Evaluating Crash Consistency for PM Software using Intel Pin

Satya Prakash

Follow this and additional works at: https://lib.dr.iastate.edu/creativecomponents

Part of the Data Storage Systems Commons

Recommended Citation
https://lib.dr.iastate.edu/creativecomponents/608

This Creative Component is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Creative Components by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Evaluating Crash Consistency for PM Software using Intel Pin

by

Satya Prakash

A Creative Component submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Dr. Mai Zheng, Major Professor

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this report. The Graduate College will ensure this report is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University
Ames, Iowa
2020

Copyright © Satya Prakash, 2020. All rights reserved.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>CHAPTER 1. OVERVIEW</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Programming PM is hard</td>
<td>4</td>
</tr>
<tr>
<td>1.1.2 Intel Pin Tool</td>
<td>6</td>
</tr>
<tr>
<td>1.1.3 Virtual to Physical address</td>
<td>7</td>
</tr>
<tr>
<td>1.1.4 Workload Types</td>
<td>9</td>
</tr>
<tr>
<td>CHAPTER 2. RELATED WORK</td>
<td>11</td>
</tr>
<tr>
<td>CHAPTER 3. IMPLEMENTATION</td>
<td>12</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>12</td>
</tr>
<tr>
<td>3.1.1 PM Emulation</td>
<td>13</td>
</tr>
<tr>
<td>3.1.2 Intel pin tool installation</td>
<td>13</td>
</tr>
<tr>
<td>3.1.3 Log Writes</td>
<td>14</td>
</tr>
<tr>
<td>3.1.4 Replay Writes</td>
<td>15</td>
</tr>
<tr>
<td>3.1.5 Mnemosyne Installation</td>
<td>16</td>
</tr>
<tr>
<td>3.1.6 NVML Installation</td>
<td>17</td>
</tr>
<tr>
<td>CHAPTER 4. RESULTS</td>
<td>19</td>
</tr>
<tr>
<td>4.1 Real-world bugs</td>
<td>19</td>
</tr>
<tr>
<td>4.2 Synthetic Bugs</td>
<td>23</td>
</tr>
<tr>
<td>CHAPTER 5. SUMMARY AND FUTURE WORK</td>
<td>27</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>28</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Figure 1.1</td>
<td>Instruction Sets Type for Persistent Memory</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>PM Interface using load/store</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Programming PM Example-1</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>Programming PM Example-2</td>
</tr>
<tr>
<td>Figure 1.5</td>
<td>Programming PM Example-3</td>
</tr>
<tr>
<td>Figure 1.6</td>
<td>Programming PM Example-4</td>
</tr>
<tr>
<td>Figure 1.7</td>
<td>Virtual to Physical address-1</td>
</tr>
<tr>
<td>Figure 1.8</td>
<td>Virtual to Physical address-2</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Log and Replay writes</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Record writes</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Replay Logs1</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Replay Logs2</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Commands to Run Tool</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Btree Real World Patch</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Btree Real World Bugs</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Btree Double Add Real World Patch</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Btree Double Add Real World Bugs</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Rbtree Real World Patch</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Rbtree Real World Bugs</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Rbtree Synthetic Patch</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>Rbtree Synthetic Bugs</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>Ctree Synthetic Patch_1</td>
</tr>
<tr>
<td>Figure 4.10</td>
<td>Ctree Synthetic Bugs_1</td>
</tr>
<tr>
<td>Figure 4.11</td>
<td>Ctree Synthetic Patch_2</td>
</tr>
<tr>
<td>Figure 4.12</td>
<td>Ctree Synthetic Bugs_2</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this creative component. First and foremost, Dr. Mai Zheng for his guidance, patience and support throughout this work and the writing of this creative component. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my colleague Om Gatla and Prakhar Bansal for their efforts and contributions to this work.
Ongoing advancement in non-volatile memory such as NVDIMMs have prompted to huge improvement in the area of persistent memory. It is faster, byte addressable and can persist without power. It allows direct manipulation of data in memory unlike other memory system like hard disk and SSD. It furthers overcomes the limitation of file system overhead that incurs extra burden to application to handle crash during failure. A Persistent program needs to correctly implement certain crash consistency mechanisms such as undo and redo logging. The program should be able to recover to consistent state after failure. Due to volatile caching and reordering of writes within memory hierarchy, programs need to carefully manage the order in which writes become persistent when implementing crash consistent software.

Persistent-memory applications ensure the consistency of persistent data by inserting ordering points between writes to PM allowing the construction of higher-level transaction mechanisms. PM System have introduced new instructions such as CLWB and SFENCE from x86 and DC CVAP from ARM to ensure ordering and further introduced high level transactional libraries to ensure persistence.

Required by the crash consistency guarantee, that is a program returns to a consistent state and resumes the execution after a failure, a testing tool is expected to detect inconsistencies during the entire procedure of execution, recovery, and resumption. Therefore, we proposed new method that will the log all the I/Os using intel pin tool and replay the all the I/Os and check the consistency of program by comparing the initial image and a final image. We are checking the consistency of program post failure by emulating the failure by removing some of I/O’s writes while replaying to check if program can recover itself after crash.
CHAPTER 1. OVERVIEW

1.1 Introduction

Persistent memory (PMEM), occupying the memory bus, is a solid-state high-performance byte-addressable memory device. By the virtue of its location, PMEM is granted access to memory buses that enables it to match its capabilities with that of DRAM in terms of speed and latency and also equals the non-volatility level of NAND flash. As the saying goes, there are two sides to every coin; one also has to overcome major challenges to successfully carry out the programming in PM without any glitch. Firstly the limited durability of data in PM and then in the event of power failure, the inability to retain the caches in the processor results in an inconsistent state on recovery if data modifications are to be made. Finally, write-back processor caches can re-order updates to PM, implying that even ordered updates may reach PM out of order.

In order to make data durable enough, consistent updates to PM is ensured by writing application to feed data to PM and waiting till it becomes durable. Precisely ‘clflushopt’ and ‘clwb’ instructions, incorporated in x86-64 architecture specification as shown in figure 1, are used to clean out or write-back a specific line from the cache hierarchy to memory. The subsequent fence instruction assists the completion of the outstanding flushes and write-backs by freezing the thread for the time being. The data in line A can be shielded from any crash if completion of data in cache is achieved by ‘sfence’ by simply using the instruction sequence ‘clwb and sfence’. The Non-temporal instructions (NTIs) present in the applications help to directly enter into to PM by bypassing the unnecessary cache and add to that the durability and ordering is further ensured by letting the write-combining buffers (WCB) to drain completely using the sfence instruction. These operations are used by the programmers to ensure the desired durability and order updates for consistency by directly moving data to PM as per the need(1).
The task of enforcing consistent updates to PM can be delegated to libraries like Mnemosyne, NV-Heaps, and NVML. Libraries provide useful functionality such as memory allocation, memory leak avoidance, type safety, durable transactions and atomic updates. These libraries provide a transaction interface, shown in Figure 1.1, that provides atomic and durable updates of multiple objects. Figure 1.1 illustrates how libraries provide an atomic update operation that persists a value atomically. This interface frees programmers from the burden of manual data movement. However, the general-purpose nature of these libraries can preclude low-level software optimization and results in conservative ordering constraints. For example, atomic transactions may not be needed for some data structures, such as an append-mostly log or copy-on-write trees.
Legacy applications written for filesystems can gain the performance benefits of PM by using PM-aware file systems such as PMFS, BPFS, NOVA and DAX-based filesystems on Linux such as ext2, ext4, XFS-DAX. These filesystems bypass the operating system block layer and directly update data and metadata in PM. Hence, they can provide stronger reliability and consistency guarantees than traditional file systems by persisting data synchronously, rather than asynchronously as done for hard drives and SSDs. So far we discussed the advantages of persistent memory but implementing the persistent memory from programmer point of view is tough. Next section explains why programming PM is hard.
1.1.1 Programming PM is hard

Here one particular example is explained in depth and is examined in the light to show that PM programming is hard to implement.

Figure 1.3 Programming PM Example-1

In this example, an array data structure is provided to us and is asked to update the second item B to G. So to begin with, B is firstly updated in crash consistent manner. This can be achieved by following undo logging mechanism. It is a two-step process; to begin with, the backup of existing data is made. Once the backup is built up, in-place update is performed to the second item B to G and finally updated is committed to save the array successfully. To guarantee cross consistency, write to each step is needed to persist first before proceeding to any further steps and all updates need to be persisted at the end.
In figure 1.3 shows the cross consistent mechanism with three steps backup, update and commit. First program makes backup of data and set valid bit to 1, write backup and valid bit and then update array and unset valid bit and writeback all updates.
It seems correct but what if there is failure before persistence barrier and there is reordering therefore it is possible that backup is not persisted, but bit is valid. Thus, we can conclude programming for crash consistency is very hard. first programmer must implement cross consistency mechanism and then correct implementation by using persist barrier whenever needed.

1.1.2 Intel Pin Tool

Intel Pin (2) provides a platform for building instrumentation tools. A pin tool consists of instrumentation, analysis, and callback routines. Instrumentation routines inspect the application’s instructions and insert calls to analysis routines. Analysis routines are called when the program executes an instrumented instruction and often perform ancillary tasks. The program invokes callbacks when an event occurs, for example, when it is about to exit. we have developed a simple pin tool that prints the memory addresses of all data and data a program writes. Instruction is an instrumentation routine that Pin calls the first time the program executes an instruction, so the routine can specify how it should be instrumented. If the instruction writes memory, this example pintool inserts a call to Address—an analysis routine—and directs Pin to pass it the memory reference’s effective address. Immediately before a memory reference executes, the program calls
Address, which prints the address to a file. The program invokes a callback routine, Fini, when it exits. Instrumentation and callback routines are registered in the pintool’s main function.

Pin uses a just-in-time (JIT) compiler to insert instrumentation into a running application. The JIT compiler recompiles and instruments small chunks of binary instructions immediately prior to executing them. Pin stores the modified instructions in a software code cache where they execute in lieu of the original application instructions. The code cache allows Pin to generate code regions once and reuse them for the remainder of program execution, amortizing compilation costs. Pin’s average base overhead is 30 percent, and user-inserted instrumentation adds to the time.

1.1.3 Virtual to Physical address

There are two approaches we can take to get the physical address(3):

1. Add a syscall to the kernel that, given a virtual address, will return the physical address. However, modifying the kernel breaks the rule of doing everything from user space so we must rule this out.

2. Use the pagemap file for a process to get the frame a page is mapped to and then use that to seek into /dev/pmem and replay the write command there.

Using this approach, it is entirely possible to translate a virtual address to a physical address in user space.

/proc/[pid]/pagemap

The pagemap provides user space access to how the kernel is managing the pages for a process. It is a binary file so extracting information from it is a little bit tricky. there are 64 bits worth of information for every page. We are interested in bits 0-54, the page frame number. In order to get the page frame number for a given page from the pagemap, we need to determine the offset into the pagemap to seek to. This can be done as such: Given an address, we divide it by the page size and then multiply by 8. There are 64 bits, or 8 bytes, of info for each page. Then we seek to that position in the file and read the first 7 bytes. We are interested in bits 0-54. That is a total of 55 bits. So, we read the first 7 bytes (56 bits) and clear bit 55. But 55 is the soft-dirty flag which
Now that we have the page frame number, we can easily calculate the physical address of our buffer as such as shown in figure 1.6 & 1.7.

Figure 1.7  Virtual to Physical address-1

\[
\text{physical addr} = (\text{page frame number} \gg \text{PAGE SHIFT}) + \text{distance from page boundary of buffer}
\]

where \text{PAGE_SHIFT} is a kernel define. For my x86_64 system, it was defined as 12.
1.1.4 Workload Types

This sections explains the different types of libraries which can be used to test our tool.

- Mnemosyne

Mnemosyne(4) provides a simple interface for programming with persistent memory. Programmers declare global persistent data with the keyword persistent or allocate it dynamically. Mnemosyne provides primitives for directly modifying persistent variables and supports consistent updates through a lightweight transaction mechanism. Compared to past work on disk-based persistent memory, Mnemosyne is much lighter weight, as it can store data items as small as a word rather than a virtual memory page.

- PMDK

The Persistent Memory Development Kit (PMDK)(5) is a collection of libraries and tools for System Administrators and Application Developers to simplify managing and accessing persistent memory devices. Tuned and validated on both Linux and Windows, the libraries build on the Direct...
Access (DAX) feature which allows applications to directly access persistent memory as memory-files.

- **PMFS**

PMFS (6) is a file system for persistent memory. The file system is optimized to be lightweight and efficient in providing access to persistent memory that is directly accessible via CPU load/store instructions. It manages the persistent memory directly and avoids the block driver layer and page cache layer and thus provides synchronous reads and writes to persistent area. It supports all the existing POSIX style file system APIs so that the applications need not be modified to use this file system. In addition, PMFS provides support for huge pages to minimize TLB entry usage and speed up virtual address lookup. PMFS’s mmap interface can map a file’s data directly into the process’s address space without any intermediate buffering. This file system has been validated using DRAM to emulate persistent memory. Hence, PMFS also provides an option to load the file system from a disk-based file into memory during mount and save the file system from memory into the disk-based file during unmount. PMFS also guarantees consistent and durable updates to the file system meta-data against arbitrary system and power failures. PMFS uses journaling (undo log) to provide consistent updates to meta-data.
CHAPTER 2. RELATED WORK

Papers and reports pertaining to evaluation of crash consistency have been on the rise since persistent memory is gaining pace. The paper ‘PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs’ (7) proposes a crash consistency testing framework that is both flexible and fast. PMTest provides flexibility by providing two basic assertion-like software checkers to test two fundamental characteristics of all CCS: the ordering and durability guarantee. These checkers can also serve as the building blocks of other application-specific, high-level checkers. PMTest enables fast testing by deducing the persist order without exhausting all possible orders. PMTest tools require a lot of manual efforts to cover all the cases thus might consume lots of time. It cannot be used as a generic tool, need to be linked application with PMTest code. It does not consider the bugs after recovery.

‘Cross-Failure Bug Detection in Persistent Memory Programs (8)’ talks about a tool that detects cross-failure bugs by considering failures injected at all ordering points in pre-failure execution and checking for cross-failure races and cross-failure semantic bugs in the post-failure continuation. Consistency depends critically on the order of persistent memory access in both pre-failure and post-failure execution. Because hardware may reorder persistent memory accesses both before and after failure, validation of crash-consistent programs requires holistic analysis of both execution stages. XFDetector has detected four new bugs in three pieces of PM software: one of PMDK examples, a PM-optimized Redis database and a PMDK library function. XFDetector is slower as compared to PM test. It needs to run code too many times based on ROI.
CHAPTER 3. IMPLEMENTATION

The generation of test image is initiated by running Mnemosyne and NVML application by capturing the I/O commands using the Intel pin tool, so generated commands, are parallely saved in a file. These commands are further used to generate the test image by simply replaying the write commands and data using physical address calculated by virtual address.

Figure 3.1 Log and Replay writes

3.1 Introduction

This section primarily focuses on setup which helped in implementing the entire experiment. The experiment is conducted on a machine having Intel Core i7 3.00GHz CPU, 8GB main memory, and 4 GB emulated persistent memory. The operating system is Linux distribution based, Ubuntu 16.04 LTS with kernel v4.4, Compiler GCC/G++-4.8.4. As soon as the application is run, the actual data and the data size (bytes) are recorded in data.out file by the memory tracer while taking into consideration the actual offset with respect to physical address into log.out file. All
the write commands are recorded by log.out file and the corrupted image is further reproduced by simply trimming off the write command based on range of ID given.

3.1.1 PM Emulation

To permanently allocate space for persistent memory in a Ubuntu system, first we need to add a kernel boot parameter.

Edit /etc/default/grub with sudo privilege.

Change line:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
```

To

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash memmap=4G!4G"
```

4GB will be sufficient for our purpose. Save the changes and execute:

```
sudo update –grub
```

Then reboot the system to make the parameter take effect. After the restart, you shall see a new PMEM device with directory /dev/pmem0m or /dev/pmem0 with command:

```
$ df -Th
```

Create mounting point for PMEM device:

First format the raw partition /dev/pmem0 we got from previous steps:

```
$ mkfs. ext4 -F /dev/pmem0
```

Then create a mounting point with the name you wish, here we will name it as pmem:

```
$ mkdir /mnt/pmem
```

Finally, mount the device to the mounting point with DAX option:

```
$ mount -o dax /dev/pmem0 /mnt/pmem
```

Now you can use this persistent memory device like a normal file folder

3.1.2 Intel pin tool installation

Download the pin program tarball
$ wget http://software.intel.com/sites/landingpage/pintool/downloads/pin-xxxx-gcc.4.4.7-linux.tar.gz

Untar it

$ tar -xzf pin-xxxx-gcc.4.4.7-linux.tar.gz

Managed platforms support

$ setenv INTEL JIT PROFILER64 The Pin kit full path /intel64/bin/libpinjitprofiling.so

Building the Example Tools

$ cd source/tools/ManualExamples

$ make all TARGET=intel64

3.1.3 Log Writes

```c
static VOID RecordMem(VOID * lp, CHAR r, VOID * addr, INT32 size, BOOL isPrefetch)
{
    struct io_header_t io_header;
    intptr_t paddr=0;
    static UINT64 lcount = 0;
    vlr_t to_phys_user(apaddr, (uintptr_t)addr);
    if(paddr > 0x100000000 && paddr < 0x17fffffff)
    {
        lcount++;
        //
        io_header.count = lcount;
        io_header.paddr = paddr - 0x100000000;
        io_header.size = size;
        fwrite(&io_header,sizeof(struct io_header_t),1,trace);
        Tracefile <<lcount << dec << setw(2) << size;
        if (!isPrefetch)
            EmitMem(addr, size);
        Tracefile << endl;
    }
}
```

Figure 3.2  Record writes
3.1.4 Replay Writes

```c
if(optind < argc){
    printf("non-option ARGV-elements: ");
    while (optind < argc)
        printf ("%s ", argv[optind++]);
    putchar ("n");
}

printf("PTE: Replay: disk_file = %s\n", disk_file);
printf("PTE: Replay: io_header_file = %s\n", headerlog_file);
printf("PTE: Replay: io_data_file = %s\n", datalog_file);
int io_header_fd = open(headerlog_file, O_RDONLY);
    if(io_header_fd < 0){
        printf("PFE: ERROR in opening %s \n", headerlog_file);
        return -1;
    }

int io_data_fd = open(datalog_file, O_RDONLY);
    if(io_data_fd < 0){
        printf("PFE: ERROR in opening %s \n", datalog_file);
        return -1;
    }

int disk_fd = open(disk_file, O_RDWR | O_SYNC);
    if(disk_fd < 0){
        printf("PFE: ERROR in opening %s \n", disk_file);
        return -1;
    }

uint32_t offset = 0;
for(int i=0;i< total_io_count;i++)
{
    uint64_t cur_offset = sizeof(struct io_header_t)*i;
    ret = pread(io_header_fd, &io_header, sizeof(struct io_header_t), cur_offset);
    if(ret < 0){
        printf("PFE: ERROR at pread %s\n", header_t, i);
        return -1;
    }
}
```

Figure 3.3  Replay Logs1
3.1.5 Mnemosyne Installation

Dependencies:

- SCons: A software construction tool
- GCC 6.2.1 or above
- GLIBC 2.19 or above
- Libconfig

```c
uint32_t offset = 0;
for(int i=0; i< Total_io_count;i++)
{
    uint64_t cur_offset = sizeof(struct io_header_t)*i;
    ret = pread(io_header_fd, &io_header, sizeof(struct io_header_t), cur_offset);
    if(ret < 0)
    {
        printf("PFE: ERROR at pread #"PRIu64" header from io_header log !!\n", i);
        return -1;
    }
    unsigned char *data_buf = malloc(io_header.size,1);
    if(data_buf == NULL){
        printf("PFE: ERROR at calloc !!\n");
        return -1;
    }
    ret = pread(io_data_fd, data_buf, io_header.size, offset);
    if(ret < 0){
        printf("error is reading data");
        return -1;
    }
    ret = pwrite(disk_fd, data_buf, io_header.size, io_header.paddr);
    if(ret < 0){
        printf("error in writing data");
        return -1;
    }
    offset+= io_header.size;
    free(data_buf);
}
```

- gelf

```bash
$ apt-get install libconfig-dev libconfig9
```

- libevent (For memcached

```bash
$ apt-get install libevent-dev
```

Figure 3.4   Replay Logs2
ALPS persistent memory allocator

ALPS Dependencies (on Ubuntu):

cmake
libattr1-dev
libboost-all-dev
libevent-dev
libnuma1
libnuma-dev
libyaml-cpp-dev

$ cd usermode/library/pmalloc/include/alps

$ mkdir build

$ cd build

$ cmake .. -DTARGET ARCH MEM=CC-NUMA -DCMAKE BUILD TYPE=Release

$ make

3.1.6  NVML Installation

To build this library, we need to install the following required packages on the build system:

autoconf

pkg-config

To build the latest development version, just clone this tree and build the master branch:

$ git clone https://github.com/pmem/nvml

$ cd nvml

$ make

If we want to compile, and hopefully run the built-in tests, with a different compiler, we have to provide the CC and CXX variables. For example:

$ make CC=clang CXX=clang++
Run below commands to run the tool.

**Collecting Traces**

```
root@localhost:~ /home/satyap/Desktop# ./collect -d ./input -i ./input -o trace.out
```

**Replaying Traces**

```
root@localhost:~ /home/satyap/Desktop# ./replay -d data.out -f logs.out -h /mnt/pm/sm/test
```

**Figure 3.5 Commands to Run Tool**
CHAPTER 4. RESULTS

This section contains the Experiment and the results obtained by running the PMTests since we used PMTest as reference for conducting our experiment.

4.1 Real-world bugs

pmdk_btree.patch reproduces the bug found in btree_map.c.

Bugs present is extra tx add: Add persistent data that will not be modified in a transaction to the log.

```
diff --git a/nvml/src/examples/libmemobj/tree_map/btree_map.c b/nvml/src/examples/libmemobj/tree_map/btree_map.c
index b488fed..f6d2061 100644
--- a/nvml/src/examples/libmemobj/tree_map/btree_map.c
+++ b/nvml/src/examples/libmemobj/tree_map/btree_map.c
@@ -184,7 +184,7 @@ btree_map_create_split_node(TOID(struct tree_map_node) node,
                        
                        // a bug
                      +  // TX_ADD(node); // a bug
```

Figure 4.1 Btree Real World Patch

```
ASSIGN ERROR: btree_map.c:191: Address range [0x7f51c3421e80, 0x7f51c3421f80] is not TransactionAdded before modified.
ASSIGN ERROR: btree_map.c:191: Address range [0x7f51c3421e80, 0x7f51c342200] is not TransactionAdded before modified.
ASSIGN ERROR: btree_map.c:202: Address range [0x7f51c3422270, 0x7f51c3422280] is not TransactionAdded before modified.
ASSIGN ERROR: btree_map.c:199: Address range [0x7f51c342200, 0x7f51c342210] is not TransactionAdded before modified.
ASSIGN ERROR: btree_map.c:202: Address range [0x7f51c342218, 0x7f51c342220] is not TransactionAdded before modified.
ASSIGN ERROR: btree_map.c:199: Address range [0x7f51c342200, 0x7f51c342210] is not TransactionAdded before modified.
ASSIGN ERROR: btree_map.c:202: Address range [0x7f51c342200, 0x7f51c342210] is not TransactionAdded before modified.
ASSIGN ERROR: btree_map.c:204: Address range [0x7f51c3421e80, 0x7f51c3421f80] is not TransactionAdded before modified.
ASSIGN ERROR: btree_map.c:191: Address range [0x7f51c3421e80, 0x7f51c3421f80] is not TransactionAdded before modified.
ASSIGN ERROR: btree_map.c:191: Address range [0x7f51c3421e80, 0x7f51c342200] is not TransactionAdded before modified.
TRANSACTION WARNING: tx.c:1724: Address range [0x7f51c3022000, 0x7f51c30220e0] overlaps with previously TransactionAdded addresses.
```

Figure 4.2 Btree Real World Bugs
pmdk_btree_doubleadd.patch reproduces the double TX_ADD performance issue found in btree_map.c.
Figure 4.4 Btree Double Add Real World Bugs

\texttt{pmdk_rbtree.patch} reproduces the bug addressed is missing undo log entry in rbtree example
Figure 4.5  Rbtree Real World Patch
4.2 Synthetic Bugs

`rbtree_backup_4.patch` contains bugs associated with missing or misplaced backup of persistent objects.

```c
diff --git a/nvml/src/examples/libpmemobj/tree_map/rbtree_map.c b/nvml/src/examples/libpmemobj/tree_map/rbtree_map.c
index 0f08935..eb6f002 100644
--- a/nvml/src/examples/libpmemobj/tree_map/rbtree_map.c
+++ b/nvml/src/examples/libpmemobj/tree_map/rbtree_map.c
@@ -230,7 +230,7 @@ rbtree_map_insert_bst(TOID(struct rbtree_map) map, TOID(struct tree_map_node) n)

    TX_SET(n, parent, parent);

-    pmemobj_tx_add_range_direct(dst, sizeof(*dst));
+    // pmemobj_tx_add_range_direct(dst, sizeof(*dst));
      PM_EQUIV(dst, n);
    }
```

Figure 4.7 Rbtree Synthetic Patch
Figure 4.8  Rbtree Synthetic Bugs

.ctree_backup_1.patch contains bugs related to Correct reorder of backup.

diff --git a/nvml/src/examples/libmemobj/tree_map/ctree_map.c b/nvml/src/examples/libmemobj/tree_map/ctree_map.c
index 140cc97..c687f67 100755
--- a/nvml/src/examples/libmemobj/tree_map/ctree_map.c
+++ b/nvml/src/examples/libmemobj/tree_map/ctree_map.c
@@ -301,8 +301,9 @@ ctree_map_remove(PMEMobjpool *pop, TOID(struct ctree_map) map, uint64_t key)
     TX_BEGIN(pop) {
         pmemobj_tx_add_range_direct(leaf, sizeof(*leaf));
         PM_EQU(leaf->key, 0);
-        PM_EQU(leaf->slot, OID_NULL);
+        PM_EQU(leaf->slot, OID_NULL);
         // PM_EQU(leaf->slot, OID_NULL);
     } TX_END
+    PM_EQU(leaf->slot, OID_NULL);
     } else {
         /*
             * In this situation:
Figure 4.9  Ctree Synthetic Patch_1
**ctree_backup_2.patch** contains bugs associated with missing or misplaced backup of persistent objects.

```diff
diff --git a/nvml/src/examples/libpmemobj/tree_map/ctree_map.c b/nvml/src/examples/libpmemobj/tree_map/ctree_map.c
index 140cc97..8e0e897 100755
--- a/nvml/src/examples/libpmemobj/tree_map/ctree_map.c
+++ b/nvml/src/examples/libpmemobj/tree_map/ctree_map.c
@@ -316,7 +316,7 @@ ctree_map_remove(PMEMobj_pool *pop, TOID(struct ctree_map) map, uint64_t key)
 struct ctree_map_entry *dest = parent;
     TOID(struct ctree_map_node) node;
     TOID_ASSIGN(node, parent->slot);
-    pmemobj_tx_add_range_direct(dest, sizeof(*dest));
+    // pmemobj_tx_add_range_direct(dest, sizeof(*dest));
     PM_EQU(*dest, D_RU(node)->entries[0].key == leaf->key);

```

**Figure 4.11  Ctree Synthetic Patch_2**
Figure 4.12  Ctree Synthetic Bugs_2
CHAPTER 5. SUMMARY AND FUTURE WORK

In-depth analysis of the behavior of various memory applications like pmdk, Mnemosyne, especially under faulty condition, has been done to proficiently expedite the process of finding bugs. Intel pin tool has been successfully demonstrated as a great asset to emulate the buggy application. To overcome the glitch, a general logging library has been created that can facilitate in locating the bugs in crash consistent application in both, pre-recovery and post-recovery state. Numerous tests and trials have been left for the future due to time constraints. Future work relates to deeper analysis of different mechanisms to further evaluate the effectiveness of this tool. There are also many ideas that could be exploited to try to obtain the most effective way for increasing the reliability and further authenticate the usage of this tool by verifying the correctness for wider array of problems. The behavior of PM software needs to be investigated in detail in the event of any crash and improve its efficacy. To pin point the location of bug in the application, an intensive study is prerequisite to track partial write with respect to PM software. The tool needs some exposure to different workloads to further enhance its scope for different challenges
BIBLIOGRAPHY


