PaCaR: Improved Buffered I/O Locality on NUMA Systems with Page Cache Replication

EuroSys'26

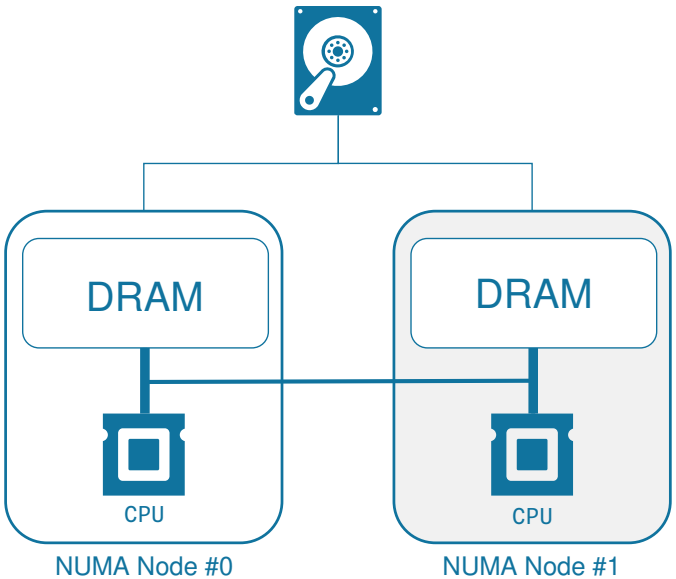
Jérôme Coquisart[1], Julien Sopena[2], Redha Gouicem[1]

[1]RWTH Aachen University, [2]Sorbonne Université

April 28, 2026

Performance and NUMA Architectures

Datacenters predominantly feature multi-socket hardware with Non-Uniform Memory Access (NUMA) architectures.

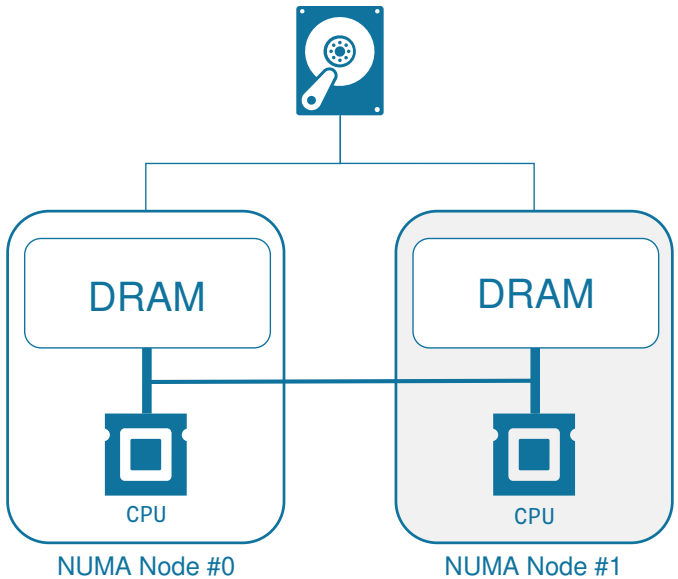


Performance and NUMA Architectures

Datacenters predominantly feature multi-socket hardware with Non-Uniform Memory Access (NUMA) architectures.

Challenges:

- **Latency increases** across nodes.
- **Memory bandwidth decreases** across nodes.



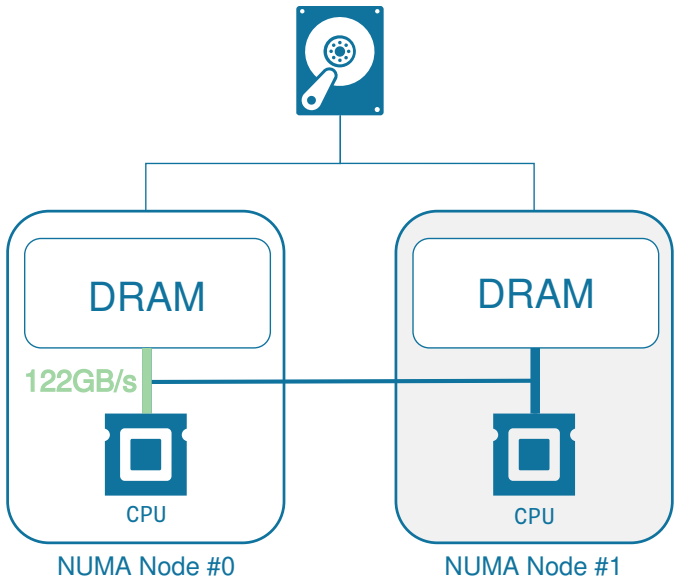
Processor		Latency (ns)			Bandwidth (GB/s)		
Model	*Release Date*	*Local*	*Remote*	*Penalty*	*Local*	*Remote*	*Penalty*
2× Intel Xeon Gold 5320	2023	94.6	162.3	**+71.6%**	122.7	55.5	**−54.7%**
2× Intel Xeon Silver 4410Y	2023	109.1	181.7	**+66.5%**	141.4	63.7	**−54.9%**
2× Intel Xeon Silver 4314	2022	89.3	152.5	**+70.8%**	131.5	34.5	**−73.8%**
2× AMD EPYC 7301	2018	91.2	267.6	**+193%**	17.3	7.4	**−57.4%**
4× Intel Xeon Gold 6130	2017	87.4	146.9	**+68.1%**	104.0	17.2	**−83.4%**
2× Intel Xeon Gold 6130	2017	89.6	143.3	**+59.9%**	109.8	34.4	**−68.6%**
2× Intel Xeon E5-2630	2012	80.8	131.9	**+63.2%**	38.6	18.6	**−51.8%**

Performance and NUMA Architectures

Datacenters predominantly feature multi-socket hardware with Non-Uniform Memory Access (NUMA) architectures.

Challenges:

- **Latency increases** across nodes.
- **Memory bandwidth decreases** across nodes.



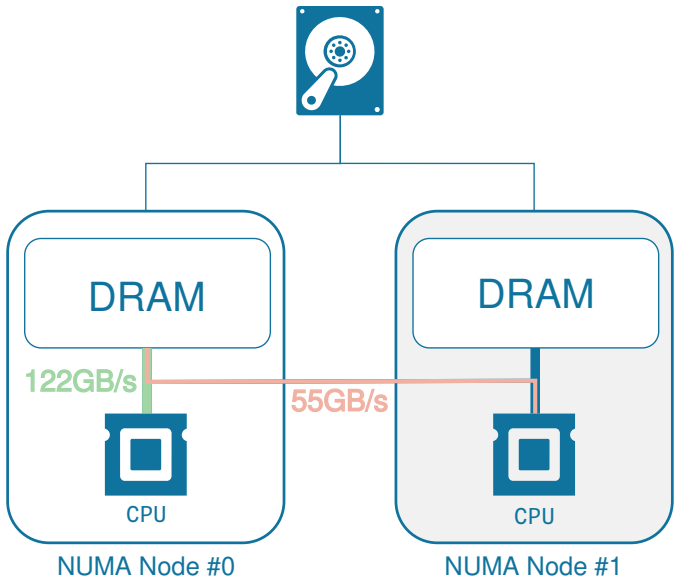
Processor		Latency (ns)			Bandwidth (GB/s)		
Model	*Release Date*	*Local*	*Remote*	*Penalty*	*Local*	*Remote*	*Penalty*
2× Intel Xeon Gold 5320	2023	94.6	162.3	**+71.6%**	122.7	55.5	**−54.7%**
2× Intel Xeon Silver 4410Y	2023	109.1	181.7	**+66.5%**	141.4	63.7	**−54.9%**
2× Intel Xeon Silver 4314	2022	89.3	152.5	**+70.8%**	131.5	34.5	**−73.8%**
2× AMD EPYC 7301	2018	91.2	267.6	**+193%**	17.3	7.4	**−57.4%**
4× Intel Xeon Gold 6130	2017	87.4	146.9	**+68.1%**	104.0	17.2	**−83.4%**
2× Intel Xeon Gold 6130	2017	89.6	143.3	**+59.9%**	109.8	34.4	**−68.6%**
2× Intel Xeon E5-2630	2012	80.8	131.9	**+63.2%**	38.6	18.6	**−51.8%**

Performance and NUMA Architectures

Datacenters predominantly feature multi-socket hardware with Non-Uniform Memory Access (NUMA) architectures.

Challenges:

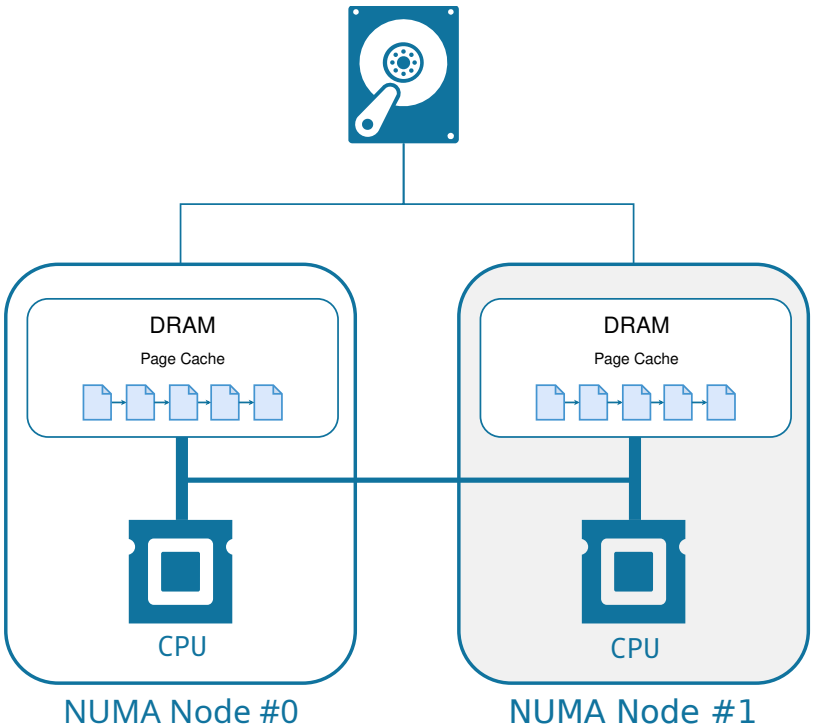
- **Latency increases** across nodes.
- **Memory bandwidth decreases** across nodes.



Processor		Latency (ns)			Bandwidth (GB/s)		
Model	*Release Date*	*Local*	*Remote*	*Penalty*	*Local*	*Remote*	*Penalty*
2× Intel Xeon Gold 5320	2023	94.6	162.3	**+71.6%**	122.7	55.5	**−54.7%**
2× Intel Xeon Silver 4410Y	2023	109.1	181.7	**+66.5%**	141.4	63.7	**−54.9%**
2× Intel Xeon Silver 4314	2022	89.3	152.5	**+70.8%**	131.5	34.5	**−73.8%**
2× AMD EPYC 7301	2018	91.2	267.6	**+193%**	17.3	7.4	**−57.4%**
4× Intel Xeon Gold 6130	2017	87.4	146.9	**+68.1%**	104.0	17.2	**−83.4%**
2× Intel Xeon Gold 6130	2017	89.6	143.3	**+59.9%**	109.8	34.4	**−68.6%**
2× Intel Xeon E5-2630	2012	80.8	131.9	**+63.2%**	38.6	18.6	**−51.8%**

Linux's Page Cache and Locality

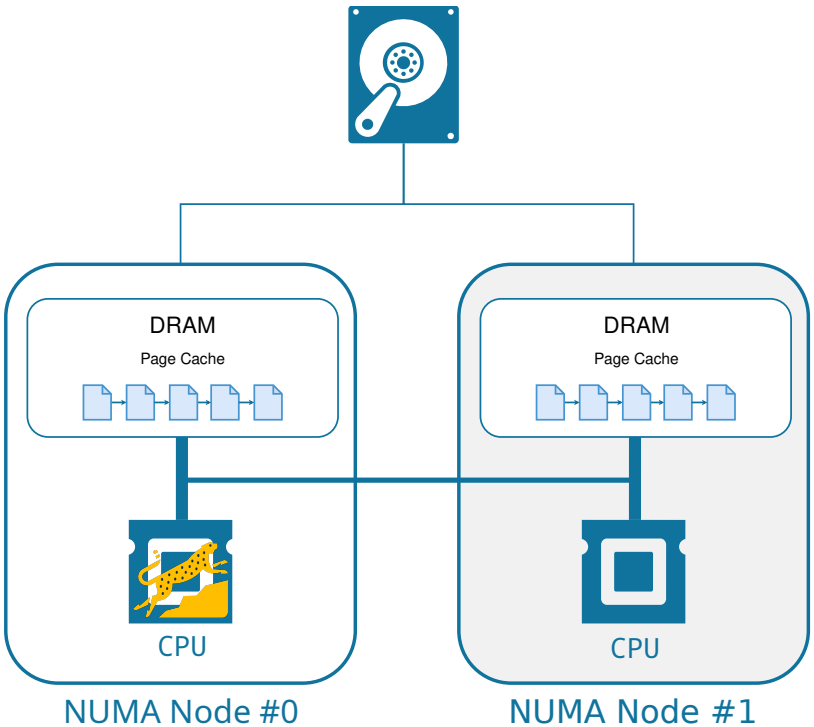
Let's first look at an example with RocksDB.



Linux's Page Cache and Locality

Let's first look at an example with RocksDB.

RocksDB starts on node 0

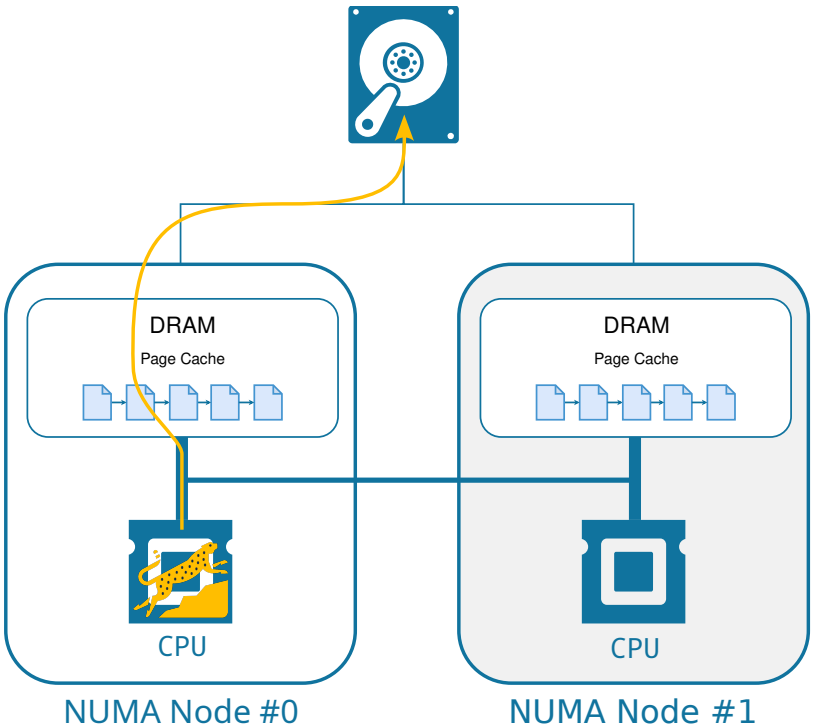


Linux's Page Cache and Locality

Let's first look at an example with RocksDB.

RocksDB starts on node 0

- RocksDB performs I/O operations

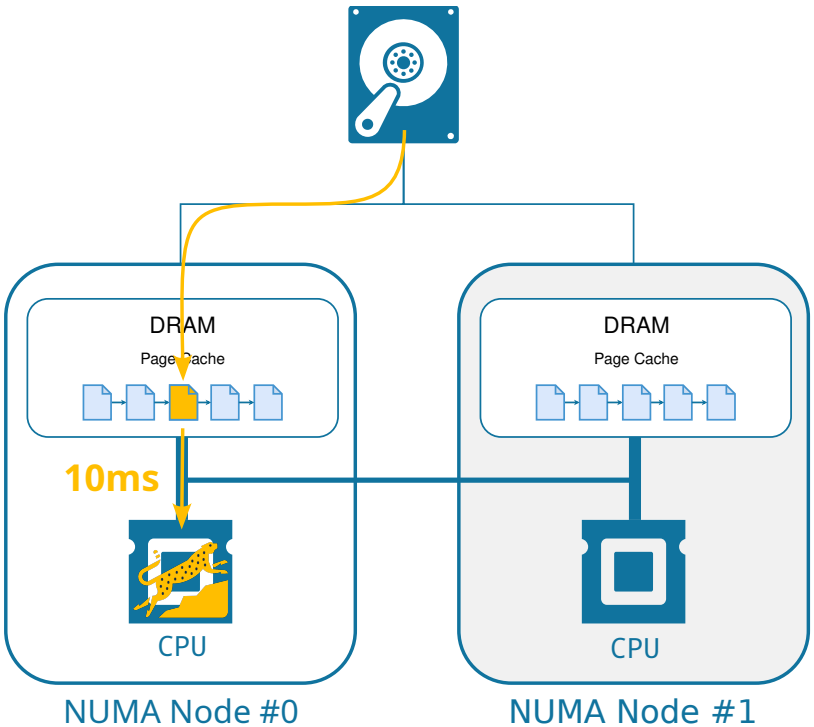


Linux's Page Cache and Locality

Let's first look at an example with RocksDB.

RocksDB starts on node 0

- RocksDB performs I/O operations
- Linux transparently populates the page cache

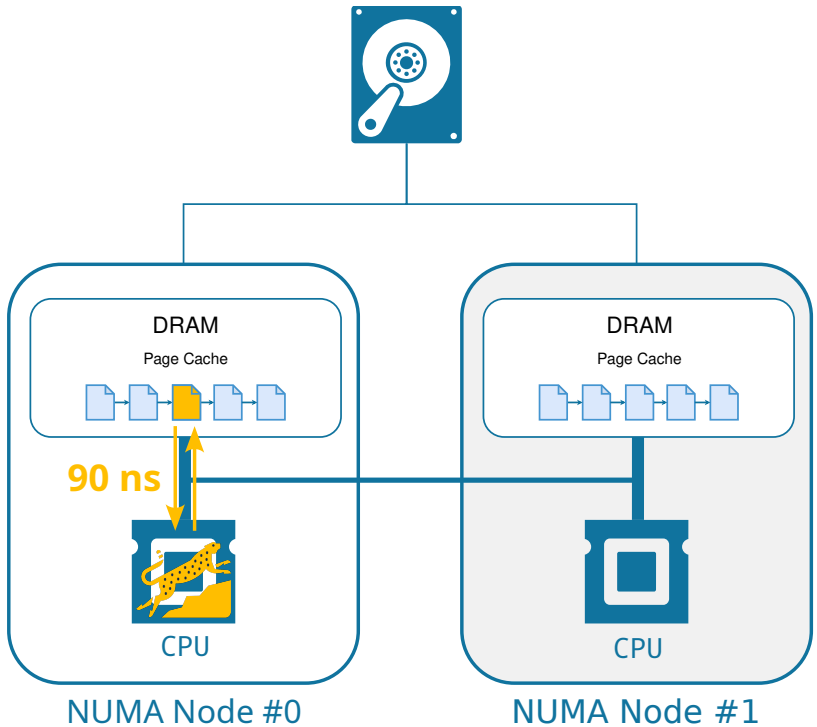


Linux's Page Cache and Locality

Let's first look at an example with RocksDB.

RocksDB starts on node 0

- RocksDB performs I/O operations
- Linux transparently populates the page cache
- Data is available through the page cache

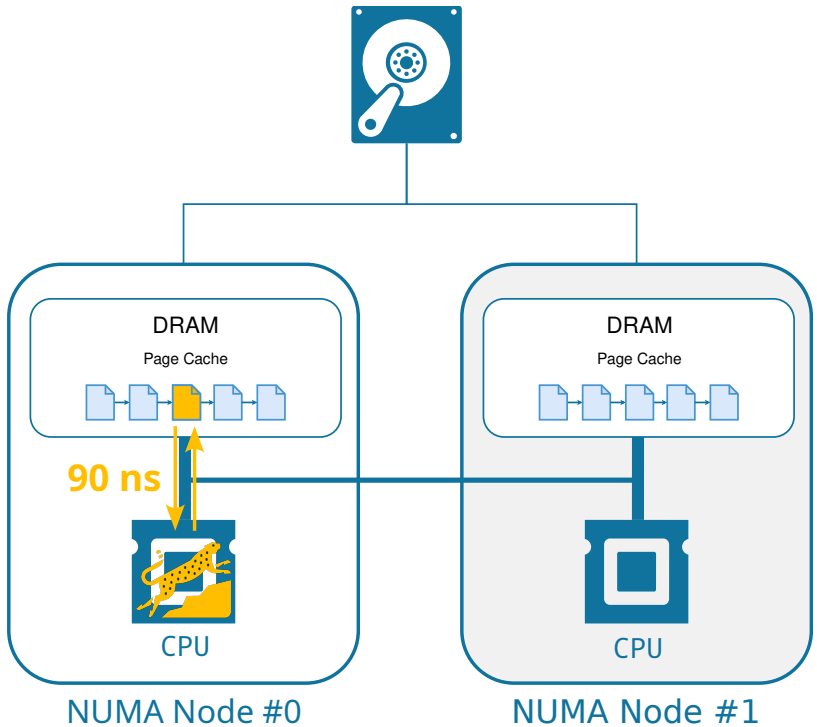


Linux's Page Cache and Locality

Let's first look at an example with RocksDB.

RocksDB starts on node 0

- RocksDB performs I/O operations
- Linux transparently populates the page cache
- Data is available through the page cache
- **Application throughput 28GB/s**



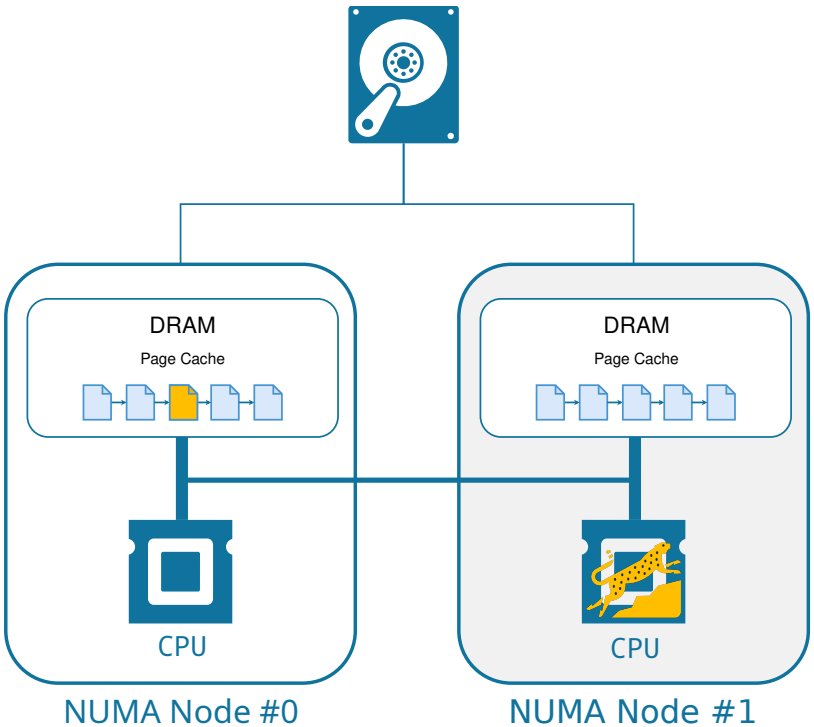
Linux's Page Cache and Locality

Let's first look at an example with RocksDB.

RocksDB starts on node 0

- RocksDB performs I/O operations
- Linux transparently populates the page cache
- Data is available through the page cache
- **Application throughput 28GB/s**

RocksDB restarts on node 1



Linux's Page Cache and Locality

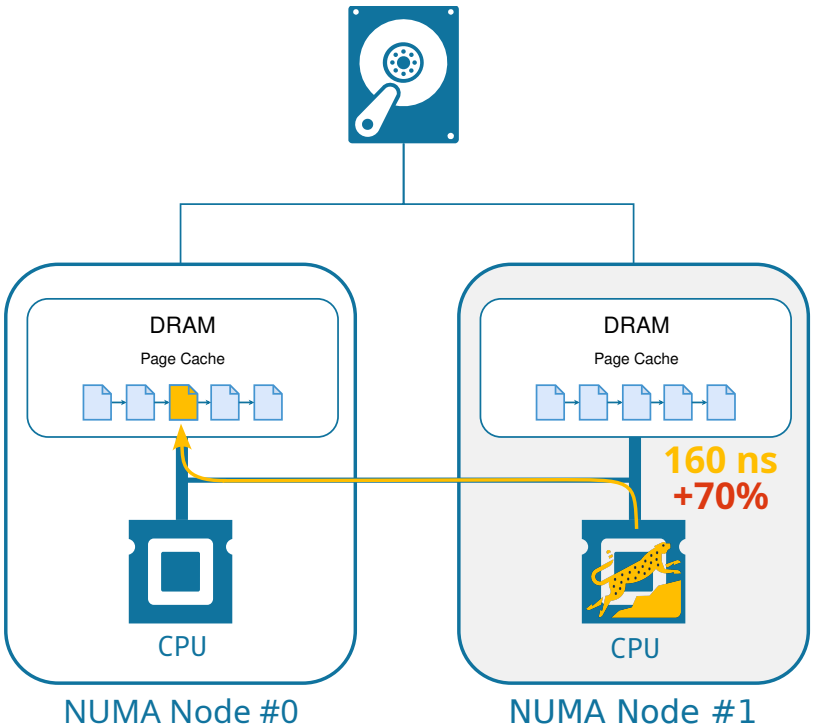
Let's first look at an example with RocksDB.

RocksDB starts on node 0

- RocksDB performs I/O operations
- Linux transparently populates the page cache
- Data is available through the page cache
- **Application throughput 28GB/s**

RocksDB restarts on node 1

- The same data is accessed



Linux's Page Cache and Locality

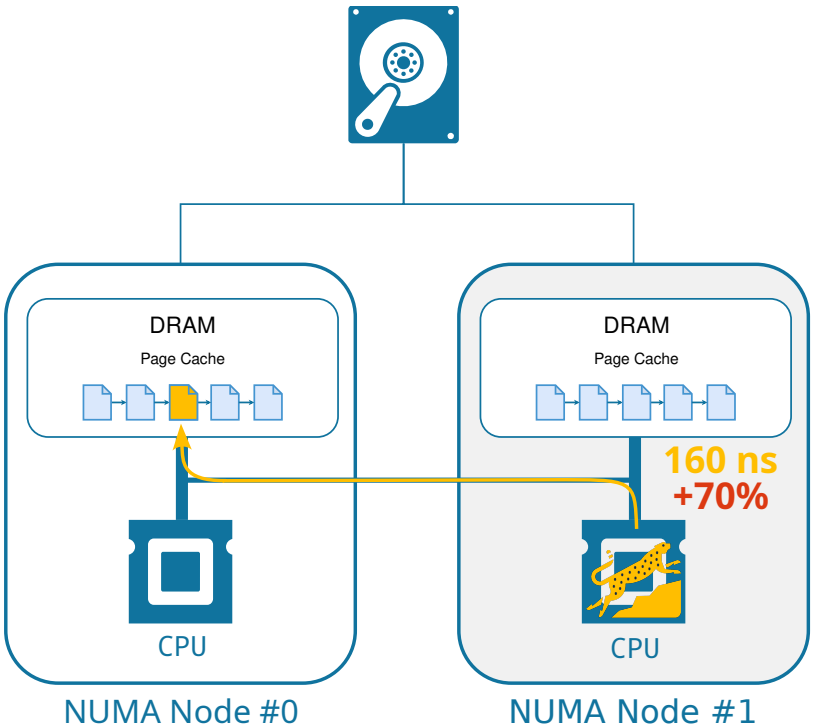
Let's first look at an example with RocksDB.

RocksDB starts on node 0

- RocksDB performs I/O operations
- Linux transparently populates the page cache
- Data is available through the page cache
- **Application throughput 28GB/s**

RocksDB restarts on node 1

- The same data is accessed
- **Application throughput 25.2GB/s**



Linux's Page Cache and Locality

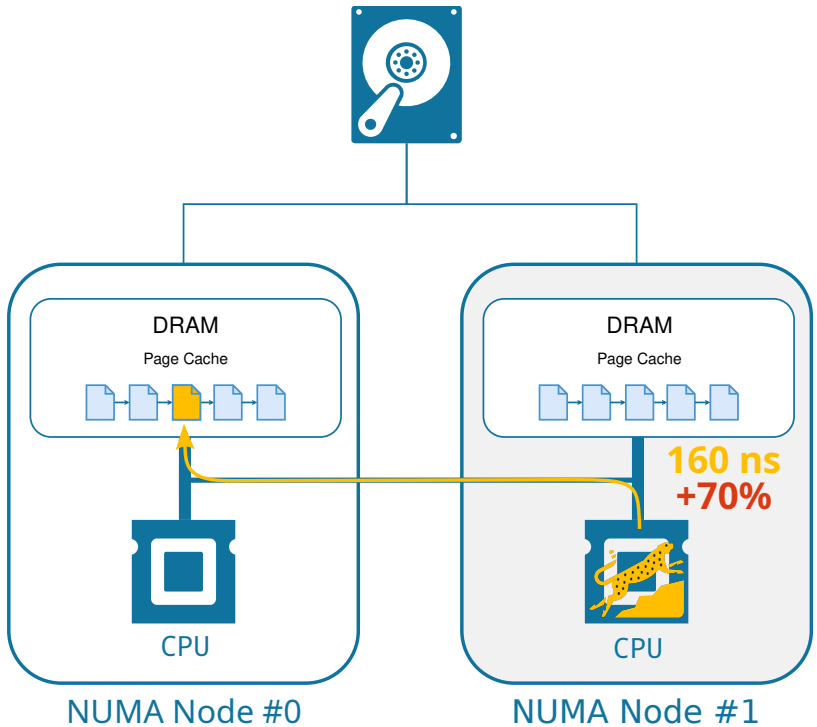
Let's first look at an example with RocksDB.

RocksDB starts on node 0

- RocksDB performs I/O operations
- Linux transparently populates the page cache
- Data is available through the page cache
- **Application throughput 28GB/s**

RocksDB restarts on node 1

- The same data is accessed
- **Application throughput 25.2GB/s**



Performance impact

We observe a 10% penalty in bandwidth and 70% penalty in latency.

Addressing the Lack of Locality

Manual solution : *numactl* (memory and CPU pinning)

Addressing the Lack of Locality

Manual solution : *numactl* (memory and CPU pinning)

- ✖ Doesn't scale

Addressing the Lack of Locality

Manual solution : *numactl* (memory and CPU pinning)

- ✖ Doesn't scale
- ✖ Doesn't solve the problem for applications spanning multiple nodes

Addressing the Lack of Locality

Manual solution : *numactl* (memory and CPU pinning)

- ✖ Doesn't scale
- ✖ Doesn't solve the problem for applications spanning multiple nodes

Automatic solution : *NUMA balancing* (memory and/or thread migration)

Addressing the Lack of Locality

Manual solution : *numactl* (memory and CPU pinning)

- ✖ Doesn't scale
- ✖ Doesn't solve the problem for applications spanning multiple nodes

Automatic solution : *NUMA balancing* (memory and/or thread migration)

- ✔ Works with anonymous memory

Addressing the Lack of Locality

Manual solution : *numactl* (memory and CPU pinning)

- ✖ Doesn't scale
- ✖ Doesn't solve the problem for applications spanning multiple nodes

Automatic solution : *NUMA balancing* (memory and/or thread migration)

- ✔ Works with anonymous memory
- ✔ Works with `mmap` files

Addressing the Lack of Locality

Manual solution : *numactl* (memory and CPU pinning)

- ✖ Doesn't scale
- ✖ Doesn't solve the problem for applications spanning multiple nodes

Automatic solution : *NUMA balancing* (memory and/or thread migration)

- ✔ Works with anonymous memory
- ✔ Works with `mmap` files
- ✖ Does **not** work with the page cache

Addressing the Lack of Locality

Manual solution : *numactl* (memory and CPU pinning)

- ✖ Doesn't scale
- ✖ Doesn't solve the problem for applications spanning multiple nodes

Automatic solution : *NUMA balancing* (memory and/or thread migration)

- ✔ Works with anonymous memory
- ✔ Works with `mmap` files
- ✖ Does **not** work with the page cache
- ✖ Doesn't solve the problem for applications spanning multiple nodes

Addressing the Lack of Locality

Manual solution : *numactl* (memory and CPU pinning)

- ✖ Doesn't scale
- ✖ Doesn't solve the problem for applications spanning multiple nodes

Automatic solution : *NUMA balancing* (memory and/or thread migration)

- ✔ Works with anonymous memory
- ✔ Works with `mmap` files
- ✖ Does **not** work with the page cache
- ✖ Doesn't solve the problem for applications spanning multiple nodes

Problem

How to improve locality for I/O intensive workloads on NUMA hardware?

PaCaR : Page Cache Replication

We propose PaCaR:

A page cache replication mechanism integrated to the Linux kernel's VFS, where each NUMA node gets a local copy of the page cache.

PaCaR : Page Cache Replication

We propose PaCaR:

A page cache replication mechanism integrated to the Linux kernel's VFS, where each NUMA node gets a local copy of the page cache.

Goals:

- Target read-heavy I/O intensive workloads
- Limited memory consumption
- Transparent to the user and application
- Minimal changes to existing stacks

PaCaR : Page Cache Replication

We propose PaCaR:

A page cache replication mechanism integrated to the Linux kernel's VFS, where each NUMA node gets a local copy of the page cache.

Goals:

- Target read-heavy I/O intensive workloads
- Limited memory consumption
- Transparent to the user and application
- Minimal changes to existing stacks

4 Mechanisms:

1. Page replication
2. Consistency
3. Local write optimization
4. Memory footprint limitation

Mechanism 1: Page Replication

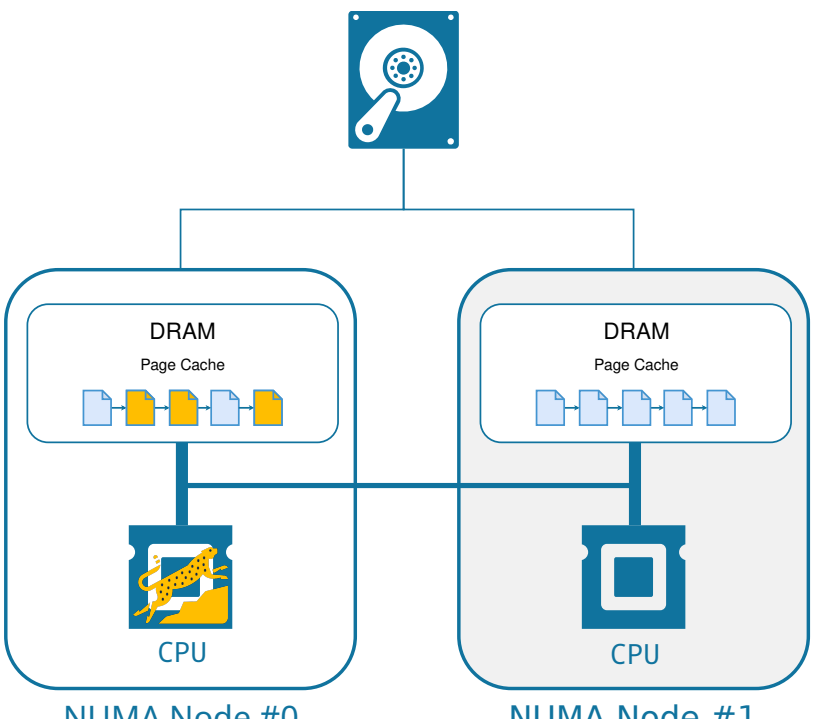
Problem

How to improve the locality of the page cache for read-heavy workloads?

Mechanism 1: Page Replication

Problem

How to improve the locality of the page cache for read-heavy workloads?



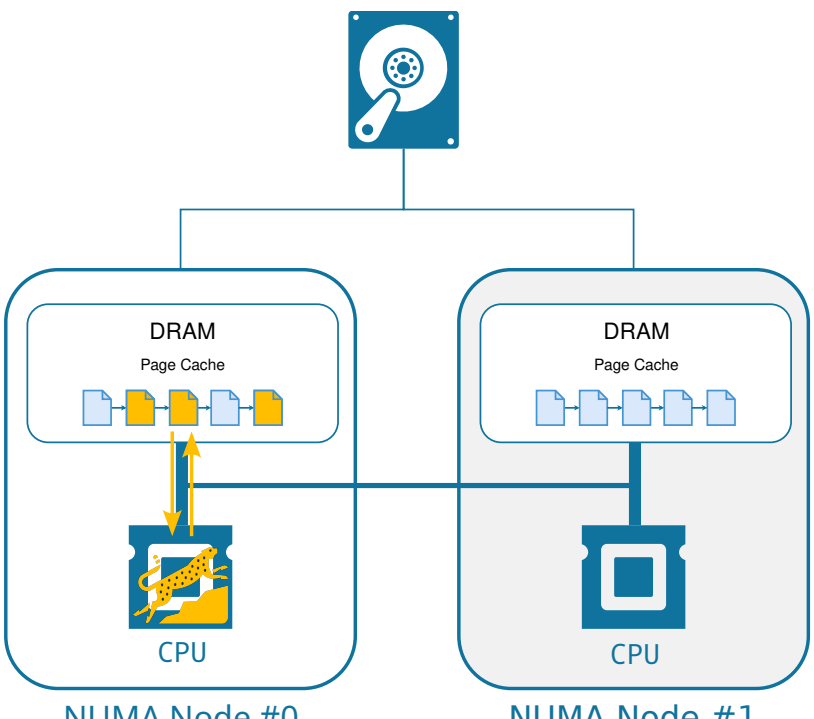
Mechanism 1: Page Replication

Problem

How to improve the locality of the page cache for read-heavy workloads?

On local read:

- Simply read the local page



Mechanism 1: Page Replication

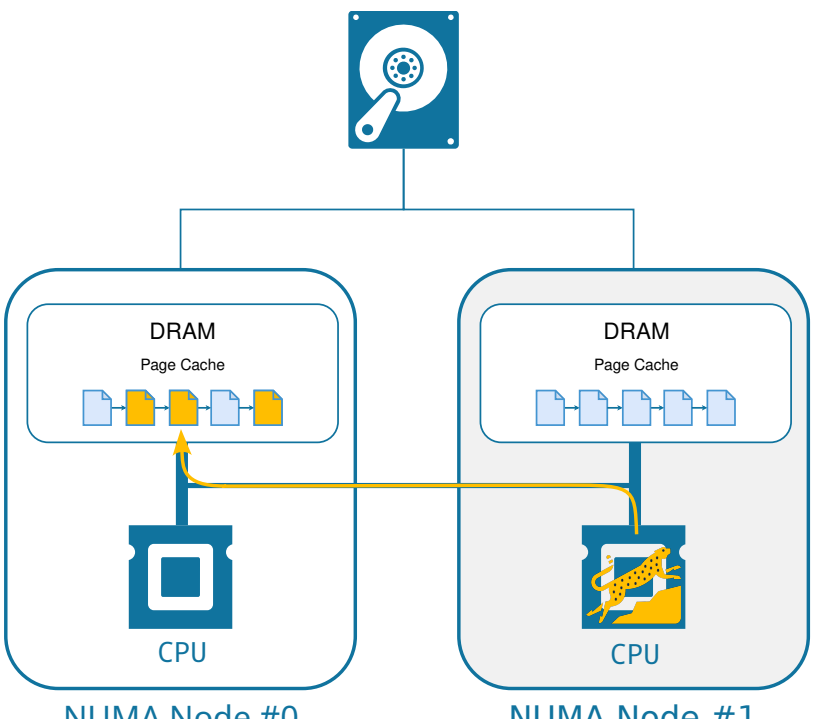
Problem

How to improve the locality of the page cache for read-heavy workloads?

On local read:

- Simply read the local page

On remote read:



Mechanism 1: Page Replication

Problem

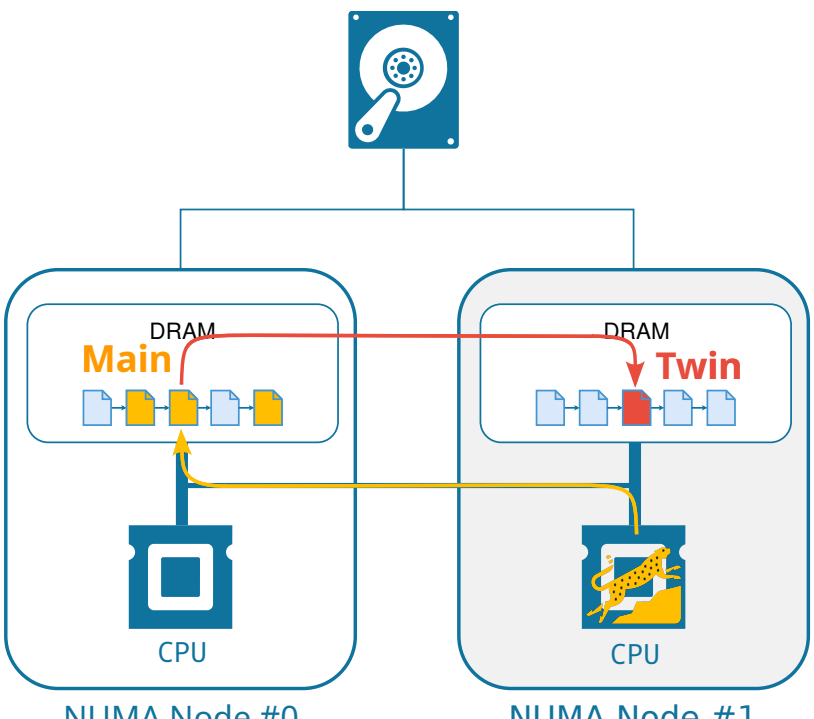
How to improve the locality of the page cache for read-heavy workloads?

On local read:

- Simply read the local page

On remote read:

- Replicate the data locally
- The original page is called *main*
- The replicated page is called *twin*



Mechanism 1: Page Replication

Problem

How to improve the locality of the page cache for read-heavy workloads?

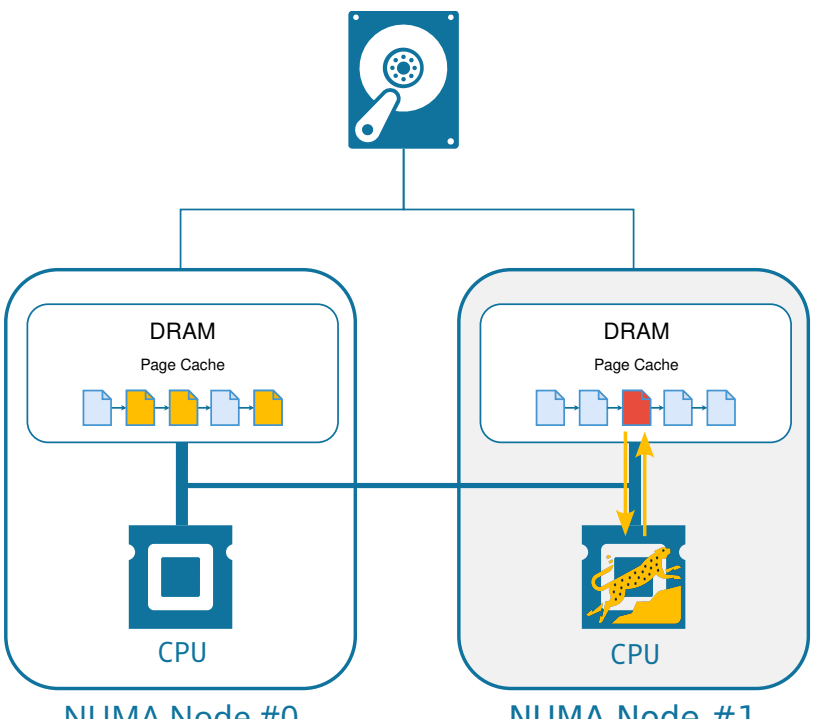
On local read:

- Simply read the local page

On remote read:

- Replicate the data locally
- The original page is called *main*
- The replicated page is called *twin*

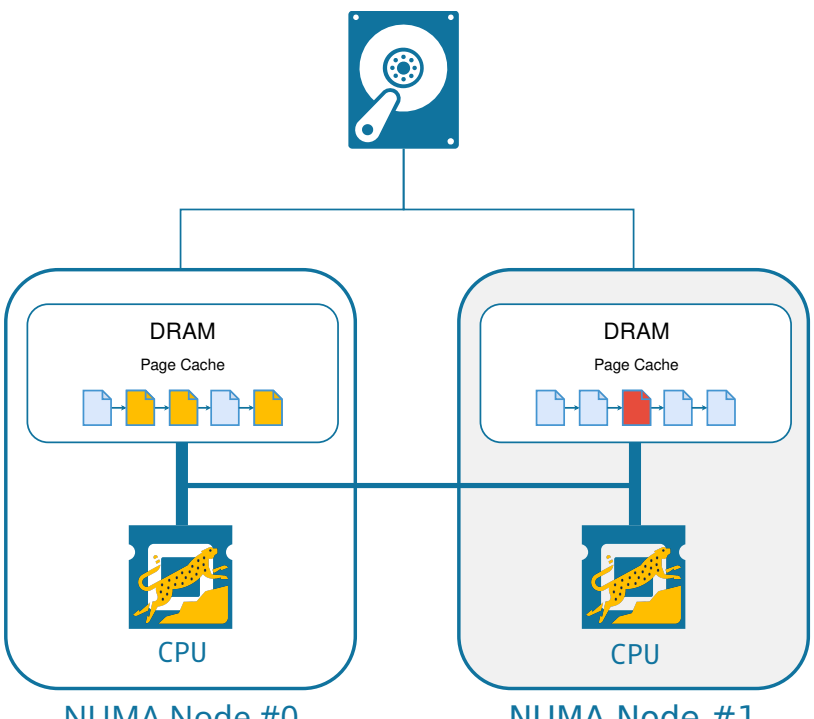
Subsequent I/Os are local no matter where the application runs.



Mechanism 2 : Consistency

Problem

How to ensure consistent access to the page cache?



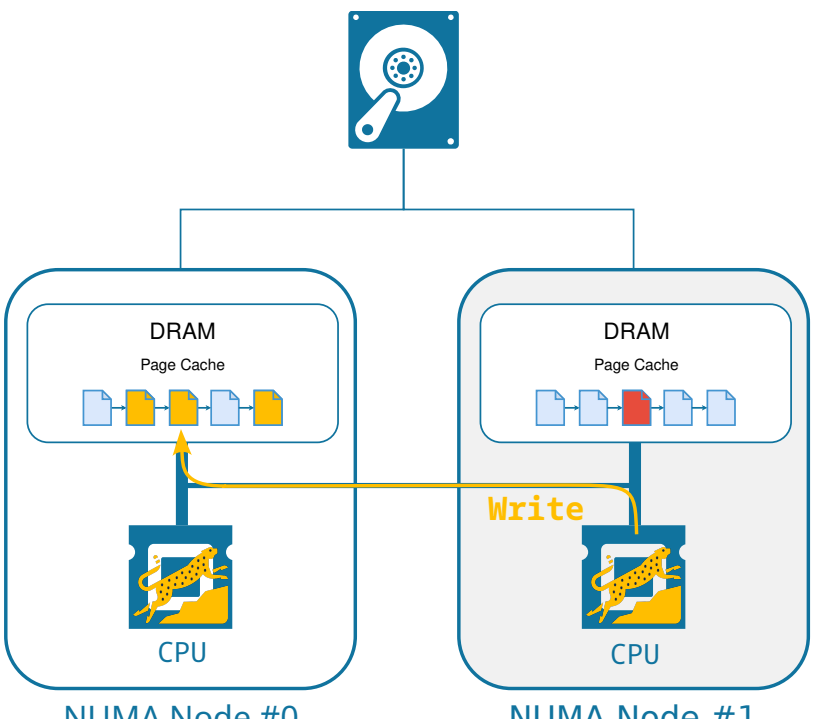
Mechanism 2 : Consistency

Problem

How to ensure consistent access to the page cache?

On write:

- **Always** write to the *main* page.



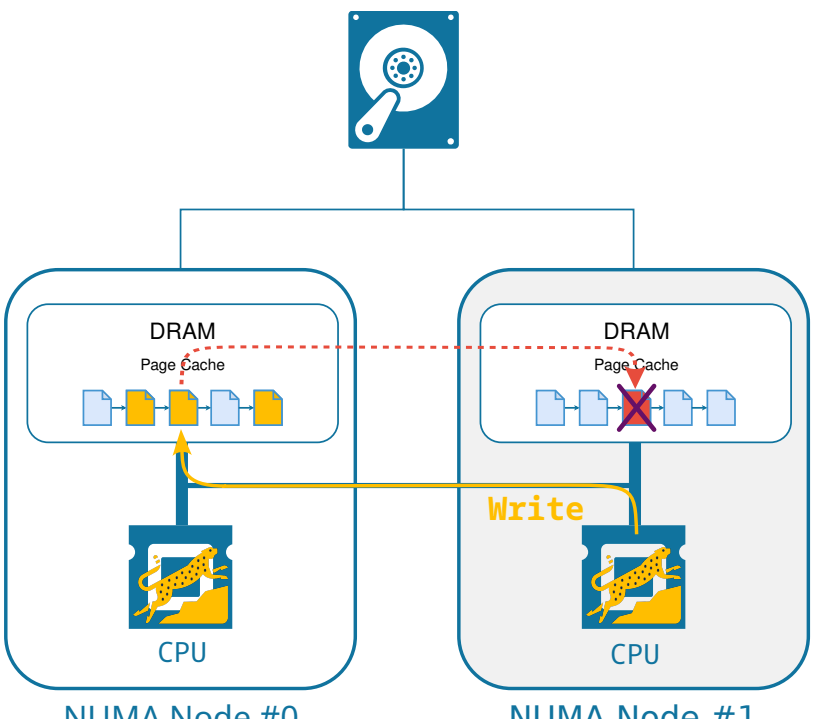
Mechanism 2 : Consistency

Problem

How to ensure consistent access to the page cache?

On write:

- **Always** write to the *main* page.
- Batch invalidation: Invalidate all associated *twins* using a single flag switch.



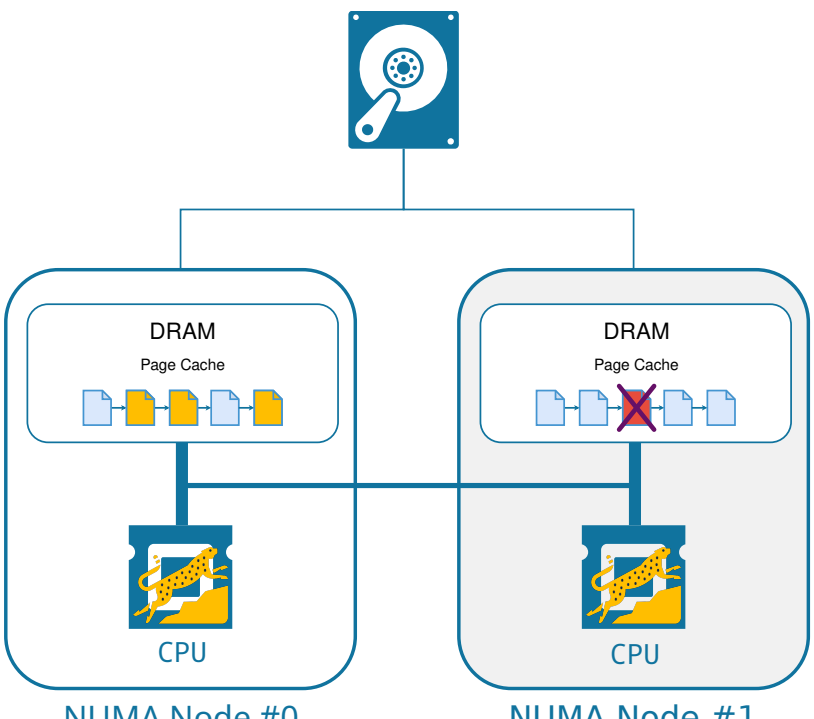
Mechanism 2 : Consistency

Problem

How to ensure consistent access to the page cache?

On write:

- **Always** write to the *main* page.
- Batch invalidation: Invalidate all associated *twins* using a single flag switch.



Mechanism 2 : Consistency

Problem

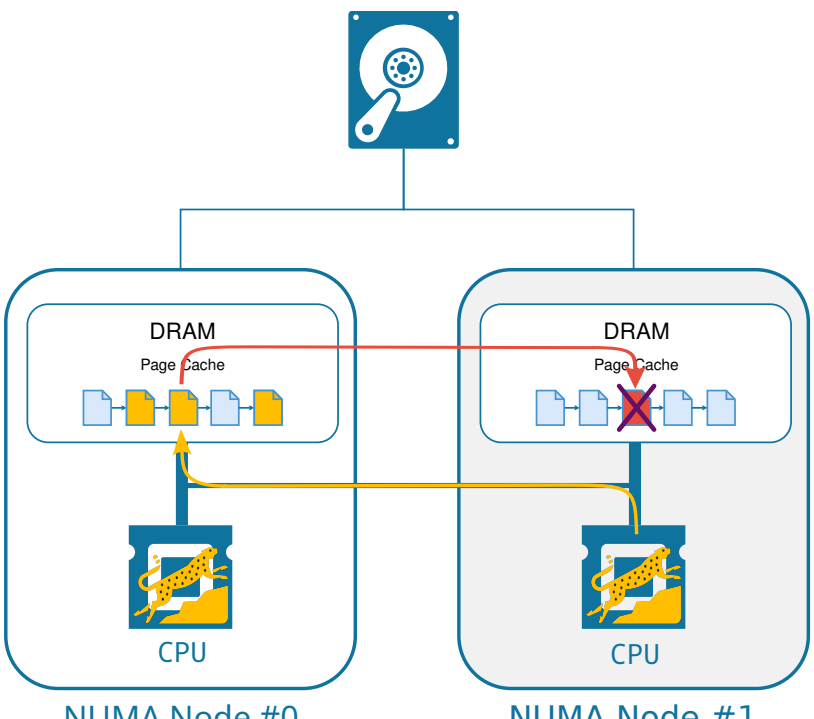
How to ensure consistent access to the page cache?

On write:

- **Always** write to the *main* page.
- Batch invalidation: Invalidate all associated *twins* using a single flag switch.

On subsequent read:

- On-demand refresh: Stale replicas are updated via a direct copy from the *main*.



Mechanism 2 : Consistency

Problem

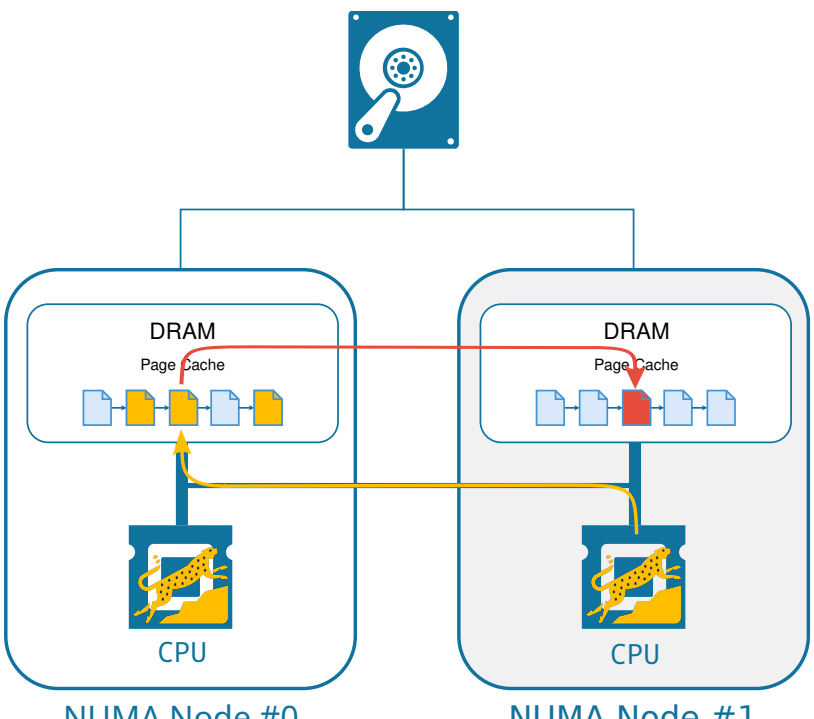
How to ensure consistent access to the page cache?

On write:

- **Always** write to the *main* page.
- Batch invalidation: Invalidate all associated *twins* using a single flag switch.

On subsequent read:

- On-demand refresh: Stale replicas are updated via a direct copy from the *main*.



Mechanism 2 : Consistency

Problem

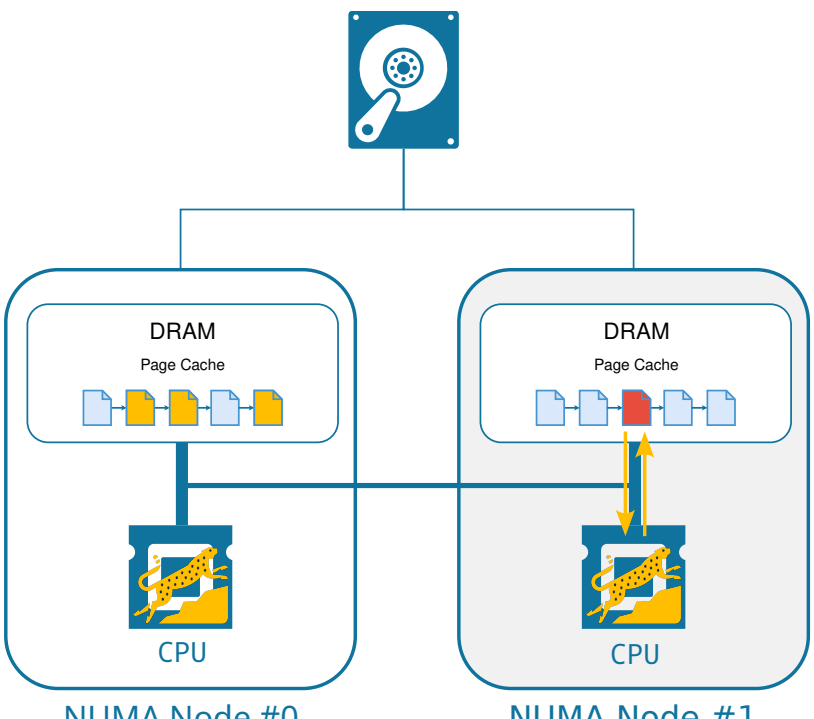
How to ensure consistent access to the page cache?

On write:

- **Always** write to the *main* page.
- Batch invalidation: Invalidate all associated *twins* using a single flag switch.

On subsequent read:

- On-demand refresh: Stale replicas are updated via a direct copy from the *main*.
- Local access: Read is local again, the system maintains consistency while reducing latency.



Mechanism 3 : Local Write Optimization

Problem

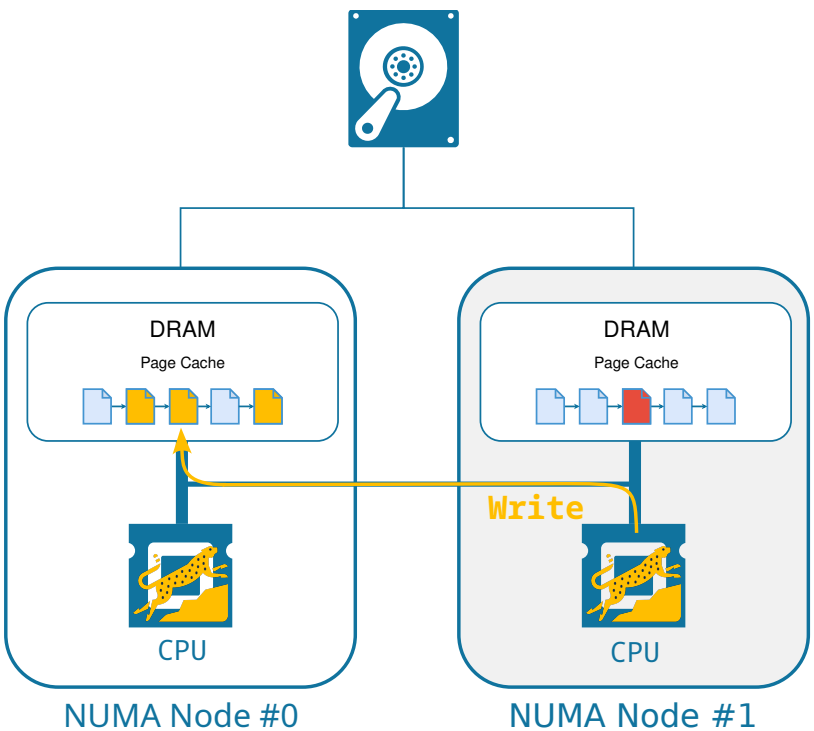
How to minimize the cost of the consistency algorithm?

Mechanism 3 : Local Write Optimization

Problem

How to minimize the cost of the consistency algorithm?

PaCaR monitors excessive remote write operations:



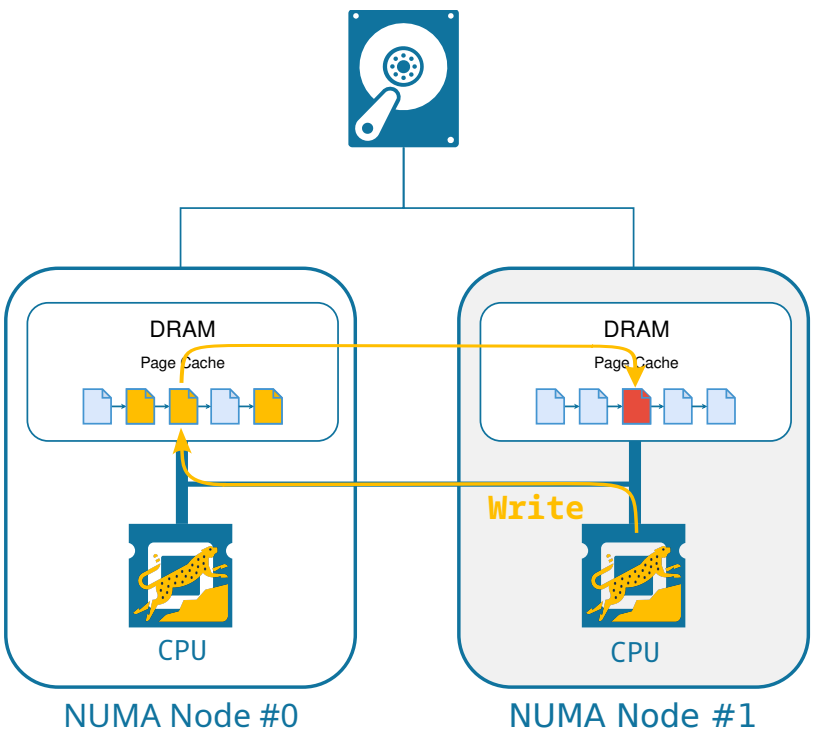
Mechanism 3 : Local Write Optimization

Problem

How to minimize the cost of the consistency algorithm?

PaCaR monitors excessive remote write operations:

- Upon detection, switch *main* and *twin*



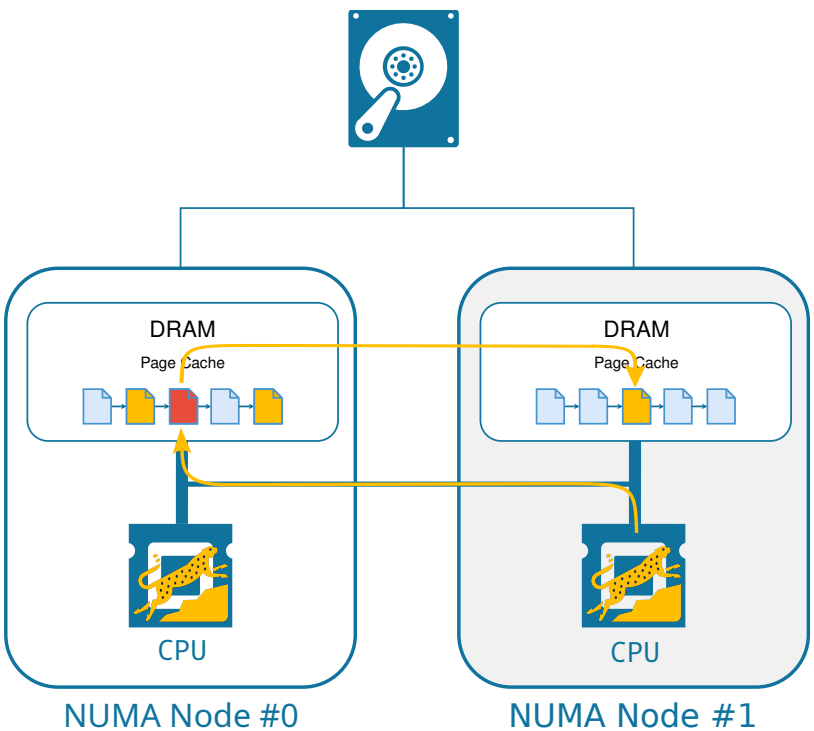
Mechanism 3 : Local Write Optimization

Problem

How to minimize the cost of the consistency algorithm?

PaCaR monitors excessive remote write operations:

- Upon detection, switch *main* and *twin*
- The *twin* becomes the new *main*



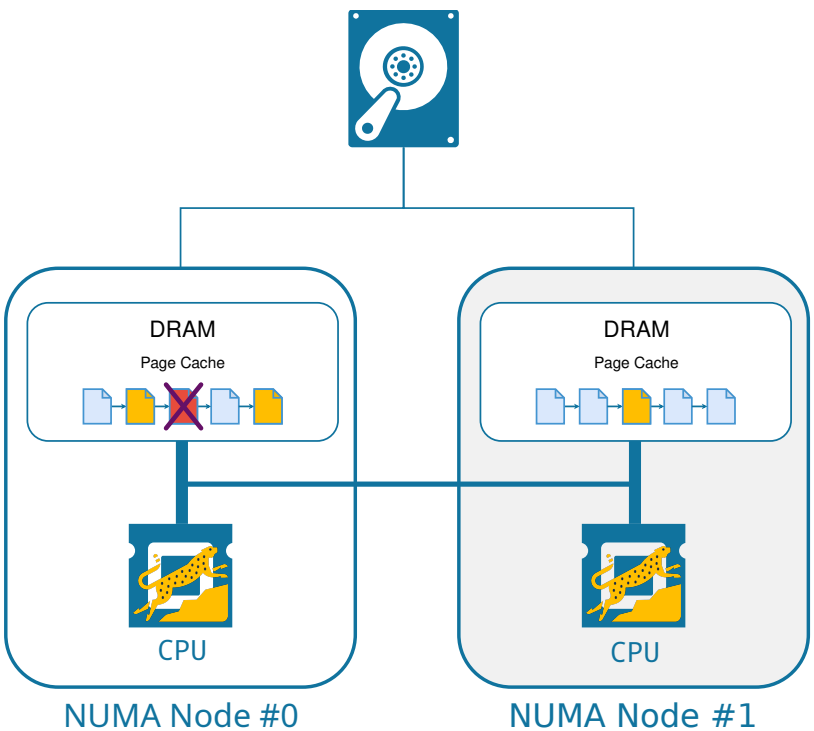
Mechanism 3 : Local Write Optimization

Problem

How to minimize the cost of the consistency algorithm?

PaCaR monitors excessive remote write operations:

- Upon detection, switch *main* and *twin*
- The *twin* becomes the new *main*
- The former *main* becomes an invalidated *twin*



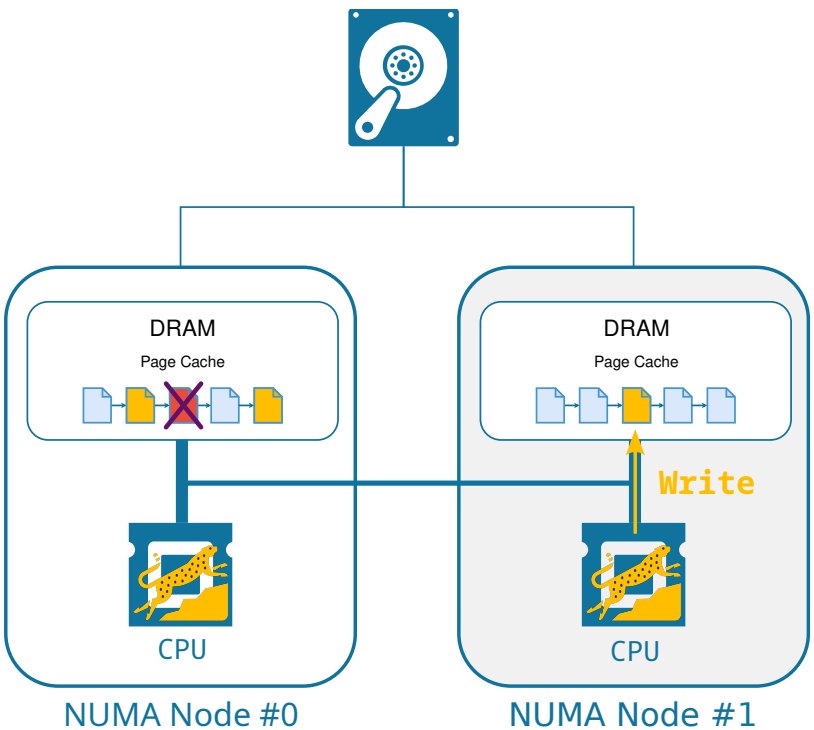
Mechanism 3 : Local Write Optimization

Problem

How to minimize the cost of the consistency algorithm?

PaCaR monitors excessive remote write operations:

- Upon detection, switch *main* and *twin*
- The *twin* becomes the new *main*
- The former *main* becomes an invalidated *twin*
- The write is now local



Mechanism 4 : Memory Footprint Limitation

Problem

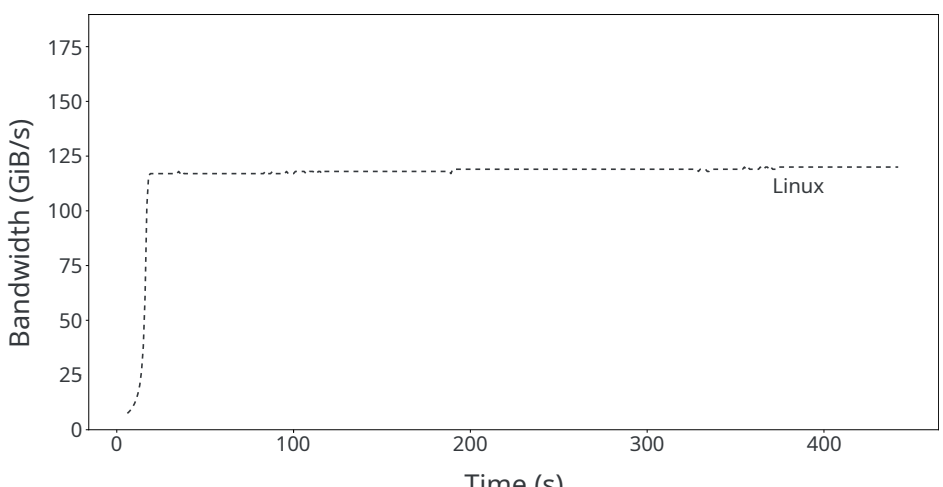
How to replicate the page cache without disrupting Linux memory management?

Mechanism 4 : Memory Footprint Limitation

Problem

How to replicate the page cache without disrupting Linux memory management?

Let's run an I/O-intensive and measure the I/O bandwidth over time

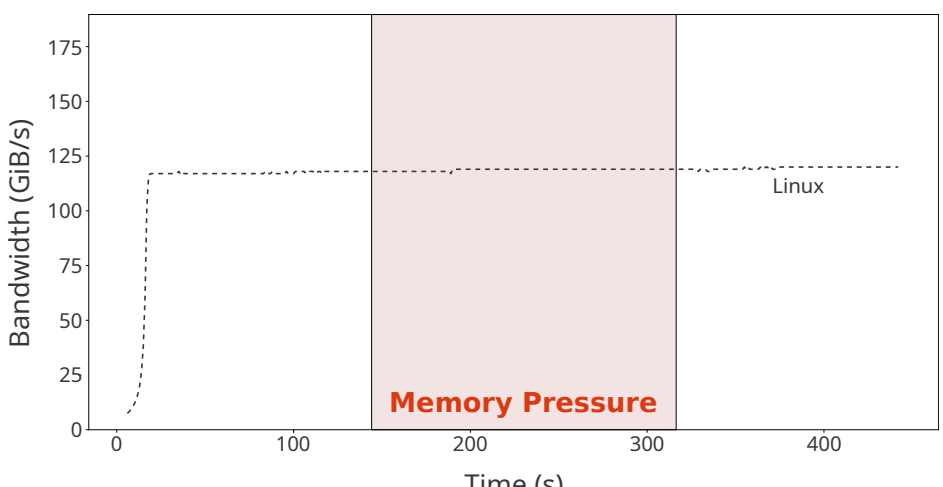


Mechanism 4 : Memory Footprint Limitation

Problem

How to replicate the page cache without disrupting Linux memory management?

Let's run an I/O-intensive and measure the I/O bandwidth over time, under memory pressure.



Mechanism 4 : Memory Footprint Limitation

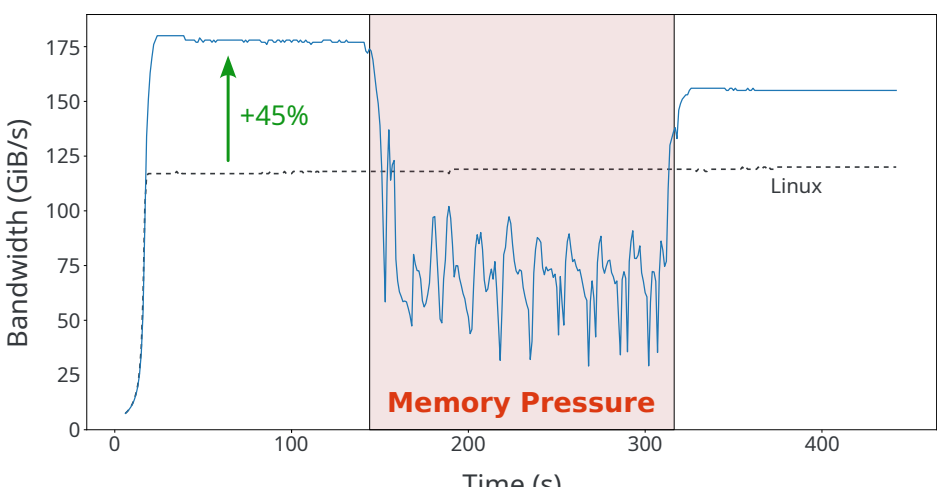
Problem

How to replicate the page cache without disrupting Linux memory management?

Let's run an I/O-intensive and measure the I/O bandwidth over time, under memory pressure.

PaCaR:

- Limit replication to remote-accessed pages
- Integrate with Linux's LRU list to evict cold pages



Mechanism 4 : Memory Footprint Limitation

Problem

How to replicate the page cache without disrupting Linux memory management?

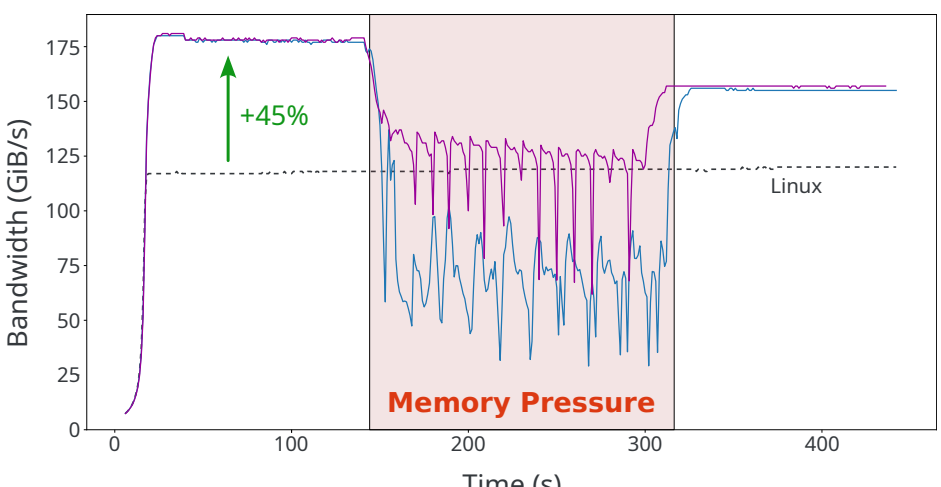
Let's run an I/O-intensive and measure the I/O bandwidth over time, under memory pressure.

PaCaR:

- Limit replication to remote-accessed pages
- Integrate with Linux's LRU list to evict cold pages

Switch on eviction:

- Elect a new *main* from among the *twins* during eviction



Mechanism 4 : Memory Footprint Limitation

Problem

How to replicate the page cache without disrupting Linux memory management?

Let's run an I/O-intensive and measure the I/O bandwidth over time, under memory pressure.

PaCaR:

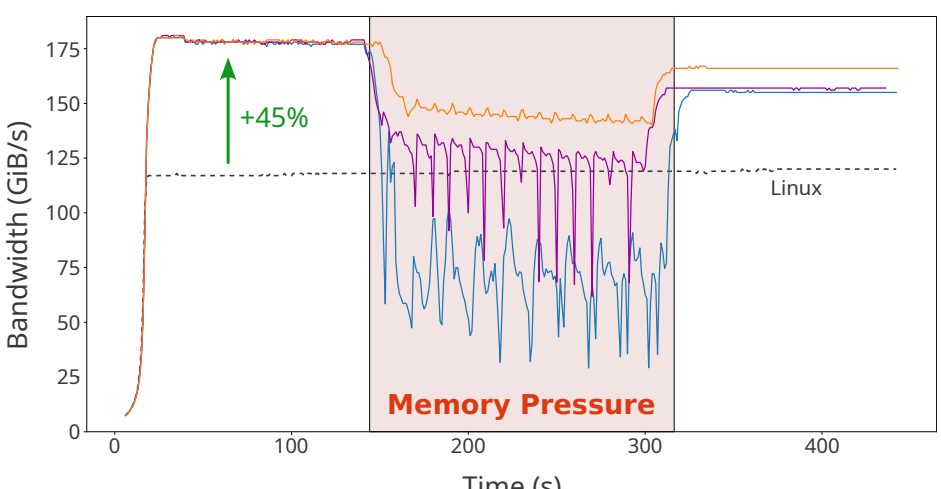
- Limit replication to remote-accessed pages
- Integrate with Linux's LRU list to evict cold pages

Switch on eviction:

- Elect a new *main* from among the *twins* during eviction

Pressure mitigation:

- Halt replication under heavy memory pressure



Mechanism 4 : Memory Footprint Limitation

Problem

How to replicate the page cache without disrupting Linux memory management?

Let's run an I/O-intensive and measure the I/O bandwidth over time, under memory pressure.

PaCaR:

- Limit replication to remote-accessed pages
- Integrate with Linux's LRU list to evict cold pages

Switch on eviction:

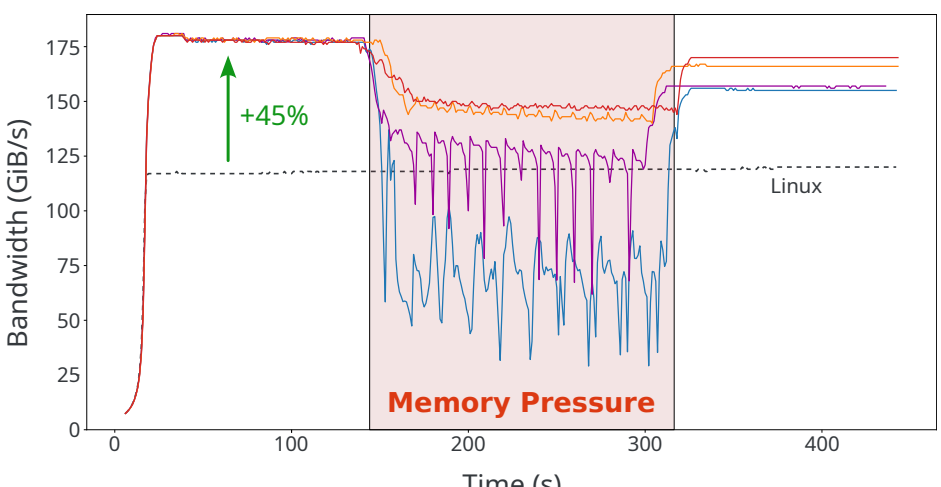
- Elect a new *main* from among the *twins* during eviction

Pressure mitigation:

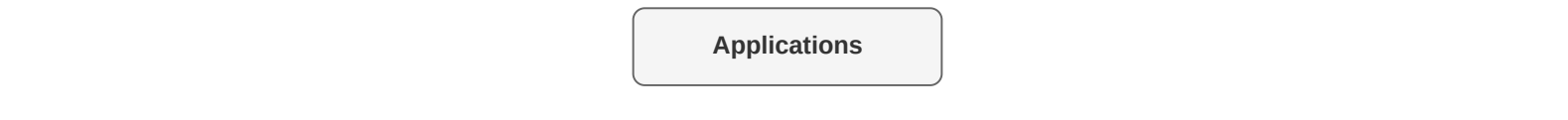
- Halt replication under heavy memory pressure

Combined mechanisms:

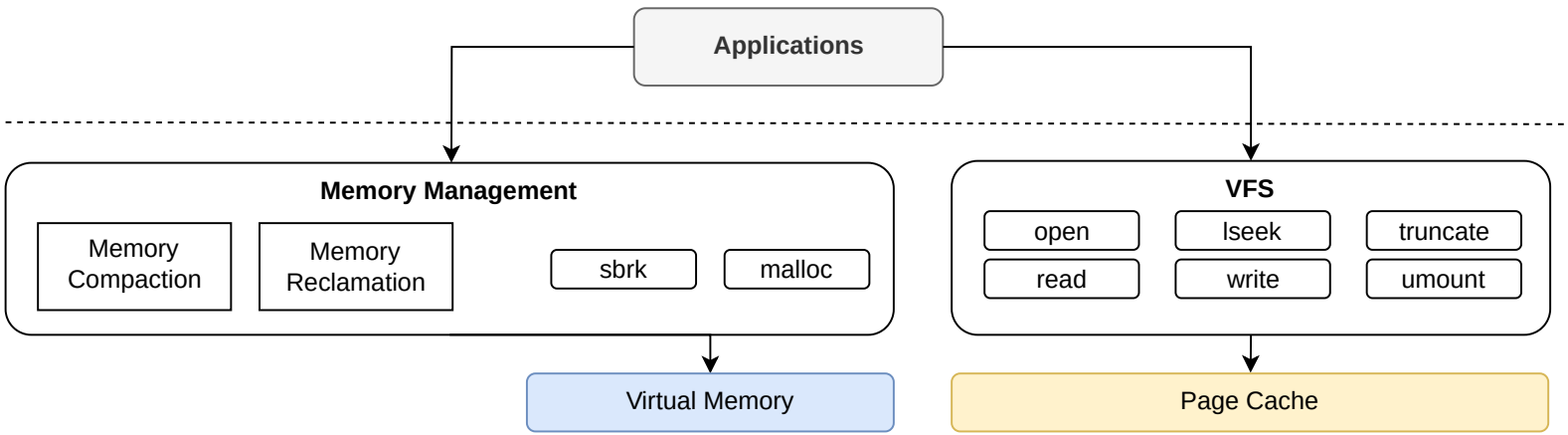
- Preserve efficiency even under memory pressure



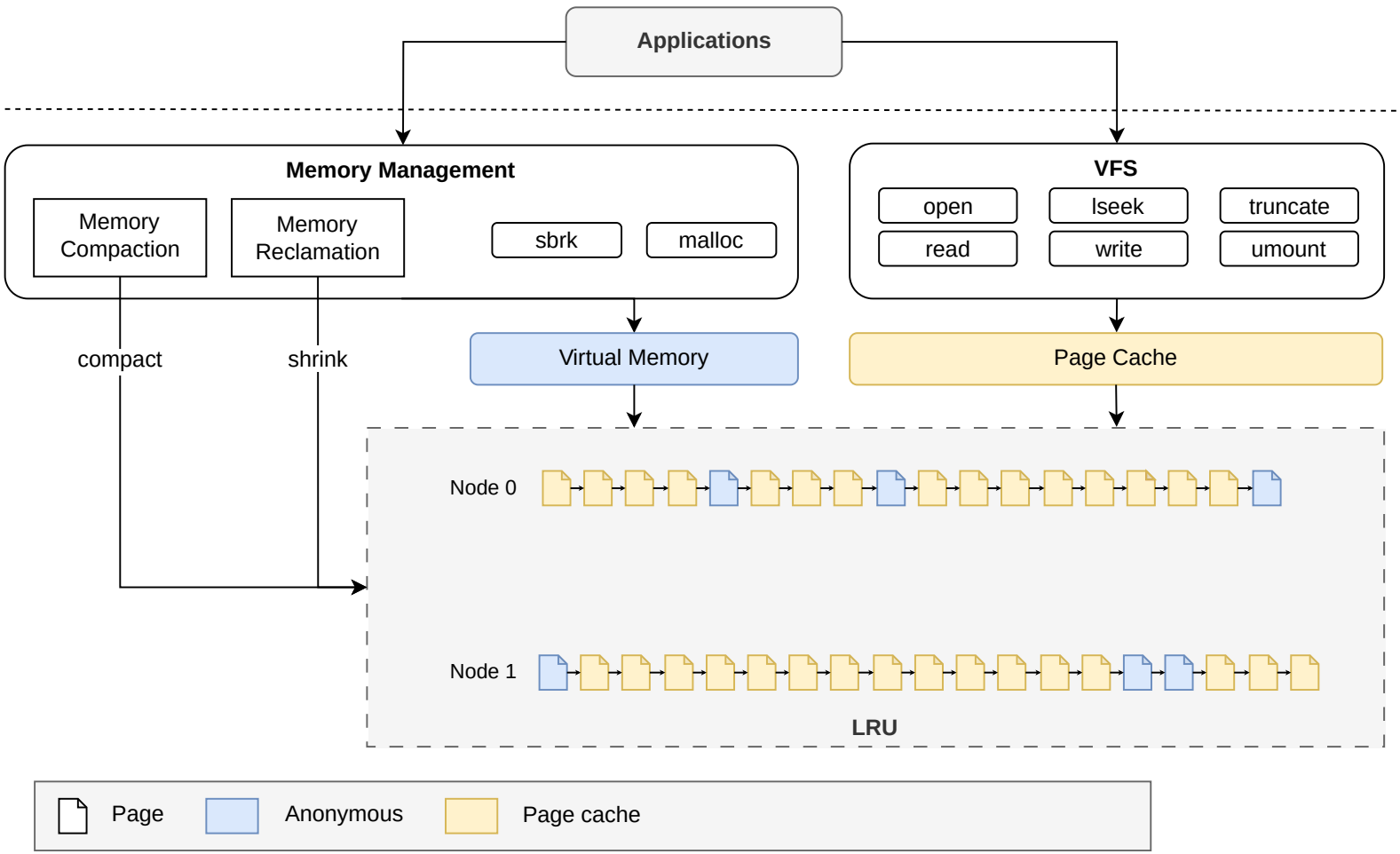
PaCaR Technical Implementation Overview



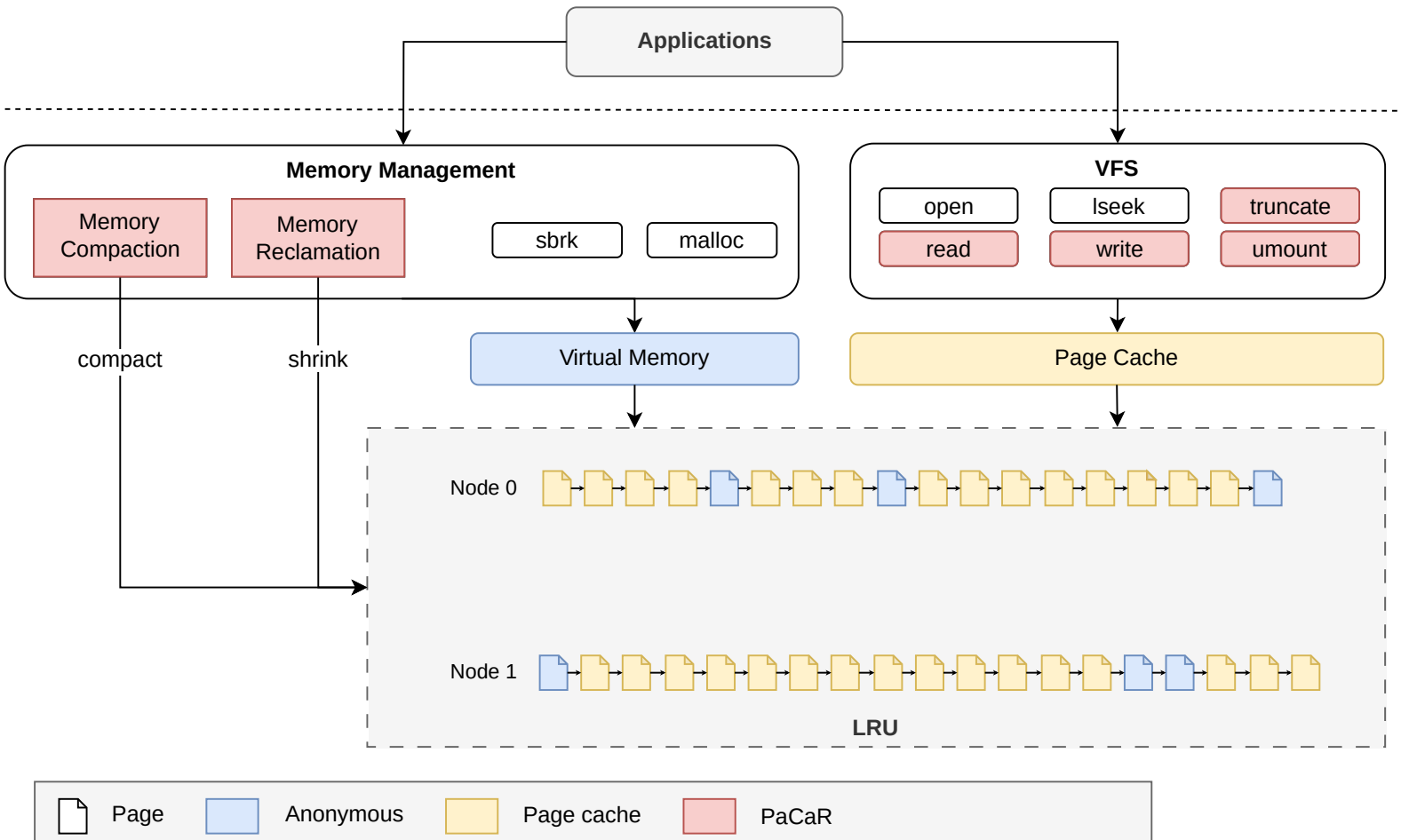
PaCaR Technical Implementation Overview



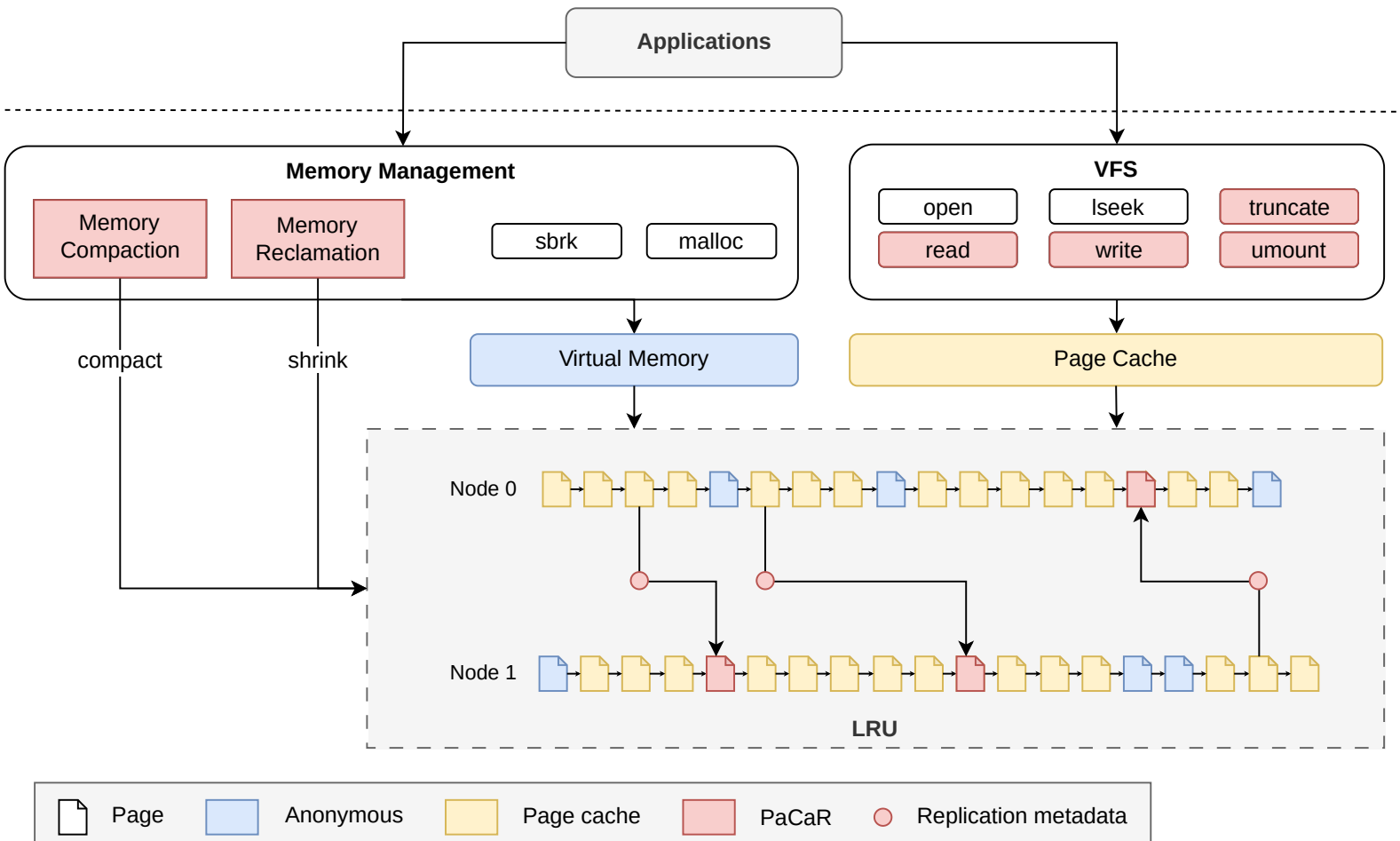
PaCaR Technical Implementation Overview



PaCaR Technical Implementation Overview



PaCaR Technical Implementation Overview



Evaluation

Test bench :

- NUMA: 2x Intel(R) Xeon(R) Silver 4410Y
- 24 threads per node, 256 GiB total memory

Comparison: :

- **Baseline :** Linux Kernel LTS 6.12 + default NUMA balancing
- **PaCaR :** Same kernel + PaCaR + all mechanisms enabled

Evaluation

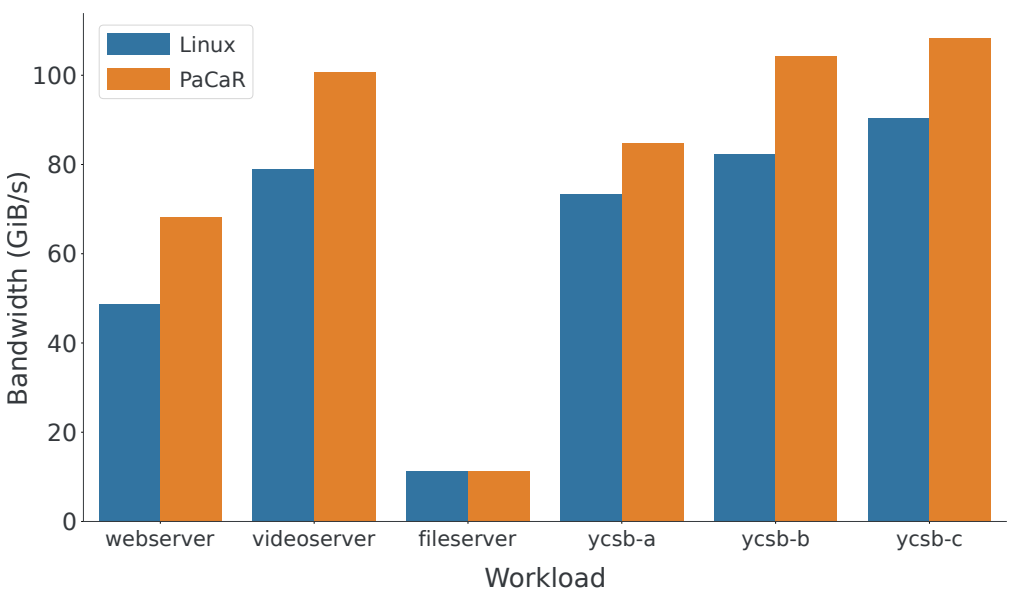
Test bench :

- NUMA: 2x Intel(R) Xeon(R) Silver 4410Y
- 24 threads per node, 256 GiB total memory

Comparison: :

- **Baseline :** Linux Kernel LTS 6.12 + default NUMA balancing
- **PaCaR :** Same kernel + PaCaR + all mechanisms enabled

I/O-intensive applications with Filebench



Evaluation

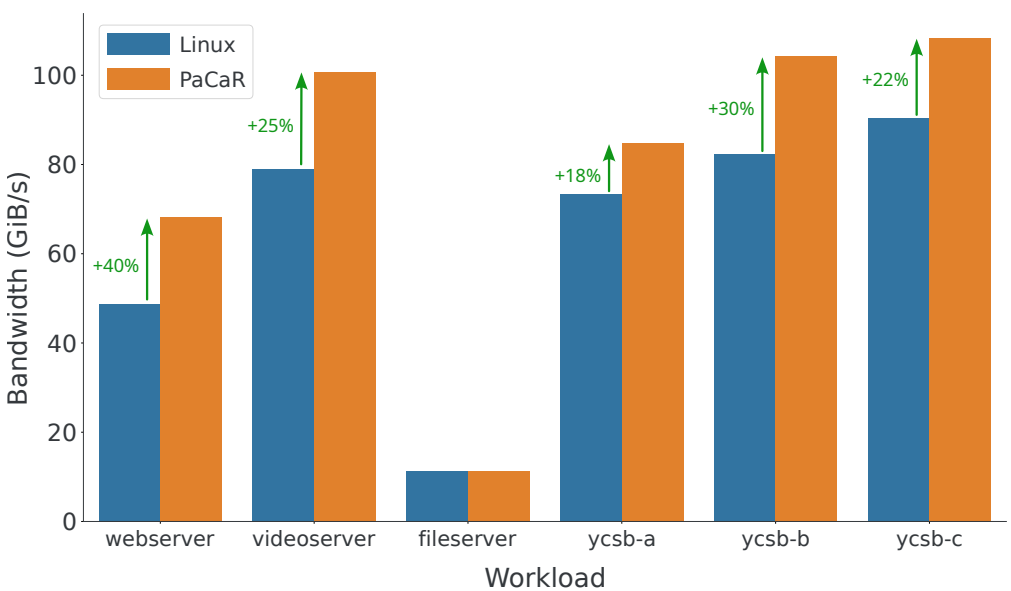
Test bench :

- NUMA: 2x Intel(R) Xeon(R) Silver 4410Y
- 24 threads per node, 256 GiB total memory

Comparison: :

- **Baseline :** Linux Kernel LTS 6.12 + default NUMA balancing
- **PaCaR :** Same kernel + PaCaR + all mechanisms enabled

I/O-intensive applications with Filebench



Evaluation

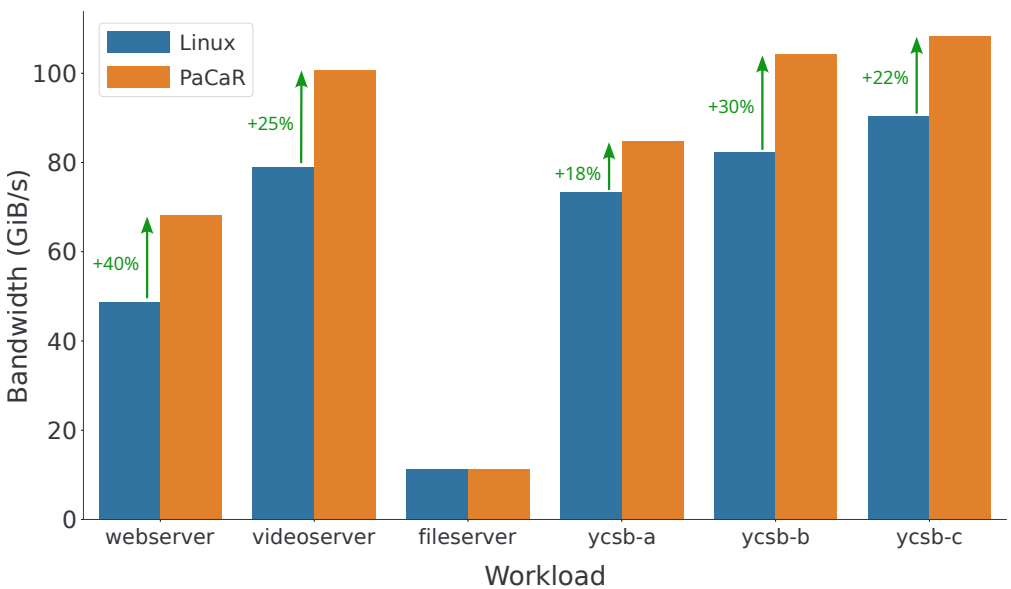
Test bench :

- NUMA: 2x Intel(R) Xeon(R) Silver 4410Y
- 24 threads per node, 256 GiB total memory

Comparison: :

- **Baseline :** Linux Kernel LTS 6.12 + default NUMA balancing
- **PaCaR :** Same kernel + PaCaR + all mechanisms enabled

I/O-intensive applications with Filebench



RocksDB's dbbench

Workload	Operations	Bandwidth
multiread random	**+7.9%**	**+8.0%**
update random	±0.0%	−0.7%
append random	−0.1%	−0.3%

Evaluation

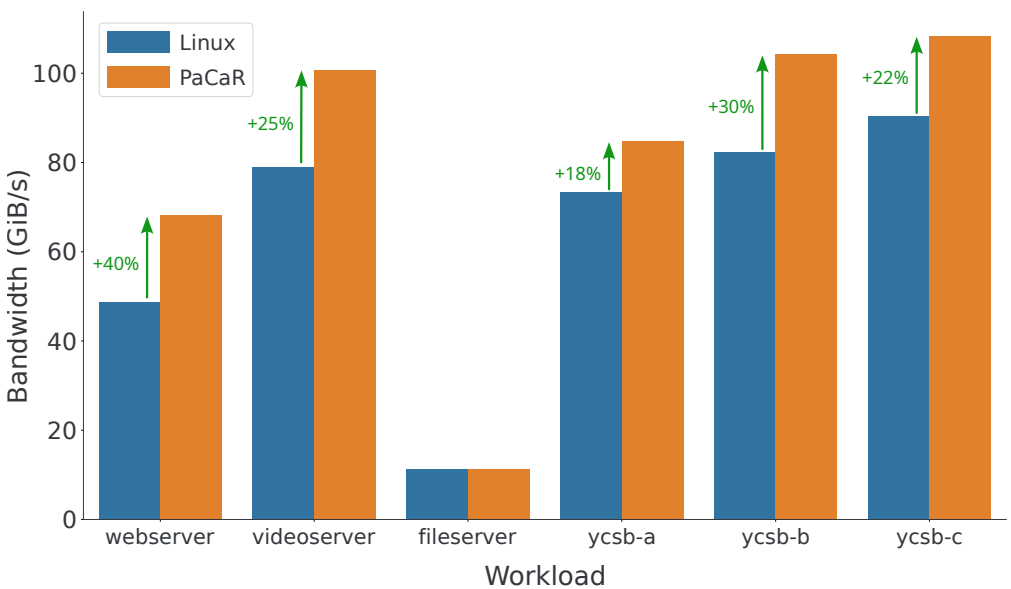
Test bench :

- NUMA: 2x Intel(R) Xeon(R) Silver 4410Y
- 24 threads per node, 256 GiB total memory

Comparison: :

- **Baseline :** Linux Kernel LTS 6.12 + default NUMA balancing
- **PaCaR :** Same kernel + PaCaR + all mechanisms enabled

I/O-intensive applications with Filebench



RocksDB's dbbench

Workload	Operations	Bandwidth
multiread random	**+7.9%**	**+8.0%**
update random	±0.0%	−0.7%
append random	−0.1%	−0.3%

- Improved bandwidth on read-heavy synthetic and real applications
- No impact on write-heavy workloads

Conclusion

PaCaR makes the page cache NUMA-aware:

- Dynamic page cache replication
- Consistent writes
- Integration with Linux's memory management
- In-kernel implementation for userspace transparency

Conclusion

PaCaR makes the page cache NUMA-aware:

- Dynamic page cache replication
- Consistent writes
- Integration with Linux's memory management
- In-kernel implementation for userspace transparency

Evaluation:

- Read-heavy workloads: Up to 40% improvement in Filebench and 8% in RocksDB
- Write-heavy workloads: Minimal overhead despite our consistency algorithm

Conclusion

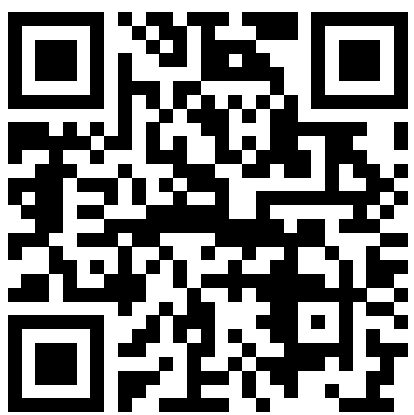
PaCaR makes the page cache NUMA-aware:

- Dynamic page cache replication
- Consistent writes
- Integration with Linux's memory management
- In-kernel implementation for userspace transparency

Evaluation:

- Read-heavy workloads: Up to 40% improvement in Filebench and 8% in RocksDB
- Write-heavy workloads: Minimal overhead despite our consistency algorithm

Code & Paper



PaCaR: Improved Buffered I/O Locality on NUMA Systems with Page Cache Replication, Jérôme Coquisart, Julien Sopena, Redha Gouicem