DI-MMAP: A High Performance Memory-Map Runtime for Data-Intensive Applications

DISCS 2012
Nov. 16, 2012

Brian Van Essen, Henry Hsieh (UCLA), Sasha Ames, Maya Gokhale

Lawrence Livermore National Laboratory

LLNL-PRES-602272
This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Motivation

Enable scalable out-of-core computations for data-intensive computing.

Effectively integrate non-volatile random access memory into the HPC node's memory architecture.

Address data-intensive computing scalability challenges:

- Use node-local NVRAM to support larger working sets
- DRAM-cached NVRAM to extend main memory

Allow latency-tolerant applications to be oblivious to transitions from dynamic to persistent memory when accessing out-of-core data.

HPC Challenges and opportunities

- Data-intensive high-performance computing applications:
  - processing of massive real-world graphs
  - bioinformatics / computational biology
  - streamline tracing (in-situ VDA)
- Creating data-intensive architecture is costly and power-intensive
  - In traditional HPC architecture DRAM per core is going down
  - DRAM is expensive: cost and power
- NVRAM technologies promise:
  - lower latency
  - higher density
  - better concurrency
  - minimal static power $\rightarrow$ lower average power

Data-Intensive High-Performance Computing

Data-Intensive Applications:

- large data sets
- large working sets that exceed capacity of main memory
- memory bound
  - irregular data access
  - latency sensitive
  - minimal computation

Latency-tolerant algorithms:

- highly concurrent
- avoid bulk synchronous communication
- potentially asynchronous execution

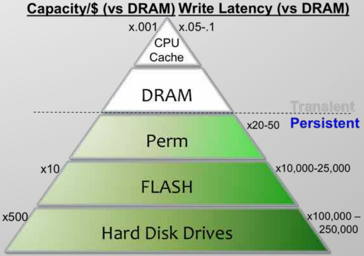
Integrating future NVRAM

Peripherally attached storage in near term

- 2-4 year horizon
- Existing PCIe-attached Flash storage

High-performance PCIe-attached NVRAM

- Low access latency
- Efficient random access
- Faster peripheral bus



Challenges for HPC Runtime

Integrating high-performance storage requires:

- explicit out-of-core algorithms
- seamless integration of storage into memory hierarchy
  - *e.g.* high-performance memory-map

Linux memory-map runtime does not:

- scale well with increased concurrency
- perform well when memory is not freely available

. . . optimize memory-map runtime for data-intensive computing

Direct I/O or memory-mapped I/O

Direct I/O - direct access to NVRAM pages

- Avoids overheads of software stack
- Good for fetching multiple pages of data at once

Memory-Mapped I/O - map file/device into app's virtual memory

- Good for word-level access
- Word access to cached pages is at memory speeds
- Eliminates dichotomy between storage and memory
  - Data structures easily transition out-of-core
  - Can sacrifice performance

Memory-mapped I/O can seamlessly extended the memory hierarchy

Data-intensive memory-map runtime (DI-MMAP)

A high-performance alternative to Linux `mmap`:

- performance scales with increased concurrency
- performance does not degrade under memory pressure
- explicit assignment from data structures to buffers

DI-MMAP features:

- a fixed sized page buffer
- minimal dynamic memory allocation
- a simple FIFO buffer replacement policy
- preferential caching for frequently accessed pages

Using DI-MMAP

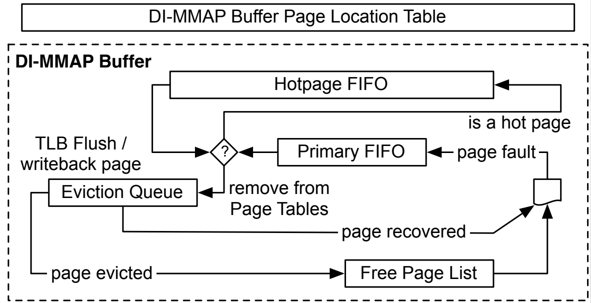
The DI-MMAP device driver:

1. is loaded into a running Linux kernel
2. it allocates a fixed amount of main memory for page buffering
3. it creates a control interface file in the /dev filesystem

Once loaded:

1. the control file is then used to create pseudo-files in `/dev`
2. pseudo-files link (i.e. redirect) to block devices in the system
3. accesses to a pseudo-file are redirected to the linked block device
4. pseudo-file is memory mapped into the applications virtual memory space

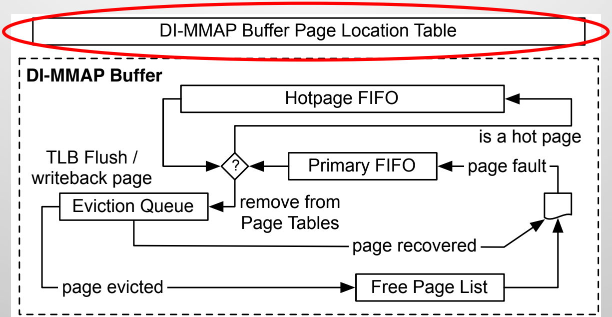
Buffer Management



Minimize the amount of effort needed to find a page to evict:

- In the steady state a page is evicted on each page fault
- Track recently evicted pages to maintain temporal reuse
- Allow bulk TLB operations to reduce inter-processor interrupts

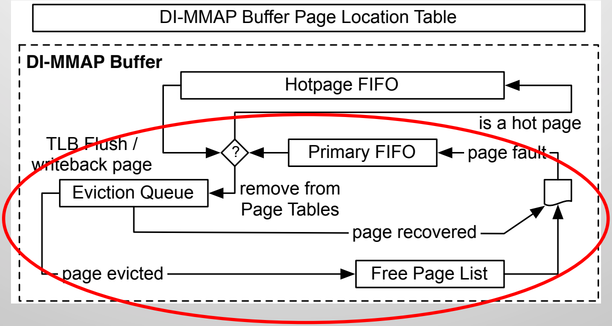
Buffer Management



Minimize the amount of effort needed to find a page to evict:

- In the steady state a page is evicted on each page fault
- Track recently evicted pages to maintain temporal reuse
- Allow bulk TLB operations to reduce inter-processor interrupts

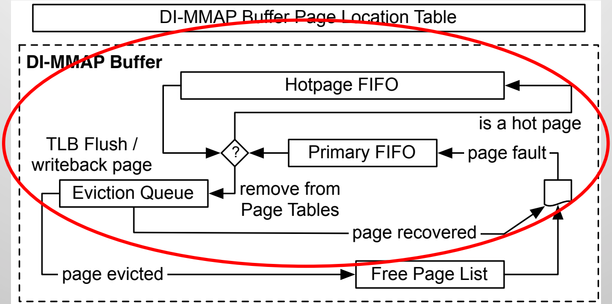
Buffer Management



Minimize the amount of effort needed to find a page to evict:

- In the steady state a page is evicted on each page fault
- Track recently evicted pages to maintain temporal reuse
- Allow bulk TLB operations to reduce inter-processor interrupts

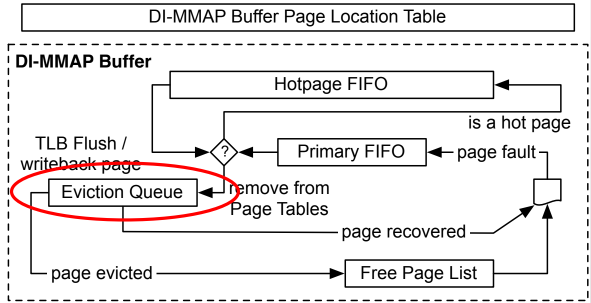
Buffer Management



Minimize the amount of effort needed to find a page to evict:

- In the steady state a page is evicted on each page fault
- Track recently evicted pages to maintain temporal reuse
- Allow bulk TLB operations to reduce inter-processor interrupts

Buffer Management



Minimize the amount of effort needed to find a page to evict:

- In the steady state a page is evicted on each page fault
- Track recently evicted pages to maintain temporal reuse
- Allow bulk TLB operations to reduce inter-processor interrupts

Livermore random I/O testbench (LRIOT)

Currently lack tools to effectively measure and evaluate NVRAM

- high speed
- highly concurrency
- tolerate complex and unstructured access patterns

FIO: industry standard for benchmarking

- Does not scale well
- Cannot mix concurrency with both processes and threads

LRIOT: high concurrency / high throughput benchmarking tool

- Supports a mixture of processes and threads
- Multiple random and deterministic access patterns
- More deterministic timing measurements

LRIOT system setup

Test platform:

- 16 core AMD 8356 Opteron system @ 2.3GHz
- 64 GiB of DRAM
- RHEL 6 2.6.32
- 3$\times$ 80 GiB SLC NAND Flash Fusion-io ioDrive PCIe 1.1 x4 cards
  - striped RAID 0

Benchmark:

- uniform random I/O pattern
- 6.4 million reads (unique pages) $\rightarrow$ 24 GiB working set
- 128 GiB file

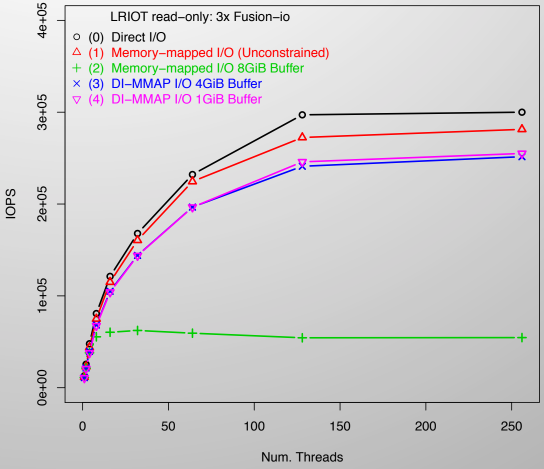
Read-only LRIOT benchmark

Linux `mmap`:

- Unconstrained performs well
- drops dramatically with 8GiB of page cache

DI-MMAP:

- much better with fixed sized buffer
- only loses 15% performance from direct I/O with 1 GiB buffer



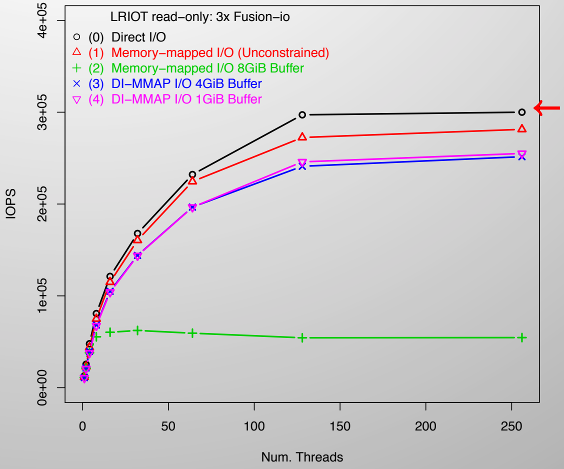
Read-only LRIOT benchmark

Linux `mmap`:

- Unconstrained performs well
- drops dramatically with 8GiB of page cache

DI-MMAP:

- much better with fixed sized buffer
- only loses 15% performance from direct I/O with 1 GiB buffer



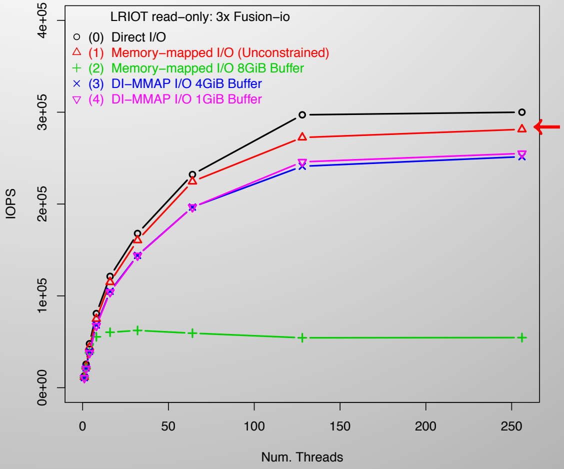
Read-only LRIOT benchmark

Linux `mmap`:

- Unconstrained performs well
- drops dramatically with 8GiB of page cache

DI-MMAP:

- much better with fixed sized buffer
- only loses 15% performance from direct I/O with 1 GiB buffer



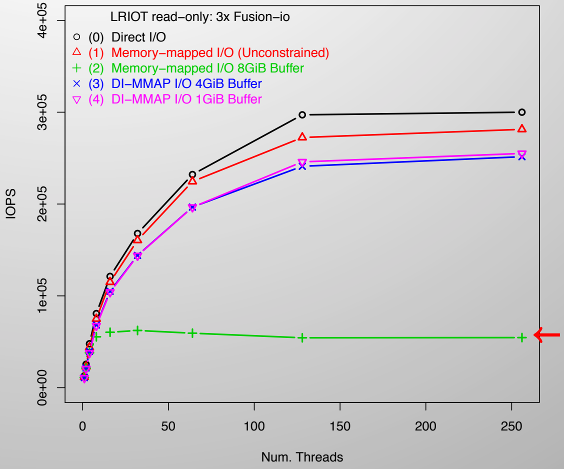
Read-only LRIOT benchmark

Linux `mmap`:

- Unconstrained performs well
- drops dramatically with 8GiB of page cache

DI-MMAP:

- much better with fixed sized buffer
- only loses 15% performance from direct I/O with 1 GiB buffer



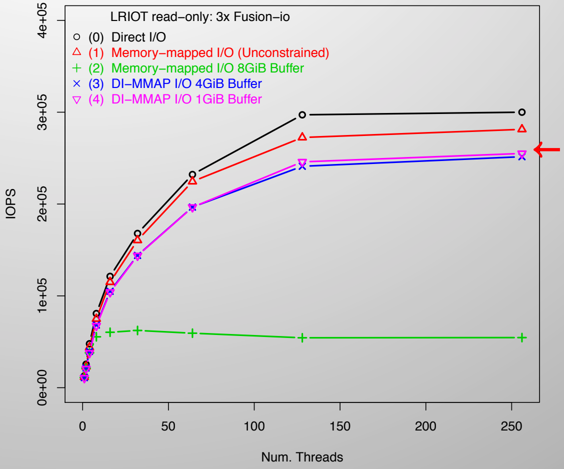
Read-only LRIOT benchmark

Linux `mmap`:

- Unconstrained performs well
- drops dramatically with 8GiB of page cache

DI-MMAP:

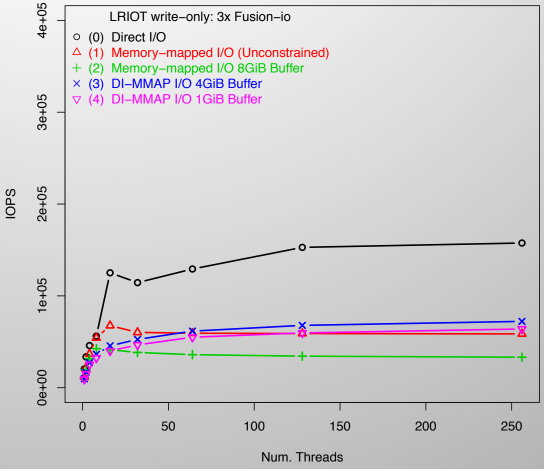
- much better with fixed sized buffer
- only loses 15% performance from direct I/O with 1 GiB buffer



Write-only LRIOT benchmark

DI-MMAP:

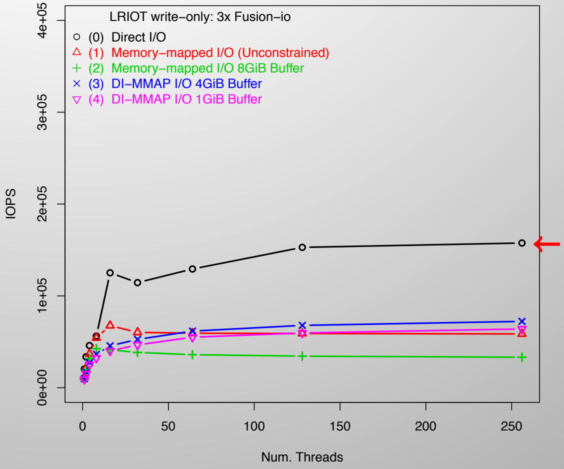
- on par with unconstrained Linux `mmap`
- $>2\times$ Linux `mmap` with 8GiB page cache
- does not match performance of direct I/O
  (subject to further investigation)



Write-only LRIOT benchmark

DI-MMAP:

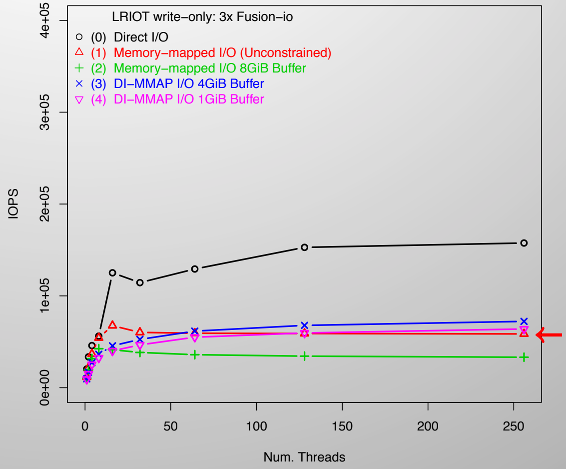
- on par with unconstrained Linux `mmap`
- $>2\times$ Linux `mmap` with 8GiB page cache
- does not match performance of direct I/O
  (subject to further investigation)



Write-only LRIOT benchmark

DI-MMAP:

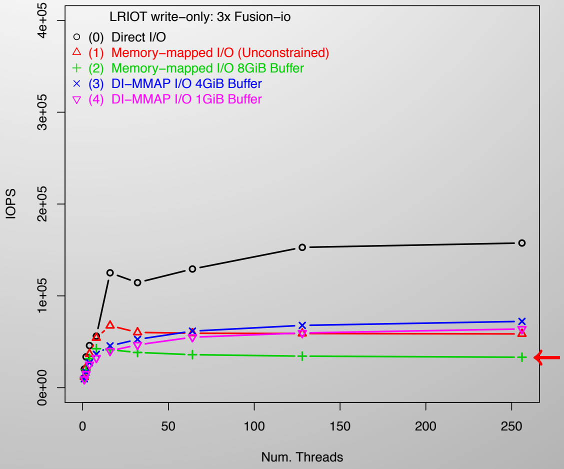
- on par with unconstrained Linux `mmap`
- $> 2\times$ Linux `mmap` with 8GiB page cache
- does not match performance of direct I/O
  (subject to further investigation)



Write-only LRIOT benchmark

DI-MMAP:

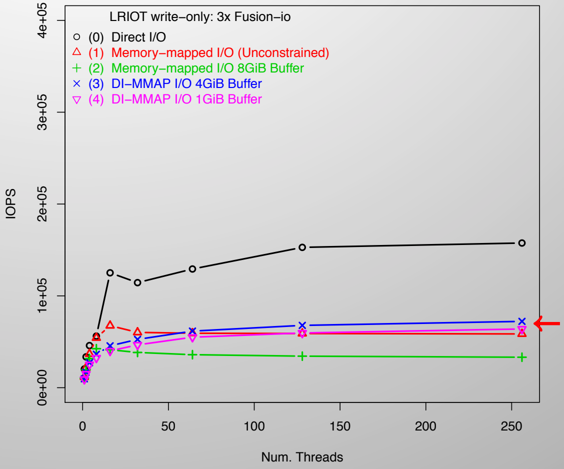
- on par with unconstrained Linux `mmap`
- $> 2\times$ Linux `mmap` with 8GiB page cache
- does not match performance of direct I/O (subject to further investigation)



Write-only LRIOT benchmark

DI-MMAP:

- on par with unconstrained Linux `mmap`
- $> 2\times$ Linux `mmap` with 8GiB page cache
- does not match performance of direct I/O (subject to further investigation)



Microbenchmarks system setup

Test platform:

- 8 core AMD 2378 Opteron system @ 2.4GHz
- 16 GiB of DRAM
- 2$\times$ 200 GiB SLC NAND Flash Virident tachIOn Drive PCIe 1.1 x8

Benchmarks:

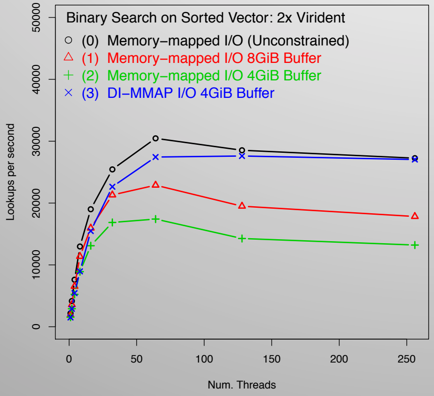
1. Binary search on sorted vector
2. Lookup on Ordered Map (Red-Black Tree)
3. Lookup on Unordered Map (Hash Table)

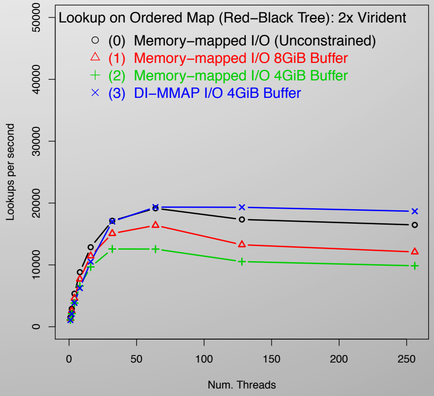
- database size ranged from $\sim$ 112GiB to $\sim$ 135GiB
- each micro-benchmark issued 2^{20} queries

Microbenchmarks: BST and Ordered Map

DI-MMAP:

- significantly exceeds the performance of Linux mmap when each is constrained to an equal amount of buffering
- approaches the performance of mmap with no memory constraint

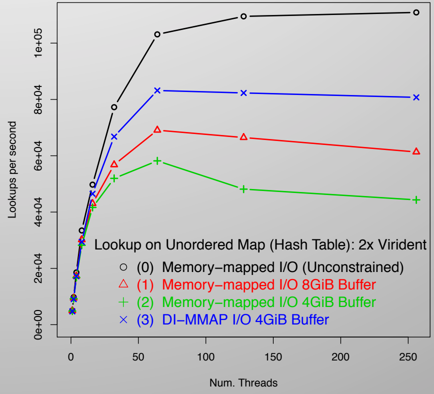




Microbenchmarks: Unordered map

DI-MMAP:

- significantly exceeds the performance of Linux mmap when each is constrained to an equal amount of buffering
- approaches the performance of mmap with no memory constraint



Metagenomic Search & Classification

Metagenomics:

- sequencing of heterogenous genetic fragments
- fragments (aka reads) may be derived from many organisms

Application queries a database of genetic markers called k-mers:

- length k sequences out of a DNA, RNA, or protein alphabet
- k-mer database stored in Flash storage
- access patterns to the datasets are extremely random
- classification requires global view of reference database

Two tests:

- k-mer lookup
- sample classification

Metagenomics Search & Classification system setup

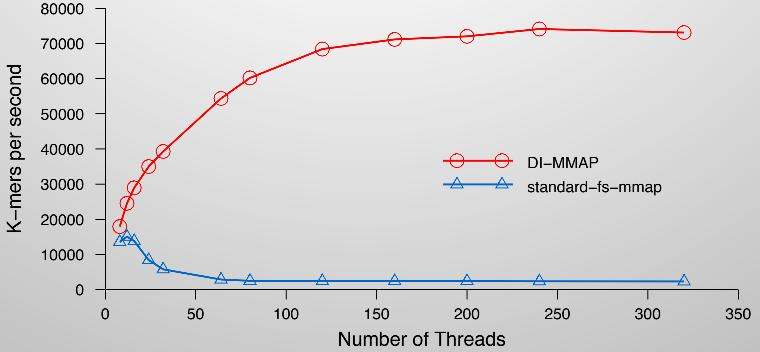
Test platform:

- 4 socket, 40 core, Intel E7 4850 @ 2 GHz
- 1 TiB DRAM
- Linux kernel 2.6.32 (Red Hat Enterprise 6).
- 2$\times$ Fusion-io 1.2 TB ioDrive2 cards PCIe-2.0 x4
  - RAID 0
  - block sizes of 4 KiB
- 16 GiB DRAM available for buffer cache

Application:

- k = 18
- database size is 635 GiB

K-mer lookup

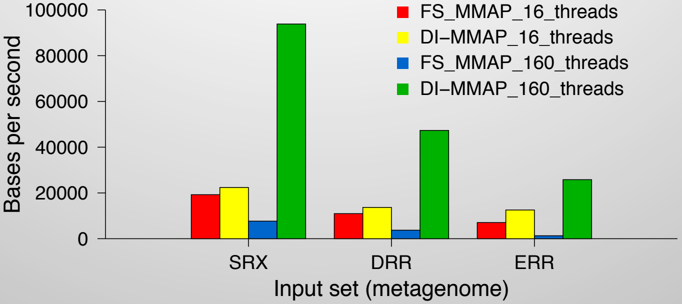


Peak Performance:

- 16 threads with Linux `mmap`
- 240 threads for DI-MMAP

Lookups per second with DI-MMAP is 4.92× higher than with Linux `mmap`

Metagenomic Sample Classification



Near peak performance:

- 16 threads for Linux `mmap`
- 160 threads for DI-MMAP

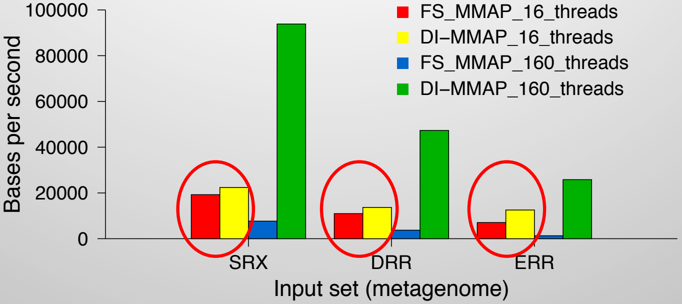
Performance advantage of DI-MMAP vs. Linux `mmap`:

- 4.88× for SRX input set
- 3.66× for ERR input set

Performance varies with:

- % of redundant k-mers
- diversity of metagenome (*e.g.* ERR)

Metagenomic Sample Classification



Near peak performance:

- 16 threads for Linux `mmap`
- 160 threads for DI-MMAP

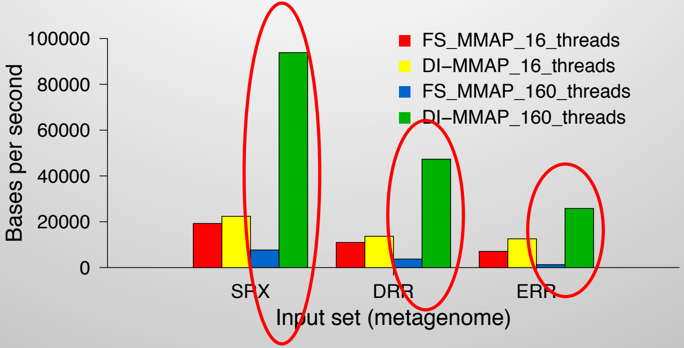
Performance advantage of DI-MMAP vs. Linux `mmap`:

- 4.88× for SRX input set
- 3.66× for ERR input set

Performance varies with:

- % of redundant k-mers
- diversity of metagenome (*e.g.* ERR)

Metagenomic Sample Classification



Near peak performance:

- 16 threads for Linux `mmap`
- 160 threads for DI-MMAP

Performance advantage of DI-MMAP vs. Linux `mmap`:

- 4.88× for SRX input set
- 3.66× for ERR input set

Performance varies with:

- % of redundant k-mers
- diversity of metagenome (*e.g.* ERR)

Conclusions

The data-intensive memory-map (DI-MMAP) runtime:

1. provides scalable, out-of-core performance for data-intensive applications
2. allows increased performance of algorithms with increased concurrency
3. performance does not significantly degrade with smaller buffer size

DI-MMAP:

- provides a viable solution for scalable out-of-core algorithms
- offloads the explicit buffering requirements from the application to the runtime
- allows the application to access its external data through a simple load/store interface
- hides much of the complexity of data movement
- approaches the raw, peak performance of direct I/O

Thank You!

Questions?

Open source release is in progress:

https://computation.llnl.gov/casc/dcca-pub/dcca/Data-centric_architecture.html