Scientific Report for STSM project Developing a Dynamic Verification Tool for Atomic Dataflow (ADF) Programs

Erdal Mutlu
ermutlu@ku.edu.tr

January 7, 2014

Chair of Action IC1001	Dr. Paolo Romano (romanop@gsd.inesc-id.pt)
Science Officer	Mr. Ralph Stuebner (ralph.stuebner@cost.eu)
Administrative Officer	Ms. Aranzazu Sanchez (aranzazu.sanchez@cost.eu)
Reference Number	ECOST-STSM-IC1001-090913-034424
COST Action	IC1001
Action Description	Transactional Memories: Foundations, Algorithms, Tools and Applications (EURO-TM)
STSM Period	09.09.2013 to 08.12.2013
Participant	Erdal Mutlu Research Center for Multi-Core Software Engineering Koc University
Host	Dr. Osman Unsal Department for Computer Architecture for Parallel Paradigms Barcelona Supercomputing Center

1 Purpose of the STSM

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. Especially, hybrid concurrent programming models that combine shared memory with dataflow abstractions became popular as it is more convenient for parallelization of the most applications.

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 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 short term scientific mission (STSM) is to identify these bugs and design a verification tool that can facilitate automated behavior exploration targeting their detection.

We propose 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.

2 Description of the work carried out during the STSM

In this section, we motivate the use of and the need for a verification tool for ADF and explain our randomized behavior exploration tool.

2.1 Motivation

In this subsection, we illustrate the effectiveness of ADF programming model on a simple example and describe an unexpected execution scenario for motivating our dynamic verification method. In this example, two input streams of integers, x and y, are given and the goal is to provide, for each pair of tokens consumed from these streams, the larger of the two values, and also to maintain the global maximum of all values received on both streams. Figure 1 shows how this program is implemented in pure dataflow, shared memory and the ADF programming models. In pure dataflow, maintaining the global maximum is inefficient (the max of two tokens needs to be fed back to the task for the next step) whereas in the shared memory model, the global maximum is treated as the mutable state which can easily be updated in-place. However, condition variables are needed to detect the availability of new input tokens. ADF combines the best of two worlds. The atomicity of the code in the "atomic" block in Figure 1-b and c can be ensured using locks or transactional memory.

```
in(x);
in(y);
in(g_max);
{
   x = max(x, y);
   g_max = max(g_max, z);
   out(z);
   out(g_max);
}
```

a)

```
cond_wait(x);
read(x);
cond_wait(y);
read(y);
z = max(x, y);
atomic{
  if(z > g_max)
    g_max = z;
}
```

b)

```
in(x);
in(y);
{
   z = max(x, y);
   atomic {
      if(z > g_max)
        g_max = z;
   }
   out(z);
}
```

c)

Figure 1: Computing maxima using: a) dataflow, b) shared memory and c) ADF

```
in(x);
in(y);
{
   z1 = max(x, y);
   z2 = min(x, y);
   atomic {
      if(z1 > g_max)
         g_max = z1;
   }
    // Do some work!
   atomic {
      if(z2 < g_min)
         g_min = z2;
   }
   out(z1);
   out(z2);
}
```

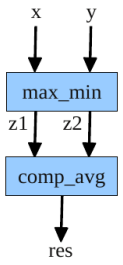
a) max_min

```
in(z1);
in(z2);
{
   avg1 = avg(z1, z2);
   avg2 = avg(g_max, g_min);

   if(avg1 > avg2)
      res = avg1;
   else
      res = avg2;

   out(res);
}
```

b) comp_avg



c) Dataflow

Figure 2: Motivating example

Due to the asynchronous concurrent execution of 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, *max_min* (Figure 2.1) 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 2.1-c, the dependencies between these tasks can be expressed with ADF programming model naturally as shown in Figure 2.1-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. Imagine an execution where first pair from the input streams are processed by *max_min* and than 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.

2.2 System Overview

In this subsection, we give background information about the "Probabilistic Concurrency Testing(PCT)" [3] and provide implementation details of our dynamic verification method for ADF programming model.

Ensure: Program P, integer n, k, and d
Require: $d \geq 0, n \geq maxthreads(P), k \geq maxsteps(P)$
  function PCT(n, k, d)
    for all $tId \in n$ **do**
      $priority[tId] = rand(d, d + n)$;
    end for
    for $i = 0 \rightarrow d - 1$ **do**
      $change\_points[i] = rand(1, k)$;
    end for
    while $availableThreads > 0$ **do**
      $ScheduleThread()$;
      if $currentStep \in change\_points$ **then**
        $newPriority = i \in \{0, ..., d - 1\}$
        $priority[currTId] = newPriority$
      end if
    end while
  end function

Figure 3: The PCT algorithm

2.2.1 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 point*s. 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 is able to find all bugs of depth d by simply having $d - 1$ number of change points.

Consider a program with n threads that together execute at most k instructions. Assuming that we want to find bugs with depth d, the pseudocode for the PCT algorithm in this setting is given in Figure 3. PCT provides a guarantee of finding a bug of depth d with the probability at least $1/nk^{d\text{-}1}$. The algorithm maintains a *priority* and *change_points* list for holding the priorities assigned to the threads and randomly chosen priority change points. After initializing these lists, the algorithm starts the scheduling of threads and executes until there are no available threads while checking for possible priority change points.

2.2.2 Design 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 own relaxed synchronizations using transactional memory in ADF tasks, which can possibly 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 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 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.

Further, we devised a monitoring mechanism for checking globally defined invariants during an execution. Differently from pure shared memory programming models, writing global properties as assertion is not practical within ADF tasks that are executing in a dataflow fashion. 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.

2.3 Preliminary Experimental Setup

As the first set of applications for our experiments, we decided to use the ADF implementations of DWARF [1] benchmark applications. These applications are mostly scientific algorithms like branch and bound, linear algebra, etc. that have a structured dataflow with little shared memory accesses. We believe this to be a good setup for discovering possibly missed cases in dataflow-heavy implementations.

For the second experimental setup, we plan to use the ADF implementation of the 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.

3 Future collaboration with host institution

Both participant sides (BSC and Koc University) are currently discussing the opportunity of publishing a joint paper with the experimental results (Section 2.3) for evaluating our method devised in this short visit.

4 Confirmation by the host institution of the successful execution of the STSM

The host Prof. Osman Unsal from BSC confirms that Erdal Mutlu achieved all the targets that we defined for this collaboration with distinction. He analyzed possible new kinds of bugs that can be caused by the use of hybrid concurrent programming models. And also he've devised a randomized exploration method for investigating possible behaviours of hybrid programming models combining shared memory concurrent models with dataflow programming models such as ADF.

References

[1] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[2] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Producing scheduling that causes concurrent programs to fail. PADTAD '06, pages 37–40, New York, NY, USA, 2006. ACM.

[3] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM.

[4] V. Gajinov, S. Stipic, O.S. Unsal, T. Harris, E. Ayguade, and A. Cristal. Integrating dataflow abstractions into the shared memory model. SBAC-PAD, pages 243–251, 2012.

[5] Intel. Intel threading building blocks - flow graph. `http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm`.

[6] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[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] Chris Seaton, Daniel Goodman, Mikel Luján, and Ian Watson. Applying dataflow and transactions to Lee routing. In *Workshop on Programmability Issues for Heterogeneous Multicores*, 2012.