NTT

Innovative R&D by NTT

Introducing PMDK into PostgreSQL

Challenges and implementations towards PMEM-generation elephant

Takashi Menjo†, Yoshimi Ichiyanagi†
†NTT Software Innovation Center

PGCon 2018, Day 1 (May 31, 2018)

My background

- **Worked on system software**
  - Distributed block storage (Sheepdog)
  - Operating system (Linux)

- **First time to dive into PostgreSQL**
  - Try to refine open-source software by a new storage and a new library
  - Choose PostgreSQL because the NTT group promotes it

Any discussions and comments are welcome :-)

Overview of my talk

1. **Introduction**
   - Persistent Memory (PMEM), DAX for files, and PMDK

2. **Hacks and evaluation**
   i. XLOG segment file (Write-Ahead Logging)
   ii. Relation segment file (Table, Index, ...)

3. **Tips related to PMEM**
   - Programming, benchmark, and operation

4. **Conclusion**

Overview of my talk

1. **Introduction**
   - Persistent Memory (PMEM), DAX for files, and PMDK

2. **Hacks and evaluation**
   i. XLOG segment file (Write-Ahead Logging)
   ii. Relation segment file (Table, Index, ...)

3. **Tips related to PMEM**
   - Programming, benchmark, and operation

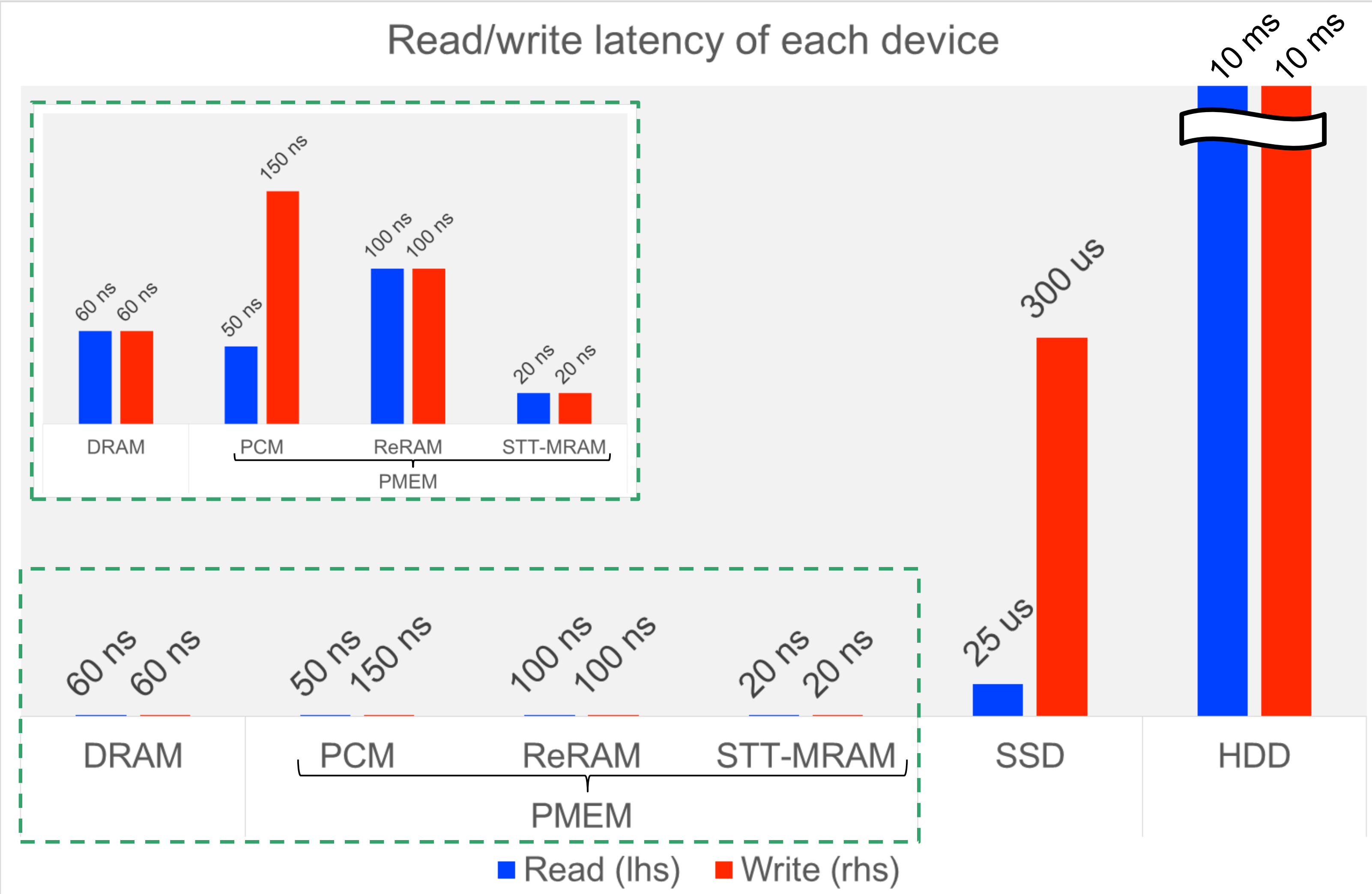
4. **Conclusion**

Persistent Memory (PMEM)

- **Emerging memory-like storage device**
  - Non-volatile, byte-addressable, and as fast as DRAM

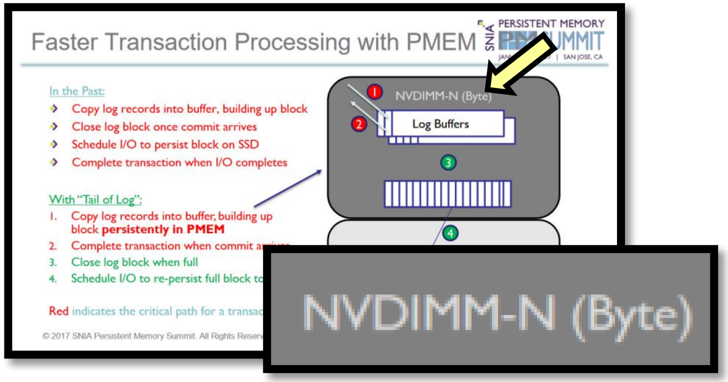
- **Several types released or announced**
  - **NVDIMM-N** (Micron, HPE, Dell, ...) ← We use this.
    - Based on DRAM and NAND flash
  - HPE Scalable Persistent Memory
  - Intel Persistent Memory (in 2018?)

How PMEM is fast

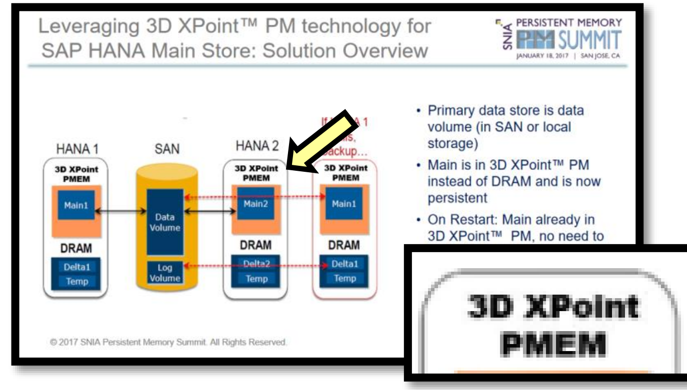


[Source: J. Arulraj and A. Pavlo. How to Build a Non-Volatile Memory Database Management System (Table 1). Proc. SIGMOD '17.]

Databases on the way to PMEM

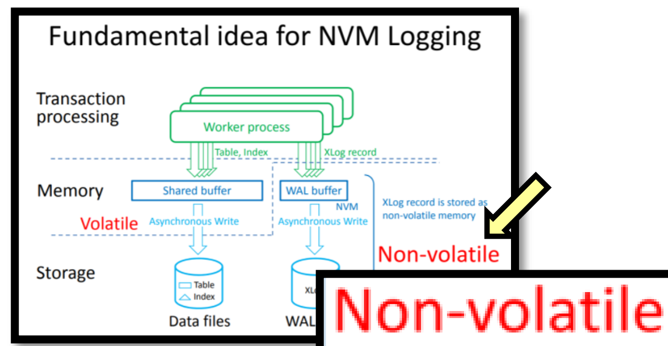


Microsoft SQL Server 2016
utilizes NVDIMM-N for Tail-of-Log [1]



SAP HANA is leveraging
PMEM for Main Store [2]

MariaDB Labs is
collaborating with Intel [3]



"Non-volatile Memory Logging"
in **PGCon 2016** [4]

[1] https://www.snia.org/sites/default/files/PM-Summit/2017/presentations/Tom_Talpey_Persistent_Memory_in_Windows_Server_2016.pdf
[2] https://www.snia.org/sites/default/files/PM-Summit/2017/presentations/Zora_Caklovic_Bringing_Persistent_Memory_Technology_to_SAP_HANA.pdf
[3] https://mariadb.com/about-us/newsroom/press-releases/mariadb-launches-innovation-labs-explore-and-conquer-new-frontiers
[4] https://www.pgcon.org/2016/schedule/attachments/430_Non-volatile_Memory_Logging.pdf

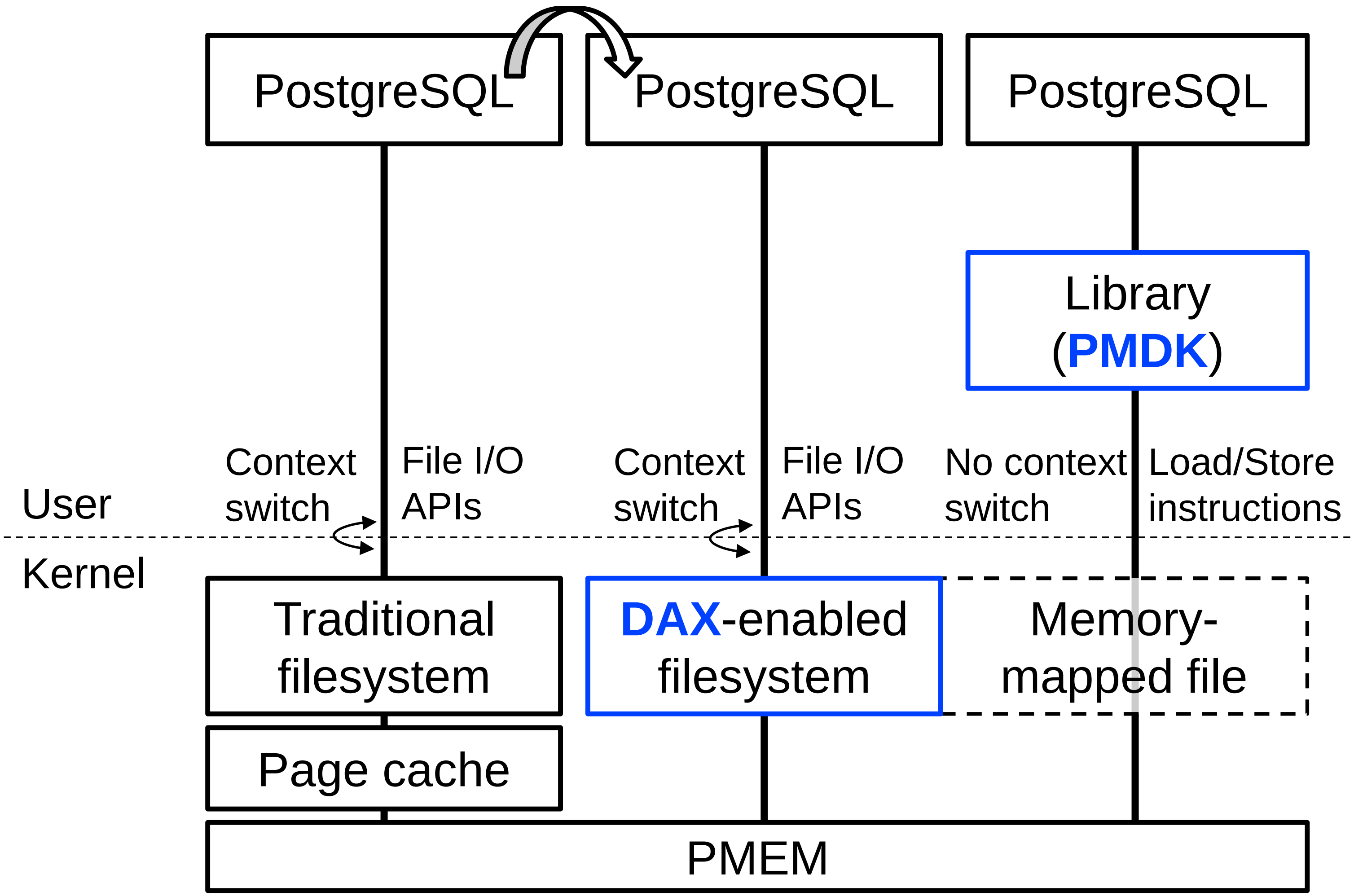
What we need to use PMEM

- **Hardware support**
  - BIOS detecting and configuring PMEM
  - ACPI 6.0 or later: NFIT
  - Asynchronous DRAM Refresh (ADR)
  - :

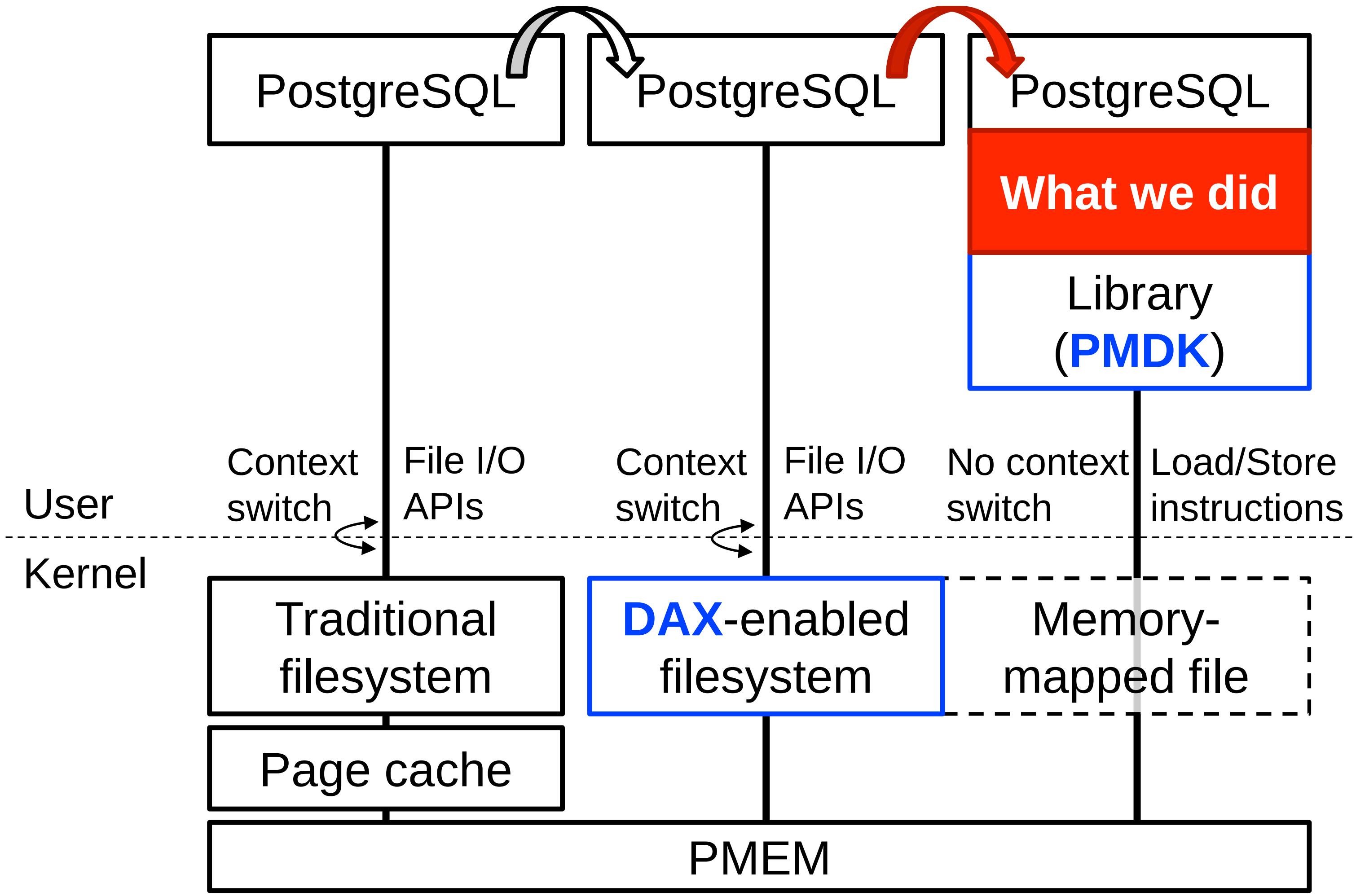
  (ACPI 6.0 and ADR: For NVDIMM, at least.)

- **Software support**
  - Operating system (device drivers)
  - **Direct-Access for files (DAX)**
    - Linux (ext4 and xfs) and Windows (NTFS)
  - **Persistent Memory Development Kit (PMDK)**
    - Linux and Windows, on x64

DAX, PMDK, and what we did



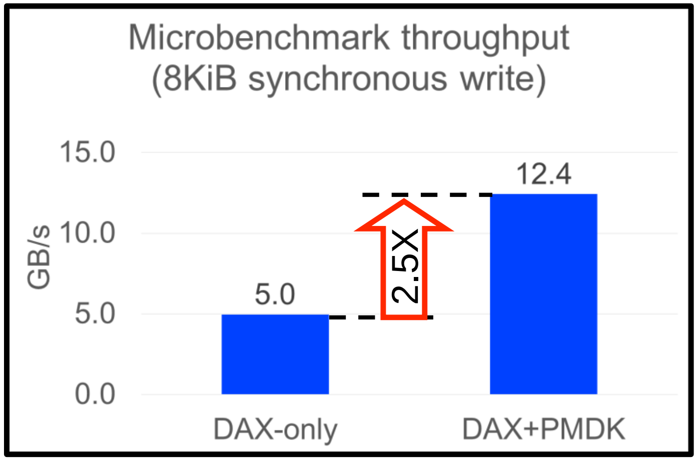
DAX, PMDK, and what we did



Benefits of DAX and PMDK

- **With DAX only**
  - Use PMEM faster without change of the application
- **With DAX and PMDK**
  - Improve the performance of I/O-intensive workload
    - By reducing context switches and the overhead of API calls

- **Micro-benchmark**
  - DAX+PMDK is 2.5X as fast as DAX-only



(HPE NVDIMM, Linux kernel 4.16, ext4, PMDK 1.4)

- **Try to introduce PMDK into PostgreSQL**

Overview of my talk

1. **Introduction**
   - Persistent Memory (PMEM), DAX for files, and PMDK
2. **Hacks and evaluation**
   1. XLOG segment file (Write-Ahead Logging)
   2. Relation segment file (Table, Index, ...)
3. **Tips related to PMEM**
   - Programming, benchmark, and operation
4. **Conclusion**

Approach

- **How to hack**
  - ✓ **Replace read/write calls with memory copy**
    - ➢ Easier way, reasonable for our first step
  - Have data structures on DRAM persist on PMEM directly, similar to in-memory database

- **What we hack**
  - ✓ **i) XLOG segment files**
    - ➢ Critical for transaction performance
  - ✓ **ii) Relation segment files**
    - ➢ Many writes occur during checkpoint
  - Other files (CLOG, pg_control, ...)

How to hack

✓ **Replace read/write calls with memory copy**

	read/write (POSIX)	libpmem (PMDK)
Open	`fd = open (path, ...);`	`pmem = pmem_map_file (path, len, ...);`
Write	`nbytes = write (fd, buf, count);`	`pmem_memcpy_nodrain (pmem, buf, count);`
Sync	`ret = fdatasync(fd);`	`pmem_drain();`
Read	`nbytes = read (fd, buf, count);`	`memcpy // from <string.h> (buf, pmem, count);`
Close	`ret = close(fd);`	`ret = pmem_unmap(pmem, len);`

i) XLOG segment file

- **Contains Write-Ahead Log records**
  - Guarantees durability of updates
    - By having the records persist before committing transaction
  - Fixed length (16-MiB per file)
    - Each file has a monotonically increasing “segment number”

- **Critical for transaction performance**
  - Backend cannot commit a transaction before the commit log record persists on storage
  - A transaction takes less time if the record persists sooner

i) How we hack XLOG

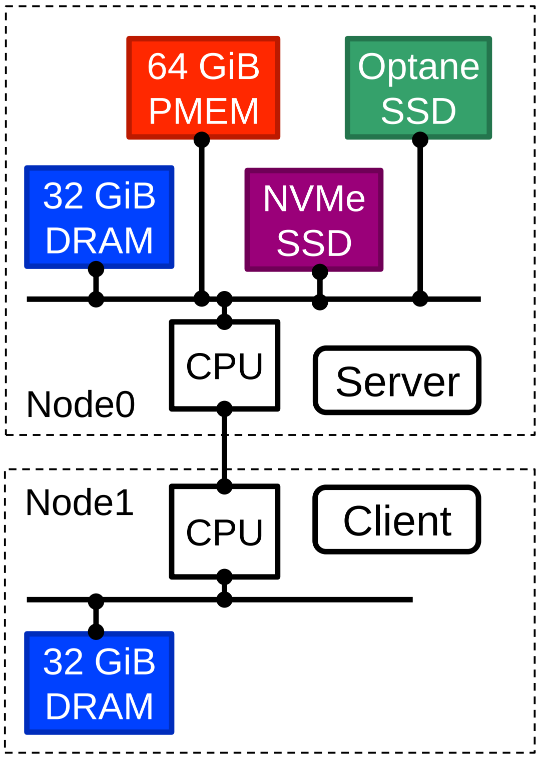
- **Memory-map every segment file**
  - Fixed-length (16-MiB) file is highly compatible with memory-mapping
- **Memory-copy to it from the XLOG buffer**

- **Patch <backend/access/xlog.c> and so on**
  - 15 files changed, 847 insertions, 174 deletions
  - **Available on pgsql-hackers mailing list** (search “PMDK”)

i) Evaluation setup

Hardware	
CPU	E5-2667 v4 x 2 (8 cores per node)
DRAM	[Node0/1] 32 GiB each
PMEM (NVDIMM-N)	[Node0] 64 GiB (HPE 8GB NVDIMM x 8)
NVMe SSD	[Node0] Intel SSD DC P3600 400GB
Optane SSD	[Node0] Intel Optane SSD DC 4800X 750GB
Software	
Distro	Ubuntu 17.10
Linux kernel	4.16
PMDK	1.4
Filesystem	ext4 (DAX available)
PostgreSQL base	a467832 (master @ Mar 18, 2018)
postgresql.conf	
{max,min}_wal_size	20GB
shared_buffers	16085MB
checkpoint_timeout	12min
checkpoint_completion_target	0.7

NUMA node



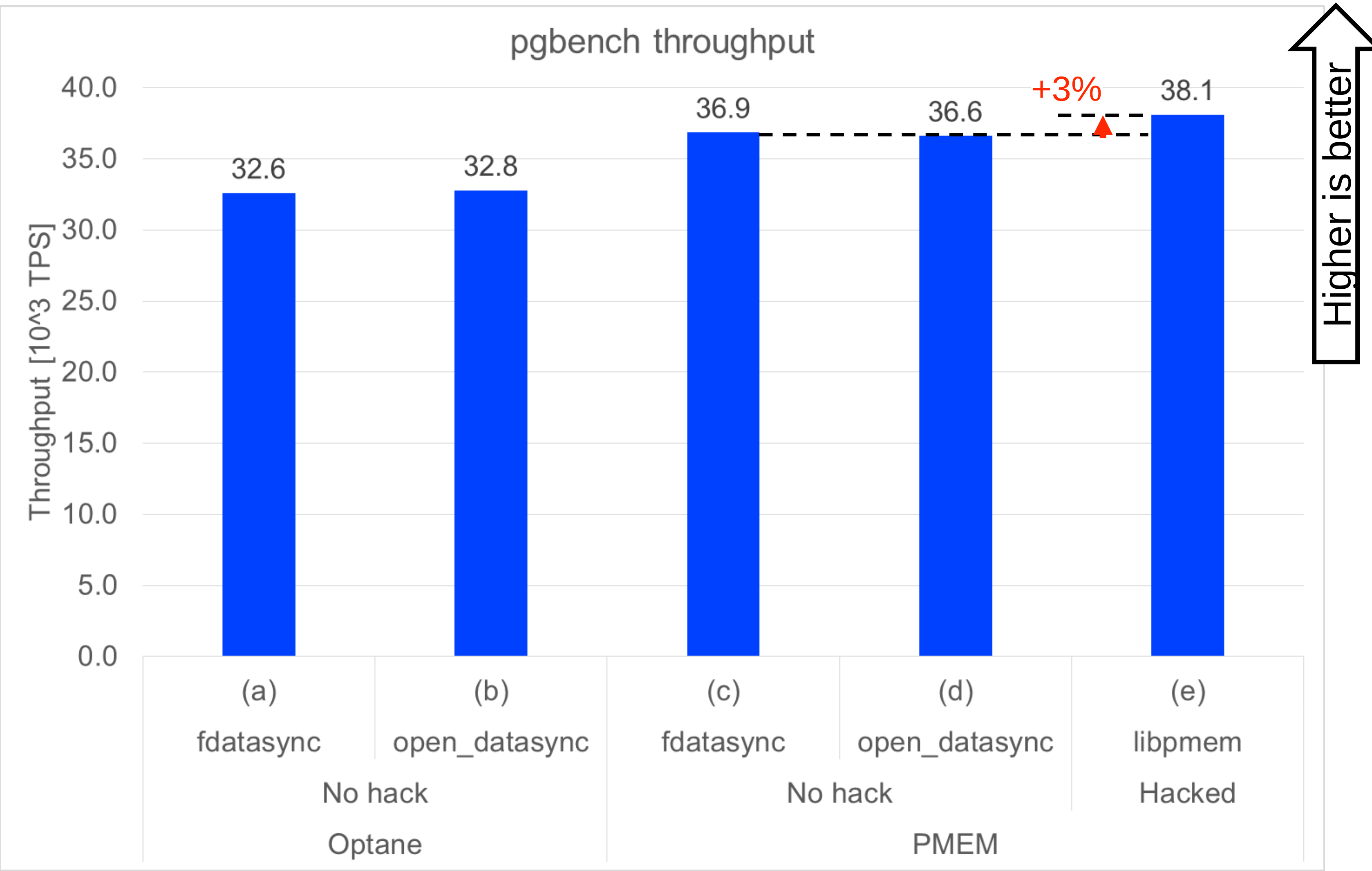
i) Evaluation of XLOG hacks

- **Compare transaction throughput by using pgbench**
  - pgbench -i -s 200
  - pgbench -M prepared -h /tmp -p 5432 -c 32 -j 32 -T 1800
    - The checkpoint runs twice during the benchmark
    - Run 3 times to find the median value for the result

- **Conditions:**

	(a)	(b)	(c)	(d)	(e)
wal_sync_method	fdatasync	open_datasync	fdatasync	open_datasync	**libpmem (New)**
PostgreSQL	No hack		No hack		**Hacked with PMDK**
FS for XLOG	ext4 (No DAX)		ext4 (DAX enabled)		
Device for XLOG	Optane SSD		PMEM		
PGDATA	NVMe SSD / ext4 (No DAX)				

i) Results



pgbench throughput
Throughput [10^3 TPS]
40.0
35.0
30.0
25.0
20.0
15.0
10.0
5.0
0.0
32.6
32.8
36.9
36.6
38.1
+3%
Higher is better
(a)
fdatasync
(b)
open_datasync
(c)
fdatasync
(d)
open_datasync
(e)
libpmem
No hack
No hack
Hacked
Optane
PMEM

i) Discussion

- **Improve transaction throughput by 3%**
  - Roughly the same improvement as Yoshimi reported on pgsql-hackers
  - Seems small in the percentage, but not-so-small in the absolute value (+1,200 TPS)

- **Future work**
  - Performance profiling
  - Searching for a query pattern for which our hack is more effective

Abstract:

> PMEM. The result show that, in regard to WAL, we achieve up to 1.8x more TPS in customized INSERT-oriented benchmark. We propose the patches containing approx.

NTT

Overview of my talk

1. **Introduction**
   - Persistent Memory (PMEM), DAX for files, and PMDK

2. **Hacks and evaluation**
   1. XLOG segment file (Write-Ahead Logging)
   2. Relation segment file (Table, Index, ...)

3. **Tips related to PMEM**
   - Programming, benchmark, and operation

4. **Conclusion**

ii) Relation segment file

- **So-called data file (or checkpoint file)**
  - Table, Index, TOAST, Materialized View, ...
  - Variable length up to 1-GiB
    - A huge table and so forth consist of multiple segment files

- **Critical for checkpoint duration**
  - Dirty pages on the shared buffer are written back to the segment files
  - A checkpoint takes less time if the pages are written back sooner

ii) How we hack Relation

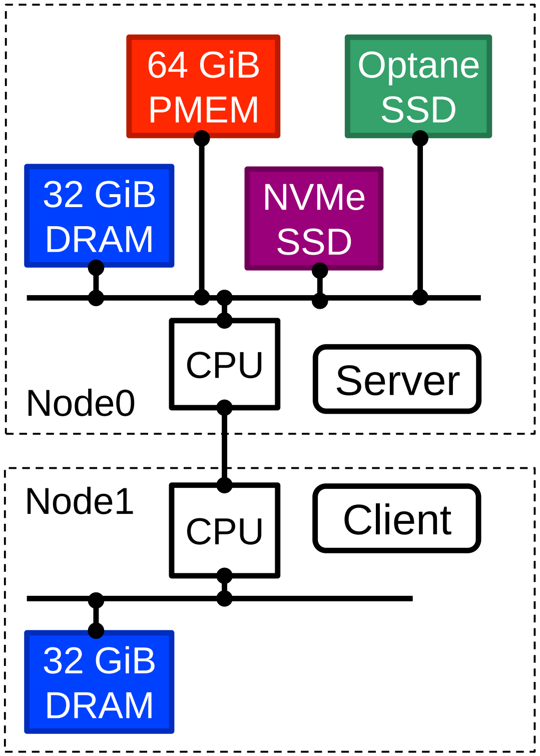
- **Memory-map only every 1-GiB segment file**
  - Memory-mapped file cannot extend or shrink
  - Remapping the file seems difficult for me to implement
- **Memory-copy to it from the shared buffer**

- **Patch <backend/storage/smgr/md.c> and so on**
  - 2 files changed, 152 insertions
  - Under test, not published yet

ii) Evaluation setup

Hardware	
CPU	E5-2667 v4 x 2 (8 cores per node)
DRAM	[Node0/1] 32 GiB each
PMEM (NVDIMM-N)	[Node0] 64 GiB (HPE 8GB NVDIMM x 8)
NVMe SSD	[Node0] Intel SSD DC P3600 400GB
Optane SSD	[Node0] Intel Optane SSD DC 4800X 750GB
Software	
Distro	Ubuntu 17.10
Linux kernel	4.16
PMDK	1.4
Filesystem	ext4 (DAX available)
PostgreSQL base	a467832 (master @ Mar 18, 2018)
postgresql.conf	
{max,min}_wal_size	20GB
shared_buffers	16085MB
checkpoint_timeout	1d
checkpoint_completion_target	0.0

NUMA node



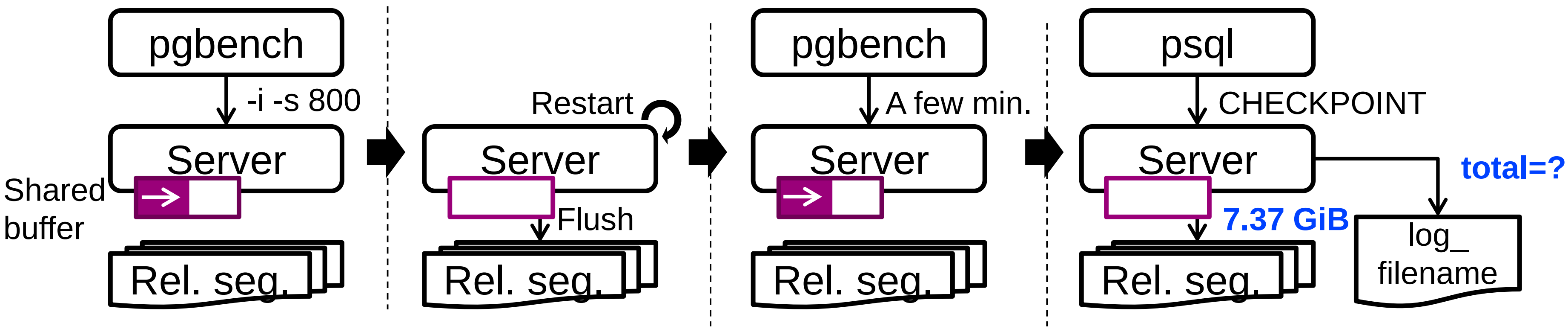
<= Not to kick ckpt automatically

<= To complete ckpt ASAP

NTT

ii) Evaluation of Relation hacks

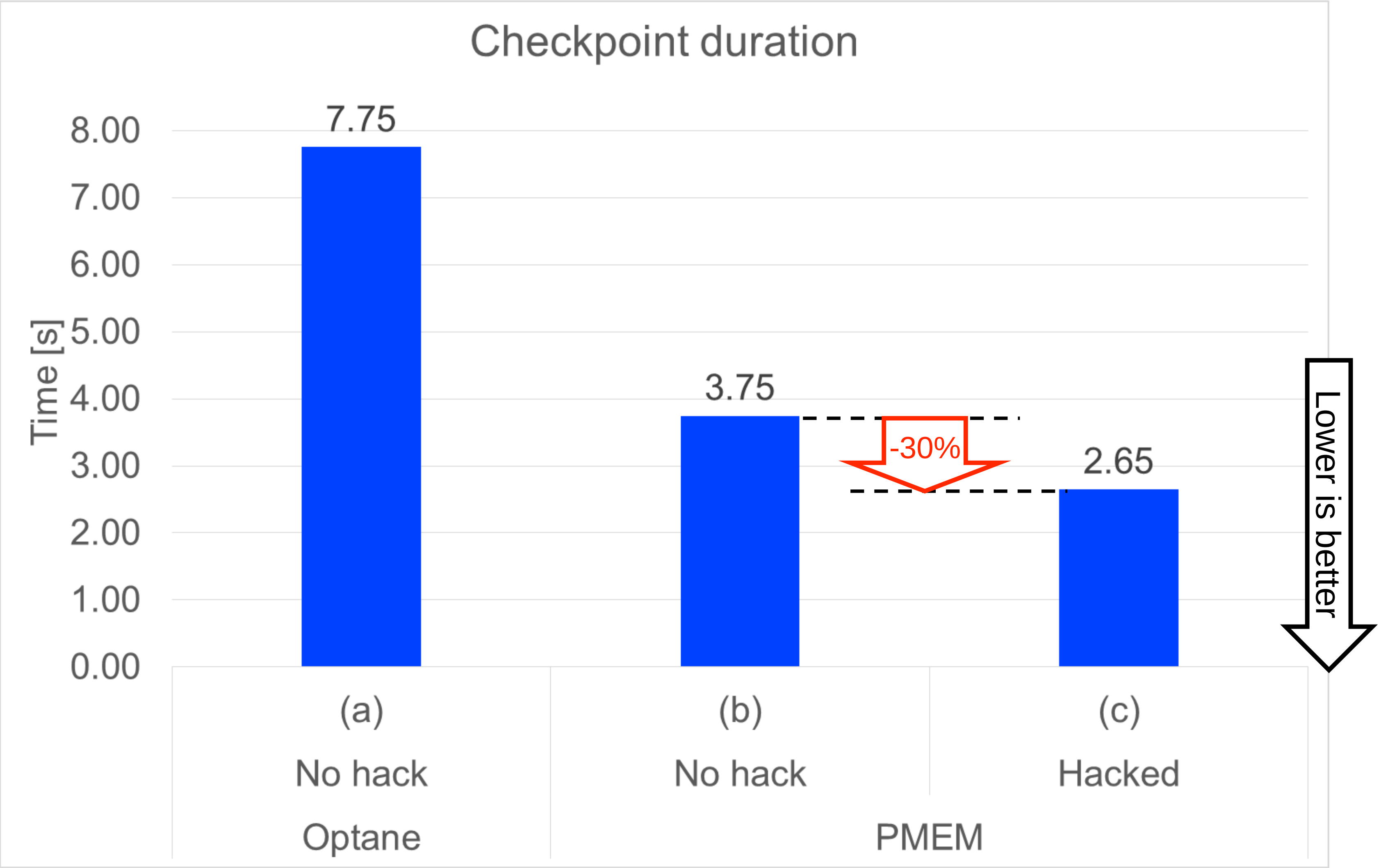
- **Compare checkpoint duration time as follows:**



- **Conditions:**

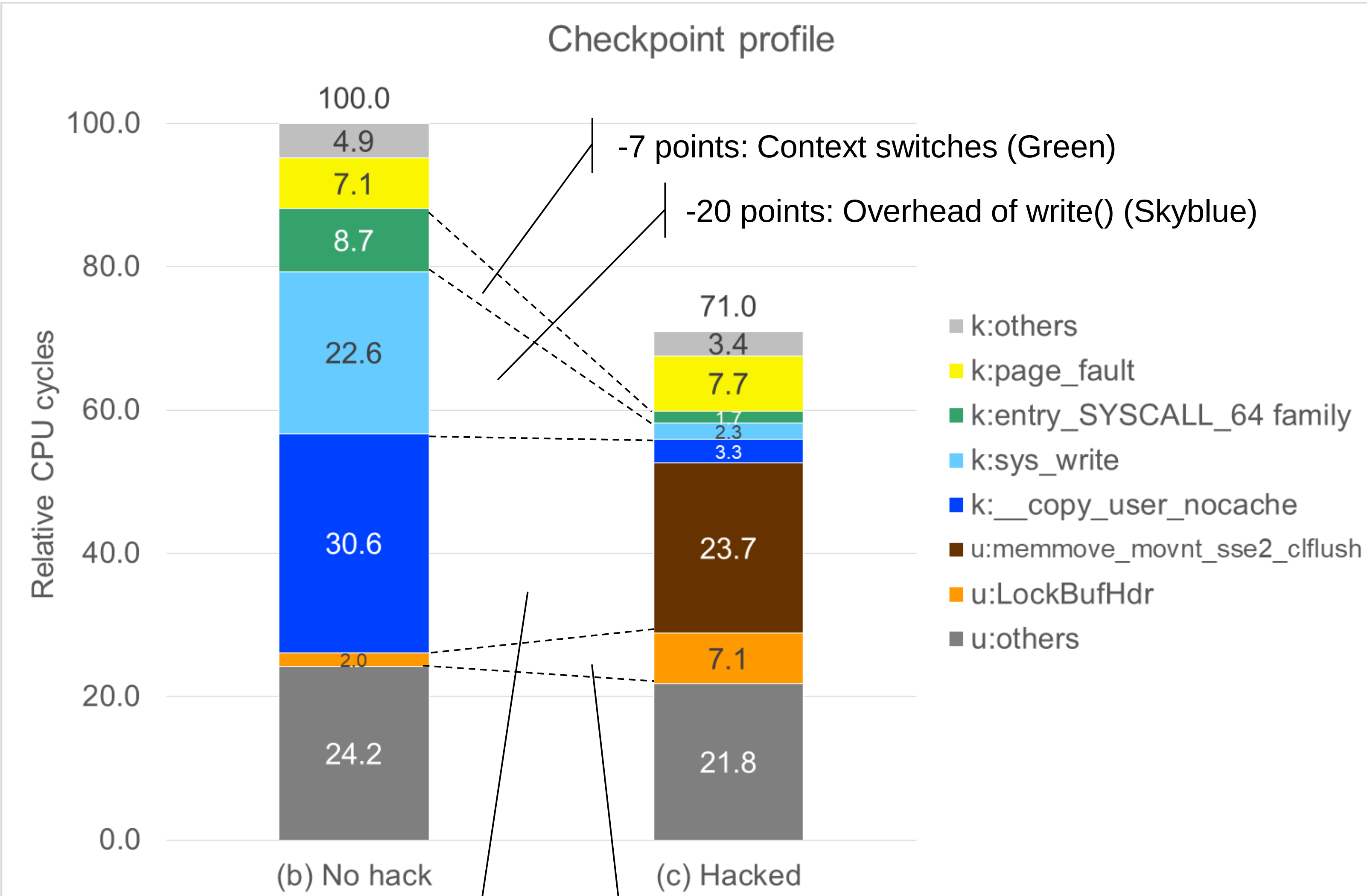
	(a)	(b)	(c)
PostgreSQL	No hack	No hack	**Hacked with PMDK**
FS for PGDATA	ext4 (No DAX)	ext4 (DAX enabled)	
Device for PGDATA	Optane SSD	PMEM	
Profile by Linux perf?	No	Yes	

ii) Results



Checkpoint duration
Time [s]
8.00
7.00
6.00
5.00
4.00
3.00
2.00
1.00
0.00
7.75
3.75
2.65
-30%
Lower is better
(a)
No hack
Optane
(b)
No hack
(c)
Hacked
PMEM

ii) Profiling



-3 points: Memory copy
(Deep blue and brown)

+5 points: LockBufHdr (Orange)

ii) Discussion

- **Shorten checkpoint duration by 30%**
  - The server can give its computing resource to the other purposes
- **Reduce the overhead of system calls and the context switches**
  - Benefits of using memory-mapped files!
- **The time of LockBufHdr became rather longer**
  - Open issue…

Conclusion of evaluation

- **(i) Improve transaction throughput by 3%**
  - With 1,000-line hack for WAL
- **(ii) Shorten checkpoint duration by 30%**
  - With 150-line hack for Relation
- **We must bring out more potential from PMEM**
  - Not so bad in an easier way, but far from "2.5X" in micro-benchmark
  - I think another way is to have data structures on DRAM persist on PMEM directly

Overview of my talk

1. **Introduction**
   - Persistent Memory (PMEM), DAX for files, and PMDK

2. **Hacks and evaluation**
   i. XLOG segment file (Write-Ahead Logging)
   ii. Relation segment file (Table, Index, ...)

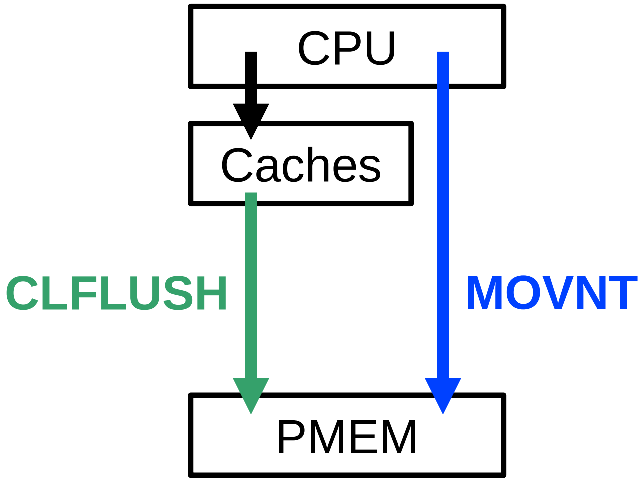
3. **Tips related to PMEM**
   - Programming, benchmark, and operation

4. **Conclusion**

CPU cache flush and cache-bypassing store

- **The data should reach nothing but PMEM**
  - Don't stop at half, volatile middle layer such as CPU caches
  - Or it will be lost when the program or system crashes

- **x64 offers two instruction families**
  - **CLFLUSH** – Flush data out of CPU caches to memory
  - **MOVNT** – Store data to memory, bypassing CPU caches



- **PMDK supports both**
  - pmem_flush
  - pmem_memcpy_nodrain

Memory-mapped file and Relation extension

- **The two are not compatible**
  - Memory-mapped file cannot be extended while being mapped
- **Neither naive way is perfect**
  - Remapping a segment file on extend is time-consuming
  - Pre-allocating maximum size (1GiB per segment) wastes PMEM free space
- **We must rethink traditional data structure towards PMEM era**

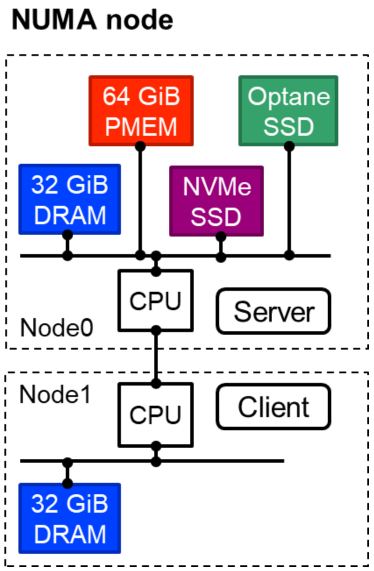
Page table and hugepage

- **Hugepage will improve performance of PMEM**
  - By reducing page walk and Translation Lookaside Buffer (TLB) miss
- **PMDK on x64 considers hugepage**
  - By aligning the mapping address on hugepage boundary (2MiB or 1GiB) when the file is large enough
- **Pre-warming page table for PMEM will also make the performance better**
  - By reducing page fault on main runtime

Controlling NUMA effect

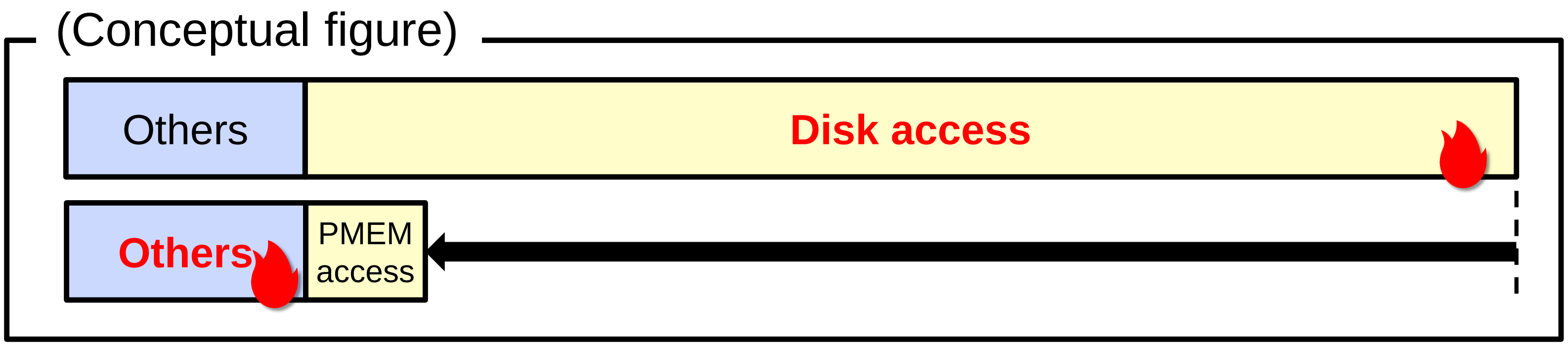
- **Critical for stable performance on multi process-ing system**
  - Accessing to local memory (DRAM and/or PMEM) is fast while remote is slow
    - This applies to PCIe SSD, but PMEM is more sensitive

- **Binding processes to a certain node**
  - numactl --cpunodebind=X --membind=X -- pgctl …



New common sense of hotspot

- **Something other than storage access could be hotspot of transaction when we use PMEM**



- **Such as…**
  - Concurrency control such as locking
  - Redundant internal memory copy
  - Pre-processing such as SQL parse
    - We fell into this trap and avoided it by prepared statement

Overview of my talk

1. **Introduction**
   - Persistent Memory (PMEM), DAX for files, and PMDK

2. **Hacks and evaluation**
   i. XLOG segment file (Write-Ahead Logging)
   ii. Relation segment file (Table, Index, ...)

3. **Tips related to PMEM**
   - Programming, benchmark, and operation

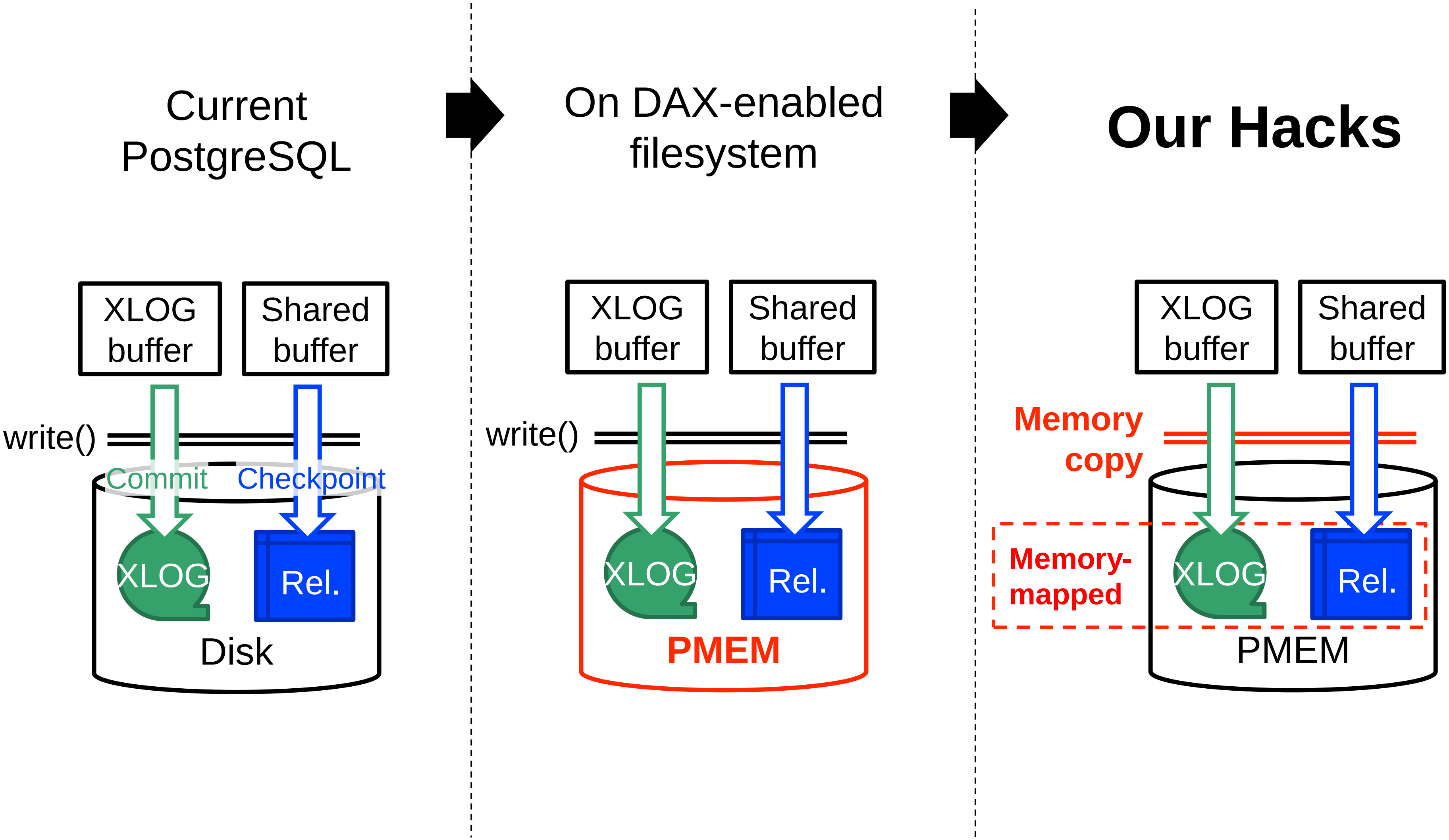
4. **Conclusion**

Conclusion

- **Applied PMDK into PostgreSQL**
  - In an easier way to use memory-mapped files and memory copy
- **Achieved not-so-bad results**
  - +3% transaction throughput
  - -30% checkpoint duration
- **Showed tips related to PMEM**
  - PMEM will change software design drastically
  - We should change software and our mind to bring out PMEM's potential much more
  - Let's try PMEM and PMDK :)

Backup slides

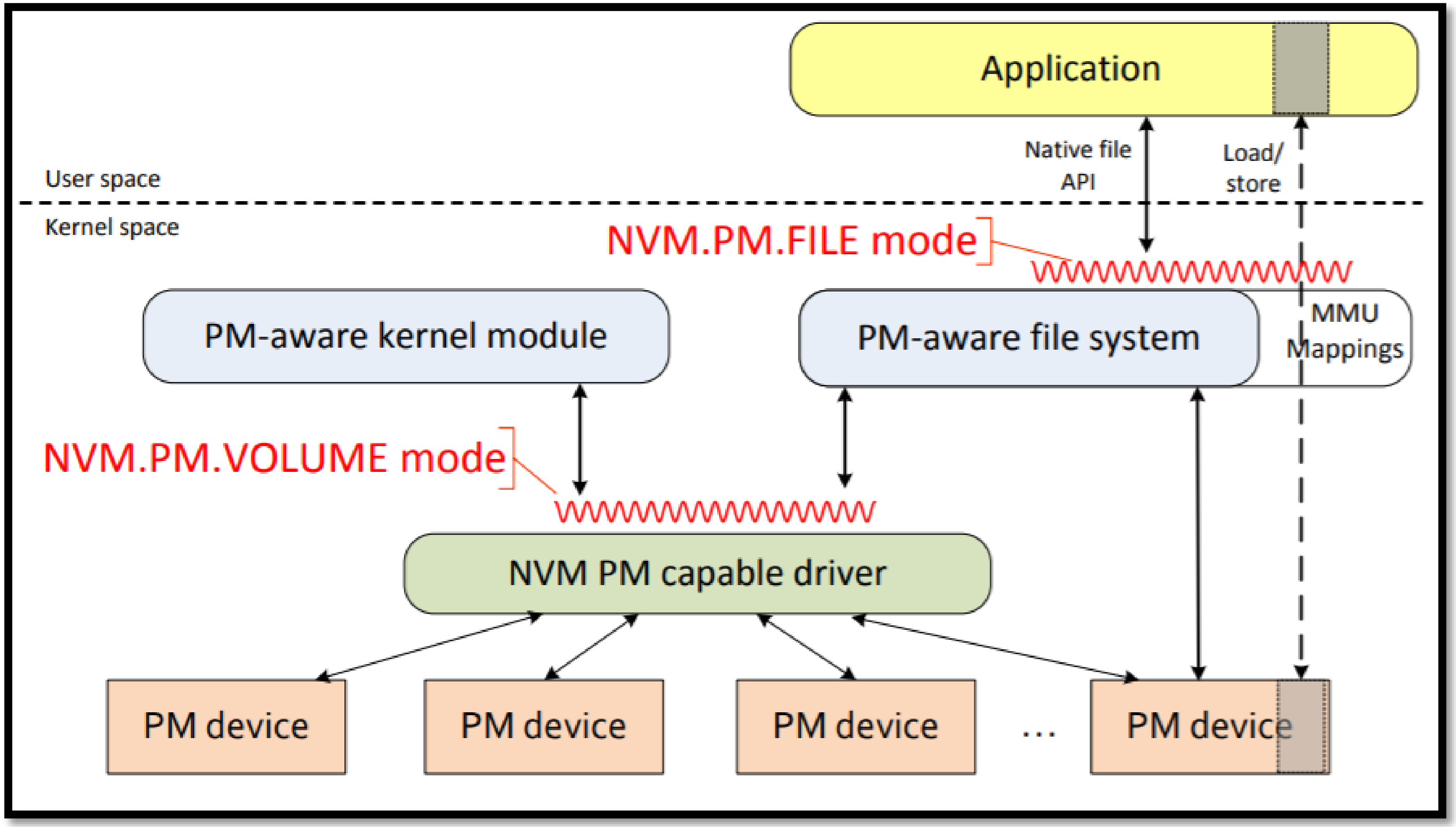
How to hack



Current PostgreSQL
On DAX-enabled filesystem
Our Hacks
XLOG buffer
Shared buffer
write()
Commit
Checkpoint
XLOG
Rel.
Disk
XLOG buffer
Shared buffer
write()
XLOG
Rel.
PMEM
XLOG buffer
Shared buffer
Memory copy
Memory-mapped
XLOG
Rel.
PMEM

SNIA NVM Programming Model

- **Defines behavior between user space and OS**
  - Here we focus on **NVM.PM.FILE** mode



(https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf)

API walkthrough

<table>
<tr><th></th><th>read/write</th><th>memory-mapped file</th><th>PMDK (libpmem)</th></tr>
<tr><td>Open</td><td><code>fd = open(
  path, flags, mode);</code></td><td><code>fd = open(
  path, flags, mode);</code></td><td rowspan="4"><code>pmem = pmem_map_file(
  path, len, flags, mode,
  ...);</code></td></tr>
<tr><td>Allocate</td><td>-</td><td><code>// map cannot be extended
// so pre-allocate the file
err = posix_fallocate(
  fd, 0, len);</code></td></tr>
<tr><td>Map</td><td>-</td><td><code>pmem = mmap(
  NULL, len, ..., fd, -1);</code></td></tr>
<tr><td>(Close)</td><td>-</td><td><code>// accessing file via mapped
// address; not fd any more
ret = close(fd);</code></td></tr>
<tr><td>Write</td><td><code>nbytes = write(
  fd, buf, count);</code></td><td><code>memcpy(
  pmem, buf, count);</code></td><td rowspan="2"><code>// bypassing cache if able
// instead of flushing it
pmem_memcpy_nodrain(
  pmem, buf, count);</code></td></tr>
<tr><td>Flush</td><td>-</td><td><code>for(i=0; i<count; i+=64)
  _mm_clflush(pmem[i]);</code></td></tr>
<tr><td>Sync</td><td><code>ret = fdatasync(fd);</code></td><td><code>_mm_sfence();</code></td><td><code>pmem_drain();</code></td></tr>
<tr><td>Read</td><td><code>nbytes = read(
  fd, buf, count);</code></td><td><code>memcpy(
  buf, pmem, count);</code></td><td><code>memcpy(
  buf, pmem, count);</code></td></tr>
<tr><td>Unmap</td><td>-</td><td><code>ret = munmap(pmem, len);</code></td><td><code>ret = pmem_unmap(pmem, len);</code></td></tr>
<tr><td>Close</td><td><code>ret = close(fd);</code></td><td>-</td><td>-</td></tr>
</table>

(**Blue**: Intel Intrinsics; **Red**: PMDK (libpmem))

NTT