PC Based Escape Analysis in the Java Virtual Machine

Manfred Jendrosch, Gerhard W. Dueck, Charlie Gracie, and André Hinkenjann

Abstract—Current computer architectures are multi-threaded and make use of multiple CPU cores. Most garbage collections policies for the Java Virtual Machine include a stop-the-world phase, which means that all threads are suspended. A considerable portion of the execution time of Java programs is spent in these stop-the-world garbage collections. To improve this behavior, a thread-local allocation and garbage collection that only affects single threads, has been proposed. Unfortunately, only objects that are not accessible by other threads (“do not escape”) are eligible for this kind of allocation. It is therefore necessary to reliably predict the escaping of objects. The work presented in this paper analyzes the escaping of objects based on the line of code (program counter – PC) the object was allocated at. The results show that on average 60-80% of the objects do not escape and can therefore be locally allocated.

Index Terms—Escape analysis, garbage collection, Java virtual machine, memory management.

I. INTRODUCTION

Java is an object-oriented, platform independent programming language and the most used language according to the TIOBE Programming Community Index in February 2013 [1]. The Java source code is compiled into bytecode that is then executed by the Java Virtual Machine (JVM).

The JVM acts as a layer between the running program and the operating system and provides additional functions, e.g. process, thread, and memory management. Most objects in Java are stored in a memory space called the heap and unreachable objects are reclaimed by a central Garbage Collection procedure (GC). A considerable portion of the execution time of Java programs is spent in garbage collections.

Current computer architectures are multi-threaded and make use of multiple CPU cores. Most GC policies use the stop-the-world paradigm, which means that all threads in the JVM need to be suspended during a GC. The stop-the-world phases significantly increase the overall runtime in highly multi-threaded environments. An important goal in the research for the JVM is therefore the reduction of the overall time spent in garbage collections.

One way to tackle this issue is the idea of a thread-local allocation and garbage collection, which only blocks the current thread and does not affect any other threads. Unfortunately, only objects that are not accessible by any other threads (“do not escape”) are eligible for a thread-local allocation and garbage collection. An escape analysis is therefore necessary to reliably predict the escaping of objects.

The escape analysis presented in this paper is based on the Program Counter (PC). That means that the analysis is performed based on where in the program the object is allocated. The hypothesis is, that objects allocated at a specific place in the program will exhibit similar behavior.

A. Background

1) Garbage collection

Most garbage collection policies involve the following three basic parts:

1) Garbage detection: Identify all live and dead objects.
2) Reclamation of memory: After all dead objects are clearly identified, the memory of these objects can be reclaimed.
3) Defragmentation: All live objects are compacted to a contiguous space.

Not every garbage collector performs the defragmentation step. Some garbage collectors combine steps 2 and 3 in one traversal.

2) Escape analysis

An object that is allocated by one thread and later is referred to from another thread is said to be escaped. Escape analysis can be performed in two different ways:

- Static analysis: Identify objects eligible for a thread-local allocation during compile time.
- Runtime analysis: Identify eligible objects during runtime.

B. Problem

As already mentioned in the introduction, a considerable portion of the execution time of Java programs is spent in garbage collections. Garbage collections can be triggered at fixed time slots or whenever free memory falls below a given threshold. Unfortunately most garbage collectors perform stop-the-world global collections, so that all threads need to be suspended during the GC. This issue is even more detrimental in multi-threaded environments where many threads are suspended, even though they might have enough memory to continue execution.

This garbage collection behavior makes the utilization of Java for real-time applications difficult, because any garbage collection will stop the whole execution of the programs. Interval and duration of these garbage collections are not deterministic, so that most Java garbage collectors cannot be used to fulfill tough real-time requirements.
C. Thread-Local Memory Management

The idea to solve these issues is a thread-local allocation and garbage collection that does not affect any other thread during a GC. The allocation of as many objects as possible in a thread-local memory area would result in the following improvements:

- **Implicit GC:** The thread-local memory area can be reused after the associated thread dies.
- **Local GC:** When the local memory area is full, a local GC can be performed that will only impact the current thread.
- **Less effort for global GC:** Due to the fact that more objects are allocated in the thread-local memory area, fewer objects need to be allocated on the global heap. This will result in shorter processing time for each garbage collection run and fewer GC invocations overall.
- **Less fragmentation on the global heap:** Objects with a short lifecycle – that often cause fragmentation – might be eligible for a thread-local allocation, so that the global heap may become less fragmented.

The realization of a thread-local allocation and garbage collection for all objects would create a potential performance boost, but unfortunately not all objects are eligible for thread-local allocation. Due to the fact that the associated thread can only access a thread-local area, only non-escaping objects qualify for this kind of allocation. A thread-local allocation and garbage collection can therefore only work well, if the prediction of the escaping of an object has a high degree of accuracy. If the prediction is wrong, the object needs to be copied back to the global heap and all reference pointers must be updated. This procedure will probably need more time than the time saved with a thread-local allocation and garbage collection. The foundation for a reliable and effective thread-local garbage collection is therefore a highly accurate escape prediction.

The escape analysis presented in this paper is a runtime analysis and it is based on the line of code (program counter – PC) the object was allocated at. This metric allows a greater insight into the escaping of objects than e.g. a class based escape analysis. The runtime analysis provides the ability to collect statistics about the escaping of objects and dynamically initiate an appropriate escape handling. A static escape analysis otherwise is limited to the analysis and modification of the bytecode at compile time and therefore, needs to be conservative when it cannot determine with certainty if an object will escape or not.

II. RELATED WORK

Some researchers have performed escape analysis in the JVM. Some of these approaches are briefly introduced in this section.

Jones and King [2] use a snapshot escape analysis that takes a snapshot of the whole system when enough classes are loaded by the JVM and perform a static analysis of these classes. The objects are categorized into the three classes: never escaping (local), maybe escaping (optimistically local) and always escaping (global). The bytecode is then modified to allocate the objects according to their classification.

Steensgaard [3] utilizes a static escape analysis at compile time. Whenever an object is determined as eligible for a thread-local allocation, the new() allocation instruction is replaced by a threadNew() instruction in the bytecode. The modified JVM can then allocate all threadNew() objects in the thread-local regions while the objects allocated with the classic new() instruction are still allocated in the shared heap as usual.

Marlow and Jones [4] allow references from global to local objects protected by a read barrier and therefore do not really perform a classic escape analysis. Whenever the read barrier notices that a foreign object is accessed, the owner of this object is advised to copy it out to the shared heap before the execution can be continued.

Domani et al. [5] perform a runtime escape analysis that has a higher accuracy than a static escape analysis. Instead of analyzing all objects during compile time and modifying the corresponding bytecode, Domani’s approach utilizes a write barrier to detect and analyze the escaping of objects. This write barrier is triggered whenever a reference is created from one object to another. When the new reference points to a non-escaped object allocated by another thread, this object is marked as escaped. With this approach, the objects are only marked as escaped when they really escape. With a static analysis, objects might be categorized as potentially escape or guaranteed to escape from the start, although the actual escaping is happening at the end of the program’s execution.

The PC based escape analysis presented in this paper uses the same approach as Domani et al. in terms of the write barrier escape marking, but analyzes the escaping of objects not only globally. Instead the escape analysis is broken down to every single PC value and can therefore achieve high accuracy.

III. INSTRUMENTATION OF THE JVM

The Java Virtual Machine had to be instrumented to enable an escape analysis based on the program counter. This instrumentation is introduced in this section, details are provided in the thesis this paper is based on [6].

A. Extension of the VMOBject

Each object in the Java Virtual Machine is internally represented by a VMOBject that holds general information about the object, e.g. its class and certain kinds of flags. For the escape analysis, an object needs to keep track of its allocation thread and the program counter it was allocated from. Unfortunately, the VMOBject does not provide the native possibility to store this data, so that it needs to be extended as shown in Fig. 1. The new fields allocationThreadAndFlags and allocationProgramCounter are initialized with the corresponding data at the allocation of an object.

```
+ allocationThreadAndFlags: VMTThread
+ allocationProgramCounter: Integer
+ TLGC_OBJECT_GET_ALLOC_THREAD(object)
+ TLGC_OBJECT_GET_ALLOC_PROGRAM_COUNTER(object)
+ TLGC_OBJECT_SET_ESCAPED(object)
+ TLGC_OBJECT_GET_FLAGS(object)
+ TLGC_OBJECT_GET_TRANSITION(object)
+ TLGC_OBJECT_SET_ESCAPED(object)
+ TLGC_OBJECT_GET_TRANSITION(object)
+ TLGC_OBJECT_GET_FLAGS(object)
+ TLGC_OBJECT_SET_TRANSITION(object)
```

Fig. 1. Extension of the VMOBject.

To correctly identify and count the escaping of objects, the
The following boolean flags are stored per object:
- **Escaped**: Indicates, if the object is escaped or not.
- **Seen**: Indicates, if the object was seen during garbage collection.
- **Transition**: Special bit needed to correctly handle the transition from non-escaping objects into the escaping state.

Due to the fact that the **VMOBJECT** is the central object class in the JVM, it is important to keep its memory footprint small. Since the pointer to a thread has an empty byte, this can be used to store the three flags mentioned above without using additional memory.

**B. Escape Marking**

The extensions made to the **VMOBJECT** enable the marking of objects as escaped during the runtime of a Java program. To be sure that the escaping of every object is recognized, the creation and update of each reference in the JVM needs to be intercepted in the write barrier.

Fig. 2 shows the sequence diagram of the added write barrier checks. For every `objectStore()` and `arrayObjectStore()` additional methods are hooked in, to analyze the source and destination objects of the reference that will be created.

An object may escape in one of the following ways:
1) An object that was allocated by another thread references the object.
2) An object that has already been marked as escaped references the object.
3) A reference to the object is stored into a VM structure that is globally accessible.

Whenever the write barrier check detects one of these three scenarios, it triggers the `tlrgcMarkAsEscaped()` method. This method marks the transitive closure of the object as escaped, because all objects reachable from the now escaping object need to be considered as escaped as well. Fig. 3 and Fig. 4 illustrate the recursive escape marking with an example object structure.

Additionally to the three categories mentioned above, special cases exist, where objects are guaranteed to escape, so that they can directly be marked as escaped at allocation time:
- **Classes**: Each class object is only held once per JVM, so that they are accessible by all threads.
- **Static Variables**: Class static variables are accessible by all threads.
- **Interned Strings**: The JVM reuses interned strings and char arrays, so that these objects need to be marked as escaped as well.
- **JNI Global References**: Objects that are made available via JNI are globally accessible.

**C. Collecting Data**

After all objects have been successfully marked as escaped at the write barrier, a suitable point to collect and aggregate the escape data needs to be found. The JVM provides the methods `preCollect()` and `postCollect()` which are executed just before and after a single garbage collection run. The `preCollect()` is the ideal place to hook in the escape analysis code. Fig. 5 shows the hook in a sequence diagram.

The method `tlrgcEscapeAnalysisPreCollect()` iterates all objects on the heap and aggregates the escaping of the objects globally and more fine grained based on the program counter. The aggregated values are kept for each GC and accumulated over the whole runtime of the program.
IV. ANALYSIS OF THE RESULTS

To obtain reliable results, the following tests and benchmarks were executed with the modified JVM to collect and aggregate escape analysis data.

- **Simple Test**: A simple Java test class that explicitly triggers two garbage collections via System.gc(). This test was mainly used for debugging the escape analysis instrumentation code.
- **Test Different Objects**: This test case was created to verify the correct determination of the program counter and associated line of code in the PC based escape data. It allocates different kinds of objects and explicitly triggers multiple GCs.
- **Eclipse Normal**: Start of the Eclipse IDE for C/C++ Developers in the Indigo release version [7] with the normal workspace of the research project with over 80 opened projects.
- **SPECjbb2005**: This benchmark evaluates the performance of the server side part of Java by emulating a three-tier client/server application [8].
- **SPECjvm2008**: This benchmark measures the performance of the JRE by executing several real life applications and benchmarks. In combination of all those tests, the key duty of SPECjvm2008 is to exercise the Java core functionality [9].

The following sections will present the global and PC based escape results.

A. **Global Escape Results**

The tests that only include a few garbage collections and are basically displaying the JVM startup (SimpleTest, TestDifferentObjects and EclipseEmpty) show that during the startup more than 50% of the objects may escape (as depicted in Fig. 6). These three tests are excluded from the further evaluation, because they are not really representative for typical Java applications.

The other three benchmarks (EclipseNormal, SPECjbb2005, and SPECjvm2008) show that on average 20-40% of the objects escape. This means that 60-80% of the objects do not escape and are therefore eligible for thread-local allocation and garbage collection.

B. **PC Based Escape Results**

Fig. 7 shows the aggregated PC Based Escape Results for EclipseNormal, SPECjbb2005 and SPECjvm2008. These results are very encouraging, because the diagram indicates that the escaping per program counter can be split into the following categories:

1. **Almost never escape**: less than 10% of the objects allocated at a given PC escape.
2. **Unpredictable escaping**: between 10% and 90% of the objects allocated at a given PC escape.
3. **Almost always escape**: more than 90% of the objects allocated at a given PC escape.

A very interesting fact is, that the PC based escape analysis can identify most objects either as almost never or almost always escape, which makes an escape prediction reliable. Additionally, 60-80% of the total objects are allocated from a program counter that never escapes. Therefore, the thread cannot only safely handle these objects, but more importantly, it can be accurately predicted that they will not escape.

Further details about the PC based escape analysis can be found in [6].

![Fig. 7. PC based escape results – compared.](image)

V. CONCLUSION

The results obtained from the PC based escape analysis are promising and show that a high percentage of objects never escape, so that they can be allocated thread-locally without penalty. Additionally, the PC based approach could successfully identify program counters with high escape and non-escape percentages, which enables a reliable prediction of the escaping of objects and is a good base for the decision if objects should be thread-locally allocated and garbage collected. This work has laid the basis for a thread-local memory management.

REFERENCES


Manfred Jendrosch received his bachelor degree in computer science from the Bonn-Rhein-Sieg University of Applied Sciences in 2008 and his Dual Master degree from the University of New Brunswick and the Bonn-Rhein-Sieg University of Applied Sciences in 2013. He is currently working for the Smart Grid Division at Siemens Canada Ltd. in Fredericton, New Brunswick, Canada.

Gerhard W. Dueck received the BSc, master, and PhD degrees in computer science from the University of Manitoba, Winnipeg, Manitoba, Canada, in 1983, 1986, and 1988, respectively. He is currently a professor in the Faculty of Computer Science at the University of New Brunswick. His research interests include dynamic memory management for the Java Virtual machine, reversible logic, Reed Muller expansions, multiple-valued logic, and digital design.

Charlie Gracie received his bachelor degree in computer science the University of New Brunswick. He is currently working as a Garbage Collection Technology Architect at IBM in Ottawa, Canada. His research interest include optimization of the Java Virtual Machine, program performance analysis, software development, and software scalability.

André Hinkenjann is a research professor at the Department of Computer Science of the Bonn-Rhein-Sieg University of Applied Sciences in Sankt Augustin, Germany. He is the founding director of the Institute of Visual Computing, where he heads a group doing research in the areas of interactive environments and high performance, high quality graphics. His research interests include computer graphics, GPGPU and parallel, high performance rendering. Since 2010 he is also an adjunct professor at the University of New Brunswick, Fredericton, Canada.