CHAPTER 12

Debugging Persistent Memory Applications

Persistent memory programming introduces new opportunities that allow developers to directly persist data structures without serialization and to access them in place without involving classic block I/O. As a result, you can merge your data models and avoid the classic split between data in memory—which is volatile, fast, and byte-addressable—with data on traditional storage devices, which is non-volatile but slower.

Persistent memory programming also brings challenges. Recall our discussion about power-fail protected persistence domains in Chapter 2: When a process or system crashes on an Asynchronous DRAM Refresh (ADR) enabled platform, data residing in the CPU caches that has not yet flushed to the persistent memory media is lost. This is not a problem with volatile memory because all the memory hierarchy is volatile. With persistent memory, however, a crash can cause permanent data corruption. How often must you flush data? Flushing too frequently yields sub-optimal performance, and not flushing often enough leaves the potential for data corruption.

Chapter 11 described several approaches to designing data structures and using methods such as copy-on-write, versioning, and transactions to maintain data integrity. Many libraries within the Persistent Memory Development Kit (PMDK) provide transactional updates of data structures and variables. These libraries provide optimal CPU cache flushing, when required by the platform, at precisely the right time, so you can program without concern about the hardware intricacies.

This programming paradigm introduces new dimensions related to errors and performance issues that programmers need to be aware of. The PMDK libraries reduce errors in persistent memory programming, but they cannot eliminate them. This chapter describes common persistent memory programming issues and pitfalls

and how to correct them using the tools available. The first half of this chapter introduces the tools. The second half presents several erroneous programming scenarios and describes how to use the tools to correct the mistakes before releasing your code into production.

pmemcheck for Valgrind

`pmemcheck` is a Valgrind (http://www.valgrind.org/) tool developed by Intel. It is very similar to `memcheck`, which is the default tool in Valgrind to discover memory-related bugs but adapted for persistent memory. Valgrind is an instrumentation framework for building dynamic analysis tools. Some Valgrind tools can automatically detect many memory management and threading bugs and profile your programs in detail. You can also use Valgrind to build new tools.

To run `pmemcheck`, you need a modified version of Valgrind supporting the new `CLFLUSHOPT` and `CLWB` flushing instructions. The persistent memory version of Valgrind includes the `pmemcheck` tool and is available from https://github.com/pmem/valgrind. Refer to the README.md within the GitHub project for installation instructions.

All the libraries in PMDK are already instrumented with `pmemcheck`. If you use PMDK for persistent memory programming, you will be able to easily check your code with `pmemcheck` without any code modification.

Before we discuss the `pmemcheck` details, the following two sections demonstrate how it identifies errors in an out-of-bounds and a memory leak example.

Stack Overflow Example

An out-of-bounds scenario is a stack/buffer overflow bug, where data is written or read beyond the capacity of the stack or array. Consider the small code snippet in Listing 12-1.

Listing 12-1. stackoverflow.c: Example of an out-of-bound bug.

```
32  #include <stdlib.h>
33
34  int main() {
```

```
35          int *stack = malloc(100 * sizeof(int));
36          stack[100] = 1234;
37          free(stack);
38      return 0;
39  }
```

In line 36, we are incorrectly assigning the value `1234` to the position `100`, which is outside the array range of `0-99`. If we compile and run this code, it may not fail. This is because, even if we only allocated 400 bytes (100 integers) for our array, the operating system provides a whole memory page, typically 4KiB. Executing the binary under Valgrind reports an issue, shown in Listing 12-2:

Listing 12-2. Running Valgrind with code Listing 12-1.

```
$ valgrind ./stackoverflow
==4188== Memcheck, a memory error detector
...
==4188== Invalid write of size 4
==4188==    at 0x400556: main (stackoverflow.c:36)
==4188==  Address 0x51f91d0 is 0 bytes after a block of size
400 alloc'd
==4188==    at 0x4C2EB37: malloc (vg_replace_malloc.c:299)
==4188==    by 0x400547: main (stackoverflow.c:35)
...
==4188== ERROR SUMMARY: 1 errors from 1 contexts (suppressed:
0 from 0)
```

Because Valgrind can produce long reports, we show only the relevant *"Invalid write"* error part of the report. When compiling code with symbol information (`gcc -g`), it is easy to see the exact place in the code where the error is detected. In this case, Valgrind highlights line 36 of the `stackoverflow.c` file. With the issue identified in the code, we know where to fix it.

Memory Leak Example

Memory leaks are another common issue. Consider the code in Listing 12-3.

Listing 12-3. leak.c: Example of a memory leak.

```
32  #include <stdlib.h>
33
34  void func(void) {
35      int *stack = malloc(100 * sizeof(int));
36  }
37
38  int main(void) {
39      func();
40      return 0;
41  }
```

The memory allocation is moved to the function `func()`. A memory leak occurs because the pointer to the newly allocated memory is a local variable on line 35, which is lost when the function returns. Executing this program under Valgrind shows the results n Listing 12-4:

Listing 12-4. Running Valgrind with code Listing 12-3.

```
$ valgrind --leak-check=yes ./leak
==4413== Memcheck, a memory error detector
...
==4413== 400 bytes in 1 blocks are definitely lost in loss
record 1 of 1
==4413==    at 0x4C2EB37: malloc (vg_replace_malloc.c:299)
==4413==    by 0x4004F7: func (leak.c:35)
==4413==    by 0x400507: main (leak.c:39)
==4413==
==4413== LEAK SUMMARY:
...
==4413== ERROR SUMMARY: 1 errors from 1 contexts (suppressed:
0 from 0)
```

Valgrind shows a loss of 400 bytes of memory allocated at `leak.c:35`. To learn more, please visit the official Valgrind documentation (http://www.valgrind.org/docs/manual/index.html).

Intel® Inspector - Persistence Inspector

Intel® Inspector - Persistence Inspector is a run-time tool that developers use to detect programming errors in persistent memory programs. In addition to cache flush misses, this tool detects:

- Redundant cache flushes and memory fences
- Out-of-order persistent memory stores
- Incorrect undo logging for the PMDK

Persistence Inspector is included as part of Intel® Inspector, an easy-to-use memory and threading error debugger for C, C++, and Fortran that works with both Windows* and Linux operating systems. It has an intuitive graphical and command-line interfaces, and it can be integrated with Microsoft Visual Studio*. Intel® Inspector is available as part of Intel® Parallel Studio XE (https://software.intel.com/en-us/parallel-studio-xe) and Intel® System Studio (https://software.intel.com/en-us/system-studio).

This section describes how the Intel® Inspector tool works with the same out-of-bounds and memory leak examples from Listings 12-1 and 12-3.

Stack Overflow Example

The Listing 12-5 example demonstrates how to use the command-line interface to perform the analysis and collect the data, and then switches to the GUI to examine the results in detail. To collect the data, we use the `inspxe-cl` utility with the `-c=mi2` collection option for detecting memory problems:

Listing 12-5. Running Intel® Inspector with code Listing 12-1.

```
$ inspxe-cl -c=mi2 -- ./stackoverflow

1 new problem(s) found
    1 Invalid memory access problem(s) detected
```

Intel Inspector creates a new directory with the data and analysis results, and prints a summary of findings to the terminal. For the stackoverflow app, it detected one invalid memory access.

After launching the GUI using `inspxe-GUI`, we open the results collection through the *File -> Open -> Result* menu, and navigate to the directory created by `inspxe-cli`. The directory will be named `r000mi2` if it is the first run. Within the directory is a file named `r000mi2.inspxe`. Once opened and processed, the GUI presents the data shown in Figure 12-1.

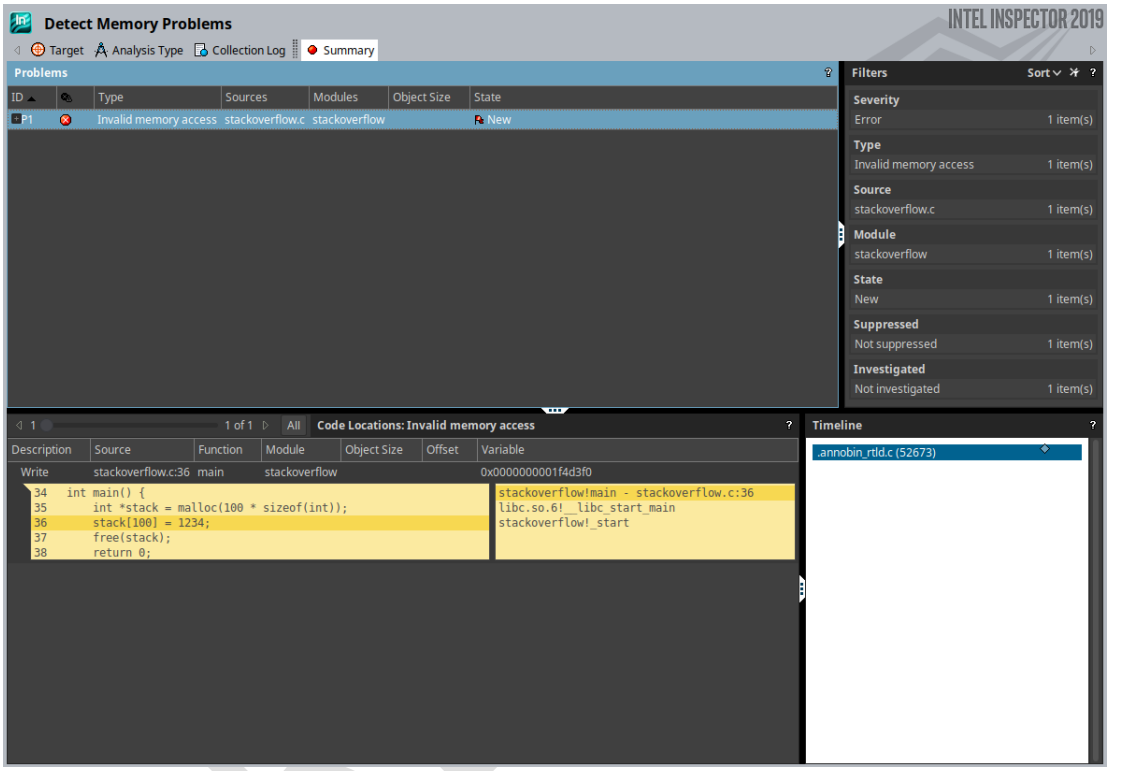

Figure 12-1. GUI of Intel® Inspector showing results for Listing 12-1.

The GUI defaults to the *Summary* tab to provide an overview of the analysis. Since we compiled the program with symbols, the *Code Locations* panel at the bottom shows the exact place in the code where the problem was detected. Intel Inspector identified the same error on line 36 that Valgrind found.

If Intel Inspector detects multiple problems within the program, those issues are listed in the *Problems* section in the upper-left area of the window. You can select each problem and see the information relating to it in the other sections of the window.

Memory Leak Example

The Listing 12-6 example runs Intel® Inspector using the `leak.c` code from Listing 12-2 and uses the same arguments from the `stackoverflow` program to detect memory issues.

Listing 12-6. Running Intel® Inspector with code Listing 12-2.

```
$ inspxe-cl -c=mi2 -- ./leak

1 new problem(s) found
    1 Memory leak problem(s) detected
```

The Intel Inspector output is shown in Figure 12-2 and explains that a memory leak problem was detected. When we pen the `r001mi2/r001mi2.inspxe` result file in the GUI, we get something similar to what is shown in the lower left section of Figure 12-2.

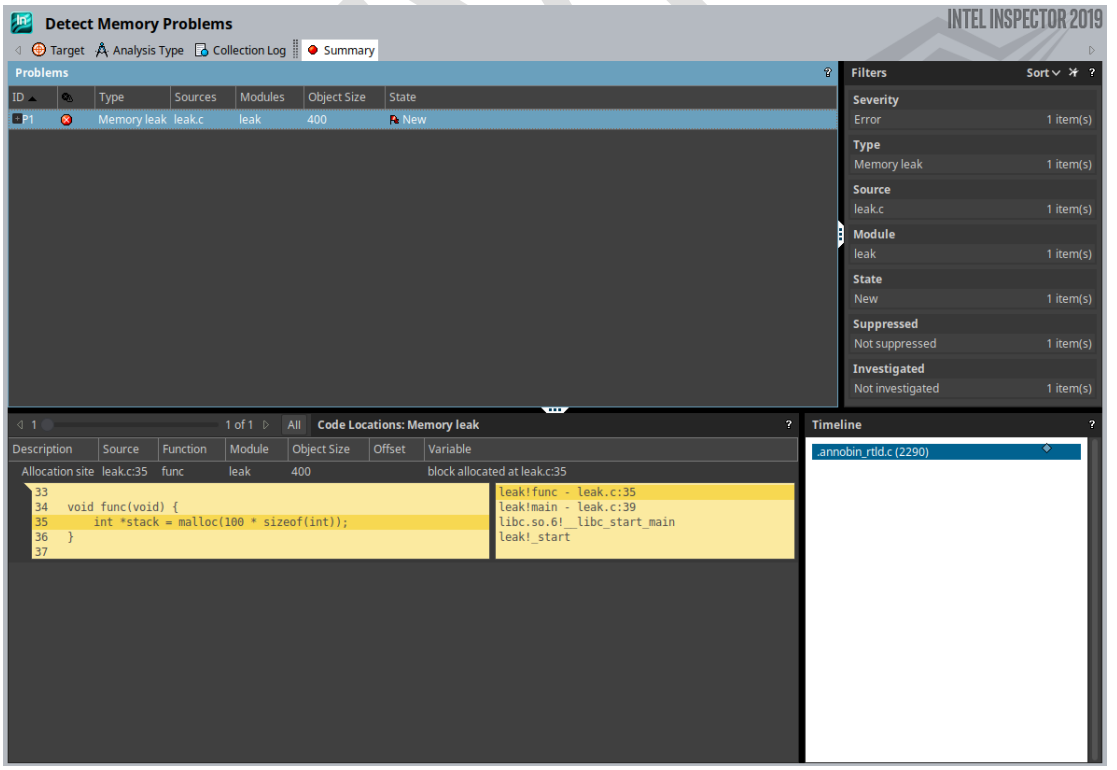

Figure 12-2. GUI of Intel® Inspector showing results for Listing 12-2.

The information related to the leaked object is shown above the code listing:

- Allocation site (source, function name, and module)
- Object size (400 bytes)
- The variable name that caused the leak.

The right side of the *Code* panel shows the call stack that led to the bug (call stacks are read from bottom to top). We see the call to `func()` in the `main()` function on line 39 (`leak.c:39`), then the memory allocation occurs within `func()` on line 35 (`leak.c:35`).

The Intel Inspector offers much more than what we presented here. To learn more, please visit the documentation (https://software.intel.com/en-us/intel-inspector-support/documentation).

Common Persistent Memory Programming Problems

This section reviews several coding and performance problems you are likely to encounter, how to catch them using the `pmemcheck` and Intel Inspector tools, and how to resolve the issues.

The tools we use highlight deliberately added issues in our code that can cause bugs, data corruption, or other problems. For `pmemcheck`, we show how to bypass data sections that should not be checked by the tool and use macros to assist the tool in better understanding our intent.

Non-Persistent Stores

Non-persistent stores refer to data written to persistent memory but not flushed explicitly. It is understood that if the program writes to persistent memory, it wishes for those writes to be persistent. If the program ends without explicitly flushing writes, there is an open possibility for data corruption. When a program exits gracefully, all the pending writes in the CPU caches are flushed automatically. However, if the

program were to crash unexpectedly, writes still residing on the CPU caches could be lost.

Consider the code in Listing 12-7 that writes data to a persistent memory device mounted to `/mnt/pmem` without flushing the data.

Listing 12-7. Example of writing to persistent memory without flushing.

```
32  #include <stdio.h>
33  #include <sys/mman.h>
34  #include <fcntl.h>
35
36  int main(int argc, char *argv[]) {
37      int fd, *data;
38      fd = open("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
39      posix_fallocate(fd, 0, sizeof(int));
40      data = (int *) mmap(NULL, sizeof(int), PROT_READ |
41                          PROT_WRITE, MAP_SHARED_VALIDATE |
42                          MAP_SYNC, fd, 0);
43      *data = 1234;
44      munmap(data, sizeof(int));
45      return 0;
46  }
```

- Line 38: We open `/mnt/pmem/file`
- Line 39: We make sure there is enough space in the file to allocate an integer by calling `posix_fallocate()`
- Line 40: We memory map `/mnt/pmem/file`
- Line 43: We write `1234` to the memory
- Line 44: We unmap the memory

If we run `pmemcheck` with Listing 12-7, we will not get any useful information because `pmemcheck` has no way to know which memory addresses are persistent and which ones are volatile. This may change in future versions. To run `pmemcheck`, we pass `--tool=pmemcheck` argument to `valgrind` as shown in Listing 12-8. The result shows no issues were detected.

Listing 12-8.: Running pmemcheck with code Listing 12-7.

```
$ valgrind --tool=pmemcheck ./listing_12-7
==116951== pmemcheck-1.0, a simple persistent store checker
==116951== Copyright (c) 2014-2016, Intel Corporation
==116951== Using Valgrind-3.14.0 and LibVEX; rerun with -h for
copyright info
==116951== Command: ./listing_12-9
==116951==
==116951==
==116951== Number of stores not made persistent: 0
==116951== ERROR SUMMARY: 0 errors
```

We can inform `pmemcheck` which memory regions are persistent using a `VALGRIND_PMC_REGISTER_PMEM_MAPPING` macro shown on line 52 in Listing 12-9. We must include the `valgrind/pmemcheck.h` header for pmemcheck, line 36, that defines the `VALGRIND_PMC_REGISTER_PMEM_MAPPING` macro and others.

Listing 12-9. Example of writing to persistent memory using Valgrind macros without flushing.

```
33  #include <stdio.h>
34  #include <sys/mman.h>
35  #include <fcntl.h>
36  #include <valgrind/pmemcheck.h>
37
38  int main(int argc, char *argv[]) {
39      int fd, *data;
40
41      // open the file and allocate enough space for an
42      // integer
43      fd = open("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
44      posix_fallocate(fd, 0, sizeof(int));
45
46      // memory map the file and register the mapped
47      // memory with VALGRIND
48      data = (int *) mmap(NULL, sizeof(int),
49              PROT_READ|PROT_WRITE,
50              MAP_SHARED_VALIDATE | MAP_SYNC,
51              fd, 0);
52      VALGRIND_PMC_REGISTER_PMEM_MAPPING(data,
```

```
53                                      sizeof(int));
54
55      // write to pmem
56      *data = 1234;
57
58      // unmap the memory and un-register it with
59      // VALGRIND
60      munmap(data, sizeof(int));
61      VALGRIND_PMC_REMOVE_PMEM_MAPPING(data,
62                                              sizeof(int));
63      return 0;
64  }
```

We remove persistent memory mapping identification from `pmemcheck` using the `VALGRIND_PMC_REMOVE_PMEM_MAPPING` macro. As mentioned above, this is useful when you want to exclude parts of persistent memory from the analysis. Listing 12-10 shows executing `pmemcheck` with the modified code in Listing 12-9, which now reports a problem.

Listing 12-10. Running pmemcheck with code Listing 12-9.

```
$ valgrind --tool=pmemcheck ./listing_12-9
==8904== pmemcheck-1.0, a simple persistent store checker
...
==8904== Number of stores not made persistent: 1
==8904== Stores not made persistent properly:
==8904== [0]    at 0x4008B4: main (listing_12-9.c:56)
==8904==        Address: 0x4027000      size: 4 state: DIRTY
==8904== Total memory not made persistent: 4
==8904== ERROR SUMMARY: 1 errors
```

See that `pmemcheck` detected that data is not being flushed after a write in `listing_12-9.c`, line 56. To fix this, we create a new `flush()` function, accepting an address and size, to flush all the CPU cache lines storing any part of the data using the CLFLUSH machine instruction (`__mm_clflush()`). Listing 12-11 shows the modified code.

Listing 12-11. Example of writing to persistent memory using Valgrind with flushing.

```
33  #include <emmintrin.h>
34  #include <stdint.h>
```

```
35  #include <stdio.h>
36  #include <sys/mman.h>
37  #include <fcntl.h>
38  #include <valgrind/pmemcheck.h>
39
40  // flushing from user space
41  void flush(const void *addr, size_t len) {
42      uintptr_t flush_align = 64, uptr;
43      for (uptr = (uintptr_t)addr & ~(flush_align - 1);
44               uptr < (uintptr_t)addr + len;
45               uptr += flush_align)
46          _mm_clflush((char *)uptr);
47  }
48
49  int main(int argc, char *argv[]) {
50      int fd, *data;
51
52      // open the file and allocate space for one
53      // integer
54      fd = open("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
55      posix_fallocate(fd, 0, sizeof(int));
56
57      // map the file and register it with VALGRIND
58      data = (int *)mmap(NULL, sizeof(int),
59              PROT_READ | PROT_WRITE,
60              MAP_SHARED_VALIDATE | MAP_SYNC, fd, 0);
61      VALGRIND_PMC_REGISTER_PMEM_MAPPING(data,
62                                         sizeof(int));
63
64      // write and flush
65      *data = 1234;
66      flush((void *)data, sizeof(int));
67
68      // unmap and un-register
69      munmap(data, sizeof(int));
70      VALGRIND_PMC_REMOVE_PMEM_MAPPING(data,
71                                       sizeof(int));
72      return 0;
73  }
```

Running the modified code through `pmemcheck` reports no issues, as shown in Listing 12-12.

Listing 12-12. Running pmemcheck with code Listing 12-11.

```
$ valgrind --tool=pmemcheck ./listing_12-11
==9710== pmemcheck-1.0, a simple persistent store checker
...
==9710== Number of stores not made persistent: 0
==9710== ERROR SUMMARY: 0 errors
```

Because Intel Inspector - Persistence Inspector does not consider an unflushed write a problem unless there is a write dependency with other variables, we need to show a more complex example than writing a single variable in Listing 12-7. You need to understand how programs writing to persistent memory are designed to know which parts of the data written to the persistent media are valid and which parts are not. Remember that recent writes may still be sitting on the CPU caches if they are not explicitly flushed.

Transactions solve the problem of half-written data by using logs to either roll-back or apply uncommitted changes; thus, programs reading the data back can be assured that everything written is valid. In the absence of transactions, it is impossible to know whether or not the data written on persistent memory is valid, especially if the program crashes.

A writer can inform a reader that data is properly written in one of two ways, either by setting a “valid” flag, or by using a watermark variable with the address (or the index, in the case of an array) of the last valid written memory position.

Listing 12-13 shows pseudo-code for how the “valid” flag approach could be implemented.

Listing 12-13. Pseudo code showcasing write dependency of var1 with var1_valid.

```
1  writer() {
2          var1 = "This is a persistent Hello World
3                  written to persistent memory!";
4          flush (var1);
5          var1_valid = True;
6          flush (var1_valid);
7  }
8
9  reader() {
10          if (var1_valid == True) {
11                  print (var1);
12          }
14  }
```

The `reader()` will read the data in `var1` if the `var1_valid` flag is set to `True` (line 10), and `var1_valid` can only be `True` if `var1` has been flushed (lines 4 and 5).

We can now modify the code from Listing 12-7 to introduce this "valid" flag. In Listing 12-14, we separate the code into writer and reader programs and map two integers instead of one (to accommodate for the flag). Listing 12-15 shows the *reading* to persistent memory example.

Listing 12-14. Example of writing to persistent memory with a write dependency; the code does not flush.

```
33  #include <stdio.h>
34  #include <sys/mman.h>
35  #include <fcntl.h>
36  #include <string.h>
37
38  int main(int argc, char *argv[]) {
39      int fd, *ptr, *data, *flag;
40
41      fd = open("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
42      posix_fallocate(fd, 0, sizeof(int)*2);
43
44      ptr = (int *) mmap(NULL, sizeof(int)*2,
45                         PROT_READ | PROT_WRITE,
46                         MAP_SHARED_VALIDATE | MAP_SYNC,
47                         fd, 0);
48
49      data = &(ptr[1]);
50      flag = &(ptr[0]);
51      *data = 1234;
52      *flag = 1;
53
54      munmap(ptr, 2 * sizeof(int));
55      return 0;
56  }
```

Listing 12-15. Example of reading from persistent memory with a write dependency.

```
33  #include <stdio.h>
34  #include <sys/mman.h>
35  #include <fcntl.h>
```

```
36
37  int main(int argc, char *argv[]) {
38      int fd, *ptr, *data, *flag;
39
40      fd = open("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
41      posix_fallocate(fd, 0, 2 * sizeof(int));
42
43      ptr = (int *) mmap(NULL, 2 * sizeof(int),
44                         PROT_READ | PROT_WRITE,
45                         MAP_SHARED_VALIDATE | MAP_SYNC,
46                         fd, 0);
47
48      data = &(ptr[1]);
49      flag = &(ptr[0]);
50      if (*flag == 1)
51          printf("data = %d\n", *data);
52
53      munmap(ptr, 2 * sizeof(int));
54      return 0;
55  }
```

Checking our code with Persistence Inspector is done in three steps.

Step 1: We must run the before-unfortunate-event phase analysis (see Listing 12-16), which corresponds to the writer code in Listing 12-14.

Listing 12-16. Running Intel® Inspector – Persistence Inspector with code Listing 12-14 for before-unfortunate-event phase analysis.

```
$ pmeminsp cb -pmem-file /mnt/pmem/file -- ./listing_12-14
++ Analysis starts


++ Analysis completes
++ Data is stored in folder
"/data/.pmeminspdata/data/listing_12-14"
```

The parameter `cb` is an abbreviation of *check-before-unfortunate-event*, which specifies the type of analysis. We must also pass the persistent memory file that will be used by the application so that Persistence Inspector knows which memory accesses correspond to persistent memory. By default, the output of the analysis is

stored in a local directory under the `.pmeminspdata` directory. (You can also specify a custom directory; run `pmeminsp -help` for information on the available options.)

Step 2: We run the *after-unfortunate-event* phase analysis (see Listing 12-17). This corresponds to the code that will read the data after an unfortunate event happens, such as a process crash.

Listing 12-17. Running Intel® Inspector – Persistence Inspector with code listing 12-15 for after-unfortunate-event phase analysis.

```
$ pmeminsp ca -pmem-file /mnt/pmem/file -- ./listing_12-15
++ Analysis starts

data = 1234

++ Analysis completes
++ Data is stored in folder
"/data/.pmeminspdata/data/listing_12-15"
```

The parameter `ca` is an abbreviation of *check-after-unfortunate-event*. Again, the output of the analysis is stored in `.pmeminspdata` within the current working directory.

Step 3: We generate the final report. For this, we pass the option `rp` (abbreviation for `report`) along with the name of both programs, as shown in Listing 12-18.

Listing 12-18: Generating a final report with Intel® Inspector – Persistence Inspector from the analysis done in Listings 12-16 and 12-17.

```
$ pmeminsp rp -- listing_12-16 listing_12-17
#=============================================================
# Diagnostic # 1: Missing cache flush
#-------------------
  The first memory store
    of size 4 at address 0x7F9C68893004 (offset 0x4 in
/mnt/pmem/file)
    in /data/listing_12-16!main at listing_12-16.c:51 - 0x67D
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
```

```
    in /data/listing_12-16!_start at
<unknown_file>:<unknown_line> - 0x534

  is not flushed before

  the second memory store
    of size 4 at address 0x7F9C68893000 (offset 0x0 in
/mnt/pmem/file)
    in /data/listing_12-16!main at listing_12-16.c:52 - 0x687
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
    in /data/listing_12-16!_start at
<unknown_file>:<unknown_line> - 0x534

  while

  memory load from the location of the first store
    in /data/listing_12-17!main at listing_12-17.c:51 - 0x6C8

  depends on

  memory load from the location of the second store
    in /data/listing_12-17!main at listing_12-17.c:50 - 0x6BD

#============================================================
# Diagnostic # 2: Missing cache flush
#------------------
  Memory store
    of size 4 at address 0x7F9C68893000 (offset 0x0 in
/mnt/pmem/file)
    in /data/listing_12-16!main at listing_12-16.c:52 - 0x687
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
    in /data/listing_12-16!_start at
<unknown_file>:<unknown_line> - 0x534

  is not flushed before

  memory is unmapped
    in /data/listing_12-16!main at listing_12-16.c:54 - 0x699
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
```

```
    in /data/listing_12-16!_start at
<unknown_file>:<unknown_line> - 0x534


Analysis complete. 2 diagnostic(s) reported.
```

The output is very verbose, but it is easy to follow. We get two missing cache flushes (diagnostics 1 and 2) corresponding to lines 51 and 52 of `listing_12-16.c`. We do these writes to the locations in the mapped persistent memory pointed by variables `flag` and `data`. The first diagnostic says that the first memory store is not flushed before the second store while, at the same time, there is a load dependency of the first store to the second. This is exactly what we intended.

The second diagnostic says that the second store (to the flag) itself is never actually flushed before ending. Even if we flush the first store correctly before we write the flag, we must still flush the flag to make sure the dependency works.

To open the results in the Intel Inspector GUI, you can use the `-insp` option when generating the report; for example:

```
$ pmeminsp rp -insp -- listing_12-16 listing_12-17
```

This generates a directory called `r000pmem` inside the analysis directory (`.pmeminspdata` by default). Launch the GUI running `inspxe-gui` and open the result file by going to *File -> Open -> Result,* and selecting the file `r000pmem/r000pmem.inspxe`. You should see something similar to what is shown in Figure 12-3.

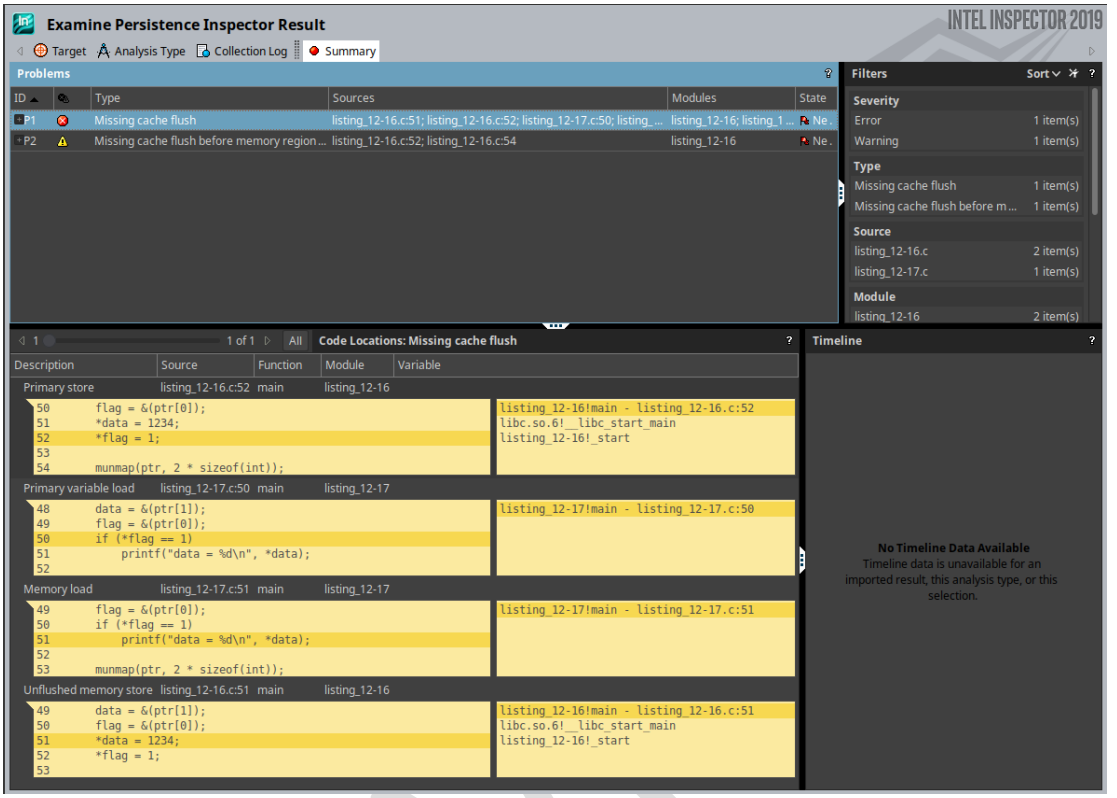

Figure 12-3. GUI of Intel® Inspector showing results for Listing 12-18 (diagnostic 1).

The GUI shows the same information as the command line analysis but in a more readable way by highlighting the errors directly on our source code. As Figure 12-3 shows, the modification of the flag is called “primary store.”

In Figure 12-4, the second diagnosis is selected in the `Problems` pane, showing the missing flush for the flag itself.

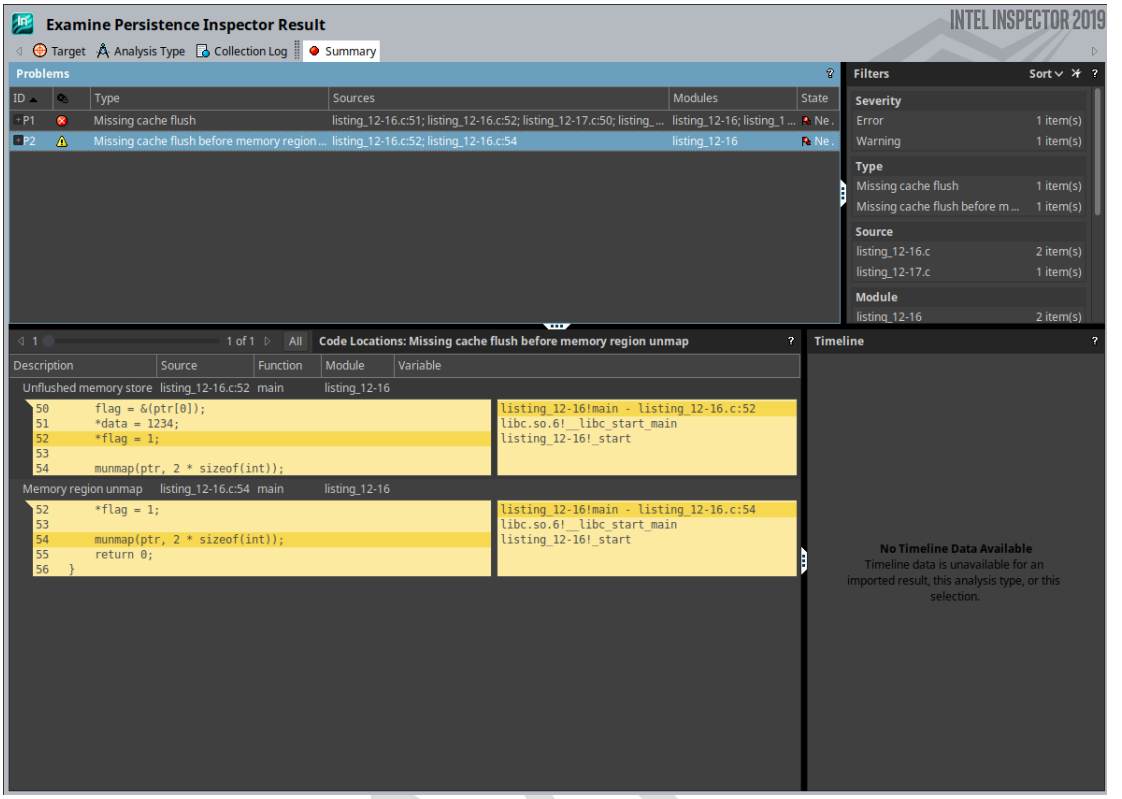

Figure 12-4. GUI of Intel® Inspector showing results for Listing 12-20 (diagnostic #2).

To conclude this section, we fix the code and rerun the analysis with Persistence Inspector. The code in Listing 12-19 adds the necessary flushes to Listing 12-14.

Listing 12-19. Example of writing to persistent memory with a write dependency. The code flushes both writes.

```
33  #include <emmintrin.h>
34  #include <stdint.h>
35  #include <stdio.h>
36  #include <sys/mman.h>
37  #include <fcntl.h>
38  #include <string.h>
39
40  void flush(const void *addr, size_t len) {
41      uintptr_t flush_align = 64, uptr;
42      for (uptr = (uintptr_t)addr & ~(flush_align - 1);
43              uptr < (uintptr_t)addr + len;
```

```
44              uptr += flush_align)
45          _mm_clflush((char *)uptr);
46  }
47
48  int main(int argc, char *argv[]) {
49      int fd, *ptr, *data, *flag;
50
51      fd = open("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
52      posix_fallocate(fd, 0, sizeof(int) * 2);
53
54      ptr = (int *) mmap(NULL, sizeof(int) * 2,
55                         PROT_READ | PROT_WRITE,
56                         MAP_SHARED_VALIDATE | MAP_SYNC,
57                         fd, 0);
58
59      data = &(ptr[1]);
60      flag = &(ptr[0]);
61      *data = 1234;
62      flush((void *) data, sizeof(int));
63      *flag = 1;
64      flush((void *) flag, sizeof(int));
65
66      munmap(ptr, 2 * sizeof(int));
67      return 0;
68  }
```

Listing 12-20 executes Persistence Inspector against the modified code from Listing 12-19, then the reader code from Listing 12-15, and finally running the report, which says that no problems were detected.

Listing 12-20. Running full analysis with Intel® Inspector – Persistence Inspector with code Listings 12-19 and 12-15.

```
$ pmeminsp cb -pmem-file /mnt/pmem/file -- ./listing_12-19
++ Analysis starts


++ Analysis completes
++ Data is stored in folder
"/data/.pmeminspdata/data/listing_12-19"
```

```
$ pmeminsp ca -pmem-file /mnt/pmem/file -- ./listing_12-15
++ Analysis starts

data = 1234

++ Analysis completes
++ Data is stored in folder
"/data/.pmeminspdata/data/listing_12-15"


$ pmeminsp rp -- listing_12-19 listing_12-15
Analysis complete. No problems detected.
```

Stores Not Added into a Transaction

When working within a transaction block, it is assumed that all the modified persistent memory addresses were added to it at the beginning, which also implies that their previous values are copied to an undo log. This allows the transaction to implicitly flush added memory addresses 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 an address that is not added to the transaction is a bug that you must be aware of.

Consider the code in Listing 12-21 that uses the `libpmemobj` library from PMDK. It shows an example of writing within a transaction using a memory address that is not explicitly tracked by the transaction.

Listing 12-21. Example of writing within a transaction with memory address not added to the transaction.

```
33  #include <libpmemobj.h>
34
35  struct my_root {
36      int value;
37      int is_odd;
38  };
39
40  // registering type 'my_root' in the layout
41  POBJ_LAYOUT_BEGIN(example);
42  POBJ_LAYOUT_ROOT(example, struct my_root);
43  POBJ_LAYOUT_END(example);
44
```

```
45  int main(int argc, char *argv[]) {
46      // creating the pool
47      PMEMobjpool *pop= pmemobj_create("/mnt/pmem/pool",
48                          POBJ_LAYOUT_NAME(example),
49                          (1024 * 1024 * 100), 0666);
50
51      // transation
52      TX_BEGIN(pop) {
53          TOID(struct my_root) root
54              = POBJ_ROOT(pop, struct my_root);
55
56          // adding root.value to the transaction
57          TX_ADD_FIELD(root, value);
58
59          D_RW(root)->value = 4;
60          D_RW(root)->is_odd = D_RO(root)->value % 2;
61      } TX_END
62
63      return 0;
64  }
```

Note: For a refresh on the definitions of a layout, root object, or macros used in Listing 12-21, see Chapter 7 where we introduce `libpmemobj`.

In lines 35-38, we create a `my_root` data structure, which has two integer members: `value` and `is_odd`. These integers are modified inside a transaction (lines 52-61), setting `value=4` and `is_odd=0`. On line 57, we are only adding the `value` variable to the transaction, leaving `is_odd` out. Given that persistent memory is not natively supported in C, there is no way for the compiler to warn you about this. The compiler cannot distinguish between pointers to volatile memory versus those to persistent memory.

Listing 12-22 shows the response from running the code through `pmemcheck`.

Listing 12-22. Running pmemcheck with code Listing 12-21.

```
$ valgrind --tool=pmemcheck ./listing_12-21
==48660== pmemcheck-1.0, a simple persistent store checker
==48660== Copyright (c) 2014-2016, Intel Corporation
==48660== Using Valgrind-3.14.0 and LibVEX; rerun with -h for
copyright info
```

```
==48660== Command: ./listing_12-21
==48660==
==48660==
==48660== Number of stores not made persistent: 1
==48660== Stores not made persistent properly:
==48660== [0]    at 0x400C2D: main (listing_12-25.c:60)
==48660==       Address: 0x7dc0554      size: 4 state: DIRTY
==48660== Total memory not made persistent: 4
==48660==
==48660== Number of stores made without adding to transaction:
1
==48660== Stores made without adding to transactions:
==48660== [0]    at 0x400C2D: main (listing_12-25.c:60)
==48660==       Address: 0x7dc0554      size: 4
==48660== ERROR SUMMARY: 2 errors
```

Although they are both related to the same root cause, `pmemcheck` identified two issues. One is the error we expected; that is, we have a store inside a transaction that was not added to it. The other error says that we are not flushing the store. Since transactional stores are flushed automatically when the program exits the transaction, finding two errors per store to a location not included within a transaction should be common in `pmemcheck`.

Persistence Inspector has a more user-friendly output, as shown in Listing 12-23.

Listing 12-23. Generating a report with Intel® Inspector – Persistence Inspector for code Listing 12-21.

```
$ pmeminsp cb -pmem-file /mnt/pmem/pool -- ./listing_12-21
++ Analysis starts


++ Analysis completes
++ Data is stored in folder
"/data/.pmeminspdata/data/listing_12-21"
$
$ pmeminsp rp -- ./listing_12-21
#============================================================
# Diagnostic # 1: Store without undo log
#-------------------
  Memory store
```

```
    of size 4 at address 0x7FAA84DC0554 (offset 0x3C0554 in
/mnt/pmem/pool)
    in /data/listing_12-21!main at listing_12-21.c:60 - 0xC2D
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
    in /data/listing_12-21!_start at
<unknown_file>:<unknown_line> - 0x954

  is not undo logged in

  transaction
    in /data/listing_12-21!main at listing_12-21.c:52 - 0xB67
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
    in /data/listing_12-21!_start at
<unknown_file>:<unknown_line> - 0x954


Analysis complete. 1 diagnostic(s) reported.
```

We do not perform an *after-unfortunate-event* phase analysis here because we are only concerned about transactions.

We can fix the problem reported in Listing 12-23 by adding the whole root object to the transaction using `TX_ADD(root)`, as shown on line 53 in Listing 12-24.

Listing 12-24. Example of adding an object and writing it within a transaction.

```
32  #include <libpmemobj.h>
33
34  struct my_root {
35      int value;
36      int is_odd;
37  };
38
39  POBJ_LAYOUT_BEGIN(example);
40  POBJ_LAYOUT_ROOT(example, struct my_root);
41  POBJ_LAYOUT_END(example);
42
43  int main(int argc, char *argv[]) {
44      PMEMobjpool *pop= pmemobj_create("/mnt/pmem/pool",
```

```
45                          POBJ_LAYOUT_NAME(example),
46                          (1024 * 1024 * 100), 0666);
47
48      TX_BEGIN(pop) {
49          TOID(struct my_root) root
50              = POBJ_ROOT(pop, struct my_root);
51
52          // adding full root to the transaction
53          TX_ADD(root);
54
55          D_RW(root)->value = 4;
56          D_RW(root)->is_odd = D_RO(root)->value % 2;
57      } TX_END
58
59      return 0;
60  }
```

If we run the code through `pmemcheck`, as shown in Listing 12-25, no issues are reported.

Listing 12-25. Running pmemcheck with code Listing 12-24.

```
$ valgrind --tool=pmemcheck ./listing_12-24
==80721== pmemcheck-1.0, a simple persistent store checker
==80721== Copyright (c) 2014-2016, Intel Corporation
==80721== Using Valgrind-3.14.0 and LibVEX; rerun with -h for
copyright info
==80721== Command: ./listing_12-24
==80721==
==80721==
==80721== Number of stores not made persistent: 0
==80721== ERROR SUMMARY: 0 errors
```

Similarly, no issues are reported by Persistence Inspector in Listing 12-25.

Listing 12-25. Generating report with Intel® Inspector – Persistence Inspector for code listing 12-24.

```
$ pmeminsp cb -pmem-file /mnt/pmem/pool -- ./listing_12-24
++ Analysis starts


++ Analysis completes
```

```
++ Data is stored in folder
"/data/.pmeminspdata/data/listing_12-24"
$
$ pmeminsp rp -- ./listing_12-24
Analysis complete. No problems detected.
```

After properly adding all the memory that will be modified to the transaction, both tools report that no problems were found.

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 can potentially corrupt data. This can occur in PMDK, for example, where the library maintains a different transaction per thread. If two threads write to the same object within different transactions, after an application crash, a thread might overwrite modifications made by another thread in a different transaction. In database systems, this problem is known as *dirty reads.* Dirty reads violate the isolation requirement of the ACID (atomicity, consistency, isolation, durability) properties, as shown in Figure 12-5.

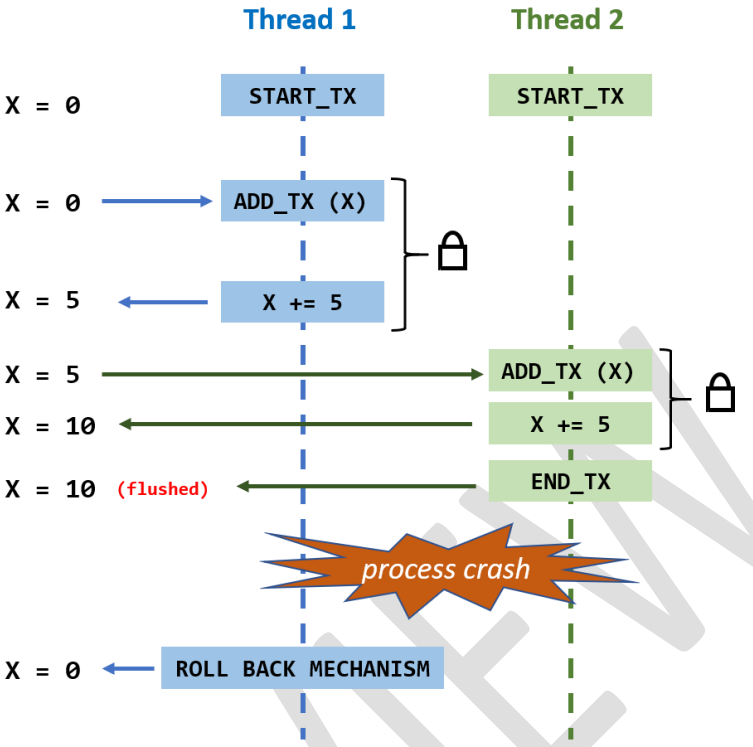

Figure 12-5. The rollback mechanism for the unfinished transaction in thread 1 is also overriding the changes made by thread 2, even though the transaction for thread 2 finishes correctly.

In Figure 12-5, time is shown in the y-axis with time progressing downward. These operations occur in the following order:

- Assume `X=0` when the application starts.
- A `main()` function creates two threads: Thread 1 and Thread 2. Both threads are intended to start their own transactions and acquire the lock to modify `X`.
- Since Thread 1 runs first, it acquires the lock on `X` first. It then adds the `X` variable to the transaction before incrementing `X` by 5. Transparent to the program, the value of `X` (`X=0`) is added to the undo log when `X` was added to the transaction. Since the transaction is not yet complete, the application has not yet explicitly flushed the value.

- Thread 2 starts, begins its own transaction, acquires the lock, reads the value of `X` (which is now 5), adds `X=5` to the undo log, and increments it by 5. The transaction completes successfully and Thread 2 flushes the CPU caches. Now, `x=10`.
- Unfortunately, the program crashes after Thread 2 successfully completes its transaction, but before Thread 1 was able to finish its transaction and flush its value.

This scenario leaves the application with an invalid, but consistent, value of `x=10`. Since transactions are atomic, all changes done within them are not valid until they successfully complete.

When the application starts, it knows it must perform a recovery operation due to the previous crash and will replay the undo logs to rewind the partial update made by Thread 1. The undo log restores the value of `X=0`, which was correct when Thread 1 added its entry. The expected value of `X` should be `X=5` in this situation, but the undo log puts `X=0`. You can probably see the huge potential for data corruption that this situation can produce.

We describe concurrency for multi-threaded applications in Chapter 14. Using `libpmemobj-cpp`, the C++ language binding library to `libpmemobj`, concurrency issues are very easy to resolve because the API allows us to pass a list of locks using lambda functions when transactions are created. Chapter 8 discusses `libpmemobj-cpp` and lambda functions in more detail.

Listing 12-26 shows how you can use a single mutex to lock a whole transaction. This mutex can either be a standard mutex (`std::mutex`) if the mutex object resides in volatile memory, or a pmem mutex (`pmem::obj::mutex`) if the mutex object resides in persistent memory.

Listing 12-26. Example of a libpmemobj++ transaction whose writes are both atomic—with respect to persistent memory—and isolated—in a multithreaded scenario. The mutex is passed to the transaction as a parameter.

```
transaction::run (pop, [&] {
      ...
      // all writes here are atomic and thread safe
      ...
 }, mutex);
```

Consider the code in Listing 12-27 that simultaneously adds the same memory region to two different transactions.

Listing 12-27. Example of two threads simultaneously adding the same persistent memory location to their respective transactions.

```
33  #include <libpmemobj.h>
34  #include <pthread.h>
35
36  struct my_root {
37      int value;
38      int is_odd;
39  };
40
41  POBJ_LAYOUT_BEGIN(example);
42  POBJ_LAYOUT_ROOT(example, struct my_root);
43  POBJ_LAYOUT_END(example);
44
45  pthread_mutex_t lock;
46
47  // function to be run by extra thread
48  void *func(void *args) {
49      PMEMobjpool *pop = (PMEMobjpool *) args;
50
51      TX_BEGIN(pop) {
52          pthread_mutex_lock(&lock);
53          TOID(struct my_root) root
54              = POBJ_ROOT(pop, struct my_root);
55          TX_ADD(root);
56          D_RW(root)->value = D_RO(root)->value + 3;
57          pthread_mutex_unlock(&lock);
58      } TX_END
59  }
60
61  int main(int argc, char *argv[]) {
62      PMEMobjpool *pop= pmemobj_create("/mnt/pmem/pool",
63                        POBJ_LAYOUT_NAME(example),
64                        (1024 * 1024 * 10), 0666);
65
66      pthread_t thread;
67      pthread_mutex_init(&lock, NULL);
68
69      TX_BEGIN(pop) {
70          pthread_mutex_lock(&lock);
71          TOID(struct my_root) root
72              = POBJ_ROOT(pop, struct my_root);
```

```
73          TX_ADD(root);
74          pthread_create(&thread, NULL,
75                         func, (void *) pop);
76          D_RW(root)->value = D_RO(root)->value + 4;
77          D_RW(root)->is_odd = D_RO(root)->value % 2;
78          pthread_mutex_unlock(&lock);
79          // wait to make sure other thread finishes 1st
80          pthread_join(thread, NULL);
81      } TX_END
82
83      pthread_mutex_destroy(&lock);
84      return 0;
85  }
```

- Line 69: The main thread starts a transaction and adds the root data structure to it (line 73).
- Line 74: We create a new thread by calling `pthread_create()` and have it execute the `func()` function. This function also starts a transaction (line 51) and adds the root data structure to it (line 55).
- Both threads will simultaneously modify all or part of the same data before finishing their transactions. We force the second thread to finish first by making the main thread wait on `pthread_join()`.

Listing 12-28 shows code execution with `pmemcheck`, and the result warns us that we have *overlapping regions registered in different transactions*.

Listing 12-28. Running pmemcheck with Listing 12-27.

```
$ valgrind --tool=pmemcheck ./listing_12-27
==97301== pmemcheck-1.0, a simple persistent store checker
==97301== Copyright (c) 2014-2016, Intel Corporation
==97301== Using Valgrind-3.14.0 and LibVEX; rerun with -h for
copyright info
==97301== Command: ./listing_12-27
==97301==
==97301==
==97301== Number of stores not made persistent: 0
==97301==
```

```
==97301== Number of overlapping regions registered in
different transactions: 1
==97301== Overlapping regions:
==97301== [0]    at 0x4E6B0BC: pmemobj_tx_add_snapshot (in
/usr/lib64/libpmemobj.so.1.0.0)
==97301==    by 0x4E6B5F8: pmemobj_tx_add_common.constprop.18
(in /usr/lib64/libpmemobj.so.1.0.0)
==97301==    by 0x4E6C62F: pmemobj_tx_add_range (in
/usr/lib64/libpmemobj.so.1.0.0)
==97301==    by 0x400DAC: func (listing_12-27.c:55)
==97301==    by 0x4C2DDD4: start_thread (in
/usr/lib64/libpthread-2.17.so)
==97301==    by 0x5180EAC: clone (in /usr/lib64/libc-2.17.so)
==97301==  Address: 0x7dc0550    size: 8    tx_id: 2
==97301==    First registered here:
==97301== [0]'   at 0x4E6B0BC: pmemobj_tx_add_snapshot (in
/usr/lib64/libpmemobj.so.1.0.0)
==97301==    by 0x4E6B5F8: pmemobj_tx_add_common.constprop.18
(in /usr/lib64/libpmemobj.so.1.0.0)
==97301==    by 0x4E6C62F: pmemobj_tx_add_range (in
/usr/lib64/libpmemobj.so.1.0.0)
==97301==    by 0x400F23: main (listing_12-27.c:73)
==97301==  Address: 0x7dc0550    size: 8    tx_id: 1
==97301== ERROR SUMMARY: 1 errors
```

Listing 12-29 shows the same code run with Persistence Inspector, which also reports *"Overlapping regions registered in different transactions"* in Diagnostic 25. The first twenty-four diagnostic results were related to stores not added to our transactions corresponding with the locking and unlocking of our volatile mutex; these can be ignored.

Listing 12-29. Generating report with Intel® Inspector – Persistence Inspector for code Listing 12-27.

```
$ pmeminsp rp -- ./listing_12-27
...
#=================================================================
# Diagnostic # 25: Overlapping regions registered in different
transactions
#------------------
  transaction
    in /data/listing_12-27!main at listing_12-27.c:69 - 0xEB6
```

```
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
    in /data/listing_12-27!_start at
<unknown_file>:<unknown_line> - 0xB44

  protects

  memory region
    in /data/listing_12-27!main at listing_12-27.c:73 - 0xF1F
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
    in /data/listing_12-27!_start at
<unknown_file>:<unknown_line> - 0xB44

  overlaps with

  memory region
    in /data/listing_12-27!func at listing_12-27.c:55 - 0xDA8
    in /lib64/libpthread.so.0!start_thread at
<unknown_file>:<unknown_line> - 0x7DCD
    in /lib64/libc.so.6!__clone at
<unknown_file>:<unknown_line> - 0xFDEAB


Analysis complete. 25 diagnostic(s) reported.
```

Memory Overwrites

Where multiple modifications to the same persistent memory location occur before the location is made persistent (that is, flushed), a memory overwrite occurs. This is a potential data corruption source if a program crashes because the final value of the persistent variable can be any of the values written between the last flush and the crash. It is important to know that this may not be an issue if it is in the code by design. We recommend using volatile variables for short-lived data, and only write to persistent variables when you want to persist data.

Consider the code in Listing 12-30, which writes twice to the `data` variable inside the `main()` function (lines 62 and 63) before we call `flush()` on line 64.

Listing 12-30. Example of persistent memory overwriting—variable data—before flushing.

```
33  #include <emmintrin.h>
34  #include <stdint.h>
35  #include <stdio.h>
36  #include <sys/mman.h>
37  #include <fcntl.h>
38  #include <valgrind/pmemcheck.h>
39
40  void flush(const void *addr, size_t len) {
41      uintptr_t flush_align = 64, uptr;
42      for (uptr = (uintptr_t)addr & ~(flush_align - 1);
43              uptr < (uintptr_t)addr + len;
44              uptr += flush_align)
45          _mm_clflush((char *)uptr);
46  }
47
48  int main(int argc, char *argv[]) {
49      int fd, *data;
50
51      fd = open("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
52      posix_fallocate(fd, 0, sizeof(int));
53
54      data = (int *)mmap(NULL, sizeof(int),
55              PROT_READ | PROT_WRITE,
56              MAP_SHARED_VALIDATE | MAP_SYNC,
57              fd, 0);
58      VALGRIND_PMC_REGISTER_PMEM_MAPPING(data,
59                                         sizeof(int));
60
61      // writing twice before flushing
62      *data = 1234;
63      *data = 4321;
64      flush((void *)data, sizeof(int));
65
66      munmap(data, sizeof(int));
67      VALGRIND_PMC_REMOVE_PMEM_MAPPING(data,
68                                       sizeof(int));
69      return 0;
70  }
```

Listing 12-31 shows the report from `pmemcheck` with the code from Listing 12-30. To make `pmemcheck` look for overwrites we must use the `--mult-stores=yes` option.

Listing 12-31. Running pmemcheck with Listing 12-30.

```
$ valgrind --tool=pmemcheck --mult-stores=yes ./listing_12-30
==25609== pmemcheck-1.0, a simple persistent store checker
==25609== Copyright (c) 2014-2016, Intel Corporation
==25609== Using Valgrind-3.14.0 and LibVEX; rerun with -h for
copyright info
==25609== Command: ./listing_12-30
==25609==
==25609==
==25609== Number of stores not made persistent: 0
==25609==
==25609== Number of overwritten stores: 1
==25609== Overwritten stores before they were made persistent:
==25609== [0]    at 0x400962: main (listing_12-30.c:62)
==25609==       Address: 0x4023000      size: 4 state: DIRTY
==25609== ERROR SUMMARY: 1 errors
```

`pmemcheck` reports that we have overwritten stores. We can fix this problem by either inserting a flushing instruction between both writes, if we forgot to flush, or by moving one of the stores to volatile data if that store corresponds to short lived data.

At the time of publication, Persistence Inspector does not support checking for overwritten stores. As you have seen, Persistence Inspector does not consider a missing flush an issue unless there is a write dependency. In addition, it does not consider this a performance problem because writing to the same variable in a short timespan is likely to hit the CPU caches anyway, rendering the latency differences between DRAM and persistent memory irrelevant.

Unnecessary Flushes

Flushing should be done carefully. Detecting unnecessary flushes, such as redundant ones, can help improve code performance. The code in Listing 12-328 shows a redundant call to the `flush()` function on line 64.

Listing 12-32. Example of redundant flushing of a persistent memory variable.

```
33  #include <emmintrin.h>
34  #include <stdint.h>
35  #include <stdio.h>
```

```
#include <sys/mman.h>
#include <fcntl.h>
#include <valgrind/pmemcheck.h>

void flush(const void *addr, size_t len) {
    uintptr_t flush_align = 64, uptr;
    for (uptr = (uintptr_t)addr & ~(flush_align - 1);
            uptr < (uintptr_t)addr + len;
            uptr += flush_align)
        _mm_clflush((char *)uptr);
}

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_VALIDATE | MAP_SYNC,
            fd, 0);

    VALGRIND_PMC_REGISTER_PMEM_MAPPING(data,
                                       sizeof(int));

    *data = 1234;
    flush((void *)data, sizeof(int));
    flush((void *)data, sizeof(int)); // extra flush

    munmap(data, sizeof(int));
    VALGRIND_PMC_REMOVE_PMEM_MAPPING(data,
                                     sizeof(int));
    return 0;
}
```

We can use `pmemcheck` to detect redundant flushes using `--flush-check=yes` option, as shown in Listing 12-33.

Listing 12-33. Running pmemcheck with Listing 12-32.

```
$ valgrind --tool=pmemcheck --flush-check=yes ./listing_12-32
==104125== pmemcheck-1.0, a simple persistent store checker
==104125== Copyright (c) 2014-2016, Intel Corporation
==104125== Using Valgrind-3.14.0 and LibVEX; rerun with -h for
copyright info
==104125== Command: ./listing_12-32
==104125==
==104125==
==104125== Number of stores not made persistent: 0
==104125==
==104125== Number of unnecessary flushes: 1
==104125== [0]    at 0x400868: flush (emmintrin.h:1459)
==104125==    by 0x400989: main (listing_12-32.c:64)
==104125==      Address: 0x4023000     size: 64
==104125== ERROR SUMMARY: 1 errors
```

To showcase Persistence Inspector, Listing 12-34 has code with a write dependency, similar to what we did for Listing 12-11 in Listing 12-19. The extra flush occurs on line 65.

Listing 12-34. Example of writing to persistent memory with a write dependency. The code does an extra flush for the flag.

```
33  #include <emmintrin.h>
34  #include <stdint.h>
35  #include <stdio.h>
36  #include <sys/mman.h>
37  #include <fcntl.h>
38  #include <string.h>
39
40  void flush(const void *addr, size_t len) {
41      uintptr_t flush_align = 64, uptr;
42      for (uptr = (uintptr_t)addr & ~(flush_align - 1);
43              uptr < (uintptr_t)addr + len;
44              uptr += flush_align)
45          _mm_clflush((char *)uptr);
46  }
47
48  int main(int argc, char *argv[]) {
49      int fd, *ptr, *data, *flag;
50
51      fd = open("/mnt/pmem/file", O_CREAT|O_RDWR, 0666);
```

```
52      posix_fallocate(fd, 0, sizeof(int) * 2);
53
54      ptr = (int *) mmap(NULL, sizeof(int) * 2,
55              PROT_READ | PROT_WRITE,
56              MAP_SHARED_VALIDATE | MAP_SYNC,
57              fd, 0);
58      data = &(ptr[1]);
59      flag = &(ptr[0]);
60
61      *data = 1234;
62      flush((void *) data, sizeof(int));
63      *flag = 1;
64      flush((void *) flag, sizeof(int));
65      flush((void *) flag, sizeof(int)); // extra flush
66
67      munmap(ptr, 2 * sizeof(int));
68      return 0;
69  }
```

Listing 12-35 uses the same `reader` program from Listing 12-15 to show the analysis from Persistence Inspector. As before, we first collect data from the `writer` program, then the `reader` program, and finally run the report to identify any issues.

Listing 12-35. Running Intel Inspector - Persistence Inspector with Listing 12-34 (writer) and Listing 12-15 (reader).

```
$ pmeminsp cb -pmem-file /mnt/pmem/file -- ./listing_12-34
++ Analysis starts


++ Analysis completes
++ Data is stored in folder
"/data/.pmeminspdata/data/listing_12-34"


$ pmeminsp ca -pmem-file /mnt/pmem/file -- ./listing_12-15
++ Analysis starts

data = 1234

++ Analysis completes
```

```
++ Data is stored in folder
"/data/.pmeminspdata/data/listing_12-15"


$ pmeminsp rp -- ./listing_12-34 ./listing_12-15
#=============================================================
# Diagnostic # 1: Redundant cache flush
#-------------------
  Cache flush
    of size 64 at address 0x7F3220C55000 (offset 0x0 in
/mnt/pmem/file)
    in /data/listing_12-34!flush at listing_12-34.c:45 - 0x674
    in /data/listing_12-34!main at listing_12-34.c:64 - 0x73F
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
    in /data/listing_12-34!_start at
<unknown_file>:<unknown_line> - 0x574

  is redundant with regard to

   cache flush
    of size 64 at address 0x7F3220C55000 (offset 0x0 in
/mnt/pmem/file)
    in /data/listing_12-34!flush at listing_12-34.c:45 - 0x674
    in /data/listing_12-34!main at listing_12-34.c:65 - 0x750
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
    in /data/listing_12-34!_start at
<unknown_file>:<unknown_line> - 0x574

  of

  memory store
    of size 4 at address 0x7F3220C55000 (offset 0x0 in
/mnt/pmem/file)
    in /data/listing_12-34!main at listing_12-34.c:63 - 0x72D
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
    in /data/listing_12-34!_start at
<unknown_file>:<unknown_line> - 0x574
```

The Persistence Inspector report warns about the redundant cache flush within the `main()` function on line 65 of the `listing_12-34.c` program file - "`main at listing_12-34.c:65`". Solving these issues is as easy as deleting all the unnecessary flushes, and the result will improve the application's performance.

Out-of-Order Writes

When developing software for persistent memory, remember that even if a cache line is not explicitly flushed , that does not mean the data is still in the CPU caches. For example, the CPU could have evicted it due to cache pressure or other reasons. Further, the same way that writes not flushed properly may produce bugs in the event of an unexpected application crash, so do automatically evicted dirty cache lines if they violate some expected order of writes that the applications rely on.

To better understand this problem, explore how flushing works in the x86_64 and AMD64 architectures. From the user space, we can issue any of the following instructions to ensure our writes reach the persistent media:

- CLFLUSH
- CLFLUSHOPT (needs SFENCE)
- CLWB (needs SFENCE)
- Non-temporal stores (needs SFENCE)

The only instruction that ensures each flush is issued in order is `CLFUSH` because each `CLFLUSH` instruction always does an implicit fence instruction (`SFENCE`). The other instructions are asynchronous and can be issued in parallel and in any order. The CPU can only guarantee that all flushes issued since the previous `SFENCE` have completed when a new `SFENCE` instruction is explicitly executed. Think of `SFENCE` instructions as synchronization points (see Figure 12-6). For more information about these instructions, refer to the Intel® software developer manuals and the AMD software developer manuals.

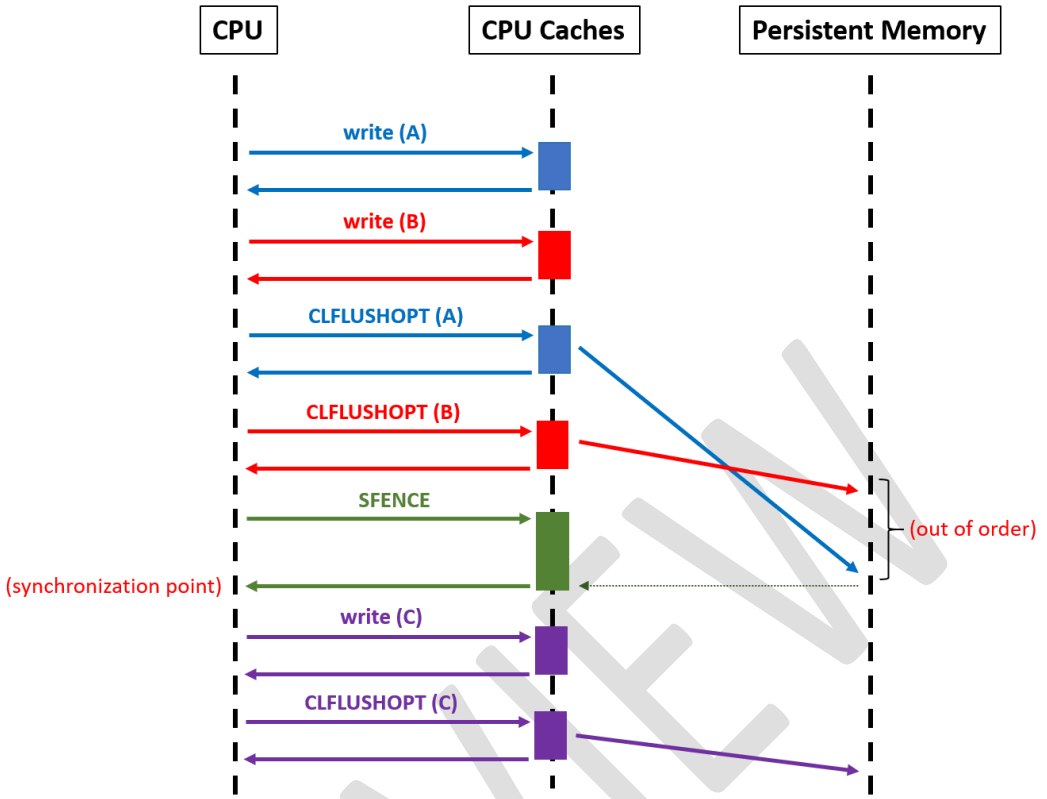

Figure 12-6. Example of how asynchronous flushing works. The SFENCE instruction ensures a synchronization point between the writes to A and B on one side, and to C on the other side.

As Figure 12-6 shows, we cannot guarantee the order with respect to how `A` and `B` would be finally written to persistent memory. This happens because stores and flushes to `A` and `B` are done between synchronization points. The case of `C` is different. Using the `SFENCE` instruction, we can be assured that `C` will always go after `A` and `B` have been flushed.

Knowing this, you can now imagine how out-of-order writes could be a problem in a program crash. If assumptions are made with respect to the order of writes between synchronization points, or if you forget to add synchronization points between writes/flushes where strict order is essential (think of a "valid flag" for a variable write, where the variable needs to be written before the flag is set to valid), you may encounter data consistency issues. Consider the pseudo-code in Listing 12-36.

Listing 12-36. Pseudo code showcasing an out-of-order issue.

```
1  writer () {
2          pcounter = 0;
```

```
 3          flush (pcounter);
 4          for (i=0; i<max; i++) {
 5                  pcounter++;
 6                  if (rand () % 2 == 0) {
 7                          pcells[i].data = data ();
 8                          flush (pcells[i].data);
 9                          pcells[i].valid = True;
10                  } else {
11                          pcells[i].valid = False;
12                  }
13                  flush (pcells[i].valid);
14          }
15          flush (pcounter);
16  }
17
18  reader () {
19          for (i=0; i<pcounter; i++) {
20                  if (pcells[i].valid == True) {
21                          print (pcells[i].data);
22                  }
23          }
24  }
```

For simplicity, assume that all flushes in Listing 12-36 are also synchronization points; that is, `flush()` uses `CLFLUSH`. The logic of the program is very simple. There are two persistent memory variables: `pcells` and `pcounter`. The first is an array of tuples `{data, valid}` where `data` holds the data and `valid` is a flag indicating if `data` is valid or not. The second variable is a counter indicating how many elements in the array have been written correctly to persistent memory. In this case, the `valid` flag is not the one indicating whether or not the array position was written correctly to persistent memory. In this case, the flag's meaning only indicates if the function `data()` was called; that is, whether or not `data` has meaningful data.

At first glance, the program appears correct. With every new iteration of the loop, the counter is incremented and then the array position is written and flushed. However, `pcounter` is incremented *before* we write to the array, thus creating a discrepancy between `pcounter` and the actual number of committed entries in the array. Although it is true that `pcounter` is not flushed until after the loop, the program is only correct after a crash if we assume that the changes to `pcounter` stay in the CPU caches (in that case, a program crash in the middle of the loop would simply leave the counter to zero).

As mentioned at the beginning of this section, we cannot make that assumption. A cache line can be evicted at any time. In the pseudo-code example in Listing 12-36, we could run into a bug where `pcounter` indicates the array is longer than it really is, making the `reader()` read uninitialized memory.

The code in Listings 12-37 and 12-38 provide a C++ implementation of the pseudo-code from Listing 12-36. Both use `libpmemobj-cpp` from the PMDK. Listing 12-37 is the writer program and 12-38 is the reader.

Listing 12-37. Example of writing to persistent memory with an out-of-order write bug.

```
33  #include <emmintrin.h>
34  #include <unistd.h>
35  #include <stdio.h>
36  #include <string.h>
37  #include <stdint.h>
38  #include <libpmemobj++/persistent_ptr.hpp>
39  #include <libpmemobj++/make_persistent.hpp>
40  #include <libpmemobj++/make_persistent_array.hpp>
41  #include <libpmemobj++/transaction.hpp>
42  #include <valgrind/pmemcheck.h>
43
44  using namespace std;
45  namespace pobj = pmem::obj;
46
47  struct header_t {
48      uint32_t counter;
49      uint8_t reserved[60];
50  };
51  struct record_t {
52      char name[63];
53      char valid;
54  };
55  struct root {
56      pobj::persistent_ptr<header_t> header;
57      pobj::persistent_ptr<record_t[]> records;
58  };
59
60  pobj::pool<root> pop;
61
62  int main(int argc, char *argv[]) {
63
```

```
64      // everything between BEGIN and END can be
65      // assigned a particular engine in pmreorder
66      VALGRIND_PMC_EMIT_LOG("PMREORDER_TAG.BEGIN");
67
68      pop = pobj::pool<root>::open("/mnt/pmem/file",
69                                   "RECORDS");
70      auto proot = pop.root();
71
72      // allocation of memory and initialization to zero
73      pobj::transaction::run(pop, [&] {
74          proot->header
75              = pobj::make_persistent<header_t>();
76          proot->header->counter = 0;
77          proot->records
78              = pobj::make_persistent<record_t[]>(10);
79          proot->records[0].valid = 0;
80      });
81
82      pobj::persistent_ptr<header_t> header
83          = proot->header;
84      pobj::persistent_ptr<record_t[]> records
85          = proot->records;
86
87      VALGRIND_PMC_EMIT_LOG("PMREORDER_TAG.END");
88
89      header->counter = 0;
90      for (uint8_t i = 0; i < 10; i++) {
91          header->counter++;
92          if (rand() % 2 == 0) {
93              snprintf(records[i].name, 63,
94                       "record #%u", i + 1);
95              pop.persist(records[i].name, 63); // flush
96              records[i].valid = 2;
97          } else
98              records[i].valid = 1;
99          pop.persist(&(records[i].valid), 1); // flush
100     }
101     pop.persist(&(header->counter), 4); // flush
102
103     pop.close();
104     return 0;
105 }
```

Listing 12-38. Reading the data structure written by Listing 12-37 to persistent memory.

```
#include <stdio.h>
#include <stdint.h>
#include <libpmemobj++/persistent_ptr.hpp>

using namespace std;
namespace pobj = pmem::obj;

struct header_t {
    uint32_t counter;
    uint8_t reserved[60];
};
struct record_t {
    char name[63];
    char valid;
};
struct root {
    pobj::persistent_ptr<header_t> header;
    pobj::persistent_ptr<record_t[]> records;
};

pobj::pool<root> pop;

int main(int argc, char *argv[]) {

    pop = pobj::pool<root>::open("/mnt/pmem/file",
                                 "RECORDS");
    auto proot = pop.root();
    pobj::persistent_ptr<header_t> header
        = proot->header;
    pobj::persistent_ptr<record_t[]> records
        = proot->records;

    for (uint8_t i = 0; i < header->counter; i++) {
        if (records[i].valid == 2) {
            printf("found valid record\n");
            printf("  name   = %s\n",
                           records[i].name);
        }
    }

    pop.close();
```

```
    74      return 0;
    75  }
```

Listing 12-37 (writer) uses the `VALGRIND_PMC_EMIT_LOG` macro to emit a pmreorder message when we get to lines 66 and 87. This will make sense later when we introduce out-of-order analysis using `pmemcheck`.

Now we will run Persistence Inspector first. To perform out-of-order analysis, we must use the `-check-out-of-order-store` option to the report phase. Listing 12-39 shows collecting the before and after data, and then running the report.

Listing 12-39: Running Intel Inspector - Persistence Inspector with Listing 12-37 (writer) and Listing 12-38 (reader).

```
$ pmempool create obj --size=100M --layout=RECORDS
/mnt/pmem/file

$ pmeminsp cb -pmem-file /mnt/pmem/file -- ./listing_12-37
++ Analysis starts


++ Analysis completes
++ Data is stored in folder
"/data/.pmeminspdata/data/listing_12-37"


$ pmeminsp ca -pmem-file /mnt/pmem/file -- ./listing_12-38
++ Analysis starts

found valid record
  name   = record #2
found valid record
  name   = record #7
found valid record
  name   = record #8

++ Analysis completes
++ Data is stored in folder
"/data/.pmeminspdata/data/listing_12-38"


$ pmeminsp rp -check-out-of-order-store -- ./listing_12-37
./listing_12-38
```

```
#=============================================================
# Diagnostic # 1: Out-of-order stores
#-------------------
  Memory store
    of size 4 at address 0x7FD7BEBC05D0 (offset 0x3C05D0 in
/mnt/pmem/file)
    in /data/listing_12-37!main at listing_12-37.cpp:91 -
0x1D0C
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
    in /data/listing_12-37!_start at
<unknown_file>:<unknown_line> - 0x1624

  is out of order with respect to

  memory store
    of size 1 at address 0x7FD7BEBC068F (offset 0x3C068F in
/mnt/pmem/file)
    in /data/listing_12-37!main at listing_12-37.cpp:98 -
0x1DAF
    in /lib64/libc.so.6!__libc_start_main at
<unknown_file>:<unknown_line> - 0x223D3
    in /data/listing_12-37!_start at
<unknown_file>:<unknown_line> - 0x1624
```

The Persistence Inspector report identifies an out-of-order store issue. The tool says that incrementing the counter in line 91 (`main at listing_12-37.cpp:91`) is out of order with respect to writing the valid flag inside a record in line 98 (`main at listing_12-37.cpp:98`).

To perform out-of-order analysis with `pmemcheck`, we must introduce a new tool called `pmreorder`. The `pmreorder` tool is included in PMDK from version 1.5 onwards. This standalone Python tool performs a consistency check of persistent programs using a store reordering mechanism. The `pmemcheck` tool cannot do this type of analysis, although it is still used to generate a detailed log of all the stores and flushes issued by an application that `pmreorder` can parse. For example, consider Listing 12-40.

Listing 12-40. Running pmemcheck to generate a detailed log of all the stores and flushes issued by Listing 12-37.

```
$ valgrind --tool=pmemcheck -q --log-stores=yes --log-stores-
stacktraces=yes
  --log-stores-stacktraces-depth=2 --print-summary=yes
  --log-file=store_log.log ./listing_12-37
```

The meaning of each parameter is as follows:

- `-q` silences unnecessary `pmemcheck` logs that `pmreorder` cannot parse.
- `--log-stores=yes` tells `pmemcheck` to log all stores.
- `--log-stores-stacktraces=yes` dumps stacktrace with each logged store. This helps locate issues in your source code.
- `--log-stores-stacktraces-depth=2` is the depth of logged stacktraces. Adjust according to the level of information you need.
- `--print-summary=yes` prints a summary on program exit. Why not?
- `--log-file=store_log.log` logs everything to `store_log.log`.

The `pmreorder` tool works with the concept of "engines." For example, the `ReorderFull` engine checks consistency for all the possible combinations of reorders of stores and flushes. This engine can be extremely slow for some programs, so you can use other engines such as `ReorderPartial` or `NoReorderDoCheck`. For more information, refer to the `pmreorder` page, which has links to the man pages (https://pmem.io/pmdk/pmreorder/).

Before we run `pmreorder`, we need a program that can walk the list of records contained within the memory pool and return `0` when the data structure is consistent, or `1` otherwise. This program is similar to the reader shown in Listing 12-41.

Listing 12-41. Checking the consistency of the data structure written in Listing 12-37.

```
33  #include <stdio.h>
34  #include <stdint.h>
35  #include <libpmemobj++/persistent_ptr.hpp>
36
37  using namespace std;
38  namespace pobj = pmem::obj;
39
```

```
40  struct header_t {
41      uint32_t counter;
42      uint8_t reserved[60];
43  };
44  struct record_t {
45      char name[63];
46      char valid;
47  };
48  struct root {
49      pobj::persistent_ptr<header_t> header;
50      pobj::persistent_ptr<record_t[]> records;
51  };
52
53  pobj::pool<root> pop;
54
55  int main(int argc, char *argv[]) {
56
57      pop = pobj::pool<root>::open("/mnt/pmem/file",
58                                   "RECORDS");
59      auto proot = pop.root();
60      pobj::persistent_ptr<header_t> header
61          = proot->header;
62      pobj::persistent_ptr<record_t[]> records
63          = proot->records;
64
65      for (uint8_t i = 0; i < header->counter; i++) {
66          if (records[i].valid < 1 or
67                                records[i].valid > 2)
68              return 1; // data struc. corrupted
69      }
70
71      pop.close();
72      return 0; // everything ok
73  }
```

The program in Listing 12-41 iterates over all the records that we expect should have been written correctly to persistent memory (lines 65-69). It checks the `valid` flag for each record, which should be either `1` or `2` for the record to be correct (line 66). If an issue is detected, the checker will return 1 indicating data corruption.

Listing 12-42 shows a three-step process for analyzing the program:

1. Create an object type persistent memory pool, known as a memory-mapped file, on `/mnt/pmem/file` of size 100MiB, and name the internal layout “RECORDS.”
2. Use the `pmemcheck` Valgrind tool to record data and call stacks while the program is running.
3. The `pmreorder` utility processes the `store.log` output file from `pmemcheck` using the `ReorderFull` engine to produce a final report.

Listing 12-42. First, a pool is created for Listing 12-37. Then, pmemcheck is run to get a detailed log of all the stores and flushes issued by Listing 12-37. Finally, pmreorder is run with engine ReorderFull.

```
$ pmempool create obj --size=100M --layout=RECORDS
/mnt/pmem/file


$ valgrind --tool=pmemcheck -q --log-stores=yes --log-stores-
stacktraces=yes --log-stores-stacktraces-depth=2 --print-
summary=yes --log-file=store.log ./listing_12-37


$ pmreorder -l store.log -o output_file.log -x
PMREORDER_TAG=NoReorderNoCheck -r ReorderFull -c prog -p
./listing_12-37
```

The meaning of each `pmreorder` option is as follows:

- `-l store_log.log` is the input file generated by `pmemcheck` with all the stores and flushes issued by the application.
- `-o output_file.log` is the output file with the out-of-order analysis results.
- `-x PMREORDER_TAG=NoReorderNoCheck` assigns the engine `NoReorderNoCheck` to the code enclosed by the tag `PMREORDER_TAG` (see lines 66-87 from listing 12-37). This is done to focus the analysis on the loop only (lines 89-105 from listing 12-37).
- `-r ReorderFull` sets the initial reorder engine. In our case, `ReorderFull`.
- `-c prog` is the consistency checker type. It can be `prog` (program) or `lib` (library).

- `-p ./checker` is the consistency checker.

Opening the generated file `output_file.log`, you should see entries similar to those in Listing 12-43 that highlight detected inconsistencies and problems within the code.

Listing 12-43. Content from 'output_file.log' generated by pmreorder showing a detected inconsistency during the out-of-order analysis.

```
WARNING:pmreorder:File /mnt/pmem/file inconsistent
WARNING:pmreorder:Call trace:
Store [0]:
    by  0x401D0C: main (listing_12-37.cpp:91)
```

The report states that the problem resides at line 91 of the listing_12-37.cpp writer program. To fix `listing_12-37.cpp`, move the counter incrementation after all the data in the record has been flushed all the way to persistent media. Listing 12-44 shows the corrected part of the code.

Listing 12-44. Fix Listing 12-37 by moving the incrementation of the counter to the end of the loop (line 95).

```
86      for (uint8_t i = 0; i < 10; i++) {
87          if (rand() % 2 == 0) {
88              snprintf(records[i].name, 63,
89                       "record #%u", i + 1);
90              pop.persist(records[i].name, 63);
91              records[i].valid = 2;
92          } else
93              records[i].valid = 1;
94          pop.persist(&(records[i].valid), 1);
95          header->counter++;
96      }
```

Summary

This chapter provided an introduction to each tool and describes how to use them. Catching issues early in the development cycle can save countless hours of debugging complex code later on. This chapter introduced three valuable tools—Persistence

Inspector, `pmemcheck`, and `pmreorder`—that persistent memory programmers will want to integrate into their development and testing cycles to detect issues. We demonstrated how useful these tools are at detecting many different types of common programming errors.

The Persistent Memory Development Kit (PMDK) uses the tools described here to ensure each release is fully validated before release. The tools are tightly integrated into the PMDK Continuous Integration (CI) development cycle so you can quickly catch and fix issues.