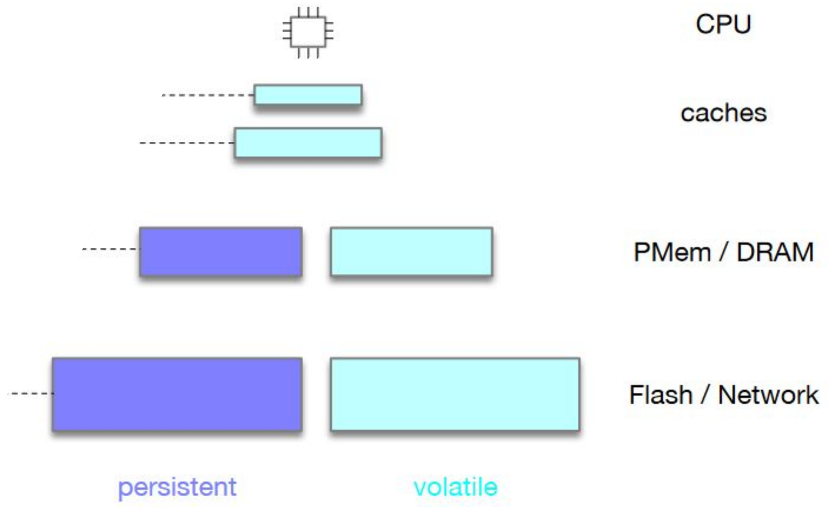
Data Pipes
Declarative Control over Data Movement

Lukas Vogel, Daniel Ritter, Danica Porobic,
Pınar Tözün, Tianzheng Wang, Alberto Lerner

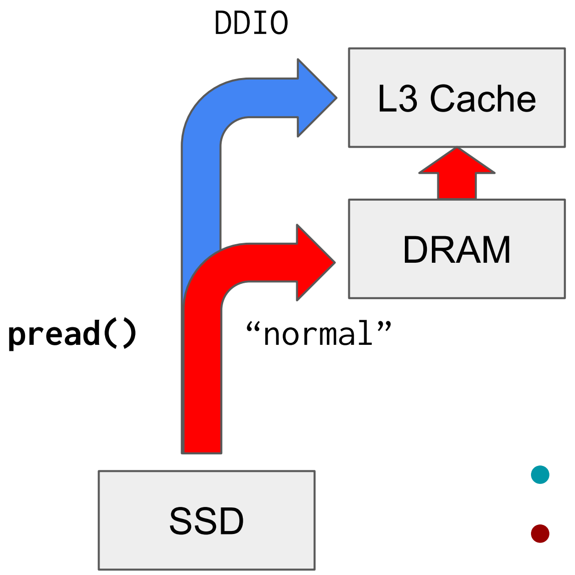
Status Quo: Data movement is a bottleneck

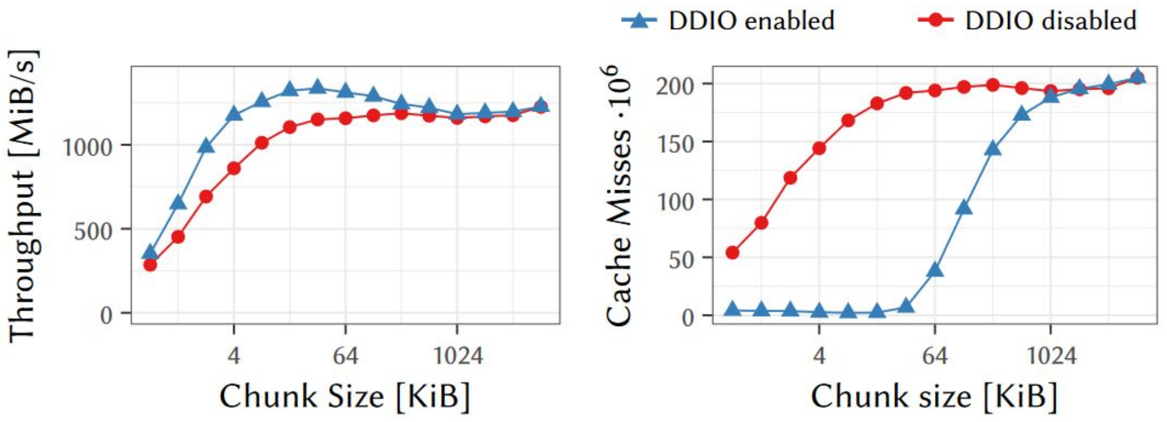


- Wildly different hardware at every level
  - Characteristics (latency, throughput, …)
  - Granularity (Cache line, page, block, …)
  - Access patterns (sequential, random)
- Applications spend a lot of time moving data
  - **Explicitly:** e.g., `pread`
  - **Implicitly:** e.g., CPU caches
- Moving data is complex
  - APIs (`pread`, `mmap`, `io_uring`, SPDK, S3,…)
  - Protocols (NVMe/PCI, HTTP, SATA, DDR(-T))
  - Instructions (`CLFLUSH(OPT)`, `CLWB`, `CLDEMOTE`)
  - **CPU involvement**
- We'd like to utilize "**shortcuts**"
  - Such as DMA (I/OAT), DDIO
  - Exist between nearly any device pair
  - We call them "data movement primitives"

Unfortunately, data movement primitives are difficult to utilize…

Reason 1: Primitives are hard to control





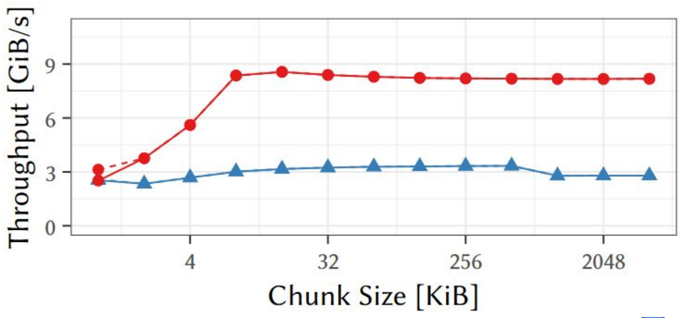
(a) Throughput of DDIO workload (b) L3 Cache Misses (from perf)

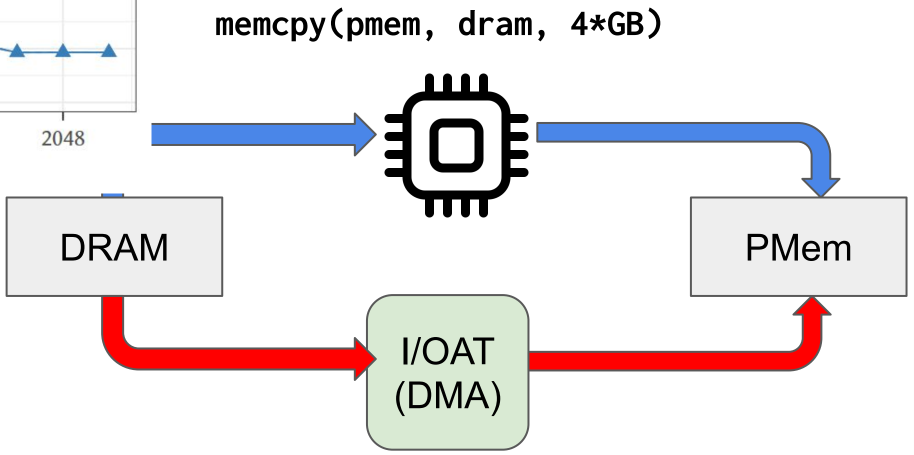
- DDIO reduces data traffic
- Same command, but two paths
- Only tuning knob: global **undocumented** MSR

⇒ Hard to control!

Xeon Gold 6212U, 24 Cores,
~1.5 MB L3 Cache/Core,
Samsung 970 Pro (PCIe 3)

Reason 2: Primitives are hard to target

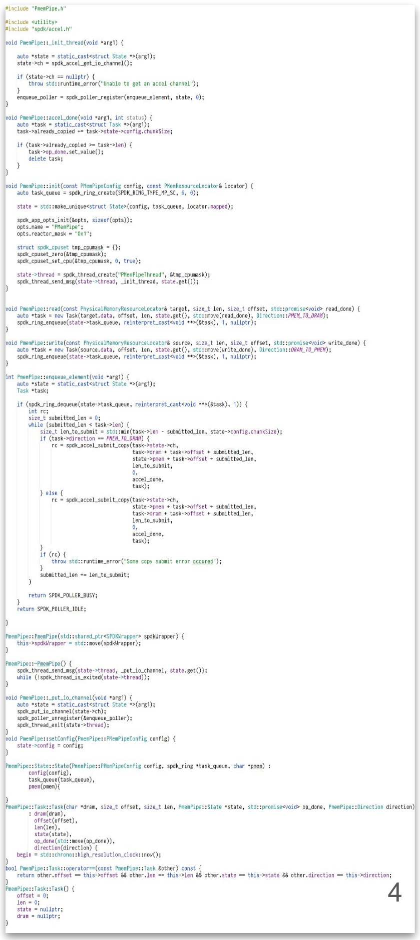




- Same sink, but two paths
- Fast path hard to develop for

⇒ Hard to use!

Xeon Gold 6212U, 24 Cores,
192 GB DRAM, 768 GB PMem

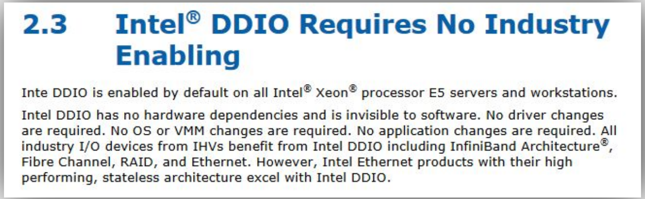


```
#include "PmemPipe.h"

#include <utility>
#include "spdk/accel.h"

void PmemPipe::_init_thread(void *arg1) {

    auto *state = static_cast<struct State *>(arg1);
    state->ch = spdk_accel_get_io_channel();

    if (state->ch == nullptr) {
        throw std::runtime_error("Unable to get an accel channel");
    }
    enqueue_poller = spdk_poller_register(enqueue_element, state, 0);
}

void PmemPipe::accel_done(void *arg1, int status) {
    auto *task = static_cast<struct Task *>(arg1);
    task->already_copied += task->state->config.chunkSize;

    if (task->already_copied >= task->len) {
        task->op_done.set_value();
        delete task;
    }
}

void PmemPipe::init(const PMemPipeConfig config, const PMemResourceLocator& locator) {
    auto task_queue = spdk_ring_create(SPDK_RING_TYPE_MP_SC, 6, 0);

    state = std::make_unique<struct State>(config, task_queue, locator.mapped);

    spdk_app_opts_init(&opts, sizeof(opts));
    opts.name = "PMemPipe";
    opts.reactor_mask = "0x1";

    struct spdk_cpuset tmp_cpumask = {};
    spdk_cpuset_zero(&tmp_cpumask);
    spdk_cpuset_set_cpu(&tmp_cpumask, 0, true);

    state->thread = spdk_thread_create("PMemPipeThread", &tmp_cpumask);
    spdk_thread_send_msg(state->thread, _init_thread, state.get());
}


void PmemPipe::read(const PhysicalMemoryResourceLocator& target, size_t len, size_t offset, std::promise<void> read_done) {
    auto *task = new Task(target.data, offset, len, state.get(), std::move(read_done), Direction::PMEM_TO_DRAM);
    spdk_ring_enqueue(state->task_queue, reinterpret_cast<void **>(&task), 1, nullptr);
}

void PmemPipe::write(const PhysicalMemoryResourceLocator& source, size_t len, size_t offset, std::promise<void> write_done) {
    auto *task = new Task(source.data, offset, len, state.get(), std::move(write_done), Direction::DRAM_TO_PMEM);
    spdk_ring_enqueue(state->task_queue, reinterpret_cast<void **>(&task), 1, nullptr);
}

int PmemPipe::enqueue_element(void *arg1) {
    auto *state = static_cast<struct State *>(arg1);
    Task *task;

    if (spdk_ring_dequeue(state->task_queue, reinterpret_cast<void **>(&task), 1)) {
        int rc;
        size_t submitted_len = 0;
        while (submitted_len < task->len) {
            size_t len_to_submit = std::min(task->len - submitted_len, state->config.chunkSize);
            if (task->direction == PMEM_TO_DRAM) {
                rc = spdk_accel_submit_copy(task->state->ch,
                                            task->dram + task->offset + submitted_len,
                                            state->pmem + task->offset + submitted_len,
                                            len_to_submit,
                                            0,
                                            accel_done,
                                            task);
            } else {
                rc = spdk_accel_submit_copy(task->state->ch,
                                            state->pmem + task->offset + submitted_len,
                                            task->dram + task->offset + submitted_len,
                                            len_to_submit,
                                            0,
                                            accel_done,
                                            task);
            }
            if (rc) {
                throw std::runtime_error("Some copy submit error occured");
            }
            submitted_len += len_to_submit;
        }

        return SPDK_POLLER_BUSY;
    }
    return SPDK_POLLER_IDLE;

}

PmemPipe::PmemPipe(std::shared_ptr<SPDKWrapper> spdkWrapper) {
    this->spdkWrapper = std::move(spdkWrapper);
}

PmemPipe::~PmemPipe() {
    spdk_thread_send_msg(state->thread, _put_io_channel, state.get());
    while (!spdk_thread_is_exited(state->thread));
}

void PmemPipe::_put_io_channel(void *arg1) {
    auto *state = static_cast<struct State *>(arg1);
    spdk_put_io_channel(state->ch);
    spdk_poller_unregister(&enqueue_poller);
    spdk_thread_exit(state->thread);
}
void PmemPipe::setConfig(PmemPipe::PMemPipeConfig config) {
    state->config = config;
}

PmemPipe::State::State(PmemPipe::PMemPipeConfig config, spdk_ring *task_queue, char *pmem) :
        config(config),
        task_queue(task_queue),
        pmem(pmem){

}
PmemPipe::Task::Task(char *dram, size_t offset, size_t len, PmemPipe::State *state, std::promise<void> op_done, PmemPipe::Direction direction)
        : dram(dram),
          offset(offset),
          len(len),
          state(state),
          op_done(std::move(op_done)),
          direction(direction) {
    begin = std::chrono::high_resolution_clock::now();
}
bool PmemPipe::Task::operator==(const PmemPipe::Task &other) const {
    return other.offset == this->offset && other.len == this->len && other.state == this->state && other.direction == this->direction;
}
PmemPipe::Task::Task() {
    offset = 0;
    len = 0;
    state = nullptr;
    dram = nullptr;
}
```

Primitives have to be re-thought

- Problem: Hard to utilize, hard to control
  - Vendor-specific, transparent
  - Varying APIs, unconventional or not widely supported

- The culprit: **implicit**, **bottom-up approach**
  - Vendors see primitives as accelerators for specific use-cases
  - Breaks down when straying from the happy path

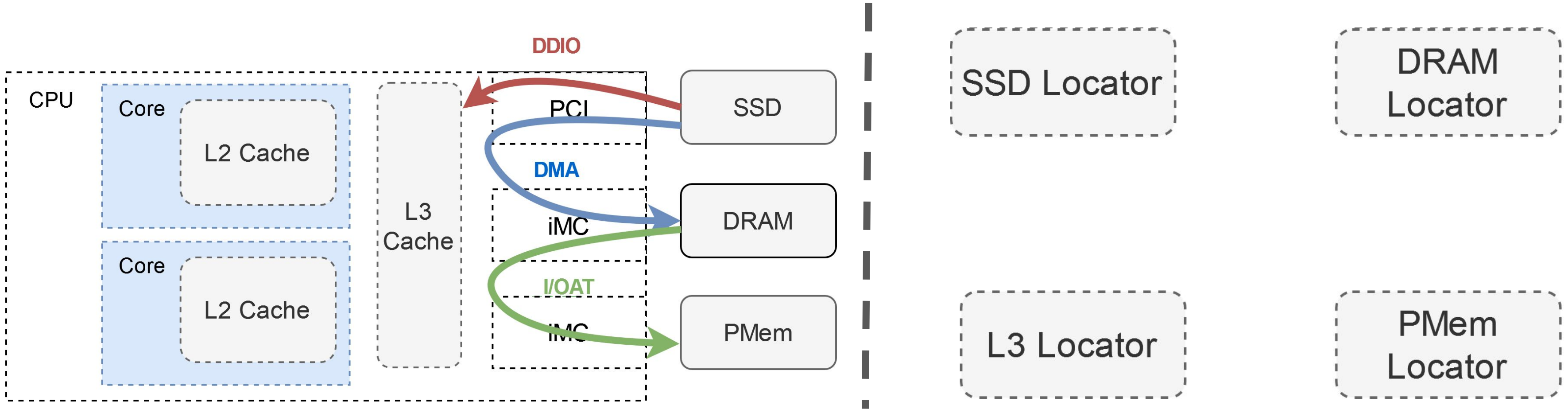

2.3 Intel® DDIO Requires No Industry Enabling

Inte DDIO is enabled by default on all Intel® Xeon® processor E5 servers and workstations.

Intel DDIO has no hardware dependencies and is invisible to software. No driver changes are required. No OS or VMM changes are required. No application changes are required. All industry I/O devices from IHVs benefit from Intel DDIO including InfiniBand Architecture®, Fibre Channel, RAID, and Ethernet. However, Intel Ethernet products with their high performing, stateless architecture excel with Intel DDIO.

- Proposal: **Declarative**, **top-down data movement**
  - Build a **unified abstraction**
  - Give control to the developers
  - We call it: **Data Pipes**

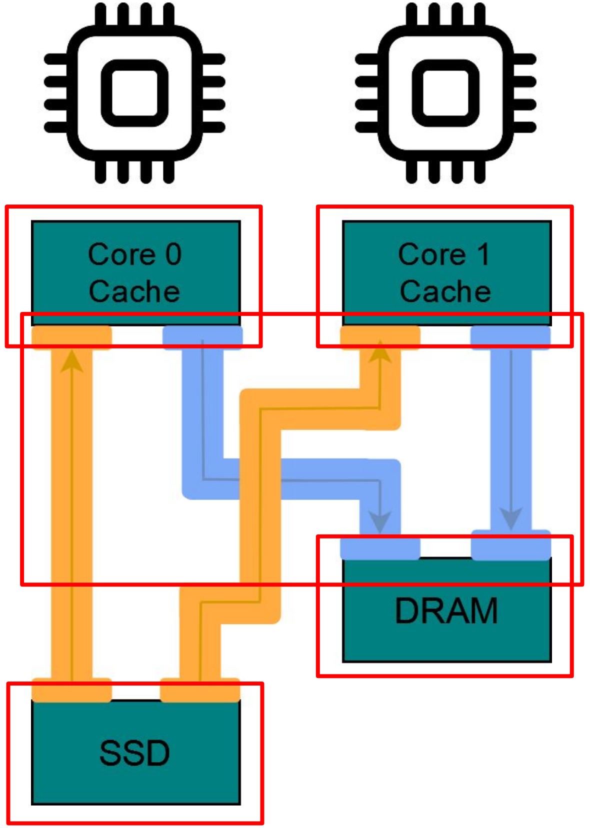
Data Pipes



- “Typed” locators make **data location** explicit
- Pipes make **data movement** explicit
- **Top-down** approach is **declarative** and shows **intentions**
- Framework underneath decides on how data is actually moved
  - Employ DMA hardware capabilities
  - Software fallback

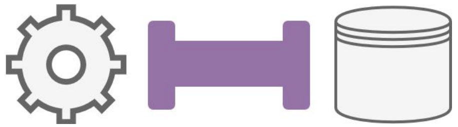
Data Pipes Example

```
SSDResourceLocator ssd(“file/path”);

DRAMResourceLocator dram(1 * GB);

parallel_per_core {

      CacheResourceLocator cache(CACHE_SZ);

      Pipe ssd_up(ssd, cache);

      Pipe cache_down(cache, dram);

      ssd_up.transfer(1 * MB);

      sort(cache, 1 * MB);

      cache_down.transfer(1 * MB);

}
```

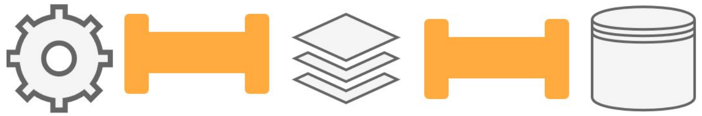


Please take a look at the paper for an **inversion of control** and an **OS-supported** approach

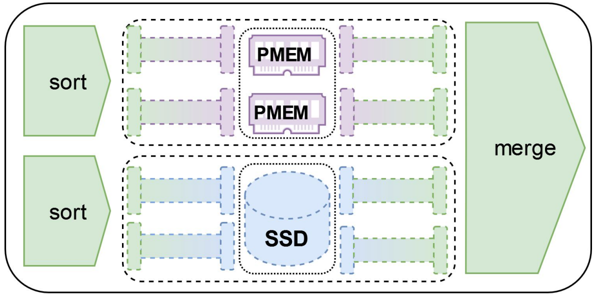
Properties of Data Pipes (1/2)



Declarative

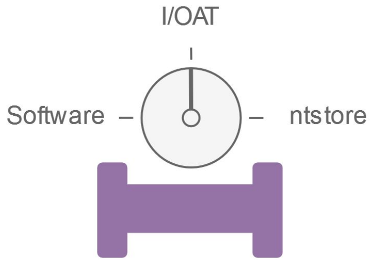


Composable

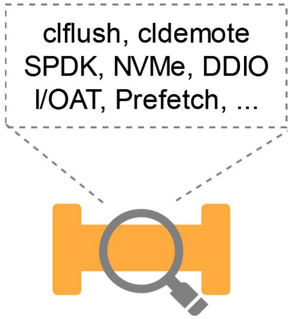


Optimizable

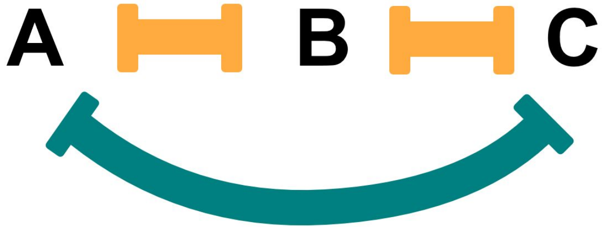
Properties of Data Pipes (2/2)



Configurable



Orthogonal to existing primitives



Adaptable to new primitives

Research Agenda

- Current Prototype: Data Pipes in user space (e.g., SPDK)

- Integrate Data Pipes into the OS
  - Common baseline for different libraries
  - Reserve resources managed by the Data Pipe runtime (e.g., no swapping)
  - API with `epoll`: See paper!

- Cloud Infrastructure Opportunities
  - Predictability => Reduce overprovisioning (noisy neighbors!)
  - Common API helps customers to migrate to new device generations

Hardware Vendor Wishlist

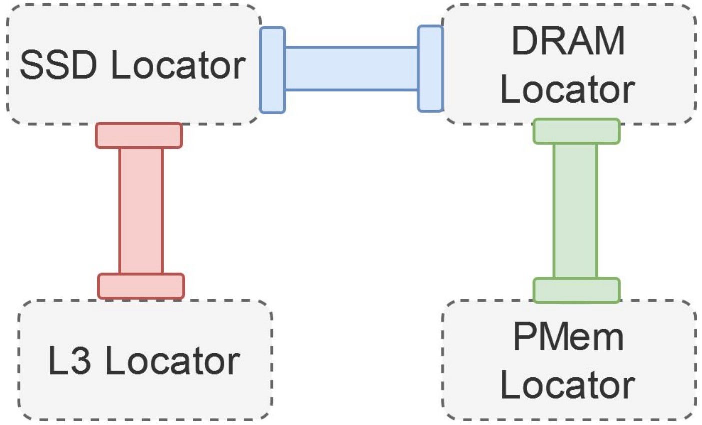
- Give us more access to the hardware!
  - Expose state and tuning knobs of DDIO, I/OAT, …
  - More control over the caching layer

- Intel’s integrated DMA unit lacks some features
  - Lacking documentation
  - Low bandwidth for smaller data transfers
  - Limited number of channels

- Increased reach of DMA units
  - Some device pairs have no acceleration, e.g. DMA directly into the registers?

Conclusions

- Data movement is a bottleneck in data-intensive systems
- Existing data movement primitives are heterogeneous and hard to use
- **Data Pipes**: Abstraction to …
  - … present a uniform interface to the application
  - … make data movement primitives
    - **top-down,**
    - **declarative,**
    - and **explicit.**



Thank you for your attention!

Inversion of Control approach

```
SSDResourceLocator ssd(“file/path”);

DRAMResourceLocator dram(1*GB);

PipeRuntime runtime;

runtime.fork_and_start();

vector<future> futures;

Pipe dram_down(dram, ssd, &runtime);
```

```
parallel_per_core {

    promise<void> write_promise;

    futures.push_back(write_promise.get_future());

    // Do some computation with data in DRAM

    dram_down.transfer(20 * MB, write_promise);

}

wait_all(futures).wait();

futures.clear();
```

OS supported approach

```
SSDResourceLocator ssd(“file/path”);

DRAMResourceLocator dram(1*GB);

int dram_downpipe_fd = create_pipe(dram,
ssd);

int epoll_fd = epoll_create1(0);

epoll_event dram_pipe_op;

dram_pip_op.events = EPOLLTRANSFER;

dram_pip_op.fd = dram_downpipe_fd;

epoll_ctl(epoll_fd, EPOLL_CTL_ADD, 0,
&dram_pipe_op)
```

```
parallel_per_core {

     // Wait until the pipe has capacity to transfer

     epoll_event event;

     epoll_wait(epoll_fd, &event, 1);

     pipe_transfer(dram_downpipe_fd);

}
```