moving file write flushes to userpace

disks vs persistent memory

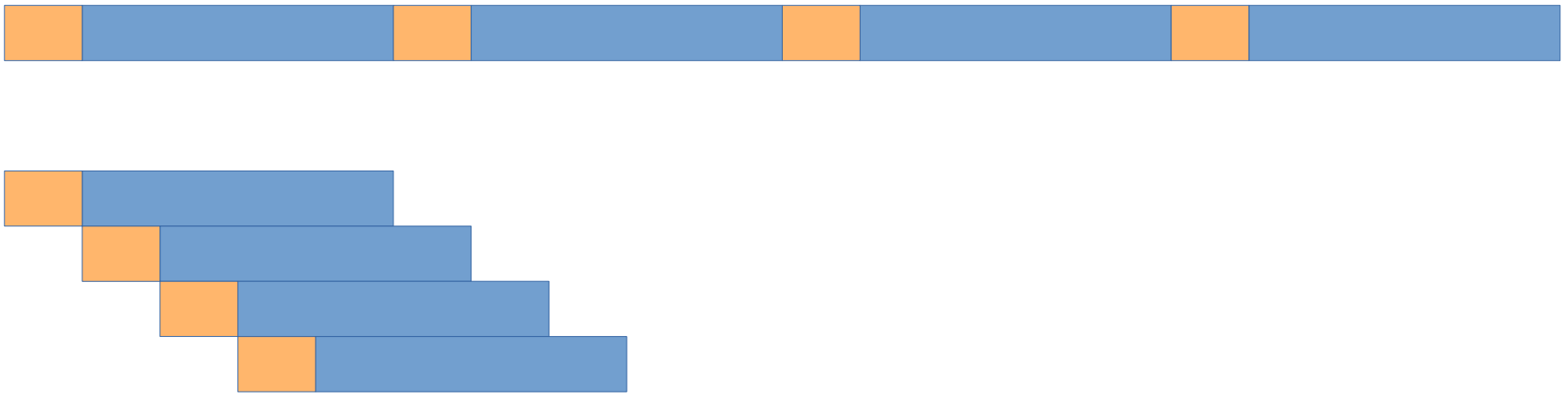
every transaction needs two flushes

Otherwise, reordered or torn writes would corrupt data.

WAL can reduce that to one flush, at a large cost, that's still worth it on HDDs but not anywhere else.

A filesystem's metadata *also* needs a transaction to update. Add more layers and the number of flushes explode exponentially.

and fsync() waits for completion



so do other filesystem APIs

Such as *msync*() for mmapped files.

There's *sync_file_range*() but it merely allows syncing multiple pieces of data at the same time, waiting for completion just once.

proposed: fsbarrier()

- `fsbarrier(int fdA, int fdB);`

  Would ensure any writes to fdA before the call are no less durable than writes to fdB started after the call returns.

  Alas, only some filesystems can implement that reasonably. Such as btrfs (where it'd use *flushoncommit* for that single file), but not eg. ext4 or xfs.

  Even on ext4, it would greatly help on nbd.

  But is not in the kernel.

a nbd client in Pluto's orbit mounting a disk on Earth suffers from half-ping of 9 hours

That means, a single transaction takes 36 hours with *fsync*().

With *fsbarrier*(), it completes immediately.

Unless you require durability, but even then you can persist any number of transactions in a single round-trip within 18 hours.

so, can we do better?

- On HDDs, not at all. No concurrent writes.
- On SSD, that'd be possible, we just lack kernel support. It seems no one is working on that.
- On NVDIMMs, that's implemented from the very start, in hardware.

SSDs actually do delay a part of the writes

They have a battery (which some vendors skip for cost reasons) that completes some part of the write.

Alas, you still have to call the kernel (syscall), the kernel has to copy the whole page you written (4096 bytes) to page cache, issue a PCIe request (or much worse, SATA), wait for the transfer to complete, request an ACK from the disk, and only then it can return to you.

By then, the write is in the disk controller's memory, safe because of a battery or supercapacitor.

NVDIMMs are byte-addressable

(Ok, writes and reads still go a cacheline at a time.)

And if mounted with **-o dax**, they can be accessed directly, like any other form of memory. No need to change the whole page at a time, copy it around, and so on.

But the data still needs to be flushed.

syscalls would be too slow

Any syscall requires a context switch to enter the kernel. Compared to memory writes, this takes ages.

Thus, we need some other way to signal that the data has to get persisted.

But some architectures (except eg. RISCV) already have control over CPU caches. We can use that.

cache flushes

- on x86, that's CLFLUSH, CLFLUSHOPT, CLWB
- on arm, that's DC CVAC, DC CVAP
- On powerpc, that's DCBST

  Any of the above starts flush on a single cacheline.

  You then wait for a series of flushes by issuing a store fence (SFENCE, DMB [ISH]ST, EIEIO).

  Obviously, you use a library for that (*libpmem*).

and the fences don’t even wait

The processor doesn’t have to stop processing subsequent instructions, it merely tells the memory controller to not reorder.

(And the processor is not allowed to let subsequent instructions have *observable* effects before the ACK.)

ADR

There's an issue, though. Unlike disks, memory didn't need to care about completing writes during power loss. Until now.

Like SSDs there's a battery or supercap that has just enough juice to last these final milli- or microseconds to finish everything, shut the lights off and lock the doors.

But the machine needs to tell the DIMM the power is going down.

hardware implementations

- NVDIMM-N: regular DRAM with flash storage attached
- Intel Optane DC Persistent Memory: 3D Xpoint

  Both require hardware support from the processor and motherboard to get ADR.
- Emulation: `memmap=4G!16G`; it's not quite as persistent as these above.

using in software

Unlike the traditional filesystem API (*write*(), *read*(), etc), persistent memory use requires *mmap*()ing the whole file beforehand. The kernel will still do page faults to note down which pages are dirty (it doesn't trust you...) so it can write them out eventually. But with userspace flushes as described above, any data you wrote won't get lost on unexpected crash or power loss.

- For more advanced uses, such as allocations, there are higher level libraries such as libpmemobj, then pmemkv on top of that, but that's out of scope for this talk.

but we still want to speed up old-style disks

Whether with *barrier*() or some other way...