Storage Management in the NVRAM Era

Steven Pelley
University of Michigan
spelley@umich.edu

Thomas F. Wenisch
University of Michigan
twenisch@umich.edu

Brian T. Gold
Oracle Corporation
brian.t.gold@gmail.com

Bill Bridge
Oracle Corporation
bill.bridge@oracle.com

Preliminaries

Storage technology	Random read latency	Durable?
Disk	10ms	✓
Flash	90μs	✓
DRAM	100ns	✗
NVRAM	50-1000ns [IBM]	✓

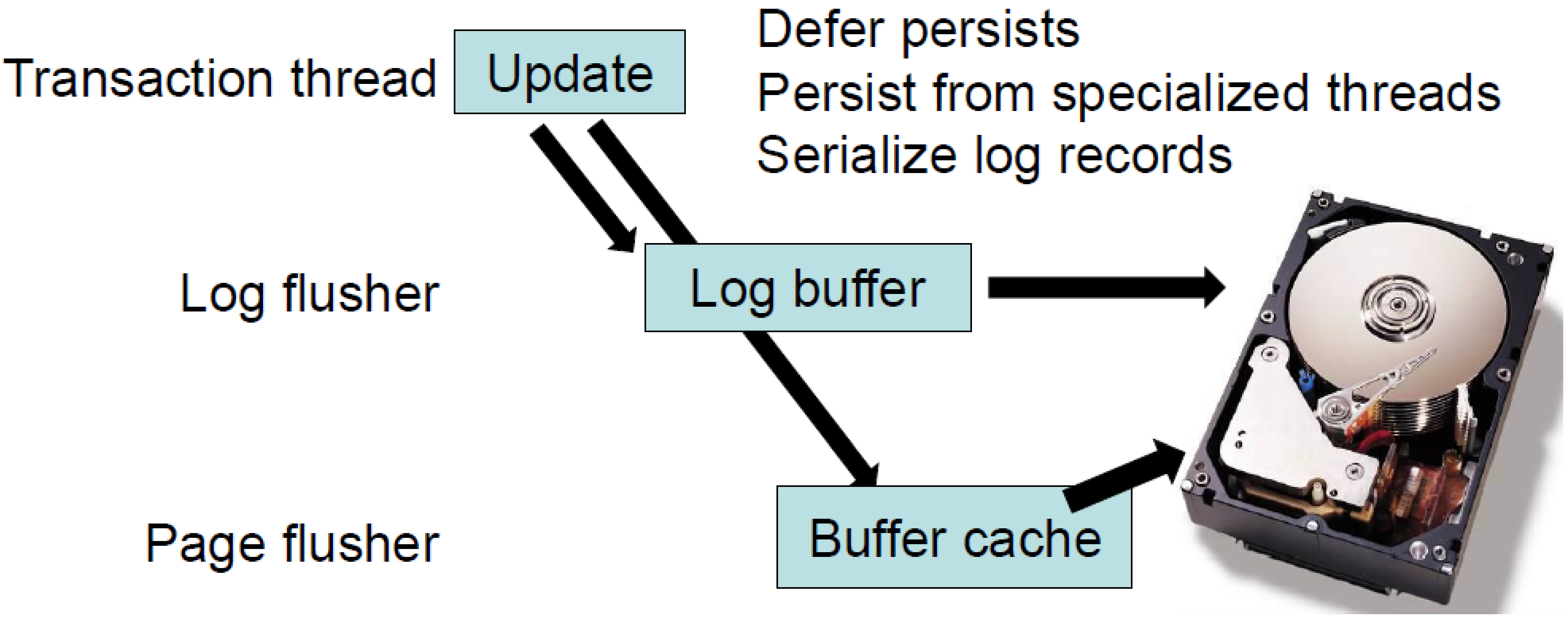
- Disk
- Flash
  - e.g. SSD (Solid State Disk)
- DRAM (Dynamic Random Access Memory)
- NVRAM (Non-Volatile Random Access Memory)
  - Retains its information when power is turned off
  - e.g. phase change, memristor and STT-RAM
- OLTP (On-Line Transaction Processing)
  - Traditional solution: DRAM + Disk
  - Can NVRAM revolutionize OLTP durability management?

Summary

- What
  - Redesign durable storage and recovery management for OLTP to take advantage of the low latency and byte-addressability of NVRAM
- Why
  - Disk and Flash are slow while fast NVRAM has emerged as a viable alternative
- How
  - NVRAM Disk-Replacement
  - NVRAM In-Place Updates
  - NVRAM Group Commit

NVRAM Disk-Replacement

Write Ahead Logging (WAL) via ARIES



NVRAM Disk-Replacement (cont.)

- Pros
  - Insensitive to large persist barrier delays

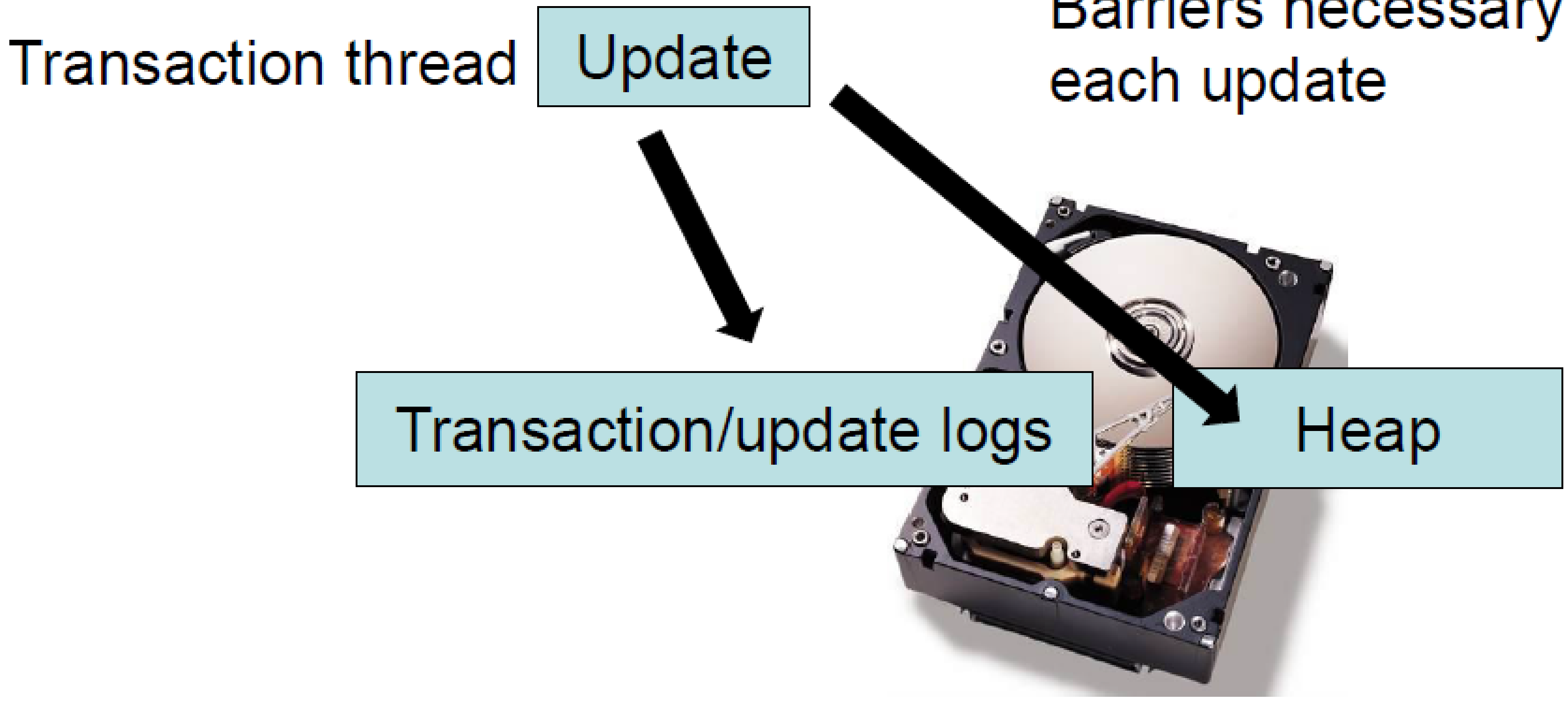
1. Enforcing the order in which data persistently writes to the device
2. Notifying the user that their data are durable (e.g., to commit a transaction)

Persist barriers can introduce expensive synchronous delays on transaction threads

- Cons
  - However, it assumes IO delays are the dominant performance bottleneck and trades off software overhead to minimize IO

NVRAM In-Place Updates

Persist immediately
Remove centralized log
Barriers necessary at each update



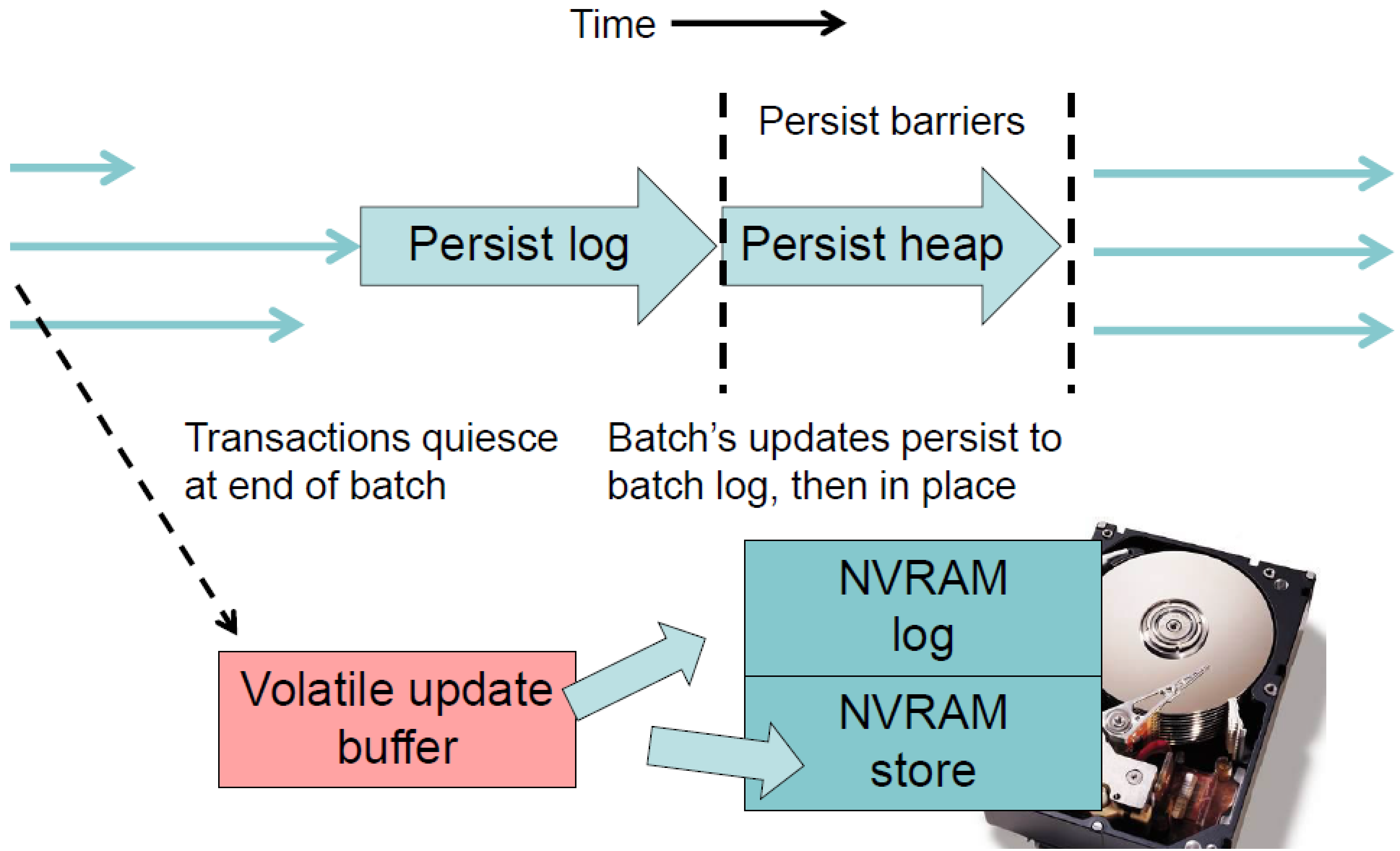
NVRAM In-Place Updates (cont.)

- Pros
  - Removes expensive software overhead
  - Excels when persist barriers delays are short
- Cons
  - Introduces persist barriers on transactions' critical paths
  - As persist barrier latency increases performance suffers

NVRAM Group Commit

- Can we have both NVRAM Disk-Replacement's persist barrier latency insensitivity and NVRAM In-Place Updates's low software overhead?
  - Yes!
- It should require fewer persist barriers than NVRAM In-Place Updates and avoids NVRAM Disk-Replacement's logging
  - Executing transactions in batches, whereby all transactions in the batch commit or (on failure) all transactions abort

NVRAM Group Commit



Operating System	Ubuntu 12.04
CPU	Intel Xeon E5645 2.40 GHz
CPU cores	6 (12 with HyperThreading)
Memory	32 GB

Table 2: Experimental system configuration.

Modeling unavailable devices

- Run database on real hardware
  - Log and db heap on RAMDisk(or just in DRAM)
  - Introduce precise delays (20ns precision using x86 RDTSCP) to model persist barrier latency
- Build recovery mechanisms in software
  - Shore-MT: research platform for high performance transaction processing
  - Rely on dirty bit fields to track buffer pool writes during transaction, page latch, or batch
- Workloads
  - TPCC, TPCB and TATP

Workload	Scale factor	Size	Write transaction
TPCC	70	9GB	New order
TPCB	1000	11GB	
TATP	600	10GB	Update location

Table 3: Workloads and transactions.

Recovery management performance

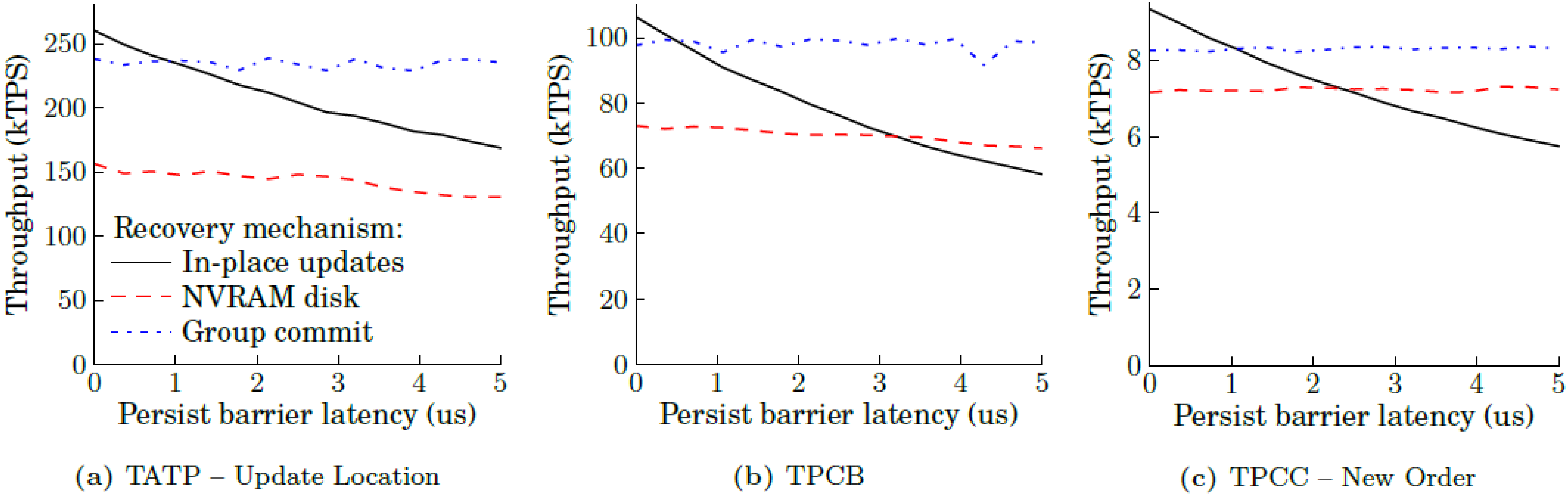


Figure 5: Throughput vs persist barrier latency. *In-Place Updates* performs best for zero-cost persist barriers, but throughput suffers as persist barrier latency increases. *NVRAM Disk-Replacement* and *NVRAM Group Commit* are both insensitive to increasing persist barrier latency, with *NVRAM Group Commit* offering higher throughput.

Transaction Latency

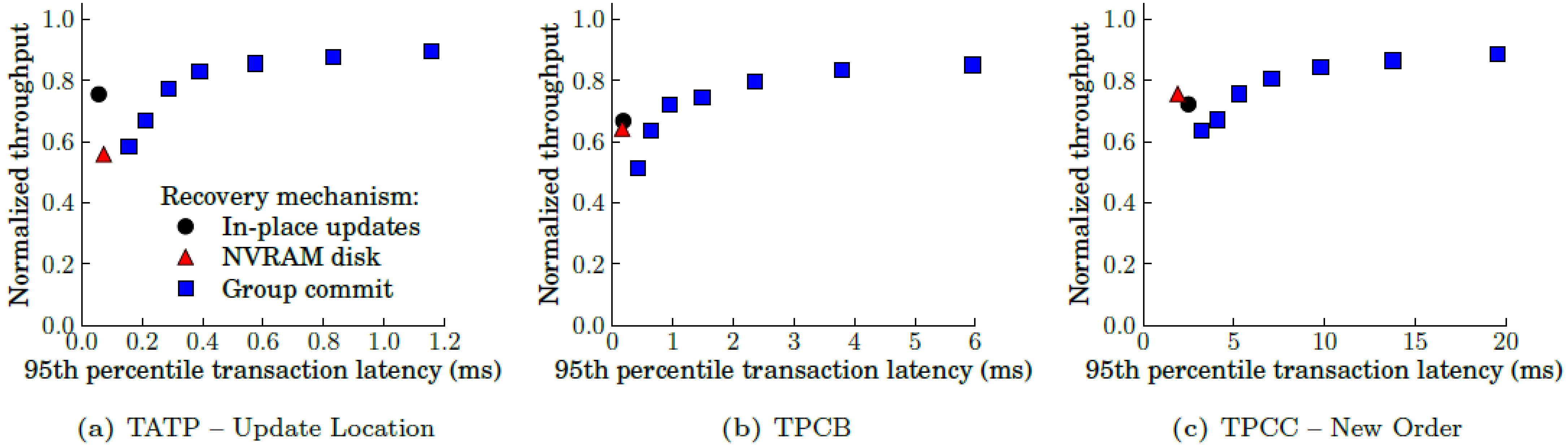


Figure 6: 95th percentile transaction latency. Graphs normalized to *In-Place Updates* 0µs persist latency. Experiments use 3µs persist latency. *NVRAM Group Commit* avoids high latency persist barriers by defering transaction commit.

Conclusion

- Disk-based software carries baggage
- Frequent persist synchronization also slow
- New software and memory systems improve performance and simplify software design
- Future work: NVRAM optimizations

Multi-Core, Main-Memory Joins: Sort *vs.* Hash Revisited

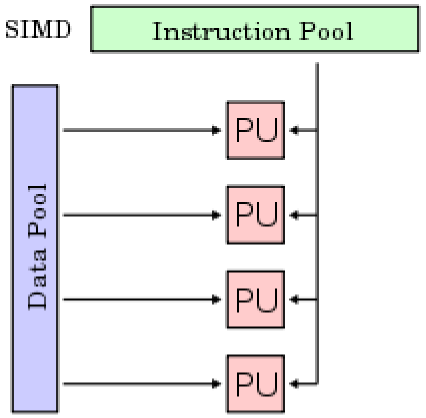
Cagri Balkesen, Gustavo Alonso
Systems Group, ETH Zurich
Switzerland
{name.surname}@inf.ethz.ch

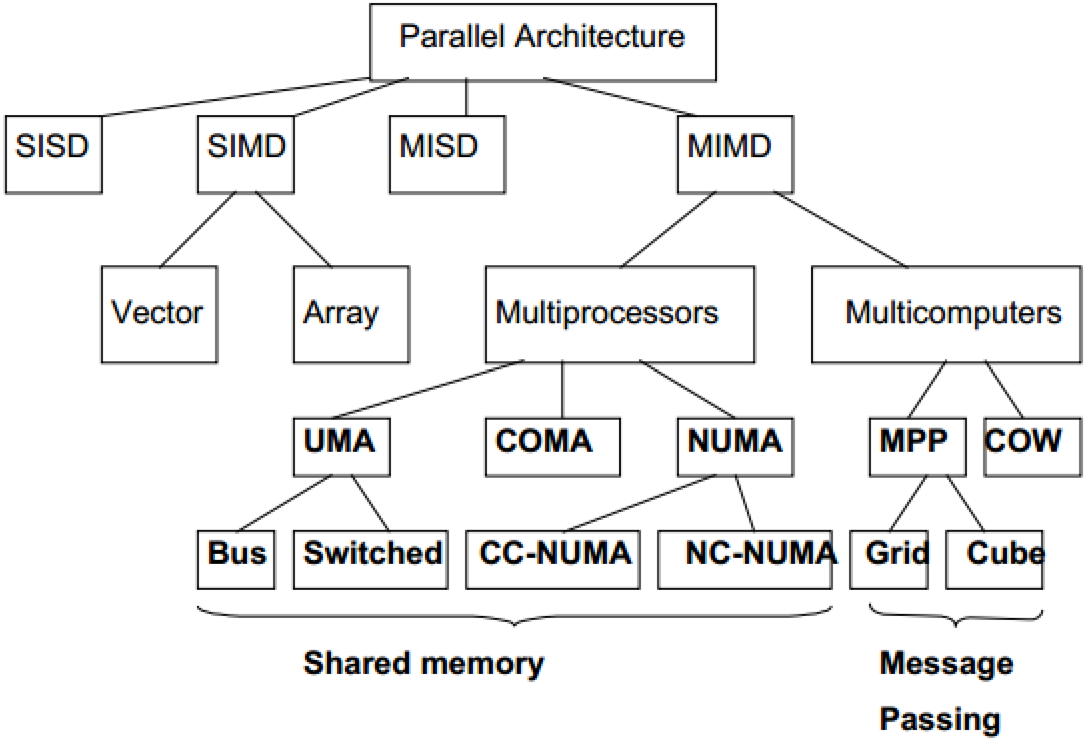
Jens Teubner
TU Dortmund University
Germany
jens.teubner@cs.tu-dortmund.de

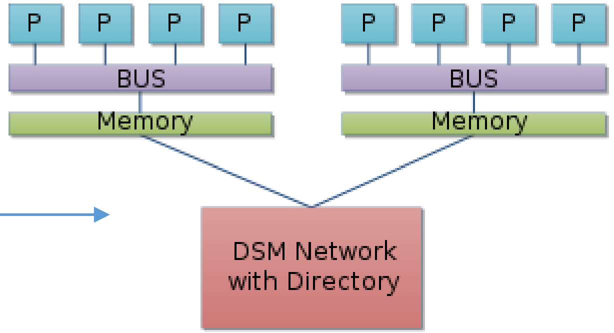
M. Tamer Özsu
University of Waterloo
Canada
tamer.ozsu@uwaterloo.ca

Preliminaries

- Disk seek and transfer
- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Sort-merge join
- Hash join
- SIMD (Single Instruction Multiple Data)
- NUMA (Non-uniform memory access)







Summary

- What
  - Experimentally study the performance of main-memory, parallel, multi-core join algorithms, focusing on sort-merge and (radix-)hash join
- Why
  - A wide range of experimental factors and parameters: algorithm design, data sizes, relative table sizes, degree of parallelism, use of SIMD instructions, effect of NUMA, data skew, and different workloads
  - Many of these parameters and combinations thereof were not foreseeable in earlier studies, and our experiments show that they play a crucial role in determining the overall performance of join algorithms
- How
  - Implement original and optimized sort-merge and hash join in the multi-core and main-memory envirement

Sort-based Join

- Strategies to implement sorting in a hardware-conscious manner
- Sorting Networks
  - Min/Max instructions
- Speedup Through SIMD
  - Shuffle operations

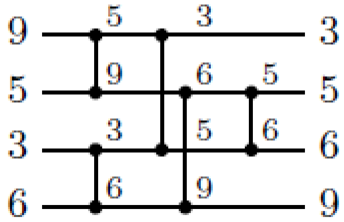


Figure 1: Even-odd network for four inputs.

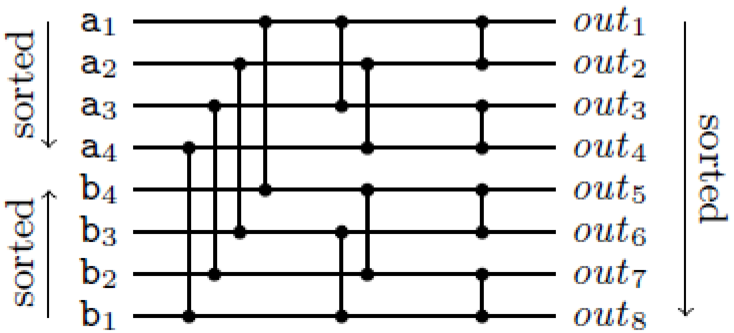


Figure 2: Bitonic merge network.

Sort-based Join (cont.)

- Cache Conscious sort joins
- In-Register Sorting
  - With runs that fit into (SIMD) CPU registers
- In-Cache Sorting
  - Where runs can still be held in a CPU-local cache
- Out-of-Cache Sorting
  - Once runs exceed cache sizes

Hash-based Join

- While efficient, hashing results in random access to memory, which can lead to cache misses
- As a result, a partitioning phase to the hash joins is introduced to reduce cache misses
- Radix Partitioning
  - Considering as well the effects of translation look-aside buffers(TLBs)
- Software-Managed Buffers
  - The idea is to allocate a set of buffers, one for each output partition and each with room for up to N input tuples

Experimental Setup

short notation	algorithm
m-way	Sort-merge join with multi-way merging
m-pass	Sort-merge join with multi-pass naïve merging
mpsm	Our impl. of massively parallel sort-merge [2]
radix	Parallel radix hash join [15, 4]
n-part	No-partitioning hash join [5, 4]

Table 1: Algorithms analyzed.

	A (adapted from [2])	**B** (from [15, 4])
size of *key* / *payload*	4 / 4 bytes	4 / 4 bytes
size of R	$1600 \cdot 10^6$ tuples	$128 \cdot 10^6$ tuples
size of S	$m \cdot 1600 \cdot 10^6$ tuples, m = 1,..,8	$128 \cdot 10^6$ tuples
total size R	11.92 GiB	977 MiB
total size S	$m \cdot 11.92$ GiB	977 MiB

Table 2: Workload characteristics.

Sort-merge join performance

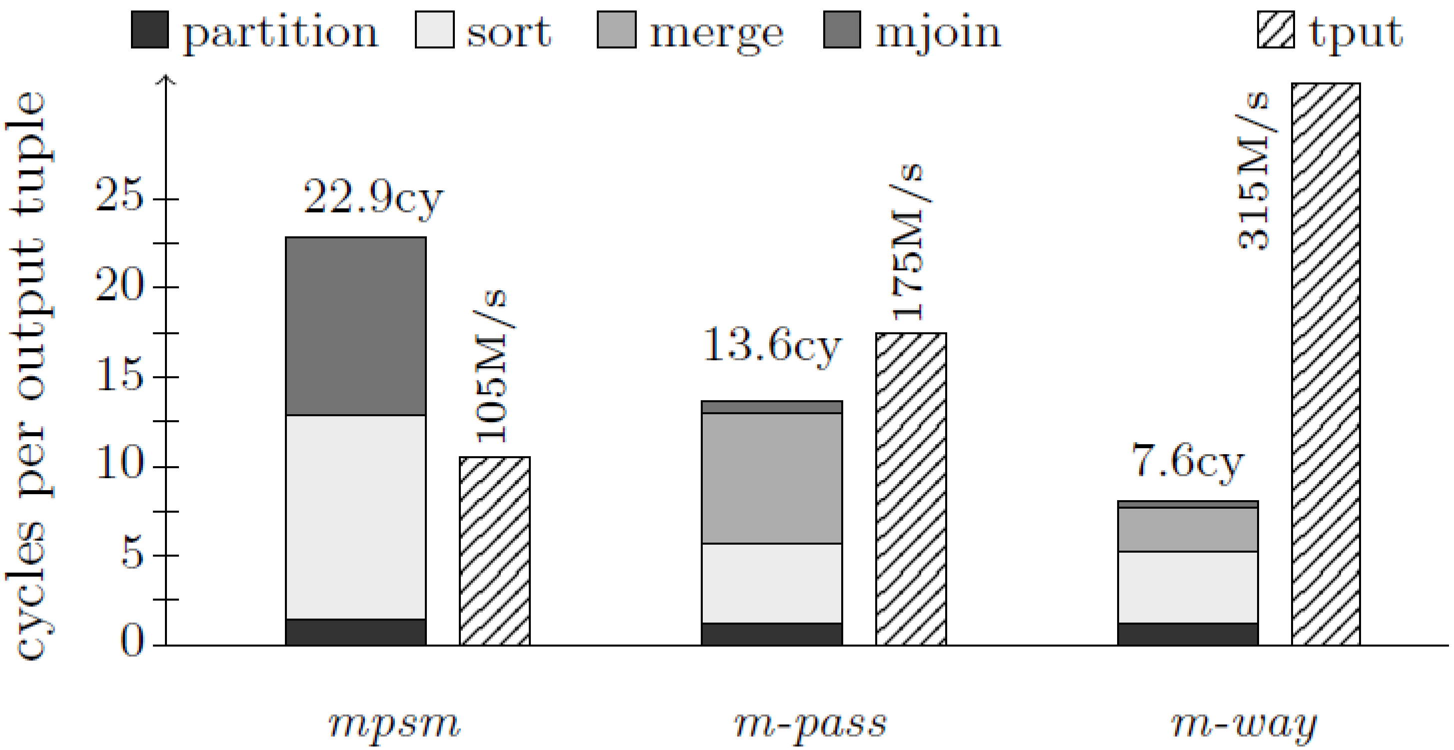
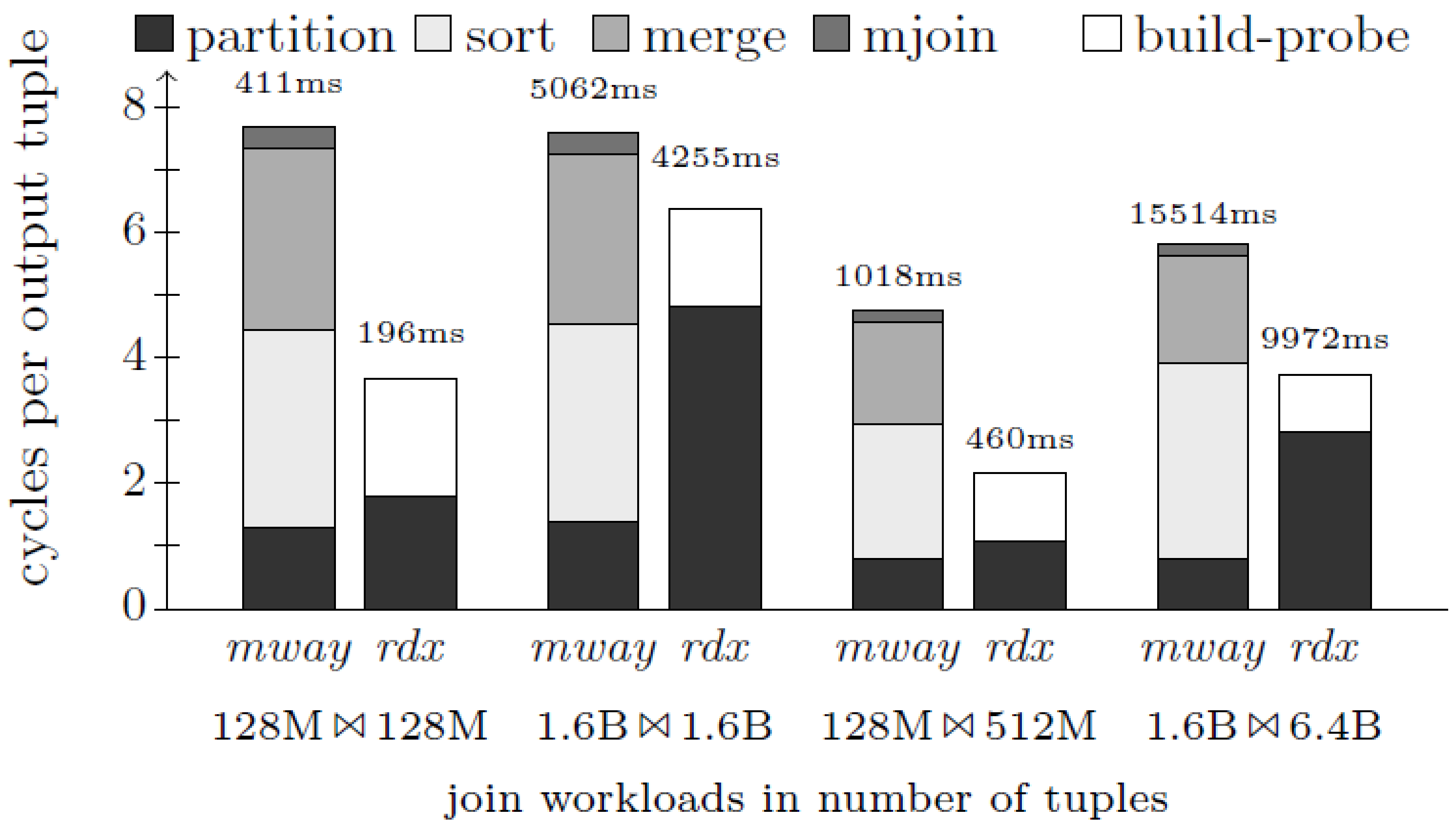


Figure 11: Performance breakdown for sort-merge join algorithms. Workload A. Throughput metric is output tuples per second, *i.e.* $|S|/\text{execution time}$.

Sort or Hash?



partition
sort
merge
mjoin
build-probe
cycles per output tuple
0
2
4
6
8
411ms
196ms
5062ms
4255ms
1018ms
460ms
15514ms
9972ms
mway rdx
mway rdx
mway rdx
mway rdx
128M ⋈ 128M
1.6B ⋈ 1.6B
128M ⋈ 512M
1.6B ⋈ 6.4B
join workloads in number of tuples

Effect of Input Size

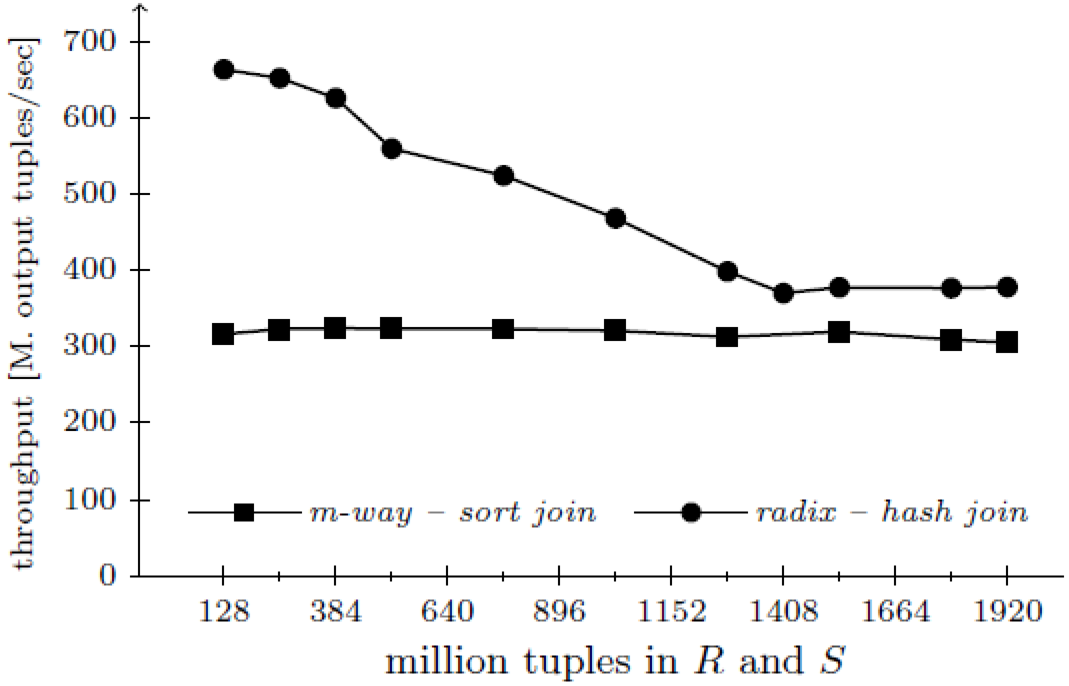


Figure 15: Sort vs. hash with increasing input table sizes ($|R| = |S|$). Throughput metric is total output tuples per second, *i.e.* $|S|/\text{execution time}$.

Effect of Skew

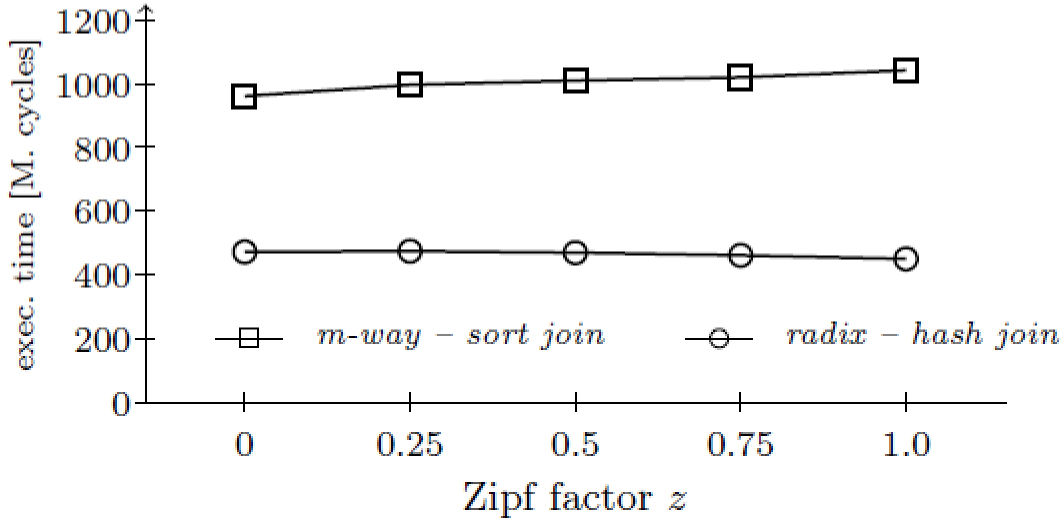
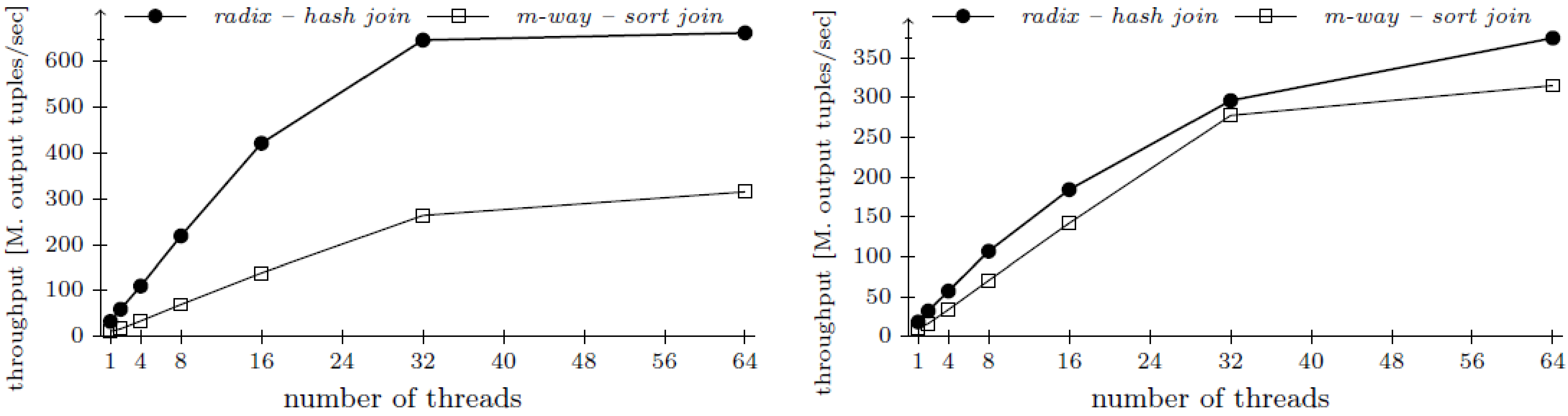


Figure 16: Join performance when foreign key references follow a Zipfian distribution. Workload B.

Scalability Comparison



(a) 977 MiB $\bowtie$ 977 MiB (128 million 8-byte tuples) (b) 11.92 GiB $\bowtie$ 11.92 GiB (1.6 billion 8-byte tuples)

Figure 17: Scalability of sort vs. hash join. Throughput is in output tuples per second, *i.e.* $|S|/$execution time.

Sort vs. Hash with All Algorithms

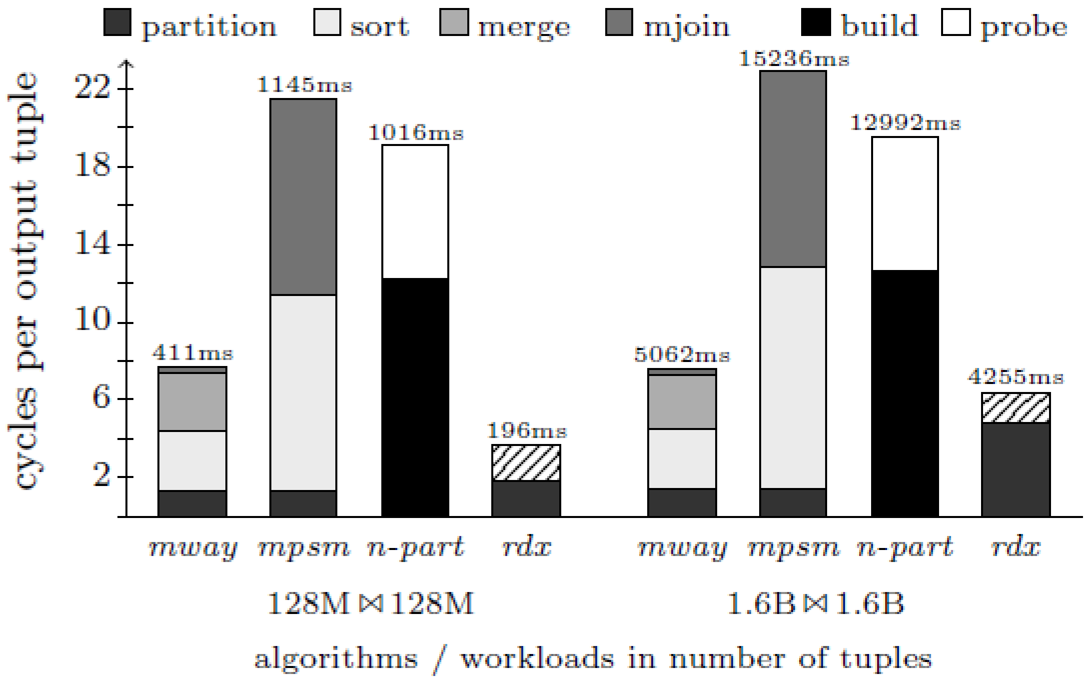


Figure 18: Sort vs. hash join comparison with extended set of algorithms. All using 64 threads.

Conclusion

- Hash-based join algorithms still have an edge over sort-merge joins despite the advances on the hardware side
- Sort-merge join turns out to be more comparable in performance to radix-hash join with very large input sizes

Thanks!