

Race Detection Techniques for Applications using Asynchronous Programming Models

by

Erdal Mutlu

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 Science and Engineering



**KOÇ
UNIVERSITY**

September, 2016

**Race Detection Techniques for Applications using Asynchronous
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

Erdal Mutlu

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:

Assoc. Prof. Serdar Taşiran

Assist. Prof. Didem Unat

Assist. Prof. Cemal Yılmaz

Prof. Atilla Gürsoy

Assist. Prof. Tankut Barış Aktemur

Date: _____



To my family, my wife and friends

ABSTRACT

Asynchronous programming models has become ubiquitous in concurrent application development for the modern computation platforms, i.e. mobile, web and desktop applications. Similar to all concurrent programs, reasoning about asynchronous programs is challenging for the developers. Different than legacy concurrency constructs (i.e. threads), asynchronous constructs (i.e. callbacks) can invert the control and data flow as the execution order is not linear and may change depending on the dispatch of asynchronous events. This may result in subtle concurrency bugs that can have a big impact on the development cycle as it is notoriously hard to detect, reproduce, and fix them. Data races have been considered as the main source of unintended non-determinism in concurrent programs and seen as a symptom of a higher-level concurrency error. Although race detection in legacy concurrent programming models have attracted a great deal of attention from both the research communities and commercial counterparts, there is a critical need for race detection techniques for asynchronous programs.

To address these needs, this thesis introduces techniques for detecting data races on applications using asynchronous programming constructs as a way of concurrency. We introduce race detection technique that is applicable to different asynchronous programming models. We focused mainly on JavaScript web applications and applications using hybrid programming models that integrates dataflow programming construct into shared memory programming models. Our techniques attempts to combine advantages of both static and dynamic analysis. We executed the application code only *once*, yet we explore *multiple* execution order with the help of our static analysis approach. The key idea behind our analysis technique is that it encodes the effects of concurrent updates on the shared state by merging the values as opposed

to separately considering different orderings. One way to see our approach is that we explore *neighboring* schedules for a particular runtime execution by analyzing possible re-orderings of the concurrent entities with a single pass over the observed execution.

As the first part of our study, we explored JavaScript web applications using asynchronous constructs for user interaction and server requests. First, we conducted empirical studies for identifying the real damages of data races in the web applications by devising randomized and systematic exploration of server requests (`XMLHttpRequest`s) orderings. As opposed to other types of applications, web applications have a more forgiving nature for data races on *transient state*, since their effects are ephemeral. This is because these races only occur in invisible to the user portions of the program state, or, at worst, UI glitches that are either unnoticed by the user, or disappear if the user reloads the page. In the light of our empirical study, we suggested that a more useful way to think about data races on the web is by focusing on *persistent state*, such as client-local cookies, `localStorage` and `sessionStorage` mechanisms, as well as server-based side effects. We evaluated our technique on 26 web sites that heavily uses asynchronous constructs, we demonstrate that benign races (on transient state) are considerably more common than harmful ones (on persistent state). Finally, we conducted an exploratory study on the reported race related bugs on JavaScript libraries that uses new asynchronous constructs: `Promise`s.

In the second part of the thesis, we worked on hybrid programming models combining dataflow constructs with share memory programming model. These hybrid programming models provide programmers with dataflow abstractions for defining tasks as the main execution unit with corresponding data dependencies between each other. Contrary to the pure dataflow model which assumes side-effect free execution of the tasks, these models allow tasks to share the data using some form of thread synchronization, such as locks or transactional memory (TM). We devised a randomized exploration technique with probabilistic guarantees for exploring possible execution orders of the tasks composing the application. Later, we devised a modular race de-

tection technique which can be used for detecting data races in hybrid programming models. We presented a modular representation of *happens before* relations (execution orders) imposed by different programming models, i.e. dataflow and shared memory, which can be used for extending the detection mechanism for different type of shared memory synchronization systems, i.e. lock-based, transactional memory. Finally, we implemented our detection mechanism as a generic tool prototype that can be easily extended for different type of hybrid programming models.



ÖZETÇE

Asenkron programlama modelleri, mobil, masaüstü ve web uygulamaları gibi modern bilişim platformlarındaki eşzamanlı geliştirme uygulamalarında yaygın kullanımdadır. Bütün eşzamanlı programlarda olduğu gibi, asenkron programlarda da akış mantığını oturtmak geliştirici için zorlayıcıdır. Geçmişten günümüze gelmiş eşzamanlı yapılardan (örn. iş parçacıkları) farklı olarak, asenkron yapılarda (örn. geri çağırma) yürütmenin doğrusal bir sıralaması olmadığından ve bu sıralama asenkron olayların icrasına bağlı değişebildiğinden, bu yapılar kontrol ve veri akışını tersine çevirebilir. Bu durumda yakalaması güç eşzaman hataları ortaya çıkabilir ve bu hatalar farkedilmesi, yeniden üretilmesi ve tamir edilmesi zor olduğundan yazılım sürecinde ciddi etkilere sebep olabilir. Yarış durumları, eşzamanlı programlarda istemsiz belirsizliğin ana kaynağı olarak görülmekte ve yüksek seviyelerdeki eşzaman hatalarının göstergesi olarak görülmektedir. Geçmişten günümüze gelmiş eşzamanlı programlama modellerinde yarış durumlarını ortaya çıkarmak için hem araştırma çevrelerinden hem de sanayi çevrelerinden yoğun ilgi gösterilmiş olsa da, asenkron programlama modellerinde yarış durumlarını ortaya çıkarmak için kullanılacak metodlara kritik bir ihtiyaç vardır.

Bu tez, bahsedilen ihtiyaçları karşılamak için, eşzaman methodu olarak asenkron programlama yapıları kullanan uygulamalarda yarış durumlarını ortaya çıkarmak için yeni teknikler sunmaktadır. Sunduğumuz bu yarış durumu belirleme tekniği farklı asenkron programlama modelleri üzerine uygulanabilir. Ana olarak, JavaScript web uygulamaları ve veriakışı programlama yapılarını paylaşımlı hafıza üzerine uygulayan melez programlama modelleri üzerinde odaklandık. Tekniğimiz; statik ve dinamik analiz yaklaşımlarının avantajlarını biraraya getirmeyi amaçlamaktadır. Yürütme kodu sadece bir kere çalıştırılır, fakat birçok yürütme sıralaması statik analiz yaklaşımı

kullanılarak incelenir. Bu tekniğin temelindeki görüş, tekniğin paylaşımlı hafıza noktaları üzerindeki eşzamanlı değişimleri farklı sıralamalarda birbirinden bağımsız incelemek yerine birleştirilmiş bir yapıda saklamasından gelir. Tekniğimizi açıklamanın bir yolu, belirli bir yürütme için olası farklı sıralamaları gözönünde bulundurarak, komşu yürütme sıralamalarını tek bir yürütme üzerinden incelemektir.

Çalışmamızın ilk adımı olarak, JavaScript web uygulamalarındaki asenkron yapıları kullanıcı etkileşimi ve sunucu yanıtları üzerinden inceledik. Öncelikle, yarış durumlarının web uygulamalarındaki zararlarını belirleyebilmek için rastgele seçilmiş ve sistematik seçilmiş sıralamalı sunucu istemleri (`XMLHttpRequest`) kullanılarak deneysel çalışmalar yapılmıştır. Diğer uygulama çeşitlerine nazaran, web uygulamaları geçici durumlarda gözlenen yarış durumları konusunda, etkilerin kısa ömürlü olmasından dolayı daha affedici bir yapıya sahiptir. Bunun sebebi, bu yarış durumlarının sadece programın gözle görülmeyen kullanıcı kısımlarında oluşması ya da, en kötü ihtimalle, kullanıcı arayüzünde farkedilmeyen veya sayfayı yenileyerek yok edilebilen aksaklıklardan oluşmasındandır. Bu deneysel çalışmanın sonuçlarından yola çıkarak, web uygulamalarındaki yarış durumlarında, kalıcı durum üzerinde odaklanmanın daha kullanışlı olacağını önerdik. Bu kalıcı durumlar kullanıcı tarafında saklanan çerezler, `localStorage` ve `sessionStorage` mekanizmaları olabileceği gibi sunucu tarafındaki yan etkiler de olabilir. Geliştirdiğimiz tekniği, asenkron yapıları yoğun şekilde kullanan 26 web sitesi üzerinde değerlendirdik ve zararsız yarışların (geçici durumlarda gözlenen) zararlı yarışlardan (kalıcı durumlarda gözlenen) çok daha yaygın olduğunu gözlemledik. Son olarak, JavaScript kütüphanelerinde yarış durumu ile ilgili hataları bildirmek için kullanılan yeni bir asenkron yapı - `Promise` ler- üzerinde bir araştırma çalışması hazırladık.

Tezin ikinci kısmında, veriakışı yapılarını paylaşımlı hafıza programlama modelleriyle biraraya getiren melez programlama modelleri üzerinde çalıştık. Bu melez programlama modelleri, programcılara görevleri ana yürütme birimi olarak ve aralarındaki veri bağıntılarıyla birlikte tanımlamak için veriakışı soyutlaması sağlar. Saf

veriakışı modelinin tersine, ki bu model görevlerin yan etki olmadan yürütüleceğini varsayar, melez modeller görevlerin iş parçacığı senkronizasyonu kullanarak (örn. kilit ve işlem belleğı) veri paylaşmasına olanak verir. Olası yürütme sıralamaları ve uygulamayı oluşturan görevleri araştırmak için, olasılık garantili rastgeleleştirilmiş bir araştırma tekniğı geliştirdik. Daha sonra, melez programlama modellerinde yarış durumlarını ortaya çıkarmak için kullanılacak modüler bir yarış belirleme tekniğı geliştirdik. Farklı programlama modelleri tarafından -örneğin veriakışı ve paylaşımli hafıza- dayatılan önce-olur ilişkileri (yürütme sırası) için modüler bir gösterim sunduk. Bu programlama modellerine örnek olarak veriakışı ve paylaşımli hafıza verilebilir, ki bunlar belirleme mekanizmasını kilit bazlı ya da işlem hafızası bazlı senkronizasyon sistemleri gibi farklı tipteki paylaşımli hafızalar için genişletmede kullanılabilir. Son olarak, geliştirdiğimiz yarış belirleme mekanizmasını genel bir prototip araç olarak kodladık, böylece geliştirdiğimiz bu araç farklı tiplerdeki melez programlama modelleri için de kolayca genişletilebilir.

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to my thesis advisor, Serdar Taşiran, for his guidance, patience and support throughout my Ph.D. studies. His experience and wisdom added a lot to my graduate experience and inspired me to be a better researcher. I am really grateful for countless opportunities, he has provided me from the beginning of this journey. I also wanted to thank the other committee members of this thesis, Didem Unat, Cemal Yılmaz, Atilla Gürsoy, and Tankut Barış Aktemur for critical reading of this thesis and for their guidance in making this thesis better. I would also give special thanks to Benjamin Livshits from Microsoft Research (MSR) whose valuable guidance and advices helped me in adapting intensive academical research.

I am very thankful to Türk Telekom for funding my research, HiPEAC for giving me the opportunity to have an industrial internship in ST Microelectronics, and Euro-TM for funding my short term visit to Barcelona Supercomputing Center (BSC) during my studies. Although these financial support is hugely appreciated, the greatest benefit for me was to be able to work closely with countless high-class researchers. I am really grateful to my advisors Francesco Papariello and Giuseppe Desoli from ST Microelectronics, and Osman Ünsal and Adrian Cristal from Barcelona Supercomputing Center (BSC) for showing me different aspects of computer science and engineering.

I would like to give my thanks to all my friends from Koç University, for their friendship and support, especially the old and new members of ENG-230, İsmail, Ömer, Can, Hassan, İdil, Ayşenur, Burcu, Süha, Erkut, and Maryam. I am also grateful to my close friends Oya, Gülçin, Berfin, Berkay, Erman, İbrahim, Elif, Buğra, and countless others from Sabancı University for always being there for me.

I must give immense gratitude to my parents Yakup and Mbeyyen, my aunt Mbeccel, and my grandparents for their endless support and patience. Finally, I would like to give my heartily thanks to my lovely wife Burcu for her extreme support and for always believing me. Without her patience and wisdom, it will be impossible for me to continue in hard days of my life.



TABLE OF CONTENTS

List of Tables	xvii
List of Figures	xviii
Nomenclature	xx
Chapter 1: Introduction	1
Chapter 2: Background and Related Work	9
2.1 Asynchronous Programming Models	9
2.1.1 JavaScript Execution Model	10
2.1.2 Dataflow on Shared Memory Execution Model	12
2.2 Data Race Detection Literature	16
2.2.1 Race Detection for JavaScript Applications	16
2.2.2 Race Detection for Applications using Hybrid Programming Models	19
2.3 Summary	22
I Race Detection for JavaScript Web Applications	24
Chapter 3: Introduction	25
Chapter 4: Observable Races in JavaScript Applications	28
4.1 Overview	28
4.1.1 Exposing Races with Systematic Exploration	31
4.1.2 Smoking Gun	32

4.1.3	Prioritizing Races Using a Crowd	34
4.1.4	Schedule Shepherding	36
4.1.5	Key ideas	36
4.2	Randomized Exploration	37
4.3	Systematic Exploration	39
4.4	Discussion	41
4.5	Summary	42
Chapter 5:	Detecting JavaScript Races that Matter	44
5.1	Overview	44
5.1.1	What is a Data Race?	45
5.1.2	Motivating Examples	48
5.1.3	Trace Processing	50
5.1.4	Algorithm Summary	53
5.1.5	Implementation	54
5.2	Formalization	55
5.2.1	Defining Happens-Before	56
5.2.2	Modeling and Analysis of Traces	57
5.2.3	Defining Traces	58
5.2.4	Interpreting Traces	60
5.2.5	Detecting Races	61
5.3	Evaluation	63
5.3.1	Experimental Setup	67
5.3.2	Detection Results	68
5.3.3	False Positives	73
5.3.4	Promise Related Race Conditions	74
5.4	Discussion	76
5.5	Conclusions	77

II Race Detection for Applications using Hybrid Programming Models	78
Chapter 6: Introduction	79
Chapter 7: Randomized Testing Scheduler for Hybrid Programming Models	82
7.1 Motivation	82
7.2 System Overview	84
7.2.1 Probabilistic Concurrency Testing	84
7.2.2 Our Method and Implementation	84
7.3 Summary	88
Chapter 8: Race Detection for Programs using Hybrid Programming Models	89
8.1 Overview	89
8.1.1 Data Races in Hybrid Programming Models	91
8.2 Preliminary	92
8.2.1 Actions in Hybrid Programming Models	93
8.2.2 Defining Happens-Before Relation and Data Race	94
8.3 Race Detection Mechanism	96
8.3.1 Modeling and Analysis of Traces	96
8.3.2 Defining Traces	97
8.3.3 Interpreting Traces	97
8.3.4 Detecting Races	100
8.4 Race Detection Implementation	101
8.4.1 Race Detection Tool Implementation	101
8.5 Preliminary Experiments	105
8.6 Conclusion	107

Chapter 9: Future Work and Conclusion	108
9.1 Limitations and Future Work	108
9.1.1 Limitations of Race Detection for JavaScript Applications . .	108
9.1.2 Limitations of Race Detection for Hybrid Applications	109
9.2 Conclusion	110
Bibliography	115



LIST OF TABLES

4.1	Top 20 XHR-heavy Web sites in our experiments.	40
5.1	Characteristics of our benchmarks web sites.	64
5.2	Trace statistics and processing times.	65
5.3	Races found by our analysis.	66
5.4	Node.js modules with reported issues related to race conditions on Promise constructs.	75
8.1	Preliminary analysis results for hybrid applications.	106

LIST OF FIGURES

2.1	JavaScript execution model components [13].	10
2.2	Basic event loop implementation.	12
2.3	JavaScript asynchronous execution example.	13
2.4	Computing maxima using: a) dataflow, b) shared memory and c) hybrid programming models.	14
3.1	Typical use of XHR.	26
4.1	Screenshots obtained from two different XHR timings for an artificially- broken version of the FedEx Web site.	29
4.2	Original and modified XHR timings for the artificially-broken version of the FedEx site.	30
4.3	Overall distribution of response times for the Optimizely snippet world- wide, with the average response time.	33
4.4	Total mobile browser distribution.	35
4.5	XHR chaining.	39
5.1	Multiple XHR race example.	51
5.2	Sample trace illustrating cookie races.	52
5.3	Merging two states after an XHR.	53
5.4	Trace analysis rules.	59
5.5	<code>sessionStorage</code> manipulation in <code>milliyet.com.tr</code>	69
5.6	Race on <code>innerHTML</code> manipulation in <code>radikal.com.tr</code>	70
5.7	Race on <code>document.cookie</code> manipulation in <code>gazetta.it</code>	72
5.8	Trace from <code>optimum.net</code> illustrating a false positive.	73

7.1	Motivating buggy example for hybrid programming models.	83
7.2	Probabilistic Concurrency Testing (PCT) algorithm.	85
8.1	Data race resulted by incorrect synchronization between tasks.	91
8.2	Hybrid race detection trace analysis rules.	98
8.3	Memory access trace sample from instrumented Atomic DataFlow (ADF) runtime.	102
8.4	Race detection implementation overview.	103



NOMENCLATURE

ADF	Atomic DataFlow Programming Model
API	Application Programming Interface
CPU	Central Computing Unit
DOM	Document Object Model
GPU	Graphics Processing Unit
HAR	HTTP Archive
HPC	High Performance Computing
IDE	Integrated Development Environment
IoT	Internet of Things
JS	JavaScript
JSON	JavaScript Object Notation
OS	Operating System
PC	Personal Computer
PCT	Probabilistic Concurrency Testing
TM	Transactional Memory
TBB	Thread Building Blocks
UI	User Interface
XHR	XMLHttpRequest

Chapter 1

INTRODUCTION

Today, concurrent programming models that leverage multiple CPU and GPU cores within the modern computation platforms replaced the way programmers think and write code. While it was more convenient to write sequential desktop programs for old platforms (i.e. personal computers (PCs), high performance computers (HPCs)), today computation platforms (i.e. mobile devices, heterogeneous hardware devices) became more concurrent, distributed and asynchronous. Writing concurrent/asynchronous programs is a genuinely difficult problem for the developers. Because of the non-deterministic ordering of multiple threads and/or asynchronous sources, handling program state correctly is a challenging task. Although correctness definition of a program may change depending on its specification, one common correctness criteria for concurrent/asynchronous programs is being data-race-free. Generally, concurrent programming models try to ease program development by exposing synchronization mechanisms (i.e. locks, barriers etc) for limiting the possible ordering of actions hence avoiding the undesired data races in the program. Lately, there is a growing demand for asynchronous programming models as responsiveness became increasingly important on modern computing platforms, i.e. mobile, web and desktop applications.

With the emergence of always-connected mobile platforms, it has become crucial to achieve high availability for user interaction within limited hardware resources. For solving these problems, dynamic programming languages like JavaScript and asynchronous programming constructs became very popular for not only mobile [1, 6] and web application [2] developments but also new universal OS application [5] developments. In event-driven programming models, developers define asynchronous events

like user interactions (i.e. mouse/button clicks) or network events (i.e. `XMLHttpRequest`/Ajax calls) and their corresponding responses from the run-time system. Until recently, callbacks are the accepted way of defining these actions for specific asynchronous events, but some undesired/untrusted properties of callbacks [48, 99] resulted in rise of another asynchronous constructs called **Promise**s. Although **Promise**s were proposed decades ago [31], it became even more widely used within modern mobile platforms. With **Promise** constructs, developers can define future values for asynchronous events and can assign not only single actions but also sequences of actions by chaining **Promise**s on these values. The assigned (sequence of) actions are only triggered once the corresponding future values became materialized as a result of an asynchronous action (i.e. user action or network response).

On another front of the modern programming models, data-driven asynchronous programming models are becoming popular. Legacy concurrent programming models have to make trade-offs between having heavy, coarse grained locks and light, fine-grained locks. Recently proposed hybrid programming models [34, 78, 80], try to integrate dataflow programming constructs into shared memory programming models to avoid unnecessary usage of synchronization mechanisms. The main novelty of these proposals is that developers are given the freedom of defining their programs as dataflow graphs where the nodes are execution tasks and edges are the input-output relation between these tasks while preserving a shared state between possibly parallel task execution. With the success of these proposals, multiple commercial versions are adopted into well known concurrent programming libraries like Intel's TBB [3], Microsoft's Task Parallel Library [4] and open source counter-parts like in [7]. With the existence of these dataflow graphs, underlying framework can make assumptions about the ordering of events without the need of explicit synchronization mechanisms. Another advantage of these models is the possibility of distributing the tasks to different underlying hardware like multiple CPU and GPU cores. These features make these hybrid programming models ideally suited for emerging high performance computing (HPC) and stream processing applications.

As the case with all concurrent programming models, reasoning about asynchronous programs is challenging for the developers. Different than legacy concurrency constructs (i.e. threads), asynchronous constructs (i.e. callbacks) can invert the control and data flow as the execution order is not linear and may change depending on the dispatch of asynchronous events. Recent surveys and studies [57, 32, 77] also focuses on the common difficulties that programmers have while using these asynchronous constructs. Misuse of these constructs may result in subtle concurrency bugs that can have a big impact on the development cycle. Even worse, these bugs can occur in production resulting in a bigger impact on the application. The main culprit of these kind of bugs, is data races happening on concurrently running entities. In the last decade, numerous race detection mechanisms [91, 29, 23, 104, 106, 17, 63, 98, 16] are proposed for uncovering data races in legacy concurrent programming models. While emerging asynchronous programming models becoming popular each passing day, there is very limited such development tools, i.e. race detection tools, for the developers.

To address these problems, this thesis introduces techniques for detecting data races on programs using asynchronous programming constructs as a way of concurrency. We introduce race detection techniques that are applicable to asynchronous programming models. However, we focused mainly on JavaScript web applications and applications using hybrid programming models that integrates dataflow programming construct into shared memory programming models.

The key challenges in developing race detection, are minimizing the reported false positives and maximizing the efficiency of the detection techniques. Proposed static race techniques [24, 28, 74, 105] analyze the source code of the program without actually executing it. Although these techniques achieve high coverage on the source code by reasoning about multiple program paths at once, they also tend to report many false positives. On the other hand, dynamic race detection techniques [91, 104, 106, 17, 29, 23, 63, 98, 16, 65] typically check for races over a recorded execution by monitoring the accesses to the shared memory locations and synchroniza-

tion operations. Different than static counterparts, dynamic techniques can achieve low number of false positives, however, they miss races that are not present in the observed execution. Another disadvantage of dynamic techniques is the instrumentation overhead (ThreadSanitizer [98] introduces 5x-15x slowdown on the overall execution) for recording relevant information (i.e. memory access, synchronization operations) required for the analysis.

On the other hand, there are other challenges that are more programming language specific, i.e. dynamic languages like JavaScript. The use of dynamic features like `eval` functions, which is commonly used [88, 89] for dynamic code generation at runtime, makes developing static analysis challenging. Most of the proposed static analysis techniques [8, 44, 102] for JavaScript applications totally forbid use of `eval` functions in their analysis. Another limitation in case of static race detection is precisely reasoning about the happens-before relation between asynchronous event handlers since the execution order may differ depending on the ordering of the actual events (i.e. user interactions, server request). On the other hand dynamic analysis, including various forms of symbolic exploration of possible schedules, falls victim to code coverage [92, 66].

For addressing these issues with different JavaScript analysis techniques, we proposed a race detection mechanism which attempts to combine advantages of both static and dynamic analysis. We execute the application only *once* for dynamically recording the observed memory accesses and dispatched asynchronous events, while statically exploring *multiple* execution ordering. Our technique constructs a *happens before* ordering between asynchronous execution entities and uses it for encoding possible re-ordering (neighboring) schedules of the initial trace. One way to see our approach is that it explores these *neighboring schedules* for a particular runtime execution by analyzing possible re-orderings of the concurrent entities. Our static analysis technique is based on encoding the effects of possible different schedules by merging the states and keeping track of multiple values written *concurrently*. As opposed to considering each possible re-ordering separately, we only construct a memory map

containing these merged values once. As such, our technique scales well for analyzing long executions with high number of asynchronous constructs.

Another challenge for developing race detection for asynchronous JavaScript applications is the lack of universally accepted formal *happens-before* definition. While previously proposed race detection techniques [79, 42] tries to define the *happens-before* relations over low-level memory accesses, the relation definition leads to being too restrictive causing many false positives. As opposed to these proposals, we define our *happens-before* relation on higher-level language constructs; execution blocks which can be *sequential* blocks as well as asynchronous *event handler* blocks. We argue that this definition matches well with the actual execution model as JavaScript code blocks are executed in a *run-to-completion* manner where execution is uninterrupted.

Next, we implemented a prototype race detection tool using our detection algorithm for JavaScript web application. We instrumented Firefox web browser [70] for collecting traces containing relevant memory access and event handler actions required by the detection mechanism. Our instrumented web browser can be used for browsing daily websites with negligible slow-down on the browsing experience. Finally, we evaluated the performance of our race detection tool on a set of applications with heavy use of asynchronous constructs, i.e. server requests through `XMLHttpRequest`.

Later, we generalized and extended our technique for hybrid programming models that combines dataflow constructs with shared memory programming models. Different than *event-driven* nature of JavaScript applications, these models employ a more structured *data-driven* concurrency where programmers describe their programs as a task graph with input/output dependencies. Typically, in these hybrid programming models the execution of tasks are governed by the availability of the input data and programmers are provided with shared memory synchronization mechanisms (i.e. locks, atomic blocks) for protecting concurrently accessed shared memory locations.

We propose a *modular happens-before* relation definition that can be constructed by using earlier *happens-before* definitions of dataflow and shared memory programming models (i.e. locks [62], transactional memory [49]). With this definition, the

race detection mechanism can be extended for different hybrid models that uses different shared memory synchronization mechanisms like locks or transactional memory. Finally, we implemented a generic race detector prototype for Atomic DataFlow (ADF) [34] applications which can be easily extended for different hybrid programming models.

Thesis Contributions:

Part I - Race Detection for JavaScript Web Applications

- We introduce the notion *observable races*, as determined by visual inspection of the web application and developed a dynamic analysis mechanism for systematically exploring all possible orderings of asynchronous calls in order to exercise potentially defective schedules.
- We propose a new view of benign and harmful data races in JavaScript web applications, and argue that harmful races should be the primary focus of analysis tools, due to them affecting the persistent client- or server-side state of the applications.
- We propose a lightweight exploration algorithm for finding data races in runtime traces of JavaScript programs. A key advantage for the scalability of our approach is that it does not require multiple program runs and can operate on the basis of a single execution via use of static analysis over the recorded execution trace.
- We analyze the presence of races in modern web applications, both in the context of micro-benchmarks collected from related work and various bug reports, and macro-benchmarks that include widely-used web sites. We find and investigate a total of 19 harmful and 621 benign races in 26 web sites, with only 2 observed false positives.

Part II - Race Detection for Applications using Hybrid Programming Models

- We present a randomized exploration tool for characterizing and exploring behaviors of programs written using hybrid programming models combining dataflow constructs with shared memory programming model.
- We formalize a modular *happens-before* relation for hybrid programming models combining dataflow constructs with shared memory programming model. The formalization can be easily extended for different hybrid programming models.
- We propose a lightweight race detection mechanism for applications using hybrid programming models. Our detection mechanism attempts to combine the advantages of static and dynamic analysis by recording a runtime execution trace and later doing an efficient static analysis for exploring neighboring schedules.
- We implement a generic race detector prototype that can be easily extended for different hybrid programming models. We provide software hooks for implementing new parser modules in order to ease implementation effort for different programming models and trace structures.

Thesis Outline:

- Chapter 2 provides background and state of the art for race detection and asynchronous programming models. It first describes asynchronous programming models and their execution models. Later, it defines *data races* in general and for specific programming models. Finally, it presents, state of the art on race detection and bug detection for different asynchronous models.
- Chapter 3 presents an introduction for the studies conducted for race detection in JavaScript races.
- Chapter 4 describes our systematic exploration technique for detecting races on JavaScript web applications. It first describes current state in data race

definition in JavaScript races and introduces a notion of *observable races* that determined by visual inspection of the resulting web page. Later, it describes mechanisms for exploring web pages in a randomized and systematic way. Finally, it provides a discussion on the research questions that has to be answered for a better understanding of races in the context of JavaScript web applications.

- Chapter 5 presents our race detection mechanism for JavaScript web applications. It starts with giving an overview of the races in context of JavaScript web applications using real-world examples and then describes our race detection mechanism informally. Later, it gives a formalization for the JavaScript applications and our detection mechanism. Finally, it gives evaluation of the detection mechanism on real-world JavaScript web applications.
- Chapter 6 gives an introduction for the studies conducted for race detection in applications using hybrid programming models that combine dataflow constructs with shared memory programming models.
- Chapter 7 describes a randomized exploration technique for hybrid programming models. First, it motivates the use of and the need for a verification tool for hybrid programming models. And later it describes our randomized behavior exploration tool.
- Chapter 8 presents our race detection mechanism for applications using hybrid programming models. It starts by giving a overview of the hybrid programming models and motivates the need for race detection tools over motivating examples. Secondly, it gives some preliminary description on these programming model. Later, it describes a formalization for a *modular happens-before* relation for hybrid programming models and formally describes our detection mechanism. Finally, it describes our implementation efforts for our generic race detection prototype for hybrid programming models.
- Chapter 9 discusses possible future directions and concludes the thesis.

Chapter 2

BACKGROUND AND RELATED WORK

In this chapter, we give background information about the asynchronous programming models. We also explain in detail the execution model of asynchronous programming models, namely JavaScript and dataflow on shared memory. Finally, we present the state of the art for race detection in asynchronous programs and give a conclusion.

2.1 *Asynchronous Programming Models*

Asynchronous programming models became widely popular with responsiveness becoming the number one requirement in most modern computing environments, from mobile devices to web servers. In general, processes in these models communicate between each other by posting tasks to queues of the corresponding process which is later executed. As opposed to general shared memory concurrent programming models, there is no need for heavy synchronization mechanism which can cripple the performance enormously. Today, one can see asynchronous constructs being used almost in any programming language and computing environment. However, implementing correct asynchronous programs is challenging since the execution being asynchronous makes following the control and data dependencies extremely hard. Recent studies [57, 77] shows the difficulties that programmers face while using these asynchronous programming models. With this thesis, we address the need for testing and verification effort in asynchronous programs by developing generic race detection tool for different asynchronous programming models. In the following sections, we will focus on describing the execution models of two asynchronous programming models

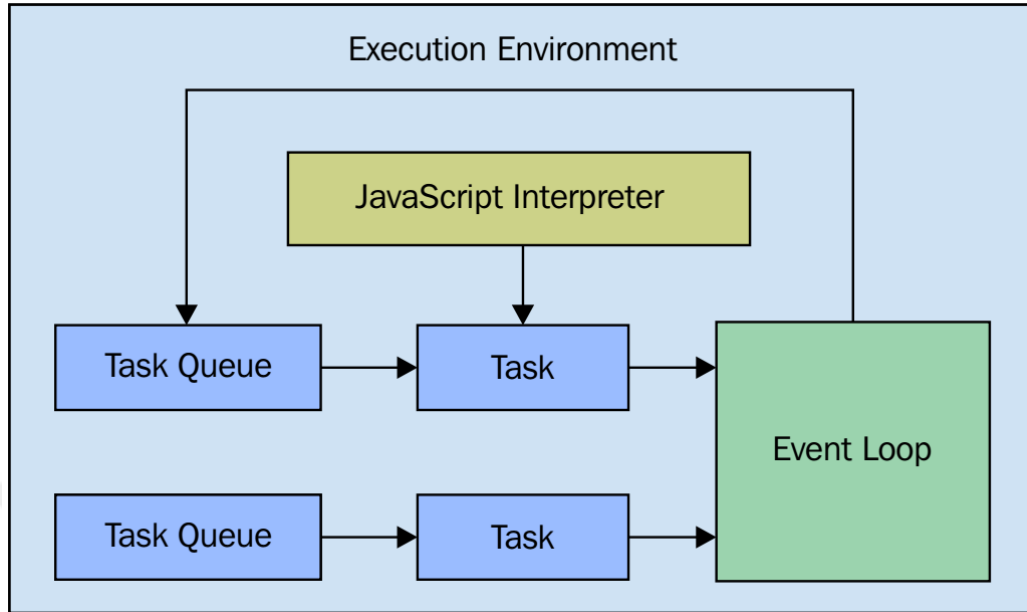


Figure 2.1: JavaScript execution model components [13].

that are the main focus of this thesis.

2.1.1 JavaScript Execution Model

JavaScript is designed as a *single threaded* and *run-to-completion* language in favor of simple execution mechanics and ease of development. However the nature of the web complicates the actual system by a great deal. Concurrent nature of today’s web pages and applications, which have to handle large amount of dynamically changing data on the run, enforces JavaScript to embody some kind of concurrency. Instead of using powerful multi-threading parallelism, JavaScript uses a *single threaded, event loop* based asynchronous execution model. In a JavaScript program, every code block is abstracted as tasks and are executed in a *run-to-completion* manner. In other words, while a task is being executed by the execution engine, it cannot be preemptively interrupted by another execution body. In general JavaScript applications have “event-driven” asynchrony meaning that the asynchronous executions happen due to handler function assigned for specific events.

Below we describe the components of a generic JavaScript execution model, as seen in Figure 2.1:

- **Execution environment:** JavaScript engine doesn't run in *isolation*, meaning that it runs within a hosting environment. This can be one of the wide range of environments from web browsers (e.g., Firefox [70], Windows Edge [67], Chrome [38]) or web servers (e.g., Node.js [76]) to Internet of Things(IoT) (e.g., JerryScript [90]). This environment includes everything that a JavaScript program can interact with, i.e. stack and heap memory, other web libraries.
- **JavaScript interpreter:** The component that is responsible for parsing JavaScript source code and generating corresponding jobs to be executed by the execution environment.
- **Task queue:** Every action that has to be fulfilled by the execution environment (e.g., execution of code block, handling an event) corresponds to a task in the environment. As a result each execution environment must at least have a task queue where each task is queued for execution. However, in most of the engines there are multiple task queues designated for different types of events (e.g., user interactions, rendering). The tasks in these queues are scheduled in a first-in-first-out (FIFO) fashion.
- **Event loop:** This component corresponds to the main execution mechanism which is named after its implementation (Figure 2.2). An execution environment only has one event loop which is responsible for executing each task from the available task queues. As JavaScript is executed in a *single threaded* manner, there is only one event loop in each environment.

The main characteristic of the asynchronous events in JavaScript is being *non-blocking* which enables to achieve impressive amount of performance even though the execution is performed by a single thread. The same principle is also behind the

```
1  while(queue.waitForMessage()){  
2      queue.processNextMessage();  
3  }
```

Figure 2.2: Basic event loop implementation.

server-side JavaScript with Node.js [76] as it uses a similar event loop for handling asynchronous requests efficiently.

Figure 2.3 gives a simple example for possible asynchronous event handlers in a JavaScript program. The execution of this source code will start by parsing the source code and triggering corresponding tasks to be handled by the execution environment. In this example, the program first defines two functions (line 2 and 7) and later they are assigned as specific event handlers, namely button click (line 15) and server request load events (line 21). However, they are not directly put to the task queues to be scheduled immediately, they are triggered once the corresponding events dispatched. In the end the execution of these code segments, after being parsed, will be order as follows: ③ → ④ → ⑤ → ⑥ → ⑦ → ⑧. As execution of this interpreted source code is executed in a *run-to-completion* fashion, even if registered events like button click event occurred, the execution will not be preempted to execute corresponding handler function, but instead it will be put to the task queue to be executed later. In the end, the execution order of ① and ② will depend on the timing of the actual events happening and corresponding event handler being put into the task queues to be scheduled in the next iteration of the event loop.

2.1.2 Dataflow on Shared Memory Execution Model

Using dataflow constructs on general shared memory programming model has become popular especially in high performance computing (HPC) applications because of its potential on the performance gain for a set of widely used applications. Shared memory multi-threading is ubiquitous in concurrent programs, however it is hard

```
1 // Callback for parsing server response
2 function onLoad(){ // 1
3     console.log('load', JSON.parse(this.responseText));
4 }
5
6 // Callback for button click event
7 function onClick(){ // 2
8     console.log('click', Date.getTime());
9 }
10
11 // Select the button element
12 var button = document.querySelector('button'); // 3
13
14 // Adding onClick event handler to button
15 button.addEventListener('click', onClick); // 4
16
17 // Creating server request object
18 var xhr = new XMLHttpRequest(); // 5
19
20 // Adding onLoad event handler to request
21 xhr.addEventListener('load', onLoad); // 6
22
23 // Send a ``GET`` request for ``data.json``
24 xhr.open('get', 'data.json'); // 7
25 xhr.send(); // 8
```

Figure 2.3: JavaScript asynchronous execution example.

to implement correct and efficient code. By contrast, in the dataflow programming model, the execution of an operation is constrained only by the availability of its input data – a feature that makes dataflow programming convenient and safe when it fits the problem at hand.

Recent proposal from academia [78, 34, 95, 80] and their commercial adaptations [3, 4, 7], provide programmers with dataflow abstractions for defining tasks as the main execution unit with corresponding data dependencies between each other. Contrary to the pure dataflow model which assumes side-effect free execution of the tasks, these models allow tasks to share the data using some form of thread synchronization, such as locks or transactional memory (TM) [54]. This way, they facilitate

<pre> in(x); in(y); in(g_max); { x = max(x, y); g_max = max(g_max, z); out(z); out(g_max); } </pre>	<pre> cond_wait(x); read(x); cond_wait(y); read(y); z = max(x, y); atomic{ if(z > g_max) g_max = z; } </pre>	<pre> in(x); in(y); { z = max(x, y); atomic { if(z > g_max) g_max = z; } out(z); } </pre>
a)	b)	c)

Figure 2.4: Computing maxima using: **a)** dataflow, **b)** shared memory and **c)** hybrid programming models.

implementation of complex algorithms for which shared state is the fundamental part of how the computational problem at hand is naturally expressed.

Figure 2.4 illustrates different versions of a “maxima” function implemented in pure dataflow, shared memory and the hybrid programming models. In this example, two input streams of integers, x and y , are given and the goal is to provide, for each pair of tokens consumed from these streams, the larger of the two values, and also to maintain the global maximum of all values received on both streams. In pure dataflow, maintaining the global maximum is inefficient (the max of two tokens needs to be fed back to the task for the next step) whereas in the shared memory model, the global maximum is treated as the mutable state which can easily be updated in-place. However, condition variables are needed to detect the availability of new input tokens. Hybrid programming models combine the best of two worlds. The atomicity of the code in the “atomic” block in Figure 2.4-b and c can be ensured using locks or transactional memory that are conventionally used in shared memory programming models.

Although underlying execution environment might change between hybrid programming models, the main components are as follows:

- **Work queues:** Execution of each task body is abstracted as a work in the execution environment. Hybrid programming models use queues for stacking the

enabled task bodies waiting to be scheduled. Similar to JavaScript programming model, hybrid models utilize use of multiple queues for different tasks. In the context of hybrid programs, different queues might be used for specific tasks and their instances.

- **Thread pools:** Actual execution of the tasks are carried out by threads created on the underlying shared memory system. In case of dataflow execution, most execution environment generates a thread pool in the beginning of the execution to avoid additional thread creation cost every time a task is enabled. The enabled tasks in the work queues are scheduled to available threads in the thread pool and executed concurrently.

Execution of a hybrid program is based on execution of tasks composing the program as their input data dependencies are satisfied. The execution, in general, is “data-driven” as task executions are triggered by input data. Each task in the program consumes a set of input data producing a set of output data which might trigger execution of other tasks in the task dependency graph. Later, each executed task will be waiting a new set of input data to be triggered.

At a given time in the execution, each task can be either enabled, which means all required input data is available, or disabled, meaning that it is waiting for some input data possibly from other tasks. In the beginning of the execution, all tasks described in the program are disabled by default. Later, the execution begins by triggering the initial tasks that start the execution of the tasks graph. As the inputs made available to the tasks, they became enabled and put to the *work queues* which are later scheduled by available threads in the *thread pool*. As the execution advances, tasks whose input data requirements are met are made enabled and put to the queue to be scheduled.

2.2 Data Race Detection Literature

Data races are always considered as symptoms of a higher level concurrency bugs in multithreaded programs. Even though there has been a great amount of proposals on detecting races in conventional multithreaded programs [91, 23, 29, 17, 106, 104, 98], the efforts for race detection on asynchronous programming models are still limited. In this section, we will focus on the literature for race detection for JavaScript applications and hybrid programs that use asynchronous programming models combining dataflow constructs with shared memory programming models.

2.2.1 Race Detection for JavaScript Applications

This section covers some of the recent work on finding races in JavaScript programs and asynchronous code as well as program analysis mechanisms for JavaScript applications.

Race detection for asynchronous programs: Hsiao *et al.* propose an approach to finding a subset of asynchronous races in Android applications, focusing on races that lead to use-after-free violations (i.e. uses of a freed pointer) [43]. While this tool also does offline analysis of a single execution trace, their focus is on computing an explicit happens-before relation, which they apply to the trace in order to find accesses that may lead to user-after-free possibilities. They employ some heuristics to minimize the possibility of false positives.

Maiya *et al.* [61] *et al.* focus on a systematic exploration of possible schedules using a UI explorer and reasoning about the obtained traces using a race detector. A precise model of the Android execution life cycle is key to avoiding false positives, while the remaining cases are still high.

Race detection for JavaScript applications: Zheng *et al.* [107] propose a static technique to detect potential races in JavaScript applications. More recently, Petrov *et al.* [79] and Raychev *et al.* [86] have observed the potential for asynchrony creating out-of-order execution and developed a notion of race conditions for Web applications

written in JavaScript. In principle, race conditions can arise because of accesses to data shared among components of a Web page which are not ordered by proper synchronization, or, more formally, a happens-before relation. Of course, on a Web page, the entire DOM is (a giant blob of) global state, creating the potential for races.

Petrov *et al.* [79] define a happens-before relation for Web pages and generalize the notion of race conditions to take into account cases where, logically, there are unordered accesses to the same resource. The authors present a dynamic method for detecting races in a given execution of a Web page, explore similar executions that could potentially be racy, and, in later work [86] identify and filter out large sets of benign races. Hong *et al.* [42] propose a testing framework along with an execution model for asynchronous event handlers in JavaScript. Using defined execution model, the framework will execute a JavaScript application for collecting the executed asynchronous event handlers and then generates test cases for testing different orderings of these events. Later, the resulting DOM structure of each testcase is checked to the reference constructed by the initial execution.

Program analysis for JavaScript applications: A number of analyses have been propose for JavaScript in recent years; here we highlight only a handful. Additionally, several aspects of the language such as the use of `eval` [88, 46] and trying to understand JavaScript performance [85, 87].

Saxena *et al.* [93] first proposed a taint-guided fuzzing technique for finding client-side code injection attacks and later [92] devised a symbolic-execution based framework for client-side JavaScript code analysis. They also developed an automated tool for generating high-coverage test cases to systematically explore its execution space. They employ different techniques for covering both the events space and the value space of a JavaScript application. They propose a symbolic execution mechanism that will systematically explore the value space for each input field in an web pages. On the other hand, they devise a randomized exploration strategy for exploring the events space on GUI components, however, they do not cover any other events like server requests.

Rozzle [51] proposes the idea of lightweight multi-execution in the context of a JavaScript engine, similar to our work in this thesis. The goal of Rozzle is to expand the impact of malware detectors by increasing code coverage and thereby observing more, potentially malicious, code. In terms of techniques, Rozzle is probably the closest runtime exploration approach to the work described in this thesis.

Li *et al.* [56] devised a framework called SymJS for automatic testing client-side JavaScript applications using symbolic execution and automated event explorer. The authors present a symbolic execution engine for JavaScript by extending the Rhino [71] JavaScript engine that is used for automated testing. SymJS uses an automated event exploration for generating event driven test cases. Although this exploration incorporates user interactions, event exploration falls short on covering other asynchronous events like XHR callbacks.

Jalangi framework [96] promises its users; selective record-and-replay mechanism for JavaScript source code and useful abstractions for having shadow values associated with any value in the program. Using Jalangi users can define their own dynamic analysis like concolic testing [35, 97], checking for bad coding practices [37], detecting performance issues [36] or detecting crash paths [60].

Jensen *et al.* [45] proposes a stateless model checker, similar to Chess [72], for event-driven application for systematically exploring the non-determinism in the JavaScript application. The model checker is equipped with partial order reduction for reducing the search space and a replay mechanism for executing events in a specific order.

A project by Chugh *et al.* focuses on staged analysis of JavaScript and finding information flow violations in client-side code [18]. A valuable feature of that work is its support for dynamically loaded and generated JavaScript in the context of what is generally thought of as whole-program analysis.

The Gatekeeper project [39, 40] proposes a points-to analysis together with a range of queries for security and reliability as well as support for incremental code loading. Gulfstream [40] is a successor of the Gatekeeper project whose focus is on incremental

analysis and dynamic code loading.

Sridharan *et al.* [100] presents a technique for tracking correlations between dynamically computed property names in JavaScript programs. Their technique allows them to reason precisely about properties that are copied from one object to another as is often the case in libraries such as jQuery. Their technique only applies to libraries written in JavaScript, so stubs for the DOM and Windows APIs are still needed.

Schafer *et al.* study the issue of auto-completion for JavaScript and propose a solution to JavaScript auto-completion within IDEs using a combination of static and runtime techniques [94].

Madsen *et al.* [59] proposes the idea of a *use analysis* for the purposes of call graph construction in JavaScript applications that use large frameworks and libraries. Their use analysis is combined with a points-to analysis for the rest of the application.

2.2.2 Race Detection for Applications using Hybrid Programming Models

To our best knowledge, there is no prior work directly designed for race detection on hybrid programming models that combine dataflow constructs with shared memory programming model. In this section, we will give the state of the art on the race detection techniques for conventional multithreaded programming models and then focus on recent proposals for analysis of programs implemented using hybrid programming models.

Race detection for multithreaded programs: Dynamic race detection mechanisms typically monitor the synchronization operations and memory accesses to determine if there are racing accesses to a memory location. Generally, there are two types of dynamic race detection approaches: *lockset based* [91, 104, 106, 17] and *happens-before relation based* [29, 23, 63, 98, 16, 65].

Lockset based race detection makes use of *locksets* that stores the locks held by each thread at any given time in the execution. Typically, the race detection mechanism tries to check if all accesses to the shared memory follows a locking discipline. ERASER [91] is one of the well-known examples of *lockset based* race detection

algorithm that tries to enforce a locking discipline where each access to a shared variable is protected by some common lock. However, this locking discipline is overly strict which results in having many false positives. Although there are many proposals [104, 106, 17] to reduce the number of false positives, there are still cases that cannot be handled precisely. In general, *lockset based* race detection mechanisms are *sound* and *efficient* but not *precise*. In dynamic analysis context, *soundness* corresponds to not missing any races in observed executions. In other words, *sound* race detection mechanisms might give false positives but do not miss any real races in the execution.

Happens-before relation based race detection mechanisms use the happens-before relationship [53] for tracking the nondeterministic accesses to the shared memory during execution. Typically, a race is declared if two accesses to a shared memory, where at least one of the accesses is a “write” operation, are not ordered by a *happens-before* relation.

Eli *et al.* [81, 82] proposed DJIT⁺ algorithm that uses *vector-clocks* [64] for representing happens before relation where every thread and synchronization variable is assigned a vector of logical clocks. With this compact representation, detection mechanism constructs a partial ordering of the events in the execution and declare a race if there are conflicting accesses to a shared memory location.

Elmas *et al.* [23] proposed a sound, precise and efficient race detection mechanism, called GOLDLOCKS, which uses a lockset based algorithm for computing the happens-before relation between the accesses. GOLDLOCKS uses a static analysis for detecting thread-local accesses to eliminate extra detection overhead for these accesses. It also provides a Java runtime exceptions called `DataRaceException` for bringing the race to the attention of the programmer.

FASTTRACK [29] is a precise and efficient race detection mechanism build for multithreaded Java programs. The detection mechanism builds upon DJIT⁺ algorithm, by using the fact that use of general vector-clocks is not necessary most of the times. FASTTRACK uses a lightweight representation for the happens-before relation that

reduces both time and space overheads.

Program analysis in hybrid programming models:

StarSscheck [15] checks for inconsistent behaviors in task-based parallel application written in StarSs programming model [52]. In StarSs programming model, programmers have to annotate their sequential applications by defining task regions and their input-output dependencies in order to get performance advantageous of the underlying system. Any missed annotation on the input-output dependencies might affect the runtime causing failures in the executions. StarSscheck verifies whether the programmer correctly annotates the application, however it does not provide a race detection over these programs. Instead the tool only checks the correctness of input/output streams of tasks specified by the user. Moreover the algorithm assumes that the tasks execute over the copy of the shared data rather than actually accessing the shared memory. Hence the tool fails to detect races on tasks concurrently accessing common shared memory locations that are not specified through input streams by the user.

Feng *et al.* [26] propose an SPBag algorithm for detecting data races for multi-threaded applications developed in Cilk programming language [12] with spawn-sync computation. 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. Similar to other happens-before based race detection mechanisms, they define a *precede* and *in-parallel* relation for detecting races over possibly concurrent thread executions. Later, Raman *et al.* [84] extended their work by adding support for *async-finish* computation. However this type of asynchronous execution constructs is different than dataflow constructs as the execution is not triggered by input data availability.

Vechev *et al.* [103] propose a static analysis approach to verify determinism in task parallel programs. Their technique composed of two phases. First, they apply a sequential analysis on each task body and compute an over-approximation of accessed memory locations, under the assumption that all memory locations accessed

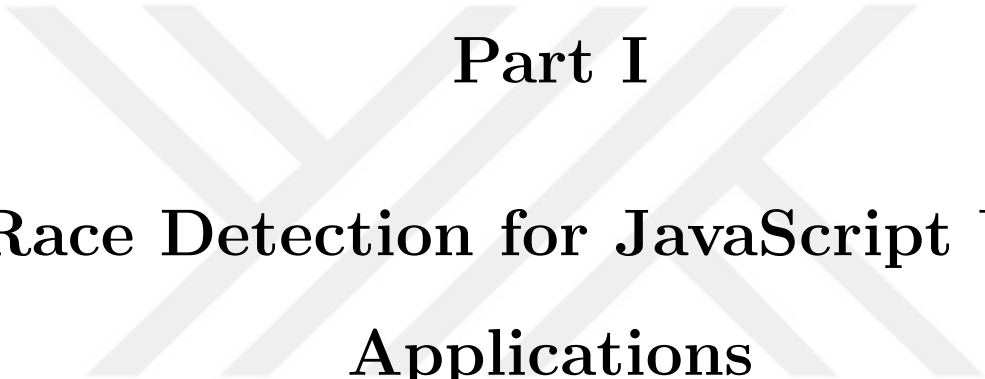
by parallel task bodies are independent. Later, their analysis tries to prove whether the independence assumption holds by checking each pair of statements that may execute in parallel and look for conflicting accesses. While such static analysis can prove the absence of non-determinism in task based programs, it fails to handle usage of synchronization mechanisms (i.e. locks, atomic blocks) within tasks bodies. Another shortcoming of this analysis is that it can be quite expensive to establish the independence of the statements with large tasks bodies that can be executed in parallel.

2.3 Summary

In this chapter, we discussed the trends in use of asynchronous programming models in modern computing platforms and the need for testing and verification effort in asynchronous programs. More specifically, we investigated the execution models of asynchronous JavaScript applications and hybrid applications that uses dataflow constructs with shared memory programming models. This background study on the execution models, gave us the required insight of the programming models which we later use for defining *happens before* relations within an execution of the applications. Later, we presented the state of the art on race detection mechanism as well as other analysis applied in context of asynchronous JavaScript applications and hybrid application using dataflow constructs in combination to shared memory synchronization mechanism.

In context of JavaScript applications, current state of proposed race detection techniques fails to describe accurate *happens before* relations which results in high number of false positives. While different filters for classifying *harmful* races is proposed for these detection techniques, the main focus remained to be races on the *transient state* which has an ephemeral effect in web applications. Different than legacy concurrent applications, JavaScript races on the program state is mostly invisible to the user portions of the applications. Hence, we argue that races on *persistent state* (i.e. cookies, localStorage) has more *harmful* effects in web applications.

On the other hand, hybrid programming models using dataflow constructs along with shared memory programming model have very limited tool support for detecting data races. While there are multiple proposals for race detection for pure shared memory concurrent applications and pure dataflow application, they can not be directly applied to this class of hybrid programs. In an effort to address this need, we proposed a race detection solution for different flavors of such hybrid programming models. Our techniques is based on a *modular happens before* definition which is constructed by using earlier definitions for pure shared memory and pure dataflow programming models. This way, we can re-use the previous research and also ease the development efforts for extending our race detection technique for new hybrid programming models.



Part I

Race Detection for JavaScript Web Applications

Chapter 3

INTRODUCTION

Today's client-side Web applications have been developed largely without much concern around multi-threaded execution. When it comes to client-side Web programming, today JavaScript powers the majority of large and popular Web applications. JavaScript execution is single-threaded. Yet the complex needs of sites such as Facebook, Outlook, Google Maps, and the like have lead to *asynchrony* becoming a common way to program complex Web applications. It is asynchronous processing that is responsible for interactive and responsive user interfaces (UIs) that operate without blocking the UI thread or requiring a reload, as did Web applications of the late 1990s. JavaScript's event-driven execution model matches the asynchronous call paradigm very well. Large JavaScript applications extensively use callbacks, function closures that are placed on the event loop for later processing, for handling asynchronous events (e.g. user actions, server events). Perhaps the most common use of asynchrony is JavaScript Web applications communicating with back-end servers using a `XMLHttpRequest` (XHR) object, as shown in Figure 3.1. Here, a network connection is open and a `onreadystatechange` handler is set up to process return data as it is received from the network.

Despite JavaScript lacking conventional threads, the presence of asynchrony creates a potential for races. In particular, the ordering of event executions in JavaScript as well as the timing of completions of asynchronous requests is non-deterministic, affected by, e.g., network delays resulting in data races. As initial JavaScript specification doesn't include any synchronization mechanisms, developers are forced to use error-prone ad-hoc solutions like callback chaining, implementing conditional waits on variables. With the approved ECMAScript6 specification [21], a new asynchronous

```
1  var req = new XMLHttpRequest();
2  req.open("GET", "login.php?name="+ username, true);
3  // set up the callback
4  req.onreadystatechange = function () {
5      if(req.readyState == 4) {
6          // do something
7      }
8  };
9  req.send(null);
```

Figure 3.1: Typical use of XHR.

construct called **Promise**'s became available for nesting asynchronous callbacks and integrating error handlers for these. While these constructs are good for ordering callback, they don't solve the issue of having data races as there might still be conflicting chains with one another.

In the first part of our study on races in JavaScript applications, Chapter 4, we focused on races that has an *observable* consequences on the clients side views. In an attempt to investigate real damage of JavaScript races in the wild, we devised randomized and systematic exploration mechanisms for practicing different server response orderings. Using these exploration techniques, we aimed to distinguish “well-behaved” orderings with no visual discrepancies from erroneous ones that results in *observable* races. We also proposed a crowd-sourcing scheme for covering different orderings for finding “well-behaved” network schedules which can be used to “shepherd” the orderings towards correct execution.

In the second part of our study, Chapter 5, we argue that one should distinguish between JavaScript races that have persistent consequences and those that are ephemeral. We find that the majority of races in JavaScript have no persistent consequences — we dub these races *benign*. This is because these races only result in invisible to the user portions of the program state, or, at worst, UI glitches that are either unnoticed by the user, or disappear if the user reloads the page. Given the forgiving nature of JavaScript execution, where failures of individual event handlers

force the scheduler to terminate the current event handler and move execution to the next one, these kinds of failures are not as important as previously believed. We suggest that a more useful way to think about data races on the web is by focusing on *persistent state*, such as client-local cookies, `localStorage` and `sessionStorage` mechanisms, as well as server-based side effects. The latter are achieved via `POST` calls to the server (`GET` calls are designed for reads and are supposed to be idempotent, and thus are not frequently used for state updates). Our experiments confirm that the number of such *harmful* races is quite modest, yet it is these kinds of races that are more likely to be considered serious and fixed by developers.

It is not our goal to provide a sound over-approximation of all possible races in a given application or site, but, instead to provide a *lightweight exploration* algorithm that allows for the exploration of multiple schedules while only requiring a single run. This approach can be used for testing, including collaborative testing by a large number of beta- or crowd-sourced testers. A number of companies nowadays provide commercial crowd-sourced testing services for web sites (<http://www.utest.com>), which can be used as a cheap and practical way to increase coverage.

Chapter 4

OBSERVABLE RACES IN JAVASCRIPT APPLICATIONS

In this chapter of the thesis, we explain our initial study on JavaScript race where we advocate for the notion of *observable races*, focusing on concurrency conditions that lead to visually apparent glitches caused by non-determinism within the runtime scheduler on the network.

We propose and investigate ways to find observable races via *systematically exploring* possible network schedules and *shepherding* the scheduler towards correct executions. We propose crowd-sourcing both to spot when different schedules lead to visually broken sites and also to determine under what environment conditions (OS, browser, network speed) these schedules may in fact happen in practice for some fraction of the users.

4.1 Overview

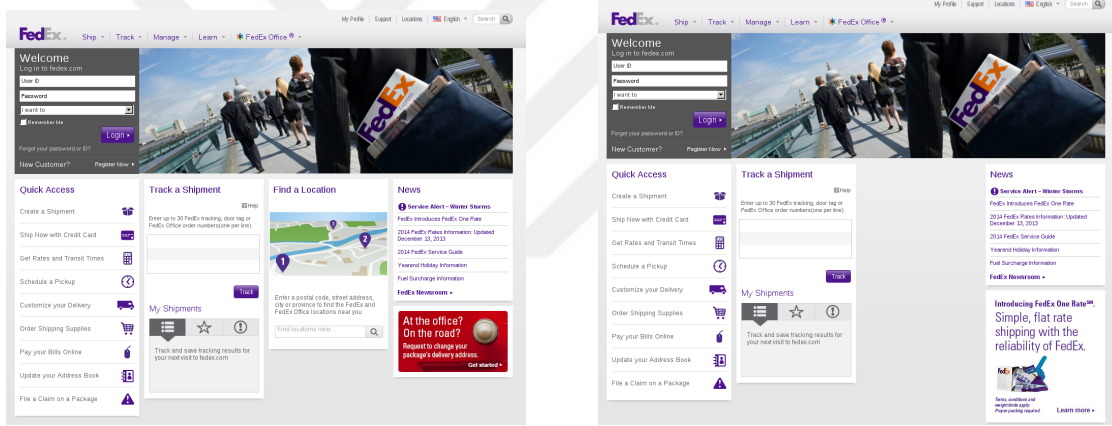
As is the case for shared-memory concurrent programs, race conditions in Web pages are low-level events that may be symptomatic of higher-level concurrency and design errors. However, race conditions are only a *proxy* for serious concurrency errors and can be inaccurate as a correctness criterion in two ways:

- **False positives:** as investigated in [79, 86] race conditions may be entirely *benign* and not lead to user-observable errors, data corruption or loss. Moreover, Web users generally have lower expectations of robustness than users of systems code or, say, compilers. The most common reaction to a misbehaving Web application may be to just reload the underlying page.

- **False negatives:** fixing a race condition may not fix the real concurrency error.

In an example reported in [107], a request issued by the user via a mouse click is applied *not* to the data currently displayed to the user, but to data that is in the process of being received. In this example, the real bug is an atomicity violation and a race condition is the symptom. In examples such as this one, it is possible to fix the race condition (the symptom) but not the actual bug.

We argue that so far, there has not been a satisfactory oracle proposed that distinguishes benign and harmful Web races proposed in the literature. In this chapter, we endeavor to develop such an oracle.



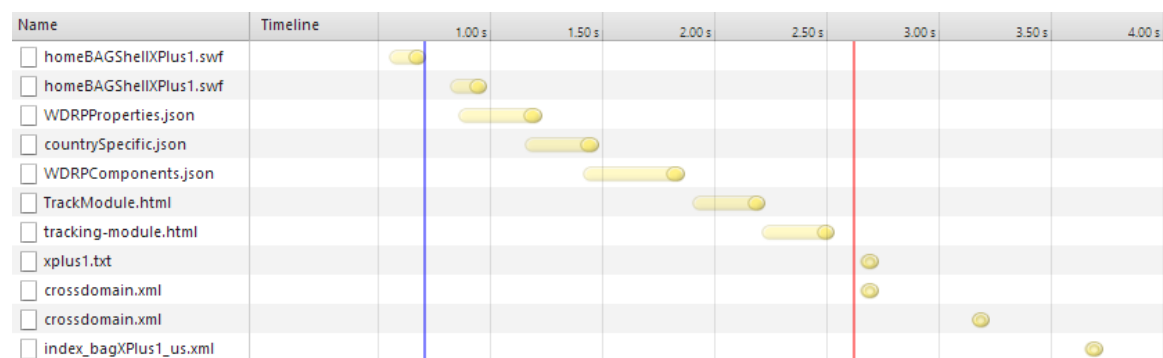
(a) Screenshot for one unmodified XHR call timing.

(b) Screenshot for an alternative, randomly-delayed XHR timing.

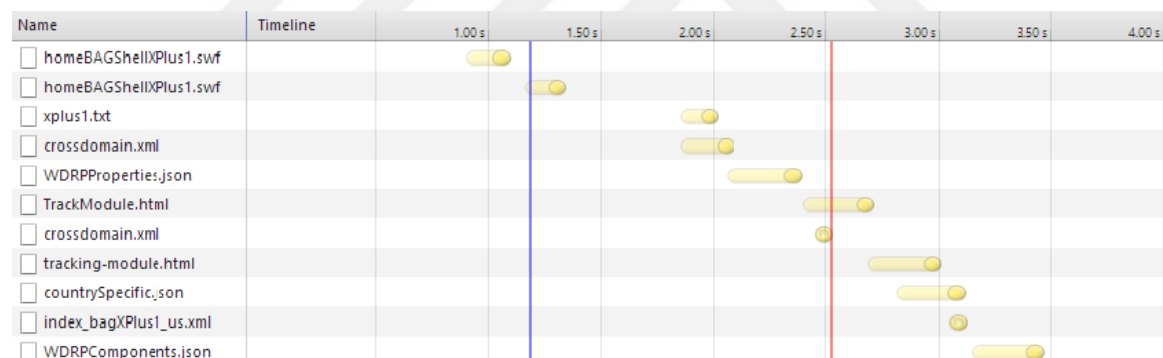
Figure 4.1: Screenshots obtained from two different XHR timings for an artificially-broken version of the FedEx Web site.

Observable races: Our approach is the mirror opposite of the prior works, instead of finding potential races and then filtering them to focus on the more damaging ones, we *start* with the category of races that is decidedly noticeable by the user. We wish to find races that are both provably damaging and fixable; it is not our goal to be exhaustive.

We call races that lead to buggy behaviors that are visible to the end-user *observable races*. A fundamental issue with bug detection tools, both static and runtime, is *explaining* analysis results in a way that is understandable by the developer, and ultimately leads to a fix. Focusing on observable races naturally allows us to create *repros* for our findings, i.e. visually obvious proofs of there being a race¹.



(a) Well-behaved timing.



(b) Defective timing resulting in observable race.

Figure 4.2: Original and modified XHR timings for the artificially-broken version of the FedEx site.

For an example of a behavior discrepancy that is visible to the end user and results in a semantically different output for the web page, consider the two screenshots shown in Figures 4.1a and 4.1b. These screenshots were obtained from two different XHR

¹Admittedly, some “deeper” properties such as “does this race lead to monetary loss” or “does this race lead open up a security vulnerability” cannot be addressed with our definition.

timings (shown in Figure 4.2a and 4.2b) on an artificially-broken version of the FedEx web site, in which we replaced some synchronous XHR calls with asynchronous ones in order to be able to illustrate the semantic visual differences (the "smoking gun") that we target with our approach.

4.1.1 Exposing Races with Systematic Exploration

In order to trigger different concurrency-related behaviors of a Web application, we effectively virtualize the network interface exposed to JavaScript applications via XMLHttpRequest (XHR). We provide a mechanism to wrap XHR calls and systematically explore all possible XHR call orders. Our technique uses a proxy based dynamic instrumentation tool [50] to employ source-level instrumentation for controlling XHR calls.

In our initial effort (Section 4.2), we investigated different orderings of XHR calls using a randomized exploration mechanism by applying randomized delays to each XHR call. We experimented with XHR-heavy Web sites chosen from the Alexa index and we explored a large number of XHR orderings in a brute-force manner. Surprisingly, even on sites in which this approach was able to explore all possible XHR interleavings, we did not detect any *observable races*. While this might possibly be explained by the fact that for Web sites with observable races, the brute-force approach did not exercise any of the observable race scenarios, we believe this explanation to be unlikely. We instead conjecture that observable Web races in the wild are extremely rare. In this work, we build a systematic XHR schedule exploration tool (Section 4.3) that, combined with the rest of our approach described below, will serve as a detection tool for observable races. We are going to use this tool to test our conjecture that observable races are extremely rare.

Exhaustive systematic exploration of XHR schedules concretely is not the only way to detect observable races; indeed, one could perform symbolic execution or multi-execution, both of which have been tried for JavaScript before [92, 51]. Once we have at least two different XHR timings on a given application for which there

is indication that the visual outputs are different, we can compare the outputs in a more semantic manner.

I know it when I see it: The most direct way to perform such a comparison is to capture *screenshots* that the different XHR call orders result in, in an effort to spot visual differences. There are several approaches to being able to spot visual differences due to races:

- the developer can be involved in the process, yet the number of schedules may prove so large that considering all before/after screenshots will be prohibitively costly;
- screenshot differencing may be done automatically [19]. While in many cases the before and after images are exactly the same, in some cases there are subtle differences. Moreover, these differences are often semantic: a slight difference in page layout such as a table border being moved by a millimeter may result in a lot of pixels being different between the two images;
- the task of screenshot differencing can be given to humans who are, we argue, better suited to the task of determining which differences are semantic and which has minor effects in appearance.

Our current implementation uses a combination of the last two approaches to build a semantic oracle: if the before/after images are not the same, we ask crowd workers to decide if they are indeed semantically different and, if so, how. Note that the issue of network non-determinism needs to be addressed: in general, we need to replay the same network responses if we hope to get the same visual result. This is because the majority of popular Web pages use ads, which often change on every page load and need to be memorized and replayed back.

4.1.2 *Smoking Gun*

The *repros* described above usually take the form of a before/after screenshot pair, the before looking as expected by the user, the after being broken in some way. They

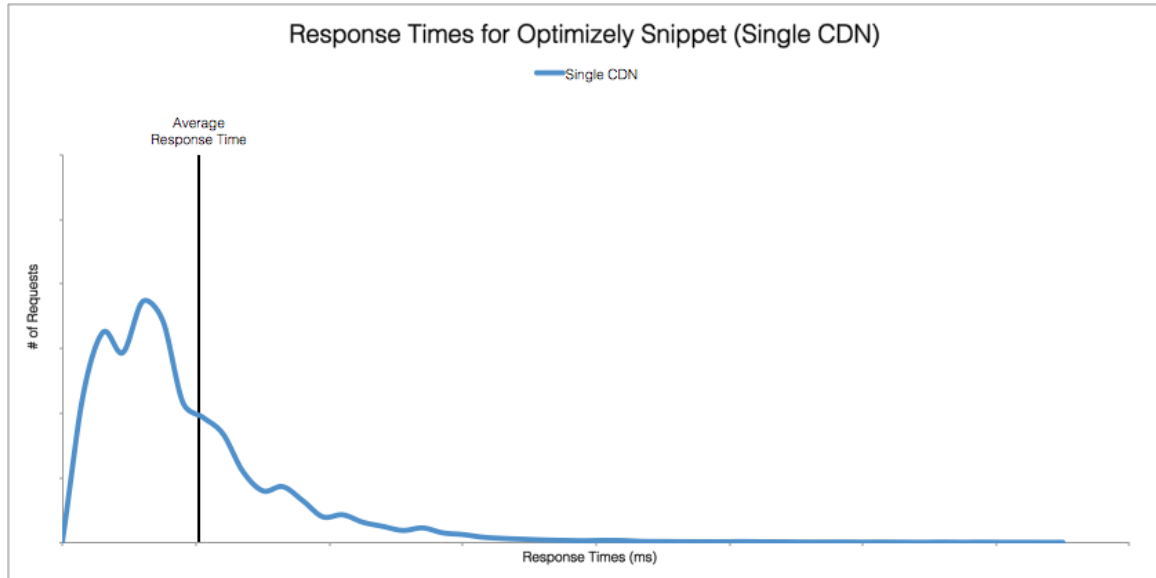


Figure 4.3: Overall distribution of response times for the Optimizely snippet world-wide, with the average response time.

play a tremendous role in creating convincing and actionable error reports.

It is important to recognize that there is a fair bit of inertia when it comes to trusting the results of analysis tools, especially static analysis tools that reason about “the hypothetical.” This is due to an informal belief that the bug will not occur in practice. But more fundamentally, even in safety-critical code, developers are often fearful of fixing races, unless they can be clearly convinced of the damage, for fear of introducing new errors, unmasking other errors, introducing performance or security issues, etc.

It is our position that showing screenshots with visual defects to the developer would serve as a more convincing argument than presenting hypothetical races found via either static analysis or runtime exploration. It is especially important to note that developers might be reluctant to consider results they consider hypothetical for applications that have already been deployed and tested in the field. To summarize, our ultimate goal is to present the developer with a combination of the following as an error report:

- Scenario description (workload and at least two different XHR schedules);
- implicit constraints being violated, such as a happens-before relation that does not hold for the offending execution;
- before/after screenshots;
- environment/network conditions that lead to the screenshots above.

4.1.3 Prioritizing Races Using a Crowd

Finding an error “in the wild” as opposed to finding one in an execution that has never been observed in real-world conditions will lead to bug reports of significantly higher intrinsic value. We can use a crowd of users not only for distinguishing between “good” and “bad” screenshots of a Web application shown in browser. We can, in fact, use a crowd for *confirming* a particular race as observed in an actual execution. A key element of this strategy is to record (and profile) crowd workers based upon their environment characteristics such as the OS they use, the browser they run, the device from which they access the Web, as well as network characteristics (3G, 4G, WiFi, T3, T1).

One way of using the crowd for exploring the landscape of interleavings possible in the wild is to

- wrap XHR calls with instrumentation and logging code, and have the crowd members log the order of callbacks they experience,
- ask users or have a runtime monitor whether a harmful outcome was observed, and
- store both buggy and well-behaved interleavings, along with the user’s configuration information.

This can be seen as crowd-sourcing the exploration of different schedules induced not by systematic exploration but by different configurations, in addition to crowd-

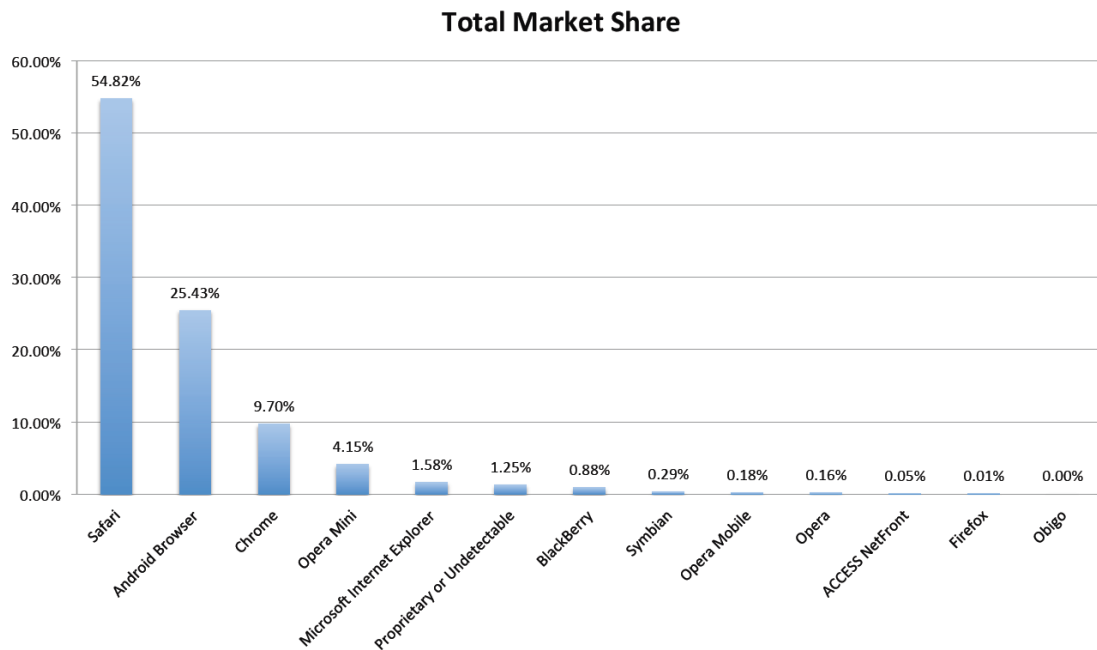


Figure 4.4: Total mobile browser distribution.

sourcing the determination of whether a Web page output is semantically broken or not.

The challenge of this strategy will be in exploring the long tail of the population when it comes to these characteristics. In particular, it is easy to explore the common case such as Internet Explorer 10, 32-bit on Windows 7 or Chrome 22 on MacOS X, but finding someone who uses a particularly uncommon version of a rare browser on a slow connection is likely to be both more challenging and more fruitful, as these configurations are unlikely to have surfaced during testing. Being able to selectively solicit users with uncommon environment combination is key in making this approach work well.

If we also have statistics about what percentage of users are in every segment (OS, browser, device, network speed), we can also extrapolate to predict how common every race may be in practice based on the data we sample via the crowd. Developers would find this information useful for prioritizing bugs.

4.1.4 Schedule Shepherd

For Web applications in wide use, it is expected that in “most common cases” the Web page must not be resulting in serious concurrency errors. We build upon this intuition and record “well-behaved” schedules/interleavings for a Web application, i.e., interleavings that do not lead to outcomes identified as errors during crowdsourcing. We propose the use of *schedule shepherding* to avoid observable races.

Based on our brute-force randomized exploration experiments, we conjecture that *observable races* are rare events. Yet, they may occur for widely-deployed applications because of the inherent *diversity* of devices, browsers, operating systems, and network conditions. While races in deployed applications are unlikely to appear for the most common configurations, we observe a long-tail phenomenon in terms of response latency (Figure 4.3) and also the browser being used (Figure 4.4). As a result, uncommon configurations may not be covered in testing, leading to the potential for rare, but damaging races in deployed code. Moreover, even if races happen in the field, perhaps, for uncommon configurations, users are likely to chalk whatever they see up to “random glitches” so common in today’s Web and not report them to the developer. It is our goal to shepherd the uncommon but buggy interleavings towards well-behaved ones.

By orchestrating the order of execution of wrapped XHR calls (by delaying the execution of a callback, if necessary) we can *guide* or shepherd the execution order of XHR responses/callbacks toward known, well-behaved schedules. While such an approach would not be viable for extremely critical systems software, it provides a lightweight fix for concurrency errors that might otherwise remain completely undressed.

4.1.5 Key ideas

To summarize, the key ideas put forth in this chapter are:

Observable races. The notion of observable races, as determined by visual inspection of the Web application.

Systematical exploration for XHR call orders. A mechanism for systematically exploring all possible orderings of asynchronous calls in order to exercise potentially defective schedules.

Crowd-sourcing for finding concurrency bugs. A crowd-sourcing approach for determining whether a concurrency error has resulted in observable non-determinism.

Smoking gun detection We propose ways to find observable data races “in the wild”, as opposed to in hypothetical, synthetic executions. This approach involves using a crowd of users in a different capacity than before.

Schedule shepherding. Shepherding of Web-page non-determinism in order to avoid outcomes determined to be harmful via crowd-sourcing.

4.2 Randomized Exploration

For our randomized exploration, we focused on Web sites from different contexts. We analyzed some Web sites that have been reported to contain data races that are classified as harmful in prior work [79, 86]. We also investigated Web sites that make extensive use of XHR calls (Figure 4.1) that we obtained by analyzing the Alexa top 4,000 sites. With the help of scriptable WebKit mechanisms [41, 47], we were able to inject our wrapped XHR call methods for applying our randomized delay strategy to Web sites under test and also create Web site screenshots at the end of the page load event.

We first provide key features of our approach for determining observable races. In order to identify widely used, XHR-call-heavy Web sites, we examined the Alexa top 4,000 site list. Out of 4,000 Web sites, we selected 900 Web sites containing at least one XHR call. We believe we would have ended up with a higher count if we had explored sites that require authentication such as Outlook, Facebook, etc. We used an automated network traffic recording mechanism that collects HTTP Archive (HAR) file for each Web site in the Alexa top 4,000 sites. After analyzing the collected HAR

files, we were able to count the number of XHR calls made for each Web site.

Figure 4.1 presents data about sites we investigated that are heavily-dependent on XHR calls. The table provides information about the number of times XHR was called at runtime as a result of visiting the page. Additionally, the table shows statistics about the mime types of the responses obtained via XHR and also the types of the XHR calls made (XHR calls can also be made synchronously). This allows us to distinguish between JSON, JavaScript code, and plain text being returned. In order to be able to trigger different response time orderings, we wrapped XHR calls with randomized delays. To record the visual consequences of different XHR orderings, we captured the screenshot of the loaded Web site with and without randomized delays. We automated this procedure for easy repeatability. Our automated procedure works as follows;

1. We capture network traffic records and two screenshots of a Web site by loading it twice before applying any delay mechanisms
2. We capture the network traffic record and the screenshot of the Website after our randomized delays applied.
3. We eliminate cases in which the unmodified and randomly-delayed versions of the page produce the same screenshot by using a masking mechanism [19].
4. We then crowd-source the determination of whether the randomly-delayed version of the Web page is broken.

Although, in theory, there are $n!$ possible XHR call orders for n asynchronous XHR calls, in practice this number can be lower as Web application developers use *chained* XHR calls where one callback function initiates another XHR call. Chaining creates constraints that restrict the number of possible schedules. An example of chaining is shown in Figure 4.5.

For our randomized exploration experiment, we selected a couple of Web sites with an average number of XHR calls. A representative example is `http://www.news24.`

```
1 req.onreadystatechange = function () {  
2     if(req.readyState == 4) {  
3         // first handler  
4         var req2 = new XMLHttpRequest();  
5         req2.onreadystatechange = function () {  
6             // second handler  
7             ...  
8         }  
9     }  
10 };
```

Figure 4.5: XHR chaining.

com with 7 asynchronous calls which should have at most $7! = 5,040$ possible orders. We then applied randomized exploration mechanism without any prior analysis for *chained* XHR calls. Even though we employed our randomized mechanism in a brute-force manner on these sites, we were only able to investigate around $1,600/5,040$ distinct XHR call orders without observing any visual defects on the generated screenshots.

Thus, we conjecture that *observable races* are rare events and to uncover such events systematical exploration with the knowledge of *chained* XHR calls is needed.

4.3 Systematic Exploration

For our systematic exploration, we carried out a source-level instrumentation of XHR calls using a proxy-based dynamic instrumentation tool for JavaScript, *AjaxScope* [50]. This approach uses a client-side proxy positioned between the Web server the browser to dynamically capture and rewrite JavaScript code. We wrapped XHR calls with extra recording mechanisms for capturing the XHR scheduling order of the Web site.

Our exploration starts by recording an initial order of the XHR calls observed during the loading of the page using *AjaxScope* instrumentation. We also devised a logging mechanism for capturing information about *chained* XHR calls. When systematically exploring XHR schedules, these chaining dependencies are taken into

Web Site	Mime Types			Classifying XHR		
	JSON	JS	Text	Sync	Async	Total
*.mlb.com	9	2	24	10	25	35
discussions.apple.com	26	0	0	0	26	26
www.aljazeera.net	17	1	5	0	23	23
www.gazzetta.it	3	0	19	10	25	22
wireless.att.com	3	0	15	2	16	18
www.welt.de	0	0	18	0	18	18
www.tvguide.com	1	5	11	0	17	17
www.optimum.net	3	1	12	0	16	16
www.fujitv.co.jp	7	8	1	3	13	16
www.bild.de	2	0	12	0	14	14
www.nasa.gov	12	0	2	0	14	14
news.qq.com	0	0	12	0	12	12
www.zaobao.com	0	0	12	0	12	12
www.girlsgogames.com	9	0	3	4	8	12
www.sports.ru	3	0	8	0	11	11
www.premierleague.com	2	0	9	2	9	11
www.eltiempo.com	0	0	10	0	10	10
www.myvideo.de	9	0	0	0	9	9
www.politico.com	0	1	8	0	9	9
www.att.com	3	0	6	2	7	9

Table 4.1: Top 20 XHR-heavy Web sites in our experiments.

account so our exploration tool never attempts to exercise two chained XHR calls in the wrong order.

Starting from the recorded initial XHR order and *chained* XHR call information, our exploration will generate all possible XHR schedule permutations to be tested. For each generated schedule, our tool enforces the corresponding schedule and takes a screenshot of the resulting Web page on the browser. In order to load a Web page with a desired XHR call schedule, we instrumented each XHR callback function to be

executed as according to the provided order. Given a schedule, our instrumentation controls the execution of each triggered callback by cross-checking with the schedule. Any callback that is triggered earlier, according to the desired schedule, will be delayed until its own turn. We give each XHR a unique ID. Consider a given desired order of execution of XHR callbacks, expressed as a permutation of XHR IDs. When responses from the network arrive, the corresponding callbacks are not immediately executed. Instead, the execution of the callback for XHR with ID i is delayed until i is the next ID in the given permutation.

Below is the step-by-step explanation of our mechanism:

1. Capture initial order of XHR calls and the information about *chained* XHR calls using instrumentation provided by AjaxScope.
2. Depending on the initial order and information about *chained* XHR calls, generate all possible XHR schedules for exploration.
3. For each generated schedule, enforce that Web page is loaded with XHRs following this schedule, delaying them as needed.
4. Collect screenshots for each enforced schedule to be checked for *observable races*.

4.4 Discussion

Compared to other runtime environments, concurrency in the browser has not been exhaustively specified or formalized. When it comes to the issue of *scheduling*, non-determinism can be caused by both the built-in scheduler and environment conditions such as the order of network message arrival. While researchers have explored the issue of runtime races, their findings have not proven actionable for the Web developer for two reasons. First, because of a high false positive rate, the developer does not know which races to fix. Second, due to the absence of explicit concurrency primitives for programming on the Web platform, the developer does not know how to fix them. While it is tempting to ask for tools and solutions, we claim that the sources and

extent of the problem are far from being well understood at this point. The research on web applications has to be extended to answer following questions.

Schedule diversity landscape. What kind of schedules are common in complex Web applications? Is it the case that most or all widely-used browsers effectively follow the same schedule or are there significant differences in terms of orders that result from Chrome vs. Firefox? What is the density distribution like? Is it the case that there is a *long tail* in terms of observed schedules?

Smoking gun. Do obscure, synthetically-generated schedules actually correspond to schedules that may be observed *in the wild*? Can we actually detect these uncommon races in the wild, recoding the conditions under which they are possible?

Observability vs. damage. Can we correlate races to *damage* to the user, application state, etc. It is not obvious what the “worst thing that can happen” is. Is it just a broken page or can observable races lead to logical bugs resulting in, say, financial losses or, possibly, enable security vulnerabilities?

Better runtime monitors. While our focus has been on *visually* observable races, we can also explore runtime monitoring as an oracle for finding damaging discrepancies due to scheduling differences.

4.5 Summary

In this chapter, we conducted empirical studies focused on finding the real damages in JavaScript races. Instead of focusing to all races in an JavaScript web applications like the previous proposals, we argue that the real focus should be on the races that has real consequences on the client’s browsing experience. Hence we focused on races that has an *observable* consequences on the clients side views. In an attempt to investigate real damage of JavaScript races in the wild, we devised randomized and systematic exploration mechanisms for practicing different server response orderings.

Using these exploration techniques, we aimed to distinguish “well-behaved” orderings with no visual discrepancies from erroneous ones that results in *observable* races. We argue that this kind of classification jobs can easily be crowd-sourced for covering different orderings for finding “well-behaved” network schedules which than can be used to “shepherd” the orderings towards correct execution.

We also conducted detailed asynchronous construct usage analysis on Alexa top 4,000 site list and than investigated for *observable races* using our exploration techniques. Our experiments showed that even in web sites that makes heavy usage of asynchronous constructs, the number of *observable races* are very low. In the light of this study, we turned our race detection effort to look for races on *persistent state* (i.e. `cookies`, `localStorage`) instead of the ones on *transient state* (i.e. program state) since their effects are ephemeral in context of web applications.

Chapter 5

DETECTING JAVASCRIPT RACES THAT MATTER

In this chapter of the thesis, we argue for a different formulation of what it means to have a data race in a JavaScript application and distinguish between benign and harmful races, affecting *persistent* browser or server state. We further argue that while benign races — the subject of the majority of prior work — do exist, harmful races are exceedingly rare in practice (19 harmful vs. 621 benign). Our results shed a new light on the issues of data race prevalence and importance.

To find races, we also propose a novel lightweight runtime symbolic exploration algorithm for finding races in traces of runtime execution. Our algorithm eschews schedule exploration in favor of smaller runtime overheads and thus can be used by beta testers or in crowd-sourced testing. In our experiments on 26 sites, we demonstrate that benign races are considerably more common than harmful ones. We also conducted an exploratory study on the reported race related bugs on JavaScript libraries that uses new asynchronous constructs: **Promise**. This study showed that it is still common to have race conditions even with the better abstractions for asynchronous programming in JavaScript.

5.1 Overview

There is increased interest in data races in asynchronous programs and, in particular, JavaScript. Static and runtime methods have been explored. A fundamental challenge with static analysis for JavaScript is that it is quite difficult to even *enumerate* all the relevant code, as much of the JavaScript code is produced with the help of `eval` calls which is used for dynamically generating code at runtime. As a result, the traditional

advantage of static analysis, namely, full path coverage largely does not apply to this problem. As such, the ability to make sound statements about the lack of data races is compromised [40, 46, 88, 107]. Additionally, reasoning about runtime events such as multiple `XmlHttpRequest`s (XHRs), whose callbacks race against each other statically is fundamentally very difficult, because the happens-before relation is quite complex.

Runtime techniques in this space are also vulnerable to losing precision, which leads tool authors to develop heuristics to eliminate potential false positives [43, 61, 86]. They also suffer from the lack of *coverage* and the inability of making sound guarantees about the lack of races. Additionally, runtime techniques that involve combinatorial *schedule exploration* can run into scalability challenges, especially when the number of possible handlers to schedule is high [61, 86].

Our technique attempts to combine the advantages of static and runtime analysis. We execute the code only *once*, yet we explore *multiple* execution orders. As such, our technique scales well, while increasing the coverage of a single-pass runtime analysis. One way to see our approach is that it explores *neighboring* schedules for a particular runtime execution by analyzing possible re-orderings of the asynchronous event handlers. We foresee this approach as being especially useful in the context of beta- or crowd-sourced testing: having a large number of users will naturally increase code coverage. At the same time, the users' sessions will not be significantly slowed down. It should be noted that in the browser, slowing down the browser runtime runs the risk of modifying the behavior of timeout set with `setTimeout` and `setInterval`; additionally, the runtime may actively attempt to terminate slow-running events.

5.1.1 What is a Data Race?

Several possibly definitions of data races have been proposed for web applications [73, 79, 86]. They all center around the idea of writes to shared state that are performed by callbacks. Some of the races in prior work are caused by user interactions and browser-induced timing.

In this work, our chief focus is on the `XmlHttpRequest` (XHR) mechanism, which allows client-side code to request data from servers:

```
1  var xhr = new XMLHttpRequest();
2  xhr.open("GET", "http://www.data.com/mydata.json");
3  xhr.onreadystatechange = function(e, d){
4  ...
5  };
6  xhr.send(null);
```

The code above is for a typical `GET` request that obtains JSON data from a server and schedules an asynchronous `onreadystatechange` callback to process the data once it arrives. Multiple such callbacks can be outstanding, creating the possibility of what we dub an XHR-XHR race, if these callbacks write to shared state. Moreover, commonly, while synchronous XHR execution is possible, XHRs are scheduled to be dispatched asynchronously, to maintain a responsive client-side UI [73]. In the rest of the chapter, we shall focus on asynchronous XHRs.

Secondly, a single XHR callback can race with the browser, resulting in the `state` variable being set to either 1 or 2. This is because the browser may have multiple script blocks, some of which may be scheduled either before or after the callback, depending on the callback's arrival and how fast the browser is rendering content:

```
1  <script>
2  var xhr = new XMLHttpRequest();
3  xhr.open("GET", "http://www.data.com/mydata.json");
4  xhr.onreadystatechange = function(e, d){
5      state = 1;
6  };
7  xhr.send(null);
8  </script>
9  ...a lot of text and images here...
10 <script>
11     state = 2;
12 </script>
```

Thirdly, and even more subtly, if the user opens the same site in multiple browser *tabs*, it is possible for these tabs to lead to concurrent execution. Two instance of

the code below may race with each other when run in different tabs, resulting in a cookie-based race on line 5:

```
1  <script>
2  var xhr = new XMLHttpRequest();
3  xhr.open("GET", "http://www.data.com/mydata.json");
4  xhr.onreadystatechange = function(e, d){
5      document.cookie = "value=" + Math.Random();
6  };
7  xhr.send(null);
8  </script>
```

The happens-before relation for asynchronous callbacks is defined by the creation order. The XHR callback is preceded by the code that creates the XHR (`xhr.send`). A specific case of this is what we call nested (or chained) XHRs, when callbacks are defined one within another. Practically, this is about the only way for the developer to ensure that there is ordering of XHR callbacks, so we see this programming pattern quite a bit.

```
1  <script>
2  var state = 0;
3  var promise1 = new Promise(doAsyncWork);
4  var promise2 = new Promise(doMoreAsyncWork);
5  promise1.then(function (){
6      state = 1;
7  }).then(function (){
8      state = 3;
9  });
10 promise2.then(function (){
11     state = 2;
12 });
13 </script>
```

Lately, with the new JavaScript language specifications [21, 22] developers are provided with new asynchronous constructs such as `Promise`s. With these constructs one can chain asynchronous event handlers via use of chaining methods such as `then`. Above example shows usage of these constructs via chaining different asynchronous handlers. However even with the new asynchronous callback ordering constructs like

Promises, the chained callbacks can race with each other. While these constructs support a more intuitive way of ordering callback like the code above, chains can still be racing with other asynchronous callback chains. While the updates on shared **state** variable on line 6 and line 8 are ordered by the use of **then** construct, there is still a possible race between updates on line 12 as they are not ordered.

5.1.2 Motivating Examples

In an effort to understand the possible impact of data races on the web, we spent some time analyzing bug reports for open-source projects located on GitHub. Below we describe some of the examples of subtle server-side bugs from GitHub. In the interest of fairness we should mention that these examples of races reported as GitHub issues were not particularly common bugs for JavaScript projects, an intuition that is largely confirmed by our results in Section 5.3.

Example 1 [Old server state.] Issue #79 for the WheatonWHALE project¹ describes the following situation:

1. The user reloads the page;
2. `onbeforeunload` listener fires, and the data is saved to server;
3. the page is loaded up again, and asks the server for the data;
4. the client-side JavaScript code loads up the old (outdated) data;
5. client state is saved to the server.

In the last step, the old, outdated data is saved to the server, essentially ignoring data updates. The culprit is the fact that steps 1 and 3 can race with each other: the data load request may arrive before the save is processed. The implemented fix makes data updates synchronous. $\square$

¹<https://github.com/WheatonWHALE/whaleweb/issues/79>

Example 2 [Racing for a user ID.] A somewhat similar situation that has to do with the issue of stale data obtained from the server is captured in issue #20 in a project called LikeLines². LikeLines provides users with an in-browser video player with a navigable heat map of interesting regions for the videos they are watching. This case describes two racing XHR calls that are issued to the backend server during initialization by the following functions: 1) `createSession` and 2) `aggregate`. The first call is to create a new session for recording user interactions. The second call is needed for drawing a heat map.

The problem arises when a user has not contacted the backend server before. In this case, both XHR calls will be issued *without* a cookie, and in both cases the server will create a *new* user ID. This is clearly a problem because interaction sessions are tied to a user ID in this applications. If the cookie from the call to `aggregate` “wins” (i.e., arrives last), then subsequent calls to the server will contain a user ID that does not match the interactions session. □

Example 3 [Racing `Promise` chains for downloaded file.] Another case of a race effecting the persistent storage is reported one of the popular package management application Bower³. Bower delivers a package manager where users can automatically download packages used by their applications and widely used for web development.

The reported issue rooted from a race condition that happens when multiple Bower instances try to install the common dependent packages. The erroneous execution is detected with the following steps:

1. Multiple Bower instances downloads the dependent packages to temporary folder;
2. Faster instance completes the downloading and moves the dependent packages to cache;

²<https://github.com/ShinNoNoir/likelines-player/issues/20>.

³<https://github.com/bower/bower/issues/933>.

3. Slow instance completes the downloading and tries to move the dependent packages to cache;
4. Bower throws `ENOTEMPTY` error as slow instance would try to move the common package to cache where it already exists.

□

In addition to the GitHub issues discussed above, below we list an illustrative example inspired by some of the samples from prior work [86, 107], although prior work did not focus on the issue of asynchronous XHRs.

Example 4 [Racing with the browser.] Consider the code in Figure 5.1. For convenience, we mark every handler above with a number. The happens-before relation induced by this code example is as follows: $\textcircled{1} \leftarrow \textcircled{2}$, $\textcircled{1} \leftarrow \textcircled{3}$, $\textcircled{3} \leftarrow \textcircled{4}$, $\textcircled{4} \leftarrow \textcircled{5}$. As such, our exploration algorithm will consider the possibility $\textcircled{1} \leftarrow \textcircled{2} \leftarrow \textcircled{3}$ and $\textcircled{1} \leftarrow \textcircled{3} \leftarrow \textcircled{2}$. Similarly, because $\textcircled{2}$ and $\textcircled{5}$ are weakly ordered, traces in which $\textcircled{2}$ happens before or after $\textcircled{5}$ will be considered.

While one can explore these traces via a search in the schedule space, we choose to do so via data flow. We keep track of the event handlers that may be “concurrent” and mark the writes that they make to the same locations as weak writes. Because $\textcircled{2}$ and $\textcircled{3}$ can race, the value of `document.cookie` will be either `var1=1` or `var1=2`. Similarly, for 2 and 5, the value of `document.cookie` will be either `var1=1` or `var1=5`. To preserve precision, our algorithm maintains existing happens-before relations such as those between $\textcircled{1}$ and $\textcircled{2}$ and $\textcircled{1}$ and $\textcircled{4}$. □

5.1.3 Trace Processing

Figure 5.2 shows a simple trace obtained by running the code in Figure 5.1 that illustrates cookie-based races. We start processing the lines 1–4 where the XHR is opened and send to the remote server, we mark the XHR callback as an active callback which can be executed asynchronously any time in the future. As we process the first XHR callback on lines 6–10, we will record the write value made to cookie variable

```
1  <html>
2  <script>                                     // 1
3      var xhr = new Xhr();
4      xhr.open('', false);
5      xhr.onreadystatechange = function(){      // 2
6          document.cookie = 'var1=1';
7      };
8      xhr.send();
9      //for(i=0;i<10000000; i++) console.trace(i);
10 </script>
11 <!-- <input id='mydiv' /> -->
12 <script>                                     // 3
13     document.cookie = 'var1=2';
14 </script>
15 ...
16 <script>                                     // 4
17     var xhr2 = new Xhr();
18     xhr2.open('', false);
19     xhr2.onreadystatechange = function(){      // 5
20         document.cookie = 'var1=3';
21     };
22     xhr2.send();
23 </script>
24 </html>
```

Figure 5.1: Multiple XHR race example.

into the memory map where values for each variable id (i.e. 242159440).

While recording the written value, we will look for values of the variable that are written by any callbacks that may be racing with each other, and complain about a race if there is any. As we continue processing the trace, we will record the value written on lines 12-14 by first checking the earlier values of `document.cookie`. As this sequential code segment can race with the earlier XHR callback (there is no happens-before edge), our processing will record a race on `document.cookie`, while adding a new value for the cookie into the memory map. As we process the trace, a new XHR is opened and send to the server on lines 16–19 and added to the active callbacks list.

```

1  READ PROP ID[99044816] = "open" : JSFunction
2  XHR [0000001] Open GET
3  READ PROP ID[98817648] = "send" : JSFunction
4  XHR [0000001] Send
5  XHR [0000001] Callback
6  BEGIN XHR_Callback [0000001]
7      READ ID[235205312] = document : JSObject
8      WRITE PROP ID[242159440] = cookie : JSInteger      (1)
9      Cookie [1020D800] Write "var1=1"
10     ...
11     READ ID[235205312] = document : JSObject
12     WRITE PROP ID[242159440] = cookie : JSInteger      (1)
13     Cookie [1020D800] Write "var1=5"
14 END XHR_Callback [0000001]
15
16 READ ID[235205312] = document : JSObject
17 WRITE PROP ID[242159440] = cookie : JSInteger      (2)
18 Cookie [1020D800] Write "var1=2"
19
20 READ PROP ID[99044816] = "open" : JSFunction
21 XHR [0000002] Open GET
22 READ PROP ID[98817648] = "send" : JSFunction
23 XHR [0000002] Send
24 XHR [0000002] Callback
25 BEGIN XHR_Callback [0000002]
26     READ PROP ID[108330176] = "readyState": JSInteger  (4)
27     READ ID[235205312] = "document" : JSObject
28     WRITE PROP ID[242159440] = "cookie" : JSInteger    (3)
29     Cookie [1020D800] Write "var1=3"
30 END XHR_Callback [0000002]

```

Figure 5.2: Sample trace illustrating cookie races.

Later, the callback for the second XHR will be processed, when a write to the cookie is performed on lines 21–26. While processing the write operation, the values for `document.cookie` will be checked and a race will be recorded, as two XHR callbacks are marked as racing, resulting in different cookie values.

5.1.4 Algorithm Summary

Here we provide the underlying intuition for our approach, with a more formal treatment relegated to Section 5.2. The key idea behind our approach is to consider alternative scenarios within a given trace. We do not attempt to force exploration of UI interactions, for example, however, we do explore the possibilities of different schedules that may occur because of the order of arrival of asynchronous event handlers that are part of the trace.

Our approach effectively performs static analysis on a trace that is collected at run-time, as a way to consider different schedules. When considering multiple execution of XHR callbacks as shown in Figure 5.3, the key observation is that instead of *separately* considering each of the possible schedules, we can *encode* the effect of the possible race by merging the state and keeping track of multiple, merged values. This is analogous to doing meets in static dataflow or abstract interpretation-style analysis, as an alternative to a costly meet-over-all-paths (MOP)

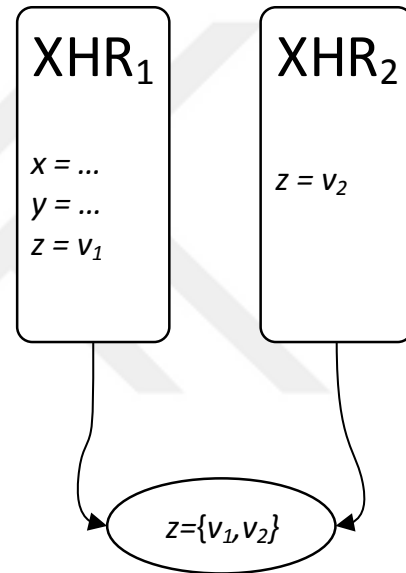


Figure 5.3: Merging two states after an XHR.

for variable z with multiple values coming in for two racing XHRs, $z = v_1$ and $z = v_2$, we keep track of both values $\{v_1, v_2\}$; not of course that if $v_1 = v_2$, no need to keep two copies of the same values exists.

We formulate our race detection algorithm as a *dataflow analysis* on the values within a given execution trace. We flag a possible race if multiple values may flow to a sensitive location, indicating a presence of scheduling dependencies; these sensitive locations are persistent storage such as `document.cookie`, `localStorage`, `sessionStorage`, and the DOM, etc. These latter locations serve as the *sinks* of our

data flow analysis.

Returning to the example in Figure 5.1, we can represent the race between handlers ② and ⑤ as an assignment `document.cookie = {'var1=2', 'var1=3'}` as a merge node after handler ⑤. This of course represents a direct flow of multiple values to a sensitive persistent storage location `document.cookie`. More interesting cases involve multiple steps of propagation.

5.1.5 Implementation

To collect execution traces, we have instrumented the most recent version of the Firefox web browser. Our changes span both the SpiderMonkey JavaScript engine to track data propagation through the memory of the browser as well as operations on `document.cookie`, `localStorage`, and the like, which are recorded by instrumenting the DOM. Our instrumentation spans over three main components of Firefox:

- **XPCOM (Cross Platform Component Object Model):** In order to record the triggered XHR callbacks, we instrumented the event queue in Firefox in `nsThread.cpp`. When events are taken from the queue for execution, we mark XHR `readystatechange` events as well as button clicks initiated by the user.
- **Gecko (Layout engine):** We also need to instrument DOM API implementation of Firefox for recording updates made to DOM elements of interest. We achieved this by modifying various DOM class implementations like `nsGlobalWindow.cpp`, `DOMStorage.cpp`, etc.
- **SpiderMonkey (JavaScript engine):** Lastly, we instrumented the JavaScript interpreter for recording value manipulation on variables and objects and also to mark the start and end points of XHR callback execution. The former is achieved by instrumenting the JavaScript bytecode interpreter on `Interpreter.cpp` and `jsapi.cpp`.

Overall, our instrumentation is quite sparse and we believe can be easily migrated to another open-source browser such as Chromium. We have added a total of about 450 lines of instrumentation code to Firefox to collect our traces. A total of 13 files were modified.

Deployment strategies: The process of race detection is something that can be performed both online⁴, as the application is running, as well as offline, as an *auditing* step. We envision that as part of beta-testing, traces from multiple users can be analyzed. Note that as we will highlight in Section 5.3, even relatively simple-looking sites can create long traces with a large number of events. At the same time, the number of events relevant to asynchrony and scheduling is relatively small. Our analysis for finding potential races is implemented as a linear pass over the trace. However, if desired, this is something that can be parallelized as well, by splitting longer traces to be analyzed on different machines.

Our focus in this work has been on *lightweight* exploration of asynchrony in JavaScript applications. When broader exploration of the schedule space is desired, a good strategy is to use a collaborative approach to testing, with a number of beta testers or crowd-sourced testers using the web page or application.

5.2 Formalization

Our race detection algorithm processes an execution trace one entry at a time, maintaining a memory map and the happens-before relation as it progresses. In the following, we first define our happens-before relation, then describe how we formally model and analyze traces.

⁴According to the data in Figure 5.2, our analysis is fast enough to be run online, so the beta testers only need to use a differently compiled version of the browser or perhaps a browser with a flag that they turn on. The results of such exploration can be centrally collected and communicated to the site developers.

5.2.1 Defining Happens-Before

Execution blocks: Since JavaScript execution is single-threaded, the execution of event handlers and XHR callbacks is carried out non-preemptively, without interruption. We find it convenient to represent the executions of event handlers, XHR callbacks, and portions of script code executed without pre-emption (“sequential blocks”) as execution blocks with unique IDs. Other entries in an execution trace will also be assigned unique IDs as detailed later⁵.

Defining the happens-before relation: There is no universally accepted definition of the happens-before relation for web-based JavaScript code. We define the happens-before relation (denoted $\leftarrow$) not at the level of low-level memory accesses, but at the level of higher-level language constructs, based on causality information we abstract from JavaScript operational semantics as was illustrated in Figure 5.1. Within each uninterrupted execution block, trace entries are ordered by the program, and, therefore, happens-before order. We define and record a happens-before order between blocks such that if $id \leftarrow id'$ then all trace entries in id happen before those in id' . We order $id \leftarrow id'$ if id' appears *later* in the trace and one of the following hold:

1. Both blocks are sequential blocks. Sequential blocks are ordered by $\leftarrow$ in the order they appear in the trace because of the browser-imposed ordering.
2. Both blocks are event-handling blocks. In order to reflect the order of user interactions, event handling blocks are ordered by $\leftarrow$ in the order they occur in the trace to reflect the order of user interactions.
3. id' is an XHR callback, and the XHR `send` for id occurs within the block with ID id ; this is because the XHR callback can only happen after the `send` operation.
4. id' is a `Promise` callback chained to the block with ID id using chaining constructs `then`, `catch`, `all` or `race`.

⁵Note that these IDs are not to be confused with the statically-assigned numbers in Figure 5.1 as, for instance, the point in an execution where an XHR callback is registered and the point where it is executed are different.

5.2.2 Modeling and Analysis of Traces

In the rest of this section, we formally explain our race detection approach. The set of memory locations (“locations”) manipulated by a JavaScript program is denoted by $Locs$, and the set of values they can take by Val . These variables can take on values of types such as integers or floating-point numbers, strings and object references. The value of an uninitialized locations is $\perp_{Val}$. We treat each field of each object as a unique, separate ordinary location for race-detection purposes. `document.cookie`, `sessionStorage` and `localStorage` are variables manipulated on a web browser by JavaScript programs. The type of each of these variables is a key-value store and they are persisted, i.e., stored on disk. KV denotes the set of these locations.

At each point in the execution, the value of each $kv \in KV$ is a partial map from keys $Keys$ to values $\mathcal{V}$. To treat read and write accesses in the trace uniformly, we treat each (kv, ky) pair as a memory location with name $(kv, ky) \in Locs$. We use $\perp_{Val}$ to represent the value for keys not in the store. Accesses to distinct keys in a given $kv \in KV$, similarly to accesses to different memory locations, do not race with each other. The set of locations that represent DOM elements and are written to using the setter for the `innerHTML` property of a DOM element is denoted by $DOMElts \subseteq Locs$.

The memory map: The value of each location v at each point in the execution is represented by a memory map named $\mathcal{V}$. $\mathcal{V}(v)$ consists of set of pairs of the form (vl, id) where $vl \in Val$ and $id \in IDs$. Intuitively, $(vl, id) \in \mathcal{V}(v)$ means that in this trace or a re-ordering of this trace that has the happens-before relation, v may have the value vl written to it by the block id . When processing a trace entry with ID id is a write access to v , we make critical use of the happens-before relationship computed up to that point. All entries in the memory map $\mathcal{V}(v)$ by trace entries with ID id' that happen before id are removed, representing the fact that they have been overwritten by a later access that happens after them. Other entries in $\mathcal{V}(v)$ are there due to “concurrent” execution blocks and are therefore not removed.

5.2.3 Defining Traces

Formally, a trace is a finite sequence of trace entries $\langle \lambda_0, \lambda_1, \lambda_2, \dots, \lambda_n \rangle$ of the following types:

- **XHR, Promise callbacks, event handling, and sequential blocks:** Trace entries $\lambda = CBBegin(id)$ and $\lambda = CBEnd(id)$ denote the beginning and end of the execution of the callback for the XHR or **Promise** blocks with ID id . $\lambda = HandlerBegin(id)$ and $\lambda = HandlerEnd(id)$ do the same for event handler blocks, and $\lambda = SeqBegin(id)$ and $\lambda = SeqEnd(id)$ for sequential blocks.
- **Key, location, innerHTML accesses:** $\lambda = keyWr(kv, ky, vl, varsRd, id)$ denotes the writing of the value vl for the key ky in the key-value store kv within the block with ID id . The value vl has been computed immediately prior to the write trace entry as the result of an expression over the memory locations $varsRd$. $\lambda = keyRm(kv, ky, id)$ denotes the removal of the value the key ky from kv within block id . $\lambda = varWrite(v, vl, id)$ denotes the writing of the value vl to the location v within block id . Finally, $\lambda = setHTML(hElt, hVal, id)$ denotes the setting of the `innerHTML` property of a DOM element $hElt$ to value $hVal$ within block id .
- **POST, XHR send:** $\lambda = post(url, id, id_{in}, vl, varsRd)$ is a **POST** request XHR call or Window with ID id and the call occurs within an execution block with ID id_{in} . The data posted vl is the computed result of an expression over the memory locations $varsRd$. $\lambda = xhrSend(id, id_{in})$ denotes an XHR **send** operation for the XHR object with ID id that takes place within an execution block with ID id_{in} . This **send** is a **GET** request.
- **Promise chaining constructs:** Trace entry $\lambda = promiseChain(id_{prm}, ids_{chn})$ denotes the one of the chaining operations (**then**, **all**, **race**) of a **Promise** object with ID id_{prm} with the execution blocks with IDs ids_{chn} .

EVT-HANDLER-BEGIN		EVT-HANDLER-END	
$\lambda = \text{HandlerBegin}(id)$	$HB' = \text{TransClose}(HB \cup \{(id_{Evt}, id)\})$	$\lambda = \text{HandlerEnd}(id)$	$id'_{Evt} = id$
$(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB', \mathcal{P}, id_{Seq}, id_{Evt})$		$(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id'_{Evt})$	
SEQ-BLK-BEGIN		SEQ-BLK-END	
$\lambda = \text{SeqBegin}(id)$	$HB' = \text{TransClose}(HB \cup \{(id_{Seq}, id)\})$	$\lambda = \text{SeqEnd}(id)$	$id'_{Seq} = id$
$(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB', \mathcal{P}, id_{Seq}, id_{Evt})$		$(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB, \mathcal{P}, id'_{Seq}, id_{Evt})$	
XHR-POST			
$\lambda = \text{post}(url, id, id_{in}, vl, varsRd)$	$\mathcal{P}' = \mathcal{P} \cup \{(v, id) v \in varsRd\}$	$HB' = HB \cup \{(id_{in}, id)\}$	
$(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB', \mathcal{P}', id_{Seq}, id_{Evt})$			
XHR-SEND			
$\lambda = \text{xhrSend}(id, id_{in})$	$HB' = \text{TransClose}(HB \cup \{(id_{in}, id)\})$		
$(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB', \mathcal{P}, id_{Seq}, id_{Evt})$			
PROMISE-CHAIN			
$\lambda = \text{promiseChain}(id_{prm}, ids_{chn})$	$HB' = \text{TransClose}(HB \cup \{(ids_{chn}, id_{prm})\})$		
$(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB', \mathcal{P}, id_{Seq}, id_{Evt})$			
CB-BEGIN			
$\lambda = \text{CBBegin}(id)$	$id'_{CB} = id$		
$(\mathcal{V}, HB, \mathcal{P}, id_{CB}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB, \mathcal{P}, id'_{CB}, id_{Seq}, id_{Evt})$			
CB-END			
$\lambda = \text{CBEnd}(id)$	$id'_{CB} = \perp$		
$(\mathcal{V}, HB, \mathcal{P}, id_{CB}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB, \mathcal{P}, id'_{CB}, id_{Seq}, id_{Evt})$			
KEY-WRITE			
$\lambda = \text{keyWr}(kv, ky, vl, varsRd, id)$			
$\mathcal{V}' = \mathcal{V}[(kv, ky) := \mathcal{V}(kv, ky) \cup \{(vl, id)\}]$	$\setminus \{(vl', id') id' \leftarrow id \text{ or } id' = id\}$		
$(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}', HB, \mathcal{P}, id_{Seq}, id_{Evt})$			
KEY-REMOVE			
$\lambda = \text{keyRm}(kv, ky, id)$	$\mathcal{V}' = \mathcal{V}[(kv, ky) := \mathcal{V}(kv, ky) \cup \{(\perp, id)\}]$		
$\setminus \{(vl', id') id' \leftarrow id \text{ or } id' = id\}$			
$(\mathcal{V}, HB, \mathcal{P}, id_{CB}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}', HB, \mathcal{P}, id_{CB}, id_{Seq}, id_{Evt})$			
WRITE			
$\lambda = \text{varWrite}(v, vl, id)$	$\mathcal{V}' = \mathcal{V}[(v) := \mathcal{V}(v) \cup \{(vl, id)\}] \setminus \{(vl', id') id' \leftarrow id \text{ or } id' = id\}$		
$(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}', HB, \mathcal{P}, id_{Seq}, id_{Evt})$			
SET-DOM			
$\lambda = \text{setHTML}(hElt, hVal, id)$			
$\mathcal{V}' = \mathcal{V}(hElt) := \mathcal{V}(hElt) \cup \{(vl, id)\}$	$\setminus \{(vl', id') id' \leftarrow id \text{ or } id' = id\}$		
$(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}', HB, \mathcal{P}, id_{Seq}, id_{Evt})$			

Figure 5.4: Trace analysis rules.

5.2.4 Interpreting Traces

Given a trace $Trace = \langle \lambda_0, \lambda_1, \lambda_2, \dots, \lambda_n \rangle$, our race detection algorithm analyzes it by processing it sequentially, one log entry at a time. The algorithm maintains analysis state represented by the tuple $\Sigma = (\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt})$. Here, $\mathcal{V}$ is the memory map. HB is the happens-before relation, which is a partial order over IDs . Whenever new elements are added to HB by a trace processing rule, the transitive closure of the relation is taken to obtain the resultant HB . HB is initially the empty relation. $\mathcal{P}$ is a list of pairs of the form (v, id) where the location v has been read while computing the value submitted by a XHR POST request with ID id . If, while analyzing the postfix of the trace, our algorithm discovers that v 's value while being read by id can be affected by scheduling non-determinism, a race on the read of v in id is declared. Finally, id_{Seq} , and id_{Evt} are the IDs of the last sequential block processed or the last event handling block processed by our algorithm, respectively, or $\perp$ if no such callback or block exists.

Rules for updating the analysis state $\Sigma = (\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt})$ are given in Figure 5.4 and explained below.

Callbacks: CB-BEGIN and CB-END keep track of the ID of the ongoing XHR or `Promise` callback block. SEQ-BLK-BEGIN processes the log entry indicating the beginning of a sequential block with ID id and adds the pair (id_{Seq}, id) to the happens-before relation. SEQ-BLK-END resets id_{Seq} to id . The rules ensure that the occurrence order of sequential blocks in the trace and their happens-before order coincide. EVT-HANDLER-BEGIN and EVT-HANDLER-END operate similarly to the corresponding SEQ-BLK rules. The order of occurrence in the trace of the event handlers is the same as their happens-before order. Event handlers and sequential blocks are not ordered with respect to each other.

Location updates: KEY-WRITE handles the case where key ky in the key-value map kv is updated. We declare a potentially-harmful race if $|Val(v)| > 1$ for any location v read while the new value for the key is being computed ($v \in varsRd$) indicating the potential for non-determinism. The rule computes $\mathcal{V}'(kv, ky)$ from

$\mathcal{V}(kv, ky)$ by adding the pair (vl, id) and removing all pairs (vl', id') such that $id' \leftarrow id$. This particular handling of write accesses in the trace is key to our symbolic execution algorithm. Write accesses overwrite *only* the values that have been written by entries that have a happens-before relationship with them, but the values written by potentially concurrent entries are left intact. KEY-REMOVE (not shown) writes the value $\perp_{val}$ to the key ky . WRITE and SET-DOM are similar to KEY-WRITE above, updating $\mathcal{V}$ for a location v or a DOM element $hElt$ by removing value-id pairs overwritten as dictated by the happens-before relation and adding the new value-id pair written by the current trace entry.

POST, send: The rule XHR-POST declares a potentially harmful race if $|\mathcal{V}(v)| > 1$ for any location $v \in varsRd$, since at least one location used in the computation of vl , the data posted, has the potential for non-determinism. XHR-SEND updates the happens-before relation. $xhrSend(id, id_{in})$ indicates that the send for the XHR with ID id has taken place in block id_{in} . $id_{in} \leftarrow id$. This rule and the fact that CB-BEGIN and CB-END do not modify the happens-before relation encode the fact that only chained XHR calls are ordered by $\leftarrow$ with respect to each other⁶.

Promise chaining: Similar to XHR-SEND rule, PROMISE-CHAIN updates the happen-before relation. $promiseChain(id_{prm}, ids_{chn})$ indicates that **Promise** object with ID id_{prm} is chained with set of **Promise** objects with IDs ids_{chn} . **Promise** specification includes multiple chaining constructs (**then**, **all** and **race**) which can be encoded with our happens-before relation definition: $ids_{chn} \leftarrow id_{prm}$.

5.2.5 Detecting Races

We say $\mathcal{V}(v)$ has *non-determinism potential on a location v* when the memory map contains at least two different values for v , i.e., when there are pairs (vl, id) and

⁶Note that it is possible to define the happens-before relationship differently, for instance, declaring xhr_i to happen before xhr_j if the send entry for xhr_j appears later in the trace than the end of the callback for xhr_i . While such a definition may capture the happens-before relationship observed in a particular schedule more precisely, our definition narrows in on ordering relationships enforced by the JavaScript semantics and excludes those that may have taken place differently in different executions of the same program driven by the same user interaction.

(vl', id') in $\mathcal{V}'(v)$, such that $vl \neq vl'$ and $id \neq id'$. Our algorithm declares a race on a location v while evaluating the WRITE, SET-DOM, and KEY-WRITE rules if the memory map $\mathcal{V}'(v)$ computed by the rule has non-determinism potential on v . We also declare a race when evaluating the XHR-POST and KEY-WRITE rules, if a variable read when computing the value posted or written, $v \in varsRd$, $\mathcal{V}(v)$ has non-determinism potential for v . Of these races, only the ones associated with KEY-WRITE and XHR-POST rules are deemed to be harmful races.

Consider a prefix of the trace in a “quiet” state such that, at the end of the prefix, no XHR callback or execution block is in progress. Suppose that our algorithm has signaled non-determinism potential on a location v while processing a trace entry λ within the last block or callback with ID id in this prefix. Then a different re-ordering of the the execution blocks and/or XHR callbacks while leaving the happens-before relation in the prefix intact may result in a different final value for v , as explained next.

Let (vl, id) and (vl', id') be in $Val(v)$ such that $id' \neq id$ and $vl \neq vl'$. It must be the case that while both id' and id wrote to v , these two execution blocks are not ordered by the happens-before relationship. Otherwise, either (vl, id) or (vl', id') would have been removed from $Val(v)$, according to the update rules of our algorithm and how they use the happens-before relation in variable updates. Therefore, it is possible to modify the execution by delaying the execution of the block with ID id' until *after* the execution of the block with ID id . In this case, the final value of the location v or (kv, ky) would be different at the final state of the newly-obtained execution. This points to a potentially different result produced *purely* as a result of XHR callback scheduling non-determinism.

There are two sources of false positives in our race detection approach. The first is the assumption made in the previous paragraph while obtaining a new execution by delaying the execution of the block with ID id' past other blocks. The assumption is that control decisions made in the original execution based on data values written by the block id are *not modified* in a way that makes the reordered execution infeasible.

In our empirical experience, such cases are rare and can be ruled out by inspection or replaying and validating the reordered execution. The second source is the check performed when a value is written to a persisted location or sent on the network. In these cases, if, while computing the written or sent value, a location with non-determinism is read, our algorithm signals a potentially harmful race. This approach is conservative, i.e., non-determinism in the value of one of the locations in an expression may not result in non-determinism in the value of the result. Even in cases where this is the cause of a false alarm, we believe that the potentially different data values flowing to a persisted output location may be of concern to programmers.

5.3 Evaluation

This section describes the experimental evaluation we have performed on a set of 26 complex pages and our extended analysis on **Promise** related race conditions in the wild and how our tool addresses such cases.

Research Questions: Our goal has been to address the research questions listed below.

RQ1: How common are races on *persistent state* such as `document.cookie`, `localStorage`, and side-effects such as `POST` requests? We consider these to be the most harmful races there are in web applications.

RQ2: How common are races on `sessionState`? Session state is cleared on browser restarts. However, given that the browser is often not restarted for days if not weeks, data in `sessionState` can persist for long periods of time, if not permanently.

RQ3: How common are races on *transient state* such as memory locations and DOM elements? These are generally not the kind of errors that we deem to be highly problematic and, moreover, often these are not even observable by the user.

	XHR	Persistent state writes				Session state writes		DOM writes			
Web site	open send Callbacks Nested	Cookie Nested Cookie Nested POST s	local Storage Nested local Storage	session Storage Nested session Storage	Set innerHTML Nested Set innerHTML INPUT Element Nested INPUT						
edition.cnn.com	3 3 13 0	35 0 0 0	71 28 1	3 30 0	12,065 3 31 0						
mlb.mlb.com	25 25 92 0	68 2 0 0	8 8 2	0 0 2	93 46 39 0						
news.qq.com	8 8 28 0	17 0 0 0	0 0 0	0 0 0	153 10 64 2						
wireless.att.com	22 22 72 3	79 4 0 0	46 32 0	11 0 2	902 847 60 5						
aljazeera.net	12 12 32 2	24 0 0 0	6 0 0	0 0 0	1,095 29 46 7						
bidl.de	32 32 86 1	75 11 0 0	515 1 14	14 1 2	589 17 164 0						
eltiempo.com	10 10 20 0	81 0 0 0	521 0 0	1 1 0	160 2 72 0						
fedex.com.us	10 10 18 0	88 1 0 0	2 0 0	2 0 0	251 25 172 0						
fujitv.co.jp	41 41 122 2	38 0 0 0	27 0 0	16 0 0	172 22 34 0						
gazetta.it	22 22 67 4	115 15 0 0	68 30 0	19 0 2	119 24 52 0						
girlsgogames.com	18 18 44 0	92 4 0 0	30 6 0	11 0 0	21 8 43 0						
indb.com	4 4 12 1	14 0 0 0	4 0 0	0 0 0	66 3 42 0						
milliyet.com.tr	7 7 19 0	46 0 0 0	0 0 0	4 0 0	51 19 210 143						
myvideo.de	20 20 87 0	99 0 0 0	115 0 0	8 0 0	86 0 23 0						
nasa.gov	35 35 113 18	148 0 0 0	11 0 0	8 0 0	703 82 62 8						
ntv.com.tr	10 10 14 0	58 1 0 0	55 0 0	293 0 0	112 3 66 0						
nytimes.com	8 8 0 0	10 0 0 0	6 0 0	2 0 0	35 0 46 0						
optimum.net	29 29 78 8	174 3 0 0	6 0 0	6 0 0	426 258 182 21						
politico.com	27 27 65 9	231 47 0 0	36 18 0	1 0 0	110 19 248 10						
premierleague.com	21 21 101 0	31 0 0 0	0 0 0	0 0 0	223 11 47 0						
radikal.com.tr	13 11 20 0	111 0 0 0	33 0 0	2 0 0	85 5 59 0						
sports.ru	28 28 90 1	48 0 0 0	84 0 0	0 0 0	128 58 52 1						
sportx.com	5 5 0 0	33 0 0 0	31 0 0	0 0 0	49 0 38 0						
tvguide.com	15 15 26 2	90 6 0 0	15 0 0	23 0 0	728 586 57 0						
welt.de	23 23 44 1	112 0 0 0	366 0 0	4 0 0	546 19 68 0						
zaobao.com	37 37 121 0	83 0 0 0	0 0 0	0 0 0	127 36 125 78						

Table 5.1: Characteristics of our benchmarks web sites.

Website	Trace Size (MB)	Compressed Trace Size (MB)	Browsing Time (sec)	Analysis Time (sec)
imdb.com	14.28	1.21	58	2
zaobao.com	30.75	1.40	38	6
news.qq.com	30.77	1.01	78	4
sporn.com	37.49	1.92	49	6
nytimes.com	40.58	1.26	47	8
fujitv.co.jp	45.79	1.39	72	8
girlsgogames.com	51.98	1.92	78	10
gazetta.it	93.02	3.07	182	17
radikal.com.tr	94.20	3.41	102	17
fedex.com.us	95.44	3.30	58	19
milliyet.com.tr	95.78	3.93	133	16
myvideo.de	104.82	4.38	121	18
welt.de	125.11	5.39	233	26
bild.de	138.47	5.74	119	22
mlb.mlb.com	142.81	4.61	81	21
wireless.att.com	151.22	4.47	87	24
tvguide.com	152.90	6.39	250	27
ntv.com.tr	162.09	6.50	158	27
aljazeera.net	169.87	6.72	93	28
sports.ru	209.68	8.33	257	29
edition.cnn.com	216.09	8.13	115	35
eltiempo.com	222.65	11.43	169	43
optimum.net	224.22	9.06	244	41
premierleague.com	264.08	8.16	244	47
politico.com	271.29	11.22	261	48
nasa.gov	407.76	40.91	425	56

Table 5.2: Trace statistics and processing times.

Web site	document.cookie	localStorage	sessionStorage	innerHTML	INPUT element	Memory	postMessage
edition.cnn.com	0	0	0	0	0	12	0
mlb.mlb.com	1	0	0	0	0	28	0
news.qq.com	0	0	0	1	0	6	0
wireless.att.com	1	0	0	9	0	43	0
aljazeera.net	0	0	0	2	1	4	0
bild.de	0	0	0	0	0	58	0
eltiempo.com	0	0	0	0	0	9	0
fedex.com.us	1	0	0	1	0	10	0
fujitv.co.jp	0	0	0	1	0	13	0
gazetta.it	2	0	0	0	0	19	0
girlsgogames.com	2	1	0	0	0	0	0
imdb.com	0	0	0	0	0	8	0
milliyet.com.tr	0	0	3	0	0	5	0
nasa.gov	0	0	0	9	0	14	0
ntv.com.tr	0	0	0	0	0	13	0
optimum.net	2	0	0	10	3	22	0
politico.com	6	0	0	0	2	224	11
radikal.com.tr	0	0	0	1	0	11	0
sports.ru	0	0	0	0	0	17	0
tvguide.com	0	0	0	0	0	15	0
welt.de	0	0	0	0	0	19	2
zaobao.com	0	0	0	0	7	11	0
Totals	15	1	3	34	13	561	13
	$\Sigma = 19$			$\Sigma = 47$		$\Sigma = 574$	

Table 5.3: Races found by our analysis.

5.3.1 Experimental Setup

In our previous work, Chapter 4, we used an instrumented browser to crawl all sites from Alexa’s top 5,000 list. For our experiments here, we used 26 site of these sites that made heavy use of XHRs.

Site statistics: Table 5.1 summarizes various aspects of the sites we picked. We specifically separate the number of operations within XHR callbacks, as these can lead to races. Columns 2–5 show the number of observed XHR `open` and `send` operations, executed XHR `onreadystatechange` callbacks, and the number of callbacks *within* other callbacks (so-called nested XHRs). Columns 6–11 focus on the use of persistent storage. Columns 6–7 show the number of writes to `document.cookie` as well as writes nested in an XHR callback. Column 8 shows the number of XHR `POST` operations in an XHR callback. Columns 9–10 give the number of writes to `localStorage` both in general and within XHR callbacks. Columns 11–12 give the same information for `sessionStorage`. Lastly, columns 13–16 give information about various forms of DOM manipulation such as setting `innerHTML` and changing the contents of `INPUT` elements; both totals and nested variants of these counts are provided.

Trace statistics: The workload used to collect these counts was simply loading the page and applying basic user interactions like button-link clicks. The counts for DOM manipulation are generally higher than those for `document.cookie`, `localStorage`, or `sessionStorage`. It is natural to expect more races on DOM elements as well, compared to more uncommonly used persistent elements. Table 5.2 summarizes information about the traces we used for our analysis. The compression ratios range between 9.97 and 33.83. The percentage of time it takes to analyze a trace compared to the time to record a trace ranges between 3% and 32%.

Detection Time: Table 5.2 shows the trace collection and analysis time as a function of trace size. Analysis time grows approximately linearly with the size of the trace and is a fraction of the trace recording time shown in the second column.

5.3.2 Detection Results

In this section we describe and analyze several representative races found with our approach.

Example 5 [`sessionStorage` in `milliyet.com.tr`.] In the case of `www.milliyet.com.tr`, the race on `sessionStorage` was caused by a shared variable `namespace` that is used for generating the key name. The variable is written at two different locations, one being an XHR callback that is executed using `jQuery.ajax` method. The execution steps are listed below and the relevant code is shown in Figure 5.5:

1. As a page loads, an XHR is created using the `jQuery.ajax` method (included in `www.milliyet.com.tr/D/j/base.js?v=20`) and sent to the server (lines 3–5);
2. once the response is received a user-defined callback is executed using `jQuery.ajax.done` (lines 19–22);
3. `namespace` variable is set to empty string at the end of `jQuery.ajax.done` method (line 21);
4. an external library is initialized by setting the `namespace` variable to `'__uv_'` (line 46);
5. `namespace` variable is subsequently used for generating a `sessionStorage` key (lines 43–45).

In the last step, the value of `namespace` is used for setting an item on the `sessionStorage` for recording user interactions with the web page. In this particular trace, we observe that multiple items are added to the `sessionStorage` using the `namespace` as a prefix (i.e. `__uv_autoprompt_disabled`, `__uv_r`). Any future read operations on the `sessionStorage` keys will depend on this prefix `namespace`. As the XHR callback can race with the `namespace` writes which sets the value for `namespace` to the empty string, it may result in a failed read operation. $\square$

```
1 <script>
2   function MilGraphWithStatsV2(...){
3     jQuery.ajax(method, url,
4       // 1: www.milliyet.com.tr/D/j/base.js?v=20
5       function (data) { //Process data });
6     ...
7   }
8 </script>
9 ...a lot of text and images here...
10 <script>
11   jQuery.extend({
12     ajax: function (url, options){
13       ...
14       var xhr = new XMLHttpRequest();
15       xhr.open(method, url)
16       xhr.onreadystatechange = done;
17       xhr.send(null);
18       ...
19       function done(...){
20         // 2: www.milliyet.com.tr/D/j/base.js?v=20
21         namespace = ""; // write to sessionStorage
22       }
23     });
24 </script>
25 ...
26 <script>
27   // 3: widget.uservoice.com/uE6Mda0sQpbMbkaEhFaUig.js
28   r.prototype.get = function(t) {
29     if (this.storage) {
30       var e = this.storage.getItem(this.makeKey(t));
31       return e;
32     }
33   };
34   r.prototype.set = function(t, e) {
35     this.storage &&
36       this.storage.setItem(this.makeKey(t),
37         JSON.stringify(e))
38   };
39   r.prototype.remove = function(t) {
40     this.storage &&
41       this.storage.removeItem(this.makeKey(t))
42   };
43   r.prototype.makeKey = function(t) {
44     return r.namespace + t
45   };
46   r.namespace = "__uv_"; // write to sessionStorage
47 </script>
```

Figure 5.5: sessionStorage manipulation in milliyet.com.tr.

```
1 <html> // 1: index.html
2 <a id="a_hayat" class="category"
3   href='Javascript:AddCategory("hayat");'>HAYAT</a>
4 <a id="a_spor" class="category"
5   href='Javascript:AddCategory("spor");'>SPOR</a>
6 <div id="r5_content_upd"></div>
7 <script>
8   function AddCategory(categoryname){
9     mynewCategoriesValue += categoryname;
10    GetArticle(mynewCategoriesValue);
11  }
12  function GetArticle(categoryValue){
13    var c_url = c_srcDomain + categoryValue;
14    $("#r5_content_upd").load(c_url, cbFunction);
15  }
16 </script>
17 </html>
```

Figure 5.6: Race on `innerHTML` manipulation in `radikal.com.tr`.

Example 6 [`innerHTML` in `radikal.com.tr`.] Another race-inducing execution trace is obtained from `www.radikal.com.tr`, where two XHR callbacks race with each other when it comes to writing to `innerHTML` property of a `<div>` block in the DOM. In this news web site, users can filter displayed news according to their preferred categories, by using category links on the page. Each click action on a category link will generate an XHR call to server to gather news for the corresponding category. The race is captured within the execution steps listed below; the corresponding code is shown in Figure 5.6:

1. User clicks on a category link (lines 2–3) for enabling a category on the web page triggering the `AddCategory` method (lines 8–11);
2. `AddCategory` method will then trigger `GetArticle` method (line 10);
3. `GetArticle` method (lines 12–15) makes an XHR call using `jQuery.load` for `<div>` with id `r5_content_upd` (line 14);

4. user clicks another category link (lines 4–5) triggering another XHR call;
5. both XHR calls sets `innerHTML` of the same `<div>` on the DOM (line 8).

At the end of this execution, the displayed DOM will depend on the order of the responses returned from the server as the `<div id= r5_content_upd>` is directly affected by the asynchronous XHR callback. $\square$

Example 7 [`document.cookie` in `gazetta.it`.] In this example of a `document.cookie` race from `www.gazetta.it`, the web site uses an application monitoring library (DynaTrace Real User Monitoring found at `dynatrace.com`) for recording and POST ing user actions and browsing experience to a remote server. Each POST request initiated by this library updates the key `dtCookie` of the `document.cookie` with the response value from the server. A race occurs when *multiple* XHR calls try to write the new response value to `document.cookie`. The execution steps are listed below, with line references from the code shown in Figure 5.7:

1. At the page load time, an XHR call is created for initializing the monitoring process (line 9);
2. the data is sent to a URL constructed from `dtCookie` value (lines 6–8);
3. `dtCookie` key is updated with the server’s response (lines 35–37);
4. a new XHR is created at `onLoad` DOM event for posting loading time of the page (lines 11–13);
5. callbacks of each XHR try to write to the same `document.cookie` key `dtCookie` (lines 35–37).

The value of `dtCookie` key depends on the order of the XHR callbacks, causing possible issues with future POST operations, as the value is used for URL generation. $\square$

```
1 <script>
2 // 1: rum-dytrc.gazzetta.it
3 //      /ajax/dtagent60_bjnprs3t_7082.js
4 function sb(a, b) {
5     var d;
6     d = d + "?" + "dtCookie\x3d"
7       + encodeURIComponent(v(Ua)) + ";";
8     + encodeURIComponent(document.location.href);
9     fb(cb, d, c, e)
10 }
11 E(q, "load", function(){
12     fb(m, b.path, n, b.data);
13 })
14 function fb(a, b, c, d) {
15     var h;
16     h.onreadystatechange = function() {
17         4 == a.readyState && (200 == a.status ?
18             e(a.responseText) : db && B.sf &&
19             eb.push({path: b,data: c}), a = m)
20     }
21     h.open("POST", b, c);
22     h.send(d);
23 }
24 function e(a) {
25     a = a && a.split("|");
26     for (var b = 1; b < a.length; b++) {
27         var c = a[b].indexOf("dtCookie\x3d");
28         if (-1 < c) {
29             y("dtCookie",
30                 decodeURIComponent(a[b].substr(c + 9)));
31             break;
32         }
33     }
34 }
35 function y(a, b) {
36     document.cookie = a+"\x3d"+b
37         %+"\x3dpath\x3d/" + w.domain;
38 }
39 <\script>
```

Figure 5.7: Race on document.cookie manipulation in gazzetta.it.

```

1 XHR [126F0800] Send
2 ...
3 READ ID[235205312] = document : JSObject
4 WRITE PROP ID[242159440] = cookie : JSString
5 Cookie [1020D800] Write "fsr.s=%7B%22v2%22%7D" (1)
6 ...
7 READ ID[235205312] = document : JSObject
8 WRITE PROP ID[242159440] = cookie : JSString
9 Cookie [1020D800] Write "fsr.s=%7B%22v2%22%2C%7D" (2)
10 ...
11 XHR [126F0800] Callback
12 BEGIN XHR_Callback
13 ...
14 READ ID[235205312] = document : JSObject
15 READ PROP ID[242159440] = cookie : JSInteger - (3)
16     "fsr.s=%7B%22v2%22%2C%7D"
17 ...
18 READ ID[235205312] = document : JSObject
19 WRITE PROP ID[242159440] = cookie : JSString
20 Cookie [1020D800] Write
21     "fsr.s=%7B%22v2%22%2C%22%3A1%7D" (4)
22 END XHR_Callback

```

Figure 5.8: Trace from optimum.net illustrating a false positive.

5.3.3 False Positives

In looking for possible false positives, we have decided to focus our attention on both persistent races (19) and those on the DOM state (47). Out of the 66 races we investigated, we identified only *two cases* as false positives. These are `document.cookie` races in `fedex.com` and `optimum.net`.

Figure 5.8 shows the trace from `optimum.net`, with writes to the cookie key `fsr.s` at multiple locations (lines 6, 10, and 22), one of which is within an XHR callback. In this case, `fsr.s` is used for collecting site statistics where the key value is updated by concatenating new values (lines 11–22). Although the XHR callback may happen in between two cookie writes on lines 5 and 6, causing a different value for the key, at the end of the execution the cookie key `fsr.s` will contain all the written values but in different orders. In this case, the program treats this value as a set and not a

list, so, while we catch the non-determinism correctly, this is not really a bug in the program. The false positive in `fedex.com` is very similar, with a race is on cookie key `s_sess`, updated within an XHR callback.

Benign memory races: We would also like to highlight an example of a memory race that does not have persistent consequences. On `radikal.com.tr`, there is a global variable “duration” that is used to measure time elapsed at different points throughout the execution on which there is a race between its update in sequential code and in an XHR callback. The value of this memory location does not propagate to any persisted state and cause non-determinism.

5.3.4 *Promise Related Race Conditions*

Even though **Promise** constructs are included in the ECMAScript specification by the release of the ECMAScript6 [21], they have been used in server side JavaScript (Node.js) community for a while with the use of external **Promise** libraries. For investigating possible races conditions with the use of this libraries, we conducted an exploratory study on reported issues on the modules using popular Node.js **Promise** libraries, namely Q and Bluebird. For this exploratory study, we extracted full list of Node.js modules that has a dependency to the **Promise** libraries, Q (over 4300 dependent modules) and Bluebird (over 4000 dependent modules), and than conducted an issue search on related GitHub repositories. For this search, we used GitHub’s issue tracking system by issuing queries “race condition” and to limit our focus on the important cases, we looked for cases that are classified as “bug” by the developers. In Table 5.4, you can find number of such issues reported in the corresponding modules’ reported issues. Later, we conducted a manual inspection on this cases to find their relation to use of **Promise** libraries. With this inspection, we reported on the issues that are caused by a race condition on the used **Promise** constructs.

This study shows that even with the use of more structured asynchrony constructs like **Promise** s, race conditions still happens in practice. To address this, we extended our race detection mechanism for detecting data races for **Promise** s. On the other

Module Name	Module Issues	Race Related Bugs
appium-stable	https://github.com/appium/appium/issues	9
bem	http://github.com/bem/bem-tools/issues	1
cca	https://github.com/MobileChromeApps/mobile-chrome-apps/issues	2
clay-webdriverio	https://github.com/webdriverio/webdriverio/issues	1
commoner	https://github.com/reactjs/commoner/issues	1
dalek-internal-actions	https://github.com/dalekjs/dalek-internal-actions/issues	1
fielddb	https://github.com/OpenSourceFieldlinguistics/FieldDB/issues	1
meteor-pathwatcher-tweaks	https://github.com/atom/node-pathwatcher/issues	1
phantomas	https://github.com/machre/phantomas/issues	2
sockethub	https://github.com/sockethub/sockethub/issues	2
steroids	https://github.com/AppGyver/steroids/issues	3
webdriverio	https://github.com/webdriverio/webdriverio/issues	1
webgme	http://github.com/webgme/webgme/issues	6
airsonos	https://github.com/stephen/airsonos/issues	1
bookshelf	https://github.com/tgriesser/bookshelf/issues	1
clay-sequelize	https://github.com/sequelize/sequelize/issues	5
deployd	https://github.com/deployd/deployd/issues	1
ghost	https://github.com/TryGhost/Ghost/issues	7
kraken-js	https://github.com/krakenjs/kraken-js/issues	1
pimatic	https://github.com/pimatic/pimatic/issues	2
remotestoragejs	https://github.com/remotestorage/remotestorage.js/issues	4
strider	https://github.com/Strider-CD/strider/issues	2

Table 5.4: Node.js modules with reported issues related to race conditions on Promise constructs.

hand, as these constructs are not default in most of the web browsers the usage by the client side web applications is limited, hence we couldn't explore the possible data races on these web sites. However with the help of our race detection tool, developers can check for race conditions that might have effects on the persistent storage.

5.4 Discussion

Returning to the research questions in Section 5.3, we clearly see that according to Table 5.3, races on persistent state (RQ 1) are quite uncommon, with only 19 for 26 sites. Races on session state (RQ 2) are also uncommon (only 3 races observed), in part because session state is used not as frequently as cookies (Figure 5.2). Lastly, races on transient state are considerably more frequent. While changes to HTML content may technically be races, one could argue that most of these are ephemeral. Indeed, the execution model of JavaScript-based web pages induces a great deal of non-determinism. This is because many if not most web pages are not the same across multiple reloads: the ads on the sides of the page change; content of pages often changes (consider a rapidly changing news site), sometimes the experience of the first visit to a page and subsequent ones is different because of cookies. The browser user has been trained not to expect much consistency and, when everything else fails, to reload the page. Additionally, the JavaScript execution model is extremely permissive: errors are “swallowed” by the runtime (the current event handler is terminated) and in many cases pages can survive exceptions and keep on running.

Our observations mesh well with the anecdotal experience of the user encountering any problem on the web: one only needs to reload the page for the problem to go away. In a sense, web programming is very forgiving. This is different from thread-related races in desktop applications and also data races in mobile apps [61, 43], where researchers are able to replicate races with obvious visual consequences (mangled or upside down images on the screen, for instance).

As such, we argue that data races that only affect the *transient* portion of the shared state such as the DOM do not matter. The ones that *do* matter are races that affect persistent state such as `document.cookie`, `localStorage`, as well as POST

requests to the server. According to the results in Figure 5.3, persistent or *harmful* races are considerably more uncommon. However, it is these races, we argue, that have a higher chance of being fixed by developers. Lastly, ours is a *neighborhood exploration* technique. Given an execution trace, we explore the possible interleavings of callbacks. We record enough information to be *sound* for a given trace, but we choose to primarily focus our attention on write-write races on persistent state.

5.5 Conclusions

This study proposes an alternative way of looking at what constitutes a race in web applications written in JavaScript. We advocate a focus on races that are caused by asynchronous callbacks and their order of arrival, primarily investigating races produced by the `XmlHttpRequest` (XHR) mechanism. Unlike prior work which concluded that there is ample potential for races in JavaScript, our findings suggest that given the forgiving nature of JavaScript applications, damaging, persistent races are considerably more rare.

Nevertheless, we propose a lightweight algorithm that explores different schedules in the “neighborhood” of a particular runtime trace. Our approach avoids the imprecision of static analysis and the combinatorial explosion and scalability issues of runtime schedule exploration. Our static analysis technique achieves good performance over other techniques by encoding the effects of multiple schedule ordering into the written values via merging the *concurrently* written values. Our experiments showed that our race detection technique can scale enough to be applied online.

We find and investigate a total of 19 harmful races and 621 benign races in 26 web sites. We also conducted an exploratory study on race related bug reports on JavaScript libraries that uses well known `Promise` libraries to investigate on how common are race conditions with the use of new asynchronous programming constructs. Our study showed that race conditions are still possible even with the use of better asynchronous constructors and race detection tools can help developers to detect such cases in development time.

Part II

Race Detection for Applications using Hybrid Programming Models

Chapter 6

INTRODUCTION

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

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

Using the dataflow programming model in conjunction with shared memory mechanisms can make it convenient and natural for programmers to express the parallelism inherent in a problem as evidenced by recent proposals [34, 78, 80] and adoptions [3, 4, 7]. These hybrid programming models provide programmers with dataflow abstractions for defining tasks as the main execution unit with corresponding data dependencies. Contrary to the pure dataflow model which assumes side-effect free execution of the tasks, these models allow tasks to share the data using some form of thread synchronization, such as locks or transactional memory (TM). In this way, they facilitate implementation of complex algorithms for which shared state is the fundamental part of how the computational problem at hand is naturally expressed.

Enabling a combination of different programming models provides a user with a wide choice of parallel programming abstractions that can support a straightforward implementation of a wider range of problems. However, it also increases the likelihood of introducing concurrency bugs, not only those specific to a given well-studied pro-

programming model, but also those that are the result of unexpected program behavior caused by an incorrect use of different programming abstractions within the same program. Since the hybrid dataflow models we consider in this study are quite novel, many of the bugs that belong to the latter category may not have been studied. The goal of this part is to identify these bugs and design verification tool and race detection tool support for the developers using hybrid programming models. To achieve this goal, we conducted set of empirical studies over a representative model of this class of hybrid programming models.

In the first study of this part, Chapter 7, we present a dynamic verification tool for characterizing and exploring behaviors of programs written using hybrid dataflow programming models. We focus in particular on the Atomic DataFlow (ADF) programming model [34] as a representative of this class of programming models. In the ADF model, a program is based on tasks for which data dependencies are explicitly defined by a programmer and used by the runtime system to coordinate the task execution, while the memory shared between potentially concurrent tasks is managed using transactional memory (TM). While ideally these two domains, dataflow and shared memory, should be used properly in a program, any misused data dependency or missed synchronization between concurrent tasks can lead to an unexpected interleaving causing an incorrect program behavior.

We devised a randomized scheduling method for exploring programs implemented using ADF programming model. The key challenge in our work was precisely characterizing and exploring the concurrency visible and meaningful to the programmer, as opposed to the concurrency present in the dataflow runtime or TM implementations. For exploration of different interleavings, we adapted the dynamic exploration technique “Probabilistic Concurrency Testing (PCT)” [14] to ADF programs in order to amplify the randomness of observed schedules [11]. For shared memory concurrent programs, PCT provides probabilistic guarantees for bug detection. By properly selecting the scheduling points that PCT randomly chooses from, we aim to provide a similar guarantee for ADF programs.

As the second study of this part, Chapter 8, we presented a lightweight race detection mechanism for this class of hybrid programming models. Our detection mechanism attempts to combine the advantages of static and dynamic analysis by recording a runtime execution trace and later doing an efficient static analysis by exploring neighboring schedules. With the hybrid nature of these programming models, it can be challenging to handle synchronization between possibly concurrent dataflow tasks and commonly accessible shared state. We argue that our race detection mechanism can be useful detecting possible concurrency bugs before deployment.

We presented our formalization of a *modular happens-before* relation for Atomic DataFlow (ADF) [34] programming model, yet it can be easily extended for different models. The main challenge in developing a race detection mechanism for this kind of hybrid programming models is to define a reasonable happens-before relation over the executing components. We present a modular representation of execution orders imposed by different programming models, i.e. dataflow and shared memory, which can be used for extending the detection mechanism for different type of shared memory synchronization systems, i.e. lock-based, transactional memory.

Finally, we implemented our detection mechanism as a generic detection tool prototype that can be configured for different types of hybrid programming models where concurrent accesses are protected with different synchronization mechanisms. The tool designed to integrate well with the available testing frameworks, i.e. our randomized testing scheduler can extend the explored space by collecting multiple traces.

Chapter 7

RANDOMIZED TESTING SCHEDULER FOR HYBRID PROGRAMMING MODELS

In this chapter, we present a dynamic verification technique for a class of concurrent programming models that combine dataflow with shared memory programming. In this class of hybrid concurrency models, programs are built from tasks whose data dependencies are explicitly defined by a programmer and used by the runtime system to coordinate task execution.

Differently from pure dataflow, tasks are allowed to have shared state which must be properly protected using synchronization mechanisms, such as locks or transactional memory (TM). While these hybrid models enable programmers to reason about programs, especially with irregular data sharing and communication patterns, at a higher level, they may also give rise to new kinds of bugs as they are unfamiliar to the programmers. We identify and illustrate a novel category of bugs in these hybrid concurrency programming models and provide a technique for randomized exploration of program behaviors in this setting.

7.1 Motivation

In this section, we describe an unexpected execution scenario for motivating our dynamic verification method. Due to the asynchronous concurrent execution of tasks in the ADF model, users can face unexpected execution orders causing atomicity violations between dataflow tasks. To illustrate such a behavior, consider two ADF tasks in Figure 7.1, *max_min* that compute the maximum and minimum values from two input streams while updating a global minimum and maximum, and *comp_avg*

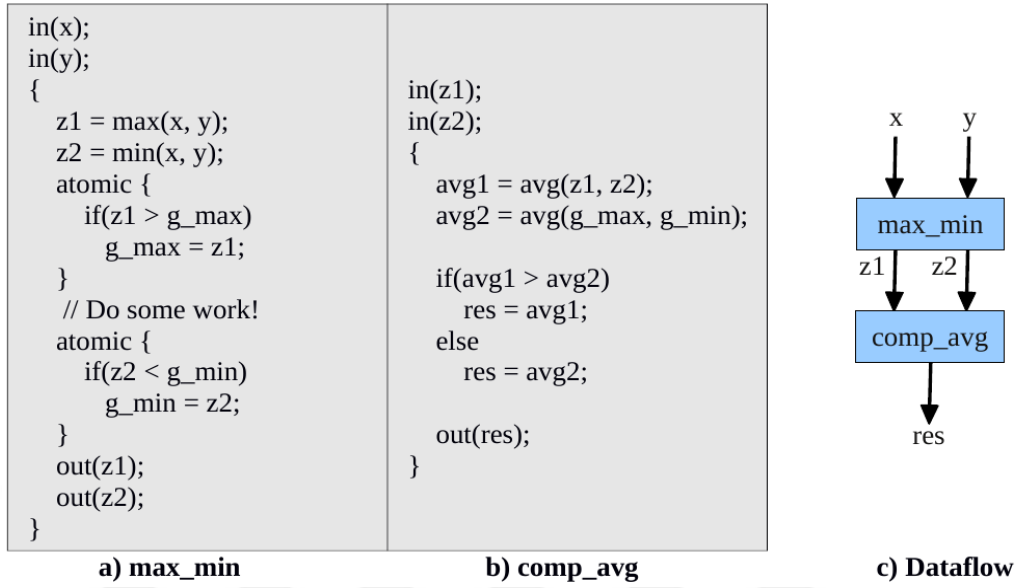


Figure 7.1: Motivating buggy example for hybrid programming models.

that uses the output streams provided by *max_min* for comparing the average values of *g_max* and *g_min* with the input values and returning the bigger one. As seen in Figure 7.1-c, the dependencies between these tasks can be using the expressed with ADF programming model naturally as shown in Figure 7.1-a and b. However, while these particular implementations appear correct separately, when combined, they may result in unexpected behavior in an ADF execution. As the updates on the global variables, *g_max* and *g_min*, are performed in separate atomic blocks, concurrently running tasks can read incorrect values of global variables. Consider an execution where the first pair of integers from the input streams *x* and *y* are processed by *max_min* and then passed to *comp_avg*. During the execution of *comp_avg*, *max_min* can start to process the second pair and update *g_max* value, causing *comp_avg* to read the new *g_max* value from the second iteration while reading *g_min* value from the first one. Such concurrency scenarios that arise due to an interaction between dataflow and shared memory may be difficult to foresee for a programmer and are not addressed properly by verification methods for pure dataflow or pure shared memory model.

7.2 System Overview

7.2.1 Probabilistic Concurrency Testing

The ‘Probabilistic Concurrency Testing (PCT)’ method relies on the observation that concurrency bugs typically involve unexpected interactions among few instructions that are executed by a small number of threads [58]. For capturing these unexpected thread interactions, PCT defines a *bug depth* parameter as the minimum number of ordering constraints that are sufficient to find a bug and uses a randomized scheduling method, with provably good probabilistic guarantees, to find all bugs of low depth.

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

Consider a program with n threads that together execute at most k instructions. Assuming that we want to find bugs with depth d , the pseudocode for the PCT algorithm in this setting is given in Figure 7.2. PCT provides a guarantee of finding a bug of depth d with the probability at least $1/nk^{d-1}$. The algorithm maintains a *priority* and *change_points* list for holding the priorities assigned to the threads and randomly chosen priority change points. After initializing these lists, the algorithm starts the scheduling of threads and executes until there are no available threads while checking for possible priority change points.

7.2.2 Our Method and Implementation

The ADF programming model has an inherently asynchronous concurrent execution model, where tasks can be enabled and executed multiple times. In addition, pro-

Ensure: Program P , integer n , k , and d

Require: $d \geq 0, n \geq \text{maxthreads}(P), k \geq \text{maxsteps}(P)$

```

function PCT( $n, k, d$ )
  for all  $tId \in n$  do
     $\text{priority}[tId] = \text{rand}(d, d + n);$ 
  end for
  for  $i = 0 \rightarrow d - 1$  do
     $\text{change\_points}[i] = \text{rand}(1, k);$ 
  end for
  while  $\text{availableThreads} > 0$  do
     $\text{ScheduleThread}();$ 
    if  $\text{currentStep} \in \text{change\_points}$  then
       $\text{newPriority} = i \in \{0, \dots, d - 1\}$ 
       $\text{priority}[\text{currTId}] = \text{newPriority}$ 
    end if
  end while
end function

```

Figure 7.2: Probabilistic Concurrency Testing (PCT) algorithm.

grammers are allowed to provide their custom synchronization using transactional memory to protect certain code blocks (not necessarily entire tasks) in ADF tasks. This can potentially influence the dataflow execution. In order to fully investigate behaviors of programs written using a hybrid model such as ADF, the dynamic exploration technique has to be aware of both the dataflow structure and the specifics of the shared memory synchronization mechanism. Furthermore, the dynamic verification tool should not simply instrument the platform implementations for transactional memory, atomic blocks and dataflow. This would not only be very inefficient, but it would also not provide value to the programmer. The user of a hybrid concurrent

programming model is not interested in the concurrency internal to the platform implementing the model, which should be transparent to the programmer, but only in the non-determinism made visible at the programming model level.

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

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

1. Whenever a task becomes enabled, randomly assign one of n priority values between d and $d + n$ to a thread associated with the task.
2. Pick $d - 1$ random priority change points $k_1, \dots, k_{d-1}$ in the range of $[1, k]$ and associate priority value of i to k_i .
3. Schedule a thread with the highest priority and execute it sequentially. When a thread reaches the i -th change point, change its priority to i .

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

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

We implemented our exploration technique as a separate testing mechanism into the ADF framework. With this mechanism, users can choose the testing scheduler for

exploring the behaviors of their applications with different task ordering for a given bug depth. Differently from conventional testing, our technique provides probabilistic guarantees for finding bugs and the overall detection probability can be increased by running our technique multiple times.

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

Consider the motivating example in Figure 7.1 with input streams of length 2, our exploration technique can catch the the described buggy behavior with bug depth 2 as follow:

Initialization Random priorities between $d-(n + d)$ (2-6 as the length of the input streams is 2, there can be at most 4 enabled tasks) will be assigned to the enabled tasks. As the only enabled task is *max_min*, let's assume it is given a priority of 4.

Later, $d - 1$ (1) priority change points will be assigned randomly among the start and end points of all atomic sections, assume this change point (as we are exploring bug depth 2) is chosen to be at the end of first atomic block in *max_min* task.

First iteration The scheduler starts the execution by choosing the task with the highest priority. When the execution comes to a priority change point, the priority is lowered causing scheduler to check for a task with higher priority. In this case, *max_min* will continue to execute as there is no other enabled task.

After finishing the execution *max_min* task will enable the *comp_avg* task resulting in a priority assignment to it. Assume that the scheduler assigned 2 as the priority for the *comp_avg*.

The next set of inputs from the streams will enable *max_min* task again with new assigned priority to be 3.

Second iteration Now scheduler will choose the enabled task with the highest priority for execution, which is *max_min* in this case.

While executing the *max_min* task, the priority will be changed at the priority change point and set to 1.

As a result scheduler will now choose *comp_avg* to execute causing the buggy behavior explained in Section 7.1.

7.3 Summary

In this chapter, we identified and illustrated a novel category of bugs in the hybrid concurrency programming models that make use of dataflow and shared memory programming models and provided a technique for randomized exploration of program behaviors in this setting. While these hybrid programming models both ease the programmability of concurrent programs and provides higher performance than pure dataflow and pure shared memory counterparts, using synchronization mechanism from these domains may result in subtle concurrency bugs.

In an effort to implement this exploration technique, we investigated and instrumented Atomic DataFlow (ADF) framework by adding a single-threaded testing scheduler. On top of this testing scheduler, we implemented our randomized exploration technique based on “Probabilistic Concurrency Testing (PCT)” [14]. Later, we have investigated ADF implementations of DWARF benchmark applications [9] and detected anomalies in some of the applications.

The conducted research in this chapter, form the basis for our next chapter where we further investigate possible data races in this class of hybrid programming models.

Chapter 8

RACE DETECTION FOR PROGRAMS USING HYBRID PROGRAMMING MODELS

In this chapter, we present our race detection mechanism for hybrid programming models that combine dataflow with shared memory programming. Our detection mechanism attempts to combine the advantages of static and dynamic analysis by recording a runtime execution trace and later doing an efficient static analysis. One way to see our approach is that it explores *neighboring* schedules for a particular runtime execution by analyzing possible re-orderings of the *concurrent* tasks and composing atomic blocks with a single pass over the observed execution actions.

We give a formal definition of a *modular happens-before* relation for hybrid programming models and later describe our race detection mechanism formally. While we based our formal definitions on Atomic DataFlow (ADF) programming model, but both the formal *happens-before* relation and our generic race detection prototype implementation can be easily extended for different hybrid programming languages using dataflow constructs with different shared memory synchronization mechanisms (i.e. locks, transactional memory).

8.1 Overview

There is an increased interest in hybrid programming models that combines dataflow programming model with shared memory constructs [78, 27, 80, 34, 3, 4, 7] due to its potential performance gain over pure shared memory or pure dataflow programming models. Although there are minor differences, the main idea behind these programming models is to decrease the amount of synchronization between concurrently exe-

cuting entities (threads, processor cores etc.) via use of dataflow like constructs while keeping the advantages of being able to access shared state effectively.

Programs are represented in terms of tasks and data dependencies between each other. Typically, programmers are required to explicitly defines input and/or output dependencies between tasks. Later, this information is used for building up the dependency graph (statically or dynamically) leveraging advantageous of dataflow model for governing the execution of these tasks. As the execution progresses, tasks become enabled by receiving the required data from previous tasks and gets scheduled by the runtime. While pure dataflow programming models enforce the tasks to be side-effect free, proposed hybrid programming models can benefit from the shared memory programming model for accessing shared complex data structures (arrays, maps etc.) without any additional copy operations. Naturally, concurrent accesses to the shared state needs synchronization which is commonly provided by the programming model itself, while programmers are given the freedom to define their own synchronization using atomic block within tasks.

While these hybrid programming models enabled different levels of abstraction for programming highly concurrent programs, it also enables new types of subtle concurrency bugs to appear. While ideally these two domains, shared memory and dataflow synchronization, should be well defined within a program, misused synchronization might result in incorrect behavior. While there are proposed solutions [15, 103, 84, 26] for different task-based asynchronous programming models, there is none that directly targets this kind of hybrid programming model.

We propose a race detection mechanism that attempts to combine advantages of both static and dynamic detection techniques. Our detection mechanism is based on static analysis of the traces collected at runtime. The collected traces contains information about each memory access as well as happens-before information about the executed tasks. While trace is collected from single execution, our detection mechanism explores multiple (neighboring) orders by our efficient static analysis. The key idea behind our static analysis is that we can encode the effects of different

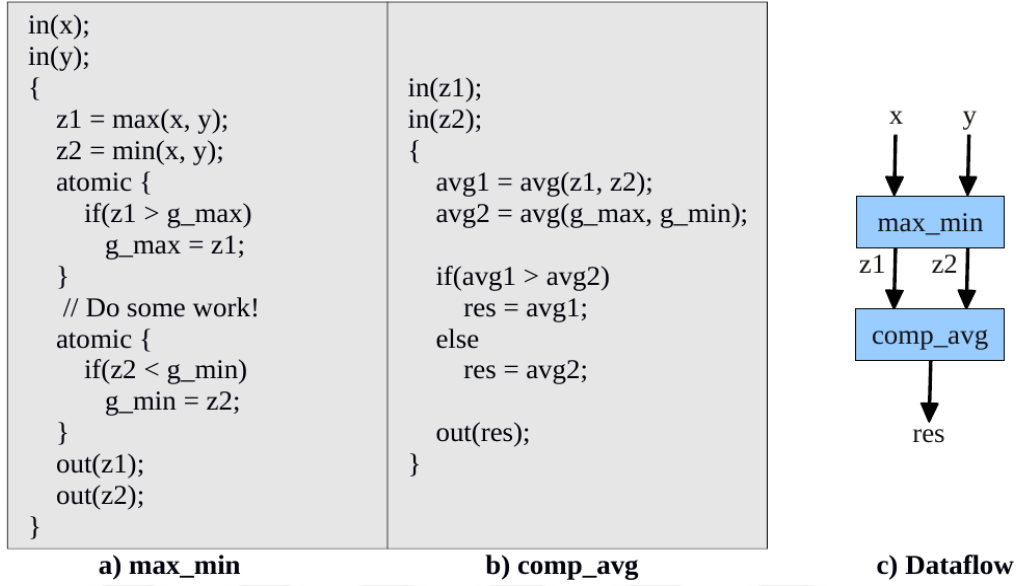


Figure 8.1: Data race resulted by incorrect synchronization between tasks.

ordering of tasks within each memory location by merging the values. This way our technique can check the effects of different schedules by constructing a memory map which keeps track of merged values written by concurrent entities. To identify these concurrent entities in an execution, we define a *modular happens before* relation by combining the earlier ordering definitions for shared memory [49, 62] and dataflow programming models. While our definition of *happens-before* is focused on Atomic DataFlow (ADF) [34] programming model, it can be easily adapted to other hybrid programming model as the detection mechanism and *happens-before relation* definition is easy to extend.

8.1.1 Data Races in Hybrid Programming Models

As the underlying execution model is based shared memory systems, the data race definition can be derived from the earlier definitions [91, 29, 83, 75]. A data race is a condition where two concurrently executing entities (i.e threads or tasks) access a shared memory without proper synchronization, and at least one of the accesses is a ‘write’ operation. There are two possible sources of data race in hybrid programming

models: missing data dependency and incorrect synchronization between concurrent tasks.

While most of the hybrid programming models provide default synchronization mechanism for tasks, they also give programmers freedom to introduce their own synchronization for performance gain. This freedom may result in cases where programmers miss proper synchronization between concurrently executing tasks. To illustrate such a behavior, consider two ADF tasks in Figure 8.1, *max_min* that compute the maximum and minimum values from two input streams while updating a global minimum and maximum, and *comp_avg* that uses the output streams provided by *max_min* for comparing the average values of *g_max* and *g_min* with the input values and returning the bigger one.

As seen in Figure 8.1-c, the dependencies between these tasks can be expressed with ADF programming model naturally as opposed to other programming models, Figure 8.1-a and b. However, while these particular implementations appear correct separately, when combined, they may result in unexpected behavior in an ADF execution. As the updates on the global variables, *g_max* and *g_min*, are performed in separate atomic blocks, concurrently running tasks can read incorrect values of global variables. Consider an execution where the first pair of integers from the input streams *x* and *y* are processed by *max_min* and then passed to *comp_avg*. During the execution of *comp_avg*, *max_min* can start to process the second pair and update *g_max* value, causing *comp_avg* to read the new *g_max* value from the second iteration while reading *g_min* value from the first one resulting in a “read-write” race. Such concurrency scenarios that arise due to an interaction between dataflow and shared memory may be difficult to foresee for a programmer and are not addressed properly by verification methods for pure dataflow or pure shared memory model.

8.2 Preliminary

Our race detection is based on happens before relation [53] that is generated at runtime and then analyzed statically for any conflicting accesses to the shared memory

locations. Similar to other happens before based race detection mechanisms, we first describe the possible actions of a hybrid program and later define a modular happens before relation upon the synchronization constructs available in hybrid programming models. In the remaining of this section, we will give some preliminary descriptions that includes description of actions in hybrid programming models, happens before relation and data race definition for hybrid programs.

8.2.1 Actions in Hybrid Programming Models

We start by describing possible actions that a hybrid program, that uses dataflow constructs upon shared memory programming model, can take. These actions are separated into two categories: memory access actions and synchronization actions. Although there are other types of actions (i.e. function calls, arithmetic operations) that a hybrid program can take, we will omit them as they are not directly related to data races. During the execution of a program, at each step a task can take one of the actions from the following categories:

- **Memory access actions:** `read( $t, blk, addr, vl$ )` by task t reads the current value vl of memory address $addr$ within atomic block blk and `write( $t, blk, addr, vl$ )` by task t writes the value vl to memory address $addr$ within atomic block blk . In case of non-atomic memory access blk can be omitted from the action description.
- **Synchronization actions:** In hybrid programming models there are two types of synchronization actions that programmers can benefit from: dataflow and shared memory synchronization. While dataflow synchronization is based on input-output relations between tasks, the shared memory synchronization constructs can vary widely. Depending on the underlying runtime one might use traditional lock based synchronization or as in our case transactional memory (TM) constructs.

- **Dataflow synchronization:** `pass-token( $t, t'$ )` by task t sending an output token to task t' that corresponds to fulfilling the input-output dependency between tasks.
- **Shared memory synchronization:** `begin-atomic( $t, blk$ )` by task t marking the start of an atomic block blk and `end-atomic( $t, blk$ )` by task t marking the end of an atomic block blk . Similarly, we can extend these actions for different shared memory synchronization constructs like locks by adding `lock( $t, lVar$ )` and `unlock( $t, lVar$ )` respectively.

8.2.2 Defining Happens-Before Relation and Data Race

As with all new programming models, one has to have a new happens before relation defined for detecting races. Happens before relation over the execution of a program gives a partial order for each performed action which is then used to explore multiple possible orderings of the actions taken. In the case of hybrid programming models, defining happens before relation can be defined modularly by combining ordering relations defined for each programming model. For our case, we defined our happens before relation for hybrid programs using happens before relation defined for dataflow and shared memory models that uses TM as their synchronization mechanism.

- **Dataflow order:** In dataflow programming models, *dataflow order* (denoted by $\leftarrow_{dfo}$) is defined by using the input-output relations over the tasks. We order $id_t \leftarrow_{dfo} id'_t$ if task with ID id_t sends its output to task with ID id'_t . With dataflow order, we basically represent a partial ordering between each access action performed within a task. With $id_t \leftarrow_{dfo} id'_t$, we describe an execution where each access action in id_t is performed before the access actions within id'_t .
- **Transactional order:** For shared memory models using TM, we follow *transaction order* (denoted by $\leftarrow_{tmo}$) which is a total order over execution of all transactions (atomic blocks). We order $id_{blk} \leftarrow_{tmo} id'_{blk}$, if atomic block with ID

id_{blk} is executed before atomic block with ID id'_{blk} . With transactional order, we give a total ordering between atomic blocks hence on their access actions. With $id_{blk} \leftarrow_{tmo} id'_{blk}$, we describe an execution where each access action in id_{blk} is performed before the access actions within id'_{blk} .

- **Hybrid happens-before order:** We define *hybrid happens-before* partial order (denoted by $\leftarrow_{hbb}$) as the transitive closure of the program order for all tasks, *dataflow order* and *transaction order* over the executed actions. We say $id_{act} \leftarrow_{hbb} id'_{act}$, if one of the following holds:

- both actions, id_{act} and id'_{act} , are executed by the same task with ID id_t and id_{act} performed before id'_{act} .
- $id_t \leftarrow_{dfo} id'_t$ where id_{act} and id'_{act} are executed by tasks with ID id_t and id'_t , respectively.
- $id_{blk} \leftarrow_{tmo} id'_{blk}$ where id_{act} and id'_{act} are executed within atomic blocks id_{blk} and id'_{blk} , respectively.

Data Races: Two memory accesses are called to be conflicting if they refer to the same shared memory address and at least one of them is a write access. Similar to most of the happens-before based race detection mechanisms, we define a race condition as two conflicting accesses not ordered by the happens-before relationship, i.e., not separated by proper synchronization operations.

We say that action id_{act} is racing with action id'_{act} , if and only if the following conditions are met;

- at least one of the actions, id_{act} and id'_{act} , is a *write* access,
- there is no *hybrid happens-before* between actions which means neither $id_{act} \leftarrow_{hbb} id'_{act}$ nor $id'_{act} \leftarrow_{hbb} id_{act}$ holds.

8.3 Race Detection Mechanism

In this section, we will formally define our race detection approach using the *hybrid happens-before* relation we described in the previous section. We will first describe our dynamically collected trace components and corresponding interpretation rules, and finally describe our detection mechanism.

8.3.1 Modeling and Analysis of Traces

The set of memory locations (“locations”) manipulated by a hybrid program is denoted by $Locs$, and the set of values they can take by Val . These variables can take on values of types such as integers or floating-point numbers, strings and object references. The value of an uninitialized locations is $\perp_{Val}$. We treat each field of each object as a unique, separate ordinary location for race-detection purposes.

For identifying the corresponding task and atomic block ID, id_t and id_{blk} , that accessed to a memory location, we define a set of IDs that consists of task and block IDs as (id_t, id_{blk}) pairs. For simplicity, we will use id_{hyb} for describing a pair in IDS . In case of access from a non-atomic block, the value for block ID is $\perp_{ID}$.

The memory map: The value of each location v at each point in the execution is represented by a memory map named $\mathcal{V}$. $\mathcal{V}(v)$ consists of set of pairs of the form (vl, id_{hyb}) where $vl \in Val$ and $id_{hyb} \in IDs$. Intuitively, $(vl, id_{hyb}) \in \mathcal{V}(v)$ means that in this trace or a re-ordering of this trace that has the hybrid happens-before relation, v may have the value vl written to it by the ID id_{hyb} . When processing a trace entry with the ID id_{hyb} is a write access to v , we make critical use of the hybrid happens-before relationship computed up to that point. All entries in the memory map $\mathcal{V}(v)$ by trace entries with ID id'_{hyb} that happened before id_{hyb} are removed, representing the fact that they have been overwritten by a later access that happens after them. Other entries in $\mathcal{V}(v)$ are there due to “concurrent” execution blocks and are therefore not removed.

Similarly, the last reader of each location v at each point in the execution is represented by a map named $\mathcal{R}$. $\mathcal{R}(v)$ consists of a set of identifiers, IDs_{reader} , where

$IDS_{reader} \subseteq IDS$. Intuitively, $id_{hyb} \in \mathcal{R}(v)$ means that in this trace or a re-ordering of this trace that has the hybrid happens-before relation, v last read by the ID pair id_{hyb} . When processing a trace entry which is a read access by the ID id_{hyb} , we remove the any identifiers with ID id'_{hyb} that happened before id_{hyb} are removed, resulting only the “concurrent” reader identifiers to be left in the set. $\langle \lambda_0, \lambda_1, \lambda_2, \dots, \lambda_n \rangle$ of the following types:

8.3.2 Defining Traces

Formally, a trace is a finite sequence of trace entries $\langle \lambda_0, \lambda_1, \lambda_2, \dots, \lambda_n \rangle$ of the following types:

- **Memory Location accesses:** $\lambda = memWrite(v, vl, id_{blk}, id_t)$ denotes the writing of the value vl to the memory location v by the atomic block id_{blk} within task id_t . $\lambda = memRead(v, vl, id_{blk}, id_t)$ denotes the reading of the value vl from the memory location v by the atomic block id_{blk} within task id_t .
- **Synchronization operations:** $\lambda = passToken(id_t, id'_t)$ denotes the propagation of a token from task with ID id_t to task with ID id'_t . $\lambda = beginAtomic(id_t, id_{blk})$ denotes that the task with ID id_t began executing an atomic block with ID id_{blk} . $\lambda = endAtomic(id_t, id_{blk})$ denotes that the task with ID id_t finished executing an atomic block with ID id_{blk} .

8.3.3 Interpreting Traces

Given a trace $Trace = \langle \lambda_0, \lambda_1, \lambda_2, \dots, \lambda_n \rangle$, our race detection algorithm analyzes it by processing it sequentially, one log entry at a time. The algorithm maintains analysis state represented by the tuple $\Sigma = (\mathcal{V}, \mathcal{R}, DF, TM, id_{blk})$. Here, $\mathcal{V}$ is the memory map and $\mathcal{R}$ is the *reader IDS* map. DF is the dataflow order which is a partial ordered over *task IDs*. Whenever new elements are added to DF by processing a trace rule, the transitive closure of the relation is taken to obtain the resultant DF . TM is the transactional order that is a total order over *atomic block IDs*. Whenever

MEM-WRITE

$$\begin{array}{c}
\lambda = \text{memWrite}(v, vl, id_{blk}, id_t) \\
\mathcal{V}' = \mathcal{V}[(v) := \mathcal{V}(v) \cup \{(vl, (id_t, id_{blk}))\} \setminus \{(vl', (id'_t, id'_{blk})) \mid id'_{blk} \leftarrow_{tmo} id_{blk} \\
\text{or } id'_t \leftarrow_{df_o} id_t \text{ or } (id'_t = id_t \text{ and } id'_{blk} = id_{blk})\}] \\
\hline
(\mathcal{V}, \mathcal{R}, DF, TM, id_{blk}) \xrightarrow{\lambda} (\mathcal{V}', \mathcal{R}, DF, TM, id_{blk})
\end{array}$$

MEM-READ

$$\begin{array}{c}
\lambda = \text{memRead}(v, vl, id_{blk}, id_t) \\
\mathcal{R}' = \mathcal{R}[(v) := \mathcal{R}(v) \cup \{(id_t, id_{blk})\} \setminus \{(id'_t, id'_{blk}) \mid id'_{blk} \leftarrow_{tmo} id_{blk} \\
\text{or } id'_t \leftarrow_{df_o} id_t \text{ or } (id'_t = id_t \text{ and } id'_{blk} = id_{blk})\}] \\
\hline
(\mathcal{V}, \mathcal{R}, DF, TM, id_{blk}) \xrightarrow{\lambda} (\mathcal{V}, \mathcal{R}', DF, TM, id_{blk})
\end{array}$$

PASS-TOKEN

$$\begin{array}{c}
\lambda = \text{passToken}(id_{from}, id_{to}) \quad DF' = \text{TransClose}(DF \cup \{(id_{from}, id_{to})\}) \\
\hline
(\mathcal{V}, \mathcal{R}, DF, TM, id_{blk}) \xrightarrow{\lambda} (\mathcal{V}, \mathcal{R}, DF', TM, id_{blk})
\end{array}$$

BEGIN-ATOMIC

$$\begin{array}{c}
\lambda = \text{beginAtomic}(id_t, id_{newBlk}) \quad TM' = \text{TransClose}(TM \cup \{(id_{blk}, id_{newBlk})\}) \\
\hline
(\mathcal{V}, \mathcal{R}, DF, TM, id_{blk}) \xrightarrow{\lambda} (\mathcal{V}, \mathcal{R}, DF, TM', id_{blk})
\end{array}$$

END-ATOMIC

$$\begin{array}{c}
\lambda = \text{endAtomic}(id_t, id_{newBlk}) \quad id'_{blk} = id_{newBlk} \\
\hline
(\mathcal{V}, \mathcal{R}, DF, TM, id_{blk}) \xrightarrow{\lambda} (\mathcal{V}, \mathcal{R}, DF, TM, id'_{blk})
\end{array}$$

Figure 8.2: Hybrid race detection trace analysis rules.

new elements are added to TM by processing a trace rule, the transitive closure of the relation is taken to obtain the resultant TM . Finally, id_{blk} is the ID of the last finished atomic block processed by our algorithm, or $\perp$ if no such block exists.

Rules for updating the analysis state $\Sigma = (\mathcal{V}, \mathcal{R}, DF, TM, id_{blk})$ are given in Figure 8.2 and explained below.

Memory accesses: MEM-WRITE handles the write action where memory location, v , is update to value, vl , by the atomic block ID, id_{blk} , within execution of task ID, id_t . We declare a write-write race if $|Val(v)| > 1$ indicating the potential for

non-determinism on the memory location. Also we declare a read-write race if there exists an ID pair, $(id'_t, id'_{blk}) \in Readers(v)$ such that $(id'_t, id'_{blk}) \not\prec_{hbb}(id_t, id_{blk})$, indicating that there is a non-deterministic read action due to a *concurrent* write action. The rule computes $\mathcal{V}'(v)$ from $\mathcal{V}(v)$ by adding the pair $(vl, (id_t, id_{blk}))$ and removing all pairs $(vl', (id'_t, id'_{blk}))$ such that $id'_{blk} \leftarrow_{tmo} id_{blk}$ and $id'_t \leftarrow_{dfo} id_t$. MEM-READ handles the read action where the value, vl , is read at memory location, v , by the atomic block ID, id_{blk} , within task ID, id_t . We declare a write-read race if there exists a pair $(vl, (id'_t, id'_{blk}))$ where $(id'_t, id'_{blk}) \not\prec_{hbb}(id_t, id_{blk})$, indicating that there is a non-deterministic read action due to an earlier *concurrent* write action. The rule updates the $\mathcal{R}$ set by adding (id_t, id_{blk}) and removing all pairs (id'_t, id'_{blk}) such that $id'_{blk} \leftarrow_{tmo} id_{blk}$ and $id'_t \leftarrow_{dfo} id_t$.

This particular handling of write and read accesses in the trace is key to our symbolic execution algorithm. Write accesses overwrite *only* the values that have been written by entries that have a happens-before relationship with them, but the values written by potentially concurrent entries are left intact. On the other hand, we also keep the identifiers of the latest *concurrent* read actions on memory locations, which later checked against write actions for detection possible non-deterministic read actions.

Synchronization operations: PASS-TOKEN log entry process indicates the token pass from task with ID id_{from} to task with ID id_{to} and updates the *dataflow order*, DF , by adding the pair (id_{from}, id_{to}) . The rules ensure that the dataflow (input-output) relations in the executions coincides with the *dataflow order*. BEGIN-ATOMIC rule handles the start of a new atomic block with ID id_{newBlk} within execution of task with ID id_t . The corresponding update is made on the *transaction order*, TM , by adding the pair (id_{blk}, id_{newBlk}) . Doing so ensures that the total order on the execution order of the atomic blocks is preserved. Finally, END-ATOMIC handles the end of an atomic block with ID, id_{newBlk} , within execution of task with ID id_t , which only updates the last finished atomic block ID, $atomicBlkId$, with id_{newBlk} . This rule used for ordering the atomic blocks with the next start of the new atomic block.

8.3.4 Detecting Races

Our race detection algorithm detects both types of data races, i.e. *write-write* and *write-read / read-write*. We say $\mathcal{V}(v)$ has *write-write race on a memory location v* when the memory map contains at least two different values for v , i.e., when there are pairs $(vl, (id_t, id_{blk}))$ and $(vl', (id'_t, id'_{blk}))$ in $\mathcal{V}(v)$, such that $vl \neq vl'$ and $id_t \neq id'_t$ and $id_{blk} \neq id'_{blk}$. These races are detected while evaluating the MEM-WRITE rule if the memory map $\mathcal{V}(v)$ computed by the rule has non-determinism potential on v .

Our algorithm will declare a *read-write race on memory location v* while processing a write action with pair $(vl, (id_t, id_{blk}))$, if the readers map, $\mathcal{R}(v)$, contains an ID that is *concurrent* to the current writer, i.e. there exists a reader with ID $(id'_t, id'_{blk}) \in \mathcal{R}(v)$ such that $(id'_t, id'_{blk}) \not\prec_{hbb}(id_t, id_{blk})$. Similarly, our algorithm will detect a *write-read race on memory location v* while processing a read action with pair $(vl, (id_t, id_{blk}))$, if the memory map, $\mathcal{V}(v)$, contains a value that is written by a *concurrent* write entry, i.e. there exists a written value pair $(vl', (id'_t, id'_{blk})) \in \mathcal{V}(v)$ such that $(id'_t, id'_{blk}) \not\prec_{hbb}(id_t, id_{blk})$.

Consider a prefix of the trace in a “quiet” state such that, at the end of the prefix, no task execution is in progress. Suppose that our algorithm has signaled non-determinism potential on a memory location v while processing a trace entry λ within the last task ID id_t in this prefix. Then a different re-ordering of the tasks while leaving the happens-before relation in the prefix intact may result in a different final value for v , as explained next.

Let $(vl, (id_t, id_{blk}))$ and $(vl', (id'_t, id'_{blk}))$ be in $Val(v)$ such that $id'_{blk} \neq id_{blk}$, $id'_t \neq id_t$ and $vl \neq vl'$. It must be the case that while both (id'_t, id'_{blk}) and (id_t, id_{blk}) wrote to v , these two execution blocks are not ordered by the happens-before relationship. Otherwise, either $(vl, (id_t, id_{blk}))$ or $(vl', (id'_t, id'_{blk}))$ would have been removed from $Val(v)$, according to the update rules of our algorithm and how they use the happens-before relation in memory location updates. Therefore, it is possible to modify the execution by delaying the execution of the block with ID (id'_t, id'_{blk}) until *after* the execution of the block with ID (id_t, id_{blk}) . In this case, the final value of the lo-

cation v would be different at the final state of the newly-obtained execution. This points to a potentially different result produced *purely* as a result of XHR callback scheduling non-determinism. A similar case will be also present in the case of *read-write/write-read races* where the read value may have been different by delaying a *concurrent* read or write action.

8.4 Race Detection Implementation

In this section, we elaborate on tool implementation of our race detection algorithm for hybrid programming models combining dataflow constructs with shared memory programming models. In general, our race detection prototype composed of two main parts: trace parser and race detection core where our detection technique resides. Race detection core provides simple software hooks for extending the trace parser to be used for different hybrid programming models and trace structure. Our race detection prototype can be used for detecting races on other hybrid programming model with simple extensions, one has to implement trace collecting instrumentation for these models as well as a parser with our race detection hooks.

8.4.1 Race Detection Tool Implementation

We implemented a prototype of our race detection technique for hybrid programs using C# programming language. The prototype composed of detection core and parsing interfaces to make it easily extensible for other programming models. The detection core contains a *memory map* representing the accessed memory locations in the executed program and a *happens-before graph* representing the partial ordering of the executed tasks. At each memory access (read/write) the memory map is updated according to the formal rules define in Section 8.3. We separated the parsing from the detection core as a design decision to make the tool extensible, one can easily implement its own parser using our hooks to the race detection mechanism while leaving the detection core intact.

We instrumented Atomic DataFlow (ADF) runtime for collecting memory and

```

1  0 RD 0x6465ec 0
2  0 RD 0x6465f0 32768
3  0 RD 0x7ffdba964560 1048577
4  0 RD 0x7ff474b28012 32
5  0 WR 0x6465ec 32771
6  ...
7  1 RD 0x6465ec 32771
8  1 RD 0x7ff474b30013 32
9  1 RD 0x6465ec 32771
10 1 RD 0x6465f0 32768
11 1 WR 0x6465ec 65540
12 ...
13 2 WR 0x7ff46c001e80 0
14 2 RD 0x7ff474b28013 104
15 2 RD 0x7ff474b28017 32
16 2 RD 0x7ff474b28014 97
17 2 WR 0x7ff46c001be1 97

```

Figure 8.3: Memory access trace sample from instrumented Atomic DataFlow (ADF) runtime.

synchronization accesses information (formally described in Section 8.3) conducted by the program-under-test. We used our single threaded testing scheduler (described in Chapter 7) for recording the task executions in a *run-to-completion* manner meaning that the execution is not preemptively interrupted by other task executions. Our instrumented runtime generates two separate trace files; one with *happens before* relations between executed tasks and one with each memory access information (read/write accesses) from each tasks. Figure 8.3 shows a simple memory access trace example, each line in the trace represents the following informations: task identifiers executing the access, operation type (Read (RD) or Write (WR)), memory address in hex representation (i.e. 0x6465ec) and corresponding value read or write. On the other hand, *happens before* trace is composed of entries with pairs of task identifiers representing a *hybrid happens before* edge.

Figure 8.4 illustrates general flow of our race detection tool over an overview of the system structure which is later explained in details. The execution steps of our

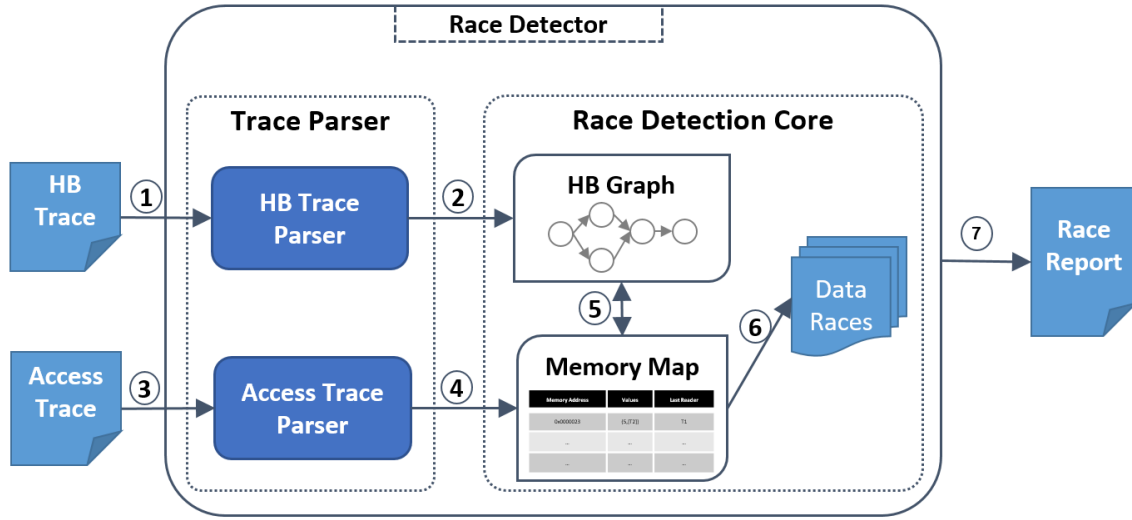


Figure 8.4: Race detection implementation overview.

detection tool is listed below:

1. *HB trace parser* processed the trace containing the happens-before relations line-by-line;
2. Each line represents a *happens-before relation* between two tasks which is stored as a graph using `AddHBEdge` method;
3. After finishing constructing the happens before graph by parsing *HB trace*, our tool starts parsing the memory access trace line-by-line;
4. Each line in *access trace* contains memory accessing action (read-write) which are applied to the memory map by using `WriteValue` and `ReadValue` methods, at each access we check for possible races according the rules formally described in Figure 8.2;
5. For each memory access *happens-before relation* information is gathered from the *HB graph*;

6. If a race is detected, a data race is recorded to the detected races;
7. After finishing parsing the whole *access trace* all detected races are reported.

Race Detection Core: Our race detection technique is based on an abstraction representation of the actual memory map where each concurrent memory access is represented as merged values. With each memory access the corresponding memory location is checked for possible races with the use of *happens-before relations* between executing blocks. Each *read/write* operation exercised in the trace is simulated over a memory map by applying the corresponding formal rules using the *happens-before relation* represented as a graph. For ease of extendibility, our race detection core provides software hooks (API methods) for each memory accesses and synchronization accesses for updating the memory map and the happens before graph:

- **AddHBEdge(string fromBlockID, string toBlockID):** Method for updating the *happens-before relation* graph with a new edge which corresponds to having a *hybrid happened-before* edge between the input parameters: $\text{fromBlockID} \leftarrow_{hbb} \text{toBlockID}$.
- **WriteValue(string memID, string writerID, string memValue):** Method for exercising a *write* operation that takes the memory identifier, writer's block identifier and the written value as a parameter and updates the memory map accordingly. During the update operation a race check is applied for detecting *write-write* and *read-write* races on the memory address using the *happens before relations*.
- **ReadValue(string memID, string readerID):** Method for applying a *read* operation that takes the memory identifier and reader's block identifier as a parameter and updates the memory map with the latest reader identifier. Similar to the write operation, a race check is done for detecting *write-read* races.

Trace Parser: In our race detector prototype, we separated the trace parsers from race detection core for making it easy to extend for different traces and hybrid programming models. Although implemented parsers are specific to our trace structure, one can extend these parsers for processing different traces collected from different hybrid models by using the provided software hooks by the race detection core. In the current setting, we implement two parsers for processing two types of traces:

- **Access trace** containing each *read/write* access exercised by each executing task. Each entry in the trace consists of task identifier, memory access type (read-write), memory address and corresponding value. Each line of the trace is parsed and applied on the *memory map* by using the `WriteValue` and `ReadValue` methods provided by the race detection core. And as the trace is collected via a single threaded *run-to-completion* scheduler, the accesses by each tasks are well separated in the trace.
- **Happens-Before (HB) trace** containing the *happens-before relation* between each executing task. Each entry line in the trace consists of two tasks identifiers representing the *hybrid happens before* edge, $\leftarrow_{hbb}$, between these tasks. Each line is parsed and inserted into the *happens before graph* by using `AddHBEdge` method provided by the race detection core. As our *happens before* relation is defined over coarse grain constructs (i.e. tasks), our graph representation does not grow a lot but one can easily employ garbage collection techniques by removing the processed tasks (parsed by access trace parser) from the graph.

8.5 Preliminary Experiments

In this section, we describe our preliminary experiments on evaluating our race detection technique for hybrid programming models using dataflow constructs along shared memory programming model.

Our micro benchmark is composed of three well known HPC applications from DaSH [33] benchmark that is implemented using ADF programming model:

	Trace Stats		Analysis Results	
Application Name	Trace Size (MB)	Number of Tasks	Analysis Time (sec)	Found Races
Dense Matrix Multiplication	46.10	1198	10.53	0
Graphical Models	14.30	220	14.54	1
Combinatorial Logic	0.04	50	0.03	1

Table 8.1: Preliminary analysis results for hybrid applications.

Dense Matrix Multiplication performs matrix multiplication on dense matrices with the help of individual worker tasks that concurrently computes the partial results for different portions of the matrix.

Graphical Models performs the Viterbi algorithm [30] for the Hidden Markov model to find the most likely sequence of states that can explain each observation provided by the input data.

Combinatorial Logic performs simple bit-level operations on very large amount of data using tasks parallelism for achieving high throughput for different applications like CRC calculation or data mining.

For evaluating the performance of our race detection tool, we executed these applications using instrumented ADF runtime using single threaded testing scheduler to collect execution traces. These traces contain the relevant memory access information for each task and their corresponding input-output dependency information.

Table 8.1 shows the initial results from our race detection tool along with some statistical information on the analysis. We were able to detect and manually verify two race conditions resulting in non-deterministic output depending on the ordering of the concurrently running tasks.

8.6 Conclusion

In this chapter, we presented our race detection technique for hybrid programming models that combine dataflow construct with shared memory programming models. Our detection technique is based on *static* analysis of an execution trace that is collected *dynamically* from a single run of the application. The key idea behind our static analysis technique is the observation that effects of different orderings of the executed tasks can be obtained by merging the updates on share memory locations. Our analysis efficiently reconstruct a memory map of the observed execution that encodes the effects of different possible schedules.

We started by explaining the available actions that a hybrid application can take. As our detection technique is based on *happens before* relation between concurrently executing entities, we start by defining a *modular happens before* relation that can be extended easily for different flavors of hybrid programming models. The key observation, we have is that while using dataflow constructs along with different shared memory synchronization mechanisms (i.e. locks, atomic blocks), we can also represent a *hybrid happens before* relation definition reusing the previous definitions for corresponding synchronization mechanisms. Here, we describe our *hybrid happens before* relation based on Atomic DataFlow (ADF) programming model, which uses transactional memory (TM) for handling concurrent updates to the shared memory, by combining two ordering definitions: *dataflow order* and *transactional order*.

Later, we formally describe our race detection technique using the *hybrid happens before* relation over the recorded trace actions by governing the formal trace analysis rules. Finally, we described our implementation effort for our generic race detection prototype employing our race detection technique. We build our prototype extendible by separating the trace parsing and race detection into different modules. The race detection core module provides software hooks for implementing new parser modules making it easier to be used for different programming models and trace structures. Later, we conducted some preliminary experiments for evaluating our detection tool on our micro benchmark of HPC applications.

Chapter 9

FUTURE WORK AND CONCLUSION

9.1 *Limitations and Future Work*

In this section, we discuss current limitation of our race detection techniques along with future directions for addressing these limitations.

9.1.1 *Limitations of Race Detection for JavaScript Applications*

JavaScript programming language is constantly evolving with each new specification [21, 22]. Currently, our race detection prototype is capable of detecting races caused by asynchronous server requests (`XMLHttpRequest`'s), user interactions (button clicks) and `Promise` constructs but with each specification new asynchronous programming constructs are introduced such as `async-await` [22]. However, it is possible to extend our detection technique and tool by defining *happens before* relation between new asynchronous constructs and adding necessary trace analysis rules.

Even though our evaluation showed small numbers of *false positives*, reported *false positives* can be classified as a limitation of our detection technique. As discussed in Section 5.3, the main cause of *false positives* is the commuting accesses to the shared state. Currently, our detection mechanism sacrifices precision in favor of being lightweight which is more desired in web development but additional analysis can be practiced over the reported results for increasing the precision.

Another limitation of our prototype detection tool is the support for other web browsers. Although we focused our effort on instrumenting Firefox browser for trace collection, our instrumentation is quite sparse and we believe can be easily migrated to other open-source browsers such as Chromium or server side JavaScript engines like Node.JS.

The following list summarizes potential future research directions on race detection in asynchronous JavaScript applications:

- Both our prototype race detection and Firefox instrumentation are developed as a research tool, hence, there is a lot of room for improving the tools scalability and performance.
- Our current implementation focused on data races caused by asynchronous server requests (`XMLHttpRequest`), user interactions (button clicks) and `Promise` constructs. New asynchronous constructs such as `async-await` can be also added as a source of possible data races.
- We instrumented Firefox web browser for collecting execution traces, one improvement can be extending the instrumentation effort to different JavaScript engines such as Chromium web browser and server side JavaScript engine Node.JS.
- To address the issues related to commuting accessing being *false positives*, additional commutativity checking analysis can be cooperated with the current detection mechanism. Current version avoids these kind of costly post analysis procedures in favor of being lightweight.
- Finally, recent advances in the type systems for JavaScript programming language [25, 68, 101], introduces new directions for detecting asynchronous programming anti-patterns [10, 69, 55] using type checkers. We think that having such static time checks for anti-patterns at developers disposal will help avoiding unintended non-determinism in asynchronous JavaScript applications.

9.1.2 Limitations of Race Detection for Hybrid Applications

Hybrid programming models combining dataflow constructs with shared memory constructs became widely popular with the research proposals [?, 78, 27, 80] and commercial counterparts [7, 4, 3]. Even-though our race detection technique is applicable to all

hybrid programs, main focus of our prototype tool is “Atomic DataFlow (ADF)” [34] which has relatively small user base. Hence, the benchmark space for evaluating our race detection prototype is limited. As an ongoing work, we are working on evaluating our prototype with applications from DWARF benchmark [9] implemented using ADF programming model [33].

Another limitation of our current instrumentation framework is the trace sizes. As we have to record each memory access as well as the synchronization operations and given the long running nature of scientific applications, the trace size can get quite high. Hence, an optimized representation or additional archiving capabilities is required for long running applications.

The following list summarizes potential future research directions on race detection in hybrid applications:

- Similar to our race detection for asynchronous JavaScript application, our prototypes are developed for research purposes and can be improved a lot for scalability and performance.
- One possible improvement on the performance of the analysis can be employing parallelization methods (i.e. MapReduce [20]) for trace parsing.
- Our current implementation of the instrumentation is based on Atomic DataFlow (ADF) programming model which has a limited user base. One future direction can be extending the instrumentation for more popular hybrid programming model such as OmpSs [78].

9.2 Conclusion

Asynchronous programming models has become ubiquitous in concurrent application development for the modern computation platforms, i.e. mobile, web and desktop applications. Similar to all concurrent programming models, writing *correct* programs using asynchronous constructs is hard. Use of asynchronous constructs, i.e.

callbacks, can invert the control flow as the execution order of these event handlers are unknown prior to actual execution. This may result in subtle concurrency bugs that can have a big impact on the development cycle as they are notoriously hard to detect, reproduce, and fix. Data races have been considered as the main source of unintended non-determinism in concurrent programs and seen as a symptom of a bigger concurrency bug. Although race detection in legacy concurrent programming models have attracted a great deal of attention from the research communities and commercial counterparts, there is a critical need for race detection techniques for modern asynchronous programs.

In this thesis, we address the need for race detection techniques in modern asynchronous applications. We propose a novel lightweight race detection techniques for asynchronous programming models which attempts to combine advantages of both static and dynamic analysis. We executed the application code only *once*, yet we explore *multiple* execution order with the help of our static analysis approach. The key observation is that our analysis technique encodes the effects of concurrent updates on the shared state by merging the values as opposed to separately considering different orderings. One way to see our approach is that we explore *neighboring* schedules for a particular runtime execution by analyzing possible re-orderings of the concurrent entities with a single pass over the observed execution.

As the first part of our study, we explored JavaScript web applications using asynchronous constructs for user interaction and server requests. First, we conducted empirical studies for identifying the real damages of data races in the web applications by devising randomized and systematic exploration of server requests (`XMLHttpRequests`) orderings. Our main focus was to classify *harmful races* as opposed to identifying all data races to be as important. As opposed to other types of applications, web applications have a more forgiving nature for data races on actual memory, since their effects are ephemeral. This is because these races only result in invisible to the user portions of the program state, or, at worst, UI glitches that are either unnoticed by the user, or disappear if the user reloads the page. In the light of our empirical study, we

suggested that a more useful way to think about data races on the web is by focusing on *persistent state*, such as client-local cookies, `localStorage` and `sessionStorage` mechanisms, as well as server-based side efforts.

For evaluating our race detection technique, we implemented a race detection tool along with a instrumented Firefox web browser. Later we conducted an extensive web site analysis for finding ones with heavy use of asynchronous constructs, i.e. server requests through `XMLHttpRequest`s, and used these for our experiments. In our experiments on 26 sites, we demonstrate that benign races are considerably more common than harmful ones. Finally, we conducted an exploratory study on the reported race related bugs on JavaScript libraries that uses new asynchronous constructs: `Promise`s. This study showed that it is still common to have race conditions even with the better abstractions for asynchronous programming in JavaScript web applications.

In the second part of the thesis, we worked on hybrid programming models that combines dataflow constructs with share memory programming model. These hybrid programming models provide programmers with dataflow abstractions for defining tasks as the main execution unit with corresponding data dependencies. Contrary to the pure dataflow model which assumes side-effect free execution of the tasks, these models allow tasks to share the data using some form of thread synchronization, such as locks or transactional memory (TM). First, we devised a randomized exploration technique with probabilistic guarantees for exploring possible execution orders of the tasks composing the application. Later, we devised a modular race detection technique which can be used for detecting data races in hybrid programming models. We presented a modular representation of execution orders imposed by different programming models, i.e. dataflow and shared memory, which can be used for extending the detection mechanism for different type of shared memory synchronization systems, i.e. lock-based, transactional memory.

Finally, we implemented our detection mechanism as a generic tool prototype that can be extended for different type of hybrid programming models where programmers are provided with different synchronization mechanism (i.e. locks, atomic blocks).

Thesis Contributions:

Part I - Race Detection for JavaScript Web Applications

- We introduce the notion *observable races*, as determined by visual inspection of the web application and developed a dynamic analysis mechanism for systematically exploring all possible orderings of asynchronous calls in order to exercise potentially defective schedules.
- We propose a new view of benign and harmful data races in JavaScript web applications, and argue that harmful races should be the primary focus of analysis tools, due to them affecting the persistent client- or server-side state of the applications.
- We propose a lightweight exploration algorithm for finding data races in runtime traces of JavaScript programs. A key advantage for the scalability of our approach is that it does not require multiple program runs and can operate on the basis of a single execution via use of static analysis over the recorded execution trace.
- We analyze the presence of races in modern web applications, both in the context of micro-benchmarks collected from related work and various bug reports, and macro-benchmarks that include widely-used web sites. We find and investigate a total of 19 harmful and 621 benign races in 26 web sites, with only 2 observed false positives.

Part II - Race Detection for Applications using Hybrid Programming Models

- We present a randomized exploration tool for characterizing and exploring behaviors of programs written using hybrid programming models combining dataflow constructs with shared memory programming model.

- We formalize a modular *happens-before* relation for hybrid programming models combining dataflow constructs with shared memory programming model. The formalization can be easily extended for different hybrid programming models.
- We propose a lightweight race detection mechanism for applications using hybrid programming models. Our detection mechanism attempts to combine the advantages of static and dynamic analysis by recording a runtime execution trace and later doing an efficient static analysis for exploring neighboring schedules.
- We implement a generic race detector prototype that can be easily extended for different hybrid programming models. We provide software hooks for implementing new parser modules in order to ease implementation effort for different programming models and trace structures.

BIBLIOGRAPHY

- [1] Google: Android operating system. http://en.wikipedia.org/wiki/Android_%28operating_system%29.
- [2] Html5. <http://en.wikipedia.org/wiki/HTML5>.
- [3] Intel: Intel threading building blocks - flow graph. http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm.
- [4] Microsoft: Task parallel library - dataflow. <http://msdn.microsoft.com/en-us/library/hh228603.aspx>.
- [5] Microsoft: Windows 8 operating system. http://en.wikipedia.org/wiki/Windows_8.
- [6] Microsoft: Windows mobile operating system. http://en.wikipedia.org/wiki/Windows_Mobile.
- [7] Openmp: Openmp 4.0 specification. <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>.
- [8] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05*, pages 428–452, Berlin, Heidelberg, 2005. Springer-Verlag.
- [9] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiawicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A

- view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, Oct. 2009.
- [10] D. Atchley. Es6 promise patterns and anti-patterns. www.datchley.name/promise-patterns-anti-patterns/.
- [11] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Producing scheduling that causes concurrent programs to fail. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '06, pages 37–40, New York, NY, USA, 2006. ACM.
- [12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.
- [13] A. Boduch. *JavaScript Concurrency*. Packt Publishing, 2015.
- [14] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM.
- [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] J.-D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented pro-

- grams. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 258–269, New York, NY, USA, 2002. ACM.
- [17] M. Christiaens and K. D. Bosschere. Trade: Data race detection for java. In *Proceedings of the International Conference on Computational Science-Part II*, ICCS '01, pages 761–770, London, UK, UK, 2001. Springer-Verlag.
- [18] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for Javascript. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM.
- [19] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: A web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 748–759, New York, NY, USA, 2012. ACM.
- [20] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
- [21] ECMA. ECMAScript 6th edition language specification. <http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf>, 2015.
- [22] ECMA. ECMAScript 7th edition language specification. <http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf>, 2016.
- [23] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 245–255, New York, NY, USA, 2007. ACM.

- [24] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.
- [25] Facebook. Flow: a static type checker for javascript. <https://flowtype.org/>.
- [26] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 1–11, New York, NY, USA, 1997. ACM.
- [27] 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. Ilinckas, 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.
- [28] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM.
- [29] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. *SIGPLAN Not.*, 44(6):121–133, June 2009.
- [30] G. D. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, March 1973.
- [31] D. P. Friedman and D. S. Wise. The impact of applicative programming on mul-

- tiprocessing. *Bloomington: Indiana University, Computer Science Department*, 1976.
- [32] H. Fukuda and P. Leger. *Proposals for Modular Asynchronous Web Programming: Issues and Challenges*, pages 91–102. Springer International Publishing, Cham, 2015.
- [33] 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.
- [34] 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.
- [35] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.
- [36] L. Gong, M. Pradel, and K. Sen. Jitprof: Pinpointing jit-unfriendly javascript code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 357–368, New York, NY, USA, 2015. ACM.
- [37] L. Gong, M. Pradel, M. Sridharan, and K. Sen. Dlint: Dynamically checking bad coding practices in javascript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 94–105, New York, NY, USA, 2015. ACM.

- [38] Google. Chrome web browser. <https://www.google.com/chrome/>.
- [39] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th Conference on USENIX Security Symposium, SSYM'09*, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.
- [40] S. Guarnieri and B. Livshits. GULFSTREAM: Staged static analysis for streaming JavaScript applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development, WebApps'10*, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association.
- [41] A. Hidayet. Phantomjs. <http://phantomjs.org/quick-start.html>.
- [42] S. Hong, Y. Park, and M. Kim. Detecting concurrency errors in client-side javascript web applications. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 61–70, March 2014.
- [43] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 326–336, New York, NY, USA, 2014. ACM.
- [44] D. Jang and K.-M. Choe. Points-to analysis for javascript. In *Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09*, pages 1930–1937, New York, NY, USA, 2009. ACM.
- [45] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems,*

- Languages, and Applications*, OOPSLA 2015, pages 57–73, New York, NY, USA, 2015. ACM.
- [46] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 34–44, New York, NY, USA, 2012. ACM.
- [47] L. Jouanneau. Slimerjs. <http://docs.slimerjs.org/0.9/release-notes.html>.
- [48] K. Kambona, E. G. Boix, and W. De Meuter. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, DYLA '13, pages 3:1–3:9, New York, NY, USA, 2013. ACM.
- [49] 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.
- [50] E. Kiciman and B. Livshits. Ajaxscope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 17–30, New York, NY, USA, 2007. ACM.
- [51] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 443–457, Washington, DC, USA, 2012. IEEE Computer Society.
- [52] 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.

- [53] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
- [54] J. Larus and C. Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, July 2008.
- [55] N. Lawson. We have a problem with promises. <https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html>.
- [56] G. Li, E. Andreassen, and I. Ghosh. Symjs: Automatic symbolic testing of javascript web applications. In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE 2014, pages 449–459, 2014.
- [57] Y. Lin and D. Dig. Refactorings for android asynchronous programming. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 836–841, Nov 2015.
- [58] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.
- [59] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 499–509, New York, NY, USA, 2013. ACM.
- [60] M. Madsen, F. Tip, E. Andreassen, K. Sen, and A. Møller. Feedback-directed instrumentation for deployed javascript applications. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 899–910, New York, NY, USA, 2016. ACM.

- [61] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 316–325, New York, NY, USA, 2014. ACM.
- [62] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.
- [63] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 134–143, New York, NY, USA, 2009. ACM.
- [64] F. Mattern. Virtual time and global states of distributed systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*, pages 215–226. North-Holland, 1988.
- [65] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 24–33, New York, NY, USA, 1991. ACM.
- [66] A. M. Memon. An event-flow model of gui-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, Sept. 2007.
- [67] Microsoft. Microsoft edge web browser. <https://www.microsoft.com/en-us/windows/microsoft-edge/>.
- [68] Microsoft. Typescript: Javascript that scales. <https://www.typescriptlang.org/>.

- [69] C. Monkey. Promise anti-patterns. <http://taoofcode.net/promise-anti-patterns/>.
- [70] Mozilla. Mozilla firefox web browser. <https://www.mozilla.org/en-US/firefox/>.
- [71] Mozilla. Rhino javascript engine. <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino>.
- [72] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.
- [73] E. Mutlu, S. Tasiran, and B. Livshits. I know it when i see it: Observable races in javascript applications. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 1:1–1:7, New York, NY, USA, 2014. ACM.
- [74] 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.
- [75] 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.
- [76] Node.js. Node.js. <https://nodejs.org/en/>.
- [77] S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen. A study and toolkit for asynchronous programming in c#. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1117–1127, New York, NY, USA, 2014. ACM.

- [78] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151, Sept 2008.
- [79] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pages 251–262, New York, NY, USA, 2012. ACM.
- [80] 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.
- [81] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '03*, pages 179–190, New York, NY, USA, 2003. ACM.
- [82] E. Pozniansky and A. Schuster. Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, Mar. 2007.
- [83] S. Qadeer and S. Tasiran. Runtime verification of concurrency-specific correctness criteria. *Int. J. Softw. Tools Technol. Transf.*, 14(3):291–305, June 2012.
- [84] 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.
- [85] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: Comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of*

- the 2010 USENIX Conference on Web Application Development*, WebApps'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.
- [86] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 151–166, New York, NY, USA, 2013. ACM.
- [87] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 677–694, New York, NY, USA, 2011. ACM.
- [88] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag.
- [89] G. Richards, S. Lebesne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.
- [90] Samsung. JerryScript - a javascript engine for internet of things. <http://jerryscript.net/>.
- [91] 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.

- [92] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.
- [93] P. Saxena, S. Hanna, P. Poosankam, and D. X. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.
- [94] M. Schaefer, M. Sridharan, J. Dolby, , and F. Tip. Effective smart completion for JavaScript. Technical Report RC25359, IBM Research, Mar. 2013.
- [95] C. Seaton, D. Goodman, M. Lujn, and I. Watson. Applying dataflow and transactions to lee routing. In *In Proceedings of Fifth Workshop on Programmability Issues for Heterogeneous Multicores*, 2012.
- [96] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498. ACM, 2013.
- [97] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.
- [98] 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.
- [99] K. Simpson. *You Don't Know JS: Async & Performance*. O'Reilly Media, 2015.

- [100] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of javascript. In *Proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP'12*, pages 435–458, Berlin, Heidelberg, 2012. Springer-Verlag.
- [101] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in javascript. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, pages 425–437, New York, NY, USA, 2014. ACM.
- [102] P. Thiemann. Towards a type system for analyzing javascript programs. In *Proceedings of the 14th European Conference on Programming Languages and Systems, ESOP'05*, pages 408–422, Berlin, Heidelberg, 2005. Springer-Verlag.
- [103] 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.
- [104] C. von Praun and T. R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '01*, pages 70–82, New York, NY, USA, 2001. ACM.
- [105] 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, ESEC-FSE '07*, pages 205–214, New York, NY, USA, 2007. ACM.
- [106] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM*

- Symposium on Operating Systems Principles*, SOSP '05, pages 221–234, New York, NY, USA, 2005. ACM.
- [107] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 805–814, New York, NY, USA, 2011. ACM.

