Flexible Reference-Counting-Based Hardware Acceleration for Garbage Collection

José A. Joao, Onur Mutlu, Yale N. Patt

ISCA 2009

presented by Theo Weidmann, 29th of October

Background & Problem

What is Garbage Collection?

- Garbage Collection: Automatic reclaiming of memory occupied by unreachable (“dead”) objects
- Objects are unreachable if there are no more pointers to it
- First described in 1960

Why use Garbage Collection?

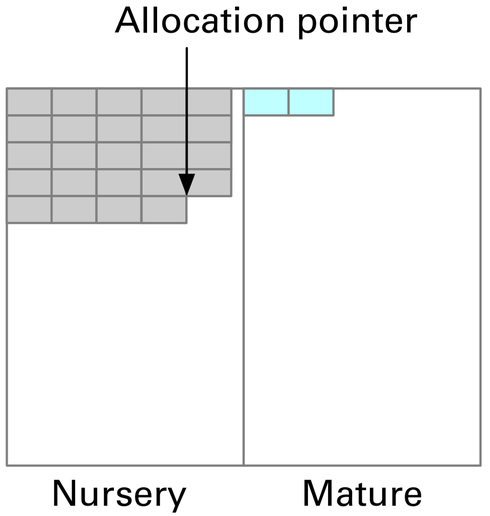
- Garbage Collection avoids memory management bugs
  - Example: Use-after-free (dangling pointer)
  - Serious security implications
  - Hard to find

- Featured in many modern managed languages
  - Java, C#, Swift

Two Main Ways to Garbage Collect
Tracing Garbage Collection

- Start from a root set (stack, global variables, registers) and recursively follow all pointers until you have reached all objects

- One option: Generational Collection
  - “Objects die young”
  - Copy all reached objects into another region called “mature”, then set allocation pointer of “nursery” to beginning



- Objects reclaimed only eventually when the garbage collector runs

Two Main Ways to Garbage Collect

Reference Counting

- Count existing pointers to every object

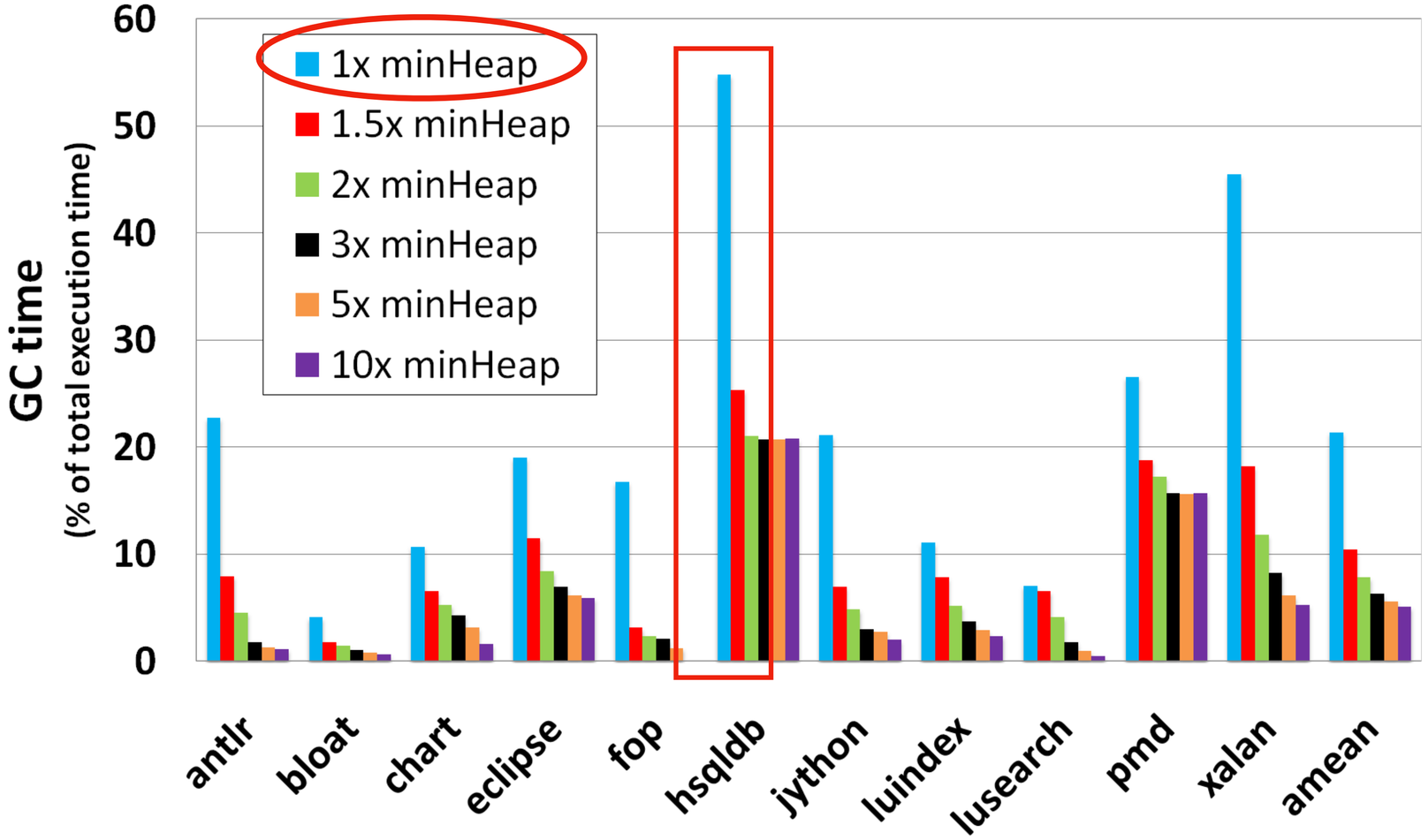
- Whenever a new pointer to an object is created → increment counter
- Whenever a pointer to an object is destroyed → decrement counter
- When count reaches 0 → object is dead and memory can be reclaimed

- Objects can be reclaimed immediately

Overhead of Software Garbage Collection

- Reference Counting
  - Counters are incremented and decremented very often
  - In parallel programming languages these counters must be synchronized
  - Often counter changes within a short interval cancel each other out
    - Example: Traversing a linked list
- Tracing Collection
  - “Stop the world” → Garbage Collector Pauses
  - Concurrent: Require extensive synchronization → Overhead

Overhead of Software Garbage Collection



Hardware Garbage Collection

NOT SUITABLE FOR GENERAL PURPOSE SYSTEMS

- Ties the ISA and microarchitecture to a specific algorithm
  - Different programs require different garbage collection algorithms to perform best
- They introduce major changes to the processor
  - Expensive to develop, test and verify
- Certain software optimizations (such as compacting) become impossible

Goal

Goal

- Provide hardware acceleration to reduce overhead
- Without limiting the flexibility of software collectors

Hardware Accelerated Memory Management (HAMM)

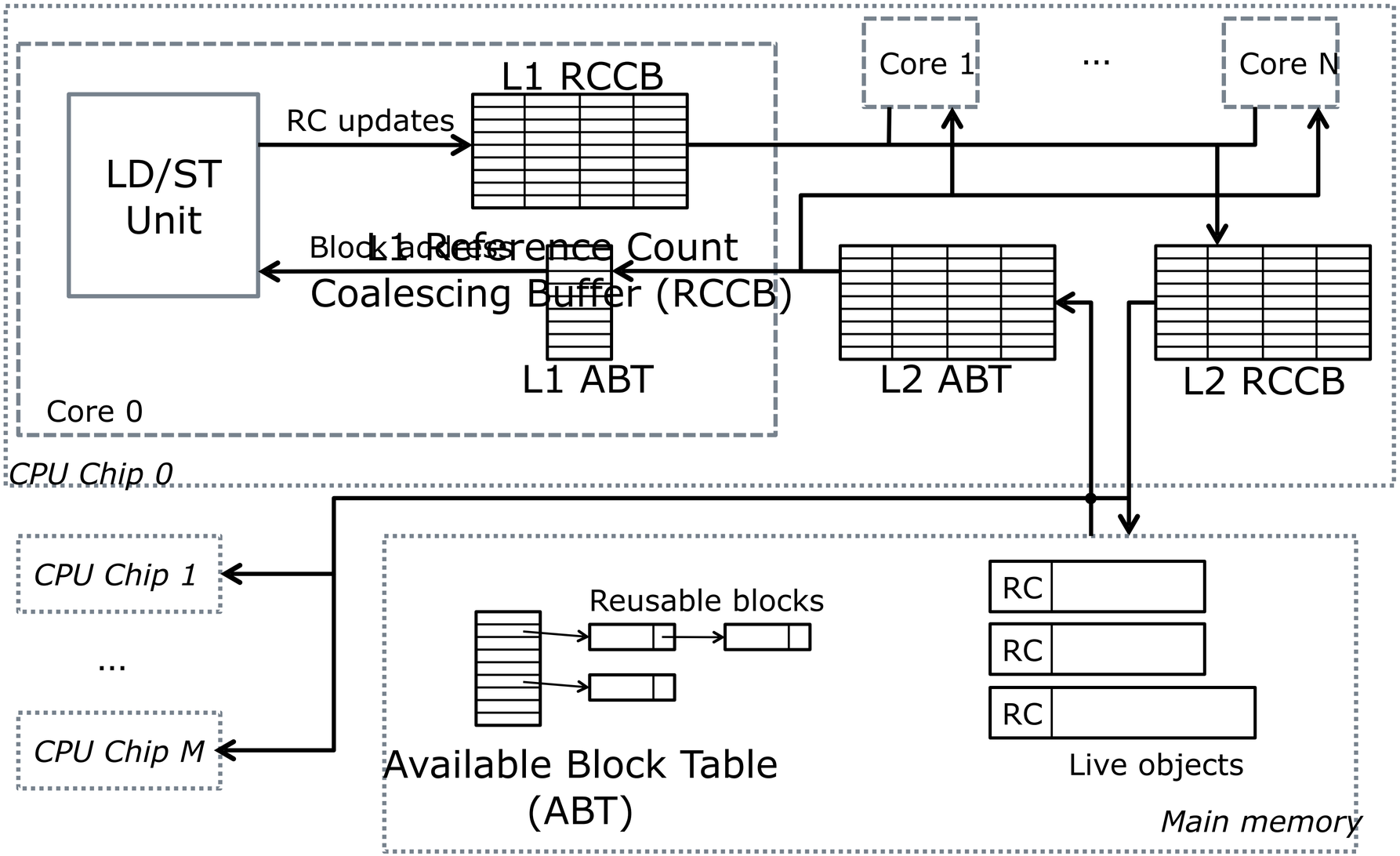
Idea

- Implement simple, flexible garbage collection in hardware
  - To keep HAMM flexible and cheap, only partially implement reference counting in hardware
  - Does not handle destructors, cyclic references, large counts and might not be able to scan all dead objects for pointers → expensive and causes rigidity
- Software garbage collectors collect remaining objects
- Overhead is reduced by less frequent software garbage collection

Mechanism Overview

- Hardware implements basic reference counting
- Hardware maintains a list of dead objects whose space can be immediately reused: Available Block Table (ABT)
- ISA is extended with
  - load/store operations specifically for pointers
  - `REALLOCMEM` to request a block for reuse from the ABT
  - `ALLOCMEM` to inform hardware about a newly allocated object

Hardware of HAMM



Software: Updated Allocation Algorithm

Software: Updated Allocation Algorithm

```
// Available Block Table
addr ⟵ REALLOCMEM size

if addr == 0 then
    // ABT does not have a free block, follow software allocation
    addr ⟵ allocate using software method
end if

// Initialize block starting at addr
ALLOCMEM addr, size
```

Software: Updated Allocation Algorithm

```
// Available Block Table
addr ⟵ REALLOCMEM size

if addr == 0 then
    // ABT does not have a free block, follow software allocation
    addr ⟵ allocate using software method
end if

// Initialize block starting at addr
ALLOCMEM addr, size
```

Software: Updated Allocation Algorithm

```
// Available Block Table
addr ⟵ REALLOCMEM size

if addr == 0 then
    // ABT does not have a free block, follow software allocation
    addr ⟵ allocate using software method
end if

// Initialize block starting at addr
ALLOCMEM addr, size
```

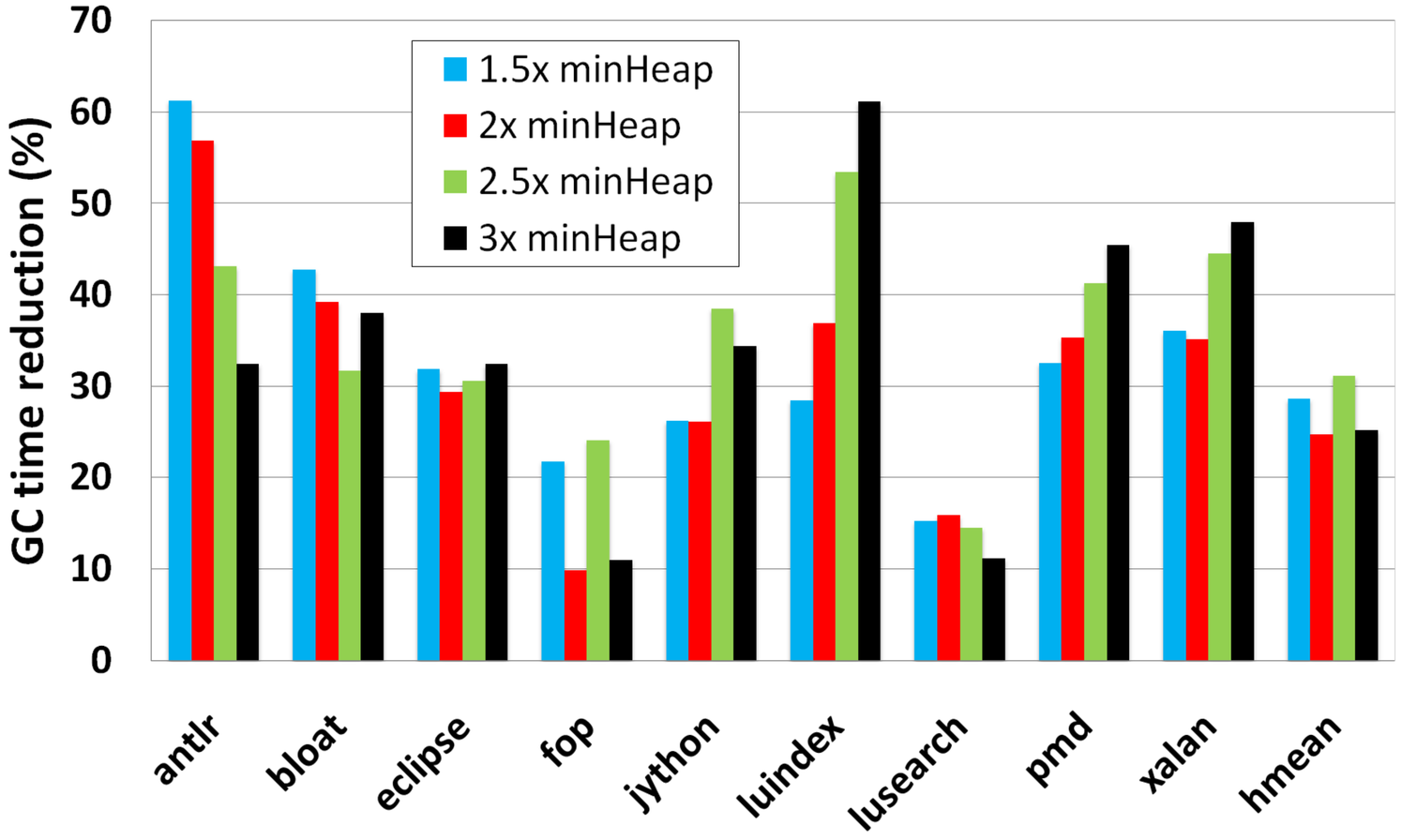
Software: Updated Allocation Algorithm

```
// Available Block Table
addr ⟵ REALLOCMEM size

if addr == 0 then
    // ABT does not have a free block, follow software allocation
    addr ⟵ allocate using software method
end if

// Initialize block starting at addr
ALLOCMEM addr, size
```

Benchmarking in Simulation

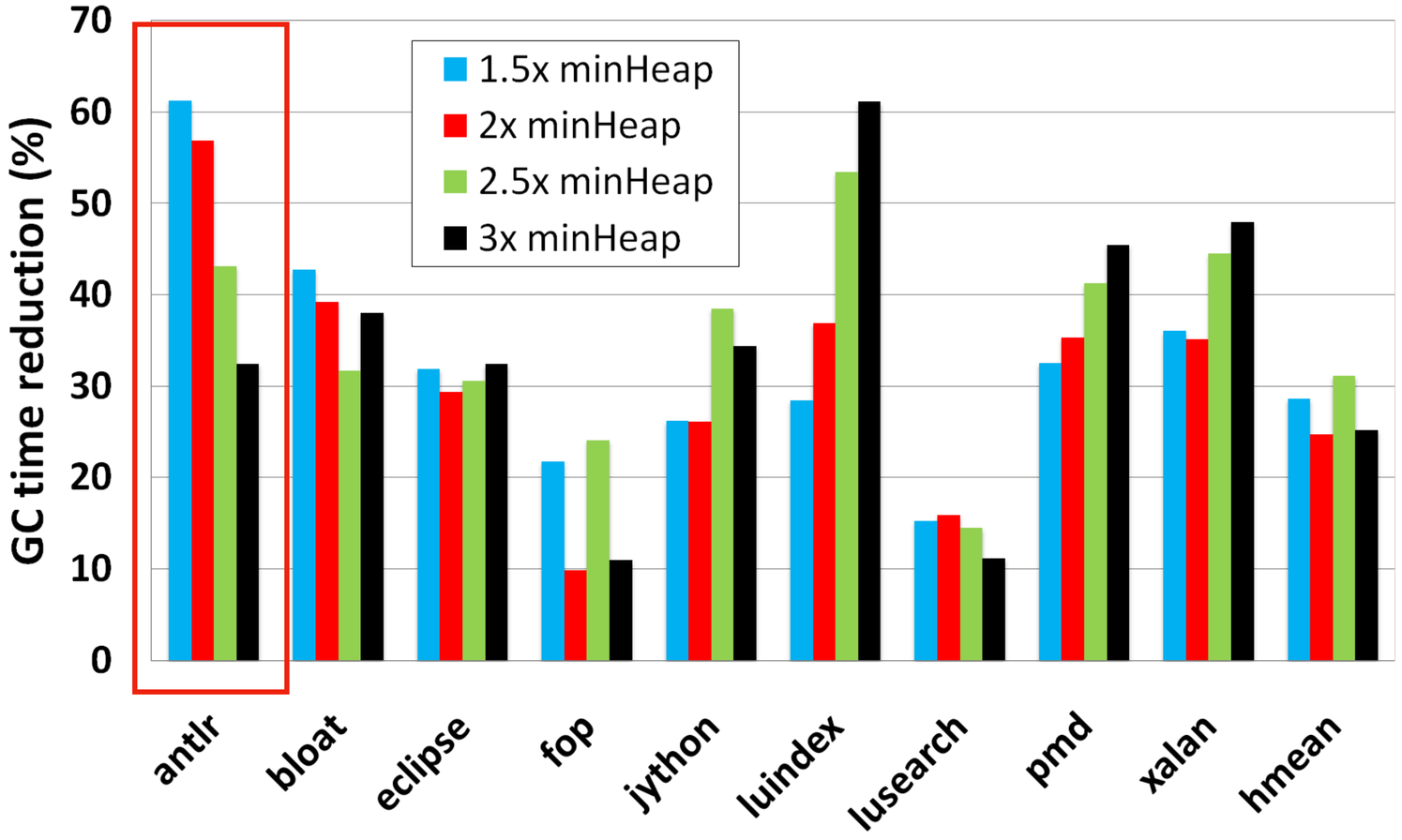
- Goal is to reduce GC time
- DaCapo benchmark executed on research JVM on a hardware simulator simulating HAMM
- Comparing GC time with and without HAMM

Benchmarking in Simulation



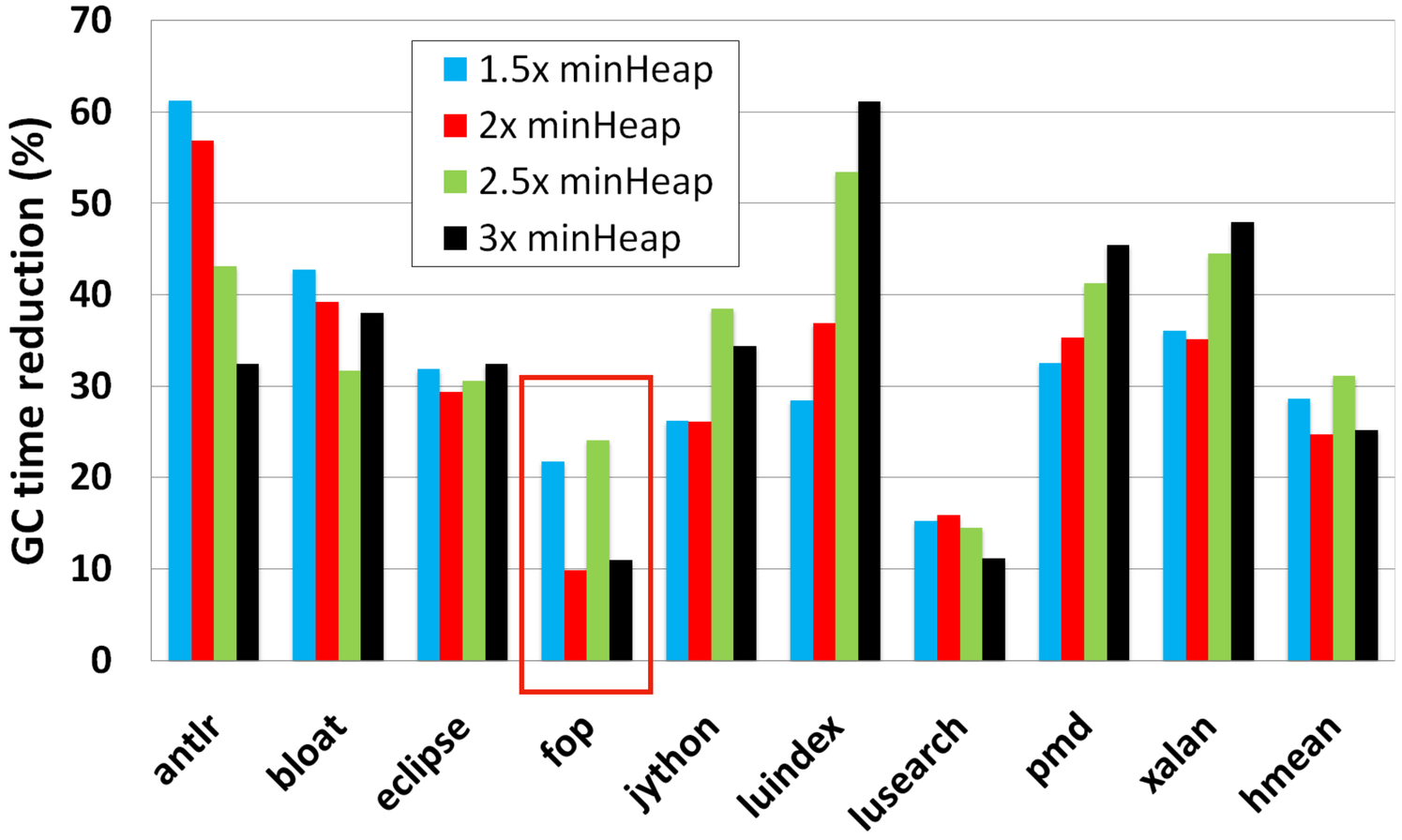
Reduction by up to 60%

Benchmarking in Simulation



Programs with lots of short-lived objects benefit greatly

Benchmarking in Simulation



HAMM benefits are smaller with mostly long-lived objects

Benchmarking in Simulation

Average Garbage Collection Time Reduction **31%**
At least 10% for heap sizes from **1.5x to 3x** minHeap

Results

- HAMM reduces GC time
  - 69% of the new objects reuse memory blocks
  - Reuse of heap memory → reduces GC cycles (50% nursery / 52% full)
  - Delaying GC gives objects more time to die → fewer objects are copied to mature region (21% less on average)
- Maximum L1 cache miss increase: 4%
  Maximum L2 cache miss increase: 3.5%
  → no significant overhead introduced

Conclusion

Executive Summary

- **Problem:** Garbage collection is useful in avoiding bugs but has significant overhead
- **Goal:** Reduce overhead while keeping hardware flexible
- **Solution:** Implement partial reference-counting in hardware so it can provide free blocks to software allocator
- **Hardware Implementation:**
  - Does not affect critical path
  - ISA extended with several instructions for pointers and allocation
- **Software Implementation:**
  - Allocation algorithm is updated to query hardware for free block
- **Effect:** Reduces GC time by 31% on average while not adversely affecting overall performance

Discussion

Strengths

- Elegant idea yet very effective
- Mechanism is optional, non-disruptive/backwards compatible
- Can be easily integrated into existing systems
- Hardware changes do not affect critical path
- General purpose system may greatly benefit
- Comprehensive evaluation
- Easy to understand paper

Weaknesses

- Only useful in environments that can use tracing collection in software
  - For example: Existing C++ applications with std::shared_ptr could hardly benefit, tracing garbage collection in C++ is problematic
  - Another example: Native iOS/macOS applications also rely on “Automatic Reference Counting”
  - Tracing collection is considered bad in real-time applications (e.g. games) due to pauses
  - Not accelerating “memory management” in general but only “tracing collection”
- Objects with finalizers/destructors cannot be managed in hardware
  - Mentioned workaround in paper: Initialize reference counter to max so that hardware won’t mark object as dead
  - Destructors: An issue with many approaches in garbage collection[1]

[1] Hans-J. Boehm. 2003. Destructors, finalizers, and synchronization. SIGPLAN Not. 38, 1 (January 2003), 262–272.

Questions

Open Discussion

- Would results be different for non-generational/non-copying collectors?
- Could this mechanism be adapted to do reference counting as would be required for C++, Swift…? What would the trade-offs be?
  - What can we learn from HAMM?
- What other run-time mechanisms could be accelerated by special hardware?
  - RAM-Swapping, Type-Checking?

Outlook
State of The Art 2020

- Focus on in-memory processing
  - GC performance is ultimately limited by memory bandwidth
    - Jaeyoung Jang, Jun Heo, Yejin Lee, Jaeyeon Won, Seonghak Kim, Sung Jun Jung, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2019. **Charon: Specialized Near-Memory Processing Architecture for Clearing Dead Objects in Memory.** In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52). Association for Computing Machinery, New York, NY, USA, 726–739.

Outlook
State of The Art 2020

- Radically rethinking memory hierarchy as we know it
  - Suited for memory-safe languages like Java, Go, Rust
  - Instead of caches, different levels of pads where objects live
  - Pointers become mere abstractions
  - Po-An Tsai, Yee Ling Gan, and Daniel Sanchez. 2018. **Rethinking the memory hierarchy for modern languages.** In Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51). IEEE Press, 203–216.

Flexible Reference-Counting-Based Hardware Acceleration for Garbage Collection

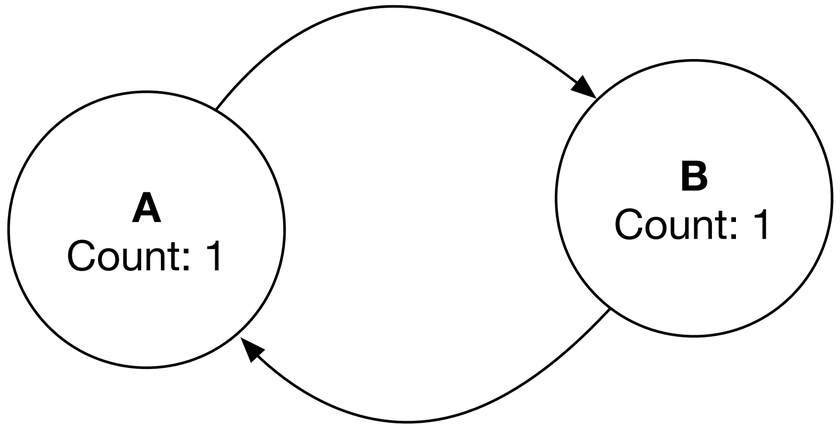
José A. Joao, Onur Mutlu, Yale N. Patt

ISCA 2009

presented by Theo Weidmann, 29th of October

Backup Slides

Issue in RC: Cycles



Can be solved by having weak references: References that do not count.