PERFORMANCE ANALYSIS AND OPTIMIZATION OF SKIP LISTS FOR MODERN MULTI-CORE ARCHITECTURES

Anish Athalye and Patrick Long

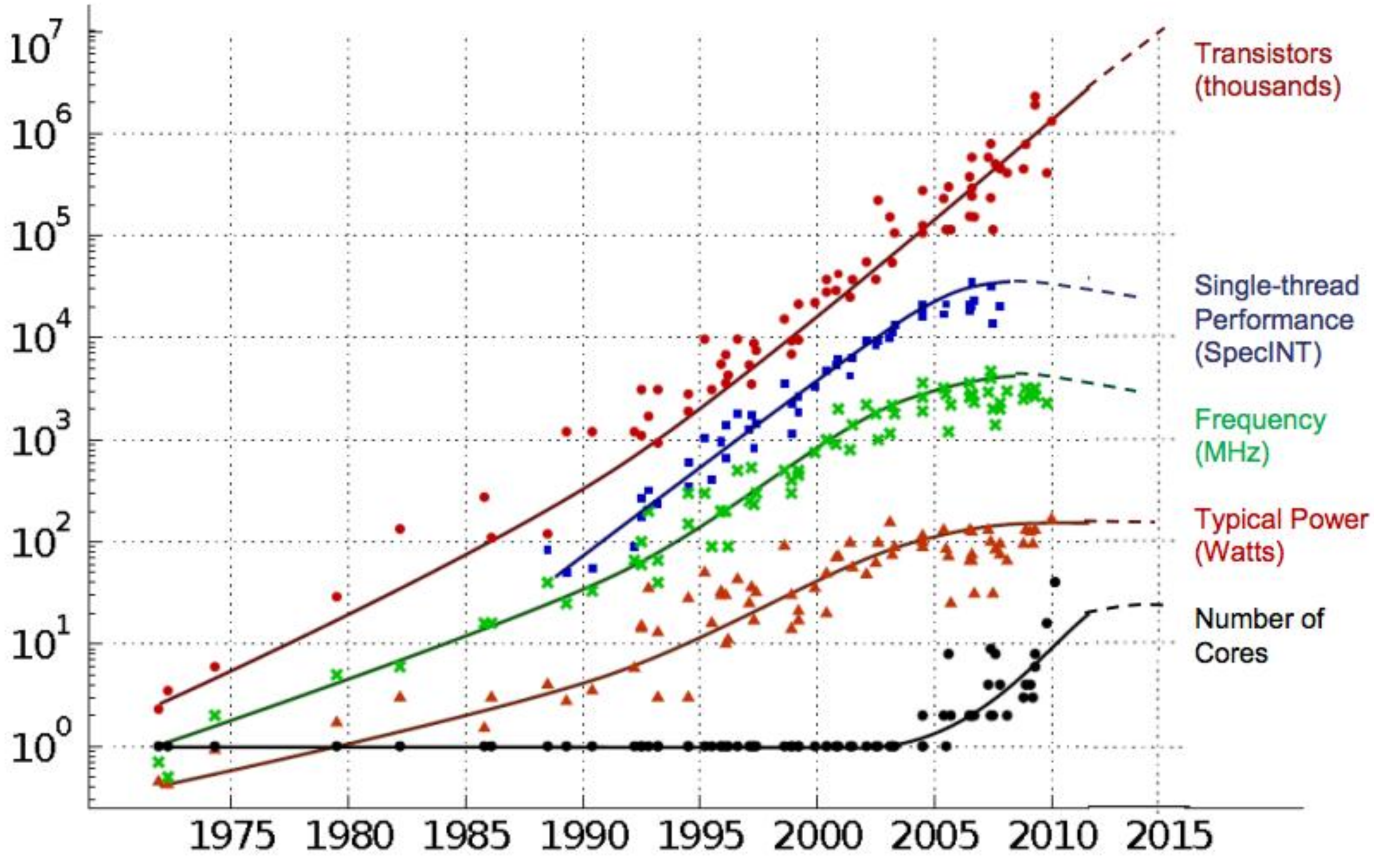
Mentors: Austin Clements and Stephen Tu

3rd annual MIT PRIMES Conference

Sequential Origins

- Research on data structures produced several fast *sequential* designs
  - Not designed for concurrent access
- In the 90's, many extended to support concurrent operations
- Multi-core processors changing
  - Exponential growth in # of cores
  - New architectures

Parallelism is the future



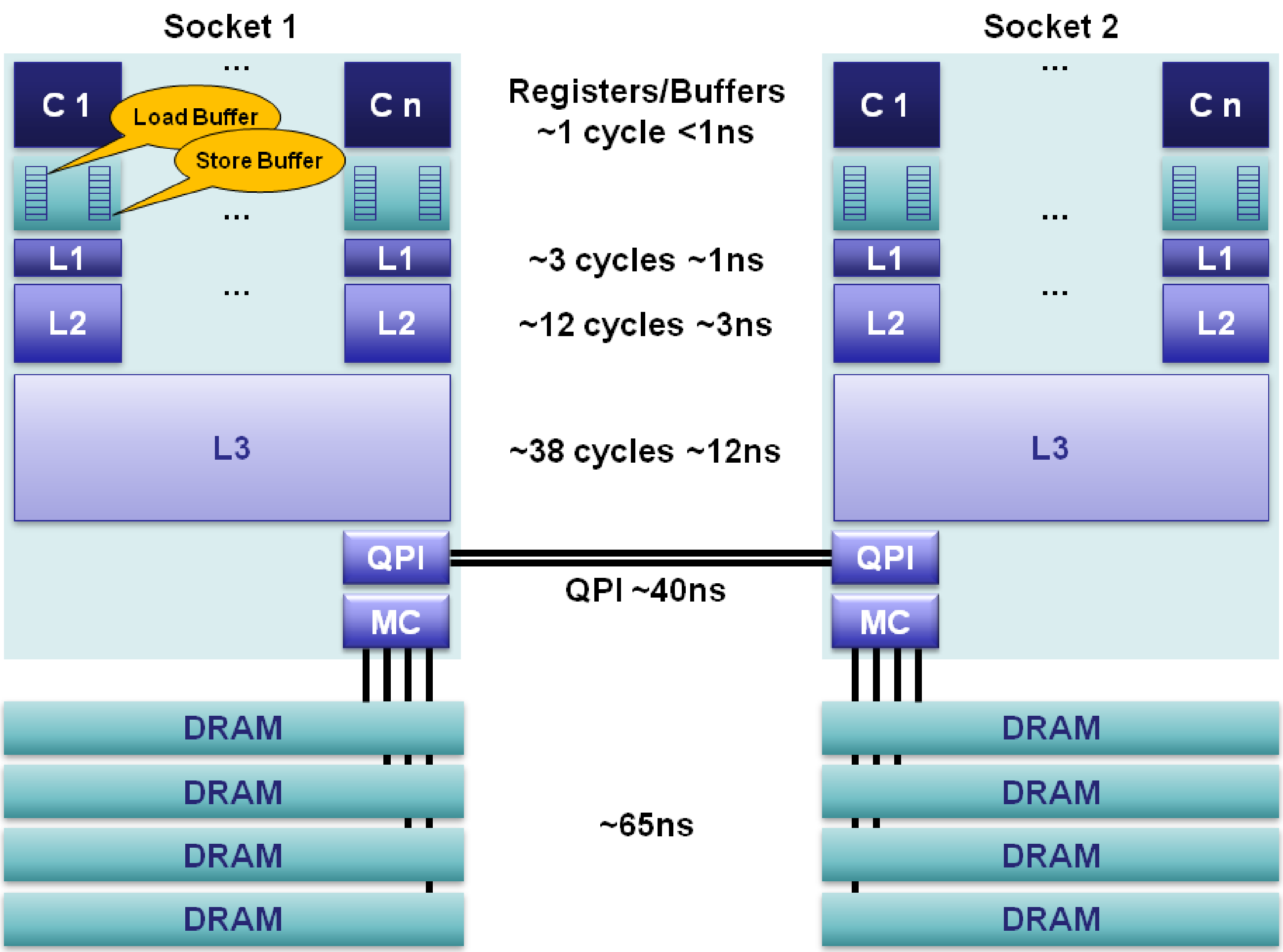
Source: Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Contributions

- Focused on the skip list
- Techniques to improve scalability
- Basic analytical model of scalability performance
- Analyses of implementations in Java and C++

MODERN MULTI-CORES

Cache Hierarchy



Source: http://mechanical-sympathy.blogspot.co.uk/2013/02/cpu-cache-flushing-fallacy.html

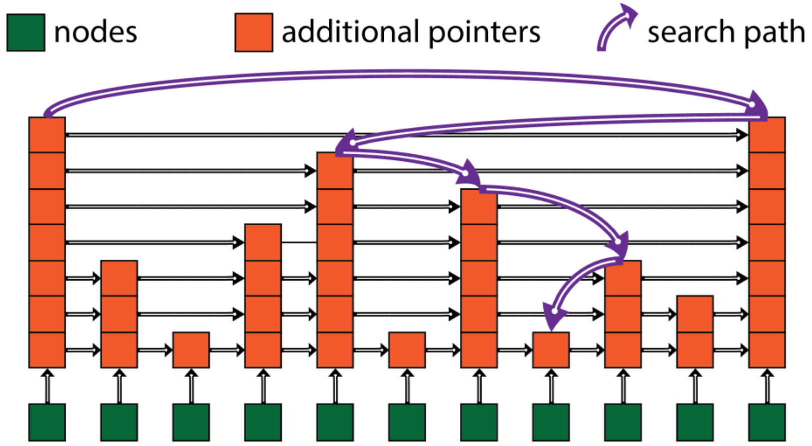
Cache Coherence and Ownership

- Concurrent access and mutation presents a challenge
- Cache coherence protocol
  - Cores must take ownership to mutate data
    - Concurrent writes cause contention
  - Cores must have up-to-date copy to read data
    - Requires cache line transfer if data modified
- Cache contention creates bottlenecks

SKIP LISTS

Skip Lists

- Probabilistic data structure implementing an ordered set
  - 3 operations: insert, delete, lookup
  - Average case complexity O(log n)
- Stores a sorted list
- Hierarchy of linked lists connect increasingly sparse subsequences of elements
  - Randomized with geometric distribution for O(log n) performance
  - Auxiliary lists allow for efficient operation
  - No global updates (rebalancing, etc.) due to probabilistic nature
- Can be efficiently parallelized



Skip Lists

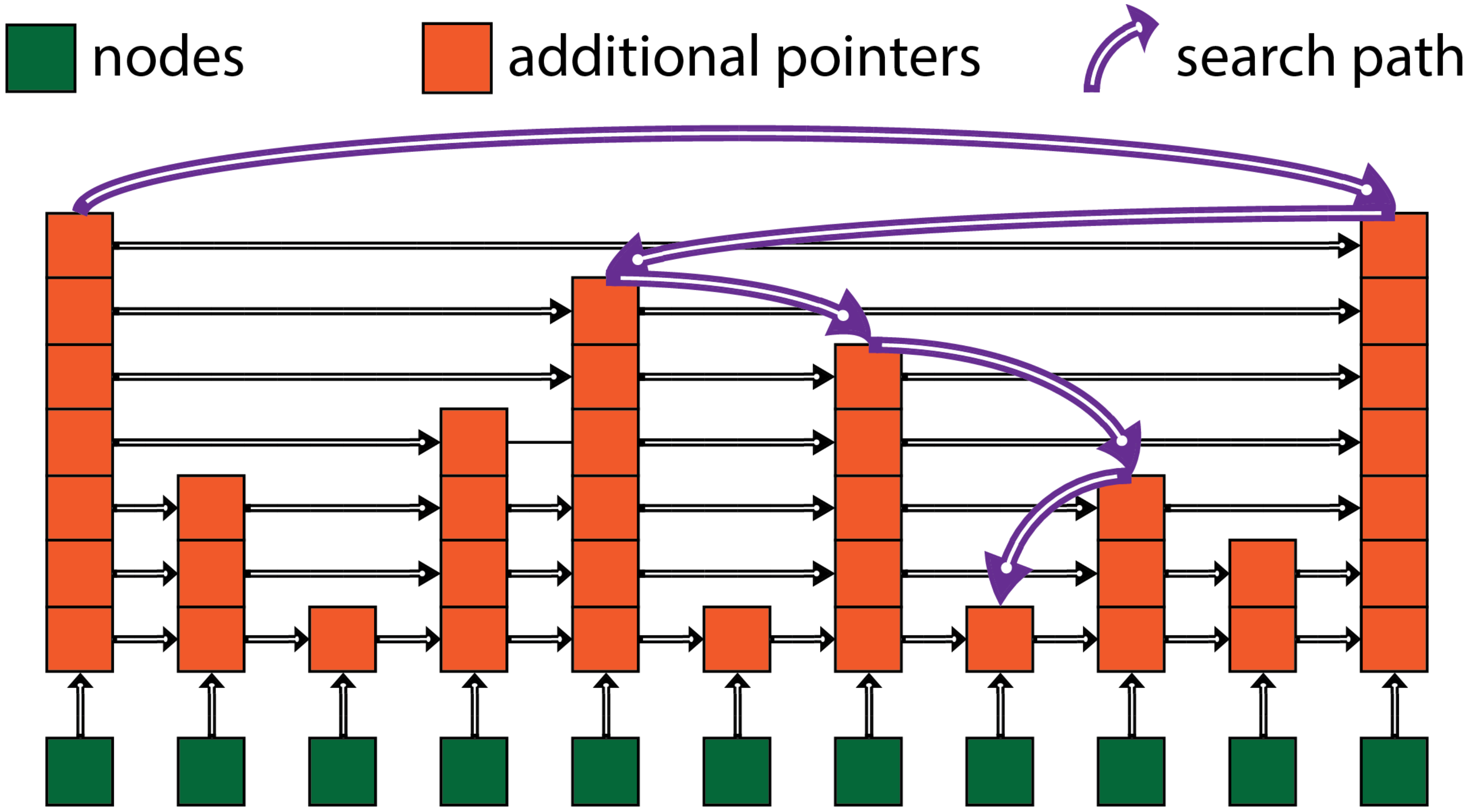


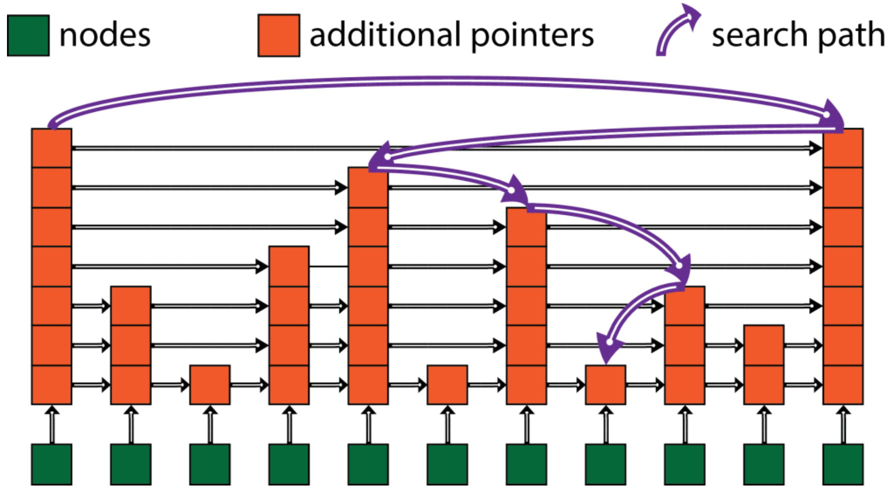
Diagram of a Skip List

Skip List Algorithm

- Based on the lock-based concurrent skip list in *The Art of Multiprocessor Programming* by Herlihy and Shavit
- Fine-grained locking for insertion and deletion
- Wait-free lookup
- Locks ensure skip list property is maintained
  - Higher level lists are always contained in lower level lists

Skip List Algorithm

- Lookup
  - Similar to binary search
- Insertion
  - Find new node’s predecessors and successors
  - Lock predecessors
  - Validate that links are correct
  - Link new node to predecessors and successors



Preliminary Model

- Basic model to analyze scalability by estimating cache coherence traffic
  - Ignored higher levels of linked lists
  - Treated lookups as instantaneous operations
  - Assumed threads inserted nodes simultaneously
- Measured expected cache line transfers as thread count and size varied
- Modeled as balls and bins problem

Preliminary Model

- ***k*** threads, ***n*** element skip list
- Find expected value of $\sum_{i=1}^{n} f(i)$

$$f(i) = \begin{cases} 0, & \text{bucket } i \text{ has fewer than 2 balls} \\ x-1, & \text{bucket } i \text{ has } x \text{ balls} \end{cases}$$

- Explicit formula for expected cache line transfers ***t***

$$t(n,k) = k - n + \sum_{i=1}^{n} \frac{S(k, n-i)(n-i)!\binom{n}{i} i}{n^k}$$

$$S(n,k) = \frac{1}{k!} \sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i} i^n$$

- Verified formula using Monte Carlo simulation

IMPLEMENTATION AND OPTIMIZATION

Implementation and Optimization: C++

- Custom read-copy update (RCU) garbage collector
  - Avoids read-write conflicts
- Padded relevant data structures to a cache line
  - Avoids false sharing

Implementation and Optimization: Java

- Simplified contains method implementation
  - Avoided keeping track of predecessors and successors
- Avoiding generics/autoboxing
- Pre-allocated arrays internal to operations
- Tried a bunch of hacks to increase the scalability of the Java implementation
  - Nothing improved scalability

PERFORMANCE

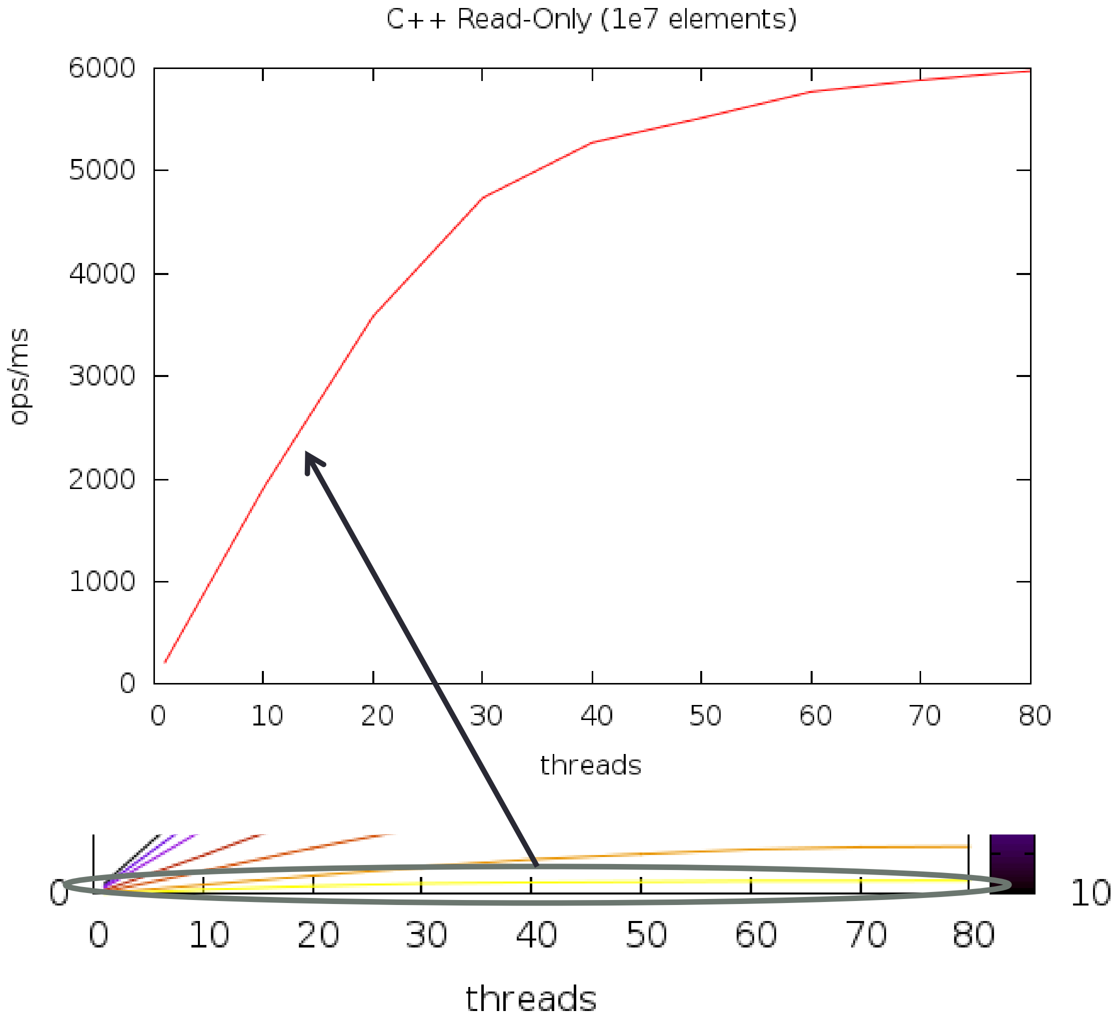
Experimental Setup

- 80-core machine
  - 8 x 10-core Intel Xeon @ 2.40 GHz
  - 256 GB of RAM
  - Linux
- C++ with tcmalloc or jemalloc
- Java with HotSpot and OpenJDK VM

Procedure

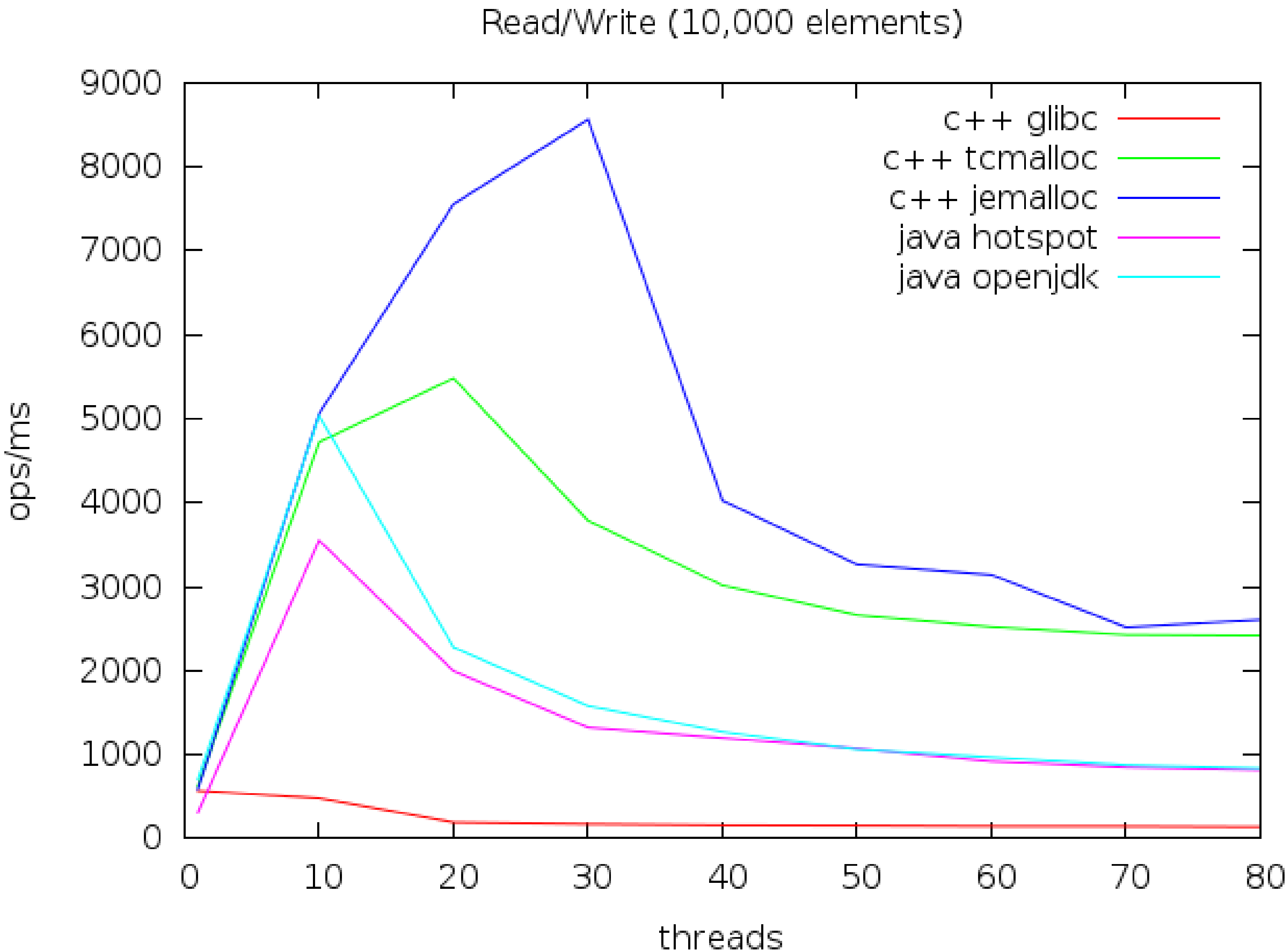
- Varied thread count and size
- Fixed size skip list with uniformly distributed key space
- Read-only benchmark
  - Threads concurrently search for random elements
- Read-write benchmark
  - Threads concurrently add and then remove elements
  - Size was maintained within a constant factor
- Measured total throughput
- Warmed up the JVM before performing tests

Read-only scalability



Java scaled similarly

Read-write scalability



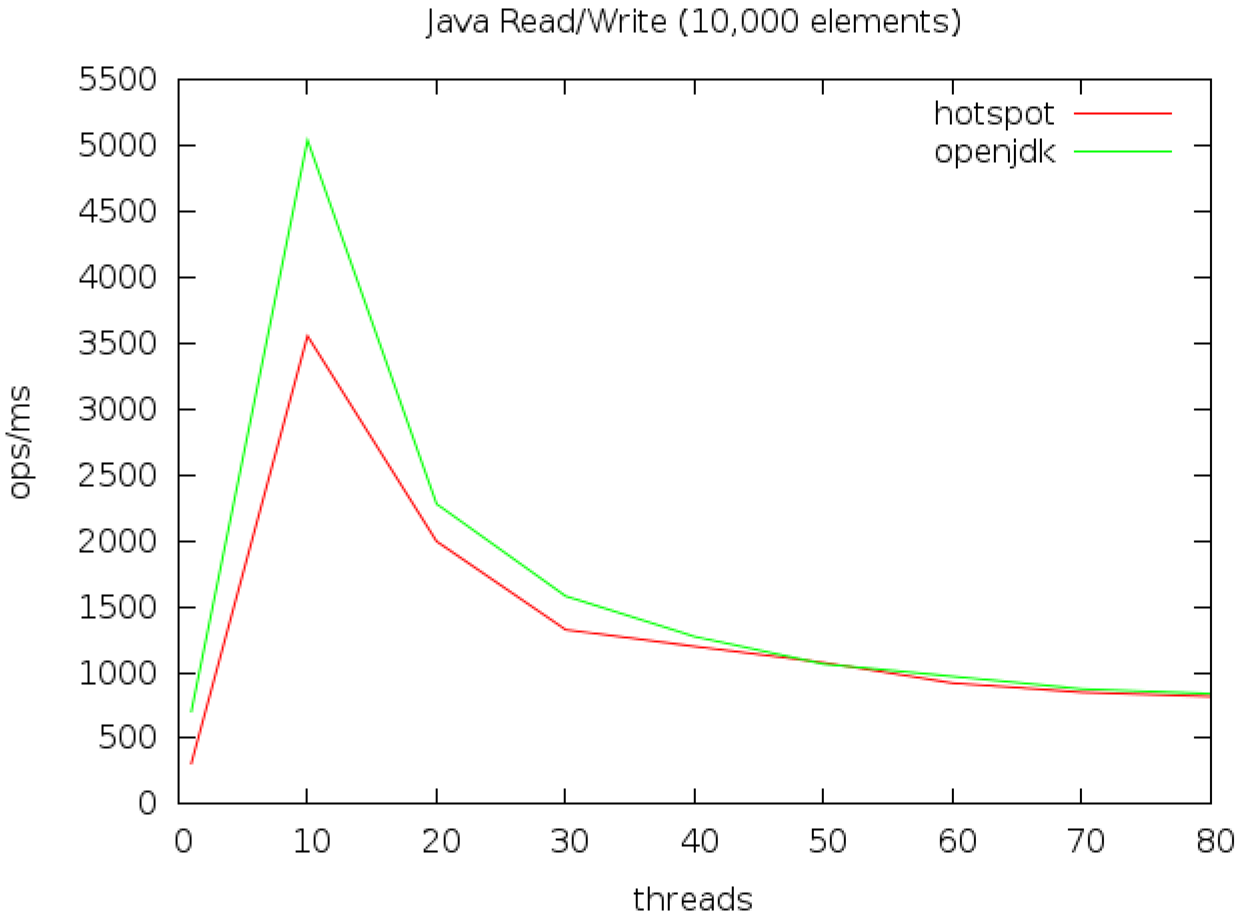
Read-write scalability

- Read-write benchmarks did not scale linearly
- Skip list reached maximal throughput between 10 and 40 cores, depending on the implementation
- Two reasons for drop-off in performance
  - Lock contention between threads, especially in small (≤ 1000 element) skip lists
    - As thread count increases, lock contention increases and relative performance decreases
  - As thread count increases, relative memory allocator performance decreases

Java vs. C++

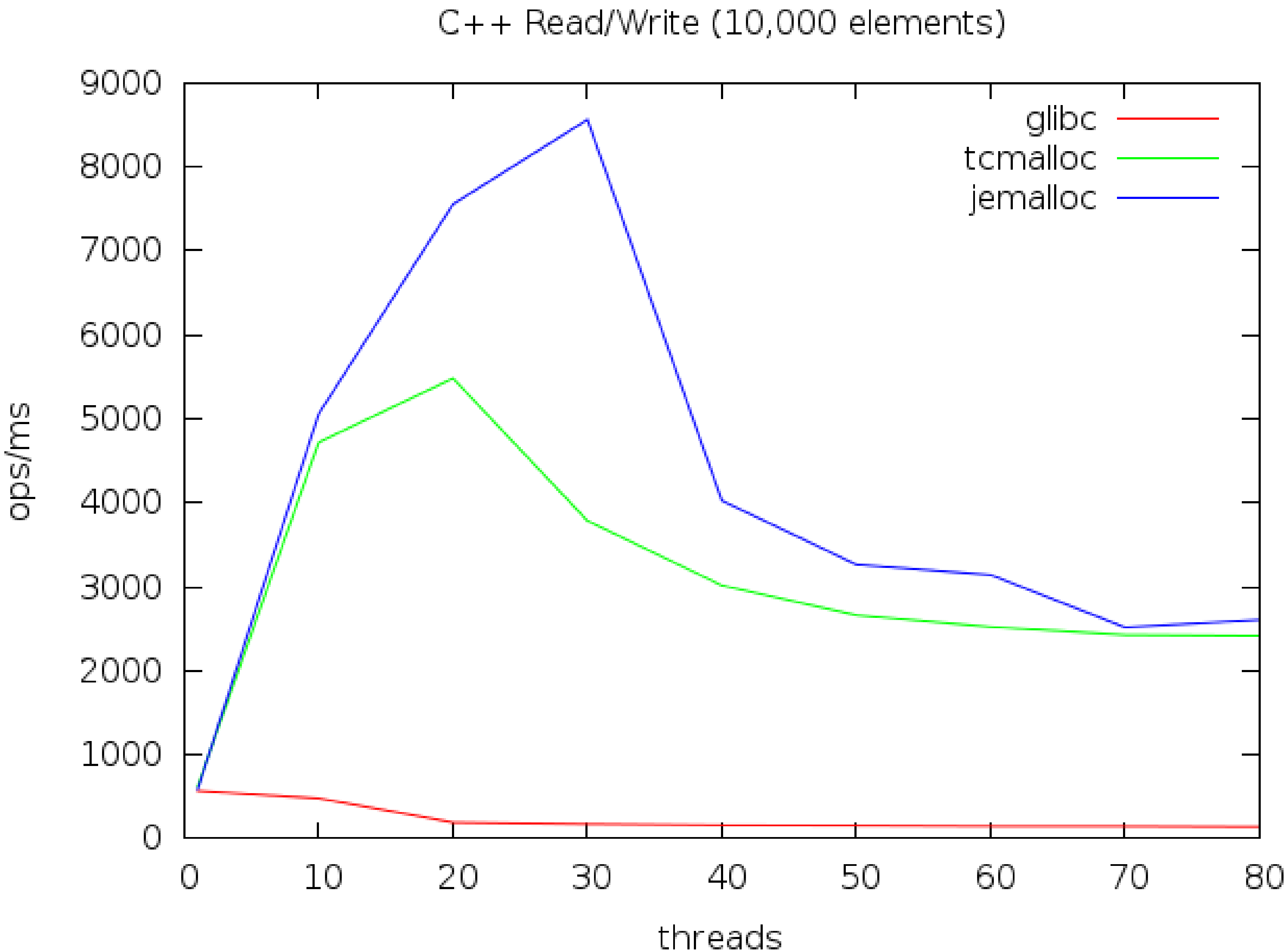
- C++ implementation faster than Java implementation
- Memory deallocation
  - Especially difficult in the multithreaded case
  - Java has GC, C++ requires manual memory management
- Java’s compacting GC speeds up memory allocation
  - Keeps heap unfragmented
  - Increment pointer, return old value
  - Minimal synchronization in multithreaded case using TLABs
- C++’s default glibc memory allocator is bad

Java VMs



Java Read/Write (10,000 elements)
hotspot
openjdk
ops/ms
threads
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
10
20
30
40
50
60
70
80

C++ Memory Allocators



Conclusion

- Implemented and analyzed performance of concurrent skip lists
- All implementations scale well for read-only
- C++ glibc allocator doesn’t scale, alternatives scale better

Future Work

- Better model the performance of the skip list
  - Take into account search time, asynchronous reads and writes, and hierarchy of linked lists
  - More general model using queuing theory and Markov models
- Measure performance hit caused by lock contention
  - Custom memory allocator
  - Redesigned benchmark to avoid synchronization for memory allocation

Thanks

- Thanks to…
  - Austin Clements and Stephen Tu for excellent mentorship throughout the process
  - Professor Kaashoek and Professor Zeldovich for the project idea and mentorship
  - MIT PRIMES for providing us this great opportunity for research
  - Our parents for all their support