2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)

Research on Stack-allocation based JVM Garbage Collection

Xiaoliang Xu
College of Computer Science
Hangzhou Dianzi University
Hangzhou, Zhejiang, 310018, P.R.China
xxl@hdu.edu.cn

Jiang Shen
College of Computer Science
Hangzhou Dianzi University
Hangzhou, Zhejiang, 310018, P.R.China
sueja@163.com

***Abstract*—In order to reduce the costs of Java virtual machine garbage collector, a stack-allocation based on garbage collection algorithm is proposed in this paper. The stack frame is improved to support the object storage, and the stackable objects are identified with expanded byte code when the compiler compiles the Java source code. The stackable objects are allocated to Java stack and released instantly when they leave their scope, and the other objects are still allocated to heap and released by garbage collector. Experimental result shows, compared to traditional garbage collector, that new algorithm improves the performance of memory allocating and releasing, reduces the burden of garbage collector, and runs faster.** *(Abstract)*

Keywords Java virtual machine; garbage collection; stack allocation (key words)

I. INTRODUCTION

One of the attractions of Java is that it automatically manages all memory usage [1]. So the programmer does not have to worry about deallocating the memory when it is no longer needed. The problem is that garbage collection can significantly slow down the execution of a program [2].

There has been a lot of work on how to improve the garbage collection [3]. Optimization technology of garbage collection algorithms in recent years focus on stack-allocation garbage collection. Stack-allocation of objects is a promising approach to reduce the work of a garbage collection [4]. McDowell gives experimental evidence that there are many opportunities for stack-allocation in Java programs, and suggests that would be interesting, but does not actually allocate data on the stack[5]. It gives an analysis algorithm that can be applied to stack allocation in object oriented languages, but no implementation is mentioned. Existing stack-allocation techniques require a static analysis [6-8] at runtime with additional overheads. [9] presents a new technique for doing optimistic stack allocation of objects and describing a object stack to store nonescaped object, but it is only limited to intraprocedural analysis.

In this paper, the Java stack frame is improved to support data storage of object types, and a stack-allocation based garbage collection algorithm is presented. The stack objects are identified with expanded byte code when the compiler compiles the Java source code. Stack objects are allocated to Java stack and released immediately when they leave out their scope. Other objects are allocated to heap and released by garbage collector.

II. HEAP AND STACK IN TRADITIONAL JVM

Java heap and Java stack in modern Java virtual machine are described as following [10].

The memory for the new object is allocated from a single heap when a class instance or array is created in a running Java application. There is only one heap inside a Java virtual machine instance, all threads share it. The Java virtual machine has an instruction that allocates memory on the heap for a new object, but it has no instruction for letting that memory free. It is the virtual machine that works for deciding whether and when to free memory occupied by objects that are no longer referenced by the running application. The heap is managed by a garbage collector in Java virtual machine. A garbage collector's primary function is to automatically reclaim the memory used by objects that are no longer referenced by the running application. It may also move objects as the application runs to reduce heap fragmentation.

When a new thread is launched, the Java virtual machine creates a new Java stack for the thread. A Java stack stores a thread's state in discrete frames. The Java virtual machine only performs two operations directly on Java Stacks: pushes and pops frames. When a thread invokes in a Java method, the virtual machine creates and pushes a new frame onto the thread's Java stack. This new frame then becomes the current frame. As the method executes, it can use the frame to store parameters, local variables, intermediate computations, and other datas.

The stack frame consists of three parts: local variables, operand stack, and frame data. The sizes of the local variables and operand stack depend upon the need of each individual method. These sizes are determined at compiling time and included in the class file data for each method. When the Java virtual machine invokes in a Java method, it will check the class data to determine the number of words required by the method in the local variables and operand stack. It creates a stack frame of the proper size for the method and pushes it onto the Java stack. The local variable section of the Java stack frame is organized as a zero-based array of words. Instructions that use a value from the local variable section provide the zero-based array with an index. Values of type *int*, *float*, reference, and *returnAddress* occupy one entry in the local variables array. Values of type byte, short, and char convert to *int* before being stored into the local variables. Values of type long and double occupy two consecutive entries in the array.

978-1-4244-6542-2/$26.00 © 2010 IEEE

```
r1 = new Obj1; //global object①
{
        Obj2 r2 = new Obj2;②
        {
                Obj3 r3 = new Obj3; ③
        }
        … ④
}
… ⑤
```

Figure 1. Java program

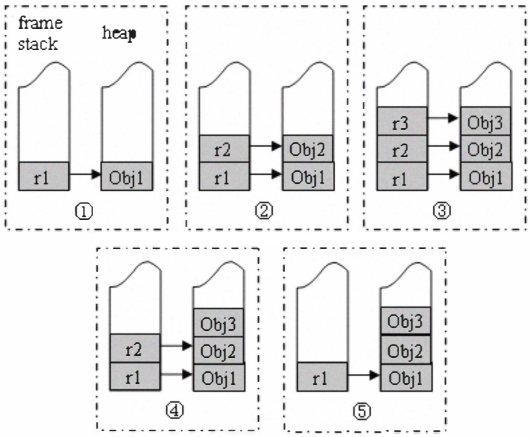

Figure 2. the model of heap and stack

Figure 1 is a section of Java program. Figure 2 is the variation of the model of heap and stack when Java virtual machine executes the Java program which is described in Figure 1.

The local variable area of Java stack frames supports data storage of basic types and reference types, but it does not support the data storage of objects types. As shown in Figure 2, memory space for a new created Java object is allocated to heap, and the memory space for reference variable which points to the Java object is allocated from Java frame. Obj2 and Obj3 have short lifetime. They are released immediately when they leave their scope. This memory allocation strategy has drawbacks: 1) Memory allocation is inefficient. 2) Memory is not reclaimed immediately. 3) More memory footprint is needed. The better solution for the problem is supporting for the object storage of Java frames which will be described in third section.

III. TACK-ALLOCATION BASED GARBAGE COLLECTION ALGORITHM

In order to achieve memory allocation of the stackable object efficiently in Java virtual machine, a stack-allocation based garbage collection algorithm is proposed, and the problem of stackable object is recognized, Java stack frame transformation, as well as memory allocation and reclamation must be solved.

A. Recognition of Java Stackable Object

Local objects could be declared, and then be used in C++. These local objects are allocated from the runtime stack. The local object's life cycle, allocation and recycling instructions are identified at compiling time. Objects memory spaces are allocated and reclaimed efficiently in this way. Although Java does not support syntax of local objects, all the objects are dynamically allocated to heap through this new instruction. All objects are not reclaimed by explicit instruction. They are reclaimed by garbage collector. Java does not support syntax of local object, but most of objects which dynamically allocated from heap have the local characteristics. They have a very short life cycle, and with first-in-last-out characteristics. All such objects could be allocated from Java stack. These objects are also called stackable objects in this paper.

One method of recognizing Java stackable objects is to expand new syntax to identify these objects by Java program language. Java programmers could control the objects' allocation flexibly, but this solution conflict with Java's Principles of encapsulation. In this paper, the Java stackable objects are identified by compiler and marked with expanded byte code *snew*. This approach is transparent to the programmer. At present, the escape analysis is a primary way for the identification of stackable objects. Escape analysis is a static analysis method to determine whether the survival time of data exceeds its static field survival time. If the object O does not escape out of the method M, then O must not escape out of the thread which created it. So the O could be allocated from stack frame of M.

B. Improvement of Java Stack Frame

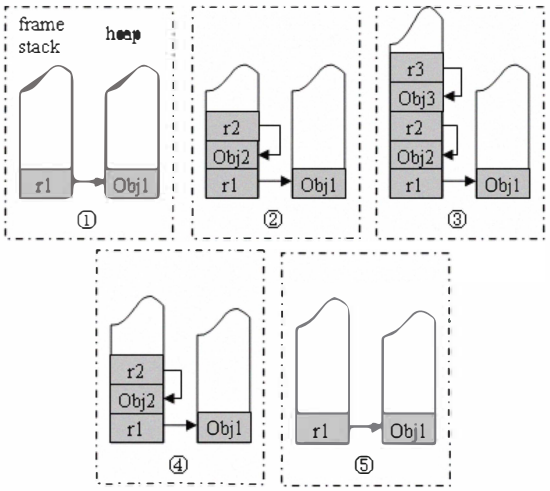

Figure 3. the improved heap and stack model

This section improve that Java stack frame not only can support data storage of basic types and reference types, but also to support data storage of object types directly. Figure 3 describes the memory variation of improved Java stack frame when it is running Java program in Figure 1. The Obj2 and Obj3 are released without explicit instruction. Compared

to memory allocation of traditional Java virtual machine in Figure 2, local objects aren't allocated to heap.

C. Memory Allocation and Reclamation

Java byte code is executed by JVM at runtime. Memory space of the following Java object is allocated to current stack frame while JVM meets the byte code instruction *snew*, and the size of the stack frame is determined at compiling time without determining whether it overflows. Memory space of the following Java object is allocated to heap space while JVM meets the byte code instruction *new*, and the Java virtual machine determines if there is enough free memory in the heap. If there is not enough free memory, garbage collector will be launched to collect unreferenced objects. Memory allocation and reclamation algorithm are shown in Figure 4.

Stack frame in stack space is reclaimed as a unit. When the thread is leaving out a method, Java virtual machine will pop up a stack frame. The reclamation of the stack frame space is determined at compiling time without explicit instructions. The next local object will reuse the memory space when objects' lifetime is end. The reclamation of heap space is the work of garbage collector. Java virtual machine will launch garbage collector when there is not enough free memory space.

```
(1)New(N) =
(2)        if N == snew
(3)                allocate from object stack
(4)        else if N == new
(5)                if free_pool is empty
(6)                        garbage_collection()
(7)                endif
(8)         allocate from heap
(9)        endif
```

Figure 4. allocation and reclaim algorithm

IV. EXPERIMENT

A. The Simulated Machine

A simulated K Virtual Machine [11] is built with the memory allocation and garbage collection subsystem. The simulated system uses 4 bytes per pointer. Each simulated object is the size of its member variables plus a 4 bytes header. The high 24 bits of header is the length of a object, and the first bit is used for mark bit of garbage collection. The object header is shown in Figure 5. The mark-sweep algorithm is used to collect the garbage where it is on the heap.

The heap is grouped with free chunks. It is just one single chunk at the beginning of the system. With the conduct of allocating and deallocating, the heap is divided into several small chunks which are connected into list. The first allocation algorithm is used to allocate heap memory for the system. The group's architecture of free heap memory is shown in figure 6.

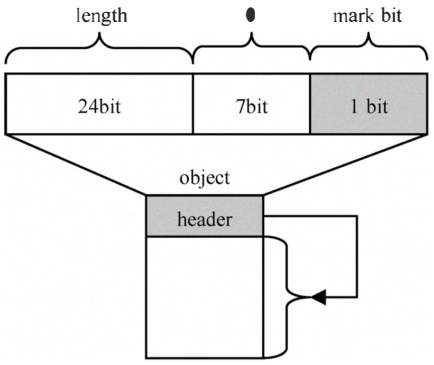

Figure 5. object herder

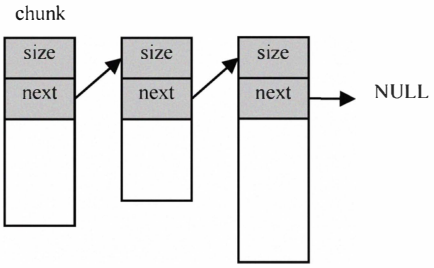

Figure 6. Free list

In order to simplify the system, the only allocation statement of the programs is taken account into. A subsystem is implemented to generate allocation statements which contain only two types. All the statements are randomly generated according to the following rules in figure 7.

① S→{LSS}
② S→ε
③ L→E
④ E→new
⑤ E→snew

Figure 7 the rule of statements

B. Result

The relationship between the number of garbage collection and percentage of stackable objects is shown in figure 8. The number of garbage collection is linear decreasing with the increasing of the percentage of stackable objects. The larger the percentage of the stackable objects, the fewer the number of garbage collection. The garbage collator becomes the traditional mark-sweep garbage collection algorithm when the percentage of stackable object is zero, and this is the worst performance of the garbage collector.

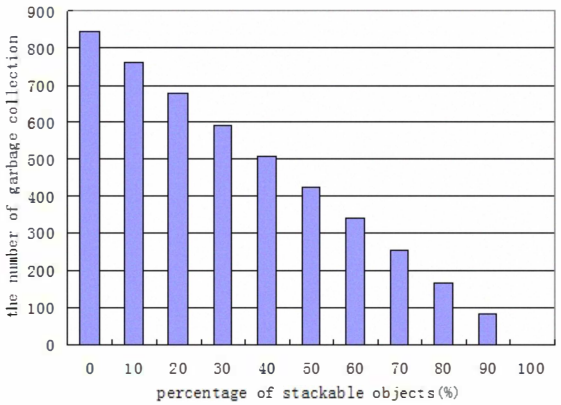

Figure 8. the number of garbage collection

The relationship of garbage collection time and percentage of stackable objects is shown in figure 9. The time of garbage collection is linear decreasing with the increasing of the percentage of stackable objects. The larger the percentage of the stackable objects, the fewer the time of garbage collection. The garbage collator becomes the traditional mark-sweep garbage collection algorithm when the proportion of stackable object is zero.

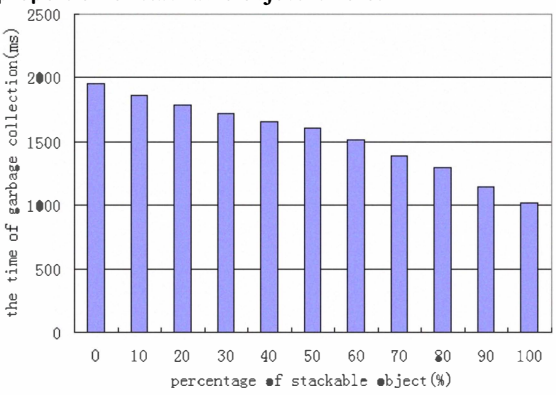

Figure 9. the time of garbage collection

In most programs, the percentage of objects that could have been allocated on a stack instead of on the heap ranged from zero to possibly as high as 56% [2].So this algorithm is effective to collect garbage.

V. CONCLUSION

A stack-allocation based algorithm for Java virtual machine garbage collection is described in this paper, and the Java stack frame is improved to support data storage of object types. The stackable objects are determined at compiling time through the source code analysis and marked with expanded instruction. They are allocated from the improved stack frame and released immediately when they leave out their scope at runtime. All the other objects are still allocated from heap and released by garbage collector. Compared with traditional garbage collection algorithm, Experimental result shows that the new algorithm optimizes the JVM'S memory allocation and reclamation, and also reduce the burden of the garbage collector.

ACKNOWLEDGMENT

This research is supported by the Special Funds for Key Program of the China No. 2009ZX01039-002-001-04, and the Science & Technology Research Program of Zhejiang Province, China No. 2007C11070.

REFERENCES

[1] J. Gosling, B. Joy, G. Steele, "The Java Language Specification", Addison Wesley, 1996.

[2] C. E. McDowell, "Reducing garbage in Java", ACM SIGPLAN Notices, September, 1998, Volume 33, Issue 9, pp. 84-86.

[3] Richard Jones, Rafael Lins, "Garbage Collection", John Wiley and Sons, Ltd., 1996.

[4] D. Gay, B. Steensgaard, "Fast Escape Analysis and Stack-allocation for Object-based Programs", In Compiler Construction, 9th International Conference (CC 2000), 2000, pp. 82-93.

[5] C. Ruggieri, T. P. Murtagh, "Annual Symposium on Principles of Programming Languages", Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1988, San Diego, California, United States, pp. 285-293.

[6] B. Blanchet, "Escape Analysis for Object Oriented languages: Application to Java", In conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99), November, 1999, pp. 20-34.

[7] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, S. P. Midkiff, "Escape Analysis for Java", In conference on Object-Oriented Programming, System, Languages and Applications (OOPSLA'99), November, 1999, pp. 1-19.

[8] J. Whaley, M. Rinard, "Compositional Pointer and Escape Analysis for Java Programs", Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 1999, Denver, Colorado, United States, pp. 187-206.

[9] Erik Corry, "Optimistic Stack Allocation For Java-Like Languages", Proceedings of the 5th international symposium on Memory management, 2006, Ottawa, Ontario, Canada , pp. 162 – 173.

[10] Bill Venners, "Inside the Java Virtual Machine", McGraw-Hill Companies, January 6, 2000.

[11] Sun Microsystems, Inc, "J2ME Building Blocks for Mobile Devices", 2000, http://java.sun.com/products/kvm/wp/KVMwp.pdf