Concurrent Collection as an Operating System Service for Cross-Runtime Cross-Language Memory Management

Michal Wegiel Chandra Krintz
Computer Science Department
Univ. of California, Santa Barbara

UCSB Technical Report 2010-15, July, 2010

ABSTRACT

We present GC-as-a-Service (GaS), a cross-runtime, cross-language garbage collection (GC) library that can be used to simplify the implementation of runtime systems, and that exploits available multicore technologies. GaS decouples GC from other runtime components and exposes a fine-grain API for use by GC-cooperative runtimes of different programming languages for heap memory management. GaS provides concurrent, on-the-fly GC and avoids moving objects for use as a precise or conservative collector. We integrate GaS within production-quality runtime systems for Python and Java. Our experimental evaluation shows that using GaS as an alternative to tightly integrated GC introduces modest overhead and that GaS reduces pause times significantly for Python and Java programs.

1. INTRODUCTION

Managed Runtime Environments (MREs, Virtual Machines, VMs) for high-level, object-oriented (OO) programming languages are increasingly complex, which makes them challenging to architect, extend, and understand. One of the most complex components in MREs is automatic memory management (garbage collection, GC). State-of-the-art GC algorithms, i.e. parallel, concurrent, and on-the-fly GCs [33, 22], capable of taking advantage of multi-core processors, are notoriously difficult to implement, especially in conjunction with other MRE components (loaders, compilers, schedulers, etc).

One way to address GC complexity is to decouple, modularize, and facilitate reuse of GC implementations [8, 7, 6, 5, 21]. In this paper, we investigate the design and implementation of a portable GC library (which we call GC as-a-service (GaS)). GaS represents a different point in the GC design space because of its unique combination of goals: (i) cross-MRE GC library for static/dynamic languages (ii) modern GC (concurrent, on-the-fly) for *cooperative* MREs (unlike Boehm GC [7]) (iii) GC-MRE decoupling (unlike recent on-the-fly GCs [16, 17]) (iv) low-overhead interface using C-based native API (unlike MMTk and GCTk [5, 6]).

We employ the GaS library within production-quality MREs for Java (HotSpot Java Virtual Machine (JVM)) and Python (cPython) and compare GaS GC against state-of-the-art GCs. Our empirical evaluation includes concurrent, parallel, tracing GCs as well as hybrid tracing/reference counting GCs. We discuss the trade-offs we make with the GaS design and their performance implications. We also investigate the performance of other approaches that provide GC across languages such as those that cross language boundaries and that employ a single MRE for multiple languages.

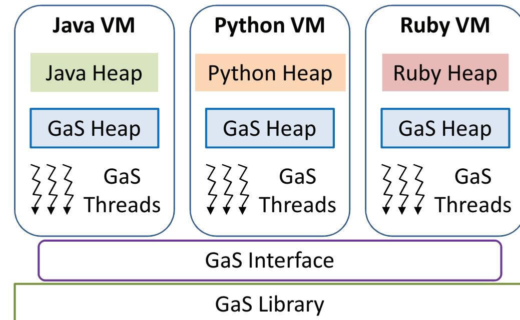

Figure 1: GaS architecture.

2. GaS OVERVIEW

Figure 1 presents the high-level architecture of GaS. GaS provides a shared C library that is accessible via the GaS interface and that can be used by MREs for different languages (e.g. Java, Python, Ruby) to integrate garbage collection (GC) into the runtime. Each MRE dedicates some number of threads to GaS GC (concurrent, on-the-fly GC) and maps a virtual memory region which GaS manages. MREs also have the option of allocating certain types of objects (e.g. immortal objects or internal data structures) in their private heaps and managing them independently of GaS.

We design GaS to support MREs for dynamic and static languages which implement diverse memory management strategies, including reference counting, tracing, object-moving, and non-moving GCs. Our goal is to enable GC portability at the library (i.e. binary) level (without recompiling the library, or modifying the GC algorithm).

The rationale behind GaS is to enhance modularity and separation of concerns in the design and implementation of MREs and to enable building new MREs from reusable components. GaS abstracts away the GC functionality, thus enabling construction of an MRE with a modern GC subsystem without expert knowledge about concurrent and on-the-fly GCs. By treating GC as a component, GaS facilitates research in other, non-GC, MRE subsystems. In addition, GaS enables integration of a high-quality GC into MREs that lack modern GCs, e.g. scripting language MREs that employ stop-the-world, single-threaded collectors (reference

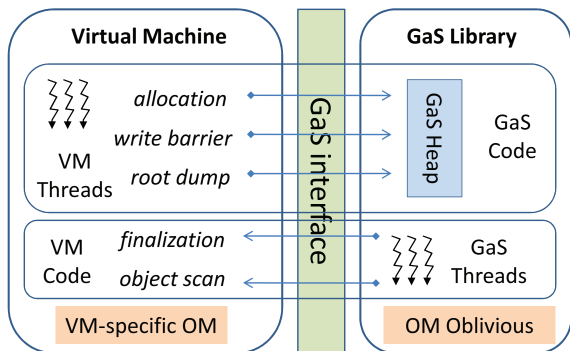

Figure 2: GaS interface.

counting with cycle detection for Python and PHP, mark-sweep for Ruby). In Section 6 we show that even highly-optimized, sophisticated MREs, such as HotSpot JVM, can benefit from GaS.

3. GaS DESIGN

GaS is a parallel (i.e. uses multiple GC threads), concurrent (i.e. collects most objects without stopping the application threads (mutators)), and on-the-fly (i.e. stops one thread at a time) GC. The rationale behind this configuration is that concurrent, on-the-fly GCs are difficult to implement, thus it is practical to provide such GCs as a service/library. In addition, many MREs are latency-sensitive, e.g. Ruby is used for server-side scripting and its stop-the-world GC is a limiting factor – concurrent GCs avoid stop-the-world collection which can introduce large pauses. Finally, as multi-core processors become ubiquitous, concurrent GC is increasingly suitable for fully utilizing and extracting high performance from modern systems.

GaS does not move objects because some MREs (e.g. Python) assume that object addresses remain constant and others (e.g. Mono) require support for object pinning and conservative root scan. GaS uses free-list allocation and thread-local allocation buffers (TLABs) for fast, unsynchronized, bump-pointer allocation in the common case. TLABs are vital for supporting multi-threaded MREs.

The GaS GC algorithm is an adaptation of extant snapshot-at-the-beginning (SATB) on-the-fly GC [16, 17, 14, 15]. Our extensions decouple GC from the MRE and simplify the MRE-GC interface on the MRE side. Existing on-the-fly GCs rely on system-wide handshakes with mutator threads and maintain per-thread buffers to implement write barriers and to determine quickly if another marking iteration is needed [16, 13]. GaS avoids such tight-coupling and moves GC logic out of the MRE as much as possible.

3.1 GaS Interface

Figure 2 depicts how MREs interact and cooperate with GaS. An MRE first initializes the GaS library by specifying the number of GC threads, TLAB size, and GC threshold (percentage heap usage that triggers a GC), and by providing a mapped virtual memory region for the GaS heap. The GaS interface consists of operations performed by MRE threads (allocation, write barrier, and root dump) and by the GaS threads (finalization and object scan).

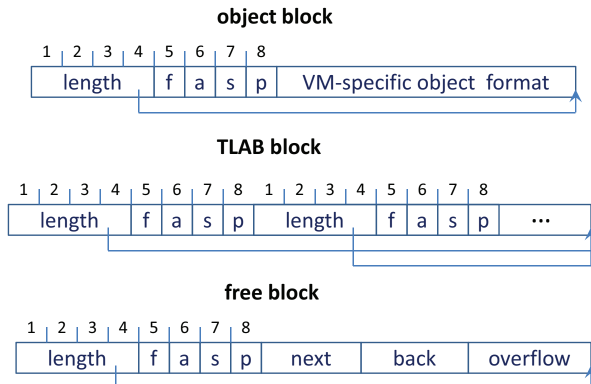

Figure 3: Block format in the GaS heap.

An MRE requests TLABs from GaS and performs most allocations within a TLAB. To allocate large objects, an MRE requests a TLAB of a specific size and then proceeds to intra-TLAB allocation. The GaS protocol for allocation and write barrier (described in detail in Section 3.2) is kept to a minimum so that the compiler can inline this code at allocation and reference store sites.

Before each GC, GaS requests a root dump. An MRE responds to this request by identifying objects (for GaS to mark) in the GaS heap that are reachable from thread stacks, global memory areas, and/or non-GaS generations. GaS invokes MRE-provided callbacks to scan objects for references and to indicate that a particular object is about to be reclaimed (to support finalization).

3.2 GaS Heap Layout

We divide the GaS heap into blocks. Each block starts with a header, whose format is shown in Figure 3. The header size is one machine word (we assume 64bit words) so that it can be atomically loaded/stored. GaS supports fully-concurrent unsynchronized sequential scans over heap blocks.

There are three block types: an object block, TLAB block, and free block. The block header consists of 5 fields: block length (4 bytes), block format (f, 1 byte), and three 1-byte GC flags: recently-allocated (a), scanned (s), and pending (p). We make each field at least 1-byte in size so that we can use atomic read/write (most architectures support single-byte atomic memory access but do not support bit-wise atomic access).

Object blocks are followed by an MRE-specific object representation, which is not interpreted by GaS. Thus, GaS adds one word of space overhead per object. GC flags have meaning only for object blocks. New objects have their recently-allocated flag set. Whenever the GaS GC marks a live object, it sets its scanned flag. Objects with their pending flag set will be scanned by the collector.

We initialize each word in a TLAB block so that we can treat it as the start of a new, shorter TLAB. For example if the first TLAB word contains length = 8, then the second TLAB word contains length = 7, etc. This approach enables atomic allocation of objects in TLABs. To allocate an object spanning 5 words, we simply store a new object header (with length = 5) at the beginning of the TLAB. Such a

store happens atomically and the remaining part of a TLAB immediately has the right TLAB header (with the correct, shorter length). Thus, an ongoing concurrent heap block traversal cannot be confused by object allocation when we transition from a TLAB block to an object block. In addition, object allocation amounts to a single word store which the compiler inlines.

The length of object/TLAB blocks does not use the entire machine word. However, the limit of 16GB per object is typically sufficient in practice (e.g. in Java an object cannot exceed 16GB). Free blocks can use larger length values by storing their actual length in the overflow field (which has machine-word width).

When a TLAB fills up, we retire it (we insert a dead object into the remaining free space) and replace it with a new TLAB. TLAB allocation, like all freelist operations, employs synchronization. The freelist is a double-linked list of free blocks.

GaS uses a conditional SATB [16, 17] write barrier, that it executes before each store. The barrier first loads the previous pointer value (about to be overwritten by a store), checks if it belongs to the GaS heap, and if so sets the pending flag on the corresponding object. For example, before a store $*p = v$ happens we execute:
$if\ (is_in_gas_heap(*p))\ set_pending_flag(*p);$
For efficient heap membership checks, the MRE should map the GaS heap above or below all other object regions in an MRE – in such a setting a single border comparison suffices.

3.3 GaS GC Algorithm

GaS GC comprises four concurrent phases: flag clearing, root dump, object marking, and object sweeping. GC threads use barrier synchronization to meet at subsequent GC phases. GaS imposes no pauses if an MRE is capable of performing a root dump without halting the mutator threads. A new GC cycle starts once the heap usage crosses the specified GC threshold.

We do not use a marking bitmap but instead mark object headers (the scanned flag) directly. This enables us to avoid atomic compare-and-swap (CAS) operations during marking because one byte can be stored atomically. Since we do not synchronize GC threads during marking, multiple GC threads may end up scanning the same object – we find that this happens rarely and we mitigate it via dynamic load balancing among the GaS GC threads.

Flag Clearing. Flag clearing is a concurrent phase where a single GC thread traverses over the heap blocks and clears the GC flags. This step has a similar effect to activating the snapshot mode in extant SATB GCs [16, 17]. However, in GaS, the snapshot mode is active all the time, meaning that all objects are allocated live (the recently-allocated flag set) and mutators always use a SATB write barrier (setting the pending flag for objects whose incoming pointers are overwritten). This approach simplifies the MRE-GC protocol and decouples GC and an MRE (no handshakes, state-dependent write barriers, etc., are required).

During flag clearing, GaS computes a balanced heap partitioning used in the subsequent, parallel heap traversals. GaS divides the heap into equal-size chunks at block boundaries. In later traversals, each GC thread uses its own chunk only.

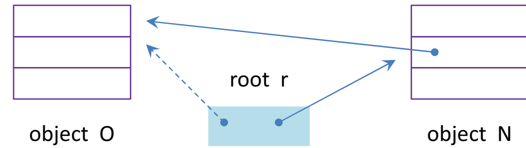

Figure 4: Root updates and concurrent marking.

Root Dump. In the second GC phase, an MRE finds roots into the GaS heap and reports them to GaS by setting the pending flag for root objects. Depending on the MRE, root dump may require scanning registers, thread stacks, and global memory areas. An MRE may need to stop the mutator threads to find roots. Since GaS is an on-the-fly GC, an MRE is allowed to stop one thread at a time to avoid long pauses. In Sections 3.4 and 3.5, we describe how root dump can be done efficiently in MREs using tracing and reference counting, respectively.

Marking. Object marking is parallel and concurrent. Due to concurrent object mutations, GaS occasionally performs several marking iterations before converging to a stable live object graph. In each iteration, every GC thread scans its own heap chunk for objects with the pending flag set. If no such objects are found by the concurrent block traversal, the marking phase is complete. Pending objects that GaS finds are recursively (using depth-first search) scanned and marked (by setting the scanned flag). Recursive marking stops on already-scanned objects (potentially marked in previous marking iterations). GaS uses dynamic load balancing during marking (randomized work stealing) for scalability. GaS marks objects in-place (i.e. uses object headers) and, unlike some SATB GCs, does not use per-mutator marking buffers (to further decouple GC from the threading subsystem).

During the 2nd and later marking iterations, recursive marking stops on already-marked objects and on recently-allocated objects (the 1st iteration stops only on already-marked). This guarantees GC termination. Assuming there is N objects in the heap when the GC cycle starts, and all new objects are flagged as recently-allocated, GC will finish after N iterations at most. In practice 2 or 3 iterations suffice.

Figure 4 explains why this strategy is correct, i.e., it cannot lead to leaving some live objects unmarked. Since we stop the 2nd and later iterations of marking on recently-allocated objects, we need to guarantee that it is impossible that a recently-allocated object has a pointer to a live object that is otherwise unreachable and is not flagged as pending. Note that this is possible during the first marking, when we mark from roots. Consider an example in Figure 4. Root r initially points to object O. Then, object N is allocated, and a pointer in N is set to point to object O. Next, root r is updated to point to N. Now we have a configuration where O is reachable only through N. Note that N is recently-allocated and still needs to be scanned. The reason for this is that our snapshot write barrier (SATB WB)

does not capture root pointer updates (only heap pointer updates). However, the 2nd and later marking iterations ignore roots and mark from pending objects only. Thus, the newly-allocated objects do not have to be scanned once the first marking iteration completes. Reconsider our example in Figure 4 but assuming that r is not a root but a field in a heap object. On r update, object O is flagged pending and thus will be scanned by GC even if we do not scan N.

Sweeping. Sweeping is parallel and concurrent. Each GC thread scans its heap chunk in an attempt to find a potentially-free block (i.e. either a freelist block or a dead object). This step is done without synchronization with mutators which perform concurrent allocation and might use free blocks in the meantime. Once a GC thread finds a potentially-free block, it acquires the freelist lock and continues scanning as long as it encounters reclaimable blocks (dead objects or free blocks). If the GC thread finds a contiguous region of sufficient length, it coalesces the region into a single free block and adds it to the freelist. Immediately prior, the thread invokes the finalizer on all dead objects. If a finalizer resurrects an object (the MRE $finalize$ callback indicates this to GaS), then the object will be finalized again once it becomes unreachable next time. Finally, the GC thread releases the freelist lock and looks for another potentially-free block in its chunk.

3.4 GaS and Tracing GC

Incorporating GaS into tracing MREs is relatively straightforward because such MREs already implement support for object scanning, root dump, and asynchronous finalization. Generational MREs in addition support card tables/remembered sets and write barriers.

In generational MREs, we extend the card table (or remembered sets) so that it is possible to quickly find not only inter-generational pointers but also pointers into the GaS heap. A minor collection then suffices to implement the root dump operation in GaS.

In non-generational MREs, we add a write barrier that captures pointers leading into the GaS heap as they are created. For each reference store we check if the new pointer points into the GaS heap and, if so, flag the object it points to as pending. After the flag clearing phase, GaS concurrently scans the memory regions in the MRE that might contain GaS roots, and relies on the write barrier to deal with roots that go by GaS unnoticed during the scan.

3.5 GaS and Reference Counting GC

Reference counting MREs associate a reference count with each object and rely on two operations: $incref$ (increment the count) and $decref$ (decrement the count) to detect and reclaim dead objects. Such MREs cannot reclaim cycles unless cycle detection is run periodically. Each reference update in the heap or on the stack invokes $decref$ for the old reference value and $incref$ for the new reference value.

To integrate GaS into a reference counting MRE, we make the $incref$ and $decref$ operations conditional. For pointers belonging to the GaS heap that point to an object in the GaS heap, we do not use reference counts. In all other cases $incref$ and $decref$ have their original semantics. In particular, outgoing and incoming pointers in the GaS heap are subject to reference counting and so are pointers outside of the GaS heap.

In this design, all objects in the GaS heap whose reference count is non-zero are roots for GaS GC (because they are pointed to from outside of the GaS heap). Thus, the root dump operation amounts to a concurrent scan of the GaS heap in search of objects with non-zero reference counts. Note that no pauses are required for a root dump. To deal with the race condition that might hide a root from GaS, we introduce a write barrier in $incref$: if the reference count goes from 0 to 1, we flag the object as pending. Thus, if a root scan sees reference count of 0, which later becomes 1, we do not miss a root.

The SATB write barrier piggybacks on $decref$ and is only needed in case the decrement is performed in the GaS heap. In addition, we modify $decref$ so that it does not call the object finalizer if the reference count drops to 0 in the GaS heap (GaS calls finalizers during sweeping).

3.6 GaS Extensions

Although GaS is a non-moving GC, we can extend it to perform (non-moving) generational collection. Instead of physical partitioning of the heap, we employ logical partitioning. Each object has an age field, incremented during each GC cycle until the object becomes old. Minor GCs mark only young objects and stop on old objects. A write barrier identifies old objects that contain pointers to young objects.

To support conservative GCs, we extend GaS with an object start array that enables GaS to quickly determine if a given address is the start of an object. GaS does not need to update pointers thus conservative roots do not pose a problem. GaS computes the object start array during the clearing phase and uses it during the root dump phase.

4. IMPLEMENTATION

We have implemented the GaS library in C and have integrated it into the HotSpot JVM 1.6 and cPython 3.1. The HotSpot JVM uses a generational heap layout while cPython employs hybrid reference counting/cycle detection. Both VMs use 2-word object headers. HotSpot employs safepoints for root scan, which halt all mutators, and uses a three-level, circular, unified object/class model.

Our implementation of the GaS GC assumes sequential consistency, i.e. there is global order on writes and all threads see the same order. We use memory fences after the root dump phase to ensure store visibility. We use POSIX synchronization primitives (barriers, mutexes, and condition variables).

In HotSpot, we inline the GaS write barrier and object allocation in the template interpreter and in the code generated by the server (C2) compiler. We map the GaS heap at the constant border above all other generations, which reduces the membership checks to comparing a register with a constant. We use minor GC (based on parallel copying in the young generation) to find roots in thread stacks. For roots in other generations, we perform concurrent generation scan and introduce a write barrier to capture pointers into the

GaS heap. We have found this approach to result in shorter pause times than if we instead leverage card tables (we discuss these alternatives in Section 3.4). We use the GaS heap for the young and old generation and leave the permanent generation as part of the MRE-private heap.

In cPython, we extend C macros INCREF and DECREF to implement conditional reference counting. We synchronize the GC and the VM interpreter after root dump and before marking by acquiring and immediately releasing the global interpreter lock (to ensure all write barriers have finished executing). cPython does not have safepoints and thus GaS imposes no pauses. Note that regular cPython does impose pauses for (1) cycle detection and (2) whenever freeing large data structures after $decref$. The GaS heap is located at a fixed precompiled address in the virtual memory. We implement GaS support in cPython for a single data structure: the binary search tree, which is sufficient to evaluate GaS using our benchmark described in detail in Section 6.

5. RELATED WORK

The work most related to GaS is the Boehm GC [8, 7]. Boehm GC is a widely-used GC library providing a conservative collector for uncooperative runtimes (such as C and C++). Boehm GC supports stop-the-world serial and parallel collection. In contrast, GaS focuses on concurrent, on-the-fly GC for cooperative runtimes (precise roots, write barriers, TLAB allocation etc.) Moreover, the GC interface in Boehm GC essentially consists of two functions GC_MALLOC and GC_REALLOC. GaS interface is more fine-grain to be able to leverage runtime type-safe mechanisms for object scanning, finalization, and root dump (GaS and the MRE cooperate to a greater degree).

GC frameworks such as UMass GC Toolkit [21], GCTk [6], and MMTk [5] are different from and complementary to GaS. The UMass GC Toolkit (designed in the context of persistent Smalltalk and Modula-3) focuses on generational copying stop-the-world GC algorithms. GaS addresses concurrent, on-the-fly GC. GCTk, and MMTk are GC frameworks written in Java, created in the context of the Jikes RVM. Their goal is to support a number of different GCs to enable their comparative evaluation and GC research.

GCTk/MMTk have been used for non-Java languages, although such porting is not well-documented in the literature. For languages other than Java, however, these frameworks require crossing the C-Java language boundary for each GC operation (or translation/reimplementation of the entire framework). Crossing the C-Java language boundary incurs high overhead (in Section 6.4 we investigate the overhead of such crossings) and is therefore impractical for C-based runtimes (of which most MREs are).

GaS takes an alternative approach – the GC library and interface are written in C and do not require execution of an additional managed runtime (such as a JVM) to implement and use GC. The MRE-GC interface in GaS also differs from GCTk/MMTk in terms of granularity and encapsulation. By taking an MRE-neutral approach, GaS can afford fine-grain MRE-GC library interaction. In contrast, GCTk/MMTk in non-Java-based MREs must either use coarse-grain MRE-GC library interaction or break library encapsulation (because of the high cost of cross-language calls). Since MRE-GC interaction in inherently fine-grain (allocation/scanning/write barriers are frequent), to achieve good performance, non-Java-based MREs must replicate the GCTk/MMTk GC implementation in the MRE. GaS supports efficient direct fine-grain calls between GC and a MRE while maintaining the library encapsulation.

GaS is also simpler and more lightweight than GCTk/MMTk (where the approach is to support as many different GCs as possible, including object-moving GCs). Unlike GCTk/MMTk, GaS focuses on concurrent, on-the-fly GC and takes into account all restrictions placed on GC by different MREs (e.g. non-moving GC in cPython). GaS uses a GC algorithm designed specifically for a portable loosely-coupled GC library. The approach in GCTk/MMTk is to design the interface so that it supports diverse extant GCs.

Another way of reusing a GC implementation between two MREs is to implement an MRE in a high-level language, e.g. Jython, JRuby are Python/Ruby interpreters that run on top of a JVM and use JVM GC. The two key issues with such MRE layering is performance overhead (we investigate this empirically in Section 6.5), and incomplete/incompatible standard libraries (due to the extensive engineering effort required to make layering work).

Another system, called CoLoRS [32], provides cross-language, type-safe object sharing using POSIX shared memory for MREs that execute on the same physical hardware at the same time and interoperate. CoLoRS uses concurrent, on-the-fly GC for the shared memory region that each MRE maps into its address space. The CoLoRS GC however is tightly integrated into its runtime, and defines a new object and synchronization model for shared objects that it manages. GaS adds per-object headers and relies on MRE-native object model and synchronization.

VMKit [20] is a framework that eases the development of high-level MREs and thus enables experimentation with new languages and MREs and/or new language features. VMKit consists of a low-level and a high-level layer. The low-level layer provides threading support, GC-based memory management, and a JIT compiler that translates language-independent intermediate representation of programs. The high-level layer defines such aspects as object model, type system, call semantics, and method dispatch. VMKit glues together LLVM for JIT support, MMTk for GC, and POSIX thread library for multi-threading. VMKit translates MMTk into the LLVM intermediate representation in its entirety. VMKit performance, however, is orders of magnitude worse than production systems. GaS is orthogonal to VMKit in that GaS can be used as a GC component in the VMKit framework. Note, however, that GaS can be integrated not only with MRE frameworks, but also with general- and special-purpose MREs for both dynamic and static languages.

XIR [28] is a compiler-MRE interface that separates the compiler backend from an MRE. An XIR extension mechanism allows an MRE to express the machine-level implementation of object operations. The interface has a modest impact on compilation time without reducing performance. GaS is similar to XIR in its overall goal however GaS targets GC and XIR targets JIT compilation.

The idea of modularizing an MRE motivates the design and implementation of LadyVM [19]. LadyVM links three third-party software components: LLVM, Boehm GC, and GNU Classpath, to implement a Java VM. Similarly to VMKit, LadyVM can use GaS as a replacement for its GC component to enable modern, high-quality, concurrent GC.

Compiler libraries like LLVM [23] and VPU [25] enable modular approach to integrating JITs into VMs. LLVM is a compiler infrastructure designed for compile-time, link-time, and run-time optimization of programs written in arbitrary programming languages. LLVM supports a language-independent instruction set and type system. VPU is a high-level code generation utility that performs most of the complex tasks related to code generation, including register allocation, and which produces good-quality C ABI-compliant native code.

JnJVM [27] is a modular JVM that supports dynamic addition or replacement of its own modules without service interruption and state loss. JnJVM uses dynamic aspect weaving techniques and component architecture. GaS could potentially be used in JnJVM as a GC module.

The Common Language Infrastructure (CLI) [24] is an open specification (ECMA335) that describes the executable code format and runtime environment for multiple, static, high-level languages to be used on different computer platforms. All CLI-compatible languages compile to the Common Intermediate Language (CIL), which abstracts away the platform hardware. CLI is similar to GaS in that it provides GC (among other services) for multiple languages but it differs in that CLI uses monolithic architecture with built-in GC. GaS provides GC in a form of a library and targets multi-language support via the provision of a cross-MRE GC.

GNU Classpath [1] is a GNU project to create free core class libraries for use with virtual machines and compilers for Java. The Classpath library can be used with different VMs – it has a similar goal to GaS but pertains to core classes (not GC) and targets Java (not multiple MREs).

XMem [30], Singularity [18], MVM [11], and KaffeOS [3] provide isolation and sharing between MREs or tasks/processes and implement a common memory management system across them. GaS GC differs from GCs in these systems in that it is modular, loosely-coupled, and portable across different MREs and languages.

6. EXPERIMENTAL EVALUATION

Our primary goal with this evaluation is to show that a cross-language, cross-runtime GC that is implemented as a C library, offers competitive performance (in terms of application execution time, GC pause times, and other GC metrics) compared to tightly-integrated VM-specific collectors in production-quality VMs. We find that GaS significantly reduces pause times and introduces modest overhead on overall execution time. In this section, we also investigate the tradeoffs associated (i) with the way GC is integrated into a runtime systems (built-in vs. a native/non-native library) and (ii) with different GC designs (generational vs. non-generational, moving vs. non-moving, concurrent vs. stop-the-world).

We first compare GaS to state-of-the-art GCs in the C-based runtimes for Python and Java. We use cPython (`http://docs.python.org/py3k/`) and the HotSpot JVM (`http://openjdk.java.net`). cPython implements a single-threaded Reference Counting [10] with generational stop-the-world Cycle Detection (RC/CD) [4]. The HotSpot JVM implements two concurrent, parallel, and generational GCs: Garbage-First (G1) [13] and Concurrent Mark Sweep (CMS) [26]. Table 6 summarizes the main characteristics of these GCs compared to GaS.

GC	G1	CMS	RC/CD	GaS
concurrent	yes	yes		yes
on-the-fly				yes
parallel	yes	yes		yes
moving	yes	yes		
tracing	yes	yes	yes	yes
reference counting			yes	
generational	yes	yes	yes	

Table 1: High-level comparison of evaluated GCs.

RC/CD divides the heap into three generations. Once the number of objects with non-zero reference counts in the youngest generation reaches a specific threshold, RC/CD traces the object graph to find and free possible reference cycles within this generation. Survivors are promoted to the older generation. Generation $i+1$ gets collected after the specified number of collections of generation i. RC/CD does not move objects and segregates object into generations logically (i.e. it maintains a list of objects in each generation).

CMS [26] is a mostly-concurrent incremental GC based on the mostly-parallel collection algorithm described by Boehm et al [7]. HotSpot JVM implements CMS in the old generation and overloads generational write-barriers to identify objects that are modified during concurrent marking (these objects must be rescanned to ensure that the concurrent marking phase marks all live objects). CMS imposes two pauses per GC cycle: for initial marking and for remarking. CMS does not move/compact objects except for promotion to the old generation and copying within the young generation.

G1 [13] is a concurrent GC designed to meet a soft real-time goal with high probability, while achieving high throughput. G1 performs marking concurrently but halts mutators during object evacuation. Marking identifies regions that contain few live objects and that can be evacuated within a given pause time limit (with high probability). Each region has an associated remembered set, which indicates all locations that might contain pointers to (live) objects within the region. At carefully scheduled points, G1 stops the mutator threads and performs an evacuation pause. G1 is generational – regions holding current TLABs are treated as young and always belong to the evacuation set. G1 opportunistically moves objects to gradually defragment the heap.

6.1 Methodology

For our experiments, we use a dedicated machine with a quad-core Intel Xeon and 8GB main memory. Each core is clocked at 2.66GHz and has 6MB cache. Our platform runs 64-bit Ubuntu Linux 8.04 (Hardy) with the 2.6.24 SMP kernel.

We use HotSpot JVM from OpenJDK 6 build 19 (released April 2010) compiled with GCC 4.2.4 in the 64-bit mode. Our configuration employs the server (C2) compiler, biased locking, and two concurrent GCs: G1 (garbage-first) and CMS (concurrent mark-sweep) in a generational heap. In case of CMS, the young generation uses a parallel copying GC [2].

For the Java experiments, we employ the DaCapo'08 [12] and SPECjbb'00 benchmarks. We use the default input for DaCapo and 1 warehouse with 75s runs for SPECjbb. We disable explicit GC invocation. For the Python experiments, we use the open-source cPython 3.1.1 (released August 2009) compiled with GCC 4.2.4 in the 64-bit mode. Our Python benchmarks include PyBench (a collection of tests that provides a standardized way to measure the performance of Python implementations), a set of Shootout cPython benchmarks (`http://shootout.alioth.debian.org/`), and PyStone (a standard synthetic Python benchmark). Since there are no standard memory-intensive benchmarks for Python, we implement our own GC benchmark, called BST, which we model after SPECjbb. BST executes a number of iterations against a balanced binary search tree. Each iteration comprises 3 lookups, 1 insert, and 1 delete. This emulates realistic workloads by simulating an in-memory database.

We investigate the sensitivity of GaS to different parameter values across benchmarks. For the Java GCs, we vary four GC parameters: TLAB size, young generation size, number of GC threads, and GC-start threshold. We use the recommended values of this parameters (as described in the HotSpot documentation) for our detailed per-benchmark evaluation. For the Python RC/CD GC we vary one parameter: the GC-start threshold which controls the frequency of cycle detection in the young generation. RC/CD has no other parameters that significantly affect GC.

We evaluate the Java and Python GCs using four main metrics: throughput (execution time), GC pause times (average and maximum), minimum mutator utilization (MMU), and minimum required heap size. We do so across a range of heap sizes starting at the minimum heap size to at least its double. Note that concurrent GC requires more heap space than stop-the-world GC due to delayed garbage reclamation and allocations happening during collection. In cPython RC/CD, there is no reliable standard way of setting the heap size, therefore we do not vary the heap size in this case. We repeat each measurement a minimum of 5 times and report standard deviation as appropriate (error bars in plots).

6.2 Java Benchmarks

Table 2 details per-benchmark, GC metrics for GaS, G1, and CMS. These experiments use our baseline GC parameters. The TLAB size is 4kB, we use 2 GC threads, the GC-start threshold is 50% (i.e. collection starts once half of the heap is filled), and the young generation size is fixed at 8MB (the HotSpot documentation recommends the young generation size to be set to 4MB times the number of GC threads).

We next evaluate the impact of each GC parameter on the different GC metrics. When measuring pause times and execution time/throughput we use the minimum heap size that each benchmark requires to run under GaS, G1, and CMS.

Pause Times and MMU. In Table 2 we report both average and maximum pauses (in milliseconds for GaS, and as number of times decrease relative to G1 and CMS). Across benchmarks, average pause times in GaS are shorter by 12x compared to G1 and 6x compared to CMS. Maximum pause times in GaS are shorter by 8x compared to G1 and by 7x compared to CMS (across benchmarks).

Figure 5 shows the minimum mutator utilization (MMU) plots for the benchmarks and GCs. MMU curves [9] lend insight into the distribution of GC pauses across program execution. Mutator utilization for a given time window w is defined as the fraction of the window w during which the mutator executes (as opposed to GC). Minimum mutator utilization for time period p is the lowest mutator utilization across all time windows of size p during program execution. Thus, the x-intercept of a MMU curve is the maximum pause time and the asymptotic y-value corresponds to application throughput.

We do not include GC write barriers when computing MMU – we only take GC pauses into account. GaS achieves better utilization than G1 and CMS for all benchmarks.

Throughput. In the last three Columns in Table 2 we show per-benchmark execution time/throughput for GaS and percentage overhead of GaS relative to G1 and CMS. Across the DaCapo benchmarks GaS imposes 9.7% overhead compared to G1 and 11.6% overhead compared to CMS. For JBB, throughput reduction due to GaS is 5.7% relative to G1 and 6.3% relative to CMS. GaS overhead is mostly caused by GC write barriers and is overestimated here because our implementation of the write barriers is not as optimized as it could be.

Figure 6 shows per-benchmark execution time as a function of heap size. Each plot starts at the minimum heap size. CMS and G1 have similar performance for our benchmarks. We do not observe significant execution time increase for minimum heap sizes typical of stop-the-world GC. This is because GCs run on separate cores and only slow the program down for short pauses during which little processing takes place.

Heap Size. In Columns 2–4 we report minimum required heap size for each benchmark (for GaS in MB and for G1 and CMS as number of times decrease relative to GaS). GaS requires larger minimum heap sizes than G1 (by 3x on average) and CMS (by 5x on average) because of three reasons. First, G1 and CMS are generational and thus tolerate allocation bursts better and place less pressure on the concurrent GC which executes for the old generation only. Second, GaS does not move objects and thus suffers from fragmentation (CMS uses a copying GC in the young generation and G1 performs opportunistic block-based compaction). Third, GaS adds a per-object header word, which may matter in benchmarks that allocate small objects. Each of these reasons is a consequence of a primary GaS design goal to be portable across runtimes and languages with different memory management subsystems.

Note that in case of concurrent GC, heap overprovisioning

Bench-mark	Minimum Heap			Average Pause			Maximum Pause			Execution Time		
	GaS [MB]	vs.G1 [x incr.]	vs.CMS [x incr.]	GaS [ms]	vs.G1 [x decr.]	vs.CMS [x decr.]	GaS [ms]	vs.G1 [x decr.]	vs.CMS [x decr.]	GaS [s]	vs.G1 [% incr.]	vs.CMS [% incr.]
antlr	40	2.0	4.0	0.8	2.5	1.5	2.5	2.1	1.5	10.9	6.0	9.9
bloat	140	4.7	14.0	0.5	3.8	1.4	2.0	2.4	2.6	25.5	11.5	17.9
chart	100	1.0	3.3	0.3	12.0	6.2	2.0	4.8	3.0	25.7	8.9	9.2
eclipse	400	8.0	5.7	0.6	7.3	3.4	3.4	4.4	4.6	71.9	11.4	15.5
hsqldb	290	1.9	1.9	0.5	36.3	20.6	1.4	17.2	22.9	12.7	-1.6	-1.5
jython	80	4.0	2.7	0.8	2.3	0.8	3.9	1.2	1.5	29.9	4.8	7.8
luindex	120	2.4	12.0	0.3	7.5	3.7	1.5	4.2	3.1	25.6	4.8	4.4
pmd	250	3.6	4.2	0.3	27.2	15.0	1.2	31.9	18.2	20.3	12.1	16.7
xalan	80	1.3	0.4	0.8	4.7	4.2	4.1	1.8	2.8	23.6	29.0	24.6
average	167	3.2	5.4	0.5	11.5	6.3	2.4	7.8	6.7	27.3	9.7	11.6
										Throughput		
										[kbops]	[% decr.]	[% decr.]
jbb	110	1.8	2.2	0.5	12.9	4.3	1.9	7.2	6.3	3.9	5.7	6.3

Table 2: Per-benchmark comparison of Java GCs: G1, CMS, and GaS. Columns 2–4 show the minimum required heap size: for GaS in MB and number of times increase relative to G1 and CMS. In Columns 5–7 and 8–10 we report average and maximum pause times: for GaS in ms and number of times decrease relative to G1 and CMS. The last three Columns show execution time in seconds (for the DaCapo benchmarks) and throughput in kilo-operations per second (for JBB). We report percentage execution time increase and throughput decrease relative to G1 and CMS.

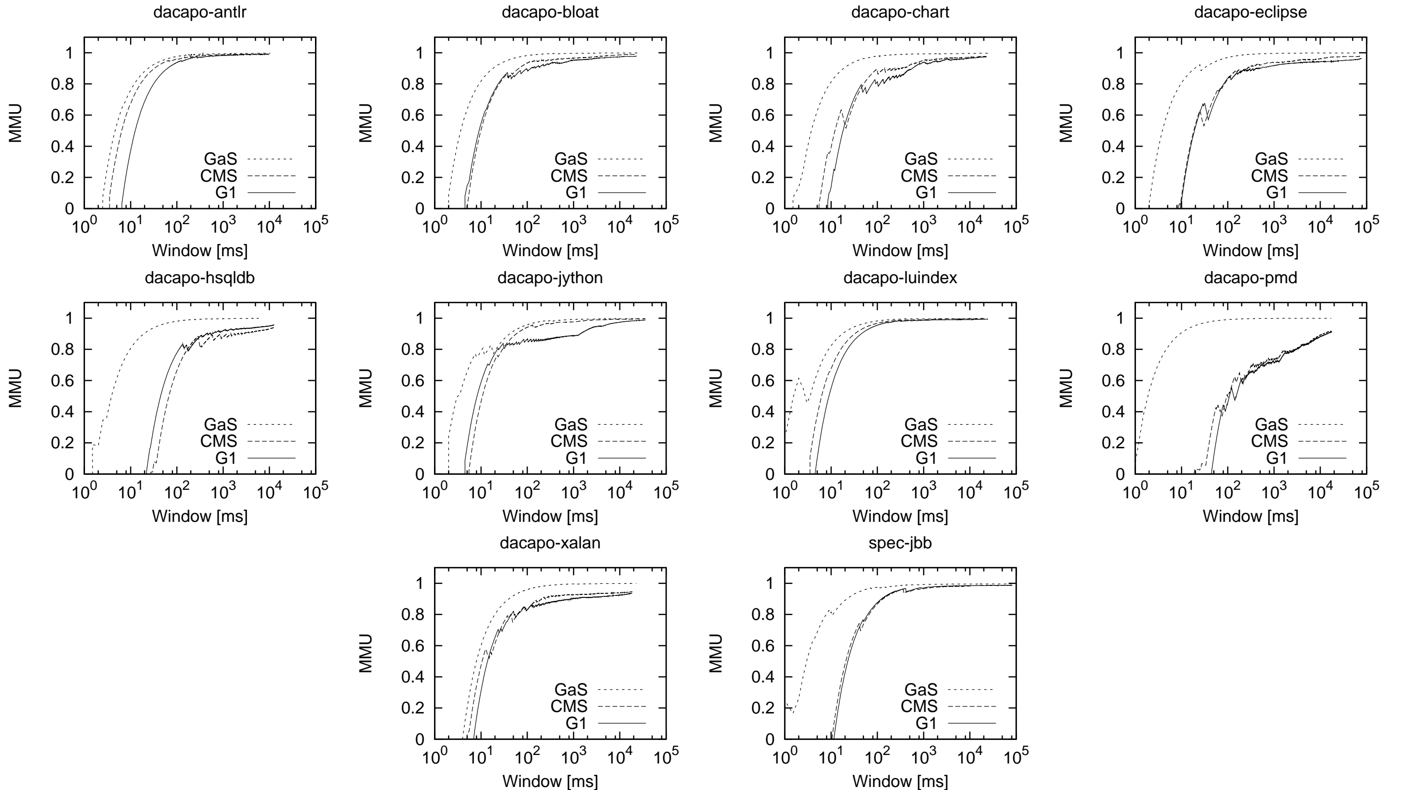

Figure 5: Minimum mutator utilization (MMU) for the DaCapo benchmarks and JBB. We compare GaS with two HotSpot GCs: G1 and CMS. In all the plots, the x-axis (logarithmic scale) is a MMU window size (in ms). The x-intercept is the maximum pause time.

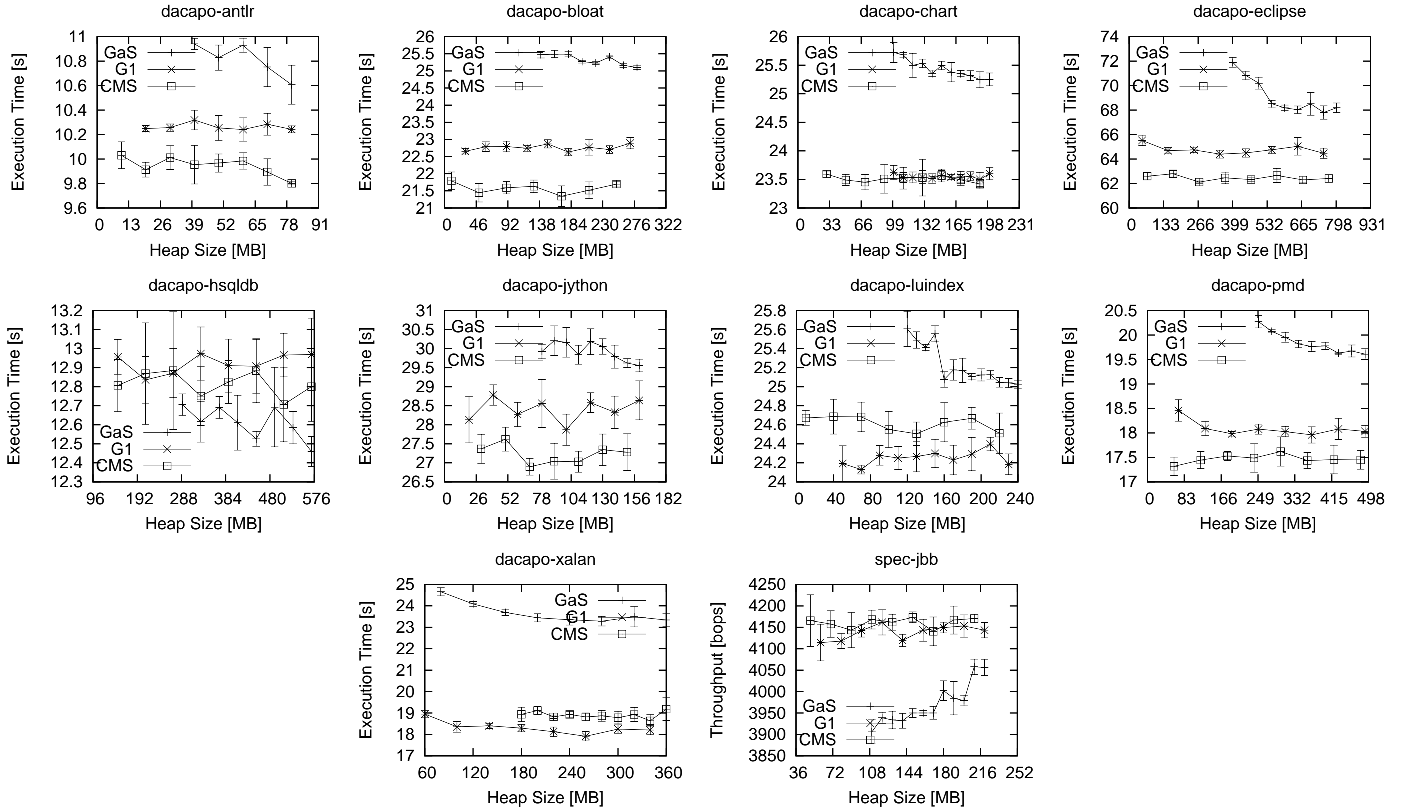

Figure 6: Execution time (for the DaCapo benchmarks) and throughput (for JBB'00) as a function of heap size. We compare GaS with two HotSpot GCs: G1 and CMS. Each plot starts at the minimum heap size.

GC Metric	Baseline Parameters	TLAB [kB]		GC Threshold [%]		GC Threads		Young Gen. [MB]	
		1	16	20	80	1	3	2	32
Relative to HotSpot G1									
Throughput [% decr.]	8.40	7.54	7.86	10.29	6.95	5.24	8.15	3.50	12.25
Avg. Pause [x decr.]	10.12	9.76	10.43	12.77	8.73	14.77	8.29	4.32	31.34
Max. Pause [x decr.]	8.26	7.58	8.34	7.58	9.33	10.76	7.85	5.25	20.89
Min. Heap [x incr.]	3.07	2.30	4.69	3.07	3.86	3.11	3.10	4.63	1.81
Relative to HotSpot CMS									
Throughput [% decr.]	9.97	9.60	9.59	11.19	9.50	7.80	9.85	2.99	13.50
Avg. Pause [x decr.]	5.52	5.19	5.57	6.48	5.81	6.69	4.44	2.10	13.45
Max. Pause [x decr.]	7.67	6.50	6.94	6.53	8.29	8.59	8.05	3.80	21.68
Min. Heap [x incr.]	5.05	3.33	8.11	5.10	4.60	5.23	5.27	6.26	4.64

Table 3: Impact of the GC parameters on the GC metrics in Java. We report all metrics (throughput, avg. and max. pauses, and min. heap) for G1 (first part, Rows 3–6) and CMS (second part, Rows 8–11) relative to GaS. The second Column contains results for the baseline parameters: 4kB TLAB, 2 GC threads, 50% GC-start threshold, and 8MB young generation. Each of the subsequent Columns shows the impact of one GC parameter while the other 3 are kept at the baseline.

GC Metric	GaS Baseline	GC-Start Threshold [number of young unreclaimed objects]					
		70		700		7000	
		RC/CD	Relative	RC/CD	Relative	RC/CD	Relative
Min. Heap [MB]	17	7	2.4 [x incr.]	10	1.7 [x incr.]	34	0.5 [x incr.]
Avg. Pause [ms]	0	0.2	–	0.8	–	5.4	–
Max. Pause [ms]	0	56.6	–	100.3	–	302.9	–
Time [ms/10^3iters]	39.3	38.0	3.6 [% incr.]	36.3	8.4 [% incr.]	35.3	11.4 [% incr.]

Table 4: GC metrics for GaS and Python RC/CD for different values of the young generation threshold in RC/CD (70, 700, and 7000). We report minimum heap in MB, average and maximum pauses, and BST throughput (time per 1000 iterations). Column 2 shows the results for GaS in its baseline configuration. The following Columns compare GaS and RC/CD for different thresholds.

does not impact performance significantly (unlike in case of stop-the-world GC [31, 29]). That is, across all the benchmarks, giving G1 and CMS much more heap does not improve their performance.

Sensitivity to GC Parameters. To evaluate the parameter sensitivity of GaS, we vary the TLAB size between 1kB and 16kB, the young generation size between 2MB and 32MB, the number of GC threads between 1 and 3 (note that we have only 4 cores and we need to leave one core for the actual program), and the GC-start threshold between 20% and 80%. Our baseline values of GC parameters (reported previously) are medians of these ranges.

In Table 3, we present how our GC metrics (throughput, average and maximum pause times, and minimum heap) depend on GC parameters. The Table consists of two parts. The first part (Rows 3–6) reports the GC metrics for GaS relative to G1 and the second part (Rows 8–11) relative to CMS. We vary one GC parameter at a time and keep the remaining 3 parameters at their baseline values. Column 2 corresponds to the baseline values of all 4 parameters. Each of the following Columns (3–10) reports the impact of one parameter: TLAB size, GC-start threshold, number of GC threads, and young generation size. We report GC metrics for two extreme values of each GC parameter. For each benchmark we use the minimum heap size in which all experiments for the benchmark run. We report average results across benchmarks (DaCapo and JBB).

The young generation size has the greatest impact on all GC metrics. For small sizes (2MB), GaS degrades throughput 3-4% relative to G1 and CMS. For large sizes (32MB) throughput degradation is 12-14%. GaS converges to G1/CMS performance as G1/CMS approach non-generational GC.

G1/CMS pause times increase significantly for larger young generation sizes (up to 22-31 times longer than for GaS). For small sizes, G1/CMS pause times are 2-5 times worse than for GaS. This is because G1 and CMS are generational hybrids of concurrent and stop-the-world GC and trade throughput for pause times. Note that GaS does not have this tradeoff. Finally, large young generation sizes increase the minimum heap size in G1/CMS because during minor GCs more objects get promoted and, as a result, there is more pressure on the concurrent GC.

Dedicating fewer threads to GC in all collectors prolongs pause times and decreases throughput. TLAB size impacts only minimum heap size – large TLABs require that GaS uses more heap than G1 and CMS. This is because allocation rate is higher with large TLABs. GC-start threshold has only a minor impact on the GC metrics.

6.3 Python Benchmarks

To evaluate cPython hybrid GC, our BST benchmark creates both cyclic (collected by tracing) and acyclic (collected by reference counting) garbage. To create cycles we use self-referencing objects. We investigate 3 configurations: all-cyclic, all-acyclic, and 50% cyclic. Our main evaluation uses the last one. We have evaluated the all-cyclic and all-acylic configurations relative to the 50% cyclic one using our GC metrics. We have found that the all-cyclic configuration has shorter pauses (by 21-22%), larger minimum heap size (by 15%), and 3% worse execution time. The all-acyclic configuration has shorter pauses (by 49-56%), smaller minimum heap size (by 41%), and better execution time (by 5%). When RC/CD relies only on tracing, it imposes more overhead, uses 2x more heap, and has up to 2x longer pauses than when it uses only reference counting.

We allocate 15-level trees in BST. The live data set size does not impact RC/CD in Python because the cost of tracing in this GC depends mostly on the number of objects that are reachable from potential cycles (it does not matter if they are live or dead). The cost of tracing in GaS is proportional to the size of live data.

In Table 4 we report the GC metrics for GaS and RC/CD. Column 2 shows the results for GaS that correspond to our baseline GC parameters (2 GC threads, 50% GC-start threshold, and 4kB TLABs). We report the minimum heap (we instrument cPython to measure it), pause times, and execution time per BST iteration.

Columns 3–8 compare GaS with RC/CD for 3 different values of the main RC/CD parameter (the GC-start threshold). For its default value (700) GaS requires 1.7x more heap and has 8% lower throughput relative to RC/CD. However, GaS imposes no pauses, while RC/CD does (up to 100ms, and 0.8ms on average).

Setting the GC-start threshold to 70 results in more frequent GCs in RC/CD. This results in shorter pauses (0.2ms on average and 57ms maximum), worse throughput (only 4% better than GaS), and lower minimum heap. Similar space/time tradeoffs can be observed when the young generation threshold is 7000. Now CD GC is relatively rare but each cycle is expensive. Pause times increase (5.4 ms on average and 303ms maximum), throughput improves (11% better than GaS), and the minimum heap increases (exceed-

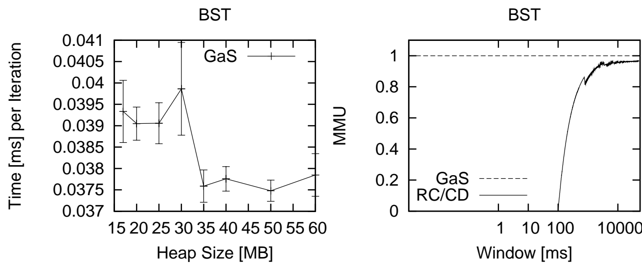

Figure 7: Results for Python and the BST benchmark. The left plot shows execution time per BST iteration as a function of heap size for GaS (in RC/CD heap size is not a GC parameter). The right plot is MMU for GaS and RC/CD. Note that since GaS does not impose pauses its MMU is at 1.0 across the window sizes.

ing 2x GaS).

The left plot in Figure 7 shows how sensitive per-iteration execution time in BST is on heap size in GaS. BST throughput is 3% better for heap sizes that are 2 times the minimum. Heap size is not a GC parameter for RC/CD.

The right plot in Figure 7 shows MMU for GaS and RC/CD. Since GaS in cPython has no pauses, its MMU equals 1.0 for all window sizes. In RC/CD, the maximum pause time is 100ms (we use the default 700 GC-start threshold). RC/CD approaches GaS utilization for window sizes above 1 second. The MMU plots do not take write barriers/conditional RC into account (only pause times). In RC/CD we only measure pause times caused by tracing. Reference counting imposes negligible pauses in BST because whenever we delete nodes we free one node at a time.

Table 5 shows execution time statistics for the Python benchmarks. These benchmarks are not memory intensive and do not exercise GaS GC like the BST benchmark does. On average, the overhead of GaS extensions is 3%.

6.4 Overhead of Cross-Runtime Calls

We next investigate the performance overhead of other ways of integrating a GC into an MRE. We first consider the approach that implements the GC in Java (e.g. MMTk and GCTk) that is then integrated into a C-based runtime. We evaluate the cost of crossing the runtime boundaries. We measure the overhead incurred by the up/down calls through the Java Native Interface (JNI) – the mechanism through which Java and C programs interact. We consider the key GC operations: object allocation and object scan.

We implement object allocation as a Java method `Object allocate(int size)` which takes object size as input and returns the allocated object. We upcall this method from C via JNI. Object scan is represented as a native method `int scan(Object o, Object[] b)` whose arguments are a reference to an object to scan and a reference to a buffer for pointers found in the scanned object. The method returns the number of references found. We downcall this native method from Java using JNI. Since our goal is to measure the JNI overhead, the *allocate* and *scan* methods do not perform any processing: *allocate* returns NULL and *scan* returns 4 pointers. We duplicate both methods in C and call them directly from C (without gcc inlining) to compare direct calls with JNI calls.

Benchmark	Execution Time [s]	% GaS Overhead
pybench	3.91	2.05
pystone	4.12	3.40
binary-trees	6.71	3.13
fannkuch	1.95	8.72
mandelbrot	15.48	2.13
meteor-contest	2.26	4.42
n-body	8.44	-0.59
spectral-norm	14.28	3.01
average	7.14	3.28

Table 5: Execution time overhead in GaS for standard Python benchmarks relative to RC/CD.

We run 10 experiments, each consisting of 10^6 calls. On average, when compared to direct (but not inlined) C calls, JNI upcalls for *allocate* are 76x slower and downcalls for *scan* are 5x slower. Downcalls are faster than upcalls because the HotSpot JIT compiler optimizes native calls extensively. For 10^6 calls, upcalls for *allocate* introduce 225ms of overhead and downcalls for *scan* incur 92ms of overhead.

The DaCapo benchmarks allocate between 2.4 and 161 million objects (with the mean of 18 million) whereas the number of live objects during a GC cycle reaches between 2.8 thousand and 3.2 million (with the mean of 104 thousand). Thus, the JNI overhead for allocation can range from 0.54s to 36s of execution time. Similarly, the JNI overhead for scanning (assuming 25 collections per program execution) can range from 64ms to 7.4s of execution time.

Such overhead is likely unacceptable for C-based runtimes which typically are tuned for high performance. MRE-neutral, C-based GC library is both easier to integrate into such runtimes and offers significantly better performance.

6.5 Overhead of Runtime Layering

We next consider another alternative approach to using a single GC for multiple programming languages: runtime layering. In this study, we investigate the cost of using a production-quality Java runtime to host a non-Java language. In particular, we compare the performance of Python benchmarks for Jython 2.5.1 (a Python runtime that executes on top of a JVM – the HotSpot JVM in our case), versus using cPython v2.6.

We omit the raw data due to space constraints, and summarize our findings here. For pybench and pystone, Jython is 2.5x and 1.74x slower than cPython. For the shootout benchmarks (those which Jython supports without extensive benchmark modifications) Jython is 2.97x (meteor-contest), 1.34x (spectral-norm), 2.24x (fannkuch), 1.72x (binary-trees), and 2.22x (n-body) slower. On average, cPython is 2.1x faster than Jython.

Re-using a Java runtime (and Java GC) to implement runtimes for other languages introduces significant overhead (in addition to being complex and time-consuming from the engineering standpoint). An alternative, simpler, and more efficient approach to incorporating a modern GC and memory management subsystem into a new or extant C-based runtime is to use a GC library like GaS.

6.6 Lines of Code

We next compare GaS, HotSpot G1/CMS, and Python RC/CD using lines-of-code, to lend insight into the approximate implementation effort required for each GC. The GaS library is around 1100 lines of C/C++. The integration/glue code in both Python and HotSpot is around 200 lines.

The implementation of G1 and CMS in HotSpot is around 30,000 and 22,000 lines of C/C++. RC/CD in cPython is 8,400 lines of C (note that reference counting code is scattered across the whole runtime). This suggests that GaS GC library is simpler to implement than G1, CMS, and RC/CD. In addition, 200 lines of the GaS integration code is 2 orders of magnitude fewer than that which is required to implement a modern GC from scratch in an MRE.

6.7 Summary

We have compared GaS with two generational, concurrent GCs for Java and a hybrid tracing/reference-counting GC for Python. GaS significantly improves pause times and MMU across all benchmarks and GCs. GaS requires larger heap sizes and imposes modest execution time overhead because it is non-generational and non-moving (unlike G1 and CMS) and concurrent (unlike RC/CD). GaS is non-moving so that it is able to support runtimes (such as Python) that make assumptions about object addresses.

We also investigate the performance sensitivity to different GC parameters on the GC metrics. We find that GaS minimum heap sizes and throughput converge to G1/CMS and RC/CD once the GC parameters mitigate the generational advantage of these GCs. We measure the overheads associated with other approaches to implementing a GC in an MRE (via cross-language calls and via runtime layering) and find that using a GC library in C-based runtimes is significantly simpler and more efficient.

7. CONCLUSIONS

We contribute GaS, a lightweight, cross-MRE, cross-language GC library that provides concurrent, on-the-fly, non-moving GC. GaS can be integrated into MREs for static (e.g. Java) and dynamic (e.g. Python) languages via a fine-grain, low-overhead GC interface. GaS is a stand-alone C-based library for GC-cooperative MREs. GaS GC adapts the SATB algorithm for loose coupling between GC and an MRE. The GaS library makes no assumptions about object model, threading, JIT, and memory management strategy (tracing, reference counting, generations, etc.) in an MRE. We implement GaS and integrate it within production-quality MREs for Java and Python. Our experimental evaluation shows that in comparison to built-in, tightly-coupled GCs, GaS can improve pause times significantly and offers competitive performance even when compared to generational GCs. The GaS library reduces the development effort required for implementing a state-of-the-art on-the-fly GC. The library can be used as a modern GC component both in extant MREs and when building new MREs for new or existing languages.

8. REFERENCES

[1] GNU Classpath. `http://www.gnu.org/software/classpath`.

[2] HotSpot Virtual Machine Garbage Collection. `http://java.sun.com/javase/technologies/hotspot/gc`.

[3] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *OSDI*, 2000.

[4] D. F. Bacon and V. Rajan. Concurrent cycle collection in reference counted systems. In *ECOOP*, 2001.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *ICSE*, 2004.

[6] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *PLDI*, 2002.

[7] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6), 1991.

[8] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *SP&E*, 18(9), 1988.

[9] P. Cheng and G. Blelloch. A parallel, real-time garbage collector, 2001.

[10] T. W. Christopher. Reference count garbage collection. *SP&E*, 14(6), 1984.

[11] G. Czajkowski and L. Daynes. Multitasking without compromise: A virtual machine evolution. In *OOPSLA*, 2001.

[12] The DaCapo Benchmark Suite. `http://dacapobench.org`.

[13] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM*, 2004.

[14] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL*, 1994.

[15] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL*, 1993.

[16] T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanorer. Implementing an on-the-fly garbage collector for Java. *SIGPLAN Not.*, 36(1), 2001.

[17] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for Java. *SIGPLAN Not.*, 35(5), 2000.

[18] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, 2006.

[19] N. Geoffray, G. Thomas, C. Clément, and B. Folliot. A lazy developer approach: Building a JVM with third party software. In *PPPJ*, 2008.

[20] N. Geoffray, G. Thomas, J.Lawall, G. Muller, and B. Folliot. VMKit: a Substrate for Managed Runtime Environments. In *VEE*, 2010.

[21] R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. Technical Report COINS 91-47, 1991.

[22] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[23] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[24] E. Meijer, R. Wa, and J. Gough. Technical overview of the Common Language Runtime, 2000.

[25] I. Piumarta. The virtual processor: fast, architecture-neutral dynamic code generation. In *VMRTS*, 2004.

[26] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. *SIGPLAN Not.*, 36(1), 2001.

[27] G. Thomas, N. Geoffray, C. Clément, and B. Folliot. Designing highly flexible virtual machines: the JnJVM experience. *Softw. Pract. Exper.*, 38(15), 2008.

[28] B. Titzer, T. Wurthinger, D. Simon, and M. Cintra. Improving Compiler-Runtime Separation with XIR. In *VEE*, 2010.

[29] M. Wegiel and C. Krintz. The Mapping Collector: Virtual memory support for generational, parallel, and concurrent compaction. In *ASPLOS*, 2008.

[30] M. Wegiel and C. Krintz. XMem: Type-Safe, Transparent, Shared Memory for Cross-Runtime Communication and Coordination. In *PLDI*, 2008.

[31] M. Wegiel and C. Krintz. Dynamic prediction of collection yield for managed runtimes. In *ASPLOS*, 2009.

[32] M. Wegiel and C. Krintz. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. In *OOPSLA*, 2010.

[33] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. IWMM'95.