Parallel Computing 71 (2018) 42–57

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Output nondeterminism detection for programming models combining dataflow with shared memory

Hassan Salehe Matar[*], Erdal Mutlu, Serdar Tasiran, Didem Unat

Koç University, Rumelifeneri Yolu, Sarıyer, Istanbul 34450, Turkey

ARTICLE INFO

Article history:
Received 22 July 2016
Revised 4 September 2017
Accepted 23 November 2017
Available online 24 November 2017

Keywords:
Nondeterminism detection
Happens-before
Dataflow models
Shared memory
Data races
LLVM

ABSTRACT

Implementing highly concurrent programs can be challenging because programmers can easily introduce unintended nondeterminism, which has the potential to affect the program output. We propose and implement a technique for detecting unintended nondeterminism in applications developed on shared memory systems with dataflow execution model. Such nondeterminism bugs may be caused by missing or incorrect ordering of task dependencies that are used for ensuring certain ordering of tasks. The proposed method is based on the formulation of happens-before relation on tasks executions in a dataflow dependency graph. Its implementation is composed of two main phases; log recording and detection. For recording the necessary information from the execution, the tool instruments the dataflow framework and the applications, on top of the LLVM compiler infrastructure. Later it processes the collected log and reports on the found output nondeterminism in the execution. The tool can integrate well with the development cycle to provide the programmer with a testing framework against possible nondeterminism bugs. To demonstrate its effectiveness, we study a set of benchmark applications written in Atomic DataFlow programming model and report on real nondeterminism bugs in them.

1. Introduction

Nondeterminism has many uses in parallel applications and can be explored by a runtime scheduler to improve performance of concurrent tasks. However, *output* nondeterminism due to the improper ordering of tasks may produce different outputs at different runs, which is not desirable [1–4]. Output nondeterminism is regarded as a concurrency bug in parallel programs if it is not an intended behavior and particularly can be harmful for many scientific applications. Even though discovering such errors is difficult as it needs a deep exploration of different orderings of executing tasks, detecting them is necessary to ensure the correctness and reliability of applications.

This paper presents an algorithm for detecting output nondeterminism errors in dataflow applications executing in shared memory. In such an execution model, a program is implemented as data flowing between tasks while concurrent updates on the shared memory are protected by shared memory synchronization mechanisms such as locks, transactional memory, etc. [5–7]. Since a programmer has to specify the flow of data in the task graph [8], it can be difficult to implement a large application that is error-free; some of the dependencies can be easily missed. This paper focuses on a technique to assist the programmer in identifying output nondeterminism and reporting the related bugs for dataflow programs in shared memory.

[*] Corresponding author.
E-mail addresses: hmatar@ku.edu.tr (H.S. Matar), ermutlu@ku.edu.tr (E. Mutlu), stasiran@ku.edu.tr (S. Tasiran), dunat@ku.edu.tr (D. Unat).

https://doi.org/10.1016/j.parco.2017.11.008
0167-8191/© 2017 Elsevier B.V. All rights reserved.

To capture possible execution orderings of tasks, we propose a formulation of *happens-before* relation [9] between tasks in the dataflow dependency graph. Our happens-before relation model is based on dependencies between tasks and precisely captures the partial ordering of events between tasks. The proposed definition of happens-before handles well the dynamic creation of tasks as program executes. Based on the happens-before model, we develop a dynamic output nondeterminism error detection tool, called *DFinspec*, which automatically inspects a program and reports potential output nondeterminism bugs. We apply the tool to applications written in Atomic Dataflow programming model (ADF) [6], an example of dataflow implementations in shared memory, and demonstrate the tool's effectiveness on the DaSH benchmark suite [10]. The tool detected real concurrency bugs, that are previously unknown by the benchmark authors, and that are verified by manual inspection.

Part of the tool is developed as an LLVM compiler module to instrument the program during compilation by adding callbacks for detecting output nondeterminism. These callbacks record necessary information for output nondeterminism detection during execution of the instrumented application. The tool does not require the total number of tasks to be known in advance, as it adapts well with dynamic creation of new tasks as the program runs.

Our contributions can be summarized as below:

- A happens-before relation which captures the causal relations between tasks in a dataflow program running on shared memory. Our relation can handle the dynamic creation of tasks as the program runs.
- An automatic output nondeterminism error detection tool, which implements the happens-before relation model and detects bugs on real applications. The tool is modular and can be extended to detect output nondeterminism bugs in applications developed on programming models combining dataflow constructs with shared memory.
- An evaluation of the tool, which shows that the tool is capable of discovering real and synthesized output nondeterminism errors in a set of applications written in ADF.

2. Background and motivation

A task is a basic unit of computation in dataflow applications comprising a sequence of program statements that make use of the application data. In a dataflow program, a task graph is a collection of tasks connected through dependencies on *data* or *tokens*. Tasks are connected together by unique token buffers which create a data dependency such that a task which completes its execution triggers a succeeding task connected through the buffer by sending a token. A task becomes runnable when all tokens it requires are available. If more than one task is simultaneously runnable at a given point in time, they can execute in any order or concurrently.

If simultaneously runnable tasks concurrently execute and modify the same memory locations, different scheduling (i.e; order of execution) of these tasks may result in nondeterministic final values on these memory locations. Output nondeterminism in dataflow applications running on shared memory can occur between two tasks if the following two conditions are satisfied: *(i)* there is no ordering between these tasks enforced by a data dependency edge in the dataflow graph, and *(ii)* both tasks write different data values to the same shared memory location, or one task modifies the memory location while the other only reads from that location. Indeed, these conditions apply for all settings regardless whether a whole task is protected by transactional memory as in [6], or synchronization primitives such as internal locks [5,11]. We refer to nondeterminism arising due to these conditions in a program as *output* nondeterminism because it may produce different final outputs at different runs on the same input.

2.1. Motivating example

In dataflow applications the output nondeterminism can occur when a programmer misses necessary dependency between concurrent tasks. We provide a simple bank simulation example to motivate the problem. Fig. 1(a) simulates four bank account operations namely; account initialization, depositing, withdrawing and commissioning implemented in the ADF programming model. Assume that the intention of the programmer is to make *CommissionTask* execute after all other operations as in Fig. 1(c). Instead the programmer mistakenly implements the ADF program that creates four tasks to perform these bank operations as in Fig. 1(a) and (b). In this program the first task *OpenAccountTask* initializes the account balance `balance`, and then it sends tokens to three tasks *CommissionTask*, *DepositTask* and *WithdrawTask* (lines 3–5), which are concurrent to each other. Upon receiving the token, each task executes its action. Notice that all three tasks (*DepositTask*, *WithdrawTask* and *CommissionTask*) update `balance` in arbitrary order as dependency among them does not exist (see Fig. 1(b)). In ADF, by default, a task is *atomic*, or non-preemptive [6]. Therefore, either *CommissionTask* executes and completes before any of *WithdrawTask* and *DepositTask* or both or vice versa. Depending on which task executes first, the final value of `balance` differs, which is not the intention of the programmer.

To eliminate the nondeterminism bug, it suffices to add task dependencies from *DepositTask* and *WithdrawTask* to *CommissionTask*, as in Fig. 1(c) at lines 9 and 14. This ordering creates data dependency among these tasks as shown in Fig. 1(d). In the given example, the ordering between *WithdrawTask* and *DepositTask* does not matter because these two tasks commute; the scheduler can nondeterministically schedule these tasks without affecting the final program output.