SDC 19

September 23-26, 2019
Santa Clara, CA

SMB3 Push Mode

Low-latency RDMA to Persistent Memory

Mathew George
Tom Talpey

Microsoft

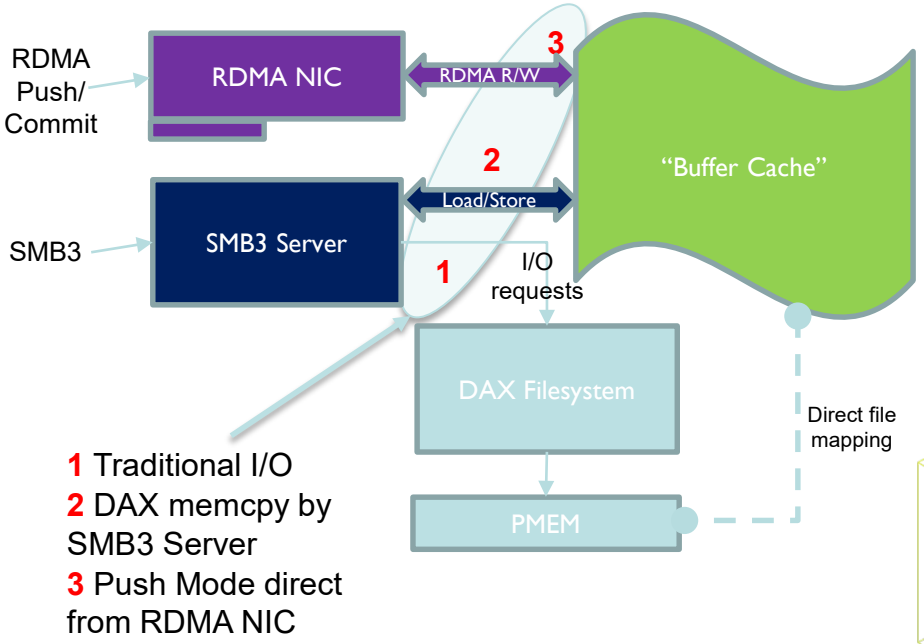
Outline

- Push Mode background (Tom)
- Push Mode prototyping and results (Mathew)

Background

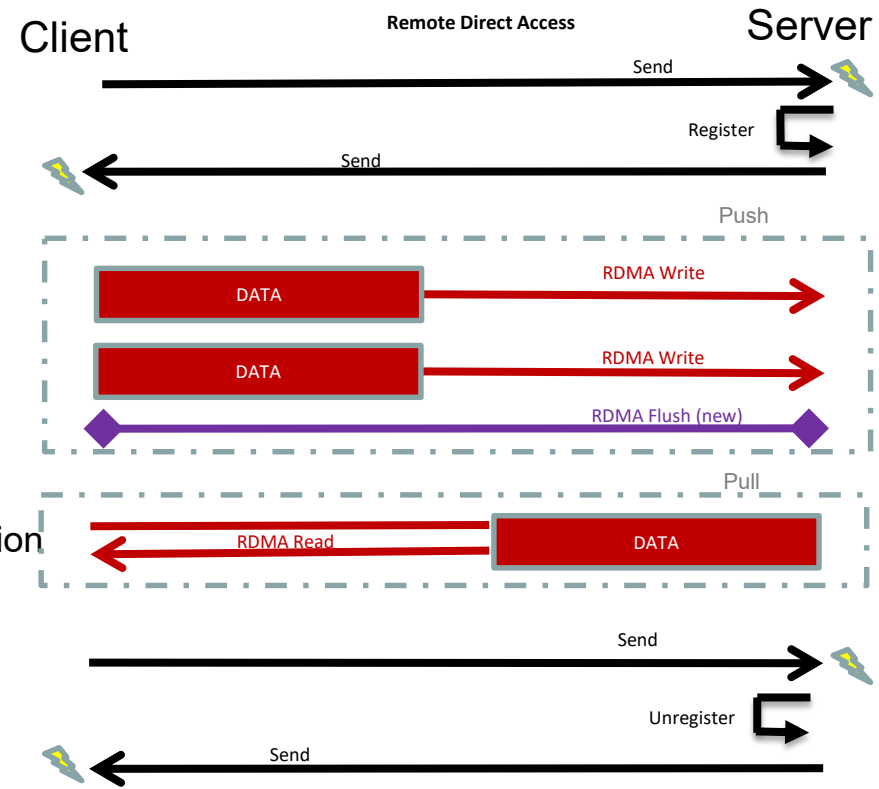
SMB3 Push Mode

- SMB3 RDMA and “Push Mode”
- Enables zero-copy remote read/write to DAX file, with ultra-low latency and overhead
- Multiple implementation options
- Phase 3 (RDMA) is ultimate low-latency



SMB3 Push Mode Setup

- Basic steps:
  - Open DAX-enabled file
  - Obtain a lease
  - Request a push-mode registration from the server
  - While (TRUE)
    - Push (or pull) data
    - Commit data to durability
    - Minimal/no CPU intervention
  - Release registration
  - Drop lease
  - Close handle



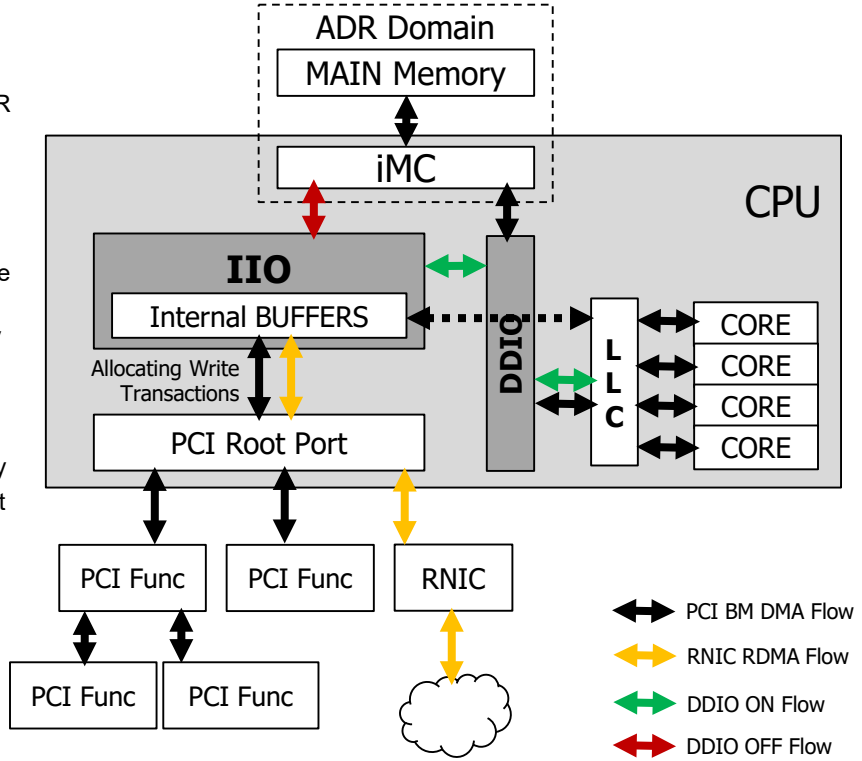
RDMA Flush

- RDMA Protocol extension ("Flush") under way
  - Providing a remote guarantee of Durability
  - Ordered, Flow controlled, acknowledged
    - Initiator requests specific byte ranges to be made durable
    - Responder acknowledges only when durability complete
  - Additional "Selectivity" and "Scope" semantics
  - Strong consensus in IBTA, IETF, and RDMA community
  - Standards process still not complete…
- PCI Extension also envisioned
  - Allow RDMA NIC to perform flush without invoking CPU
    - Lowest latency, best performance
  - PCI SIG discussion is TBD

Background: RDMA with DRAM

Intel HW Architecture (SDC 2016)

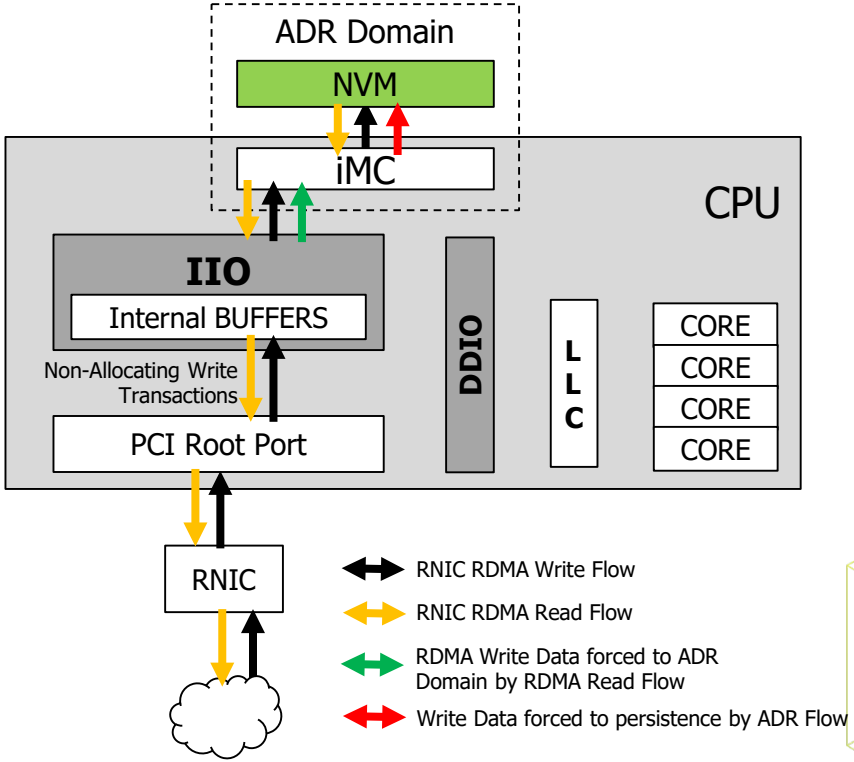
- **ADR – Asynchronous DRAM Refresh**
  - Allows DRAM contents to be saved to NVDIMM on power loss
  - ADR Domain – All data inside of the domain is protected by ADR and will make it to NVM before supercap power dies. The integrated memory controller (iMC) is currently inside of the ADR Domain.
- **IIO – Integrated IO Controller**
  - Controls IO flow between PCIe devices and Main Memory
  - "Allocating write transactions"
    - PCI Root Port will utilize write buffers backed by LLC core cache
    - Data buffers naturally aged out of cache to main memory
  - "Non-Allocating write transactions"
    - PCI Root Port Write transactions utilize buffers not backed by cache
    - Forces write data to move to the iMC without cache delay
  - Enable/Disable via BIOS setting globally per platform, per Root PCI Port, or per PCI Transaction (see detail slide)
- **DDIO – Data Direct IO**
  - Allows Bus Mastering PCI & RDMA IO to move data directly in/out of LLC Core Caches
  - Good for data which CPU needs to process
  - But defers persistence, and forces CPU to flush
  - Allocating Write transactions will utilize DDIO



Workaround: RDMA to PMEM

Intel HW Architecture

- **Disabling DDIO**
  - Enable “non-allocating Write” transactions per PCI Root Port
    - Requires PCI configuration change(s)
    - Forces RDMA Write data directly to iMC
    - Enable on PCI Root Port/Slot with RNIC
  - Follow RDMA Write(s) with RDMA Read to force remaining IIO buffer write data to ADR Domain
    - RDMA Read acts as a fencing function for the previous non-allocating write data and forces remaining write data out of the Root Port / IIO pipeline
    - Must force a PCI Read on the local PCI Root Port that handled the writes
    - Since RDMA Write and Read are silent, there is no CPU impact (or visibility)
    - RDMA Read – Read can be for any address, length > 0



8

Reconfiguration: DDIO disable

- Disabling Allocating Write Flows for the IIO
  - All memory writes for all devices connected to selected PCI Root Port will be affected
  - Offset 0x180 - perfctrlsts_0
    - Bit 7 – use_allocating_flow_wr
      - Set to 0 – **Non-allocating writes will be generated** for all memory writes for all devices on Root Port
    - Bit 3 - nosnoopopwren
      - Set to 0 – Use allocating or **non-allocating flows as specified by Bit 7**
- Use a PCI Configuration utility to rewrite register
  - Use Device Manager / View by Connection to identify PCI Root Port
  - 8 bits @ 0x180: 0x99 (default) ► 0x11

INTEL XEON PROCESSOR E5 V2 PRODUCT FAMILY DATA SHEET MARCH 2014
IIO CONTROL REGISTERS

8.2.82 perfctrlsts_0

Performance Control and Status Register 0.

Type: CFG	PortID: N/A		
Bus: 0	Device: 0	Function: 0	
Bus: 0	Device: 1	Function: 0-1	
Bus: 0	Device: 2	Function: 0-3	
Bus: 0	Device: 3	Function: 0-3	
Offset: 0x180			

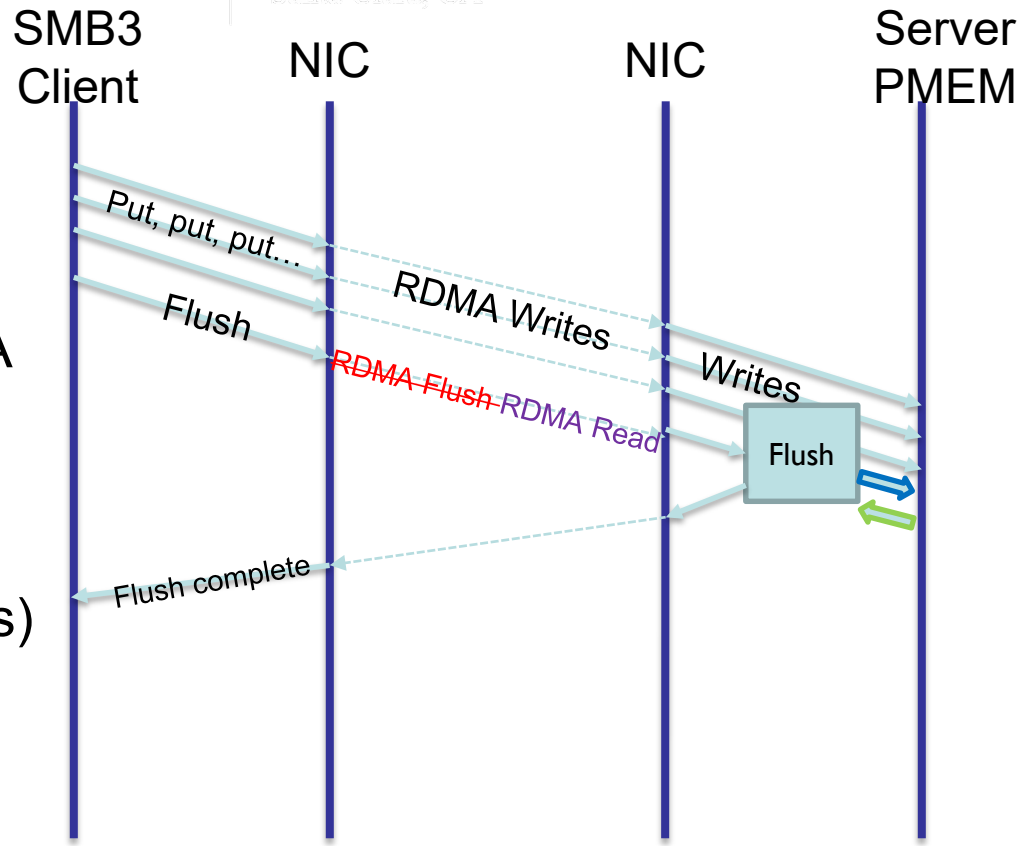
Bit	Attr	Default	Description
20:16	RW	0x18	outstanding_requests_gen1:
13:8	RW	0x30	outstanding_requests_gen2:
7:7	RW	0x1	use_allocating_flow_wr: Use Allocating Flows for 'Normal Writes' on VC0 and VCp 1: Use allocating flows for the writes that meet the following criteria. 0: Use non-allocating flows for writes that meet the following criteria. (TPH=0 OR TPHDIS=1 OR (TPH=1 AND Tag=0 AND CIPCTRL[28]=1)) AND (NS=0 OR NoSnoopOpWrEn=0) AND Non-DCA Write Note: VC1/VCm traffic is not impacted by this bit in Dev#0 When allocating flows are used for the above write types, IIO does not send a Prefetch Hint message. Current recommendation for BIOS is to just leave this bit at default of 1b for all but DMI port. For DMI port when operating in DMI mode, this bit must be left at default value and when operating in PCIe mode, this bit should be set by BIOS. Note there is a coupling between the usage of this bit and bits 2 and 3. TPHDIS is bit 0 of this register NoSnoopOpWrEn is bit 3 of this register

Bit	Attr	Default	Description
3:3	RW	0x0	nosnoopopwren: Enable No-Snoop Optimization on VC0 writes and VCp writes This applies to writes with the following conditions: NS=1 AND (TPH=0 OR TPHDIS=1) 1: Inbound writes to memory with above conditions will be treated as non-coherent (no snoops) writes on Intel QPI 0: Inbound writes to memory with above conditions will be treated as allocating or non-allocating writes, depending on bit 4 in this register. If TPH=1 and TPHDIS=0 then NS is ignored and this bit is ignored VC1/VCm writes are not controlled by this bit since they are always non-snoop and can be no other way. Current recommendation for BIOS is to just leave this bit at default of 0b. Refer to the Transaction Flow chapter in EDS Volume 3 for what needs to be guaranteed at the system/usage model level for BIOS to set this bit.

9

Workaround Replication Exchange

- Reconfigure **Server**'s PCI Root
- SMB3 client **substitutes** RDMA Read for RDMA Flush
- Measure performance gain(s) to write DAX file



SMB3 Prototype and Results

SMB3 RDMA Overview

- Near zero configuration RDMA
  - Discovery using multichannel.
  - Layered on top of SMBDirect (MS-SMBD)
  - Support for IB, iWARP, RoCE, RoCEv2 networks.
- Uses
  - RDMA send/receive for small payloads.
  - RDMA read/write for large reads and writes.

Recap : SMB3 RDMA Direct Placement

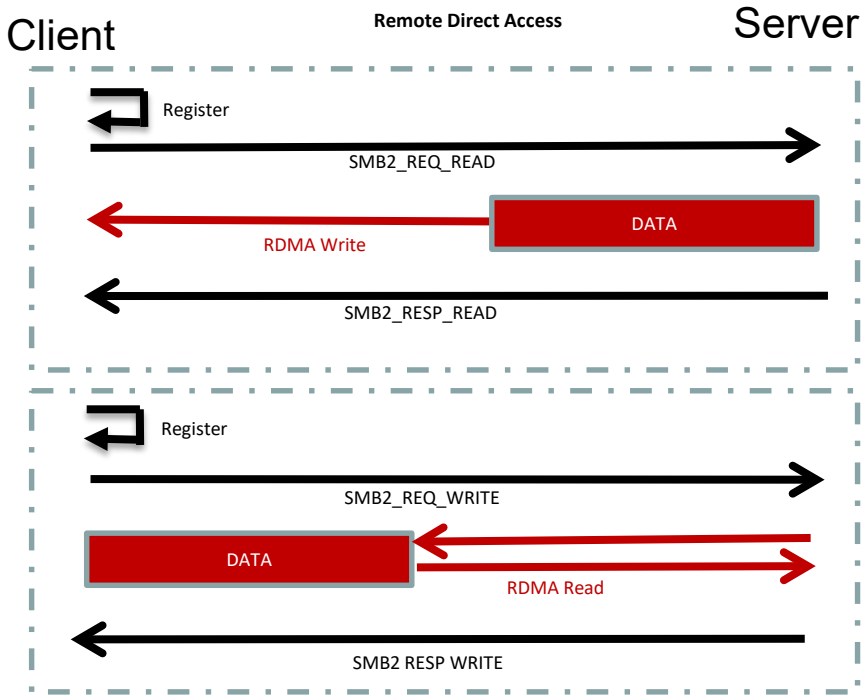
For every IO, client registers memory with RDMA (fast register memory regions.)

Sends buffer descriptors to server.

Server pushes/pulls data.

CPU processing on server

- SMB2 request execution.
- Registration of local buffers
- At least 2 interrupts per IO.



Prerequisites for SMB3 Push Mode

- DAX filesystem which can provide layout of a file allowing direct byte-granularity access via mapped memory.
  - Full PM device access can be achieved via a virtual file which maps the entire device.
- Ability to acquire a lease which guarantees the validity of the layout.
  - Filesystem support is needed.
- Protocol does not require software processing of data once it is placed in persistent memory.

Prerequisites for SMB3 Push Mode

- Ability to register potentially very large (100s of GBs), physically contiguous memory regions.
  - 1 - 2 GB limits on current implementations.
  - Changes needed to SMBDirect, NDKPI and NIC drivers & hardware
  - Uses “slow” registration path. (NDK_REGISTER_HUGE_MR)
- Reversal of roles – server registers memory, client reads/writes data.
  - Changes needed to SMBDirect, SMB Client and Server.
- Guarantee durability of data in persistent memory.
  - Explicit flush primitive or CPU cache bypass OR software flush.

Mapping for direct access

- Client requests server to map DAX file for direct-access.

```
typedef struct _LMR_MAP_RANGE_FOR_DIRECTACCESS_REQUEST {
   ULONG  AccessFlags;
   UINT64 Offset;
   UINT64 Length;
} LMR_MAP_RANGE_FOR_DIRECTACCESS_REQUEST;
```

- Server queries DAX filesystem to get “physical extents” of file.
  - Optionally memory map the file to get a VA range.
- Register the physical extents with RDMA NIC for remote access.
  - Query the RDMA token and build “large” RDMA descriptor.
- Return RDMA descriptor to client

```
typedef struct _LMR_MAP_RANGE_FOR_DIRECTACCESS_RESPONSE {
   UINT64 DirectAccessMappingId;
   ULONG  cbDescriptor;
   UCHAR  Descriptor[1];
} LMR_MAP_RANGE_FOR_DIRECTACCESS_RESPONSE;
```

Push Mode RDMA Read

- Find connection where file handle was direct mapped.
- Issue RDMA-read to application buffer using the RDMA token.
  - Validate that offset/length falls within direct mapped region.
  - Connection will be terminated otherwise.
- No SMB2 protocol processing
- Server CPU is not interrupted.
- Single client-side interrupt - RDMA read completion

Push Mode RDMA Write

- Find connection where file handle was direct mapped.
- Issue RDMA-write from application buffer using the RDMA token.
  - Validate that offset/length falls within direct mapped region.
  - Connection will be terminated otherwise.
- Issue “RDMA flush” operation
  - RDMA driver could map this to an RDMA read if DDIO is disabled.
- Server CPU is not interrupted.
- Single client-side interrupt - RDMA write completion

Push Mode RDMA Write w/ software flush

- Find connection where file handle was direct mapped.
- Issue RDMA-write from application buffer using the RDMA token.
  - Validate that offset/length falls within direct mapped region.
  - Connection will be terminated otherwise.
- Issue “SMB2 software flush” (pipelined with RDMA write)
  - Implemented as a “dummy write” without payload.

```
#define SMB2_WRITEFLAG_FLUSHONLY (0x00000004)
```

  - SMB2 server receives request and flushes written region using appropriate CPU primitive (clwb, clflush, clflushopt)
    - VA mapping is needed!
- Single server interrupt for SMB2 software flush.
- 2 client-side interrupts - RDMA write & SMB2 flush completion.

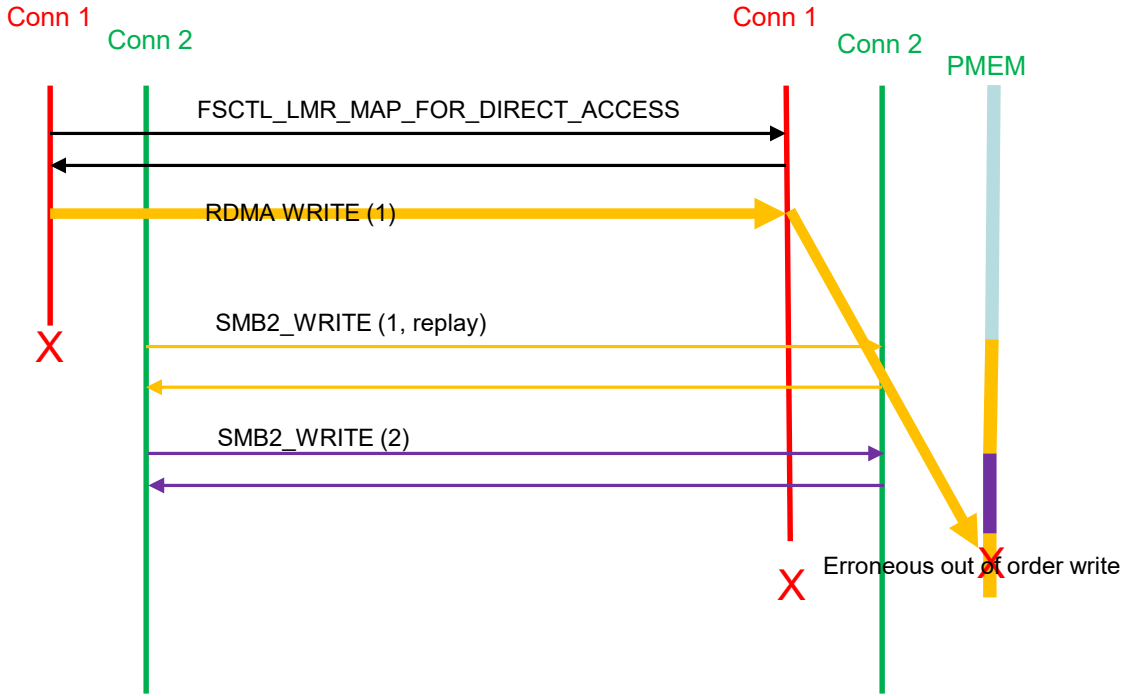
Securing direct-mapped regions

- Protection domains (PDs) are used to secure access to RDMA resources (connections, memory registrations etc.)
- Protection domain is assigned to a connection when created.
  - All connections share same PD.
  - Assign one PD per connection. (Windows approach !)
- This means memory registrations are affinitized to a connection.
  - Explosion of memory registrations !
- For our prototype implementation, all IOs on a direct-mapped handle are bound to a single connection.
  - No multichannel load balancing.

RDMA Write ordering issues

- Can we mix SMB2 writes with RDMA push mode writes ?
  - Yes, but need to enforce correct write-write ordering.
- How do we fence RDMA writes ?
  - No software processing on the server.
- Remove memory registration on server before replaying write.
  - Late arriving RDMA writes will automatically tear down connection.
- Should SMB client do automatic fallback ? Or defer to app ?

Failover and error handling



Latency measurement (NVDIMM)

- Dell PowerEdge R740XD, 2 x Intel Xeon Silver 4110 @ 2.10 GHz with 8 Cores
  - 16 GB NVDIMM-N
  - 25 Gbps RDMA NIC (Manufacturer A)
  - Optimal NV flush method is CLWB
- NVDIMM formatted in DAX mode w/NTFS with single 2 GB file mapped to system process.
- Single RDMA registration of size 2 GB bound to a single RDMA connection.
- Modified cfstest.exe tool to request direct mapping of file.
- These numbers are based on experimental code and hardware!

Latency measurements (NVDIMM-Read)

```
cfstest.exe r-read s:\file.bin  /o:<QD> /b:<BLOCKSIZE> /s:2G
/writethrough [/mapfordirectaccess:2G]
```

Latency (us) →	SMB3 Perf Ctr : SMB Client Shares\Avg. sec/Read	SMB3 w/Push-mode Perf Ctr : SMB Client Shares\Avg. sec/Read
QD=1, 4K	51	15
QD=2, 4K	52	16
QD=8, 4K	117	52
QD=1, 64K	77	40
QD=2, 64K	81	49
QD=8, 64K	184	123

* CPU utilization on server is zero.

Latency measurements (NVDIMM-Write)

```
cfstest.exe r-write s:\file.bin /o:<QD> /b:<BLOCKSIZE> /s:2G
/writethrough [/mapfordirectaccess:2G]
```

Latency (us) → Perf Ctr : SMB Client Shares\Avg. sec/Write	SMB3	Push-mode w/ software flush	Push-mode (DDIO disabled)
QD=1, 4K	81	26	14
QD=2, 4K	84	32	15
QD=8, 4K	143	72	59
QD=1, 64K	131	48	38
QD=2, 64K	145	51	42
QD=8, 64K	219	107	124

Latency measurement (Intel 3D XPoint DIMM)

- 2x Intel 2nd Gen Xeon Scalable Gold 6252 (24 core, 2.10Ghz)
  - 12x 126 GB Optane DC Persistent Memory Module
  - 25 Gbps RDMA NIC (Manufacturer B)
- PM disk formatted in DAX mode w/NTFS with single 1 GB file mapped to system process.
- Single RDMA registration of size 1 GB bound to a single RDMA connection.
- Modified cfstest.exe tool to request direct mapping of file.

Read Latency (Intel 3D XPoint DIMM)

```
cfstest.exe r-read s:\file.bin  /o:<QD> /b:<BLOCKSIZE>
/s:1G /writethrough [/mapfordirectaccess:1G]
```

Latency (us) →	SMB3 Perf Ctr : SMB Client Shares\Avg. sec/Read	SMB3 w/Push-mode Perf Ctr : SMB Client Shares\Avg. sec/Read
QD=1, 4K	68	29
QD=2, 4K	72	29

- Double latency compared to NVDIMM setup
  - RDMA NIC is likely the cause.
- Compared to SMB3 write, there is still a significant gain.

Write Latency (Intel 3D XPoint DIMM)

```
cfstest.exe r-write s:\file.bin /o:<QD> /b:<BLOCKSIZE> /s:1G
/writethrough [/mapfordirectaccess:1G]
```

Latency (us) → Perf Ctr : SMB Client Shares\Avg. sec/Write	SMB3	Push-mode w/ software flush	Push-mode (DDIO disabled)
QD=1, 4K	81	32	30
QD=2, 4K	84	33	31

- Similar observations as read.
- Extra cost of software flush appears minimal.

Applications and Challenges

- Persistent log replication infrastructure
- S2D high performance NVDIMM tier storage
  - Expose entire PM device for direct access
- High speed memory to memory transfer.
  - Example would be migrating VM memory
- Security issues opening server memory to clients
  - Limited to secure datacenter environments
  - Can we use a shared protection domain to share memory registrations across connections ?


Thank you!