Bridging the gap between transactional memory and two emerging hardware technologies: non-volatile memory & heterogeneous computing

Paolo Romano
romano@inesc-id.pt
Lisbon University & INESC-ID

Roadmap

- About me
- About IST & INESC-ID
- Introduction to Transactional Memory
- Transactional Memory & emerging HW technologies:
  - Non-Volatile Memory [IPDPS'18/JPDC'19]
  - Heterogeneous Computing [PACT'19]

About me

- MSc at Tor Vergata (2002)
- PhD at Sapienza (2004-2007)
- Senior Researcher at INESC-ID, Lisbon, Portugal (2008-today)
- Assistant Professor, Computer Engineering, IST, U. Lisbon (2011-2015)
- Associate Professor, Computer Engineering, IST, U. Lisbon (2015-today)

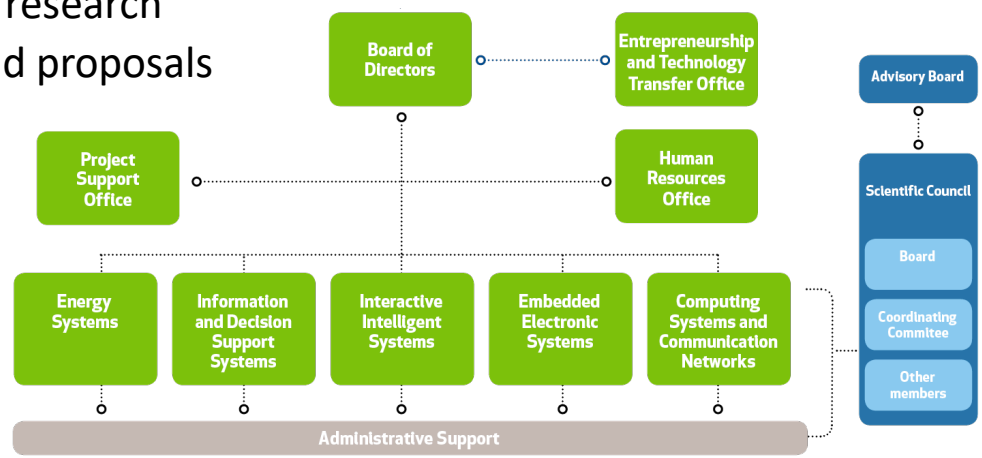
About IST

- IST, Lisbon University:
  - Top engineering school of Portugal
  - Two sites:
    - Alameda (Lisbon center)
    - Tagus Park (half-way to the Atlantic Ocean)

- Computer Engineering Department:
  - 91 Faculty members, 5 scientific areas
  - Pioneering open search process for faculty positions

About INESC-ID

- Research center affiliated with IST
  - Partly owned by IST
    - No-profit & private nature enables agile processes (e.g., hiring, purchases)
  - Hosts researchers (mostly IST faculty members) with diverse background
    - Strong impulse to pursue **interdisciplinary** research
    - Support for both project administration and proposals
  - 20th anniversary in 2019!



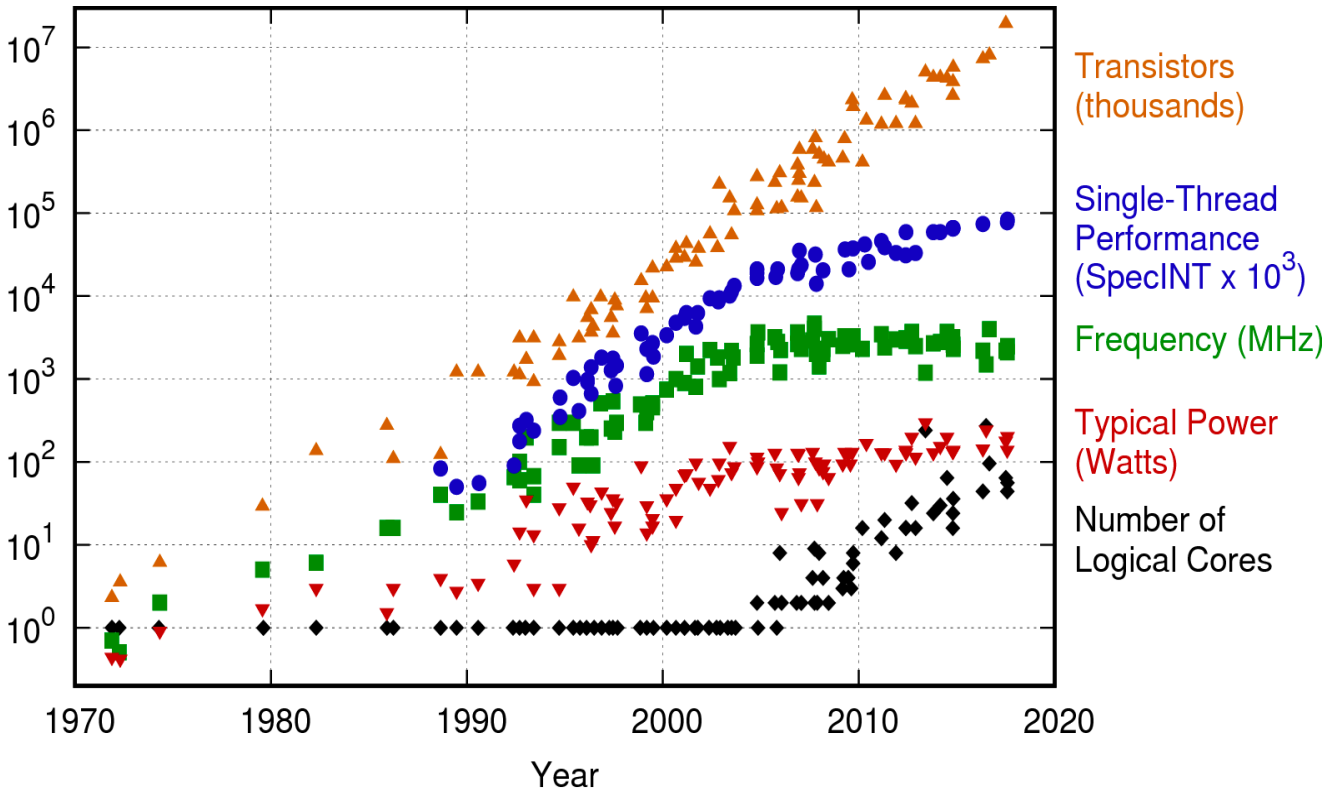
Roadmap

- About me
- About IST & INESC-ID
- Introduction to Transactional Memory
- Transactional Memory & emerging HW technologies:
  - Non-Volatile Memory [IPDPS'18/JPDC'19]
  - Heterogeneous Computing [PACT'19]

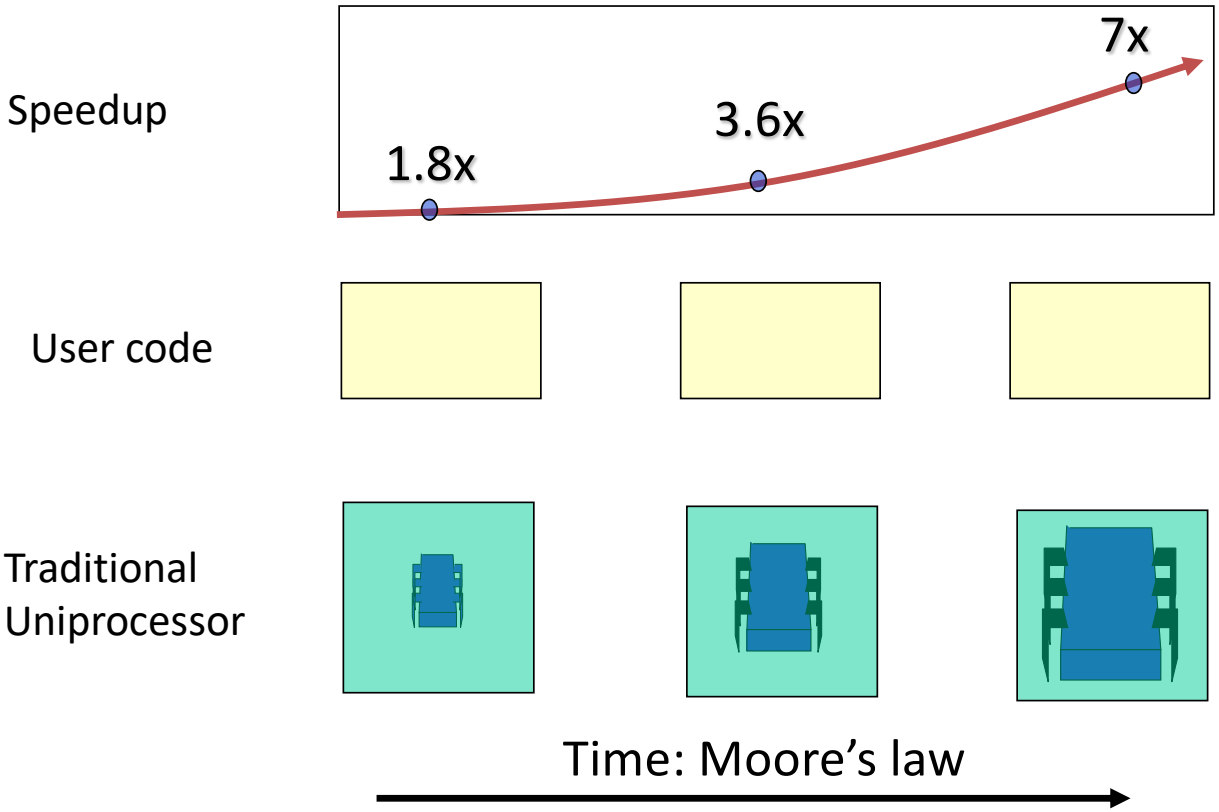
The era of free performance gains is over

- Over the last 30 years:
  - new CPU generation
    - ➔ free speed-up
- Since 2003:
  - CPU clock speed plateaued…
  - but Moore's law chase continues:
    - Multi-cores, Hyperthreading…

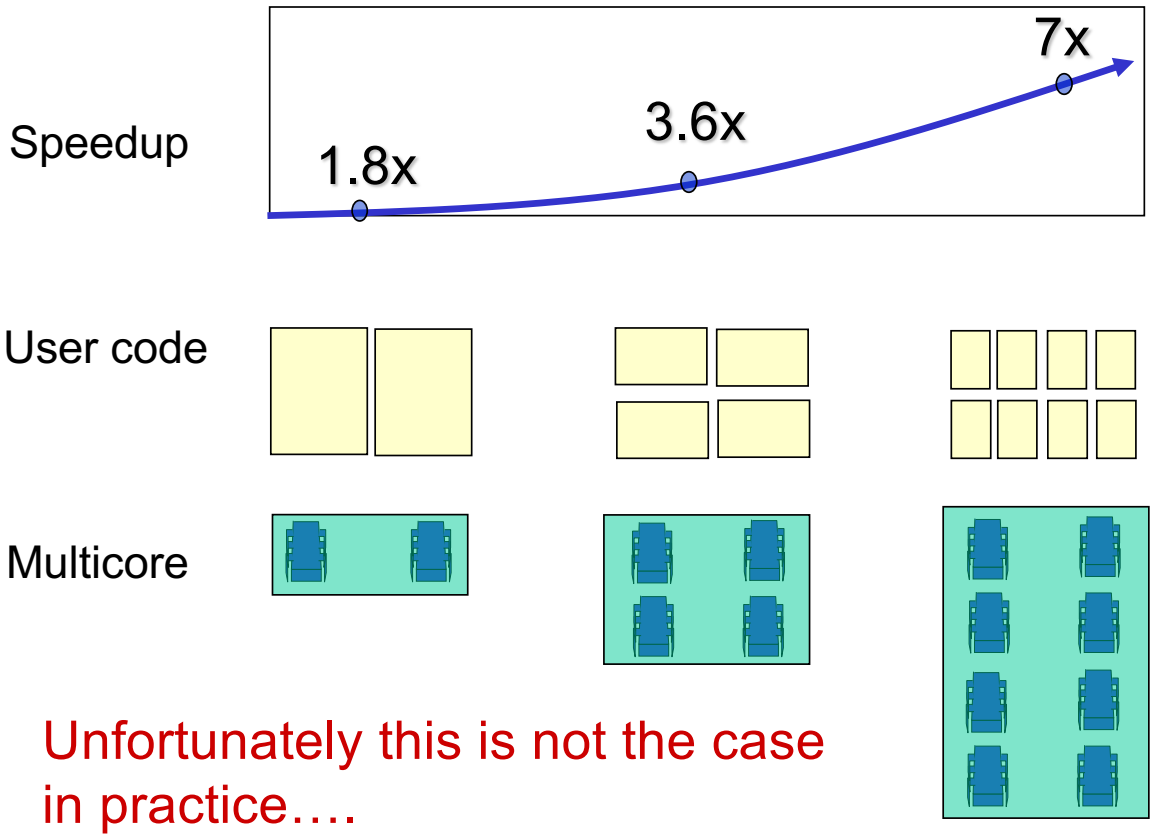
FUTURE IS PARALLEL



Traditional Software Scaling

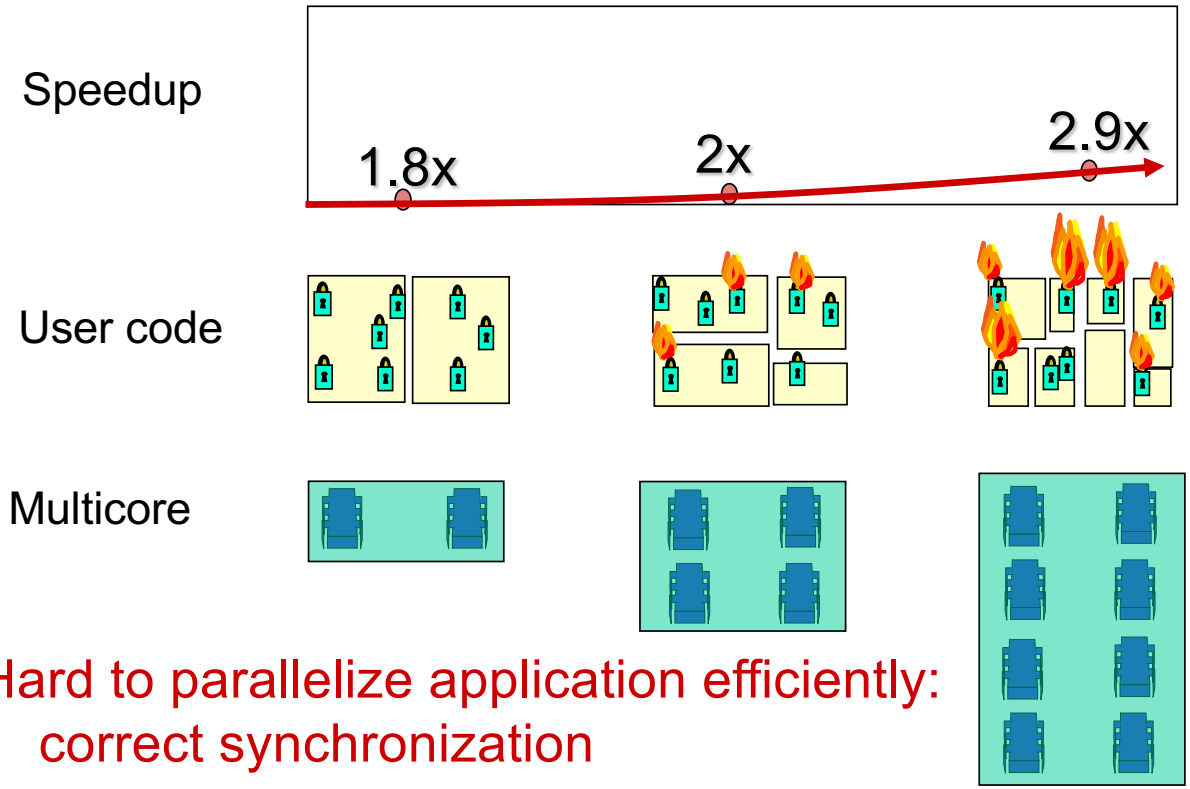


Ideal Multicore Software Scaling



Unfortunately this is not the case in practice….

Real-World Multicore Scaling



Hard to parallelize application efficiently:

- correct synchronization
- load balancing
- data locality

Coarse-grained Locking?

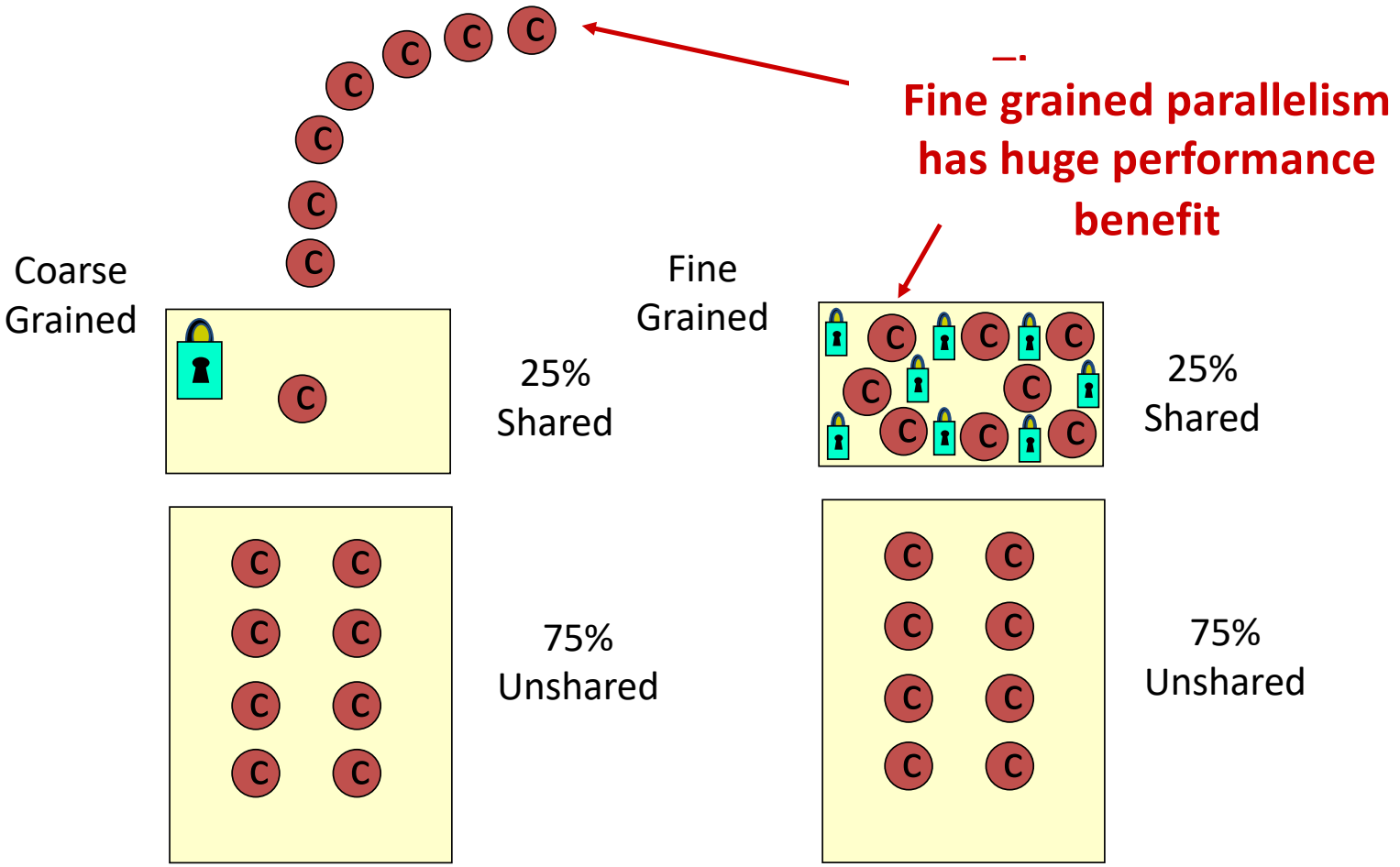
simple but does not scale

Amdahl's Law:
Speedup = 1/(ParallelPart/N + SequentialPart)

Pay for N = 128 cores
SequentialPart = 25%

As num cores grows the effect of 25% becomes more acute
2.3/4, 2.9/8, 3.4/16, 3.7/32….

Fine-grained Locking?



Fine-grained Locking?
easier said than done

- Fine grained locking is **hard to get right:**
  - deadlocks, livelocks, priority inversions:
  - complex/undocumented lock acquistion protocols
  - no composability of existing software modules

... and a **verification nightmare:**

- subtle bugs that are extremely hard to reproduce

Lock-based synchronization does not support modular programming

- Synchronize moving an element between lists

```
void move(list l1, list l2, element e)
{ if (l1.remove(e)) l2.insert(e); }
```

- Assume remove/insert acquire a per-list lock
- Consider two threads that execute:

Thread1		*Thread2*
move(list1,list2,e)		move(list2,list1,e')
list1.lock() ➔ OK		list2.lock() ➔ OK
list2.lock() ➔ wait T1		list1.lock() ➔ wait T2

Transactional memory (TM)

```
atomic
        A.withdraw(3)
        B.deposit(3)
end
```

- Same idea as in a ACI~~D~~ database transaction:
  - "Write simple sequential code & wrap **atomic** around it".
  - Hide away synchronization issues from the programmer
    - Programmers say what should be made atomic…
      and not how atomicity should be achieved
  - way simpler to reason about, verify, compose
  - similar performance to fine-grained locking
    - via speculation & possibly hardware support

TM : Brief historic overview

- Original idea dating back to early 90s
  - Herlihy/Moss ISCA 1993 ➔ hardware-based
- Largely neglected until advent of multi-cores (~2003)
- Over the last 15 years:
  - one of the hottest research topics in parallel computing
- Since ~2013:
  - IBM and Intel CPUs ship with hardware support for TM
- Standardization of language supports for C/C++
- Integration in most popular programming languages

How does it work?

- Various implementations are possible:
  - Software (STM):
    - instrumentation of read and write accesses
  - Hardware (HTM):
    - extension of the cache consistency mechanism
  - Hybrid (HyTM)
    - mix of the two worlds that tries to achieve the best of both

STM

- **Many** algorithms proposed over the last decade(s):
  - DSTM,JVSTM,TL,TL2,LSA,TinySTM,SwissTM,TWM,NOREC,AVSTM…

- Key design choices
  - word *vs* object vs field based
  - single-version *vs* multi-version
  - in-place write+undo logs *vs* deferred writes+redo logs
  - lock-based *vs* lock-free
  - commit-time locking *vs* encounter-time locking
  - safety and progress semantics
  - …

Example STM Algorithm : TL2 (Transactional Locking 2)

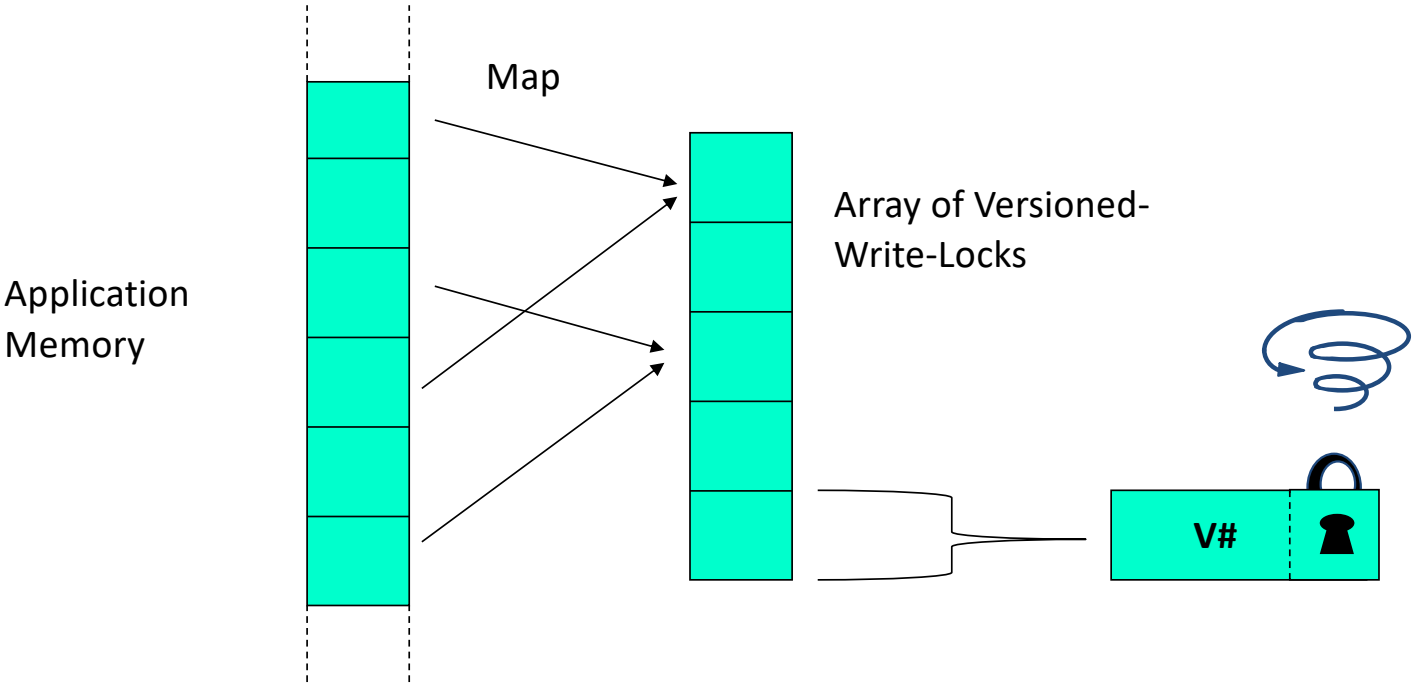
Dave Dice, Ori Shalev, and Nir Shavit.
Transactional locking II. DISC 2006

TL2 overview

- Key design choices
  - **word-** vs object vs field based
  - **single-version** vs multi-version
  - in-place write + undo logs vs **deferred writes + redo logs**
  - **lock-based** vs lock-free
  - **lazy locking** vs eager locking
  - visible vs **invisible reads**
  - progress : **no deadlock**, no livelocks, no abort for RO tx

achieved via an external contention manager
(e.g., exponential back-off of aborted transactions)

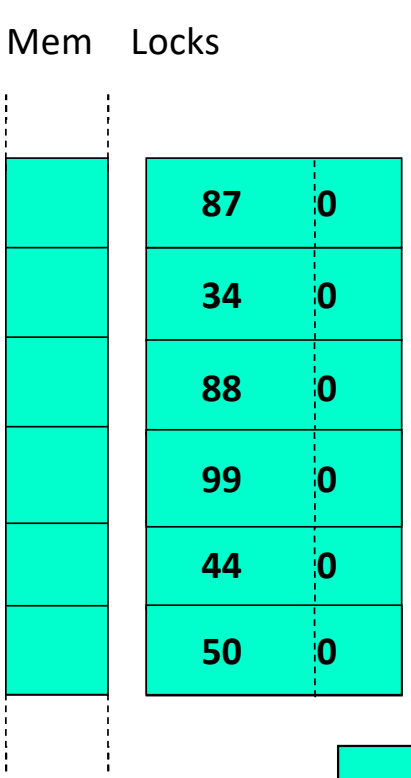
Versioned Locks

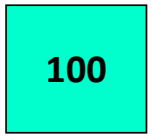


PS = Lock per Stripe (separate array of locks)

PO = Lock per Object (embedded in object)

Read-only Transactions





VClock

On Tx begin

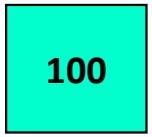
RV ← VClock

On Read

read lock, read mem, read lock: check unlocked, unchanged, and v# <= RV

On Commit

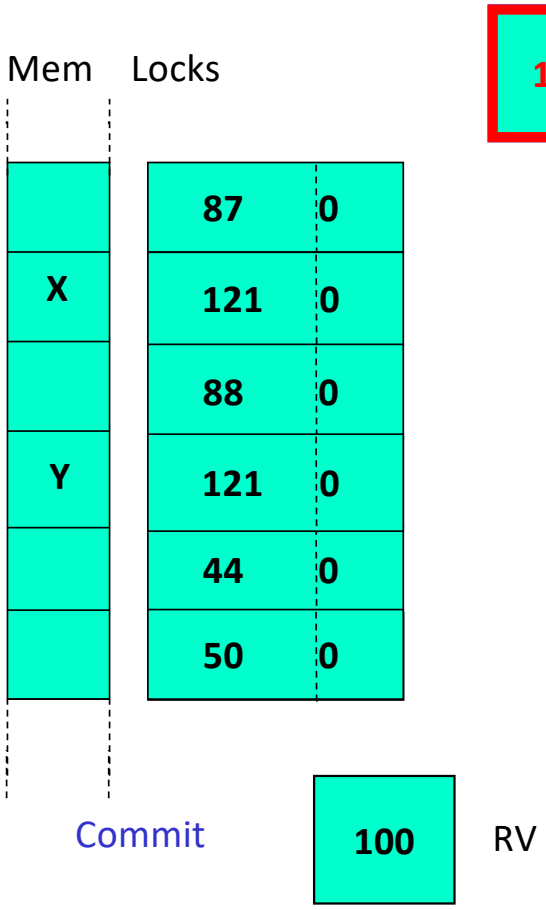
nothing to be done!



RV

Reads from a consistent snapshot of memory.
No need to track and validate read set!

Update transactions



On Tx begin

RV ← VClock

On Read/Write

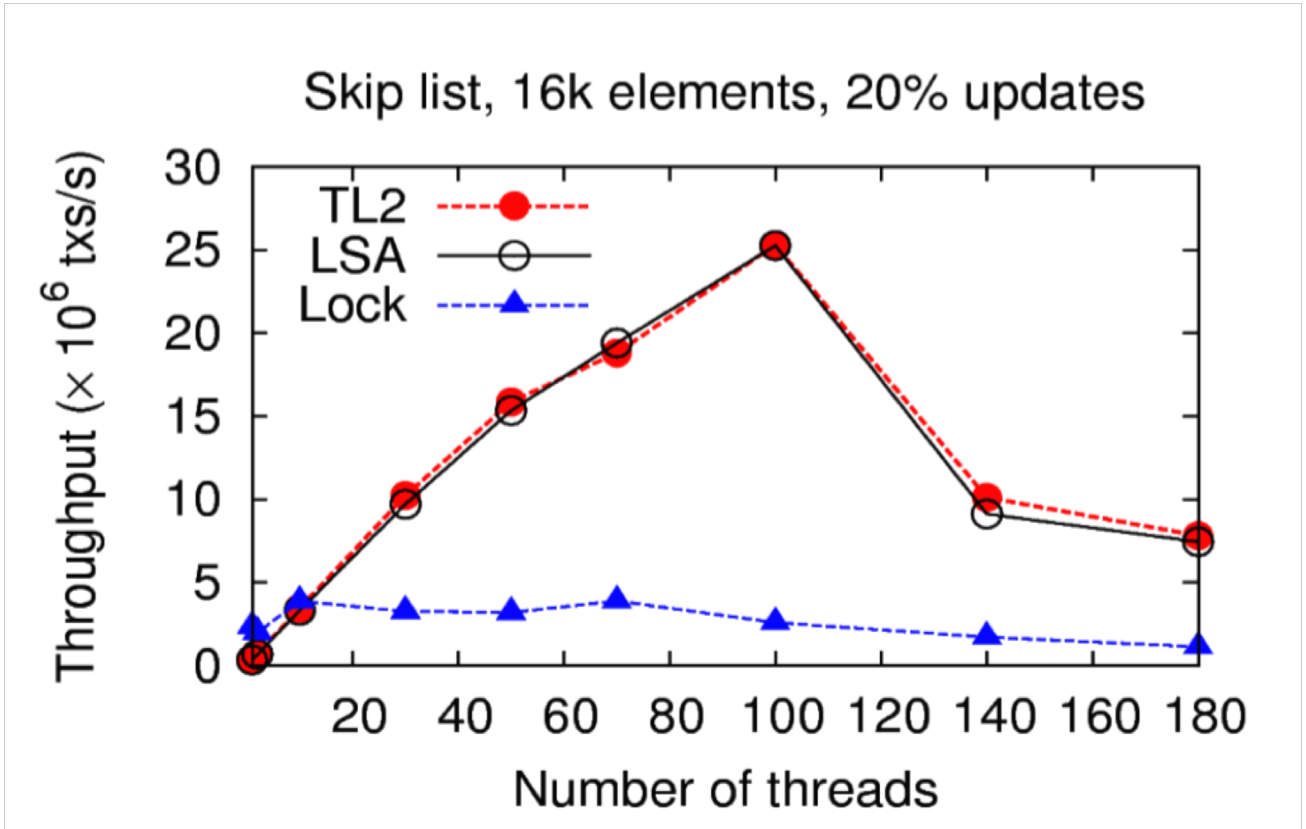
check unlocked and v# <= RV

then add to Read/Write-Set

On Commit

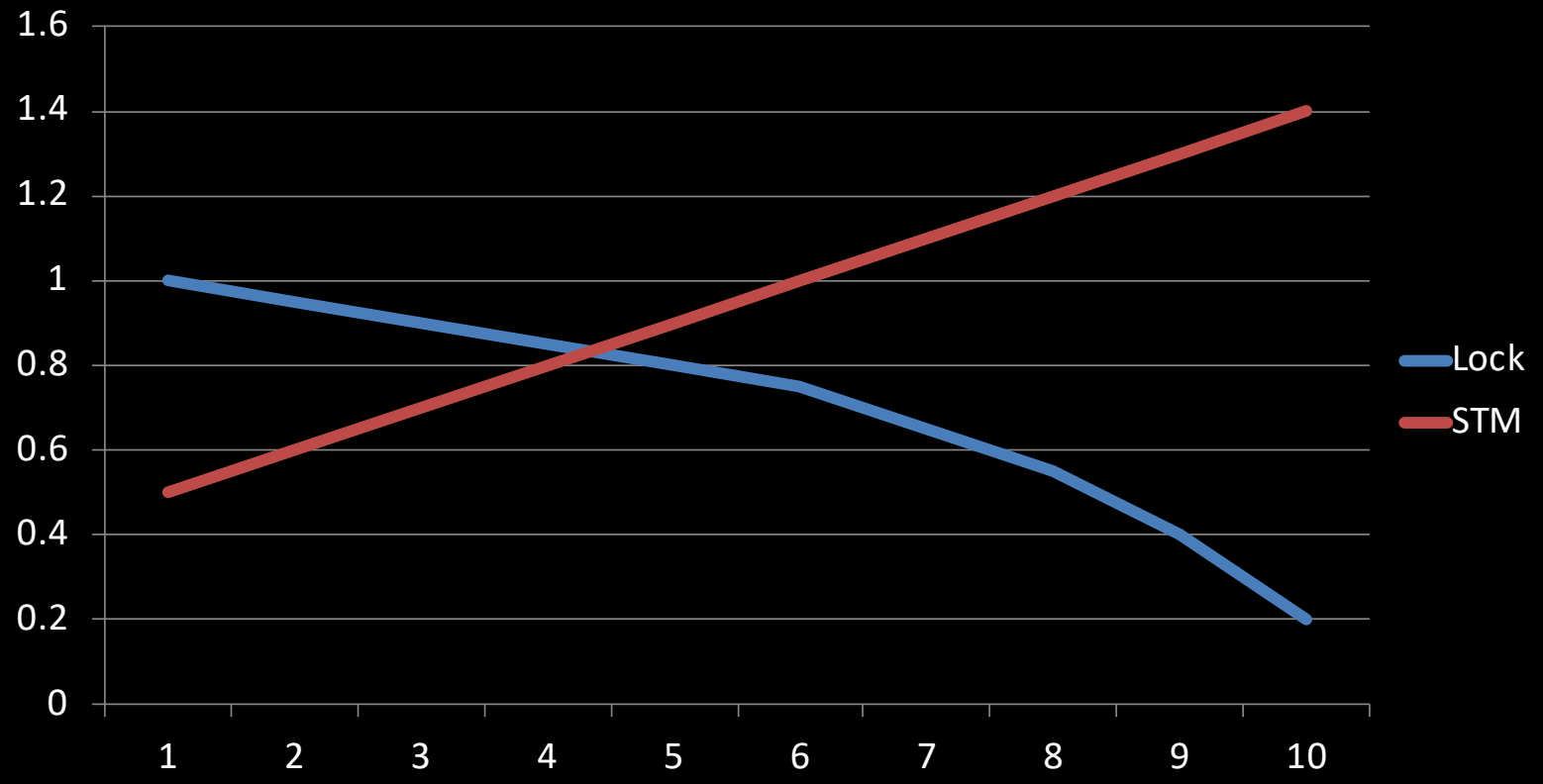
1. Acquire Locks
2. WV = F&I(VClock)
3. Validate each v# <= RV
4. Release locks with v# ← WV

STM Performance: the bright side



(Azul – Vega2 – 2 x 48 cores)

STM Performance: the dark side



Sources of overhead in STMs

- STM scalability is as good if not better than fine-grained locking, but overheads are much higher

- Key sources of overhead:
  - *Instrumented accesses* – constant overhead on every read/write
  - *Readset validation* – linear overhead on read items

Let the hardware do the dirty work → Hardware TM

How does it work?

- Various implementations are possible:
  - Software (STM):
    - instrumentation of read and write accesses
  - Hardware (HTM):
    - extension of the cache consistency mechanism
  - Hybrid (HyTM)
    - mix of the two worlds that tries to achieve the best of both

HTM is now available in several CPUs

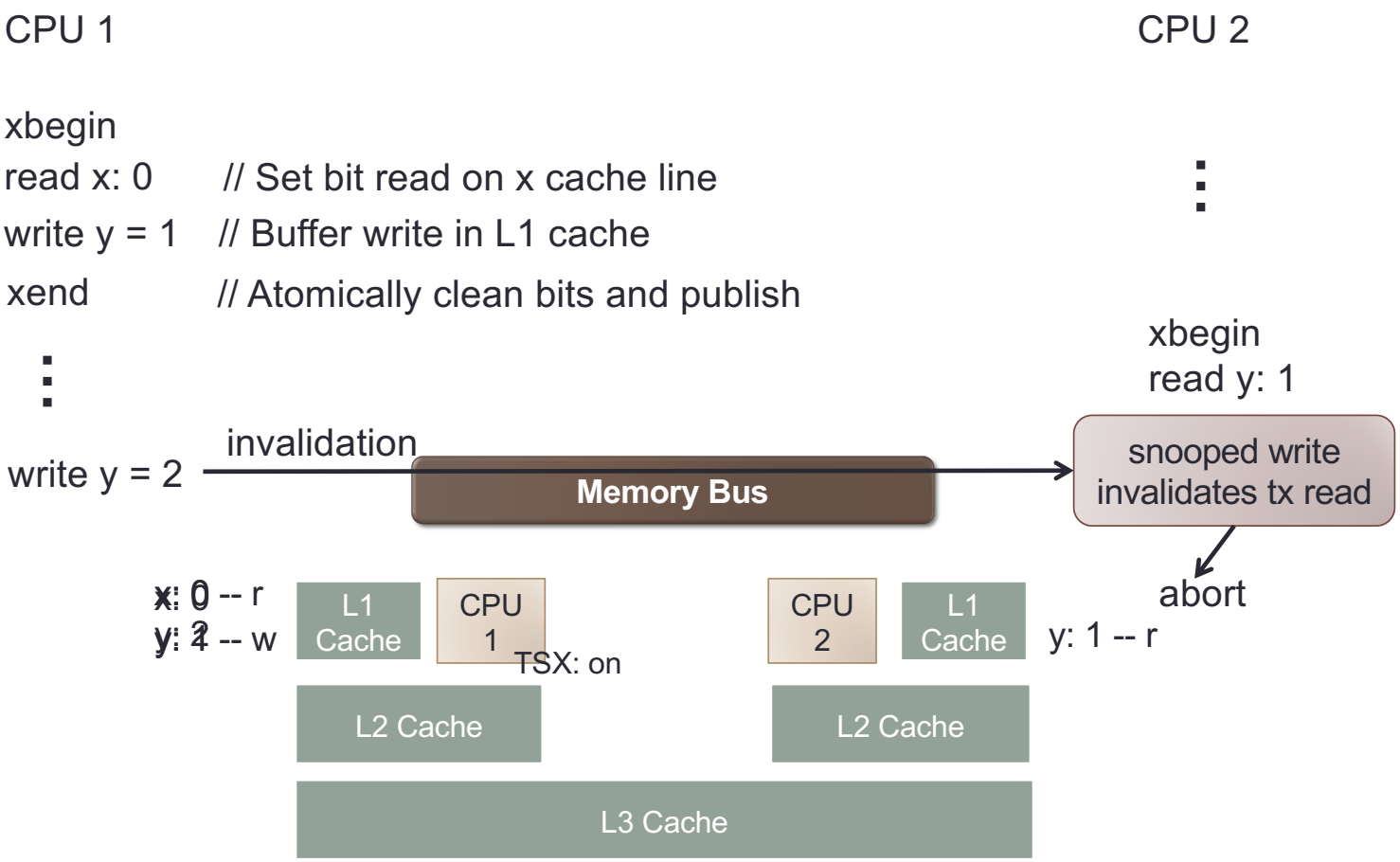
- Intel: most CPUs since Haswell family (~2013)
- IBM: BG/Q, zEC12, Power8

- HTM implementations are NOT born equal…
- Yet, they share two important commonalities:

1. Based on cache coherency protocol

2. Best-effort nature

Overview of Intel's HTM: TSX



HTM's best effort nature

No progress guarantees:

- A transaction may **always** abort

…due to a number of reasons:

- Capacity of caches
- Forbidden instructions (e.g., system calls)
- Faults and signals
- Contending transactions, aborting each other

HTM alone is not enough

Fallback plan!

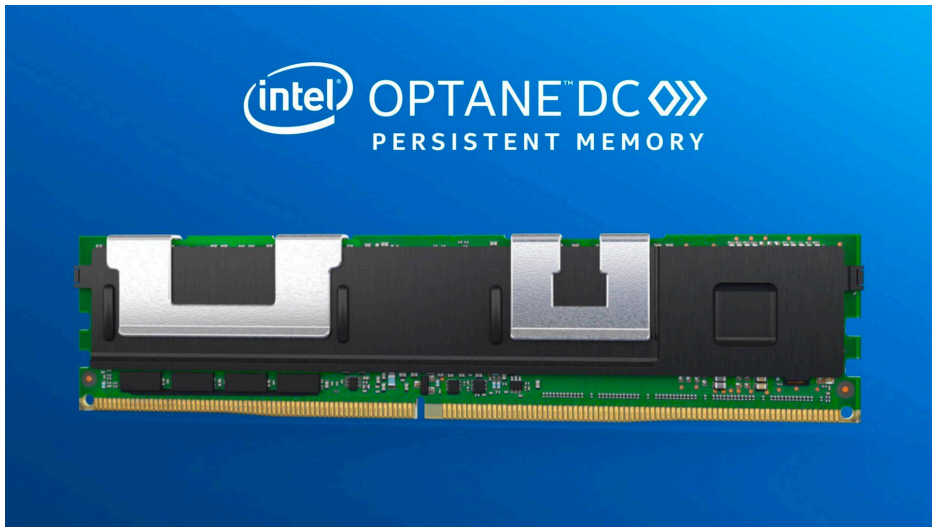
- After a few attempts using HTM, the tx is executed using software synchronization:
  - Single global lock (***current*** standard approach)
    - PRO: success guarantee, support for not-undoable ops.
    - CON: no parallelism (extermination of concurrent hw tx)
  - STM → **Hybrid TM**
    - PRO: fallback path does support parallelism
    - CON: large synchronization overheads btw HTM and STM

Roadmap

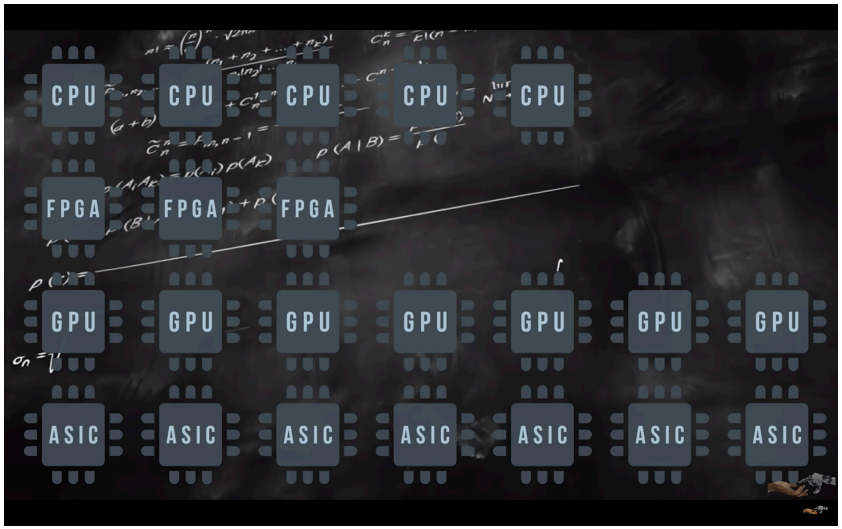
- About me
- About IST & INESC-ID
- Introduction to Transactional Memory
- Transactional Memory & emerging HW technologies:
  - Non-Volatile Memory [IPDPS'18/JPDC'19]
  - Heterogeneous Computing [PACT'19]

Two emerging hardware technologies

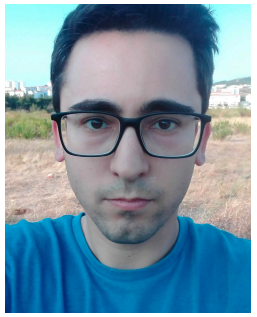
- Non-Volatile Memory



- Heterogeneous architectures



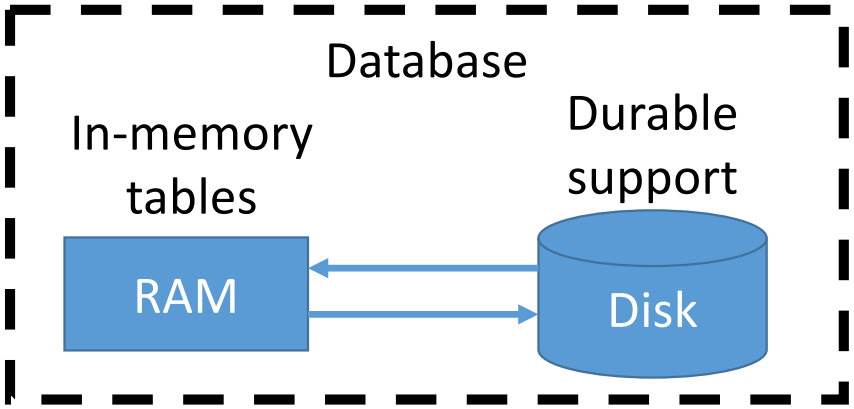
Roadmap

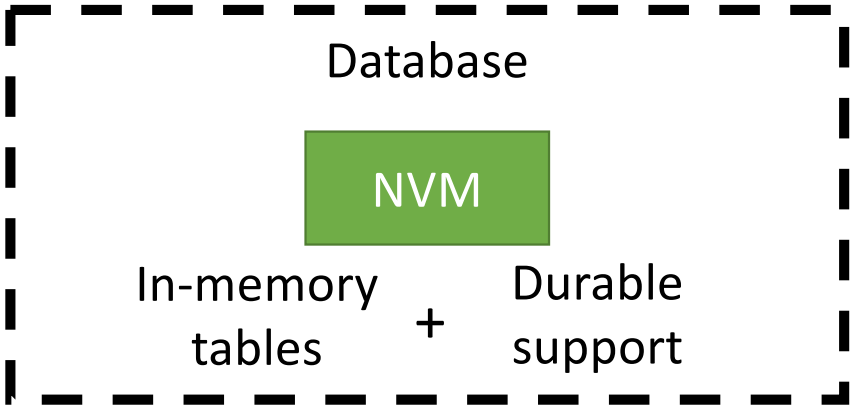


- About me
- About IST & INESC-ID
- Introduction to Transactional Memory
- Transactional Memory & emerging HW technologies:
  - Non-Volatile Memory [IPDPS'18/JPDC'19]
  - Heterogeneous Computing [PACT'19]

Non-Volatile Memory (NVM)

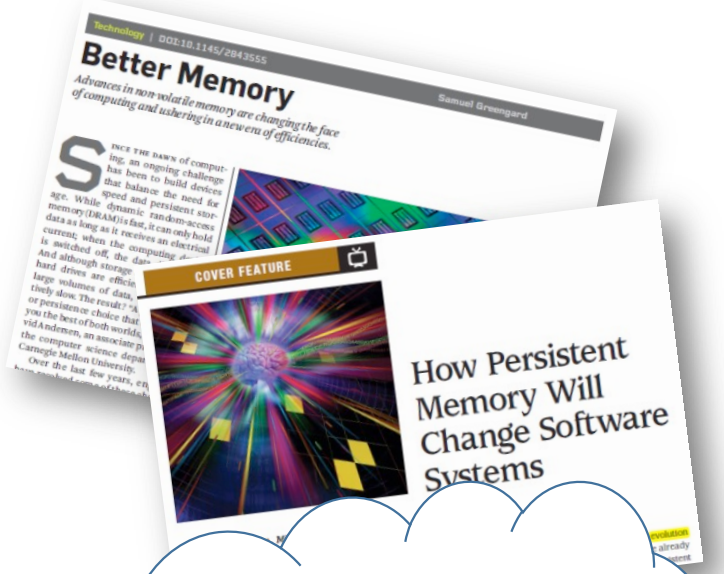
- Fast byte-addressable storage
- Higher density/lower cost per byte when compared with volatile RAM
- Expect writes to be slower than RAM (2x-5x):
- Subject to wear off upon write (technology dependent)





Free durability?

These are exciting times for programmers



Concurrent programming will become easy!

Working memory and persistent store will become the same!

Can we combine both hardware revolutions?

Well, not directly…

Can we combine both hardware revolutions?

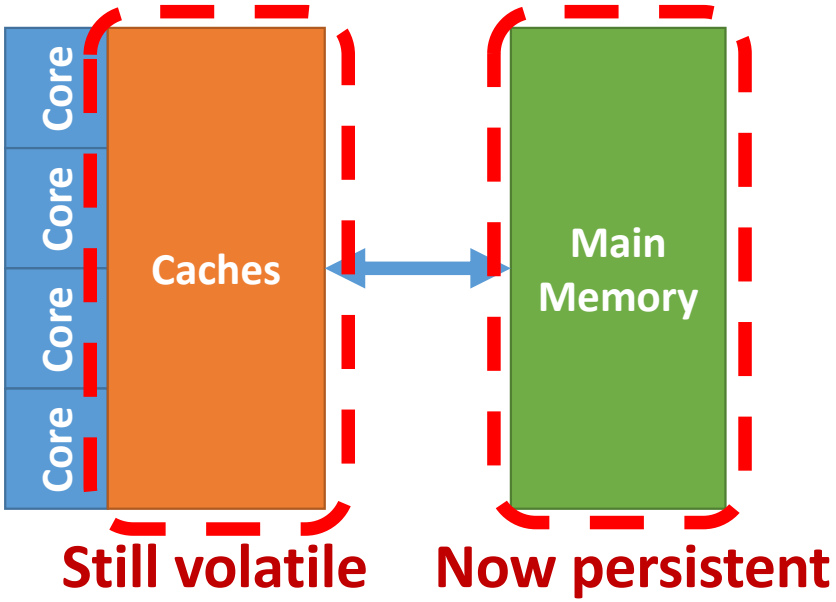
Well, not directly…

NV-HTM can do it on **unmodified hardware**
by leveraging hardware-software co-design…

…while achieving up to 10x better performance
when compared to solutions that
modify the hardware

Non-Volatile Memory: the bad news…

- CPU Caches (most likely) will continue being volatile:
  - What is effectively written into memory?

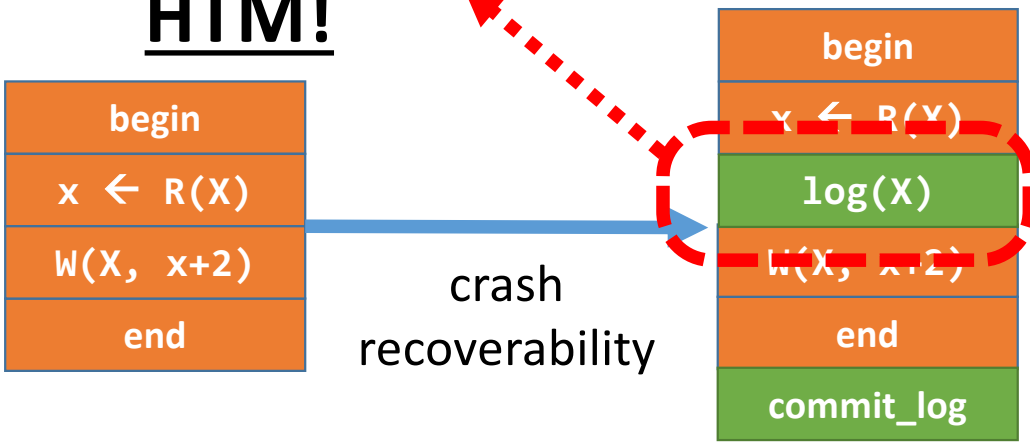


- Applications must explicitly bypass caches:
  - clflush, clflushopt, clwb
  - Else:
    - writes are not guaranteed to enter PM
    - writes may be reordered
- What about applications that require atomic access/transactions to memory regions?

Integrating NVM and Software-based TM

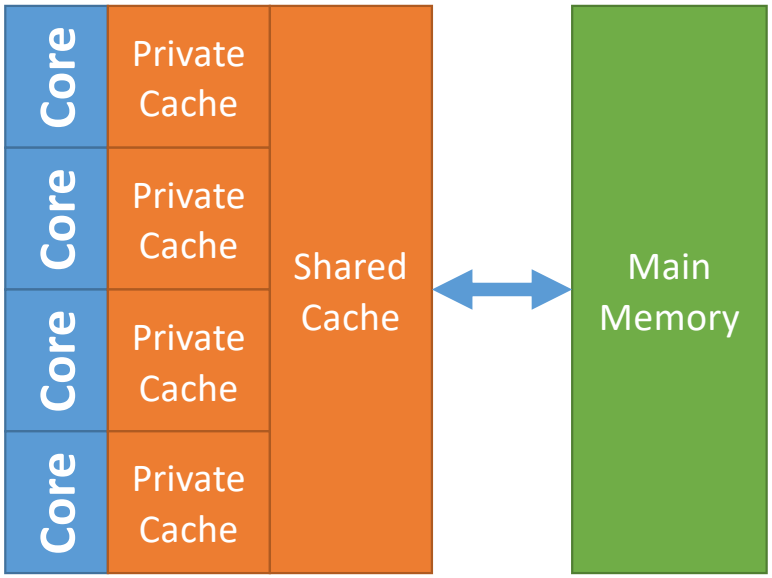
- Durability of transactions regulated via software concurrency is well-understood: decades of literature in DBMS area!
- Example based on a recent PM-oriented software-based approach [**ASPLOS'16**]:
  - Upon write
    1. Lock the value
    2. Log (flush) the old value
    3. Do the write
  - Upon commit
    1. Flush write-set
    2. Add commit marker
    3. Unlock values
    4. Destroy log

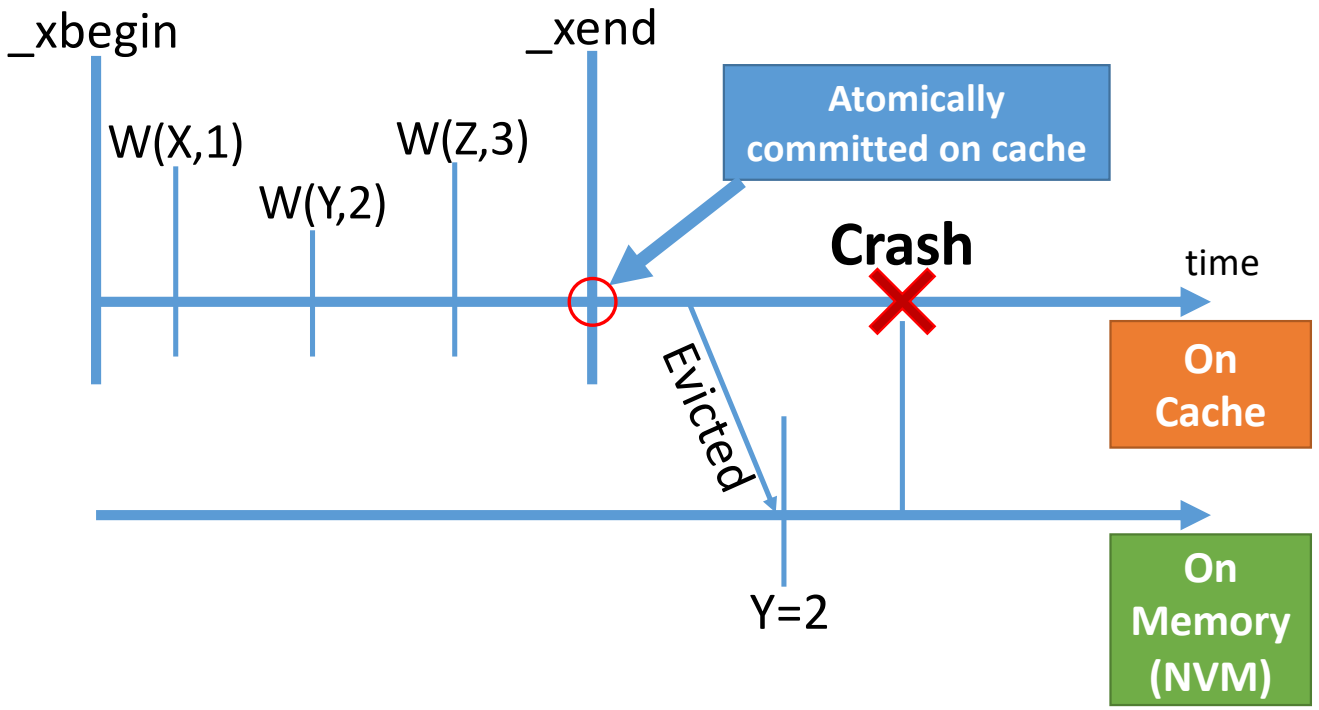
Unfortunately not possible with HTM!



Hardware Transactional Memory (HTM)

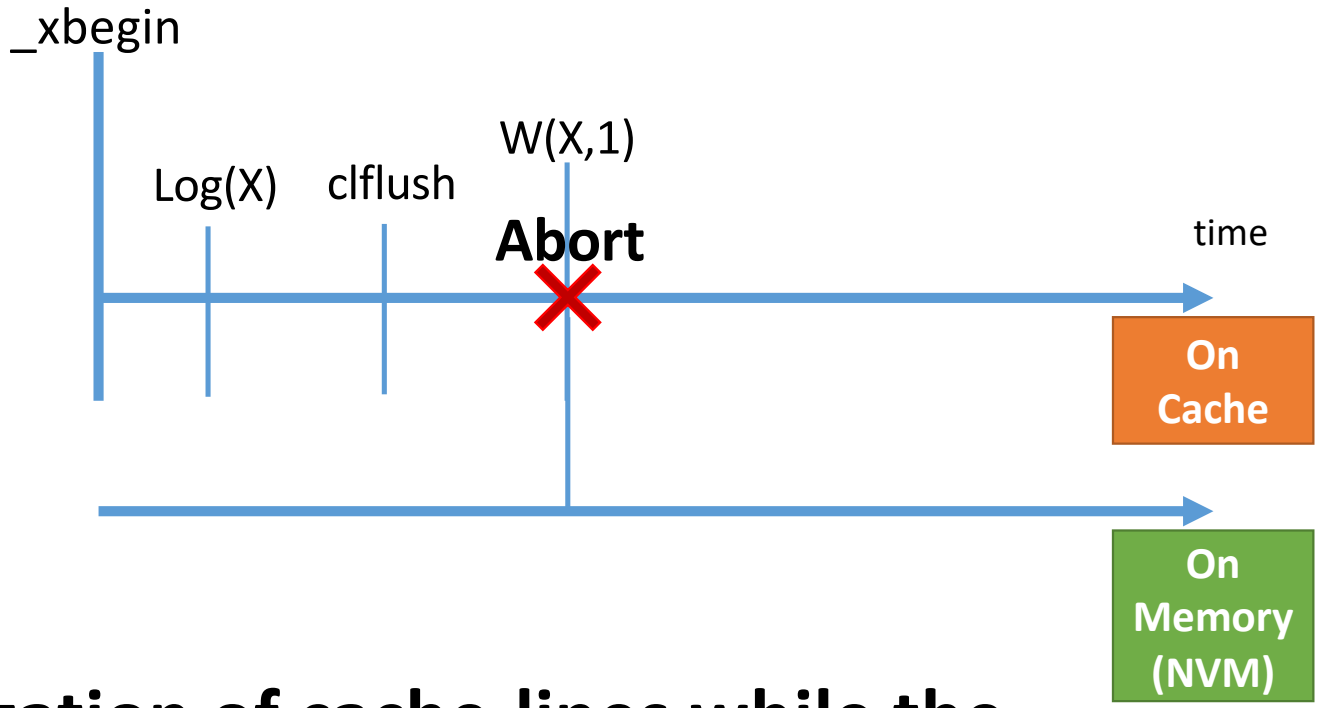
Concurrency is built on on cache coherency protocols [**ISCA'93**]





Example of a story of a non-durable (and non-atomic after recovery) transaction!

Hardware Transactional Memory (HTM)



Externalization of cache-lines while the transactions is running is not allowed!

Related Work

STM-based solutions[**ASPLOS'11**, **ASPLOS'16**]

- build on DBMS literature on logging schemes:
  - adapted & optimized for PM
  - flexible design
  - boilerplate on each load and store

Drawbacks:

- **STM incurs much larger overhead than HTM!**
- **Do not work with HTM**

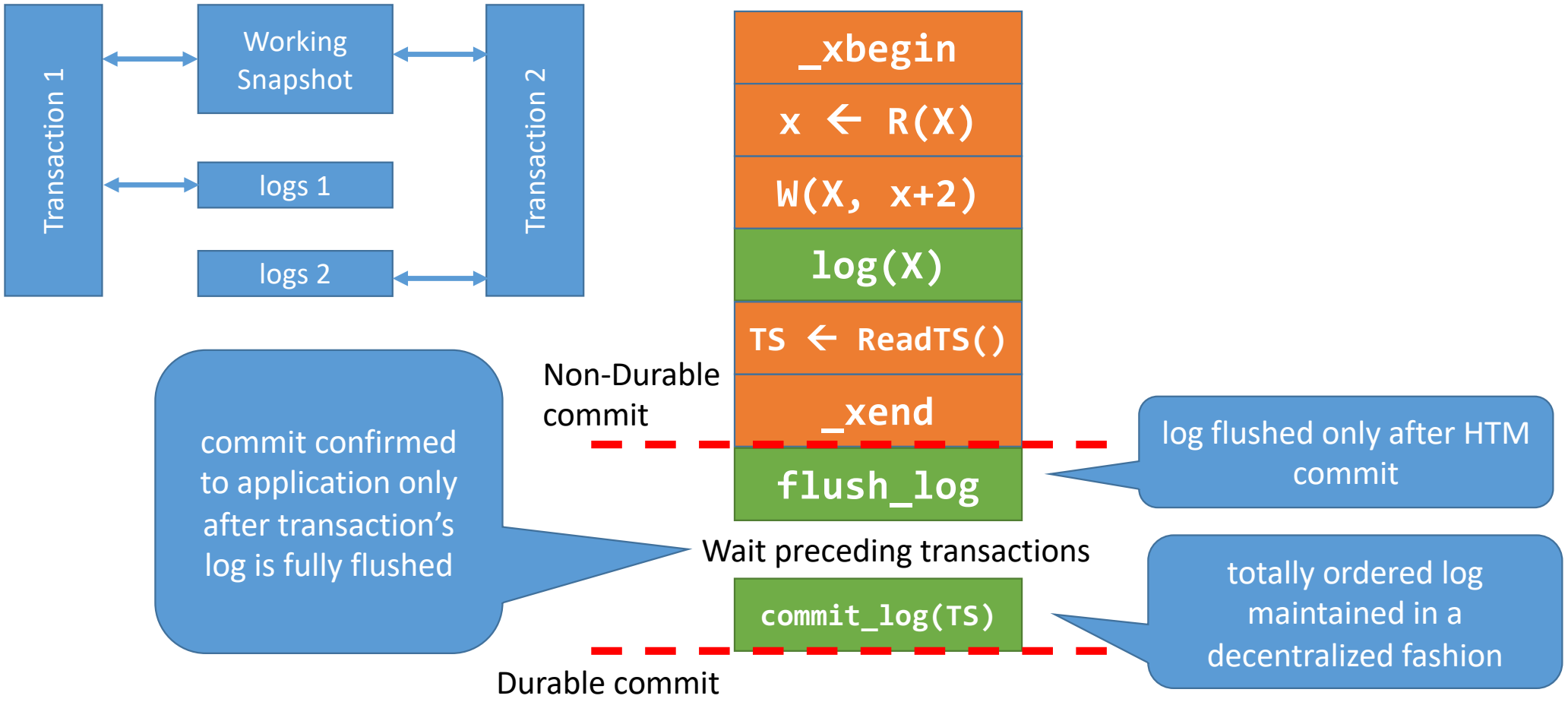
HTM-based solutions [**DISC'15**, **CAL'15**]

- Rely on modified HTM implementation
- PHTM [**DISC'15**]:
  - Flush cache-lines within transaction
  - Order writes to logs via additional locks
  - Commit flushes a commit marker

Drawbacks:

- **Incompatible with commodity HTM**
- **Additional locks reduce concurrency and available capacity**

NV-HTM: Transaction logging – 1/3



NV-HTM: Transaction logging – 1/3

Pros:

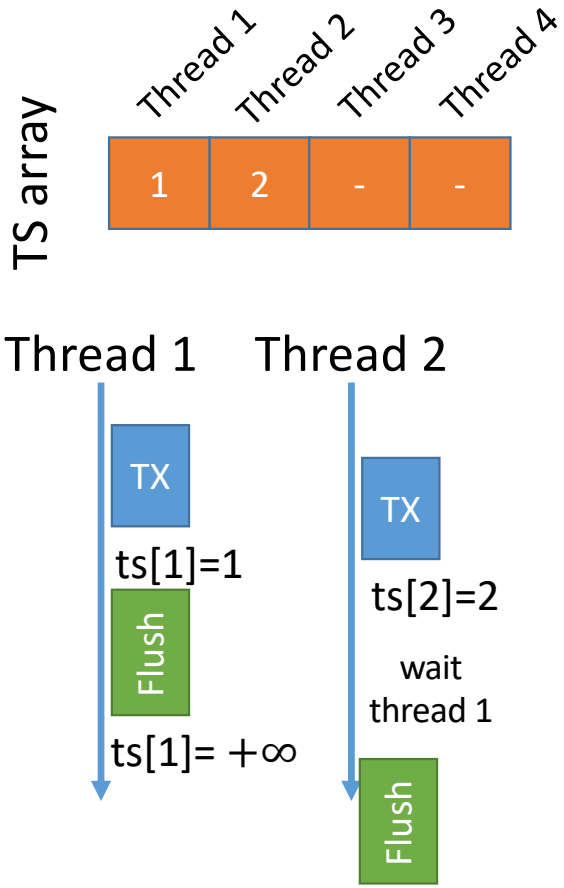
✓ Ensure interoperability with existing HTM systems!

✓ Avoid contention hot-spots to maximize scalability

Challenge:

- If a transaction is durable, all transactions it depends upon also are:
  - novel synchronization scheme based on physical clock
- Upon crash:
  - no guarantee that updates of non-durably committed transaction hit PM
  - possible corrupted snapshot upon failure!

Ordering Transactions



- Each thread advertises current TS in TS array
- While the transaction is not running, the advertised TS is $+\infty$, i.e., ts[i] = $+\infty$
- Before the transaction starts, a timestamp is taken → ts[i] = ReadTS()
- After non-durable commit:
  1. Advertise the TS taken inside the transaction
  2. Flush all log entries but the commit marker
  3. Wait while there is an older transaction (smaller TS)
  4. Flush commit marker

NV-HTM: Transaction logging – 1/3

Pros:

✓ Ensure interoperability with existing HTM systems!

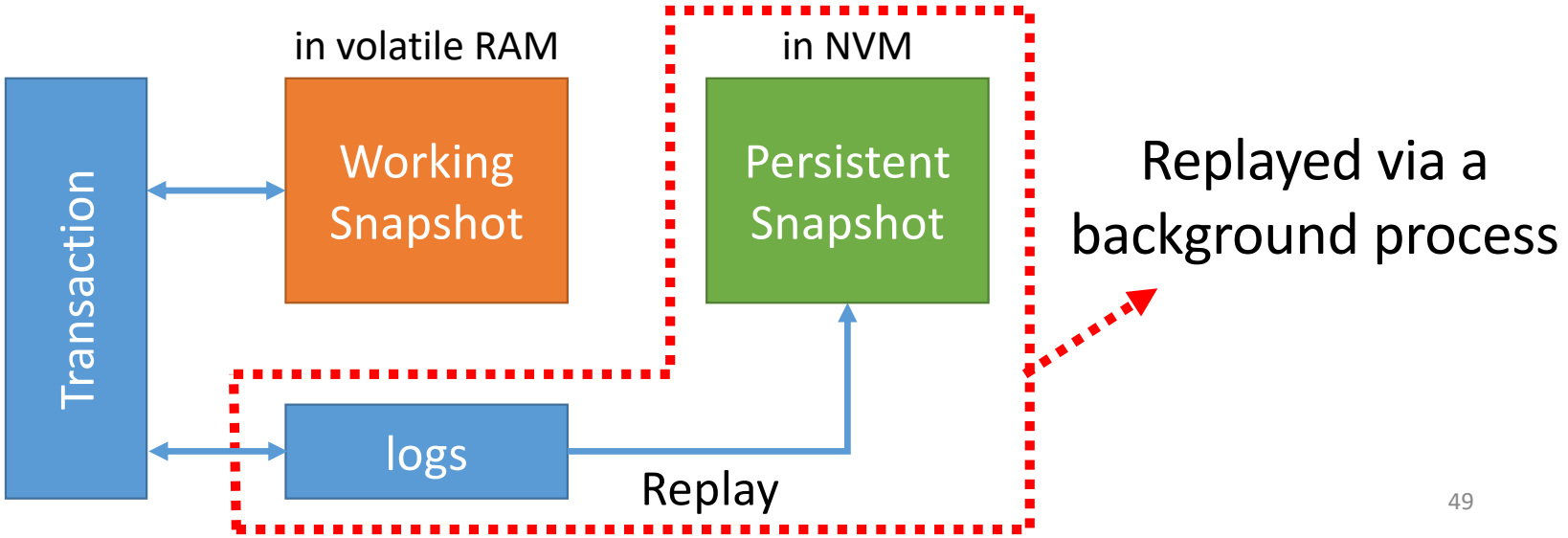
✓ Avoid contention hot-spots to maximize scalability

Challenge:

- If a transaction is durable, all transactions it depends upon also are:
  - novel synchronization scheme based on physical clock
- Upon crash:
  - no guarantee that updates of non-durably committed transaction hit PM
  - possible corrupted snapshot upon failure!

NV-HTM: Working and Persistent Snapshots – 2/3

- Application writes in a (volatile) **working snapshot**
- Logged writes are replayed asynchronously to produce a consistent **persistent snapshot** on PM
  - via background checkpoint process



NV-HTM: Working and Persistent Snapshots – 2/3

Pros:

✓ Writes to PM are 2x-5x slower than on volatile RAM!

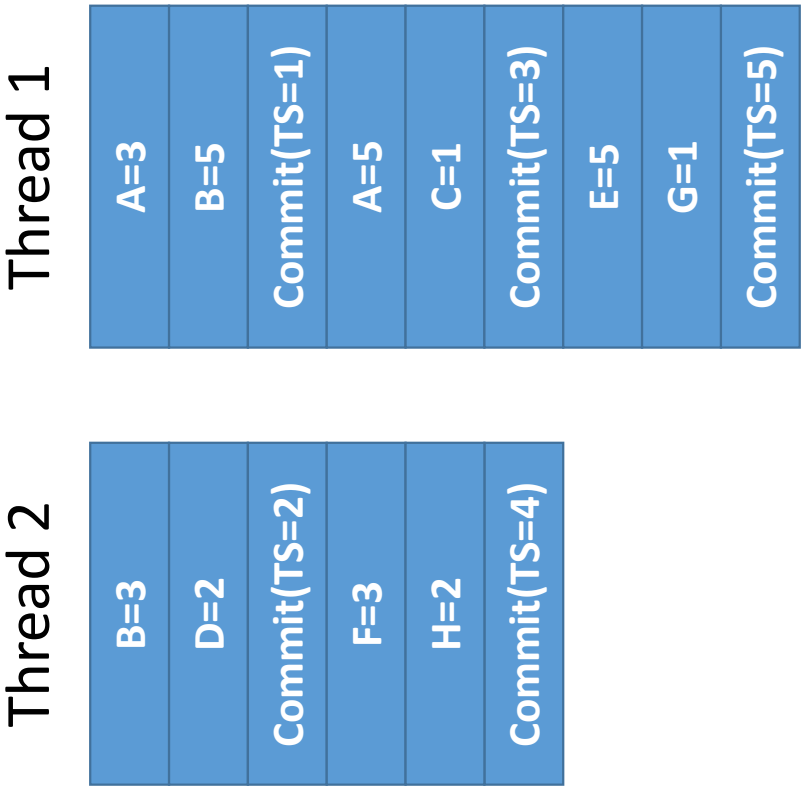
✓ Provides opportunity to filter redundant (duplicate) writes in the log

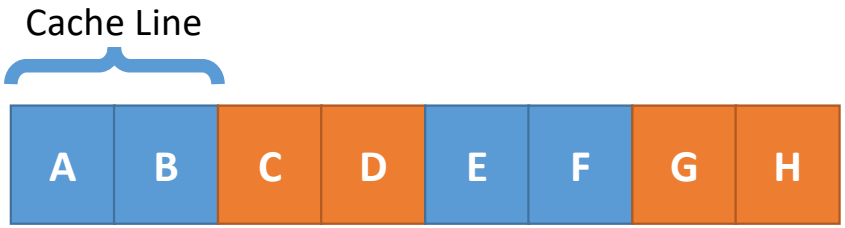
- less writes/flushes === longer life for PM!

Challenge:

- Memory efficiency: avoid maintaining 2 full copies of application's memory

Log filtering





The Checkpoint Process may follow different policies to flush the logs:

- Naïve approach: flush every log entry:
  - **Forward No Filtering (FNF)**
- Replay all writes but flush each updated cache line only once:
  - **Forward Flush Filtering (FFF)**
- Scan logs backwards and write/flush only most recent update:
  - **Backward Filtering Checkpointing (BFC)**

NV-HTM: Working and Persistent Snapshots – 2/3

Pros:

✓ Writes to PM are 2x-5x slower than on volatile RAM!

✓ Provides opportunity to filter redundant (duplicate) writes in the log

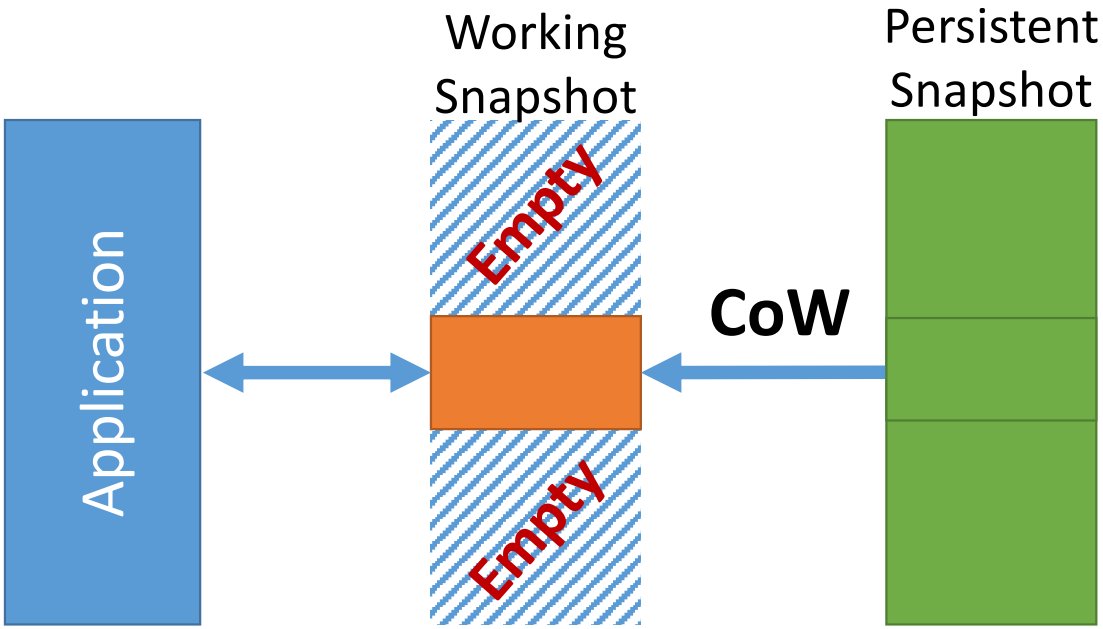
- less writes/flushes === longer life for PM!

Challenge:

- Memory efficiency: avoid maintaining 2 full copies of application's memory

Memory efficiency via CoW – 3/3

- Efficient management of working and persistent snapshot via OS/HW-assisted Copy-on-Write mechanism:
  - duplicate on volatile memory only regions actually modified by application



Recovering from a crash

1. Checkpoint Process replays any pending logged transaction
   - Updated persistent snapshot

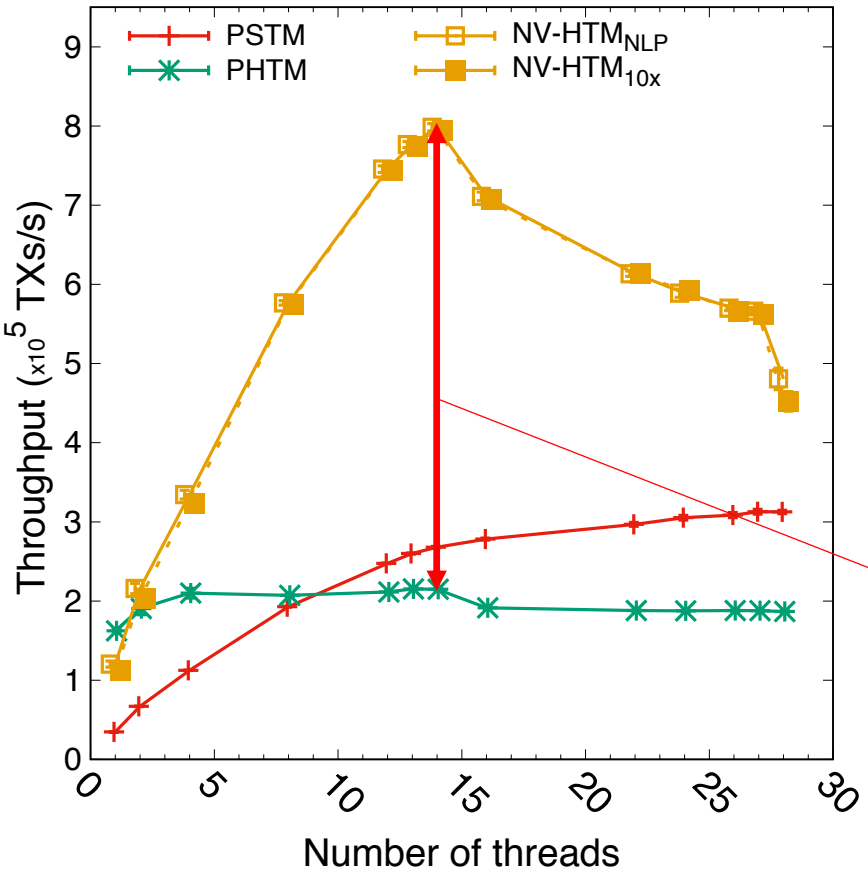
2. Fork the Checkpoint Process:
   - Checkpoint Process mmaps the Persistent Snapshot in shared mode

3. Worker Process mmaps the Persistent Snapshot in private mode
   - Obtains a volatile copy of the Persistent Snapshot (the Working Snapshot)
   - OS ensures Copy-on-Write

Experimental evaluation

- System configuration:
  - 14C/28T TSX enabled Intel Xeon Processor (E5-2648L v4), 22MB L3 cache
  - 32 GB RAM
  - Emulate write to PM latency by spinning for 500ns
- Synthetic Benchmark: Bank
- STAMP Benchmark Suit [**IISWC'08**]
- Baselines:
  - PHTM [**DISC'15**]
  - PSTM [**ASPLOS'11**]

STAMP benchmarks



- Comparison for Kmeans (High contention)
- NV-HTM$_{NLP}$: enough capacity for all writes
- NV-HTM$_{10x}$: logs are 1/10 of all writes
  - Checkpoint Manager has minimal impact in throughout

Up to ~4x greater throughput than PHTM

STAMP benchmarks

	Vacation (high)		Kmeans (high)		Yada	
	writes	flushes	writes	flushes	writes	flushes
NV-HTM$_{NLP}$	12.62	4.601	27.00	8.249	18.36	6.016
NV-HTM$_{10\times}$	13.06	4.798	27.03	8.253	25.55	10.10
PHTM	55.01	10.84	45.99	5.000	77.44	14.91
PSTM	174.4	98.08	198.1	125.0	68.69	45.61

only ~13% extra
less 2.72x writes
less 6.72x writes
writes on average
than PHTM in
than PSTM in
filtering approach
average
average

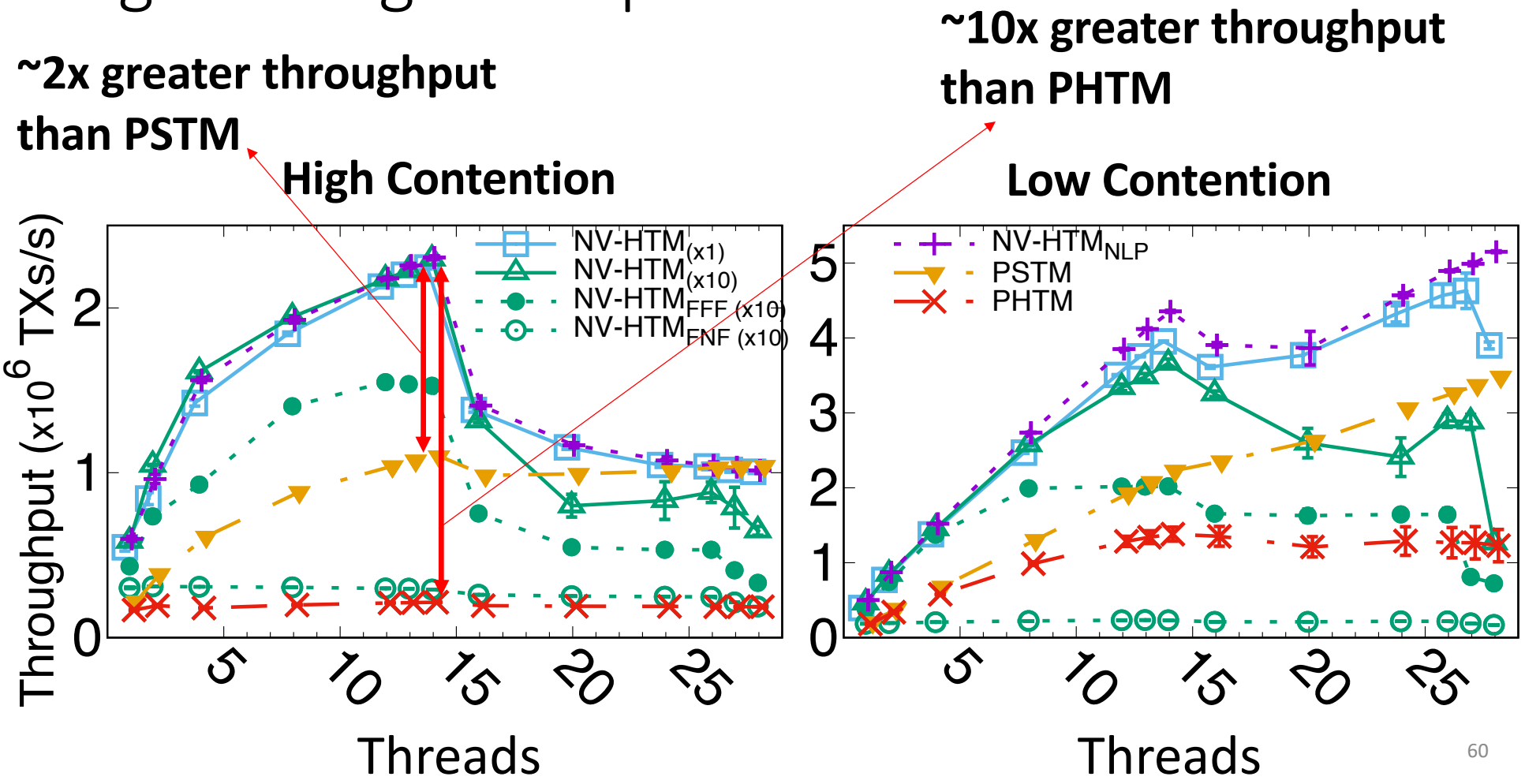
Average Writes and Flushes per transaction

- In average, NV-HTM$_{x10}$ produces 2.72x less writes than PHTM and 6.72x less than PSTM, while only producing 13% more writes than NV-HTM$_{NLP}$

Log filtering - Comparison

- Solutions:
  - NV-HTM$_{NLP}$: very large log → Checkpoint Process never awakes
  - NV-HTM$_{x1}$: Log size is comparable (~85%) to the amount of writes
  - NV-HTM$_{x10}$: Log size is 1/10 the number of writes
  - NV-HTM$_{FFF}$: Flush Filtering
  - NV-HTM$_{FNF}$: No Filtering
- Bank: High contention workload vs Low contention workload
  - Different amount of writes
  - Vary the pressure on the Checkpoint Process

Log filtering - Comparison



Log filtering - Comparison

	NV-HTM					PHTM	PSTM
	NLP	(85%)	$(10\times)$	$(10\times)_{FFF}$	$(10\times)_{FNF}$		
Writes	4.5	4.5	4.55	8.1	8.1	9.1	35.2
Flushes	1.8	1.8	1.83	1.83	6.4	6.3	21.2

Average Writes and Flushes per transaction

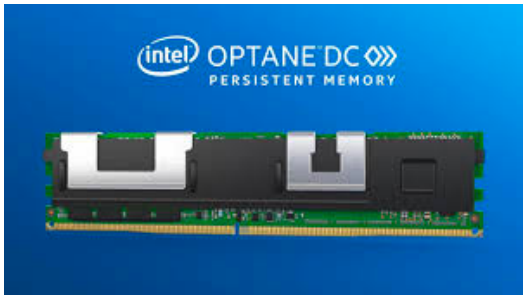
- NV-HTM$_{x10}$ produces 11.6x less flushes than PSTM

Summing up

- NV-HTM efficiently combines HTM and NVM
  - Reduced overheads within Hardware Transactions
    - Worker threads only need to flush data outside the transaction
  - Aims to reduce the number of writes to NVM
    - Checkpoint Process effectively filters repeated writes/flushes
- Does not requires hardware changes
- Up to x10 better throughput and 11.6x less flushes than state-of-the-art solutions

Ongoing & Future work on TM + NVM

- Intel has finally made NVM commercially available
  - Every previous work was based on simulation…
  - Need to reassess actual performance on realistic system

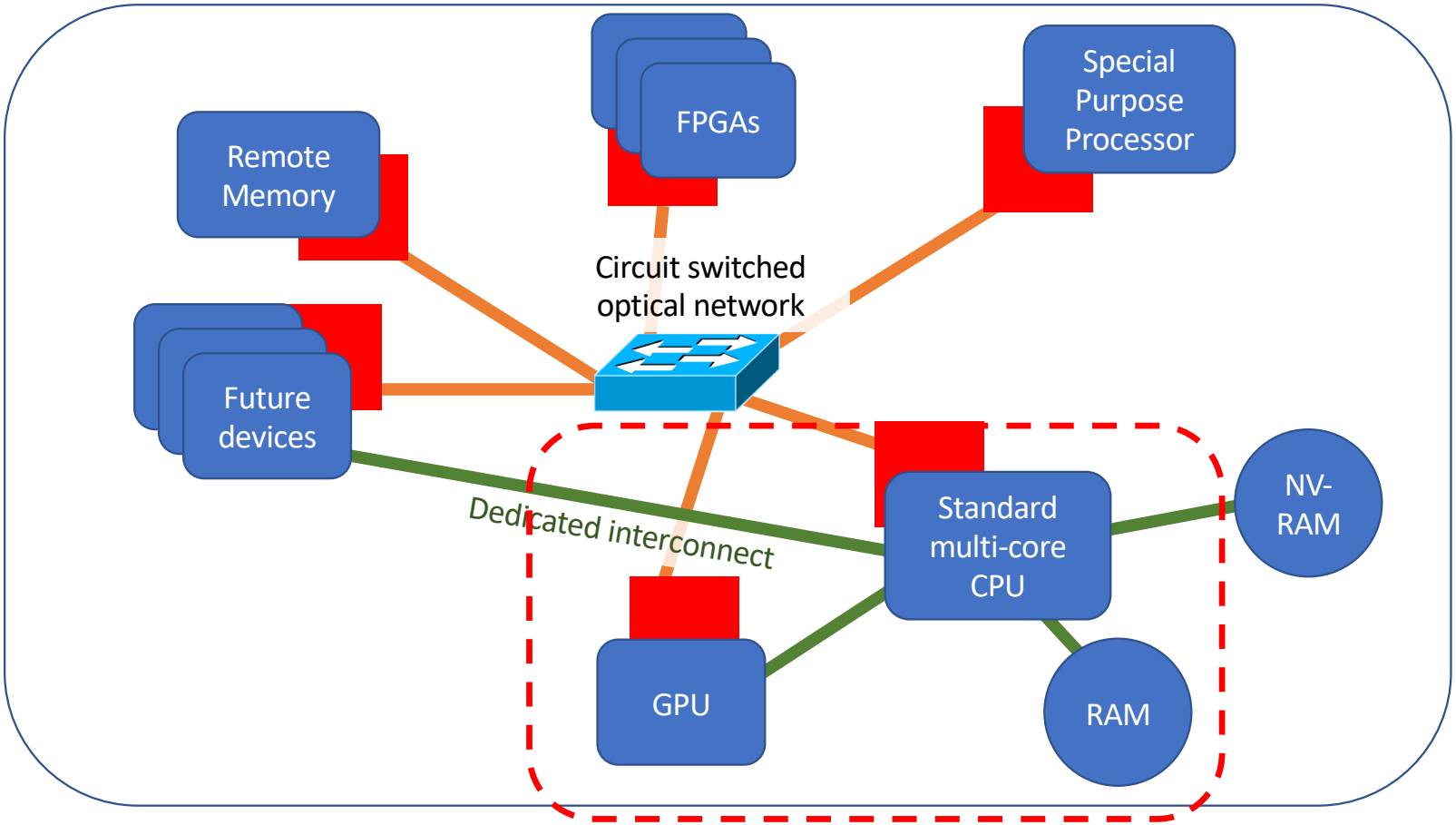


- NV-HTM introduces a serial step in commit phase:
  - Waiting for previous transactions to be durably committed, before a new transaction can be durably committed (flush commit marker to NVM)
    - Latency for flushing commit marker is on critical path of execution
    - Can limit throughput especially if NVM latency is high
  - Ongoing work on how to bypass this limitation

Roadmap

- About me
- About IST & INESC-ID
- Introduction to Transactional Memory
- Transactional Memory & emerging HW technologies:
  - Non-Volatile Memory [IPDPS'18/JPDC'19]
  - Heterogeneous Computing [PACT'19]

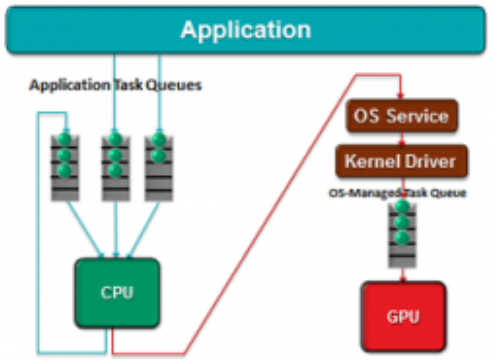
Post-Moore Architectures: heterogeneity

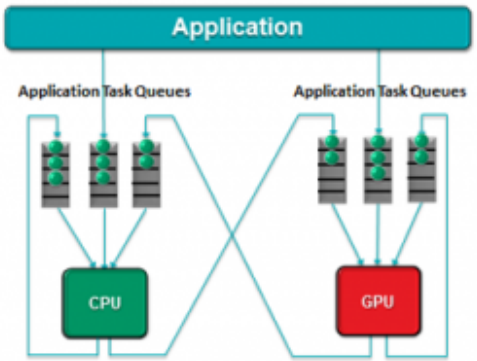


Application development trends in the heterogeneous computing era

The old way:

- CPU serves as master
- Bulk work done on hw accelerators



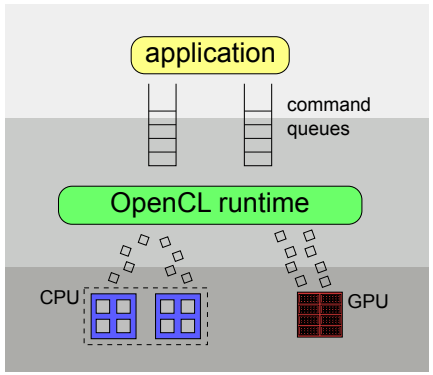


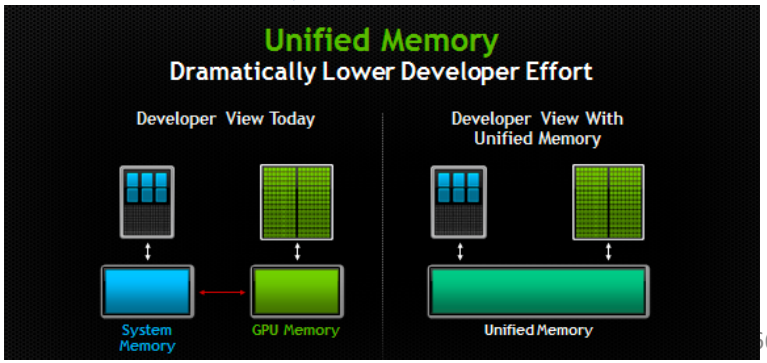
Recent trend:

- Cooperative approach
- CPUs are better fit for, e.g.,:
  - Latency sensitive appl.
  - Fine grained tasks

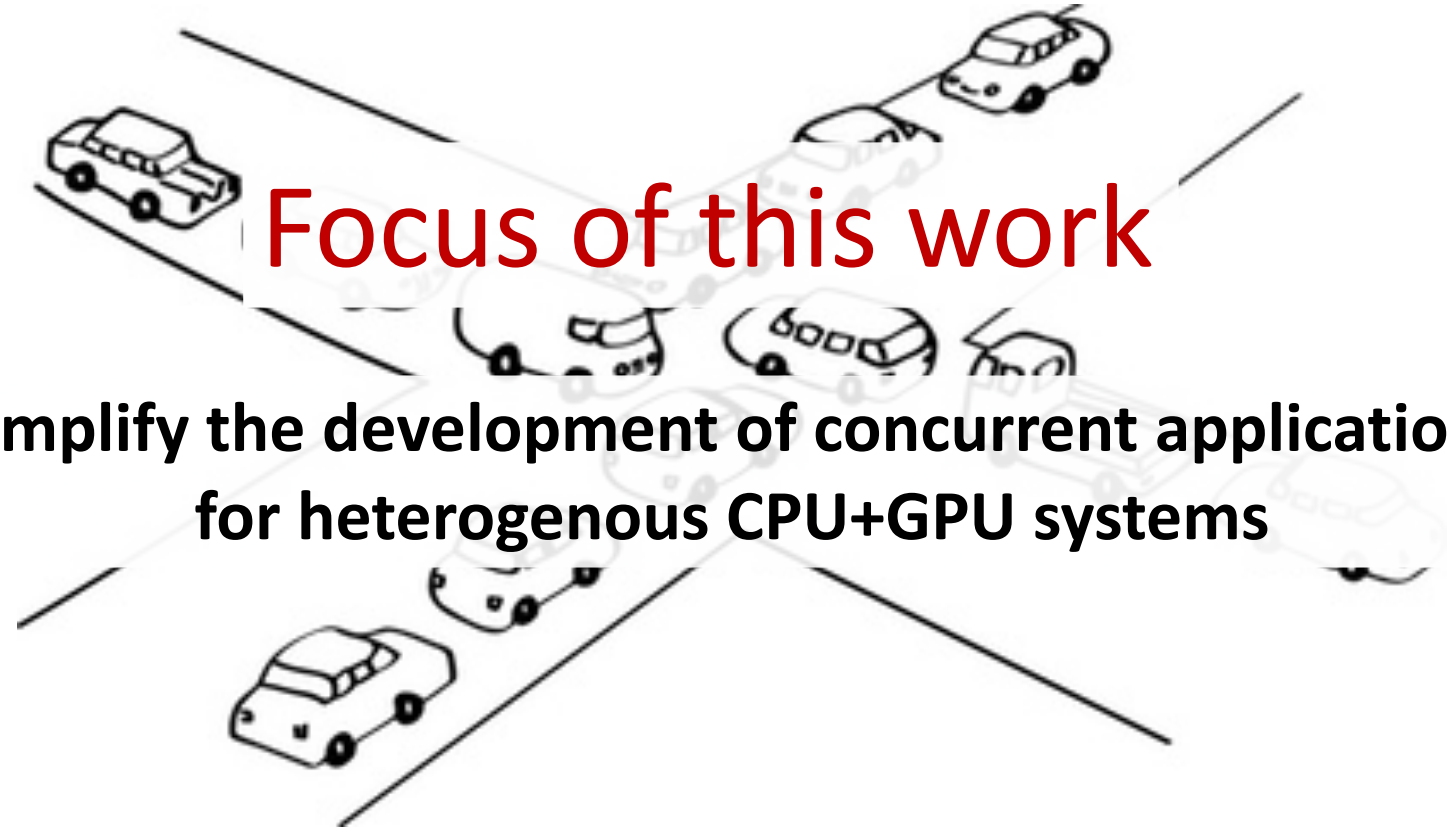
New abstractions to simplify application development both at the hw and sw level

Focus of this work





Many applications are inherently concurrent



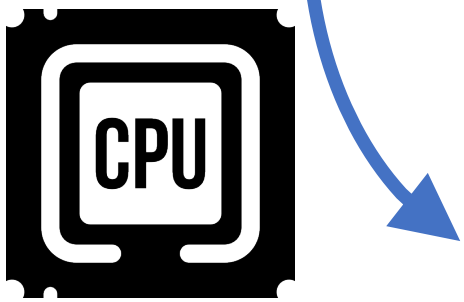
Focus of this work

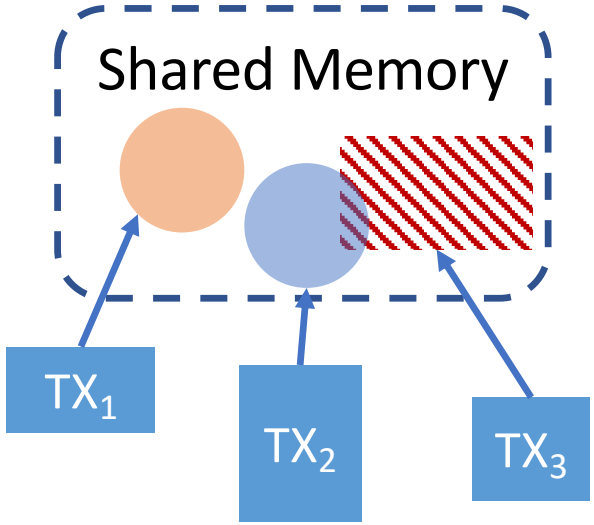
Simplify the development of concurrent applications for heterogenous CPU+GPU systems

Transactional Memory

CPU TM

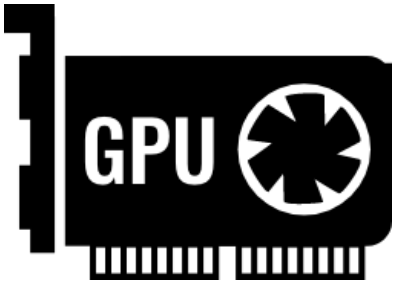
- Mature research
- Widely available in:
  - **Software**
  - **Hardware**
  - **combinations thereof**





GPU TM

- More recent
- Adapted for GPUs
  - Highly parallel architecture
  - Threads execute lockstep



HeTM

Transactional Memory
for CPU+GPU systems

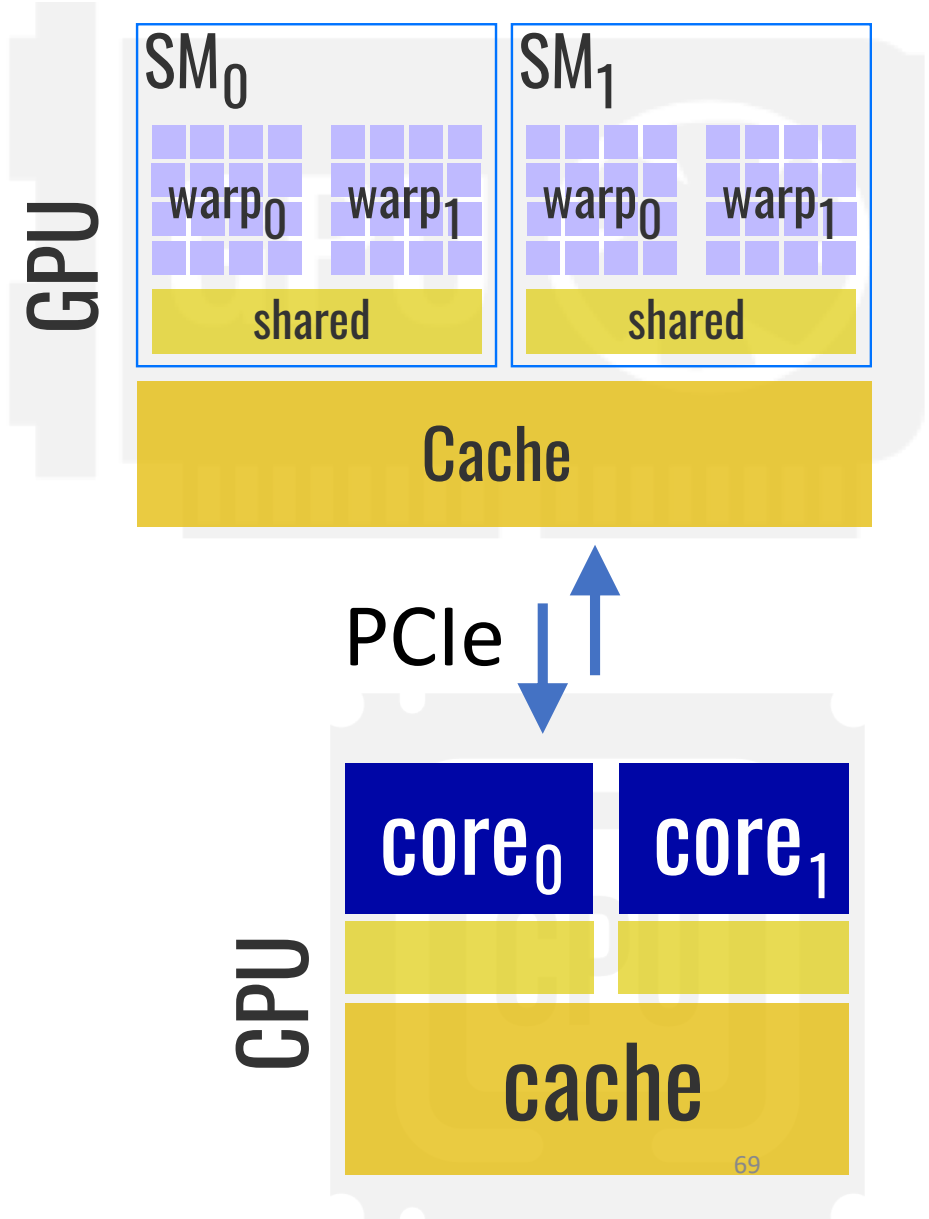
Challenges

Existing TM implementations rely on fast intra-device communication

Serial inter-device communication makes fine-grained synchronization difficult

Need to revisit the TM abstraction and consistency criteria

Build a system upon this new abstraction



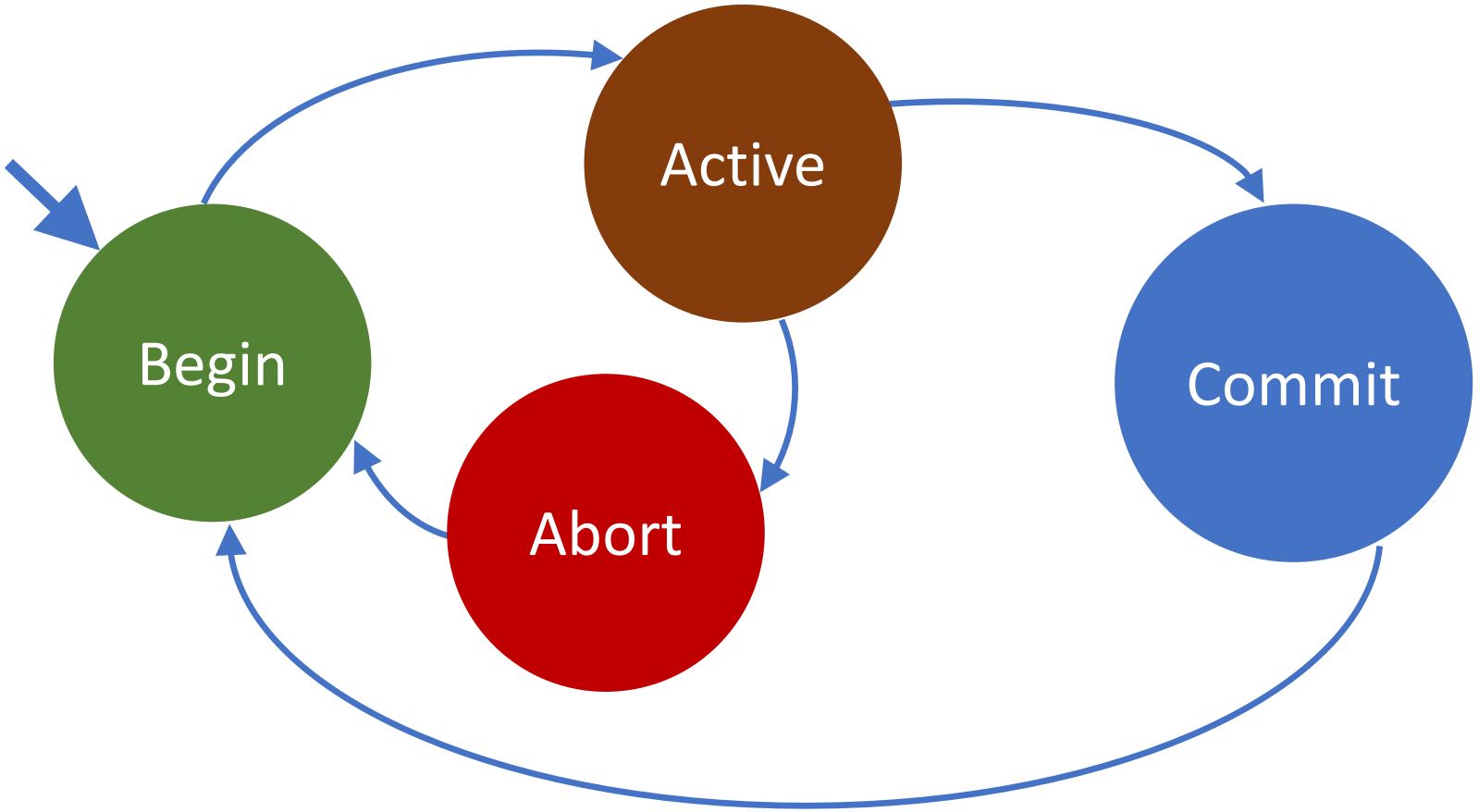
Correctness guarantee for traditional TM

P1. *The behavior of every **committed transaction** has to be justifiable by the same sequential execution containing only committed transactions, without contradicting real-time order.*

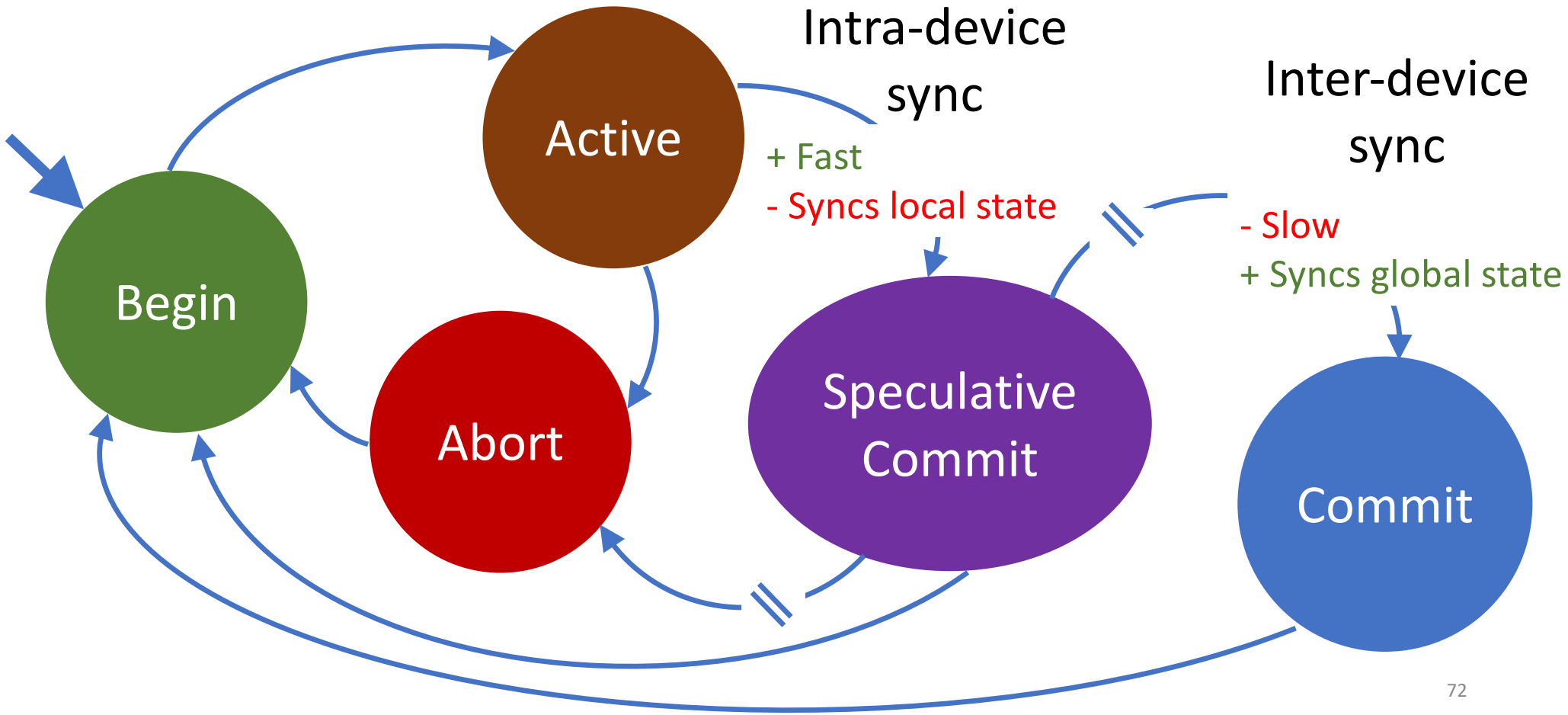
P2. *The behavior of any **active transaction**, even if it eventually aborts, has to be justifiable by some sequential execution (possibly different) containing only committed transactions.*

Hard notion of **committed transaction**:
need to transfer single transaction metadata over PCIe

Correctness guarantee for traditional TM



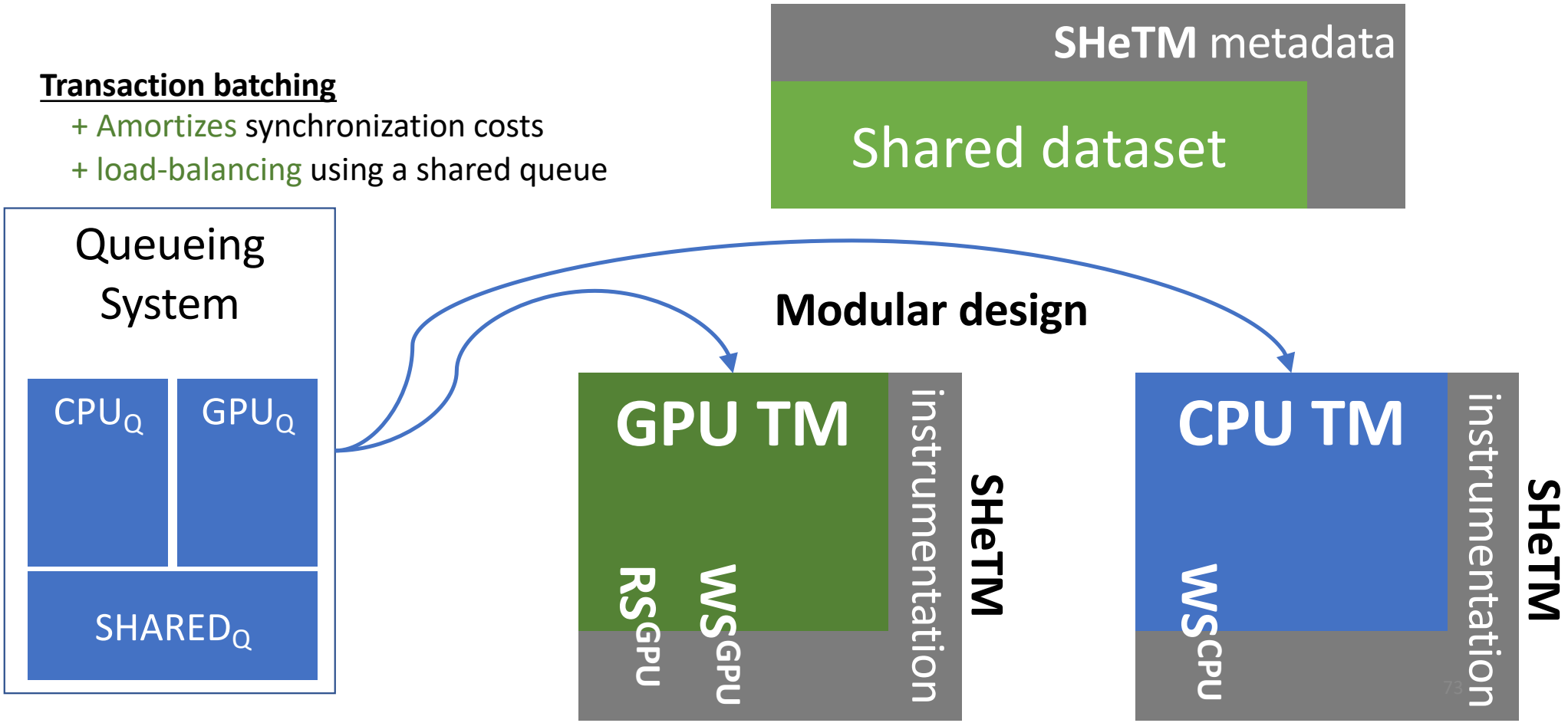
Correctness guarantee for HeTM



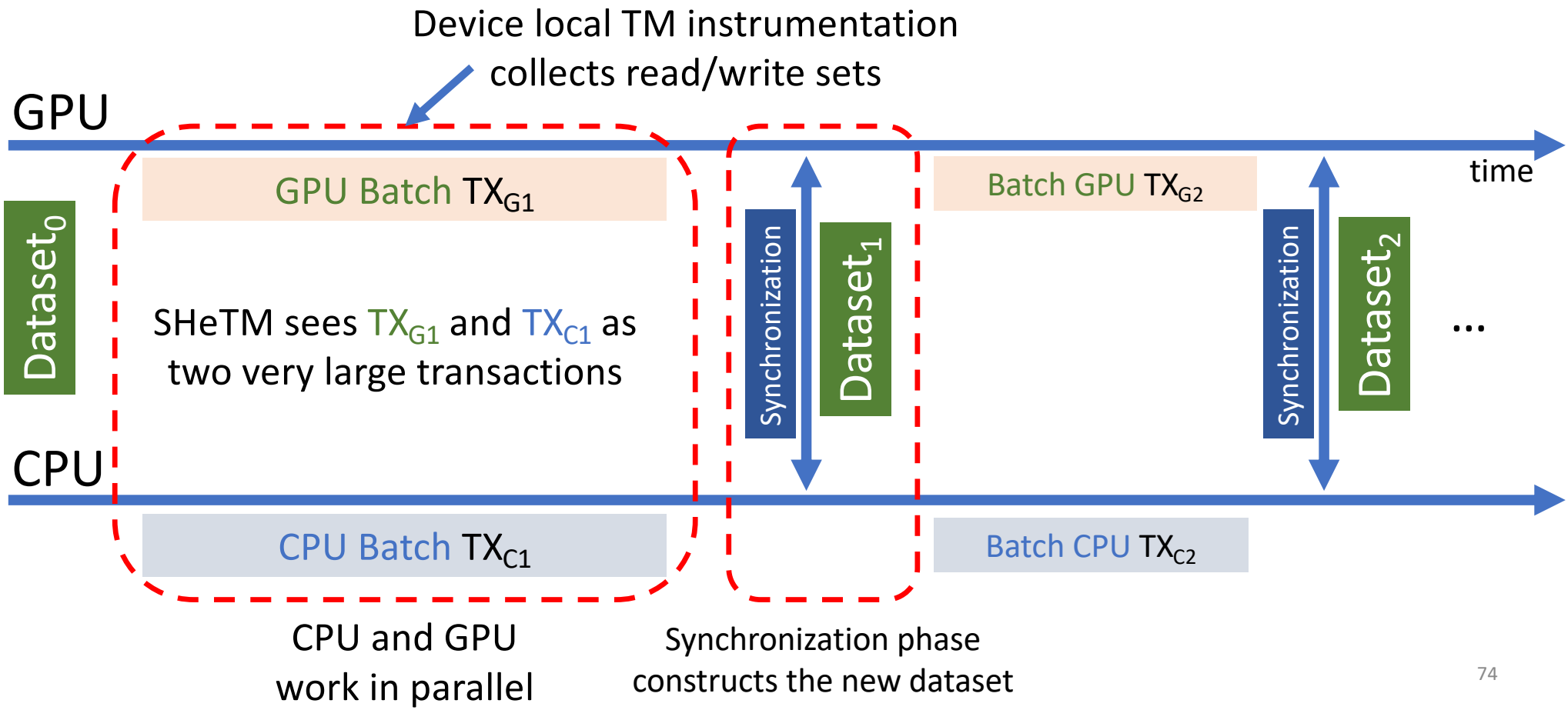
Speculative HeTM (SHeTM): architecture

Transaction batching

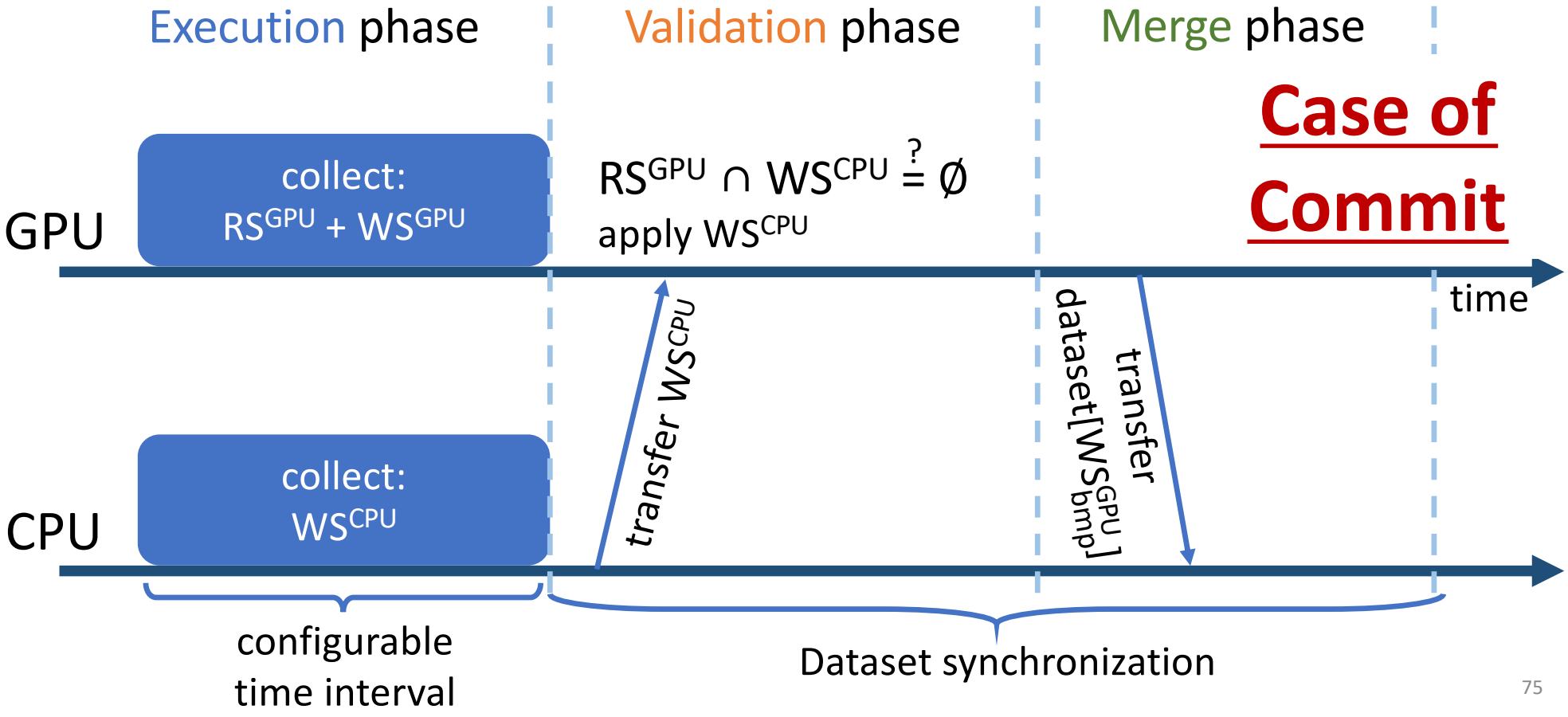
+ Amortizes synchronization costs
+ load-balancing using a shared queue



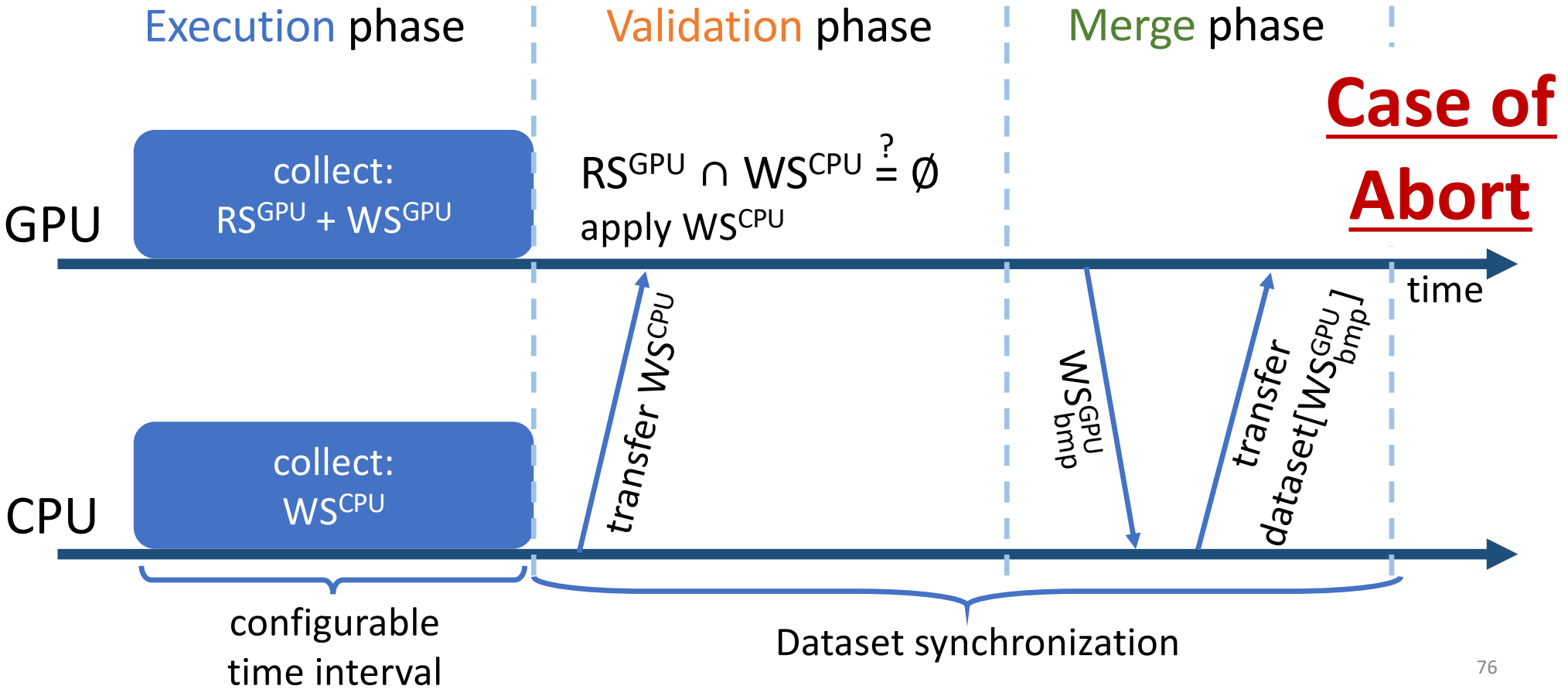
Speculative HeTM (SHeTM): overview



Base (unoptimized) idea

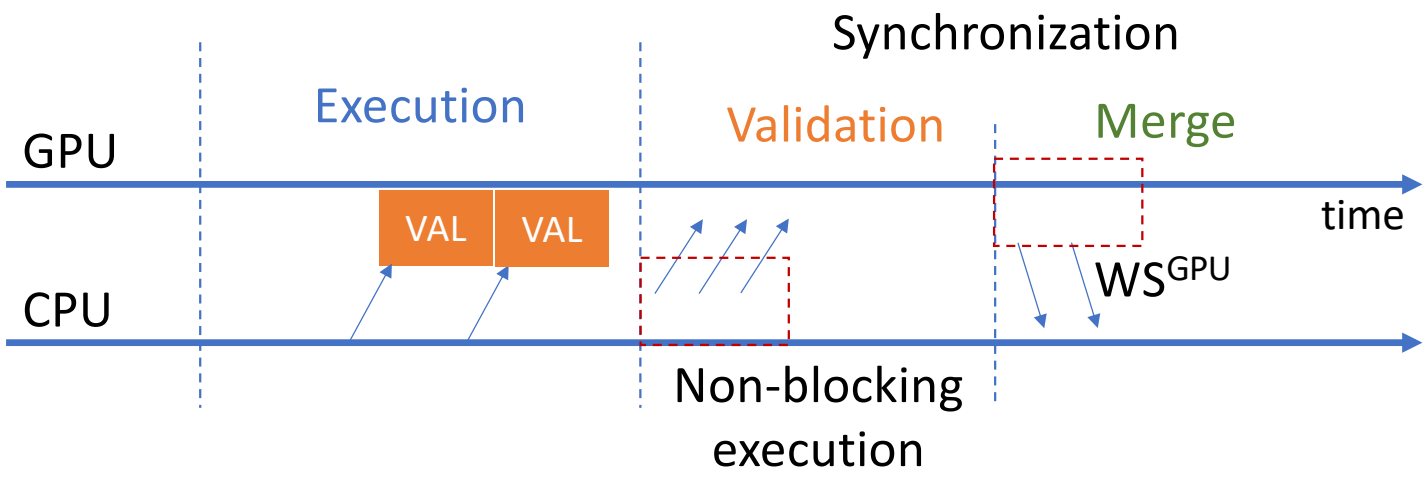


Base (unoptimized) idea



Optimizations

- Synchronization imposes significative overheads!
- Some optimizations:
  - Early validation kernels may reduce wasted work
  - Execution of transactions can be overlapped with synchronization stages



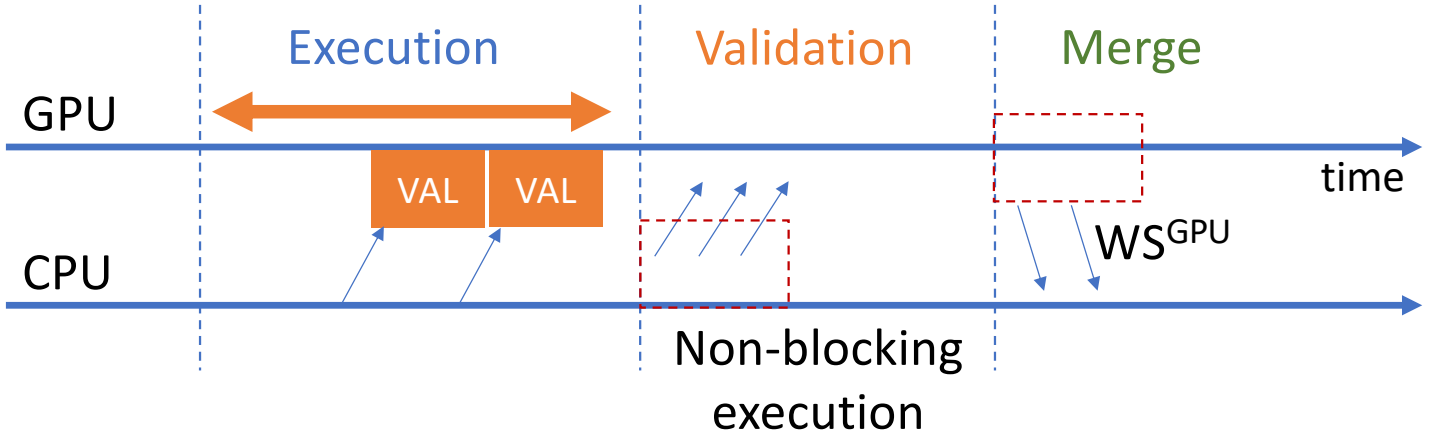
Details in the paper

Evaluation

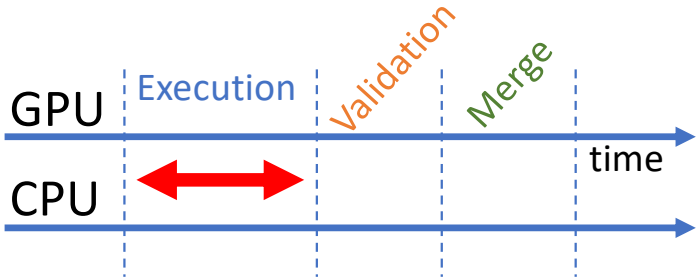
- Intel Xeon E5-2648L v4 (14C/28T, HTM, 32GB DRAM)
- Nvidia GTX 1080 (8GB XDDR5, driver 387.34, CUDA 9.1)
- CPU TM:
  - Intel's hardware TM implementation (TSX)
  - TinySTM in the paper
- GPU TM:
  - PR-STM [**EuroPar'15**]
- Synthetic benchmark
  - Random memory accesses on array of integers
- MemcachedGPU-TM
  - Popular web caching application

Synthetic benchmark

- Evaluate the impact of the duration of the Execution phase
  - Overhead of synchronization
- Benefits of two main optimizations
  1. Early validation
  2. Overlapping execution and synchronization

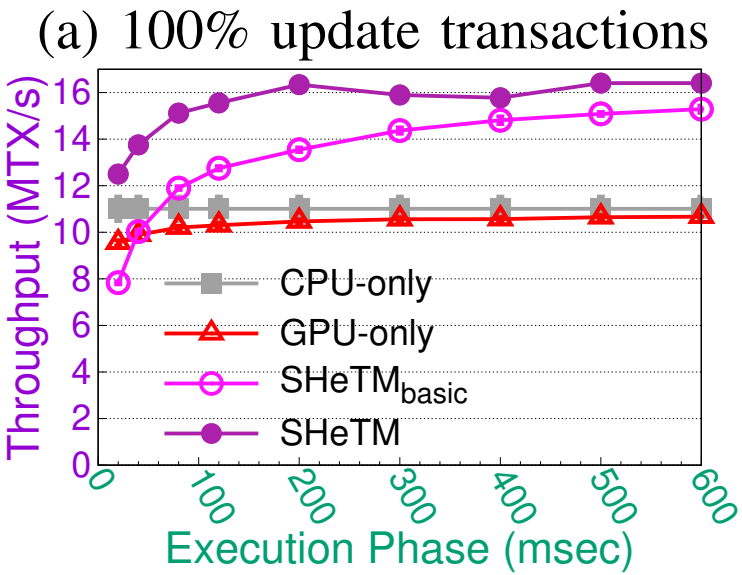


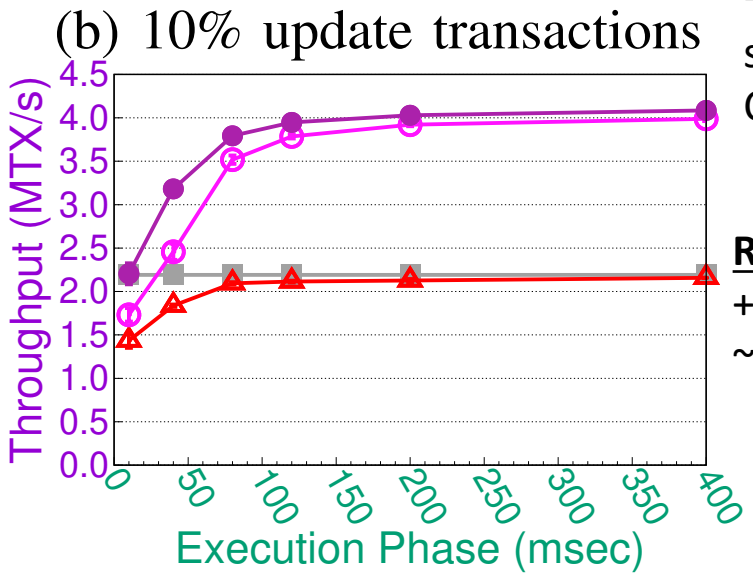
Synthetic benchmark – Execution time



In this experiment:

- no inter-devices conflicts (stresses the overheads of commit batches)

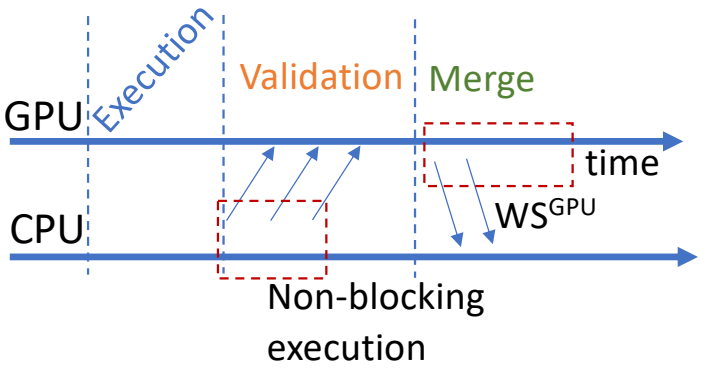




Write intensive workloads:
- stress more SHeTM
still only ~25% below sum CPU+GPU performance

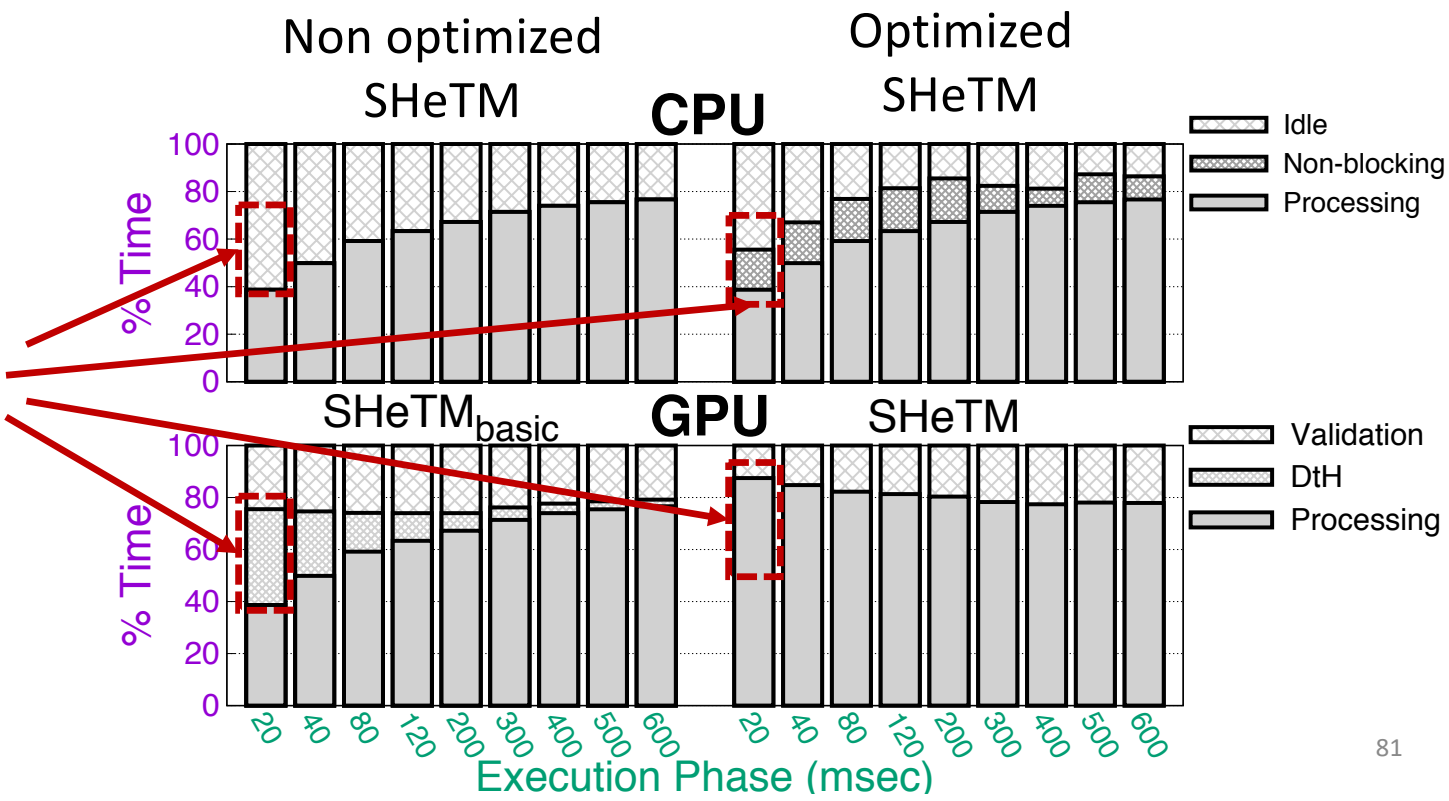
Read intensive workloads:
+ SHeTM throughput is ~95% the sum CPU+GPU

Synchronization overlapping



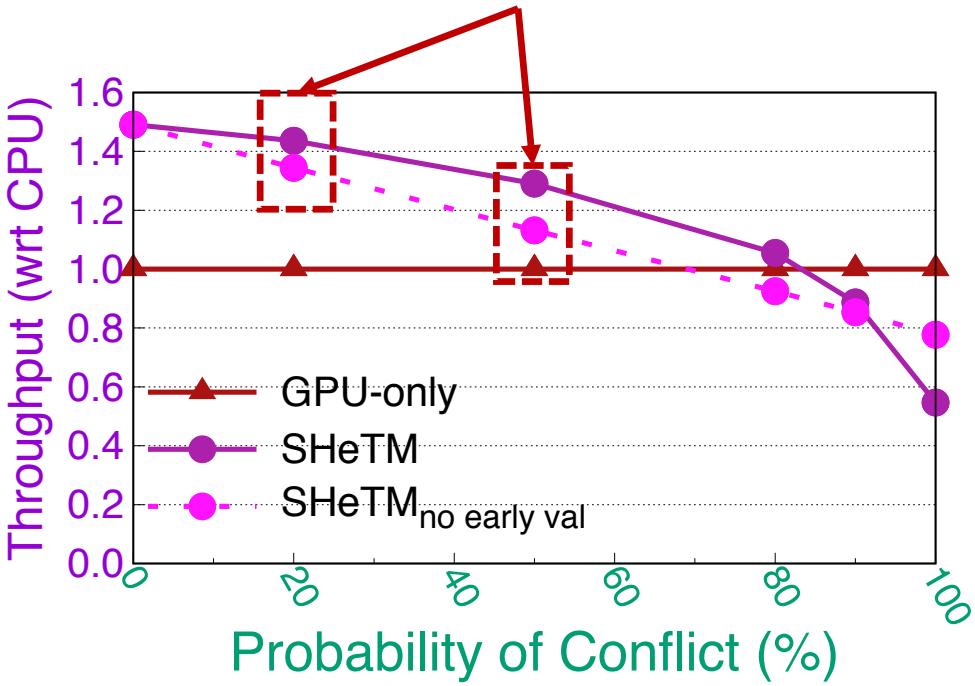
Significative reduction on CPU and GPU idle time:

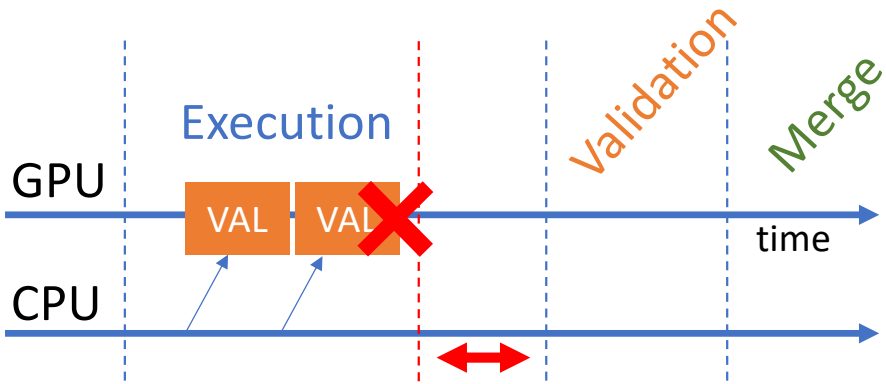
- CPU: 60% ➔ 45%
- GPU: 60% ➔ 20%



Synthetic benchmark – Early validation

Significative gains in medium contention scenarios





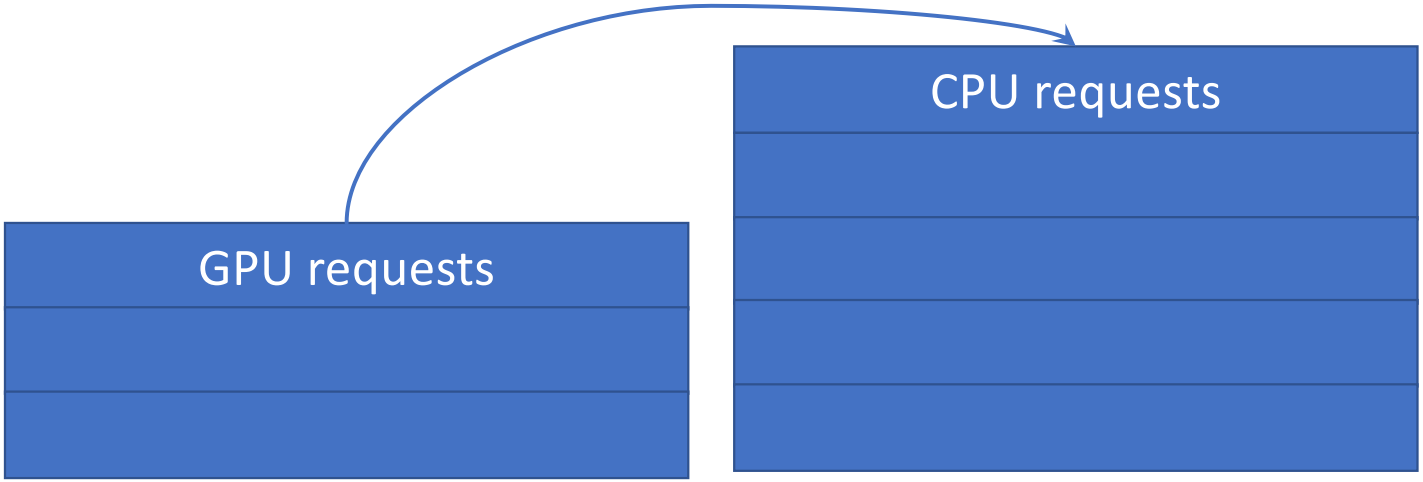
Early validation can detect and decrease the effective duration of conflicting batches

MemcachedGPU-TM

- Popular object caching system built by Facebook
- [**SoCC'15**]: port of Memcached to GPU
  - Complex lock-based scheme that unnecessarily restricts concurrency
- Workload:
  - 99.9% of GETs and key frequency follow a Zipfian distribution ($\alpha = 0.5$)
  - Keys partitioned based on last bit:
    - Odd keys ➔ GPU; Even keys ➔ CPU
  - Emulate load unbalances:
    - vary the popularity of keys maintained by GPU and CPU
    - GPU steals CPU requests (non-zero probability of conflicting in a key)

MemcachedGPU-TM

- Emulate load unbalances:
  - vary the popularity of keys maintained by GPU and CPU
  - GPU steals CPU requests (non-zero probability of conflicting in a key)



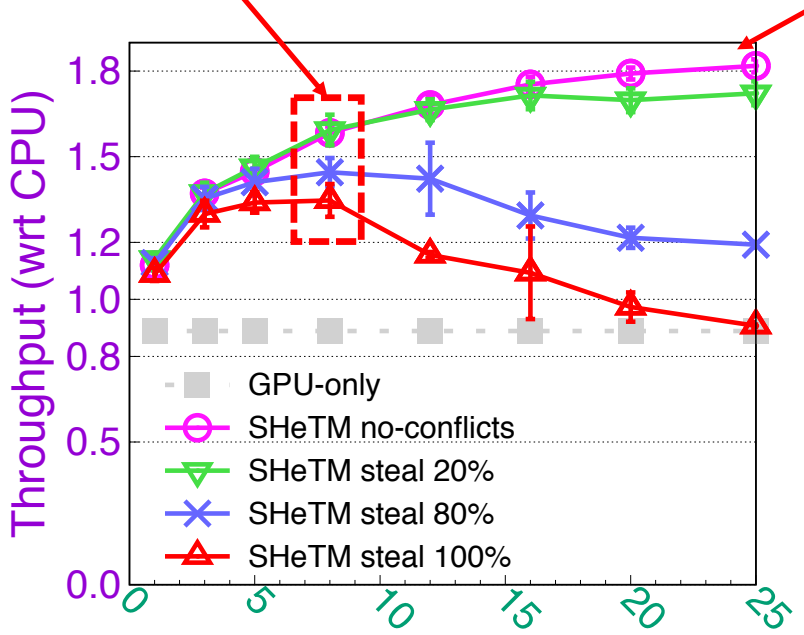
GPU Steal with probability X% (X=100% means that GPU operates only on the keys assigned to CPU)

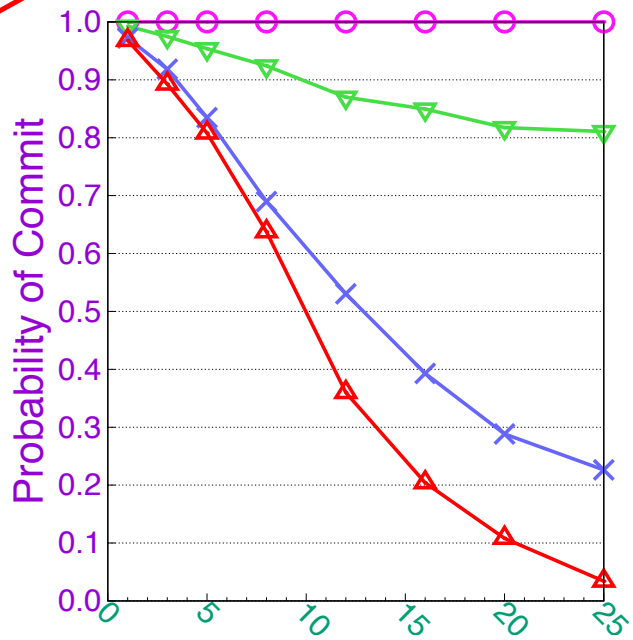
The higher the “steal” probability, the higher the inter-device contention probability

MemcachedGPU-TM

Tuning the durations allows high contention workloads to still benefit from CPU+GPU

overhead is ~10% in absence of contention





Execution phase (msec)

Ongoing & future work on TM + GPUs

- Extend SHeTM to support multiple GPUs
- Exploit integrated GPUs to accelerate STMs
- Design of STMs for GPUs

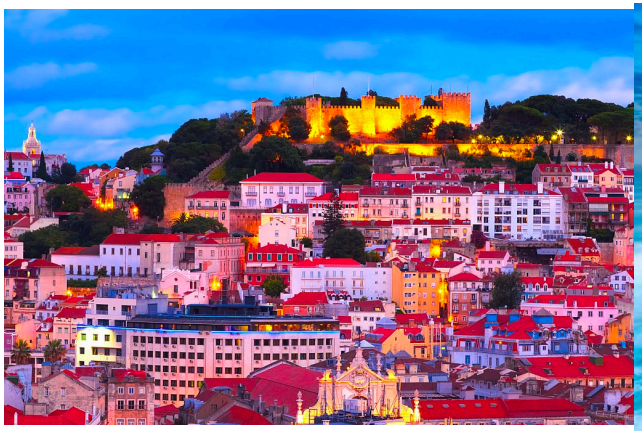
Conclusions

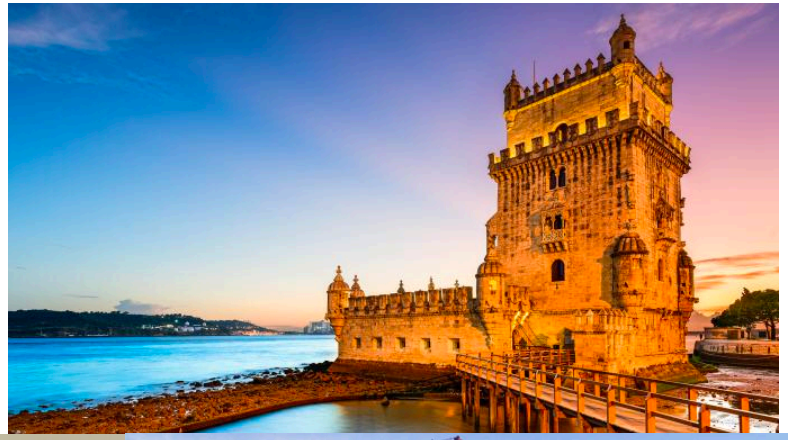
- TM is a promising paradigm for simplifying concurrent programming
  - Very hot research topic in the 1st decade of 2000
  - Today adopted in mainstream processors & programming languages

- New challenges/research opportunities are opened due to emergence of new hardware technologies:
  - Non-volatile memory
  - Heterogeneous architectures

- I would be glad to start collaborations on these fronts:
  - Get in touch with me: romano@inesc-id.pt
  - and meet f2f - I will be in Rome till Dec. 4

…or consider visiting my group in Lisbon!





Thanks for the attention!

Q&A