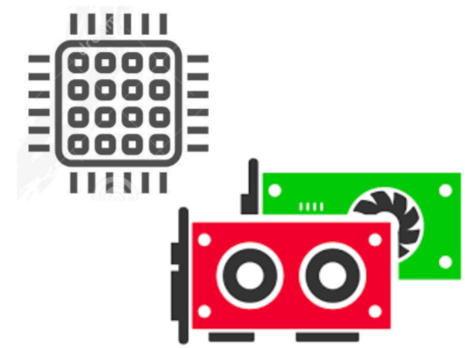
Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems

Kun Suo^, **Jia Rao**^, Hong Jiang^ and Witawas Srisa-an*

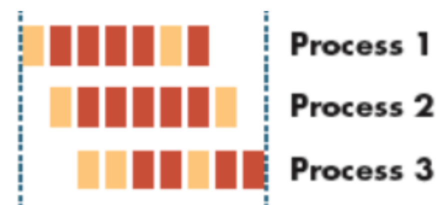
^ The University of Texas at Arlington

* The University of Nebraska at Lincoln

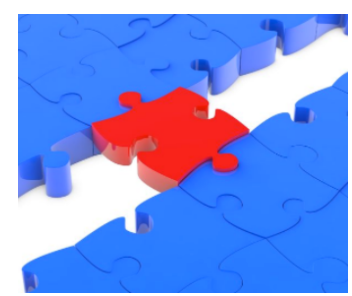
Exploiting Parallelism



- The rise of multicore architectures and other forms of hardware parallelism
  - Multi-core processors, accelerators, multi-queue devices, co-processors, etc.
- Exploiting parallelism in multicore systems
  - Application runtimes → **meet app-specific needs**
  - Operating systems → **balance load, preserve locality, save energy**
  - Synchronization primitives → **minimize overhead**



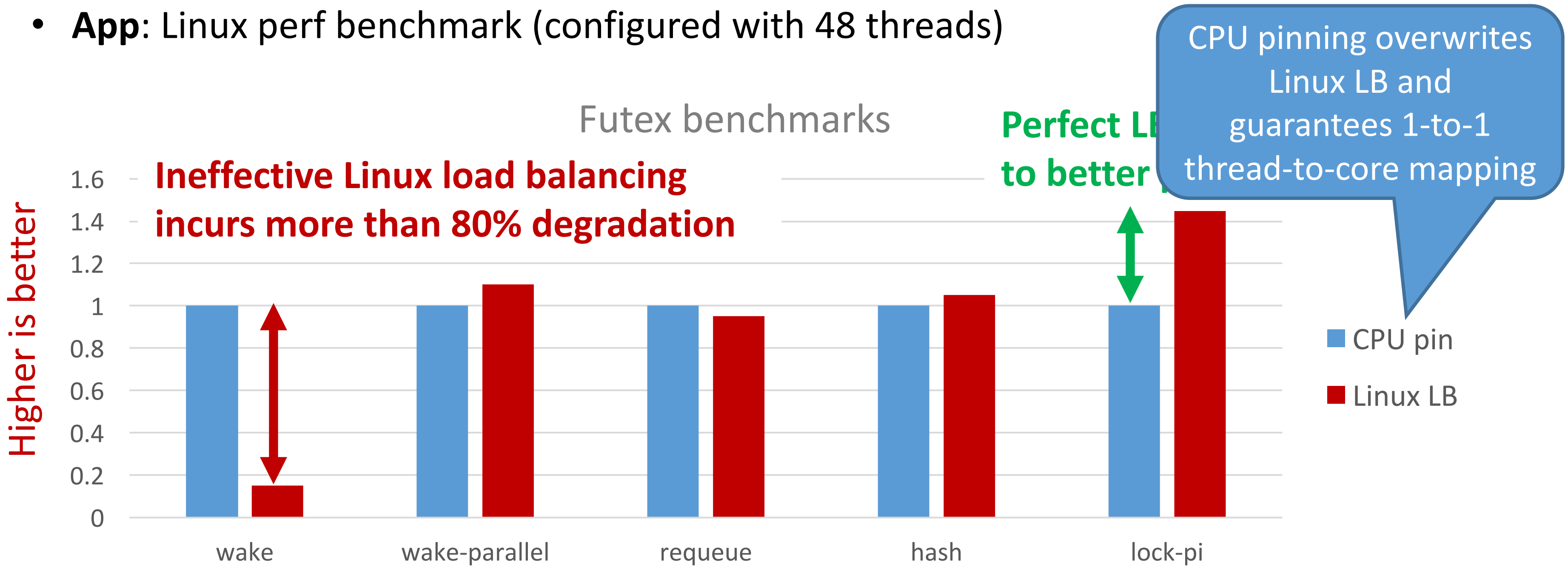
- Interplays among runtime, OS, and synchronization not well understood
  - **Runtime assumes guaranteed thread-level parallelism**
  - **OS schedules threads based on their CPU demands**



Semantic gap

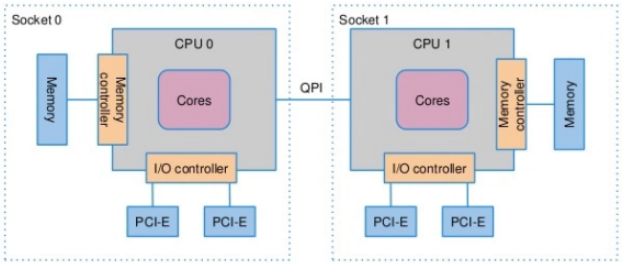
Intricate Program-OS Interaction

- **Hardware & OS**: 4-socket 12-core Intel Xeon E5-4640, 512GB memory, Linux 4.14
- **App**: Linux perf benchmark (configured with 48 threads)

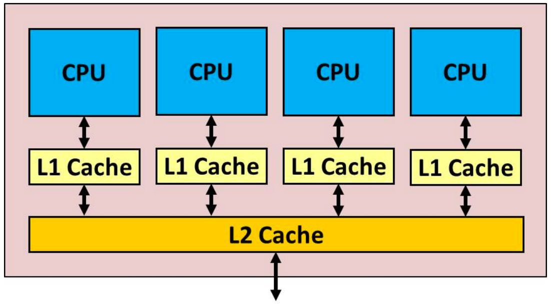


Reasons of Imperfect Balancing

- Only runnable/ready threads are eligible for load balancing
- OS may choose not to migrate threads



Data locality

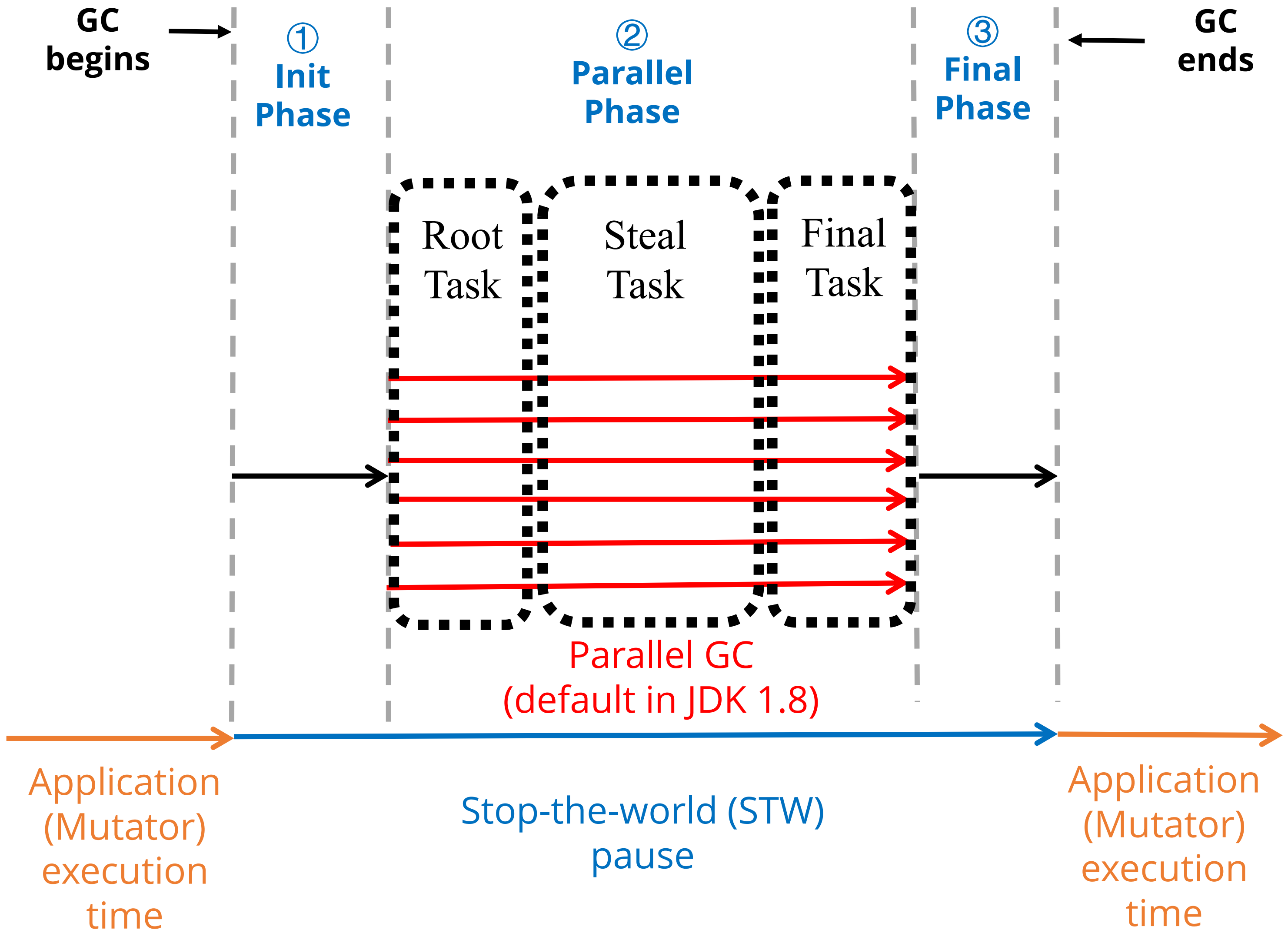


Cache hotness

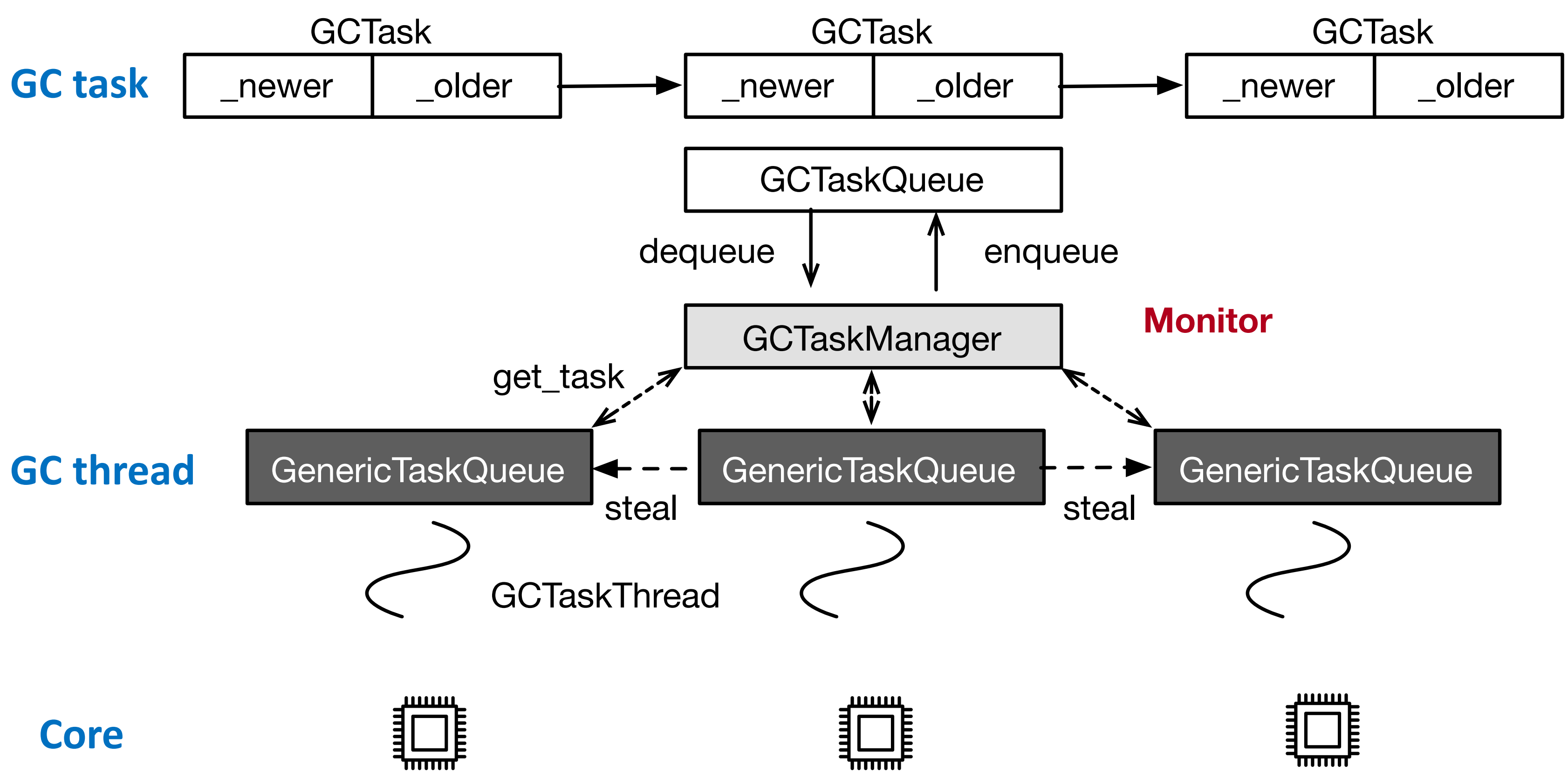
Energy saving

Harmful interactions between parallel programs and the OS scheduler

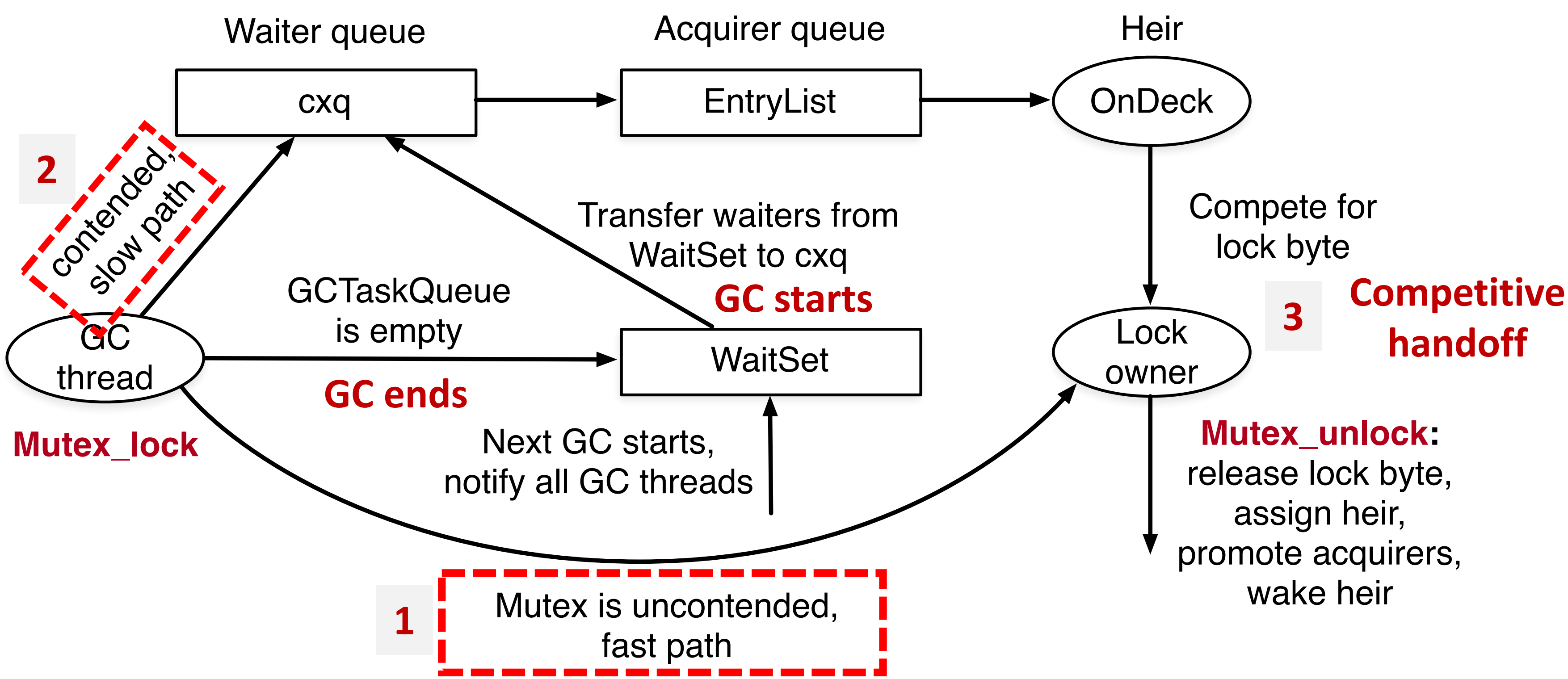
Parallel GC in HotSpot JVM



Assigning Tasks to GC Threads

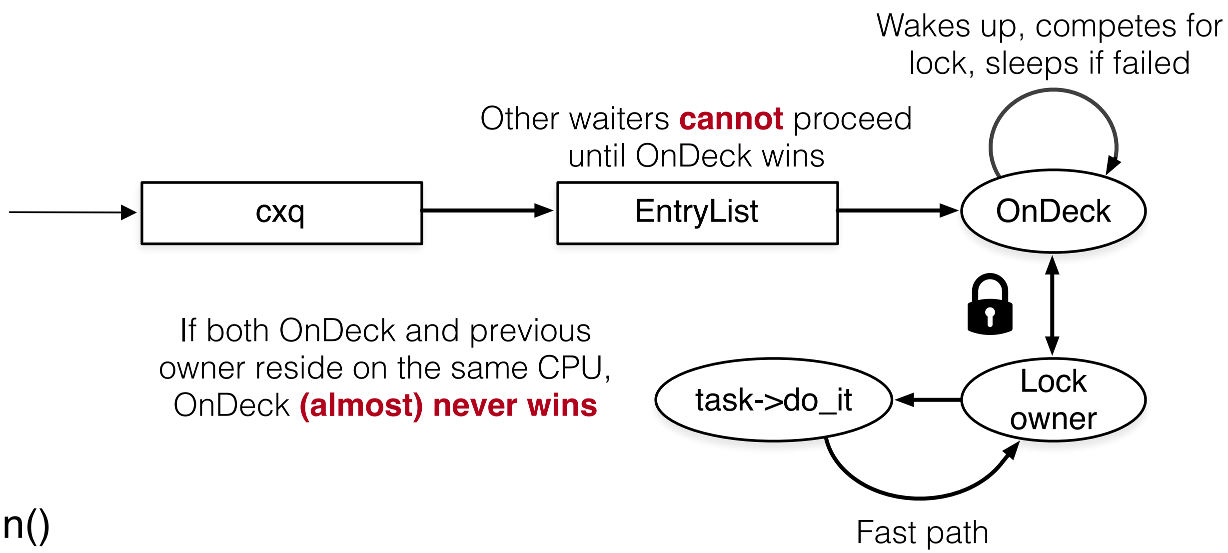


Native `mutex` Lock in HotSpot



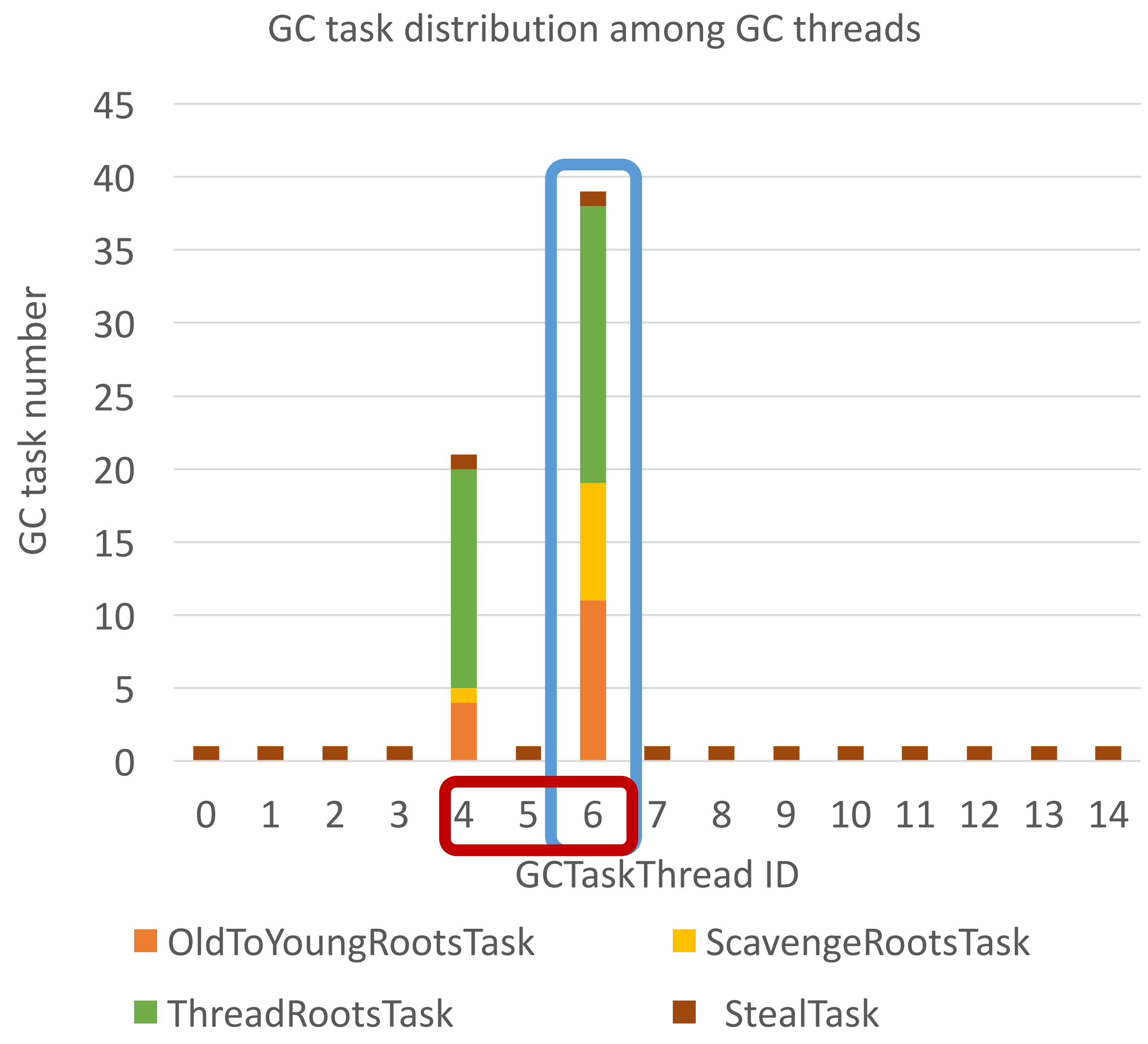
CPU Stacking and Unfair Locking

GC threads sleeping
on the condition variable

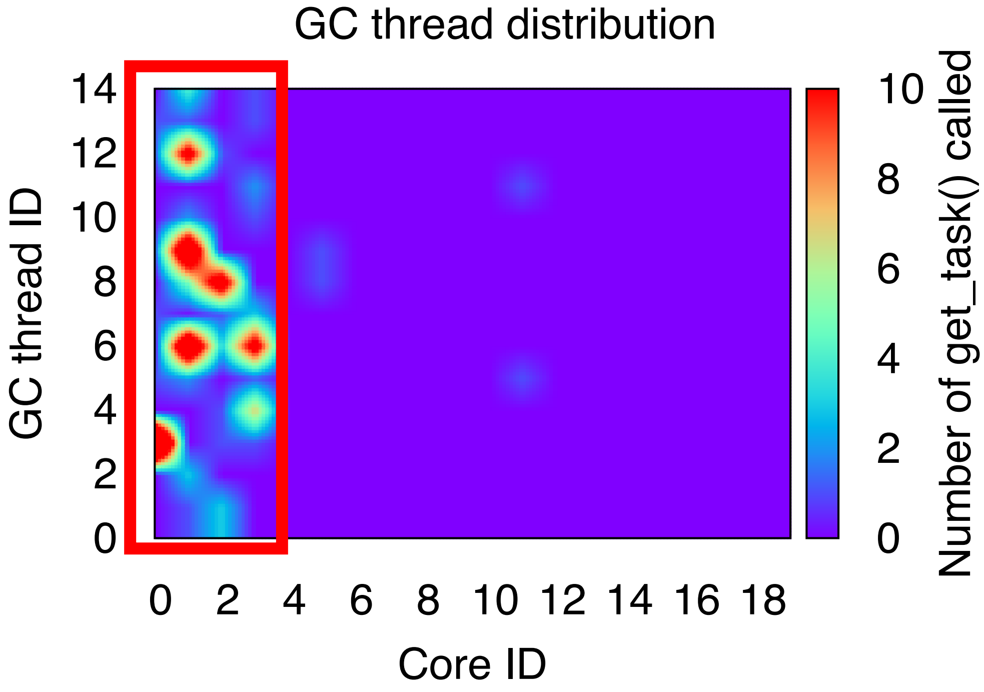



```
void GCTaskThread::run()
{
   for (;/* ever */;){
       GCTask* task = manager()->get_task();
       task->do_it();
   }
}
```

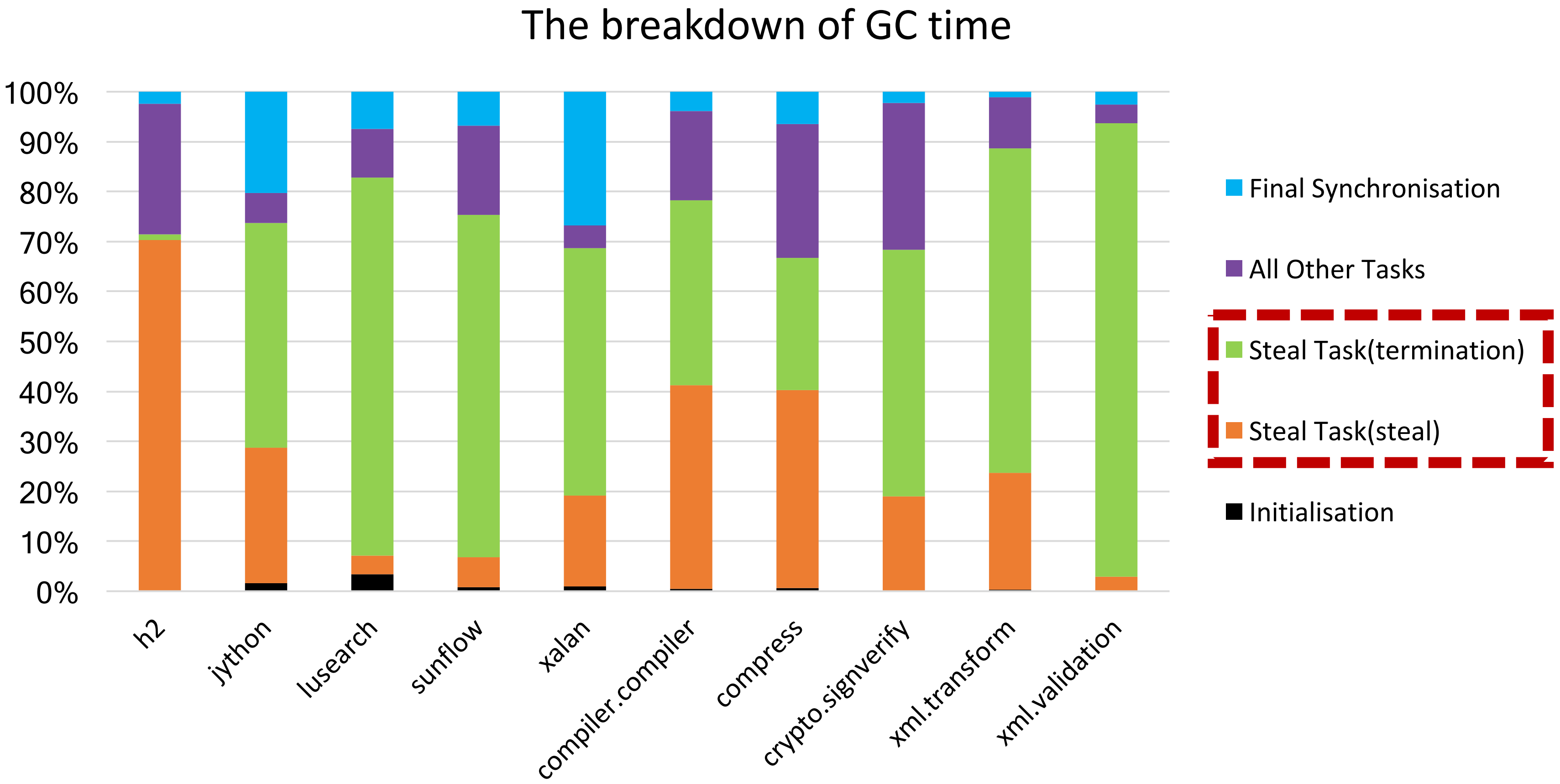
Loss of Concurrency



Loss of Concurrency



Inefficient Work Stealing



Why Work Stealing fails to Address the Imbalance?

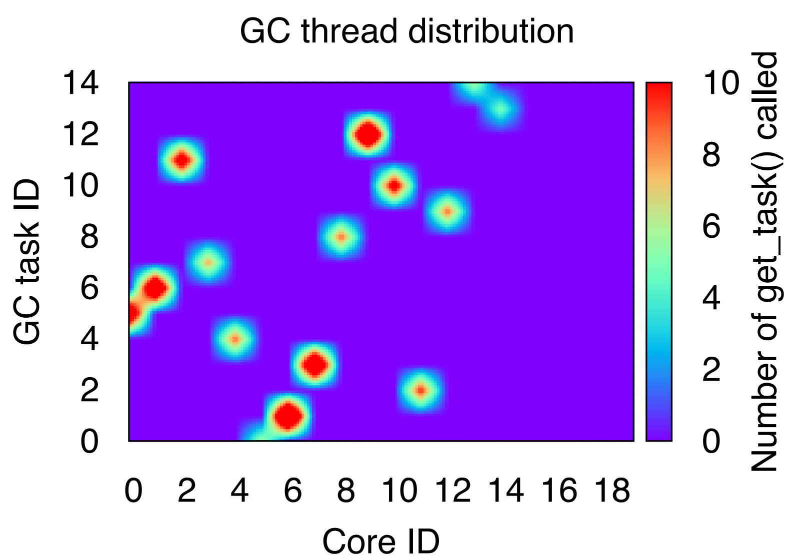
- HotSpot work stealing
  - Randomly pick up two GC threads and steal from the one with a longer queue
  - A GC thread enters a distributed termination protocol after 2*N failed steal attempts

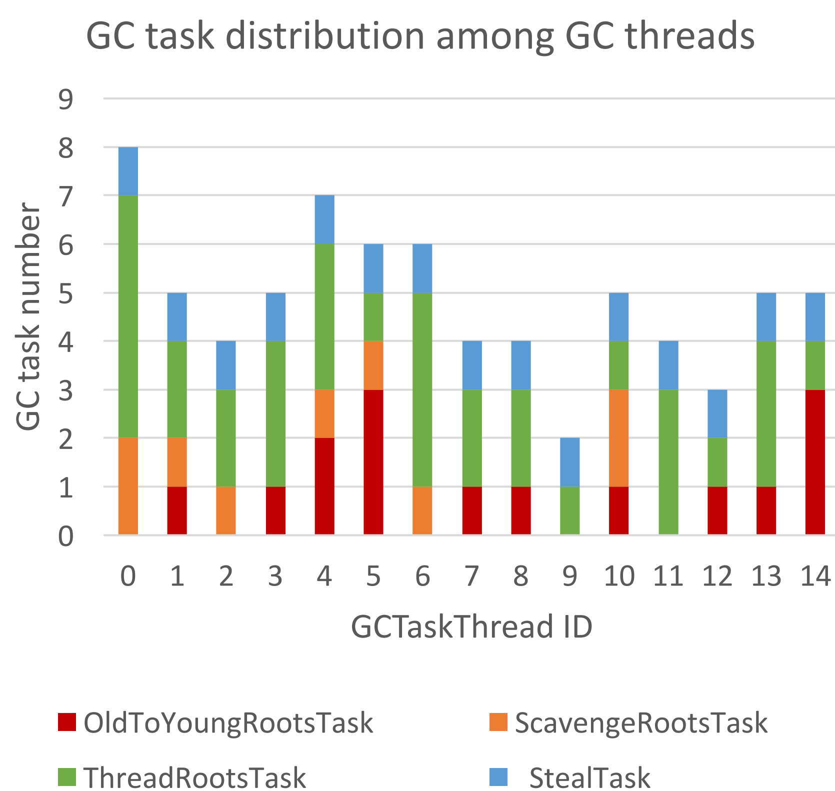
Two random choices stealing not effective if there is significant task imbalance among GC threads

Our Approaches

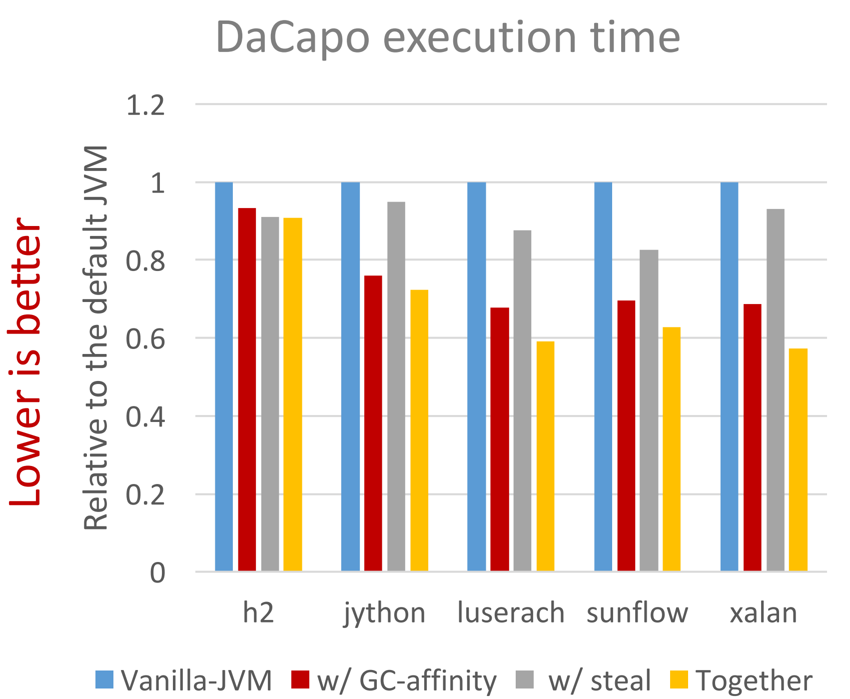
- GC thread affinity
  - Dynamically bind GC threads to separate cores, considering load
- Optimized work stealing
  - Semi-random stealing
  - Only steal from live threads, ~~2*N~~ $2*N_{live}$ *attempts*

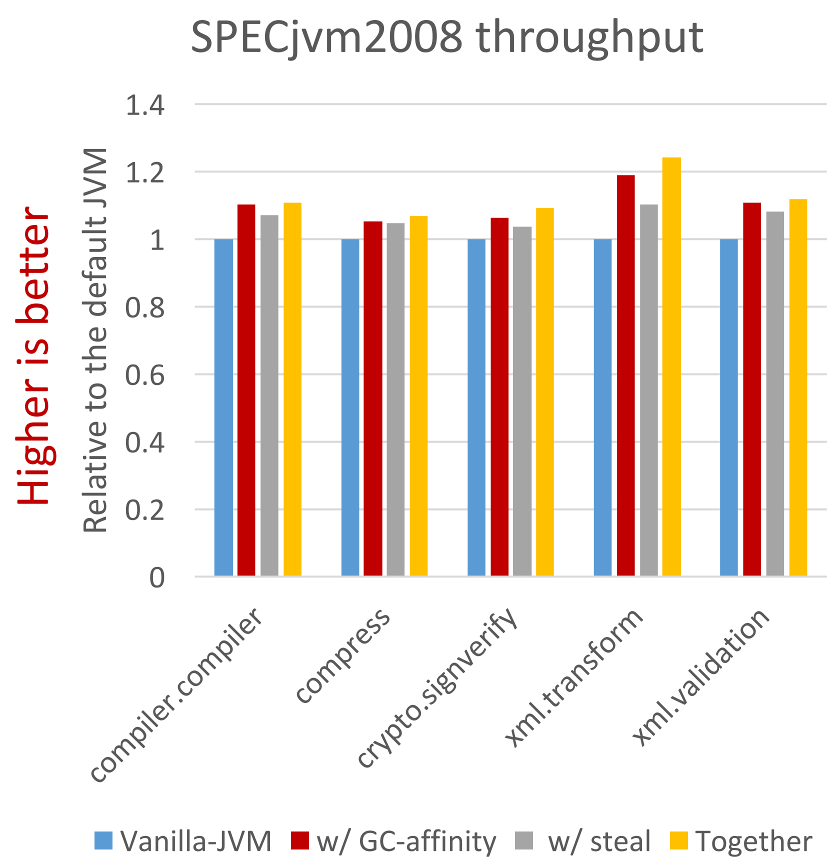
Mitigating the GC Imbalance





Improvement on Overall Performance



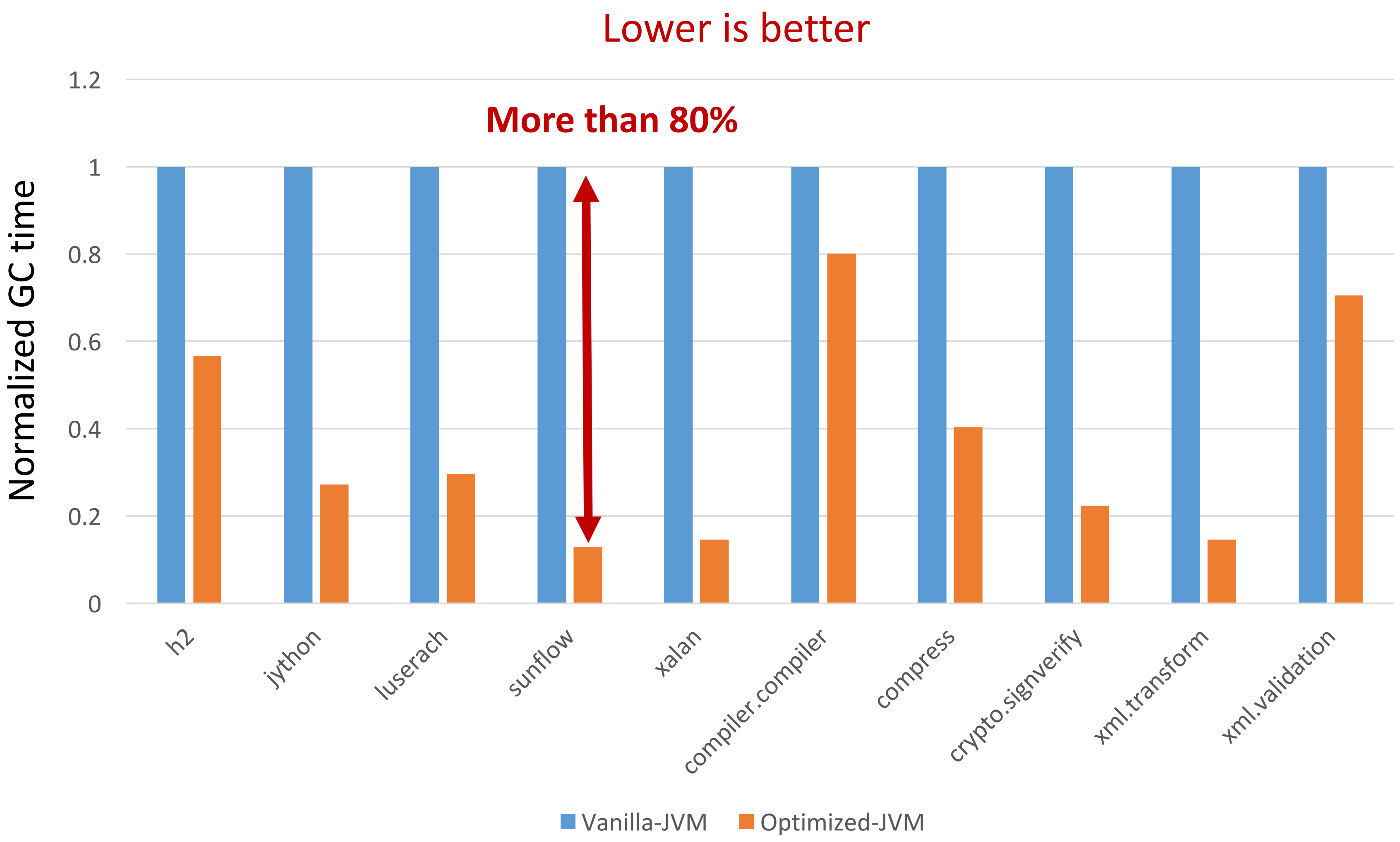


thread affinity **30.4%**

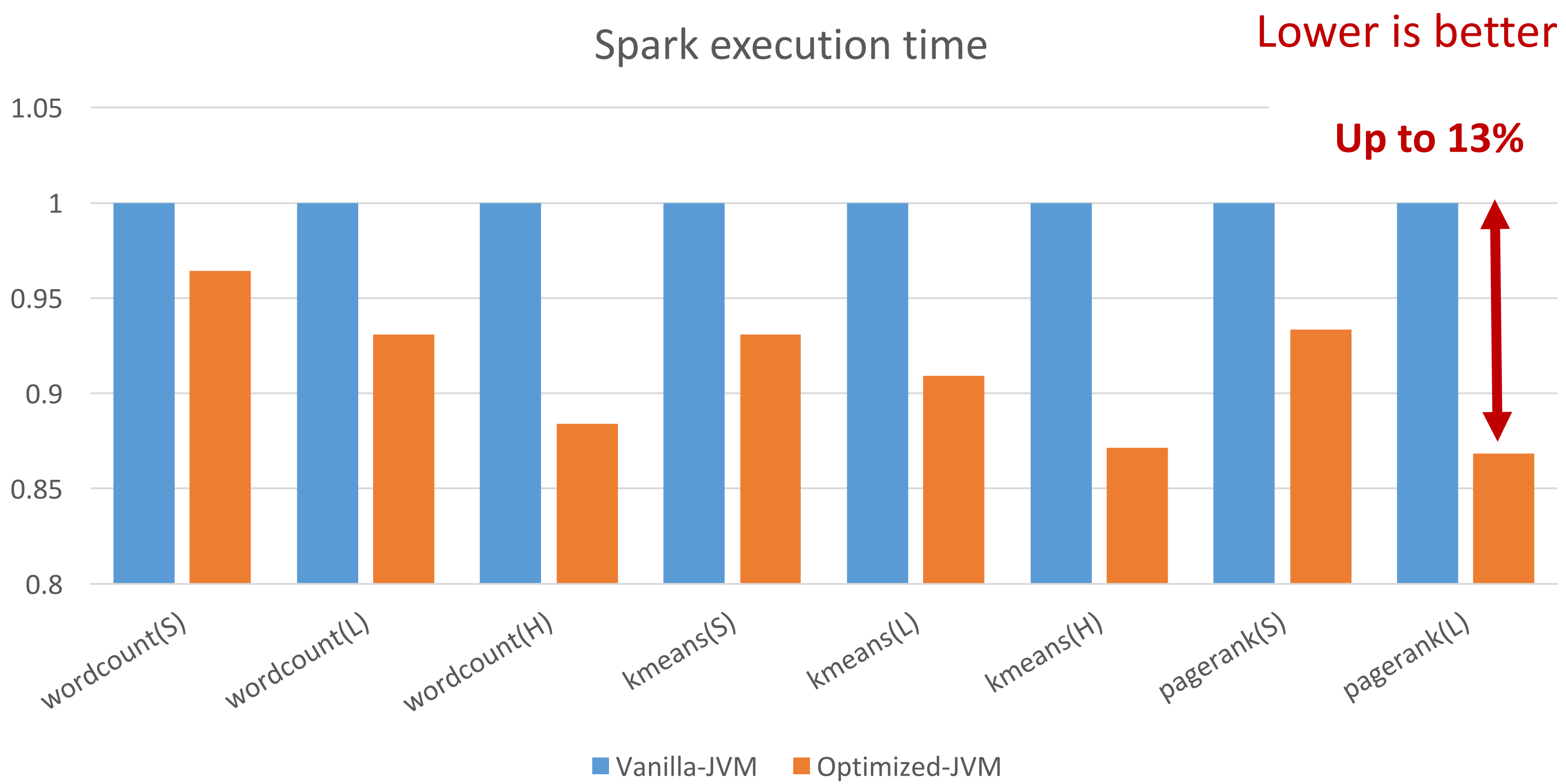
optimized stealing **17.5%**

combined **42.5%**

Improvement on GC time

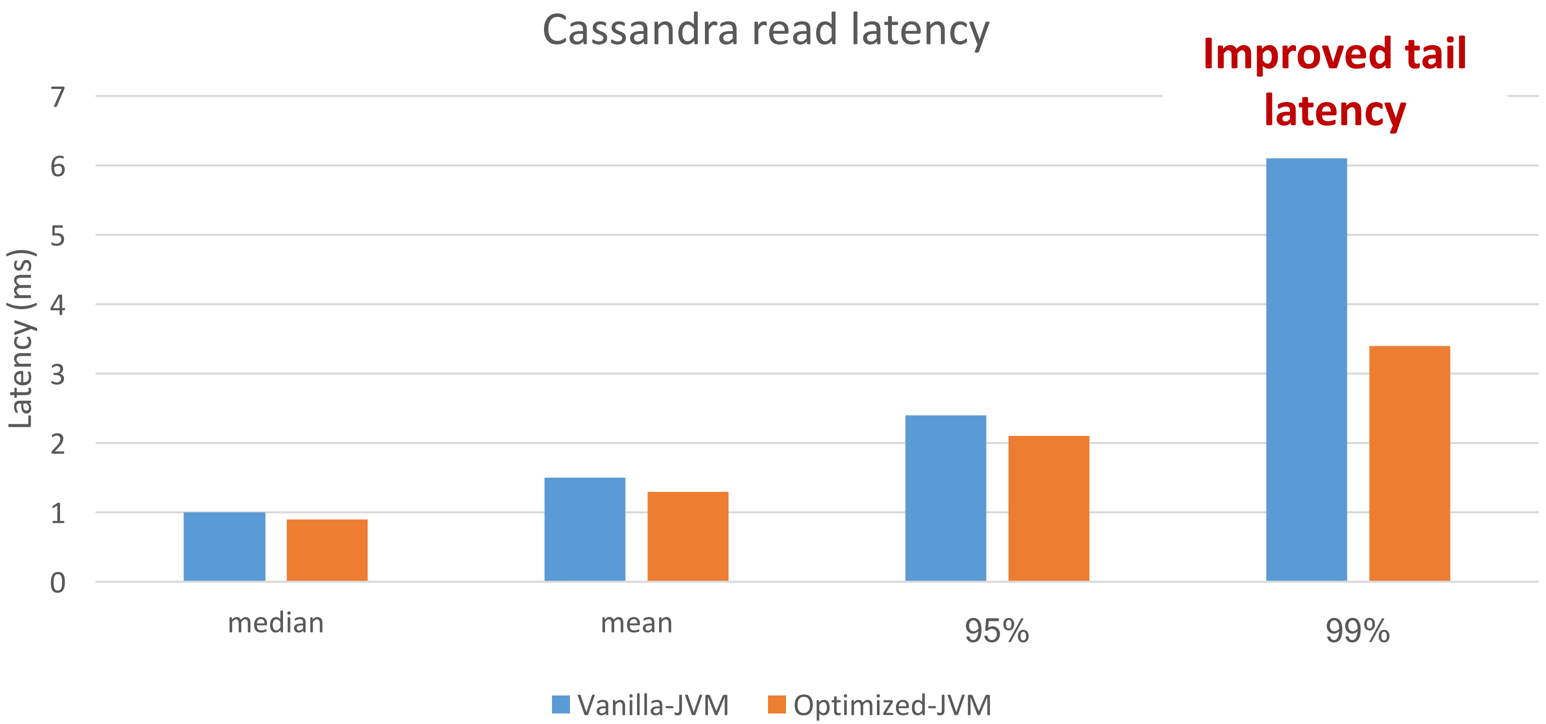


Application Results



Application Results

Lower is better



More Results in the Paper

- Scalability
- Different heap sizes
- Multiple Java programs
- Comparison with NUMA-aware GC thread placement and work stealing [Gidra-ASPLOS'13]

Insights & Takeaways

- Thread stacking can be mitigated through more **frequent** OS load balancing, but not eliminated
  - Enable SMT, disable power saving, ignore NUMA
- Possibly a bigger problem than inefficient GC
  - Inherent tradeoff between sync and OS scheduling
    - Sync -- limit concurrent lock contenders
    - OS -- most effective if all threads are active
  - Up to 68% perf. difference in PARSEC benchmarks

- More general solution in OS scheduling
- Rethinking sync optimization: OS friendly vs. unfriendly

Thank you !

Questions?