Resource Management in Native Languages Using Dynamic Binary Instrumentation (PIN)

Nachiketa Chatterjee, Saurabh Singh Thakur and Partha Pratim Das

Abstract Managed programming languages like Java and C# perform resource management as a part of their language specification. They use a runtime system like JVM or CLR for the management. In contrast native languages like C and C++, designed to provide strong foundation for programs requiring speed or tight coupling with operating system or hardware, are used with manual resource management. These do not require the runtime system. Naturally, it will be nice to have a managed layer for native languages which can be plugged in as and when we want to manage resources in any point of time during execution. In this paper, we present a GC Pintool which automates the garbage collection for C programs at run time using PIN (a framework for dynamic binary instrumentation). Efficacy of the GC Pintool has been tested over various benchmark C programs and our GC approach using PIN is found to be correct and precise.

Keywords Garbage collection · Memory leak · Dynamic instrumentation

N. Chatterjee (✉)
A. K. Choudhury School of Information Technology, University of Calcutta, Kolkata, India
e-mail: nachiketa.chatterjee@gmail.com

S.S. Thakur
School of Information Technology, Indian Institute Technology, Kharagpur, India
e-mail: saurabhjan07@gmail.com

P.P. Das
Department of Computer Science and Engineering, Indian Institute Technology, Kharagpur, India
e-mail: partha.p.das@gmail.com

© Springer India 2016

R. Chaki et al. (eds.), *Advanced Computing and Systems for Security*,
Advances in Intelligent Systems and Computing 396,
DOI 10.1007/978-81-322-2653-6_8

1 Introduction

The application programming languages available in market can be categorized into two types depending upon their style of execution and resource management. Most of the popular languages like Java, C#, etc., can automatically manage their resources such as memory, graphics wizard, etc. So that they usually are termed as *managed languages*. But, the managed languages need a runtime system for execution that adds additional overhead. In contrast, native languages like C and C++ facilitate users to write high performance and responsive applications with direct interaction with hardware resources. But, the user experiences the overhead to manage resources manually. With the speed and flexibility of C and C++ comes increased complexity along with the complications in memory management. Objects must be created and destroyed explicitly in program, and small mistakes in this process can cause severe complications.

Garbage collection (GC) is the most popular automated technique for memory management. A garbage collector detects objects that are not being used by the program and attempts to reclaim the memory (garbage) occupied by those objects. It liberates the programmer from the responsibility of taking care of dynamically allocated memory and is based on the following principles:

1. To identify the objects in memory that cannot be accessed any further, and
2. To destroy these objects and reclaim the memory used by them.

While garbage collection is used, certain categories of bugs, as described below, get eliminated or are substantially reduced.

1.1 Memory Leak

When an object becomes unreachable, the program fails to free its memory and leak occurs. This memory then becomes unavailable to the system. Series of memory leaks may cause a program to crash due to memory exhaustion. Even if not, then also it can have an adverse effect on performance. Chunks of allocated but unused memory cause fragmentation. It destroys the spatial locality and this can result in poor cache performance or an increase in paging.

1.2 Dangling Pointer

When the program holds more than one pointer to a memory, the memory is made free through one of them, and is then accessed by another, an illegal dereferencing happens for the dangling pointer. By the time of access, the memory may have been

reassigned for some other use. Such dangling pointer access, therefore, may lead to unpredictable results.

Managed programming languages currently in use perform Garbage Collection automatically. So, it will be nice to have a managed layer for native languages which can be plugged in when we want to manage resources in any point of time during execution.

We organize the paper as follows. Section 2 describes about the various resource management techniques in brief. Section 3 discusses about the strategy of resource management using dynamic instrumentation. Section 4 explains about the details of implementation of our resource management framework; discusses its various modules and their implementation. The results for the test cases are presented in Section 5. In this section, we will discuss our test plan and the various benchmark codes that we have used to test the GC Pintool. Section 6 concludes and suggests some possible future research directions resulting from this work.

2 Resource Management Techniques

There are a couple of classical resource management mechanisms available in managed languages those that have been summarized as below. Also, we found some attempts of recourse management for native languages.

2.1 Classical Resource Management

In general, a garbage collection process involves three basic steps as shown in Fig. 1, i.e., scanning the root, marking, sweeping, and an optional step of compacting. There are two generic tasks that a garbage collector needs to perform

1. Distinguish between garbage and live objects and
2. Remove garbage to reclaim memory.

The garbage collection process is initiated with the root scanning that examines if a memory object includes any pointer. A reachability graph is then formed with

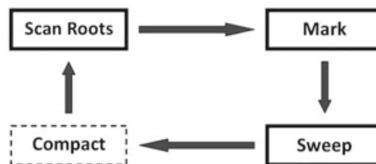

Fig. 1 Typical garbage collection cycle

the objects and the pointers to the objects. Naturally, one can traverse this graph to compute if an object is reachable. When an object is reachable (from the root) in this graph, the program can (potentially) access it by navigating the pointers to it and other objects. Hence, all reachable objects are detected and marked. Objects that are not marked are not reachable, and are garbage by definition [1]. One can then sweep (or walk) through the heap and deallocate the unmarked objects in the process of reclamation. Alternately, in some systems, compaction is used on the heap to narrow the gaps created by unmarked objects that are removed. Compaction minimizes external fragmentation [2] at the cost of object relocation. Recently, several garbage collection techniques have been proposed with their respective properties. They can be grouped as follows:

Reference Counting is one of the oldest resource management techniques, where every object has a counter holding the number of pointers that point to the object. The counter is incremented when a new pointer starts pointing to the object. When a pointer referencing the object is reassigned, the counter is decremented. The object becomes unreachable by the *mutator* [3] (executing program) as soon as the counter becomes zero. It is then returned to the free pool. The strength of this technique apart from its simplicity is immediate recovery of unreferenced memory objects [4]. However, this method cannot handle cyclic references as the two (or more) objects on the cycle are having nonzero counter value [5]. They are left unreclaimed.

Mark and Sweep GC, as the name suggests, first *marks* the reachable objects and then *sweeps* them. It adheres strongly to the two-phase abstraction of garbage collection. Marking is done recursively starting from the root set. For sweep, the heap is scanned linearly and all unmarked objects are reclaimed [3, 6].

Copying Collector copies reachable objects from one part of memory to another [3] and reclaims the garbage in the process.

Generational Garbage Collectors are based on *generational hypotheses* and attempt to improve performance by using the age of objects. The *weak generational hypothesis* professes that most objects must die when they are young [7]. In contrast, the *strong generational hypothesis* assumes that as an object becomes older, it becomes less likely to die. Generational collectors have been shown to generally outperform their nongenerational counterparts [8], and are today the most commonly used type of collectors for the majority of systems.

Incremental Garbage Collectors are meant to minimize the disruptiveness of collectors, specifically, those that have long pauses. For this, the collector is run in tandem with the application program so that it can gradually do its work [9].

2.2 Resource Management for Native Language

Two approaches for adding GC to C, namely *Conservative GC* [10] and *Precise GC* [11], have been proposed earlier.

Conservative GC attempted to automate GC for C program where the programmer needs to link with *Boehm GC Library* and use its allocator/deallocator

functions. This GC technique operates roughly in four phases of the Mark-Sweep algorithm. But it falls short in identifying pointer variables in some cases where it is unable to determine whether a word is a pointer or wrongly assumes the dead pointers as roots and leaves objects indefinitely in the heap. For long-running programs like an IDE, a web server or an operating system kernel, *Conservative GC* does not perform well. While managing threads and continuations, it can potentially cause unbounded memory use due to linked lists [12]. The problem is caused by liveness vagueness [13], rather than type imprecision.

Precise GC [11] is an improvement over the conservatism of *Conservative GC* [10] in terms of memory and time. It uses Magpie and performs source-to-source transformation that rely on an ontology of objects including "root references in heap," "location of reference in every kind of object," and "types of objects in the heap." This results in overhead for the mutator besides requiring additional programmer effort and related complexities. For example, while using *Precise GC* for a C program, a pointer must never be extracted from a variable typed as long—not at least after a collection has taken place since the variable was assigned. However, compared to *Conservative GC*, *Precise GC* makes weaker assumptions on the compiler and the architecture. The original program is transformed to explicitly cooperate with the GC.

In this paper, we have investigated the solution to the problems of memory management in C programs. To overcome the shortcoming of *Conservative GC* [10] and *Precise GC* [11], we use dynamic instrumentation that does not need any modification in the source code as were required for both the above mentioned GC techniques. In this work, a GC Pintool has been developed which automates the process of garbage collection for C Programs. The tool addresses a wide range of memory management issues for C programs and efficiently improves the performance of C programs in context with the memory management.

3 Resource Management Using Dynamic Instrumentation

To design a pluggable managed layer for native languages, we have used a dynamic instrumentation framework called Pin [14]. It is a platform for runtime binary instrumentation of applications running on Linux. A broad range of program analysis tools, called *Pintools*, can be built under this framework. The instrumentation of binary at runtime dynamically generates code, allows generic morphing, and alleviates the need to modify or recompile the source.

Here, we designed the GC Pintool in Fig. 2 to identify the dereferenced memory addresses allocated during the application execution, but never freed up. Then, after the end of each block execution all dereferenced memory can be released using this tool and logged. PIN can be executed in the probing mode for dynamic invocation of GC Pintool as and when required.

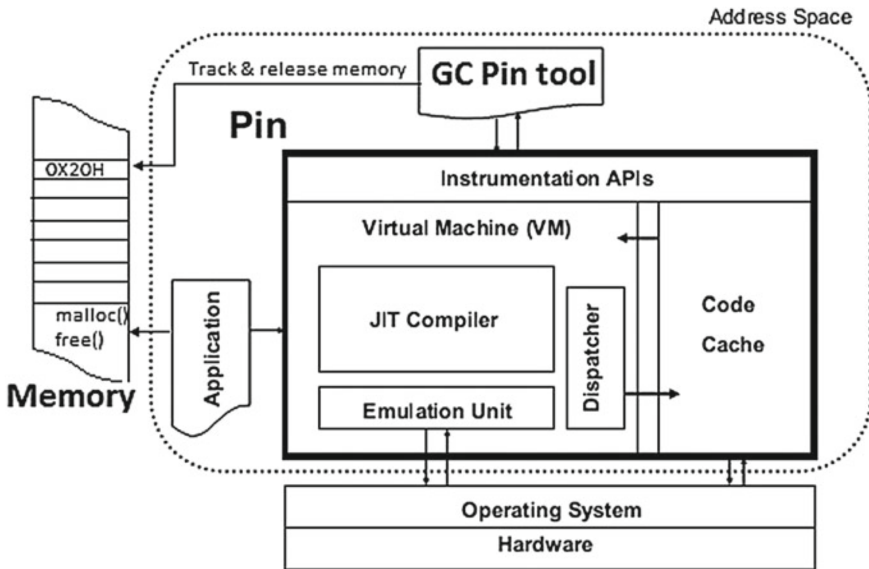

Fig. 2 Software architecture of PIN

4 Implementation

In this section, we will discuss the implementation infrastructure for GC Pintool. The GC Pintool follows the basic principle of garbage collection, that is

1. Find inaccessible objects in memory and
2. Free these objects and reclaim the memory.

Further, since the GC Pintool is a dynamic instrumentation tool, so every action happens at runtime which ensures zero modification in the source code. Now, first the modus operandi of the tool is explained and then later various Pin APIs that have been used by the Pintool to make GC possible at run time are described. Throughout the discussion carried over in this section, the two terms namely Mutator: to symbolize the application program; and Collector: to signify the GC Pintool, has been frequently used. Initially, when mutator starts its execution under Pins control, the collector sets its breakpoint as per the instrumentation routines. The major breakpoints for the instrumentation are as follows:

1. Calling of any user function
2. Calling of any Memory Allocator function like malloc(), calloc(), and realloc().
3. Calling of any Memory Deallocator function like free().
4. Return from any user function

Whenever any function of mutator is called, it is recorded by the collector. Collector further waits for any heap allocation that is, dynamic memory allocation to be made by the mutator. The collector captures the memory location of the heap being allocated and save it in its master data structure along with some other information in which scope (function) the allocation is made. In this manner, all the allocations made by the mutator get captured by the collector in its data structure. If there is any deallocation made by mutator, it is also checked in the data structure and that log is removed from the data structure. When any function completes its execution, the collector program sweeps away all dynamic memory allocated by that function. Important is, collector takes care of the allocations made against global or, if any reference is passed to some other function then in those cases, collector does not make any deallocation rather that the scope is changed for those allocations suitably. This way the collector ensures that there is no illegal or premature deallocation. Now, below we will present the GC Pintool algorithm

```
Algorithm: GC Pintool (GCMAP<fname, memaddr>, fname, memaddr)

 This Algorithm Logs the details of Memory allocated dynamically and Free the memory
 when it is Leaked or becomes Garbage, simultaneously during the execution of C Program.

// Global Initialization:
GCMAP<fname, memaddr> ; // Global Map to keep Garbage Record
char* fname; // To store Name of function being called
Address memaddr, mem_free; // To store the memory addresses returned by Allocator function

// This part will execute when any user/main function is called.
fname = Function_Name;

IF (Memory Write)
  IF(Memory Write == Global Reference)
    Global; fname = main; // scope change
    GCMAP.insert(<fname, memaddr>);
  End IF
  IF(Memory Write == Parameter Reference)
    fname = Function_Name; // scope change
    GCMAP.insert(<fname, memaddr>);
  End IF
// This part will execute when Mem. Allocator functions
// that is, malloc(), calloc() & realloc() are called.
GCMAP.insert (<fname, memaddr>);
// memaddr store the address returned by  memory allocator function

// This part will execute when free() function is called
mem_free = Address returned by free();
While (GCMAP.begin() to GCMAP.end())
   IF(GCMAP.memaddr == mem_free)
      GCMAP.erase (<memaddr>);
   End IF
End While

// This part will execute at the Exit of any user/main function.
While ( GCMAP.begin() to GCMAP.end() );
  free(GCMAP.memaddr) // Garbage Created by Returned Function
GCMAP.erase(GCMAP.memaddr);
```

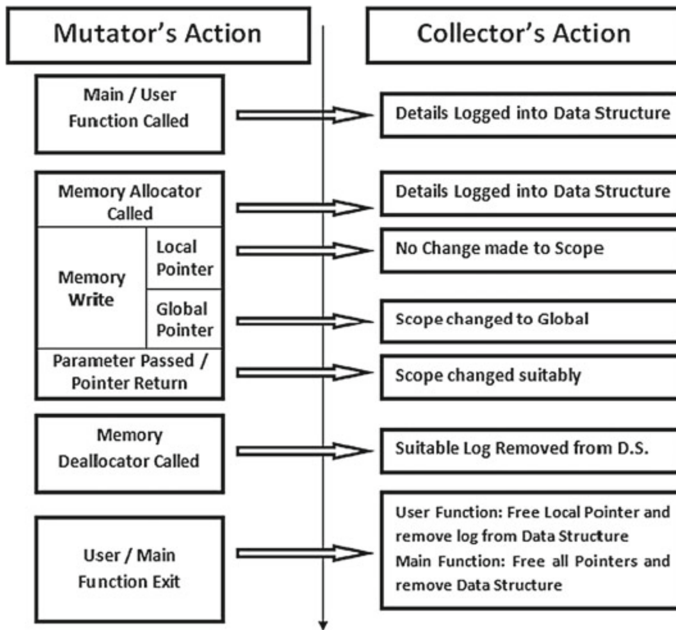

Fig. 3 GC approach using dynamic instrumentation

The Flow Chart depicting the GC approach as described in above algorithm is as shown in Fig. 3.

In order to keep the logs of memory addresses which are allocated or deallocated dynamically, a master data structure has been used by the GC Pintool. During the execution of the program, this data structure keeps on changing as per the instrumentation and accordingly GC Pintool collects information regarding performing the garbage collection at a suitable point of time. Figure 4 shows the typical transition in the data structure during the program execution.

Some important Pin APIs used in the Pintool are discussed below

RTN_InsertCall(RTN Rtn, IPOINT Action, AFUNPTR Funptr, ⋯) This API is used to insert a call relative to a routine (rtn) and a suitable action is taken like IPOINT_BEFORE to call funptr before execution of rtn, or IPOINT_AFTER for immediately before the return from rtn. There are various IARG_TYPE arguments to pass to funptr.

PIN_CallApplicationFunction(Const CONTEXT * Ctxt, THREADID Tid, CALLINGSTD_TYPE Cstype, AFUNPTR OrigFunPtr, ⋯) This API allows the tool to call a function inside the application. The function is executed under control of Pin's JIT compiler, and the application code is instrumented normally. Tools should not make direct calls to application functions when Pin is in JIT mode.

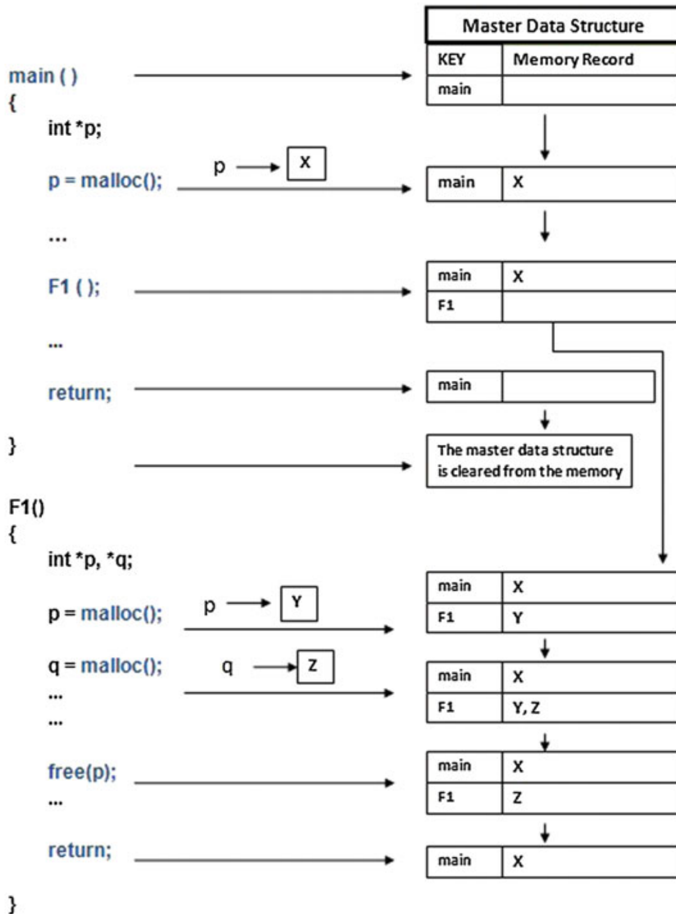

Fig. 4 Transition in data structure during program execution

For that reason, to call mutator's free() this API has been used. This API in turn deallocates the memory that has been allocated dynamically by the mutator.

PIN_SafeCopy(VOID * Dst, Const VOID * Src, Size_T Size) This function is used by our tool to ensure safe access to the original content of the application's memory from our tool. This API is helpful in confirming the type of reference, i.e., local, global, etc. On this basis, the GC Pintool makes decision regarding the scope of the allocation.

The instrumentation algorithm in GC Pintool has been tested on various benchmark programs and is found to perform accurate detection of memory errors and garbage collection for C programs.

5 Functional and Performance Testing

In this section, we first present the test plan for correctness of GC Pintool and then discuss how the performance of the tool has been assessed.

5.1 Correctness of GC Pintool

Table 1 presents a test plan for the GC Pintool. It covers different scenarios for memory issues in terms of variables of a C program while dynamic memory allocation is used. Based on the test plan, a benchmark test suite comprising 168 small to medium C programs was created. In addition, 10 C programs with memory issues as reported in different user forums were also added to the test suite. GC Pintool performed correctly for the whole test suite.

5.2 Performance of GC Pintool

Since GC Pintool relies on dynamic instrumentation, the performance of a C program running under the tool is expected to be significantly degraded compared to a natively running C program. To estimate the performance impact, we recall that GC Pintool performs two primary tasks: (1) Detect memory leaks and other issues and (2) Releases memory that is no more usable.

The detection is implicit because unlike usual memory tools, no report is made to the user; rather the information is used by the tool to release appropriate resources. So, in this part, GC Pintool closely resembles the functionality of Valgrind and we can make a direct comparison. It may be noted that Valgrind experiences 10–50 times slowdown[1] and we would expect similar behavior.

In comparison with the Valgrind's behavior, we prepared eight C programs containing a variety of memory scenarios from Table 1 and executed them for five varying data sets (using 10 MB–2.5 GB memories) under both Valgrind and GC Pintool.[2] While GC Pintool always performed correctly, we find that it runs about 35 % faster compared to Valgrind. This is rather encouraging, given that Valgrind is a widely used tool.

In the other part (where memory is actually released to achieve GC), there is no reference to compare against. So, we run GC Pintool (with doing GC) and compare against the run of the tool that only performs the detection, but does not release. We find that for the above programs, our tool runs about one order slower when it performs GC.

[1]"2.1. What Valgrind does with your program" in http://valgrind.org/docs/manual/manual-core.html.

[2]We needed to tweak the tool to report memory issues.

Table 1 Test plan for GC pintool

Type	Func.	Block	Case 1	Case 2	Case 3	Case 4
Scenarios for local variables						
Local	Single	Function	1 Alloc.	2 Alloc.	2 Alloc.	2 Alloc.
			No Dealloc.	1 Dealloc.	2 Dealloc.	No Dealloc.
		Nested	Alloc. outer	Alloc. outer	Alloc. inner	Alloc. inner
			Dealloc. outer	Dealloc inner	Dealloc. outer	Dealloc. inner
		Loop	Alloc.	Alloc. outer	Alloc. inner	Alloc. inner
			No Dealloc.	Dealloc. outer	Dealloc. outer	Dealloc. inner
		Recursive	Alloc.			
			No Dealloc.			
	Two	Function	1 Alloc.	2 Alloc.	2 Alloc.	2 Alloc.
			No Dealloc.	1 Dealloc.	2 Dealloc.	No Dealloc.
		Nested	Alloc. outer	Alloc. outer	Alloc. inner	Alloc. inner
			Dealloc outer	Dealloc. inner	Dealloc. outer	Dealloc. inner
		Loop	Alloc. outer	Alloc. inner	Alloc. inner	
			Dealloc. outer	Dealloc. outer	Dealloc. inner	
		Recursive	Alloc.			
			No Dealloc.			
Scenarios for global variables						
Global	Single	Function	1 Alloc.	2 Alloc.	2 Alloc.	2 Alloc.
			No DeAlloc.	1 Dealloc.	2 Dealloc.	No Dealloc.
		Nested	Alloc. outer	Alloc. outer	Alloc. inner	Alloc. inner
			Dealloc. outer	Dealloc. inner	Dealloc. outer	Dealloc. inner
		Loop	Alloc outer	Alloc inner	Alloc inner	
			Dealloc outer	Dealloc outer	Dealloc inner	
		Recursive	Alloc.			
			No Dealloc.			
	Two	Function	Alloc in 1	2 Alloc in 1	2 Alloc in 1	2 Alloc in 2
			Dealloc in 2	No Dealloc	1 Dealloc	No Dealloc.
Scenarios for parameters						
Param	Two	Function	One Alloc., Reference returned back	Two Alloc., One Dealloc, One Reference returned back	Two Alloc. Two Dealloc.	Reference is Pointer to Pointer

Type	Func.	Block	Cases
Scenarios for member variables and special cases			
Data member	Single	Function	Allocation against a data member of structure
			Allocation against a pointer to a structure
			Allocation against pointer to a structure till out of memory
Local			Pointer reassignment
			Allocation against the array of pointers

6 Conclusion

Memory Management is an important aspect for any programming language. Inefficient management of the memory in a program may lead to various consequences like memory leak, dangling pointer, double free error, etc., resulting in slow down of the program, fragmentation, incorrect execution, premature GC, or program termination. As a solution to these memory management problems, specifically for C programs, a novel garbage collection approach has been proposed and developed in this paper that uses dynamic binary instrumentation which is accurate and does not need any modification in the source code.

The GC Pintool has been successfully tested over a large set of distinct test codes. It takes care of all the dynamically allocated memory whether it is allocated against a local pointer variable, global pointer variable, or passed as parameter to some other function. The tool deallocates the reserved memory at proper time during the execution of the program. The tool has been shown to successfully detect any kind of memory leak error and performs the garbage collection suitably. Further, it is capable of handling issues like pointer reassignment, allocation made against the data members of an object, arrays, array of pointer, and pointer to an array. GC Pintool overcomes the drawbacks of *Conservative GC* [10, 12] and *Precise GC* [11] and has been shown to run faster than Valgrind for memory leak/error detection.

The GC Pintool is a work in progress. The present version is a functional prototype, intended to operate on moderately large C programs to provide an understanding of its behavior and to provide a platform for adding future enhancements. The present tool may be extended in the following directions:

- *Support for C++* Internally GC Pintool already has most of the infrastructure required for C++. We can log the memory allocated and deallocated by new and delete operators, but the challenge will be to deal with the constructors and destructors of an object.
- *Improving Efficiency* At present, GC Pintool performs selective instrumentation which has helped to restrict the slowdown while dynamic instrumentation is used. This is manifest in GC Pintool which has a better performance than Valgrind when it is used only for detection of memory issues, but no *Garbage Collection* (GC) is actually done.

  Unfortunately, while doing the GC, GC Pintool is confronted with a second level of slowdown because it needs to dynamically call appropriate free function for any memory that is about to be leaked. It may be noted that this call actually does not exist in the user code and hence needs to be inserted at the runtime. GC Pintool achieves this by using PIN_CallApplicationFunction, a function of PIN. Incidentally, this function has a lot of overhead and substantially slows down the GC Pintool further. We are working on a few schemes to overcome this shortcoming—one is to move to a lazy collection strategy and the other is to use the fact that GC only needs to call a fixed function, namely, free.

Further, the multimap structure used to log memory allocations and deallocations, may be improved to reduce space complexity.

- *Testing on Legacy programs* GC Pintool should be tested on large and complex legacy programs so as to ensure its applicability in production environments.

References

1. Cohen, J.: Garbage collection of linked data structures. Comput. Surv. **13**(3), 341–367 (1981)
2. Cohen, J., Nicolau, Alexandru: Comparison of compacting algorithms for garbage collection. ACM Trans. Program. Lang. Syst. **5**(4), 532–553 (1983)
3. Wilson, P., Johnstone, M., Neely, M., Boles, D.: Dynamic storage allocation: a survey and critical review. In: Proceedings of the International Workshop on Memory Management, Kinross Scotland (UK) (1995)
4. Jones, R.E.: Garbage collection: algorithms for automatic dynamic memory management. Wiley, Chichester (1996)
5. Harold McBeth, J.: On the reference counter method. Commun. ACM **6**(9), 575 (1963)
6. McCarthy, John: Recursive functions of symbolic expressions and their computation by machine. Commun. ACM **3**, 184–195 (1960)
7. Ungar, D.: Generation scavenging: a non-disruptive high performance storage reclamation algorithm. In: Proceedings of the ACM Symposium on Practical Software Development Environments, pp. 157–167 (1984)
8. Blackburn, S.M., Cheng, P., McKinley, K.-S.: Myths and reality: the performance impact of garbage collection. In: Sigmetrics—Performance 2004, Joint International Conference on Measurement and Modeling of Computer Systems, New York (2004)
9. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M:. On-the-fly garbage collection: an exercise in cooperation. Commun. ACM **21**(11), 965–975 (1978)
10. Boehm, H.-J.: Space efficient conservative garbage collection. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, pp. 197–206. ACM Press (1993)
11. Rafkind, J., Wick, A., Regehr, J., Flatt, M.: Precise garbage collection for C. In: Proceedings of ISMM 09 International Symposium on Memory Management, pp. 39–48. ACM Press (2009)
12. Boehm, H.-J.: Bounding space usage of conservative garbage collectors. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 93–100. ACM Press (2002)
13. Hirzel, M., Diwan, A., Henkel, J.: On the usefulness of type and liveness accuracy for garbage collection and leak detection. ACM Trans. Program. Lang. Syst. **24**(6), 593–624 (2002)
14. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Pin, H.K.: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (Chicago, IL, USA, June 12–15, 2005)