持久内存快速编程手册

Intel PRC Optane CoE dennis.wu@intel.com

Introduction

英特尔®傲腾™持久内存以创新的内存技术重新定义了传统存储架构，将高性价比的大容量内存与数据持久性巧妙地结合在一起，以合理的价格提供大型持久内存层级。凭借在内存密集型工作负载，虚拟机密度和快速存储容量方面的突破性性能水平，英特尔®傲腾™持久内存 (Intel Optane®™PMem) 可加速 IT 转型，以支持数据时代对算力的需求。全新的 PMem 200 系列与第三代英特尔®至强®可扩展处理器(icelake, ICX)搭配，入门级 PMem 系列与第二代英特尔®至强®可扩展处理器(cascalake, CXL)搭配，与针对数据库，数据分析和虚拟化等基础设施等工作负载打造的软件生态系统保持兼容，有助于更加有效地挖掘数据的潜在价值。开发人员可以利用行业标准的持久内存编程模式，构建更简单，更强大的应用，确保对数据中心的投资能够适应未来的需求。(了解更多关于英特尔®傲腾™持久内存的信息，请访问: https://www.intel.cn/content/www/cn/zh/products/memory-storage/optane-dc-persistent-memory.html)。

持久内存产品可通过不同的方法使用，有些用法对应用来说是透明的。例如，所有持久内存产品都支持存储接口和标准文件 API，就像固态盘 (Solid State Disk, SSD) 一样，访问 SSD 上的数据非常简单且易于理解，因此不在我们的讨论之列。或者将持久内存配置成内存模式，系统的持久内存的使用方式和系统内存一样，应用不需要做任何更改，所以也不在我们的讨论之列。我们将重点介绍持久内存式访问，即应用管理驻留在持久内存中的可字节寻址的数据结构。英特尔编写了一本持久内存编程的书如图 1 所示，中文版也会在 2021 年发表。这个快速编程手册可以作为该书的一个补充，让开发人员快速了解持久内存相关的编程方法和重要概念。该手册不会涉及持久内存配置及使用的方方面面，而只是通过一些示例说明持久内存编程的一些重要概念。我们介绍的部分用例具有易失性，仅将持久内存用来扩展内存容量，但主要介绍持久性用例，即持久内存中的数据结构不会受到系统崩溃和电源故障的影响，在这些事件发生期间仍然确保数据结构的一致性。我们同时也会列举一些其它存储的示例，主要目的可以给大家一些直观的比较和感受。

图 1 持久内存编程英文版

每项新技术的兴起总会引发新的思考，持久内存也不例外。构建与开发解决方案时，请考虑持久内存的以下特征：

- 持久内存的性能（吞吐量、延迟和带宽）远高于 NAND，但是可能低于 DRAM。
- 不同于 NAND，持久内存很耐用。其耐用性通常比 NAND 高出多个数量级，可以超过服务器的生命周期。
- 持久内存模块的容量远大于 DRAM 模块，并且可以共享相同的内存通道。支持持久内存的应用可原地更新数据，无需对数据进行序列化/反序列化处理。
- 持久内存支持字节寻址（类似于内存）。应用可以只更新所需的数据，不会产生任何读取-修改-写入（read-modify-write，RMW）开销。
- 数据与 CPU 高速缓存保持一致。持久内存可提供直接内存访问 (direct memory access, DMA) 和远程直接内存访问 (remote direct memory access, RDMA) 操作。
- 写入持久内存的数据不会在断电后丢失。
- 权限检查完成后，可以直接从用户空间访问持久内存上的数据。数据访问不经过任何内核代码、文件系统页面缓存(page cache)或中断。
- 持久内存上的数据可立即使用，也就是说：
    - 系统通电后即可使用数据。
    - 应用不需要花时间来预热高速缓存。
    - 它们可在内存映射后立即访问数据。
- 持久内存上的数据不占用 DRAM 空间，除非应用将数据复制到 DRAM，以便更快地访问数据。
- 写入持久内存模块的数据位于系统本地。应用负责在不同系统之间复制数据。

应用开发人员通常会考虑内存驻留（memory-resident）数据结构和存储驻留（storage-resident）数据结构。就数据中心应用而言，开发人员要谨慎地在存储中保持一致的数据结构，即使系统崩溃时也不例外。这个问题通常可以使用日志技巧（如预写日志）来解决，先将更改写入日志，然后再将其刷新到持久存储中。如果数据修改过程中断，应用可以借助日志中的信息，在重启时完成恢复操作。这样的技巧已存在多年；但正确的实施方法开发难度很大，维护起来也很耗时。开发人员通常要依赖于数据库、编程库和现代文件系统的组合来提供一致性。即便如

此，最终还是要应用开发人员设计一种策略，在运行时和从应用和系统崩溃中恢复系统时确保存储中数据结构的一致性。

英特尔开发的 PMDK 经过多年的发展，涵盖了大量开源库（如图 2 所示）可帮助应用开发人员和系统管理员简化持久内存设备的管理和访问。该套件的开发工作与持久内存的操作系统支持是同步的，因此可确保库充分利用通过操作系统接口提供的所有特性。接下来的章节里，主要通过示例一一介绍 PMDK 相关库在持久内存编程方面的使用。

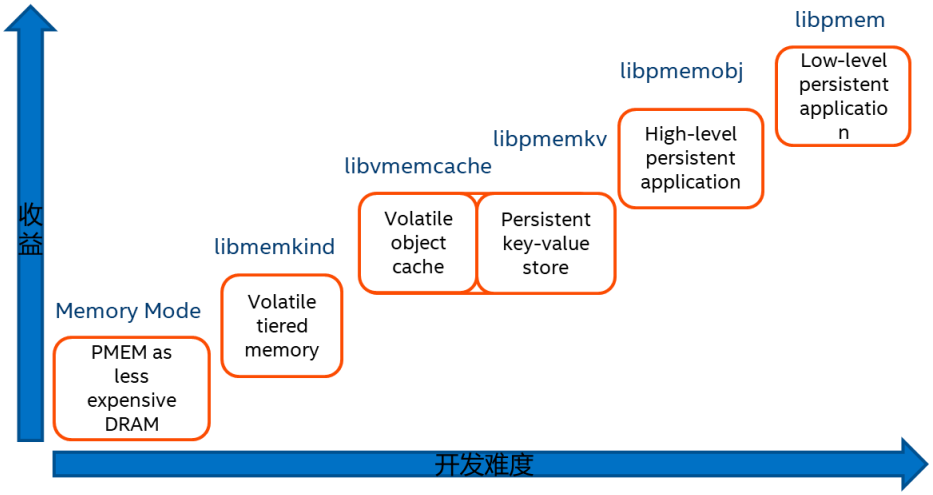


图 2 PMDK 相关的开发库

一个简单的声明：在这个文档中间的所有示例可能存在有一些 bug，所以这些参考示例不建议作为您产品级代码的一部分。这些例子在测试中的数据，也只是给您一个参考，是在一定的条件和环境下面获得的数据，如果你不能复现这些数据（差别很大），可以联系我们。

Environment Setup

在环境设置中，我们在此并不特别限定某种环境，只是列出我们的示例运行的环境。

CPU	Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz
DRAM	16G*12=192GB
PMEM	128GB*12=1.5TB (App Direct Mode)
SSD	P4510
PMDK	1.10 (include libpmem/libpmemobj/libpmemblk)
PMEMKV	1.4
memkind	1.10.1

libpmem

libpmem 是一个非常底层的 C 语言库，负责处理与持久内存相关的 CPU 指令、以最佳方式将数据复制到持久内存以及文件映射。有关 libpmem 的更多信息，请访问 http://pmem.io/pmdk/libpmem/。

我们的示例使用的 C 代码在 Linux 上构建与运行，使用 libpmem 把持久内存作为 SSD 存储的持久缓存页，需要考虑这样的特性：

1. 一个 4T 的 SSD 设备可以分成 1024*1024 个 block，每个 block 里面有 1024 个 4KB 大小的页, 这些页面要保存在持久内存中并持久化。
2. 要考虑每一个页的事务性，即在系统奔溃或者突然断电的情况下不能出现脏页。
3. 所有的持久内存内的数据可以恢复。
4. 在这个例子中我们只考虑单线程的情况。

所有的页面需要保存在持久内存中间，需要考虑到以下几点：

1. 数据的对齐，数据写到持久内存最好要 64 字节对齐，所以要考虑数据的布局。
2. block 中的 4K 页不能使用原地更新，因为一旦发生了断电，更新可能被中断从而产生脏的数据。
3. block 中的 4K 页可以通过分配器去进行管理，但是由于我们只有一个固定大小的数据需要管理，可以简单的通过 meta data 来管理。
4. 数据的布局关系到数据的恢复，不能保存绝对的指针而必须是一个 offset，以便在重启或崩溃后恢复数据。

示例 1（360 行）中使用的 libpmem 核心接口主要包括：

1. pmem_base = (char *)pmem_map_file(filename, PMEM_SIZE, PMEM_FILE_CREATE, 0666, &mapped_len, &is_pmem))
2. pmem_memset_persist(pmem_base,0x00,PMEM_SIZE);
3. pmem_memset_persist(pmem_meta,0x0,8); 原子操作
4. pmem_memcpy_persist(PAGE_FROM_META(page_new), content, 4096); 非原子操作
5. pmem_memcpy_persist(page_new,&atomic_value,8); 原子操作
6. pmem_persist(page_old,8); 原子操作

在持久内存的编程里面，个人认为以下这两点是持久内存编程的基础，所有的上层的库和应用都是基于这两个设定来保证数据的完整性和一致性。

1. *pmem_memcpy_persist 和 pmem_memcpy_nodrain 有什么区别怎么去理解？pmem_memcpy_persist 可以保证数据已经写到了持久内存，pmem_memcpy_nodrain 不能保证。像 non-temporal write 是一个乱序的写入，我们不能清楚哪些数据已经落到了持久*

内存，哪些还没有，而 fence（drain）可以保证所有写的数据都落到持久内存中。ADR 和 eADR 是对这个部分的实现有影响，如果是 ADR，在数据持久化的过程中有以下 2 个步骤 step1：数据写入 CPU Cache （这个地方没有谈及 NTW,NTW 不经过 cache） step2： 数据通过 clflushopt/clflush/clwb 等指令刷新到持久内存中，fense 在 clflush 和 clwb 是必须的。而 eADR，由于整个 cpu cache 中的数据在突然断电的情况下会保存到持久内存，所以没有必要经过上述 step2，只需要将数据写到 CPU cache 就可以了。

2. *每个支持持久内存的平台都有一套具备原子性的原生内存操作。在英特尔硬件上，原子持久存储为 8 字节。因此，如果正在传输与持久内存对齐的 8 字节存储过程中程序或系统发生崩溃，在恢复时，这 8 字节包含的要么全是老内容，要么全是新内容。英特尔处理器拥有存储超过 8 字节的指令，但这些指令不具有故障原子性，所以发生电源故障等事件时可能会遭到破坏，变成一个脏的新旧数据的混合。*

示例 1：libpmem 实现持久页缓存

```
#include <iostream>
#include <chrono>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <libpmem.h>
#include <time.h>
#include <string.h>

#define BLOCK_PAGE_NUM 1024
#define REQ_BLOCK 10
#define REQ_PAGE 0x3ff
#define DISK_BLOCK_NUM 1048576 //1024*1024 blocks
#define MAGIC_NUMBER 0xabcdabcd
/**************************************
 * \
 * pmem layout as:
 ---8byte magic number---|--- 8byte page_meta[0]---|---8byte page_meta[1]---|...|---
4k page[0]---|---4k page[1]---|
Page meta 和真正的 4k 页有对应关系。
rebuild the block dram and freelist structure after the system restart.
 */
//8 bytes atomic,必须一起更新。不能出现一个域更新了，而另外的域还没有更新的情况
typedef struct page_meta {
  uint64_t valid:1; //valid 表示该 page 是否已经分配和使用，如果是 0，表示该 page 没有被使用
  uint64_t sn:31;  //sn 来表示页的新旧，sn 越大表示页越新，old page 一般 sn 会被重置为 0
  uint64_t req_id:32;  //req_id, which is in the range from 1~1024*1024*1024
}page_meta_t;
//magic number，可以不定义这个数据结构，因为位置是固定的
typedef struct pmem_layout {
  uint64_t magic_number;
  uint64_t page_offset;    //real page offset，保证这个 offset 4K 对齐。
}pmem_layout_t;
char * pmem_base; //持久内存的基地址
uint64_t page_address;  //真正的也得首地址。

#define PAGE_SIZE 4096
```

```
#define PMEM_META_SIZE sizeof(page_meta_t)
#define MAGIC_NUM_SIZE sizeof(pmem_layout_t)
#define PMEM_SIZE 100*1024*1024*1024UL  //定义我们使用持久内存的大小 100GB
#define PMEM_PAGE_NUMBERS ((PMEM_SIZE-MAGIC_NUM_SIZE)/(PAGE_SIZE+PMEM_META_SIZE)-1)
//根据持久内存的大小，我们可以知道我们有多少个内存页
#define PAGE_ID_FROM_META(meta) (((uint64_t)meta-MAGIC_NUM_SIZE-
(uint64_t)pmem_base)/PMEM_META_SIZE+1) //根据持久内存的 page meta 而知道 page_id
#define PAGE_FROM_META(meta) (char *)(page_address + PAGE_SIZE * (((uint64_t)meta -
MAGIC_NUM_SIZE- (uint64_t)pmem_base)/PMEM_META_SIZE)) //meta->real page
#define PAGE_META_FROM_ID(id) (page_meta_t *)
((uint64_t)pmem_base+MAGIC_NUM_SIZE+(id-1)*PMEM_META_SIZE) //从 pageid 知道 page meta

/***************************************
 * *
 * disk structure,这个数据结构放在内存中，在初始化的时候扫描 page meta 数据，恢复 disk 内存数据结
构
 * [[page_id1,page_id2]* BLOCK_PAGE_NUM],cache_pages_cnt]* DISK_BLOCK_NUM
 */
typedef struct block_page {
  uint64_t page_id1:32;
  uint64_t page_id2:32;
}block_page_t;
typedef struct block_data {
  block_page_t pages[BLOCK_PAGE_NUM];
  uint32_t cached_pages_cnt;  //if the count over one threshold, might need to write
back to the SSD.
}block_data_t;
typedef struct disk {
  block_data_t blocks[DISK_BLOCK_NUM];
}disk_t;
disk_t * disk;

//a free list that always pick the free page from the free_list[free_num-1];
//初始化的时候扫描 page meta 数据，恢复 free pages 的数据结构
typedef struct free_pages{
  uint64_t free_num;
  page_meta_t ** free_list;
}free_pages_t;
free_pages_t * free_pages;

//初始化，传入的是文件的名称，文件名称也可以定义在头文件中间。
int cbs_init(const char * filename)
{
  size_t mapped_len, pagecache_num;
  int is_pmem;
  int i;
  //将持久内存映射到虚拟地址空间并得到基地址 pmem_base
  if ((pmem_base = (char *)pmem_map_file(filename, PMEM_SIZE,
                                         PMEM_FILE_CREATE, 0666,
                                         &mapped_len, &is_pmem)) == NULL) {
    perror("Pmem map file failed");
    return -1;
  }
  printf("pmem_map_file mapped_len=%ld, pmem_base=%p, is_pmem=%ld\n", mapped_len,
pmem_base, is_pmem);
  pmem_layout_t * pmem_data =(pmem_layout_t *) pmem_base;
  page_meta_t * pmem_meta = (page_meta_t
*)((uint64_t)pmem_base+sizeof(pmem_layout_t));
```

```
  //分配内存数据结构
  disk=(disk_t *)calloc(sizeof(char),sizeof(disk_t));
  if(disk == NULL) return -1;

  free_pages = (free_pages_t*)(malloc(sizeof(free_pages_t)));
  free_pages->free_list =(page_meta_t **)
malloc(PMEM_PAGE_NUMBERS*sizeof(page_meta_t *));
  if(free_pages == NULL || free_pages->free_list == NULL) {
    printf("DRAM space is not enough for store the free_list\n");
    return -1;
  }

  //check magic number, 如果 magic number 已经写过了，那可以恢复 disk 数据结构，否则所有的页都
是 free 的，建立 free pages 数据结构
  if(pmem_data->magic_number != MAGIC_NUMBER) {
    //first time init and write the whole block structure to 0
    //pmem_memset_persist(pmem_base,0x00,PMEM_SIZE);
    pmem_memset_persist(pmem_meta,0x0,sizeof(page_meta_t)*PMEM_PAGE_NUMBERS);
    page_address = (uint64_t)(((uint64_t )pmem_meta +
sizeof(page_meta_t)*PMEM_PAGE_NUMBERS + PAGE_SIZE) & 0xfffffffffffff000);
    pmem_data->page_offset = page_address - (uint64_t)pmem_base;
    pmem_persist(&(pmem_data-> page_offset),sizeof(pmem_data-> page_offset));

    pmem_data->magic_number = MAGIC_NUMBER;
    pmem_persist(pmem_data,sizeof(pmem_data->magic_number));

    for(i=0;i<PMEM_PAGE_NUMBERS;i++) {
      free_pages->free_list[i]=pmem_meta;
      pmem_meta+=1;
    }
    free_pages->free_num=PMEM_PAGE_NUMBERS;
  }
  else {
    int j=0;
    uint64_t block_id,page_id;
    uint64_t req_id;
    //magic number check pass, that means we might have free pages caches.
    for(i=0;i<PMEM_PAGE_NUMBERS;i++) {
      if(pmem_meta->valid == 0) {
        free_pages->free_list[j]=pmem_meta;
        pmem_meta+=1;
        j++;
      } else {
        //fill the disk structure.
        req_id = pmem_meta->req_id;
        block_id = req_id >> 10;
        page_id = req_id & 0x3ff;
        if(disk->blocks[block_id].pages[page_id].page_id1 == 0) {
          disk->blocks[block_id].pages[page_id].page_id1 =
PAGE_ID_FROM_META(pmem_meta);
          disk->blocks[block_id].cached_pages_cnt++;
        }else {
          disk->blocks[block_id].pages[page_id].page_id1 =
PAGE_ID_FROM_META(pmem_meta);
        }
        pmem_meta +=1;
      }
    }
    free_pages->free_num=j;
    page_address=pmem_data->page_offset + (uint64_t) pmem_base;
```

```
  }
  printf("init done, pagecache_num=%d,free page number=%d\n",PMEM_PAGE_NUMBERS,
free_pages->free_num);
  return 0;
}

int get_cached_count()
{
  int i;
  int cnt=0;
  for(i=0;i<DISK_BLOCK_NUM;i++) {
    cnt+=disk->blocks[i].cached_pages_cnt;
  }
  return cnt;
}

//写入或者更新一个页，输入 req_id 表示更新到什么地方，content 表示更新的内容
int write_req(uint64_t req_id, unsigned char * content) {
  //req_id>>10 是 block_id(1024 pages/block); req_id&0x3ff 是该 block 中的 page_id.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;
  uint64_t atomic_value=0;
  uint64_t page_id1, page_id2;
  uint64_t free_num=free_pages->free_num;

  if(block_id > DISK_BLOCK_NUM) {
    printf(" write req_id is not valid and over the disk range\n");
    return -1;
  }
  page_id1=disk->blocks[block_id].pages[req_page_id].page_id1;
  page_id2=disk->blocks[block_id].pages[req_page_id].page_id2;
  page_meta_t ** free_list = free_pages->free_list;
  if(free_num ==0 ) {
    printf("thre is no free pages in the pmem\n");
  }

  if(page_id1==0 && page_id2==0) {
    //there is no page in the location, add one page; Get one free page meta in the
free_list
    page_meta_t * page_new = free_list[free_num-1];
    pmem_memcpy_persist(PAGE_FROM_META(page_new), content, 4096);  //从 pagemeta 中拿到
4k page，将数据写入并持久化
    atomic_value = req_id<<32|1;
    pmem_memcpy_persist(page_new,&atomic_value,8); //更新 page meta valid=1, sn=0,
req_id, atomic 写入

    //更新内存中的数据结构，free number 减一， block 中 page_id1 更新，cached_page_cnt 加一
    free_pages->free_num-=1;
    disk->blocks[block_id].pages[req_page_id].page_id1=PAGE_ID_FROM_META(page_new);
    disk->blocks[block_id].cached_pages_cnt++;
  }
  else if((page_id1!=0 && page_id2==0)||(page_id1==0 && page_id2!=0)) {
    //已经有一个数据，此时需要更新，不能原地更新，必须先写道新的位置上
    page_meta_t * page_old;
    if(page_id1!=0) {
      page_old=PAGE_META_FROM_ID(page_id1);
    }else {
      page_old=PAGE_META_FROM_ID(page_id2);
    }
```

```
    if(page_old->sn!=0) {
      page_old->sn=0;
      pmem_persist(page_old,8);  //老的页先把 sn reset 为 0
    }
    //找到一个新的 page meta 和 4kpage，先将 4k 数据写入，如果写入过程断电，由于 page_meta 不会更新，所以恢复不会出错。
    page_meta_t * page_new = free_list[free_num-1];
    pmem_memcpy_persist(PAGE_FROM_META(page_new), content, 4096);
    atomic_value = req_id<<32|1<<1|1;  //sn=1,  req_id, valid=1, 原子写入
    pmem_memcpy_persist(page_new,&atomic_value, 8);
    disk->blocks[block_id].pages[req_page_id].page_id2=PAGE_ID_FROM_META(page_new);

    //如果此时断电，在这个 req_id 上有两个页都是有效的，其中 sn=0 的是老的数据，sn=1 的是新的数据。
    //将老的数据 free，将老的 meta page_old 写入 0,  sn=0; valid=0, req_id=0;
    atomic_value=0;
    pmem_memcpy_persist(page_old,&atomic_value, 8);
    free_list[free_num-1]=page_old;
    disk->blocks[block_id].pages[req_page_id].page_id1=0;
  }
  else if(page_id1!=0 && page_id2!=0) {
    // 如果在这个 req_id 上两个页都是有效的，检查 sn，sn 越大的数据越新。
    page_meta_t * page1_meta, *page2_meta;
    page1_meta=PAGE_META_FROM_ID(page_id1);
    page2_meta=PAGE_META_FROM_ID(page_id2);
    if(page1_meta->sn < page2_meta->sn) { //page_id2 是新的数据
      //更新 page_id1,让 page_id1 对应的数据更新
      pmem_memcpy_persist(PAGE_FROM_META(page1_meta), content, 4096);
      atomic_value = req_id<<32|(page2_meta->sn++)<<1|1;
      pmem_memcpy_persist(page1_meta,&atomic_value, 8);

      atomic_value=0; //释放 page_id2 对应得数据
      pmem_memcpy_persist(page2_meta,&atomic_value, 8);
      disk->blocks[block_id].pages[req_page_id].page_id2=0;
      free_num++;
      free_pages->free_num=free_num;
      free_list[free_num-1]=page2_meta;
    } else {
      pmem_memcpy_persist(PAGE_FROM_META(page2_meta), content, 4096);
      atomic_value = req_id<<32|(page1_meta->sn++)<<1|1;
      pmem_memcpy_persist(page2_meta,&atomic_value, 8);

      atomic_value=0;
      pmem_memcpy_persist(page2_meta,&atomic_value, 8);
      disk->blocks[block_id].pages[req_page_id].page_id1=0;
      free_num++;
      free_pages->free_num=free_num;
      free_list[free_num-1]=page1_meta;
    }
  }
  return 0;
}

void * read_req(uint64_t req_id) {
  // req_id and content; only after the init success, then this API can be called.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;
  uint64_t atomic_value=0;
  uint64_t page_id1, page_id2;
```

```
  if(block_id > DISK_BLOCK_NUM) {
    printf("read req_id is not valid and over the disk range\n");
    return NULL;
  }

  page_id1=disk->blocks[block_id].pages[req_page_id].page_id1;
  page_id2=disk->blocks[block_id].pages[req_page_id].page_id2;

  if((page_id1!=0 && page_id2 == 0) ||(page_id1==0 && page_id2!=0)) { // only page1
is there
    page_meta_t * page_meta;
    if(page_id1!=0) {
      page_meta=PAGE_META_FROM_ID(page_id1);
    }else {
      page_meta=PAGE_META_FROM_ID(page_id2);
    }
    return PAGE_FROM_META(page_meta);

  } else if(page_id1!=0 && page_id2!=0) { // two pages
    page_meta_t * page1_meta=PAGE_META_FROM_ID(page_id1);
    page_meta_t * page2_meta=PAGE_META_FROM_ID(page_id2);
    if(page1_meta->sn < page2_meta->sn) {
      return PAGE_FROM_META(page2_meta);
    } else {
      return PAGE_FROM_META(page1_meta);
    }
  } else {
    printf("cache is not exist\n");
  }

  return NULL;
}

void delete_page(uint64_t req_id) {
  // req_id and content; only after the init success, then this API can be called.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;
  uint64_t atomic_value=0;
  uint64_t page_id1, page_id2;
  if(block_id > DISK_BLOCK_NUM) {
    printf("read req_id is not valid and over the disk range\n");
    return ;
  }
  page_id1=disk->blocks[block_id].pages[req_page_id].page_id1;
  page_id2=disk->blocks[block_id].pages[req_page_id].page_id2;

  uint64_t free_num=free_pages->free_num;
  page_meta_t ** free_list = free_pages->free_list;
  if(free_num ==0 ) {
    printf("there is no free pages in the pmem\n");
  }
  atomic_value=0;
  if((page_id1!=0 && page_id2 == 0) ||(page_id1==0 && page_id2!=0)) { // only one
page is there
    page_meta_t * page_meta;
    if(page_id1!=0) {
      page_meta=PAGE_META_FROM_ID(page_id1);
      disk->blocks[block_id].pages[req_page_id].page_id1=0;
    }else {
      page_meta=PAGE_META_FROM_ID(page_id2);
      disk->blocks[block_id].pages[req_page_id].page_id2=0;
    }
    pmem_memcpy_persist(page_meta,&atomic_value, 8);
```

```
    free_num++;
    free_pages->free_num=free_num;
    free_list[free_num-1]=page_meta;
  } else if(page_id1!=0 && page_id2!=0) { // two pages
    page_meta_t * page1_meta=PAGE_META_FROM_ID(page_id1);
    page_meta_t * page2_meta=PAGE_META_FROM_ID(page_id2);
    if(page1_meta->sn < page2_meta->sn) {
      pmem_memcpy_persist(page1_meta,&atomic_value, 8);
      pmem_memcpy_persist(page2_meta,&atomic_value, 8);
    } else {
      pmem_memcpy_persist(page2_meta,&atomic_value, 8);
      pmem_memcpy_persist(page1_meta,&atomic_value, 8);
    }
    disk->blocks[block_id].pages[req_page_id].page_id1=0;
    disk->blocks[block_id].pages[req_page_id].page_id2=0;
    free_num++;
    free_list[free_num-1]=page1_meta;
    free_num++;
    free_list[free_num-1]=page2_meta;
    free_pages->free_num=free_num;
  } else {
    printf("cache is not exist\n");
  }
}

#define WRITE_COUNT 100000
#define OVERWRITE_COUNT 10000
int main()
{
   // calculate the time
  unsigned char * page_content=(unsigned char *)malloc(PAGE_SIZE);
  uint64_t i=0;
  auto start=std::chrono::steady_clock::now();
  auto stop=std::chrono::steady_clock::now();
  std::chrono::duration<double> diff=stop-start;

  unsigned char * read_content;
  memset(page_content,0xab,PAGE_SIZE);

  start=std::chrono::steady_clock::now();
  cbs_init("/mnt/pmem0/cbs_file");
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"cbs_init time "<<diff.count()<<std::endl;
  std::cout<<"cached page count" << get_cached_count()<<std::endl;

  start = std::chrono::steady_clock::now();
  for(i=0;i<WRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"write_req time "<<diff.count()/WRITE_COUNT<<std::endl;

  memset(page_content,0xcd,PAGE_SIZE);

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
```

```
  std::cout<<"overwrite write_req update take time "<<
diff.count()/OVERWRITE_COUNT<<std::endl;

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    read_content=(unsigned char *)read_req(i);
    memcpy(page_content,read_content,PAGE_SIZE);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite read_req take time
"<<diff.count()/OVERWRITE_COUNT<<std::endl;
  printf("the page should fill with paten 0xcd, 0x%x\n", page_content[0]);

  start = std::chrono::steady_clock::now();
  for(i=OVERWRITE_COUNT;i<WRITE_COUNT;i++) {
    read_content=(unsigned char *)read_req(i);
    memcpy(page_content,read_content,PAGE_SIZE);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite->write count read_req take time
"<<diff.count()/(WRITE_COUNT-OVERWRITE_COUNT)<<std::endl;
  printf("the page should fill with patern 0xab, 0x%x\n", page_content[0]);

  //start = std::chrono::steady_clock::now();
  //for(i=0;i<WRITE_COUNT;i++) {
  //  delete_page(i);
  //}
  //stop=std::chrono::steady_clock::now();
  //diff=stop-start;
  //std::cout<<"delete write count take time "<<diff.count()/WRITE_COUNT<<std::endl;
  return 0;
}
```

编译"g++ cbs_req.cpp -o cbs_req -lpmem -O2",然后使用“taskset -c 2 ./cbs_req 运行这段代码，我们看到写一个 4KB 页大概花费 2.7us，恢复的过程大概时 0.1s；测试的具体结果如下：

```
➜  ~ taskset -c 2 ./cbs_req_new
pmem_map_file mapped_len=107374182400, is_pmem=1
init done, pagecache_num=26163298,free page number=26163298
cbs_init time 0.201417
cached page count0
write_req time 4.26528e-06
overwrite write_req update take time 2.15705e-06
overwrite read_req take time 1.06379e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 1.07138e-06
the page should fill with patern 0xab, 0xab
➜  ~ vim cbs_req_new.cpp
➜  ~ taskset -c 2 ./cbs_req_new
pmem_map_file mapped_len=107374182400, is_pmem=1
init done, pagecache_num=26163298,free page number=26063298
cbs_init time 0.108116
cached page count100000
write_req time 2.19311e-06
overwrite write_req update take time 2.71467e-06
overwrite read_req take time 1.04566e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 1.06975e-06
the page should fill with patern 0xab, 0xab
```

程序员如果只想完全原始地访问持久内存并且无需库提供分配器或事务功能，那么可以将 libpmem 用作开发的基础，libpmem 支持应用程序采用自定义内存管理和恢复逻辑来实现高性能的持久化应用。对于大多数程序员而言，libpmem 非常底层，使用它有一定的难度。

该程序中有一个性能优化点（由于我们第一代 CLX+AEP，CLWB 不能正确的工作，会导致在 cacheline flush 时，cache 中的数据被刷出到持久域的时候同时被 invalid，导致如果该数据再次被读就会 cache miss，从而影响到数据读的性能），大家可以想一想可以如何优化？

在该程序中，page_meta 的大小是 8 个字节，当一个 page_meta 写完后会 invalid 整个 64 字节，从而导致后面的 page_meta 读的时候产生 page cache。优化的方法就是：

```
typedef struct page_meta {
  uint64_t valid:1; //valid 表示该 page 是否已经分配和使用，如果是 0，表示该 page 没有被使用
  uint64_t sn:31;   //sequence number 来表示页的新旧，sequence 越大表示页越新，old page 一般 sn 会被重置为 0
  uint64_t req_id:32;  //req_id, which is in the range from 1~1024*1024*1024
  uint64_t padding[7]; //for performance, avoid false sharing "R-M-F"
}page_meta_t;
//magic number，可以不定义这个数据结构，因为位置是固定的
typedef struct pmem_layout {
  uint64_t magic_number;
  uint64_t page_offset;    //real page offset，保证这个 page address 4K 对齐。
  uint64_t padding[6];
}pmem_layout_t;
```

原先的 page_meta 只是 8 个字节，现在增加到了 64 字节，这样就会大大优化性能，此时性能能够做到从 **2.7us 优化到 1.7us （reduce latency 60%）**：

```
➜  ~ taskset -c 2 ./cbs_req_new
pmem_map_file mapped_len=107374182400, is_pmem=1
init done, pagecache_num= 25811100,free page number=23588351
cbs_init time 5.33871
cached page count0
write_req time 4.46733e-06
overwrite write_req update take time 1.71743e-06
overwrite read_req take time 1.04520e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 1.06970e-06
the page should fill with patern 0xab, 0xab
➜  ~ taskset -c 2 ./cbs_req_new
pmem_map_file mapped_len=107374182400, is_pmem=1
init done, pagecache_num= 25811100,free page number=23488351
cbs_init time 0.847513
cached page count100000
write_req time 1.75316e-06
overwrite write_req update take time 1.82072e-06
overwrite read_req take time 1.04660e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 1.05974e-06
the page should fill with patern 0xab, 0xab
```

当你切换到 ICX+BPS 的平台后，CLWB 可以正常工作，在 cacheline 被刷新后， 并不会从 cache 中 invalid，所以 cache 的数据仍然可以访问而不会产生 cache miss。在 ICX+BPS 这个优化不是必须的。下面的结果是在 ICX+BPS 下，不做任何 padding 之后的结果：写的结果类似（稍微

慢一点，这个与 CPU 的频率有关），但是读的性能比 CLX+AEP 有 10%~20%的提升，具体的原因在这里我们不详细讨论，但是我们可以知道一点，CLWB 在 ICX+BPS 下可以正常的工作。

```
[root@localhost ~]# taskset -c 2 ./cbs_req_new
pmem_map_file mapped_len=107374182400, is_pmem=1
init done, pagecache_num=26163298,free page number=26163298
cbs_init time 55.6604
cached page count0
write_req time 1.60401e-06
overwrite write_req update take time 1.80627e-06
overwrite read_req take time 1.07942e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 8.9152e-07
the page should fill with patern 0xab, 0xab
[root@localhost ~]# taskset -c 2 ./cbs_req_new
pmem_map_file mapped_len=107374182400, is_pmem=1
init done, pagecache_num=26163298,free page number=26063298
cbs_init time 0.164735
cached page count100000
write_req time 1.86702e-06
overwrite write_req update take time 2.00082e-06
overwrite read_req take time 8.18381e-07
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 8.41381e-07
the page should fill with patern 0xab, 0xab
```

另一个问题是在这个程序中，当第一次创建和映射文件后，第一次写的性能需要 4.59us，这个的主要原因什么？主要原因是，第一次写的时候，会产生缺页中断将虚拟地址和真实的物理页相连接并将该映射写入页表，缺页中断会导致清零的操作代价较大，但一旦完成写入，虚拟地址和物理地址的连接已经建立，后面就不会产生缺页中断。所以，我们可以在第一次初始化的时候，通过将每个虚拟页的地址写入一个 0，就可以完成缺页中断。

```
  //check magic number, 如果 magic number 已经写过了，那可以恢复 disk 数据结构，否则所有的页都是 free 的，建立 free pages 数据结构
  if(pmem_data->magic_number != MAGIC_NUMBER) {
    //first time init and write the whole block structure to 0
    //pmem_memset_persist(pmem_base,0x00,PMEM_SIZE);

    pmem_memset_persist(pmem_meta,0x0,sizeof(page_meta_t)*PMEM_PAGE_NUMBERS);
    pmem_data->magic_number = MAGIC_NUMBER;
    pmem_persist(pmem_data,sizeof(pmem_layout_t));

    for(i=0;i<PMEM_PAGE_NUMBERS;i++) {
      free_pages->free_list[i]=pmem_meta;
      pmem_memset_persist(PAGE_FROM_META(pmem_meta),0x0,8);//pre-fault
      pmem_meta+=1;
    }
    free_pages->free_num=PMEM_PAGE_NUMBERS;
  }
```

虽然第一次初始化的时间从 5.33s 增加到了 43.46s，却可以降低第一次写入的时间到 **1.53us**。

```
➜  ~ taskset -c 2 ./cbs_req_new
pmem_map_file mapped_len=107374182400, is_pmem=1
init done, pagecache_num= 25811100,free page number=23588351
cbs_init time 43.4666
cached page count0
write_req time 1.53175e-06
overwrite write_req update take time 1.69467e-06
overwrite read_req take time 1.04470e-06
```

```
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 1.06070e-06
the page should fill with patern 0xab, 0xab
➜  ~ taskset -c 2 ./cbs_req_new
pmem_map_file mapped_len=107374182400, is_pmem=1
init done, pagecache_num= 25811100,free page number=23488351
cbs_init time 0.846013
cached page count100000
write_req time 1.75408e-06
overwrite write_req update take time 1.82161e-06
overwrite read_req take time 1.04456e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 1.06950e-06
the page should fill with patern 0xab, 0xab
```

可以使用工具 pmemcheck 来检查程序的正确性比如什么数据在持久内存中应该持久化去没有调用相应的函数，安装 https://github.com/pmem/valgrind, 然后测试你的程序“valgrind --tool=pmemcheck ./your_program”，可以通过 log 来检查你的程序。由于 pmemcheck 是一个运行非常重的工具，所以对于应用的持久内存大小是有限制的。但是由于 pmemcheck 主要是来检测功能的，所以我们可以将我们的内存池 从 100GB 减少到 10GB 来进行验证测试。

```
#define PMEM_SIZE 10*1024*1024*1024UL  //定义我们使用持久内存的大小
```

然后可以运行这样的程序来检查：

```
➜  ~ valgrind --tool=pmemcheck ./cbs_req_new
==248561== pmemcheck-1.0, a simple persistent store checker
==248561== Copyright (c) 2014-2020, Intel Corporation
==248561== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==248561== Command: ./cbs_req_new
==248561==
pmem_map_file mapped_len=10737418240, is_pmem=1
init done, pagecache_num=2616328,free page number=2616328
cbs_init time 1.41468
cached page count0
write_req time 0.000361661
overwrite write_req update take time 0.000359447
overwrite read_req take time 5.06452e-08
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 2.46624e-08
the page should fill with patern 0xab, 0xab
==248561==
==248561== Number of stores not made persistent: 0
==248561== ERROR SUMMARY: 0 errors
➜  ~ valgrind --tool=pmemcheck ./cbs_req_new
==248566== pmemcheck-1.0, a simple persistent store checker
==248566== Copyright (c) 2014-2020, Intel Corporation
==248566== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==248566== Command: ./cbs_req_new
==248566==
pmem_map_file mapped_len=10737418240, is_pmem=1
init done, pagecache_num=2616328,free page number=2516328
cbs_init time 0.0506498
cached page count100000
write_req time 0.000358157
overwrite write_req update take time 0.00035847
overwrite read_req take time 5.01243e-08
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 2.46348e-08
the page should fill with patern 0xab, 0xab
==248566==
```

```
==248566== Number of stores not made persistent: 0
==248566== ERROR SUMMARY: 0 errors
```

Call For Action:

1. http://pmem.io/pmdk/libpmem/;
2. https://github.com/pmem/pmdk/tree/master/src/examples/libpmem
3. https://github.com/twitter/pelikan/blob/master/src/datapool/datapool_pmem.c
4. 如果要将一个 block 下的多个更新作为一个事务，使用 libpmem 需要考虑的比较复杂。比如需要有一个事务开始及完成的标识，如果事务没有完成，重启之后需要清理该 block 中的所有新的数据回到原有的数据。在事务开始之前，清理该 block 中所有有两份 page 版本的数据，只保留一个版本。主要完成这样的函数 int write_req_tx(uint64_t block_id, uint64_t *req_page_id, unsigned char ** content, int number) {}; 事务也是需要用到 8 字节的原子性来却保那些数据真正的完成了写入。

Tips:

1. 保存 offset 而不是真正的地址，因为每次 mmap()可能会映射到不一样的基地址。
2. 使用 8 字节的原子性作为 flag 来保证所写入的完整性和一致性。
3. 不要原地更新超过 8 字节的数据，而是使用 sequency number（SN）来保证老的数据不会被覆盖，直到新的数据写成功才可以释放老的数据。
4. 使用持久内存中的数据记录来恢复内存中的数据结构，这样可以减少编程的复杂性，同时可以提升性能。

libpmemobj

libpmemobj 是一种提供事务对象存储的 C 库，可为持久内存编程提供动态内存分配器、事务和常规功能。该库可以解决在持久内存编程时遇到的许多常见的算法和数据问题。如果选择使用 C 编程语言，而且需要使用通用内存分配器和事务时，则可以使用 libpmemobj。基于持久化内存的挑战，数据需要保持数据原子性、断电一致性，同时能够恢复数据。libpmemobj 是一个具有事务性的通用的持久化库，可以解决持久化内存所面临的各个挑战。如图 3 libpmemobj 的接口框架， libpmemobj 依赖 libpmem 提供持久化的原语支持，并且提供和 libpmem 接口类似的原语接口（Primitives APIs），这些接口和 libpmem 一样不支持事务性的操作，应用需要考虑数据原子性、一致性以及数据的恢复。libpmemobj 利用统一的日志（unified logs）来实现数据的事务特性。持久化内存的原子分配和事务的接口（Atomic APIs 、Tracactional APIs、Action APIs）都是基于这些统一的日志来实现的。为了保证数据的可恢复性，需要考虑持久化内存的位置的独立性，所以应用不能利用直接地址来保存数据。libpmemobj 使用偏移指针（offset pointers），这是相对于内存池基地址的偏移来表示一个数据对象。持久化内存是以文件的方式暴露给应用，通过 MMAP 来获得线性地址，让应用可以使用字节访问的方式直接访问持久化内存。

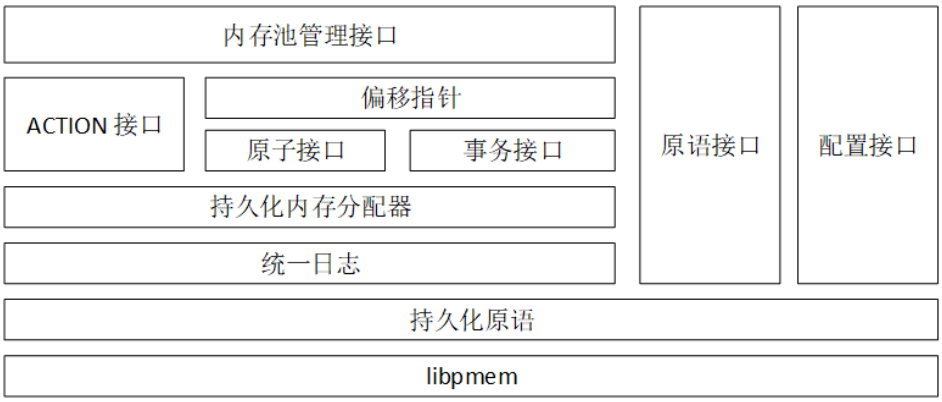


图 3 libpmemobj 的接口框架

libpmemobj 接口的申明在 libpmemobj.h 这个头文件中， 如果你已经正确的安装了 PMDK 的库那么头文件默认会安装在程序可以正确找到路径的位置。当你正确的引用了 libpmemobj.h 和链接了 libpmemobj、libpmem 的库之后，你就可以使用 libpmemobj 的接口。libpmemobj 的接口非常的多，具体接口的描述可以通过 https://pmem.io/pmdk/libpmemobj/查询。

使用 libpmemobj 把持久内存作为 SSD 存储的持久缓存页，需要考虑这样的特性：

1. 一个 4T 的 SSD 设备可以分成 1024*1024 个 block，每个 block 里面有 1024 个 4KB pages, pages 需要保存在持久内存中并持久化。
2. 要考虑每一个页的事务性，即不能出现脏页。
3. 数据在系统奔溃或者突然断电的情况下，正确的恢复数据
4. 在这个例子中我们只考虑单线程的情况。

所有的 pages 都通过 libpmemobj 保存在持久内存中间，需要考虑到以下几点：

1. libpmemobj 已经考虑到了对齐的问题。
2. block 中的 4K 页在更新时可以通过事务的方式去实现，首先将要修改的数据保存在 undo log 中，然后修改数据。如果事务成功，undo log 会被删除；如果事务失败，在恢复数据时，会从 undo log 中恢复原有的数据。
3. block 中的 4K 页可以直接通过 libpmemobj 的分配器去进行管理。
4. libpmeobj 中所有的对象都是保存的 offset pointer。

示例 2（270 行）中使用的 libpmemobj 核心接口主要包括：

1. pop=pmemobj_create(filename, LAYOUT_NAME,PMEM_SIZE, 0666);
2. pop = pmemobj_open(filename, POBJ_LAYOUT_NAME(cbs_cache));
3. PMEMoid root = pmemobj_root(pop, sizeof(struct root));
4. rootp = (struct root *)pmemobj_direct(root);
5. OID_IS_NULL(block_oid)
6. block_oid = pmemobj_tx_zalloc(sizeof(struct block), 0);
7. PMEMoid page_oid = pmemobj_tx_alloc(sizeof(struct page), 0);
8. pmemobj_tx_add_range_direct(pagep->rpage,PAGE_SIZE);

9. pmemobj_tx_free(page_oid);
10. TX_BEGIN(pop) {
11. }TX_END

示例 2 libpmemobj 实现持久化的页缓存

```
#include <iostream>
#include <chrono>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <libpmem.h>
#include <time.h>
#include <string.h>
#include <cassert>

//使用 libpmeobj 来保存所有的 page cache
#include <libpmemobj.h>

#define PAGE_SIZE 4096
#define BLOCK_PAGE_NUM 1024
#define REQ_BLOCK 10
#define REQ_PAGE 0x3ff
#define DISK_BLOCK_NUM 1048576 //1024*1024 blocks
#define PMEM_SIZE 100*1024*1024*1024UL  //定义我们使用持久内存的大小

/***************************************
 * \
 * pmem layout as:
 --block_oid[0]--|--block_oid[1]--|...|--[[[page_oid][page_oid]...]cached_page_cnt]--|
rebuild the block dram structure after the system restart.
 */
//cbs cache layout in pmem
#define LAYOUT_NAME "cbs_cache"
POBJ_LAYOUT_BEGIN(cbs_cache);
POBJ_LAYOUT_ROOT(cbs_cache, struct root);
POBJ_LAYOUT_END(cbs_cache);

//block conents have the BLOCK_PAGE_NUM, pages.
struct page {
  unsigned char rpage[PAGE_SIZE];
};

struct block { /* array-based queue container */
  PMEMoid  pages[BLOCK_PAGE_NUM];
  uint64_t cached_page_cnt;
};
//root object max size is 15G, defined in the head file. root size is limited, so at the poc, we just se the BLOCK NUM as 1000;
struct root {
  PMEMoid blocks[DISK_BLOCK_NUM];
};

struct root * rootp=NULL;
PMEMobjpool *pop=NULL;

//初始化，传入的是文件的名称，文件名称也可以定义在头文件中间。
int cbs_init(const char * filename)
{
  //printf("cbs_init, the filename=%s\n",filename);
```

```
  pop = pmemobj_create(filename, LAYOUT_NAME,
                                PMEM_SIZE, 0666);
  if (pop != NULL) {
    printf("pmemobj_create successfully\n");
  } else {
    pop = pmemobj_open(filename, POBJ_LAYOUT_NAME(cbs_cache));
    if (pop == NULL) {
      printf("failed to open the pool\n");
      return 1;
    }
  }

  PMEMoid root = pmemobj_root(pop, sizeof(struct root));
  rootp = (struct root *)pmemobj_direct(root);
  if(rootp==NULL) {
    printf("rootp is NULL, please check your persistent memory pool\n");
    return 1;
  }
  return 0;
}

int get_cached_count() {
  int i;
  int cnt;
  PMEMoid block_oid;
  struct block * blockp=NULL;
  for (i=0;i<DISK_BLOCK_NUM;i++) {
    block_oid=rootp->blocks[i];
    if(!OID_IS_NULL(block_oid)) {
      blockp=(struct block *) pmemobj_direct(block_oid);
      cnt+=blockp->cached_page_cnt;
    }
  }
  return cnt;
}

//写入或者更新一个页，输入 req_id 表示更新到什么地方，content 表示更新的内容
int write_req(uint64_t req_id, unsigned char * content) {
  //req_id>>10 是 block_id(1024 pages/block); req_id&0x3ff 是该 block 中的 page_id.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;
  assert(req_page_id<BLOCK_PAGE_NUM); //req id and req number in one block, can't
over 1024; req id should not duplicate.
  TX_BEGIN(pop) {
    PMEMoid block_oid = rootp->blocks[block_id];   //block_oid,找到相应的 block

    if(OID_IS_NULL(block_oid)) { //if block_oid is NULL, 我们先分配相应的 block 还有相应的
page
      block_oid = pmemobj_tx_alloc(sizeof(struct block), 0);
      struct block * blockp=(struct block *) pmemobj_direct(block_oid);
      assert(blockp!=NULL);
      memset(blockp,0x0,sizeof(struct block));
      PMEMoid page_oid = pmemobj_tx_alloc(sizeof(struct page), 0);
      struct page * pagep = (struct page *) pmemobj_direct(page_oid);
      assert(pagep!=NULL);
      memcpy(pagep->rpage,content,PAGE_SIZE);

      //rootp->blocks[block_id] update，更新之前需要做 snapshot
      pmemobj_tx_add_range_direct(&rootp->blocks[block_id],sizeof(PMEMoid));
      rootp->blocks[block_id]=block_oid;
```

```
      pmemobj_tx_add_range_direct(&blockp->pages[req_page_id],sizeof(PMEMoid));
      blockp->pages[req_page_id]=page_oid;

      pmemobj_tx_add_range_direct(&blockp->cached_page_cnt,sizeof(uint64_t));
      blockp->cached_page_cnt++;
    } else {  //block 已经在持久内存中了
      struct block * blockp=(struct block *) pmemobj_direct(block_oid);
      PMEMoid page_oid = blockp->pages[req_page_id];
      if(OID_IS_NULL(page_oid)) {//如果相对应的 page 还没有分配，分配 page
        page_oid = pmemobj_tx_alloc(sizeof(struct page), 0);
        struct page * pagep= (struct page *)pmemobj_direct(page_oid);
        assert(pagep!=NULL);
        memcpy(pagep->rpage,content,PAGE_SIZE);
        // update need to tx_add snapshot;
        pmemobj_tx_add_range_direct(&blockp->pages[req_page_id],sizeof(PMEMoid));
        blockp->pages[req_page_id]=page_oid;

        pmemobj_tx_add_range_direct(&blockp->cached_page_cnt,sizeof(uint64_t));
        // 8 bytes, no need to snapshot.
        blockp->cached_page_cnt++;
      }else { //如果已经分配了 page，将原有 page 的内容填入 undo log;  通过
pmemobj_tx_add_range_direct(pagep->rpage,PAGE_SIZE);
        struct page * pagep= (struct page *)pmemobj_direct(page_oid);
        assert(pagep!=NULL);
        pmemobj_tx_add_range_direct(pagep->rpage,PAGE_SIZE);
        memcpy(pagep->rpage, content,PAGE_SIZE);
      }
    }
  }TX_END
  return 0;
}

void * read_req(uint64_t req_id) {
  // req_id and content; only after the init success, then this API can be called.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;
  PMEMoid block_oid = rootp->blocks[block_id];
  if(OID_IS_NULL(block_oid)) {
    return NULL;
  }else {
    struct block * blockp=(struct block *) pmemobj_direct(block_oid);
    PMEMoid page_oid = blockp->pages[req_page_id];
    if(OID_IS_NULL(page_oid)) {
      return NULL;
    }else {
      struct page * pagep= (struct page *)pmemobj_direct(page_oid);
      assert(pagep!=NULL);
      return pagep->rpage;
    }
  }
}

void delete_page(uint64_t req_id) {
  // req_id and content; only after the init success, then this API can be called.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;
  PMEMoid block_oid = rootp->blocks[block_id];
  if(OID_IS_NULL(block_oid)) {
    return ;
  } else {
    struct block * blockp=(struct block *) pmemobj_direct(block_oid);
```

```
    PMEMoid page_oid = blockp->pages[req_page_id];
    if(OID_IS_NULL(page_oid)) {
      return ;
    } else {
      TX_BEGIN(pop) {
        pmemobj_tx_free(page_oid);
      }TX_END
    }
  }
}

#define WRITE_COUNT 100000
#define OVERWRITE_COUNT 10000
int main()
{
  // calculate the time
  unsigned char * page_content=(unsigned char *)malloc(4096);
  uint64_t i=0;
  auto start=std::chrono::steady_clock::now();
  auto stop=std::chrono::steady_clock::now();
  std::chrono::duration<double> diff=stop-start;

  unsigned char * read_content;
  memset(page_content,0xab,4096);

  start=std::chrono::steady_clock::now();
  cbs_init("/mnt/pmem0/cbs_cache");
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"cbs_init time"<<diff.count()<<std::endl;

  std::cout<<"cached page count"<<get_cached_count()<<std::endl;

  start = std::chrono::steady_clock::now();
  for(i=0;i<WRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"write_req time"<<diff.count()/WRITE_COUNT<<std::endl;

  memset(page_content,0xcd,4096);

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite write_req update take time "<<
diff.count()/OVERWRITE_COUNT<<std::endl;

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    read_content=( unsigned char *)read_req(i);
    memcpy(page_content,read_content,PAGE_SIZE);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite read_req take time
"<<diff.count()/OVERWRITE_COUNT<<std::endl;
  printf("the page should fill with paten 0xcd, 0x%x\n", read_content[0]);
```

```
  start = std::chrono::steady_clock::now();
  for(i=OVERWRITE_COUNT;i<WRITE_COUNT;i++) {
    read_content=( unsigned char *)read_req(i);
    memcpy(page_content,read_content,PAGE_SIZE);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite->write count read_req take time
"<<diff.count()/(WRITE_COUNT-OVERWRITE_COUNT)<<std::endl;
  printf("the page should fill with patern 0xab, 0x%x\n", read_content[0]);

  //start = std::chrono::steady_clock::now();
  //for(i=0;i<WRITE_COUNT;i++) {
  //  delete_page(i);
  //}
  //stop=std::chrono::steady_clock::now();
  //diff=stop-start;
  //std::cout<<"delete write count take time "<<diff.count()/WRITE_COUNT<<std::endl;
  return 0;
}
```

编译"g++ cbs_req_obj.cpp -o cbs_req_obj -lpmemobj -O2",然后使用“taskset -c 2 ./cbs_req_obj 运行这段代码，我们看到写一个 4KB 页大概耗时 5.6us，恢复的过程大概时 0.03s；

```
➜  ~ taskset -c 2 ./cbs_req_new_pmemobj
pmemobj_create successfully
cbs_init time0.110575
cached page count0
write_req time7.33781e-06
overwrite write_req update take time 5.61578e-06
overwrite read_req take time 1.0457e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 1.11605e-06
the page should fill with patern 0xab, 0xab
➜  ~ taskset -c 2 ./cbs_req_new_pmemobj
cbs_init time0.0364766
cached page count100000
write_req time5.60939e-06
overwrite write_req update take time 5.58853e-06
overwrite read_req take time 1.05356e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 1.10197e-06
the page should fill with patern 0xab, 0xab
```

libpmemobj 创建的示例更简洁和直观，可以帮助程序员简化程序设计，减少程序创建时的错误，同时可以确保数据的事务性。但是 libpmemobj 由于要使用 redo 和 undo 这样的机制来保证数据的事务性，所以性能上有较大的开销。而且在空间上 libpmemobj 的对象 obj 至少会有 64 字节的开销，如果数据对象较小，那空间开销可能会非常的大。可以使用“pmempool info -O /mnt/pmem0/cbs_cache|more”来观察对象空间的大小。由于在事务中，snapshot 的创建时非常关键的，我们可以使用 pmemcheck 的工具“valgrind --tool=pmemcheck ./your_program”

对我们所写的程序进行检查。

Call For Action:

1. https://pmem.io/pmdk/libpmemobj/

2. https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj

Tips:

1. libpmemobj 不适合用于小的数据对象，因为 pmemobj 的一个对象的开销为 64 字节。
2. 如果您使用事务接口，在需要更新数据的时候，请不要忘记添加数据的快照 undo log。性能开销较大。
3. 如果要让多个操作成为一个事务，使用 libpmemobj 是一个很好的选择，这样可以减少开发的难度。只需要让多个操作放到 TX_BEGIN 和 TX_END 之间。

libpmemblk

libpmemblk 是用于管理固定大小数据块阵列的 C 库。它提供故障安全接口，以通过基于缓冲区的函数更新数据块。libpmemblk 仅用于需要简单固定数据块阵列，且不需要直接字节级访问数据块的情况。所以 libpmemblk 的接口简单且非常直接。在我们的示例中，主要使用的接口包括：

1. pbp = pmemblk_create(filename, ELEMENT_SIZE, POOL_SIZE, 0666);
2. pbp = pmemblk_open(filename, ELEMENT_SIZE);
3. nelements = pmemblk_nblock(pbp);
4. pmemblk_write(pbp, content, req_id)
5. pmemblk_read(pbp, buf, req_id)

所以这个库非常适合我们全文讨论的使用持久内存作为页缓存的需求。其实现示例 3 所示：

示例 3 libpmemblk 实现持久化页缓存

```
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <libpmemblk.h>
#include <iostream>
#include <chrono>
#include <cassert>


/* size of the pmemblk pool -- 100 GB */
#define POOL_SIZE ((uint64_t)((1<< 30)*100UL))

/* size of each element in the pmem pool */
#define ELEMENT_SIZE 4096

// pmem block pool
PMEMblkpool *pbp;
size_t nelements;

int cbs_init(const char * filename)
{
  /* create the pmemblk pool or open it if it already exists */
  pbp = pmemblk_create(filename, ELEMENT_SIZE, POOL_SIZE, 0666);
```

```
  if (pbp == NULL)
    pbp = pmemblk_open(filename, ELEMENT_SIZE);

  if (pbp == NULL) {
    perror(filename);
    return -1;
  }

  /* how many elements fit into the file? */
  nelements = pmemblk_nblock(pbp);
  printf("file holds %zu elements", nelements);
  return 0;
}

int write_req(uint64_t req_id, unsigned char * content) {
  assert(req_id < nelements);
  if (pmemblk_write(pbp, content, req_id) < 0) {
    perror("pmemblk_write");
    return -1;
  }

  return 0;
}

void * read_req(uint64_t req_id, unsigned char * buf) {
  assert(req_id<nelements);
  /* read the block at index 10 (reads as zeros initially) */
  if (pmemblk_read(pbp, buf, req_id) < 0) {
    perror("pmemblk_read");
    return NULL;
  }
  return buf;
}

#define WRITE_COUNT 100000
#define OVERWRITE_COUNT 10000
int main()
{
  // calculate the time
  unsigned char * read_content;
  unsigned char * page_content=(unsigned char *)malloc(4096);
  uint64_t i=0;
  auto start=std::chrono::steady_clock::now();
  auto stop=std::chrono::steady_clock::now();
  std::chrono::duration<double> diff=stop-start;

  memset(page_content,0xab,4096);

  start=std::chrono::steady_clock::now();
  cbs_init("/mnt/pmem0/cbs_pmblk");
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"cbs_init time"<<diff.count()<<std::endl;

  //std::cout<<"cached page count"<<get_cached_count()<<std::endl;
  start = std::chrono::steady_clock::now();
  for(i=0;i<WRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"write_req time"<<diff.count()/WRITE_COUNT<<std::endl;
```

```
  memset(page_content,0xcd,4096);
  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite write_req update take time "<<
diff.count()/OVERWRITE_COUNT<<std::endl;

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    read_content=( unsigned char *)read_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite read_req take time
"<<diff.count()/OVERWRITE_COUNT<<std::endl;
  printf("the page should fill with paten 0xcd, 0x%x\n", read_content[0]);

  start = std::chrono::steady_clock::now();
  for(i=OVERWRITE_COUNT;i<WRITE_COUNT;i++) {
    read_content=( unsigned char *)read_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite->write count read_req take time
"<<diff.count()/(WRITE_COUNT-OVERWRITE_COUNT)<<std::endl;
  printf("the page should fill with patern 0xab, 0x%x\n", read_content[0]);

  pmemblk_close(pbp);
  return 0;
}
```

编译“g++ cbs_req_pmemblk.cpp -o cbs_req_pmemblk -lpmemblk -O2”，然后跑“taskset -c 2 ./cbs_req_pmemblk”，第一次写 4k 是 4.2us，第二次写是 3us，读是 1.35us.

```
➜  ~ taskset -c 2 ./cbs_req_pmemblk
file holds 26188559 elementscbs_init time0.0573344
write_req time4.19133e-06
overwrite write_req update take time 2.99101e-06
overwrite read_req take time 1.2958e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 1.348e-06
the page should fill with patern 0xab, 0xab
➜  ~ taskset -c 2 ./cbs_req_pmemblk
file holds 26188559 elementscbs_init time0.00895083
write_req time2.99173e-06
overwrite write_req update take time 2.92603e-06
overwrite read_req take time 1.27759e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 1.34219e-06
the page should fill with patern 0xab, 0xab
```

pmemkv

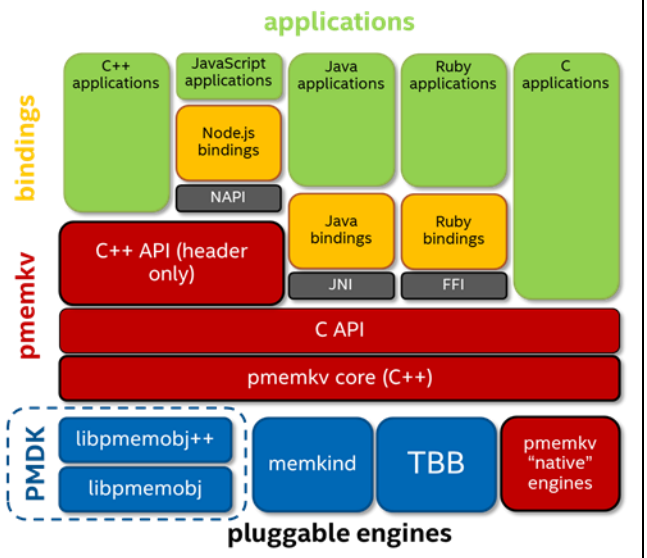
在云环境中，平台通常是虚拟化的，且应用程序高度抽象化，以避免对底层硬件细节作出显式假定。问题是：如果物理设备仅与特定服务器保持本地连接，那么如何在云原生环境中简化持久内存编程呢？

其中一个解决方法是键值(KV)存储。这种数据存储方式旨在用简单易用的 API 存储、检索和管理相关数据。市面上有许多键值数据存储解决方案。它们的特性各不相同，其 API 也是针对不同的用例而设计，但它们的核心 API 保持不变，它们全都可以提供 put、get、remove、exists、open、close 等方法。KV 引擎的底层数据结构决定了 key 的组织方式，或 value 的索引方式，这些数据结构将直接影响到 KV 引擎在不同工作负载下的性能，如点查询（ point query） 较多时，hash-based index 会比较合适，而范围查询（ range search） 更适合采用 b+tree, skiplist 等数据结构。

如图 4 所示（165 行），pmemkv 实现基于持久内存的本地 KV 引擎，此时如何管理 value 将成为影响性能的关键。为此 pmemkv 采取了以下设计：可插拔的引擎，有些引擎是 pmemkv 实现的，有的是从外部的项目移植过来的。持久化的引擎基于 libpemobj/libpmemobj++实现；非持久化的引擎是基于 memkind，而并发是基于 intel TBB。pmemkv 的核心是基于 c++实现的，其不涉及持久内存。pmemkv 的原生 API 是 C 和 C++。还提供其他编程语言绑定，如 JavaScript、Java, 和 Ruby，也可以轻松添加其他语言。pmemkv 基于各种数据结构，已经实现了非常多的 kv 引擎，我们非常希望大家可以贡献自己的引擎,或者使用我们的 pmemkv 作为你 kv 引擎的参考设计。 现有的引擎有这么几种类型和分类：

1. persistent 和 volatile （持久和易失），持久化引擎可以快速恢复数据，volatile 性能会更好。
2. 排序或者不排序， 排序可以时候 range scan 而不排序适合点查。
3. 并发或者单线程。

从支持的数据结构来看，我们有 concurrent hashmap/ sorted hashmap/ b+树/concurrent skiplist/radix 等等。你可以直接使用 pmemkv，也可以获得灵感来做为您的 kv 引擎的参考。 我们非常希望收到客户的使用反馈帮助我们进一步去支持更多的数据结构，更多的 kv 设计。



Engine name	Description	Experimental	Persistent	Concurrent	Sorted
blackhole	Accepts everything, returns nothing	No	-	-	-
vsmap	Volatile sorted hash map	No	No	No	Yes
vcmap	Volatile concurrent hash map	No	No	Yes	No
cmap	Concurrent hash map	No	Yes	Yes	No
tree3	Persistent B+ tree	Yes	Yes	No	No
stree	Sorted persistent B+ tree	Yes	Yes	No	Yes
csmap	Persistent sorted map based on concurrent skiplist (blocking erase)	Yes	Yes	Yes	Yes
radix	Persistent sorted single-threaded prefix tree	Yes	Yes	No	Yes

图 4 pmemkv 架构和支持的 kv 引擎

pmemkv 支持的引擎有很多，我们选择 cmap 来将持久内存作为 SSD 持久缓存的，其中 key 就是 req_id，而 value 就是一个 4K 的 page。我们可以观察单线程下，使用 cmap 的性能。示例 4 pmemkv 实现持久化页缓存使用的核心接口如下：

6. pmemkv_config *cfg = pmemkv_config_new();
7. pmemkv_config_put_string(cfg, "path", path)
8. pmemkv_config_put_uint64(cfg, "force_create", fcreate)
9. pmemkv_config_put_uint64(cfg, "size", size)
10. pmemkv_open("cmap", cfg, &db);
11. pmemkv_put(db, str, strlen(str), (const char *)content, PAGE_SIZE);
12. pmemkv_get(db, str, strlen(str),pmemkv_get_value, &val);

示例 4 pmemkv 实现持久化页缓存

```
#include <iostream>
#include <chrono>
#include <stdio.h>
#include <unistd.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <libpmemkv.h>
#include <string>
#include <cassert>

#define PAGE_SIZE 4096
#define PMEM_SIZE 100*1024*1024*1024UL

pmemkv_db *db=NULL;

pmemkv_config* config_setup(const char* path, const uint64_t fcreate, const uint64_t
size)
{
  pmemkv_config *cfg = pmemkv_config_new();
```

```
  assert(cfg != nullptr);

  if (pmemkv_config_put_string(cfg, "path", path) != PMEMKV_STATUS_OK) {
    fprintf(stderr, "%s", pmemkv_errormsg());
    return NULL;
  }

  if (pmemkv_config_put_uint64(cfg, "force_create", fcreate) != PMEMKV_STATUS_OK) {
    fprintf(stderr, "%s", pmemkv_errormsg());
    return NULL;
  }

  if (pmemkv_config_put_uint64(cfg, "size", size) != PMEMKV_STATUS_OK) {
    fprintf(stderr, "%s", pmemkv_errormsg());
    return NULL;
  }

  return cfg;
}

//初始化，传入的是文件的名称，文件名称也可以定义在头文件中间。
int cbs_init(const char * filename)
{
  int s;
  pmemkv_config *cfg=NULL;

  //这个地方关于 force_create 的行为不是很符合客户的使用习惯，我们将重新定义 create_if_missing
这个 option， 详细的请参考 https://github.com/pmem/pmemkv/issues/576
  if(access(filename,F_OK)==0) {
    cfg = config_setup(filename, 0,PMEM_SIZE);
  } else {
    cfg = config_setup(filename, 1,PMEM_SIZE);
  }
  assert(cfg != NULL);

  //cmap open with the cfg.
  s = pmemkv_open("cmap", cfg, &db);
  assert(s == PMEMKV_STATUS_OK);
  assert(db != NULL);
  return 0;
}

//写入或者更新一个页，输入 req_id 表示更新到什么地方，content 表示更新的内容
int write_req(uint64_t req_id, unsigned char * content) {
  int s;
  char str[20];
  //itoa(req_id,string, 10);
  sprintf(str,"%ld",req_id);
  s = pmemkv_put(db, str, strlen(str), (const char *)content, PAGE_SIZE);
  assert(s==PMEMKV_STATUS_OK);
  return 0;
}

size_t get_key_count()
{
  size_t cnt;
  int s;
  s=pmemkv_count_all(db,&cnt);
  assert(s == PMEMKV_STATUS_OK);
  return cnt;
}
```

```
void pmemkv_get_value(const char *value, size_t valuebytes, void *val) {
  //printf("0x%x\n",value[0]);
  //printf("%ld\n",valuebytes);
  //*(unsigned char **) val=(unsigned char *)value;
  //call back function mainly for atomic? that means out of the call back function,
you might not get the real value content or might changed by other thread?
  memcpy((unsigned char *)val,value,valuebytes);
  return;
}

unsigned char * read_req(uint64_t req_id,unsigned char * val ) {
  int s;
  char str[20];
  sprintf(str,"%ld",req_id);
  //val is a heap with 4096.
  s = pmemkv_get(db, str, strlen(str),pmemkv_get_value, val);
  assert(s == PMEMKV_STATUS_OK);
  return val;
}

void delete_page(uint64_t req_id) {
  return;
}

#define WRITE_COUNT 100000
#define OVERWRITE_COUNT 10000
int main()
{
  // calculate the time
  unsigned char * page_content=(unsigned char *)malloc(4096);
  uint64_t i=0;
  auto start=std::chrono::steady_clock::now();
  auto stop=std::chrono::steady_clock::now();
  std::chrono::duration<double> diff=stop-start;

  unsigned char * read_content;
  memset(page_content,0xab,4096);

  start=std::chrono::steady_clock::now();
  cbs_init("/mnt/pmem0/cbs_pmemkv");
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"cbs_init time"<<diff.count()<<std::endl;
  size_t cnt=get_key_count();
  std::cout<<"kv count"<<cnt<<std::endl;

  start = std::chrono::steady_clock::now();
  for(i=0;i<WRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"write_req time"<<diff.count()/WRITE_COUNT<<std::endl;

  memset(page_content,0xcd,4096);

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
```

```
  std::cout<<"overwrite write_req update take time "<<
diff.count()/OVERWRITE_COUNT<<std::endl;

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    read_content=read_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite read_req take time
"<<diff.count()/OVERWRITE_COUNT<<std::endl;
  printf("the page should fill with paten 0xcd, 0x%x\n", page_content[0]);

  start = std::chrono::steady_clock::now();
  for(i=OVERWRITE_COUNT;i<WRITE_COUNT;i++) {
    read_content=read_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite->write count read_req take time
"<<diff.count()/(WRITE_COUNT-OVERWRITE_COUNT)<<std::endl;
  printf("the page should fill with patern 0xab, 0x%x\n", page_content[0]);

  free(page_content);
  //start = std::chrono::steady_clock::now();
  //for(i=0;i<WRITE_COUNT;i++) {
  //  delete_page(i);
  //}
  //stop=std::chrono::steady_clock::now();
  //diff=stop-start;
  //std::cout<<"delete write count take time "<<diff.count()/WRITE_COUNT<<std::endl;
  return 0;
}
```

编译"g++ cbs_req_pmemkv.cpp -o cbs_req_pmemkv -lpmemkv -O2",然后使用"taskset -c 2 ./cbs_req_pmemkv 运行这段代码，我们看到第一次写 4KB 页大概消耗 45us，第二次大概消耗 10us，恢复的过程大概消耗 0.07s；可以使用"pmempool info -O /mnt/pmem0/cbs_pmemkv|more"来观察对象空间的大小。

```
➜  ~ taskset -c 2 ./cbs_req_pmemkv
cbs_init time0.104486
kv count0
write_req time 4.53356e-05
overwrite write_req update take time 1.2625e-05
overwrite read_req take time 2.17473e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 3.36632e-06
the page should fill with patern 0xab, 0xab
➜  ~ taskset -c 2 ./cbs_req_pmemkv
cbs_init time0.0789432
kv count100000
write_req time 9.77862e-06
overwrite write_req update take time 9.67322e-06
overwrite read_req take time 2.1298e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 2.09206e-06
the page should fill with patern 0xab, 0xab
```

Call For Action:

1. https://github.com/pmem/pmemkv/tree/master/examples

2. https://pmem.io/pmemkv/
3. 自己开发一个简单的 KV 引擎集成到 pmemkv 这个框架中。

Tips:

1. pmemkv 仅关注 pmem 的键值存储接口。应用程序级别的性能主要取决于 pmemkv 的算法。有很多优化机会。
2. pmemkv 当前仅使用 PMEM，DRAM 没有完全使用，这可能会导致性能不够好。正在进行一些优化工作。
3. 借助 libpmemobj 支持，在 pmemkv 上的恢复性能非常好。某些引擎基于 libpmemobj，因此它具有与 libpmemobj 相同的局限性。

memkind

memkind 是用户可扩展堆管理器，构建在 jemalloc 之上，它可以控制各种内存之间的堆分区。由于在各种系统、环境中都需要分配内存，并且没有统一的标准，所以 memkind 为了实现统一的函数分配而生：http://memkind.github.io/memkind/memkind_arch_20150318.pdf。memkind_create_pmem()采用 tmpfile 函数创建，在创建的目录中不会显示，并且当程序退出后创建的文件也会释放/删除。并且 memkind_create_pmem()函数调用后，并不会立即创建文件，而是在 memkind_pmem_mmap()函数调用后才会创建。当首次更新一个 memkind 分配的地址，会产生一个缺页中断，这个缺页中断的代价较大，其目的是为了建立虚拟地址到物理地址的连接。而 memkind_free 函数可能会触发这个中断。而使用环境变量“export MEMKIND_HOG_MEMORY=1”可以避免这个中断，从而不会再次产生缺页中断。如何通过预先访问避免第一次访问的缺页中断代价，现在仍然在讨论中间。

```
bool pmem_extent_dalloc(extent_hooks_t *extent_hooks,
                        void *addr,
                        size_t size,
                        bool committed,
                        unsigned arena_ind)
{
    bool result = true;
    if (memkind_get_hog_memory()) {
        return result;
    }

    // if madvise fail, it means that addr isn't mapped shared (doesn't come from
pmem)
    // and it should be also unmapped to avoid space exhaustion when calling large
number of
    // operations like memkind_create_pmem and memkind_destroy_kind
    errno = 0;
    int status = madvise(addr, size, MADV_REMOVE);
    ...
}
```

示例 5 中所使用的 memkind 的核心代码如下：

1. int err = memkind_create_pmem(filename, PMEM_SIZE, &pmem_kind);
2. disk=(disk_t *) memkind_calloc(MEMKIND_DEFAULT,1,sizeof(disk_t));

3. page = (unsigned char *)memkind_malloc(pmem_kind, PAGE_SIZE);
4. pmem_memcpy_nodrain(page,content,PAGE_SIZE);
5. memkind_free(pmem_kind,page);
6. memkind_free(MEMKIND_DEFAULT,disk);
7. memkind_pmem_destroy (pmem_kind);

示例 5 持久内存实现易失的页缓存

```
#include <iostream>
#include <chrono>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <libpmem.h>
#include <memkind.h>
#include <time.h>
#include <string.h>
#include <unistd.h>

#define PAGE_SIZE 4096
#define BLOCK_PAGE_NUM 1024
#define REQ_BLOCK 10
#define REQ_PAGE 0x3ff
#define DISK_BLOCK_NUM 1048576 //1024*1024 blocks
#define PMEM_SIZE 100*1024*1024*1024UL

/***************************************
page are in the PMEM (allocated from the memkind)
 */
typedef struct block_page {
  unsigned char * page;
}block_page_t;
typedef struct block_data {
  block_page_t pages[BLOCK_PAGE_NUM];
  uint32_t cached_pages_cnt;  //if the count over one threshold, might need to write
back to the SSD.
}block_data_t;

typedef struct disk_s {
  block_data_t blocks[DISK_BLOCK_NUM];
}disk_t;

disk_t * disk;
struct memkind *pmem_kind;

//初始化，传入的是文件的名称，文件名称也可以定义在头文件中间。
int cbs_init(const char * filename)
{
  int i;
  int err = memkind_create_pmem(filename, PMEM_SIZE, &pmem_kind);
  if (err) {
    perror("memkind_create_pmem()");
    return 1;
  }
  disk=(disk_t *) memkind_calloc(MEMKIND_DEFAULT,1,sizeof(disk_t));
  if(disk == NULL) {
    return 1;
  }
  return 0;
}
```

```c
int get_cached_count()
{
  int i;
  int cnt=0;
  for(i=0;i<DISK_BLOCK_NUM;i++) {
    cnt+=disk->blocks[i].cached_pages_cnt;
  }
  return cnt;
}

//写入或者更新一个页，输入 req_id 表示更新到什么地方，content 表示更新的内容
int write_req(uint64_t req_id, unsigned char * content) {
  //req_id>>10 是 block_id(1024 pages/block); req_id&0x3ff 是该 block 中的 page_id.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;

  if(block_id > DISK_BLOCK_NUM) {
    printf(" write req_id is not valid and over the disk range\n");
    return -1;
  }
  unsigned char * page=disk->blocks[block_id].pages[req_page_id].page;
  if(page == NULL) {
    page = (unsigned char *)memkind_malloc(pmem_kind, PAGE_SIZE);
    disk->blocks[block_id].pages[req_page_id].page=page;
  }
  if (page != NULL) {
    pmem_memcpy_nodrain(page,content,PAGE_SIZE);
    //memcpy(page,content,PAGE_SIZE);
  }
  return 0;
}

void * read_req(uint64_t req_id) {
  // req_id and content; only after the init success, then this API can be called.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;
  if(block_id > DISK_BLOCK_NUM) {
    printf("read req_id is not valid and over the disk range\n");
    return NULL;
  }

  return disk->blocks[block_id].pages[req_page_id].page;
}

void delete_page(uint64_t req_id) {
  // req_id and content; only after the init success, then this API can be called.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;

  if(block_id > DISK_BLOCK_NUM) {
    printf(" write req_id is not valid and over the disk range\n");
    return;
  }

  unsigned char * page=disk->blocks[block_id].pages[req_page_id].page;

  if (page != NULL) {
    memkind_free(pmem_kind,page);
    disk->blocks[block_id].pages[req_page_id].page=NULL;
  }
}
```

```
#define WRITE_COUNT 100000
#define OVERWRITE_COUNT 10000
int main()
{
  // calculate the time
  unsigned char * page_content=(unsigned char *)malloc(4096);
  uint64_t i=0;
  auto start=std::chrono::steady_clock::now();
  auto stop=std::chrono::steady_clock::now();
  std::chrono::duration<double> diff=stop-start;

  char * read_content;
  memset(page_content,0xab,4096);

  start=std::chrono::steady_clock::now();
  cbs_init("/mnt/pmem0");
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"cbs_init time "<<diff.count()<<std::endl;
  std::cout<<"cached page count" << get_cached_count()<<std::endl;

  start = std::chrono::steady_clock::now();
  for(i=0;i<WRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"write_req time "<<diff.count()/WRITE_COUNT<<std::endl;

  memset(page_content,0xcd,4096);

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite write_req update take time "<<
diff.count()/OVERWRITE_COUNT<<std::endl;

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    read_content=(char *)read_req(i);
    memcpy(page_content,read_content,PAGE_SIZE);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite read_req take time
"<<diff.count()/OVERWRITE_COUNT<<std::endl;
  printf("the page should fill with paten 0xcd, 0x%x\n", read_content[0]);

  start = std::chrono::steady_clock::now();
  for(i=OVERWRITE_COUNT;i<WRITE_COUNT;i++) {
    read_content=(char *)read_req(i);
    memcpy(page_content,read_content,PAGE_SIZE);

  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite->write count read_req take time
"<<diff.count()/(WRITE_COUNT-OVERWRITE_COUNT)<<std::endl;
  printf("the page should fill with patern 0xab, 0x%x\n", read_content[0]);
```

```
  //start = std::chrono::steady_clock::now();
  //for(i=0;i<WRITE_COUNT;i++) {
  //  delete_page(i);
  //}
  //stop=std::chrono::steady_clock::now();
  //diff=stop-start;
  //std::cout<<"delete write count take time "<<diff.count()/WRITE_COUNT<<std::endl;
  memkind_free(MEMKIND_DEFAULT,disk);
  memkind_pmem_destroy (pmem_kind);

  return 0;
}
```

使用“g++ cbs_req_memkind.cpp -o cbs_req_memkind -lmemkind -lpmem -O2”编译后运行“taskset -c 2 ./cbs_req_memkind”得到的性能，第一次分配和写入 4K 页需要 5.25us,而第二次的更新写入只需要 958 ns 不到 1us：

```
➜  ~ taskset -c 2 ./cbs_req_memkind
cbs_init time 0.0954005
cached page count0
write_req time 5.25884e-06
overwrite write_req update take time 9.58645e-07
overwrite read_req take time 1.14597e-06
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 1.08656e-06
the page should fill with patern 0xab, 0xab
```

memkind 还可以管理 numa 节点上内存的分配，如果你有多个 numa 节点可以使用 memkind 来进行分配。其中持久内存可以配置成为系统中的 numa 节点。Kernel5.1 以上会支持，这样在系统中可以使用 MEMKIND_DAX_KMEM 的静态类型（static kinds）来直接访问持久内存。性能应该和上述示例没有区别。如果你有兴趣可以尝试编写这样的示例。

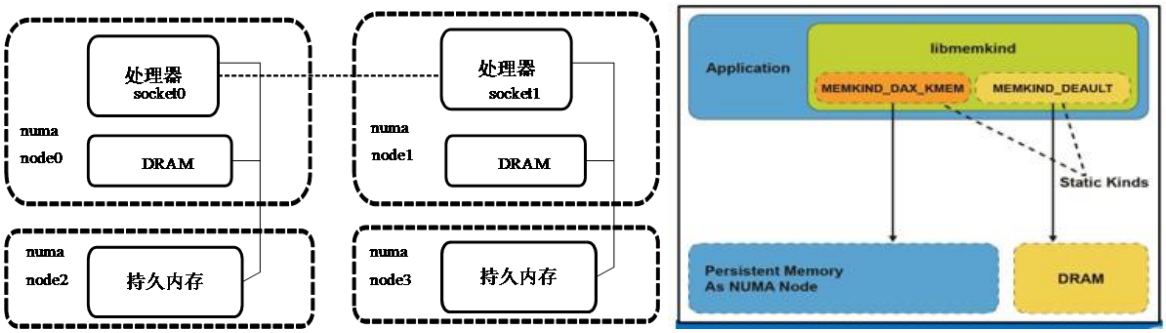


图 5 持久内存配置为系统的 numa 节点

由于有多个 kind 类型的内存使用 memkind 来进行分配，在释放这些内存时，我们很难判断这个内存来自于哪一个 kind，所以在 memkind 中可以支持使用无类型的释放 memkind_free(NULL,page); memkind 会自动找出类型并去释放，这个会带来一定的性能代价。

Call For Action:

1. https://github.com/memkind/memkind
2. http://memkind.github.io/memkind/

Tips:

1. 应用需要自己来决定那些空间放在 DRAM 中，那些放在 PMEM 中。
2. 注意环境变量“export MEMKIND_HOG_MEMORY=1”的使用，如果 page fault 的影响非常大，可以使用 export MEMKIND_HOG_MEMORY=1。

The baseline store on DRAM and SSD

为了让大家对持久内存编程方法和性能方面有一个更加直观的感受，我们将上述示例要求的 4K 页写入 DRAM 和写入普通的 SSD。写入 DRAM 的性能我们可以参考最后的 memkind 示例即可，将 pmem_kind 改为 MEMKIND_DEFAULT，或者将 memkind 的接口换成 glibc 的接口就可以了。

```
#include <iostream>
#include <chrono>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
//#include <libpmem.h>
#include <memkind.h>
#include <time.h>
#include <string.h>
#include <unistd.h>

#define PAGE_SIZE 4096
#define BLOCK_PAGE_NUM 1024
#define REQ_BLOCK 10
#define REQ_PAGE 0x3ff
#define DISK_BLOCK_NUM 1048576 //1024*1024 blocks
//#define PMEM_SIZE 100*1024*1024*1024UL


/***************************************
page are in the PMEM (allocated from the memkind)
 */
typedef struct block_page {
  unsigned char * page;
}block_page_t;
typedef struct block_data {
  block_page_t pages[BLOCK_PAGE_NUM];
  uint32_t cached_pages_cnt;  //if the count over one threshold, might need to write
back to the SSD.
}block_data_t;

typedef struct disk_s {
  block_data_t blocks[DISK_BLOCK_NUM];
}disk_t;

disk_t * disk;
//struct memkind *pmem_kind;

//初始化，传入的是文件的名称，文件名称也可以定义在头文件中间。
int cbs_init(const char * filename)
{
  int i;
  //int err = memkind_create_pmem(filename, PMEM_SIZE, &pmem_kind);
  //if (err) {
```

```
  //    perror("memkind_create_pmem()");
  //    return 1;
  //}
  disk=(disk_t *) memkind_calloc(MEMKIND_DEFAULT,1,sizeof(disk_t));
  if(disk == NULL) {
    return 1;
  }
  return 0;
}

int get_cached_count()
{
  int i;
  int cnt=0;
  for(i=0;i<DISK_BLOCK_NUM;i++) {
    cnt+=disk->blocks[i].cached_pages_cnt;
  }
  return cnt;
}

//写入或者更新一个页，输入 req_id 表示更新到什么地方，content 表示更新的内容
int write_req(uint64_t req_id, unsigned char * content) {
  //req_id>>10 是 block_id(1024 pages/block); req_id&0x3ff 是该 block 中的 page_id.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;

  if(block_id > DISK_BLOCK_NUM) {
    printf(" write req_id is not valid and over the disk range\n");
    return -1;
  }
  unsigned char * page=disk->blocks[block_id].pages[req_page_id].page;
  if(page == NULL) {
    page = (unsigned char *)memkind_malloc(MEMKIND_DEFAULT, PAGE_SIZE);
    disk->blocks[block_id].pages[req_page_id].page=page;
  }
  if (page != NULL) {
    //pmem_memcpy_nodrain(page,content,PAGE_SIZE);
    memcpy(page,content,PAGE_SIZE);
  }
  return 0;
}

void * read_req(uint64_t req_id) {
  // req_id and content; only after the init success, then this API can be called.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;
  if(block_id > DISK_BLOCK_NUM) {
    printf("read req_id is not valid and over the disk range\n");
    return NULL;
  }

  return disk->blocks[block_id].pages[req_page_id].page;
}

void delete_page(uint64_t req_id) {
  // req_id and content; only after the init success, then this API can be called.
  uint64_t block_id = req_id >> REQ_BLOCK;
  uint64_t req_page_id = req_id & REQ_PAGE;

  if(block_id > DISK_BLOCK_NUM) {
    printf(" write req_id is not valid and over the disk range\n");
    return;
```

```
  }

  unsigned char * page=disk->blocks[block_id].pages[req_page_id].page;

  if (page != NULL) {
    memkind_free(MEMKIND_DEFAULT,page);
    disk->blocks[block_id].pages[req_page_id].page=NULL;
  }
}

#define WRITE_COUNT 100000
#define OVERWRITE_COUNT 10000
int main()
{
  // calculate the time
  unsigned char * page_content=(unsigned char *)malloc(4096);
  uint64_t i=0;
  auto start=std::chrono::steady_clock::now();
  auto stop=std::chrono::steady_clock::now();
  std::chrono::duration<double> diff=stop-start;

  char * read_content;
  memset(page_content,0xab,4096);

  start=std::chrono::steady_clock::now();
  cbs_init("/mnt/pmem0");
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"cbs_init time "<<diff.count()<<std::endl;
  std::cout<<"cached page count" << get_cached_count()<<std::endl;

  start = std::chrono::steady_clock::now();
  for(i=0;i<WRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"write_req time "<<diff.count()/WRITE_COUNT<<std::endl;

  memset(page_content,0xcd,4096);

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite write_req update take time "<<
diff.count()/OVERWRITE_COUNT<<std::endl;

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    read_content=(char *)read_req(i);
    memcpy(page_content,read_content,PAGE_SIZE);
    assert (page_content[0]==0xcd);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite read_req take time
"<<diff.count()/OVERWRITE_COUNT<<std::endl;
  printf("the page should fill with paten 0xcd, 0x%x\n", page_content[0]);
```

```
  start = std::chrono::steady_clock::now();
  for(i=OVERWRITE_COUNT;i<WRITE_COUNT;i++) {
    read_content=(char *)read_req(i);
    memcpy(page_content,read_content,PAGE_SIZE);
    assert(page_content[0]==0xab);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite->write count read_req take time
"<<diff.count()/(WRITE_COUNT-OVERWRITE_COUNT)<<std::endl;
  printf("the page should fill with patern 0xab, 0x%x\n", page_content[0]);

  for(i=0;i<WRITE_COUNT;i++) {
    delete_page(i);
  }
  ////sleep(10);
  start = std::chrono::steady_clock::now();
  for(i=0;i<WRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"write_req time "<<diff.count()/WRITE_COUNT<<std::endl;

  memset(page_content,0xcd,4096);

  start = std::chrono::steady_clock::now();
  for(i=0;i<OVERWRITE_COUNT;i++) {
    write_req(i,page_content);
  }
  stop=std::chrono::steady_clock::now();
  diff=stop-start;
  std::cout<<"overwrite write_req update take time "<<
diff.count()/OVERWRITE_COUNT<<std::endl;

  //start = std::chrono::steady_clock::now();
  //for(i=0;i<WRITE_COUNT;i++) {
  //  delete_page(i);
  //}
  //stop=std::chrono::steady_clock::now();
  //diff=stop-start;
  //std::cout<<"delete write count take time "<<diff.count()/WRITE_COUNT<<std::endl;
  return 0;
}
```

使用 memkind 去分配 DRAM，通过编译” g++ cbs_req_dram.cpp -o cbs_req_dram -lmemkind -O2”，然后测试得到的结果，第一次的写由于有 page fault，所以需要 1.5us，但是第二次只需要 696ns：读的性能大概在 253ns~331ns。

```
➜  ~ taskset -c 2 ./cbs_req_dram
cbs_init time 0.00107337
cached page count0
write_req time 1.50446e-06
overwrite write_req update take time 6.9639e-07
overwrite read_req take time 2.53707e-07
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 3.31147e-07
the page should fill with patern 0xab, 0xab
write_req time 1.30782e-06
overwrite write_req update take time 6.96456e-07
```

而如果将 4k 页已经相应的 meta data 写到 SSD，我们需要保证将 4k 页还有相应的 meta data 保存到 ssd 中。我们可以使用 libpmem 示例的方法来保存数据到 SSD 中并从 SSD 中间恢复数据。Libpmem 示例的方法是使用了 pmem_memcpy 的方法将要持久化的数据写入 PMEM，而我们可以使用 pmem_msync 的方法将数据写入到 SSD。在示例 1 的基础上做这些改动：

1. 将 pmem_map_file(filename,PMEM_SIZE, PMEM_FILE_CREATE, 0666, &mapped_len, &is_pmem), filename 指向在 ssd 上的文件，同时 is_pmem 将会是 false，将 is_pmem 设置为全局变量。
2. 在使用 pmem_memcpy_persist;pmem_persist;pmem_memset_persistent 的地方使用类似下面的操作：

```
if(is_pmem) {
  pmem_memcpy_persist(PAGE_FROM_META(page_new), content, 4096);
} else {
  memcpy(PAGE_FROM_META(page_new),content,4096);
  pmem_msync(PAGE_FROM_META(page_new),4096);
}
```

这样就可以完成将数据写入 SSD 的过程，当然这个过程中 meta 的写入。为了避免 page cache 的影响，在测试之前我们会先将 page cache drop 掉“echo 3 > /proc/sys/vm/drop_caches”，这样第一次必须从 SSD 介质中去读数据，所以性能大概是在 23us~30us；写的性能大概在 430us; 而第二次读由于已经在 page cache 中， 此时读是 364ns~390ns,和 DRAM 的性能几乎相当。Ssd 可以保证一个扇区（sector）的断电原子性，所以在这个里面 8 字节应该是可以保证原子性的。

```
➜  ~ echo 3 > /proc/sys/vm/drop_caches
➜   ~ taskset -c 2 ./cbs_req_ssd
pmem_map_file mapped_len=107374182400, is_pmem=0
init done, pagecache_num=23588351,free page number=23588351
cbs_init time 107.153
cached page count0
overwrite read_req take time 7.109e-09
the page should fill with paten 0xcd, 0x20
overwrite->write count read_req take time 7.03638e-09
the page should fill with patern 0xab, 0x20
write_req time 0.000269097
overwrite write_req update take time 0.000282651
overwrite read_req take time 4.84718e-07
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 5.37396e-07
the page should fill with patern 0xab, 0xab
➜  ~ echo 3 > /proc/sys/vm/drop_caches
➜   ~ taskset -c 2 ./cbs_req_ssd
pmem_map_file mapped_len=107374182400, is_pmem=0
init done, pagecache_num=23588351,free page number=23488351
cbs_init time 6.6092
cached page count100000
overwrite read_req take time 2.31789e-05
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 3.03323e-05
the page should fill with patern 0xab, 0xab
write_req time 0.000439356
overwrite write_req update take time 0.000431005
overwrite read_req take time 3.64774e-07
```

```
the page should fill with paten 0xcd, 0xcd
overwrite->write count read_req take time 3.90219e-07
the page should fill with patern 0xab, 0xab
```

思考，这个地方可以如何优化？

1. 是不是需要每次写操作都需要 sync？
2. Metadata 的设计可不可以简化，是否直接将 req_id 转成文件的 offset 并写入就可以了?是否使用 CRC 来检查数据完整性和一致性？为了简化，我们只考虑将 4K 页数据写入 SSD（当然 SSD 的型号，容量大小等等可能对性能有影响，我们用 P4510 为例），不考虑任何的 meta data。

Summary

我们使用了 libpmem，libpmemobj，pmemkv，以及 memkind 来实现几乎相同的功能，把持久内存当成页缓存，所不同的是 memkind 是易失的。您可以把这些代码在您的持久内存的服务器上编译并运行，从而深入的了解持久内存编程的方法和需要注意的地方，因为应用的功能和性能完全取决于你对持久内存的理解和编程的方法。现在我们就这几种实现的异同，从代码量、性能、实现的难度、还有空间的额外开销等方面来比较这些解决方案，如表 1 所示：就这个场景而言，我我们可以推荐使用 libpmem 或者 libpmemblk。

表 1 libpmem，libpmemob， pmemkv 及 memkind 实现 4K 页缓存比较

	persistent	Code line	Easy to impliment	first 4k write (us)	Sec 4k write(us)	Read 4k addr(us)*	Meta data space bytes/4K Page
libpmem	Yes	400	Hard	1.5*	1.7~1.9	1.04	64*
libpmemobj	Yes	280	medium	7.34	5.6	1.05~1.1	64
libpmemblk	Yes	120	easy	4.2	2.9~3.0	1.35	?
Pmemkv(cmap)	Yes	165	easy	45	10	2.2	64
memkind	No	180	easy	5.25	0.958	1.08	NK
DRAM(memkind)	No	180	easy	1.3~1.5	0.696	0.253~0.331	NK

* libpmem meta data 64 bytes is for the cache optimization. In ICX+BPS, can change back to 8 bytes overhead.

*read will real copy the 4k page from PMEM to DRAM.

根据客户的反馈，客户需要加速 WAL 的写入，通过持久内存的使用大概有下面的一些需求：

1. Log 的大小一般可能就 100 多字节，每个 log 都必须保证持久化，也就是写入 SSD 时候必须每次保证刷新。SSD 每次都需要写入 4K 页面，所以存在严重的写放大的问题。
2. 可以使用持久内存来作为一个持久化 buffer,在背后将 log 写入 SSD。

我们的示例使用的 C 代码在 Linux 上构建与运行，可以在 github 上找到我们的代码还有跑测方法 https://github.com/guoanwu/pmem_program/blob/master/wal_accel

实验的示例：

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

uint64_t ntime_since(const struct timespec *s, const struct timespec *e)
{
        int64_t sec, nsec;

        sec = e->tv_sec - s->tv_sec;
        nsec = e->tv_nsec - s->tv_nsec;
        if (sec > 0 && nsec < 0) {
                sec--;
                nsec += 1000000000LL;
        }
        /*
         * time warp bug on some kernels?
         */
        if (sec < 0 || (sec == 0 && nsec < 0))
                return 0;

        return nsec + (sec * 1000000000LL);
}

#define LOG_NUM 10000

unsigned char * logs[LOG_NUM];
const unsigned char
allChar[63]="0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

void generateString(unsigned char* dest,const unsigned int len)
{
    unsigned int cnt,randNo;
    srand((unsigned int)time(NULL));
    for(cnt=0;cnt<len;cnt++)
    {
      randNo=rand()%62;
      *dest=allChar[randNo];
      dest++;
    }
    *dest='\0';
}

//generate LOG_NUM logs with different size from 50~250
void gen_log()
{
    int i,size;
    unsigned char * dest;
    srand((unsigned int)time(NULL));
```

```
    //generate logs
    for(i=0;i<LOG_NUM;i++)
    {
      size=rand()%(250-50+1)+50; //50~250 bytes.
      dest=(unsigned char *)malloc(size+1);
      generateString(dest, size);
      logs[i]=dest;
    }
}

int main()
{
    int fd, size;
    char *buffer;
    struct timespec start, end;
    int i=100;
    int j;

    gen_log();
    //write log, we must append write
    fd = open("/mnt/nvme0/wal.log", O_WRONLY|O_CREAT|O_APPEND);

    clock_gettime(CLOCK_REALTIME, &start);
    //1M log write into the log
    while(i--) {
      for(j=0;j<LOG_NUM;j++)
      {
         write(fd,logs[j],strlen((const char *)logs[j]));
     fdatasync(fd); //in the normal code, we need to fdatasync and with the wal accellerate, we
need to use the AOFGUARD_DISABLE_SYNC=yes to ignore this sync; the data sync is called in the
write
      }
    }
    clock_gettime(CLOCK_REALTIME, &end);
    printf("the durating for 1m logs=%ld\n",ntime_since(&start,&end));

    //fd: the file is closed
    close(fd);

    //read progress:since when open the log again, it will copy the data in the pmem to the
NVMe, so the O_RDWR must be used.
    fd = open("/mnt/nvme0/wal.log", O_RDWR);

    struct stat stat;

    fstat(fd,&stat);
    printf("fd=%d,filesize=%ld read the log from the head with the size 300
\n",fd,stat.st_size);

    //lseek(fd, 0, SEEK_SET);
    buffer=malloc(300);
    size = read(fd, buffer, 300);  //read data to a buffer
    printf("size=%d, buffer=%s, bufferlen=%ld\n",size, buffer, strlen(buffer));
    close(fd);
    free(buffer);
}
```

After compile and run and get result as:

With PMEM accelerate	w/o PMEM accelerate
➜ wal_accel git:(master) ✗ make run_acc export AOFGUARD_DISABLE_SYNC=yes export AOFGUARD_NVM_DIR=/mnt/pmem0 export AOFGUARD_NVM_SIZE_MB=512 export LD_LIBRARY_PATH=./lib export AOFGUARD_FILENAME_PATTERN=.log LD_PRELOAD=./lib/libaofguard_inject.so taskset -c 0 ./wal_acc stat.st_size=145752064, aofguard->file.fsync_len=145752064 the durating for 1m logs=2384793801 Dennis aofguard_deinit called stat.st_size=291856200, aofguard->file.fsync_len=289406976	➜ wal_accel git:(master) ✗ make run_org taskset -c 0 ./wal_acc the durating for 1m logs=161089599880 fd=4,filesize=462771276 read the log from the head with the size 300 size=300, buffer=xx, bufferlen=300

fd=5,filesize=289406976 read the log from the head with the size 300 size=300, buffer=xx, bufferlen=300	