Evaluating Garbage Collection Performance Across Managed Language Runtimes

Yicheng Wang*†, Wensheng Dou*†‡§, Yu Liang*†, Yi Wang*†, Wei Wang*†‡§, Jun Wei*†‡§, Tao Huang*†
*Key Lab of System Software, State Key Laboratory of Computer Science, Institute of Software Chinese Academy of Sciences
†University of Chinese Academy of Sciences, Beijing
§Nanjing Institute of Software Technology, ‡University of Chinese Academy of Sciences, Nanjing
{wangyicheng19, wsdou, liangyu22, wangyi22, wangwei, wj, tao}@otcaix.iscas.ac.cn

***Abstract*—Modern managed language runtimes (e.g., Java, Go and C#) rely on garbage collection (GC) mechanisms to automatically allocate and reclaim in-memory objects. The efficiency of GC implementations can greatly impact the overall performance of runtime-based applications. To improve GC performance, the academic and industrial communities have proposed several approaches to evaluate the GC implementations in an individual runtime. However, these approaches target a specific managed language (e.g., Java), and cannot be used to compare the GC implementations in different runtimes.**

In this paper, we propose `GEAR`, an automated approach to construct consistent GC workloads for different managed language runtimes, which can further be used to evaluate GC implementations across different runtimes. Specifically, we design a group of runtime-agnostic Memory Operation Primitives (`MOP`), which can portray the memory usage information that influences GC. `GEAR` can further automatically convert a `MOP` program into runtime-specific programs for the target runtimes, which serve as a consistent GC workload for different runtimes. To build `MOP` programs with real-world GC workloads, we instrument the commonly-used runtime Java Virtual Machine (JVM) to collect the memory operation trace during a Java application's execution, and then transform the memory operation trace into a `MOP` program. The experimental result on three widely-used runtimes (i.e., Java, Go and C#) shows that `GEAR` can generate consistent GC workloads for different runtimes. We further conduct a comprehensive study on these three runtimes, and reveal some interesting findings about their GC performance, providing useful guidance for improving their GC implementations.

***Index Terms*—Garbage collection, managed language runtime, performance evaluation**

I. Introduction

Developers are increasingly turning to managed languages (e.g., Java, Go, C#, JavaScript and Python) due to their security and flexibility [1], [2]. Managed language runtimes usually adopt automatic memory management mechanisms, i.e., garbage collection (GC), to automatically allocate and reclaim in-memory objects. GC can greatly simplify coding, reduce memory-related errors and improve code reusability for developers [3], thus improving development efficiency.

Despite the advantages of GC, GC imposes non-negligible performance overhead on application execution [4]–[6]. GC activities (e.g., GC pauses) can potentially affect the overall throughput and latency of applications [7]–[10]. To reduce GC's performance overhead on applications, existing managed language runtimes have designed and implemented various GC mechanisms [11], [12]. For example, Java runtime HotSpot implements mostly-concurrent ZGC [13] to reduce GC pauses, and C# runtime CLR implements generational server GC [14] to improve application throughput.

To better analyze and optimize GC performance, the academic and industrial communities have proposed some approaches and benchmarks to evaluate GC implementations in individual managed language runtimes [15]–[19]. For example, the Dacapo benchmark [15] develops multiple Java applications with typical memory usage patterns, and has been widely used in evaluating the different GC implementations in Java runtimes [20]–[23].

Different managed language runtimes usually have different GC implementations. Thus, it is appealing to understand which GC implementation in different runtimes can achieve better performance [24]–[29]. By comparing the GC implementations in different runtimes, we can identify performance shortcomings in specific GC implementations, and provide useful guidance for GC optimization. For a fair comparison, reimplementing a GC mechanism in another runtime [30] is a possible solution, but would take huge development effort. We still lack an effective approach to scientifically evaluate GC implementations in different runtimes.

Recently, researchers have proposed some approaches to compare the overall performance [31]–[35], energy efficiency [36]–[38], and compiler performance [39]–[41] in different runtimes. They usually leverage the applications written in different languages with the same functions to perform the comparison. However, these applications with the same functions can still have varying memory usage in different runtimes. For example, the commonly-used library *Hashmap* has different memory usage in Java and Go due to different implementations. As a result, these approaches cannot ensure that the GC workload of evaluation applications remains consistent across different runtimes. To make things worse, these approaches [31], [42] build their experimental applications in different languages from scratch, which requires significant development effort and cannot be easily extended to other managed languages and applications.

To scientifically compare GC implementations across different runtimes, we introduce *GC Evaluation Across Runtimes* (`GEAR`) in this paper. `GEAR` is designed to create consistent GC

Wensheng Dou is the corresponding author.

workloads for different runtimes, which create objects with the same structure and reference relationships, and change object reference relationships in the same execution order in different runtimes. There are two main challenges in achieving this goal: (1) How to create programs that share a consistent GC workload for different runtimes? (2) How to efficiently create these programs to exhibit real-world GC workloads?

To address the first challenge, we first design a group of runtime-agnostic Memory Operation Primitives (MOP). MOP can portray the memory usage information that influences GC, including object structure, object creation, object reference update, and method call. We then establish the conversion relations between MOP and the target managed languages, and automatically convert a MOP program into runtime-specific programs for the target runtimes.

For the second challenge, we first instrument the commonly-used runtime Java Virtual Machine (JVM) to collect the memory operation trace during a Java application's execution. This trace includes class information, time-ordered object operation, along with the method and thread signature at which the trace is generated. We then automatically transform and simplify a memory operation trace into a MOP program, which preserves the same object operations and method call stacks as the original Java application.

To validate the effectiveness of GEAR, we select three popular managed language runtimes for our evaluation, i.e., Java, GO and C#. We manually craft three MOP programs and automatically build 20 MOP programs from real-world Java applications sampled in GitHub, Guava Benchmark [43] and Rosetta Code [44], and convert them into runtime-specific programs for the target managed language runtimes. For the three manually crafted MOP programs, the memory profiling results show that the objects created by our generated programs in different runtimes have consistent counts, size and reference relationships, with a maximum difference of 5.0%. This consistency demonstrates GEAR's capability to construct consistent GC workloads across different runtimes. Additionally, we compare the object statistics of the generated programs with their original Java applications for the 20 real-world applications. The experimental result shows a maximum difference of 4.8%, further demonstrating GEAR's capability to build real-world GC workloads.

We further perform an empirical study on the performance of the three representative GC implementations in these runtimes, i.e., Java ZGC [13], Go GC and C# server GC. These implementations are selected based on their popularity and diversity. We design 16 typical GC workloads for these GC implementations, including 3 typical software systems simulating different memory usage patterns (web server, database and big data framework), 4 basic data structures (stack, queue, tree and array), and 9 real-world GC workloads, which are generated by GEAR based on the applications sampled from Github and Guava Benchmark [43]. We collect GC metrics, execution metrics, and resource usage metrics through logging and profiling, and perform variable control analysis on data sizes, heap sizes, and program parallelism.

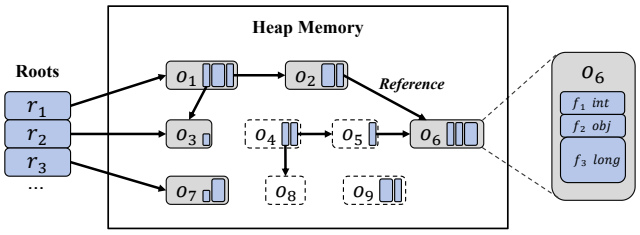

Fig. 1. The typical object model in a managed language runtime.

TABLE I
THE MAIN DESIGN CHOICES IN OUR STUDIED GCS

GC	Compact	Generational	Concurrency
Java ZGC	Moving	no	Mostly
Go GC	Non-Moving	implicit	Partially
C# GC	Moving	yes	None

During this study, we reveal several interesting findings, some of which cannot be inferred or induced by the design choices of corresponding GC implementations. We summarize some main findings as follows. (1) Java ZGC and Go GC perform significantly worse under the single-linked list structure compared with that under the array structure, while C# GC is less affected by different object reference relationships. This indicates that Java ZGC and Go GC have performance issues when dealing with the single-linked list structure. (2) Go GC performs relatively worse under programs with adequate memory budget, since its heuristic heap adjustment is too strict on memory usage. This indicates that Go GC should optimize its heap adjustment strategy. (3) Go GC scales better when increasing application parallelism compared to Java ZGC and C# GC. In design, Java ZGC and C# GC should have similar scalability as Go GC. However, our experiment shows that Java ZGC even suffers slowdowns when parallelism doubles in some cases. This indicates scalability issues in Java ZGC. These findings can provide valuable insights for GC and runtime developers to improve their GC implementations.

In summary, we make the following contributions.

- We propose a group of runtime-agnostic Memory Operation Primitives (MOP) that can portray the memory operations related to GC, and are used to create consistent GC workloads for different runtimes.
- We propose an approach to instrument the JVM to collect the memory operation trace during Java application execution, and automatically transform memory operation traces into MOP programs.
- We evaluate GEAR on manually written and automatically generated MOP programs, and the experimental result shows that GEAR can generate consistent GC workloads for different runtimes .
- We conduct an empirical study on 3 widely-used GC implementations by executing 16 consistent GC workloads, and reveal several interesting findings.

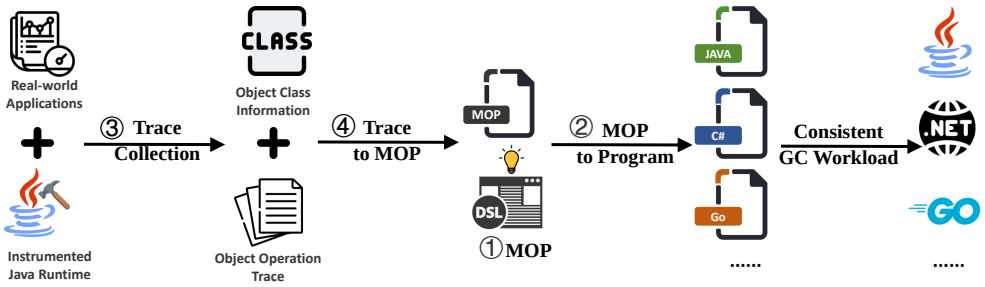

Fig. 2. GEAR overview.

II. Background

In the managed language runtimes, heap memory is a region where dynamic memory allocation of objects takes place. Objects allocated on the heap memory do not have fixed lifetimes and are managed by GC mechanisms.

A. Typical Object Model

Objects are essential components in managed language runtimes. They are instances of classes, with their size determined by the number, type and order[1] of fields defined by their classes. As shown in Fig. 1, an object field can store a primitive type value or a reference to another object. A reference is a pointer that enables one object to access another, forming a reference graph of interconnected objects in the heap. GC roots are references to objects that must live, including static fields and local variables in active methods [46].

B. Typical GC Workflow

Managed language runtimes leverage GC mechanisms to reclaim memory occupied by dead objects. Modern GC algorithms like mark-sweep and mark-compact reclaim dead objects in periodic GC cycles.[2] A typical GC cycle is usually triggered when the heap usage reaches a certain threshold. During a GC cycle, the GC mechanism typically marks live objects through reachability analysis, which starts from GC roots, and traverses the object reference graph through the chain of references. The objects identified during reachability analysis are marked live (e.g., o_1, o_2, o_3, o_6 and o_7 in Fig. 1), while others are considered dead (e.g., o_4, o_5, o_8 and o_9 in Fig. 1). The memory occupied by these dead objects can be reclaimed by the GC mechanism.

C. Design Choices in GC Implementations

Different managed language runtimes have different GC implementations, because they have different design choices and implement GC algorithms in different ways. Table I shows the main design choices adopted in our studied three GCs.

- **Memory Compact**. Non-moving GC like Go GC [11] will not relocate live objects after reclaiming dead objects, while moving GC will relocate live objects to compact memory and reduce memory fragmentation.
- **Generational Memory**. Generational GC like C# server GC [14] divides the heap memory into generations, and frequently performs minor GC to collect short-lived objects in the young generation at a small cost. Non-generational GC like Java ZGC [13] targets at objects in the whole heap in all GC cycles. Go GC manages young objects and old objects differently but in the same region.
- **GC Concurrency**. Non-Concurrent GC like C# server GC performs all its work in the stop-the-world pauses. Partially-Concurrent GC like Go GC performs some of their work concurrently with the application's execution. Mostly-Concurrent GC like Java ZGC leverages load barrier to minimize the pauses and performs most GC work concurrently with the application.

Besides the aforementioned design choices, there exist more differences between GC implementations in different managed runtimes, such as heap layout and object pointers. These mechanisms ultimately lead to differences in GC performance.

III. GC Evaluation Across Runtimes

In this paper, we propose *GC Evaluation Across Runtimes* (GEAR) to create consistent GC workloads for different managed language runtimes. Fig. 2 shows the overview of GEAR. We first design a group of runtime-agnostic Memory Operation Primitives to portray the memory usages that influence GC (① *MOP*). We then convert a MOP program into runtime-specific programs for the target runtimes, which serve as a consistent GC workload for different runtimes (② *MOP to Program*). To build MOP programs with real-world GC workloads, we instrument the Java Runtime to collect the memory operation trace during a Java application's execution (③ *Trace Collection*). Then, we transform the memory operation trace into a MOP program that preserves the GC workload of the original Java application (④ *Trace to MOP*). The generated programs can serve as a consistent GC workload for different runtimes.

A. Memory Operation Primitives

We design Memory Operation Primitives (MOP) to portray the memory usage information that influences GC, which can serve as the foundation for the generation of runtime-specific programs. To achieve this goal, we follow two principles in the design of MOP. (1) MOP only contains memory operations that can influence GC workloads, computational operations (e.g., basic arithmetic and comparison operations) are not involved,

[1]The order of fields in an object can affect memory alignment [45].

[2]Reference counting GC algorithm may reclaim an object when its reference count reaches zero.

$PrimitiveType$:=bool | char | byte | int | float | double
$ClassDefinition$:=class *ClassType* extends *ClassType*{$FieldDefinition*$}
$FieldDefinition$:=static $Type$ *FieldName* | $Type$ *FieldName*
$Type$ ::=$PrimitiveType$ | *ClassType*
$ObjectCreation$:=NewInstance(*ClassType*)
| NewArray(*ClassType*, $IntLiteral$, $IntLiteral$)
$LeftExpression$:=*Identifier* | *Identifier.FieldName* | *ClassType.FieldName*
| *Identifier*[$IntLiteral$]
$RightExpression$:=$ObjectCreation$ | $MethodCall$ | $LeftExpression$
| $null$
$Assignment$:=$LeftExpression$ = $RightExpression$
$Statement$:=$Assignment$ | $MethodCall$ | $Loop$
| $ThreadCreation$
$MethodDefinition$:=$ReturnType$ *MethodName*($Parameter*$)
{$Statement*$}
$ReturnType$:=*ClassType* | void
$MethodCall$:=*MethodName*($ArgumentList$)
$Loop$:=Loop(*Identifier*, $IntLiteral$, $IntLiteral$, $IntLiteral$)
{$Statement*$}
$ThreadCreation$:=spawn $MethodCall$

Fig. 3. The syntax of MOP.

since these operations cannot impact the GC performance in different runtimes. (2) The statements in MOP are runtime-agnostic. MOP does not use standard classes (e.g., String and Hashmap) or operations (e.g., ArrayCopy) in a specific runtime (e.g., Java), since they have different memory behaviors under different runtimes. For example, a String instance in Java has 2 fields and occupies 24 bytes, whereas a String in C# has 1 field only and occupies 16 bytes. Instead, MOP only uses primitive types and basic object operations, which can be used to build complex classes and operations that have consistent memory behaviors in different runtimes.

Based on the analysis of the workflows in popular GC implementations, we identify that GC workloads are influenced by object number, size and object reference relationships. Consequently, in the design of MOP, we utilize the memory usage information that influences these factors, including (1) *Object Structure* statements to depict the object size, (2) *Object Creation* statements to depict the object number and (3) *Object Reference Update* statements to depict the object reference relationships. Fig. 3 shows the syntax of MOP.

1) Object Structure: The memory size of an object depends on its structure, which influences the GC copy workload of the object. MOP uses *ClassDefinition* to depict the object structure, including the types, numbers, and orders of fields, all of which affect the object memory layout. MOP supports fields of primitive types and custom types, defined as either instance fields or static fields. To accommodate common managed languages, MOP also retains the class inheritance mechanism, allowing subclasses to inherit fields from superclasses.

2) Object Creation: The number of created objects contributes to the total consumption of heap memory, affecting the trigger of a GC cycle. MOP uses the *ObjectCreation* statement to describe the creation of objects that will be allocated in heap memory. The *ObjectCreation* statement supports normal object creation and array object creation. The creation of all objects requires specifying the class, the creation of array objects additionally requires length and dimension.

3) Object Reference Update: Modern GC mechanisms mark objects along the object references, and the object reference relationship affects the parallelism and sequence of GC processing. MOP uses the *Assignment* to change the reference relationship between objects. The changed reference in the *LeftExpression* of a *Assignment* can be a field of an instance, a static field of a class or a member of an array. The *RightExpression* can be the same types allowed in the *LeftExpression*, as well as a new object, a method return value or a null value.

4) Code Structure: MOP introduces *MethodDefinition* and *Loop* statements to reduce code size and increase the expression ability. *MethodDefinition* in MOP allows parameters, a return value and a block of other statements. The *Loop* statement in MOP requires an iteration index with its start value, end value and step size, which indicates the statements in its block will be executed multiple times. Additionally, MOP introduces *ThreadCreation* to build multithreaded memory operations, which indicates the spawn of a new thread to execute a specific method.

TABLE II
ILLUSTRATIVE CONVERSIONS FROM MOP TO JAVA, GO AND C#.

MOP	Java	Go	C#
byte, int, long	byte, int, long	int8, int32, int64	byte, int, long
class CName	class CName	type CName struct	class CName
NewInstance()	new CName()	&CName{}	new CName()
NewArray()	new Type[]	make([]Type, l)	new Type[]
Loop(){}	for() {}	for() {}	for() {}
ret name() {}	ret name(){}	func name() ret{}	ret name(){}
spawn func()	Thread(()->f())	go func()	Thread(()=>f())

B. MOP Programs to Runtime-Specific Programs

To create runtime-specific programs for the target managed language runtimes from a MOP program, we first establish the conversion relations between MOP and the target managed languages. Table II shows the conversion examples of Java, Go and C#. GEAR can also be extended to other managed languages that use a similar object model. Based on these conversion relations, it is straightforward to automatically convert a MOP program into a runtime-specific program for a target runtime. In the following, we use Class Structure and Method Conversion for illustration.

1) Class Structure Conversion: We first convert the class name and inheritance relationship using the syntaxes of target languages, and use a basic class with no fields as the default superclass of all defined classes (like *Object Class* in Java). We then convert the names and types of all fields in the class definitions, with primitive types converted to types in target languages that have the same size (e.g., long to int64 in Go), and reference types converted to object or pointer.

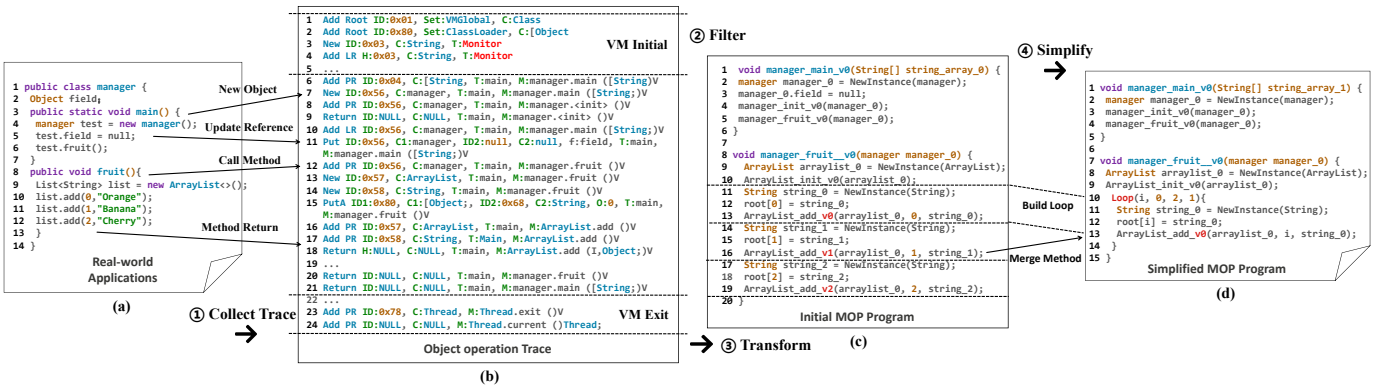

Fig. 4. An illustrate example for collecting memory operation trace and transforming memory operation traces into MOP programs.

2) Method Conversion: We first convert the name, return value, and parameters of the method according to the method declaration syntaxes in target languages. Then, for the method body, we split the code blocks and statements, and parse each statement using pattern matching. For each statement part, we perform the conversion according to the established conversion relations, and obtain the final method body by joining the converted statements and code blocks. For the thread spawn in MOP, we leverage the respective thread creation approaches in the target languages to execute the methods, and use their join approaches to synchronize the created threads.

C. Memory Operation Trace Collection

To build MOP programs that reflect real-world GC workloads, we first instrument the source code of a managed language runtime. This approach enables us to collect the memory operations of a real-world application during its execution, without changing its original code or memory behavior. In theory, we could instrument any managed language. We choose the Java runtime because it is the commonly used runtime with abundant applications. Specifically, we instrument the Java Virtual Machine (JVM) to collect the class information of created objects and sequential traces of all basic memory operations during a Java application's execution.

1) Class Information Collection: We collect the class information of created objects during the loading and initialization of the class. The collected information includes the types and names of classes as well as their fields. As each class is loaded only once when it is first used during execution, the traced class information is non-repetitive and non-redundant.

2) Memory Operation Collection: We collect the object creation trace when allocating heap memory for a new object. The ID and class of the new object are recorded (e.g., Line 7 in Fig. 4(b)). We also collect the trace of reference updates and references load when accessing an object field or a member of an array. Two linked objects and corresponding filed information are recorded (e.g., Line 11 in Fig. 4(b)). For method call information, we record the parameters input, changes to local variables, and the return value (e.g., Line 12-18 in Fig. 4(b)). For each trace, we also record the signature of the current method and thread during collection, which facilitates the transformation of method definition and thread creation later. In addition, we record the GC roots of class metadata and VM global variables in the JVM (e.g., Line 1-2 in Fig. 4(b)), which help construct the complete object reference graph later. We illustrate how we instrument the runtime to obtain this information in Section IV-A.

D. Memory Operation Traces to MOP Programs

To build a MOP program from a memory operation trace collected by the instrumented JVM, we first filter the traces to exclude application-unrelated logic. Then, we perform an initial transformation from the trace to MOP operation by operation. After that, we simplify the MOP program by merging duplicate method definitions and rebuilding loop structures.

1) Trace Filtering: We filter out the operations of application logic and exclude VM-related operations. As the example shown in Fig. 4, we first classify the traces based on thread tags and discard the operations of VM-related threads (e.g., Monitor thread). Then, from the operations of the main thread, we extract the operations between the call and return of the *Main* function to isolate the operations generated by application logic, and discard all operations collected during VM exit, which occurs after the *Main* function return. For the operations during VM initialization, we find that directly discarding them may lead to the omission of some root object operations. Consequently, we retain the operations of objects whose IDs appear in the *Main* function operations (e.g., Line 2 in Fig. 4(b)). We treat them as static variables and use a static block to perform their operations before the *Main* function.

2) Initial Transformation: During the initial transformation, we transform the class information into the class definitions in MOP, build the method definitions, and transform the object operation trace to MOP statements in the method body. The transformation of class information is straightforward. We iterate over all classes and their fields, and include them in the class definition. Then, starting from the *Main* function, we sequentially transform the memory operations to the corresponding MOP statements, and construct a new method when encountering a method call operation. During

Algorithm 1: MOP Program Simplification

```
Function Merge (methodCallTree) do
    methods ← getLeafMethods(methodCallTree)
    if methods = ∅ then
        return
    foreach m ∈ methods do
        h ← Hash(m)
        if h ∈ hashTable then
            merge m with hashTable[h]
        else
            hashTable[h] ← m
    foreach n ∈ methodCallTree do
        replaceCallsWithHash (n, hashTable)
    Merge (methodCallTree)
Function BuildLoop (m, wLen) do
    start ← 1
    while start + 2 · wLen ≤ mLen do
        prevHash ← Hash(m, start, wLen)
        curHash ← Hash(m, start + wLen, wLen)
        if prevHash = curHash then
            loop ← createLoop(m, start, wLen)
            cur ← start + wLen
            while cur + wLen ≤ mLen do
                nextHash ← Hash(m, cur + wLen, wLen)
                if curHash = nextHash then
                    addToLoop (loop, cur + wLen, wLen)
                    cur ← cur + wLen
                else
                    break
            start ← cur
        else
            start ← start + 1
```

the transformation of each method, a table of variables and object IDs is maintained, which helps us determine the source of objects processed within the method. Fig. 4(c) shows an example of the initial transformation.

3) MOP Program Simplification: The initial MOP program can be extremely large, since the repeated definitions of the same method and the unrolling of loops in original applications. To reduce code size, we first merge method definitions with the same process flow (e.g., *ArrayList_add* method in Fig. 4(c)). Algorithm 1 shows the merging algorithm. Starting from the methods at the leaf nodes of the method call tree, we encode each method by hashing its signature (name, parameters, return type), statements, and the hash values of the methods it calls. For methods sharing the same hash value, they are determined to have the same definition and workflow, and we merge them into the same method definition.

To construct loop structures in MOP (i.e., *Loop (index, start, end, step){Loop body}*), we identify and merge consecutive code segments that exhibit identical statements. As shown in Algorithm 1, for a method of considerable length, we employ a sliding window to detect repeated code segments in it. Each window, defined by a specified length, is encoded by hashing the statements that it contains. If the hashed value remains unchanged as the window slides, the code segments are identified as repeated operations. Then we merge these repeated code segments into a single *loop body*, and extract the values for the *loop conditions* based on the number of repetitions and the changing pattern of the index. Our approach can reconstruct loops in the original applications with regular index changes and consistent execution paths. However, loops with complex conditions cannot be reconstructed.

IV. Implementation

A. JVM Instrumentation

We instrument the most commonly-used Java runtime, the HotSpot VM in OpenJDK 17, to collect the memory operation trace during a Java application's execution. We focus on instrumenting the processing of specific bytecodes and keywords in the interpreter of JVM. To ensure that no operations are omitted due to the hierarchical compilation of the JVM, we use the compile command *-Xint* to disable the JIT compiler. For bytecodes that are processed in the template table with assembly instructions, we save all registers and call the collection functions to record the related information. We modify approximately 1000 lines of code in HotSpot.

1) Class Information: We collect the information of loaded classes when the VM parses the class file for metadata. We traverse all the fields and output their names and types.

2) Object Creation: We collect the object creation information by intercepting the handling of the bytecode *new*, which is the entry point for most allocation of normal objects and array objects. We also intercept other JNI methods used for creating objects, including object cloning and object constructors. In the instrumentation, we output the hashcode, class name of the object, and the basic dimension information of array objects.

3) Object Reference: We collect the object reference information by intercepting the handling of bytecodes related to writing/loading fields (e.g., *putfield*/*getfield*) and writing/loading array members (e.g., *aastore*/*aaload*). We also intercept the processing logic that implicitly modifies references, including *ArrayCopy*, object clone, and related JNI methods. In the instrumentation, we output the hashcode of objects on both ends of reference, as well as the offset and name of the field being written or loaded.

4) Root Object: For root objects related to active methods, we collect their information by monitoring the stack frame of the JVM interpreter. Specifically, we iterate through the *Local Variable Table* in the frame when a method is called, collect the information of input parameters. We monitor the bottom of the frame when the method returns to collect the information about return value. For VM-related root objects, we collect their information by monitoring *Oop Storage Set*. To collect the information of root objects related to class metadata, we monitor the *Oop Handles* of ClassLoader.

5) Thread Information: We collect the information about the thread that performs the object operation by acquiring the handler of the current thread. We attach the name of the current thread to each item in the trace.

B. Special Feature Handling

The conversion from a MOP program to runtime-specific programs is straightforward for most managed languages. However, some special language features in Go require additional handling. First, the Go language does not support static fields. To address this, we create a helper class for each class that has static fields, and use a global instance to store its static values. Second, Go does not support class and inheritance

relationship, GEAR uses interfaces and embedded structs to construct an equivalent object structure, which does not affect the object model in the Go runtime. Third, Go enforces restrictive type conversion, which makes GEAR unable to handle complex inheritance relationships in Java.

V. Evaluation

We evaluate GEAR on three widely-used managed language runtimes (i.e., Java, Go and C#), and answer the following two research questions.

- **RQ1:** Can GEAR create programs that share a consistent GC workload across different managed language runtimes?
- **RQ2:** Can GEAR create programs for different runtimes that exhibit real-world GC workloads? And at what cost?

A. Experimental Methodology

1) Environment Setup: We perform our evaluation on a physical machine with 20-core Intel(R) Xeon(R) Gold 5215 CPU, 128 GB of memory, running the CentOS Linux release 8.0.1905. We select three popular managed languages for our evaluation, i.e., Java, Go, and C#. The conversion relations for three target languages take 98, 128, and 104 lines of code, respectively. It generally takes about one day to add support for a language. We use the latest stable version of the runtime for each language, i.e., OpenJDK 17.0.2 HotSpot VM, Go 1.22.3, .NET 8.0.3.

2) Experimental Applications: We first manually craft 3 MOP programs, i.e., *WebServer* (141 LoC), *Database* (1012 LoC), and *BigData* (482 LoC), which simulate representative memory usage patterns in software systems. We use the core workflows and classes in the memory management of Apache Tomcat [47], Apache Cassandra [48] and Apache Flink [49] to craft *WebServer*, *Database* and *BigData*, respectively.

For real-world GC workload generation, we use 20 Java applications sampled from three data sources as shown in Table IV. (1) We first search the keyword *GC Benchmark* on GitHub [50]. In the 20 search results written in Java, we select the top 3 related projects. (2) We sample 6 applications from Guava Benchmark [43]. These applications involve the creation and manipulation of complex objects [51], which can lead to memory overhead and GC behavior. We sample these applications based on the criteria that they evaluate the creation of custom data structures, rather than computational algorithms or primitive types. The applications meeting the criteria are expected to create more objects and trigger complex GC behavior. From 13 applications that meet the criteria, we randomly select 6 applications. (3) We sample 11 Java solutions from Rosetta Code [44]. Rosetta Code offers programming tasks with corresponding solutions written in multiple languages. These solutions have been used in related work to evaluate the performance of different runtimes [32]. Rosetta Code contains 1229 tasks in total, 1173 of which have Java solutions. We sample these Java solutions based on the criteria that they use self-defined classes, rather than primitive types only. Solutions meeting the criteria are expected to create complex objects and trigger complex GC behavior. From the 119 Java solutions that satisfy the criteria, we randomly select 11 solutions.

TABLE III
Memory Comparison for Generated Runtime-Specific Programs

Application	Object Stats	Java	Go	C#	Diff %
WebServer	Count (Billion)	1172	1135	1195	5.0%
	Size (GB)	86705	83483	84762	3.7%
Database	Count (Billion)	108.7	108.9	109.2	0.5%
	Size (GB)	2252	2269	2355	4.4%
BigData	Count (Billion)	148.0	144.3	143.6	2.9%
	Size (GB)	2831	2836	2849	0.7%

3) Experimental Process: To validate the GC workload consistency of runtime-specific programs generated by GEAR, we focus on the total number, total size, and reference relationships of objects created in each runtime. We utilize profiling tools and GC logs in the target managed language runtimes to collect the required information. Specifically, we use $PrintClassHistogram$ for object statistics in Java, $gotoolpprof$ for Go, $dotnettrace$ and PerfView [52] tool for C#. Meanwhile, we use the heap dump tool to obtain heap snapshots at the common checkpoints and check object reference relationships using these snapshots.

B. Consistent GC Workload Construction

We convert the 3 manually crafted MOP programs into runtime-specific programs within a second. All created programs are executable on the target managed language runtimes without any manual modification.

We collect and calculate the total number and sizes of heap objects created under different runtimes using tools introduced in Section V-A3. We find certain memory optimizations adopted by the runtimes (e.g., compressed pointers, on-stack replacement) can reduce the number and size of objects created in the heap. To verify the consistency of GC workloads in all runtimes, we turn off these optimizations in our evaluation.

The statistical results after disabling memory optimization are listed in Table III. We can see that for all three applications, the maximum difference in object numbers and sizes is less than 5%. The difference arises from the loading and instantiation of different standard classes, as well as different logging processes in different runtimes. We further compare the similarity of the object reference relationships using the heap snapshots at the common checkpoints. We build the object reference graph based on objects and references recorded in the dump files, and compare the nodes and edges between the graphs from different runtimes. The results show over 95% similarity of objects among runtimes, aligning well with MOP programs. These results demonstrate GEAR's capability to construct a consistent GC workload in different runtimes.

C. Real-World Workload Generation

We present the metrics for real-world workload generation in Table IV. It can be observed that the Java applications tracing (the fourth column in Table IV) takes on average 32× longer time than the original application execution (the third

TABLE IV
REAL-WORLD GC WORKLOAD GENERATION

Source	Application	Orig	Trace	To MOP	Gen	Diff
Github	jvm-gc[4]	2.1s	34s / 1.2GB	152s / 629KB	2.7s	1.7%
	gcbenchmark[5]	1.5s	31s / 0.9GB	219s / 3113KB	2.6s	1.6%
	gcbench[6]	3.0s	35s / 1.0GB	242s / 462KB	4.2s	3.7%
Guava	Segment	2.7s	71s / 2.3GB	527s / 2615KB	3.8s	4.6%
	Monitor	1.4s	25s / 0.9GB	105s / 909KB	1.7s	1.6%
	SetCreation	3.0s	63s / 2.1GB	449s / 5915KB	3.6s	4.0%
	Map	2.9s	61s / 2.0GB	512s / 6074KB	3.6s	3.5%
	BinaryTree	2.1s	63s / 2.2GB	540s / 543KB	2.6s	3.8%
	ImmutableList	2.5s	44s / 1.3GB	223s / 2039KB	3.3s	4.4%
Rosetta	AVL Tree	106ms	24s / 0.8GB	127s / 248KB	153ms	3.1%
	Calkin-Wilf	142ms	25s / 0.9GB	105s / 315KB	263ms	4.1%
	AST interpreter	177ms	29s / 1.0GB	102s / 340KB	192ms	4.0%
	Fast Fourier	155ms	26s / 0.9GB	103s / 257KB	169ms	3.5%
	Numeric Error	102ms	21s / 0.9GB	95s / 229KB	123ms	4.8%
	Ormiston Triples	296ms	38s / 1.6GB	419s / 7758KB	396ms	4.2%
	Bell Numbers	281ms	36s / 1.5GB	315s / 4307KB	351ms	2.5%
	Resistor Mesh	114ms	31s / 1.1GB	167s / 975KB	170ms	3.5%
	Range	129ms	28s / 1.0GB	119s / 395KB	159ms	3.2%
	Tarjan	138ms	25s / 1.0GB	103s / 497KB	147ms	3.1%
	Universal Turing	177ms	24s / 0.9GB	121s / 540KB	238ms	3.3%
Average		1.15s	36.7s / 1.3GB	237.3s / 1908KB	1.52s	3.4%

column in Table IV). In the step of transforming memory traces into MOP programs, GEAR builds each MOP program from its trace within 10 minutes (the fifth column in Table IV). We observe that loop construction and method merging reduce the code size of the initially transformed MOP program by 84%, resulting in each final MOP program size of less than 8 MB. All generated Java and C# programs are executable on their respective runtimes.

The generated Java programs (the sixth column in Table IV) take 1.3× execution time on average compared to the original Java applications. The slowdown is mainly caused by repeated definitions of methods that have different execution paths, as well as undiscovered loops in the original Java applications. These repetitions reduce the effectiveness of the Just-In-Time (JIT) compiler, which compiles frequently executed methods and loops into faster native code during execution to speed up execution. We also compare the object statistics between generated Java programs and their corresponding original Java applications (the last column in Table IV), the maximum difference is 4.8%, which mainly comes from the repeated creation of root objects in VM global variables. The object statistics of user-defined classes are nearly identical. This result demonstrates that GEAR is capable of building real-world GC workloads from Java applications.

VI. CROSS-RUNTIME GC PERFORMANCE STUDY

We conduct an empirical study to analyze the performance differences of GC implementations across different runtimes and explore their potential root causes.

A. Study Methodology

1) Target GC Implementations: We use the same runtimes as described in Section V, and select one representative GC implementation in each runtime, i.e., Java ZGC, Go GC and C# server GC. Java and C# runtimes offer multiple GC implementations. In our study, we select the target GC implementations based on their popularity and the diversity of their designs.

2) GC Parameters: All target runtimes provide various parameters for tuning GC performance (e.g., the GC thread number, GC trigger conditions, region sizes, generation ratio, etc.). These parameters generally have default heuristic values and are adaptively adjusted during execution. We did not explicitly modify any of these parameters. As memory optimizations described in Section V-B may break the consistency of GC workloads, we turn off these memory optimizations in our main comparison, and specially analyze the effects of these optimizations in Section VI-B7.

3) Target Applications: We include the 12 MOP applications introduced in Section V-A2 in our comparison. These include three representative memory usage patterns (i.e., WebServer, Database and BigData) and nine real-world GC workloads generated from applications that are designed to evaluate the creation of data structures (i.e., three applications from GitHub and six applications from Guava Benchmark). Applications from Rosetta Code are omitted because they focus more on computation rather than memory usage. Additionally, we develop four MOP programs that employ multithreading to construct basic data structures with diverse object reference graphs, including *stack* (singly linked list reference graph), *queue* (doubly linked list reference graph), *tree* (binary tree reference graph), and *array* (array-based reference graph). In these four MOP programs, we use element objects within the data structures of the same class, and ensure a consistent number of unit objects. These applications cover a variety of object numbers, sizes and reference relationships, and object lifecycles, which represent diverse GC workloads.

4) Controlled Variables: To study the performance of GC implementations in handling different data sizes, we define *small(1×)*, *medium(1.5×)*, and *big(2×)* three data sizes for each application. Similarly, we define *low(1×)* and *high(2×)* two levels of parallelism. To evaluate the GC implementations under different memory pressures, we configure the heap size to 1×, 2×, and 3×, with 1× heap size near to the minimum size that all tested runtime will not trigger *Out of Memory* error. In summary, each application has $2 \times 3 \times 3 = 18$ different GC workloads. In addition, for the four basic data structure applications, we introduce *easy* and *hard* modes to simulate scenarios with high parallelism and large object graphs, as well as low parallelism with small object graphs. Table V summarizes the applications and their configurations.

5) Metrics Collection: We enable GC logging in the target runtimes and profile CPU and memory usage during program execution. We analyze these logs and profiles to collect metrics on execution time, GC performance and average resource usage. Each program is executed on a target runtime three times with a specified data size, application parallelism and heap size. We report the evaluation results by averaging the GC and execution metrics across the three runs.

[1]https://github.com/anadoba/jvm-gc-benchmark
[2]https://github.com/kbannach/jvm-gc-benchmark
[3]https://github.com/jeremysinger/gcbench

TABLE V
TARGET APPLICATIONS AND CONFIGURATIONS

Application	Parameters		
	1X Data Size	*1X Parallelism*	*1X Heap Size*
Stack-easy	24 GB	10	3 GB
Stack-hard	24 GB	1	256 MB
Queue-easy	24 GB	10	3 GB
Queue-hard	24 GB	1	256 MB
Tree-easy	24 GB	10	3 GB
Tree-hard	24 GB	1	256 MB
Array-easy	24 GB	10	3 GB
Array-hard	24 GB	1	256MB
WebServer	300 GB	10	2GB
DataBase	60 GB	10	6GB
BigData	150 GB	10	12GB
jvm-gc	90 GB	1	1GB
gcbenchmark	90 GB	5	1GB
gcbench	30 GB	5	1GB
Segment	30 GB	1	1GB
Monitor	30 GB	1	1GB
SetCreation	30 GB	1	1GB
Map	30 GB	1	1GB
BinaryTree	30 GB	1	1GB
ImmutableList	30 GB	1	1GB

B. Results and Analyses

In Fig. 5, we present the average execution time of each application, calculated across six loads and three heap sizes. We observe that the execution times for the same GC workloads vary significantly across different GC implementations, with the maximum difference for a single GC workload ranging from 19% to 312%. To understand these differences, we conduct a controlled variable analysis focusing on different application features and configurations, examining detailed GC metrics and execution metrics.

1) Reference Relationships: In the four applications that construct basic data structures, we use element objects in these data structures with the same class, number and size. The only difference lies in the object connectivity graphs, which we verify by examining heap dumps.

We find that Java ZGC achieves the best performance across all four applications under all configurations. An in-depth analysis reveals that Java ZGC consistently has the lowest average CPU usage, which is 39% lower than Go GC and 48% lower than C# GC. Since the four applications are write-intensive, we infer that the heavy write barrier overhead in Go GC and C# GC contributes to their lower performance. In contrast, Java ZGC employs a lightweight load barrier, providing significant advantages for write-intensive workloads.

We also find that Java ZGC and Go are more susceptible to object reference relationships, and they execute slower in applications where the object graph exhibits lower connectivity and aggregation. Specifically, for applications *stack*, *queue* and *tree*, where objects form lengthy linked lists or deep binary trees, Java ZGC and Go GC execute 1.66× / 1.44×, 1.67× / 1.45 × and 1.91× / 1.69× slower, respectively, compared to their execution in application *array*. In-depth analysis reveals that Java ZGC and Go GC struggle to perform efficient parallel GC working in singly and doubly linked lists, leading to long GC cycles. For instance, Java ZGC's average marking time in application *stack* is 2.1× to 3.8× of that in application *array*. In contrast, we observe that C# exhibits relatively stable performance across these four applications. Detailed analysis shows that the difference in single GC time for C# between four applications is less than 38%, demonstrating C#'s advantage in handling object reference graphs with singly or doubly linked lists.

> **Finding 1**: *Java ZGC and Go GC exhibit worse performance in applications with linked list reference graphs, whereas C# GC performance is less affected by object reference relationships. This indicates that Java ZGC and Go GC have performance issues when dealing with the linked list structure.*

2) Memory Usage Patterns: We evaluate three memory usage patterns commonly observed in software systems.

The usage pattern of *WebServer* involves continuously generating objects that are generally short-lived with simple reference relationships, and its memory requirement is relatively low. Consequently, we find that 98.2% GC cycles in generational C# GC are minor GC cycles targeting the young generation, with only 0.3% objects are long-lived enough to be promoted to the old generation.

C# GC achieves the best execution time, but it takes 7.38× GC pause time compared Go GC, which presents a trade-off between GC pause and throughput. Conversely, Go GC exhibits particularly poor execution performance in *WebServer*. In-depth analysis shows that Go GC is overly conservative in memory usage. Its heuristic GC trigger threshold is far below other GC implementations, resulting in only 29.5% / 24.1% average memory consumption and 3.35× / 16.84× GC cycle count compared to C# GC / Java ZGC.

> **Finding 2**: *In applications with an adequate memory budget, Go GC performs the worst among the evaluated GC implementations because its heap adjustment is overly restrictive. This indicates that Go GC should optimize its heap adjustment strategy.*

The usage pattern of application *Database* is continuously generating long-lived objects with complex reference relationships (e.g., write cache), and periodically releasing them (e.g., flush cache to disk). Consequently, we observe that the generational collector C# GC repeatedly performs major GC to mark and compact the whole heap. Furthermore, 90.8% of the created objects are promoted to the old generation under 1× heap size configuration.

As a result, C# GC performs the worst in application *Database*, and it executes even worse as the configured heap size increases to 2× and 3×. Our in-depth analysis reveals that increasing the heap size does not reduce the number of long-lived objects promoted to the old generation, with promotion ratios of 89.6% and 88.9% under the 2× and 3× heap size configurations, respectively. Conversely, non-generational Java ZGC and non-moving Go GC perform well in *Database* application. They experience no promotion overhead for long-lived objects, and exhibit higher memory efficiency due to reduced memory fragmentation.

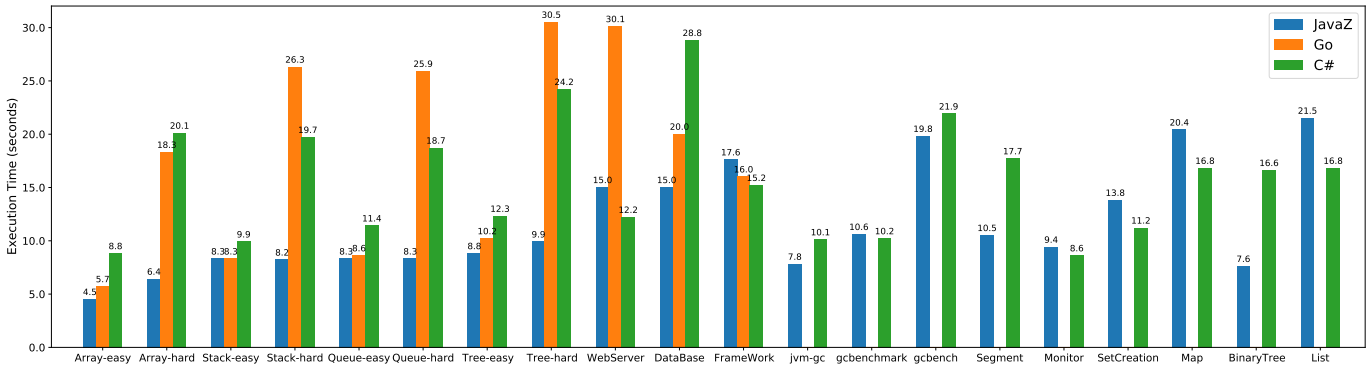

Fig. 5. Average execution time of different applications on target GC implementations. We do not have data of Go for applications from GitHub and Guava Benchmark, because Go does not support the complex inheritance relationships in these applications.

Finding 3: *In applications that continuously generate long-lived objects, C# GC performs 1.33× and 1.92× worse than Java ZGC and Go GC. This indicates the generational GC implementation of C# GC struggles to control the promotion overhead in such applications.*

The usage pattern of application *Bigdata* is maintaining numerous long-lived objects (i.e., cached data) throughout the execution, which occupies a significant proportion of memory. Consequently, we observe all GC implementations suffer from high-frequency and long-duration GC cycles due to intensive memory usage. Both GC time and execution time grow rapidly with the increase of data size. For the example of C# GC, when the data size increases to 1.5×, the GC time and execution time increase by 2.01× and 2.99× respectively.

The mostly-concurrent GC implementation Java ZGC does not perform well in application *Bigdata*. An in-depth analysis reveals that it suffers from frequent allocation stalls, occurring in 84% of its GC cycles. Allocation stalls arise due to memory exhaustion when GC cycle is still in progress, caused by long-running GC cycles. Java ZGC experience extremely long GC cycles in application *Bigdata*, because it performs whole heap marking in every GC cycle, resulting in its average GC cycle time is 84.7× that of generational C# GC.

Finding 4: *In applications with a large proportion of long-lived objects, Java ZGC executes 15.7% slower than C# GC due to long-duration GC cycles and frequent allocation stalls. This indicates the non-generational implementation of Java ZGC struggles with controlling the overhead of whole heap marking in such applications.*

3) Parallelism: We configure different levels of parallelism under the same total workloads for each application to study the impact of high concurrency. The result shows that Go runtime achieves up to 1.62× execution speedup when the parallelism increases from 1× to 2×, which is the most significant among tested runtimes. This indicates its GC implementation has minimal impact on CPU contention. By contrast, Java ZGC exhibits the worst performance improvement under increased parallelism, with even 1.15× slowdown in the application *BigData* when parallelism increases. An in-depth analysis reveals that Java ZGC suffers from 1.27× GC time increase and 1.27× average CPU usage increase as parallelism grows. We infer it incurs additional overhead synchronizing object pointer states under high parallelism.

Finding 5: *Java ZGC shows slowdowns in some applications when parallelism doubles, while Go GC consistently executes faster as parallelism increases. This indicates Java ZGC struggles to manage synchronization overhead under high parallelism.*

4) Data Sizes: We vary the data sizes from 1× to 1.5× and 2× for each application to study the scalability of GC implementations. The results show that Go GC has the best scalability, with its execution time increasing by 1.43× and 2.08× as the data size increases to 1.5× and 2×, respectively. The reason is that Go GC is implemented non-moving GC and its workflow will not change as the memory usage increases. As a result, its GC time grows linearly with the data size by 1.49× and 2.08× under 1.5× to 2× data sizes. Other evaluated GC implementations may experience different of GC cycles when the memory usage increases, leading to more complex and heavier GC overhead.

Finding 6: *Go scales the best as the data size increase, because its non-moving design ensures the linear relationship between GC overhead and data size.*

5) Heap Sizes: We configure the heap sizes from 1× to 2× and 3× for each application to study the performance of GC implementations under relaxed and intensive memory pressure. For the same application, the evaluated GC implementations generally perform better as the heap size grows.

6) CPU Consumption: We analyze the average CPU consumption under different GC implementations, the result shows C# generally consumes fewer CPU resources, using 41% less than Java ZGC in evaluated applications. The primary reason is that C# GC is a non-concurrent GC implemen-

tation. It does not incur additional GC workload for concurrent processing, resulting in minimal computational complexity.

7) Memory Optimization: We compare the execution metrics and GC metrics with memory optimizations enabled and disabled. For compressed pointer optimization in the Java runtime, we use Java G1 instead of Java ZGC for comparison, as Java ZGC does not support compressed pointer. Our results show that compressed pointer optimization reduces memory consumption by 20.6% on average, results in 23.8% GC time reduction, leading to up to 1.20× execution speedup. For on-stack replacement in the Go runtime, we find it reduces 7% to 32% heap object creation, results in 13% GC time reduction and 1.08× execution speedup. In summary, both compressed pointer optimization and on-stack replacement enhance GC performance and execution efficiency, making them valuable features for adoption in GC implementations in other runtimes.

VII. Discussion

Extending to other runtimes. MOP is designed based on the typical object model used in modern managed language runtimes, and includes only fundamental memory operations. This enables MOP to be converted to other runtimes that use similar object model, such as JavaScript, Python and Objective-C, regardless of the underlying mechanisms employed by the runtimes to achieve automatic memory management. To apply GEAR to a new language, we need to establish the conversion relations between MOP and the target language. For developers familiar with the target language, this typically requires around 100 lines of code and about one day of work, representing a relatively small effort.

Threats to validity. For the target applications, we carefully design and sample them to encompass diverse memory usage patterns. These applications vary in object quantities, sizes, reference relationships, and object lifecycles, representing diverse GC workloads. We test each application with three different data sizes, two levels of application parallelism, and three heap sizes, resulting in 18 distinct workloads. This comprehensive approach ensures our evaluation covers a representative sample of GC workloads.

Different runtimes will introduce some noise due to their varying execution efficiency of the same instructions. However, this kind of noise should have consistent consequences across different applications for different runtimes. In our experiment, we mainly analyze the inconsistent behaviors across different applications for different runtimes. In such cases, the inconsistent behaviors should be caused by the difference in GC implementations.

VIII. Related Work

A. Cross-language Evaluation

Lion et al. conducted an in-depth performance analysis of Java, Go, JavaScript and Python, using C++ as a baseline [31]. Prokopski et al. studied code-copying optimizations in the SableVM, OCaml, and Yarv interpreters [53]. Oliveira et al. investigated the energy consumption of different Android app development approaches (Java, JavaScript and C/C++) [37]. Marr et al. designed a benchmark suite for evenly comparing a common subset of language abstractions [39]. TailBench developed a methodology for measuring latency-critical applications in C++ and Java [42]. Nanz et al. utilized Rosetta Code [44] to present statistical findings [33] and studied the usability and performance of Chapel, Cilk, and Go in multicore workloads [32]. All these approaches do not specifically focus on GC performance evaluations. Thus, our work is orthogonal to these studies.

B. GC Algorithm Evaluation

Cai et al. [54] conducted a purified evaluation method to assess the minimum CPU and time overhead of application threads, subsequently calculating the relative overhead proportions of various GC algorithms. Sareen et al. [55] evaluated the spatial overhead of different GC algorithms, examining the impact of collection frequency, locality effects, and the influence of delayed collection by comparing them with immediate allocation and release methods. Zhao [22] restructured the classic G1 algorithm and performed modular quantitative evaluations of each technique employed. Blackburn et al. [56] and Yang et al. [57] analyzed the costs of read and write barriers in GC algorithms. Xu et al. [7] and Sriram et al. [58] conducted comparative testing of traditional GC algorithms within big data processing frameworks, evaluating the CPU and pause time costs of various GC implementations. All these works focus on evaluating GC implementations within the same runtime. Our work complements these studies.

IX. Conclusion

In this paper, we propose GEAR to create consistent GC workloads for different managed language runtimes. We first propose a group of runtime-agnostic Memory Operation Primitive MOP to portray the memory operations related to GC, and automatically convert a MOP program into runtime-specific programs. To build MOP programs with real-world GC workloads, we instrument the JVM to collect memory operation traces during a Java application's execution, and automatically transform and simplify a memory operation trace into a MOP program. We further perform an empirical study on three widely-used runtimes, and reveal some interesting findings about their GC performance.

In the future, we plan to extend GEAR to support additional runtimes and use existing MOP programs to evaluate their GC implementations, thereby identifying their potential performance shortcomings. Additionally, we plan to automatically generate more MOP programs, aiming to discover potential logic bugs in GC through these generated MOP programs.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China (62072444, 62302493), Major Project of ISCAS (ISCAS-ZD-202302), Basic Research Project of ISCAS (ISCAS-JCZD-202403), and Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044). This work was also partially supported by Huawei.

References

[1] T. S. BV, "Tiobe index for july 2024," Jul. 2024, accessed: 2024-07-31. [Online]. Available: https://www.tiobe.com/tiobe-index/

[2] GitHub, "The state of the octoverse," https://octoverse.github.com, 2024, accessed: 2024-07-24.

[3] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*, 2023.

[4] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications," in *Proceedings of International Symposium on Memory Management (ISMM)*, 2013, pp. 119–130.

[5] K. Suo, J. Rao, H. Jiang, and W. Srisa-an, "Characterizing and optimizing hotspot parallel garbage collection on multicore systems," in *Proceedings of EuroSys Conference (EuroSys)*, 2018, pp. 1–15.

[6] Y. Yu, T. Lei, W. Zhang, H. Chen, and B. Zang, "Performance analysis and optimization of full garbage collection in memory-hungry environments," *ACM SIGPLAN Notices*, vol. 51, no. 7, pp. 123–130, 2016.

[7] L. Xu, T. Guo, W. Dou, W. Wang, and J. Wei, "An experimental evaluation of garbage collectors on big data applications," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2019, pp. 540–583.

[8] R. Bruno, L. P. Oliveira, and P. Ferreira, "NG2C: Pretenuring garbage collection with dynamic generations for hotspot big data applications," in *Proceedings of ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2017, pp. 2–13.

[9] R. Bruno, D. Patricio, J. Simão, L. Veiga, and P. Ferreira, "Runtime object lifetime profiler for latency sensitive big data applications," in *Proceedings of EuroSys Conference (EuroSys)*, 2019, pp. 1–16.

[10] M. Wu, Z. Zhao, Y. Yang, H. Li, H. Chen, B. Zang, H. Guan, S. Li, C. Lu, and T. Zhang, "Platinum: A CPU-efficient concurrent garbage collector for tail-reduction of interactive services," in *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2020, pp. 159–172.

[11] The Go Authors, "A guide to the go garbage collector," https://tip.golang.org/doc/gc-guide, 2024, accessed: 2024-07-24.

[12] P. S. Foundation, "gc — garbage collector interface," https://docs.python.org/3/library/gc.html, 2024, accessed: 2024-07-24.

[13] OpenJDK, "Z Garbage Collector (ZGC)," https://wiki.openjdk.org/display/zgc/Main, Accessed: 2024-07-25.

[14] Microsoft Learn, "Workstation vs. server garbage collection (gc)," https://learn.microsoft.com/dotnet/standard/garbage-collection/workstation-server-gc, 2023, accessed: 2024-07-24.

[15] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proceedings of ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006, pp. 169–190.

[16] A. Prokopec, A. Rosa, D. Leopoldseder, G. Duboscq, P. Tuma, M. Studener, L. Bulej, Y. Zheng, A. Villazon, D. Simon *et al.*, "Renaissance: Benchmarking suite for parallel applications on the jvm," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 31–47.

[17] S. P. E. Corporation, "Specjbb®2015," https://www.spec.org/jbb2015/, 2024, accessed: 2024-07-24.

[18] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Proceedings of IEEE International conference on data engineering workshops (ICDEW)*, 2010, pp. 41–51.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of ACM symposium on Cloud computing*, 2010, pp. 143–154.

[20] N. Cohen and E. Petrank, "Data structure aware garbage collector," in *Proceedings of International Symposium on Memory Management (ISMM)*, 2015, pp. 28–40.

[21] A. M. Yang, E. Österlund, J. Wilhelmsson, H. Nyblom, and T. Wrigstad, "Thingc: Complete isolation with marginal overhead," in *Proceedings of ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2020, pp. 74–86.

[22] W. Zhao, S. M. Blackburn, and K. S. McKinley, "Low-latency, high-throughput garbage collection," in *Proceedings of ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2022, pp. 76–91.

[23] M. Wu, L. Mao, Y. Lin, Y. Jin, Z. Li, H. Lyu, J. Tang, X. Lu, H. Tang, D. Dong *et al.*, "Jade: A high-throughput concurrent copying garbage collector," in *Proceedings of European Conference on Computer Systems*, 2024, pp. 1160–1174.

[24] "Which language has a better garbage collection, java or golang?" https://www.quora.com/Which-language-has-a-better-garbage-collection-Java-or-Golang, 2024, accessed: 2024-07-29.

[25] "Garbage collection in java vs. c++ and python," https://www.quora.com/How-does-garbage-collection-work-in-Java-compared-to-other- languages-such-as-C-or-Python, 2024, accessed: 2024-07-29.

[26] H. News, "Anyone have a good comparison of .net vs java gcs? i never had trouble with the ..." https://news.ycombinator.com/item?id=25879344, 2024, accessed: 2024-07-29.

[27] "Which language has better garbage collection, java or .net (c#)? why?" https://www.quora.com/Which-language-has-better-garbage-collection-Java-or-NET-C-Why, 2024, accessed: 2024-07-29.

[28] M. Maas, K. Asanović, and J. Kubiatowicz, "Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 138–143.

[29] R. Bruno and P. Ferreira, "A study on garbage collection algorithms for big data environments," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–35, 2018.

[30] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu, "Gerenuk: Thin computation over big native data using speculative program transformation," in *Proceedings of ACM Symposium on Operating Systems Principles*, 2019, pp. 538–553.

[31] D. Lion, A. Chiu, M. Stumm, and D. Yuan, "Investigating managed language runtime performance: Why {JavaScript} and python are 8x and 29x slower than c++, yet java and go can be faster?" in *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2022, pp. 835–852.

[32] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, vol. 1, 2015, pp. 778–788.

[33] S. Nanz, S. West, K. S. Da Silveira, and B. Meyer, "Benchmarking usability and performance of multicore languages," in *Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 183–192.

[34] S. Jiang, R. Zeng, Z. Rao, J. Gu, Y. Zhou, and M. R. Lyu, "Revealing performance issues in server-side webassembly runtimes via differential testing," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 661–672.

[35] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Oil and water? high performance garbage collection in java with mmtk," in *Proceedings. 26th International Conference on Software Engineering*. IEEE, 2004, pp. 137–146.

[36] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021.

[37] W. Oliveira, R. Oliveira, and F. Castor, "A study on the energy consumption of android app development approaches," in *Proceedings of IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2017, pp. 42–52.

[38] M. Weber, C. Kaltenecker, F. Sattler, S. Apel, and N. Siegmund, "Twins or false friends? a study on energy consumption and performance of configurable software," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2098–2110.

[39] S. Marr, B. Daloze, and H. Mössenböck, "Cross-language compiler benchmarking: are we fast yet?" *ACM SIGPLAN Notices*, vol. 52, no. 2, pp. 120–131, 2016.

[40] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 662–676.

[41] O. Larose, S. Kaleba, H. Burchell, and S. Marr, "Ast vs. bytecode: interpreters in the age of meta-compilation," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 318–346, 2023.

[42] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[43] Google, "Guava: Google core libraries for java," https://github.com/google/guava, 2024, accessed: 2024-07-25.
[44] Rosetta Code, "Rosetta code: Programming tasks for various languages," https://rosettacode.org/wiki/Rosetta_Code, 2024, accessed: 2024-07-29.
[45] Baeldung, "Java memory layout explained," n.d., accessed: 2024-11-25. [Online]. Available: https://www.baeldung.com/java-memory-layout
[46] Baeldung, "Understanding java garbage collection roots," n.d., accessed: 2024-11-26. [Online]. Available: https://www.baeldung.com/java-gc-roots
[47] The Apache Software Foundation, "Apache tomcat," 2024, accessed: 2024-07-25. [Online]. Available: https://tomcat.apache.org
[48] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS operating systems review*, vol. 44, no. 2, pp. 35–40, 2010.
[49] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.
[50] GitHub, Inc., "Github: Where the world builds software," 2024, accessed: 2024-11-26. [Online]. Available: https://github.com
[51] H. Almulla, A. Salahirad, and G. Gay, "Using search-based test generation to discover real faults in guava," in *Search Based Software Engineering: 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings 9*. Springer, 2017, pp. 153–160.
[52] Microsoft, "Perfview," 2024, accessed: 2024-07-25. [Online]. Available: https://github.com/microsoft/perfview
[53] G. B. Prokopski and C. Verbrugge, "Analyzing the performance of code-copying virtual machines," *ACM Sigplan Notices*, vol. 43, no. 10, pp. 403–422, 2008.
[54] Z. Cai, S. M. Blackburn, M. D. Bond, and M. Maas, "Distilling the real cost of production garbage collectors," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, pp. 46–57.
[55] K. Sareen and S. M. Blackburn, "Better understanding the costs and benefits of automatic memory management," in *Proceedings of International Conference on Managed Programming Languages and Runtimes*, 2022, pp. 29–44.
[56] S. M. Blackburn and A. L. Hosking, "Barriers: Friend or foe?" in *Proceedings of international symposium on Memory management*, 2004, pp. 143–151.
[57] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking, "Barriers reconsidered, friendlier still!" *ACM SIGPLAN Notices*, vol. 47, no. 11, pp. 37–48, 2012.
[58] A. Sriram, A. Nair, A. Simon, S. Kalambur, and D. Sitaram, "A study on the causes of garbage collection in java for big data workloads," in *Proceedings of IEEE International Conference on Big Data (Big Data)*, 2020, pp. 5831–5833.