PC Based Escape Analysis in the Java Virtual Machine

Manfred Jendrosch [1,3], Gerhard W. Dueck [1],
Charlie Gracie [2], André Hinkenjann [3]
[1] University of New Brunswick – Faculty of Computer Science, [2] IBM Canada,
[3] Bonn-Rhein-Sieg University of Applied Sciences – Faculty of Computer Science
manfred.jendrosch@unb.ca, gdueck@unb.ca, charlie_gracie@ca.ibm.com,
andre.hinkenjann@h-brs.de

MOTIVATION

- **Introduction:** The Java Virtual Machine (JVM) executes Java programs and provides additional management functions, e.g. Garbage Collection (GC). A considerable portion of the execution time is spent in garbage collections.
- **Problem:** 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. Parallelism cannot be exploited in these stop-the-world phases.
- **Goal:** Reduce the time spent in garbage collections.
- **Idea:** Having a thread-local garbage collection that only blocks the current thread and does not affect any other threads.
- **Limitation:** Only objects that are not accessible by any other thread *(“do not escape”)* are eligible for a thread-local allocation and garbage collection.
- **Conclusion:** An ***escape analysis*** is necessary to reliably predict the escaping of objects.

BACKGROUND

- **Heap:** All objects created with the *new()* operator are stored on the heap.
- **Garbage Collection:** The JVM handles the deallocation of dead objects with a central procedure called *Garbage Collection.* A single garbage collection contains the following three steps:
  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:** Compact all live objects to a contiguous space.

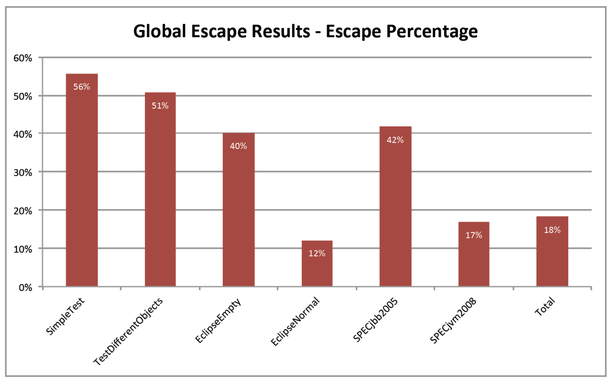
The defragmentation is not performed by every garbage collector. Some garbage collectors even combine step 2 and 3 in one traversal.

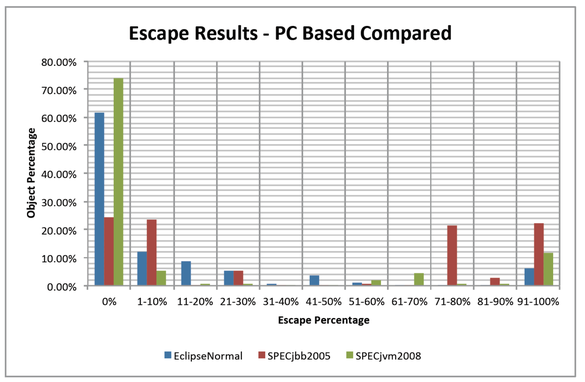
INSTRUMENTATION OF THE JVM

The gathering of data about the escaping of an object was performed based on the line of code (program counter – PC) the object was allocated from. The JVM has been extended as follows to enable a PC based escape analysis:

1. **Extension of the VMObject:** Additional fields were added to store the allocation thread and program counter for each object.
2. **Escape Marking:** As soon as a reference to a non-escaped object of another thread is created, this object and all its descendants need to be recursively marked as escaped. This escape marking procedure was inserted into the write barrier that is triggered at every creation of a reference.
3. **Collecting Data:** Each garbage collection precedes a call to the *preCollect()* method. This method has been extended to traverse all objects on the heap and read and aggregate their escape data based on the program counter.

ESCAPE ANALYSIS RESULTS





- **Reliable Escape Prediction:** The escape analysis based on the program counter can reliably predict the escaping of objects.
- **Global Results:** The global escape results show that on average 20-40% of the objects escape.
- **PC Based Results:** The PC based results show that a high percentage of the objects are allocated from program counters that almost never escape (≤ 10% escape percentage). Additionally, 60-80% of the total objects are allocated from a program counter that never escapes.
- **Conclusion**: The results show that a PC based escape analysis is a good base to decide if objects should be thread-locally allocated and garbage collected.