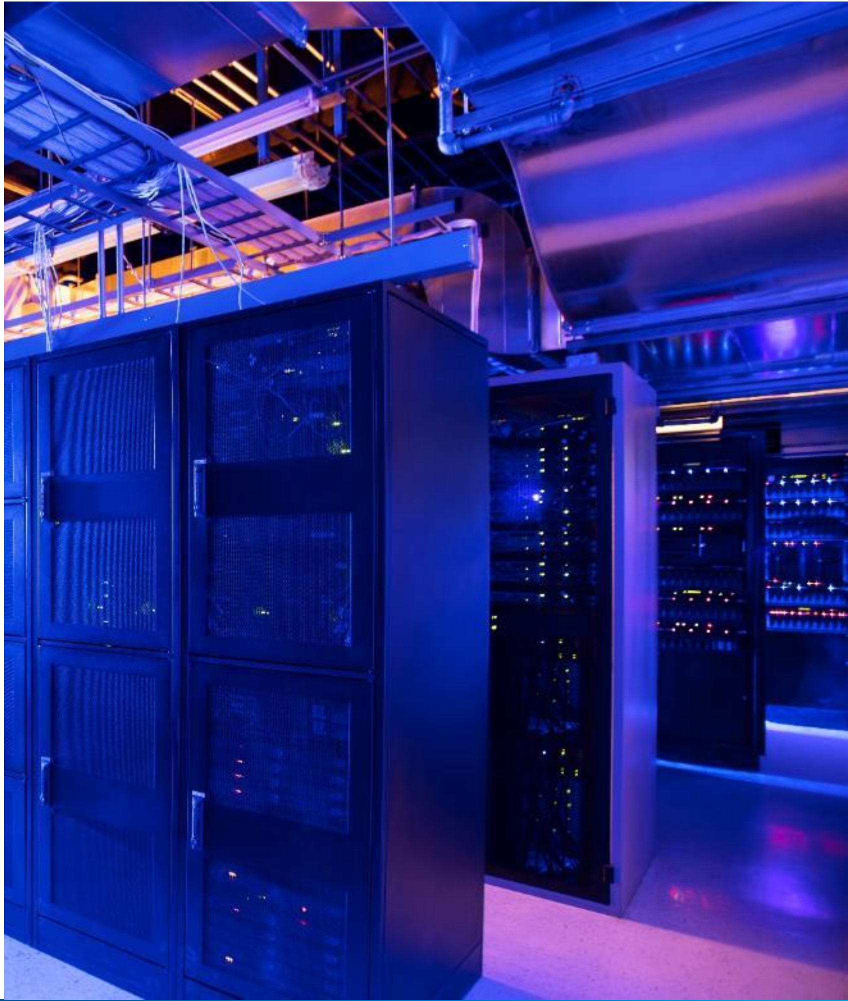
INTRODUCTION TO PMEMKV

Key/Value Store for Persistent Memory

Rob Dickinson -- pmemkv Technical Lead

AGENDA

- Why pmemkv?
- Key/Value API
- Language Bindings
- Storage Engines
- Roadmap



WHY PMEMKV?

Persistent memory programming is difficult

There are many languages in the cloud

Key/value datastores are increasingly popular

→ Large addressable market of cloud developers for an easy KV store for persistent memory

GOALS FOR PMEMKV

Technical:

Local key/value store (no networking)

Idiomatic language bindings

Simple, familiar, bulletproof API

Easily extended with new engines

Optimized for persistent memory (limit copying to/from DRAM)

Flexible configuration, not limited to a single storage algorithm

Community:

Open source, developed in the open

Friendly licensing

Outside contributions are welcome

Collaboration alternative for folks frustrated with other OS projects

Intel provides stewardship, validation on real hardware, and code reviews

Standard/comparable benchmarks

SIMPLE & PORTABLE KEY/VALUE API

Engine Lifecycle:

Start (engine_name, json_config) → engine

Stop (engine)

Engine Operations:

engine.Put (key, value)

engine.Get (key, value_or_callback)

engine.Exists (key) → true/false

engine.Remove (key)

Iteration Operations: (optional)

engine.Count ( ) → count

engine.All (key_callback)

engine.Each (key_and_value_callback)

plus range iteration versions of these!

PMEMKV SAMPLE PROGRAM -- C++

```
int main()
{
    KVEngine* kv = KVEngine::Start("vsmap", "{\"path\":\"/dev/shm/\"}");

    KVStatus s = kv->Put("key1", "value1");
    assert(s == OK && kv->Count() == 1);

    string value;
    s = kv->Get("key1", &value);
    assert(s == OK && value == "value1");

    kv->All([](const string& k) { LOG("  visited: " << k);  });

    s = kv->Remove("key1");
    assert(s == OK && !kv->Exists("key1"));

    delete kv;
    return 0;
}
```

IDIOMATIC LANGUAGE BINDINGS

Implementations:

C++ (core)

C (extern "C" on C++ core)

Java (v8 or higher)

NodeJS (v6.10 or higher)

Ruby (v2.2 or higher)

Python (coming soon!)

Goals:

No modifications to languages or runtime environments

Designed for multi-language use

Familiar object model

Familiar error handling

Language-specific types

Heap offloading (where applicable)

Consistent unit tests

PMEMKV SAMPLE PROGRAM -- JAVA

```
public static void main(String[] args)
{
    KVEngine kv = new KVEngine("vsmap", "{\"path\":\"/dev/shm/\"}");

    kv.put("key1", "value1");
    assert kv.count() == 1;

    assert kv.get("key1").equals("value1");

    kv.all((String k) -> System.out.println("  visited: " + k));

    kv.remove("key1");
    assert !kv.exists("key1");

    kv.stop();
}
```

NOTE:
Binding based on JNI
Throw exception on error
Supports String, byte[] and ByteBuffer
Overloaded iterator methods/types

PMEMKV SAMPLE PROGRAM -- NODEJS

```
const kv = new KVEngine('vsmap', '{"path":"/dev/shm/"}');

kv.put('key1', 'value1');
assert(kv.count === 1);

assert(kv.get('key1') === 'value1');

kv.all((k) => console.log(`  visited: ${k}`));

kv.remove('key1');
assert(!kv.exists('key1'));

kv.stop();
```

NOTE:
Maintained by Intel JSTC
Binding based on NAPI
Throw exception on error
Buffer support coming soon!

PMEMKV SAMPLE PROGRAM -- RUBY

```
kv = KVEngine.new('vsmap', '{"path":"/dev/shm/"}')

kv.put('key1', 'value1')
assert kv.count == 1

assert kv.get('key1').eql?('value1')

kv.all_strings {|k| puts "  visited: #{k}"}

kv.remove('key1')
assert !kv.exists('key1')

kv.stop
```

NOTE:
Binding based on FFI
Throw exception on error
Supports string & byte array
Iterator methods with type names

SOURCES OF LATENCY WITHIN HIGH-LEVEL BINDINGS

- # of round trips between high-level language & native code

- pmemkv language bindings (one round trip per operation)
  - Copy key/value data between persistent memory and high-level object
  - Create high-level object (string, byte[], reference type, callback/closure)
  - Depending on the type of high-level object...
    - Translate bytes to UTF-8
    - String interning, reference counting or GC

- pmemkv core (native code)
  - Searching indexes in DRAM
  - Updating indexes in DRAM
  - Managing transactions
  - Allocating persistent memory

- Persistent Memory (read & write latencies)

RELATIVE PERFORMANCE OF BINDINGS

	Engine: tree3			
	C++ (string)	Java (byte[])	NodeJS (string)	Ruby (string)
put_16 (ns)	1881	2057	2370	2431
get_16 (ns)	140	233	503	1304
exists_16 (ns)	101	160	339	363
all_16 (ns)	39.5	202	361	307
each_16 (ns)	47.2	273	322	462

NOTE: Single-threaded benchmarks on emulated persistent memory, 16 byte keys & values,
C++ all/each pass pointers only, C++ & Java benchmarks use pre-generated keys,
Ruby & NodeJS bindings use UTF-8 strings built during the benchmark

RELATIVE PERFORMANCE OF BINDINGS

	Engine: tree3			
	C++ (string)	Java (byte[])	NodeJS (string)	Ruby (string)
put_16 (ns)	1881	2057	2370	2431
get_16 (ns)	140	233	503	1304
exists_16 (ns)	101	160	339	363
all_16 (ns)	39.5	202	361	307
each_16 (ns)	47.2	273	322	462

NOTE: Single-threaded benchmarks on emulated persistent memory, 16 byte keys & values, C++ all/each pass pointers only, C++ & Java benchmarks use pre-generated keys, Ruby & NodeJS bindings use UTF-8 strings built during the benchmark

STORAGE ENGINES

Implementations:

cmap (PMDK concurrent map)

vsmap (memkind)

vcmap (memkind)

tree3 / stree (experimental)

caching (experimental)

Goals:

Applications can use multiple engines

Engines are optimized for different workloads & capabilities (threading, persistence, iterating, sorting)

All engines work with all utilities and language bindings

Engines can use other engines

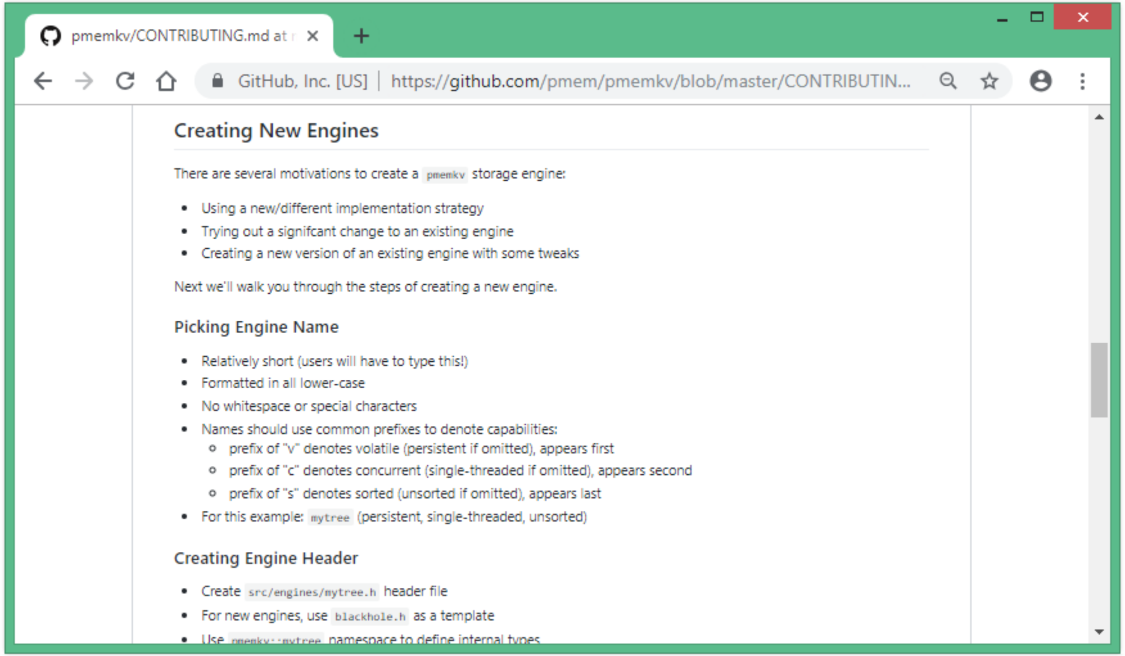
Engines from the community

AVAILABLE ENGINES

Engine Name	Description	Experimental?	Concurrent?	Sorted?
blackhole	Accepts everything, returns nothing	No	Yes	No
cmap	Concurrent hash map	No	Yes	No
vsmap	Volatile sorted hash map	No	No	Yes
vcmap	Volatile concurrent hash map	No	Yes	No
tree3	Persistent B+ tree	No	No	No
stree	Sorted persistent B+ tree	Yes	No	Yes
caching	Caching for remote Memcached or Redis server	Yes	Yes	-

CONTRIBUTING ENGINES IS EASY

https://github.com/pmem/pmemkv/blob/master/CONTRIBUTING.md#engines



Creating New Engines

There are several motivations to create a `pmemkv` storage engine:

- Using a new/different implementation strategy
- Trying out a signifcant change to an existing engine
- Creating a new version of an existing engine with some tweaks

Next we'll walk you through the steps of creating a new engine.

Picking Engine Name

- Relatively short (users will have to type this!)
- Formatted in all lower-case
- No whitespace or special characters
- Names should use common prefixes to denote capabilities:
  - prefix of "v" denotes volatile (persistent if omitted), appears first
  - prefix of "c" denotes concurrent (single-threaded if omitted), appears second
  - prefix of "s" denotes sorted (unsorted if omitted), appears last
- For this example: `mytree` (persistent, single-threaded, unsorted)

Creating Engine Header

- Create `src/engines/mytree.h` header file
- For new engines, use `blackhole.h` as a template
- Use `pmemkv::mytree` namespace to define internal types

PMEMKV ROADMAP

1.0 Release:

C/C++, Java, JavaScript, Ruby bindings

cmap, vmap, vcmap engines

Automated tests & CI environment

pmemkv_bench (port of db_bench)

Q2 2019 (estimated)

1.1 Release:

Python bindings

csmap, caching engines

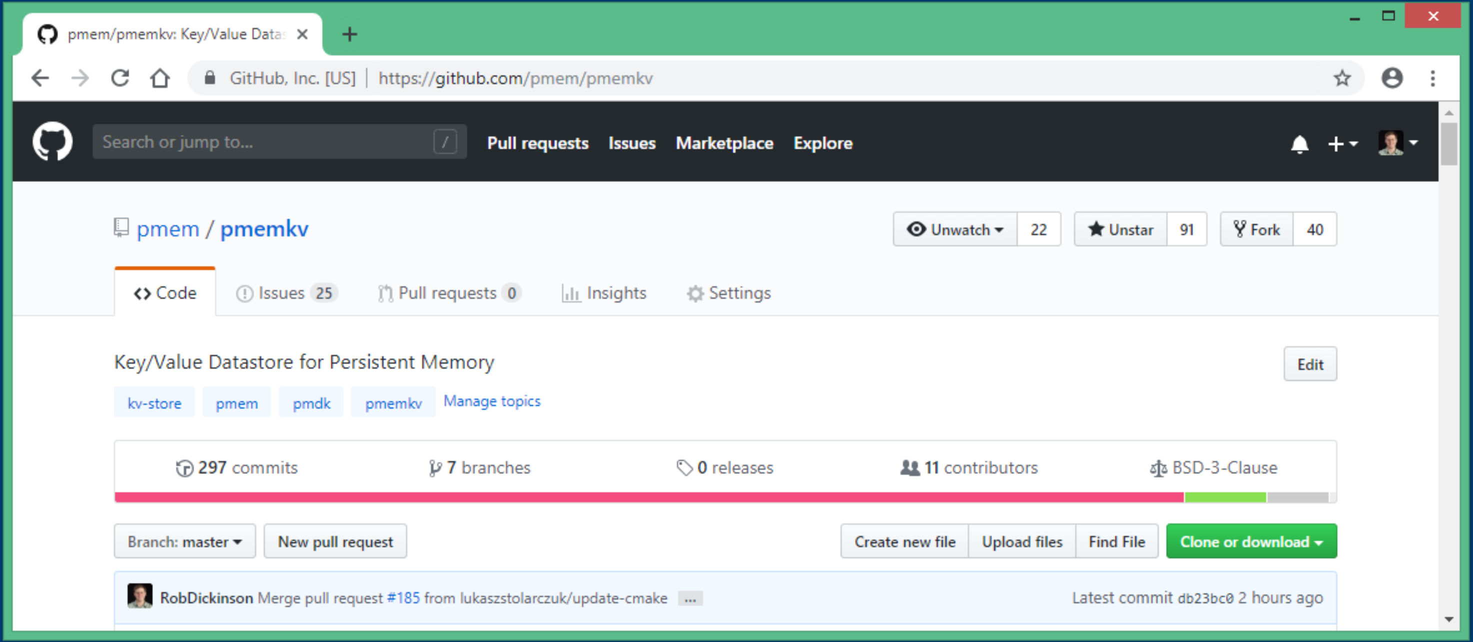
Third-party engines?

Pool partitioning

Q4 2019 (estimated)

CALL TO ACTION

Star/watch us on GitHub! https://github.com/pmem/pmemkv



intel®