Mostly-Optimistic Concurrency Control

for Highly Contended Dynamic Workloads on 1000 cores

Tianzheng Wang, University of Toronto

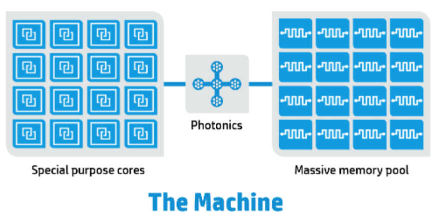
Hideaki Kimura*, Hewlett Packard Labs

* Currently with Oracle

OLTP on modern & future hardware

Multi-socket
Tens of cores

HPE Superdome X
16 sockets, 576 HW threads



HPE
The Machine

CRAY XC
with
Knights
Landing

Very high parallelism

Need ***lightweight*** concurrency control (CC)

Modern Optimistic CC 101

1. Local R/W
2. Verify
3. Commit/abort

Database A | B

Transaction 1	Transaction 2
Read set: {***A, B***}	Read set: {***A, B***}
Write set: {***A'***}	Write set: {***B'***}

Time

Transaction 2:

Read A
Read B
Write B'
Commit

Lock(B)
A, B changed?
B = B'
Unlock(B)

Transaction 1:

Read A
Read B
Write A'
Commit?

Lock(A)
A, B changed?
Unlock(A)
Abort

Hewlett Packard Enterprise
UNIVERSITY OF TORONTO

Why does OCC work well?

Only lock writes

➔ No shared memory writes for reads

Only lock at commit time

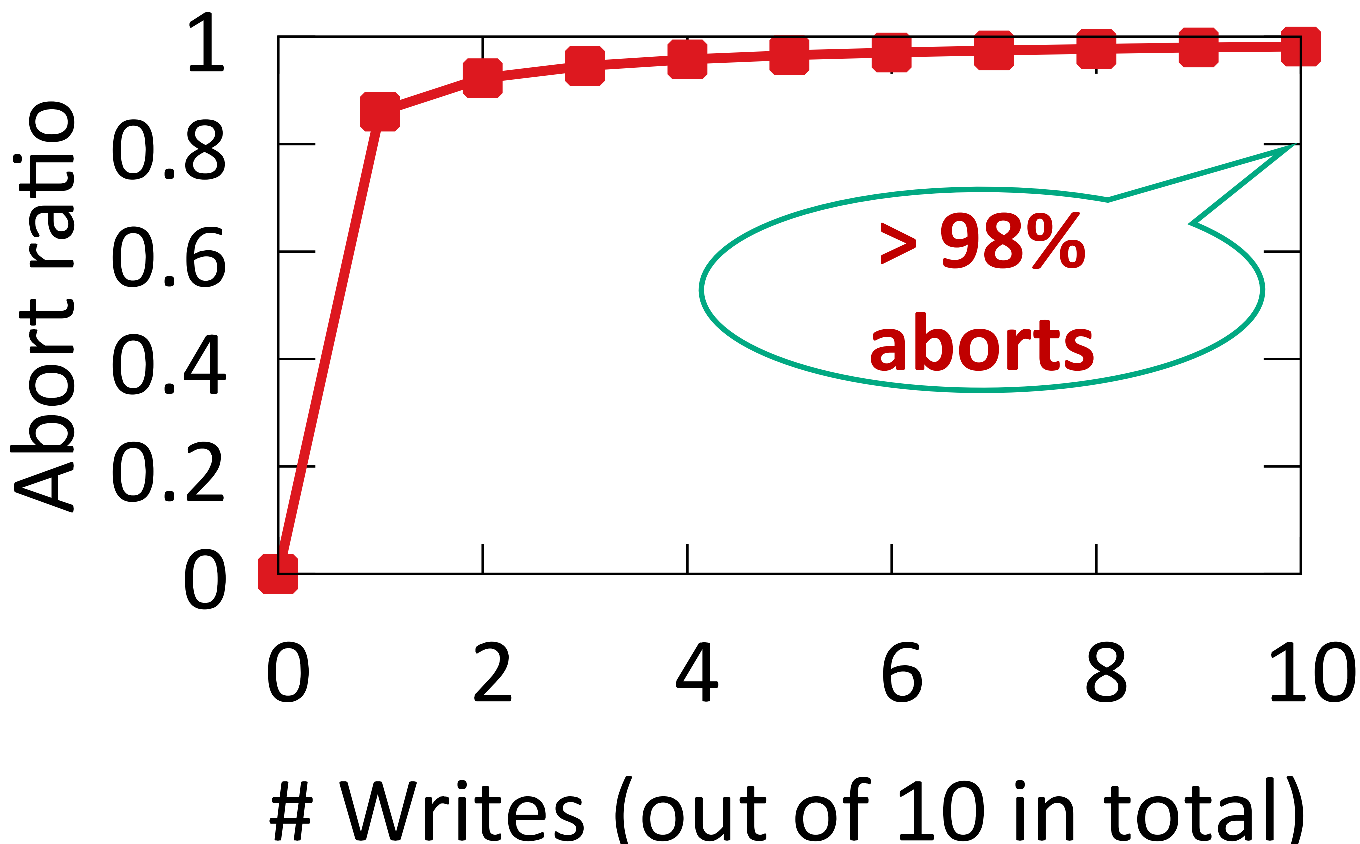
Sort writes before locking

➔ No deadlock possible

Simplifies lock implementation

High contention: OCC doesn't work

– 256 threads + 50 records YCSB, 10 ops/tx



Reads not protected

Mostly-Optimistic Concurrency Control

Key idea: protect hot records with locks

High-cont.: 2PL good at

Low contention: OCC good at

MOCC: best(2PL) + best(OCC)

Only lock *hot* records (keep OCC's benefits)

Must lock *as of* the access

Need better locks

Could deadlock

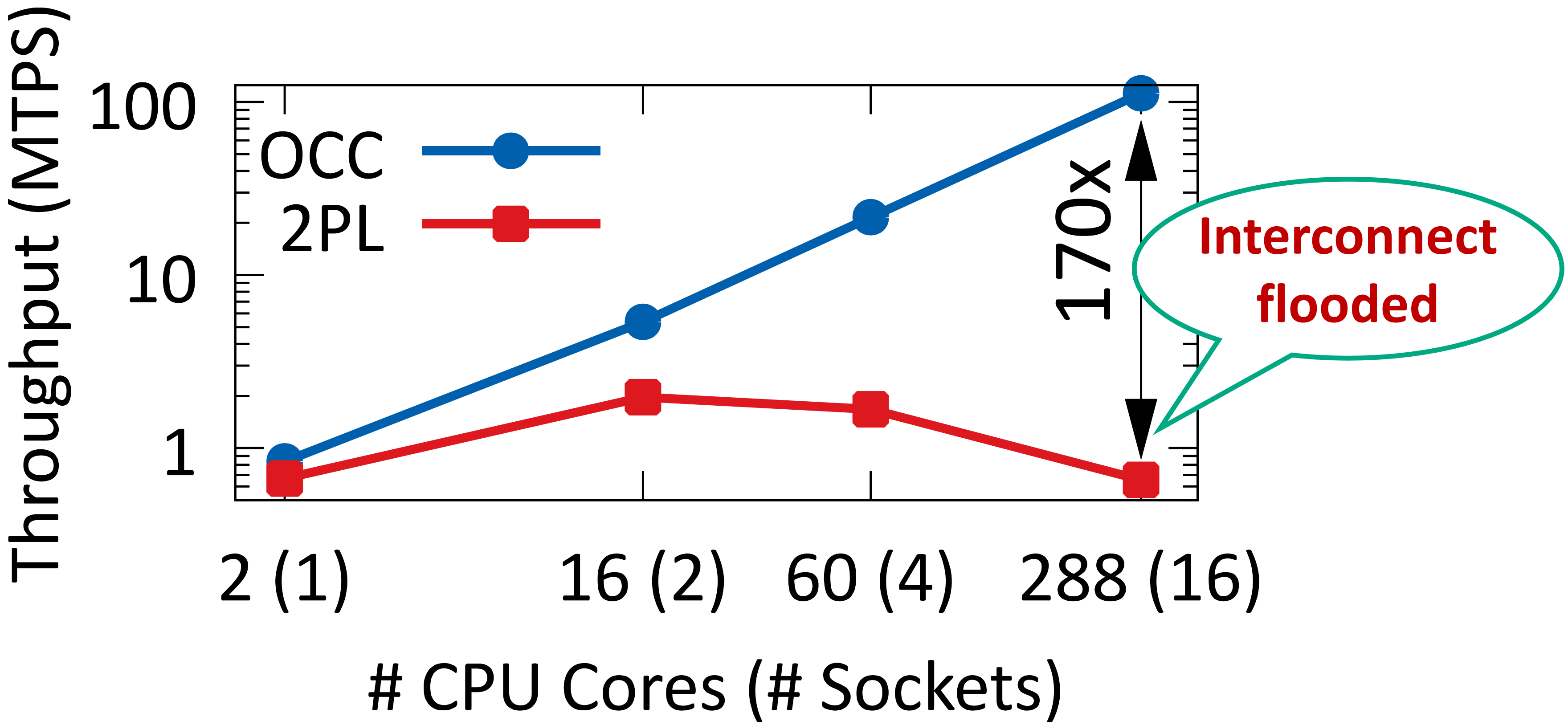
New sync. primitive

Hewlett Packard Enterprise

UNIVERSITY OF TORONTO

Must only lock hot records

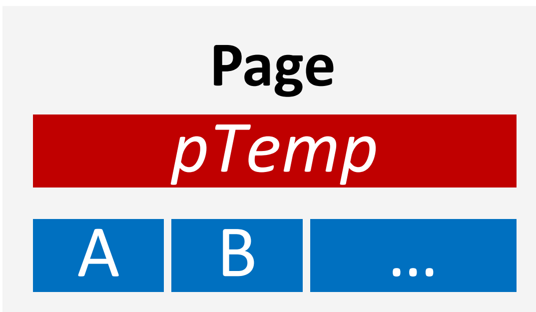
– ***Read-only***, 256 threads



Hewlett Packard Enterprise

UNIVERSITY OF TORONTO

Less physical contention with approximate counter*



Real temperature

$\sim= 2^{pTemp}$

Increment upon abort

with prob. = $1/2^{pTemp}$

Reduces cache line invalidation

Easy to tell really hot records/pages

Saves space

* R. Morris. Counting large numbers of events in small registers. *CACM 1978*

Hewlett Packard Enterprise

UNIVERSITY OF TORONTO

Lock(hot) re-introduces deadlocks

Hot records A B

Transaction 1

Read set: {*B*}

Write set: {*A'*}

Transaction 2

Read set: {*A*}

Write set: {*B'*}

Time

Write A — X-Lock(**A**)

Read B — S-Lock(**B**) - Wait for T2

Write B — X-Lock(**B**)

Read A — S-Lock(**A**) - Wait for T1

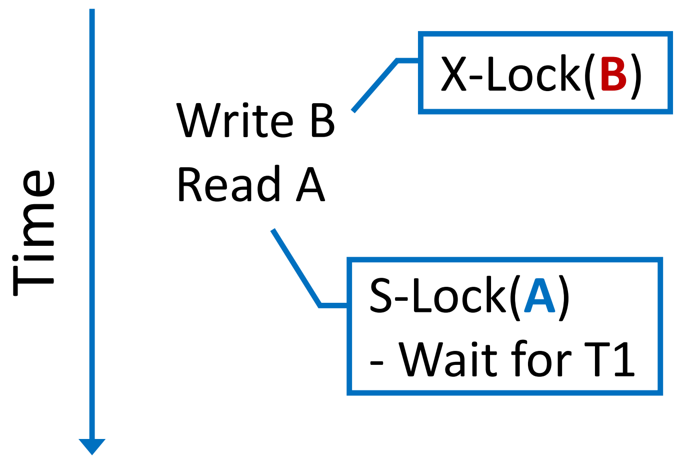
Worse: no control over application footprint

Hewlett Packard Enterprise

UNIVERSITY OF TORONTO

Problem: locks acquired out-of-order

i.e., Some locks acquired too early



What if T2 Unlock(B) now?

Canonical mode (CM): *All* locks acquired *in order*

Alphabetical, address…

Goal: keep transaction in canonical mode

Problems

1. ***Restore*** **canonical mode**
2. ***Maintain*** **canonical mode on retry**

Hewlett Packard Enterprise
UNIVERSITY OF TORONTO

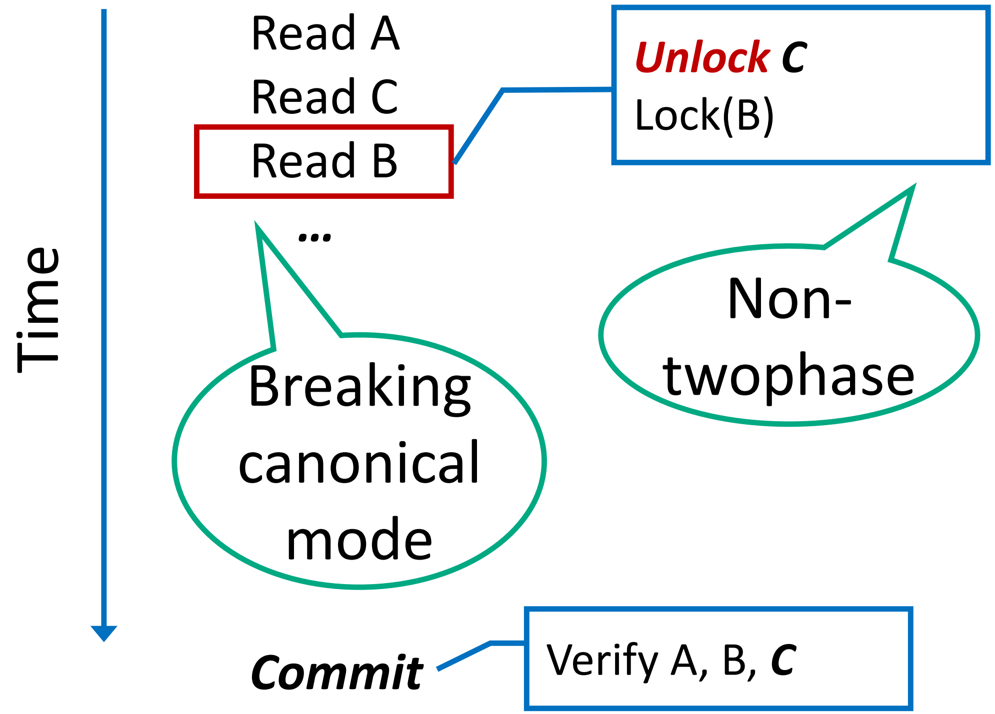
Restore canonical mode

Transaction 1

Read set: {***A, C***}

Write set: {}

Locks held: {A, C}



Non-twophase locking + OCC verification

Retrospective lock list: A safety net upon retry

Keep the footprint and lock at retry

1st run

Read set: {***A, C***}

Write set: {}

Locks held: {A, C}

Retry

Read set: {}

Write set: {}

Locks held: {}

Retro. list: {A, B, C}

Time

Read A

Read C

Read B

Abort

Hewlett Packard Enterprise

UNIVERSITY OF TORONTO

Retrospective lock list: A safety net upon retry

Keep the footprint and lock at retry

1st run

Read set: {***A, C***}
Write set: {}
Locks held: {A, C}

Retry

Read set: {***A, B, C***}
Write set: {}
Locks held: {}
Retro. list: {A, B, C}

Time

Read A
Read C
Read B
Abort

Read(A) – Check RLL, Lock A
Lock(***C***)?
...

Check RLL
Lock B
Lock C

No risk of deadlock

Hewlett Packard Enterprise
UNIVERSITY OF TORONTO

Native locking

– No centralized lock tables or blocking

– Synchronization primitive directly as database locks

Mode	Description	Use in MOCC
Read/-Write	Allows concurrent readers. Write is exclusive.	All cases
Unconditional	Indefinitely wait until acquisition.	Canonical mode.
Try	Instantaneously gives up. Does not leave `qnode`.	Non-canonical mode. Record access.
Asynchronous	Leaves `qnode` for later check. Allows multiple requests in parallel.	Non-canonical mode. Record access and pre-commit (write set).

– MOCC queuing lock = MCS RW + MCS timeout

Evaluation

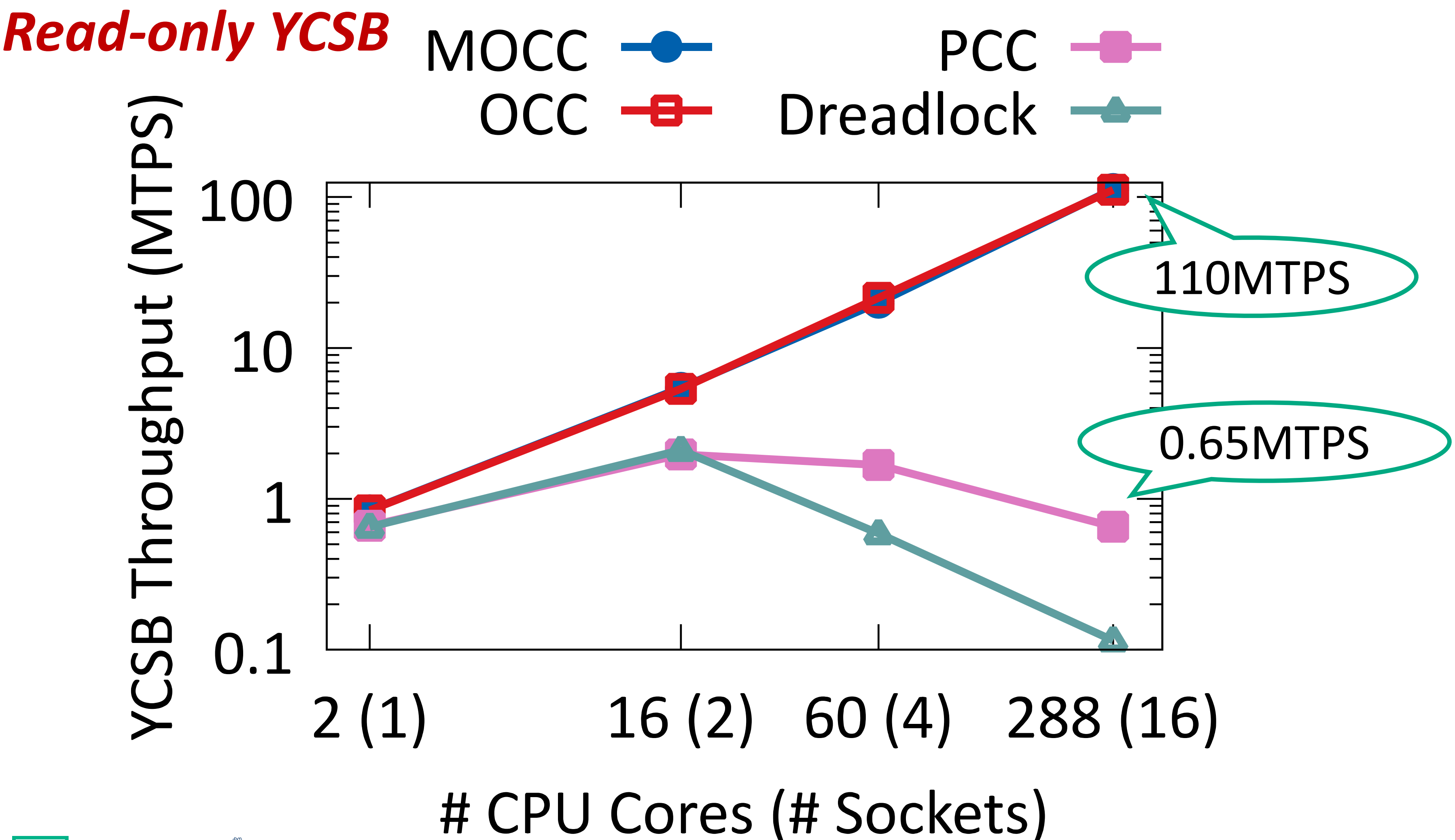
– HW: four machines of varying scale

Model	EB840	Z820	DL580	GryphonHawk
Sockets	1	2	4	16
Cores (HT)	2 (4)	16 (32)	60 (120)	288 (576)
Frequency	1.9 GHz	3.4 GHz	2.8 GHz	2.5 GHz

– YCSB for high contention workloads
  – 10 random RMWs, vary # of writes, 50 records

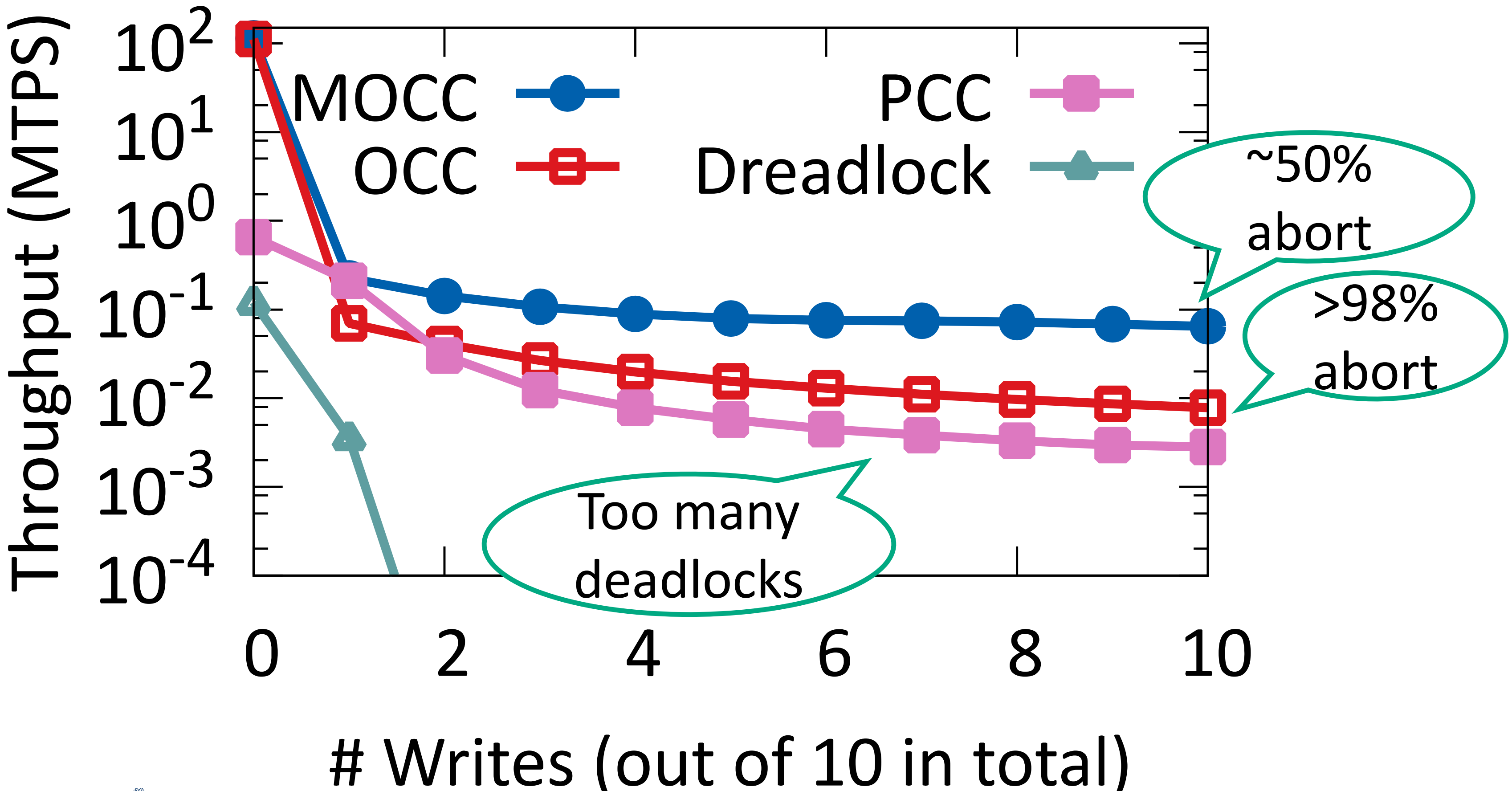
– More results/CC schemes in the paper
  – TPC-C: few conflicts ➔ same as OCC

MOCC keeps the best of OCC



Keeps away the worst of OCC

Read-write YCSB



Hewlett Packard Enterprise

UNIVERSITY OF TORONTO

TPC-C results

– Aggregate of all transactions

Scheme	Throughput [MTPS+Stdev]	Abort Ratio
MOCC	16.9±0.13	0.12%
FOEDUS	16.9±0.14	0.12%
PCC	9.1±0.37	0.07%
ERMIA	3.9±0.4	0.01%

Almost no overhead under low contention

Robust CC needed for OLTP

Mostly-optimistic concurrency control

= best(2PL) + best(OCC)

Protect hot records with locks

1. Approx. counter for temperature
2. Non-twophase lock + retrospective lock list
3. MOCC queuing lock

Find out more in our paper and code repo

https://github.com/HewlettPackard/foedus_code

Hewlett Packard Enterprise

UNIVERSITY OF TORONTO

Thank you!