Elena Koval
O. O. Boyko, research supervisor
O. V. Syanova, language adviser
SHEI “National Mining University”, Dnipropetrovsk

Lock Inference for Java

Atomicity is an important property for concurrent software, as it provides a stronger guarantee against errors caused by unanticipated thread interactions than race-freedom does. However, concurrency control in general is tricky to get right because current techniques are too low-level and error-prone. With the introduction of multicore processors, the problems are compounded. Consequently, a new software abstraction is gaining popularity to take care of concurrency control and the enforcing of atomicity properties, called atomic sections.

One possible implementation of their semantics is to acquire a global lock upon entry to each atomic section, ensuring that they execute in mutual exclusion. However, this cripples concurrency, as non-interfering atomic sections cannot run in parallel. Transactional memory is another automated technique for providing atomicity, but relies on the ability to rollback conflicting atomic sections and thus places restrictions on the use of irreversible operations, such as I/O and system calls, or serialises all sections that use such features. Therefore, from a language designer's point of view, the challenge is to implement atomic sections without compromising performance or expressivity.

This thesis explores the technique of lock inference, which infers a set of locks for each atomic section, while attempting to balance the requirements of maximal concurrency, minimal locking overhead and freedom from deadlock. We focus on lock-inference techniques for tackling large Java programs that make use of mature libraries. This improves upon existing work, which either ignores libraries, requires library implementors to annotate which locks to take, or only considers accesses performed up to one-level deep in library call chains. As a result, each of these prior approaches may result in atomicity violations. This is a problem because even simple uses of I/O in Java programs can involve large amounts of library code. Our approach is the first to analyse library methods in full and thus able to soundly handle atomic sections involving complicated real-world side effects, while still permitting atomic sections to run concurrently in cases where their lock sets are disjoint.

To validate our claims, we have implemented our techniques in Lockguard, a fully automatic tool that translates Java bytecode containing atomic sections to an equivalent program that uses locks instead. We show that our techniques scale well and despite protecting all library accesses, we obtain performance comparable to the original locking policy of our benchmarks.