iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory

Qingrui Liu[1] Joseph Izraelevitz[2] Sekwon Lee[3]
Michael L. Scott[2] Sam H. Noh[3] Changhee Jung[1]

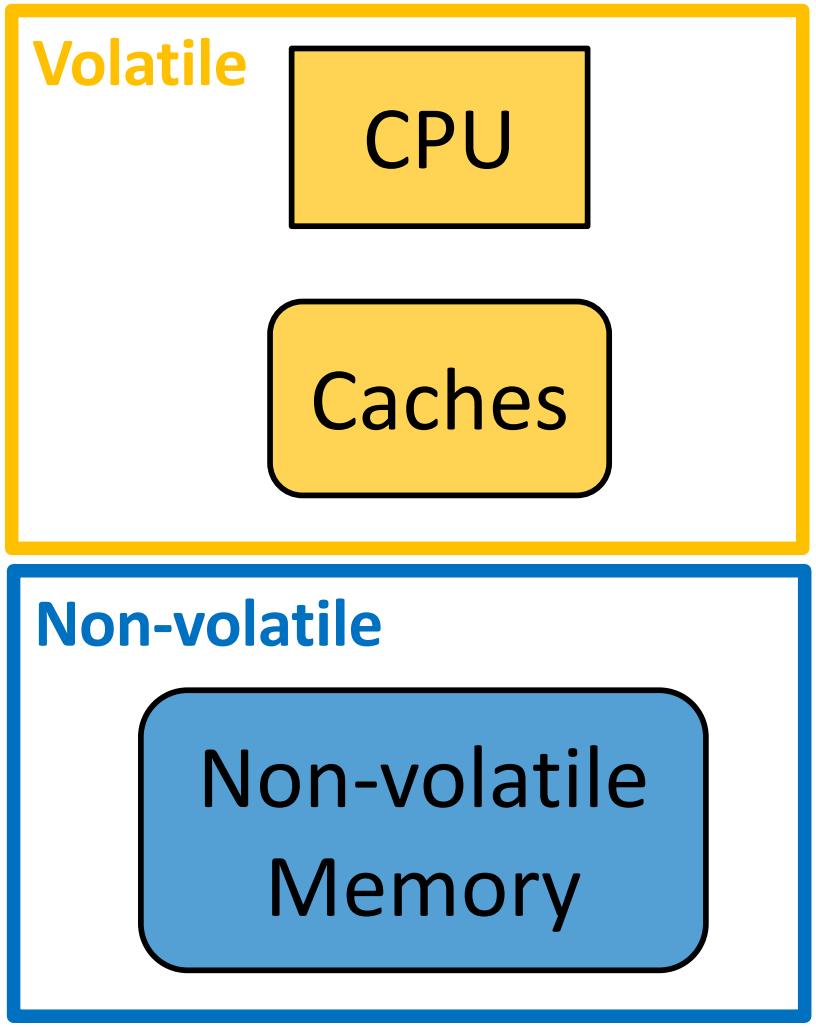
[1]Virginia Tech [2]University of Rochester [3]UNIST

CDP Workshop, CASCON, Oct. 2018
Originally presented at MICRO 2018

Nonvolatile Memory (NVM)

- DRAM near the end of its evolutionary life
  - power hungry; limited density
- Replacements likely to be nonvolatile (PCM, memristors, STT-MRAM)
- Envision machines with volatile registers and (for now) caches; nonvolatile DIMMs (maybe some DRAM, too)
- Tempting to leave long-lived data “in memory,” or to resume a crashed application from the NVM state
- Want to tolerate the same sorts of failures that file systems tolerate today

The Problem: Crash (In)Consistency




```
struct {
    int data;
    bool valid;
}
```

STORE data = 0x1111
STORE valid = true

Partial Solution: Ordering Writes

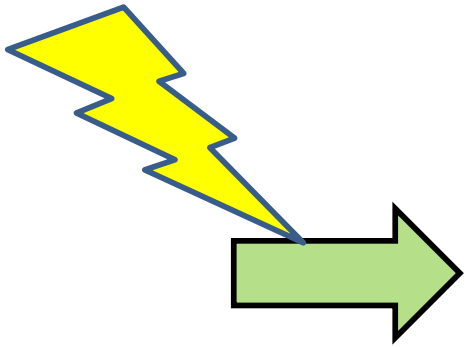
x86 Instructions (other ISAs are similar):

1. **Write back** (CLFLUSH, CLFLUSHOPT, CLWB)
   or **through** –– non-temporal store (MOVNTQ)
2. **Fence** –– memory fence (MFENCE or SFENCE)

STORE data = 0x1111
CLWB data
SFENCE
STORE valid = true
CLWB valid
SFENCE

But Ordering is Not Enough

Suppose x must always equal y



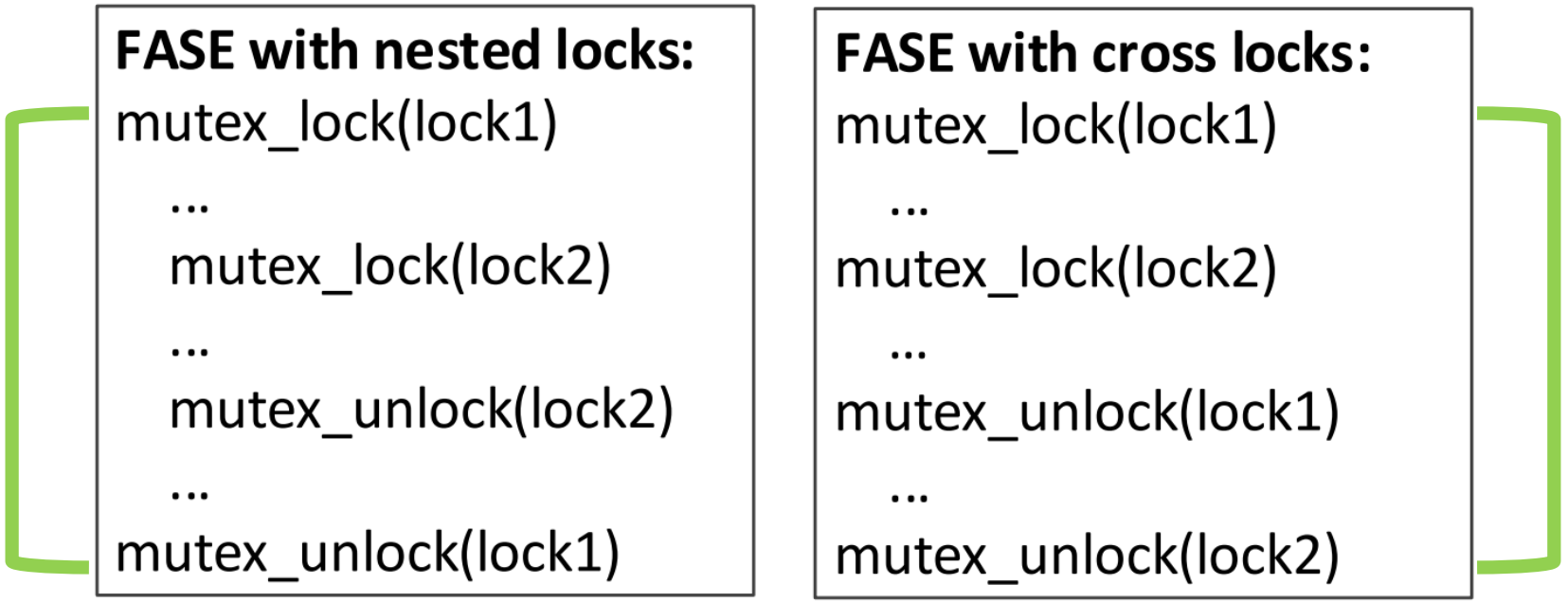
LOCK L
store x = 3
WB x
fence
store y = 3
WB y
fence
UNLOCK L

Need failure atomicity!

We assume lock-based source code

"FASE" (Failure-Atomic SEction)

[Chakraborti et al., OOPSLA'14]



Undo Logging

log old value of x
WB & fence
store x; WB
log old value of y
WB & fence
store y; WB
...
fence
mark log finished
WB & fence

Must track dependences across FASEs

Redo Logging

log new value of x
WB & fence
log new value of y
WB & fence
...
mark log complete
WB & fence
store x; WB
store y; WB
...
mark log finished
WB & fence

Must arrange to read our own writes

JUSTDO Logging [Izraelevitz et al., ASPLOS'16]

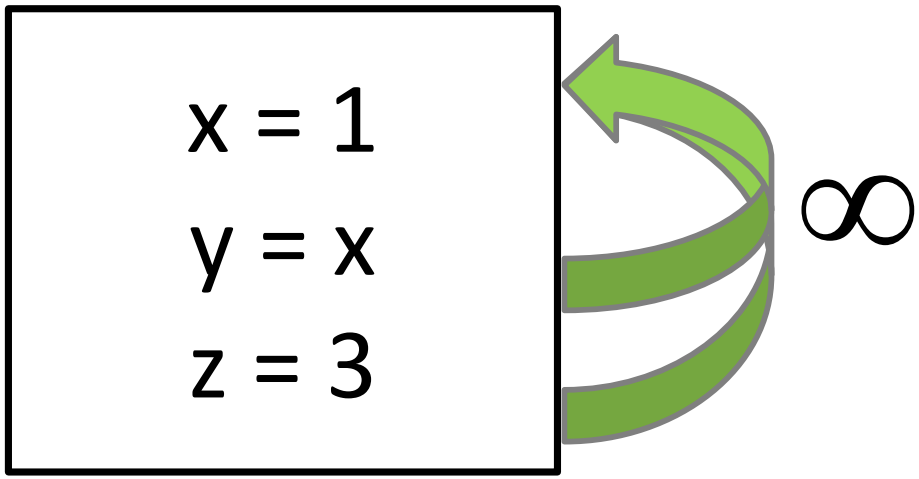
log new value of x, &x, PC
WB & fence
store x
WB & fence
log new value of y, &y, PC
WB & fence
store y
WB & fence
...

On recovery, *pick up at the most recent store*: use code of original program to execute from logged PC through end of FASE; release all locks.

- Log size is O(T+L) for T threads and L locks
- Must treat all data as "volatile" in FASEs
- WB & fence operations can be elided if caches are nonvolatile; **expensive on conventional machines**

Key Observation for iDO

A region of code is idempotent iff its prefixes can be re-executed multiple times and it will still produce the same result.



Output: x = y = 1; z = 3

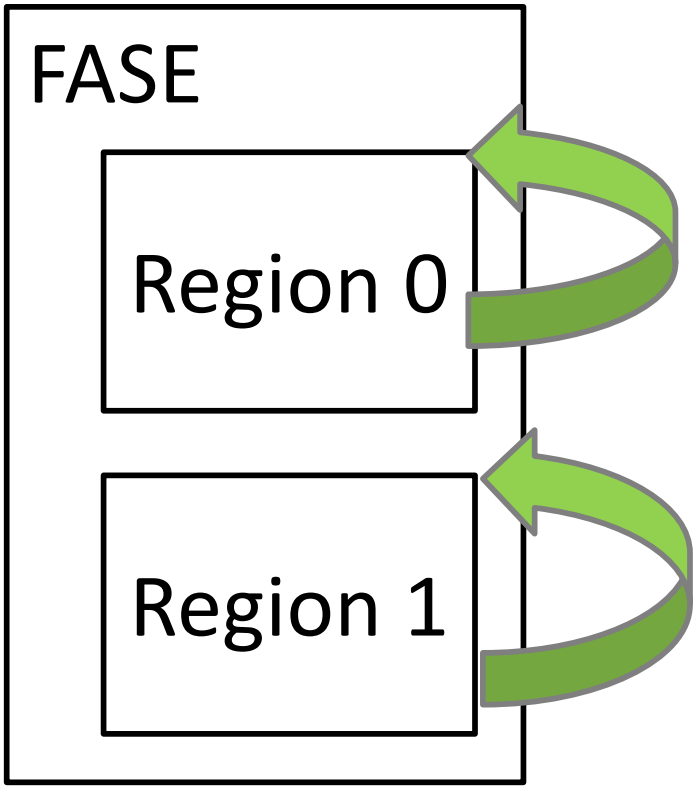
Don’t have to log at every store!

iDO Logging ≈ JUSTDO + Idempotence

Log space is still O(T+L)

On recovery, resume FASE at the beginning of the interrupted idempotent region

- No need for happens-before FASE tracking
- No need to take care to read own writes
- Small bounded log per thread



Idempotent Regions

- Leverage analysis of deKruif et al. [PLDI'12]
- Typical region is just a few stores
- Can be *very* large:

```
FASE{
  for (int i = 0; i < len; ++i)
    array[i] = i
}
```

- Could be extended with better alias analysis or code restructuring

Evaluation

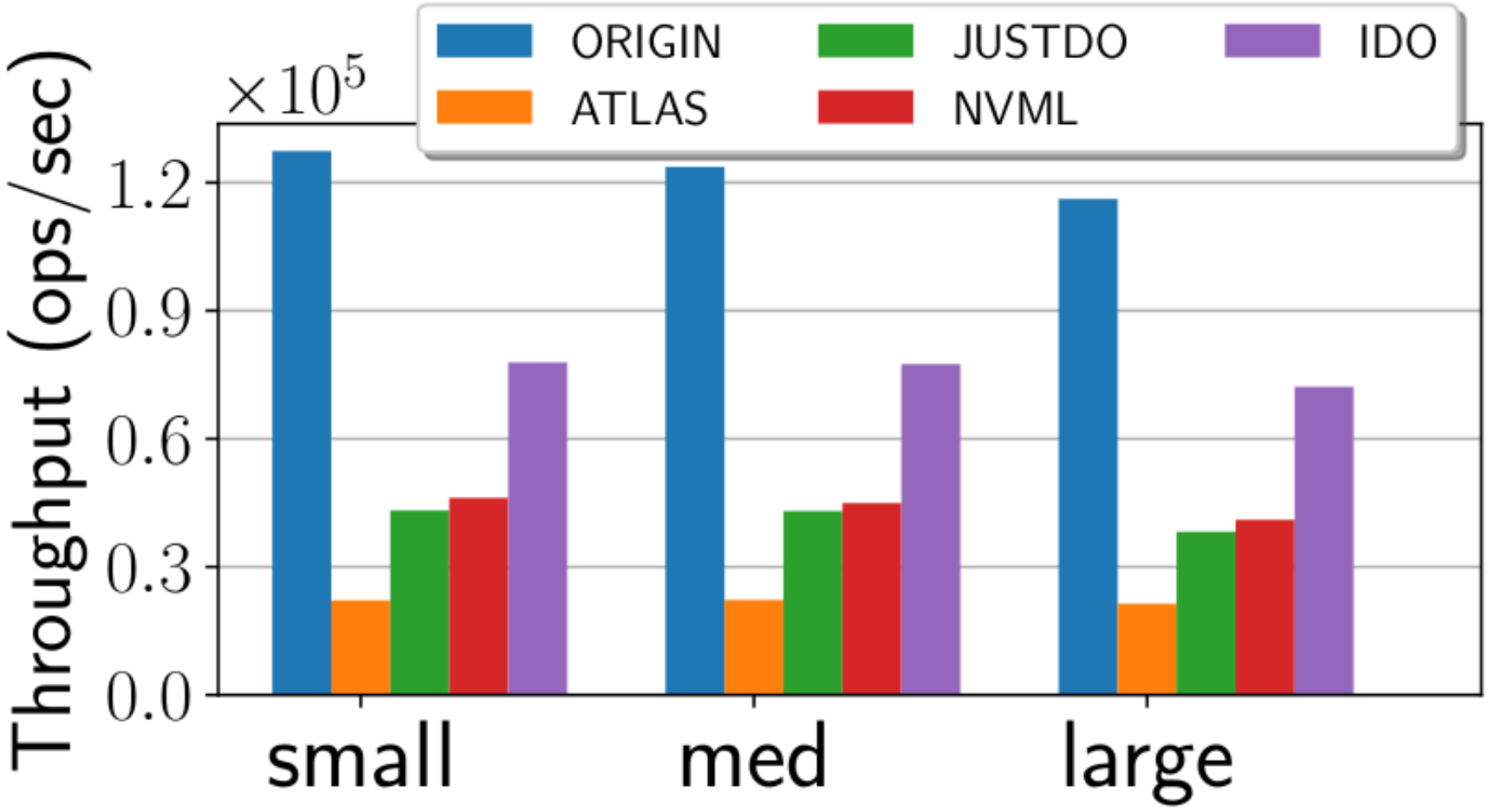
Compare iDO with:

- ATLAS [OOPSLA'14]: FASE + undo logging
- JUSTDO [ASPLOS'16]: FASE + resumption
- NVThreads [EuroSys'17]: FASE + copy-on-write
- Mnemosyne [ASPLOS'11]: Txns + redo logging
- NVML [FAST'15]: Txns + undo logging

Run on 4-socket, 64-core AMD Opteron 6276 server

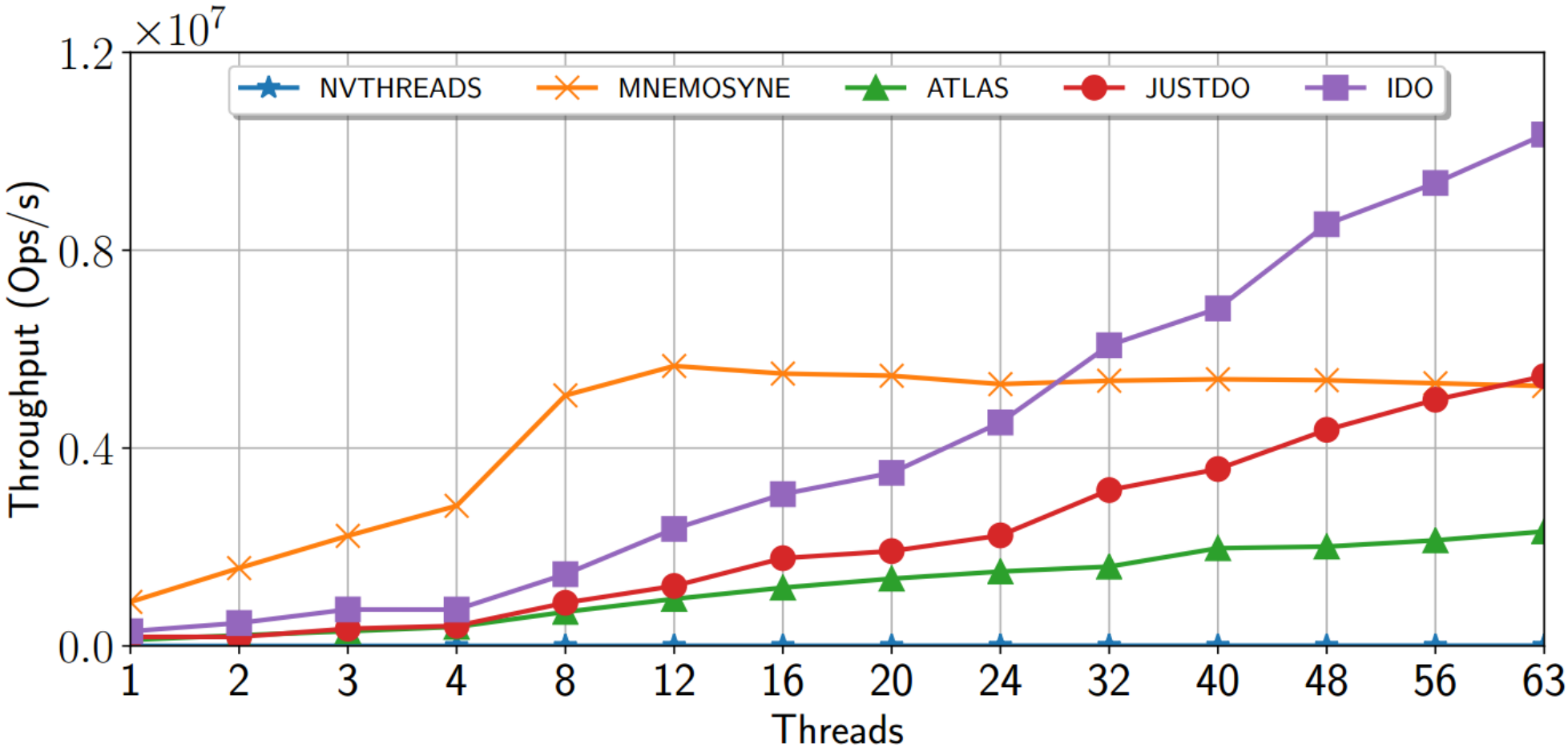
Assume CLFLUSH+SFENCE over DRAM ≈ CLWB+SFENCE over NVM; MICRO paper includes sensitivity analysis

Performance



Redis throughput for databases with 10K, 100K, and 1M-element key ranges (single threaded)

Scalability



Hash map

Conclusion

- Compiler-directed failure atomicity for data in nonvolatile memory
- Makes resumption-based recovery practical on machines w/ volatile caches
- Better performance than FASE-based undo and redo
- Excellent scalability
- Fast recovery

UNIVERSITY of ROCHESTER

MICRO paper available at:

www.cs.rochester.edu/research/synchronization/

www.cs.rochester.edu/u/scott/

Failure-Atomic NVM systems

ACID properties:

- **A**tomicity: All or nothing execution.
- **C**onsistency: defined by program semantics; when complete, transaction transitions from one consistent memory state to another.
- **I**solation: transactions cannot see each other's updates until their commit time.
- **D**urability (Persistency): updates of transactions survive crashes.