SDC 18
September 24-27, 2018
Santa Clara, CA
www.storagedeveloper.org

Challenges for Implementing PMEM Aware Application with PMDK

Yoshimi Ichiyanagi

NTT Software Innovation Center

Outline

1. Introduction
2. Background and Motivation
3. How to use PMEM
4. Challenges for implementing PMEM aware applications
5. Challenges for performance evaluation to get valid results

Outline

1. **Introduction**
2. Background and Motivation
3. How to use PMEM
4. Challenges for implementing PMEM aware applications
5. Challenges for performance evaluation to get valid results

Introduction

- NTT Software Innovation Center is part of NTT Laboratories
- Worked on system software
  - Distributed file system (HDFS)
  - Operating system (Linux kernel)
- Trying to rewrite open-source software using new storage and new library

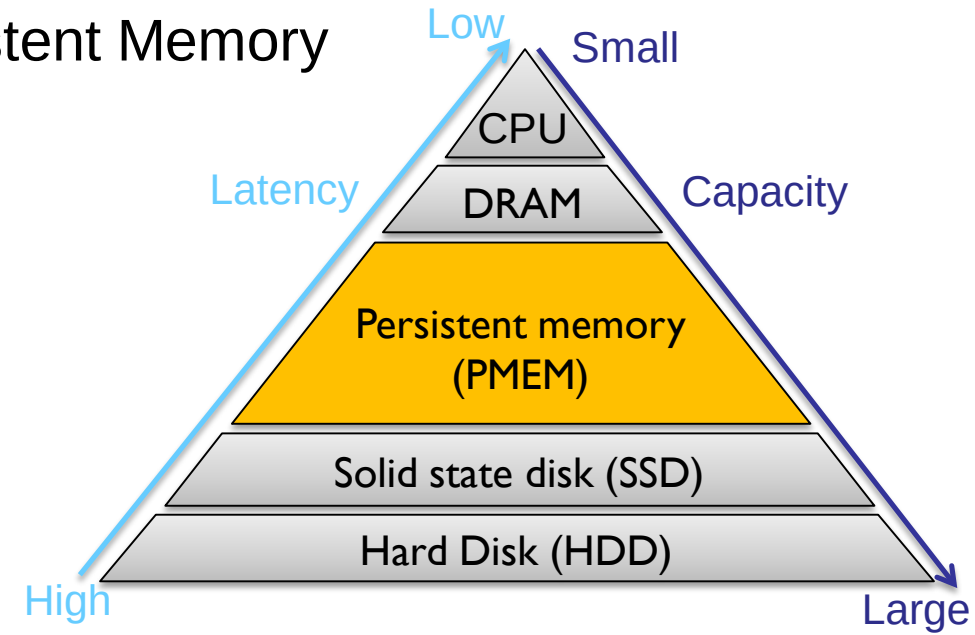
Outline

1. Introduction
2. **Background and Motivation**
3. How to use PMEM
4. Challenges for implementing PMEM aware applications
5. Challenges for performance evaluation to get valid results

Background

- Persistent memory (PMEM) begins to be supplied
  - NVDIMM-N
  - Intel® Optane™ DC Persistent Memory

- PMEM features are:
  - Memory-like features
    - Low-latency
    - Byte-addressable
  - Storage-like features
    - large-capacity
    - non-volatile



Motivation

- Trying to rewrite storage applications since PMEM features are utilized
  - RDBMS (e.g. PostgreSQL)
  - Message queue systems (e.g. Apache Kafka)
  - etc.

- Let me share my challenges about PMEM
  - Implementation
  - Performance evaluation

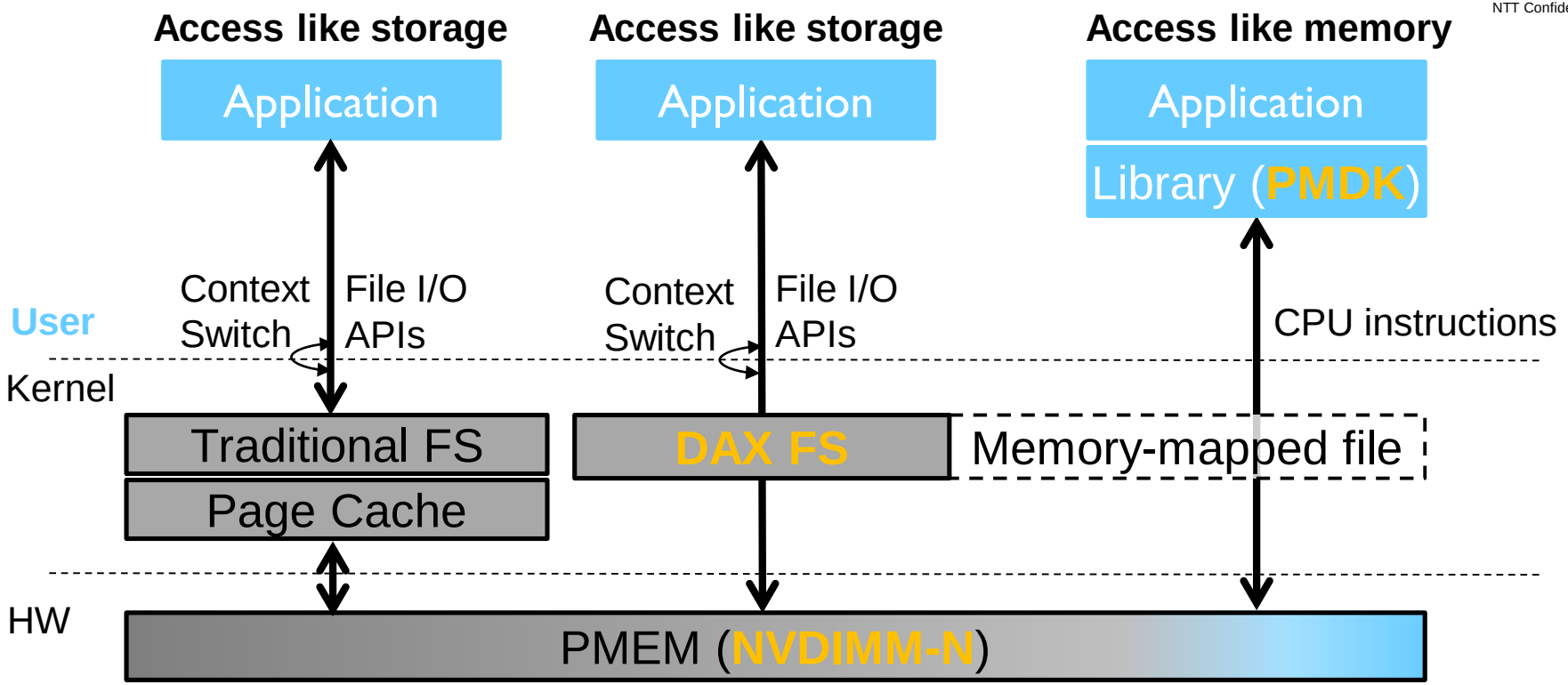
Outline

1. Introduction
2. Background and Motivation
3. **How to use PMEM**
4. Challenges for implementing PMEM aware applications
5. Challenges for performance evaluation to get valid results

HW/SW components

- What kind of PMEM is used?
  - **NVDIMM-N**
- What Linux kernel support is necessary to use PMEM?
  - **Direct-Access for files (DAX)**
    - Supported by ext4 and xfs
- What user library is used?
  - **Persistent Memory Development Kit (PMDK)**

DAX FS and PMDK



Benefits of DAX FS and PMDK

- With DAX FS only
  - Not necessary to rewrite applications
- With DAX FS and PMDK
  - Necessary to rewrite applications
  - Performance of I/O-intensive workload is greatly improved

PMDK features

- Application accesses PMEM like memory
  - Memory-mapped file (mmap file)
- Application developers can select fine-grained sync size
  - Details are on the next slide
- CPU instructions suitable for copy data size are selected
  - 8 / 16 / 32 / 64 bytes registers
  - MOVNT & SFENCE
    - without CPU caches

PMDK sync function

- pmem_msync()
  - File metadata and written data are flushed
    - pmem_msync() calls msync syscall
    - msync syscall is general sync API for mmap file
- pmem_drain()
  - Only written data is flushed
  - pmem_drain() is faster than pmem_msync()

	pmem_msync()	Pmem_drain()
Durability	○ (Flush file metadata and written data)	△ (Flush written data only)
Performance	×	○

Implementing application with PMDK

- Trying to rewrite storage applications with PMDK
  - RDBMS – PostgreSQL (PG)
  - Message queue systems - Apache Kafka

- Let me share know-how gained by rewriting PG
  - Checkpoint file
    - Many writes occur during checkpoint
  - Write ahead logging (WAL)
    - Critical for transaction performance

Outline

1. Introduction
2. Background and Motivation
3. How to use PMEM
4. **Challenges for implementing PMEM aware applications**
5. Challenges for performance evaluation to get valid results

4. Challenges for implementation

1. How to resize checkpoint file
2. How to select sync function for WAL

4. Challenges for implementation

1. **How to resize checkpoint file**
2. How to select sync function for WAL

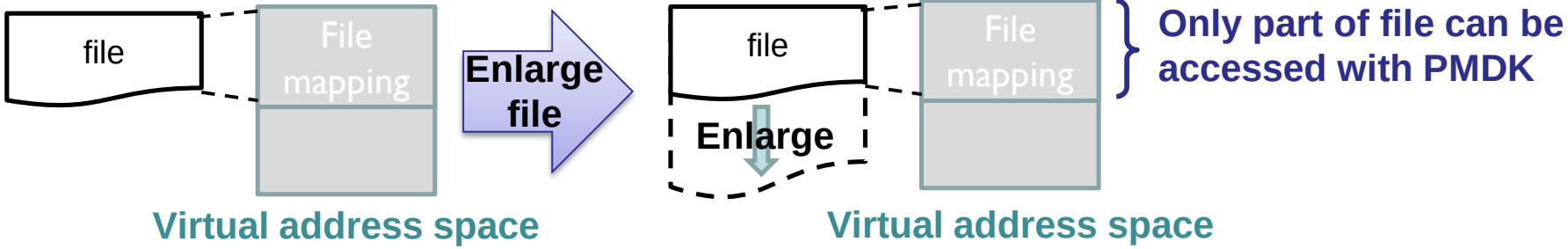
Resizing checkpoint file

- Huge table and so forth consist of multiple checkpoint files
  - Variable length up to 1GB
  - Necessary to resizing checkpoint file

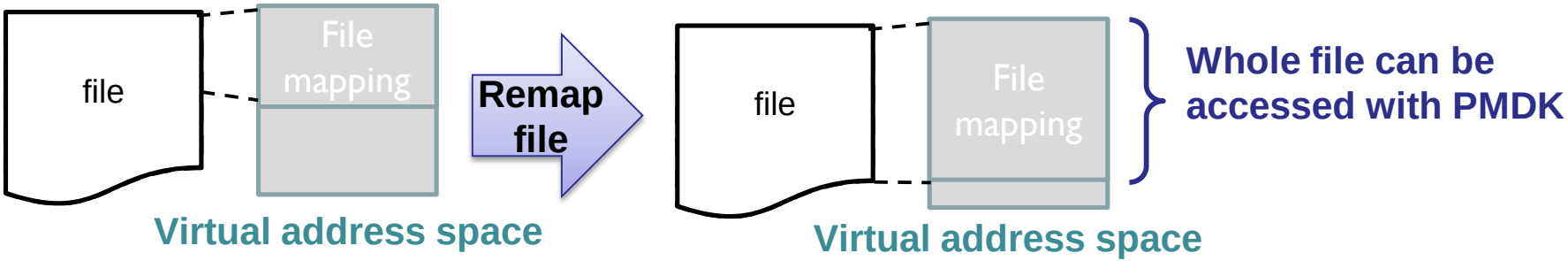
- PMDK provides APIs for mmap file
  - Difficult to resize mmap file without overhead
  - Best practice is to access only fixed-size file
    - We changed how to access only 1GB checkpoint file

How to resize mmap file

- Enlarge file



- Remap file



Implementation to enlarge file

- 3 function calls are added to use PMDK

	DAX FS	DAX FS and PMDK
Open	`fd = open(path, ...);`	`addr1 = pmem_map_file (path, len1, ...);`
Unmap		`pmem_unmap(addr1, len1);`
Extend		`truncate(path, len2);`
Remap		`addr2 = pmem_map_file (path, len2, ...);`
Close	`close(fd);`	`pmem_unmap(addr2, len2);`

Implementation to shrink file

- 2 function calls are added to use PMDK

	DAX FS	DAX FS and PMDK
Open	`fd = open(path, ...);`	`addr1 = pmem_map_file (path, len1, ...);`
Unmap		`pmem_unmap(addr1, len1)`
Shrink	`ftruncate(fd, len2);`	`truncate(path, len2);`
Remap		`addr2 = pmem_map_file (path, len2, ...);`
Close	`close(fd);`	`pmem_unmap(addr2, len2);`

Resizing file with PMDK

- Difficult to use PMDK unless file size is fixed
  - Repeating remapping many times degrades performance
    - Remapping file has large overhead
      - By using PMDK, munmap() /close()/open()/mmap() syscall is called again
  - Mapping large file may make file system full
- Best practice is to use fixed-size file only

4. Challenges for implementation

1. How to resize checkpoint file
2. **How to select sync function for WAL**

Selecting sync function for WAL

- How to write log to WAL file
  - Initialization – create file and fill file with zero
    - Necessary to flush file metadata
  - Synchronous logging - sequential synchronous write
    - Necessary to flush only written data

- PMDK sync APIs
  - pmem_msync() – file metadata and written data are flushed
  - pmem_drain() – only written data is flushed

PMDK sync function for WAL

- Initialization of WAL file
  - Necessary to flush WAL file metadata
  - pmem_msync()

- Synchronous logging
  - Necessary to flush only written data
  - pmem_drain() or pmem_msync()
    - pmem_drain() is faster than pmem_msync()
    - We selected pmem_drain()

Comparing PMDK sync functions

- I ran microbenchmark to compare pmem_msync() to pmem_drain()
  - Preprocessing
    - WAL initialization
    - Create file and fill file with zero and flush file metadata
  - Measurement processing
    - Synchronous logging
    - Overwrite data and flush written data

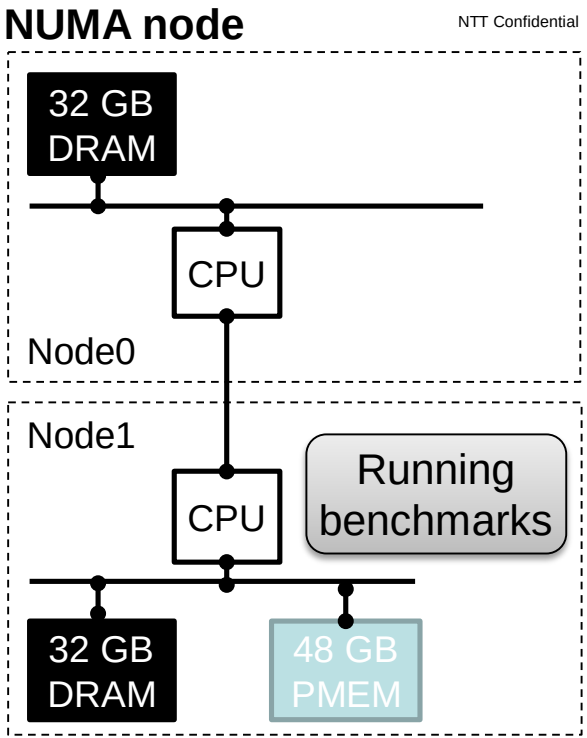
Microbenchmark

- Synchronous logging

	1. DAX FS fdatasync()	2. DAX FS and PMDK pmem_msync()	3. DAX FS and PMDK pmem_drain()
Open	open()	**pmem_map_file()**	**pmem_map_file()**
	while () {	while () {	while() {
Write	write()	**pmem_memcpy_nodrain()**	**pmem_memcpy_nodrain()**
Sync	**fdatasync()**	**pmem_msync()**	**pmem_drain()**
Loop N	}	}	}
Close	close()	**pmem_unmap()**	**pmem_unmap()**

Evaluation setup

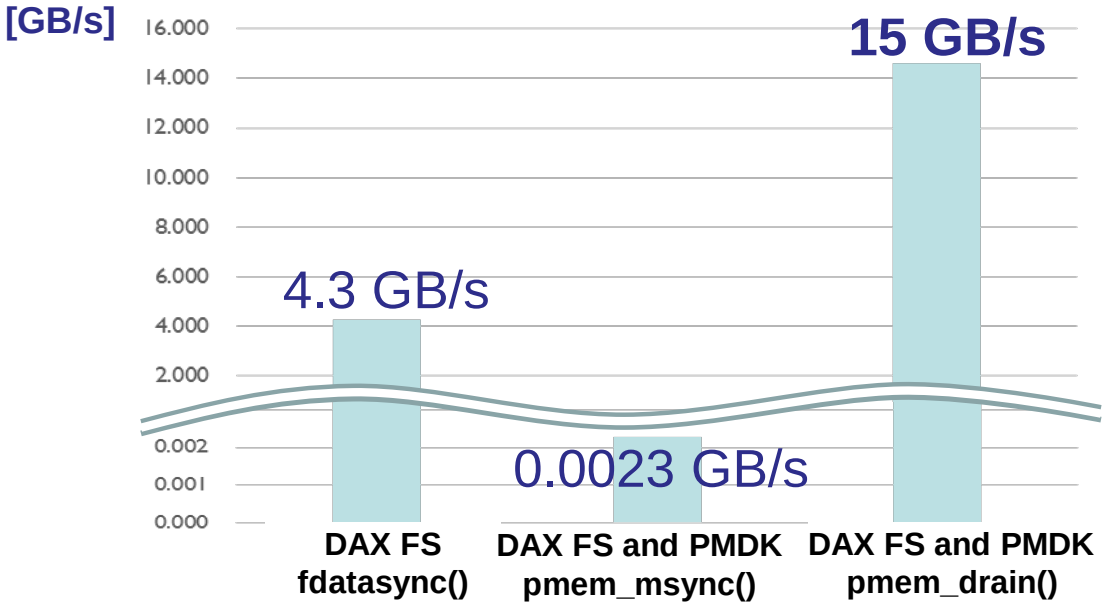
Hardware	
CPU	E5-2667 v4 x 2 (8 cores per node)
DRAM	[Node0/1] 32 GB each
PMEM (NVDIMM-N)	[Node1] 48 GB (HPE 8GB NVDIMM x 6)
Software	
Distro	Ubuntu 16.04
Linux kernel	4.17.9*
PMDK	1.4.1
Filesystem	ext4 (DAX available)



*: git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git

Performance evaluation - microbenchmark

- pmem_drain() is fastest of 3 patterns as expected



- Total written data is 10 GB
- Block is 8 KB

PMDK sync functions

- pmem_drain() greatly improves performance of I/O-intensive workload

- You should use pmem_drain() with caution
  - pmem_drain() can’t flush file metadata
    - pmem_msync() should be called by application that uses file metadata such as
      - Time of last modification
      - Time of last access
  - pmem_drain() doesn’t work without pmem_memcpy_nodrain()

Outline

1. Introduction
2. Background and Motivation
3. How to use PMEM
4. Challenges for implementing PMEM aware applications
5. **Challenges for performance evaluation to get valid results**

5. Challenges for performance evaluation

- Difficult to get valid results in PG performance evaluation

- What is valid result?
  - Avoid Non-Uniform Memory Access (NUMA) effects
  - Avoid CPUs becoming hotspots
    - tuning application for PMEM

5. Challenges for performance evaluation

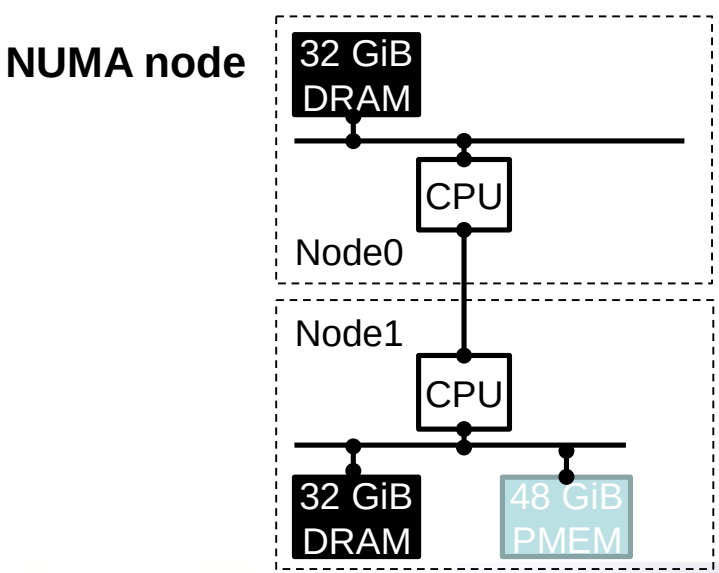
1. NUMA effect
2. Tuning application for PMEM

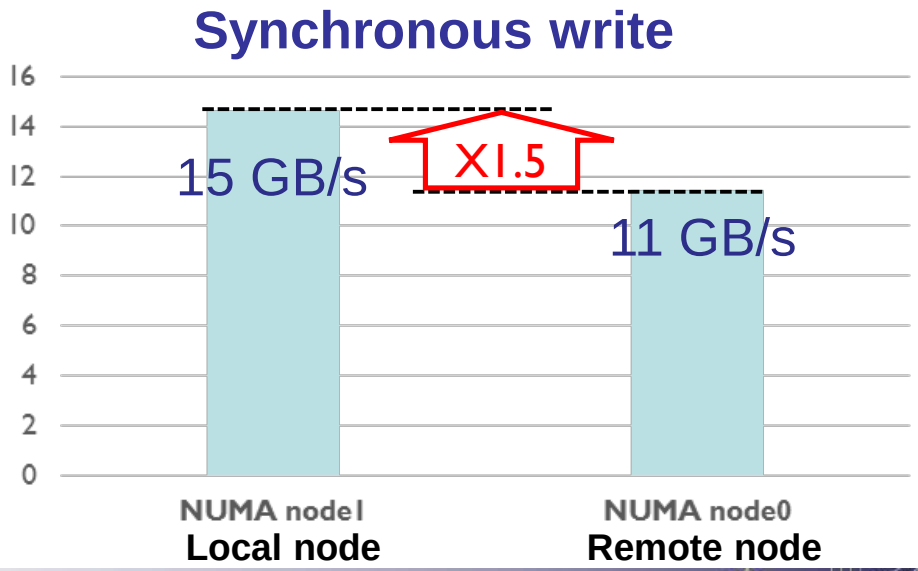
5. Challenges for performance evaluation

1. **NUMA effect**
2. Tuning application for PMEM

NUMA effect

- Synchronous write is about 1.5 times faster on local NUMA node than on remote NUMA node





5. Challenges for performance evaluation

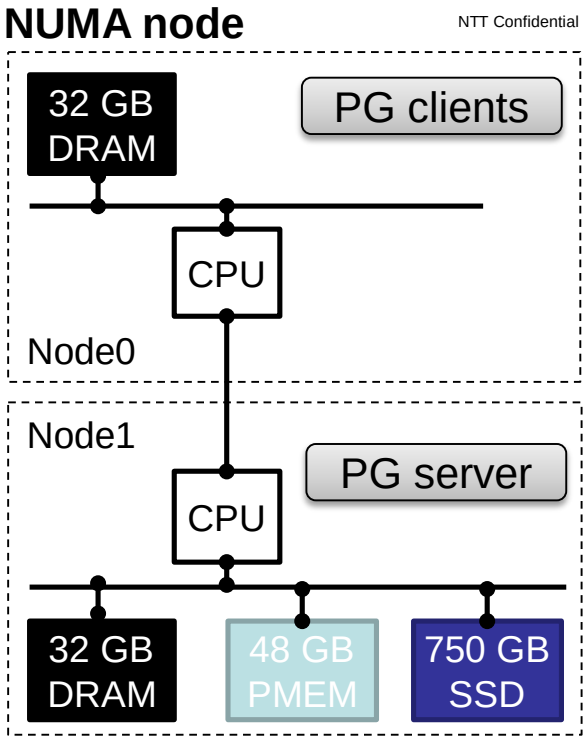
1. NUMA effect
2. **Tuning application for PMEM**

Tuning application for PMEM

- Important to avoid calculation processing becoming hotspot
  - Better to use Stored Procedure in PG
    - Stored Procedure improves PG performance since user-defined functions are pre-compiled and stored in PG serve
    - pgbench -c 16 -j 16 -T 1800 -r [db_name] **-M prepared**

Evaluation setup

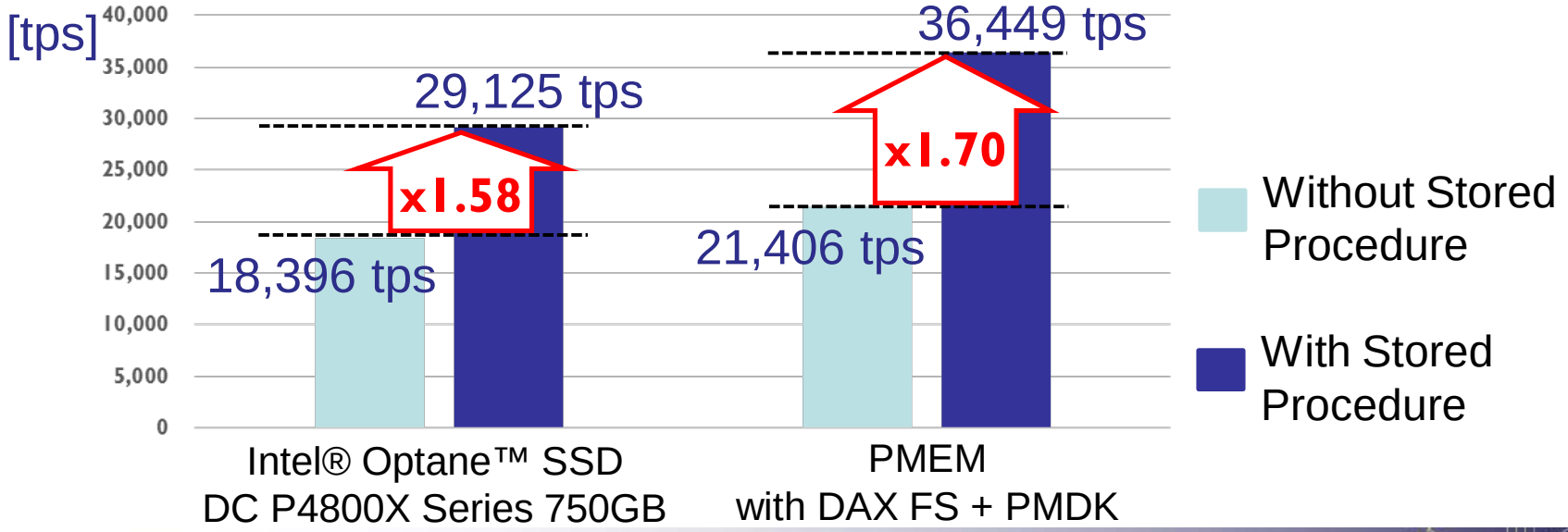
Hardware	
CPU	E5-2667 v4 x 2 (8 cores per node)
DRAM	[Node0/1] 32 GB each
PMEM (NVDIMM-N)	[Node1] 48 GB (HPE 8GB NVDIMM x 6)
SSD	Intel® Optane™ SSD DC P4800X Series 750GB
Software	
Distro	Ubuntu 16.04
Linux kernel	4.17.9
PMDK	1.4.1
Filesystem	ext4 (DAX available)
PostgreSQL	10.4 Beta*



*: https://www.postgresql.org/message-id/C20D38E97BCB33DAD59E3A1%40lab.ntt.co.jp

Stored Procedure

- Improvement ratio using PMEM is improved by 12% compared with using SSD



Conclusion

- Difficult to implement PMEM aware applications
  - Resizing file with overhead
    - Best practice is to access only fixed-size file
  - Selecting inappropriate sync function seriously degrades performance or durability

- Difficult to get valid results in performance evaluation
  - Avoiding NUMA effect
  - Avoiding CPUs becoming hotspots