

Runtime Race Detection for Shared Memory Programming Models

by

Hassan Salehe Matar

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

in

Computer Sciences and Engineering



June, 2018

Runtime Race Detection for Shared Memory Programming Models

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a doctoral dissertation by

Hassan Salehe Matar

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

Asst. Prof. Dr. Didem Unat

Asst. Prof. Dr. Ayşe Yilmazer

Prof. Dr. Attila Gürsoy

Assoc. Prof. Dr. Öznur Özkasap

Asst. Prof. Dr. Tankut Barış Aktemur

Date: _____



To Ibrahim and Ismail

ABSTRACT

This dissertation proposes methods for detecting runtime data races in three shared memory programming models: (i) OpenMP tasks, (ii) dataflow programming models such as Atomic DataFlow (ADF), and (iii) the POSIX Threads in embedded systems. The need for methods of detecting races in these models is fueled by the fact that they are commonly used in the HPC, parallel programming, and concurrent programming communities but there is a lack of tools to detect races in these programming models.

A *determinacy race* is a condition which occurs when concurrently executing entities (e.g; tasks) access the same memory location without specified ordering between them and at least one access is a write to that memory location. As a result, a program with determinacy races may produce different final output results at different runs on the same input. One potential problem when writing parallel programs with OpenMP is to introduce determinacy races where for a given input, the program may unexpectedly produce different final outputs at different runs. Such startling behavior can result from the incorrect ordering of OpenMP tasks. We present a method to detect determinacy races in OpenMP tasks at runtime. Based on OpenMP program semantics, our proposed solution models an OpenMP program as a collection of tasks with inferred dependencies among them where a task is implicitly created with a *parallel* region construct or explicitly created with a *task* construct. We define happens-before relation among tasks based on such dependencies for determining an execution order when detecting determinacy races. Based on this formalization, we developed a tool, TaskSanitizer, which detects and reports concurrent memory accesses whose tasks do not have common dependencies. Finally, TaskSanitizer works at runtime, has been able to find bugs in micro-benchmarks and it is reasonably effi-

cient to be utilized in a working environment. Archer is an efficient tool for detecting data races in OpenMP programs between concurrent threads. In contrast, we detect determinacy races where ordering between concurrent components is missing. Archer may fail to detect such cases and it also misses concurrent tasks executed by the same thread. By building the happen-before relations on tasks rather than threads, we can catch these situations.

We decided to call determinacy races in ADF as *output nondeterminism* because all tasks are atomic and therefore different final program outputs can be observed at different runs of the same buggy program and input. Output nondeterminism is possible if programmer does not specify necessary dependency between tasks which access the same memory locations. This is possible as implementing highly concurrent programs can be challenging because programmers can easily introduce unintended nondeterminism, which has the potential to affect the program output. Such unintended nondeterminism is output nondeterminism which is a special determinacy race where a program produces different final outputs at different runs on the same input, without such intention of the programmer. We propose and implement a technique for detecting output 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 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.

Lastly, we propose EmbedSanitizer, a tool for detecting concurrency data races in 32-bit ARM-based, multithreaded, POSIX Threads C/C++ applications. We motivate the idea of detecting data races in embedded systems software natively; without virtualization or emulation or use of alternative architecture. Detecting data races in applications on a target hardware provides more precise results and increased throughput and hence enhanced productivity of the developer. EmbedSanitizer extends ThreadSanitizer, a race detection tool for 64-bit applications, to do race detection for 32-bit ARM applications. We evaluate EmbedSanitizer using PARSEC benchmarks on an ARMv7 CPU with 4 logical cores and 933MB of RAM. Our race detection results precisely match with results when the same benchmarks run on 64-bit machine using ThreadSanitizer. Moreover, the performance overhead of EmbedSanitizer is relatively low as compared to running race detection on an emulator, which is a common platform for embedded software development.

This dissertation proposes a method for detecting determinacy races and it demonstrates its effectiveness using the OpenMP tasks. Moreover, it presents a technique for detecting determinacy races, dubbed output nondeterminism, in dataflow programming models such as the ADF. Lastly, it proposes a tool for detecting data races in 32-bit POSIX Threads applications for embedded systems. It is with anticipation that this dissertation will benefit both the industry and research communities. It also opens doors for further research on race detection for better solutions.

ÖZETÇE

Bu tez, üç paylaşımlı bellek programlama modelinde yarış durumu tespit edebilmek için yöntemler sunuyor: (i) OpenMP görevleri, (ii) Atomik DataFlow (ADF) gibi veri-akışı programlama modelleri, ve (iii) gömülü sistemlerdeki POSIX iş iplikleri. Bu modellerde yarış durumlarını tespit etme ihtiyacı HPC, paralel programlama ve koşut zamanlı programlama topluluklarında sıkça kullanılması gerçeğinden güç almıştır ama bu programlama modellerinde yarış durumu tespit edecek araçların eksikliği bulunmaktadır.

Belirlilik yarışı, koşut yürüyen varlıklar (mesela görevler) aynı hafıza bölgesine aralarında belirli bir sıralama olmadan eriştiklerinde ve içlerinden en az bir erişim o bölgeye yazma olduğunda oluşan bir durumdur. Sonuç olarak, belirlilik yarışı içeren programlar aynı girdinin farklı koşumlarında farklı sonuç çıktısı üretebilirler. OpenMP ile paralel program yazarken potansiyel problemlerden biri belirlilik yarışı oluşmasıdır öyle ki verilen girdi için program farklı koşumlarında beklenmedik bir şekilde farklı son çıktılar üretebilir. Böyle şaşırtan davranışlar OpenMP görevlerinin yanlış sıralanmasından doğabilir. OpenMP görevlerindeki belirlilik yarışlarının işleyiş süresi içinde tespiti için bir yöntem sunuyoruz. OpenMP programlarının anlamına dayanarak, önerdiğimiz çözüm, bir OpenMP programını aralarında çıkarım bağımlılıkları olan görev koleksiyonları olarak modelliyor, öyle ki her görev ya üstü kapalı olarak bir *parallel* bölge yapısı tarafından yaratılıyor ya da doğrudan bir *görev* yapısı tarafından yaratılıyor. Biz belirlilik yarışlarını tespit ederken yürütme sırasını belirlemek için böyle bağımlılıklara dayanan bir önce-olur ilişkisi tanımlıyoruz. Bu biçimselleştirmeye dayanarak görevlerin ortak bağımlılıklarının olmadığı koşut hafıza erişimlerini tespit eden ve raporlayan TaskSanitizer isimli aracı geliştirdik. Son olarak, TaskSanitizer işleyiş süresinde çalışıyor, mikro-değerlendirme deneylerinde hataları bulabildi ve çalışma

ortamlarında faydalanılabilecek kadar etkili. Archer OpenMP programlarında koşut zamanlı iş iplikleri arasında veri yarış durumu tespiti için etkili bir araçtır. Aksine, biz koşut zamanlı bileşenler arasında sıralama olmadığı durumlarda oluşan belirlilik yarışlarını tespit ediyoruz. Archer bu durumları tespit etmekte başarısız olabilir ve aynı iş iplikleri tarafından çalıştırılan koşut görevleri kaçırıyor. İş iplikleri yerine görevler üzerinde önce-olur ilişkisi inşa ederek bu durumları yakalayabiliyoruz.

ADF'deki belirlilik yarışlarını *çıkta belirsizliği* olarak adlandırmaya karar verdik çünkü bütün görevler atomiktir ve bu yüzden hatalı program ve girdinin farklı koşullarında farklı çıktılar gözlemlenebilir. Eğer programcı aynı hafıza bölgelerine erişen görevler arasındaki gerekli bağımlılıkları belirlemezse, çıkta belirsizliği mümkün hale gelir. Bu durum gayet mümkündür çünkü yüksek seviyede koşut zamanlı programları gerçekleştirmek çetrefilli bir iştir ve programcılar kolayca program çıktısını etkileme potansiyeline sahip istemsiz belirsizlikler uygulamaya koyabilir. Böyle istemsiz belirsizlikler programcının niyeti dışında aynı girdi ile farklı koşullarda farklı sonuçlar üreten özel belirlilik yarışlarıdır. Veri-akışı çalışma modeli paylaşımli bellek sistemlerde geliştirilmiş uygulamalardaki çıkta belirsizliğini tespit etmek için bir teknik öneriyor ve gerçekleştiriyor. Böyle belirsizlik hataları görevlerin bazı sıralamalarını sağlamak için kullanılan görev bağımlılıklarının eksik veya yanlış sıralanmasından dolayı oluşabilir. Önerilen yöntem, veri-akışı bağımlılık çizgesinde görevler üzerinde önce-olur ilişkisinin formüle edilmesine dayanmaktadır. Gerçekleşmesi iki ana fazdan oluşur: günlük kayıtları ve tespiti. Yürütümden gerekli bilgiyi kaydetmek için, araç LLVM derleyici altyapısı üzerinde, veri-akışı çatısı ve uygulamalarını ölçüm araçlarıyla donatır. Sonra, yürütümde bulunan çıkta belirsizliklerinden toplanan günlük ve raporları işler. Programcıya olası belirsizlik hatalarına karşın bir test çatısı sağlamak için, araç geliştirme döngüsüne entegre edilebilir. Etkinliğini göstermek için, ADF modelinde yazılan bir değerlendirme deney kümesi ile çalıştık ve onlardaki gerçek belirsizlik hatalarını raporladık.

Son olarak; 32-bit ARM tabanlı, çok iş örgülü, POSIX iş örgüleri kullanan C/C++ uygulamalarında koşut zamanlı veri yarışlarını tespit eden bir araç olan Embed-

Sanitizer'ı öneriyoruz. Gömülü sistem yazılımlarında koşut zamanlı veri yarışlarını sanallaştırma, emülasyon ya da başka bir mimari kullanmadan yerel olarak tespit etme fikrini teşvik ediyoruz. Hedef donanımda çalışan uygulamalardaki veri yarışlarını tespit etmek daha kesin sonuçlar, artan verimlilik sağlar ve böylece geliştiriciye yüksek üretkenlik sağlar. EmbedSanitizer, 64-bit uygulamalar için bir yarış algılama aracı olan ThreadSanitizer'ı 32-bit ARM uygulamalarında yarış tespiti yapacak şekilde geliştirir. EmbedSanitizer'ı 933MB RAM'i ve 4 mantıksal çekirdeği olan ARMv7 işlemcili bir makinede PARSEC değerlendirme deneylerini kullanarak değerlendiriyoruz. Yarış tespiti sonuçlarımız aynı değerlendirme deneyinin ThreadSanitizer kullanan 64-bit bir makinede verdiği sonuçlarla eksiksiz biçimde uyuşmakta. Ayrıca, EmbedSanitizer'ın başarımlar ek yükleri gömülü yazılım geliştirmek için yaygın bir platform olan bir emülatörde yarış algılamasını çalıştırmaya göre daha düşük.

Bu tez, belirlilik yarışlarını tespit etmek için bir yöntem sunuyor ve OpenMP görevlerini kullanarak etkililiğini gösteriyor. Buna ek olarak, ADF gibi veri akışı programlama modellerinde belirlilik yarışlarını tespit etmek için bir yöntem sunuyor. Son olarak, bu tez gömülü sistemlerde 32-bit POSIX iş örgüleri uygulamalarındaki veri yarışlarını tespit etmek için bir araç sunuyor. Bu tezin hem endüstriye hem de araştırma topluluklarına faydalı olacağı bekleniyor. Ayrıca bu tez yarış algılamada daha iyi çözümler için ileri araştırmaların kapısını açıyor.

ACKNOWLEDGMENTS

I send my special gratitude to my adviser *Asst. Prof. Didem Unat*. She has been inspirational and motivational to excel in my PhD studies and research. She has also been the reason for me to successfully fulfill my PhD. My thanks to *Assoc. Prof. Serdar Tasiran* for securing funding for my entire PhD candidature. I recognize the warmth and care of my former MSRC research lab-mates who made me feel at home at Koç University and Istanbul: *Erdal Mutlu, Ismail Kuru, Burcu Kulahçioğlu Özkan*, and *Suha Orhun Mutluergil*. My research colleagues at ParCoreLab *Muhammad Nufail Farooqi, Najeeb Ahmad, Muhammad Aditya Sasongko, Pirah Noor Soomro, Burak Bastem, Mohammad Laghari, Doğa Dikbayır*, and *Can Aknesil* have been great sources of knowledge and a new family which has been nurturing me. Special thanks to my thesis jury which comprises my thesis advisor, *Prof. Dr. Attila Gürsoy*, *Assoc. Prof. Dr. Öznur Özkasap*, *Asst. Prof. Dr. Ayşe Yılmaz* from Istanbul Technical University, and *Asst. Prof. Dr. Tankut Barış Aktemur* from Özyeğin University. I am also grateful of the research funding from the Affordable Safe & Secure Mobility Evolution (ASSUME) project for smart mobility. Lastly, special thanks to Arçelik A.Ş for providing a television, software and technical support for my research.

I would also like to express my deepest gratitude to my wife and my whole family for their patience throughout my research.

TABLE OF CONTENTS

List of Tables	xiv
List of Figures	xv
Nomenclature	xvii
Chapter 1: Introduction	1
Chapter 2: Background on Race Detection	6
2.1 Data Races	6
2.2 Determinacy Races	7
2.3 Race Detection	9
2.4 Race Detection Algorithms	10
2.5 Race Detection Tools	12
Chapter 3: Related Work	13
3.1 Race Detection for OpenMP	13
3.2 Nondeterminism Detection	14
3.3 Race Detection for POSIX Threads	16
3.3.1 Race Detection of POSIX Threads in Embedded Systems . . .	17
Chapter 4: TaskSanitizer: Runtime Determinacy Race Detection for OpenMP Tasks	18
4.1 Introduction	18
4.2 Background in OpenMP Tasks	19
4.3 Determinacy Race Detection	21

4.3.1	Definition and Motivating Example	21
4.3.2	Formalizing Task Operations	22
4.3.3	Happens-Before Relations Between Task Operations	23
4.3.4	Determinacy Race Detection Algorithm	25
4.4	Implementation	27
Chapter 5: DFinspec: Nondeterminism Detection for ADF		30
5.1	Introduction	30
5.2	Motivation	31
5.3	Proposed Approach	32
5.3.1	Defining Actions of a Dataflow Application	32
5.3.2	Happens-before Relation of Task Actions	33
5.3.3	Output Nondeterminism Detection Rules	35
5.3.4	Nondeterminism Detection Algorithm	36
5.3.5	Commuting Tasks	39
5.4	Implementation	43
5.5	Runtime Instrumentor	43
5.6	Application Instrumentor	44
5.7	Logger	44
5.8	DFchecker: Output nondeterminism detection	46
5.9	Detecting commuting writer tasks	49
Chapter 6: EmbedSanitizer: Race Detection for PThread in 32-bit Embedded ARM		51
6.1	Introduction	51
6.2	Motivation	53
6.3	Method	54
6.4	Architecture and Workflow	54
6.4.1	Installation	56

Chapter 7: Evaluation	57
7.1 Evaluating Runtime Determinacy Race Detection for OpenMP Tasks	57
7.1.1 Precision Evaluation of TaskSanitizer	59
7.1.2 Comparing Detection with Archer	60
7.1.3 Overhead Evaluation	61
7.2 Evaluation on Detecting Output Nondeterminism for ADF	61
7.2.1 Detecting Real Output Nondeterminism Bugs	63
7.2.2 Detecting Synthesized Output Nondeterminism Bugs	65
7.2.3 False Output Nondeterminism Due to Commutativity	67
7.2.4 Overhead	68
7.2.5 Limitations	70
7.3 Results on Race Detection for POSIX Threads in 32-bit Embedded ARM	70
7.3.1 Precision Evaluation	71
7.3.2 Performance Evaluation	72
 Chapter 8: Conclusion	 74
 Bibliography	 76

LIST OF TABLES

5.1	Summary of logging formats	45
7.1	Comparing detection results of TaskSanitizer against Archer	59
7.2	Experimental results from 10 applications and the motivating bank example.	64
7.3	Showing number of synthetic bugs added into dataflow applications and a number of those detected by <i>DFinspec</i>	66
7.4	Showing both real bugs and false alarms. The false alarms were due to commutative tasks and were eliminated altogether by applying the algorithm discussed in section 5.3.5.	68
7.5	Execution times and overheads of instrumentation, logging and output nondeterminism detection phases of <i>DFinspec</i>	69
7.6	Experimental results to compare race detection in ARMv7 using <i>EmbeddedSanitizer</i> vs in x86_64 with <i>ThreadSanitizer</i>	72

LIST OF FIGURES

2.1	Example program with a data race at line 3 between threads T_1 and T_2 .	7
2.2	OpenMP example illustrates explicit and implicit tasks and their logical flow dependency between tasks. The code example has a determinacy race.	8
4.1	OpenMP example illustrates explicit and implicit tasks and their logical flow dependency between tasks. The code example has a determinacy race.	20
4.2	Defining a task as a sequence of tasksegments (taskseg) and synchronizations	23
4.3	Example with four categories of HB relation among operations of tasks	25
4.4	Showing implementations of TaskSanitizer: architecture and tool flow	27
5.1	A simple banking ADF example (a) code with nondeterministic behavior between <i>WithdrawTask</i> , <i>DepositTask</i> and <i>CommissionTask</i> . The flow graph (b) shows dependency between these tasks indicating that <i>DepositTask</i> , <i>WithdrawTask</i> and <i>CommissionTask</i> are concurrent. The code in (c) shows the correct implementation of the code in (a). The flow graph (d) shows the correct dependency which ensures that <i>CommissionTask</i> executes last, thus ensuring a deterministic execution of the banking operations. <i>DepositTask</i> and <i>WithdrawTask</i> can execute in any order because they can not affect the final output.	31
5.2	happens-before formulas for capturing the ordering of actions in a dataflow application.	35

5.3	The architecture and action flow of <i>DFinspec</i> tool.	43
6.1	A motivating example with two threads concurrently accessing a shared queue. A thread in (a) reads video signals from TV antenna and puts them into the queue, (b) reads from the queue and display to a screen.	53
6.2	High level abstraction of <i>ThreadSanitizer</i> and <i>EmbedSanitizer</i> in LLVM/Clang. In (a) <i>ThreadSanitizer</i> : essential LLVM modules for race detection. In (b) <i>EmbedSanitizer</i> : same modules modified to instrument and detect races for 32-bit ARM	55
6.3	Showing the automated process for building <i>ThreadSanitizer</i> for the first time.	56
7.4	Slowdown of determinacy race detection in programs as input size increases	61
7.5	Task dependency graph of <i>sparse linear algebra</i> showing a missing dependency (shown as a dash line) between the <i>bmod</i> and <i>lu0</i> operations. This causes an output nondeterminism bug captured by our tool and manually verified.	65
7.6	Task graphs of applications showing removed data flow dependencies to introduce output nondeterminism bugs	65
7.7	Slowdown comparison of race detection on ARMv7 vs on Qemu-ARM	73

NOMENCLATURE

AC	-	Account
API	-	Application Programming Interface
ADF	-	Atomic DataFlow
CPU	-	Central Processing Unit
DAG	-	Directed Acyclic Graph
HB	-	Happens-Before Relation
HPC	-	High Performance Computing
ID	-	Unique Identifier
IR	-	LLVM's Intermediate Representation of Code
JIT	-	Just-In-Time Compilation
LLVM	-	Low Level Virtual Machine
MB	-	Megabyte
OpenMP	-	Open Multi-Processing
OMPT	-	OpenMP Performance Tools API
POSIX	-	The Portable Operating System Interface
QEMU	-	Quick Emulator
TBB	-	Intel Threading Building Blocks
TV	-	Television

Chapter 1

INTRODUCTION

Multicore systems are everywhere; from smartphones to televisions, to tablets, to clusters. This has forced programmers to write applications that benefit from concurrency that these systems provide in terms of performance and efficiency. Unfortunately, reasoning about concurrency is hard and challenging in order to write logically and semantically correct programs while attaining optimum performance and efficiency. It is even trickier to write concurrent software for shared memory where concurrently executing entities (e.g; threads) read from and write to common memory locations and their updates are visible to other entities. Moreover, the mechanisms to coordinate the access to these locations can lead to program inconsistency and performance degradation.

To address some of the challenges for writing software for shared memory multicore systems, experts have devised programming models. A programming model for multicore is a paradigm or a way of expressing parallelism in concurrent programs. It simplifies reasoning about concurrency by providing Application Programming Interfaces (API) that programmers use to express different concurrency semantics in their program. Moreover, these models provide mechanisms for enforcing program consistency and improved performance. The commonly used programming models for writing parallel applications for shared memory multicore systems are: (i) OpenMP [4,18], (ii) dataflow programming models such as the Atomic DataFlow(ADF) [27], and (iii) the *POSIX Threads* [14].

OpenMP [4, 18] is the commonly used programming model in the High Performance Computing(HPC) community for shared memory programs. In OpenMP,

programmers specify their parallel parts of their code using the *pragma* compiler directive and a compiler with OpenMP support interprets these pragmas and injects appropriate OpenMP APIs to ensure consistent parallel execution of the program. The OpenMP runtime is responsible for creating threads, transparently from the programmer, which execute parallel parts of the program. Moreover, the runtime uses available hardware resources (e.g; available CPUs) to optimize performance of the programs. For program correctness, the API provides thread synchronization constructs which are expressed as clauses or attributes in the pragma directive or through explicit API calls.

Programming data-parallel programs in OpenMP is straightforward for improved performance, however, writing correct task-parallel programs can be challenging. Specifically, the programmer has to use specific pragma attributes to specify sharing of memory locations between parallel regions to ensure consistent executions. More importantly, a keen understanding of the API, program design and reasoning about concurrency is necessary to implement task-parallel programs which attain high runtime performance. To leverage these challenges to the runtime, OpenMP has introduced shared memory dataflow execution model [3]. In OpenMP tasks, programmers specify computations in units called *tasks* which can be executed by concurrent threads. Moreover, a programmer specifies execution ordering between tasks executing in shared memory through data dependencies where a succeeding task never executes until the preceding task terminates.

In similar fashion to OpenMP tasks, Atomic Dataflow (ADF) [27] is a shared memory programming model where tasks execute atomically; intermediate memory updates are not visible until task terminates. This guarantees that if two concurrent tasks update the same memory location, only one task is guaranteed to progress at a time. In this model, specifying task ordering through data dependency avoids the cost of identifying the conflicting tasks while maintaining program consistency.

POSIX Threads [14], also known as *Pthread*, is a widely used execution model in shared memory programs. In this model, a programmer specifies parallel computa-

tions in software threads which are scheduled for execution by the operating system. Despite its efficient API support, it is the programmer's responsibility to ensure that threads access shared memory variables using atomic constructs supported by the hardware or with proper synchronization mechanisms provided by the model (e.g; mutexes, condition variables, etc) both for program correctness and runtime performance.

Despite a key success of these programming models, programmers can introduce data races into their programs. A data race (or a race condition) is a condition which occurs when two concurrently executing threads access a shared memory location without proper synchronization and at least one of these accesses (or operations) is a *write* [51,56]. In this context, the term *access* or *accesses* refers to a set of memory operations with at least one memory write. Availability of data races in a program can be a symptom of other concurrency errors such as atomicity and linearizability violations, deadlocks, and nondeterministic behaviors like memory violations and program crashes. A race can occur if shared variables are accessed by different threads without a common synchronization to protect those variables. In OpenMP tasks and ADF, a *determinacy race* may occur when concurrently executing tasks access the same memory location without specified ordering between them and at least one access is a write to that memory location. There is a subtle difference between a data race and a determinacy race. A data race is caused by improper synchronization of concurrent threads on shared memory accesses while a determinacy race is due to missing of ordering between two concurrent entities which access common shared memory locations. Finally we decided to call determinacy races in ADF as *output nondeterminism* because all tasks are atomic and therefore different final program outputs can be observed at different runs of the same program and input. Output nondeterminism is possible if programmer does not specify necessary dependency between tasks which access the same memory locations. As a result, different final outputs are produced at different program runs [44]. This behavior may be undesired for some applications like the scientific workloads where accuracy of the results is important.

There have been efficient tools for detecting data races in shared memory programming models. *Archer* detects data races in OpenMP applications [9]. Moreover, *ThreadSanitizer* [61] and *Intel Inspector* [31] are industrial-level race detection tools for POSIX Threads applications. Unfortunately, these tools are still insufficient as they have a number of limitations. *Archer* detects data races in OpenMP programs between concurrent threads and therefore it fails to detect races between concurrent tasks in case they are scheduled to and are executed by one thread. Therefore, it traces improper synchronization of threads rather than improper ordering of execution entities (e.g; tasks). Furthermore, to our knowledge, there is no prior work on output nondeterminism detection in dataflow applications in shared memory, such as the ADF. Finally, *Intel Inspector* detects data races only in x86_64 platforms whereas *ThreadSanitizer* supports only 64-bit platforms.

In this dissertation, we propose techniques for detecting data races in the commonly used shared memory programming models to address the limitations of the research in the literature. In particular, we propose the following:

- (a) A technique for detecting runtime determinacy races in OpenMP tasks;
- (b) A method for detecting output nondeterminism in Atomic DataFlow(ADF) applications; and
- (c) An approach for detecting data races for POSIX Threads programs in 32-bit embedded systems.

This dissertation is organized as follows. Background on data race detection is presented in Chapter 2, while the related work is discussed in Chapter 3. In Chapter 4 we propose a technique for detecting runtime determinacy races for OpenMP tasks. Our key contributions are: (a) A formal definition of the determinacy races and a technique for detecting such races in OpenMP tasks. To our knowledge, no prior work has been done for detecting determinacy races in OpenMP tasks with mixed structures of critical and non-critical sections. (b) Determinacy race detection tool for OpenMP called *TaskSanitizer* [46]. (c) Evaluation of our method using micro-

benchmark applications and comparison of results against a race detection tool for OpenMP programs.

In Chapter 5 we propose DFinspec, our method for detecting output nondeterminism in ADF applications, with three main contributions: (a) A happens-before relation which captures the partial relations between tasks in a dataflow program running on shared memory. Our relation can handle the dynamic creation of tasks as program runs; (b) 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; (c) An evaluation of our proposed solution, which shows that it is capable of discovering real and synthesized output nondeterminism errors in a set of applications written in ADF.

Chapter 6 discusses our proposed solution for detecting data races in POSIX Threads for embedded systems software. Our key contributions on detecting races for embedded systems are: (a) A tool for detecting data races in C/C++ multi-threaded programs for 32-bit embedded ARM. The tool is easily accessed through Clang compiler chain like *ThreadSanitizer*; (b) We motivate the idea of supporting race detection in native embedded systems hardware and show usability of race detection to such architectures; (c) Evaluation of applicability of our approach by detecting races in real TV software and by running PARSEC benchmark applications on a TV with ARM Cortex A17 (ARMv7) CPU and limited memory of 933 MB.

The remaining parts of the dissertation are organized as following. Chapter 7 discusses evaluation strategies and experimental results for the methods we propose. Finally, the conclusion is presented in Chapter 8.

Chapter 2

BACKGROUND ON RACE DETECTION

In this section we discuss concepts related to race detection and categorize them into five groups. First, we provide a definition of a data race. Second, we introduce a special type of data races called *determinacy races*. Third, we talk about race detection in general. Fourth, we present race detection algorithms in literature and classify them based on their underlying methods and efficiency. We finalize by examining race detection tools and their limitations on detecting data races in shared memory programming models.

2.1 Data Races

A data race [51] occurs when two concurrently executing units (e.g; threads) access a shared memory location without proper synchronization and at least one of these accesses is a write. It occurs if three conditions are satisfied in a concurrent program [56]: (a) presence of memory accesses performed on a shared memory location by different threads, (b) at least one of them is a write, and (c) they are performed without a common synchronization.

As an example in POSIX Threads execution model, a data race to a shared memory can occur if it is not protected by the same lock in all of its accesses by concurrent threads. Figure 2.1 shows a simple Pthread example where two threads concurrently increment a shared variable `counter` at line 3. The access in thread T_1 is protected by a lock `lck1` whereas access in T_2 is protected by a different lock `lck2`. Since no same lock is used to access `counter`, there is a data race.

Presence of data races in a program can be a symptom of other concurrency errors such as atomicity and linearizability violations, or deadlocks. Moreover, data races


```
int counter = 0;

/** Function for thread T1 */
1 void forThread1(void *) {
2   pthread_mutex_lock(lck1);
3   counter++;
4   pthread_mutex_unlock(lck1);
5 }

/** Function for thread T2 */
1 void forThread2(void *) {
2   pthread_mutex_lock(lck2);
3   counter++;
4   pthread_mutex_unlock(lck2);
5 }
```

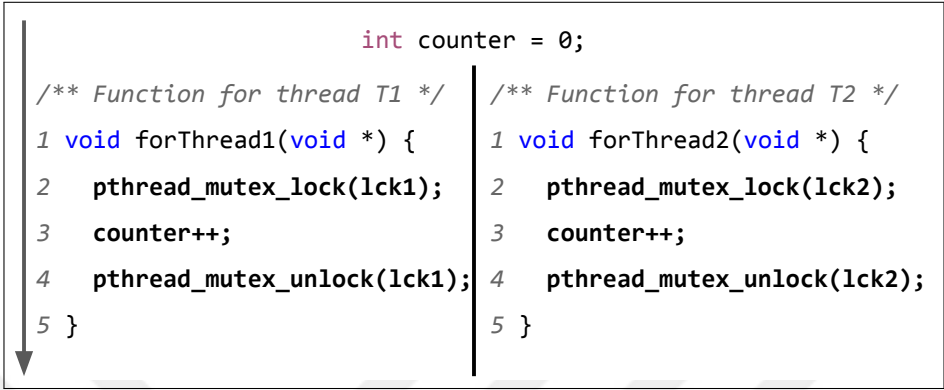


Figure 2.1: Example program with a data race at line 3 between threads T_1 and T_2 .

may result in unpredictable behaviors like memory violations and program crashes. In general, data races have different consequences depending on the type of software. First, presence of data races in real-time systems like medical devices, automotive and airplanes can cause human tragedy. Incidents of Therac-25, a radiotherapy medical device, have caused human losses in the past due to data races [41]. Second, data races can cause inconsistent program results which may impact on research finding or on scientific applications and enterprise systems, respectively. Last, degradation of quality of services in less critical applications such as TV broadcasting software, and internet browsing tools may be played in part by presence of data races. These races may produce inconsistent results or program crashes that can just be eliminated by restarting the program. Nevertheless, finding and fixing data races in any software is vital to ensure safe, secure and consistent program execution.

2.2 Determinacy Races

A *determinacy race* is a condition which occurs when concurrently executing entities (e.g; tasks) access the same memory location without specified ordering between them and at least one access is a write to that memory location [22,40,57,58]. As a result, a program with determinacy races may produce different final output results at different runs on the same input [44]. Determinacy races are possible if the programmer does

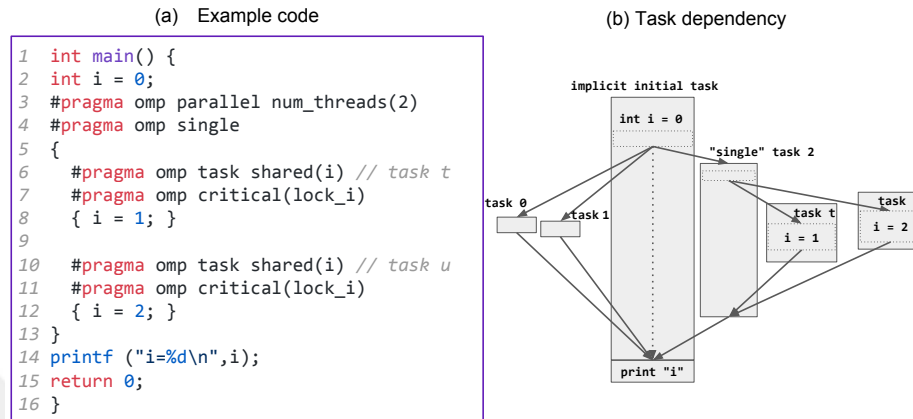


Figure 2.2: OpenMP example illustrates explicit and implicit tasks and their logical flow dependency between tasks. The code example has a determinacy race.

not specify necessary dependency between concurrent tasks which access the same memory locations. Since there is no specific order defined by the programmer, the scheduler is free to execute the tasks in any order or concurrently.

Formally, a determinacy race occurs between two tasks if the following two conditions are satisfied: (i) there is no ordering between these tasks enforced by task dependency, and (ii) both tasks access a common shared memory location and at least one access is a write. If simultaneously runnable tasks 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.

We have provided a simple OpenMP program in Figure 2.2, where there is no specified dependency between tasks t and u . As a result, their critical sections can execute in any order and thus the final result for i can either be 1 or 2 despite the fact that accesses to the shared variable are protected by a common lock. Unless the developer intends the program to behave as such, only one deterministic result is expected. The same issue arises if one of the tasks reads the value of i in a critical region and the other task writes to i . It is worth noting that in a typical program these two tasks might have been created in separate function calls, thus the critical sections may be well far apart from each other and can be easily overlooked.

A tasking 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. [23, 27, 53]. Since a programmer has to specify dependency among concurrent tasks which access common shared memory, it can be difficult to implement a large application that is error-free; some of the dependencies can be easily missed. 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 due to determinacy races.

Not only a proper synchronization of memory accesses is necessary but also a proper ordering of these memory operations ensures the absence of determinacy races. 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.

2.3 Race Detection

Identifying data races in concurrent programs is necessary to eliminate concurrency bugs. Race detection is a technique which aims to find races in concurrent programs for shared memory models. In general settings, it involves locating parts of the program which exhibit a race and is programmer's responsibility to revisit their program codes and fix relevant issues. One advantage of race detection is its automation that is generally more efficient and resource effective than human's manual inspection of the source code. Therefore, there is an active research on detecting data races for concurrent applications to aid programmers in identifying them for elimination [9, 22, 24, 25, 28, 29, 31, 33–35, 40, 44, 47–49, 54, 60, 61, 64, 66, 67, 72].

There are two approaches to detecting data races in concurrent programs. The first is static race detection where program is analyzed at source or binary level without running it [33, 35, 49, 66, 67]. The common approach to identify shared memory locations statically is by symbolic execution. Unfortunately, detecting races statically

is insufficient or unsound because it does not include the runtime behavior of the program and thus it tends to produce significant number of *false positives* and *false negatives*. A false positive is a warning about the presence of a data race where in reality there is none, and false negative means that the tool fails to report a real race in the program.

The second line of work that is commonly used for detecting races is runtime or dynamic race detection [9, 24, 25, 28, 31, 44, 48, 54, 60, 61, 64, 72]. In this setting, races are detected based on program events – such as memory reads, writes and synchronization operations – at runtime while the program executes. These events can be identified through instrumentation which inspects binary by identifying relevant operations and injecting runtime monitors which report when these events happen at runtime. Runtime race detection can be on-the-fly or post-mortem. On-the-fly race detection occurs when the event happens while the program is running whereas post-mortem involves collection of program events at runtime and race detection is done once the program terminates. Detection of races in these scenarios depends on whether the race exhibits itself in the program execution. It can as well be hidden or obscured depending on the given program input.

2.4 Race Detection Algorithms

Race detection algorithms aim to detect and signal races in concurrent programs. They are normally implemented in independent runtime libraries which are injected by race detection tools into programs for detecting races. Algorithms keep metadata which track ordering or synchronization of program events in concurrent programs. A race detection algorithm is said to be *precise* if it does not produce any false alarms (false-positives or false negatives) about races for a given execution of a program [24]. On the other hand, *sound* race detection algorithms avoid false positives but may miss real races (i.e; false negatives) [62]. *Unsound* race detection algorithms, nevertheless, contain both false positives and false negatives [60].

The most common algorithms for race detection rely on either *happens-before* [38]

or *lockset*-based approaches [60] or the hybrid of these approaches [12, 61]. Lockset algorithms track all synchronization locks which a thread acquires to access shared memory locations. This helps to identify memory locations accessed without locks or common locks as sources of races [60]. Moreover, they compute a set of locks acquired to access a shared memory location [60]. The aim is to see if all accesses to a shared memory by different threads in the program use common locks. Lockset approaches tend to be unsound and are undesired for practical race detection.

A happens-before is a partial relation, among program events, which aims to infer ordering among them and help identify events not ordered but access common memory locations as sources of data races [24, 54]. Event w happens before event x if there is an *enforced ordering* such that w is guaranteed to execute before x in timeline. In concurrent programs, an order can be enforced by one of the following scenarios:

- (a) *Program order*: events in a thread program sequence are assumed to execute in their order;
- (b) *Synchronizes-with order*: a lock release is ordered before its subsequent acquire;
- (c) *Transitive closure of events*: if event x happens-before event y and that y happens-before event z , then x happens-before z .

Among these two approaches, happens-before algorithms are *precise*: in a given concurrent program execution with races, they always report at least one real race [24]. One example of happens-before algorithms is FastTrack [24]. It is an efficient and precise race detection algorithm which improves on purely happens-before vector clock algorithms such as DJIT++ [54]. FastTrack optimizes performance by showing that majority of memory access patterns do not require a whole vector clock to detect data races. Instead, an *epoch*, a simple pair of thread identifier and clock suffices. Without sacrificing precision, this significantly improves the performance of race detection of a single memory access from $O(n)$ to $O(1)$ where n is the number of concurrent threads in the program under test. Moreover, its runtime performance is better than most of the race detection algorithms in the literature [72]. Finally, there are further improvements to FastTrack algorithm but tend to sacrifice precision [28].

2.5 Race Detection Tools

Race detection tools implement algorithms to detect races in applications. There are two types of tools commonly used to detect races. The first is a set of tools for detecting races statically, without running the program under test [55,67]. The other is for runtime race detection tools which perform race detection based on events at runtime [9,25,31,44,61]. Furthermore, there are two methods employed to get events information at runtime. A commonly used method is through instrumentation of program source code or binary. This injects callbacks which monitor the occurrence of events with time when the target program runs. The second approach is the use of special event signals implemented in virtual machines as in RoadRunner [25].

Race detection tools can be characterized based on three categories: precision, performance, software and hardware target. Precision is related to the guarantee of detecting races when indeed there are in a particular execution trace. Precise race detection tools rely on algorithms they implement. Tools which implement HB-based algorithms, like FastTrack and DJIT++, tend to be precise whereas Lockset-based tools are unsound. Runtime performance of race detection tools depends on the computational complexity of the race detection employed as well as the instrumentation technique used in the tool. Moreover, performance optimization of the tool in overall improves performance. Compile-time instrumentation as in [9,44,61], outperforms binary instrumentation mostly done by tools like Intel PIN [43] and DynamoRio [13].

One of the most successful tools is ThreadSanitizer [61]. It uses custom optimization and a combination of HB and Lockset race detection algorithms to achieve both runtime performance and more precision. In our proposed tools we aim at more precision and high performance. Moreover, FastTrack [24] is a precise (no false negatives) race detection algorithm for multithreaded applications originally implemented in the RoadRunner dynamic framework [25] for Java programs. RoadRunner instruments the bytecodes of the program and performs the runtime analysis to detect races. It also supports other race detection algorithms like Eraser [60].

Chapter 3

RELATED WORK

Our related work focuses on race detection for shared memory programming models: OpenMP tasks, Atomic Dataflow(ADF), and POSIX Threads. We also discuss the differences between related works and our proposed solutions.

3.1 *Race Detection for OpenMP*

Archer is an efficient tool for detecting data races in OpenMP programs between concurrent threads [9]. Through LLVM, it uses static analysis polyhedral techniques to ignore sequential code and instrument concurrent portion of the program. Then it uses runtime analysis to detect races in those parts by employing *ThreadSanitizer* [61] race detector in the background. In contrast, we detect determinacy races where ordering between concurrent components is missing. *Archer* may fail to detect such cases and it also misses concurrent tasks executed by the same thread. By building the happen-before relations on tasks rather than threads, we can catch these situations.

Determinacy race detection in [63] targets task-based programming models with *async*, *finish* and *future* constructs. There are works on detecting determinacy races in a very strict two-dimensional pipeline parallel program structures which restrict task dependency to at most two [20, 71]. Other works target determinacy races [22, 40, 57, 58] for structured parallelism programming models like X10 and Habanero. Most work targets data race detection [24, 36, 45, 60, 61] which manifest as a result of improper synchronization in programs.

DFinspec [44] proposes a technique for detecting output nondeterminism for Atomic Dataflow (ADF) [27] programs due to missing or improper ordering among tasks. It assumes that all concurrent portions of the program execute in atomic tasks. Unlike

ADF, in OpenMP tasks are not atomic, thus the proposed solution in DFinspec would not work on OpenMP programs. The Starsscheck tool [15] identifies inconsistencies in *pragma* annotations for programs written in Starss programs [37]. The tool verifies that the programmer correctly annotates the application by checking the input and output dependencies of tasks. By assuming that a task accesses shared memory through only input dependencies, it fails to detect concurrent tasks accessing shared memory locations that are not specified through input dependencies.

A closely related work [22] proposes an algorithm for detecting *determinacy* races for Cilk programs [11] in which a spawned thread may execute concurrently with parent or sibling threads. These threads may need proper synchronization for shared memory accesses. We target OpenMP tasks where a task becomes runnable when all its dependencies are satisfied. Vechev et. al [66] uses a static sequential analysis to verify determinism for task-based parallel programs by leveraging numerical abstractions. They locate code sections that can execute concurrently and check for dependent memory accesses between those sections.

Differently from the related work, we propose a technique for detecting *determinacy* races at runtime for OpenMP tasks which to our knowledge has not been explored before. Moreover, we do not explicitly check for inconsistency in *pragma* annotations as is done by Starsscheck [15]. We rather detect *determinacy* races which can be caused by missing necessary dependencies among tasks.

3.2 Nondeterminism Detection

Most of the recent research focus on detecting data races in structured multithreaded programs [22, 40, 52, 57]. Another closely related work by Feng and Leiserson [22] proposes an algorithm for detecting data races (named as *determinacy* races) for multithreaded applications developed in Cilk [11]. They target structured multithreaded programs in which a spawned thread may execute concurrently with parent or sibling threads and may need proper synchronization for shared memory accesses. On the other hand we target dataflow applications where a task becomes runnable

when it receives all incoming tokens. A similar work by Lee and Schardl presents two algorithms for detecting data races for Cilk programs that contain reducers and hyperobjects [40].

DDOS [30] detects nondeterminism in distributed systems where the source of nondeterminism might be related to network issues, timing variations in the operating system and message arrival order. In our case, the source of nondeterminism is due to missing data dependencies among computing entities. Devietti et al. [19] have a different approach to the problem of nondeterminism. They present techniques for ensuring determinism in multithreaded programs by suggesting hardware modifications such that the multithreaded applications execute deterministically with very low performance overhead. Their work does not address the nondeterminism problem in dataflow tasks on shared memory because the concept of threads is abstracted away by the dataflow runtime and scheduler.

Vechev et. al [66] propose a static analysis approach to verify determinism for multithreaded programs. Their approach targets task-based parallel programs to simplify reasoning while applying sequential analysis. With the help of numerical abstractions, they locate portions of code that can execute concurrently and check for dependent memory accesses between them and report nondeterminism. Since their approach is static, it tends to generate false alarms. Moreover their technique relies on numerical abstractions which are computationally expensive.

SingleTrack [59] is a dynamic analysis tool for verifying conflict freedom and external serializability of deterministically parallel multithreaded programs. This verification ensures deterministic execution of multithreaded programs but does not address nondeterminism as we are addressing in our tool.

We supplement the related work by defining the problem of output nondeterminism and propose a technique implemented as a tool to detect it in concurrent dataflow programming models that target shared memory systems. This nondeterminism bugs may be caused by missing or incorrect ordering of task dependencies that are used for ensuring certain ordering of task executions.

3.3 Race Detection for POSIX Threads

S. Hong and M. Kim surveyed a number of tools for detecting races in multithreaded programs [29]. Nevertheless we discuss a few tools related to our proposal.

ThreadSanitizer [61] is an industrial-level and open-source race detection tool for Go, and C/C++ concurrent applications for 64-bit architectures and it is accessible through GCC and LLVM/Clang [39] using a compiler flag. It instruments the program under compilation by identifying shared memory and synchronization operations and injecting race detection runtime callbacks. The instrumented executable is then run on a target platform for detecting races. *ThreadSanitizer* has been successful mainly for two reasons. First, it uses a hybrid of *happens-before* and *lockset* algorithms to improve its precision. Second, it uses 64-bit architectural capability to store race detection meta-data called *shadow memory* for performance and memory efficiency. The authors of *ThreadSanitizer* claim that extending it for 32-bit applications is *unreliable and problematic* [6]. We adopt its instrumentation part to implement a race detection tool for 32-bit POSIX Threads embedded systems applications.

Intel Inspector XE [31] and *Valgrind DRD* [7,50] are tools similar to *ThreadSanitizer* [61] in detecting data races for C/C++ concurrent programs. Additionally, despite running on native hardware, they have limited support for emerging platforms like the 32-bit ARM architectures; a target for one of our proposed projects.

Olszewski et al. propose Aikido; a framework and a tool for detecting data races on a general multithreaded program [52]. It uses both hardware support, through a hypervisor, and dynamic binary rewriting techniques to detect shared data at a coarse granularity (a shared page). Aikido needs a hypervisor and instrumentation of the shared pages using the DynamoRIO [13] instrumentation platform and this relatively slows down the program. Differently from Aikido, we propose a solution for detecting races on the target embedded hardware.

3.3.1 Race Detection of POSIX Threads in Embedded Systems

Zeus Virtual Machine[®] Dynamic Framework [68, 69] is a hardware-agnostic platform which contains tools for detecting runtime data races for kernel and user-space multithreaded applications. These tools rely on virtualization and may have overhead challenges and may abstract away real target system interactions with external peripherals like sensors. Moreover, these tools are proprietary and no much relevant information is in the literature.

Most of the related solutions for detecting data races in embedded systems do target low end interrupt based, non-multithreaded embedded systems [16, 64, 65, 70]. Therefore, these solutions can not directly apply to the POSIX multithreaded software for embedded systems. Moreover, Keul [35] and Chen [17] use static analysis techniques for race detection in interrupt-driven systems applications. Unfortunately, these techniques do not capture the runtime behavior of the program. Therefore, they fail to infer many of execution patterns which would otherwise result in data races.

Chapter 4

TASKSANITIZER: RUNTIME DETERMINACY RACE DETECTION FOR OPENMP TASKS

4.1 Introduction

OpenMP 3.0 introduced shared memory task execution model [2] in which programmers specify computations in units called *tasks*, which can be executed by concurrent threads. In OpenMP 4.0 [3], a programmer can specify execution order between tasks through *in* and *out* data dependencies, where a succeeding task waits for the completion of the preceding task's execution. Even though programmers have more flexibility to express various types of parallelism with the new tasking attributes, these new features can introduce subtle bugs if the operational semantics and scheduling policy of the OpenMP runtime are not reasoned about. One of such concurrency bugs is a *determinacy race* which occurs when concurrently executing entities access the same memory location without specified ordering between them and at least one access is a write to that memory location [22, 40, 57, 58]. As a result, a program with determinacy races may produce different final output results at different runs on the same input [44]. Determinacy races are possible if the programmer does not specify necessary dependency between concurrent tasks which access the same memory locations. Since there is no specific order defined by the programmer, the scheduler is free to execute the tasks in any order or concurrently.

The existing state-of-the-art runtime race detection tools for OpenMP such as Archer [9] – and general race detectors [29]– check for proper locking in programs which protects shared memory objects but can fail to detect determinacy races which stem from improper ordering of executions. Protecting memory accesses with critical

sections or other explicit locking is not sufficient to avoid determinacy races. Rather, proper ordering of the executing entities is essential to avoid undesirable nondeterminism in OpenMP programs for correctness.

We present an algorithm to detect determinacy races in OpenMP programs by utilizing the concept of OpenMP tasks and their dependencies. Unlike the state-of-the-art race detection tools [9] that rely on *happens-before* model at thread level, we apply *happens-before* model at task level, which provides the advantage of reducing randomness due to scheduling. We implement our algorithm as an open source tool based on compile-time instrumentation through LLVM [39] compiler pass to instrument shared memory accesses in the program. The tool uses the OpenMP Performance Tools API (OMPT) [21] to monitor OpenMP-related events such as task creation, scheduling, and execution. In summary, the main contributions of this research are:

- A formal definition of the determinacy races and a technique for detecting such races in OpenMP tasks. To our knowledge, no prior work has been done for detecting determinacy races in OpenMP tasks with mixed structures of critical and non-critical sections.
- Determinacy race detection tool for OpenMP called *TaskSanitizer* [46].
- Evaluation of our method using micro-benchmark applications and comparison of results against a race detection tool for OpenMP programs.

4.2 *Background in OpenMP Tasks*

Explicit tasks in OpenMP can be created with the construct *omp task*, which is readily available since OpenMP 3.0 [2]. For each task, OpenMP creates a work block which includes a sequence of program statements and the data environment. This block is set aside to be executed by a thread until the runtime schedules it. Starting with OpenMP 4.0 [3], it is possible to specify execution order among explicit tasks using the *depend* clause, where a programmer specifies input and output data dependencies between

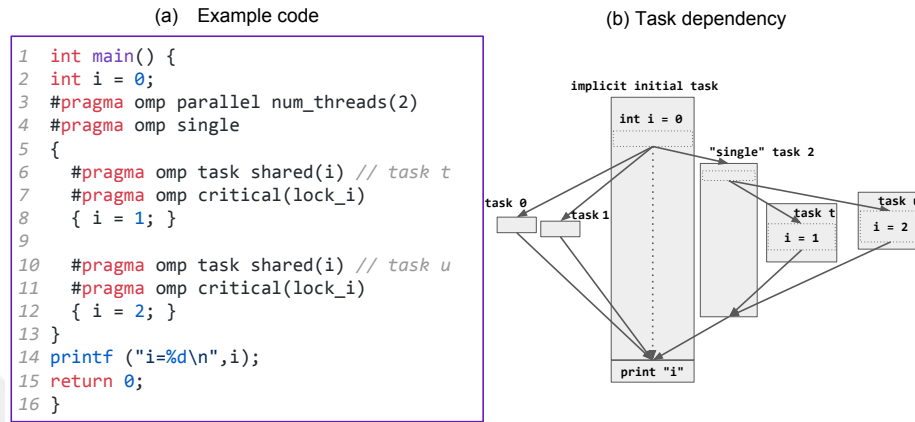


Figure 4.1: OpenMP example illustrates explicit and implicit tasks and their logical flow dependency between tasks. The code example has a determinacy race.

tasks. A collection of tasks through dependencies forms an implicit task dependency graph in which a task is not runnable until all its dependencies are satisfied. The runnable tasks can then be scheduled by the OpenMP runtime. If two or more tasks are simultaneously runnable at a given point in time, they can execute in any order or concurrently.

Every part of an OpenMP program executes in a task assigned to one or more threads. For example, implicit tasks can be generated at parallel regions with the OpenMP *parallel* construct and each implicit task is executed to completion by one thread in the thread group of the parallel region [2]. Figure 4.1 shows a simple OpenMP program, where a default implicit task is created as part of the main program. This task then creates two implicit tasks through the parallel region at line 3. One of these tasks executes the *single* region at line 4, which creates two explicit tasks t and u at lines 6 and 10, respectively. Both of these tasks have critical sections, in which they set different values to a shared variable i . This example has a determinacy race which is explained in detail in Section 4.3.

4.3 Determinacy Race Detection

In this section, we first define determinacy races and present motivation on detecting them with the help of an OpenMP example. Then, we formally define a task with its operations and we devise happens-before (HB) relations between these operations for capturing partial ordering among them. Finally we use the defined HB relations to present our algorithm for detecting determinacy races.

4.3.1 Definition and Motivating Example

Determinacy race occurs between two tasks if the following two conditions are satisfied: (i) there is no ordering between these tasks enforced by task dependency, and (ii) both tasks access a common shared memory location and at least one access is a write. If simultaneously runnable tasks 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.

Many runtime race detection algorithms [24, 60, 61] do not take the notion of dependency into account. They monitor proper synchronization of threads on memory accesses to detect races. In this work, we monitor the proper ordering of tasks and critical sections to ensure that different possible ordering of critical sections in these tasks always generate a single, deterministic final program state. This helps the programmer to notice if nondeterminism was not intentional.

We have provided a simple OpenMP program in Figure 4.1, where there is no specified dependency between tasks t and u . As a result, their critical sections can execute in any order and thus the final result for i can either be 1 or 2 despite the fact that accesses to the shared variable are protected by a common lock. Unless the developer intends the program to behave as such, only one deterministic result is expected. The same issue arises if one of the tasks reads the value of i in a critical region and the other task writes to i . It is worth noting that in a typical program these two tasks might have been created in separate function calls, thus the critical sections may be well far apart from each other and can be easily overlooked.

4.3.2 Formalizing Task Operations

In order to establish HB relations and set up rules between tasks for detecting determinacy races, we first define relevant task operations:

- `create(t,u)`: task t creates task u .
- `wait(t,u)`: task t awaits termination of task u at *taskwait* or at a barrier.
- `read(t,mem)`: task t reads value from shared memory location mem .
- `write(t,mem,v)`: task t writes value v to shared memory location mem .
- `out(t,u,x)`: signifies dependency from task t to task u through storage location x . Task t is the predecessor and u is the dependent task.
- `in(u,t,x)`: signifies dependency from task t to task u through storage location x . Task u becomes runnable once t completes its execution.

Having defined task operations, we elaborate on shared memory accesses and associate them to segments of a task, rather than the task itself. We define a task as an enclosed sequence of unique *tasksegments* and *synchronization* operations executed together, as shown in Figure 4.2. A tasksegment is a sequence of consecutive shared memory accesses between two synchronization operations in a task. Therefore, a synchronization operation in a task ends the current tasksegment and a new tasksegment starts at the next shared memory access operation in the task after the synchronization operation. We define synchronization operations as operations which trigger execution among tasks and are `create`, `wait`, `out`, and `in`. For example, Figure 4.3 shows three tasks (a parent and two child tasks) but contains four tasksegments. In other words, in our formal task operations we differentiate the code bodies (e.g. tasksegment $s1$ and tasksegment $s4$) that result from imperfectly nested tasks. Since this is necessary to establish HB relations, we revise the shared memory access operations as follows:

$$\begin{aligned} \mathbf{task}_t &\equiv \left[\begin{array}{l} \text{create}(t,u), \text{wait}(t,u), \text{out}(t,u,x), \\ \text{in}(u,t,x), \text{taskseg}(t,s) \end{array} \right]^+ \\ \mathbf{taskseg}_{(t,s)} &\equiv \left[\begin{array}{l} \text{read}(t,s,\text{mem}), \text{write}(t,s,\text{mem},v) \end{array} \right]^+ \end{aligned}$$

Figure 4.2: Defining a task as a sequence of tasksegments (taskseg) and synchronizations

- $\text{read}(t,s,\text{mem})$ shared memory access that appears in tasksegment s where task t reads a value from shared memory location mem .
- $\text{write}(t,s,\text{mem},v)$ shared memory access that appears in tasksegment s where task t writes value v to shared memory location mem .

4.3.3 Happens-Before Relations Between Task Operations

For partial ordering of operations in an OpenMP program, we use happens-before (HB) ordering of events [38] by employing dependency among synchronization operations. Happens-before relation is a transitive-closure relation. For given three operations a , b , and c if there is an HB relation from a to b and from b to c , then there is an HB relation from a to c . We will infer this relation while categorizing HB relations between tasks operations. We use symbol \prec to refer to an HB relation in general and use \prec_π to refer to an inferred HB relation due to transitive-closure property.

$$a \prec b \wedge b \prec c \rightarrow a \prec_\pi c$$

We identify four types of HB edges among operations between tasks. These are (i) an HB relation among memory operations performed within a tasksegment; (ii) between a task and its child task through *create*; (iii) relation between *out* and *in* dependency operations; and (iv) relation at *wait* operation. We then use these HB relations to infer HB relations among tasksegments in tasks.

1. HB by program order: This is the basic type of HB relation where program operations within a tasksegment are ordered according to their execution sequence. Similarly, tasksegments within a task are ordered by program order.

2. HB relation by task dependency: If tasks t and u have a commonly specified data dependency such that u has an input dependency from t , then all tasksegments – as well as their enclosing memory operations – in t happen-before all tasksegments in u .

$$\frac{out(t, u, x) \prec in(u, t, x)}{\forall_{taskseg(t,a)} \forall_{taskseg(u,b)} taskseg(t,a) <_{\pi} taskseg(u,b)}$$

3. HB relation between a task and its child task: tasksegments of a task which execute before creating a child task happens-before the tasksegments executed in the created child task. For two tasks t and u :

$$\frac{create(t, u)}{\forall_{taskseg(t,a)} taskseg(t,a) <_{\pi} create(t, u) \rightarrow \forall_{taskseg(u,b)} taskseg(t,a) <_{\pi} taskseg(u,b)}$$

4. HB relation at taskwait and barrier synchronizations: The last operation of a child task happens before the *taskwait* or implicit barrier synchronization operation of the parent task. Therefore, all tasksegments of such task have HB relation with subsequent tasksegments of the parent task after the wait operation is completed.

$$\frac{wait(t, u)}{\forall_{taskseg(t,a)} wait(t, u) <_{\pi} taskseg(t,a) \rightarrow \forall_{taskseg(u,b)} taskseg(u,b) <_{\pi} taskseg(t,a)}$$

We use example Figure 4.3 to illustrate the four categories of HB relations. The memory operations at lines 11 and 12 belong to the same tasksegment $s3$ and thus are ordered by program order. Moreover, there is an HB relation between memory operations at lines 4 and 7 because their corresponding tasksegments have an HB relation through task creation synchronization operation as task t executing the single

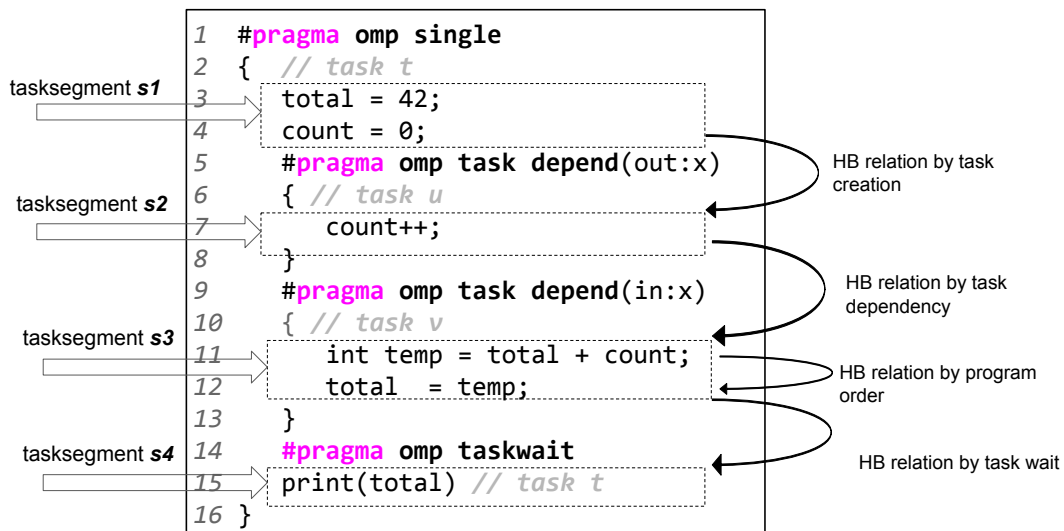


Figure 4.3: Example with four categories of HB relation among operations of tasks

region creates an explicit task u at line 5. Moreover, all operations in tasksegment s_2 happen-before all operations in tasksegment s_3 because of specified dependency between tasks u and v . Finally, memory operations in tasksegments s_3 and s_4 happen before the print statement in tasksegment s_4 because of the *wait* synchronization operation at line 14. Without *taskwait*, we would not be able to establish an HB relation between s_4 with s_2 or s_3 .

4.3.4 Determinacy Race Detection Algorithm

Algorithm 4 provides pseudo-code for determinacy race detection between any two memory operations (α and β) in an OpenMP program. Between lines 4 and 9, it retrieves information of the operations: their task identifiers (IDs), tasksegment IDs as well as the memory addresses they accessed. Then at line 10, the algorithm checks if the operations access the same memory location and belong to two different tasks and tasksegments. At line 11, it checks if the corresponding tasksegments do not have an HB relation as inferred using the four HB types from Section 4.3.3. If there is no HB, then it reports a determinacy race bug if one operation is a *write* and the other a *read* at lines 12 and 13. In the case that they both are *write* actions, it reports a

determinacy race if they are not commutative (lines 14 - 16).

Algorithm 4 Detecting determinacy race between two shared memory operations

```

1: procedure CHECKDETERMINACYRACE( $\alpha$ ,  $\beta$ )
2:   Input:  $\alpha$   $\triangleright$  a shared memory operation
3:   Input:  $\beta$   $\triangleright$  another shared memory operation
4:    $t \leftarrow \text{getTaskID}(\alpha)$ 
5:    $u \leftarrow \text{getTaskID}(\beta)$ 
6:    $seg_1 \leftarrow \text{getTasksegmentID}(\alpha)$ 
7:    $seg_2 \leftarrow \text{getTasksegmentID}(\beta)$ 
8:    $mem_1 \leftarrow \text{getMemoryAddress}(\alpha)$ 
9:    $mem_2 \leftarrow \text{getMemoryAddress}(\beta)$ 
10:  if  $mem_1 = mem_2$  and  $t \neq u$  and  $seg_1 \neq seg_2$  then  $\triangleright$  on different tasks
11:    if not  $\text{HappensBefore}(seg_1, seg_2)$  then  $\triangleright$  check if no HB
12:      if  $\text{isWrite}(\alpha) \neq \text{isWrite}(\beta)$  then  $\triangleright$  one write, one read
13:        REPORTBUG( $\alpha$ ,  $\beta$ )
14:      else if  $\text{isWrite}(\alpha)$  and  $\text{isWrite}(\beta)$  then  $\triangleright$  both write
15:        if not  $\text{isCommutative}(\alpha, \beta)$  then  $\triangleright$  check commutativity
16:          REPORTBUG( $\alpha$ ,  $\beta$ )
17:        end if
18:      end if
19:    end if
20:  end if
21: end procedure

```

Detecting Commutative Operations:

Shared memory accesses can result in falsely detected determinacy races if these accesses involve in commutative arithmetic operations between same-lock critical sections. Two concurrent arithmetic operations on a shared memory location are commutative if their order of execution does not alter the final value produced. For

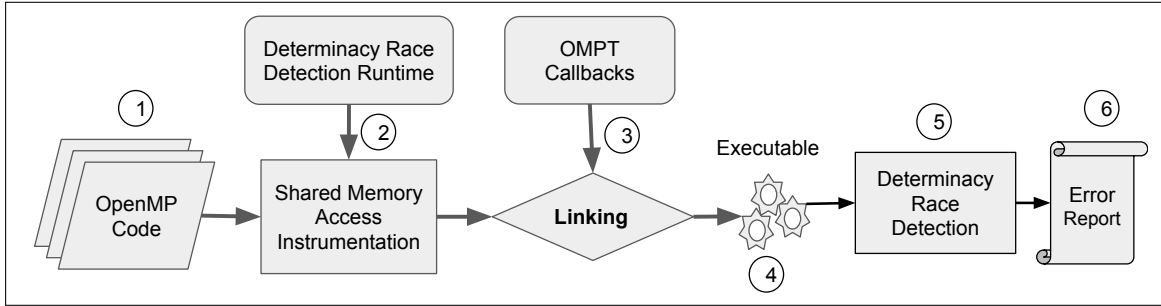


Figure 4.4: Showing implementations of TaskSanitizer: architecture and tool flow

example, if $var += temp1$ and $var -= temp2$ are in two different same-lock critical sections, then re-ordering them does not affect the final value of var . Thus in line 16 of Algorithm 4, we use the formalization of commutativity operation detection proposed in [44] to identify such memory actions and do not report determinacy races on them.

4.4 Implementation

As shown in Figure 4.4, we implement our method as a tool that has three main parts (i) instrumentation; (ii) inferring happens-before relation between program operations; and (iii) determinacy race detection at runtime.

1. **Instrumentation:** We instrument an OpenMP program source code at compile-time through LLVM / Clang infrastructure [39]. The instrumentation injects our determinacy race detection runtime callbacks, which implement Algorithm 4, in step ②. We customize the shared memory instrumentation module of ThreadSanitizer [61] to identify shared memory operations and associated source code line numbers and functions for traceability in case of determinacy races. Moreover, we identify and store program statements which are in critical sections. These are later used by our algorithm to detect commutative operations on potential determinacy races where our tool does not report them if the ordering of those critical sections does not alter the final output.

2. **Constructing HB relations:** To capture HB ordering between tasks and operations, we implemented a module that uses the OMPT interface [21] in step ③ of Figure 4.4 to register callbacks which capture synchronization operations. First, we locate the implicit tasks as well as explicit tasks defined using the tasking clause for specifying the ordering of program events. Second, task dependencies through *depend* clause as well as custom synchronization idioms such as locks and barriers are located to reason about the happens-before ordering. Finally, we use these operations to infer HB relations between task operations. Moreover, we assign a unique identification to each task and tasksegment at creation, during program execution. This has three advantages (a) Unique ID differentiates different instances of the same task code block or tasksegment executed at different times. (b) A task may run to completion by a single thread or its parts may be scheduled to different threads. Similarly two concurrent tasks may be executed by the same thread. Our approach is transparent from threads, hence regardless which thread(s) execute a task, a unique ID preserves its dependencies with other tasks and avoids false determinacy race alarms. (c) Each tasksegment has the same set of HB meta-data, as opposed to each memory operation, thus unique ID of the tasksegment is used to retrieve HB metadata for each of its memory operations.
3. **Runtime determinacy race detection:** As shown in Figure 4.4, we link the library we implemented at step ③ to produce the instrumented executable binary, which executes at step ④. At step ⑤ relevant program events are captured at runtime and detection is performed and a bug report is generated in step ⑥. The tool reports a pair of line numbers where a common shared memory location was accessed by concurrent tasks. This pair is helpful for the developer to revisit the source code and eliminate determinacy races. This module also implements the technique proposed in [44] to check if operations with determinacy races are commutative as they execute in critical sections of the same lock given that their execution order does not affect final output of

the program to reduce false positives.



Chapter 5

DFINSPEC: NONDETERMINISM DETECTION FOR ADF

5.1 Introduction

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 required by the task 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 [27], or synchronization primitives such as internal locks [32,53]. 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.

5.2 Motivation

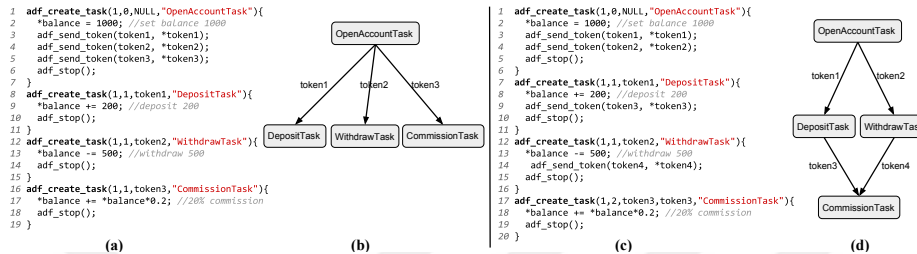


Figure 5.1: A simple banking ADF example (a) code with nondeterministic behavior between *WithdrawTask*, *DepositTask* and *CommissionTask*. The flow graph (b) shows dependency between these tasks indicating that *DepositTask*, *WithdrawTask* and *CommissionTask* are concurrent. The code in (c) shows the correct implementation of the code in (a). The flow graph (d) shows the correct dependency which ensures that *CommissionTask* executes last, thus ensuring a deterministic execution of the banking operations. *DepositTask* and *WithdrawTask* can execute in any order because they can not affect the final output.

In dataflow applications the output nondeterminism can occur when a programmer misses necessary dependency between concurrent tasks. We provide a simple bank simulation example in Figure 5.1 to motivate the problem we are addressing. The idea behind this example is the simulation of bank operations using tasks where the intended last operation is the calculation of the commission using a given rate and the current balance in the bank account. We show that if the programmer misses necessary data dependency among the tasks, the program produces nondeterministic final results. To understand this missing dependency we provide both the wrong and correct implementations.

Figure 5.1(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 Figure 5.1(c). Instead the programmer mistakenly implements the ADF program that creates four tasks to perform these bank operations as in 5.1(a) and 5.1(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 Figure 5.1(b)). In ADF, by default, a task is *atomic*, or non-preemptive [27]. 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 Figure 5.1(c) at lines 9 and 14. This ordering creates data dependency among these tasks as shown in Figure 5.1(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.

5.3 Proposed Approach

We develop an algorithm for detecting nondeterminism in dataflow applications running on shared memory. Our algorithm uses *happens-before* relations that we devise for partially ordering different actions performed by the tasks. In this section we provide a formal definition of the happens-before relations and necessary rules to capture nondeterminism, and then state our detection algorithm.

5.3.1 Defining Actions of a Dataflow Application

To detect nondeterminism in dataflow applications running on shared memory we need to identify and model all relevant program actions performed by a task. The set of actions, denoted as *Actions*, can be used to represent the relevant operations performed by a dataflow program. We first define the list of actions and then set up rules for those actions that are necessary to capture the ordering of events in an execution trace of a dataflow application. These actions are:

- $start(tid)$: a task with a unique identification tid starts execution. This is the first action of a task once scheduled by the runtime.
- $send(tid, token, buff)$: a task tid sends a token value $token$ to another task through output token buffer with identifier $buff$.
- $receive(tid, token, buff)$: a task tid receives a token value $token$ from another task through input token buffer $buff$.
- $read(tid, addr)$: a memory access where task tid reads a value from a memory address $addr$.
- $write(tid, addr, val)$: a memory access where task tid writes value val to a memory address $addr$.
- $end(tid)$: a task tid terminates execution. This is the last action of a task before its termination.

In the subsequent discussions, we will often omit the parameters of the action instances performed by a task and we will refer to individual parameters of an action in the form of $action.parameter$. For example, to refer to tid argument of an action instance $\beta = send(tid, token, buff)$ we will use $\beta.tid$.

We classify tasks as *reader* or *writer* tasks based on their accesses to a shared memory location. A task is *reader* of $addr$ if all of its accesses to $addr$ are read operations. On the other hand, a task is *writer* of address $addr$ if it writes at least once to that address. This classification is used in Section 5.3.3 to define rules necessary for detecting output nondeterminism. In the subsequent discussions, the write action of a task to a memory address denotes its last write action to that location and, for simplicity, the read action is the first read action to the memory location.

5.3.2 Happens-before Relation of Task Actions

Having defined the actions, we now provide a definition of *happens-before* relation between the actions of tasks. The happens-before relation is a partial ordering of actions performed within a task or between tasks. This gives a formal way of setting an order between actions in an execution trace. Given any two actions α and β , we

say that action α *happens-before* action β if there is a partial order between them and this order puts α before β in an execution. If α and β are performed by the same task, we say that there is a happens-before relation between them ensured by *program order*. If these actions are performed by different tasks, their happens-before relation is ensured by data dependencies between these tasks. Actions of task A happen before actions of task B if A sends a token to B which resumes execution after receiving the token through an input token buffer.

To formally define a *happens-before* relation between the actions performed by tasks, we adopt a notation presented in [47]. The happens-before relation, denoted as $\prec \subseteq Action \times Action$ is a binary relation that is irreflexive and transitive. For a given finite execution trace of actions $\pi = \alpha_0.\alpha_1\dots\alpha_n$ we denote $\alpha <_{\pi} \beta$ if action α happens before action β in π . The necessary rules to capture the happens-before relation among actions in an execution trace of a dataflow program are as follows:

- **Start:** This rule orders the *start* action of a task before all the other actions performed by the same task. The *start* action of a task tid happens-before all actions subsequently performed by that task. Formally, this rule is summarized in formula (1) in Figure 5.2.
- **Token:** This rule sets the ordering between *send* and *receive* actions of the same token through the same data buffer by two different tasks. It specifies that the *send* action of a token happens-before the *receive* action of the same token (rule 2 in Figure 5.2).
- **Read/Write:** Specifies the ordering of memory actions on a given memory location. Memory actions in a task are ordered by program order. For actions in different tasks, the order is preserved if there is a happens-before relation between these tasks.
- **End:** This rule captures the fact that all actions in a task are ordered to happen before the task terminates.

- (1) Start:

$$\frac{\alpha = \textit{start}, \beta \in \{\textit{send}, \textit{receive}, \textit{read}, \textit{write}, \textit{end}\}, \alpha.\textit{tid} = \beta.\textit{tid}}{\alpha \prec \beta}$$
- (2) Token:

$$\frac{\alpha = \textit{send}, \beta = \textit{receive}, \alpha.\textit{buff} = \beta.\textit{buff}, \alpha.\textit{token} = \beta.\textit{token}}{\alpha \prec \beta}$$
- (3) Read/Write:

$$\frac{\alpha, \beta \in \{\textit{read}, \textit{write}\}, \alpha.\textit{addr} = \beta.\textit{addr}, \alpha <_{\pi} \beta}{\alpha \prec \beta}$$
- (4) End:

$$\frac{\alpha \in \{\textit{send}, \textit{receive}, \textit{read}, \textit{write}, \textit{end}\}, \beta = \textit{end}, \alpha.\textit{tid} = \beta.\textit{tid}}{\alpha \prec \beta}$$

Figure 5.2: happens-before formulas for capturing the ordering of actions in a dataflow application.

Motivating example revisited We revisit the motivating example in Figure 5.1 to elaborate the rules summarized in Figure 5.2. There is data dependency between *OpenAccountTask* and *CommissionTask* which forms a happens-before relation between these tasks. As rule 2 implies, the action to send *token1* by *OpenAccountTask* happens before the action to receive *token1* by *CommissionTask*. Similarly there is a happens-before relation between *OpenAccountTask* and *DepositTask* since *OpenAccountTask* sends *token2* and *DepositTask* receives it. This is also true between *OpenAccountTask* and *WithdrawTask* on *token3*. Hence, by the transitive closure of happens-before relation, all memory access actions to the variable `balance` in *OpenAccountTask* do happen before those in the remaining three tasks. Unfortunately this is not the case between any pair of the tasks *CommissionTask*, *WithdrawTask* and *DepositTask* since there is no specified data dependency edge between these tasks and thus no happens-before relation among them.

5.3.3 Output Nondeterminism Detection Rules

Using the happens-before rules we modeled in the previous section, we propose an algorithm for detecting nondeterminism in a dataflow application on shared memory. A program developed on a dataflow programming model on shared memory is

nondeterministic if the following two criteria hold:

1. There exists two *writer* tasks with actions $\alpha = \mathit{write}(tid_1, \mathit{addr}, \mathit{val}_1)$, and $\beta = \mathit{write}(tid_2, \mathit{addr}, \mathit{val}_2)$, respectively, which write different values ($\mathit{val}_1 \neq \mathit{val}_2$) to the same memory location addr . Or, there exists a *reader* task and a *writer* task with actions $\alpha = \mathit{read}(tid_1, \mathit{addr})$, and $\beta = \mathit{write}(tid_2, \mathit{addr}, \mathit{val}_2)$, respectively, which access the same memory location addr .
2. The two tasks, $\alpha.tid_1$ and $\beta.tid_2$, which performed the actions in (1) do not have a happens-before relation; $\alpha \not\prec \beta$ and $\beta \not\prec \alpha$.

Our algorithm detects nondeterminism on a shared memory location addr when there are two memory access actions in a dataflow execution trace π performed by two different tasks which satisfy the two conditions above. Formally, given two write actions $\alpha = \mathit{write}(tid_1, \mathit{addr}, \mathit{val}_1)$ and $\beta = \mathit{write}(tid_2, \mathit{addr}, \mathit{val}_2)$ with $tid_1 \neq tid_2$ and $\mathit{val}_1 \neq \mathit{val}_2$, or a read and write actions; $\alpha = \mathit{read}(tid_1, \mathit{addr})$ and $\beta = \mathit{write}(tid_2, \mathit{addr}, \mathit{val}_2)$ with $tid_1 \neq tid_2$, then there is a nondeterminism error on addr if neither of the following rules is satisfied:

1. $\alpha <_{\pi} \mathit{send}(tid_1, \mathit{token}, \mathit{buff})$ and $\mathit{receive}(tid_2, \mathit{token}, \mathit{buff}) <_{\pi} \beta$, or
2. $\beta <_{\pi} \mathit{send}(tid_2, \mathit{token}, \mathit{buff})$ and $\mathit{receive}(tid_1, \mathit{token}, \mathit{buff}) <_{\pi} \alpha$

5.3.4 Nondeterminism Detection Algorithm

By applying the rules in section 5.3.3 to detect nondeterminism, we develop a nondeterminism detection algorithm summarized in Algorithm 1. The algorithm loops through all the actions in the execution trace π of a dataflow program. When the algorithm encounters a *start* action in π , it creates a new happens-before set of identifiers of tasks which happen before this task. When the action is a receive action, the happens-before set of the receiver task is expanded by adding all task identifiers from

the happens-before set of the task which sent the token. This preserves the happens-before transitive closure of actions. When a memory access action is inspected in π , the algorithm checks for output nondeterminism between concurrent memory actions inspected so far to the associated memory location. The check applies the rules in section 5.3.3 with the help of transitive happens-before set of the task that performed this action. In the following paragraphs we walk through the algorithm pseudocode and provide details.

The looping through all actions in the execution trace π starts at line 9. At line 10, it checks if the current action under test represents the start of a new task in which case the algorithm constructs a new set containing the identifier of the task itself at line 11. We refer to this set as a *HB* set. At line 12 it is stored in a list of HB sets for future retrievals. At line 13 if the current action is a *send* action, it is stored in a tokens set (line 14). This information is later retrieved, at lines 15 and 16, when a *receive* action with the same token value and the buffer *ID* is processed from π . Since this matches the sender and the receiver of the token, the HB set of the receiver task of the token inherits the HB set of the sender at lines 17 and 18. This constructs a happens-before transitive closure of actions performed by these two tasks.

Starting at line 19, the rest of the algorithm concentrates on nondeterminism detection for a memory access action. There are two conditions associated with each memory access action in the execution trace π : (a) a first read action by a reader task which performs only reads on the memory location; (b) the last write action by a writer task which performs at least one write to the memory location. In case of (a), among all concurrent tasks actions to the memory location before are retrieved at line 20, the corresponding writer and reader tasks that do not have a happens-before relation with the current task performing the action are identified and the values they wrote are compared with the current value at lines 23 - 25. Nondeterminism is reported, at line 24, if the current writing task is different than the previous writing task, the values written by these tasks are different and the tasks involved do not

have a happens-before relation between them. In case of (b), there is nondeterminism between the reader task of $\beta.addr$ and all concurrent writer tasks which do not have HB relation with that task (lines 28 - 31). Similarly, there is nondeterminism in the case that β is a read action with all concurrent writes which do not have HB relation with the performing task. After these checks, the relevant information about the current memory access is saved for future output nondeterminism checking at line 33.

Algorithm 1 has worst-case and best-case computation complexity of $O(n^2)$ and $\Omega(n)$, respectively, where n is the number of actions in the execution trace. A possible scenario for the worst case is when there are n writes to a single address by different n tasks. Moreover, the best case is when there are no tokens involved and no shared memory accesses. There are a number of technical improvements and optimization to this algorithm. We describe some of them in Section 5.4.

Algorithm 1 Detecting output nondeterminism

```

1: procedure CHECKNONDETERMINISM(  $\pi$  )
2:   Input:  $\pi$   $\triangleright$  an execution trace
3:   Var  $\beta, \alpha$   $\triangleright$  actions
4:   Set  $\gamma_{addr}$   $\triangleright$  concurrent memory accesses to  $\beta.addr$ 
5:   List  $MemACsets$   $\triangleright$  concurrent memory accesses to addresses
6:   Set  $HBsets$   $\triangleright$  happens-before sets for tasks
7:   Set  $HB_{\beta}$   $\triangleright$  HB set for a task of action  $\beta$ 
8:   List  $Tokens$   $\triangleright$  stores tokens already sent
9:   for  $\beta$  in  $\pi$  do
10:    if  $\beta$  is start then
11:       $HB_{\beta} \leftarrow \{\beta\}$ 
12:       $HBsets \leftarrow HBsets \cup \{HB_{\beta}\}$ 
13:    else if  $\beta$  is send then
14:       $Tokens.insert(\beta)$ 
15:    else if  $\beta$  is receive then

```

```

16:          $\alpha \leftarrow Tokens.get(\beta.token)$ 
17:          $HB_{sender} \leftarrow HBsets.get(\alpha)$ 
18:          $HB_{\beta} \leftarrow HB_{\beta} \cup HB_{sender} \cup \{\alpha\}$ 
19:     else if  $\beta$  is memory access then
20:          $\gamma_{addr} \leftarrow MemACsets.get(\beta.addr)$ 
21:         for  $\alpha$  in  $\gamma_{addr}$  do
22:             if  $\beta$  is write and  $\alpha$  is write and  $\beta.tid \neq \alpha.tid$  then
23:                 if  $\beta.val \neq \alpha.val$  and  $\alpha.tid \notin HB_{\beta.tid}$  then
24:                     REPORTNONDETERMINISM( $\beta, \alpha$ )
25:                 end if
26:             else if  $\beta$  is write and  $\alpha$  is read OR
27:                  $\beta$  is read and  $\alpha$  is write then
28:                 if  $\beta.tid \neq \alpha.tid$  and  $\alpha.tid \notin HB_{\beta.tid}$  then
29:                     REPORTNONDETERMINISM( $\beta, \alpha$ )
30:                 end if
31:             end if
32:         end for
33:          $\gamma_{addr} \leftarrow \gamma_{addr} \cup \{\beta\}$ 
34:     end if
35: end for
36: end procedure

```

5.3.5 Commuting Tasks

Our proposed output nondeterminism detection algorithm fails to capture precisely all concurrent action patterns that lead to output nondeterminism in a dataflow program on shared memory by two concurrent writer tasks. One possible case is that it fails to capture commuting operations by commutative tasks. We give a simple definition to identify commutative tasks and propose a solution to eliminate the nondeterminism errors reported by our algorithm.

Two concurrent writer tasks are *commutative* if a change in their execution order does not change the final output of the program. As an example consider two tasks, tid_1 and tid_2 , incrementing a value at a memory location `addr`. As long as these tasks are atomic the execution order between these tasks does not matter. Our algorithm for detecting output nondeterminism proposed in the previous section has a limitation because our definition of happens-before relation for a given execution does not identify commuting writer tasks and therefore the algorithm flags them as a source of output nondeterminism because the values written by each task to the memory location is different. However this is not a real bug because the final program output is the same regardless of task execution order.

To overcome this limitation and reduce false alarms, we provide a simple definition to identify commutative writer tasks. Consider two tasks with IDs tid_1 and tid_2 which both modify a shared memory address `addr` by applying a sequence of one of the simple reduction operations¹. Assume that the task tid_1 performs a sequence, say S_1 , of n such operations whose one of operands is `addr` and each operation in this sequence can be denoted as α_i for $1 \leq i \leq n$. Similarly, tid_2 performs a sequence S_2 of m reduction operations of the same type whose one of operands is `addr` and each operation can be denoted as β_j for $1 \leq j \leq m$ as shown in Listing 5.1.

Listing 5.1: Sequences of reduction operations which involve `addr` as operand. S_1 belongs to task tid_1 and S_2 to tid_2 .

$$\begin{aligned} S_1 &= \alpha_1 \ \alpha_2 \ \alpha_3 \ \dots \ \alpha_n \\ S_2 &= \beta_1 \ \beta_2 \ \beta_3 \ \dots \ \beta_m. \end{aligned}$$

Then the writer tasks tid_1 and tid_2 commute on shared memory `addr` if these two sequences only contain a commutative reduction operation, thus these operators can be performed in any order without affecting the final result of `addr`. One way to verify that the basic reduction operations involving `addr` as operand from both S_1 and S_2 sequences are commutative is to symbolically execute them combined in two

¹A reduction operation can be one of the following operations: `+`, `-`, `*`, `/`, `&`, `|`, `^`, `&&`, `||`, `min` and `max`.

alternatives: operations on **addr** from tid_1 followed by those in tid_2 , and vice versa, as shown in the following expression:

$$\alpha_1\alpha_2\alpha_3\dots\alpha_n\beta_1\beta_2\beta_3\dots\beta_m \equiv \beta_1\beta_2\beta_3\dots\beta_m\alpha_1\alpha_2\alpha_3\dots\alpha_n$$

Once Algorithm 1 on section 5.3.4 detects an output nondeterminism for write-write cases (lines 23 - 25), we traverse through the sequence of operations on each task, starting from the beginning of the task up to the memory access which showed nondeterminism, and identify the operations which use the memory address as operand and construct the sequences S_1 and S_2 . Next, arithmetic operations in S_1 commute with those in S_2 if the following rule holds.

$$(1) \forall_{\alpha_i \in S_1} \forall_{\beta_j \in S_2} \alpha_i \beta_j \equiv \beta_j \alpha_i$$

Algorithm 2 checks if two tasks tid_1 and tid_2 commute at memory write actions α and β , respectively. The inputs to this algorithm are the sequences of all actions and arithmetic operations of the two tasks, the write actions α and β which indicated nondeterminism, and the address **addr** where nondeterminism occurred. At lines 8 - 14 it searches for all types of basic arithmetic and logic reduction operations with **addr** as operand from execution sequence of task tid_1 . It stores these operations in a set, at line 11. This set contains unique reduction operators at the end of search. At line 15 a similar procedure is repeated for actions and operations in task tid_2 where a separate set of operators is generated.

The algorithm checks if the rule one above is satisfied by the reduction operations on the shared memory **addr**, at lines 16 - 20, by comparing the two sets Op_α and Op_β . If they contain only commutative reduction operations which can arbitrarily operate in any order without affecting final result then we have proved that the tasks commute at memory write actions α and β . Otherwise the tasks tid_1 and tid_2 do not commute and the detected output nondeterminism is reported. Algorithm 2 can be executed between the lines 23 and 24 in Algorithm 1 to verify whether the detected output nondeterminism is a false warning or not due to commutative operations. Finally, the computation complexity of Algorithm 2 is $\Theta(|S_1||S_2|)$, where S_1 and S_2 are sequences

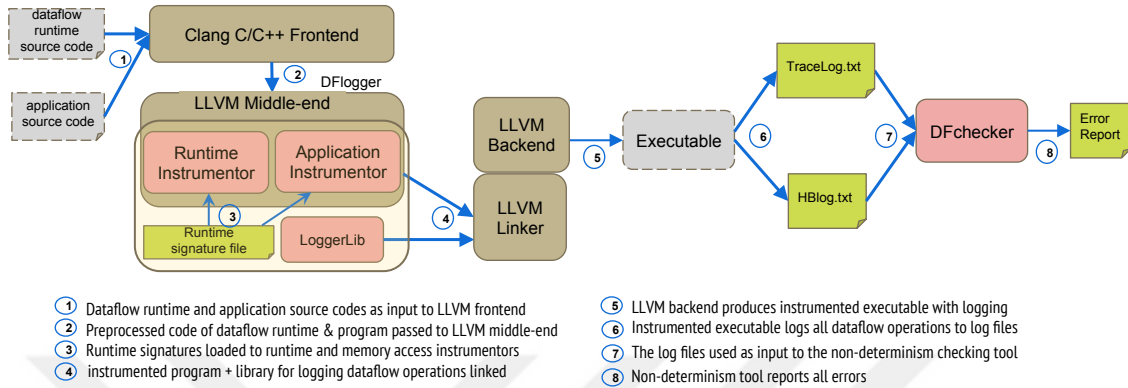
of all actions from two writer tasks involved in commutative operations.

Algorithm 2 Checking if two tasks commute on memory operations at an address `addr`.

```

1: function IFCOMMUTE(  $S_1, S_2, \alpha, \beta, \text{addr}$ )
2:   Input:  $S_1$  ▷ sequence of all actions from  $tid_1$ 
3:   Input:  $S_2$  ▷ sequence of all actions from  $tid_2$ 
4:   Input: addr ▷ memory location in question
5:   Input:  $\alpha, \beta$  ▷ nondeterministic actions by  $tid_1, tid_2$ 
6:   Set  $Op_\alpha$  ▷ reduction operators on addr by  $tid_1$ 
7:   Set  $Op_\beta$  ▷ reduction operators on addr by  $tid_2$ 
8:   for  $\alpha_i$  in  $S_1$  and  $\alpha_i \neq \alpha$  do
9:     if  $\alpha_i$  is a reductionOp then
10:      if addr  $\in$  operands( $\alpha_i$ ) then
11:         $Op_\alpha := Op_\alpha \cup \{\alpha_i\}$ 
12:      end if
13:    end if
14:  end for
15:  ▷ Repeat Line 8-14 for  $\beta$  and  $S_2$ 
16:  if  $Op_\alpha$  commutes with  $Op_\beta$  then
17:    return true
18:  else
19:    return false
20:  end if
21: end function

```

Figure 5.3: The architecture and action flow of *DFinspec* tool.

5.4 Implementation

In this section we present the *DFinspec* tool that we have developed for detecting output nondeterminism. We start with the components of the tool and their functions. Then we discuss how the tool detects output nondeterminism in dataflow programs. Lastly, we discuss how we address the false alarms due to commutative tasks and certain limitations of the tool.

The architecture of *DFinspec* is shown in Figure 5.3 and it mainly comprises three flow actions. First, it instruments the program and its dataflow runtime by injecting special functions which are launched during program execution. These functions record information about task actions discussed in Section 5.3.1 for nondeterminism detection. The instrumentation is done at compile time through the middle-end of the LLVM/Clang compiler. Second, the recorded information about actions performed by tasks is saved into log files. Lastly, our tool uses these log files to do output nondeterminism checking using Algorithm 1. The main modules of the tool are discussed in the following sections.

5.5 Runtime Instrumentor

This module accepts the function signatures (dataflow API), which initialize and terminate the dataflow runtime, from an input file. Then it instruments the runtime

to identify them to serve as initialization and termination points for the logging module. The signature file provides extension flexibility to support more than one dataflow runtimes. This module also instruments the runtime to identify *send* actions that output tokens to other tasks, which is important for establishing happens-before relations between senders and receivers of tokens.

5.6 *Application Instrumentor*

This module tracks the beginning and the ending of a task and adds logging callbacks at these points to log the starting and ending of task execution. The logged information helps to identify and categorize all relevant actions performed by a task. Moreover, it tracks relevant function calls to receive tokens in the task which are used to establish task dependency – hence the happens-before relation between tasks. To do this, it instruments all library calls related to incoming tokens in a task by inserting appropriate logging function callbacks for registering the tokens. This module also inserts a logging callback which logs the memory address and the value read from or written to the address for each load and store instruction, respectively.

5.7 *Logger*

Logging actions: The injected callbacks by the instrumentation record to a log file the relevant information of actions executing inside the tasks. One of the challenges with logging is to find an efficient and compact form to present the logged data. We devise the log format below for each logged line to save space and simplify the parsing of the recorded logs.

$$\langle tid \rangle \langle actionID \rangle \langle extraParameters \rangle$$

In this format;

- *tid* is a unique identification number of a task that performed the action. All actions performed by this task are uniquely identified by this *tid*.

- *actionID* is one-character code representing the type of an action.
- *extraParameters* represents extra information of the logged action; e.g. for memory actions, *extraParameters* translates to the tuple $\langle \mathit{address} \rangle \langle \mathit{value} \rangle \langle \mathit{lineNo} \rangle$. There are no extra parameters for the action $\mathit{end}(tid)$. Table 5.1 summarizes all the extended formats of *extraParameters* for different actions of a dataflow program.

Action	Action ID	Extra parameters
$\mathit{start}(tid)$	B	$\langle \mathit{task\ name} \rangle$
$\mathit{receive}(tid, token, buff)$	C	$\langle \mathit{sender\ task\ ID} \rangle$
$\mathit{send}(tid, token, buff)$	S	$\langle \mathit{buffer\ ID} \rangle$
$\mathit{read}(tid, addr, val)$	R	$\langle \mathit{address} \rangle \langle \mathit{lineNo} \rangle$
$\mathit{write}(tid, addr, val)$	W	$\langle \mathit{address} \rangle \langle \mathit{value} \rangle \langle \mathit{lineNo} \rangle$
$\mathit{end}(tid)$	E	-

Table 5.1: Summary of logging formats

At instrumentation phase, it is difficult to identify all actions that belong to tasks and exclude others which belong to the runtime or outside the task regions. Therefore, we develop a simple check during logging of an action to identify if the action has executed inside a task. Basically, when a task performs an action, a relevant callback is executed to check if the action is $\mathit{start}(tid)$ and a flag is set in the logger to mark that a task with ID tid has started execution. Likewise if the action is $\mathit{end}(tid)$, the logger is notified that the task has completed its execution and further logging for that task stops. The rest of the actions are logged normally if the task has started execution and has not terminated.

The module also determines reader and writer tasks for each shared memory and logs relevant actions to the file. For each reader task of a given memory location, it logs to the file only the first read action. To identify a task as reader of a memory address and record its first read, the logger module tracks all the task accesses to the address to verify that all are read accesses. If task writes at least once to that address

then it becomes a writer task. Similarly, it logs only the last write action of a writer task for an address because only the result of this action is visible to other tasks in ADF. To classify a task as a writer and eventually save its write to an address, the module tracks all task writes to that address and records the last. With this approach, only one memory access per task is saved to the log file for each address, greatly reducing the log size.

Logging happens-before relation between tasks: To improve efficiency and performance of output nondeterminism detection, we log the information about data dependency between tasks into a separate file, called *HBlog.txt* as shown in Figure 5.3. In a program if there is data flowing through a token from task tid_1 to task tid_2 then there is a happens-before relation between these tasks and therefore a pair of task IDs “ $tid_1\ tid_2$ ” is recorded in this file. This log file is generally very small in size compared to the log file which records the actions in a given execution.

In our implementation, each token is uniquely identified by its value and address of the token buffer it passes through from the sender to the receiver. We keep a simple hash map of these token values and the memory addresses as keys and the unique identifier of the task which sends the token. When a new task receives a token, a lookup is performed in the hash map using the token value and the buffer address to retrieve the sender’s identifier. This establishes the relation between the sender and the receiver of the token.

5.8 *DFchecker: Output nondeterminism detection*

Output nondeterminism detection is done once the information of all actions in an execution trace are logged into a file. The *DFchecker* module of our tool performs the detection by analyzing the log file line-by-line. Before parsing the action log file, a simple directed acyclic graph (DAGgraph) is constructed from the happens-before log file. The vertices of this graph are the identifiers of tasks and the edges are the dataflow dependencies between tasks. The tail of an edge designates the source of a

token and the arrow points to the receiver.

To maintain the happens-before relation between tasks as a transitive closure, we use a data structure similar to *Serial-bag* in the SP-algorithm [22]. In our implementation an *Sbag* is a simple set of unique task identifiers which represent a happens-before relation. An *Sbag* of a given task contains the integer IDs of tasks which happened before this task. Accessing this structure has an overall $O(1)$ complexity because it uses *unordered map* implementation from the C++11 programming language [5].

To minimize the memory usage and computation, a task inherits (does not copy) the *Sbag* from the preceding (parent) task which has token output in a buffer to this task. Therefore no new *Sbag* is created when a task starts execution unless it is an initial task or its parent task has multiple token output buffers to different tasks. This means if a terminating task outputs token in buffers to different succeeding tasks, only one of the new child tasks inherits its *Sbag* and the remaining tasks clone it. To do this, an *Sbag* of a task keeps the count of succeeding tasks; an information obtained from the happens-before log file. This count is decremented by one when a succeeding task clones the *Sbag*. A task inherits the *Sbag* when the count reduces to one.

Algorithm 3 presents an efficient algorithm for constructing *happens-before* transitive closures in linear time. At lines 8 and 9, it uses the DAGgraph to check if the new task has tasks which sent tokens to it. Then at lines 10 - 16 it searches for an *Sbag* to inherit from the *Sbags* of the preceding sender tasks. In the case that there are no preceding tasks that send tokens to the new task, a new *Sbag* is created at lines 19 - 23. Finally, the task IDs in the *Sbags* of these preceding tasks are merged to form a transitive closure of happens-before between this task and the preceding tasks (lines 24 - 29). At line 28 the child count *chCount* is decremented by one for each *Sbag* of the preceding tasks so that the future succeeding tasks that have happens-before relation with these tasks can inherit the *Sbag*.

The *DFchecker* module implements the output nondeterminism detection Algorithm 1 which is executed for every action in the action *log* file(i.e; trace). If the log line of an action represents the start of a new task, then Algorithm 3 is launched

to create a happens-before *Sbag* of this task. The rest of actions in the trace are processed as discussed in Section 5.3.4. Finally, output nondeterminism detected is reported only if the corresponding tasks do not commute (see Section 5.3.5).

Algorithm 3 Constructing happens-before transitive closure.

```

1: function GENERATETASKHB( tid )
2:   Input: tid                                ▷ tid is a task identifier
3:   Var sBag                                    ▷ SBag of a task tid
4:   Var pBag                                    ▷ SBag of a parent task to tid
5:   Graph DAGgraph                             ▷ graph of nodes and tokens
6:   Set serial_bags                             ▷ holds the SB bags of tasks
7:   Set parentTasks                            ▷ sets of parent tasks IDs to tid
8:   parentTasks ← DAGgraph.getParents( tid )
9:   if parentTasks is not Empty then
10:    for p_tid in parentTasks do
11:      pBag ← serial_bags.getSBag( p_tid )
12:      if pBag.chCount ≡ 1 then
13:        pBag.append( p_tid )
14:        pBag.Count ← DAGgraph.chlds( tid )
15:      end if
16:    end for
17:    serial_bags.setOwner( tid, sBag )
18:  end if
19:  if no serial_bags.getSBag( tid ) then
20:    sBag ← createNewSbag( tid )
21:    sBag.chCount ← |DAGgraph.Chlds( tid )|
22:    serial_bags.addSBag( tid, sBag )
23:  end if
24:  for p_tid in parentTasks do

```

```

25:     pBag ← serial_bags.getSBag( p_tid )
26:     sBag.append( pBag )
27:     sBag.appendParent( p_tid )
28:     decrement pBag.chCount
29: end for
30: end function

```

5.9 Detecting commuting writer tasks

The output nondeterminism detection algorithm cannot identify commuting actions between concurrent tasks (see Section 5.3.5) as it relies on different values written to a shared memory by these tasks. Therefore, we extend *DFinspec* tool to detect the commuting writer tasks actions.

Once our algorithm signals output nondeterminism at a memory address `addr` between two write actions from two tasks that do not have happens-before relation, *DFinspec* accesses the history to `addr` from each task to identify all actions and basic arithmetic operations manipulating `addr`. It determines if these two groups of actions and arithmetic operations would be intertwined without affecting the final result of `addr`. To do this, the tool keeps the sequences of program instructions of each task body in the form of *LLVM's intermediate representation (IR)* [39]. It uses them in Algorithm 2 to determine if the write actions from these tasks commute on `addr` and thus the tool does not report output nondeterminism.

To illustrate how *DFinspec* checks for commutativity, consider the concurrent tasks *DepositTask* and *WithdrawTask* in the motivating example from section 5.2. Listing 5.2 shows the relevant IR of these tasks. At line 1, *DepositTask* reads from memory the current value of `balance` and stores it into register `%a`. At line 2 it adds 200 to this value and stores the result into another register `%b`. Finally at line 3 it stores the content of register `%b` back to `balance`. On the other hand, *WithdrawTask* loads the current content of `balance` into register `%x`. Then at line 5 it subtracts the content

of `%x` with 500 and stores the result in register `%y`. Finally it stores back the final value to `balance` at line 6.

From the IR, we can deduce that *DepositTask* performs an addition operation; `balance := balance + 200` and *WithdrawTask* performs a subtraction operation; `balance := balance - 500`. These tasks can execute in any order without affecting the final value of `balance` as they are commutative, hence *DFinspec* does not report output nondeterminism on them.

Listing 5.2: Instructions of *WithdrawTask* and *DepositTask* from the motivating example

DepositTask

```
1:  %a = load float , float* @balance , align 4
2:  %b = fadd float %a , 2.000000e+02
3:  store float %b , float* @balance , align 4
```

WithdrawTask

```
4:  %x = load float , float* @balance , align 4
5:  %y = fsub float %x , 5.000000e+02
6:  store float %y , float* @balance , align 4
```

This approach for eliminating false alarms has limitations because the backtracking analysis is static. Moreover, we examine the instructions at basic block level without considering branches within a task body which may also modify shared memory location under test. It is an open research to extend our implementation to follow nested basic blocks and function calls.

Chapter 6

EMBEDSANITIZER: RACE DETECTION FOR PTHREAD IN 32-BIT EMBEDDED ARM

Techniques for detecting data races at runtime in pthread applications are still in demand. The main problem with available solutions are limited in part by at least one of (a) runtime overhead, (b) low race detection precision, (c) dependency on target hardware architecture. First, solutions with high overhead are impractical in large applications. Second, tools with high false alarms degrade programmer productivity by manually filtering real races from false warnings. Last, runtime race detection involves instrumentation which relies on the target hardware instruction architecture. Many practical solutions are platform-specific and thus there is always need for solutions for each available hardware architecture. In this regard we propose a method for detecting data races in Pthread applications running on 32-bit embedded systems. We first introduce the problem, motivate it, and propose our solution.

6.1 Introduction

Embedded systems are everywhere: from TVs to robots to smartphones to Internet of Things. Moreover, the computing capability of these systems has tremendously increased in recent years due to multicore support. This has enabled the implementation of complex multithreaded parallel applications. Unfortunately, these applications are prone to concurrency errors such as data races. These bugs are hard to detect in nature and the availability of relevant tools for embedded systems is still limited.

Most of the software development environment for Embedded systems rely on hardware emulations, which tend to be slow. Race detection of embedded system software through emulation can add even more overhead. Nevertheless, to run soft-

ware for race detection on a real hardware not only provides precise race reports but also is faster and hence more productive. On the other hand, many practical race detection tools for C++ applications have not focused on embedded system architectures. Therefore, the alternative is to compile 32-bit embedded C++ applications for other architectures and do race detection there. Unfortunately, some parts of the software that use special features of the target hardware may not be checked due to unavailability of such features in the alternative platforms. Further, it is more appealing to use full features of the software on the target devices for race detection.

We propose a tool named *EmbedSanitizer* [1, 45] for detecting data races for multithreaded 32-bit Embedded ARM software at runtime by running the instrumented application in the target platform. There are two advantages of this approach: (a) parts of software which use unique features, like sensors and actuators, can be analyzed. (b) enhanced developer productivity and throughput attained due to increased performance of race detection compared to hardware emulator. Our tool modifies *ThreadSanitizer* [61] to support race detection for the embedded ARMv7 architecture. Moreover, LLVM/Clang is modified to support *EmbedSanitizer* so it launches in a similar manner to *ThreadSanitizer*. For simplicity, *EmbedSanitizer* has an automated script which downloads necessary components and builds them together with LLVM/Clang as a cross-compiler. Multithreaded C/C++ programs through this compiler are instrumented and finally run on the target 32-bit ARM hardware for race detection.

Our key contributions on detecting races for embedded systems are:

- (a) We present a tool for detecting data races in C/C++ multithreaded programs for 32-bit embedded ARM. The tool is easily accessed through Clang compiler chain like *ThreadSanitizer*.
- (b) We motivate the idea of supporting race detection in native embedded systems hardware and show usability of race detection on such architectures.
- (c) We evaluate our tool and show its applicability by running PARSEC benchmark applications on a TV with ARMv7 CPU.

6.2 Motivation

We aim to promote utilization of existing race detection tools by adapting them to different hardware architectures. To show benefits of this approach, consider a theoretical multithreaded example in Figure 6.1. It models a TV software component which has two concurrent threads. `ReceiveThread` reads TV signals from an antenna and puts data in a shared queue `queue`. Then `DisplayThread` removes the data from the queue and displays on the TV screen. For the sake of motivation, the implementation of the queue is abstracted away but uses no synchronization to protect concurrent accesses. Since `ReceiveThread` and `DisplayThread` do not use a common a lock (LK1 & LK2 are used) to protect accesses to `queue`, there is a data race at lines 5(a) and 4(b).

0	VideoSignalQueue queue;	0
<pre> 1 void ReceiveThread() { 2 while(true) { 3 Signal s = receive(); // from antenna 4 acquire_lock(LK1) 5 queue.put(s); 6 release_lock(LK1) 7 } 8 } </pre> <p style="text-align: right;">(a)</p>	<pre> 1 void DisplayThread() { 2 while(true) { 3 acquire_lock(LK2) 4 Signal s = queue.get(); 5 release_lock(LK2) 6 display(s); // to screen 7 } 8 } </pre> <p style="text-align: right;">(b)</p>	

Figure 6.1: A motivating example with two threads concurrently accessing a shared queue. A thread in (a) reads video signals from TV antenna and puts them into the queue, (b) reads from the queue and display to a screen.

Assume that the developer chooses a method other than the proposed one for race detection. She has two challenges: (1) Modeling the *receipt* as well as the *display* of the video signal data. (2) After that, she can do race detection on an alternative architecture, emulation or virtualization rather than the target architecture. Further overhead is incurred if emulation or virtualization is used. Conversely, the target hardware already has these features and may be faster and thus increasing developer productivity. Moreover, the advantage of instrumenting program and later detecting

races on a target hardware is that the developer uses real features for receiving and displaying the signals. This aligns exactly well with our proposed solution.

6.3 Method

EmbedSanitizer improves on *ThreadSanitizer*. It can also be launched through Clang’s compiler flag `-fsanitize=thread`. To achieve this, we modified the LLVM/Clang compiler argument parser to support instrumentation of 32-bit ARM programs when the relevant flag is supplied at compile time. Next, *EmbedSanitizer* enhances parts of the *ThreadSanitizer* to instrument the target program. Furthermore, it replaces the 64-bit race detection runtime with a custom implementation of the efficient and precise FastTrack race detection algorithm, for 32-bit platforms. In this section, we discuss the important parts of *EmbedSanitizer* as well as its simplified installation process.

6.4 Architecture and Workflow

Workflow of the *ThreadSanitizer* and the changes done by *EmbedSanitizer* are described in Figure 6.2. Figure 6.2(a) shows default and unmodified relevant components of *ThreadSanitizer* in LLVM/Clang. In Figure 6.2(b) these parts are modified to enable instrumentation and detection of races for 32-bit ARM applications.

At ① in Figure 6.2(a), the Clang front-end reads the compiler arguments and parses them. If the target architecture is 64-bit, Clang passes the program under compilation through *ThreadSanitizer* compiler pass for instrumentation ②. The pass then identifies all shared memory operations in the program and injects relevant race detection callbacks which are implemented in a race detection runtime library called *tsan*. Furthermore, the instrumented application and the runtime are linked together by the linker ③ to produce an instrumented executable ④. This executable once runs on a target 64-bit platform, it reports race warning in the program. We modify components in the workflow as discussed next.

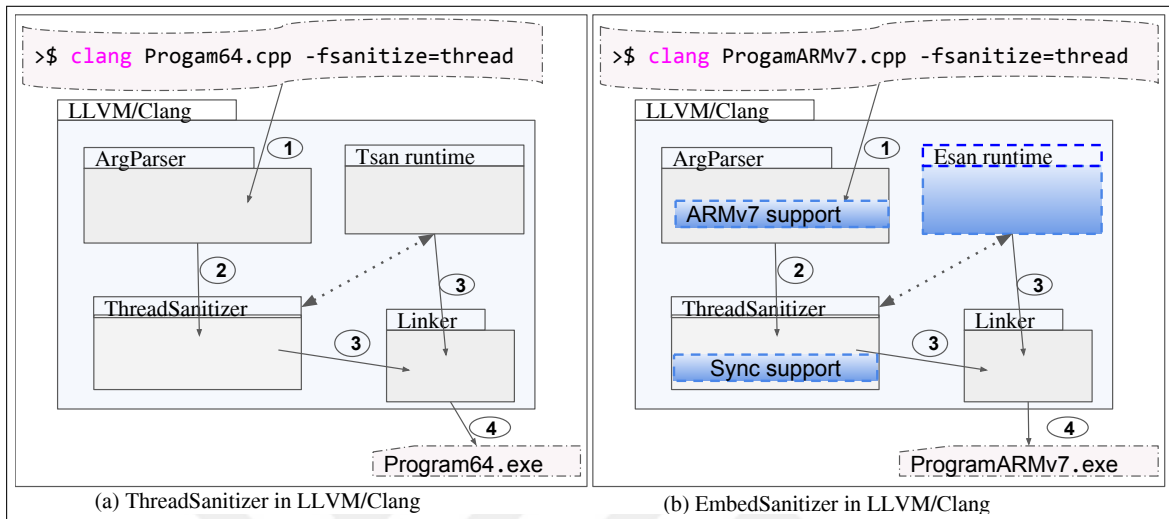


Figure 6.2: High level abstraction of *ThreadSanitizer* and *EmbedSanitizer* in LLVM/Clang. In (a) *ThreadSanitizer*: essential LLVM modules for race detection. In (b) *EmbedSanitizer*: same modules modified to instrument and detect races for 32-bit ARM

(a) Enabling Instrumentation of 32-bit ARM Code in LLVM/Clang: We modify the argument parser of LLVM/Clang to support instrumentation once *EmbedSanitizer* is in place, Figure 6.2(b). Therefore, if `-fsanitize=thread` flag is passed while compiling a program for 32-bit ARM code, the instrumentation takes place. To do this we identified the locations where Clang processes the flag and checks the hardware before skipping the launching of *ThreadSanitizer* instrumentation module because of unsupported architecture.

(b) Modifying the *ThreadSanitizer* Instrumentation Pass: Despite its instrumentation pass, *ThreadSanitizer* has become complex, partly due to its integration into the LLVM’s compiler runtime. We extended the available instrumentation pass to identify and instrument synchronization events and inject relevant callbacks and kept instrumentation of memory accesses as it is.

(c) Implementation of Race Detection Runtime: The default race detection runtime in *ThreadSanitizer* uses memory shadow structures which rely on 64-bit ar-

chitectural support. Due to the complicated structure of ThreadSanitizer, it was not possible to adopt its runtime for 32-bit ARM platform. Therefore, we implemented a race detection runtime by applying the FastTrack race detection algorithm. The library is then compiled for 32-bit ARM and is linked to the final executable of the embedded program at compile time.

6.4.1 Installation

Figure 6.3 shows the building process of LLVM compiler infrastructure with *EmbedSanitizer* support. To simplify this process we developed an automated script with five steps. In the first step, it downloads the LLVM source code from the remote repository. Then it replaces files of the LLVM/Clang compiler argument (flags) parser with our modified code to enable *ThreadSanitizer* support for ARMv7. Third, the LLVM code is compiled using GNU tools to produce a cross-compiler which targets 32-bit ARM and supports our tool, *EmbedSanitizer*. Fourth, the race detection runtime which we implemented is compiled separately and integrated into the built cross-compiler binary. Finally, the built cross-compiler is installed which can eventually be used to compile 32-bit ARM applications with race detection support. This whole process is applied once.

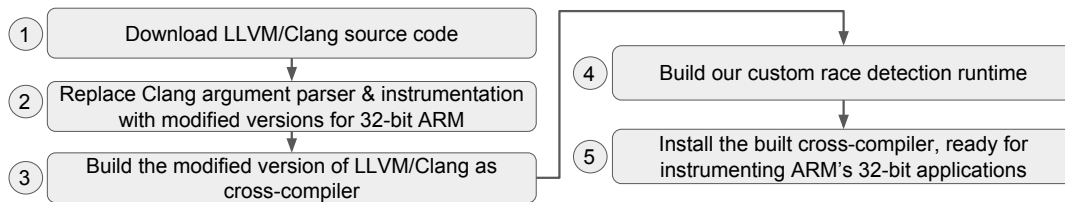


Figure 6.3: Showing the automated process for building *ThreadSanitizer* for the first time.

Chapter 7

EVALUATION

In this chapter we present and discuss evaluation results from detecting determinacy races in OpenMP tasks. Moreover, we show a complete set of experimental evaluation for detecting output nondeterminism in ADF applications. Finally, we present experimental results for detecting data races in POSIX Threads applications in 32-bit embedded systems.

7.1 *Evaluating Runtime Determinacy Race Detection for OpenMP Tasks*

We evaluate our tool on nine micro-benchmarks on three categories: (a) the number and nature of determinacy races reported as well as no determinacy races reported in correct programs, (b) detection comparison with Archer [9], (c) the runtime overhead with respect to input size. We first provide a brief summary of the applications before discussing evaluation results. The first five applications are custom implementations with races, accessible through TaskSanitizer¹.

- ***RacyBackgroundExample:*** implements the example in Figure 4.1. There are two tasks each containing a critical section associated with the same lock. One task sets 1 to shared variable i while the other sets 2 without enforced dependency thus exhibiting a determinacy race as these operations do not commute even though they are in critical sections.
- ***RacyBanking:*** We mimic the motivating banking example in [44]. An initial task sets the account balance to 1000. Then three concurrent tasks access the

¹<https://github.com/hassansalehe/TaskSanitizer/tree/master/src/benchmarks>

account balance without specified dependency among them, thus causing three determinacy races and the updates on the account do not commute.

- ***RacyFibonacci***: This program computes Fibonacci of a given number n using *memoization* technique of caching intermediate results in a shared integer array. A task for n creates two concurrent child tasks to compute Fibonacci of $n-1$ and $n-2$, respectively, and each stores its result in the *memoization* array. The task then sums the results from the array after a synchronization barrier with the child tasks. There are determinacy races in this example on five program locations between two concurrent sibling tasks as they access the memoization array without inferred dependency between them.
- ***RacyMapReduce***: constructs histogram of words from a text file. It splits the input text into four chunks. Then each chunk is processed by *map* tasks. The partial results are merged into a final histogram by *reduce* tasks which are concurrent to each other, exhibiting four determinacy races while inserting new words into the final histogram and updating word counts.
- ***RacyPointerChasing***: traverses a singly-linked list and creates an explicit task for each node to insert a number to the node for the purpose of forming an arithmetic sequence in the linked-list. In this program, two random nodes in the list mistakenly contain common memory address for storing their terms which breaks the arithmetic sequence. As a result, their corresponding tasks concurrently write values to the memory, causing a determinacy race.
- ***sectionslock1-orig-no***: As part of the DataRaceBench micro-benchmark suite [42], this program creates two parallel sections, which have critical sections in which one section increases a shared variable by 1 and other section increases it by 2. There are no determinacy races because these operations in critical sections commute and our tool does not report a bug.

- ***taskdep1-orig-no***: As part of DataRaceBench, the program creates two explicit tasks with the first task setting 1 to a shared variable and the succeeding sibling task setting 2. These tasks have specified dependency between them and thus no determinacy races.
- ***taskdep3-orig-no***: As part of DataRaceBench, this program creates two explicit tasks. The first task has dependency with each of the other sibling tasks which are concurrent to each other. Since the concurrent tasks only read from a shared variable, there is no determinacy race.
- ***taskdependmissing-orig-yes***: As part of DataRaceBench, this program creates two concurrent explicit tasks which have no dependency in between. They modify a shared variable and thus constitute a determinacy race.

Table 7.1: Comparing detection results of TaskSanitizer against Archer

Application	Input size	Number of tasks	Known races	TaskSanitizer Races found	Archer Races found
RacyBackgroundExample	-	6	1	1	0
RacyBanking	-	11	3	3	2
RacyFibonacci	5	137	8	8	11
RacyMapReduce	-	17	4	4	1
RacyPointerChasing	14	34	1	1	0
sectionslock1-orig-no	-	2	0	0	0
taskdep1-orig-no	-	6	0	0	0
taskdep3-orig-no	-	8	0	0	0
taskdependmissing-orig-yes	-	6	1	1	0 or 1

7.1.1 Precision Evaluation of TaskSanitizer

Table 7.1 lists the reported bugs by our tool, TaskSanitizer and number of determinacy races known in advance for micro-benchmarks. In ***RacyBackgroundExample*** two concurrent tasks execute two critical sections which each sets different value to a shared memory location. This exhibits a determinacy race since the tasks do not have HB relation and their memory operations do not *commute* in critical sections. Our tool does not check for commutativity in remaining buggy programs as their

operations happen outside critical sections. Even though tasks with critical sections in *sectionslock1-orig-no* do have dependency, there is no determinacy race reported because increment operation in these sections *commute*. Finally, our tool does not report false positives in the remaining programs.

7.1.2 Comparing Detection with Archer

We compare our determinacy race detection results with data race detection results of Archer [9], which is an efficient tool based on ThreadSanitizer for detecting data races. Data race detection in Archer differs from determinacy race detection in our approach on two essences: (i) It relies on thread-level concurrency and thus it fails to detect races in concurrent tasks scheduled to execute by the same thread. (ii) It aims at detecting violations of locking critical sections which have shared memory accesses whereas our method focuses on different ordering of events leading to determinacy races.

As shown on Table 7.1, Archer failed to detect races in *RacyBackgroundExample* and *RacyPointerChasing* despite multiple runs. Archer fails to detect the race in *RacyBackgroundExample* because memory operations are protected by a common lock. However, our tool detects determinacy races because the locks do not enforce deterministic ordering and thus the program can produce different results at different runs.

Archer does not detect a race in *taskdependmissing-orig-yes* and other buggy programs when concurrent tasks in the program are scheduled to execute with one thread. Therefore, Archer detects the race only if two tasks are executed by different threads whereas our tool detects the determinacy race in the program at all runs. This is because Archer depends on program threads to infer concurrency whereas our approach abstracts away threads and detects determinacy races at task level. Moreover, the number of races it reported on the remaining buggy programs varied from zero to the expected depending on scheduling of concurrent tasks to different threads. However it detected two races in *RacyBanking* and did not produce false

alarms in correct programs.

7.1.3 Overhead Evaluation

Even though the focus of this work is the method for detecting determinacy races, we also measured the slowdown of determinacy race detection in the micro-benchmark applications which accept varying input sizes, namely *RacyFibonacci* and *RacyPointerChasing* as shown in Figure 7.4. By increasing input size, we calculated execution times of the application without determinacy race detection as well as with detection. We calculated slowdown by dividing detection time by execution time without detection. The determinacy race detection slowdown from this experimental setting ranges from 1.0 to 1.26X.

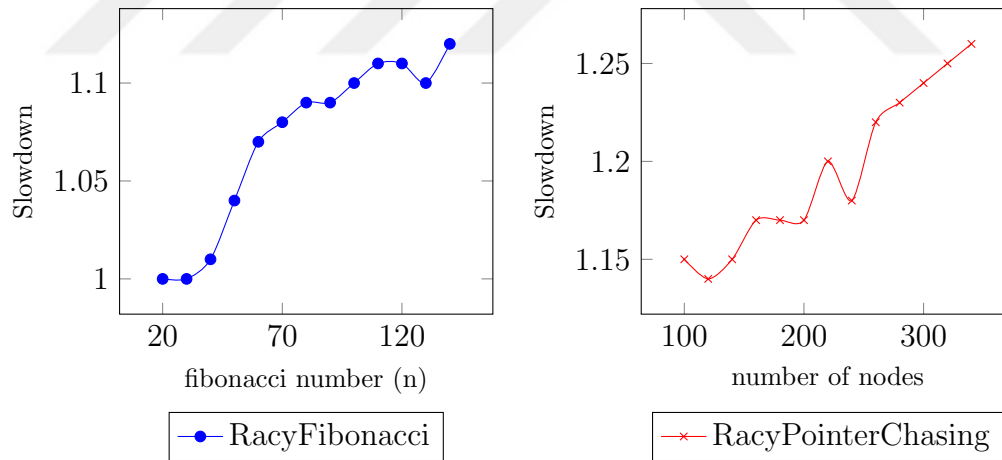


Figure 7.4: Slowdown of determinacy race detection in programs as input size increases

7.2 Evaluation on Detecting Output Nondeterminism for ADF

In this section, we provide experimental results for ten of the Berkeley dwarfs [8], which are implemented in ADF as a part of the DaSH benchmark suite [26]. In the evaluation we also include the motivating example that we presented in Section 5.2. We conduct two types of experiments. First, we apply our tool to the motivating example and the benchmark applications using reasonable input sizes to detect any

output nondeterminism errors in them. Then we introduce synthetic bugs on to four of the error-free applications to check if the tool detects them. We evaluate the following applications from the benchmark suite:

(a) *Branch and bound* implements an approximation algorithm to solve the Traveling Salesman Problem (TSP), where it uses an assignment problem to generate lower bounds. In this application some tasks are responsible for generation of subproblems and others for solving them and determining lower bounds for the next iterations.

(b) *Combinatorial logic* performs bit-level operations on large data set. In the simple implementation in the benchmark suite, some tasks read and divide the input into small chunks and the rest count the number of 1s from the input data.

(c) *Dense linear algebra* in this setting performs matrix multiplication of real dense matrices. Individual worker tasks concurrently compute values for a number of result matrix cells by using rows and columns from two input dense matrices, respectively.

(d) *Finite state machine* implements pattern matching algorithm using pattern characters as finite states. It takes a large chunk of text as input and finds all occurrences of the matching pattern. The *initial* tasks partition the input text into smaller chunks whereas *find* tasks take them and search for the pattern. If a match is found, the position is stored into a doubly linked list shared among the *find* tasks.

(e) *Graph models* constructs a logical graph from an observation binary input. From this input the nodes of the graph are variables and edges are modelled as conditional probabilities.

(f) *Map Reduce* constructs a word histogram from a text input file. The initial tasks divide the input into chunks and these chunks are later individually read and put in lists by the *map* tasks where are sorted and partial histograms per chunk are calculated by the *reduce* tasks. The *sum* tasks aggregate the partial results to the final histogram.

(g) *Sparse linear algebra* performs LU decomposition of a sparse real matrix stored in compressed-storage format. Since we detected a real output nondeterminism bug in this application, more information is provided in Section 7.2.1.

(h) *Spectral methods* implements 3D fast Fourier transform (FFT) algorithm. *fftX* task computes FFT along x-plan, whereas *transposeXY* and *transposeZX* perform transpose operations. Moreover, *barrier* tasks synchronize the transpose tasks.

(i) *Structured grid* uses iterated Jacobi approach to compute a square mesh. In this application, the computation is partitioned into a number of tasks which perform a number of iterations before sending tokens to new computation tasks.

(j) *Unstructured grid* is similar to structured grid but it works on irregular set of triangular planes.

7.2.1 Detecting Real Output Nondeterminism Bugs

Table 7.2 shows the results of the *DFinspec* tool applied to the applications as they are available in the benchmark suite. The table shows the input size and task count of each application. Our tool identified two output nondeterminism bugs in the motivating bank example discussed on Section 5.2. In addition from the DaSH benchmark suite, *DFinspec* detects one real output nondeterminism bug in the *Sparse linear algebra* application due to a missing dependency that the authors of this application might have forgotten to add. *DFinspec* also detects three bugs in *Finite state machine* caused by concurrent access of a doubly linked list where the shared pointers are modified. In *Unstructured grid*, the bug is due to concurrent access to a shared memory location which acts as temporary buffer for local operations. Our tool did not report any output nondeterminism bugs from the remaining applications which have correct dependencies that we verified by manual inspection and output analysis.

Case study: SparseLU decomposition Our tool detected output nondeterminism in *sparse linear algebra* application. *Sparse linear algebra* performs LU decomposition on an $n \times n$ dimensional sparse matrix A whose elements are square blocks of size $m \times m$. Generally n is larger than m . Given matrix A , this application performs the matrix operations in an iterative manner along the diagonal of the matrix. Each iteration i begins with an *lu0* operation at the diagonal entry block $A[i, i]$. Then

Application name	Input size	Task count	Bugs found
motivating_bank_example	-	4	2
branch_bound	500 TSP nodes	1362	0
combinatorial_logic	131072 bits	69	0
dense_linear_algebra	1000x1000 matrix	1000	0
finite_state_machine	104M characters	203	3
graph_models	2000 states	1198	0
map_reduce	146729 words	128	0
sparse_algebra	6x6 matrix	19	1
spectral_methods	256x256 matrix	320	0
structured_grid	64x64 matrix	2152	0
unstructured_grid	853 grid vertices	332	1

Table 7.2: Experimental results from 10 applications and the motivating bank example.

forward Gaussian elimination (*fwd*) is performed on each of the matrix blocks in the corresponding row i starting at block entry $A[i, i+1]$ to $A[i, n]$. Moreover *bdiv* operations are performed on all blocks from $A[i+1, i]$ to $A[n, i]$ in the corresponding column i . To complete the iteration, *bmod* operates on all the inner block elements of the matrix from element $A[i+1, i+1]$ to $A[n, n]$ each using the block elements from the corresponding row and column that were updated by *fwd* and *bdiv*, respectively.

To summarize, *lu0* operation on a diagonal entry triggers *fwd* operations on all elements in the corresponding row. Moreover, it triggers *bdiv* operations on the matrix blocks in the corresponding column. *bdiv* and *fwd* operations then trigger *bmod* operations in the inner blocks which eventually trigger *lu0*, *fwd*, and *bdiv* operations in the next iteration. This is partly captured by a dependency graph in Figure 7.5.

Incorrect assignment of dependency tokens between tasks executing *lu0*, *bmod*, *bdiv* and *fwd* operations would lead to output nondeterminism into the application. Figure 7.5 shows a missing dependency between *bmod(2,2)_i1* task which executes *bmod* operation at block $A[2,2]$ at iteration 1 and *lu0(2,2)* task which executes *lu0* operation at $A[2,2]$ during iteration 3. This means there is no execution order enforced between these tasks: either task can execute before the other and the final computation in the matrix can be different. To show that this is the case and affirm

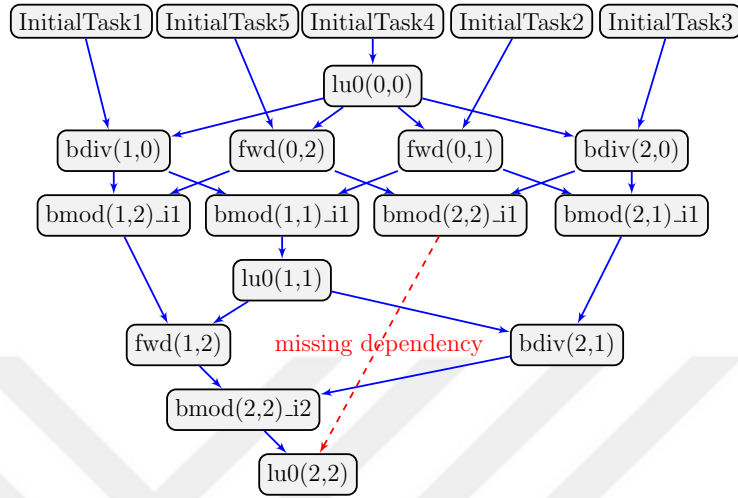


Figure 7.5: Task dependency graph of *sparse linear algebra* showing a missing dependency (shown as a dash line) between the *bmod* and *lu0* operations. This causes an output nondeterminism bug captured by our tool and manually verified.

that this is a really determinism violation we enforced different orders of execution of these tasks. Different orderings of tasks produced different final matrices.

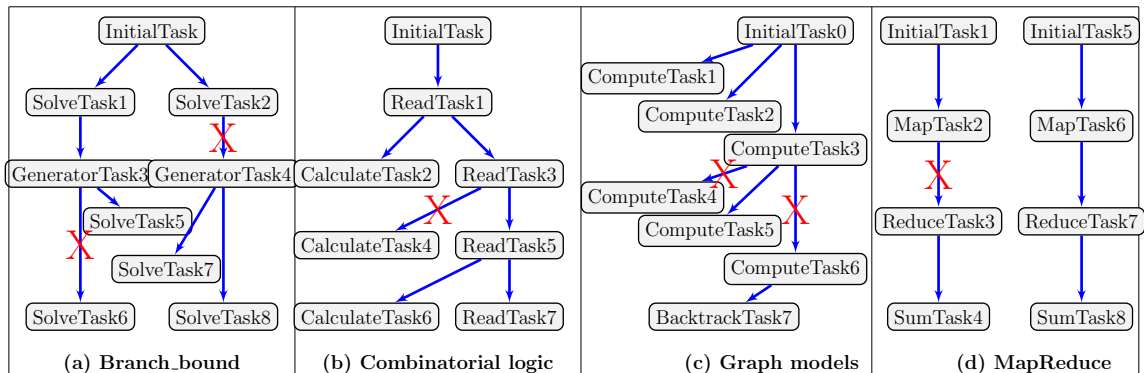


Figure 7.6: Task graphs of applications showing removed data flow dependencies to introduce output nondeterminism bugs

7.2.2 Detecting Synthesized Output Nondeterminism Bugs

To access the capability of our tool in detecting output nondeterminism bugs, we injected synthetic bugs into the benchmark applications which originally do not have

output nondeterminism bugs. Basically we broke some task dependencies for the tasks that modify shared memory addresses. Our purpose was to check if our tool was able to detect those bugs and report them.

We injected the output nondeterminism bugs by removing dependency between tasks without altering their execution flow. Figure 7.6 shows dependency edges that were removed. To decide where to remove dependencies we first ran the applications and generated the output task graphs (see Figure 7.6). From these graphs we selected a few dependencies and removed them so that their corresponding tasks would execute in any order thus potentially affecting the final program output.

Table 7.3 lists the benchmarks with the number of synthetic bugs injected and found. Indeed our tool was able to detect all of the synthesized bugs with the exception of the bug added in *combinatorial logic*.

In the case of *combinatorial logic*, we removed data dependencies between the tasks *ReadTask3* and *CalculateTask4*. *ReadTask3* reads a portion of input data set from a file and creates a block of data for *CalculateTask4* which then reads it and counts the number of 1s. Since *ReadTask3* and *CalculateTask4* have a *producer-consumer* relation, removing data dependency edge between them introduces a serious bug which affects the final output. Unfortunately our tool was not able to detect this bug because our definition of output nondeterminism bases on write actions not write-read actions between participating tasks. This is not the case for *ReaderTask3* and *CalculateTask4* because *CalculateTask4* only reads the data from memory locations that were modified by *ReadTask3*.

Application name	Input size	Task count	Bugs introduced	Bugs detected
branch_bound	12 TSP nodes	9	2	2
combinatorial_logic	9824 bits	8	1	1
graph_models	36 states	8	2	2
map_reduce	1478 words	8	1	1

Table 7.3: Showing number of synthetic bugs added into dataflow applications and a number of those detected by *DFinspec*

7.2.3 False Output Nondeterminism Due to Commutativity

In sections 5.3.5 and 5.9 we discussed that our output nondeterminism detection algorithm (Algorithm 1) generates false alarms mostly because it fails to identify commuting writer tasks actions. These actions are concurrent writes to a share memory from tasks which do not have *happens-before* relation but they *commute* (i.e; their execution order does not change the final output) on the set of these actions. Table 7.4 compares the real bugs with the number of false alarms reported by our tool before applying the the algorithm described in Section 5.3.5.

The false alarm in the motivating example is due to commutative operation by *DepositTask* and *WithdrawTask* on a variable `balance`. Since *DepositTask* adds 200 to and *WithdrawTask* reduces 500 from `balance`, the order of these operations does not matter and leaves `balance` with one final value. False alarms in *combinatorial logic* are due to the tasks which count number of 1s from chunks of input data and add the result to a global counter. These tasks do commute because the order of addition into the counter does not matter. Similarly, the sum tasks in *map reduce* commute because they perform reduction operations by aggregating histogram of individual input words. In *graph models* all *ComputeTask* tasks increment a shared variable called `states_processed` to keep the number of input chunks already processed. Since these are aggregate operations, the compute tasks indeed commute and the reported warnings due to these actions are false alarms.

In *branch and bound*, there were false alarms from a few *solve* tasks due to accessing two global variables which store the suboptimal result and an upper bound. In each branch-and-bound iteration the solve tasks try to improve the suboptimal result by taking it to a better next solution state. To do so, a solve task first reads the result from the global memory and in case it produces a better result it writes it back to the global variable. Since each solve task tries to access different next state of the current suboptimal solution, their order of actions on these global variables can be arbitrary and does not affect the final result of the execution, hence the tasks commute.

In *dense linear algebra* a number of tasks use sub-rows and sub-columns of the

operand matrices to partially compute and later reduce their results to a single cell of the result matrix. These commuting writer tasks have task dependency specified in the program and thus have *happens-before* relation.

Application name	Input size	Task count	Real bugs	False alarms due to commutative operations
motivating_bank_example	-	4	2	1
branch_bound	500 TSP nodes	1362	0	2 recurring on 6 tasks
combinatorial_logic	131072 bits	69	0	1 recurring on 33 tasks
dense_linear_algebra	1000x1000 matrix	1000	0	0
finite_state_machine	104M characters	803	3	1 recurring on 102 tasks
graph_models	2000 states	1198	0	1 recurring on 1196 tasks
map_reduce	146729 words	128	0	2 recurring on 32 tasks
sparse_algebra	6x6 matrix	19	1	0
spectral_methods	256x256 matrix	320	0	5
structured_grid	64x64 matrix	2152	0	0
unstructured_grid	853 grid vertices	332	1	2

Table 7.4: Showing both real bugs and false alarms. The false alarms were due to commutative tasks and were eliminated altogether by applying the algorithm discussed in section 5.3.5.

7.2.4 Overhead

To assess the feasibility of our *DFinspec* tool, we present execution times and overheads of instrumentation, action logging, and detection phases of the tool. Instrumentation at compile time uses LLVM compiler pass we developed to identify all task actions and add callbacks which collect relevant information at runtime for output nondeterminism detection. To evaluate overhead of the pass, two execution times for each application are measured: compilation time without instrumentation and with instrumentation. These results are reported on columns 2 and 3, respectively, and the resulting overhead on column 4 in Table 7.5. From the ten applications, the instrumentation overhead ranges between 17% and 82%. Instrumenting at compile-time is significantly faster than using runtime binary instrumentation frameworks like Intel PIN [43] and DynamoRIO [13] which heavily rely on Just-In-Time (JIT) compilation.

To measure the impact of logging the task actions, we first collect the execution time of each uninstrumented application (column 7). Then the execution time of the

instrumented application as it logs the task actions (column 8). The slowdown due to logging ranges between 5.5x and 86.3x. The logging slowdown is due to the fact that all the content is sequentially written to a single file and is protected by a global lock. We have improved the logging overhead by: (a) decreasing write frequency to the disk by accumulating large chunks of data before writing to a log file, (b) using *C++11*'s unordered maps which guarantee constant-time complexity [5] for storing and accessing logging metadata. (c) eliminating most part of shared metadata by using *C++11 thread_local* specifier for thread local storage. (d) logging only one access (read or write) per task as in the case of ADF tasks, the last write action by a task to an address is sufficient to detect nondeterminism.

The last column in Table 7.5 shows execution times of detecting output nondeterminism by parsing an action log file for each application. These results are relatively reasonable. However, we notice that the overhead increases with the log file size, which depends on the application and input size.

Application name	Compilation			Execution						Detection
	Base (sec)	Instrumentation (sec)	Slow-down	Input Size	Task Count	Base (sec)	With Logging (sec)	Slow-down	Log Size	Analysis Time (sec)
2*branch_bound	2*1.42	2*1.92	2*1.34x	500 nodes	1362	2.35	87.91	37.4x	1.5 GB	148.89
				2000 nodes	4468	7.66	409.13	53.4x	5.8 GB	638.71
2*combinatorial_logic	2*0.63	2*0.75	2*1.18x	131,072 bits	69	12.73	70.81	5.6x	74 KB	0.06
				2,097,152 bits	1029	204.21	1123.79	5.5x	1.2 MB	0.37
2*dense_linear_algebra	2*0.26	2*0.31	2*1.20x	1000x1000 matrix	1000	7.73	167.40	21.6x	707 MB	57.84
				2000x2000 matrix	8000	65.18	1353.79	20.8x	5.8 GB	490.55
2*finite_state_machine	2*0.46	2*0.54	2*1.17x	104M characters	803	1.41	85.37	60.5x	2.9 GB	177.89
				800M characters	803	7.33	173.87	86.3x	5.7 GB	378.43
2*graph_models	2*0.61	2*0.84	2*1.38x	2000 states	798	14.33	827.91	57.7x	18 GB	1475
				4000 states	798	63.41	4952.95	78.1x	119 MB	3.38
2*map_reduce	2*0.82	2*1.41	2*1.71x	146,729 words	128	0.62	4.43	7.1x	52 MB	16.42
				1,173,848 words	512	4.18	29.50	7.1x	373 MB	46.90
2*sparse_algebra	2*0.63	2*0.95	2*1.52x	1250x1250 matrix	1878	3.28	75.82	23.1x	312 MB	35.86
				2000x2000 matrix	6259	10.47	263.83	25.2x	1.2 GB	125.83
2*spectral_methods	2*0.47	2*0.86	2*1.82x	128x128 matrix	320	0.26	8.27	31.8x	190 MB	16.92
				256x256 matrix	320	1.45	78.56	54.2x	1.5 GB	134.95
2*structured_grid	2*0.47	2*0.63	2*1.34x	64x64 matrix	2152	0.55	21.11	38.4x	406 MB	35.14
				128x128 matrix	4383	2.75	133.39	48.5x	4.1 GB	294.40
2*unstructured_grid	2*0.65	2*0.92	2*1.41x	853 vertices	332	0.06	2.09	34.8x	46 MB	3.81
				21333 vertices	726	1.61	101.77	63.2x	2.7 GB	271.97

Table 7.5: Execution times and overheads of instrumentation, logging and output nondeterminism detection phases of *DFinspec*.

7.2.5 Limitations

Our main contribution is to present a technique and a tool for detecting output nondeterminism in dataflow applications on shared memory. However, the current version of the tool has a number of limitations. Despite the fact that our technique targets all dataflow programs, our tool has only been experimented with the applications developed for ADF, which ensures all tasks are atomic. We plan to extend it to support and experiment on other models. Nonetheless our demonstration with the ADF applications covers the wide spectrum of dataflow applications on shared memory in general because they are similar.

Another limitation is that the tool misses a sound and complete algorithm for detecting commuting writer tasks although our current approach eliminates most of the false alarms due to commuting writer tasks.

The number and type of bugs detected by our tool depend on the execution trace collected. In turn it depends on the type and nature of the application and the input size. Moreover, since our analysis works with a single execution trace for each application at a time, it can not detect all bugs reflected *only* on other possible execution traces of the program.

7.3 Results on Race Detection for POSIX Threads in 32-bit Embedded ARM

Our ultimate goal is to evaluate our approach by detecting data races in POSIX multithreaded smart TV software. However, we evaluated our primary implementation of the approach by detecting races on a set of Pthread PARSEC benchmark applications by running them in the 32-bit embedded ARM smart TV.

Our preliminary evaluation is based on two categories. First, we want to see how the precision of race detection in *EmbedSanitizer* deviates from that of *ThreadSanitizer* [61] since *EmbedSanitizer* extends it by using its instrumentation features, and implements a custom FastTrack [24] for detecting races. Second, we want to compare the overhead of *EmbedSanitizer* when running on a target embedded device against

when running on an emulator. The key motivation is to show that running race detection on a target device is better than on emulation.

For experimental setup, we built LLVM/Clang, with *EmbedSanitizer* tool, as a cross-compiler in a development machine running Ubuntu 16.04 LTS with Intel i7 (x86_64) CPU and 8 GB of RAM. As our benchmarks, we picked four(4) of the PARSEC benchmark [10] applications. We adapted them to Clang compiler and our embedded system architecture. A short summary about the applications we used for evaluation is given below.

- *Blackscholes*: parallelizes the calculation of pricing options of assets using the Black-Scholes differential equation.
- *Fluidanimate*: uses spatial partitioning to parallelize the simulation of fluid flows which are modeled by the Navier-Stokes equations using the renowned Smoothed particle hydrodynamics.
- *Streamcluster*: is a data-mining application which solves the k-means clustering problem.
- *Swaptions*: employs Heath-Jarrow-Morton framework with Monte Carlo simulation to compute the price of a set of Swaptions.

7.3.1 Precision Evaluation

Precision indicates how effective is a method on detecting real races. For our technique on detecting races for POSIX Threads applications in embedded systems, we compare the race reports detected by *EmbedSanitizer* against *ThreadSanitizer*. To do this we run the same benchmark applications with *ThreadSanitizer*, as well as with *EmbedSanitizer*. The instrumented program using *ThreadSanitizer* is run on an x86_64 machine, whereas the binary compiled through *EmbedSanitizer* is executed on ARM Cortex A17 TV. In this setting of four PARSEC benchmark applications, in an application where *ThreadSanitizer* reported races, *EmbedSanitizer* also reported

Table 7.6: Experimental results to compare race detection in ARMv7 using *EmbedSanitizer* vs in x86_64 with *ThreadSanitizer*.

Benchmark	Input size	Threads	Addresses	Reads	Writes	Locks	<i>ThreadSanitizer</i>	<i>EmbedSanitizer</i>
							Races	Races
Blackscholes	4K options	2+1	28686	5324630	409590	0	NO	NO
Fluidanimate	5K particles	2+1	149711	25832663	8457516	790	YES	YES
Streamcluster	512 points	2+1	11752	21710589	352605	2	YES	YES
Swaptions	400 simulations	2+1	243945	11000763	3377226	0	NO	NO

them as shown in Table 7.6. Therefore *EmbedSanitizer* did not sacrifice any race detection precision.

7.3.2 Performance Evaluation

Runtime performance is a determining factor on practicability of a race detection method. Therefore, to compare race detection slowdown of our approach, we ran non-instrumented and instrumented versions of the benchmarks on embedded TV with ARM-Cortex A17 CPUs of 4 logic cores and 933MB of RAM, and on Qemu-ARM emulator running on a workstation. The slowdown is calculated as a ratio of the execution time of the instrumented program with race detection on and the execution time of the program without race detection. The number of threads was 3 because using the full set of 4 logical cores was crashing the TV. Next, the input sizes were the same in each benchmark setting. Results in Figure 7.7 show that detecting races in an emulator incurs between 13x and 371x slowdown whereas the slowdown in the TV is between 12x and 214x. In overall, results in Figure 7.7 suggest that detecting races in a target hardware is faster than in an emulator.

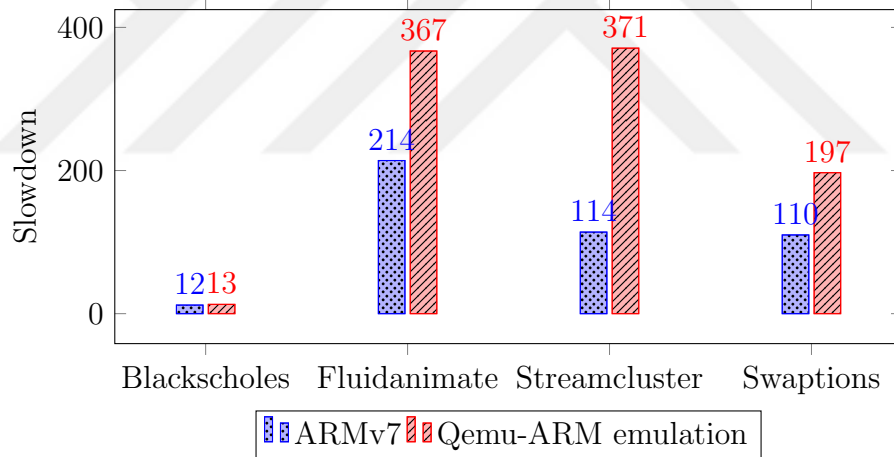


Figure 7.7: Slowdown comparison of race detection on ARMv7 vs on Qemu-ARM

Chapter 8

CONCLUSION

In this dissertation, we have discussed techniques we researched for detecting data races for shared memory programming models at runtime. In our approaches, we have targeted the commonly used models: (i) OpenMP tasks, (ii) the shared memory dataflow programming models such as Atomic DataFlow (ADF), and (iii) the POSIX Threads (Pthread) for embedded systems.

We proposed a method to detect determinacy races in OpenMP tasks where unintended missing dependency between tasks can result in nondeterministic execution. We define happens-before relation among tasks based on their dependencies for determining an execution order when detecting determinacy races and implement our algorithm as a tool on top of ThreadSanitizer. We evaluated our solution with a set of small applications in terms of bug detection and overhead. The tool successfully finds bugs in benchmarks and its efficiency is reasonable.

We presented an output nondeterminism detection technique for parallel programming models that combine dataflow semantics with shared memory programming constructs. We provided a *happens-before* relation definition between dataflow tasks and formulated our output nondeterminism detection technique upon that. We also devised a commutativity check for minimizing the reported false positives on commuting tasks. To show the effectiveness of our technique, we implemented an output nondeterminism detection tool called *DFinspec* for ADF applications by instrumenting them at compile time using LLVM compiler infrastructure. We tested the tool on several applications written in ADF programming model and reported real and synthesized bugs. It is an open research area to expand our tool by providing support for similar other tasking models like the Intel TBB which can benefit a wider community.

Finally, we presented a tool called *EmbedSanitizer*, for detecting data races for applications targeting 32-bit ARM architecture. *EmbedSanitizer* extends *ThreadSanitizer*, a race detection tool widely accessible through Clang and GCC, by enhancing its instrumentation. Moreover, we implemented our own 32-bit version of race detection runtime to replace *ThreadSanitizer*'s race detection runtime which is incompatible with 32-bit ARM. Our custom race detection library adopts FastTrack, an efficient and precise happens-before based algorithm. To evaluate the consistency of *EmbedSanitizer*, we used four PARSEC benchmark applications. First, we evaluated the precision of the tool by comparing the race report behavior with that of *ThreadSanitizer*. Next, we compared its slowdown with running race detection on the Qemu emulator as a representative for testing ARM code in a high-end developer platform.

Detecting races in shared memory programming models is NP-hard. To detect all races in a program, all possible executions of a program need to be done and their traces checked for races. This dissertation does not propose solutions for detecting all races in a program. However, it proposes methods for detecting races in programming models where such methods are insufficient.

BIBLIOGRAPHY

- [1] Embedsanitizer: <https://www.github.com/hassansalehe/embedsanitizer>.
- [2] OpenMP 3.0 API, www.openmp.org/wp-content/uploads/spec30.pdf.
- [3] OpenMP 4.0 Complete Specifications, <http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>.
- [4] OpenMP Specifications, Available: <http://openmp.org/>.
- [5] `std::unordered_map`. http://en.cppreference.com/w/cpp/container/unordered_map.
- [6] ThreadSanitizer Documentation, <https://clang.llvm.org/docs/ThreadSanitizer.html>.
- [7] Valgrind DRD, 2017.
- [8] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
- [9] S. Atzeni, G. Gopalakrishnan, Z. Rakamarić, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller. Archer: Effectively spotting data races in large openmp applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 53–62, May 2016.
- [10] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

- [11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995.
- [12] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10*, pages 255–268, New York, NY, USA, 2010. ACM.
- [13] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '03*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.
- [14] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [15] P. M. Carpenter, A. Ramirez, and E. Ayguade. *Euro-Par 2010 - Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part I*, chapter Starsscheck: A Tool to Find Errors in Task-Based Parallel Programs, pages 2–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [16] R. Chen, X. Guo, Y. Duan, B. Gu, and M. Yang. Static data race detection for interrupt-driven embedded software. In *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement - Companion*, pages 47–52, June 2011.
- [17] R. Chen, X. Guo, Y. Duan, B. Gu, and M. Yang. Static data race detection for interrupt-driven embedded software. In *2011 Fifth International Conference*

- on Secure Software Integration and Reliability Improvement - Companion*, pages 47–52, June 2011.
- [18] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.
- [19] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV*, pages 85–96, New York, NY, USA, 2009. ACM.
- [20] D. Dimitrov, M. Vechev, and V. Sarkar. Race detection in two dimensions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '15*, pages 101–110, New York, NY, USA, 2015. ACM.
- [21] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Coptly, R. Dietrich, X. Liu, E. Loh, and D. Lorenz. *OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis*, pages 171–185. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [22] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.
- [23] A. Fernández, V. Beltran, X. Martorell, R. Badia, E. Ayguad, and J. Labarta. Task-based programming with ompss and its application. In L. Lopes, J. ilinskas, A. Costan, R. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 601–612. Springer International Publishing, 2014.

- [24] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, pages 121–133, New York, NY, USA, 2009. ACM.
- [25] C. Flanagan and S. N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '10*, pages 1–8, New York, NY, USA, 2010. ACM.
- [26] V. Gajinov, S. Stipić, I. Erić, O. S. Unsal, E. Ayguadé, and A. Cristal. Dash: A benchmark suite for hybrid dataflow and shared memory programming models: with comparative evaluation of three hybrid dataflow models. In *Proceedings of the 11th ACM Conference on Computing Frontiers, CF '14*, pages 4:1–4:11, New York, NY, USA, 2014. ACM.
- [27] V. Gajinov, S. Stipic, O. Unsal, T. Harris, E. Ayguade, and A. Cristal. Integrating dataflow abstractions into the shared memory model. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 243–251, Oct 2012.
- [28] O.-K. Ha and Y.-K. Jun. An efficient algorithm for on-the-fly data race detection using an epoch-based technique. *Sci. Program.*, 2015:13:13–13:13, Jan. 2015.
- [29] S. Hong and M. Kim. A survey of race bug detection techniques for multithreaded programmes. *Software Testing, Verification and Reliability*, 25(3):191–217, 2015.
- [30] N. Hunt, T. Bergan, L. Ceze, and S. D. Gribble. Ddos: Taming nondeterminism in distributed systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13*, pages 499–508, New York, NY, USA, 2013. ACM.

- [31] Intel. Intel Inspector XE. <https://software.intel.com/en-us/intel-inspector-xe>, 2017.
- [32] IntelTBB. Intel threading building blocks. http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm, 2017.
- [33] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the 19th International Conference on Computer Aided Verification, CAV'07*, pages 226–239, Berlin, Heidelberg, 2007. Springer-Verlag.
- [34] G. Kestor, O. S. Unsal, A. Cristal, and S. Tasiran. T-rex: A dynamic race detection tool for c/c++ transactional memory applications. In *Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14*, pages 20:1–20:12, New York, NY, USA, 2014. ACM.
- [35] S. Keul. Tuning static data race analysis for automotive control software. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pages 45–54, Sept 2011.
- [36] I. Kuru, H. S. Matar, A. Cristal, G. Kestor, and O. Unsal. Parv: Parallelizing runtime detection and prevention of concurrency errors. In S. Qadeer and S. Tasiran, editors, *Runtime Verification*, pages 42–47. Springer Berlin Heidelberg, 2013.
- [37] J. Labarta. Starss: A programming model for the multicore era. In *PRACE Workshop "New Languages & Future Technology Prototypes" at the Leibniz Supercomputing Centre in Garching (Germany)*, 2010.
- [38] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

- [39] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
- [40] I.-T. A. Lee and T. B. Schardl. Efficiently detecting races in cilk programs that use reducer hyperobjects. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '15*, pages 111–122, New York, NY, USA, 2015. ACM.
- [41] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.
- [42] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin. Dataracebench: A benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17*, pages 11:1–11:14, New York, NY, USA, 2017. ACM.
- [43] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05*, pages 190–200, New York, NY, USA, 2005. ACM.
- [44] H. S. Matar, E. Mutlu, S. Tasiran, and D. Unat. Output nondeterminism detection for programming models combining dataflow with shared memory. *Parallel Computing*, 71:42 – 57, 2018.
- [45] H. S. Matar, S. Tasiran, and D. Unat. *EmbedSanitizer: Runtime Race Detection Tool for 32-bit Embedded ARM*, pages 380–389. Springer International Publishing, Cham, 2017.

- [46] H. S. Matar and D. Unat. Source code and user guide for euro-par 2018 paper: Runtime determinacy race detection for openmp tasks. Figshare (2018). Code., <https://doi.org/10.6084/m9.figshare.6392252>, 2018.
- [47] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev. Sdnracer: Detecting concurrency violations in software-defined networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15*, pages 22:1–22:7, New York, NY, USA, 2015. ACM.
- [48] E. Mutlu, V. Gajinov, A. Cristal, S. Tasiran, and O. Unsal. Dynamic verification for hybrid concurrent programming models. In B. Bonakdarpour and S. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 156–161. Springer International Publishing, 2014.
- [49] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06*, pages 308–319, New York, NY, USA, 2006. ACM.
- [50] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.
- [51] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, Mar. 1992.
- [52] M. Olszewski, Q. Zhao, D. Koh, J. Ansel, and S. Amarasinghe. Aikido: Accelerating shared data dynamic analyses. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII*, pages 173–184, New York, NY, USA, 2012. ACM.

- [53] A. Pop and A. Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, Jan. 2013.
- [54] E. Pozniansky and A. Schuster. Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurrency and Computation: Practice & Experience*, 19(3):327–340, 2007.
- [55] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Practical static race detection for c. *ACM Trans. Program. Lang. Syst.*, 33(1):3:1–3:55, Jan. 2011.
- [56] S. Qadeer and S. Tasiran. Runtime verification of concurrency-specific correctness criteria. *International Journal on Software Tools for Technology Transfer*, 14(3):291–305, 2012.
- [57] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In *Proceedings of the First International Conference on Runtime Verification, RV’10*, pages 368–383, Berlin, Heidelberg, 2010. Springer-Verlag.
- [58] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’12*, pages 531–542, New York, NY, USA, 2012. ACM.
- [59] C. Sadowski, S. N. Freund, and C. Flanagan. *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, chapter SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs, pages 394–409. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

- [60] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, SOSP '97*, pages 27–37, New York, NY, USA, 1997. ACM.
- [61] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications, WBIA '09*, pages 62–71, New York, NY, USA, 2009. ACM.
- [62] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *ACM SIGPLAN Notices*, volume 47, pages 387–400. ACM, 2012.
- [63] R. Surendran and V. Sarkar. Dynamic determinacy race detection for task parallelism with futures. In Y. Falcone and C. Sánchez, editors, *Runtime Verification*, pages 368–385, Cham, 2016. Springer International Publishing.
- [64] G. M. Tchamgoue, K. H. Kim, and Y.-K. Jun. *Dynamic Race Detection Techniques for Interrupt-Driven Programs*, pages 148–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [65] G. M. Tchamgoue, K. H. Kim, and Y.-K. Jun. Verification of data races in concurrent interrupt handlers. *International Journal of Distributed Sensor Networks*, 9(11):953–593, 2013.
- [66] M. Vechev, E. Yahav, R. Raman, and V. Sarkar. Automatic verification of determinism for structured parallel programs. In *Proceedings of the 17th International Conference on Static Analysis, SAS'10*, pages 455–471, Berlin, Heidelberg, 2010. Springer-Verlag.
- [67] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software*

engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 205–214. ACM, 2007.

- [68] B. Wire. Parallocity licenses zeus virtual machine dynamic analysis framework to h3c technologies. <http://www.businesswire.com/news/home/20121211005482/en/Parallocity-Licenses-Zeus-Virtual-Machine%E2%80%99s-ZVM-U-Dynamic-Analysis>, 2012.
- [69] B. Wire. Akamai selects parallocity’s zvm-u dynamic software analysis framework. <http://www.businesswire.com/news/home/20130305005107/en/Akamai-Selects-Parallocity%E2%80%99s-ZVM-U-Dynamic-Software-Analysis>, 2013.
- [70] X. Wu, Y. Wen, L. Chen, W. Dong, and J. Wang. Data race detection for interrupt-driven programs via bounded model checking. In *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, pages 204–210, June 2013.
- [71] Y. Xu, I.-T. A. Lee, and K. Agrawal. Efficient parallel determinacy race detection for two-dimensional dags. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP ’18*, pages 368–380, New York, NY, USA, 2018. ACM.
- [72] M. Yu, S.-M. Park, I. Chun, and D.-H. Bae. Experimental performance comparison of dynamic data race detection techniques. *ETRI Journal*, vol. 39, no. 1, Feb. 2017, 39(1):124–134, Feb. 2017.