

Allocation-Phase Aware Thread Scheduling Policies to Improve Garbage Collection Performance

Feng Xian, Witty Srisa-an, Hong Jiang

University of Nebraska-Lincoln

`{fxian,witty,jiang}@cse.unl.edu`



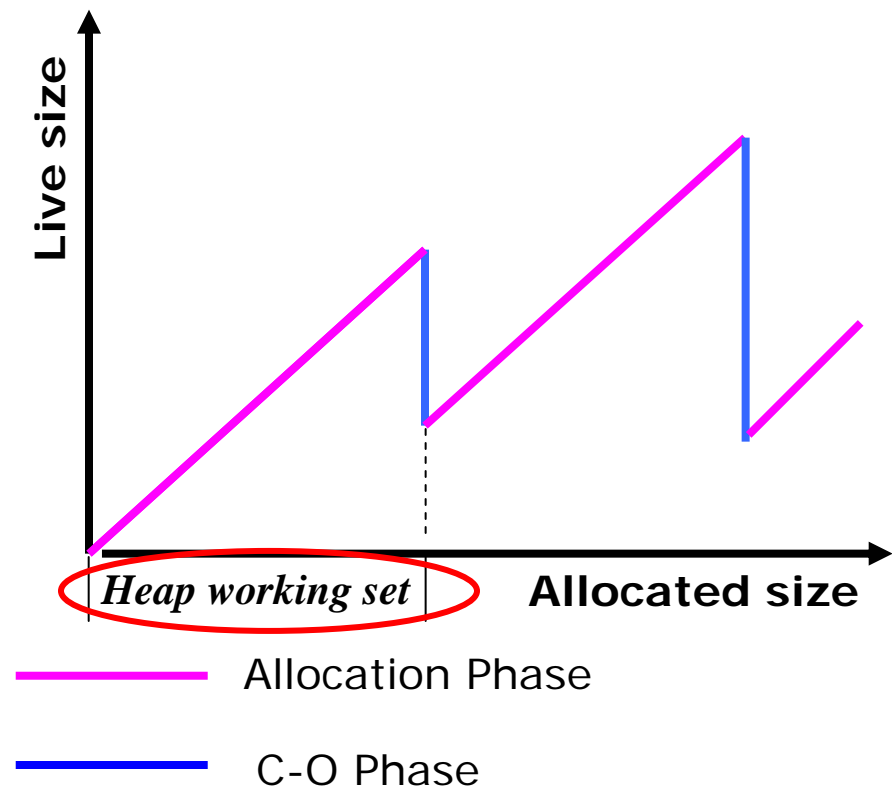


Problem Statement

- Thread scheduling can negatively affect garbage collection performance
 - By making schedulers cognizant of allocation behavior, we can improve GC performance, leading to improved overall performance

Allocation Phase Behavior

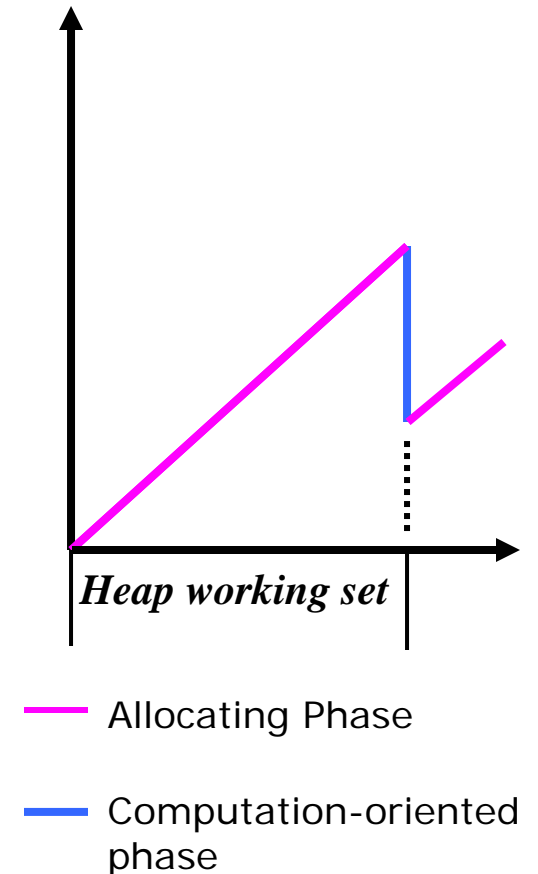
- Execution of Java threads consists of phases:
 - **Allocation phase:** an execution area where many objects are created but not too many die
 - **Computation-oriented phase (C-O phase):** an execution area where not many objects are created but many die

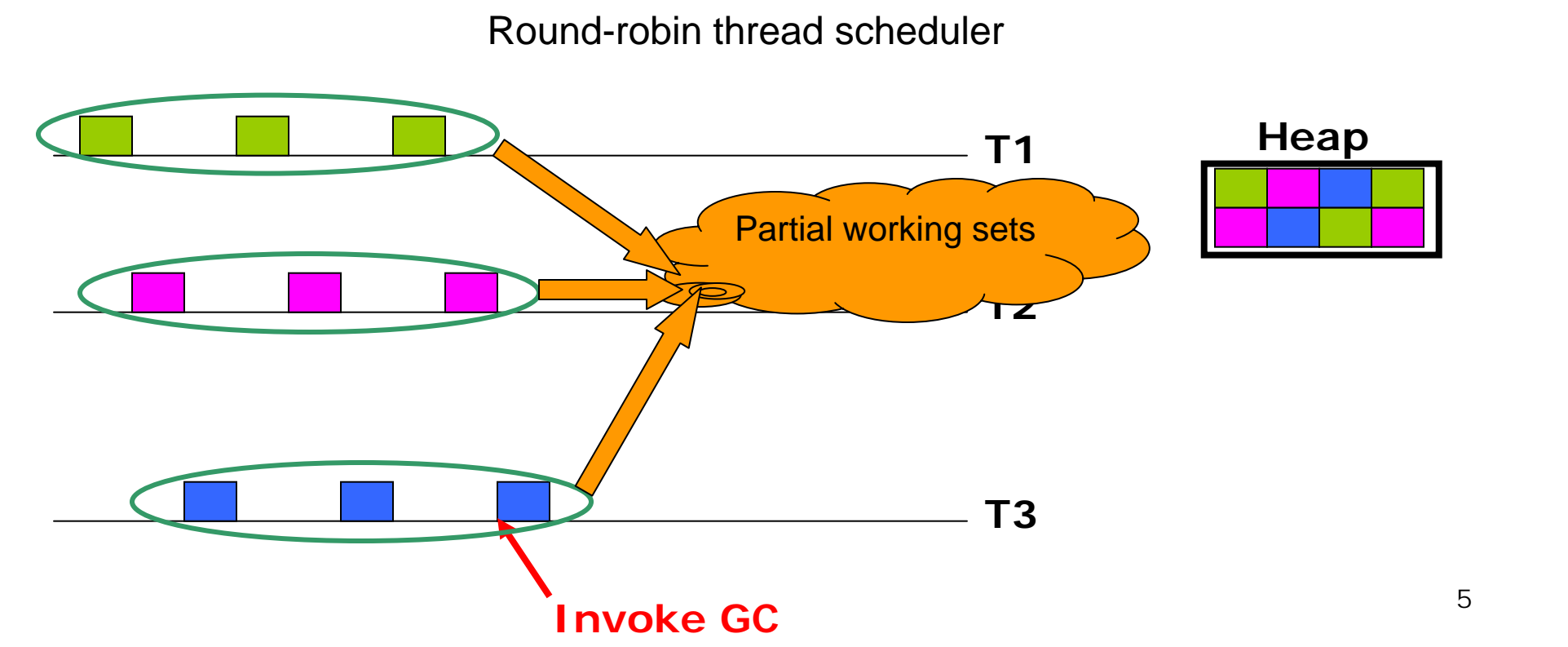
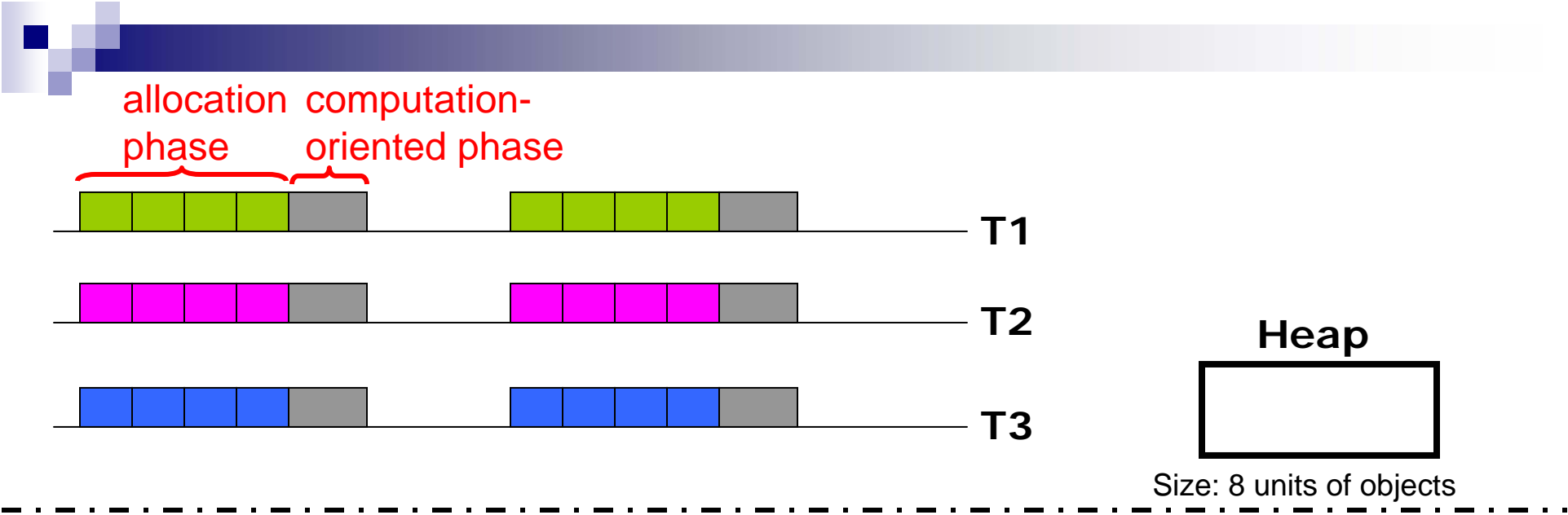


Effect of Phases in Multi-threaded Java Applications

■ Observation

- In time-sharing schedulers, each allocation phase can take multiple scheduling quanta to complete (3 to 22)
 - Threads are frequently suspended in middle of allocation phases
 - Scheduling prevents threads from completing *heap working sets* or *creating partial working set*
 - When GC is called, heap is mainly **occupied by partial working sets**
 - GC is less efficient







Phase-aware Thread Scheduler

- Schedule threads in such a way that minimizes the number of *partial working sets* in the heap

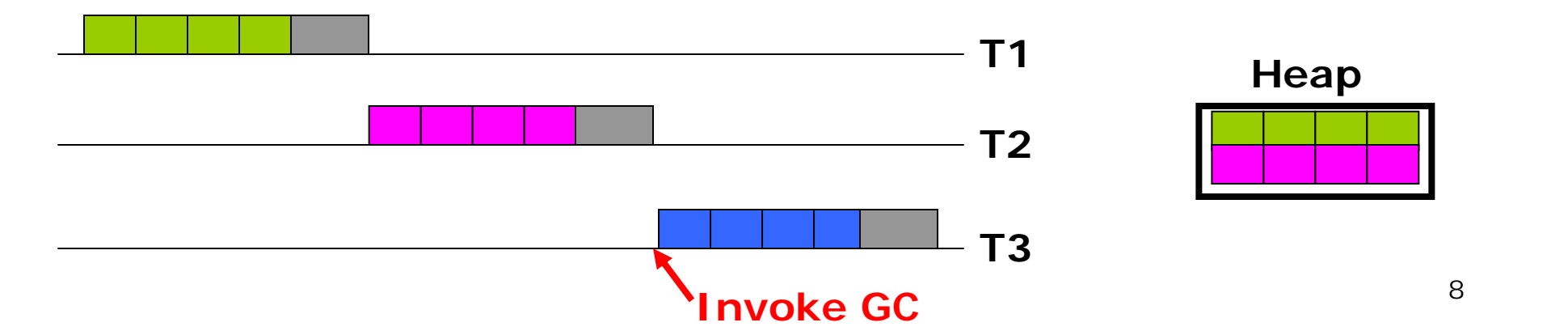
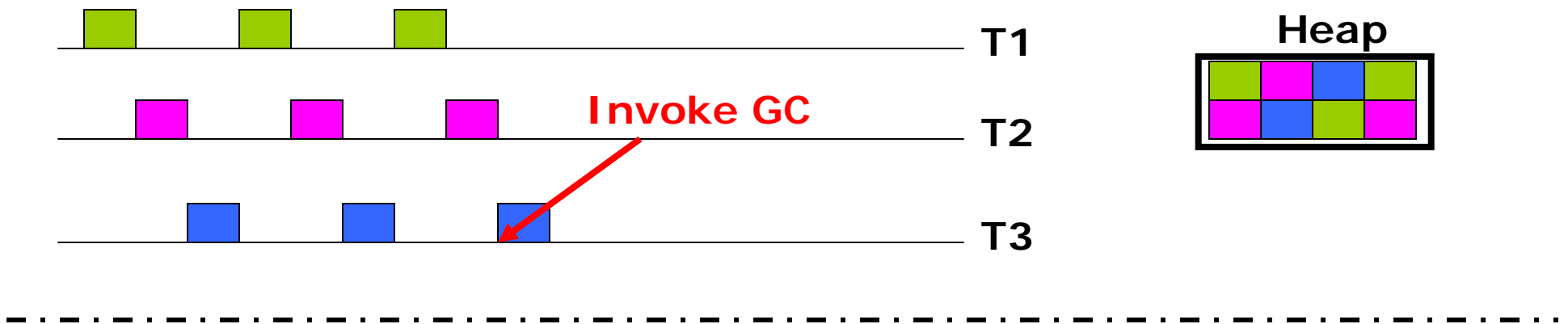


Outline

- Effect of Phase-Aware Scheduling
- Strategies
 - Memory-Quantum Round-Robin (MQRR)
 - Least Allocation-Rate First (LARF)
- Simulation-based Evaluation
- Related Work
- Conclusions and Future Work



Round-robin thread scheduler





Memory-Quantum Round-Robin (MQRR)

■ Goal

- Schedule threads in such a way that makes them complete heap working sets

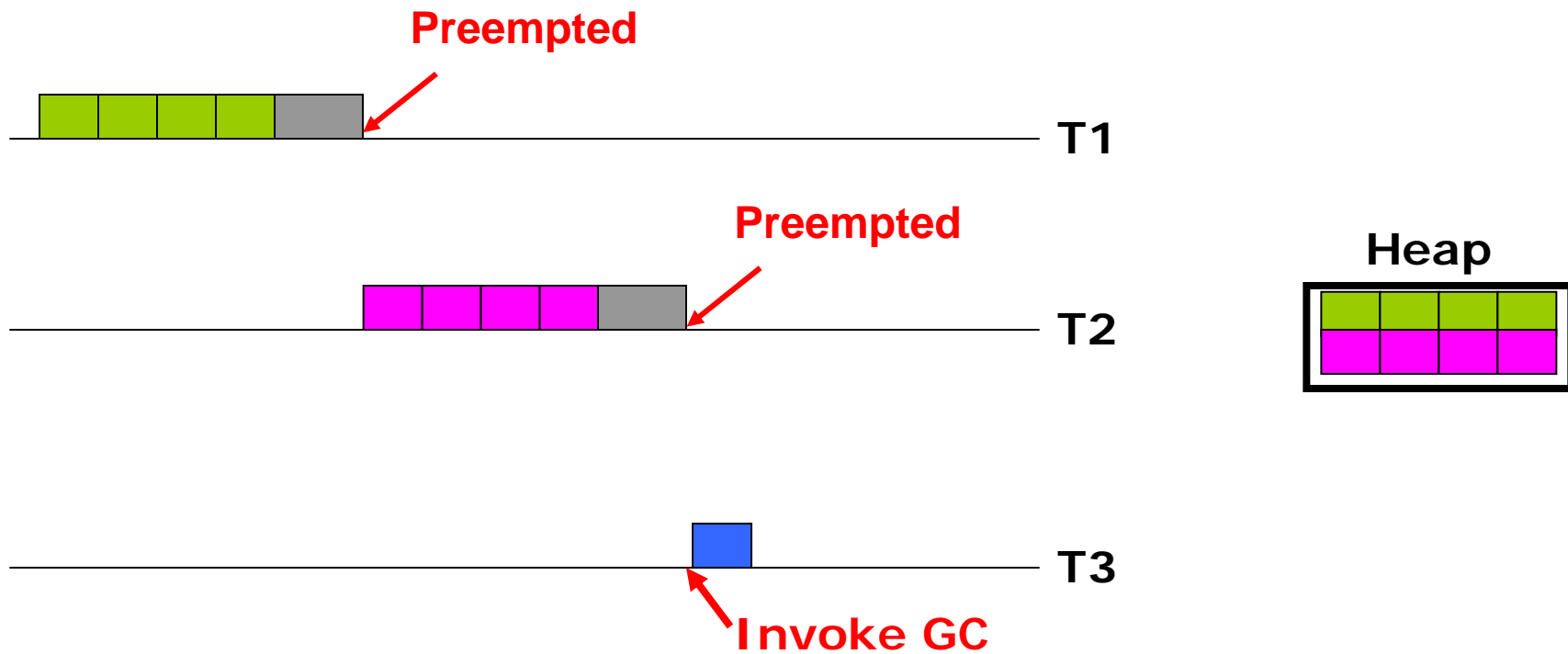
■ Observation

- Time quantum and heap usage do not relate
 - T1: 10KB (20 ms); T2: 10KB (10ms)

■ Solution

- Assign each thread a memory quantum instead of time quantum
- Memory quantum size should be tuned to be slightly larger than the most common heap working set

MQRR



Memory quantum size= 4+ units of objects



Least Allocation-Rate First (LARF)

- Goal
 - Schedule threads in such a way that maximizes the number of dead objects in the heap while using time-based schedulers
- Observations
 - Threads in C-O phases free objects
 - Threads in C-O phases do not allocate objects, thus low allocation rate
- Solution
 - Schedule threads with lowest allocation rate first
- Advantage: can work with time-based scheduler
- Disadvantage: does not take working-set size into account



LARF

- Measure the allocation rate every time quantum for each thread
- Record the most recent allocation rate of each thread
- Keep track of the lowest allocation rate and the corresponding thread
- Schedule thread with the lowest allocation rate

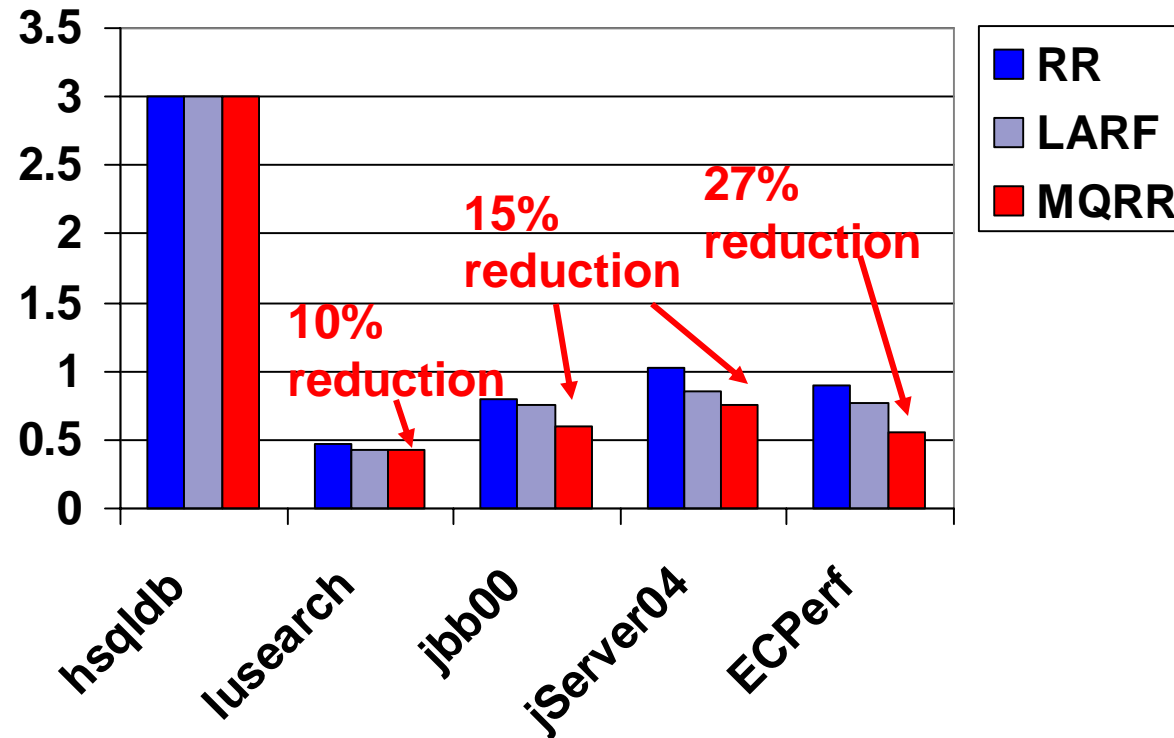


Evaluation Methodology

- Trace-driven simulation
 - Compare the GC performance and overall performance of LARF and MQRR with RR (round-robin)
- Benchmarks
 - *Hsqldb, lusearch, SPECjbb2000, SPECjAppServer2004, ECPerf*
- Parameters
 - CPU quantum of RR is $1.14 \cdot 10^{-3}$ secs (based on average length of timeslices)
 - Memory-quantum size of MQRR is 10KB (slightly over average working set size)
 - Heap size = 3*Maximum live size (reasonable size with reasonable GC overhead)

GC Performance (Mark/cons ratio)

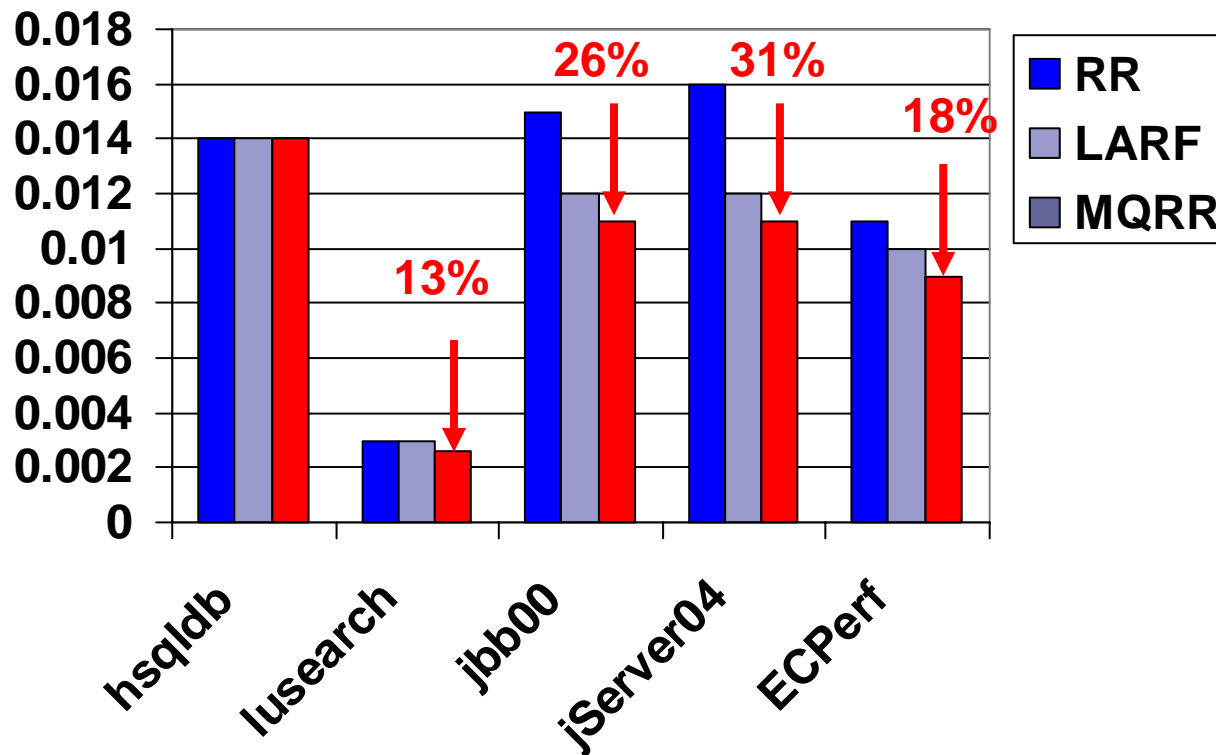
$$mark / cons = \frac{\text{Volume of copied objects}}{\text{Volume of allocated objects}} \quad [Baker'93]$$



Collector = GenMC

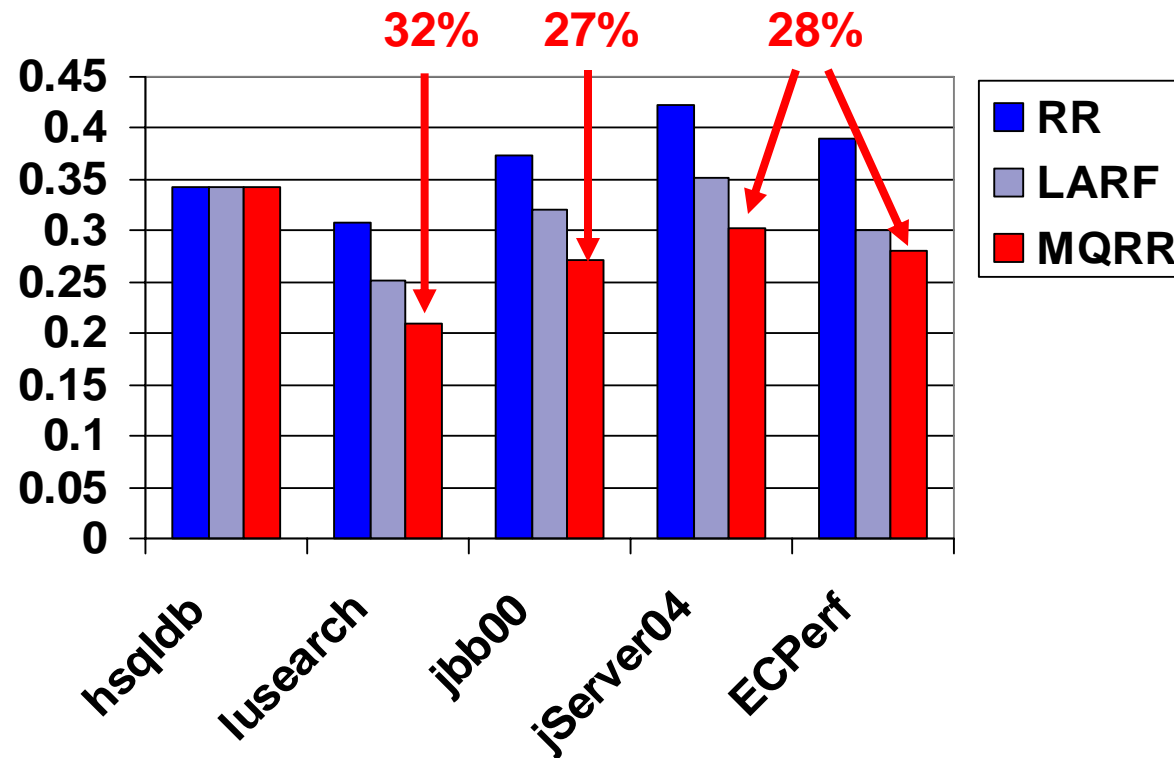
GC Performance (Avg. Pauses)

$$avg. \text{ pauses} = avg \left\{ \frac{\text{amount copied of } GC_i}{\text{heap size}} \right\} \quad [Hirzel'03]$$



Collector = GenMC

GC Performance (Max. Pauses)



Collector = GenMC



Application Performance

- Reduction in overall execution time
 - -0.1% to 3% for LARF
 - 0.2% to 9% for MQRR
- Reduction in the turn-around time
 - -0.1% to 12% for LARF
 - 0.1% to 13% for MQRR



Other Observations

- Work very well with generational collectors
 - Benefit gained through minor collection due to smaller heap space (nursery)
- Achieve marginal benefit with non-generational collectors
 - Large heaps can fit more working sets
- Work very well in tight heap situations (e.g. application server under heavy demands)
 - Tight heap means less space to fit complete working sets



Related Work

- Use virtual memory information to improve GC performance
 - CRAMM [Yang'06], Isla Vista Heap Sizing [Grzegorzczuk'07]
- Make scheduling decision based on resource availability
 - Capriccio [von Behren'03], Dedequeuees [Narlikar'03]



Conclusions and Future Work

Conclusions

- Current time sharing based thread scheduler is not GC friendly
- Proposed two phase aware thread scheduling strategies: LARF and MQRR
- GC improvement (mark/cons ratio): 0% 16%(LARF), 0% 27%(MQRR)

Future work

- Implement a phase aware thread scheduler in Linux

Allocation-Phase Aware Thread Scheduling Policies to Improve Garbage Collection Performance

Feng Xian, Witty Srisa-an, Hong Jiang

University of Nebraska-Lincoln

`{fxian,witty,jiang}@cse.unl.edu`



Partial support for this work was provided by NSF under grants CNS-0720757 and CNS-0411043.