



Accelerating Page Migration and Making the Migration Core Composable

Kernel Memory Management Microconference

Shivank Garg, AMD
Zi Yan, NVIDIA



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Agenda

1. Why now?

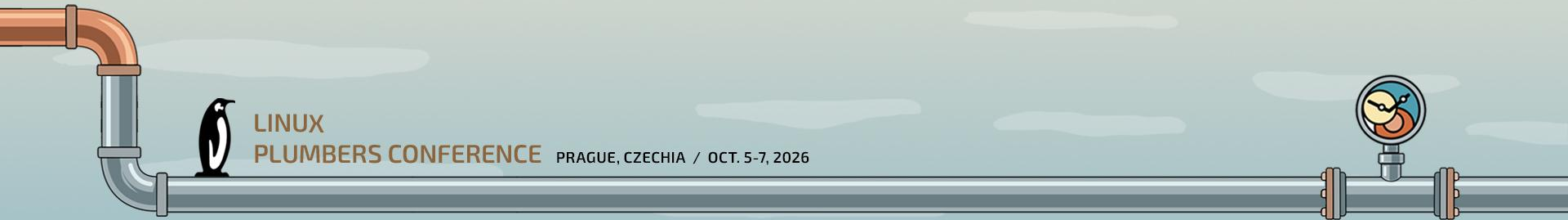
Tiered memory makes migration a hot path

2. Migration redesign (work in progress)

Various optimization, refactoring and asks.

3. Bringing everything together

Performance results, discussion



Where the time actually goes?

Profiled **move_pages()** on DRAM -> DRAM
AMD EPYC Zen 6, 7.3-rc1+

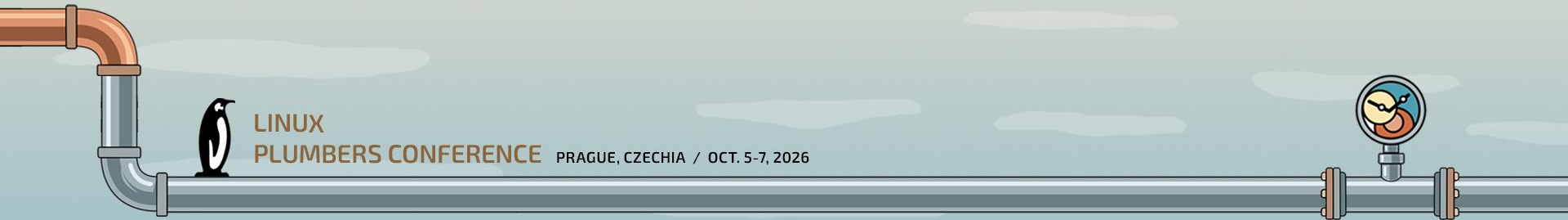
Folio copying: major overhead for large folios

Modern systems have underutilized:

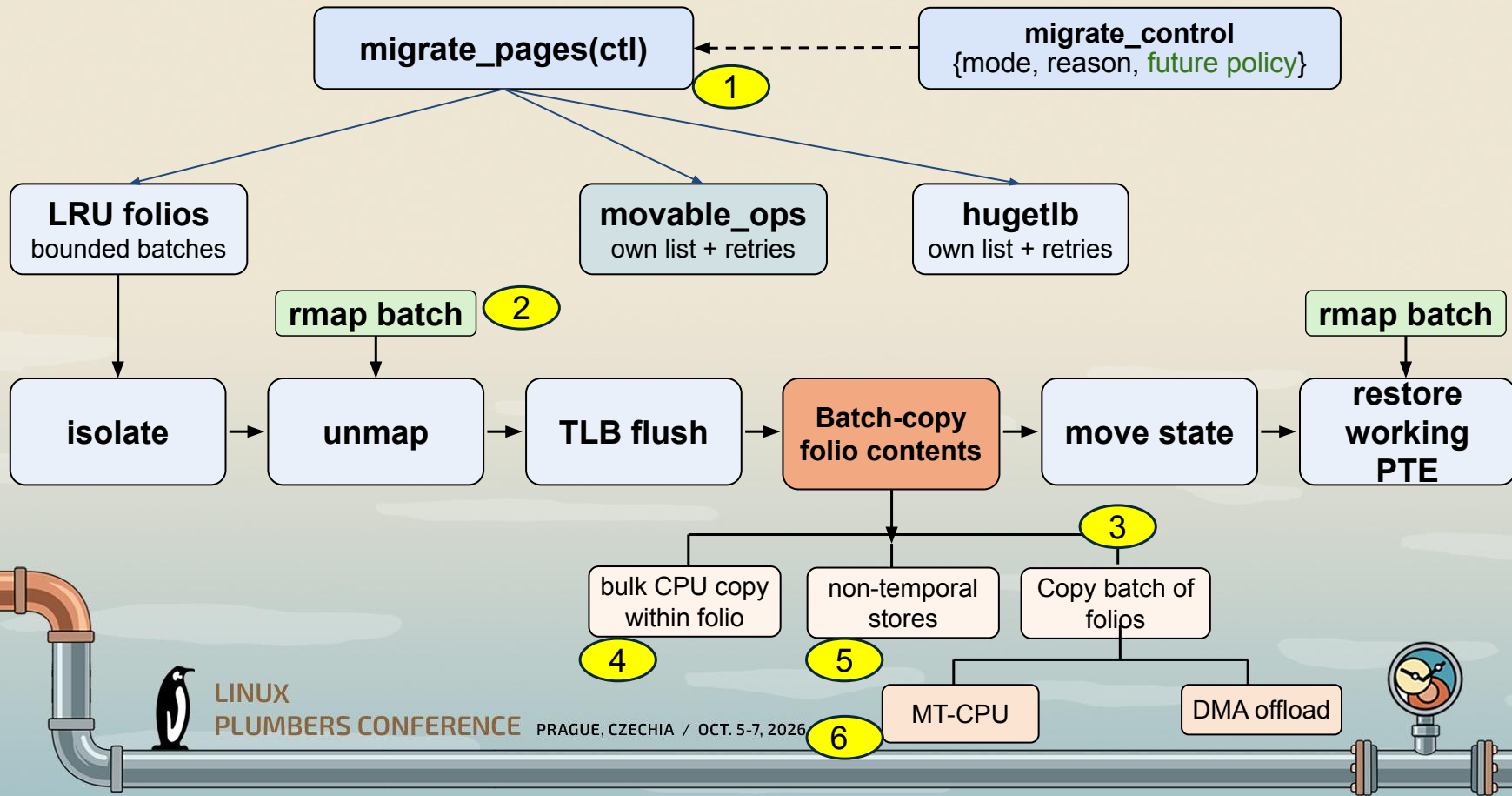
- memory bandwidth
- hardware accelerators
- idle CPUs

=> Leverage them to offload the copying work.

Phase Folio sizes	4 KiB	1 MiB	2 MiB
Folio copy	12%	51%	96%
Rmap walks (unmap + restore)	29%	48%	1%
LRU / accounting	11%	<1%	<1%
Destination alloc	4%	1%	1%
Other	44%	~0	2%

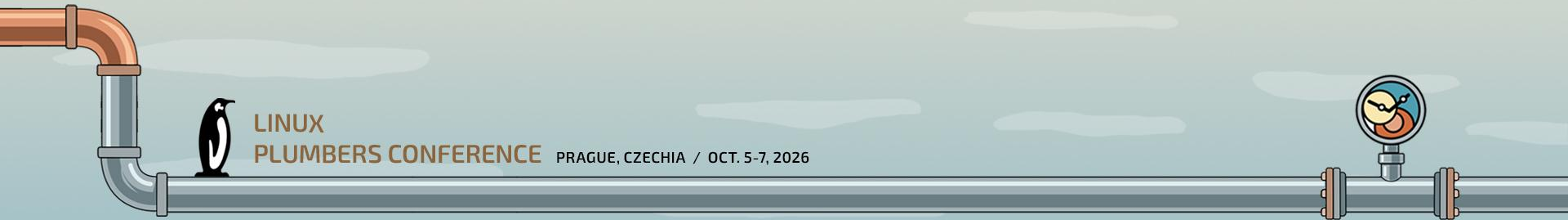


Migration redesign: work in progress



Migration redesign: Current Status

- ① [mm/migrate: separate migration paths and carry migration policy](#): RFC V1
- ② [mm: batch rmap walks during large folio migration](#): V2
- ③ [Accelerate page migration with batch copying and hardware offload](#): RFC V6
- ④ [mm: batch page copies in folio_copy\(\) and folio_mc_copy\(\)](#): RFC V1
- ⑤ [Demote to lower tier using non-temporal stores](#) (from Yiannis): RFC V2
- ⑥ [mtcopy: introduce multi-threaded page copy routine](#) (from Zi): RFC V3



Migration Policy: migrate_mode is not enough

Current:

```
migrate_pages(mode, reason)
```

```
-> move_to_new_folio(mode) // reason is not passed
```

```
-> aops->migrate_folio(mode)
```

```
-> folio_mc_copy() /* no reason/policy */
```

Today, only migrate_mode reaches the copy path.

A new usecase, such as non-temporal stores, would need another mode.

However, existing code relies on enum's ordering:

```
mode != MIGRATE_ASYNC, mode < MIGRATE_SYNC
```

```
enum migrate_mode {  
    MIGRATE_ASYNC,  
    MIGRATE_SYNC_LIGHT,  
    MIGRATE_SYNC,  
} => how much may migration block?
```

Keep migrate_mode for blocking behavior only.



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Migration Policy: introduce migrate_control

Proposed:

```
struct migrate_control {  
    enum migrate_mode    mode;  
    enum migrate_reason reason; // Why is migration happening?  
    /* new future policy - Extensible without another API change */  
}
```

migrate_pages(ctl)

-> move_to_new_folio(ctl)

-> aops->migrate_folio(ctl)

-> folio copy can act on intent

Context follows the migration path end-to-end, from the caller to folio copy path.

Is this the right shape and one-time API churn worth it?



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Migration design: migrate_control

- Policy that could follow:
 - Cache Intent : Non-temporal stores or normal store
 - Preferred copy engine : DMA / single CPU / multi-threaded CPU, ...
 - Constraints : Caller's latency / batch constraints

But does each policy need to be explicit, or can core derive it from reason?

- Case of Non-temporal store during demotion:

A: CALLER STATES IT

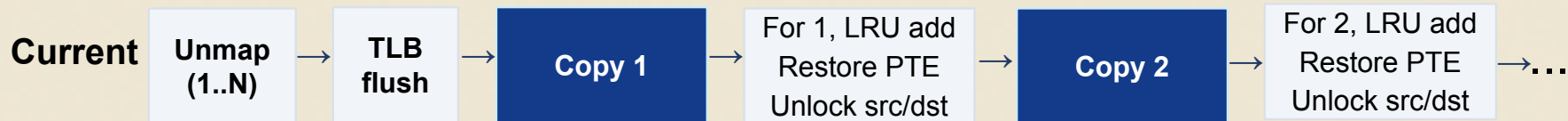
```
demote_folio_list()  
  ctl.reason = MR_DEMOTION  
  ctl.cache  = NON_TEMPORAL  
                |  
                v  
            copy uses NT stores
```

B: CORE DERIVES IT FROM REASON

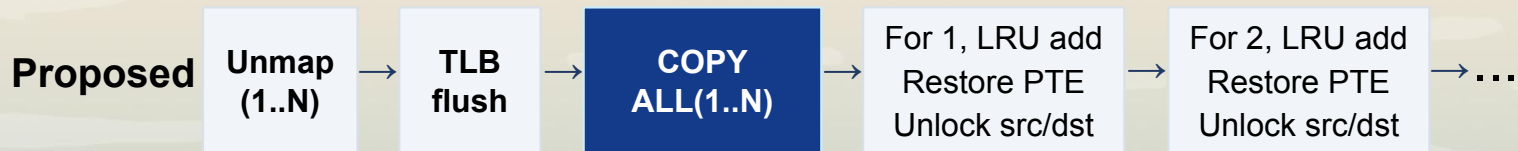
```
demote_folio_list()  
  ctl.reason = MR_DEMOTION  
                |  
                v  
    core: MR_DEMOTION -> NON_TEMPORAL  
                |  
                v  
            copy uses NT stores
```

*I'm inclined to keep reason only (B).
Adding cache intent here adds complexity and risks duplicating same info at two places.*

Batch copy and offload: first folio latency



Current batch limit, N: `NR_MAX_BATCHED_MIGRATION = 512`



Batch copying delays completion of the first folio because PTE restoration and folio unlock begins only after the entire batch is copied.



LINUX

PLUMBERS CONFERENCE

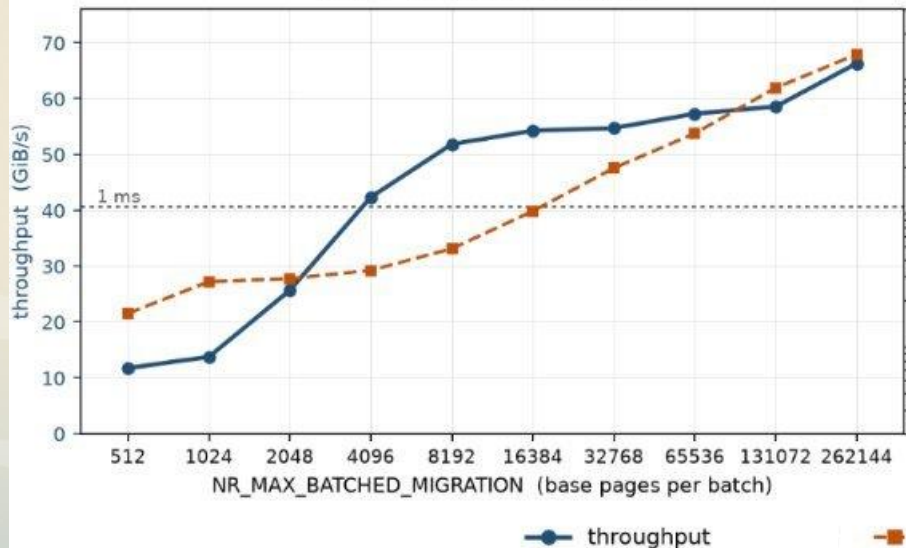
PRAGUE, CZECHIA / OCT. 5-7, 2026



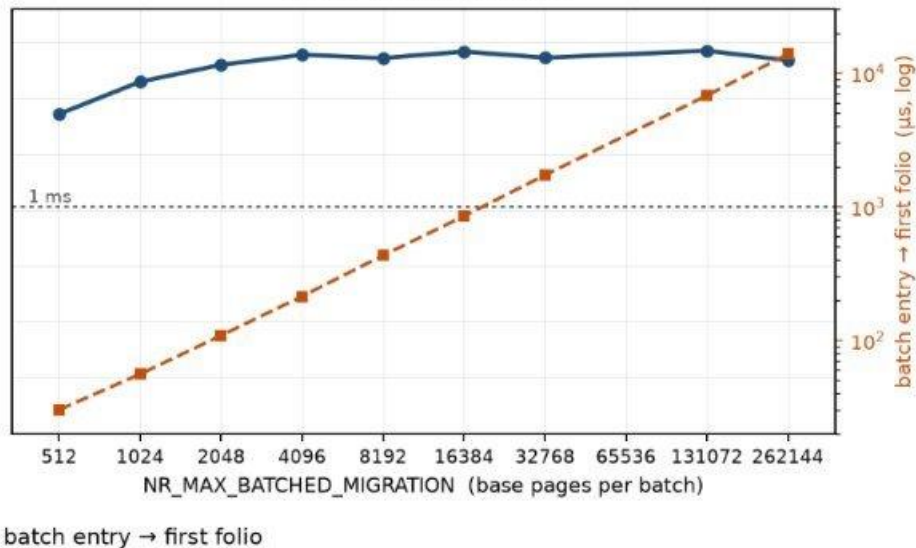
Batch copy and offload: Throughput vs. first-folio latency

Workload: `move_pages()`, 1GiB DRAM -> DRAM migration, 2MiB folios

Zen 3 · PTDMA, 16 channels



Zen 6 · SDXI, 1 channel



Concern: Larger batches improve throughput, but increases first-folio latency.
The optimal batch size depends on the copy engine and hardware.

What first-folio latency is acceptable, and should the batch-size limit be configurable?



Batch copy and offload: who selects the copy engine?

One host can have several copy-engine options:

- Single-threaded CPU, the default
- Multi-threaded CPU
- DMA Offload

No engine always wins:

- Small folios, such as 4KiB: CPU
- Large folios, such as 2MiB: DMA
- Larger batches: Better amortize DMA setup cost

Who picks the copy engine?

- CALLER: Maximum flexibility, but every caller must learn engines and topology
- PROVIDER: Admin enables the engine system-wide, selection policy and fallback remain within the provider.
- CORE: Caller supplies constraints, provider gives capability, and the core selects an engine and fallback.

Which inputs should drive selection: folio order, batch size, topology, or latency budget?

Current RFC: Provider selects the engine.

Refactor: separate migration classes

Current:

```
migrate_pages()  
|  
+--> migrate_hugetlbs()  
|  
`--> migrate_pages_batch()  
      LRU folios and movable_ops  
      pages together
```

Proposed:

```
migrate_pages(control)  
|  
+--> migrate_hugetlbs           owns list + retries  
|  
+--> migrate_movable_ops_pages  owns list + retries  
|  
`--> migrate_lru_folios         batching + sync fallback
```

Separate ownership simplify the class-specific list handling and retry policy from the common migration path.

Remaining:

movable_ops pages are not folios. However, they currently use page->lru and folio-based callbacks get_new_folio / put_new_folio.

What should the non-folio migration interface look like?



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Refactor: simplify synchronous fallback

Today:

```
migrate_pages_sync()  
|  
+--> migrate_pages_batch(),  
|     MIGRATE_ASYNC, 3 retries  
|  
`--> for each remaining folio:  
      create single-entry list  
      migrate_pages_batch(),  
      caller's migrate_mode, 7 retries
```

A one-folio retry runs the whole multi-folio engine.

Proposed:

```
migrate_pages_sync()  
|  
+--> migrate_pages_batch(),  
|     MIGRATE_ASYNC, unchanged  
|  
`--> retry remaining folios:  
      migrate_folio_sync_one()  
      alloc dst -> lock -> unmap -> move  
      success, or undo and retry on the next pass
```

Should single-folio retries get their own path, or keep reusing migrate_pages_batch()?



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Folio copy today: one 4 KiB page at a time

Normal copy:

```
folio_copy()  
|  
`--> for each 4 KiB page:  
    |--> copy_highpage()  
    `-> copy_page()  
        cond_resched()
```

Machine-check copy:

```
folio_mc_copy()  
|  
`--> for each 4 KiB page:  
    |--> copy_mc_highpage()  
    `-> copy_mc_to_kernel(PAGE_SIZE)  
        cond_resched()
```

Limitation: Copies are done one `PAGE_SIZE` at a time, so per-call overhead is paid for every 4KiB page (512x for a 2 MiB folio). For instance, each `rep movsb` / `rep movsq` invocation incurs a startup costs.

Opportunity:

- `copy_page()` is already architecture-specific: extend it to copy a page range, `copy_pages()`.
- `copy_mc_to_kernel()` can take variable length to copy full folio.



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Folio copy: copy an entire large folio

Normal copy:

```
folio_copy(N = n * PAGE_SIZE)
|
`-> copy_highpages(N)
    |
    `--> copy_pages(N): entire folio
```

Machine-check copy:

```
folio_mc_copy(N = n * PAGE_SIZE)
|
`-> copy_mc_highpages(N)
    |
    +--> copy_mc_to_kernel(N)
        on error: queue the first
        poisoned page, return -EHWPOISON
```

*Fallback paths are omitted for simplicity.

Proposal: Add `copy_pages()` (*plan for RFCv2, not posted yet*), a range version of `copy_page()`, and extend both paths to copy the whole folio in one call.

Measured: On AMD EPYC Zen 6, `move_pages()` throughput for 2 MiB folios improves by **39%** on the CPU copy path.

Is `copy_pages()` the right generic API for bulk folio copy()?



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Bringing it together: End-to-End migration throughput

AMD EPYC Zen 6, `move_pages()` end-to-end throughput, GiB/s

s1: Base Kernel (CPU)

s2: **s1** + batched rmap unmap and restore (CPU)

s3: **s2** + batch copy with DMA offload (SDXI, 1 channel)

s4: **s2** + bulk copy within each large folio (CPU)

Config	DRAM -> DRAM 1MiB Folio	DRAM -> DRAM 2MiB Folio	DRAM -> CXL 1MiB Folio	DRAM -> CXL 2MiB Folio
s1	10.33	19.61	9.88	18.69
s2	16.50 1.6x	19.68	15.73 1.6x	18.20
s3	37.95 3.7x	69.39 3.5x	25.09 2.5x	36.27 1.9x
s4	21.63	27.45 1.4x	21.19	27.25 1.5x

s1 as baseline



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

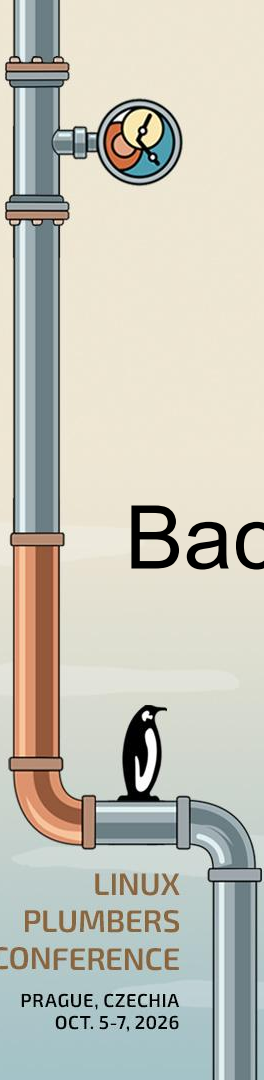


A vertical grey pipe with orange-colored joints and a circular gauge with a yellow face and blue needle. Below, a horizontal pipe section has a small penguin perched on top.

Thank you!

LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA
OCT. 5-7, 2026

A vertical pipe with a pressure gauge and a penguin on a horizontal section. The pipe is grey with orange-brown joints. A circular pressure gauge with a yellow face and blue needle is attached to the upper part of the vertical pipe. At the bottom, the pipe turns 90 degrees into a horizontal section where a small black and white penguin is perched.

Backup slides

LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA
OCT. 5-7, 2026

What a migrate_pages() caller passes (today)

```
struct migration_target_control mtc = {    // allocator cookie - passed as private
    .nid      = target_nid,                // demotion node
    .nmask    = &allowed_mask,            // nodemask
    .gfp_mask = GFP_HIGHUSER_MOVABLE ..., // flags
    .reason   = MR_DEMOTION,              // reason #1
};
```

```
migrate_pages(list,
    alloc_demote_folio, NULL,            // get_new_folio, put_new_folio
    (unsigned long)&mtc,                 // private
    MIGRATE_ASYNC,                       // mode
    MR_DEMOTION,                         // reason #2
    &nr_succeeded);
```



migrate_pages() call flow (today)

```
migrate_pages(from, get_new_folio, put_new_folio, private, mode, reason, ...)
-> migrate_hugetlbs() // hugetlb dedicated pass
-> migrate_pages_batch() or migrate_pages_sync() -> migrate_pages_batch()
-> migrate_folio_unmap()
    -> get_new_folio(src, private) // sees mtc only
        -> alloc_migration_target()
            -> alloc dst folio
-> try_to_unmap_flush()
-> migrate_folio_move(mode, reason, private)
    -> move_to_new_folio(mode) // reason stops here
    -> a_ops->migrate_folio(mode) or ...
        -> __migrate_folio(mode)
            -> folio_mc_copy() // no info of migrate reason, single-threaded copy
-> add dst folio to lru
-> remove_migration_ptes(), unlock folio, and other business
-> on failure: put_new_folio(dst, private), remove_migration_ptes()
```



System Information

HARDWARE

Platform AMD EPYC Zen 6, 256 CPUs, SMT on

NUMA NPS-2

Nodes 0 and 1 have CPUs, ~250 GB each

Remote tier CXL node 2, no CPUs, ~250 GB

Devices two SDXI 1.1 devices

SDXI DMA dma0chan0, dma1chan0

WORKLOAD

`move_pages(2)` system call

1 GiB anonymous memory per run, different folio sizes,

Using `MADV_HUGEPAGE` at the requested order and `mbind()` to the source node



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Throughput vs First folio latency trade off on Zen 6 (SDXI)

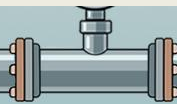
MEASURED SWEEP: Zen 6, DRAM, 2 MiB folios

Runtime knob changes NR_MAX_BATCHED_MIGRATION, Budget is in base pages

first = migrate_pages_batch() entry -> first completed folio

Budget (pages)	Folios	CPU GiB/s	CPU first (μs)	SDXI GiB/s	SDXI first (μs)
512	1	26.24	70	57.26	30
1,024	2	26.83	72	63.03	56
2,048	4	26.86	76	66.06	109
4,096	8	27.25	82	67.85	214
8,192	16	26.35	102	67.26	432
16,384	32	27.25	125	68.44	850
32,768	64	26.27	192	67.33	1,724

CPU throughput stays near 27 GiB/s; SDXI reaches ~68 GiB/s while first-completion time rises with batch size.



Batched rmap and DMA offload impact on Zen 3 (PTDMA)

MEASURED:

AMD EPYC ZEN 3, 2-socket, NPS-1, 7.2-rc5+ (mm-new)

batched rmap + DMA offload on PTDMA (DCBM), 16 channels

Folio	Vanilla	Batch-copy offload (DCBM)	DCBM + restore	DCBM + restore + unmap
4 KiB	3.05	2.76	2.77	2.74
64 KiB	5.85	10.66	11.56	17.55
256 KiB	6.26	13.64	13.68	34.76
1 MiB	6.58	14.25	15.27	44.28
2 MiB	10.61	65.26	65.25	65.66

Rmap batching contribution: 256 KiB (2.54×) and 1 MiB (3.11×).



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Throughput vs first folio latency trade off on Zen 3 (PTDMA)

MEASURED - ORDER 9, 2 MiB FOLIOS

move_pages(2): Throughput (GiB/s) and first-folio latency (us)

Total migrated data fixed at 1 GiB first = migrate_pages_batch() entry → first completed/restored folio

DMA16 N	GiB/s	first μ s
512	11.75	159
1024	13.76	275
2048	25.69	289
4096	42.36	334
8192	51.85	485
16384	54.26	925
32768	54.72	1942
65536	54.73	3535
131072	58.58	7747
262144	66.31	13801



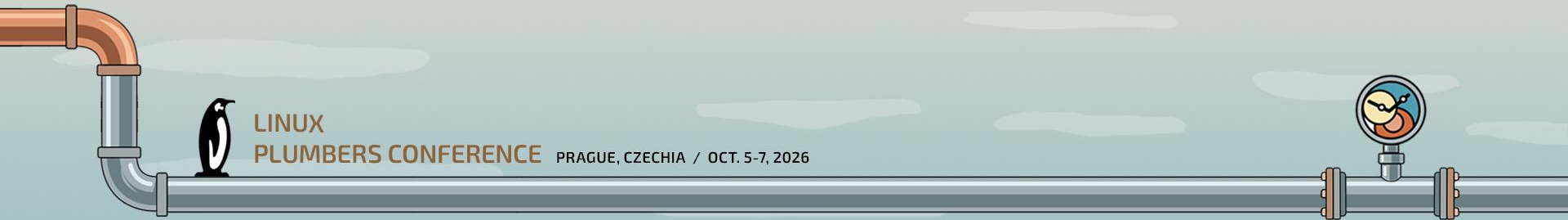
LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Multithreaded page copy

- The implementation is ready, but the concern on how to charge spawned kthread runtime back to the issuing thread is still present
- TODO:
 - Require substantial sched code changes to implement
 - Can prototype using sched_ext
- **Deferred until sched part is ready**



COPYRIGHT AND DISCLAIMER

©2026 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.

