ABSTRACT

BUDHIRAJA, RICHA. Supporting Position Independence in NVM and Porting New and Legacy C Code to NVM. (Under the direction of Dr. Xipeng Shen.)

The current programming paradigm was developed keeping in view that processor's main memory is volatile and byte-addressable whereas the data on storage devices is accessible in chunks of blocks. However, with the emergence of byte addressable non-volatile memory technologies, the existing programming paradigm will need to be changed in order to fully utilize the benefits of byte addressable persistent memory. One such change would be the use of relative pointers instead of absolute pointer references, in order to have consistent object references across different executions of a program written on persistent memory. This work explores various methods for offering such position independence on Non-Volatile Memory, investigates the challenges for equipping legacy C code with such support, and also presents some compiler tools for easing the manual porting efforts.

© Copyright 2016 by Richa Budhiraja

All Rights Reserved

Supporting Position Independence in NVM and Porting New and Legacy C Code to NVM

by
Richa Budhiraja

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Science

Raleigh, North Carolina

2016

APPROVED BY:

Dr. Xipeng Shen Committee Chair	Dr. Guoliang Jin
Dr. Huiyang Zhou	

DEDICATION

To my parents and my brother.

BIOGRAPHY

Richa Budhiraja was born in Delhi, India. She has obtained her Bachelor's degree in Electronics and Instrumentation from Uttar Pradesh Technical University in 2012. She has worked as a Systems Engineer at Infosys Ltd., Bangalore for a year (2013-2014), after completing her four months training at Infosys Ltd., Mysore (2012-2013). She was admitted to NC State in the Fall of 2014 as a Master's student to the Department of Computer Science, and has worked under Dr. Xipeng Shen as a Research Assistant since Summer 2015.

ACKNOWLEDGMENTS

I am highly grateful to my thesis advisor Dr. Xipeng Shen, for giving me the opportunity to work in his research group, and for guiding me throughout the course of my thesis work. I learnt a lot about compilers and systems, and about research in general through this experience. I am also thankful to Dr. Youfeng Wu for his insightful reviews and guidance throughout this thesis work.

I am thankful to my thesis committee members, Dr. Guoliang Jin and Dr. Huiyang Zhou, for their support in my research work.

I am thankful to my colleagues for being so helpful and inspiring. And, I am also very thankful to my roommates, friends and family for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

1.1 Background

A computer's memory hierarchy consists of many layers, and each layer has its own distinct set of features from the layer above or below it. The difference can be in terms of the memory capacity, the process technology and the material used for making the memory cells, the latency of reads and writes by the processor from and to the memory layer, the bandwidth provided by the bus interconnects, the duration of time it is expected to hold on to the values stored on the memory, and the wear and tear associated with that memory technology.

The memory hierarchy, for many years, in almost all of the systems, has more or less looked like as shown in Figure 1.1.

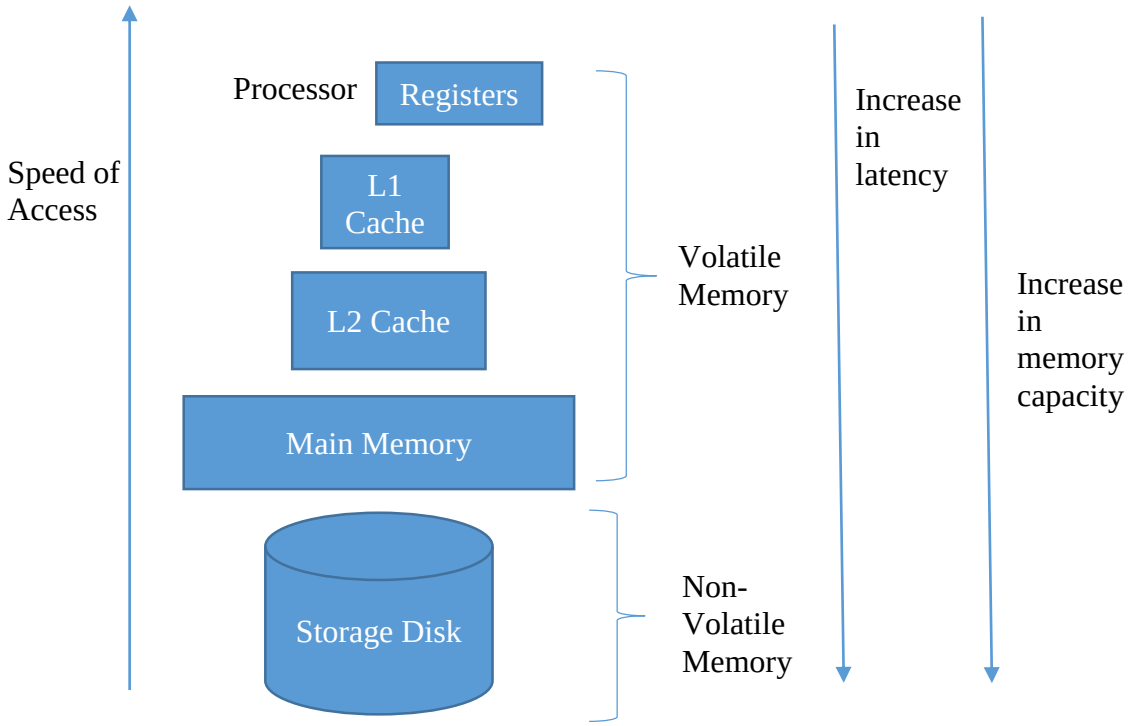


Figure 1.1 A general view of memory hierarchy

As, can be seen from the above figure, the last level of memory hierarchy consists of the storage devices which are non-volatile and have a considerably larger size as compared to the volatile memory layers. However, the drawback here is the time taken to fetch data from the storage devices to the processor for processing is also quite large.

The volatile memory layers on the other hand, have much lower latencies for read and write as compared to the storage devices, but are volatile in nature. That is all the information stored on the volatile memory is lost, as soon as the power supplied to such memory devices is switched off.

For a long time, the storage devices had mainly consisted of the hard disk drives. The time taken while reading from or writing to such devices depends on three factors:

1. Seek Time, which is the time it takes the head to reach the requested track.
2. Rotational delay, which is the time it takes for the head to reach the sector from which the data has to be read.
3. Transfer Time, which is the allowed bandwidth for data transfers from/to the disk.

Hence, being an electromechanical device, it is prone to a lot of delays for performing read and write operations.

Recently, solid state drives which consist of integrated circuits for storing the data persistently, and which also offer a very low read/write latency as compared to the hard disk drives, have been employed in computers as storage devices.

However, more recently technologies such as PCM [1], STT RAM [2], and RRAM [3] are being developed, which in future will be able to support very high memory capacities. These new storage devices have the combined benefits of the two types of memory earlier seen in the memory hierarchy:

1. They are non-volatile, and
2. The read and write latency is very small compared to the existing storage devices.

Given such a memory device, which has a high memory capacity, has less latency of access and is capable of storing the data in a persistent manner, there can be a number of ways in which such devices can be utilized in order to make applications more efficient. Some of

the programming models based on how NVM is used in the system are discussed in [4]. Two of the ways in which NVM can be used in devices are:

1. Use the non-volatile memory as a storage device just like how they have traditionally been used. This use will require the usual block device driver to read from or write data to the memory device in chunks of blocks. This also makes it easy for the existing applications to port from using the existing storage devices to using the upcoming NVM technology based storage devices.
2. Another very interesting use of such devices would be to have these non-volatile memory devices shown as main memory to the processor, as shown in Figure 1.2. The processor would be able to access such devices through load/store instructions, without going through a number of interface layers, as is done when accessing data on a traditional non-volatile memory device.

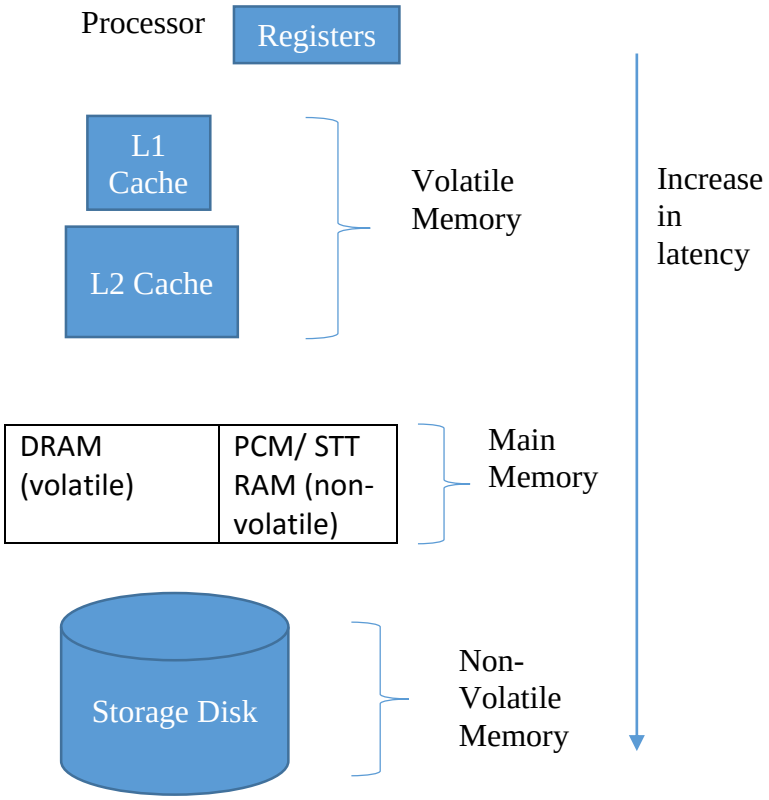


Figure 1.2 The New Memory Hierarchy

When a non-volatile memory will be used as a main memory to the processor, some changes will be required in the system software such as Operating Systems to incorporate this new memory hierarchy, because how the processor interacts with such a non-volatile memory will be different than how it used to previously interact with the non-volatile devices or the main memory for that matter as discussed in [5] [6]. The programming paradigm will also need to be changed to take into consideration the new memory hierarchy. For example, the new programming paradigm will have to provide support for:

1. Allocation, Re-allocation and De-allocation of objects on persistent memory.
2. Access to data on persistent memory.
3. Position Independence for object reuse.
4. Atomicity, Consistency, Isolation and Durability (ACID) properties which are required in a traditional transactional system.
5. Concurrency for making sure that the data can be accessed by multiple threads concurrently without making the data inconsistent.
6. Legacy code, i.e., the new programming paradigm should make it easy for the existing applications and programs to be transformed such that they can utilize the new memory hierarchy for better performance.

This list is only a small subset of issues that will need to be considered while programming on top of such a memory hierarchy. This thesis work addresses the 3rd and the 6th point in the list mentioned above. This thesis work will explore the various techniques for supporting position independence in the new non-volatile memory technologies and will also discuss the compiler tools which can be used for porting C programs to NVM, by introducing position independence in them.

In order to address the challenges that will be faced while programming on a new memory hierarchy, many researchers in academia and in industry have been working on the development of libraries that will provide support for ease of programming on such systems. Some of these libraries are NVML [7], NV Heaps Library [8], Mnemosyne Library [9]. As part of this thesis, the implementation of the various methods for supporting position

independence is done on NVML library, and hence it will be discussed in detail throughout this document. The thesis will also briefly discuss the other two libraries mentioned above.

1.2 Problem of Position Independence

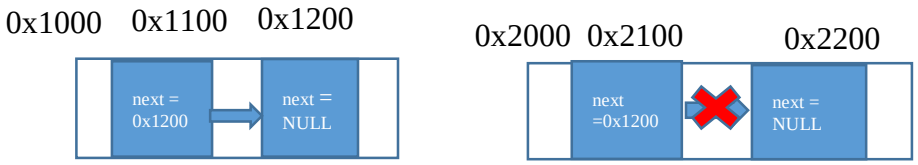
The paradigm used by the libraries which provide programming support on NVM, for exposing NVM to the application process, is to mmap(2) the files existing on NVM devices into the address space of a process. So, we can think of non-volatile memory as consisting of a number of NV Regions [10] as seen in the virtual address space of a process. These NV Regions will have a base address which will serve as the starting address of the NV Region, and also a unique identification number to identify the NV Region across different runs and processes. The NV Region can also have a name associated with it. Each NV Region will have a root object through which all other objects existing in the NV Region, can be reached.

Now, since in order to protect a system from buffer overflow attacks, a technique called the Address Space Layout Randomization (ASLR) is employed in many systems, which maps various sections such as stack and heap, of a user's address space randomly at different virtual addresses at the time of a process's execution. Hence, there will be no guarantee that the NV Regions are always mapped at the same virtual address space of a process, i.e. the starting address of an NV Region will always be different for different processes that will map it, and also this mapping will differ across different executions of the same programs.

Hence the problem of position independence arises when objects created in an NV Region, contain references to other objects also on same/some other NV Region, are later reused across different runs and/or different processes.

For example, suppose a process creates a linked list and stores the nodes in an NV Region, it has created. Let's assume the NV Region has been mapped in the program's address space at 0x1000 and the program creates a linked list with the first node at address 0x1100 and the next field of this node points to a node at address 0x1200. Then suppose, the program exits and during its next execution, the program opens the same NV Region to reuse the objects it had created (and stored in NVM) in the previous run. However, if the NV Region gets mapped

to a different address, say address 0x2000, the program might crash due to the presence of incorrect pointer values. As can be seen from the diagram, in the second run, the first node's next field still points to address 0x1200, but the second node is actually at 0x2200.



a) Run 1 b) Run 2

Figure 1.3 Problem of using absolute address references on NVM

Hence, supporting position independence in non-volatile memory, basically means to ensure that the pointers in any object point to correct locations even when they are loaded at different virtual addresses in the memory.

The above example only states the problem in terms of intra-region pointers. However, we can also have cross region pointers, in which the pointer and the object lie in different NV Regions. Hence, there is a need to support position independence in this scenario also.

1.3 Previous Work

1.3.1 Serialization-Deserialization

In order to store objects to a file on disk, a technique called serialization-deserialization of objects is employed. Serialization of objects is performed when the objects have to be written to a storage device, in which case the objects are converted into stream of bits. While reading back from the storage and into main memory, these streams of bits are converted back into the object by a process called the Deserialization of objects.

During the serialization of objects, all pointer references containing absolute addresses are first converted to a position independent value and then the object is stored to the file. This can be done by either,

1. Storing the offset from the start of the file to the object.
2. Assigning unique identifiers to the objects and then referring to these objects using the identifiers instead of using the addresses.

This transformation, from an absolute address to a position independent value, is called pointer unswizzling. The opposite of unswizzling is called swizzling [11] of pointers, which is performed during the deserialization of object, i.e. when the objects are read from the storage and into the memory. In this step, the offsets or the unique identifiers are converted back to absolute virtual addresses.

Serialization-Deserialization has an overhead on the performance of an application which reads objects from and writes objects to the non-volatile memory quite often. Such an overhead, though may easily be hidden during the fetch operation from slow storage devices due to their access latency, it might lead to inefficient performance in the emerging non-volatile memory systems.

1.3.2 Based Pointer

The Microsoft's C/C++ compiler provides a __based keyword [12], which is used to declare pointers that are at a certain offset from some other pointer. Example usage as provided by Microsoft:

```
void *vpBuffer;
struct llist_t {
  void __based( vpBuffer ) *vpData;
  struct llist_t __based( vpBuffer ) *llNext;
};
```

Here, the pointer variable vpBuffer contains the base address and all the other pointers used in the linked list structure, will hold an offset value from the address value being held by

vpBuffer. As can be seen from the above example, the base address of the list objects is part of the type of the objects.

1.3.3 Fat Pointer

The fat pointer is basically a data structure which has two fields; one for carrying the base address of the NV Region and another for carrying the offset from the base of the NV Region. In this case, there cannot be any cross-region pointers. A slightly modified form of the fat pointer, which supports cross region pointers, was being used until recently in the libpmemobj library of the NVML libraries, in which the structure consists of the Region ID of the NV Region and the offset from the base of the NV Region to the address of the object. Now, the library is using a smart pointer to support position independence.

1.4 Overview of New Methods

Given below are the two new methods for supporting position independence that are described in [10], and will be discussed in greater details in the later chapters of this thesis.

a. Off-holder Method: In this method, the pointer carries the offset from the address of the pointer to the address of the target object. This method can only support intra-region pointers, as during different runs of the program, the NV Regions will be mapped at different addresses and hence the absolute address calculation from the offset values may fail.
b. Region ID in Value or the RIV method: In this method, the Region ID of the NV Region as well as the offset from the base of the NV Region to the target object is stored within the 64-bit value of the pointer. This method involves a table look up at run time to fetch the base address of the NV Region, given the RID of the Region. The base address is then added to the offset value contained in the pointer to form the absolute address. This method will support both inter as well as intra-region pointers.

This thesis will evaluate the performance of the four different methods of supporting position independence mentioned so far, i.e.

i. Based Pointer

ii. Fat Pointer

iii. Off-holder

iv. RIV

1.5 Related Work

As mentioned in section 1.1, there are two other libraries that provide APIs for programming with persistent memory. In this section, we will briefly mention how these two libraries deal with the problem of position independence.

1.5.1 Mnemosyne library

Mnemosysne library [9] introduces the semantic to declare global persistent variable with the keyword “pstatic” and also provides a way to dynamically allocate persistent data. However, their work does not support position independence of the pointer references and makes sure that the persistent memory regions are mapped to the same virtual address ranges in subsequent runs also.

The library provides a keyword “persistent” to annotate that a pointer is pointing to a persistent target. The paper mentions that this keyword has similar use as that of “const” keyword. Their annotation however does not follow the rule of orthogonality [13] in that, annotating a target as persistent does not mean that all its members will be persistent. The keyword is provided so that there will not be any assignment operations between volatile pointers and non-volatile pointers as these kind of assignments are considered as error prone by the authors.

The paper also mentions the shortcomings of having a non-volatile pointer pointing to a volatile object or a variable, as the volatile objects will be meaningful only during the first execution of the program and will contain inconsistent or incorrect values during the subsequent execution of the program. Their keyword hence, also helps in removing these kind of errors.

Lastly, Mnemosyne library uses smart pointers, for garbage collection through reference counting.

1.5.2 NV-heaps library

The NV-heaps [8] enumerates the four types of pointers that will exist in a program written to execute on the new memory hierarchy. These are:

1. Volatile Pointers to Non-Volatile Targets
2. Non-Volatile Pointers to Volatile Targets
3. Non-Volatile Pointers to Non-Volatile Targets in the same NV Region
4. Non-Volatile Pointers to Non-Volatile Targets in different NV Regions,

And of-course there will also be volatile pointers to volatile objects. The authors claim that both the second and fourth kinds of pointers are error prone and should be avoided. They also claim that the pointers from non-volatile region to volatile region will be rendered meaningless and useless after the first run of the program.

In summary, the reasons provided by [8] for avoiding the fourth kind of pointer, which is the inter-region or the cross region pointer are:

1. Program can crash if there is a pointer from one NV Region to another NV Region, say NV Region A to NV Region B. Then, if NV Region B is closed or deleted then there will be a stray pointer in NV Region A which will be pointing to a non-existent object now. And this might lead to run time errors.
2. The cross region pointers will lead to a much complicated garbage collection process, because at any point of time, we will not be able to discern whether there is a pointer variable in some other region that is pointing to a particular object in, say, NV Region A.

Although it is understandable that both the situations can cause run time errors, but in some of the cases we need to have cross region pointers. For example, we might need two different NV-Heaps or NV Regions, which may or may not contain similar data, and can have different permission rules applied to both the heaps. The application may require access to objects in NV-Heap A through objects in NV-Heap B. Also, if the NV-Regions or the NV-Heaps are defined to be of a fixed size, then we may need another NV Region

for allocating more objects, if an earlier NV Region reached its full capacity. The overflowed NV Region will need to have access to all the other NV Regions which were created to handle this overflow. Hence, we will need to have cross-region pointers in this case also.

And there will be a need to have Non-volatile to volatile pointers because for some of the calculations, we might need some dummy objects whose life time will need to be only for a loop or for a function. In such cases, assuming that the writes to persistent memory take longer time and that there will be some wearing costs associated with the non-volatile memory, it is better for the performance of the application to have pointers from Non-volatile to volatile memory. Also, there will be some structures which should be ephemeral and new objects of which are required to be created on each instantiation of the application, such as the database connection objects, the locks that are needed to be acquired on in memory data structures such as Btree for DBMS applications, etc.

Hence, we will need cross-region pointers, as well as NV Region to volatile target pointers as well, however we will need to make sure that before or while closing the NV Region, the pointer from NV to volatile regions are all equated to NULL, so that the future execution of the program will not lead to run time errors. And as mentioned above, we will need sophisticated methods for handling garbage collection when we have inter-NV Region pointers.

The NV-Heaps library uses relative pointers for NV to NV and also for V to NV pointers, which are 128 bits long. The relative pointers used in this library have a 64-bit relative pointer and dynamic type information to prevent assignments among mismatched pointer types.

Chapter 2
Support for Position Independence on NVM

2.1 Introduction to NVML library

NVML library [7] [14] consists of six different libraries which provide an interface to the programmers for programming on persistent memory. These library provides support for memory management, transactions, logging, etc., for programs that will run on persistent memory. These six libraries are libpmemobj, libpmemblk, libpmemlog, libpmem, libvmem and, libvmmalloc.

This thesis work has used libpmemobj library to implement the various methods of supporting position independence. While using libpmemobj library for executing a program, it is not necessary for the persistent memory to be present in the machine. The library can discern at the initialization of the library about whether the file being mapped to the user space belongs to a persistent memory aware file system or to a traditional disk file system. If the file belongs to a persistent memory aware file system, writes to the persistent memory are made consistent by issuing ‘clflush’ instruction calls whereas for a normal file system ‘msync’ instruction calls are issued.

2.1.1 Terminology

The SQLite [15] source code is also transformed based on how this library is used for programming with persistent memory. Hence, it is necessary to go through some of the common terms which are specific to this library and will be used frequently in the subsequent sections. This section enumerates some of these terms.

a. Pools: The persistent memory is exposed by the OS as memory-mapped files. And these memory-mapped regions are termed as pools. Each pool has a unique identifier, called the pool ID associated it. The libpmemobj library uses linux’s uuid_generate function to generate a unique pool ID for each pool created. The uuid_generate returns a 16 bytes or

a 128 bits long value, which is then XORed (in a first byte with last byte manner) to produce a 64-bit value, which they use for identifying the pool.

They can also be thought of as NV Regions, which were mentioned in section 1.2. The terms pool and NV Regions and hence pool ID and Region ID may be used interchangeably throughout the document.

This library provides an API for creating, opening and closing a pool. The pool handle returned by the pool creation/open functions is used while allocating objects in that pool.

b. Layout: All programs which use this library should define a layout, which will store the information about various types that are being used in the program, which is needed to generate typed persistent pointer types that will be used in the program. Section 2.1.3 discusses the typed persistent pointers in detail. Given below is an example of how the layout is declared in a program:

```
POBJ_LAYOUT_BEGIN(example_store);
POBJ_LAYOUT_ROOT(example_store, struct my_root);
POBJ_LAYOUT_TOID(example_store, struct Index);      // for fat pointer method
POBJ_LAYOUT_END(example_store);
```

Here, the “example_store” is the name of the pool layout, and ‘struct Index’ is a type that is used in the program, and objects of this type are allocated on heap, during the program’s execution.

c. Root: The object through will all the other objects stored in a pool can be reached.

2.1.2 Allocation of Objects

The libpmemobj library can be used to either perform transactional allocations and reads and writes need to be part of a transactional block, or there can be atomic or non-transactional allocations. In this work, only atomic allocations are used and transaction blocks are not used for the read and write operations, as the main goal of this thesis is to evaluate the performance of the methods for supporting position independence.

In the atomic or non-transactional allocations, two lists are maintained. The object is first allocated to a transactional list, and initialized to either zero or the values provided in the

constructor function for the object. If the allocation and the initialization are both successful, the object is moved to a user list, which actually is the list that contains all the objects of a particular type and the objects of a particular type can be reached using those lists. Each type used in a program will have a corresponding list in the persistent memory pool.

2.1.3 Consistent Writes

There are two ways in which the programmer can make consistent writes to the persistent memory:

i. By flushing the cache contents after every write
ii. By using libpmemobj's transaction API, which manages the flushing for the programmers.

Which flush statement should be used, is discerned at the initialization of the pools, and it depends on whether the memory on which the libpmemobj library is working, is really persistent memory or not.

2.1.4 The Typed Persistent Pointer

The libpmemobj library provides fat pointer as a way to support position independence in their library. Fat pointer is basically a structure defined as:

```
typedef struct pmemoid{
        uint64_t pool_uuid_lo;
        uint64_t off;
}PMEMoid;
```

Here, the pool_uuid_lo is the unique identifier of a pool in which the objects will reside. The field named 'off' is the offset from the base of the pool to the location where a particular object resides in a particular pool, identified by pool_uuid_lo.

However, only using this structure to support position independence would not work, as there is no information about the type of object to which this fat pointer will be pointing to. For example,

PMEMoid btree = pmemobj_alloc(pop, struct Btree, sizeof(struct Btree), NULL, NULL);

PMEMoid index = pmemobj_alloc(pop, struct Index, sizeof(struct Index), NULL, NULL);

Then, there will be no compilation error, if the pointers are assigned to each other like this:

```
btree = index;
```

The above assignment however will lead to run time errors, if the code tries to access fields of Btree object. Also after translating a fat pointer to an absolute address pointer, the programmer will explicitly have to associate the type with the address, so that the object's fields can be accessed. For example:

```
PMEMoid index = pmemobj_alloc(pop, struct Index, sizeof(struct Index), NULL, NULL);
Struct Table *tab = ((struct Index *)pmemobj_direct(index))->pTab;
```

Here, the pmemobj_direct is a function that converts the fat pointer to an absolute address at which the object is situated.

To summarize, we need to associate a type with the fat pointer:

1) To generate compile time errors for incompatible types
2) To facilitate easy conversion from fat pointer to a particular type pointer carrying the absolute address.

In order to meet these two requirements, the library uses named unions, in which the fat pointer, as well as the type of the pointer is embedded. These named unions however need to be declared first before they can be used in the program to declare typed persistent pointers.

The macro TOID_DECLARE is used for declaring the named union which is used as a typed persistent pointer. The macro TOID is used in code while declaring a variable which is of the typed persistent pointer type.

That is to say the TOID_DECLARE macro will create a type which will be of type union _toid_##type##_toid. Now variables of this type can be declared in user's program that will point to the persistent objects of the type mentioned in the macro. The library also associates a unique type number for each type used in the library, which is used when storing objects of a particular type in their specific lists. The TOID_DECLARE macro is defined as:

```
#defineTOID_DECLARE(type, type_num)\
typedef uint8_t _toid_##type##_toid_id[(type_num)];\
```

```
TOID(type)\
{\
        PMEMoid oid;\
        type *_type;\
        _toid_##type##_toid_id *_id;\
}
```

The declaration of such typed persistent pointer looks like this:

```
TOID_DECLARE(struct Btree, 1);
TOID_DECLARE(struct Index, 2);
```

Here, the numbers ‘1’ and ‘2’ are the type numbers that are assigned to type ‘struct Btree’ and to type ‘struct Index’, respectively. Another way to declare all the named union types that will be used in a program is to declare these types while defining the layout of a program, as described in section 2.1.1. For example:

```
POBJ_LAYOUT_BEGIN(my_layout);
POBJ_LAYOUT_TOID(my_layout, struct Btree);
POBJ_LAYOUT_TOID(my_layout, struct Index);
POBJ_LAYOUT_END(my_layout);
```

In this case, there is no need to explicitly specify the type number for the named union type that will be used in the program. The library will assign type numbers to these declared types in a consecutive manner.

There are two macros provided by the library that are used for translating a fat pointer to an absolute pointer. These are: D_RO and D_RW, which are used while reading an object and when writing to an object respectively. The library uses cuckoo hash tables for storing the pool ID and pool’s base address entries. These entries are looked up while translating the fat pointer to an absolute address. The library using two hash tables for this, and hence, the translation of the fat pointer to an absolute address requires computation of two hash functions for deciding which table entry to fetch.

Advantages:

1. Efficient lookup process for the base address of the pool using the pool ID contained in the PMEMoid structure, by using cuckoo hash tables.

2. Since, the typed persistent pointer carries with it the pool ID information, we can have cross region pointers, which may be useful in applications which require inter-pool object references.

Drawbacks:

1. The size of the typed persistent pointer is twice the size of a normal pointer, because it is essentially a struct variable. It occupies 128 bits or 16 bytes.
2. This type of pointer is not compatible with the 64-bits volatile pointers and hence a conversion is required.
3. There is a need to put a named union wrapper around the fat pointer and this named union should be declared before the typed persistent pointer can be used, which adds to the complexity of writing a program.

2.2 Implementation of the Methods

This section mentions describes two new methods which can be used for supporting position independence while programming on non-volatile memory. All the methods mentioned in this section do not need to be wrapped in named unions for using them as persistent pointers, since these are just 64 bit values, they can be easily assigned to pointer variables, which already have a particular type associated with them. As long as the programmer knows which variable is carrying which kind of value and remembers to translate the value to absolute address, the program will compile and execute correctly.

2.2.1 Off-holder

In an off-holder [10] pointer method, the pointer carries the offset from pointer's own address to the object's address. Hence, while translating the value held by an off-holder pointer to an absolute address of an object, the address of the pointer is added to the value contained in the pointer. While writing the offset, the address of the pointer is subtracted from the object's absolute address, to which it is pointing, and is stored in the pointer as an off-holder value. For Example:

Translating an off-holder value:

```
BtShared *pBt = p->pBt;
```

If p->pBt contains an off-holder value, then while converting it to an absolute address, we will have to perform,

```
p->pBt = &p->pBt + p->pBt;
```

Writing to an off-holder pointer:

While assigning a persistent object's address to an off-holder pointer variable:

```
p->pBt = pBt;
```

If pBt contains absolute address of a non-volatile object in the same pool as member variable p->pBt lies, then we will have to perform the following operations:

```
p->pBt = pBt - &p->pBt;
```

Advantages:

1. There is no need for the table lookup to find the base address of the NV pool or NV Region.
2. The size of the off-holder pointer is the same as the size of a normal pointer, i.e. 64 bits on a 64-bit machine.

Disadvantages:

1. While assigning the off-holder pointer value to a different off-holder pointer, the value first needs to be converted to an absolute form and then the new off-holder value is calculated by subtracting the L.H.S pointer address from the absolute object address before assigning the value to the L.H.S pointer variable. Hence, this conversion while writing the off-holder value will have an extra overhead.
2. Cannot point to objects belonging to different pools.

2.2.2 Region ID in Value or RIV

RIV method [10] supports position independence in a similar way the fat pointer method does, in that, RIV pointer also holds the offset from the base address of the NV Region/pool to the

address of the object and it also holds the Region ID or the pool ID, but it does that in a single 64-bit value, and hence, is called the Region ID in value pointer.

The 64-bit address of an object might be segregated according to this idea:

11…1	XXX…X	00000…0
l1	l2	l3

As can be seen in the text above, the value of the 64-bit pointer is divided into three parts. The first l1 bits are all ones and signify that the pointer is pointing to a non-volatile memory region.

The next l2 bits signify the base address of the pool in which the object is located. The last l3 number of bits will contain the information about the offset of the object from the base of the NV Region/pool, and hence basically gives the information about the size of a NV Region.

If the absolute address of an object is in the form described above, then the RIV pointer can have the value in this form,

00…..0	XXX…X	YYY…Y
(64-l4-l3)	l4	l3

Here the last l3 bits contain the offset value from the base address of the pool to the object's address, the next l4 bits carry the pool ID and the next (64-l4-l3) bits may consist of all zeroes.

Just like in the fat pointer method adopted by the libpmemobj library, this method also depends on the lookup of base address by using the Region/pool ID value obtained from the pointer value. However, instead of using a cuckoo hash table, two directly mapped tables are maintained, one is for looking up the RID given the base address, and hence called the RID lookup table, the other one is used for looking up the base address of a NV Region/pool given the RID of the NV Region/pool, and hence is called the Base lookup table.

These two tables are populated at the time of NV Region/pool creation or at the time when the NV Region /pool is opened. The table entries are created at specific addresses in the user address space which can be formed by doing some simple bit manipulation to key values of the table entries. For RID lookup table, given the base address in the form of:

11…1 xx…x 00…0
l1 l2 l3

The virtual address at which the RID value is stored can be calculated like this,

getRID(base):

l1 l2 l3

nvbase = base & 00…0 11….1 00….0;
ridAddr = 11…1 00…0 00….0 00…0 | (nvbase >> (l3 – l4));
RID = *(ridAddr);

The RID look up table will look like:

Virtual address	RID
11…1 00…0 xx…x 00…0	RID5
…….	
11…1 00…0 xx…x 00…0	RID2
…….....	
11…1 00…0 xx…x 00…0	RID7

l1 l3-l4 l2 l4

Similarly, for the Base look up table, given the RID in the form of,

RID = xx..x
l4

the base address can be calculated as,

nvbase = getBase(RID):
baseAddr = (RID << l2) | 11…1 00…1 00…0 00…0;
l1 l3-l4 l4 l2
nvbase = (*baseAddr);

The Base lookup table will look like:

Virtual address	NVBase
11…1 00…1 xx…x 00…0	000101
…….	
11…1 00…1 xx…x 00…0	000001
…….....	

11…1 00…1 xx…x 00…0	000111
l1 l3-l4 l4 l2	

Whenever there is a need to translate the RIV value to an absolute address, the following is done:

1. The RID value is extracted from the RIV value and then a base address lookup is done.
2. The offset value is extracted from the RIV value and is added to the base address fetched in the first step.

Now, the NV Regions cannot be placed at any address that can be formed using nvbase bits. If the nvbase is smaller than 00…...0 1….,

(l3-l4-2)

then there will be a collision with the NV Region address range and the placement of translation tables.

The various combinations for the l1, l2, l3, and l4 bits will give various ways in which the NV space can be organized. For example, the bits in the virtual address are segregated like this: l1 = 2, l2 = 28, l3 = 34, l4 = 30 means that there can be 2^30 NV regions, which can each be 2GB in size.

The adoption of RIV method has several constraints on the virtual address space of a process, where NV Regions can be mapped. For this approach to work, we need starting few (l1) bits of the NV region addresses to be one. So that we can distinguish whether the pointer is pointing to a volatile or a non-volatile object or variable.

However, this constraint cannot be fulfilled with the existing 64-bit address space organization on machines with x86_64 instruction set. Since, such systems use only the lower 48-bits for addressing, and the address space of a process on such machines has a canonical structure [16], as shown in figure 2.1, in which the user space ranges from 0x0000000000000000 to 0x00007fffffffffff and the kernel space ranges from 0xfffff00000000000 to 0xffffffffffffffff. The space from 0x0000ffffffffffff to 0xfffff0000000000 is left unused for future use.

Figure 2.1 Canonical Address Space in x86_64 processors

As can be seen in the above figure, in a canonical address space organization, the 47th bit from right determines whether the address belongs to the user space or the kernel space. Also, the bits from 47th position onwards have the same value as the 47th position bit. If the 47th bit is set then all the other bits from 48th position till 63rd bit position are also set. If the 47th bit is 0 then all the rest of the bits from 48th position till the 63rd bit position are also 0.

Given such an organization, the RIV method implementation had to be changed, without making changes to the OS and/or hardware. Hence, the higher addresses in the user space were considered from 0x0000400000000000 address for mapping NV Regions. For the existing address space organization, the following values were taken for the various bits, which did not take into consideration the type information of the pointer values:

1. l1 = 18, but would consist of 17 zeros and one 1
2. l2 = 12
3. l3 = 34

4. l4 = 30

The implementation of the libpmemobj library is changed slightly to incorporate all these factors:

1. The pool ID is shortened to a 32-bit value, by once again XORing (first byte with last byte manner) the bytes in the previously obtained 64-bit value, and later this value is masked with the l4 bits all set to one, to extract l4 bits for the pool ID.
2. libpmemobj library provides a function 'util_map_hint' which is used to calculate a hint address that serves as the starting address for the NV Region/pool and also is given as an address to the mmap(2) function call.

   In order to support RIV method, the util function definition is slightly changed to introduce the concept of l1 bits and nvbase starting values.

   The function 'util_map_hint' uses /proc to determine a hint address for mmap(). It opens up /proc/self/maps and looks for the first unused address in the process address space that is greater or equal to the address value calculated by:

   (NVMBASE|1ULL <<(64-l1-(l3_l4()-2)-1));

   Where the macro NVMBASE is set to ((uintptr_t)1 << 46) to factor in the fact that only addresses greater than or equal to 0x0000400000000000 will be used for mapping non-volatile regions. Since this is just a prototype, in future uses the l1 bits can have more than one set bit, and in that case this macro will need to be modified to incorporate that. However, for the purpose of this thesis, only one configuration was used as mentioned above.

   And we need another bit to be set at location (l1 + (l3-l4-2) +1)th position from the left, which is done by the above instruction. Hence, this ensures that the starting address of the NV Region is consistent with the implementation of the RIV method.

   The function also checks whether the address range would be large enough to hold 2^l3 bits. After a hint address is constructed, this address and the flag "MAP_FIXED" is given to mmap(2) which maps the pool file specified by the filename given at the time of pool

creation, at the hint address in user space. Hence, the hint function makes sure that the starting range of the NV Region address follows the RIV implementation.

3. Another aspect of the RIV implementation is that at the time of pool creation, the Region ID Lookup table and the Base Lookup table should be populated so that either values can be looked up for translation of RIV value into absolute addresses and from absolute address to RIV value, during execution.

   The addresses at which the table entries should be placed is determined by performing some bitwise operations to the base or the RID value. After performing these operations, the addresses that are generated are provided to the mmap function with the MAP_ANONYMOUS and MAP_FIXED flag so that these entries can be placed at the designated positions without any file backing up this mapping. When the pool is closed the entries from the two directly mapped tables are removed.

The RID lookup table is looked up when there is a translation from the absolute address to the RIV form, in which case the following steps are followed:

1. Extract the base address of the NV Region from the absolute address, where the object/target lies.
2. Use getRID macro to fetch the RID value from the RID lookup table.
3. Extract the offset from the absolute address.
4. Concatenate the left shifted RID value with the offset extracted in step 3 to form the RIV value.

Advantages:

1. RIV pointers occupy less space than the fat pointers.
2. Intra and well as Inter-Region pointers can be supported through RIV pointers.
3. Efficient look up mechanism by using directly mapped tables, which also conserve space that they occupy, i.e.,
   - For RID look-up table, the space occupied would be = number of NV-Regions that can be created by a process*space occupied by one RID entry

Space for RID look-up table = (2^l2) * l4 bits, where l2 is the number of bits representing the nvbase and l4 is the number of bits used for representing RID, and hence the space that will be occupied by one RID entry.

- Similarly, space occupied by the base look up table = (2^l4) * l2 bits, where l4 is the number of bits representing RID and l2 is the number of bits required for a single nvbase value entry into the table.

Hence, total amount of space occupied by the two look up tables: (2^l2 * l4) + (2^l4 * l2) bits.

However, this is the upper bound on the space occupied by the two tables. In reality, the space occupied by the tables would be dependent on the number of NV Regions that are opened by a process at a time. Hence, if there are 'N' NV Regions opened by a process at one point of time then the space occupied by these two tables would be: N * (l4 + l2) bits. For example, in our example it will be N * 58.

2.2.3 Based Pointer

The based pointer approach has been implemented on libpmemobj library in order to evaluate the performance of different methods that support position independence. The based pointers have already been implemented in the Microsoft C/C++ compilers [12] as mentioned in section 1.3.2.

The based pointer for the purpose of this experiment is implemented as a global variable which holds the address of a pool or the address of the root of a pool. All the other pointers in a pool, will hold an offset value from the address held by the based pointer to the address where the object really lies.

Since, for the purpose of experiments, only one pool is considered, hence only one global variable was declared.

2.3 Support for Run-time type checks

While transforming the SQLite code, it was realized that there is a need to provide a run time type checking mechanism to deal with situations in which we cannot discern at compile time whether the pointer is pointing to a volatile object or a non-volatile object. And maybe in future we might also want to distinguish different types of non-volatile pointers from each other at run time as well. Depending on the number of types of pointers used in a system, the first few bits can be reserved to identify the type of a pointers. In this thesis work, four different types of pointers are used in the programs. Hence, we may reserve two bits for type checking during run time.

However, since off-holder value can be both positive as well as negative, errors may arise while translating pointer values due to the ambiguity in deciding the type of the pointer, because the first few bits in off holder will all be one, if a negative offset value was being stored. However, in order to mitigate this, we might have to do some extra work. Instead of keeping the first bit as signed bit, the translation mechanism can keep and check the (length of the offset +1) th bit to discern whether the offset value is negative or not, while the rest of the higher bits are cleared and ORed with the type number (first 2 bits) of the off-holder pointer, after evaluating the offset value.

In another approach, we can reserve the first 3 bits to hold the pointer type and sign of the offset value in case of the off-holder pointer. In this approach the signed bit is left as is, and the pointer type is resolved by checking the two bits adjacent to the signed bit. Hence, off-holder pointer will map to two types: signed and unsigned.

In the implementation of this method on libpmemobj library, the following bit values are used for the pointers.

Table 2.1 Run-time support for different types of pointers

Pointer Method	Pointer Type	64-bit Pointer value with type	Sign of the value
Off-holder	3	011xxx.....x	+
Off-holder	7	111xxx……x	-

Table 2.1 (continued)

RIV	1	001xxx……..x	+
Based	2	010xxx……..x	+
Volatile	0	000xxx……..x	+

The RIV implementation will change according to the number of bits reserved for the type. One configuration with a 2-bit type reservation would be to have l3 = 32 and l4 = 30 with l1= 18 and l2 = 14. For a three-bit type discerning method, l3 can be kept as 32 bits and l4 = 29 bits. However, this is just an example and the choice for the different lengths of l1, l2, l3 and l4 will depend on how the system would be configured by the user.

Chapter 3
Porting New and Legacy Code to NVM

In order to evaluate the performance of various position independence methods, SQLite was selected as a benchmark application. For transforming the SQLite source code to four different forms, each with a different method for supporting position independence, CLANG [17] libtooling was used to build a transformation tool.

The main requirement was that the tool should be able to put all or some objects on persistent pools/NV-Regions/Heaps, and then use the persistent pointers which support position independence for accessing those objects.

Over the course of this thesis, six such tools were developed. This chapter will describe each of the design decisions used for the transformation tools, what were the challenges faced during transformation of SQLite source code and in general, any C program to a form which can support position independent references, for the programs to run on persistent memory. This chapter will highlight the problems that might be faced while transforming legacy C programs, and how they were dealt with.

3.1 Introduction to CLANG libtooling

For all the tools mentioned henceforth, CLANG's libtooling API was used to statically analyze the source code and perform transformations by using the Rewriter class of the CLANG's libtooling API. The version of CLANG used in these transformation tools is 3.4. A little background about CLANG and its libtooling API is necessary to understand how the transformation tools were built using CLANG.

CLANG is a front end for LLVM compiler, which is an open source project, and it provides a user friendly API for accessing the AST (Abstract Syntax Tree) formed during front end parsing phase. Hence, these APIs can be used for example for doing source code analysis or for source transformations.

The API provided by CLANG can recursively traverse the AST and all its nodes. The CLANG's AST consists of Declarations and Statements as nodes. The top level nodes are all Declarations, and Statements are part of function declarations that have body.

The statements are further broken down into different expressions, and even these expressions can be traversed through the AST. All expressions and variable declarations have a type from source associated with them, which can also be accessed through expression objects and variable declaration objects.

CLANG also provides a class called the Rewriter class through which source code can be easily transformed. For example, by selecting the range of source code that needs to be replaced by a string containing the transformed code. The Rewriter class provides many API calls, but the tools mentioned in the upcoming sections mainly use these five types of calls to write to the source files:

1. InsertTextBefore
2. InsertTextAfter
3. ReplaceText
4. ConvertToString
5. overwriteChangedFiles

The expressions that are used in the statements or as subexpressions in other expressions can belong to many different classes. The classes which our tool mainly deals with are:

1. MemberExpr:

   This is a class for representing expressions which are structure or union members.

   These are of the form: X->Y or X.Y, where X is the base expression and Y is the member field of X.

2. ArraySubscriptExpr:

   This is a class for representing expressions which are array elements and consists of the array name and the subscript within the square brackets. For example X [0] or X [1] [2], where X, X [1] are the base expressions respectively.

3. DeclRefExpr:

   This is a class for representing the expressions which are references to a declared variable.

   For example:

   Line 1: int X = 1;
   …
   Line 6: X = Y;
   …..

   In this example, line 6 is a BinaryOperator expression with opcode "=", and L.H.S and R.H.S. expressions of this binary expression belong to DeclRefExpr class.

Given below is a brief overview of how tools can be created by using the libtooling API provided by CLANG.

1. Instantiate an object of ClangTool which provides the utility to run a front end action over a set of files. Information about how the files are compiled and the list of files that need to be compiled is needed to instantiate this object.
2. The information about the list of compilation commands and the list of files on which the compilation or tool actions need to be performed is provided by CommonOptionsParser object, which is basically a parser for options common to all command-line Clang tools.
3. On this compiler tool instance, a custom FrontendAction is run, which will perform the front end action we want to be performed.
4. After the tool is run, an ASTConsumer is created which consumes the AST as it is being parsed or when it is done being parsed. We can override the functions, HandleTranslationUnit to traverse or consume the entire parsed AST and/or override the function HandleTopLevelDecl to consume the AST as it is being parsed.
5. Then the ASTVisitor class is instantiated to traverse the AST nodes. We can override the functions such as VisitFunctionDecl, VisitVarDecl, VisitBinaryoperator to visit the nodes, analyze the information contained in those nodes and use the rewriter to rewrite the source code at the locations in source code where these nodes map to.

6. The Rewriter writes all the changes that are being made to the source code in a string buffer and a call to overwriteChangedFiles dumps all these changes to file. Although the changes are flushed to the file, the AST that has already been formed is not changed and it remains immutable for the entire run of the tool. The files need to be loaded again into the compiler tool instance and the AST has to be parsed again so that the changes made in the source file are visible in a new formed AST in the second run of the tool.

3.2 The basic Idea of Transformation Methods

This thesis work has explored the following methods for transforming C programs such that they will support relative pointers and use an NVM library for allocating objects on persistent memory:

1. For the typed persistent pointers, which are described in section, let the user decide whether:
   a. All the objects in the program needs to be put on the persistent heap
   b. Objects having a particular type need to be put on persistent heap. In this case the question of whether the orthogonal persistence should be applied for persisting the objects or not should be answered by the user. In case of orthogonal persistence all objects reachable from a persistent object are made persistent. Hence there are two cases:
      i. Automatically promote all objects reachable from the persistent objects.
      ii. Let the user define only one type, for example a data structure named struct Btree at a time, all objects of which should be transformed.
2. For value based persistent pointers, in which the only difference between a persistent pointer and a volatile pointer, is the way in which they hold address values, then there are two ways to carry out transformations:
   a. Let the user/application programmer use pragma directive to annotate a certain object as persistent and let the transformation tool or the compiler treat it differently.

However, in this case also there is a decision that needs to be taken by the user as to whether:

i. Automatically promote all member pointer fields in the object that is made persistent to point to persistent object.
ii. Let the user decide which member fields he/she needs to be similarly transformed.

b. Let the user define only one type, for example a data structure named struct Btree at a time, all objects of which should be transformed.

Also, in these cases, let the run time decide whether the pointer is pointing to volatile memory or to the non-volatile memory, and also which type of pointer value is contained in it, i.e. whether the pointer is an off-holder, a based or an RIV pointer.

3.3 Transformation Rules

This sections provides an overview of the transformation rules that should be applied to the program for it to be able to run on the NVML library using relative pointers. These rule are:

1. Include the library specific header files in the source files:
   a. ‘libpmemobj.h’
   b. ‘riv.h’ if the pointers are being transformed to Region ID in Value pointer types
   c. ‘offholder.h’ if the pointers are being transformed to Off-holder pointer types
   d. ‘based.h’ if the pointers are being transformed to Based pointer types
2. Define the layout as mentioned in section 2.1.1, after the ‘include’ preprocessor directives in the file in which main function is defined.
3. Declare the root object type for the pool.
4. Declare the type and type numbers in other source files
5. Create a pool in main function.
6. Depending on the tool, find which all instances to change to persistent type and gather all the information needed to do a successful transformation. For example, in the first three tools:
   a. Find at which all places, the allocation function has to be changed to a library specific function call. Keep track of the functions in which this allocation function call is made.

b. Keep track of which functions are calling which all functions, like a call graph. Because we will need to change the function declaration of all those functions in which we are changing the malloc function to pmemobj_alloc function and we need to pass the pool object to those functions, and to any functions which call those functions.

For example:

```
void enqueue (int iNode, int iDist, int iPrev){
   #pragma pmem_riv
      QITEM *qNew = (QITEM *) malloc(sizeof(QITEM));
     ....
```

In this example, for malloc to be changed to 'POBJ_ALLOC_TRIV(pop,...)', where pop is the pool handle, which is being passed from the main function, the definition of this function should be changed to,

```
void enqueue (PMEMObjpool *pop, int iNode, int iDist, int iPrev),
```

and the call expression of this function in any other function should also be changed. For example, in the below code the call to enqueue function should be changed to, 'enqueue(pop, chStart, 0, NONE);':

```
int dijkstra(int chStart, int chEnd) {
   .....
   enqueue (chStart, 0, NONE);
   ....
}
```

This also means that the dijkstra function declaration should be changed to: dijkstra(PMEMObjpool *pop, int chStart, int chEnd).

And likewise its call expression in other functions should be changed.

7. Change the dereferencing of the pointers depending on which type of pointer is being declared. The following table mentions the various macros used for reading and writing pointer values:

Table 3.1 Macros for translating relative pointer values to absolute address values

Macro	Use
D_RO	Translates a fat pointer to absolute address value
readoh	Translates an off-holder value to an absolute address value
writeoh	Stores the offset value from the pointer's address to object's address into the pointer variable
R_RIV	Translates the RIV value to absolute address value
translate_address	Determines at run time, the type of persistent pointer (off-holder, RIV or based) being dereferenced and then translates the address from the persistent pointer value to an absolute address, by using pointer type specific translation method.

8. Change the function calls used for managing allocation on volatile heap to libpmemobj library specific calls. The following table maps the changes made in a program that uses the normal C allocation functions to libpmemobj library specific function:

Table 3.2 Mapping C library specific allocation functions to libpmemobj library specific allocation functions

C library specific allocation functions	libpmemobj allocation functions	Pointer Type
malloc	POBJ_ALLOC_TOID(pool_handle, type, size, constructor_function, constructor_args_size);	Fat pointer
	Can call any one of the allocation function listed in this table. POBJ_ALLOC_TOID, POBJ_ALLOC_TRIV, POBJ_ALLOC_BASED	Off-holder
	POBJ_ALLOC_TRIV(pool_handle, type, size, constructor_function, constructor_args_size);	RIV
	POBJ_ALLOC_BASED (pool_handle, type, size, constructor_function, constructor_args_size);	Based
realloc	POBJ_REALLOC (pool_handle, previous_pointer, type, size);	Fat Pointer
	Can call any one of the reallocation function listed in this table POBJ_REALLOC_TOID, POBJ_REALLOC_RIV, POBJ_REALLOC_BASED	Off-holder
	POBJ_REALLOC_TRIV(pool_handle, previous_pointer, type, size);	RIV

Table 3.2 (continued)

	POBJ_REALLOC_BASED(pool_handle, previous_pointer, type, size);	Based
free	POBJ_FREE(pointer_to_persistent_pointer)	Fat Pointer
	Can call any one of the free methods, but will need to convert the pointer to those forms.	Off-holder
	POBJ_FREE_TRIV(pointer_value)	RIV
	POBJ_FREE_BASED(pointer_value)	Based

9. Append the code to close the pool in the main function.

Transformations specific to fat pointer method:

Table 3.3 Transformation rules for the Typed Persistent Pointer (libpmemobj specific fat pointers)

Operations	Original Form and Example	Transformed Form and Example
Pointer Assignments	lhs_pointer_variable = rhs_pointer_variable Example: pNext = pSub->pNext	TOID_ASSIGN(lhs_pointer_variable , rhs_pointer_variable.oid) Example: TOID_ASSIGN(pNext, D_RO(pSub)->pNext.oid)

Pointer Equality checker	If(x->y == a) …
	Example:
	
	If(pBt->pWriter == p) …
		If(TOID_EQUALS(D_RO(x)->y, a)) …
		Example:
		
		If(TOID_EQUALS(D_RO(pBt)->pWriter, p)) …

Null pointer check	If (pointer_var) …
	Example:
	
	if(pBt){
	….
		!TOID_IS_NULL(pointer_var)
		Example:
		
		if(!TOID_IS_NULL(pBt)){
		….

	If (!pointer_var) …
	Example:
	
	if(! pBt){
	….
		If(TOID_IS_NULL(pointer_var)) …
		Example:
		
		if(TOID_IS_NULL(pBt)){
		….

Table 3.3 (continued)

Initialization to NULL	type *pointer_var = NULL;
	Example:
	
	pBt->pWriter = 0;
	or
	pBt->pWriter = NULL;
		TOID(Type) pointer_var = TOID_NULL(Type);
		Example:
		
		D_RO(pBt)->pWriter = TOID_NULL(struct Btree);

Declaration of Pointer Variable	type *pointer_var; Example: struct BtShared *pBt; or BtShared *pBt;	TOID(Type) pointer_var; Example: TOID(struct BtShared) pBt;

Declaration of a Pointer Member field in a struct	struct S{
	…
	struct type *member_name;
	….
	}
	Example:
	
	struct Btree {
	…..
	BtShared *pBt;
	….
	}
		struct S{
		…
		TOID(struct type) member_name;
		….
		}
		Example:
		
		struct Btree {
		…..
		TOID(struct BtShared) pBt;
		….
		}

Parameter type and Return type in Function Declaration	struct return_type* dummy_function(struct param1 *pointer, struct param2 *pointer2…); Example: SQLITE_PRIVATE Schema* sqlite3BtreeSchema(Btree*, int, void(*)(void *));	TOID(struct return_type) dummy_function(TOID(struct param1) pointer, TOID(struct param2) pointer2...); Example: SQLITE_PRIVATE TOID(struct Schema) sqlite3BtreeSchema(TOID(struct Btree) , int, void(*)(void *));

A point to be noted here is that, when rewriting the function or variable declarations in fat pointer transformed code, all the types that are being transformed, are written as their underlying type names, and not as their typedef names.

Example, if the original line of code is:

```
Btree *p = NULL;
```

And if underlying type of Btree is struct Btree, then the transformed code would be written as:

```
TOID (struct Btree) p = TOID_NULL(struct Btree);
```

This was done to make it easy for the tool to correctly append the type declarations either directly or through layout declaration.

3.4 Implementation

3.4.1 Changes to CLANG to support transformations

Although the tools are written by extending the API provided by CLANG 3.4, certain changes were made to CLANG source code for features that were required by the tool but not provided by CLANG.

The main change to the CLANG source code was done to the CompilationDatabase class. As mentioned previously, CLANG uses the CommonParserObject to parse the common compilation commands and the files given for compilation at the command prompt. But for a project containing multiple files and packages, the above class would be insufficient to capture all the details about the compilation. CLANG provides a CompilationDatabase interface, whose methods can be overridden to read the compilation database consisting of commands and the files given to tool for compilation. One of the classes that extends this interface is JSONCompilationDatabase, which basically reads the compilation database from a JSON file, named the compilation_database.json. This file basically contains a collection of JSON objects consisting of compilation files and the compilation commands, and can be hand written or can be generated using an open source tool called Bear [18], by giving the following command in the top most directory of a project where the main or the top level Makefile exists: bear make.

This command generates the compilation_database.json in the same directory as the top level Makefile. The clang tool will get the compilation commands and the files for compilations from this JSON file and will then perform the transformation.

However, when CLANG reads this JSON file and stores the information about the compilation, it makes them visible to other functions of the JSONCompilationDatabase class, but makes those functions that are used for accessing this information as ‘const,’ so that these objects are immutable. Hence, once the compilation database is loaded it cannot be changed.

However, in our tools especially the ones which required the change from a normal pointer to a typed persistent pointer, this can lead to a problem, because while transforming the source files we have to consider that, a translation unit consisting of some source files will have some included header files. In our tools we make sure that nothing happens to the system header files and that only header files belonging to the directory of source being transformed are considered for transformation.

Now the source header files can be shared among different Translation Units. So, if during the traversal of one of the ASTs a certain function declaration is changed in a header file, the formation of another AST that uses this function through a header inclusion will be jeopardized. Since, the declaration in the header will not match the function call used in the source file. In order to remove such an error while transforming, the following rule was applied:

1. Transform only the .c source files in all of the translation units
2. Reload the compilation database but this time having only the source header files
3. Transform the header files using the data collected in the previous passes, with the constraint that if header file A is included in header file B, but header file B is not included in any other header files, then B should be transformed first and then A. This is done to make sure the compilation process is not broken due to incompatible type errors.

Hence, for reloading the compilation database object, we had to changes CLANG source code such that we can modify the list containing the compilation files for the last pass where only the header files are translated to form the AST.

In all the tools, the following rule is followed:

If a single file is being compiled then use CommonParserOption to populate the compilation commands list and compilation file list, else use JSONDatabaseCompilation object, and reload this object while transforming the header files.

3.4.2 Transformation Tools

This section gives a brief overview of the design idea of the tools, how to run the tools, what were the challenges faced during transformation using each tool and also what are the short comings of each tool. These tools follow the designs and transformation rules as mentioned in section 3.1 and section 3.2 respectively of this document.

3.4.2.1 NVM converter

The NVM converter was designed with the goal that any pointer pointing to an object should be converted to a relative pointer. Also, a global array of primitive or non-primitive type was changed such that those arrays were allocated on persistent memory pool.

The type of position independence method that should be introduced in the source code should be provided at the compile time of the tool. The following command is given during compilation of the tool:

make CPPFLAGS+=-Dtype

where ‘type’ can have one of the following four values:

i) TOID for fat pointer
ii) TPI for off-holder
iii) TRIV for RIV
iv) BASE for Based pointer.

The name of file or the source package that needs to be transformed should to be given as an input argument to the tool, along with an option (‘f’ or ‘d’) that signifies whether the name is of a file or a source directory.

3.4.2.1.1 Design

The tool consists of three passes, and a pass in this tool refers to the process of traversing an AST. There are two passes per .c source file and one pass per header file.

1. Pass 0

   This pass occurs when the AST is being parsed and is basically used for collecting information about the source code. After each declaration is parsed, the HandleTopLevelDecl function is invoked, and we collect information such as

   a. The various types being used in the source files, and the number of times a pointer pointing to an object of a particular type is being referenced. This was mainly done to generate a root object that will point to an object of a type which has the highest number of occurrences in the main file. This was a simple algorithm that was followed so that the root object can be written automatically by the transformation in the main file, but it doesn't need to work like this. A user can declare the root depending on his/her design decisions.
   b. The information about which data structure is declared in which file, because we will need this information during transformation of those structures. This pass also collects the information about which function calls which all functions.

2. Pass 1

   The information collected from Pass 0 can be used to make the transformations as mentioned in section 3.3 of this document.

   There is a small difference in transforming the code when a pointer value is being read, in case of this tool than all the other tools for fat pointer method. Instead of using one macro for translating the fat pointer to an absolute address pointer, two macros are provided by the libpmemobj library:

   a. D_RW macro is used when target objects are being modified.

b. D_RO macro is used when the target objects are just being read. This macro basically returns the ‘const’ pointer to the object, hence the object is read only. Hence, in this first tool, if a pointer variable that is being transformed belongs to L.H.S of a binary expression, then the transformation tool appends D_RW instead of D_RO.
   For example

   pBt = D_RO(p)->pBt; // here object pointed to by p is read only, hence D_RO is used for dereferencing the pointer.

   D_RW(p)->pBt = pBt; // here a member of p is being written to, hence D_RW macro is being used.

   This feature was later removed so as to make transformations easier and only the macro D_RO was used for both read and modify operations to the object.

The above two passes are applied to all source .c files, before starting Pass 2.

3. Pass 2
   The tool is reloaded with only the header files for transformation. The function declarations and struct declarations are transformed according to the transformation rules mentioned in section 3.3 of this document.

3.4.2.1.2 Issues

The transformation using this tool, when done to simple one source file based code was correct for both compiling and for executing the program. However, when this tool was used for transforming the SQLite code, an overwhelming number of compilation errors were received because:

1. Only a single level of MemberExpr was transformed, whereas the source code contained a lot of inner MemberExpr embedded in them.
2. Many of the rewrites were not proper.
3. Only malloc was checked as the allocation function.
4. Other memory allocation functions were not taken into consideration such as free and realloc.

5. The rewriting of expressions that involved expanded macros was not correct in most of the cases, due to the failure of the tool to compute the correct source location where the expression needs to be replaced.
6. A lot of rewriting involved replacing the source code, rather than inserting the source code, which might have led to multiple replacements for a single expression that would have been visited more than once as two different nodes in an AST, resulting in an overwrite of a previous change with the latest changes within a pass.

3.4.2.2 NVM Mini-converter

This tool was built to reduce the scope of the transformations from all objects to a set of objects that belong to a particular type. In this tool, the rule of orthogonality is followed, such that all member pointer fields (pointing to some struct variable/object) of the selected structure type will also be considered as the types, whose objects need to be allocated on the persistent heap. For example:

If we select struct Btree as the data structure whose objects need to be put on to heap, then all the pointer member variables declared in its type definition needs to be considered for transformation like it itself is being considered. The struct Btree is defined in SQLite source code as:

```
struct Btree {
  sqlite3 *db;       /* The database connection holding this btree */
  BtShared *pBt;     /* Sharable content of this btree */
  u8 inTrans;
   …….
};
```

If tool is provided with the name of Btree data structure for transformation, it will automatically consider structures sqlite3 and BtShared, and all other structures reachable from sqlite3 and BtShared while transforming the source code.

The inputs that should be provided to the tool are:

1. Type: The initial or first type whose objects need to be treated as persistent objects and pointers to these objects need to be treated as persistent pointers.
2. Source File/Directory to be transformed.
3. Option ‘f’ or ‘d’ depending on whether we are transforming a single source file or a directory.

The information about the type of transformation that needs to be applied, as in which relative pointers to use, is provided during the compilation of the tool, just like in the previously mentioned transformation tool.

3.4.2.2.1 Design

The algorithm followed for this tool is also similar to the algorithm followed in the NVM converter tool, with only few changes:

1. Pass 0

   This pass collects the information about:

   a. The structure types whose pointer variables need to be changed to persistent pointer types, and all objects belonging to any of these types need to be allocated on persistent heap.
   b. The function calls and the names of functions which are allocating objects of types, which are being considered for transformation.
   c. The file names of the various source and header files is collected, and information about which file has which structure declared in it.
   d. The number of variables, having underlying types as one of the types considered for transformation, so that a root object having a member pointer field pointing to an object of type having the highest count, can be created. This is again done to automatically create the root structure and is not a necessary step.

2. Pass 1

   In this pass, all the transformations are applied to the source files belonging to a single AST unit by following the transformation rules mentioned in section 3.3.

   The above two passes are followed for all the ASTs, before pass 2 is applied.

3. Pass 2

   In this pass, transformations are applied to the header files by following the transformation rules mentioned in section 3.3.

3.4.2.2.2 Issues

The problem in this tool was also that there were a lot of compilation errors:

1. The return type was not getting properly overwritten.
2. Only malloc was checked as the allocation function.
3. Other memory allocation functions were not taken into consideration such as free and realloc.
4. The rewriting of expressions that involved expanded macros was not proper in most of the cases, due to the failure of the compiler to compute the correct source location where the expression needs to be replaced.
5. Some of the structure definitions were embedded within other struct definitions:

```
struct A{
        int I;
        struct B{
                int x;
                int y;
        } *A;
}
```

Actual code:

```
struct AggInfo {
        ….
        struct AggInfo_col {
           Table *pTab;
           int iTable;
```

```
        ….
    }*aCol;
    …..
}
```

In above case, the tool will try to change the type of member pointer field named ‘aCol’ to a typed persistent type, but that will go wrong because according to the rule of transformation, the type of the member pointer field should look like this:

```
struct AggInfo {
    ….
    TOID(struct AggInfo_col) *aCol;
    …..
}
```

In that case, the struct information of AggInfo will be lost. The work around is to put all embedded struct definitions outside, and then do the transformation.

6. A lot of rewriting involved replacing the source code, rather than inserting the source code, which might have led to multiple replacements for a single expression that would have been visited more than once as two different nodes in an AST, resulting in having the last replacement getting persisted.

3.4.2.3 NVM object converter

This tool uses annotations provided by the user to change the placement of an object from volatile memory to a persistent pool. The tool used custom pragma directives and their handlers registered with CLANG to mark volatile heap allocations that needed to be transformed to libpmemobj library specific allocation statements and then applied transformation rules on the references of that object. In this tool, principle of orthogonal persistence is followed, and all member pointer references reachable from a persistent object are also treated as being pointing to persistent memory.

In this tool the user writes in the source file, #pragma annotation followed by the type of relative pointer to be used for that object, where type can be pmem_riv, pmem_offholder, pmem_based.

Initially the tool was supposed to be written for conversion to typed persistent pointers, however there were a lot of challenges faced while designing rules for such transformations,

hence, only single value pointers which can be in the form of off-holder, RIV, or based pointers were considered later for the tool. It was concluded that for this tool, the transformation cannot be done for a typed persistent pointer, as there will be a lot of compilation errors and it will take a lot of effort by the programmer for transforming the SQLite code, or any other large code base for that matter. The following sections will mention the design and issues for both the cases, i.e. for typed persistent pointer and for value based pointers.

3.4.2.3.1 Typed Persistent Pointer

The object annotated by pragma pmem_type should be made persistent and the pointer pointing to that object is a typed persistent pointer. Since, there will not be a uniformity across the program about how an object of any type is allocated, accessed, sent across functions and is being equated to return values from other functions, the transformation using this tool is very difficult.

In the following section a brief description about the tool design is given, and the issues faced during the designing phase of the tool.

3.4.2.3.1.1 Design

The tool was initially designed to have these passes:

1. Pass 0
    i. Collect information about the object types that are being annotated as persistent, and the functions in which these annotations are used.
    ii. Populate a map that will consists of all the types of the member pointer variables present in the type declaration of the persistent objects.
    iii. Collect information about the function calls, and which data structure is defined in which file.

Since, in our experiments, we are just substituting persistent memory for volatile memory heap and hence are concerned only about the initial execution of the program, because our main aim is to analyze the time measurements of different relative pointer methods, hence, there was no

need to have an algorithm which finds the root, because we need the root only when we are reusing the objects we created in previous run, therefore the algorithm that decides the structure of the root object was removed.

2. Pass 1
    i. Duplicate the function declaration and definitions in which annotations were used by using the source rewriter. Rename the original function to original_name_pmem, and apply the transformations to the objects (in the original function, since we have access to its AST nodes only) and their references, that were annotated as being persistent according to the rules mentioned in section 3.3 of this document. This is done for all the functions in which the annotations are present.
    ii. Duplicate the structure definitions in the source code and transform the original structure definition according to the rules mentioned in section 3.3 of this document, and rename the original structure as original_name_pmem. This needs to be done for all the structures that are being considered for transformation.

    We need to perform such duplication of the code, because it may happen that after transformation there can be a function that has transformed, such that its function declaration might have changed to expect a typed persistent pointer as its parameter, instead of a volatile pointer. However, there may exist some function calls of the same function in the program, which can still be using a volatile pointer as argument to that function call. Hence, the transformation will lead to a lot of compilation errors.

    For example,

    If there is a function:

```
void foo(){

    #pragma PMEM
    struct X *x = malloc(sizeof(struct X));
    bar(x);
    ...

}
```

and another function

```
void fn3(){

    struct X *x = malloc(sizeof(struct X));
    bar(x);
    ...
}
```

and function bar was declared like this before the transformation:

```
void bar(struct X *p);
```

and now has transformed to:

```
void bar(TOID(struct X) p);
```

Likewise, maybe not all objects of a certain type are required to be put on persistent heap, then changing and keeping only one struct definition will lead to more compilation errors, unless all occurrences of objects of a particular type and function calls are transformed.

3.4.2.3.1.2 Issues

Apart from the duplication of code issues mentioned in the previous section, the transformations using this design will have the following issues as well:

1. Pointer Assignments:

   For example:

   ```
   #pragma pmem_riv
   struct X *x = malloc(sizeof(struct X));
   struct X *y;
   y = x;
   ```

   Then, the question is whether both x and y should be made as persistent pointers, because if y is seen as a pointer to a volatile object, then the transformation tool will not transform

the references of y. So, if later we have a member expression of the form, y->x->z, then it will not be transformed to D_RO(y->x)->z, if it sees y as a volatile pointer variable. But if y is made a persistent pointer also and if later,

```
    ...
y = malloc(sizeof(struct X));
```

Since, the above statement does not have #pragma annotation, then for the code following this statement all references of y should be treated as pointer to a volatile object. But in order to implement such a rule, the tool will have to keep a track of where and how a reference to a pointer variable is being used and then accordingly transform the source code. The other solution is to automatically make this allocation also on persistent heap.

There is even a bigger challenge, when we have a conditional or loop statement. For example:

```
#pragma pmem_riv
struct X *x = malloc(sizeof(struct X)); // object should be allocated on persistent heap
struct X *y;
if(..)
  y = x; // y has to be converted to a persistent pointer type
else
  y = malloc(sizeof(struct X)); // y is pointing to volatile heap
...
abc = y->field; //  dereferencing y and reading some field of the structure
```

In this case, there will be an ambiguity as to how should y and its fields be dereferenced, as a volatile pointer or as a persistent heap pointer.

2. Duplication of structure definitions

   If we duplicated the structures, to include persistent types for the field member variables in the definition of a structure, then we will have a problem, because the two structures,

one which contains all the volatile member references and the one with all the persistent member references will not match. For example, if we have two structures X and Y,

```
struct X{
        int a;
        struct Y *y;
}

struct Y{
        Int b;
        Int c;
}
```

If an object of struct X type is annotated to be transformed, then according to the transformation algorithm for this tool, the structure will be duplicated and defined as:

```
struct X_pmem{
        Int a;
        TOID(struct Y) y;
}
```

Then, let's say we have a statement in which we assign the absolute address of a persistent object to a pointer variable which is meant to carry the address of the volatile data structure, that has been transformed.

Then, we cannot do such an assignment, because the structure definitions are different for these two variables.

```
#pragma pmem_riv
struct X *y;     // This variable points to a persistent object
struct X *x;     // This variable has not been annotated and hence is believed to have an absolute address value
….
x = pmemobj_direct(y);        // this assignment is not possible because of type mismatch
x->y = …;                     // cannot assign absolute address to a volatile pointer type and use that to access field, because struct X will be different from struct X_pmem
```

3.4.2.3.2 Value based Persistent Pointer

The issues mentioned above cannot be resolved if we considered the transformation in terms of typed persistent pointers. If value based pointers were considered, then there will not be a need to change the structure definitions and conversion from persistent relative pointers to pointers holding absolute address would also be possible. Earlier, this would have resulted in errors because typed persistent pointer and volatile pointer (carrying the absolute address) would be pointing to two different structures after transformation, i.e. in case of the typed persistent pointer object transformation tool.

The user will have to specify which object he/she wants to move to persistent memory by using the following pragmas:

a. #pragma pmem_riv

b. #pragma pmem_based

c. #pragma pmem_offholder

which will allocate the object that is being allocated in the next line of code to persistent memory, and the address is returned in the form of persistent pointer value. This value may or may not be converted to absolute form before being assigned to the variable.

In the final version of this tool, the transformation was such that the value returned by the allocation functions are in the form of relative addresses. However, a tool in which allocations return absolute addresses to the persistent pointers will also work well. For example, consider a small snippet from a source file which implements the dijkstra's algorithm:

```
void enqueue (int iNode, int iDist, int iPrev)
{
#pragma pmem_riv
  QITEM *qNew = (QITEM *) malloc(sizeof(QITEM));
#pragma pmem_riv
  QITEM *qLast = qHead;

  if(qNew == qLast){
   printf("both the pointers point to same object\n");
  }
  if (!qNew)
```

```
  {
   fprintf(stderr, "Out of memory.\n");
   exit(1);
  }
 qNew->iNode = iNode;
 qNew->iDist = iDist;
 qNew->iPrev = iPrev;
 qNew->qNext = NULL;

 if (!qLast)
  {
   qHead = qNew;
  }
 else
  {
   while (qLast->qNext) qLast = qLast->qNext;
   qLast->qNext = qNew;
  }
 g_qCount++;
}
```

Then the tool will transform the source as follows:

```
void enqueue (PMEMobjpool *pop, int iNode, int iDist, int iPrev)
{
#pragma pmem_riv
 QITEM *qNew = POBJ_ALLOC_TRIV(pop, struct _QITEM, sizeof(QITEM),
NULL, NULL);
#pragma pmem_riv
 QITEM *qLast = qHead;

 if(qNew == qLast){
  printf("both the pointers point to same object\n");
 }
 if (!qNew)
  {
   fprintf(stderr, "Out of memory.\n");
   exit(1);
  }
 R_RIV(qNew)->iNode = iNode;
```

```
  R_RIV(qNew)->iDist = iDist;
  R_RIV(qNew)->iPrev = iPrev;
  R_RIV(qNew)->qNext = NULL;

  if (!qLast)
    {
      qHead = qNew;
    }
  else
    {
      while (R_RIV(qLast)->qNext) qLast = R_RIV(qLast)->qNext;
      R_RIV(qLast)->qNext = qNew;
    }
  g_qCount++;

}
```

3.4.2.3.2.1 Design

The three pragmas mentioned in the previous section were registered with CLANG, and were used to attach to certain statements and declaration, the persistent pointer type, if they had an annotation specified just before their declaration. Two attributes were added to Stmt class and DeclRefGroup class: isPMEM and PMEMtype. Later on during the AST traversal in the tool, all the declarations (Decl), which are part of the binary statement containing malloc, or DeclStmt, which are annotated as being persistent, were also passed on these attributes.

The following passes are executed in this tool:

1. Pass 0
   i. All the Declarations present in the statements (BinaryOperator or DeclStmt) annotated with #pragma pmem_type, are marked as persistent and the persistent type is set as an attribute in those node objects
   ii. Collect information about the function calls as being done in the previous tools.
   iii. If a reference of a persistent memory object, is passed as an argument to a function, then mark the Decl associated with the ParamDecl as persistent.

2. Pass 1
   i. Perform the transformation according to the rules mentioned in section 3.3, on all the variables and their references which are marked as persistent.
   ii. Show a warning if the L.H.S and R.H.S pointers are of incompatible types in a BinaryOperator expression.
3. Pass 2

   After the above two passes have been completed for all the source files, pass 2 is performed, which makes changes to the declaration of functions in the header files, which have been transformed.

While transforming the SQLite code, this tool was appended with a smaller tool, which automatically used to write pragma annotations for a particular position independence method, before all the malloc invocations. A later version of this tiny tool was written to append these annotations to a set of malloc invocations, which were being called to allocate memory for specific types of objects.

3.4.2.3.2.2 Issues

1. One of the issues with transformation of SQLite code using object transformation tool was that there were a lot of compilation errors, due to wrong transformations as mentioned earlier, and certain changes to the source code had to be done manually. Another issue was that there were a lot of run time errors and they were taking a lot of time to resolve. Also, the malloc calls were embedded inside wrapper allocation functions of SQLite source, and hence the tool was changed to accommodate a set of SQLite wrapper allocation functions, and new rules were provided, such that those wrapper functions were replaced by libpmemobj specific allocation function.
2. The transformation tool was trying to consider function calls and the return values received from function calls, while trying to transform the code. As in finding the ambiguities while passing the arguments, or receiving the pointer values from the function calls, and sure enough the transformation algorithm was very complicated.

3.4.2.4 NVM Incremental converter

This tool transforms the source code in an incremental manner, one structure at a time, with inputs from user. There are three versions of this tool, which were separately written for four different relative pointers:

i. RIV method or Based pointer method
ii. Off-holder method
iii. Fat pointer method

The reason for having a separate tool for Off-holder method is that, since the pointers for RIV and Based methods for supporting position independence hold values which are universal and their translation does not change even if they are reassigned, i.e. to say for example

Idx->pTable->pIndex = ptab->pIndex;

Then the assignment is straightforward, because the value stored in ptab->pIndex whether in the form of RIV or based offset value, doesn't need any other information for translation. However, for off-holder method, we first need to form the absolute address of the object pointed by R.H.S pointer variable, then assign the off-holder value by subtracting the address of the L.H.S pointer variable and the absolute address value held by R.H.S pointer.

Also, the RIV and based pointer values can be sent across the function calls and returned from function calls as is, but in case of off-holder pointer, these values should be changed to absolute address first, before either sending them as arguments to functions or returning the pointer values from functions. This is done because these values will not transform to the correct absolute address, because the variables declared in a function are all stack variables and they are pushed onto stack during a function call and are popped out of the stack as soon as the function call returns. It should be noted that the RIV/BASED transformations can also be performed using a similar approach, i.e., to convert to absolute address before sending across functions.

Hence, the rules applied for transforming the source code in a form that supports position independence in these cases would be different for RIV/BASED and off-holder methods.

And the above mentioned two tools will be different than the tool for fat pointer method, because in a fat pointer, we are using a struct to carry two 64 bit values, i.e. the pool ID and the offset from the base, whereas in the other methods a single 64-bit value is used for carrying the relative address of the target. Hence, the transformation rules for fat pointer method will be different from the rules for the rest of the methods.

In all the tools in which the allocation returns a relative pointer rather than an absolute pointer, care must be taken that a statement like this:

Btree *p = &ptab->pBt;

The tool should not make dereferencing of p as a persistent pointer. It should be treated as an absolute pointer. So, whenever tool finds such a statement, it associates the persistent pointer type as “ambi” and the tool is programmed such that on these “ambi” type variables it does not do any transformations.

In all these three tools, the passes were all segregated and the transformation was done structure by structure. And transforming all the occurrences of a particular type was done in a minimum of 3 passes, and mostly after each pass, the code was compiled to see if there were any compilation errors. If not, then when the transformation for one structure was complete, the SQLite code was tested for a small set of commands to see whether the code was properly transformed or not.

This tool takes into consideration which all SQLite allocation functions need to be transformed to libpmemobj allocation functions. The details about which parameter in SQlite allocation function maps to which parameter of libpmemobj allocation function is also provided to the tool by hand coding these details for now. In future, the tool can be improved to accept input from user regarding which all allocation functions to consider while transforming the code, and about the parameter indices corresponding to the parameters required in the libpmemobj library.

After studying about SQLite, the following allocation functions were selected for transformation.

Table 3.4 Mapping SQLite allocation functions to libpmemobj specific allocation functions

SQLite Function	libpmemobj Function
sqlite3_malloc64	POBJ_ALLOC_XXX…
sqlite3_malloc	
sqlite3DbMallocRawNN	
sqlite3DbMallocRaw	
sqlite3Malloc	
sqlite3DbMallocZero	POBJ_ALLOC_XXX…
sqlite3MallocZero	Memset all bytes allocated to 0
sqlite3_realloc64	POBJ_REALLOC_XXX..
sqlite3_realloc	
sqlite3Realloc	
sqlite3DbRealloc	
sqlite3_free	POBJ_FREE_XXX…
sqlite3DbFree	

The consecutive pass to transform the header files was removed because for this tool, we are not making any changes to the function declarations and data structure declarations. And we are declaring the pool object as global.

The transformation tool should be told about which pass it is going to run during the compilation of the tool itself. We can later change these individual compilations for each pass into one tool compilation by reloading the JSON compilation database after each transformation pass. This feature was not added to make sure that if after any transformation pass, there were some compilation errors due to some wrong transformation, such compilation errors were dealt with first before the second pass of transformation so that the source compilation during transformations would not break.

However, a more sophisticated transformation tool will be able to remove the reasons that can lead to wrong transformations that in turn lead to compilation errors. A common reason for compilation errors is that the transformation involving macros that are expanded in the macro preprocessing step can cause incorrect insertion of read macros in the source file, as

these macro expansions are really not rewritten in the file but the read macros from the transformation tools are. For example: Consider the following macro definition in SQLite:

```
/*
** A macro to discover the encoding of a database.
*/
#define SCHEMA_ENC(db) ( (db)->aDb[0].pSchema->enc)
```

Now, according to the transformation applied by our tool, the member variable pSchema here is a persistent pointer. Let's assume that it is a RIV pointer. So, for a statement like this in SQLite code:

```
….
if( !db->mallocFailed ) ENC(db) = SCHEMA_ENC(db);
sqlite3BtreeLeave(db->aDb[0].pBt);
….
```

While transforming using the CLANG tool, the macro would be expanded, and hence the source transformation produces this code:

```
if( !db->mallocFailed ) ENC(db) = R_RIV(SCHEMA_ENC(db);
sq)lite3BtreeLeave(db->aDb[0].pBt);
```

which results in compilation error. Whereas the transformation tool should have really produced:

```
if( !db->mallocFailed ) ENC(db) = R_RIV((db)->aDb[0].pSchema)->enc;
 sqlite3BtreeLeave(db->aDb[0].pBt);
```

A more sophisticated tool may have to make sure the macros are expanded as well as written in the source code, in order to tackle these cases.

3.4.2.3.1 RIV/Based Pointers

There are two separate passes performed in this tool, and each pass has two AST traversals. The pass that needs to be performed is specified to the tool at the time of its compilation. These passes perform very basic transformations and are named like number_pass, where number signifies the sequence number of the pass. Here, by pass we mean an entire run of the tool.

3.4.2.3.1.1 Design

The actions performed by this tool are:

1. zero_pass
    a. Appends the code for opening and closing the pool file in the main function and the declaration of the types used in the file, and the header files being used. Now, since the transformation is structure by structure, with user specifying which structure needs to be transformed, only one declaration per transformation is appended.
    b. Appends the read macros to the expressions that are of the type struct that needs to be changed in the source code, and all of whose objects need to be made persistent. For example, if there is a statement like this:

        p->pBt->db = p->db;

        If both the variable p and the member field pBt are persistent pointers (RIV type), then this is how the transformation would look like:

        R_RIV(R_RIV(p)->pBt)->db = R_RIV(p)->db;
2. first_pass

    Replaces the SQLite allocation function calls with libpmemobj specific function calls.

3.4.2.3.2 Off-holder Pointer

There are three separate passes performed in this tool, and each action has two AST traversals. As with the previous tool, the pass that needs to be performed is specified to the tool at the time of its compilation.

3.4.2.3.2.1 Design

The various passes performed by the tool are:

1. zero_pass
    a. This pass is similar to the zeroth pass mentioned in the first tool for RIV/Based pointer transformations. It appends the code for opening and closing the pool and, also the include directives and type declarations. The type declarations are needed in case of these relative pointers also because the libpmemobj stores the objects of a particular type in a particular list and hence this type number is used during allocations to find the list that contains the objects of a particular type.
    b. It also appends the read macros to the expressions which are of certain type. However, the algorithm in this case is a little different. In the previous tool, only the struct name was given which needed to be transformed. However, in this case the user needs to provide the name of the struct which is being transformed and a list of structures which have already been transformed (also includes the struct which is being transformed).

       This is because the off-holder pointer holds the offset from the pointer in the non-volatile region to the object in the same non-volatile region. Hence, this makes this compulsory for the base expression type to have either already been made persistent through a previous transformation or that it is being made persistent in the current transformation, if there is a member expression that is of the type that is being transformed in the current pass.

       For example:

       Consider the following expression:

           p->pBt->pSchema

       If in the current run the underlying struct type of p is considered for transformation, then p will be considered as a volatile pointer holding an absolute address of the persistent object, and hence this expression will not be transformed. In a later transformation, if underlying struct of member field pBt is being considered for

transformation, this expression will be considered for transformation, because p is pointing to a persistent object, and p->pBt is a pointer on persistent memory and is now going to point to a persistent object. Hence the expression will be transformed as:

```
readoh(p->pBt)->pSchema
```

However, if underlying struct of p was not considered for transformation in a previous pass, but in the current pass the underlying struct of pBt is considered for transformation, then the above expression would remain the same, because now the expression will mean that, there is a volatile pointer p which is pointing to a volatile object, and it has a field member pointer pBt, which is also on volatile memory, and is pointing to a persistent object by holding its absolute address.

Hence, there is a need to specify which all structures have already been transformed for off-holder method.

c. During this pass we also check whether the parent is BinaryOperator expression and whether the opcode is "=" and whether the L.H.S is the same as the given expression. In this case also we don't transform the expression, because here the value is being written into and not being read.

For example, the following line of code

```
BtShared *pBt = p->pBt;
```

is transformed to

```
BtShared *pBt = readoh(p->pBt);
```

However, this line of code:

```
p->pBt = pBt;
```

is not transformed in this pass and remains the same.

2. first_pass

In this pass all the assignment operations are changed which are assigning one pointer variable to another and the type of the pointer variables is one of the structures that we are transforming.

For example, if we have the following original line of code:

```
p->pBt = pBt;
```

then, it will be transformed to the given line of code if the variable p and its member field pBt are both pointing to persistent data structures.

```
writeoh(&p->pBt, pBt);
```

Here also the logic is almost the same as the logic we followed for transforming member expressions, as explained in the previous pass.

3. second_pass

   Change the function calls for allocations to library specific function calls.

3.4.2.3.3 Fat Pointers

There are two separate passes performed in this tool, and each pass has two AST traversals. As with the previous tools, the pass that needs to be performed is specified to the tool at the time of its compilation. However, the transformation for fat pointers take much manual effort as compared to the other three methods, because of the inherent property of the fat pointer being a struct rather than a 64-bit value.

3.4.2.3.3.1 Design

The various passes performed by the tool are:

1. zero_pass

   This pass performs the same transformations as the zeroth pass in the previous two tools. It appends the code for opening and closing the pools, and to append the include preprocessor directive and type declarations.

   It also inserts the read macro for converting the fat pointers to absolute address pointers. But the catch here is that if we directly add that macro to the source code, the code will not be able to recompile during the second pass, because D_RO macro takes only the named union and will not work on the 64 bit values that the pointers contain in the source code. Hence, instead of inserting read macro which accepts a named union, we

can insert a stub macro which can just return the value as is, without performing any operations on it, and later this stub macro can be replaced by the original macro D_RO, with the help of any find/replace editor option.

Because the macro D_RO was written to perform operations on fat pointer, hence, ideally we can't transform the expressions without first changing the allocation functions and the types of the structure member fields. If we apply all transformation rules at once, then it will lead to many compilation errors and run time errors as was seen in the previous tool designs.

Currently, the tool does not support this 'find and replace all occurrences of a particular macro' feature, because the trouble with a compiler tool is that the macros would have already expanded and hence, their occurrences cannot be replaced with D_RO by a straightforward AST query, but a more sophisticated tool will be able to make such a change.

2. first_pass:

   In this pass, all the allocation function calls are changed to library specific calls. These are also changed first to stub macros which will not break the compilation for the next pass, and then later changed to the fat pointer specific allocation function calls.

3. second_pass:

   These are the transformations applied in this pass and the transformations follow the rules mentioned in section 3.3.

   1. Change the return type of the function declarations for both with and w/o body.
   2. Change the type of the parameter declarations in function declarations and definitions.
   3. Change the condition expression in if, while and for statements as mentioned in section 3.3.
   4. Change assignment operations.
   5. Transform the declaration statement.
   6. In record declaration change the type of the member field.

3.4.2.3.4 Issues

Value based pointer transformations had the following issues:

There were some run time issues due to the fact that some of the cases were not handled by the transformation tool. Also, wherever we can have volatile as well as persistent objects of the same type, the dereferencing of pointer will depend on the run time value held by a pointer, hence for such cases, instead of using a particular read macro, we will have to use a macro that determines at run time the type of the pointer and then translates the value to an absolute address accordingly.

Fat pointer transformation had the following issues:

Apart from having some compile time errors owing to some of the cases that are not handled by the tool, there were some run time errors as well, similar to the ones mentioned for value based transformations, and hence, some manual changes to the code had to be made. For example:

a. SQLite has a struct named sqlite3, which acts as a connection object for the database. This struct was not transformed as it should be created newly for each execution of the program, and does not need to be made persistent. Now this struct has two field members:

```
struct sqlite3{
        struct Db *aDb;          // all backends
        ….
        struct Db aDbStatic[2]; // Static space for the 2 default backends
        …..
}
```

So, if we tried to transform such that all objects of type struct Db are allocated on persistent heap, and all expressions are accordingly changed. Then, this might lead to some trouble during compile time or run time, because the field member aDbStatic is statically allocated on volatile heap, whereas field member aDb points to an object on volatile heap. Since, we cannot be sure whether a struct Db pointer is pointing to a persistent heap allocated object

or a statically allocated object during compile time, we will need some mechanism to discern the type at run time. However, in this case, because the fat pointer is a struct and is incompatible with a 64-bit pointer value, the determination at runtime cannot be done. And since, the flow of the SQLite source was dependent in many parts on the interplay between the statically allocated struct Db objects and heap allocated struct Db object, the transformation was leading to a run time error, and hence, the transformation from struct Db had to be removed, for the purpose of our experiments.

Although, this wasn't an issue in other methods, because the type of the pointer can be determined during the run time, and then the translation method can be applied accordingly, just to even out the transformations across all the version, the transformation from struct Db was removed for other methods as well, for the purpose of our experiments.

b. In one of the functions in SQLite, we have two variables defined as

```
Index *pProbe;              /* An index we are evaluating */
Index sPk;                  /* A fake index object for the primary key */
```

And then we have the following lines of code:

```
if( pSrc->pIBIndex ){
   pProbe = pSrc->pIBIndex; // points to a persistent heap allocated object
  }else if( !HasRowid(pTab) ){
   pProbe = pTab->pIndex;    // points to a persistent heap allocated object
  }else{
   /* There is no INDEXED BY clause.  Create a fake Index object in local
   ** variable sPk to represent the rowid primary key index.  Make this
   ** fake index the first in a chain of Index objects with all of the real
   ** indices to follow */
   Index *pFirst;                  /* First of real indices on the table */
   memset(&sPk, 0, sizeof(Index));
   sPk.nKeyCol = 1;
   ……
   pProbe = &sPk;      // points to a stack variable
}
….
```

Then later if pProbe is dereferenced, we will not know whether it points to an object on persistent heap or to a stack variable. Although with the value based pointers, type can be discerned at run time, however, with fat pointer, the types cannot be discerned at run time, and hence, even this stack variable should be made a persistent object, and freed before exiting the function.

c. Another transformation that went wrong was that, there were certain structures which had a union member field, and within that union member, there were pointer member fields, of the type that needed to be transformed. This transformation when applied for fat pointers, resulted in run time errors, hence, these members were not transformed.

The three features that are not supported in the tool, and for which user had to make changes manually to the source code are:

1. While transforming the conditional expressions, the tool does not check for binary operations with && or || operator in a compound statement.
2. The tool does not handle the case when NULL value is being written as one of the two possible cases of the ternary statement.
   For example:

```
pPk = HasRowid(D_RO(pTab)) ? TOID_NULL(struct Index) :
sqlite3PrimaryKeyIndex(pTab);
```

   Here, the TOID_NULL was manually inserted, when pointed by a compilation error.
3. Increment and post increment operations in a ‘for’ loop.

3.5 Conclusion

The transformation tools will require manual input from the user, and also for an effective result, the transformations should be performed in an incremental manner for large code bases, be it through types or through object annotations.

The transformation tools discussed in this thesis, work at data structure level or object level for transformations. The tools should be modified to support persistent pointers to

primitive data types as well. Also, an approach for transformation tool for porting programs to NVM, that is mentioned in the thesis but not implemented, requires the user to provide the type or the object to be transformed, and the field members that need to be transformed or those that need not be transformed. That is to say, user should be able to tell which objects of a particular type, or any object irrespective of its type, should be allocated on persistent memory, and also, should be able to specify either the members within the structures that should be made persistent or the members which shouldn't be made persistent.

All the tools mentioned above created only one persistent pool in the transformed source code, but the programs might require more pools for their operations. However, a more sophisticated tool would need to be implemented for such cases. If there were multiple pools in a program, then there could be a field in an object that could point to objects in the same region/pool or objects in another NVRegion/pool. If a typed persistent pointer was used in this case, then we will have to duplicate the structures to two different types: one containing the field type as Off-holder and another containing the field type as RIV. So, that will result in having three types for the same underlying data structure. And if there are any interplay between these objects, which have the same underlying structure but are of different types, owing to the different types of persistent pointers they are using as their member fields, then it will be very difficult to transform the source code, and will require a lot of manual effort from the user and also some design changes in the program.

However, in case of value based persistent pointer, if the complexity of programs in terms of usage of these relative pointers is increased then we can use the common macro for translating the pointer values, so that the type of the pointers is discerned during the run time, and the dereferencing is done accordingly.

Although a transformation tool should/may take a lot of active inputs from the programmer on how to change the existing program and then perform the transformation, however, some design changes will also be required to be made to the program for it to use this library or any other persistent memory programming API and the persistent memory optimally.

Chapter 4
Experiments and Results

In order to evaluate the performance of different methods that support position independence, we have chosen to transform SQLite code into four different versions using three different transformation tools, which belong to the incremental converter category of the transformation tools. The experiments were run on Intel's PMEP [19] machine, which basically provided an emulated persistent memory system. Although throughout the document, there have been a lot of mention about SQLite and its source code, this section will formally introduce how the SQLite source is structured and what were the final outcomes of the transformation on SQLite.

4.1 Introduction about the PMEP machine

PMEP stands for Persistent Memory Emulator Platform and was developed by Intel for research purposes. It emulates DRAM as a persistent memory [20] by reserving certain physical memory range as persistent memory (type 12), so that addresses falling in those ranges are regarded as going to persistent memory by the OS and the memory controller is changed slightly so that additional delays can be introduced for accesses to the persistent memory address range.

The persistent memory can be reserved using the "memmap" kernel parameter, which takes as input the starting address of physical memory and the range of the persistent memory required. After setting up the kernel configuration for persistent memory, a persistent memory aware file system should be mounted on the persistent device.

4.2 SQLite and its Architecture

SQLite is a very light relational database engine which can be embedded within a process and does not require a separate server program. It is an open source software and is used in many

android applications. It creates a single file for a database, hence the entire database is very easy to port across machines.

As can be seen in the Figure 4.1, the SQLite [15] library basically has the following components:

1. SQL compiler

   The SQL compiler parses the SQL commands and generates a VDBE code.

2. Virtual Machine

   The VDBE (Virtual Database Engine) code generated by the SQL compiler is executed on the virtual machine.

3. The Backend consists of three modules:

   a. Btree

      The SQLite database is maintained on disk using a Btree data structure. There is a separate Btree for each table and index in the database. All the Btrees are stored in the same disk file in some particular format.

   b. Pager

      This module feeds the B-tree module with the cached pages, whenever the B-tree module requests for information from the disk. The page cache also provides the rollback and atomic commit abstraction and takes care of locking of the database file.

   c. OS Interface

      This module provides an interface to SQLite for interacting with the Operating System.

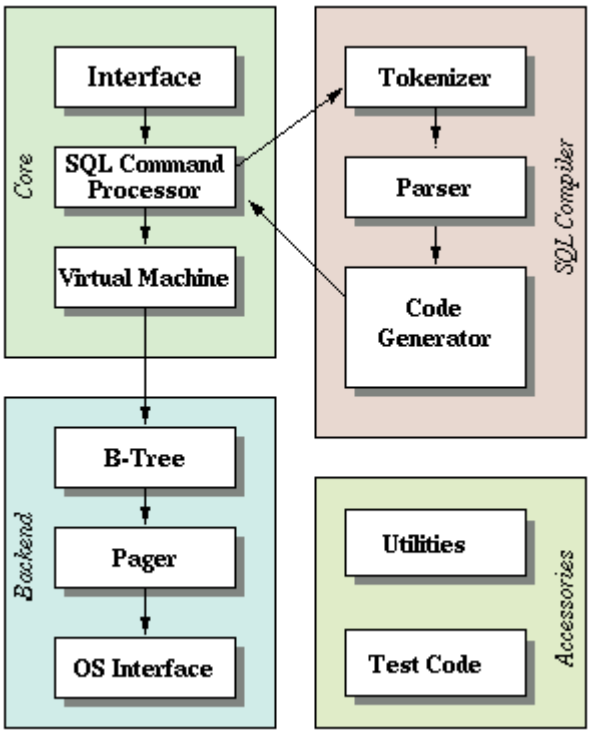


Figure 4.1 Architecture of SQLite

4.3 SQLite Transformed Version

Since, Btree is an important data structure in SQLite, it was chosen for the first transformation that must be applied to the SQLite code, and all other potential data structures were also chosen from the structures that were reachable from this structure. However, many of the structures that were reachable from Btree belonged to different modules, such as the Parser, the Virtual Machine, and the Paging module. Hence, most of these structures were not considered for transformation, as their objects were not so critical so as to be made persistent. There were some data structures for locks and for connections, which were also not considered. Given below is a one-line description of the data structures that were indeed transformed successfully for the value based pointers.

Table 4.1 SQLite data structures that were transformed, along with their description.

Data Structure Name	Description
struct Btree	A database connection contains a pointer to an instance of this object for every database file that it has open.
struct BtShared	An instance of this object represents a single database file. A single database file can be in use at the same time by two or more database connections. When two or more connections are sharing the same database file, each connection has its own private Btree object for the file and each of those Btrees points to this one BtShared object.
struct Column	Information about each column of an SQL table is held in an instance of this structure.
struct Db	Each database file to be accessed by the system is an instance of the following structure.
struct FKey	Each foreign key constraint is an instance of the following structure.
struct Index	Each SQL index is represented in memory by an instance of the following structure.
struct KeyInfo	An instance of the following structure is used to control the comparison of the two index keys.
struct Savepoint	All current savepoints are stored in a linked list.
struct Schema	An instance of the following structure stores a database schema.
struct SubProgram	A sub-routine used to implement a trigger program.
struct Table	The schema for each SQL table and view is represented in memory by an instance of the following structure.
struct Trigger	Each trigger present in the database schema is stored as an instance of this struct.
struct TriggerPrg	At least one instance of the following structure is created for each trigger that may be fired while parsing an INSERT, UPDATE or DELETE statement.
struct TriggerStep	An instance of struct TriggerStep is used to store a single SQL statement that is a part of a trigger-program.
struct VdbeFrame	When a sub-program is executed, a structure of this type is allocated to store the current value of the program counter, as well as the current memory cell array and various other frame specific values stored in the Vdbe struct.

However, as mentioned in section 3.4.2.3.4, the transformations on struct Db for fat pointer method did not result in an error free code, and hence, the transformation was taken off of this structure, and likewise for struct KeyInfo, there was an explicit cast from a volatile object pointer of a different type (which was not being transformed) to an object pointer of struct KeyInfo which was being transformed, and since this address space ambiguity cannot be handled for fat pointer method, the transformation of struct KeyInfo was also taken off of other versions as well.

4.4 Time measurements and Analysis

The performance of the position independence methods, was tested on the transformed SQLite engine and two small programs: B-Tree and List. This section first presents the time measurement and analysis of the smaller Benchmark programs, and then discusses the results for the SQLite tests.

The time measurements for the smaller programs were performed on two different machines, one without persistent memory emulation, and the other machine used was Intel's PMEP machine.

The number of times, a persistent pointer is read and written to, as compared to the total number of reads and writes in the program, that is the number of L1 data cache loads and stores, were also recorded by using perf tool. This data for B-Tree program is summarized in Table 4.2 and Table 4.3, for Insert and Find operations, respectively.

Table 4.2 Read and Write counts for B-Tree Benchmark's Insert operation.

L1-dcache-loads	Count of persistent pointer reads	Percentage of Persistent Pointer Reads to Total Reads	L1-dcache-stores	Count of writes to a persistent pointer	Percentage of Persistent Pointer Writes to Total Writes
177,487,013	143988000	81.1259	23,777,211	12000	0.0505

Table 4.3 Read and Write counts for B-Tree Benchmark's Find operation.

L1-dcache-loads	Count of persistent pointer reads	Percentage of Persistent Pointer Reads to Total Reads	L1-dcache-stores	Count of writes to a persistent pointer	Percentage of Persistent Pointer Writes to Total Writes
176,672,569	72006000	40.7567515	18,436,501	0	0

The time measurements obtained, when B-Tree Benchmark was run on DRAM, with no emulation, and using a normal file system, are shown in Figure 4.2 and Figure 4.3, for Insert and Find operations, respectively. Similarly, the time measurements obtained, when B-Tree Benchmark was run on PMEP system with persistent memory latency set to 115 ns, with ext4, DAX enabled file system, are shown in Figure 4.4 and Figure 4.5, for Insert and Find operations, respectively. The graphs show that all the programs which use persistent pointers, have execution times more than the execution time taken by the program which uses only volatile pointers. Since, fat pointer is the default method for providing position independence on libpmemobj library, and all the rest of the methods are built on top of this method, the fat pointer method, performs slightly better than the other methods, in most of the cases, while all the other types have almost the same execution times.

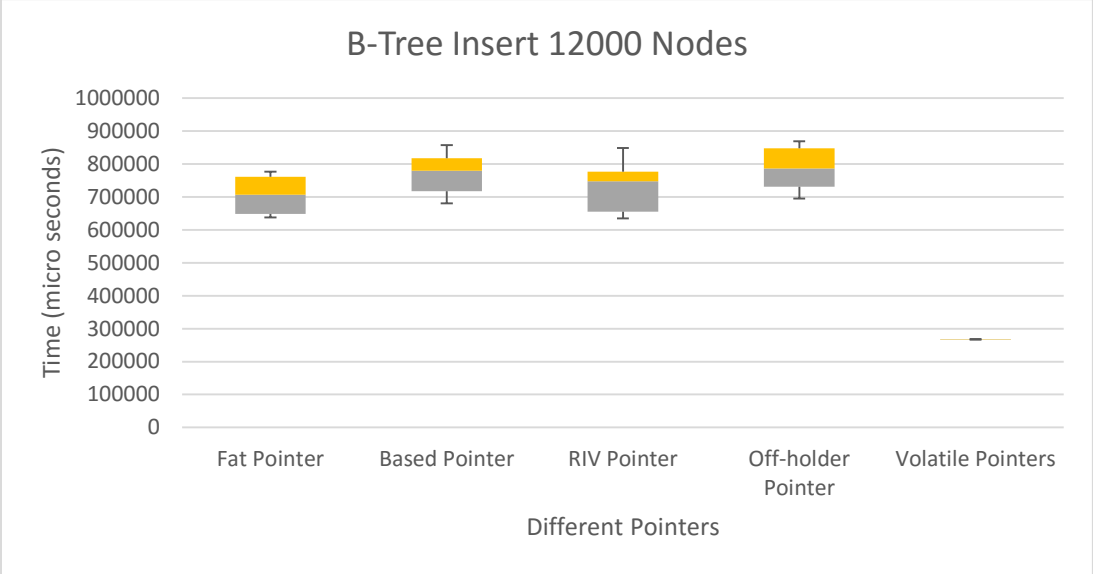


Figure 4.2 Box plot of time measurements recorded for Insert operation (B-Tree Benchmark).

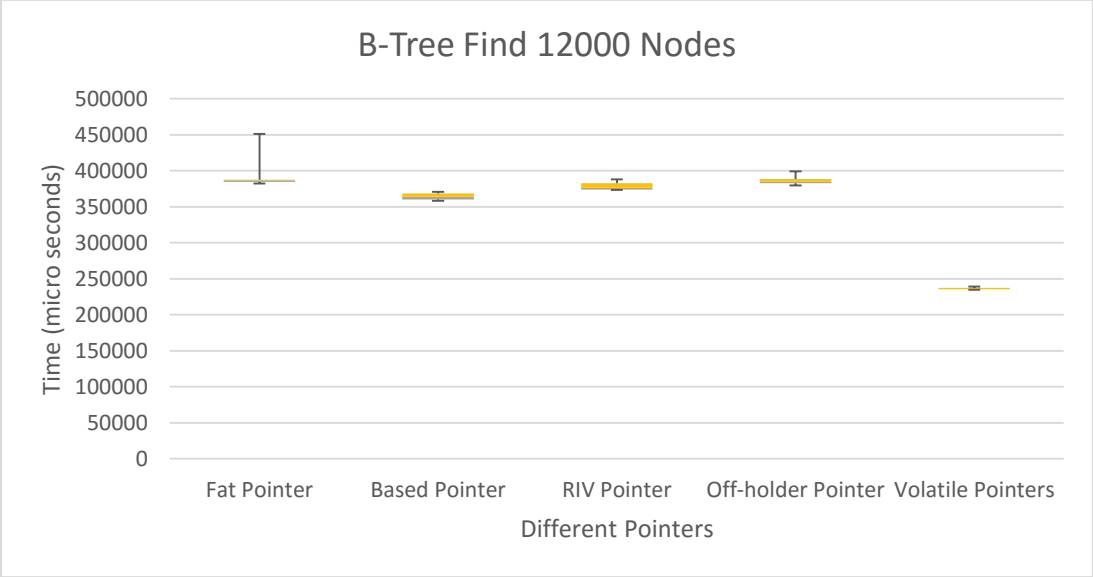


Figure 4.3 Box plot of time measurements recorded for Find operation (B-Tree Benchmark).

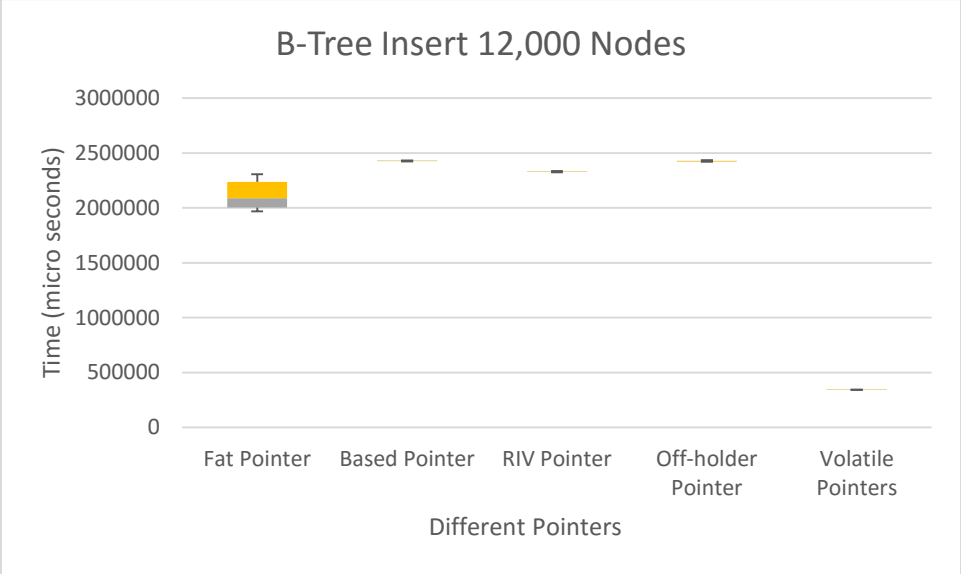


Figure 4.4 Box plot of time measurements recorded for Insert operation (B-Tree Benchmark), on PMEP system.

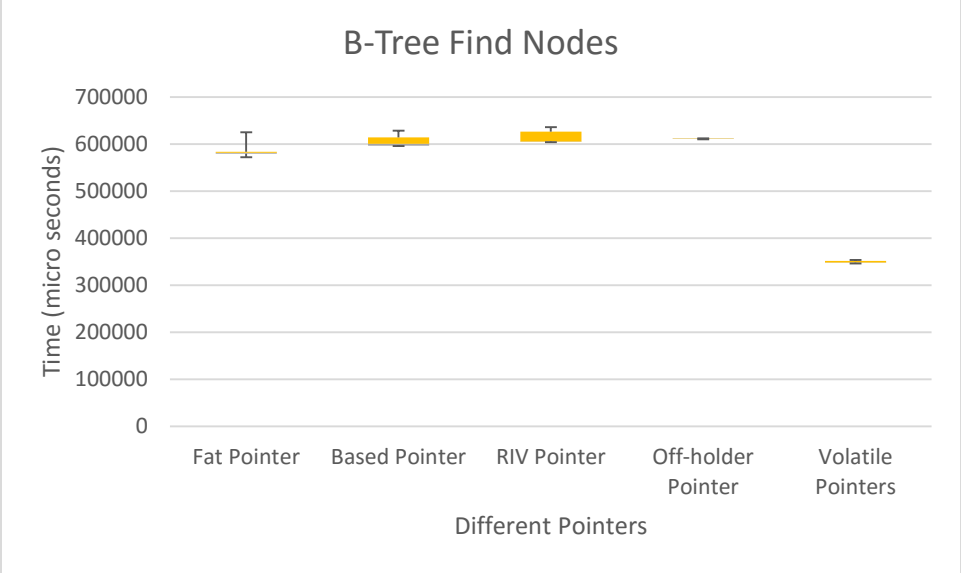


Figure 4.5 Box plot of time measurements recorded for Find operation (B-Tree Benchmark), on PMEP system.

In graphs shown in Figure 4.4, we can see that the execution time of programs using persistent pointers, is more than five times the execution time of the program containing all

volatile pointers. Also, for the ‘Find’ operation, the execution time of programs consisting of persistent pointers, is more than 1.6 times the program consisting of volatile pointers.

The read and write counts for List Benchmark is summarized in Table 4.4 and Table 4.5, for Add and Traversal operations, respectively. In the Add operation, 30000 nodes are added to the list, and in Traversal operation, all the 30000 nodes are traversed 10000 times.

Table 4.4 Read and Write counts for List Benchmark’s Add operation.

L1-dcache-loads	Count of persistent pointer reads	Percentage of Persistent Pointer Reads to Total Reads	L1-dcache-stores	Count of writes to a persistent pointer	Percentage of Persistent Pointer Writes to Total Writes
517,331,924	449985000	86.9819	48,921,786	30000	0.0613

Table 4.5 Read and Write counts for List Benchmark’s Traverse operation.

L1-dcache-loads	Count of persistent pointer reads	Percentage of Persistent Pointer Reads to Total Reads	L1-dcache-stores	Count of writes to a persistent pointer	Percentage of Persistent Pointer Writes to Total Writes
329,794,040	300010000	90.9689	16,941,968	0	0

The time measurements obtained, when List Benchmark was run on DRAM, with no emulation, and using a normal file system, are shown in Figure 4.6 and Figure 4.7, for Add/Insert and Traverse operations, respectively. And the time measurements obtained, when List Benchmark was run on PMEP system with pm-latency set to 115 ns, are shown in Figure 4.8 and Figure 4.9, for Add/Insert and Traverse operations, respectively. The graph results for List Benchmark, are similar to those obtained in B-Tree Benchmark.

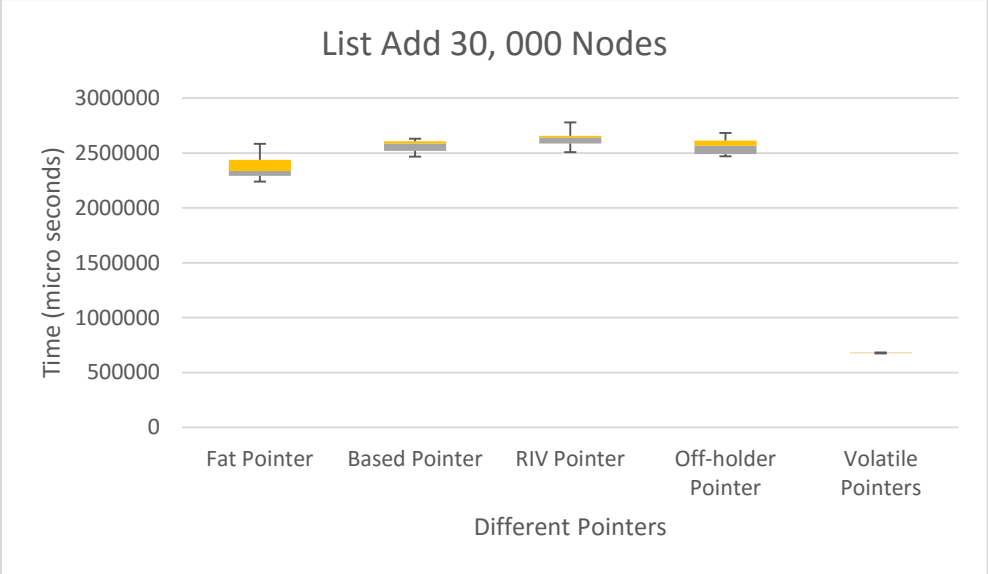


Figure 4.6 Box plot of time measurements recorded for Add operation (List Benchmark).

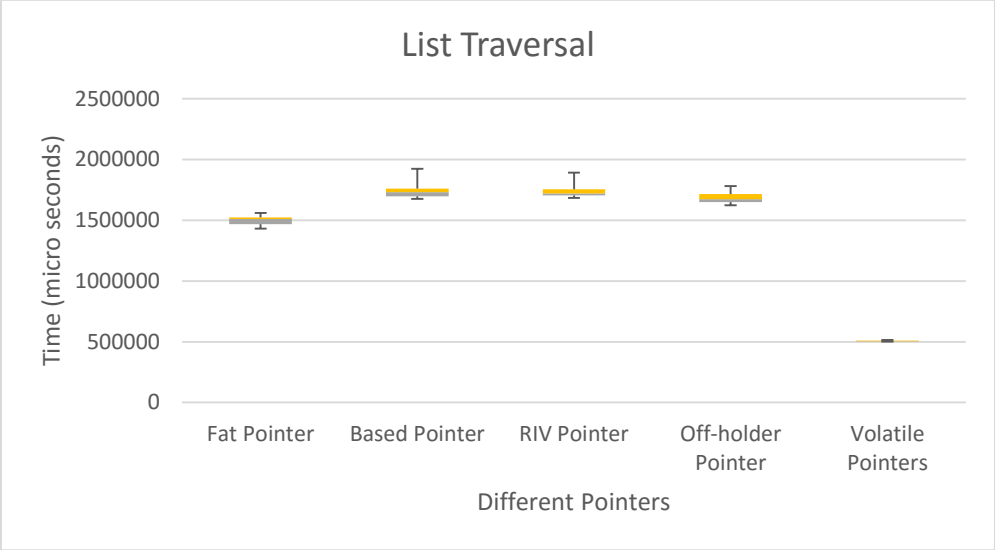


Figure 4.7 Box plot of time measurements recorded for Traverse operation (List Benchmark).

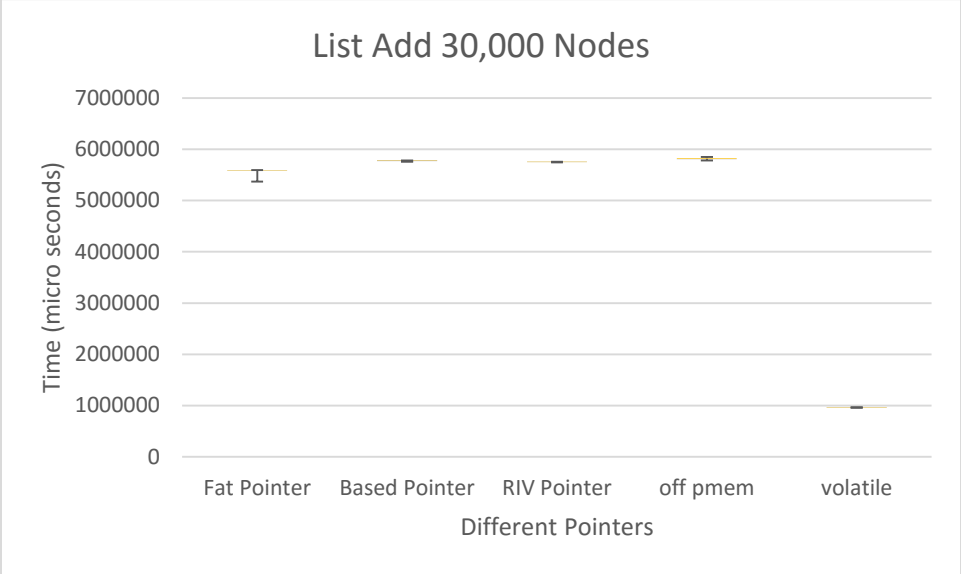


Figure 4.8 Box plot of time measurements recorded for Add operation (List Benchmark), on PMEP system.

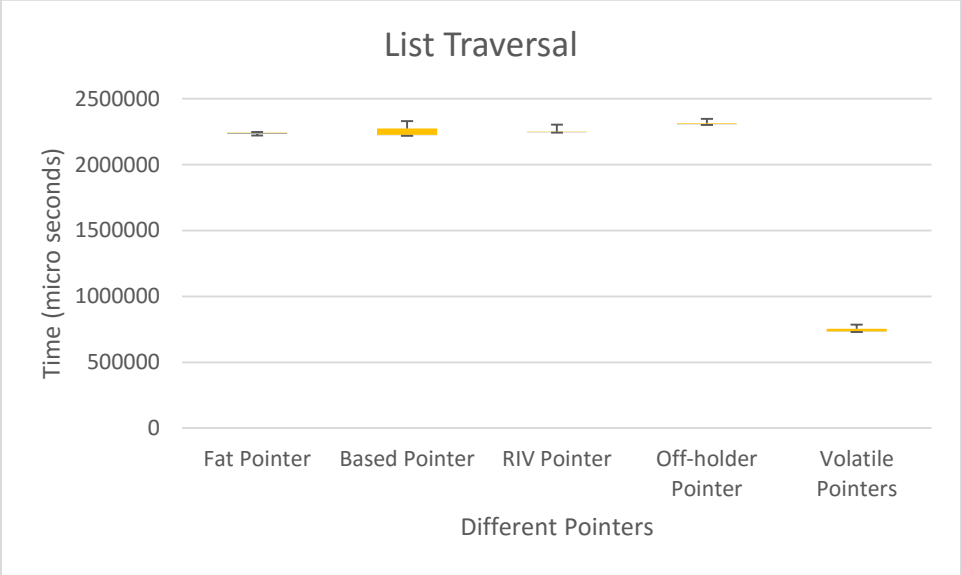


Figure 4.9 Box plot of time measurements recorded for Traverse operation (List Benchmark), on PMEP system.

The results for List Benchmark, obtained on PMEP machine, shows that while the execution times for programs consisting of persistent pointers don't differ much with each other, they are more than five times higher in case of node insertion operation, and more than three times higher in case of node traversal operation, than the program consisting of only volatile pointers.

The transformed versions of SQLite source code were evaluated using a speed-test which consisted of 12 tests as enumerated in the Table 4.6, on the PMEP machine, with the persistent memory latency set to 115 ns. Table 4.6 also, contains the time measurements obtained for different SQLite transformed versions, and the actual SQLite source code. The actual SQLite program used the normal file system for storing the data and for logging, whereas the transformed versions used the persistent device partition on PMEP machine for allocating the objects of the type that were transformed. The timings mentioned in the table are in seconds.

Table 4.6 Time Measurements for tests performed on different SQLite versions.

Tests	RIV	Off-holder	Based	Fat pointer	Only volatile pointers
1000 INSERTs	114.423	106.686	108.22	115.352	59.954
25000 INSERTs in a transaction	0.55	0.552	0.585	0.499	0.323
100 SELECTs without an index	0.566	0.55	0.558	0.553	0.184
100 SELECTs on a string comparison	1.378	1.351	1.368	1.401	0.641
Creating an index	0.245	0.388	0.338	0.256	0.224
5000 SELECTs with an index	0.373	0.346	0.345	0.358	0.531
INSERTs from a SELECT	0.431	0.464	0.465	0.448	0.24
DELETE without an index	0.427	0.501	0.501	0.407	0.205
DELETE with an index	0.3	0.359	0.35	0.365	0.179
A big INSERT after a big DELETE	0.384	0.492	0.434	0.431	0.253

Table 4.6 (continued)

A big DELETE followed by many small INSERTs	0.181	0.222	0.173	0.182	0.119
DROP TABLE	0.275	0.325	0.393	0.273	0.143
Total Time	119.533	112.236	113.734	120.525	63.955

The graph in Figure 4.10, summarizes the timing results from Test 2 through Test 12. Test 1 is removed from the graph, because of the high difference between the timing values obtained in Test 1 with the timing values obtained for rest of the tests.

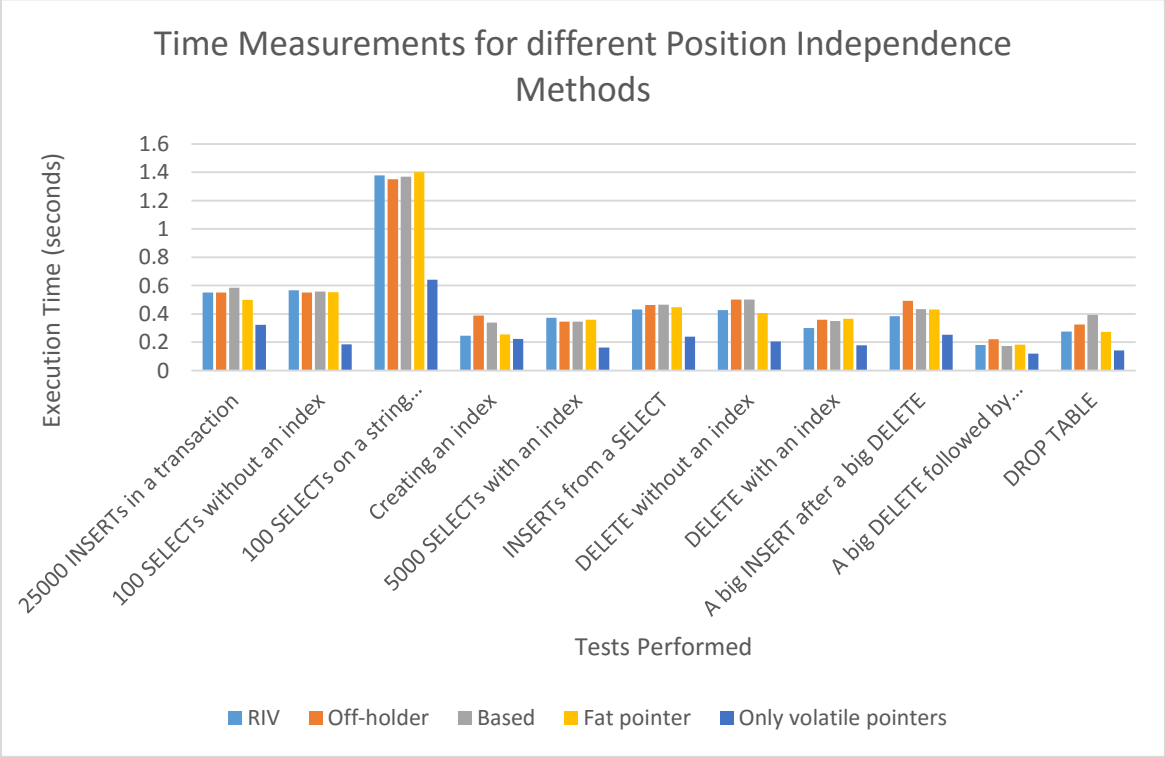


Figure 4.10 Bar graph depicting the time measurements for different tests, across all four types of pointers.

As can be seen from the table records and the bar graph, SQLite engine that uses persistent pointers for some objects of some particular types, take almost twice the time for

executing same sequence of tests, as compared to the actual SQLite engine that consists of only absolute pointers.

However, a sound conclusion based on these results cannot be made, since the number of times a persistent pointer is being read or written to is very small (not even one percent) as compared to the total number of reads and writes in the program.

The libpmemobj library's implementation of the fat pointer method is such that, the most recent pair of base address and corresponding pool id of a pool, is stored in a global structure variable, such that these values are cached for the next dereference of the persistent pointer. If the next pointer being dereferenced also belongs to the same pool as the previous pointer had belonged, then the base address can be picked up from this global variable, instead of going through a look up in the cuckoo hash table, which is more time consuming. This kind of implementation mirrors the TLBs used for virtual address to absolute address translation in an MMU of a computer system, in which certain virtual page number, physical frame number mappings are cached for processes, for faster translation, instead of going to the page table (extra memory access) and then obtaining the physical frame number. This optimization is performed in hardware, which make this process even faster. Hence, a similar approach can be utilized for the RIV methods as well, in which the NV-Region ID or the pool ID and the base address of the NV-Region or the pool can be stored on a TLB like cache in hardware, which will make this lookup even more efficient.

Chapter 5
Conclusion

The incorporation of emerging non-volatile memory as processor's main memory in the system will lead to a change in the programming paradigm and re-design of many system software and applications. The thesis presented two of the challenges that will be faced while adopting the new memory hierarchy model: support for Position Independence, support for Legacy Code. The thesis described four different methods for supporting position independence: Fat pointer, Based pointer, Off-holder pointer, and the RIV pointer. Each method requires a run-time translation of the value held by the pointer to an absolute address form, while dereferencing the persistent pointer, and hence there will be a certain overhead while using such pointers.

Another factor that should be considered while evaluating such methods is the space required by the persistent pointers. Although the space required by the based pointer, off-holder pointer, and the RIV pointer will be the same as required by a normal volatile pointer, i.e. 64-bits on a 64-bit operating system, however, the space required by the fat pointer will be twice the size of a normal volatile pointer.

The off-holder and based pointers can only support intra-region references, whereas the fat pointer and RIV methods can support inter-region references, and require some table look up during the translation from relative address held by the pointers to absolute address of the target.

Since, RIV and Off-holder pointer values, are 64-bit values, they can be easily incorporated into the legacy C programs without making much changes to the program. However, with fat pointer method, there will arise some scenarios in which direct rule based transformations will fail, and more manual efforts by the programmer will be required for porting the existing programs to NVM. Although, the based pointer also carries a 64-bit value, and the porting of SQLite to NVM, which required incorporating this type of pointer in source

code, was straight forward for our scenario, because only one pool (base address) was considered for our experiments, however, for applications, requiring many pools (and hence many base addresses), the porting may not be as straight forward.

Although, there will be various ways in which the existing C programs can be ported to NVM, using tools, such as those mentioned in section 3.4.2, for a successful porting (such that persistent memory can be utilized efficiently), manual efforts and some re-designing of the applications and system software will be required. For example, for applications such as SQLite, the Paging module, which is used to page in and page out data from the disk, so as to make it available to the Btree module of SQLite, as and when needed, may no longer be required, as now the data on the persistent memory can be directly accessed.

Chapter 6
Future Work

The overhead of translating relative pointers at run time to absolute addresses, along with the read-write latency of the persistent memory may lead to a higher execution time of the programs, and hence, ways to optimize such methods will be required in future, which will require help from hardware of the system. The hardware can be used for example, for efficient look-up of base addresses given the region ID or pool ID of the NV-Region or the pool.

The thesis discusses about supporting position independence for persistent heap allocated objects, however, a programmer might want to use global and/or static variables in the program, which may be required to put on persistent memory as well. Also, the thesis does not discuss how the function pointers will be dealt with, given the new programming model. These programming features should also be considered in the future, as these features will also be required to undergo some change in the new programming paradigm. And, once again the changes to these features should be such that the existing programs consisting of function pointers and global and/or static variables, can be easily ported to persistent memory.

REFERENCES

[1] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi and K. E. Goodson, "Phase change memory," in *Proceedings of the IEEE, 98(12):2201-2227*, 2010.

[2] E. Kultursay, M. Kandemir, A. Sivasubramaniam and O. Mutlu, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[3] C. Xu, X. Dong, . N. P. Jouppi and Y. Xie, "Design Implications of Memristor-Based RRAM Cross-Point Structures," in *2011 Design, Automation & Test in Europe*, 2011.

[4] A. Rudoff, *Programming Models for Emerging Non-Volatile Memory Technologies,* vol. 38, ;login:, 2013.

[5] A. Badam, "How Persistent Memory Will Change Software Systems," in *IEEE*, 2013.

[6] K. Bailey, L. Ceze, S. D. Gribble and H. M. Levy, "Operating System Implications of Fast, Cheap, Non-Volatile Memory," in *Proceedings of the 13th USENIX Conference on Hot topics in Operating Systems*, 2011.

[7] "NVML," [Online]. Available: https://github.com/pmem/nvml.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. M. Gupta, R. Jhala and S. Swanson, "NV-Heaps: Making Persistent Objects Fast and Safe," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, ASPLOS*, 2011.

[9] H. Volos, A. J. Tack and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011.

[10] X. Shen, Y. Wu and C. Wang, *Support for C/C++ Programming on Non-Volatile Memory: Specification and Rationale,* 2013.

[11] "Pointer Swizzling," [Online]. Available: https://en.wikipedia.org/wiki/Pointer_swizzling.

[12] "Based Pointers (C)," [Online]. Available: https://msdn.microsoft.com/en-us/library/sbw639kb.aspx.

[13] M. Jordan and M. Atkinson, "Orthogonal Persistence for the Java™ Platform: Specification and Rationale," SMLI TR-2000-94, 2000.

[14] "Persistent Memory Programming," [Online]. Available: http://pmem.io/.

[15] "SQLite," [Online]. Available: https://www.sqlite.org/.

[16] "Canonical Address space," [Online]. Available: https://en.wikipedia.org/wiki/X86-64.

[17] "CLANG," [Online]. Available: http://clang.llvm.org/.

[18] "Build EAR Tool," [Online]. Available: https://github.com/rizsotto/Bear.

[19] S. R Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran and J. Jackson, "System Software for Persistent Memory," in *EuroSys 2014, Proceedings of the Ninth European Conference on Computer System*, 2014.

[20] "Persistent Memory Wiki," [Online]. Available: https://nvdimm.wiki.kernel.org/.