, 3–8, 10–13, 16, 19, 20
The complexity of performing in-place updates leads to bugs.	4–7, 14, 15
Recovery code that deals with rebuilding in-DRAM state is a significant source of bugs.	1, 3, 7, 11, 13, 16, 19
Complex new features for increasing resilience can introduce new crash consistency bugs.	2, 9–12
A majority of observed bugs can only be exposed by simulating crashes during system calls.	3–6, 9–13, 19, 20
Short workloads were sufficient to expose many crash consistency bugs.	1–6, 9–20
Many bugs were exposed by replaying a few small writes onto previously persistent state.	3–6, 9–13, 19, 20


Non-crash consistency bugs. While working with FLYTRAP, we also found an additional eight non-crash-consistency bugs not included in Table 2. These bugs occur during regular execution and do not require a crash to be exposed. We were able to find these bugs because they caused KASAN errors, segmentation faults, or incorrect behavior that our consistency checks could detect. For example, using the fuzzer, we discovered that NOVA does not properly handle `write()` calls where the number of bytes to write is extremely large; it will allocate all remaining space for the file, causing most subsequent operations to fail. FLYTRAP detected this


- While a number of recent tools focus on finding missing or duplicate cache flushes and fences in PM applications [5, 16, 19, 21, 31–33, 39], we found that a majority

crush-consistency bugs： a majority of the observed crash-consistency bugs are logic issues rather than PM programming errors. Logic bugs are issues that cannot be fixed by adding cache ine flushes or store fences.



Bug #	File System	Consequence	Affected system calls	Type
1	NOVA	File system unmountable	All	Logic
2	NOVA	File is unreadable and undeletable	`mkdir, creat`	PM
3	NOVA	File system unmountable	`write, pwrite, link, unlink, rename`	Logic
4	NOVA	Rename atomicity broken (file disappears)	`rename`	Logic
5	NOVA	Rename atomicity broken (old file still present)	`rename`	Logic
6	NOVA	Link count incremented before new file appears	`link`	Logic
7	NOVA	File data lost	`truncate`	Logic
8	NOVA	File data lost	`fallocate`	Logic
9	NOVA-Fortis	Unreadable directory or file data loss	`unlink, rmdir, truncate`	PM
10	NOVA-Fortis	File is undeletable	`write, pwrite, link, rename`	Logic
11	NOVA-Fortis	FS attempts to deallocate free blocks	`truncate`	Logic
12	NOVA-Fortis	File is unreadable	`truncate`	Logic
13	PMFS	File system unmountable	`truncate`	Logic
14 &15	PMFS&WineFS	Write is not synchronous	`write, pwrite`	PM
16	PMFS	Out-of-bounds memory access	All	Logic
17 &18	PMFS&WineFS	File data lost	`write, pwrite`	PM
19	WineFS	File is unreadable and undeletable	All	Logic
20	WineFS	Data write is not atomic in strict mode	`write, pwrite`	Logic

Table 2: Bugs found by FLYTRAP, their consequences, and the system calls that they affect.

Most assumptions about file-system crash consistency do not carry across to persistent memory, where the consistency mechanisms and guarantees are significantly different.

Pmemcheck

Discover Persistent Memory Programming Errors with Pmemcheck;

- Non-persistent stores: This refers to **data written to persistent memory but not flushed explicitly. This is clearly a problem, since data not flushed may still sit on the CPU caches and could be lost** if the process were to crash unexpectedly.

example:

```c
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <valgrind/pmemcheck.h>//之后省略头文件
int main (int argc, char *argv[]) {
        int fd, *data;
    //open the file
        fd = open ("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
    //posix_fallocate() is used to make sure there is enough space to allocate an
integer
        posix_fallocate (fd, 0, sizeof (int));
    //the file is memory-mapped
        data = (int *)mmap (NULL, sizeof (int), PROT_READ|PROT_WRITE,
                            MAP_SHARED, fd, 0);
    //the pointer is registered to pmemcheck with the macro
        VALGRIND_PMC_REGISTER_PMEM_MAPPING (data, sizeof (int));
    // data is written to persistent memory and the file unmapped
        *data = 1234;
        ////bug occurs because the data is not being flushed after the write
        munmap (data, sizeof (int));
    //unmapping the file
        VALGRIND_PMC_REMOVE_PMEM_MAPPING (data, sizeof (int));
        return 0;
}
```

- Stores not added into a transaction: When working within a transaction block, it is **assumed that all the modified persistent memory locations have been added to it** at the beginning (which also implies that their previous values are copied to an undo log). This allows the transaction to flush at the end of the block or roll back to the old values in the event of an unexpected failure. **A modification within a transaction to a location that is not added to the transaction is most surely a bug** that pmemcheck will warn about.

example:

```c
int main (int argc, char *argv[]) {
        int fd, *data;
        fd = open ("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
        posix_fallocate (fd, 0, sizeof (int));
        data = (int *)mmap (NULL, sizeof (int), PROT_READ|PROT_WRITE,
                            MAP_SHARED, fd, 0);
        VALGRIND_PMC_REGISTER_PMEM_MAPPING (data, sizeof (int));
        VALGRIND_PMC_START_TX;//“start” a transaction
        *data = 1234;
        flush ((void *)data, sizeof (int));//解决Non-persistent stores的bug
        VALGRIND_PMC_END_TX;//“end” a transaction
    //这里只是标出了start and end transaction的位置，但是并没有真正的add something to
the transaction.
        munmap (data, sizeof(int));
        VALGRIND_PMC_REMOVE_PMEM_MAPPING (data, sizeof (int));
        return 0;
}
```

- **Memory added to two different transactions**: In the case where one program can work with multiple transactions simultaneously, adding the same memory object to multiple transactions has the **potential to corrupt data**. This is the case in PMDK, for example, where the library maintains a different transaction per thread. **If two threads write to the same object within transactions, there are some scenarios where one thread crashing can override other non-crashing threads' modifications**.

example：出现两个线程给一个PM位置写入的操作；应该当使用lock mechanism;

```
int main (int argc, char *argv[]) {
        int fd, *data;
        pthread_t thread;
        fd = open ("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
        posix_fallocate (fd, 0, sizeof (int));
        data = (int *)mmap (NULL, sizeof (int), PROT_READ|PROT_WRITE,
                            MAP_SHARED, fd, 0);
        VALGRIND_PMC_REGISTER_PMEM_MAPPING (data, sizeof (int));
        //create transaction number n
        VALGRIND_PMC_START_TX_N (0);
        VALGRIND_PMC_ADD_TO_TX_N (0, data, sizeof (int));//this line of code is used to sovle the problem of not adding stores into a transaction
        //create a thread that will run the function func()
        pthread_create (&thread, NULL, func, (void *) data);
        //func的代码省略, it creates a new transaction with id 1, 和transaction 0 做一样的事情: it adds data to the transaction, writes to it, and then ends the transaction
        *data = 1234;
        flush ((void *)data, sizeof (int));////解决Non-persistent stores的bug
        pthread_join (thread, NULL);
        VALGRIND_PMC_END_TX_N (0);
        munmap (data, sizeof(int));
        VALGRIND_PMC_REMOVE_PMEM_MAPPING (data, sizeof (int));
        return 0;
}
```

- Memory overwrites: This refers to the case where **multiple modifications to the same persistent memory location occur before the location is made persistent**. In general, it is always better to **use volatile memory for short-lived data**.

example: fix this by either inserting a flushing instruction between the writes or eliminating one of the writes.

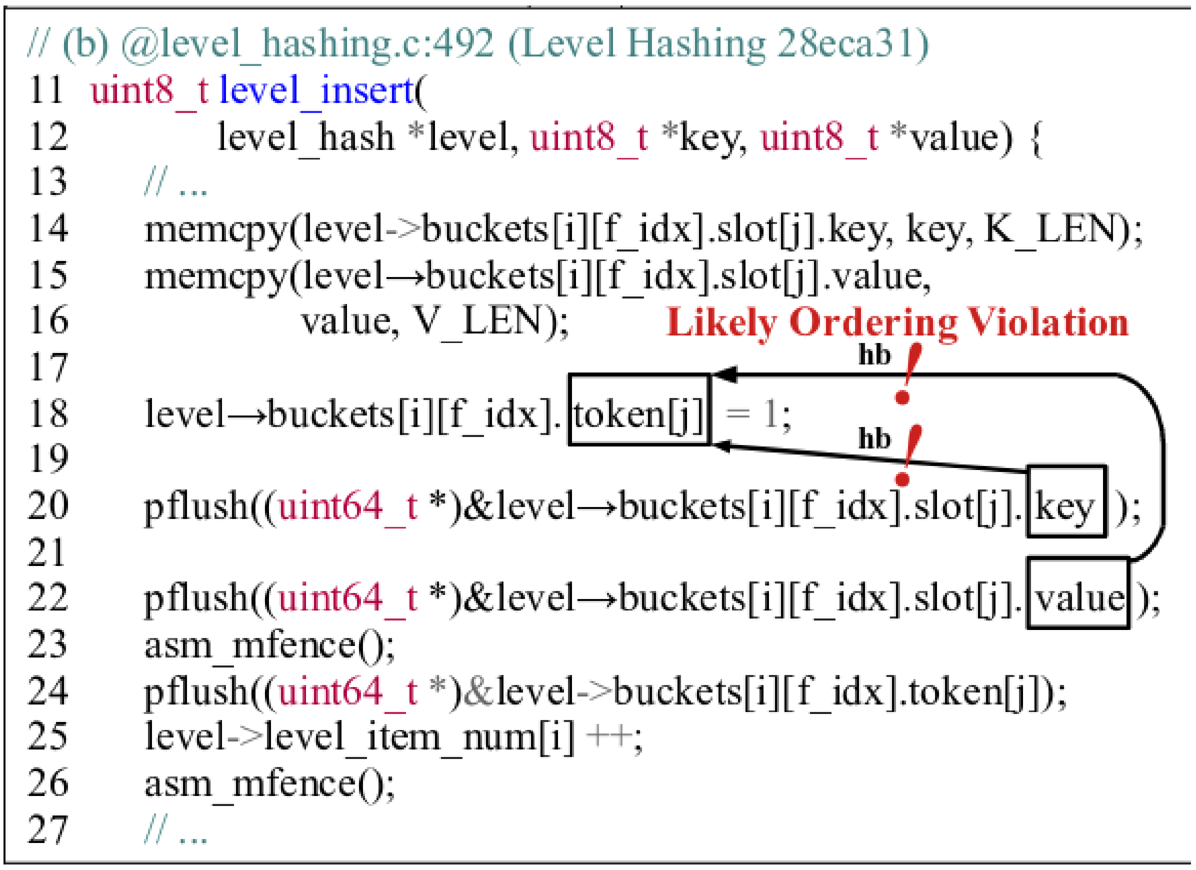
```
int main (int argc, char *argv[]) {
        int fd, *data;
        fd = open ("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
        posix_fallocate (fd, 0, sizeof (int));
        data = (int *)mmap (NULL, sizeof (int), PROT_READ|PROT_WRITE,
                            MAP_SHARED, fd, 0);
        VALGRIND_PMC_REGISTER_PMEM_MAPPING (data, sizeof (int));
        *data = 1234;
        *data = 4321;
        //easy to see that the program writes to data twice before flushing.
        flush ((void *)data, sizeof (int));
        munmap (data, sizeof(int));
        VALGRIND_PMC_REMOVE_PMEM_MAPPING (data, sizeof (int));
        return 0;
}
```

- Unnecessary flushes: Flushing should be done carefully. Detecting unnecessary flushes (such as redundant ones) can help in improving code performance.

```
int main (int argc, char *argv[]) {
        int fd, *data;
        fd = open ("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
        posix_fallocate (fd, 0, sizeof (int));
        data = (int *)mmap (NULL, sizeof (int), PROT_READ|PROT_WRITE,
                            MAP_SHARED, fd, 0);
        VALGRIND_PMC_REGISTER_PMEM_MAPPING (data, sizeof (int));
        *data = 1234;
        flush ((void *)data, sizeof (int));
        flush ((void *)data, sizeof (int));
        //the program flushes data twice.
        munmap (data, sizeof(int));
        VALGRIND_PMC_REMOVE_PMEM_MAPPING (data, sizeof (int));
        return 0;
}
```

Witcher

pater website

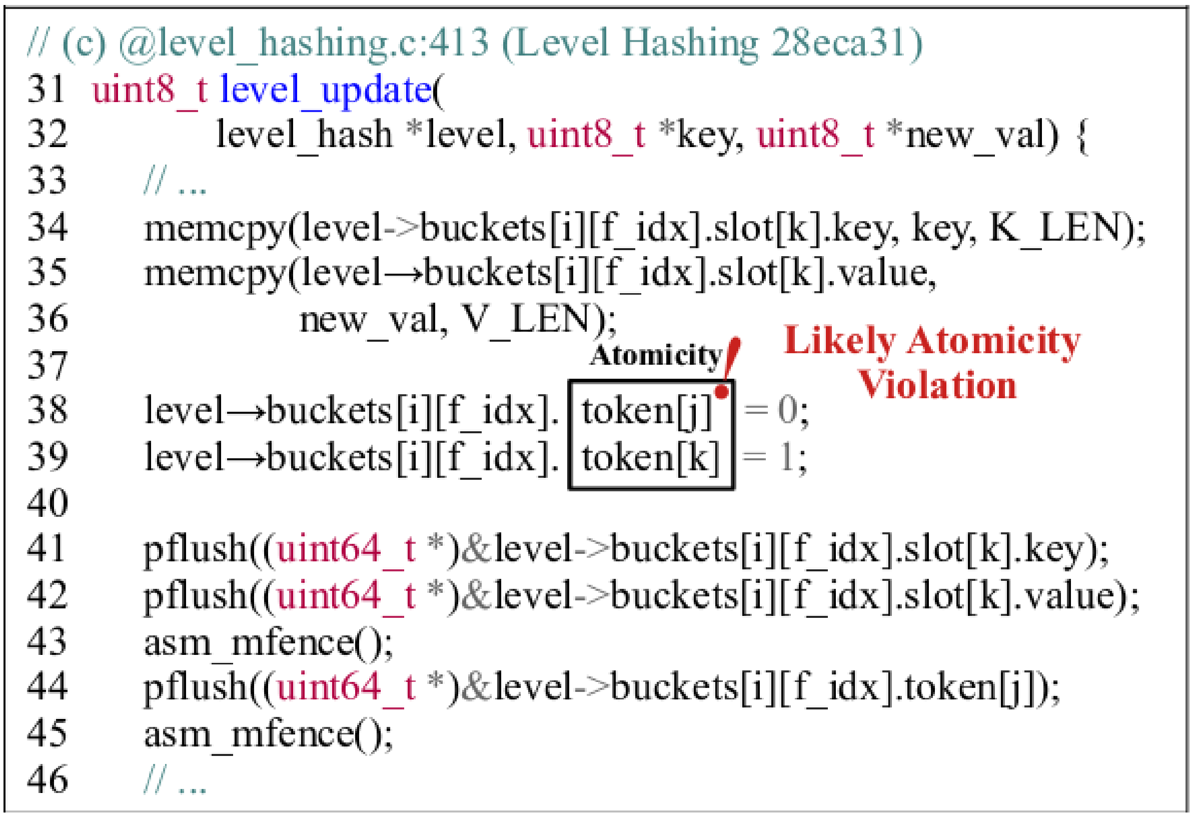


```
// (b) @level_hashing.c:492 (Level Hashing 28eca31)
uint8_t level_insert(
        level_hash *level, uint8_t *key, uint8_t *value) {
    // ...
    memcpy(level->buckets[i][f_idx].slot[j].key, key, K_LEN);
    memcpy(level→buckets[i][f_idx].slot[j].value,
            value, V_LEN);

    level→buckets[i][f_idx].token[j] = 1;

    pflush((uint64_t *)&level→buckets[i][f_idx].slot[j].key);

    pflush((uint64_t *)&level→buckets[i][f_idx].slot[j].value);
    asm_mfence();
    pflush((uint64_t *)&level->buckets[i][f_idx].token[j]);
    level->level_item_num[i] ++;
    asm_mfence();
    // ...
```



```
// (c) @level_hashing.c:413 (Level Hashing 28eca31)
uint8_t level_update(
        level_hash *level, uint8_t *key, uint8_t *new_val) {
    // ...
    memcpy(level->buckets[i][f_idx].slot[k].key, key, K_LEN);
    memcpy(level→buckets[i][f_idx].slot[k].value,
            new_val, V_LEN);

    level→buckets[i][f_idx].token[j] = 0;
    level→buckets[i][f_idx].token[k] = 1;

    pflush((uint64_t *)&level->buckets[i][f_idx].slot[k].key);
    pflush((uint64_t *)&level->buckets[i][f_idx].slot[k].value);
    asm_mfence();
    pflush((uint64_t *)&level->buckets[i][f_idx].token[j]);
    asm_mfence();
    // ...
```

Jaaru

Pmfuzz

Cross-failure bug detection in persistent memory programs

Cross-failure bug detection in persistent memory programs;

PMTest

AGAMOTTO

Agamotto: How Persistent is your Persistent Memory Application?;