Dynamic Monitor Allocation in the Java Virtual Machine

Marcel Dombrowski, Kenneth B. Kent, Michael Dawson
University of New Brunswick, IBM Canada
Faculty of Computer Science
marcel.dombrowski@unb.ca, ken@unb.ca, michael_dawson@ca.ibm.com

Outline

•Changing static monitor allocation in the JVM to a dynamic approach.
•New approach is faster than the reference JVM and consumes less memory.

Motivation

The Java Virtual Machine (JVM) relies on space and time efficient programming as it is the bottleneck for all Java user programs. Therefore, research on more efficient algorithms needs to be conducted in order to approach this goal.
Synchronization across threads in the JVM is done using locks. According to the Java Language specification every object needs to be able to be used for locking. As not every object is actually used for locking this can lead to a significant memory overhead and a performance hit.
Research has been conducted into thinning out the lock to reduce the memory footprint. [1] introduces a technique to reduce locking to a single compare-and-swap operation. Onodera showed in [2] a hybrid approach of using lightweight locks and, if contention occurs, inflating these lockwords to heavyweight monitors. These reduce the fixed overhead per object to a single word while at the same time achieving good performance.

[1] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, “Thin locks,” ACM SIGPLAN Notices, vol. 33, pp. 258–268, May 1998.
[2] T. Onodera, “A study of locking objects with bimodal fields,” ACM SIGPLAN Notices, 1999.

Background

Java is one of the most popular programming languages (Ranked #2 in the TIOBE Index, November 2012). It has been designed as an object-oriented programming language with a garbage collection component that handles memory allocation/deallocation of objects. Java has been designed to be operating system and machine independent. This behaviour is achieved by encapsulating all programs in a Virtual Machine, the JVM.
This JVM enables the programs to run in a sandboxed environment, with which additional security can be introduced. Java features a hybrid execution approach: code can run interpreted and compiled. The Just-in-Time compiler (JIT) compiles bytecode into native code which then runs machine dependant.

Problem

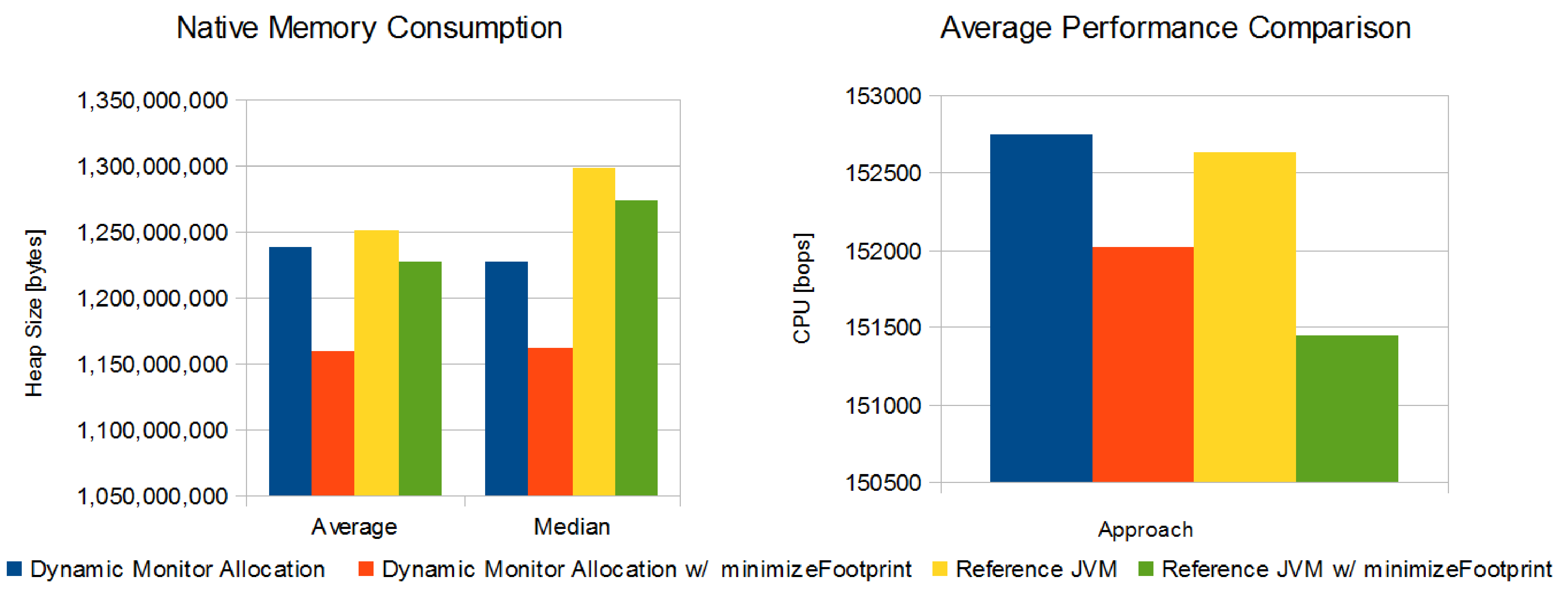
As of now every object in the IBM JVM receives a lockword, even if the object is never used for locking. This behaviour introduces a memory overhead and research needs to be conducted into reducing the memory footprint of the object model.
Previous approaches focused on thinning out the object header, by e.g. reducing the lockword size and locking algorithm. But nevertheless every object still received a lockword.

Solution

•Removal of the lockword embodied in the object (it can be decided which objects receive lockwords).
•If an object without a lockword is used for locking, a dynamic lockword is created for this object,
  •by moving the object to a location which offers enough space for the object to also hold the lockword,
  •and instantiating the lockword, so that it is ready to use by the JVM.
•The Java Object Model has been adapted to be aware of the possible locations of the lockword.

Results

Testing with the SPECjbb2005 benchmark yielded a performance increase of 0.47% compared to the reference implementation, as well as a memory decrease of 5.51%. The reference JVM features an option which removes lockwords for most objects, but then uses more heavyweight monitors which are computationally expensive. Our approach gave a performance increase of 0.86% compared to this JVM, with a memory decrease of 3.75%. Our approach with the lockword removal for most objects gave a performance increase of 0.78% and a memory decrease of 8.79% over the reference JVM with the same options.



UNB

IBM Centre for Advanced Studies - Atlantic

FACULTY OF COMPUTER SCIENCE