HiPC 2019

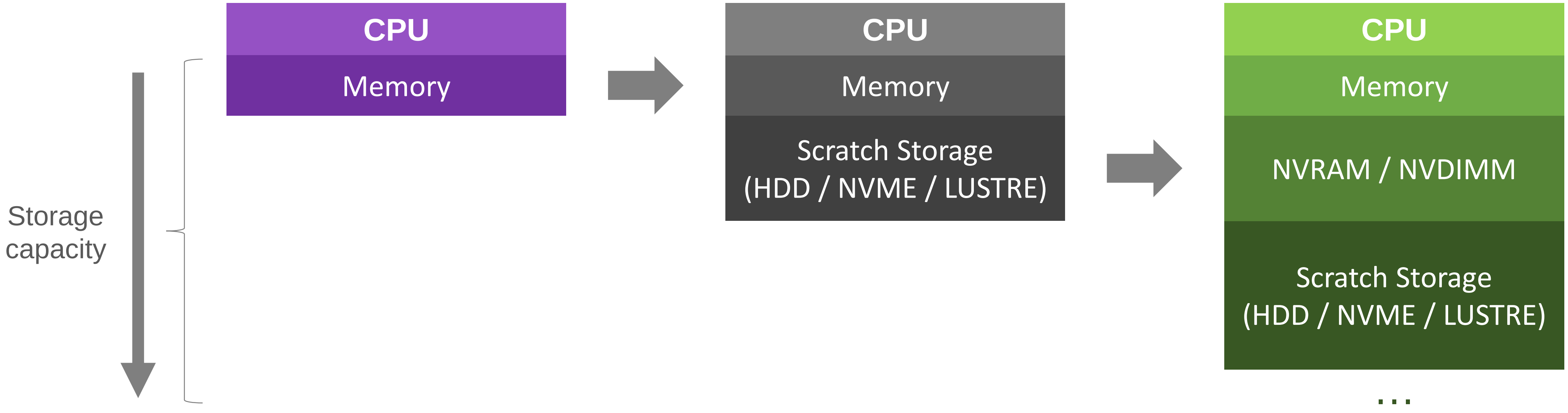
uMMAP-IO: User-level Memory-mapped I/O for HPC

Sergio Rivas-Gomez | Alessandro Fanfarillo | Sebastien Valat | Christophe Laferriere | Philippe Couvee | Sai Narasimhamurthy
KTH KTH ATOS ATOS ATOS Seagate

Stefano Markidis
KTH

Introduction > Increasing heterogeneity per node

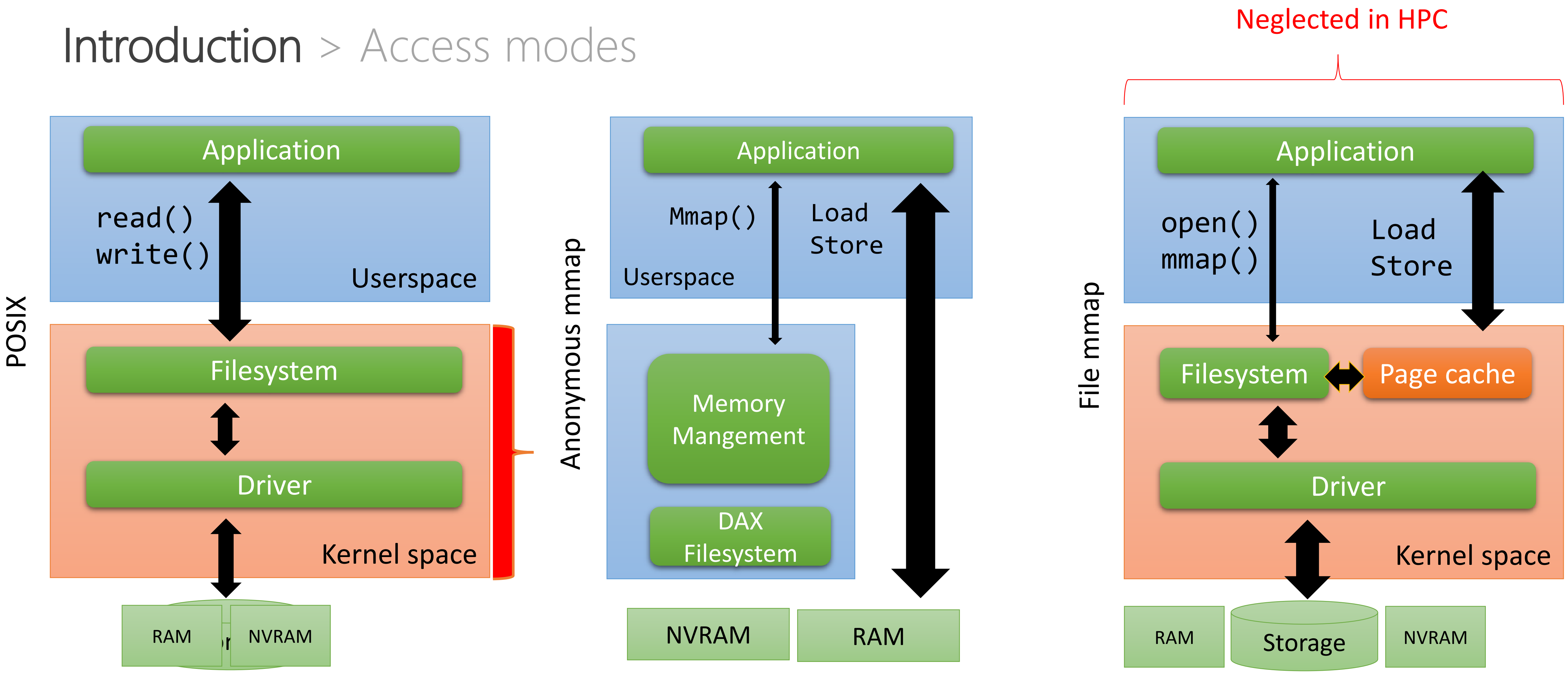
Emerging storage technologies are evolving so rapidly that the existing gap between main memory and I/O subsystem performances is thinning. Next-generation supercomputers will feature a variety of Non-Volatile RAM (NVRAM), with different performance characteristics, next to traditional DRAM:



Introduction > Wide-range of interfaces

Furthermore, **memory and storage are programmed using separate interfaces.** In fact, over the past two decades, the diversity of standardized functionality has grown considerably (e.g., collective I/O operations in MPI IO):

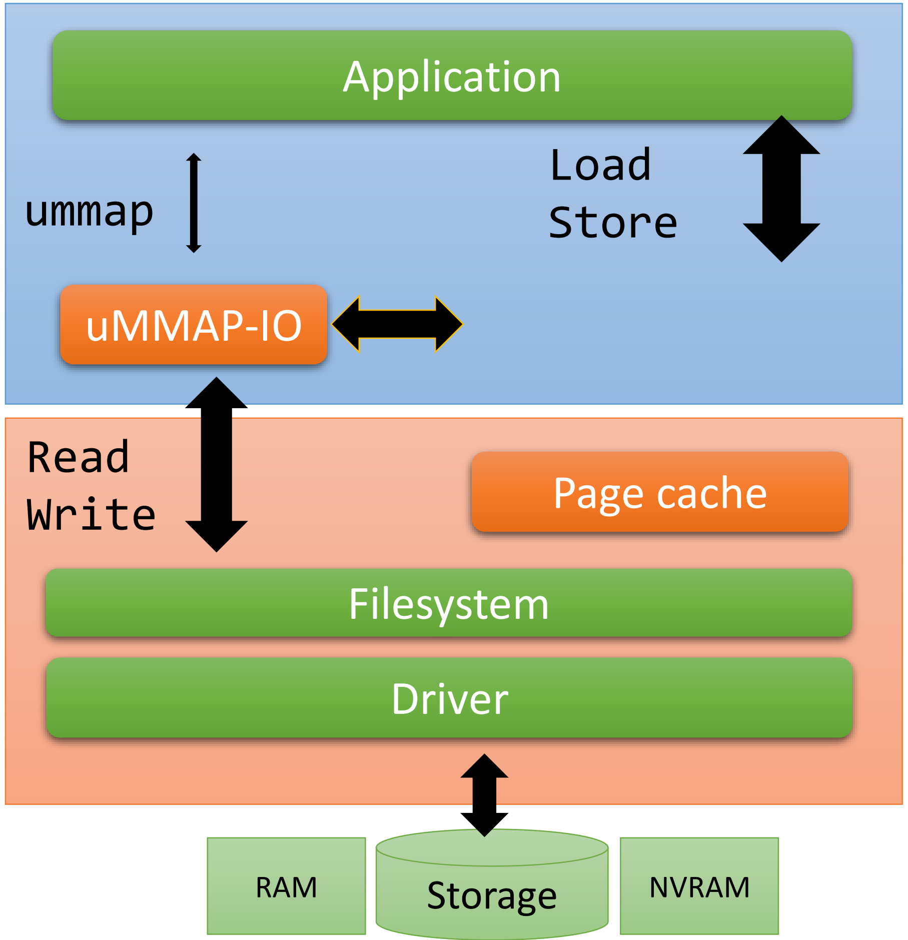
Introduction > Access modes



Proposal > A user-space managed memory mapping IO

We want to **fully handle** the Memory **mapping** to the FS in **user space**.

Can be transparently replaced to RAM or NVRAM mapping.

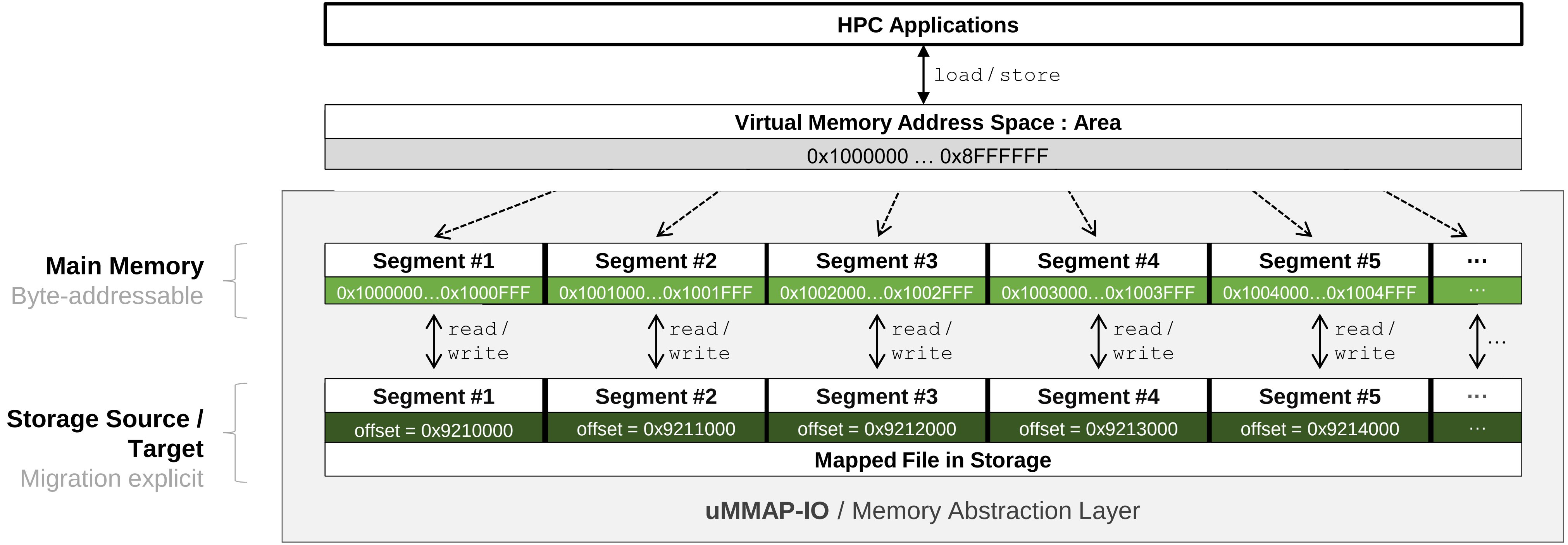


Proposal > fundamental mechanism

- **mmap()** a memory segment
- **mprotect()** is to disalow read/write access via **PROT_NONE**
- Use **SEGFAULT** capture to detect **first access**:
  - On **read**, **fetch** and allow
  - On **write**, fetch is needed, allow and mark **dirty**
- **madvise()** to release page via **DONT_NEED** flag

Methodology > Memory Abstraction Layer

User-level Memory-mapped I/O (uMMAP-IO) provides a memory abstraction layer to manage allocations. Such abstraction separates virtual address spaces from their correspondent mapping to storage:



Methodology > Prog. Interface

The interface of uMMAP-IO is designed to resemble the memory-mapped I/O implementation of the OS.

We define several functions that provide the support of the functionality featured in our library:

- `ummap` - **Establishes a memory-mapping of a file.** Applications can choose seg. size, offset in file, I/O thread sync. freq., or even read content (i.e., R bit).
- `umsync` - **Performs selective data synchronizations,** and optionally allows to reduce the RSS.
- `umremap` - **Allows to re-configure a given mapping,** useful for fault-tolerance (e.g., incr. checkpoints).
- `umunmap` - **Releases a memory-mapping,** allowing to perform a data synchronization, if required.

```
...
// Open the file descriptor for the mapping
int fd = open("/path/to/file", flags, mode);

// Ensure that the file has space (optional)
ftruncate(fd, size);

// Create the memory-mapping with uMMAP-IO
ummap(size, segsize, prot, fd, offset, sync_freq, FALSE,
      policy, (void **)&baseptr);

// It is now safe to close the file descriptor
close(fd);

// Set some random value using load / store
for (off_t i = 0; i < size; i++)
{
    baseptr[i] = 21;
}

// Alternative: Use traditional memory functions
memset(baseptr, 21, size);

// Synchronize with storage to ensure data consistency
umsync(baseptr, FALSE);

// Finally, release the mapping if no longer needed
umunmap(baseptr, TRUE);
...
```

Methodology > Prog. Interface

The interface of uMMAP-IO is designed to resemble the memory-mapped I/O implementation of the OS.

We define several functions that provide the support of the functionality featured in our library:

- `ummap` - **Establishes a memory-mapping of a file.** Applications can choose seg. size, offset in file, I/O thread sync. freq., or even read content (i.e., R bit).
- `umsync` - **Performs selective data synchronizations,** and optionally allows to reduce the RSS.
- `umremap` - **Allows to re-configure a given mapping,** useful for fault-tolerance (e.g., incr. checkpoints).
- `umunmap` - **Releases a memory-mapping,** allowing to perform a data synchronization, if required.

```c
...
// Open the file descriptor for the mapping
int fd = open("/path/to/file", flags, mode);

// Ensure that the file has space (optional)
ftruncate(fd, size);

// Create the memory-mapping with uMMAP-IO
ummap(size, segsize, prot, fd, offset, sync_freq, FALSE,
      policy, (void **)&baseptr);

// It is now safe to close the file descriptor
close(fd);

// Set some random value using load / store
for (off_t i = 0; i < size; i++)
{
    baseptr[i] = 21;
}

// Alternative: Use traditional memory functions
memset(baseptr, 21, size);

// Synchronize with storage to ensure data consistency
umsync(baseptr, FALSE);

// Finally, release the mapping if no longer needed
umunmap(baseptr, TRUE);
...
```

Methodology > Prog. Interface

The interface of uMMAP-IO is designed to resemble the memory-mapped I/O implementation of the OS.

We define several functions that provide the support of the functionality featured in our library:

- **ummap** … Applic… thread sync. freq., or even read content (i.e., R bit).
- **umsync - Performs selective data synchronizations**, and optionally allows to reduce the RSS.
- **umremap - Allows to re-configure a given mapping**, useful for fault-tolerance (e.g., incr. checkpoints).
- **umunmap - Releases a memory-mapping**, allowing to perform a data synchronization, if required.

```
...
// Open the file descriptor for the mapping
int fd = open("/path/to/file", flags, mode);

// Ensure that the file has space (optional)
ftruncate(fd, size);

// Create the memory-mapping with uMMAP-IO
ummap(size, segsize, prot, fd, offset, sync_freq, FALSE,
      policy, (void **)&baseptr);

// It is now safe to close the file descriptor

...e random value using load / store
..._ i = 0; i < size; i++)
{
    baseptr[i] = 21;
}

// Alternative: Use traditional memory functions
memset(baseptr, 21, size);

// Synchronize with storage to ensure data consistency
umsync(baseptr, FALSE);

// Finally, release the mapping if no longer needed
umunmap(baseptr, TRUE);
...
```

As with the memory-mapped I/O of the OS,
a file must be opened and resized for the mapping

Methodology > Prog. Interface

The interface of uMMAP-IO is designed to resemble the memory-mapped I/O implementation of the OS.

We define several functions that provide the support of the functionality featured in our library:

- **ummap - Establishes a memory-mapping of a file.** Applications can choose seg. size, offset in file, I/O thread sync. fre
- **umsync - Perfo** and optionally allows to reduce the RSS.
- **umremap - Allows to re-configure a given mapping,** useful for fault-tolerance (e.g., incr. checkpoints).
- **umunmap - Releases a memory-mapping**, allowing to perform a data synchronization, if required.

```
...
// Open the file descriptor for the mapping
int fd = open("/path/to/file", flags, mode);

// Ensure that the file has space (optional)
ftruncate(fd, size);

// Create the memory-mapping with uMMAP-IO
ummap(size, segsize, prot, fd, offset, sync_freq, FALSE,
      policy, (void **)&baseptr);

// It is now safe to close the file descriptor
close(fd);

// Set some random value using load / store
for (off_t i = 0; i < size; i++)

// Alternative: Use traditional memory functions
memset(baseptr, 21, size);

// Synchronize with storage to ensure data consistency
umsync(baseptr, FALSE);

// Finally, release the mapping if no longer needed
umunmap(baseptr, TRUE);
...
```

The mapping can be configured using the opened file and other settings (e.g., evict policy for out-of-core)

Methodology > Prog. Interface

The interface o
memory-mapp

At this point, **the provided pointer can be utilized with `load` / `store`**, or even conventional mem. functions

We define several functions that provide the support of the functionality featured in our library:

- **`ummap` - Establishes a memory-mapping of a file.** Applications can choose seg. size, offset in file, I/O thread sync. freq., or even read content (i.e., R bit).
- **`umsync` - Performs selective data synchronizations**, and optionally allows to reduce the RSS.
- **`umremap` - Allows to re-configure a given mapping**, useful for fault-tolerance (e.g., incr. checkpoints).
- **`umunmap` - Releases a memory-mapping**, allowing to perform a data synchronization, if required.

```
...
// Open the file descriptor for the mapping
int fd = open("/path/to/file", flags, mode);

                  file has space (optional)
                 );

                 ry-mapping with uMMAP-IO
                 e, prot, fd, offset, sync_freq, FALSE,
        policy, (void **)&baseptr);

// It is now safe to close the file descriptor
close(fd);

// Set some random value using load / store
for (off_t i = 0; i < size; i++)
{
    baseptr[i] = 21;
}

// Alternative: Use traditional memory functions
memset(baseptr, 21, size);

// Synchronize with storage to ensure data consistency
umsync(baseptr, FALSE);

// Finally, release the mapping if no longer needed
umunmap(baseptr, TRUE);
...
```

Methodology > Prog. Interface

The interface of uMMAP-IO is designed to resemble the memory-mapped I/O implementation of the OS.

We define several functions that provide the support of the functionality featured in our library:

- `ummap` - **Establishes a memory-mapping of a file.** Applications can choose s thread sync. freq., or even
- `umsync` - **Performs selective data synchronizations,** and optionally allows to reduce the RSS.
- `umremap` - **Allows to re-configure a given mapping,** useful for fault-tolerance (e.g., incr. checkpoints).
- `umunmap` - **Releases a memory-mapping**, allowing to perform a data synchronization, if required.

```
...
// Open the file descriptor for the mapping
int fd = open("/path/to/file", flags, mode);

// Ensure that the file has space (optional)
ftruncate(fd, size);

// Create the memory-mapping with uMMAP-IO
ummap(size, segsize, prot, fd, offset, sync_freq, FALSE,
      policy, (void **)&baseptr);

// It is now safe to close the file descriptor
close(fd);

                          e using load / store
                          ze; i++)

;

// Alternative: Use traditional memory functions
memset(baseptr, 21, size);

// Synchronize with storage to ensure data consistency
umsync(baseptr, FALSE);

// Finally, release the mapping if no longer needed
umunmap(baseptr, TRUE);
...
```

When data consistency must be guaranteed, selective synchronizations can be performed

Methodology > Prog. Interface

The interface of uMMAP-IO is designed to resemble the memory-mapped I/O implementation of the OS.

We define several functions that provide the support of the functionality featured in our library:

- `ummap` - **Establishes a memory-mapping of a file.** Applications can choose seg. size, offset in file, I/O thread sync. freq., or even read content (i.e., R bit).
- `umsync` - **Performs selective data synchronizations** and optionally allows to reduce the RSS.
- `umremap` - **Allows to re-configure a given mapping**, useful for fault-tolerance (e.g., incr. checkpoints).
- `umunmap` - **Releases a memory-mapping**, allowing to perform a data synchronization, if required.

```
...
// Open the file descriptor for the mapping
int fd = open("/path/to/file", flags, mode);

// Ensure that the file has space (optional)
ftruncate(fd, size);

// Create the memory-mapping with uMMAP-IO
ummap(size, segsize, prot, fd, offset, sync_freq, FALSE,
      policy, (void **)&baseptr);

// It is now safe to close the file descriptor
close(fd);

// Set some random value using load / store
for (off_t i = 0; i < size; i++)
{
    baseptr[i] = 21;
}

aditional memory functions
ze);

// Synchronize with storage to ensure data consistency
umsync(baseptr, FALSE);

// Finally, release the mapping if no longer needed
umunmap(baseptr, TRUE);
...
```

When finished, we **unmap**

Methodology > Prog. Interface

The interface of uMMAP-IO is designed to resemble the

The "malloc-like" interface provides an easy-to-use alternative for existing scientific applications

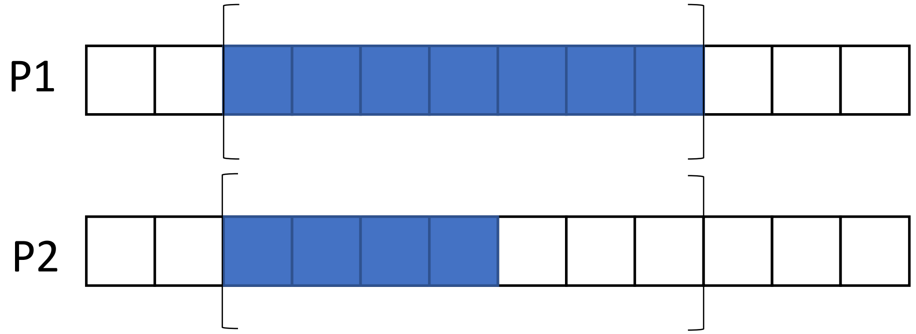
the functionality featured in our library:

- `ummap` - **Establishes a memory-mapping of a file**. Applications can choose seg. size, offset in file, I/O thread sync. freq., or even read content (i.e., R bit).
- `umsync` - **Performs selective data synchronizations**, and optionally allows to reduce the RSS.
- `umremap` - **Allows to re-configure a given mapping**, useful for fault-tolerance (e.g., incr. checkpoints).
- `umunmap` - **Releases a memory-mapping**, allowing to perform a data synchronization, if required.

```
...
// Create the file-backed allocation
baseptr = umalloc("/path/to/file", size);

// Set some random value using load / store
for (off_t i = 0; i < size; i++)
{
    baseptr[i] = 21;
}

// Alternative: Use traditional memory functions
memset(baseptr, 21, size);

// Synchronize with storage to ensure data consistency
usync(baseptr);

// Finally, release the mapping if no longer needed
ufree(baseptr);
...
```

Methodology > Policy

- Configure the user-space **eviction rules**
- **Maximum size** of user space page **cache**
- **Policies** : FIFO, LIFO, pLRU, WIRO
- Can maximum memory of **multiple processes**
  - Via **shared segment** communication

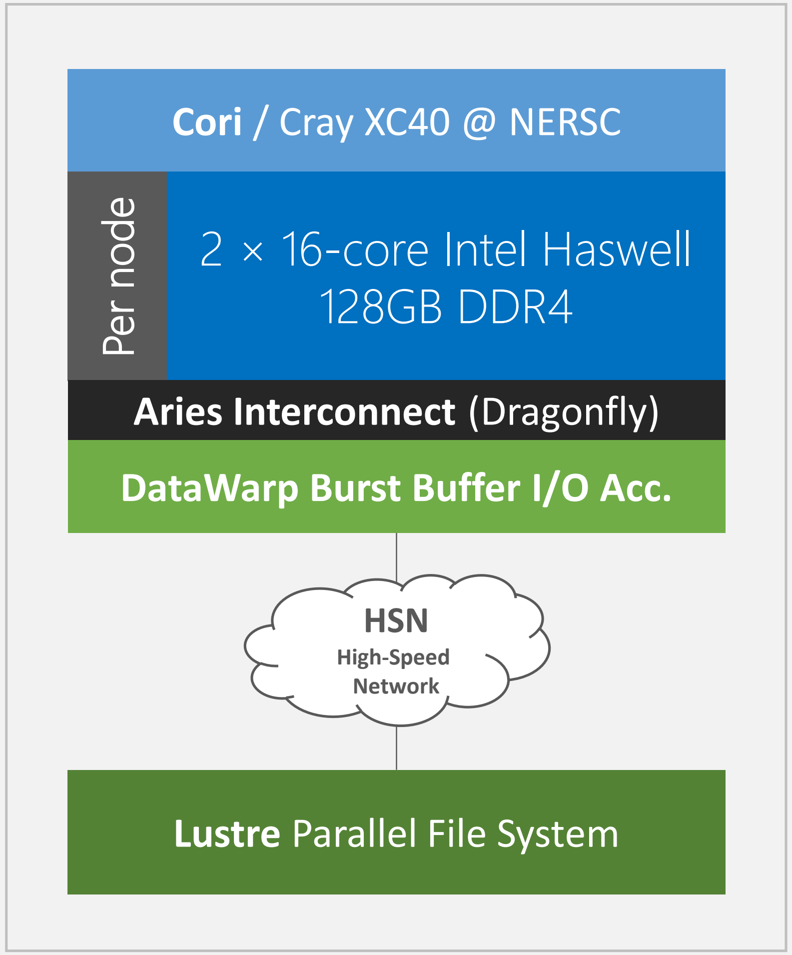


Evaluation > Experimental setup

The first testbed is a **leadership-class supercomputer from the National Energy Research Scientific Computing Center (NCAR)**:

Specifications of Supercomputer "Cori"	
Nodes	2388 × Haswell-based compute nodes
Processor	2 × 16-core Haswell E5-2698v3 @ 2.3GHz
Memory	128GB DRAM per node
Storage	Cray DataWarp Burst Buffer (DVS v0.9) + Lustre PFS (Client v2.7.5) with 248 × OST Servers
Software	SUSE SLES v12 / ICC v18.0.1 and Intel MPI v2018.up1

The goal is to **understand how uMMAP-IO compares to the memory-mapped I/O using Lustre and the Burst Buffer**, which is not supported in the latter case.



Cray XC40 Supercomputer

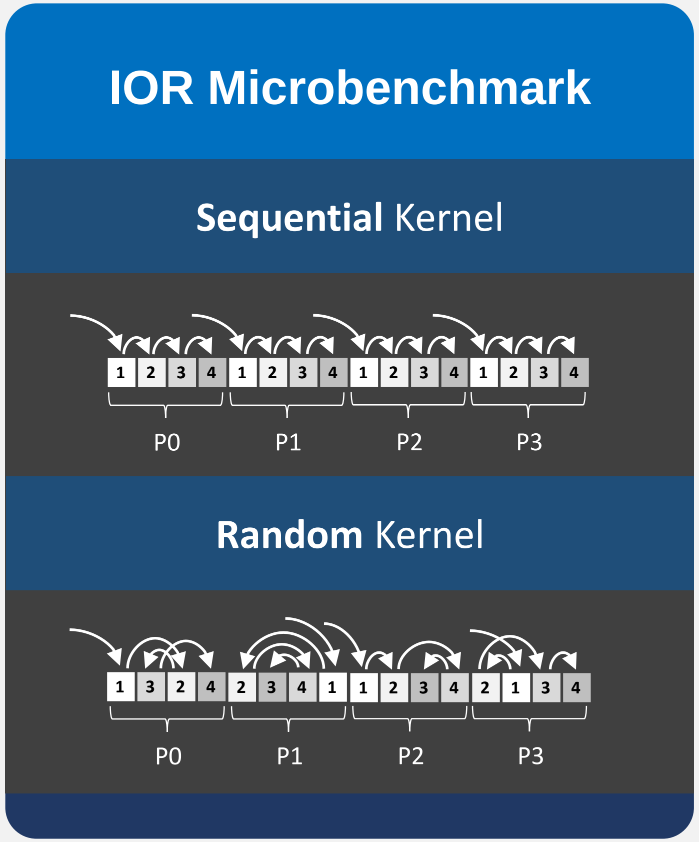
Evaluation > IOR Microbenchmark

Interleaved Or Random (IOR) [4] is a microbenchmark for evaluating the performance of traditional PFS.

IOR measures sustained throughput performing large amounts of small, non-overlapping I/O accesses over shared files. For this purpose, two types of kernels are executed:

- **Sequential.** Illustrates sequential accesses over the file.
- **Random.** Demonstrates pseudo-random accesses.

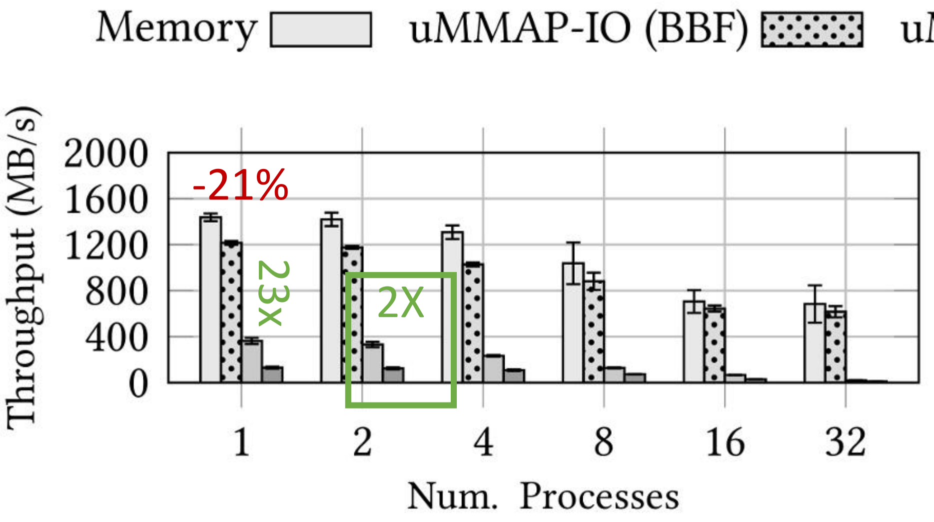
The type of workload (i.e., small data transfers) **represents the worst-case for uMMAP-IO** and should benefit the traditional memory-mapped I/O of the OS. **Hence, the importance of IOR.**



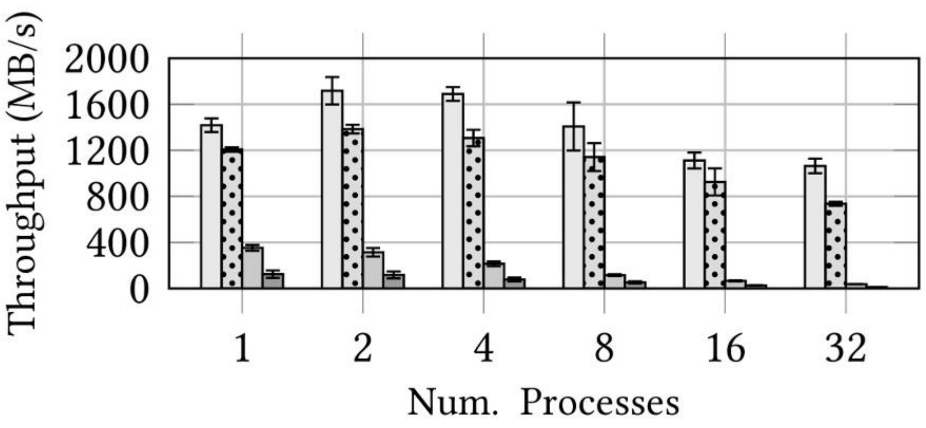
[4] https://github.com/hpc/ior

Evaluation > IOR Microbenchmark [Cori]

The figure shows performance of `SEQ` and `RND` kernels on Cori with an I/O transfer block of 256KB, 1GB allocation per process and 256KB segments. The results illustrate conventional mem. allocations, uMMAP-IO, and MMAP-IO:



(a) **IOR Weak Scaling** (SEQ Kernel / Ind. BW)



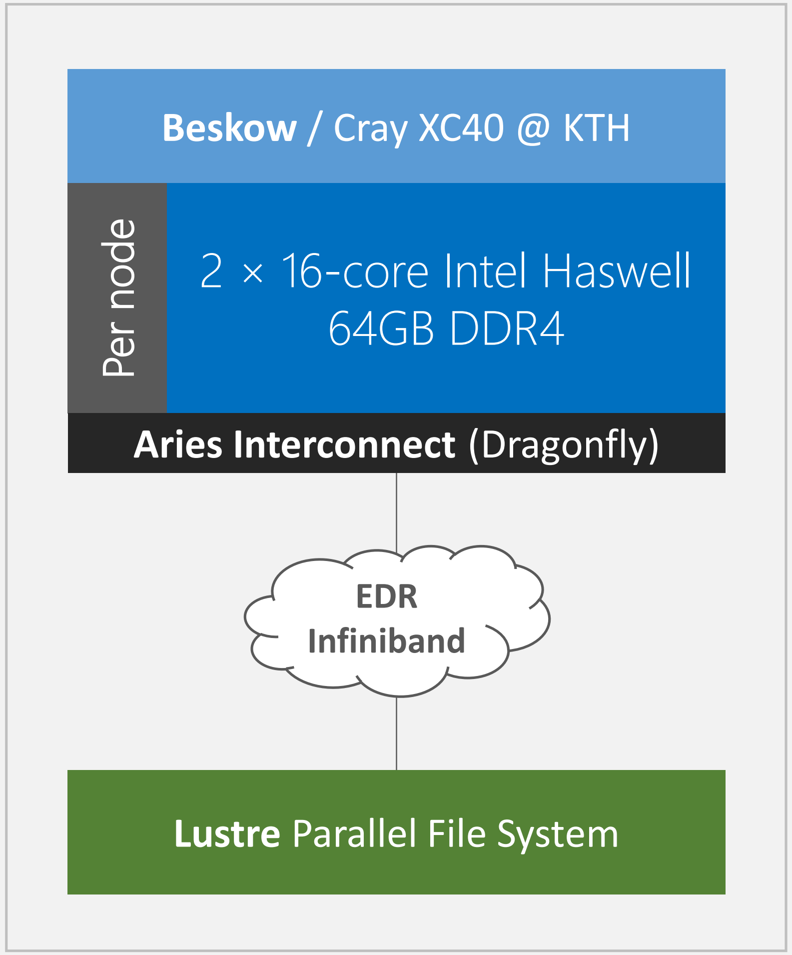
(b) **IOR Weak Scaling** (RND Kernel / Ind. BW)

Evaluation > Experimental setup

The second testbed is one of the two supercomputers at KTH, with storage provided by a Lustre parallel file system:

Specifications of Supercomputer "Beskow"	
Nodes	1676 × Haswell-based compute nodes
Processor	2 × 16-core Haswell E5-2698v3 @ 2.3GHz
Memory	64GB DRAM per node
Storage	Lustre PFS (Client v2.5.2) with 165 × OST Servers
Software	SUSE SLES v11 / CCE v8.3.4 and Cray-MPICH v7.0.4

We choose this cluster to run the larger-scale evaluations, but limiting the execution times and the amount of I/O operations.



Cray XC40 Supercomputer

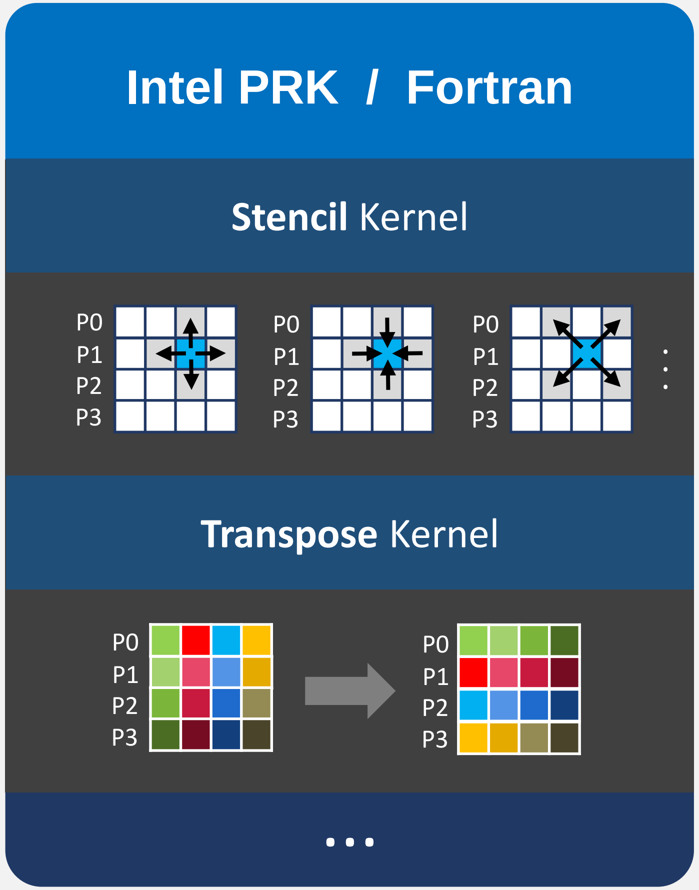
Evaluation > Intel PRK

We use Intel's Parallel Research Kernels (PRK) [5] on Beskow to evaluate some of the most common workloads in applications.

From the MPI-based kernels, we pick the most relevant ones:

- **Stencil.** Applies a data-parallel stencil operation to a two-dimensional array.
- **Transpose.** Stresses comm. & memory with regular, unit strided reads, and non-unit strided writes (and vice versa).
- ...

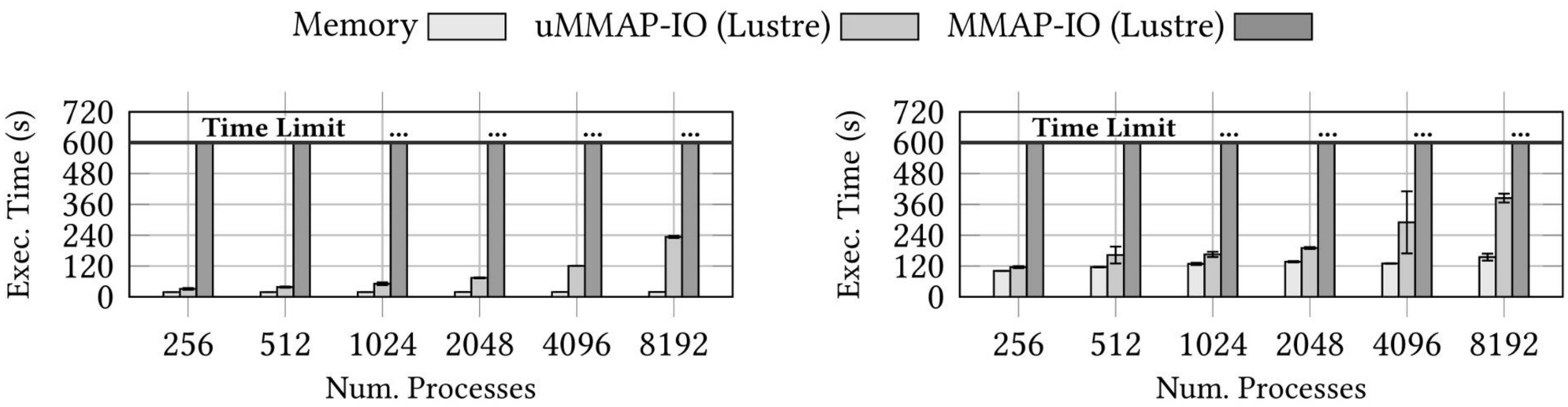
We run 100 iterations per kernel and enforce storage syncs. every 25 iterations to simulate application checkpoints.



[5] https://github.com/ParRes/Kernels

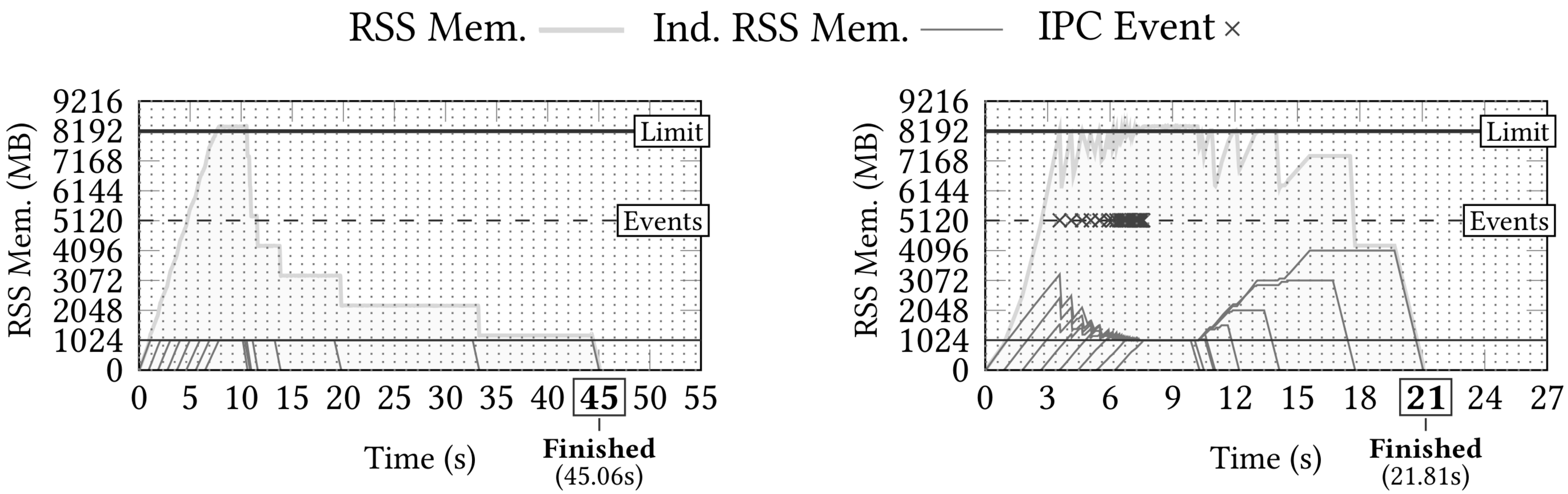
Evaluation > Intel PRK [Beskow]

The figure shows performance of `Stencil` and `Transpose` kernels utilizing a **fixed block dimension of 4096×4096 <u>per process</u>** (weak scaling). The results illustrate conventional mem. allocations, uMMAP-IO, and MMAP-IO:



(a) **PRK Stencil** (4K×4K block / 100 Iter.) (b) **PRK Transpose** (4K×4K block / 100 Iter.)

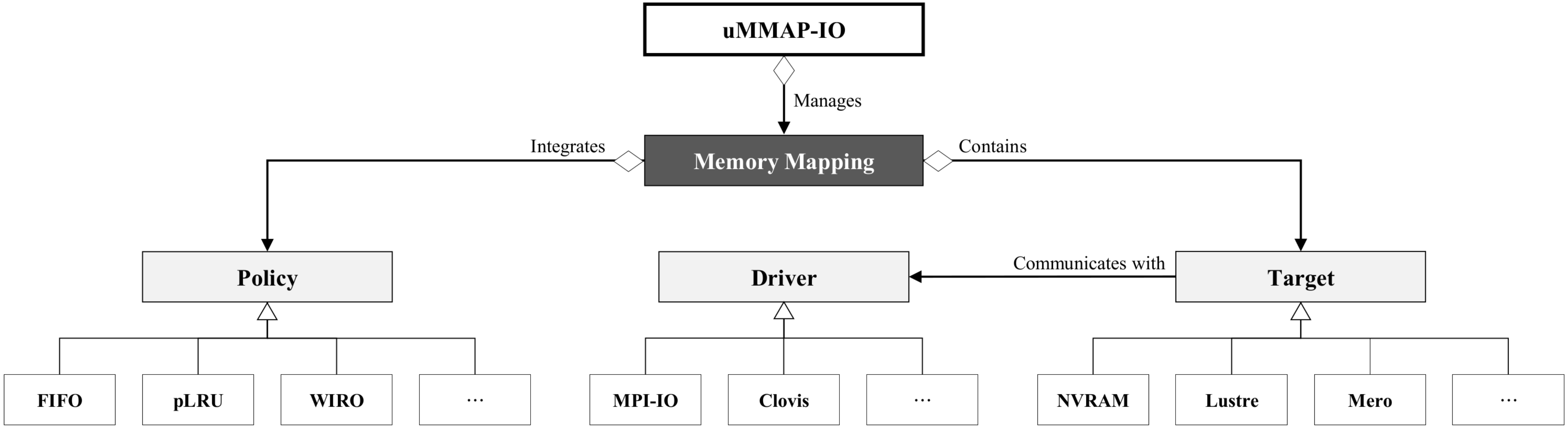
Evaluation > Maximum cache size handling



(a) **Static** Memory Management (b) **Dynamic** Memory Management

Future > various IO drivers

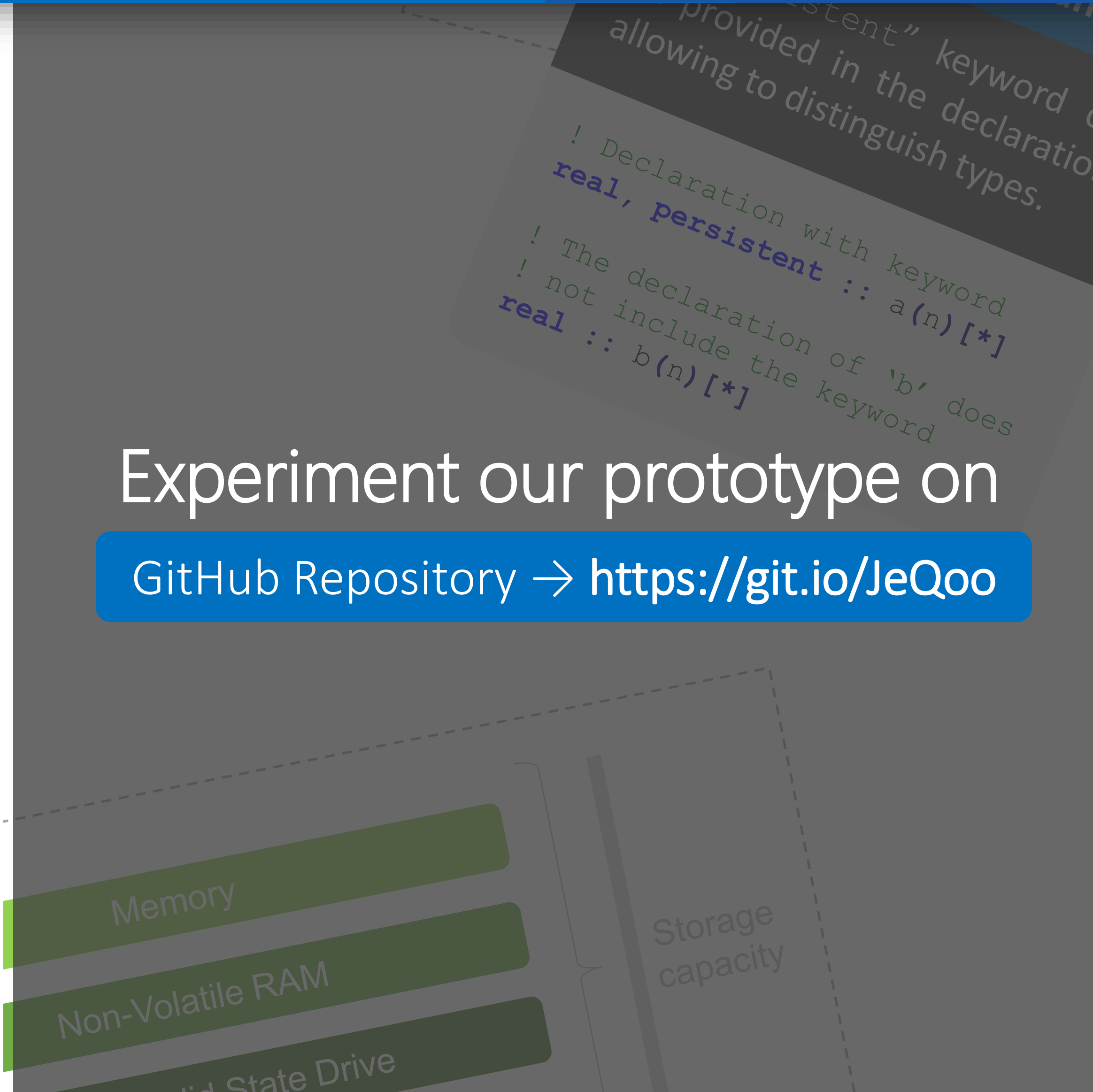
Now want to propose multiple driver abstraction not to limit to posix.



Conclusions

Memory mapping can be the common interface:

- **uMMAP-IO remove some limitations of OS memory mapping**
- **Can be used on any storage**
- **We even showed performance improvement**
- **In user space so can be easily extended and tuned**



Experiment our prototype on

GitHub Repository → **https://git.io/JeQoo**

European Commission | Horizon 2020 European Union funding for Research & Innovation

Acknowledgements

Part of this work is funded by the European Commission through the Sage2 Project

[Grant agreement no. 800999 | More information on http://www.sagestorage.eu]

Thank you for coming!

Questions?