New Challenges for Virtual-Memory Management

What are efficient OS abstractions for large memory objects?
New Challenges for Virtual-Memory Management

- Memory Subsystem
  - Designed for single CPU
  - Volatile state
  - Designed for MiB – GiB

What are efficient OS abstractions for large memory objects?
New Challenges for Virtual-Memory Management

Many processing elements

CPU
CPU
CPU

Memory Subsystem

- Designed for single CPU
- Volatile state
- Designed for MiB – GiB

Memory
New Challenges for Virtual-Memory Management

Many processing elements

CPU
CPU
CPU
Accelerator
RDMA

Memory Subsystem
- Designed for single CPU
- Volatile state
- Designed for MiB – GiB

Memory
New Challenges for Virtual-Memory Management

Many processing elements

- CPU
- CPU
- CPU
- Accelerator
- RDMA

Memory Subsystem

- Designed for single CPU
- Volatile state
- Designed for MiB - GiB

Memory
New Challenges for Virtual-Memory Management

- Many processing elements
  - CPU
  - CPU
  - CPU
  - Accelerator
  - RDMA

Memory Subsystem

- Large and diverse memory
  - DRAM
  - DRAM
  - DRAM

Designed for single CPU
Volatile state
Designed for MiB – GiB

What are efficient OS abstractions for large memory objects?
New Challenges for Virtual-Memory Management

Many processing elements
- CPU
- CPU
- CPU
- Accelerator
- RDMA

Designed for single CPU
Volatile state
Designed for MiB – GiB

Large and diverse memory
- DRAM
- DRAM
- DRAM

What are efficient OS abstractions for large memory objects?
New Challenges for Virtual-Memory Management

Many processing elements
- CPU
- CPU
- CPU
- Accelerator
- RDMA

Designed for single CPU
- Volatile state
- Designed for MiB – GiB

Large and diverse memory
- DRAM
- HBM
- Persistent

What are efficient OS abstractions for large memory objects?
New Challenges for Virtual-Memory Management

Many processing elements

CPU
CPU
CPU
Accelerator
RDMA

Memory Subsystem

Large and diverse memory

DRAM
DRAM
Persistent
HBM

Designed for single CPU
Volatile state
Designed for MiB – GiB
New Challenges for Virtual-Memory Management

Many processing elements
- CPU
- CPU
- CPU
- Accelerator
- RDMA

Large and diverse memory
- DRAM
- DRAM
- Persistent
- HBM

What are efficient OS abstractions for large memory objects?
New Challenges for Virtual-Memory Management

What are efficient OS abstractions for large memory objects?
Motivation

Goal: Efficient handling of virtual memory
Motivation

**Goal:** Efficient handling of virtual memory

Mapping $\sim7$ GiB of data into an application's address space
**Motivation**

**Goal:** Efficient handling of virtual memory

- Conventional file mapping (pre-faulted)
  - High setup costs
  - Create new page tables
  - Populate with pages from page cache

Mapping ~7 GiB of data into an application's address space.
Motivation

Goal: Efficient handling of virtual memory
- Conventional file mapping (pre-faulted)
  - High setup costs
  - Create new page tables
  - Populate with pages from page cache

Our solution: Morsels

Mapping ~7 GiB of data into an application's address space
Memory Objects - Morsel Concept
Memory Objects - Morsel Concept

- Base Pages
- Huge Pages
- Morsel
- Many individual pages
- Internal fragmentation
- Sparsely populated
- Fully self-contained
- Mounted by single entry
- First class OS object

Virtual Address Space
Physical Address Space
Memory Objects - Morsel Concept

Many individual pages

Internal fragmentation

Sparsely populated

Fully self-contained

Mounted by single entry

First class OS object

Physical Address Space

Virtual Address Space

Physical Address Space
Memory Objects - Morsel Concept

- Base Pages
- Huge Pages
- Morsel
- Many individual pages
- Internal fragmentation
- Sparsely populated
- Fully self-contained
- Mounted by single entry
- First class OS object
- Physical Address Space
- Virtual Address Space

halbuer@sra.uni-hannover.de  Morsels: Explicit Virtual Memory Objects
Memory Objects - Morsel Concept

- Base Pages
- Huge Pages
- Morsel
- Many individual pages
- Internal fragmentation
- Sparsely populated
- Fully self-contained
- Mounted by single entry
- First class OS object

Physical
Address Space
Virtual
Address Space

Many individual pages
Memory Objects - Morsel Concept

Base Pages
Huge Pages
Morsel
Many individual pages
Internal fragmentation
Sparsely populated
Fully self-contained
Mounted by single entry
First class OS object
Physical Address Space
Virtual Address Space

Many individual pages
Memory Objects - Morsel Concept

- Many individual pages
- Internal fragmentation
- Base Pages
- Huge Pages
- Virtual Address Space
- Physical Address Space
- First class OS object
- Fully self-contained
- Mounted by single entry
- Sparsely populated
- Huge Pages

halbuer@sra.uni-hannover.de
Morsels: Explicit Virtual Memory Objects
Memory Objects - Morsel Concept

- Base Pages
- Huge Pages
- Morsel

- Many individual pages
- Internal fragmentation

Virtual Address Space

Physical Address Space

Virtual Address Space

Many individual pages
Memory Objects - Morsel Concept

Many individual pages
Internal fragmentation
Sparsely populated

Base Pages

Huge Pages

Morsel

Virtual Address Space

Physical Address Space

Virtual Address Space

halbuer@sra.uni-hannover.de Morsels: Explicit Virtual Memory Objects
Memory Objects - Morsel Concept

- Base Pages
- Huge Pages
- Morsel

Many individual pages

Internal fragmentation

Sparsely populated

Fully self-contained

Virtual Address Space

Physical Address Space

Virtual Address Space

halbuer@sra.uni-hannover.de  Morsels: Explicit Virtual Memory Objects
Memory Objects - Morsel Concept

- Base Pages
- Huge Pages
- Morsel

- Many individual pages
- Internal fragmentation
- Sparsely populated

- Mounted by single entry
- Fully self-contained

Virtual Address Space

halbuer@sra.uni-hannover.de  Morsels: Explicit Virtual Memory Objects
Memory Objects - Morsel Concept

- Base Pages
- Huge Pages
- Morsel
- Many individual pages
- Internal fragmentation
- Fully self-contained
- Mounted by single entry
- Sparsely populated
- First class OS object
- Virtual Address Space

Physical Address Space

Virtual Address Space

halbuer@sra.uni-hannover.de  Morsels: Explicit Virtual Memory Objects
Memory Objects - Morsel Concept

- Good multi-core scalability
- No locking required
- Persistent objects on NVM
- Fully self-contained design
- Crash consistent implementation
- Huge-page support
- Increase TLB coverage
- Transparency on object granularity
- Sharing with devices
- Reduce management overhead for DMA

Morsel

IOMMU
Accelerator
SSD
NIC
Atomic
operations
Object-specific
transparent
huge pages
Direct sharing
with devices
Memory Objects - Morsel Concept

- Good multi-core scalability
  - No locking required
Memory Objects - Morsel Concept

- Good multi-core scalability
  - No locking required

- Persistent objects on NVM
  - Fully self-contained design
  - Crash consistent implementation
Memory Objects - Morsel Concept

- Good multi-core scalability
  - No locking required

- Persistent objects on NVM
  - Fully self-contained design
  - Crash consistent implementation

- Huge-page support
  - Increase TLB coverage
  - Transparency on object granularity

Atomic operations

Object-specific transparent huge pages

Morsel

IOMMU

Accelerator

SSD

NIC

Atomic

operations

Direct sharing
with devices

halbuer@sra.uni-hannover.de  Morsels: Explicit Virtual Memory Objects
Memory Objects - Morsel Concept

- Good multi-core scalability
  - No locking required
- Persistent objects on NVM
  - Fully self-contained design
  - Crash consistent implementation
- Huge-page support
  - Increase TLB coverage
  - Transparency on object granularity
- Sharing with devices
  - Reduce management overhead for DMA

---

[Diagram showing Morsel interactions with IOMMU, SSD, Accelerator, NIC, Atomic operations, and direct sharing with devices.]
Case Study 1: User-Space Read-Only File Cache

**Evaluation:** Prototypic implementation in Linux
Case Study 1: User-Space Read-Only File Cache

**Evaluation:** Prototypic implementation in Linux

**Task:** Inference with a large language model
Case Study 1: User-Space Read-Only File Cache

**Evaluation:** Prototypic implementation in Linux

**Task:** Inference with a large language model

- Conventional file mapping (pre-faulted)
  - Uses individual 4-KiB pages
  - Populated from the page cache
  - Time increases with mapping size
Case Study 1: User-Space Read-Only File Cache

Evaluation: Prototypic implementation in Linux

Task: Inference with a large language model

- Conventional file mapping (pre-faulted)
  - Uses individual 4-KiB pages
  - Populated from the page cache
  - Time increases with mapping size

llama.cpp with SelFee model: 6.82 GiB

<table>
<thead>
<tr>
<th>OS Page Cache + Shared File Mapping</th>
<th>MMap Model</th>
<th>Initialize Model</th>
<th>Inference (per Token)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Time [ms]</td>
<td>232</td>
<td>293</td>
<td>265</td>
</tr>
</tbody>
</table>

Morsel mapping
- Cache provides model data as Morsel
- Single indivisible unit
- Mapped in constant time

Result: 45% reduced startup time

Morsels: Explicit Virtual Memory Objects

halbuer@sra.uni-hannover.de
Case Study 1: User-Space Read-Only File Cache

Evaluation: Prototypic implementation in Linux

Task: Inference with a large language model

- Conventional file mapping (pre-faulted)
  - Uses individual 4-KiB pages
  - Populated from the page cache
  - Time increases with mapping size

- Morsel mapping
  - Cache provides model data as Morsel
  - Single indivisible unit
  - Mapped in constant time

llama.cpp with SelFee model: 6.82 GiB

<table>
<thead>
<tr>
<th>OS Page Cache</th>
<th>MMap Model</th>
<th>Initialize Model</th>
<th>Inference (per Token)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Time [ms]</td>
<td>232</td>
<td>293</td>
<td>265</td>
</tr>
</tbody>
</table>

Morsel mapping provides a 45% reduced startup time compared to conventional mapping.

Morsels: Explicit Virtual Memory Objects
Case Study 1: User-Space Read-Only File Cache

**Evaluation:** Prototypic implementation in Linux

**Task:** Inference with a large language model

- Conventional file mapping (pre-faulted)
  - Uses individual 4-KiB pages
  - Populated from the page cache
  - Time increases with mapping size

- Morsel mapping
  - Cache provides model data as Morsel
  - Single indivisible unit
  - Mapped in constant time

---

**llama.cpp** with SelFee model: 6.82 GiB

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Run Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMap Model</td>
<td>232, 293, 265</td>
</tr>
<tr>
<td>Initialize Model</td>
<td>288, 261</td>
</tr>
<tr>
<td>Inference (per Token)</td>
<td>293, 261</td>
</tr>
</tbody>
</table>

**Graph:**
- **OS Page Cache + Shared File Mapping**
- **Morsel Cache**

**Result:** 45% reduced startup time
Case Study 1: User-Space Read-Only File Cache

**Evaluation:** Prototypic implementation in Linux

**Task:** Inference with a large language model

- Conventional file mapping (pre-faulted)
  - Uses individual 4-KiB pages
  - Populated from the page cache
  - Time increases with mapping size

- Morsel mapping
  - Cache provides model data as Morsel
  - Single indivisible unit
  - Mapped in constant time

---

**llama.cpp with SelFee model: 6.82 GiB**

- **MMap Model**
- **Initialize Model**
- **Inference (per Token)**

---

Run Time [ms]

- OS Page Cache + Shared File Mapping: 232, 293, 265
- Morsel Cache: 288, 261

**Result:** 45% reduced startup time

---

**Unaffected**

---

halbuer@sra.uni-hannover.de  Morsels: Explicit Virtual Memory Objects
Case Study 1: User-Space Read-Only File Cache

**Evaluation:** Prototypic implementation in Linux

**Task:** Inference with a large language model

- Conventional file mapping (pre-faulted)
  - Uses individual 4-KiB pages
  - Populated from the page cache
  - Time increases with mapping size

- Morsel mapping
  - Cache provides model data as Morsel
  - Single indivisible unit
  - Mapped in constant time

**Result:** 45% reduced startup time

```plaintext
llama.cpp with SelFee model: 6.82 GiB

<table>
<thead>
<tr>
<th></th>
<th>MMap Model</th>
<th>Initialize Model</th>
<th>Inference (per Token)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS Page Cache + Shared File Mapping</td>
<td>232</td>
<td>293</td>
<td>265</td>
</tr>
<tr>
<td>Morsel Cache</td>
<td>288</td>
<td>261</td>
<td>0.371</td>
</tr>
</tbody>
</table>
```

Run Time [ms]
Case Study 2: Processing Pipeline

Comparing Morsels with memfd

1,000 packets per type

Up to 256 MiB packet size

Morsels: 39% reduced round-trip time on average

Use-space time draws lower bound

Waiting time

Kernel processing

= Round-trip time

Conclusion: Morsels enable efficient handling of virtual memory!

Unix Domain Socket: Passing File Descriptors

<table>
<thead>
<tr>
<th>Packet Size [MiB]</th>
<th>Round-Trip Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>100</td>
</tr>
<tr>
<td>128</td>
<td>200</td>
</tr>
<tr>
<td>192</td>
<td>300</td>
</tr>
<tr>
<td>256</td>
<td></td>
</tr>
</tbody>
</table>

memfd

Morsel

Locally-Weighted Reg.

Round-Trip Time

halbuer@sra.uni-hannover.de Morsels: Explicit Virtual Memory Objects
Case Study 2: Processing Pipeline

- Comparing Morsels with memfd

**Unix Domain Socket: Passing File Descriptors**

1. Buffer Provider
2. (2)
3. (3)
4. (4)

**memfd/Morsel**

<table>
<thead>
<tr>
<th>Packet Size [MiB]</th>
<th>Round-Trip Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>100</td>
</tr>
<tr>
<td>128</td>
<td>200</td>
</tr>
<tr>
<td>192</td>
<td>300</td>
</tr>
<tr>
<td>256</td>
<td></td>
</tr>
</tbody>
</table>

*Locally-Weighted Reg.*

**Memfd:**

- Write Access: 0
- Read Access: 64
- Write Access: 128
- Write Access: 192
- Write Access: 256

**Morsel:**

- Write Access: 0
- Write Access: 64
- Write Access: 128
- Write Access: 192
- Write Access: 256
Case Study 2: Processing Pipeline

- Comparing Morsels with memfd

Unix Domain Socket: Passing File Descriptors

(1) Buffer Provider
(2) Data Source
(3)
(4)

memfd/Morsel
Write Access

Packet Size \([\text{MiB}]\)

Round-Trip Time \([\text{ms}]\)

Locally-Weighted Reg.

Round-Trip Time

0 64 128 192 256
0
100
200
300
memfd

0 64 128 192 256
Morsel
Case Study 2: Processing Pipeline

- Comparing Morsels with memfd

Unix Domain Socket: Passing File Descriptors

(1) Buffer Provider
(2) Data Source
(3) Checksum

memfd/Morsel Write Access Read Access Write Access
Case Study 2: Processing Pipeline

- Comparing Morsels with memfd

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>memfd/Morsel</td>
</tr>
<tr>
<td>64</td>
<td>100</td>
<td>Morsel</td>
</tr>
<tr>
<td>128</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unix Domain Socket: Passing File Descriptors

(1) Buffer Provider
(2) Data Source
(3) Checksum
(4) Sender

memfd/Morsel Write Access
Read Access Write Access

halbuer@sra.uni-hannover.de Morsels: Explicit Virtual Memory Objects
Case Study 2: Processing Pipeline

- Comparing Morsels with memfd
  - 1,000 packets per type
  - Up to 256 MiB packet size

Unix Domain Socket: Passing File Descriptors

1. Buffer Provider
2. Data Source
3. Checksum
4. Sender

memfd/Morsel
Write Access
Read Access
Write Access

Packet Size [MiB]
Round-Trip Time [ms]
Locally-Weighted Reg.
Round-Trip Time

0 64 128 192 256
0
100
200
300
memfd
0 64 128 192 256
Morsel

halbuer@sra.uni-hannover.de Morsels: Explicit Virtual Memory Objects 7 – 8
Case Study 2: Processing Pipeline

- Comparing Morsels with memfd
  - 1,000 packets per type
  - Up to 256 MiB packet size

### Unix Domain Socket: Passing File Descriptors

1. **Buffer Provider**
2. **Data Source**
3. **Checksum**
4. **Sender**

**memfd/Morsel**
- **Write Access**
- **Read Access**
- **Write Access**

### Packet Size vs. Round-Trip Time

- **Locally-Weighted Reg.**
- **Round-Trip Time**

<table>
<thead>
<tr>
<th>Packet Size [MiB]</th>
<th>Round-Trip Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>memfd</td>
</tr>
<tr>
<td>64</td>
<td>Morsel</td>
</tr>
<tr>
<td>128</td>
<td></td>
</tr>
<tr>
<td>192</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
</tr>
</tbody>
</table>

Graphical representation of packet size vs. round-trip time with Locally-Weighted Reg. regression line.
Case Study 2: Processing Pipeline

- Comparing Morsels with memfd
  - 1,000 packets per type
  - Up to 256 MiB packet size

Unix Domain Socket: Passing File Descriptors

<table>
<thead>
<tr>
<th>Packet Size [MiB]</th>
<th>memfd</th>
<th>Morsel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>128</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>192</td>
<td>300</td>
<td>400</td>
</tr>
<tr>
<td>256</td>
<td>400</td>
<td>500</td>
</tr>
</tbody>
</table>

Locally-Weighted Reg. Round-Trip Time

memfd/Morsel Write Access Read Access Write Access

memfd

Morsel

Packet Size [MiB]

Round-Trip Time [ms]

Memfd: 39% reduced round-trip time on average

Use-space time draws lower bound

Waiting time

Kernel processing

= Round-trip time

Conclusion: Morsels enable efficient handling of virtual memory!
Case Study 2: Processing Pipeline

- Comparing Morsels with memfd
  - 1,000 packets per type
  - Up to 256 MiB packet size
- Morsels: 39% reduced round-trip time on average

![Diagram showing Unix Domain Socket: Passing File Descriptors](image)

<table>
<thead>
<tr>
<th>Packet Size [MiB]</th>
<th>Round-Trip Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>memfd</td>
<td></td>
</tr>
<tr>
<td>Morsel</td>
<td></td>
</tr>
</tbody>
</table>

Locally-Weighted Reg. - Round-Trip Time

- memfd/Morsel
- Write Access
- Read Access
- Write Access

halbuer@sra.uni-hannover.de Morsels: Explicit Virtual Memory Objects
Case Study 2: Processing Pipeline

- Comparing Morsels with memfd
  - 1000 packets per type
  - Up to 256 MiB packet size

- Morsels: 39% reduced round-trip time on average
Case Study 2: Processing Pipeline

- Comparing Morsels with memfd
  - 1,000 packets per type
  - Up to 256 MiB packet size

- Morsels: 39% reduced round-trip time on average

- Use-space time draws lower bound
  - Waiting time
  - Kernel processing
  = Round-trip time

Unix Domain Socket: Passing File Descriptors

<table>
<thead>
<tr>
<th>Packet Size [MiB]</th>
<th>Round-Trip Time [ms]</th>
<th>User-Space Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>128</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>192</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>256</td>
<td>400</td>
<td>400</td>
</tr>
</tbody>
</table>

Locally-Weighted Reg.

memfd
- Write Access
- Read Access
- Write Access

Morsel
- Write Access
- Read Access
- Write Access

halbuer@sra.uni-hannover.de

Morsels: Explicit Virtual Memory Objects
Case Study 2: Processing Pipeline

- Comparing Morsels with memfd
  - 1 000 packets per type
  - Up to 256 MiB packet size

- Morsels: 39% reduced round-trip time on average

- Use-space time draws lower bound
  + Waiting time
  + Kernel processing
  = Round-trip time

**Conclusion:** Morsels enable efficient handling of virtual memory!
Morsels: Highly efficient handling of large memory objects

**Problem:** Conventional memory management

- Many individual entities (pages)
- Huge pages induce fragmentation
- No persistent state

**Solution:**

- Indivisible unit (page-table subtree)
- Ready for persistent memory
- Efficiently sharable

**Results**

- Constant mapping time (x1000 faster for 7 GiB)
- 39% reduced RTT for shown pipeline
Problem: Conventional memory management
- Many individual entities (pages)
- Huge pages induce fragmentation
- No persistent state

Solution: Morsels
- Indivisible unit (page-table subtree)
- Ready for persistent memory
- Efficiently sharable
Morsels: Highly efficient handling of large memory objects

- **Problem:** Conventional memory management
  - Many individual entities (pages)
  - Huge pages induce fragmentation
  - No persistent state

- **Solution:** Morsels
  - Indivisible unit (page-table subtree)
  - Ready for persistent memory
  - Efficiently sharable

- **Results**
  - Constant mapping time (x1000 faster for 7 GiB)
  - 39% reduced RTT for shown pipeline
Case Study 2: Processing Pipeline

Run-Time (Normalized to memfd) [%]

Component
- Pagefault
- Munmap
- proc:source
- proc:checksum
- proc:sender

Initial allocation 41% reduced runtime

Conclusion: Morsels enable efficient handling of virtual memory!
Case Study 2: Processing Pipeline

Run-Time (Normalized to memfd) [%]

Component
- Pagefault
- Munmap
- proc:source
- proc:checksum
- proc:sender

Write
write

Initial allocation 41% reduced runtime

Conclusion: Morsels enable efficient handling of virtual memory!
Case Study 2: Processing Pipeline

Run-Time (Normalized to memfd) [%]

Component:
- Pagefault
- Munmap
- proc:source
- proc:checksum
- proc:sender

Initial allocation is reduced by 41%.

Conclusion: Morsels enable efficient handling of virtual memory!
Case Study 2: Processing Pipeline

Component
- Pagefault
- Munmap
- proc:source
- proc:checksum
- proc:sender

- Write
- Read
- Write

Initial allocation 41% reduced runtime

Conclusion: Morsels enable efficient handling of virtual memory!
Case Study 2: Processing Pipeline

Run-Time (Normalized to memfd) [%]

Call Stack
- sender
- asm_exc_page..
- exc_p..
- do_
- checksum
- datasource
- asm_exc_pag..
- exc_p..
- do_u_
- memfd

Morsel
- Initial allocation

Component
- Pagefault
- Munmap
- proc:source
- proc:checksum
- proc:sender

Conclusion: Morsels enable efficient handling of virtual memory!
Case Study 2: Processing Pipeline

![Diagram of memory allocation and runtime comparison with Morsels]

**Component**
- Pagefault
- Munmap
- proc:source
- proc:checksum
- proc:sender

**Run-Time (Normalized to memfd) [%]**

- **Initial allocation** to memfd
- **41% reduced runtime**

**Conclusion:** Morsels enable efficient handling of virtual memory!
Conclusion: Morsels enable efficient handling of virtual memory!