CUSTOMIZING DATA SERVICES FOR FUN AND PROFIT

COMPONENTS: BAKE AND FRIENDS

PHILIP CARNS
Argonne National Laboratory

February 6, 2020

SESSION OBJECTIVE

Putting the “storage” in storage service!

- The tutorial thus far has focused mostly on RPC infrastructure
- In this session we are going to switch gears and talk about a specific microservice that was built using that RPC infrastructure
- A hands-on example can be found in the mochi-boot-camp/ecp-am-2020/sessions/hands-on/bake directory.

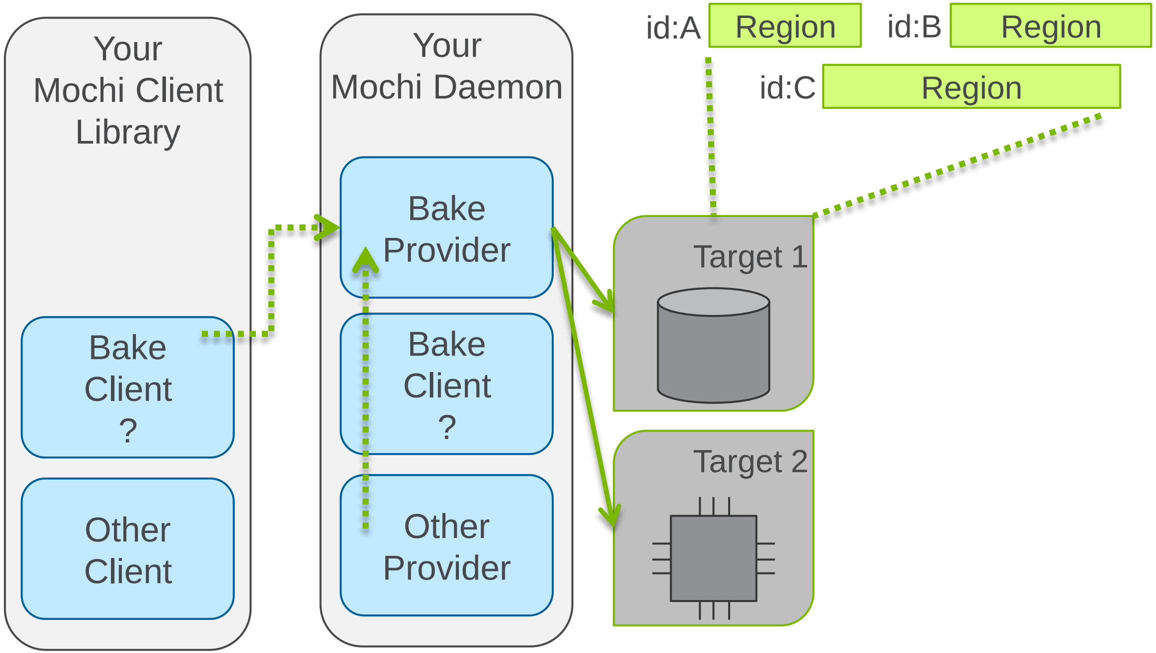
BAKE

What is it, and where does it fit in the Mochi ecosystem?

- **Bake** is a composable service that **provides access to raw data regions that can be accessed locally, by other processes, or over the network**.
- Bake can operate atop persistent memory or local file systems
  - We'll talk more about storage backends later in the presentation
- The API and semantics are a little unconventional because Bake is just a building block: it was never meant to be used directly as a standalone service.
- Think of it as the raw storage component of a higher level data service

BAKE AS A BUILDING BLOCK

- Bake providers can be "embedded" in your service.
- Make Bake client calls from within your service daemon or from other clients.
- Targets can be thought of as storage devices (or storage containers).
- Regions are uniquely referenceable objects / blobs / segments within a target. They can be any size.



BAKE SEMANTICS

Somewhat like an object store… but different

- Conceptual assumptions:
  - Some other component (sdskv or other) will index and name regions
    - Implication: Bake itself does not provide a name space
  - Some other component (ch-placement or other) will do placement/striping
    - Implication: Bake itself does not shard data
- API limitations:
  - Caller cannot dictate IDs for regions (Bake assigns them at creation time)
  - Caller cannot change size of regions once created (no appending)
  - Caller must explicitly “persist” regions when desired
- API non-limitations:
  - Concurrent and partial readers and writers are welcome (and encouraged)!

BAKE CLIENT CODE EXAMPLE

```
#include "bake-client.h"

bake_client_t bcl;
bake_provider_handle_t bph;
bake_target_id bti;
bake_region_id_t rid;

bake_client_init(mid, &bcl);

margo_addr_lookup(mid, svr_addr_str, &svr_addr);
bake_provider_handle_create(bcl, svr_addr, mplex_id, &bph);

bake_probe(bph, target_number, bti, &num_targets);

bake_create(bph, bti, 1024, &rid);
bake_write(bph, rid, 0, buffer, 1024);
bake_persist(bph, rid, 0, 1024);

bake_provider_handle_release(bph);
margo_addr_free(mid, svr_addr);

bake_client_finalize(bcl);
```

- **bake_client**: local client instance, can be used to access many remote providers
- **provider_handle**: references one remote provider
- **target**: references a single target in a provider
- **region**: references a blob within a target

BAKE CLIENT CODE EXAMPLE

```
#include "bake-client.h"

bake_client_t bcl;
bake_provider_handle_t bph;
bake_target_id bti;
bake_region_id_t rid;

bake_client_init(mid, &bcl);

margo_addr_lookup(mid, svr_addr_str, &svr_addr);
bake_provider_handle_create(bcl, svr_addr, mplex_id, &bph);

bake_probe(bph, target_number, bti, &num_targets);

bake_create(bph, bti, 1024, &rid);
bake_write(bph, rid, 0, buffer, 1024);
bake_persist(bph, rid, 0, 1024);

bake_provider_handle_release(bph);
margo_addr_free(mid, svr_addr);

bake_client_finalize(bcl);
```

Look up a Bake provider by it’s Mercury address (svr_addr_string).

Then create a “provider handle”, which is a local reference to that provider.

Probe to get a reference (target id) to a specific target on the provider. There could be multiple.

BAKE CLIENT CODE EXAMPLE

```
#include "bake-client.h"

bake_client_t bcl;
bake_provider_handle_t bph;
bake_target_id bti;
bake_region_id_t rid;

bake_client_init(mid, &bcl);

margo_addr_lookup(mid, svr_addr_str, &svr_addr);
bake_provider_handle_create(bcl, svr_addr, mplex_id, &bph);

bake_probe(bph, target_number, bti, &num_targets);

bake_create(bph, bti, 1024, &rid);
bake_write(bph, rid, 0, buffer, 1024);
bake_persist(bph, rid, 0, 1024);

bake_provider_handle_release(bph);
margo_addr_free(mid, svr_addr);

bake_client_finalize(bcl);
```

Create, write, read, persist as many regions as you would like to within the target. The region ids (and also the target ids) are durable and can be stored in another component for future reference.

This example creates, writes, and persists a single region.

BAKE CLIENT CODE EXAMPLE

```
#include "bake-client.h"

bake_client_t bcl;
bake_provider_handle_t bph;
bake_target_id bti;
bake_region_id_t rid;

bake_client_init(mid, &bcl);

margo_addr_lookup(mid, svr_addr_str, &svr_addr);
bake_provider_handle_create(bcl, svr_addr, mplex_id, &bph);

bake_probe(bph, target_number, bti, &num_targets);

bake_create(bph, bti, 1024, &rid);
bake_write(bph, rid, 0, buffer, 1024);
bake_persist(bph, rid, 0, 1024);

bake_provider_handle_release(bph);
margo_addr_free(mid, svr_addr);

bake_client_finalize(bcl);
```

Release reference to provider and address when done.

Note that there is no need to release target or region ids; think of those as opaque numeric identifiers that are passed by value.

BACK-END STORAGE OPTIONS FOR BAKE

HOW BAKE MAKES BYTES PERSISTENT

Current status as of February 2020

- Bake uses PMDK to store regions in persistent memory
  - This is the default back-end for BAKE
  - PMDK is developed and maintained by Intel
  - Best suited for byte-addressible memory-like devices (like Optane DIMMs)
  - Stable on x86 and ARM architectures
  - Preliminary (PMDK 1.9) support for POWER architectures (like Summit)
- PMDK can be used without access to persistent memory devices:
  - On tmpfs (aka ramdisk): no persistence
  - On local file systems (like ext4): persistent, but probably not fully performant

PMDK BACK-END SUMMARY

"Persistent Memory Development Kit"

- PMDK is not a Mochi component, but rather a family of external libraries that work well with Mochi and supports emerging storage technology, including:
  - Libpmem: basic memory access, control over persistence (flushing)
  - Libpmemobj: object store
  - Pmemkv: key/value store
- Bake uses libpmemobj
  - Includes portable primitives for flushing, atomicity, and persistent references
  - Enables RDMA directly to storage
  - Low latency (i.e., no need to context switch in RPC handler)
- Even if you don't have persistent memory devices, PMDK is actually a reasonable way to organize data in memory that can be staged in or out.

HOW BAKE MAKES BYTES PERSISTENT

Near-future roadmap

- The most recent release of Bake includes a pluggable back-end that can translate Bake data into different local storage formats, so that Bake is not hard coded to use PMDK or persistent memory
- See (old) proof of concept in spack as bake@dev-file-backend
  - Will be ported forward to new modular architecture
- The file back-end uses ABT-IO for local file system (disk) storage
  - Will work on any unprivileged local file system directory
  - Employs a log-structured direct-io file layout
  - Log entries are buffered to memory for pipelined RDMA access
  - Leverages the ANL-developed ABT-IO library to interact with local files

ABT-IO BACK-END SUMMARY

ABT-IO stands for “Argobots I/O”

- ABT-IO is a Mochi library (it can be used independently from Bake) that provides Argobots-aware wrappers for common POSIX file access functions
  - e.g., abt_io_open(), abt_io_pwrite(), and abt_io_close()
- *Argobots-aware* means that when those functions block, they yield execution so that other Argobots threads can make progress in the mean time
  - The actual I/O servicing is delegated to a thread pool
  - Callers can safely invoke blocking I/O operations directly within a Mochi RPC handler without interfering with other RPC handlers
- In the Bake case, ABT-IO is used to access a log-structured back-end data file
- ABT-IO is also a straightforward example of how to wrap any similar sort of blocking work for use in highly-concurrent Mochi environments

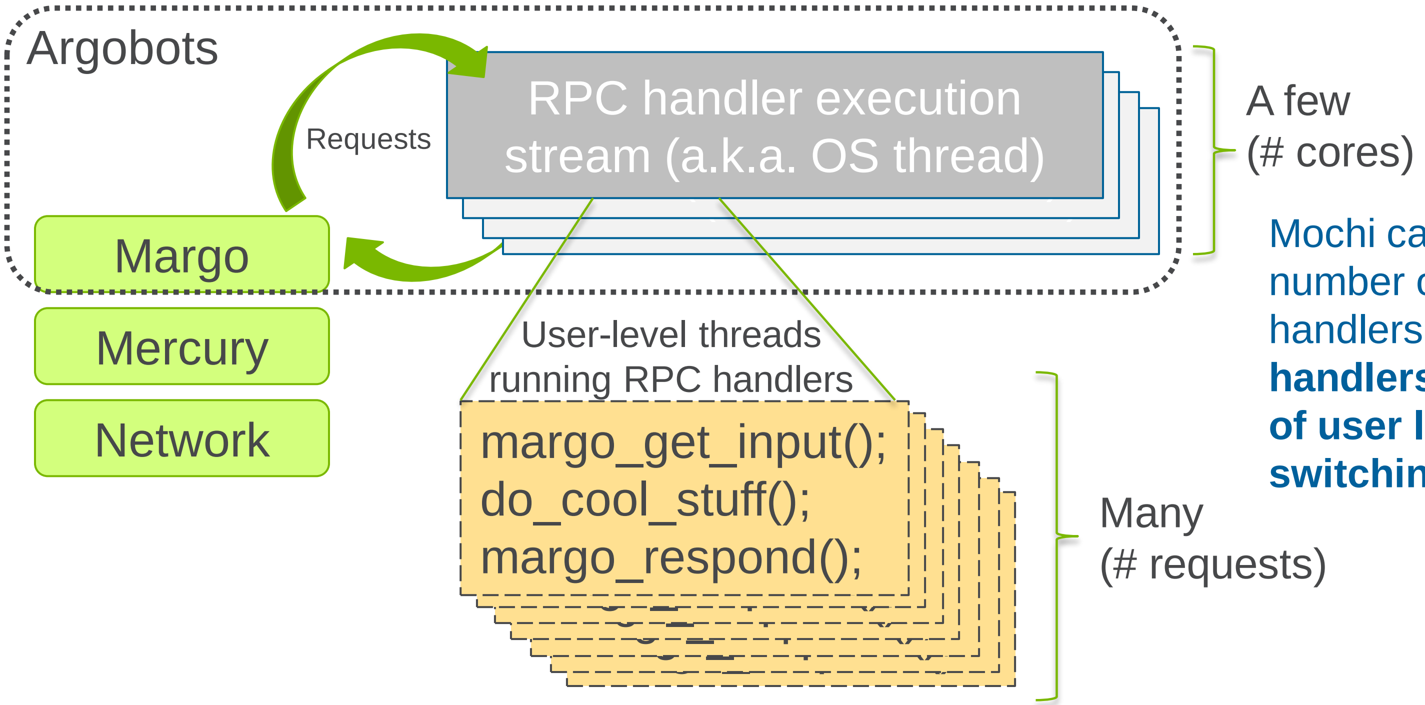
THANK YOU!

SUPPLEMENTAL MATERIAL:

ABT-IO DETAILS

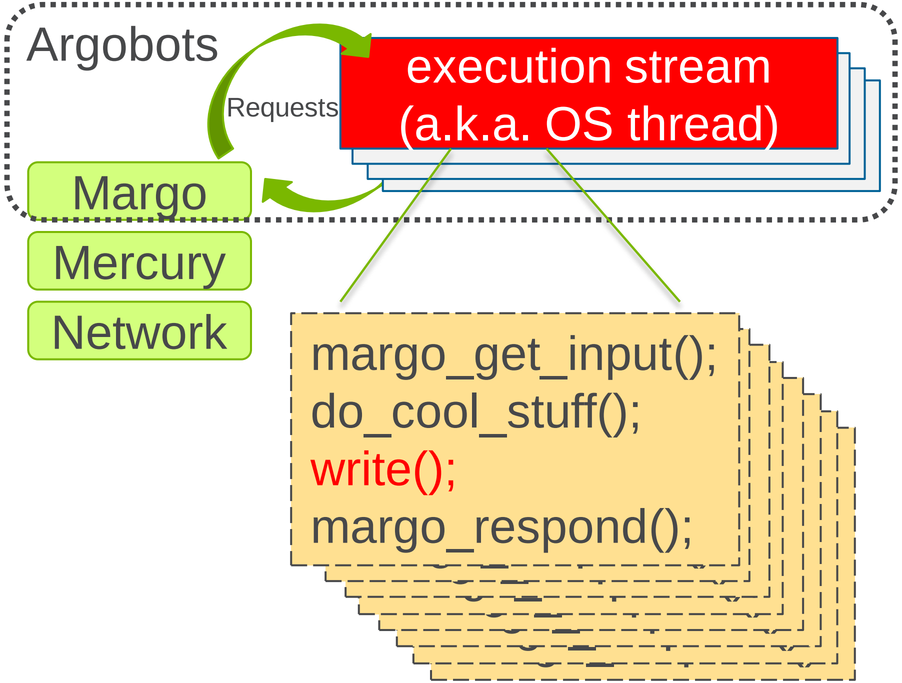
REVIEWING GENERAL MOCHI CONCEPTS

What does a Mochi server daemon look like?



Mochi can run a *very* large number of concurrent request handlers … **assuming those handlers can take advantage of user level context switching in Argobots.**

SUPPOSE YOU WANT TO ACCESS FILES IN A REQUEST HANDLER



- What happens if you call write() in your RPC handler?
- write() is a blocking OS system call and is *not* Argobots aware.
- It would therefore block the execution stream (a.k.a. OS thread) from doing anything else while it waits.
- This can clog up Mochi progress:
- Other RPC handlers (user-level threads), and possibly the network progress engine itself, **may not be able make concurrent progress**.

HOW TO MAKE BLOCKING CALLS IN MOCHI?

We have options:

A. Deploy more RPC handler threads?

- Simple! Just change one argument to margo_init(… <rpc_thread_count>)
- Some drawbacks:
  - Now one configuration parameter is simultaneously altering request concurrency, I/O concurrency, and core usage on the node.
  - Can have unpredictable results.

B. Delegate blocking calls to use set-aside resources?

- Enable the caller (an RPC handler) to suspend and task switch, in user space, while another pool of execution streams does the actual work.
  - Finer grained control: distinction between different types of concurrency
  - More RPC handlers can be kept in flight.

Abt-io is a Mochi component that implements option B.

ABT-IO FROM THE COMPONENT USER PERSPECTIVE

At startup:

```
abt_io_init(<thread_count>);
```

```
margo_get_input();
do_cool_stuff();
write();
margo_respond();
```

```
margo_get_input();
do_cool_stuff();
abt_io_pwrite();
margo_respond();
```

Abt-io calls replace standard POSIX system calls with Argobots-aware equivalents that will not clog up core Mochi progress.

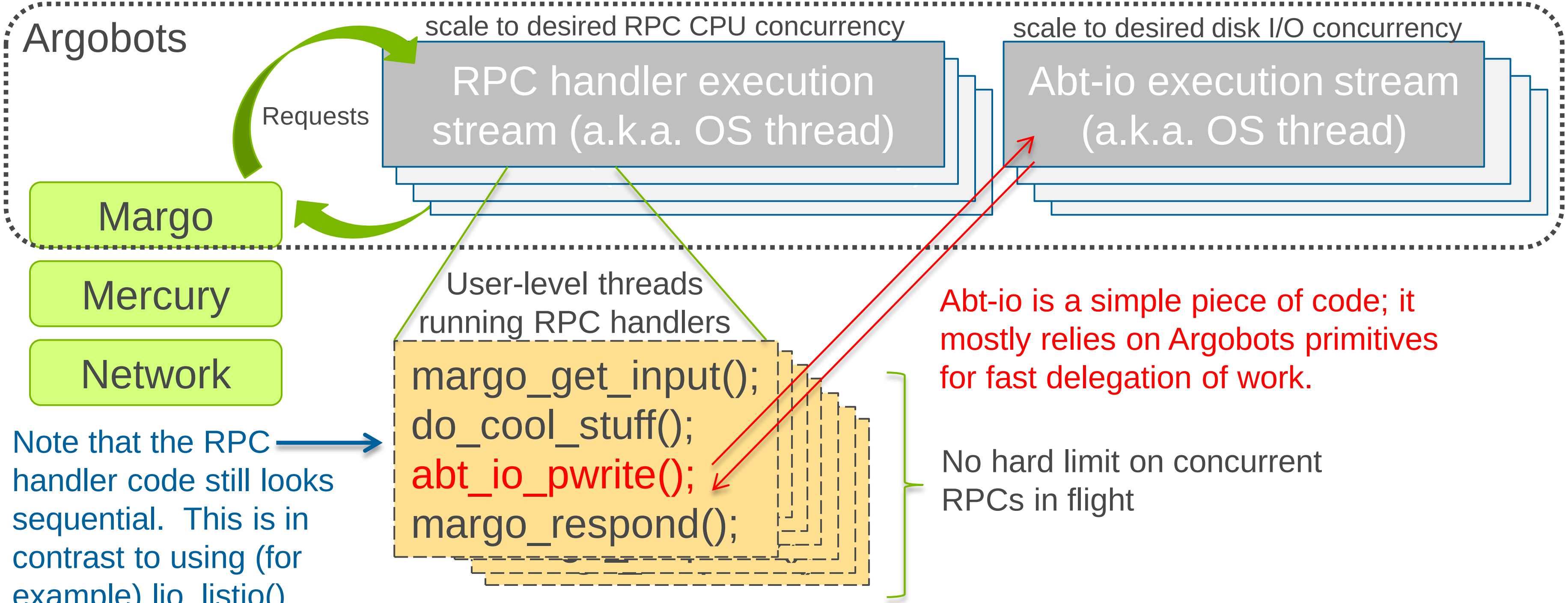
At shutdown:

```
abt_io_finalize();
```

- Some minor API differences:
  - abt_io_ prefix
  - return –errno (instead of using global errno variable)
  - You usually want pread/pwrite in place of read/write in concurrent situations

REVISITING A DAEMON WITH ABT-IO

What abt-io does under the hood



Note that the RPC handler code still looks sequential. This is in contrast to using (for example) lio_listio().

Abt-io is a simple piece of code; it mostly relies on Argobots primitives for fast delegation of work.

No hard limit on concurrent RPCs in flight

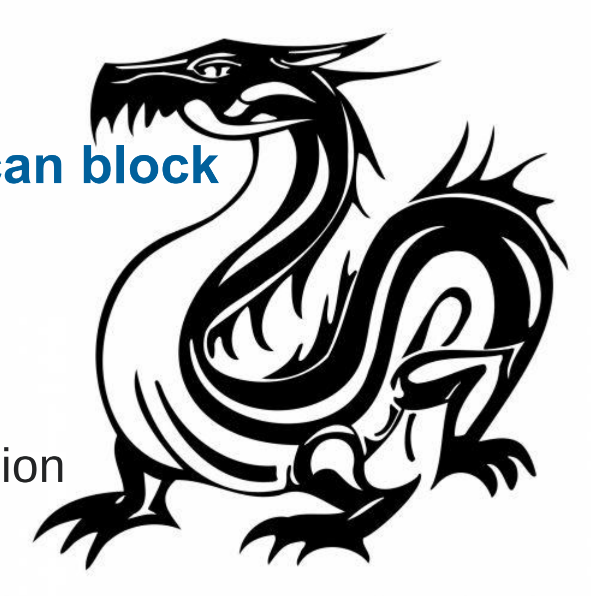
TO BLOCK OR NOT TO BLOCK

What does it mean for a function call to “block” really?

- Technically, both write() and abt_io_pwrite() “block” in that they do not return control to the calling function until they complete.
  - The real distinction is that the latter is Argobots-aware, and uses cooperative user-space scheduling to let others threads proceed while it waits.
- What if you really (from a software engineering point of view) want split nonblocking calls, as in the post/wait model?
  - Abt-io can do that too; see the **_nb()** variant of each function, and the corresponding **abt_io_op_wait().**
  - Uses the same engine and resources as normal abt_io functions
  - Analogous to the nonblocking “i” functions in Margo
- No meaningful difference in performance or concurrency, but can be helpful in some software design patterns.

A CAUTIONARY TALE

File system I/O calls aren't the only things that can block



- What's special about file I/O calls?
  - Nothing really. They are just common examples of blocking calls in data services, so we provide a Mochi component to help.
  - I/O calls don't consume CPU, so we can over-provision threads easily, but that's just a detail.
- The upshot: *any* blocking call that isn't Argobots-aware could potential starve an RPC execution stream.
  - MPI calls are another common example; we'll cover this later.
  - Generally: either provision your threads accordingly (and keep the Margo progress engine itself in its own thread), or if the use case is common enough to warrant it, make a wrapper component like abt-io.

SUPPLEMENTAL MATERIAL:
PMDK DETAILS

SOME NVRAM CONCEPTS

Or: why would you even want a library for this?

- In this context we mean NVRAM as in Intel's "Optane DC Persistent Memory"
  - Device is on the memory bus
  - Byte addressible
  - Persistent
- It *looks* like DRAM, and notionally you can access it the same way:
  - Map an NVRAM device into your virtual address space
  - Use it like you would any other memory
- But this misses a few details:
  - What if you reboot, access the device elsewhere, or just from another process?
    - Normal virtual memory pointers aren't persistent
  - How do you control persistence? It's fast in relative terms, but surely not "free".

ENTER PMDK

"Persistent Memory Development Kit"

- This isn't a Mochi component, but rather an external library that works well with Mochi and supports emerging storage technology.
- Actually a family of libraries with some helpful data structures, a few notable ones are:
  - Libpmem: basic memory access, control over persistence (flushing)
  - Libpmemobj: object store
  - Pmemkv: key/value store
- What if you don't have NVDIMM hardware?
  - You can run PMDK on DRAM (tmpfs) or an mmapped block device (slowly)
  - Possibly useful for DRAM memory regions you would like to asynchronously persist, or to prepare for future NVDIMM hardware

IMPLICATIONS OF PMDK FOR MOCHI

Some usage tips

- All data stored in PMDK is ultimately accessed like normal memory (load/store/memcpy/etc) with normal virtual addresses.
- **Great news #1**: you can perform RDMA operations on these virtual addresses!
  - This means that margo_bulk_create() can be called on a pointer provided by PMDK to register it for remote access with margo_bulk_transfer()
  - Is this the fastest method? Unclear (more on this later), but very convenient.
- **Great news #2**: PMDK operations are far lower latency than disk or SSD access; there is no need to wrap (as in abt-io), just use directly in RPC handler.
  - No system calls
  - No PCI bus or network fabric transit
  - No kernel page cache or block layer
  - No R/M/W of media blocks

A BASIC PMDK (LIBPMEMOBJ) EXAMPLE

Opening the device

```
#include <libpmemobj.h>

PMEMobjpool *pool;
PMEMoid oid1, oid2;
char* buffer;

pool = pmemobj_open(argv[1], NULL);

pmemobj_alloc(pool, &oid1, 1024, 0, NULL, NULL);

buffer = pmemobj_direct(oid1);

sprintf(buffer, "Hello ");

pmemobj_persist(pool, buffer, strlen(buffer)+1);

pmemobj_close(pool);
```

Open the “pool”. Could be a special device file, a file within a DAX-enabled file system, or a normal file. Must be formatted for libpmemobj.

Close the pool. Once it is closed, you can copy or move the whole thing if you would like; OIDs will still work.

A BASIC PMDK (LIBPMEMOBJ) EXAMPLE

Creating an object

```
#include <libpmemobj.h>

PMEMobjpool *pool;
PMEMoid oid1, oid2;
char* buffer;

pool = pmemobj_open(argv[1], NULL);

pmemobj_alloc(pool, &oid1, 1024, 0, NULL, NULL);

buffer = pmemobj_direct(oid1);

sprintf(buffer, "Hello ");

pmemobj_persist(pool, buffer, strlen(buffer)+1);

pmemobj_close(pool);
```

Create an object. You must specify it’s size up front. It can be deleted, but it **cannot** grow. The OID is persistent and will still be valid if you open the pool later.

A BASIC PMDK (LIBPMEMOBJ) EXAMPLE

Creating an object

```
#include <libpmemobj.h>

PMEMobjpool *pool;
PMEMoid oid1, oid2;
char* buffer;

pool = pmemobj_open(argv[1], NULL);

pmemobj_alloc(pool, &oid1, 1024, 0, NULL, NULL);

buffer = pmemobj_direct(oid1);

sprintf(buffer, "Hello ");

pmemobj_persist(pool, buffer, strlen(buffer)+1);

pmemobj_close(pool);
```

Retrieve a “normal” pointer to the data in the object. This is a volatile, virtual memory address.

Do whatever you would normally do with a memory pointer. Cast it, put strings in it, memcpy, margo_bulk_create() etc.

A BASIC PMDK (LIBPMEMOBJ) EXAMPLE

Creating an object

```
#include <libpmemobj.h>

PMEMobjpool *pool;
PMEMoid oid1, oid2;
char* buffer;

pool = pmemobj_open(argv[1], NULL);

pmemobj_alloc(pool, &oid1, 1024, 0, NULL, NULL);

buffer = pmemobj_direct(oid1);

sprintf(buffer, "Hello ");

pmemobj_persist(pool, buffer, strlen(buffer)+1);

pmemobj_close(pool);
```

If you modified the object, you have to "persist" it. This performs a platform/device optimized flush to make sure the memory is really in the NVRAM (and not in, for example, a CPU cache).

OTHER LIBPMEMOBJ FEATURES

Turn to the PMDK documentation if you are interested!

- Typed objects: macros that can present objects as particular C types (and thus benefit from compiler type checking) rather than void*
- Transactions: group sets of operations together to be atomically applied or not
  - NOTE: some operations, like libpemobj_create() are automatically atomic!
- Iterators: to iterate through objects
- Miscellaneous management features:
  - Combine multiple pools into one
  - Mirror pools to remote machines via libfabric
  - Device management features via NDCTL