IBM Research

Garbage Collection for Embedded Systems

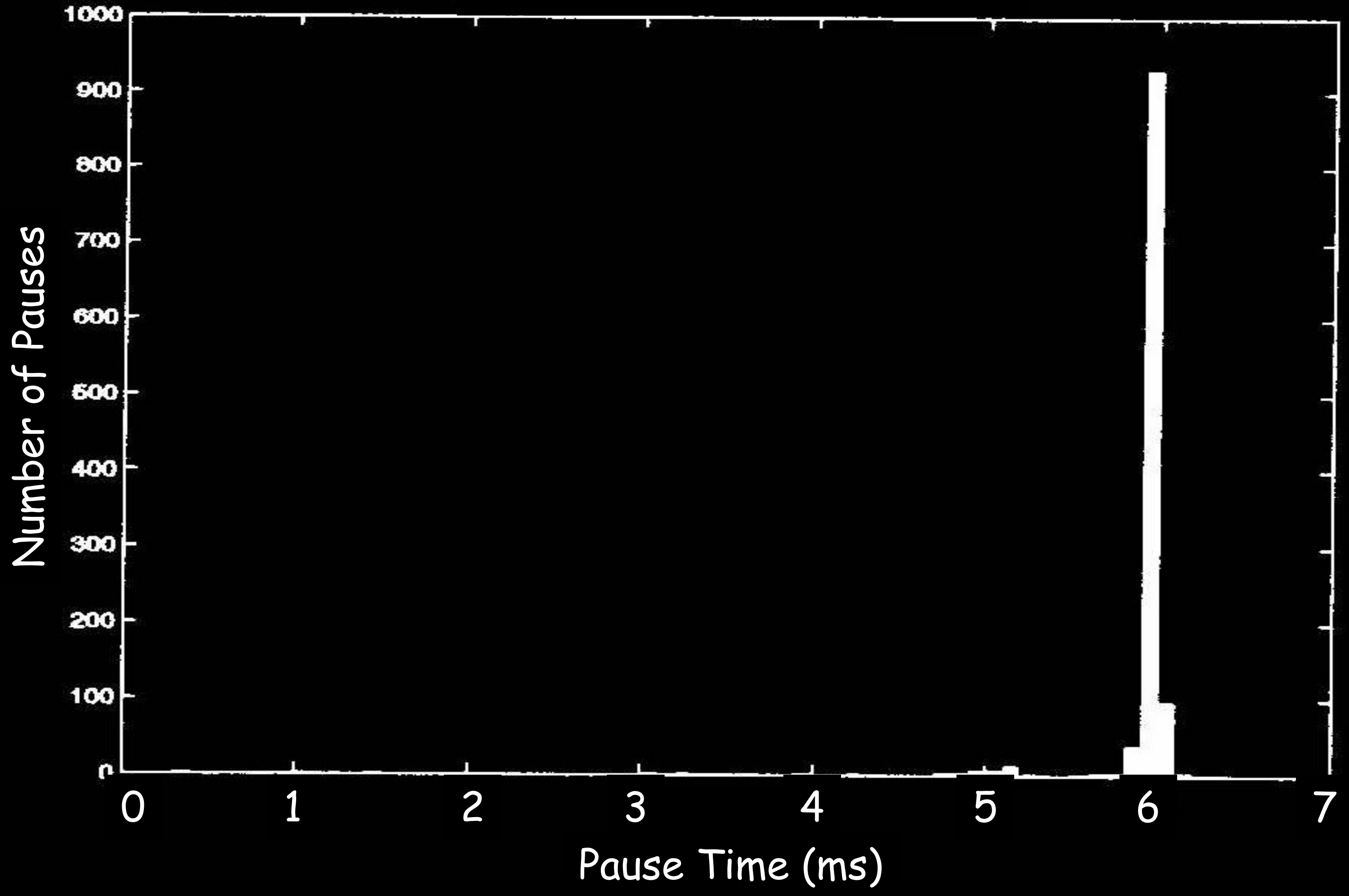
David F. Bacon, Perry Cheng, David Grove
IBM T.J. Watson Research Center

EMSOFT: 28 September 2004

Background: Hard Real-time Garbage Collection for Java

- Metronome collector
  - Implemented in IBM Research Virtual Machine
  - High-quality optimizing JIT
- Time-based vs. work-based scheduling
  - aka Time-triggered vs. event-triggered
  - But provable convergence
    - Collection finishes before application runs out of memory

Metronome Pause Times (SPECjvm javac)



Second Generation Metronome

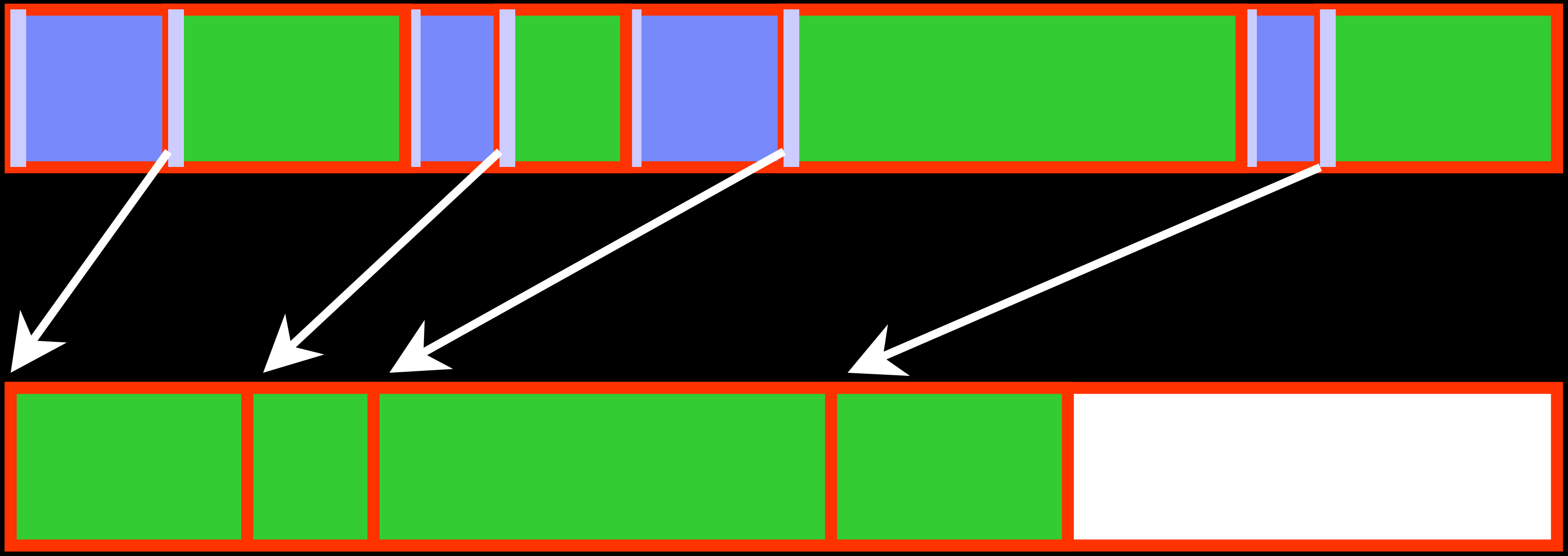
- Implementation in IBM J9 JVM Product
  - Targets embedded systems (400K ROM, 64K RAM, 32K Heap)
  - Goals: sub-millisecond response time; higher throughput
- Stage 1 (this talk): Evaluate Embedded Collectors
  - Build standard ("stop-the-world") collectors in J9 JVM
  - Introduce our code base (virus approach to tech transfer)
  - Gain credibility with developers (they think we're dilettantes)
- How does embedded environment change standard assumptions?

What's Different in Embedded Garbage Collection?

- Code size of collector matters (~30KB)
  - Complex optimizations ruled out due to ROM constraints
- Code size of generated code matters
  - Inlining allocation, write barriers, etc. has cost
- Memory overhead constraints (16KB – 4MB)
  - Semispace collectors waste 50%
  - Generational wastes nursery, remembered set, code size
- Compaction required
  - May run at close to theoretical minimum heap size
  - Must be able to run for a very long time
- Reliability
  - Premium on simple algorithms with simple invariants

Garbage Collectors

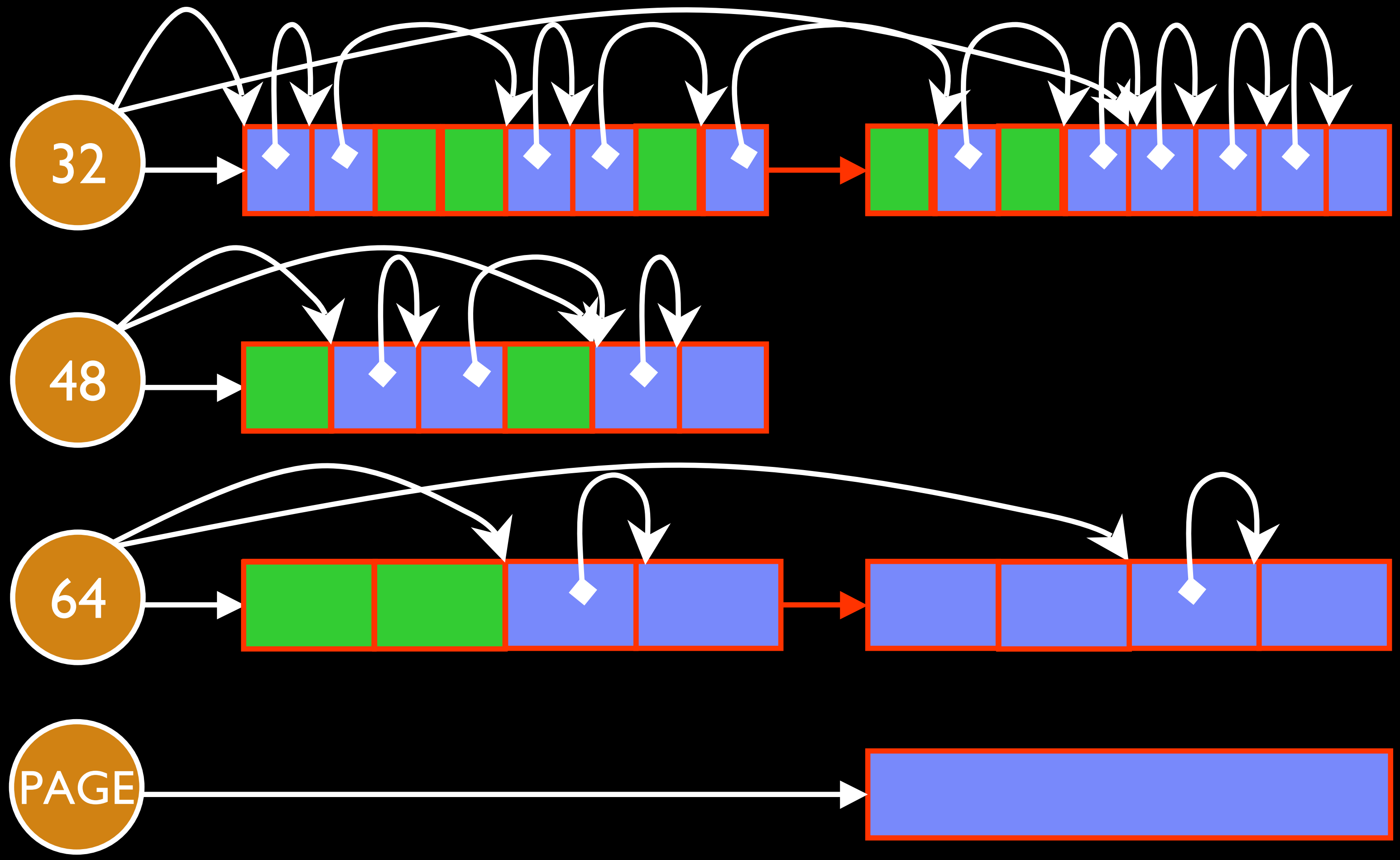
Mark-Compact Algorithm (MC)



Comments about Mark-Compact

- Advantages
  - Simple and small algorithm (40 years old!)
    - easy to debug and maintain
  - Time and space both predictable
  - Robust – no pathological fragmentation
- Disadvantages
  - Space overhead (8%) due to forwarding pointers
  - Compaction must occur on every GC (significant time increase)
  - Cannot take advantage of dynamic abundance of space
  - Not Incrementalizable: can not be base for real-time collector
- Not used in modern Java virtual machines

Paged Mark/Sweep/Defragment Algorithm (PMSD)



Comments about Paged Mark/Sweep/Defragment

- Advantages
  - Performs defragmentation only as needed
  - Performs well when locality of size is high
    - Almost no defragmentation needed
  - Compatible with real-time / incremental collection
- Disadvantage
  - More complicated (higher maintenance cost)
  - Collector code (ROM) is bigger

Measurements

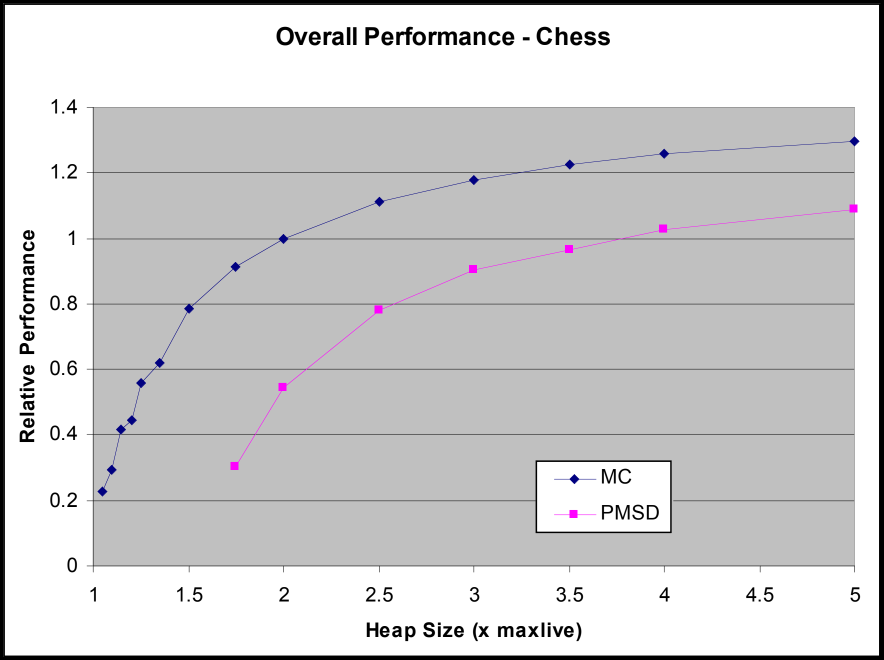
Implementation

- MC and PMSD in J9 J2ME
  - Both about equally optimized
  - Target Linux/ARM and Win32/x86
- Hardware
  - Intel XScale PXA 255 (400 MHz) – Sharp Zaurus SL-6000
  - Intel Pentium III M (1.2 GHz) – IBM Thinkpad T23
    - Usually entire heap fits in L2 Cache

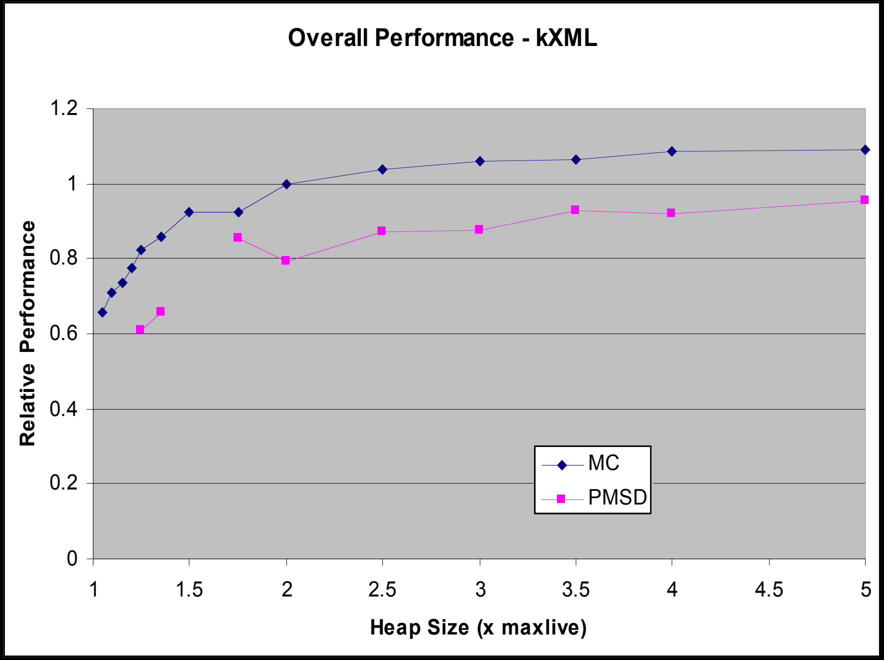
MC vs. PMSD: Server (SPECjvm)

- Live data size 10 - 200 MB
- Heap size 40 MB – 1 GB
- Each collector suffers about 8% space overhead
  - Each wins on about half the benchmarks
  - But can eliminate almost all space overhead in Mark/Compact
    - (later in talk)
- PMSD is faster for larger heap ratios
- Many server systems use PMS (not PMSD)
  - No defragmentation!
  - Fragmentation solution: buy more RAM

MC vs. PMSD: Embedded (EEMBC)



Chess Benchmark

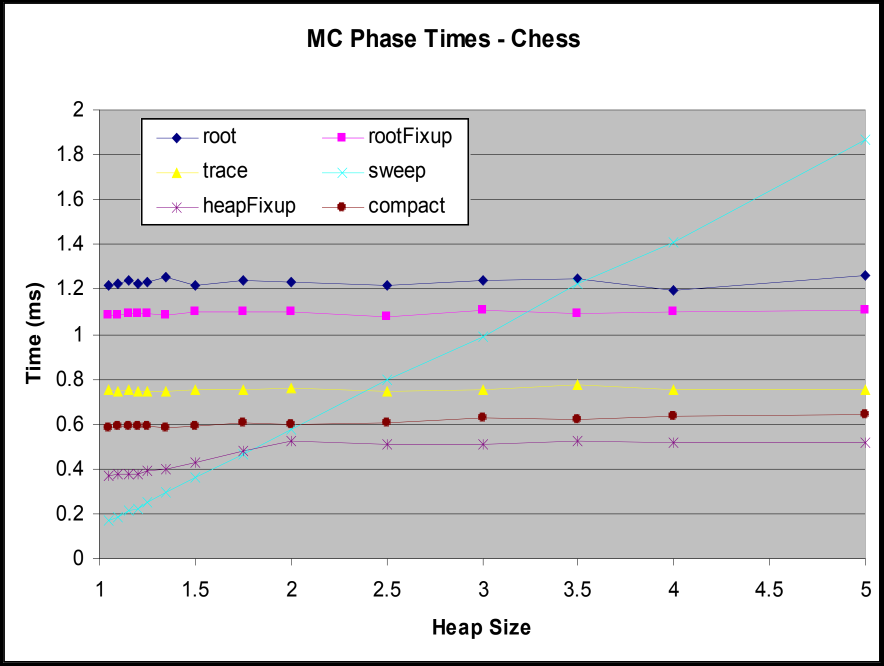


kXML Benchmark

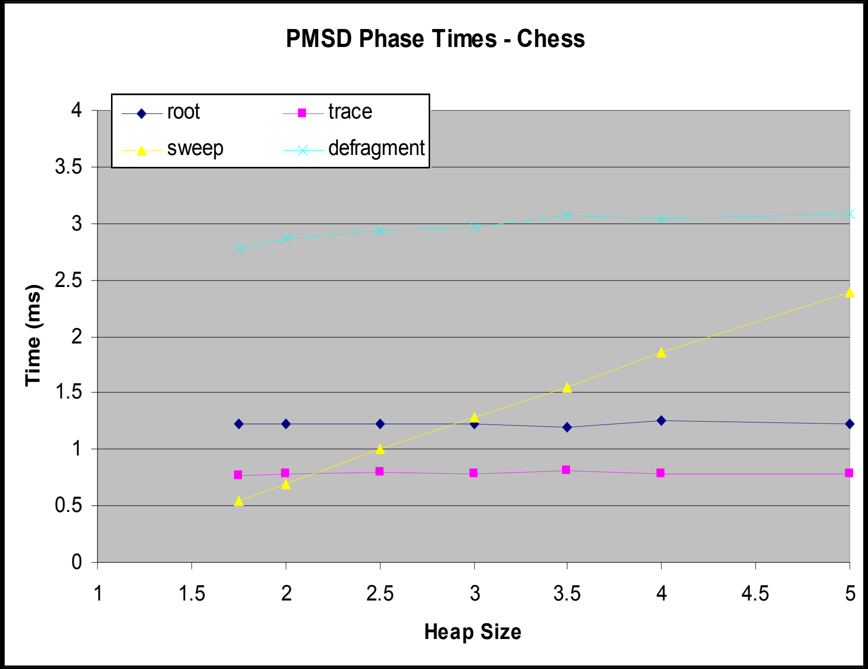
MC vs. PMSD: Embedded (EEMBC)

- Live data size 31 – 224 KB
- Heap size 34 KB – 700 KB
- For medium allocation rate (PNG, Crypto)
  - 85% of peak with only 1.3 times live data size
- For low allocation rate (Parallel, RegExp)
  - 90% of peak with only 1.05 times live data size

Collector Cost: MC vs. PMSD - About Equal



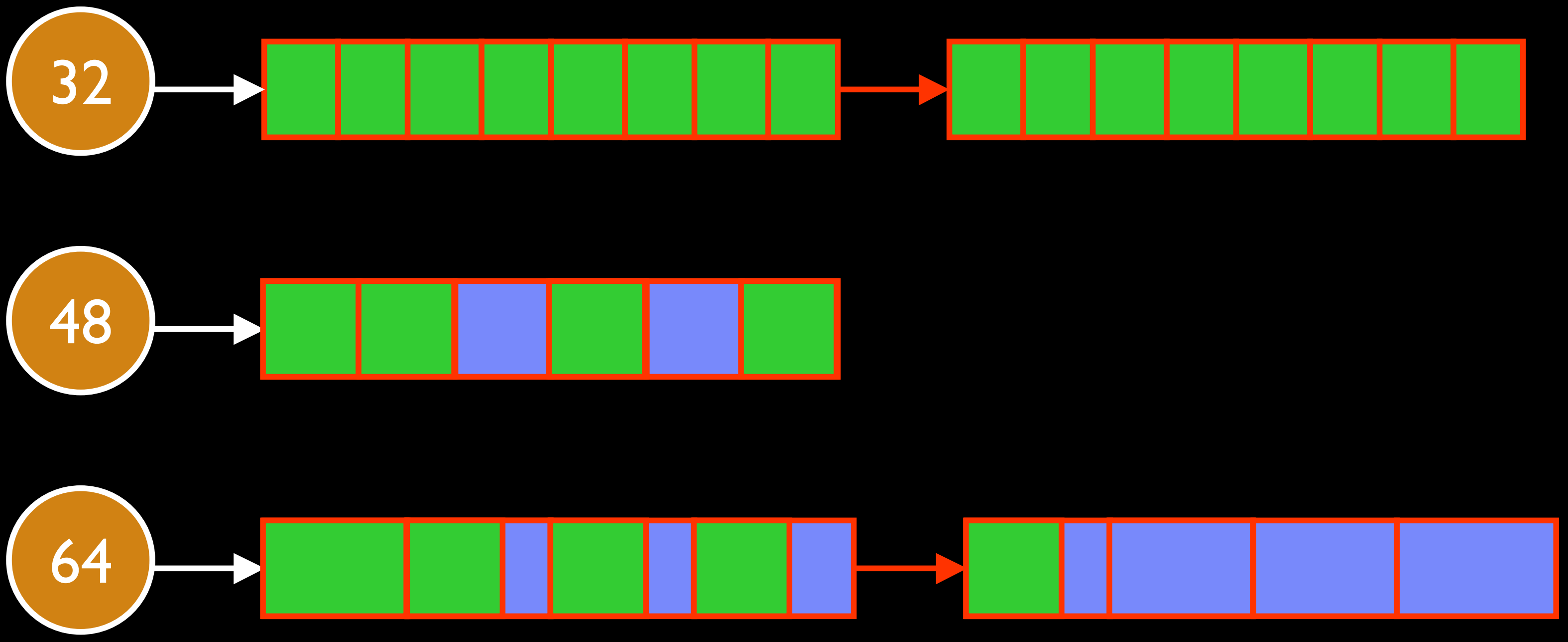
Mark-Compact Collector



Paged Mark/Sweep/Defragment Collector

Lessons

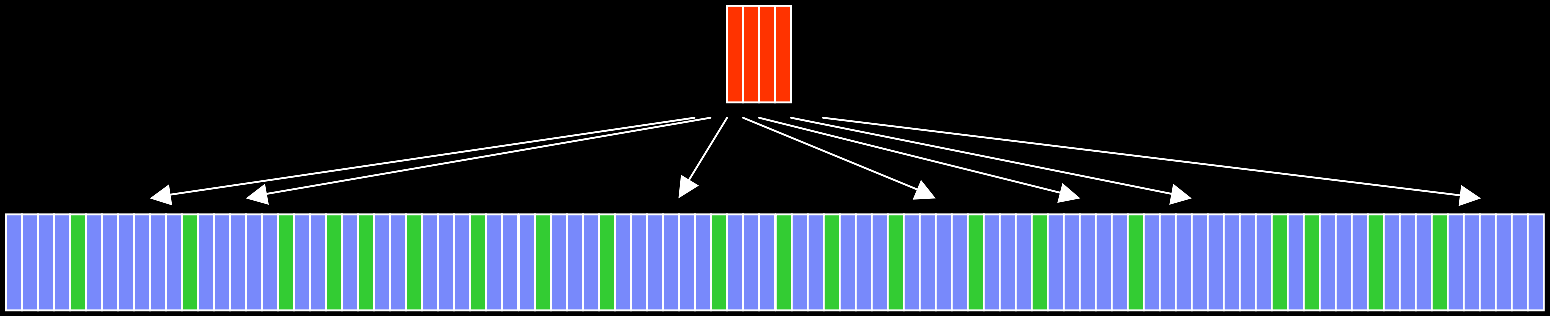
Lesson: For PMSD, Small Object Fragmentation Kills



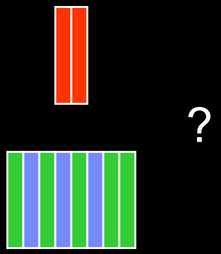
- Scale down page size?
  - Overhead kills. Can't scale down linearly with heap size

Lesson: For PMSD, Large Object Fragmentation Kills

- Allocation in large (server) heap
  - Ratio of big objects to heap size 100:1 to 10,000:1
  - Probability of big enough free region is high



- Allocation in small (embedded) heap
  - Ratio of big objects to heap size is around 16:1; Probability is low



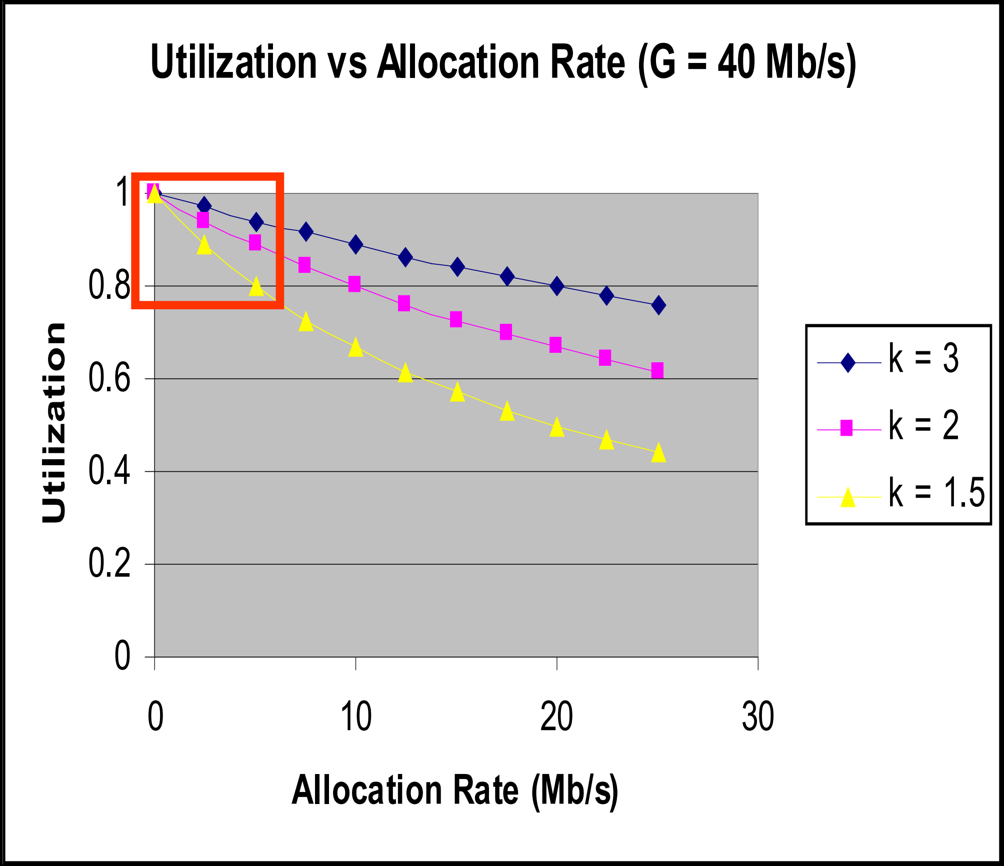
Lesson: Generations Less Useful in Embedded

- Relative allocation rate is usually lower
  - in small heaps
  - in embedded applications
- Nursery slows allocation into main heap
  - less of a factor if already low
  - less benefit if nursery small (not enough time to die)
  - Entire heap is smaller than most nurseries
- Nursery reduces ability to run in very tight heaps
  - Nursery itself "fragments" the memory

Why is Generational Less Effective?

- Assume
  - Steady state live data
  - K = heap size
    - in multiples of live data
  - A = allocation rate
    - Application "load"
  - G = GC tracing rate
    - Collector speed

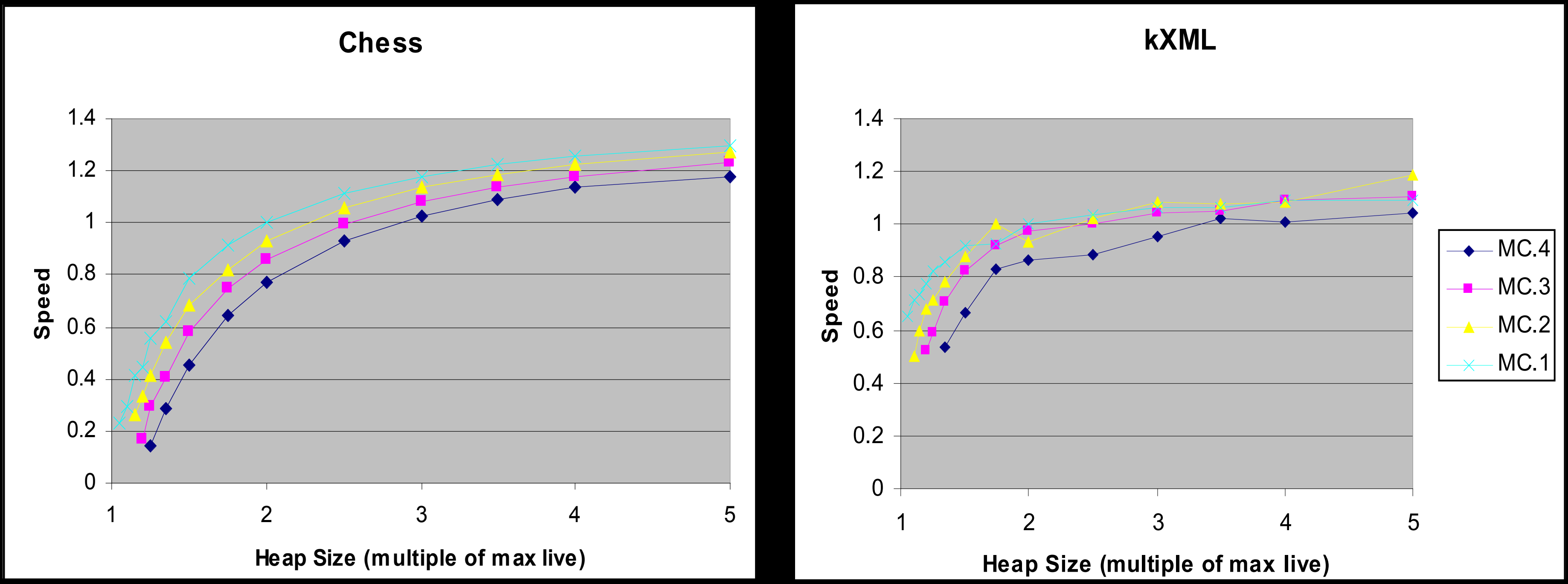
- Utilization
  - u = (k-1) / ((k-1) + (A/G))



Object Compression (details in paper)

- J9 originally required 4 words/object
- New techniques
  - Forwarding pointer compression (RBT)
  - "0-bit" hash code (Mashtable)
- Highly compact, high performance object model
  - 1 word/object
  - Almost never any runtime overhead

Performance Impact of Object Compression



Conclusions and Status

- Embedded systems very sensitive to GC design
  - Non-linear scaling effects
  - Below 1 MB: Use non-generational Mark-Compact
  - 1 – 8 MB: Use Generational Mark-Compact
  - Above 8 MB: Use Generational PMSD
- Need to study implications for power
  - Spend power on RAM or collecting more often?
- Good garbage collection can beat C!
  - 1 word/object less than malloc/free
  - No fragmentation for MC; can be large for malloc/free
- MC Collector and Object Compression Shipped
  - Interpreter-based generational Metronome working

Questions?

(or lunch?)