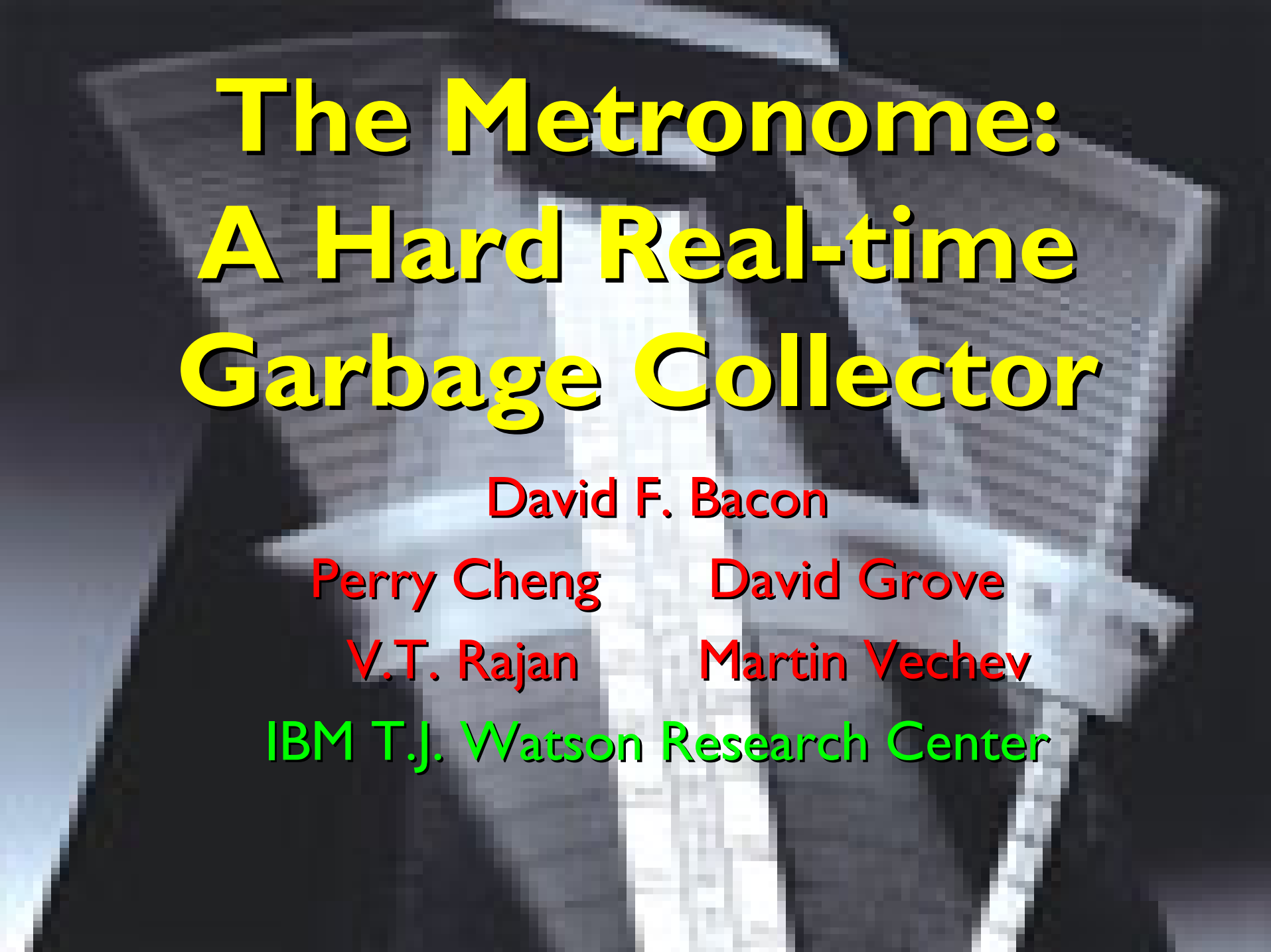


The Metronome: A Hard Real-time Garbage Collector

David F. Bacon

Perry Cheng David Grove

V.T. Rajan Martin Vechev

IBM T.J. Watson Research Center

Real Time: The Final Frontier

- Real-time systems growing in importance
  - Becoming more complex
  - BMW CEO: "80% of innovation in SW"
- Programmers left behind
  - Still using assembler and C
  - Lack productivity advantages of Java
  - Often must co-exist w/large, non-real-time Java application
- Result
  - Complexity
  - Low reliability or very high validation cost

Scope

- Time-sensitive applications
  - Traditional "hard" real-time systems
  - Soft real-time systems (games, media, ...)
  - OS components: device drivers, fault handlers, …
- Uniprocessor
  - Multiprocessors rare in real-time systems
  - Complication: collector must be finely interleaved
  - Simplification: memory model easier to program
    - No truly concurrent operations
    - Sequentially consistent

Metronome Project Goals

- Make GC feasible for hard real-time systems
- Provide simple application interface
- Develop technology that is efficient:
  - Throughput, Space comparable to stop-the-world
- **BREAK THE MILLISECOND BARRIER**
  - While providing even CPU utilization

Where's the Beef?

- Maximum pause time < 4 ms
- Utilization > 50% ±2%
- Memory requirement < 2 X max live

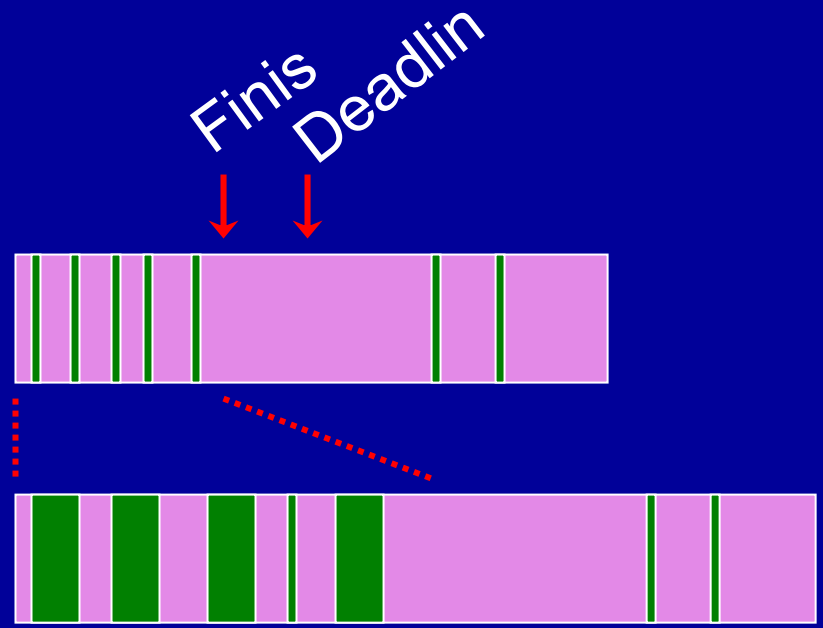
Controlling Time: Scheduling Collection

The Baker Fallacy

"A real-time list processing system is one in which the time required by the elementary list processing operations…is bounded by a small constant" [Baker'78]

Why Doesn't It Work?

- Operations not smooth
- Constant not so small
- Traps are not smooth

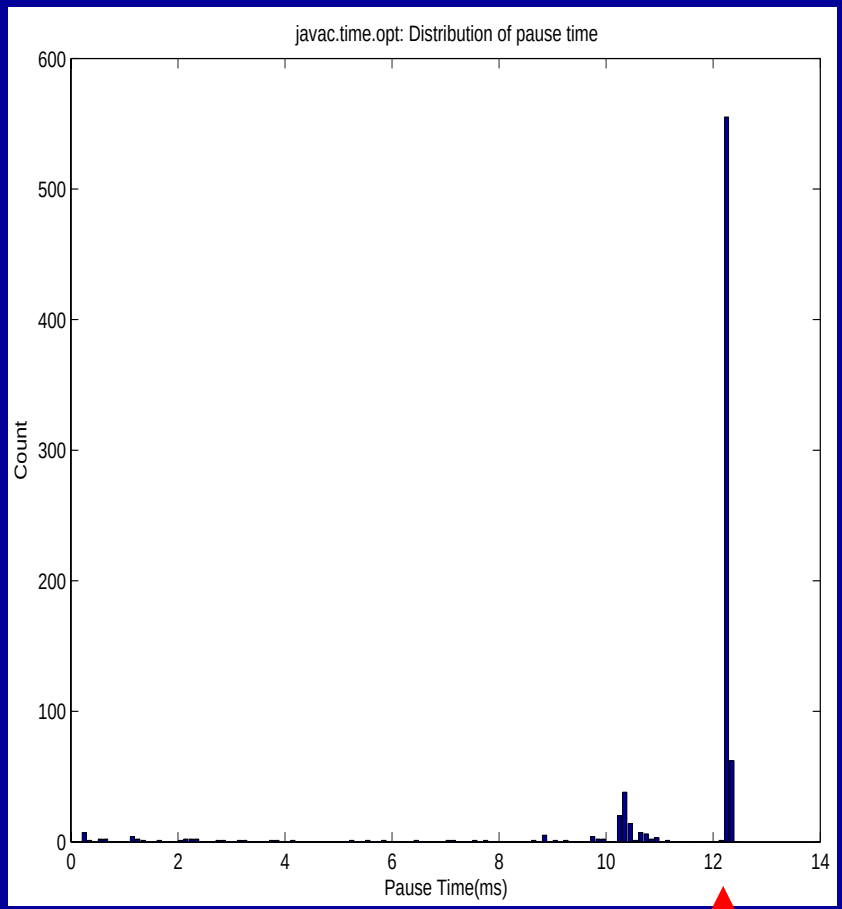


Scheduling: Work vs. Time

- Baker approach is work-based
  - On barriers/allocations do a little GC work
  - For constant k, utilization may drop to $1/k$
- Metronome is time-based
  - Tick (user), Tock (GC)
  - Utilization is guaranteed to be tick/(tick+tock)
    - Minimum Mutator Utilization (MMU)
  - Issue: can we keep up with the mutator?

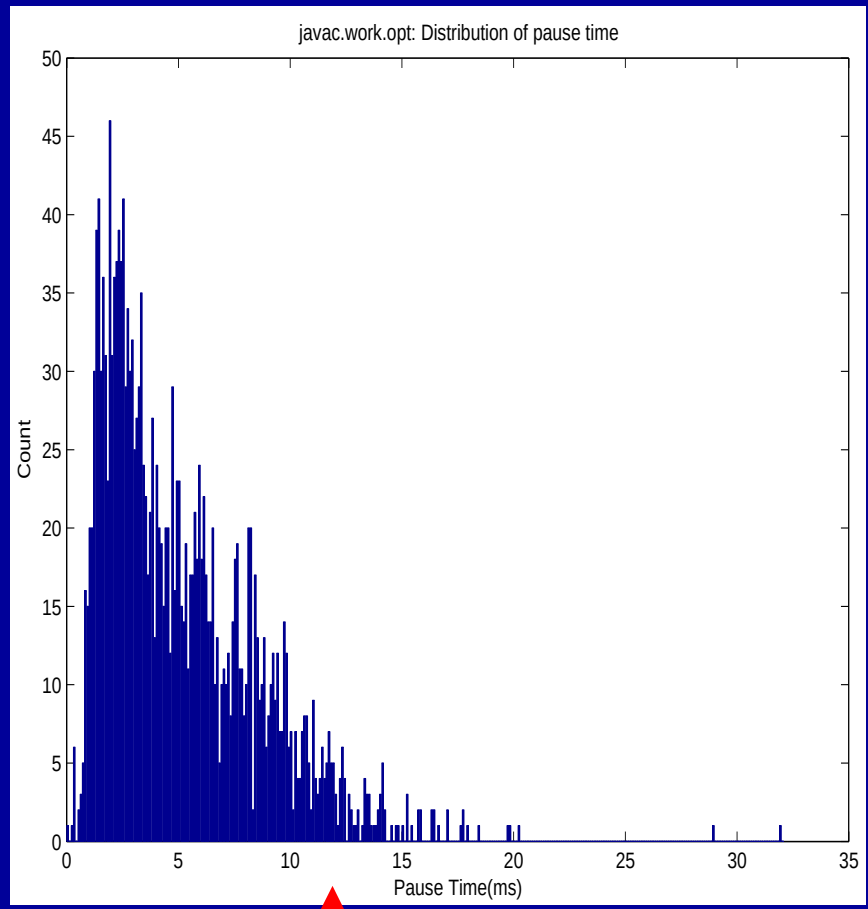
Pause time distribution: javac

Time-based Scheduling



12 ms

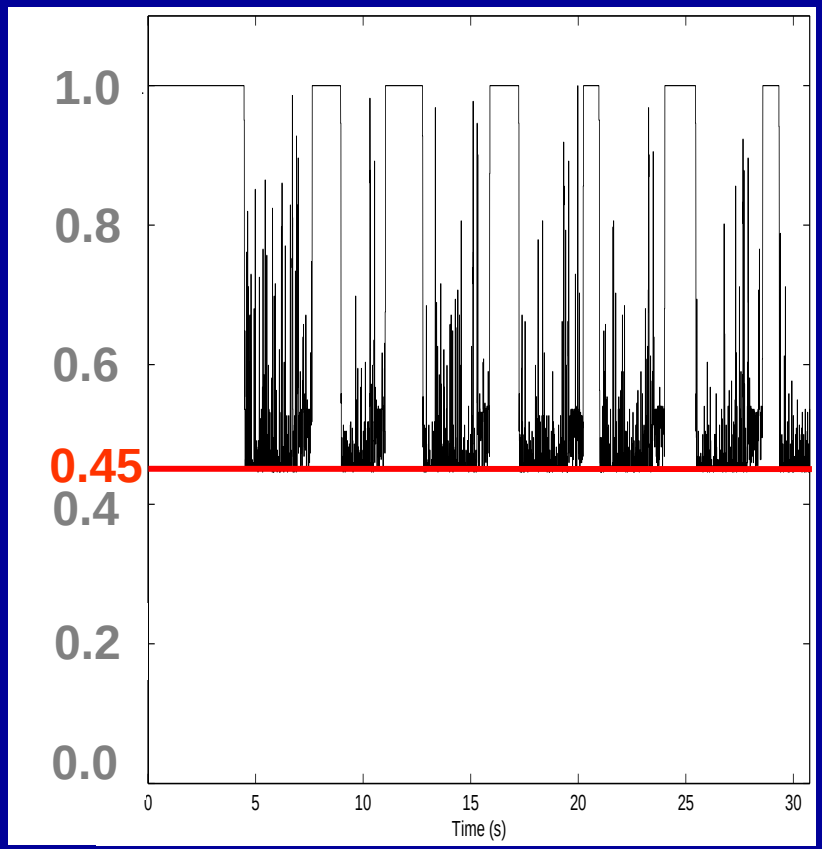
Work-based Scheduling



12 ms

Utilization vs Time: javac

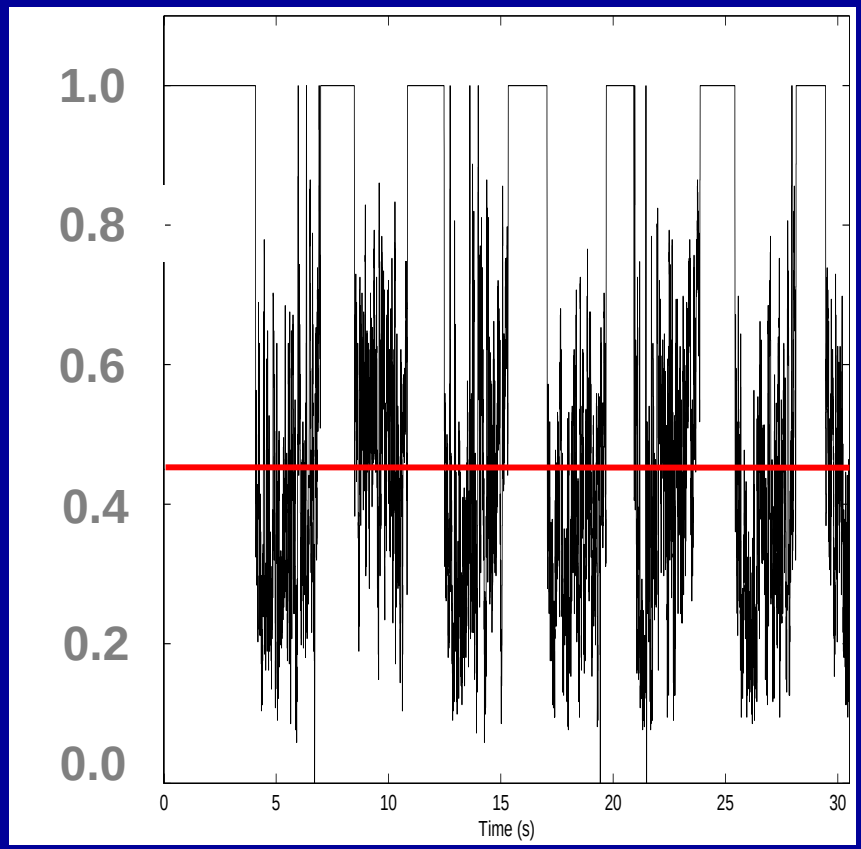
Time-based Scheduling



Time (s)

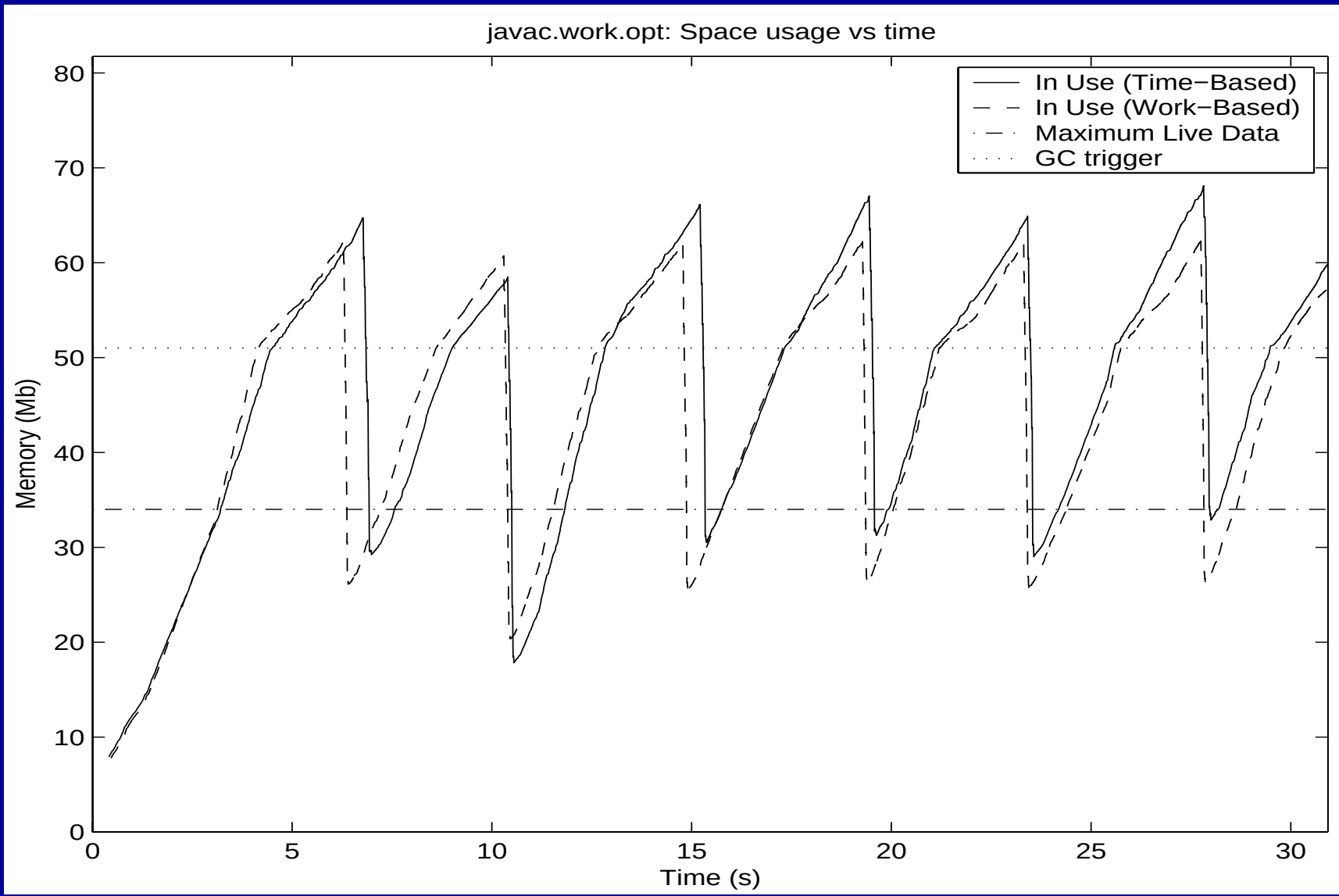
Utilization (%)

Work-based Scheduling

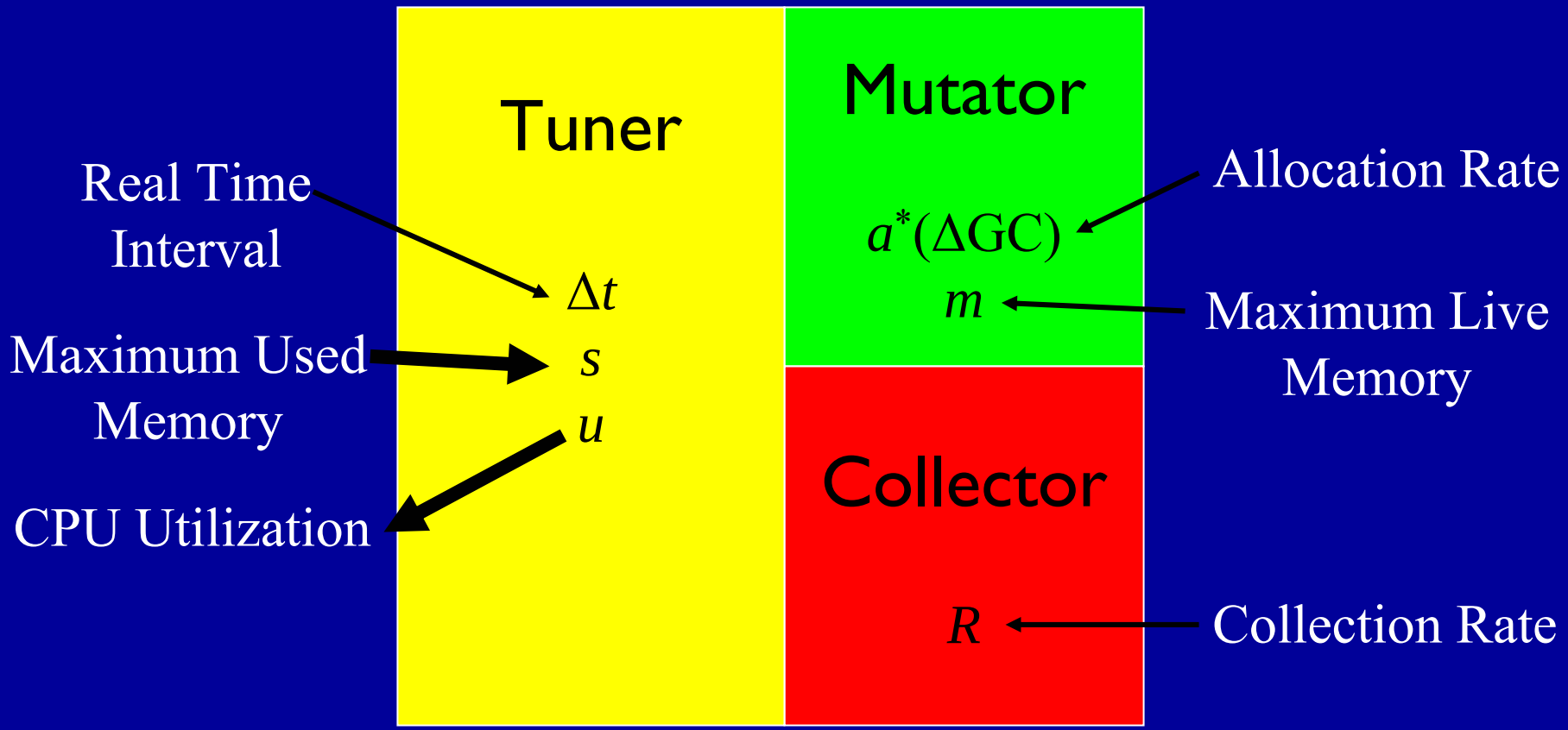


Time (s)

Space Usage: javac

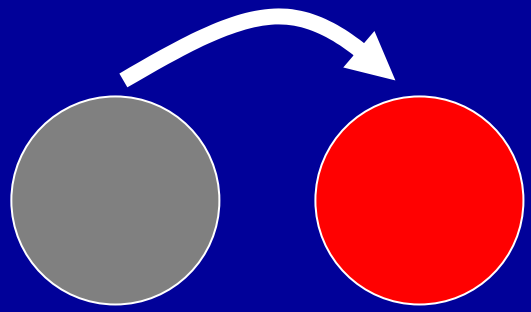


Parameterization

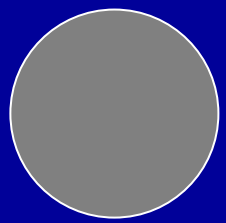


Controlling Space: Layout and Compaction

Two Basic Organizations



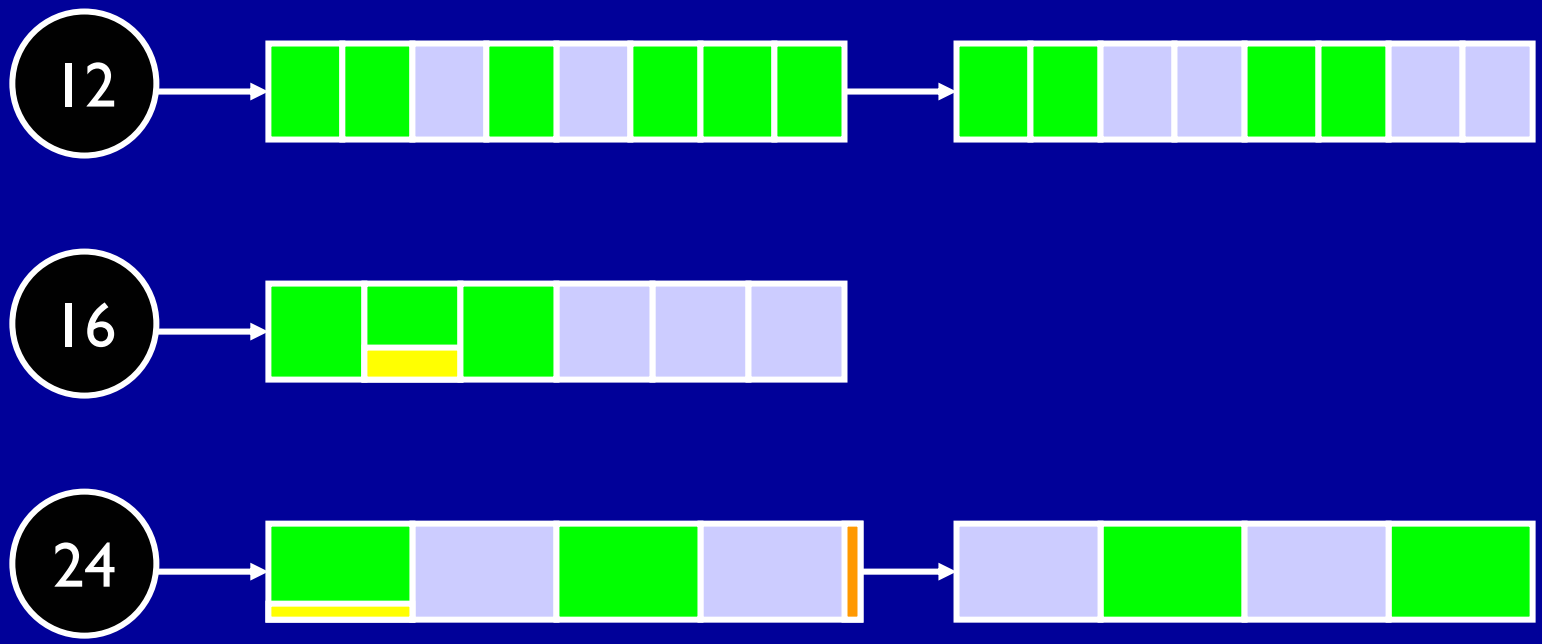
- Semi-space Copying
  - Concurrently copies between spaces
  - Cost of copying, consistency
  - Fragmentation 2x, but bounded



- Mark-sweep (non-copying)
  - Fragmentation avoidance, coalescing
  - Fragmentation unbounded.

Segregated Free-list Architecture

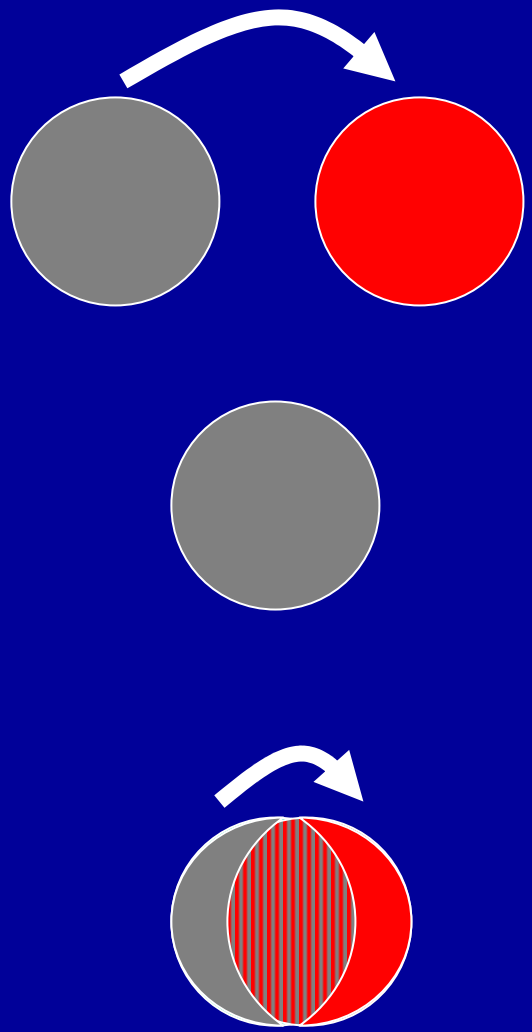
- Suffers from 5 (!) kinds of fragmentation
  - Internal and page-internal: bound (e.g. 1/8)
  - External: Break up large objects (arraylets)
  - Page external: Defragment
  - Size external: eat it (up to 1 page/size class)



Selective Defragmentation

- When do we move objects?
  - When there is fragmentation
- Usually: program exhibits *locality of size* (λ)
  - Dead objects are re-used quickly
  - Fragmentation is low
- But: defragment either when
  - Dead objects are not re-used for n GC cycles
  - Free pages fall below limit for performing a GC
- In practice: we move 2-3% of live data
  - Major improvement over copying collector
  - Very resilient to adversary programs

Metronome Adapts



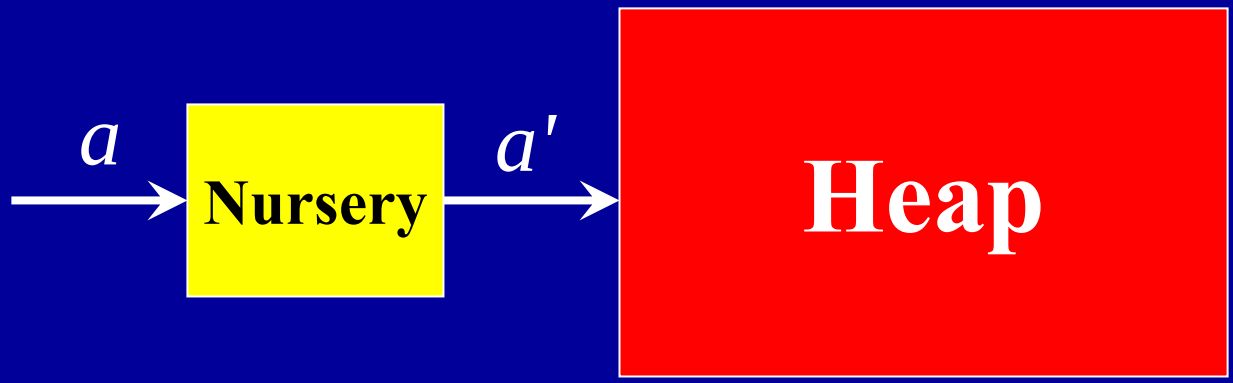
- Semi-space Copying
  - Concurrently copies between spaces
  - Fragmentation 2x, but bounded
- Mark-sweep (non-copying)
  - Fragmentation avoidance, coalescing
  - Fragmentation unbounded
- ***The Metronome***
  - Mark-sweep, selective defragmentation
  - Best of both, adaptively adjusts

Metronome:
The Next Generation

Time to Get Serious

- Significant interest
  - But Jikes RVM is not product-level
- Pause time/MMU meets wide class of needs
  - But utilization during GC is an issue
- Next-generation Implementation
  - In IBM's J9 product (no commitment expressed…)
  - Higher performance
    - Generational
  - Native threads (challenging)
  - Binary distribution to groups at Berkeley, Salzburg

Generational RTGC



- Allocation rate is single biggest factor
  - Nursery reduces heap allocation rate
- Our STW GC: collects 64K in 4ms (ARM)
  - So, collect small nursery synchronously
- Survival rate usually < 15% (with tricks)
  - But predictability is harder (interface/methodology issue)

Conclusions

- The Metronome provides true real-time GC
  - First collector to do so without major sacrifice
    - Short pauses (4 ms)
    - High MMU during collection (50%)
    - Low memory consumption (2x max live)
- Critical features
  - Time-based scheduling
  - Hybrid, mostly non-copying approach
  - Integration w/compiler

Future Work

- Two main goals:
  - Reduce pause time, memory requirement
  - Increase predictability
- Pause time:
  - Expect sub-millisecond using current techniques
  - For 10's of microseconds, need interrupt-based
- Predictability
  - Studying parameterization of collector
  - Good research area

Backup Slides

Components of the Metronome

- Incremental mark-sweep collector
  - Mark phase fixes stale pointers
- Selective incremental defragmentation
  - Moves < 2% of traced objects
- Time-based scheduling
- Segregated free list allocator
  - Geometric size progression limits internal fragmentation

Old New

Support Technologies

- Read barrier: to-space invariant [Brooks]
  - New techniques with only 4% overhead
- Write barrier: snapshot-at-the-beginning [Yuasa]
- Arraylets: bound fragmentation, large object ops

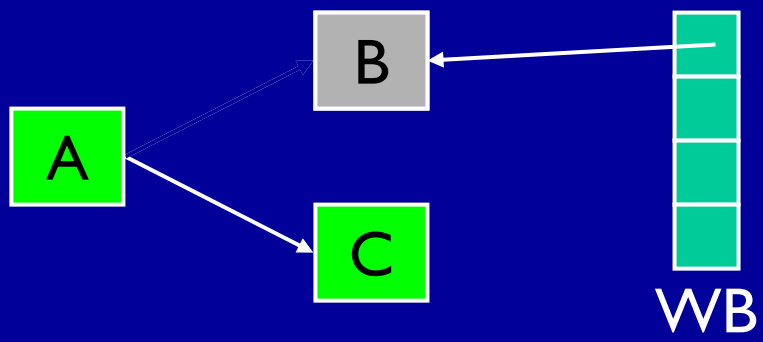
Old New

Limiting Internal Fragmentation

- Choose page size P and block sizes s_k such that
  - $s_k = s_{k-1}(1+\rho)$
  - $s_{max} = P\,\rho$
- Then
  - Internal and page-internal fragmentation $< \rho$
- Example:
  - $P = 16KB$, $\rho = 1/8$, $s_{max} = 2KB$
  - Internal and page-internal fragmentation $< 1/8$

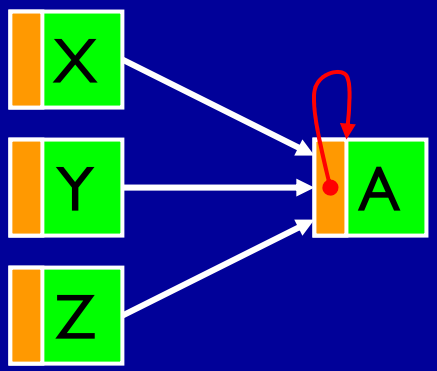
Write Barrier: Snapshot-at-start

- Problem: mutator changes object graph
- Solution: write barrier prevents lost objects
- Logically, collector takes atomic snapshot
  - Objects live at snapshot will not be collected
  - Write barrier saves overwritten pointers [Yuasa]
  - Write buffer must be drained periodically

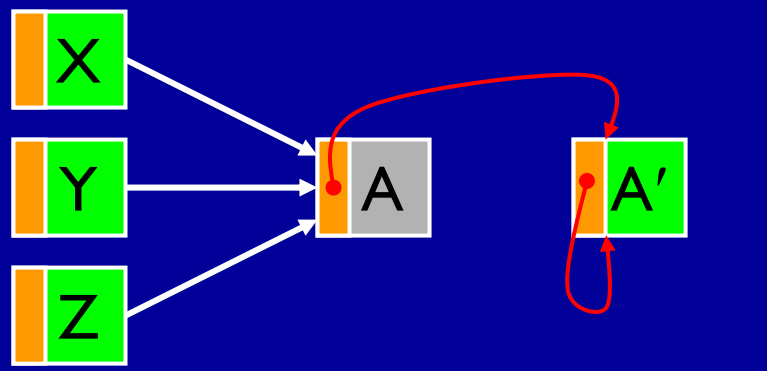


Read Barrier: To-space Invariant

- Problem: Collector moves objects (defragmentation)
  - and mutator is finely interleaved
- Solution: read barrier ensures consistency
  - Each object contains a forwarding pointer [Brooks]
  - Read barrier unconditionally forwards all pointers
  - Mutator never sees old versions of objects



BEFORE



From-space To-space

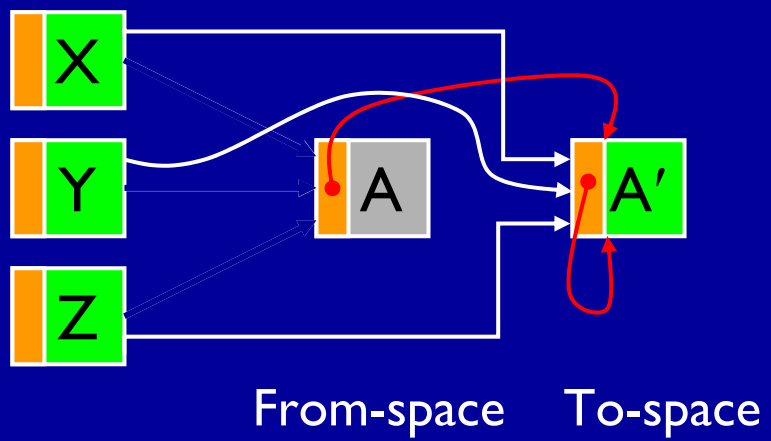
AFTER

Incremental Mark-Sweep

- Mark/sweep finely interleaved with mutator
- Write barrier ensures no lost objects
  - Must treat objects in write buffer as roots
- Read barrier ensures consistency
  - Marker always traces correct object
- With barriers, interleaving is simple

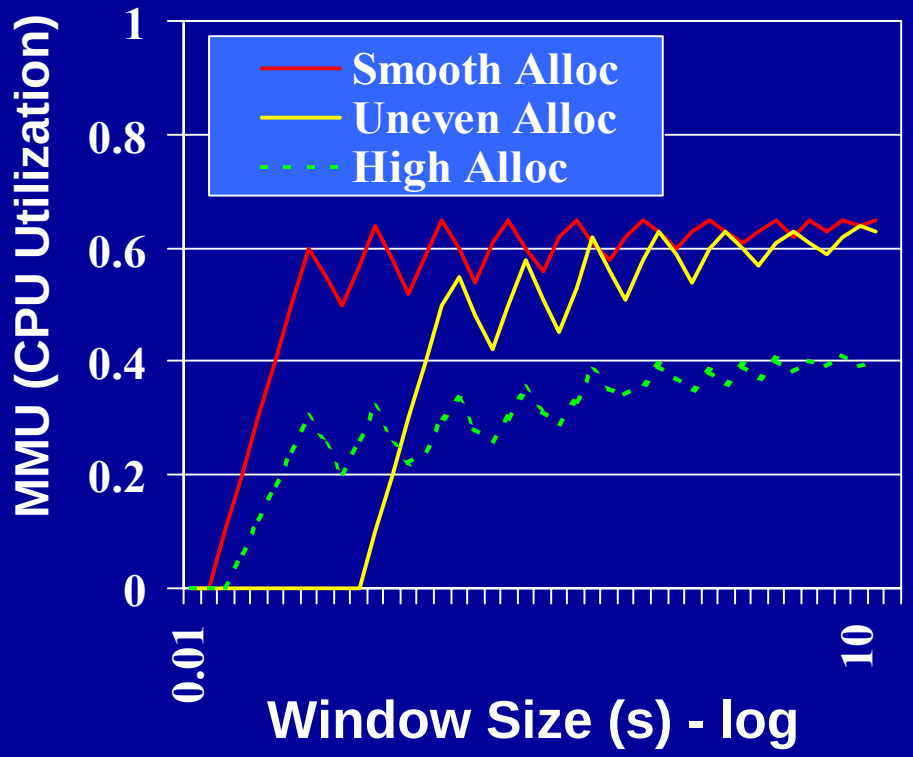
Pointer Fixup During Mark

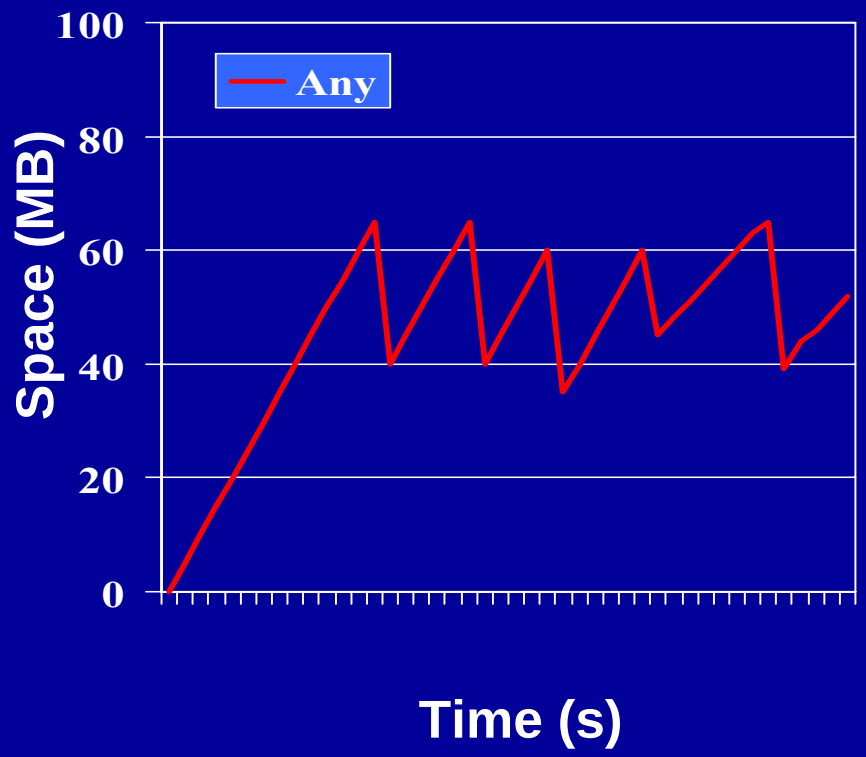
- When can a moved object be freed?
  - When there are no more pointers to it
- Mark phase updates pointers
  - Redirects forwarded pointers as it marks them
- Object moved in collection *n* can be freed:
  - At the end of mark phase of collection *n+1*



Work-based Scheduling

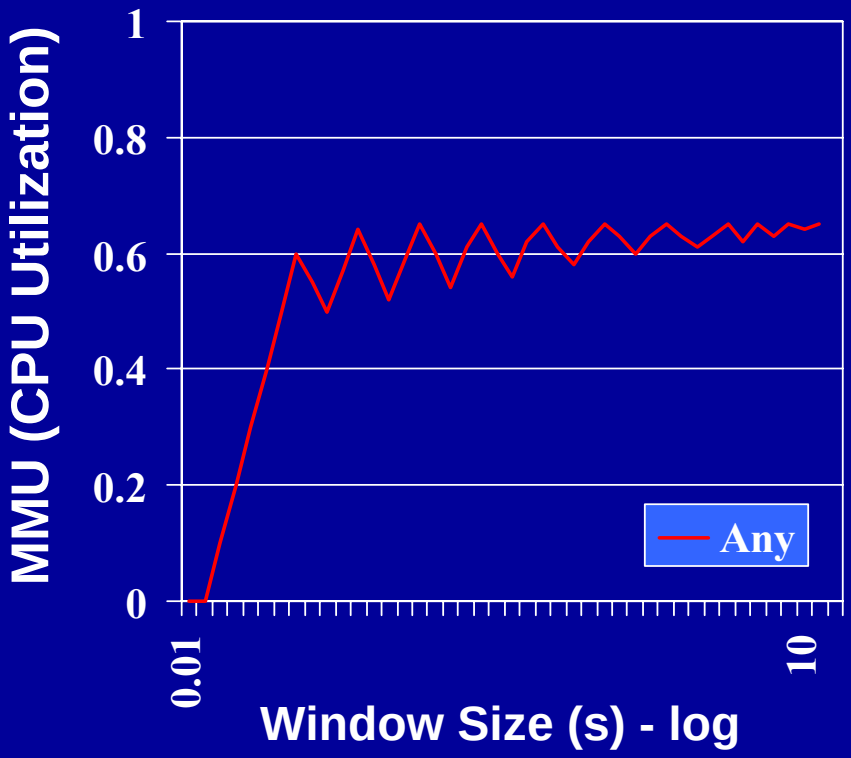
- Trigger the collector to collect C_W bytes
  - Whenever the mutator allocates Q_W bytes

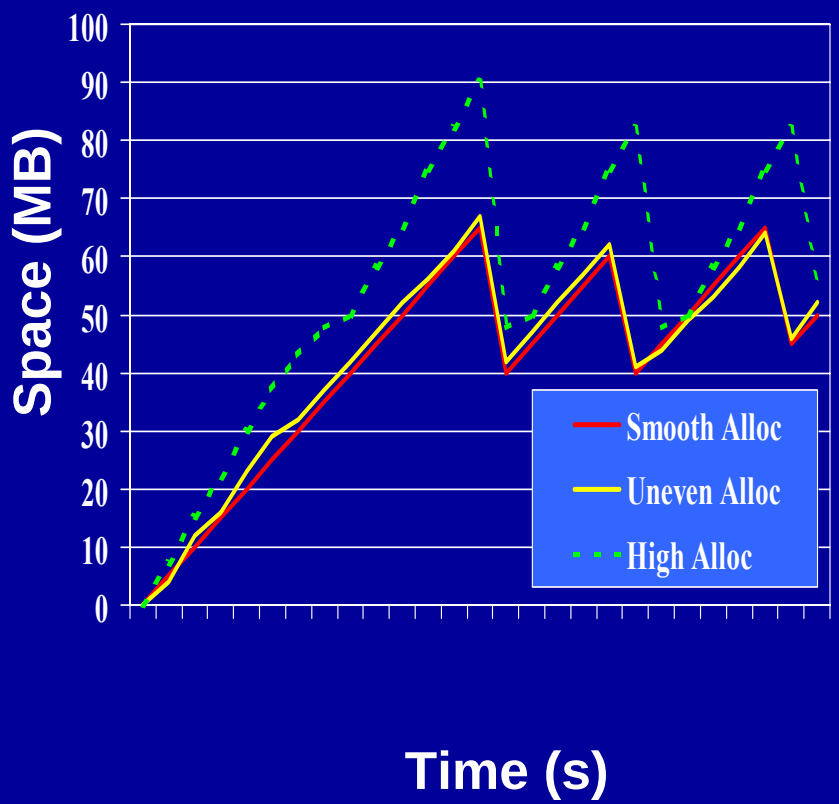




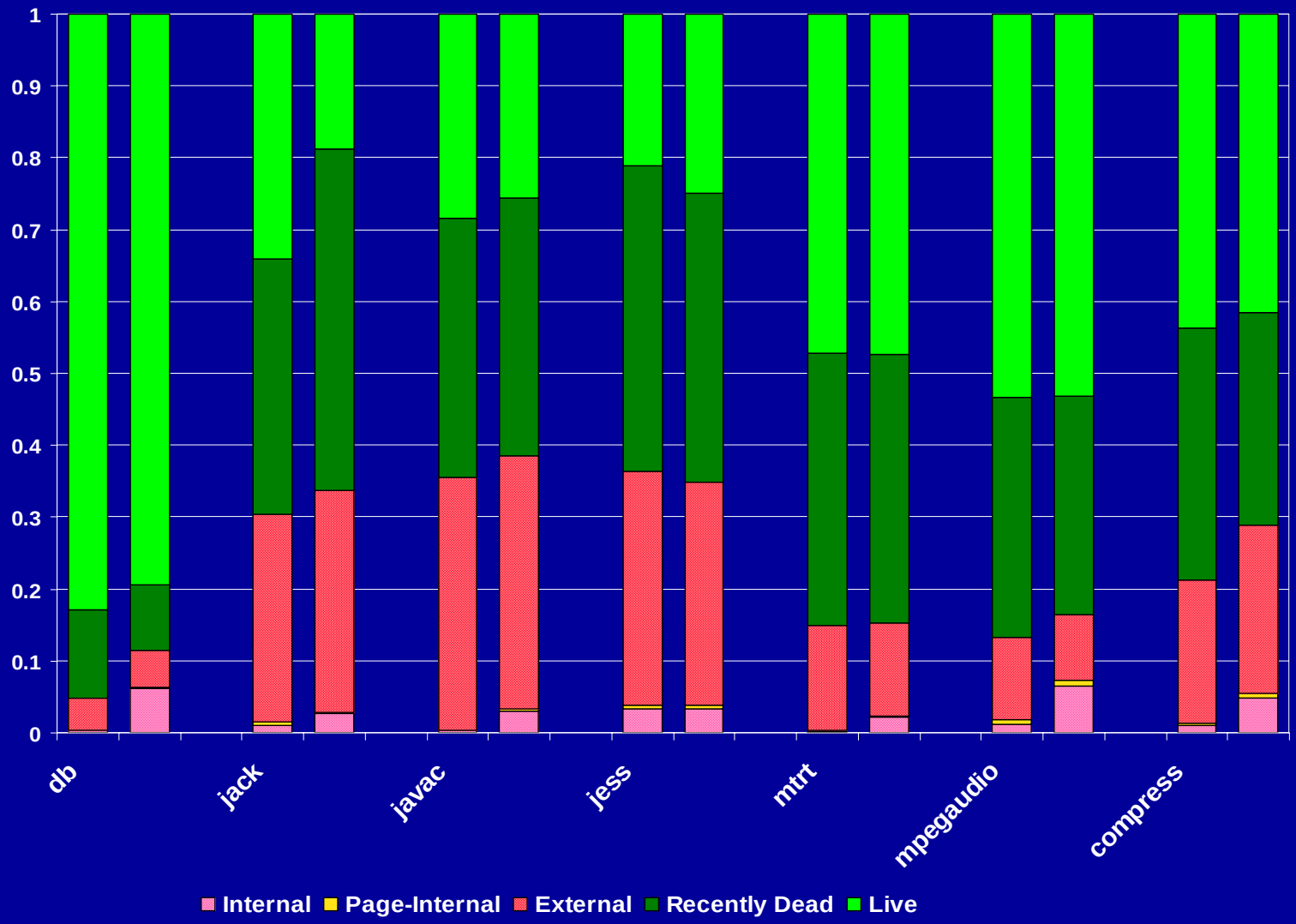
Time-based Scheduling

- Trigger collector to run for C_T seconds
  - Whenever mutator runs for Q_T seconds





Fragmentation: ρ=1/8 vs ρ=1/2



Read Barrier Optimizations

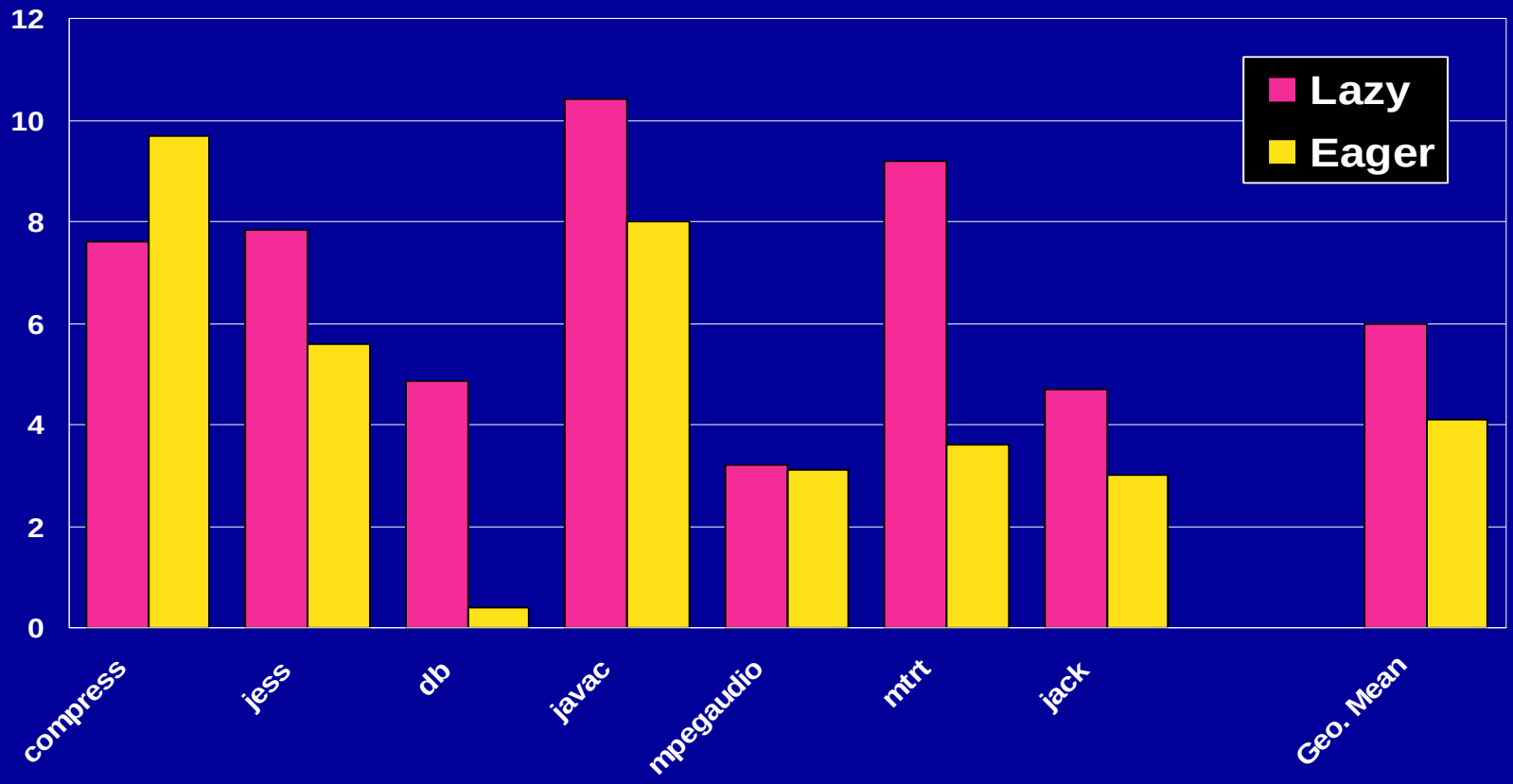
- Barrier variants: when to redirect
  - Lazy: easier for collector (no fixup)
  - Eager: better for performance (loop over a[i])
- Standard optimizations: CSE, code motion
- Problem: pointers can be null
  - Augment read barrier for GetField(x,offset):

```
tmp = x[offset];
return tmp == null ? null : tmp[redirect]
```

  - Optimize by null-check combining, sinking

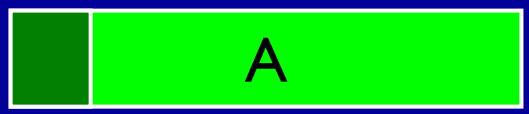
Read Barrier Results

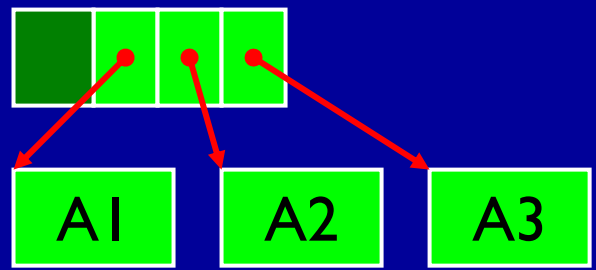
- Conventional wisdom: read barriers too slow
  - Previous studies: 20-40% overhead [Zorn,Nielsen]



Arraylets

- Large arrays create problems
  - Fragment memory space
  - Can not be moved in a short, bounded time
- Solution: break large arrays into arraylets
  - Access via indirection; move one arraylet at a time
- Optimizations
  - Type-dependent code optimized for contiguous case
  - Opportunistic contiguous allocation



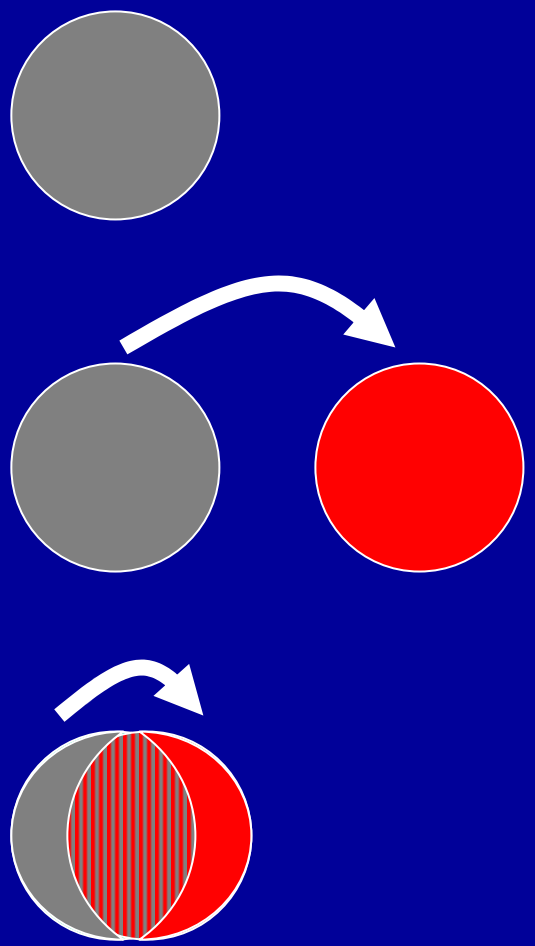


Locality of Size: λ

- Measures reuse
- Normalized: $0 \leq \lambda \leq 1$
- Segregated by size

$$\lambda = \sum_i \min (f_i / f,\ a_i / a)$$

λ in Real-time GC Context



- Mark-sweep (non-copying)

  Assumes λ=1

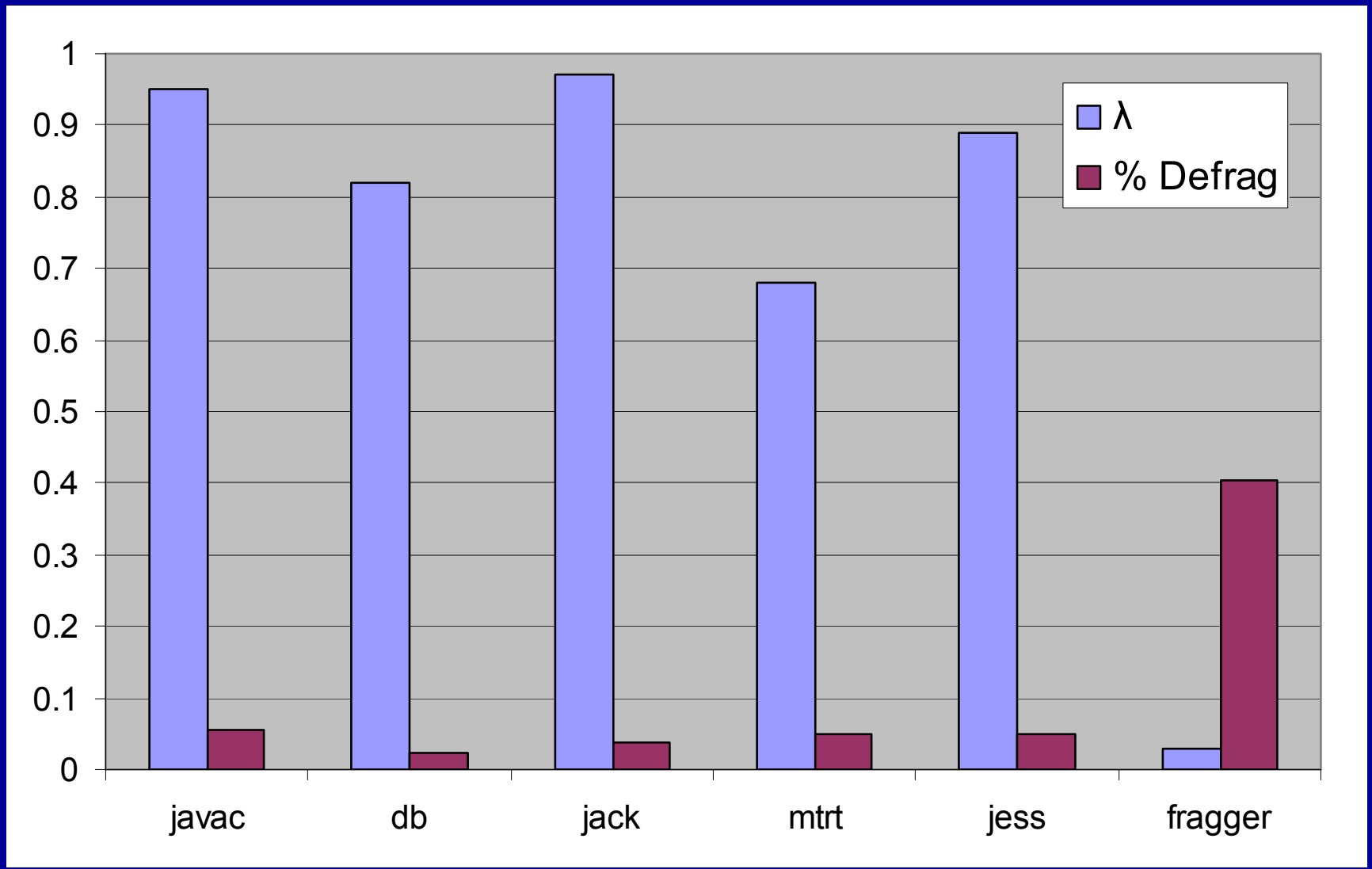
- Semi-space Copying

  Assumes λ=0

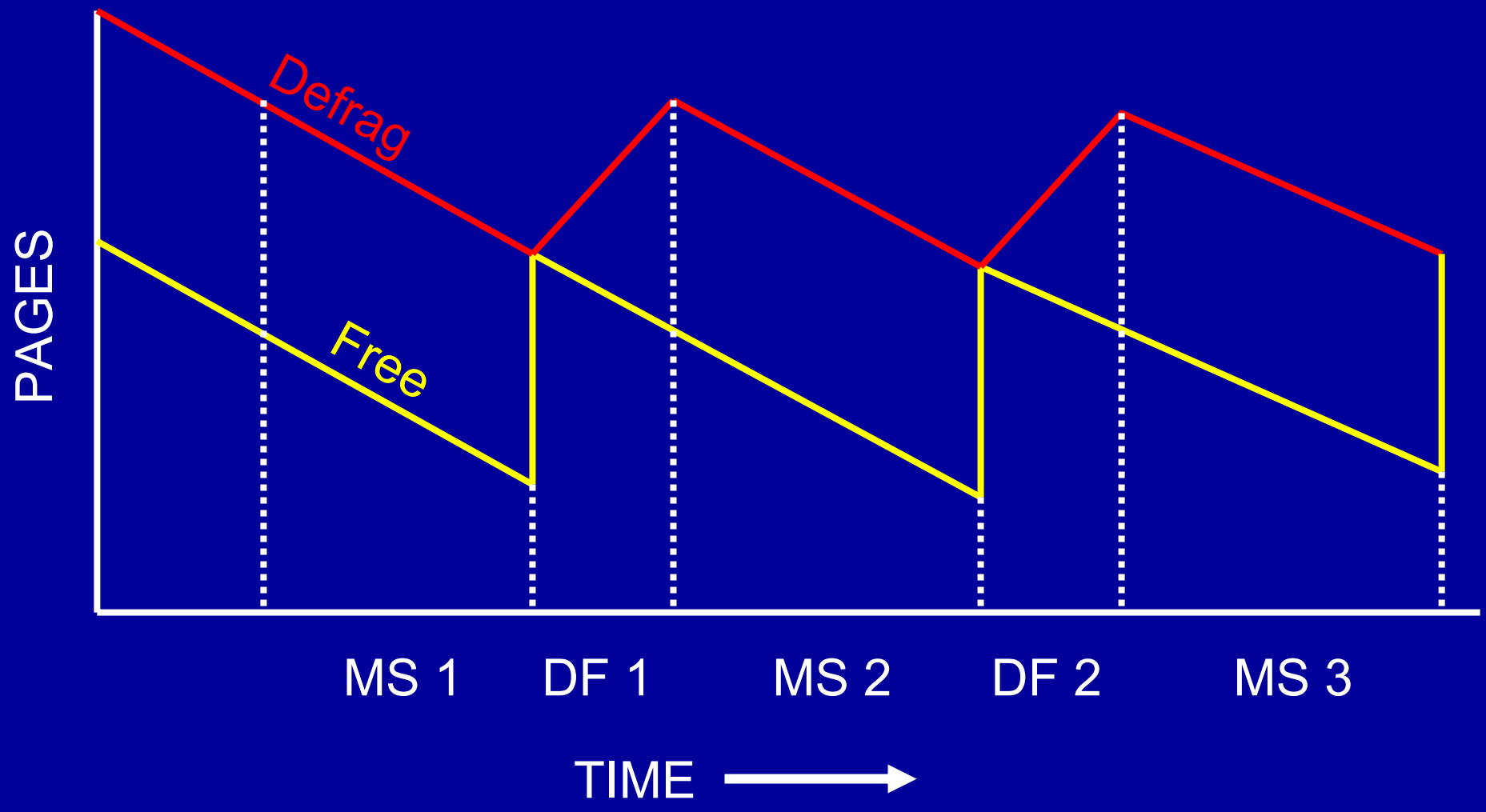
- ***The Metronome***

  Adapts as λ varies

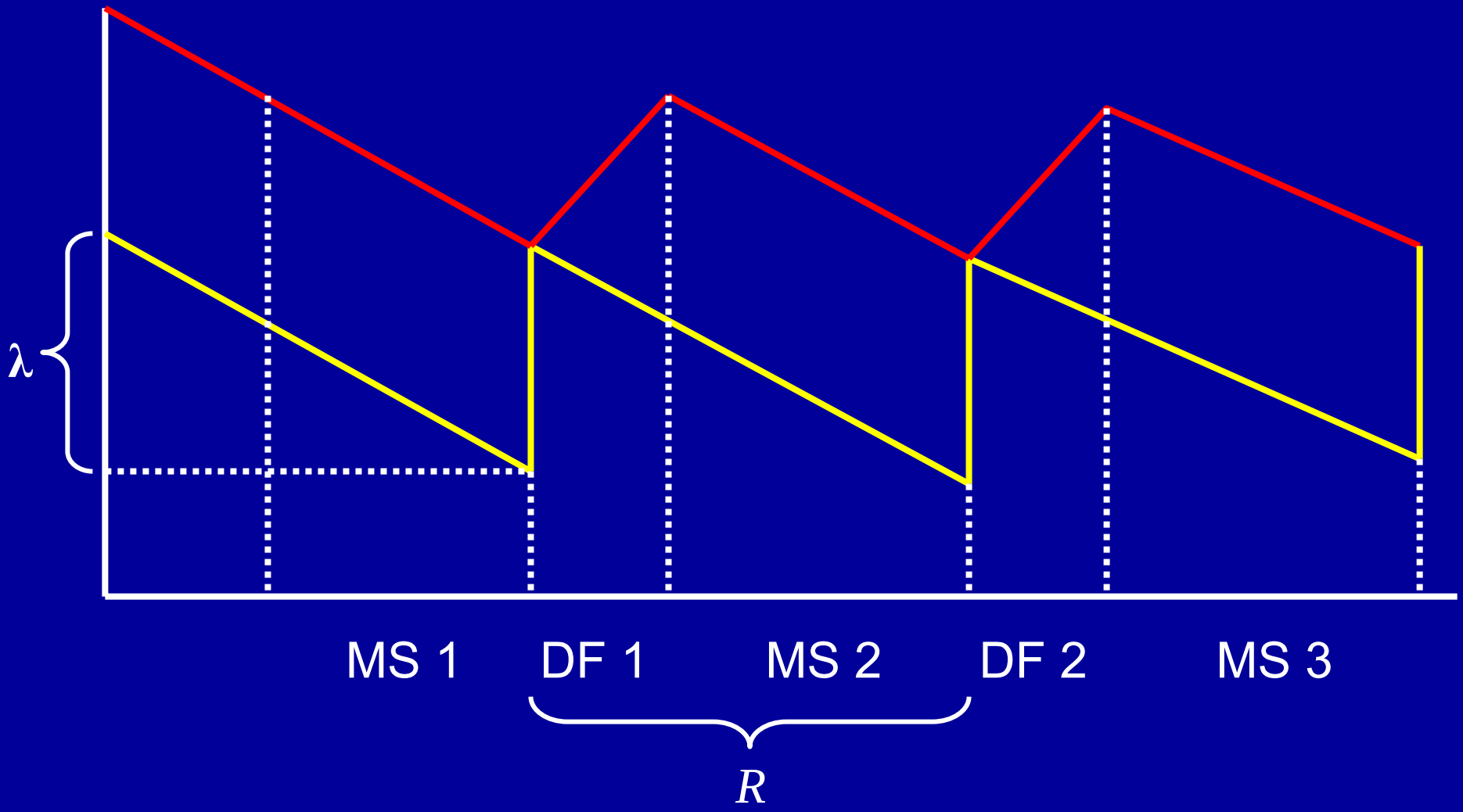
λ in Practice



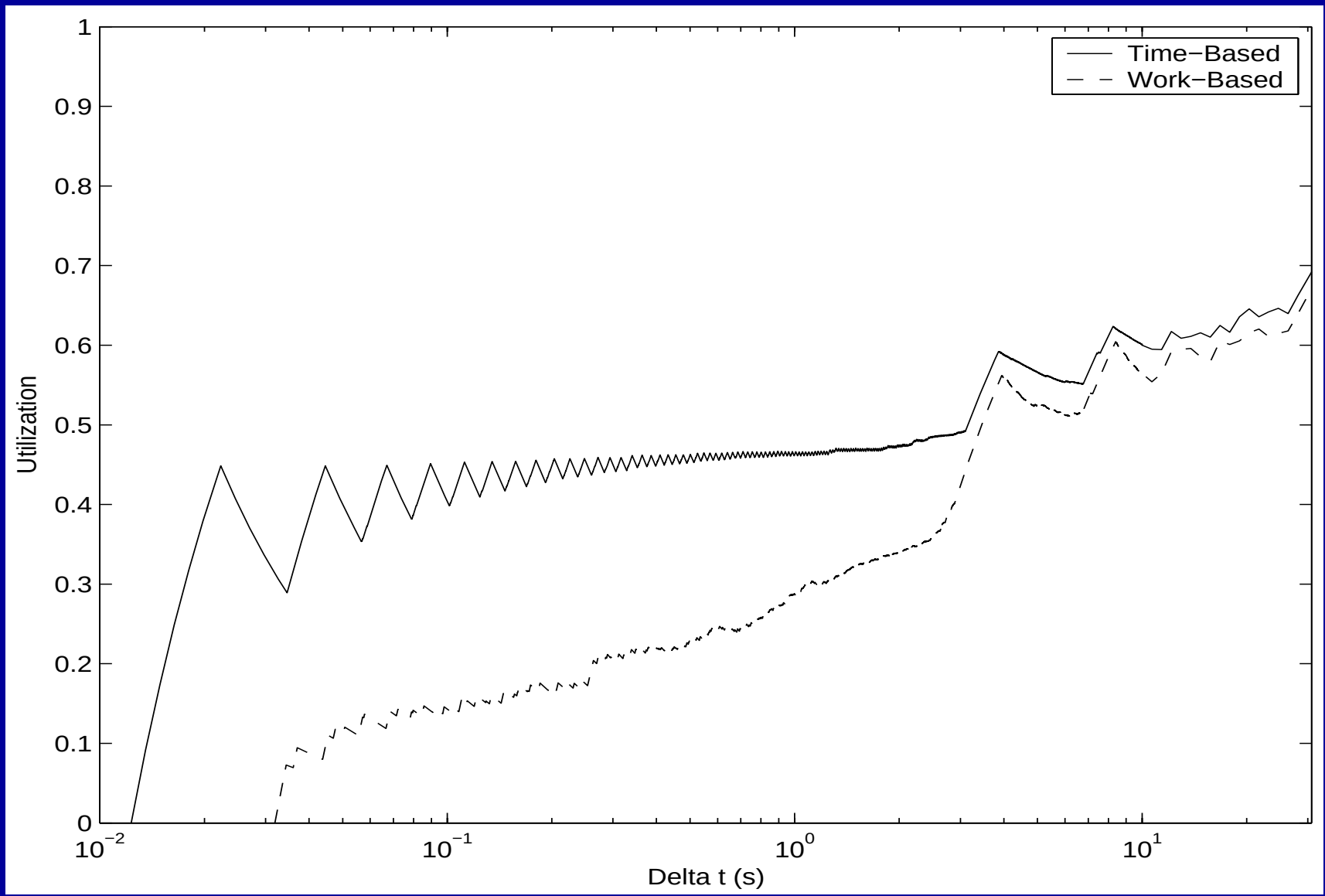
Triggering Collection



Factors in Space Consumption



MMU: javac


Time−Based
Work−Based
Utilization
Delta t (s)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
10^{-2}
10^{-1}
10^{0}
10^{1}