Dynamic Verification for Hybrid Concurrent Programming Models

Erdal Mutlu[1], Vladimir Gajinov[2], Adrián Cristal[2,3],
Serdar Tasiran[1], and Osman S. Unsal[2]

[1] Koc University
{ermutlu,stasiran}@ku.edu.tr
[2] Barcelona Supercomputing Center
{vladimir.gajinov,adrian.cristal,osman.unsal}@bsc.es
[3] IIIA - CSIC - Spanish National Research Council

Abstract. We present a dynamic verification technique for a class of concurrent programming models that combine dataflow and shared memory programming. In this class of hybrid concurrency models, programs are built from tasks whose data dependencies are explicitly defined by a programmer and used by the runtime system to coordinate task execution. Differently from pure dataflow, tasks are allowed to have shared state which must be properly protected using synchronization mechanisms, such as locks or transactional memory (TM). While these hybrid models enable programmers to reason about programs, especially with irregular data sharing and communication patterns, at a higher level, they may also give rise to new kinds of bugs as they are unfamiliar to the programmers. We identify and illustrate a novel category of bugs in these hybrid concurrency programming models and provide a technique for randomized exploration of program behaviors in this setting.

Keywords: Dynamic verification, dataflow, transactional memory.

1 Introduction

Most modern computation platforms feature multiple CPU and GPU cores. For many large applications, it is more convenient for programmers to make use of multiple programming models to coordinate different kinds of concurrency and communication in the program. In this paper, we explore hybrid concurrent programming models that combine shared memory with dataflow abstractions.

Shared memory multi-threading is ubiquitous in concurrent programs. By contrast, in the dataflow programming model, the execution of an operation is constrained only by the availability of its input data – a feature that makes dataflow programming convenient and safe when it fits the problem at hand.

Using the dataflow programming model in conjunction with shared memory mechanisms can make it convenient and natural for programmers to express the parallelism inherent in a problem as evidenced by recent proposals [4,9] and adoptions [5,7,8]. The proposed hybrid programming models [4,9] provide

B. Bonakdarpour and S.A. Smolka (Eds.): RV 2014, LNCS 8734, pp. 156–161, 2014.
© Springer International Publishing Switzerland 2014

programmers with dataflow abstractions for defining tasks as the main execution unit with corresponding data dependencies. Contrary to the pure dataflow model which assumes side-effect free execution of the tasks, these models allow tasks to share the data using some form of thread synchronization, such as locks or transactional memory (TM). In this way, they facilitate implementation of complex algorithms for which shared state is the fundamental part of how the computational problem at hand is naturally expressed.

Enabling a combination of different programming models provides a user with a wide choice of parallel programming abstractions that can support a straightforward implementation of a wider range of problems. However, it also increases the likelihood of introducing concurrency bugs, not only those specific to a given well-studied programming model, but also those that are the result of unexpected program behavior caused by an incorrect use of different programming abstractions within the same program. Since the hybrid dataflow models we consider in this paper are quite novel, many of the bugs that belong to the latter category may not have been studied. The goal of this work is to identify these bugs and design a verification tool that can facilitate automated behavior exploration targeting their detection.

We present a dynamic verification tool for characterizing and exploring behaviors of programs written using hybrid dataflow programming models. We focus in particular on the Atomic DataFlow (ADF) programming model [4] as a representative of this class of programming models. In the ADF model, a program is based on tasks for which data dependencies are explicitly defined by a programmer and used by the runtime system to coordinate the task execution, while the memory shared between potentially concurrent tasks is managed using transactional memory (TM). While ideally these two domains should be well separated within a program, concurrency bugs can lead to an unexpected interleaving between these domains, leading to incorrect program behavior.

We devised a randomized scheduling method for exploring programs written using ADF. The key challenge in our work was precisely characterizing and exploring the concurrency visible and meaningful to the programmer, as opposed to the concurrency present in the dataflow runtime or TM implementations. For exploration of different interleavings, we adapted the dynamic exploration technique "Probabilistic Concurrency Testing (PCT)" [3] to ADF programs in order to amplify the randomness of observed schedules [2]. For shared memory concurrent programs, PCT provides probabilistic guarantees for bug detection. By properly selecting the scheduling points that PCT randomly chooses from, we aim to provide a similar guarantee for ADF programs.

In this paper, we motivate the use of and the need for a verification tool for ADF, explain our randomized behavior exploration tool and describe the experimental evaluation we are undertaking.

2 Motivation

In this section, we describe an unexpected execution scenario for motivating our dynamic verification method. Due to the asynchronous concurrent execution of

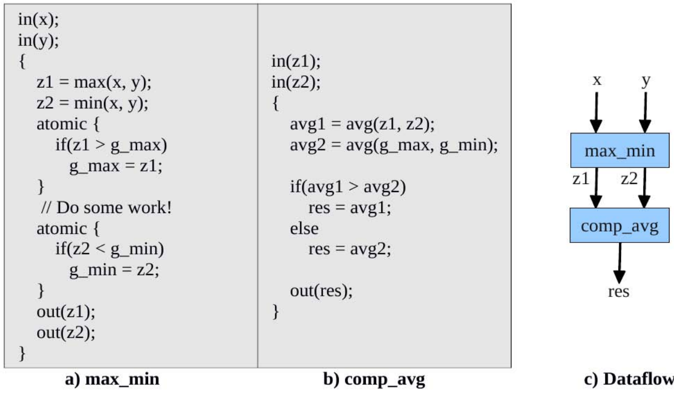

Fig. 1. Motivating example

tasks in the ADF model, users can face unexpected execution orders causing atomicity violations between dataflow tasks. To illustrate such a behavior, consider two ADF tasks in Figure 1, *max_min* that compute the maximum and minimum values from two input streams while updating a global minimum and maximum, and *comp_avg* that uses the output streams provided by *max_min* for comparing the average values of *g_max* and *g_min* with the input values and returning the bigger one. As seen in Figure 1-c, the dependencies between these tasks can be using the expressed with ADF programming model naturally as shown in Figure 1-a and b. However, while these particular implementations appear correct separately, when combined, they may result in unexpected behavior in an ADF execution. As the updates on the global variables, *g_max* and *g_min*, are performed in separate atomic blocks, concurrently running tasks can read incorrect values of global variables. Consider an execution where the first pair of integers from the input streams x and y are processed by *max_min* and then passed to *comp_avg*. During the execution of *comp_avg*, *max_min* can start to process the second pair and update *g_max* value, causing *comp_avg* to read the new *g_max* value from the second iteration while reading *g_min* value from the first one. Such concurrency scenarios that arise due to an interaction between dataflow and shared memory may be difficult to foresee for a programmer and are not addressed properly by verification methods for pure dataflow or pure shared memory model.

3 System Overview

3.1 Probabilistic Concurrency Testing

The "Probabilistic Concurrency Testing (PCT)" method relies on the observation that concurrency bugs typically involve unexpected interactions among few instructions that are executed by a small number of threads [6]. For capturing these unexpected thread interactions, PCT defines a *bug depth* parameter as the minimum number of ordering constraints that are sufficient to find a bug and

uses a randomized scheduling method, with provably good probabilistic guarantees, to find all bugs of low depth.

PCT makes use of a priority based scheduler that maintains randomly assigned priorities for each thread. During execution, the scheduler schedules only the thread with the highest priority until it becomes blocked by another thread or finishes its execution. For simulating the ordering constraints, the PCT scheduler also maintains a list of *priority change points*. Whenever the execution reaches a priority change point, the scheduler changes the priority of the running thread to a predetermined priority associated with the change point. With this mechanism, the PCT method can potentially exercise all bugs of depth d by simply using $d-1$ points.

Consider a program with n threads that together execute at most k instructions. Assuming that we want to find bugs with depth d, PCT provides a guarantee of finding a bug of depth d with the probability at least $1/nk^{d-1}$.

3.2 Our Method and Implementation

The ADF programming model has an inherently asynchronous concurrent execution model, where tasks can be enabled and executed multiple times. In addition, programmers are allowed to provide their custom synchronization using transactional memory to protect certain code blocks (not necessarily entire tasks) in ADF tasks. This can potentially influence the dataflow execution. In order to fully investigate behaviors of programs written using a hybrid model such as ADF, the dynamic exploration technique has to be aware of both the dataflow structure and the specifics of the shared memory synchronization mechanism. Furthermore, the dynamic verification tool should not simply instrument the platform implementations for transactional memory, atomic blocks and dataflow. This would not only be very inefficient, but it would also not provide value to the programmer. The user of a hybrid concurrent programming model is not interested in the concurrency internal to the platform implementing the model, which should be transparent to the programmer, but only in the non-determinism made visible at the programming model level.

We build upon the PCT algorithm but redefine priority assignment points, making use of TM transaction boundaries for priority change point assignment. Rather than using the original ADF work-stealing scheduler based on a pool of worker threads, we have devised a new scheduler that creates a thread with a randomly assigned priority for each enabled task and sequentially schedules the threads by honoring their priorities. Likewise, instead of using the original priority change point assignment from the PCT method, we narrowed possible priority change point locations to the beginning and the end of atomic regions only.

Given an ADF program with at most n enabled tasks that together execute at most k regions (atomic and non-atomic), our exploration method tries to find bugs of depth d as follows.

1. Whenever a task becomes enabled, randomly assign one of n priority values between d and $d+n$ to a thread associated with the task.
2. Pick $d-1$ random priority change points $k_1,...,k_{d-1}$ in the range of $[1,k]$ and associate priority value of i to k_i.

3. Schedule a thread with the highest priority and execute it sequentially. When a thread reaches the i-th change point, change its priority to i.

With this randomized scheduler, our exploration technique provides the following guarantee.

Given an ADF program with at most n enabled tasks that together execute at most k regions (atomic and non-atomic), our exploration method finds a bug of depth d with probability at least $1/nk^{d-1}$.

We implemented our exploration technique as a separate testing mechanism into the ADF framework. With this mechanism, users can choose the testing scheduler for exploring the behaviors of their applications with different task ordering for a given bug depth. Differently from conventional testing, our technique provides probabilistic guarantees for finding bugs and the overall detection probability can be increased by running our technique multiple times.

Our tool also provides a monitoring mechanism for checking globally-defined invariants during an execution. We provide the users with the capability to write global invariants on shared variables. These can be checked at every step by our tool, or at randomly assigned points in the execution.

Consider the motivating example in Figure 1 with input streams of length 2, our exploration technique can catch the the described buggy behavior with bug depth 2 as follow:

Initialization. Random priorities between d-$(n + d)$ (2-6 as the length of the input streams is 2, there can be at most 4 enabled tasks) will be assigned to the enabled tasks. As the only enabled task is *max_min*, let's assume it is given a priority of 4.
Later, $d - 1$ (1) priority change points will be assigned randomly among the start and end points of all atomic sections, assume this change point (as we are exploring bug depth 2) is chosen to be at the end of first atomic block in *max_min* task.

First iteration. The scheduler starts the execution by choosing the task with the highest priority. When the execution comes to a priority change point, the priority is lowered causing scheduler to check for a task with higher priority. In this case, *max_min* will continue to execute as there is no other enabled task.
After finishing the execution *max_min* task will enable the *comp_avg* task resulting in a priority assignment to it. Assume that the scheduler assigned 2 as the priority for the *comp_avg*.
The next set of inputs from the streams will enable *max_min* task again with new assigned priority to be 3.

Second iteration. Now scheduler will choose the enabled task with the highest priority for execution, which is *max_min* in this case.
While executing the *max_min* task, the priority will be changed at the priority change point and set to 1.
As a result scheduler will now choose *comp_avg* to execute causing the buggy behavior explained in Section 2.

4 Conclusion and Ongoing Work

This paper identifies and illustrates a novel category of bugs in the hybrid concurrency programming models that make use of dataflow and shared memory programming models, and provides a technique for randomized exploration of program behaviors in this setting.

We have started investigating ADF implementations of DWARF [1] benchmark applications. These applications are mostly numerical computations that have a structured dataflow with little shared memory accesses. We believe these to be a good initial set of benchmarks for discovering possibly missed cases in dataflow-heavy implementations.

In later experimental work, we plan to investigate the dynamic verification of the ADF implementation of a parallel game engine. In this complex application, the game map is divided between different tasks that process the objects moving between map regions. Dataflow is used to coordinate the execution of tasks that correspond to different game regions, whereas the TM synchronization is used to protect lists of objects, associated with each game region, that hold all the objects physically located within a region. By using the game engine application, we wish to evaluate how well our exploration method behaves with performance-critical applications characterized with highly-irregular behavior.

References

1. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Commun. ACM 52(10), 56–67 (2009)
2. Ben-Asher, Y., Eytani, Y., Farchi, E., Ur, S.: Producing scheduling that causes concurrent programs to fail. In: PADTAD 2006, pp. 37–40. ACM (2006)
3. Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. In: ASPLOS XV, pp. 167–178. ACM (2010)
4. Gajinov, V., Stipic, S., Unsal, O., Harris, T., Ayguade, E., Cristal, A.: Integrating dataflow abstractions into the shared memory model. In: SBAC-PAD, pp. 243–251 (2012)
5. Intel: Intel threading building blocks - flow graph, `http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm`
6. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In: ASPLOS XIII, pp. 329–339. ACM (2008)
7. Microsoft: Task parallel library - dataflow, `http://msdn.microsoft.com/en-us/library/hh228603.aspx`
8. OpenMP: Openmp 4.0 specification, `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`
9. Seaton, C., Goodman, D., Luján, M., Watson, I.: Applying dataflow and transactions to Lee routing. In: Workshop on Programmability Issues for Heterogeneous Multicores (2012)