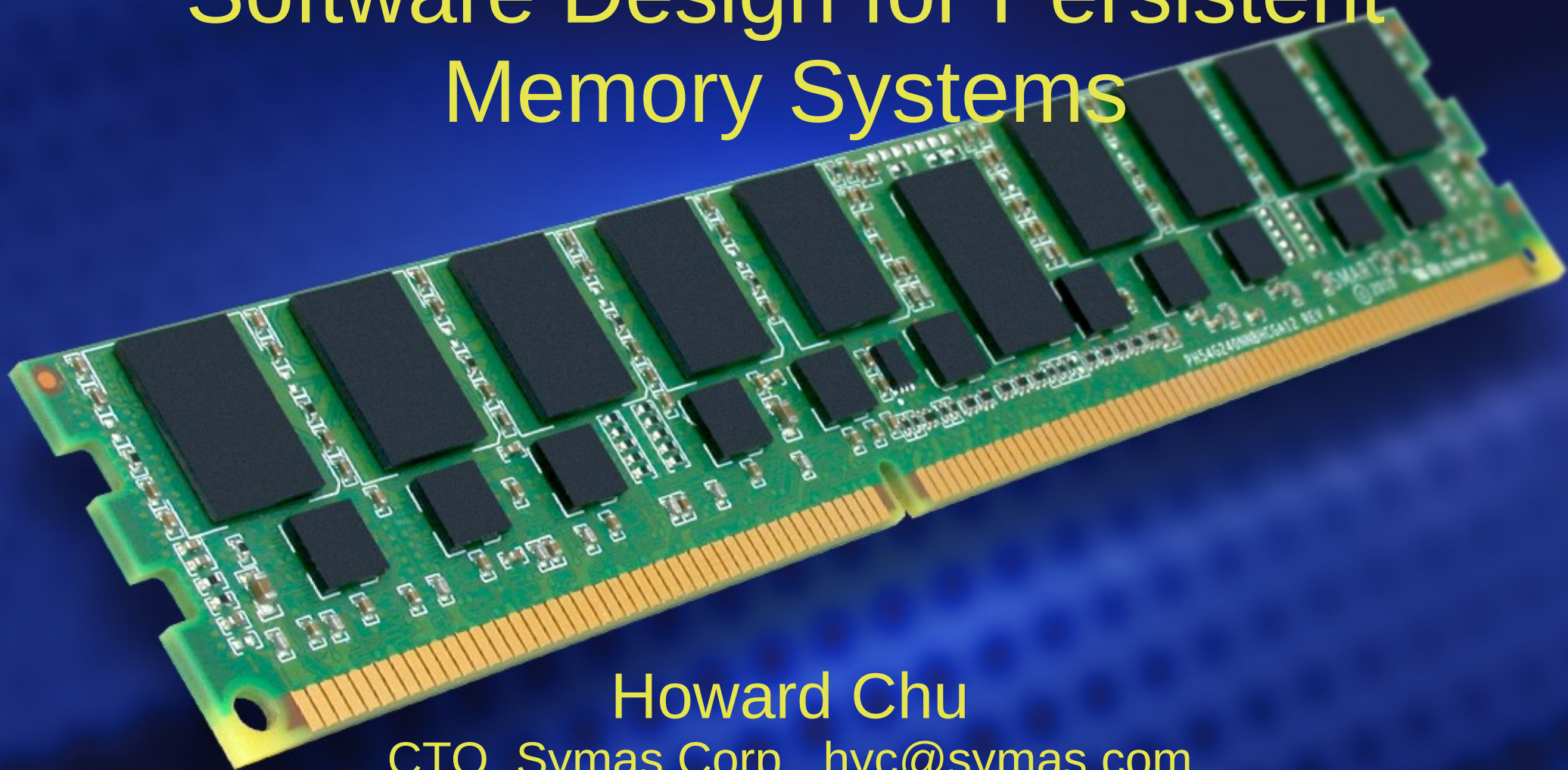




Software Design for Persistent Memory Systems



Howard Chu

CTO, Symas Corp. hyc@symas.com

2018-03-07

Personal Intro

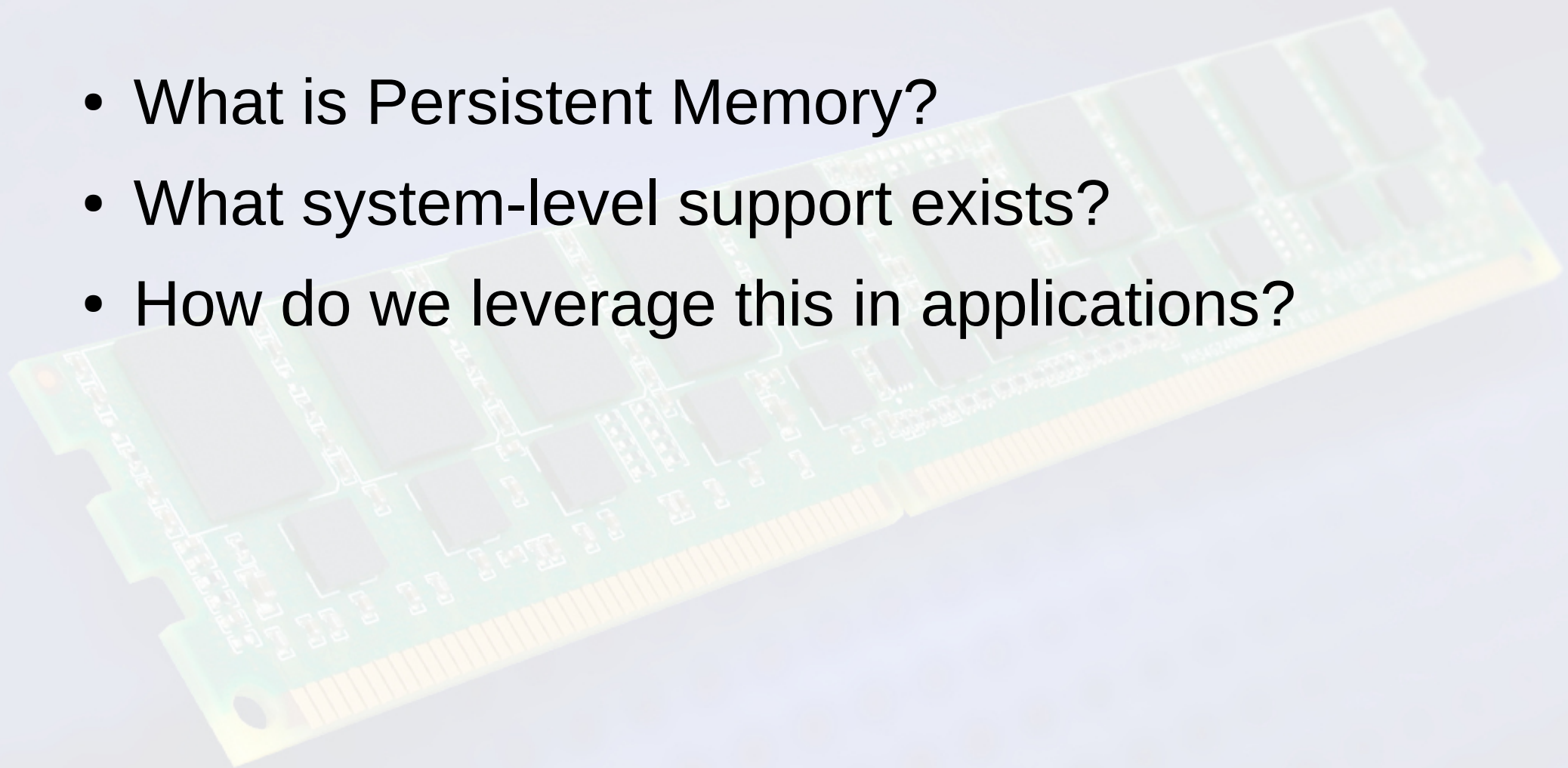
- Howard Chu
 - Founder and CTO Symas Corp.
 - Developing Free/Open Source software since 1980s
 - GNU compiler toolchain, e.g. "gmake -j", etc.
 - Many other projects...
 - I never use a software package without contributing to it
 - Worked for NASA/JPL, wrote software for Space Shuttle, etc.

Personal Intro

- Career Highlights
 - 2011- Author of LMDB, world's smallest, fastest, and most reliable embedded database engine
 - 1998- Main developer of OpenLDAP, world's most scalable distributed data store
 - 1995 Author of PC-Enterprise/Mac, world's fastest AppleTalk stack and Appleshare file server
 - 1993 Author of faster-than-realtime speech recognition using Motorola 68030
 - 1991 Inventor of parallel make support in GNU make

Topics

- What is Persistent Memory?
- What system-level support exists?
- How do we leverage this in applications?




What is Persistent Memory

- Non-volatile, doesn't lose contents when system is powered off
- Can be thought of as battery-backed DRAM
 - billed as byte-addressable storage, but really is still constrained to cacheline granularity
 - being used as a new layer in system memory hierarchy, between regular DRAM and secondary storage (SSD, HDD)
 - ideally, will replace regular DRAM completely

What is Persistent Memory

Competitive Memory Component Landscape

Characteristics	STT-MRAM	DRAM	3D Xpoint	Resistive RAM	Low Power CBRAM	NAND
Supplier		Multiple	Intel/Micron	Crossbar	Adesto	Multiple
Latency R/W	70ns/70ns	40-180ns ¹	100ns/500ns	100ns/100µs	1µs/25-100µs	25µs - 1500µs ²
Endurance	10 ¹⁰ -10 ¹²	10 ¹⁵	10 ⁶ -10 ⁷	10 ⁵ -10 ⁶	10 ⁵	10 ² -10 ⁵
Persistent	Yes	No	Yes	Yes	Yes	Yes
Interface	ST-DDR3/ST-DDR4	DDR3/DDR4	Proprietary	Flash-like	SPI	NAND
Status	Shipping	Shipping	Shipping Optane	R&D	Production	Shipping
Density Path	Gigabit+	Gigabit+	64Gb+	Terabit	64Mb	Gigabit+

- Pre-production STT-MRAM startups:
 - Avalanche Technologies – 32Mb SPI interface NOR Flash alternative, not shipping volume
 - Spin Transfer Technologies – Technology demonstration only
 - Crocus – magnetic sensor focus



1. Read/write symmetric, latency spans bus idle and bus busy
 2. Fastest read latency for SLC; slowest write latency for TLC

What is Persistent Memory

- STT-MRAM is the leading technology for now
 - performance equivalent to DRAM
 - endurance approaching DRAM (10^{12} vs 10^{15} writes)
 - ST-DDR3, ST-DDR4 DIMMs available - drop-in compatible with DDR3/DDR4
 - Still lags in density, 256Mbit parts reaching market now
 - Fabricated on 40nm process
 - Compared to 8Gbit DDR4 DRAM chips already mainstream, on 10nm process
 - Production on 22nm process expected later this year

What is Persistent Memory

- Other possibilities exist
 - actual battery-backed DRAM DIMMs (BBU DIMM)
 - offered up to 72 hours of persistence
 - deprecated, no longer marketed
 - Flash-backed DRAM DIMMs (NVDIMM)
 - typically with a super-capacitor onboard
 - copies DRAM to flash on system shutdown
- All of these are more expensive than regular DRAM

System-Level Support

- Requires both BIOS and OS support
 - POST must use non-destructive memory test, or just skip memory test
 - Kernel must recognize NV memory
 - Linux kernel boot args can be used to explicitly mark memory as persistent
 - Current state of OS support is extremely primitive

System-Level Support

- Kernel treats persistent memory as a block device
 - you can create a filesystem on top and use it as a glorified RAMdisk
 - Congratulations, welcome to the state of the art of 1986.
 - you can use it as cache dedicated to a particular set of devices
 - using dm-cache, bcache, flashcache, etc.
 - but these solutions are written for Flash SSDs, and aren't optimal for persistent RAM
 - current designs assume only a small subset of system memory is persistent

System-Level Support

- Future support must account for systems with 100% persistent memory
 - Kernel page cache manager must be modified to utilize hot cache contents left by previous bootup
 - "persistent memory" must become just "memory" - used for system-wide device caching, instead of isolated in its own block device

System-Level Support

- Whether system is 100% persistent RAM or not, memory should be managed by kernel and not require direct management at user level
 - current usage as distinct block device requires a user to manually manage it
 - explicitly copy files to it
 - when the space gets full the user must choose some files to delete, in order to make room for new files
 - instead, used as part of the system cache, the OS can page data in and out as needed, without any user intervention

Application Design

- Mindset
- Design Concepts
- Implementation Choices
- Other Details
 - Concurrency Control
 - Free Space Management
 - Byte Addressability
- Endgame

Application Design

- Requires a different mindset
 - Should not view "memory" and "storage" as distinct concepts - must adopt "single-level store"
 - Storage and RAM are interchangeable, via memory-mapping
 - Data structures that are intended to be persistent must be written atomically - interruption of updates must not leave corrupt or inconsistent states
 - Avoid temptation to take "memory-only" / "main memory" design approach

Application Design

- Problems with "main memory" approach
 - A law of computing: data always grows to exceed the size of available space
 - There will always be larger/slower/cheaper memory in addition to fast in-core memory: there will always be a hierarchy of storage
 - You must design for growth, and take this hierarchy into account

Design Concepts

- Essentially, persistent data structures must provide ACID transaction semantics
 - persistent RAM gives Durability, implicitly
 - the rest is up to you
- Atomicity can be actual, or effective
 - Actual: you only support modifications that can be performed with a single atomic update
 - Effective: you use undo/redo logs to allow recovery from interrupted updates

Design Concepts

- If you go for "effective atomicity" you'll need to have complex locking mechanisms to protect intermediate update states
- Once you go down the path of complex locking, you also have to deal with deadlocks, backoffs, and retries
- All of this involves a great deal of additional code on top of the actual data structure code
- Complex locking will not scale well across multiple CPU sockets

Design Concepts

- If you use undo/redo logs you'll need to build a robust crash detection mechanism, as well as a crash recovery procedure to recover from incomplete transactions
- The undo log will also be needed to execute transaction abort/rollback in normal (non-crashed) operation
- The log will be a central bottleneck in all write operations
- Logs will need explicit management - pruning/etc

Design Concepts

- Better approach is to use MVCC (Multi-Version Concurrency Control) with a single pointer to the current version
 - Once a new version has been constructed, a single atomic write to the version pointer can be used to make it visible
 - Since each transaction operates on its own version of the data structure, transactions have perfect Isolation

Design Concepts

- Best solution, based on constraints so far:
 - data structure must be storage oriented, for growth - not a memory-only structure
 - data structure must have atomic update visibility
- Use a B+tree
 - inherently suited to caching, memory hierarchy
 - using Copy-on-Write, can expose a new modification simply by updating a pointer to the root of a new tree version
 - a new update can be simply aborted/rolled back just by omitting the pointer update, no undo/redo logs needed

Implementation

- Successful implementation requires explicit control over memory layout of data structures
 - structures must be CPU cacheline aligned, both for performance and for integrity
 - this precludes implementing in most higher level languages

Implementation

- We're now clearly talking about a storage library
 - there's a lot of details to manage, but they can be hidden in a library
 - written in a low level language
 - should use something like C
 - easily callable from any other language
 - mature, portable, flexible
 - direct control over memory layout
 - allows identical layout for "in-memory" and "on-disk" representation

More Design Choices

- Multi-process concurrency, or just multi-thread?
 - Multi-thread in a single process is simpler
 - doesn't require shared memory for interprocess coordination
 - Multi-process concurrency is more flexible
 - allows administrative tools to query and operate regardless of whether the main application is running
- Single-writer or multiple writer?
 - Single-writer is simpler, eliminates possibility of deadlocks
 - Multi-writer requires complex locking, conflict detection
 - and still boils down to single-writer anyway, given the requirement of atomic visibility

Implementation

- Use mmap to expose data to callers
 - Use a read-only mmap, otherwise random overwrites will be persisted, causing unrecoverable corruption
 - Pointers to data in map can be returned directly to callers on data fetch requests, thus avoiding expensive malloc/copy operations
 - This requires that data values are always stored contiguously, even if values are larger than B+tree page size

Implementation

- Can optionally use writable mmap
 - Opens a window to corruption vulnerability
 - Requires explicit cache flush instructions, to ensure writes are pushed from CPU cache out to RAM (if not using msync)
 - No performance benefit over readonly mmap
 - writing a page requires that it first get faulted in, wasted effort if the entire page is going to be overwritten
 - May not be worth the cost in reliability and portability
 - forcing a CPU cache flush is highly system-dependent

Concurrency Control

- Systems commonly offer reader/writer semantics
 - 1 writer can operate exclusively, or arbitrary number of readers
 - writer and readers cannot operate simultaneously
- Done properly, an MVCC-based design allows readers to run wait-free, taking no locks
 - writer should be able to operate concurrently with arbitrary number of readers

Free Space Management

- With MVCC, storage space rapidly fills up with old/obsolete versions of data
- Most applications will have no use for the old versions
- Reclaiming space from obsolete versions will be critical for long term usability
- "Background" garbage collection (GC) is a commonly practiced approach but is not viable

Free Space Management

- Background GC assumes there's always spare CPU and I/O capacity
 - GC can consume more CPU and I/O bandwidth than the actual user workload
 - which then leads to requiring complex runtime profiling and throttling implementations
 - Thus it will either require over-provisioning of system resources, or GC will always cause user-visible pauses in processing
- Better to track page usage in foreground and reuse old pages when they become available
 - Yields consistent write throughput without any pauses

Free Space Management

- Tracking page availability has a direct impact on concurrency
 - Must record which readers are referencing which old versions, to know which old versions can be purged/reclaimed
 - Could just use a simple counter, recording the oldest version still in use
 - but accessing the counter becomes a bottleneck for readers
 - Better to use an array with one slot per reader
 - array slots must be cacheline aligned
 - slots can be updated by readers and checked by writers without taking any locks

Byte Addressability

- Highly touted feature of NVRAM-based storage
- Largely a red herring
 - Can be useful for current RAMdisk-style approaches, but these are evolutionary dead ends
 - Eventually the industry will wake up to the fact that reinventing reset-survivable RAMdisks was a waste of time and money
 - NVRAM will eventually be integral to the system cache, and the system cache is necessarily page-based

Endgame

- Based on the given design constraints:
 - atomicity, persistence, robustness, simplicity, efficiency
 - single-level store, blurring the line between memory and storage
- You'll end up with something that looks a lot like LMDB

LMDB Overview

- LMDB "Lightning Memory-Mapped Database"
- embedded key/value store implemented with a B+tree
- as the name indicates, it uses memory mapped data
 - defaults to read-only mmap
 - zero-copy reads: retrieved data points directly into mmap
 - zero-copy writes: optionally supports writable mmap

LMDB Overview

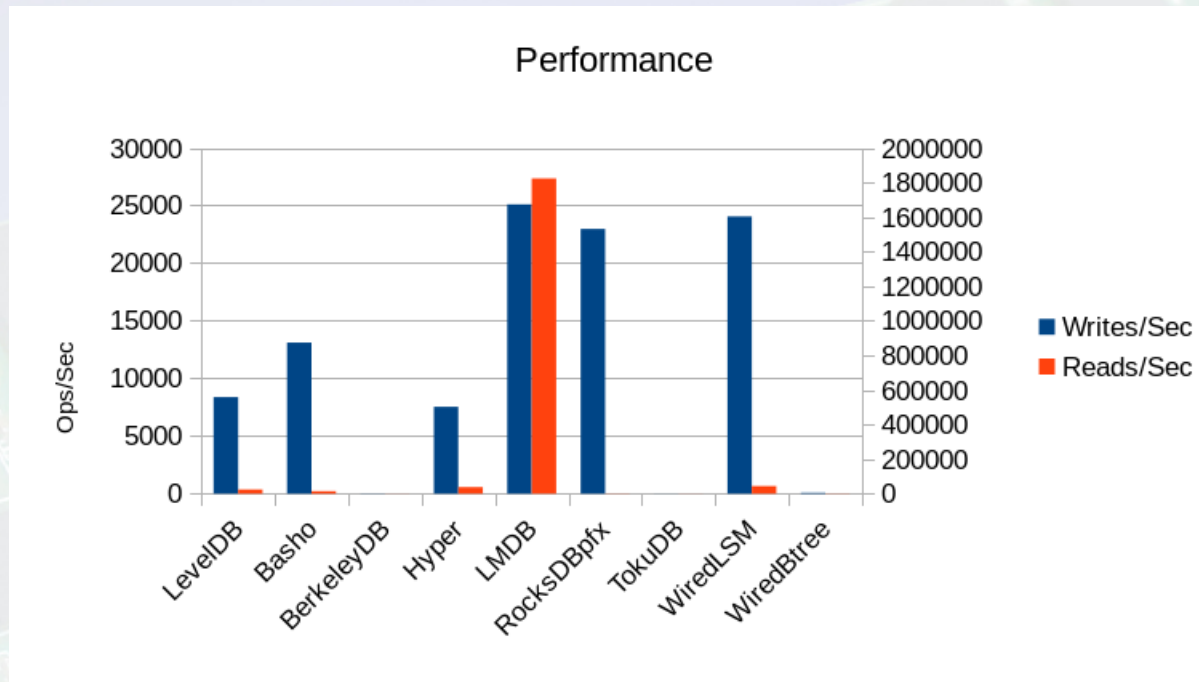
- full ACID transaction semantics
- MVCC concurrency control
 - writers don't block readers, readers don't block writers
 - a pair of page pointers are used to point to the current tree version
- single writer
 - no need for callers to handle deadlocks or retries

LMDB Overview

- No undo/redo logs
 - Uses Copy-on-Write
 - Intermediate tree states are never visible, cannot be corrupted by system crashes
- No garbage collection
 - space freed by a transaction is recorded in a 2nd B+tree living in the same space
 - writers reuse whatever available free space as needed
- No tuning or administrative overhead
 - zero-config

LMDB Overview

- Unrivalled read performance on any hardware and any data volume



- 1 billion record DB, ~120GB, on HP DL585 G5 with 128GB RAM, 16 cores
- 16 read threads concurrent with 1 write thread

Summary

- Persistent RAM is approaching price parity with regular DRAM, will be more common soon
- Current OS support is primitive and needs further improvement
- If you enjoy low level programming, the design constraints of writing an always-consistent data structure may be interesting to explore
- Otherwise, just use LMDB and don't worry about it

Questions?

