

Intel® Inspector Persistence Inspector

User's Guide

Revision 0.2

Table of Contents

1.	Introducing Persistence Inspector	4
2.	Using Persistence Inspector	7
2.1	Setting up Environment	9
2.2	Preparing Your Application	10
2.2.1	Understanding Your Application	10
2.2.2	Notifying Persistence Inspector of After-Unfortunate-Event Phase Start and Stop	10
2.2.3	Debug Build vs. Release Build	12
2.3	Analyzing Before-Unfortunate-Event Phase	12
2.4	Analyzing After-Unfortunate-Event Phase	13
2.5	Reporting Issues Detected	14
3.	Understanding a Persistence Analysis Report	14
3.1	Missing Cache Flush	14
3.2	Redundant or Unnecessary Cache Flushes	15
3.3	Missing Memory Fences	16
3.4	Redundant Memory Fences	17
3.5	Out-of-order Persistent Memory Stores (Preview Feature)	18
3.6	Update without Undo Logging	19
3.7	Undo Logging without Update	20
4.	Using the Intel® Inspector GUI to Open a Persistence Analysis Report	20
4.1	Generating Report	20
4.2	Opening Project	21
4.3	Configuring Search Directories	21
4.4	Investigating Problem Reports	22
5.	Legal Information	24

1. Introducing Persistence Inspector

Byte addressable non-volatile or persistent memory devices using new technologies, such as 3D XPoint™ technology by Intel Corporation and Micron Technology, Inc., can be directly attached to memory controllers. Such a device is often referred to as an NVDIMM. Data in NVDIMMs can survive system or program crashes. The access latencies of NVDIMMs are comparable to those of DRAMs. Programs read from/write to NVDIMMs using regular CPU load/store instructions. However, a store to persistent memory does not become persistent immediately because of caching. The data persists only after it is out of the cache hierarchy and is visible to the memory system. Also because of caching, the order of persistence may not be the same as the order of store. Consider the following code example 1:

Example 1: writing an address book to persistent memory

```
1      #include <stdio.h>
2      #include <fcntl.h>
3      #include <sys/file.h>
4      #include <sys/mman.h>
5      #include <string.h>
6
7      struct address {
8          char name[64];
9          char address[64];
10         int valid;
11     };
12
13     int main()
14     {
15         struct address *head = NULL;
16         int fd;
17
18         fd = open("addressbook.pmem", O_CREAT|O_RDWR, 0666);
19         posix_fallocate(fd, 0, sizeof(struct address));
20         head = (struct address *)mmap(NULL, sizeof(struct address),
21             PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
22         close(fd);
23
24         strcpy(head->name, "Clark Kent");
25         strcpy(head->address, "344 Clinton St, Metropolis, DC 95308");
26         head->valid = 1;
27
28         munmap(head, sizeof(struct address));
29
30         return 0;
31     }
```

In example 1, persistent memory is exposed as file “addressbook.pmem” and mapped into the process address space using regular file system APIs. After the persistent memory is mapped in, the program accesses the memory directly at line 24, 25 and 26. If there is power loss at line 27, one of the following scenarios can take place in the persistent memory:

- None of “head->name”, “head->address”, and “head->valid” has made its way to the memory system and become persistent;
- All of them have become persistent; or
- Any one or two of them, but not all, have become persistent.

In case that “head->valid” has made its way to the persistent memory but “head->name” and/or “head->address” has not, the following program example 2 will read invalid data.

Example 2: reading an address book from persistent memory

```
1      #include <stdio.h>
2      #include <fcntl.h>
3      #include <sys/file.h>
4      #include <sys/mman.h>
5
6      struct address {
7          char name[64];
8          char address[64];
9          int valid;
10     };
11
12     int main()
13     {
14         struct address *head = NULL;
15         int fd;
16
17         fd = open("addressbook.pmem", O_RDWR);
18         head = (struct address *)mmap(NULL, sizeof(struct address),
19             PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_NORESERVE, fd, 0);
20         close(fd);
21
22         if (head->valid == 1) {
23             printf("%s\n", head->name);
24             printf("%s\n", head->address);
25         }
26
27         munmap(head, sizeof(struct address));
28
29         return 0;
30     }
```

The caching effect presents a big challenge to persistent memory software development. To guarantee data is recoverable and/or consistent after a power failure or system crash, you need to reason where and when to explicitly flush data out of the cache hierarchy to the memory system.

Example 3: writing an address book to persistent memory - corrected

```
1      #include <stdio.h>
2      #include <fcntl.h>
3      #include <sys/file.h>
4      #include <sys/mman.h>
5      #include <string.h>
6      #include <emmintrin.h>
7
8      #define _mm_clflushopt(addr) \
9          asm volatile(".byte 0x66; clflush %0" : "+m" (*(volatile char *)addr));
10
11     struct address {
12         char name[64];
13         char address[64];
14         int valid;
15     };
16
17     int main()
18     {
19         struct address *head = NULL;
20         int fd;
21
22         fd = open("addressbook.pmem", O_CREAT|O_RDWR, 0666);
23         posix_fallocate(fd, 0, sizeof(struct address));
24         head = (struct address *)mmap(NULL, sizeof(struct address),
25             PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
26         close(fd);
27
28         strcpy(head->name, "Clark Kent");
29         strcpy(head->address, "344 Clinton St, Metropolis, DC 95308");
30         _mm_clflushopt(&(head->name));
31         _mm_clflushopt(&(head->address));
32         _mm_sfence();
33         head->valid = 1;
34         _mm_clflushopt(&(head->valid));
35         _mm_sfence();
36
37         munmap(head, sizeof(struct address));
38
39         return 0;
40     }
```

The code at line 30 to line 32 explicitly flushes “head->name” and “head->address” before “head->valid” is set at line 33. These 3 lines enforce the persistence order that “head->name” and “head->address” must persist before “head->valid” persists. The code at line 34 and 35 flushes “head->valid” to enforce “head->valid” is persistent before the persistent memory file is unmapped. It is worth noting that the `_mm_flushopt()` and `_mm_sfence()` intrinsic functions here can be replaced with `msync()` calls, but still, you need to figure out where to make the calls.

It is also worth noting that uncovering programming errors of cache flush misses is very hard, if not impossible, in normal development testing.

Persistence Inspector is a run-time tool for developers to detect such programming errors in persistent memory programs. In addition to cache flush misses, it also detects

- Redundant cache flushes
- Missing store fences
- Redundant store fences
- Out-of-order persistent memory stores (preview feature)
- Incorrect undo logging for the Persistent Memory Development Kit (PMDK), previously known as the Non-Volatile Memory Library (NVML). (<http://pmem.io>)

It is important to note that there is absolutely no guarantee the Persistence Inspector will find all issues in your application.

2. Using Persistence Inspector

Upon restart after an unfortunate event, such as a power failure or system crash, a persistent memory application must validate consistency and/or perform recovery of data stored in memory. Typically, a persistent memory application can be divided into 2 phases by an unfortunate event that can potentially causes data corruption or inconsistency: the phase that executes before the unfortunate event and the phase that executes after the unfortunate event. In the before-unfortunate-event phase, the application runs its normal flow, reading from and writing to the persistent memory. The after-unfortunate-event phase checks data consistency and recovers inconsistent data to consistent states before the application resumes normal operation.

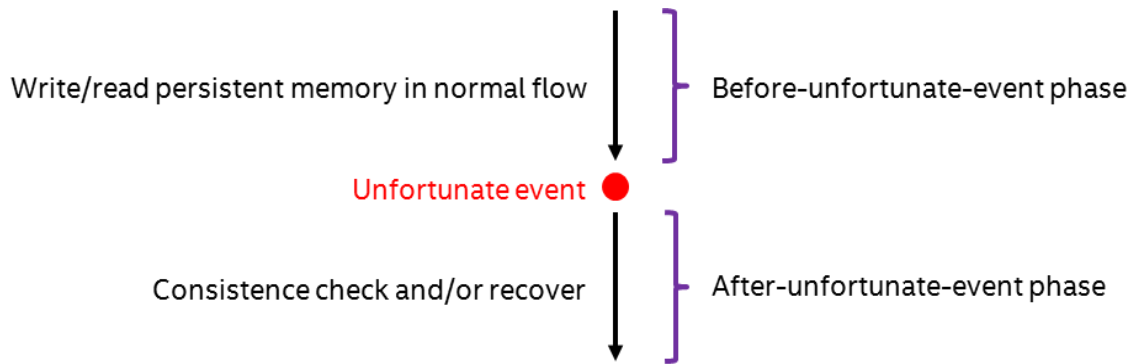


Figure 1: An unfortunate event divides a persistent memory application into 2 phases

The actual code of these two phases can reside in the same program or in different programs.

Accordingly, the Persistence Inspector analyzes your application in 2 phases: the before-unfortunate-event phase and the after-unfortunate-event phase. Fortunately, you don't need to force an unfortunate event to occur for the Persistence Inspector to do the analysis, because the Persistence Inspector discovers potential issues by assuming an unfortunate event can possibly occur at any point of application execution.

However, if your application uses PMDK transactions and you only care about undo log checking, you can skip the after-unfortunate-event phase. This is because the after-unfortunate-event phase is taken care of by PMDK transaction support.

Figure 2 shows tool workflow.

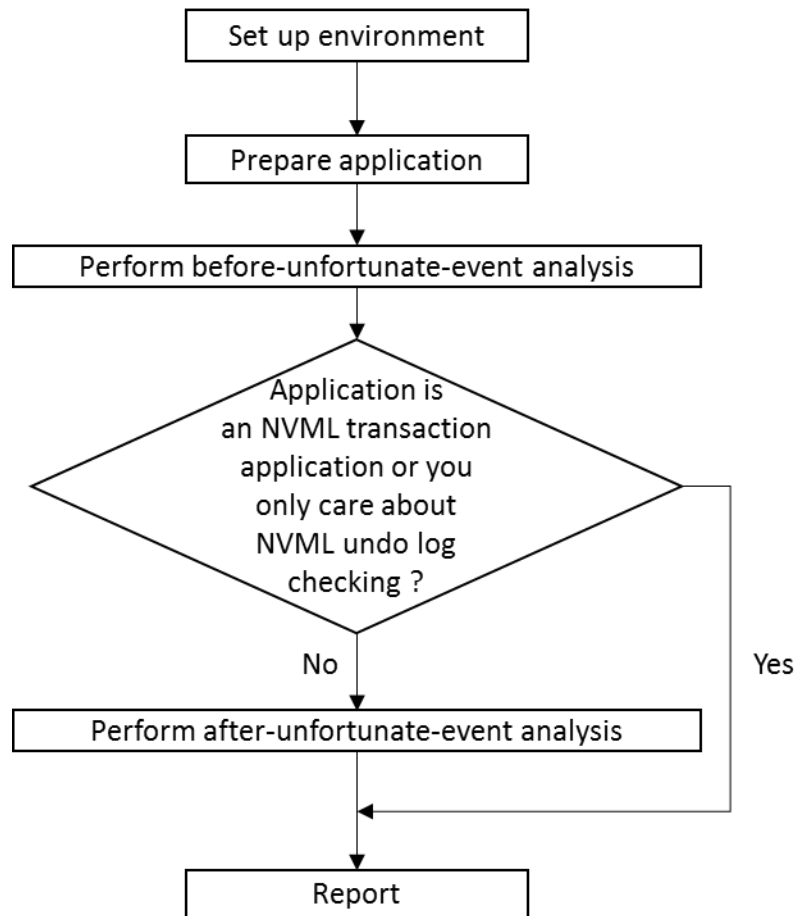


Figure 2: Persistence Inspector workflow

2.1 Setting up Environment

After the tool files are installed in a directory of choice, such as `/home/joe/pmeminsp`, add the Persistence Inspector path to the `PATH` and `LD_LIBRARY_PATH` environment variables. For example:

```
$ export PATH=/home/joe/pmeminsp/bin64:$PATH
$ export LD_LIBRARY_PATH=/home/joe/pmeminsp/lib64:$LD_LIBRARY_PATH
or
$ setenv PATH /home/joe/pmeminsp/bin64:$PATH
$ setenv LD_LIBRARY_PATH /home/joe/pmeminsp/lib64:$LD_LIBRARY_PATH
```

To verify the tool is installed and set up correctly, enter: "pmeminsp".

\$ pmeminp

Type 'pmeminsp help' for usage.

2.2 Preparing Your Application

2.2.1 Understanding Your Application

As stated earlier, a persistent memory application typically consists of 2 phases: a before-unfortunate-event phase and an after-unfortunate-event phase. To use the Persistence Inspector, you need to identify code associated with these 2 phases.

The before-unfortunate-event phase executes the code you want the tool to check and the after-unfortunate-event phase executes the code you want to run after a power failure or system crash.

If your application is non-transactional, you have probably written code to check data consistency and/or recover inconsistent data to consistent states before using the data after a restart (in the case of a power failure or a crash). In our aforementioned address book application, Example 1 is the code of the before-unfortunate-event phase and Example 2 is the code of the after-unfortunate-event phase.

If your application is constructed with PMDK transactions, because PMDK transaction runtime support is responsible for data consistency and recovery, the after-unfortunate-event code resides in the PMDK transaction runtime.

2.2.2 Notifying Persistence Inspector of After-Unfortunate-Event Phase Start and Stop

After the after-unfortunate-event phase is identified, it is highly recommended you notify the tool where the after-unfortunate-event phase starts and stops. If you skip this step, the Persistence Inspector will work very hard to find these itself, but you may experience more overhead and performance slowdown.

By default, the Persistence Inspector starts after-unfortunate-event phase analysis immediately after the first persistent memory file is mapped into the process address space, and stops immediately after the last persistent memory file is unmapped out of the process address space. This may not match exactly what the application does. The application may start the after-unfortunate-event phase much later after the first persistent memory file is mapped in, or may stop much earlier before the last persistent

memory file is unmapped out. In this case, telling the Persistence Inspector when to start and stop after-unfortunate-event phase analysis helps the tool run more efficiently.

Persistence Inspector provides a set of APIs for your application to notify the tool at run time of after-unfortunate-event phase analysis start and stop:

```
#define PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT 0x2
void __pmeminsp_start(unsigned int phase);
void __pmeminsp_pause(unsigned int phase);
void __pmeminsp_resume(unsigned int phase);
void __pmeminsp_stop(unsigned int phase);
```

To notify the tool of after-unfortunate-event phase start, all you need is a call to `__pmeminsp_start(PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT)` right before the after-unfortunate-event phase starts in your application. Similarly, to notify the tool of after-unfortunate-event phase stop, all you need is a call to `__pmeminsp_stop(PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT)` right after the after-unfortunate-event phase ends in your application.

The `__pmeminsp_pause(PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT)` and `__pmeminsp_resume(PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT)` calls give you finer control to pause and resume analysis after the analysis is started and before it is stopped.

For example, if the after-unfortunate-event phase is the duration of a function `recover()` call, you can simply place `__pmeminsp_start(PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT)` at the entry of this function and `__pmeminsp_stop(PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT)` at the exit of this function:

Example 4: notify the Persistence Inspector of after-unfortunate-event phase start and stop

```
1      #include "pmeminsp.h"
2      ... ..
3      ... ..
4      void recover(void)
5      {
6          __pmeminsp_start(PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT);
7          ... ..
8          __pmeminsp_stop(PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT);
9      }
```

```

10         void main()
11         {
12             .....
13             ... = mmap(.....);
14             __pmeminsp_stop((PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT);
15             .....
16             recover();
17             .....
18         }

```

Persistence Inspector APIs are defined in `libpmeminsp.so`, so please make sure correct options are specified when you build your applications. For example, “-I /home/joe/pmeminsp/include -L /home/joe/pmeminsp/lib64 -lpmeminsp”.

2.2.3 Debug Build vs. Release Build

Persistence Inspector can work with your release build or debug build binaries. However, for the Persistence Inspector to pinpoint issues to correct source locations, a debug build with symbols and without optimizations (`-g -O0`) is recommended.

2.3 Analyzing Before-Unfortunate-Event Phase

The available commands to run the before-unfortunate-event analysis phase are:

- `pmeminsp check-before-unfortunate-event [options] -- <cb-prog>`
- `pmeminsp cbue [option] -- <cb-prog>`
- `pmeminsp cb [option] -- <cb-prog>`

The `<cb-prog>`, or before-unfortunate-event program, is the target program executed before an unfortunate event. This is also the target program in which you expect the tool to uncover persistent memory programming issues.

This phase stores the intermediate result in a folder. We strongly recommend you use the smallest data set possible to minimize the file size and get the best tool performance.

Probably the most useful options are `-pmem-file` and `-pmem-file-base-path`. These two options notify the tool of persistent memory files to analyze:

`-pmem-file <persistent-memory-file>`

The `<persistent-memory-file>` is the pathname of a persistent memory file to be analyzed. This option can be repeated multiple times if the program `<cb-prog>` operates multiple persistent memory files.

If the persistent memory file name(s) is unknown before the program <cb-prog> starts, but the folder or directory that contains the persistent memory file(s) is known, option -pmem-file-base-path can be used:

`-pmem-file-base-path <persistent-memory-file-base>`

The <persistent-memory-file-base> is the pathname of a folder or directory containing the persistent memory files of interest. This option can be repeated multiple times if the program operates multiple persistent memory files in multiple folders or directories.

Please note that these two options are redundant if your application is PMDK-based. Persistence Inspector automatically traces all persistent memory files managed by the PMDK even if these options are absent.

2.4 Analyzing After-Unfortunate-Event Phase

The available commands to run the after-unfortunate-event analysis phase are:

- `pmeminsp check-after-unfortunate-event [options] -- <ca-prog>`
- `pmeminsp caue [option] -- <ca-prog>`
- `pmeminsp ca [option] -- <ca-prog>`

The <ca-prog>, or after-unfortunate-event program, is the program you run after an unfortunate event.

This phase also stores the intermediate result in a folder. We strongly recommend you use the smallest data set possible to minimize the file size and get the best tool performance.

Probably the most useful options are `-pmem-file` and `-pmem-file-base-path`. These two options notify the tool of persistent memory files to analyze:

`-pmem-file <persistent-memory-file>`

The <persistent-memory-file> is the pathname of a persistent memory file written by the before-unfortunate-event program. This option can be repeated multiple times if multiple persistent memory files were written by the before-unfortunate-event program.

If the persistent memory file name(s) is unknown, but the folder or directory that contains the persistent memory file(s) is known, option `-pmem-file-base-path` can be used:

`-pmem-file-base-path <persistent-memory-file-base>`

The <persistent-memory-file-base> is the pathname of a folder or directory containing the persistent memory files of interest. This option can be repeated multiple times if multiple persistent memory files in multiple folders or directories were written by the before-unfortunate-event program.

Please note that these two options are redundant if your application is PMDK-based. Persistence Inspector automatically traces all persistent memory files managed by the PMDK even if these options are absent.

2.5 Reporting Issues Detected

To generate a report of issues detected, simply run one of the following commands:

- `pmeminsp report [option] -- <prog>`
- `pmeminsp report [option] -- <cb-prog> <ca-prog>`
- `pmeminsp rp [option] -- <prog>`
- `pmeminsp rp [option] -- <cb-prog> <ca-prog>`

Here, <cb-prog>, or before-unfortunate-event program, is the program you ran in the before-unfortunate-event phase, and <ca-prog>, or after-unfortunate-event program, is the program you ran in the after-unfortunate-event phase. The tool will take <prog> as the same program executed in both phases if only one <prog> is specified in the command line.

The report is shown in plain text by default.

3. Understanding a Persistence Analysis Report

3.1 Missing Cache Flush

A missing cache flush of a persistent memory store (first store) is always with reference to a later persistent memory store (second store). Its potential adverse effect is that the second store gets persistent but the first store does not if an unfortunate event, such as a power loss, occurs after the second store.

```
strcpy(head->name, "Clark Kent");  
strcpy(head->address, "344 Clinton St, Metropolis, DC 95308");  
head->valid = 1;
```

In this code snippet, cache flushes of “head->name” and “head->address” are missing before “head->valid” is set to 1. If there is a power failure after “head->valid” is set to 1, it is possible that “head->valid” has the value of 1 but “head->name” and/or “head->address” contain invalid data after the system is restarted.

A report of a missing cache flush (example below) shows source locations of both the first store and second store along with call stacks. If information is available, the report also

shows the reason why the missing cache flush of the first store is a potential issue: the dependence of memory loads from the two memory locations.

The first memory store

```
in /home/joe/pmемinsp/addressbook/writeaddressbook!main at writeaddressbook.c:24 - 0x6ED
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43
in /home/joe/pmемinsp/addressbook/writeaddressbook!_start at <unknown_file>:<unknown_line> - 0x594
```

is not flushed before

the second memory store

```
in /home/joe/pmемinsp/addressbook/writeaddressbook!main at writeaddressbook.c:26 - 0x73F
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43
in /home/joe/pmемinsp/addressbook/writeaddressbook!_start at <unknown_file>:<unknown_line> - 0x594
```

while

memory load from the location of the first store

```
in /lib/x86_64-linux-gnu/libc.so.6!strlen at <unknown_file>:<unknown_line> - 0x889DA
```

depends on

memory load from the location of the second store

```
in /home/joe/pmемinsp/addressbook/readaddressbook!main at readaddressbook.c:22 - 0x6B0
```

3.2 Redundant or Unnecessary Cache Flushes

A redundant or unnecessary cache flush is a cache flush that can be removed in the analyzed execution path without affecting program correctness.

Although a redundant or unnecessary cache flush does not affect program correctness, it can potentially affect program performance.

Take the following code snippet as an example:

```
strcpy(head->name, "Clark Kent");
strcpy(head->address, "344 Clinton St, Metropolis, DC 95308");
_mm_clflushopt(&(head->name));
_mm_clflushopt(&(head->address));
_mm_sfence();
head->valid = 1;
_mm_clflushopt(&(head->valid));
_mm_sfence();
_mm_clflushopt(&(head->name));
```

```
_mm_clflushopt(&(head->address));  
_mm_sfence();
```

The repeated cache flushes at the end are redundant because “head->name” and “head->address” are already flushed.

A redundant or unnecessary cache flush is always reported with reference to a previous persistent memory store and, if information is available, a previous cache flush to the same persistent memory store.

Cache flush

```
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!test_redundant_flush at main.cpp:134 - 0x1721  
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:52 - 0x151F  
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:48 - 0x14FD  
in /home/joe/pmемinsp//tests/pmemdemo/src/pemmdemo!main at main.cpp:231 - 0x1C74  
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43  
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!_start at <unknown_file>:<unknown_line> - 0x12A4
```

is redundant with regard to cache flush

```
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!test_redundant_flush at main.cpp:135 - 0x1732  
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:52 - 0x151F  
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:48 - 0x14FD  
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:231 - 0x1C74  
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43  
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!_start at <unknown_file>:<unknown_line> - 0x12A4
```

of memory store

```
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!test_redundant_flush at main.cpp:133 - 0x170F  
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:52 - 0x151F  
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:48 - 0x14FD  
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:231 - 0x1C74  
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43  
in /home/joe/pmемinsp/tests/pmemdemo/src/pemmdemo!_start at <unknown_file>:<unknown_line> - 0x12A4
```

3.3 Missing Memory Fences

A missing memory store fence is always with reference to a persistent memory store (first store) that is flushed and a later persistent memory store (second store). Its potential adverse effect is that the second store may complete and become persistent before the first store is flushed in the case if the second store and the flush of the first store are reordered.


```

strcpy(head->name, "Clark Kent");
strcpy(head->address, "344 Clinton St, Metropolis, DC 95308");
_mm_clflushopt(&(head->name));
_mm_clflushopt(&(head->address));
head->valid = 1;

```

In this code snippet, a memory store fence is missing after “head->name” and “head->address” are flushed but before “head->valid” is set to 1. If there is a power failure after “head->valid” is set to 1, it is possible that “head->valid” has the value of 1 but “head->name” and/or “head->address” contain invalid data after the system is restarted.

A report of a missing memory fence (example below) shows source locations of both the first store and the second store along with call stacks. If information is available, the report also shows the reason why a missing memory fence before the second store is a potential issue: the dependence of memory loads from the two memory locations.

Memory store

```

of size 12 at address 0x7F2573F69044 (offset 0x44 in /home/joe/pmемinsp/tests/addressbook/addressbook.pmem)
in /home/joe/pmемinsp/tests/addressbook/writeaddressbook!main at writeaddressbook.c:79 - 0x74E
in /lib64/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x2042F
in /home/joe/pmемinsp/tests/addressbook/writeaddressbook!_start at <unknown_file>:<unknown_line> - 0x594

```

is flushed but the flush is not fenced before

memory store

```

of size 4 at address 0x7F2573F69080 (offset 0x80 in /home/joe/pmемinsp/tests/addressbook/addressbook.pmem)
in /home/joe/pmемinsp/tests/addressbook/writeaddressbook!main at writeaddressbook.c:83 - 0x7B8
in /lib64/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x2042F
in /home/joe/pmемinsp/tests/addressbook/writeaddressbook!_start at <unknown_file>:<unknown_line> - 0x594

```

while

memory load from the location of the first store

```

in /lib64/libc.so.6!__memcpy_sse2_unaligned_erms at <unknown_file>:<unknown_line> - 0x90A9C

```

depends on

memory load from the location of the second store

```

in /home/joe/pmемinsp/tests/addressbook/readaddressbook!main at readaddressbook.c:44 - 0x71B

```

3.4 Redundant Memory Fences

A redundant memory store fence is redundant for persistent ordering but may be required for other ordering purposes. It is your responsibility to determine if a redundant fence is required for other ordering purposes.

Although a redundant store fence does not affect program execution correctness, it can potentially affect performance.

```
strcpy(head->name, "Clark Kent");
strcpy(head->address, "344 Clinton St, Metropolis, DC 95308");
_mm_clflushopt(&(head->name));
_mm_clflushopt(&(head->address));
_mm_sfence();
_mm_sfence();
head->valid = 1;
```

In this code snippet, the second memory store fence is redundant.

A report of redundant memory fence (example below) shows source locations of the redundant memory store fence along with call stacks.

Memory fence is redundant for persistence ordering but may be necessary for other purposes

```
/home/joe/pmемinsp/tests/addressbook/writeaddressbook!pmem_sfence at writeaddressbook.c:61 - 0x6A2
in /home/joe/pmемinsp/tests/addressbook/writeaddressbook!main at writeaddressbook.c:86 - 0x7F2
in /lib64/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x2042F
in /home/joe/pmемinsp/tests/addressbook/writeaddressbook!_start at <unknown_file>:<unknown_line> - 0x594
```

3.5 Out-of-order Persistent Memory Stores (Preview Feature)

This is a preview feature. It may or may not be available in future releases.

Out-of-order persistent memory stores are two stores whose correct persistence order cannot be enforced.

In our address book example, “head->valid” should persist after “head->name” and “head->address” persist. Now consider the possible persistence orders of “head->valid” and “head->name” in the following code:

```
head->valid = 1;
strcpy(head->name, "Clark Kent");
```

If there is a power failure after “head->valid” is set to 1, “head->valid” can potentially get evicted from the cache hierarchy and become persistent before “head->name” does. It is also worth pointing out that explicit cache flushes will not enforce a correct order.

Out-of-order persistent memory stores are always reported with reference to two memory stores:

Memory store

```
/home/joe/pmемcheck/mytest/writename6!writename at writename6.c:13 - 0x6E0
in /home/joe/pmемcheck/mytest/writename6!main at writename6.c:21 - 0x72D
```

```
in /home/joe/pmemcheck/mytest/writename6!main at writename6.c:20 - 0x721
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43
in /home/joe/pmemcheck/mytest/writename6!_start at <unknown_file>:<unknown_line> - 0x594
```

should be after

memory store

```
in /home/joe/pmemcheck/mytest/writename6!writename at writename6.c:14 - 0x6EB
in /home/joe/pmemcheck/mytest/writename6!main at writename6.c:21 - 0x72D
in /home/joe/pmemcheck/mytest/writename6!main at writename6.c:20 - 0x721
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43
in /home/joe/pmemcheck/mytest/writename6!_start at <unknown_file>:<unknown_line> - 0x594
```

This feature is currently disabled by default. It is known that certain false positives may be reported.

3.6 Update without Undo Logging

It is your responsibility to undo-log a memory location before it is updated in a PMDK transaction. Usually, developers call PMDK function `pmemobj_tx_add_range()` or `pmemobj_tx_add_range_direct()` or use macro `TX_ADD()` to undo-log a memory location. If the memory location is not undo-logged before it is updated, the PMDK will fail to roll back changes to the memory location if the transaction is not successfully committed.

An issue of update without undo logging is reported with reference to a memory store and a transaction in which the memory is updated:

Memory store

```
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!test_tx_without_undo at main.cpp:190 - 0x1963
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:175 - 0x1877
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:163 - 0x180D
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:245 - 0x1CF4
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!_start at <unknown_file>:<unknown_line> - 0x12A4
```

not undo logged in transaction

```
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!test_tx_without_undo at main.cpp:185 - 0x1921
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:175 - 0x1877
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:163 - 0x180D
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:245 - 0x1CF4
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!_start at <unknown_file>:<unknown_line> - 0x12A4
```

3.7 Undo Logging without Update

It is your responsibility to undo-log a memory location before it is updated in a PMDK transaction. Usually, developers call PMDK function `pmemobj_tx_add_range()` or `pmemobj_tx_add_range_direct()` or use macro `TX_ADD()` to undo-log a memory location. If the memory location is undo-logged but never updated inside a PMDK transaction, performance may be degraded and/or the memory may be rolled back to a dirty/uncommitted/stale value if the transaction is not successfully committed.

An issue of undo logging without update is reported with reference to a PMDK undo logging call and a transaction in which the memory is undo-logged.

Memory region is undo logged

```
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!test_tx_without_update at main.cpp:190 - 0x1963
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:175 - 0x1877
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:163 - 0x180D
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:245 - 0x1CF4
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!_start at <unknown_file>:<unknown_line> - 0x12A4
```

but is not updated in transaction

```
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo! test_tx_without_update at main.cpp:185 - 0x1921
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:175 - 0x1877
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:163 - 0x180D
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:245 - 0x1CF4
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at <unknown_file>:<unknown_line> - 0x21F43
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!_start at <unknown_file>:<unknown_line> - 0x12A4
```

4. Using the Intel® Inspector GUI to Open a Persistence Analysis Report

4.1 Generating Report

Intel® Inspector - Persistence Inspector (`pmeminsp`) generates a text report on the console that can be read without additional tools. However, in order to get the benefits of Intel Inspector's graphical interface, the report should be generated using one of the following commands:

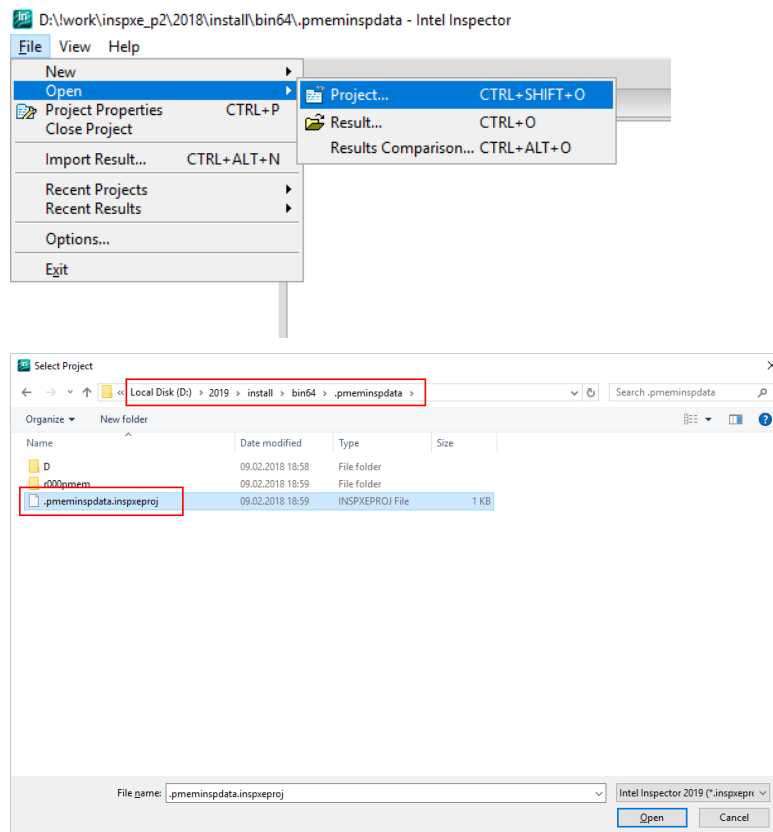
- `pmeminsp report -insp [option] -- <prog>`
- `pmeminsp report -insp [option] -- <cb-prog> <ca-prog>`

The “-insp” command line option guides the tool to generate a project folder in the default location (<current_folder>/.pmeminspdata) with a new result inside. If the project folder and appropriate file inside already exist, they will not be overwritten. In this case, the new result will be added to that project.

Additionally, the “-result <path>” option can be used to specify another location for the report.

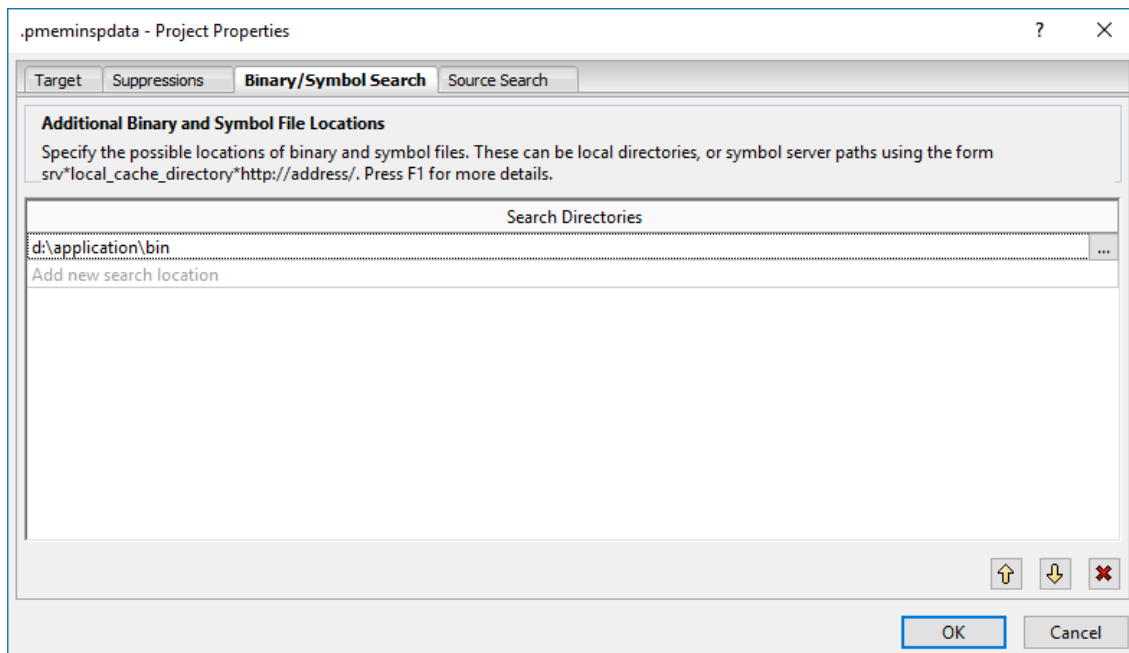
4.2 Opening Project

Start the Intel Inspector GUI from the command line (inspxe-gui) or the install shortcut. Then use the “File->Open->Project...” command to open the project file.



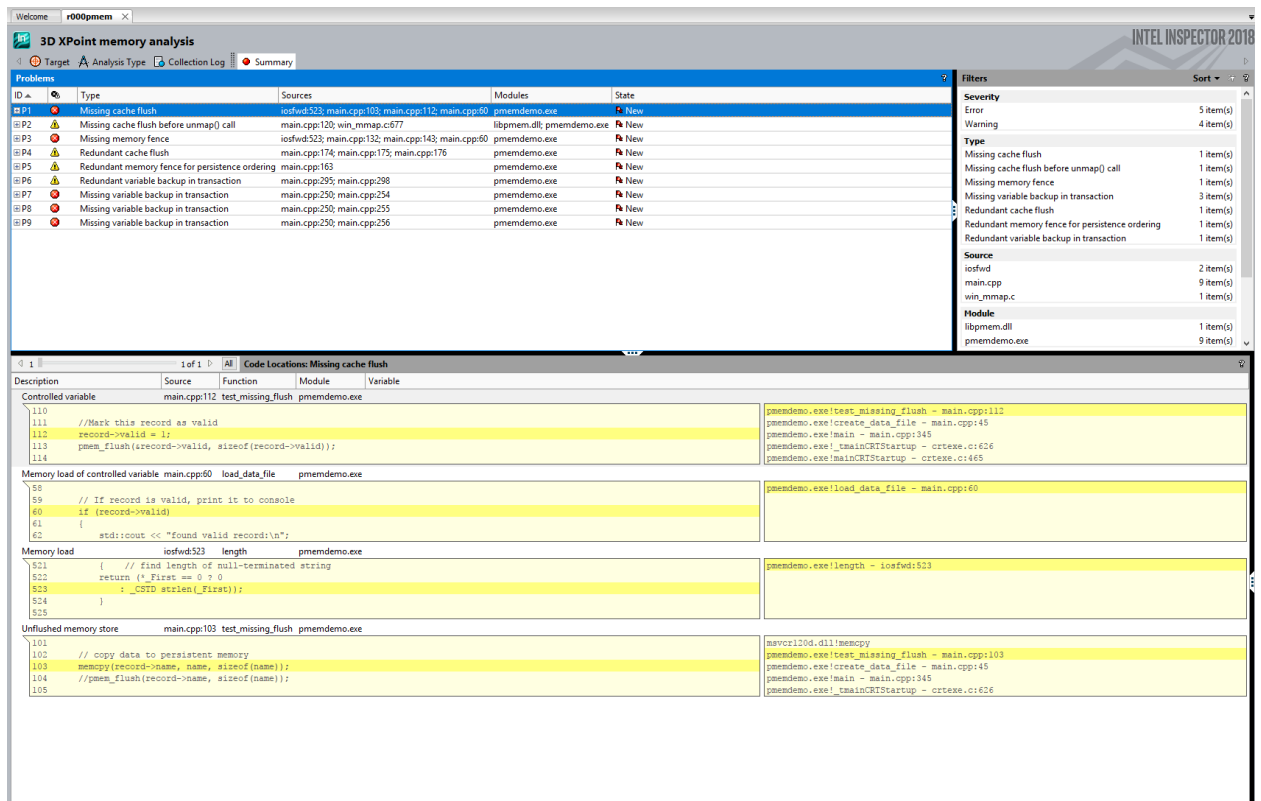
4.3 Configuring Search Directories

In most cases, you do not need to configure these folders. However, if the application was copied from another location or another system, you will need to configure additional search directories for symbol and/or source files resolution. To open this window, go to the “File -> Project Properties” menu item.



4.4 Investigating Problem Reports

The Intel Inspector GUI has two main views: “Summary” and “Sources”. When you open a result, you will see the problems summary.



“Code Locations” view shows the top 5 call frames of important locations in the application that caused this problem or warning report. To see more source lines or complete call stacks, double-click the specific location to open the “Sources” tab.

5. Legal Information

Intel, the Intel logo, and 3D XPoint are trademarks of Intel Corporation in the U.S. and/or other countries.

* Other names and brands may be claimed as the property of others.

Copyright 2017-2018 Intel Corporation.

This software and the related documents are Intel copyrighted materials, and your use of them is governed by the express license under which they were provided to you (License). Unless the License provides otherwise, you may not use, modify, copy, publish, distribute, disclose or transmit this software or the related documents without Intel's prior written permission.

This software and the related documents are provided as is, with no express or implied warranties, other than those that are expressly stated in the License.