UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1745

Model Checking of Software Systems under Weak Memory Models

TUAN-PHONG NGO

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2019

ISSN 1651-6214
ISBN 978-91-513-0506-6
urn:nbn:se:uu:diva-366361

Dissertation presented at Uppsala University to be publicly examined in 2446, Department of Information Technology, Polacksbacken (Lägerhyddsvägen 2), Uppsala, Monday, 21 January 2019 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Dr. Viktor Vafeiadis (Max Planck Institute for Software Systems).

Abstract

Ngo, T.-P. 2019. Model Checking of Software Systems under Weak Memory Models. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1745. 61 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0506-6.

When a program is compiled and run on a modern architecture, different optimizations may be applied to gain in efficiency. In particular, the access operations (e.g., read and write) to the shared memory may be performed in an out-of-order manner, i.e., in a different order than the order in which the operations have been issued by the program. The reordering of memory access operations leads to efficient use of instruction pipelines and thus an improvement in program execution times. However, the gain in this efficiency comes at a price. More precisely, programs running under modern architectures may exhibit unexpected behaviors by programmers. The out-of-order execution has led to the invention of new program semantics, called weak memory model (WMM). One crucial problem is to ensure the correctness of concurrent programs running under weak memory models.

The thesis proposes three techniques for reasoning and analyzing concurrent programs running under WMMs. The first one is a sound and complete analysis technique for finite-state programs running under the TSO semantics (Paper II). This technique is based on a novel and equivalent semantics for TSO, called Dual TSO semantics, and on the use of well-structured transition framework. The second technique is an under-approximation technique that can be used to detect bugs under the POWER semantics (Paper III). This technique is based on bounding the number of contexts in an explored execution where, in each context, there is only one active process. The third technique is also an under-approximation technique based on systematic testing (a.k.a. stateless model checking). This approach has been used to develop an optimal and efficient systematic testing approach for concurrent programs running under the Release-Acquire semantics (Paper IV).

The thesis also considers the problem of effectively finding a minimal set of fences that guarantees the correctness of a concurrent program running under WMMs (Paper I). A fence (a.k.a. barrier) is an operation that can be inserted in the program to prohibit certain reorderings between operations issued before and after the fence. Since fences are expensive, it is crucial to automatically find a minimal set of fences to ensure the program correctness. This thesis presents a method for automatic fence insertion in programs running under the TSO semantics that offers the best-known trade-off between the efficiency and optimality of the algorithm. The technique is based on a novel notion of correctness, called Persistence, that compares the behaviors of a program running under WMMs to that running under the SC semantics.

Keywords: Model checking, Concurrent program, Weak memory model

Tuan-Phong Ngo, Department of Information Technology, Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-0506-6
urn:nbn:se:uu:diva-366361 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-366361)

Dành cho bố mẹ, Chi, Thùy và Quỳnh-Anh
Dedicated to my parents, Chi, Thuy and Quynh-Anh

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I **The Best of Both Worlds: Trading Efficiency and Optimality in Fence Insertion for TSO.** Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan Phong Ngo. In *European Symposium on Programming* (ESOP), 2015. [15]

II **A Load-Buffer Semantics for Total Store Ordering**. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. In *Special issue: Selected Papers of the Conference on Concurrency Theory (CONCUR 2016)*, Logical Methods in Computer Science, 2018. [8]
This is an extended version of the paper **The Benefits of Duality in Verifying Concurrent Programs under TSO**, published in *International Conference on Concurrency Theory* (CONCUR), 2016. [5]

III **Context-Bounded Analysis for POWER**. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. In *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS), 2017. [6, 7]
Nominated to be one the best papers at the Joint European Conferences on Theory and Practice of Software (ETAPS), 2017.

IV **Optimal Stateless Model Checking under the Release-Acquire Semantics**. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. In *Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), 2018. [13]

I am the main author of these papers. The ideas originated and were developed in discussions with other authors. I was the sole implementor. I wrote the papers together with other authors. Reprints were made with permission from the publishers.

Other publications

The following papers were not included in this thesis.

V **Precise and Sound Automatic Fence Insertion Procedure under PSO.** Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. In *Networked Systems* (NETYS), 2015. [14]

VI **Replacing Store Buffers by Load Buffers in TSO**. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. In *Verification and Evaluation of Computer and Communication Systems* (VECoS), 2018. [9]

Acknowledgement

I really enjoyed my seven years of life in Uppsala. During these years, my research was very productive, mainly thanks to the interactions with many friendly people that I would like to thank.

First of all, I want to express my deep gratitude and appreciation to Dr. Mohamed Faouzi Atig who is my guide and my mentor for my five-year doctoral studies. Faouzi was the first teacher to teach me the principles of programming theory and was the main advisor of my Master thesis as well. Faouzi has been giving me much creative and wise advice. I cannot thank him enough for all the time he has spent working with me throughout these years.

My gratitude also goes to my second advisor, Prof. Parosh Aziz Abdulla, for his trust in me and his encouragement to try out new ideas. The algorithmic techniques discussed in this thesis would not have been fully developed and proved correct without his help. I am deeply grateful for all his guidance and support.

I would like to thank Dr. Philipp Rümmer, my third advisor, for his advice, moral support, encouragement, and also for all the valuable graduate courses he gave and which I attended with great pleasure.

I am very thankful to my co-authors Prof. Bengt Jonsson and Prof. Ahmed Bouajjani. Thank you for your encouraging comments and for the pleasant discussions we had during the work.

My thanks also go to the previous and current members (in alphabetical order) of the Verification group in which I am a member: Phi Diep Bui, Aiswarya Cyriac, Sofia Cassel, Frédéric Haziza, Lisa Kaati, Carl Leonardsson, Ahmed Rezine, Othmane Rezine, Amendra Shrestha, Jari Stenman, Cong Quy Trinh, and Yunyun Zhu. I will always remember the Friday lunches with you.

I would also like to thank many people from the department who supported me during my study. In particular, thanks to Dr. Kostis Sagonas and Magnus Lång for their useful discussions and comments to improve the dynamic partial order reduction technique presented in this thesis. Special thanks also go to Magnus for helping me with the Swedish summarization of this thesis. Thanks to Prof. Anders Berglund for your guidance as a coordinator when the first time I came to Sweden seven years ago.

This list would not be complete without mentioning my life-long Vietnamese friends. Without their support, I would not have survived so far from my tropical country! Quý - Hà, Điệp - Dịu, Phúc - Trâm Anh, Thành - Diễm, Lâm - Thùy, Minh - Thủy, Tuấn - Oanh, Long - Thái, Tuân - Liên, Hoàng

Long, Tuấn Anh, Nam, Lan, Thảo, Hoàng Tuấn, Thư, Dung, Mận, Viện - Nga, Trung - Hà, Linh, Hùng Al, thank you for always being here for me. There are also other friends that I have not met for a long time. After this thesis, I hope that I can meet them soon.

Finally, I would like to express my deepest gratitude and love to my family in Vietnamese.

Con cám ơn Bố mẹ vì đã sinh ra và giáo dục con thành người. Mong muốn lớn nhất của con là Bố mẹ luôn mạnh khỏe! Con cũng biết ơn Ông bà ngoại rất nhiều vì đã vất vả vì gia đình con. Cám ơn các Anh chị và các cháu đã luôn ở bên gia đình chúng em.

Luận văn này dành tặng cho gia đình, nhất là vợ Quỳnh Anh và hai con - Chi và Thùy. Chồng cám ơn vợ vì tất cả. Không có vợ, chồng nhiều khi đã lạc lối! Chi và Thuỳ, các con là những đứa trẻ mạnh mẽ và dũng cảm nhất đối với bố. Không có các con, bố có thể đã kết thúc luận văn này sớm hơn, nhưng sẽ không trải qua khoảng thời gian hạnh phúc đã qua.

Funding:
This thesis was funded by the Linnaeus center of excellence UPMARC (Uppsala Programming for Multicore Architectures Research Center).

Sammanfattning på Svenska

När ett program kompileras för och körs på en modern processorarkitektur så används ett flertal knep för att öka hur effektivt det kör. Exempelvis kan operationer som läser eller skriver det delade minnet komma att köras i oordning, det vill säga i en annan ordning än de står skrivna i programkoden. När en processor ordnar om operationer kan den använda sina pipelinear mer effektivt, och därmed köra programmet snabbare. I enkeltrådiga program är denna ordnade körning osynlig för programmeraren, då denne fortfarande kan arbeta under minnesmodellen Sequential Consistency (SC). Detta gäller dock inte för parallella program som kommunicerar via delat minne. I dessa fall kan program som kör på moderna processorarkitekturer uppvisa beteenden som inte avsetts av sina författare. Till exempel så kan parallella algoritmer såsom lås och producent-konsument bete sig inkorrekt. Att operationer körs i oordning har lett till introduktionen av nya programsemantiker, *svaga minnesmodeller*, för att beskriva vilka typer av operationer som tillåts omordnas med varandra. En viktig vetenskaplig frågeställning är hur man kan garantera att ett parallellt program beter sig korrekt under en svag minnesmodell.

Den här avhandlingen presenterar tre tekniker för att resonera kring och analysera parallella program som kör på svaga minnesmodeller.

Den första (artikel II) är en precis och uttömmande teknik för program med finit tillståndsrymd som kör på minnesmodellen Total Store Order (TSO). TSO är en av de vanligaste modellerna och motsvarar dels vad som erbjuds av Sun Microsystems flerkärniga SPARC-processorer, och dels formaliseringar av minnesmodellen för Intels x86. Denna teknik bygger på en ny men ekvivalent semantik för TSO, som vi kallar "Dual TSO" (den duala TSO-semantiken), och ett välstrukturerat ramverk för att uttrycka programmen som övergångssystem (eng. *transition systems*). Vi visar att den duala TSO-semantiken tillåter (i) att förenkla korrekthetsanalys under TSO, (ii) att drastiskt skalbarheten jämfört med befintliga tekniker, och (iii) att genrealisera korrekthetsanalysen till att tillåta parametrisk analys. Duala TSO-semantiken ger ett nytt avgörbarhetsresultat inom parametrisk verifiering, och vidare, en verifieringsalgoritm som är mer generell och effektivare i praktiken än de för begränsade parametrar.

Den andra tekniken (artikel III) är en under-approximation som kan användas för att hitta buggar under minnsemodellen POWER, som erbjuds av IBMs PowerPC-processorarkitektur. TSO tillåter skrivningar att köra om läsningar (av andra minnesadresser). POWER tillåter därtill, under komplexa villkor, omordning av alla sorters läs- och skrivoperationer. Då beslutsproblemet om ett programtillstånd är nåbart under POWER-semantiken är oavgörbart, även för program med en finit tillståndsrymd, så baseras den presenterade tekniken på en heuristisk sökstrategi. Tekniken söker bara genom körningar där antalet gånger schemaläggaren tillåts byta ut den körande tråden begränsas till en konstant k. Intuitionen som inspirerade denna heuristik är att många icketriviala parallellismbuggar kan upptäckas även med små värden på k. Att begränsa antalet trådbyten låter oss reducera beslutsproblemet till det motsvarande under SC-minnesmodellen genom att definiera en kod-till-kod-konversion. Genom att utöka det välkända verktyget för modellbaserad testning CBMC har vi byggt en prototyp och tillämpat den på ett flertal program, vilket visar tillämpligheten av vår metod.

Den tredje tekniken (artikel IV) är baserad på tillståndsfri modellbaserad testning (eng. stateless model checking, SMC) och verifierar program där alla körningar är av begränsad längd under minnesmodellen Release-Acquire (RA). RA är ett fragment av C++11-minnesmodellen där alla skrivningar är “release atomic” och alla läsningar är “acquire atomic”, och är både praktiskt samt beter sig väl matematiskt. RA-semantiken är en god kompromiss mellan prestanda och användarvänlighet. Tekniken utforskar alla grafer av programmets operationer bestående av programordning (eng. program order) och läsordning (eng. read-from). Jämför detta med äldre tekniker, som därutöver också inkluderar koherensordning, dvs. ordningen av skrivningar till samma address. Då programfel såsom krasher eller brott mot invarianter (eng. assertions) endast orsakas av läsordningen så undviker vi en betydlig källa till redundant utforskning. Vi presenterar en algoritm för tillståndsfri modellbaserad testning som är optimal i bemärkelsen att den endast utforskar varje graf en gång. Detta optimalitetsresultat är strikt bättre än tidigare jämförbara optimalitetsresultat som också inkluderar koherensordning i grafen. Vi har implementerad denna algoritm i ett verktyg, Tracer, och visar experimentellt att Tracer kan vara betydligt snabbare än dåvarande spjutspets-verktyg för RA-minnesmodellen.

Slutligen angriper den här avhandlingen problemet att hitta en uppsättning av barriär-operationer som gör ett program korrekt under någon svar minnesmodell utan att kompromissa med dess effektivitet (artikel I). En barriäroperation (eng. fence/barrier) kan stoppas in i ett program för att förhindra omordning av vissa typer av operationer före barriären med vissa typer efter barriären, svaga minnesmodeller till trots. Då barriäroperationer kan orsaka kraftigt reducerad effektivitet så är det kritiskt att kunna hitta en minimal uppsättning av dem som fortfarande gör programmet korrekt. Denna avhandling presenterar en metod för att automatiskt hitta dessa uppsättningar för parallella program som kör på TSO, och erbjuder den bästa kompromissen mellan effektivitet och minimalitet. Tekniken bygger på en ny definition av korrekthet, döpt “Persistence”, som jämför beteendet av programmet under en svag minnesmodell med det under SC.

Contents

Acknowledgement **ix**

Sammanfattning på Svenska **xi**

1 Introduction **15**
- 1.1 Background . . . 15
  - 1.1.1 Testing . . . 16
  - 1.1.2 Model Checking . . . 17
- 1.2 Weak Memory Models . . . 25
  - 1.2.1 The SC Semantics . . . 25
  - 1.2.2 The Need of Weak Memory Models . . . 26
  - 1.2.3 The TSO Semantics . . . 26
  - 1.2.4 The POWER Semantics . . . 28
  - 1.2.5 The Release-Acquire Semantics . . . 31
  - 1.2.6 Fences in Weak Memory Models . . . 34
- 1.3 Challenges in Model Checking for Weak Memory Models . . . 36

2 Summary of Contributions **39**
- 2.1 Paper I: The Best of Both Worlds: Trading Efficiency and Optimality in Fence Insertion for TSO . . . 39
- 2.2 Paper II: The Benefits of Duality in Verifying Concurrent Programs under TSO . . . 42
- 2.3 Paper III: Context-Bounded Analysis for POWER . . . 44
- 2.4 Paper IV: Optimal Stateless Model Checking under the Release-Acquire Semantics . . . 46

3 Conclusions and Directions for Future Work **49**

References **53**

1. Introduction

In this chapter, we present the background of the thesis and the motivation behind it. First, we explain why it is important to detect failures (a.k.a. bugs) as soon as possible in software systems. We also briefly introduce *testing* as a widely-used technique by programmers to detect software failures. After that, we give an overview of *model checking*, a technique that can address some of the limitations of testing. Then, we focus on two different variants of model checking methods, namely *state-space exploration* and *stateless model checking*. Finally, we give an overview of *weak memory models*: the reason for their introduction and the kind of behaviors they can introduce. We discuss the challenges in model checking of software running under weak memory models.

1.1 Background

Software systems are ubiquitous in every aspect of our daily lives. For example, they can control important applications such as nuclear power plants or air transportations. They are also used to manage our smartphones, the telecommunication network, and the Internet. Therefore, it is important to ensure the correctness of these software systems. Indeed, any *failure*, also known as *bug*, of the software systems can have costly and catastrophic repercussions in terms of money and human lives.

Although it is crucial to detect failures in software systems, *analyzing* such systems can be a challenging task. In particular, some systems might have a huge (even infinite) number of possible input combinations. Moreover, for efficiency reasons, most of the used software systems consist of *concurrent programs*. In concurrent programs, a computing task is conducted through the collaboration of many sequential tasks. More precisely, concurrent programs make use of the *multicore technologies*, in which sequential *processes* (of the concurrent programs) are run on different *processors* (a.k.a. cores) while sharing the *main memory*. Because of a potentially huge number of interactions between different processes in concurrent programs, the analyzing such systems is even harder.

In the following, we will present testing and model checking, two approaches used to find software failures and ensure, in some cases, the correctness (i.e., free of bugs) of software systems.

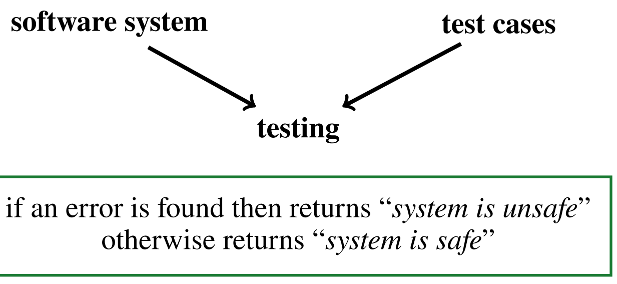
software system
test cases
testing
if an error is found then returns "*system is unsafe*"
otherwise returns "*system is safe*"

Figure 1.1. Basic form of testing.

1.1.1 Testing

Testing [33] is the most straightforward and widely-used technique by programmers to detect failures in software systems. Figure 1.1 presents a primary form of testing which consists of running the software under some specific conditions, called *test cases*, and then checking whether the result for a given input matches the expected output. In fact, testing can range from manual and ad-hoc techniques to fully-automated ones [51]. Furthermore, testing can be used in many applications such as for detecting errors in user frontend, user backend, network communication, and hardware circuits [51]. Thus, testing is often an important part of the software development process [109].

Although testing is an essential phase in the software development, it has a significant drawback. More precisely, *"program testing can be a very effective way to show the presence of bugs but is hopelessly inadequate for showing their absence"* [64]. This means that testing can only detect bugs in the scenarios specified by the given test cases. However, testing will never be able to detect failures that are not described by these scenarios. One might simply think that to show the absence of bugs, we only need to write all test cases that cover all possible scenarios of using the software. However, the number of these scenarios can be enormous. In the worst case, it can even be infinite. This can be the result of a huge number of possible inputs that can be given to the software together with a large number of possible executions. As a result, testing often not be able to cover all possible executions of the software in a reasonable (finite) amount of time.

Moreover, the evolution of the software industry in recent decades makes the problem of checking software correctness much harder. In fact, the improvement of CPU's performance by increasing the number of transistors on a chip (following Moore's law [120]) is not applicable anymore. This has resulted in the fact that nowadays chips contain several processors (a.k.a. cores) that can run in parallel in order to obtain a higher performance. However, the introduction of multicore technologies has made the problem of detecting software failures even harder. This is mainly due to a larger number of possible interactions between the processes running on different cores, making *concur-*

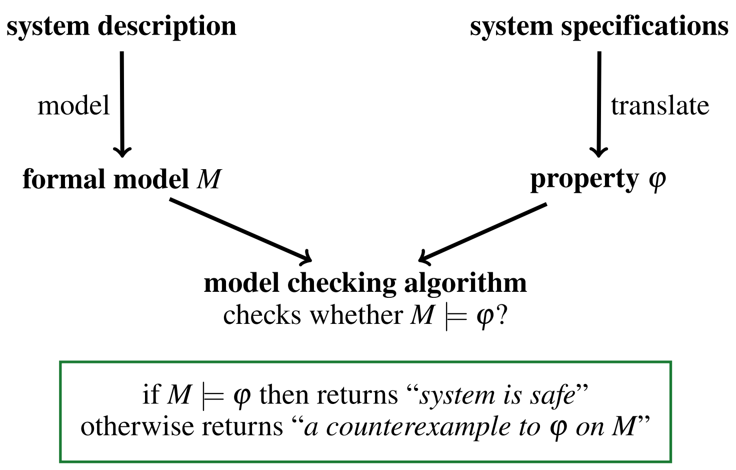
system description
system specifications
model
translate
formal model M
property φ
model checking algorithm
checks whether $M \models \varphi$?
if $M \models \varphi$ then returns "*system is safe*"
otherwise returns "*a counterexample to φ on M*"

Figure 1.2. Basic form of model checking.

rency errors hard to detect and reproduce. For this reasons, concurrency errors are often called *heisenbugs* [122].

1.1.2 Model Checking

Unlike testing, *model checking* is exhaustive. This means that it can show the absence of failures while testing can only show their presence [51]. Model checking is a *fully automatic* method, also known as the *push-button* method, that can be applied to complex sequential programs and concurrent programs. Model checking can be used to complement testing in order to overcome its limitations. First, model checking can detect bugs that are not easy to find and reproduce using classical testing methods. Second, model checking can guarantee the correctness of a software system when no errors are found during its exploration.

Figure 1.2 presents a *basic classical form* of model checking [54] which consists of three main components:

1. *Formal model*: M is a *finite-state graph*, also known as *Kripke* structure, representing a formal model of a software system. M represents the *state space* of the software, i.e., all behaviors of the system.
2. *Property*: φ is a *formula* describing a correctness property of the finite-state model M. We can identify two main classes of properties for model checking problems: namely *safety properties* and *liveness properties*.
    - Safety properties specify that nothing bad will happen in the system. For example, a safety property might specify that a floating point overflow error never occurs in a program. Checking safety property can be done by analyzing *finite executions* [30]. Moreover, this analysis can be reduced to check the violations of some

assertions in the system. When an assertion is violated in a state, we can declare that state to be a *bad* one. The model is said to be *unsafe* if any bad state is reachable from the *initial states* of the system. Otherwise, the system is said to be *safe*. Checking whether some given states are reachable in a software system is one of the most important model checking questions. This checking is also known as the *state reachability problem* [30].
  - Liveness properties specify that something good will eventually happen in the system. For instance, a liveness property can specify that the program will eventually terminate and return a result; or when a process requests a shared resource, it will eventually gain access to the resource. Checking liveness properties is often considered to be related to *infinite executions*.

   This thesis mainly focuses on addressing the state reachability problem for concurrent programs running under weak memory models (more details will be given in Section 1.2.2).
3. *Algorithm*: a decision procedure is used to determine whether the formal model M satisfies the property φ, denoted by $M \models \varphi$. Moreover, the decision procedure can produce *counterexamples* in the case where $M \not\models \varphi$, i.e., the model does not satisfy the property.

We can identify *two* challenges in model checking:

1. *The modeling challenges*: How to extend the model checking technique beyond the finite-state graph. Many systems are *infinite-state*, i.e., they have an *unbounded number* of states. This unboundedness can be due to the domain of the variables manipulated in the program, the number of participated processes, or the size of the used data structures. One approach to verifying such systems (and show the decidability of the model checking problems) consists of the construction of a finite-state *abstraction* of the system. Another approach involves the use of the framework of *well-structured transition systems* [16, 67] to show the decidability of the model checking problems for infinite-state systems [17, 11, 28]. There are also *approximate* techniques that can be used either to find, in an automatic manner, bugs such as bounded model checking [37, 36] or prove the correctness of systems such as predicate abstraction [78, 31].
2. *The algorithmic challenges*: How to design scalable model checking algorithms to handle real-life problems. During the checking of a software system, we have to deal with the explosion of states, even in the case where the system is finite-state. In fact, most real-life systems have a huge number of states. The number of states is often exponential in the number of its variables and processes. For example, a concurrent program with n processes, where each process has m local states, can have m^n states, i.e., an exponential number of states. The explosion of states in model checking is also known as the *state-explosion problem* [52].

In the following, we describe *four* techniques that have been used to address the state-space explosion problem.

- The first one is *symbolic model checking* [118, 42]. In this approach, a binary decision diagram (BDD) [41] is often used to compactly represent a set of states. Using BDDs can give us a much more efficient encoding than the explicit presentation. Therefore, it can greatly improve the efficiency of a model checking algorithm [42].
- The second technique is *counterexample-guided abstraction refinement* [47, 48, 50]. The technique tries to find an abstraction of the system by not taking into account unnecessary details. If the abstraction is not precise enough to model the system and a spurious counterexample is returned, then the abstraction is automatically refined using the counterexample.
- The third technique is *partial-order reduction* (POR) applied for concurrent programs [49, 129, 150, 74]. The technique exploits the independence between operations of different processes to reduce the number of explored executions.
  This thesis proposes an efficient improvement for POR applied to concurrent programs running under the Release-Acquire semantics (more details will be given in Sections 1.2.5 and 2.4).
- The last technique is *context-bounded model checking* [132] also applied to concurrent programs. The technique was first implemented in the CHESS tool [121]. It is based on bounding the number of *contexts* in an explored execution where, in each context, there is only one *active* process. Since we limit the number of context switches, we might cut off a large part of the state space and therefore can reduce the number of visited states. One advantage of the heuristic is that many concurrency bugs can be found using only a small number of context switches [132]. For different memory models (more details will be given in Section 1.2), context-bounded model checking has been studied and applied [143, 87, 29, 26, 68, 88, 144, 124, 145]. Moreover, some context-bounded model checking techniques for weak memory models (more details will be given in Section 1.2.2) have been implemented in the CSEQ tool [144, 145].
  This thesis proposes for the first time an efficient context-bounded model checking algorithm for concurrent programs running under the POWER semantics (more details will be given in Sections 1.2.4 and 2.3).

Below, we give an introduction to two major classes of model checking algorithms: state-space exploration and stateless model checking. Later, we will see that the algorithms proposed in Paper I – III are concrete instances of the state-space exploration class (more details will be given in Sections 2.1-

Algorithm 1: An algorithm for state-space exploration [75].

```
Input: a model $M$, a property $\varphi$, a starting state $s_0$, and an
       empty hash table $H$.
1  $S = \{s_0\}$;
2  while $S \neq \varnothing$ do
3      remove $s$ from $S$;
4      if $s$ does not satisfy $\varphi$ then
5          return $M \not\models \varphi$;
6      if $s$ is not already in $H$ then
7          add $s$ to $H$;
8          $T$ = getTransitions$(M, s)$;
9          for $t \in T$ do
10             $s'$ = getSuccessor$(M, s, t)$;
11             add $s'$ to $S$;
12 return $M \models \varphi$;
```

2.3) and the algorithm proposed in Paper IV belongs to the stateless model checking class (more details will be given in Sections 2.4).

State-space exploration

State-space exploration is one of the most successful class of model checking algorithms for analyzing the correctness of software systems [75]. It can be used to check both safety and liveness properties. State-space exploration has been implemented in tools, such as SVM [118], SPIN [84], BLAST [34], Java PathFinder [151], UPPAAL [106], and TLA+ [105].

A state-space exploration algorithm consists of exploring the state-space graph M and checking that no bad state can be reached from a *starting state* s_0. Algorithm 1 gives pseudocode for performing the search, following [75]. First, it uses two main data structures: a set S and a hash table H. The set S stores all states that will be explored. Meanwhile, the hash table H stores all states that have already been explored. Note that using the hash table H allows us to efficiently check whether a state s has been explored by searching s in H (line 6). At each step in the while loop at line 2, given an encountered state s in S, the algorithm explores all *successors* of s by executing all *enabled transitions* from s (lines 8-10). The set of all transitions enabled in the state s in the model M is given by the function *getTransitions*(M,s). Meanwhile, the state reached from the state s after the execution of a transition t in M is given by *getSuccessor*(M,s,t). During the exploration, if the state s is encountered and s does not satisfy the property φ, then the algorithm will stop and return $M \not\models \varphi$, i.e., the model M does not satisfy the property φ (line 5). Otherwise, the algorithm will terminate when the set S becomes empty and will return $M \models \varphi$, i.e, the model M satisfies the property φ (line 12). In the latter case, it also means that all states that can be reached from the starting state have been explored.

It is important to note that Algorithm 1 assumes that each state s must be presented by a *unique identifier*. Then by using these identifiers, the data structures S and H can explicitly store all needed states. Although different state-space exploration algorithms can implement variants of search techniques (see the next paragraph) or use binary decision diagram (BDD [41]), such as in symbolic model checking methods [118, 42], to compactly represent a set of states, all of them must be based on this critical assumption.

It is also important to note that there are different classes of search algorithms for state-space exploration. Algorithm 1 can be seen as a concrete instance of the *forward search* class [107]. In this class, the starting state is the same as the *initial state* of the system, and the algorithm will explore all encountered states until it finds a state that does not satisfy the property. Otherwise, the algorithm will explore all states that can be reached from the starting state. A different class of algorithms for state-space exploration is the *backward search* [107] in which the starting state is a *goal state* s^{goal} such that s^{goal} does not satisfy φ. From the starting state, a backward algorithm will proceed in a backward manner until the initial state of the system is encountered. Backward search algorithms have been used in many model checking approaches, especially for infinite-state systems, such as in the well-structured transition systems [16, 67]. The combination of the forward search and backward search is known as the *forward-backward search* [107].

Later, we will see that the algorithms proposed in Papers I – III are different variants of Algorithm 1 with different search stratergies. To be more precise, the algorithms in Papers I and III use a forward search to check some safety properties of concurrent programs running under the TSO semantics (more details will be given in Section 1.2.3). Meanwhile, the algorithm in Paper II is based on a backward search and the well-structured transition system [16, 67].

State-space exploration model is *stateful* in the sense that to explore all reachable states, we have to maintain a representation of all needed states in the data structures S and H. For a large program, that usually encounters the state-explosion problem, the number of states is typically huge. As a consequence, the representation of all visited states can be huge and complex. Thus, it is hard to efficiently store this representation (e.g., as a string of bits) in the memory at each step of the state-space exploration algorithm [75]. To overcome the problem of memory consumption in the state-space exploration algorithms, people introduce the class of stateless model checking algorithms that we will consider in the next paragraph.

Stateless model checking (SMC)

In the previous paragraph, we have seen that storing a representation of all needed states in the state-space exploration algorithms may require the use of excessive memory. *Stateless model checking* (SMC) algorithms [75] offer a way to avoid using too much memory during the exploration of the system. To be more precise, SMC algorithms explore the state space by *exhaustively* generating all *necessary* executions of the model. The examined executions can be created systematically under different *schedules* (i.e., an interleaving of operations from different processes) *one after one* such that at each point in time, it is sufficient to keep only one execution in memory. This brings down the memory consumption from a large number of needed states to some constants, proportional to the maximal length of explored executions.

It is important to note that stateless model checking algorithms are usually considered in concurrent programs that are *deterministic*, in the sense that they have fixed input and do not contain any data non-determinism. For example, they must have a fixed value returned by any call to the operating system. A consequence of this is that the execution of each process is also deterministic, in the sense that whenever we run the same execution, we will end up at the same state. Thus, by resetting all shared variables and local registers (resp. all processes) to their initial values (resp. initial states), and then replaying a particular schedule, we can obtain a unique execution and reach a unique state. Therefore, states of the programs in the stateless model checking algorithms can be encoded using the schedules used to achieve them. This eliminates the need to store any other state information (such as the values of shared variables, the values of local registers, and the local states of processes) as in the case of the state-space exploration algorithms.

It is also important to note that the model M corresponding to the software system must be *finite* and *acyclic*. To the best of our knowledge, all SMC tools (e.g. [75, 69, 3, 134, 122, 125, 126]) use this assumption. The reason is that an infinite or cyclic model can have infinite executions. Meanwhile, SMC techniques rely on inspecting the operations in a current execution to explore the other potential executions; therefore if the current execution is infinite, then these methods will not be able to generate the other executions. One popular mechanism to obtain finite and acyclic model is unrolling all loops in a given input program to a predetermined bound [18] as a preprocessing step for the stateless model checking algorithms.

In contrast to state-space exploration that can be used for both safety and liveness properties, SMC is often suitable to find violations of safety properties since it considers only finite executions.

In its basic form, a stateless model checking algorithm is a trade-off between small memory consumption and a high number of unnecessary explored executions, where the same state may be visited many times. Since concurrent processes can interleave their operations, the number of executions is generally exponential in the length of them. Therefore, a naive implementation of SMC can generate a large number of executions where many of them lead to the same state. To overcome this problem, partial order reduction (POR) techniques [49, 129, 150, 74] have been studied and applied to stateless model checking [69, 3, 137, 134]. The integration of partial order reduction and stateless model checking techniques is known as *dynamic partial order reduction* (DPOR) [69].

The main idea of partial order reduction techniques is to consider an execution as a partially-ordered graph of operations. An operation is ordered with respect to others using conflicting relations. Two operations are said to be *conflicting* if we can observe the order in which they appear in an execution, such as through the values of shared variables or local registers. Note that if two operations are conflicting, then we might consider them in a race condition [104]. The partial order on the operations is called the *happens-before relation* since it indicates intuitively which operations can be seen to happen before the others.

The happens-before relation can be used to define *equivalence classes* over the set of executions. Two executions are said to be equivalent if they induce the same happens-before relation between their operations. Under the sequential consistency (SC) memory model [103] (more details will be given in Section 1.2.1), such equivalence classes are called *Mazurkiewicz* traces [117]. For other memory models (more details will be given in Section 1.2.2), the natural generalization of Mazurkiewicz traces is called *Shasha-Snir* traces [141].

A Mazurkiewicz/Shasha-Snir trace characterizes an execution of a system by *three* relations between operations:

- $\texttt{po}$: *program order* totally orders the operations of each process. These operations contain *write* and *read* operations to some memory locations (variables) and other internal operations of processes such as local assignment, conditional branch, assume, and assertion.
- $\texttt{co}$: *coherence* totally orders the writes to each memory location. We use $\texttt{co}^x$ to denote the coherence order related to the memory location (variable) x.
- $\texttt{rf}$: *read-from* connects each write with the reads that read its value. We define the notation $\texttt{rf}^x$ in a similar way to $\texttt{co}^x$.

Under different memory models (more details will be given in Section 1.2), the $\texttt{co}$ and $\texttt{rf}$ relations need not be derived from the global order in which operations occur in an execution (as in the case under SC). Therefore, each particular memory model imposes restrictions on how these relations may be combined.

It is important to note that Mazurkiewicz/Shasha-Snir traces can be used to formalize the semantics of many memory models, e.g., in [140, 127, 112, 136, 23, 101, 94, 99, 70]. Moreover, Mazurkiewicz/Shasha-Snir traces can also be used in *Robustness*, a correctness criterion for concurrent programs running under weak memory models (more details will be given in Section 2.1).

Since all executions having the same Mazurkiewicz/Shasha-Snir trace keep all conflict operations in the same order by using the happens-before relation, they will have the same observable behavior (i.e., the same state). Hence, it is sufficient for an analysis technique to explore only necessary executions, i.e. exploring only one execution per Mazurkiewicz/Shasha-Snir trace. Dynamic

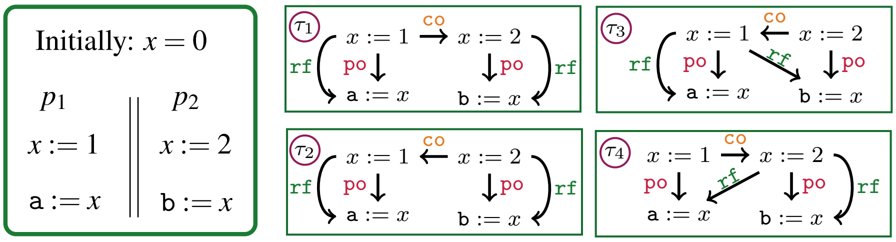

Figure 1.3. A simple concurrent program (left) and its Mazurkiewicz/Shasha-Snir traces (right).

partial order reduction (DPOR) [69] tries to reduce the number of unnecessary executions while still guaranteeing to investigate at least one execution per Mazurkiewicz/Shasha-Snir trace. The first DPOR algorithm that explores one and only execution per Mazurkiewicz/Shasha-Snir trace is optimal dynamic partial order reduction (ODPOR) [3, 4]. ODPOR is then enhanced by the notion of *observer* [25] that considers a coarser equivalence class of executions than Mazurkiewicz/Shasha-Snir trace. The key idea of ODPOR with observers is to lazily build the happens-before relation. In fact, ODPOR with observers can generate exponentially fewer executions than Mazurkiewicz/Shasha-Snir traces in several cases. These ODPOR techniques have been implemented in the CONCUERROR tool [3, 25] for Erlang programs and the NIDHUGG [12, 2, 25] tool for C programs.

Later, we will see that Paper IV considers a weaker notion of traces than Mazurkiewicz/Shasha-Snir ones. To be more precise, instead of characterizing a trace by po, rf, and co as in Mazurkiewicz/Shasha-Snir traces, we define a trace just based on po and rf (more details will be given in Section 2.4). Indeed, in order to check the violation of some assertions, we only need to care about (i) the position of the assertions (i.e., the program order po) and (ii) the valuation of the assertions (i.e., the read-from relations rf) in traces. Removing the coherence order co in the definition of the trace can let us reduce the number of equivalence classes of executions and therefore reduce the number of explored executions necessary to check a software system.

Example 1.1 (Mazurkiewicz/Shasha-Snir traces)**.** Figure 1.3 shows a simple program with two processes, p_1 and p_2, that communicate through a shared variable x. The process p_1 (resp. p_2) writes to the location (a.k.a. variable) x and reads from it into a local register a (resp. b). We want to explore the set of possible executions of this program, e.g., to check whether the program can satisfy a $= 2$ and b $= 1$ upon termination. Under SC, there are six possible different executions of the program. None of them leads to the checking behavior. These executions fall into four equivalence classes, represented by the four possible Mazurkiewicz/Shasha-Snir traces τ_1, τ_2, τ_3, and τ_4 in Figure 1.3. A DPOR algorithm based on Mazurkiewicz/Shasha-Snir traces (e.g., [3, 12, 154, 2, 69]) must thus explore at least four executions. Meanwhile, ODPOR technique [3, 4] will generate exactly four executions (i.e., one and only one execution per Mazurkiewicz/Shasha-Snir trace). In this example, ODPOR with observers [25] will also generate four executions. □

1.2 Weak Memory Models

In this section, we start with the *sequential consistency* (SC) semantics, a commonly assumed memory model for software systems by programmers. After this, we consider *weak memory models*. A weak memory model allows the reordering of operations in order to improve performance. We give an introduction to three particular weak memory models that are addressed in this thesis, namely TSO, POWER, and Release-Acquire. Finally, we introduce *fences* which limit the reordering of operations and hence can be used to prevent some unexpected behaviors caused by weak memory models.

1.2.1 The SC Semantics

One of the most classical and intuitive memory models for software systems is *Sequential Consistency* (SC) [103]. Following the definition in [103], a concurrent system is sequentially consistent if:

1. *"the result of any execution is the same as if the operations of all processors were executed in some sequential order"*; and
2. *"the operations of each individual processor appear in this sequence in the order specified by its program"*.

The first criterion of the definition means that the execution of the system must follow some sequential order of the operations of all processes. Note that the order of operations for each process run on a processor is specified by the program order (po). Therefore, the second criterion means that the execution must also maintain the program order of the operations from each process.

Intuitively, we can understand that there is a *switch* between processes under the SC semantics. The switch randomly chooses a process, one at a time, and lets this process execute its operations atomically. When a process executes operations, it performs following the program order. After being executed, an operation is visible immediately to all other processes. Observe that SC is one of the few memory models (and maybe the only) that offers the possibility of making the operations immediately visible to all processes. Then, the switch repeats the same procedure until there are no more operations to be performed by the processes. Because of this reason, the SC semantics is also known as the *interleaving semantics* [63].

Sequential consistency is the most well-known semantics used by programmers. However, SC is often too strong to be used in order to obtain high performance for concurrent programs running in multicore machines [81]. In the next section, we will introduce weak memory models that are more relaxable and applicable to concurrent programs to provide better performance than the SC semantics.

1.2.2 The Need of Weak Memory Models

For *performance reasons*, modern architectures may reorder memory access operations of processes. This is due to complex buffering and caching mechanisms that make the response to memory queries (i.e., read operations) faster, and allow to speed up computations by parallelizing independent operations and computation flows. Therefore, operations may not be visible to all processes at the same time as in the case of the SC semantics. Moreover, they are not necessarily seen in the same order by different processes when they concern different memory locations. Roughly, modern architectures adopt weaker models (in the sense that they allow more behaviors) due to the relaxation in various ways of the program order. Examples of such weak models are TSO [140, 127] adopted in Intel x86 machines, POWER [112, 136, 23] adopted in PowerPC machines, and the ARM model [23, 70, 130] adopted in ARM machines.

Besides implemented by hardware, weak memory can also be seen in programming languages. In fact, programming-language compilers employ many optimizations that can introduce reorderings of memory accesses [115, 152, 138]. These optimizations are well-studied and successfully applied to single-thread programs where they are transparent to programmers. However, they can cause observable memory access reorderings when applied to concurrent programs and hence break the correctness of the programs [147, 43, 149, 138, 139]. Examples of such weak memory models in programming languages are Java [113] and C/C++11 [89, 101, 94, 99]. Many researchers have been studying the problems of checking the correctness of software systems under Java (e.g. [92, 24, 20]) and C/C++11 (e.g. [125, 126, 97, 148, 95, 146, 65, 80, 44]).

Below, we will give an introduction to three weak memory models that are addressed in this thesis: TSO, POWER, and Release-Acquire.

1.2.3 The TSO Semantics

Total Store Ordering (TSO) is a classical model corresponding to the relaxation adopted by Sun's SPARC multiprocessors [153] and to formalizations of the x86-TSO memory model [140, 127]. In TSO, a *store buffer* is inserted between each process and the main memory. The buffer behaves like an unbounded perfect (non-lossy) first-in-first-out (FIFO) channel that carries the pending store operations of the process.

Each process executes operations one by one according to the program's control flow, in the same way as under SC. When a process performs a *write* (store) operation, it does not immediately update the memory with the new value. Instead, the store is appended to the end of the process's store buffer. Nondeterministically, at any point in time, a store may be dequeued from the store buffer of any process. The memory is then updated according to the dequeued store. This is called an *update* operation. When a process executes a *read* (load) operation of a memory location (variable), it typically reads the value directly from memory, in the same way as under the SC semantics. However, if the same process has previously written to the same memory location and this store is still pending, then the load must read the value from the most recently enqueued store on that memory location, instead of reading from memory. This technique in TSO is known as *buffer-forwarding* or *read-own-write-early* (ROWE).

In the TSO semantics, the buffers allow stores to be arbitrarily delayed and thus reordered with subsequent loads by the same process. Because of this reason, TSO is said to allow $W \rightarrow R$ reordering, i.e., loads are allowed to overtake stores to different memory locations.

The unboundedness of the buffers implies that the state space of the system is infinite even in the case where the input program is finite-state. Therefore, checking programs running under the TSO memory model is a challenging problem. The decidability of the state reachability problem for programs running under the TSO semantics has been obtained by constructing equivalent models that replace the perfect store buffers by *lossy channels* [28, 27, 10].

The TSO and SC semantics are instances of *multi-copy-atomic* memory models where two different processes must observe the *effects* of two write operations to different memory locations from some other processes in the same order. Note that to the best of our knowledge, all memory models preserve the coherence property, meaning that all processes must agree on the same order of all write operations to the same variables. Later, we will see that POWER (more details will be given in Section 1.2.4) and Release-Acquire (more details will be given in Section 1.2.5) models are instances of *non-multi-copy-atomic* memory models. In contrast to multi-copy-atomic models, non-multi-copy-atomic models allow two different processes to observe the effects of two write operations to different memory locations from some other processes in different orders[1].

The TSO semantics is weaker than the SC semantics in the sense that it allows all behaviors of concurrent programs running under SC. Below, we give an example of a small concurrent program, known as `Dekker` or `Store Buffer` (`SB`). The program has a behavior that is forbidden under the SC semantics but allowed under the TSO semantics.

[1]This also can be represented as "two different threads may observe a store of a third thread at different times" as in [100] where *thread* has the same meaning as process.

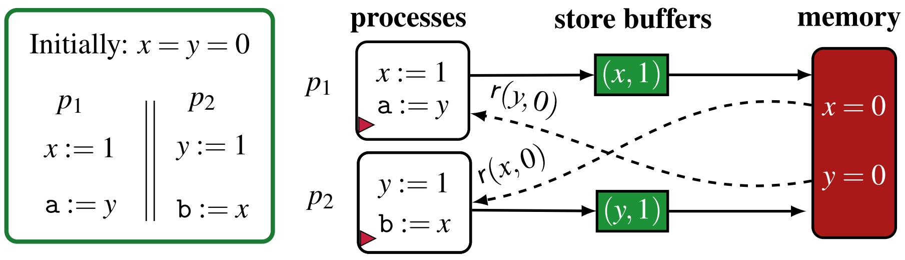

Figure 1.4. Dekker's mutual exclusion is broken under TSO.

Example 1.2 (TSO is weaker than SC)**.** The left part in Figure 1.4 presents a small program which contains two processes p_1 and p_2, accessing two shared variables x and y. The process p_1 stores to x and loads from y. The process p_2 symmetrically stores to y and loads from x. At the beginning of the program, all the variables are initialized to 0. After two processes have executed all their operations, we are interested in checking whether both processes p_1 and p_2 can load value 0 from variables y and x respectively.

Under the SC semantics (described in Section 1.2.1), there are three possible final states after executing this program. First, p_2 can execute entirely after p_1, in which case we would end up in a final state where p_1 has read the initial value 0 of y into its register a, and p_2 has read the value 1, written by p_1, from x into its register b. Symmetrically, we will end up in a different final state, where a holds the value 1, and b holds the value 0. Another possibility is that both stores $x := 1$ and $y := 1$ are executed in some order, before the loads a $:= y$ and b $:= x$. In this case, we will end up in a final state where both a and b hold the value 1.

Therefore, it is impossible, under SC, to end up in a state where both a and b hold the value 0. This is because, if a $:= y$ is executed before the store $y := 1$, then the load b $:= x$ must necessarily be executed after the store $x := 1$ and will therefore read the value 1 since $x := 1$ comes before a $:= y$, and b $:= x$ comes after $y := 1$.

However, under the TSO semantics, it is possible to observe a final state of the Dekker program, where both processes read the value 0. The right part in Figure 1.4 illustrates an execution of this program under TSO. First, p_1 enqueues the store $x := 1$ into its store buffer, and then it reads the value 0 of y from the memory. Symmetrically, p_2 enqueues the store $y := 1$ into its store buffer, and then it reads the value 0 of x from the memory. Finally, both processes dequeue and update their stores from buffers to the memory. $\square$

1.2.4 The POWER Semantics

While Sun's SPARC multiprocessors adopt the TSO memory model, PowerPC machines adopt POWER memory model [112, 136, 23]. Unlike TSO, POWER does not have any store buffer. Instead, each operation will be executed in three steps; namely, they are *fetched*, *initialized*, and then *committed*. Furthermore, a write operation may be *propagated* to other processes. Below, we explain how a process executes two main kinds of operations: the read and the write.

Roughly, when a process executes a write operation, it first needs to fetch the operation. In general, fetching all operations in a process is done following the program order (po). After being fetched, a write operation needs to be initialized. Initializing a write operation means that we evaluate all its expressions, including address expression and written-value expression. In other words, after the initializing step, we know precisely the variable that the operation will write to and the value that it will write. The next step after initializing is committing. Right after the commitment, the process will propagate the write operation to itself. Following this, the process is allowed to propagate the operation to other processes nondeterministically. Propagating a write operation to a process means that we will enable the process to observe and read from that write. The idea here is that POWER memory model keeps track of the coherence order (co) of all memory stores to each memory location (a.k.a. variable). A write operation is allowed to propagate to a process if the process has not observed another write that is after the propagating write in the coherence order. After a process receives a propagated write operation, its local view is updated to reflect that the recently propagated write is the most recent one it has observed.

In a similar way to the write, when a process executes a read operation, it first needs to fetch the read following the program order. After being fetched, the read operation needs to be initialized. Initializing a read operation means that we calculate which variable that the operation will read from and which value it will read. The initializing needs to satisfy the following mechanism. When a process p is going to read from a variable x, if the process has some own writes to the same variable that have not been committed, then p will get the value from the most recent uncommitted own write on that memory location. Otherwise, the process receives the value from the most recent propagated write operations (that can come from itself or other processes). This mechanism is a generalization of read-own-write-early (ROWE) technique in TSO. The final step in executing a read operation is committing. There is no propagation for a read.

It is important to note that the commitment of the write and read operations has to be done following some order constraints. These constraints are called *dependency relations*. They are *address dependencies*, *data dependencies*, and *control dependencies*. We will not go into detail to explain these dependencies in this introduction. We recommend readers to read Paper III [6, 7] for further information.

Because of the way the POWER semantics executes operations in several steps, POWER is said to allow all four basic kinds of memory access reorderings: $W \rightarrow W$, $W \rightarrow R$, $R \rightarrow W$, and $R \rightarrow R$. Furthermore, it allows the be-

```
Initially: x = y = 0

  p1     ||   p2
 x := 1  ||  a := y

 y := 1  ||  b := x
```

Figure 1.5. The MP program.

haviors associated with non-multi-copy-atomic memory models. Recall that non-multi-copy-atomic models allow two different processes to observe the effects of two write operations to different memory locations from some other processes in different orders (see more details in Example 1.4).

The POWER semantics is weaker than the SC semantics in the sense that it enables all behaviors of concurrent programs running under SC. Moreover, it is not clear how to compare POWER to TSO[2]. Below, we give an example of a small concurrent program, known as `Message Passing` (`MP`). The program has a behavior that is forbidden under the SC/TSO semantics but allowed under the POWER semantics. Then, we give a variation of the `MP` program where we will illustrate the behavior associated with non-multi-copy-atomic models.

Example 1.3 (POWER is weaker than SC and can be weaker than TSO)**.** Figure 1.5 presents a small program which contains two processes p_1 and p_2, accessing two shared variables x and y. Moreover, process p_2 has two registers `a` and `b`. At the beginning of the program, all the variables and registers are initialized to 0. Process p_1 has two write operations that set x and y to 1. Process p_2 loads the values of y and x into `a` and `b` respectively. After the two processes have executed all their operations, we are interested in checking whether the values of `a` and `b` are 1 and 0 respectively.

We observe that to satisfy this behavior, p_2 must read 1 from y; and while it is reading y, it should not see that x has been set to 1. Since at the beginning of the program, all variable values are 0, the value 1 for y read by p_2 must be written by process p_1.

This behavior is not observed (or allowed) under the SC/TSO semantics. In fact, under the SC semantics, the program order between the two write operations to x and y requires process p_1 to set x and y to 1 in order. As a consequence, when p_2 reads 1 from y, it must see that x has been set to 1. A similar reasoning can be applied to the case of the TSO semantics.

However, under the POWER semantics, it is possible to observe this behavior. First, p_2 fetches the two read operations from y and x. After that, it

[2]To the best of our knowledge, there is no formal comparison of the POWER and TSO semantics.

Initially: $x = y = 0$

p_1	p_2	p_3
$x := 1$	$\texttt{a} := y$	$\texttt{c} := x$
$y := 1$	$\texttt{b} := x$	$\texttt{d} := y$

Figure 1.6. A variation of the MP program shows a non-multi-copy-atomic behavior.

initializes the fetched operation $\texttt{b} := x$ and loads 0 from x into register b, and then commits the read. Next, p_1 fetches the write operation on x to initialize, commit, and propagate it to itself and p_2. Then, p_1 executes the write operation on y in a similar way. Finally, p_2 resumes its execution by initializing the fetched operation $\texttt{a} := y$ to load 1 from y that is the value just propagated from p_1, and then committing the read. □

Example 1.4 (Non-multi-copy-atomic behavior). Figure 1.6 presents a variation of the MP program which contains three processes p_1, p_2, and p_3. In a similar way to the MP program, process p_1 has two write operations that set x and y to 1, and process p_2 loads the values of y and x into a and b respectively. The remaining process p_3 loads the values of x and y into c and d sequentially. In Example 1.3, we have seen that it is possible to allow p_2 to get the value of a as 1 and the value of b as 0. It means that p_2 has seen the effect of the write $y := 1$ before the effect of the write $x := 1$. In this example, we show that POWER allows p_2 and p_3 to see the effects of the two writes in a different order. We note that this behavior cannot be observed in any instance of multi-copy-atomic memory models such as the SC/TSO semantics.

We show how the three processes perform their operations to allow the non-multi-copy-atomic behavior. First, we allow p_1 and p_2 to execute the same sequences of steps in a similar way to Example 1.3. Then, we allow p_1 to propagate the two writes to p_3. In the final step, p_3 fetches, initializes, and commits its two read operations. Since the two writes $x := 1$ and $y := 1$ have been propagated to p_3, p_3 can load value 1 from both x and y. Since the read operation $\texttt{c} := x$ is before $\texttt{d} := y$ in program order, we say that p_3 has seen the effect of the write $x := 1$ before the effect of the write $y := 1$. This order between the two write operations is different from the order of them that has been observed by p_2. □

1.2.5 The Release-Acquire Semantics

The *Release-Acquire* (RA) fragment [99] is a useful and well-behaved fragment of the C/C++11 memory model [89, 101, 94, 32] where all writes are release atomic accesses and all reads are acquire atomic accesses. The RA semantics strikes a good balance between performance and programmability. It allows high-performance implementations while still provides sufficiently strong guarantees for fundamental concurrent algorithms (such as the read-copy-update mechanism) [99]. Many researchers get interested with the RA semantics [99, 133, 93]. In the following, we will describe the axiomatic semantics of RA that is given in [99]. The axiomatic semantics will use the relations $\texttt{po}$, $\texttt{rf}$, and $\texttt{co}$ (defined in Section 1.1.2). Moreover, the semantics will use the *from-read* relation $\texttt{fr}$ that can be derived from the read-from relation $\texttt{rf}$ and the coherence order relation $\texttt{co}$. Formally, the *from-read* relation $\texttt{fr}$ is defined as the union of all $\texttt{fr}^x$ where $\texttt{fr}^x := (\texttt{rf}^x)^{-1};\texttt{co}^x$ denotes the composition of $(\texttt{rf}^x)^{-1}$ and $\texttt{co}^x$. Intuitively, the from-read relation connects each read with a write $\texttt{co}$-after the write from which the read takes its value.

We note that if one wants to formalize RA by using operational semantics as in the cases of SC, TSO, and POWER, he/she can use the operational one with timestamps [93]. The axiomatic and operational semantics are equivalent [93].

Following [99], we can explain the Release-Acquire semantics by describing the set of *observable executions* whose associated Mazurkiewicz/Shasha-Snir traces do not contain certain forbidden cycles. Formally, an execution of a system is observable (or consistent) under RA if the relation $(\texttt{po} \cup \texttt{rf} \cup \texttt{co}^x \cup \texttt{fr}^x)$ is acyclic for all variable x where $\texttt{po}$ is the program order, $\texttt{co}$ is the coherence order, $\texttt{rf}$ is the read-from relation, and $\texttt{fr}$ is the from-read relation of the associated Mazurkiewicz/Shasha-Snir trace of the execution.

Similarly to the POWER semantics, the RA semantics is also an instance of non-multi-copy-atomic memory models where two different processes may observe the effects of two write operations to different memory locations from some other processes in different orders. For example, it allows the behavior given in Example 1.4.

The RA semantics is weaker than the SC/TSO semantics in the sense that it allows all behaviors of concurrent programs running under SC/TSO [99]. It is not clear how to compare the RA semantics to the POWER semantics. Below, we give an example of a small concurrent program, known as `2+2W`. The program has a behavior that is forbidden under the SC/TSO semantics but allowed under the RA semantics. Then, we reuse the `MP` program to show a case where we observe that POWER can be weaker than RA.

Example 1.5 (RA is weaker than SC/TSO)**.** The left part in Figure 1.7 presents a small program which contains two processes p_1 and p_2, accessing two shared variables x and y. At the beginning of the program, all shared variables are initialized to 0. Process p_1 has two write operations that set x to 2 and y to 1. Symmetrically, process p_2 has two write operations that set y to 2 and x to 1. After the two processes have executed all their operations, we are interested in checking whether both the values of x and y can be 2.

Initially: $x = y = 0$

p_1	p_2
$x := 2$	$y := 2$
$y := 1$	$x := 1$

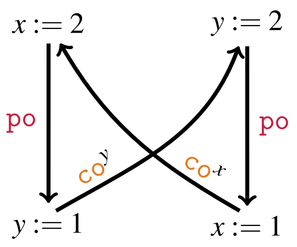

Figure 1.7. The 2+2W program (left) and a Mazurkiewicz/Shasha-Snir trace (right).

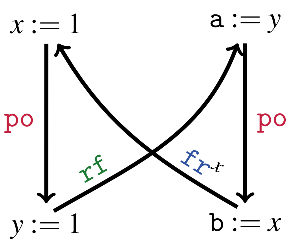

Figure 1.8. A Mazurkiewicz/Shasha-Snir trace of the MP program in Figure 1.5.

This behavior is not observed (allowed) under the SC/TSO semantics. In fact, under the SC semantics, to end up in a final state where the value of y is 2, the write $y := 1$ must be updated to the memory before the write $y := 2$. Then, since $x := 2$ must be updated to the memory before the write $y := 1$ (following $\mathtt{po}$) and $x := 1$ must be updated to the memory after the write $y := 2$ (following $\mathtt{po}$), the write $x := 2$ must be updated to the memory before the write $x := 1$. Therefore, in this final state, the value of x is 1 and not 2. A similar reasoning can be applied to the case of the TSO semantics.

However, under the RA semantics, it is possible to observe this behavior. The right part of Figure 1.7 gives the Mazurkiewicz/Shasha-Snir trace of any execution that leads to the behavior. Because there is no cycle in $(\mathtt{po} \cup \mathtt{co}^x)$ or $(\mathtt{po} \cup \mathtt{co}^y)$ (the relations $\mathtt{rf}$ and $\mathtt{fr}$ are empty in this example), this trace is observable (or allowed) under RA. □

Example 1.6 (POWER can be weaker than RA)**.** We reuse the MP program given in Figure 1.5 to show that the POWER semantics can be weaker than the RA semantics. Recall that in the MP program, we are interested in checking whether the value of $\mathtt{a}$ is 1 and the value of $\mathtt{b}$ is 0.

Although the interesting behavior in the MP program is observed under the POWER semantics (see Example 1.3), it is not allowed under RA. Figure 1.8 gives the Mazurkiewicz/Shasha-Snir trace of any execution that leads to the behavior. It is easy to see that to end up in a final state leading to the behavior, the read $\mathtt{a} := y$ must read from the write $y := 1$. Moreover, the read $\mathtt{b} := x$ must read from the initial write $x := 0$. Since the initial write to x takes place before any other write to x, there is a from-read ($\texttt{fr}$) relation from $\texttt{b} := x$ to $x := 1$. We see that there is a cycle in the relation $(\texttt{po} \cup \texttt{rf} \cup \texttt{co}^x \cup \texttt{fr}^x)$. Therefore, the trace is not observable under RA. □

In the next paragraph, we will introduce *fences* which forbid the unexpected behaviors of programs running under weak memory models. To be more precise, if programs are safe under SC but unsafe under a weak memory model, we can use fences to *correct* the programs under the weak memory model.

1.2.6 Fences in Weak Memory Models

In concurrent programs, it is common that multiple processes communicate using memory accesses (i.e., writes and read to the shared memory). This communication often follows some predefined orders by programmers. For example, in Dekker's mutual exclusion algorithm (see Figure 1.4) we have already seen that the correctness of the algorithm depends on the two writes $x := 1$ and $y := 1$ happening before the two reads $\texttt{a} := y$ and $\texttt{b} := x$ respectively.

To enforce such necessary orderings between memory accesses, hardware architectures or programming languages provide special *fence* instructions, which can be inserted by programmers when needed. Fence is also known as *memory barrier* [81]. Fence instructions from different memory models may follow different semantics and have different behaviors, but they all prohibit certain reorderings between memory access operations that are issued before and after the fences.

It is important to note that fences are *expensive* operations [82, 71, 110, 66]. Therefore, we usually need to *minimize* their use in software systems to obtain high performance. Later, we will see that Paper I proposes an efficient algorithm to find a minimal set of fences that guarantees the correctness of a concurrent program running under weak memory models (see more details in Section 2.1).

The TSO semantics supports only one kind of fences, called `mfence`. The `mfence` fence forces the execution of a process to be blocked until the store buffer of the process is empty. As a result, the fences in TSO will enforce the ordering between a write and any `po`-after read operation. In other words, a `mfence` operation ensures that all `po`-before write operations of the fence must be updated to the shared memory before the execution of all `po`-after read operations of the same fence.

The POWER semantics supports several kinds of fences: `sync`, `lwsync`, and `isync`. The `sync` fence is the strongest fence under POWER. It will enforce all four types of memory access orderings: (i) between a write and any `po`-after write, (ii) between a write and any `po`-after read, (iii) between a read and any `po`-after read, and (iv) between a read and any `po`-after write. Furthermore, the `sync` fence has a *cumulative* effect in the sense that when a process

```
Initially: x = y = 0

p1          ||   p2
x := 1      ||   y := 1
mfence      ||   mfence
a := y      ||   b := x
```

Figure 1.9. Dekker's mutual exclusion algorithm with fences inserted for correct operation under TSO.

p_1 executes a `sync` operation and later a write operation, all write operations (can be from any process p_2 and to any variable), which have been seen by p_1, must be observed by third processes p_3 before the later write operation of p_1. We will not go into detail to explain `lwsync` and `isync` fences in this introduction. We recommend readers to read Paper III [6, 7] for further information.

Unlike TSO and POWER semantics, the RA semantics does not have any specific instruction for fences. RA implements fences using *read-modify-write* (RMW) operations to an unused distinguished location [99]. Intuitively, an RMW operation reads the memory location and writes a new value into it simultaneously. We denote the fences under RA as `RAfence`.

Below, we provide an example to explain how fences can eliminate unexpected behaviors of programs running under weak memory models. In particular, we give a corrected version of the Dekker's mutual exclusion program (see Figure 1.4) under TSO. Then, we provide a variation of the `2+2W` program where we will explain how to use Mazurkiewicz/Shasha-Snir traces to reason about the behaviors of fences under the RA semantics.

Example 1.7 (Fencing Dekker for TSO)**.** To make Dekker's mutual exclusion program (given in Figure 1.4) work under the TSO memory model, we have to enforce orderings between the two writes $x := 1$ and $y := 1$ and the two reads `a` $:= y$ and `b` $:= x$ respectively. We do this by inserting one `mfence` instruction per process, between each of those pairs, as shown in Figure 1.9. □

Example 1.8 (Fences under RA)**.** Figure 1.10 presents a variation of the `2+2W` program (given in Figure 1.7) that contains `RAfence` operations under RA. We will explain how we can use Mazurkiewicz/Shasha-Snir traces to reason about the behaviors of fences under the RA semantics.

In a similar way to the `2+2W` program, the left part in Figure 1.10 presents a small program which contains two processes p_1 and p_2, accessing two shared variables x and y. The process p_1 has two write operations that set x to 2

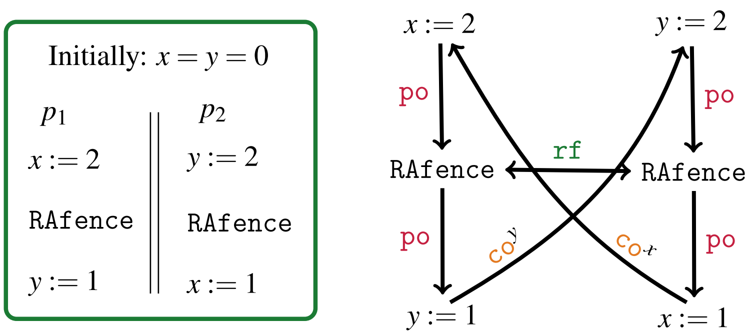

Figure 1.10. The 2+2W program with fences inserted under RA (left) and a Mazurkiewicz/Shasha-Snir trace (right).

and y to 1. Symmetrically, the process p_2 has two write operations that set y to 2 and x to 1. Notably, each process has a `RAfence` instruction between its two write operations. We are interested in checking the same behavior as in the `2+2W` program, i.e., after the two processes have executed all their operations, we check whether both the values of x and y can be 2. As explained in Example 1.5, this behavior is allowed in the `2+2W` program running under the RA semantics. We will show that the behavior is forbidden in the fence-inserted version.

The right part of Figure 1.10 presents the shape of Mazurkiewicz/Shasha-Snir traces of any execution that leads to the behavior. Since a `RAfence` operation is an RMW operation to an unused distinguished location, there must be `rf` and `co` relations between them. Observe that the relation `rf` between two RMW operations is coincident with their `co` relation. Therefore, for readability reasons, we only show the `rf` relation for `RAfence` operations. There are only two possible cases: (i) the `rf` relation is from the left `RAfence` operation to the right one, or (ii) the `rf` relation is from the right `RAfence` operation to the left one.

We consider the first case. The second case can be explained similarly. We see that there is a cycle in the relation $(\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co}^x \cup \mathtt{fr}^x)$. It starts from $x := 2$ to the fence of p_1, then to the fence of p_2, then to $x := 1$, then finally to $x := 2$. Therefore, the corresponding Mazurkiewicz/Shasha-Snir trace is not allowed under the RA semantics. As a consequence, the behavior is also forbidden under RA. □

1.3 Challenges in Model Checking for Weak Memory Models

Reasoning the behavior of software systems running under weak memory models (including TSO, POWER, and RA) is much more difficult than for systems running under the sequentially consistent memory. Indeed, checking a software system running under the TSO memory model poses a difficult challenge since the unboundedness of the store buffers implies that the state space of the system is infinite even in the case where the input system is finite-state. For an infinite-state system, a model checker cannot naively enumerate all possible behaviors because the model checking algorithm would never terminate. This is even worse for the case of the POWER and RA semantics because they might allow more behaviors than the TSO semantics [136, 99].

For the TSO semantics, the decidability of the state reachability problem has been obtained by constructing equivalent models that replace the perfect store buffers by *lossy channels* [28, 27, 10]. However, these constructions are complicated and involve several ingredients that lead to inefficient model checking algorithms. For instance, they require each message inside a lossy channel to carry (instead of a single store operation) a full snapshot of the memory representing a local view of the memory contents by the process. Furthermore, handling lossy channels requires guessing the contents of the channels (each channel corresponding to one process) and then validating the guessing on these channels [28]. Another way to handling lossy channels, that has been implemented in the MEMORAX tool [10, 11], is using one unified channel for all processes and explicitly using pointer variables (each corresponding to one process) inside the channel. The two approaches severely suffer the state-explosion problem. This leaves open the following question:

(**Q1**) *How to define more efficient model checking techniques for software systems running under the TSO semantics?*

Another question for model checking of concurrent programs running under the TSO semantics can be the followings:

(**Q2**) *How to efficiently find a set of a minimal set of fences that guarantees the correctness of a concurrent program running under the TSO semantics?*

Paper II and Paper I (see more details in Chapter 2) will respectively address the first and second questions.

Moreover, one interesting model checking problem for concurrent programs running under the TSO semantics is *parameterized systems*. Here, we consider software systems, e.g., mutual exclusion protocols, that consist of an *arbitrary* number of processes. Parameterized model checking aims to prove the correctness of the system regardless of the number of processes. The open questions for parameterized model checking of software systems running under TSO can be the followings:

(**Q3**) *Is the state parameterized reachability problem for concurrent programs running under TSO decidable?*

(**Q4**) *Can parameterized model checking techniques work well for concurrent programs running under TSO?*

(**Q5**) *Are these parameterized model checking techniques efficient compared to the existed model checking techniques for bounded-size instances?*

To the best of our knowledge, our result in Paper II is the first one to address question (**Q3**). Paper II also gives answers to questions (**Q4**) and (**Q5**) by performing experiments with our DUALTSO tool. Note that besides DUALTSO, new tools can handle parameterized model checking for concurrent programs running under TSO. One of them is CUBICLE-W that uses SMT solvers [56, 55, 57].

For the POWER semantics, it has been shown in [60] that the state reachability problem for POWER is undecidable (even for finite-state programs). Therefore, the question is be the followings:

(**Q6**) *How to define approximate techniques to efficiently analyze software systems running under the POWER semantics for which we can obtain the decidability of the state reachability problem?*

Necessarily, such methods should be based on over or under-approximate analyses. Paper III (see more details in Chapter 2) addresses this question using context-bounded model checking [132] by extending the work in [26, 29] to the case of programs running under POWER. Note that for the case of finite, acyclic, and deterministic programs, the question can also be addressed by applying stateless model checking techniques. For example, an SMC technique for concurrent programs running under the POWER semantics has been proposed in [12] and implemented in the NIDHUGG tool [2, 12].

For the RA semantics, there are still no results about the decidability of the state reachability problem. For the subclass of this problem where we consider only finite, acyclic, and deterministic systems, it is quite straightforward to see that the state reachability is decidable. For this subclass, there are some results about using stateless model checking to obtain efficient model checking algorithms [97, 126, 125]. These techniques will explore all behaviors of a software system by examining each equivalent class of executions at least once. Therefore, the questions for model checking of software systems running under the RA semantics can be the followings:

(**Q7**) *Is the state reachability problem for concurrent programs running under the RA semantics decidable?*

(**Q8**) *Is it possible to obtain more efficient stateless model checking techniques for software systems running under the RA semantics?*

Paper IV addresses question (**Q8**) by defining a novel SMC technique based on a new notion of equivalent classes of executions (see more details in Chapter 2).

2. Summary of Contributions

In this chapter, we give a short overview of our four peer-reviewed papers. We will explain the problem addressed by each paper and its main contributions.

2.1 Paper I: The Best of Both Worlds: Trading Efficiency and Optimality in Fence Insertion for TSO

An essential problem in model checking of software systems running under weak memory models is to effectively find a set of fences that ensures program correctness. Manual fence placement is time-consuming and error-prone due to complex behaviors introduced by the memory models. The challenge then is to develop methods for *automatic fence placement*. A fence insertion algorithm requires an underlying model checking algorithm that checks the correctness of the program for a given set of fences. This is necessary to be able to decide whether the current set of fences is sufficient, or whether additional fences are needed to achieve correctness.

Designing such an algorithm is hard since we face a crucial trade-off between two criteria, namely:

- *Efficiency*: The algorithm needs to be able to carry out efficient analysis of complex program behaviors that arise due to intricate reorderings of program operations. This complexity is for instance reflected by the fact that standard operational definitions for weak memory models [127, 140] use *unbounded store buffer* semantics, thus giving rise to an infinite-state space even in the case where the original program is finite-state. Furthermore, since we are dealing with concurrent programs, the algorithm should scale well when we increase the number of processes and the number of variables.
- *Optimality*: The algorithm should derive sets of fences that are as close to optimal as possible. More precisely, we are required to prevent *under-fencing* (i.e., inserting too few fences) since this would result in unsound program behaviors. We also need to avoid *over-fencing* (i.e., adding too many fences) since this would result in a degradation of the program performance (see e.g., [21, 71, 110, 1, 66], for descriptions of the high cost of fences on CPU-intensive concurrent programs).

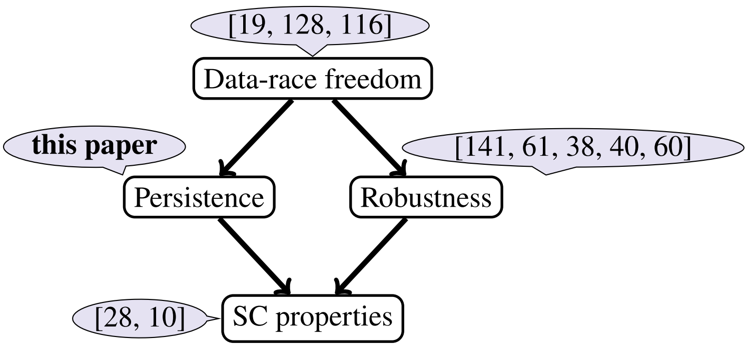

Figure 2.1. Correctness criteria.

In this context, identifying *good correctness properties* is crucial since a given property represents a particular choice in the trade-off between efficiency and optimality. Figure 2.1 shows some correctness criteria ordered according to their strength. We recommend readers to read Paper I for more information about them. At one extreme, we may require that the program is *data-race-free* [19, 128, 116]. Checking the data-race-free condition can be performed efficiently (e.g., [66]), but it will cause over-fencing, hence failing the optimality criterion. In fact, some data races are in reality not harmful. For example, two racy implementations of a work-stealing queue [119, 108] perform well under TSO without requiring fences. At the other extreme, we may consider *SC properties* such as safety and liveness properties. This would result in smaller sets of fences than in the previous case, but the model checking problem becomes significantly harder or even undecidable [28, 27], thus failing the efficiency criterion. In fact, it has been shown that checking safety properties for a finite-state program running under TSO is decidable but has a *non-primitive recursive* complexity [28, 10]. Furthermore, the verification problem becomes even undecidable when considering liveness property [28]. Another correctness criterion is *Robustness* (also known as stability) [141, 61, 38, 40, 60]. A program is robust under a weak memory model if it generates the same set of Mazurkiewicz/Shasha-Snir traces under the memory and the SC semantics. Robustness is weaker than SC safety properties but stronger than data-race-free criterion. It has been studied for different memory models such as TSO, PSO, and POWER [141, 61, 38, 40, 60]. An algorithm for checking Robustness for concurrent programs running under TSO is implemented in the TRENCHER tool [38].

In this paper, we present an approach for automatic fence insertion in concurrent programs running under the TSO memory model that gives a good trade-off between efficiency and optimality. To this end, we make the following contributions.

1. *Persistence*: We introduce a novel notion of correctness, called *Persistence*, that, as demonstrated by our experimental data, provides the best-known trade-off between efficiency and optimality. Persistence considers the traces of a program and extracts two parameters, namely:
   a) *program order*: the order in which instructions are executed within the same process; and
   b) *store order*: the order in which different write operations hit the shared memory. Note that the store order here is a total order on all writes to *all* shared variables.

   The program is deemed to be persistent if it generates the same program and store orders under the TSO and SC semantics. If a program is persistent then it reaches identical sets of configurations (where a configuration is a global state of the program) under TSO and SC. In particular, if the program is correct w.r.t. a given safety property under SC, then it will also be correct w.r.t. the same safety property under TSO.

   In a similar way to Robustness, Persistence is weaker than SC safety properties but stronger than data-race-free criterion. Moreover, Persistence is *incomparable* to Robustness. Both Persistence and Robustness are based on comparing the set of traces of a concurrent program running under a weak memory model and the SC semantics. Persistence can be weaker than Robustness in the sense that Persistence does not require the read-from relations (`rf`) in its definition of traces. Meanwhile, Robustness can be weaker than Persistence in the sense that Robustness replaces the store order to all variables in Persistence by coherence order `co` (see Section 1.1.2). Recall that coherence order `co` only totally orders the write operations to each memory location. We recommend readers to read Paper I for further comparisons between Persistence, Robustness, and other correctness criteria.
2. *Pattern*: We present an algorithm that automatically reduces the problem of checking Persistence to the problem of checking whether a given program exhibits a specific pattern when running under SC. The proof of correctness of the reduction is highly non-trivial and relies on intricate definitions of different types of runs that may be performed by the program. Despite the high complexity of the proof, the definition of the pattern is extremely simple. Crucially, from the efficiency point of view:
   a) We need only to perform one state reachability query on a single target program, and
   b) There is no explosion in the size of the target program since it contains the same number of processes and only two extra variables compared to the original program (regardless of the number of processes).

   After reducing the checking Persistence to the state reachability analysis of an SC program, we can use any existing model checking tool for concurrent programs running under SC. We use SPIN [83] in the current implementation of our tool.

3. *Fence Insertion*: We present an algorithm that produces a *minimal set* of fences needed to ensure the Persistence of a given program. The set is minimal in the sense that removing any fence in this set makes the program non-persistent. The algorithm is counterexample-guided, i.e., it uses counterexamples generated by the Persistence checking algorithm to infer the minimal needed set of fences to remove all the counterexamples.
4. *Abstraction*: We present a general abstraction framework that is compatible with the notion of Persistence, in the sense that Persistence of the abstract program implies Persistence of the concrete program. We instantiate the framework by defining different abstraction functions that allow reducing both the number of variables and processes, thus significantly limiting the state-explosion problem, and allowing scalability to large numbers of processes.
5. *Tool*: We have implemented an open-source and publicly available tool, called PERSIST, that we use to evaluate our framework on a wide range of benchmarks. PERSIST uses SPIN as a backend tool for checking state reachability queries for concurrent programs running under SC. Since SPIN runs on finite-state programs, our experiments are carried out only on such programs. We carry out an extensive comparison with state-of-the-art tools, such as TRENCHER [38], MEMORAX [10], REMMEX [111], and MUSKETEER [21]. Our data show that Persistence indeed provides a good trade-off between efficiency and optimality. More precisely, PERSIST can perform fence insertion efficiently on all the benchmarks without inserting large numbers of unnecessary fences.

2.2 Paper II: The Benefits of Duality in Verifying Concurrent Programs under TSO

In this paper, we address the problem of model checking *safety properties* of concurrent programs running under the TSO semantics. Recall that checking programs running under the TSO memory model is a challenging problem (see Section 1.3).

We introduce an alternative (yet equivalent) semantics to the classical TSO semantics that is more amenable to efficient algorithmic model checking and for the extension to parameterized systems. To this end, we make the following contributions.

1. *The Dual TSO semantics*: We introduce the *Dual TSO semantics* which is an alternative (yet equivalent) semantics to the classical TSO semantics, in the sense that any given set of processes will reach the same set of local states under both semantics.

In the Dual TSO semantics, we use a *dual view* where *load buffers* are used instead of store buffers. The main idea is to use load buffers that will store pending load operations (more precisely, values that will potentially be taken by future load operations) rather than store buffers (that contain store operations). The flow of information will now be in the reverse direction, i.e., store operations are performed by processes atomically on the main memory, while values of variables are propagated non-deterministically from the memory to the load buffers of the processes. When a process performs a load operation, it can fetch the value of the variable from the head of its load buffer.
The Dual TSO semantics allows us to understand the reachability problem of concurrent programs running under the TSO semantics in a totally different perspective. Furthermore, it offers several significant advantages for formal reasoning and program model checking as follows.

   a) *Simplify the analysis of the state reachability problem under TSO*: The Dual TSO semantics allows transforming the load buffers to lossy channels without adding the costly overhead that was necessary in the case of store buffers. This means that we can assume w.l.o.g. that any message in the load buffers (except a finite number of messages) can be lost in a non-deterministic manner. Hence, we can apply the theory of well-structured transition systems [16, 67] in a straightforward way that leads to a much simpler proof of the decidability of the state reachability problem. The absence of extra overhead means that we obtain more efficient algorithms and better scalability (as shown by our experimental results).
   b) *Extend the decision procedure easily to the parametric case*: Dual TSO allows extending the well-structured transition system [16, 67] to perform model checking of *parameterized systems*.
   It is not obvious how to perform model checking for parameterized systems running under the TSO semantics. For instance, extending the framework of [10], would involve an unbounded number of pointer variables, thus leading to channel systems with unbounded message alphabets. In contrast, as we show in this paper, the pure nature of the Dual TSO semantics allows a straightforward extension of our model checking algorithm to the case of parameterized model checking. This is the first time a decidability result is established for model checking of parameterized systems running under the TSO semantics. Notice that this result takes into account *two* sources of infinity: the number of processes and the size of the buffers.

2. *Tool*: Based on our framework, we have implemented a tool, called DUALTSO and applied it to a broad set of benchmarks. The experiments demonstrate the efficiency (in running time) of DUALTSO compared to the MEMORAX tool [10, 11] by two orders of magnitude in average. For

more information, MEMORAX is also a sound and complete tool that can check the correctness of concurrent programs running under the TSO semantics. The experiments also show the feasibility of parameterized model checking for concurrent programs running under the Dual TSO semantics. In fact, besides its theoretical generalization, parameterized model checking is practically crucial in this setting: as our experiments show, it is much more efficient than checking bounded-size instances (starting from three or four processes) in running time and memory consumption.

2.3 Paper III: Context-Bounded Analysis for POWER

Apprehending the effects of all weak behaviors allowed in weak memory models is extremely hard. In Papers I–II, we have seen that TSO enables reordering stores to past loads (of different variables), which reflects the use of store buffers. POWER allows most of the possible reordering between store and load operations (see Section 1.2.4). Much work has been devoted to formalizing memory models that captures the program semantics corresponding to the models such as TSO [140, 127] and POWER [136, 135, 23, 112]. Still, programming against weak memory models is a hard and error-prone task. Therefore, developing model checking approaches under weak memory models is of paramount importance. In particular, it is crucial in this context to have efficient algorithms for automatic bug detection.

This paper presents an algorithmic approach for checking the state reachability problem for concurrent programs running under the POWER semantics. To this end, we make the following contributions.

1. *Context-bounded model checking for POWER*: Our approach consists of introducing a parametric under-approximation schema in the spirit of context bounding [132, 121, 102, 98, 26, 29]. *Context-bounded model checking* has been proposed in [132] as a suitable approach for efficient bug detection in concurrent programs. Indeed, it has been shown experimentally that concurrency bugs usually show up after a small number of context switches [123].
   In the context of weak memory models, context-bounded model checking has been extended in [26, 29] to the case of programs running under TSO. The work we present here aims at extending this approach to the case of POWER. This extension is challenging due to the complexity of POWER and therefore requires developing new techniques that are different from, and much more involved than, the ones used in the case of TSO. To that aim:
   a) We introduce a *new concept* of context that is suitable for POWER. Intuitively, the architecture of POWER is similar to a distributed system with a replicated memory, where each process has its own

replica, and where operations are propagated between replicas according to some specific protocol. Our definition of context is based on this architecture. Therefore, we consider a context as an execution segment for which there is precisely one *active* process. All actions within a context are either operations issued by the active process or propagation actions performed by its memory system. An execution is called *k-bounded* if it can be divided into a sequence of at most k contexts.

b) We introduce a *code-to-code translation* that helps to analyze a concurrent program running under the POWER semantics by examining a corresponding translated concurrent program running under the SC semantics. In our analysis, we consider only executions that are bounded by a k-bounded number of contexts. Notice that while we bound the number of contexts in an execution, we do not put any bound on the lengths of the contexts, nor on the size of the memory system.
We prove that for every bound k, and for every concurrent program P, the code-to-code translation returns another concurrent program P' such that for every k-bounded execution in P running under the POWER semantics there is a corresponding k-bounded execution of P' running under the SC semantics that reaches the same state and vice-versa. Thus, the context-bounded state reachability problem for P can be reduced to the context-bounded state reachability problem for P' under SC.
We show that the program P' has the same number of processes as P plus two additional processes, and only $O(|\mathscr{P}| \cdot |\mathscr{X}| \cdot k + |\mathscr{R}|)$ additional shared variables and local registers compared to P, where $|\mathscr{P}|$ is the number of processes, $|\mathscr{X}|$ is the number of shared variables, and $|\mathscr{R}|$ is the number of local registers in P. Furthermore, the obtained program P' has the same type of data structures and variables as the original one P. As a consequence, we obtain for instance that for finite-data programs, the context-bounded model checking for concurrent programs running under the POWER semantics is decidable. Moreover, our code-to-code translation allows leveraging existing verification tools for concurrent programs to carry out verification of safety properties under POWER.

2. *Tool*: To show the applicability of our approach, we have implemented our technique in a prototyping tool, namely POWER2SC. We have used CBMC version 5.1 [53] as the backend tool for solving the SC state reachability problem. We have carried out several experiments showing the efficiency of our approach. More precisely, we compare POWER2SC to GOTO-INSTRUMENT [22] and NIDHUGG [2, 12]. GOTO-INSTRUMENT implements a bounded model checking technique for concurrent programs running under then POWER semantics. Meanwhile, NIDHUGG

implements a stateless model checking algorithm based on the equivalent classes of Mazurkiewicz/Shasha-Snir traces. Our experimental results confirm the assumption that concurrency bugs manifest themselves within small bounds of context switches in many benchmarks. They also prove that POWER2SC can be more efficient (in running time) and scalable than GOTO-INSTRUMENT and NIDHUGG in several cases.

2.4 Paper IV: Optimal Stateless Model Checking under the Release-Acquire Semantics

Ensuring correctness of concurrent programs is difficult since one must consider all the different ways in which processes can interact. A successful technique for finding concurrency bugs (i.e., defects that arise only under some process schedulings), and for verifying their absence, is *stateless model checking* (SMC) [75] (see Section 1.1.2). SMC faces the problem that the number of possible process schedulings grows exponentially with the length of program execution, and must, therefore, be equipped with techniques to reduce the number of explored executions. The most prominent one is *partial order reduction* [150, 129, 72, 49], adapted to SMC as *dynamic partial order reduction* (DPOR). DPOR was first developed for concurrent programs that execute under the standard sequential consistency (SC) model [69, 137, 3]. In recent years, DPOR has been adapted to hardware-induced weak memory models, such as TSO and PSO [2, 154], and language-level concurrency models, such as the C/C++11 memory model [125, 126, 97]. DPOR is based on the observation that two executions can be regarded as equivalent if they induce the same ordering between conflicting operations (see Section 1.1), and therefore it is sufficient to explore at least one execution in each equivalence class. Under SC, such equivalence classes are called *Mazurkiewicz traces* [117]; for weak memory models, the natural generalization of Mazurkiewicz traces are called *Shasha-Snir traces* [141] (see Section 1.1.2).

As an illustration, Figure 2.2 shows a simple program with two processes, p_1 and p_2, that communicate through a shared variable x. The process p_1 (resp. p_2) writes to the variable and reads from it into a local register $\mathtt{a}$ (resp. $\mathtt{b}$). We want to explore the possible executions of this program, e.g., to check whether the program can satisfy $\mathtt{a} = 2$ and $\mathtt{b} = 1$ upon termination. Under many memory models, including SC, TSO, PSO, RA, and POWER, executions of the program in Figure 2.2 fall into four equivalence classes, represented by the four possible Mazurkiewicz/Shasha-Snir traces τ_1, τ_2, τ_3, and τ_4 in Figure 2.2[1]. A DPOR algorithm based on Mazurkiewicz/Shasha-Snir traces (e.g., [3, 12]) must thus explore at least four executions. However, it is possible to reduce

[1] A Mazurkiewicz/Shasha-Snir trace also includes operations corresponding to initial values of the variables, but these can be ignored for this example.

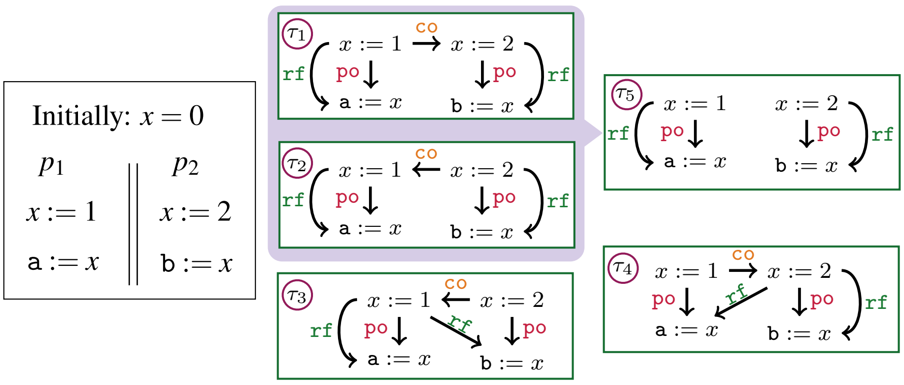

Figure 2.2. A simple concurrent program (left) and Mazurkiewicz/Shasha-Snir traces and weak traces (right).

this number further. Namely, a closer inspection reveals that the two traces τ_1 and τ_2 are equivalent, in the sense that each process goes through the same sequences of local states and computes the same results. This is because τ_1 and τ_2 have the same program order (po) and read-from (rf) relation. Their only difference is how writes are ordered by co, but this is not relevant for the computed results.

The preceding example illustrates that there is a potential for improving the efficiency of DPOR algorithms by using a *weaker equivalence*, namely *weak trace*, induced only by po and rf. In this example, the improvement is modest (reducing the number of explored traces from four to three), but it can be significant, sometimes even exponential, for more extensive programs. Several recent DPOR techniques try to exploit the potential offered by such a weaker equivalence [126, 125, 85, 86, 45]. However, except for the minimal case of an acyclic communication graph [45], they are far from optimally doing this, since they may still explore a significant number of different executions with the same rf relations. Therefore, the challenge remains to define an optimal and efficient DPOR algorithm based on weak traces. The optimality imposes that we need to explore *exactly* one execution for each weak trace (that is characterized by the po and rf relations).

In this paper, we present a fundamentally new approach to define DPOR algorithms, which optimally explores only the equivalence classes defined by the program order and read-from relations. Our method is developed for the RA semantics [99]. For more details, we make the following contributions.

1. *Saturated semantics*: A significant challenge for our DPOR algorithm is to efficiently determine all continuations of a currently explored trace that lead to some RA-consistent trace. E.g., for the program in Fig. 2.2, letting both processes read from the write of the other process leads to RA-inconsistency. We solve this problem by defining a *saturation* operation which extends a weak trace with a *partial* coherence relation. The

extended trace contains precisely coherence edges that must be present in any corresponding Mazurkiewicz/Shasha-Snir trace.

2. *DPOR algorithm*: Our DPOR algorithm maintains a saturated version of the currently explored weak trace, and can, therefore, examine *precisely* those weak traces that are RA-consistent, without performing useless explorations. When a read operation is added, the algorithm determines the set of write operations from which it can obtain its value while preserving the RA-consistency, and branches into a separate continuation for each such write operation. When a write operation is added, the algorithm merely adds it to the trace. It can be proven that this preserves the RA-consistency, and also keeps the trace saturated (a slight modification is needed for atomic read-modify-write operations). The algorithm must also detect if some previous read may read from a newly added write, and then backtrack to allow the write to be performed before that read.
   We prove that our DPOR algorithm is *optimal* for weak traces, in the sense that:
   a) Any exploration eventually leads to a terminated RA-consistent execution, i.e., the algorithm never blocks because it discovers that it is about to perform redundant or wasted explorations.
   b) Each RA-consistent weak trace is explored precisely once.

   Moreover, our DPOR algorithm spends *polynomial time* for each generated execution.
   Our saturation technique and the DPOR algorithm presented in this paper can be extended to cover atomic read-modify-write (RMW) operations and locks. This extension also satisfies the same strong optimality results (in (a) and (b)) and the same polynomial time complexity per execution.
3. *Tool*: We have implemented our saturation operation and the DPOR algorithm in a tool, called TRACER, and applied it to many challenging benchmarks. We compare our tool with other state-of-the-art stateless model checking tools running under the RA semantics, namely CDSCHECKER [125, 126] and RCMC [97]. The experiments show that TRACER always generates optimal numbers of executions w.r.t. weak traces in all benchmarks. On many benchmarks, this number is much smaller than the ones produced by the other tools. The results also show that TRACER has better performance (in running time) and scales better for more extensive programs, even in the case where it explores the same number of executions as the other tools.

3. Conclusions and Directions for Future Work

In this thesis, we have considered model checking of software systems running under weak memory models.

First, we have introduced the background knowledge relevant to understanding our four peer-reviewed papers. We have shown the importance and the difficulty of finding software failures in light of the evolution of the use of computer systems over recent decades, through testing and model checking. For model checking concurrent programs, we have looked particularly into state-space exploration and stateless model checking techniques. We have examined the behaviors caused by the SC, TSO, POWER, and RA memory models, and seen several examples that motivate these models. Finally, we have discussed some challenges of model checking of software systems running under weak memory models.

In the four peer-reviewed papers, we introduce a number of new techniques for analyzing concurrent programs running under weak memory models.

In Paper I, we present a method for automatic fence insertion in concurrent programs running under the TSO semantics [140, 127] that provides the best-known trade-off between efficiency and optimality. The method can efficiently handle complex aspects of program behaviors such as unbounded buffers and large numbers of processes. Furthermore, it can find small sets of fences needed for ensuring the correctness of the program. In conclusion, we propose a novel notion of correctness, called *Persistence*, that compares the behavior of a concurrent program running under the weak memory semantics with that running under the traditional sequential consistency (SC) semantics [103]. We give an algorithm that reduces the fence insertion problem under TSO to the state reachability problem for programs running under SC. Furthermore, we provide an abstraction scheme that substantially increases scalability to large numbers of processes. Based on our method, we have implemented the PERSIST tool and run it successfully on a wide range of benchmarks.

In Paper II, we address the problem of model checking safety properties of concurrent programs running under the TSO memory model. Known decision procedures for this model are based on complex encodings of store buffers as lossy channels [28, 27, 10]. These procedures assume that the number of processes is fixed. However, it is essential in general to prove the correctness of a system in a parametric way with an arbitrarily large number of processes. In this paper, we introduce an alternative (yet equivalent) semantics, namely the *Dual TSO semantics*, to the classical TSO model that is more amenable for model checking of concurrent systems and can be extended to model checking of parameterized systems. For that, we adopt a dual view where load buffers are used instead of store buffers. The flow of information is now from the memory to load buffers. We show that this new semantics allows (i) to simplify drastically the analysis of the state reachability problem under TSO and (ii) to extend the decision procedure easily to the parametric case. Indeed, the Dual TSO semantics allows us to simplify the decidability proof of the state reachability problem for concurrent programs Moreover, the Dual TSO semantics also allows us for the first time obtain a parameterized model checking algorithm that is more general and more efficient in practice than the one for bounded instances (from three or four processes).

In Paper III, we propose an *under-approximate state reachability analysis* algorithm for programs running under the POWER memory model [112, 136, 23], in the spirit of the work on context-bounded model checking initiated by Qadeer et al. [132] for detecting bugs in concurrent programs (supposed to be running under the classical SC model). To conclude, we first introduce a new notion of context that is suitable for reasoning about executions under POWER, which generalizes the one defined in [26, 29] for the TSO memory model. Then, we provide a polynomial size reduction of the context-bounded state reachability problem under POWER to the same problem under SC: Given an input concurrent program P, our method produces a concurrent program P' such that, for a fixed number of context switches, running P' under SC yields the same set of reachable states as running P under POWER. The generated program P' contains the same number of processes as P and operates on the same data domain. By leveraging the existing model checker CBMC [53], we have implemented the POWER2SC tool and successfully run it on a wide range of of benchmarks.

In Paper IV, we present a framework for the efficient application of *stateless model checking* (SMC) to concurrent programs running under the Release-Acquire (RA) fragment [99] of the C/C++11 memory model [89, 101, 94, 32]. Our approach is based on *weak traces*. It explores the possible program orders, which define the order in which instructions of a thread are executed, and read-from relations, which specify how reads obtain their values from writes. This is in contrast to previous approaches, which also explore the possible coherence orders, i.e., orderings between conflicting writes. Since unexpected test results such as program crashes or assertion violations depend only on the read-from relation, we avoid a potentially significant source of redundancy. Our framework is based on a novel technique for determining whether a particular read-from relation is feasible under the RA semantics. We define an SMC algorithm which is *optimal* in the sense that it explores each program order and read-from relation exactly once. This optimality result is strictly stronger than previous analogous optimality results, which also take the coherence order into account. Finally, we have implemented our framework in the TRACER tool. Experiments show that TRACER can be significantly faster than state-of-the-art tools that can handle the RA semantics in many benchmarks.

Future work.
In Paper I, there are several open and hard questions to consider in the future. This includes studying three possible essential correctness criteria with different strength (c.f. Figure 2.1). The first criterion is PO&CO where a program is deemed to be correct if the traces of the program running under TSO and SC agree on (i) program order `po` and (ii) coherence order `co`. The second criterion is PO&RF where a program is deemed to be correct if the traces of the program running under TSO and SC agree on (i) program order `po` and (ii) read-from relations `rf`. This criterion is related to our weak traces (see Section 2.4). Both PO&CO and PO&RF give entirely different and weaker correctness conditions compared to Persistence or Robustness [141, 61, 38, 40, 60]. Checking PO&CO and PO&RF (e.g., through finding appropriate patterns) are important and challenging open problems. The third open question is checking the condition PO, a weakening of both PO&CO and PO&RF, where the program is considered if its TSO and SC traces need only to agree on program order. Finally, it is crucial to develop frameworks that allow checking the different stability conditions for other weak memory models, as done in [23].

In Paper II, the future work consists of applying our model checking techniques of safety properties to other memory models and combining with predicate abstraction to handle programs with unbounded data domain.

In Paper III, one possibility is to extend our framework to cover other models such as ARM [70, 131, 130], Java [113], and C/C++11 [89, 101, 94, 32]. For instance, for ARM memory model, it is an open question to see the efficiency of context-bounded model checking for concurrent programs running under ARM and to compare it with other verification techniques [58, 79]. It is also important to consider other under-approximation techniques, and in particular to examine notions of context that are different from the one we have used in the paper.

In Paper IV, although we only consider the RA semantics, we believe that our approach is general and can be extended to other memory models. For instance, we can extend the approach to the SRA (Strong RA) semantics [99] by a modification of the sets of readable and visible operations. It is interesting to check whether we can also handle the relaxed fragment of C/C++11 [89, 101, 94, 32] by employing a swapping mechanism for event speculations similar to the one we have proposed in this paper for treating postponed write operations. For memory models such as SC [103], our saturation scheme is necessarily not complete, since saturation for such models amounts to solving an NP-complete problem [45]. However, we believe that by maintaining saturated traces (as we do in this paper) and only running the costly operations mentioned in [45] "by demand", we can substantially improve efficiency even under the SC semantics. Another interesting direction for future work is to use an even weaker relation than weak traces (e.g., the ones in [85, 86]), while still maintaining a polynomial time complexity for each explored execution. Finally, while most existed SMC techniques (e.g. [75, 69, 3, 134, 122, 125, 126]) might be hard to parallelize, our DPOR algorithm is possible to be adapted for massive parallelism.

References

[1] Trencher webpage. http://concurrency.cs.uni-kl.de/trencher.html. 2018-09-03.

[2] P.A. Abdulla, S. Aronis, M. Faouzi Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. In *TACAS 2015*, volume 9035 of *LNCS*, pages 353–367, 2015.

[3] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *POPL 2014*, pages 373–384, 2014.

[4] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64(4):25:1–25:49, 2017.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. The benefits of duality in verifying concurrent programs under TSO. In *CONCUR 2016*, pages 5:1–5:15, 2016.

[6] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. Context-bounded analysis for POWER. In *TACAS 2017*, pages 56–74, 2017.

[7] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. Context-bounded analysis for POWER. *CoRR*, abs/1702.01655, 2017.

[8] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. A load-buffer semantics for total store ordering. *LMCS 2018*, Volume 14, Issue 1, 2018.

[9] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. Replacing store buffers by load buffers in TSO. In *VECoS 2018*, pages 22–28, 2018.

[10] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under TSO. In *TACAS 2012*, volume 7214 of *LNCS*, pages 204–219, 2012.

[11] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In *TACAS 2013*, pages 530–536, 2013.

[12] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In *CAV 2016*, volume 9780 of *LNCS*, pages 134–156, 2016.

[13] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the Release-Acquire semantics. In *OOPSLA 2018*, 2018.

[14] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. Precise and sound automatic fence insertion procedure under PSO. In *NETYS 2015*, pages 32–47, 2015.

[15] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Ngo Tuan Phong. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *ESOP 2015*, pages 308–332, 2015.

[16] Parosh Aziz Abdulla, Karlis Čerāns, Bengt Jonsson, and Tsay Yih-Kuen. General decidability theorems for infinite-state systems. In *LICS 1996*, pages 313–321, 1996.

[17] Parosh Aziz Abdulla and Giorgio Delzanno. Parameterized verification. *STTT 2016*, 18(5):469–473, 2016.

[18] Michael Abrash. *Michael Abrash's Graphics Programming Black Book, with CD: The Complete Works of Graphics Master, Michael Abrash*. Coriolis Group Books, 10th edition, 1997.

[19] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE TPDS*, 1993.

[20] Elvira Albert, Anindya Banerjee, Sophia Drossopoulou, Marieke Huisman, Atsushi Igarashi, Gary T. Leavens, Peter Müller, and Tobias Wrigstad. Formal techniques for Java-Like programs. In *ECOOP 2008*, pages 70–76, 2008.

[21] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. In *CAV 2014*, 2014.

[22] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *ESOP 2013*, pages 512–532, 2013.

[23] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *TOPLAS 2014*, 36(2):7:1–7:74, 2014.

[24] Afshin Amighi, Pedro de Carvalho Gomes, Dilian Gurov, and Marieke Huisman. Provably correct control flow graphs from Java bytecode programs with exceptions. *STTT 2016*, 18(6):653–684, 2016.

[25] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In *TACAS 2018*, pages 229–248, 2018.

[26] M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In *CAV 2001*, volume 6806 of *LNCS*, pages 99–115, 2011.

[27] M.F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. What's decidable about weak memory models? In *ESOP 2012*, volume 7211 of *LNCS*, pages 26–46, 2012.

[28] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *POPL 2010*, pages 7–18, 2010.

[29] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. Context-bounded analysis of TSO systems. In *FPS 2014*, pages 21–38, 2014.

[30] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[31] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 2001*, pages 203–213, 2001.

[32] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber.

Mathematizing C++ concurrency. In *POPL 2011*, pages 55–66, 2011.

[33] Boris Beizer. *Software Testing Techniques (2Nd Ed.)*. 1990.

[34] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT 2007*, 9(5-6):505–525, 2007.

[35] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electr. Notes Theor. Comput. Sci.*, 66(2):160–177, 2002.

[36] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[37] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS 1999*, pages 193–207, 1999.

[38] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *ESOP 2013*, 2013.

[39] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Robustness against relaxed memory models. In *Software Engineering 2014*, pages 85–86, 2014.

[40] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In *ICALP 2011*, pages 428–440, 2011.

[41] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[42] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[43] Soham Chakraborty and Viktor Vafeiadis. Validating optimizations of concurrent C/C++ programs. In *CGO 2016*, pages 216–226, 2016.

[44] Soham Chakraborty and Viktor Vafeiadis. Formalizing the concurrency semantics of an LLVM fragment. In *CGO 2017*, pages 100–110, 2017.

[45] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *PACMPL 2017*, 2(POPL):31:1–31:30, 2017.

[46] M. Christakis, A. Gotovos, and K. Sagonas. Systematic testing for detecting concurrency errors in Erlang programs. In *ICST 2013*, pages 154–163, 2013.

[47] Edmund M. Clarke. Counterexample-guided abstraction refinement. In *TIME-ICTL 2003*, page 7, 2003.

[48] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[49] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State space reduction using partial order techniques. *STTT 1999*, 2(3):279–287, 1999.

[50] Edmund M. Clarke, Anubhav Gupta, and Ofer Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(7):1113–1123, 2004.

[51] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[52] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER 2011*, pages 1–30,

2011.

[53] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS 2004*, volume 2988 of *LNCS*, pages 168–176, 2004.

[54] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Proc. IBM workshop on Logics of Programs*, volume 131 of *LNCS*, 1982.

[55] Sylvain Conchon, David Declerck, and Fatiha Zaïdi. Compiling parameterized x86-TSO concurrent programs to cubicle- *W*. In *ICFEM 2017*, pages 88–104, 2017.

[56] Sylvain Conchon, David Declerck, and Fatiha Zaïdi. Parameterized model checking modulo explicit weak memory models. In *IMPEX and FM&MDD 2017*, pages 48–63, 2017.

[57] Sylvain Conchon, David Declerck, and Fatiha Zaïdi. Cubicle-*W* : Parameterized model checking on weak memory. In *IJCAR 2018*, pages 152–160, 2018.

[58] Mads Dam, Roberto Guanciale, and Hamed Nemati. Machine code verification of a tiny ARM hypervisor. In *TrustED 2013*, pages 3–12, 2013.

[59] Brian Demsky and Patrick Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *OOPSLA 2015*, pages 20–36, 2015.

[60] Egor Derevenetc. *Robustness against Relaxed Memory Models*. PhD thesis, University of Kaiserslautern, 2015.

[61] Egor Derevenetc and Roland Meyer. Robustness against Power is PSpace-complete. In *ICALP 2014*, volume 8573, pages 158–170, 2014.

[62] Egor Derevenetc, Roland Meyer, and Sebastian Schweizer. Locality and singularity for store-atomic memory models. In *NETYS 2017*, pages 133–148, 2017.

[63] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[64] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[65] Marko Doko and Viktor Vafeiadis. A program logic for C11 memory fences. In *VMCAI 2016*, pages 413–430, 2016.

[66] Yuelu Duan, Abdullah Muzahid, and Josep Torrellas. WeeFence: toward making fences free in TSO. In *ISCA 2013*, 2013.

[67] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *TCS 2001*, 256(1-2):63–92, 2001.

[68] Bernd Fischer, Omar Inverso, and Gennaro Parlato. CSeq: A concurrency pre-processor for sequential C verification tools. In *ASE 2013*, pages 710–713, 2013.

[69] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL 2005*, pages 110–121, 2005.

[70] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *POPL 2016*, pages 608–621, 2016.

[71] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579,

University of Cambridge, Computer Laboratory, 2004.

[72] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* PhD thesis, University of Liège, 1996.

[73] P. Godefroid, B. Hammer, and L. Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using VeriSoft. In *ISSTA 1998*, pages 124–133, 1998.

[74] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *CAV 1993*, 1993.

[75] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *POPL 1997*, pages 174–186, 1997.

[76] Patrice Godefroid. Software model checking: The VeriSoft approach. *FMSD 2005*, 26(2):77–101, 2005.

[77] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI 2005*, pages 213–223, 2005.

[78] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV 1997*, pages 72–83, 1997.

[79] Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. Provably secure memory isolation for linux on ARM. *Journal of Computer Security*, 24(6):793–837, 2016.

[80] Mengda He, Viktor Vafeiadis, Shengchao Qin, and João F. Ferreira. Reasoning about fences and relaxed atomics. In *PDP 2016*, pages 520–527, 2016.

[81] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition.* Morgan Kaufmann, 2012.

[82] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[83] Gerard Holzmann. *Spin model checker, the: primer and reference manual.* Addison-Wesley Professional, first edition, 2003.

[84] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[85] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI 2015*, pages 165–174, 2015.

[86] Shiyou Huang and Jeff Huang. Maximal causality reduction for TSO and PSO. In *OOPSLA 2016*, pages 447–461, 2016.

[87] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-CSeq: A context-bounded model checking tool for multi-threaded c-programs. In *ASE 2015*, pages 807–812, 2015.

[88] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *CAV 2014*, pages 585–602, 2014.

[89] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++.* International Organization for Standardization, Geneva, Switzerland, February 2012.

[90] Jonathan Jacky. Model-based testing with Spec#. In *ICFEM 2004*, pages 5–6, 2004.

[91] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte.

Model-Based Software Testing and Analysis with C#. Cambridge University Press, January 2008.

[92] Bart Jacobs, Joachim van den Berg, Marieke Huisman, and Martijn van Berkum. Reasoning about Java classes (preliminary report). In *OOPSLA 1998)*, pages 329–340, 1998.

[93] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about Release-Acquire consistency in Iris. In *ECOOP 2017*, pages 17:1–17:29, 2017.

[94] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL 2017*, pages 175–189, 2017.

[95] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A formal C memory model supporting integer-pointer casts. In *PLDI 2015*, pages 326–335, 2015.

[96] M. Kokologiannakis and K. Sagonas. Stateless model checking of the linux kernel's hierarchical read-copy-update (tree RCU). In *SPIN 2017*, pages 172–181, Santa Barbara, CA, USA, 2017.

[97] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *PACMPL 2018*, 2(POPL):17:1–17:32, 2018.

[98] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV 2009*, volume 5643 of *LNCS*, pages 477–492, 2009.

[99] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *POPL 2016*, pages 649–662, 2016.

[100] Ori Lahav and Viktor Vafeiadis. Explaining relaxed memory models with program transformations. In *FM 2016*, pages 479–495, 2016.

[101] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI 2017*, pages 618–632, 2017.

[102] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *FMSD 2009*, 35(1):73–97, 2009.

[103] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9):690–691, 1979.

[104] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[105] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[106] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[107] Steven M. LaValle. *Planning algorithms*. Cambridge University Press, 2006.

[108] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *OOPSLA 2009*, 2009.

[109] Nancy G. Leveson. *Safeware - system safety and computers: a guide to preventing accidents and losses caused by technology*. Addison-Wesley, 1995.

[110] Changhui Lin. *Imposing Minimal Memory Ordering on Multiprocessors*. PhD thesis, USA, 2013. AAI3600582.

[111] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN 2011*, volume 6823, pages 144–160, 2011.

[112] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In *CAV 2012*, volume 7358, pages 495–512, 2012.

[113] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL 2005*, pages 378–391, 2005.

[114] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models, October 2012.

[115] Daniel Marino, Abhayendra Singh, Todd D. Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A case for an SC-preserving compiler. In *PLDI 2011*, pages 199–210, 2011.

[116] Daniel Luke Marino. *Simplified Semantics and Debugging of Concurrent Programs via Targeted Race Detection*. PhD thesis, 2011.

[117] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, 1986.

[118] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[119] M. Michael, M. Vechev, and V. Saraswat. Idempotent work stealing. In *PPoPP 2009*, 2009.

[120] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[121] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI 2007*, pages 446–455, 2007.

[122] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI 2008*, pages 267–280, 2008.

[123] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI 2007*, pages 446–455, 2007.

[124] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy sequentialization for the safety verification of unbounded concurrent programs. In *ATVA 2016*, pages 174–191, 2016.

[125] Brian Norris and Brian Demsky. CDSChecker: checking concurrent data structures written with C/C++ atomics. In *OOPSLA 2013*, pages 131–150, 2013.

[126] Brian Norris and Brian Demsky. A practical approach for model checking C/C++11 code. *TOPLAS 2016*, 38(3):10:1–10:51, 2016.

[127] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOL 2009*, 2009.

[128] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP 2010*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010.

[129] Doron A. Peled. All from one, one for all, on model-checking using representatives. In *CAV 1993*, volume 697 of *LNCS*, pages 409–423, 1993.

[130] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Promising compilation to ARMv8 POP. In *ECOOP 2017*, pages 22:1–22:28, 2017.

[131] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL 2018*, 2(POPL):19:1–19:29, 2018.

[132] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS 2005*, pages 93–107, 2005.

[133] Azalea Raad, Ori Lahav, and Viktor Vafeiadis. On parallel snapshot isolation and Release/Acquire consistency. In *ESOP 2018*, pages 940–967, 2018.

[134] O. Saarikivi, K. Kähkönen, and K. Heljanko. Improving dynamic partial order reductions for concolic testing. In *ACSD 2012*, pages 132–141, 2012.

[135] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *PLDI 2012*, pages 311–322, 2012.

[136] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI 2011*, pages 175–186, 2011.

[137] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *HVC 2006*, pages 166–182, 2006. LNCS 4383.

[138] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *POPL 2011*, pages 43–54, 2011.

[139] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, 2013.

[140] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *CACM 2010*, 53, 2010.

[141] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *TOPLAS 1998*, 10(2):282–312, April 1988.

[142] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. A separation logic for a promising semantics. In *ESOP 2018*, pages 357–384, 2018.

[143] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Verifying concurrent programs by memory unwinding. In *TACAS 2015*, pages 551–565, 2015.

[144] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy sequentialization for TSO and PSO via shared memory abstractions. In *FMCAD 2016*, pages 193–200, 2016.

[145] Ermenegildo Tomasco, Truc Lam Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Using shared memory abstractions to design eager sequentializations for weak memory models. In *SEFM 2017*, pages 185–202, 2017.

[146] Viktor Vafeiadis. Formal reasoning about the C11 weak memory model. In *CPP 2015*, pages 1–2, 2015.

[147] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL 2015*, pages 209–220, 2015.

[148] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: a program logic for C11 concurrency. In *OOPSLA 2013*, pages 867–884, 2013.
[149] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. In *SAS 2011*, pages 146–162, 2011.
[150] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, volume 483 of *LNCS*, pages 491–515, 1990.
[151] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *ISSTA 2004*, pages 97–107, 2004.
[152] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In *ECOOP 2008*, pages 27–51, 2008.
[153] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.
[154] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *PLDI 2015*, pages 250–259, 2015.