Garbage-First Garbage Collection by David Detlefs, Christine Flood, Steve Heller & Tony Printezis

Presented by Edward Raff

Motivational Setup

- Java Enterprise World
  - High end multiprocessor servers
  - Large heaps (that contain many live objects)
- Soft Real Time requirements
  - Needs to be responsive.
  - Let the user tell us exactly how responsive we need to be (specified desired max pause time in milliseconds)

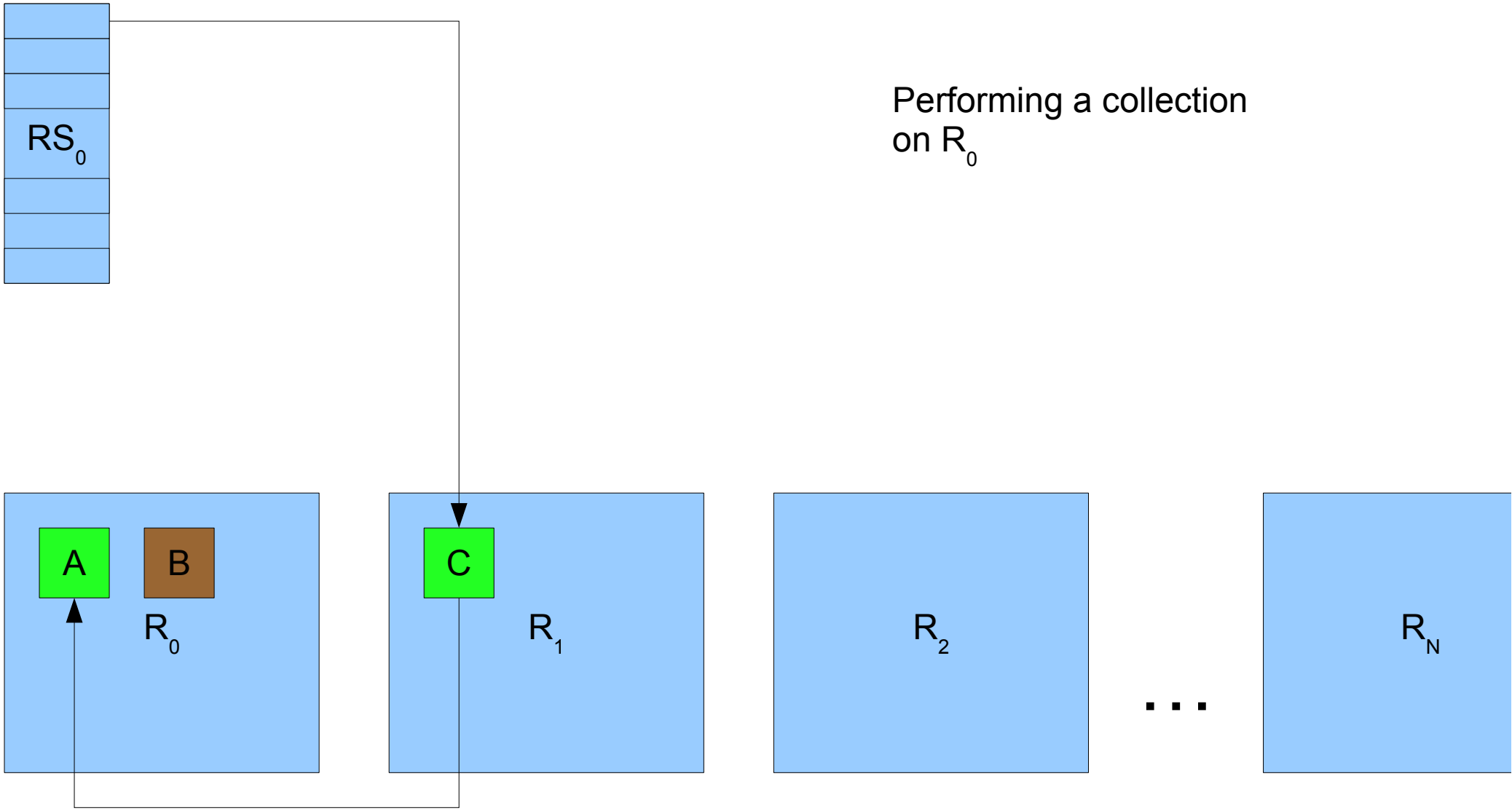
To Obtain

- Concurrent Collection
  - Still stop-the-world
- Parallel Collection
- High throughput
- “PAC”ish (Probably Approximately Correct) configurable pause times & frequency

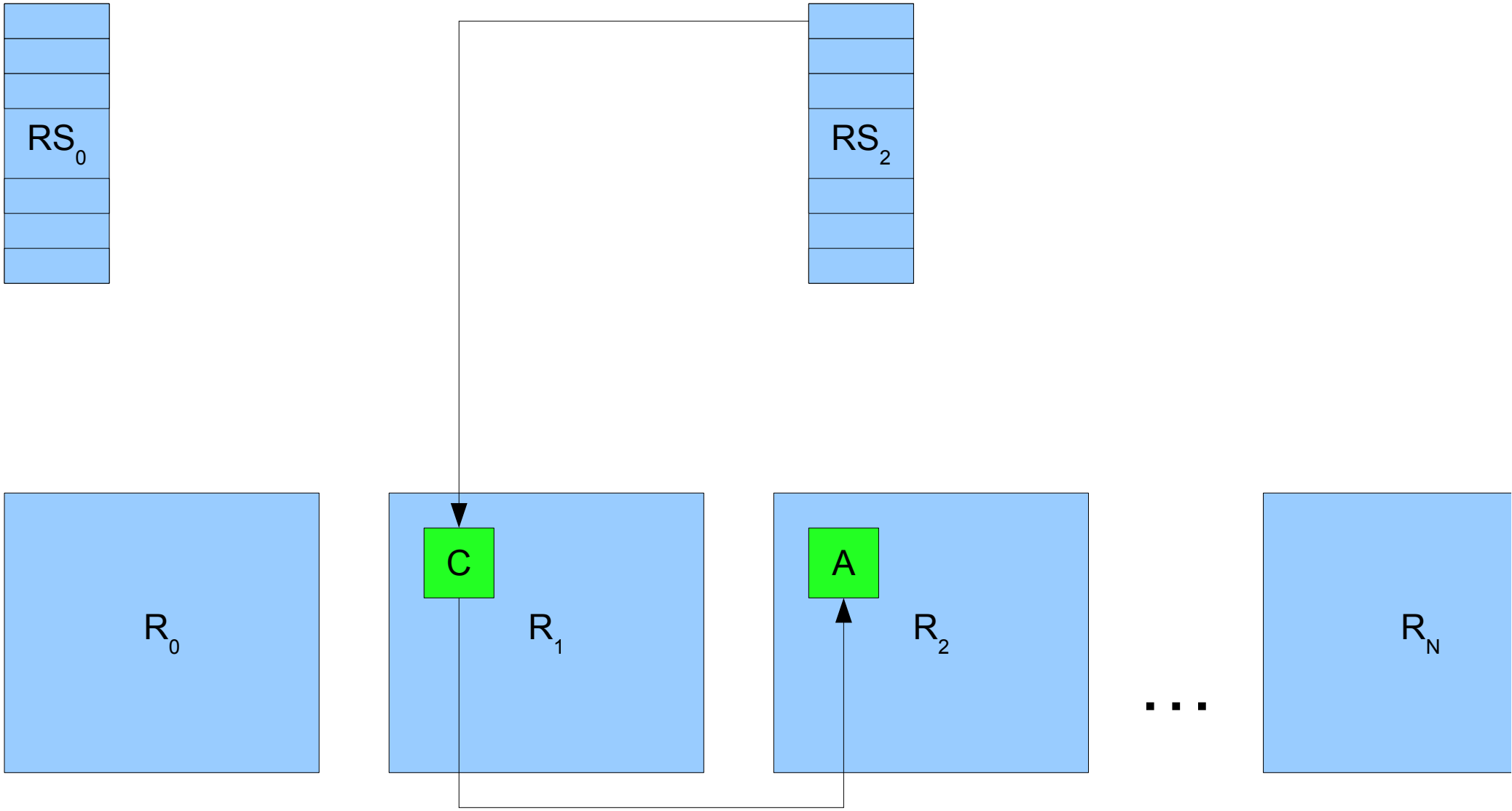
Method Set Up

- Many moving parts, discuss Heap layout first
  - Heap is subdivided into several regions of equal size
    - “humongous” objects (>= 3/4 region size) are allocated in a special area
  - Each region maintains its own Remembered Set (RS), which keeps track of all objects that point to an object in its own region
  - 2 sets of bitmaps for each region. 1-bit mark for each 64 bits in the region. One bitmap is for the current collection, the other is for the previous

Heap Macro View

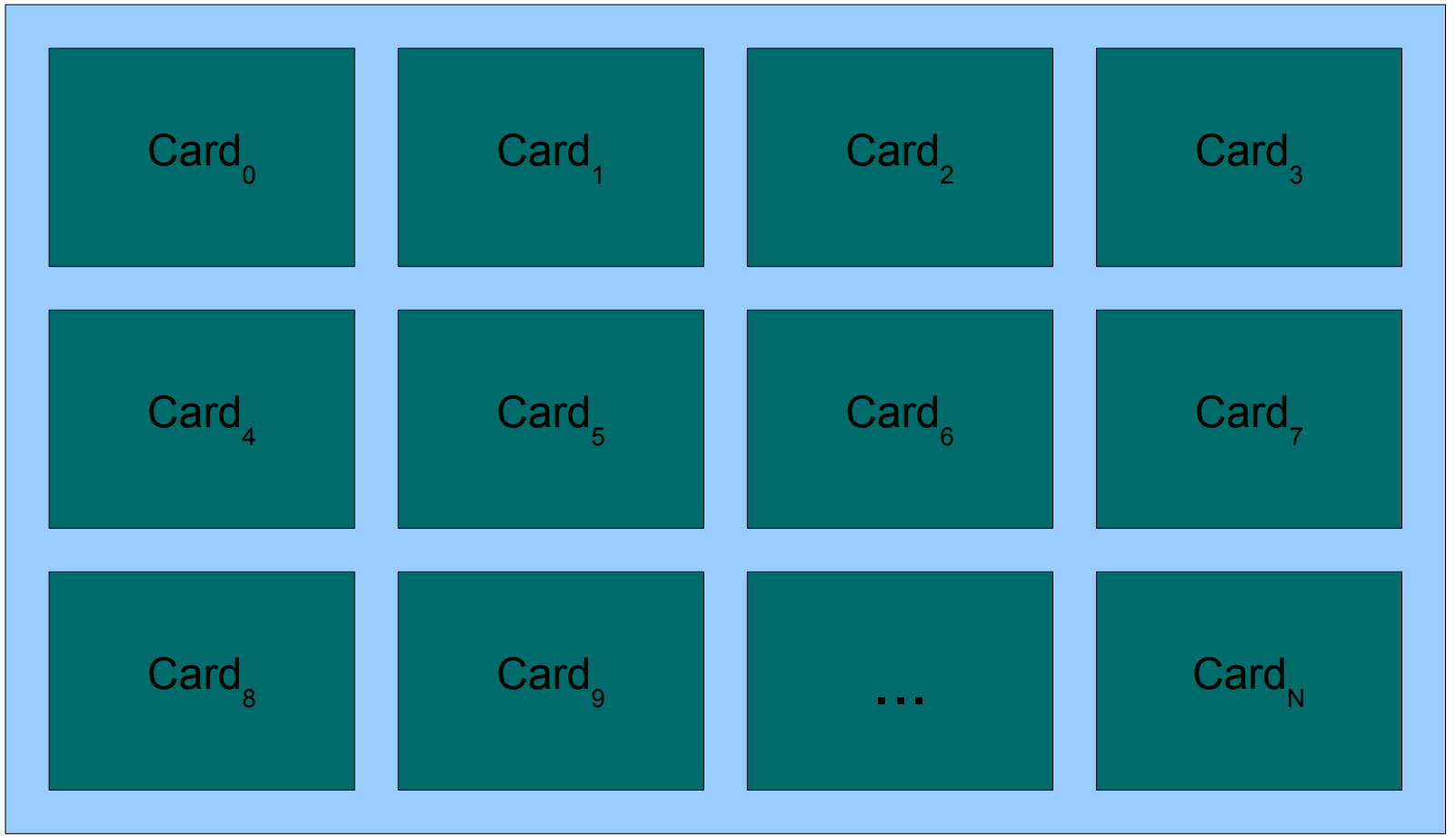


Heap Macro View: After Collection

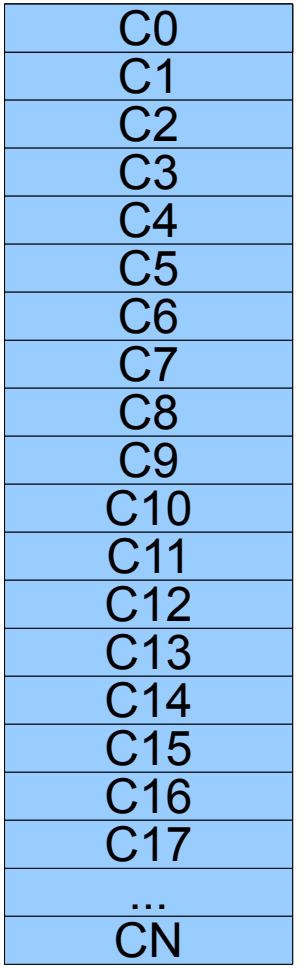


Heap: Region View

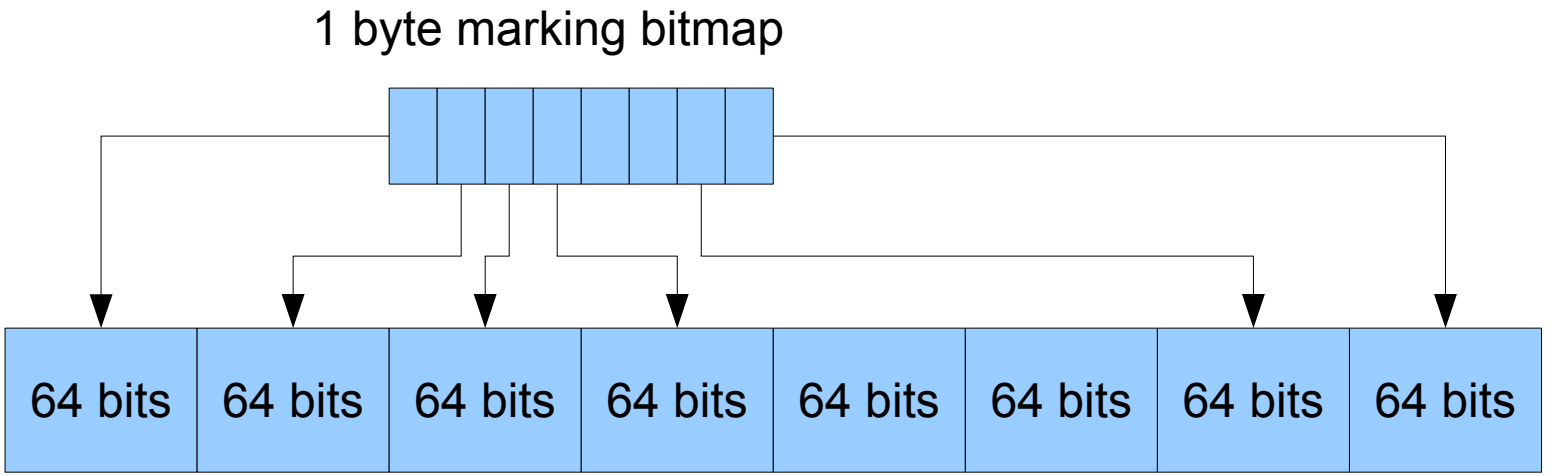
Each Region is composed of
several 512 byte *cards*



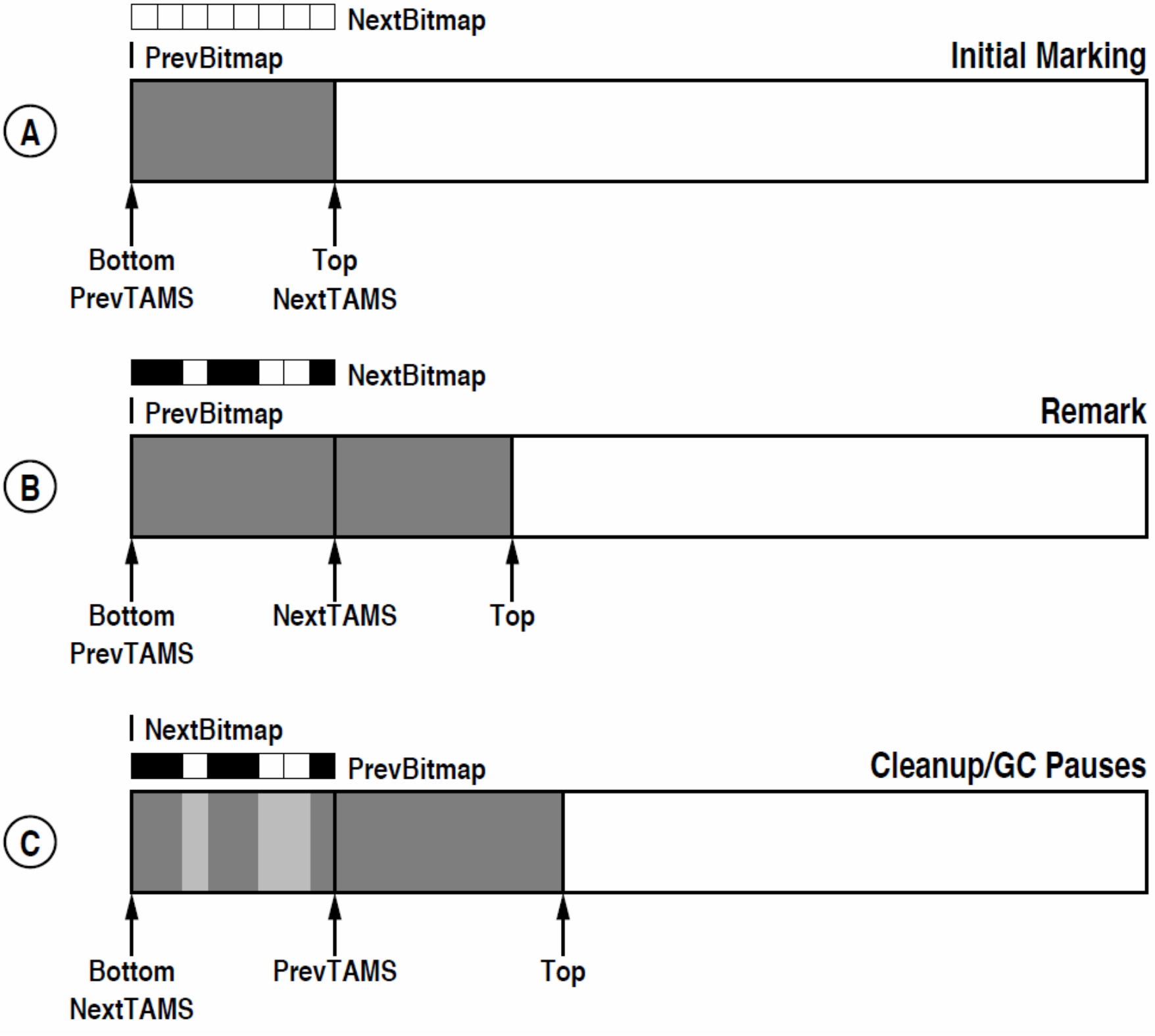
1 bit 'dirty'
marks for
each card

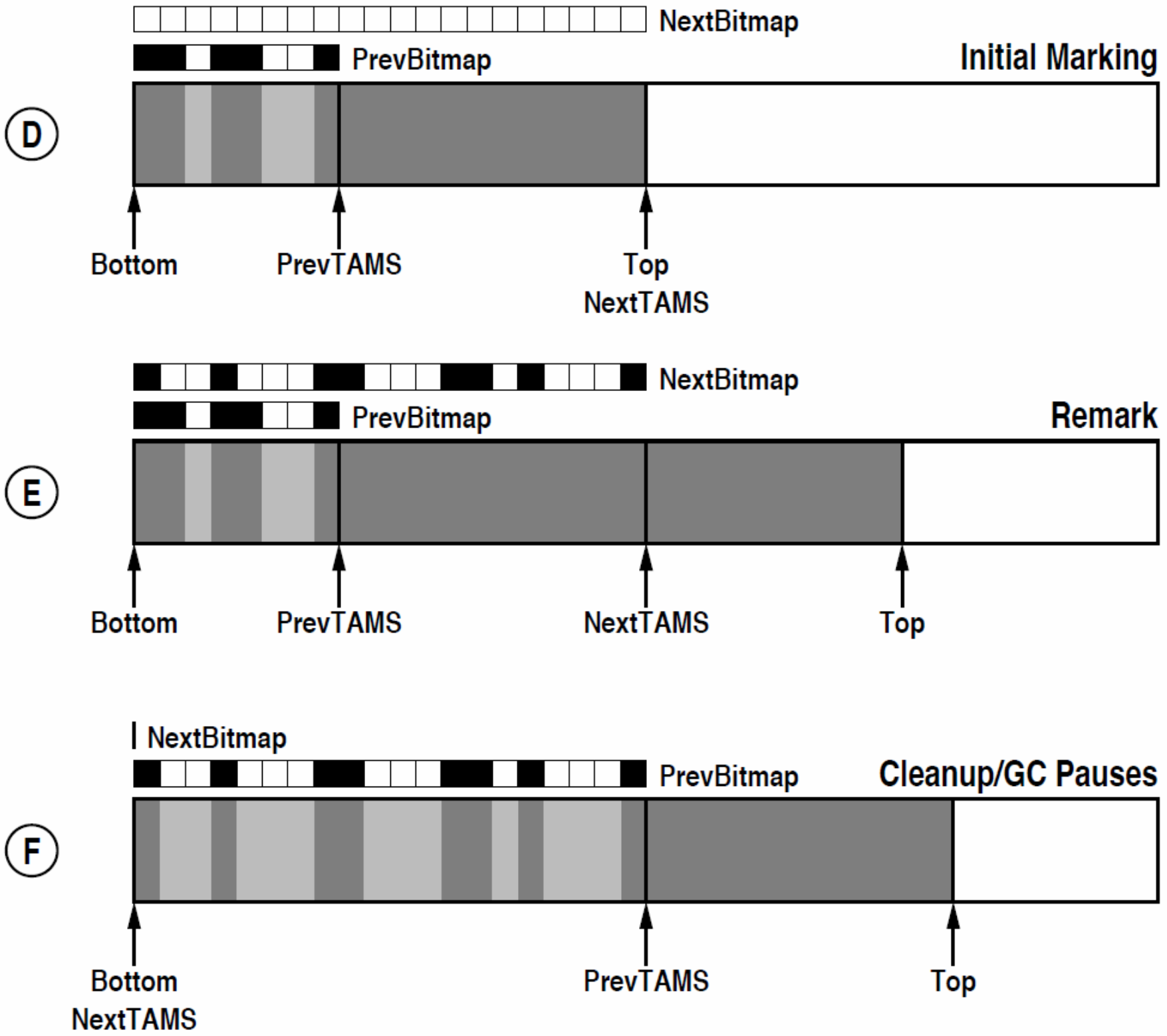


Card Micro View



Marks the sections that are live or dead


NextBitmap
PrevBitmap
Initial Marking
A
Bottom
PrevTAMS
Top
NextTAMS
NextBitmap
PrevBitmap
Remark
B
Bottom
PrevTAMS
NextTAMS
Top
NextBitmap
PrevBitmap
Cleanup/GC Pauses
C
Bottom
NextTAMS
PrevTAMS
Top


NextBitmap
PrevBitmap
Initial Marking
D
Bottom
PrevTAMS
Top
NextTAMS
NextBitmap
PrevBitmap
Remark
E
Bottom
PrevTAMS
NextTAMS
Top
NextBitmap
PrevBitmap
Cleanup/GC Pauses
F
Bottom
NextTAMS
PrevTAMS
Top

Write Barrier

- Write barrier needs to update the Remembered Sets
  - Ignore same region writes
  - If a card is already remembered (Dirty), ignore it
  - Ignore null writes
  - Use bit ops to perform fast region checks
- Marking also needs Write Barrier logic
  - Skip book keeping if the region is not being concurrently marked
  - Do not log nulls

Exploiting the Structure: Parallelism and Concurrency

- Snapshot of the heap when we stop the world to be used concurrently
- Regions can be marked by different threads during mutator execution concurrently with the mutators
- Regions can be collected in parallel
  - The bitmaps are important for this. Mutators alter the objects, collectors alter the bitmaps. Changes to the heap are made by the collector during stop-the-world phases
- Each mutator thread has a buffer of changes

Exploiting the Structure: High Throughput

- Can mark live and dead objects from the snapshot
  - Approximate, but it is a lower bound on the true value (anything added after the snapshot is considered 'live')
- We can use the lower bound to select the regions with the lowest liveness to collect.
  - Collecting regions with the largest amount of garbage first, hence – Garbage First
- By collecting the high garbage regions first, we maximize the memory collected and do not waste time in regions with no garbage

Exploiting the Structure: "PAC" pause times

- We can model the cost of performing a collection

$$Cost(cs) = V_{\text{fixed}} + U \cdot d + \sum_{\text{Region } r \in cs} (S \cdot rsSize(r) + C \cdot liveBytes(r))$$

cs : set of regions to collect

V_{fixed} : fixed cost, constant

U : average cost to scan a card

d : number of dirty cards that need to be updated

S : cost of scaning the remember set

C : cost per byte to evacuate a live object

Exploiting the Structure: “PAC” pause times, Continue

- Course initial estimates for these values, updated during execution
  - Keep track of standard deviation (σ) of the values as well
- Use model to select a number of blocks we can collect in the time requested, use the σ to be conservative

Generational?

- Concept of generations is semi constant, each region is a “generation”, and we treat them differently by how many objects die off in every generation. We treat them differently by mortality rates
- Paper considers generational by mandating all regions currently used for allocations to be part of the collected set.

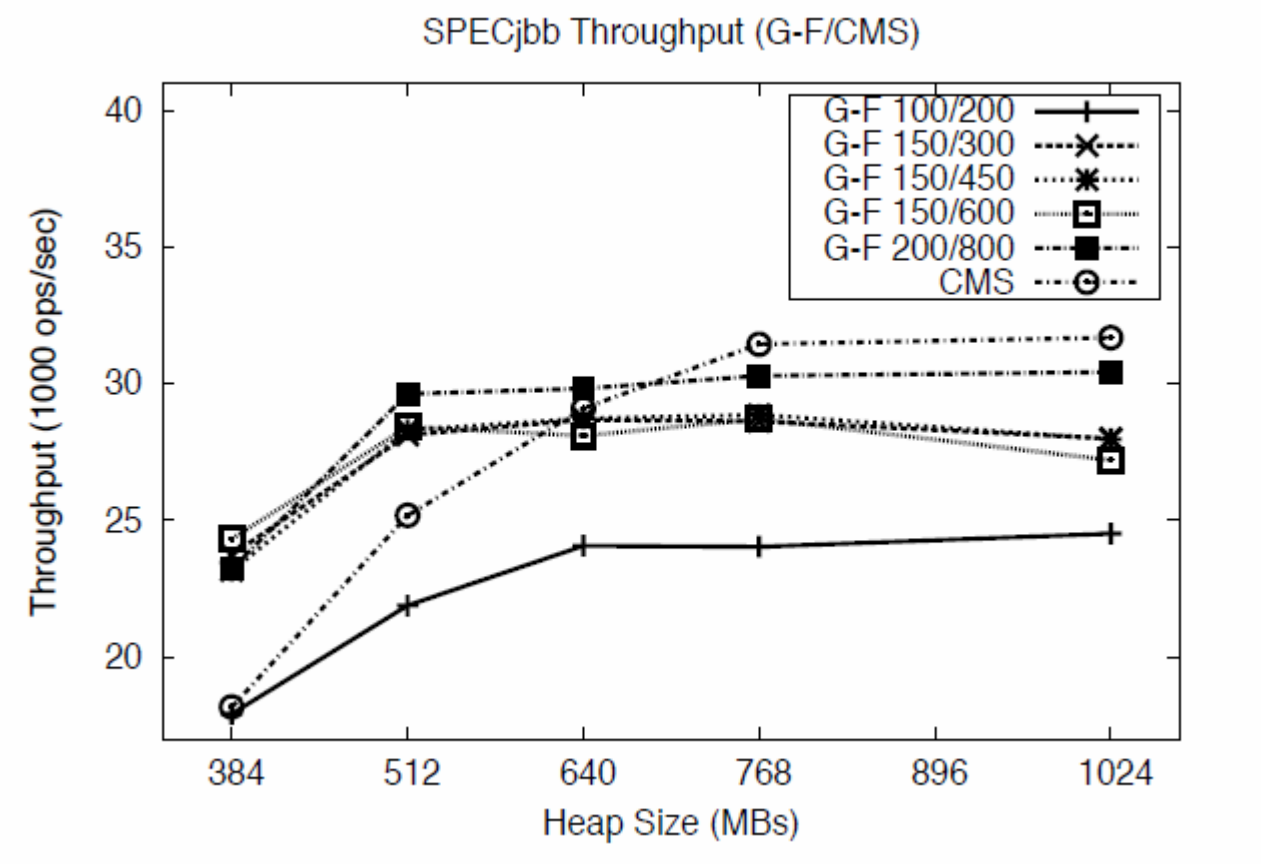
Extra Work Done

- Use the model to intelligently schedule a collection (part of meeting soft real time goal)
  - Use a Hard and Soft Limit to control when we give up procrastinating and force collections
- Special space for the popular objects, can reduce the RS size. Promote them to their own special region

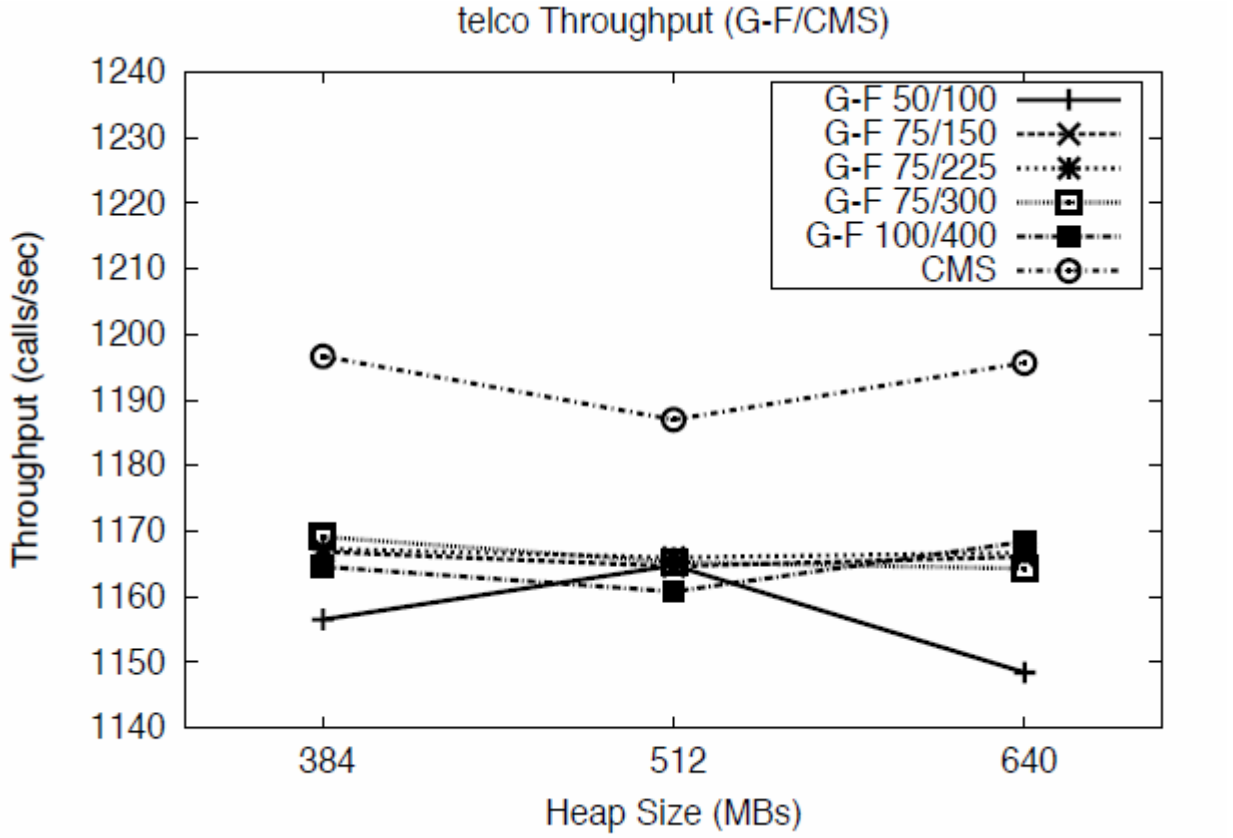
Results: Soft Real Time Compliance

Benchmark/ configuration	Soft real-time goal compliance statistics by Heap Size								
	V%	avgV%	wV%	V%	avgV%	wV%	V%	avgV%	wV%
SPECjbb	**512 M**			**640 M**			**768 M**		
G-F (100/200)	4.29%	36.40%	100.00%	1.73%	12.83%	63.31%	1.68%	10.94%	69.67%
G-F (150/300)	1.20%	5.95%	15.29%	1.51%	4.01%	20.80%	1.78%	3.38%	8.96%
G-F (150/450)	1.63%	4.40%	14.32%	3.14%	2.34%	6.53%	1.23%	1.53%	3.28%
G-F (150/600)	2.63%	2.90%	5.38%	3.66%	2.45%	8.39%	2.09%	2.54%	8.65%
G-F (200/800)	0.00%	0.00%	0.00%	0.34%	0.72%	0.72%	0.00%	0.00%	0.00%
CMS (150/450)	23.93%	82.14%	100.00%	13.44%	67.72%	100.00%	5.72%	28.19%	100.00%
Telco	**384 M**			**512 M**			**640 M**		
G-F (50/100)	0.34%	8.92%	35.48%	0.16%	9.09%	48.08%	0.11%	12.10%	38.57%
G-F (75/150)	0.08%	11.90%	19.99%	0.08%	5.60%	7.47%	0.19%	3.81%	9.15%
G-F (75/225)	0.44%	2.90%	10.45%	0.15%	3.31%	3.74%	0.50%	1.04%	2.07%
G-F (75/300)	0.65%	2.55%	8.76%	0.42%	0.57%	1.07%	0.63%	1.07%	2.91%
G-F (100/400)	0.57%	1.79%	6.04%	0.29%	0.37%	0.54%	0.44%	1.52%	2.73%
CMS (75/225)	0.78%	35.05%	100.00%	0.54%	32.83%	100.00%	0.60%	26.39%	100.00%

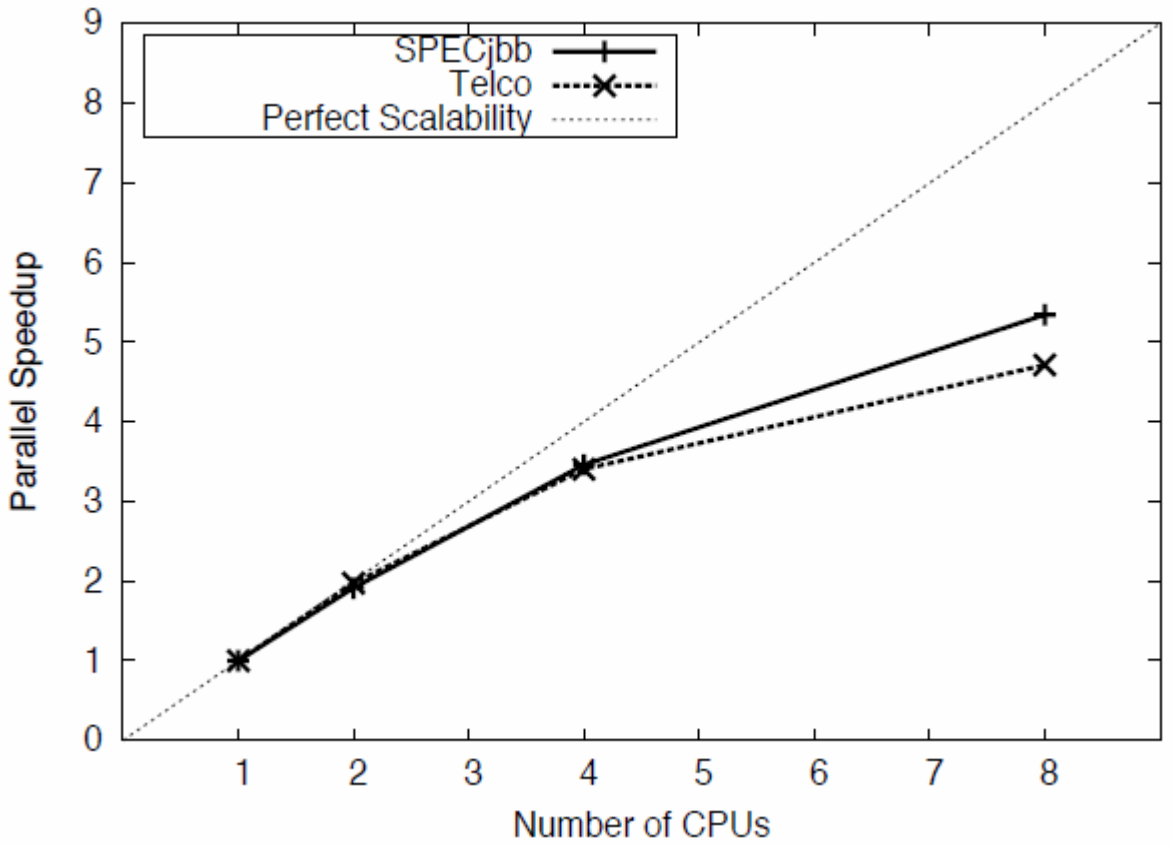
Throughput


SPECjbb Throughput (G-F/CMS)
Throughput (1000 ops/sec)
40
35
30
25
20
384
512
640
768
896
1024
Heap Size (MBs)
G-F 100/200
G-F 150/300
G-F 150/450
G-F 150/600
G-F 200/800
CMS

Throughput


telco Throughput (G-F/CMS)
Throughput (calls/sec)
1240
1230
1220
1210
1200
1190
1180
1170
1160
1150
1140
384
512
640
Heap Size (MBs)
G-F 50/100
G-F 75/150
G-F 75/225
G-F 75/300
G-F 100/400
CMS

Parallel Speedup (Stop the world activities)



Some Thoughts

- Very Memory Greedy Collector
  - Lots of meta data
  - Relies on excess memory to procrastinate collections and meet real time requirements
  - Is using the “client” JRE valid?
- What is the effect of not concurrently marking?
- What is the effect of a mutator heavily using all available threads when attempting to concurrently mark?

User Supplied Pause time

- The Soft Real Time goal requires the user to specify the desired maximum pause time (and optionally, a confidence in the form of standard deviations)
- Using the model, we could turn around and instead have the user specify throughput and a desired mutator overhead
  - Would this be helpful? Why not try a version that maximizes throughput?

Note about the σ = 1

- Chebyshev's inequality
  - Absolute lower bound probability 0.5 that a collection takes too long
- Normal Distribution
  - ≈ 0.16 probability that a collection takes too long (0.07 for σ = 1.5)
- Results are in the range [0.008, 0.043]
  - Over long runs, we can model the differences and attempt to rescale so that its probability matches N(0, 1.5) - avoiding super short collections on a better behaved application

Mixing the JIT & GC

- Could we perform new optimizations using the information form both the JIT & GC
- There are methods that are hot in the JIT sense (take long time to execute / called often)
- Define hot methods from the GC sense (perform lots of allocations)
- Make the JIT perform optimizations based on hot GC methods
  - Make certain methods remember the region they were allocating into (and lock the region from others)

Illustration: Normal

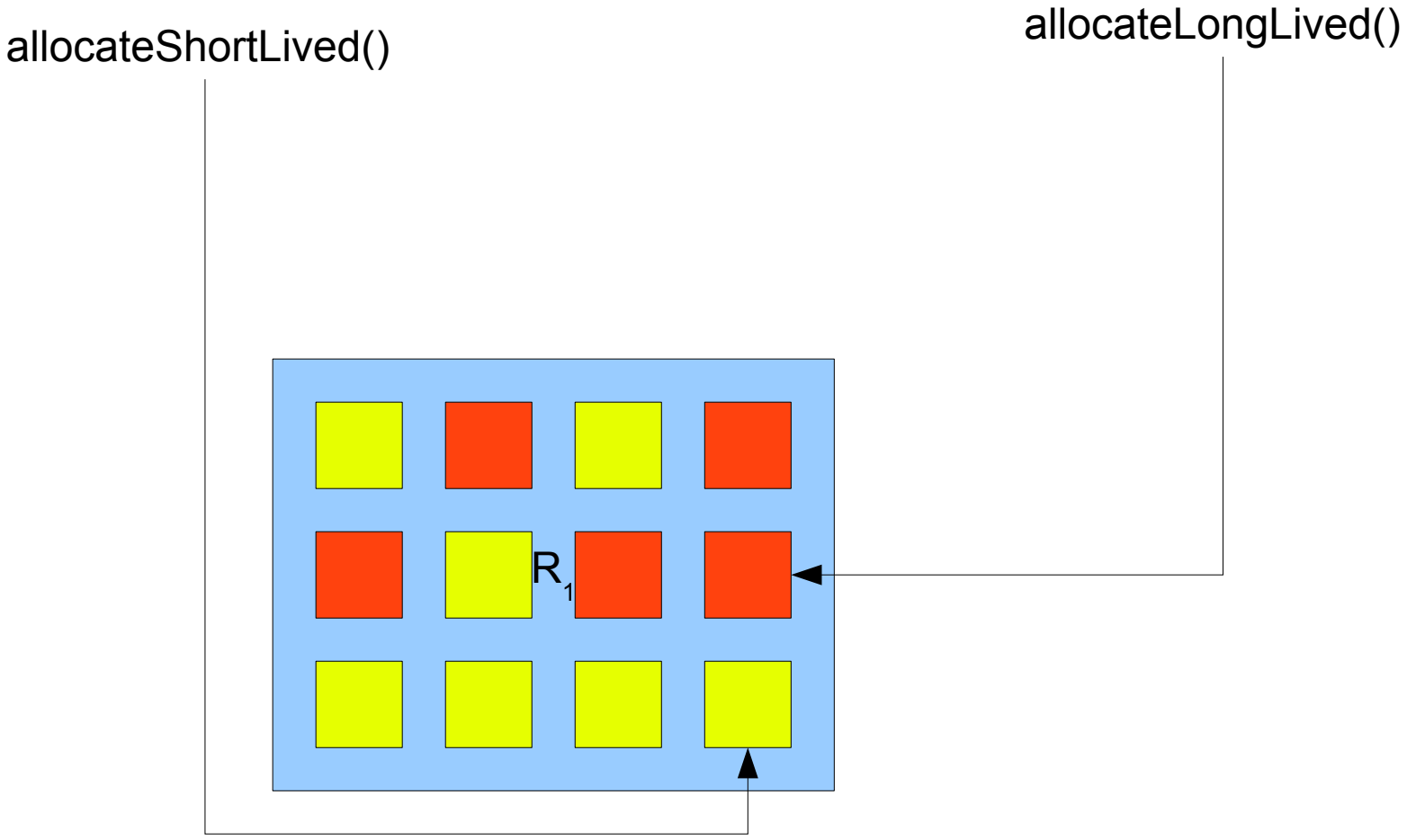
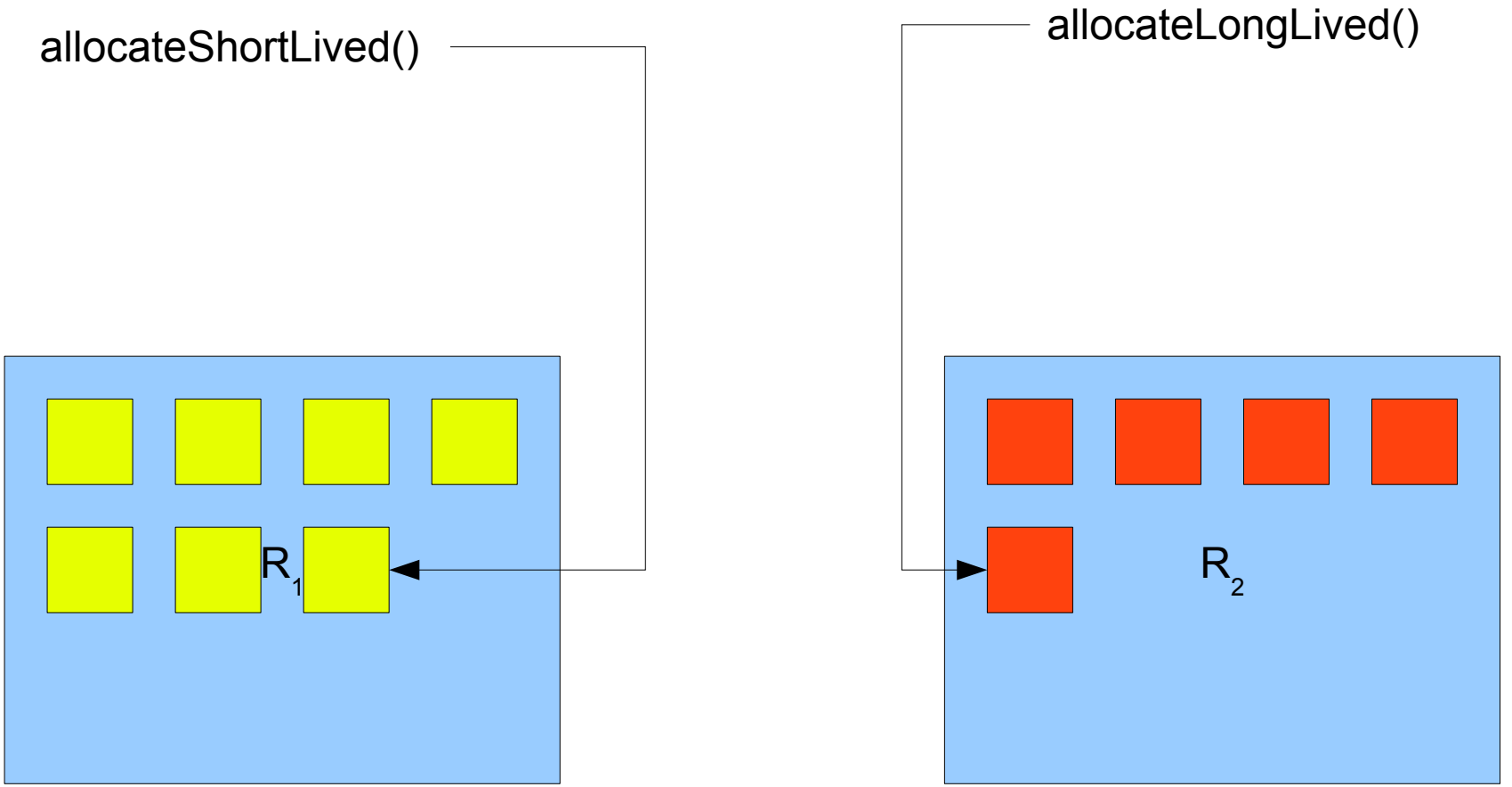


Illustration: Optimized



Mixing the JIT & GC

- By identifying methods by allocation type, we can perform population segregation. We don't need to keep track of object life times, but lifetime consistency given the creating method
  - Var[p(object mortality | method)] < C
- Heuristic hot methods (according to the JIT) are the only methods called often enough to make the optimization meaningful
  - Hot methods have more data
  - JIT is already doing extra work for them