Opportunities for Systems Optimization in the Era of Non-Volatile Memory

A Seminar Report
submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

by

Muhammed Unais P
Roll No: 164050009

Guide:

Prof. Purushottam Kulkarni

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai

Contents

List of Tables

List of Figures

1. Introduction

For last three decades, operating system and application design follows same memory hierarchy containing : high speed, volatile memory technologies like static random-access memory and dynamic random-access memory; slow and non-volatile magnetic disks for secondary storage; the flash/solid state drives with low power consumption. Since the flash devices lies in between random-access memory and magnetic disk in terms of performance and cost, they got wide spread usage for various purposes, as storage devices in mobile devices and for caching in main frame systems.

Now, the non-volatile memory technologies are emerging, and their impact on operating system and application design will be significant. Currently, spin-transfer-torque - magnetoresistive random-access memory (SST-MRAM) can provide less read latency as compared to non-volatile random-access memory (NVRAM), and will be possible to entirely replace the DRAM world. Currently all of these technologies are not available in the market, and their large scale production is not started. The capacity of those available in the market is very small, only few gigs, and their price is 8 times larger than what we expected. But in near future, NVRAM will be started producing commercially, and they will be cheaper than flash drives.

DRAM-like byte-addressability and disk-like persistency are the two important features of NVRAM. Many researches are already done in the applications of NVRAM. These researches gives an insight of what are main impact of these technologies on existing operating systems and applications. Basically, there three ways in which the NVRAM are used in application and operating systems design. Many applications and file systems are still not persistent memory aware, and hence, persistent memory is used through the block device interface. While this way can make legacy applications benefit from persistent memory, such a usage does not fully exploit the non-volatile memories. The second way is building a new persistent memory-aware file system. In this case, persistent memory is accessed through the memory interface by file systems, which further provide a block interface to applications. The layout of data structures of the file systems is specially optimized to leverage persistent memorys characteristics. Such a usage provides better usage of persistent memory in the file system layer, but a lot of application data is still not persistent memory optimized. The third one is building persistent memory-optimized applications, which can directly access persistent memory through load/store instructions. Based on the different uses of NVRAM, its impact reflect on different layers of operating systems, as a main memory, in file systems, in storage layer, etc.

Since the cloud computing is more popular, and many big data applications are migrated to cloud computing, impacts of NVRAM on virtualization are significant. Since the cloud provides started provisioning platforms for in-memory computing instance, we can envision that non-volatile random-access memory will be an indispensable part of cloud intrastate.

1.1 Challenges of Managing Non-Volatile Memory

Managing non-volatile memory like conventional memory is not a good solution either. To guarantee consistency and durability, non-volatile structures must meet a host of challenges, many of which do not exist for volatile memories. For instance, pointers from non-volatile data structures into volatile memory are inherently unsafe, because they are meaningless after the program ends. The system must also perform some kind of logging if non-volatile structures are to be robust in the face of application or system failure.

Trusting the average programmer to get it right in meeting these challenges is both unreasonable and dangerous for non-volatile data structures. An error in any of these areas will result in permanent corruption that neither restarting the application nor rebooting the system will resolve. Hence lower layer should provide interfaces to manage the non-volatile memory efficiently. The important challenges associated with managing non-volatile memory are give below.

1. Consistent and atomic updates
2. Ensuring consistency of synchronization
3. Durable memory transactions
4. Avoiding persistent memory leaks

Hybrid main memory system containing both volatile memory(DRAM) and non-volatile random-access memory(NVRAM) exhibits better performance than NVM-only systems. The pages which are frequently accessed can be placed in DRAM which avoid the higher access latency of non-volatile memory. But, memory subsystems are agnostic to the how the application manages the allocated memory. Hence the following are the main challenges of implementing hybrid memory systems with performance similar to DRAM-only systems.

1. Identifying how the application data structures are stored in the allocated memory.
2. Determining the access pattern of different data structures and pages.
3. Dynamically mapping the virtual pages to DRAM and NVRAM.

1.2 Requirements of Persistent Memory Abstraction

1. **Pointer safety**
   Interfaces should provide support for preventing the programmers by corrupting the data structures by misusing pointers or making memory allocation errors.

2. **Flexible ACID transactions**
   Updating in-memory data structures are not trivial. Writing to an in-memory data structure object requires multiple `store` operations. So the lower layer should provide abstractions for consistent updates to in-memory data. Also, the system must enable programmers to move data structures between consistent states, automatically recovering to such a state after a failure.

3. **Familiar interfaces**
   The interface or abstractions provided the lower layer should be similar to the familiar interfaces for implementing volatile data structures.

4. **High performance**
   The access to the data should as fast as possible relative to the speed of underlying non-volatile memory, and the overhead caused by the lower layers should be minimum.

5. **Scalability**
   Applications should be able to scale the in-memory data structures. The lower layer which provides the persistent memory abstractions should be able to scale the persistent memory for the applications swapping the data into secondary storage.

Table 1.1: Types of problems discussed in this seminar.

Problem	Work
Implementation of abstractions for user space programming with non-volatile memory, and providing primitives for consistently updating non-volatile memory.	*Mnemosyne* [3], *NV-Heaps* [13]
Improve performance of legacy applications by introducing in-memory persistent file systems with block interface using which applications can access non-volatile memory.	*nvramdisk* [4]
Bridging the speed gap between fast random-access memory and slow flash devices using byte-addressable non-volatile memory.	*Hibachi* [5]
Meeting the large memory requirements of big data applications using hybrid main memory containing both volatile and non-volatile memory technologies.	*X-Mem* [7], [8], [9]
Designing energy efficient hybrid main memory systems by reducing the write traffic to non-volatile memory.	*Raminate* [6]
Improving the performance of non-volatile memory in virtual machine by efficiently virtualizing non-volatile memory.	A Case for Virtualizing Persistent Memory [10]
Reducing the downtime of applications running inside virtual machine in case of power outage.	*NV-Hypervisor* [11], *Temporality* [12]

1.3 Types of Problems

There are several studies has been completed on specific parts of both of these area. Table 1.1 illustrates the different types of problems that have been address by multiple researches. The the aim of this seminar is to address the broad impacts of NVRAM in different areas of operating system and application design. To attain the goal, this report is divided to multiple chapters. The chapter 2 details the the different types of non-volatile memory technologies. The broad impacts of non-volatile memory technologies on applications and operating system design are explained in the chapter 4. The chapter 5 is dedicated to to explain the impacts of NVRAM on vitalization.

2. Background of Memory Technologies

In computing world, memory systems refers to the hierarchy of storage devices with different capacities, costs, and access times. Central Processing Unit (CPU) registers sits in the top of memory hierarchy as shown in Fig:2.1 and they holds most relevant and frequently used data. But they are very small in capacity. The next level in memory hierarchy is small and fast cache memories which are used by CPU to reduce the average cost to access the data and other instructions from relatively slower main memory. Modern CPUs have multiples layers of cache called L1, L2, and L3. The L1 caches will be very close to CPU and they varies from 2 KB to 64 KB. L2 is cache is slightly larger than L1 and thus accompanied by some delay. It varies from 256 KB to 512 KB. L3 cache is the largest among all cache and its size varies from 1 MB to 8 MB.

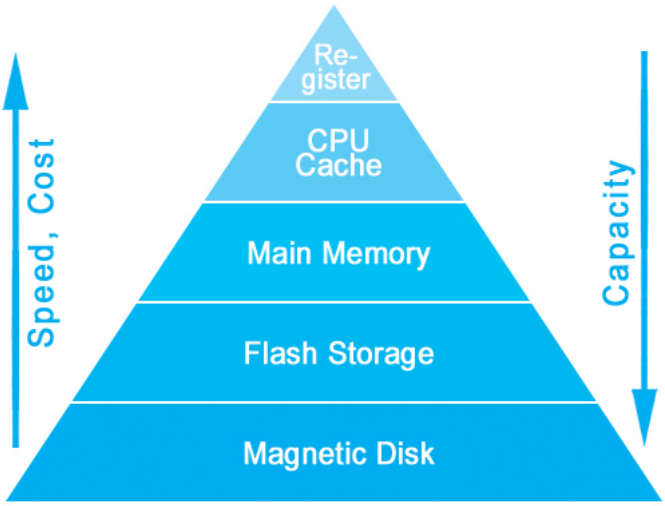

Figure 2.1: Hierarchy of memory devices

The next level in memory hierarchy is main memory. Main memory is the general purpose, relatively low cost memory used in memory hierarchy and it stages the data stored in large and slow disks.

Flash storage device lies in the next level of memory hierarchy. Flash devices are any type of drive, repository or system that uses flash memory to keep data for an extended period of time. Flash memory is ubiquitous in small computing devices and increasingly common in large business storage systems. The size and complexity of flash-based storage varies in devices ranging from portable USB flash drives, smartphones, cameras and embedded systems to enterprise-class all-flash arrays. Flash is packaged in a variety of formats for different storage purposes.

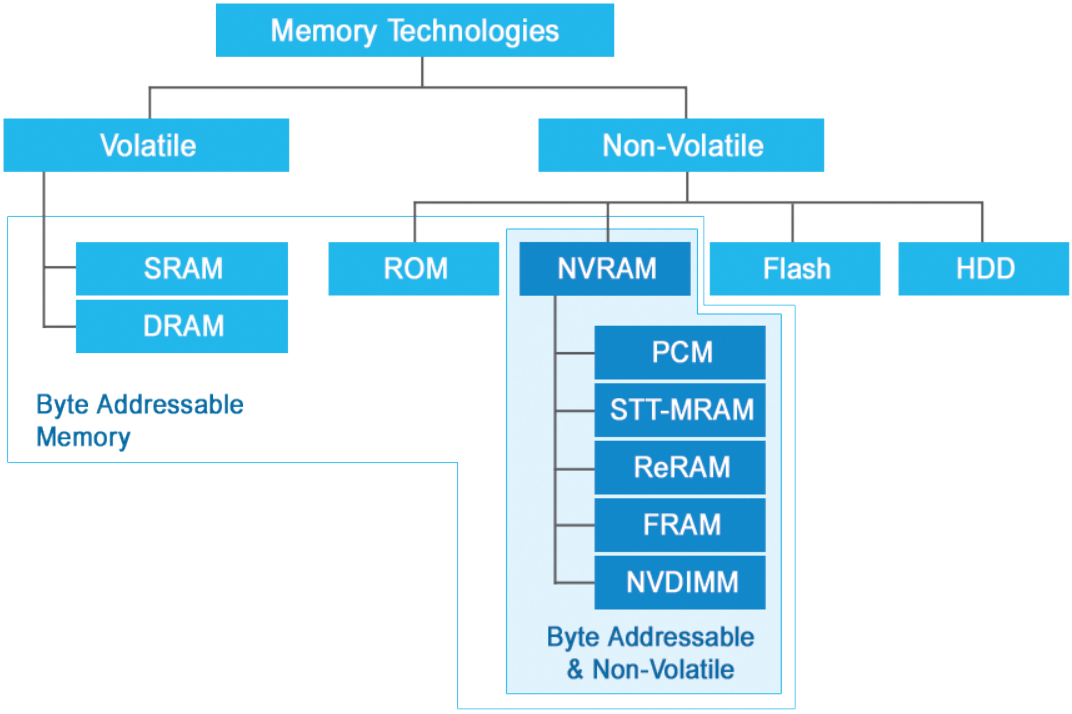

Figure 2.2: Taxonomy of memory technologies.

All the currently available memory technologies can be classified as Volatile Memory and Non Volatile Memory based on the ability to store information when power is turned off. Non volatile memory can retrieve stored information even after having been power cycled. But volatile memory requires power to maintain the stored information; it retains its contents while powered on but when the power is interrupted, the stored data is lost immediately or very rapidly. Figure 2.2 shows the taxonomy of memory technologies. Section 2.1 details the the volatile memory technologies, and section 2.2 details the non-volatile memory technologies. Finally, the section 2.3 compares the performance and the physical characteristics of different memory technologies.

2.1 Volatile Memory

Volatile memory technologies required power to retain the stored data. All the stored information will be lost immediately when the power is interrupted. Hence volatile memories are used to storing data for temporary time only. Main memory is the one of the primary example of volatile memory. Since most of the general-purpose random-access memories (RAM) are volatile, the two words RAM and volatile memory are often used interchangeably.

There are two kinds of random access memory: Static Random-Access Memory (SRAM) and Dynamic Random-Access Memory (DRAM). Each of them has its own advantages and disadvantages compared to the other because both holds data in a different ways. DRAM requires the data to be refreshed periodically in order to retain the data. SRAM does not need to be refreshed as the transistors inside would continue to hold the data as long as the power supply is not cut off.

2.1.1 Static Random-Access Memory

Static random-access memory (SRAM) can retains the data as long power is supplied, and does not require to be refresh continuously. In static RAM, each bit of memory is stored in a form of flip-flops. A flip-flop cell contains 4 or 6 transistors and never has to be refreshed. This makes static RAM significantly faster than dynamic RAM and its average access time is less than 10ns.

Advantages of static random-access memory:

- SRAM module consumes less power because it only requires a small steady current.
- Because of SRAM does not need to refresh, it is typically faster

Disadvantages of static random-access memory:

- SRAM cell takes up more space on a chip and it is less dens.
- SRAM is more expensive and harder to manufacture. A gigabyte of SRAM cache costs around $5000.

Applications of static random-access memory:

- The most common application of SRAM is to serve as cache for the processor (CPU).
- SRAM is also used as hard drive buffers.

2.1.2 Dynamic Random-Access Memory

Dynamic random-access memory (DRAM) is a type of random-access memory that stores each bit of data in a separate capacitor within an integrated circuit. The capacitor can either be charged or discharged; these two states are taken to represent the two values of a bit, conventionally called 0 and 1. Since even "nonconducting" transistors always leak a small amount, the capacitors will slowly discharge, and the information eventually fades unless the capacitor charge is refreshed periodically. Because of this refresh requirement, it is a dynamic memory as opposed to static random-access memory (SRAM) and other static types of memory.

Advantages of dynamic random-access memory

- DRAM is cheaper and densely packed as compared SRAM.
- DRAM requires less power than SRAM in an active state.

Disadvantages of dynamic random-access memory

- DRAM requires poetically refreshing the memory.
- DRAM is not salable due its physical limitations
- Relatively slow operational speed.

2.2 Non-Volatile Memory

Non-volatile memory (NVM), or non-volatile storage is a type of computer memory that can retrieve stored information even after having been power cycled (turned off and back on). Examples of non-volatile memory include read-only memory, flash memory, ferroelectric random-access memory, most types of magnetic computer storage devices (e.g. hard disk drives, solid state drives, floppy disks, and magnetic tape), optical discs, and early computer storage methods such as paper tape and punched cards. Table 2.1 presents the characteristics and performance for different non-volatile memory technologies. For sake of comparison, DRAM is also included in the table.

2.2.1 Read-Only Memory

Read-Only Memory (ROM) is "built-in" computer memory containing data that normally can only be read, not written to. It contains the programming needed to start a computer, which is essential for boot-up; it performs major input/output tasks and holds programs or software instructions. Because ROM is read-only, it cannot be changed and is permanent. There are numerous read-only memory chips located on the motherboard and a few on expansion boards. The chips are essential for the basic input/output system (BIOS), boot up, reading and writing to peripheral devices, basic data management and the software for basic processes for certain utilities.

2.2.2 Non-Volatile Random-Access Memory

Non-volatile random-access memory (NVRAM) is a random-access memory that can retain the data stored when the power is turned off (non-volatile). The word *non-volatile memory* is also widely used to refer to NVRAM. The most important properties of non-volatile random-access memory are its DRAM-like byte addressability, and disk-like persistency. This is in contrast to DRAM and SRAM, which both maintain data only for as long as power is applied.

The non-volatile memory technologies are positioned between DRAM and secondary storage (such as NAND flash), both in terms of performance and cost [1] [2]. NVRAM can provide approximately 5 times the capacity at the same cost as DRAM for less than an order of magnitude reduction in performance. The latency of different NVRAM technologies is 4 time higher than the DRAM latency, and the bandwith is 8 time lower than the DRAM bandwith. Different non-volatile memory technologies are explained below.

2.2.2.1 Phase-Change Memory

Phase-change Memory (PCM) is based on the same storage mechanism as writable CDs and DVDs, but reads them based on their changes in electrical resistance rather than changes in their optical properties. PRAM's switching time and inherent scalability make it most appealing. PRAM's temperature sensitivity is perhaps its most notable drawback, one that may require changes in the production process of manufacturers incorporating the technology.

PCM is among the most developed novel NVMs and has demonstrated promising performance (e.g., < 100 ns switching speed, $> 10^9$ cycle endurance). Although it is relatively easy for phase-change memory to meet standalone flash memory requirements, achieving both high speed and long endurance ($>10^12$ cycles) for working memories is still very challenging for PCM.

2.2.2.2 Spin-Transfer-Torque - Magnetoresistive Random-Access Memory

Spin-Transfer-Torque - Magnetoresistive Random-Access Memory (STT-MRAM) improves the writing mechanism of conventional fields witching MRAM with spin transfer torques, which is more efficient and more scalable. Hence, it is well suited for many mainstream applications, particularly as a storage technology. STT-MRAM delivers the high performance of DRAM and SRAM, has the low power and low cost of flash memory, and leverages existing CMOS manufacturing techniques and processes.

Among all non-volatile memory technologies, STT-MRAM has demonstrated the highest performance (measured by <20 ns write speed and $>10^{10}$ cycle endurance) and is the most promising candidate for embedded NVM applications. Although STT-MRAM bit cell size is much larger, STT-MRAM is still smaller than SRAM.

2.2.2.3 Resistive Random-Access-Memory

Resistive Random-Access-Memory (RRAM or ReRAM) is an non-volatile memory techology that operates by changing the resistance of a specially formulated solid dielectric material. A ReRAM device contains a component called a memristor whose resistance varies when different voltages are imposed across it. The work 'memristor' is a contraction of "memory resistor".

Higher switching speed constitutes a principal advantage of RRAM over other nonvolatile storage technologies. ReRAM technologies draws much less power than NAND flash. That makes them currently best suited for memory in sensor devices for industrial, automotive and internet of things (IoT) applications. As the cost of manufacturing for ReRAM and other memristors drops, they become competitive with NAND flash.

Trade offs exist among key ReRAM parameters, e.g., speed-retention, power-speed, endurance-retention, etc. A major challenge of ReRAM is reliability (e.g., endurance, retention), variability, and failure mechanisms. RRAM can typically achieve endurance over 10^6 cycles.

2.2.2.4 Ferroelectric Random-Access Memory

Despite the name, ferroelectric random-access memory (FRAM) technology does not use the properties of ferrous materials. Instead it uses the properties crystals of a dielectric that have a reversible electric polarization. FRAM technology has all the advantages of being a non-volatile memory along with an almost unlimited number of read-write cycles. There is also greater data reliability with ferroelectric RAM. It also offers a very low level of power consumption, making it a contender for the memory for many applications.

2.2.2.5 Non-Volatile Dual in-Line Memory Module

An non-volatile dual in-line memory module (NVDIMM) is hybrid computer memory that retains data during a service outage by integrate non-volatile NAND flash memory with dynamic random access memory (DRAM) and dedicated backup power on a single memory subsystem. The NVDIMM form factor plugs into a standard DIMM connector on a memory bus. The on-module flash memory is used exclusively to back up and restore storage in DRAM. NVDIMMs operate at or near the speed of the memory bus, but are not native plug-and-play devices. Hence, manufacturers needs to modify server motherboards and BIOS/UEFI drivers to recognize the discrete memory types presented by an NVDIMM.

The onboard DRAM controller triggers a backup process if it senses a power failure is imminent. Copy data is immediately written to the onboard flash. Once system power is restored, the NVDIMM controller reverses the process, copying data from flash back to DRAM without data loss to support a resumption of normal operations.

There are different types of NVDIMMs are available.

1. **NVDIMM-F**
   NVDIMM-F is a dual in-line memory module containing flash storage. System users will need to pair the storage DIMM alongside a traditional DRAM

2. **NVDIMM-N**
   NVDIMM-N uses uses flash storage and traditional DRAM on the same module. The computer accesses the traditional DRAM directly. In the event of a power failure, the module copies the data from the volatile traditional DRAM to the persistent flash storage, and copies it back when power is restored. It uses a small backup power source for this.

3. **NVDIMM-N is dual**
   NVDIMM-N uses DDR4 dual in-line memory module with NAND Flash storage and volatile DRAM.

4. **NVDIMM-P**
   NVDIMM-P maps NAND flash and DRAM into memory address space, and data stored in the NVDIMM-P can be accessed using normal load and write instructions. The capacity of NVDIMM-P varies from 100 giga bytes to tera bytes. NVDIMM-P provides the maximum storage capacity among the other NVDIMMs at the cost of higher access latency.

2.2.3 Solid State Drive

A solid-state drive (SSD) is a solid-state storage device that uses integrated circuit assemblies as memory to store data persistently. SSD technology primarily uses electronic interfaces compatible with traditional block input/output (I/O) hard disk drives (HDDs), which permit simple replacements in common applications. New I/O interfaces like SATA Express and M.2 have been designed to address specific requirements of the SSD technology.

SSDs have no moving mechanical components. This distinguishes them from traditional electromechanical magnetic disks such as hard disk drives (HDDs) or floppy disks, which contain spinning disks and movable read/write heads. Compared with electromechanical disks, SSDs are typically more resistant to physical shock, run silently, and have lower access time and lower latency. The consumer-grade SSDs are still roughly four times more expensive per unit of storage than consumer-grade HDD

2.3 Performance Characterization of Different Technologies

Table 2.1: Performances characteristics of different memory technologies.

-	Access Granularity	Read Latency	Write Latency	Erase Latency	Endurance
HDD	512 B	5 ms	5ms	-	$<10^{15}$
Flash/SSD	4 KB	$25\mu s$	$500\mu s$	2 ms	10^5
DRAM	64 B	50 ns	50 ns	-	$>10^{15}$
PCM	64 B	500 ns	500 ns	-	10^9
STT-MRAM	64 B	10 ns	20 ns	-	10^{10}
ReRAM	64 B	10 ns	50 ns	-	10^6
FRAM	64 B	-	10 ms	-	10^{15}
NVDIMM	64 B	$1-100\mu s$	$1-100\mu s$	-	$>10^{15}$

Table 2.1 illustrates the various properties of different memory technologies. Physical properties of these non-volatile memory technologies are encapsulated in table 2.2. DRAM, HDD, and SSD are also included to the both the tables for the sake of comparison. The table shows different properties of memory technologies like access granularity, read latency, write latency, erase latency, endurance, standby power, energy per bit, cell size, and density.

Phase change memory is the most mature emerging NVM with proven performance and would benefit from further reduction of switching power. STT-MRAM achieves the highest performance

Table 2.2: Physical characteristics of different memory technologies.

-	Standby Power	Energy per bit	Cell Size	Density
HDD	1 W	-	$(2/3)F^2$	400 GB/in^2
Flash/SSD	0	10 nJ	$4F^2$	64 GB/chip
DRAM	Refresh Power	2 pJ	$6F^2$	8 Mb/chip
PCM	0	100 pJ	$5F^2$	512 Mb/chip
STT-MRAM	0	0.022 pJ	$4F^2$	2 Mb/chip
ReRAM	0	-	$6F^2$	-
FRAM	0	2 pJ	$6F^2$	128 Mb/chip
NVDIMM	1-4 W	-	-	-

and significant R&D efforts have focused on early commercialization of it. RRAM has advantages in simplicity and potentially low-cost, but reliability needs further improvement.

The performance of memory devices can be evaluated based on scalability, speed, power, and reliability. Each of this performance metrics involves multiple subcategories. For an instance, consider the power metric. Low power devices does not means low energy consumption. Energy consumption also depends on the write and read latency. Flash memory devices have very small write energy, the but their long latency increases the power consumption. But the overall performance depends on applications. Hence the selecting the best non-volatile memory device for an application depends on the access characteristic of applications.

STT-MRAM has better read latency as compared to DRAM, but its density is very small among the other non-volatile memory devices. Phase-change memory devices requires high write energy since their write latency and energy are very large. But it provides maximum density among the other non-volatile memory devices. Similarly, ReRAM provides DRAM like read and write performance, and its density is also comparable. But, its durability is very low.

3. Implications of Non-Volatile Memory on Application Design

The emerging Non-Volatile Memory technologies promises significant amount of persistent memory available to computer systems. The byte addressability and persistency of Non-Volatile Memory technologies collapses the two different levels of memory hierarchy, *byte-addressable main memory* and *persistent storage*, into single level, *byte-addressable persistent main memory*. The performance of applications depends on the main memory available to the process and disk access characteristics. Hence if the system are equipped with non-volatile memory devices, the long latency of write requests can be reduce. It also helps to reduce the write traffic to disks.

The existing operating systems and applications are designed to work with byte addressable, volatile, low latency DRAM/SRAM technologies, and block addressable, non-volatile, high latency persistent storage devices such as magnetic disk and flash storage. So the introduction of non-volatile memory devices into memory hierarchy requires significant amount of changes in the various layers of Operating System and in the application design to exploit its features completely. The table illustrates the different ways in which non-volatile memory can be used by applications.

This chapter is dedicated to address the direct impacts of non-volatile memory on applications designs. Table 3.1 illustrates the different ways applications can make use of non-volatile memory. Each of the method has its own pros and cons. The first two types are explained in the next chapter. The impact of non-volatile memory on applications can be classified as :

1. **Explicit impact of non-volatile memory**
   Explicit impact refers to the direct effects of non-volatile memory on application design architecture because of its persistency and byte-addressability. It mainly includes the persistent virtual memory concept in which the applications can directly write to the persistent storage using load/store instructions.

2. **Implicit impact of non-volatile memory**
   Implicit impacts assumes the applications are unaware about the existence of non-volatile memory, and explain how the applications can be benefited from the non-volatile memory.

This chapter is organized as follows: The section 3.1 details the explicit impact of non-volatile memory on the application design. The next section, 3.2 illustrates the various implicit impact of non-volatile memory, a

Table 3.1: Different ways in which application can use non-volatile memory.

Method	Advantage	Disadvantage
Through block device interface	Many applications and file systems are still not PM aware, and hence, persistent memory is used through the block device interface. While this way can make legacy applications benefit from PM.	The byte addressability is not fully exploited. A large portion of PM is used for read but not write, which just treat PM as disk cache.
In-memory file systems	Persistent memory is accessed through memory interface which gives better performance. It provides block interface for applications, and hence, existing applications can make use of persistent memory.	It can provide usage of non-volatile memory in the file system layer. Applications are not persistent-memory optimized. Applications experiences delays at multiple layers including system call, virtual file systems, file system, etc.
Persistent memory optimized applications	Applications can directly access non-volatile memory using load/store instructions. Performance is restricted only by physical device properties.	Applications requires significant amount of changes. Applications designers have to worry about consistency and durability of data stored in non-volatile memory.

3.1 Explicit Impact of Non-Volatile Memory

Explicit implications refers the direct effects of non-volatile memory on applications. This section details the how to *persistent-memory-aware* applications systems. Existing operating systems and applications are designed for a strict bifurcation of devices into two types - *main memory devices* which are volatile, fast, and random byte-addressable, and *storage devices* which are persistent, slow, and block-based. The performance of these applications depends on the disk access characteristics. The latency and bandwidth of existing storage devices are slower than one order of magnitude on DRAM. The advent of non-volatile random access into memory hierarchy enables the application to make in-memory persistent data structures and avoids the continuous disk access to synchronize the dirty data.

The most obvious way to use NVRAM would be to use it as a direct replacement for DRAM. The another method is to use NVRAM with in file system as in-memory file systems, or as cache. Application does not require any changes on both of these techniques, and the legacy applications can make use of these approaches to get benefit form NVRAM. But, neither of these approaches exposes the full power of NVRAM to programmers. So operating system should expose NVRAM as a **persistent memory abstractions** to provide direct access from user space. The direct access to NVRAM reduces the latency of read and write operations by bypassing many software layers, including system calls, file systems, and device drivers.

3.1.1 Exposing Non-Volatile Memory as Persistent Region

Mnemosyne [3] exposes the NVRAM directly to application programmers through *persistent region* abstractions using which the application can directly read and write data. Mnemosyne maps the persistent regions at fixed virtual address of the process and hence only this region will be persisted across reboots. In addition, programs must take explicit steps to guarantee persistence, such as writing data with special instructions or within a transaction to ensure data makes it all the way to non-volatile memory. Figure 3.1 shows architecture of Mnemosyne.

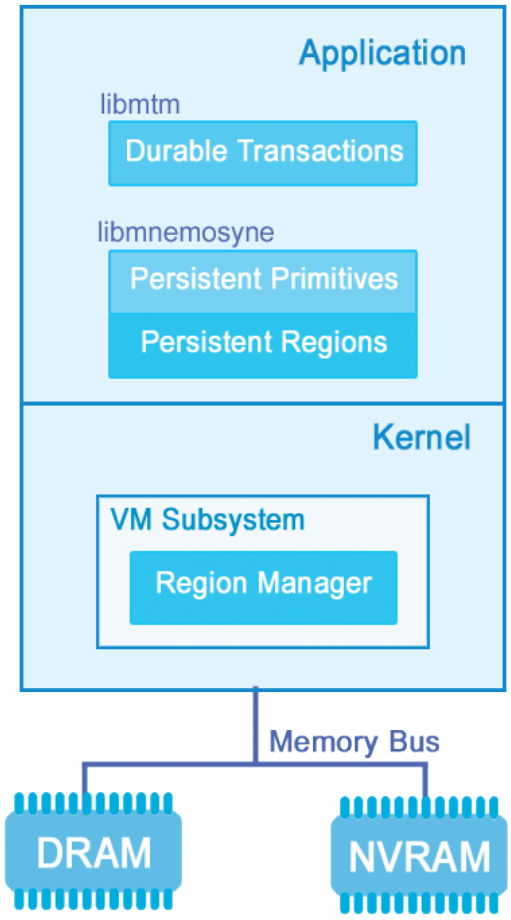

Figure 3.1: Mnemosyne architecture. User space components are implemented above the kernel level persistent region manager.

Persistent regions are virtualized in Mnemosyne by persisting the virtual address to physical address mapping, and swapping NVRAM pages to *backing files* that it allocates when creating a region. Mnemosyne provide the following abstractions to make persistent memory regions, and to reference the these persistent regions. These abstraction are implemented by a combination of kernel support, library support, and compiler support for declaring persistent variables.

1. **pstatic**
   Programmer can declare `static` persistent variables by declaring it as `pstatic`. Declaring variable as `pstatic` tells the linker to place the variable in a persistent region. Similar

to `static` variables in volatile memory systems, `pstatic` variables are initialized when the program first runs, but, it `pstatic` variables can retain their values across invocations of program.

This `pstatic` keyword inserts a compile-time annotation with the `__attribute__ ((section("persistent")))`. The static linker combines all persistent variables declared as `pstatic` into a single `.persistent` ELF section in the executable.

2. **pmap**
   Mnemosyne provides `pmap` function to explicitly create persistent regions. Functionality of `pmap` is similar to `mmap`. Mnemosyne automatically maps dynamic regions created by the program on previous invocations into the address space when it initializes. To prevent newly created sections from being lost if the application crashes, the `pmap` function takes as an in/out parameter a persistent variable to receive the regions address. A programmer deletes a persistent region by calling the `punmap` function, which takes the starting address and the length of the region.

3. **persistent heap**
   While dynamic persistent regions offer programmers a generic way to store data of any size and structure, Mnemosyne also provides a persistent heap, which enables dynamically allocated persistent variables

4. **persistent keyword**
   Mnemosyne lets programmers annotate the target of a pointer type as persistent with the `persistent` keyword. It serves as an indication to the compiler of the persistence type of the target, which can be used to identify potentially dangerous assignments of volatile address to a `persistent` pointer, and vice versa. This ensures that persistent data, which survives restarts, does not refer to volatile data that is lost.

   `persistent` keyword inserts a compile-time annotation with the `__attribute__ ((address space(1)))`. This annotation is interpreted by the semantic parser as a pointer to persistent virtual address, and it designates such annotated pointer targets in address space 1. All other pointer targets are designated in address space 0. Sparser essentially treats pointers with identical target types but different address spaces as distinct types, and warns about code that mixes pointers to different address spaces. This allows the Mnemosyne to identify the code that accidentally uses non-volatile pointers to refer to volatile address and vice versa.

3.1.2 Creating Persistent Mappings for Persistent Regions

One important challenge of exposing persistent memory regions to user space is ensuring virtual to physical address space mappings are persistent. For conventional volatile memory, virtual to physical address space mappings and mappings of pages that swapped to swap partitions are deleted when the program terminates. Operating system also clears the data swapped out to swap partitions once the program is exits. However, for persistent regions the mappings of

virtual addresses to physical persistent pages must survive system restarts. In addition, data swapped out of NVRAM must also survive system restarts.

Mnemosyne implements the persistency of mappings as a two layer approach.

1. **Region manager**
   Region manager is a kernel mode layer and it exposes NVRAM to user space as a memory-mapped file
2. **libmnemosyne library**
   It is a user space library and it associates each persistent region with a specific file and records the address of each region.

3.1.2.1 Region Manager

Mnemosyne extended the existing Linux virtual memory system to add a new sub system to provide support for persistent virtual memory. The region manager considers NVRAM as new memory zone, and created a new flag, `MAP_PERSIST`, to the mmap system call implementation to indicate the a file should be mapped to new created NVRAM zone and not DRAM. The region manager uses a **persistent mapping table** to keep track of the virtual pages currently stored in NVRAM. The persistent mapping table is also persisted at the base of NVRAM, and it stores a triple containing :

- physical frame number of NVRAM page,
- inode number of the file corresponds to a memory region, and
- page offset in that file.

Persistent manager constructs the persistent regions when the Operating System boots. It does the following operations to create persistent regions :

- Region manger scans the entire mapping table. The Physical frame number of NVRAM page is the index to mapping table.
- For each valid entry in persistent mapping table, region manager updates the Linux page descriptor.
- Region manager creates an in-memory inode for each backing file when it first come across an inode number in mapping table.
- It also creates a free list of NVRAM pages, and inserts the NVRAM pages which don't have a valid mapping in persistent table.
- At the end of system boot, the region manager evicts the NVRAM pages to corresponding backing files using the Linux page descriptors and its inodes.

The region manager does not update the page table entry corresponds the persistent regions when starting the application. So all the access to persistent regions will generate a page fault. The page fault on accessing the persistent memory regions are treated in two ways.

1. **Soft page faults**
   If the faulty page is available in the NVRAM and is not evicted to backing file, these types of faults are treated as soft page fault. Soft page fault is handled by updating the page table of the process with out copying the data from the backing file.
2. **Hard page faults** Hard page faults are handled by copying the data from backing file to NVRAM, and updating the page table. It may require to evict some page from NVRAM, if free page are not available on it.

3.1.2.2 *libmnemosyne* Library

The libmnemosyne library creates a backing for each persistent region, and records the address of each region. Mnemosyne reserves 1 TB virtual address space for creating persistent regions, which allows quick determination of whether an address belongs to persistent region. The libmnemosyne library records the address of each persistent region in **region table**, which is stored at static persistent region of each process.

The libmnemosyne library is used to create static and dynamic persistent regions. Creating persistent regions and heaps are discussed in next section.

3.1.3 Persistent Memory Interface for Applications

3.1.3.1 Creating a Dynamic Persistent Region

Dynamic persistent regions are created as follows:

- Application calls the `pmap` functions.
- `pmap` function creates an empty backing file, and allocates an inode for backing file.
- Then it invokes `mmap` system call with a flag `MAP_PERSIST`.
- The region manager updates the page table, and returns the address space allocated for the memory-backed backing file.
- `pmap` function creates an entry in region table of calling process. It records (i) starting virtual address and length of the backing file, (ii) name of the backing file, (iii)and metadata containing flags.

Metadata stored in each region table entry of a process is used to indicate the success of `pmap` operations. When an applications starts, the libmnemosyne library looks the region table and destroys the partially created regions.

3.1.3.2 Creating Persistent Regions for `pstatic` Variables

Static variables are declared using `pstatic` keyword, and for those variables libmnemosyne creates a separate persistent backing file when the program executes for first time. It also populates the new region with the initial values assigned for `static` variables. To clear the values of all `static` variables, program needs to delete the corresponding backing and restart.

3.1.4 Creation of Persistent Heap

The libmnemosyne also provides support for persistent heap. The interface provided by libmnemosyne to applications to manage the persistent heap are

- `pmalloc()`
  `pmalloc` is used to allocate memory in persistent heap. In addition to number of bytes to allocate, `pmalloc` takes a `persist` pointer as an argument to ensure persistent memory is not leaked in case of system failure.

- `pfree()`
  It deallocates the persistent memory chunk stored in persistent heaps. Conventional `free` system call takes a pointer to memory chunks stored in volatile heap. But, `pfree` call takes a pointer to a persistent pointer as an argument to ensure that the persistent pointer does not continue to point to the deallocated chunk of memory if the system fails just after a deallocation.

3.1.5 Consistent Updates

Applications requires consistency mechanism to update persistent data with out risking corruption in case of system failure. The granularity of updates to the persistent memory depends the data structures that the application uses to manage the data. So each application requires different types of consistent update mechanisms. Shadow updates, journaling, and soft updates are various techniques used in file systems to implement consistency. Primary mechanism for ensuring consistency is ordering writes to persistent memory. But there are many features of CPU and memory that reorder the write operations to persistent memory.

3.1.5.1 Challenges of Implementing Consistent Update Mechanism

A consistent update mechanisms are implemented by ordering writes. But, the order in which CPU issues the write request can be different from the order in which the writes hits the persistent memory. It will happen due to various reasons.

1. **Caching**
   Persistent memory is attached directly into memory bus, and is subject to cache by the CPU. CPU cache can greatly improve the performance by reordering the write operations to get maximum serial updates to the memory. So the order in which the write operations hits the NVRAM depend on the cache size, and its replacement policy. The different cache write policy are given below.

   (a) Write-back policy
   The cache contents are occasionally written to memory via a certain eviction policy. The unit of eviction in write-back policy is cache line.

   (b) Write-through policy
   In write-through policy, when a cache line is written, the respective contents are also written in memory. But, only the updated words are required to written to memory, not entire cache line.

   (c) Uncachable
   In uncachable policy, cache is not used.

2. Different types of memory access instruction.

3.1.5.2 Hardware Support for Implementing Consistency

Existing processors provides different hardware primitives to ensure the writes reaches the persistent memory in right order.

1. **Different store operations** Writes to the NVRAM are possible through different types of store instructions. Each of them different latency and throughput.

   (a) `movq`
   The `movq` instruction writes to the cache.

   (b) `movntq`
   This instruction not writes to cache, and it directly writes to memory.

2. **Instruction fences**

   (a) `clflush`
   The `clflush` instruction is used to flush a given cache line.

3. **CPU cache flush instructions**

(a) `mfence`
The `mfence` instruction serializes all the store and load instruction issued before a `mfence` instruction. In other words, all load and store instructions that precede the `mfence` instruction in program order become globally visible before all load/store instructions that appear after the `mfence` instruction in the program order.

(b) `sfence`
It is similar to `mfence` instruction, but it serializes only store instructions.

3.1.5.3 Various Consistency Mechanisms

1. **Single variable updates**
Mnemosyne provides support for atomically updating single variables. This is use full initializing variables and updating counters.

2. **Append updates**
Append updates writes the data to a new location after the previous updates on same data. It never modifies the existing data, and is used for log implementations. Ordering is not required among the individual store operations belongs an append update. But store operations belongs to separate append updates are ordered. The incomplete append updates are discarded in case of system failure.

3. **Shadow updates**
Shadow update writes all the data into a new location called shadow location. Once the data is persistent it modifies the pointer to old location to refer to new location. Ordering is not required among the sore operation belongs to a shadow updates, and the pointer is modified only after all store operations to the new location are finished. It is best for tree like data structures. In case of system failures, the programs should release the memory allocated for incomplete shadow updates.

4. **In-place updates**
In-place consistent update technique uses transaction mechanism that undo or redo after a system failure. As a result, the program must make a copy of either the old data for rolling back, or the new data for rolling forward. Stores updating a data structure must be ordered after stores that create the copy. Unlike the preceding consistency mechanisms, in-place updates can be used to modify any data structure and therefore enable existing volatile structures to be persistent.

3.2 Implicit Impact of Non-Volatile Memory

Implicit impacts assumes the applications are unaware about the existence of non-volatile memory, and explain how the applications can be benefited from the non-volatile memory. Applications can be benefited from non-volatile memory from different ways like storing data in-memory file systems. Using NVRAM as disk cache reduces the write traffic, and provides better performance. Details of all these are explained in the next chapter.

NVRAM enables the researchers to design and implement fault-tolerant systems. Existing NVRAM-based fault tolerance techniques take NVRAM as a fast external storage device for checkpoints. The inherent limit of these systems is that they still follow the same architecture of traditional fault tolerance model, i.e. computation with extra fault tolerance mechanism

NV-Process [13] proposes a fault-tolerance process model based on non-volatile memory. Traditional process is tightly coupled with the operating system. In contrast, NV-process decouples processes from the OS, and processes are stand-alone instances running in a self-contained way in NVRAM. When the system is power off, NV-process instances reside in the NVRAM and can continue running where they left off as soon as the OS reboots. In the view of NV-process, the OS is no longer the container of processes, but the service provider for processes. NV-process enhances the traditional process model with fault tolerance support.

4. Revisiting the Operating System Subsystems

The existing operating systems and applications are designed to work with byte addressable, volatile, low latency DRAM/SRAM technologies, and block addressable, non-volatile, high latency persistent storage devices such as magnetic disk and flash storage. The legacy applications can be benefited from the implications of of non-volatile memory on operating systems designs. The introduction of non-volatile memory into memory hierarchy requires significant amount of changes in the various layers of operating system to completely exploit the features of byte-addressable non-volatile memory.

The NVRAM can be used as persistent main memory and the applications can get benefit of directly accessing persistent storage without the overhead of entering into different Operating System layers such as system calls, file systems, block IO layer, etc. In addition to the implications of NVRAM on Operating System due its byte addressability and persistency, the other physical properties of NVRAM such as energy efficiency and scalability, make it more promising for future computer system designs. The following sub sections describes the implications of NVRAM on different layers of operating system.

4.1 File System

In computing, a file system is used to control how data is stored and retrieved. Without a file system, information placed in a storage medium would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified.

There two main impacts of non-volatile memory in file system layer:

1. In-memory file systems
2. Storing file system metadata in non-volatile memory.

4.1.1 In-Memory File Systems

Recent works on integrating non-volatile memory into operating system and application design fully exploit the non-volatile memory by introducing new interfaces and abstractions. But it requires a greater amount of changes for applications. So it is necessary to have back-ward compatible interface for ease to ease.

`nvramdisk`[4] is an in-memory persistent file system developed by extending the popular in-memory file system, `ramdisk`. The `nvramdisk` provides transnational support for `ramdisk` to persistently maintain data. The main challenges of providing transnational support on a byte-addressable NVRAM are :

1. **Atomicity of write operations to a block**
   Writing a block into NVRAM consists of multiple store instructions, and they need to execute atomically.
2. **Atomicity of store operation to a word**
   Single store operations are not atomic against unexpected power failures which may result in partially written word in memory.
3. **Ordering guarantee**
   Durability support becomes non-trivial in `nvramdisk` since the order in which the write operations reflects on the NVRAM are different from which they are issued. Hence, a subset of the written instructions may be at the volatile layer of memory hierarchy. Hence ordering guarantee is necessary condition to provide durability support for write operations.

Figure 4.1 shows architecture of memory subsystem of `nvramdisk` consisting of both DRAM and NVRAM. The `nvramdisk` disk have same memory layout of `ramdisk`. Both NVRAM and DRAM are directly attached to CPU via memory controller.

`nvramdisk` provides atomicity and durability by adopting shadow block mechanisms, metadata journaling, and type-dependent ordering. Shadow block mechanism is used to ensure atomicity of write operation. This mechanism writes the data out-of-place so that the old blocks can be recovered in case of a power failure. In addition to page tables, `nvramdisk` uses a mapping table to keep track of location of a give logical block. The mapping table translates the logical block address to virtual address(virtual page number(`VPN`)). But the updates to mapping table are not atomic. The consistency of mapping table updates are ensured by journaling the history of updates to the mapping table.

Figure 4.2 illustrates the physical and logical memory layout of `nvramdisk`. The whole available memory is partitioned into 4 memory zones, `DMA ZONE, DMA32 ZONE, Normal Zone, and NVRAM Zone`. `NVRAM Zone` represents the address space corresponds to NVRAM, and it stores metadata, shadow blocks, and actual data blocks. Metadata region consists of mapping table and journal.

When Operating System boots, it builds the free page list to avoid allocating already used pages for other processes. So operating system does the following operating on booting.

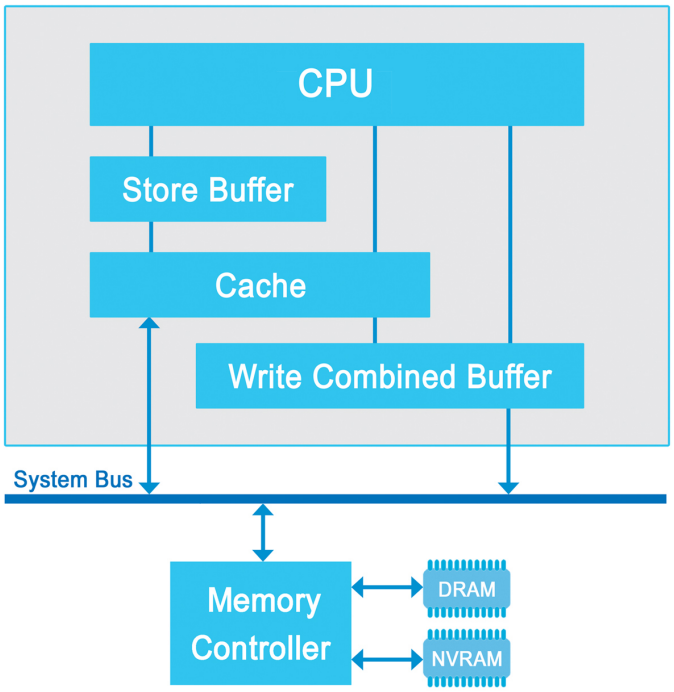

Figure 4.1: Architecture of persistent in-memory file system, *nvramdisk*.

- Operating System examines the journal to determine the `nvramdisk` has been properly unmounted. Operating system will invoke recovery procedures if the file system is in a inconsistent state.
- Operating system examines the mapping table, and removes the physical pages belonging to data region from the free page list.
- Then it examines journals, and remove the pages used as shadow blocks from the free page list.

4.1.1.1 Managing shadow blocks and journaling

`nvramdisk` ensures atomicity of write operations by adopting shadow block mechanism and journaling the updates to mapping table. When a block is written, to NVRAM, the corresponding mapping table entry is updated to refer to new location. Journaling is used to ensure the consistency of the updates to this mapping table.

The process of managing the shadow blocks and journaling contains the following 5 phases.

1. **Allocation Phase**
   `nvramdisk` allocates a shadow block in allocation phase. In addition, it creates a journal record entry in journal records.
2. **Update Phase**
   Update phase is used to update the data to allocated shadow block.

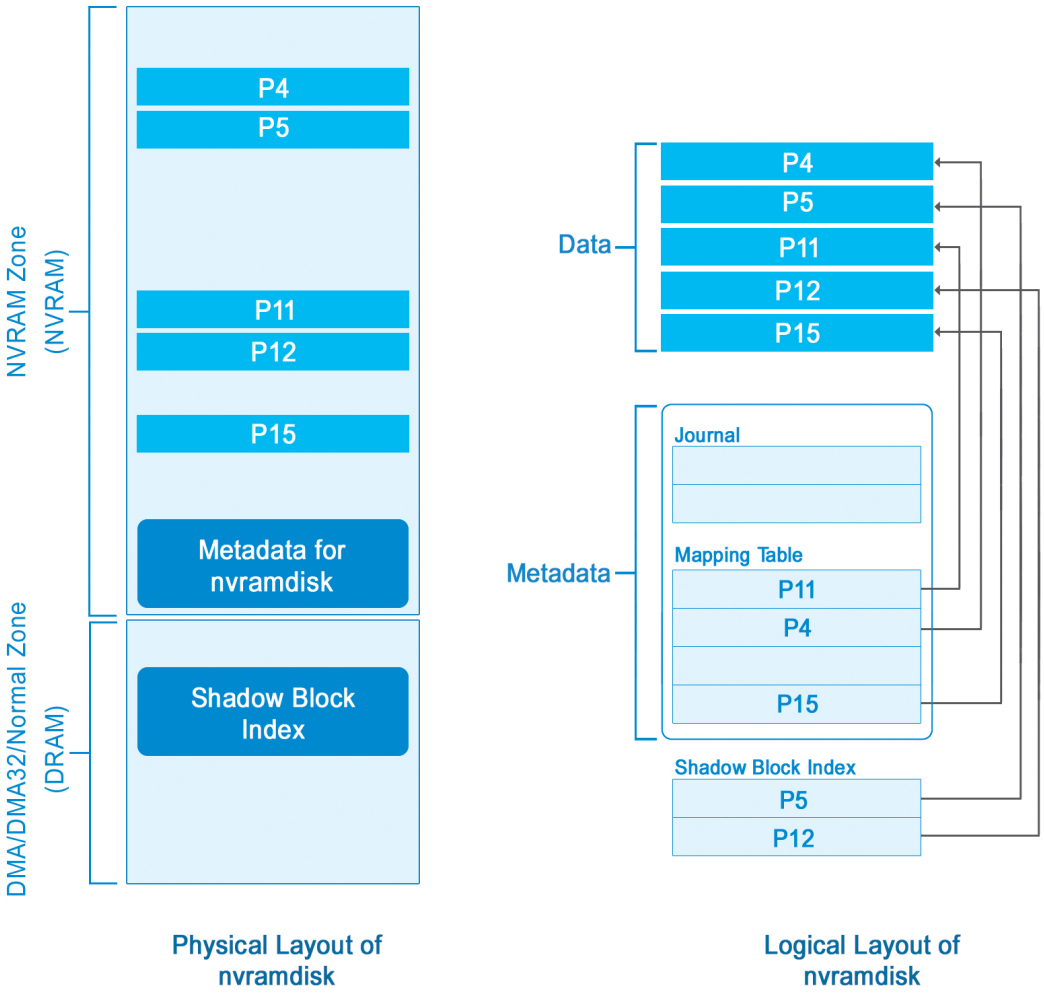

Figure 4.2: Physical and logical architecture of *nvramdisk*

3. **Journaling Phase**
   In this phase, `nvramdisk` journals the mapping table table entry updates.

4. **Checkpoint Phase**
   In the checkpoint phase, `nvramdisk` checkpoints the journal record to mapping table.

5. **Clean-up Phase**
   It is the last step, and in this step `nvramdisk` adds the obsolete block to the shadow block pool and resets the journal record.

The journal entry corresponds to each write operation maintains a virtual page number (`VPN`) and two replicas of logical block number. This is to recover the mapping table in case of system crash. Only when the two logical block number values are same in a journal entry, the logical block is valid. It helps to determine whether the system crash leaves the journal entry updates in inconsistent states. For example, only few bytes of 8 byte field is written at the time of system crash.

The journaling module does the following operations.

Table 4.1: Different states for mapping table journaling.

State	Journal Entry			Table $M[lba]$	Phase No
	VPN	L_1	L_2		
S_0	$\varnothing$	$null$	$null$	vpn_{old}	
S_1	X	$null$	$null$	vpn_{old}	
S_2	vpn_{shadow}	$null$	null	vpn_{old}	
S_3	vpn_{shadow}	X	$null$	vpn_{old}	3
S_4	vpn_{shadow}	lba	$null$	vpn_{old}	
S_5	vpn_{shadow}	lba	X	vpn_{old}	
S_6	vpn_{shadow}	lba	lba	vpn_{old}	
S_7	vpn_{shadow}	lba	lba	X	4
S_8	vpn_{shadow}	lba	lba	vpn_{shadow}	
S_9	vpn_{shadow}	X	lba	vpn_{shadow}	
S_{10}	vpn_{shadow}	$null$	lba	vpn_{shadow}	5
S_{11}	vpn_{shadow}	$null$	X	vpn_{shadow}	

- Records the new location of given logical block(ie. VPN of shadow block allocated) into VPN field in journal entry.
- Creates two replica of logical block number in journal entry.
- After updating journal entry, it updates the mapping table entry corresponds to the given logical block number to refer to the new `VPN`.
- Journal entry is reset to `NULL` when the transaction completes.

4.1.1.2 Crash Recovery

The process of updating mapping table can be divided into 12 steps, each corresponds to a different state. The table 4.1 shows these 12 steps for mapping table journaling. When the system crashes while updating a certain field, value of the field is marked '`X`'. The initial value of VPN field is Dont Care. Initially the the logical block numbers on journal entry, *L1*, and *L2*, are `NULL`.

The initial state of journaling is S_0, where mapping table entry contains the old `VPN`, logical block numbers in journal entry are `NULL`, and the `VPN` in the journal entry is don't care. When updating the mapping table, `nvramdisk` write the virtual page number of new shadow block into corresponding the journal entry. If it succeeds, the it reaches the state S_2, where the VPN field in the journal entry contains the address of newly allocated shadow block, and L_1 and L_2 are `NULL`.

If it fails to updates, the `VPN` filed will contain an incorrect value, and it leads to faulty state S_1. Then, nvramdisk creates two replicas of logical block address in the journal entry. If it succeeds to record first logical block number, the state will be changed to S_4. S_6 denotes the state at which both the replica of logical block numbers are successfully created. The states S_3 and S_4 denotes the faulty states which will happen if system fails while recording the replicas of logical block numbers.

After successfully updating the journal entry, `nvramdisk` updates the the actual mapping table entry to refer to the new location of a given logical block, and it reaches the state S_8. Then it resets the journal entry by setting `NULL` to logical block number fields, and it goes to the state S_0. In case the system fails while resetting the journal entry, it will end at either S_9, or S_{11}.

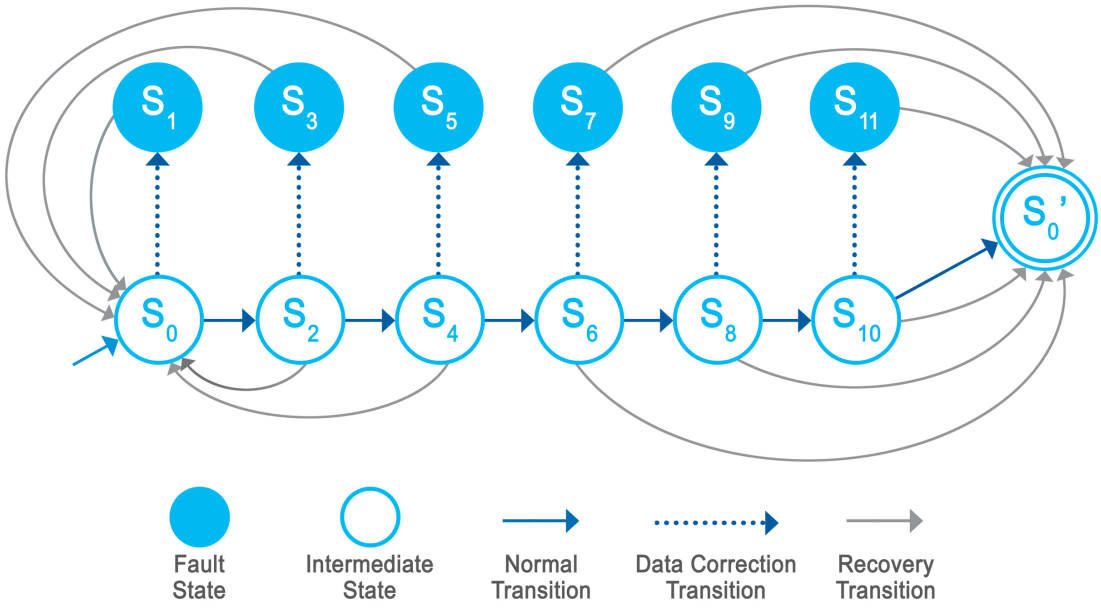

Figure 4.3: State diagram illustrating the recovery procedure.

In case of a system failure while journaling, the `nvramdisk` performs a recovery procedure while booting the system. The recovery procedure follows either **undo** or **redo** based on the state at which the system fails. If the system is found in one of of S_0,...,S_5, the recovery procedures invokes undo operations which resets all `lba` fields to `NULL`, and the system reverts to the state S_0. When the system is in one of S_6,...S_{11}, the recovery procedure performs redo operations. The redo operation updates the mapping table entry using the journal entry, and performs a clean-up.

he figure 4.3 illustrates the state diagram of these recovery procedure. Figure contains three types of states: initial state, normal states, and faulty states. Double circle denotes the initial state, and single circle denotes the normal states. The circles with cyan background denotes the faulty states. The normal transitions, transition from a normal state to next normal state, are denoted by solid cyan lines. The transitions caused by system crash are denoted by dotted lines. The recovery transitions are denoted by gray lines.

4.1.1.3 Type Dependent Ordering Guarantee

Despite the ability of NVRAM to store the data across system failures, the failures causes the data loss in volatile layer of memory hierarchy. The data saved in CPU registers, TLB caches will lost in case of system failure. So it is necessary to synchronize the data stored in these volatile layers to non-volatile NVRAM. This problem can be solved in the following ways.

1. **Write through mechanism**
   It is the safest mechanism for ensuring the ordering guarantee. But it wastes the CPU time since the write through mechanism requires each store instruction to hit to NVRAM, which order magnitude slower than CPU and CPU cache.

2. **Type dependent ordering guarantee**
   Type dependent ordering guarantee suggests that the ordering is not required among all the store operations. For example, ordering guarantee is not required among the store instructions for writing a data block. Hence it is better to use different hardware primitives based on the access characteristics of respective NVRAM regions: data block, mapping table, and journal entry.

Table 4.2: Different ordering guarantee as combinations of different cache policies, memory store instructions, and barrier instructions.

Method	Cache Policy	Store Method	Flush	Barrier
M_{CF}	WB	`movq`	`cflush`	`mfence` / `sfence`
M_{NT}	WB	`movntq`	none	`mfence` / `sfence`
M_{WT}	WT	`movq`	none	none
M_{UC}	UC	`movq`	none	none
M_{WC}	WC	`movq`	none	`mfence` / `sfence`

There are number of factors which alter the order in which memory writes hits the NVRAM, which are already explained in sections 3.1.5.1 and 3.1.5.2. Combining all of these hardware primitives, different ordering guarantee methods are possible, and they are listed in the figure 4.2. All of these 5 methods can be used to ensure ordering among the memory writes. But the throughput achieved will be different, and it depends on the memory access characteristics.

The objects in different regions which correspond mapping table, journal, and data blocks, carry different access characteristics. For example, the data regions are accessed in a granularity 4 KB. Mapping table are accessed in 8 Bye units. The journal entry is also updated as 8 Byte units. Hence it is better to use different ordering guarantee methods for different regions.

4.2 Storage Systems

Storage systems resides at the bottom of memory hierarchy. Latency of read and write to storage devices are very slow, and hence the applications performance depends on the performance storage devices. The next section explains the impact of non-volatile memory on storage layer.

4.2.1 Non-Volatile Memory as Disk Cache

Hybrid memory systems utilizing both NVRAM and DRAM technologies are promising alternatives for DRAM only systems. Storage systems can also benefit from NVRAM technologies by using NVRAM as cache. Typically storage systems rely on DRAM cache due its short access latency, and is used as read cache. But it suffers long I/O time for write requests. To avoid the long I/O time, to reduce the write traffic to slow storage disks, and to avoid the data loss, storage systems can use hybrid cache containing both NVRAM and DRAM.

Hybrid cache can be implemented by using DRAM as read cache and NVRAM as write cache. The NVRAM write cache can minimize the write traffic to the slow disks by delaying and reducing the dirty page synchronization. The effectiveness of write traffic reduction compared to DRAM-only cache is demonstrated on figure 4.4. This experiment compares the write traffic of an NVRAM-only buffer cache with hybrid cache. The DRAM-only buffer cache that periodically flushes dirty pages to disk. In hybrid cache, the amount of DRAM varies from 4 MB to 256 MB. In NVRAM, cache pages are evicted only when the capacity of NVRAM is reached. From the experiment it is clear that adding 4MB of NVRAM into hybrid cache reduces the write traffic by 55.3%.

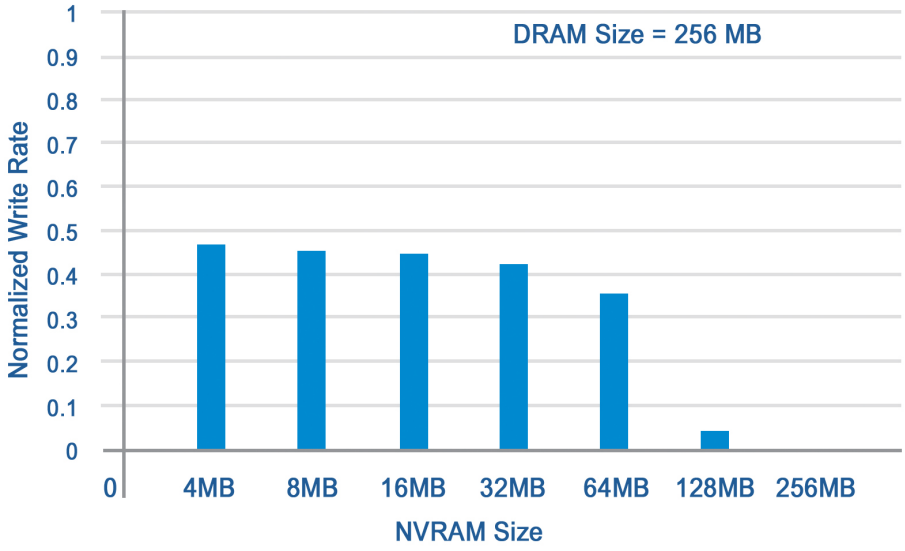

Figure 4.4: The effectiveness of write traffic reduction compared to DRAM-only systems (Data from [5])

Reducing the write traffic into persistent media does not always increase the performance. The access latency of NVRAM technologies is several time longer than the latency of DRAM. So,

when NVRAM is used to store the dirty pages, a read hit on a dirty page needs to access NVRAM, and it is slower than reading the page from DRAM.

4.2.1.1 Accessing Dirty Pages

The throughput of hybrid cache depends on the way in which the dirty pages are accessed. It can be done by page replication mechanism and page migration. Both of them have their own pros and cons.

1. **Page replication**
   On a read hit to a dirty page, the page replication mechanism triggers a page copy from NVRAM to DRAM. The main advantage of page replication mechanism is the later read to same page can be served from DRAM itself. But, if the page is not accessed multiple times, one page of cache space is wasted in DRAM.

2. **Page migration**
   Page migration requires to move the page from NVRAM to DRAM. Page migration does not waste the page, and a single page never exists in both NVRAM and DRAM. But the problem is it suffers longer average read access latency if there are updates between reads. The updates between read bring the page back to NVRAM, and is very slow.

4.2.1.2 Hibachi: A Cooperative Hybrid Cache

The hybrid cache implementation, *Hibachi* [5], propose to use page migration technique instead of page replication. This is due to the fact that read hits occur mostly in DRAM, and rarely in NVRAM. The architecture of Hibachi is illustrated on figure 4.5. The *hibachi* architecture contains two real caches and two ghost caches. Both real and ghost cache contains clean and dirty caches. The clean cache controls all the clean pages in DRAM and NVRAM. The dirty pages in NVRAM are managed by dirty cache. The ratio between the clean cache and the dirty cache capacity dynamically varies based on the current workloads tendency assisted by the two ghost caches. The ghost cache does not saves any real data, it stores only recently evicted page numbers.

4.2.1.3 Managing Clean Cache and Ghost Cache in Hibachi

The hibachi manages the dirty cache using recency-based policies such as LRU. Frequency-based policies such as LFU-Aging (Least Frequently Used with Aging) are used to maintain the clean cache. Each data page in real caches, real clean cache and real dirty cache, maintains a counter. Clean page counters are used to record their access frequencies, and dirty page counters are used for migration purpose. In addition, *Hibachi* maintains two hash mapping table and a sequential list to efficiently identify consecutive dirty pages for implementing the sequential writes to disk.

Right Prediction
Cache hit ration depends on the ability to predict whether a page will be reused in the near future.

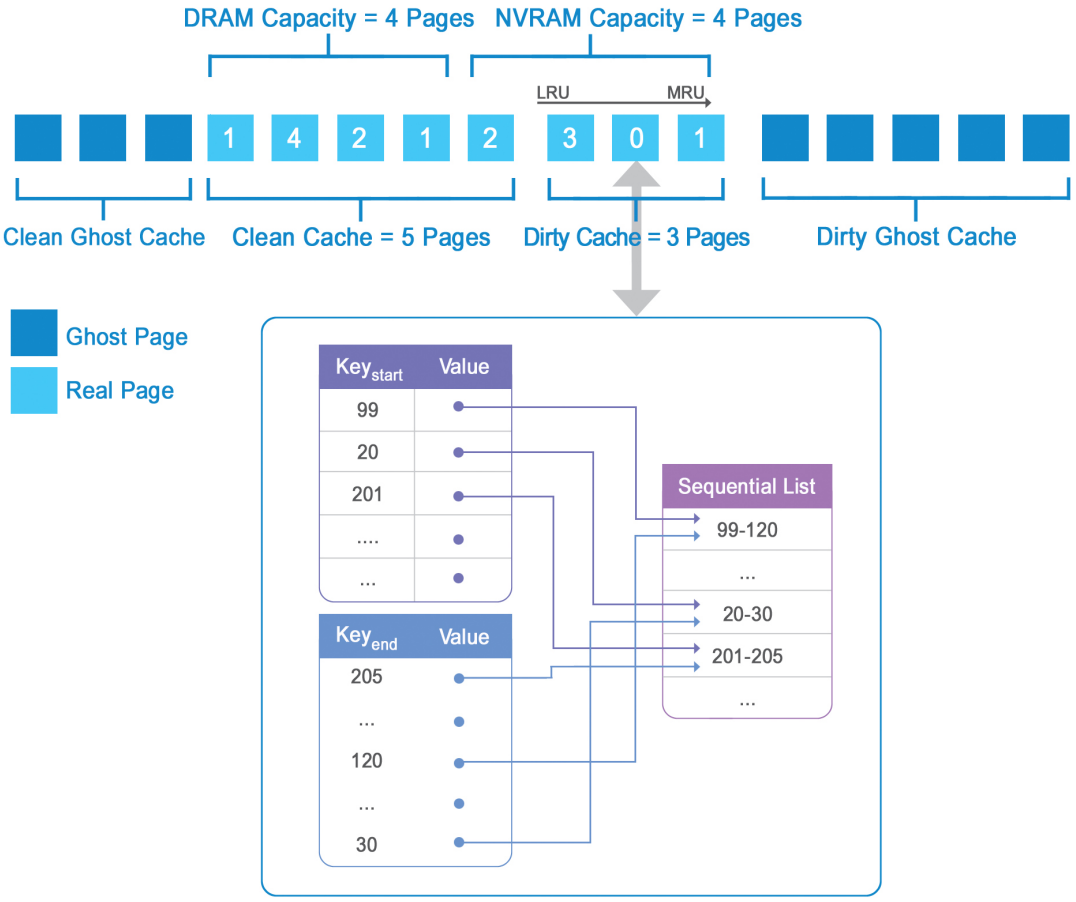

Figure 4.5: Architecture of co-operative cache, *Hibachi*

Recency and frequency are the most important matrices for predicting the reusability of a page. Since the temporal distance of read request after a read request is long, the recency is not a good metric for predicting clean page reuse. But, the temporal distance between write request after a write request are relatively short, the recency is a good metric to predict the dirty page reuse.

Since, the most of the read and write requests concentrates on small portion of pages, the frequency is also a good metric for reuse prediction. This is the reason why *Hibachi* uses LFU-Aging to manage clean cache, and LRU to manage dirty cache. LFU-Aging reduces all frequency counts by half when the average of all the frequency counters in the cache exceeds a given *average frequency threshold*.

Right Reaction

Normally, if a page in cache gets a hit, the page is moved to most recently used position, and its frequency counter is increased. But, *Hibachi* distinguishes the read hits and write hits separately to reduce the write traffic by improving the write hit rate on NVRAM, and to reduce read latency by increasing read hit rate on DRAM.

If a page is written once or rarely, but frequently read, keeping this page in NVRAM causes longer read latency. Hence it is better to migrate these types of page to DRAM. To find these types of pages, *Hibachi* increases the frequency counter of dirty pages only for read hit. For

write hits on page in dirty cache, the page is moved to most recently position since LRU is used to manage dirty cache.

When a dirty page in NVRAM is selected to evict, it first compares with least frequent accessed clean page in DRAM. If clean page's frequency is less than dirty page's frequency, then clean page is evicted from DRAM, and the dirty page is migrated to DRAM. Note that migration of dirty page from NVRAM to DRAM, or eviction of a dirty page must be synchronized to storage.

Right Adjustment
Hibachi is also capable of dynamically adjusting the size of clean cache and dirty cache in NVRAM according to the work loads. Resizing is done with the help of two ghost caches, ghost clean cache and ghost dirty cache.

- The Size of the desired dirty cache is denoted by D, and the size of desired clean cache is denoted by C.
- S is used to denote the size of maximum real cache pages that can be stored in both DRAM and NVRAM, such that C should be less than S, and D should be less than the size of NVRAM.
- If a page hits the clean ghost cache,it means that the page should not have evicted from clean cache. In this case, it increases C by one page and decreases D by one page.
- If there is a hit on page in dirty ghost cache which means the page should not have evicted from dirty cache. Hence the desired dirty cache size, D, should be increased. But the increase should be faster to keep the dirty pages in the cache longer and to reduce the write traffic. Hence, if the clean cache size, C, is smaller the dirty cache size, D, then D is increased by two pages. If the C is greater than or equal to D, then D is increased by two times of C/D.

Right Transformation
The granularity of cache page eviction *Hibachi* is different for clean cache and dirty cache. As usual, the clean cache pages are evicted one at a time when the cache needs to reclaim more space. While evicting the dirty cache pages, *Hibachi* finds and evicts the longest set of consecutive dirty pages to exploit the fast sequential writes to disk. But, if the size of longest set of consecutive dirty pages are less than a given threshold, it evicts the LRU pages from dirty cache. In both case it adds the evicted page numbers into MRU position of ghost cache.

The longest consecutive dirty pages are identified with the help of *sequential page list* shown in figure 4.5. The dirty pages with consecutive page numbers constitutes a sequential page list. The sequential page lists are are arranged in the order length of the list. In addition, there are two hash maps which are refers to corresponding sequential list. The key to first hash table the starting page number in the corresponding sequential list, and the key to other table is the ending block in number in the same list.

When adding a new page to dirty cache, *Hibachi* checks whether it can merge into any existing sequential lists by looking up two Hash Tables. If the new dirty page can merge into any existing sequential lists, *Hibachi* merges two sequential lists into one larger list. The hash maps and

corresponding sequential list information are also updated properly. If a newly added page does not have any neighbours in the dirty cache, a new sequential list entry is created, and mapping tables are updated. While synchronizing the dirty pages, either the pages belongs to a sequential list or a single page, the corresponding sequential list is deleted.

4.3 Memory Scaling

Data center applications like key-value stores, in-memory databases and data analytics handle exponentially growing data sets, but they cannot tolerate the performance degradation caused by spilling their workloads to disk. Placing their growing datasets in DRAM will be unviable in future due to the scaling issues of DRAM. This problem can be addressed by emerging Non-Volatile Memory Technologies, which can be positioned in between DRAM and secondary storage in terms of both performance and cost. But it leads to slowdown for applications running in NVRAM-only systems. To overcome this performance degradation, future systems are likely to couple NVM with a smaller amount of DRAM, resulting in a hybrid memory systems.

A simple approach for hybrid memory architecture is treating DRAM as cache for NVM and dynamically moving the data between them on demand using traditional paging mechanism. But it leads to sub optimal performance. It does not consider the fact that objects with different access characteristic can be placed on the same page.

`xMem` provides interface for users to specify tag for objects while allocating memory. Each tag corresponds to a data structure. Profiling tool identifies the memory access pattern of these data structures and based on the access pattern it automatically moves the data structures across DRAM and NVRAM.

4.3.1 Challenges of Hybrid Memory Management

Memory subsystem is agnostic to the how the application manages the allocated memory. Hence the following are the main challenges of implementing hybrid memory systems with performance similar to DRAM-only systems.

1. Identifying how the application data structures are stored in the allocated memory.
2. Determining the access pattern of different data structures and pages.
3. Dynamically mapping the virtual pages to DRAM and NVRAM.

X-Mem[7] addresses these challenges by automatically placing the data structures in appropriate memory types. X-Mem provides new memory allocation interfaces to programmers, using which the programmers can can specify a tag during memory allocation. y using a unique tag for each data structure, programmers provide the necessary information for the X-Mem run time to map each object to its data structure. The section, 4.3.2, details the architecture of X-Mem.

4.3.2 X-Mem Architecture

An overview of X-Mem architecture is shown in figure 4.6. X-Mem exposes two new interfaces,`xmalloc(int tag, size_t size) and xfree(size_t size)`, and the application uses these interface to allocate memory. Working of `xmalloc` is similar to `malloc` except, it receives a tag. All the objects of a data structures are created using same tag, and the tag is used to identify different data structures.

Newly created memory region are place in DRAM by displacing some memory regions in DRAM to NVM. `xMem` uses `mbind()` system call, to move the regions across DRAM and NVRAM, which copies the corresponding pages and updates the page tables. X-Mem uses a profiling tool to identify the access pattern of different data strictures, and appropriately places the virtual pages either into DRAM or NVRAM.

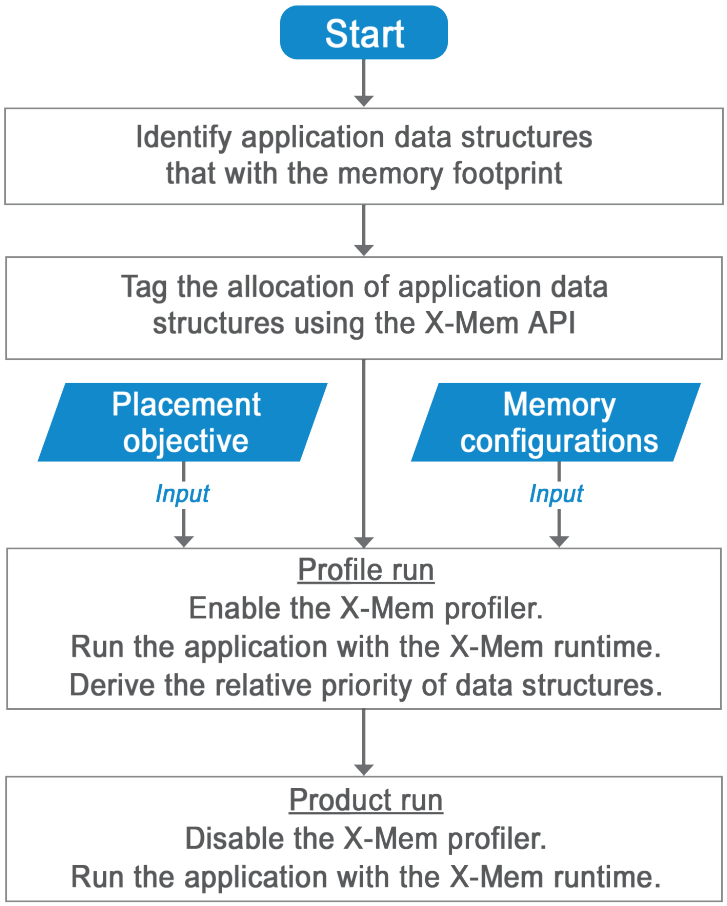

Figure 4.6: Overview of *X-Mem* architecture. X-Mem first identifies the memory access pattern of an applications, and then profiler tool determines the best memory placement policy for the applications. Based on the result of the profiling stage, the applications runs ever,and operating system uses same placement policy.

The profiling tool identifies the memory access pattern to memory regions of each data structures. It is achieved by intercepting all the memory accesses using PIN and recording the addresses in a buffer. It periodically processes this buffer and sorts the addresses recorded in the buffer.

based on the access pattern of data structures, a reference list is created by identifying the data structures whose relocation to DRAM will reduce the average memory access time. The next section, 4.3.3, details the different types memory access. The section 4.3.4 illustrates the placement of pages based on the profiling results.

4.3.3 Types of Memory Access Patterns

The stall for accessing each data structure is not the same as the time to access memory in modern computers. There are multiple cases where effective latency to access memory is much small smaller than the physical latency for certain access patters.

1. **Instructing streaming and out-of-order execution**
   The processor can hide the latency to access memory by locating multiple memory requests in the instruction stream and then using out-of-order execution to issue them in parallel to memory via non-blocking caches.
2. **Instruction prefetching**
   The process incorporate prefetchers that locate striding accesses in the stream of addresses originating from execution and prefetch ahead.

The average memory access latency in the above case varies for different access patters, sequential and non-sequential [9]. Table 4.3 illustrates the different types of access patterns.

1. **Random access pattern**
   The instruction stream consists of independent random accesses. The processor can use its execution window to detect and issue multiple requests simultaneously.
2. **Pointer chasing**
   The processor issues random accesses but the address for each access depends on the loaded value of a previous one, forcing the processor to stall until the previous load is complete.
3. **Streaming**
   The instruction stream consists of a striding access pattern that can be detected by the prefetcher to issue the requests in advance

Table 4.3: Different types of memory access patterns

	Dependent	Independent
Sequential	NA	Streaming
Non-sequential	Pointer chasing	Random

The stall latency of these access patterns are different. The figure 4.7 illustrates the stall time for read request of these access patterns for different physical latencies. Pointer chasing experiences

the maximum amount of stall latency, equal to the actual physical latency to memory. In the case of random access, out-of-order execution is effective at hiding some of the latency to memory. In the case of streaming access, the prefetcher is even more effective at hiding latency to memory.

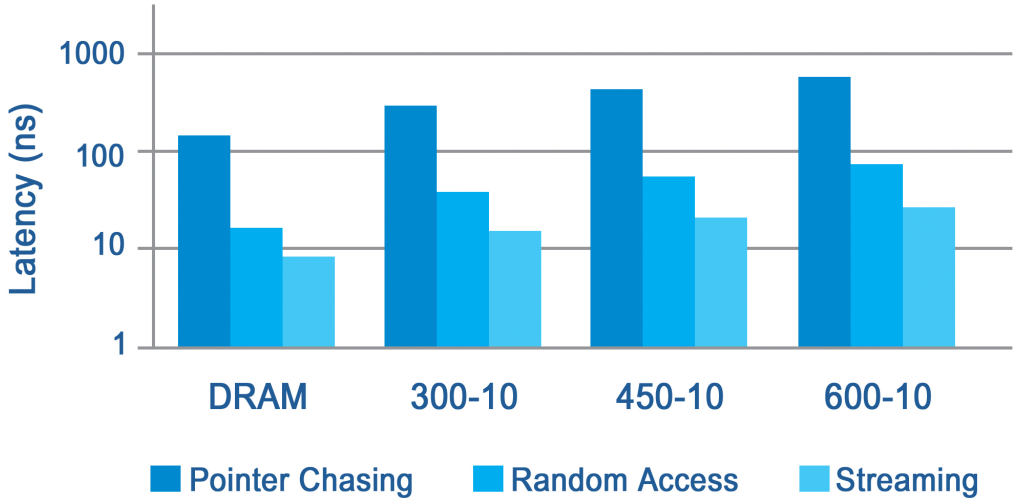

Figure 4.7: Memory read latency of pointer chasing, random access, and steaming memory access patterns. (Data from [7])

4.3.4 Profile Guided Page Placement

For each data structure tag, the profiling tool classifies accesses to its regions as one of random, pointer chasing and streaming, and counts the number of accesses of each type. The profiler uses PIN intercept memory accesses and record (for each access) the address that was accessed. This information is stored in a buffer and processed periodically. For each window of memory operations, we the profiling tool sorts the accesses in the buffer, effectively grouping accesses to the same region together. Then the profiling tool iterates through the buffer and searches the access pairs where value at the location of first access equal to the address of the location of the second access. This types of access are interpreted as pointer chasing. For streaming access, consecutive elements in the window differs by one. All other accesses are treated as random accesses.

Let $T(r)$ be the type of memory in which the region r is placed, and let $D(r)$ be the parent data structure of r. Then the average memory access for a given configuration can be estimated as follows.

$$A = \sum_{r \in Regions} \cdot \sum_{p \in Patterns} F_{D(r)}(p) L(p, T(r)) \tag{4.1}$$

The placement algorithm aims to minimize the average latency by appropriately mapping the regions to memory types. Initially, it assumes that all regions are in NVRAM. So moving some memory regions to NVRAM will reduce the average access time. So, for any memory region r the benefit of moving to DRAM is:

$$B(r) = \sum_{p \in Pattern} F_{D(r)}(p)[L(p,NVM) - L(p,DRAM)] \tag{4.2}$$

The estimated average access time in a hybrid memory system can therefore be rewritten as follows:

$$A = \sum_{r \in Regions} . \sum_{p \in Patterns} F_{D(r)}(p) L(p,NVRAM) - \sum_{r \in DRAM} B(r) \tag{4.3}$$

Greedy algorithm is used to maximize the benefit of moving regions from DRAM to NVRAM. It first sorts the regions based on their benefit value, and places the regions with more benefit into DRAM until no more space is left in DRAM.

Summary

Chapter 4 overview the implications of non-volatile memory on operating system subsystems. It includes the implications persistent memory on cache designs, file system, and hybrid main memory systems. Hybrid main memory systems, containing both random-access memory and non-volatile memory, are promising to meet the memory demands of future big data applications. In hybrid main memory with tiering architecture, we need operating system supports to place data wisely between random-access memory and non-volatile memory according to data-access patterns for better performance.

Implications of non-volatile memory on different operating system subsystems, also improves the performance of legacy applications. Th in-memory persistent file system improves the performance of applications by avoiding the high access latency of hard disk and flash devices. Some file system researches exploits the byte-addressability of non-volatile memory to store the meta data of file system. The write traffic to slow hard disks can be reduced by introducing the non-volatile memory into disk storage layer as cache devices. Cache designs containing non-volatile memory reduces the write traffic by delaying the synchronization of dirty data with secondary storage. Some cache designs uses the persistent memory as second level cache to store dirty pages, and some other researches considered both DRAM and non-volatile memory to design cooperative cache.

5. Implications of Non-Volatile Memory on Virtualization

Persistent memory has gained significant interests from both hardware and software sides. With the proliferation of hardware and software systems for PM, as well as the emerging in-memory computing instances for big data from major cloud providers, the persistent memory will be an indispensable part of cloud computing systems. Current researches shows that cloud will likely incorporate non-volatile memory to increase the services reliability, scalability, and availability. A hypervisor can trivially support non-volatile memory as DRAM. Virtualizing NVRAM as DRAM does not fully exploit its properties. The section 5.1 details the virtualization of non-volatile memory in virtual machine environment. The implications of non-volatile random-access memory at hypervisor level are explained in section 5.2.

5.1 Virtualizing Non-Volatile Memory

Virtualization of non-volatile memory are are motivated by two important trends in current computer world.

- The byte addressability and persistency make PM very promising for big-data applications demanding low latency and high throughput, where crash recovery is a indispensable issue.
- Cloud has been a standard computing platform to run many big-data applications.

A hypervisor can trivially support NVRAM as DRAM. But there are many benefits if persistent memory is virtualized, and hypervisor is able to control it.

1. **Cost effectiveness**
   Statically and directly provisioning physical PM to guest VMs are not only inflexible but also not cost effectiveness. By adding an indirection between the VMs and the PM, the hypervisor can provide more flexible resource management to maximize the utilization of PM according to their characteristics. For example, for those read-intensive workloads running on PM, a hypervisor can use DRAM or SSD to emulate PM to release more physical PM to other VMs, while still retaining performance and persistence of PM.

2. **Performance**
   Virtualizing NVRAM also helps to bridge the performance variation between DRAM and NVRAM. For example, the hypervisor may leverage DRAM to transparently serve some read-mostly workloads for PM with inferior read performance.

3. **Ease of management**
   Persistent memory virtualization provided by the hypervisor can greatly ease the applications management of PM. For example, a VM can request a large PM and use it as block device leaving the entire PM management for hypervisor. In the same time, the VM can request to expose persistent memory as a directly attached memory device for its own fine-grained control over PM.

Virtualizing NVRAM have a lot of key benefits, But there are many challenges. The section 5.1.1 describes the challenges of virtualizing NVRAM. There are two typical ways to virtualize PM similar to traditional server virtualization:

1. **Full-virtualization**
   Full vitalization provides functionally-identical virtualized PM to guest VM. It also provides transparency.

2. **Para-virtualization**
   It abstracts a similar but not completely identical interface. Para virtualization requires modification for operating system running in virtual machine, but, it provides better performance as compared to full virtualization.

Full virtualization of NVRAM is detailed in section 5.1.2, and section 5.1.3 details para virtualization. In both of these virtualization techniques, NVRAM can be exposed to virtual machines either as *memory devices*, or as *block devices*. Exposing NVRAM as memory device provides better performance since it load/store operations bypasses most of the software layers. But, exposing as block device enables the existing applications to benefit from PM.

5.1.1 Challenges of Virtualizing Persistent Memory

There are several technical challenges in providing efficient virtualization for PM.

1. Current virtualization lacks support for an appropriate abstraction for PM.

2. Persistent memory exhibits different performance and price characteristics from DRAM. So it is critical to consolidate persistent memory to maximize cost-efficiency, performance, and endurance.

3. Ensuring proper and efficient crash recovery under a power failure.

5.1.2 Full-Virtualization of Non-Volatile Memory

5.1.2.1 Exposing NVRAM using memory interface

Persistent memory devices such PCM and MRAM are attached directly to memory bus using DRAM like interface, and are accessed using normal load/store instructions. The memory interface is virtualized through *namespace* virtualization in VPM, [10]. The architecture of VPM is explained in section 5.1.2.3

A namespace, including address range and caching modes. Namespace is delivered by the virtual BIOS to the guest OS during the booting process. Specifically, Intel x86 provides a set of memory type range registers, which can be leveraged by the hypervisor to virtualize the namespace. The mapping between guest PM pages and host PM pages are managed by using nested page tables.

5.1.2.2 Exposing Non-Volatile Memory Using Block Interface

NVRAM technologies like NVDIMM supports block interfaces. They also can be virtualized by configuring virtual BIOS to emulate PM device. It requires exposing PM-related control registers to VM, and access to these registers are controlled by *trap-and-emulate* mechanisms. Hypervisor can the PM through the memory interface even it is virtualized as block device. In this case, hypervisor first maps the entire PM to its own memory space, and reads or writes the PM according to the commands issued by the guest VMs.

5.1.2.3 Architecture of VPM

The architecture of VPM is show in figure 5.1. VPM is able to run multiple virtual machines with different virtualization technologies. In the figure 5.1, the VM1 runs in para virtualized mode and VM2 runs in full virtualized mode. At the host machine, the PM is connected using memory interface. VPM leverages two-dimension address translations like nPT to transparently map the guest PM pages to host PM pages.

5.1.2.4 Managing Persistency in Full-Virtualization

VPM ensures all memory writes to virtualized PM will be applied to physical PM. It is done by write protecting the nested page tables. Hence, all writes to guest PM pages that are originally stored in a non-volatile storage, causes a trap, and remaps the corresponding pages to physical PM. The mapping s determined by the access pattern of guest virtual machine, and is detailed in 5.1.5.1.

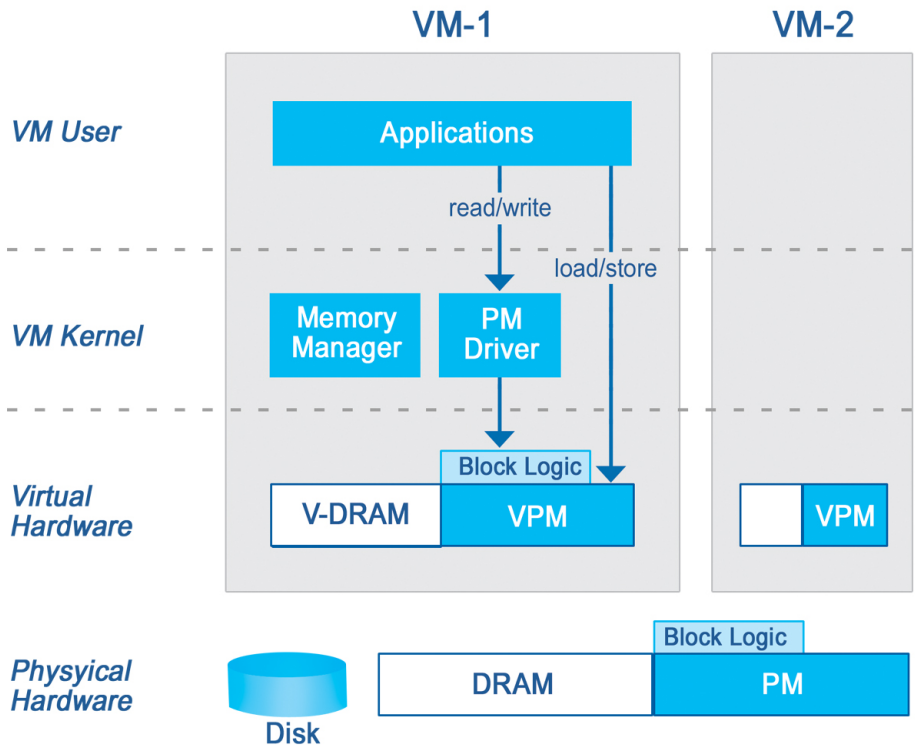

Figure 5.1: Architecture of virtualized persistent memory in virtual machine environment

5.1.3 Para-Virtualization of Non-Volatile Memory

Para virtualization can provide optimal performance since it is aware about the guest semantic information. The VPM, [10], provides a modified version of existing memory virtualization technique by creating new hypercalls. The table 5.1 lists all of newly added these hypercalls. The hypercall `vpm_persist` notifies that a specific guest PM pages to be persisted. When this hypercall is invoked, the VPM moves the page from non-volatile storage to persistent media, and updates nPT. The `vpm_barrier` call is used to set up a barrier till previously issued `vpm_persist` completes, and the data gets persisted.

Table 5.1: Hypercalls for virtualizing non-volatile memory

`vpn_persist`	notifies hypervisor that a specific range of guest PM is to be persisted
`vpn_lock` / `vpm_unlock`	lock & unlock the content of a specific range of guest PM to avoid it being modified by the hypervisor
`vpn_barrier`	persist all prior flushed guest PM to physical PM

5.1.4 Supporting Crash Recovery

Crash recovery is provided by persisting the key states regarding virtualizing PM across power failures or system crashes. Also, the operating systems running inside the virtual machines should have its own mechanisms to recover the data by leveraging states accessible from guest PM.

The key states required to persist across system fails are given below.

1. The nested page table and reverse mapping table containing host PM address to guest PM address mapping.
2. Memory tracking information are also preserved to enable the continuous access prediction and improve wear leveling.

However, those states cannot be directly stored on PM since they are updated rather frequently. Hence, the VPM uses a separate data structure in PM to book keep the mappings, and used DRAM to store nPT for accesses by CPU. It uses a write-ahead style logging to ensure consistent crash recovery: before updating the nPT, VPM firstly performs the corresponding page allocation and content migration, then updates its mapping structure stored in PM and lastly applies the updates to nPT.

The hypervisor does not explicitly perform data recovery. Instead, on every startup of guest VMs, the hypervisor will automatically recover data either from disks or PM, according to the mapping information, stored in PM, which the PM nPT will also be populated accordingly

5.1.5 Consolidating Persistent Memory

The VPM efficiently tracks and predicts the access characteristics of guest VMs to consolidate virtual PMs into less physical PMs. VPM then leverages the tracking and prediction information to transparently remap pages to save PM or improve performance.

5.1.5.1 Tracking Persistent Memory Access and Page Migration

VPM leverage *access bit* and *dirty bit* in nPT to track working set information. This approach avoids the expensive frequently trapping to get working set in formations. VPN scans the nPT entries periodically, and records those bits. It also clear such bits to rescan them later in next period. Since it requires frequent sampling, it may incur large overhead on performance. Hence VPM only samples the write access.

Page displacement is done by remapping from PM pages to different storage media on the fly. VPM achieves this by dynamically changing the mappings on nested page tables. The swapping pages from physical PM requires the page the page need to be persistent on disk(or SSD).

PM Tracking for Full Virtualization:
The tracking process invokes VM-Exit for every 500 ns. VPM employs a two-level scanning to avoid scanning all the entries for each round. The first level scans the hot pages, and the second level scans the relatively cold pages in clock-like manner. The hot pages are scanned in each round. Page entries will be moved between the two levels according to the frequency that it is modified.

The EPT-violation are handled by allocating native PM since it requires that all updates are applied to physical PM, which under the worst situation, needs to reclaim an already allocated PM Page by unmapping it from the page table. The page for unmapping is selected using CLOCK algorithm.

PM Tracking for Para Virtualization: Access tracking for para virtualization is relatively easy since the tracking information can be directly obtained when processing `vpm_persist` APIs. VPM shares with each guest VM a 128MB non-volatile memory metadata region containing the lock section, and ring queue section. The lock section holds locks for each guest PM pages, while the ring section holds `vpm_persist` requests. On processing `vpm_persist` requests, the corresponding lock is acquired, copies the content into non-volatile media, and releases the lock. On processing `vpm_barrier` requests, VPM flushes the whole ring queue for this VM.

5.2 Implications at Hypervisor level

5.2.1 Hypervisor-Based Persistence for Virtual Machines

In data centers, most common system failures are due to power outages and it causes long down time. It violates the service level agreements, and causes for loss of run time data. The data loss can be mitigated by applying various techniques like logging and check pointing. However, these techniques requires additional resources, and delay recovery. Protective technologies, which specifically address power outages, such as uninterruptable power supply (UPS) can prevent the loss of run time data, by extending the time until the inevitable shutdown. But, once the power returns, it requires a time consuming system restart.

NV-RAM can retain in-memory data without an external source of power. The benefit of a system equipped with NV-RAM lies in the ability to store all run time data of an application inside persistent memory without a significant decrease in performance. Hence after system restart all run time data is still available without the need of time-consuming data retrieval from background storage or the rerun of requests.

However, there are many challenges for implementing persistent virtual machines. Most important challenges is all CPU state such as registers and caches are volatile and will be lost in case of a power outage. NV-Hypervisor[11] address this challenges by detecting the power outage before the system fails, and persists the required states. Temporality, a NVRAM-based virtualization platform, [12] provides better memory management techniques, and provides persistent memory consolidation.

5.2.1.1 Architecture of NV-Hypervisor

Figure 5.2 illustrates the architecture of NV-Hypervisor. The minimum persistent executable unit in Temporality is a VM, and a VM an be subdivided into the following components: a RAM image of the guest VM, a CPU state and virtual device states. RAM image saves all run time data of VM. The CPU state represents the current state of CPU. The virtual devices are described as independent stateful modules that communicate via the virtual bus. The state of the virtual devices and CPU can be combined as a virtual environment.

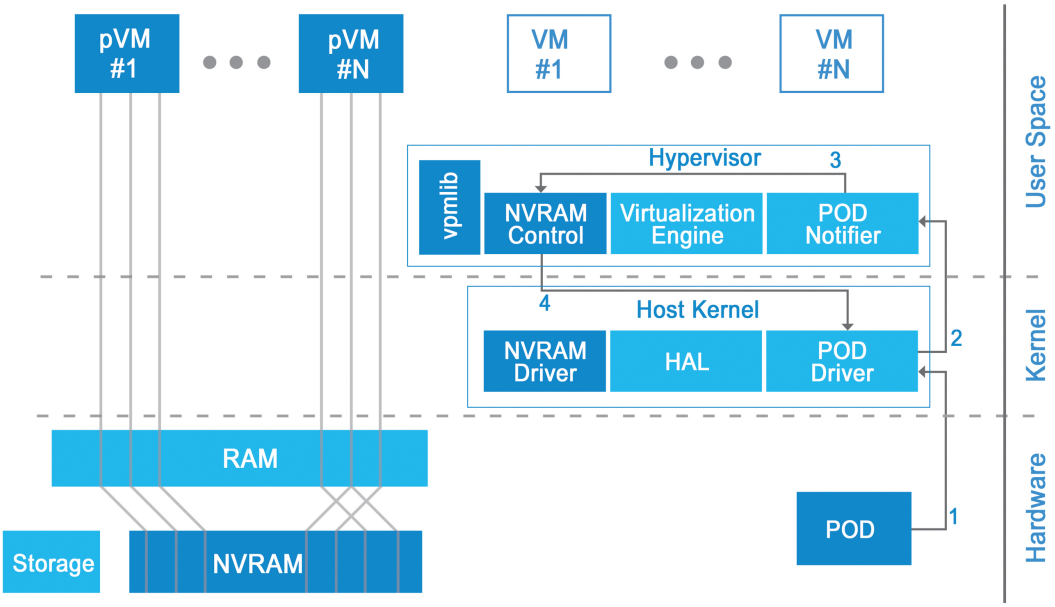

Figure 5.2: Architecture of *Temporality*

Temporality uses a power outage detector(POD) to detects the power failures. This detector measures the current voltage of the external power supply and generates an interrupt if power fails. The capacitors provide residual energy for generating stable voltage between 40 to 60 ms after detecting a power outage. This residual energy contained in capacitors of the power supply is used to save relevant volatile CPU state to NVRAM after detecting a power outage.

5.2.1.2 Creation of Persistent Virtual Machine

Creation and startup of a persistent virtual machine is very similar to an ordinary volatile virtual machine but requires two modifications:

1. For virtual machines which requires persistency, The memory is allocated in NVRAM.
2. Information about the assigned memory and other state needs to be made persistent to enable recovery.

NV-Hypervisor uses mmap and unmap system calls to allocate and deallocate memory for virtual machines. While allocating and deallocating memory, the mappings are also persisted on NVRAM.

5.2.1.3 Handling of a Power Fault

The power outage is handled as follows:

1. The power outage detector(POD) rises a non-maskable interrupt, once the drop of the input voltage is detected.
2. The interrupt is caught by the POD-driver, which notifies the NV-Hypervisor about the upcoming power outage.
3. The NV-Hypervisor saves volatile state of virtual devices and virtual CPU states of persistent VMs to NVRAM.
4. The POD-driver regains control and stops any memory operations, freezes CPUs and flushes the caches.

5.2.1.4 Recovery After Power Outage

The virtual environment will be made persistent before system fails. The host operating system boots like a normal system with out support for NVRAM. Once the operating system and NV-Hypervisor are up and running, NV-Hypervisor retrieves the information about the persistent virtual machines which was running at the time of power failure. Then it reassigns the memory and integrates the virtual device information. The recovery procedure is executed as follows:

- Hypervisor creates an empty VM. The empty VM does not have any virtual memory, and it has a default state for the virtual environment
- Hypervisor connects all regions of persistent memory, which where mapped to VM before the power outage, to VM's address space.
- Later, hypervisor replaces the default virtual environment of VM with state that is persisted before power outage.

5.2.1.5 Consolidating Non-Volatile Memory for Persistent-Virtual-Machines

NV-Hypervisor does not provides any specific memory management service for NV-RAM. When persistent VMs are restored all virtual memory regions need to be reconnected to the exact same locations as assigned before the shutdown. Temporality extends the NV-Hypervisor with capabilities to allocate and restore memory with over-commitment. The memory allocation and deallocation to guest VM achieved by existing `mmap` and `munmap` system call. But at the time of memory allocations, hypervisor stores the memory states in persistent memory.

The over-commitment is achieved by lazy allocation of memory pages to guest VM, and coupling it with swapping. When VM is created, no physical page are allocated for VM. Instead, it creates

a flat region without any permissions in the virtual memory by means of the `mmap` system call. Hence, a page fault is generated when the VM tries to access the page. The page fault handler assigns the first free page in NV-RAM to accessed virtual memory page, and updates persistent page table stored in NVRAM. When NVRAM is fully occupied, the page fault handler swaps out a page from NVRAM to swap file, unmaps the swapped-out page from virtual memory, and maps it to the currently accessed virtual memory page.

6. Conclusion and Future Work

This seminar addressed the broad impacts of emerging NVRAM technologies on operating system and applications design. Currently, the performance the performance of available NVRAM technologies are almost similar to DRAM performances. Upcoming memory technologies will soon provide a new way for applications to store persistent data at near DRAM speeds.

The energy footprint of non-volatile memory technologies impacts significantly on modern computer system design since, the existing DRAM consumes more power for refreshing the data, and are are already reached their extent on scalability. However, for some technologies, the write energy is much larger than that required in DRAM.

Another challenge of using NVRAM technologies as main memory is their low endurance. Hence, the NVRAM-only system are not good for running applications with frequent updates to memory. Avoiding problems with endurance will require some intelligence to decide what pages should migrate to NVM, and when; the OS appears the best place to apply that intelligence. As a solution, different types of hybrid main memory system containing large amount of NVRAM and small fraction of DRAM are already proposed. But the design of memory systems containing combinations of NVRAM technologies with intelligent page placement and migrations are not explored well.

Now, the cloud computing is the plat form for many applications, with different requirements. Hence, the cloud providers have to start exposing NVRAM for the applications that are hosted cloud platforms. Researches already proposed multiple ways for exposing and managing the NVRAM in full and para virtualized environments. Due to light weight execution environment, the containers are also getting more popularity than virtual machines. But, the implications of NVRAM in container environment are not well studied, and it will be an interesting research problem.

The following section details the future works possible using non-volatile memory.

1. **Virtualizing non-volatile memory for container**
   Directly attaching the non-volatile memory to containers, and allowing the applications to manage the it, does not exploits all the of features of non-volatile memory. Also, different persistent memory technologies exhibits different access patterns and physical properties. Hence, it is critical to consolidate persistent while virtualizing to containers to maximize cost-efficiency, performance, and endurance. But the current memory virtualization lacks support for an appropriate abstraction of persistent main memory.

2. **NVRAM-only memory system with combination of multiple technologies**
   The different non-volatile memory technologies exhibits different performance and price characteristics. Some technologies provides higher read and write latency with a energy. Some other technology consumes less power, but their endurance will is very small. Hence memory system with multiple technologies can overcome drawback of individual technologies with intelligent placement of memory pages. Many researches already proposed methods to identify application behaviour and access pattern, and to dynamically moving page across DRAM and NVRAM. Their researches considered only one or two parameters. So it will be interesting to experiment on NVRAM-only systems by considering different parameters like access latency, endurance, and energy.

Bibliography

[1] An Chen. **A review of emerging non-volatile memory (NVM) technologies and applications**. Solid-State Electronics. Volume 125, November 2016, Page 1

[2] Sparsh Mittal and Jeffrey S. Vetter. **A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems**. IEEE Transactions on Parallel and Distributed Systems, Vol. 27, No. 5, May 2016 1537.

[3] Haris Volos, Andres Jaan Tack, and Michael M. Swift. **Mnemosyne: Lightweight Persistent Memory**. ASPLOS11, March 511, 2011, Newport Beach, California, USA.

[4] Jaemin Jung and Youjip Won. **nvramdisk: A Transactional Block Device Driver for Non-Volatile RAM**. IEEE Transactions on Computers, Vol. 65, NO. 2, February 2016.

[5] Ziqi Fan, Fenggang Wu, Dongchul Park, Jim Diehl, Doug Voigt, and David H.C. Du. **Hibachi: A Cooperative Hybrid Cache with NVRAM and DRAM for Storage Arrays**. 34th International Conference on Massive Storage Systems, and Technology (MSST 2018), May 14 18, 2018.

[6] Takahiro Hirofuchi and Ryousei Takano. **RAMinate: Hypervisor-based Virtualization for Hybrid Main Memory Systems**. Proceeding SoCC '16 Proceedings of the Seventh ACM Symposium on Cloud Computing, Pages 112-125.

[7] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. **Data tiering in heterogeneous memory systems**. Proceeding EuroSys '16 Proceedings of the Eleventh European Conference on Computer Systems, Article No. 15.

[8] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. **Scalable High Performance Main Memory System Using Phase-Change Memory Technology**. Proceeding ISCA '09 Proceedings of the 36th annual international symposium on Computer architecture Pages 24-33.

[9] Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. **Exploiting NVM in large-scale graph analytics**. Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads, Article No. 2.

[10] Liang Liang, Rong Chen, Haibo Chen, Yubin Xia, KwanJong Park, Binyu Zang, and Haibing Guan. **A Case for Virtualizing Persistent Memory**. SoCC '16 Proceedings of the Seventh ACM Symposium on Cloud Computing, Pages 126-140 .

[11] Vasily A. Sartakov and Rdiger Kapitza. **NV-Hypervisor: Hypervisor-Based Persistence for Virtual Machines**. 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.

[12] Vasily A. Sartakov, Arthur Martens, and Ruediger Kapitza. **Temporality a NVRAM-based Virtualization Platform**. 2015 IEEE 34th Symposium on Reliable Distributed Systems.

[13] Xu Li, Kai Lu, Xiaoping Wang, and Xu Zhou. **NV-process: A Fault-Tolerance Process Model Based on Non-Volatile Memory**. APSys '12, July 23-24, 2012, Seoul, S. Korea.

[14] https://en.wikipedia.org/wiki/Non-volatile_memory

[15] http://www.radio-electronics.com/info/data/semicond/memory/pram-phase-change-random-access-memory-basics-tutorial.php