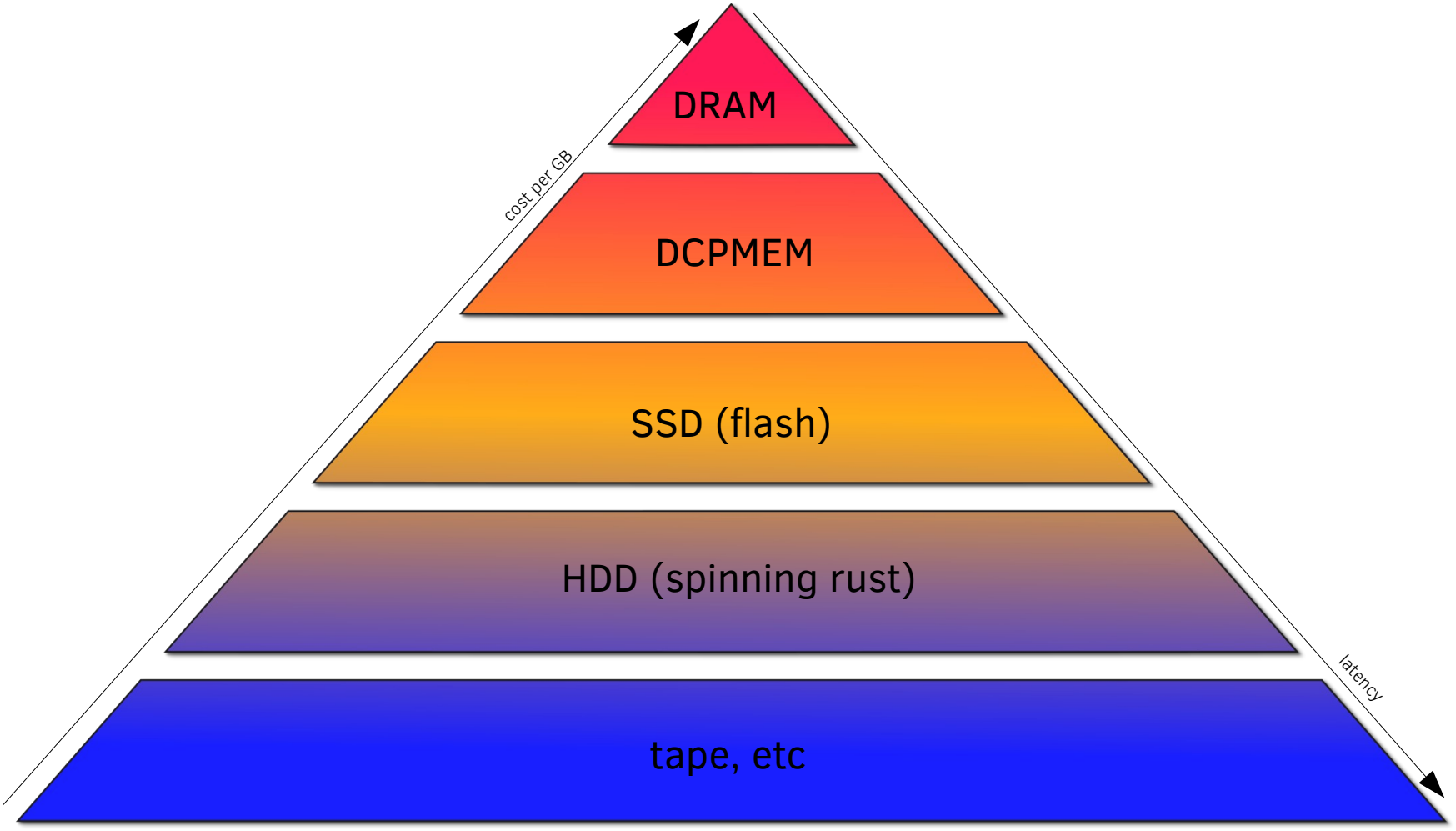
Persistent Memory

Why? Because existing hot-tier storage:

- is too slow
- requires kernel syscalls to access
- is not byte-addressable, thus requires copying


DRAM
DCPMEM
SSD (flash)
HDD (spinning rust)
tape, etc
cost per GB
latency

NVDIMM-N

Regular DRAM with a battery and some flash glued on, saves its contents on power loss.

- as much DRAM as flash: full DRAM speed
- more flash than DRAM: speed degrades to disk speeds once workload size goes beyond DRAM

  (ie, hardware-assisted swap)

Optane DC PMEM

- New type of internal medium (3D Xpoint).
- Slower than DRAM, much faster than flash.
- Current gen: Apache Pass.

How fast?

Alas, all benchmarks lie.

Especially vendor benchmarks.

So let's look at a paper from U of California:

https://arxiv.org/pdf/1903.05714.pdf

Latency of small random reads

- HDD: 20-5ms
- SSD: 100µ-25µs (2019 high-end NVMe)
- Apache Pass: 305ns random, 169ns predictable (UoC data)
- DRAM: 81ns (UoC data)

Bandwidth of linear reads

- Apache Pass: ~40GB/s on single CPU socket (UoC data)

But what about writes?

What do you mean by “the write is complete”?

It takes a long time to settle. But we can do many staggered writes in parallel.

Which could bite on an unexpected power loss… unless there’s a battery, a supercap, a gas-driven generator, etc. And if we know we’re going down.

ADR

All current implementations of persistent memory require a notification of power going down. This is done via a special hardware link, called ADR.

Once the signal is sent, all pending writes get finalized.

Bad news

Alas, hardware you have likely has no ADR yet.

It requires support both from CPU and motherboard. Thus sorry, you need to buy new gear to put these shiny NVDIMMs into.

Emulation

On x86: append to kernel's cmdline:

memmap=4G!16G

to get 4GB of emulated pmem (at 16th gigabyte).

On non-x86 use device tree:

Documentation/devicetree/bindings/pmem/pmem-region.txt

But that’s hardware… what about support in software?

memory mode

Needs no software but ipmctl to set it up.

- uncached: used same as any other memory
- cached: takes all your DRAM to use as a “cache”. A cacheline not present in DRAM will be fetched (swapped in) from pmem.

manual allocations: libvmem

Provides a malloc-like interface, so you can mix regular malloc() from DRAM for hot data, with vmem_malloc() for colder stuff.

Yet, using libvmem is not a good idea.

memkind

Modern machines tend to be NUMAed. A cross-socket/chiplet access takes a long time.

memkind does all what vmem could, but also knows about NUMA and other “kinds” of memory than pmem, such as HBW.

kernel: HMEM

A hybrid approach between hardware memory mode and manual allocations: in the absence of hints, the kernel will guess what you want, and migrate pages automatically, avoiding migrating a page you use just once.

as disk

You can use pmem same as a regular disk. Just

mkfs.ext4 /dev/pmem0

and that's… it?

write atomicity

Alas, not. Traditional disks, both HDD and SSD, guarantee block atomicity: a whole sector is either written completely or not at all. Persistent memory offers only 8-byte atomicity, which leads to torn writes on unaware software.

sector mode

Using ndctl, you can set a a region (“namespace”) into “sector mode” that does emulate a traditional block device, via something akin to FTL. But this has a speed penalty.

fsdax

Once you know no software you use relies on such block atomicity, you can do all writes in-place. But that still requires the old read()+write() interface (or mmap() with msync()s).

DAX

But, as pmem is memory, can't you write to it… directly? Sure you can: mount with -o dax (supported by ext4 and xfs, soon btrfs) but the processor will then keep your data in its cache for a long time, potentially forever.

userspace flushes

After writing to memory, you can use an unprivileged instruction like CLFLUSH, CLFLUSHOPT or CLWB to tell the processor you want that data to committed to the memory (it was in L1/L2/L3 cache until then). As this is highly processor-dependant, use libpmem for that.

flushing, ordering, etc

Alas, even with libpmem, you need to observe discipline, always flushing (and draining the flush) every thing you wrote.

This tends to be hard to get right.

persistent leaks

The vast majority of uses need some kind of allocations. But, once you allocate a piece of memory, you need two actions: to mark it as allocated, and to link it to whatever data structure you use. And the power loss can happen at any moment.

libpmemlog

There are some special cases, like an append-only (or ring) log. Here, you can just write past the end of already used memory, flush, then update the end-pointer. Thanks to 8-byte atomicity, that's easy to do.

libpmemblk

Another common case is a set of uniform-sized blocks that need to be written atomically. This can be done by libpmemblk which uses a sort of FTL-like structure.

libpmemobj

And for the complex case, when you have objects of different sizes, there's pmemobj which does the allocations and persistency for you.

atomic API

Within libpmemobj, one of the APIs can do the aforemented crash-safe allocating and linking.

You reserve a piece of memory, initialize it, then once you're done, you request to atomically mark that reservation as allocated, and link it into your structure.

transactional API

Or alternatively, you can request a series of operations to be done as a transaction. All writes you do are logged, allowing for previous contents to be restored in case of an unexpected power loss.

```
 and NEWS files.
 .
 It can be run on several .deb archives at a time to get a list of all
 changes that would be caused by installing or upgrading a group of
 packages. When configured as an APT plugin it will do this
 automatically during upgrades.

Package: apt-utils
Status: install ok installed
Priority: important
Section: admin
Installed-Size: 1083
Maintainer: APT Development Team <deity@lists.debian.org>
Architecture: arm64
Source: apt
Version: 1.8.2
Depends: apt (= 1.8.2), libapt-inst2.0 (>= 1.0.5), libapt-pkg5.0 (>= 1.3~rc2)
ibc6 (>= 2.17), libdb5.3, libgcc1 (>= 1:3.0), libstdc++6 (>= 5.2)
Description: package management related utility programs
 This package contains some less used commandline utilities related
 to package management with APT.
 .
```

libpmemobj-cpp

Alas, using libpmemobj from C code is unwieldy, requires obscure syntax, peppered with macros. C++ allows wrapping that complexity.

The library also provides a set of STL-like containers.

Persistency is overrated

But, why can’t I just use an UPS? Yes, UPSes fail, but so do disks, and these fancy new DIMMs.

Persistency is overrated

… unless you can replicate it

pool replicas

All the high-level libraries (libpmemlog, libpmemblk, libpmemobj) can duplicate operations you do into multiple pools. The pools may be somewhere else on the same machine (local), or...

RDMA

A pool can be accessed remotely. This can in theory be done using a regular network, but with speeds involved, there's no point to even bother.

But, a RDMA-capable NIC ("RNIC") can copy memory writes from one machine, send it to another, and persist without the CPU even knowing about that.

RDMA

- In buster: one-way replication only. The target system can't even read the data while it's used as a replication slave. It's good for a hot standby.
- Being implemented: 1 read-write node, slaves can read but not write.
- Future: ...?

use of this stack

As of Buster, no outside package uses the PMDK stack yet. There's upstream code in qemu, fio and ceph (not enabled in Debian yet) – but most of the support still needs to be written. There's a bunch of unmerged upstream patchsets, forks, and so on…

No ETACS yet.