Programming for Non-Volatile Main Memory Is Hard

Jinglei Ren
Microsoft Research
jinren@microsoft.com

Qingda Hu
Tsinghua University
hqd13@mails.tsinghua.edu.cn

Samira Khan
University of Virginia
samirakhan@virginia.edu

Thomas Moscibroda
Microsoft Research
moscitho@microsoft.com

ABSTRACT

Using non-volatile memory as *main memory* (NVMM) can largely improve the performance of applications, but adds to the challenge of programming – it turns out to be very error-prone to write real-world NVMM programs, especially with object-oriented programming. This paper presents a field study of erroneous NVMM programs written by programmers who are trained to use a general NVMM programming interface. We performed the field study in a training workshop of 30 participants. Our observations and derived best practices offer a reference for future NVMM programming techniques design. Toward that end, we propose a taxonomy of latest NVMM programming techniques and, accordingly, a set of paradigms that can reduce the risk of NVMM-specific bugs. The paradigms incorporate a minimal NVMM library interface design and a new design pattern inspired by the field study.

ACM Reference format:
Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. 2017. Programming for Non-Volatile Main Memory Is Hard. In *Proceedings of APSys '17, Mumbai, India, September 2, 2017,* 8 pages.
https://doi.org/10.1145/3124680.3124729

1 INTRODUCTION

Emerging non-volatile memory (e.g., 3D XPoint [11], PCM [15, 22], STT-RAM [2, 14], ReRAM [1]) is both persistent and byte-addressable with access latency comparable to DRAM. Therefore, it provides a unique opportunity to merge main memory and secondary storage, and access persistent data directly with CPU load and store instructions. Such non-volatile main memory (NVMM) improves system performance and energy efficiency by granting direct and fast access to persistent data [5, 12, 18, 25, 26, 28].

Numerous software and hardware mechanisms have been proposed to manage data in NVMM [3, 5, 10, 13, 16, 17, 19, 21, 23, 25, 26, 29]. The main goal of the mechanisms is to protect consistent data from being corrupted by a system crash, i.e., crash consistency. Traditionally, such protection is realized by a filesystem or database, but with NVMM, programmers have to use a programming language or library to do so. As it turns out, however, the resulting NVMM programming approach is very error-prone, especially with object-oriented programming.

Most prior works focus on low-level mechanisms that protect crash consistency of NVMM data. However, it is unclear how well programmers can learn and adopt those allegedly easy-to-use protections in real world. Therefore, we carried out a field study to answer the question. Particularly, 30 participants are trained in a workshop to use a typical but simplified NVMM programming interface. Participants are required to write code for a programming problem during the workshop, and their code is collected for offline analysis. The results show that it is prone to introduce subtle bugs in NVMM programming: omitting necessary protection for NVMM data operations – referred to as *lack of protection*, and mixing volatile and non-volatile data in protection – referred to as *over-protection*. Those bugs can lead to data inconsistency issues in the case of a system crash. Worse still, popular object-oriented programming exacerbates the bug proneness because NVMM data operations are encapsulated and segmented by classes.

Based on the observations from the workshop, we follow two steps to build up a picture of future programmer-friendly NVMM programming techniques. First, we examine existing NVMM data protection mechanisms and their programming approaches. They are classified into four categories according to the protection granularity: *access-level* protection, *code-block-level* protection, *object-level* protection, and *program-level* protection. This taxonomy makes a base for further discussion of lack-of- and over-protection bugs.

Second, we propose programming paradigms for different categories of protection mechanisms so as to help programmers write correct NVMM programs in real-world projects. Particularly, for access-level protection, we design a new *dichotomy* design pattern that enforces principles for NVM allocation and access through two different roles of classes. For code-block-level protection where persistent transactions are employed, we design a minimal programming interface to shallow programmers' learning curves and still prevent typical bugs we observed. For object-level protection, we compare the template approach adopted by Intel NVML [10] and the compiler approach that adds new language features. Both approaches are bug proof but have different advantages and disadvantages. For program-level protection, although it avoids the bugs under discussion, we articulate other programming constrains for properly using this type of protection. These constrains are not fully expressed in related papers.

APSys '17, September 2, 2017, Mumbai, India

ACM ISBN 978-1-4503-5197-3/17/09…$15.00
https://doi.org/10.1145/3124680.3124729

As far as we know, this is the first work to study the challenge of programming applications over software/hardware NVMM systems. We contribute **(1)** observations from a training workshop for NVMM programming, **(2)** a taxonomy of crash consistency protection mechanisms for NVMM, and **(3)** programming paradigms to reduce NVMM-specific bugs, including a minimal transactional interface design and a new design pattern. Furthermore, we open source `libptm`, a prototype library with the proposed transactional interface, at https://github.com/persper/libptm. This project aims to use a *one-page* manual to articulate how to use the library, achieving real ease of use (In contrast, Intel NVML [10] has tens of manual pages). We expect our work to enable easier adoption of NVMM in real-world applications in future.

In the rest of the paper, we motivate our work in §2, review the workshop field study in §3, introduce the taxonomy in §4, and describe programming paradigms in §5.

2 MOTIVATION

NVMM-based systems enable a significant performance and energy gain as they bypass the traditional heavyweight filesystems or databases [5, 12, 18, 25, 26, 28]. In order to guarantee crash consistency of updates to NVMM, prior works proposed various protection mechanisms [3, 5, 10, 13, 16, 17, 19, 21, 23, 25, 26, 29]. Crash consistency means that an update is atomically done on NVMM data even if a system crash interrupts the update. From a programmer's perspective, a key requirement of those mechanisms is that the program has to specify *all NVMM accesses*, in order to trigger proper protection. This is a unique requirement of programming for NVMM. Failure to fulfill the requirement may cause data inconsistency upon a system crash, leading to serious program errors and malfunction.

Moreover, adoption of NVMM in real-world projects will inevitably encounter object-oriented programming (OOP). According to the TIOBE index [24], mainstream OOP languages such as C++ easily take up a popularity share of over 50% in total. OOP complicates the above requirement as NVMM allocations and accesses become encapsulated and segmented. The inheritance and composition blur the placement of data, in either volatile or non-volatile memory. This largely increases the risk of misuse of the protection mechanisms.

In practice, how easy or hard is it for programmers to fulfill the requirement? In what way do programmers fail the fulfillment? What bugs give rise to such failure? To answer the questions in a real setting, we organized a NVMM programming workshop and conducted an in-depth field study of programmers using typical NVMM data protection mechanisms. Through the study, we identified two types of common bugs. To deal with the bugs, we classify NVMM data protection mechanisms and propose general programming paradigms based on our observations from the field study.

3 FIELD STUDY

Error-prone programming threatens the potential application of NVMM. In this section, we start with our NVMM programming workshop, a field study to expose the problem behind the error-proneness. Then, we analyze results of the field study and their implications.

3.1 Workshop Organization

We organized a NVMM programming workshop which attracted 30 student interns of Microsoft Research from ten universities in three countries to participate. They span from undergraduates to PhD students, and all have solid background in programming. In the two-hour workshop, we first introduced NVMM programming using a simplified syntax, and then gave participants a programming task (§3.2). Participants wrote the programs in paper and handed them in for offline analysis.

For brevity, the simplified syntax covers two main activities in NVMM programming, data allocation and access: (1) `nv_new` is used to allocate data in NVMM, a counterpart of C++ `new`; (2) `__nv(variables ...){ statements; ... }` is used to annotate a block of code lines that access certain variables in NVMM, and guarantee crash consistency of the variables as a whole. The access syntax is similar to a *durable* and *atomic* transaction (a.k.a. persistent transaction). Other trivial code such as NVMM configuration, management, zeroing and naming is all left out.

The syntax is decoupled from any specific NVMM system implementation. For example, Intel NVML [10] requires calling `pmemobj_tx_add_range()` on any NVMM data to modify, which is abstracted by the `__nv(...)` annotation. No matter how a NVMM system is constructed, allocations and accesses are inevitable and essential. So, we trained participants with the focus on the inherent NVMM programming methodology.

3.2 Problem and Insight

We present a use scenario to participants and ask them to write a program using NVMM to realize the demanded functionality. To precisely define the demand, we hand out a regular DRAM program as an example of the expected program behavior. Therefore, the task can also be interpreted as converting a DRAM program to NVMM for persistence.

3.2.1 Code Example. The scenario is to write a persistent counter of events. The example program is listed in Figure 1. A class `Counters` is defined to count the events through its function

```
1  class Counters {
2    int *counts;
3   public:
4    Counters(int n) { counts = new int[n]; }
5    int Increase(int i) { return ++counts[i]; }
6  };
7  class TimedCounters : public Counters {
8    time_t timestamp;
9   public:
10   TimedCounters(int n) : Counters(n) {
11     time(&timestamp); // get time now
12   }
13   time_t GetTime() { return timestamp; }
14 };
15 int main() {
16   TimedCounters tc(1);
17   cout << "Since " << tc.GetTime() << ": ";
18   cout << tc.Increase(0) << endl;
19 }
```

Figure 1: The C++ program used in our workshop. Participants' task is to port it to NVMM.

`Increase()`. It incorporates a widely used optimization for multi-threading: an array of integers is created so that threads can be statically assigned to different integers to avoid contention. Suppose another developer reuses `Counters` and extends it by recording since when the counts are calculated. Then, we expect `Counters` to be inherited and extended with a timestamp, resulting in a new class `TimedCounters`.

3.2.2 Bugs. Now we analyze two typical erroneous programs written by participants. Using the above syntax, Figure 2 (a) shows a plausible design to fulfill the task. `Counters` manages both allocation of the counts on NVMM (Line 5) and access to them (Line 8). It follows the encapsulation principle of OOP [7]. In addition, to make `TimedCounters` persistent, any instance of the class is allocated in NVMM (Line 29) and all accesses are protected (Line 17).

Although the overall program seems to work, there is a tricky bug in it. `TimedCounters` inherits the pointer `counts` from `Counters` , and the pointer is finally allocated in NVMM by the main function (Line 29). Therefore, access to the pointer should be protected. However, assignment to the pointer in `Counters` (Line 5) is not protected. This can leave the pointer wild on a crash, because its value is not ensured to be persisted (e.g., it may still stay in CPU caches instead of having been flushed to NVMM). We refer to this type of bug as *lack of protection*.

Besides, Figure 2 (b) illustrates another type of bug, *over-protection*. `Counters` is the same as in (a). `TimedCounters` realizes persistence by directly putting the timestamp on NVMM. Since the main function does not allocate the `TimedCounters` object in NVMM, there is no lack-of-protection bug as in (a). However, the NVMM protection at Line 17 would wrongly apply to the pointer `ptr_timestamp` which is placed in DRAM. Such over-protection brings about a performance penalty as it runs unnecessary NVMM protection mechanisms, and threatens integrity of the mechanisms by mixing unexpected DRAM data into them. More importantly, this is a warning of problematic design and messy data placement. Even though certain NVMM systems can automatically filter DRAM accesses, such a warning should be avoided.

To identify those bugs, we have to go through multiple layers of classes. Suppose `Counters` is from a third-party library. Readers may wonder whether the usage of the superclass `Counters` is safe or not. In theory, OOP promises to save our effort to know every detail of class internals. But the introduction of NVMM tends to invalid the promise. We can see that such segmented view due to OOP complicates reasoning of NVMM programs. As a result, mismatch of data placement (DRAM vs. NVMM) and protection causes many bugs in NVMM programming.

In summary, lack of protection as shown in Figure 2 (a) and over-protection as shown in Figure 2 (b) can cause inconsistency or run-time errors. Applying OOP, a dominant programming paradigm, aggravates the issue, because objects hide implementation details while being open to extension [7]. OOP blurs where data members of a class/object are ultimately placed (DRAM vs. NVMM) and how they should be accessed.

3.3 Observations

After the workshop, we carefully examined all participants' programs. Besides the above bug analysis, we draw three observations.

```
1  class Counters {
2    int *counts;
3   public:
4    Counters(int n) {
5      counts = nv_new int[n];
6    }
7    int Increase(int i) {
8      __nv (counts[i]) {
9        return ++counts[i];
10     }
11   }
12 };
13 class TimedCounters : public Counters {
14   time_t timestamp;
15  public:
16   TimedCounters(int n) : Counters(n) {
17     __nv (timestamp) {
18       time(&timestamp); // get time now
19     }
20   }
21   time_t GetTime() {
22     __nv (timestamp) {
23       return timestamp;
24     }
25   }
26 };
27 int main() {
28   TimedCounters *tc =
29       nv_new TimedCounters(1);
30   cout << "Since " << tc->GetTime() << ": ";
31   cout << tc->Increase(0) << endl;
32 }
```

(a) Lack of protection at Line 5 on the condition that `tc` is in NVMM as Line 29 indicates.

```
1  class Counters {
2    int *counts;
3   public:
4    Counters(int n) {
5      counts = nv_new int[n];
6    }
7    int Increase(int i) {
8      __nv (counts[i]) {
9        return ++counts[i];
10     }
11   }
12 };
13 class TimedCounters : public Counters {
14   time_t *ptr_timestamp; // a pointer
15  public:
16   TimedCounters(int n) : Counters(n) {
17     __nv (ptr_timestamp, *ptr_timestamp) {
18       ptr_timestamp = nv_new time_t;
19       time(ptr_timestamp); // get time now
20     }
21   }
22   time_t GetTime() {
23     __nv (*ptr_timestamp) {
24       return *ptr_timestamp;
25     }
26   }
27 };
28 int main() {
29   TimedCounters tc(1);
30   cout << "Since " << tc.GetTime() << ": ";
31   cout << tc.Increase(0) << endl;
32 }
```

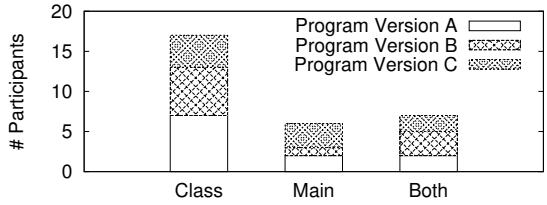
(b) Over-protection at Line 17 on the condition that `tc` is in DRAM as Line 29 indicates.

Figure 2: Problematic programs with two typical bugs.

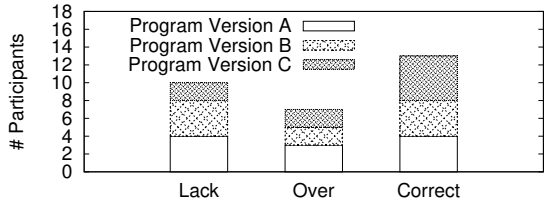
First, *ad hoc design for individual classes is more error-prone than applying a general principle to all classes* (principled design). In the field study, out of 13 correct programs, only 1 program persists `Counters` and `TimedCounters` in different ways. 87.5% programs following the ad hoc design are buggy. In contrast, only 45.5% programs following the principled design have bugs.

Second, *participants put NVMM allocations in various places along the class hierarchy of the program, which drives different program designs and contributes to problematic ones*. Figure 3 (a) shows the numbers of different allocation behaviors. 56.7% programs have NVM allocation in class definition; 20.0% have it in the main function; and the rest 23.3% have it in both class definition and the main function.

One concern is that the allocation behavior of the example DRAM program may influence participants' choice. To shield the influence, we randomly hand out three versions of the DRAM program: Version A allocates data by `new` in only one class (i.e., Figure 1); B does in both classes; C does not do such allocation. Figure 3 shows composition of the three versions. We do not see remarkable influence of different versions on participants' results.



(a) Placement of NVMM allocations.



(b) Bugs in matching NVMM accesses with allocations.

Figure 3: Results of the field study.

Third, *the most challenging part of the problem is matching NVMM access protection with the NVMM allocation behavior* – that confirms our insight and the focus of our work. Figure 3 (b) depicts distributes of two types of the mismatch: "Lack" refers to lack of protection; "Over" means over-protection; "Correct" is code without those bugs.

The above three observations lead to a *guideline* towards bug-proof NVMM programming: *general principles* have to be established to stipulate NVMM *allocation* and *protection* behaviors, especially in class composition and inheritance.

4 TAXONOMY OF PROTECTION

Since NVMM programming bugs are mostly due to misuse of crash consistency protection, a review of different protection mechanisms is the first step to address the bugs. We summarize existing protection mechanisms and classify them by the *protection granularity*, as follows.

4.1 Access-Level Protection

This low-level approach fully exposes the requirement of NVMM programming to programmers. That means a programmer has to specify *every* NVMM access. In addition, to express the crash consistency semantics (i.e., which updates constitute one atomic unit), a number of updates can be grouped into a transaction. The syntax `__nv(variables ...){ statements; ... }` used in §3 is a generic example of this category, where `variables...` specify all NVMM accesses in a transaction and `{...}` defines the scope of the transaction. More concretely, Intel NVML [10] includes a `libpmemobj` library to offer access-level protection.

4.2 Code-Block-Level Protection

To save programmers' burden on annotating every NVMM access and make it less error-prone, a compiler can help do the work by automatically instrumenting CPU loads and stores within a transaction [8]. Mnemosyne [26] and log-structured NVMM [9], for example, leverage this approach. A programmer only need to specify a block of code as the transaction scope, and all memory accesses within the scope automatically trigger callbacks to the underlying NVMM system (note that DRAM accesses have to be filtered out). If we follow the syntax in §3, this approach is like using `__nv{ statements; ... }` without the need to specify every NVMM access.

4.3 Object-Level Protection

The above approaches focus on data *operations*. Another approach is to specify *objects* to be persistent and automatically protect all accesses to those objects. Intel NVML [10] supports this by C++ bindings. NVML utilizes a template `p<>` which overwrites, for example, the assignment operator = so that all updates through assignment trigger protection. The template can be applied to basic data types (e.g., `int`) and a complex object can be made of basic ones. Different from NVML, NVL-C [6] requires programmers to specify non-volatile pointers and all referenced objects must be placed in NVMM.

More advanced compiler techniques even help automate the process of identifying persistent objects. NVMOVE [4] analyzes legacy source code of an application and help programmers identify classes that need to be persistent in NVMM if the code is ported to NVMM.

Another branch of work provides persistent data structures [25, 27], such as NV-tree. These data structures are internally protected, and the protection is transparent to programmers who never modify the data structures.

4.4 Program-Level Protection

This approach protects the whole program assuming that all objects locate in NVMM. Whole-system persistence (WSP) [19] and ThyNVM [23] are two representatives of this category. They are architecture designs to efficiently support persistence of the whole memory space. WSP flushes volatile CPU states using residual energy on a power outage; ThyNVM frequently generates checkpoints during execution time. They resume the program execution after a

crash to ensure data durability and consistency. Unlike other protection mechanisms which are destructive to legacy code, program-level protection adds persistence to even traditional DRAM-oriented programs with few modifications, and transparently guarantees crash consistency of NVMM data.

Choice of the protection level or granularity depends on the requirements of the program or programmers. Low-level or fine-grained protection mechanisms typically lead to high efficiency and system performance, but impose heavy burdens on programmers which may incur a high development cost. In contrast, high-level or coarse-grained protection mechanism have the opposite properties.

Based on the knowledge of different protection mechanisms and their taxonomy, we will describe several programming paradigms to reduce bugs due to misuse of protection mechanisms as well as other reasons.

5 PROGRAMMING PARADIGMS

The observations of our field study indicate the need for *programming paradigms* that define patterns, interfaces or constrains in programming. Different categories of protection mechanisms expect different programming paradigms. We address the main bug-inducing aspect of each protection category, and propose a specific paradigm accordingly. These paradigms are in various forms.

5.1 Design Pattern

The access-level protection offers the most flexibility of programming among all four categories. However, programmers have to directly deal with both bug types as discussed in §3. Our idea is to form a design pattern to structure the classes in such a way that placement of data and its access protection have a general principle to follow. Note that this approach does not rely on any changes to the compiler as required by code-block-level or object-level protection.

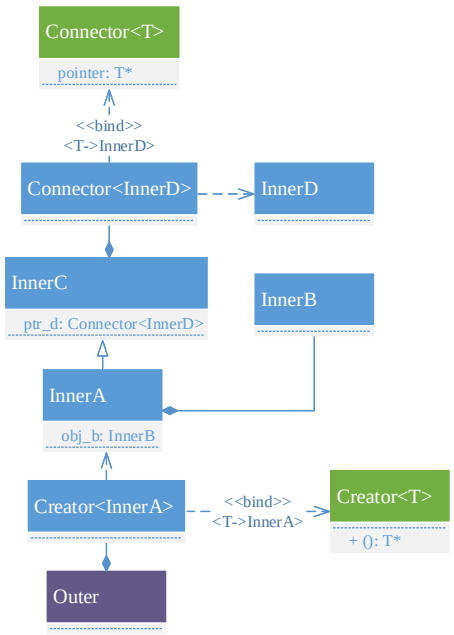


Figure 4: Class diagrams of a dichotomy design pattern.

We divide classes around NVMM into two positions, *inner* and *outer*. Inner classes manage NVMM data, and outer classes use the managed data. This design pattern is named *dichotomy* because a class of one position is only allowed to be extended by a class of the same position. Such extension includes both inheritance and composition. Moreover, an inner class references another inner class by an instance of the predefined class `InnerPtr`. As for interaction between inner and outer classes, it is realized by another predefined class, `Creator`. Figure 4 presents an overview of the classes in the dichotomy design pattern. We elaborate them as follows.

(1) Inner classes take the responsibility to protect accesses to NVMM data. They can safely assume all their objects are ultimately located in NVMM – this is an important assumption we maintain to ease programming and reduce bugs. The principle is that inner classes can *only* connect with other inner classes, by one of the following two ways. First, regular composition or inheritance. For example, in Figure 4, `InnerA` is composed of `InnerB` and inherits `InnerC`. Second, an `InnerPtr` object, if one inner class holds a persistent pointer to another inner class. For example, the buggy Line 5 of Figure 2 (a) contains such a pointer. It should be replaced by an `InnerPtr` object as defined in Figure 5 (a), which protects

```
1  template <class T>
2  class InnerPtr {
3    T *pointer;
4   public:
5    T *operator=(T *p) {
6      __nv(pointer) {
7        return pointer = p;
8      }
9    }
10   T *operator->() {
11     __nv(pointer) {
12       return pointer;
13     }
14   }
15   T &operator*() {
16     __nv(*pointer) {
17       return *pointer;
18         }
19   }
20   T &operator[](int i) {
21     __nv(pointer[i]) {
22       return pointer[i];
23     }
24   }
25 };
```

(a) `InnerPtr` protects pointer assignment.

```
1  template <class T>
2  class Creator {
3   public:
4    template <class... Types>
5    T *operator()(Types... args) {
6      return nv_new T(args...);
7    }
8  };
```

(b) `Creator` enforces allocation of an object in NVMM.

Figure 5: Sample implementation of `InnerPtr` and `Creator`. `T` is a template to bind to any class or type; `args` are arbitrary arguments to pass to `nv_new`.

assignment to the pointer and prevents the lack-of-protection bug. In Figure 4, `InnerC` maintains a persistent pointer to `InnerD` by holding an instance of `InnerPtr` bound to `InnerD`.

(2) Use of the inner classes is through `Creator`, which offers an interface for outer classes to instantiate inner classes. Figure 5 (b) shows a basic form of `Creator`. It ensures that the object of an inner class is allocated in NVMM, but the pointer to the object is not persistent (different from `InnerPtr`). In Figure 4, an instance of `InnerA` is created by `Creator`. We regard C++ main function as a special outer class.

Finally, we show in Figure 6 the correct NVMM program for our field study, applying the dichotomy design pattern. We can regard `time_t`, `Counters` and `TimedCounters` all as inner classes, so composition or inheritance among them are allowed. But a `InnerPtr` instance is required to reference `counts` that is allocated in NVMM (Line 5). Then, the main function uses `TimedCounters`, and instantiates it by a `Creator` object (Line 29).

```
1  class Counters {
2   InnerPtr<int> counts;
3  public:
4   Counters(int n) {
5     counts = nv_new int[n];
6   }
7   int Increase(int i) {
8     __nv (counts[i]) {
9       return ++counts[i];
10    }
11  }
12 };
13 class TimedCounters : public Counters {
14  time_t timestamp;
15 public:
16  TimedCounters(int n) : Counters(n) {
17    __nv (timestamp) {
18      time(&timestamp); // get time now
19    }
20  }
21  time_t GetTime() {
22    __nv (timestamp) {
23      return timestamp;
24    }
25  }
26 };
27 int main() {
28  Creator<TimedCounters> tc_creator;
29  TimedCounters *tc = tc_creator(1);
30  cout << "Since " << tc->GetTime() << ": ";
31  cout << tc->Increase(0) << endl;
32 }
```

Figure 6: Correct program using the dichotomy design pattern.

Throughout this section, we keep to our simplified syntax to demonstrate the essence of the design pattern. But in practice, certain NVMM systems may involve additional operations, such as naming objects in NVMM, or retrieving a native pointer from a persistent pointer that contains extra metadata [5, 6, 10]. A favorable fact is that most those operations can be done in `InnerPtr` and `Creator`, hiding details from other classes.

Overall, the dichotomy design pattern for access-level protection mechanisms manages to assign classes to two exclusive positions and specify a coherent NVMM allocation and access principle accordingly. With the design pattern, programmers need not reason about ad hoc allocation or access protection behaviors among classes. Thus, our observations from the field study justify the effectiveness of such a programming paradigm in practical use.

5.2 Minimal Transaction Interface

The code-block-level protection assumes that the compiler instruments memory accesses and filters out those to volatile data. This capability not only saves programmers' effort spent in manually annotating NVMM accesses, but also alleviates the over-protection issue. DRAM accesses automatically bypass the NVMM data protection even if they are annotated. This is a convenient feature in many cases and helps reduce the risk of writing problematic code.

While access-level-protection-based Intel NVML [10] entails tens of pages of a tutorial to introduce all sorts of its APIs, we set up a *goal* to design a *minimal* transaction API with code-block-level protection. It is minimal in the sense that only *one-page* manual is necessary to explain the usage of the interface. We specify the interface here, and open source a library prototype that works with the interface at https://github.com/persper/libptm. The full interface of our library consists of three parts.

5.2.1 Allocation. NVMM data lives longer than processes. We assume it is managed by the operating system as files. But for one process, our library associates only a *single* data file with it. So, programmers call `popen("data/file/path")`, typically in the beginning of the program, to get started.

To allocate and give back DRAM areas, traditional programs call `malloc` / `free` in C or `new` / `delete` in C++. Our interface provides their counterparts for NVMM, `pmalloc` / `pfree` and `pnew` / `pdelete`, with similar semantics.

A key rationale that enables the simplicity of our interface is to map the NVMM region into a *fixed* virtual address for a certain process[1] so that programs use native virtual addresses and pointers to reference NVMM data. Consequently, a simple `popen` call is enough to setup the mapping and all allocations return native virtual addresses, with little change to the current programming convention.

5.2.2 Naming. Our interface provides a way to assign a unique string ID to a NVMM data structure, so that the data is retrievable after the process starts over. Such an ID is referred to as a seed. Suppose a program stores a tree in NVMM, then the C++ code can go like Figure 7. Usually only a very small number of seed IDs are necessary, as most NVMM data is reachable from them. When you do not want a seed ID any longer, call `pderegister("unique-id")`.

5.2.3 Annotation. We use a similar syntax to that in §3 for annotation. Figure 7 includes code snippets that show the way to annotate code blocks (Line 7 and below). Note that there is no need to manually annotate specific variables as the compiler does that automatically.

5.3 Beyond Bugs

Both object-level and program-level protection mechanisms inherently follow the guideline we derived from the field study observations (§3). Both categories of protection mechanisms are principled

[1] The current Linux kernel cannot *guarantee* this, but support of such a feature is viable considering the huge 64-bit address space. Our current library implements a best-effort `mmap` and can succeed in most cases.

```
Tree *tree = (Tree *)pretrieve("unique-id");
if (!tree) { // In case it is not created yet
  tree = pnew(Tree); // Create the tree
  pregister("unique-id", tree); // Give a name
}
// ... Hereafter use the tree as usual
__nv {
  tree->insert(value);
}
```

Figure 7: Example C++ program using our interface. When the process starts up for the first time, a persistent tree is created and registered; otherwise, the persistent tree is retrieved as a valid address is returned.

in NVMM allocation and access, so they are defensive to lack-of- and over-protection bugs. However, they incur other constrains in programming. We discuss such guideline satisfaction and constrains to characterize the resulting programming paradigms.

5.3.1 Protection Bug Proof. Most object-level protection mechanisms incorporate internal control of NVMM access, so programmers only need to specify which objects to store in NVMM. Proper access protection is automatically added, and the program is correct as long as all necessary objects are made persistent. Other object-level protection with inference ability even does not bother programmers to choose (most) persistent objects.

Similarly, the program-level protection is *software-transparent*, so programmers even have no chance to make protection bugs. However, such a bug-proof feature of the two high levels of protection is not without a price.

5.3.2 Constrains. The internally-protected persistent data structures only open up predefined interfaces to programmers and are not composable. For example, multiple operations on such data structures cannot combine into one transaction. This largely limits their application and usability. Some compilers use reachability from known persistent objects to decide persistence of other objects. This method may enlarge the persistence scope and cover unintended objects. Both false positive and false negative cases are encountered in such source code analysis tools [4].

The program-level protection mechanisms have two constrains. First, they typically manage only memory data but hardly preserve the states of network cards and other devices. That means the resumed program may encounter I/O errors. To ensure crash consistency despite of such errors, programmers have to carefully test error/exception handling paths so that the persistent data remains in a correct state when the execution falls into those rare paths.

Second, the resumed program may diverge from what have actually happened, due to indeterminism introduced by concurrency. As long as it does not resume from an instant immediately before the crash, the programs has to ensure that concerned NVMM writes are deterministic or that the indeterminism is benign. One benign case is when writes to disjoint areas can be reordered. Another benign case is to have no *user-perceived* effects on the program behavior. The program can implement such a logic that user requests are acknowledged only after persistence of NVMM data is ensured (e.g., by waiting until a NVMM checkpoint has been made). This approach follows the external synchrony model [20].

6 CONCLUSION

This paper reports our experiences and observations in a NVMM programming workshop, which acts as a field study of programmers writing code for NVMM. We find two types of NVMM-specific bugs, factors contributing to the bugs, and a guideline to avoid them. Furthermore, we define a four-class taxonomy of existing NVMM programming techniques that protect NVMM data at different levels. We share our thoughts on recommended programming paradigms for every class. Among them, the dichotomy design pattern and the minimal NVMM library interface are useful to both future NVMM programmers and NVMM programming technique developers.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Xiaofei Liao, for their valuable feedback. This work was partially supported by the National Natural Science Foundation of China (Grant No. 61502266, 61433008, 61232003), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), and the China Postdoctoral Science Foundation (Grant No. 2016T90094, 2015M580098).

REFERENCES

[1] H. Akinaga and H. Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98, 12 (2010). https://doi.org/10.1109/JPROC.2010.2070830

[2] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. 2013. Spin-transfer Torque Magnetic Random Access Memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst.* 9, 2, Article 13 (May 2013), 35 pages. https://doi.org/10.1145/2463585.2463589

[3] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-ahead System for In-memory Non-volatile Data-structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508. https://doi.org/10.14778/2735479.2735483

[4] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. 2016. NVMOVE: Helping Programmers Move to Byte-Based Persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW '16)*. Savannah, GA. https://www.usenix.org/conference/inflow16/workshop-program/presentation/chauhan

[5] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 105–118. https://doi.org/10.1145/1950365.1950380

[6] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. 2016. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. 125–136. https://doi.org/10.1145/2907294.2907303

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[8] GCC. 2017. GNU libitm. https://gcc.gnu.org/onlinedocs/libitm/. (2017).

[9] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC '17)*. Santa Clara, CA. http://jinglei.ren.systems/files/lsnvmm_slides_atc17.pptx

[10] Intel. 2016. The NVM Library. http://pmem.io/. (2016).

[11] Intel and Micron. 2017. 3D XPoint Technology. https://www.micron.com/about/our-innovation/3d-xpoint-technology. (2017).

[12] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support*

for Programming Languages and Operating Systems (ASPLOS '16). 385–398. https://doi.org/10.1145/2872362.2872392

[13] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. 399–411. https://doi.org/10.1145/2872362.2872381

[14] E. KÃijltÃijrsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceeding of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '13)*. 256–267. https://doi.org/10.1109/ISPASS.2013.6557176

[15] B.C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, E. Ipek, O. Mutlu, and D. Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30 (Jan. 2010), 131–141. https://doi.org/10.1109/MM.2010.24

[16] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. 329–343. https://doi.org/10.1145/3037697.3037714

[17] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. 2014. NVM Duet: Unified Working Memory and Persistent Store Architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. 455–470. https://doi.org/10.1145/2541940.2541957

[18] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS '13)*. Article 1, 17 pages. https://doi.org/10.1145/2524211.2524216

[19] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. 401–410. https://doi.org/10.1145/2150976.2151018

[20] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. 1–14. http://dl.acm.org/citation.cfm?id=1298455.1298457

[21] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[22] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. 2008. Phase-change Random Access Memory: A Scalable Technology. *IBM J. Res. Dev.* 52, 4 (July 2008), 465–479. https://doi.org/10.1147/rd.524.0465

[23] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*. 672–685. http://persper.org/thynvm/.

[24] TIOBE software BV. 2017. TIOBE Programming Community index. http://www.tiobe.com/tiobe-index/. (2017).

[25] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST '11)*. 61–75. http://dl.acm.org/citation.cfm?id=1960475.1960480

[26] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 91–104. https://doi.org/10.1145/1950365.1950379

[27] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*. 167–181. http://dl.acm.org/citation.cfm?id=2750482.2750495

[28] Y. Zhang and S. Swanson. 2015. A study of application performance with non-volatile main memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST '15)*. 1–10. https://doi.org/10.1109/MSST.2015.7208275

[29] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. 421–432. https://doi.org/10.1145/2540708.2540744