Science of Computer Programming 98 (2015) 735–763

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Model checking of concurrent programs with static analysis of field accesses

Pavel Parízek [a,*,1], Ondřej Lhoták [b,*]

[a] Department of Distributed and Dependable Systems, Charles University in Prague, Czech Republic
[b] David R. Cheriton School of Computer Science, University of Waterloo, Canada

ARTICLE INFO

Article history:
Received 31 July 2013
Received in revised form 24 May 2014
Accepted 15 October 2014
Available online 28 October 2014

Keywords:
Software model checking
Concurrency
Field accesses
Immutability
Static analysis

ABSTRACT

Systematic exploration of all possible thread interleavings is a popular approach to detect errors in multi-threaded programs. A common strategy is to use a partial order reduction technique and perform a non-deterministic thread scheduling choice only when the next instruction to be executed may possibly read or modify the global state. However, some verification frameworks and software model checkers, including Java Pathfinder (JPF), construct the program state space on-the-fly during traversal. The partial order reduction technique built into such a tool can use only the information available in the current state to determine whether the execution of a given instruction is globally-relevant. For example, the reduction technique has to make a thread choice at every field access on a heap object reachable from multiple threads, even in the case of fields that are really accessed only by a single thread during program execution, because it does not have any information about what may happen in the future after a particular state. These conservative decisions cause many redundant thread choices.

We propose static analyses that identify globally-relevant field accesses more precisely. For each program state, the analyses give information about field accesses that may occur in the future after the given state. The state space traversal algorithm can use this information to soundly avoid creating unnecessary thread choices, and thus to reduce the number of thread interleavings that must be explored to cover all distinct behaviors of the given program. We implemented the proposed analyses using WALA and integrated them with JPF. Results of experiments on several Java programs show that the static analyses greatly improve the performance and scalability of JPF. In particular, it is now possible to check more complex programs than before in reasonable time.

1. Introduction

Automated techniques for detecting concurrency errors in multi-threaded programs are becoming very important with the proliferation of multi-core architectures. The biggest challenge is to design techniques that have good performance and scale to large programs, so that they are practically useful.

One group of techniques is based on systematic state space traversal. Their goal is to explore all reachable states of the program under all possible thread interleavings to find property violations (errors). The number of possible thread

* Corresponding authors.
E-mail addresses: parizek@d3s.mff.cuni.cz (P. Parízek), olhotak@uwaterloo.ca (O. Lhoták).
[1] Part of this work was done while the author was at the University of Waterloo.

http://dx.doi.org/10.1016/j.scico.2014.10.008
0167-6423/

interleavings is huge even for small programs with a few threads, but many optimizations have been developed. The most prominent of them is partial order reduction (POR) [16].

The key idea of POR is to create a thread scheduling choice (thread context switch) at a point in the program execution only when the instruction to be executed next may read or modify the global state shared by multiple threads. We call the execution of the instruction *globally-relevant* in this case, and *thread-local* otherwise. We distinguish between a static instruction in the program code and its execution in a particular program state. A single static instruction can have both globally-relevant and thread-local executions in different program states. Globally-relevant instruction executions represent interaction between threads. Instructions whose execution may be globally-relevant include thread synchronization operations and accesses to objects shared between multiple threads. The effects of a thread-local instruction execution are invisible to and independent of the concurrent behavior of other threads. Thread interleavings that differ only in the scheduling of thread-local instruction executions yield the same observable behavior. Therefore, no thread scheduling choice is needed when the next instruction execution on the current thread is thread-local. To cover every possible observable behavior of the given program, it is sufficient to explore all thread interleavings that differ in the order of globally-relevant instruction executions.

More specifically, the state space traversal with partial order reduction works as follows. At any point in the program's execution, one thread is running, and some subset of other threads are ready to run. Let i be the next instruction to be executed on the currently running thread. In order to explore all thread interleavings, a verification tool would have to explore the interleaving where i is executed next, and also the interleavings in which actions of other threads occur before i. Typically, the tool starts with the state space path in which the current thread continues and executes i, and later backtracks to the current state and switches to each of the other ready threads in turn to allow them to run before executing i. However, suppose that the execution of i is thread-local. In that case, the program behavior is independent of whether i is executed before or after other threads have been allowed to run, and therefore it is sufficient to execute i first and not consider switching to other threads at this point. As a result, the sequences of executed instructions can be divided into blocks such that execution of all but the first instruction in each block is thread-local. The state space is then explored by treating each block as an atomic step. That is, all interleavings of such blocks are explored, rather than interleavings at the granularity of individual instructions. This way, if the verification tool runs to completion, it is guaranteed that the behaviors of all interleavings of executed instructions have been explored, because the observable program behavior depends only on thread switches immediately before the execution of the first instruction of each block.

Nevertheless, despite partial order reduction and many other optimizations, systematic checking of all thread interleavings with distinct behaviors is still very time-consuming and is thus currently applicable only to relatively small programs. A specific challenge is to determine, for each instruction used in the program code, which of its executions are globally-relevant and which are thread-local. There exist both static and dynamic approaches (see, for example, [13,15,16]) that provide conservative approximations of different precision. The efficiency of state space traversal depends crucially on precisely identifying as many thread-local instruction executions as possible, so that the blocks are as long as possible. Performance suffers if the blocks are shorter than necessary, because shorter blocks imply more thread scheduling choices and therefore exponentially more thread interleavings to explore.

Two well-known tools that implement state space traversal and some form of POR are Java Pathfinder [20] and CHESS [29]. In this paper we focus on the partial order reduction technique used in Java Pathfinder (JPF). We describe partial order reduction, some other important concepts, and especially the proposed techniques in the context of model checking of multi-threaded Java programs with JPF.

JPF constructs the program state space on-the-fly, i.e., it is not created in advance before the state space traversal, and therefore the partial order reduction technique in JPF can use only information in the current state to determine whether the execution of a given instruction may be globally-relevant. This strategy is too conservative because it does not look ahead in the program execution and does not consider any information about what may happen in the future. As a consequence, many unnecessary thread scheduling choices may be created during the state space traversal, if the POR technique determines imprecisely that the execution of some instruction may be globally-relevant when it is actually thread local.

In addition to many more thread interleavings that must be explored, there is another reason why performance of JPF suffers due to unnecessary scheduling choices created during the state space traversal. At every thread scheduling choice, JPF performs garbage collection, serializes the current program state, and performs state matching to determine whether the state has already been encountered. These are expensive operations — they may take up to half of the running time of JPF [34]. Thus excessive thread scheduling choices significantly increase the running time of JPF.

Considering only accesses to fields of heap objects, a solution used by the POR technique in JPF is to consider a field access on a heap object as possibly globally-relevant if the object is reachable from multiple threads via a chain of references (pointers) in the given program state. It performs the dynamic escape analysis proposed in [13] to determine whether a heap object is reachable from multiple threads. Note that some of the threads may not actually access the object during the program execution. For example, execution of an instruction may be considered globally relevant because it writes to a field of an object reachable from other threads, but if no other thread will ever access that field, then the thread scheduling choice at the field write is not necessary. This strategy used to identify thread-local field accesses is imprecise for two reasons. First, the strategy conservatively assumes that every object that is reachable from multiple threads in the current dynamic heap will actually be accessed by those threads in the future. Second, the strategy does not distinguish individual