Explicit-State Model Checking of Concurrent x86-64 Assembly

Abhijith Ananth Bharadwaj

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Freek Verbeek
Michael S. Hsiao

May 13, 2020
Blacksburg, Virginia

Keywords: *Partial Order Reduction, x86, Machine-Code Verification, Software Model Checking, Concurrency*

Explicit-State Model Checking of Concurrent x86-64 Assembly

Abhijith Ananth Bharadwaj

ABSTRACT

The thesis presents Xavier, a novel tool-set for model checking of concurrent x86-64 assembly programs, via Partial Order Reduction (POR). Xavier presents a realistic platform for systematically exploring and analyzing the state-space of concurrent x86 assembly programs, with the aim of detecting bugs via assertion failures in mainstream programs. Recently, a number of state-of-the-art model checking solutions have been introduced to efficiently explore the state-space of concurrent programs, using POR algorithms. However, such solutions are inefficient while analyzing stateful programming languages, such as the x86 assembly language, due to the solutions' higher level of abstraction. To this end, Xavier makes two contributions: i) a novel order-sensitivity based POR algorithm, that is applicable to concurrent x86 assembly, ii) an x86 machine-model that can accurately perform relaxed-consistency emulation of concurrent x86 assembly, without the need for any translations. We demonstrate the applicability of Xavier through an evaluation on several classical mutual-exclusion benchmarks and mainstream benchmarks from the Userspace Read-Copy-Update (URCU) concurrency library, where the benchmarks range from $250 - 3700$ lines of x86 assembly. The framework is the first that supports systematic model checking of concurrent x86 assembly programs, and the effectiveness of Xavier is demonstrated by reproducing a concurrency issue of threads accessing intermediate states in the URCU library, which stems from an assumption violation.

This work is supported in part by the US Office of Naval Research under grant N00014-17-1-2297.

Explicit-State Model Checking of Concurrent x86-64 Assembly

Abhijith Ananth Bharadwaj

GENERAL AUDIENCE ABSTRACT

Sound verification of multi-threaded programs necessitate a systematic analysis of program state-spaces that result from thread interactions. Consequently, model-checking [1], [2] has been one of the prominent methods used to tackle the verification of multi-threaded programs. However, existing model-checking solutions are inefficient while analyzing stateful programming languages, such as the x86 assembly language, due to the solutions' higher level of abstraction. Therefore, the thesis presents Xavier, a novel tool-set and a realistic platform for systematically exploring and analyzing the state-space of mainstream concurrent x86 assembly programs, with the aim of detecting bugs via assertion failures. To this end, Xavier makes two contributions: i) a novel order-sensitivity based Partial Order Reduction algorithm, which efficiently explores the state space of concurrent x86 assembly, ii) an x86 machine-model that can accurately emulate the execution of concurrent x86 assembly, without the need for any translations. We demonstrate the applicability of Xavier through an evaluation on several classical mutual-exclusion and mainstream benchmarks from the Userspace Read-Copy-Update (URCU) concurrency library, where the benchmarks range from $250 - 3700$ lines of x86 assembly. Moreover, we demonstrate the effectiveness of Xavier by reproducing a concurrency issue in the URCU library, which manifests as a result of an assumption violation.

This work is supported in part by the US Office of Naval Research under grant N00014-17-1-2297.

Dedication

To my parents, and my wife Poojashree.

Acknowledgments

At the foremost, I would like to express my sincere gratitude to my advisors Dr.Freek Verbeek and Prof.Binoy Ravindran for providing me a platform in their research group to explore my enthusiasm on research. I would also like to thank them for sharing their immense knowledge though their continuous guidance and support throughout the course of my degree. Besides my advisors, I would also like to thank Dr.Michael Hsiao for being an integral part of the thesis committee and for sharing his knowledge through insightful comments and questions.

I would like to thank my fellow labmates Jae-Won Jang, SengMing Yeoh, Md Syadus Sefat, Joshua Bockenek, Mincheol Sung, Yihan Pang, A K M Fazla Mehrab, Ian Roessle, Xiaoxin An, Balaji Arun, Ho-Ren Chuang and Cathlyn Stone from the Systems Software Research Group for all the insightful discussions and the good times we've had in the past two years.

I would also like to thank my previous wonderful colleagues M. Achutha KiranKumar V., Bindumadhava S. S. and Dr.Disha Puri for introducing me to Formal Verification and for inciting my interest in research.

Finally, I would like to thank the most important people in my life: my amazing parents Vijayalakshmi K M and Anantha Padmanabha M S and my wonderful wife Poojashree N S, for their patience and for supporting my aspirations at every step. My parents have always been the light of my life and I credit every bit of my enthusiasm to their willingness to support me. Likewise, my most loving wife has been the source of the unwavering strength that I needed to face the hardships in the pursuit of this degree. It beats me to know what have I done to deserve her as the better half of my life.

Contents

List of Figures

List of Tables

Explicit-State Model Checking of Concurrent x86-64 Assembly

Abhijith Ananth Bharadwaj

July 9, 2020

Chapter 1

Introduction

Verification of correct concurrent software is non-trivial, since concurrent programs can have additional sources of bugs compared to sequential programs. This is so, primarily due to the non-deterministic behavior of programs owing to interleaving thread interactions. Moreover, the manifestation of bugs in concurrent programs are often dependent on the order of thread execution [3] - such bugs are also termed as *Heisenbugs*. Hence, concurrency bugs are inherently more difficult to be reproduced by trivial *testing*, as systematic analysis is key to analyzing the combinatorial nature of thread interactions. Therefore, *formal analysis* is essential in proving correctness of concurrent programs.

The non-deterministic inter-leavings of thread interactions have long been considered as the predominant source of bugs in concurrent programs. At the very least, the sound formal analysis of a concurrent program requires a systematic exploration of the thread interactions. Hence *Stateless Model Checking* [1] has been the preferred method to verify concurrent programs. However, the combinatorial and non-deterministic nature of interleaved executions can cause an exponential increase in the number of states needed to be examined. Any model checker hoping to verify concurrent programs must systematically take into account the multiple possibilities of thread interactions. Hence, most modern model checkers targeting concurrent programs tend to primarily focus on mitigating the state-space explosion problem.

One of the predominant techniques used in the concurrency community to tackle the state-space explosion problem is *Partial Order Reduction* (POR). POR algorithms [4]–[6] are techniques used to efficiently explore the state-space of a concurrent program. POR algorithms define equivalence classes of traces, where the traces in a class are considered equivalent based on a pre-defined criterion. POR algorithms then reduce the portion of the program state-space explored by intelligently limiting redundant trace explorations from the equivalence classes. However, such techniques are predominantly applied to either the source-code

or a compiler Intermediate Representation (IR), such as LLVM or Java Bytecode, which are competent platforms to discern and analyze thread interactions.

However, apart from the non-deterministic nature of thread interleavings, there can be several other less considered and equally insidious external factors that can influence the execution of a concurrent program. The optimizations enforced by the compiler may produce assembly code with different behavior than the source-code, which can introduce bugs in the compiled program [7], [8]. The thread scheduling model of the Operating System (OS) and its implementation of concurrency primitives often perform run-time optimization by embedding assembly code in libraries. Additionally most modern processors, in view of achieving better performance, relax the memory consistency during execution through implementing the *Total Store Order* or the *Partial Store Order* memory models [9], [10], which can also contribute to the complexities in assembly code.

The aforementioned complexities can potentially contribute to behaviors that would not manifest at the source-code level. This is so, since most of these modifications and optimizations are effected at levels lower than the source-code. However, the resulting behaviours will be readily apparent in the assembly code. Hence verification of multi-threaded programs at a level higher than the assembly can result in admitting several of the aforementioned factors into the Trusted Code Base (TCB) of the verification effort [11]. In contrast, verification of multi-threaded programs at the assembly level can provide a tractable solution that accounts for the aforementioned external influences. Therefore the thesis aims to contribute a novel and an efficient tool-set to formally verify multi-threaded programs at the assembly level.

1.1 Motivation for Analysis of Assembly

Source-code verification has long been used to prove correctness over program behavior. The semantics of source-code is evidently at a higher level of abstraction than assembly. This makes verification at the source-code level appealing, as the effort required is significantly less compared to assembly verification. However, for several reasons, source-code verification may not always be a viable strategy, and verifying the assembly may be more advantageous. We will detail few such reasons:

1. *Compiler trustworthiness.* Verification at the source-code level requires the compiler and the programming language implementation to be admitted into the Trusted Code Base (TCB). Even with extensively tested compilers, compiler translations can still introduce bugs in the compiled program [7], [8], which can affect concurrent executions. The source-code optimizations such as instruction re-ordering, instruction deletions can potentially allow threads to access illegal intermediate states [12]. The different optimization strengths of the compiler can also expose new interactions between threads.

Therefore, correctness in program behavior at the source-code level may not translate to actual concurrent execution.

However, assembly verification does not require the compiler to be in the TCB. Assembly programs are downstream products of compiler translations, and can thereby witness the behaviors introduced by the compiler [11]. Therefore, verification of assembly programs can expose the concurrency issues introduced by the compiler.

2. *Assumptions in source-code verification.* Verification at the source-code level requires making assumptions over several external factors. For example, source-code verification requires trusting the implementation of external libraries. Multi-threading libraries often perform code optimizations by embedding or replacing code with assembly during compile-time. Hence such implementations are included in the TCB, as they cannot be exposed at the source-code level. Moreover, source-code verification also requires including implementation of the language semantics into the TCB. It is usually the underlying language semantics that provides support to concurrency constructs such as thread creations, memory barriers etc., which unless also verified, needs to be added to the TCB.

   However, assembly programs include both the optimizations by the libraries and the implementation of the language semantics. Hence,verifying assembly programs can account for such behaviors, thereby offering higher confidence over execution correctness.

3. *Low-level correctness.* Even with the availability of verified mainstream compilers [13], verification of low-level constructs such as memory isolation [14] and execution consistency cannot be guaranteed unless the low-level implementations of such constructs are modelled and verified. The source-code possesses an abstracted view of shared memory interactions, thereby necessitating admitting semantics of the underlying memory model to be in the TCB. Moreover, source code verification usually assumes sequential execution consistency. However, modern processors typically re-order instructions to relax the execution consistency (TSO, PSO, etc.). These behaviors are usually not visible at the source-code level.

   However, verification of such low-level constructs is more amenable at the assembly level. Assembly contains an explicit view of the memory and assembly programming makes less assumptions about these low-level constructs. Also, assembly programs interact directly with such language constructs, which necessitates precise modelling of such constructs for assembly program verification.

4. *Availability of source-code.* Most production-ready programs have software modules that are integrated at the machine-code. Moreover, the source-code for such modules

may be Intellectual Properties and not be available for verification, rendering source-code level verification impossible. In such cases, verifying the assembly will be more advantageous, as it does not rely on the presence of the source-code.

Therefore, in view of the above caveats, we maintain that verification of the assembly code is more advantageous than verifying source-code.

1.2 Overview of State-Of-The-Art

Concurrent verified compilers. Verified compilation of sequential code has a long history of contributions, notably by [13], [15]–[20]. However, the contributions on verified compilation of concurrent programs are comparatively minimal. The most notable work is this domain is by CompCertTSO [21], which details the verified compilation of a C-like concurrent programming language, designed to provide x86-TSO [10], [22] based relaxed consistency execution on x86 processors. Such work is important in proving that the intent of the program is preserved during the translation by the compiler. However, it is not the intent of such works to prove correctness over program behavior.

Translation validation. Translation validation has been another avenue that has been explored to verify assembly, where a relation is established between the source code and the compiled artifact. To this end, the contributions in [23], [24] are the most notable. The work in [23] proves correctness over assembly code by establishing a refinement relation between the compiled ARM assembly and the high-level C source code. However, even as the method shows excellent scalability, at the foremost it requires the source code. The work in [24] tackles assembly verification by establishing a refinement relation between the assembly and an abstract code defined in the paper, which does not necessitate the requirement of the source-code. However, both the works have been mainly focused on single threaded execution.

Partial Order Reduction. POR [4]–[6] methods have long been used in efficient model-checking of the correctness of concurrent programs. POR is a class of algorithms that defines heuristics for efficient state-space exploration of concurrent programs. More details are provided in Chapter 2. The state-of-the-art in POR algorithms have contributed towards being more efficient at exploring new states in a concurrent program [25]–[27] and also towards applying POR to different execution consistency models [28]–[31]. However, since the assembly language is an imperative language and due to the low-level nature of assembly, the state-of-the-art in POR is not directly applicable to assembly programs. We will provide more details in the following chapters.

1.3 Thesis Contribution

In order to perform formal analysis of multi-threaded x86 programs, the thesis contributes XAVIER, an explicit state model checker that consumes off-the-shelf x86 assembly. The rest of the thesis will refer to x86 assembly as simply the assembly. The aim of this model checker is to detect illegal states of threads by way of encountering assertion violations during execution.

In the view of the tool-set, the contributions in this thesis is twofold.

1. *We present a novel* Order-Sensitive *Dynamic POR algorithm, which efficiently exposes and explores the interactions between concurrent assembly threads.*

   The goal of the POR algorithm is to efficiently set up trace explorations that allow the tool-set to uncover the (possibly) several end-states of assembly program. To that end, the POR algorithm includes the following features.

   - The POR algorithm is not restrictive to a declarative programming model, and is thereby applicable to the assembly programming language.
   - The POR algorithm is order-sensitive, wherein the equivalence classes are sensitive to the order of selective actions in the trace. More details will be provided in Chapter 4 and 5.
   - The POR algorithm is dynamic, as the equivalence classes are defined on-the-fly.
   - Each trace exploration is a *stateful* simulation of the program that allows the tool-set to uncover new program states, wherein the safety properties can be evaluated (via assertions in the program). We call the simulations stateful, as every program state achieved during a simulation is stored for use by the POR algorithm to efficiently set-up new explorations.

2. *We present an* x86 machine model *that emulates the concurrent execution environment of an x86 processor in a relaxed consistency setting.*

   The goal of the *machine model* is to provide a realistic simulating environment for executing concurrent x86-64 assembly. To that end, the machine model design incorporates the following features

   - The machine model is designed to consume off-the-shelf x86 assembly programs.
   - The machine model draws inspiration from [32]–[34] in providing interpreter style semantics to several instructions, including concurrency instructions, of the Intel IA-32e 64-bit ISA (x86-64) [35].

- The machine model incorporates a byte-addressable memory model, as well as semantics for the x86-TSO Relaxed Memory Model (RMM).
- The machine model provides semantics to several APIs from the POSIX thread library, and semantics to several system calls sourced from the Linux kernel implementations.

In view of the above design features, the machine model is capable of emulating execution of concurrent assembly programs extracted from C programs that utilize the POSIX thread library for multi-threading.

The combination of the novel POR algorithm with the machine model presents several advantages:

- Inclusion of a *sound* POR algorithm guarantees the exploration of all unique end-states of the program, and thereby establish the absence of bugs.
- Inclusion of the machine model allows the POR algorithm to be applicable to concurrent x86 assembly programs.
- The machine model is designed to consume off-the-shelf x86 assembly programs, without the need for any translations or pre-processing. This imparts higher confidence over the effort as program translations are not a part of our TCB.
- The machine model allows the verification effort to be downstream of the compiler, thereby allowing the tool-set to identify illegal program behaviors that are a product of compiler translations.
- Incorporating the byte-addressable memory model allows for precise emulation of memory interactions by the x86 instructions.
- The machine model implements the x86-TSO RMM to provide a realistic, relaxed consistency environment for executing and verifying assembly. As a result, the emulation of x86 programs provided by the machine model is precise and thereby offers higher confidence over execution correctness.
- Verifying assembly programs does not rely on the presence of the source-code. Therefore, due to the inclusion of the x86 machine model, the verification effort is not reliant on the presence of the source code.

Finally, we demonstrate the applicability of Xavier by applying it on several classical mutual exclusion benchmarks and on several data-structures from a mainstream concurrency library- the Linux Userspace RCU (URCU). We have applied Xavier on 17 case-studies, ranging from 250 to 3700 lines of assembly. The benchmarks chosen vary in complexity,

both in terms of the lines of assembly program executed and the constructs used to achieve concurrency.

The goal of choosing the classical benchmarks is to demonstrate the applicability of Xavier to concurrent programs that achieve mutual exclusion through typical means such as spin-locks. Subsequently, the goal of choosing benchmarks from the URCU library is to demonstrate the applicability of Xavier to programs that achieve concurrency through means other than acquiring mutual-exclusion locks. Moreover, the results of the experiments have not only been used to demonstrate the applicability of Xavier, but also the effectiveness of the implementation. This was achieved through intentionally violating a behavioral assumption in one of the benchmarks, which was subsequently exposed by the implementation.

1.4 TCB of the Thesis

We will now discuss about the several factors of the machine model that are included in the TCB of Xavier, our tool-set.

1. We include the semantics implemented for the concurrency instructions of x86 ISA into the TCB. The semantics provided to such instructions have been tested against their specifications in the Intel (R) manuals [35]. However, the semantics for such instructions have not been formally verified.
2. We include the x86-TSO RMM implementation provided to support relaxed consistency execution into the TCB. The implementation follows the semantics from [10], [22], but, however, has not been formally verified.
3. We include the implementation of the Linux system calls into the TCB. The implementation semantics have been sourced from the Linux kernel implementations, and tested against corresponding simulations in GDB [36]. However, the semantics have not been formally verified.

1.5 Thesis Organization

The subject matter of the thesis is organized into the following chapters.

- In Chapter 2, we provide the necessary background for our solution, and an overview of the related works that have influenced the design decisions in this work.
- Chapter 3 provides two small motivating examples that first demonstrate the requirements for verifying concurrent assembly, and then the gains of the approach.

- Chapter 4 provides a formal view of the Concurrency model that underlines the novel POR algorithm.
- Chapter 5 provides a comprehensive description of the algorithm and its constructs.
- Chapter 6 outlines the Machine model, which is the implementation of algorithm to support assembly.
- Chapter 7 presents the experimental results of applying the tool-set on classical benchmarks, and on several data-structures from Linux URCU.
- Chapter 8 concludes the thesis with discussions on limitations of our approach and future work.

Chapter 2

Background and Related Work

This chapter provides the necessary background related to the concepts involved in this work. Section 2.1 begins the chapter by introducing Partial Order Reduction (POR), and provides details about several POR algorithms that this work derives inspiration from. The chapter then continues to Section 2.2 which introduces the required background on Relaxed Consistency memory models, and the application of POR to the verification of such models. Section 2.3 provides background on x86 assembly verification. Section 2.4 completes the chapter by contrasting the several works mentioned to the thesis contributions.

2.1 Partial Order Reduction

Model Checking. Formal analysis of concurrent programs, for soundness in verification, require systematic exploration of the thread interleavings. Model checking [37] has been one of the preferred methods employed to verify concurrent software. Model checking systematically explores the state-space of a given program and verifies the validity of each state achieved in doing so. However, model checking of concurrent software faces two considerable challenges: a) Model checkers typically store a large set of visited states which renders a systematic approach at verifying realistic concurrent programs intractable, and b) the nondeterminism in inter process communication can potentially result in an exponential blowup of the traces to be explored.

Stateless Model Checking. Stateless Model Checking (SMC) [1] has been one of the most predominantly used techniques to combat state-space explosion in concurrent program verification. SMC works through scheduled exploration of traces that capture all possible orderings of interactions between threads, without storing the state details. Hence, on one hand, Stateless model checkers can be very memory efficient. On the other hand, the stateless nature of SMC can potentially lead to exploring redundant traces reaching the same

states. Several techniques such as *context bounding* [38] and *depth bounding* [39] have been introduced in the past to combat the exploration redundancy. Amongst them, one of the most promising techniques is *Partial Order Reduction* (POR) [4], [40]–[42].

Partial Order Reduction. Partial Order Reduction (POR) algorithms were introduced to limit trace equivalent explorations, through exploiting *conflicts* or *dependencies* between thread actions. POR algorithms limit the trace exploration by, first, identifying classes containing equivalent traces. POR considers two traces to be equivalent if they adhere to the same *partial order* between dependent actions (called *Mazurkiewicz traces* [43]). Intuitively, two traces are considered to be equivalent if the execution order of *independent* actions in one trace is a permutation of the other. Secondly, POR algorithms then pursue a limited subset of explorations among the equivalent execution traces. Therefore, the main crux of POR algorithms is defining the the equivalence relation for identifying the trace-equivalence classes. POR algorithms are guaranteed to preserve soundness in state exploration by executing at-least one trace from all trace-equivalence classes, thereby exploring only a subset of the entire trace-space. Additionally, the soundness in exploration guarantees to cover all possible behaviors reachable in every interleaving.

Classification. POR algorithms can be broadly classified into two classes: *Static* POR and *Dynamic* POR. *Static* POR [44] rely on defining partial order between conflicting events *statically*, before execution. However, Static POR algorithms more than often over-approximate trace non-equivalence, as they can estimate dependencies that may not manifest during execution. Therefore, Static POR algorithms can potentially undertake redundant explorations. Several classes of works were subsequently introduced aiming at increasing the efficiency of POR. *Dynamic* POR (DPOR) algorithms [41] observe and define ordering between dependencies on-the-fly, which allows for a more fine-grained recognition of trace equivalencies. Intuitively, an action is executed in the current step only if it was proven to be dependent on an action from a previous step. Hence, DPOR algorithms resolve dependencies dynamically, thereby reducing the number of traces explored on-the-fly.

Works by Abdulla et. al. [45], [46] have then improved upon DPOR by introducing *Optimal Dynamic Partial-Order Reduction* (DPOR) which guarantees to explore only one trace from an equivalence class, hence *optimality*. A caveat, however, is that Optimal-DPOR *optimal* only under a *stateless* setting, by requiring execution steps to be unconditionally (in)dependent at all possible states.

Context Sensitivity. Several later works on DPOR have aimed at achieving a equivalence classes coarser than Mazurkiewicz traces, in view of reducing redundant trace exploration. One such foray is *Context Sensitivity* [26], [27], where the equivalence relation limits explorations that are executionally different but *statefully* redundant. These works utilize *context*

sensitive independence, where two actions a and b are required to be independent only in the in the *state* that they are executed from. Intuitively, a and b are independent from state s if executing either $a \cdot b$ or $b \cdot a$ from s results in the same state. Thus, context sensitive algorithms can be optimal in the number of global states reached.

Value sensitivity. While context sensitivity does result in a coarser dependency relation, it comes at the cost of having to execute the actions to decide independence and can also potentially initiate exploration of redundant traces. On the other hand, the dependency relations in all of the above works are inherently generic as they are insensitive to the values taken by the action variables. In contrast, works by [25], [27]–[29], [47] utilize a coarser dependency relation by taking the variable states into context, thereby achieving coarser equivalence classes. The initial works in this regard [28], [47] relied upon computationally expensive *Oracle-based* methods (such as requiring SMT solvers) to achieve a coarser trace scheduling. The later works by [25], [29] have developed static-analysis based methods to construct value-sensitive equivalence classes, which are exponentially coarser than their counterparts.

The POR algorithm presented in this paper is *dynamic* in nature, and the equivalence classes are dynamically calculated based on identifying stateful conflicts. The following chapters will describe the mechanisms of the algorithm in detail.

2.2 Relaxed Consistency Execution

Sequential Consistency. One of the most basic memory models for describing threads-memory interactions in a concurrent system is *Sequential Consistency* (SC) [48]. Under SC, concurrent threads possess a sequentially consistent view of the memory: the memory interactions across threads are assumed to follow a sequential order, with the memory operations in each thread following the pre-defined program order. The SC model is fairly intuitive from the perspective of a programmer since it can reasonably emulate program behavior. However, realizing sequentially consistent behavior is inefficient, as both the hardware and the compiler are required to explicitly serialize memory operations which otherwise can be executed in parallel [9]. Therefore, most modern processors implement *relaxed-consistency* behaviors where program ordering in a thread can be violated for certain instructions to achieve better performance.

Total Store Ordering. The realistic relaxed-consistency behavior of programs on most modern x86 processors can be emulated through using the x86-TSO memory model [10]. The relaxed-consistency execution in x86-TSO is achieved by allowing a write followed by a read, accessing different memory locations, to proceed out-of-program-order. Figure 2.1

represents the abstract model of x86-TSO, which has been sourced from [10] and provides semantics for the following operations.

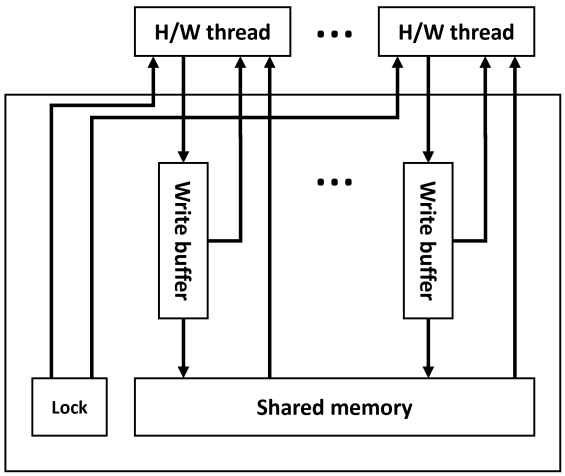


Figure 2.1: x86-TSO block diagram

1. **Store:** The TSO semantics buffers all writes to the memory by threads using per-thread store-buffers. That is, any thread that executes an x86 instruction that intends to write to the memory will not be allowed to do so immediately. Instead, the memory store by the thread is first buffered in the thread-specific store buffer, which is then flushed later to update the memory.

2. **Load:** Any thread that executes an x86 instruction that intends to read data from a memory address will first query the store-buffer for a corresponding store to the memory address. If a corresponding store is present, then the store value is forwarded to the instruction. The Load query will be forwarded to the memory only if a corresponding store is not found in the thread specific buffer.

3. **Update:** An Update operation by a thread will always flush a single store operation from the thread-specific store buffer to the memory. Flushing a store to the memory will update the value in the memory at the memory locations pointed to by the flushed store operation. The Update from a thread is non-deterministic and the update operation flushes stores of a thread in the order of their enqueuing. However, the TSO model forces no ordering between updates of different threads.

4. **Fence:** A Fence is a memory barrier operation that imposes the program ordering between actions before and after the execution of the Fence. Semantically, a Fence cannot be executed by a thread when its store buffer is non-empty.

5. **LOCK:** The x86 Instruction-Set Architecture (ISA) provides the LOCK prefix that when used allows Read-Modify-Write (RMW) instructions to acquire exclusive ownership of the appropriate bus, and in extension the memory, for the duration of the instruction. In essence, the LOCK prefix renders the RMW instruction atomic. To execute a LOCK'd instruction under the TSO semantics, a thread first acquires the bus lock and then executes the instruction. Before relinquishing the bus lock, the store buffer of the thread is flushed to the memory to guarantee atomicity.

TSO and POR. Verification of concurrent programs under relaxed consistency settings is non-trivial, since allowing instruction re-ordering can significantly blow-up the verification state-space. However, doing so is necessary to achieve realistic program verification. Several works such as [28], [29], [31], [49] have explored the application of DPOR to verify concurrent programs under relaxed memory settings. The main advantage of such approach is the ability to expose several more program behaviors which could be then exploited to fine-tune the interleaving semantics. Since the thesis is focused on assembly verification, the approach taken draws inspiration from these works in providing verification semantics for the x86-TSO memory model.

2.3 x86 Assembly Verification

With x86 being one of the more prominent architectures used for desktop and server processors, program verification at the assembly level has been a prominent area of research. There has generally been two prongs in research on x86 program verification: On one hand, in order to reason over assembly, several previous works [30], [32]–[34], [50] have contributed to developing execution semantics of the x86 ISA. On the other hand, several previous works [33], [51]–[53] have contributed in engineering the infrastructure around the semantics for program verification. But, to the best of our knowledge, there exists no other work till date that can efficiently emulate *and* model-check concurrent assembly execution, under relaxed consistency. Thus this section will delineate the previous works that this paper draws inspiration from in defining both the execution semantics and emulation environment for multi-threaded assembly programs.

The work on ISA semantics draws inspiration from [34] and [32]. Dasgupta et.al., in [34], have presented executable semantics for the most comprehensive set of x86 instructions till date, which supports the execution of sequentially consistent and single threaded programs. We utilized the formalism in [34] as a starting step in defining execution semantics for x86 concurrency and synchronization operations. However, the assembly verification infrastructure presented draws inspiration from [32]. The tool suite by Goel et. al., in [32], [33], formalizes about 33% user level instructions of the x86 ISA in the ACL2 theorem prover, with a goal of both explicitly simulating and formally reasoning over assembly. They have

also formalized the x86 system call (syscall) instruction, with execution support for system calls provided by the underlying OS for simulation.

2.4 Model Checking of Concurrent x86 Assembly

POR and x86 assembly. The main aim of the work is to present a model checking tool-set, Xavier, that can emulate and efficiently verify concurrent assembly execution. Incidentally, the "Efficiency" part is achieved by exploring the concurrent x86 machine state through applying POR. This, as stated in Section 1.3, is the first contribution of Xavier. However, as a critical distinction, the variations of POR algorithms presented in Section 2.1 apply to abstract languages, i.e, either to the source code or to a form of its intermediate representation (e.g, LLVM or Java Bytecode). In contrast, the approach taken involves applying POR to the x86 assembly, where each instruction can potentially have several side effects.

Any sound model checker consuming assembly will have to accurately emulate and track the modifications to the system state, through a concrete execution and memory model, which makes the verification model *stateful*. In order to take complete advantage of the stateful setting, we have drawn inspiration from [25] in making the POR algorithm sensitive to the value of the state constituents, and thereby *stateful*. What differentiates our algorithm from the state-of-the-art is its dependence on *order-sensitivity* to recognize equivalence classes. The following chapters will delineate this contribution in-depth.

Concurrent x86 assembly verification. The second contribution of Xavier, as mentioned in Section 1.3, is similar to the contributions in [32], [33]. It focuses on both defining the execution semantics for the x86 ISA and on modelling an external non-deterministic execution environment [54]. However, there are several distinctions of Xavier from the state-of-the-art.

The first distinction is the support for concurrency and the focus on the emulation model for multi-threaded assembly execution. The execution model supports dynamic thread creation by providing execution semantics for the Linux *pthread* library. Secondly, the framework provides support for defining semantics for system calls, thereby not relying on the underlying OS for sound execution. Additionally, the execution model is not limited to *sequential consistency*, as the memory model included in Xavier supports the TSO relaxed memory model[10]. Finally, the approach of the thesis is geared towards developing an emulation framework for concurrent assembly programs, rather than completeness in defining execution semantics for x86 ISA.

Chapter 3

POR for x86 Assembly

This chapter is intended to i) delineate the requirements for efficiently verifying multi-threaded assembly programs using POR algorithms and ii) demonstrate the benefits of applying POR at the assembly level. To that end, the chapter is organized as follows. Section 3.1 details the requirements that any POR algorithms verifying concurrent assembly should uphold. Section 3.2 details the several benefits of applying POR algorithms to assembly.

3.1 Requirements

Due to the large semantic gap, program verification at the assembly level is typically more involved than verification at either the source or the byte-code level. Hence, the application of traditional POR algorithms to assembly programs can be faced with several challenges. We will attempt to demonstrate such challenged and the requirements of applying POR to assembly with the help of an example.

Figure 3.1a contains the snippet of a concurrent program with two threads. The program contains a global variable `y` and two integer pointers `a` and `b`. The two threads first carry out a simple concurrent update of `y`, and then the two threads update the memory locations pointed to by `a` and `b` respectively. Figure 3.1b, on the other hand, represents the assembly equivalent of the program in Figure 3.1a. We have chosen to represent only the lines of assembly that directly correspond to the lines in the source-code.

We will now use the code snippet in Figure 3.1 to demonstrate the requirements for verifying a concurrent assembly program through POR.

1. POR algorithms traditionally recognize program variables through their declarative names. While higher level programs can be clearly expressed through variables, as-

```
assume  y :: int = 1
        a :: int* = ∅
        b :: int* = ∅

// thread 1        // thread 2
  y = 1;             y = 2;
 *a = 1;            *b = 2;
```

(a) Source code

```
// thread 1                                  // thread 2
   mov dword ptr [rip+@], 0x1 # <y>          mov dword ptr [rip+$], 0x1 # <y>
// ------                                    // ------
   mov rax, rdi                              mov rax, rdi
   mov dword ptr [rax], 0x1                  mov dword ptr [rax], 0x2
```

(b) Assembly code

Figure 3.1: Assembly verification requirements

sembly programs have no notion of variables. This can be seen from the global variable `y`. While the variable is addressed unceremoniously in the source-code, `y` is accessed through `rip` relative addressing in the assembly. This makes recognizing trace equivalences in the usual fashion non-trivial. Hence, any POR algorithm working at the assembly level will have to be independent of *declarative* semantics of the program, i.e., be independent of variable names.

2. An assembly program would have access to a limited set of hardware registers, which would be used to represent program variables. Hence, the set of hardware registers could potentially be reused while representing a large set of variables. Thus, in order to reason over the state of registers, any POR algorithm working at the assembly level will require to be *stateful*.

3. Pointers in higher level languages are typically associated with a type and have an abstracted view of the underlying memory. Pointers in the assembly, on the other hand, have no notion of types. Moreover, pointers in assembly are more precise as they deal with byte-addressable memory. Thus, any POR algorithm looking to efficiently handle pointers in assembly will require a strong byte-addressable memory model.

3.2 Benefits of applying POR to assembly

The aim of this section is to present an example that demonstrates the benefits of applying POR to assembly. Consider the following code snippet of concurrent string manipulation.

There are two threads t_1 and t_2 communicating over a shared contiguous piece of memory storing characters. Thread t_1 performs one action by initializing the memory region with a string. Thread t_2 performs three actions by concurrently initializing and subsequently writing new characters to this memory region.

```
assume  str :: string = "zz"
        itr :: char* = ∅

// thread 1          // thread 2
  1. str = "ab";       1. str = "ab";
                          for(itr = str.begin(); itr ≠ str.end(); ++itr)
                            2. *itr = 'a';
```

Figure 3.2: A toy program with two threads

- *Scalability.* All of the POR algorithms in the literature define dependency between *homogeneous actions*, i.e., actions of threads that work on the same underlying datatype. At a higher level of abstraction than the assembly, defining dependencies between non-homogeneous actions requires establishing a formal relationship between, *at the least*, all datatypes used in a program. This hiders scalability. However, all datastructures in the assembly are represented as a string of bytes. Thus, references to all datastructures in the assembly are homogeneous, which aids in scalability.

- Figure 3.2 can be used to demonstrate the second disadvantage with applying POR at an abstract level. Consider the following notation: let t_i^j donate the j^{th} action of thread i. Considering a stateless dependency relation, there exists 4 Mazurkeiewicz orderings.

$$O_1 :\ t_1^1 \cdot t_2^1 \cdot t_2^2 \cdot t_2^2 \qquad O_2 :\ t_2^1 \cdot t_1^1 \cdot t_2^2 \cdot t_2^2 \qquad O_3 :\ t_2^1 \cdot t_2^2 \cdot t_1^1 \cdot t_2^2 \qquad O_4 :\ t_2^1 \cdot t_2^2 \cdot t_2^2 \cdot t_1^1$$

  Any sound POR algorithm will explore at least the two traces O_1 and O_4 (since $\{O_1, O_2, O_3\}$ lead to the same end state). A POR algorithm that depends on only the Mazurkeiwz orderings will explore, at the least, all four traces. On the other hand, a POR algorithm that whose dependency relation is defined for homogeneous actions and is sensitive to the state of the dependent variables will explore only three traces (O_1, O_3 and O_4). This is so, since t_1^1 and t_2^1 are homogeneous writes writing the same value.

  However, in the assembly, all of the writes to memory are a string of bytes, irrespective of the underlying data-structure. This makes memory access actions in the assembly truly homogeneous, allowing to form a much coarser dependency relation. In the above

example, the string initialization actions (t_1^1 and t_2^1) are both seen as writes to of bytes to contiguous memory locations. Hence the actions t_1^1 and the first instance of t_2^2 are also seen as *independent*, as the overlapping memory regions get the same value written. Thus our POR algorithm will explore only two traces (O_1 and O_4) as $\{O_1,O_2, O_3\}$ form an equivalence class.

- Defining *stateful* dependency relations between actions operating on pointers at higher levels of abstraction requires extensive points-to analysis. This is so, since a pointer variable can potentially reference several datastructures during the lifetime of a program. However, such points-to analysis are unnecessary while analyzing pointers in the assembly. This is so since the assembly maintains a rigid and explicit model of the underlying memory. Therefore, memory locations of datastructures referenced in the program are explicitly calculated before the reference in assembly. This allows defining coarser dependency relations between assembly instructions with lesser computational effort, than actions at a higher level of abstraction.

Chapter 4

POR Algorithm: Definitions

This chapter will present the technical and formal background used in the rest of the work. The chapter is staggered as follows; Section 4.1 presents the basics for the concurrency model that underlines the algorithm contributed in this thesis. Subsequently, Section 4.2 will introduce several examples that will be used to explain critical definitions and concepts of the work. Finally, Section 4.3 we will introduce several essential definitions that the algorithm is built upon.

General notations. Given a sequence of elements $X = [x_0, x_1, \ldots, x_n]$, we use $\mathcal{E}(X)$ to denote the elements of X. Given a sequence $X = [x_0, x_1, \ldots, x_n]$ and an element $x_i \in \mathcal{E}(X)$, we define split $:: x \mapsto [x] \mapsto ([x], [x])$ as a function that returns a tuple of lists such that $\text{split}(x_i, X) = ([x_0, \ldots, x_{i-1}], [x_i, \ldots, x_n])$. Similarly, we define the function split_at $:: x \mapsto [x] \mapsto ([x], [x])$ as a function that returns a tuple of lists such that $\text{split\_at}(x_i, X) = ([x_0, \ldots, x_i], [x_{i+1}, \ldots, x_n])$.

4.1 Model basics

We consider the concurrency model $\mathcal{M}$ as a system working upon a multi-threaded assembly program $\mathcal{P}$ of $\{p_1, \ldots, p_j\}$ variable threads that communicate through manipulating *shared memory*. We make no distinction between threads and processes, as all threads are considered to be spawned by the same program. Thread p_1 behaves as the main thread of the program, and is initialized statically at the beginning of the simulation with a unique starting state $\mathtt{s}_0$. Subsequently, $\{p_2, \ldots, p_j\}$ are threads dynamically created during the execution of the program. The concurrency model imposes no restrictions on the number of threads in the system and makes no assumptions about the constituents of the states; the underlying machine model is entirely responsible for defining the states and their manipulations.

Model semantics. We describe the execution behavior of $\mathcal{M}$ as a transition system $\mathcal{M} = \{\Sigma, \mathcal{A}, \ \mathtt{s_0}, \rightarrow, \text{enabled}\}$. The characteristics of $\mathcal{M}$ are as follows;

- Σ is the set of all the program states reached during the program analysis.
- $\mathtt{s_0} \in \Sigma$ is the initial state of the transition system. $\mathtt{s_0}$ is supplied externally to the model, and every valid execution in $\mathcal{M}$ should start at the initial state.
- $\mathcal{A} = \bigcup_{i=1}^{j} \mathcal{A}_i$ represents the set of all the actions to be performed by the program. Each thread p_i of the program is assumed to execute a set $\mathcal{A}_i = \{\mathtt{a_i^1}, \ldots, \mathtt{a_i^j}\}$ of atomic *actions*, resulting in advancing the program *state* at each execution step.
- $\rightarrow :: \Sigma \mapsto \mathcal{A} \mapsto \Sigma$ is the step function that, given a state and an action produces the next state.
- enabled $:: \Sigma \mapsto \{\mathcal{A}\}$ represents the set of actions enabled in a state for execution.

In a given program state $\mathtt{s}$, a thread in the system can either be blocked from execution, or can have an action *enabled* to be executed. A thread is considered to be blocked in the present state if i) it has no more actions to execute, or ii) if the instruction semantics require it to be blocked (unsuccessful lock acquire or exceptions etc.). Moreover, there can be multiple actions enabled in a state, since the state can potentially have multiple un-blocked threads. Additionally, given an action $\mathtt{a_i} \in \mathcal{A}_i$, $\widehat{\mathtt{a_i}}$ represents the thread p_i executing the action.

From each program state $\mathtt{s} \in \Sigma$ the model requires the execution of a *single* atomic action $\mathtt{a} \in \text{enabled}(\mathtt{s})$, thereby advancing both the shared state of the system and also the local state of the executing thread. Consequently, from a state $\mathtt{s} \in \Sigma$, $\text{chosen}(\mathtt{s}, \mathtt{a}) :: \Sigma \mapsto \mathcal{A} \mapsto \mathbb{B}$ represents the action $\mathtt{a} \in \text{enabled}(\mathtt{s})$ chosen to be executed. We represent $\mathtt{s} \xrightarrow{a} \mathtt{s'}$ as a legal transition from states $\mathtt{s} \in \Sigma$, by executing the action $\mathtt{a} \in \text{enabled}(\mathtt{s}) \wedge \text{chosen}(\mathtt{s}, \mathtt{a})$ in state $\mathtt{s}$, to achieve the state $\mathtt{s'} \in \Sigma$. Additionally, we call a state $\mathtt{s} \in \Sigma$ *final*, notation $\text{final}(\mathtt{s})$, if $\text{enabled}(\mathtt{s}) = \emptyset$, i.e., there are no actions enabled in state $\mathtt{s}$, from all threads, to continue execution.

4.2 Running Examples

The rest of this section will introduce several definitions that are vital to describe our algorithm. Therefore, we will use the following examples to explain these definitions in detail.

Figure 4.1 represents a toy multi-threaded program $\mathcal{P}_1$ with 4 threads. The state of the program is comprised of four *global* variables $(\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z})$. An action is considered *global* if

assume $\mathtt{s}_0 \equiv \mathtt{w} = \mathtt{x} = \mathtt{y} = \mathtt{z} = 0$

// thread 1	// thread 2	// thread 3	// thread 4
1. x = y;	2a. w = 1 ;	3a. skip;	4. z = w;
	2b. skip;	3b. y = z;	

Figure 4.1: Example program $\mathcal{P}_1$

accesses the state of the global variables, and *local* otherwise. All the threads perform at-least one *global* action, which work on the global variables, with zero or more local actions. Each action is annotated with a unique identifier, which will be used to represent the action in an trace. There can be several possible execution interleavings amongst the execution ordering of the thread actions, resulting in several possible maximal traces.

Using the example to describe the constructs of this work requires defining the abstract concurrency model for the the program $\mathcal{P}_1$. This entails instantiating the program $\mathcal{P}_1$ as a transition system $\mathcal{M} = \{\Sigma, \mathcal{A}, \ \mathtt{s}_0, \rightarrow, \text{enabled}\}$, where

- Σ is the set of all the program states reached during the program analysis.
- $\mathtt{s}_0 \in \Sigma$ is the initial state of the transition system with the values of all the variables $\{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$ initialized to 0.
- $\mathcal{A} = \bigcup_{i=1}^{4} \mathcal{A}_i$ represents the set of all the actions to be performed by the program.
- The step function $\rightarrow :: \Sigma \mapsto \mathcal{A} \mapsto \Sigma$ is defined straightforwardly as executing an atomic action from a state, thereby updating the value of the global variables.
- enabled $:: \mathcal{S} \mapsto \{\mathcal{A}\}$ represents the set of actions enabled in a state for execution. The actions in a thread are enabled in thread order. For example, in thread 2, action `2b` will be enabled only after the execution of `2a`. A given thread p_i is considered to be blocked in a state `s` only if enabled(`s`) $= \emptyset$.

assume $\mathtt{s}_0 \equiv \mathtt{w} = \mathtt{x} = \mathtt{y} = \mathtt{z} = 0$

// thread 1	// thread 2	// thread 3	// thread 4
1. x = y;	2. w = 1 ;	3. y = z;	4. z = w;

Figure 4.2: Example program $\mathcal{P}_2$

Program $\mathcal{P}_2$ presented in Figure 4.2 is a simplified version of the program $\mathcal{P}_1$ in Figure 4.1: we have chosen to omit the *local* actions of the threads to achieve a simplified representation. Defining the abstract concurrency model for the the program $\mathcal{P}_2$ is similar in every way to that of $\mathcal{P}_1$, except that threads in program $\mathcal{P}_2$ perform only one global action.

```
assume  s0 ≡ x = y = z = 0
// thread 1      // thread 2      // thread 3
 1a. x = y;       2. z = x;       3. y = z;
 1b. x = 1;
```

Figure 4.3: Example program $\mathcal{P}_3$

```
assume  s0 ≡ x = y = z = 0
// thread 1      // thread 2      // thread 3
 1a. y = 1;       2. z = x;       3. y = z;
 1b. x = y;
```

Figure 4.4: Example program $\mathcal{P}_4$

Figure 4.3 and Figure 4.4 represent another toy multi-threaded programs $\mathcal{P}_3$ and $\mathcal{P}_4$ with three threads. Defining the abstract concurrency model for the the program $\mathcal{P}_3$ and $\mathcal{P}_4$, follows the same procedure as for program $\mathcal{P}_1$. This entails instantiating the program $\mathcal{P}_3$ and $\mathcal{P}_4$ as a transition system $\mathcal{M} = \{\Sigma, \mathcal{A}, \ \mathtt{s}_0, \rightarrow, \text{enabled}\}$, where

- Σ is the set of all the program states reached during the program analysis.
- $\mathtt{s}_0 \in \Sigma$ is the initial state of the transition system with the values of all the variables $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ initialized to 0.
- $\mathcal{A} = \bigcup_{i=1}^{3} \mathcal{A}_i$ represents the elements of all the actions to be performed by the program. Each thread $p_i \cdot i \in \{1, \ldots, 3\}$ of the program executes at-least one atomic action, with zero local actions, resulting in advancing the program state at each execution step.

The rest of the instantiations are the same as from the program $\mathcal{P}_1$.

4.3 Model Definitions

We will now provide several key definitions in this section that will be used to define the behavior of the concurrency model, and the succeeding algorithm.

Traces. We define a trace τ as an ordered sequence of actions intended to be executed from a given state $\mathtt{s} \in \Sigma$. Given a trace τ and a state $\mathtt{s} \in \Sigma$, we define the executability of the trace τ from the given state $\mathtt{s}$ as follows;

Definition 4.3.1. ***Executability.*** *A trace* $\tau = [\mathtt{a}_m, \ldots, \mathtt{a}_n]$ *is executable from a state* $\mathtt{s} \in \Sigma$*, notation* executable$(\mathtt{s}, \tau)$*, if and only if we can define a list of states* $\mathcal{S}(\tau) = [\mathtt{s}_m, \ldots, \mathtt{s}_{n+1}]$ *such that:*

$$\mathtt{s}_m = \mathtt{s} \ \wedge \ \forall_{m \leq i \leq n} \ \mathtt{s}_i \xrightarrow{\mathtt{a}_i} \mathtt{s}_{i+1}$$

Intuitively, the execution of a trace represents the series of legal transitions that the system can perform by executing the sequence of actions, starting from the given state.

Example 4.3.1. ***(executability)****. We will consider an example execution from Figure 4.1 to understand executability. Consider the state* $\mathtt{s} = (\mathtt{w} = \mathtt{x} = \mathtt{y} = \mathtt{z} = 0)$ *and a trace* $\tau = [\mathtt{2a}, \mathtt{2b}, 4]$*. The state* $\mathtt{s}$*, incidentally, is equivalent to the initial state of the system. Hence, a trace of* $\mathcal{P}_1$ *that respects thread ordering should be executable from* $\mathtt{s}$*. And indeed so, we have*

$$\text{executable}(s, [\mathtt{2a}, \mathtt{2b}, 4])$$

This is because, we can construct the witness list of states $\mathcal{S}(\tau) = [\mathtt{s}, \mathtt{s}_1, \mathtt{s}_2, \mathtt{s}_3]$ *from the definition of executability, that corresponds to a legal execution of* τ*.* $\mathtt{s}_1 = \text{step}(\mathtt{s}, \mathtt{2a})$ *gives us a legal state* $\mathtt{s}_1 = (\mathtt{w} = 1, \mathtt{x} = \mathtt{y} = \mathtt{z} = 0)$ *by executing the action* $\mathtt{2a}$ *from the state* $\mathtt{s}$*. Since action* $\mathtt{2b}$ *is a skip, we have* $\mathtt{s}_2 = \mathtt{s}_1$*. Finally, we get* $\mathtt{s}_3 = step(\mathtt{s}_2, 4)$*, where* $\mathtt{s}_3 = (w = z = 1, x = y = 0)$*. Hence, we can represent the computation of* τ *from* $\mathtt{s}$ *as*

$$\mathtt{s} \xrightarrow{\mathtt{2a}} \mathtt{s} \xrightarrow{\mathtt{2b}} \mathtt{s}_2 \xrightarrow{4} \mathtt{s}_3$$

Now consider the trace $\tau' = [\mathtt{2b}, \mathtt{3a}, \mathtt{3b}]$ *and state* $\mathtt{s} = (\mathtt{w} = \mathtt{x} = \mathtt{y} = \mathtt{z} = 0)$*. Trace* τ' *does respect thread ordering in threads. But we have*

$$\neg \, \text{executable}(\mathtt{s}, [\mathtt{2b}, \mathtt{3a}, \mathtt{3b}])$$

This is because τ' *fails the definition of executability, We cannot define a state* $\mathtt{s}'$ *such that* $\mathtt{s} \xrightarrow{\mathtt{2b}} \mathtt{s}'$*, since action* $\mathtt{2b} \notin enabled(\mathtt{s})$*. Thus the trace* τ' *is not executable from* $\mathtt{s}$*.*

Given an trace $\tau = [\mathtt{a}_0, \ldots, \mathtt{a}_m]$, we define τ as a *valid* trace, if and only if τ is executable from the initial state $\mathtt{s}_0$. In other words, the the execution of τ from $\mathtt{s}_0$ will result in a valid computation of $\mathcal{M}$. We represent the computation of τ on as $\mathtt{s}_0 \xrightarrow{[\mathtt{a}_0, \ldots, \mathtt{a}_m]} \mathtt{s}_{m+1}$ where

$$\mathtt{s}_0 \xrightarrow{[\mathtt{a}_0, \ldots, \mathtt{a}_m]} \mathtt{s}_{m+1} = \mathtt{s}_0 \xrightarrow{\mathtt{a}_0} \mathtt{s}_1 \xrightarrow{\mathtt{a}_1} \mathtt{s}_2 \xrightarrow{\mathtt{a}_2} \ldots \mathtt{s}_m \xrightarrow{\mathtt{a}_m} \mathtt{s}_{m+1}$$

Every valid trace begins with the initial state $\mathtt{s}_0$ of $\mathcal{M}$. Thus, the execution of a valid trace τ can be used to characterize the state of the system. Notation $\mathtt{s}_{[\tau]}$ denotes the state of $\mathcal{M}$ achieved after computing τ on $\mathcal{M}$. Therefore, while determining the executability of a trace τ' from a state $\mathtt{s}'$, we can overload the use of executable$(\mathtt{s}', \tau')$ by passing in $\mathtt{s}_\tau$ as the

first parameter, where $\mathtt{s}_{[\tau]} = \mathtt{s}'$. Hence, *notation* executable(τ, τ') indicates that the trace τ' will be executable only after the execution of the valid trace τ. Additionally, a valid trace $\tau = [\mathtt{a}_0, \ldots, \mathtt{a}_m]$ is considered to be *maximal* if its last state $\mathtt{s}_{m+1}$ is final. Finally, we use $\tau \cdot \tau'$ denote the extension of a valid trace τ with another trace τ'. That is, for a trace τ, $\tau \cdot \tau'$ represents a valid trace if

1. τ is valid
2. τ is not maximal
3. We have executable(τ, τ'), that is τ' is executable after τ, from state $\mathtt{s}_{[\tau]}$.

We will explain the definition of *trace extension* with an example

Example 4.3.2. ***(trace extension)***. *We will revisit the example from Figure 4.1 to explain the extension of traces. Consider the trace* $\tau = [1, 2a, 3a]$ *being executed from the initial state* $\mathtt{s}_0 = (\mathtt{w} = \mathtt{x} = \mathtt{y} = \mathtt{z} = 0)$. *The trace* τ *is executable from the initial state* $\mathtt{s}_0$, *since it follows thread order in all the involved threads. Hence, we have*

$$\text{executable}(\mathtt{s}_0, [1, 2a, 3a])$$

Trace τ *is a valid trace since it was executed from the initial state. Hence, we can represent the state of the system, after the computation of tau as* $\mathtt{s}_{[\tau]} = (\mathtt{w} = 1, \mathtt{x} = \mathtt{y} = \mathtt{z} = 0)$. *Now consider the trace* $\tau' = [2b, 3b, 4]$. *For* τ' *to be executable after* τ, *we must have i)* τ *being valid, ii)* τ *not being maximal. And indeed, we have so since* τ *was executed from the initial state and* enabled$(\mathtt{s}_{[\tau]}) \neq \emptyset$. *We must then have iii) The trace* τ' *is executable from* $\mathtt{s}_{[\tau]}$. *And indeed, that is so since we can construct the witness list of states* $\mathcal{S}(\tau') = [(\mathtt{w} = 1, \mathtt{x} = \mathtt{y} = \mathtt{z} = 0), (\mathtt{w} = 1, \mathtt{x} = \mathtt{y} = \mathtt{z} = 0), (\mathtt{w} = \mathtt{z} = 1, \mathtt{x} = \mathtt{y} = 0)]$, *that represent the computation of* τ' *after* τ. *The actual construction of the witness list of states is left to the reader. Therefore, we have*

$$\text{executable}(\tau, \tau')$$

Consider another trace $\tau'' = [2a, 3b, 4]$. *In this case,* τ'' *is not executable after* τ, *since* τ'' *does not respect the definition of executability. That is because action* $2a \notin \text{enabled}(\mathcal{S}_\tau)$. *Hence, we have*

$$\neg\, \text{executable}(\tau, \tau'')$$

Reachability. Given a step function $\rightarrow$, we use $\rightarrow^* :: \Sigma \times \Sigma \mapsto \mathbb{B}$ to denote reachability. That is, $\mathtt{s} \rightarrow^* \mathtt{s}'$ denotes that there exists some trace τ that is executable from $\mathtt{s}$ such that $\mathtt{s}'$ can be reached by executing τ' from $\mathtt{s}$. Using reachability, we define *freach* as follows;

Definition 4.3.2. ***freach.*** *The* freach *of state* $\mathtt{s}$*, notation* freach($\mathtt{s}$)*, is defined as the set of final states reachable from* $\mathtt{s}$*. That is*

$$\text{freach}(\mathtt{s}) \equiv \{\mathtt{s_f} \cdot final(\mathtt{s_f}) \ \wedge \ \mathtt{s} \rightarrow^* \mathtt{s_f}\}$$

We denote freach as a set of states, since there can be multiple final states reachable from a given state. This is because each state reached in the system can possibly have multiple actions enabled. And consequently, the order in which the enabled actions are executed can influence the configuration of the final state reached. We will demonstrate this definition in the following example

Example 4.3.3. ***(freach, Part I).*** *We will revisit the program in Figure 4.1 to demonstrate this definition. Consider the initial state* $\mathtt{s_0} = (\mathtt{w} = \mathtt{x} = \mathtt{y} = \mathtt{z} = 0)$ *of the program.* freach($\mathtt{s_0}$) *denotes the set of all final reachable states from* $\mathtt{s_0}$*. Since* $\mathtt{s_0}$ *is the initial state of the system,* freach($\mathtt{s_0}$) *is a set of all possible, unique, final states of the program. That is,*

$$\text{freach}(\mathtt{s_0}) \equiv (\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}) = \{(1,0,0,0), (1,0,0,1), (1,0,1,1), (1,1,1,1)\}$$

This is because we can define at-least four maximal traces (for example (1,2a,2b,3a,3b,4), (1,2a,2b,4,3a,3b), (1,4,2a,2b,3a,3b), (2a,2b,4,3a,3b,1)) that are executable from the initial state s_0*, and can lead to the four final states respectively.*

(freach, Part II) *Consider another state of the system* $\mathtt{s'} = (\mathtt{w} = 1, \mathtt{c} = \mathtt{y} = \mathtt{z} = 0)$*. State* $\mathtt{s'}$ *is a legal state of the program, since it can be achieved by executing the trace* $\tau = [2a]$ *from the initial state* $\mathtt{s_0}$*. The* freach *of state* $\mathtt{s'}$ *will then be*

$$\text{freach}(\mathtt{s'}) \equiv (\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}) = [(1,0,0,1), (1,0,1,1), (1,1,1,1)]$$

This is because we can define at-least three traces (e.g., $\{[\mathtt{2b}, 1, \mathtt{3a}, \mathtt{3b}, 4]$*,* $[\mathtt{2b}, 1, 4, \mathtt{3a}, \mathtt{3b}]$*,* $[\mathtt{2b}, 4, \mathtt{3a}, \mathtt{3b}, 1]\}$*) that are executable from state* $\mathtt{s'}$ *and can lead to the three final states. Moreover,* freach($\mathtt{s'}$) *does not include the final state* $(\mathtt{w} = 1, \mathtt{x} = \mathtt{y} = \mathtt{z} = 0)$ *since there can be no trace that would be executable from* $\mathtt{s'}$ *and can reach the final state* $(\mathtt{w} = 1, \mathtt{x} = \mathtt{y} = \mathtt{z} = 0)$

Example 4.3.3 clearly demonstrates the influence that choosing actions to execute from a state have over reachability. As seen from Example 4.3.3 Part I, every possible end state of the program is reachable from the state $\mathtt{s_0}$. Since $\mathtt{s_0}$ is the initial state, all possible valid traces are rendered executable, thereby allowing all end states to be reachable from $\mathtt{s_0}$. Whereas in Example 4.3.3 Part II, the number of states that are reachable from state $\mathtt{s'}$ is limited. The action 2a was first chosen to be executed from the state $\mathtt{s_0}$. This resulted in the variable w taking the value of $\mathtt{w} = 1$ in the state $\mathtt{s'}$. Beginning from state $\mathtt{s'}$, there can be no other enabled actions that assign a value to the variable w, which restricts the value of w to be 1 in all the final states reachable from $\mathtt{s'}$.

We can examine the execution of the valid trace $\tau = [\texttt{2a}, \texttt{4}, \texttt{3a}, \texttt{3b}, \texttt{1}]$ to understand this behavior better. We coin the term *dataflow* to any sequence of actions involved in a chain of computation. Moreover, from the initial state $\texttt{s}_0$, every possible final state is reachable. However, by executing the action $\texttt{2a}$ $\texttt{w} = \texttt{1}$ from the initial state, the data $\texttt{w} = \texttt{1}$ now *flows* to every later state that utilizes the value of $\texttt{w}$. This restricts the value of $\texttt{w}$ to be $\texttt{1}$ in all final states henceforth. Similarly, extending the valid trace $[\texttt{2a}]$ with $[\texttt{4}]$ results in the data $\texttt{1}$ *flowing* from $\texttt{w}$ to $\texttt{z}$, which restricts the value of $\{\texttt{w}, \texttt{z}\} = \texttt{1}$ in all states reached henceforth. Finally, extending the valid trace $[\texttt{2a}, \texttt{4}]$ with $[\texttt{3a}, \texttt{3b}, \texttt{1}]$ results in the data $\texttt{w} = \texttt{1}$ *flowing* to $\texttt{x}$ via $\texttt{z}, \texttt{y}$. This restricts the final reachable state to be $\{\texttt{w}, \texttt{x}, \texttt{y}, \texttt{z}\} = \{\texttt{1}, \texttt{1}, \texttt{1}, \texttt{1}\}$.

We term the influence that executing actions in an order have over the reachable final states as a *order-sensitivity*. In Example 4.3.3 Part II, executing the action $\texttt{2a}$ from state $\texttt{s}_0$ to achieve the state $\texttt{s}'$. resulted in the data $w = 1$ *flowing* to the states in the rest of the exploration that directly or indirectly utilizes the value of w. Hence, all final states that have $\texttt{w} = \texttt{1}$ became reachable from the state $\texttt{s}'$. Alternatively, if the action $\texttt{1}$ $(\texttt{x} = \texttt{y})$ was executed from the initial state $\texttt{s}_0$, to achieve state $\texttt{s}''$, then every final state that have $\texttt{x} = \texttt{0}$ would have become reachable from $\texttt{s}''$. The algorithm, while exploring new states, dynamically detects these order-sensitivities in the current trace and sets. Formally, we define a order-sensitivity as follows.

Definition 4.3.3. ***order-sensitivity.*** *Let $\tau = \tau_1 \cdot \tau_2$ be a valid trace, where $\tau_2 = [\texttt{a}_0, \texttt{a}_1, \ldots, \texttt{a}_n]$ is a valid extension after trace τ_1. Trace τ_2 is* order-sensitive *after trace τ_1, notation* order-sensitive(τ_1, τ_2)*, if and only if:*

$$\exists \tau'. \text{executable}(\tau_1, \tau') \wedge \texttt{a}_0 \notin \tau' \wedge (\tau') = \texttt{a}_n \wedge \text{freach}(\tau_1 \cdot \tau') \not\subseteq \text{freach}(\tau_1 \cdot \tau_2)$$

Let $\texttt{s}$ be the state reached after execution of trace τ_1, which we will term as the *start-state* of the order-sensitivity. order-sensitivity indicates that the execution of the trace τ_2 from state $\texttt{s}$, caused at least one final state to become unreachable from $\texttt{s}_{[\tau]}$. This was because the last action $\texttt{a}_n$ in τ_2 recorded a dataflow that the first action $\texttt{a}_0$ was not a part of. Moreover, there exists some alternative trace where $\texttt{a}_n$ is executed *before* action $\texttt{a}_0$ such that this final state does become reachable. In other words, executing $\texttt{a}_n$ before $\texttt{a}_0$ renders $\texttt{a}_0$ as the new recipient of the dataflow, that uncovers at least one final state unreachable after the execution of τ.

Example 4.3.4. ***(order-sensitivity)*** *We revisit the running example. Consider the trace $\tau = [\texttt{1}, \texttt{2a}, \texttt{2b}, \texttt{4}, \texttt{3a}, \texttt{3b}]$. Intuitively, one expects a dataflow from action 2a $(\texttt{w} = \texttt{1})$ via action 4 $(\texttt{z} = \texttt{w})$ to action 3b $(\texttt{y} = \texttt{z})$: the data-value 1 flows from $\texttt{w}$ via $\texttt{z}$ to $\texttt{y}$. Delaying this dataflow until after execution of action 1 caused final state $\{\texttt{w}, \texttt{x}, \texttt{y}, \texttt{z}\} = \{1, 1, 1, 1\}$ to become unreachable. Indeed, we have:*

$$\text{order-sensitive}([\,], [\texttt{1}, \texttt{2a}, \texttt{2b}, \texttt{4}, \texttt{3a}, \texttt{3b}])$$

This holds, since there exists an alternative trace $\tau' = [\mathtt{2a}, \mathtt{4}, \mathtt{3a}, \mathtt{3b}]$ *that does not have action* `1` *in it and ends with action* `3b`. *The final state* $\{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\} = \{1, 1, 1, 1\}$ *is reachable after execution of* τ'*, but is unreachable after execution of* τ.

In the Example 4.3.4, the action sequence $[\mathtt{2a}, \mathtt{4}, \mathtt{3a}, \mathtt{3b}]$ in τ represented a dataflow that action `1` was not initially a part of. However, executing the trace $\tau' = [\mathtt{2a}, \mathtt{4}, \mathtt{3a}, \mathtt{3b}]$ before action `1` resulted in `1` now becoming the recipient of the dataflow. This shuffle in ordering resulted in uncovering a new final state $\{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\} = \{1, 1, 1, 1\}$.

Note that τ' is the minimal trace needed to flow the data-value 1 from `w` to `z` in the initial dataflow. Omitting any of the actions would either make the trace τ' non-executable after τ, or the variable z would end up with value 0. Consequently, action `1` would not be the new recipient of the dataflow in the shuffled trace, Thus τ' is the most suitable trace to explore in order to uncover new final states. Therefore, we are interested in the minimal trace needed to replay a dataflow, without the loss of data, which we call as the *dataflow-sequence*.

Definition 4.3.4. ***dataflow-sequence (dfs).*** *Let* $\tau_2 = [\mathtt{a}_0, \mathtt{a}_1, \ldots, \mathtt{a}_n]$ *be an order-sensitive trace after a valid trace* τ_1. *The* dataflow sequence *of this order-sensitivity, notation* $\mathrm{dfs}(\tau_1, \tau_2)$, *is defined as the largest action sequence* `as` *such that the predicate* $P_{\mathrm{dfs}}(\mathtt{as})$ *holds. Here predicate* P_{dfs} *is inductively defined by the following equations:*

$$\begin{aligned} &\textit{(base)} && P_{\mathrm{dfs}}([\mathtt{a}_n]) \\ &\textit{(rec.)} && \textit{if} \left\{ \begin{array}{l} \mathtt{as} = [\mathtt{a}_i, \ldots, \mathtt{a}_n] \\ P_{\mathrm{dfs}}(\mathtt{as}) \\ 0 < j < i \\ \mathrm{necc\_for}(\mathtt{a}_j, \mathtt{as}) \\ \forall_{j<k<i}. \neg\, \mathrm{necc\_for}(\mathtt{a}_k, \mathtt{as}) \end{array} \right\} \textit{then } P_{\mathrm{dfs}}(\mathtt{a}_j, \mathtt{as}) \end{aligned}$$

The predicate necc_for *is defined as follows. Consider an action* $\mathtt{a} \in \mathcal{E}(\tau_2)$, *such that* $\tau' \cdot \tau'' = \mathrm{split}(\mathtt{a}, \tau_2)$. *The action* $\mathtt{a} \in \mathcal{E}(\tau_2)$ *is necessary to be included in* $\mathtt{as} = \mathrm{dfs}(\tau_1, \tau_2)$, *notation* $\mathrm{necc\_for}(\mathtt{a}, \mathtt{as})$, *if*

$$\mathrm{necc\_for}(\mathtt{a}, \mathtt{as}) = \exists \mathtt{a}' \in \mathtt{as} \;.\; \widehat{\mathtt{a}} = \widehat{\mathtt{a}'} \;\vee \text{order-sensitive}(\tau_1 \cdot \tau', \tau'')$$

Definition 4.3.4 details the predicate that is required to construct the dataflow-sequence, and the construction happens in the opposite order of execution. The $\mathrm{dfs}(\tau_1, \tau_2)$ is the minimal sequence of actions in τ_2 that is required to replay the dataflow in order-sensitive(τ_1, τ_2) after the execution of τ_1. From the definition of order-sensitivity, action $\mathtt{a}_n$ in τ_2 is the recipient of the dataflow, and hence is trivially required to replay the dataflow. This is the base case of P_{dfs}.

The recursive case of P_{dfs} defines the rest of the actions from τ_2 that are required to construct the dataflow. The definition uses the predicate necc_for, expressed *"necessary for"*, to define this requirement. While constructing the *dataflow-sequence* $\texttt{as}$ to replay the dataflow in τ_2, action $\texttt{a} \in \mathcal{E}(\tau_2)$ is necessary to be included in $\texttt{as}$, if

1. There is an action $\texttt{a}' \in \texttt{as}$ that is executed by the same thread as that of action $\texttt{a}$. *or*
2. Action a itself initiates a dataflow to the last action $\texttt{a}_\texttt{n}$ of τ_2.

The first requirement for necc_for preserves the thread order in the threads, while replaying the dataflow. The second requirement is necessary to preserve actions that take part in the actual dataflow to the last action $\texttt{a}_\texttt{n}$ of τ_2. We will illustrate the construction of a *dataflow-sequence* with an example.

Example 4.3.5. ***(dfs).*** *Consider the order-sensitive trace in presented in Example 4.3.4, that is* order-sensitive$([\,], [\texttt{1}, \texttt{2a}, \texttt{2b}, \texttt{4}, \texttt{3a}, \texttt{3b}])$ *where* $\tau_1 = [\,]$ *and* $\tau_2 = [\texttt{1}, \texttt{2a}, \texttt{2b}, \texttt{4}, \texttt{3a}, \texttt{3b}]$. *The dataflow in the example occurred from action* $2a$ $(\texttt{w} = \texttt{1})$ *via action* 4 $(\texttt{z} = \texttt{w})$ *to action* $3b$ $(\texttt{y} = \texttt{z})$. *The* dfs *of this dataflow should be comprised of actions that enable the dataflow. And indeed, we have*

$$\text{dfs}(\tau_1, \tau_2) = [\texttt{2a}, \texttt{4}, \texttt{3a}, \texttt{3b}]$$

The dfs *for the dataflow in* τ_2 *is constructed using the predicate* P_{dfs} *as follows:*

1. $P_{\text{dfs}}([\texttt{3b}])$ *trivially holds from the base case, since action* $\texttt{3b}$ *records the dataflow*
2. $P_{\text{dfs}}([\texttt{3a}, \texttt{3b}])$ *holds, since from the recursive case,* necc_for$(\texttt{3a}, [\texttt{3b}])$ *holds. This is because, from the definition of* necc_for*, action* $3a$ *is necessary to enable action* $\texttt{3b}$ *as* $\widehat{\texttt{3a}} = \widehat{\texttt{3b}}$
3. $P_{\text{dfs}}([\texttt{4}, \texttt{3a}, \texttt{3b}])$ *holds, since from the recursive case,* necc_for$(\texttt{4}, [\texttt{3a}, \texttt{3b}])$ *holds. This is because* order-sensitive $([\texttt{1}, \texttt{2a}, \texttt{2b}], [\texttt{4}, \texttt{3a}, \texttt{3b}])$ *holds. Intuitively, action* $\texttt{4}$ *is included in the* dfs*, since it participates in the original dataflow by passing on the value of variable* $\texttt{w}$ *to* $\texttt{y}$ *in action* $\texttt{3b}$*, through the variable* $\texttt{z}$.
4. $P_{\text{dfs}}(\texttt{2b}, [\texttt{4}, \texttt{3a}, \texttt{3b}])$ *does not hold since, firstly, the base case does not hold. Secondly, the recursive case does not hold either as* necc_for$(\texttt{2b}, [\texttt{4}, \texttt{3a}, \texttt{3b}])$ *does not hold. This is because i)* $\nexists \texttt{a}' \in \{\texttt{4}, \texttt{3a}, \texttt{3b}\}$ *such that* $\widehat{\texttt{2b}} = \widehat{\texttt{a}'}$ *and ii) there is no dataflow in* $[\texttt{2b}, \texttt{4}, \texttt{3a}, \texttt{3b}]$*. That is,* order-sensitive $([\texttt{1}, \texttt{2a}], [\texttt{2b}, \texttt{4}, \texttt{3a}, \texttt{3b}])$ *does not hold*
5. $P_{\text{dfs}}([\texttt{2b}, \texttt{4}, \texttt{3a}, \texttt{3b}])$ *holds since from the recursive case, there is a dataflow from action* $\texttt{2a}$ *to* $\texttt{3b}$*, via* $\texttt{4}$ *in* τ_2*. That is,* order-sensitive $([\texttt{1}], [\texttt{2a}, \texttt{2b}, \texttt{4}, \texttt{3a}, \texttt{3b}])$ *holds*

The aim of the algorithm is to dynamically construct maximal traces that explores all unique end states of the program. However, as seen from Definition 4.3.3, order-sensitivities in a trace can result in limiting the final reachable states from the trace. The dataflow-sequence of such order-sensitive traces can be *replayed* from the corresponding start-state, in order to uncover these hidden final reachable states. However, not all dataflow-sequences will be replayable, from the start-state of their corresponding order-sensitive traces. Therefore, we will introduce here the concept of *Replayability*. We define an order-sensitivity to be replayable, if the dataflow-sequence for the order-sensitivity is replayable. The following definition will detail the semantics of Replayability.

Definition 4.3.5. ***Replayability***. *Let $\tau = \tau_1 \cdot \tau_2$ be a valid trace, where $\tau_2 = [\mathtt{a_0}, \mathtt{a_1}, \ldots, \mathtt{a_n}]$ represents an order-sensitive after the valid trace τ_1. Let $\tau_{as} = \text{dfs}(\tau_1, \tau_2)$ be the dataflow-sequence for a the order-sensitivity in τ_2. The order-sensitivity in τ_2 is replayable if τ_{as}, the* dfs *of* order-sensitive(τ_1, τ_2) *adheres to the predicate* re-playable(τ_1, τ_{as})*, where*

$$\begin{aligned}\text{re-playable}(\tau_1, \tau_{as}) = {} & (1)\,\text{executable}(\tau_1, \tau_{as}) \wedge \\ & (2)\,\nexists \tau' \tau'' \cdot \tau' \cdot \tau'' = \tau_2 \wedge \tau' \neq [\,] \wedge \tau'' \neq [\,] \wedge \text{order-sensitive}\,\tau_1, \tau'\end{aligned}$$

Let $\mathtt{s}$ be the start-state of order-sensitive(τ_1, τ_2). The action sequence τ_{as} can only be replayed from the state $\mathtt{s}$, if $\tau \cdot \tau_{as}$ represents a valid trace. Case (1) of the predicate expresses this soundness condition, by requiring τ_{as} be executable from the state $\mathtt{s}$. Intuitively, replaying the order-sensitivity requires executing the action a_0 after the valid trace $\tau_1 \cdot \tau_{as}$. This execution can be unsound if executing $\mathtt{a_0}$ after the actions from τ_{as} violates the thread order of $\widehat{\mathtt{a_0}}$.

Case (2) preserves the integrity of the dataflow during its replay from state $\mathtt{s}$. Essentially, the trace τ_{as} would be unable to replay the dataflow, if the action $\mathtt{a_0}$ is an integral part in computation chain of the dataflow. Intuitively, executing the trace τ_{as} before executing the action $\mathtt{a_0}$ from state $\mathtt{s}$ prevents the dataflow from occurring. We will explain these definitions in the following examples.

Example 4.3.6. ***(replayability, case (1)).*** *Consider the example program in Figure 4.3. We will examine a trace in the execution of the program $\mathcal{P}_3$, to understand the case (1) of replayability. Consider the valid trace $\tau = [\mathtt{1a}, \mathtt{1b}, 2, 3]$. There exists a dataflow from action 1b ($\mathtt{x} = 1$), via the action 2 ($\mathtt{z} = \mathtt{x}$) to action 3 ($\mathtt{y} = \mathtt{z}$). And indeed we have*

$$\text{order-sensitive}([], [\mathtt{1a}, \mathtt{1b}, 2, 3])$$

This is because we can define another valid trace $\tau' = [2, 3]$ that does not have the action 1a *in it and ends with the action* 3*. The final state $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\} = \{1, 0, 0\}$ is reachable after the*

execution of τ', but is unreachable after the execution of τ. In order to replay this order-sensitivity we will construct the τ_{as}, the dataflow-sequence of the order-sensitive trace. Hence, we have

$$\tau_{as} = \text{dfs}([], \tau) = [\texttt{1b}, 2, 3]$$

The construction of the dfs *is left to the reader. The trace τ_{as}, as stated in Definition 4.3.4, contains the necessary actions required to replay the order-sensitivity. However, the order-sensitivity is not replayable since*

$$\neg\, \text{re-playable}([], \tau_{as})$$

From case (1) of Definition 4.3.5, the dfs *τ_2 is not executable from the start-state of the program. Executing the trace τ_{as} from the start-state will violate the thread order in* `Thread1`*, since replaying the order-sensitivity entails executing the action* `1b` *before* `1a`*.*

Remark. *On a side note, the final state $\{\texttt{x}, \texttt{y}, \texttt{z}\} = \{1, 0, 0\}$ can be uncovered by replaying* order-sensitive$([1a], [1b, 2, 3])$

Example 4.3.7. ***(replayability, case (2)).*** *Consider the example program in Figure 4.4. We will examine a trace in the execution of the program $\mathcal{P}_4$, to understand the case (2) of replayability. Consider the valid trace $\tau = [\texttt{1a}, \texttt{1b}, 2, 3]$. We have*

$$O_1 \equiv \text{order-sensitive}([\texttt{1a}], [\texttt{1b}, 2, 3])$$

This is because we can define another valid trace $\tau' = [3]$ that does not have the action `1b` *in it and ends with the action* `3`*. The final state $\{\texttt{x}, \texttt{y}, \texttt{z}\} = \{0, 0, 0\}$ is reachable from the execution of τ' after $[\texttt{1a}]$, but is unreachable after the execution of τ.*

In order to replay this order-sensitivity we will construct the τ_{as}, the dataflow-sequence of the order-sensitivity. Hence, we have

$$\tau_{as} = \text{dfs}([\texttt{1a}], \tau) = [2, 3]$$

The construction of the dataflow-sequence is left to the reader. The trace τ_{as}, as stated in Definition 4.3.4, contains the necessary actions in $[\texttt{1b}, 2, 3]$, that is required to replay the order sensitivity. However, we have

$$\neg\, \text{re-playable}([\texttt{1a}], \tau_{as})$$

This is because, from case (2) of Definition 4.3.5, we also have

$$O_2 \equiv \text{order-sensitive}([\texttt{1a}], [\texttt{1b}, 2])$$

From the Definition 4.3.3, by replaying O_1, action `1b` *becomes the recipient of the dataflow in $[\texttt{1b}, 2, 3]$. Moreover, by replaying O_2, action* `1b` *also becomes the recipient of the dataflow in $[\texttt{1b}, 2]$. Thus by transitivity, the action* `1b` *is an integral part of the dataflow in $[\texttt{1b}, 2, 3]$, which contradicts Definition 4.3.3.*

Chapter 5

POR Algorithm: Pseudo Code

The intent of this chapter is to present and explain our Partial Order Reduction algorithm, which is one of the core contributions of our work. The chapter is organized as follows. We will first present the basic version of our algorithm, in Section 5.1, with suitable soundness arguments which will be used to delineate the core concepts of the work. Subsequently, we will present a running example in Section 5.2 that describes the step-wise execution of our algorithm. We will then present the enhancements that were incorporated to better the performance of the algorithm, in Section 5.3. Ultimately, in Section 5.4 we will present the final version of our POR algorithm, which mirrors our implementation.

5.1 POR base algorithm

We are now ready to present a basic version of our main Partial Order Reduction algorithm, which is documented in Algorithm 1. The algorithm is described in a recursive fashion, and takes two inputs; iii) the current exploration trace and ii) a dataflow sequence to be replicated. The algorithm also requires the abstract concurrency model for a concurrent program, as it utilizes the concurrency constructs described in the chapter 4.

Algorithm 1: POR base algorithm

Requires: The concurrency model constructs
Initially: $por([\,],\ [\,])$

```
1  def por(τ, τ_df, ):
2      s ← s_[τ]
3      g ← (1) if τ_df ≠ ∅ then head(τ_df)
           (2) else pick_fair(enabled(s))
4      if g is none then
5          return
6      else
7          τ' ← τ · [g]
8          τ'_df ← tail(τ_df)
9          por(τ', τ'_df)
10         while ∃(τ1, τ2) . τ1 · τ2 = τ' ∧ order-sensitive(τ1, τ2) do
11             τ_df ← dfs(τ1, τ2)
12             if re-playable(τ1, τ_df) then
13                 por(τ1, τ_df)
```

The algorithm is basically a depth-first exploration of the state graph. However, not all actions are chosen to be explored from a state and thus not all states are explored. The entry point to the algorithm is the recursive function *por*(). The initial call of the *por*() function takes as input ii) the initial empty trace and ii) an empty dataflow sequence. Additionally, the aim of the algorithm is to quit after exploring all unique final states of the program.

Depth-First Execution. Any call to the function *por*() first tries to advance the current exploration state, by choosing an action to be executed. This can be seen in Line 3 of the algorithm. If such an action cannot be found, the algorithm performs the base case by returning from the recursive call (Line 5). If an action can be found, the algorithm performs the recursive case (Lines 6 - 13) by continuing the depth-first exploration. In the recursive case, the algorithm first extends the current trace with the chosen action (Line 7), shrinks the dataflow sequence by one action (Line 8), and and advances the depth-first exploration through the recursive call in Line 9.

Line 3 of the algorithm depicts how an action is chosen to be executed from the present state. An action would be chosen from either the dataflow sequence that needs to be replayed, or from the enabled pool of the current execution state. From Definition 4.3.4, the actions in a dataflow sequence are required to be executed in the sequence order, to replay a dataflow. Consequently, the algorithm first gives execution preference to the actions from the dataflow sequence, by picking the sequence actions over enabled actions. This is repre-

sented by case (1) of the assignment in Line 3.

Alternatively, case (2) of Line 3 represents the semantics of picking an action from the enabled set. The algorithm proceeds to pick an action from the enabled set only when there are no actions to be executed from the dataflow sequence and given that there are actions enabled for execution in the current program state. The case (2) of Line 3 makes sure that there can be only one action picked from an execution state since, from the concurrency model, there can be only one action chosen from an execution state.

The depth-first exploration, from Line 9, proceeds until the algorithm can find an atomic action to further the execution progress. However, the algorithm will eventually reach the *base* case when there are no more actions to be executed from an execution state. The concurrency model defines such a state as a *final* state and the corresponding execution trace as a *maximal* trace. Upon executing a maximal trace, the recursion stack unwinds the exploration stack into Line 10. The algorithm then proceeds to construct new traces that can reach new final execution states.

State exploration. Lines 10 - 13 details the construction of new traces to reach new final states. The algorithm does so by recognizing order-sensitive sub-traces, initially, in the current maximal trace (Line 10). Recall from Definition 4.3.3, the execution of an order-sensitive trace renders at-least one final state unreachable. Hence, in order to uncover unreachable states the algorithm first constructs the dataflow sequence (Line 11) for all such order-sensitivities, and utilizes the constructed sequences to uncover new final states.

The algorithm then confirms the constructed dataflow sequences to be re-playable from the start-state of the order-sensitive traces (Line 12). If so, the algorithm then proceeds to replay each constructed dataflow sequence, from start-state of the corresponding order-sensitive trace (Line 13). The replay is achieved by passing the dataflow sequence as a parameter to a new recursive *por*() call, which is parameterized to begin exploration from the initial state of the order-sensitive. The *por*() call then, in depth-first fashion proceeds to explore new maximal traces that can uncover the newly reachable final states. Finally, when all of the order-sensitive traces have been examined in the current maximal trace, the recursion stack unwinds the algorithm to return from Line 13.

5.2 Running Example

We will now replay the algorithm on a running example. Consider the example program $\mathcal{P}_4$ presented in Figure 4.2 (see Page 21). The Figure 5.1 shows the exploration tree of unique

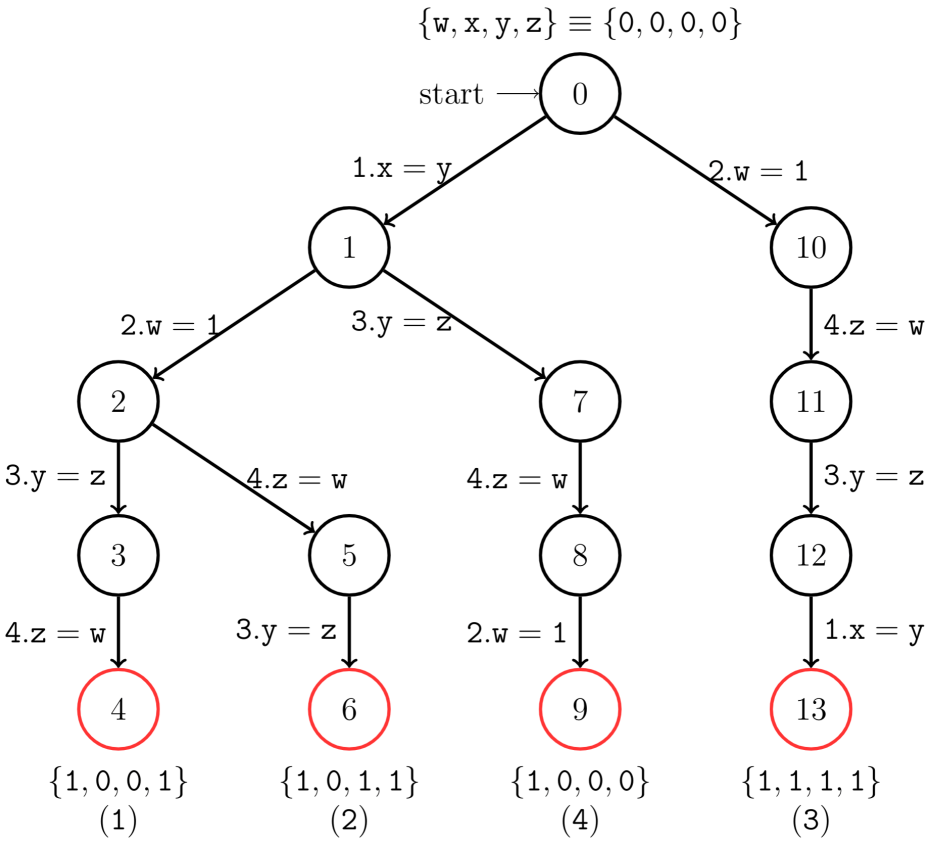


Figure 5.1: State exploration for program $\mathcal{P}_4$

end states obtained by running the algorithm on the program $\mathcal{P}_4$. Each node in the tree represents a state reached by the algorithm, and each arrow represents a transition. Every transition is annotated with the thread ID performing the transition and atomic action performed by the thread, from the execution state. The states $(\mathtt{s}_4, \mathtt{s}_6, \mathtt{s}_9, \mathtt{s}_{13})$ represent the unique final states reached during exploration and are annotated with their state configuration, and the order in which they are explored.

The algorithm begins exploration from the initial state $\mathtt{s}_0$ of the program. Hence, the initial function call for the algorithm becomes $por([\,], [\,])$. In depth-first fashion, the algorithm explores the first valid maximal trace $\tau_{m1} = [1, 2, 3, 4]$ leading to the first final state $\mathtt{s}_4 \equiv \{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\} = \{1, 0, 0, 1\}$. The trace τ_{m1} is executable trace from the state s_0 as, from Definition 4.3.1, we can construct the witness set of states $\mathcal{S}(\tau_{m_1}) = [\mathtt{s}_0, \mathtt{s}_1, \mathtt{s}_2, \mathtt{s}_3, \mathtt{s}_4]$ that represent a valid execution. Since the algorithm does not set up any dataflow replays, until a final state is reached, the actions for the first maximal trace are chosen for execution, at random, entirely from the enabled set (case (2) of Line 3).

Upon reaching the first maximal trace, the recursion backtracks to Line 10. The algorithm recognizes the first order-sensitivity in τ_{m1}, that is, order-sensitive($[1,2],[3,4]$). Here $\tau_1 = [1,2]$ and $\tau_2 = [3,4]$. This holds since, from Definition 4.3.3, there exists an alternative trace $\tau' = [4]$ that does not have action 3 in it and ends with action 4. The final state $s_6 \equiv \{w,x,y,z\} = \{1,0,1,1\}$ is reachable after execution of $\tau_1 \cdot \tau'$, but is unreachable after execution of τ. Therefore, the algorithm proceeds to uncover the currently unreachable state. The algorithm first defines the dataflow sequence $\tau_{df} = [4]$ (Line 11). The construction of the dataflow sequence happens according to Definition 4.3.4, where τ_{df} is the minimal sequence in τ_2 that is required to replay the dataflow. The algorithm then finds the dataflow sequence to be re-playable after τ_1 (Line 12), as from Definition 4.3.5, τ_{df} is executable after τ_1. Hence, the algorithm sets-up the dataflow replay from the state $s_{[\tau_1]} = s_2$, by calling $por([1,2],[4])$ (Line 13).

The algorithm now begins a new branch of execution from the state s_2. Since the dataflow sequence $\tau_{df} = [4]$ needs to be replayed, the algorithm chooses the action 4 from τ_{df} for execution (case (1) of Line 3). The algorithm then continues execution, in depth-first fashion and by choosing actions from the enabled pool, to reach the second new final state $s_6 \equiv \{w,x,y,z\} = \{1,0,1,1\}$. $\tau_{m2} = [1,2,4,3]$ is the second maximal trace corresponding to this execution. Upon reaching the second maximal trace, the recursion backtracks to Line 10. The algorithm recognizes the second order-sensitivity in τ_{m2}, that is, order-sensitive($[\,],[1,2,4,3]$), which makes the final state $s_{13} \equiv \{w,x,y,z\} = \{1,1,1,1\}$ become unreachable. Here $\tau_1 = [\,]$ and $\tau_2 = [1,2,4,3]$. Therefore, the algorithm proceeds to uncover this unreachable state. The algorithm defines the dataflow sequence $\tau_{df} = [2,4,3]$ (Line 11). The construction of the dataflow sequence happens according to Definition 4.3.4. τ_{df} is the minimal sequence in τ_2 that is required to replay the dataflow, and is re-playable after τ_1 (Line 12). The actual construction of the dataflow sequence is left to the reader. The algorithm then proceeds to set-up the dataflow replay from the state $s_{[\tau_1]} = s_0$, by calling $por([\,],[2,4,3])$ (Line 13).

The algorithm now begins a new branch of recursive execution from the state s_0. The algorithm first replays the dataflow trace τ_{df} (case (1) Line 3) and proceeds to reach the third new final state $s_{13} \equiv \{w,x,y,z\} = \{1,1,1,1\}$. $\tau_{m3} = [2,4,3,1]$ is the third maximal trace corresponding to this execution. Upon reaching the third maximal trace, the recursion backtracks to Line 10. The algorithm, in customary fashion, proceeds to find order-sensitive traces in τ_{m3}. However, there are no order-sensitive traces in τ_{m3} that can lead to a new state that has not been reached before. Therefore, even though the base algorithm has the provision to explore these redundant order-sensitive traces, our implementation of the algorithm contains enhancements which will prohibit the algorithm in doing so. We will detail the intuition behind these enhancements and their implementations in later chapters. Therefore the recursion now backtracks to continue finding order-sensitive traces in the maximal trace τ_{m2}. There are no new order-sensitive traces in the maximal trace τ_{m2} that can

lead to states unexplored before. Hence the algorithm further backtracks to continue finding order-sensitive traces in the maximal trace τ_{m1}.

Upon backtracking to the maximal trace τ_{m1}, the algorithm finds the third order-sensitive trace. That is order-sensitive($[1], [2, 3, 4]$), which makes the final state $\mathtt{s_9} \equiv \{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\} = \{1, 0, 0, 0\}$ become unreachable. Here $\tau_1 = [1]$ and $\tau_2 = [2, 3, 4]$. Therefore, the algorithm defines the dataflow sequence $\tau_{df} = [3, 4]$ (Line 11), that will uncover this unreachable final state. Since τ_{df} is re-playable after τ_1, the algorithm then proceeds to set-up the dataflow replay from the state $\mathtt{s}_{[\tau_1]} = \mathtt{s_1}$, by calling $por([1], [3, 4])$ (Line 13). The depth-first execution from this recursive call leads to uncovering the final state $\mathtt{s_9}$, through the maximal trace $\tau_{m4} = [1, 3, 4, 2]$. In the maximal trace τ_{m4}, the algorithm finds no order-sensitive traces that can lead to a new state that has not been reached before. Hence, the algorithm backtracks to the execution of the maximal trace τ_{m1}. As the algorithm has now exhausted exploring all order-sensitive traces in τ_{m_1}, the algorithm then finishes execution by completely unwinding the recursion stack.

5.3 Improvements to the base algorithm

The algorithm presented in Figure 1 is intended as a basic version that represents the critical concepts of this work. As noted in the previous chapter, Algorithm 1 is sound in its execution, meaning that, given a concurrent program the algorithm will explore *all* unique final states of the program. However, the algorithm provides room for several enhancements, which are incorporated by the actual implementation, to increase the execution performance. These enhancements come in the form of i) an over-approximation for dynamically detecting order-sensitivity in execution traces and ii) an intelligent way of recognizing redundant order-sensitive traces by the way of book-keeping. Therefore, we will now detail the enhancements incorporated in our implementation of the algorithm, by providing an argument for their requirement and by describing their mechanisms.

Over-approximation of order-sensitivity. The Definition 4.3.3 of Chapter 4 details the semantics of recognizing order-sensitive execution among valid execution traces. Intuitively, Order-sensitivity in a trace represents the fact that execution of the trace actions, in the order defined by the trace, results in one or more of the final states becoming unreachable after the trace execution. Therefore, given the execution of a valid trace, determining order-sensitivity in the trace requires establishing two factors: (i) the final reachable state from the current trace and (ii) a permutation of the current trace that can uncover a final state that is unreachable after the execution of the current trace. However, following the definition to establish order-sensitivity in a trace presents a nuanced requirement.

Consider the execution trace $\tau_{m1} = [1, 2, 3, 4]$ of program $\mathcal{P}_4$ represented in Figure 5.1. For the trace τ_{m1}, we have

$$\text{order-sensitive}([1, 2], [3, 4])$$

Fulfilling the requirement (i) for establishing the above order-sensitivity in this maximal trace is trivial. That is, given its execution, the final reachable state from the maximal trace τ_{m1} is the state $\mathtt{s}_4$. This is trivially true since $\mathtt{s}_4$ is the state reached after the execution of the valid trace τ_4. Hence, for the trace τ_{m1} we have,

$$\mathtt{s}_4 \equiv \{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\} = \{1, 0, 0, 1\} \in \text{freach}(\mathtt{s}_{\tau_{m_1}})$$

However, fulfilling the requirement (ii) is not straightforward. Fulfilling requirement (ii) entails both constructing the permutation sequence (the dataflow sequence) that contain the original trace actions in a different order, and finding the final reachable states of that permutation sequence.

From definition 4.3.4, the construction of the permutation sequence is recursively dependent on the definition of order-sensitivity. Moreover, even if the dataflow sequence is defined as a 'simple' permutation of the original trace, establishing all of its final reachable states requires the complete execution of all possible maximal traces, that contain the permutation sequence as a prefix.

Having to execute a new trace in order to establish order-sensitivity would be a hindrance to the performance. The hindrance would be even more pronounced if the execution of the new trace does not resulting in uncovering a new final state. Therefore, our implementation of the algorithm includes an over-approximation of Order-sensitivity, that can recognize order-sensitive traces dynamically without having to execute new traces. The semantics of this over-approximation will be detailed in Chapter 6.4.

Execution of Redundant Traces. The algorithm presented above, as stated before, is the basic version and can potentially explore redundant traces. We will revisit the running example to explain this case. Consider the scenario where the algorithm has completed the execution of the maximal trace $\tau_{m1} = [1, 2, 3, 4]$ to reach the first final state $\mathtt{s}_4 \equiv \{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\} = \{1, 0, 0, 1\}$. The recursion backtracks to Line 10 and detects the first order-sensitivity in the maximal trace, that is

$$\text{order-sensitive}([1, 2], [3, 4])$$

As recorded in the example replay, the algorithm then begins the execution of $\tau_{m2} = [1, 2, 4, 3]$, to reach the second final state $\mathtt{s}_6 \equiv \{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\} = \{1, 0, 1, 1\}$.

After the execution of τ_{m2}, the recursion backtracks to Line 10 to find order-sensitive traces in τ_{m2}. One of the order-sensitive traces that the algorithm can find in τ_{m2} is

$$\text{order-sensitive}([1, 2], [4, 3])$$

However, there is no provision in the base algorithm that can stop the exploration of this order-sensitivity. The exploration of this order-sensitivity is redundant as, in the process the algorithm will explore the trace $\tau' = [1, 2, 3, 4]$ which is the same as τ_{m1} and has already been executed before. Therefore, our implementation of the algorithm includes a book-keeping mechanism that keeps track of the order-sensitive traces previously explored, in order to stop redundant executions.

5.4 Algorithm Back-end Implementation Requirements

In the course of describing the algorithm, we have referenced several constructs that any implementation of the algorithm has to support. The constructs necessary to be implemented range from the function semantics necessary for the Abstract Concurrency Model (described in Section 4) to the algorithm specific improvements (described in Section 5.3). We will utilize this section to detail all such requirements, and use the section as a prelude to the introduction of the Machine model (Section 6), which details our algorithm implementation.

State Automaton Constructs

To analyze a concurrent program, the algorithm requires the abstract concurrency model representation of the program. The abstract concurrency model is described as a transition system $\mathcal{T} = \{\Sigma, \mathcal{A}, \ \mathtt{s_0}, \rightarrow, \text{enabled}\}$. The state automaton constructs of the transition system is detailed in Section 4. Hence, any implementation of the algorithm has to provide the semantics for the constructs utilized by the transition system that adhere to the following description.

- $\Sigma, \ \mathtt{s_0}$: The implementation should provide meaningful semantics to the system state, to help track the progress of both the algorithm and the underlying concurrent program. The state of the system should easily be able to distinguish between the individual thread states (local state $\mathcal{L}$) and the state of the program (global state $\mathcal{G}$). The implementation should also provide semantics to the initial state of both the program and the algorithm.
- $\mathcal{A}$: The algorithm implementation should provide meaningful semantics to the instruction set of the program. The instruction semantics is necessary to both describe the execution state and help evaluate the functions utilized by the algorithm.
- $\rightarrow$: The implementation should provide semantics to the transition function of the concurrency model. The transition function is necessary to advance the execution state by utilizing the instruction semantics of the program.

- enabled : The algorithm implementation should provide semantics to the enabled function, that adheres to the semantics of the program and the system state.

Algorithm Constructs

In addition to the State Automaton constructs of the concurrency model, the algorithm utilizes several functions, described by the concurrency model, to decide and advance the course of state-space exploration. Hence, it is necessary for the algorithm implementation to provide implementation semantics to the constructs listed below

- order-sensitivity: An integral function of the algorithm is to determine order-sensitivity in explored traces, in order to find new final states. Hence, any implementation of the algorithm should provide an implementation to the order-sensitivity semantics, in order to explore order-sensitivity in executed traces. As stated before, our implementation of the algorithm provides an over-approximated implementation for determining order-sensitivity in a trace. The implementation semantics will be described in the following section.
- Dataflow Sequence construction: From Definition 4.3.4, construction of the Dataflow Sequence is necessary to explore order-sensitivities in a trace. Therefore, any implementation of the algorithm should implement the semantics for constructing dataflow sequences.

The aforementioned constructs are necessary to be provided by any implementation of the algorithm, for sound execution. As a case in point, our implementation of the algorithm provides versions of these constructs that are tuned for executing assembly. Therefore, with the information presented in this section, we are now ready to present our implementation of the algorithm - The Machine Model.

Chapter 6

Machine Model

The intent of this chapter is to present and explain the machine model, which is another core contribution of this work. The chapter is organized as follows. We will first present the design structure of the machine model in Section 6.1. Subsequently, we will present the design goals considered in engineering the machine model in Section 6.2. In Section 6.3, Section 6.4 and Section 6.5 we then present the technical details about the machine model structure.

The machine model is designed to serve as an *explicit state* model checking interface between the POR algorithm and the assembly programs. While the POR algorithm is responsible for scheduling execution traces, the machine model is responsible for initializing, maintaining and analyzing the deterministic state changes according to the program execution. Moreover, the intent of the machine model is to provide a *modular* and an *extensible* framework for verification. In this chapter, we will detail the parameters considered in making these design decisions and provide technical details on the constituents of the machine model.

6.1 Design Structure

Popular instruction simulators such as [36], [55]–[58] have been often used to efficiently emulate and analyze assembly execution. There has also been notable effort, [33], in building assembly simulators that can formally reason over assembly execution. These simulators either have the ISA semantics written in higher level languages like C/C++ to optimize execution, or use the underlying OS to provide execution for OS constructs like interrupts and system calls. However, most modern assembly simulators are either tuned towards single threaded execution, or provide minimal support for run-time analysis of multi-threaded programs. Moreover, these simulators utilize optimized, but, sequentially consistent memory models to support execution. Hence, the need for a realistic multi-threaded x86 emulator justifies the decision of designing the interactive machine model and to support the TSO

relaxed consistency [10] environment.

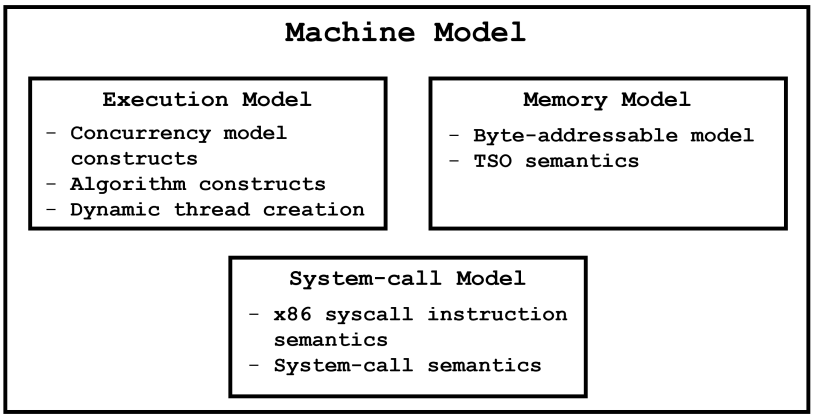


Figure 6.1: machine model Structure

For the sake of brevity and simplistic design, as shown in Figure 6.1, we have bifurcated the machine model into three modules - the *Execution Model* the *Memory Model* and the *System-call Model*. The requirements for the three modules are as follows.

Since the machine language is imperative in nature, emulating multi-threaded x86 assembly execution requires an *execution model* that can execute x86 assembly, supports dynamic thread creation and can simultaneously track the execution of all spawned threads. The execution model must also be a framework, in order to accommodate addition of semantics for new x86 instructions. Moreover, the execution model is required to be stateful in nature, in order to maintain and advance the state information of the program constructs, processor specific constructs and the model specific artifacts.

Several constructs of an x86 program can potentially access and manipulate the memory, hence a *memory model* is necessary to support the semantics of such constructs. Moreover, in order to support the program execution under a realistic setting, the memory model needs to support relaxed consistency in program execution.

Finally, user-level programs often request resources from the underlying OS through system-calls. Hence, in order to provide better control over program execution, a *system-call model* is required to provide semantics for system-call instructions that are encountered during execution. Having discussed the design intent, we will now detail the several design goals that were considered during the engineering of the machine model.

6.2 Design goals

The design of the machine model constituents is complex enough to warrant the need for the following several design goals. For instance, with increasing complexity of assembly programs, the machine model should be simplistic enough, so as to not hinder its *Usability*. Also, the machine model has to be *Accurate* in faithfully emulating the execution of the assembly code and be *Efficient* in its representation of the program states. Finally, the machine model should preserve *Scalability*, with increasing complexity of the program execution. We will now discuss about these factors in detail.

Usability. While the POR algorithm is responsible for scheduling execution, the machine model is responsible for both emulating the assembly execution *and* analyzing the states reached for violations in program behavior. The machine model is also responsible for providing debugging support, when assertion violations are encountered. However, with increasing threads and complexity of the program, the number of states reached can become quite large and complex. Therefore, it is necessary for the machine model to maintain *Usability*, by striking a balance between the *"ease of use"* and the complexity of the information provided to the user.

Maintaining a simplified and compartmentalized implementation of the state is one of the main solutions used to preserve the *Usability*. The information produced by the machine state, on one hand, would often be necessary for the user during program debug. For example, the program state includes information about the per-thread register values, system flags and the memory contents. It would be necessary to enable access to these information to the user during program debug. On the other hand, the machine state also contains information that are extraneous at the user level. For example, when debugging memory constructs, it would be cumbersome to the user to interact directly with the underlying complex memory model. Moreover, users would prefer a simple linear memory abstraction of the memory model. Other state information pertaining to the model artifacts such as debug flags, interface status and program setup would require to be completely abstracted, as they provide no valuable information to the user during simulation and debug. Hence, in order to preserve usability, the execution model state is designed to present information at different levels of abstraction to the user, based on the debugging autonomy.

The machine model is also designed to preserve usability during program setup. Similar to other mainstream assembly simulators [36], [58] the machine model does not require any pre-processing done over the assembly. The machine model extracts all configuration and thread local information, required for program setup, directly from the object file. More details about the program setup and usability are presented in the following sections.

Accuracy and Efficiency. One of the main design goals of the machine model was to retain *Accuracy* in representation of the x86 machine state and the ISA semantics. For example, the register bank is represented with 64-bit or 128-bit machine integers, rather than unbounded abstract integers. The memory is represented as an array of bytes, whose access supports all forms of pointer arithmetic. The memory model is augmented with TSO relaxed consistency, to provide a realistic emulation environment. Moreover, no simplifications were performed on the implementation of ISA semantics and the semantics were sourced from the Intel (R) ISA manuals [35]. Additionally, the machine model also preserves accuracy in the implementation of library functions, such as the system calls and the POSIX-thread library functions, by sourcing the semantics from Linux library implementations.

Pursuing accuracy and detail in the machine model implementations comes with an inherent trade-off with *Efficiency*, as the model becomes more complex. Therefore, the machine model mitigates this trade-off by abstracting the information supply to the user, based on the depth of interaction required. For example, if the user is just interested in a simulation, all debugging capabilities are turned off and an abstract representation of the execution is presented. On the other hand, if the user intent is debugging, then access to the machine state and instrumentation functions are provided, based on the requirement. Moreover, simple constructs such as lists and maps are used in the state implementation, to manage the dynamically growing threads states, to balance complexity and efficiency.

Scalability The machine model is implemented as a framework in order to provide support for adding semantics for new instructions from the x86 ISA and implementations of new system calls. Hence the size and complexity of the machine model can potentially grow with the addition of new features. Hence, the constructs of the machine model are implemented with scalability in mind, such that addition of new features should not hamper either the performance or the soundness of preexisting functionality.

Having discussed about both the design intent and the considerations taken during the engineering of the machine model, we are now ready to present the technical details of the machine model constituents. Therefore, in the following sections, we will describe, in detail, the execution model, the memory model and the system-call model.

6.3 Execution Model

During our research, we found that there was a requirement for a realistic debugging tool that supports the relaxed consistency execution of *concurrent* assembly. The function of a simulator satisfying such requirements was to enable dynamic instrumentation of the concurrent machine state, under relaxed consistency. Hence, the execution model is engineered

as an emulator, to emulate execution of x86 instructions under the x86-TSO [10] relaxed memory setting. The execution model is interfaced with the POR algorithm, to achieve the interleaved thread execution, where the algorithm acts as an instruction scheduler, performing scheduling decisions during run-time. Therefore, designing the POR algorithm to be interactive allows for analyzing and instrumenting x86 assembly and provide various levels of scheduling autonomy to the user during execution. Moreover, the the interactive emulation of a concurrent program is performed as a linearization to a single threaded execution, which assists in the run-time program analysis and debug.

The execution model presented in this work draws inspiration from the works by [32], [33]. The responsibilities of the execution model are broadly defined as follows

R.1 To provide the framework that instantiates the state automaton constructs, defined in section 5.4. The state automaton constructs are required by the concurrency model, which is entirely responsible for guiding the simulation.

R.2 To provide the framework that instantiates the algorithm constructs, defined in section 5.4. The algorithm constructs define functions that the algorithm uses to decide the course of the algorithm execution.

R.3 To provide semantics to a thread creation library, in order to allow the execution model to dynamically spawn threads.

Hence, we will now describe the constituents of the execution model that fulfill the above requirements.

6.3.1 Implementation of State Automaton Constructs

This subsection details the fulfillment of the requirement **R.1** by the execution model. The different state automaton constructs required by the Concurrency model are described in Chapter 4. The abstract concurrency model is described as a transition system $\mathcal{T} = \{\Sigma, \mathcal{A}, \ \mathtt{s_0}, \rightarrow, \text{enabled}\}$. The machine model implementation of these constructs are described as follows.

Program state Σ. From Section 4.1, Σ is defined as the set of all the program states reached during the program analysis. We define a program state object $s_i :: \mathcal{L} \times \mathcal{G}$ to be comprised of the set $\mathcal{L}$ of *local* states that are private to the threads and the global state $\mathcal{G}$, *shared* by all threads of the system.

Figure 6.2 details the composition of the Program state of the machine model. The program state of the machine model, `PState`, is a *concrete*, mutable object that keeps track of the

simulation progress. `PState` is also a *singular* object, since the simulation deals with a program at a time.

data `PState` :
  `shared` :: `GState`
  `threads` :: *map* (`tid` ↦ `TState`)

(a) Program state

data `GState` :
  `memory` :: *map* (`addr` :: `64bit` ↦ `data` :: `8bit`)
  `sync` :: *Bool*

(b) Global shared state

data `TState` :
  `regs` :: `Registers`
  `flag` :: *map* (`Flags` ↦ 𝔹)
  `buffer` :: [`Write`]

(c) Thread state

Figure 6.2: State configuration

The configuration of the program state object `PState` is as follows

1. `threads`: From Figure 6.2a, the *local* state $\mathcal{L}$ is instantiated by the member field `threads`. The field maintains a dynamic map from unique thread identifiers *tid* to the thread states `TState`. Each thread upon creation is assigned a unique thread ID, with 0 representing the *main* thread.

   The state for the main thread is initialized statically during program setup, the semantics of which will be detailed later in the chapter. Whenever a new thread is spawned, the state for the thread is initialized and a corresponding mapping is added to the field. Consequently, when a thread dies, the mapping for the thread is removed from the field.

2. `shared`: From Figure 6.2a, the *global* shared state $\mathcal{G}$ is instantiated by the member field `shared`. The purpose of this field is to record the global state data-structure `GState`that is shared amongst all the threads.

The configuration of the global state data-structure `GState` is detailed as follows.

1. `memory`: The field represents the byte-addressable memory block that maps 64bit unsigned integer addresses to dynamically allocated bytes. The raw memory block physically supports allocation up-to 2^{64} bytes, but in order to preserve compatibility with existing modern x86 implementations, the block is logically limited to support

2^{52} bytes. In order to keep the memory framework lightweight, memory is allocated and de-allocated precisely and upon request. The in-depth design considerations and memory initialization is discussed in-detail in the section 6.4.

2. `sync`: This field records the several flags shared between the threads, used for System call implementations. The semantics of this field will be elaborated upon in Section 6.5.

Type	Configuration	Comments
General-Purpose Registers	16 64-bit registers	Registers such as rax, rbx,..., etc. 32-bit and 16-bit representations are also provided, wherever supported by the Intel (R) manuals [35]
Instruction Pointer	1 64-bit register	Register rip
Segment Registers	6 64-bit registers	cs, ds, es, fs, gs, ss registers
XMM registers	16 128-bit registers	Registers such as xmm0, xmm1,..., xmm15. 64-bit representations of the registers are also provided, wherever supported by the Intel (R) manuals [35]
Flag registers	5 1-bit registers	We currently support zf, cf, sf, of and pf semantics

Table 6.1: Supported X86 register stack

The thread state data-structure `TState`, on the other hand, records the state information of the constituents of a thread. The important fields of `TState` are as follows:

1. `regs`: This field records the state of the different x86 general purpose hardware registers supported by the implementation. Table 6.1 records these different registers and their configuration. Each thread upon creation receives a exclusive copy of registers which is thread local and is initialized to zeros.

   Similar to the memory, the hardware registers are mappings from the register names to their values, where the value size differs according to the register specification. And unlike the memory, the mappings are allocated statically. Each thread looking to update its register states does so first by accessing its register stack from the program

state, by using its unique thread ID and then by overwriting the required fields of register stack with the modified data.

2. `flags`: The execution of an x86 assembly instruction can have several side effects, one amongst which is the modifications to the flags register. This field records such modifications to the hardware flags of the threads. Similar to the registers, each thread receives a unique copy of the flags, which are reset upon initialization. The flag modifications follow the specifications from the Intel IA-32e architecture and the different flags implemented in the model is shown in table 6.1.

3. `buffer`: In order to support relaxed consistency execution, the memory implementation follows the x86-TSO relaxed consistency model explained in Section 2.2. Consequently, this data-structure is the implementation of the per-thread write buffer, shown in Figure 2.1, where the record keeps track of the buffered per-thread byte-addressable writes to the memory. The functions of this field will be presented in detail in the Section 6.4.

Initial State $\mathtt{s}_0$: From Section 4.1, the state $\mathtt{s}_0 \in \Sigma$ is the initial state of the transition system, which is required to be supplied by the machine model. The initial state of the machine model will be a combination of the state of the state of the program and the execution model. The initial state of the program will be the state of the memory locations that the program initializes. This information is extracted from the data sections of the executable binary. The initial state of the machine model would entail initializing the execution model and the memory. Therefore, the following steps are taken by the execution model in defining the initial state $\mathtt{s}_0$:

- The execution model is initialized with the main thread, which involves initializing the register stack and the heap of the thread.
- The register stack of the main thread is created and the general purpose registers of the thread are initialized to all 0s. The flags of the initial threads are also reset.
- The instruction pointer of the main thread is initialized to the beginning of the program, by reading the "Task State Segment" of the binary.
- The execution heap is initialized with the initialized local and global variables of the program, read from the .data segment of the binary and with the uninitialized variables of the program, read from the .bss section of the binary.
- The arguments for the main function is written to the system heap.
- The thread local variables for the main thread are initialized by extracting the thread local information from the "Thread Local Storage" sections (.tdata and .tbss) of the binary.

Program actions $\mathcal{A}$: Section 4.1 describes $\mathcal{A} = \bigcup_{i=1}^{j} \mathcal{A}_i$ as the domain of all the atomic actions to be performed by the program. The semantics of these actions are necessary to advance the state of the simulation, during program exploration. The concurrency model depends solely on its implementation to provide semantics to the atomic actions executed during exploration. Hence, the machine model, in particular the execution model is responsible for providing semantics to the executed actions.

```
type Instr :
    x86_instr  ::  x86 Instruction
  | pop_buf    ::  Write
```

Figure 6.3: Machine actions type

The responsibility of the execution model is to map the actual instructions performed by a thread on the hardware to the atomic actions expected by the concurrency model. Figure 6.3 depicts this very mapping. The execution model instantiates the atomic actions executed by a thread with *Machine actions* of type `Instr`. A *Machine action* is an instruction that works on the program state of the machine model - thread specific registers and flags and the shared memory.

An atomic action is considered *local* if it accesses *only* the local state $\mathcal{L}$ of its executing thread. Conversely, an instruction is considered *global* if it accesses the shared *global* state $\mathcal{G}$ of the system. An action of type `Instr` can either be an actual x86 hardware instruction (`x86_instr`), or the flush of a single instruction `Write` in the thread buffer (`pop_buf`). The semantics of the x86 instructions are given by the Instruction-Set Architecture *(ISA Model)*, whereas the semantics for a buffered write is given by the *memory model* (TSO in this case). We will discuss in detail about the buffered writes in section 6.4.

ISA Model: Considering x86 machine programs, the domain of actions is the Instruction-Set Architecture. Hence the execution model provides semantics to the x86 instructions though implementing a model of the x86 ISA. The x86 ISA model draws inspiration from [32], [33], by implementing an interpreter-style operational semantics [59] for x86 instructions. Similar to [32], the ISA model concentrates on the 64-bit implementation of the IA-32e architecture (x86-64), with support for all addressing modes. The semantics of each instruction models the transition relation of the concurrency model, by taking as input a program state and returning the appropriately modified new program state.

The concurrency model also defines the requirements for an action to be either *local* or *global*. From Chapter 4, an atomic action is considered *local* if it accesses *only* the local state $\mathcal{L}$ of its executing thread. In similar lines, a machine action `a :: Instr` of a thread is local

if it accesses only the local state of the thread (`regs` and `flags`). Conversely, an action is considered *global* if it accesses the shared *global* state $\mathcal{G}$ of the program. Hence, a machine action `a` :: `Instr` of a thread is considered global, if it accesses information from the memory.

The (enabled) **Function**. From Section 4.1, the function enabled :: $\Sigma \mapsto \{\mathcal{A}\}$ represents the set of atomic actions enabled in a state for execution. Intuitively, given a program state `s`, enabled(`s`) returns the set of actions that can potentially be executed by threads that are not blocked from execution.

The semantics of the enabled function provided by the machine model are as follows. For a given program state `s`,

$$\text{enabled}(\texttt{s}) = \mathit{map}\ (\text{fetch}\ \ \texttt{s})\ \{\mathit{tids}\}$$

where,

- The fetch function, written fetch :: $\Sigma \mapsto \mathit{tid} \mapsto \mathcal{A}$, is used to fetch and decode a *Machine action* of the thread with the given thread ID *tid*. From Figure 6.3, a Machine action can either be an x86 instruction (`x86_instr`), or a buffer `Write` (`pop_buf`). The fetch function can choose non-deterministically, between the two instruction types.

  If the action chosen is an x86 instruction (`x86_instr`), then the fetch function fetches an instruction pointed to by the instruction pointer `rip` of the given thread in the present state. If `rip` of the thread is within the program limit of the current thread, then the corresponding instruction is returned. Else fetch returns $\perp$. If the action chosen is a `Write`, (`pop_buf`), then the fetch function fetches the `Write` at the head of the thread-specific store buffer in the present state. A `Write` can only be fetched if the store buffer is non-empty. The semantics of store-buffering is detailed in Section 6.4.

  However, when the fetch function for a thread returns $\perp$that thread is considered to be blocked from execution. That is, a thread is blocked in a state, if it has neither a (`x86_instr`) or a (`pop_buf`) to execute from that state.

- $\{\mathit{tids}\}$ represents the set of the thread IDs of all the threads in the program, including the main thread

The Transition Relation($\rightarrow$). From Section 4.1, $\rightarrow :: \Sigma \mapsto \mathcal{A} \mapsto \Sigma$ is defined as the transition relation of the transition system $\mathcal{T}$. Intuitively, $\rightarrow$ represents the transitional effects of executing an atomic action from a given state.

The execution model implements $\rightarrow$ as a step function. The step function, written step :: $\Sigma \mapsto \mathcal{A} \mapsto \Sigma$, is the transition function for the execution model, responsible for decoding

and executing an action $a \in \text{enabled}(\mathtt{s})$ from the given program state $\mathtt{s}$. The action to be executed is chosen and fetched by the concurrency model and the step function returns the newly achieved program state upon execution. The action to be executed represents an x86 instruction, whose execution semantics is given by the ISA model.

6.3.2 Implementation of Algorithm Constructs

This subsection details the fulfillment of the requirement **R.2** by the execution model. In addition to the State Automaton constructs of the concurrency model, as mentioned in Section 5.4, the algorithm utilizes several functions, described by the concurrency model, to decide and advance the course of state-space exploration. Hence, it is necessary for the algorithm implementation to provide implementation semantics to the following constructs.

Order-Sensitivity: From the Definition 4.3.3, we have understood the importance and the need for identifying order-sensitivity in execution traces. However, from Section 5.3, we have also understood the redundancies involved in identifying order-sensitivity in traces, by following the definition strictly. Hence, the execution model provides an over-approximation function for identifying order-sensitivities in an execution trace. The execution model implements Order-sensitivity, which is a property over a trace, using the following function:

Pre-requisites: We will now setup the pre-requisites for understanding the over-approximation function:

- Let $\tau = \tau_1 \cdot \tau_2$ be a valid execution trace, where $\tau_2 = [\mathtt{a_i}, \mathtt{a_{i+1}}, \ldots, \mathtt{a_j}]$ represents an order-sensitive trace after the valid trace τ_1. Let $\mathcal{S}(\tau_2) = [\mathtt{s_i}, \ldots, \mathtt{s_{j+1}}]$ be the states of execution of τ_2.
- The actions $[\mathtt{a_i}, \mathtt{a_{i+1}}, \ldots, \mathtt{a_j}]$ are of type `Instr`and therefore access either the program registers or the memory.
- Let $\mathcal{R}_i = \{r_i^1, \ldots, r_i^n\}$ and $\mathcal{R}_j = \{r_j^1, \ldots, r_j^m\}$ be the byte-addressable memory locations accessed by the actions $\mathtt{a_i}$ and $\mathtt{a_j}$ respectively, where $\mathcal{R} = \mathcal{R}_i \cap \mathcal{R}_j$ represents the set of memory locations shared between the two actions $\mathtt{a_i}$ an $\mathtt{a_j}$.

Definition 6.3.1. ***x86-order-sensitivity.*** *The order-sensitivity in τ_2, after the execution of τ_1, can be over-approximated by the* x86-order-sensitivity relation*,*

notation x86-order-sensitive(τ_1, τ_2)*, if the predicate* $\mathrm{P}_{\text{x86-os}}(\mathtt{s_i}, \mathtt{a_i}, \mathtt{s_j}, \mathtt{a_j})$ *holds. The predicate* $\mathrm{P}_{\text{x86-os}}(\mathtt{s_i}, \mathtt{a_i}, \mathtt{s_j}, \mathtt{a_j})$ *holds if* $\mathcal{R} \neq \emptyset$ *and*

$$
\begin{array}{rll}
if\ \exists r \in \mathcal{R}\ . & (1)\ \ \text{write}(\mathtt{a_i},\ \mathtt{s_i},\ r,\ val)\ \wedge\ val' = \text{read}(\mathtt{a_j},\ \mathtt{s_j},\ r) & \\
& \vee\ (2)\ \ val' = \text{read}(\mathtt{a_i},\ \mathtt{s_i},\ r)\ \wedge\ \text{write}(\mathtt{a_j},\ \mathtt{s_j},\ r,\ val) & \Big\} \implies val \neq val' \\
& \vee\ (3)\ \ \text{write}(\mathtt{a_i},\ \mathtt{s_i},\ r,\ val)\ \wedge\ \text{write}(\mathtt{a_j},\ \mathtt{s_j},\ r,\ val') & \Big\} \\
& & else \implies False
\end{array}
$$

Where

- write($\mathtt{a}$, $\mathtt{s}$, r, val) *represents the write of the byte value val to the memory location addressed by r, by the action* $\mathtt{a}$ *in the state* $\mathtt{s}$*.*
- $val = \text{read}(\mathtt{a}, \mathtt{s}, r)$ *represents the read of the byte value val from the memory location addressed by r, by the action* $\mathtt{a}$ *in the state* $\mathtt{s}$*.*

Note that action $\mathtt{a_i}$ is the first action in τ_2 and action $\mathtt{a_j}$ is the last action in τ_2. The over-approximation function stems from the intuition behind order-sensitivity: choosing actions to execute can influence the final states reached from the execution. The cases of the function represent this intuition as follows:

- Case (1): Reading the value val' from the memory region r at the end of τ_2 indicates that the region r is restricted with the data val' after τ_2 and data val' will henceforth flow to any instruction that reads from r. Moreover, if $val \neq val'$ then the value val would not be reachable to instructions after τ_2 that read from the memory region r. This constitutes an order-sensitivity, as choosing to execute action $\mathtt{a_i}$ at the beginning of τ_2, prevents the data val from *flowing* to instructions after τ_2. Moreover, executing the action $\mathtt{a_j}$ before $\mathtt{a_i}$ *would* allow the data val to reach to instructions after τ_2. Therefore, the final states reachable from τ_2 are *sensitive* to the *order* of execution of actions $\mathtt{a_i}$ and $\mathtt{a_j}$.
- Case (2): Case (2) is the complement of case (1). Delaying the writing of data val to region r until the end of τ_2 restricts the data val to *flow* to any instructions henceforth that read from region r. Moreover, this prevents the value val' from reaching to any states after τ_2. Therefore, τ_2 constitutes order-sensitivity in this case as choosing to execute action a_j before $\mathtt{a_i}$, after τ_1, can uncover final states after τ_2 that read the data val' from r.
- Case (3): The reasoning here is similar to the other two cases. Case (3) represents an order-sensitivity in τ_2 since choosing to execute action $\mathtt{a_j}$ at the end of τ_2 restricts the data val' to *flow* to any instructions henceforth that read from region r. Moreover, this prevents the value val from reaching to any states after τ_2. Therefore, τ_2 constitutes order-sensitivity as choosing to execute action a_j before $\mathtt{a_i}$, after τ_1, can uncover final states after τ_2 that read the data val from r.

6.3.3 Providing semantics for the POSIX thread library APIs

This subsection details the fulfillment of the requirement **R.3** by the execution model. One of the major goals of the verification effort was to support dynamic thread creation in a realistic setting. One choice for modelling thread creation is to construct a simplistic abstraction of the thread creation model to spawn dynamic threads. However, the assembly language is a semantically rigid language that requires accurate environment initialization for sound execution.

Threads created in the assembly typically exhibit behaviors that are not readily apparent at the source code. For example initialization of the Thread Local Storage image [60] during thread creation, or verifying the canary bit for buffer overflow detection. Therefore, employing an abstract model of the thread creation library would, depending on the level of abstraction, require pre-processing and modification of the assembly to remove such artifacts. This could not only render the code potentially unsound, but also abstract out several verification opportunities.

Implementing a model that emulates the POSIX library thread creation semantics allows the execution model to consume mainstream code without any form of pre-execution analysis or refactoring of the assembly. We chose to implement the POSIX thread library as it is one of the more well-documented concurrency libraries with Linux repositories providing the implementation. In essence, we have provided operational semantics for the following POSIX thread APIs:

1. *pthread_create:* This API call is responsible for spawning a child thread from a parent process. The creation of a new thread involves several steps, of which the important ones are mentioned here; i) To begin with, the new thread is initialized with a new empty set of registers, flags and a store buffer, ii) a thread local space for the new thread is allocated in the memory, iii) the *pthread* struct and the Thread Local Storage (TLS) initialization image [60] is loaded onto the thread local storage. This data section uniquely identifies the new thread, iv) the arguments to the *pthread_create* function call are parsed and the `rip` of the new thread is pointed to the first instruction of the thread program. All of the above steps are executed as an atomic action.

2. *pthread_join*: This API call blocks the execution of the caller thread until the callee thread finishes execution. A thread is considered to have finished execution when its program counter `rip` moves out of the program scope. The execution of a thread with thread ID *tid* is blocked in a state s by forcing $fetch\ s\ tid = \perp$

3. *pthread_mutex_lock* and *pthread_mutex_unlock*: These are the synchronization primitives offered by the pthread library. A thread calls *pthread_mutex_lock* to acquire the lock on a *mutex* object. The caller thread blocks execution if the lock on the mutex

object has already been acquired by another thread. Conversely, *pthread_mutex_unlock* releases the lock on a mutex held by the caller.

4. *pthread_mutex_exit*: This API aborts the execution of a thread by moving the program counter of the caller thread outside the program scope.

6.4 Memory Model

Assembly programs provide a very close view of the underlying execution hardware to the programmer. The composition of such assembly programs contain instructions that assume and interact with the rigid and byte-addressable memory module of the processor. Hence, in order to support sound execution of assembly programs, we have incorporated a byte-addressable memory model to the machine model which is tuned for handling dynamic thread creation. The memory model is represented by the state variable `memory` in the global shared state `GState` and Figure 6.4 represents this abstraction.

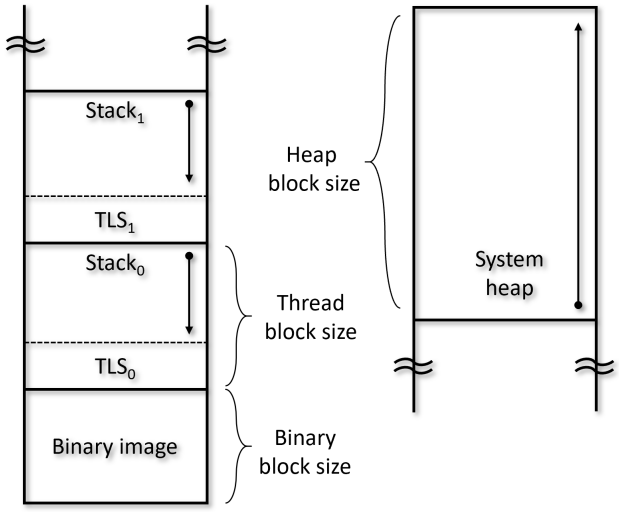


Figure 6.4: Memory block layout

Several considerations were made during the design of the memory model;

- The memory model presented in Figure 6.4 is byte-addressable and the bytes are addressed by 64-bit unsigned integers.
- The intent of the verification framework is to support one program at a time. Hence, in the view of developing a simplified memory model the entire space of the memory model is dedicated to the program under test. This allows us to abstract away memory

management principles such as address translation and virtual memory, which would not influence the soundness of the verification statement.

- Since the entire memory is dedicated to a program, the addresses accessed by the program can be and are considered to be linear in nature.
- During initialization, a part of the memory is reserved for storing the binary. The entire binary is loaded onto this space and the instructions are read from this space for execution.
- Each thread upon creation gets assigned a thread local region of memory. A part of this memory is used to store the TLS initialization image and the thread local variables of the assigned thread.
- All the allocated threads share a common heap space which is used to dynamically manage memory through *malloc*, *calloc* and *free* APIs. Additionally, the memory allocation and reclamation is based on the buddy allocator system [61].
- Each thread upon creation gets assigned a fixed stack space, allocated in the thread local memory region. The stack space is individual to the thread and the utilization can dynamically vary during the program execution.

x86-TSO Semantics. Another major goal of this verification effort was to incorporate relaxed consistency execution, to provide a realistic setting for assembly program verification. Hence, we have augmented the memory model with the x86-TSO operational semantics presented in Section 2.2. The TSO modelling draws inspiration from that by [28], [29], with additional modelling to support the byte-addressable memory of x86. We will now detail the implementation of the x86-TSO operation semantics defined in Section 2.2.

1. **Store implementation:** Consider a thread p_i in the program that executes an instruction intending to update a contiguous piece of the `memory`. The data to be stored is broken down, in Little-Endian fashion, into an array of key-value pairs. The keys are memory locations and the values are the corresponding bytes to be written and this key-value pair array is represented by a `Write` data-structure. The `Write` action is then immediately enqueued at the tail of the store buffer of p_i, before the execution of the next action.
2. **Load implementation:** A x86 instruction intending to read data from the memory will specify the size of the data to be read, through width specifiers. Hence, a memory read instruction of a thread p_i can be broken down into a series of memory accesses. To execute the read, the buffer of thread p_i is first checked for the latest value for any of the memory locations in the read. If the buffer does not contain the value for any locations, the read request for such locations are forwarded to the memory.

3. **Update implementation:** The Update operation corresponds to executing the `pop_buf` Machine action, detailed in Figure 6.3. Update operation from a thread p_i first dequeues a `Write` from the buffer head of the thread. The contents of the dequeued `Write` are then flushed to the memory in a single atomic action. An Update is considered as an action of the thread that enqueued the corresponding `Write`. The Update can occur non-deterministically at any state of execution but only after the `Write` has been enqueued onto the store buffer. Hence, by design, Updates from a thread follow the program order whereas no ordering is imposed between Updates by different threads.

4. **Fence implementation:** We implement a Fence operation by flushing the entire store buffer of the thread executing the action. The Fence instruction is always of type *x*86_*instr*, as its semantics is provided by the x86 ISA. During the execution of a Fence, the corresponding thread is blocked from executing x86 instructions (`x86_instr`) until the thread specific store buffer becomes empty. That is, during a Fence, the fetch function for the corresponding thread will only fetch `pop_buf` machine actions until the thread-specific store buffer is empty. This will force program ordering between actions before and after the Fence.

5. **LOCK implementation:** As mentioned in earlier sections, the emulation of a concurrent x86 program is performed as a linearization to a single threaded execution. Hence, there can only be one action chosen for execution at each state transition which makes the implementation of LOCK semantics straightforward: When a thread p_i encounters a LOCK'd instruction, the instruction semantics are first executed. Upon completing the instruction execution, the entire store buffer of p_i is flushed to the memory which renders the LOCK'd instruction atomic.

We will now present an example of store-buffering.

Example 6.4.1. *(store buffering). Consider a thread p_i executing a* `x86_instr` *atomic action, representing a* `mov` *x86 instruction that uses register-indirect addressing to write a piece of data to the memory.*

```
mov  DWORD PTR [RAX], 0xDABBAD00
```

The `mov` *instruction presented intends to move the 32-bit hexadecimal data* `0xDABBAD00`*, to a memory region whose address begins with the value in* `RAX`*. Let the value in* `RAX` *be* `r :: 64Word`*.*

From the semantics of x86-TSO, the execution of the above memory write will be buffered through the store buffer of p_i. To buffer the memory write, the data to be written broken-down into a list of key-value pairs, represented by the `Write` *data-structure. The data is broken down in little-endian format, as shown below.*

$$\texttt{Write w} = [(\texttt{r}, \texttt{0x00}), (\texttt{r} + 1, \texttt{0xAD}), (\texttt{r} + 2, \texttt{0xBB}), (\texttt{r} + 3, \texttt{0xDA})]$$

The memory locations associated with w are $\{r, r+1, r+2, r+3\}$. The buffer entry $\mathtt{w_1}$ *is then enqueued at the tail of the write buffer of p_i.*

Side-effects of the TSO model: Augmenting the memory model with TSO semantics brings about several interesting side-effects to the constructs of the execution model. We will now detail all such side effects:

1. `pop_buf` *machine actions are* always *Global.*
   This is true, since a `pop_buf` machine action of a thread involves flushing the corresponding `Write` to the memory. Since executing the action results in accessing the `memory` field of the shared `GState`, `pop_buf` actions are rightfully *Global.*

   On a side note, the set of memory regions accessed by a `pop_buf` is defined by all the memory addresses updated by the corresponding `Write`.

2. *A* `x86_instr` *will* never *directly update the memory.*
   This is true because, whenever a `x86_instr` intends to store its result in the memory, the corresponding store will always be store-buffered as a `Write`. Hence the execution of a `x86_instr` machine action will never directly update the memory.

3. *A Fence instruction is* always *Global.*
   A memory Fence is associated with (potentially) several `pop_buf` machine actions. Therefore, the execution of a memory Fence will always result in manipulating the shared state of the program. Hence, a Fence will always be a *global* action.

   On a side note, the set of memory regions accessed by a Fence corresponds to the union of the memory addresses updated by each `pop_buf` during the Fence.

4. *A LOCK'd* `x86_instr` *is* always *Global.*
   The LOCK prefix is always associated with a RMW instruction, whose execution will always enqueue at-least one `Write` action onto the per-thread write buffer. Therefore, the execution of a LOCK'd `x86_instr` will always result in manipulating the shared state of the program. Hence a LOCK'd `x86_instr` will always be global.

   On a side note, the set of memory regions accessed by a LOCK'd `x86_instr` corresponds to the union of the memory addresses updated by each `pop_buf` while flushing the thread-specific store buffer.

5. *A* `x86_instr` *without a LOCK prefix will be Global* if and only if *it performs a Load from the memory.*
   Since any `x86_instr` will never directly update the memory, the only way an `x86_instr`

will be *Global* is if it reads from a memoy location. Thus, a `x86_instr` will be considered *Global* only if it loads data from a memory location, irrespective of whether the data came directly from the memory or from a corresponding `Write`

6.5 System-Call Model

The aim of this verification effort is to deterministically and efficiently explore all possible thread interleavings in a multi-threaded program. However, user-level programs can encounter non-deterministic executions while requesting services from the underlying Operating System through performing system calls. User-level concurrent programs can often execute system calls such as setting CPU affinity or file management that have little or no impact on the verification statement. However, such programs may also employ system call services such as memory management (memory barriers) and synchronization (futex, mutex), which can have an impact on the interleavings observed during execution. Therefore, we have extended the machine model with a system call framework that achieves two goals: i) recognizes and avoids the execution of system calls that do not influence the verification statement, thereby reducing the non-determinism during execution and ii) provides an extensible framework for implementing semantics of system calls that can influence the concurrency behavior of the program under verification.

The system call framework is built upon providing execution semantics for the `syscall` instruction, since it is one of the most efficient and versatile instructions that x86 programs use to invoke system calls. The system call framework draws inspiration from [32]. While [32] defers the system call execution to the underlying OS, the framework provides the execution semantics for chosen system calls. At a high level, the system calls in x86 are invoked in two steps. The first step corresponds to the userland program loading the thread-specific RAX register with the appropriate system call number and calling the `syscall` instruction. The execution of the `syscall` instruction results in the second step, where the kernel level *system_call* method loads the `rip` of the user thread with the address of the appropriate system call. The system call framework works by interrupting execution in the second step and provides the relevant execution semantics for the requesting system call which are crafted based on their corresponding Linux OS implementations. The execution of a system call is performed as a single atomic action.

The decision to provide operational semantics for system calls come with several advantages; i) The system call execution becomes deterministic. Deferring the system call execution to the underlying OS can result in non-determinism, where each run can yield different results. For example, a futex wake operation on multiple threads can result in waking different threads in different runs, which makes the state space exploration non re-playable. Hence embedding the operational semantics into the system call framework makes the state

exploration deterministic and re-playable. ii) Redundant system call executions and thereby redundant trace explorations can be avoided. For example, repeated wake calls to a futex (syscall 202) can result in the OS waking the same sleeping thread every time, which can lead to initiating redundant explorations. Embedding the operational semantics into the system call framework allows for a better control on the system call execution. iii) Avoids execution of non-essential system calls. The framework allows complete control over defining the operational semantics for system calls. Hence, the semantics of calls whose execution do not have an impact on the system state can be made a pass-through.

Chapter 7

Experimental Results

This chapter will be used to detail and discuss upon the empirical results used to evaluate our work. The organization of the chapter is as follows. To begin with, in Section 7.1, we will introduce the different metrics and testcases that been used to evaluate the work, which entails demonstrating the results and discussing the initial set-up required to achieve those results. Subsequently, we will provide discussions on verified results in Section 7.2, discussions on un-verified results in Section 7.3, and discussions on unsupported results in Section 7.4. We will begin by introducing our empirical results.

7.1 Case Studies

We will now present the several case studies that we have used to evaluate our work. We have applied our approach to several racy benchmarks, with the goal of testing the usability and soundness of the implementation under various levels of contention. The source code for all of the case studies have been written in c and used the pthread model for multi-threading. Moreover, no pre-processing has been done to the source code to make it amenable to our tool. In all of our experiments we have used a Linux machine with Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz and 16GB of RAM. We use the tool Objdump, in Linux to extract the machine-code representation of the source code, however all of the initial and run-time state information is extracted from the binary.

One of the main aims of our work has been to apply our approach to verify production ready machine-code. In order to demonstrate this aim, we have divided the case studies into two buckets: i) the more classical benchmarks such as Peterson's, Dekker's and Lamport's mutual exclusion algorithm, concurrent Fibonacci series and Craig, Landin, and Hagersten CLH queue lock; and ii) to showcase our implementation's applicability on production ready

code, we have consider several concurrent data-structures such as lock-free and wait-free queues, stacks and hash-maps from the Userspace RCU code.

Classical benchmarks. The classical benchmarks serve the purpose of demonstrating the soundness of our approach, more-so than its applicability. We have chosen these benchmarks as they contain mutual-exclusion case studies from the literature. In particular, we have chosen the following benchmarks:

- `Dekker` (`DEK`) and `Lamport` (`LAM`) benchmarks each spawn two threads, which compete to write to a shared variable once.
- `Peterson` (`PET`) benchmark spawns two threads, each of which compete to write to a shared variable multiple times.
- `Fibonacci` (`FIB`) and `CLH Lock` (`CLH`) benchmarks each can spawn multiple threads, and use different locking mechanisms to synchronize writes to a shared variable.

The behavior of these case studies are easier to comprehend as they avoid races explicitly with locks, and the intent of these algorithms are well documented. Hence they present a perfect opportunity to demonstrate the soundness of our algorithm.

URCU benchmarks. We have chosen to represent several data-structures from the Linux User-space Read-Copy Update (URCU) repository[1], in order to demonstrate the applicability of our algorithm on a more mainstream and involved set of benchmarks. RCU is one of the more heavily used lock-free synchronization mechanisms, with uses in several components of the Linux kenel. The Linux URCU is similar to its kernel-side counterpart in functionality. It is extensively used by several user-land code projects, and is readily available in several mainstrean Linux distributions. In particular, we have considered the following benchmarks:

- `Wait − free Stack` (`WFS`) is a concurrent URCU data-structure that provides wait-free pushes and blocking pops. In our benchmark, the push and pop operations are performed by different producer-consumer threads.
- `Lock − free Stack` (`LFS`) is a concurrent lock-free URCU data-structure that provides lock-free pushes and concurrent blocking pops.
- `Lock − free Queue` (`LFQ`) is a concurrent lock-free URCU data-structure that provides lock-free concurrent enqueue and dequeue. In our benchmark, the enqueue and dequeue operations are performed by different producer-consumer threads.

[1]The repository can be found at git://git.liburcu.org/userspace-rcu.git

- Lock − free Hash Table (LFH) is a concurrent lock-free URCU data-structure that provides lock-free concurrent insert, delete and find operations. In our benchmark, the insert/delete and find operations are performed by different producer-consumer threads.

RCU (as well as URCU) serves as a lock-free replacement for concurrent reader-writer locking scenarios. RCU optimizes the read side critical sections since the reader threads are not required to directly synchronize with the writers, and thereby allowing the readers and writer threads to progress concurrently. Hence, compared to the classical benchmarks, the design of the URCU data-structures are involved and non-trivial. Therefore, verifying RCU through our approach will demonstrate the applicability of our implementation. Table 7.1 will document our results of verifying the above algorithms through our implementation.

Benchmark	LOA	LOA_{PT}	Traces	States	Time	Status
Classical Benchmarks						
DEK(2)	251	39	2	156	0.24	Verified
LAM(2)	227	24	2	103	0.21	Verified
PET(2)	293	130	4	872	1.08	Verified
FIB(5)	267	46	5	2798	5.57	Verified
FIB(7)	267	46	7	11789	104.3	Verified
FIB(9)	267	46	9	42477	1261.3	Verified
CLH(4)	310	50	4	7423	47.56	Verified
CLH(6)	310	50	6	21687	396.5	Verified
CLH(8)	310	50	8	59882	2186	Verified
URCU Benchmarks						
WFS(1, 1)	989	542	2	1814	2.6	Verified
WFS(1, 2)	989	542	3	11383	64.23	Verified
WFS(2, 1)	989	542	4	53134	868.42	Verified
LFS(1, 1)	683	220	6	1708	1.88	Verified
LFS(1, 2)	683	220	31	21482	41.09	Verified
LFS(2, 2)	683	220	154	$>$ 125k	1953	Verified
LFQ(2, 1)	3721	2183	–	–	–	CEX
LFH	–	–	–	–	–	Unsupported

Table 7.1: Experimental results

Evaluation. We evaluate the results of our experiments using five empirical metrics and two status metics, which are listed in the columns of Table 7.1. Each benchmark name is annotated with the number of threads being spawned dynamically (excluding the main thread). The URCU benchmarks are annotated with the producer and consumer thread numbers. The empirical metrics are described as follows

1. Lines of Assembly (`LOA`) : The total number of Lines of Assembly in the given benchmark. This metric is important as it can allude to the complexity of the program. However, note that the total number of `LOA` may not necessarily indicate the actual number of machine-code instructions actually executed, as instructions can be executed multliple times by different threads.

2. Lines of Assembly per thread ($\texttt{LOA}_{\texttt{pt}}$) : The average number of Lines of Assembly, *per thread*, in the given benchmark. This metric can indicate the number of `LOA` that a dynamically spawned thread will execute, on an average, in a maximal trace. Note that this metric does not subsume the `LOA` executed by the main thread.

3. `Traces` : The total number of maximal traces reached during the program execution. This metric is important as it can help decide the soundness of our implementation, and help analyze the final states reached during the program execution.

4. `States` : The total number of program states reached during the program exploration. This metric can help indicate the complexity of both the program and our approach.

5. `Time` : The time taken for the complete program execution. The execution time can depend on several factors; i) Internal factors such as the efficiency of implementation, program complexity, etc, that are system characteristics, and ii) External factors such as processor capacity, system memory availability, etc., that are extraneous to the implementation. Hence, the time metric is recorded as an average over several runs, in order to account for the external factors.

6. `Status` : Every benchmark is also annotated with the status of the verification activity. Status `Verified` denotes that the benchmark was completely verified by the implementation, by successfully uncovering all final states. Moreover, `Verified` indicates that there were no assertion failures found in the course of exploration. Status `CEX` (Counter Example) denotes that a counter example was found during the course of the exploration. A `CEX` is indicated by an assertion failure, which typically signals the violation of a program property. Status `Unsupported` denotes a benchmark is not supported by the implementation.

7.1.1 Program setup

Several steps were taken in configuring above benchmarks for the evaluation, which we will detail as follows. The configuration parameters can be divided into three types: i) multi-threading configuration, ii) compilation and iii) the configuration of the data shared between the threads. The configuration of these parameters were chosen specifically to both demonstrate the applicability of our approach to real-world problems and also to facilitate a conducive description.

Pre-processing and compilation. As mentioned in earlier chapters, our approach does not require any pre-processing to be done on the source code. Hence all of the benchmarks are compiled as-is, without any structural modifications or translations. We use the GNU Objdump v2.30 tool to disassemble the object file, to extract the machine-code. Moreover, one of the requirements of our approach is for the program code to be available statically during execution. Therefore, we have taken the following steps in compiling the benchmarks for the evaluation. For all the benchmarks, we have used GCC v7.5.0 for compilation, with O2 optimization levels.

1. For the classical benchmarks, since all of the execution code is standalone and contained in the source file, the benchmarks are linked statically to the executable. Since the machine model provides the semantics for the POSIX threads APIs, the pthread library is linked dynamically to the executable.

2. The URCU benchmarks rely upon the *liburcu* libraries to provide semantics for the RCU constructs. Hence, during the compilation of the URCU benchmarks, the necessary URCU libraries are linked statically to the executable. However, since the machine model provides the semantics for the POSIX threads APIs, the pthread library is linked dynamically to the executable.

Multi-threading configuration. Every benchmark is parameterized by the number of threads dynamically spawned during the course of its execution. The classical benchmarks have a single count that represents the total number of dynamically spawned threads during execution. Whereas, the URCU benchmarks are annotated with the individual thread counts of the readers and the writers. Wherever possible, the benchmark metrics have evaluated with varying thread count. Trivial case studies like the Peterson, Lamport and Dekker mutual exclusion benchmarks are evaluated for only two threads as they are specifically designed to support only the case. However, we have evaluated the rest of the benchmarks for multiple thread counts, to show both the applicability of our approach and the complexity scaling with increasing thread count.

7.2 Discussion on Verified results

Table 7.1 lists the several benchmarks that we have used to quantify our results. The table also lists the status of our experiments on each benchmark, as either "`Verified`", "`Unverified`" or "`Unsupported`". We will now discuss about the results that have been marked as "`Verified`" in the table. We use the term "`Verified`" to denote that our implementation was able to explore traces that reach all unique final states of the program. Correspondingly, the "`Traces`" column reports the traces explored by the implementation

to achieve the maximal states.

Classical benchmarks. We will first consider the classical benchmarks. For `DEK` and `LAM` benchmarks, the results are straightforward. The benchmarks contain only two threads, competing to write unique data to a shared variable. Hence, there can only be two orderings of the threads, leading to two final states, which is exactly what the implementation finds.

The `FIB` and `CLH` benchmarks, on the other hand, can spawn multiple threads (> 2). Hence, the results are non-trivial. For example, the `FIB` benchmark spawns multiple threads that individually calculate the Fibonacci series for different indices, and write the result concurrently to the same shared variable. Therefore, the number of final states reachable will be equal to the number of unique results calculated. Table 7.1 depicts this outcome.

URCU benchmarks. In contrast to the classical benchmarks, the analysis of the URCU benchmarks are non-trivial. This is so because the URCU benchmarks achieve synchronization between concurrent threads through non-trivial means. The URCU benchmarks typically achieve mutual exclusion through atomic Compare and Swap operations, which are non-blocking. Therefore, such executions can often lead to several unique final states, that differ only in the individual thread states and corresponding executions.

The threads in the URCU benchmarks behave in a producer-consumer fashion. Hence, we have evaluated the benchmarks with varying producer and consumer threads. For the verified benchmarks, we use the single producer and single consumer case to establish the base case. For the stack data-structure benchmarks, the producer threads try concurrently to push integers from a two-element array onto the stack, while the consumers try to pop the elements concurrently from the stack. Similarly, in the queue data-structure, the producers try to enqueue integers from a two element array onto the queue, whereas the consumers then try to concurrently dequeue the elements from the queue.

7.3 Discussion on Unverified results

In the preceding sections, we have presented arguments that verify the soundness of our algorithm . However, it is also in our interest to demonstrate that our implementation is capable of catching issues in the program being verified. Hence, we will discuss about one such case study (`LFQ`) in this section, wherein a functional assumption was intentionally violated in the program, and our implementation was successfully able to uncover this issue. Introducing the assumption violation, and subsequently reproducing the illegal scenario provides added confidence over the applicability of our implementation. We have recorded the status of this benchmark in Table 7.1 as `Unverified` since our implementation stops program exploration

upon encountering an assertion violation.

The Counter-Example. The issue uncovered in `LFQ` is a null-pointer dereference, and is brought about by concurrent threads being able to access the intermediate states of other threads, over the course of the data-structure manipulation. Before detailing the `CEX`, we will mention some pre-requisites:

R.1 Recall that `LFQ` is a lock-free data-structure. Hence the concurrent queue manipulations are performed through Compare and Swap (CAS) operations, rather than through acquiring Spin Locks [62].

R.2 The queue is required to contain reference to at least one object at all times. When the queue is empty, it will contain reference to only the Sentinel node.

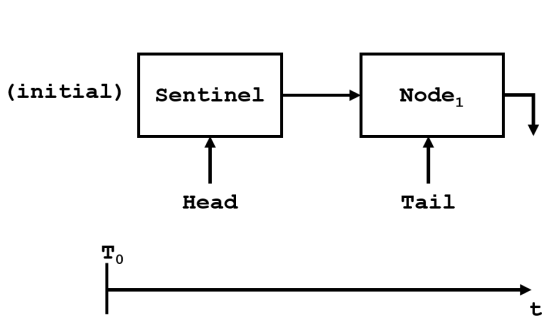
R.3 The queue maintains separate references to the Head and Tail of the structure. An enqueuer thread is allowed to manipulate only the Tail of the structure, as new nodes are enqueued at the end of the structure. Conversely, a dequeuer is allowed to manipulate only the Head of the structure, as nodes are dequeued from the beginning of the structure.

R.4 The first node to be dequeued form the structure will always be the Sentinel. Consequently, a dequeuer method will fail if the queue is empty. Moreover, if the queue has a single node, a dequeuer must enqueue a new Sentinel node before progressing.

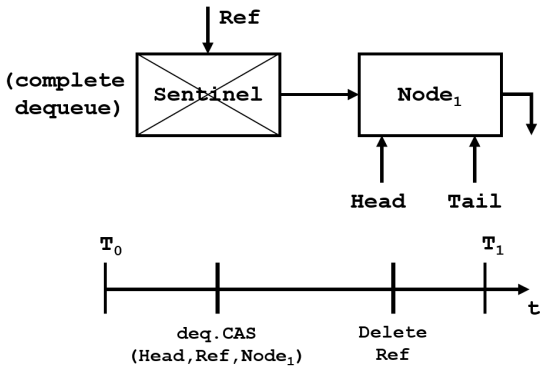
R.5 There are no restrictions for an enqueuer to enqueue a node onto the structure.

Figure 7.1 demonstrates the `CEX` found in the `LFQ` benchmark. The issue manifests through an intricate interleaving between the threads, the steps of which are detailed as follows.

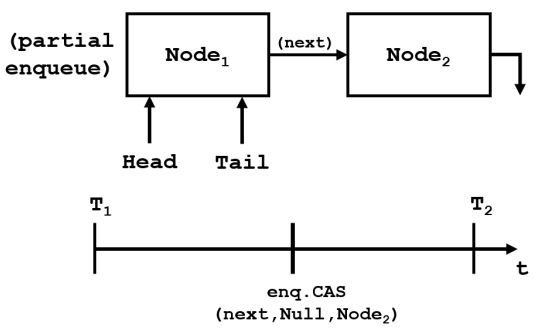
- **Initial step:** The queue for the `CEX` starts with a single node, as shown in Figure 7.1a. There are two threads concurrently working on the queue, an enqueuer and a dequeuer. The queue contains a Sentinel node, which the `Head` pointer points to. The single node, $\texttt{Node}_1$, is chained to the Sentinel, and is referenced by the `Tail` pointer.
- **Step 1:** The dequeuer takes the first step by trying to dequeue a node form the queue. From requirement **R.4**, the first node to be dequeued will be the Sentinel, and the dequeuer successfully does so. The timeline in Figure 7.1b shows the linearization of the dequeue method. It has to be noted that the dequeuer has not dequeued an actual node yet.
- **Step 2:** Since the dequeuer is not holding any mutexes on the structure (requirement **R.1**), a concurrent enqueuer enters the picture, to enqueue a new node. The enqueuer is now witness to an intermediate state of the dequeuer, but is unaware of



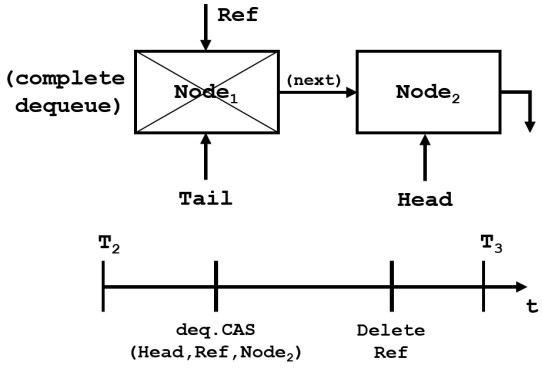
(a) Initial Step



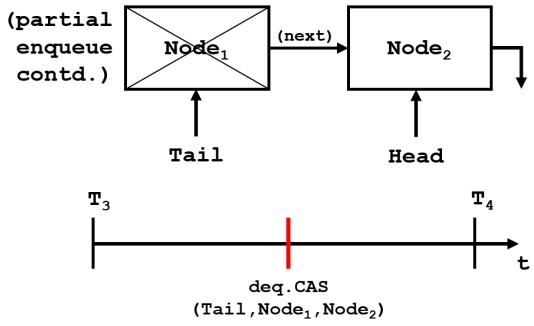
(b) Step 1



(c) Step 2



(d) Step 3



(e) Step 4

Figure 7.1: `LFQ` CEX replay

the fact as the enqueuer works only on the `Tail`. The enqueue gets partially completed, as the enqueuer succeeds in attaching the new node to the end of the structure. However, it has to be noted that the enqueue is partial, as the `Tail` of the structure has not been updated yet.

- **Step 3:** Now the deqeuer resumes its execution. Before dequeuing the first node, it checks if the node is a singleton in the queue. This is not the case, as the enqueuer had enqueued a new node in the previous step. Hence, the dequeuer does not enqueue a new Sentinel, and proceeds to successfully dequeue the node. The timeline for this method is presented in Figure 7.1d. However, it has to be noted here that since the enqueue was partial, the `Tail` of the structure now points to a dequeued and deleted node.

- **Step 4:** Finally, the enqueuer resumes to complete its method. It attempts to do so by correcting the Tail reference to point to the new enqueued node. This involves testing the current reference of the `Tail`, which points to a deleted node. Hence, this step leads to a null-pointer dereference.

The Assumption. As mentioned previously, the issue caught is not a real concurrency bug in URCU, as it violates a fundamental assumption of RCU: The Grace-period guarantee[2] [63]. RCU preserves data integrity across readers and updaters, by making sure that the updaters wait for all pre-existing readers to complete their critical sections, before updating the data-structure. This period of waiting is called the *grace period*, which is achieved by either blocking the writers or by having the writers register a callback which responds after the readers have finished their critical sections. In order to introduce the above issue, our benchmark intentionally violated the grace period guarantee, by allowing an updater to update the structure, even when a reader is holding reference to the structure constituents. This can result in readers holding references to potentially stale data.

7.4 Discussion on Unsupported Results

One of the main limitations of our approach is dynamic compilation. This is so, because the command parser in the machine model currently requires all instructions and memory to be statically present in the object file. Therefore, our implementation cannot support programs containing libraries that cannot be statically compiled The Lock-free Hash table, presented in Table 7.1 is one such example, which depends on shared memory from other libraries, that can only be located during run-time. Therefore, the example is marked as unsupported in the table.

[2]More details about the requirements can be found at: https://www.kernel.org/doc/Documentation/RCU/

Chapter 8

Conclusion and Future work

8.1 Conclusion

The thesis presents Xavier, a novel tool-set for efficient model-checking of concurrent x86 machine-code. To that end, the thesis has presented two primary contributions: The first contribution is a novel Order-Sensitivity based dynamic partial-order reduction algorithm, that efficiently explores and model checks a program under a stateful setting. The exploration is provided through dynamically recognizing and efficiently replaying order-sensitive traces. The model checking is provided by evaluating safety properties, via assertions, in every exploration state reached. The second contribution is the x86 machine model, that supports execution of concurrent x86 machine-code. Moreover, the machine model incorporates the x86-TSO memory model, for relaxed consistency execution. Therefore, the exploration algorithm, in combination with the machine model provides a realistic environment for simulating and model checking concurrent x86 machine-code.

The purpose of our approach has been to provide a platform for verifying production-ready machine-code, through identifying safety issues via assertion failures. Our implementation is capable of detecting bugs in machine-code compiled from production-ready C programs, that utilize the POSIX thread library to achieve concurrency. Moreover, our implementation is capable of evaluating the compiled machine-code directly, without requiring any special translations, and the results of our methodology are independent of the optimizations imposed by the compiler.

We have evaluated our tool set against several classical benchmarks and data-structures from the production-ready URCU library. The results of our experiments demonstrate the applicability of our approach, where the benchmarks achieve concurrency through both explicit methods (acquiring spin-locks) and through implicit methods (atomic operations such as

CAS). Our approach was successful in proving correctness of all the supported benchmarks, and as a soundness guarantee, was also able to uncover a counter-example representing the assumption violation in the `LFQ` benchmark.

8.2 Future Work

The contributions presented in this work provides good scope for future work. Chapter 7 presents the evaluation of our work on both classical and production-ready benchmarks. The results from the evaluation are sufficient to demonstrate the applicability of our tool-set. However, the results also present a current limitation of our approach - dynamic compilation. Hence, one of the future goals for this project would be to support programs that can be dynamically compiled.

One of the main features of the work is the ability to detect illegal program behaviors via failing assertions. However, this feature has a subtle limitation. While the implementation can detect generic structural issues such as null-pointer de-references or exceptions in the program, the detection of complex behavioral issues depends on the presence of adequate assertions in the program. Therefore, one avenue for future work would be to extend the dynamic instrumentation to support the on-the-fly addition of assertions.

Another possible future work is to increase the number and types of x86 instructions supported. Increasing the supported instruction count can increase the chances of a program being supported by our implementation. The current implementation supports instructions only with integer operands. Hence, one possible avenue for future work would be to support instructions that consume floating point operands. The implementation is designed as a framework where new instruction semantics can be added without affecting the soundness of previous semantics. Therefore, we consider increasing the instruction count as an engineering effort.

References

[1] P. Godefroid, "Model Checking for Programming Languages Using VeriSoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '97, Paris, France: Association for Computing Machinery, 1997, pp. 174–186, ISBN: 0897918533. DOI: 10.1145/263699.263717.

[2] E. M. Clarke Jr., T. A. Henzinger, H. Veith, and R. P. Bloem, *Handbook of model checking*, English, Cham, Switzerland, 2018. DOI: 10.1007/978-3-319-10575-8.

[3] 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*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 267–280. DOI: 10.5555/1855741.1855760.

[4] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, "State Space Reduction Using Partial Order Techniques," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, Nov. 1999, ISSN: 1433-2779. DOI: 10.1007/s100090050035.

[5] P. Godefroid, "Using partial orders to improve automatic verification methods," in *Computer-Aided Verification*, E. M. Clarke and R. P. Kurshan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 176–185, ISBN: 978-3-540-38394-9. DOI: 10.1007/BFb0023731.

[6] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper, *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer-Verlag Berlin Heidelberg, 1996, vol. 1032, ISBN: 978-3-540-60761-8. DOI: 10.1007/3-540-60761-7.

[7] V. Le, M. Afshari, and Z. Su, "Compiler Validation via Equivalence modulo Inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 216–226, ISBN: 9781450327848. DOI: 10.1145/2594291.2594334.

[8] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, San Jose, California, USA: Association for Computing Machinery, 2011, pp. 283–294, ISBN: 9781450306638. DOI: 10.1145/1993498.1993532.

[9] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: a Tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996, ISSN: 1558-0814. DOI: 10.1109/2.546611.

[10] S. Owens, S. Sarkar, and P. Sewell, "A Better x86 Memory Model: x86-TSO," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 391–407, ISBN: 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_27.

[11] R. Kumar, E. Mullen, Z. Tatlock, and M. O. Myreen, "Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB," in *Interactive Theorem Proving*, J. Avigad and A. Mahboubi, Eds., Cham: Springer International Publishing, 2018, pp. 362–369, ISBN: 978-3-319-94821-8. DOI: 10.1007/978-3-319-94821-8_21.

[12] R. Morisset, P. Pawan, and F. Zappa Nardelli, "Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 187–196, ISBN: 9781450320146. DOI: 10.1145/2491956.2491967.

[13] X. Leroy, "The CompCert C Verified Compiler: Documentation and User's Manual," Inria, Intern report, Sep. 2019, [Online]. Available: https://hal.inria.fr/hal-01091802/file/manual.pdf, pp. 1–78.

[14] F. Verbeek, J. A. Bockenek, and B. Ravindran, "Highly Automated Formal Proofs over Memory Usage of Assembly Code," in *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, A. Biere and D. Parker, Eds., ser. Lecture Notes in Computer Science, vol. 12079, Springer, 2020, pp. 98–117. DOI: 10.1007/978-3-030-45237-7_6.

[15] F. Besson, S. Blazy, and P. Wilke, "CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics," *Journal of Automated Reasoning*, vol. 63, no. 2, pp. 369–392, Aug. 2019, ISSN: 1573-0670. DOI: 10.1007/s10817-018-9496-y.

[16] S. Boldo, J. Jourdan, X. Leroy, and G. Melquiond, "A Formally-Verified C Compiler Supporting Floating-Point Arithmetic," in *2013 IEEE 21st Symposium on Computer Arithmetic*, Apr. 2013, pp. 107–115. DOI: 10.1109/ARITH.2013.30.

[17] Y. Kiam Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "The Verified CakeML Compiler Backend," *Journal of Functional Programming*, vol. 29, e2, 2019. DOI: 10.1017/S0956796818000229.

[18] A. Chlipala, "A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language," *SIGPLAN Not.*, vol. 42, no. 6, pp. 54–65, Jun. 2007, ISSN: 0362-1340. DOI: 10.1145/1273442.1250742.

[19] X. Leroy, "Formal Verification of a Realistic Compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009, ISSN: 0001-0782. DOI: 10.1145/1538788.1538814.

[20] M. O. Myreen, "Verified Just-in-Time Compiler on X86," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10, Madrid, Spain: Association for Computing Machinery, 2010, pp. 107–118, ISBN: 9781605584799. DOI: 10.1145/1706299.1706313.

[21] J. Ševčık, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency," *J. ACM*, vol. 60, no. 3, Jun. 2013, ISSN: 0004-5411. DOI: 10.1145/2487241.2487248.

[22] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010, ISSN: 0001-0782. DOI: 10.1145/1785414.1785443.

[23] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation Validation for a Verified OS Kernel," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 471–482, ISBN: 9781450320146. DOI: 10.1145/2491956.2462183.

[24] F. Verbeek, J. Bockenek, A. Bharadwaj, B. Ravindran, and I. Roessle, "Establishing a Refinement Relation between Binaries and Abstract Code," in *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design*, ser. MEMOCODE '19, La Jolla, California: Association for Computing Machinery, 2019, ISBN: 9781450369978. DOI: 10.1145/3359986.3361215.

[25] K. Chatterjee, A. Pavlogiannis, and V. Toman, "Value-Centric Dynamic Partial Order Reduction," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: 10.1145/3360550.

[26] E. Albert, M. G. de la Banda, M. Gómez-Zamalloa, M. Isabel, and P. J. Stuckey, "Optimal Context-Sensitive Dynamic Partial Order Reduction with Observers," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 352–362, ISBN: 9781450362245. DOI: 10.1145/3293882.3330565.

[27] M. Chalupa, K. Chatterjee, A. Pavlogiannis, N. Sinha, and K. Vaidya, "Data-Centric Dynamic Partial Order Reduction," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. DOI: 10.1145/3158119.

[28] S. Huang and J. Huang, "Maximal Causality Reduction for TSO and PSO," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016, Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 447–461, ISBN: 9781450344449. DOI: 10.1145/2983990.2984025.

[29] H. Shiyou and H. Jeff, "Speeding Up Maximal Causality Reduction with Static Dependency Analysis," in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, P. Müller, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 74, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 16:1–16:22, ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.16.

[30] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified Synthesis: Automatically Learning the X86-64 Instruction Set," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 237–250, ISBN: 9781450342612. DOI: 10.1145/2908080.2908121.

[31] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas, "Stateless Model Checking for TSO and PSO," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 353–367, ISBN: 978-3-662-46681-0. DOI: 10.1007/s00236-016-0275-0.

[32] S. Goel, W. A. Hunt, M. Kaufmann, and S. Ghosh, "Simulation and Formal Verification of X86 Machine-Code Programs That Make System Calls," in *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '14, Lausanne, Switzerland: FMCAD Inc, 2014, pp. 91–98, ISBN: 9780983567844. DOI: 10.1109/FMCAD.2014.6987600.

[33] S. Goel, W. A. Hunt, and M. Kaufmann, "Engineering a Formal, Executable x86 ISA Simulator for Software Verification," in *Provably Correct Systems*, M. Hinchey, J. P. Bowen, and E.-R. Olderog, Eds. Cham: Springer International Publishing, 2017, pp. 173–209, ISBN: 978-3-319-48628-4. DOI: 10.1007/978-3-319-48628-4_8.

[34] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A Complete Formal Semantics of X86-64 User-Level Instruction Set Architecture," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1133–1148, ISBN: 9781450367127. DOI: 10.1145/3314221.3314601.

[35] I. Corparation, "Intel (R) 64 and IA-32 Architectures Software Developer's Manual," *Combined Volumes, October*, 2019, Order Number: 325462-071US.

[36] E. Mercer and M. Jones, "Model Checking Machine Code with the GNU Debugger," in *Model Checking Software*, P. Godefroid, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 251–265, ISBN: 978-3-540-31899-6. DOI: 10.1007/11537328_20.

[37] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[38] M. Musuvathi and S. Qadeer, "Iterative Context Bounding for Systematic Testing of Multithreaded Programs," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, San Diego, California, USA: Association for Computing Machinery, 2007, pp. 446–455, ISBN: 9781595936332. DOI: 10.1145/1250734.1250785.

[39] J. Russell Stuart and P. Norvig, *Artificial intelligence: a modern approach*. Prentice Hall, 2009.

[40] P. Godefroid and D. Pirottin, "Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract)," in *Computer Aided Verification*, C. Courcoubetis, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 438–449, ISBN: 978-3-540-47787-7. DOI: 10.1007/3-540-56922-7_36.

[41] C. Flanagan and P. Godefroid, "Dynamic Partial-Order Reduction for Model Checking Software," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05, Long Beach, California, USA: Association for Computing Machinery, 2005, pp. 110–121, ISBN: 158113830X. DOI: 10.1145/1040305.1040315.

[42] A. Valmari, "Stubborn Sets for Reduced State Space Generation," in *Advances in Petri Nets 1990*, G. Rozenberg, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 491–515, ISBN: 978-3-540-46369-6. DOI: 10.1007/3-540-53863-1_36.

[43] W. Brauer, W. Reisig, and G. Rozenberg, "Petri Nets: Applications and Relationships to Other Models of Concurrency Advances in Petri Nets 1986, Part II Proceedings of an Advanced Course Bad Honnef, 8.–19. September 1986," in *Conference proceedings ACPN*, Springer, 1986, p. 344. DOI: 10.1007/3-540-17906-2.

[44] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün, "Static Partial Order Reduction," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Steffen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 345–357, ISBN: 978-3-540-69753-4. DOI: 10.1007/BFb0054182.

[45] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal Dynamic Partial Order Reduction," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14, San Diego, California, USA: Association for Computing Machinery, 2014, pp. 373–384, ISBN: 9781450325448. DOI: 10.1145/2535838.2535845.

[46] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction," *J. ACM*, vol. 64, no. 4, Aug. 2017, ISSN: 0004-5411. DOI: 10.1145/3073408.

[47] B. Demsky and P. Lam, "SATCheck: SAT-Directed Stateless Model Checking for SC and TSO," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015, Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 20–36, ISBN: 9781450336895. DOI: 10.1145/2814270.2814297.

[48] Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sep. 1979, ISSN: 1557-9956. DOI: 10.1109/TC.1979.1675439.

[49] N. Zhang, M. Kusano, and C. Wang, "Dynamic Partial Order Reduction for Relaxed Memory Models," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15, Portland, OR, USA: Association for Computing Machinery, 2015, pp. 250–259, ISBN: 9781450334686. DOI: 10.1145/2737924.2737956.

[50] I. Roessle, F. Verbeek, and B. Ravindran, "Formally Verified Big Step Semantics out of X86-64 Binaries," in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2019, Cascais, Portugal: Association for Computing Machinery, 2019, pp. 181–195, ISBN: 9781450362221. DOI: 10.1145/3293880.3294102.

[51] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "RockSalt: Better, Faster, Stronger SFI for the X86," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: Association for Computing Machinery, 2012, pp. 395–404, ISBN: 9781450312059. DOI: 10.1145/2254064.2254111.

[52] M. O. Myreen, "Formal Verification of Machine-Code Programs," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-765, Dec. 2009.

[53] X. Feng and Z. Shao, "Modular Verification of Concurrent Assembly Code with Dynamic Thread Creation and Termination," in *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '05, Tallinn, Estonia: Association for Computing Machinery, 2005, pp. 254–267, ISBN: 1595930647. DOI: 10.1145/1086365.1086399.

[54] J. S. Moore, "A Mechanically Checked Proof of a Multiprocessor Result via a Uniprocessor View," *Formal Methods in System Design*, vol. 14, no. 2, pp. 213–228, Mar. 1999, ISSN: 1572-8102. DOI: 10.1023/A:1008624904634.

[55] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05, [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard.pdf, Anaheim, CA: USENIX Association, 2005, p. 41.

[56] K. P. Lawton, "Bochs: A Portable PC Emulator for Unix/X," *Linux J.*, vol. 1996, no. 29es, 7–es, Sep. 1996, ISSN: 1075-3583. DOI: 10.5555/326350.326357.

[57] A. Q. Nguyen and H. V. Dang, "Unicorn: Next generation CPU emulator framework," in *Proceedings of the 2015 Blackhat USA conference*, 2015.

[58] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 190–200, ISBN: 1595930566. DOI: 10.1145/1065010.1065034.

[59] J. S. Moore, "Mechanized Operational Semantics," *Lectures in the Marktoberdorf Summer School (August 5-16, 2008)., Online*, 2008.

[60] U. Drepper, "ELF Handling for Thread-Local Storage, Version 0.20," *Red Hat*, 2005.

[61] J. L. Peterson and T. A. Norman, "Buddy Systems," *Commun. ACM*, vol. 20, no. 6, pp. 421–431, Jun. 1977, ISSN: 0001-0782. DOI: 10.1145/359605.359626.

[62] M. Herlihy and N. Shavit, "The Art of Multiprocessor Programming, Revised Reprint," *ISBN-13*, pp. 978–0 123 973 375, 2012.

[63] M. Kokologiannakis and K. Sagonas, "Stateless Model Checking of the Linux Kernel's Read–Copy Update (RCU)," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 3, pp. 287–306, Jun. 2019, ISSN: 1433-2787. DOI: 10.1007/s10009-019-00514-6.