



Linux Page Migration

Accelerating with Multi-threading and DMA Offload

Open Source Summit 2025, India

Shivank Garg
Linux Kernel Developer, AMD

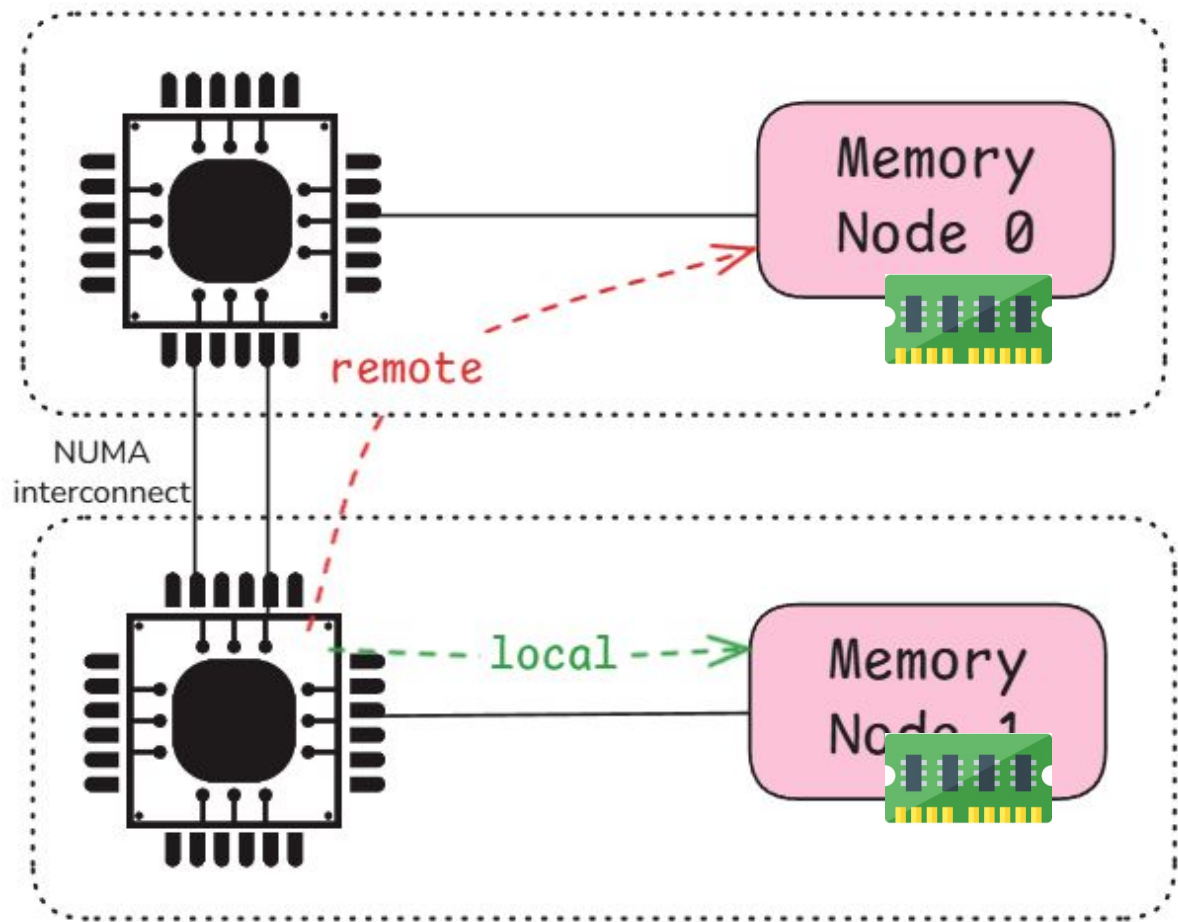
AMD 
together we advance_

Agenda

- Background
 - Page Migration - what, why and how?
 - Limitations of current Implementation
- Proposed Enhancements
 - Batching folio copy operations
 - Multi-threaded page copying
 - Hardware offloading
- Results
- Future work

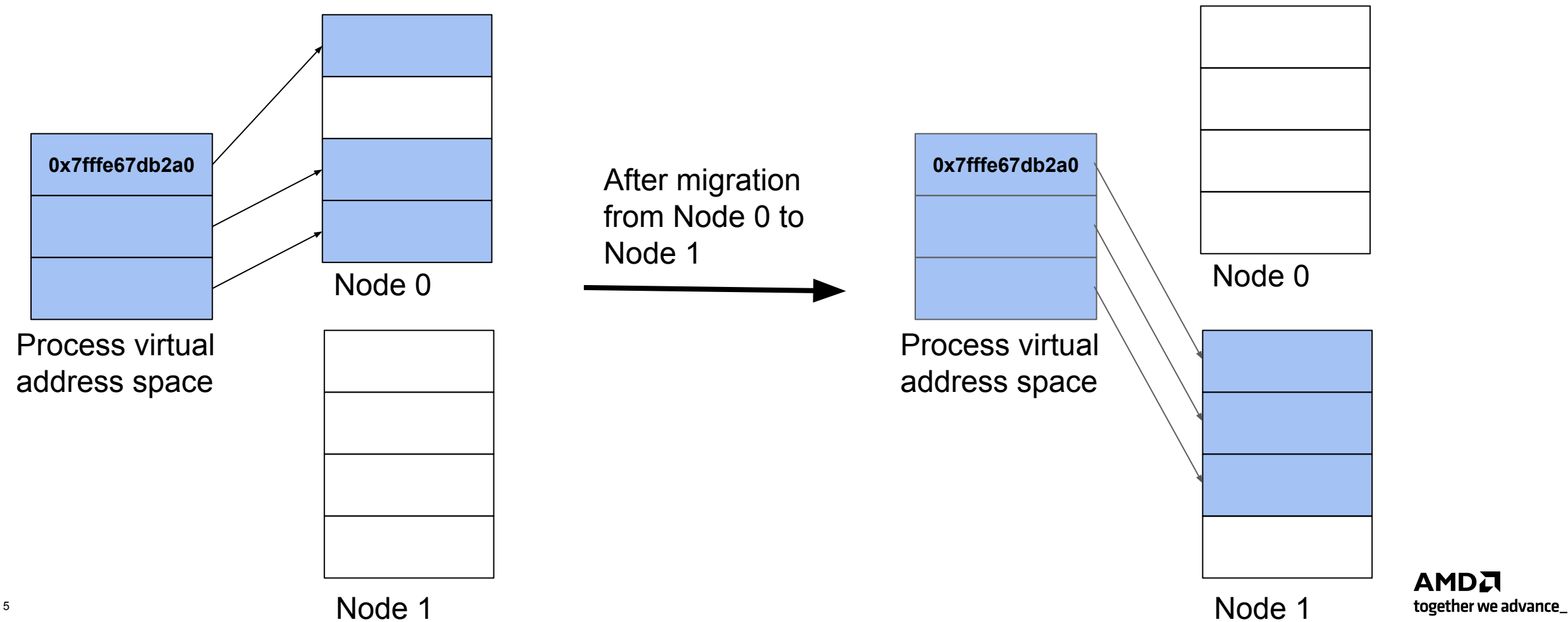
Background

Modern memory systems

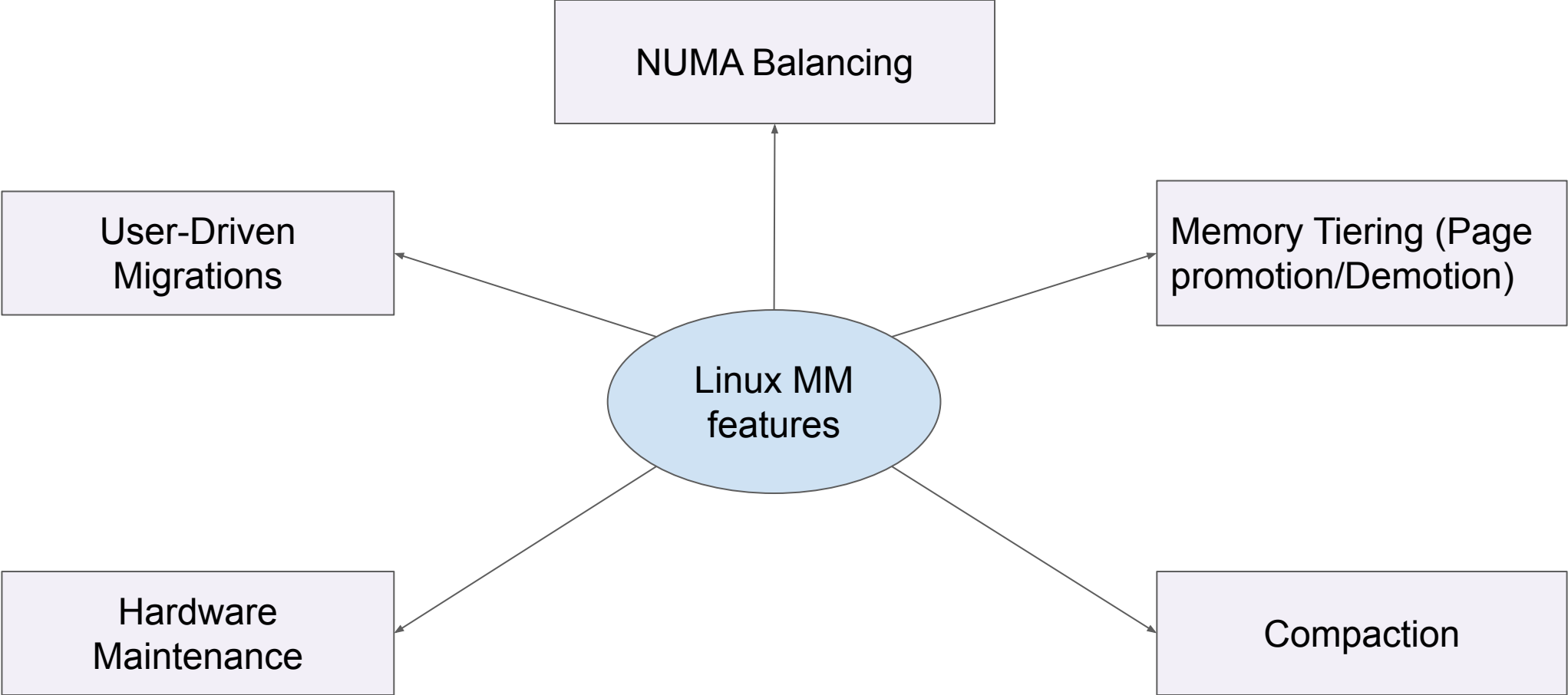


Page Migration - what?

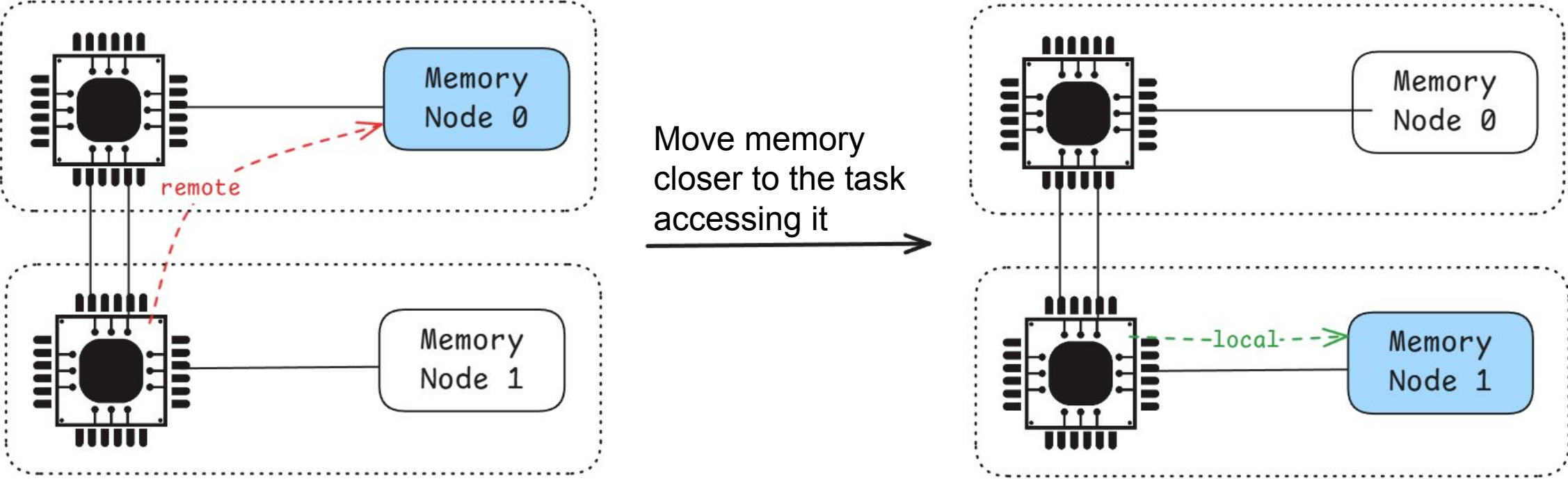
- Physical movement of memory pages to different region/node while a process is active.
- Virtual addresses used by the process remain unchanged.



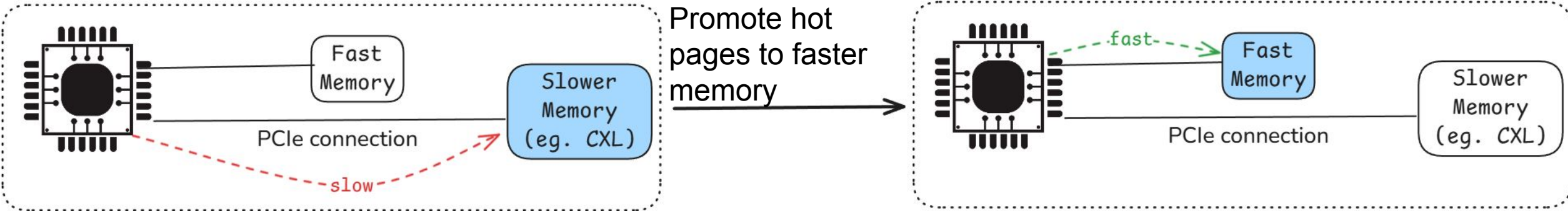
Page Migration - why?



Page Migration - NUMA balancing

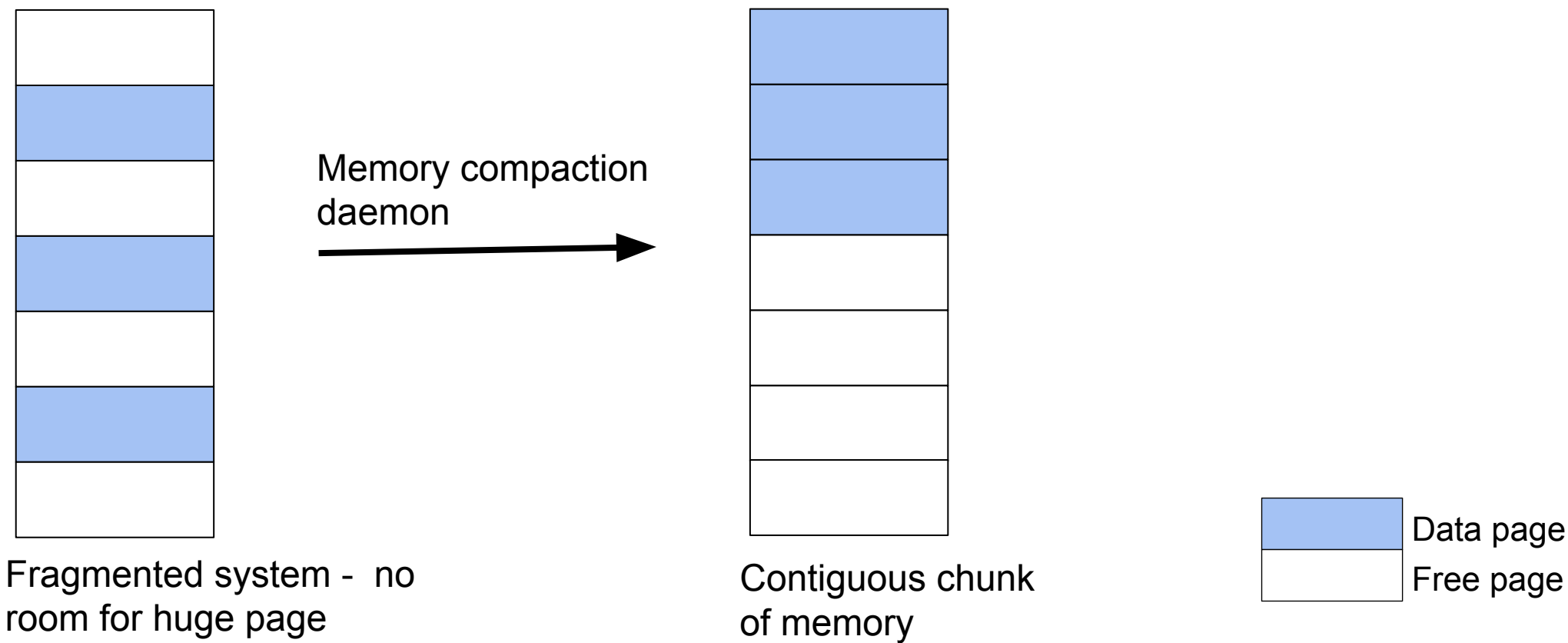


Page Migration - Memory Tiering



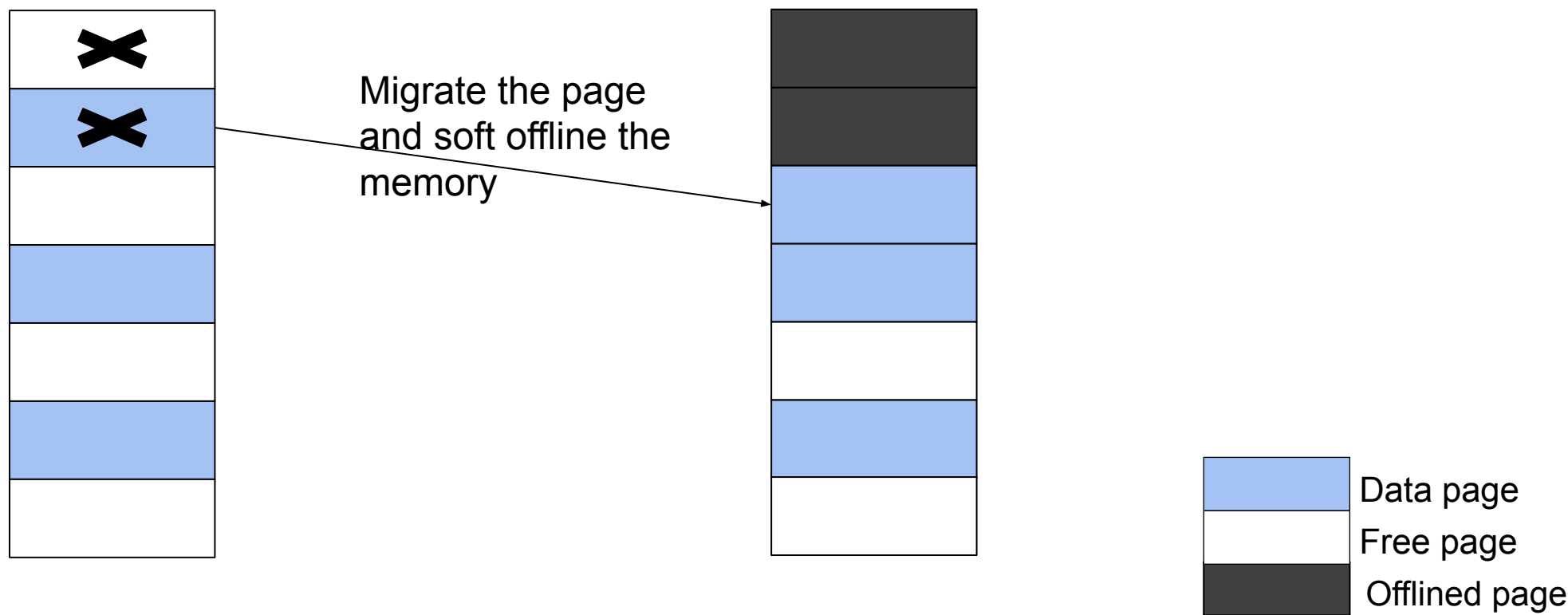
Page Migration - Compaction

Addresses fragmentation issue in the system to make space for higher-order allocations



Page Migration - Hardware Maintenance

Address possibility of memory failure (but the page isn't corrupted yet)



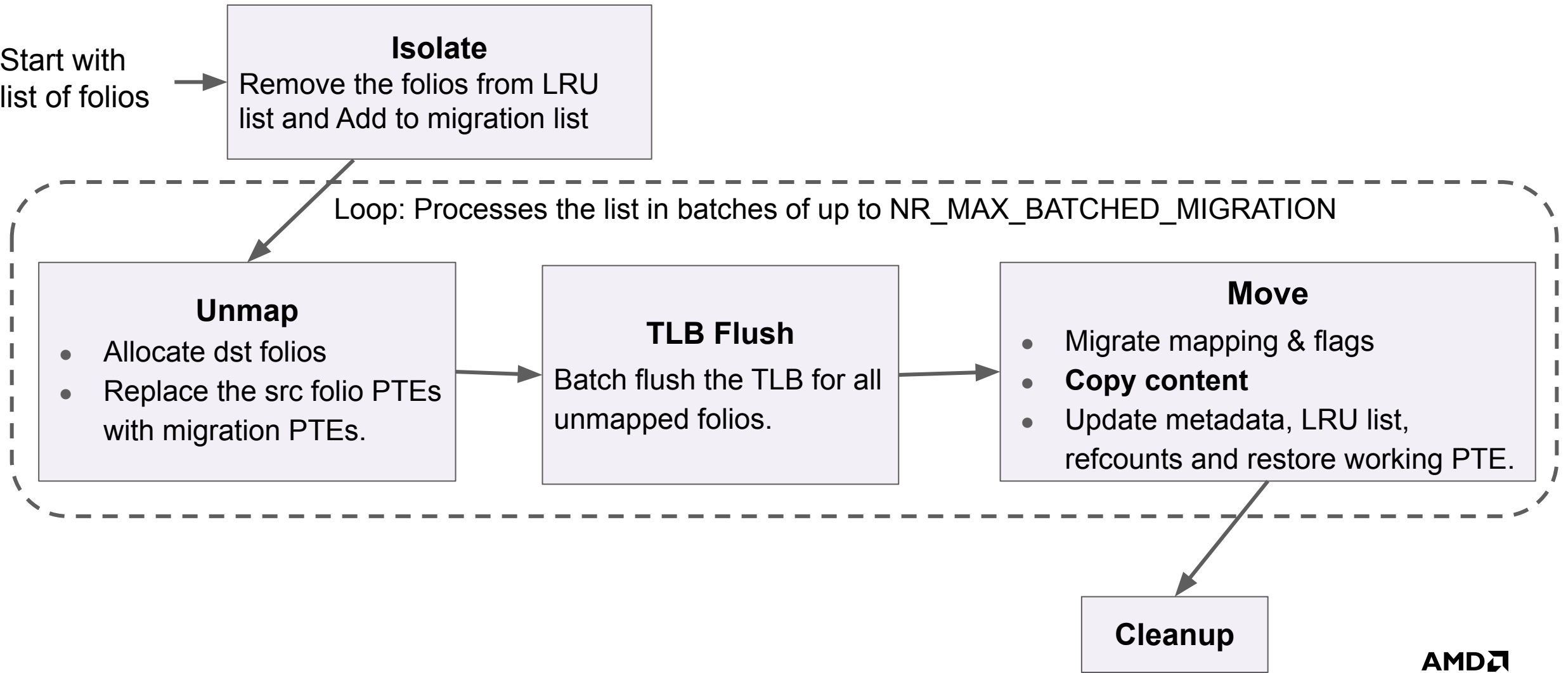
Page Migration - User-driven migrations

Allows user/tools/workloads to control memory placements from userspace to optimize memory placements in NUMA systems.

Using system calls like:

- `move_pages()`
- `migrate_pages()`
- `mbind()`

Page Migration - How?



Migration Bottleneck Problem and Proposed Solution

The Memory Migration Bottleneck

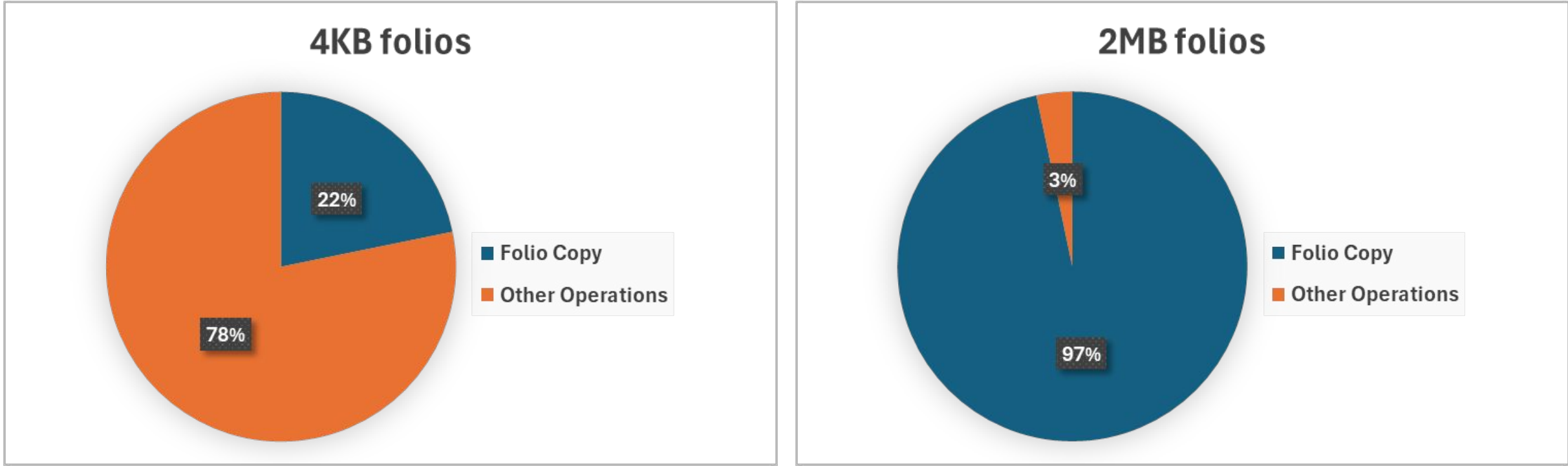
Problem Statement:

High page migration costs incurred in:

- Large scale tiered memory systems
- Frequent page promotion/demotion
- CPU-GPU coherent system with GPU memory exposed as NUMA nodes

The Memory Migration Bottleneck

Percentage of time spent in move_pages() syscall:



Folios copy overheads

The Memory Migration Bottleneck

- Modern systems are equipped with powerful hardware accelerators and high CPU core counts.
- Current memory migration implementation is single-threaded CPU copy
 - Memory bandwidth underutilized
 - Idle CPU cores are not utilized
 - Hardware acceleration potential remains untapped

Observations: The Move Towards Batching

Batch migration operations



Batch TLB flush operations



Batch folio content copying

⇒ logical next step

Optimization #1 – Batching folio_copy() operation

Current Design:

```
for_each_folio() {  
    Copy folio metadata like flags and mapping  
  
    Copy the folio content from src to dst  
  
    Update PTEs with new mappings  
}
```

Serialized copy: memory bandwidth and modern hardware underutilized

Proposed Design:

```
for_each_folio() {  
    Copy folio metadata like flags and mapping  
}  
  
Batch_copy_folio_content(dst_list, src_list)  
  
for_each_folio() {  
    Update PTEs with new mappings  
}
```

Batched Copy: Makes it easier for parallelization and hardware offloading

Optimization #2 – Multithreaded copying with CPU

Design

- Distribute a folio batch copy operations across multiple CPU threads
- Workqueues are used to copy the data to target node.
- Large folios (e.g., 2MB) get more copy throughput boost

Challenges

- Resource Accounting and Fairness: How to attribute migration costs (CPU time) correctly?

Optimization #3 – Hardware offloading

Design:

- Dynamically allocate DMA channels
- For each folio pair: map src/dst pages, prepare DMA descriptor with completion callback.
- Submit all transfers across channels in round-robin fashion, issue pending requests, then wait for completion callbacks
- Fall back to CPU to ensure successful copying, incase DMA returns error

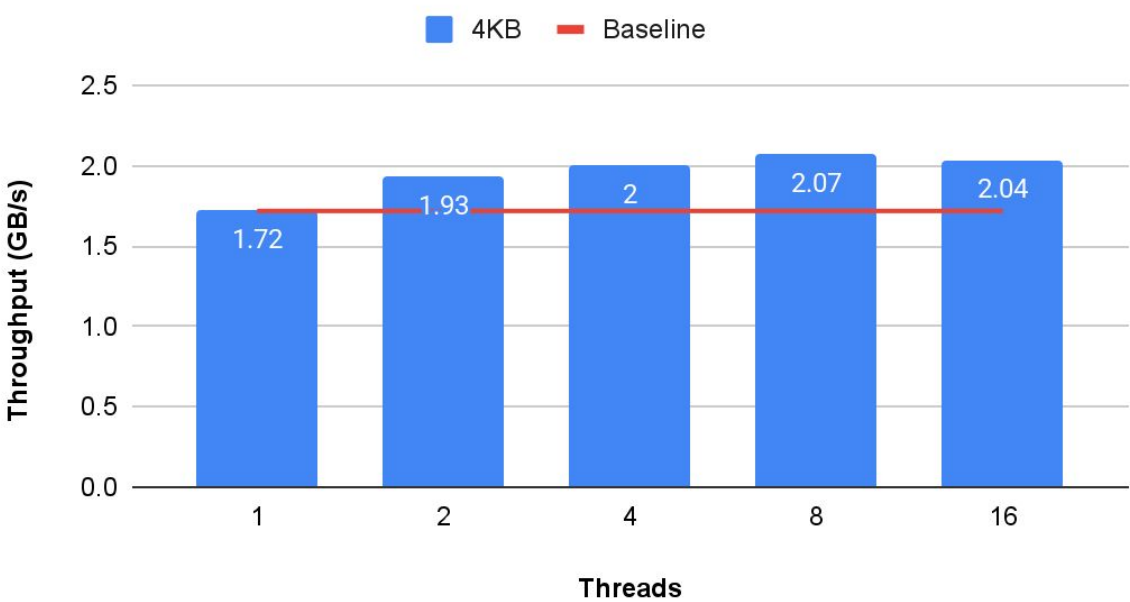
Challenges:

- Balancing DMA power and its high setup costs

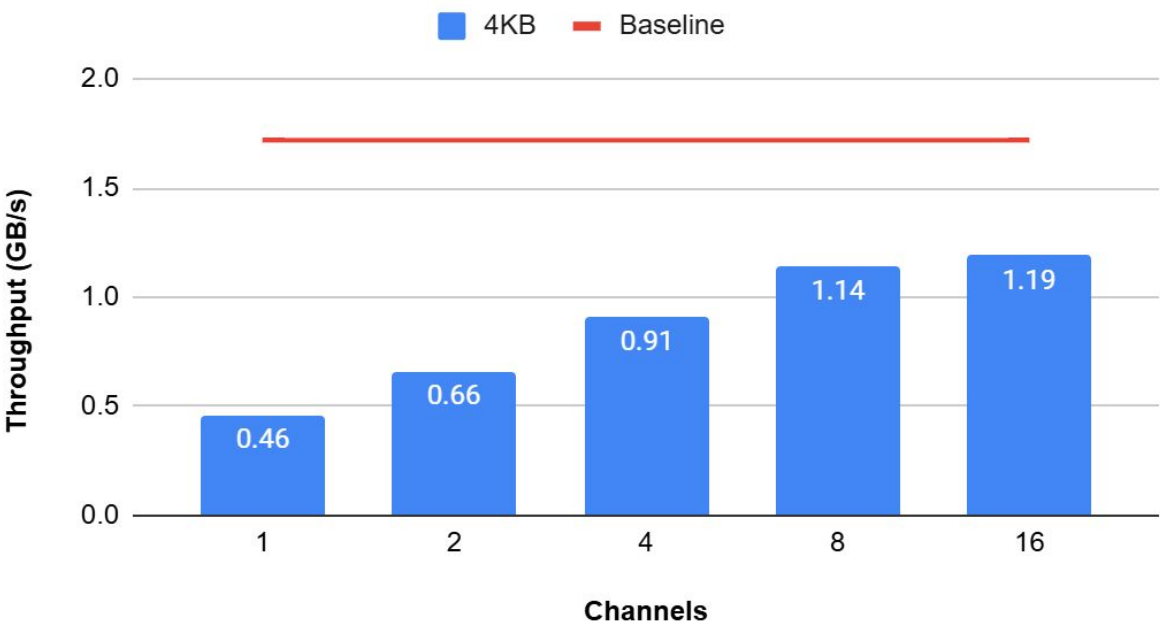
Results

Benchmarking move_pages(2) syscall throughput

4KB page (Multi-threaded CPU)



4KB page (DMA Offloading)

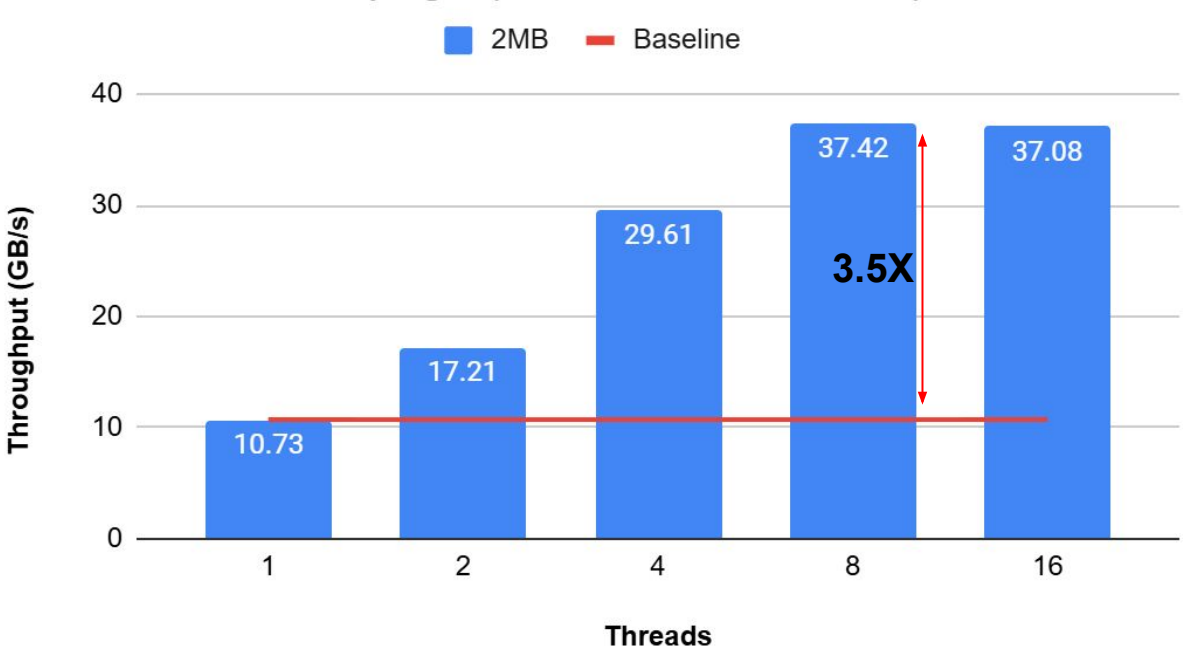


Multi-threaded CPU copying did not show much improvements while the performance degrades for DMA offload copying.

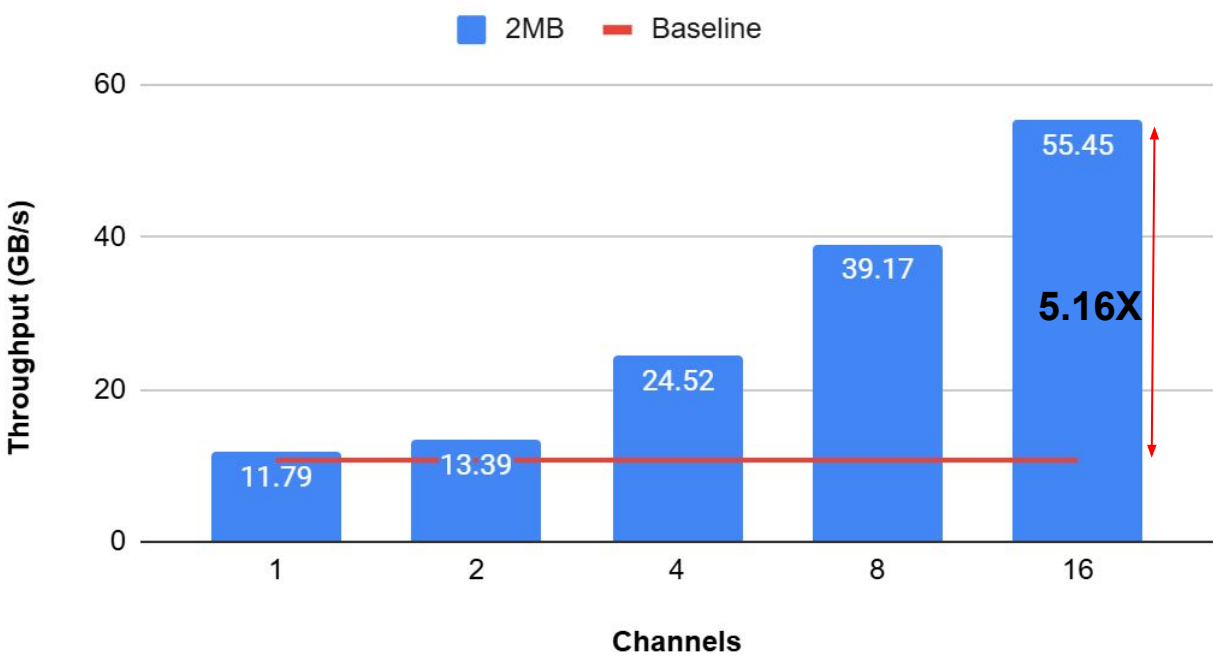
Results

Benchmarking move_pages(2) syscall throughput

2MB page (Multi-threaded CPU)



2MB page (DMA Offloading)



Baseline: Native single-threaded CPU copying.

Open Questions

1. Policy Decisions for:
 - a. DMA and CPU selections
 - b. Platform-specific scheduling of folio-copy worker threads for better bandwidth utilization
 - c. Using Non-temporal instructions for CPU-based memcpy
 - d. Upscaling/downscaling worker threads based on various factors
2. Interface requirements with offload copy APIs:
 - a. Standardizing offload copy APIs and support for different migrators
 - b. DMA mapping lifetime and overheads
3. Resources Accounting:
 - a. CPU cgroups accounting and fairness
 - b. Who bears migration cost? (Migration cost attribution)

Future Work

- Test DMA offloading performance on SDXi (Smart data stream accelerator)
- Address open questions
- Work towards upstreaming

Links and References

- Enhancements to Page Migration with Multi-threading and Batch Offloading to DMA:
<https://lore.kernel.org/linux-mm/20250319192211.10092-1-shivankg@amd.com>
- Branch: https://github.com/shivankgarg98/linux/commits/shivank/migrate_pages_optimization
- Accelerate Page migration with batching and Multi-threading:
<https://lore.kernel.org/linux-mm/20250103172419.4148674-1-ziy@nvidia.com>

Thank You

Backup

Set-up Information

AMD Zen 3 EPYC server (2 socket, 32 cores per socket, SMT Enabled)

1 NUMA node per socket, Linux Kernel 6.14.0-rc7, DVFS set to Performance,

DMA hardware: **PTDMA**

Folio copy overheads in move_page syscall in base kernel

Linux: **6.14.0-rc7 (generic)**

Test: move_pages 1GB folios (4KB and 2MB)

4KB folios

```
- 57.61% syscall
- 57.34% entry_SYSCALL_64_after_hwframe
- do_syscall_64
- 57.34% __x64_sys_move_pages
- kernel_move_pages
- 53.17% do_pages_move
- 45.01% move_pages_and_store_status
- 44.82% migrate_pages
- 44.70% migrate_pages_sync
- migrate_pages_batch
- 28.86% migrate_folio_move
- 13.90% move_to_new_folio
- 12.16% folio_mc_copy
- 11.79% copy_mc_to_kernel
memcpy_orig
```

21.8% of move_pages syscall time is spent on folio_mc_copy

2MB folios

```
- 71.03% syscall
- 70.93% entry_SYSCALL_64_after_hwframe
- do_syscall_64
- __x64_sys_move_pages
- kernel_move_pages
- 70.93% do_pages_move
- 70.81% move_pages_and_store_status
- migrate_pages
- migrate_pages_sync
- migrate_pages_batch
- 70.32% migrate_folio_move
- 68.87% move_to_new_folio
- 68.68% folio_mc_copy
- 66.54% copy_mc_to_kernel
65.95% memcpy_orig
```

96.7% of move_pages syscall time is spent on folio_mc_copy

Effect of Scheduling for MT copying

a. Spread copy threads across different CCDs

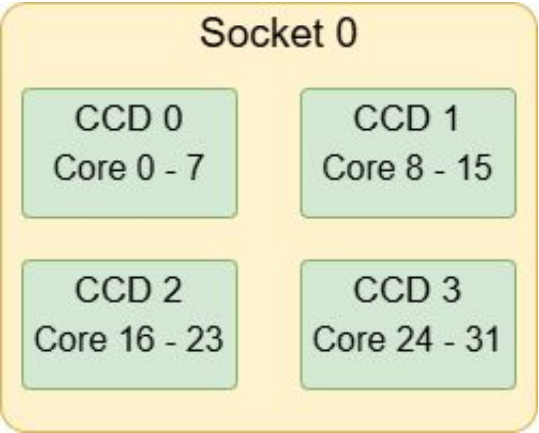
Threads	Tput (GB/s)
1	10.60±0.06
2	17.21±0.12
4	29.94±0.16
8	37.07±1.62
16	36.19±0.97

Better bandwidth utilization.

b. Fill one CCD completely before moving to next

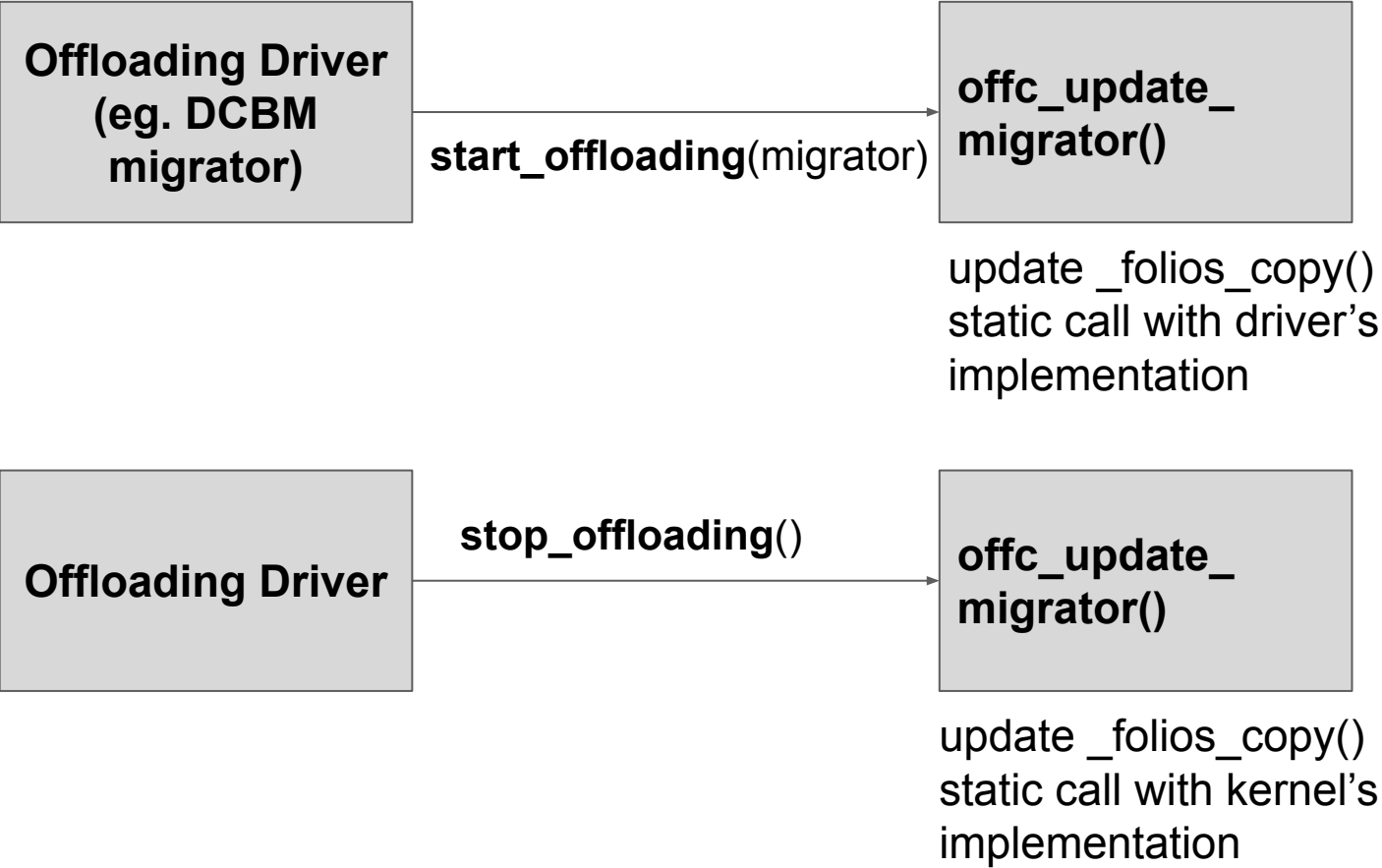
Threads	Tput (GB/s)
1	10.44±0.47
2	10.93±0.11
4	10.99±0.04
8	11.08±0.03
16	17.91±0.12

Throughput is bottlenecked by CCD Bandwidth. Increasing Thread does not help.



Total 8 CCDs.
Each socket has 4 CCDs and
each CCD has 8 cores.

Migration Offload Infrastructure for Batch-Copying



No ifs, buts, or messy conditionals to decide which batch copy method to use.

Just use `static_call()` - keeps things simple and fast with minimal overhead.

```

/* Batch copy the folios */
static_call(_folios_copy)(dst_folios,
src_folios, nr_batched_folios);
  
```

Implementation: Migration Offload Infrastructure

- Patch (From [Mike Day](#)): [mm: add support for copy offload for folio Migration](#)
- Plugin architecture via struct migrator to easily plug any offloading driver

```
struct migrator {  
    char name[MIGRATOR_NAME_LEN];  
    int (*migrate_offc)(struct list_head *dst_list, struct list_head *src_list, int folio_cnt);  
    bool (*can_migrate_offc)(struct folio *dst, struct folio *src);  
    struct rcu_head srcu_head;  
    struct module *owner;  
};
```



Ensure safe transition b/w
implementations at runtime,



Checks if offload copying is
supported for given src/dst folio
pair for given migrator



Transfers page content
between the
src and dst folios.

COPYRIGHT AND DISCLAIMER

©2025 Advanced Micro Devices, Inc. All rights reserved.

AMD, the AMD Arrow logo, EPYC and combinations thereof are trademarks of Advanced Micro Devices, Inc. Linux is a registered trademark of Linus Torvalds. Other company, product, and service names used in this publication are for identification purposes only and may be trademarks of their respective companies.

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS.' AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

