Storing STL Containers on NVM

Persistent Programming in Real Life (PIRL) 2019

HPI Hasso Plattner Institut

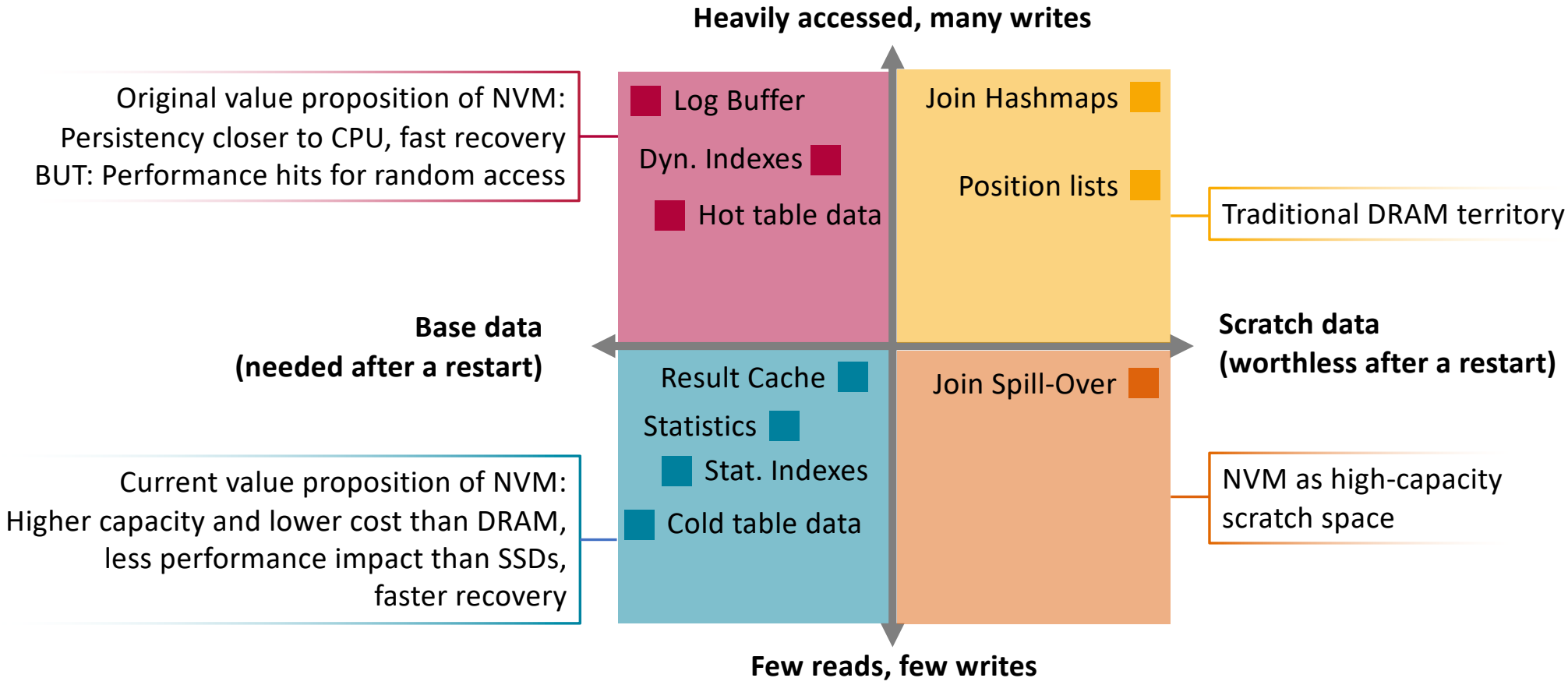
Topics Covered

- Data Characteristics in In-Memory Databases
  - The Importance of Write-Once Data
- Using PMR and Representation-Aware Containers for NVM
- Performance Evaluation

Background

- Hyrise is a Research Database developed at HPI in Potsdam, Germany
- Relational, in-memory, HTAP, open-source, C++2a
- Research topics: Self-Driving, NVM, Heterogeneous Replication, Footprint Reduction
  - Note that not everyone is working on NVM
- Similar in many concepts to SAP HANA

Data Characteristics in In-Memory Databases



Goals

- Allow an existing code base to benefit from NVM; minimize the necessary changes
  - Avoid making more code than necessary NVM-aware
  - Make it easy for non-NVM people
- Enable the use of data structures that do not yet have an NVM-aware equivalent

- Limitations that make our live easier:
  - We only write this data once
  - We do not have to care about atomicity

Disclaimer

- Casting memory that came from somewhere into a C++ object is undefined behavior
- … but it works.

- This might break if the program is recompiled using a different compiler or a different STL
- … and you won't even notice it.

- For this to be used in a product, more work needs to be done
- … but we are researchers.

Code Example

- Let‘s take a simplified in-memory table with two columns:

```
std::vector<std::tuple<double, int>> table{
  {4.2, 5},
  {1.4, 7},
  {3.5, 2}
};
```

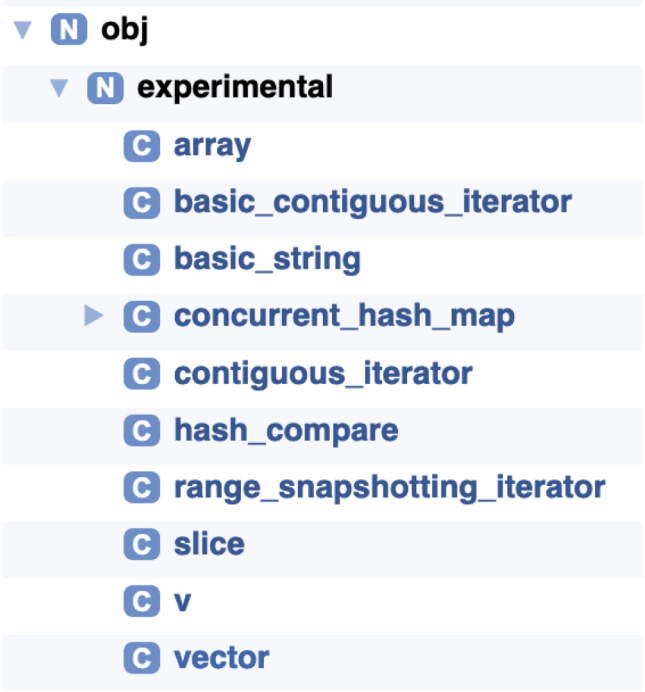
- ... and build a static (immutable) index on the first column:

```
using Value = double;
using Position = std::size_t;
using InvertedIndex = std::map<Value, Position>;

for (size_t position = 0; position < table.size(); ++position) {
  index->emplace(std::get<0>(table[position]), position);
}
```

Code Example

- So far so good. But what if we want to store that index on NVM?
- For an immutable chunk, the index is created but never updated



- Surely, libpmemobj++ can help?
- No tree-based map in libpmemobj++, but in libpmemobj:

```
* four implementations of tree maps:
** ctree - Crit-Bit using tx API of libpmemobj
** btree - B-tree using tx API of libpmemobj
** rtree - Radix-tree using tx API of libpmemobj
** rbtree - red-black tree using tx API of libpmemobj
```

First attempt with pmemobj

```
mapc = map_ctx_init(&rbtree_map_ops, pop);
if (!mapc) { /* [...] */ }
TX_BEGIN(pop) {
  map_create(mapc, &D_RW(root)->map, NULL);
  for (size_t position = 0; position < table.size(); ++position) {
    map_insert(mapc, D_RW(root)->map, std::get<0>(table[position]), new_store_item(position).oid);
  }
} TX_ONABORT {
  /* [...] */
} TX_END
```

Observations:

- We might be able to write a C++ wrapper for this
- However, we do not have a nice std::map anymore that we can pass around
- The rbtree implementation is limited to uint64_t keys – what if we want strings?
- This gives us more consistency guarantees that we care for

The path to STL containers on NVM

- Can we simply allocate our std::map on NVM?

```
template <class T>
class nvm_allocator {
  nvm_allocator() {
    // create or load pool
  }


  [[nodiscard]] T* allocate(std::size_t n) {
    return &pmem::obj::make_persistent
      <char[]>(n * sizeof(T))[0];
  }
}


using InvertedIndex = std::map</*[...]*/,
  nvm_allocator</*[...]*/>>;
```

Issues:

- Data in map is not flushed
- Early crashes lead to persistent leaks
- The nvm_allocator is all over the place
- Remapping invalidates pointers

The path to STL containers on NVM

- Let‘s track the allocations and use them to flush:

```
nvm_allocator() {
  root.inflight_allocations =
    pmem::obj::make_persistent<pmem::obj::
      experimental::vector</*[...]*/>>();
}
[[nodiscard]] T* allocate(std::size_t n) {
  T* pointer;
  pmem::obj::transaction::run(pool, [&] {
     pointer = static_cast<T*>(
       resource()->allocate(n * sizeof(T)));
     root.inflight_allocations->emplace_back(pointer, n);
  });
  return pointer;
}
void persist() {
  for (auto& [p, n] : *root.inflight_allocations) {
     pmemobj_persist(pool.handle(), p, n);
  }
  root.inflight_allocations->clear();
}
```

Issues:

- **Data in map is not flushed**
- Early crashes lead to persistent leaks
- The nvm_allocator is all over the place
- Remapping invalidates pointers

The path to STL containers on NVM

- If instead of T*, we store PMEMoids in the list, we can also use it to free incomplete data:

```
// When reopening the pool:
for (const auto& pmemoid : *root().inflight_allocations) {
  auto ret = pmemobj_tx_free(pmemoid)
  Assert(!ret, "free failed");
}
```

Issues:

- ~~Data in map is not flushed~~
- **Early crashes lead to persistent leaks**
- The nvm_allocator is all over the place
- Remapping invalidates pointers

The path to STL containers on NVM

- Currently, the nvm_allocator is part of the type definition:

```
using InvertedIndex = std::map</*[...]*/,
  nvm_allocator</*[...]*/>;
```

- We do not want to drag it through the code base
- Also, we want to be able to store some indexes on DRAM and some on NVM while using the same code

Issues:

- ~~Data in map is not flushed~~
- ~~Early crashes lead to persistent leaks~~
- **The nvm_allocator is all over the place**
- Remapping invalidates pointers

Polymorphic Memory Resources

- PMR to the rescue

```
using InvertedIndex = std::map</*[...]*/, std::pmr::polymorphic_allocator</*[...]*/>;
```

- Instead of making the nvm_allocator a type parameter, we pass in an nvm_memory_resource that supports methods such as `allocate` and `deallocate`
- For the same type, we can pass nvm_memory_resources or default_memory_resources into the constructor of the object
- The nvm_memory_resource holds the information about the pools
- In Hyrise, we use PMR for NUMA-aware allocations anyway, so there is no change needed

Small kink here: The nvm_memory_resource would have to be persisted as well so that it can be used after recovery. As this would lead to the data structures chasing their own tail, we store the information needed to recreate an equivalent resource.

Polymorphic Memory Resources

```
using InvertedIndex = std::map</*[...]*/,
  polymorphic_allocator</*[...]*/>>;

auto allocator = polymorphic_allocator</*[...]*/>{
  &nvm_memory_resource::get()};

auto index = InvertedIndex {allocator};
```

- For simplicity, we limit the example to a single memory resource (i.e., a single pool)

Issues:

- ~~Data in map is not flushed~~
- ~~Early crashes lead to persistent leaks~~
- **The nvm_allocator is all over the place**
- Remapping invalidates pointers

Polymorphic Memory Resources

- One more thing: Because PMRs propagate into PMR-aware child containers, we get support for nested STL containers for free

```
using Value = std::basic_string<char, /*[...]*/,
  polymorphic_allocator<char>>;

using InvertedIndex = std::map<Value, /*[...]*/,
  polymorphic_allocator</*[...]*/>>;

auto allocator = polymorphic_allocator</*[...]*/>{
  &nvm_memory_resource::get()};

auto index = InvertedIndex {allocator};
```

Issues:

- ~~Data in map is not flushed~~
- ~~Early crashes lead to persistent leaks~~
- **The nvm_allocator is all over the place**
- Remapping invalidates pointers

Representation-Aware Containers

- Currently, all pointers become invalid after remapping the pool somewhere else
- Easy to verify with `-pie`
- Internally, the std::map uses T* to reference other nodes in the tree
- We will never get T* to be NVM-aware

Issues:

- ~~Data in map is not flushed~~
- ~~Early crashes lead to persistent leaks~~
- ~~The nvm_allocator is all over the place~~
- **Remapping invalidates pointers**

Representation-Aware Containers

- Instead of native pointers, the containers should use "fancy pointers“ that can deal with different positions after remapping
- O‘Dwyer and Steagall [1] call this requirement *representation awareness*
- Previous uses are in IPC, where two processes might share the same memory, mapped at different locations
  - A tree written by one process should be readable by the other
  - *The benefit of boost::interprocess::offset_ptr is that if a memory region is mapped into the address space of two different processes, then every offset_ptr residing in the mapped region will identify the same object no matter which process is asking — as long as the identified object also resides in the mapped region.*

[1] https://quuxplusone.github.io/draft/fancy-pointers.html

Representation-Aware Containers

- By simply adding pointer traits to our allocator, we make sure that the reference to a tree‘s child nodes is stored as an offset to the position of the pointer:

```
template <class T>
class polymorphic_offset_allocator {
 public:
  typedef T value_type;
  typedef boost::interprocess::offset_ptr<T> pointer;
  typedef const boost::interprocess::offset_ptr<T>
    const_pointer;
  // [...]
}
```

- The index is now readable after remapping

Issues:

- ~~Data in map is not flushed~~
- ~~Early crashes lead to persistent leaks~~
- ~~The nvm_allocator is all over the place~~
- **Remapping invalidates pointers**

Representation-Aware Containers

- Why don‘t we use persistent_ptr?
- Remember that the polymorphic allocator is used for both, objects stored on DRAM and objects stored on NVM
- We cannot get a persistent_ptr for a DRAM object

Representation-Aware Containers

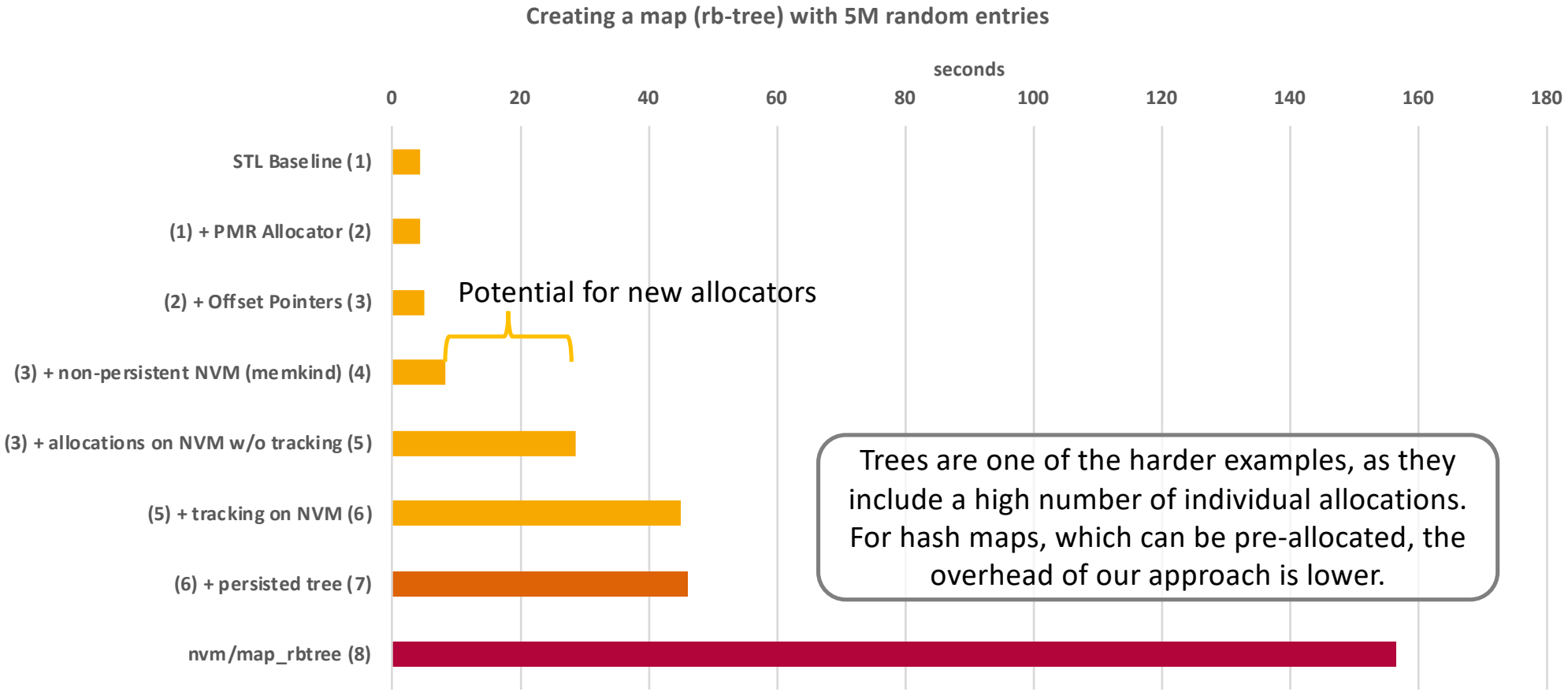
- Limitation: This currently only works in libc++
- In libstdc++, "node-based containers don't use allocator's pointer type internally“ [1]
  - Supporting representation-aware containers would be a breaking ABI change
  - Issue is unchanged since 2016

It's a bug. It is hard to fix without breaking the ABI, and not breaking the ABI is higher priority than supporting fancy pointers. I hoped to get time to finish the work for GCC 8, and for GCC 9, but now maybe it will happen next year for GCC 10.

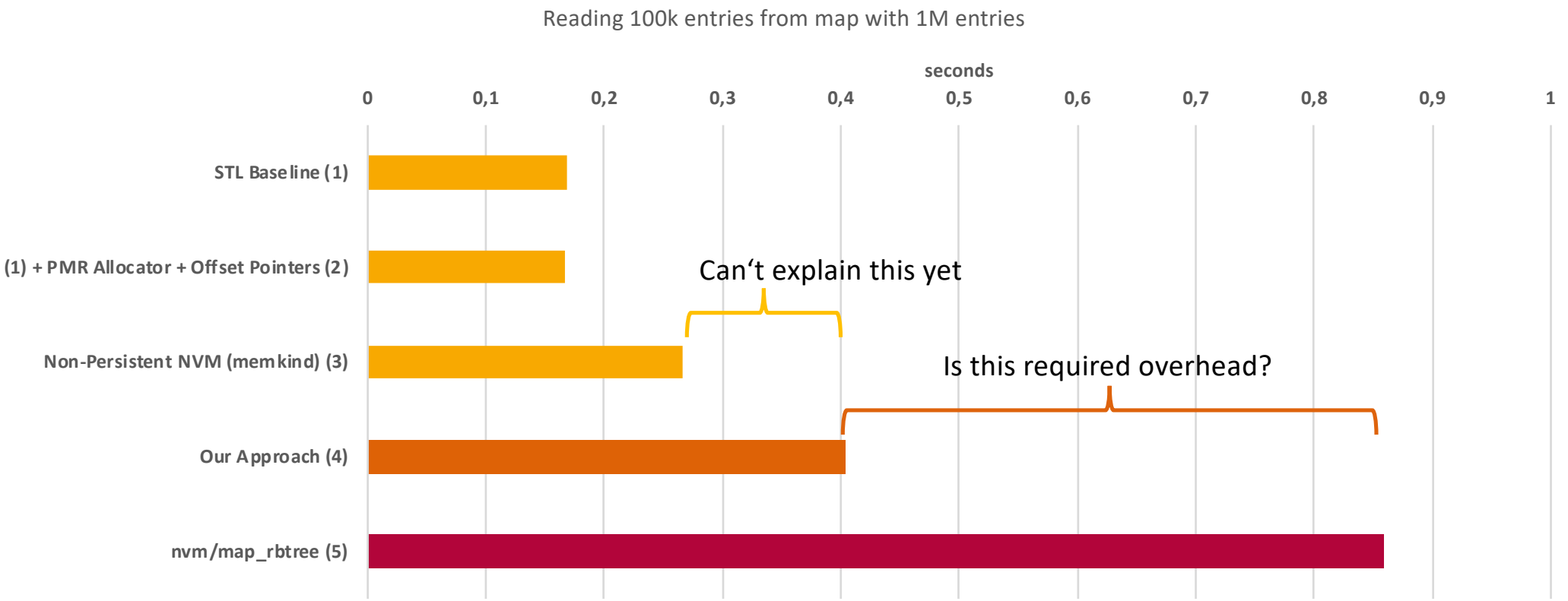
Of course if somebody wants it badly enough they could fix the bug themselves, or contract somebody to do it. GCC is free software after all. I'd be happy to share my work-in-progress patches if somebody else wanted to fix it instead of waiting for me to do the other 100 things on my TODO list first.

[1] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=57272
Screenshot from: https://www.reddit.com/r/cpp/comments/b1fyc9/does_anyone_know_when_libstdc_will_stop_using_raw/

Performance Analysis: Creation



Performance Analysis: Read Access



Topics NOT Covered

Migrations

- Multiple pools
- We can only migrate entire pools – how to balance flexibility and fragmentation?

Integration into Hyrise, our research database

- How to decide which part of the data should be migrated from DRAM to NVM and back?
- How to selectively skip recovery for chunks that are already stored on NVM?

Future Work

- We spend a lot of time in tracking allocations
  - Using a simpler allocation algorithm could make this cheaper
  - Easiest example: a monotonic_buffer_resource would mean that we don’t have to track anything anymore
  - This also makes migration easier
  - How to balance anti-fragmentation and performance?
- Make this usable
  - Right now, this is a prototype to understand feasibility and performance implications
  - Clean up code, verify implementation

Addenda

Based on the discussions during and after the talk, this slide was added to the published version

- If polymorphic objects are stored, the vtable might move due to ASLR or changes in the code. This will cause the vptr to point to an invalid address.
  - This is the same with interprocess mapping (offset_ptrs original use case)
  - Most NVM-aware data structures likely suffer from a similar problem
  - I am not aware of a solution to this other than not to store polymorphic objects
  - Solving this would be a great step towards persistency as a first-class citizen

Summary / Take Aways

- You do not need new data structures to benefit from NVM if you write data once
- Sometimes, using NVM-aware implementations might even hurt your performance

- We showed how you can adapt STL containers to be stored on and recovered from NVM
- The performance is dominated by the cost of persistent NVM allocations

- Currently, this is (a) libc++ only, and (b) illegal
- If you know anyone from libstdc++, please pitch representation-aware containers
- If you know someone in the committee, please bug them about std::bless (and don't rat me out)