Decoupling Application Logic from Persistent Memory Frameworks with AspectC++

Marcel Köppen
Universität Osnabrück
Germany
marcel.koeppen@uos.de

Birte Friesel
Universität Osnabrück
Germany
birte.friesel@uos.de

Christoph Borchert
Universität Osnabrück
Germany
christoph.borchert@uos.de

Olaf Spinczyk
Universität Osnabrück
Germany
olaf.spinczyk@uos.de

Abstract

Over the past decade, various systems and software libraries have been developed that provide crash consistency on byte-addressable persistent memory. They often require programmers to adapt their code significantly or to use special compiler plugins. Constant innovation in this evolving field makes it desirable to be able to easily switch to more recent systems without massive code refactoring, and without changing compilers.

In this paper, we show how aspect-oriented programming can be used to automatically apply crash consistency to normal, sparsely annotated C++ code. In two case studies, we find that our approach significantly reduces the amount of code required to apply state-of-the-art crash consistency frameworks such as PMDK libpmemobj++ and Pronto.

1 Introduction

Byte-addressable persistent memory (PMem) has been shown to provide new opportunities and challenges for software development. For example, programmers have to consider *crash consistency* [13, 16], that is, ensuring that application data can be recovered correctly in case of system failure. While there are numerous systems, libraries, and custom data structures that aim to solve this problem, they often affect the code massively. For example, the code in Figure 1 shows a crash-consistent C++ class written with libpmemobj++ from Intel's Persistent Memory Development Kit (PMDK) [8]. A comparison with the corresponding regular C++ code in Figure 2 reveals that there are several additional library calls for persistent allocations, explicitly defined transactions, and even the types of data members have to be wrapped.

PLOS '21, October 25, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *11th Workshop on Programming Languages and Operating Systems (PLOS '21), October 25, 2021, Virtual Event, Germany*, https://doi.org/10.1145/3477113.3487269.

Table 1. Code changes needed for persistence frameworks

Framework	Types	Wrappers	Annotations	Transactions
libpmemobj++	✓	–	–	✓
Pronto[a]	–	✓	✓	–
NVMReconstruct[b]	–	–	✓	N/A
Romulus	✓	–	–	✓

[a]Uses a preprocessor. [b]Uses a Clang compiler plugin.

With the evolution of PMem frameworks, it is desirable to decouple the application logic from the framework-specific code. In this paper we propose an aspect-oriented [10] approach that uses the AspectC++ language [18] to separate the regular C++ source code from the PMem framework. PMem-specific source code can be encapsulated in generic aspect modules that can be reused for multiple data structures and even for different programs. Our main contributions are:

- An analysis of common patterns found in PMem software shows massive code tangling (Section 2).
- Two small case studies with PMDK libpmemobj++ transactions (Section 3) and Pronto [14] (Section 4) demonstrate that our approach almost eliminates the need for boilerplate code.
- We show that our approach does not introduce persistence bugs and has no unreasonable effects on performance (Section 5).

2 Code Impact of Persistence Frameworks

This section presents an overview of the programming effort required for using several well-known PMem frameworks. The main findings are summarized in Table 1.

PMDK libpmemobj++. Intel's Persistent Memory Development Kit (PMDK) [8] is a collection of libraries for persistent memory. Libpmemobj provides a low-level C interface to PMem pools, undo-logging, and transactions. In addition, libpmemobj++ offers C++ template bindings for wrapping the members of data structures to automatically take a snapshot on write access. Pointers to persistent objects are stored as smart pointers instead of regular pointers, so that memory pools can be moved. Persistent objects are allocated and deallocated using special functions. All writes to persistent objects and all (de-)allocation operations have to be either

```
1  class snake {
2    persistent_ptr<element_list> snake_segments;
3    p<point> last_seg_position;
4    p<direction> last_seg_dir;
5  public: // ...
6    void add_segment() {
7      auto pop = pool_by_vptr(this);
8      transaction::run(pop, [&]{
9        persistent_ptr<element_shape> shape =
10        make_persistent<element_shape>(SNAKE_SEGMENT);
11       persistent_ptr<board_element> segp =
12        make_persistent<board_element>(
13           last_seg_position, shape, last_seg_dir);
14       snake_segments->push_back(segp);
15     }); }};
```

Figure 1. Panaconda with PMDK

enclosed by calls to library functions that implement a transaction or have to be passed to such a library function as a lambda. In summary, the approach is invasive. A programmer has to wrap every single member of a persistent data structure and has to define transactions explicitly. As the wrapper does not work on compound types, data structures might have to be decomposed using persistent pointers.

Pronto. Crash consistency in Pronto [14] is implemented by snapshots of the volatile program state in PMem. Between snapshots, it uses *asynchronous semantic logging*: A background thread logs the address and parameters of each public function call. After a crash, the calls can be replayed.

To create a persistent class, a wrapper class that aggregates the original type and inherits from a special base class is needed. It has to forward all public non-const methods of the aggregated type and informs the background thread. In addition, a custom memory allocator has to be used, and Pronto provides such an allocator that can be plugged into the containers of the C++ Standard Template Library (STL). Pronto requires a custom preprocessor to generate further methods for the wrapper classes, such as a custom `operator new()`, and a method to replay the log after a crash.

NVMReconstruct. NVMReconstruct [2] neither provides a persistent memory allocator nor a transaction mechanism and relies on external components implementing that functionality. It manages a persistent heap with objects that use regular pointers. Because the heap can be mapped into the applications' address space at different locations in each program run, the heap has to be reconstructed on startup: All pointers to persistent objects and member functions have to be fixed, and pointers to volatile data have to be reinitialized.

Data structures have to be annotated for reconstruction: Volatile pointers are annotated as `transient`, and a special `reconstructor` method can be defined that runs during the reconstruction phase. Persistent instances of annotated classes are allocated with `pnew` and deleted with `pdelete`, while `new` and `delete` are still available for creating volatile instances.

The new language features are implemented in a Clang compiler plugin that also collects metadata for the reconstruction phase. For example, the locations of all pointers have to be available for relocation and reinitialization.

Romulus. Romulus [3] is a library that uses two copies of the data to implement durable transactions on PMem. During a transaction, data are written to the primary copy, while a shadow copy remains untouched. Once the transaction ends, the shadow copy is updated from the primary data according to a volatile redo log that identifies the changed locations.

Application code has to wrap transactions in a pair of functions for single-threaded code, and in a lambda function as argument to `read_transaction` or `update_transaction` for multi-threaded code. Romulus provides custom functions to allocate and free persistent objects. Those objects have to be added and removed to/from the persistent store explicitly. Similar to libpmemobj++, Romulus requires members of persistent objects to be wrapped for intercepting all writes.

Summary. These examples show four recurring patterns that are key elements of the APIs of persistence libraries:

1. Programmers must use special allocation and deallocation functions to create and destroy persistent objects.
2. Classes, methods, or members of persistent objects have to be wrapped to make write operations interceptable or to provide hooks for the framework.
3. Classes or members have to be annotated to be able to discriminate between volatile and persistent objects.
4. Transactions have to be introduced explicitly.

The following sections show how the amount of code required to make regular data structures persistent can be reduced using Aspect-Oriented Programming (AOP) [10].

3 Aspect-Oriented PMDK

The code in Figure 1 shows an adapted excerpt from the example program *Panaconda*, which ships with libpmemobj++. As described in the previous section, all members of the class `snake` are wrapped: Instead of using a regular pointer, `snake_segments` is of the type `persistent_ptr` (line 2), and the types of the members `last_seg_position` and `last_seg_dir` are wrapped as well (lines 3–4). Lines 8–15 implement one transaction around two allocations of persistent objects.

In contrast, Figure 2 shows the corresponding regular C++ code for this example program. We will discuss the user-defined attribute `[[NVM::pmdk]]` in the following sections and will show that this is the only annotation needed to make the data structure persistent using two generic and fully reusable *aspects* written in AspectC++ [18].

3.1 Transactions and Undo-Logging

To implement crash consistency of persistent data structures with PMDK, all write operations and (de-)allocations have

```
1  class [[NVM::pmdk]] snake {
2    element_list *snake_segments;
3    point last_seg_position;
4    direction last_seg_dir;
5  public: // ...
6    void add_segment() {
7      element_shape *shape =
8        new element_shape(SNAKE_SEGMENT);
9      board_element *segp =
10       new board_element(last_seg_position,
11         shape, last_seg_dir);
12     snake_segments->push_back(segp);
13   }};
```

Figure 2. Panaconda with aspect-oriented approach

```
1  aspect PMDKTransactions {
2    pointcut transaction() = NVM::pmdk()
3      && !"% ...::%(...) const";
4    pointcut regular_members() = NVM::pmdk()
5      && !"%* ...::%" && !"static % ...::%";
6    advice execution(transaction()) : around() {
7      if (tx_running()) { tjp->proceed(); return; }
8      auto pop = pool_by_vptr(tjp->target());
9      transaction::run(pop, [&]{ tjp->proceed(); });
10   }
11   advice set(regular_members()) : before() {
12     transaction::snapshot(tjp->entity());
13   }};
```

Figure 3. Transaction aspect

to be run in transactions, and updated variables have to be logged. We assume that public methods of a class preserve class invariants according to the principles of object-oriented design, so that public non-const methods represent transactions (c. f. [12, 14]).

Figure 3 shows the implementation using AspectC++ to ensure that any non-const method of any annotated class is run as a transaction: We first define a *pointcut* `transaction` that represents all non-const methods of classes annotated with `[[NVM::pmdk]]`. The piece of *advice* in lines 6–9 intercepts the `execution` of such methods. If there is already a transaction running, there is nothing to do and the intercepted method can be executed via `tjp->proceed()` on the existing transaction (line 7). Else, the advice code looks up the memory pool in which the target object of the method is stored (line 8). Then a new transaction is started that executes the intercepted method in line 9.

Now that methods run in transactions, we need to save members to the undo log before write access. For regular (non-pointer) members, the implementation is straight forward: We first define a pointcut for the members that need to be logged in line 4. It contains all members in annotated classes that do not have a pointer type and that are not static. The `set` advice in line 11 is executed before any write access and takes a snapshot of the corresponding member.

3.2 Pointers

Pointers need to be handled differently. Because persistent objects can reside in different memory pools, pointers in libpmemobj++ contain the identifier of the memory pool and the offset to the pool's start address, requiring twice the storage of regular C++ pointers.

As the current version of AspectC++ can only add new members to a class, but not replace existing members, we cannot use the fat pointers of the PMDK. Therefore, our implementation uses libpmemobj++'s experimental self-relative persistent pointers that store the byte offset relative to their own address. Such pointers occupy the same space as regular C++ pointers, so that both regular and self-relative pointers can be translated transparently into each other without changing the object layout.

Figure 4 shows the pointer translation implemented as a reusable aspect. For using a regular pointer as a self-relative pointer, an instance of the class `self_relative_ptr` has to be created in the original pointer's memory. This is done by using the placement new operator on the pointer's address.

The template metaprogram `SRPLifetime` referenced in line 4 (not shown in this paper) uses the JoinPoint Template Library (JPTL) [1] and the compile-time introspection information provided by AspectC++ to iterate over all members and generate the functions `srp_init()` and `spr_destruct()` (lines 5–6) that construct and destruct a self-relative pointer in every pointer-type member. These methods are generated for all annotated classes, added to them using a class slice (line 3), and called before the constructor runs in a `construction` advice (line 8) and after the destructor in a `destruction` advice (line 9).

With the self-relative pointers in place, we can now translate reads and writes to pointers. The `set` advice in line 10 replaces the writes by the advice body that determines the type of the self-relative pointer that corresponds to the original pointer type given by `JoinPoint::Entity`. The value that the application assigns to the pointer is available in `*tjp->arg<0>()` and is used to construct a self-relative pointer. Finally, the member is interpreted as a self-relative pointer and assigned its new value.

Reads from pointers are handled similarly as shown in the `get` advice in line 14: The member's memory is interpreted as a self-relative pointer and we return the corresponding regular pointer via `*tjp->result()`.

3.3 Allocation of Persistent Objects

Persistent objects in libpmemobj++ are created and deleted using special functions during a transaction. The same functionality can be achieved with custom `new` and `delete` operators. These operators are introduced transparently into annotated classes by a slice introduction using AspectC++ to implement the functionality of `make_persistent` and `delete_persistent` from the PMDK.

```
1  aspect PMDKPointers {
2    pointcut pointer_members() = NVM::pmdk() && "%* ...::%" && !"static % ...::%";
3    advice NVM::pmdk() : slice class {
4      using SRPLifetime = JPTL::MemberIterator<JoinPoint, SRPLifetimes>::EXEC::SRPLifetime;
5      void srp_init() { SRPLifetime::srp_init(this); }
6      void srp_destruct() { SRPLifetime::srp_destruct(this); }
7    };
8    advice construction(NVM::pmdk()) : before() { tjp->target()->srp_init(); } // Construct self_relative_ptrs
9    advice destruction(NVM::pmdk()) : after() { tjp->target()->srp_destruct(); } // Destruct self_relative_ptrs
10   advice set(pointer_members()) : around() {
11     using PTR = self_relative_ptr<std::remove_pointer_t<JoinPoint::Entity>>;
12     *reinterpret_cast<PTR*>(tjp->entity()) = PTR(*tjp->arg<0>());
13   }
14   advice get(pointer_members()) : around() {
15     using PTR = self_relative_ptr<std::remove_pointer_t<JoinPoint::Entity>>;
16     *tjp->result() = reinterpret_cast<PTR*>(tjp->entity())->get();
17   }};
```

Figure 4. Pointer aspect that transparently transforms pointer members into self-relative PMDK pointers

3.4 Limitations

In addition to the limitations of PMDK[1], we currently do not support transactions that acquire locks. The PMDK supports acquiring locks at the beginning of a transaction, but they have to be specified in the call to `transaction::run(...)`. A current limitation of the self-relative pointers is that they can only point to objects in the same memory pool.

4 Aspect-Oriented Pronto

This section presents a prototypical aspect-oriented implementation of the Pronto [14] preprocessor[2]. Although Pronto uses a custom preprocessor, there is still some boilerplate code that a programmer has to write. Figure 5 shows a wrapper required for an STL `vector<char*>` as presented in the Pronto paper [14]. The wrapper has to be derived from the class `PersistentObject` (line 1) and needs to define a constructor that forwards an object identifier to the constructor of that base class (line 3) and allocates the wrapped vector instance. Every method has to be forwarded, and non-const methods such as `push_back()` in line 6 have to be enclosed by calls to `op_begin()` and `op_commit()`.

Figure 6 shows the code required for the same task with our aspect-oriented prototype. We still need a wrapper class that inherits from `PersistentObject`, but no methods have to be defined at all. The class derives from `vector`, so that all of its methods are available. By annotating the class with the attribute `[[AOPronto::persist]]` in line 1, AspectC++ generates the code needed by the Pronto runtime automatically. This wrapper can be adapted to different STL containers by replacing "vector" with the desired container class.

The original Pronto preprocessor generates several factory methods that are used by Pronto's runtime to generate instances of the wrapped type. This way Pronto ensures that

```
1  struct PersistableVector : public PersistentObject {
2    typedef char * T;
3    PersistableVector(uuid_t id): PersistentObject(id){
4      v_vector = new vector<T, STLAlloc<T>>;
5    }
6    void push_back(T value) {
7      op_begin();
8      v_vector->push_back(value);
9      op_commit();
10   }
11   size_t size() const { return v_vector->size(); }
12 private:
13   vector<T, STLAlloc<T>> *v_vector;
14 };
```

Figure 5. Persistable vector using the Pronto API

```
1 struct [[AOPronto::persist]]
2 PersistableVector : public PersistentObject,
3     public vector<char *, STLAlloc<char *>>
4 { typedef char * wrappedType;};
```

Figure 6. Persistable vector with AspectC++ Pronto support

new instances are allocated using Pronto's custom memory allocator, are assigned a unique object identifier, and can be recovered from snapshots in case of a crash. The factory methods and two constructors are also inserted into annotated classes using AspectC++ (not shown in this paper).

Pronto's log can be filled using the `AC::Action` structure that AspectC++ uses to execute the original code replaced by `around` advice. In advice code, it encapsulates the context information needed to execute the original program code at the current join point [18], including pointers to arguments and the function. For Pronto's log, the actual arguments have to be copied into memory managed by Pronto and the argument pointers in the `AC::Action` structure have to be adjusted.

Figure 7 shows a simplified excerpt of the aspect that implements the logging feature. The piece of `call` advice in lines 1–7 notifies the background logging-thread on method

[1]Objects need to be trivially copyable, so they can be stored in the undo log.
[2]According to Pronto's authors the original preprocessor and its source code are lost. We thus had to deduce its functionality from the output available in the Pronto artifact archive [15]. Our prototype is therefore limited, but able to successfully run the vector benchmark from said archive.

```
1  advice call(/*...*/&&"% ...::%()"): around() {
2    Savitar_thread_notify(3, tjp->target(),
3      1, &(tjp->action()));
4    tjp->proceed();
5    Savitar_thread_wait(tjp->target(),
6      tjp->target()->log);
7  }
8  advice persistentClass() : slice class {
9    //...
10   size_t Play(uint64_t tag,
11     uint64_t *args, bool dry) {
12     //...
13     AC::Action * action = (AC::Action *)args;
14 /*...Unpack arguments, fill action->_args...*/
15     if (!dry) {
16       Savitar_thread_notify(2, this, 1);
17       action->trigger();
18       Savitar_thread_wait(this, this->log);
19     }
20     //...
21   }};
```

Figure 7. Simplified excerpt from the Pronto aspect using AC::Action for logging and replaying of method calls

call. For brevity, we assume that the method has no arguments, so that a pointer to the `AC::Action` instance returned from `tjp->action()` can be used without modifications. After the original method has been executed using `tjp->proceed()`, the advice code waits for the background thread to complete its log entry.

When the log is replayed, Pronto calls the method `Play` (lines 10–21). It finds the logged `AC::Action` instance at the start of the argument buffer `args`. If there were arguments, they would have to be extracted from `args` and linked in the array `action->_args` to rebuild the `AC::Action` structure. Finally, the method can be replayed by calling `action->trigger()`, again logged to make this replay crash-safe.

We currently use the parameter `tag` to store the argument count. If this parameter was combined with `AC::Action` in Pronto's log, logging performance could be improved as shown in Subsection 5.2, but this would require a change to Pronto's runtime.

5 Evaluation

In this section, we evaluate our aspects for PMDK and Pronto.[3]

5.1 PMDK

For the evaluation[4], we converted the PMDK example program Panaconda to regular C++ code. Panaconda is an implementation of the game Snake that stores its state in a persistent memory pool. Figure 8 shows a simplified UML class diagram of the program, in which all classes except `game` use persistent memory. For instance, the class template `list` implements a doubly linked data structure that dynamically allocates and deletes persistent objects.

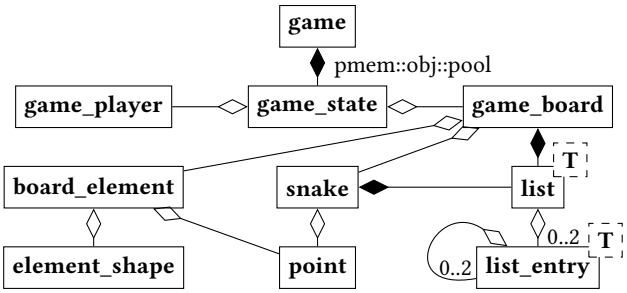

Figure 8. Simplified UML class diagram of Panaconda

We instantiated that class template explicitly for the type `board_element`, because AspectC++ does not support weaving in class templates, yet. Thus, we could apply our aspect-oriented approach to all classes that use persistent memory as depicted in Figure 8.

Correctness. To make sure that we do not introduce consistency bugs, we ran *pmemcheck* from the PMDK, *PMDebugger* [5], and *Intel Inspector* [9] on both implementations of Panaconda. We started each game on a fresh pool, let the snake run into a wall and then quit the game.

For the original implementation from the PMDK, Intel Inspector reported 104 errors: 103 times "Store without undo log", and one time "Undo log without update". No errors were reported for the aspect-oriented implementation.

Framework-specific lines of source code. To quantify the effect of our approach on the direct use of libpmemobj++ in the source code, we counted the lines of code that directly use parts of the framework in the original version of Panaconda and in our aspect-oriented version, and the total lines of code excluding comments and empty lines. In the original code, we found libpmemobj++-specific code in 146 out of 1,104 lines (13.22 %), in the aspect-oriented version in 28 out of 1,091 lines (2.57 %).

The remaining 28 lines deal with loading and setting up the persistent memory pool and cannot be automated easily.

Performance. We compare the execution time of different push and pop operations on the doubly linked list that was used in the Panaconda example[5] to a version of the list that has been augmented with our aspects, and a variant of the original list that uses `self_relative_ptr` instead of `persistent_ptr`.

[3]All measurements were carried out on a Dell PowerEdge R740 server with 2x Intel Xeon Gold 5218 CPU running at 2.3 GHz, 384 GiB DRAM (12×32 GiB DDR4-2666), and 1.5TiB Intel Optane Persistent Memory 100 Series (12×128 GiB) with HyperThreading and TurboBoost both disabled. We used one region of 6 interleaved DCPMMs on socket 0 in AppDirect mode with EXT4 mounted with `-o dax`. The server ran Debian Bullseye with Debian kernel 5.10.46-3.

[4]We used PMDK 1.11.0 and libpmemobj++ 1.13.0 compiled from source, AspectC++ 2.3 (Debian 2.3-4), and GCC 10.2.1 with flags -DNDEBUG -O2.

[5]We extended the list implementation by a destructor that deletes all list entries, we changed deleting entries, so that the values contained in list entries are not deleted when a list entry is deleted, and we fixed a leak of a deleted list_entry in the pop_back method.

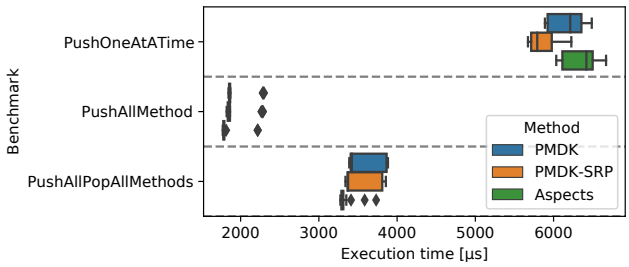

Figure 9. Execution time of list benchmarks (1000 elements)

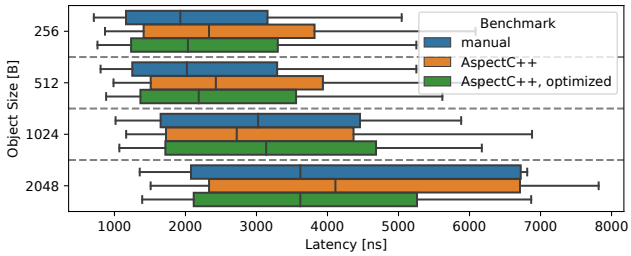

Figure 10. Pronto benchmark

The benchmarks for both PMDK and aspects operate on a memory pool of 1 GiB size that is initialized with 100000 objects of a point class with random coordinate values. An array in the pool stores persistent pointers to these objects so that they can be used for all benchmarks.

We evaluated the following scenarios of iterated operations for N = 10, 100, 1000, 10000, and 100000 elements: Push all elements with a transaction inside the loop (*PushOneAtATime*), push all elements in a method defined on the list (*PushAllMethod*), and push all elements in a method defined on the list, then pop all elements in another method defined on the list (*PushAllPopAllMethods*). We ran each benchmark 20 times and deleted the linked list after each run of a single benchmark.

Figure 9 shows the execution time of the different benchmarks on the original list (PMDK), the list with self-relative pointers (PMDK-SRP), and the list using AspectC++ (Aspects) for 1000 elements. All list versions show a comparable performance, with the SRP version slightly better than the PMDK version. The Aspects version performs slightly better for all benchmarks but PushOneAtATime. For more than 1000 elements, the median execution time for the Aspects version is up to 3.2 % slower than the PMDK version for PushOneAtATime, and almost identical to the PMDK version for the other benchmarks.

5.2 Pronto

We ran the existing vector benchmark available from the Pronto artifact archive version 1.1 [15] using the GCC 7.5.0 compiler. In the original benchmark, the authors compare the average latency of insertions into a volatile STL vector and a wrapped vector that is persisted by Pronto (*manual*). We added two vectors to that benchmark: the aspect-augmented version of the vector (*AspectC++*), and a vector that uses `AC::Action` without a leading log entry for a `tag` (AspectC++, *optimized*).

The results of the benchmark are shown in Figure 10. For the *manual* Pronto benchmark, we can reproduce the results presented in Figure 7 of the original paper. The latency for the *AspectC++* version of the vector is about 1 µs larger than for the *manual* Pronto version, while the *optimized* AspectC++ version performs similar to the *manual* Pronto version.

6 Related Work

In addition to using software frameworks as analyzed in Section 2, different approaches to handle persistent data have been discussed.

Coy et al. [4] and Elkhouly et al. [6] propose to extend compilers with special `pragma` statements to apply checkpointing in parts of the code. This approach uses annotations to introduce persistence into programs, but switching to another framework would require a compiler change.

Other researchers suggest extensions of languages by persistence concepts: Kolli et al. [11] propose an extended C++ memory model, Cohen et al. [2] add keywords to C++ to discriminate between volatile and persistent data, and go-pmem [7] extends the Go language by a persistent heap. While persistence-aware languages enable optimizations by the compiler, they limit flexibility for programmers to change persistence methods in different parts of the code, which our approach does allow.

AOP has been used to apply transactions to annotated code by Riegel et al. [17] in Java and Köppen et al. [12] using AspectC++. The latter approach builds a tailored data-structure to apply cache-line transactions on PMem [19], underlining the flexibility of a solution based on AspectC++.

7 Conclusion

AOP and the syntax of AspectC++ might at first appear difficult and complex. However, using user-defined attributes, the integrated introspection API, and template metaprograms are common AspectC++ idioms. No new language elements had to be introduced for our two examples.

The power of the approach lies in the ability to separate the technical details of a persistence framework from the application logic without the need for special-purpose preprocessors or compiler/language extensions. The separation allows developers to use ordinary classes for persistent objects, to evolve the code more easily, and to change the persistence framework if necessary.

The minimal performance degradation that we measured has no fundamental reasons. We hope that it can be avoided completely by fine-tuning our aspects or improving the code patterns generated by the aspect weaver in the near future.

References

[1] Christoph Borchert. 2017. *Aspect-Oriented Technology for Dependable Operating Systems.* Dissertation. Technische Universität Dortmund. https://doi.org/10.17877/DE290R-17995

[2] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-Oriented Recovery for Non-Volatile Memory. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 153:1–153:22. https://doi.org/10.1145/3276523

[3] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18).* Association for Computing Machinery, New York, NY, USA, 271–282. https://doi.org/10.1145/3210377.3210392

[4] Tyler Coy, Shuibing He, Bin Ren, and Xuechen Zhang. 2020. Compiler Aided Checkpointing Using Crash-Consistent Data Structures in NVMM Systems. In *Proceedings of the 34th ACM International Conference on Supercomputing (ICS '20).* Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3392717.3392755

[5] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021).* Association for Computing Machinery, New York, NY, USA, 503–516. https://doi.org/10/gjrs5t

[6] Reem Elkhouly, Mohammad Alshboul, Akihiro Hayashi, Yan Solihin, and Keiji Kimura. 2019. Compiler-Support for Critical Data Persistence in NVM. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (Dec. 2019). https://doi.org/10.1145/3371236

[7] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. 2020. Go-Pmem: Native Support for Programming Persistent Memory in Go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20).* 859–872. https://www.usenix.org/conference/atc20/presentation/george

[8] Intel. 2019. *Persistent Memory Development Kit.* https://github.com/pmem/pmdk

[9] Intel. 2021. *Intel Inspector.* https://www.intel.com/content/www/us/en/develop/tools/oneapi/components/inspector.html

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *ECOOP'97 — Object-Oriented Programming (Lecture Notes in Computer Science)*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer, Berlin, Heidelberg, 220–242. https://doi.org/10.1007/BFb0053381

[11] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-Level Persistency. *ACM SIGARCH Computer Architecture News* 45, 2 (June 2017), 481–493. https://doi.org/10.1145/3140659.3080229

[12] Marcel Köppen, Jana Traue, Christoph Borchert, Jörg Nolte, and Olaf Spinczyk. 2019. Cache-Line Transactions: Building Blocks for Persistent Kernel Data Structures Enabled by AspectC++. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems (PLOS '19).* Association for Computing Machinery, New York, NY, USA, 38–44. https://doi.org/10.1145/3365137.3365396

[13] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. 1991. The ObjectStore Database System. *Commun. ACM* 34, 10 (Oct. 1991), 50–63. https://doi.org/10.1145/125223.125244

[14] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20).* Association for Computing Machinery, New York, NY, USA, 789–806. https://doi.org/10.1145/3373376.3378456

[15] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. *Pronto: Easy and Fast Persistence for Volatile Data Structures (1.1).* Lausanne, Switzerland. https://doi.org/10.5281/zenodo.3605351

[16] 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 International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48).* Association for Computing Machinery, New York, NY, USA, 672–685. https://doi.org/10.1145/2830772.2830802

[17] Torvald Riegel, Christof Fetzer, and Pascal Felber. 2006. Snapshot Isolation for Software Transactional Memory. In *First ACM SIGPLAN Workshop on Languages, Compilers and Hardware Support for Transactional Computing.*

[18] Olaf Spinczyk and Daniel Lohmann. 2007. The Design and Implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software* 20, 7 (Oct. 2007), 636–651. https://doi.org/10.1016/j.knosys.2007.05.004

[19] Jana Traue. 2018. *Fine-Grained Transactions for NVRAM.* Ph.D. Dissertation. BTU Cottbus - Senftenberg, Germany. https://opus4.kobv.de/opus4-btu/frontdoor/index/index/docId/4636