A Java Compiler for Many Memory Models – extended abstract

Samuel P. Midkiff
IBM TJ Watson Research

Jaejin Lee
Dept. of Computer Science and Engineering, Michigan State University

David A. Padua
Dept. of Computer Science, University of Illinois at Urbana-Champaign

1 Introduction

The issue of memory models for languages like Java is both extremely important and poorly understood. The memory model is important because the set of legal outcomes of the program is intimately tied to the definition of the memory model. That these issues have been poorly understood is initially surprising, but less so when one considers that Java is the first widely used language that both supports explicit parallelism, and attempts to define the allowable outcomes of a program, even in the presence of data races. Moreover, because Java attempts to support a simple programming model, and to aid the development of correct, secure programs, the memory model itself should be easy to understand and as transparent as possible to the implications of programming idioms. It is widely believed that the current Java Memory Model has failed to deliver on these promises and has been called fatally flawed [5].

Prof. Pugh, with the cooperation of the many contributors on the memory model mailing list [6], is developing a new memory model for Java. It is hoped that this memory model, along with various programming idioms, will overcome the many deficiencies of the current memory model. This approach has the key benefit that it fits in well with both the current structure of JVM's and compilers, wherein the memory model is a fixed attribute of the programming language; and with the current Java standards process; and is therefore the only practical way to overcome the current problems in the current environment. It has the defect that by specifying a fixed memory model as part of the language standard, it requires the definition of the language and JVM to be changed in order to implement it. Just as importantly, it requires the definition of the language and JVM to be changed whenever a severe problem in the latest memory model is encountered – is an expensive and time consuming process.

The approach we will discuss in our presentation, and in the remainder of this abstract, is radically different. Rather than having a fixed memory model for the language, we allow the memory model to be a property of a class. This has two distinct benefits: First, prototyping new memory models to find those with good, overall properties is possible using a common compiler and JVM infrastructure; and second, the memory model can be changed to meet the needs of a particular application without redefining the language. For example, for some applications it might be desirable to facilitate ease of programming at the expense of performance by picking a sequentially consistent memory model.

2 The Architecture of our Compiler

The compiler, which extends the the Jalapeño system [1], accepts a `.class` file annotated with a memory model specification, and produces an optimized form of the program annotated with information about the orderings of memory operations that must be preserved for the execution to conform to the specified

memory model. The compiler then produces an executable that maps this intermediate representation onto a (possibly relaxed) underlying hardware consistency model.

Our compiler represents the program using the Concurrent Static Single Assignment (*CSSA*) form[4]. When the `.class` file is put into an intermediate language form, the CSSA graph will have all orderings implied by the consistency model for that class. Escape analysis[3] can be applied to the program to determine what variables are actually shared by two or more threads, and whose accesses must be ordered by the requirements of the memory model. An incremental *critical cycle analysis* will be used to refine the CSSA graph so that orderings implied by the consistency model, but not required in the current execution, can be relaxed. As was shown in [7], the memory model implied orderings that *must* be preserved in a thread are affected by the order of accesses of shared variables in other threads, and the relaxing of orders that are implied by the memory model that are not necessary to be enforced will be (conservatively) uncovered by our incremental critical cycle analysis. Next, the program represented by the CSSA graph will be optimized using techniques similar to those in [4], which describes the application of classical optimizations to explicitly parallel programs. Finally, an efficient mapping of orderings still imposed by the memory model onto the underlying hardware consistency model will be done, using extensions of the results presented in [2].

The new aspects of this work are 1) the development of a compilation system that supports programmable memory models; 2) internal representations of programs within a compiler that accommodate different memory models; 3) the application of delay set analysis to non-sequentially consistent programs; 4) an implementation and development of classical optimizations targeting explicitly parallel programs; and 5) the implementation within a Java compiler, and development of new optimizations for mapping a programming language consistency model onto a hardware consistency model. The outcome of this work will be 1) a tool for the prototyping of memory models; 2) a tool for developing languages with programmable memory models; 3) research into the relative efficiency of different memory models using a common suite of optimizations and a common hardware targets.

References

[1] Bowen Alpern, C. R. Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn-Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] Jaejin Lee David Padua. Hiding relaxed memory consistency with compilers. In *The IEEE Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2000.

[3] J.-D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings ACM 1999 Conference on Object-Oriented Programming Systems (OOPSLA '99)*, pages 1–19, Nov. 1999.

[4] J. Lee, D. Padua, and S.P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1999.

[5] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(1):1–11, 2000.

[6] Java memory model mailing list archive, October 1999. `http://www.cs.umd.edu/ pugh/java/memoryModel/archive`.

[7] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.