Improving Stack Allocation on Eclipse OpenJ9 using Precise Static Analysis

Nikhil T R
IIT Mandi, India
b16066@students.iitmandi.ac.in

Dheeraj
IIT Mandi, India
b17041@students.iitmandi.ac.in

Swapnil Rustagi
IIT Mandi, India
b17104@students.iitmandi.ac.in

Arjun Bharat
IIT Madras, India
cs17b006@smail.iitm.ac.in

Vijay Sundaresan
IBM Canada
vijaysun@ca.ibm.com

Andrew Craik
IBM Canada
ajcraik@ca.ibm.com

Daryl Maier
IBM Canada
maier@ca.ibm.com

Manas Thakur
IIT Mandi, India
manasthakur@iitmandi.ac.in

V. Krishna Nandivada
IIT Madras, India
nvk@iitm.ac.in

1 Abstract

Modern object-oriented programming languages such as Java and C# have a managed runtime. The idiomatic way to write programs in Java is to organize computation in classes, which are then instantiated to create objects allocated on the heap memory. These objects interact with each other through methods, and are deallocated post usage by a garbage collector. While the usage of objects gives higher-level programming abstractions, it also leads to multiple performance overheads. We focus on two such overheads: (i) accessing the fields of objects on the heap can be more expensive (in the worst case requires two memory accesses) than stack accesses (requires at most one memory access); and (ii) memory management (garbage collection) overheads. Thus, larger the number of objects allocated on the heap, higher may be the overheads.

In this context, an important compiler optimization performed in context of languages with a managed runtime is *stack allocation*: If an object can be allocated on the stack frame of a method, the occupied memory is automatically freed as soon as the method finishes its execution during runtime. A popular way to identify objects that can be allocated on the stack frame of a method is by performing *escape analysis* [1], which essentially checks if a given object is local to the method in which it is allocated.

As an example of the possible scenarios in which objects can be allocated on the stack, consider the Java code snippet shown in Figure 1. Say the abstract object(s) allocated at line l are represented as O_l. Here, the object O_5 is clearly local to the method `foo` and can be allocated on its stack. Further, if we analyze the method `bar`, it can be deduced that the object O_4 can also be allocated on the stack of `foo`. Object O_6 becomes reachable from a static field (g) and hence in general should remain on heap; however, if found useful, it can still be allocated on `foo`'s stack till the call to `bar` at line 8. Such possibilities of allocating a large number of objects on stack are usually prevalent even in real world code; however, finding them requires performing precise (in this case, interprocedural and flow-sensitive) escape analysis, which consumes time and memory, and is in general expensive. Such a precise analysis can be prohibitive in the context of managed runtimes.

```
1  class C {
2    static D g;
3    void foo() {
4      D x = new D();
5      D y = new D();
6      x.f = new D();
7      // y used but doesn't escape
8      bar(x); }
9    void bar(D p) {
10     g = p.f;
11   }
12 } /*class C*/
13 class D { D f; }
```

Figure 1. A Java code snippet to demonstrate possibilities of stack allocation.

In order to balance the tradeoff between the precision of program analysis and the time taken during JIT compilation, traditional just-in-time (JIT) compilers in popular JVMs, such as Testarossa and C2 in Eclipse OpenJ9 [2] and Oracle HotSpot [3], respectively, perform conservative intraprocedural analyses on the method being compiled. Consequently, existing JIT compilers will be able to stack-allocate only the object O_5 for the toy code shown in Figure 1 (ignoring the possibilities of inlining).

Thakur and Nandivada recently proposed a framework called PYE [4] as a way to use the results of precise static analyses during JIT compilation. PYE encodes the dependences between various changeable parts of a program as *conditional values*, and resolves the associated conditions during JIT compilation. Among other optimizations, PYE was shown to improve the number of synchronized operations that could be elided using a precise *thread-escape analysis*.

AORCPT '20, November 10–13, 2020, Virtual Event
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In this project, we are working on extending the idea of splitting the analysis between static and JIT compilation from PYE to enable the stack allocation of a larger number of objects compared to the existing imprecise escape analysis in Eclipse OpenJ9. In the proposed talk, we would discuss our static escape analysis, an optimistic scheme that allocates objects on the stack during JIT compilation based on hints obtained from static analysis, and a planned scheme to split the range in which a given object (or some fields of it) stays on the stack versus the heap. We would also present some initial results that show a good potential for improvement.

About the Speaker

Manas Thakur is an Assistant Professor at Indian Institute of Technology Mandi, India. His research interests include program analysis and compiler optimizations.

References

[1] Bruno Blanchet. 2003. Escape Analysis for JavaTM: Theory and Practice. *ACM Trans. Program. Lang. Syst.* 25, 6 (Nov. 2003), 713–775. DOI: http://dx.doi.org/10.1145/945885.945886

[2] Eclipse OpenJ9. 2020. The Eclipse OpenJ9 Virtual Machine. https://www.eclipse.org/openj9/. (2020).

[3] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot^{TM} Server Compiler. In *Proceedings of the 2001 Symposium on Java^{TM} Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1267847.1267848

[4] Manas Thakur and V. Krishna Nandivada. 2019. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 16 (July 2019), 37 pages. DOI: http://dx.doi.org/10.1145/3337794