Flexible Reference-Counting-Based Hardware Acceleration for Garbage Collection

José A. Joao† Onur Mutlu§ Yale N. Patt†

†ECE Department
The University of Texas at Austin
{joao, patt}@ece.utexas.edu

§Computer Architecture Laboratory
Carnegie Mellon University
onur@cmu.edu

ABSTRACT

Languages featuring automatic memory management (garbage collection) are increasingly used to write all kinds of applications because they provide clear software engineering and security advantages. Unfortunately, garbage collection imposes a toll on performance and introduces pause times, making such languages less attractive for high-performance or real-time applications. Much progress has been made over the last five decades to reduce the overhead of garbage collection, but it remains significant.

We propose a cooperative hardware-software technique to reduce the performance overhead of garbage collection. The key idea is to reduce the frequency of garbage collection by efficiently detecting and reusing dead memory space in hardware via hardware-implemented reference counting. Thus, even though software garbage collections are still eventually needed, they become much less frequent and have less impact on overall performance. Our technique is compatible with a variety of software garbage collection algorithms, does not break compatibility with existing software, and reduces garbage collection time by 31% on average on the Java DaCapo benchmarks running on the production build of the Jikes RVM, which uses a state-of-the-art generational garbage collector.

Categories and Subject Descriptors: C.1.0 [Processor Architectures]: General; C.5.3 [Microcomputers]: Microprocessors; D.3.4 [Processors]: Memory management (garbage collection)

General Terms: Design, Performance.

Keywords: Garbage collection, reference counting.

1. INTRODUCTION

Garbage collection (GC) is a key feature of modern "managed" languages because it relieves the programmer from the error-prone task of freeing dynamically allocated memory when memory blocks are not going to be used anymore. Without this feature that manages memory allocation/deallocation automatically, large software projects are much more susceptible to memory-leak and dangling-pointer bugs [27]. These hard-to-find bugs significantly increase the cost of developing and debugging software and can damage the quality of software systems if they are overlooked by software testers and survive into the delivered code. Thus, many recent high-level programming languages include garbage collection as a feature [5, 34]. However, garbage collection comes at a cost because it trades memory space and performance for software engineering convenience. On the one hand, memory space occupied by dead objects is not reclaimed until garbage collection is executed. On the other hand, performing garbage collections too frequently degrades performance unacceptably because each garbage collection takes a significant amount of time to find dead objects. Thus, the memory space required to run a program in a managed environment with reasonable performance is usually significantly larger than the space required by an equivalent program efficiently written with manual memory management.

Garbage collection performs two distinct functions. First, it distinguishes objects reachable from a valid pointer variable ("live objects") from unreachable objects ("dead objects"). Algorithms to determine object reachability are variations of either reference counting [15] or pointer tracing [26]. Reference counting keeps track of the number of references (pointers)[1] to every object. When this number is zero, it means the object is dead. Pointer tracing recursively follows every pointer starting with global, stack and register variables, scanning every reachable object for pointers and following them. Any object not reached by this exhaustive tracing is dead. Second, after the garbage collector identifies dead objects, it makes their memory blocks available to the memory allocator. Depending on the collection and allocation algorithms, preparing memory blocks to be reused by the memory allocator may require extra memory accesses.

A garbage collector has several sources of overhead. First, it requires processor cycles to perform its operations, which are not part of the application (*mutator* in GC terminology) itself. Second, while it is accessing every live object, even the ones that are not part of the current working set, it pollutes the caches. Third, it can delay the application in different ways. Stop-the-world collectors just stop the application while they are running. Concurrent collectors impose shorter pause times on the application, but require significant synchronization overhead. When memory space is tight, garbage collection frequency may increase because every time the application is unable to allocate memory for a new object, the garbage collector has to run to free the memory left by recently dead objects. Consequently, garbage collection has a potentially significant overhead that limits the applicability of managed languages when memory is not abundant and performance/responsiveness is important [7]. Examples of memory-constrained systems are embedded and mobile systems that are usually memory-constrained for space and power reasons, and highly consolidated servers that need to efficiently use the available physical memory for consolidating multiple virtual machines.

Figure 1 shows the fraction of total execution time spent on garbage collection for the Java DaCapo benchmarks with a production-quality generational garbage collector for different heap sizes, relative to the minimum heap size that can run the benchmark.[2] The overheads of garbage collection with tight heap sizes are very significant for several benchmarks, in the 15-55% range. For this reason, running a program with tight heaps is usually avoided, which results in over provisioning of physical memory, thereby increasing cost and power consumption. Even with large heap sizes, the overhead of garbage collection is still significant (6.3% on average for 3x minHeap).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA'09, June 20–24, 2009, Austin, Texas, USA.
Copyright 2009 ACM 978-1-60558-526-0/09/06 ...$5.00.

[1]A *reference* is a link from one object to another. In this paper we use the terms *reference* and *pointer* interchangeably.

[2]We run the benchmarks on Jikes RVM with the large input set on an Intel Core2 Quad

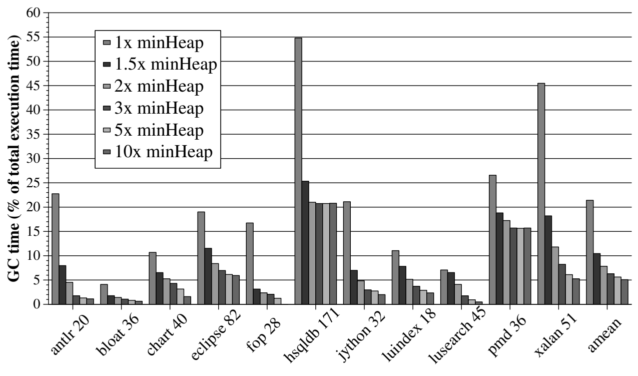


Figure 1: Garbage collection overheads for DaCapo benchmarks with different heap sizes (minHeap values in MB shown in labels)

Software garbage collectors provide different trade-offs in terms of memory space, performance penalty and pause times. Most of the currently used and best-performing collectors are "generational collectors" [35]. Generational collectors exploit the generational hypothesis, which states that in many cases young objects are much more likely to die than old objects. New objects are allocated into a memory region called "younger generation" or "nursery." When the nursery fills up, surviving objects are moved to the "older generation" or "mature" region.[3] Generational collectors usually combine a bump-pointer allocator [4] that provides fast contiguous allocation and good locality for new objects, an efficient copying collector for the nursery [35], and a space-efficient collector for the mature region, for example mark-sweep [26]. Nursery collections can be frequent but are usually fast because young objects that die quickly are neither traced nor copied to the older generation. Full-heap collections are expensive but infrequent when the heap is reasonably large compared to the working set of the application and the virtual machine. Even with very efficient multi-generation collectors like the one used in the .NET framework, the total overhead of garbage collection can be significant (12% on average for the applications shown in [12]).

Reference counting is an alternative to tracing garbage collection. Instead of determining reachability by periodically tracing all pointers, reference counting can find dead objects as soon as they become unreachable and their memory blocks can be immediately reused to allocate other objects, eliminating the need for explicit garbage collections. However, there are two fundamental limitations of reference counting. First, the overhead of updating the reference counts on every pointer creation and destruction is very high and makes immediate reference counting unattractive in software. The overhead is even higher in multithreaded systems, which require synchronization for all reference count updates. Some research proposals in reference counting ([16, 24, 9]) skip reference count updates for pointers in registers and on the stack and offer more competitive overheads, but require processing the pointers on the stack and registers before determining that any object is effectively dead. A second basic limitation of reference counting is that it is not complete, i.e. it cannot easily detect the death of objects that are part of cyclic data structures. Therefore, running either a cycle detection algorithm or a complete garbage collection algorithm is eventually required.

Hardware Garbage Collectors. Proposals have been made to implement garbage collection fully in hardware [32, 36, 29, 30, 28, 33]. Unfortunately, fully implementing garbage collection in hardware is undesirable for general purpose processors due to at least three major reasons. First, a hardware implementation ties a particular garbage collection algorithm into the ISA and the microarchitecture. There is no single memory allocation and collection policy that performs the best for every application and memory configuration [23] and the hardware implementation has to pick one particular mechanism. A hardware implementation of a scheme that works well on average may not be the best memory management platform for many applications, and the rigid hardware implementation cannot adapt beyond simple tuning. Second, the cost of developing, verifying and scaling previously proposed full-hardware solutions is very high, because they introduce major changes to the processor [29, 30, 28, 33] or the memory [32, 36] architecture. Third, full-hardware collectors miss opportunities available at the software level, such as taking advantage of garbage collections to change the memory layout to improve locality [19, 17, 12]. In summary, hardware GC makes a rigid trade-off of reduced flexibility for higher performance on specific applications, and this trade-off is not suitable for general purpose systems.

Given the importance of garbage collection in modern languages and its performance overhead, we hypothesize some form of hardware acceleration for GC that improves performance without eliminating the flexibility of software collection is highly desirable, especially in future processors with higher levels of resource integration.

Hardware trends. The number of transistors that can be integrated on a chip continues to increase and the current industry trend is to allocate them to extra cores on a CMP. However, Amdahl's law [3] and the difficulty of parallelizing irregular applications to a high degree are turning the attention of computer architects towards asymmetric solutions with multiple core configurations and special-purpose hardware blocks that work as accelerators for commonly used functionality. Given that garbage collection is an increasingly common and significant source of overhead in modern languages, it is important to accelerate it via dedicated hardware support.

Our goal is to provide architectural and hardware acceleration support that can reduce the overhead of garbage collection without limiting the flexibility that a software collector can offer. We propose a hardware-assisted form of reference counting that works as a complement to any existing garbage collection algorithm to reduce its overhead. When our low-cost reference counting mechanism can safely indicate that an object is dead, the object's memory block can be reused to allocate another object, delaying the need to garbage collect. However, since reference counting is not complete, software GC is still eventually required, but much less frequently. Our proposal gives the memory allocator the choice of either reusing a memory block from the hardware-managed pool of newly-found dead objects, or allocating a new block with the default software allocation algorithm. This approach combines the reduced GC overhead given by the hardware support with the flexibility of software allocation to improve locality or adapt to specific characteristics of applications. Thus, the hardware-software approach can potentially achieve better performance than a pure-software or pure-hardware approach.

Basic Idea. Our proposal, Hardware-Assisted Automatic Memory Management (HAMM), has two key components. The first maintains a reference count field in each object's header in main memory. The compiler modifies pointers using new ISA instructions and the hardware processes the resulting reference count updates. These updates are consolidated in *Reference Count Coalescing Buffers (RCCB)* both at the core (L1 RCCB) and at the chip (L2 RCCB) level to reduce the frequency of reference count updates in memory. Entries evicted from the L2 RCCB update the reference count in the object header. The RCCBs are able to filter 96.3% of the reference count updates on average for our benchmarks, effectively neutralizing the main disadvantages of reference counting, i.e. the significant increase in memory traffic and the need to synchronize on every reference count update. The second component makes memory blocks used by dead objects promptly available to the software allocator for reuse. It does so through simple hardware-maintained free-block lists called *Available Block Tables (ABT)* in main memory and both at the chip level and at the core level, which are refilled as memory blocks are

Q6600 processor.

[3]More generally, subject to a *tenuring threshold* (i.e. objects that have survived some number of nursery collections are promoted to the mature region)

reused and objects are found dead. The ABTs allow memory block reuse without delaying or introducing long-latency cache misses into the timing-critical memory allocators. We provide new ISA instructions for the allocator and garbage collector to interact with the proposed hardware support. Our evaluation shows that the combination of hardware-accelerated reference counting and support for memory block reuse allows significant reduction in the overhead of GC without significantly affecting the performance of the application itself.

Contributions. This paper makes the following contributions:

1. We propose a cooperative hardware-software mechanism to allow reuse of memory blocks without garbage collection, thereby reducing the frequency and overhead of software GC.
2. Unlike previously proposed pure-hardware or pure-software garbage collection techniques, our proposal accelerates GC in hardware while letting the software memory allocator and garbage collector make higher-level decisions, for example about where to locate objects to improve locality. Our comprehensive proposal provides primitives for faster GC in hardware without limiting the flexibility of software collectors that can be built on top of the primitives. Our proposal is compatible with many existing software GC algorithms.
3. Our evaluation shows that our proposal reduces the execution time spent on garbage collection by 31% on average for a set of the Java DaCapo benchmarks running on Jikes RVM, a research virtual machine with a state-of-the-art generational garbage collector. We show that our proposal saves garbage collection time in both generations without significantly affecting the execution time spent in the application (*mutator*).

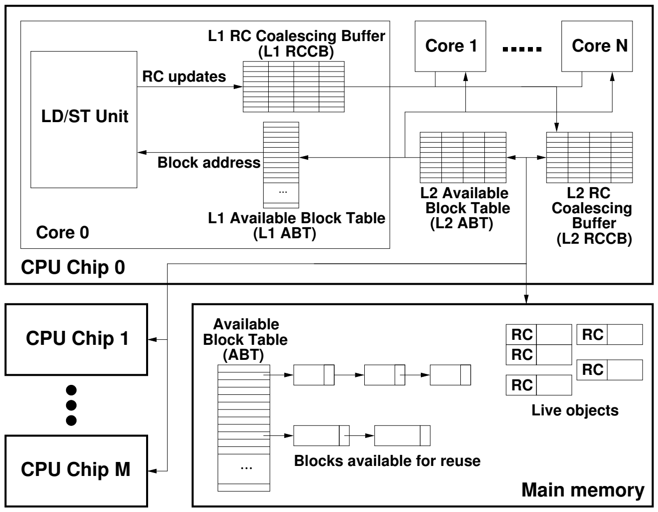
2. HARDWARE-ASSISTED AUTOMATIC MEMORY MANAGEMENT (HAMM)

2.1 Basic Concept

We propose HAMM, a hardware-software cooperative reference counting (RC) mechanism that detects when objects die and enables reusing their memory blocks to allocate objects, delaying the need for garbage collection and therefore reducing the overhead of garbage collection using acceleration support in hardware. The strategic decision about whether to reuse a memory block or to allocate a new memory block is left to the software memory allocator. Our objective is not to replace the software garbage collector in hardware, but to reduce its overhead with simple hardware primitives. Thus, our reference counting mechanism does not need to accurately track all objects as long as it accurately indicates that an object is dead, so it can be designed to keep the required software and hardware support at reasonable cost. The baseline garbage collection algorithm is eventually executed to collect every dead object, but these collections are needed much less frequently and with less overhead with HAMM.

The hardware support provided by HAMM consists of two components: 1) **reference count tracking:** structures that keep track of reference counts for objects, 2) **memory block reuse handling:** structures that determine and quickly supply available memory blocks to the software allocator.

Figure 2 shows an overview of the additional hardware required for HAMM. Each object has a reference count field (RC) in its header. When instructions that create or destroy pointers commit, they issue reference count update operations (increment or decrement, respectively) that are cached in an *L1 Reference Count Coalescing Buffer (L1 RCCB)* in each processor core. The purpose of the RCCB is to merge multiple reference count updates for the same object to reduce the effective number of updates processed beyond each core. A large fraction (90.6% on average) of all reference count updates are filtered by the L1 RCCB. A second level *L2 Reference Count Coalescing Buffer (L2 RCCB)* merges RC updates (e.g., for shared objects) from all cores on the chip before effectively applying the RC updates to



L1 RC Coalescing Buffer (L1 RCCB)
RC updates
LD/ST Unit
Block address
L1 Available Block Table (L1 ABT)
Core 0
Core 1
Core N
L2 Available Block Table (L2 ABT)
L2 RC Coalescing Buffer (L2 RCCB)
CPU Chip 0
CPU Chip 1
CPU Chip M
Available Block Table (ABT)
RC
Live objects
Blocks available for reuse
Main memory

Reference count tracking (L1 and L2 RCCBs, RC field in each object's header): determine dead objects without explicit GC and with low overhead.

Memory block reuse handling (L1, L2, and memory ABTs): enable fast memory reallocation into available memory blocks found by RC.

Figure 2: Overview of the major hardware structures provided by HAMM (components are not to scale)

the reference count field in each object's header. The purpose of the RCCB hierarchy is twofold: 1) to mitigate the performance impact of RC updates on the interconnect and memory traffic and 2) to efficiently handle the synchronization of RC updates across cores with minimum interference on the application.

When an object's reference count becomes zero, i.e. it does not have any references to it and therefore has become garbage, its memory block is added to the *Available Block Table (ABT)*, which is a data structure stored in main memory. We classify memory blocks according to their reusable size into one of 64 *size classes*. Conceptually, ABTs are a set of free-block lists, one for each size class. To avoid introducing unnecessary delays on the allocator, our design includes a single-entry L1 ABT at the core level and an N-entry L2 ABT at the chip level (assuming there are N cores per chip), which are refilled as memory blocks are reused.

When the memory allocator requests a block, the L1 ABT is accessed with the corresponding block size class. If a block large enough to satisfy the request is present in the L1 ABT, the hardware supplies its address to the software allocator, and requests another block from the L2 ABT to refill the corresponding L1 ABT entry. As with most of the communication traffic and extra processing introduced by our proposal, this request is not time-critical and can be satisfied with lower priority with respect to other requests. Eventually, the L2 ABT will also be refilled with either a newly found dead object or from the lists of available blocks kept in the memory ABT.

Our proposal requires support from the compiler, the ISA, and the hardware for 1) object allocation, 2) tracking references, 3) determining object death, 4) reallocation of memory blocks, and 5) interacting with the software garbage collector. We will next explain in more detail how our proposal works in each of these aspects.

2.2 Object Allocation

We introduce two new instructions in the ISA to let the software memory allocator interact with the supporting hardware. First, the allocator must be able to query the hardware for the availability of a reusable memory block of the required size. We provide the instruction REALLOCMEM for this purpose. If this instruction returns a non-zero address, which points to an available memory block, allocation can proceed on that block. Otherwise, the allocator proceeds with its default allocation policy, for example bump-pointer allocation [4] or free-list allocation [26], commonly used in software memory allocators. Next, the allocator must inform the hardware of the existence of a newly allocated memory block using the new ALLOCMEM instruction. Table 1 describes these new instructions.

Table 1: HAMM ISA instructions for object allocation

Format	Description
REALLOCMEM reg1, reg2	Requests a memory block of size reg2 from the hardware-maintained local Available Block Table (L1 ABT). It returns in reg1 the starting address of an available block or zero if no block could be found, and in reg2 the actual size of the block. Condition codes are set based on reg1.
ALLOCMEM reg1, reg2	Informs the hardware of a newly allocated block for an object of size reg2 referenced with address reg1, so that the first reference to the object is created in the hardware reference counting mechanism. ALLOCMEM creates an entry in the local L1 RCCB with a reference count of one (i.e., reg1), and writes the 6-bit size class to the object header.

2.3 Reference Counting

Object reuse is provided by hardware-assisted reference counting. Each object has a reference count field in its object header. Reference count updates are generated as a side effect of the instructions that modify pointers and are then consolidated in a hierarchy of Reference Count Coalescing Buffers, which eventually updates the reference count field in each object's header.

Compiler and ISA support are required to properly update the reference counts. Anytime a pointer *P* is stored, the program is creating a new reference to the object at address *P*. If the instruction storing the pointer overwrites a previous value *Q* of the pointer, the program is also effectively destroying a reference to the object at address *Q*. To keep track of creation and destruction of references during program execution, a special "store pointer" instruction is used every time a pointer is written to a register or memory location. Store pointer instructions update the reference counts for the pointer that is created or destroyed, in addition to actually storing the pointer. There are two variants of the store pointer instruction: one that writes a new pointer and one that overwrites an existing pointer. When both types of these instructions commit, they generate an *increment reference count* or *incRC* operation for the stored reference. Additionally, instructions that overwrite an existing reference generate a *decrement reference count* or *decRC* operation for the overwritten reference. Additionally, these instructions can be simple STORE (or MOV to register/memory), PUSH or POP (similar to MOV, but copying to or from the top of the stack). Table 2 describes the format and semantics of the store pointer instructions.

Table 2: HAMM ISA *store pointer* instructions

Format	Description
MOVPTR reg/mem, src_reg	Stores a new reference in register src_reg to a reg or mem and increments the reference count for src_reg.
MOVPTROVR reg/mem, src_reg	Overwrites an existing reference in reg or mem with a reference in src_reg, increments the reference count for src_reg and decrements the reference count for the overwritten reference originally in reg/mem.
PUSHPTR reg/mem	Pushes a reference in reg/mem onto the stack and increments the reference count.
POPPTR reg/mem	Pops a reference from the stack into reg/mem.
POPPTROVR reg/mem	Pops a reference from the stack, overwriting an existing reference in reg/mem, whose reference count is decremented.

Software reference counting performs basically the same operations as the store pointer instructions, but it requires synchronization to modify a pointer value while reading the previous value of the pointer, e.g. using a compare-and-swap instruction and the corresponding check for success and eventual retry loop. By adding the RC updates to the semantics of the ISA instruction that actually stores the pointer, we let the hardware cache coherence mechanism handle the synchronization without unnecessary overhead.[4] When the cache line holding the pointer is modified, the overwritten value of the pointer is read[5] and propagated down the pipeline until the instruction commits and performs the RC update operations on the L1 RCCB: a *decRC* operation for the old value of the pointer and an *incRC* operation for the new value of the pointer.

2.3.1 Handling References in Registers

References in registers can also be destroyed by overwriting the register with a non-reference value. We extend the register file with an extra Reference bit. This bit is set when the register is written to by any of the store pointer instructions. It is cleared when the register is overwritten by any other instruction. When the bit is cleared, the original value marked as a reference is carried down the pipeline and when the instruction commits, a *decRC* operation is performed on the overwritten reference.

2.3.2 Handling References on the Stack

References on the stack can be implicitly destroyed by simply going out of scope, for example by updating the stack pointer on returning from a subroutine. We extend the L1 cache with one extra Reference bit per word, which is set when a store pointer instruction writes to the word, and cleared when 1) any other instruction writes to the word or 2) a POPPTR or POPPTROVR instruction pops the word from the stack. The processor lazily performs *decRC* operations on references left in the invalid region of the stack frame, either when the word marked as a reference is overwritten because the stack regrew or when the cache line is evicted from L1.

2.3.3 Reference Count Coalescing Buffers (RCCBs)

The *incRC* and *decRC* operations generated by any of the mechanisms previously explained should not directly update the reference count field in the object header due to two reasons. First, the resulting increase in memory traffic would be very significant because references are frequently created and destroyed, especially in register and stack variables. Second, many of these frequent RC updates affect the same set of objects in the current working set of the program and many cancel each other in a short amount of time. Thus, only the net value of all RC updates for a certain object matters, not the individual updates. Consider for example, that while a linked list is being traversed the current pointer changes from one node to the next on each iteration, incrementing and decrementing the RC for every node in the list. However, after the traversal finishes without changing the list, the net effect of all those RC updates is zero.

For these reasons, we consolidate RC update operations in the Reference Count Coalescing Buffers (RCCBs) before applying them to the reference count in the object through the memory system. RCCBs are set-associative hardware tables. They are indexed and fully-tagged based on the reference value (virtual address), and each entry has a signed integer reference count delta field (RCD). The RCD field in the L1 RCCB on each core is incremented by each *incRC* operation and decremented by each *decRC* operation. When an L1 RCCB entry is evicted, it updates the corresponding on-chip L2 RCCB. If there is already an entry in the L2 RCCB for the same reference, the RCD value coming from the L1 RCCB is simply added up to the corresponding RCD field in the L2 RCCB entry.

2.4 Object Death

An object can be declared dead anytime after its reference count becomes zero. We process all RC increments and decrements non-speculatively and off the critical path, i.e. when instructions commit. RC updates on each core are first consolidated locally on the

[4]The overhead of writing to the header of heavily shared objects could still be significant, but is already mitigated by the coalescing of multiple updates in the RCCB hierarchy and it could be further mitigated —if needed— by disabling HAMM for heavily-shared objects on a per-page basis, because our reference counting is not required for correctness.

[5]We assume an allocate-on-write-miss policy on the L1 cache.

L1 RCCB, independently of other cores' L1 RCCBs. In general, entries evicted from an L1 RCCB update the on-chip L2 RCCB, and entries evicted from an L2 RCCB update the reference count in the corresponding object header in main memory. When the updated reference count in the object header is zero, the object *may be* dead. RC updates make their way to the object header as they are evicted from L2 RCCBs. If references to a shared object are created/destroyed by threads running on different processors, it is possible for a decrement to reach the object's reference count field while an increment is still stored in some other RCCB. To confirm that the object is effectively dead, all RCCBs in the system are checked for pending RC updates to that object, and if there is none, the object is declared dead.[6]

2.5 Memory Block Reuse

When a reference count is confirmed to be zero, the corresponding memory block can be reused. To allow reuse, the block address has to be made readily available to a processor executing a REALLOCMEM instruction. This is the function of the Available Block Table (ABT) hierarchy. The ABT, which is stored in main memory, is conceptually a table of free-block lists, one per size class, implemented as linked lists of available memory blocks. The ABT itself contains the head pointers of the free lists and is indexed based on size class. When an object is found dead, the starting address of its memory block is inserted at the head of the free-list corresponding to the size class field in the object header. The linkage pointer is written into the now-useless header of the memory block.

Efficient memory allocators are designed to avoid delays produced by memory accesses that miss in the cache or synchronization with other threads. To keep this efficiency, the execution of the REALLOCMEM instruction by the memory allocator should not introduce cache misses or synchronization overheads. Thus, we use the L1 and L2 ABTs to prefetch the addresses of available memory blocks from the memory ABT, so that the processor executing REALLOCMEM only accesses its local L1 ABT, and if no block is found there, it executes the default allocator code without further delay.

Each core has an L1 ABT table with one entry per size class, and each N-core chip has an L2 ABT with N entries per size class. These ABTs contain addresses of available blocks that were found dead at the local L2 RCCB or were previously retrieved from the memory ABT. After a REALLOCMEM instruction reuses the block in the L1 ABT, a refill request is sent to the L2 ABT, which in turn gets a refill from the head of the corresponding memory ABT list.

2.6 Garbage Collection

Our proposal does not eliminate the need for garbage collection but makes garbage collection less frequent. The software garbage collector makes the software aware of all available blocks, rendering the information in the ABTs obsolete. Thus, ABTs must be reset when a software collection starts.

Pending RC updates in the RCCBs have to be applied to the RC field in the object header before or during software GC for two reasons. First, copying or compacting collectors move objects to another memory region to free memory space for new objects and adjust all pointers in the system to point to the new address of the relocated object. The reference count in the object header is moved with the rest of the object. So, any RC update left in an RCCB after moving the object would be lost because the RCCB holds the old address of the object. Second, after the collection ends, memory blocks used by dead objects become free space and can be reused in different ways. If any pending update to one of these blocks is left in an RCCB, it can eventually update a memory location in the freed memory area that is not a valid RC field anymore, which would be a serious error.

For these two reasons, when a software collection starts, it triggers a lazy flush of all RCCB entries, which is done in hardware with low priority as the collection progresses. Meanwhile, a copying/compacting software GC can concurrently make progress, but it cannot move an object without confirming that the object does not have pending RC updates. We add the instruction FLUSHRC for this purpose. If the lazy flush of all RCCBs is still in progress, the FLUSHRC instruction proactively flushes all RCCB entries for a specific object and waits until the object header is effectively updated. After the lazy RCCB flush is complete, further FLUSHRC instructions encountered can simply be ignored without even querying the RCCBs. Table 3 shows the format of the FLUSHRC instruction.

[6]All RCCBs are also queried for pending updates if the updated RC in the object header is negative, meaning that there must be a pending increment in some RCCB.

Table 3: HAMM ISA instruction for garbage collection

Format	Description
FLUSHRC reg	Flush all pending RC updates in RCCBs for the object at address reg to its reference count in the object header.

3. IMPLEMENTATION

Our proposal requires the extensions to the ISA discussed in the previous section, which are new instructions that do not break compatibility with existing software that does not use our extensions. The compiler or the interpreter has to be modified to make use of the new instructions. Similarly, the allocator and the garbage collector (Section 2.6) have to be changed to interact with the architectural changes. In this section we explain in more detail these modifications to the software stack and the implementation of our hardware mechanisms. More detailed explanations are provided in a technical report [20].

3.1 Compiler Support

Since the compiler or the interpreter for a managed language like Java or C# knows exactly when it is writing or overwriting a pointer, it conveys this information to the hardware using the new store pointer instructions in the ISA described in Section 2.3. This allows the hardware to efficiently track the creation and destruction of references, and to keep accurate reference counts for each object of interest.

The memory allocator has to be slightly modified as we show in Algorithm 1 for a simple bump pointer allocator. First, the allocator checks if there is a block available for reuse, using the REALLOCMEM instruction. Since REALLOCMEM only checks the local L1 ABT, it does not introduce any significant overhead that can slow down the fast path of the allocator. Once the allocation is complete, either as a reallocation or as a new allocation, the ALLOCMEM instruction informs the hardware of the existence of the new block.

Algorithm 1 Modified bump pointer allocator

```
$addr \leftarrow$ REALLOCMEM $size$
if ($addr == 0$) then
    // ABT does not have a free block, follow software allocation
    $addr \leftarrow bump\_pointer$
    $bump\_pointer \leftarrow bump\_pointer + size$
    if ($bump\_pointer\ overflows\ local\ heap\ chunk$) then
        Use Slow-path Allocator (synchronized allocation of a new chunk for
        local consumption, trigger GC if necessary, etc.)
    end if
else
    // ABT returns a free block, use address provided by hardware
end if
Initialize block starting at $addr$
ALLOCMEM $object\_addr, size$
```

3.2 Reference Fields in Dead Objects

The hardware efficiently keeps track of reference counts for all objects of interest. As we explained in Section 2.3, *incRC* and *decRC* operations are non-speculatively generated by store pointer instructions, and additional *decRC* operations are generated when references are overwritten in registers or on the stack. These operations eventually update the object's reference count and are not on the critical path of the store instruction.

Reference counts for pointer fields found in dead objects are also decremented to increase the coverage of HAMM. The processor attempts to scan an object for pointers as soon as it is found dead. Object scanning requires the knowledge of which words in the object layout contain references. Our mechanism uses a 1K-entry Object Reference Map (ORM) table in reserved main memory for each application. This table is indexed and fully tagged with the unique type identifier used in each object header to determine the class of each object, which is usually a pointer to the type information in the runtime system. Each ORM entry has a 1-word bitmap (32 or 64 bits depending on the architecture), where each bit indicates that the corresponding word in the object layout is a pointer. This table is updated by the software runtime system for the most commonly used types. When an object has to be scanned for references, the processor reads the type identifier from the object header, accesses the ORM and gets the reference bitmap,[7] which is aligned and applied to the Reference bits in the L1 cache line containing the object. After that, the same hardware that issues *decRC* operations for discarded stack frames (Section 2.3.2) is used to issue *decRC* operations for the references found by object scanning.

Note that correctness does not require reference counts to be timely decremented as long as no increment is left unprocessed before deciding an object is dead. If some reference counts are not decremented when they should be or are not decremented at all, only the opportunity to reuse the memory for an object that became garbage is lost. Dead objects that are missed by our proposal will eventually be collected by the default software GC.

3.3 Multiple Managed Memory Regions

Up to this point in the paper, we have assumed our scheme is designed to handle memory reuse for a single memory region in a single application. We now extend our mechanism to handle multiple concurrent applications. Also, each managed application can allocate objects into different memory regions with different allocation policies, and memory blocks in each of these regions have to be reused and collected independently. For example, a generational collector uses two separate memory regions —the nursery and the older generation— and each of them is assigned to specific ranges of the virtual memory space, contiguous or not. Therefore, HAMM has to differentiate memory blocks belonging to different memory regions. We extend the page table entries and the TLBs to include a 2-bit region ID within the current application (process). ID 0 prevents our mechanism from tracking references to objects in that page and IDs 1 through 3 can be used to identify valid memory regions. Additionally, each application that wants to use our mechanism obtains a 6-bit application ID. Together, the application ID and the region ID define an 8-bit *allocation pool ID*, which is made part of the tags in RCCBs and L2 ABTs to avoid aliasing of addresses across different applications. Table 4 summarizes the extensions to HAMM to handle multiple allocation pools. On process context switches, RCCBs are flushed to memory for the evicted process and unused L1 and L2 ABT entries are returned to the corresponding memory ABT.

3.4 Hardware Cost

Table 5 shows the storage cost of HAMM assuming a 64-bit baseline architecture and a 4-core symmetric CMP. Since we are assuming all addresses require the full 64 bits, this is a pessimistic estimate of the amount of required storage. Additional combinational and sequential logic is also required to implement HAMM. However, none of the structures of our mechanism is on the critical path. HAMM also allocates data structures in main memory, as detailed in Table 6.

HAMM uses one byte in the object header for the reference count, which works as a special kind of unsigned saturated counter, i.e. once it reaches its maximum value it cannot be further incremented or decremented. We also use an extra 6-bit *size class* field in the header, written by the ALLOCMEM instruction. It is not difficult to find spare bits in the object layouts of most managed runtime systems, especially in 64-bit architectures, because most header fields can be significantly compressed [1]. Many runtime systems also reserve an extra word in each object header for garbage collection information (e.g. mark bits) or dynamic profiling, where our extra fields could also fit. Our evaluation makes this assumption, but if the object header cannot accommodate the extra fields, we consider adding 2 bytes per object. If we assume the average object size is 48 bytes [6], the overhead would be 4.1% of the allocated memory.

3.5 Limitations

The effectiveness of our proposal is limited by its ability to discover dead objects and reuse their space. Any cyclic data structure cannot be reference-counted correctly without an additional algorithm to handle circular references. Thus, any object that is part of a cyclic data structure or that is referenced from an object in a cyclic data structure will not be found dead by HAMM.

The limitation on the size of the reference count field to 8 bits excludes the most heavily connected objects from benefiting from HAMM, after the reference count reaches saturation.

Our ABT hierarchy is basically a set of segregated free-block lists, which is well suited for free-list allocators. However, bump-pointer allocators are not limited to predetermined block sizes and they avoid internal fragmentation. If a block of arbitrary size is allocated, our mechanism assigns it to the next smaller size class, which produces some internal fragmentation on block reuse. If REALLOCMEM is called with an arbitrary size, the L1 ABT entry for the immediately larger size is used, also introducing internal fragmentation.

Weak references are used in many managed languages as references that do not make an object reachable, i.e. they do not prevent an object that is not referenced otherwise from being collected. Our proposal does not currently differentiate between weak references and regular references, which can limit the ability to find some dead objects that would be considered dead by the software GC.

Objects of classes with a *finalizer* method cannot be reused immediately after being found dead because the finalizer has to be run on them before their memory space is reused or overwritten.[8]

4. EXPERIMENTAL METHODOLOGY

We evaluate our proposal on the DaCapo [6] benchmark suite version 2006-10-MR2 with the large input set. We run the benchmarks twice and report results for the second run because dynamic compilation significantly impacts the results of the first run [8]. We report results for all benchmarks in the suite except for `chart` and `hsqldb`, which could not run on our simulation infrastructure.

We use the production configuration of Jikes RVM [2] 2.9.3 running on Simics [25] 3.0.29. The production configuration of Jikes RVM uses its best performing [7] generational collector with a copying collector for the nursery and a mark-sweep collector for the older generation, which we call GenMS. We implement the functionality of our proposed hardware structures, as well as the new ISA instructions described in Section 2, in a Simics module. Simics is a functional simulator and does not provide accurate execution times. Evaluating hardware support for garbage collection is very challenging. To accurately determine the reduction in GC time with HAMM, we should run each benchmark to completion on a cycle accurate simulator. However, it is infeasible to perform these detailed microarchitectural simulations because they would run for a very large number of instructions (several hundred billions for some benchmarks). We divide the execution time of the program into two components: (1) the garbage collector, and (2) the application itself, or mutator. We then apply different simulation approaches for each component.

[7]Most objects are smaller than the 128 or 512 bytes we can map with this bitmap, and we only do partial scanning on larger objects.

[8]We can easily prevent specific objects from being found dead with our mechanism by initializing the reference count field in the object header to its maximum saturating value.

Table 4: Extensions for Multiple Managed Memory Regions

Structure	Description of changes
L1 and L2 RCCB	Extended with the allocation pool ID, which becomes part of the tag field along with the reference virtual address.
Page table and TLBs	Each entry extended with the 2-bit region ID, which allows every *incRC* or *decRC* operation to determine the region (and allocation pool) ID before accessing the L1 RCCB.
Memory ABT	Each allocation pool has its own memory ABT because memory blocks belonging to different applications and memory regions cannot be intermixed.
L1 ABT	Exclusively used by the application that is currently running on the core and is allocating into a particular region, since in each phase of an application most of the allocations are to a specific memory region, e.g. the nursery or the older generation.
L2 ABT	Each L2 ABT entry must also be extended with the corresponding allocation pool ID.

Table 5: Storage Cost of Hardware Structures for HAMM

Structure	Description	Cost
L1 RCCB	One table per core, indexed and fully-tagged with the object reference (virtual address). Each RCCB entry contains a VALID bit, the reference value (8-byte virtual address), a reference count delta (RCD, 4 bits, 2's complement), and the allocation pool ID (8 bits, also part of the tag). Each RCCB entry uses 77 bits, plus 2 extra bits per set for pseudo-LRU replacement. The L1 RCCB on each processing core is a 512-entry, 4-way set associative table.	4.8 KB per core
L2 RCCB	Similar to L1 RCCB, but is a 4K-entry, 4-way set-associative table, at the CMP level.	38.75 KB
L1 ABT	64-entry table, indexed by size class. Each entry has the block virtual address (8 bytes) and a VALID bit.	520 B per core
L2 ABT	256-entry, 4-way set-associative table, indexed by size class and tagged by allocation pool ID. Each entry has the block virtual address (8 bytes), the allocation pool ID (8 bits) and a VALID bit.	2.3 KB
General Purpose Registers	Extended with 1 Reference bit per register. Our machine configuration uses 128 physical registers.	128 bits per core
L1 Data Cache	Extended with 1 Reference bit per L1 data cache word.	1 KB per core
	Total for a 4-core CMP	66.5 KB

Table 6: HAMM Data Structures in Main Memory

Usage	Description
Memory ABT	64-entry pointer table per application and memory region. Assuming a typical generational heap with two regions (generations), the ABTs would require 1KB per running managed application.
Object Reference Map (ORM)	1K-entry table with 16 bytes per entry, totaling 16KB, per running managed application.
Configuration of each allocation pool	Table with constants required by HAMM: 1) offset of the reference count field in the object header, 2) offset of the size class field in the object header, 3) offset of the type identification field in the object header, 4) offset of the block starting address, all with respect to the address used in all references to the object, and 5) 64-entry table with the object size for each size class used in the memory region.

Table 7: Baseline processor configuration

Front End	64KB, 2-way, 2-cycle I-cache; fetch ends at the first predicted-taken branch; fetch up to 3 conditional branches or 1 indirect branch
Branch Predictors	16KB (31-bit history, 511-entry) perceptron branch predictor; 1K-entry, 4-way BTB with pseudo-LRU replacement; 512-entry tagged target cache indirect branch predictor; 64-entry return address stack; min. branch misprediction penalty is 20 cycles
Execution Core	4-wide fetch/issue/execute/retire; 128-entry reorder buffer; 128 physical registers; 64-entry LD-ST queue; 4-cycle pipelined wake-up and selection logic; scheduling window is partitioned into 4 sub-windows of 32 entries each
On-chip Caches	L1 D-cache: 64KB, 4-way, 2-cycle, 2 ld/st ports; L2 unified cache: 4MB, 8-way, 8 banks, 16-cycle latency; All caches: LRU repl. and 64B lines
Buses and Memory	150-cycle minimum memory latency; 32 DRAM banks; 8B-wide core-to-memory bus at 4:1 frequency ratio
Prefetcher	Stream prefetcher (32 streams and 4 cacheline prefetch distance)

To evaluate the effect of our mechanism on GC time, we need to determine the time spent on GC during the whole benchmark execution, because our proposal is aimed at eliminating a substantial number of software GC instances and also substantially changes the work done in each remaining software GC instance. Sampled simulation is not useful because we cannot infer the whole program GC time from microarchitectural results for a number of GC slices. The elimination of GC work is much more significant than the microarchitectural side effects of our proposal on the execution of the garbage collector, namely locality changes. Thus, we rely on the observation from Buytaert et al. [10] that GC time is an approximately linear function of the surviving objects. This first-order approximation is accurate at the system level because in the copying nursery collector, the work performed by the collector is roughly proportional to the *surviving memory space*, which has to be copied to the older generation. In the older-generation (mark-sweep collector with lazy sweep used in Jikes RVM), the work performed by the collector is proportional to the *number of surviving objects* because the collector actively performs the marking phase by tracing the survivors and the allocator sweeps blocks on demand.

Application behavior is affected by HAMM mainly in two ways. First, HAMM accesses main memory to update reference counts, scan objects, and update the ABTs. Even though these extra memory accesses are not on the critical path of execution and do not directly delay any instructions, they can delay the application by consuming memory bandwidth and possibly causing cache pollution. Second, reuse of memory blocks significantly changes the memory access pattern because objects are located at different addresses with respect to the memory layout of the baseline. That is, HAMM changes memory locality. To evaluate the effects of HAMM on application (*mutator*) performance, we use a cycle-accurate x86 simulator with the machine configuration shown in Table 7. We simulate slices of 200M instructions taken at the same algorithmic point from the application[9] in the baseline and with HAMM and compare their performance.

5. RESULTS

5.1 Garbage Collection Time

Figure 3 shows the normalized execution time spent in garbage collection in the baseline (baseline-GenMS) and with our proposal (GenMS + HAMM), computed with the GC time model explained in Section 4. The x-axis is the heap size relative to the minimum size (minHeap) that allows full execution of the benchmark on the baseline Generational Mark-Sweep collector (GenMS). Our proposal significantly reduces GC time for most benchmarks and heap sizes. The best improvement is 61% for `antlr` with heap size 1.5x minHeap. HAMM eliminates the need for software garbage collection in `fop` with 4x minHeap. The worst result is for `lusearch` at 1.2 minHeap, where HAMM slightly increases GC time by 0.6%.[10]

Figure 4 shows the estimated reduction in GC time for 1.5x minHeap to 3x minHeap, which are heap sizes that provide a reasonable trade-off between garbage collection overhead and memory requirements for most applications. On average, HAMM reduces GC time by 29% for 1.5x minHeap and 31% for 2.5 minHeap. The reduction in GC time is at least 10% for each benchmark in the 1.5x to 3x minHeap range.

[9] We make sure no garbage collection happens during the simulated slice.

[10] Even though HAMM reduces the number of nursery collections, the total space copied to the older generation —i.e. the actual work done by the nursery collector— increases by 6% in this case.

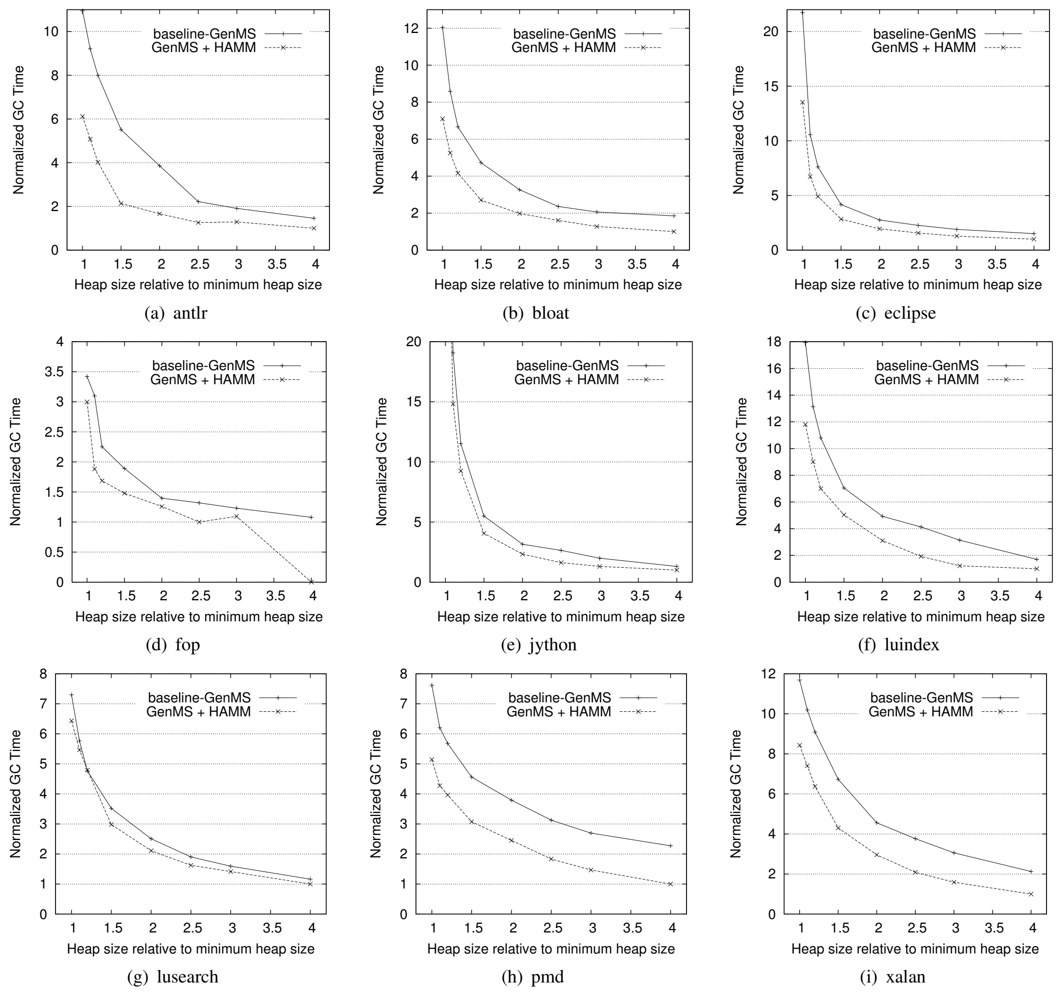


Figure 3: Estimated garbage collection time with HAMM compared to the baseline Generational Mark-Sweep collector.

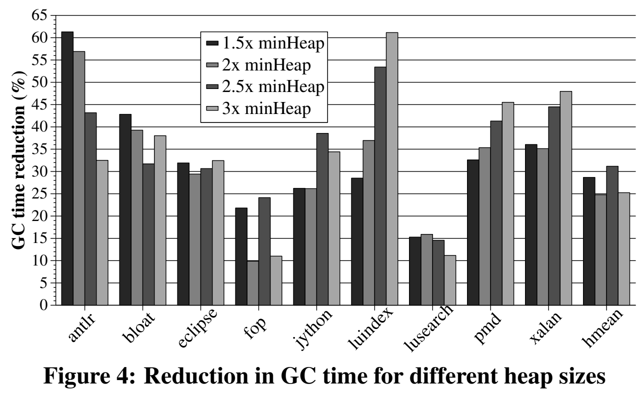


Figure 4: Reduction in GC time for different heap sizes

5.2 Where are the Benefits Coming from? Memory Block Reuse Facilitated by HAMM

To provide insight into the effects of our mechanism on GC time we analyze the memory block reuse enabled by HAMM. Figure 5 shows the number of nursery-allocated objects with HAMM for 1.5x minHeap, normalized to the baseline GenMS. The bar shows objects allocated by our mechanism and is split into the number of objects allocated to new space (via software) and reallocated to reused memory blocks (via HAMM). On average, 69% of the new objects reuse memory blocks in the nursery. It is worth noting that the reuse of nursery space does not directly translate into total GC time reduction because: (1) short-lived nursery objects are easily reused but they do not significantly increase the cost of nursery collections in the baseline because many of the nursery collections eliminated by HAMM are relatively fast, as long as there are not many surviving objects in the nursery, and (2) even though HAMM eliminates 52% of nursery collections on average (Figure 6), the remaining collections have to do more work, i.e. a larger fraction of the nursery is alive when it is collected and thus, more live objects are copied to the older generation. As we show in Figure 7, HAMM reduces bytes copied to the older generation by only 21% on average.

Figure 8 shows the number of objects allocated to the older generation with HAMM for 1.5x minHeap, normalized to the same number in the baseline GenMS. In general, HAMM reduces the total number of objects that are allocated into the older generation (by 21% on average) because by significantly delaying nursery collections, it gives more time to objects to die before they have to be promoted to the older generation. The bar in Figure 8 is split into objects allocated to new space and reallocated to reused memory blocks and shows great variation in memory block reuse in the older generation, ranging from 10% for `fop` to 70% for `xalan`, with an average of 38%. The number of full-heap collections, shown in Figure 9, is reduced (by 50% on average) with HAMM because the reuse of memory blocks —both in the nursery and in the older generation— delays filling up the heap. The space occupied by objects surviving

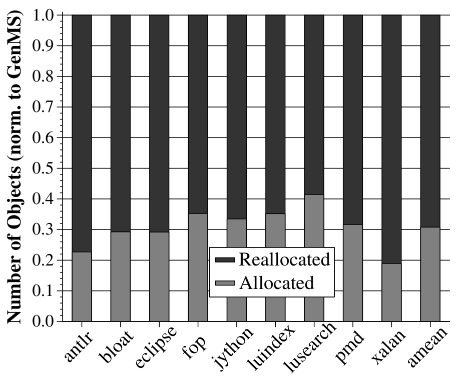


Figure 5: Objects newly allocated and reallocated in the nursery

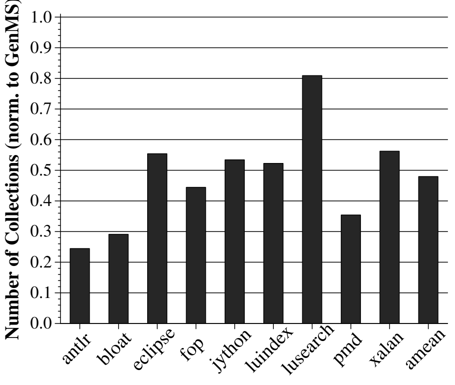


Figure 6: Number of nursery collections

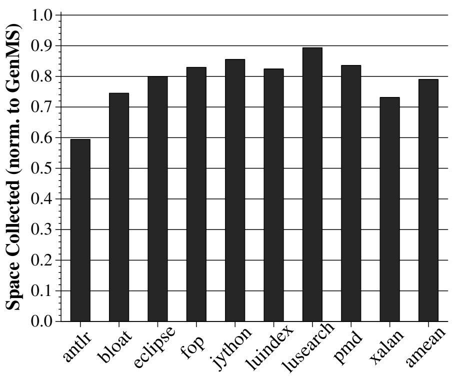


Figure 7: Total surviving space copied during nursery collections

full-heap collections is shown in Figure 10.[11] On average, full-heap collections with HAMM find 49% less surviving space compared to the baseline GenMS, which reduces the amount of time spent in each full-heap collection. By reducing both the number and time of full-heap collections, HAMM reduces GC overhead.

We conclude that HAMM significantly reduces GC time because 1) it reduces the number of nursery collections by 52% and even though each collection does more work, the total space copied to the older generation is reduced by 21%, 2) it reduces both the number (by 50%) and total surviving space (by 49%) of full-heap collections.

5.3 Analysis of Garbage Collection Results

We analyze the behavior of some representative benchmarks, based on the data from the previous sections.

`Antlr` is a parser that does not accumulate many long-lived objects over its execution [6], making it a very good candidate for the baseline generational collector. With smaller heaps, e.g. 1.5x minHeap, HAMM reduces GC time by 61% because it allows the allocator to reuse space for 77% of the new objects in the nursery, eliminating 76% of the nursery collections and *all* of the full heap collections. With larger heaps (e.g. 3x minHeap), the nursery also proportionally grows, increasing the time between nursery collections and giving enough time to more objects to die. Thus, the overhead of the software GC is reduced because fewer objects survive nursery collections and have to be copied to the older generation. Even though the baseline GenMS collector is very efficient with larger heaps, HAMM still reduces GC time by 32% with 3x minHeap.

`Fop` accumulates long-lived objects, almost linearly increasing the required heap space over time, but it does not require full-heap collections for 1.5x minHeap or larger. Most objects die very young, making the baseline generational collector very efficient. However, HAMM still eliminates 22% of the GC overhead for 1.5x minHeap, because it reuses 65% of the memory blocks in the nursery and eliminates 56% of nursery collections. With 4x minHeap, HAMM completely eliminates the need for software GC because block reuse prevents the nursery from growing up to the size that triggers GC.

`Xalan` iterates over allocating a significant number of objects and releasing most of them all together. As heap size increases from 1.5x to 3x minHeap, HAMM's GC time reduction also increases from 36% to 48% because memory block reuse is very significant (81% for the nursery and 70% for the older generation, for 1.5x minHeap).

5.4 Effect on Mutator Performance

Figure 11 shows the impact of HAMM on the performance of the mutator. The average performance degradation is 0.38%, mainly due to the modified memory behavior. Our proposal changes the memory behavior of the mutator in two ways. First, it introduces extra memory accesses to (1) update the reference counts in object headers, (2) scan dead objects for pointers to recursively decrement their reference counts, and (3) update the ABT and handle the linked lists of free blocks. Second, by reusing memory blocks instead of using the default allocation algorithm, HAMM changes the memory access pattern of the application in difficult-to-predict ways. For the nursery, the changes in locality introduced by our memory reuse mechanism are similar to hybridizing the default bump-pointer allocator with a free-list allocator.

Our experiments show that spatial locality in the mutator is slightly degraded for most benchmarks, i.e. HAMM increases L1 cache misses by up to 4% (Figure 12), which in turn increases the pressure on the L2 cache. The exceptions are `jython` and `luindex`, where L1 cache misses are reduced by 78% and 58%, respectively. In these benchmarks, HAMM improves spatial locality because the quick reuse of memory blocks for new objects allows better utilization of the available cache space. However, this effect does not appear at the L2 cache level (Figure 12), where cache misses increase for all benchmarks by up to 3.4%, leaving only `luindex` with a 0.3% performance improvement. Even though only 0.3% to 5% of the additional memory accesses introduced by HAMM become main memory accesses, HAMM itself is responsible for 0.6% to 6.8% of L2 cache misses (Figure 13). These extra cache misses do not have a significant impact on performance because they are not on the critical path of execution and do not directly delay any instruction.

We conclude that the effect of HAMM on the mutator is negligible. Hence, HAMM significantly improves overall performance because it reduces GC time while keeping mutator time nearly constant. The reduction of GC overhead with HAMM also enables running applications with smaller heap sizes with a reasonable overhead, which is a significant advantage on memory-constrained systems.

6. RELATED WORK

We describe relevant related work in software-based reference counting, hardware-based garbage collection, and compiler analyses and optimizations. In contrast to most previous work, we provide flexible hardware support for garbage collection, rather than fully implementing garbage collection in hardware.

6.1 Software-Based Reference Counting

Software approaches have been proposed to reduce the overhead of updating reference counters. Deutsch and Bobrow [16] first proposed deferred reference counting, i.e. ignoring RC updates on register and stack variables and examining them later. Levanoni and Petrank [24] proposed coalescing multiple updates to the same pointer slot between reference counting collections. Blackburn and McKinley [9] proposed a hybrid generational collector using RC only for the older generation and updating references from the nursery only

[11] Figures 9 and 10 do not include `fop` because it does not require any full-heap collection even with the baseline GenMS collector. Also, the bars for `antlr` are zero because HAMM actually eliminates all full-heap collections.

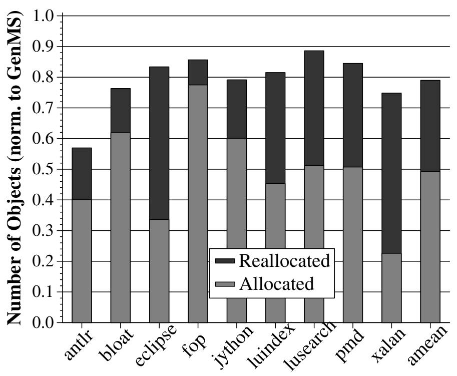


Figure 8: Objects allocated and reallocated in the older generation

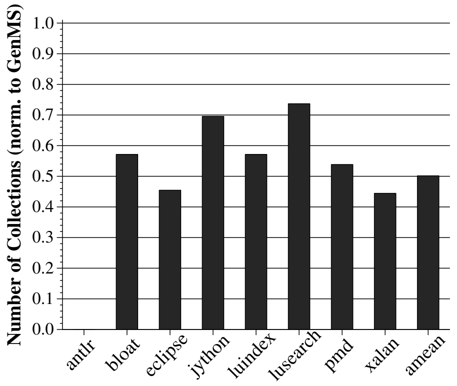


Figure 9: Number of full-heap collections

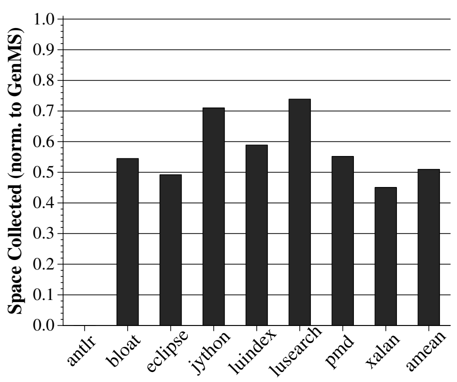


Figure 10: Total surviving space during full-heap collections

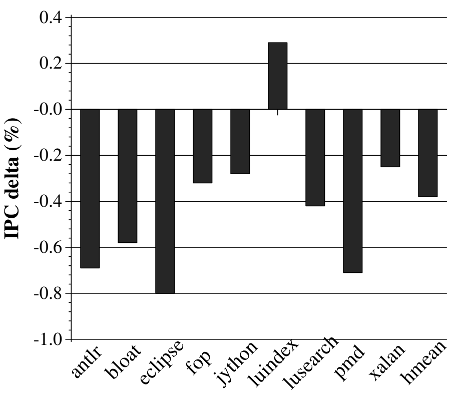


Figure 11: Mutator performance

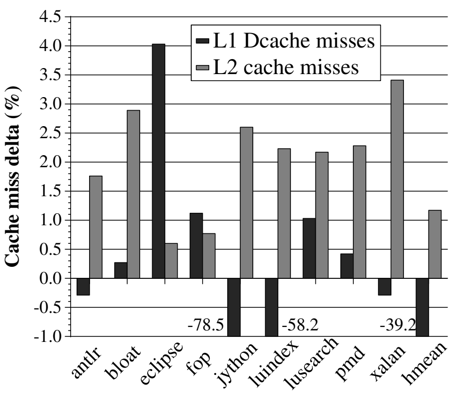


Figure 12: L1D and L2 cache misses

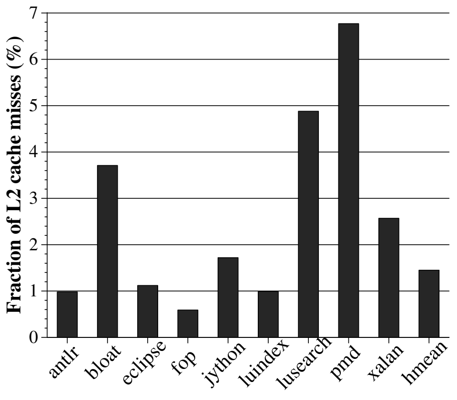


Figure 13: L2 misses due to HAMM

when it is collected. In general, these approaches give up immediate reclamation of dead objects and require an explicit garbage collection phase to process pending RC updates and find dead objects. In contrast to these approaches we propose a hardware accelerator that uses reference counting to reduce the need for software GC. Our proposed flexible hardware and architectural support can be extended to be used by these software approaches to reduce the overhead of reference counting even more.

6.2 Hardware-Based Garbage Collection

Schmidt and Nielsen [32] proposed a hardware-based GC mechanism based on garbage-collected memory modules (GCMM), which include both RAM and a microprocessor capable of running a real-time copying collection algorithm locally on the GCMM. Wise et al. [36] proposed a hardware self-managing heap memory using *reference counting memory* modules that include reference counters, logic to generate RC increments and decrements when pointers are overwritten and logic to perform GC. Both of these designs require the whole heap to be allocated to custom memory modules and prevent the software from implementing different allocation and collection policies. In contrast, our proposal works with a general-purpose architecture using standard memory modules and does not tie the hardware to a particular allocator and GC implementation.

Peng and Sohi [31] first proposed in-cache reference counting to assist GC. Chang and Gehringer [11] proposed a reference counted cache managed by a coprocessor, which handles object allocation and in-cache collection for one memory region at a time. Memory space used by objects that are allocated and spend all their lifetime in the object cache can be reused. In contrast, HAMM is not limited to objects in the cache, can handle multiple memory regions, and can be used in multiprocessor systems.

Meyer [29, 30, 28] and Stanchina and Meyer [33] proposed concurrent garbage collection algorithms based on a full hardware coprocessor implementation targeting embedded systems. Objects are known to the hardware and pointers are a native data type. The garbage collection coprocessor is microprogrammed to perform a single algorithm, while our proposal can support different general-purpose garbage collection algorithms. Pure hardware GC implementations limit the flexibility of software GC algorithms. Our key difference is that we provide acceleration support for software GC rather than using only a rigid hardware GC implementation. As a result, our proposal is (1) more flexible, (2) less costly and complex in terms of hardware since it does not require a full GC implementation or custom memory, and (3) is fully-compatible with and does not change the performance of applications that use pure-software collection instead of our architectural support.

Click et al. [14] described Pauseless GC, a concurrent GC with partial compaction supported by efficient hardware read barriers. There are two major differences between Pauseless GC and HAMM. First, HAMM and Pauseless GC target different design points. HAMM targets reducing GC time and resources required by GC in the general-purpose domain, complementing a wide spectrum of software GC algorithms. Pauseless GC includes hardware tightly coupled to their software GC, and is part of a custom platform (from processor to OS and JVM) specifically designed for high throughput and short pause times on servers, with hundreds of cores and huge heaps. Second, Azul's hardware (read barrier and GC traps) does not reduce software GC work as HAMM does, but only mitigates overheads on application threads. Click et al. does not mention their GC cost in terms of processor/memory resources, which is our main interest. We note that both approaches can be complementary. For example, if HAMM is used to reduce resource requirements of Pauseless GC, more cores/cycles will be available to the server application.

6.3 Compiler Analyses and Optimizations

Joisha [21] presented a static analysis and optimization technique called RC subsumption that reduces the number of RC updates for stack references on a non-deferred RC collector. Joisha [22] generalized several optimizations under the idea of overlooking roots and provided a flow-sensitive intraprocedural analysis technique that allows non-deferred RC to perform close to deferred RC in most benchmarks. These compiler optimizations can be combined with HAMM to significantly reduce the number of RC update operations due to stack references, reducing the pressure on the RCCB hierarchy.

Stack allocation [13] places objects that the compiler knows will only live inside a function on the stack instead of on the heap. These objects are automatically "collected" without any overhead when the stack frame is invalidated, i.e. when the object goes out of scope. Stack allocation requires accurate escape analysis to decide for each allocation site that no reference to an object can remain live after the function ends. Also, stack-allocated objects must go out of scope to reclaim their memory. HAMM does not require any compiler analyses and is able to reclaim the space of any object —not just local objects— as soon as there are no more live references to it.

Guyer et al. [18] proposed Free-Me, a static analysis technique that safely includes explicit object reclamation function calls (free) in managed code, when the compiler can be sure an object is dead. Unlike HAMM, compiler analyses are limited by the accuracy and scope of pointer analysis and can typically reclaim only very short-lived objects. Free-Me could be synergistically combined with HAMM because it can discover dead objects before all their references go out of scope or are overwritten, enabling earlier memory block reuse.

7. CONCLUSION

We introduced HAMM, a cooperative hardware-software technique that allows quick reallocation of memory blocks by finding dead objects with hardware-assisted reference counting. HAMM provides new primitives for faster garbage collection in hardware without limiting the flexibility of software collectors that can be built on top of the primitives. Our evaluation shows that HAMM reduces the execution time spent on garbage collection by 31% on average for a set of the Java DaCapo benchmarks running on Jikes RVM, a state-of-the-art research virtual machine with a state-of-the-art generational garbage collector. In addition to significantly reducing the performance overhead of GC for heap sizes appropriate for a state-of-the-art software GC, HAMM allows applications to run with much smaller heap sizes with a reasonable overhead, which is a significant advantage on memory-constrained systems. We believe HAMM opens up possibilities to explore new garbage collection algorithms that can take better advantage of the proposed architectural support.

Acknowledgments

We thank Cliff Click, Hyesoon Kim, Kathryn McKinley, Erez Petrank, Bjarne Steensgaard, Greg Wright, Ben Zorn, Chang Joo Lee, Eiman Ebrahimi, Rustam Miftakhutdinov, Veynu Narasiman, Aater Suleman, other members of the HPS research group, and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the support of the Cockrell Foundation, Microsoft Research and Intel Corporation. Part of this work was done while José Joao and Onur Mutlu were at Microsoft Research. We also acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources.

REFERENCES

[1] A.-R. Adl-Tabatabai et al. Improving 64-bit Java IPF performance by compressing heap references. In *CGO'04*.

[2] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[4] A. W. Appel. Simple generational garbage collection and fast allocation. *SPE*, 19(2):171–183, 1989.

[5] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[6] S. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*.

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. *SIGMETRICS Perform. Eval. Rev.*, 32(1):25–36, 2004.

[8] S. M. Blackburn et al. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, 2008.

[9] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: fast garbage collection without a long wait. In *OOPSLA'03*.

[10] D. Buytaert, K. Venstermans, L. Eeckhout, and K. D. Bosschere. Garbage collection hints. In *HIPEAC'05*.

[11] J. M. Chang and E. F. Gehringer. Evaluation of an object-caching coprocessor design for object-oriented systems. In *ICCD'93*.

[12] W. Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In *PLDI'06*.

[13] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *OOPSLA'99*.

[14] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *VEE'05*.

[15] G. E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, 1960.

[16] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, 1976.

[17] S. Z. Guyer and K. S. McKinley. Finding your cronies: static analysis for dynamic object colocation. In *OOPSLA'04*.

[18] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-Me: a static analysis for automatic individual object reclamation. In *PLDI'06*.

[19] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *OOPSLA'04*.

[20] J. A. Joao, O. Mutlu, and Y. N. Patt. Flexible reference-counting-based hardware acceleration for garbage collection. Technical Report TR-HPS-2009-001, The University of Texas at Austin, Apr. 2009.

[21] P. G. Joisha. Compiler optimizations for nondeferred reference-counting garbage collection. In *ISMM'06*.

[22] P. G. Joisha. Overlooking roots: a framework for making nondeferred reference-counting garbage collection fast. In *ISMM'07*.

[23] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[24] Y. Levanoni and E. Petrank. An on-the-fly reference counting collector for Java. In *OOPSLA'01*.

[25] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[26] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

[27] J. McCarthy. History of LISP. In *History of programming languages I*, pages 173–185. ACM, 1981.

[28] M. Meyer. A true hardware read barrier. In *ISMM'06*.

[29] M. Meyer. A novel processor architecture with exact tag-free pointers. *IEEE Micro*, 24(3):46–55, 2004.

[30] M. Meyer. An on-chip garbage collection coprocessor for embedded real-time systems. *RTCSA*, 00:517–524, 2005.

[31] C. Peng and G. S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, CS Dept., University of Wisconsin-Madison, 1989.

[32] W. J. Schmidt and K. D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ASPLOS'94*.

[33] S. Stanchina and M. Meyer. Mark-sweep or copying?: a "best of both worlds" algorithm and a hardware-supported real-time implementation. In *ISMM'07*.

[34] D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

[35] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Not.*, 19(5):157–167, 1984.

[36] D. S. Wise, B. C. Heck, C. Hess, W. Hunt, and E. Ost. Research demonstration of a hardware reference-counting heap. *Lisp and Symbolic Computation*, 10(2):159–181, 1997.