persistent memory semantics in programming languages

overview of the ongoing research

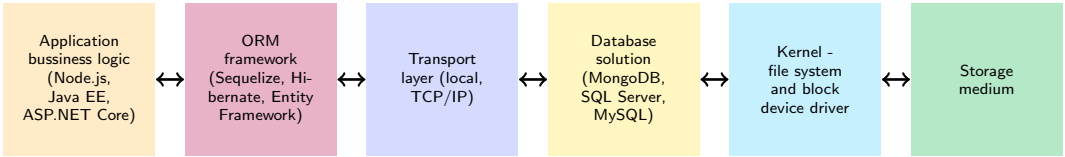
Piotr Balcer

Intel Technology Poland
piotr.balcer@intel.com

October 6, 2016

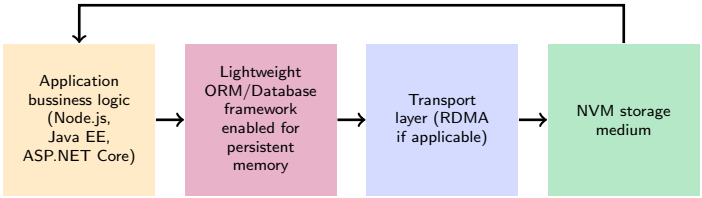
persistent memory primer

- Next-Gen NVDIMMs provide fast, byte-addressable and non-volatile memory
- This unique combination will allow data to live much closer to the code
- Traditional storage stack wastes a lot of CPU time and gets away with it because the underlying medium is slow



persistent memory primer

- Next-Gen NVDIMMs provide fast, byte-addressable and non-volatile memory
- This unique combination will allow data to live much closer to the code
- Traditional storage stack wastes a lot of CPU time and gets away with it because the underlying medium is slow
- Potentially zero-copy software storage stack



problem statement

- Got a huge block of non-volatile memory in the address space, now what?
- Programming ecosystems were not designed with the notion of byte-addresable memory being persistent

libc with persistent memory?

```
void *ptr = malloc(sizeof (struct my_persistent_data)); //from pmem
```

This can be made to work[1]... if you can guarantee that your application will always gracefully exit OR you don't care about your data (e.g. caching)

[1] Libraries enabling this use case: libvmem, libmemkind

persistency domain

- Persistent memory introduces the concept of fail-safe atomicity
- Need to remember about various memory hiding places in the CPU pipeline (CPU caches, memory controller)

Example

```
strcpy(&persistent_memory_buffer, "Brianna");
```

Power interruption at various stages:

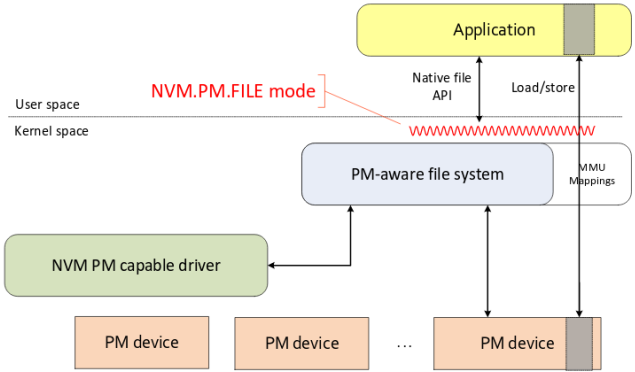
- `00000000` - before copying started
- `Brian000` - in between flushing cachelines to the storage medium
- `000anna0` - data can get reordered
- `Brianna0` - success!

chronology of select work in the persistent memory research field

March 2011	Publication of Mnemosyne and NV-Heaps papers
October 2014	Publication of Atlas programming model paper from HP Labs
January 2015	Publication of REWIND research paper
March 2015	NVM Direct source code first publised by Oracle
June 2015	NVML: First significant release of libpmemobj
February 2016	Publication of NOVA: NVM Log-structured File System
April 2016	Publication of pVM: Persistent Virtual Memory paper
April 2016	Publication of NVL-C paper

SNIA NVM Programming Model

- Most of the industry has come together in the SNIA NVM Programming Technical Work Group and produced the **NVM Programming Model**
- `http://www.snia.org/tech_activities/standards/curr_standards/npm`



- One of the most influential body of work
- A very complex solution that requires custom kernel, compiler and additional libraries
- Introduces a lot of new, unique at the time, concepts

Example

```
1 pstatic htRoot = NULL;
2
3 update_hash(key, value) {
4   atomic {
5     pmalloc(sizeof(*bucket), &bucket));
6     bucket->key = key;
7     bucket->value = value;
8     insert(hash, bucket);
9   }
10 }
```

Mnemosyne: Lightweight Persistent Memory
by Michael M. Swift, Andres Jaan Tack, Haris Volos

`https://research.cs.wisc.edu/sonar/projects/mnemosyne/`

A closer look at the mnemosyne API

```
1  pstatic htRoot = NULL;
2
3  int
4  main()
5  {
6    if (!htRoot)
7      pmalloc(N * sizeof(*bucket), &htRoot);
8
9    ...
10 }
11
12 void
13 update_hash(key, value)
14 {
15   atomic {
16     pmalloc(sizeof(*bucket), &bucket);
17     bucket->key = key;
18     bucket->value = value;
19     insert(hash, bucket);
20   }
21 }
```

- Static persistent variable, survives application restarts
- In this case it will contain the address of a root object
- Root object is the start point of persistent applications

A closer look at the mnemosyne API

```
1  pstatic htRoot = NULL;
2
3  int
4  main()
5  {
6    if (!htRoot)
7      pmalloc(N * sizeof(*bucket), &htRoot);
8
9    ...
10 }
11
12 void
13 update_hash(key, value)
14 {
15   atomic {
16     pmalloc(sizeof(*bucket), &bucket);
17     bucket->key = key;
18     bucket->value = value;
19     insert(hash, bucket);
20   }
21 }
```

- Special memory allocator interface
- Allows for atomic operations
- The hashmap is allocated into the previously mentioned static persistent root

A closer look at the mnemosyne API

```
1 pstatic htRoot = NULL;
2
3 int
4 main()
5 {
6   if (!htRoot)
7     pmalloc(N * sizeof(*bucket), &htRoot);
8
9   ...
10 }
11
12 void
13 update_hash(key, value)
14 {
15   atomic {
16     pmalloc(sizeof(*bucket), &bucket);
17     bucket->key = key;
18     bucket->value = value;
19     insert(hash, bucket);
20   }
21 }
```

- **Atomic** keyword marks the ACID-like transaction
- In principle similar to software transactional memory

Transactional Synchronization Extensions (TSX)

TSX is a recent addition to the x86 ISA that allows executing code atomically in terms of visiblity to other threads. It has nothing to do with fail-safe atomicity!

NV-Heaps

- The first attempt at next-gen persistent memory in C++
- Explicit file-backed persistent memory region and root objects
- Relative C++ pointers
- Automatic garbage collection

Example

```
1 class NVList : public NVObject
2   DECLARE_POINTER_TYPES(NVList);
3   DECLARE_MEMBER(int, value);
4   DECLARE_PTR_MEMBER(NVList::NVPtr, next);
5
6 void remove(int k) {
7   NVHeap * nv = NVHOpen("foo.nvheap");
8   NVList::VPtr a = nv->GetRoot<NVList::NVPtr>();
9   AtomicBegin {
10    if (a->get_next() != NULL) a->set_value(5);
11  } AtomicEnd; }
```

NV-Heaps: making persistent objects fast and safe with next-gen NVM
by J Coburn, A M. Caulfield, A Akel, L M. Grupp, R K. Gupta, R Jhala, S Swanson

https://cseweb.ucsd.edu/~swanson/papers/Asplos2011NVHeaps.pdf

- Observes that visibility (threading) and persistency (fail safety) critical sections are similar
- Uses locks for transaction boundaries
- The first use of LLVM to introduce persistent memory semantics

Example

```
1 typedef struct object { int value; } object;
2
3 int main() {
4   NVM_Initialize();
5   uint32_t my_region = NVM_FindOrCreateRegion(
6     "/mnt/pmem/my_pmem_region", O_RDWR, NULL);
7   object *obj = nvm_alloc(sizeof(object));
8   lock(...); obj->value = 5; unlock(...);
9   NVM_CloseRegion(my_region);
10  NVM_Finalize();
11 }
```

Atlas: Leveraging Locks for Non-volatile Memory Consistency
by Dhruva R. Chakrabarti, Hans-J. Boehm, Kumud Bhandari

`https://github.com/HewlettPackard/Atlas`

REWIND

- Comprehensive comparision versus traditional DBMSes
- Proposes using STM like support for persistent updates
- Interesting comparision of internal logging methods

Example

```
1 void  remove (node* n) {
2   persistent atomic {
3     if (n == tail) tail = n->prv;
4     if (n == head) head = n->nxt;
5     if (n->prv) n->prv->nxt = n->nxt;
6     if (n->nxt) n->nxt->prv = n->prv;
7     delete (n);
8   }
9 }
```

REWIND: Recovery Write-Ahead System for In-Memory NV Data-Structures
by Andreas Chatzistergiou, Marcelo Cintra, Stratis D. Viglas

http://www.vldb.org/pvldb/vol8/p497-chatzistergiou.pdf

- A set of simple libraries. Hardware agnostic, no custom kernels or compilers required
- Implements the SNIA programming model
- Provides atomic API in addition to transactions (just like CAS and TM)

Example

```
1 typedef struct foo {
2   PMEMoid bar; // persistent pointer
3   int value;
4 } foo;
5
6 int main() {
7   PMEMobjpool *pop = pmemobj_open(...);
8   TX_BEGIN(pop) {
9     TOID(foo) root = POBJ_ROOT(foo);
10    D_RW(root)->value = 5;
11  } TX_END; }
```

NVML: Non-Volatile Memory Library
by Intel Corporation

`http://pmem.io`

An example of libpmemobj C transaction

```
1  struct list { TOID(struct node) first; }
2  struct node { TOID(struct node) next; int value; }
3  int main() {
4    PMEMobjpool *pop = pmemobj_open("my_list", ...);
5    add_element(pop, POBJ_ROOT(pop, struct list), 5);
6  }
7  int add_element(PMEMobjpool *pop,
8      TOID(struct list) list, int value) {
9    TX_BEGIN(pop) {
10     TOID(struct node) n = TX_ALLOC(struct node);
11     D_RW(n)->value = value;
12     D_RW(n)->next = D_RW(list)->first;
13     TX_SET(D_RW(list), first, n);
14   } TX_ONCOMMIT {
15     ret = 0;
16   } TX_ONABORT {
17     ret = 1;
18   } TX_END
19
20   return ret;
21 }
```

- Definition of persistent types
- Special TOID macro is used to provide type-safety without compiler extensions

An example of libpmemobj C transaction

```
1 struct list { TOID(struct node) first; }
2 struct node { TOID(struct node) next; int value; }
3 int main() {
4   PMEMobjpool *pop = pmemobj_open("my_list", ...);
5   add_element(pop, POBJ_ROOT(pop, struct list), 5);
6 }
7 int add_element(PMEMobjpool *pop,
8     TOID(struct list) list, int value) {
9   TX_BEGIN(pop) {
10    TOID(struct node) n = TX_ALLOC(struct node);
11    D_RW(n)->value = value;
12    D_RW(n)->next = D_RW(list)->first;
13    TX_SET(D_RW(list), first, n);
14  } TX_ONCOMMIT {
15    ret = 0;
16  } TX_ONABORT {
17    ret = 1;
18  } TX_END
19
20  return ret;
21 }
```

- Instantiation of the object memory pool, a representation of a persistent memory region.
- The file is opened, the underlying storage type is identified and appropriate storage synchronization primitives are loaded (clflush, msync, or even a flush over fabric - replication)

An example of libpmemobj C transaction

```
1 struct list { TOID(struct node) first; }
2 struct node { TOID(struct node) next; int value; }
3 int main() {
4   PMEMobjpool *pop = pmemobj_open("my_list", ...);
5   add_element(pop, POBJ_ROOT(pop, struct list), 5);
6 }
7 int add_element(PMEMobjpool *pop,
8     TOID(struct list) list, int value) {
9   TX_BEGIN(pop) {
10    TOID(struct node) n = TX_ALLOC(struct node);
11    D_RW(n)->value = value;
12    D_RW(n)->next = D_RW(list)->first;
13    TX_SET(D_RW(list), first, n);
14  } TX_ONCOMMIT {
15    ret = 0;
16  } TX_ONABORT {
17    ret = 1;
18  } TX_END
19
20  return ret;
21 }
```

- The root object concept is still alive
- In libpmemobj the root object always exists and is initially zeroed

An example of libpmemobj C transaction

```
1  struct list { TOID(struct node) first; }
2  struct node { TOID(struct node) next; int value; }
3  int main() {
4    PMEMobjpool *pop = pmemobj_open("my_list", ...);
5    add_element(pop, POBJ_ROOT(pop, struct list), 5);
6  }
7  int add_element(PMEMobjpool *pop,
8      TOID(struct list) list, int value) {
9    TX_BEGIN(pop) {
10     TOID(struct node) n = TX_ALLOC(struct node);
11     D_RW(n)->value = value;
12     D_RW(n)->next = D_RW(list)->first;
13     TX_SET(D_RW(list), first, n);
14   } TX_ONCOMMIT {
15     ret = 0;
16   } TX_ONABORT {
17     ret = 1;
18   } TX_END
19
20   return ret;
21 }
```

- Due to the fact libpmemobj does not extend the compiler or add precompilation step to the compilation process, a set of inventive transactional macros was created.
- The C-style exceptions are created using setjmp and longjmp

An example of libpmemobj C transaction

```
1 struct list { TOID(struct node) first; }
2 struct node { TOID(struct node) next; int value; }
3 int main() {
4   PMEMobjpool *pop = pmemobj_open("my_list", ...);
5   add_element(pop, POBJ_ROOT(pop, struct list), 5);
6 }
7 int add_element(PMEMobjpool *pop,
8     TOID(struct list) list, int value) {
9   TX_BEGIN(pop) {
10    TOID(struct node) n = TX_ALLOC(struct node);
11    D_RW(n)->value = value;
12    D_RW(n)->next = D_RW(list)->first;
13    TX_SET(D_RW(list), first, n);
14  } TX_ONCOMMIT {
15    ret = 0;
16  } TX_ONABORT {
17    ret = 1;
18  } TX_END
19
20  return ret;
21 }
```

- Manual instrumentation is required for effectively every memory operation
- Special transactional memory allocator routines are provided

- A template library that brings persistent memory into modern C++
- Seamless integration with the language, extensive use of all the new features
- Ongoing work on integration with the C++ standard library

Example

```
1 class foo {
2   persistent_ptr<bar> bar_ptr;
3   p<int> bar_value;
4 };
5
6 int main() {
7   pool<foo> pop = pool<foo>::open(...);
8   auto f = pop.get_root();
9   transaction::exec_tx(pop, [&] {
10    f->value = 5; }
11 }
```

Persistent Memory Extensions to libstdc++/libc++

For more information on the topic of C++ see the talk by Tomasz Kapela later today (16:00 in Potsdam I-II)

- AFAIK the first public attempt at enabling pmem in an interpreted language
- Builds on top of NVML
- Currently written entirely in python, future performance work planned in CPython

Example

```
1 class Foo(PersistentObject):
2   def __init__(self):
3     self.bar = 0
4
5 pop = PersistentObjectPool("my_pmem_data", "c")
6 if pop.root is None:
7   pop.root = pop.new(Foo)
8
9 with pop.transaction():
10  pop.root.bar = 5
```

pynvm: Non-volatile memory for Python
by Christian S. Perone, R. David Murray

`https://github.com/pmem/pynvm/`

Singly-linked list in pynvm

```
1  from nvm.pmemobj import (PersistentObjectPool, PersistentObject)
2  class Node(PersistentObject):
3    def __init__(self, next, value):
4      self.next = next
5      self.value = value
6
7  class List(PersistentObject):
8    def __init__(self):
9      self.first = None
10
11   def insert(self, value):
12     with self._p_mm.transaction():
13       self.first =
14         self._p_mm.new(Node, self.first, value)
15
16   def print_all(self):
17     n = self.first
18     while (n is not None):
19       print(n.value)
20       n = n.next
21
22 pop = PersistentObjectPool("my_pmem_data", "c")
```

- A special PersistentObject class, when inherited, automagically makes a persistent object

Singly-linked list in pynvm

```
1 from nvm.pmemobj import (PersistentObjectPool, PersistentObject)
2 class Node(PersistentObject):
3   def __init__(self, next, value):
4     self.next = next
5     self.value = value
6
7 class List(PersistentObject):
8   def __init__(self):
9     self.first = None
10
11  def insert(self, value):
12    with self._p_mm.transaction():
13      self.first =
14        self._p_mm.new(Node, self.first, value)
15
16  def print_all(self):
17    n = self.first
18    while (n is not None):
19      print(n.value)
20      n = n.next
21
22 pop = PersistentObjectPool("my_pmem_data", "c")
```

- Context manager that governs over the lifecycle of a transaction
- Replaces the ugly and error-prone C macros
- There's no need to use manually add objects to a transaction

Singly-linked list in pynvm

```
1 from nvm.pmemobj import (PersistentObjectPool, PersistentObject)
2 class Node(PersistentObject):
3   def __init__(self, next, value):
4     self.next = next
5     self.value = value
6
7 class List(PersistentObject):
8   def __init__(self):
9     self.first = None
10
11  def insert(self, value):
12    with self._p_mm.transaction():
13      self.first =
14        self._p_mm.new(Node, self.first, value)
15
16  def print_all(self):
17    n = self.first
18    while (n is not None):
19      print(n.value)
20      n = n.next
21
22 pop = PersistentObjectPool("my_pmem_data", "c")
```

- A simple persistent object pool manager
- It's carried around by all PersistentObjects for easy access

- Introduces a concept of automatically releasing persistent locks
- Very verbose and heavy extensions to C

Example

```
1 persistent struct mystruct { nvm_mutex mutex;
2 int count; }
3
4 nvm_desc desc; //region descriptor
5 int increment(mystruct ^my) {
6   int ret;
7   @ desc {
8     nvm_xlock(%my=>mutex);
9     my=>count@++;
10    ret = my=>count;
11  } return ret; }
```

NVM Direct: Open Source Non-Volatile Memory API
by Bill Bridge, Oracle

`https://github.com/oracle/nvm-direct`

pVM

- Entirely novel approach, extends the virtual memory kernel subsystem
- Adds a nvmmap(), increases performance by avoiding VFS

Example

```
1 imgobj = nvmalloc(obj1, size, NULL)
2
3 BEGIN_TRANS(imgobj)
4   memcpy(imgobj, pixels, size)
5   convert_image(imgobj)
6 END_TRANS(imgobj)
```

pVM: persistent virtual memory for efficient capacity scaling and object storage by Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan

http://dl.acm.org/citation.cfm?id=2901325

NVL-C

- By far the most comprehensive research into pmem C extension
- A modular implementation based on LLVM, a base for further work
- Uses libpmemobj as persistency backend

Example

```
1 void add(int k) {
2   nvl_heap_t *heap = nvl_open("foo.nvl");
3   nvl list *a = nvl_get_root(heap, list);
4
5   #pragma nvl atomic heap(heap) {
6     nvl list *b = nvl_alloc_nv(heap, 1, list);
7     b->value = k; b->next = a->next; a->next = b;
8     nvl_close(heap);
9   }
10 }
```

NVL-C: Static Analysis Techniques for Efficient, Correct Programming of NVM by Joel E. Denny, Seyong Lee, Jeffrey S. Vetter

`http://dl.acm.org/citation.cfm?id=2907303`

summary

- Many approaches to managing persistent memory have emerged in the last couple of years. Some of them stand in a complete contradiction of others, many build on top of one another, but all of them advance the state of the art.
- Let's hope at least one of us is right ;)

Thank you! Questions?