Real-Time Wait-Free Queues

using Micro-Transactions

Fadi Meawad[1], Karthik Iyer[1],
Martin Schöberl[2], Jan Vitek[1]

[1] S3Lab
Computer Science Department
Purdue University

[2] Department of Informatics and
Mathematical Modeling
Technical University of Denmark

Compare-and-Swap (CAS), and its limitations?

What is CAS?

- Compare and Swap (or sometimes called Compare and Exchange)
- *Atomically* compares the value of a current memory location to a given value, and if it is the same, the memory location is updated with a new value
- Initial hardware realizations were slow, the performance of CAS is now comparable to regular instructions

Limitations?

- Operates on a *single* memory location
- Some algorithms require *Multi-word CAS (MCAS)*

Transactional Memory (TM)

What is Transactional Memory?

- An alternative synchronization infrastructure
- Transactions are:
  - non-blocking
  - serializable
  - atomic read/write operations

Why not just use it?

- Software TM (STM) exhibit unacceptable performance.
- Hardware TM (HTM) require a programmer's awareness of Cache and Buffer sizes.

Is TM a good alternative for MCAS?

Maybe?

- An efficient Hardware Implementation Optimized for Micro-Transactions can replace MCAS
- Smaller Transactions are more likely to *Commit*
- Micro-Transactions will fit in most Caches and Buffers (HTM).

Maybe Not?

- A single CAS or 2 Consecutive CAS is faster than a Hardware Micro-Transaction (on modern hardware)
- Small Transactions might incur a high overhead

Where do we find an efficient Hardware Micro-Transactions?

Java Optimized Processor (JOP)

- Hardware Implementation of the Java Virtual Machine
- Time Predictable
- Low Level WCET Analysis
- Implemented as a soft-core CMP in FPGA with up to 8-cores in an Altera Cyclone II FPGA

Transactions in JOP:

- Fully Associative buffer cache local to each core caching changed data (write-set)
- Set of tag memories read and not cached (read-set)
- Conflict: Read-set of one Transaction *interferes* with the write-Set of another one.
- Conflict detection happens on Commit, Commits are serialized.

Implementation

Variations:

- JOP does not have native support for CAS
- Each CAS is either simulated with TM or with Lock leading to 4 variations:
  CAS_LOCK, CAS_TM, LOCK, TM
  - Example:

```
// CAS_TM
@atomic boolean CASHead(Node oldh, Node newh) {
  if (head == oldh) { head = newh;  return true;
   } else return false;
}
```

Queues:

- FIFO Queues
- insert at tail, remove from head

Example: Queue Implementations

Singly Linked Queue (SLQ):

- with 4 variants

Doubly Linked Queue (DLQ):

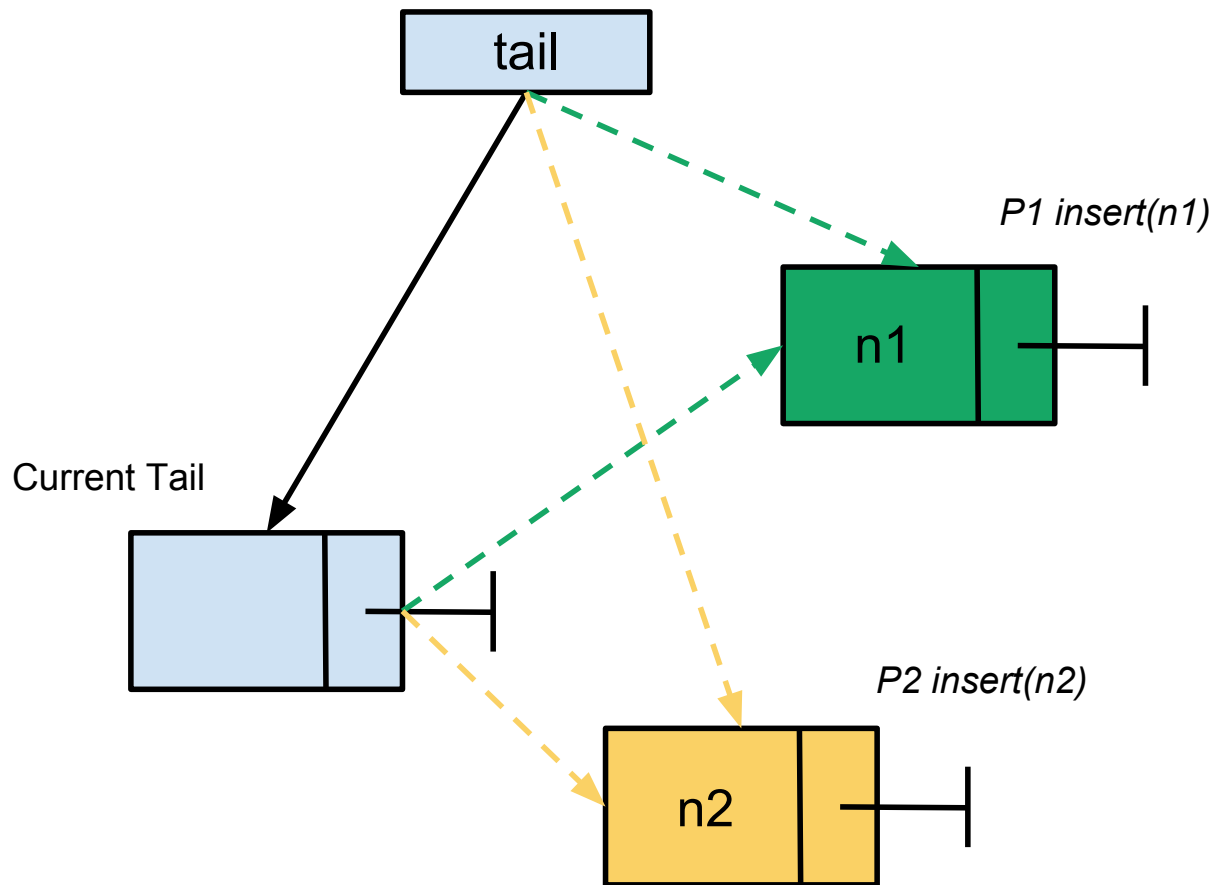
- with 4 variants
- Optimistic, in case an inconsistency occur, fixed later

Limited Capacity Queue (LIMQ):

- Wait-free implementations usually limit to a single reader or a single writer
- only TM and Lock implementations

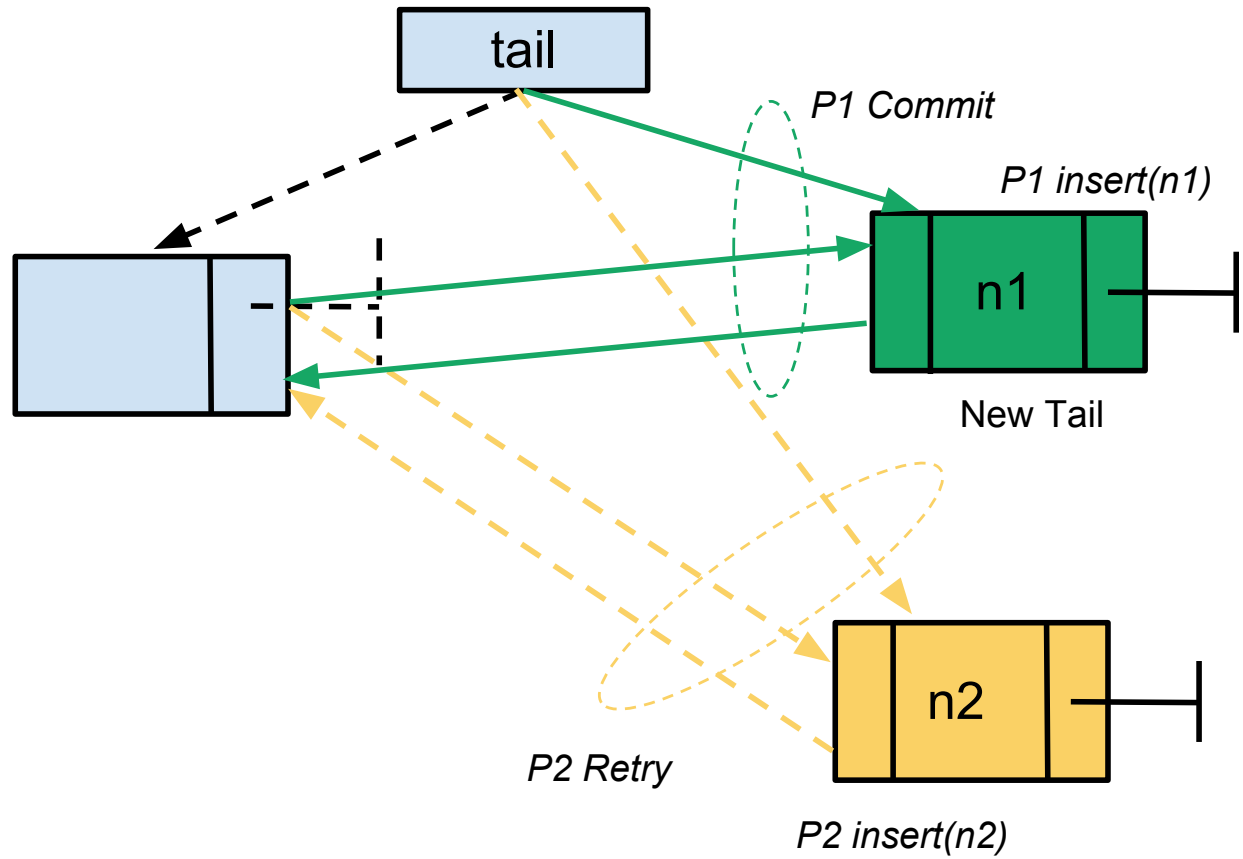
Singly Linked Queue (SLQ)

- 4 variants
- Uses 2 Consecutive CAS



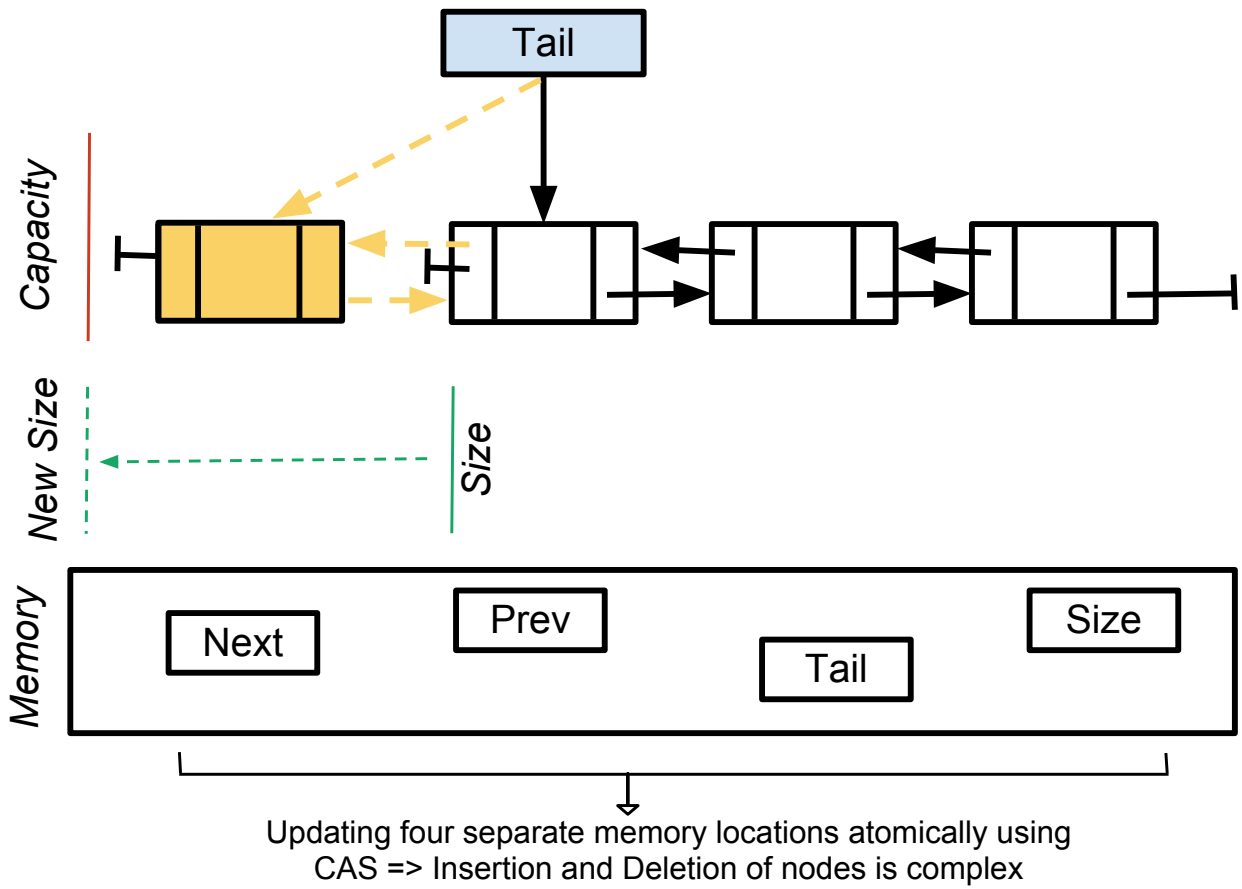
Doubly Linked Queue (DLQ)

- 4 variants
- Optimistic, in case an inconsistency occur, fixed later, uses a single CAS



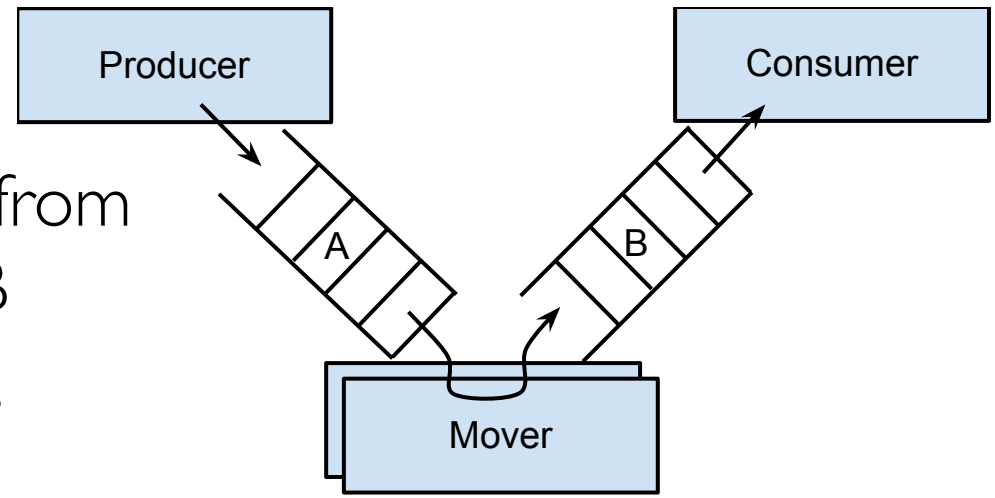
Limited Capacity Queue (LIMQ)

- Current implementations limit to a single reader or a single writer
- Only TM and Lock implementations

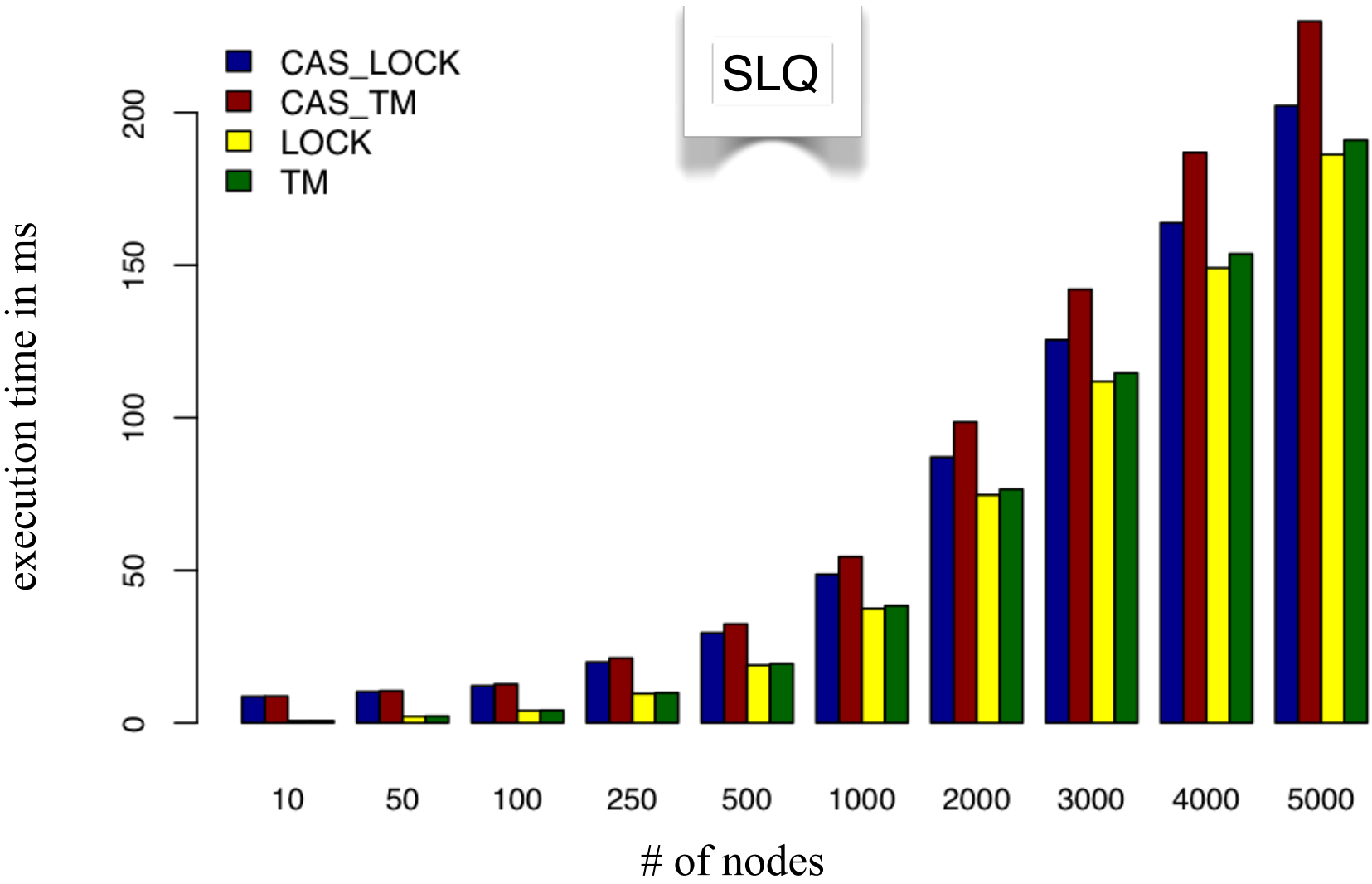


Experimentation & Evaluation

- FPGA programmed with a symmetric shared-memory multi-processor hardware system with 4 JOP cores.
- Using Altera DE2-70 Dev. board consisting of a Cyclone II EP2C70 FPGA.
- **Producer-Consumer Framework:**
  - Producer inserts into queue A
  - 2 Movers removes from A and inserts into B
  - Consumer removes from B

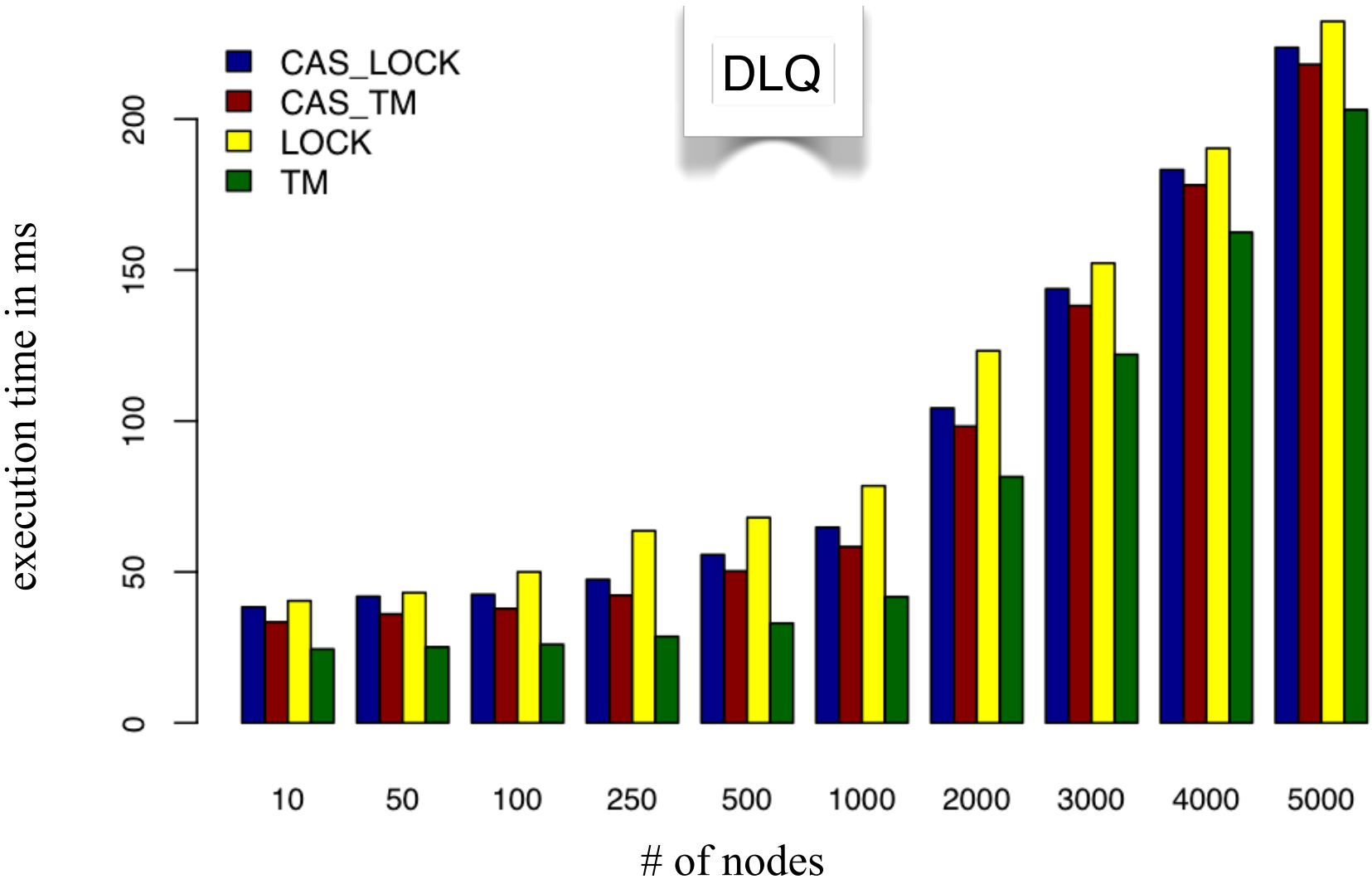


Evaluation - Singly Linked Queue



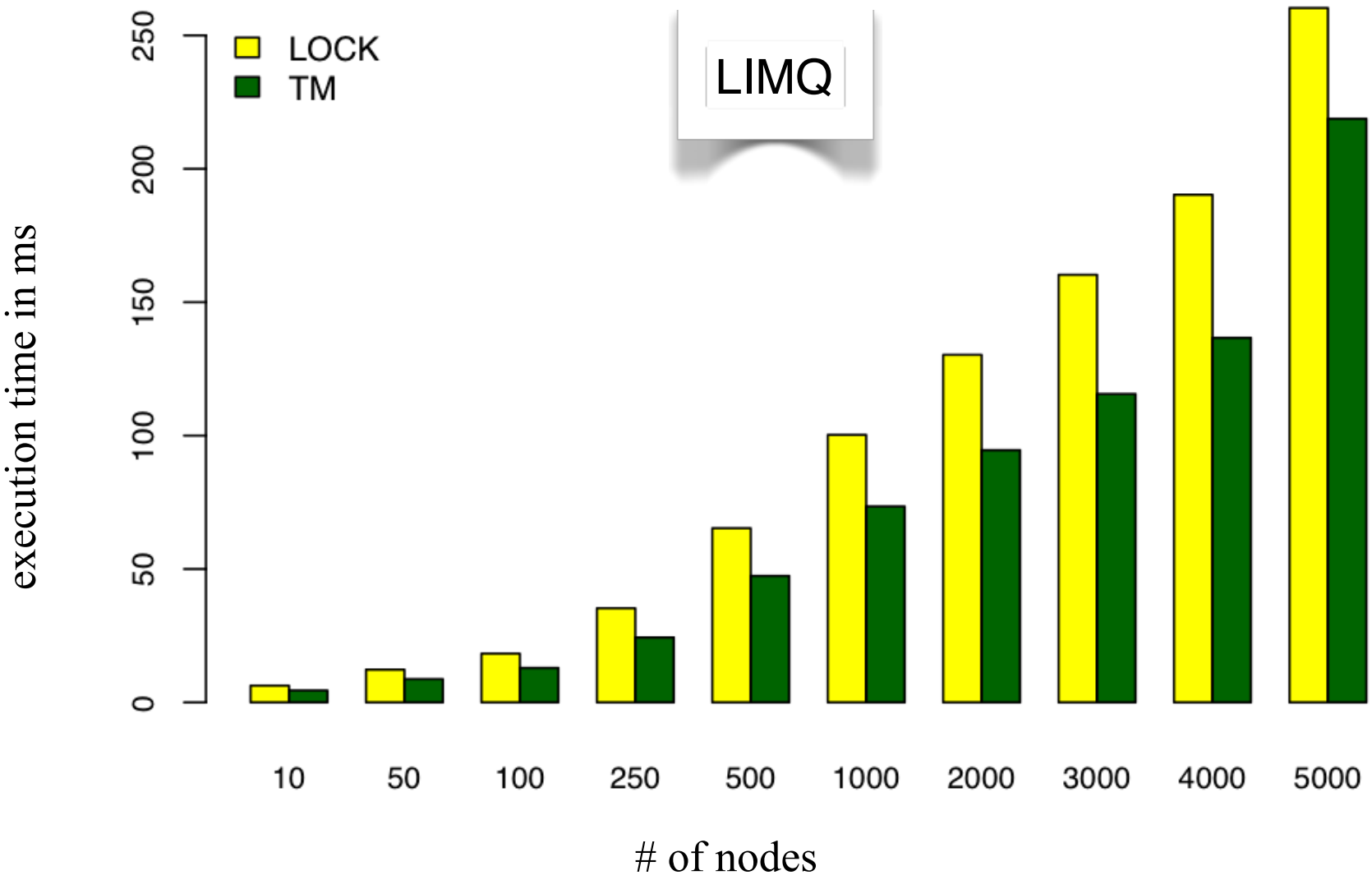
Running on JOP with 4 cores, each JOP core has a local 4 KB instruction cache and 1 KB stack cache. Using Altera DE2-70 Development board with Cyclone II EP2C70 FPGA.

Evaluation - Doubly Linked Queue



Running on JOP with 4 cores, each JOP core has a local 4 KB instruction cache and 1 KB stack cache. Using Altera DE2-70 Development board with Cyclone II EP2C70 FPGA.

Evaluation - Limited Capacity Queue

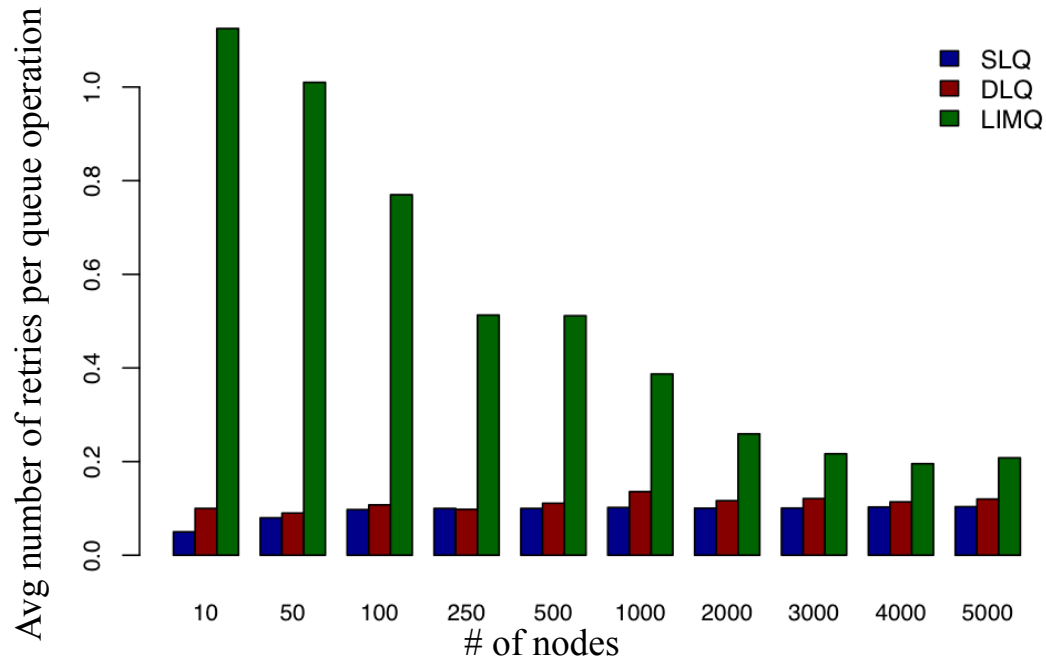


Running on JOP with 4 cores, each JOP core has a local 4 KB instruction cache and 1 KB stack cache. Using Altera DE2-70 Development board with Cyclone II EP2C70 FPGA.

Evaluation - Set Sizes and Retries

		Read Set	Write Set	Read-Write Set
TM	SLQ	7	2	7
	DLQ	11	4	12
	LIMQ	11	5	12
CAS_TM	SLQ	3	1	3
	DLQ	3	1	3

- Larger sets lead to more retires (higher chance of collision)
- Retries can also happen due to contention on a single location (ex Size in LIMQ)



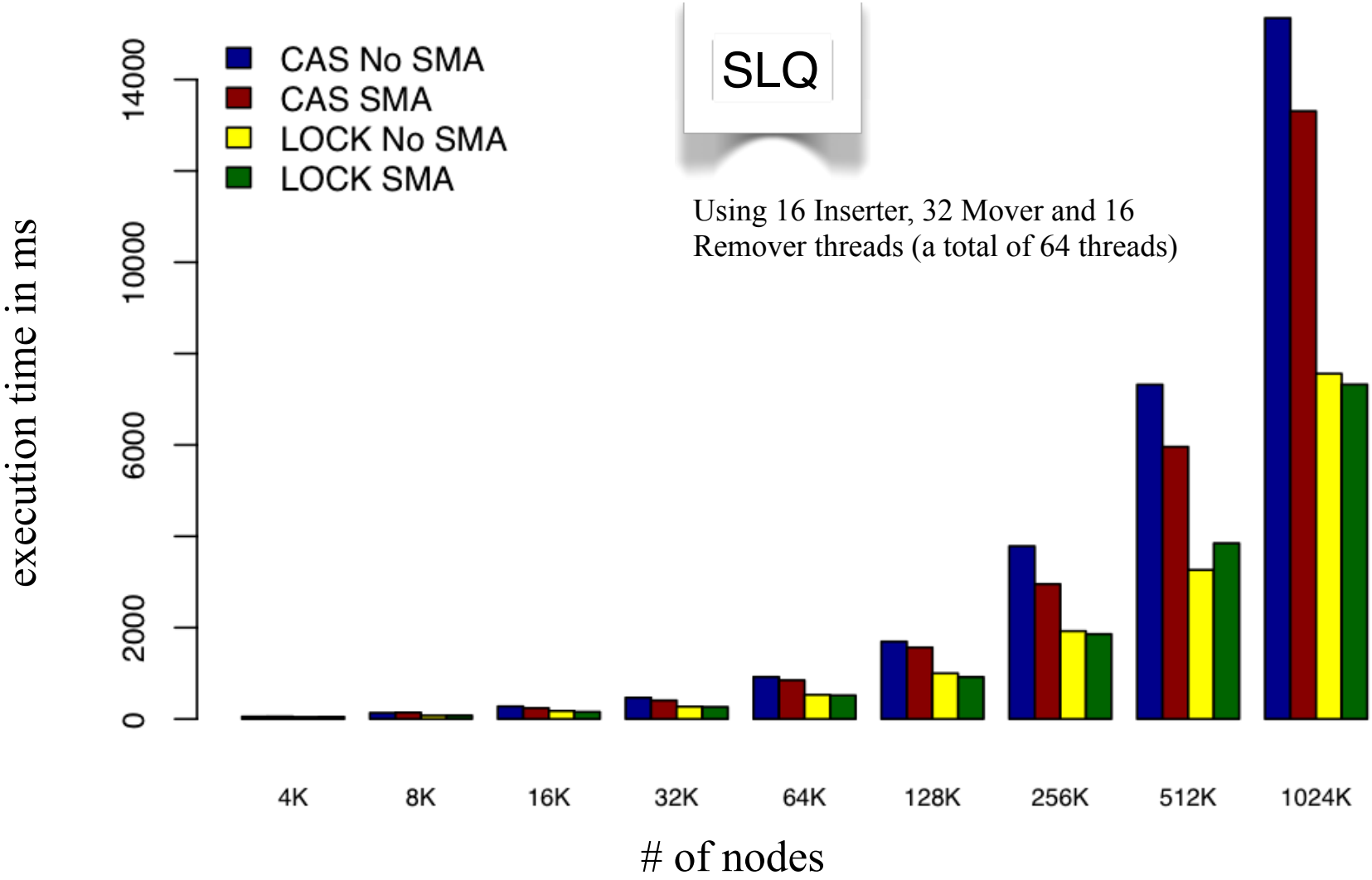
Running on JOP with 4 cores, each JOP core has a local 4 KB instruction cache and 1 KB stack cache. Using Altera DE2-70 Development board with Cyclone II EP2C70 FPGA.

What happens if we go beyond 4 cores?

Azul

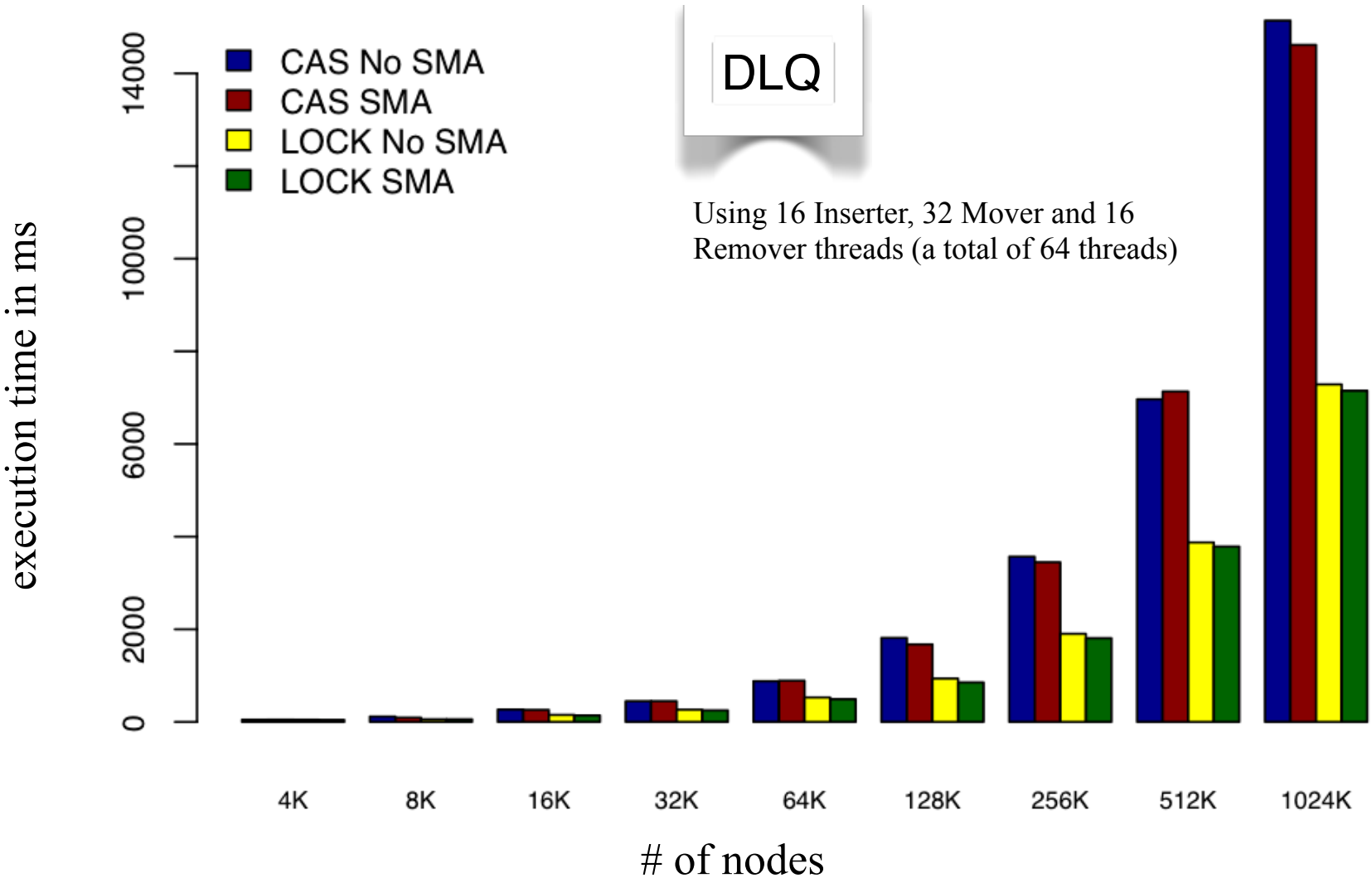
- Azul Vega 3 3310B, with two 54-core processors and 48GB of RAM
- Running on top of the Azul Virtual Machine with the Concurrent Pauseless GC
- Running using Speculative Multi-address Atomicity (SMA) that attempts to run *"synchronized"* blocks transactionally
- Running using from 4 up to 128 threads with similar results
- Aiming to test scalability
- We do not have a measure of commits/retries.

Azul Evaluation - SLQ



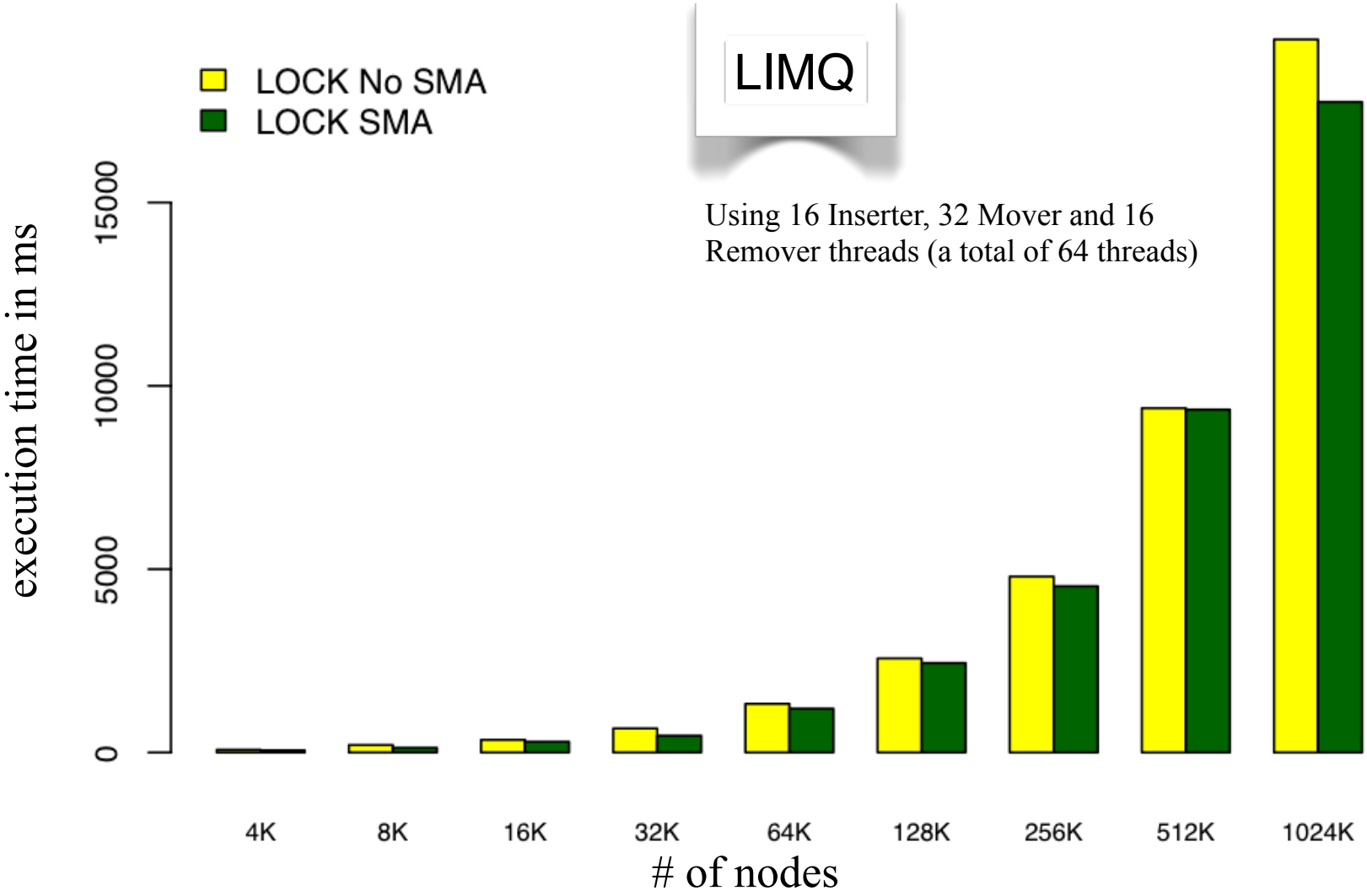
Running on an Azul Vega 3 3310B, with two 54-core processors and 48GB of RAM, on top of the Azul Virtual Machine with the Concurrent Pauseless GC.

Azul Evaluation - DLQ



Running on an Azul Vega 3 3310B, with two 54-core processors and 48GB of RAM, on top of the Azul Virtual Machine with the Concurrent Pauseless GC.

Azul Evaluation - LIMQ



Running on an Azul Vega 3 3310B, with two 54-core processors and 48GB of RAM, on top of the Azul Virtual Machine with the Concurrent Pauseless GC.

Can we bound the number of retries?

Worst Case Execution Time Analysis

- From Schoeberl, Brandner, and Vitek in SAC'10: number of retries r is bounded to n − 1 on a n core multiprocessor
- Assuming periodic threads, non-overlapping periods and execution deadline not exceeding the period then:

$$t_{wcet} = t_{na} + (r+1)t_{amax} \quad (1)$$

- Substituting the number of retries with number of cores and splitting the non-atomic time into within the operations (naimax) and outside (nae) we get.

$$t_{na} = t_{naimax} + t_{nae} \quad (4)$$

$$t_{wcet} = t_{nae} + (m)(t_{naimax} + (n \times t_{amax})) \quad (5)$$

WCET(cont'd)

- JOP Analysis framework provides an accurate cycle count for each of the operations

put_tr	put	get_tr	get
315	807	306	475

Number of Cycles required for the different methods (assuming no-retries)

- These numbers can be used to calculate a *lower-bound* on the external non-atomic section, given the number of operations (atomic sections) per thread.

Conclusion / Future Work

Results

- Transactional memory is an interesting alternative to traditional concurrency control mechanisms

Future Work

- Larger data-structures including HashTables, Double-Ended Queues and Graph Structures
- TM aware scheduler that would allow tighter bounds on Real-Time based on the WCET analysis