Flexible Reference-Counting-Based Hardware Acceleration for Garbage Collection

José A. Joao*
Onur Mutlu‡
Yale N. Patt*

* HPS Research Group
University of Texas at Austin

‡ Computer Architecture Laboratory
Carnegie Mellon University
Motivation: Garbage Collection

- Garbage Collection (GC) is a key feature of Managed Languages
  - Automatically frees memory blocks that are not used anymore
  - Eliminates bugs and improves security

- GC identifies dead (unreachable) objects, and makes their blocks available to the memory allocator

- Significant overheads
  - Processor cycles
  - Cache pollution
  - Pauses/delays on the application
Software Garbage Collectors

- **Tracing collectors**
  - Recursively follow every pointer starting with global, stack and register variables, scanning each object for pointers
  - Explicit collections that visit all live objects

- **Reference counting**
  - Tracks the number of references to each object
  - Immediate reclamation
  - Expensive and cannot collect cyclic data structures

- **State-of-the-art: generational collectors**
  - Young objects are more likely to die than old objects
  - Generations: nursery (new) and mature (older) regions
Overhead of Garbage Collection

![Graph showing the overhead of garbage collection for different applications and heap sizes.](image-url)
Hardware Garbage Collectors

- Hardware GC in general-purpose processors?
  - Ties one GC algorithm into the ISA and the microarchitecture
  - High cost due to major changes to processor and/or memory system
  - Miss opportunities at the software level, e.g. locality improvement

- Rigid trade-off: reduced flexibility for higher performance on specific applications

- Transistors are available
  - Build accelerators for commonly used functionality
  - How much hardware and how much software for GC?
Our Goal

- Architectural and hardware acceleration support for GC
  - Reduce the overhead of software GC
  - Keep the flexibility of software GC
  - Work with any existing software GC algorithm
Basic Idea

- Simple but incomplete hardware garbage collection until the heap is full
- Software GC runs and collects the remaining dead objects
- Overhead of GC is reduced
Hardware-assisted Automatic Memory Management (HAMM)

- Hardware-software cooperative acceleration for GC
  - Reference count tracking
    - To find dead objects without software GC
  - Memory block reuse handling
    - To provide available blocks to the software allocator
  - Reduce frequency and overhead of software GC

- Key characteristics
  - Software memory allocator is in control
  - Software GC still runs and makes high-level decisions
  - HAMM can simplify: does not have to track all objects
ISA Extensions for HAMM

- Memory allocation
  - REALLOCMEM, ALLOCMEM

- Pointer tracking (store pointer)
  - MOVPTR, MOVPTROVR
  - PUSHPTR, POPPTR, POPPTROVR

- Garbage collection
Overview of HAMM

LD/ST Unit

L1 RCCB

RC updates

Block address

L1 ABT

Core 1

…

Core N

L2 RCCB

L2 ABT

CPU Chip 0

Core 0

Available Block Table (ABT)

Reusable blocks

RC

Live objects

CPU Chip 1

…

CPU Chip M

Main memory
Modified Allocator

\[ \text{addr} \leftarrow \text{REALLOCMEM size} \]

if (addr == 0) then

// ABT does not have a free block \( \rightarrow \) regular software allocator

\[ \text{addr} \leftarrow \text{bump\_pointer} \]

\[ \text{bump\_pointer} \leftarrow \text{bump\_pointer} + \text{size} \]

... 

else

// use address provided by ABT

end if

// Initialize block starting at \text{addr}

\[ \text{ALLOCMEM object\_addr, size} \]
Example of HAMM

LD/ST Unit
incRC A

ALLOCMEM A, size

Core

CPU Chip

Main memory

L2 RCCB

Available Block Table (ABT)

Reusable blocks

RC

RC

A

0

L2 ABT

L1 ABT

L1 Reference Count Coalescing Buffer (RCCB)

RC updates

Block address

A: 1
Example of HAMM

L1 Reference Count Coalescing Buffer (RCCB)

LD/ST Unit

Block address

L1 ABT

L2 ABT

L2 RCCB

CPU Chip

Available Block Table (ABT)

Reusable blocks

RC

RC

dead

Main memory

Core

Available Block Table (ABT)
Example of HAMM

LD/ST Unit

RC updates

Block address

prefetch

L1 ABT

L2 ABT

L2 RCCB

CPU Chip

Available Block Table (ABT)

Main memory

RC

RC

dead

Prefetch
ISA Extensions for HAMM

- Memory allocation
  - REALLOCMEM, ALLOCMEM

- Pointer tracking (store pointer)
  - MOV PTR, MOVPTROVR
  - PUSH PTR, POPPTR, POPPTROVR

- Garbage collection
  - FLUSHRC
Methodology

- Benchmarks: DaCapo suite on Jikes Research Virtual Machine with its best GC, GenMS

- Simics + cycle-accurate x86 simulator
  - 64 KB, 2-way, 2-cycle I-cache
  - 16 KB perceptron predictor
  - Minimum 20-cycle branch misprediction penalty
  - 4-wide, 128-entry instruction window
  - 64 KB, 4-way, 2-cycle, 64B-line, L1 D-cache
  - 4 MB, 8-way, 16-cycle, 64B-line, unified L2 cache
  - 150-cycle minimum memory latency

- Different methodologies for two components:
  - GC time estimated based on actual garbage collection work over the whole benchmark
  - Application: cycle-accurate simulation with microarchitectural modifications on 200M-instruction slices
GC Time Reduction

![Bar Chart]

- 1.5x minHeap
- 2x minHeap
- 2.5x minHeap
- 3x minHeap
Since GC time is reduced by 29%, HAMM is a win
Why does HAMM work?

- HAMM reduces GC time because
  - Eliminates collections: **52%/50%** of nursery/full-heap
  - Enables memory block reuse for **69%** of all new objects in nursery and **38%** of allocations into older generation
  - Reduces GC work: **21%/49%** for nursery/full-heap

- HAMM does not slow down the application significantly
  - Maximum L1 cache miss increase: **4%**
  - Maximum L2 cache miss increase: **3.5%**
  - HAMM itself is responsible for only **1.4%** of all L2 misses
Conclusion

- Garbage collection is very useful, but it is also a significant source of overhead
  - Improvements on pure software GC or hardware GC are limited

- We propose HAMM, a cooperative hardware-software technique
  - Simplified hardware-assisted reference counting and block reuse
  - Reduces GC time by 29%
  - Does not significantly affect application performance
  - Reasonable cost (67KB on a 4-core chip) for an architectural accelerator of an important functionality
  - HAMM can be an enabler encouraging developers to use managed languages
Thank You!

Questions?