On the Application of Formal Techniques for Dependable Concurrent Systems

Vom Fachbereich Informatik der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
eines Doktor rerum naturalium (Dr. rer. nat.)
vorgelegt von

Habib Saissi, Msc

aus Safi, Marokko

Referenten:
Prof. Neeraj Suri, Ph.D.
Prof. Dr. Johannes Kinder

Datum der Einreichung: 15. Februar 2019
Datum der mündlichen Prüfung: 29. März 2019

Darmstadt 2019
D17

Habib Saissi: *On the Application of Formal Techniques for Dependable Concurrent Systems*
Darmstadt, Technische Universität Darmstadt
Tag der mündlichen Prüfung: 29.03.2019
Jahr der Veröffentlichung der Dissertation auf TUprints: 2019
URN: urn:nbn:de:tuda-tuprints-86009

On the Application of Formal Techniques for Dependable Concurrent Systems

By Habib Saissi

In memory of my grand-fathers, Thami Saissi
and Haddou Bakzaza

ERKLÄRUNG

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwendnung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 15. Februar 2019

Habib Saissi

ABSTRACT

The pervasiveness of computer systems in virtually every aspect of daily life entails a growing dependence on them. These systems have become integral parts of our societies as we continue to use and rely on them on a daily basis. This trend of digitalization is set to carry on, bringing forth the question of how *dependable* these systems are. Our dependence on these systems is in acute need for a justification based on rigorous and systematic methods as recommended by internationally recognized safety standards. Ensuring that the systems we depend on meet these recommendations is further complicated by the increasingly widespread use of concurrent systems, which are notoriously hard to analyze due to the substantial increase in complexity that the interactions between different processing entities engenders.

In this thesis, we introduce improvements on existing formal analysis techniques to aid in the development of dependable concurrent systems. Applying formal analysis techniques can help us avoid incidents with catastrophic consequences by uncovering their triggering causes well in advance. This work focuses on three types of analyses: data-flow analysis, model checking and error propagation analysis. Data-flow analysis is a general static analysis technique aimed at predicting the values that variables can take at various points in a program. Model checking is a well-established formal analysis technique that verifies whether a program satisfies its specification. Error propagation analysis (EPA) is a dynamic analysis whose purpose is to assess a program's ability to withstand unexpected behaviors of external components. We leverage data-flow analysis to assist in the design of highly available distributed applications. Given an application, our analysis infers rules to distribute its workload across multiple machines, improving the availability of the overall system. Furthermore, we propose improvements to both explicit and bounded model checking techniques by exploiting the structure of the specification under consideration. The core idea behind these improvements lies in the ability to abstract away aspects of the program that are not relevant to the specification, effectively shortening the verification time. Finally, we present a novel approach to EPA based on symbolic modeling of execution traces. The symbolic scheme uses a dynamic sanitizing algorithm to eliminate effects of non-determinism in the execution traces of multi-threaded programs. The proposed approach is the first to achieve a 0 % rate of false positives for multi-threaded programs.

The work in this thesis constitutes an improvement over existing formal analysis techniques that can aid in the development of dependable concurrent systems, particularly with respect to availability and safety.

KURZFASSUNG

Der Einzug von Computersystemen in nahezu allen Bereichen des täglichen Lebens führt zu einer zunehmenden Abhängigkeit von ihnen. Diese Systeme sind zu einem festen Bestandteil unserer Gesellschaft geworden, da wir sie täglich nutzen und uns auf sie verlassen. Dieser Digitalisierungstrend wird sich fortsetzen, so dass sich die Frage stellt, wie zuverlässig diese Systeme sind. Unsere Abhängigkeit von diesen Systemen erfordert dringend Rechtfertigung aufgrund strenger und systematischer Methoden, die von international anerkannten Sicherheitsstandards empfohlen werden. Die Sicherstellung, dass die Systeme, auf die wir angewiesen sind, die Empfehlungen der Sicherheitsstandards erfüllen, wird durch den zunehmend verbreiteten Einsatz von nebenläufigen Systemen zusätzlich erschwert. Nebenläufige Systeme sind bekanntermaßen schwer zu analysieren, da die Interaktionen zwischen verschiedenen Verarbeitungseinheiten erheblich komplexer werden. In dieser Arbeit stellen wir Verbesserungen an bestehenden formalen Analysetechniken vor, um die Entwicklung zuverlässiger nebenläufiger Systeme zu unterstützen. Die Anwendung formaler Analysetechniken kann dazu beitragen, Vorfälle mit katastrophalen Folgen zu vermeiden, indem ihre auslösenden Ursachen frühzeitig aufgedeckt werden.

Diese Arbeit konzentriert sich auf drei Arten von Analysen: Datenflussanalyse, Model Checking und Error-Propagation-Analyse. Die Datenflussanalyse ist eine allgemeine statische Analysetechnik, die darauf abzielt, die Werte vorherzusagen, die Variablen an verschiedenen Stellen eines Programms annehmen können. Model Checking ist eine etablierte formale Analysetechnik, mit der überprüft wird, ob ein Programm seinen Spezifikationen erfüllt. Die Error-Propagation-Analyse (EPA) ist eine dynamische Analyse, deren Zweck es ist, zu bewerten ob ein Programm unerwartetem Verhalten externer Komponenten standhalten kann. Wir nutzen Datenflussanalysen, um das Design hochverfügbarer verteilter Anwendungen zu unterstützen. Bei einer Anwendung werden in unserer Analyse Regeln festgelegt, um die Arbeitslast auf mehrere Maschinen zu verteilen und die Verfügbarkeit des Gesamtsystems zu verbessern. Darüber hinaus schlagen wir Verbesserungen sowohl für explizites Model Checking als auch für Bounded Model Checking vor, indem die Struktur der betrachteten Spezifikation genutzt wird. Die Kernidee hinter unseren Verbesserungen liegt in der Fähigkeit, Aspekte des Programms, die für die Spezifikation nicht relevant sind, zu abstrahieren. Schließlich stellen wir einen neuen Ansatz für die EPA vor, der auf der symbolischen Modellierung von Ausführungen basiert. Der symbolische Ansatz

verwendet einen dynamischen Sanitizing-Algorithmus, um die Auswirkungen von Nicht-Determinismus in Ausführungen von Multi-Threaded-Programmen zu beseitigen. Der vorgeschlagene Ansatz ist der erste, der bei Multi-Threaded-Programmen eine False-Positives Rate von 0 % erzielt.

Die vorliegende Arbeit stellt eine Verbesserung gegenüber bestehenden formalen Analysetechniken dar, die zur Entwicklung zuverlässiger nebenläufiger Systeme beitragen können, insbesondere hinsichtlich der Verfügbarkeit und Sicherheit.

ACKNOWLEDGMENTS

As passionate as I was about video games back in the 1990's and early 2000's, I aspired to become a video game developer for a long period of my childhood. I had witnessed the progress being made in graphics rendering first hand, transitioning from 2D to 3D models. Joining the computer science program at the university, however, made me discover a new passion: the theoretical aspect of computation. There have been ups and downs since then but I'm glad I've made the choices that I've made.

My father has always been my role model. He taught me to be curious about things and see the value in not restricting myself to one subject of knowledge. My mother has been always there for me, being my refuge when things were not easy. My sister's warmth and cheerfulness provided me with the necessary energy to push forward no matter what. The new additions to the family, Illy and Yanis, have brought me joy with their smiles and liveliness. I'm very grateful for my whole family's unconditional love and support during my journey so far. None of this would have been possible if it weren't for all of you.

I would like to thank my advisor, Neeraj Suri, for the trust he put in me, giving me the freedom to pursue my own research interests while watching over me not to stray away from my objectives. His constructive critique and guidance helped shaping my ideas and allowed me to grow as a researcher. I'm deeply appreciative for his support to make this happen.

I would also like to thank Johannes Kinder for accepting to be my external reviewer, and Thomas Schneider, Kay Hamacher and Stefan Katzenbeisser for being on my committee.

Hatem, I consider myself very lucky that I met you at DEEDS and that we could forge such a close friendship. We've been through quite a lot together and I hope that our friendship won't stop at this. Thank you my friend for the soothing park walks, the discussions about all and nothing, and for being such a good listener. Our trips abroad were a lot of fun and we definitely should plan for more.

Olli, my office buddy, thanks for the late philosophical discussions and reintroducing the theoretician that I am to the more practical aspects of computer science. I've rarely met someone with such readiness to help a friend. Thank you for that!

Monsieur Nicolas, thank you for introducing me to the world of PC gaming again and engineering the best LaTeX table ever. I really enjoyed our discussions about politics, history and football although we still have to reach a consensus on the "mustard conundrum".

Thanks Tsveti (Schatzmaus) for the fun conversations and consistently sabotaging my diet plans with tasteless french fries. Unlike you, I hope that you keep going to the Power-Fit training and yes, I'm still planning to visit Bulgaria!

Thank you Sabine for your help with various paper work and bringing Haley into our office life. I really enjoyed our morning coffees and baking discussions!

Thank you Ute for your admin and hardware support. I really hope we get to try out your famous mousse au chocolat soon again.

Thanks Patrick for being part of the formal methods subgroup and for your critical but valuable insights regarding my ideas. Our trips to Japan and Hungary were a lot of fun!

Salman, I was a bit disappointed when you shaved your mustache but I'm at least glad that you joined the black coffee drinker fraction. Our Biryani lunch was such a feast. It goes without a saying that I hope you invite me soon again!

Thank you Stefan for introducing me to the world of craft beer and for the nice discussions on the Japanese culture.

Thanks Heng for your lectures on the stock exchange market and the differences between Japanese and Chinese logographs.

Thanks, Yiqun for making me try the most smell-intensive liquor I've had in my life. That was indeed a once in a lifetime experience!

Thank you Marco for being such a good help throughout my PhD endeavor. Your work ethic and humility immensely inspired me and motivated me to improve myself as a person and researcher.

Peter, I'm very grateful that our paths crossed and glad that we became such good friends. Thank you for introducing me to the world of formal methods and helping me out throughout my PhD. Discussing papers and new ideas with you was such a delight. Thanks to you I've learned to see the beauty and elegance of rigorous proofs and formalisms. Our trips to China, Japan and Morocco as well as my frequent visits to Berlin are important highlights of my PhD journey. Vive le Tour de France!

Thanks Lion for the enlightening discussions on linguistics and the peculiarities of the German language. Thank you for consistently correcting my mistakes and never failing to provide the right explanation for the not so intuitive rules of the language. Our trips to Morocco and Hungary were great!

Many thanks to the rest of my DEEDS contemporaries: Ahmed, Daniel, Giancarlo, Hamza, Jesus, Kubi, Ruben, Tasuku, Thorsten, and Zhazira. Thank you all for making DEEDS a great place for exchanging ideas and fruitful collaboration.

Last but not least, I would like to thank Marta for being part of my life. Thank you for your patience and relentless support throughout my PhD years.

Habib Saissi
Darmstadt, March 13, 2019

CONTENTS

Part I

INTRODUCTION

1

INTRODUCTION

The pervasive use of digital technologies in virtually all aspects of daily life entails our growing dependence on their reliable delivery of services. For instance, our road infrastructures are largely governed by computerized systems that deal with congestion using smart traffic lights. Our cars consist of more electronic units than ever before. It has become unthinkable, and illegal in many countries, to drive a car without safety mechanisms, such as the anti-lock braking system (ABS), which are only enabled by dedicated software on board. This trend is estimated to carry on as reported by the visual networking index [Cis18]. According to the report, the number of connected devices is expected to reach 28.5 billion (3.6 devices per capita) by 2022 as opposed to approximately 20 billions in 2018. Machine-to-machine (M2M) units, which currently account for 6 % of the number of connected devices and are particularly relevant for safety, are expected to grow at an even higher rate, making up 51 % of the total number of devices by 2022. This continuous and ever expanding automation of processes in various domains, especially the ones where safety is a major concern, raises the question of how *dependable* these systems are. While the benefit of using these systems is undeniable, it is important to be able to justify our reliance on them based on methodical means. The more complex these systems grow, the higher the need for rigorous and systematic methods. This thesis advocates the use of formal techniques to provide this much needed justification.

Many existing safety standards recommend the usage of formal methods in different stages of software development of safety critical systems. A safety critical system is a system whose failure may lead to severe consequences such as injuries, fatalities, damage to the environment, unauthorized disclosure of information or financial loss [Som+15]. Figure 1 shows a simplified overview of the current international safety standards for such systems. The IEC61508 standard [IEC10] is of particular interest as it forms a basis for many other domain specific standards such as ISO 26262 [ISO11] for the automotive industry or EN 50126 [EN517] for railway systems. The IEC61508 standard defines four safety integrity levels (SIL) that evaluate the risk involved in each functionality of safety critical systems, with SIL 1 being the lowest level and SIL 4 the highest. Intuitively, the higher the SIL for a specific functionality, the more rigorously it has to be

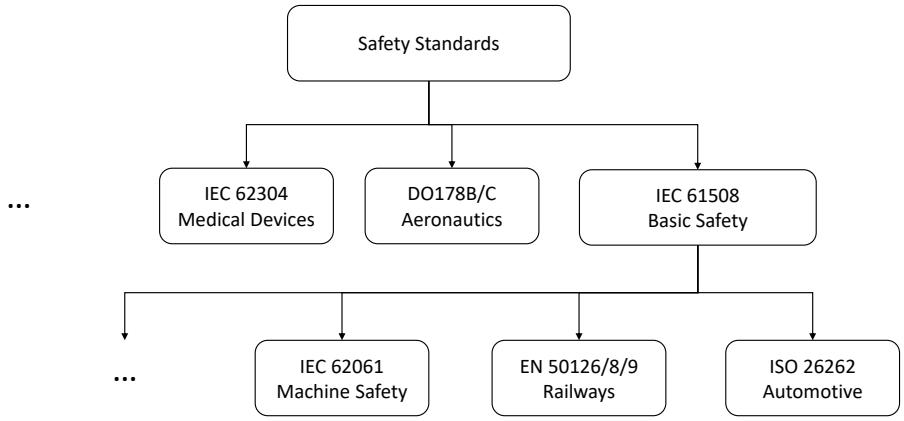


Figure 1: Overview of the different safety standards recommending formal methods for safety critical systems.

developed and assessed. All of the standards based on the IEC61508 standard "highly recommend" the usage of formal methods for SIL 4 functionalities, with EN 50126 recommending them even for SIL 1 and 2 and highly recommending them for SIL 3 and 4 functionalities.

Ensuring that safety critical systems meets the recommendations of the safety standards is further complicated by the increasingly widespread use of parallelization. Over the last three decades, we have witnessed a paradigm shift towards more parallelization of computer systems due the physical limitations on CPU power [PH13]. In Figure 2, we show an overview of the main approaches to achieve that. The figure shows two different granularity levels for parallelization. First, a system can be parallelized on the level of a single machine/device by exploiting multiple CPU architectures and operating systems (OS) scheduling to allow for better hardware utilization, thus boosting the performance. In this case, different programs run multiple threads or processes to handle different functionalities simultaneously. For instance in Figure 2, device 1 is running multiple programs which are running multiple threads in parallel (represented by the edges connecting the program to the operating system). One of these threads could be handling user input, another thread could draw the user interface or handle communication with a database (represented by the other program on the same machine). Second, parallelization can be achieved by having different system services run on different machines and communicate through message passing. These machines can either be locally distributed such as electronic control units (ECU) within a vehicle or a distributed database in a financial data center, or geographically distributed in a large scale system. In Figure 2, all the devices are running in parallel and are communicating through message passing. Although they belong to the same system, these devices can be spread across the globe and not necessarily under the hood of the same vehicle. We refer to the first type of parallelized systems as a multi-threaded program and the second as a distributed

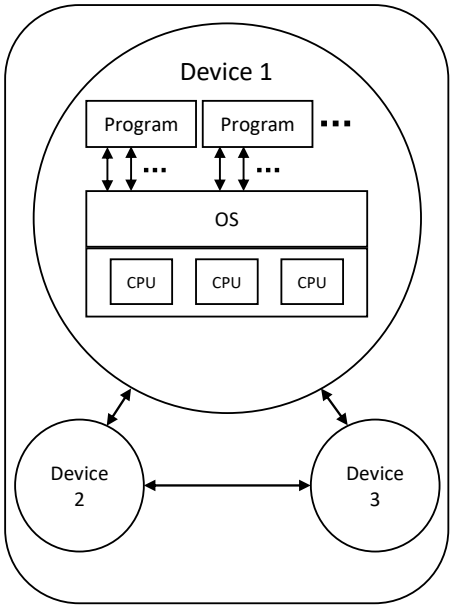


Figure 2: The different levels of parallelization. Programs run simultaneously on a single machine thanks to multi-core architectures and OS scheduling. On a higher level, devices are running in parallel and communicate through message passing.

system. Note that both paradigms can coexist in a single system as visualized in Figure 2.

While apparently different, these two scenarios share two commonalities. In both cases different processing entities are running concurrently and cooperate to deliver the expected services. In order to cooperate, these processing entities influence each other through direct modifications to a common global shared state or through message-passing so that the timing of these modifications and how they relate to each other on the execution timeline becomes important. Concurrency, however, allows these entities to operate simultaneously making it hard to predict the order of these modifications. Analyzing systems with such non-determinism is a daunting task due to the additional complexity arising from interactions between the processing entities. The inherent non-determinism in concurrent systems can be detrimental to the proper functioning of safety critical systems and might result in hazardous consequences [LT93]. The need for a rigorous justification is, therefore, even higher for such concurrent systems.

1.1 FORMAL PROGRAM ANALYSIS

Formal analysis techniques can be used to strengthen our trust in safety critical systems. In this thesis, we explore how these techniques can be used and improved to ensure the dependability of safety critical concurrent systems. In the following, we briefly discuss a subset of

existing formal analysis techniques covered in the thesis. We start by covering different aspects common to all of these techniques.

Static Vs. Dynamic Analysis

Formal analysis techniques can be used to predict a program's behavior and assert its operational properties. The ability to predict the program's behavior before deployment plays a major role in justifying our trust in systems we depend on. These techniques can be either *static* or *dynamic*. Static analysis is an umbrella term for a panoply of analysis techniques that examine a static representation of programs, e.g., source or machine code. Such techniques include, but are not limited to, data flow analysis, control flow analysis or abstract interpretation [NNH15] among others. Given a program, the static analysis techniques attempt to predict its behavior when executed.

The dynamic counterparts of static analysis techniques examine program executions rather than the full program. Given a set of program executions, dynamic analyses attempt to infer the general behavior of the program. Since dynamic approaches analyze concrete executions, they can be more precise compared to static analysis. However, as only a subset of the executions are analyzed, the analysis is limited to what can be inferred from them. For instance, if a specific control flow branch is taken by none of the examined executions, dynamic analysis techniques can not take that branch into account.

Approximation

An ideal analysis technique should be precise and cover all possible behaviors of the program. Such an ideal analysis can, however, not exist in general as it has been shown in [Lan92]. Furthermore, analysis techniques often face efficiency limitations that hamper their applicability to real systems. To overcome these limitations, a widely used approach is approximation. Over-approximating techniques include more program behaviors than are allowed [CC77] by abstracting its actual behavior, attempting to prove the non-existence of undesirable executions. On the other hand, under-approximating techniques limit the possible behavior of the program to less than what is possible [BCC+99], and attempt to prove the existence of undesirable behavior.

The soundness of an analysis technique reflects its ability to provide correct positive answers while completeness refers to its ability to provide correct negative answers. Approximating techniques can be either sound or complete depending on how they abstract or limit the behavior of a program. For instance, over-approximations techniques are sound if they only prove the absence of undesirable behavior for correct programs (no false positives), and are complete if only

incorrect programs are proven to not contain undesirable behavior (no false negatives).

Data Flow Analysis

Data flow analysis is a static analysis technique whose purpose is to infer the set of possible values program variables can take at various program locations [NNH15]. For this purpose, the control flow graph is harnessed to examine the possible execution paths of the program and reason about the values of the variables. The analysis can be simplified by ignoring the control flow graph, effectively combining all control flow paths together. While the resulting information is an over-approximation, and therefore is inaccurate, this simplification can considerably speed up the analysis and can be sufficient for many application cases.

Model Checking

Formal verification techniques such as model checking [CJGK+18] have made major strides toward more efficiency and practicality in the course of the last 30 years. Roughly speaking, a model checking tool (model checker) takes as input the program and a specification to verify whether it is satisfied. The program under examination can either be a simplified model described in a modeling language such as the *Promela* language [Spi] or the actual implementation with all the details (e.g., [CKL04]). There are benefits and drawbacks to both approaches. Model checking a simplified model of the program is significantly less complicated, and therefore less costly, than model checking the implementation. This, however, comes at the cost of the reported results being only directly applicable to the model and not to the implementation. Nevertheless, both approaches have merit and can be used in tandem. Model-based model checking can be used to fix reasoning flaws in the underlying algorithms in the early stages of development while the implementation can be model checked in later stages to ensure that no bugs have been introduced in the implementation phase.

The specification of a program can be expressed differently depending on the required expressiveness. For instance, a safety property could be expressed solely using a predicates about the allowed states. A liveness property [Lam77], which requires that a system makes progress in the form of regular occurrence of certain events, requires a more sophisticated logic such as linear-time-logic (LTL) [CJGK+18]. The main goal of a model checker is to systematically explore every possible behavior of the program and check whether it is allowed. Alternatively, a model checker can also be employed to check the existence of desirable behaviors among all possible executions. Concretely,

to verify whether a program satisfies the specification, a model checker systematically explores every possible state (e.g., safety property) and every possible execution sequence (e.g., liveness property).

Model checking techniques can be static or dynamic. The explored program can be given concrete inputs, and in that case we speak of explicit model checking, or symbolic inputs (e.g., symbolic execution, BMC). Explicit model checking requires the exploration of every possible interleaving of instructions in the program given concrete input. It is, therefore, only meaningful for concurrent programs since otherwise it is equivalent to testing. Explicit model checking is considered to be a dynamic approach since it examines concrete executions of the program, potentially missing program behavior that is never triggered by the given input. Symbolic approaches, on the other hand, are static with some of them being a combination of dynamic and static analysis (e.g., concolic execution [Sen07]). A well-established symbolic approach is bounded model checking (BMC) [BCC+99]. A bounded model checker encodes the behavior of a program in a formula such that its satisfying assignments can be directly mapped to concrete executions. The generated formula is then extended by the negation of the property, i.e., the specification, that the program has to satisfy. Every assignment to such a formula represents an execution of the program that violates the specified property. The length of program executions that is covered by the formula is bound by a finite number so that only a subset of all possible executions is encoded. In other words, BMC considers an under-approximation of the program's full possible behavior.

Error Propagation Analysis (EPA)

A prominent dynamic program analysis technique is EPA. EPA analyzes how software bugs affect program control and data flow at run time, which is useful for error detector placement [HJS02; CSW+17] and robustness testing [NWC+18]. For this purpose, programs are mutated similarly to mutation testing to simulate realistic software bugs as well as hardware defects [NWC+18]. Such modifications include, for instance, the introduction of `NULL` pointer accesses, randomly generated bit flips in the value of a variable or deliberate race conditions. To determine the effects of the introduced bugs on program execution, EPA compares bug-affected (faulty run) against bug-free (golden run) execution. Any deviation between the faulty run and golden run is then reported to analyze the program's ability to deal with unexpected behavior.

Given this context, this thesis

1. proposes a novel approach to designing distributed systems that leverages a specially tailored static data-flow analysis,

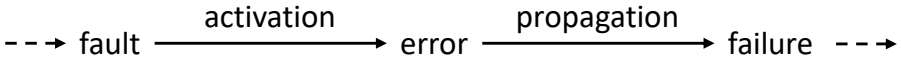


Figure 3: The fundamental chain of threats to dependability [ALR+04].

2. develops and presents two approaches to improve model checking of concurrent systems, and

3. presents an approach based on symbolic modeling of program executions to assist in error propagation analysis of multi-threaded programs.

The rest of this chapter is organized as follows. First, we provide a precise definition for dependability and present the different threats to it in Section 1.2. We then discuss the role of formal analysis techniques in enabling the dependability of concurrent systems. In Sections 1.3 and 1.4, we summarize the contributions presented in this thesis and the resulting publications. Finally, Section 1.5 provides an outline for the content of the thesis.

1.2 THE ROLE OF PROGRAM ANALYSIS IN BUILDING DEPENDABLE SYSTEMS

Our reliance on safety critical systems is based on the premise that they are dependable. We start by precisely defining what is meant by dependability. We then describe the different factors that can threaten it and the usual means to achieve it.

Dependability is the ability of a system to function correctly that can be justifiably trusted [ALR+04]. In other words, dependability expects the trust we put in systems to deliver their intended service to be justifiable. Concretely, dependability is an attribute of a system that aggregates its reliability, availability, safety, integrity and maintainability where:

- *reliability* describes the continuity of correct service of the system,
- *availability* ensures that the system is ready for correct service when needed,
- *safety* follows from the absence of catastrophic consequences on the user(s) and the environment,
- *integrity* is assured with the absence of improper alterations, and
- *maintainability* reflects the system's ability to undergo modifications and repairs.

In Figure 3, we depict the classical chain of dependability threats [ALR+04]. The figure shows how a typical chain of events can lead to the failure of a system, compromising its dependability. A *fault* is a flaw in the

program design or implementation that when *activated* leads to an *error*. That is, the execution of a fault is the cause for the occurrence of errors. If a fault is not activated, it is said to be dormant. A fault can remain dormant until enabling conditions are satisfied leading to its activation (e.g., specific program inputs or interleaving of events in a concurrent program). A *failure* is the deviation of a system from its correct behavior that is observable by other components in the system and which violates any of the dependability attributes. The failure of a system is caused by the *propagation* of an error to the boundaries which in turn is the activation of a fault in the system. For instance, a system is said to have failed in case an existing flawed logic in the program (fault) has been executed (error) affecting the correctness of the systems output (failure).

Building dependable systems involves developing systems that cope with the threats of faults, errors and failures. To this aim, a plethora of means have been developed to prevent (1) errors by reducing the number of faults and therefore the potential for their activation and (2) failure by containing the propagation of errors.

These means can be categorized into four classes:

- *Fault Prevention:* preventing the occurrence of faults. This category encompasses software design and best practices in development that are targeted towards producing program code with a minimal number of faults.

- *Fault Removal:* reducing the number of faults before deployment. Fault removal measures consists of techniques whose aim is to examine the produced program code for the existence of faults so that they can be removed. Typical examples for these techniques are software testing, EPA or model checking. For instance, EPA's aim is to identify deviations between faulty and golden executions with the same input. A reported deviations signals that the injected fault has been activated and that the resulting error indeed propagated if the output of the program also deviates. In this case, the identified fault is the missing logic that handles such unexpected external behavior (i.e., the injected fault).

- *Fault Tolerance:* avoiding system failure even in the presence of faults/errors.

- *Fault Forecasting:* estimating the present number of, the future incidence, and the likely consequences of faults.

Thus, formal program analysis techniques can be applied for fault prevention and removal. The contributions presented in this thesis are focused on improving and leveraging formal analysis techniques to improve the dependability of concurrent systems in terms of safety and availability specifically.

1.3 CONTRIBUTIONS

In this section we summarize, the four contributions covered by this thesis. We formulate three research questions and present the corresponding contributions. The first contribution deals with fault prevention using a novel approach to design highly available distributed systems based on static analysis. The second and third contributions improve existing model checking techniques. Finally, in the fourth contribution we present a novel approach for EPA of concurrent systems.

Research Question (RQ1): *Can static analysis techniques assist in designing highly available distributed systems?*

Availability, an attribute of dependability, is one of the major challenges of distributed systems. For these systems, the ability to scale out in order to serve a large number of clients is a highly desirable property. Typically this is achieved by replicating services across multiple servers and dynamically dispatching client requests. Full replication of the services is expensive due to the amount of synchronization necessary to keep the servers' state consistent. Moreover, the cost of synchronization grows dramatically for large-scale systems because of the distance between the servers. A major line of work exist to achieve higher availability by relaxing the level of consistency needed by the servers [SDM+10; DHJ+07]. Trading the consistency of the servers' state for high availability is, however, unacceptable for dependable systems. To this aim we present our first contribution, a static analysis based approach that high availability without sacrificing state consistency.

Contribution (C1): *Operation partitioning: A Technique to Scale out Single-Server Systems [SSS19]*

OLTP[1] applications with high workloads that cannot be served by a single server need to scale out to multiple servers. Typically, scaling out entails assigning a different partition of the application state to each server. But data partitioning is at odds with preserving the strong consistency guarantees of ACID[2] transactions, a fundamental building block of many OLTP applications. The more we scale out and spread data across multiple servers, the more frequent distributed transactions accessing data at different servers will be. With a large number of servers, the high cost of distributed transactions makes scaling out ineffective or even detrimental. Our first contribution introduces *Operation Partitioning*, a novel paradigm to scale out OLTP applications that

1 OLTP: Online transaction processing.
2 ACID: Atomicity, consistency, isolation and durability.

require ACID guarantees. Operation Partitioning indirectly partitions data across servers by partitioning the application's *operations* through static analysis. This partitioning of operations yields to a lock-free *Conveyor Belt* protocol for distributed coordination, which can scale out *unmodified* applications running on top of *unmodified* database management systems. We implement the protocol in a system called Gyro and use it to scale out two applications, TPC-W and RUBiS. Our experiments show that Gyro can increase maximum throughput and reduce latency compared to MySQL Cluster while at the same time providing a stronger isolation guarantee (serializability instead of read committed).

Research Question (RQ2): *Can the structure of a specification property be leveraged to improve the efficiency of model checking?*

Model checking approaches have been widely used to minimize the number of software bugs by systematically exploring the state space of a program. This comes, however, at the cost of scalability and applicability. The cost of systematic exploration is worsened in the case of concurrent programs as the state space of a program grows exponentially with the number of processes/threads. This problem is traditionally referred to as the state explosion problem [Val98]. Many approaches have been proposed to circumvent the state explosion problem [AAB+17; AAJ+18; BKS+11; AKT13]. Most prominent approaches are based on the partial-order reduction (POR) theory [Maz87]. The next two contributions present two orthogonal approaches that can be used to improve the efficiency of model checking.

Contribution (C2): *Decomposition-based Explicit Model Checking for Message-Passing Protocols [SBM+13]*

Our second contribution is an efficient model checking approach for distributed message-passing protocols. Key to the achieved efficiency is a novel stateful model checking strategy that is based on the decomposition of states into a relevant and an auxiliary part according to the specification property. We formally show this strategy to be sound, complete, and terminating for general finite-state systems. As a case study, we implement the proposed strategy within Basset/MP-Basset, a model checker for message-passing Java programs. Our evaluation with fault-tolerant message-passing protocols shows that the proposed stateful optimization is able to reduce model checking time and memory by up to 69 % compared to the naive stateful search, and 39 % compared to partial-order reduction.

Contribution (C3): *Bounded Model Checking of Concurrent Programs based on Symbolic Projections [SBS15]*

In our third contribution, we propose a novel optimization of bounded model checking (BMC) for better run-time efficiency. Specifically, we define projections, an adaptation of dynamic program slices, and instruct the bounded model checker to check projections only. Given state properties over a subset of the program's variables, we prove the soundness of the proposed optimization. Furthermore, we propose a symbolic encoding of projections and implement it for a prototype language of concurrent programs. We have developed a tool called PBMC to evaluate the efficiency of the proposed approach. Our evaluation with various concurrent programs demonstrates the potential of projections to enable efficient verification.

Our focus in this thesis is on enhancing model checking of concurrent systems based on techniques that are orthogonal to POR. Additionally, we have developed two novel approaches based on the POR theory and applied them to multi-threaded programs. The results of this joint work have been published in [MSB+16; MSB+17].

Research Question (RQ3): *Can the interaction patterns between threads be harnessed to achieve a sound error propagation analysis for multi-threaded programs?*

Error propagation analysis assumes the ability to compare golden runs against faulty runs. The argument that there can only be a deviation between a golden run and a faulty run if a fault has been activated and propagated only holds for deterministic programs. Repeated execution of a non-deterministic program with identical inputs can deviate even when fault is not activated. For this reason, non-deterministic programs such as multi-threaded programs constitute a major challenge for EPA. Previous approaches work around this by ignoring certain aspects of the program, for example considering only control flow deviations [TP13], or using unsound methods such as likely invariants [EPG+07] as in [CWS+17]. We present next our final contribution, the first EPA approach to support multi-threaded programs.

Contribution (C4): *Nullifying Scheduling Non-determinism of Concurrent Execution Traces in Error Propagation Analysis [SWS+19]*

Modern computing systems improve application performance by relaxing execution determinism, for instance by allowing the CPU scheduler to interleave the execution of several threads. While beneficial for performance, such execution non-determinism affects programs' execution traces and hampers the comparability of repeated execu-

tions. Our final contribution proposes *Trace Sanitizer*, a novel approach for execution trace comparison in error propagation analyses (EPA) of multi-threaded programs. Trace Sanitizer can identify and compensate for non-determinism sources that are either due to dynamic memory allocation or non-deterministic scheduling. We formulate a condition under which Trace Sanitizer is guaranteed to achieve a 0 % false positive rate and automate its verification using SMT solving techniques. The key idea behind the formulated condition is that non-deterministic scheduling can be eliminated if the interaction pattern between the threads is deterministic. We perform a comprehensive evaluation of Trace Sanitizer on execution traces from the PARSEC and Phoenix benchmarks. We find that, unlike existing approaches, Trace Sanitizer can fully eliminate false positives without increasing the false negative rate for a specific class of programs.

1.4 PUBLICATIONS

The following published material has been, partly verbatim, included in this thesis:

- Habib Saissi, Marco Serafini, and Neeraj Suri. "Gyro: A Modular Scale-out Layer for Single-Server DBMSs". In: *USENIX Annual Technical Conference (ATC'19), (under submission)* (2019)

- Habib Saissi, Péter Bokor, Can Arda Muftuoglu, Neeraj Suri, and Marco Serafini. "Efficient Verification of Distributed Protocols Using Stateful Model Checking". In: *IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2013, pp. 133–142

- Habib Saissi, Péter Bokor, and Neeraj Suri. "PBMC: Symbolic Slicing for the Verification of Concurrent Programs". In: *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer. 2015, pp. 344–360

- Habib Saissi, Stefan Winter, Oliver Schwahn, Karthik Pattabiraman, and Neeraj Suri. "Trace Sanitizer: Eliminating Effects of Non-Determinism on Execution Traces". In: *International Symposium on Software Testing and Analysis (ISSTA'19), (under submission)* (2019)

The following previous publications, while related to different aspects covered in this thesis, have not been included:

- Tasuku Ishigooka, Fumio Narisawa, Kohei Sakurai, Neeraj Suri, Habib Saissi, Thorsten Piper, and Stefan Winter. *Method and System for Testing Control Software of a Controlled System*. US Patent 9575877. 2017

- Habib Saissi, Péter Bokor, Marco Serafini, and Neeraj Suri. "To Crash or Not To Crash: Efficient Modeling of Fail-Stop Faults". In: *Invited paper, International Workshop on Logical Aspects of Fault-Tolerance (LAFT in assoc. with LICS)*. Springer. 2011

- Can Arda Muftuoglu, Habib Saissi, Péter Bokor, and Neeraj Suri. "Scalable verification of distributed systems implementations via messaging abstraction". In: *ACM 23rd Symposium on Operating Systems Principles (SOSP) WiP section*. ACM. 2011

- Tasuku Ishigooka, Habib Saissi, Thorsten Piper, Stefan Winter, and Neeraj Suri. "Practical Use of Formal Verification for Safety Critical Cyber-Physical Systems: A Case Study". In: *IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*. IEEE. 2014, pp. 7–12

- Patrick Metzler, Habib Saissi, Péter Bokor, Robin Hesse, and Neeraj Suri. "Efficient Verification of Program Fragments: Eager POR". in: *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer. 2016, pp. 375–391

- Patrick Metzler, Habib Saissi, Péter Bokor, and Neeraj Suri. "Quick Verification of Concurrent Programs by Iteratively Relaxed Scheduling". In: *IEEE/ACM 32nd International Conference on Automated Software Engineering (ASE)*. IEEE Press. 2017, pp. 776–781

- Tasuku Ishigooka, Habib Saissi, Thorsten Piper, Stefan Winter, and Neeraj Suri. "Practical Formal Verification for Model-Based Development of Cyber-Physical Systems". In: *IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. IEEE. 2016, pp. 1–8

- Tasuku Ishigooka, Habib Saissi, Thorsten Piper, Stefan Winter, and Neeraj Suri. "Safety Verification Utilizing Model-Based Development for Safety Critical Cyber-Physical Systems". In: *Journal of Information Processing* 25 (2017), pp. 797–810

- Abraham Chan, Stefan Winter, Habib Saissi, Karthik Pattabiraman, and Neeraj Suri. "IPA: Error Propagation Analysis of Multi-Threaded Programs Using Likely Invariants". In: *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2017, pp. 184–195

1.5 THESIS ORGANIZATION

We structure the thesis based on the two parallelization levels introduced in Figure 2. In Part ii, we cover contributions dealing with distributed systems and we consider multi-threaded programs in Part iii.

In Chapter 3, we present our operation partitioning scheme to scale out distributed systems *(C1)*. Chapter 3 covers our decomposition-based approach to explicit model checking of distributed message-passing protocols *(C2)*. Subsequently, Chapter 4 introduces our projections based bounded model checking approach *(C3)* and Chapter 5 outlines our novel trace sanitizing technique for sound EPA *(C4)*. We summarize the overall contributions in Chapter 6.

Part II

DEPENDABILITY OF DISTRIBUTED SYSTEMS

2

SCALING OUT ACID APPLICATIONS WITH OPERATION PARTITIONING

This chapter presents operation partitioning, our first contribution. Operation partitioning employs a specially tailored static analysis on applications to infer how their workload can be distributed and uses Conveyor Belt, a novel distributed protocol, to achieve high availability with strong consistency guarantees. The content of this chapter is based on material from [SSS19].

We start by discussing the trade-off involved in partitioning the workload of distributed systems in Section 2.1. We then give an overview of the system built around operation partitioning in Section 2.2. Sections 2.3 and 2.4 describe the static analysis used by operation partitioning and the Conveyor Belt protocol. We prove the correctness of the protocol in Section 2.5 and describe our implementation Gyro in Section 2.6. Next, we describe two application uses cases and our evaluation in Sections 2.7 and 2.8, respectively. Finally, we discuss the related work (Section 2.9) before concluding the chapter.

2.1 THE PARTITIONING DILEMMA

Online transaction processing (OLTP) applications, such as online shopping services, bidding services, or social networking systems, need to scale in order to handle demanding workloads. One common way to increase capacity is to run the application on top of multiple servers, a process that is called *scale out*. These applications often use ACID transactions with *strong consistency guarantees*, which give the impression of being executed in some sequential order even if they are executed concurrently.

It is well known that strong consistency guarantees substantially simplify the design of applications, but make scaling out challenging. A common approach to scale out is *data partitioning*, which partitions the persistent state of the application across multiple servers. If a transaction needs to access data across multiple partitions, it is executed as a *distributed transaction*, which requires coordination across multiple servers. Distributed transactions are costly and represent the main bottleneck hindering scale out. The more servers we use, the more frequent distributed transactions become. As such, there is a bound on the degree of scale out that can be achieved with ACID

applications. For example, our evaluation shows that the TPC-W benchmark on MySQL Cluster reaches its peak performance with four servers, after which adding more servers is not beneficial anymore.

Scaling out efficiently entails solving two problems: finding a good way to partition data, and finding an efficient algorithm to keep servers consistent. In this chapter we introduce the concept of Operation Partitioning, a novel approach to address these two problems in an integrated manner.

Operation Partitioning takes an indirect approach to data partitioning. It maps each client operation to a specific server responsible for executing it, trying to associate conflicting operations to the same server whenever possible. This partitioning of the operations yields, indirectly, a (partial) partitioning of the data. By focusing on partitioning operations rather than data, Operation Partitioning makes it possible to perform partitioning based only on *static analysis* of the application code. This analysis is entirely automated, unlike existing data partitioning approaches that require human expertise and/or running samples of a workload in order to come up with good partitions (e.g. [CJZ+10; PCZ12]). In addition, the analysis can be applied to *unmodified* application code, without the need for the user to provide additional information about the semantics of the application.

Operation Partitioning not only makes partitioning easier, it also enables designing a more efficient coordination algorithm, called *Conveyor Belt* protocol, that guarantees serializability across multiple servers [Pap79]. The protocol obviates one of the main sources of inefficiency of distributed transactions: holding locks at multiple servers until a transaction is completed. Conveyor Belt is a *lock-free* protocol, which critically relies on the *operation classification* produced by the static analysis of the application code. Servers use a token passing scheme to execute “global" operations that, according to the classification, require coordination with other servers. When a server receives a global operation, it simply puts it on hold until it receives the token, without impairing the progress of other “local" operations that require no coordination. Once a server gets the token, global operations are executed efficiently in a batch. Our evaluation shows that the performance of Operation Partitioning is superior to data partitioning with distributed transactions, both in terms of performance with a given number of servers and in terms of maximum number of servers that can be effectively utilized.

Compared to recent techniques to speed up distributed transactions, such as Calvin [TDW+12], Lynx [ZPZ+13], Rococo [MCZ+14], Callas [XSL+15], and others [FA15; SLS+95; SCD+17], the Conveyor Belt protocol has two main advantages. First, existing techniques require additional information about the semantics of the application, which must be provided by the user and might not be trivially available, or might not be available at all in some application. In addition,

they require extending the application to provide this information and/or modifying the application code (e.g. to chop transactions). The Conveyor Belt protocol does not require any knowledge about the semantics of the application, as it only relies on the automatic Operation Partitioning process. This means that the Conveyor Belt protocol can be used to scale out *unmodified* applications. Second, these techniques must be implemented by designing a new database management or key-value store system. The Conveyor Belt protocol, by contrast, operates on top of *unmodified* single-server database management systems (DBMSs) providing ACID transactions. Using an unmodified DBMS, without requiring any specific low-level support for distributed transactions, makes it easier to run Conveyor Belt on top of a wide range of technologies as a middleware.

To show the practical viability of our approach, we present Gyro, a new middleware to scale out Java applications (Web applications running on Apache Tomcat in our use cases) and unmodified JDBC-compatible databases (MySQL in our use cases). We used Gyro to scale out two common OLTP benchmarks, TPC-W and RUBiS. In a LAN setup, where all servers are running within one datacenter, Gyro increases maximum throughput by 4.2x and decreases minimal latency by 58.6x compared to MySQL Cluster, a prototypical system based on data partitioning. This is particularly remarkable if we consider that Gyro is not only faster but also provides a significantly stronger consistency guarantee (serializability instead of read committed isolation, which is the only isolation level offered by MySQL Cluster). In a WAN (i.e., geographically distributed) setup, scaling out using Gyro reduces latency by up to 47.9x and increases throughput by up to 2.91x compared to a centralized setting.

Overall, this chapter makes the following contributions:

- We introduce Operation Partitioning, a scale out solution for OLTP applications that requires ACID transactions. Operation Partitioning is the first approach to use automated static analysis to indirectly partition data;
- present the Conveyor Belt protocol, an efficient lock-free coordination algorithm that relies on the operation classification produced by Operation Partitioning;
- implement Gyro, a middleware that uses Operation Partitioning to scale out unmodified DBMSs with ACID transactions;
- use Gyro to scale out TPC-W and RUBiS. In a LAN setting, Gyro outperforms MySQL Cluster by 4.2x in terms of throughput and 58.6x in terms of latency. In a WAN setting, Gyro improves throughput and latency by up to 2.9x and 47.9x respectively.

2.2 OVERVIEW

Operation Partitioning considers the problem of improving the throughput and latency of an ACID application running on top of a DBMS by scaling out, i.e., running instances of the DBMS on top of multiple servers. These DBMS instances are kept consistent by running the Conveyor Belt protocol on top of them. The protocol coordinates the execution of operations and guarantees serializability. We now give an overview of the steps required by Operation Partitioning.

Offline Static Analysis. The Operation Partitioning process consists of three main steps, which are separate but intertwined. First, an *automated partitioning* step is performed to determine how to partition operations. Operation Partitioning requires that the code of the application is known a priori. This is a sound assumption for many Web and enterprise OLTP applications, since they typically run a fixed set of transactions. The partitioning algorithm statically analyzes read-write conflicts between operations to minimize cross partition conflicts. Partitioning avoids coordination by routing conflicting operations to the same server as much as possible. Operations that have no conflicts with operations at other servers can be executed locally and immediately, without coordination with other servers. In particular, partitioning tries to minimize the type of conflicts that require coordination in the Conveyor Belt protocol. We describe the automated partitioning algorithm in Section 2.3.1.

Next, the *operation classification* step uses the partitioning obtained in the previous step to classify operations as local or global, based on the amount of coordination they require. Unlike global operations, local operations can be executed immediately without distributed coordination. Operations classification, which is also an automated process, is described in Section 2.3.2.

Online Scale-Out Algorithm. The previous two steps of offline analysis produce a partitioning criteria and an operation classification. These are taken as input by the *Conveyor Belt protocol*, which runs the application on multiple servers and ensures consistency. The protocol is described in Section 2.4.

The protocol is implemented by Gyro, a scale out middleware that integrates with unmodified applications and interfaces with unmodified external DBMSs. We describe the technical details of this integration in Section 2.6.

2.3 OPERATION PARTITIONING

We start by describing the first two steps in our approach: automated partitioning algorithm and operations classification.

Application Code: Transactions vs. Operations. We consider applications keeping all their persistent state in a DBMS. The application

code consists of a set of **transactions** that modify the state of the DBMS. Transactions are expressed as procedures having a certain number of input parameters. For example, a transaction could be the procedure `createCart(sid)`, which creates a shopping cart with id `sid`. An **operation** corresponds to a request to execute the transaction with a set of concrete values for its input parameters. For example, a client operations can invoke the operation `createCart(5)` to create a cart with id 5.

Operation Conflicts. The application state is stored by the DBMS, and logically consists of a set of variables (i.e., tuples). A state assignment (or simply state) S assigns a value to each variable accessed by the application. Let O be the set of all possible operations that can be executed by the application. The read set $R(o)$ of an operation $o \in O$ consists of all variables that o may read when it executes on any state S. Similarly, the write set $W(o)$ of o is the set of all variables that o may write to if it executes on any state S. Two operation o_1 and o_2 have a *write conflict* if their write sets intersect, i.e., $W(o_1) \cap W(o_2) \neq \emptyset$. We say that o_1 *reads from* o_2 if $R(o_1) \cap W(o_2) \neq \emptyset$. In either cases, we say that o_1 and o_2 conflict with each other.

2.3.1 *Automatic Partitioning*

The automatic partitioning step generates a partitioning of operations that minimizes conflicts. We now describe how we automate this process.

To identify operation conflicts we need to specify read and write sets of the operations. First, we show how to extract and express read and write sets from the source code. Next, we describe the automated partitioning algorithm, which takes read and write sets as input and determines an *operation partitioning array* P. The operation partition array associates every transaction t to one of its input parameters. This *partitioning parameter* is used by the Conveyor Belt protocol to route every operation o of type t to a server. After an operation partitioning array P is determined, classifying operations is straightforward and automatic as we will see.

Extracting Rread/Write Sets. An OLTP application usually has a relatively small number of transaction, which can correspond to a huge number of possible operations. Therefore, the operation Partitioning algorithm operates at the level of granularity of transactions, and for each transaction determines a read and a write set. These sets are determined in a static and pessimistic way: they include all variables that could be accessed in any execution performed against any database state. An entry e in either sets is a pair $e = \langle A, C \rangle$, where A is a set of *accessed attributes* and C is a *condition*.

The accessed attributes set in the read set contains all table attributes (i.e., columns) that are read and returned as output of the transaction.

In the write set, it contains all table attributes that are updated by the transaction. The condition of a read or write set is the predicate used to select the specific rows in the table for which the attributes are modified.

Read and write sets are generic concepts, but we now give a concrete example based on the type of applications we targeted in this work. These applications consist of a set of transactions that access a database through SQL queries. Consider for example the `doCart` transaction in the TPC-W benchmark, which updates a shopping cart with id `sid` by adding, removing or updating item with id `iid` in a quantity `q`. The pseudocode of the transaction is the following:

```
doCart(sid, iid, q){
    ...
    exec("UPDATE SHOPPING_CARTS
    SET QTY = q WHERE ID = sid
    AND I_ID = iid");
    ...
}
```

Operation Partitioning extract reads and write sets by looking at *all* SQL statements contained in the transaction, regardless of the execution path. While conservative, this approach has proven good enough for our purpose. We used Java parser [Jpa] to extract SQL queries and to map input parameters to the used query parameters.

With this information at hand, we can define read and write sets. Each SQL statement corresponds to an entry in one of the sets. Consider for example the SQL statement highlighted in the pseudocode and rename the table `SHOPPING_CARTS` as `SC` for brevity. This statement corresponds to a write set entry e. The accessed attribute for e is specified in the `UPDATE` clause, so $e.A =$`SC.QTY`. Insert SQL query also correspond to entries in the write set and their accessed attribute is specified in the `INSERT` statement, while for read set entries the accessed attribute corresponds to the `SELECT` query. The condition of the entry corresponds to the content of the `WHERE` clause of the query, so in this case $e.C = (\texttt{SC.ID = sid} \wedge \texttt{SC.I\_ID = iid})$. The condition binds the value of the input parameters of the transaction, which are `sid` and `iid` in this case, with the values of the table attributes of the specific rows for which the attributes in $e.A$ are accessed by the transaction, `SC.ID` and `SC.I_ID = iid` in our example.

Conflict Detection Phase. The partitioning algorithm is illustrated in Algorithm 1. The first phase of the algorithm is *conflict detection*, which looks at all pairs of transactions that have a conflict on some table attribute. A conflict between transactions occurs if some of the operations relative to these transaction can conflict, according to the definition of Section 2.3.2. For each pair of transactions (t, t'), it builds a condition predicate $C_{t,t'}$, in disjunctive normal form, that expresses the condition that the values of the input parameters of t and t' must

Algorithm 1: Partitioning algorithm.

input : Set T of transactions
input : Read set R_t and write set W_t for each transaction t
output: Array P of partitioning parameters $P[t]$ for each transaction t

```
// Conflict detection
```
foreach *pair* $t, t' \in T$ **do**
  $C_{t,t'} \leftarrow$ *false*;
  if $\exists r \in R_t, w \in W_{t'} : r.A \cap w.A \neq \emptyset$ **then**
    $C_{t,t'} \leftarrow C_{t,t'} \vee (r.C \wedge w.C)$;
  if $\exists w \in W_t, r \in R_{t'} : w.A \cap r.A \neq \emptyset$ **then**
    $C_{t,t'} \leftarrow C_{t,t'} \vee (w.C \wedge r.C)$;
  if $\exists w \in W_t, w' \in W_{t'} : w.A \cap w'.A \neq \emptyset$ **then**
    $C_{t,t'} \leftarrow C_{t,t'} \vee (w.C \wedge w'.C)$;
  if $C_{t,t'}$ *is satisfiable* **then**
    Conflicts $\leftarrow$ *Conflicts* $\cup\, C_{t,t'}$;

```
// Partitioning optimization
```
return $\min_P$ cost$(P,$ *Conflicts*$)$;

```
// Estimate the volume of conflicts
```
function cost(P, *Conflicts*)
  foreach $C_{t,t'} \in$ *Conflicts* **do**
    $k \leftarrow P[t]$;
    $k' \leftarrow P[t']$;
    foreach *table attribute* A **do**
      remove all clauses $(k = A \wedge k' = A \wedge \ldots)$ from $C_{t,t'}$;
    if $C_{t,t'}$ *not satisfiable* **then**
      remove $C_{t,t'}$ from *Conflicts*;
  return $\sum_{C_{t,t'} \in \mathit{Conflicts}} \mathit{weight}(t) + \mathit{weight}(t')$;

take so that a conflict occurs on the same row(s) of the same table(s). In other words, the condition characterizes the set operations of the two transactions that are conflicting. If a conflict between the two transactions is possible, $C_{t,t'}$ is added to a set called *Conflicts*. Note that we also consider self-conflicts, that is, conflicts between two operations of the same transactions where $t = t'$.

Let us consider again the TPC-W example. The `createCart` transaction creates a new row in the `SHOPPING_CARTS` table (again renamed `SC` for brevity) such that `SC.ID = sid`, where `sid` is the id of the shopping cart and is an input parameter of `createCart`:

```
createCart(sid){
   ...
   exec("INSERT INTO SHOPPING_CARTS
   (ID) VALUES (sid)");
   ...
}
```

The write set of `createCart` contains entry $e = \langle$ `SC.ID,SC.ID = sid` $\rangle$. Given the write set of `doCart`, we derive that there is a write-write conflict between the two transactions with condition $C_{t,t'}$:

$$(\texttt{SC.ID = sid}) \wedge (\texttt{SC.ID = sid'}) \wedge (\texttt{SC.I\_ID = iid'})$$

where `sid` is a parameter of `createCart` and `sid'` and `iid'` are parameters of `doCart`.

Partitioning Optimization Phase. The next phase is called *partitioning optimization* and it finds the operation partitioning array P that minimizes global operations, as defined in Section 2.3.2. The partitioning can reduce the cost of conflicts by mapping two conflicting operation to the same partition, and thus server, such that the conflict becomes local.

The cost function finds out the potential an operation partitioning has to eliminate conflicts. Consider two transactions t and t' that conflict, and let k and k' be the parameters used for their partitioning. Operation Partitioning uses the same deterministic routing function for all operations, so two operations with the same value of their partitioning parameters k and k' will be sent to the same server. Therefore, all conflicts that arise because of a necessary condition $k = k'$ will be local to one server, and they will not require global coordination. The most common case when this condition arises is when k and k' are used to identify a row based on the value of the same attribute A, so there is a clause in the conflict condition of the form: $(k = A \wedge k' = A \wedge \ldots)$

Let us revisit again our running TPC-W example and let P be an operation partitioning array such that `sid` is the partitioning parameter for both `doCart` and `createCart` transactions. The conflict condition in the previous equation is of the form $(k = A \wedge k' = A \wedge \ldots)$, where k =`sid`, k' =`sid'`, and A =`SC.ID`. This condition is equivalent to saying that the conflict among the two transactions arises only if `sid=sid'`. As the same deterministic routing function is used for both transactions, conflicting operations will always be sent to the same server. This means that we can remove this conflict from the *Conflicts* set.

After removing all conflicts that become local thanks to an operation partitioning array P, we can estimate the cost of the remaining global conflicts by summing up the weight of the conflicting transactions in *Conflicts*. If we assign to each transaction a weight of 1, the algorithm will minimize the number of conflicting transactions. If an estimate of the relative frequency of the transaction is known, it can be used as a weight to improve cost estimation.

The algorithm searches for the operation partitioning array that minimizes the cost. In the workloads we considered, and in most practical transactional workloads, the number of transaction types and their parameters is not very large, so an exhaustive search of all possible partitionings to find the best one is feasible. However, the algorithm can also use of more sophisticated search strategies.

Multiple Partitioning Parameters. The full algorithm also considers multiple partitioning attributes by looking at each parameter independently to find a partition. If in all cases the resulting partition is the

same, we consider the operation local and send it to that partition. Otherwise, it is not possible to map the operation to one partition and it is marked as global.

Applicability of The Algorithm. Although our static analysis tool targets transactional applications using SQL statements, Algorithm 1 is generic and can be applied to other types of applications. For example, a key-value store can be seen as a single table with two attributes. In our implementation, however, we target application code using basic SQL queries. For partitioning, we require that potential partitioning parameters are involved in `WHERE` clauses only in atomic conditions in an equality form. The rest of the clause can contain arbitrary conditions. Parameters used in atomic conditions that are not in equality form are ignored for partitioning, and other alternatives are tried out. We also do not consider complex SQL constructs such as nested queries and triggers.

2.3.2 *Classes of Operations*

With a partitioning of operations at hand, we can now describe the operation classification logic.

Operation Partitioning identifies two classes of operations: *local* and *global*. Local operations are partitioned, so they need to be executed by a specific server, but they do not require prior coordination or to be replicated. Even though a local operation l can have conflicts, no operation executed at a different server than the one assigned to l depends on the effect of executing it. On the other hand, global operations require coordination before they are executed and are replicated.

Local and Global Operations. Consider now the set O of operations that have some conflict with some other operations. We classify these operations as local or global by first partitioning them and by assigning each partition to a different server in the system. We then classify each operation as follows. An operation o is a *local operation* if: (i) o does not have a write conflict with any other operation in a different partition, and (ii) no other operation from a different partition reads from o. We denote with L_p the set of local operations in the partition assigned to a server p. A local operation l associated to a specific server can be executed immediately at that server without any prior coordination. In fact, it follows from conditions (i) and (ii) that no other operation associated with another server depends on the effects of l.

The rest of operations that are not local are called *global operations*. We denote with G_p the set of global operations in the partition assigned to server p. Since executing global operations entails coordination among servers in Conveyor Belt, it is important to find an operation partitioning that minimizes them. Note that global operations are also assigned to partitions, and are therefore only executed by a dedicated

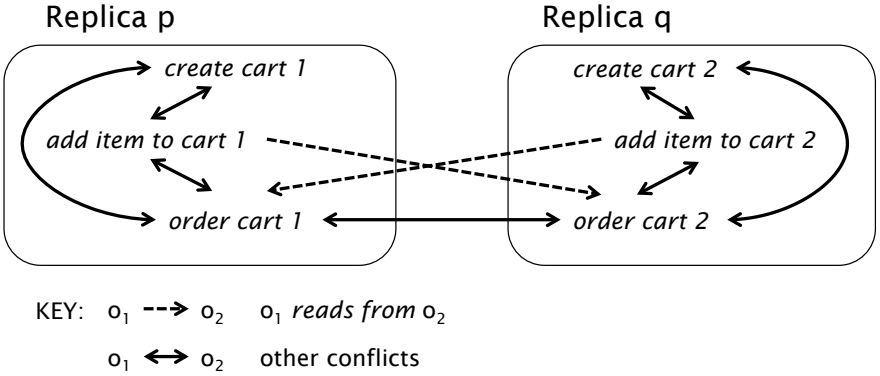


Figure 4: A classification of operations in the online store example. The *order* operation is global, the other operations are local.

server, because they may read from other local operations which are only seen by that server. Allowing global operations to execute on arbitrary servers might results in them attempting to read unavailable data.

Example. Consider the example of an online store application whose operations must be classified. The application has transactions that allow clients to create a cart, add and remove items, and eventually proceed to checkout. Each cart is assigned a unique id. For each cart id c we have the following three operations: *create* a cart c, *add* a quantity a of items of type t to c provided that there are sufficient items on stock, and finally *order* all items currently present in the cart c.

Assume that operations are partitioned based on the value of the cart id c. The conflicts among operations in two sample partitions are illustrated in Figure 4. For example, operations that add an item to a cart do that only if the item is available in the stock. The stock level of an item can be modified by order operations, which remove elements from it. Therefore, add operations on cart c read from order operations for the same cart, and order operations have write conflicts with other order operations on different carts.

Operation Partitioning classifies operations according to the partitioning and their conflicts as follows. Order operations are *global* because they have write conflicts with operations in other partitions, and because add operations read from them. All other operations are *local* because they either have no conflicts with operations at other servers (e.g., create cart operations) or they only read from remote operations (e.g., add to cart operations).

Comparison With Data Partitioning. Operation Partitioning can be seen as a mechanism to generate partial data partitions. The union of write sets of every local operation in a partition corresponds to a data partition. The subset of the write set of a global operation that is part of cross partition conflicts constitute the set of items that

have to be replicated. For instance, in Figure 4, there are two data partitions each containing a cart. Every cart is assigned to the partition of the operations manipulating it. Since the order operation, a global operation, writes to an item that is also written to by order operations from other partitions, the item has to be replicated.

Conversely, it is possible to get an Operation Partitioning from a data partitioning scheme. Given a data partitioning function f that maps every data entry to a partition, we generate an operation partitioning by ensuring that every local operation has no entries in its conflict set belonging to two distinct partitions. More formally, an operation o is local if for every conflicting operation o', for every variables $x, y \in W(o) \cap W(o')$ or $W(o) \cap R(o')$, we have $f(x) = f(y)$. In that case, the server responsible for o also maintains the partition $f(x)$.

2.4 THE CONVEYOR BELT PROTOCOL

We now describe Conveyor Belt, a lock-free scale-out coordination algorithm that runs applications on multiple servers to increase their performance compared to a single-server configuration. The protocol considers applications where Operation Partitioning has already been applied to classify the application's operations. The classification allows a server to immediately execute and reply to as many local operations as possible without coordinating with other servers. Conveyor Belt implements *serializability* [Pap79], i.e, all clients observe the same sequential execution order of operations. We provide a correctness proof of the protocol in the Section 2.5.

Preliminaries. We start by clarifying the functioning of the application running with Conveyor Belt. The application is ran by an event-driven multi-threaded server. Whenever a event, such as the receiving of a client request, is triggered, the server dispatches its handling to a thread. When a server receives requests (REQ message) from clients, the assigned thread executes the request and sends a (REPLY message) back or a redirect message (MAP message) if the client contacted the wrong server.

Conveyor Belt orchestrates the execution of the application by invoking its request execution logic. In the pseudocode, we abstract this logic as an $execute(o)$ function, which executes an operation o and produces a reply r, along with a state update u that we describe shortly. We consider applications that store their state on a database management system (DBMS). Each server runs a local stand-alone DBMS instance, i.e., instances of the DBMS at different servers do not communicate with each other. When an application executes an operation, it accesses its local state by invoking a sequence of database queries on the local DBMS instance. All queries invoked by the same operation are enclosed within a single database transaction. We as-

Algorithm 2: The Conveyor Belt algorithm for server p.

```
1  upon receive ⟨REQ, o, c⟩ msg from client c where
2      if o ∈ L_p then
3          r, ∗ ← execute(o);
4          send ⟨REPLY, r⟩ msg to c;
5      else if o ∈ G_p then
6          append ⟨o, c⟩ to Q;
7      else
8          q ← replica such that o ∈ L_q ∪ G_q;
9          send ⟨MAP, q⟩ msg to c;
10 upon event receiveToken(T)
11     foreach ⟨u, q⟩ ∈ T do
12         if p = q then
13             remove ⟨u, q⟩ from T;
14         else
15             apply(u);
16     Q′ ← atomic-snapshot(Q);
17     foreach ⟨o, c⟩ ∈ Q′ do
18         r, u ← execute(o);
19         append ⟨u, p⟩ to T;
20         send ⟨REPLY, r⟩ msg to c;
21         remove ⟨o, c⟩ from Q;
22     passToken(T);
```

sume that the DBMS can execute transactions in parallel and guarantee serializability.

The state update u returned by $execute(o)$ is the update-only query that includes all updates to the database generated during the execution of o. Extracting u is one of the features of the Gyro system, which we detail in Section 2.6. When value of the state update is irrelevant, we it this with a star.

Conveyor Belt executes each operation only once, at a single server. It replicates the effects of operations executed at other servers simply by directly applying the corresponding state update onto the local DBMS instance. This is denoted in the pseudocode by the function $apply(u)$.

The algorithm requires executing operation partitioning and classification as preliminary steps before it is started. These steps partition operations among the sets $\{L_1, \ldots, L_N, G_1, \ldots, G_N\}$, where N is the number of servers in the system and L_p and G_p contain the local and global operations, respectively, assigned to server p.

Handling Local Operations. The pseudocode of the Conveyor Belt algorithm for a server p is shown in Algorithm 2. The algorithm handles operations differently based on their classifications. Local operations are executed locally and a reply is immediately sent back to the client without coordination (Lines 2-4).

We now explain why local operations do no require coordination. As discussed in Section 2.3.2, the updates made by local operations are not directly read by any other operation running at other servers. However, these updates might indirectly impact remote operations

transitively, my means of a global operation. Consider again the cart example of Section 2.3.2. If a (local) *add* operation adds an item i to a cart and a subsequent (global) *order* operation places an order for that cart, the result is that the stock of item i is reduced, and this impacts operations at all servers. The Conveyor Belt protocol in this case only propagates the state update of the *order* operation, which includes the reduction in the stock of item i. This is sufficient to ensure serializability, and there is no need to propagate the fact that the item was previously added to the cart. The protocol can thus avoid propagating local operation thanks to its use of state updates, or in other words, of passive instead of active replication [JS13]: propagating and executing operations at all servers would require Conveyor Belt to propagate the *add* operation as well in order to guarantee a consistent execution of the *order* operation at all servers. we refer the reader to the proof of correctness Section 2.5.

Handling Global Operations. Global operations require coordination among servers to agree on a total order of execution. Conveyor Belt uses a token based scheme. The token is passed around in a predefined order to ensure that global operations are totally ordered. At any time, only the server holding the token, also called the primary, is allowed to execute global operations. Otherwise, the server appends the operations to a queue Q for execution at a later time. Note that the queue Q must be thread-safe since Algorithm 2 is multi-threaded.

Handling the Token. Upon receiving the token T, server p invokes one RECEIVETOKEN(T) event at p, becoming the primary server. Like any other event, a specific RECEIVETOKEN(T) event is handled in isolation by a single thread, while multiple other threads might be concurrently handling client requests. The token contains a sequence of tuples $\langle u, q \rangle$ where u is the update of a global operation that has previously been executed at some server q. As soon as a server becomes primary, it applies all the updates in the token that are from other servers and removes its own updates as they have been already applied at all other servers (Lines 11-15). Next, the primary needs to execute the global operations that have been enqueued locally into Q. In order to ensure liveness, the primary copies an atomic snapshot Q' of the Q queue containing global operations submitted to p that have been waiting for execution (Line 16). This is because Q is concurrently modified by multiple threads. Without copying an atomic snapshot, p might stay stuck executing incoming global operations in Q that are constantly being appended by other threads, and never give up the token. Then, p iterates over all global operations that have been pending up to that point in the Q' queue (Lines 17-21). The server executes each operation o in the queue, sends a reply r to the client c and appends the resulting update u to the token before removing the operations that have been appended.

For efficiency reasons, our actual implementation of the Conveyor Belt protocol executes the operations in Q' in parallel. Consequently, the DBMS executes multiple concurrent transactions generated by these operations. Gyro must be able to extract the logical serial order in which the DBMS executes these concurrent transactions, since this serial order must be the same as the order in which the corresponding state updates are added to the token and thus applied at other servers. Section 2.6 describes how Gyro achieves this.

Finally, the server gives up the primary role by calling PASSTOKEN(T) to pass the token to the next server (Line 22).

Redirections. If clients know how the operations are partitioned, they send their operations directly to the server responsible for it. This should be the common case as it is for our example applications. However, if clients send an operation to the wrong server, the server will reply with the identity of the server responsible for the operation (Lines 8-9).

Fault Tolerance. The Conveyor Belt protocol considers replication for fault tolerance as a complementary and orthogonal issue. More precisely, the protocol assumes that each of the servers can tolerate faults and there is no message loss. Making a server fault tolerant is an orthogonal issue: for example, a Paxos group could implement the abstraction of a logical fault tolerant server. For message loss, the token can simply be passed using a reliable channel among fault-tolerant servers.

2.5 CORRECTNESS PROOF

In this section, we prove that the Conveyor Belt protocol (Algorithm 2) correctly implements serializability.

2.5.1 *Token-Passing Scheme*

The protocol uses a primary-backup scheme to execute global operations, based on a token passing scheme that acts as a broadcast algorithm. We now show the properties of the token passing algorithm.

Lemma 1. *The token passing scheme used by the Conveyor Belt protocol to broadcast state updates of global operations satisfies Primary Order atomic broadcast [JRS11; JS13].*

Proof. A Primary Order atomic broadcast protocol satisfies the standard properties of atomic broadcast, namely:

- *integrity:* if some server delivers an update u then some server has appended u to the token;
- *total order:* if some server delivers update u before u' then any server that delivers u' must deliver u before u';

- *agreement:* if some server p deliver u and some other server q delivers u', then either p delivers u' or q delivers u.

These properties holds for the token scheme since the order of the updates appended to the token is never altered and updates are only removed from the token once all servers has received them (Line 13).

There are two additional properties that Primary Order atomic broadcast satisfies. The first additional property is *primary order:* servers must apply updates in the order in which they were broadcast by the primaries that produced them. This is necessary because otherwise older state updates might overwrite values written by newer state updates. In particular, *local* primary order requires that the delivery order is consistent with the local broadcast order of a primary during each primary epoch, while *global* primary order requires that the delivery order is consistent with the total order of the primary epochs in which the message was broadcasted.

The second property, called *primary integrity*, guarantees that the primary role can transition safely from one server p to another server q. Primary integrity requires the following: if a new primary epoch e starts at server p, a state update u is broadcasted during a prior primary epoch, and u is eventually delivered by some server, then p must deliver u before it starts e. This property guarantees that the new primary p obtains the full final state resulting from previous epoch *before* it starts producing new state updates. State updates are incremental and they should only be applied on the state from which they were produced. Interleaving state updates from different epochs can result in incorrect executions.

It is easy to see that the token passing scheme satisfies these properties. Primary order is guaranteed because the updates are appended to the token in the same order of their execution by the primary (Lines 18-19) and because of the total order preservation of the token. This is true because the used queue is atomic updates appended to the token are applied by all the other servers in that same order (Lines 11-15). The token scheme satisfies primary integrity by ensuring that every primary applies all the updates in the token from previous epochs before executing pending global operations (Lines 11-15). □

2.5.2 *Serializability Proof*

We now show that Conveyor Belt guarantees serializability, that is, the relative order of global operation is consistent across all servers. Before the proof, we need to introduce some notation and definitions.

Definitions. In the Conveyor Belt protocol, operations are executed concurrent by multiple threads. However, they are executed by an underlying DBMS which, by assumption, serializes their execution. Therefore, we consider in the proof that each server executes operations in a sequential total order. A DBMS running at a server only

executes state updates of global operation of other servers. In the proof, we will not distinguish between the two cases of executing a global operation or its state update, that is, we say that a server executes a global operation g also when it executes the corresponding state update.

We call an *execution* of the system the sequence of operations invoked on the distributed system up to a given time, and pair each operation with the reply that the service produced for it. In order to show that the Conveyor Belt protocol satisfies *serializability* [Pap79], we need to show that, for each possible execution e, there exists a total order T of the operation-reply pairs of e such that executing of the operations in the specified order on a single instance of the service will produce the same replies as in e.

It is important to stress that this total execution order T is a *logical* order. It describes how clients observe the behavior of an application scaled out by a coordination protocol like the Conveyor Belt protocol. The actual implementation of the coordination protocol simply has to exhibit a behavior that is *equivalent* to this total order. The implementation of this property is protocol-dependent. In the Conveyor Belt protocol, for example, local operations are not executed by all servers. Nonetheless, local operations are still totally ordered in T, and the system behaves *as if* these operations were executed by all servers.

We now introduce additional notation. Let T'_p be the execution order of all global operations executed in e relative to server p. T'_p is defined as follows. Let g be a global operation that appears in e. If $g \in G_p$, then T'_p orders g according to the order in which g is executed at p. Else, T'_p orders g according to the order in which the state update generated from g is applied at p. Since in each server the token thread executes and applies global operations sequentially, T'_p is a total order, and it reflects the order in which global operations modify the state of p. Note that the total order for two servers might contain a different set of operations. For simplicity, we treat sometimes T'_p as a set and use the notation $g \in T'_p$ to say that operation g appears in the total order T'_p.

Serializability Proof. The proof is in three steps. First, we show that each server orders pairs of global operations consistently. Next, we show that the relative execution order of local and global operations is consistent across all servers. Finally, we show that pairs of local operations are executed consistently.

Lemma 2. *Given two servers p and q, their total orders T'_p and T'_q have a common prefix which includes every operation in $T'_p \cap T'_q$.*

Proof. This lemma directly follows from the fact that the state updates of global operations are broadcasted and delivered using the token scheme. The token scheme guarantees that the delivery order of state updates is consistent across all servers. , so the pairwise order of global

operations in $T'_p \cap T'_q$ that are neither in G_p nor in G_q is consistent in T'_p and T'_q, and we are done.

Consider now the ordering of two global operations g and g' such that $g \in G_p$ or $g \in G_q$ (remember that global operations are partitioned). We consider only the case $g \in G_p$ without loss of generality, and we have two sub-cases:

Case I: If $g' \in G_p$, then the primary order property of the token scheme guarantees that the delivery order of the state updates of g and g' at q is consistent with the order in which the operations were executed by p, and we are done.

Case II: If $g' \notin G_p$, we consider two sub-cases:

Case II.a: g' precedes g in T'_p. The primary integrity property guarantees that, before p becomes a primary and starts executing new operations, p also delivers all operations sent by previous primaries that are ever delivered by q. Therefore, g' is delivered at p before g is executed, so the same order will appear in T'_q.

Case II.b: g' follows g in T'_p. There exists some server r such that $g' \in G_r$. If r sends g' in a primary epoch after p sends g, then r delivers g before it executes and sends g' by primary integrity. Therefore, server r delivers g before g'. Because of the total order guaranteed by the token scheme, every server must deliver g before g', including p. This implies that g must precede g' in T'_p, which is a contradiction.

If r sends g' in a primary epoch before p sends g, then g' precedes g in T'_p by primary integrity. It follows that g' precedes g in T'_q too. Assume by contradiction that this does not hold and g precedes g' in T'_q. This would imply that q delivers the state update for g before the one for g'. Because of the total order property of the token scheme, also p should deliver the state updates in the same order, so p should deliver the state update for g before producing it, a contradiction. □

Since all servers order global operations consistently, we define the order of global operations in the total order T according to the order T'_p of any server p.

Next, we can show how pairs of operations, one local and one global, are ordered among each other. Let $l \in L_p$ be a local operation executed by server p. Let B^l_p (resp. A^l_p) be the set of global operations whose state updates have been delivered at p before (resp. after) p executes l. The total order T orders l after all global operations in B^l_p and before the global operations in A^l_p.

For this order to be sound, we need to show that l the state updates of all global operations in B^l_p are reflected in the state upon which l is executed, and that the state update of l is reflected in the state upon which all global operations in A^l_p are executed. The first claim directly follows from the fact that l is executed by p after the operations in B^l_p. We now show the second claim.

Lemma 3. *The state update generated by executing l at server p is applied to the state upon which each global operation $g \in A_p^l$ is executed.*

Proof. We consider two cases. *Case I:* If $g \in G_p$, then g is executed at p and after l, by definition, so the state updated of l is reflected in the local state of p upon which g is executed, and we are done. *Case II:* If $g \in G_q$ with $q \neq p$, then g does not directly read any variable from l by definition (see Section 2.3.2). However, assume that, if operations were executed in the total order T, the state update of l would determine the state upon which g is executed through one (or more) operation o that reads a value v written by l and, because of reading v, writes some value read by g. We need to show that these operations are actually executed before g. We consider the case where there is only one operation o propagating the changes of l to g. If o reads from l then o is an operation assigned to server p. If g, which is assigned to server q, reads from an operation o at another server p, then o is a global operation. As shown in Lemma 2 that global operations are consistently ordered by all servers. We have also shown that this order reflects the execution order of global operations, so if o precedes g in the total order T, then the state update of o takes effect before g is executed. The case where l influences a global operation in A_p^l through a chain of operations $o_1, \ldots, o_n$ follows by induction using a similar argument for each pair of subsequent operations in the chain. □

The last case to consider is the ordering of pairs of local operations l_1, l_2. If $l_1, l_2 \in L_p$ are executed by same server p, their correct order follows the local execution order at p. The ordering between two local operations at different sites can be arbitrary, since neither observes the other by definition (see Section 2.3).

After showing that the pairwise order of all operations form a consistent total order, and that this total order is consistent with the execution order of the operations, we can conclude that:

Theorem 1. *The Conveyor Belt protocol (Algorithm 2) satisfies serializability.*

2.6 THE GYRO SYSTEM

We have developed Gyro, a middleware that uses the Conveyor Belt protocol at its core to ensure coordination-freedom of local operations. Gyro supports multi-threaded applications, where concurrent threads execute operations on a shared application state. Each server stores the application state in a local DBMS offering serializable transactions. We implemented Gyro in Java and it consists of about 2k lines of code.

Overview. In our implementation the mutli-threaded application is a web application with a pool of threads to handle incoming HTTP

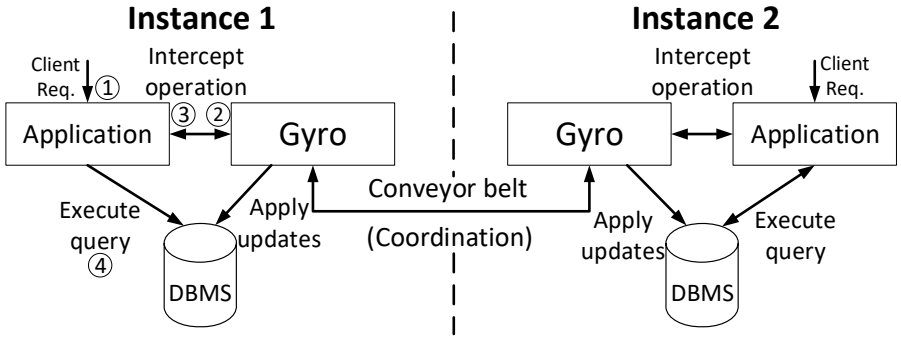


Figure 5: Gyro system overview in a deployment with two servers. The numbered arrows indicate the execution flow of operations that are mapped to the server. Global operations are held after step 2 until the server acquires the token. The Conveyor Belt protocol propagates state updates for global operations executed at other servers. Gyro directly applies these state updates onto the DBMS.

requests. Gyro works by intercepting the interaction between the application threads and the DBMS.

A key design choice in Gyro was that we wanted to scale out unmodified applications running on top of an unmodified external DBMSs. In particular, the DBMS is seen as a black box by Gyro. Our current implementation of Gyro intercepts JDBC interactions between application threads and the DBMS, so we can support any data store offering a JDBC interface interacting with a serializable DBMS. For example, in our evaluation we use (unmodified) MySQL as underlying data store. This makes Gyro easier to adopt and allows dealing with fault tolerance at the data store level. Furthermore, the application does not have to be manually modified to work together with Gyro but is instead automatically instrumented.

In Figure 5, we show an overview of a multi-threaded server application running with Gyro. Gyro interacts directly with the application, and the other servers. Before an application starts executing an operation, it invokes Gyro and waits until it is allowed to proceed according to the Conveyor Belt protocol. When it is time to execute the operation, Gyro gives back control to the application to resume. Next, the application executes an instrumented version of the operation that allows the recording of the resulting updates. The application interacts directly with the DBMS and delivers the operation updates in the same order of their execution back to Gyro. Additionally, Gyro runs a separate module that deals with token passing and applies received updates directly to the database.

In the description of the Conveyor Belt in Section 2.4, we left out a few implementation details about the interaction between Gyro, the application server, and the underlying DBMS, namely: how to extract state updates, how to manage and parallelize the execution of global operations, and how to trace the sequential execution order of global operations. We describe these details in the following.

Extracting State Updates. First, we describe how Gyro extracts a state update from the execution of an operation. Gyro does this by intercepting the execution flow of operations in the application. We do this by automatically instrumenting the application code to enable the interaction with Gyro. An operation can execute multiple accesses to the underlying DBMS. Before the first access, the operation starts a transaction, which is terminated when the operation terminates. This ensures that the local execution of concurrent operations is equivalent to a sequential execution on the DBMS. The interaction between the application and Gyro is completely transparent. In order to obtain the state updates produced by global operations, Gyro records changes to the DBMS state by intercepting interactions between the application and the DBMS, which occur through JDBC. Every time the application begins executing a global operation, our instrumentation generates an operation object that is used to store the state updates. Gyro then uses the operation object as a wrapper to JDBC: every time the application invokes a statement s mutating the state (e.g., `UPDATE`), it does so through the operations object instead of JDBC. The operation object appends s to the sequence of SQL query statements invoked within the operation and then passes s to JDBC. At the end of the transaction, the sequence of SQL statements in the operation object represents the sequence of state mutations that can be executed by other servers to reproduce the operation, that is the update that has to be passed to the other servers.

Parallelizing The Execution of Global Operations. We now describe how our implementation of the Conveyor Belt protocol in Gyro handles global operations. The handling of local operations is identical to the description in Algorithm 2, but Gyro optimizes the handling of global operations by executing them in parallel. As discussed in Section 2.4, there is a single thread that handles the event of receiving a token. We call this thread the *token thread*. We call the other threads that handle global operations, and are waiting for the permission of executing them, the *handling threads*. In Algorithm 2 the handling thread appends a global operation to Q and returns, leaving the actual execution of the operation to the token thread. This would require the handling thread to store a copy of the HTTP request handling context necessary to reply to the client and make it available to the token thread. As this would induce a substantial overhead, we opted for having the handling thread wait for the server to receive the token before executing the operation with the necessary HTTP request context. Concretely, we extend the queue Q to contain an initially locked lock for every pending operations that the handling thread attempts to acquire and goes to sleep until it is unlocked by the token thread. When the token thread executes, it iterates over the pending operations and notifies the sleeping threads to proceed and execute the operation. The token thread then blocks until all pending operations finished

execution using a semaphore initialized with the number of pending operations. Once an operation finishes execution, it adds its update the token queue and decreases the counter of the semaphore. When the token thread is awakened again it releases the token and returns. This implementation has the additional benefit of speeding up the execution of global operations as they are handled concurrently by multiple threads. Since global operations are executed concurrently, it is important that the order in which the transaction updates are added to the token correspond to the order in which they were executed by the database.

Tracing The Sequential Order of Global Operations. Next, we show how we ensure that the execution order of global operations matches their order in the token sequence of updates. Gyro assumes that the DBMS provides serializability, so it executes transactions in a total order. Gyro must record the serial execution order of the database to make sure that the state updates are broadcast consistently with this order. To this end, the wrapper operation objects uses a reference queue U to the token to capture the order of state updates.

In our implementation we assume that transactions ensure serializability using pessimistic locking: before a transaction accesses a data item i, the transaction acquires a lock and releases it only after the transaction is committed or aborted. When the application requires a transaction t for operation o to commit, Gyro intercepts this call, appends to U the state update u_o produced by o, and then invokes the commit. Since the DBMS uses pessimistic locking, Gyro knows that t has already taken locks on all the data items it accesses before it invokes the commit. Therefore, any concurrent transaction t' for an operation o' that has a conflict with t will not be able to invoke commit until t has committed and released its locks. The thread executing t' will thus append o' to U only after t has finished appending o, so the order of the operations in U is consistent with the execution order of t and t'. Updates that do not conflict can be added to U in any order: Gyro uses a concurrent queue implementation to allow safe concurrent updates from multiple threads.

2.7 CASE STUDIES

We present two widely used benchmarks as case studies: TPC-W, an online store system [Tpc], and RUBiS, an auction website [Rub]. Both benchmarks are implemented in the Java programming language as Java servlets running inside an Apache Tomcat container.

TPC-W. TPC-W [Tpc] is an online bookstore. It handles 14 different client requests such as browsing through books, creating users, adding books to shopping carts or ordering book. The application keeps a persistent state in a database of 10 tables. A `SHOPPING_CARTS` table to store the shopping carts for every user or a `ITEMS` table for the

Application	Transaction classification				Total
	L	G	L/G	Read-only	
TPC-W	15	5	–	13	20
RUBiS	14	4	8	17	26

Table 1: Request classification for the case studies.

available books, among others. On average a client request invokes between 2 and 3 operations. In total, there are 20 transactions of which 13 are read-only. The rest of the operations either update, delete or insert records in, possibly multiple, tables. In TPC-W, operation Partitioning could identify 15 local and 5 global (see Table 1). The local transactions mainly involve updating customer data, and are partitioned by customer id, or manipulations of the shopping carts, and are partitioned by cart id. Gyro allows generating server-specific unique ids, which guarantee that clients requests partitioned by a given id can be served by the that generated that id. This is important in WAN settings. Global transactions involve ordering books or administrative operations such as updating the books list.

RUBiS. RUBiS [Rub] is an online auction web application modeled after eBay [Eba]. RUBiS defines 20 client requests, such as putting items for sale, viewing personal profiles, bidding or browsing items. The persistent state of the application is stored in 8 tables database. For example, the `BIDS` table stores the currents bids and the `USERS` the registered bidders. Similar to TPC-W , the requests handlers are not atomic and consist of invocations of multiple operations. There are 26 transactions in total of which 17 are read-only. In RUBiS, operation Partitioning uses a double-key scheme, whereby many operations are partitioned by both user id and item id. If both parameters route to the same server, the operation is considered local, otherwise it is considered global. Such partitioning scheme yields 14 local, 4 global, and 8 local/global transactions. The local transactions involve the user browsing through his personal profile. Global operations include a global search for items based on some criteria or browsing through a user's own bought items. Local/global operations involve bidding, buying and selling.

2.8 EXPERIMENTS AND EVALUATION

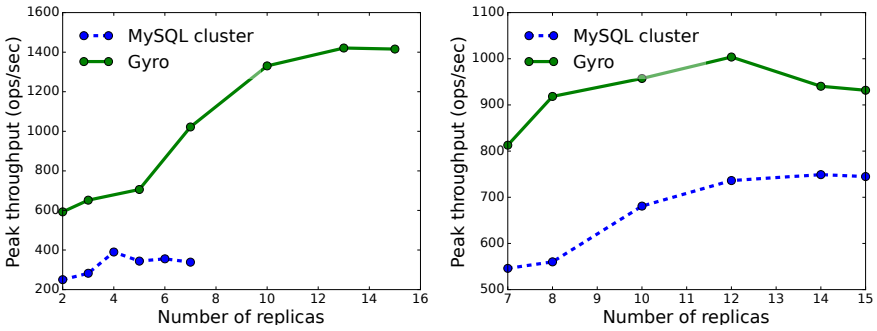
To evaluate our approach we design three set of experiments to answer the following research questions:

RQ 1 How does Conveyor Belt (Gyro) compare to a traditional database that scales out using data partitioning and distributed transactions?

RQ 2 How does Gyro scale out in a geographically distributed setting?

Locations	G	J	US	B	A
Germany (G)	X	253	92	193	314
Japan (J)		X	153	282	188
United States (US)			X	145	229
Brazil (B)				X	322

Table 2: Inter-site latencies among servers in the WAN setting (ms).



(a) TPC-W. (b) RUBiS.

Figure 6: Scalability of Gyro and MySQL Cluster in a LAN setup.

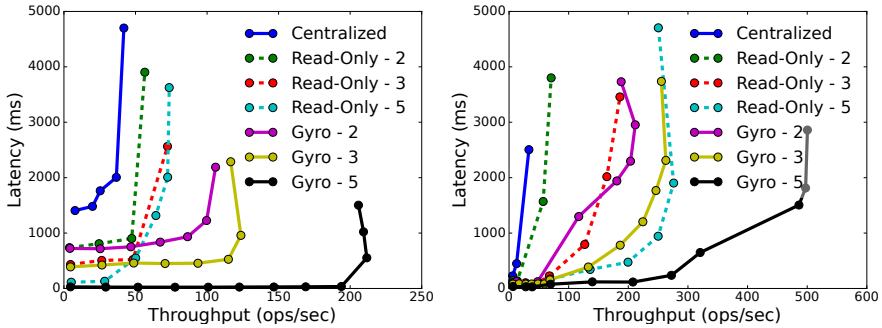
RQ 3 What is the minimum fraction of local operations that is sufficient to see performance improvements with Gyro?

Experimental Setup. We run our experiments on Amazon EC2 T2 Medium instances (nodes). Each node has 4 GBGB of RAM, two virtual cores and is equipped with an Amazon EBS standard SSD with a maximal bandwidth 10 000 IOPS. The nodes run Ubuntu Server 14.04 LTS 64, MySQL 5.5.49-0 and Apache Tomcat 7.0.52.

In the LAN experiments, all servers are located in the same site (datacenter) in Germany. For the WAN (geographically distributed) experiments, we place servers in five different sites to simulate a geographically distributed system. The sites are in Germany (G), Japan (J), US east (US), Brazil (B), and Australia (A). We add these locations in the aforementioned order. For example, a configuration with three locations consists of servers in G, J, and US. Table 2 reports the inter-site latencies among servers.

We used separate client nodes, which have identical configuration as the servers and are located in the same sites. In the WAN setting, we use five client nodes in every configuration, one for each location, and direct requests to the closest server. We equally distribute client threads across client nodes.

Benchmarks. We use TPC-W and RUBiS to evaluate Gyro. Both come with multiple workload mixes. We use a bidding mix with 15 % write operations for RUBiS and the shopping mix with 30 % write operations for TPC-W. Both workloads exhibit a considerable number of local operations that can be leveraged by Gyro.



(a) TPC-W. (b) RUBiS.

Figure 7: Gyro vs. baselines in a WAN (geographically distributed) setup.

2.8.1 *RQ 1: Data Partitioning Comparison*

In this experiment we compare the performance of Gyro, against an approach based on data partitioning and distributed transactions. MySQL Cluster is a version of the popular MySQL DBMS extended with data partitioning capabilities. It horizontally partitions the database and assigns a partition to each server. It uses distributed transactions, with pessimistic locking and two-phase commit, for operations that span multiple partitions. We choose MySQL Cluster as a baseline because it is a prototypical system combining data partitioning and distributed transactions, and because it is often used as reference for comparison by other state of the art work on distributed transactions like Callas [XSL+15].

It is important to note that MySQL Cluster can only provide the *read committed* isolation level, whereas Gyro provides *serializability*, which is significantly stronger and more expensive to achieve. Nonetheless, Gyro is still able to achieve a large speedup over MySQL Cluster.

For both benchmarks, we carefully partitioned the database manually using MySQL Cluster. After running the Operation Partitioning algorithm, we extracted the resulting data partitioning scheme and applied it to MySQL Cluster. That is, we use the same data partitioning that result from the operation Partitioning we apply to the benchmarks. For instance, in TPC-W we partition according to customer and cart ids.

We setup each node to serve as MySQL Cluster server and a data node that stores exactly one data partition. We additionally designate one node as the manager for the initial setup. We use a LAN setting, which is more favorable for MySQL Cluster as distributed transactions are known to perform much better over LANs than over WANs.

We examine the scalability of both approaches. In this local setting, we intensify the workload by increasing the number of clients. In Figure 6 we show how the peak throughput develops while varying the number of servers in the system for TPC-W and RUBiS. Peak

throughput is defined as the maximum throughput a system can sustain while ensuring an average latency of less than 2000 ms.

Figures 6a and 6b show the same trend for both TPC-W and Rubis: as the number of servers grows, the increased cost of distributed coordination eventually outweighs the gain of additional resources to run transactions that require no coordination. This upper bound in scalabilty represents the inherent cost of achieving strong consistency in the workloads we consider, which are not perfectly partitionable. Having said that, both figures 6a and 6b show that Gyro scales much better than MySQL Cluster. In the case of TPC-W we can see that while the performance of MySQL Cluster starts to degrade with configurations of more than 4 nodes, Gyro continues to deliver at a much higher throughput until it reaches a configuration of 13 servers. On the other hand, with the RUBiS workload, Gyro and MySQL Cluster reach a point of saturation at the same configuration, namely 12 servers, but still consistently achieves higher throughput. Overall, Gyro outperforms MySQL Cluster both in terms of maximal throughput and latency by up to 58.6x for latency and about 4.2x for throughput in the case of TPC-W. For RUBiS, Gyro achieves a 1.4 maximal throughput speedup and reduces the latency up to 35.7x.

Gyro performs significantly better than MySQL Cluster due to the distributed transactions used by the latter to lock rows. The necessary coordination with remote machines in MySQL Cluster prevents the progress of concurrent conflicting transactions that access the same rows. In contrast, Gyro does not lock rows. When a server receives global operations that require remote coordination, Gyro merely enqueues the operations until the server gets the token. This allows other concurrent local operations to make progress.

TPC-W and RUBiS show different results due to different read-only operation ratios. In TPC-W many of the local operations are write operations that, in MySQL Cluster, involve distributed transactions. Therefore, TPC-W benefits tremendously from operation partitioning. The RUBiS's workload contains more local operations, but a much larger fraction is read-only. RUBiS thus profits from the read-only transaction optimizations implemented by MySQL Cluster. These results highlight that existing DBMSs already require minimal coordination for read-dominated workloads. The more a workload is write-heavy, and thus hard to scale out, the more using Gyro pays off.

2.8.2 *RQ 2: Scaling Out in WANs*

The previous experiments showed the scale-out capabilities of Gyro in a LAN setting. We now evaluate Gyro in a WAN (i.e., geographically distributed) setting, where coordination is even more expensive and scalability is more challenging. We use two baselines: (1) a standard MySQL (without Gyro) a single server **(centralized)**, and (2) an im-

Configuration	TPC-W	RUBiS
Centralized	1390	416
Gyro– 2	671 (2.1x)	182 (3.3x)
Gyro– 3	436 (3.2x)	155 (2.7x)
Gyro– 5	29 (47.9x)	35 (11.9x)
Read-Only – 2	902 (1.5x)	145 (2.9x)
Read-Only – 3	521 (2.7x)	131 (3.2x)
Read-Only – 5	129 (10.8x)	96 (4.3x)

Table 3: Request latency in milliseconds with light load in a WAN setting. The reported improvements in brackets are relative to the centralized case.

plementation where read-only operations are executed by one server without coordination, like local operations. This is a common optimization offered by many systems **(read-only)**. All these variants guarantee serializability, so the applications have the impression of interacting with a single server and don't need to be modified to account for inconsistencies.

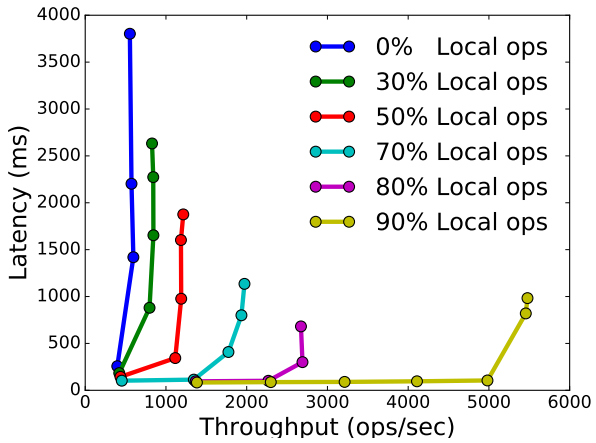
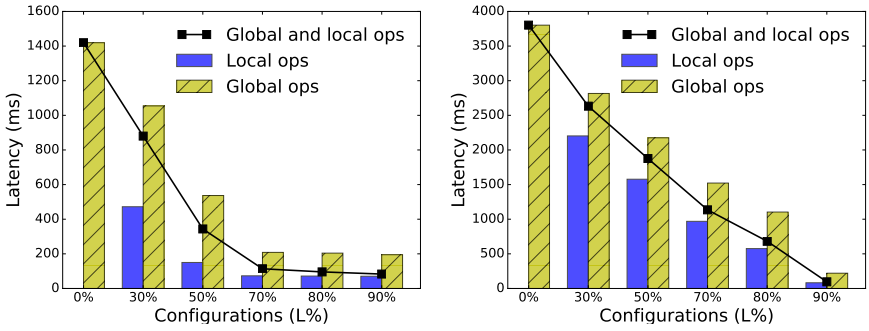


Figure 8: Gyro with different local operation ratios.

First, we compare the latency of Gyro in different configurations when the system is not overloaded. In Table 3, we report the latency improvement over the centralized setting of each configuration, from two to five with TPC-W and RUBiS.

Gyro achieves significant latency reduction, of more than one order of magnitude, because it reduces the need for coordination. For instance, RUBiS with 3 servers the latency is 3.2x less that of a centralized server and 2.7 for the read-only baseline. The performance is best when a server datacenter is available in every geographical location of the clients. In fact, for the 5 server configuration the latency is 47.9x less for TPC-W and 11.9x for RUBiS. In contrast, the latency when using the read-only optimizations is only 10.8x less for TPC-W and 4.3x for RUBiS. This is because the majority of operations can be



(a) Light load.

(b) Heavy load.

Figure 9: Latency comparison of Gyro with different local operation ratios using micro-benchmarks.

served by the local server where clients are located. This is especially the case for the five servers case where the latency is at its lowest: 29 ms for RUBiS and 35 ms for TPC-W.

Next, we shift our attention to both throughput and latency with more intense workloads (Figure 7). We stress the system by increasing the number of clients until the latency reaches 5 seconds. The single server in the centralized case start to saturate quickly, at few tens of operations per second. Read-only optimization significantly reduces latency and increases throughput for both workloads and especially for RUBiS which is more read-dominated. Gyro, however, has a much larger impact as it allows the local execution of many more operations, both read-only and not. The effect in terms of throughput over both the centralized and read-only baselines is substantial. Gyro enables multiple sites to execute operations in parallel and results in much higher maximum throughput.

Overall, Gyro improves the maximum throughput compared to the read-only setting. For instance, in the five servers configuration there is an increase of the maximal throughput by 291 % for TPC-W and 181 % for RUBiS. In terms of scalability, Figures 7a and 7b show that Gyro scales very well until at least five geo-locations, which is a fairly high number in many practical settings. By contrast, the read-only baseline maxes out already with three servers, especially with TPC-W where the gain from using additional servers in terms of throughput is marginal. Like in the LAN case, if we keep adding locations, we expect scalability to asymptotically stop because of the increasing coordination cost induced by executing global operations.

2.8.3 *RQ 3: Micro-Benchmarks*

We now examine the performance of Gyro more in detail. We analyze the effect of different local operations ratios on Gyro's performance using a synthetic workload where we can precisely specify these ratios.

The execution time of operations (global or local) is fixed to 5 ms. We use a WAN setup with three servers and vary the percentage of local operations in the workload from 0 % to 90 %.

Figure 8 confirms that the performance of Gyro is highly sensitive to the fraction of local operations in the workload. For instance, with a workload of 30 % of local operations the system starts to saturate already around 600 ops/s while in a workload of 90 % local operations the saturation starts only around the 5477 ops/s. This can be explained by the additional coordination overhead of global operations unlike local operations which can be served by the nearest server.

Figure 9a shows mean latencies for local and global operations with a light load (far from saturation). The average latency of all operations decreases as we add more local operations to the mix and he have less global operations queuing up. As expected, in all configurations the latency of local operations is much lower and is between 2.23x and 3.75x less compared to that of global operations. For instance, in a configuration with 70 % of local operations, the mean latency is 195 ms for local operations and 70 ms for global operations (2.78x less). The overall latency stabilizes with 70 % local operations or more. By contrast, in a configuration with a higher load (Figure 9b), the overall latency continues to fall even after the 70 % threshold observed in Figure 9a. The reason is that the saturation of the system does not only occur because of the large fraction of global operations queuing up but also because of the overall volume of requests.

2.9 RELATED WORK

Scaling out client-server applications is an important topic and it has been the subject of a large volume of work. We now review it and position the Operation Partitioning approach in this landscape. For space reasons, we do not review the literature on fault-tolerant replication algorithms since fault tolerance can be treated as an orthogonal issue to distributed transactions. We leave combining the two problems, along the lines of work like [ZSS+15], as future work.

Data Partitioning. The problem of finding an optimal database design is NP-hard [MBS88]. Nonetheless, a large number of heuristics for data partitioning have been proposed, such as [CJZ+10; PCZ12; QKD13; TNS+14]. These techniques require substantial offline effort, including running a representative workload, collecting samples, defining an accurate cost model of the system performance, and sometimes user guidance in identifying the best solution. Operation Partitioning indirectly obtains a partial data partitioning scheme, much like existing work, but it is entirely automated and based on static analysis.

Distributed Transactions. The typical approach to implement distributed transactions, which is used in many practical database management systems, is to lock the rows accessed by the transaction

and to use two phase commit to conclude the transaction. Since this approach is expensive, there has been much work on speeding up distributed transactions. Spanner speeds up read-only transactions through the use of synchronized clocks [CDE+13]. H-Store speeds up ACID transactions that access only a single partition. It supports multi-partition transactions using standard locking and two-phase commit protocols. Our evaluation shows that the Conveyor Belt is superior to a standard two phase commit transnational system with locks. ElasTraS [DEAA09], G-Store [DAEA10], and MegaStore [BBC+11] only support ACID transactions within the boundary of a single partition or key group, and do not offer full transactional support like Gyro.

Several approaches like Calvin [TDW+12], Lynx [ZPZ+13], Rococo [MCZ+14], Callas [XSL+15] and others [FA15; SLS+95; SCD+17], have been proposed to improve the performance of distributed transactions, but they typically require implementing a novel database management or data store system, unlike the Conveyor Belt protocol which is a middleware running on top of an unmodified black-box, single-server DBMS offering a JDBC interface. In addition, they require additional knowledge about the semantics of the application that must be provided by the user, sometimes by restructuring the code. Modifying and extending the application code in this sense can be complex and cumbersome, and sometimes unfeasible. The Conveyor Belt protocol does not have this requirement, since Operation Partitioning applies to unmodified application code. Yet, Conveyor Belt provides competitive performance speedups. While the Callas algorithm supports serializability, the actual Callas prototype system only provides the *read committed* isolation level, just like the MySQL Cluster system it is based upon. Gyro provides serializability instead, which is significantly more expensive than read committed isolation. Nonetheless, Gyro achieves similar speedups over MySQL Cluster as the Callas results reported in [XSL+15].

SDD-1 [BRG+78] is related to our approach in that it uses *transaction classes*, but still differs in several aspects. First, a key pre-step to achieve good performance in SDD-1 is that the user provides a good grouping of transaction into classes, but SDD-1 offers no support for it. In our approach, we automatically generate operation partitions that can be leveraged by our protocol based on static analysis. Second, SDD-1 replicas executing global operations need blocking coordination based on timestamps. This algorithm was pioneering work on distributed transactions, but is less efficient than algorithms based on distributed locks [Ber17], which we compare against.

Weakly Consistent Scale-Out Approaches. Most algorithms using replication to scale out offer only weak consistency guarantees: eventual consistency [PST+96; DHJ+07] session consistency [TDP+94], causal consistency [LFK+11], timeline consistency [CRS+08], and Parallel Snapshot Isolation [SPA+11].

Recent work proposes strengthening weak consistency with *invariants*, like in the Red/Blue model [LPC+12], the Explicit Consistency model [BDF+15], and Invariance Confluence [BFF+14]. Requiring developers to define good invariants is challenging. Also, even with invariants, the system will still show a weakly-consistent behavior that would not occur in a sequential execution. Unlike these approaches, Operation Partitioning support serializability [Pap79], as required by ACID applications.

Treaties. Prior work on treaties combines scale-out replication and strong consistency for subset of operations. Informally, treaties allow replicas to agree to split the value of a certain field and to share the splits. For example, in a ticket sale application, replicas can agree on a treaty where each take a share of the available tickets, so that they do not need to coordinate every time they sell a ticket unless they sell out their share. Treaties make specific assumptions on the applications they target: concurrent transactions must make small commutative modifications to a shared global quantity at different replicas, and their outcome must not be sensitive to such small modifications. Examples of treaties are the escrow protocol [O'N86], the demarcation protocol [BGM92], Homeostasis [RKB+15], and time-limited warranties [LMA+14]. Work related to the idea of treaties has also investigated relaxed notions of consistency such as bounded inconsistency [YV00] or consistency rationing [KHA+09]. Operation Partitioning is more generic as it does not make assumptions on the application, as treaties do. Operation Partitioning can be applied to any application, whereas treaties require either user knowledge about the application semantics or the use of special languages, like in Homeostasis.

2.10 CONCLUSION

We introduced Operation Partitioning, a technique that allows scaling out applications while preserving serializability. We implement our technique in a middleware, called Gyro that can be used with an unmodified DBMS. Our experiments with two user application TPC-W and RUBiS show that Gyro is very effective in both LAN and WAN settings.

3

EFFICIENT VERIFICATION OF DISTRIBUTED PROTOCOLS USING STATEFUL MODEL CHECKING

The previous chapter treated the availability aspect of dependabilty. In the rest of the thesis, we will focus on the safety aspect employing either model checking techniques or error propagation analaysis as measures for fault removal. The results in this chapter have been published in [SBM+13].

A major hurdle for applying model checking techniques directly to program implementations is the resulting complexity and large size of their state space. In this chapter, we propose decomposition-based explicit model checking as a means to handle this complexity by pruning portions of the state space that are irrelevant to the checked specification. The chapter is organized as follow: In Section 3.1 we provide an overview of contributions made in this chapter. We then provide a motivating example in Section 3.2 to show the potential of decomposition-based model checking. Section 3.3 decribes the theoretical contribution of the chapter and provides a correctness proof of the proposed algorithm. In Sections 3.4 and 3.5 we describe our implementation and evaluate it using distributed message-passing protocols. We provide a discussion on related work in Section 3.6 and conclude the chapter in Section 3.7.

3.1 OVERVIEW

Software model checking (MC) [God97; GKS05] is a practical branch of verification for checking the actual implementation of the system. The wide usability comes at the price of low scalability as the model checking of even simple single-process programs can take several hours (or go off-scale) using state-of-the-art techniques [KKB+12].

Verification complexity gets even worse for concurrent programs that run on loosely coupled processes. Our focus is on distributed protocols for various mission-critical (fault-tolerant) applications where rigorous verification is desired. Example applications include atomic broadcast [JRS11], storage [GGL03], diagnosis [SBS+11b], etc. Although the verification of fault-tolerant distributed systems is known to be a hard problem due to concurrency and faults, MC has proven to be useful for debugging and verifying small instances of deployed

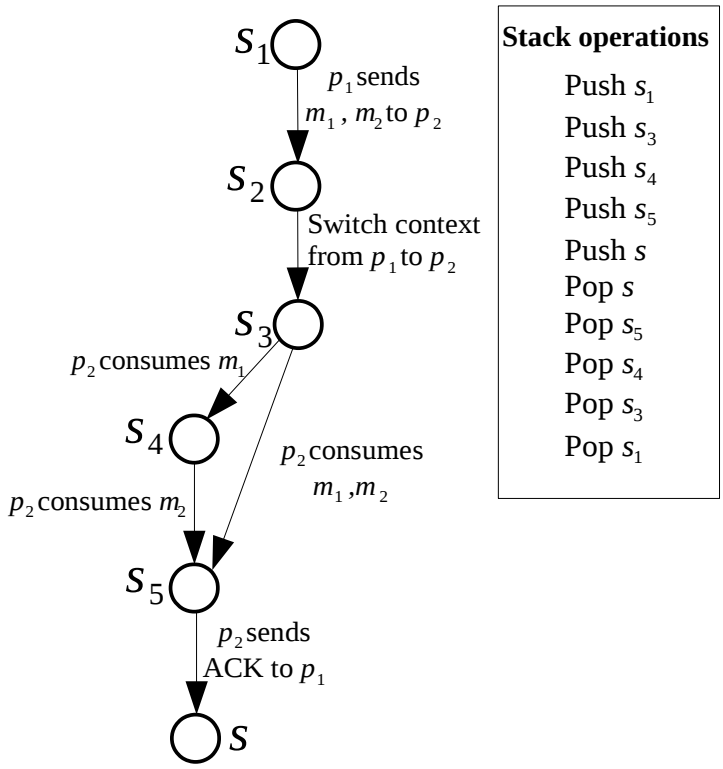
protocols; recent approaches include MaceMC [KAJ+07], CrystalBall [YKK+09], Modist [YCW+09; GWZ+11], Basset [LDM+09] and its extensions/optimizations [BKS+11; BSS+09; MSB+11].

In MC, the possible executions of a system are modeled in terms of a state graph, where states (i.e., nodes) can be thought of as snapshots of the entire system (e.g., state of the servers, clients, communication channels) and transitions (i.e., edges) model any event that may alter the system's state (e.g., lines of code, function blocks). For MC to be scalable, the size of the graph must be feasible to manage, a challenge that is often referred to as the *state explosion* problem. An efficient and simple approach is *stateful depth-first search* [CJGK+18], where the state graph is abstracted by 1) a sequence of states (called stack) that corresponds to the last run of the system, and 2) a set of states that have been explored during the model checking (called visited states).

In this chapter, we propose a general and sound approach to reduce the size of both the stack and the visited states for improved scalability of MC. Key to the proposed reduction is the concept of *decomposition* that we observe to be present in the implementation of real systems. For example, implementations of distributed systems are typically decomposed into different aspects or execution modes (i.e., runnable configurations of the system under verification) of the system such as synchronization, GUI, automatic execution, or logging. Despite the richness of implementations, the specifications subject to model checking very often consider only a subset of all these aspects. Roughly speaking, our reduction approach consists in utilizing decomposition so that only selected aspects are model checked against the specification without having to modify the implementation.

Our Theoretical Contributions. We propose a formal framework that characterizes decomposition by distinguishing between *relevant* and *auxiliary* state information. The decomposition is always with respect to a subset of all transitions of the system corresponding to the execution mode of interest. We show a use of this characterization for more scalable stateful depth-search, called *decomposition-based stateful search*, and prove the soundness of the proposed approach. The input of the framework (beside the specification of the system) is a sound decomposition. Although showing the soundness of a decomposition can be as hard as model checking itself, we argue that this can be done using suitable static analysis and we justify this claim by showing an implementation for general distributed systems implemented in Java.

Our Prototype Implementation. We implement decomposition-based stateful model checking within Basset [LDM+09; BKS+11], an explicit-state model checker for general message-passing Java programs. We then apply our decomposition framework to optimize the verification of distributed message-passing algorithm. In this decomposition, the auxiliary part of the state stores the latest messages delivered by a process. This is utilized only for debugging, that is, to analyze runs



(b)

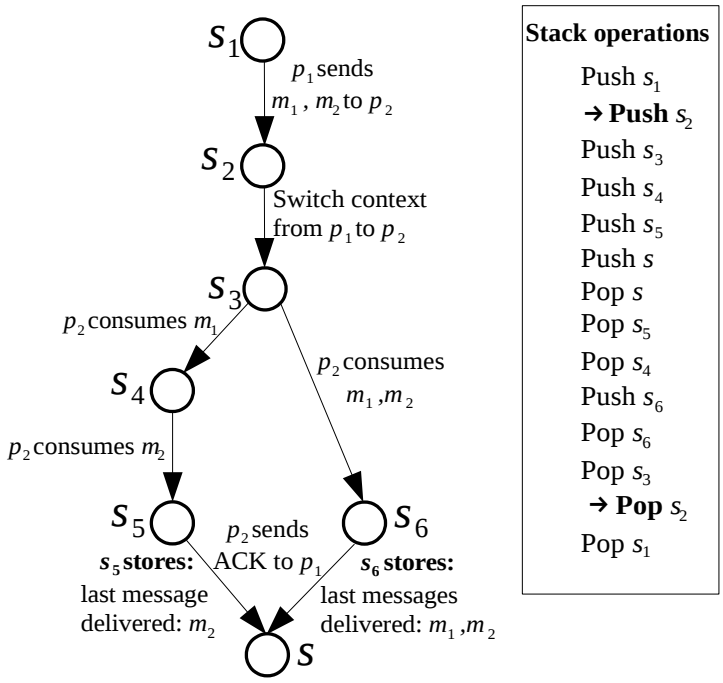
Figure 10: Naive depth-first search (DFS) example.

where the desired properties of the system are violated This execution mode is irrelevant, say, for the fault-tolerance aspects of the system and the corresponding state information can be safely decomposed as auxiliary.

Our Evaluation. We use our prototype implementation to evaluate the proposed decomposition-based stateful search with various fault-tolerant message-passing protocols such as Paxos consensus [Lam98], Zookeeper atomic broadcast [JRS11], and distributed storage [ABND95]. The decomposition-based stateful optimization improves on the naive stateful search both in terms of search time and memory by up to 69 %. We also compare decomposition-based stateful search with partial-order reduction [GVLH+96], an optimization known to be efficient for fault-tolerant message-passing protocols [BKS+11; BSS+09]. Our experiments show that the two optimizations, when used together, result in enhanced reductions achieving an improvement of 39 % compared to settings with partial-order reduction only.

3.2 MOTIVATING EXAMPLE

We give the intuition of the proposed reduction approach through a simple message-passing example with two processes, p_1 and p_2. Process p_1 sends two messages m_1 and m_2 to process p_2. Process p_2



(a)

Figure 11: Decomposition-based stateful search (DBSS) example.

stores in its local state the messages it receives. It is possible for m_2 to arrive later than m_1 at p_2 due to network delays and p_2 can process available messages (m_1 and m_2) in one atomic step. After receiving m_1 and m_2, p_2 sends an acknowledgment message to p_1.

Figure 10 shows the state graph of the this example system as explored by a naive DFS and the corresponding operations of the search stack. Note that s_5 and s_6 are different states because they store different messages histories.

Decomposition. Suppose that the message history information is not subject to the verification. As a result, this part of the state is labeled as auxiliary. This decomposition is sound with respect to the transitions shown in Figure 10 because these transitions only depend on the non-auxiliary (i.e., relevant) part of the state. Note that although the system may contain additional transitions, in Figure 10 only those transitions are depicted that are relevant in target execution mode.

Selective Hashing. We propose a reduction framework for more scalable stateful depth-first search by making use of the decomposition of a system. Firstly, we introduce *selective hashing*, which modifies the naive search in that it only stores the relevant state in the set of visited states. In our example, the state graph resulting from selective hashing is shown in Figure 11. Note that states s_5 and s_6 collapse into the same state because they only differ with respect to their message histories. The gain of selective hashing is that it directly reduces the size of the state graph that is explored by the model checker.

Selective Push-on-Stack. Secondly, we introduce *selective push-on-stack*, which is based on the observation that the transition system may have *single enabled* transitions, i.e., transitions that are exclusively enabled in a state. Since single enabled transitions are non-concurrent with any other transitions, states where these transitions are executed do not have to be used for backtracking, although they have to be remembered as visited states. Therefore, states with single enabled transitions do not have to be pushed onto the search stack. Consider, for instance, the single enabled transition t from s_2 to s_3 in Figure 11. Since t is the only transition that can be executed in s_2, no state remains unvisited if s_2 is not backtracked by the search. The application of selective push-on-stack to our example single-enabled transition leads us to the search stack in Figure 11, where s_2 is not involved in any stack operation. The information of visiting s_2 is stored in a different stack, which is returned when a counterexample is found. Note that selective push-on-stack visits the same states as the naive search but in shorter time with fewer stack operations.

3.3 GENERAL REDUCTION FRAMEWORK

Our general model for decomposition is presented in Section 3.3.1. The proposed verification approach and its properties are explained in Section 3.3.2 and Section 3.3.3, respectively.

3.3.1 *System Model*

We adopt a general and abstract model of programs [AW04; BSS10]. The program maintains a global state and can execute transitions (e.g., line of codes) to reach other states. Formally, a program is represented as a transition system $TS = (S, S_0, T)$ where:

- S is a finite set of possible states of the program.
- $S_0 \subseteq S$ is a set of initial states. For simplicity, we assume that there is a single initial state $s_I \in S_0$.
- $T = \{t \mid t \subseteq S \times S\}$ is a finite set of transitions.

A transition $t \in T$ is *enabled in* state $s \in S$ and we write $t \in \mathit{enabled}(s)$, if there is $s' \in S$ such that $(s, s') \in t$. Consequently, we define $\mathit{enabled}(s)$ as a set of transitions enabled in s. To simplify the discussion, we assume that transitions are deterministic, i.e given a state $s \in S$ and a transition $t \in T$ enabled in s, there is a single state $s' \in S$ such that $(s, s') \in t$. A *path* of the model is defined as a finite sequence $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n$, where $s_1, \ldots, s_n \in S$, $t_1, \ldots, t_{n-1} \in T$ and for all $1 \leq i < n$ it holds that $(s_i, s_{i+1}) \in t_i$. For convenience, we write $s_1 \xrightarrow{t_1 \ldots t_{n-1}} s_n$. A state s is called *reachable* if there exists a path $s_I \xrightarrow{t_1 \ldots t_n} s$.

Decomposition. Let $TS = (S, S_0, T)$ be a transition system, where T may be a subset of all transitions of the system S_{rel} and an "auxiliary" part $s_{aux} \in S_{aux}$ such that $s = (s_{rel}, s_{aux})$. Whether a state fragment can be marked as relevant depends on the system and the property being checked. This should be done by a domain expert.

To formalize our approach, we introduce the function $h : S \to S_{rel}$, to extract the relevant part of states such that for a state $s = (s_{rel}, s_{aux}) \in S$ we have $h(s) = s_{rel}$. Given two states $s, s' \in S$, a transition $t \in T$ with $(s, s') \in t$ is said to be *single enabled* in s iff t is the only transition enabled in s. Intuitively, a single enabled transition is non-concurrent with any other transition.

Intuitively, we require that, given two states s and $s' \in S$ with the same relevant part, a transition t is enabled in s only if it is enabled in s'. Moreover, the execution of t in s and s' results in two states s_1 and s'_1 with the same relevant part. Formally, we characterize a sound decomposition as:

Definition 1 (Decomposition)**.** *Given a transition system $TS = (S, S_0, T)$ and a set S_{rel} of states, we say that TS can be decomposed along S_{rel} if:*

- (State decomposition) $S \subseteq S_{rel} \times S_{aux}$.
- (Transition decomposition (1)) $\forall s, s' \in S, t \in T : h(s) = h(s') \Rightarrow (t \in enabled(s) \Leftrightarrow t \in enabled(s'))$.
- (Transition decomposition (2)) $\forall s, s', s_1 \in S, t \in T : (h(s) = h(s') \wedge s \xrightarrow{t} s_1) \Rightarrow (\forall s'_1 \in S : s' \xrightarrow{t} s'_1 \Rightarrow h(s_1) = h(s'_1))$.

3.3.2 *Decomposition-based Stateful MC*

Preserved Specifications. We assume that the specification of the system is given in form of state properties. As state properties refer to the "global" state of the system, they constitute a key class of properties of various distributed systems [YKK+09; YCW+09; MSB+11; GWZ+11]. Formally, given a transition system $TS = (S, S_0, T)$, we define a *state property* f as $S \to \{true, false\}$. The state property *holds* for a transition system if it returns true for every reachable state. Let $TS = (S, S_0, T)$ be a transition system that can be decomposed along a set of states S_{rel} and $f : S \cup S_{rel} \to \{true, false\}$ a state property. We say that f is *decomposed* if for all reachable $s \in S$ it holds that $f(s) = f(h(s))$. Intuitively, f depends solely on the relevant part of a state.

The Algorithm. Algorithm 3 shows the pseudo-code of the proposed decomposition-based stateful search (DBSS). The call next($enabled(s)$) non-deterministically returns one transition from $enabled(s)$. The calls *pop* and *push* respectively correspond to the usual stack operations of removing and adding an element to the stack. Calling *peek* returns the top-most element of the stack without removing it. In principle, the

Algorithm 3: Decomposition-based stateful search (DBSS) algorithm for transition system TS and state property f.

```
1  function DBSS(TS, f)
3      Stack stack ← ∅
5      Set reached ← ∅
7      Stack CE-stack ← ∅
9      State s ← s_I
11     stack.push(s)
13     CE-stack.push(h(s))
15     while stack ≠ ∅ do
17         while enabled(s) ≠ ∅ do
19             Transition t ← next(enabled(s))
21             enabled(s) ← enabled(s) \ {t}
23             State s' ←t s
25             //selective hashing
26             if h(s') ∉ reached then
27                 reached ← reached ∪ {h(s')}
29                 CE-stack.push(h(s'))
31                 //selective push-on-stack
33                 if next(enabled(s')) is not single enabled s' then
34                     stack.push(s')
36                 s ← s'
38             if ¬f(s) then
40                 return s, CE-stack
42         s ← stack.pop()
44         while h(s) ≠ CE-stack.peek() do
46             CE-stack.pop()
48     return true
```

DBSS algorithm modifies the naive depth-first search (DFS) algorithm. The algorithm uses a *stack* to remember the explored paths. The stack is also used for backtracking in case of branching. To avoid redundancy by visiting a state more than once, the set *reached* stores the visited states. The use of *reached* constitutes the fundamental optimization of the stateful search. The outer loop at line 15 assures that every visited state is checked for branching. The loop at line 17 guarantees that every enabled transition is explored.

The first modification to DFS is *selective hashing* in lines 26 and 27. Instead of remembering in *reached* a new visited state s, the algorithm remembers only the relevant state part (27). Two states with the same relevant parts are considered to be equivalent (26). Note that, in contrast to the *reached* set, the entire state is pushed on the stack. This corresponds to allowing the verification of unmodified implementations, where transitions can be meaningfully executed only in full-fledged states containing both the relevant and auxiliary parts. The second modification is *selective push-on-stack* in line 32. Instead of pushing every newly visited state s' onto the search stack, s' is only pushed if the transition enabled in s' is *not* a single enabled transition. Since the single enabled transition is the only enabled transition in

s', no branches that possibly lead to new reachable states are missed. We add another stack to the algorithm, *CE-stack*, to keep track of the relevant part of the states that compose the explored path. *CE-stack* also serves the purpose of keeping track of all executed transitions including those skipped from backtracking because of selective push-on-stack. If a bug is found, that is the condition in line 38 holds, we return the reached state s and *CE-stack* as a counterexample path leading to the state violating the property. Otherwise, the state property holds and *true* is returned.

Liveness Model Checking. Using the notations from the definition above, as DBSS explores a subset of TS' which behaves like TS, the algorithm can be modified to *generate* TS. Doing so, standard algorithms can be used to check liveness properties (e.g. written in temporal logics [CJGK+18]) that cannot otherwise be expressed through state properties.

Analysis. By exploiting decomposition, the DBSS algorithm explores only a "relevant state graph". In other words, given a transition system TS' that can be decomposed along S, the transition system explored by DBSS simulates another transition system $TS = (S, S_0, T)$ which is subsumed by TS'. Intuitively, given some state property f, $DBSS(TS', f)$ explores a transition system that behaves like TS. As TS is subsumed by TS' (and is thus a smaller transition system), DBSS improves time and memory efficiency over DFS of TS'. This analytic claim will be substantiated by our experiments in Section 3.5. Formally, we can define a subsumption relation between the two transition systems TS' and TS:

Definition 2 (Subsumption)**.** *Given two transition systems* $TS = (S, S_0, T)$ *and* $TS' = (S', S'_0, T')$*, we say that* TS' *subsumes* TS *and write* $TS \subseteq TS'$ *if:*

- TS' *can be decomposed along* S *and*
- $\forall s, s_1 \in S : (\exists t \in T : s \xrightarrow{t} s_1) \Leftrightarrow (\exists t' \in T', \exists s', s'_1 \in S' : s' \xrightarrow{t'} s'_1 \wedge h(s') = s \wedge h(s'_1) = s_1)$.

Note that there is no need of proving that the above subsumption relation indeed applies for TS' and the transition system explored by $DBSS(TS', f)$. The above discussion has been added to better highlight the source of reduction of DBSS.

3.3.3 *Correctness of DBSS*

We now show that the DBSS algorithm can be used for the verification of decomposed state properties without missing bugs and also without falsely concluding the truth of the property. Formally, we prove that the algorithm is sound, complete and terminating.

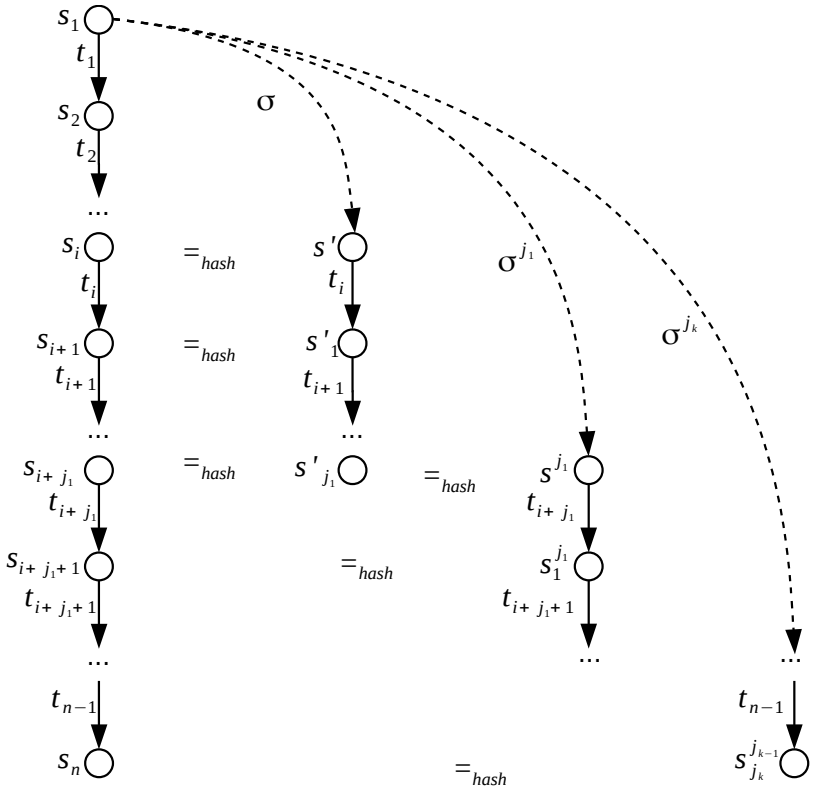


Figure 12: Illustration of proof of Lemma 5.

For convenience, given a transition system TS and a decomposed property f, we may refer to $DBSS(TS, f)$ by writing $DBSS(TS)$. We introduce the following terminology to ease the following discussion. A state s is said to be *explored* and transition is *fired* if there is a state s' such that line 23 of DBSS is executed.

First, we show that assuming that a state s is not yet explored, all transitions in $enabled(s)$ are fired by DBSS.

Lemma 4. *Given a transition system $TS = (S, S_0, T)$, if a state $s' \in S$ is explored in $DBSS(TS)$ and the condition of line 26 holds, then every $t \in enabled(s')$ is fired in s'.*

Proof. Let t be a transition in $enabled(s')$. If t is next$(enabled(s'))$, then t is fired in the next iteration of the while loop in line 17. Otherwise, if $enabled(s') > 1$ (cf. condition of line C3), s' is pushed onto the stack and every transition $t' \neq t \in enabled(s')$ is fired when s' is backtracked in the depth-first search. Note that s' is guaranteed to be backtracked after each transition fired in s' because the state graph is finite. □

We prove that if a state is reachable in TS, there is at least a state explored by $DBSS(TS)$ with the same relevant information. We will use the following notation for convenience: Given a path $\sigma = s_1 \xrightarrow{t_1 \ldots t_{n-1}} s_n$ and $s_n \xrightarrow{t_n \ldots t_{l-1}} s_l$, the path $s_1 \xrightarrow{t_1 \ldots t_{l-1}} s_l$ can be written as $\sigma \xrightarrow{t_n \ldots t_{l-1}} s_l$.

Lemma 5. *Given a transition system $TS = (S, S_0, T)$, $\forall s \in S$, if s is reachable in TS, then there exists a state $s' \in S$ such that $h(s) = h(s')$ and s' is explored by $DBSS(TS)$.*

Proof. The proof is indirect and is illustrated in Figure 12 using the following notation: Given two states $s, s' \in S$, we write $s =_h s'$ if and only if $h(s) = h(s')$. Indirectly, we assume the following: Let $s_n \in S$ be a reachable state in TS so that there is no other state s'' explored by $DBSS(TS)$ with $h(s_n) = h(s'')$. Let $\sigma' = s_1 \xrightarrow{t_1 \ldots t_{n-1}} s_n$ be a path leading to s_n in TS. We know that $n > 1$ because the initial state $s_I = s_1$ is explored by the algorithm. Consequently, there must be $2 \leq i < n$ such that t_i is not fired in s_i. Lemma 4 implies that the condition of line 26 does not hold when s_i is explored. This means that there is a state s' reachable via a path σ explored by $DBSS(TS)$ such that $h(s') = h(s_i)$ and $h(s') \notin$ *reached* when explored. $i < n$ since otherwise s_n would be explored. From Definition 1 (transition decomposition (1)), we also know that $t_i \in$ *enabled*(s_i) and $t_i \in$ *enabled*(s'). Furthermore, t_i is fired in s' (Lemma 4) and it holds that $h(s_{i+1}) = h(s'_1)$ where $s' \xrightarrow{t_i} h(s'_1)$ (transition decomposition (2)). Let $0 < j_1 \leq n - i$ be the highest natural number such that $\sigma \xrightarrow{t_i} s'_1 \xrightarrow{t_{i+1} \ldots t_{j_1 - 1}} s'_{j_1}$ is explored by $DBSS(TS)$. Because of transition decomposition, we know that $h(s'_{j_1}) = h(s_{i+j_1})$. Therefore, $j_1 = n - i$ would imply a contradiction, as this would mean that there is a state s'' such that $h(s_n) = h(s'')$ which is explored. Since j_1 is the highest such index, Lemma 4 implies that the condition of line 26 does not hold when s'_{j_1} is explored. So there must be a state s^{j_1} reachable via a path σ^{j_1} explored by $DBSS(TS)$ such that $h(s^{j_1}) = h(s'_{j_1})$ and $h(s^{j_1}) \notin$ *reached* when explored. Because of transitivity, we know that $h(s^{j_1}) = h(s_{i+j_1})$. Let $0 < j_2 \leq n - i - j_1$ be the highest natural number such that $\sigma^{j_1} \xrightarrow{t_{i+j_1}} s_1^{j_1} \xrightarrow{t_{i+j_1+1} \ldots t_{j_2-1}} s_{j_2}^{j_1}$ is a path explored by $DBSS(TS)$. We know from Lemma 4 and the decomposition definition that such a path exists and that $h(s_{j_2}^{j_1}) = h(s_{i+j_1+j_2})$. Since j_2 is the highest such index, Lemma 4 implies that the condition of line 26 does not hold when $s_{j_2}^{j_1}$ is explored. So there must be a state s^{j_2} reachable via a path σ^{j_2} explored by $DBSS(TS)$ such that $h(s^{j_2}) = h(s_{j_2}^{j_1})$ and $h(s^{j_2}) \notin$ *reached* when explored. We know that $h(s^{j_2}) = h(s_{i+j_1+j_2})$. Continue the construction. Let $0 < j_1, j_2, \ldots, j_k$ be natural numbers such that $i + j_1 + j_2 + \ldots + j_k = n$. The construction ends because $1 \leq k \leq n - i$. Inductively, we have that $h(s_n) = h(s_{j_k}^{j_{k-1}})$. Since $s_{j_k}^{j_{k-1}}$ is explored by $DBSS(TS)$, we have a contradiction. □

Theorem 2 (Soundness)**.** *Given a transition system TS and a decomposed property f, if $DBSS(TS, f)$ returns true, then f holds for TS.*

The algorithm returning *true* is a guarantee that the verified program satisfies the property f.

Proof. Assume that $DBSS(TS, f)$ returns *true*. This means that for every state s explored by $DBSS(TS, f)$, we have $f(s)$. Now we suppose that there exists a reachable state $s' \in S$ such that $\neg f(s')$. From Lemma 5 we know that there exists a state $s'' \in S'$ explored by $DBSS(TS, f)$ such that $h(s'') = h(s')$. Since f is decomposed, we have $\neg f(s'')$ which is a contradiction. □

Theorem 3 (Completeness)**.** *Given two transition systems TS' and TS such that $TS \subseteq TS'$ and a decomposed property f, if $DBSS(TS', f)$ returns s and CE-stack, then:*

- *$\neg f(s)$ and s is reachable in TS' and*
- *CE-stack contains a path from $h(s_I)$ to $h(s)$ in TS where s_I is the initial state of TS'.*

In case a bug violating a property f is found, DBSS returns a state where f does not hold, and a path in the subsumed transition system leading to it. In practice, such a path is sufficient for debugging as the subsumed system contains all relevant state information.

Proof. $DBSS(TS', f)$ returns a state s and *CE-stack* when the condition in line 38 is satisfied. This means that we have $\neg f(s)$. Since s is explored by DBSS it is trivially reachable in TS'.

Now we prove that *CE-stack* contains a path leading to $h(s)$ in TS. Since DBSS is a DFS, the sequence of states in the *CE-stack* when exploring s is a path σ from $h(s_I)$ to $h(s)$. The path exists in TS because of Definition 2. □

Theorem 4 (Termination)**.** $DBSS(TS, f)$ *terminates for any transition system TS and state property f.*

The termination of the program follows directly the assumption that the transition system is finite-state and the algorithm is based on DFS.

Proof. Assuming that the calls in the algorithm terminate, we have to check whether the two loops at line 17 and 15 terminates. The loop at 17 terminate because in each iteration one element is removed from $enabled(s)$ (line 21) and the number of enabled transitions is finite. In line 15, no more states are pushed to the stack once every state has been explored and therefore included in *reached*. Since the number of states is finite and after each iteration one element is removed from the stack (line 42), the loop terminates after a finite number of iterations. Note that the loop at line 44 is guaranteed to terminate since the set of relevant parts of states in *stack* is included in *CE-stack*. □

3.4 IMPLEMENTING DBSS IN JPF/MP-BASSET

In this section, we present a general application and implementation of the conceptual reduction framework described in Section 3.3. The following application instantiates, implements, and evaluate the decomposition-based reduction framework for general message-passing systems written in Java. A direct implication of our results is the enhanced scalability of model checking Java-based implementations of message-passing systems. We also discuss how our implementation can be used for symmetry reduction of replication-based (fault-tolerant) message-passing protocols.

We implement the proposed decomposition-based stateful search (DBSS) within the MP-Basset model checker for message-passing systems [BKS+11]. [1] The source of our implementation of DBSS/MP-Basset can be downloaded under [Mpb]. In the core of MP-Basset, the model checker Java Pathfinder [Jpf] (JPF) implements depth- and breadth first search of multi-threaded Java programs. MP-Basset builds upon JPF's architecture to enable writing and model checking message-passing Java programs. In essence, JPF consists of the *core* search engine and a *model*. Intuitively, the core is responsible for the search, whereas the model constitutes the Java program under verification. The model, in this case MP-Basset, runs in a separate Virtual Machine implemented by JPF. JPF itself is implemented in Java and it runs within the Java Virtual Machine of the host system. Our decomposition based approach extends JPF's core with selective hashing and push-on-stack. As the reduction is based on the decomposition of the system, this information is obtained from the JPF Virtual Machine, which contains all system-specific information. Communication between core and model is done via JPF's Model Java Interface (MJI). First in Section 3.4.1 we will discuss how our decomposition model applies to the MPBasset case study. Section 3.4.2 (respectively, in Section 3.4.3) explains how the *hash* function (and *auxiliary* predicate) is implemented within the architecture of Basset/MP-Basset and JPF.

3.4.1 *Decomposition*

From Definition 2, $TS = (S, S_0, T)$ corresponds to the message-passing program as defined in Basset/MP-Basset's input language (a Java library for message-passing), whereas $TS' = (S \times S_{aux}, S'_0, T')$ is determined by the program executed by the JPF virtual machine. The argument that it is a sound decomposition implicitly follows from the (sound) implementation of the model checkers Basset [LDM+09] and MP-Basset [BKS+11].

1 Our instrumentation would analogously apply for Basset [LDM+09], the precursor of MP-Basset.

Every Java method specified by the message-passing program can always be executed if the method's guard (a concept implemented by Basset/MP-Basset) is enabled. In Basset/MP-Basset, S_{aux} contains the set of messages processed by the last transition, as explained in the example in Section 3.2. Note that S_{aux} might contain any data as long as the decomposition property can be shown. We (manually) verify that state properties are decomposed by checking if the property (Java assertion) only involves variables of the message-passing program and other variables for context switching required to satisfy transition decomposition.

3.4.2 *Selective Hashing*

The JPF's stateful optimization is implemented by serializing the state of the JPF Virtual Machine; the outcome of serialization is stored as a reached state. We explain this mechanism using code excerpts of JPF core (Listing 1). The serialization uses two data structures, a *(reference) queue* (Line 20) and a *buffer* (Line 21). The queue is an array containing references of objects in the JPF Virtual Machine. The buffer is an array of integers where each element holds the value of a primitive type. Initially, the queue contains references of the topmost classes of program. The serialization of the system state is done by calling the process method (Line 3). The references in the queue are processed one-by-one (Line 5) where the reference itself (Line 27) and the content of the referenced object is added to the buffer (Lines 35-50). Every object is a composition of primitive (e.g., int) and non-primitive types (e.g., `HashMap`). If it is a primitive type, its value is added to the buffer (Line 48), otherwise the reference is added to the end of the queue for further processing (Line 45). The serialization of the state terminates if every reference in the queue has been processed (Line 4). Finally, JPF uses a hashing function (not depicted) to compute the hash value of the buffer.

Serialization Example. Consider the simple actor program in Listing 2 written in Basset's Java library. Actors correspond to processes in our general system model. The `Driver` class is used to create the initial actors. In this example, two actors of class `FooActor` are created. For simplicity, no message is sent in this example. Consider the state of the system after executing the main function of the `Driver` class. In this state, the process method of the serializer is called with `refQueue`=$[ref_{a1}, ref_{a2}]$ where ref_{a1} and ref_{a2} denote the references of the two actors. As a result of the serialization (before calling the hash function), the buffer will contain $[1, 2, 11, 12]$ (whereas `refQueue` will contain $[ref_{a1}, ref_{a2}, ref_{foo1}, ref_{foo2}]$ where ref_{foo1} and ref_{foo2} denote references of `FooClass` in `a1` and `a2`, respectively).

Our Design of Selective Hashing. The heart of our implementation of selective hashing is an additional condition (Line 26) applied during

```
1  public class ReferenceQueue{
2    //...
3    public void process(ElementInfoProcessor proc){
4      for(Entry e = markHead; e!=null;){
5        proc.processElementInfo(e.refEi);
6        //...
7      }
8    }
9    public void processActorQueue(MPSerializer proc){
10     for(Entry e = markHead; e!= null;){
11       //...
12       processNamedFields(ei, ci, fields);
13       }
14   }
15 }
16
17 public class MPSerializer extends FilteringSerializer {
18   //FilteringSerializer implements ElementinfoProcessor{
19   //...
20   protected ReferenceQueue refQueue;
21   protected IntVector buf = new IntVector(4096);
22   public void processElementInfo(ElementInfo ei){
23     Fields fields = ei.getFields();
24     ClassInfo ci = ei.getClassInfo();
25     //SELECTIVE HASHING
26     if(StringSetMatcher.isMatch(ci.getName(), includeClasses, excludeClasses)){
27       buf.add(ci.getUniqueId());}
28       actorQueue = new ReferenceQueue();
29       actorQueue.add(ei);
30       actorQueue.processActorQueue(this);
31       //...
32       //processNamedFields(ei, ci, fields);
33       }
34   }
35   protected void processNamedFields(ElementInfo ei, ClassInfo ci, Fields fields){
36     FinalBitSet refs = getInstanceRefMask(ci);
37     //...
38     int[] values = fields.asFieldSlots();
39     for(int i = 0; i < values.length; i++){
40       //...
41       int v = values[i];
42       if(refs.get(i)){
43         //...
44         actorQueue.add(ei);
45         //refQueue.add(ei);
46         //...
47       } else
48         buf.add(v);
49     }
50   }
51   //...
52 }
```

Listing 1: Selective hashing via modified serializer in JPF. Our changes are highlighted in gray.

serialization. This condition enforces the rule that a reference is only processed if it is selected for inclusion[2]. In our current setting (not depicted) the set of excluded classes is empty whereas the set of included

2 Our implementation utilizes Basset's `StringSetMatcher` method.

```
public class FooActor extends Actor{
  int id;
  FooClass fooClass;
  public FooActor(int id){
    this.id = id;
    fooClass = new FooClass(id);
  }
  class FooClass{
    int foo;
    public FooClass(int id){
      foo = id+10;
    }
  }
}

public class Driver extends TestDriver{
  public static void main(String[] args){
    //...
    ActorName a1, a2;
    a1 = PlatformUtil.createActor(FooActor.class, 1);
    a2 = PlatformUtil.createActor(FooActor.class, 2);
    //...
  }
}
```

Listing 2: Message-passing system with two actors.

classes consists of classes extending `Actor` and the class (called `Cloud`) holding the set of pending messages. A new reference queue is created for each such class and it is processed recursively (Lines 28-30) similar to the original serializer. We remark that this mechanism cannot be implemented using the standard JPF API. Although include/exclude classes are supported by JPF, they are used to include/exclude *every* reference in the queue. Therefore, JPF's serializer makes no difference between an object reference within and outside an `Actor` (or `Cloud`) class.

Our Structured Serialization. We now explain another benefit of our solution which relates to symmetry reduction [MDC06], a promising optimization of model checking of distributed systems. Intuitively, most distributed systems are symmetric with respect to replicated processes, where replication may serve different goals such as fault-tolerance or enhanced performance. It has been shown that symmetry reduction can be extremely efficient in various practical applications of distributed systems [MDC06; BDH02]. Unfortunately, symmetry reduction has not yet established itself as an efficient *software* verification technique. In fact, to the best of our knowledge, the only attempt to implement general purpose symmetry reduction for software verification was the SymmSpin extension of the Spin model checker [CDE+08], an implementation that is no longer maintained [Sym].

Our modified JPF serializer for selective hashing paves the way for implementing symmetry reduction for message-passing systems à la Basset/MP-Basset. We aim at process-based symmetries that arise

from the free permutation of local process states. [3] JPF serializes the current state irrespective of the structure of the state. Therefore, we call JPF's serialization *unstructured*. The unstructured approach is in contrast to ours where actors (processes) and pending messages are serialized in isolation and appended to the final result. We call this *structured* serialization, which we apply for selective hashing. We observe that structured serialization can also be used for implementing symmetry reduction. The idea is that the output of structured serialization can be used to *canonicalize* (or normalize [ID96; CDE+08]) the state, which corresponds to mapping each state into a unique state by permuting the local states of processes. Canonicalization is the common way to implement symmetry reduction [MDC06] because it allows that only canonicalized states (and their successor states) need to be explored. Note that canonicalization is impossible using the unstructured serializer because the local state of a process is unknown to the serializer, as shown in the following example.

Symmetry Example. Assume that the system in Listing 2 is symmetric with respect to the IDs of the actors. This means that another execution of the system where actor `a1` and `a2` are given IDs, respectively, 2 and 1 (and not 1 and 2 as in Listing 2) is indistinguishable by the property of interest (i.e., the property holds or fails in both executions). The result of unstructured serialization in these two execution examples would be $[1, 2, 11, 12]$ and $[2, 1, 12, 11]$, respectively. Let the first state be the canonicalized state. After serialization, it is impossible to find out that $[2, 1, 12, 11]$ can be canonicalized into $[1, 2, 11, 12]$ because the information that $\langle 1, 11\rangle$ and $\langle 2, 12\rangle$ constitute the local state of actor a1 and a2, respectively, is dismissed throughout serialization. The structured serializer, on the other hand, outputs $[1, 11, 2, 12]$ and $[2, 12, 1, 11]$ for the respective states and it is aware of the information of how states are structured along processes, i.e., $[\langle 1, 11\rangle, \langle 2, 12\rangle]$ and $[\langle 2, 12\rangle, \langle 1, 11\rangle]$. Therefore, our structured serializer makes the canonicalization of the states for process symmetries possible.

3.4.3 *Selective Push-on-Stack*

Our implementation of selective push-on-stack is based on JPF's mechanism for the systematic exploration of branching execution. Intuitively, the execution of the Java program can branch if, given a state of the program, methods of different threads can be executed concurrently. In JPF, *choice generators* are used to associate such methods with the state. Depending on the interactions between processes (which are Java threads), they are obtained automatically by JPF (using a coarse over-approximation of concurrency) or they are registered by the user. Every time a new state is visited by JPF, the `forward` method is called (see Listing 3) and the state is pushed onto the search stack

3 There are efficient techniques to detect such symmetries [ID96; BDH02].

```
1  public class JVM{
2    //...
3    public boolean forward(){
4      //...
5      //SELECTIVE PUSH-ON-STACK
6      if(isBranchState()){
7        backtracker.pushSystemState();
8        updatePath();
9      }
10   }
11
12   private boolean isBranchState(){
13     return getChoiceGenerator() != null &&
14       !(getChoiceGenerator instanceof ThreadChoiceGenerator)
15     }
16 }
```

Listing 3: Selective push-on-stack with JPF's choice generators. Our changes are highlighted in gray.

(Line 8). According to the proposed selective push-on-stack strategy, this push operation is done conditionally (Line 6). The condition in our implementation is specific to Basset/MP-Basset where we know that a choice generator of type `ThreadChoiceGenerator` corresponds to single enabled transitions. This choice generator is responsible for switching context between actors (cf. example in Section 3.2) and it never specifies branching the execution (i.e., the set of enabled transition in the current state consists exactly of one transition). Therefore, selective push-on-stack can be implemented simply by checking if the current choice generator is a `ThreadChoiceGenerator` (Lines 13-14).

3.5 EVALUATION WITH FAULT-TOLERANT PROTOCOLS

In this section, we evaluate DBSS with representative fault-tolerant message-passing protocols. We measure the gain of DBSS compared to the highly optimized model checker MP-Basset [BKS+11; BSS+09]. The evaluation compares model checking time and memory (the number of visited states) for MP-Basset and DBSS.

Target Protocols and Properties. Our evaluation is based on the following protocols: Paxos consensus [Lam98], a regular register protocol in the style of ABD [ABND95] , and Zab atomic broadcast [JRS11]. We argue that these protocols constitute a representative and practical selection of fault-tolerant large-scale protocols. Firstly, these are all crash-tolerant protocols. The crash fault-model is widely used, also because a large and practical class of non-crash faults can be transformed into crash faults, as shown in [CFJ+12]. Secondly, Paxos, regular register, and Zab are conceptual and/or known to be practically relevant. For example, Paxos algorithm is in the core of commercial replication services [YCW+09], or the Zab protocol is part of Yahoo's Zookeeper open-source library used in different real deployments [Zoo]. As the

Protocol (# of procs.)	Configuration description
Paxos (6)	2 proposers each issuing ≤ 1 proposal, 3 acceptors and 1 learner
Faulty-Paxos (6)	Paxos (6) setting + One acceptor accepts all proposals (instead of those without a prohibiting promise
Faulty-Paxos (7)	Paxos (6) setting + 1 acceptor remembers the last accepted proposal (instead of the highest numbered accepted proposal)
Register (5)	3 base objects, 1 reader and a single writer
Faulty-Register (5)	Register (5) setting + Read finishing after concurrent write to return written value (instead of the value of the last preceeding write)
Register (6)	Register (5) setting with 4 base objects
Faulty-Register (6)	Faulty-Register (5) setting with 4 base objects
Register (7)	Register (6) setting with 5 base objects
Zab (6)	3 leaders, 3 followers
Zab (7)	4 leaders, 3 followers

Table 4: Configurations used in our experiments.

implementation of the protocols is not available to us[4], we use our prototype Java implementation in each case. In our evaluation, we use different settings of the above protocols (see description in Table 4). For the Paxos configurations we checked the agreement property which should hold [Lam98]. For Register we checked for the regularity property which should also be satisfied [ABND95]. In the case of the Zab protocol, we checked for liveness, a property that does not hold in general [JRS11]. We encoded the liveness property as a state property as demonstrated in [MSB+11]. In addition to the protocols and their specified properties, we *inject* faults in the Paxos (Faulty-Paxos) and Register (Faulty-Register) protocols and/or their properties to evaluate the debugging capability of DBSS.

Experimental Setup/Reduction Types. We run our experiments in a Deterlab testbed [Det] with 2 GHz Dual Xeon processors and 2 GiB memory, running on Ubuntu v.10.04. We compare the execution times and total number of visited states of different reduction types for each protocol. For comparability, DBSS uses the same scheduling of the transitions as MP-Basset's naive search.

First, we evaluate stateful against stateless model checking. Note that the search always terminates for our acyclic examples. We then evaluate DBSS without selective push-on-stack[5]. DBSS without selective push-on-stack experiments show the added benefit of push-on-stack technique exclusively. We only expect time reduction for these

4 Although Zookeeper is an open-source project, the code of Zab cannot be extracted as a stand-alone protocol.

5 In this reduction type, a new state s' is always pushed onto the stack even if the condition of line 32 in Algorithm 3 does not hold.

Configuration	Reduction Type	States	Time	Time reduction
Paxos (6)	MP-Basset stateful	13 044 613	22 h19 min	N/A
	DBSS without SPoS	5 606 047	11 h1 min	51 % (SF)
	DBSS	**5 606 047**	**10 h22 min**	**54 % (SF)**
	POR	191 081	23 min43 s	98 % (SF)
	DBSS + POR	**117 369**	**14 min22 s**	**39 % (POR)**
Faulty-Paxos (6)	MP-Basset stateful	96 802	10 min3 s	94 % (SL)
	DBSS	**70 543**	**8 min20 s**	**17 % (SF)**
	POR	3050	36 s	94 % (SF)
	DBSS + POR	**2786**	**31 s**	**14 % (POR)**
Faulty-Paxos (7)	MP-Basset stateful	>71 914 839	>192 h	N/A
	DBSS	**>66 651 310**	**>192 h**	**N/A**
	POR	129 533	21 min18 s	N/A
	DBSS + POR	**124 976**	**19 min29 s**	**9 % (POR)**
Register (5)	MP-Basset stateful	89 041	7 min31 s	14 % (SL)
	DBSS without SPoS	59 306	6 min20 s	16 % (SF)
	DBSS	**59 306**	**5 min39 s**	**25 % (SF)**
	POR	10 896	1 min5 s	86 % (SF)
	DBSS + POR	**8590**	**53 s**	**18 % (POR)**
Faulty-Register (5)	MP-Basset stateful	4965	34 s	66 % (SL)
	DBSS	**3376**	**27 s**	**21 % (SF)**
	POR	1936	24 s	29 % (SF)
	DBSS + POR	**1483**	**20 s**	**17 % (POR)**
Register (6)	MP-Basset stateful	2 269 797	5 h22 min	90 % (SL)
	DBSS without SPoS	1 475 845	4 h31 min	16 % (SF)
	DBSS	**1 475 845**	**4 h8 min**	**23 % (SF)**
	POR	96 641	11 min1 s	97 % (SF)
	DBSS + POR	**57 187**	**7 min26 s**	**33 % (POR)**
Faulty-Register (6)	MP-Basset stateful	94 348	10 min49 s	85 % (SL)
	DBSS	**56 222**	**8 min14 s**	**24 % (SF)**
	POR	4642	51 s	90 % (SF)
	DBSS + POR	**4642**	**48 s**	**6 % (POR)**
Register (7)	MP-Basset stateful	51 465 807	>192 h	N/A
	DBSS	**35 692 316**	**>192 h**	**N/A**
	POR	2 986 657	8 h21 min	N/A
	DBSS + POR	**1 656 212**	**6 h3 min**	**28 % (POR)**
Zab (6)	MP-Basset stateful	4580	54 s	86 % (SL)
	DBSS	**1876**	**26 s**	**38 % (SF)**
	POR	N/A	N/A	N/A
Zab (7)	MP-Basset stateful	8198	2 min4 s	80 % (SL)
	DBSS	**3132**	**38 s**	**69 % (SF)**
	POR	N/A	N/A	N/A

Table 5: Evaluation results of DBSS with/without selective push-on-stack (SPoS) compared with MP-Basset with/without stateful and partial-order reduction (POR) optimizations. Time reduction is computed with respect to base cases MP-Basset stateless (SL), stateful (SF), and stateful with partial-order reduction (POR).

experiments, as this reduction type cannot achieve memory reduction. The third reduction type measures the performance of DBSS, as explained in Section 3.3.2. Finally, we apply stateful partial-order

reduction (POR) alone (base case), then in combination with DBSS.[6] We use POR wherever it is applicable. For example, MP-Basset's implementation of POR does not apply for Zab (see more details later about the assumptions made by POR and DBSS).

Reduction Results. The results of our experiments are shown in Table 5. In the table, we report the results of DBSS with/without selective push-on-stack (SPoS) compared to MP-Basset with/without stateful and partial-order reduction (POR) optimizations. The results in the time reduction column are computed with respect to base cases MP-Basset stateless (SL), Stateful (SF), and stateful with partial-order reduction (POR). We omit the reductions in term of number of visited states as it is proportional to time reductions. In fault-injected instances, the search is stopped after finding the first bug, hence the search is non-exhaustive. We write `N/A` (not available) if POR is not available for the experiments or the reduction percentage is not available due to timeout (192 hours). For exhaustive searches that end with timeout, the value in the states column indicates the number of visited states at the time when the search stops.

Our main observations are as follows:

- **Stateful outperforms stateless search.** The stateful search finishes earlier than the stateless one in *all* exhaustive and fault-injected experiments – only the reduction of stateful over stateless search is shown in Table 5. In some cases (e.g. Paxos(6)), the stateful search terminates where the stateless search is infeasible (given our timeout). In other cases, stateful model checking reduces the search time by up to 94 % compared to stateless model checking.

- **DBSS improves efficiency.** DBSS is highly efficient as shown by the exhaustive search results, reducing the total number of visited states by up to 57 % and model checking time by up to 54 %. It also finds bugs up to 69 % faster than stateful model checking.

- **Selective push-on-stack time efficient.** Selective push-on-stack reduces model checking time by up to 9 % (see Register (5) experiment). Fault-injected cases with DBSS without selective push-on-stack are not displayed as they follow the same reduction trend as the exhaustive experiments.

- **DBSS efficient with POR.** When DBSS is used with POR, DBSS reduces model checking time and memory by up to 39 %, compared to the experiments with only POR.

6 MP-Basset implements different POR algorithms; we apply static POR for our experiments as it is more efficient than dynamic POR for the considered class of protocols [BSS+09].

Assumptions by POR/DBSS. The reduction achieved by POR can be significantly more than by DBSS. For example, POR reduces model checking time by 98 % for Paxos (6), whereas DBSS achieves a reduction of 54 %. This is only true given the assumptions made by POR that the execution of certain transitions is commutative [GVLH+96]. The soundness of POR can only be guaranteed if this assumption is verified. DBSS, in contrast to POR, makes no assumptions about the commutativity of transitions. For example, the simple static analysis in MP-Basset's POR implementation [BSS+09] is not applicable for Zab to verify the assumptions required by POR. DBSS is still applicable in this case and it achieves a time reduction of up to 69 %.

Scalability. We observe that the reduction achieved by DBSS changes with the number of processes. In fact, the time reduction of "DBSS + POR" is 18 %, 33 %, and 28 %, for the register with 5, 6, and 7 processes, respectively. One reason of this trend can be in the majority voting mechanism that the register (similarly to Paxos and Zab) uses for fault-tolerance. The majority of voters contains 2, 3 and 3 processes for Register (5), (6) and (7), respectively. A larger majority means more "equivalent" states for selective hashing because the writer has more choice in contacting different voters to observe the same voting result. This explains the improved reduction from 18 % to 33 % and also the more or less constant reduction of 33 % and 28 %. Note that DBSS experiments (without POR) show a slightly different trend for the register: 25 % and 23 % for Register (5) and (6) (timeout for Register (7)). We speculate that there are collisions on the outputs of the hash function due to the large number of states in these experiments.

3.6 RELATED WORK

Model Checkers. Mainstream software model checkers include explicit-state checkers for C programs such as Verisoft [God97] and Spin [Hol97], for Java programs [Jpf], symbolic execution engines such as DART for C [GKS05] or KLEE for low-level (byte)code [CEF+96], and dedicated solutions for message-passing systems such as Modist [YCW+09] or Mace [KAJ+07]. Selective push-on-stack is inherently related to depth-first search and, as such, it can be implemented in any explicit-state model checker (like Verisoft, Spin, Modist, or Mace). On the other hand, selective hashing is not restricted to explicit-state model checking and it can also be used to decrease the number of variables needed for a symbolic encoding of the state.

Some model checkers (such as Mace [KAJ+07]) offer the user an interface to exclude certain state information from the representation of the state. As a result, similar to selective hashing, the excluded state information is not considered by stateful model checking. It is, however, left to the user to guarantee the correctness of model check-

ing. Our notion of decomposition formalizes a sufficient condition of correctness, which can be applied by users of these model checkers.

Reductions. Broadly-studied and intuitive reductions are partial-order (POR) [GVLH+96] and symmetry reductions (SR) [MDC06]. Figure 11 demonstrates that DBSS is not a special case of these reductions. Firstly, POR is based on the idea of swapping the order of commutative transitions but the path ($s_1 \to s_2 \to s_3 \to s_6 \to s$) that is excluded in the reduced state graph in Figure 11 cannot be obtained by re-ordering the transitions of another path in the graph. Formally, considering the mainstream POR semantics, Figure 11 is not a stubborn/persistent/ample set reduction on the exploration in Figure 10 because in every state of the reduced state graph the number of enabled transitions is the same as in the unreduced one. Secondly, SR is based on the symmetrical structure of the state graph but there is no such symmetry in Figure 10. Formally speaking, there is no permutation acting over the set of states (the formal notion of symmetry [MDC06]) that would preserve the transition relation: In order to symmetry reduce Figure 10 into Figure 10, a permutation would have to transpose s_5 and s_6 but these two states are "asymmetric" because of s_4.

To the best of our knowledge, all known reduction approaches that work with depth-first search, such as SR, POR, or dynamic interface reduction (DIR) [GWZ+11], can be directly combined with DBSS. Reductions of stateless model checking such as symmetric transitions [God99] or dynamic partial-order reduction [FG05] would only benefit from selective push-on-stack. The reduction achieved by DBSS is based on the assumption of a sound decomposition. Other reductions are also based on (other) assumptions: POR assumes the commutativity of executing transitions, SR depends on symmetric execution patterns, DIR needs to be tailored depending on how the execution of different processes can interleave.

3.7 CONCLUSION

We have proposed decomposition-based stateful search (DBSS) as an improvement of explicit-state software model-checking. Given a sound decomposition we showed the correctness of DBSS. Also, we have built a proof-of-concept implementation based on the Java Pathfinder search engine for general message-passing Java programs. Our evaluation of DBSS with various representative message-passing protocols shows extensive reduction both in time and state space. We show that DBSS outperforms partial-order reduction (POR) in most cases but can also be combined with it to achieve even higher reductions.

Part III

DEPENDABILITY OF MULTI-THREADED PROGRAMS

4

PBMC: SYMBOLIC PROGRAM SLICING ON CONCURRENT PROGRAMS

In the last chapter we presented an approach that can improve the efficiency of explicit model checking based on the notion of decomposition of states. We now shift our attention to a symbolic variant of model checking, bounded model checking (BMC). Performing BMC on concurrent programs is known to be a hard problem. After building the BMC formula that encodes the program's behavior and the specification, an SMT solver is used to find possible assignments, i.e., executions violating the specification. While generating the BMC formula can be done efficiently, the solving time usually dominates the overall runtime. In this chapter, we leverage the structure of the specification to enhance the BMC formula to guide the solver towards assignments (i.e., executions) that are relevant. The contributions presented in this chapter are based on a previously published work in [SBS15].

This chapter is organized as follows. We start by providing a general overview of the contribution in Section 4.1. Sections 4.2 and 4.3 provide a motivating example and discuss the related work. In Section 4.4, we formalize and prove the correctness of projections. Section 4.5 describes the symbolic encoding of projections and Section 4.6 shows our evaluation results.

4.1 OVERVIEW

Automated verification of complex programs is known to be a hard problem. The complexity of the task grows exponentially when the considered programs exhibit concurrent behavior [Val98]. Bounded model checking (BMC) [BCC+99] is a widely used verification technique, e.g., [CKL04]. In BMC, a formula encoding the behavior of the program is computed and passed to an SMT/SAT solver along with the negation of the property. The solver then checks whether there exists an execution leading to a state violating the property. Thanks to the recent advances in the field of SAT solving, bounded model checking is becoming a problem solution for verification of concurrent programs [AKT13; Esb; Llb]. The efficiency can be greatly improved by constraining the search space depending on the property of interest.

Concretely, the encoding of the program is constrained by excluding behavior that is irrelevant or redundant with respect to the property. For example, the solver can be instructed to partially order (instead of totally order) transitions of the program, e.g., [AKT13].

In this chapter, we propose *projections* to constrain the search space of a bounded model checker. Conceptually, projections are slices of a program with respect to a set of variables. They are especially useful for the analysis of concurrent programs. For example, a projection with respect to the local state of a process may exclude transitions of other non-interfering processes. As a result, the interleavings of the excluded transitions (exponential in the number of transitions) do not have to be considered by the solver. Intuitively, projections consist of executions which only contain transitions directly or indirectly affecting the variables of interest.

Contributions. We introduce projections, an adaptation of program slices to general transition systems, and show that they preserve the relevant safety properties of a program. The idea of projections is a general one and is independent of how the program states are explored. Our second contribution is that we present a symbolic encoding of projections for concurrent programs; we call the encoding PBMC (projection-based BMC) because it can be used for efficient bounded model checking. Note that although we concentrate on concurrent programs our result equally holds for single-threaded programs. Interestingly, PBMC can be seen as a form of *dynamic program slicing* [AH90]. In contrast to existing program analysis approaches where static program slicing is applied prior to the actual analysis, e.g. [CKL04; DHH+06; RH07] , PBMC enumerates slices on-the-fly. The resulting slices are as precise as the dynamic ones because they are calculated based on feasible executions. To the best of our knowledge, PBMC is the first application of dynamic program slicing with BMC. Our final contribution is that we implement a prototype of PBMC and use it to verify simplified versions of a set of concurrent programs where program slicing in its traditional form fails to reduce the size of the program. The experiments show substantial verification time reductions compared to traditional BMC. In summary this chapter makes the following contributions:

- We propose a formal work for projections and prove the soundness of our approach,
- we implement PBMC a projection based bounded model checker for a prototype language of concurrent programs communicating through shared variables, and
- we evaluate the performance of PBMC and compare it to traditional BMC.

4.2 MOTIVATING EXAMPLE

We motivate our approach using a simple example. Consider the program shown in Figure 13a consisting of three concurrent and sequential processes sharing different variables and an array B. We label the instructions on the different program locations l_1, l_2, l_3 and l_4. Assume that we are verifying a property which involves only the variable y. That is, we are interested in the values that y can take in any possible run of the program. There are 12 possible interleavings of the instructions. Assuming that initially $a = 0, b = 1, x = 0$ and $B[k] = 10 + k$ for every k-th position in the array, we list in Figure 13b the possible runs which may be relevant. Every transition in the runs corresponds to a concrete execution of an instruction. For instance, t_5 and t_2 are two different transitions, but correspond to the same instruction l_4. When analyzing the program statically[1], one can only reason in terms of instructions. By doing that, it is clear that the program contains three *"dependencies"*: The execution of l_3 always *depends on* that of l_1 because l_1 always writes to variable a which l_3 reads from. For the same reason, the execution of l_4 depends on l_3. More interestingly, the execution of l_3 depends on l_2 only if $a = b$. This corresponds in Figure 13b to the sequence σ_3, as transition t_3 writes in a position in B that t_7 reads from. We refer to σ_1, σ_2 and σ_3 as *projections* of the program on the set of variables $\{y\}$. Intuitively, in order to preserve all the possible values that y can take, it is enough to consider sequences where every transition t either writes to y, or there is another transition after it which also writes to y and transitively depends on t. For example in σ_3, transition t_3 is included because it influences t_7 which in turn influences t_8. Transition t_8 is kept in the projection since it writes to y. In general, projecting a run on a set of variables F means that we keep every transition that writes to variables in F or affects another transition after it already included in the projection.

Given that the safety property of interest involves a subset of variables, the verification time of such programs can be greatly reduced if we constrain the exploration to projected executions. We use BMC to symbolically describe the behavior of the program and add constraints that characterize projections. By doing so, we constrain the search space of the model checker but still preserve all possible values that the variables in the property can take. If a state violating the property is reachable, a projection is returned. Furthermore, as projections contain only relevant instructions, they are easier to interpret and analyze.

Note that static program slicing techniques [Wei81] applied to this program would not help in the reduction of the search space as it will return a copy of the whole program. This is due to the conditional

1 Ignoring our assumption about the initial values of the variables.

$$
\begin{array}{lll}
P_1\{ & P_2\{ & P_3\{ \\
\quad l_1 : a = 1; & \quad l_2 : B[b] = 5; & \quad l_3 : x = B[a]; \\
\quad \} & \quad \} & \quad l_4 : y = x + 7; \\
 & & \quad \}
\end{array}
$$

(a) An example concurrent program.

σ_1	σ_2	σ_3
$t_1 : x = 10$	$t_3 : a = 1$	$t_3 : a = 1$
$t_2 : y = 17$	$t_4 : x = 11$	$t_6 : B[1] = 5$
	$t_5 : y = 18$	$t_7 : x = 5$
		$t_8 : y = 12$

(b) 3 projections on y out of 12 possible runs.

Figure 13: A motivating example

dependencies of instructions accessing arrays and the concurrency, in which case it's not clear before execution whether an instruction affecting a variable of interest v will be actually executed before another instruction assigning a value to v. For instance, it's not clear whether l_1 will be executed before l_3. Besides, if the values of a and b depend on some non-deterministic behavior, e.g., concurrency or user input, it might not be possible to predict whether $a = b$. In this case, static program slicing conservatively over-approximates the slices.

4.3 RELATED WORK

Program Slicing. PBMC is the first approach to combine dynamic slicing and BMC. Our approach is based on a notion similar to program slicing [Wei81]. Static program slicing has been used to reduce the size of programs under verification, e.g., [CKL04; DHH+06; RH07] where a slice of the program is computed and passed to the model checker. However, the returned slice is an over-approximation of the instructions that are in fact relevant to the slicing criterion. This is due to the fact that inferring dependencies statically, a prerequisite for deriving slices, is hard with the presence of concurrency [Kri04; RH07]. In PBMC, formulas describing precisely when dependencies occur are generated passed to a solver along with the verification formula. In dynamic slicing [AH90], the program is run with concrete input and the slicing is done directly on the execution path. Although the slices returned by dynamic slicing techniques are accurate, they only concern the considered execution path. To use dynamic slicing with verification, one would have to enumerate all paths which contradicts the purpose of using slicing for reducing the number of explored

paths. The slicing in PBMC is dynamic since only reachable, and therefore feasible, paths are sliced.

Partial-Order Reduction. A common technique against state space explosion is partial-order reduction (POR) [GVLH+96]. Whereas POR's reduction comes from executing commutative transitions in one representative order, in PBMC it is based on the notion of conflicting read/write operations. Existing POR semantics, however, do not subsume projections. Widely-known POR semantics such as stubborn sets [Val98; BKS+11], persistent sets [GVLH+96], and ample sets [CJGK+18] guarantee preserving all deadlocks of the program. In some cases [GVLH+96], the preservation of deadlocks also entails that of local states. On the other hand, projections do not necessary preserve all deadlocks nor all local states. Existing POR techniques include [GFY+07; FG05; BKS+11; WKO13; KWG09; AAJ+14] among others.

BMC. An interesting way of combining slicing with BMC is described in [GG08]. Tunneling and slicing-based reduction makes use of slicing to decompose a BMC formula into disjoint smaller instances covering subsets of the program. These formulas are constructed such that the original formula is satisfiable only if at least one of the smaller instances can be satisfied. Nevertheless, the used slicing is static and therefore is imprecise.

Our approach for reduction is similar to MPOR [KWG09], where constraints are added to the BMC formula to guide the search. The constraints used in this approach are based on Mazurkiewicz's traces [Maz87], the underlying semantics for most POR theories. Our symbolic encoding of the transition system enhances the encoding used in MPOR. Furthermore, POR and projections are orthogonal techniques that can be used in combination for better reductions as demonstrated in [DHH+06]. This is the case because the definition of path projections alone still allows for two Mazurkiewicz equivalent paths to be considered in the search. We argue that our encoding can be augmented by the constraints of MPOR for better performance. To see this, consider two executions t_4, t_1, t_2, t_3 and t_2, t_1, t_4, t_3 such that t_3 depends on both t_1 and t_2. From both executions we can derive projections t_1, t_2, t_3 and t_2, t_1, t_3, respectively. Since there is no dependency between t_2 and t_1, both projected executions are Mazurkiewicz equivalent. It follows then that it is sufficient to consider one of the projections.

Encodings. We adapted the encoding used in [KWG09] which does not require unwinding of loops as in [CKL04], [BAM07] or [SW11]. Yet, unwinding loops may be beneficial and allow different encoding, e.g., using single static assignment form to reduce the number of variables in the formula. The idea behind projections is independent of the used encoding and therefore can be adapted for use with other BMC formulas. For instance, in CBMC [AKT13], transitions are associated with clock variables that reflect how they are (partially-)ordered.

Intuitively, a path corresponds to a partial order over transitions where only dependent ones are strictly ordered. Thus, constraints are used to enforce a total order on the dependent transitions. Using such an encoding, the model checker might still explore some partial orders which are not relevant to the property. Hence, two dependent transitions will still have to be ordered although they might not have any influence on the property. Given a subset of variables, we argue that projection constraints can be added to such an encoding to further reduce the number of interleavings of dependent transitions. This can be done by constraining the used read-from relation according to the definition of projections.

Other Symbolic Approaches. Another possibility is to use slicing on-demand to refine the search for assertion violations. For instance, Path slicing [JM05] is a technique that has been implemented within the Blast model checker [BHJ+07] which makes use of counterexample guided refinement techniques [CGJ+00]. In Blast, slicing is used to simplify the counterexample analysis phase that serves the purpose of refining the search. Our approach is different from path slicing in the sense that the search for bugs is constrained from the beginning using projections to guide the solver toward feasible counterexamples. PBMC, and BMC based approaches in general, are fundamentally different from Blast, and other tools such as [McM06; WKO13; PVB+13; LQL12], where the verification formulas are generated and refined incrementally with the help of the solver. In BMC, a single formula describing the whole program is computed statically and the exploration work is deferred to the SMT solver. A comprehensive discussion of the advantages and disadvantages of incremental generation and refinement of the verification formula over BMC approaches is beyond the scope of this work.

4.4 PROPERTY PRESERVATION WITH PROJECTIONS

4.4.1 *System Model*

We abstract programs by general transition systems, where a transition may read and/or write a set of variables.

General Transition Systems. Formally, the system is defined as a tuple $TS = (S, S_0, T)$ where S, $S_0 \subseteq S$, and $T = \{t | t : S \rightharpoonup S\}$ are the set of states, initial states, and the set of *transitions*, respectively. In the rest of this chapter, we will always write s_0 to refer to an initial state.

A program defines a set of atomic instructions (e.g., lines of code). In every state a (possibly empty) set of transitions is eligible for execution. For example, transitions may correspond to instructions of different processes. If the set of transitions is non-empty, one of them is executed moving the program to a unique successor state. Formally, a transition $t \in T$ is a partial function such that $t(s) = s'$ iff t can be

executed in s and it leads to state s'.[2] In that case, we say that t is enabled in s. For convenience, we also write $s \xrightarrow{t} s'$ if $t(s) = s'$.

A finite path σ in the transition system TS is a sequence $s_0 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s_n$, also written as $s_0 \xrightarrow{t_1,t_2,\ldots,t_n} s_n$, such that $t_{i+1}(s_i) = s_{i+1}$ for all $0 \leq i < n$. In that case, we write $\sigma \in TS$.

In addition, we assume that every state s assigns a value $s(v)$ to every $v \in V$, where V is the set of variables. Given a set $F \subseteq V$, we refer to the values assigned by s to variables $v \in F$ by $s(F)$. We write $s(F) = s'(F)$ for two states s and s', if for all $v \in F$, $s(v) = s'(v)$.

Dependency Relation. The execution of a transition involves reading from and writing to a subset of variables. We assign to every transition t a *read/write set* of variables [GVLH+96; Val98], denoted as $r(t)/w(t) \subseteq V$ respectively. A transition t is said to read from (write to) a variable v if $v \in r(t)$ ($v \in w(t)$). The read set of a transition contains all variables that may have an influence on whether a transition is enabled and the outcome of its execution. On the other hand, the write set of a transition consist of the variables that it might modify.

Formally, $w(t)$ is defined such that for every $v \in V$, $v \in w(t)$ iff there are $s, s' \in S$ such that $s \xrightarrow{t} s'$ and $s(v) \neq s'(v)$. Note that the write set of a transition does not include a variable it never modifies. We define the read set $r(t)$ as the smallest set such that for every $s, s' \in S$, if $s(v) = s'(v)$ for every $v \in r(t)$, then

- t is enabled in s iff t is enabled in s', and
- if t is enabled in s, then for every $v' \in w(t)$, $t(s)(v') = t(s')(v')$.

We define a *dependency* relation $D \subseteq T \times T$ to model any interference between transitions. A transition t depends on a transition t' if t reads from a variable that t' writes to. In that case, we also say that t influences t' and write $(t, t') \in D$.

Definition 3 (Dependency Relation)**.** *Given two transitions t and $t' \in T$, we say that t' depends on t and write $(t, t') \in D$ iff $r(t') \cap w(t) \neq \emptyset$.*

Note that two transitions only writing to the same variable are not considered to be dependent as the execution of one of them before the other does not influence the behavior of latter.

4.4.2 *Projections*

In this section, we propose the projection semantics and present a theorem that guarantees that preserving projections on a set of variables is a sufficient condition for preserving properties defined over those variables.

2 For simplicity we focus on deterministic systems, although our results equally hold for non-deterministic programs.

First, we give a formal definition of path projections on a set of variables. Intuitively, a projection of a path σ on a set of variables F is a sequence of transitions containing every transition t that either writes into a variable in F, or there is a transition t' after it such that t' depends on t and t' is also in the projection.

Definition 4 (Projection)**.** *Given a set of variables $F \subseteq V$ and a path $\sigma = s_0 \xrightarrow{t_1,\ldots,t_n} s_n$, $\sigma|_F = t_{j_1}, t_{j_2}, ..., t_{j_k}$ is said to be a projection of σ on F, if $1 \leq j_1 < j_2 < ... < j_k \leq n$ and for all $1 \leq i \leq n$, $i \in \{j_1, j_2, ..., j_k\}$ iff:*

(a) $w(t_i) \cap F \neq \emptyset$, or

(b) there exists $j \in \{j_2, j_3, \ldots, j_k\}$ such that $i < j$ and $(t_i, t_j) \in D$.

Property Preservation. Our main result is that projections can be used to constrain the search space of a model checker. For that purpose we must guarantee that projections on a set of variables preserve the properties of those variables. Assume two transition systems TS, TS' and a set of variables $F \subseteq V$. If for every path $\sigma = s_0 \xrightarrow{t_1,t_2,\ldots,t_n} s_n \in TS$, there exists a path $\sigma' = s_0 \xrightarrow{t'_1,t'_2,\ldots,t'_m} s'_m \in TS'$ such that $s_n(F) = s'_m(F)$, we say that TS' *preserves the properties* of F in TS. Furthermore, we say that TS' *preserves the projections* of F in TS, if for every path $\sigma = s_0 \xrightarrow{t_1,t_2,\ldots,t_n} s_n \in TS$, there exists a path $\sigma' = s_0 \xrightarrow{t'_1,t'_2,\ldots,t'_m} s'_m \in TS'$ such that $\sigma|_F = t'_1, t'_2, ..., t'_m$.

Theorem 5 (Property Preservation)**.** *Let TS and TS' be two transition systems and $F \subseteq V$ a set of variables. If TS' preserves the projections of F in TS then it also preserves the properties of F in TS.*

We prove Theorem 5 via a series of lemmas. First, we introduce the following auxiliary definitions: Let t_i, t_j be two transitions, $\alpha = t_1 t_2 \ldots t_n$ a sequence of transitions, and $\sigma = s_0 \xrightarrow{\alpha} s_n$ be the resulting path. For convenience, we will write $t_i \in \sigma$ and $t_i \in \alpha$ if $i \in \{1, 2, \ldots, n\}$. Furthermore, if $j \in \{i+1, \ldots, n\}$, we write $t_i <_\sigma t_j$ or $t_j >_\sigma t_i$.

First, we show that between two successive transitions in a projection $\sigma|_F$, the values assigned to variables in F and the ones read by any transition in $\sigma|_F$ after the second transition remain unmodified by all transitions outside the projection.

Lemma 6. *Let $\sigma = s_0 \xrightarrow{t_1,t_2,\ldots,t_n} s_n$ be a path, $F \subseteq V$ a set of variables and t_k, $t_{k'}$ two transitions such that $t_k <_\sigma t_{k'}$ and for every $t_{k-1} <_\sigma t_i <_\sigma t_{k'+1}$ $t_i \notin \sigma|_F$. For every $t_{k-1} <_\sigma t_j <_\sigma t_{k'+1}$, $t_q >_\sigma t_{k'}$ such that $t_q \in \sigma|_F$ and $v \in r(t_q)$, $s_j(F) = s_{k-1}(F)$ and $s_j(v) = s_{k-1}(v)$.*

Proof. Let σ be a path, $\sigma|_F$ its projection on a variable set $F \subseteq V$, and two transitions t_k and $t_{k'}$ as described above. We know that for all $t_{k-1} <_\sigma t_j <_\sigma t_{k'+1}$, $w(t_j) \cap F = \emptyset$. Otherwise, t_j would

be included in $\sigma|_F$ between t_k and $t_{k'}$ (Def. 4). This means that $s_{j-1}(F) = s_j(F) = s_{k-1}(F)$. Given a transition $t_q \in \sigma|_F$ such that $t_q >_\sigma t_{k'}$, we assume that there is a variable $v \in r(t_q)$ such that for a $j \in \{k, \ldots, k'\}$, $s_j(v) \neq s_{k-1}(v)$. Let j be the first such an index. This implies that $s_{k-1}(v) = s_{j-1}(v) \neq s_j(v)$. We then have $v \in w(t_j)$ and therefore $r(t_q) \cap w(t_j) \neq \emptyset$. From Definition 3 it follows that $(t_j, t_q) \in D$. Consequently, t_j should also be included in $\sigma|_F$. This contradicts our initial assumption. □

With the help of Lemma 6 we show that every projection is also a path and that it reaches a state where the assigned values to variables in F are the same as in the original path.

Lemma 7. *Let $\sigma = s_0 \xrightarrow{t_1, t_2, \ldots, t_n} s_n$ be a path and $\sigma|_F = t_{j_1}, t_{j_2}, \ldots, t_{j_k}$ its projection on variable set $F \subseteq V$. Then there exists a path $\sigma' = s_0 \xrightarrow{t_{j_1}, t_{j_2} \ldots, t_{j_k}} s'_k$ such that $s_n(F) = s'_k(F)$.*

Proof. We separately consider the case where $\sigma|_F$ is empty, i.e. contains no transition. In this case, we have for every $t_i \in \sigma$, $t_i \notin \sigma|_F$. From Lemma 6, it follows that $s_0(F) = s_n(F)$ and σ' exists as an empty path.

Now we assume that $\sigma|_F$ contains at least one transition. We start by proving, for every $1 \leq q \leq k$, the existence of the path that consists of the first q transitions of $\sigma|_F$, that $s'_q(F) = s_{j_q}(F)$ and that for every $v \in r(t_j)$ such that $t_j >_\sigma t_{j_q}$ and $t_j \in \sigma|_F$, $s'_q(v) = s_{j_q}(v)$. The proof is an induction on the number of the first q transitions in the projection. Consider the case of the first transition t_{j_1} in the projection. Since $\forall i \in \{1, \ldots, j_1 - 1\}$ $t_i \notin \sigma|_F$, we know that $s_0(F) = s_{j_1 - 1}(F)$, and that for every $v \in r(t_{j_1})$, $s_0(v) = s_{j_1 - 1}(v)$ (Lemma 6). Thus, since t_{j_1} is enabled in $s_{j_1 - 1}$, it is also enabled in s_0, and there is a state s'_1 such that $s_0 \xrightarrow{t_{j_1}} s'_1$ and $s'_1(v) = s_{j_1}(v)$ for every $v \in w(t_{j_1})$ (read set definition). Since for every $v \in F$ such that $s_0(v) \neq s'_1(v)$ is in $w(t)$ (write set definition), it follows that $s'_1(F) = s_{j_1 - 1}(F)$. We assume now that the property holds for the first q transitions and prove it after considering the $q+1$-th transition $t_{j_{q+1}}$. From Lemma 6 it follows then that $s_{j_q}(F) = s_{j_{q+1} - 1}(F)$ and $s_{j_q}(v) = s_{j_{q+1} - 1}(v)$ for every $v \in r(t_j)$ such that $t_j > t_{j_q}$ and $t_j \in \sigma|_F$. Using the induction assumption it follows then that $s'_q(F) = s_{j_{q+1} - 1}(F)$ and $s'_q(v) = s_{j_{q+1} - 1}(v)$. Consequently, t_{q+1} is enabled in s'_q and there exists a state s'_{q+1} such that $s'_q \xrightarrow{t_{q+1}} s'_{q+1}$, $s'_{q+1}(F) = s_{j_{q+1}}(F)$ and $\forall v \in w(t_{j_{q+1}})$, $s'_{q+1}(v) = s_{j_{q+1}}(v)$ (read/write set definitions). Let $v \in r(t_j)$ such that $s'_{q+1}(v) \neq s_{j_{q+1}}(v)$. This means that there is a variable $v' \in w(t_{j_{q+1}})$ such that $s'_{q+1}(v') \neq s_{j_{q+1}}(v')$ which is a contradiction.

Now that we have proved the property, we know that for $q = k$ we have $s'_k(F) = s_{j_k}(F)$ and that the path σ' exists. We know that for every $i \in \{j_k + 1, \ldots, n\}$ we have $t_i \notin \sigma|_F$ and $w(t_i) \cap F = \emptyset$ since otherwise t_i would be included in the projection (Def. 4). It implies then that $s_n(F) = s_{j_k}(F) = s'_k(F)$. □

Lemma 7 also implies that, given a transition system TS, a projection preserving transition system TS' always exists. This is true because from a path in TS one can always construct a valid projection path from it. Proving Theorem 5 is now straightforward.

Proof. Let $\sigma = s_0 \xrightarrow{t_1,t_2,\ldots,t_n} s_n$ be a path in TS and F a set of variables. The projection preservation implies that there exists a path $\sigma' = s_0 \xrightarrow{t'_1,t'_2\ldots t'_m} s'_m$ in TS' such that $\sigma|_F = \sigma'$. From Lemma 7 follows that $s_n(F) = s'_m(F)$. □

We have just proved that every reachable combination of values that the variables in F can take, is also reachable through a projection. In other words, Theorem 5 allows us to safely narrow down the search space of a model checker to projections, while still preserving all possible paths of a program.

4.5 PBMC: A SYMBOLIC IMPLEMENTATION

In this section, we show how we implemented projections semantics for process-based concurrent programs in PBMC.

4.5.1 *Process-Based Concurrent Programs.*

First, we informally describe how general concurrent programs can be expressed as a transition system. We assume a general shared memory model where a set of *processes* communicate via *shared variables*. In the corresponding transition system, a state consists of variables, and every transition is associated with a process. Processes are *sequential*. This means that two transitions that are enabled in a state must be from different processes. Hence, for every state s, a process has at most one enabled transition. Sequential processes can be modeled using an auxiliary variable for every process, called program counter. The program counter variable of a process can only be accessed by the process itself and designates the instruction that can be executed next.

4.5.2 *Projection Encoding*

Given a concurrent program, a property, and a fixed depth k, bounded model checking encodes a formula that an SMT/SAT solver can check for satisfiability. The property is true for some path iff the formula is satisfiable. More precisely, the formula is of the form $\Phi = \rho \wedge \Psi$ where ρ denotes the property formula and Ψ encodes a path of length k. The formula Φ is satisfiable iff there exists a path of at most k steps that satisfies ρ. To check whether the property ρ is valid for every possible path of a maximal length k, it suffices to replace it with its negation in Φ and prove the unsatisfiability of the resulting formula.

In the following, we explain how we encode Ψ to implement projections. The basic (unprojected) encoding adapts the structure used in [KWG09]. Let F be the set of variables which appears in the property formula. To model the changes affecting the state of the program throughout the path we create for every $v \in V$ and $0 \leq i \leq k$ a variable v^i to represent the content of v in the i-th state of the path.
Core Formula. A path is only valid if it starts from an initial state. We add a constraint I to encode this fact.

$$I := \bigwedge_{v \in V} \left(v^0 = s_0(v)\right)$$

Let L be the set of all the instructions in the program. For every transition $t \in T$, we refer to the instruction it corresponds to, with $inst(t) \in L$. Given an instruction $l \in L$, let $trans(l)$ be the set of transitions that are mapped to it. In every step $0 \leq i \leq k-1$, we model the possible selection of an instruction l using a formula denoted as T_l^i. If no instruction is selected for a step i, for instance because the length of the returned path is smaller than k, an additional constraint M makes sure that the variables remain unmodified for that step. To guide the solver to only consider projections on F, we add a constraint C_F. Setting C_F to *true* results in the solver considering every possible path. To encode all possible projections on set F, we obtain the following formula:

$$\Psi := I \wedge M \wedge C_F \wedge \bigwedge_{i=0}^{k-1} \bigwedge_{l \in L} T_l^i$$

Transition Encoding. For each instruction $l \in L$, we add an instruction constraint l^i that represents the changes that occur when l is executed at step i. We introduce variables sel^i that encode the instruction choice in every step: $sel^i = l$ iff instruction l was selected for execution in step i. Not that, due to process sequentiality, the selection of an instruction l implies the execution of a corresponding unique transition $t \in trans(l)$ given by the variables v^i. To describe the selection of instructions at different steps we make use of the sel^i variables and the instruction constraints:

$$T_l^i := sel^i = l \implies l^i$$

If an instruction l is selected for execution at step i, then l^i should hold, i.e, the variables should be updated accordingly. Otherwise, if no instruction is selected at step i, every variable in the system remains unchanged:

$$M := \bigwedge_{i=0}^{k-1} \left(\bigwedge_{l \in L} sel^i \neq l \implies \bigwedge_{v \in V} v^{i+1} = v^i \right)$$

If the depth value k is larger than the length of the path satisfying Ψ, some steps are filled with "dummy" instructions[3]. In this case, the solver will spend some time trying to figure out in which position to place the dummy instructions. We found it more efficient to force the solver to place those instructions at the beginning of the path such that there are no gaps, i.e., no dummy instruction is chosen after a "non-dummy" instruction has been selected. We do this by adding a formula that further constrains the assignment of the sel^i variables. We omit the formula as it is an optimization not necessary for the correctness of Ψ.

Projection Encoding. We describe how the projection constraint C_F is generated for a set of variables F. The dependency relation D is encoded using variables d_{ij} which are true iff there is a transition t_1 which is executed at step i and a transition t_2 executed at j such that $(t_1, t_2) \in D$. Specifically, we have:

$$d_{ij} := \bigvee_{t_1 \in T} \left(sel^i = inst(t_1) \wedge \bigvee_{t_2 \in \{t \mid (t_1,t) \in D\}} sel^j = inst(t_2) \right)$$

C_F directly translates the definition of projections (Definition 4):

$$C_F := \bigwedge_{i=0}^{k-1} \left(\bigvee_{t \in \{t' \mid w(t') \cap F \neq \emptyset\}} sel^i = inst(t) \vee \bigvee_{j=i+1}^{k-1} d_{ij} \vee \bigwedge_{l \in L} sel^i \neq l \right)$$

Informally, for every selected transition t_i either it writes into a variable included in F or there is a transition t_j in the projection after it such that $(t_i, t_j) \in D$. The last clause allows dummy instructions to be selected without rendering the formula unsatisfiable.

Examples. We show how we encode instructions based on the examples of simple assignments and if statements in our process-based concurrent system model. To model process sequentiality, we define program counter variables $pc_p \in V$ for every process p.

Let l be an *assignment* $x := e$ at a process moving the program counter from loc_1 to loc_2, then $l^i :=$

$$pc_p^i = loc_1 \wedge pc_p^{i+1} = loc_2 \wedge x^{i+1} = e^i \wedge \bigwedge_{v \in V \setminus \{x, pc_p\}} v^{i+1} = v^i$$

Considering an *if statement* $if(c)$ that moves the program counter from loc_1 to location loc_2 if c evaluates to true and to loc_3 otherwise, we have $t^i :=$

$$pc_p^i = loc_1 \wedge \Big((c \wedge pc_p^{i+1} = loc_2) \vee (\neg c \wedge pc_p^{i+1} = loc_3) \Big) \wedge \bigwedge_{v \in V \setminus \{pc_p\}} v^{i+1} = v^i$$

3 More precisely, the solver will assign a value to sel^i which does not correspond to any of the instructions.

Dependency Encoding. To illustrate how the dependency relation is encoded, we consider the motivating example in Section 4.2. Because of conflicting read/write accesses, we have three potential dependencies: l_3 depends on l_1, l_4 depends on l_3 and l_3 depends on l_2. The first two dependencies hold for any two transitions associated with the instructions. For instance, every transition corresponding to l_3 depends on every transition associated with l_1. On the other hand, the third dependency holds only if $a = b$. To encode d_{ij}, we must consider every possible dependency. First, there is a dependency if $sel^i = l_1$ and $sel^j = l_3$ or $sel^i = l_3$ and $sel^j = l_4$. For the dependency between l_3 and l_2, we must include the condition $a = b$. Concretely, the following should hold: $sel^i = l_2$, $b^i = a^j$ and $sel^j = l_3$. In summary, to have a dependency between step i and j the following formula should hold:

$$d_{ij} := (sel^i = l_1 \wedge sel^j = l_3) \vee (sel^i = l_3 \wedge sel^j = l_4) \vee (sel^i = l_2 \wedge (b^i = a^j \wedge sel^j = l_3))$$

The size of a dependency formula depends on the number of potentially dependent instructions and not on the transitions.

Implementation. We implemented PBMC using the above encoding in the Python language. The prototype is based on the Z3 SMT solver and makes use of its Python API [Z3]. We developed a simplified language that provides basic programming constructs such as assignments, if statements and while loops. The tool supports arrays in addition to boolean and integer variables through the efficient implementation of their respective theories in Z3. Every program contains a header with declarations of variables, the number of processes in the program, an optional initial state assignment and a list of properties to be verified. The body of the program lists the instructions of every process separately in the style of the example shown in Figure 13a.

We now explain the workflow of PBMC. First, the program is parsed and per instruction read/write summaries are created. For instructions accessing an integer or boolean variables the read/write sets are the same for every matching transition. In the case of instructions involving arrays, we also take note of the accessed index. The dependencies are then inferred based on the gathered summaries. For every two instructions l_1, l_2, we add a dependency for the corresponding transitions if l_1 writes to a variable v that l_2 reads from. If v is an array, we add the condition that the indices are equal. Next, the tool translates the parsed program into a Z3 formula as previously shown. Subsequently, the found dependencies are used to construct the projection constraints which are added to the formula along with the negated property to be checked and the optional initial state formula. Then, the solver is called to check the satisfiability of the whole formula. Finally, the output of the solver is interpreted and a counter example path, if existing, is reconstructed. The returned path is a projection that leads to a state where the property is violated. The

tool can be started with parameters to set up the length of paths to be considered and whether projection should be applied.

4.6 EXPERIMENTS AND EVALUATION

In this section, we present preliminary experiments and an evaluation of PBMC. We challenge our approach by choosing four benchmarks where static program slicing would return a mere copy of the program, and therefore be ineffective, to demonstrate the potential of using projections in program verification.

Next, we present the used benchmarks:

- **`Litmus Tests (Litmus):`** In our first benchmark, we generate random instructions accessing shared and local variables. The property we check in this example is whether variables assume certain values. For this case, we use five configurations ranging from 4 to 8 processes.

- **`Indexer:`** Our second benchmark is the indexer program taken from [FG05], where a shared hash table is accessed concurrently by different processes. Every process attempts to insert data into a location of the hash table. If it is already occupied, the process calculates a new hash value and retries again. This operation is repeated until an empty location is found. In the indexer program, dependencies between variable accesses result from writing to and reading from the same hash table location. The property we consider for this example is whether a hash value collisions can occur, which is known to be false for the configurations we consider in our setup [GFY+07]. We use two configurations with 2 and 4 processes.

- **`File System (FSys):`** This example was also adapted from [FG05]. In this benchmark, files are associated with inode data structures which point to memory locations where information about files are stored. For every memory location there is a busy bit indicating whether it has been allocated to an inode. Each inode and busy bit is guarded by a distinct lock to avoid race conditions. When a process picks an inode and no memory was yet allocated for it, it tries to allocate a free memory location. Here dependencies are hard to detect statically because it is not clear in advance in which order inodes will be assigned by the processes. We check for buffer overflow errors in this benchmark and use one configuration which consist of 5 processes.

- **`Dining Philosophers (DPhil):`** We implemented the dining philosophers algorithm in our prototype language. The version we use is deadlock and livelock free. While every philosopher

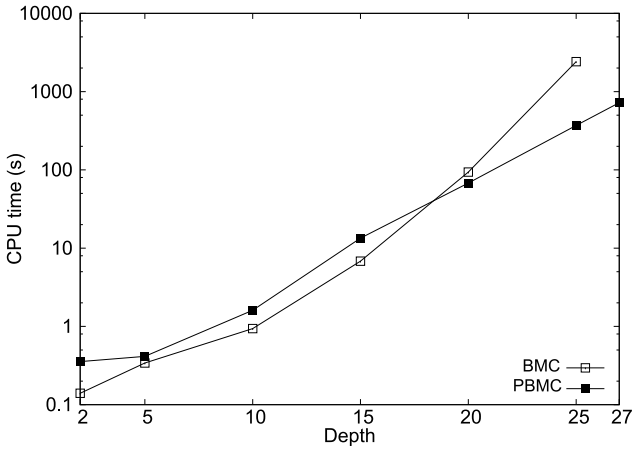


Figure 14: Comparing the total verification time of BMC and PBMC with different path depths for `Litmus` 5 and an unsatisfiable property.

(process) maintains a local state, they share an array of chopsticks. To check for the availability of chopsticks, philosophers access the shared array. For simplicity, since collisions can only occur between neighboring processes, we ignore dependencies involving non-neighboring processes. A mutex is used to guarantee atomicity of operations on the shared array elements. The property we are interested in is whether two neighbor philosophers can be eating at the same time. To challenge our tool we inject a bug in the program and evaluate its capacity of finding counter example paths. The injected bug misuses the shared mutex and thus violates the mutual exclusion property. We set the unrolling depth large enough such that the counter example can be found. For this example we use seven different configurations ranging from 5 to 15 processes.

For every example we use two setups: BMC and PBMC. In general, we observed a trade-off between the complexity of the generated constraints and the amount of reduction achieved during the actual solving. In Figure 14, we show the amount of time spent by PBMC and BMC to verify an unsatisfiable property for the litmus test example with 5 processes with different path depths. The fluctuations of the performance are explained by the fact that SMT solvers make extensive use of heuristics to explore the search space. For small path depth values, the overhead of creating the constraints and handling them by the SMT solver outperforms the reduction that is achieved by using PBMC. Only after reaching a threshold path depth of 20, we observed a clear improvement over BMC. Since the creation of the constraints can be done separately, one can efficiently reduce that threshold by parallelizing the constraint generation process. Moreover, after reaching depth 27 BMC runs out of memory (after two hours) while PBMC finishes the verification within approximately 13 minutes.

Configuration		CPU Time (s)				time red.
Name	Depth	BMC		PBMC		
		Solv.	Total	Solv.	Total	
Indexer 2	10	19.531	20.116	5.088	**5.773**	71.30 %
Indexer 4	15	15 979.298	15 981.531	5881.442	**5884.755**	63.18 %
FSys 5	30	37.058	**47.376**	1.825	58.139	—
FSys 5	60	206.297	**227.427**	14.879	227.942	—
FSys 5	70	627.547	651.991	40.469	**325.797**	50 %
FSys 5	90	949.518	981.573	10.478	**467.558**	52.37 %

Table 6: Comparison of BMC and PBMC for unsatisfiable configurations.

Configuration		CPU Time (s)				time red.
Name	Depth	BMC		PBMC		
		Solv.	Total	Solv.	Total	
DPhil 5	10	5.921	**8.830**	7.671	12.665	—
DPhil 7	10	14.391	**19.162**	14.945	22.585	—
DPhil 10	10	51.229	59.460	39.607	**52.139**	12.31 %
DPhil 12	10	77.121	88.028	66.544	**82.618**	6.14 %
DPhil 15	10	219.824	235.581	182.689	**204.752**	13.08 %
Litmus 4	20	10.649	**11.348**	13.988	15.819	—
Litmus 5	20	605.336	**606.475**	654.573	657.102	—
Litmus 6	20	3888.401	3889.363	908.573	**911.550**	76.56 %
Litmus 7	20	2611.024	2612.562	349.708	**353.544**	86.46 %
Litmus 8	20	>2 h	>2 h	59.738	**64.031**	**Inf.**

Table 7: Comparison of BMC and PBMC satisfiable configurations.

In Tables 6 and 7, we measure the improvement brought by our approach over BMC in terms of CPU time in different configurations. In the name column, we append the number of processes to the name of the used benchmark. For every experiment we specify the unrolling depth used. We report configurations where the formula is satisfiable and unsatisfiable separately in Table 7 and Table 6, respectively. The solving time column shows the amount of time spent by the solver to return an answer excluding the initial analysis and formula building steps. On the other hand, total time includes all the steps. In the reduction column, we give the reduction percentage, in terms of total time, of PBMC in comparison with BMC. From comparing the total and solving times in the table, one can see that as the program complexity increases, the time required for the two initial steps in PBMC's workflow becomes insignificant. This means that for small configurations PBMC brings no improvement in the performance, as the total time is dominated by the time spent on analyzing the program and constructing the formula. On the other hand, PBMC clearly outperforms BMC for larger configurations due to substantial reductions in solving time which becomes more significant. In summary, the results

in the table confirms the global trend that was shown in Figure 14. The relatively small reduction in the dining philosopher example can be explained by the extensive use of the globally shared mutex. In that example, all the transitions depend on the ones manipulating the mutex. This results in a large number of dependencies involving all the variables, including those in the property. In general, the larger the setting is with fewer dependencies involving the variables in the property of interest, the better is the reduction of PBMC.

4.7 CONCLUSION

We have presented projections, an enhanced dynamic slicing notion that can be combined with BMC and proved its correctness. Also, we have implemented PBMC, a bounded model checker that incorporates projections using a novel BMC encoding. By augmenting the BMC formula with projections, PBMC restrains the search space of the model checker and significantly improves on the efficiency over traditional BMC. Our evaluation with examples of concurrent programs has shown major reductions in terms of verification time compared to traditional BMC encoding, even in cases where static slicing proves ineffective.

5

TRACE SANITIZER – ELIMINATING EFFECTS OF NON-DETERMINISM ON EXECUTION TRACES

In Chapters 3 and 4, we explored how the runtime of model checking, a fault removal mechanism, can be shortened. We now come to our last contribution dealing with error propagation analysis of multi-threaded programs. The non-deterministic behavior of a multi-threaded programs constitutes a major challenge for error propagation analysis (EPA). For such programs there can be deviations even across repeated executions with the exact same input. Therefore, observed deviations between an execution and one where the program has been injected with a fault can not be invariably attributed to the activation of the fault. In this chapter, we propose a sound approach to EPA for a subclass of multi-threaded programs and provide an automated method to identify such programs. The contributions presented in this chapter are based on the material in [SWS+19].

In the first section we provide an overview of the contribution. Section 5.2 discusses related work and Section 5.3 illustrates the problems arising from non-determinism in EPA. In Section 5.4 we present our approach, Trace Sanitizer, before discussing our evaluation results in Section 5.5.

5.1 OVERVIEW

To maximize resource utilization and, thereby, system throughput, modern computer systems relax the determinism of program executions. A prominent example are preemptive CPU schedulers that dynamically assign processors to executable programs or revoke such assignments at any point of the programs' executions. Similarly, dynamic memory allocators, which assign memory to a program upon request, have the freedom to decide at which memory address the requested memory region is located, which enables the operating system to avoid collisions, minimize fragmentation, and enhance security via address space layout randomization [BDS03].

However, relaxing execution determinism has adverse effects on dynamic program analyses, such as *Error propagation analysis* (EPA) [Voa97; HJS01]. EPA analyzes how software bugs affect program control and data flow at run time, which is useful for error detector

placement [HJS02; CSW+17] and robustness testing [NWC+18]. For this purpose, programs are mutated similarly to mutation testing, but commonly using different mutation operators and mutant selection criteria. To determine the effects of the introduced bugs on program execution, EPA compares bug-affected (faulty run) against bug-free (golden run) *execution traces*, i.e., records of which program instructions have been executed in which order. If execution determinism is relaxed, instructions from different threads can appear in different orders and referenced memory addresses may differ. For EPA, such deviations between the bug-free and fault injected trace due to relaxed execution determinism constitute false positives, because they do not indicate the effects of faults.

Deterministic replay techniques [CZG+15] can eliminate deviations due to non-determinism across program executions. However, deterministic replay potentially renders EPA results invalid, as the comparison of executions in EPA is not across identical copies of a program, but across an original and a mutated version. For example, if CPU schedules are affected due to time-intensive operations introduced by the mutation, enforcing the original CPU schedule can lead to false conclusions on error propagation.

In this chapter, we propose a novel technique to perform EPA in the presence of execution non-determinism, without resorting to deterministic replay-like techniques. By identifying memory objects and replacing their concrete addresses by symbols in the traces, our algorithm canonicalizes the memory addresses. Addressing scheduling non-determinism is significantly more difficult to handle in the general case, because of possible data dependencies[1] between threads. The existence of data-dependent instructions between threads implies that the non-deterministic scheduling decisions directly impact the data values and instructions seen by each thread. That is, the order in which these instructions are observed allows deviations in the values and instructions in the trace. To deal with such traces, an EPA would require a potentially exponential number of golden runs, each possibly observing deviating data values and instructions due to different orders of data-dependent instructions.

We therefore focus on an important class of programs that we term *pseudo-deterministic*, i.e., that are (1) supposed to exhibit identical externally observable behavior across repeated executions and (2) not affected by non-deterministic external functions.

The first condition excludes both programs with races and programs that are intentionally non-deterministic. For intentionally non-deterministic programs, reference executions generally cannot serve as an oracle. We do not consider this a limitation of our technique, as no differential testing technique (including EPA) is suited for these types of programs. Races are a more problematic case, as they can result in

1 Concurrent accesses to a shared memory object, out of which at least one is writing.

non-deterministic behavior of intentionally deterministic programs, i.e., it is difficult to know upfront if a program meets our postulated condition. To determine whether a program meets this condition or not, we introduce an automated test for data races and order violations based on maximal causality models [HMR14] derived from a reference execution.

The second condition excludes programs that deliberately make use of non-deterministic external libraries, such as random number generators, because these pose a similar problem as intentionally non-deterministic programs. Due to the first condition, the second condition only puts an additional constraint on externally deterministic programs, i.e., by the second condition we exclude programs that process non-deterministic data, but for which this non-deterministic data has no effect on the program's externally observable behavior. While we have not seen a real-world program with these characteristics, a corresponding check could be implemented using a white-/black-listing mechanism for external libraries and system functions.

Prior work on performing EPA for non-deterministic traces either skips the non-deterministic parts of the trace [LM12], or uses statistical properties and likely invariants to capture the non-determinism [LJ09; CWS+17]. The former techniques may miss error propagation in important parts of the execution. The latter techniques may classify deviations from the invariants or statistical measures as errors, though these are legitimate behaviors, and are hence not sound. *To the best of our knowledge, we are the first technique to perform EPA for (a class of) non-deterministic programs that is sound, covers error propagation in non-deterministic parts of the execution, and does not require programmer support or annotations.*

Contributions. We present a novel trace sanitizing approach that deals with the non-determinism due to dynamic memory allocations and non-deterministic scheduling in pseudo-deterministic programs. In summary, we make the following contributions:

- Identify a class of multi-threaded programs, termed *pseudo-deterministic programs*, for which EPA can be sound despite relaxed execution determinism. To support the decision of whether a program is pseudo-deterministic and our technique yields sound results, we develop a novel reversibility check based on the maximal causality model to reliably identify such programs.

- Introduce a novel trace sanitizing approach for *pseudo-deterministic* multi-threaded programs. The presented approach soundly handles non-determinism due to dynamic memory allocation and non-deterministic scheduling.

- Implement our trace sanitizing algorithm in Trace Sanitizer, a trace comparison tool for EPA of multi-threaded programs.

- Evaluate the effectiveness of Trace Sanitizer in EPA based on a set of five widely used benchmarks and show that it successfully reduces the rate of false positives to 0 %, achieves a high fault coverage and reasonable performance overhead.

5.2 RELATED WORK

We classify related work into two broad categories.

Deterministic Execution. The effects of non-determinism due to multi-threading can be mitigated through the use of deterministic execution. Examples of this approach are Dthreads [LCB11] and Deterministic Parallel Java (DPJ) [BJVD+09]. These approaches work by constraining either the set of possible interleavings the OS scheduler is allowed to interpose on the program, or the set of possible programming language constructs that the developer is allowed to use. The former imposes performance overheads as the scheduler has less flexibility in ordering the program's threads to optimize for performance, while the latter imposes a burden on the programmer as they need to ensure their program is free of the problematic constructs (this includes third-party libraries used by the program).

Error Propagation Analysis (EPA). As mentioned earlier, EPA has been traditionally performed by comparing the faulty execution to a golden run (i.e., fault-free execution) of the program [CHR98; HJS02]. Most papers in this area assume that the golden run is deterministic and hence perform a simple line-by-line comparison with the golden run [LJ09]. Unfortunately, this is not the case for most multi-threaded programs. There have been three approaches that have attempted to address the issue of non-deterministic golden traces for EPA. First, DeLemos et al. [LM12] use biological sequence alignment algorithms to compare non-deterministic golden traces with faulty executions, effectively skipping the non-deterministic sections of the trace. An implicit assumption made in this technique is that most parts of the trace are deterministic, and hence skipping the non-deterministic portions is acceptable. Unfortunately, this need not be the case for multi-threaded programs as the OS scheduler has considerable freedom to vary the thread interleaving and memory ordering from one execution to another. Second, Leeke et. al. [LJ09] have attempted to characterize a golden run using statistical techniques such as clustering, and perform a coarse-grained comparison of the faulty run with the golden run with reference to these statistical characteristics. Only if there is a significant deviation in the characteristics do they consider it as an erroneous execution. However, their approach requires significant manual intervention to annotate the clusters, and also requires that the system's outputs conform to well-known statistical distributions. Further, they may not detect subtle errors that violate the event orderings of the program unless the errors result in significant deviations

from the characteristics. Finally, in recent work, Chan et. al. [CWS+17] use dynamic invariants for characterizing a non-deterministic golden run, and consider any execution that results in a violation of the invariants as an erroneous execution. This approach is unsound, as the invariants are only *likely* invariants extracted using Daikon [EPG+07]. Further, the approach is also incomplete as it may miss errors that do not violate the invariants, due to the invariants being incomplete or too permissive in their constraints.

Unlike the above two approaches highlighting the state of the art efforts, our goal is to develop an approach for performing EPA in the presence of non-determinism due to multi-threading in programs that is sound. Further, we do not attempt to constrain the set of execution orderings imposed by the OS scheduler, nor do we constrain the language features used by the programmer. Therefore, our approach also incurs lower performance overheads and requires no effort on the programmer's part.

5.3 TRACE EQUIVALENCE AND EXECUTION NON-DETERMINISM EFFECTS

While relaxing execution determinism for performance does not affect the correctness of a program execution, it may affect the execution trace recorded from that execution. If this is the case, a direct comparison of such non-deterministic traces for EPA leads to false positives. To illustrate the problem, we present a small example of a multi-threaded program in Listings 4 and 5 to illustrate the effects of memory allocation and thread scheduling non-determinism on execution traces and their comparability.

The example program in Listing 4 is a typical example of MapReduce-like programs, an important class of programs that fulfills our first definition criterion of pseudo-determinism. It defines a global array `arr` (Line 3) to store the data to be processed, initializes the content of that array (Lines 17 and 18) and then spawns two threads (Lines 20 and 21) that independently operate on different partitions of the data. The initial thread waits for the two worker threads to return (Lines 22 and 23) before it aggregates the results from their operations by printing the sum of the array elements (Line 24).

Listing 5 shows two shortened execution traces recorded from repeated executions of that program. The traces have been recorded using the EPA framework LLFI [TP13; APB14] and contain one line for each executed instruction of the program's LLVM intermediate representation (IR). The line starts with the index of the instruction in the trace. The second number is a (simplified) ID of the executing thread, followed by the instruction's name, its return value and its operand values.

```
1  #include<stdio.h>
2  #include <pthread.h>
3  int arr[2];
4  void *inc(void* arg)
5  {
6    arr[0]++;
7    pthread_exit(NULL);
8  }
9  void *dec(void* arg)
10 {
11   arr[1]--;
12   pthread_exit(NULL);
13 }
14 int main(int argc, char **argv)
15 {
16   pthread_t id1, id2;
17   arr[0] = 3;
18   arr[1] = 6;
19
20   pthread_create(&id1, NULL, inc, NULL);
21   pthread_create(&id2, NULL, dec, NULL);
22   pthread_join(id1, 0);
23   pthread_join(id2, 0);
24   printf("Result: %d\n", arr[0]+arr[1]);
25   return 0;
26 }
```

Listing 4: Example multi-threaded program.

Despite being functionally identical, an EPA on the two traces would identify them as deviating because of differing memory addresses, i.e., any number with more than one digit in Listing 5, and would proceed to analyze cause-effect chains across these deviations. Moreover, the different interleaving of instructions from different threads causes EPA to falsely identify deviations between the two execution traces. For instance, while the instructions from thread 1 and thread 2 are interleaved in the first traces, they are executed in groups in the second trace. Note that while the threads share global memory locations (a source of dependency), the accesses are never concurrent. For instance in Listing 4, although the main thread and the first thread access the first slot in the array `arr` at Lines 6 and 24, there can be no other execution of the program where Line 24 is executed before Line 6 due to the explicit synchronization call `pthread_join` (Line 22).

The goal of Trace Sanitizer is to transform these traces in a way that preserves *functionally relevant* deviations, e.g., deviating variable values, and eliminates *functionally irrelevant* deviations, e.g., deviating addresses of memory objects representing these variables. Furthermore, Trace Sanitizer leverages the explicit synchronization in multi-threaded programs to simplify the comparison in EPA.

In summary, Trace Sanitizer addresses two sources of execution non-determinism that cause spurious trace deviations.

1. **Non-deterministic memory allocations:** To keep code portable, programs should not make assumptions on memory layout and leave memory management entirely to the operating system. As

```
0  0 call-main 0 1 7ffcfe3287e8
...
1  0 alloca 7ffcfe3282e8 8
2  0 alloca 7ffcfe3282e0 8
...
3  0 store 3 603d74
4  0 store 6 603d78
5  0 call-pthread_create 0 7ffcfe3282e8 0 400ae0 0
6  0 call-pthread_create 0 7ffcfe3282e0 0 4012c0 0
7  1 call-inc 0
8  1 alloca 7f0ccbc55d58 1
9 0 load 7f0ccbc56700 7ffcfe3282e8
10 1 alloca 7f0ccbc55d50 8
11 1 store 0 7f0ccbc55d50
12 1 load 3 603d74
13 2 call-dec 0
14 2 alloca 7f0ccb454d58 8
15 1 store 4 603d74
16 2 alloca 7f0ccb454d50 1
17 2 store 0 7f0ccb454d50
18 2 load 6 603d78
19 0 call-pthread_join 0 7f0ccbc56700 0
20 0 load 7f0ccb455700 7ffcfe3282e0
21 2 store 5 603d78
22 0 call-pthread_join 0 7f0ccb455700 0
...
```

(a) Execution trace 1.

```
0  0 call-main 0 1 7ffda8e0e598
...
1  0 alloca 7ffda8e0e098 8
2  0 alloca 7ffda8e0e090 8
...
3  0 store 3 603d74
4  0 store 6 603d78
5  0 call-pthread_create 0 7ffda8e0e098 0 400ae0 0
6  0 call-pthread_create 0 7ffda8e0e090 0 4012c0 0
7  0 load 7fd5571d9700 7ffda8e0e098
8  1 call-inc 0 0
9  1 alloca 7fd5571d8d58 1
10 1 alloca 7fd5571d8d50 1
11 1 store 0 0 7fd5571d8d50
12 1 load 3 603d74
13 1 store 4 603d74
14 2 call-dec 0
15 2 alloca 7fd5569d7d58 8
16 2 alloca 7fd5569d7d50 8
17 2 store 0 7fd5569d7d50
18 2 load 6 603d78
19 2 store 5 603d78
20 0 call-pthread_join 0 7fd5571d9700 0
21 0 load 7fd5569d8700 7ffda8e0e090
22 0 call-pthread_join 0 7fd5569d8700 0
...
```

(b) Execution trace 2.

Listing 5: Execution traces from two repeated executions of the program in Listing 4

a consequence, the addresses of memory objects that programs operate on should be irrelevant to the program's functionality and should not distort execution trace comparisons for EPA or any other analysis reasoning about the program's functionality.

2. **Non-deterministic thread scheduling:** To maximize CPU utilization and thereby improve throughput, the CPU scheduler may suspend threads that execute blocking instructions, e.g., when waiting for I/O or lock access. The decision of which thread is executed after some other thread has been suspended is dynamically made by the CPU scheduler at run time and may differ across repeated program executions depending on system load and other factors. As a result, the sequence of instructions in the execution trace can deviate across repeated executions. If a program is implemented in a *thread safe* manner, these deviations do not affect the program's functionality and should not affect trace comparisons. A deviation in the order of instructions in an execution trace does not necessarily result in non-deterministic values read or written by the program. This holds especially for programs that implement the map-reduce paradigm where a workload is distributed over a set of worker threads that do not interact with each other and only report back to a master thread after the work is done. In this case, re-executing the program might result in a different interleaving of instructions but still lead to the same effects on the program's data. We show that for such programs a single execution trace is sufficient to achieve a 0% rate of false positives in EPA for multi-threaded programs.

5.4 SANITIZING ALGORITHMS

In this section we present our novel approach to address non-deterministic memory allocation and thread scheduling in EPA. The core idea behind our approach is to leverage the structure of a class of programs to apply two trace sanitizing algorithms each dealing with one of the mentioned sources of non-determinism. We introduce the notion of pseudo-deterministic traces and describe a corresponding automated reversibility check. Finally, we describe Trace Sanitizer, our prototype implementation, and show how it compares traces for sound EPA. To the best of our knowledge, Trace Sanitizer is the first EPA approach to achieve soundness for multi-threaded programs.

5.4.1 *Workflow of Trace Sanitizer*

Figure 15 gives an overview of how Trace Sanitizer achieves sound execution trace comparisons for EPA in the presence of memory and scheduling non-determinism. To obtain execution traces to compare, we first instrument the program to log the executed instructions (step ①) and generate a trace by running the instrumented program ②. These first two steps are fundamental building blocks of EPA and we can reuse the existing implementation of LLFI EPA tool [TP13], which

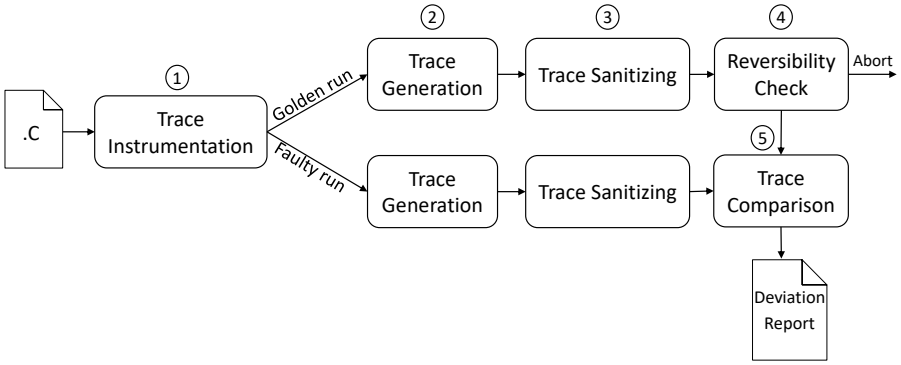


Figure 15: Overview of Trace Sanitizer.

we only slightly modify to deal with multi-threaded programs and include more data in the traces.

Next, we run our trace sanitizing algorithms on the generated trace in step ③. Following that, we run the reversibility check, to verify whether the generated trace satisfies the pseudo-deterministic condition. If the condition holds, the comparison of its traces is guaranteed to be free from false positives induced by scheduling non-determinism. In case it does not satisfy it, we abort the process. To perform EPA, we inject a fault into the instrumented program, run steps ② - ③ again to generate a faulty execution trace, and compare that trace against the fault-free trace (step ⑤) to identify how the program execution has been affected by the injected fault.

In the rest of this section, we introduce our notation and the pseudo-deterministic condition in Section 5.4.2. Then, we discuss the details of the sanitizing algorithms, the reversibility check and describe our implementation of Trace Sanitizer in Section 5.4.3.

5.4.2 *System Model*

We adopt a general and simple model to describe execution traces of a multi-threaded program. An execution trace is a sequence of events $\sigma = e_1, e_2, ..., e_n$. Every event can be directly mapped to an executed instruction such as one spawning a new threads or synchronizing with other threads. For the sake of simplicity, we focus on read and write events. We write $e \in \sigma$ for any event that has been executed by σ. To refer to the total order of events incurred by a trace σ, we write $e_i \prec_\sigma e_j$ if $i < j$ and $e_i, e_j \in \sigma$. We refer to the thread that executed an event e as $Tid(e) \in T$ where T is the set of threads that are spawned by the program during execution.

We define a dependency binary relation D between events based on the memory objects they access. Two events e and e' are said to be dependent if they both access the same object o and at least one of them is a write event. In that case, we write $(e, e') \in D$. We write

D_{tr} to refer to the transitive closure of D. That is, if $(e_i, e_j) \in D_{tr}$ and $(e_j, e_k) \in D_{tr}$ then $(e_i, e_k) \in D_{tr}$, and if $(e_i, e_k) \in D$ then $(e_i, e_k) \in D_{tr}$.

For our sanitizing algorithms to be sound and enable a false positive free EPA, the considered execution traces need to satisfy the pseudo-deterministic condition.

Definition 5 (pseudo-deterministic traces)**.** *A trace $\sigma = e_1, e_2, ..., e_n$ is said to satisfy the pseudo-deterministic condition if and only if*

1. *for every event $e \in \sigma$, the next event executed by $Tid(e)$ and the value it reads or writes are fully specified by the events $e' \prec_\sigma e$ such that $(e', e) \in D_{tr}$ or $Tid(e) = Tid(e')$(local determinism), and*

2. *for every two dependent events $(e_i, e_j) \in D$ such that $e_i \prec_\sigma e_j$, there is no other feasible permutation of the trace σ' where $e_j \prec_{\sigma'} e_i$ (reversibility).*

The local determinism condition excludes programs with inherent non-deterministic behavior. The nature of an event (control flow) and the value it reads/writes (data flow) is solely determined by the events it depends on or events executed by the same thread. Threads in a multi-threaded program acts according to the data they read. Intuitively, the next instruction to be executed by each thread and how it modifies the program's data is fully specified by the values it has read and its program counter position. For instance, the value generated by invoking a random number generator is neither fully specified by the events it depends on nor the events executed by the same thread. We refer to the subsequence of events that fully specify an event as its determining events. For a trace $\sigma = e_1, e_2, \ldots, e_n$ and event $e_i \in \sigma$, its determining events subsequence σ_{e_i} contains only events $e_j \prec_\sigma e_i$ such that $(e_j, e_i) \in D_{tr}$ or $Tid(e_j) = Tid(e_i)$. Thus, given a feasible permutation σ' of σ such that $\sigma_e = \sigma'_e$ for a common event e, the value written/read by e is guaranteed to be the same.

A trace satisfies the reversibility condition if there can be no permutation σ' where two dependent events $(e_i, e_j) \in D$ occur in a reversed order. This implies that $\sigma'_{e_j} \neq \sigma_{e_j}$ since $e_i \notin \sigma'_{e_j}$ and $e_i \in \sigma_{e_j}$. In this case, it is possible for event e_j to read/write a different value (data deviation). Thus, different data values may be observed over repeated executions of the program.

5.4.3 *Algorithms*

In Section 5.4.2 we defined the pseudo-deterministic condition that a program trace needs to satisfy for a sound EPA. It follows from the condition that if a trace satisfies the pseudo-deterministic condition, memory addresses are allocated in the same order by each thread and threads are spawned in the same order by the same parent threads for any feasible permutation of the trace. In this section, we present

Algorithm 4: Memory abstraction algorithm.

input : Execution trace σ
output: Set O of symbolic memory objects

```
1  O ← ∅;
2  foreach e ∈ σ do
3      g_idx ← e.getGlobalIndex();
4      t ← e.getThread();
5      if e.isAllocation() then
6          size ← e.getSize();
7          bAddr ← e.getBaseAddr();
8          val ← [g_idx, _];
9          l_idx ← e.getLocalIndex();
10         o ← (bAddr, t, l_idx, size, val);
11         append o to 0;
12     if e.isDeAlloaction() then
13         o ← getObject(e.getBaseAddr());
14         o.updateValidity(g_idx);
15     if e.isNewScope() then
16         sp[t] ← g_idx;
17     if e.isExitScope() then
18         foreach o ∈ O s.t. o.getThread()
             = t ∧ o.getValidityStart() > sp[t] do
19             o.updateEndValidity(g_idx);
20         sp[t] ← restoreStackPointer();
```

our two sanitizing algorithms. A memory object abstraction algorithm that deals with memory allocation non-determinism by tracking the order in which memory addresses are allocated to achieve a canonical naming where every object is uniquely identified by its position in the sequence of allocated objects. We also present a thread identity abstraction algorithm that handles non-deterministic scheduling by tracking the order in which threads are spawned relatively to their spawning thread and naming them accordingly to achieve consistent IDs across multiple executions. Finally, we describe the reversibility check, the automated test for the pseudo-deterministic condition and our implementation.

Memory Object Abstraction

We start by describing the algorithm for the memory object abstraction. The pseudo-code in Algorithm 4 is a simplified version of that process as implemented by Trace Sanitizer. Given an execution trace σ, the program outputs a set of Symbolic memory objects O that can be used to replace the concrete addresses in the original execution trace. For every event $e \in \sigma$ the algorithm first stores a global index (its position in the trace sequence) as well as the executing thread t (lines 3-4). If e is a memory allocation event, a memory object

$o = (bAddr, t, l\_idx, size, val)$ is created where $bAddr$ is its concrete base address, l_idx is a thread local index that is incremented with every new instruction executed by t, $size$ is the size of the object and val is its initial validity range starting from g_idx (lines 6-10). The object o is then added to O (line 11). The initial validity range of every added object has to be updated according to the scope where it was defined. If e is a memory de-allocation event, e.g., a call to the `free` function, the algorithm updates the validity range of the object $o \in O$ with the same base address (lines 13-14). Note that in this case o must be a heap object, and if it is never de-allocated the default range is still sound as the object becomes valid from its creation until the end of the trace. Memory objects that are defined on the stack are handled separately. If a new scope event for thread t is encountered, for example entering a new function, the current global index is stored in $sp[t]$ for later use (line 16). In case e is an exit scope event, the algorithm updates the validity of all objects that were added after $sp[t]$ and restores the previous stack pointer (lines 18-20). Objects that were added after $sp[t]$ represent the objects that have to be de-allocated because the program is leaving the scope where they were defined. The restored stack pointer is assigned the global index of the last new scope event by thread t so that once that scope is exited, the validity range of the objects defined within it can be accordingly updated.

Once the set of objects O is generated, Trace Sanitizer replaces each reference to a concrete memory address by a corresponding object eliminating trace deviations that are due to memory locations. Concretely, an address is replaced by an object if it lies within its allocated space specified by its base address and size. If a memory address matches more than one memory object, we use the validity range to identify the correct object.

Thread Identity Abstraction

Next, we discuss our thread identity abstraction process that enables the matching of threads in different execution traces. We provide a simplified version in Algorithm 5. The algorithm's goal is to achieve canonical thread IDs such that for every two executions of the same programs with the same input it is guaranteed that the same threads will receive the same id. Given a sequence of events σ, the algorithm builds a mapping function M that maps each thread ID to a canonical ID. The algorithm works by building a thread tree G where the nodes represent the spawned threads and the edges a spawning relation. If a thread $t_1 \in T$ spawns a thread t_2, we add a directed edge (t_1, t_2) between these two nodes (lines 5-7). The next step consists of traversing the tree G starting from the root node such that for every node the children are visited in their order of creation (lines 11-17). The canonical thread IDs are then recursively generated as follows:

Algorithm 5: Thread abstraction algorithm.

input : Execution trace σ
output: A map M of thread IDs to canonical IDs

```
1  M ← ⟨⟩;
2  Q ← ∅;
3  T ← σ.getAllThreads();
4  G ← (T, ∅);
5  foreach t ∈ T do
6  |  foreach t' ∈ t.getSpawnedThreads() do
7  |  |  append (t, t') to G;
8  t_c ← G.getRootNode();
9  M[t_c] ← "T_0";
10 push t_c to Q;
11 while Q ≠ ∅ do
12 |  t_c ← Q.pop();
13 |  i ← 0;
   |  // Get the spawned threads in their order of creation.
14 |  foreach t ∈ t_c.getOrderedChildren() do
15 |  |  M[t] ← M[t_c]"_"i;
16 |  |  push t to Q;
17 |  |  i ← i + 1;
```

- The root node is the main thread and is assigned the ID `"T_0"` (lines 9-10).
- Every node is assigned an ID that consists of its parent node's ID as a prefix and its position in the list of children (lines 15-17).

After running the thread identity abstraction algorithm, we use the mapping function M to rename the threads by their canonical names, and replace every reference to a thread ID in the execution trace. If a program satisfies the pseudo-deterministic condition, it is guaranteed that the canonical thread IDs match exactly, enabling the matching of all spawned threads across multiple execution traces in the trace comparison phase.

Reversibility Check

We developed an automated reversibility check to test whether an execution trace σ satisfies the pseudo-deterministic condition. The automated test focuses on the reversibility condition from Definition 5. While we manually checked the local determinism condition, the process can easily be automated using a black-listing approach where black listed calls to external libraries or system functions that are non-deterministic are systematically reported. The reversibility check is based on the maximal causality technique which has been used for race detection [HMR14]. A maximal causality formula encodes all permutations of a given trace that are guaranteed to be feasible, i.e.,

that are valid executions of the same program with the same input. While it is not guaranteed to encompass all the feasible permutations, it contains the maximal number of permutations that can be inferred from a single trace of a program [ŞCR12]. Our reversibility check uses a modified version of the maximal causality formula that omits the additional constraints that ensure that only valid executions are encoded. In our technique, we build a reversibility formula Φ_σ such that every solution of the set of constraints in the formula encodes an execution that is not necessary a valid execution. We will discuss later why this is still sufficient for a sound and complete check. The formula defines integer variables x_i for every event $e_i \in \sigma$. The variables are then constrained in their order such that only the set of permutations that are guaranteed to be feasible are allowed. For instance, if e_i is an event spawning a new thread, the formula adds a constraint $x_i < x_j$ for the first event e_j executed by the spawned thread. To guarantee the sequentiality of every thread t, a condition $x_i < x_j$ is added for every successive event by t. To prevent an overlap of two critical sections in the trace that are guarded by the same mutex, the formula adds a constraint $x_i < x'_j \wedge x_j < x'_i$ where e_i and e'_i are two mutex acquiring events and e_j and e'_j are the two corresponding mutex release events. For more details about the formula please refer to [HMR14]. Finally, we add additional constraints to encode our pseudo-deterministic condition:

$$R := \bigvee_{(e_i,e_j)\in D \wedge e_i \prec_\sigma e_j} x_j \leq x_i$$

Intuitively, the constraint encodes the fact that *any* two dependent events in the trace occur in a reversed order.

In the last step, we check the satisfiability of formula $\Phi_\sigma \wedge R$ using an SMT solver. If the solver does not return a solution, we have a proof that there cannot be any two dependent events that can occur in a reversed order and therefore the trace satisfies the pseudo-deterministic condition. If, however, the formula has been proven satisfiable, the solver returns a solution that encodes an execution trace where at least two dependent events are reversed. In this case, the trace does not satisfy the pseudo-deterministic condition.

Correctness. In our check, we omit the constraints that reduce the set of allowed permutations to only those that are guaranteed to be feasible (i.e., the read conditions in [HMR14]). Furthermore, the maximal causality model, upon which the reversibility formula is based, does not cover all feasible permutations but only the maximal number of permutations that can be inferred from a single execution [ŞCR12] since it does not include executions that take new control flow paths. These two limitations, however, don't affect the soundness nor the completeness of the reversibility check.

The soundness of the check can be inferred from the fact that our reversibility formula can only contain unfeasible permutations if the

execution is reversible. Let us assume that the check is unsound, i.e., for a trace σ that does not satisfy the reversibility condition, the reversibility formula is wrongly satisfiable. This means that the event order encoded by the reversibility formula describes an unfeasible execution σ' due to the missing read conditions from [HMR14]. Let e' be the first event in σ' that is not feasible and e the last event in the feasible prefix of σ' such that $Tid(e) = Tid(e')$. Because of local determinism we have $\sigma_e \neq \sigma'_e$ since otherwise e' would be executable in σ'. This would mean that σ'_e, and therefore also the feasible prefix of σ', contains at least a set of reversed dependent events. But this contradicts our initial assumption that σ does not satisfy the reversibility condition since the prefix of σ up to event e' is feasible.

Similarly, the completeness of the check follows from the fact that the set of permutations covered by the formula is not complete only if the considered trace is reversible. Let us assume the check is incomplete, i.e., for a trace σ that satisfies the reversibility condition, the reversibility formula is wrongly unsatisfiable. This means that there is a feasible interleaving σ' of execution σ where two dependent events occur in reversed order and that is not covered by the reversibility formula. These two events cannot both be included in σ because otherwise the reversibility formula would be satisfiable. If at least one of the events, e, is not included in σ, its determining events σ'_e must include two events that occur in reversed order and are in σ, assuming that e is the first such an event in σ'. This means, however, that the reversibility formula will be satisfiable, contradicting our initial assumption.

Trace Comparison With an Example

We use the example from Listing 4 to illustrate how the trace comparison process is performed by Trace Sanitizer. Given the execution trace from Listing 6a, the sanitizing algorithms from Algorithms 4 and 5 produce the sanitized trace in Listing 6b, and the memory object set and thread identity mapping shown in Listing 7.

Memory Object Abstraction. Initially, the set of identified memory objects is empty. The algorithm starts by iterating over all events in the execution trace σ. After reaching an allocation instruction (line 1), a new object `o4` is created and added to the set of memory objects O. The object structure contains information about the base address returned by the LLVM-IR `alloca` instruction (`7ffcfe3287e8`), the size of the allocated object (`8`), the thread executing the instruction (`0`), the local index reflecting the position of the instruction in the sequence executed by the thread (`1`) and the initial validity range of the object (`[1,33]`) where `33` is the total number of instructions in the trace.

At the invocation of the `inc` function (line 7), a new scope is created and the stack pointer for thread `1` is updated to the index of the event where the scope was entered7. This value will be used later to update the validity range of every new object created within the new scope.

```
0  0 call-main 0 1 7ffcfe3287e8
...
1  0 alloca 7ffcfe3282e8 8
2  0 alloca 7ffcfe3282e0 8
...
3  0 store 3 603d74
4  0 store 6 603d78
5  0 call-pthread_create 0 7ffcfe3282e8 0 400ae0 0
6  0 call-pthread_create 0 7ffcfe3282e0 0 4012c0 0
7  1 call-inc 0
8  1 alloca 7f0ccbc55d58 1
9 0 load 7f0ccbc56700 7ffcfe3282e8
10 1 alloca 7f0ccbc55d50 8
11 1 store 0 7f0ccbc55d50
12 1 load 3 603d74
13 2 call-dec 0
14 2 alloca 7f0ccb454d58 8
15 1 store 4 603d74
16 2 alloca 7f0ccb454d50 1
17 2 store 0 7f0ccb454d50
18 2 load 6 603d78
19 0 call-pthread_join 0 7f0ccbc56700 0
20 0 load 7f0ccb455700 7ffcfe3282e0
21 2 store 5 603d78
22 0 call-pthread_join 0 7f0ccb455700 0
...
```

(a) Execution trace from Listing 5a

```
0  T_0   call-main 0 1 o0
...
1  T_0   alloca o4
2  T_0   alloca o5
...
3  T_0   store 3 g0
4  T_0   store 6 g0+4
5  T_0   call-pthread_create-u 0 o4 0 400ae0 0
6  T_0   call-pthread_create-u 0 o5 0 4012c0 0
7  T_0_0 call-inc 0
8  T_0_0 alloca o6 1 8
9  T_0   load T_0_0 o4
10 T_0_0 alloca o7 1 8
11 T_0_0 store 0 o7
12 T_0_0 load 3 g0
13 T_0_1 call-dec 0
14 T_0_1 alloca o8 1 8
15 T_0_0 store 4 g0
16 T_0_1 alloca o9 1 8
17 T_0_1 store 0 o9
18 T_0_1 load 6 g0+4
19 T_0   call-pthread_join 0 T_0_0 0
20 T_0   load T_0_1 o5
21 T_0_1 store 5 g0+4
22 T_0   call-pthread_join 0 T_0_1 0
...
```

(b) The sanitized trace.

Listing 6: Trace sanitizing example.

Similarly, at the call to `dec` in line 16, the algorithm updates the stack pointer for thread 2 to 13. At the end of each of the functions, the scope for both `inc` and `dec` ends and the validity range for the objects created within that scope has to be updated. Every object that has been

```
g0 := {ba=603d74, t=_, l_idx=0, s=8, v=[0,33]}
...
o0 := {ba=7ffcfe3287e8, t=T_0,   l_idx=0, s=8, v=[0,33]}
o4 := {ba=7ffcfe3282e8, t=T_0,   l_idx=1, s=8, v=[1,33]}
o5 := {ba=7ffcfe3282e0, t=T_0,   l_idx=2, s=8, v=[2,33]}
o6 := {ba=7f0ccbc55d58, t=T_0_0, l_idx=0, s=8, v=[8,15]}
o7 := {ba=7f0ccbc55d50, t=T_0_0, l_idx=1, s=8, v=[10,15]}
o8 := {ba=7f0ccb454d58, t=T_0_1, l_idx=0, s=8, v=[14,21]}
o9 := {ba=7f0ccb454d50, t=T_0_1, l_idx=1, s=8, v=[16,21]}

M(0) := T_0
M(1) := T_0_0
M(2) := T_0_1
```

Listing 7: The memory object set and thread identity mapping generated by the sanitizing algorithms.

added to O after the value in the stack pointer has to be updated. For instance, the validity range of object `o6`, added at line 8 within `inc`'s scope, is updated be to `[8,15]` where `8` is the index of its allocation instruction in the trace and `15` the end of the scope. The stack pointer also has to be updated to the start of the previous scope but in this case it is not necessary since only one scope has been created by this thread.

Finally, Trace Sanitizer replaces the concrete addresses with the generated objects in every instruction in the trace. In this example there are also global variables that are accessed by the program. For the sake of brevity the Algorithm 4 does not handle global variables. Our implementation, however, handles these variables separately. Before iterating over the instructions, we add a memory object with maximal validity range for each global variable. The object `g0` in Listing 6b represents the global array `arr` defined in the example. Note that accesses to `g0` are not always at the base address. For instance, the access at line 18, occurs on the second element in the array, hence the reference `g0+4` with an `int` type of byte length 4. Encountering an address that has not been explicitly allocated leads to the creation of a new object default size. For example, the argument of the `main` function results in the creation of object `o0` (Listing 7).

Thread Identity Abstraction. The algorithm first fetches the set of threads in the trace `0, 1,` and `2` and adds nodes for these threads to G. We show the resulting spawning tree in Figure 16 (The nodes are renamed by their canonical IDs). Since thread `0` spawns threads `1` and `2`, edges $(0,1)$ and $(0,2)$ are added to G. Next, the algorithm traverses the generated tree G to map concrete thread IDs to deterministically calculated thread IDs.

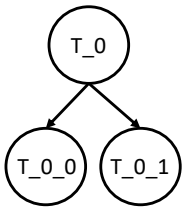


Figure 16

The initial thread `0` is mapped to ID `T_0`. Every time a new node is reached in the tree, its child nodes are traversed in the order of the corresponding threads' creation. For instance, since thread `1` was

created before 2, it will be traversed first. Following the renaming pattern presented in Algorithm 5, thread 1 is mapped to an ID consisting of its parent node ID (T_0) concatenated with an index indicating its creation order 0: T_0_0. Likewise, thread 2 is mapped to ID T_0_1 as shown in Listing 7. Finally, Trace Sanitizer replaces every reference to a thread's ID in the program using the generated map M.

Reversibility Check. To build the reversibility check formula, a unique variable is assigned to every instruction in the trace. The formula consists of two parts. The first part encodes the allowed permutations of the trace by imposing constraints on the order of the formula variables. In addition to the constraints encoding the sequentiality of every thread, we add constraints for inter-thread synchronization. The call `pthread_join` at line 19 in Listing 6a, for instance, forces the invoking thread 0, to wait for the termination of thread 2 before executing the next instruction. In this case, we add a constraint $x' < x$, where e is the variable mapped to the call to `pthread_join` and e' the last instruction executed by thread 2.

To build the second part, we identify all the dependencies between `load` and `store` instructions that read from or write to the same memory location. Consider the two instructions e and e' at lines 4 and 18, respectively. Instruction e' reads from the same memory object g0, with the same offset, that e writes to. Therefore, both instructions are dependent on each other. Since e occurs before e' in the trace, we add the constraint $x' \leq x$ to the reversibility formula to check whether the two dependent instructions can occur in the reverse order. However, the formula does not allow any permutation of the trace where instruction e occurs after e' as that would mean that the thread executing e' would start executing before it has been spawned. Since no such pair of instruction can be found for the trace, the generated reversibility formula cannot be satisfied and the trace will be declared to satisfy the reversibility condition. Note that our check ignores dependencies between instruction from the same thread as these can obviously not be reversed.

Trace Comparison. After running the reversibility check on the sanitized fault-free trace, we can safely compare it against sanitized traces from fault injection experiments with the same program processing the same input. We start by dividing the sanitized traces into subsequences where each sequence contains only instructions belonging to a single thread. Next, we match the sub-sequences belonging to the same thread and compare every instruction. Since the threads are renamed identically in traces from different executions with the same input, the set of created IDs in both traces should match in a comparison of fault-free executions. Furthermore, if the first trace had passed the reversibility check and the second trace is a re-execution of the same program with the same input, then every pair of two such sub-sequences should be identical unless one of the traces is affected

by an injected fault. This is guaranteed by the local determinism property of every thread induced by the pseudo-deterministic condition of the first trace. In other words, if dependent instructions cannot occur in a reversed order, every thread will be created in the same order by the same parent thread, and memory objects will also be allocated in the same order and by the same thread. Therefore, a deviation between both traces implies that the injected fault has been activated in the experiment and its effects on the execution show in the comparison.

Concretely, the comparison algorithm checks whether instructions in both sub-sequences occur in the same order and whether they access the same objects with the same offsets. Applying the sanitizing algorithms on both traces from listing 5 results in pairs of identical sub-sequences for every thread in the trace (`T_0`, `T_0_0` and `T_0_1`) since the first trace satisfies the pseudo-deterministic condition and the second trace is not faulty.

Implementation

We implemented our approach in Trace Sanitizer, a trace comparison tool for the purpose of EPA. Trace Sanitizer consists of two modules: (1) an instrumentation and fault injection module that is implemented as an extension of the LLFI EPA tool [TP13], and (2) a sanitization and trace comparison module. The first module adds more logging information in the trace generation process and thread safety to the LLFI tool in order to deal with concurrency in multi-threaded programs. The instrumentation and fault injection parts are performed at the level of LLVM-IR, the intermediate level representation of the LLVM compiler infrastructure. We implemented the second module in the Rust programming language and used the Z3 SMT solver [Z3; DMB08] for the reversibility check. Since the reversibility formula we generate is in the standard SMT-LIB format [BST+10] Trace Sanitizer can use a large variety of existing solvers.

5.5 EVALUATION

The goal of our evaluation is to show that the Trace Sanitizer approach eliminates *all* false positives in EPA that are due to benign execution non-determinism, and to measure the performance overhead this reduction requires. We structure our discussion along the following research questions, targeting five C/C++ programs, four of which are taken from the PARSEC [BKS+08] and Phoenix benchmarks [Pho], that satisfy the pseudo-deterministic condition.

RQ 1 What are the false positive rates resulting from non-determinism in dynamic memory allocations with and without Trace Sanitizer?

Table 8: Overview of the benchmark programs. SLOC[2] reports the source lines of code, #Th the sum of spawned threads in one run, #Inst the number of executed instructions in one run, and Mem-Sound/Sched-Sound whether false positives occurred due to memory/CPU non-determinism (✗) or not (✓).

Program	SLOC	#Th	#Inst	Mem-Sound		Sched-Sound	
				Naïve	TSAN	Naïve	TSAN
`quicksort`	198	72	45 k	✗	✓	✗	✓
`pca`	301	17	89 k	✗	✓	✗	✓
`kmeans`	425	65	44 k	✗	✓	✗	✓
`blackscholes`	393	3	91 k	✗	✓	✗	✓
`swaptions`	1118	4	1.1 M	✗	✓	✗	✓

RQ 2 What are the false positive rates resulting from CPU scheduling non-determinism with and without Trace Sanitizer?

RQ 3 What is the rate of false negatives with Trace Sanitizer?

RQ 4 What is the performance overhead of Trace Sanitizer?

5.5.1 *Target Programs and Execution Environment*

Our experiments target the five C/C++ programs listed in Table 8. The table reports the number of SLOCS of each program along with the total number of instructions and spawned threads. `quicksort` is a parallel implementation of the well-known sorting algorithm. `pca` and `kmeans` are two machine-learning algorithms taken from the Phoenix benchmark suite [Pho]. `kmeans` is an implementation of the Kmeans clustering algorithm and `pca` implements the principle component analysis statistical procedure. Additionally, we used the `blackscholes` and `swaptions` programs from the PARSEC benchmark suite [BKS+08]. The `blackscholes` programs solves the `blackscholes` partial differential equation used in pricing a portfolio of European-style stock options. `swaptions` uses Monte-Carlo simulations to compute `swaptions`, a form of financial derivatives. Each of the five programs satisfies the pseudo-deterministic condition.

We conducted all of the experiments on machines with an Intel Core i7-4790 CPU, 16 GiB of RAM, and a 500 GB SSD running Debian 8.11 with a Linux 3.16 kernel.

5.5.2 *RQ 1: False Positives from Memory Addresses*

Our work is motivated by the observation that non-determinism can lead to false positives if a naïve execution trace comparison is applied

2 generated using David A. Wheeler's 'SLOCCount'.

for EPA. We separately evaluate the effectiveness of Trace Sanitizer in eliminating false positives in EPA that are due to dynamic memory allocation and non-deterministic scheduling. We begin with an evaluation of the impact that dynamic memory allocation non-determinism has on execution traces and how well Trace Sanitizer can deal with those.

To evaluate the impact of dynamic memory allocation non-determinism independently from CPU scheduling effects, we conduct a number of trace comparisons on single-threaded executions of our target programs without any fault injections. For this purpose, we use single-threaded versions of the five programs from Table 8. As we do not inject any faults, any observed deviation across repeated executions of the same program must be a false positive. Moreover, since CPU scheduling cannot cause deviations in single-threaded programs, any observed false positive is likely due to memory non-determinism.

We generate 10 000 fault-free execution traces from the single-threaded programs by performing steps ① and ② in Figure 15 and perform a naïve line-by-line comparison, just as conventional EPA approaches would. We then run Trace Sanitizer's memory abstraction algorithm on the same traces and report the results in the Mem-Sound column of Table 8. The results show that no execution trace is identical to any other from the 10 000 repetitions and that Trace Sanitizer is able to eliminate all of these spurious deviations for all of the benchmarks. Besides demonstrating the effectiveness of our memory abstraction, this result confirms that the second criterion from our definition of the pseudo-deterministic condition in Section 5.1 is satisfied for the chosen program input.

5.5.3 *RQ 2: False Positives from CPU Scheduling*

To assess the effectiveness of Trace Sanitizer to compensate for the effects of non-deterministic CPU scheduling, we recompiled the target programs to use multiple threads. We repeatedly executed these programs without fault injections, sanitized effects of memory allocation non-determinism in their execution traces, and compared the resulting traces of the repeated executions. As no faults are injected in these executions, any deviations between traces constitute false positives. Moreover, as the effects of memory allocation non-determinism are sanitized, any observed deviations must result from CPU scheduling.

We again compare the obtained traces in a line-by-line fashion without Trace Sanitizer, and observe deviations in each comparison. We then run Trace Sanitizer's thread abstraction algorithm on the traces and perform the comparison again. The results shown in the Sched-Sound column of Table 8 demonstrate that Trace Sanitizer is able to fully eliminate false positives resulting from non-deterministic CPU scheduling for pseudo-deterministic programs.

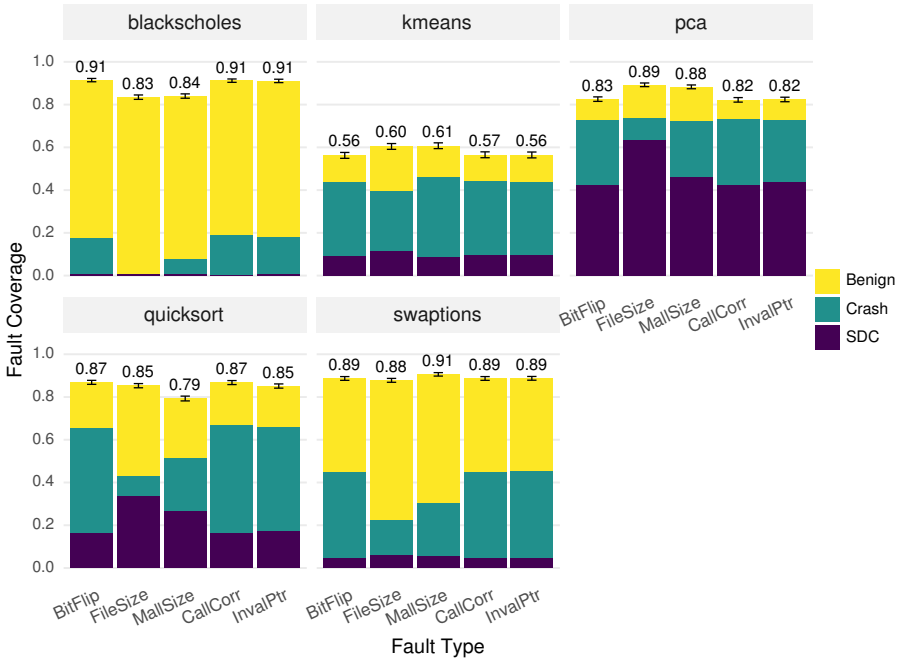


Figure 17: Fault coverage per target and fault type. The reported results are obtained from 5000 runs for each combination of target and fault type. The error bars indicate the 95 % confidence interval.

5.5.4 *RQ 3: False Negatives Introduced by Trace Sanitizer*

As we have shown in the last two subsections, Trace Sanitizer effectively eliminates false positives resulting from memory allocation and non-deterministic scheduling for traces that satisfy the pseudo-deterministic condition. Accurately measuring false negative rates in EPA experiments is challenging, because there is no oracle to distinguish between true and false negatives. If a fault is injected and no effect is observed, it is unclear whether no effect has occurred (true negative) or an effect has occurred and it was missed by the detection mechanism (false negative). Moreover, differential testing using different detection mechanisms is difficult to apply in the case of EPA for multi-threaded programs, because other approaches are not sound, and their false positives would distort the results. Therefore, we base our evaluation of false negatives on a conservative estimate. Assuming that each injected fault leads to error propagation (this may not always hold, e.g. [MA14]), each succeeding trace comparison between an injection and a fault-free run constitutes a false negative. We term the fraction of these succeeding comparisons from all comparisons the *maximal possible false negative rate (MPFNR)*.

To make sure that Trace Sanitizer does not achieve soundness at the cost of an increased false negative rate, we executed a number of fault injection experiments following the steps outlined in Figure 15 and discussed in Section 5.4.1 and assessed the MPFNR. We used the

multi-threaded versions of the benchmark programs from Section 5.5.3 and first generated and sanitized execution traces from one fault-free execution of each program. We then ran our memory and thread abstraction algorithms on the trace and performed the reversibility check to make sure that comparisons against this trace will yield sound results if the same program inputs are used. To obtain traces from faulty executions, we repeated the execution of the program with the same inputs and injected one fault per execution using the LLFI fault injection framework. In total we executed 25 000 such experiments, 5000 fault instances for each of the fault types listed in Table 9 and sanitized each of the resulting execution traces.

Figure 17 shows the *fault coverage* of EPA using Trace Sanitizer, i.e., the fraction of experiments for which the sanitized fault-injection traces differed from the fault-free trace and, thus, indicate error propagation. The MPFNR is the difference between 1 and the reported fault coverage and ranges between 44 % and 9 % depending on the program and fault type. While we cannot tell whether any of the succeeded comparisons was due to the lack of error propagation or due to a false negative of our approach, we can tell if Trace Sanitizer has any obvious blind spots by investigating the false negative rates for experiments that led to program failures (i.e., externally observable deviation from correct behavior). If the program behavior deviates from correct behavior as observed in the fault-free execution, an error *must* have propagated and missing signs of such propagation in the traces would be a false negative. To assess this, we have calculated the MPFNR for different classes of experiment outcomes that are indicated by different colors in Figure 17. A *crash* denotes cases where the program terminated abnormally after a fault was injected, whereas *SDC (silent data corruption)* indicates cases where the program terminated without error indication, but the results of its computations differed from the fault-free case. For both crashes and SDCs, we found the MPFNR to be 0 %. This demonstrates that there were no obvious cases of error propagation that were missed by Trace Sanitizer.

In summary, to the best of our knowledge Trace Sanitizer is the first to achieve a 0 % false positive rate for EPA on multi-threaded programs without increasing the false negative rate for known cases of error propagation (observed crashes and SDCs).

5.5.5 *RQ 4: Trace Sanitizer Overhead*

Achieving a high fault coverage and fully eliminating false positives comes at the cost of (a) running the reversibility check on the golden run to ensure the soundness of the approach, and (b) running the sanitization algorithms on the traces. It is important to note that the reversibility check execution time is a one time cost as the check only needs to be run once on the golden run in EPA. On the other hand,

Table 9: Overview of injected fault types

Fault Type	Short Description
BitFlip	Flips single bits in arbitrary data values.
FileSize	Increases the `size` parameter in `fread` and `fwrite` function calls for file I/O.
MallSize	Decreases the `size` parameter in `malloc` and `calloc` function calls for memory allocation.
CallCorr	Corrupts the first parameter of function calls.
InvalPtr	Corrupts the returned pointers from `malloc` and `calloc` function calls.

Table 10: Performance Results for Trace Sanitizer. Reversibility Check times are reported in minutes and EPA times in seconds. EPA times are median values over 5000 runs.

Program	Rev. Check				EPA	
	#Obj.	#Dep.	Solver [m]	Total [m]	San. [s]	Cmp. [s]
`quicksort`	38	24 650	30.36	30.38	1.57	0.3
`pca`	64	12 126	150.41	150.43	1.29	0.17
`kmeans`	31	13 460	81.93	81.94	0.79	0.13
`blackscholes`	13	2810	0.87	0.99	1.58	0.2
`swaptions`	16	22 630	116.66	144.61	8.57	2.86

running the sanitization algorithms needs to be done for each fault that is injected (typically thousands of times for obtaining statistically significant estimates). We measure the time it takes to run each of these two steps - the results are shown in Table 10.

Reversibility Check. To assess the run time overhead of the reversibility check we performed it on a golden run of each of the benchmarks. We report the total run time along with time taken by the SMT solver, the number or memory objects accessed in the trace, and the inter-thread dependencies on these objects in Table 10. For all programs, the overall time for the reversibility check ranges from approximately 1 min for `blackscholes` to 150 min for `pca` and is strongly dominated by the SMT solver's execution time. For `swaptions`, which is the only program showing a notable difference between these times, building the formula takes considerably longer than for other benchmarks due to the much higher number of instructions in the trace that the algorithm has to go through (Table 8). In addition to the solving time,

the total overhead consists of the time it takes Trace Sanitizer to build the formula, including the identification of data-dependencies in the trace. While the number of dependencies and objects, along with the total number of instructions hint at the size and complexity of the formulas generated by Trace Sanitizer, they do not directly correspond to the measured execution times. `quicksort`, for example, has a higher complexity than `kmeans` in terms of memory objects and dependencies in the traces with a comparable trace size, but takes significantly less time for the check. This indicates that it is difficult to predict the actual solving time and we suspect this to be due to randomness in the SMT solver's search space exploration of a formula.

Trace Sanitizing. Once the golden run has passed the reversibility check, Trace Sanitizer proceeds with the sanitization and comparison of faulty runs. Table 10 shows a break-down of the median time across 5000 experiments that Trace Sanitizer requires to perform these sanitization (column 6) and comparison (column 7) steps. The median time for trace sanitization ranges between 0.79 s and 8.75 s with a median absolute deviation (MAD) of 1.9 s for `swaptions` and less than 0.4 s for the other benchmarks. The trace comparison of a sanitized golden run and a faulty run takes between 0.17 s and 2.86 s with a MAD of 0.2 s for `swaptions` and under 0.02 s for the other benchmarks.

While we cannot directly compare these results to existing approaches due to the strong impact of machine configurations on performance measurements, we can provide an indirect comparison. Because Trace Sanitizer is the only sound tool for EPA trace comparisons, it does not require any manual inspection of the obtained comparison results to check for false positives. Any unsound tool requires these checks. To beat Trace Sanitizer's performance for 5000 injections in the slowest case of `swaptions`, 4400 trace diffs ($5000 \cdot 0.88$, the smallest coverage in Figure 17) would need to be inspected (manually) in less time than $\frac{5000 \cdot 8.57\,\text{s} + 144.61 \cdot 60\,\text{s}}{4400}$, which is less than 12 seconds for a diff across traces with more than a million lines (Table 8). An analogous calculation yields less than 4 seconds for manual inspection of any other benchmark. Such small times are almost impossible to achieve for any realistic program trace, including those in our evaluation. Moreover, the time taken by Trace Sanitizer will become smaller as computing becomes faster, which is not the case for manual inspection.

5.6 CONCLUSION

In this chapter, we introduced a class of multi-threaded programs that we termed pseudo-deterministic and for which EPA can be sound in the presence of non-deterministic memory allocations and CPU scheduling. We have developed a automated technique to determine whether a program belongs to this class as well as a novel trace sanitizing approach that soundly handles non-determinism. We evaluated

our prototype implementation Trace Sanitizer of the approach on five benchmark programs. We find that Trace Sanitizer is able to fully eliminate false positives, achieves a high fault coverage in an EPA study, and provides reasonable performance.

Part IV

CONCLUSION

6

CONCLUSION

Computer systems continue to permeate our lives as we grow more dependent on them. This dependence and the continuous increase in their complexity call for more rigorous methods to justify our trust in them. In fact, many of the existing safety standards recommend the use of formal methods in the development of safety critical systems. In the course of the last 30 years there have been clear trends towards more parallelization and distribution of computer systems. This trend is mainly motivated by the physical limitation on CPU clock speed as well as the necessity of bringing forward collaborative systems on large scales. The aim of this thesis is to improve on existing formal analysis techniques for dependable concurrent systems. Formal analysis techniques can be leveraged to analyze dependable systems and predict their behavior before deployment. Applying these techniques can help us avoid catastrophic accidents such as in the Therac-25 [LT93] or the Ariane 5 [Lio+96] cases by uncovering their triggering causes well in advance.

The aim of the thesis is to answer the three research questions formulated below. Answering these questions lead to four contributions with the common purpose of applying formal analysis techniques in the development process of dependable and concurrent systems. In our first contribution, we dealt with the availability aspect of dependability and proposed a prevention mechanism based on static analysis. Our second and third contributions provided improvements to existing model checked techniques dealing with the safety aspect of the dependability. In the last contribution, we presented a novel error propagation analysis technique EPA for multi-threaded programs to help with the development of dependable systems.

Research Question (RQ1): *Can static analysis techniques assist in designing highly available distributed systems?*

To answer this question we have presented our first contribution *(C1)* in Chapter 2. We explore how static data-flow analysis can be used to improve the availability of single server applications. For this purpose, we propose operation partitioning, a data-flow analysis algorithm that can infer rules under which the workload of a single server can be dis-

tributed across multiple machines without sacrificing consistency. The analysis algorithm is combined with a distributed protocol, Conveyor Belt, that leverages the inferred rules to increase the availability of the system through minimal synchronization. We implemented operation partitioning and Conveyor Belt for Java database applications in a tool that we called Gyro.

Research Question (RQ2): *Can the structure of a specification property be leveraged to improve the efficiency of model checking?*

Answering this question resulted in the contributions *(C2)* and *(C3)* discussed in Chapter 3 and Chapter 4, respectively. Our second contribution is a novel approach to improve explicit stateful model checking. The proposed approach uses a notion of decomposition that, given a specification, distinguishes between relevant and irrelevant portions of the state space. Our decomposition based model checking algorithm then systematically explores only portions of the state space that are relevant to the specification, thus reducing the overall runtime. The decomposition optimization is orthogonal to other approaches such as partial-order reduction techniques and can be combined with them to achieve better results. We implemented decomposition-based model checking in MP-Basset [Mpb], a model checker for message-passing Java program based on NASA's JPF model checker [Jpf].

Our third contribution is an improvement to bounded model checking. We introduce the notion of projections to characterize the set of executions that are relevant to a specification. We enhance the bounded model checking formula with additional constraints to express the notion of projections. The additional constraints causes the SMT solver to focus the search on relevant executions only, effectively shortening the solving times.

Research Question (RQ3): *Can the interaction patterns between threads be harnessed to achieve a sound error propagation analysis for multi-threaded programs?*

Our last contribution *(C4)* introduces an enhancement to EPA. In Chapter 5, we have proposed a novel approach that enables sound EPA on programs with non-deterministic behavior. The focus of the contribution is non-determinism due to dynamic memory allocation and CPU scheduling. We developed an automated check to verify when a sound EPA with no false-positives is possible. The automated check is based on the idea that non-deterministic scheduling can be successfully abstracted from if the interaction pattern between the threads is deterministic. We, then, proposed a canonicalization scheme, that successfully nullifies non-deterministic effects on executions of programs that pass our automated check. We implemented the automated check

and canonicalization scheme in Trace Sanitizer, a sound EPA tool for multi-threaded programs.

These contributions constitute improvements over existing formal analysis techniques that can aid in the development of dependable concurrent systems, particularly with respect to availability and safety.

LIST OF CODE LISTINGS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

BIBLIOGRAPHY

[AAB+17] Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J Stuckey. "Context-sensitive dynamic partial order reduction". In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 526–543.

[AAJ+14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. "Optimal dynamic partial order reduction". In: *Principles of Programming Languages (POPL)*. ACM Press. 2014, pp. 373–384.

[AAJ+18] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. "Optimal stateless model checking under the release-acquire semantics". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 135.

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. "Sharing memory robustly in message-passing systems". In: *Journal of the ACM (JACM)* 42.1 (1995), pp. 124–142.

[AH90] Hiralal Agrawal and Joseph R Horgan. "Dynamic program slicing". In: *Programming Language Design and Implementation (PLDI)*. ACM Press. 1990, pp. 246–256.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. "Partial orders for efficient bounded model checking of concurrent software". In: *Computer-Aided Verification (CAV)*. Springer Verlag. 2013, pp. 141–157.

[ALR+04] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[APB14] M. R. Aliabadi, K. Pattabiraman, and N. Bidokhti. "Soft-LLFI: A Comprehensive Framework for Software Fault Injection". In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. 2014, pp. 1–5.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. Vol. 19. John Wiley & Sons, 2004.

[BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo MK Martin. "Checkfence: checking consistency of concurrent data types on relaxed memory models". In: *Program Language Design and Implementation (PLDI)*. ACM Press. 2007, pp. 12–21.

[BBC+11] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services." In: *CIDR*. Vol. 11. 2011, pp. 223–234.

[BCC+99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. "Symbolic model checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer Verlag. 1999, pp. 193–207.

[BDF+15] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. "Putting consistency back into eventual consistency". In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 6.

[BDH02] Dragan Bošnački, Dennis Dams, and Leszek Holenderski. "Symmetric spin". In: *International Journal on Software Tools for Technology Transfer* 4.1 (2002), pp. 92–106.

[BDS03] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits". In: *USENIX Security Symposium*. Vol. 12. 2. 2003, pp. 291–301.

[BFF+14] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. "Coordination avoidance in database systems". In: *Proceedings of the VLDB Endowment* 8.3 (2014), pp. 185–196.

[BGM92] Daniel Barbard and Hector Garcia-Molina. "The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems". In: *Advances in Database Technology - EDBT'92*. Springer. 1992, pp. 373–388.

[BHJ+07] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. "The software model checker Blast". In: *International Journal on Software Tools for Technology Transfer* 9.5-6 (2007), pp. 505–525.

[BJVD+09] Robert L Bocchino Jr, S Adve Vikram, Danny Dig, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. "A type and effect system for deterministic parallel Java". In:

In Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA. ACM. 2009.

[BKS+08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC benchmark suite: Characterization and architectural implications". In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. 2008, pp. 72–81.

[BKS+11] Péter Bokor, Johannes Kinder, Marco Serafini, and Neeraj Suri. "Supporting domain-specific state space reductions through local partial-order reduction". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2011, pp. 113–122.

[BRG+78] Philip A. Bernstein, JB Rothnie, Nathan Goodman, and Christos A Papadimitriou. "The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case)". In: *IEEE Transactions on Software Engineering* 4.3 (1978), p. 154.

[BSS+09] Péter Bokor, Marco Serafini, Neeraj Suri, and Helmut Veith. "Role-based symmetry reduction of fault-tolerant distributed protocols with language support". In: *International Conference on Formal Engineering Methods*. Springer. 2009, pp. 147–166.

[BSS10] Péter Bokor, Marco Serafini, and Neeraj Suri. "On efficient models for model checking message-passing distributed protocols". In: *Formal Techniques for Distributed Systems*. Springer, 2010, pp. 216–223.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The smt-lib standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*. Vol. 13. 2010, p. 14.

[Ber17] Phil Bernstein. "Timestamp-based Concurrency Control in SDD-1". In: *Workshop in Failed Aspirations in Database Systems*. 2017.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1977, pp. 238–252.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[CDE+13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[CEF+96] Edmund M. Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. "Exploiting symmetry in temporal logic model checking". In: *Formal methods in system design* 9.1-2 (1996), pp. 77–104.

[CFJ+12] Miguel Correia, Daniel Gómez Ferro, Flavio Paiva Junqueira, and Marco Serafini. "Practical Hardening of Crash-Tolerant Systems." In: *USENIX Annual Technical Conference*. Vol. 12. 2012.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement". In: *Computer-Aided Verification (CAV)*. Springer Verlag. 2000, pp. 154–169.

[CHR98] J. Christmansson, M. Hiller, and M. Rimen. "An experimental comparison of fault and error injection". In: *Proc. ISSRE '98*. 1998, pp. 369–378.

[CJGK+18] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[CJZ+10] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. "Schism: A Workload-driven Approach to Database Replication and Partitioning". In: *PVLDB* 3.1-2 (2010). ISSN: 2150-8097.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A tool for checking ANSI-C programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer Verlag. 2004, pp. 168–176.

[CRS+08] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. "PNUTS: Yahoo!'s hosted data serving platform". In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1277–1288.

[CSW+17] Nicolas Coppik, Oliver Schwahn, Stefan Winter, and Neeraj Suri. "TrEKer: Tracing Error Propagation in Operating System Kernels". In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 2017, pp. 377–387.

[CWS+17] Abraham Chan, Stefan Winter, Habib Saissi, Karthik Pattabiraman, and Neeraj Suri. "IPA: Error Propagation Analysis of Multi-Threaded Programs Using Likely Invariants". In: *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2017, pp. 184–195.

[CZG+15] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. "Deterministic Replay: A Survey". In: *ACM Comput. Surv.* 48.2 (2015), 17:1–17:47.

[Cis18] Cisco. "Cisco Visual Networking Index (VNI): Forecast and Trends, 2017–2022". In: *White Paper* (2018).

[DAEA10] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. "G-store: a scalable data store for transactional multi key access in the cloud". In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 163–174.

[DEAA09] Sudipto Das, Amr El Abbadi, and Divyakant Agrawal. "ElasTraS: An Elastic Transactional Data Store in the Cloud." In: *HotCloud* 9 (2009), pp. 131–142.

[DHH+06] Matthew B Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Ranganath, Todd Wallentine, et al. "Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer Verlag. 2006, pp. 73–89.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[Det] *DeterLab*. `https://www.isi.deterlab.net/`. 2019.

[EN517] EN50126. "Railway applications - the specification and demonstration of reliability, availability, maintainability and safety (RAMS)". In: *European Committee for Electrotechnical Standardization (CENELEC)* (2017).

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. "The Daikon system for dynamic detection of likely invariants". In: *Science of Computer Programming* 69.1 (2007), pp. 35 –45.

[Eba] *ebay*. https://www.ebay.com. 2019.

[Esb] *ESBMC*. http://www.esbmc.org/. 2019.

[FA15] Jose M Faleiro and Daniel J Abadi. "Rethinking serializable multiversion concurrency control". In: *Proceedings of the VLDB Endowment* 8.11 (2015), pp. 1190–1201.

[FG05] Cormac Flanagan and Patrice Godefroid. "Dynamic Partial-order Reduction for Model Checking Software". In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2005, pp. 110–121.

[GFY+07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. "Cartesian partial-order reduction". In: *SPIN Workshop*. Springer Verlag. 2007.

[GG08] Malay Ganai and Aarti Gupta. "Tunneling and slicing: towards scalable BMC". In: *Design Automation Conference (DAC)*. IEEE Press. 2008, pp. 137–142.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2003, pp. 29–43.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed automated random testing". In: *In Programming Language Design and Implementation (PLDI*. ACM, 2005, pp. 213–223.

[GVLH+96] Patrice Godefroid, J Van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Vol. 1032. Springer Heidelberg, 1996.

[GWZ+11] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. "Practical software model checking via dynamic interface reduction". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 265–278.

[God97] Patrice Godefroid. "Model checking for programming languages using VeriSoft". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1997, pp. 174–186.

[God99] Patrice Godefroid. "Exploiting symmetry when model-checking software". In: *Formal Methods for Protocol Engineering and Distributed Systems*. Springer, 1999, pp. 257–275.

[HJS01] M. Hiller, A. Jhumka, and N. Suri. "An approach for analysing the propagation of data errors in software". In: *2001 International Conference on Dependable Systems and Networks (DSN)*. 2001, pp. 161–170.

[HJS02] M. Hiller, A. Jhumka, and N. Suri. "On the placement of software mechanisms for detection of data errors". In: *Proceedings International Conference on Dependable Systems and Networks (DSN)*. 2002, pp. 135–144.

[HMR14] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. "Maximal sound predictive race detection with control flow abstraction". In: *ACM SIGPLAN Notices* 49.6 (2014), pp. 337–348.

[Hol97] Gerard J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on software engineering* 23.5 (1997), pp. 279–295.

[ID96] C Norris Ip and David L Dill. "Better verification through symmetry". In: *Formal methods in system design* 9.1-2 (1996), pp. 41–75.

[IEC10] IEC61508. "Functional safety of electrical/electronic/programmable electronic safety-related systems". In: *IEC International Standard* (2010).

[INS+17] Tasuku Ishigooka, Fumio Narisawa, Kohei Sakurai, Neeraj Suri, Habib Saissi, Thorsten Piper, and Stefan Winter. *Method and System for Testing Control Software of a Controlled System*. US Patent 9575877. 2017.

[ISO11] ISO26262. "Road vehicles-functional safety". In: *International Standard ISO* (2011).

[ISP+14] Tasuku Ishigooka, Habib Saissi, Thorsten Piper, Stefan Winter, and Neeraj Suri. "Practical Use of Formal Verification for Safety Critical Cyber-Physical Systems: A Case Study". In: *IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*. IEEE. 2014, pp. 7–12.

[ISP+16] Tasuku Ishigooka, Habib Saissi, Thorsten Piper, Stefan Winter, and Neeraj Suri. "Practical Formal Verification for Model-Based Development of Cyber-Physical Systems". In: *IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. IEEE. 2016, pp. 1–8.

[ISP+17] Tasuku Ishigooka, Habib Saissi, Thorsten Piper, Stefan Winter, and Neeraj Suri. "Safety Verification Utilizing Model-Based Development for Safety Critical Cyber-Physical Systems". In: *Journal of Information Processing* 25 (2017), pp. 797–810.

[JM05] Ranjit Jhala and Rupak Majumdar. "Path slicing". In: *Program Language Design and Implementation (PLDI)*. ACM Press. 2005, pp. 38–47.

[JRS11] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. "Zab: High-performance broadcast for primary-backup systems". In: *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. IEEE. 2011, pp. 245–256.

[JS13] Flavio P Junqueira and Marco Serafini. "On barriers and the gap between active and passive replication". In: *Distributed Computing*. Springer, 2013, pp. 299–313.

[Jpa] `https://github.com/javaparser/javaparser`. 2015.

[Jpf] *Java Path Finder (JPF)*. `https://github.com/javapathfinder/jpf-core`. 2019.

[KAJ+07] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. "Life, death, and the critical transition: Finding liveness bugs in systems code". In: NSDI. 2007.

[KHA+09] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. "Consistency rationing in the cloud: pay only when it matters". In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 253–264.

[KKB+12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. "Efficient State Merging in Symbolic Execution". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2012, pp. 193–204.

[KWG09] Vineet Kahlon, Chao Wang, and Aarti Gupta. "Monotonic partial order reduction: An optimal symbolic partial order reduction technique". In: *Computer-Aided Verification (CAV)*. Springer Verlag. 2009, pp. 398–413.

[Kri04] Jens Krinke. "Advanced slicing of sequential and concurrent programs". In: *International Conference on Software Maintenance*. IEEE Press. 2004, pp. 464–468.

[LCB11] Tongping Liu, Charlie Curtsinger, and Emery D Berger. "Dthreads: efficient deterministic multithreading". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 327–336.

[LDM+09] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. "A framework for state-space exploration of Java-based actor programs". In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2009, pp. 468–479.

[LFK+11] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 401–416.

[LJo9] M. Leeke and A. Jhumka. "Evaluating the Use of Reference Run Models in Fault Injection Analysis". In: *Proc. PRDC '09*. 2009, pp. 121–124.

[LM12] G.S. Lemos and E. Martins. "Specification-guided Golden Run for Analysis of Robustness Testing Results". In: *Proc. SERE '12*. 2012, pp. 157–166.

[LMA+14] Jed Liu, Tom Magrino, Owen Arden, Michael D George, and Andrew C Myers. "Warranties for faster strong consistency". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 503–517.

[LPC+12] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. "Making geo-replicated systems fast as possible, consistent when necessary". In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 2012, pp. 265–278.

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu K Lahiri. "A solver for reachability modulo theories". In: *Computer-Aided Verification (CAV)*. Springer Verlag. 2012, pp. 427–443.

[LT93] Nancy G Leveson and Clark S Turner. "An investigation of the Therac-25 accidents". In: *IEEE computer* 26.7 (1993), pp. 18–41.

[Lam77] Leslie Lamport. "Proving the correctness of multiprocess programs". In: *IEEE transactions on software engineering* 2 (1977), pp. 125–143.

[Lam98] Leslie Lamport. "The part-time parliament". In: *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998), pp. 133–169.

[Lan92] William Landi. "Undecidability of static analysis". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), pp. 323–337.

[Lio+96] Jacques-Louis Lions et al. *Ariane 5 flight 501 failure*. 1996.

[Llb] *LLBMC*. `http://llbmc.org/`. 2019.

[MA14] Wes Masri and Rawad Abou Assi. "Prevalence of Coincidental Correctness and Mitigation of Its Impact on Fault Localization". In: *ACM Trans. Softw. Eng. Methodol.* 23.1 (2014), pp. 1–28.

[MBS88] Ravi Mukkamala, Steven C Bruell, and Roger K Shultz. *Design of partially replicated distributed database systems: an integrated methodology*. Vol. 16. 1. ACM, 1988.

[MCZ+14] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. "Extracting more concurrency from distributed transactions". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 479–494.

[MDC06] Alice Miller, Alastair Donaldson, and Muffy Calder. "Symmetry in temporal logic model checking". In: *ACM Computing Surveys (CSUR)* 38.3 (2006), p. 8.

[MSB+11] Can Arda Muftuoglu, Habib Saissi, Péter Bokor, and Neeraj Suri. "Scalable verification of distributed systems implementations via messaging abstraction". In: *ACM 23rd Symposium on Operating Systems Principles (SOSP) WiP section*. ACM. 2011.

[MSB+16] Patrick Metzler, Habib Saissi, Péter Bokor, Robin Hesse, and Neeraj Suri. "Efficient Verification of Program Fragments: Eager POR". In: *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer. 2016, pp. 375–391.

[MSB+17] Patrick Metzler, Habib Saissi, Péter Bokor, and Neeraj Suri. "Quick Verification of Concurrent Programs by Iteratively Relaxed Scheduling". In: *IEEE/ACM 32nd International Conference on Automated Software Engineering (ASE)*. IEEE Press. 2017, pp. 776–781.

[Maz87] Antoni Mazurkiewicz. "Trace theory". In: *Petri nets: applications and relationships to other models of concurrency*. Springer Verlag, 1987, pp. 278–324.

[McM06] Kenneth L McMillan. "Lazy abstraction with interpolants". In: *Computer-Aided Verification (CAV)*. Springer Verlag. 2006, pp. 123–136.

[Mpb] *MP-Basset*. `https://www.informatik.tu-darmstadt.de/deeds/research_1/tools_2223/mp_basset2/index.en.jsp`. 2019.

[NNH15] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[NWC+18] R. Natella, S. Winter, D. Cotroneo, and N. Suri. "Analyzing the Effects of Bugs on Software Interfaces". In: *IEEE Transactions on Software Engineering (to appear)* (2018).

[O'N86] Patrick E O'Neil. "The escrow transactional method". In: *ACM Transactions on Database Systems (TODS)* 11.4 (1986), pp. 405–430.

[PCZ12] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 61–72.

[PH13] David A Patterson and John L Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[PST+96] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. "Bayou: replicated database services for world-wide applications". In: *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*. ACM. 1996, pp. 275–280.

[PVB+13] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. "Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis". In: *Automated Software Engineering (ASE)* (2013), pp. 391–425.

[Pap79] Christos H Papadimitriou. "The serializability of concurrent database updates". In: *Journal of the ACM (JACM)* 26.4 (1979), pp. 631–653.

[Pho] *Phoenix benchmark*. `https://github.com/kozyraki/phoenix`. 2016.

[QKD13] Abdul Quamar, K Ashwin Kumar, and Amol Deshpande. "SWORD: scalable workload-aware data placement for transactional workloads". In: *International Conference on Extending Database Technology*. 2013, pp. 430–441.

[RH07] Venkatesh Prasad Ranganath and John Hatcliff. "Slicing concurrent Java programs using Indus and Kaveri". In: *International Journal on Software Tools for Technology Transfer* 9.5-6 (2007), pp. 489–504.

[RKB+15] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. "The homeostasis protocol: Avoiding transaction coordination through program analysis". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1311–1326.

[Rub] *Rubis: Rice university bidding system*. `http://rubis.ow2.org/`. 2019.

[SBM+13] Habib Saissi, Péter Bokor, Can Arda Muftuoglu, Neeraj Suri, and Marco Serafini. "Efficient Verification of Distributed Protocols Using Stateful Model Checking". In: *IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2013, pp. 133–142.

[SBS+11a] Habib Saissi, Péter Bokor, Marco Serafini, and Neeraj Suri. "To Crash or Not To Crash: Efficient Modeling of Fail-Stop Faults". In: *Invited paper, International Workshop on Logical Aspects of Fault-Tolerance (LAFT in assoc. with LICS)*. Springer. 2011.

[SBS+11b] Marco Serafini, Péter Bokor, Neeraj Suri, Jonny Vinter, Astrit Ademaj, Wolfgang Brandstatter, Fulvio Tagliabo, and Jens Koch. "Application-level diagnostic and membership protocols for generic time-triggered systems". In: *IEEE Transactions on Dependable and Secure Computing* 8.2 (2011), pp. 177–193.

[SBS15] Habib Saissi, Péter Bokor, and Neeraj Suri. "PBMC: Symbolic Slicing for the Verification of Concurrent Programs". In: *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer. 2015, pp. 344–360.

[SCD+17] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. "Bringing Modular Concurrency Control to the Next Level". In: *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*. ACM, 2017, pp. 283–297.

[ŞCR12] Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. "Maximal causal models for sequentially consistent systems". In: *International Conference on Runtime Verification*. Springer. 2012, pp. 136–150.

[SDM+10] Marco Serafini, Dan Dobre, Matthias Majuntke, Péter Bokor, and Neeraj Suri. "Eventually linearizable shared objects". In: *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM. 2010, pp. 95–104.

[SLS+95] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. "Transaction chopping: Algorithms and performance studies". In: *ACM Transactions on Database Systems (TODS)* 20.3 (1995), pp. 325–363.

[SPA+11] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. "Transactional storage for geo-replicated systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 385–400.

[SSS19] Habib Saissi, Marco Serafini, and Neeraj Suri. "Gyro: A Modular Scale-out Layer for Single-Server DBMSs". In: *USENIX Annual Technical Conference (ATC'19), (under submission)* (2019).

[SW11] Nishant Sinha and Chao Wang. "On interference abstractions". In: *Principles of Programming Languages (POPL)*. ACM Press, 2011, pp. 423–434.

[SWS+19] Habib Saissi, Stefan Winter, Oliver Schwahn, Karthik Pattabiraman, and Neeraj Suri. "Trace Sanitizer: Eliminating Effects of Non-Determinism on Execution Traces". In: *International Symposium on Software Testing and Analysis (ISSTA'19), (under submission)* (2019).

[Sen07] Koushik Sen. "Concolic testing". In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM. 2007, pp. 571–572.

[Som+15] Ian Sommerville et al. *Software engineering*. Pearson, 2015.

[Spi] *Spin*. `http://spinroot.com/spin/whatispin.html`. 2019.

[Sym] *SymmSpin*. `http://www.win.tue.nl/~lhol/SymmSpin/`. 2012.

[TDP+94] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. "Session guarantees for weakly consistent replicated data". In: *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*. IEEE. 1994, pp. 140–149.

[TDW+12] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. "Calvin: fast distributed transactions for partitioned database systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 1–12.

[TNS+14] Khai Q Tran, Jeffrey F Naughton, Bruhathi Sundarmurthy, and Dimitris Tsirogiannis. "JECB: A join-extension, code-based approach to OLTP data partitioning". In: *ACM SIGMOD International Conference on Management of Data*. 2014, pp. 39–50.

[TP13] Anna Thomas and Karthik Pattabiraman. "LLFI: An intermediate code level fault injector for soft computing applications". In: *Workshop on Silicon Errors in Logic System Effects (SELSE)*. 2013.

[Tpc] `http://www.tpc.org/tpcw/spec/tpcw_v1.8.pdf`.

[Val98] Antti Valmari. "The state explosion problem". In: *Lectures on Petri nets I: Basic models*. Springer Verlag, 1998, pp. 429–528.

[Voa97] J. Voas. "Error propagation analysis for COTS systems". In: *Computing Control Engineering Journal* 8.6 (1997), pp. 269–272.

[WKO13] Björn Wachter, Daniel Kroening, and Joël Ouaknine. "Verifying multi-threaded software with Impact". In: *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE Press. 2013, pp. 210–217.

[Wei81] Mark Weiser. "Program slicing". In: *ICSE*. IEEE Press. 1981, pp. 439–449.

[XSL+15] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. "High-performance ACID via modular concurrency control". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM. 2015, pp. 279–294.

[YCW+09] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. "MODIST: Transparent model checking of unmodified distributed systems". In: (2009).

[YKK+09] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. "CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems". In: *The 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2009.

[YV00] Haifeng Yu and Amin Vahdat. "Design and evaluation of a continuous consistency model for replicated services". In: *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association. 2000, pp. 21–21.

[Z3] *Z3 SMT Solver*. `https://github.com/Z3Prover/z3`. 2019.

[ZPZ+13] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. "Transaction chains: achieving serializability with low latency in geo-distributed storage systems". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 276–291.

[ZSS+15] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. "Building consistent transactions with inconsistent replication". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM. 2015, pp. 263–278.

[Zoo] *ZooKeeper*. https://wiki.apache.org/hadoop/ZooKeeper. 2019.